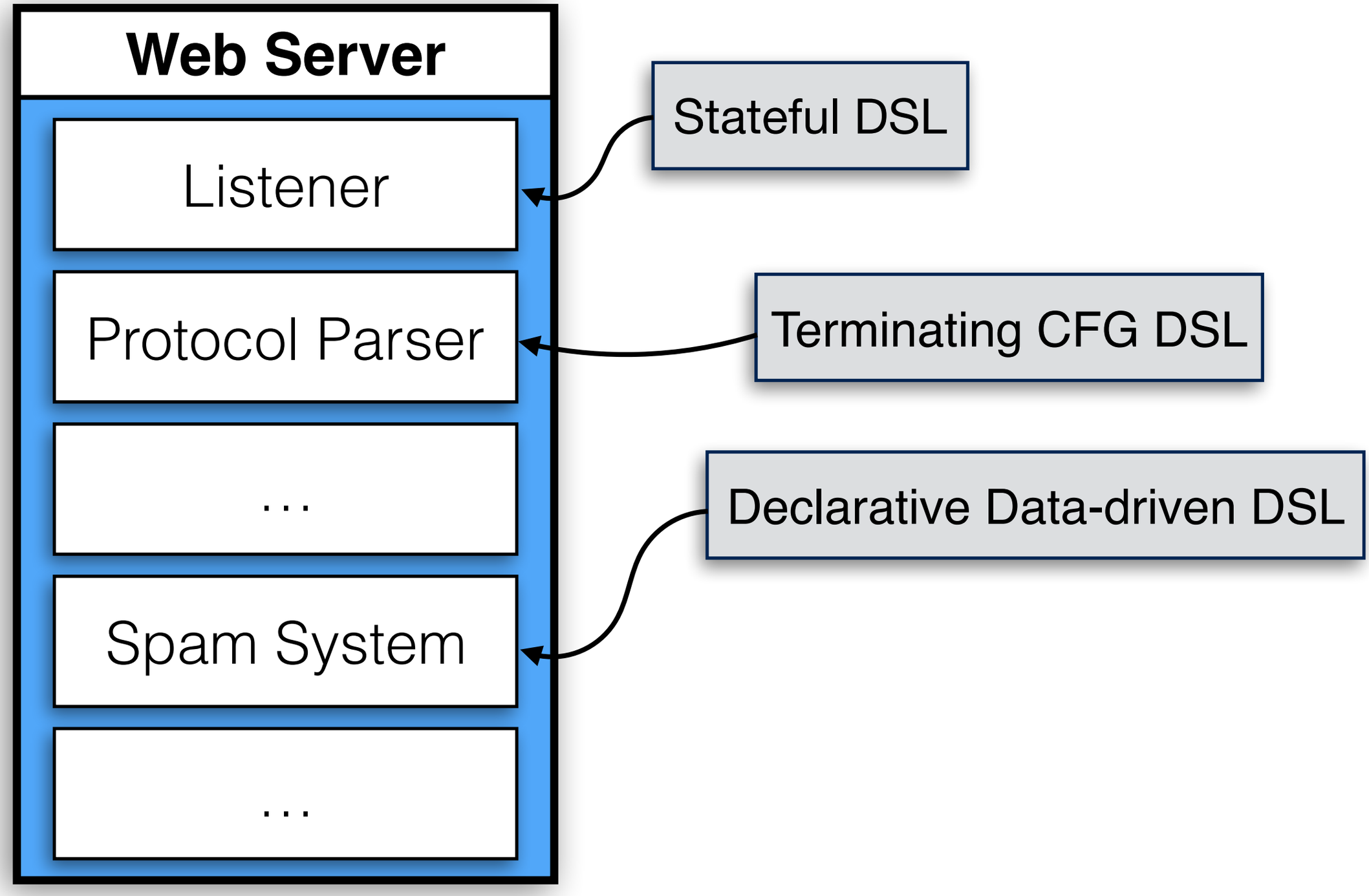
Linking Types for Multi-Language Software

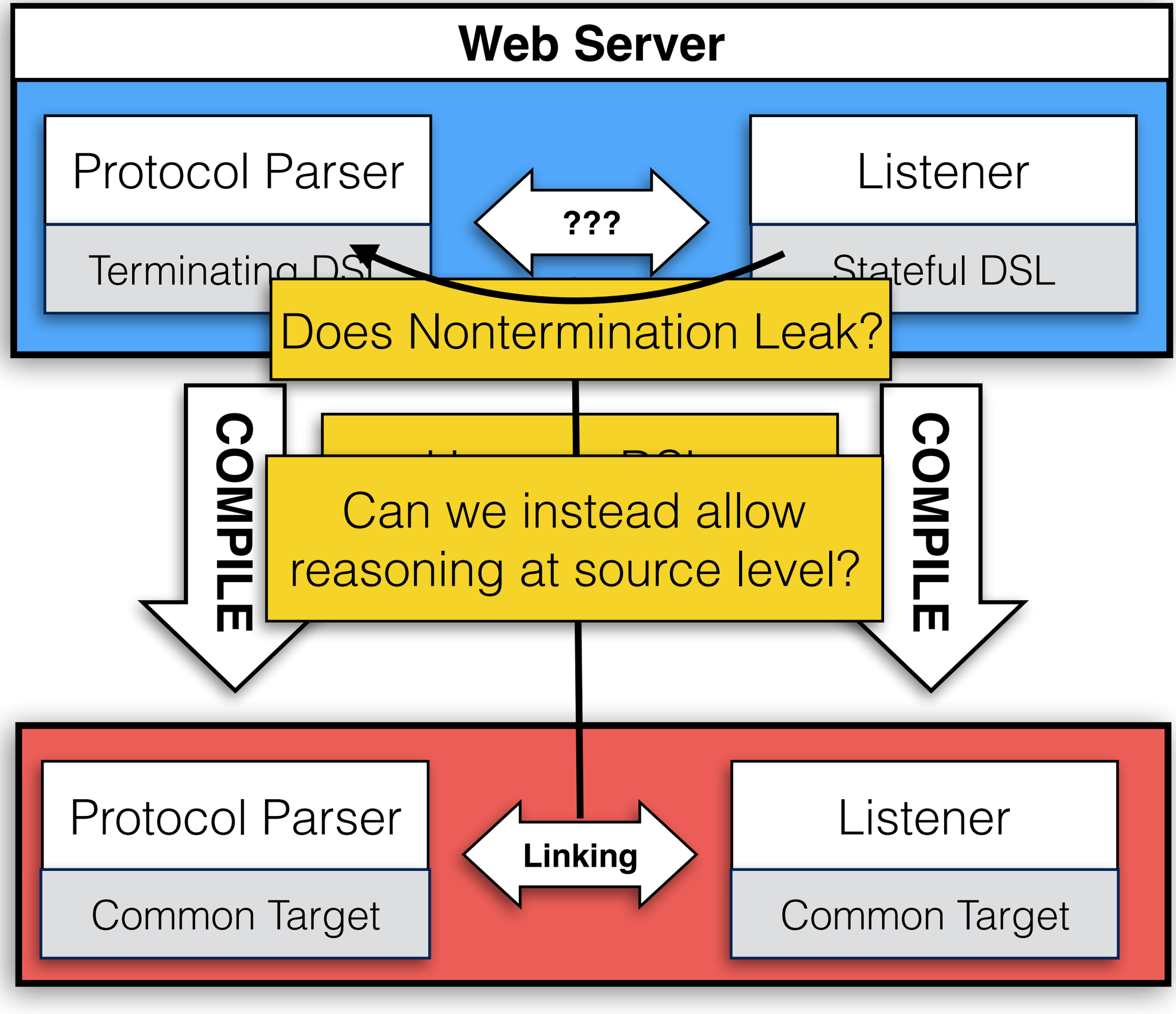
Have Your Cake and Eat It Too

Daniel Patterson and Amal Ahmed

SNAPL 2017

Multi-Language System




Web Server
Protocol Parser
Terminating DSL
???
Listener
Stateful DSL
Does Nontermination Leak?
COMPILE
Can we instead allow reasoning at source level?
COMPILE
Protocol Parser
Common Target
Linking
Listener
Common Target

Linking types are about raising programmer reasoning back to the source level

In a Simpler Setting

λ (simply-typed lambda calculus)

$$\begin{array}{lll} \tau & ::= & \mathbf{unit} \mid \mathbf{int} \mid \tau \to \tau \\ \mathbf{e} & ::= & () \mid \mathbf{n} \mid \mathbf{x} \mid \lambda \mathbf{x} : \tau . \mathbf{e} \\ & & \mid \mathbf{e}\,\mathbf{e} \mid \mathbf{e} + \mathbf{e} \mid \mathbf{e} * \mathbf{e} \end{array}$$

λ^{ref} (extended with ML references)

$$\begin{array}{lll} \tau & ::= & \ldots \mid \mathsf{ref}\,\tau \\ \mathsf{e} & ::= & \ldots \mid \mathsf{ref}\,\mathsf{e} \mid \mathsf{e} := \mathsf{e} \mid !\mathsf{e} \end{array}$$

How to reason in λ while linking with λ^{ref} ?

Refactoring is reasoning about equivalence

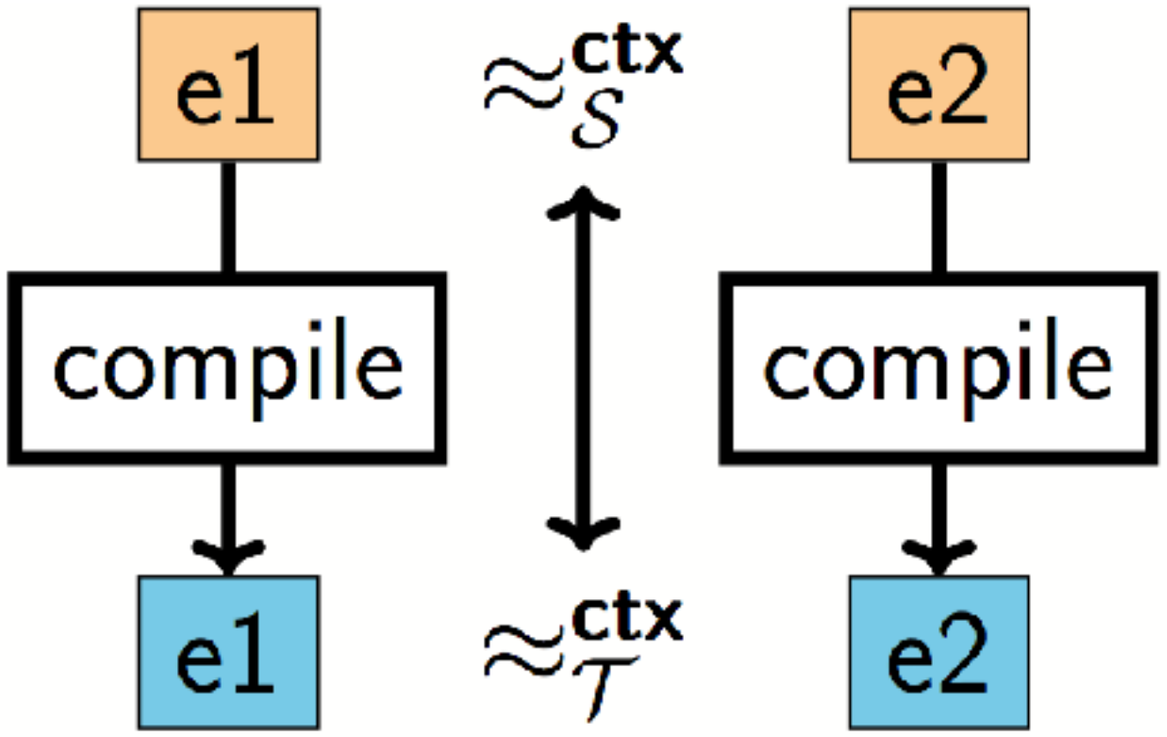
Reasoning About Refactoring

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

Should be okay because

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

Fully abstract compilers preserve equivalences



What about linking with λ^{ref}?

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

but

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

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

Is this refactoring correct?

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

It depends on what it is linked with!

$\mathbf{unit} \to \mathbf{int}$ ✓

$\mathbf{unit} \to \mathbf{int}$ ✗

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

λ with linking types extension

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

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

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

λ^{κ} Allows Programmers To Write Both

$$\mathbf{unit} \to \mathbf{int} \qquad \mid \qquad \mathbf{unit} \to \mathbf{int}$$

$$\mathbf{unit} \to \mathbf{R}^{\emptyset}\mathbf{int} \qquad \mid \qquad \mathbf{unit} \to \mathbf{R}^{\lightning}\mathbf{int}$$

Refactoring: Pure Inputs

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

```
let counter f' = let v = ref 0 in
                 let c' () = v := !v + 1; !v in f' c'
let f
in counter f
```

$\lambda c\colon \mathbf{unit} \to \mathbf{R}^{\emptyset}\mathbf{int}.\ c()$

Ill-typed, since `f` requires pure code

Refactoring: Impure Inputs

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

```
let counter f' = let v = ref 0 in
                 let c' () = v := !v + 1; !v in f' c'
let f          = λc : unit → R↯int. c()
in counter f
```

Well-typed, since `f` accepts impure code

Minimal Annotation Burden

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

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

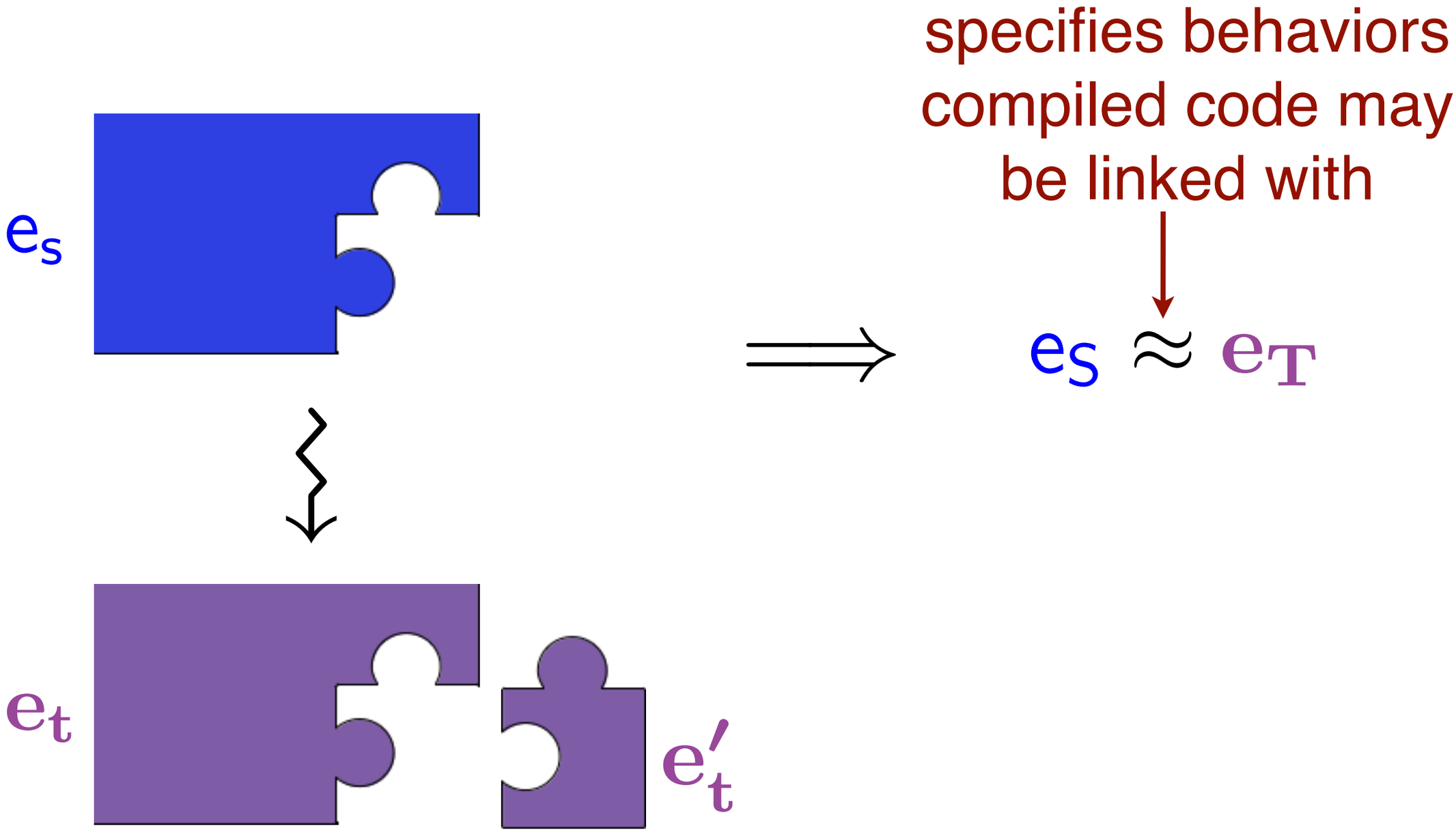
λ^{κ} must provide default translation

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

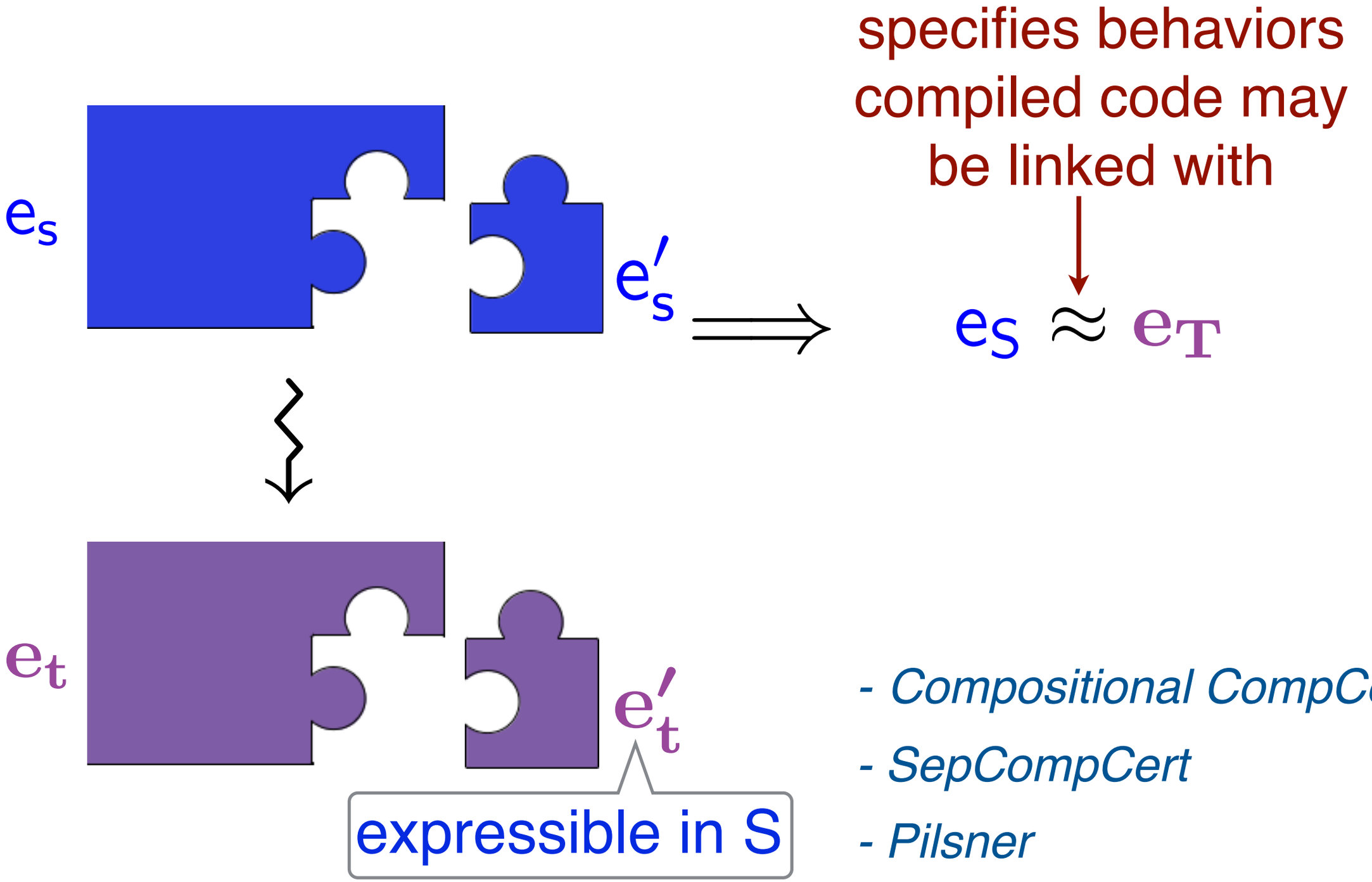
$$\forall \mathbf{e_1}, \mathbf{e_2}.\ \mathbf{e_1} \approx^{ctx}_{\lambda} \mathbf{e_2} : \tau \implies \mathbf{e_1} \approx^{ctx}_{\lambda^{\kappa}} \mathbf{e_2} : \kappa^{+}(\tau)$$

Stepping Back…

Correct Compilation of Components

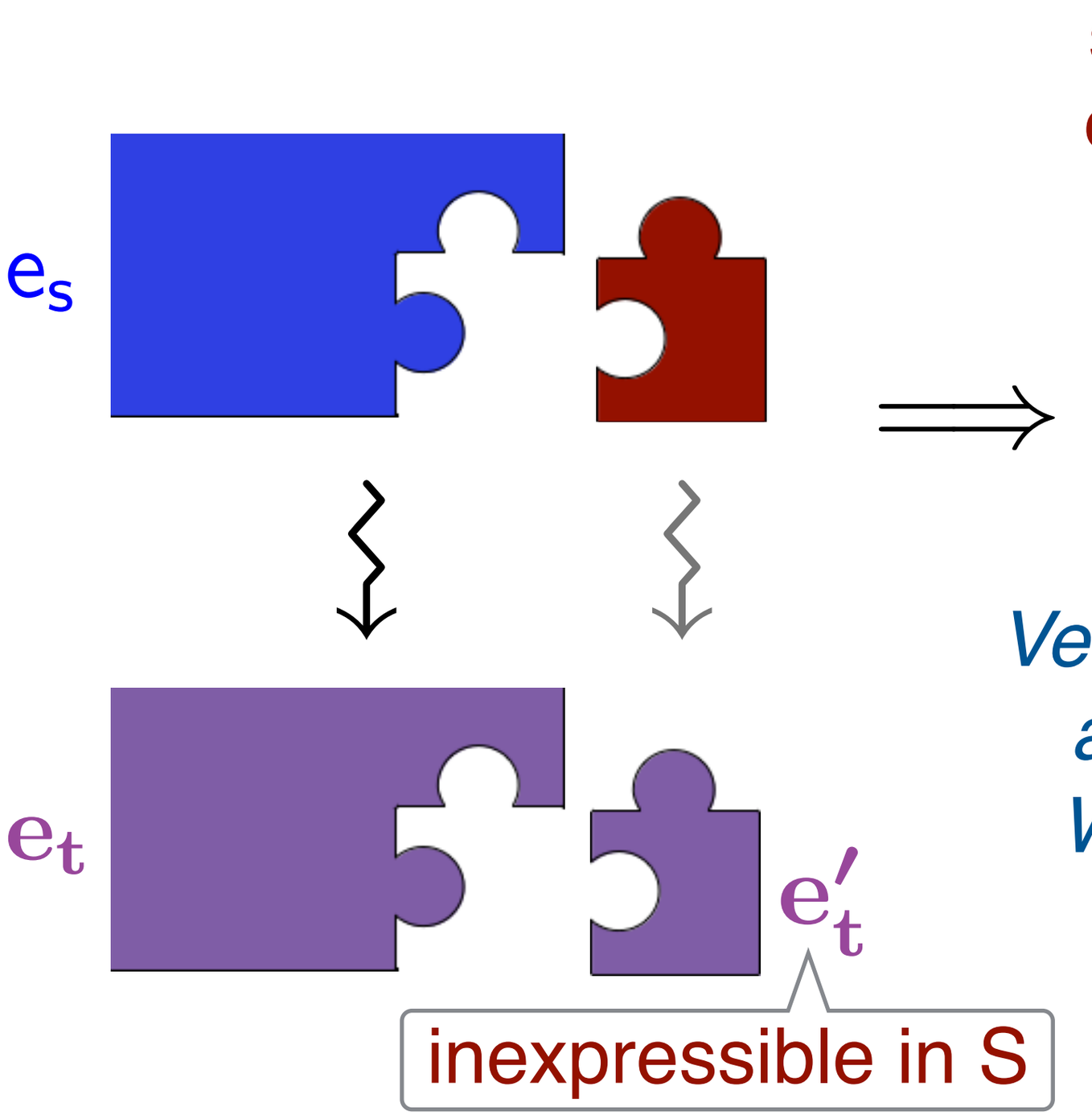


Correct Compilation of Components



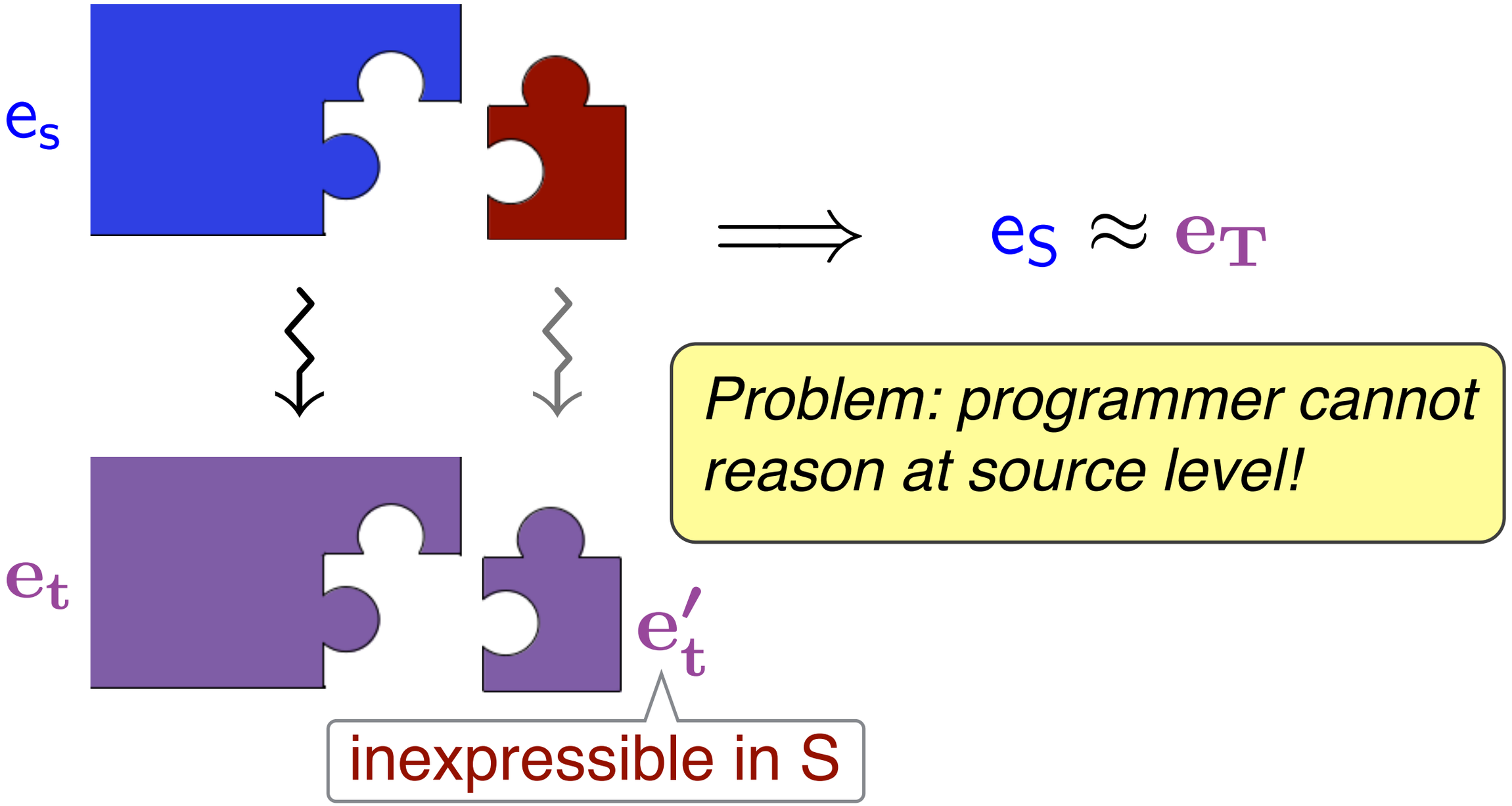
- *Compositional CompCert*
- *SepCompCert*
- *Pilsner*

Correct Compilation: Multi-Language



Verified Compilers for a Multi-Language World [SNAPL'15]

Correct Compilation: Multi-Language



Fully Abstract Compilation?

escape hatches

ML — C FFI

Rust — unsafe

Java — JNI

Language specifications are incomplete!
Don't account for linking

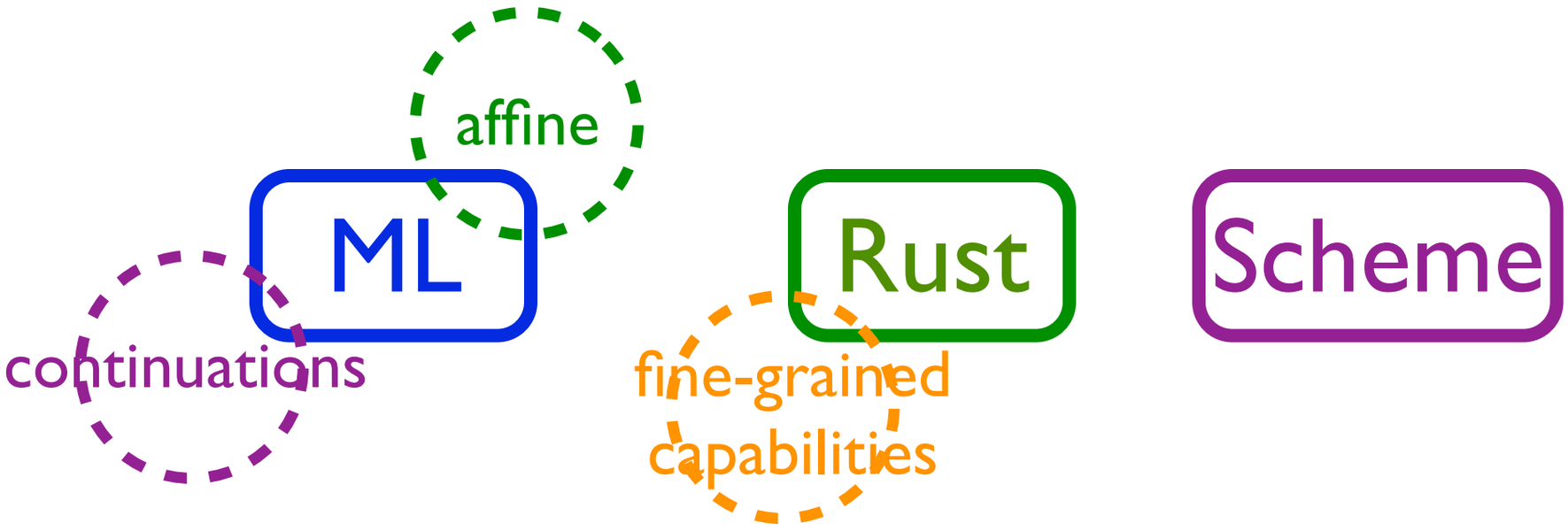
Target

Rethink PL Design with Linking Types

escape hatches

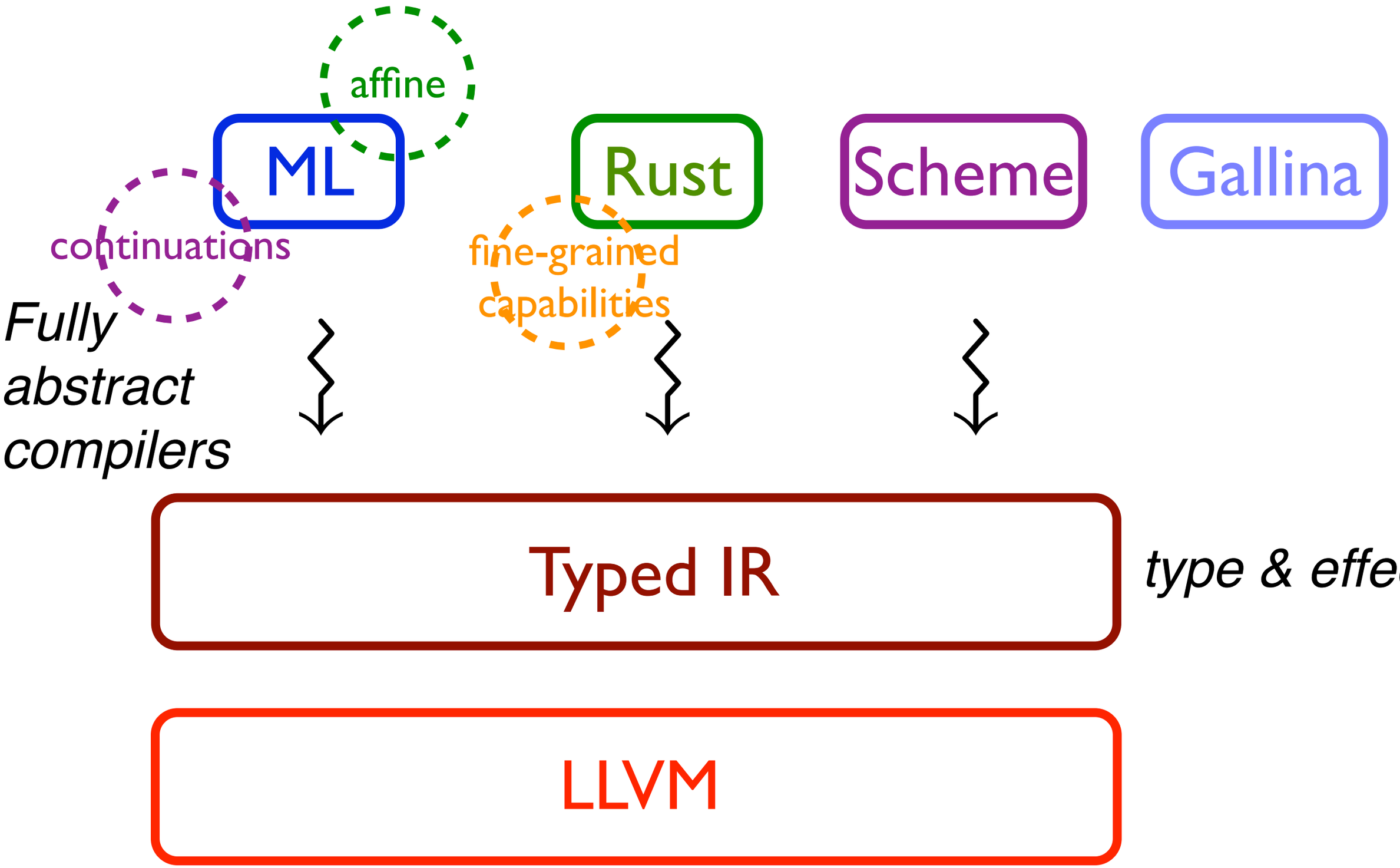
Design linking types extensions that support safe interoperability with other languages

PL Design, Linking Types

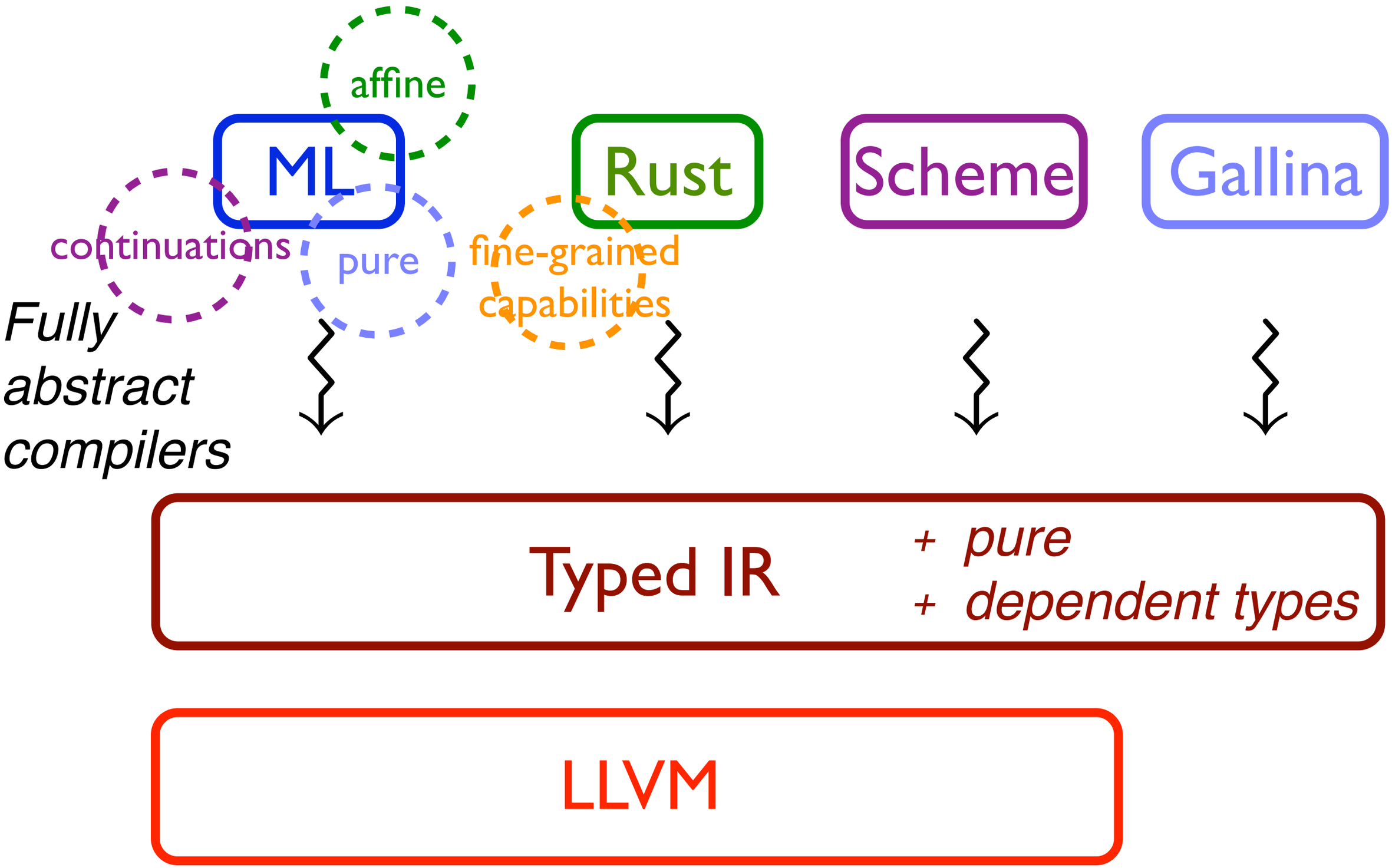


Only need linking types extensions to interact with behavior inexpressible in your language.

PL Design, Linking Types, Compilers



PL Design, Linking Types, Compilers

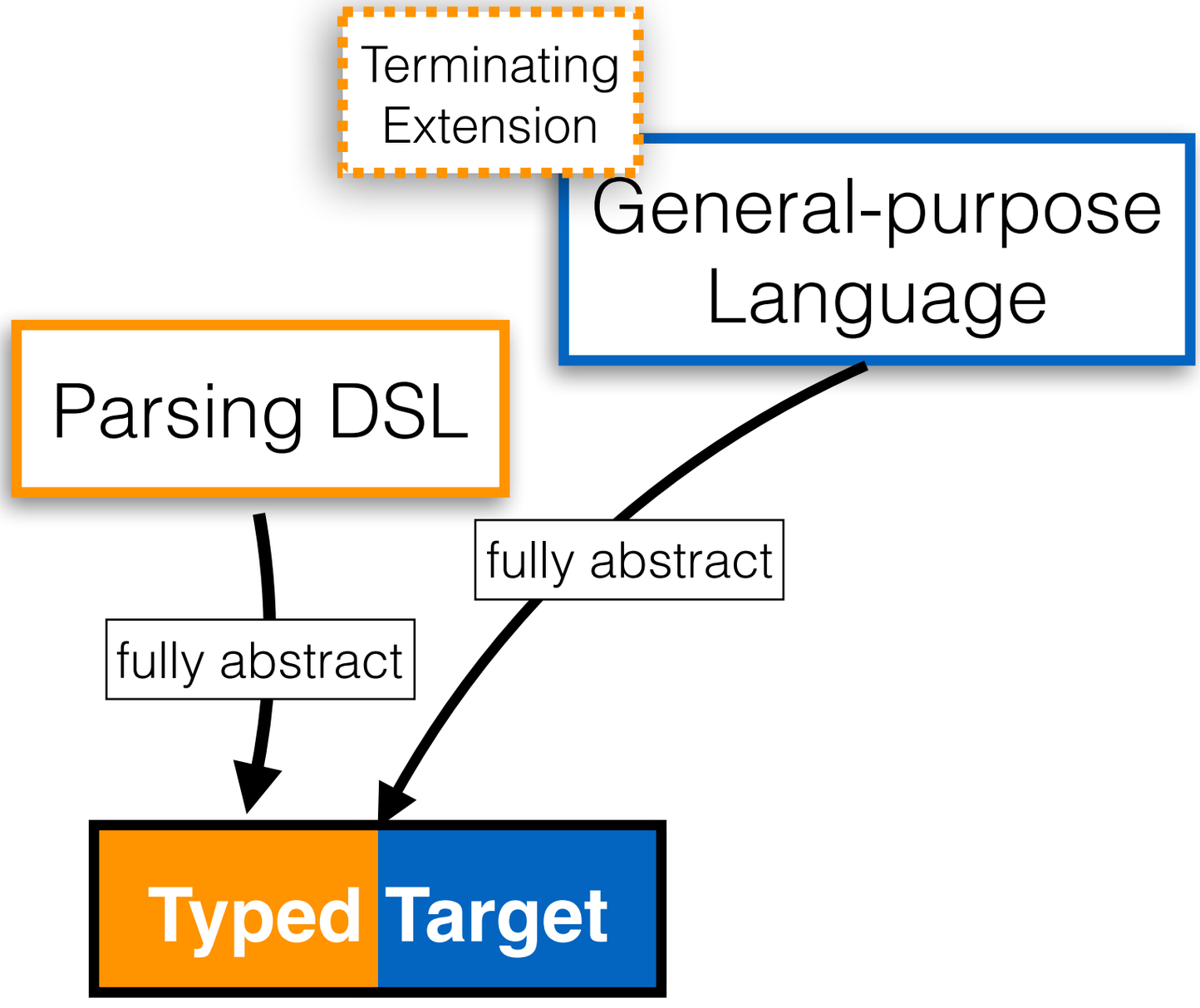


Linking Types

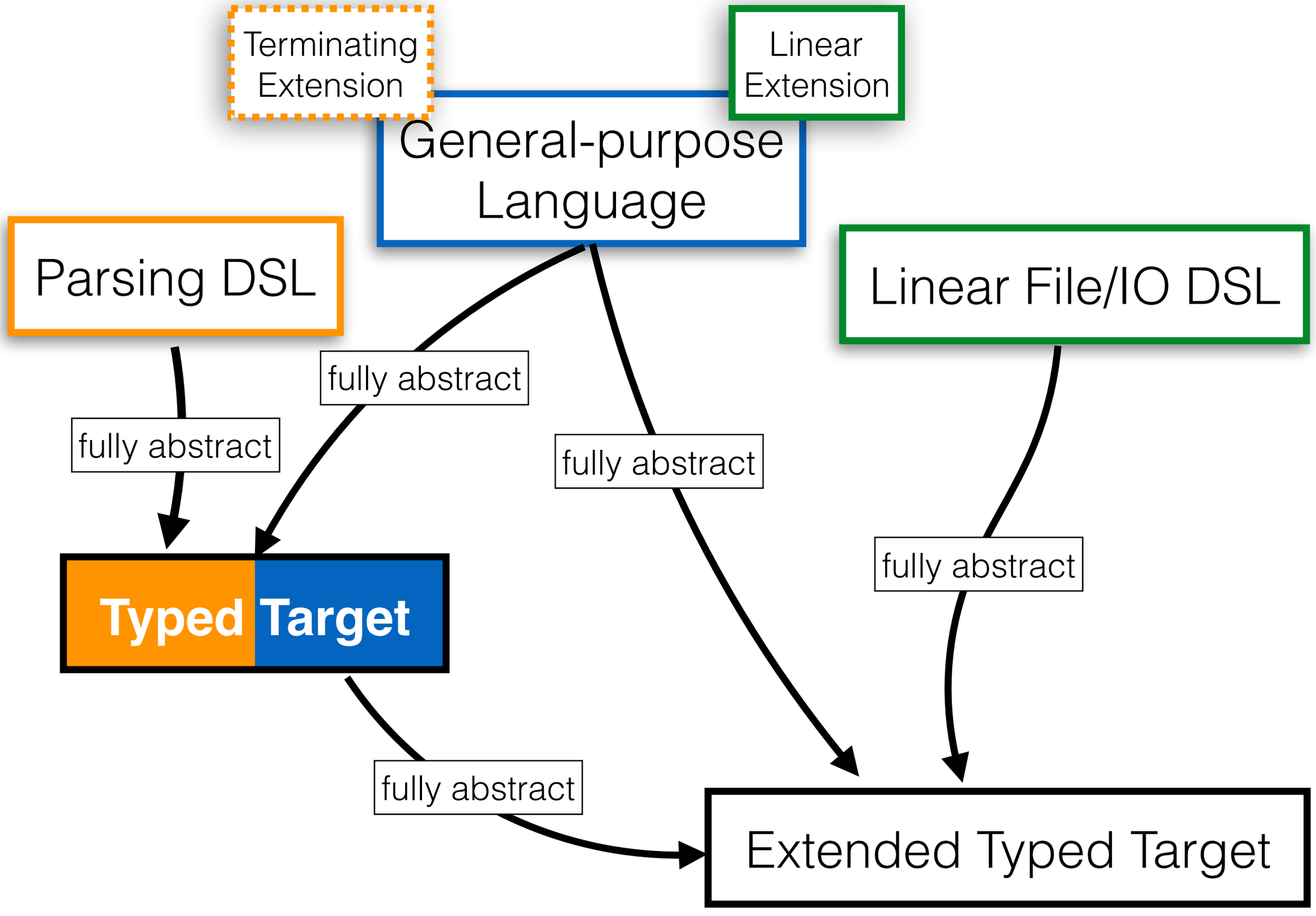
- Programmers can reason in *almost* their source languages, even when building multi-language software.
- Compilers can be fully abstract, yet support multi-language linking.

Extra slides

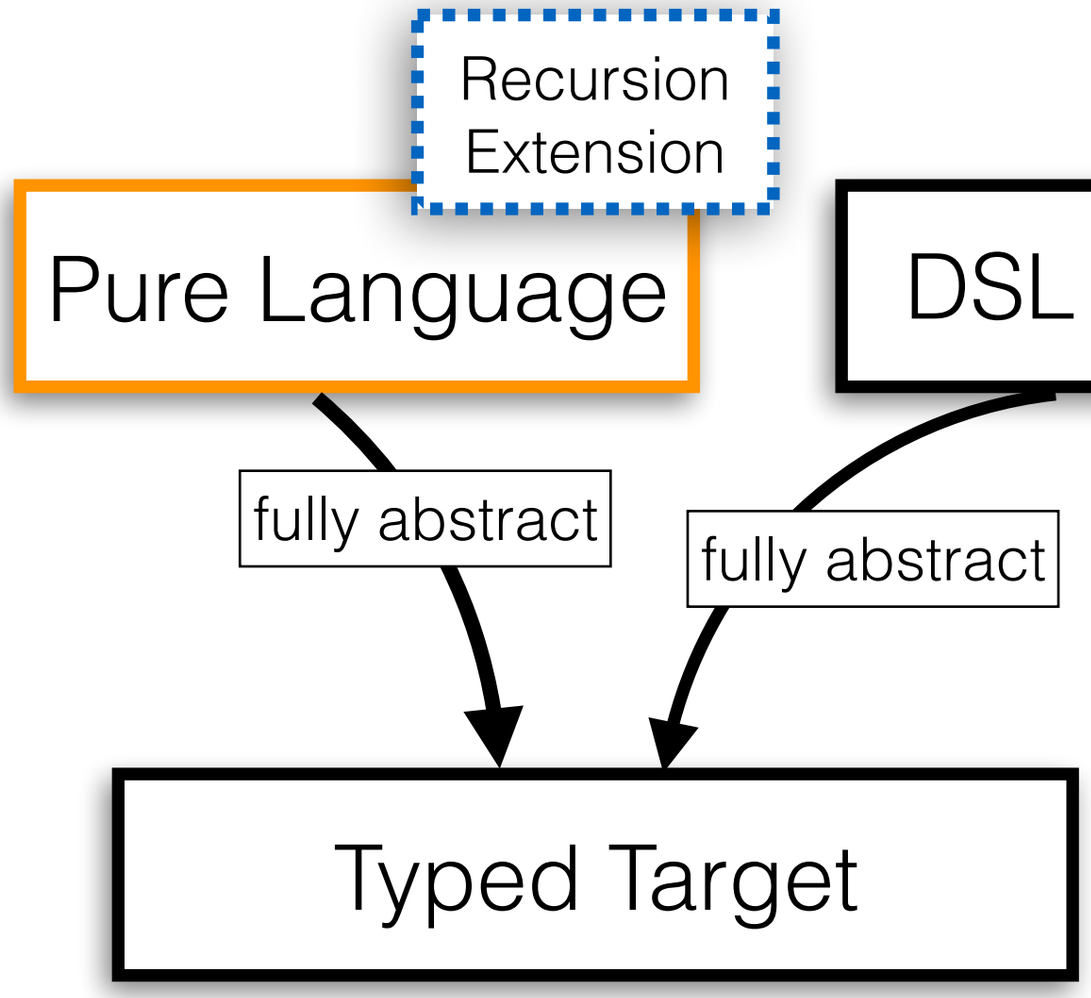
Bigger Picture: Linking Types & Compilers



Bigger Picture: Evolution



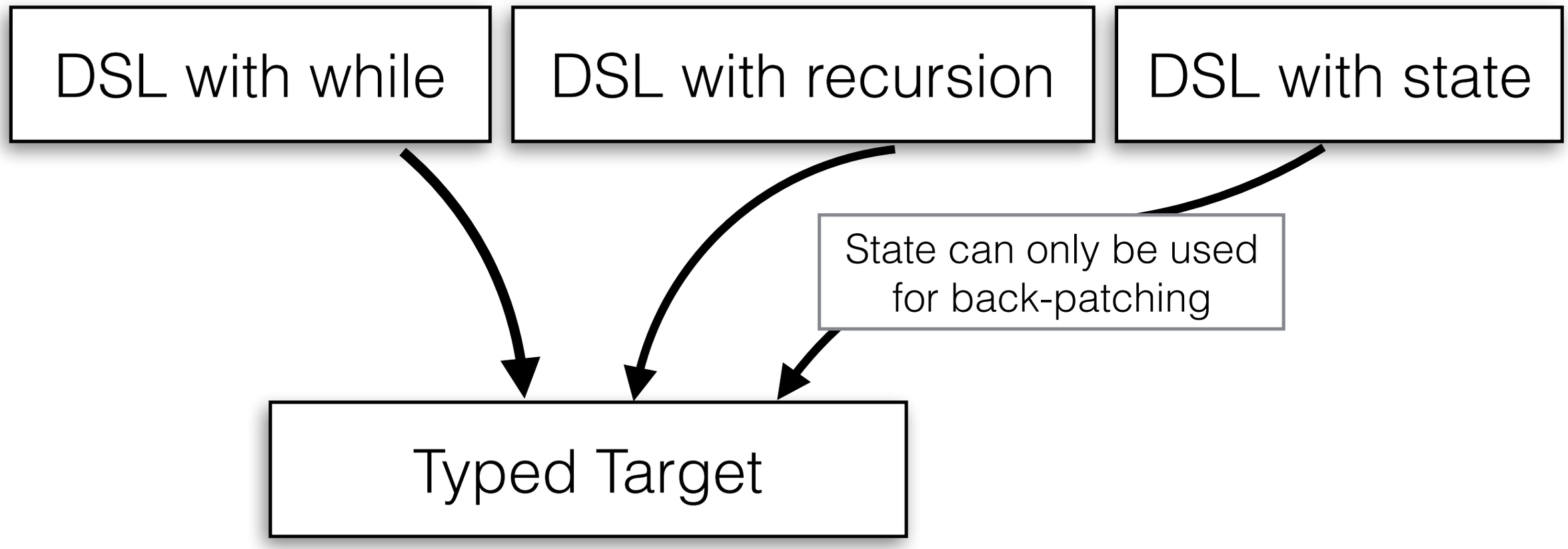
Bigger Picture: Behavior Not Features



All exceptions must be caught

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

Bigger Picture



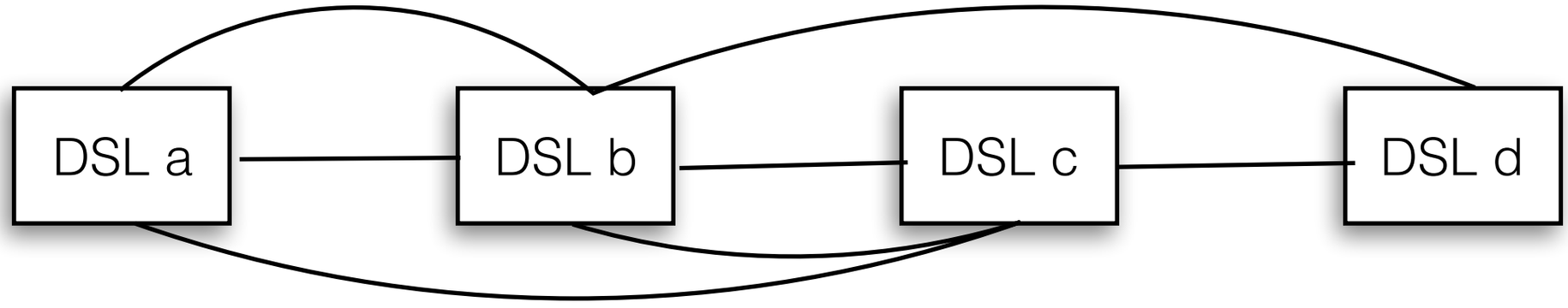
No extension needed if behavior can be represented, even if surface features are different.

Fully abstract compilers for languages with linking types

Allow programmers to reason in "almost" their source language.

But still link with code inexpressible in their language.

Vision of many domain specific languages



With Racket / Turnstile (POPL'17), each DSL can be typed, but for interaction need to reason about untyped code after expansion.

With Haskell/Scala, DSL types are encoded in host type system, so for interaction need to deal with (complex) encoding in the host language.

With linking types and fully abstract compilers, programmers need only reason in the DSL they are using.

PL Design and Compilers

ML

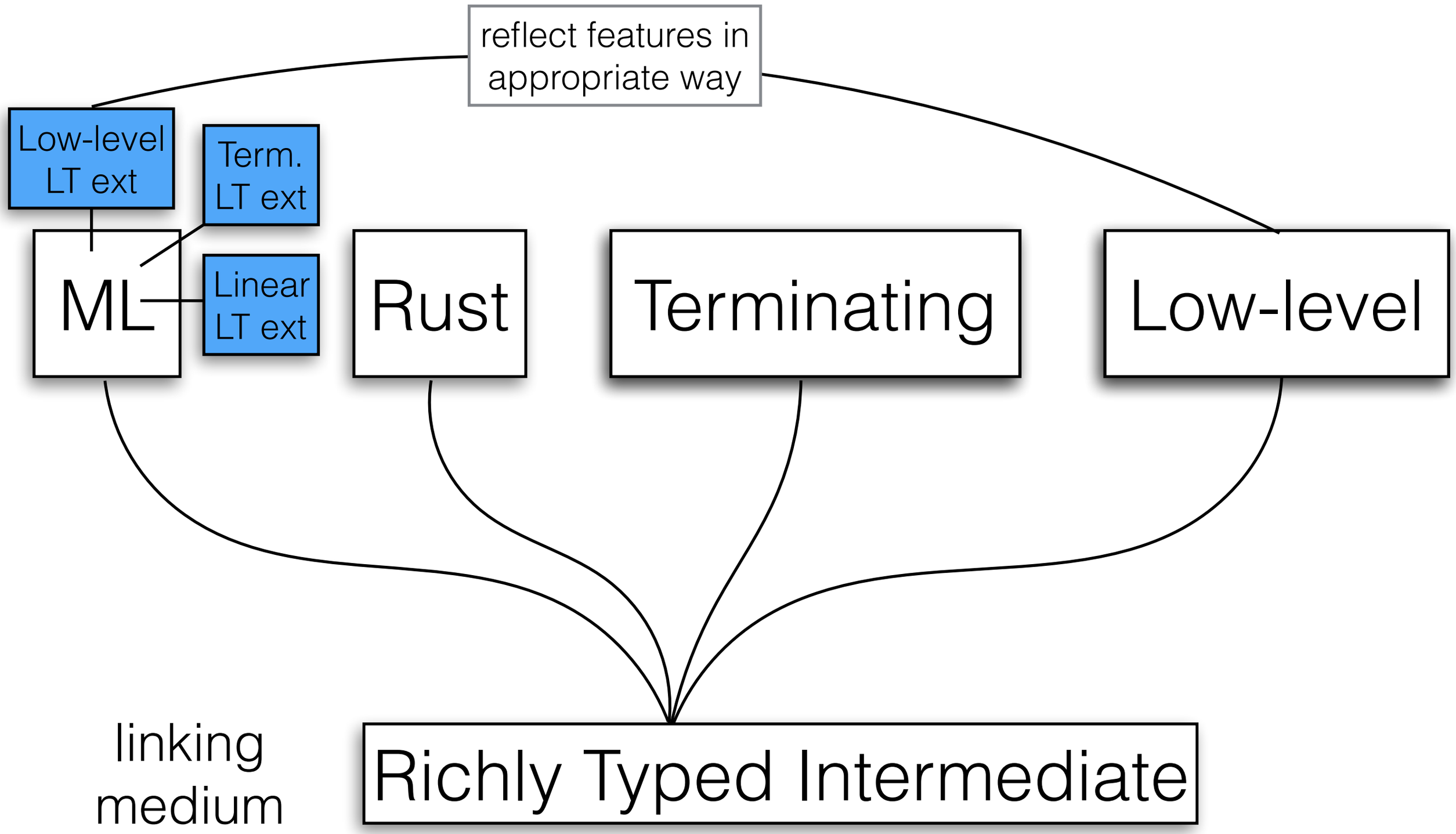
Rust

Java

C

Assembly

Linking with existing languages



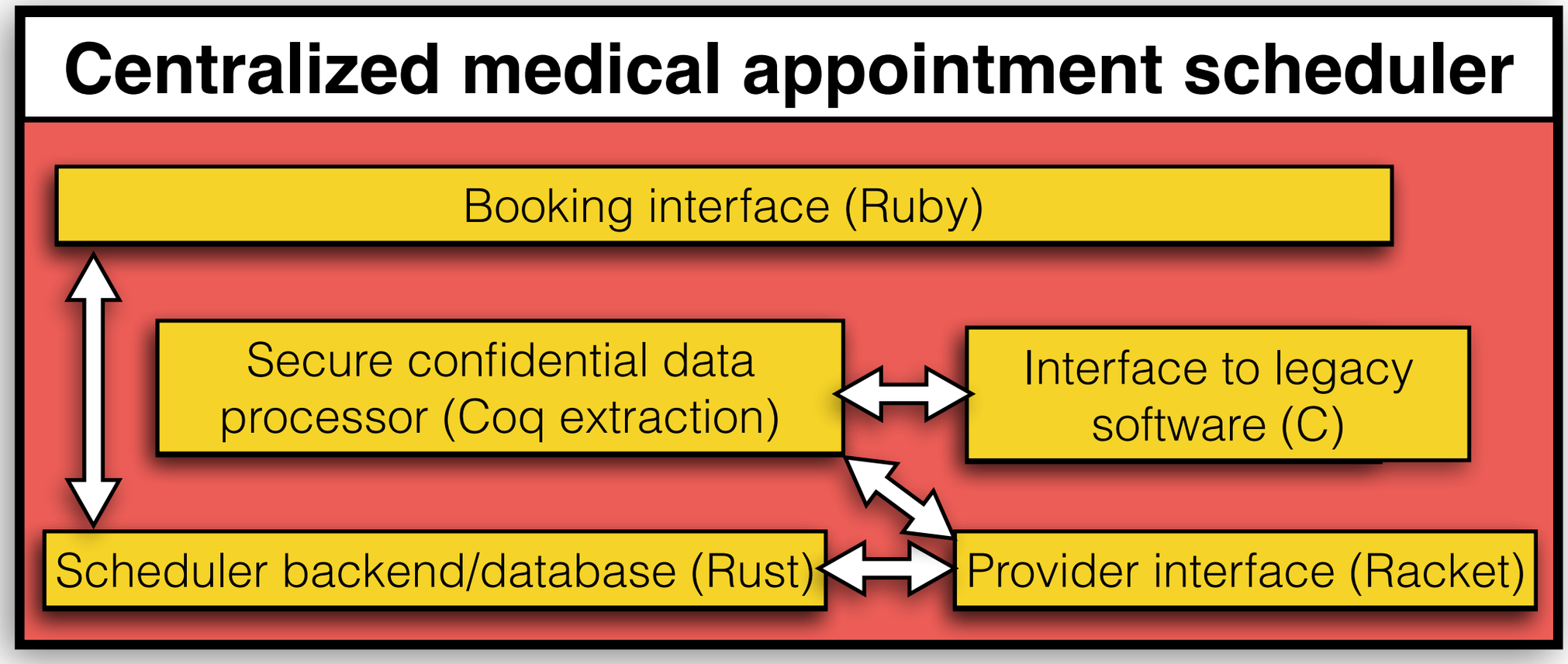
We shouldn't <u>have to</u> annotate

default translation

$$\kappa^+(\mathbf{unit}) = \mathbf{unit}$$
$$\kappa^+(\mathbf{int}) = \mathbf{int}$$
$$\kappa^+(\tau_1 \to \tau_2) = \kappa^+(\tau_1) \to \mathbf{R}^\circ\, \kappa^+(\tau_2)$$

$$\forall \mathbf{e_1}, \mathbf{e_2}.\ \mathbf{e_1} \approx^{ctx}_{\lambda} \mathbf{e_2} : \tau \implies \mathbf{e_1} \approx^{ctx}_{\lambda^\kappa} \mathbf{e_2} : \kappa^+(\tau)$$

A multi-language system:



How do we allow this:

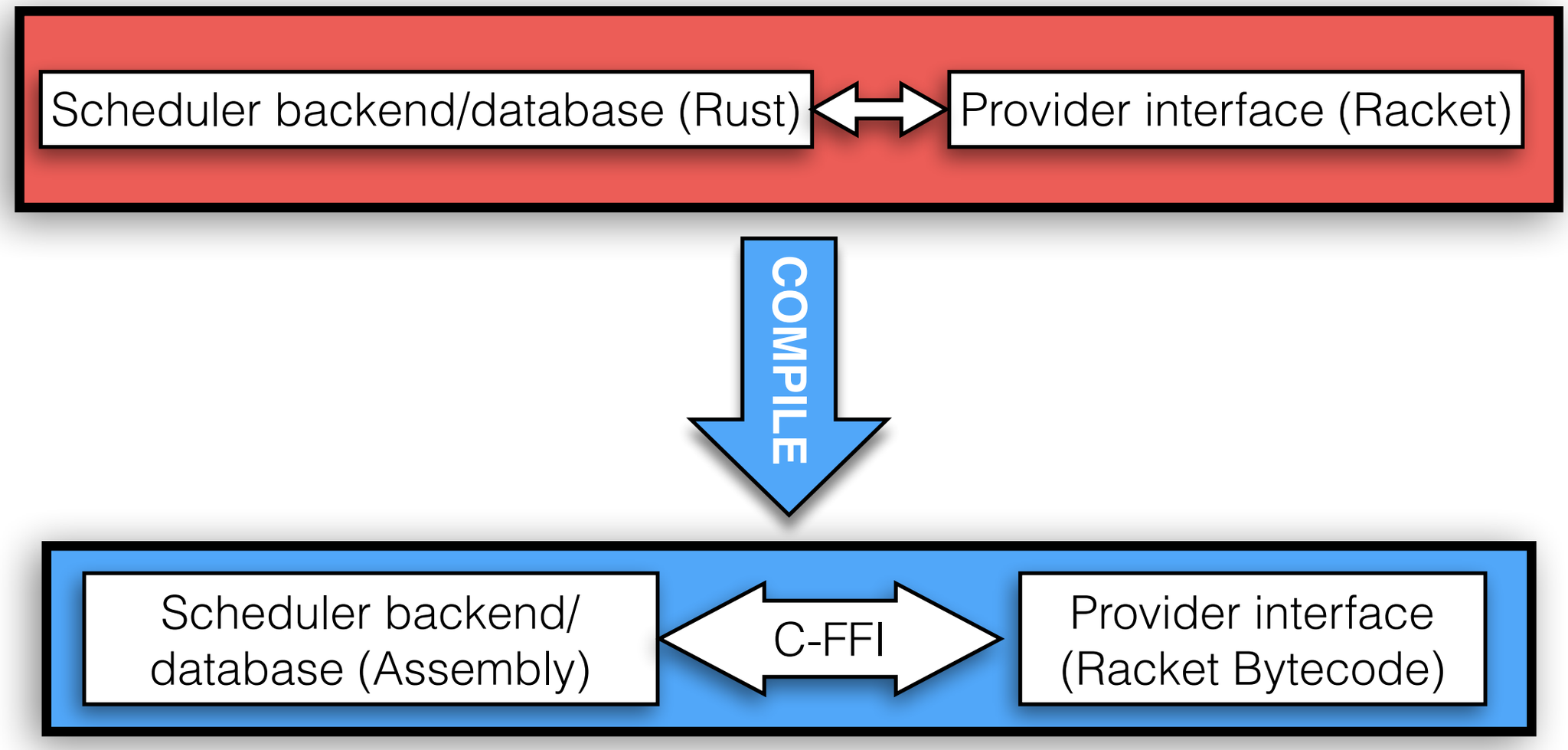
$$
\begin{aligned}
C^{\text{counter}} = \quad & \text{let } v = \text{ref } 0 \text{ in} \\
& \text{let } c'\,() = v := !v + 1;\ !v \text{ in} \\
& [\cdot]\, c'
\end{aligned}
$$

$$C^{\text{counter}}[\lambda \mathbf{c}.\, \mathbf{c}()] \Downarrow 1 \qquad C^{\text{counter}}[\lambda \mathbf{c}.\, \mathbf{c}(); \mathbf{c}()] \Downarrow 2$$

But still reason (almost) in λ ?

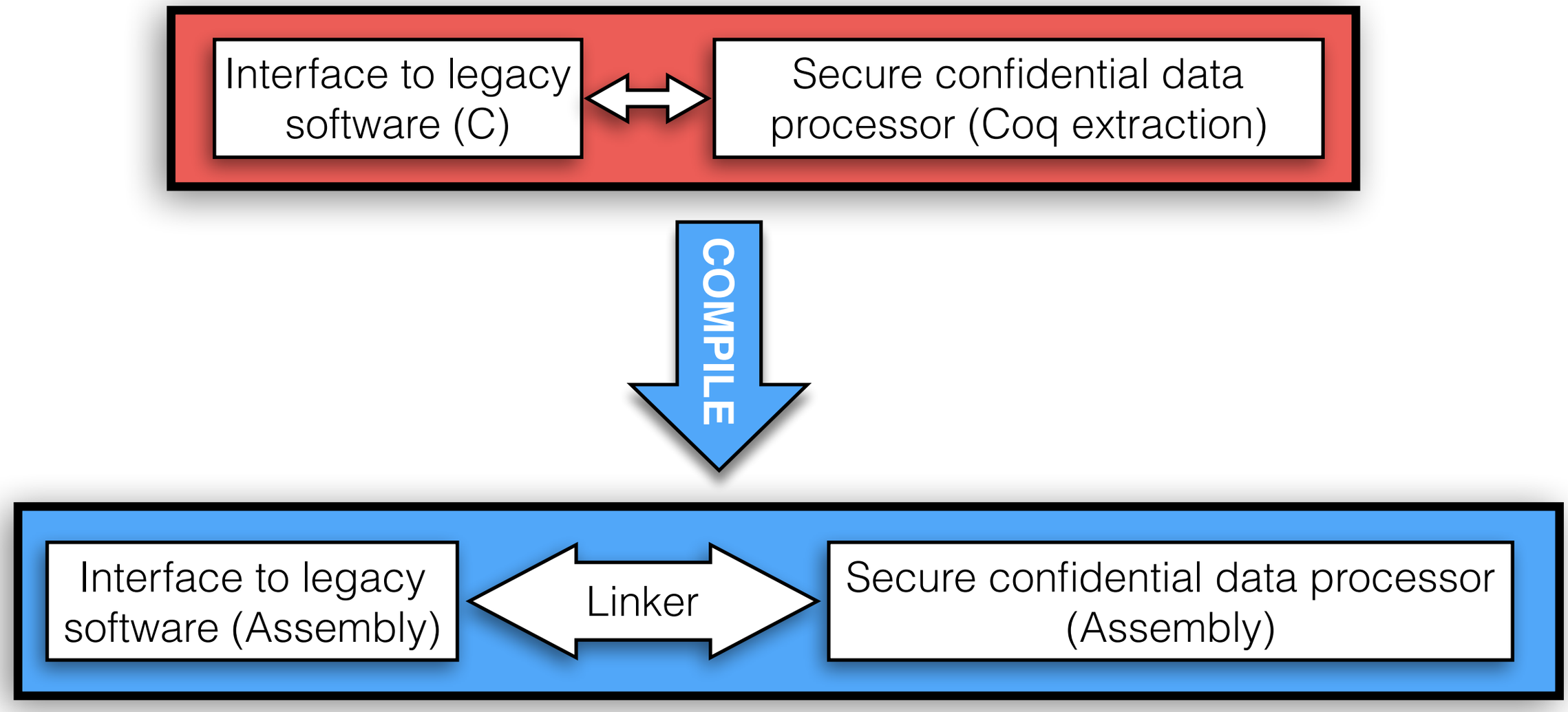
Without accounting for linking, language specifications are incomplete!

How do programmers reason about this?



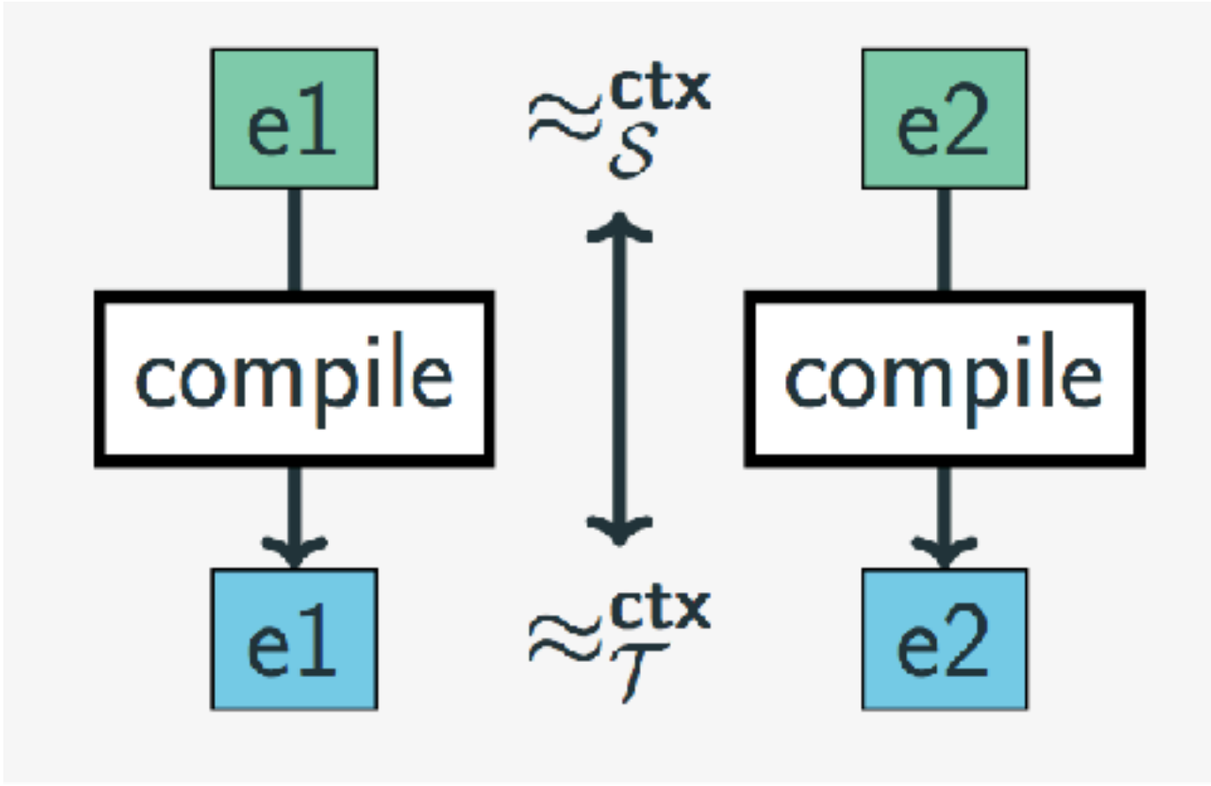
Problem: C-FFI does not respect Rust's memory invariants, so Racket can violate Rust programmers reasoning.

How do programmers reason about this?



Problem: Coq extractions remove proof obligations, so can be called with invalid arguments.

Fully abstract compilation



Programmers can reason in their own language.

What if the other language can do things inexpressible in this source?

Linking types allow programmers to reason about programs in the presence of linking.

When are these equivalent?

```
program A — λf : int → int. 1
```

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

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

$$(\mathsf{int} \to \mathsf{R}^\bullet\mathsf{int}) \to \mathsf{R}^\bullet\mathsf{int}$$

Challenges for linking types

- **Building fully abstract compilers**
- **Effect masking (to limit annotation)**
- **Designing richly typed intermediate languages**

More Details

- Languages may have multiple linking types extensions, which programmers can opt-in to depending on needs.
- Fully abstract compilers from linking types extended languages to rich target language that is medium for linking — regular compilers can be used beyond.

Benefits of linking types

- Language specification includes linking
- Low burden — opt-in annotations
- Extensions reflect features in natural way
- Multiple extensions for same language
- Backwards compatible compilers

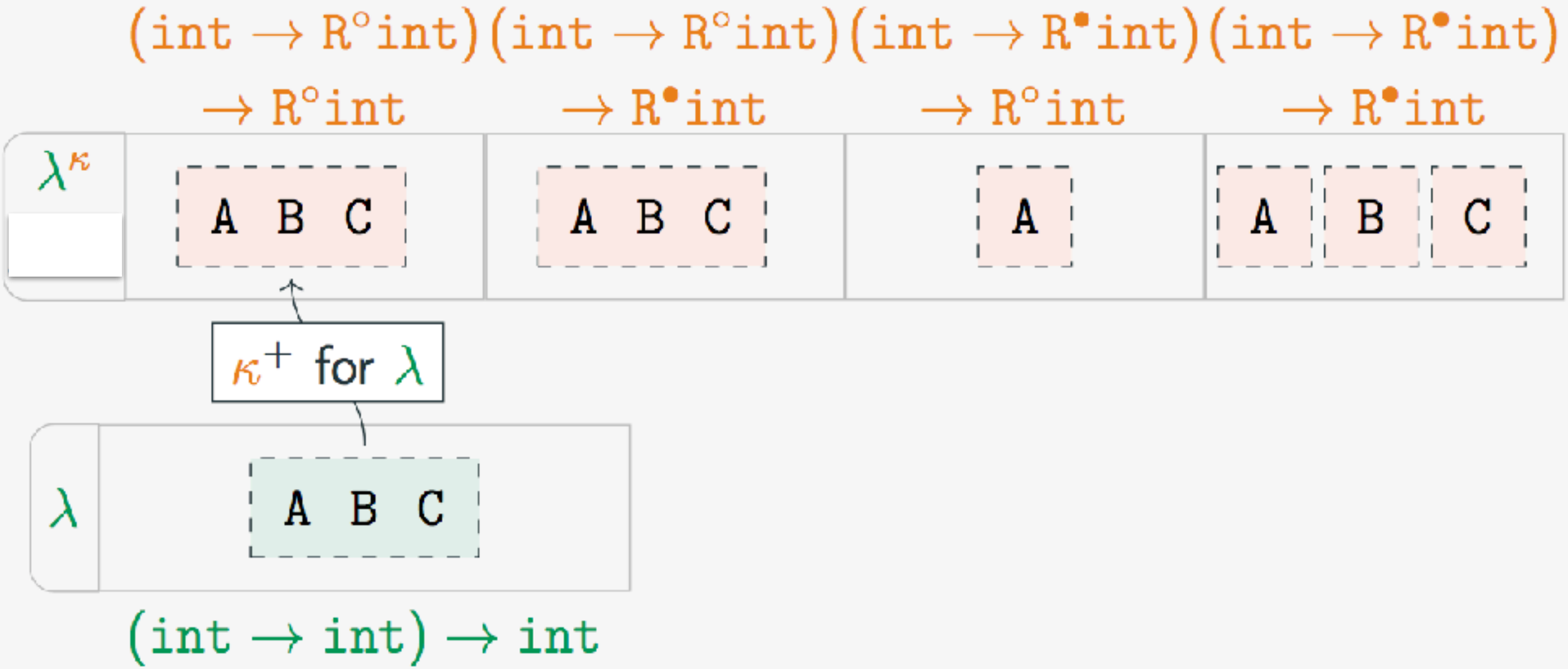
And we shouldn't be able to write new programs.

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

$$
\forall \tau.\, \mathbf{e} : \tau \implies \mathbf{e} : \kappa^-(\tau)
$$

What we can do is change equivalences

$$\texttt{program A} - \lambda \texttt{f} : \texttt{int} \to \texttt{int}.\ \texttt{1}$$
$$\texttt{program B} - \lambda \texttt{f} : \texttt{int} \to \texttt{int}.\ \texttt{f 0; 1}$$
$$\texttt{program C} - \lambda \texttt{f} : \texttt{int} \to \texttt{int}.\ \texttt{f 0; f 0; 1}$$



Let’s live in a world where programmers can use many (typed) languages, each suited to the task at hand.

Maybe programmers should even be able to easily create them:

Turnstile, a metalanguage for creating typed embedded languages.

Stephen Chang, Alex Knauth, Ben Greenman.
POPL 2017

These are types that describe behavior that does not exist in our language.

These **types** let us statically reason about **linking**.

Linking types allow our tools to provide **cross-language type errors**.