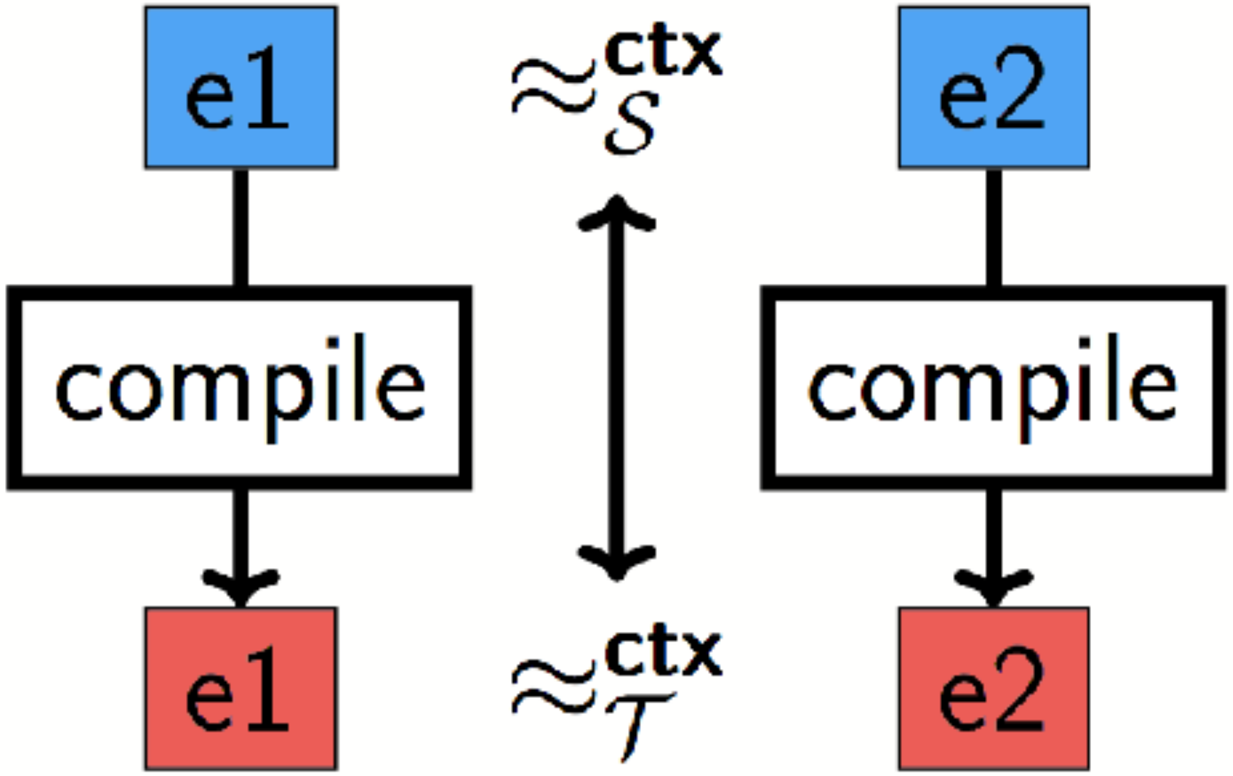
Linking Types: Bringing Fully Abstract Compilers and Flexible Linking Together

Daniel Patterson

Northeastern University

Fully abstract compilation

Fully abstract compilers preserve equivalences



- Target contexts (i.e., attackers) can't make observations impossible to make in source
- Refactoring / optimizations are not ruined by compiler
- Useful for programmer reasoning in correct compilers

But what about linking?

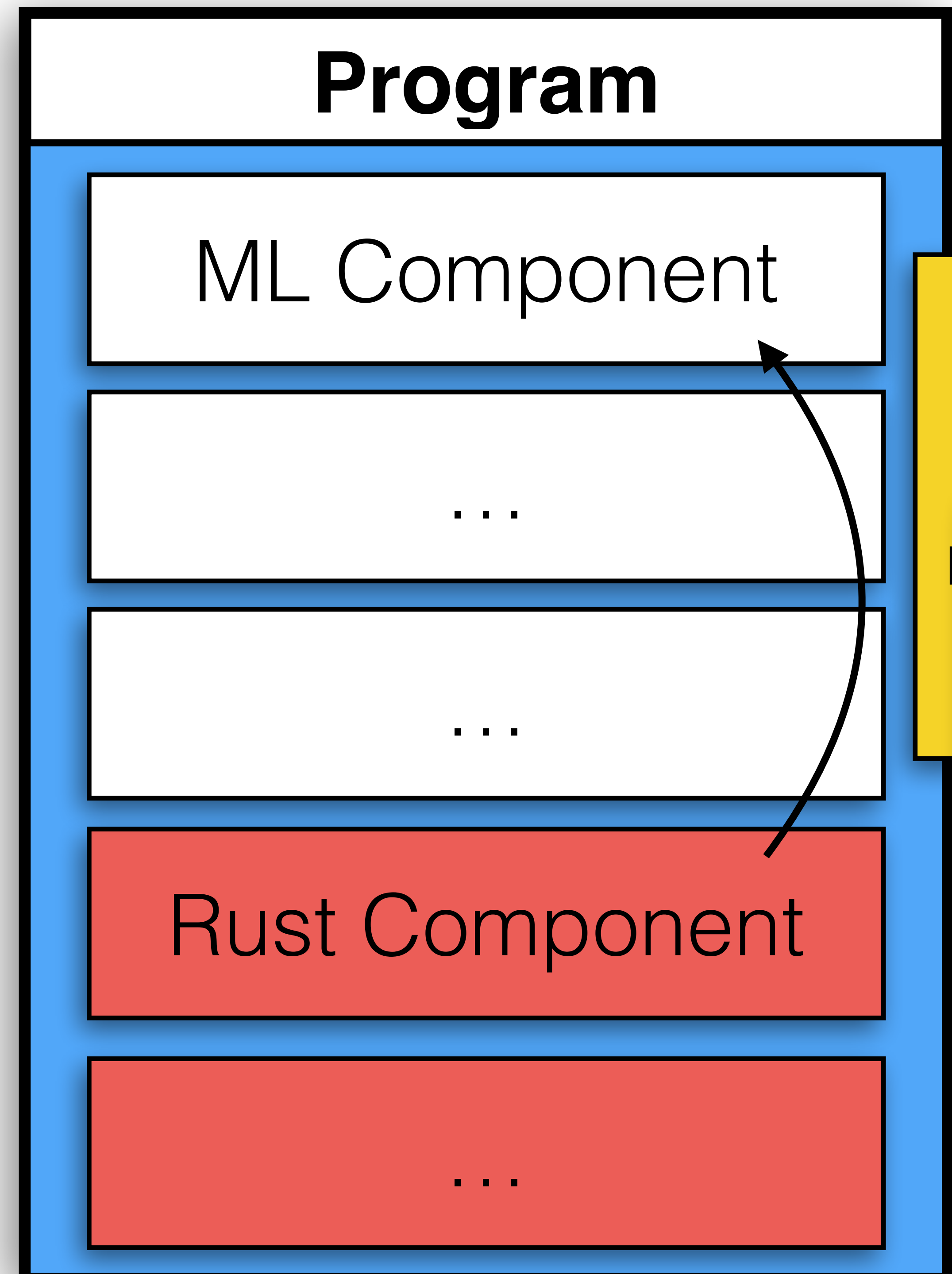
Fully abstract compilers prevent linking with code inexpressible in source language!

Often, equivalences induced by language are *too strong* to allow linking that programmer needs.

Linking types are about giving programmers control over equivalences …while retaining full abstraction

Linking Types for Multi-Language Software: Have Your Cake and Eat it Too *[Patterson-Ahmed SNAPL'17]*

Example multi-language system



For Rust owned values to flow into ML, need to

A fully abstract Rust compiler would prevent linking, but if ML programmer annotates where values are treated linearly, linking can be allowed.

In a simple setting

λ (simply-typed lambda calculus)

$$\tau ::= \mathbf{unit} \mid \mathbf{int} \mid \tau \to \tau$$

$$\mathsf{e} ::= () \mid \mathsf{n} \mid \mathsf{x} \mid \lambda \mathsf{x} : \tau.\mathsf{e} \mid \mathsf{e}\,\mathsf{e} \mid \mathsf{e} + \mathsf{e} \mid \mathsf{e} * \mathsf{e}$$

λ^{ref} (extended with ML references)

$$\tau ::= \ldots \mid \mathsf{ref}\,\tau$$

$$\mathsf{e} ::= \ldots \mid \mathsf{ref}\,\mathsf{e} \mid \mathsf{e} := \mathsf{e} \mid !\mathsf{e}$$

How to build fully abstract compiler for λ that can link with λ^{ref} ?

With a fully abstract compiler, λ programmer should be able to refactor safely.

Reasoning about refactoring

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

Should be okay because

$$\lambda \mathbf{c}.\, \mathbf{c}(); \mathbf{c}() \approx_{\lambda}^{ctx} \lambda \mathbf{c}.\, \mathbf{c}()$$

What about linking with λ^{ref}?

$$\begin{aligned}
&\texttt{let counter f}' = \mathsf{let\,v} = \mathsf{ref\,0\,in} \\
&\qquad\qquad\qquad\quad \mathsf{let\,c'\,()} = \mathsf{v} := !\mathsf{v} + 1;\ !\mathsf{v\,in\,}\texttt{f}'\,\mathsf{c}' \\
&\texttt{let f} \qquad\qquad\quad = \lambda\mathsf{c} : \mathbf{unit} \to \mathbf{int}.\,\mathbf{c}();\mathbf{c}() \\
&\texttt{in counter f}
\end{aligned} \Downarrow 2$$

but

$$\begin{aligned}
&\texttt{let counter f}' = \mathsf{let\,v} = \mathsf{ref\,0\,in} \\
&\qquad\qquad\qquad\quad \mathsf{let\,c'\,()} = \mathsf{v} := !\mathsf{v} + 1;\ !\mathsf{v\,in\,}\texttt{f}'\,\mathsf{c}' \\
&\texttt{let f} \qquad\qquad\quad = \lambda\mathsf{c} : \mathbf{unit} \to \mathbf{int}.\,\mathbf{c}() \\
&\texttt{in counter f}
\end{aligned} \Downarrow 1$$

When linked with λ^{ref}, no longer equivalent!

Is this refactoring correct?

$$\lambda \mathbf{c}.\, \mathbf{c}(); \mathbf{c}() \Rightarrow \lambda \mathbf{c}.\, \mathbf{c}() \; : (\mathbf{unit} \to \mathbf{int}) \to \mathbf{int}$$

It depends on what it is linked with!

$$\mathbf{unit} \to \mathbf{int} \checkmark \qquad \mathsf{unit} \to \mathsf{int} \; ✗$$

Programmer should be able to specify which they want, so that the compiler can be fully abstract!

λ with linking types extension

$$\tau ::= \mathbf{unit} \mid \mathbf{int} \mid \tau \to \tau$$

$$\lambda^{\kappa} \quad \tau ::= \mathbf{unit} \mid \mathbf{int} \mid \tau \to \mathbf{R}^{\emptyset}\,\tau \mid \mathbf{ref}\,\tau \mid \tau \to \mathbf{R}^{\lightning}\,\tau$$

Type and effect systems, e.g., F*, Koka

λ^{κ} allows programmers to write both

$\mathbf{unit} \rightarrow \mathbf{int}$	$\mathsf{unit} \rightarrow \mathsf{int}$
$\mathbf{unit} \rightarrow \mathbf{R}^{\emptyset}\mathbf{int}$	$\mathbf{unit} \rightarrow \mathbf{R}^{\lightning}\mathbf{int}$

Refactoring: pure inputs

$$\lambda \mathsf{c} : \mathbf{unit} \to \mathbf{R}^{\emptyset}\mathbf{int}.\, \mathsf{c}(); \mathsf{c}() \approx^{ctx}_{\lambda^{\kappa}} \lambda \mathsf{c} : \mathbf{unit} \to \mathbf{R}^{\emptyset}\mathbf{int}.\, \mathsf{c}()$$

$$\begin{aligned}
&\texttt{let counter f}' = \mathsf{let\ v = ref\ 0\ in} \\
&\qquad\qquad\qquad\quad \mathsf{let\ c}'() = \mathsf{v} := !\mathsf{v} + 1;\ !\mathsf{v\ in}\ \texttt{f}'\ \mathsf{c}' \\
&\texttt{let f} \qquad\qquad\quad = \lambda \mathsf{c} : \mathbf{unit} \to \mathbf{R}^{\emptyset}\mathbf{int}.\, \mathsf{c}() \\
&\texttt{in counter f}
\end{aligned}$$

Ill-typed, since f requires pure code

Refactoring: impure inputs

$$\lambda c : \mathbf{unit} \to \mathbf{R}^{\lightning}\mathbf{int}.\ c();c() \not\approx^{ctx}_{\lambda^{\kappa}} \lambda c : \mathbf{unit} \to \mathbf{R}^{\lightning}\mathbf{int}.\ c()$$

$$\begin{aligned}
&\texttt{let counter f}' = \mathsf{let\ v} = \mathsf{ref\ 0\ in} \\
&\qquad\qquad\qquad\quad \mathsf{let\ c}'\,() = \mathsf{v} := !\mathsf{v} + 1;\ !\mathsf{v}\ \mathsf{in}\ \mathtt{f}'\ \mathsf{c}' \\
&\texttt{let f} \qquad\qquad\ \ = \lambda c : \mathbf{unit} \to \mathbf{R}^{\lightning}\mathbf{int}.\ c() \\
&\texttt{in counter f}
\end{aligned}$$

Well-typed, since f accepts impure code

Minimal annotation burden

$$\lambda \mathbf{c} : \mathbf{unit} \rightarrow \mathbf{R}^{\emptyset}\mathbf{int}.\, \mathbf{c}(); \mathbf{c}()$$

$$\lambda \mathbf{c} : \mathbf{unit} \rightarrow \mathbf{int}.\, \mathbf{c}(); \mathbf{c}()$$

λ^{κ} must provide default translation

$$\begin{aligned} \kappa^{+}(\mathbf{unit}) &= \mathbf{unit} \\ \kappa^{+}(\mathbf{int}) &= \mathbf{int} \\ \kappa^{+}(\tau_1 \rightarrow \tau_2) &= \kappa^{+}(\tau_1) \rightarrow \mathbf{R}^{\emptyset}\, \kappa^{+}(\tau_2) \end{aligned}$$

$$\forall \mathbf{e}_1, \mathbf{e}_2.\ \mathbf{e}_1 \approx^{ctx}_{\lambda} \mathbf{e}_2 : \tau \implies \mathbf{e}_1 \approx^{ctx}_{\lambda^{\kappa}} \mathbf{e}_2 : \kappa^{+}(\tau)$$

Stepping back…

Fully Abstract Compilation?

escape hatches

ML — C FFI

Rust — unsafe

Java — JNI

Language specifications are incomplete!
Don't account for linking

Target

Rethink PL design with linking types

escape hatches

ML — C FFI

Rust — unsafe

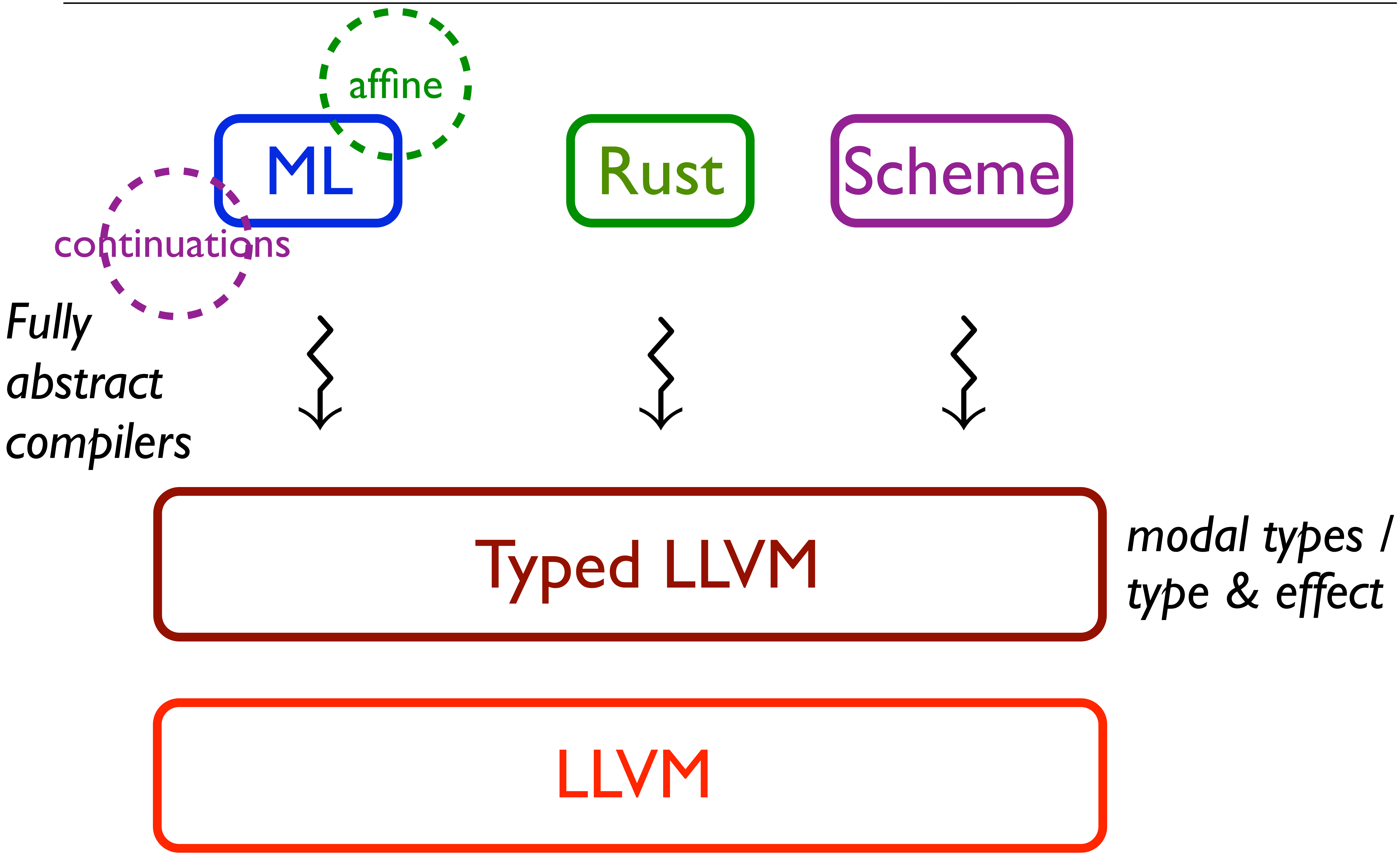
Java — JNI

Design linking types extensions that support safe interoperability with other languages

PL design, linking types

Only need linking types extensions to interact with behavior inexpressible in your language.

PL design, linking types, compilers



Linking Types

- Allow programmers to specify what they want to link with, with fine granularity.
- This allows compilers to be fully abstract, yet support multi-language linking.