Linking Types: Specifying Safe Interoperability and Equivalences

Daniel Patterson
Northeastern University
dbp@ccs.neu.edu

POPL 2017 *Student Research Competition* Extended Abstract

Note: This abstract will be easier to read if printed in color.

Introduction All programs written in high-level languages link with libraries written in lower-level languages, often to expose constructs, like threads, random numbers, or automatic serialization, that aren't possible in the high-level language. This linking usually takes place after compiling both languages to a common language, possibly assembly. In this sense, reasoning about cross-language linking means reasoning about compilation.

While most languages include cross-language linking (FFI) mechanisms, they are ad-hoc and can easily break the semantic equivalences of the source language, making it hard for source programmers to reason about correctness of their programs and hard for compiler writers to reason about correctness of their optimizations.

In this work, I design and motivate **linking types, a language-based mechanism for formally specifying safe linking with libraries utilizing features inexpressible in the source**. Linking types allows programmers to reason about their programs in the presence of behavior inexpressible in their language, without dealing with the intricacies of either the compiler or the particular language they are linking with.

Fully Abstract Compilation A key aspect of safe linking is fully abstract compilation, where any two components that are indistinguishable in the source language are indistinguishable in the target. This is accomplished by using static typing to rule out linking with bad contexts (e.g. [1], [2]) or by dynamic assertions to prevent contexts from breaking equivalences (e.g. [3]).

Fully abstract compilation enables equational reasoning and allows for safe optimizations, but only for libraries that extensionally behave like source libraries, which alone is too restrictive. For example, a fully abstract compiler from an exception-less language could link with libraries using exceptions internally, but could not link with a library implementing exceptions, as exceptions crossing the linking boundary violate full abstraction (see Figure 1).

Linking types are a **minimal extension of source language types with the desired but inexpressible behavior**. Fully abstract compilers then allow source programs to be linked with any library that is extensionally expressible in the linking-type extended language, while allowing source programmers to reason solely in terms of that extended language.

$$
\begin{array}{ll}
\mathtt{e}_1 & = \lambda\mathtt{f}.\lambda\mathtt{g}.\ \mathtt{f}\ 1; \mathtt{g}\ 2; 3 \\
\mathtt{e}_2 & = \lambda\mathtt{f}.\lambda\mathtt{g}.\ \mathtt{g}\ 1; \mathtt{f}\ 2; 3 \\
\mathtt{C}_{\mathtt{exc}}([\cdot]) & = [\cdot](\lambda\mathtt{x}.\ \mathtt{throw}\ \mathtt{x})(\lambda\mathtt{y}.\mathtt{y}) \\
 & \\
\mathtt{C}_{\mathtt{exc}}[\mathtt{e}_1] & \rightarrow \mathtt{throw}\ 1 \\
\mathtt{C}_{\mathtt{exc}}[\mathtt{e}_2] & \rightarrow \mathtt{throw}\ 2
\end{array}
$$

Figure 1: Exceptions violating full abstraction

Languages of Study While this work will eventually consider more complex source and target languages, initially I study three simple languages, with syntax in Figure 2 and selected static semantics in Figure 3. There are two source languages: the source programmer's language λ and the library writer's $\lambda^{\mathtt{ref}}$, which includes mutable references. Linking will occur after compilation to a target language $\lambda^{\mathtt{ref}}_{\mathtt{exc}}$, which also has exceptions, and is enriched with an effect type system with an exception type and a store-using marker (cf. a single static region in the effect system in [4]). The three languages share syntactic forms, which in more realistic settings corresponds to having the same calling convention, memory layout, etc.

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

Figure 2: Syntax for λ, $\lambda^{\mathtt{ref}}$, and $\lambda^{\mathtt{ref}}_{\mathtt{exc}}$.

Linking Types Specification The linking-type specification κ for λ, shown in Figure 4, includes the same

$$\frac{}{\Gamma \vdash (): \mathtt{E}_0^{\circ}\,\mathtt{unit}} \qquad \frac{\Gamma, \mathtt{x}\!:\tau \vdash \mathtt{e}: \mathtt{E}^{\rho}_{\tau_{\mathtt{exn}}}\,\tau'}{\Gamma \vdash \lambda\,\mathtt{x}\!:\tau.\mathtt{e}: \tau \to \mathtt{E}^{\rho}_{\tau_{\mathtt{exn}}}\,\tau'}$$

$$\frac{\Gamma \vdash \mathtt{e}_1\!: \tau \to \mathtt{E}^{\rho_1}_{\tau_{\mathtt{exn}}}\,\tau' \qquad \Gamma \vdash \mathtt{e}_2\!: \mathtt{E}^{\rho_2}_{\tau_{\mathtt{exn}}}\,\tau}{\Gamma \vdash \mathtt{e}_1\;\mathtt{e}_2\!: \mathtt{E}^{\rho_1 \vee \rho_2}_{\tau_{\mathtt{exn}}}\,\tau'}$$

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

$$\frac{\Gamma \vdash \mathtt{e}: \tau_{\mathtt{exn}} \qquad \vdash \tau}{\Gamma \vdash \mathtt{throw\,e}: \mathtt{E}^{\circ}_{\tau_{\mathtt{exn}}}\;\tau} \qquad \frac{\Gamma \vdash \mathtt{e}: \mathtt{E}^{\rho}_{\tau_{\mathtt{exn}}}\;\tau \qquad \vdash \tau}{\Gamma \vdash \mathtt{ref\,e}: \mathtt{E}^{\bullet}_{\tau_{\mathtt{exn}}}\;\mathtt{ref}\,\tau}$$

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

Figure 3: Selected static semantics for $\lambda^{\mathtt{ref}}_{\mathtt{exc}}$.

simplified reference effect typing as in $\lambda^{\mathtt{ref}}_{\mathtt{exc}}$, where a τ-producing computation $\mathbf{R}^{\bullet}\tau$ may mutate references whereas a computation $\mathbf{R}^{\circ}\tau$ may not.

$$\begin{array}{lrcl}
\lambda^{\kappa} & \tau & ::= & \mathbf{unit} \mid \mathbf{int} \mid \mathbf{ref}\;\tau \mid \tau \to \mathbf{R}^{\rho}\;\tau \\
 & \mathbf{e} & ::= & () \mid \mathbf{n} \mid \mathtt{x} \mid \lambda\mathtt{x}:\tau.\,\mathbf{e} \mid \mathbf{e}\,\mathbf{e} \mid \mathbf{e} + \mathbf{e} \\
 & & & \mathbf{e} * \mathbf{e} \mid \mathbf{e} - \mathbf{e} \mid \mathbf{ref}\,\mathbf{e} \mid \mathbf{e} := \mathbf{e} \mid !\mathbf{e} \\
 & \mathbf{v} & ::= & () \mid \mathbf{n} \mid \lambda\mathtt{x}:\tau.\,\mathbf{e} \mid \ell \\
\kappa^{+}(\mathtt{unit}) & & ::= & \mathbf{unit} \\
\kappa^{+}(\mathtt{int}) & & ::= & \mathbf{int} \\
\kappa^{+}(\tau_1 \to \tau_2) & & ::= & \kappa^{+}(\tau_1){\to}\;\mathbf{R}^{\circ}\;\kappa^{+}(\tau_2) \\
\kappa^{-}(\mathbf{unit}) & & ::= & \mathtt{unit} \\
\kappa^{-}(\mathbf{int}) & & ::= & \mathtt{int} \\
\kappa^{-}(\mathbf{ref}\;\tau) & & ::= & \kappa^{-}(\tau) \\
\kappa^{-}(\tau_1 \to \mathbf{R}^{\rho}\;\tau_2) & & ::= & \kappa^{-}(\tau_1){\to}\kappa^{-}(\tau_2)
\end{array}$$

Figure 4: Linking types specification κ on λ.

λ^{κ} types need only be related to λ types via embedding κ^{+} and projection κ^{-}, such that embed followed by project yields the original type. The terms of λ^{κ}, which are only used for the proof of fully abstract compilation, must include λ terms, such that a λ program can be transformed into a λ^{κ} program by replacing τ with $\kappa^{+}(\tau)$.

Note, in particular, that the types of λ^{κ} are not the union of types from λ and $\lambda^{\mathtt{ref}}$, as neither track effects. This is expected, as λ and $\lambda^{\mathtt{ref}}$ are inputs to the linking-types design, so in general they may not contain rich enough types to describe the aspects relevant to linking.

Using λ^{κ} for Linking In Figure 5, we have a function $\mathtt{fun}$ annotated with κ-linking types which is compiled (the compiler is ellided, but its type translation is shown in Figure 6) to $[\![\mathtt{fun}]\!]$ and linked against library $\mathtt{lib}$ written in $\lambda^{\mathtt{ref}}$ and compiled to $[\![\mathtt{lib}]\!]$ (which in this case happens to be syntactically identical to $\mathtt{lib}$), after which they are linked. Fully abstract compilers from λ^{κ} to $\lambda^{\mathtt{ref}}_{\mathtt{exc}}$ guarantee that all of the interactions of $[\![\mathtt{lib}]\!]$ with $[\![\mathtt{fun}]\!]$ can be explained in terms of λ^{κ} behavior.

$$\begin{array}{lcl}
\mathtt{fun} & = & \lambda\mathtt{mk}:\mathtt{int} \to \mathbf{R}^{\bullet}\,\mathbf{ref}\,\mathbf{int}. \\
 & & \lambda\mathtt{set}:\mathbf{ref}\;\mathbf{int} \to \mathbf{R}^{\bullet}\;\mathbf{int} \to \mathbf{R}^{\bullet}\;\mathbf{unit}. \\
 & & \lambda\mathtt{get}:\mathbf{ref}\;\mathbf{int} \to \mathbf{R}^{\bullet}\;\mathbf{int}. \\
 & & \quad \mathtt{let}\;\mathtt{x} = \mathtt{mk}\,0\;\mathtt{in}\;\mathtt{set}\,\mathtt{x}\,10;\,\mathtt{set}\,\mathtt{x}\,20;\,\mathtt{get}\,\mathtt{x} \\
 & & \\
[\![\mathtt{fun}]\!] & = & \lambda\mathtt{mk}:\mathtt{int} \to \mathbf{E}^{\bullet}_{0}\,\mathbf{ref}\,\mathbf{int}. \\
 & & \lambda\mathtt{set}:\mathbf{ref}\;\mathbf{int} \to \mathbf{E}^{\bullet}_{0}\;\mathbf{int} \to \mathbf{E}^{\bullet}_{0}\;\mathtt{unit}. \\
 & & \lambda\mathtt{get}:\mathbf{ref}\;\mathbf{int} \to \mathbf{E}^{\bullet}_{0}\;\mathbf{int}. \\
 & & \quad \mathtt{let}\;\mathtt{x} = \mathtt{mk}\,0\;\mathtt{in}\;\mathtt{set}\,\mathtt{x}\,10;\,\mathtt{set}\,\mathtt{x}\,20;\,\mathtt{get}\,\mathtt{x} \\
 & & \\
\mathtt{lib} & = & [\cdot]\;\;(\lambda\mathtt{n}\!:\mathtt{unit.\,ref\,n}) \\
 & & \qquad (\lambda\mathtt{r}\!:\mathtt{ref\,int.}\,\lambda\mathtt{n}\!:\mathtt{int.\,r} := \mathtt{n}) \\
 & & \qquad (\lambda\mathtt{r}\!:\mathtt{ref\,int.}\,!\mathtt{r}) \\
 & & \\
[\![\mathtt{lib}]\!] & = & [\cdot]\;\;(\lambda\mathtt{n}\!:\mathtt{unit.\,ref\,n}) \\
 & & \qquad (\lambda\mathtt{r}\!:\mathtt{ref\,int.}\,\lambda\mathtt{n}\!:\mathtt{int.\,r} := \mathtt{n}) \\
 & & \qquad (\lambda\mathtt{r}\!:\mathtt{ref\,int.}\,!\mathtt{r})
\end{array}$$

Figure 5: Example of λ linking against $\lambda_{\mathtt{ref}}$ library.

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

Figure 6: Type translation $\langle\!\langle\cdot\rangle\!\rangle$ for compiler $[\![\cdot]\!]$.

Formal Properties In general, a source language $\lambda_{\mathtt{src}}$ is enriched with a linking-type specification κ to create an extended language $\lambda^{\kappa}_{\mathtt{src}}$. The following must hold:

- $\lambda_{\mathtt{src}}$ 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, $\kappa^{-}(\kappa^{+}(\tau)) = \tau$.
- $\lambda_{\mathtt{src}}$ terms are a subset of $\lambda^{\kappa}_{\mathtt{src}}$ terms.
- $\lambda^{\kappa}_{\mathtt{src}}$ programs that only use $\lambda_{\mathtt{src}}$ terms co-diverge or co-terminate with equivalent values as the $\lambda_{\mathtt{src}}$ program obtained by applying κ^{-} to all types.

Linking is then defined by a fully abstract compiler from $\lambda^{\kappa}_{\mathtt{src}}$, where all unannotated types in the $\lambda_{\mathtt{src}}$ program are embedded with κ^{+}. This allows $\lambda_{\mathtt{src}}$ language programmers to link with components in arbitrary languages that have behavior only expressible within $\lambda^{\kappa}_{\mathtt{src}}$, without having to reason about the compilation target or details of compilation.

Collaboration This work has been done in collaboration with my advisor, Amal Ahmed. However, everything presented in this abstract is my own.

References

[1] Amal Ahmed and Matthias Blume. 2008. Typed closure conversion preserves observational equivalence. In Proceedings of the 13th ACM SIGPLAN international conference on Functional programming (ICFP ’08). ACM, New York, NY, USA, 157-168. DOI=http://dx.doi.org/10.1145/1411204.1411227

[2] Max S. New, William J. Bowman, and Amal Ahmed. 2016. Fully abstract compilation via universal embedding. In Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016). ACM, New York, NY, USA, 103-116. DOI: http://dx.doi.org/10.1145/2951913.2951941

[3] Dominique Devriese, Marco Patrignani, and Frank Piessens. 2016. Fully-abstract compilation by approximate back-translation. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’16). ACM, New York, NY, USA, 164-177. DOI: http://dx.doi.org/10.1145/2837614.2837618

[4] F. Henglein, H. Makholm, and H. Niss. Effect types and region-based memory management. In B. Pierce, editor, Advanced Topics in Types and Programming Languages. MIT Press, 2005.