Semantic Encapsulation using Linking Types

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

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

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

Abstract

Interoperability pervades nearly all mainstream language implementations, as most systems leverage subcomponents written in different languages. And yet, such linking can expose a language to foreign behaviors that are internally inexpressible, which poses a serious threat to safety invariants and programmer reasoning. To preserve such invariants, a language may try to add features to limit the reliance on external libraries, but endless extensions can obscure the core abstractions the language was designed to provide.

In this paper, we outline an approach that encapsulates foreign code in a sound way—i.e., without disturbing the invariants promised by types of the core language. First, one introduces novel *linking types* that characterize the behaviors of foreign libraries that are inexpressible in the core language. To reason about the soundness of linking, one constructs a *realizability model* that captures the meaning of both core types and linking types as sets of target-language terms. Using this model, one can formally prove when foreign behavior is *encapsulated*; that is, unobservable to core code. We show one way to discharge such proofs automatically by augmenting the compiler to insert verified *encapsulation wrappers* around components that use foreign libraries.

Inspired by existing approaches to FFIs, we develop a pair of case studies that extend a pure, functional language: one extension for state, and another for exceptions. The first allows us to implement mutable references via a library, whereas the second allows us to implement `try` and `catch` as library functions. Both extensions and the overall system are proven sound using logical relations that use realizability techniques.

CCS Concepts: • **Software and its engineering → General programming languages**.

Keywords: language interoperability, linking, type soundness, semantics, logical relations

TyDe '23, September 4, 2023, Seattle, WA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0299-0/23/09.
https://doi.org/10.1145/3609027.3609405

ACM Reference Format:
Daniel Patterson, Andrew Wagner, and Amal Ahmed. 2023. Semantic Encapsulation using Linking Types. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '23), September 4, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3609027.3609405

1 Introduction

Languages cannot exist in isolation. Foreign function interfaces (FFIs) are a critical feature of nearly all mainstream languages as they underpin core libraries and runtimes, and facilitate the construction of large systems with components written in different languages. For example, a recent study of Rust [5] found that the most common justification for `unsafe` code—which is itself more common than one might hope—is interoperability. For the sake of illustration, consider Figure 1, which defines a simple binding to C's `signal` function in Rust and in Haskell. In C, `signal` has the signature `void (*signal(int signum, void (*handler)(int)))(int)`: it takes a signal identifier and a function pointer to a handler, installs the handler to respond to that signal from now on, and returns a function pointer to the previous handler for that signal.

Looking beyond surface-level differences, there are notable similarities between the FFIs provided by these languages. Both (Rust L2–6, Haskell L2–3) import the foreign function `signal` from `libc`, specifying where to find the function and which ABI to use. In both cases, the programmer is responsible for ascribing a type to the foreign function, and the compiler will *not* check that it is accurate. We also (Rust L10-11, Haskell L9-12) demonstrate the mechanism for a core function to be passed dynamically to foreign code.

The most interesting lines are in the middle (Rust L7–9, Haskell L4-8). As per the FFI guidelines for both languages [21, 27, 47], we hide the "raw" foreign function from clients and instead export a safe *wrapper* around it. Wrappers have two primary responsibilities: (i) to *marshal data* between idiomatic core types and foreign types (e.g., `extern fn` and `usize`, or `->` and `FunPtr`); and (ii) to *encapsulate foreign behaviors* in a way that preserves the standard expectations of the core language. The former, while tricky, is generally well-understood, but the latter is murky at best. Both languages provide a mechanism to indicate that something "inherently unsafe" [21] is occuring (`unsafe` and `IO`), and the compiler will even complain when these mechanisms are omitted. But

```
type Handler = extern "C" fn(i32) -> ();
#[link(name = "c")]
extern "C" {
  #[link_name = "signal"]
  fn c_signal(s: i32, h: usize) -> usize;
}
fn signal(s: i32, h: Handler) -> usize {
  unsafe { c_signal(s, h as usize) }
}
extern "C" fn print_sig(s: i32) {
  println!("{0}", s); }
```

(a) Rust

```
type Handler = CInt -> IO ()
foreign import ccall "signal.h␣signal" cSignal
    :: CInt -> FunPtr Handler -> IO CInt
signal :: Int -> Handler -> IO Int
signal s h = do
  h <- mkHandler h;
  r <- cSignal (fromIntegral s) h
  return $ fromIntegral r
foreign import ccall "wrapper" mkHandler
    :: Handler -> IO (FunPtr Handler)
printSig :: Handler
printSig = putStrLn . show
```

(b) Haskell

Figure 1. Using `signal.h` from `libc` via the FFI.

short of some rules of thumb and best practices, the programmer has little guidance on how to actually *reason about* unsafety.

In fact, this isn't only a problem for programmers—it's a problem for *language designers* as well. Indeed, the process of linking with foreign code often invalidates one of the key safety theorems of many languages: type soundness. Typically, if a language is proven sound at all, the proof will almost certainly exclude the FFI. Unfortunately, this means that the soundness theorem is only a crude approximation of reality, as nearly all programs have FFI calls somewhere in their stack. But it is not obvious exactly how to incorporate the FFI into a soundness theorem, since the prevailing technique for proving type soundness is a syntactic progress and preservation proof. Such a proof shows that types are preserved by reduction, and if the language in question is not self-contained, then the proof must account for the behavior of the foreign code, too.

This is precisely what Matthews and Findler [40] set out to address with *multi-language semantics*, which is defined to account for the behaviors of two languages, say core language L and foreign language F. Interoperation between these languages is mediated by a *boundary*, ${}^{\tau_{\mathsf{L}}}\mathcal{LF}^{\tau_{\mathsf{F}}} e_{\mathsf{F}}$, which enables foreign code $e_{\mathsf{F}} : \tau_{\mathsf{F}}$ to be used in an L context that expects code of type τ_{L}. Boundaries play the role of, e.g, `unsafe` above, but because the semantics incorporates both constituent languages, it is *not* unsafe, but rather well-specified. Unfortunately, it is impractical to design a multi-language semantics for every pair of interacting languages.

Even worse, it is unrealistic to expect an L programmer to also know all the foreign languages $\mathsf{F}_1, \ldots, \mathsf{F}_n$ in which their libraries were written. Indeed, the most serious limitation of a multi-language semantics is that L programmers cannot benefit from the extra expressive power of F unless they write embedded F programs themselves. This multi-language approach would require, for example, Rust programmers to write embedded C code in order to use C libraries.

Instead, we want to adopt the style of existing FFIs, as shown in Figure 1, which allows programmers to leverage the power of the foreign language while continuing to program only in their language. Over the course of this paper, we will show how to develop a *sound* FFI for a language that does not yet have a way to link with foreign code. We will demonstrate the approach using a pure[1] functional language, FunLang. As motivation, suppose that a FunLang programmer wants use a foreign library for exceptions. That is, they wish to import procedures `throw` and `catch` and use them in a FunLang program, e:

$$\mathtt{import(throw : ?, catch : ?)}\ \mathtt{e}$$

The problem is that there are no types in our pure language that can faithfully account for the effectful behavior of throw and catch. FunLang types τ permit and prohibit certain behavior, even after compilation to our stateful target StackLang which has control effects. Specifically, they require *extensional purity*: that the only behaviors allowed are those that do not have any observable effects (other than divergence). Thus, even when we link with an exceptions library, we want to ensure that any term with a FunLang type τ does not produce any uncaught exceptions. A traditional multi-language semantics would forbid us from using throw and catch directly in FunLang; we would instead be forced to write a subprogram in a foreign, impure language and then export the resulting value.

To reconcile the problems with using foreign, inexpressible behavior directly within our own language, we build upon a position paper by two of the authors ([48]), which proposed *linking types*. The core idea is that language designers augment their language with types so that programmers can annotate exactly where foreign behaviors are introduced into their programs. This is not unlike the FFIs in Figure 1, where extra types (e.g., `FunPtr` and `CInt`) or qualifiers (e.g., `extern`) are added specifically to address foreign code. That paper's vision was to build fully abstract compilers that support linking with target programs that are inexpressible in the source.

In this paper, we apply the linking types idea to a more grounded but useful goal: proving type soundness in the presence of foreign functions. To do so, we require a way to connect source-level linking types to the target-level library code that they describe. We do this using *realizability models*,

[1]Throughout this paper, when we say "pure language" we mean a language with no effects other than divergence.

which interpret source types as sets of target terms. Our target in this paper, StackLang, has mutable state and control effects. In recent work, Patterson et al. [49] employ realizability models to verify the soundness of multi-language semantics. This paper takes that idea a step futher by designing realizability models for both core source types as well as linking types, and by implementing *encapsulation boundaries* that wrap components using foreign code (much like `signal` wraps `c_signal` in Figure 1).

Unencapsulatable Behaviors. Our approach relies upon being able to prove that uses of foreign code actually can be encapuslated. There are certain behaviors that cannot be encapsulated, or for which encapsulation would defeat their very purpose. For example, if we wish to add file I/O to our pure language, we must decide either to revise our notion of purity (i.e., the meaning of types) or revise the mechanics of I/O (e.g., transactions or monads). This is a question for language designers, and not what our paper addresses.

Rather, our contribution is an approach to *sound linking*; that is, linking that *does* preserve the meaning of core types. Prior work by Patterson et al. [49] showed how to achieve this when the foreign code could itself be given a core type. This work shows how to achieve this when the foreign code cannot be given a core type, but the *linking code*—that is, the core-language code that directly calls foreign functions—can be encapsulated in a way that earns it a true core type. The approach accounts for the common case in which uses of a foreign library are carefully isolated and unobservable by the rest of the program, as in Figure 1, and gives an accounting for how such uses can be *proved* sound.

Linking Types and Encapsulation Boundaries. At a high level, our approach *looks* a lot like the FFIs in Figure 1, but we patch the "escape hatches" (e.g., `unsafe`) that threaten the validity of FunLang's type soundness theorem. Whereas uses of an FFI are often accompanied by scary compiler warnings and comments like, "*trust me, it works,*" we want the *language* to formally account for the behavior of linked components *even if* they are written in a more expressive target. To do so, we extend FunLang with *linking types* τ that precisely characterize the inexpressible foreign behaviors that we want to link with. In this way, foreign behaviors become *logically* expressible in FunLang, even if they are not actually implementable in FunLang. Importantly, each linking type can be *lowered* $\downarrow\tau$ back to an FunLang-language type τ. Note that we can design multiple such extensions for a single language: e.g., for FunLang we can design extensions that separately add mutable state, exceptions, or first-class control, or an extension that adds them in some combination.

Next, to facilitate the interaction between core and extended programs, we include a *boundary* term, $\{\mathtt{e}\}_{\downarrow\tau}$, that delineates the linking code from the rest of the core program. Here, e is a program that may, e.g., throw and catch exceptions in the course of producing a τ. The boundary means its result may be used at core type $\downarrow\tau$ in the rest of the program. Like `unsafe`, we stress that adding this boundary is a *one-time* change one must make to FunLang; the same boundary term can be reused by any linking types extension. The boundary acts as a syntactic cue for the typechecker to switch to a linking types extension, much like, e.g., Rust's compiler is more liberal inside of `unsafe`.

Returning to soundness, a boundary is not only a cue for the typechecker, but may be for the *compiler* as well. Indeed, boundaries can have runtime significance. In particular, we need to ensure that the code inside can be used safely at $\downarrow\tau$. One way to do that is for every extended type τ, define a target-level *encapsulation wrapper*, $\wr\tau\int$, which dynamically enforces that its argument *behave like a* $\downarrow\tau$. In place of dynamic checks, one could instead construct specialized proofs on a case-by-case basis; our approach certainly admits this mode of use (c.f., [32]). However, we find that many guidelines for FFI use [21, 47] already encourage a very defensive style of programming that sports an abundance of checks, so in this presentation we opt to automate where possible. Proving soundness relies on these wrappers masking all foreign behaviors, which requires that we can characterize when a target term *behaves* like a core type τ. We do this by building semantic models indexed by source types τ and τ but inhabited by *target* terms. These so-called *realizability* models, described shortly, are a key ingredient to the approach.

Contributions. This paper demonstrates how to encapsulate foreign code, imported via an FFI, in a sound way—i.e., without disturbing the invariants promised by types of the core language. First, we introduce *linking types* that characterize the behaviors of foreign libraries that are inexpressible in the core language (Sections 4 and 5). We also develop *encapsulation wrappers* that are placed around all components using foreign libraries, which are demarcated by syntactic boundaries. Next, to show linking is sound, we build a *realizability model* that captures the meaning of both core types and linking types as sets of target-language terms. Using this model, we formally prove when foreign behavior is *encapsulated*; that is, unobservable to core code.

To demonstrate the viability of this approach as a way of importing and encapsulating foreign behavior, we present two case studies, each of which extends a pure functional language, FunLang. In the first case study, we develop a linking types extension for state, giving FunLang access to mutable references via a library. Then, in the second case study, we develop a linking types extension that also adds exceptions, giving FunLang access to `try` and `catch` as library functions.

The core language and both linking-types extensions are proven sound via logical relations that use realizability techniques. Crucially, the realizability model for the core language, FunLang, can be defined before extensions are even considered, which means that it does not require the semantic complexity (e.g., Kripke worlds, biorthogonality [3, 17])

$$\begin{array}{llll} \text{Base Types } \widehat{\tau} := \mathtt{unit} \mid \mathtt{bool} \mid \mathtt{int} & & \text{Imports } \mathtt{I} := \mathtt{f} : \tau \mid \mathtt{f} : \tau, \mathtt{I} \\ \text{Types } \tau := \widehat{\tau} \mid \alpha \mid \tau \times \tau \mid \tau + \tau & & \text{Programs } \mathtt{P} := \mathtt{e} \mid \mathtt{import(I)}\ \mathtt{e} \\ \quad \mid \mu\alpha.\tau \mid (\tau, \ldots, \tau) \to \tau & & \end{array}$$

$$\boxed{\vdash \mathtt{P} : \tau} \qquad \frac{\cdot; \cdot \vdash \mathtt{e} : \tau}{\vdash \mathtt{e} : \tau} \qquad \frac{\mathtt{I}; \cdot \vdash \mathtt{e} : \tau}{\vdash \mathtt{import(I)}\ \mathtt{e} : \tau} \qquad \ldots$$

$$\boxed{\mathtt{I}; \Gamma \vdash \mathtt{e} : \tau} \qquad \frac{\mathtt{x} : \tau \in \mathtt{I}; \Gamma}{\mathtt{I}; \Gamma \vdash \mathtt{x} : \tau} \qquad \ldots$$

Figure 2. Syntax & static semantics for FunLang.

of the extended models for state and exceptions. As we show, the encapsulation wrappers account for this gap in complexity and provide sufficient semantic encapsulation to be able to run impure code under a boundary while preserving extensional purity.

We structure our paper into two halves. After introducing our languages (§2) and a statement of type soundness (§3), we first show how one *implements* linking types extensions for state (§4) and state+exceptions (§5) and after, sketch how one *verifies* their soundness (§6). The technical appendix [51] includes complete language semantics, definitions, and proofs, some of which are elided in this paper.

2 Setting the Stage

2.1 A Functional Language

Our core source language, FunLang is a standard pure, eager, functional language with imports. It sports both iso-recursive types (with fold/unfold) and recursive functions, as well as sums, products, and simple base types (unit, int, bool). We present an excerpt of the syntax (typeset in blue typewriter font) and static semantics in Fig. 2. Despite the definition-like syntax, functions are still anonymous expressions; the function's name is only bound in its body, for recursive calls. For simplicity, the language does not have polymorphism, but the feature is compatible with the models and techniques that we are using (c.f., Patterson et al. [49]).

As a running example program, we'll use a fan favorite:

```
fun fib(n : int){  if n < 1 { 0 } {
    if n = 1 { 1 } { fib(n + −1) + fib(n + −2) } }}
```

While one could certainly define a standard operational semantics for FunLang, we will not do so here. Instead, like many real languages, the *observable* semantics is defined by a particular *implementation*; in this case, via compilation to a stack-based target language, StackLang.

2.2 A Stack Language

Our target, StackLang, is untyped and stack-based, and is derived from Kleffner [33], which in turn derives features from Levy [38]. It is significantly more expressive than our source language, as target languages often are. On the other hand, it is not especially low-level; it permits aggregate values and suspended computations (thunks) on the stack. An excerpt of the syntax and semantics is in Fig. 3, typeset in black typewriter font. The small-step operational semantics is defined as a relation on program configurations ⟨H ⨾ S ⨾ P⟩, which are triples of a heap, stack, and program.

Values are placed on the stack with push. The binary operators add, less?, and equal? operate on the two integers at the top of the stack. The if0 instruction conditions on the integer at the top of the stack and executes the first branch if it is zero, and the second branch otherwise. Despite its syntax, lam x.P is not a value, but a computation (as in Levy's Call-By-Push-Value [38]) that substitutes the top value on the stack for x inside P. On the other hand, the value thunk P is a suspended computation, so thunk lam x.P is analogous to a traditional lambda value. The call instruction takes the thunk P at the top of the stack and forces its computation, placing P at the head of the program. As expected, fix performs a fixpointing operation: it takes the thunk at the top of the stack and re-suspends it for recursive calls, and then forces one copy of its computation. Both idx and len operate on the array value at the top of the stack. Instructions alloc, read, write, and free perform standard heap operations, where any StackLang value can be stored in the heap. Together, shift k P and reset provide delimited control [16, 18]: shift captures the continuation until the next reset and substitutes it for k in P. The getlocs instruction provides reflective access to the heap: it takes the thunk lam and the value v at the top of the stack and maps the computation over all locations used in v (overline indicating a sequence). While this particular primitive is somewhat specialized to our case study, it can easily be implemented in most low-level languages, and some targets even provide similar abstractions for implementing GCs (e.g., [31, 55]). Unsurprisingly, noop does nothing. Finally, fail c terminates execution with the given error code. Every instruction with a type invariant on the stack uses fail Type when that invariant is not met, producing a *dynamic* type error (we ellide many here; see [51]). Other errors (Mem, Idx, Ctrl) are for unrecoverable problems that may be acceptable results according to a soundness theorem.

2.3 A Compiler for FunLang

Figure 4 presents a compiler from FunLang to StackLang, which also establishes the working operational semantics of FunLang. A base value is compiled to push v, where v is a target-level encoding of the value. Note that we use 0 both for true (to match if0) and unit. In a typical functional language, like FunLang, the values constitute a subset of the expressions, and evaluation stops at values. However, in StackLang, evaluation only stops on the empty program, and we consider the value on the top of the stack to be the result. So, whereas v is the simplest FunLang program, push v is, by analogy, the simplest StackLang program.

$$
\begin{array}{rcl}
\text{Stack } S := v, \ldots, v \mid \text{Fail } c & & \text{Error Code } c \;:=\; \textsc{Type} \mid \textsc{Idx} \mid \textsc{Mem} \mid \textsc{Ctrl} \\
\text{Progr. } P := \cdot \mid i; P & & \text{Value } v \;:=\; n \mid \text{thunk } P \mid \ell \mid [v, \ldots]
\end{array}
$$

$$
\begin{aligned}
\text{Instr. } i := {} & \text{push } v \mid \text{add} \mid \text{less?} \mid \text{equal?} \mid \text{if0 } P\ P \mid \text{lam } x.P \mid \text{call} \mid \text{fix} \\
& \mid \text{idx} \mid \text{len} \mid \text{alloc} \mid \text{read} \mid \text{write} \mid \text{free} \mid \text{shift } k\ P \mid \text{reset} \\
& \mid \text{getlocs} \mid \text{noop} \mid \text{fail } c
\end{aligned}
$$

$$
\begin{array}{rcl}
\langle H \fatsemi S \fatsemi \text{push } v; P\rangle & \longrightarrow & \langle H \fatsemi S, v \fatsemi P\rangle \quad (S \neq \text{Fail } c) \\
\langle H \fatsemi \text{Fail } c \fatsemi \text{push } v; P\rangle & \longrightarrow & \langle H \fatsemi \text{Fail } c \fatsemi \text{fail } \textsc{Type}\rangle \\
\langle H \fatsemi S, v \fatsemi \text{lam } x.P_1; P_2\rangle & \longrightarrow & \langle H \fatsemi S \fatsemi [x \mapsto v]P_1; P_2\rangle \\
\langle H \fatsemi S, \text{thunk } P_1 \fatsemi \text{call}; P_2\rangle & \longrightarrow & \langle H \fatsemi S \fatsemi P_1; P_2\rangle \\
\langle H \fatsemi S, \text{tnk } P_1 \fatsemi \text{fix}; P_2\rangle & \longrightarrow & \langle H \fatsemi S, \text{tnk}(\text{psh}(\text{tnk } P_1), \text{fix}) \fatsemi P_1; P_2\rangle \\
\langle H \fatsemi S, [v_i]_{i<n}, m \fatsemi \text{idx}; P\rangle & \longrightarrow & \langle H \fatsemi S, v_m \fatsemi P\rangle \quad (m \in [0, n]) \\
\langle H \fatsemi S, [v_i]_{i<n}, m \fatsemi \text{idx}; P\rangle & \longrightarrow & \langle H \fatsemi S \fatsemi \text{fail } \textsc{Idx}\rangle \quad (m \notin [0, n]) \\
\langle H \fatsemi S, v \fatsemi \text{alloc}; P\rangle & \longrightarrow & \langle H \uplus \{\ell \mapsto v\} \fatsemi S, \ell \fatsemi P\rangle \\
\langle H \uplus \{\ell \mapsto v\} \fatsemi S, \ell \fatsemi \text{read}; P\rangle & \longrightarrow & \langle H \uplus \{\ell \mapsto v\} \fatsemi S, v \fatsemi P\rangle \\
\langle H \uplus \{\ell \mapsto \_\,\} \fatsemi S, \ell, v \fatsemi \text{write}; P\rangle & \longrightarrow & \langle H \uplus \{\ell \mapsto v\} \fatsemi S \fatsemi P\rangle \\
\langle H \uplus \{\ell \mapsto \_\,\} \fatsemi S, \ell \fatsemi \text{free}; P\rangle & \longrightarrow & \langle H \fatsemi S \fatsemi P\rangle \\
\langle H \fatsemi S, \ell \fatsemi \text{free}; P\rangle & \longrightarrow & \langle H \fatsemi S \fatsemi \text{fail } \textsc{Mem}\rangle \quad (\ell \notin \text{dom}(H)) \\
\langle H; S; \text{shift } k\ P_1; P_2; & \longrightarrow & \langle H \fatsemi S \fatsemi [k \mapsto \text{thunk } P_2; \ldots]P_1; P_3\rangle \\
\ldots; \text{reset}; P_3\rangle & & (\text{reset} \notin P_2; \ldots) \\
\langle H \fatsemi S \fatsemi \text{shift } k\ P_1; P_2\rangle & \longrightarrow & \langle H \fatsemi S \fatsemi \text{fail } \textsc{Ctrl}\rangle \quad (\text{reset} \notin P_2) \\
\langle H \fatsemi S \fatsemi \text{reset}; P\rangle & \longrightarrow & \langle H \fatsemi S \fatsemi P\rangle \\
\langle H; S, \text{thunk lam } l.P_1, v; & \longrightarrow & \langle H \fatsemi S, \overline{\ell} \fatsemi \overline{\text{lam } l.P_1}; P_2\rangle \\
\text{getlocs}; P_2\rangle & & (\overline{\ell} = \text{flocs}(v)) \\
\langle H \fatsemi S \fatsemi \text{fail } c; P\rangle & \longrightarrow & \langle H \fatsemi \text{Fail } c \fatsemi \cdot\rangle
\end{array}
$$

Figure 3. Syntax and operational semantics for StackLang

For `if`, we compile the discriminant `e` (denoted e^+), which, according to `e`'s type, should be a program fragment that terminates with a (compiled) boolean at the top of the stack. Thus, the (dynamic) type invariant for if0 should be satisfied, and it can proceed with (the compilation of) the appropriate branch. The compilation of the binary operators is similar.

`inl` and `inr` are slightly different because the result value needs to be tagged. We use arrays to store the tag, 0 or 1, along with the payload. To move the payload value off of the stack and into an array, we use lam, which, as described earlier, is an *instruction* (not a value) that performs substitution with the value at the top of the stack. Pairs and projections are compiled similarly.

Compiling `match` is conceptually like compiling `if`, but its definition is more involved because one must destruct tagged values. e^+ should produce a tagged value at the top of the stack, so we copy it with the macro DUP (defined at the bottom of the figure), project out the payload (at index 1), SWAP the top two elements of the stack, and finally project out the tag. Now, we are ready to condition on the tag and, in the branches, substitute the payload.

In theory, one could entirely erase any remnant of the recursive operators `fold` and `unfold`. Indeed, we do just that for `fold`. However, for reasons that will become clear in §6, we introduce a noop in the compilation of `unfold`. In short, `unfold` produces an expression at a potentially larger type, which threatens the well-foundedness of our semantic model, as it is defined inductively over types. To reconcile this, we employ a standard trick and stratify the model, which requires that $\texttt{unfold}^+$ take this extra step.

$$\boxed{e \rightsquigarrow e^+}$$

$$
\begin{array}{ll}
() & \rightsquigarrow \text{push } 0 \\
\texttt{true}/\texttt{false} & \rightsquigarrow \text{push } 0/1 \\
\texttt{if } e\ \{e_1\}\ \{e_2\} & \rightsquigarrow e^+; \text{if0 } ({e_1}^+)\ ({e_2}^+) \\
n & \rightsquigarrow \text{push } n \\
e_1 < / = / + e_2 & \rightsquigarrow {e_1}^+; {e_2}^+; \text{less?}/\text{equal?}/\text{add} \\
x & \rightsquigarrow \text{push } x \\
\texttt{inl}/\texttt{inr } e & \rightsquigarrow e^+; \text{lam } x.(\text{push } [0/1, x]) \\
\texttt{match } e\ x\{e_1\}\ y\{e_2\} & \rightsquigarrow e^+; \text{DUP}; \text{push } 1; \text{idx}; \text{SWAP}; \text{push } 0; \text{idx}; \\
 & \quad \text{if0 } (\text{lam } x.{e_1}^+)\ (\text{lam } y.{e_2}^+) \\
\texttt{fold } e & \rightsquigarrow e^+ \\
\texttt{unfold } e & \rightsquigarrow e^+; \text{noop} \\
(e_1, e_2) & \rightsquigarrow {e_1}^+; {e_2}^+; \text{lam } x_2.\text{lam } x_1.(\text{push } [x_1, x_2]) \\
\texttt{fst}/\texttt{snd } e & \rightsquigarrow e^+; \text{push } 0/1; \text{idx} \\
\texttt{fun } f(x_1 : \tau_1, \ldots, x_n : \tau_2)\{e\} & \rightsquigarrow \text{push } (\text{thunk push } (\text{thunk lam } f. \\
 & \quad \text{lam } x_n. \ldots\ \text{lam } x_1.e^+), \text{fix}) \\
e(e_1, \ldots, e_n) & \rightsquigarrow e^+; {e_1}^+; \text{SWAP}; \ldots; {e_n}^+; \text{SWAP}; \text{call}
\end{array}
$$

$$\text{SWAP} \triangleq \text{lam } x.\text{lam } y.(\text{push } x;\ \text{push } y) \qquad \text{DROP} \triangleq \text{lam } x.()$$
$$\text{DUP} \triangleq \text{lam } x.(\text{push } x;\ \text{push } x)$$

Figure 4. Compiler from FunLang to StackLang

All that remains are `fun`s and application. For functions, fix does most of the heavy lifting. A compiled `fun` is a thunk that first pushes a thunk corresponding to the body (taking itself as the first argument, f), and then invokes fix. The result of fix will be to perform the fixpoint, passing itself as that first argument. The arguments are in reverse order so that effects (only divergence, for now) are observed left-to-right. Application is conceptually straightforward; the only subtelty is that the compiled function (e^+) needs to be at the top of the stack in order to be called, but it needs to run *first* for a left-to-right evaluation order. Since StackLang does not have a built-in for indexing into the stack, we shuffle the function to the front as we evaluate the arguments.

3 What Is Type Soundness?

So far, we have used the term *type soundness* somewhat loosely. In this paper, we aim to prove theorems with the following shape:

Definition 3.1 (Type Soundness, roughly)**.** For any well-typed term of type τ, no matter how many steps it takes:

1. it can take another step; or
2. it is an acceptable error (e.g., divide-by-zero); or
3. it is a value v of type τ.

Although "type soundness" is a ubiquitous idea, its concrete definition varies widely by language or by paper. Some definitions insist on termination and strengthen clause 1, or include additional components like a heap. Naturally, different definitions specialize clause 2 to an appropriate class of errors. Meanwhile, some definitions weaken the type requirement on clause 3, such that termination at any value is sufficient. Indeed, there is no single definition of type soundness and we stress the importance of identifying exactly what is suitable for one's language.

In this paper, we prove *semantic* type soundness using *logical relations*. The particular variety of logical relations we use are called *realizability models* because they specify which sets of target terms *realize* (or behave like) which source types. While our formal proofs (included in [51]) use more sophisticated models, here we provide simplified excerpts as a glimpse of our approach.

$$
\begin{array}{ll}
\mathcal{V}[\![\mathtt{unit}]\!] & = \{0\} \\
\mathcal{V}[\![\mathtt{int}]\!] & = \{\mathsf{n}\} \\
\mathcal{V}[\![\tau_1 \times \tau_2]\!] & = \{[\mathsf{v}_1, \mathsf{v}_2] \mid \mathsf{v}_1 \in \mathcal{V}[\![\tau_1]\!] \wedge \mathsf{v}_2 \in \mathcal{V}[\![\tau_2]\!]\} \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!] & = \{\mathsf{thunk\ lam\ x.P} \mid \forall \mathsf{v} \in \mathcal{V}[\![\tau_1]\!].\ [\mathsf{x} \mapsto \mathsf{v}]\mathsf{P} \in \mathcal{E}[\![\tau_2]\!]\} \\
 & \\
\mathcal{E}[\![\tau]\!] & = \{\mathsf{P} \mid \forall \mathsf{H}\,\mathsf{H}'\,\mathsf{S}\,\mathsf{S}'.\ (\langle \mathsf{H} \mathbin{⨟} \mathsf{S} \mathbin{⨟} \mathsf{P}\rangle \xrightarrow{*} \langle \mathsf{H}' \mathbin{⨟} \mathsf{S}' \mathbin{⨟} \cdot\rangle)^{\textcircled{1}} \\
 & \Rightarrow (\mathsf{S}' = \mathsf{Fail\ c})^{\textcircled{2}} \ \vee\ \exists \mathsf{v}.\ (\mathsf{S}' = \mathsf{S}, \mathsf{v}\ \wedge\ \mathsf{v} \in \mathcal{V}[\![\tau]\!])^{\textcircled{3}}\}
\end{array}
$$

The *value relation*, $\mathcal{V}[\![\tau]\!]$, is indexed by FunLang types but inhabited by StackLang values. In particular, $\mathcal{V}[\![\tau]\!]$ contains all those StackLang values v that *behave like* a τ. Consistent with our compiler (§2.3), $\mathcal{V}[\![\mathtt{unit}]\!]$ contains only 0, while $\mathcal{V}[\![\mathtt{int}]\!]$ contains all target integers n. The most interesting case is $\mathcal{V}[\![\tau_1 \to \tau_2]\!]$, which contains all programs of the form thunk lam x.P that map well-typed inputs to well-typed outputs. In particular, for any input $\mathsf{v} \in \mathcal{V}[\![\tau_1]\!]$, $[\mathsf{x} \mapsto \mathsf{v}]\mathsf{P}$ should be a *computation* in $\mathcal{E}[\![\tau_2]\!]$.

The *expression relation*, $\mathcal{E}[\![\tau]\!]$, is indexed by FunLang types but is inhabited by StackLang *computations*. Notice that $\mathcal{E}[\![\tau]\!]$ is defined to precisely match our definition of type soundness. For clarity, we annotate each component of the predicate with its relevant clause from Def. 3.1. Of critical importance is the way that $\mathcal{E}[\![\tau]\!]$ treats the heap, H. Since FunLang is pure, it should not impose any conditions on the heap whatsoever. Indeed, a P only behaves like a τ if it does so *regardless* of the heap it is run with. At the same time, we only insist on *extensional* purity: ephemeral uses of state during the course of a computation are permitted, so long as the final value is independent from the heap (which is true of all $\mathcal{V}[\![\tau]\!]$ values, by definition).

Since the model captures our notion of soundness by design, proving soundness for our implementation amounts to showing that the compilation $\mathtt{e}^+$ of any well-typed term $\vdash \mathtt{e} : \tau$ satisfies the model: $\mathtt{e}^+ \in \mathcal{E}[\![\tau]\!]$. What we will explore next is how to preserve soundness in the presence of impure, foreign behavior introduced by code that calls external libraries. The value in using realizability models as a reasoning tool is that they suggest a clear strategy: we must find a way to *encapsulate* such code so that it satisfies the model.

4 Linking with State

With a source, target, and compiler in hand, we are now ready to tackle the central problem of this paper: how to safely encapsulate inexpressible behavior.

The fib program from §2.1 is a classic example of unnecessary exponential computation. A standard trick taught in most undergraduate curricula is to use memoization, which traditionally requires mutable state. As a quick reminder, the strategy is to store intermediate results in a table so that later computations can reuse them without recomputation. In our example (see Fig. 8), the table maps inputs n to their outputs fastfib(n), so that each fastfib(n) is only computed once. What we want is to link with a mutable reference library providing alloc, read, and write functions, but because FunLang was deliberately designed without state in mind, any FunLang types we assign to them would necessarily be imprecise! Thus, our type system—and, crucially, our soundness proof—has no way to accurately account for them.

$$
\begin{array}{lll}
\text{Base Types } \widehat{\tau} & := & \mathtt{unit} \mid \mathtt{bool} \mid \mathtt{int} \\
\text{Core Type } \tau & := & \alpha \mid \widehat{\tau} \mid \tau \times \tau \mid \tau + \tau \mid \mu\alpha.\tau \mid (\tau, \ldots, \tau) \to \tau \\
\text{Ext. Type } \tau & := & \widehat{\tau} \mid \tau \times \tau \mid \tau + \tau \mid \mu\alpha.\tau \mid (\tau, \ldots, \tau) \overset{\LEFTcircle}{\to} \tau \mid \mathbf{ref}\ \tau
\end{array}
$$

$$
\frac{\mathtt{x} : \tau \in \Gamma}{\Gamma \vdash_{\mathsf{S}} \mathtt{x} : \tau}
\qquad
\frac{\Gamma, \mathtt{f} : (\overline{\tau}) \overset{\bullet}{\to} \tau', \overline{\mathtt{x} : \tau} \vdash_{\mathsf{S}} \mathtt{e} : \tau'}{\Gamma \vdash_{\mathsf{S}} \mathtt{fun\ f}(\overline{\mathtt{x} : \tau})\{\mathtt{e}\} : (\overline{\tau}) \overset{\bullet}{\to} \tau'}
$$

$$
\frac{\Gamma \vdash_{\mathsf{S}} \mathtt{e} : (\tau_1, \ldots, \tau_n) \overset{\LEFTcircle}{\to} \tau' \qquad \overline{\Gamma \vdash_{\mathsf{S}} \mathtt{e}_i : \tau_i}}{\Gamma \vdash_{\mathsf{S}} \mathtt{e}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \tau'}
\qquad
\frac{\mathrm{I}^{\mathrm{E}} \uplus \uparrow\Gamma \vdash_{\mathrm{E}} \mathtt{e} : \tau}{\mathrm{I}; \Gamma \vdash \{\mathtt{e}\}^{\mathrm{E}}_{\downarrow\tau} : \downarrow\tau}
$$

Figure 5. Linking types for state and a generic boundary

A Principled Approach. One might be tempted to *approximate* foreign behavior with existing types; e.g., $\mathtt{ref}\ \tau \sim \mathtt{int}$, $\mathtt{read} : (\mathtt{int}) \to \tau, \mathtt{write} : (\mathtt{int}, \tau) \to \mathtt{unit}$, etc. Indeed, this is what many FFIs do. However, introducing such imprecision into our types makes them less useful for reasoning about programs. Moreover, it poses a direct threat to soundness; in this case, a programmer can easily pass a "bad" integer to read/write. Instead, our approach starts by giving these foreign functions *precise* types, with which we can then work backwards through the implementation and the soundness proof. We demonstrate how to do this in a systematic way that applies to a wide variety of features.

The first step, and core idea, is to introduce new *linking types* that can describe foreign behavior. In Fig. 5, we present the extension for state, where extended language features are typeset in **pink bold font** and extended metatheory is distinguished with the S marker. We add a reference type, $\mathbf{ref}\ \tau$, without any introduction or elimination forms, since only foreign code can manipulate references. We also replace our function type with a pair of *modal* arrows, where $\overset{\circ}{\to}$ types pure functions and $\overset{\bullet}{\to}$ types stateful functions. We use $\overset{\LEFTcircle}{\to}$ when the particular mode is unimportant.

With this linking types extension, we can import a mutable reference library at a more precise type. Since FunLang does not have polymorphism, we pick a concrete type τ when we import them:

$$\mathtt{import}(\mathtt{alloc} : \tau \overset{\bullet}{\to} \mathbf{ref}\ \tau, \mathtt{read} : (\mathbf{ref}\ \tau) \overset{\bullet}{\to} \tau, \ldots)\ \ \ldots$$

τ	$\uparrow\tau$	$\downarrow\tau$
$\widehat{\tau}$	$\widehat{\tau}$	$\widehat{\tau}$
$\tau_1 \times \tau_2$	$\uparrow\tau_1 \times \uparrow\tau_2$	$\downarrow\tau_1 \times \downarrow\tau_2$
$\tau_1 + \tau_2$	$\uparrow\tau_1 + \uparrow\tau_2$	$\downarrow\tau_1 + \downarrow\tau_2$
$\mu\alpha.\tau$	$\mu\alpha.\uparrow\tau$	$\mu\alpha.\downarrow\tau$
$(\tau_1, \ldots, \tau_n)\rightarrow\tau'$	$(\uparrow\tau_1, \ldots, \uparrow\tau_n)\overset{\circ}{\rightarrow}\uparrow\tau'$	see below

τ	$\downarrow\triangleq$
$(\tau_1, \ldots, \tau_n) \overset{\LEFTcircle}{\rightarrow} \tau'$	$(\downarrow\tau_1, \ldots, \downarrow\tau_n)\rightarrow\downarrow\tau'$
$\mathbf{ref}\ \tau$	`unit`

Figure 6. Lift and lower functions for state extension

$$
\begin{aligned}
&\wr \mathbf{ref}\ \tau \int &&\triangleq \text{free; push } 0\\
&\wr (\tau_1, \ldots, \tau_n) \overset{\bullet}{\rightarrow} \tau' \int &&\triangleq \text{push (thunk lam l.push l; free); getlocs}\\
&\wr \tau \int &&\triangleq \cdot \qquad \text{for any other } \tau
\end{aligned}
$$

$$
\begin{aligned}
\text{ALLOC} &\triangleq \text{thunk push (thunk lam falloc.lam f.push f; alloc); fix}\\
\text{READ} &\triangleq \text{thunk push (thunk lam fread.lam r.push r; read); fix}\\
\text{WRITE} &\triangleq \text{thunk push (thunk lam fwrite.lam f.lam r.push r;}\\
&\qquad \text{push f; write; push 0); fix}
\end{aligned}
$$

Figure 7. State boundary enforcement & target library code

Notice that the linking types extension is purely *static*; it does not introduce any new *term-level* syntax. So, intuitively, core programs should be usable inside of extended programs, and pure extended programs should be usable inside of core programs. To make this intuition precise, we specify a pair of type-level metafunctions, *lift* and *lower*. Lift, denoted $\uparrow\tau$, maps a core type to an extended type, while lower, denoted, $\downarrow\tau$ maps an extended type to a core type.[2] The definitions of lift and lower for the state extension are given in Fig. 6. Note that core arrows $\rightarrow$ are lifted to pure arrows $\overset{\circ}{\rightarrow}$ in the extension, which is why we did not include an introduction form for $\overset{\circ}{\rightarrow}$ in Fig. 5: it suffices to build a core function and lift it! Since references cannot be used directly inside core code, lowering simply erases them. The only truly surprising case is that impure arrows are lowered to core arrows; this surely *seems* unsound, which we will address shortly.

To facilitate the interaction between core and extended programs, we rely on a generic *boundary* term, $\{\mathtt{e}\}^{E}_{\downarrow\tau}$, that delineates between linking code—in extension E—and the rest of the core program (Fig. 5). Here, $\mathtt{e}$ is a stateful program whose result is a τ when typechecked with the extension S, which means it can be used at type $\downarrow\tau$ in the rest of the program. The boundary acts as a syntactic cue for the typechecker to switch to an extension before entering the body. To do so, the typechecker extracts imports relevant to the extension and lifts every binding from the typing context. This mechanism allows us to, for example, define top-level core functions and use them as pure functions under S boundaries. A well-formedness judgment disallows different extensions from being mixed in a single import binding.

[2]The notation is inspired by the adjoint logic of Pfenning and Griffith [54].

Next, for every extended type τ, we define a target-level *encapsulation wrapper*, $\wr\tau\int$, which is code that dynamically enforces that its argument *behave like a* $\downarrow\tau$. While the formal proofs are in [51], here we give an intuitive explanation of the wrappers for state, which are defined in Fig. 7. Recall that only two types had non-obvious lower definitions: $\mathbf{ref}\ \tau$ and $\overset{\bullet}{\rightarrow}$. The former we decided to erase, and the latter we suspiciously mapped to the pure arrow. The wrappers for these types justify those decisions because they give a strategy for encapsulating stateful behavior from the rest of the program. If a boundary returns a $\mathbf{ref}\ \tau$, we free its location in memory and return a semantic unit. If a boundary returns an impure function, we use getlocs to free any memory held by that function. In both cases, purity is enforced by indirection: if the memory associated with these terms is used outside of the boundary (which is a side-effect), then a memory trap (fail Mem)[3] halts the program. All other linking types τ need no dynamic wrapper.

In Fig. 7, we also provide a StackLang implementation of a mutable reference library. While the full relation and proof showing this safe is in [51], intuitively, the functions behave as one would expect (N.B., they account for the calling convention of `FunLang`).

The Whole Picture. With all the pieces of the state extension in place, we return our attention to the `fastfib` example, defined in Fig. 8. To start, we import our foreign functions, `alloc`, `read`, and `write`, specialized to $\mathtt{int} \overset{\bullet}{\rightarrow} \mathtt{int}$ payloads so that we can store our memotable. In the body of `fastfib`, we immediately enter an S boundary so that we may use the foreign imports. We `alloc` an empty table, `mtbl`, initialized to return a sentinal value, -1, on any input. Next, we define the helper function `mutfib`, which does most of the heavy lifting. For any input `x` that is not a base case, `mutfib` begins by checking if `x` is in `mtbl`. If it is, it returns the result. Otherwise, it computes a fresh output and stores it in `mtbl` before returning it. At the top-level, `mutfib` is invoked on the input to `fastfib`. Note that whereas memoized functions can sometimes see speed-ups across top-level calls, here, different top-level calls to `fastfib` are encapsulated from one another; i.e., the table is dropped across calls. Naturally, one could write batched-input variant.

5 Exceptions

The key guarantee of a linking types extension is *sound encapsulation*, not only from core code, but from other extensions as well. Indeed, one can use different extensions in different parts of the same program. In this section, we develop another example: an extension for exceptional control flow. However, since encapsulated code needs to be isolated

[3]We consider memory traps an acceptable error in our definition of soundness.

```
import(alloc : ((int) -∘-> int) -•-> ref ((int) -∘-> int),
       read : (ref ((int) -∘-> int)) -•-> ((int) -∘-> int),
       write : (ref ((int) -∘-> int), ((int) -∘-> int)) -•-> unit)
fun fastfib(y : int){
  {let mtbl = alloc(fun f(n : int){−1}) in
   fun mutfib(x : int){
     if x = 0 {0}{if x = 1{1}{
       let m = read(mtbl) in
       if m(x) = −1{
         let r = mutfib(x + −1) + mutfib(x + −2) in
         let _ = write(mtbl, fun f(n){if n = x{r}{m(x)}}) in
         r
       }{m(x)}
     }
  }(y)}^S_int
}
```

Figure 8. Example: fibonacci memoized with state

τ	$\uparrow\tau$	$⤈\tau$
$\widehat{\tau}$	$\widehat{\tau}$	$\widehat{\tau}$
$\tau_1 \times \tau_2$	$\uparrow\tau_1 \times \uparrow\tau_2$	$⤈\tau_1 \times ⤈\tau_2$
$\tau_1 + \tau_2$	$\uparrow\tau_1 + \uparrow\tau_2$	$⤈\tau_1 + ⤈\tau_2$
$\mu\alpha.\tau$	$\mu\alpha.\uparrow\tau$	$\mu\alpha.⤈\tau$
$(\tau_1, \ldots, \tau_n) \rightarrow \tau'$	$(\uparrow\tau_1, \ldots, \uparrow\tau_n) \xrightarrow{\square} \uparrow\tau'$	see below

τ	$⤈ \triangleq$
$⤈(\tau_1, \ldots, \tau_n) \xrightarrow{\square} \tau'$	$(⤈\tau_1, \ldots, ⤈\tau_n) \rightarrow ⤈\tau'$
$⤈(\tau_1, \ldots, \tau_n) \xrightarrow{\blacksquare} \tau'$	$(⤈\tau_1, \ldots, ⤈\tau_n) \rightarrow \mathsf{U} + (⤈\tau')$
$⤈\mathbf{ref}\ \tau$	$\mathtt{unit}$

where $\mathsf{U} \triangleq \mu\alpha.\mathtt{unit} + \mathtt{int} + (\alpha \times \alpha) + (\alpha + \alpha) + ((\alpha) \rightarrow \alpha) + \alpha$
and $\downarrow\tau \triangleq \mathsf{U} + ⤈\tau$

Figure 9. Lift and lower functions for exceptions extension

$$\begin{aligned}
\text{CATCH} &\triangleq \text{thunk push (thunk lam fcatch.lam f.push f; call;} \\
&\quad \text{lam res.push [1, res]; reset); fix} \\
\text{THROW} &\triangleq \text{thunk push (thunk lam fthrow.lam exn.push [0, exn];} \\
&\quad \text{shift \_ ()); fix} \\
\wr \mathbf{ref}\ \tau \int &\triangleq \text{free; push [1, 0]; reset} \\
\wr (\tau_1, \ldots, \tau_n) \xrightarrow{\blacksquare} \tau' \int &\triangleq \text{DUP; push (thunk lam l.push l; free);} \\
&\quad \text{getlocs; lam res. push [1, res]; reset} \\
\wr \tau \int &\triangleq \text{lam res.push [1, res]; reset} \quad \text{where } \tau \notin \text{above}
\end{aligned}$$

Figure 10. Exception target library & boundary enforcement

from other parts of the program, we cannot have a single boundary with multiple extensions active simultaneously.

To freely interleave control flow and mutable state, we take the state extension from the previous section as a starting point for this new extension. While it may feel unsatisfying that these features cannot be teased apart and composed, we imagine that many practical linking types extensions really *would* bundle multiple features together, since complex foreign functions are likely to have, e.g., many different side-effects that interact in subtle and inseparable ways.

As before, first we introduce linking types. For simplicity, our modal arrows only signal whether a function is pure or impure; they do not to distinguish between different effects (i.e., state vs. control), though our approach is certainly compatible with a more granular type system. Therefore, this part of the extension is exactly the same as in the previous section. For contrast, we typeset this extension in **orange bold font**, use square modal arrows, $\xrightarrow{◪}$, and identify it with X.

Next we define lift and lower, which we present in Fig. 9. While lift is exactly the same as in the previous case study, lower is quite different. Since code under an X boundary might throw an uncaught exception, lower must account for the type of an exceptional result. Therefore, we define lower in two steps: the helper metafunction $⤈$ accounts for the success case, and the top level $\downarrow$ merges it with the exception case under a sum. Since exceptions might produce a variety of values, we use the universal type U in the exception case.[4]

Next, we provide a StackLang implementation of an exceptions library in Fig. 10. Although defining encapsulation wrappers is really the next step of the approach, the library implementation offers some intuition for the way that we model exceptions using delimited continuations in StackLang. A shift with an empty body discards the program until the next reset, which intuitively corresponds to throwing and catching an exception, respectively. In the implementation, THROW is responsible for tagging an exception value, while CATCH is responsible for tagging a success value. Note that CATCH takes a function corresponding to the computation to run.

With an intuitive understanding of StackLang exceptions, we are ready to define the target-level encapsulation wrappers. Unlike the previous extension, wrappers are required for *every* τ, because the boundary must always be prepared for an uncaught exception. In all cases, the wrapper is responsible for capturing any escaping exceptions, which it does with reset (N.B., if there is no shift under the boundary, this reset is effectively a no-op). For types with pure *values*, the encapsulation wrapper can simply tag the success value before it resets. Meanwhile, references and impure functions are handled as before, modulo tagging and a reset.

With this new extension, we can improve upon our `fib` function (Fig. 11): this time, we take a list of inputs (1), compute all of their results with a single memotable (2), and throw an exception on bad inputs (3). Notice that there is not a corresponding `catch`, since, in this example, a bad input is an unrecoverable error. Still, our program—and indeed, any program with uncaught exceptions—is safe because the boundary code catches and tags escaping exceptions.

[4]While an important aspect of exceptions are identity, which this doesn't expose statically, we wanted to focus on the other important aspect: non-local control flow.

```
import( alloc, read, write : [See fig. 8],
          catch : (() ■→ U) □→ U + U,
          throw : (U) ■→ int)
fun fiblist(lst : μα.(int × α) + unit){                ①
  {let mtbl = alloc(fun f(n : int){−1}) in              ②
   let mf = fun mutfib(y : int){[See fig. 8]} in
   fun mutfiblist(l : μα.(int × α) + unit){
     match unfold l
       x {if fst x < 0 {throw(fold inl ())} {           ③
         fold inl(mf(fst x), mutfiblist(snd x))}
       y {fold inr()}
     }
  }(lst)}^X_{U + μα.(int × α) + unit}
}
```

Figure 11. Example: fibonacci with input checks and memoization

Implement

1. Extend the type system ($\vdash_E$).
2. Define lift ($\upharpoonleft\tau$) and lower ($\downharpoonleft\tau$).
3. * Develop encapsulation wrappers ($\wr\tau\int$).

Model

4. Design a core realizability model[5] ($[\![\vdash \tau]\!]$).
5. Design an extended realizability model ($[\![\vdash_E \tau]\!]$).

Verify

6. Prove lift ($\upharpoonleft\tau$) sound.
7. * Prove encapsulation $\wr\tau\int$ enforces lower ($\downharpoonleft\tau$).
8. Prove "compatibility" lemmas.
9. Prove libraries semantically well-typed.

Figure 12. Our approach, summarized (for *, see §6.5)

6 Soundness

We follow prior work on interoperability [49] and verify semantic type soundness using realizability models, which are sets of target terms indexed by source types. The application of such models goes back to Benton and collaborators [9, 10], who used them to prove type soundness for standalone languages. Here, we first build a core model for FunLang (Item 4), and then another model for each linking types extension (Item 5). Since realizability models are inhabited by *target* terms, we can interpret linking types (e.g., **ref** τ) whose behavior is inexpressible in core FunLang. Also, the fact that the inhabitants of these models share a common operational semantics will be instrumental in proving that the boundary typing rule is sound. We summarize our approach in Figure 12.

While the full proofs are in [51], here we highlight a few key details.

6.1 Proving $\upharpoonleft$ Sound

There are two lemmas that we need to prove, one for $E = \mathsf{S}$ and one for $E = \mathsf{X}$ (Item 6), here stated as paraphrased English:

Lemma 6.1 (Lift E). *Any value* v *in the model* $\mathcal{V}^E[\![\upharpoonleft\tau]\!]$ *is also in the model* $\mathcal{V}^\lambda[\![\tau]\!]$*, and vice versa.*

Proof (Sketch). This seems initially difficult, as the extended model ($\mathcal{V}^E[\![\upharpoonleft\tau]\!]$) has additional logical structure, to account for features unknown by the core model. The key observation is that the lifted types are exactly those that do not rely upon any of that additional structure – e.g., for our model with state, they are exactly those values that have no relevant portions of the heap. Indeed, this largely motivated the design of the linking types and the lift function $\upharpoonleft$. □

6.2 Proving $\wr\tau\int$ Satisfies $\downharpoonleft$

Unlike the previous proofs, this is non-trivial. Intuitively, for the state extension, one wants to show that a program P; $\wr\tau\int$ is in $\mathcal{E}^\lambda[\![\downharpoonleft\tau]\!]$ whenever P is in $\mathcal{E}^{\mathsf{S}}[\![\tau]\!]$ (with the same logical state), but this isn't always the case. The problem is that P may manipulate the heap and end up with logical constraints on it that $\mathcal{E}^\lambda[\![\downharpoonleft\tau]\!]$ will not enforce. Indeed, that relation only admits terms that run under *arbitrary* heaps, so a term depending on a type invariant at a particular location will trigger an unacceptable Type error on some executions. On the other hand, $\downharpoonleft\tau$ is FunLang type, so how are we to proceed? For the state extension, we begin by proving a subtle variation of the statement above (Item 7). It suffices to focus on values when exceptions are uninvolved, since terms always diverge, error, or run to a value:

Lemma 6.2 (Encapsulation S). *If* v *is in* $\mathcal{V}^{\mathsf{S}}[\![\tau]\!]$ *then* push v; $\wr\tau\int$ *is in* $\mathcal{E}^{\mathsf{S}}[\![\upharpoonleft\downharpoonleft\tau]\!]$

The full proof is somewhat involved (see [51]), since we must consider each type and its associated wrapping code. Nevertheless, we can show that state is encapsulated; i.e., $\upharpoonleft\downharpoonleft\tau$ is stateless even if τ is stateful. To show that it is safe to bring encapsulated results across the boundary, we can compose this lemma with Lemma 6.1 just by evaluating intermediate terms. The proof pipeline looks roughly like this, eliding logical state (Kripke worlds, etc), and divergence/errors:

$$\mathsf{P} \in \mathcal{E}^{\mathsf{S}}[\![\tau]\!] \overset{\text{eval}}{\Rightarrow} \mathsf{v} \in \mathcal{V}^{\mathsf{S}}[\![\tau]\!] \overset{6.2}{\Rightarrow} \text{push v}; \wr\tau\int \in \mathcal{E}^{\mathsf{S}}[\![\upharpoonleft\downharpoonleft\tau]\!] \overset{\text{eval}}{\Rightarrow}$$
$$\mathsf{v}' \in \mathcal{V}^{\mathsf{S}}[\![\upharpoonleft\downharpoonleft\tau]\!] \overset{6.1}{\Rightarrow} \mathsf{v}' \in \mathcal{V}^{\lambda}[\![\downharpoonleft\tau]\!]$$

For the exception extension, we follow the same general approach, but we can no longer assume that a term will run down to a value before reaching the encapsulation wrapper, since it may throw an exception in first, so we have to prove a slightly different, but analogous lemma.

6.3 Proving Libraries Satisfy Types

The next step is to prove that each library function we are linking with satisfies the type that we are importing it at (Item 9). We note that a single library function may be compatible with multiple types, even across different extensions with different reasoning principles. This is especially relevant here because our library functions are naturally polymorphic even though FunLang is not. To reconcile this mismatch, when we prove the libraries sound, we actually do quantify over all concrete types; i.e., we treat the libraries as polymorphic in the metalanguage.

We recall all the library code used in our case studies; the first three functions are used in the state extension, and all are used in the exception extension. For each function, we also include a type we intend to import it at.

$$\begin{array}{lll}
\mathsf{ALLOC} & : (\tau) \xrightarrow{\bullet} \mathbf{ref}\ \tau & \triangleq \mathsf{t{-}p}\ (\mathsf{t{-}l}\ \mathsf{f}\mathsf{alloc.lam}\ \mathsf{x.push}\ \mathsf{x};\mathsf{alloc});\mathsf{fix} \\
\mathsf{READ} & : (\mathbf{ref}\ \tau) \xrightarrow{\bullet} \tau & \triangleq \mathsf{t{-}p}\ (\mathsf{t{-}l}\ \mathsf{f}\mathsf{read.lam}\ \mathsf{l.push}\ \mathsf{l};\mathsf{read});\mathsf{fix} \\
\mathsf{WRITE} & : (\mathbf{ref}\ \tau, \tau) \xrightarrow{\bullet} \mathtt{unit} & \triangleq \mathsf{t{-}p}\ (\mathsf{t{-}l}\ \mathsf{f}\mathsf{write.lam}\ \mathsf{x.lam}\ \mathsf{l.push}\ \mathsf{l}; \\
 & & \quad \mathsf{push}\ \mathsf{x};\mathsf{write};\mathsf{push}\ 0);\mathsf{fix} \\
\mathsf{CATCH} & : (() \xrightarrow{\blacksquare} \tau) \xrightarrow{\square} \mathsf{U} + \tau & \triangleq \mathsf{t{-}p}\ (\mathsf{t{-}l}\ \mathsf{f}\mathsf{catch.lam}\ \mathsf{f.push}\ \mathsf{f};\mathsf{call}; \\
 & & \quad \mathsf{lam}\ \mathsf{res.push}\ [1,\mathsf{res}];\mathsf{reset});\mathsf{fix} \\
\mathsf{THROW} & : (\mathsf{U}) \xrightarrow{\blacksquare} \tau & \triangleq \mathsf{t{-}p}\ (\mathsf{t{-}l}\ \mathsf{f}\mathsf{throw.lam}\ \mathsf{exn.push}\ [0,\mathsf{exn}]; \\
 & & \quad \mathsf{shift}\ \_\ ());\mathsf{fix}
\end{array}$$

$$\text{where } \mathsf{t{-}p} = \mathsf{thunk\ push} \text{ and } \mathsf{t{-}l} = \mathsf{thunk\ lam}$$

For each one, we have to show that the code is in the value relation at the corresponding type.

6.4 Compatibility Lemmas & Type Soundness

Our ultimate goal is to prove that FunLang, together with these linking types extensions, is type sound. Using the models constructed so far, we give a *semantic* proof of type soundness. First, we show that all syntactically well-typed terms belong to the model, which is done via so-called compatibility lemmas (Item 8). Second, we show that all terms in the model are well-behaved, which follows directly from the definition of the model. Composing these two steps, we conclude that syntactically well-typed programs are well-behaved.

Because the value and expression relations contain only closed terms (in the same way as the simple model shown in §3), we use closing substitutions to account for the typing contexts above. Typically, these substitutions are drawn from the model itself. i.e., we consider an open term e such that for some mapping of variables to (closed) values in the value relation γ, $\gamma(e)$ is in the (closed) expression relation.

We do something similar here, except that we cannot hope to find a closing substitution (i.e., γ) for the imported code that satisfies the core model! Indeed, the whole point of importing at linking types is that the associated programs are *outside* the core model! Thus, we close off the imports by drawing substitutions from the *extended* models instead.

All of the compatibility lemmas and their proofs can be found in [51].

6.5 Discussion

Dynamic Checks. As indicated in Items 3 and 7 of our approach, inserting wrappers that perform dynamic checks is, in some sense, optional, but the proof obligation that they discharge is certainly not. In particular, without wrappers, the compatability lemma for a boundary $\{\mathtt{e}\}^{\mathsf{E}}_{\downarrow\tau}$ requires a much stronger assumption; namely, that $\mathtt{e}$ behaves like an $\mathcal{E}^{\lambda}[\![\downarrow\tau]\!]$. This assumption percolates all the way up to the central soundness theorem, which correspondingly needs to be weakened to accept proofs that each boundary is encapsulated. Moreover, this shifts the burden of proof off of the language designer and on to the *user*, who must now verify every block of their linking code. This trade-off may be prohibitive for many applications, but for performance-critical domains, efforts like RustBelt [32] suggest that such specialized verification is viable. Also, without dynamic checks, one could potentially admit so-called *benign* global effects, like global memoization [15, 53], which are not technically encapsulated yet have no impact on the result of a program.

On the other hand, dynamic checks can be useful for security reasons; e.g., if a foreign library is loaded dynamically from a potentially-malicious source. Recent work by Sammler et al. [58] shows how to ensure *robust safety* in a low-level language via sandboxing. Just as we show that core code is safe in the presence of foreign code, they show that trusted code is safe in the presence of untrusted code, though their sub-languages have similar expressive power. Earlier work by Swasey et al. [60] shows how to verify common dynamic enforcement patterns (e.g., sealing), which may be helpful when developing encapsulation wrappers for future linking types extensions.

Beyond Safety. In this paper, we focus only on *soundness*, so our models are unary—they simply characterize well-behaved terms. Still, our approach is theoretically amenable to stronger properties, like preserving equivalences. One would instead use binary models and show that the presence of linking code does not break equivalences from the core model. This is in the spirit of recent work [28, 64] proving that the ST Monad in Haskell preserves purity, though our approach is agnostic to the particular effects or encapsulation techniques. Indeed, the semantically typed back-translation of [28] is very similar to our use of realizability models and the way they allow us to leverage encapsulation wrappers to show that encapsulated linking code behaves as a core type.

Negative Expressivity. In both of our case studies, the linking types provide *positive expressivity* [19]: they characterize a strictly larger class of programs than do base types. One could also develop a *negatively* expressive extension, which *restricts* the class of programs *under* a boundary. For example, to link FunLang with a linear language's library, we would develop a *linear* linking types extension.

In this case, lift and lower would enforce the *independence principle* of adjoint logic [54, 57], which ensures that unrestricted code does not misuse linear values. Note the shift in perspective: whereas we have been talking about protecting core from foreign, in this case we would be protecting foreign from core. Thus, a more general characterization of our approach is it ensures the integrity of the boundary, regardless of the expressive "side" on which core sits.

7 Related Work

Linking Types Position Paper In 2017, two of the authors proposed ([48]) the idea of linking types as a way of building fully abstract compilers that support linking with code inexpressible in the source language. While some of overall idea remains the same, the current goal is to prove type soundness in the presence of inexpressible behavior, not full abstraction. While type soundness is a weaker theorem, unlike full abstraction it is one that most typed languages at least aspire to satisfy, and thus our approach is one that can be readily adopted by existing typed languages. The current work also differs in strategy, building realizability models that allow encapsulation proofs, which are entirely absent from the previous approach. Indeed, the notion of a block of code $\wr\tau\int$ that enforces safe encapsulation, a key element of our approach, is not something previously considered. Without the realizability models we use, describing such target code that ensures we can move from behavior of linking types to behavior of core types is not possible, and perhaps because of that, that paper also relies on novel terms, not intended for programmer use, to inhabit the linking types. Further, that earlier work imposed more restrictive properties on the functions relating core and extended types (called $\kappa^+()$ and $\kappa^-()$ instead of $\uparrow$ and $\downarrow$)—namely that they form an embedding-projection pair, so that $\kappa^-(\kappa^+(\tau)) = \tau$. The current approach is more flexible, and therefore allows simple linking-type systems.

Multi-languages and Interoperability Many have used the idea of a syntactic multi-language from Matthews and Findler [40], where the syntax of both interoperating languages are embedded into a single language and enhanced with boundary terms [2, 25, 26, 43, 46, 50, 52, 59, 65]. However, one critical but often unnoticed issue with this approach is that type soundness is proved of the multi-language, where the behavior prescribed by some type τ in the multi-language need not be the same as the behaviors allowed by τ in the corresponding core language. For simple languages (e.g., where the only effect is divergence as in [40]) this may be an immaterial distinction, but even by adding state, the operational semantics of the multi-language now must consider the heap, and thus a pure language embedded in such a multi-language may no longer behave the same, as there is now a heap threaded through. Scherer et al. [59] consider this issue and argue for fully abstract embedding into a multi-language. In a slightly different vein, there has also been some work mixing bindings [6] and building multi-language runtimes [66], but this work does not consider formal semantic properties.

Recent work by Patterson et al. [49] more directly addresses the limitations of the syntactic multi-language approach, building realizability models for the two languages rather than a source-level operational semantics and then proving soundness in terms of that shared target-level representation. Our work makes two improvements over theirs. First, we define a notion of soundness of the core language independent of the linked code, in the form of the model for our core language. Since we use similar realizability techniques, we expect this approach could be adapted to the work of Patterson et al. [49], but, as presented, their models share logical state across the languages. Second, and more fundamentally, while they allow linking with inexpressible behavior, at the boundary such code must be given a sound type in the native language, which means that all use of that behavior must be contained in the foreign language. For example, if one wanted to write code that uses exceptions, all code using exceptions would have to be implemented in the foreign language, and only the end result (an encapsulated component) could be brought across. This contrasts with our approach, which defines an extended sublanguage where only the `try` and `catch` primitives are imported, and the rest of the code is implemented within the core language.

Foreign Function Interfaces (FFI) Many researchers have investigated FFIs and how to make them safer, often considering a particular pair of languages, where one of them is usually C [12, 23, 24, 35, 61–63]. There has also been work extending the annotations that are written down so that there is less hand-written (and thus error-prone) code to write, with much work in the context of the Haskell FFI [14, 20, 30]. While the latter certainly care about Haskell type invariants, it's not clear from these papers whether any formal soundness properties were proved.

Another approach to having rich FFIs is to co-design both languages, as has been done in the verification project Everest [11], where a low-level C-like language Low* is designed [56] to interoperate with an embedding of a subset of assembly suitable for cryptography [22].

By embedding both languages into the verification framework F*, they are able to prove rich properties about the interactions between the two languages, but this approach is less useful for existing languages, our primary focus.

Rust The Rust language has a built-in mechanism for embedding "unsafe" code that could not satisfy the typechecker of "safe" Rust. There have been efforts to characterize the semantic behavior of safe Rust ("unsafe code guidelines") and prove that some unsafe code, while syntactically not well typed, does not violate those properties. Most notably, the RustBelt [32] project gives a semantic model of λ_{Rust}

types and uses it to prove the soundness of λ_{Rust} typing rules, but also to prove that the λ_{Rust} implementations of standard library features (essentially unsafe code) are semantically sound inhabitants of their ascribed type specification. We argue that Rust's goals for unsafe and how RustBelt approached them fit into our approach: the unsafe code unquestionably has behavior inexpressible in the rest of Rust, and they already have a syntactic boundary construct: unsafe blocks. Moreover, for RustBelt, Jung et al. [32] created a lifetime logic that could be used to semantically model (some of) that behavior. While in this paper we expect syntactic type checkers for the extended language, there is no reason why a lifetime logic approach isn't equally valid, and may be necessary for sufficiently complex behavior. Since RustBelt uses the same lifetime logic to define the semantics of safe Rust types, the type functions are perhaps not as apparent, but the properties they convey are: in particular, since they do not insert code around unsafe blocks, they need to prove that the code inside satisfies a safe Rust type. Put another way, any encapsulation has to be inlined into the library implementation, rather than inserted by the compiler.

Semantic Models and Realizability Models The use of semantic models to prove type soundness has a long history [41]. We make use of step-indexed models [3, 4], 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. Our realizability models interpret source types as sets of target terms. This work follows a line of work by Benton and collaborators on "low-level semantics for high-level types" (a.k.a. "realistic realizability") [7]. Such models have been used to prove type soundness of standalone languages, specifically, Benton and Zarfaty [10] proved an imperative while language sound and Benton and Tabareau [9] proved type soundness for a simply typed functional language, 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 [7]. Krishnaswami et al. [34] make use of a realizability model to prove consistency of LNL_D a core type theory that integrates linearity and full type dependency: this is a form of interoperability, but as the FFI work above, it is a concrete instantiation for a particular problem, rather than an general approach. Such realizability models have also been used by Jensen et al. [29] to verify low-level code using a high-level separation logic, by Benton and Hur [8] to verify compiler correctness, and by New et al. [42, 44, 45] in their work on semantic foundations for gradual typing.

8 Conclusion

We have presented a framework for encapsulating foreign code which is based on the idea of giving precise *linking types* to libraries that provide behavior that is inexpressible in the core language. These types are then used to typecheck encapsulated portions of code that use the external libraries, and the results are then wrapped in boundaries that ensure they behave as a type in the core language. The technical development leverages realizability models, which interpret source types as sets of target terms and thus allow us to build different models for the different type systems, yet move terms between them in our proofs.

There are many possible extensions, both practical and theoretical. On the practical end, we think there is interesting research to be done on suitable type mechanisms for the linking types themselves. In particular, the balance of expressivity and usability that we would want suggests that some sort of extensible type system may be the right tool for this: perhaps an indexed monad, building on the work of Maillard et al. [39], or maybe an effect type system in the vein of Koka [36, 37]. On the other hand, there may also be interesting work in combining different extensions: in this work, we took the position that extensions could not be combined, but it is possible that this is too restrictive of a position, and that such an extensible type system could be combined with sets of models that capture different effects and can be composed together, as needed. Clearly, the fact that our model for both exceptions and state recapitulates the definitions of the state model seems disappointing: perhaps there is a better way.

In addition to extensions, there are also ways of making the problem more specific that we think are interesting. For example, consider recent work [13] embedding OCaml code into Coq by way of an encoding that represents the OCaml as non-deterministic functions. While the types are defined *within* Gallina, the reliance on extraction to inhabit those types leads us to believe that this sort of system could be proved sound using exactly the type of framework we propose. Indeed, the interaction of verified code, or the verification base itself, with untrusted code is an important interoperability problem, and one that we think the *linking types* framework could be useful for.

Finally, while we have focused on unary models and type soundness theorems, we think there are also possibly interesting results from considering binary models. In that setting, we can directly reason about equivalence, whether that is used to show theorems that should hold of models (e.g., refactorings that should hold in a side-effect free language), or about security properties like full abstraction.

Acknowledgements

We thank anonymous reviewers for their in-depth comments. This material is based upon work supported by the National Science Foundation under Grant No. CCF-1816837 and CCF-1453796.

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

[4] 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

[5] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J Summers. 2020. How do programmers use unsafe Rust? *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.

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

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

[8] 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

[9] 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.

[10] 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

[11] 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

[12] 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.

[13] Sylvain Boulmé and Thomas Vandendorpe. 2019. Embedding Untrusted Imperative ML Oracles into Coq Verified Code. (July 2019). https://hal.archives-ouvertes.fr/hal-02062288 This preprint has been largely rewritten and integrated into Sylvain Boulm{é}'s Habilitation in 2021, See http://www-verimag.imag.fr/ boulme/hdr.html..

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

[15] Byron Cook and John Launchbury. 1997. Disposable memo functions. In *ICFP*, Vol. 97. 310.

[16] Olivier Danvy and Andrzej Filinski. 1990. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. 151–160.

[17] Derek Dreyer, Georg Neis, and Lars Birkedal. 2012. The Impact of Higher-Order State and Control Effects on Local Relational Reasoning. *J. Funct. Program.* 22, 4–5 (aug 2012), 477–528. https://doi.org/10.1017/S095679681200024X

[18] Mattias Felleisen. 1988. The Theory and Practice of First-Class Prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 180–190. https://doi.org/10.1145/73560.73576

[19] Matthias Felleisen. 1990. On the expressive power of programming languages. In *European Symposium on Programming*. Springer, 134–151.

[20] 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.

[21] The Rust Foundation. [n. d.]. *The Rust Standard Library*. https://doc.rust-lang.org/std/keyword.extern.html

[22] 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

[23] Michael Furr and Jeffrey S. Foster. 2005. Checking Type Safety of Foreign Function Calls. 62–72.

[24] Michael Furr and Jeffrey S. Foster. 2008. Checking Type Safety of Foreign Function Calls.

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

[26] 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.

[27] Unsafe Code Guidelines Working Group. [n. d.]. *Unsafe Code Guidelines Reference*. https://rust-lang.github.io/unsafe-code-guidelines/introduction.html

[28] Koen Jacobs, Dominique Devriese, and Amin Timany. 2022. Purity of an ST monad: full abstraction by semantically typed back-translation. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–27.

[29] 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

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

[31] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. 1999. C—: A portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*. Springer, 1–28.

[32] 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)*.

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

[34] 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

[35] 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

[36] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014 (EPTCS, Vol. 153)*, Paul Blain Levy and Neel Krishnaswami (Eds.). 100–126. https://doi.org/10.4204/EPTCS.153.8

[37] Daan Leijen. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 486–499. https://doi.org/10.1145/3009837.3009872

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

[39] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra Monads for All. *Proc. ACM Program. Lang.* 3, ICFP, Article 104 (jul 2019), 29 pages. https://doi.org/10.1145/3341708

[40] 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

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

[42] 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.

[43] 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

[44] 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.

[45] 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.

[46] 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

[47] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. 2008. *Real world Haskell: Code you can believe in.* " O'Reilly Media, Inc.", Chapter 17.

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

[49] Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. 2022. Semantic Soundness for Language Interoperability. In *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2022, San Diego, California, June 13-17, 2022.* ACM.

[50] 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

[51] Daniel Patterson, Andrew Wagner, and Amal Ahmed. 2023. Semantic Encapsulation using Linking Types (Technical Appendix). (July 2023). Available at `https://dbp.io/pubs/2023/lt-tr.pdf`.

[52] 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

[53] Simon Peyton Jones, Simon Marlow, and Conal Elliott. 1999. Stretching the storage manager: weak pointers and stable names in Haskell. In *Symposium on Implementation and Application of Functional Languages*. Springer, 37–58.

[54] Frank Pfenning and Dennis Griffith. 2015. Polarized substructural session types. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, 3–22.

[55] LLVM Project. [n. d.]. *LLVM Reference.* https://www.llvm.org/docs/GarbageCollection.html#gcroot

[56] 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

[57] Klaas Pruiksma, William Chargin, Frank Pfenning, and Jason Reed. 2018. Adjoint logic. *Unpublished manuscript, April* (2018).

[58] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2019. The High-Level Benefits of Low-Level Sandboxing. *Proc. ACM Program. Lang.* 4, POPL, Article 32 (dec 2019), 32 pages. https://doi.org/10.1145/3371100

[59] 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

[60] David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and Compositional Verification of Object Capability Patterns. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 89 (oct 2017), 26 pages. https://doi.org/10.1145/3133913

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

[62] Gang Tan and Greg Morrisett. 2007. Ilea: Inter-language Analysis Across Java and C. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. ACM, New York, NY, USA,

39–56. https://doi.org/10.1145/1297027.1297031

[63] Zachary Tatlock, Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. 2008. Deep Typechecking and Refactoring. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications* (Nashville, TN, USA) *(OOPSLA '08)*. ACM, New York, NY, USA, 37–52. https://doi.org/10.1145/1449764.1449768

[64] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2017. A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–28.

[65] 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).

[66] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

Received 2023-06-01; accepted 2023-06-29