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

Software developers compose systems from components written in many different languages. A business-logic component may be written in Java or OCaml, a resource-intensive component in C or Rust, and a high-assurance component in Coq. In this multi-language world, program execution sends values from one linguistic context to another. This boundary-crossing exposes values to contexts with unforeseen behavior—that is, behavior that could not arise in the source language of the value. For example, a Rust function may end up being applied in an ML context that violates the memory usage policy enforced by Rust's type system. This leads to the question of how developers ought to reason about code in such a multi-language world where behavior inexpressible in one language is easily realized in another.

This paper proposes the novel idea of *linking types* to address the problem of reasoning about single-language components in a multi-lingual setting. Specifically, linking types allow programmers to annotate where in a program they can link with components inexpressible in their unadulterated language. This enables developers to reason about (behavioral) equality using only their own language and the annotations, even though their code may be linked with code written in a language with more expressive power.

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

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Linking, program reasoning, equivalence, expressive power of languages, fully abstract compilation

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.12

1 Reasoning in a Multi-Language World

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. Moreover, programmers should be able to reason *only* in that language when working on that component. 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. In some development shops, domain-specific languages are used in various parts of systems to better separate the logic of particular problems from the plumbing of general-purpose programming. 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

© Daniel Patterson and Amal Ahmed;
licensed under Creative Commons License CC-BY

contexts that may include features that don't exist in ML. This is a problem for programmers, because as they 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 written in their own language. But if they have linked with another language, the additional contexts from that language also need to be taken into account. Equivalence in all contexts, or *contextual equivalence*, is therefore central to programmer reasoning. Unfortunately, programmers cannot rely upon contextual equivalence of their own language. Instead, since languages interact after having been compiled to a common target, the contextual equivalence that programmers must rely upon is that of the compilation target, which may have little to do with their source language.

For programmers writing components in safe languages like OCaml, the situation is made worse by the fact that the common target is likely a low-level unsafe language like assembly which permits direct access to memory and the call-stack. An object-code linker will verify symbols, but little more. This means that whenever an OCaml programmer links with C code via the FFI, they have to contend with the fact that the C code they write can easily disrupt the equivalences they rely on when reasoning about their OCaml code. Rather than being able to rely upon tooling, the user of a C library must reason carefully about how the C code will interact, at the assembly level, with their OCaml abstractions. For example, an OCaml function that is polymorphic in its arguments could have these arguments inspected by C code it linked with, violating parametricity. On the other side, the C programmer attempting to write a library that can be linked with OCaml must keep all the invariants of OCaml in mind and attempt not to violate any of them. This is difficult and requires reasoning not only about how the OCaml and C languages work but also how they are compiled to assembly, because it is at the assembly level that they interact.

Since programmers use a language for its features and linguistic abstractions, we would like programmers to be able to reason using contextual equivalence for that language, 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 or control is exactly why the programmer is linking with a component written in another "more expressive" language. An example of additional *behavior* is a non-concurrent language linking with a thread implementation written in C. An example of additional *control* is an unrestricted language linking with a concurrent data structure written in Rust, where linear types ensure data-race freedom.

There are two ways in which a programming language $\mathcal{A}$ can be *more expressive* than another language $\mathcal{B}$, where, following Felleisen [12], 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 say that language $\mathcal{A}$ is *positively* more expressive than language $\mathcal{B}$, 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 say

that language $\mathcal{A}$ is *negatively* more expressive than language $\mathcal{B}$, 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.[1]

The greater expressivity of programming languages explored by Felleisen [12] is what we call *positive* expressivity. As far as we are aware, the notion of *negative* expressivity, presented in this *dual* way, has not appeared in the literature.

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., [19, 20, 22, 29, 32, 17]), or linking only with code compiled from the same source language [4, 5, 15, 23, 16].

One approach that does support cross-language linking is Compositional Compcert [30], which nonetheless only allows linking with components that satisfy CompCert's memory model. Another approach is the multi-language style of verified compilers by Perconti and Ahmed [28], which allows linking with arbitrary target code that may be compiled from another source language $\mathcal{R}$. This approach, which embeds both the source $\mathcal{S}$ and target $\mathcal{T}$ into a single multi-language $\mathcal{ST}$, means that compiler optimizations can 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 from $\mathcal{R}$ to $\mathcal{T}$. 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 contend that compiler writers should not get to decide what linking is allowed, and indeed, we don't think they want to. Currently compiler writers are forced to either ignore linking or make such arbitrary decisions because existing source-language specifications are incomplete with respect to linking. 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 would not 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

---

[1] Example 1: $\mathcal{A}$ has **linear types**.
Consider $\mathcal{B}$ components of type $\texttt{unit} \to \texttt{int}$:

$\texttt{c1}$ increments a counter on each call and returns it
$\texttt{c2}$ increments a counter on first call and returns it

And $\mathcal{B}$ distinguishing context:

$$\lambda \texttt{c}.\, \texttt{c}\,(); \texttt{c}\,() \quad : (\texttt{unit} \to \texttt{int}) \to \texttt{int}$$

With $\mathcal{A}$ contexts of type $(\texttt{unit} \to \texttt{int})^{\texttt{L}} \to \texttt{int}$, $\texttt{c1}$ and $\texttt{c2}$ are contextually equivalent, since each must be called exactly once.

Example 2: $\mathcal{A}$ has **polymorphism**.
Consider $\mathcal{B}$ components of type $\texttt{unit} \to \texttt{int}$:

$\texttt{p1} = \lambda \texttt{x}.\, 0$
$\texttt{p1} = \lambda \texttt{x}.\, 1$

And $\mathcal{B}$ distinguishing context:

$$\lambda \texttt{f}.\, \texttt{if}\ \texttt{f}()\texttt{=0}\ \texttt{then}\ \texttt{diverge}\ \texttt{else}\ () \quad : (\texttt{unit} \to \texttt{int}) \to \texttt{unit}$$

With $\mathcal{A}$ contexts of type $\forall\alpha.(\texttt{unit} \to \alpha) \to \texttt{unit}$, $\texttt{p1}$ and $\texttt{p2}$ are contextually equivalent, since their return values may not be inspected.

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.* [13] 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 λ^{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.[2] We include exceptions in the target as a representative of the extra control flow often present in low-level targets (e.g., direct jumps). An impure target computation $\mathrm{E}^{\bullet}_{\tau_{\mathtt{exc}}}\,\tau$ (pronounced "impure exception-raising tau computation") may access the heap while computing a value of type τ or raising an exception of type $\tau_{\mathtt{exc}}$. In contrast, a pure computation $\mathrm{E}^{\circ}_{0}\,\tau$ (pronounced "pure tau computation") may not access the heap, and cannot raise exceptions as the exception type is the void (uninhabited) type 0.

Consider the scenario where the programmer writes code in λ and wants to link with code written in λ^{ref}. Assume this linking happens after both λ and λ^{ref} have been compiled using *fully abstract* compilers to $\lambda^{\mathtt{ref}}_{\mathtt{exc}}$. We illustrate this with concrete example programs $\mathbf{e_1}$ and $\mathbf{e_2}$ which are equivalent in λ. Now consider the context $\mathsf{C}^{\mathsf{ref}}$ which implements a simple counter using a reference cell. The λ compiler, since it is fully abstract, would have to disallow linking with $\mathsf{C}^{\mathsf{ref}}$ since it can distinguish $\mathbf{e_1}$ from $\mathbf{e_2}$. More generally, in order to rule out this class of equivalence-disrupting contexts, the fully abstract compiler would have to prevent linking with any code that has externally visible effects.[3] This can be accomplished by a type-directed compiler that sends all λ arrows $\tau_1 \to \tau_2$ to pure functions $\tau_1' \to \mathrm{E}^{\circ}_{0}\,\tau_2'$, 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 like $\mathsf{C}^{\mathsf{ref}}$. But in this case, the programmer wants to link these together and is willing to lose some equivalences in order to do so.

$$\mathbf{e_1} = \lambda \mathbf{c}.\,\mathbf{c}() \qquad\qquad \mathsf{C}^{\mathsf{ref}} = \mathsf{let\ x = ref\ 0\ in}$$
$$\mathbf{e_2} = \lambda \mathbf{c}.\,\mathbf{c}();\mathbf{c}() \qquad\qquad \mathsf{let\ c'\ () = x := !x + 1; !x\ in\ [\cdot]c'}$$
$$\mathsf{C}^{\mathsf{ref}}[\mathbf{e_1}] \Downarrow 1$$
$$\forall \mathbf{C}^{\lambda}.\mathbf{C}^{\lambda}[\mathbf{e_1}] \approx_{\lambda} \mathbf{C}^{\lambda}[\mathbf{e_2}] \qquad\qquad \mathsf{C}^{\mathsf{ref}}[\mathbf{e_2}] \Downarrow 2$$

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

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

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

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

$$\boxed{\Gamma \vdash \mathtt{e} : \mathrm{E}^{\epsilon}_{\tau_{\mathtt{exn}}}\,\tau} \qquad \frac{\Gamma \vdash \mathtt{v} : \tau}{\Gamma \vdash \mathtt{v} : \mathrm{E}^{\circ}_{0}\,\tau} \qquad \frac{\Gamma \vdash \mathtt{e}_1 : \mathrm{E}^{\rho_1}_{\tau_{\mathtt{exn}}}(\tau \to \mathrm{E}^{\rho_3}_{\tau_{\mathtt{exn}}}\,\tau') \qquad \Gamma \vdash \mathtt{e}_2 : \mathrm{E}^{\rho_2}_{\tau_{\mathtt{exn}}}\,\tau}{\Gamma \vdash \mathtt{e}_1\,\mathtt{e}_2 : \mathrm{E}^{\rho_1 \vee \rho_2 \vee \rho_3}_{\tau_{\mathtt{exn}}}\,\tau'} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathrm{E}^{\rho}_{\tau_{\mathtt{exn}}}\,\tau \qquad \vdash \tau}{\Gamma \vdash \mathtt{ref}\,\mathtt{e} : \mathrm{E}^{\bullet}_{\tau_{\mathtt{exn}}}\,\mathtt{ref}\,\tau}$$

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

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

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

To enable the above linking, we present 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 λ^{ref} 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 **magenta**, whereas terms that originated in λ are **orange**. 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.

With this extension, we annotate $\mathbf{e_1}$ and $\mathbf{e_2}$ with a linking type that specifies that the input can be heap-effecting: $\lambda \mathbf{c}.\,\mathbf{c}() \not\approx^{ctx}_{\lambda^{\kappa}} \lambda \mathbf{c}.\,\mathbf{c}();\mathbf{c}() : (\mathbf{unit} \to \mathbf{R}^{\bullet}\,\mathbf{int}) \to \mathbf{R}^{\bullet}\,\mathbf{int}$. At this type, $\mathbf{e_1}$ and $\mathbf{e_2}$ are no longer contextually equivalent and, further, can be linked with the counter library.

Without the above annotation, the compiler would translate the type of $\lambda \mathbf{c}.\,\mathbf{c}()$ or $\lambda \mathbf{c}.\,\mathbf{c}();\mathbf{c}()$ from the λ type $\mathbf{unit} \to \mathbf{int}$ to the $\lambda^{\mathtt{ref}}_{\mathtt{exn}}$ type $\mathtt{unit} \to \mathrm{E}^{\circ}_{0}\,\mathtt{int}$, and the type expected by the counter from the λ^{ref} type $\mathsf{unit} \to \mathsf{int}$ to the $\lambda^{\mathtt{ref}}_{\mathtt{exn}}$ type $\mathtt{unit} \to \mathrm{E}^{\bullet}_{0}\,\mathtt{int}$. Since these are not the same, an error would be reported: that $\mathbf{unit} \to \mathbf{int}$ is not compatible with $\mathsf{unit} \to \mathsf{int}$. This error matches our intuition — that an arrow means something fundamentally different in a pure language and one that has heap effects. For advanced users, the compiler could

---

all effectful code. More realistic target languages, e.g., based on Koka [18], would support linking with code without externally visible effects.

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

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

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

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

Figure 2 Linking-types extension of λ and λ^{ref}.

explain the type translations that gave rise to that incompatibility. By contrast, with the type annotation $\mathbf{unit} \to \mathbf{R}^{\bullet}\,\mathbf{int}$ both types translate to the same $\lambda^{\mathtt{ref}}_{\mathtt{exn}}$ type $\mathtt{unit} \to \mathtt{E}^{\bullet}_{0}\mathtt{int}$ and thus no error will be raised.

With the linking-types-extended language, note that the additional *terms* are intended 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 λ^{ref} that allows λ^{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 $\lambda^{\mathsf{ref}\kappa}$—rather than λ and λ^{ref}—to $\lambda^{\mathtt{ref}}_{\mathtt{exc}}$ using the following type translation to ensure full abstraction:

$$\begin{array}{ll}
\langle\!\langle \mathbf{unit} \rangle\!\rangle & = \mathtt{unit} \\
\langle\!\langle \mathbf{int} \rangle\!\rangle & = \mathtt{int} \\
\langle\!\langle \mathbf{ref}\,\tau \rangle\!\rangle & = \mathtt{ref}\,\langle\!\langle \tau \rangle\!\rangle \\
\langle\!\langle \tau_1 \to \mathbf{R}^{\epsilon}\,\tau_2 \rangle\!\rangle & = \langle\!\langle \tau_1 \rangle\!\rangle \to \mathtt{E}^{\epsilon}_{0}\,\langle\!\langle \tau_2 \rangle\!\rangle
\end{array}$$

2.1 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 $\kappa^{-}(\tau^{\kappa})$.

$$
\begin{array}{ll}
\texttt{program A} & \lambda \texttt{f} : \texttt{int} \to \texttt{int}.\, 1 \\
\texttt{program B} & \lambda \texttt{f} : \texttt{int} \to \texttt{int}.\, \texttt{f}\, 0;\ 1 \\
\texttt{program C} & \lambda \texttt{f} : \texttt{int} \to \texttt{int}.\, \texttt{f}\, 0;\ \texttt{f}\, 0;\ 1
\end{array}
$$

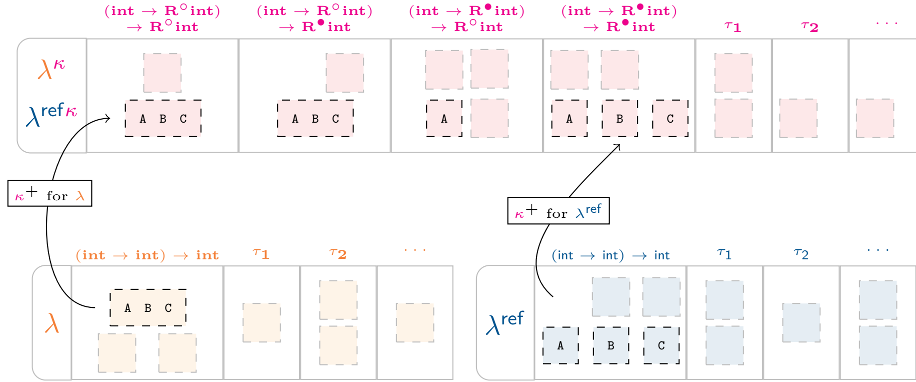

Figure 3 Equivalence classes when giving different linking types to programs.

- For any $\lambda_{\texttt{src}}$ type τ, $\kappa^{-}(\kappa^{+}(\tau)) = \tau$.
- κ^{+} preserves and reflects equivalence:
  $\forall \texttt{e}_1, \texttt{e}_2 \in \lambda_{\texttt{src}}.\ \texttt{e}_1 \approx^{ctx}_{\lambda_{\texttt{src}}} \texttt{e}_2 : \tau \iff \texttt{e}_1 \approx^{ctx}_{\lambda^{\kappa}_{\texttt{src}}} \texttt{e}_2 : \kappa^{+}(\tau)$.
- $\forall \texttt{e}, \tau.\ \texttt{e} : \tau \implies \texttt{e} : \kappa^{-}(\tau)$ when $\texttt{e}$ only contains $\lambda_{\texttt{src}}$ terms.
- A compiler for $\lambda^{\kappa}_{\texttt{src}}$ should be fully abstract, but it need only compile terms from $\lambda_{\texttt{src}}$.

Reasoning about contextual equivalence means reasoning about the equivalence classes that contain programs. Thus we can understand the effect of linking types, and of the properties that guide them, by studying how the extensions affect equivalence classes. In Figure 3, we present three programs (A, B, and C) valid in both λ and λ^{ref}. At the type $(\mathbf{int} \to \mathbf{int}) \to \mathbf{int}$, all three programs are equivalent in λ, which we illustrate by putting $\texttt{A}, \texttt{B}, \texttt{C}$ 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 λ^{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.

The top of the diagram shows equivalence classes for $\lambda^{\kappa}/\lambda^{\mathsf{ref}\kappa}$. 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 $(\mathbf{int} \to \mathbf{R}^{\circ}\mathbf{int}) \to \mathbf{R}^{\circ}\mathbf{int}$ these programs are all equivalent since this linking type requires that f be pure. At the type $(\mathbf{int} \to \mathbf{R}^{\bullet}\mathbf{int}) \to \mathbf{R}^{\bullet}\mathbf{int}$ 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 $(\mathbf{int} \to \mathbf{R}^{\circ}\mathbf{int}) \to \mathbf{R}^{\bullet}\mathbf{int}$ 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 $(\mathbf{int} \to \mathbf{R}^{\bullet}\mathbf{int}) \to \mathbf{R}^{\circ}\mathbf{int}$ 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 λ^{ref} both do this, and send the respective source $(\texttt{int} \to \texttt{int}) \to \texttt{int}$ to different types.

3 Additional Applications of Linking Types

This section contains examples of languages we would like to be able to link with λ^{ref} but which contain features that require we either rule out such linking or give up on programmers being able to reason in their source language (without linking types). We consider idealized languages here—and indeed, we believe that programmers would benefit from smaller, more special-purpose languages in a software project—but the ideas carry through to full languages with different expressivity.

3.1 Linearity in Libraries

Substructural type systems are particularly useful for modeling resources and for reasoning about where a resource must be used or when consuming a resource should render it unusable to others. Simple examples include network sockets and file handles, where opening creates the resource, reading consumes the resource and possibly creates a new one, and closing consumes the resource. An ML programmer may want to use libraries written in a linear or affine language (such as Rust) to ensure safe resource handling. But if the language with linear or affine types allows values to cross the linking boundary, ML needs to respect the linear or affine invariants to ensure soundness. For instance, if an ML component passes a value as affine to a Rust component but retains a pointer to the value and later tries to use it after it was consumed (in Rust), it violates the affine invariant that every resource may be used at most once, making the program crash. Similarly, if an ML component never consumes a linear value, it violates the linear invariant that every resouce must be used exactly once, resulting in a resource leak.

A fully abstract compiler would prevent linking in the above scenarios since two components that are equivalent in a linear/affine language can easily be distinguished by a context that does not respect linear/affine invariants. For instance, an affine function that consumes its affine input and one that does not are equivalent if the context cannot later try to consume the same input.

We can use linking types to give non-linear languages access to libraries with linear APIs. Specifically, we would extend the types of our non-linear source language λ^{ref} as follows:

$$
\begin{array}{lcl}
\phi & ::= & \mathbf{unit} \mid \mathbf{int} \mid \mathbf{ref}\,\tau \mid \tau \to \tau \\
\tau & ::= & \phi \mid \phi^{\mathbf{L}}
\end{array}
$$

$$
\begin{array}{lcl}
\kappa^{+}(\mathsf{unit}) & = & \mathbf{unit} \\
\kappa^{+}(\mathsf{int}) & = & \mathbf{int} \\
\kappa^{+}(\mathsf{ref}\,\tau) & = & \mathbf{ref}\,\kappa^{+}(\tau) \\
\kappa^{+}(\tau_1 \to \tau_2) & = & \kappa^{+}(\tau_1) \to \kappa^{+}(\tau_2)
\end{array}
\qquad
\begin{array}{lcl}
\kappa^{-}(\phi) & = & \kappa^{\mathbf{L}-}(\phi) \\
\kappa^{-}(\phi^{\mathbf{L}}) & = & \kappa^{\mathbf{L}-}(\phi) \\
\kappa^{\mathbf{L}-}(\mathbf{unit}) & = & \mathsf{unit} \\
\kappa^{\mathbf{L}-}(\mathbf{int}) & = & \mathsf{int} \\
\kappa^{\mathbf{L}-}(\mathbf{ref}\,\tau) & = & \mathsf{ref}\,\kappa^{-}(\tau) \\
\kappa^{\mathbf{L}-}(\tau_1 \to \tau_2) & = & \kappa^{-}(\tau_1) \to \kappa^{-}(\tau_2)
\end{array}
$$

Note that the target of compilation would either need to support linear types or enforce linearity at runtime—e.g., via contracts à la Tov and Pucella [31].

3.2 Terminating Protocol Parsers

For certain programming tasks, every program should terminate—for instance, HTTP protocol parsing should never end up in an infinite loop. A programmer could implement such tasks using a special-purpose language in which divergence is impossible. We still, however, need to link such terminating languages with general-purpose languages—while the protocol parser should always terminate, the server where it lives better not!

A fully abstract compiler would have to prevent such linking, since two components that are equivalent in a terminating language can easily be distinguished by a context with

nontermination. For instance, a function that calls its argument and discards the result, and one that ignores its argument are equivalent if the context provides only terminating functions as arguments, but not if the context provides a function that diverges when called.

We can use linking types to allow terminating and nonterminating languages to interact. Concretely, we can extend the types of our nonterminating language λ^{ref} with a terminating function type, written $\tau \to \tau\downharpoonleft$. The extension is as follows, but we elide cases of κ^+ and κ^- that are the same as in Figure 2:

$$\tau ::= \textbf{unit} \mid \textbf{int} \mid \textbf{ref}\,\tau \mid \tau \to \tau\downharpoonleft \mid \tau \to \tau$$

$$\kappa^+(\tau_1 \to \tau_2) = \kappa^+(\tau_1) \to \kappa^+(\tau_2)$$

$$\kappa^-(\tau_1 \to \tau_2\downharpoonleft) = \kappa^-(\tau_1) \to \kappa^-(\tau_2)$$
$$\kappa^-(\tau_1 \to \tau_2) = \kappa^-(\tau_1) \to \kappa^-(\tau_2)$$

The typing rules (elided) would likely need to rely on some syntactic termination check for functions ascribed the terminating arrow type. We could also imagine making the terminating arrow rely on a runtime timeout. The latter would require a new application typing rule to reflect that sometimes applying a terminating function might return a nonce value indicating that computation was cut off, and our language would need to be trivially extended with sum types to handle that possibility in programs.

3.3 Surfacing Cost of Computation

Some security vulnerabilities rely on the fact that the cost of a computation may be discernable (e.g., by observing time, or CPU or memory consumption). To prove the absence of such vulnerabilities, we could remove the mechanism of observation—but this is likely impossible, since even if we remove timing from our language, if the program communicates over the network timing can happen on other systems. A more promising strategy is to introduce the notion of cost (time or space) into the model and then prove that various branches are indistinguishable in that model (see, e.g. [6], [14], [9]). Nonetheless, one would not want to have to write non-security-sensitive parts of programs in one of these cost-aware languages. This motivates a linking-types extension of a non-cost-aware language—in this case again our idealized λ^{ref}—with a notion of computations with cost. As before, we only show differences from Figure 2:

$$\tau ::= \textbf{unit} \mid \textbf{int} \mid \textbf{ref}\,\tau \mid \tau \to \mathbf{C}^{\bullet}\tau \mid \tau \to \mathbf{C}^{\mathbf{N}}\tau$$

$$\kappa^+(\tau_1 \to \tau_2) = \kappa^+(\tau_1) \to \mathbf{C}^{\bullet}\kappa^+(\tau_2)$$

$$\kappa^-(\tau_1 \to \mathbf{C}^{\bullet}\tau_2) = \kappa^-(\tau_1) \to \kappa^-(\tau_2)$$
$$\kappa^-(\tau_1 \to \mathbf{C}^{\mathbf{N}}\tau_2) = \kappa^-(\tau_1) \to \kappa^-(\tau_2)$$

The type system is modal—computations $\mathbf{C}^{\mathbf{N}}\tau$ have a known cost $\mathbf{N}$, and computations $\mathbf{C}^{\bullet}\tau$ have an unknown cost. Fully abstract compilation from a cost-aware language and the above extended language would only allow known-cost computations to be passed to the cost-aware language. As before, this relies upon the target language supporting a type system that is at least as expressive, such that it can safely separate the known-cost and unknown-cost modalities.

This application of linking types echos the work by D'Silva *et al.* [11], which discusses enriching the model in which properties are investigated to encompass side channels like timing. While their work investigates machine models, ours relies upon type systems in the language where linking takes place. Further, D'Silva *et al.* envision programmers would *opt in* to security properties via annotations that would change how the compiler treated a piece of code, whereas we envision that the compiler would preserve source equivalences by default and programmers would have to *opt out* of the default fully abstract compilation by using linking types. We believe that our approach can be used with other side channels

as well, provided sufficient mechanisms exist to distinguish computations that might reveal information from those that cannot.

3.4 Gradual Typing

As we have already shown, linking types are useful when linking more precisely and less precisely typed languages. Taken to an extreme, we can add linking types to a un(i)typed language to facilitate sound linking with a statically typed language. We can do this by starting with a language with a single type, the dynamic type, and then constructing an extension that adds further types. A typed target language would then allow code compiled from a different, typed, source language to be linked with this gradually typed language. A fully abstract compiler for the extended language would have to make use of run-time checks at the boundaries between typed and untyped code, analogous to sound gradual typing.

4 Bringing Linking Types to Your Language

To understand linking types and the way they interact with existing languages, we consider an example of how a language designer would incorporate them and discuss their usefulness and viability (à la Cardelli [8]).

Day 1: Fully abstract compiler As a first step, the language designer implements and proves fully abstract a type-directed compiler for her language $\mathcal{A}$. To make it more concrete, you can consider $\mathcal{A}$ to be the language λ from earlier in the paper, but this scenario is general — you could equally consider $\mathcal{A}$ to be a language like OCaml. The compiler targets a typed low-level intermediate language $\mathcal{L}$, using an appropriate type translation to guarantee that equivalences are preserved. This, concretely, could be a target like $\lambda^{\mathtt{ref}}_{\mathtt{exn}}$, but could also be a richly-typed version of LLVM. All linking should occur in $\mathcal{L}$, which means the subsequent passes, to LLVM, assembly, or another target, 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. [25, 10]). • 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. In the example used earlier in the paper, $\mathcal{B}$ would be λ^{ref}, and the additional feature would be mutable references, but again, this is a general process that could apply to other features.

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 $\tau_{\mathcal{A}}$ 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. In the example used earlier in the paper, such a type error showed up when trying to link the λ^{ref} counter library with the λ client that had not been annotated.

Discussion • This functionality depends critically on the type-directed nature of our compilers and the presence of types in the low-level intermediate language $\mathcal{L}$, where the 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 introduced in Day 2, 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 such an advanced topic.

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 target 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.

5 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) [21] 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.

One critical aspect of such a type system is that it should be able to identify when a component is free of a given effect, even though the component may use that effect internally. For instance, a component that throws exceptions internally but handles them all should be assigned an exception-free type. We expect to draw inspiration from the effect-masking in the Koka language [18], where mutable references that never escape do not cause a computation to be marked as effectful.

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, 3, 10, 7, 24, 25]. 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 [26, 27]. 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.

6 Conclusion

Large software systems are written using combinations of many languages. But while some languages provide powerful tools for reasoning *in* the language, none support reasoning *across* multiple languages. Indeed, the abstractions that languages purport to present do not actually cohere because they do not allow the programmer to reason solely about the code she writes. Instead, the programmer is forced to think about the details of particular compilers and low-level implementations, and to reason about the target code that her compiler generates.

With *linking types*, we propose that language designers incorporate linking into their language designs and provide programmers a means to specify linking with behavior and types inexpressible in their language. There are many challenges in how to design linking types, depending on what features exist in the languages, but only through accepting this challenge can we reach what has long been promised—an ecosystem of languages, each suited to a particular task yet stitched together seamlessly into a single large software project.

Acknowledgements The authors are grateful to Matthias Felleisen for valuable discussion and feedback on linking types. This research is supported in part by the NSF (grants CCF-1453796 and CCF-1422133) and a Google Faculty Research Award.

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. Typed closure conversion preserves observational equivalence. In *International Conference on Functional Programming (ICFP), Victoria, British Columbia, Canada*, pages 157–168, September 2008.
3. 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.
4. Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In *International Conference on Functional Programming (ICFP), Edinburgh, Scotland*, September 2009.
5. 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.
6. 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.
7. William J. Bowman and Amal Ahmed. Noninterference for free. In *International Conference on Functional Programming (ICFP), Vancouver, British Columbia, Canada*, September 2015.
8. Luca Cardelli. Program fragments, linking, and modularization. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 266–277, January 1997.
9. 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.
10. 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.
11. 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.
12. Matthias Felleisen. On the expressive power of programming languages. In *Science of Computer Programming*, pages 134–151. Springer-Verlag, 1990.
13. 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 *ACM Symposium on Principles of Programming Languages (POPL), Mumbai, India*, pages 595–608, January 2015.
14. 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.
15. 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.
16. 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.

17 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.

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

19 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.

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

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

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

23 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.

24 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.

25 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.

26 Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. FunTAL: Reasonably mixing a functional language with assembly. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Barcelona, Spain*, June 2017. To appear. Available at `http://www.ccs.neu.edu/home/amal/papers/funtal.pdf`.

27 Daniel Patterson, Jamie 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`, April 2017.

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

29 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.

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

31 Jesse Tov and Riccardo Pucella. Stateful contracts for affine types. In *European Symposium on Programming (ESOP)*, March 2010.

32 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.