Linking Types for Multi-Language Software: Have Your Cake and Eat It Too

Daniel Patterson[1] **and Amal Ahmed**[2]

1 **Northeastern University, Boston MA, USA**
dbp@ccs.neu.edu
2 **Northeastern University, Boston MA, USA**
amal@ccs.neu.edu

Abstract

When implementing large software systems, programmers should be able to implement each part of the system using the programming language best suited to the task at hand. But current multi-language systems are difficult for programmers to reason about because components interact only after compilation to a lower-level target. The source-level contexts that programmers use to reason about their components no longer match the reality of the contexts that their components are running in. The additional contexts impact the notion of equivalence that justifies behavior-preserving modifications of code, whether programmer refactorings or compiler optimizations.

We advocate that language designers provide programmers with the means to reason about inputs from "other" languages, but in a way that remains close to the semantics of their language. *Linking types* are a well-specified minimal extension of the source language that allow programmers to optionally annotate where in their programs they can link with components that would not be expressible in their unadulterated source language, giving them fine-grained control over the contexts that they must reason about and the equivalences that arise.

NOTE: This paper will be much easier to follow if viewed/printed in color.

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

When building large-scale software systems, programmers should be able to use the best language for each part of the system. Using the "best language" means the language that makes it is easiest for a programmer to *reason* about the behavior of that part of the system. That might be Rust for a high-performance component, a terminating domain-specific language for a protocol parser, or a general-purpose scripting language for UI code. But it's a myth that programmers can reason in a single language when dealing with multi-language software. Even if a high-assurance component is written in Coq, the programmer must reason about extraction, compilation, and any linking that happens at the machine-level. An ML component in a multi-language system has contexts that may include features that don't exist in ML. This is a problem, because as programmers evolve complex systems, much time is spent refactoring—that is, making changes to components that should result in equivalent behavior. Programmers reason about that equivalence by thinking about possible program contexts within which the original and refactored components could be run, though usually they only think about contexts from their own language. But if we have linked with another language, somehow the additional contexts from that language also need to be taken into account. The formal property of *contextual equivalence* is therefore central to programmer reasoning. Unfortunately, progammers cannot rely upon contextual equivalence of their own language. Instead, since languages interact after having been compiled to a common

target, in effect the contextual equivalence that programmers must rely upon is that of the compilation target, which may have little to do with their source language.

Ideally, we would like programmers to be able to reason using source-language contextual equivalence, even in the presence of target-level linking. A *fully abstract* compiler enables exactly this reasoning: it guarantees that if two components are contextually equivalent at the source their compiled versions are contextually equivalent at the target. However, this guarantee comes at a steep cost: a fully abstract compiler must disallow linking with components whose behavior is inexpressible in the compiler's source language. But often that extra behavior is exactly why the programmer is linking with a component written in another "more expressive" language—for instance, linking a high-level language with a C runtime or non-concurrent language with a thread implementation.

There are two ways in which a programming language $\mathcal{A}$ can be *more expressive* than another language $\mathcal{B}$, where, following Felleisen [10], we assume both languages have been translated to a common substrate (for us, compiled to a common target), such that $\mathcal{A}$ contexts can be wrapped around $\mathcal{B}$ program fragments:

1. $\mathcal{A}$ has features unavailable in $\mathcal{B}$ that can be used to create contexts that can distinguish components that are contextually equivalent in $\mathcal{B}$. We call this "positive" expressivity, since the (larger) set of $\mathcal{A}$ contexts have more power to distinguish. For instance, $\mathcal{A}$ may have references or first-class control while $\mathcal{B}$ does not.
2. $\mathcal{A}$ has rich type-system features unavailable in $\mathcal{B}$ that can be used to rule out contexts that, at less precise types, were able to distinguish inequivalent $\mathcal{B}$ components. We call this "negative" expressivity, since type restrictions on $\mathcal{A}$ contexts result in a (smaller) set of well-typed $\mathcal{A}$ contexts that have less power to distinguish. For instance, $\mathcal{A}$ may have linear types or polymorphism while $\mathcal{B}$ does not.

The greater expressivity of programming languages explored by Felleisen [10] is what we call "positive" expressivity. As far as we are aware, the notion of "negative" expressivity has not appeared in the literature. (If this paper is accepted, we will include examples and further discussion of "negative" expressivity and its consequences.)

Linking with code from more expressive languages affects not just *programmer* reasoning, but also the notion of equivalence used by *compiler writers* to justify correct optimizations. While there has been a lot of recent work on verified compilers, most assume no linking (e.g., [17, 18, 20, 27, 29, 15]), or linking only with code compiled from the same source language [3, 4, 13, 21, 14]. One exception to this is Compositional Compcert [28], which nonetheless only allows linking with components that satisfy CompCert's memory model. The other exception is the multi-language approach to verified compilers in Perconti-Ahmed [26], which allows linking with arbitrary target code that may be compiled from another source language $\mathcal{R}$. This approach, which embedded both the source $\mathcal{S}$ and target $\mathcal{T}$ into a single multi-language $\mathcal{ST}$, meant that compiler optimizations could be justified in terms of $\mathcal{ST}$ contextual equivalence. However, as a tool for programmer reasoning, this comes at a significant cost, as the programmer needs to understand the full $\mathcal{ST}$ language and the compiler for $\mathcal{R}$. Moreover, the design of the multi-language fixes what linking should and should not be permitted, a decision that affects the notion of contextual equivalence used to reason about every component written in the source language.

We do not believe that compiler writers should get to decide what linking is allowed. Compiler writers are forced to either ignore linking or make such arbitrary decisions because our source-language specifications are incomplete with respect to linking. We argue that, instead, this should be a part of the language specification and exposed to the programmer

so that she can make fine-grained decisions about linking, which leads to fine-grained control over what contexts she must consider when reasoning about a particular component. Every compiler should then be fully abstract, which means it preserves the equivalences chosen by the programmer.

We advocate extending source-language specifications with *linking types*, which minimally enrich source-language types and allow programmers to optionally annotate where in their programs they can link with components that wouldn't be expressible in their unadulterated source language. As a specification mechanism, types are familiar, and naturally allow us to change equivalences locally. They fulfill our desire to allow the programmer fine-grained control, as they appear on individual terms of the language. A linking-types extension will also often introduce new terms (and operational semantics) intended solely for reasoning about the additional contexts introduced through linking. These new terms are a representative abstraction of potentially complex new behavior from another language that the programmer wants to link with. (This is analogous to how Gu *et al.* [11] lift potentially complex behavior in a lower abstraction layer into a simpler representation in a higher layer.) Now if the programmer reasons about contexts including those terms, she will have considered the behavior of all contexts that her component may be linked with after compilation.

We envision that language designers will provide many different linking-types extensions for their source languages. Programmers can then opt to use zero or more of these extensions, depending on their linking needs.

2 Linking Types, Formally

To formally present the basic idea of linking types, we consider a setting with two simple source languages—see Figure 1 (top)—and show how to design linking types that mediate different interactions between them. Our source languages are λ, the simply typed lambda calculus with integer base types, and $\lambda^{\mathtt{ref}}$, which extends λ with ML-like mutable references. We want type-preserving, fully abstract compilers from these source languages to a common target language. That target should have a rich enough type system so that the compiler's type translation can ensure full abstraction by using types to rule out linking with target contexts whose behavior is inexpressible in the source. Here we illustrate the idea with a fairly high-level target language $\lambda^{\mathtt{ref}}_{\mathtt{exc}}$—see Figure 1 (bottom)—that includes mutable references and exceptions and has a modal type system that can distinguish pure computations from those that either use references or raise exceptions.[1] We include exceptions in the target as a representative of the extra control flow often present in low-level targets (e.g., direct jumps). A target computation $\mathrm{E}^{\bullet}_{\tau_{\mathtt{exc}}}\ \tau$ may access the heap while computing a value of type τ or raising an exception of type $\tau_{\mathtt{exc}}$. In contrast, a computation $\mathrm{E}^{\circ}_{0}\ \tau$ may not access the heap, and cannot raise exceptions as the exception type is the void (uninhabited) type $\mathtt{0}$.

Consider the scenario where the programmer writes code in λ and wants to link with code written in $\lambda^{\mathtt{ref}}$. Assume this linking happens after both λ and $\lambda^{\mathtt{ref}}$ have been compiled using *fully abstract* compilers to $\lambda^{\mathtt{ref}}_{\mathtt{exc}}$. Below, consider the programs $\mathtt{e}_1$ and $\mathtt{e}_2$ which are equivalent in λ. The λ compiler, since it is fully abstract, would have to disallow linking with the context $\mathtt{C}^{\mathtt{ref}}$, since the latter can distinguish $\mathtt{e}_1$ from $\mathtt{e}_2$ in the target. More generally, in order to rule out this class of equivalence-disrupting contexts the fully abstract compiler would have to prevent linking with code that has externally visible effects.[2] This can be accomplished

[1] We use a modal type system here, but any type-and-effect system would suffice.
[2] For simplicity, the type system we show here doesn't support effect masking, so we rule out linking with

$$\begin{array}{llll} \lambda & \tau & ::= & \mathtt{unit} \mid \mathtt{int} \mid \tau \to \tau \\ & \mathtt{e} & ::= & () \mid \mathtt{n} \mid \mathtt{x} \mid \lambda \mathtt{x} : \tau.\mathtt{e} \mid \mathtt{e}\,\mathtt{e} \\ & & & \mathtt{e} + \mathtt{e} \mid \mathtt{e} * \mathtt{e} \mid \mathtt{e} - \mathtt{e} \\ & \mathtt{v} & ::= & () \mid \mathtt{n} \mid \lambda \mathtt{x} : \tau.\mathtt{e} \end{array} \qquad \begin{array}{llll} \lambda^{\mathtt{ref}} & \tau & ::= & \ldots \mid \mathtt{ref}\,\tau \\ & \mathtt{e} & ::= & \ldots \mid \mathtt{ref}\,\mathtt{e} \mid \mathtt{e} := \mathtt{e} \mid !\mathtt{e} \\ & \mathtt{v} & ::= & \ldots \mid \ell \end{array}$$

$$\begin{array}{llll} \lambda^{\mathtt{ref}}_{\mathtt{exc}} & \tau & ::= & 0 \mid \mathtt{unit} \mid \mathtt{int} \mid \mathtt{ref}\,\tau \mid \tau \to \mathrm{E}^{\epsilon}_{\tau_{\mathtt{exc}}}\,\tau \\ & \epsilon & ::= & \bullet \mid \circ \\ & \mathtt{e} & ::= & () \mid \mathtt{n} \mid \mathtt{x} \mid \lambda \mathtt{x} : \tau.\mathtt{e} \mid \mathtt{e}\,\mathtt{e} \mid \mathtt{e} + \mathtt{e} \mid \mathtt{e} * \mathtt{e} \mid \mathtt{e} - \mathtt{e} \mid \mathtt{throw}\,\mathtt{e} \\ & & & \mathtt{catch}\,\mathtt{e}\,\mathtt{with}\,\mathtt{val}\,\mathtt{x} \Rightarrow \mathtt{e}\,;\,\mathtt{exc}\,\mathtt{y} \Rightarrow \mathtt{e} \mid \mathtt{ref}\,\mathtt{e} \mid \mathtt{e} := \mathtt{e} \mid !\mathtt{e} \\ & \mathtt{v} & ::= & () \mid \mathtt{n} \mid \lambda \mathtt{x} : \tau.\mathtt{e} \mid \ell \end{array}$$

$\boxed{\Gamma \vdash \mathtt{v} : \tau}$

$$\frac{}{\Gamma \vdash () : \mathtt{unit}} \qquad \frac{\Gamma, \mathtt{x} : \tau \vdash \mathtt{e} : \mathrm{E}^{\rho}_{\tau_{\mathtt{exn}}}\,\tau'}{\Gamma \vdash \lambda\,\mathtt{x} : \tau.\mathtt{e} : \tau \to \mathrm{E}^{\rho}_{\tau_{\mathtt{exn}}}\,\tau'}$$

$\boxed{\Gamma \vdash \mathtt{e} : \mathrm{E}^{\epsilon}_{\tau_{\mathtt{exn}}}\,\tau}$

$$\frac{\Gamma \vdash \mathtt{v} : \tau}{\Gamma \vdash \mathtt{v} : \mathrm{E}^{\circ}_{0}\,\tau} \qquad \frac{\Gamma \vdash \mathtt{e}_1 : \tau \to \mathrm{E}^{\rho_1}_{\tau_{\mathtt{exn}}}\,\tau' \qquad \Gamma \vdash \mathtt{e}_2 : \mathrm{E}^{\rho_2}_{\tau_{\mathtt{exn}}}\,\tau}{\Gamma \vdash \mathtt{e}_1\,\mathtt{e}_2 : \mathrm{E}^{\rho_1 \vee \rho_2}_{\tau_{\mathtt{exn}}}\,\tau'} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathrm{E}^{\rho}_{\tau_{\mathtt{exn}}}\,\tau \qquad \vdash \tau}{\Gamma \vdash \mathtt{ref}\,\mathtt{e} : \mathrm{E}^{\bullet}_{\tau_{\mathtt{exn}}}\,\mathtt{ref}\,\tau}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathrm{E}^{\rho_1}_{\tau_{\mathtt{exn}}}\,\mathtt{ref}\,\tau \qquad \Gamma \vdash \mathtt{e}_1 : \mathrm{E}^{\rho_2}_{\tau_{\mathtt{exn}}}\,\tau}{\Gamma \vdash \mathtt{e}_1 := \mathtt{e}_2 : \mathrm{E}^{\bullet}_{\tau_{\mathtt{exn}}}\,\mathtt{unit}} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathrm{E}^{\rho}_{\tau_{\mathtt{exn}}}\,\mathtt{ref}\,\tau}{\Gamma \vdash !\mathtt{e}_1 : \mathrm{E}^{\bullet}_{\tau_{\mathtt{exn}}}\,\tau}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathrm{E}^{\rho}_{\tau_{\mathtt{exn}}}\,\tau \qquad \Gamma, \mathtt{x} : \tau \vdash \mathtt{e}_2 : \mathrm{E}^{\rho_2}_{\tau'_{\mathtt{exn}}}\,\tau' \qquad \Gamma, \mathtt{y} : \tau_{\mathtt{exn}} \vdash \mathtt{e}_1 : \mathrm{E}^{\rho_1}_{\tau'_{\mathtt{exn}}}\,\tau'}{\Gamma \vdash \mathtt{catch}\,\mathtt{e}\,\mathtt{with}\,\mathtt{val}\,\mathtt{x} \Rightarrow \mathtt{e}_1\,;\,\mathtt{exc}\,\mathtt{y} \Rightarrow \mathtt{e}_2 : \mathrm{E}^{\rho_1 \vee \rho_2}_{\tau'_{\mathtt{exn}}}\,\tau'} \qquad \frac{\Gamma \vdash \mathtt{e} : \tau_{\mathtt{exn}} \qquad \vdash \tau}{\Gamma \vdash \mathtt{throw}\,\mathtt{e} : \mathrm{E}^{\circ}_{\tau_{\mathtt{exn}}}\,\tau}$$

Figure 1 λ and $\lambda^{\mathtt{ref}}$ syntax (top), $\lambda^{\mathtt{ref}}_{\mathtt{exc}}$ syntax and selected static semantics (bottom).

by a type-directed compiler that sends all λ arrows $\tau_1 \to \tau_2$ to pure computations $\tau'_1 \to \mathrm{E}^{\circ}_{0}\,\tau'_2$, where τ'_1 and τ'_2 are the translations of types τ_1 and τ_2. This would rule out linking with contexts with heap effects as in our $\mathtt{C}^{\mathtt{ref}}$ example.

$$\begin{array}{ll} \mathtt{e}_1 & = \lambda \mathtt{f}.\,\mathtt{f}() \\ \mathtt{e}_2 & = \lambda \mathtt{f}.\,\mathtt{f}();\mathtt{f}() \end{array} \qquad \forall \mathtt{C}^{\lambda}.\mathtt{C}^{\lambda}[\mathtt{e}_1] \approx_{\lambda} \mathtt{C}^{\lambda}[\mathtt{e}_2] \qquad \begin{array}{rl} \mathtt{C}^{\mathtt{ref}} = & \mathtt{let}\,\mathtt{x} = \mathtt{ref}\,0\,\mathtt{in} \\ & \mathtt{let}\,\mathtt{g}\,() = \mathtt{x} := !\mathtt{x} + 1; !\mathtt{x}\,\mathtt{in}\,[\cdot]\mathtt{g} \\ \mathtt{C}^{\mathtt{ref}}[\mathtt{e}_1] \Downarrow 1 & \mathtt{C}^{\mathtt{ref}}[\mathtt{e}_2] \Downarrow 2 \end{array}$$

Unfortunately, the above type translation would even rule out linking with a simple library that implements a counter using a reference cell. When a programmer wants to link with such a library, she is willing to lose some equivalences, at least in the particular places where the counter library is used. To enable such linking, we will now design a linking-types extension for λ that includes both an extended language λ^{κ} and functions κ^{+} and κ^{-} that relate types of λ and λ^{κ}. The λ^{κ} type system includes reference types and tracks heap effects. We need to track heap effects to be able to reason about the interaction between the pure λ code and impure code that it will be linked with. This extension is shown on the left in Figure 2. The parts of λ^{κ} that extend λ are typeset in **red**, whereas terms that originated in λ are green. λ^{κ} types τ include base types **unit** and **int**, reference types **ref** τ, and a computation type $\mathbf{R}^{\epsilon}\,\tau$, analogous to the target computation type $\mathrm{E}^{\epsilon}_{\tau_{\mathtt{exn}}}\,\tau$, but without tracking exception effects. λ^{κ} terms **e** include terms from λ, as well as terms for allocating, reading, and updating references. Note that these additional terms are intended

all effectful code. More realistic target languages, e.g., based on Koka [16], would support linking with code without externally visible effects.

$$
\begin{array}{llll}
\lambda^{\kappa} & \tau & ::= & \mathbf{unit} \mid \mathbf{int} \mid \mathbf{ref}\,\tau \mid \tau \to \mathbf{R}^{\epsilon}\,\tau \\
& \mathbf{e} & ::= & () \mid \mathtt{n} \mid \mathtt{x} \mid \lambda \mathtt{x} : \tau.\,\mathbf{e} \mid \mathbf{e}\,\mathbf{e} \mid \mathbf{e} + \mathbf{e} \\
& & & \mid \mathbf{e} * \mathbf{e} \mid \mathbf{e} - \mathbf{e} \mid \mathbf{ref}\,\mathbf{e} \mid \mathbf{e} := \mathbf{e} \mid !\mathbf{e} \\
& \mathbf{v} & ::= & () \mid \mathtt{n} \mid \lambda \mathtt{x} : \tau.\,\mathbf{e} \mid \ell \\
& \epsilon & ::= & \bullet \mid \circ
\end{array}
$$

$$
\begin{array}{lcl}
\kappa^{+}(\mathtt{unit}) & = & \mathbf{unit} \\
\kappa^{+}(\mathtt{int}) & = & \mathbf{int} \\
\kappa^{+}(\tau_1 \to \tau_2) & = & \kappa^{+}(\tau_1) \to \mathbf{R}^{\circ}\,\kappa^{+}(\tau_2)
\end{array}
\qquad
\begin{array}{lcl}
\kappa^{-}(\mathbf{unit}) & = & \mathtt{unit} \\
\kappa^{-}(\mathbf{int}) & = & \mathtt{int} \\
\kappa^{-}(\mathbf{ref}\,\tau) & = & \kappa^{-}(\tau) \\
\kappa^{-}(\tau_1 \to \mathbf{R}^{\epsilon}\,\tau_2) & = & \kappa^{-}(\tau_1) \to \kappa^{-}(\tau_2)
\end{array}
$$

$$
\begin{array}{llll}
\lambda^{\mathtt{ref}\kappa} & \tau & ::= & \mathbf{unit} \mid \mathbf{int} \mid \mathbf{ref}\,\tau \mid \tau \to \mathbf{R}^{\epsilon}\,\tau \\
& \mathbf{e} & ::= & () \mid \mathtt{n} \mid \mathtt{x} \mid \lambda \mathtt{x} : \tau.\,\mathbf{e} \mid \mathbf{e}\,\mathbf{e} \mid \mathbf{e} + \mathbf{e} \\
& & & \mid \mathbf{e} * \mathbf{e} \mid \mathbf{e} - \mathbf{e} \mid \mathtt{ref}\,\mathbf{e} \mid \mathbf{e} := \mathbf{e} \mid !\mathbf{e} \\
& \mathbf{v} & ::= & () \mid \mathtt{n} \mid \lambda \mathtt{x} : \tau.\,\mathbf{e} \mid \ell \\
& \epsilon & ::= & \bullet \mid \circ
\end{array}
$$

$$
\begin{array}{lcl}
\kappa^{+}(\mathtt{unit}) & = & \mathbf{unit} \\
\kappa^{+}(\mathtt{int}) & = & \mathbf{int} \\
\kappa^{+}(\tau_1 \to \tau_2) & = & \kappa^{+}(\tau_1) \to \mathbf{R}^{\bullet}\,\kappa^{+}(\tau_2)
\end{array}
\qquad
\begin{array}{lcl}
\kappa^{-}(\mathbf{unit}) & = & \mathtt{unit} \\
\kappa^{-}(\mathbf{int}) & = & \mathtt{int} \\
\kappa^{-}(\mathbf{ref}\,\tau) & = & \mathtt{ref}\,\kappa^{-}(\tau) \\
\kappa^{-}(\tau_1 \to \mathbf{R}^{\epsilon}\,\tau_2) & = & \kappa^{-}(\tau_1) \to \kappa^{-}(\tau_2)
\end{array}
$$

Figure 2 Linking-types extension of λ and $\lambda^{\mathtt{ref}}$.

only for reasoning, so that programmers can understand the kind of behavior that they are linking with; they should not show up in code written by the programmer. If we allowed programmers to use these terms in their code, we would be changing the programming language itself, whereas linking types should only allow a programmer to change equivalences of their existing language. Our focus is *linking*, not general language extension. The last part of the linking-types extension is the pair of functions κ^{+}, for embedding λ types in λ^{κ}, and κ^{-} for projecting λ^{κ} types to λ types. We will discuss the properties that κ^{+} and κ^{-} must satisfy below.

Also shown in Figure 2 is a linking types extension of $\lambda^{\mathtt{ref}}$ that allows $\lambda^{\mathtt{ref}}$ to distinguish program fragments that are free of heap effects and can then safely be passed to linked λ code. This results in essentially the same extended language λ^{κ}; the only changes are the arrow and reference cases of κ^{+} and κ^{-} and in terms that should be written by programmers.

We can now develop fully abstract compilers from λ^{κ} and $\lambda^{\mathtt{ref}\kappa}$—rather than λ and $\lambda^{\mathtt{ref}}$—to $\lambda^{\mathtt{ref}}_{\mathtt{exc}}$ using the following type translation to ensure full abstraction:

$$\langle\!\langle \mathbf{unit} \rangle\!\rangle = \mathtt{unit} \qquad \langle\!\langle \mathbf{int} \rangle\!\rangle = \mathtt{int} \qquad \langle\!\langle \mathbf{ref}\,\tau \rangle\!\rangle = \mathtt{ref}\,\langle\!\langle \tau \rangle\!\rangle \qquad \langle\!\langle \tau_1 \to \mathbf{R}^{\epsilon}\,\tau_2 \rangle\!\rangle = \langle\!\langle \tau_1 \rangle\!\rangle \to \mathrm{E}^{\epsilon}_{0}\,\langle\!\langle \tau_2 \rangle\!\rangle$$

Properties of Linking Types For any source language $\lambda_{\mathtt{src}}$, an extended language $\lambda^{\kappa}_{\mathtt{src}}$ paired with κ^{+} and κ^{-} is a linking-types extension if the following properties hold:

- $\lambda_{\mathtt{src}}$ terms are a subset of $\lambda^{\kappa}_{\mathtt{src}}$ terms.
- $\lambda_{\mathtt{src}}$ type τ embeds into a $\lambda^{\kappa}_{\mathtt{src}}$ type by $\kappa^{+}(\tau)$.
- $\lambda^{\kappa}_{\mathtt{src}}$ type τ^{κ} projects to a $\lambda_{\mathtt{src}}$ type by $\kappa^{-}(\tau^{\kappa})$.
- For any $\lambda_{\mathtt{src}}$ type τ, $\kappa^{-}(\kappa^{+}(\tau)) = \tau$.
- κ^{+} should be equivalence preserving: $\forall \mathtt{e}_1, \mathtt{e}_2.\ \mathtt{e}_1 \approx^{ctx}_{\lambda_{\mathtt{src}}} \mathtt{e}_2 : \tau \implies \mathtt{e}_1 \approx^{ctx}_{\lambda^{\kappa}_{\mathtt{src}}} \mathtt{e}_2 : \kappa^{+}(\tau)$.
- $\forall \mathtt{e}, \tau.\ \mathtt{e} : \tau \implies \mathtt{e} : \kappa^{-}(\tau)$ when $\mathtt{e}$ only contains $\lambda_{\mathtt{src}}$ terms.
- A compiler for $\lambda^{\kappa}_{\mathtt{src}}$ should be fully abstract, but it need only compile terms from $\lambda_{\mathtt{src}}$.

We now consider how these properties play out in the context of the linking-types extensions above. In Figure 3, we present three programs, `A`, `B`, and `C`, valid in both λ and $\lambda^{\mathtt{ref}}$. At the type $(\mathtt{int} \to \mathtt{int}) \to \mathtt{int}$, all three programs are equivalent in λ, which we illustrate by putting `A`, `B`, `C` in a single equivalence box. In λ, all functions terminate, which means that calling the argument `f` zero, one, or two times before discarding the result is equivalent. However, in $\lambda^{\mathtt{ref}}$, `A`, `B`, and `C` are all in different equivalence classes, since `f` may increment a counter, which means a context could detect the number of times it was called.

```
program A   λf : int → int. 1
program B   λf : int → int. f 0; 1
program C   λf : int → int. f 0; f 0; 1
```

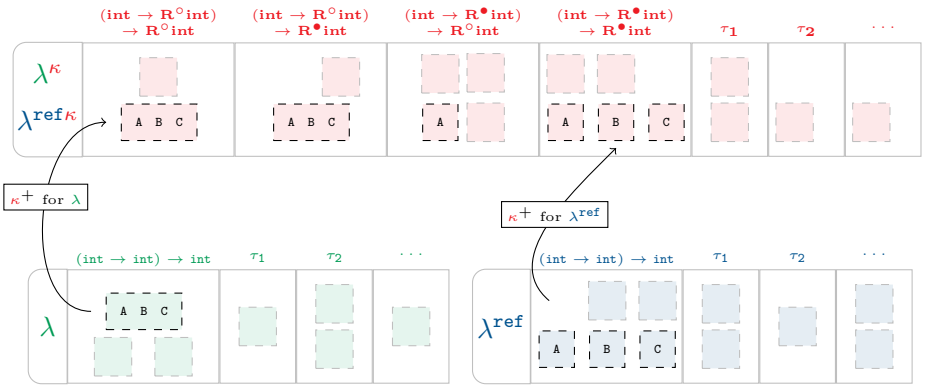

Figure 3 Equivalence classes when giving different linking types to programs.

The top of the diagram shows equivalence classes for λ^{κ}/$\lambda^{\texttt{ref}\,\kappa}$. Here we can see how equivalences can be changed by annotating these functions with different linking types. Note that equivalence is only defined at a given type, so we only consider when all three functions have been given the same linking type.

At the type $(\textbf{int} \to \textbf{R}^{\circ}\textbf{int}) \to \textbf{R}^{\circ}\textbf{int}$ these programs are all equivalent since this linking type requires that `f` be pure. At the type $(\textbf{int} \to \textbf{R}^{\bullet}\textbf{int}) \to \textbf{R}^{\bullet}\textbf{int}$ all three programs are in different equivalence classes, because the linking type allows `f` to be impure, which could be used by a context to distinguish the programs. At the type $(\textbf{int} \to \textbf{R}^{\circ}\textbf{int}) \to \textbf{R}^{\bullet}\textbf{int}$ all three programs are again equivalent. While the type allows the body to be impure, since the argument `f` is pure, no difference can be detected. The last linking type $(\textbf{int} \to \textbf{R}^{\bullet}\textbf{int}) \to \textbf{R}^{\circ}\textbf{int}$ is a type that can only be assigned to the program `A`, because if the argument `f` is impure but the result is pure the program could not have called `f`.

We can see here that κ^{+} is the "default" embedding, which has the important property that it preserves equivalence classes from the original language. Notice that κ^{+} for λ and $\lambda^{\texttt{ref}}$ both do this, but send the respective source $(\texttt{int} \to \texttt{int}) \to \texttt{int}$ to different types.

Linking Types for Advanced Languages For space reasons, we have elided discussion of four additional applications of linking types, which we briefly describe below. (If this paper is accepted, we will include a more detailed exposition with linking types extensions for $\lambda^{\texttt{ref}}$.)

- *Linearity in libraries:* By extending ML-like languages with linking types, we can use libraries implemented in a language like Rust without violating its affine invariants.
- *Terminating protocol parsers:* For certain tasks, every program should terminate. But while an HTTP protocol parser should always terminate, the server where it lives better not. Linking types can allow terminating and nonterminating languages to interact.
- *Surfacing cost of computation:* By introducing linking types for computational cost (e.g., space, time) we can facilitate linking with cost-aware languages (e.g., [5, 12, 7]) which are useful when proving security properties, though difficult for general-purpose programming. This application of linking types would allow side channels to be incorporated into the definition of observational equivalence, an issue discussed by D'Silva *et al.* [9].
- *Gradual typing:* We can add linking types to an un(i)typed language to facilitate sound linking with a statically typed language.

3 Bringing Linking Types to Your Language

To understand linking types and the way they interact with real-world languages, we consider an example of how a language designer would incorporate them and discuss their usefulness and viability (à la Cardelli [6]).

Day 1: Fully abstract compiler As a first step, the language designer implements and proves fully abstract a type-directed compiler for her language $\mathcal{A}$. She targets a typed low-level intermediate language $\mathcal{L}$, using an appropriate type translation to guarantee that equivalences are preserved. Linking should occur in $\mathcal{L}$, which means the subsequent passes need not be fully-abstract.

Discussion • Full abstraction is a key part of linking types, as it is required to preserve the equivalences that programmers rely upon for reasoning. • The representative terms added to the linking-types-extended language are used in the proof of full abstraction, which essentially requires showing that target contexts can be back-translated to equivalent source contexts. • While we use static types in our target to ensure full abstraction— and gain tooling benefits from it (explored in Day 3)—we can also use dynamic checks when appropriate (e.g. [23, 8]). • We are currently designing a language like $\mathcal{L}$, which we expect to be similar to a much more richly typed version of LLVM, such that types could be erased and existing LLVM code-generation infrastructure could be used (as discussed by Ahmed at SNAPL'15 [1]).

Day 2: Linking with more expressive code $\mathcal{A}$ programmers are happy using the above compiler since they can reason in terms of $\mathcal{A}$ semantics, even when using libraries directly implemented in $\mathcal{L}$ or compiled from other languages. But, soon the language designer's users ask to link their code with a $\mathcal{B}$ language library with features in $\mathcal{L}$ but not in $\mathcal{A}$, something that the fully abstract compiler currently prevents.

The compiler writer introduces a linking-types extension to capture the inexpressible features for her $\mathcal{A}$ programmers. She implements a type checker for the fully elaborated linking types and extends her fully abstract compiler to handle the extended $\mathcal{A}^{\kappa}$ types.

Discussion • While the linking types will in general be a new type system, no impact is seen on type inference, because linking types are never inferred: first the program will have source types inferred, and then all source types will be lifted to the linking types, using the programmer-specified annotations where present and the default κ^{+} embedding where annotations are absent.

Day 3: When can components in two languages be linked? Happily able to link with other languages, the $\mathcal{A}$ programmer uses the tooling associated with the $\mathcal{A}$ and $\mathcal{B}$ compilers to determine when a $\mathcal{B}$ component can be used at a linking point. The tool uses the compiler to translate the $\mathcal{B}$ component's type $\tau_{\mathcal{B}}$ to an $\mathcal{L}$ type $\tau_{\mathcal{L}}$ and then attempts to back-translate $\tau_{\mathcal{L}}$ to an $\mathcal{A}$ type by inverting the $\mathcal{A}$ compiler's type translation. Should this succeed, the component can be used at the type $\tau_{\mathcal{A}}$. This functionality allows the programmer to easily work on components in both $\mathcal{A}$ and $\mathcal{B}$ at once while getting *cross-language type errors* if the interfaces do not match.

Discussion • This functionality depends critically on the type-directed nature of our compilers and the presence of types in the low-level intermediate language $\mathcal{L}$, whose types become the *medium* through which we can provide useful static feedback to the programmer. • While linking these components together relies upon shared calling conventions, this is true of any linking. Currently, cross-language linking often relies upon C calling conventions.

Day 4: Backwards compatibility for programmers At the same time, another programmer continues to use $\mathcal{A}$, unaware of the $\mathcal{A}^{\kappa}$ extension, since linking types are *optional* annotations.

At lunch, she learns about linking types and realizes that the $\mathcal{C}$ language she uses could benefit from the $\mathcal{L}$ linking ecosystem. She asks the compiler writer for a $\mathcal{C}$-to-$\mathcal{L}$ compiler.

Discussion • Linking types are entirely opt-in—a programmer can use a language that has been extended with them and benefit from the compiler tool-chain without knowing anything about them. Only when she wants to link with code that could violate her source-level reasoning does she need to deal with linking types. • FFIs are usually considered "advanced material" in language documentation primarily due to the difficulty of using them safely. Since linking types enable safe cross-language linking, we hope that linking-type FFIs will not be considered an advanced topic in language documentation.

Day 5: Backwards compatibility for language designers Never a dull day for the compiler writer: she starts implementing a fully abstract compiler from $\mathcal{C}$ to $\mathcal{L}$, but realizes that $\mathcal{L}$ is not rich enough to capture the properties needed. She extends $\mathcal{L}$ to $\mathcal{L}^*$, and proves fully abstract the translation from $\mathcal{L}$ to $\mathcal{L}^*$. Since full abstraction proofs compose, this means that she immediately has a fully abstract compiler from $\mathcal{A}^\kappa$ to $\mathcal{L}^*$. She then implements a fully abstract compiler from $\mathcal{C}$ to $\mathcal{L}^*$. Programmers can then link $\mathcal{A}^\kappa$ components and $\mathcal{C}$ components provided that the former do not use $\mathcal{C}$ features that cannot be expressed in $\mathcal{A}^\kappa$. Luckily for the compiler writer, the proofs mean that the behavior of $\mathcal{A}^\kappa$, even in the presence of linking, was fully specified before and remains so. Hence, $\mathcal{A}$ and $\mathcal{A}^\kappa$ programmers need not even know about the change from $\mathcal{L}$ to $\mathcal{L}^*$.

Discussion • While implementing fully abstract compilers is nontrivial, the linking-types strategy permits a gradual evolution, not requiring redundant re-implementation and re-proof whenever changes to the language are made. • More generally, the proofs of full abstraction mean that the compiler and the target are irrelevant for programmers—behavior is entirely specified at the level of the (possibly extended by linking types) source.

4 Research Plan and Challenges

We are currently studying the use of linking types to facilitate building multi-language programs that may consist of components from the following: an idealized ML (essentially System F with references); a simple linear language; a language with first-class control; and a terminating language. We plan to develop a richly typed target based on Levy's call-by-push-value (CBPV) [19] that can support fully abstract compilation from our linking-types-extended languages. Zdancewic (personal communication on Vellvm2) has recently demonstrated a machine equivalence between a variant of CBPV and an LLVM-like SSA-based IR so this provides a path from our current intended target to a richly typed LLVM.

Realizing such a multi-language programming platform involves a number of challenges. First, implementing fully abstract compilers is nontrivial, though there has been significant recent progress by both our group and others that we expect to draw upon [2, 8, 22, 23]. Second, low-level languages such as LLVM and assembly are typically non-compositional which makes it hard to support high-level compositional reasoning. In recent work, we have designed a compositional typed assembly language that we think offers a blueprint for designing other low-level typed IRs [24, 25]. Finally, we have only begun investigating how to combine different linking-types extensions. The linking-types extensions we are considering are based on type-and-effect systems, so we believe we can create a lattice of these extensions analogous to an effect lattice.

References

1. Amal Ahmed. Verified Compilers for a Multi-Language World. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15–31, 2015.
2. Amal Ahmed and Matthias Blume. An equivalence-preserving CPS translation via multi-language semantics. In *International Conference on Functional Programming (ICFP), Tokyo, Japan*, pages 431–444, September 2011.
3. Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In *International Conference on Functional Programming (ICFP), Edinburgh, Scotland*, September 2009.
4. Nick Benton and Chung-Kil Hur. Realizability and compositional compiler correctness for a polymorphic language. Technical Report MSR-TR-2010-62, Microsoft Research, April 2010.
5. Guy E. Blelloch and Robert Harper. Cache and I/O efficient functional algorithms. In *ACM Symposium on Principles of Programming Languages (POPL), Rome, Italy*, pages 39–50, January 2013.
6. Luca Cardelli. Program fragments, linking, and modularization. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, POPL '97, pages 266–277, New York, NY, USA, 1997. ACM. URL: `http://doi.acm.org/10.1145/263699.263735`, `doi:10.1145/263699.263735`.
7. Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational cost analysis. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, January 2017.
8. Dominique Devriese, Marco Patrignani, and Frank Piessens. Fully-abstract compilation by approximate back-translation. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida*, 2016.
9. Vijay D'Silva, Mathias Payer, and Dawn Song. The correctness-security gap in compiler optmization. In *Language-theoretic Security IEEE Security and Privacy Workshop (LangSec)*, 2015.
10. Matthias Felleisen. On the expressive power of programming languages. In *Science of Computer Programming*, pages 134–151. Springer-Verlag, 1990.
11. Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 595–608, New York, NY, USA, 2015. ACM. URL: `http://doi.acm.org/10.1145/2676726.2676975`, `doi:10.1145/2676726.2676975`.
12. Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource Aware ML. In *24rd International Conference on Computer Aided Verification (CAV'12)*, volume 7358 of *Lecture Notes in Computer Science*, pages 781–786. Springer, 2012.
13. Chung-Kil Hur and Derek Dreyer. A Kripke logical relation between ML and assembly. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, January 2011.
14. Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight verification of separate compilation. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida*, pages 178–190. ACM, 2016.
15. Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML : A verified implementation of ML. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, January 2014.

16 Daan Leijen. Koka: Programming with row polymorphic effect types. In *Mathematically Structured Functional Programming, Grenoble, France*, April 2014.

17 Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*, January 2006.

18 Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

19 Paul Blain Levy. *Call-by-Push-Value*. Ph. D. dissertation, Queen Mary, University of London, London, UK, March 2001.

20 Andreas Lochbihler. Verifying a compiler for Java threads. In *European Symposium on Programming (ESOP)*, March 2010.

21 Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *International Conference on Functional Programming (ICFP), Vancouver, British Columbia, Canada*, August 2015.

22 Max S. New, William J. Bowman, and Amal Ahmed. Fully abstract compilation via universal embedding. In *International Conference on Functional Programming (ICFP), Nara, Japan*, September 2016.

23 Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. Secure compilation to protected module architectures. *ACM Transactions on Programming Languages and Systems*, 37(2):6:1–6:50, April 2015.

24 Daniel Patterson, James T. Perconti, Christos Dimoulas, and Amal Ahmed. FunTAL: Reasonably mixing a functional language with assembly. Available at `http://www.ccs.neu.edu/home/amal/papers/funtal.pdf`, November 2016.

25 Daniel Patterson, James T. Perconti, Christos Dimoulas, and Amal Ahmed. FunTAL: Reasonably mixing a functional language with assembly (technical appendix). Available at `http://www.ccs.neu.edu/home/amal/papers/funtal-tr.pdf`, November 2016.

26 James T. Perconti and Amal Ahmed. Verifying an open compiler using multi-language semantics. In *European Symposium on Programming (ESOP)*, April 2014.

27 Jaroslav Sevcik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, 2011.

28 Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional compcert. In *ACM Symposium on Principles of Programming Languages (POPL), Mumbai, India*, 2015.

29 Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Seattle, Washington*, June 2013.