Correct and Secure Compilation for Multi-Language Software

Amal Ahmed

Northeastern University

Compiler Correctness

$$s \rightsquigarrow t \quad \Longrightarrow \quad s \approx t$$

compiles to

same behavior

Semantics-preserving compilation

$$s \underset{\substack{\uparrow \\ \text{compiles to}}}{\leadsto} t \quad \Longrightarrow \quad s \underset{\substack{\uparrow \\ \text{same behavior}}}{\approx} t$$

Range of Compiler Properties…

- Type-preserving compilation (90s)
- Semantics-preserving compilation (00s…)

  = *Correct compilation*
- Fully abstract compilation

  = Equivalence-preserving and -reflecting

  = *Secure compilation*
- Security-preserving compilation
  - preserving "security types" vs. preserving noninterference

Compiler Verification

One of the “big problems” of computer science

- since *McCarthy and Painter 1967: Correctness of a Compiler for Arithmetic Expressions*
- see *Dave 2003: Compiler Verification: A Bibliography*

Compiler Verification since 2006…

Leroy ’06 : Formal certification of a compiler back-end or: programming a compiler with a proof assistant.

Lochbihler ’10 : Verifying a compiler for Java threads.

Myreen ’10 : Verified just-in-time compiler on x86.

Sevcik et al.’11: Relaxed-memory concurrency and verified compilation.

Zhao et al.’13 : Formal verification of SSA-based optimizations for LLVM

Kumar et al.’14 : CakeML: A verified implementation of ML

⋮

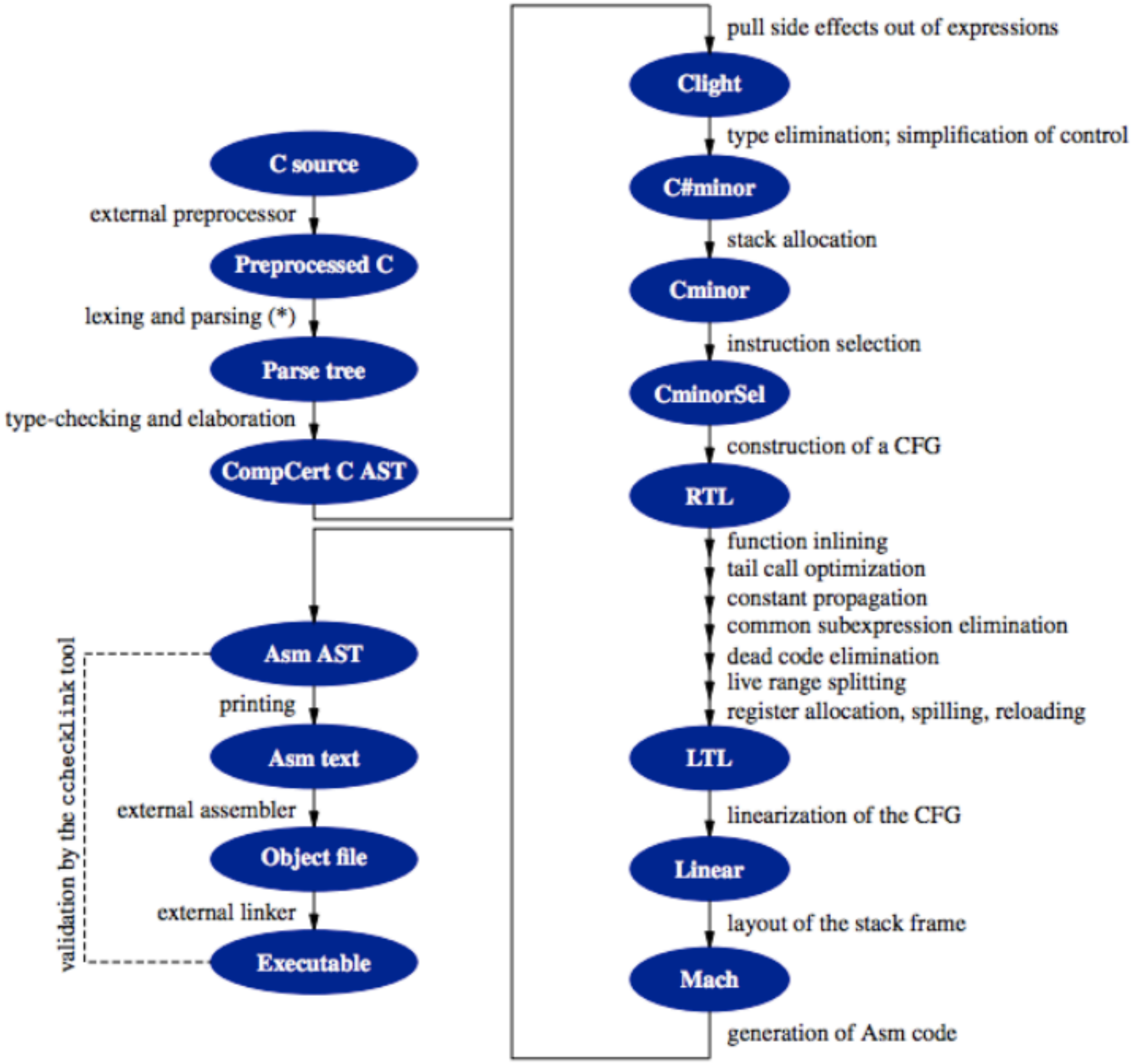
Why CompCert had such impact...

- Demonstrated that realistic verified compilers are both *feasible* and bring *tangible benefits*

> The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users. **(Yang et al. PLDI 2011)**

Why CompCert had such impact…

- Provided a proof architecture for others to follow/build on
    - CompCert memory model, uniform across passes
    - proof using simulations



C source
external preprocessor
Preprocessed C
lexing and parsing (*)
Parse tree
type-checking and elaboration
CompCert C AST

pull side effects out of expressions
Clight
type elimination; simplification of control
C#minor
stack allocation
Cminor
instruction selection
CminorSel
construction of a CFG
RTL
function inlining
tail call optimization
constant propagation
common subexpression elimination
dead code elimination
live range splitting
register allocation, spilling, reloading
LTL
linearization of the CFG
Linear
layout of the stack frame
Mach
generation of Asm code

Asm AST
printing
Asm text
external assembler
Object file
external linker
Executable

validation by the `ccheckl ink` tool

Not verified yet

(*) the parser is formally verified

Formally verified

[CompCert manual 2015]

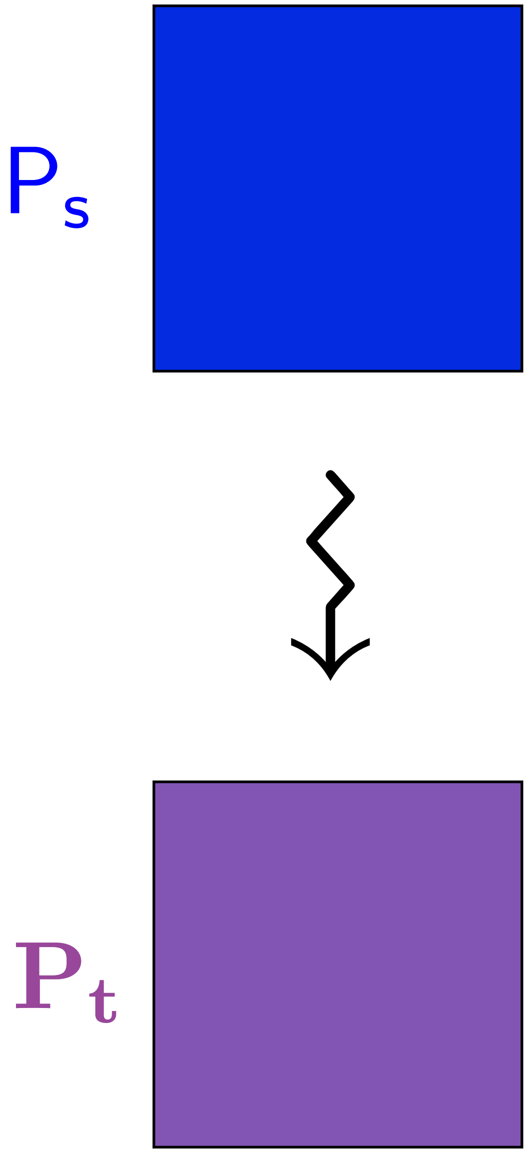
Why CompCert had such impact…

- Provided a proof architecture for others to follow/build on
  - CompCert memory model, uniform across passes
  - proof using simulations

But the simplicity of the proof architecture comes at a price…

Problem: Whole-Program Assumption

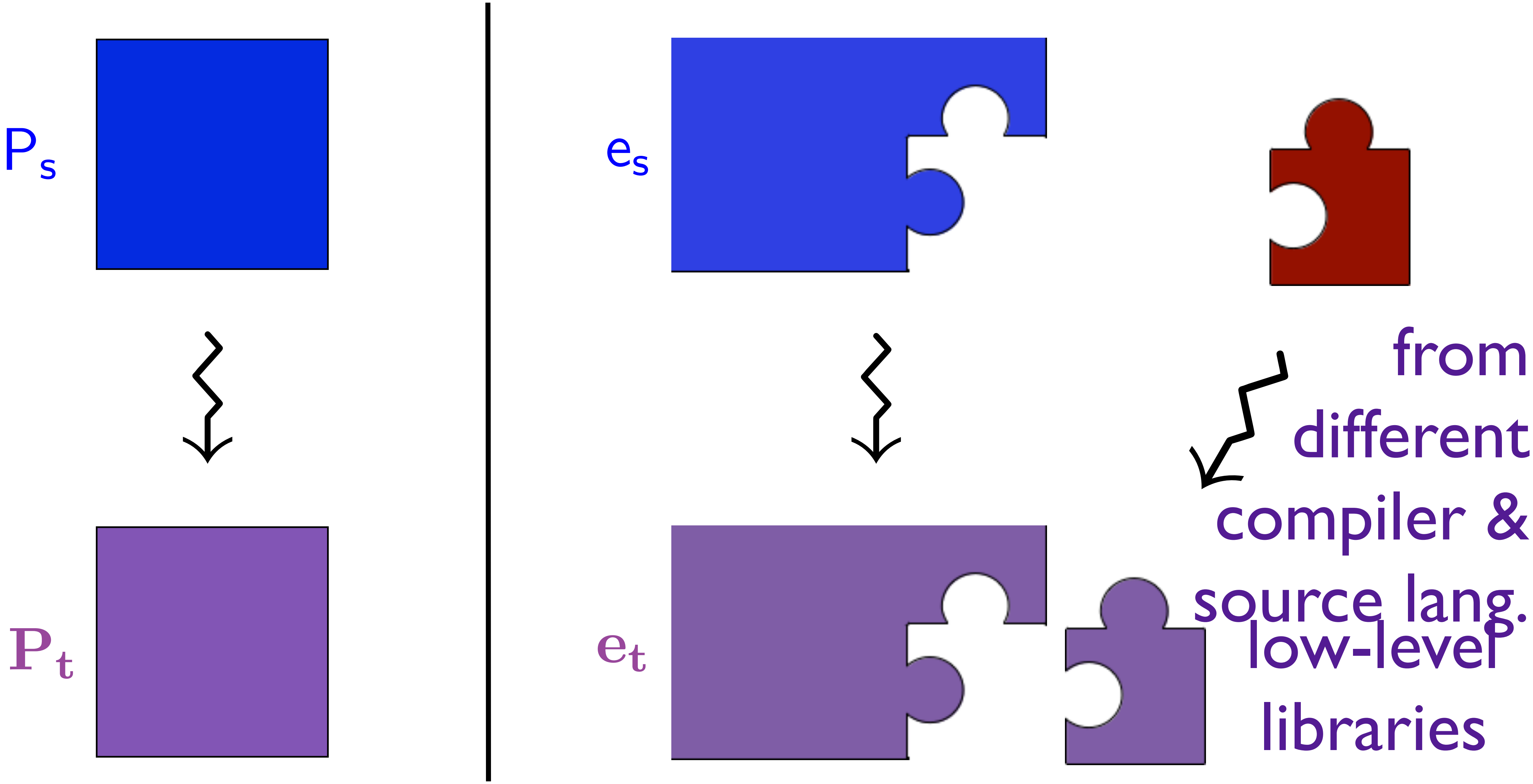
Correct compilation guarantee only applies to whole programs!



CompCert's … "formal guarantees of semantics preservation apply only to whole programs that have been compiled as a whole by [the] CompCert C [compiler]" (Leroy 2014)

Problem: Whole-Program Assumption

Correct compilation guarantee only applies to whole programs!



P_s

P_t

e_s

e_t

from different compiler & source lang.

low-level libraries

Why Whole Programs?

$$s \rightsquigarrow t \qquad \Longrightarrow \qquad s \underset{\substack{\uparrow \\ \text{expressed how?}}}{\approx} t$$

Why Whole Programs?

$$P_s \leadsto P_t \quad \Longrightarrow \quad P_s \approx P_t$$

↑

expressed how?

CompCert

$$\begin{tikzcd}
P_s \arrow[r, mapsto] \arrow[d, dash] & \ldots \arrow[r, mapsto] & P_s^i \arrow[r, mapsto] \arrow[d, dash] & P_s^{i+1} \arrow[r, mapsto] \arrow[d, dash] & \ldots \\
P_t \arrow[r, mapsto] & \ldots \arrow[r, mapsto] & P_t^j \arrow[r, mapsto, "*"] & P_t^{j+n} \arrow[r, mapsto] & \ldots
\end{tikzcd}$$

Proof composes per-pass simulations

$$P_s \rightsquigarrow P_t \implies \begin{tikzcd}
P_s \arrow[r, mapsto] \arrow[d, dash, "R_1"] & \dots \arrow[r, mapsto] & P_s^i \arrow[r, mapsto] \arrow[d, dash, "R_1"] & P_s^{i+1} \arrow[r, mapsto] \arrow[d, dash, "R_1"] & \dots \\
P_t \arrow[r, mapsto] & \dots \arrow[r, mapsto] & P_t^j \arrow[r, mapsto, "{}^*" near end] & P_t^{j+n} \arrow[r, mapsto] & \dots
\end{tikzcd}$$

$$P_t \rightsquigarrow P_u \implies \begin{tikzcd}
P_t \arrow[r, mapsto] \arrow[d, dash, "R_2"] & \dots \arrow[r, mapsto] & P_t^i \arrow[r, mapsto] \arrow[d, dash, "R_2"] & P_t^{i+1} \arrow[r, mapsto] \arrow[d, dash, "R_2"] & \dots \\
P_u \arrow[r, mapsto] & \dots \arrow[r, mapsto] & P_u^j \arrow[r, mapsto, "{}^*" near end] & P_u^{j+n} \arrow[r, mapsto] & \dots
\end{tikzcd}$$

$$P_s \rightsquigarrow P_u \implies P_s \approx P_u$$

Why Whole Programs?

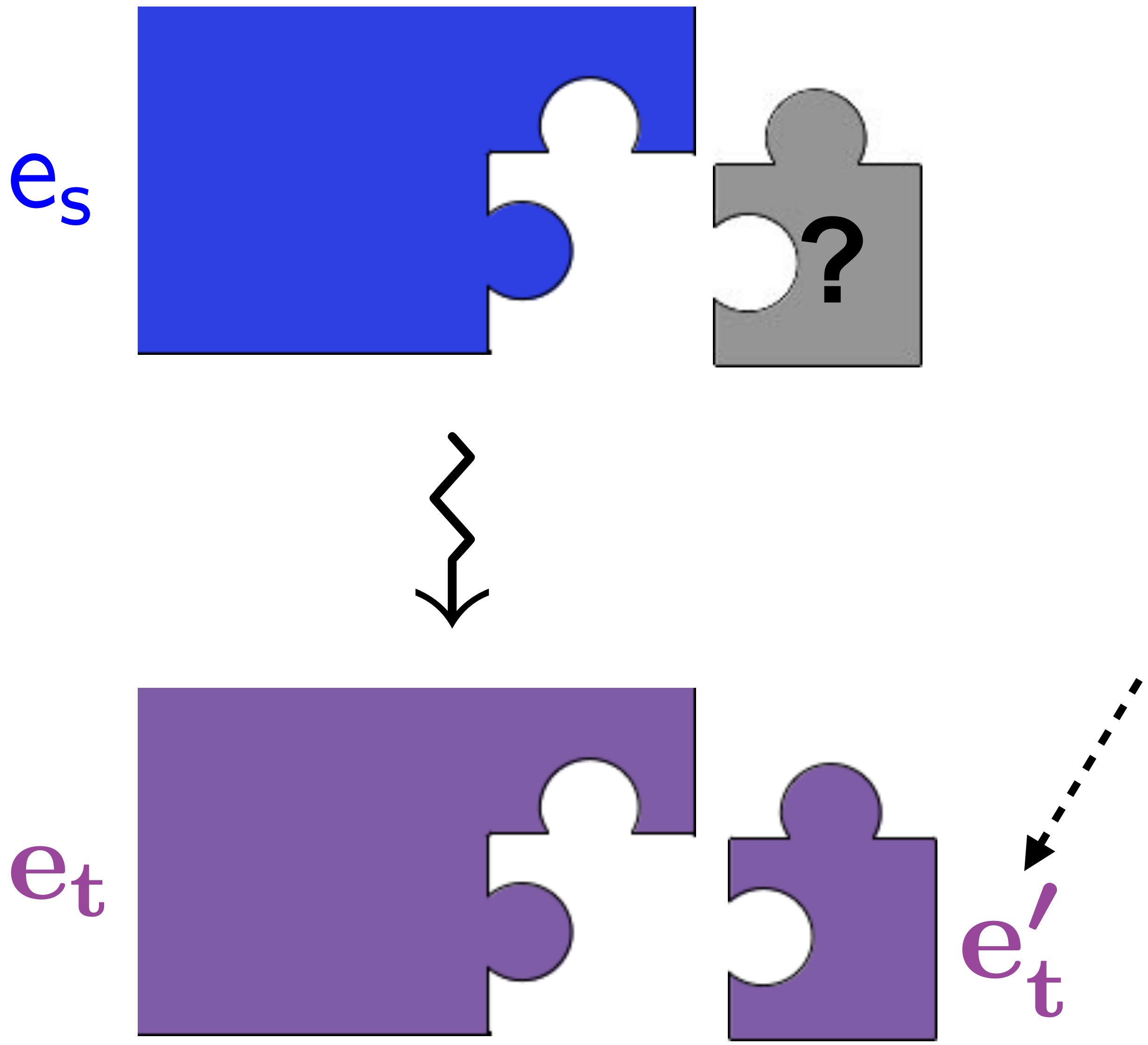
$$P_s \rightsquigarrow P_t \quad \Longrightarrow \quad P_s \underset{\uparrow}{\approx} P_t$$

“closed” simulations

CompCert

$$\begin{tikzcd}
P_s \arrow[r, mapsto] \arrow[d, dash] & \dots \arrow[r, mapsto] & P_s^i \arrow[r, mapsto] \arrow[d, dash] & P_s^{i+1} \arrow[r, mapsto] \arrow[d, dash] & \dots \\
P_t \arrow[r, mapsto] & \dots \arrow[r, mapsto] & P_t^j \arrow[r, mapsto, "{*}"] & P_t^{j+n} \arrow[r, mapsto] & \dots
\end{tikzcd}$$

Correct Compilation of Components?



$$e_S \approx e_T$$

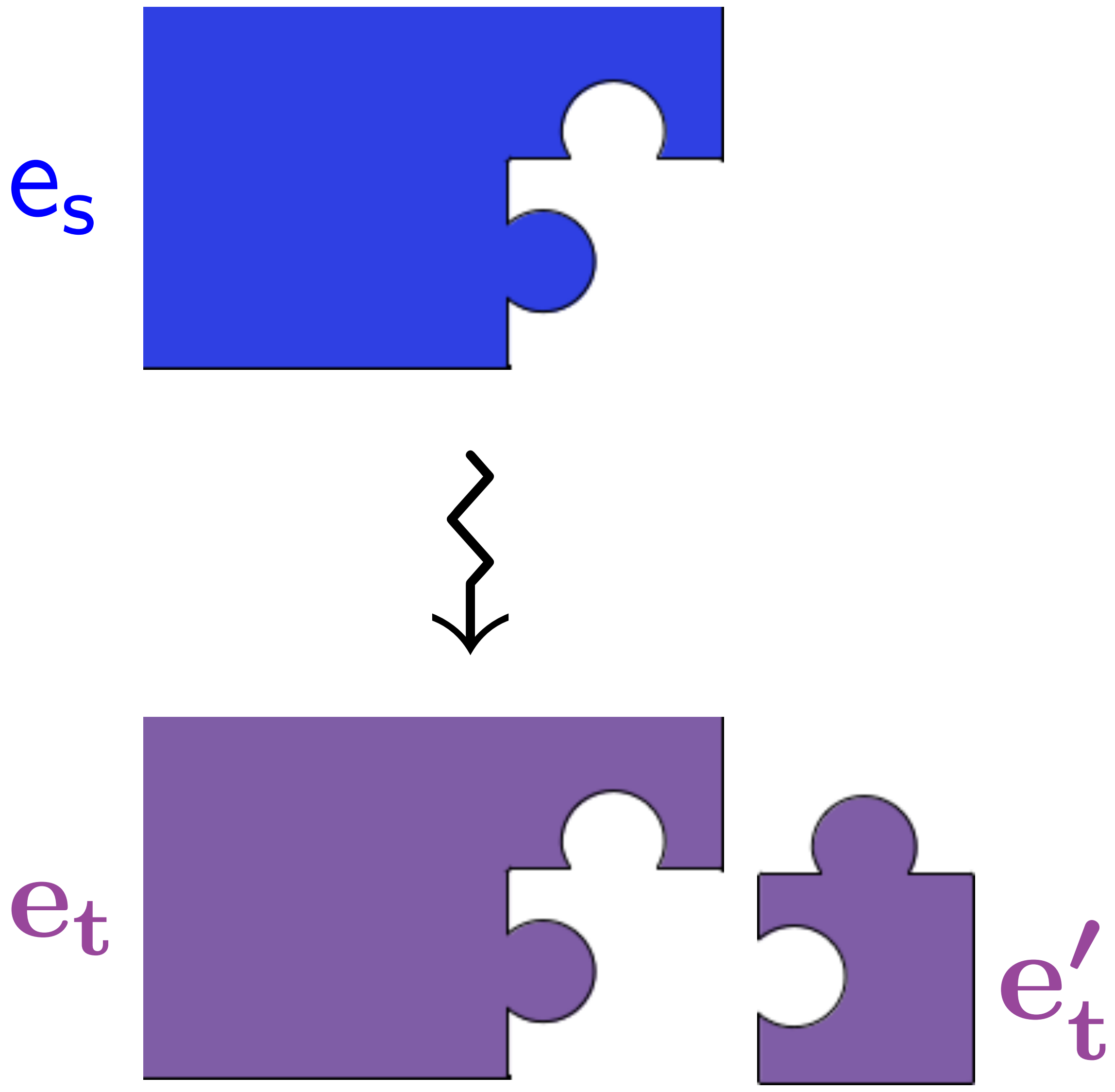
expressed how?

Produced by

- *same compiler,*
- *diff compiler for S,*
- *compiler for diff lang R,*
- *R that's **very** diff from S?*

Behavior expressible in S?

Correct Compilers, Multi-language SW



$e_S \approx e_T$

Definition should:

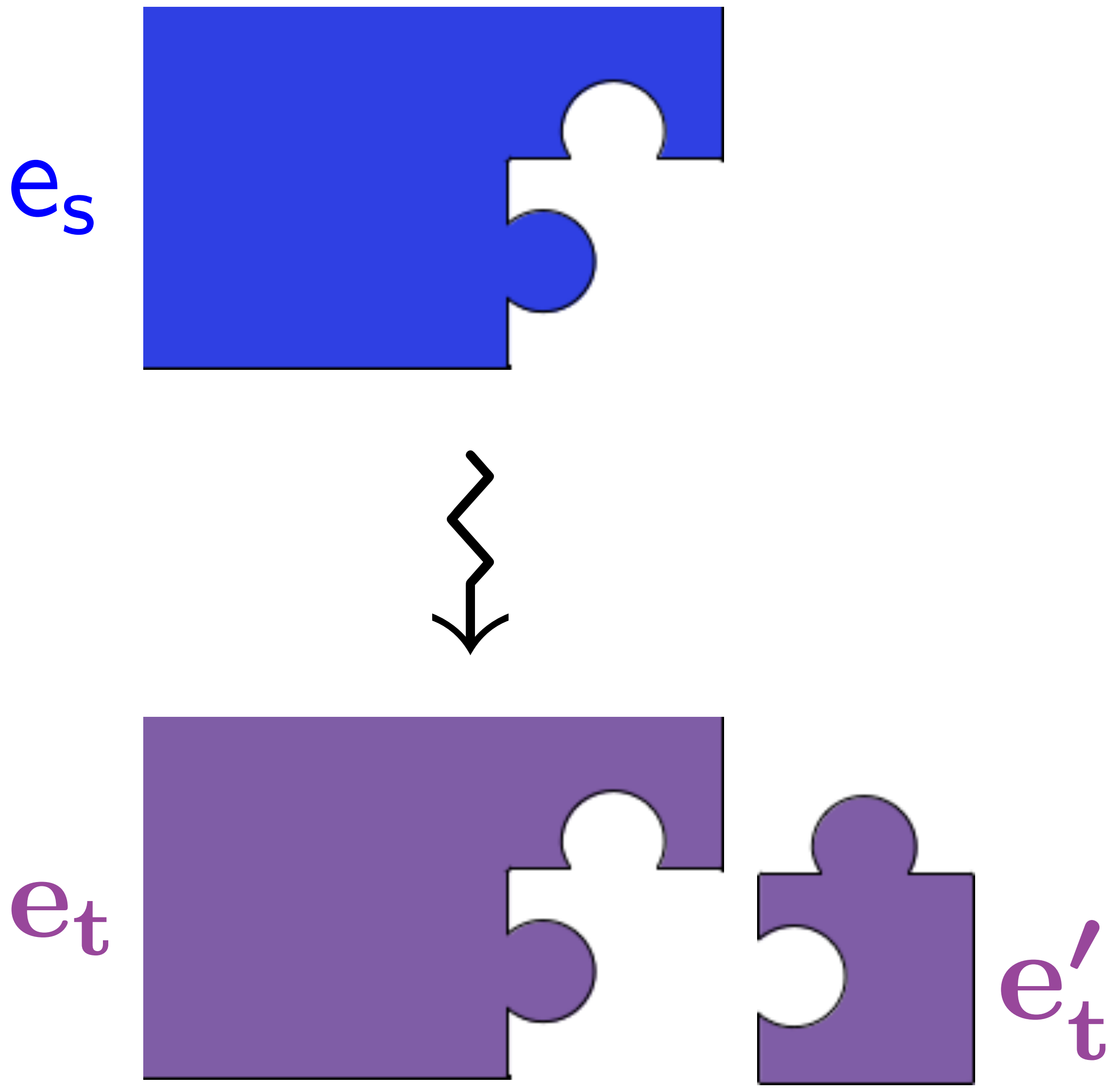
- permit **linking** with target code of arbitrary provenance
- support verification of **multi-pass** compilers

Plan

- Survey the literature: how to express $e_S \approx e_T$
  - "compositional" compiler correctness

    = correct compilation of components

Compositional Compiler Correctness



$$e_S \approx e_T$$

Dictates:

- what we can **link** with *(horizontal compositionality)* and how to check it's okay to link
- effort involved in proving *transitivity* for **multi-pass** compilers *(vertical compositionality)*

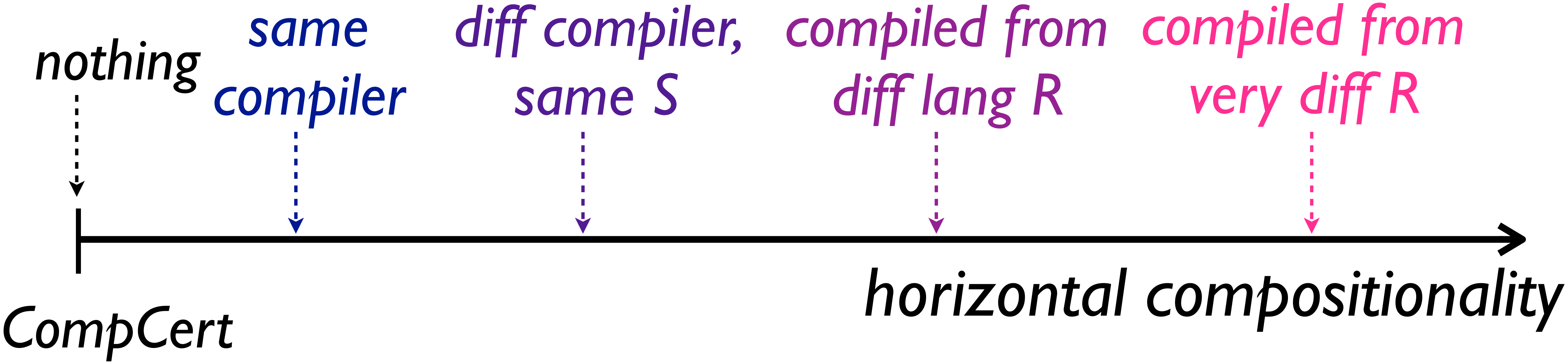
Plan

- Survey the literature: how to express $e_S \approx e_T$
- How does the choice affect:
    - what we can link with
    - how we check if some e'_t is okay to link with
    - effort required to prove transitivity
    - effort required to have confidence in theorem statement
- How to support linking with code from ***very*** different R

Plan

- Survey the literature: how to express $e_S \approx e_T$
- How does the choice affect:
    - what we can link with
    - how we check if some e'_t is okay to link with
    - effort required to prove transitivity
    - effort required to have confidence in theorem statement
- How to support linking with code from ***very*** different R
- Type-preserving compilation
- Secure (fully abstract) compilation

What we can link with



nothing

same compiler

diff compiler, same S

compiled from diff lang R

compiled from very diff R

horizontal compositionality

CompCert

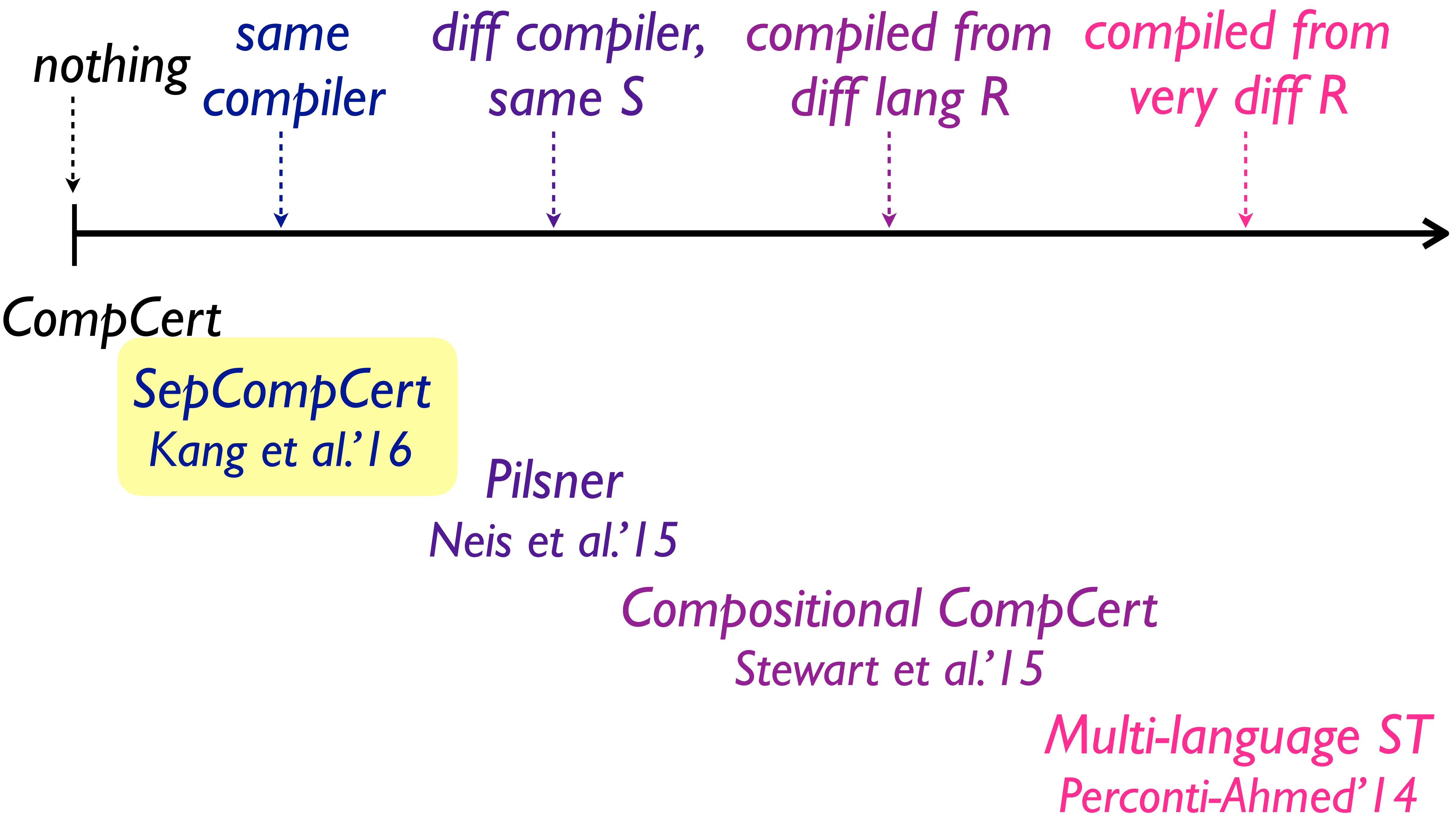
SepCompCert
Kang et al.'16

Pilsner
Neis et al.'15

Compositional CompCert
Stewart et al.'15

Multi-language ST
Perconti-Ahmed'14

What we can link with



nothing

same compiler

diff compiler, same S

compiled from diff lang R

compiled from very diff R

CompCert

SepCompCert
Kang et al.'16

Pilsner
Neis et al.'15

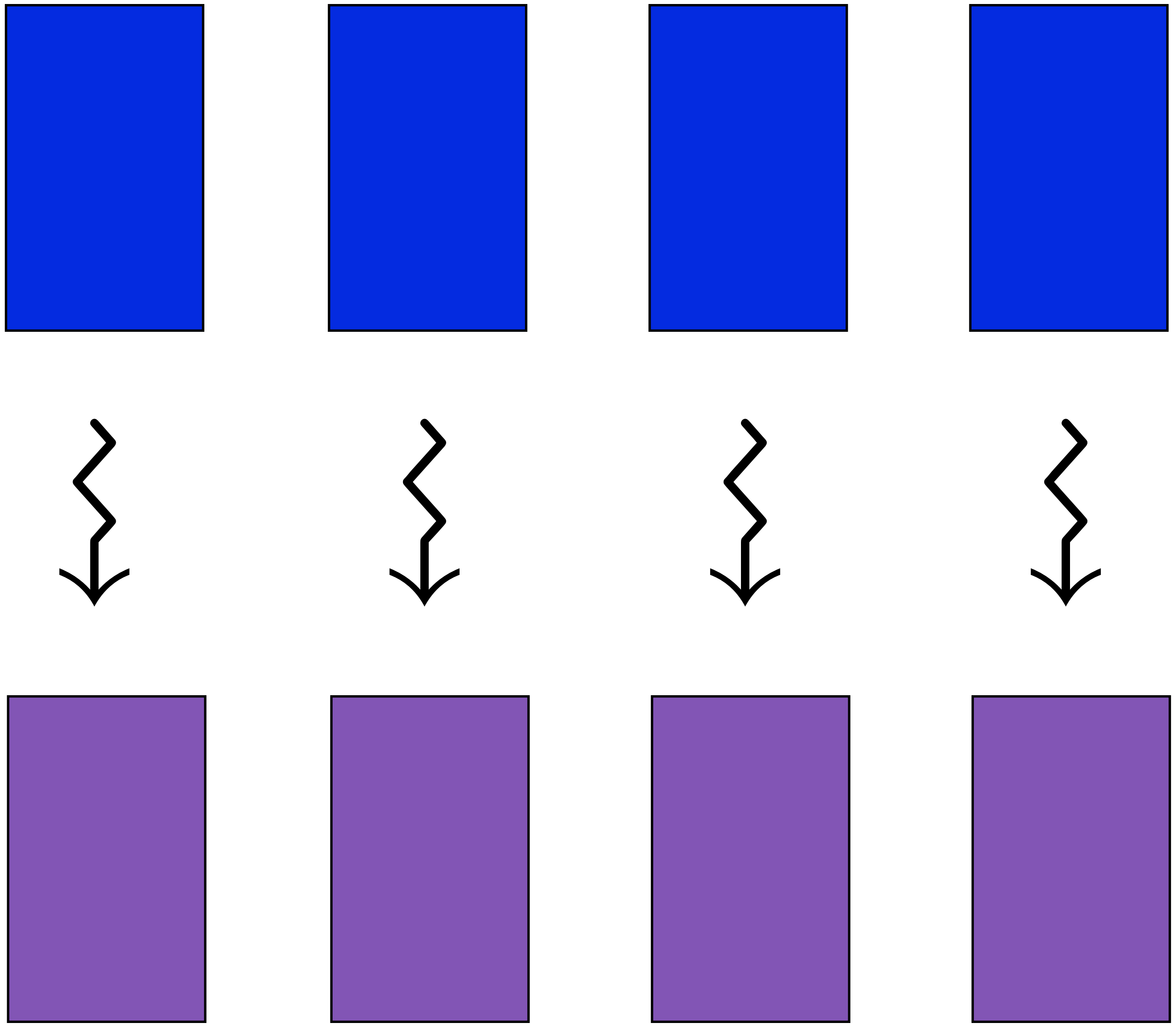
Compositional CompCert
Stewart et al.'15

Multi-language ST
Perconti-Ahmed'14

Approach: Separate Compilation (C)

SepCompCert

[Kang et al. '16]

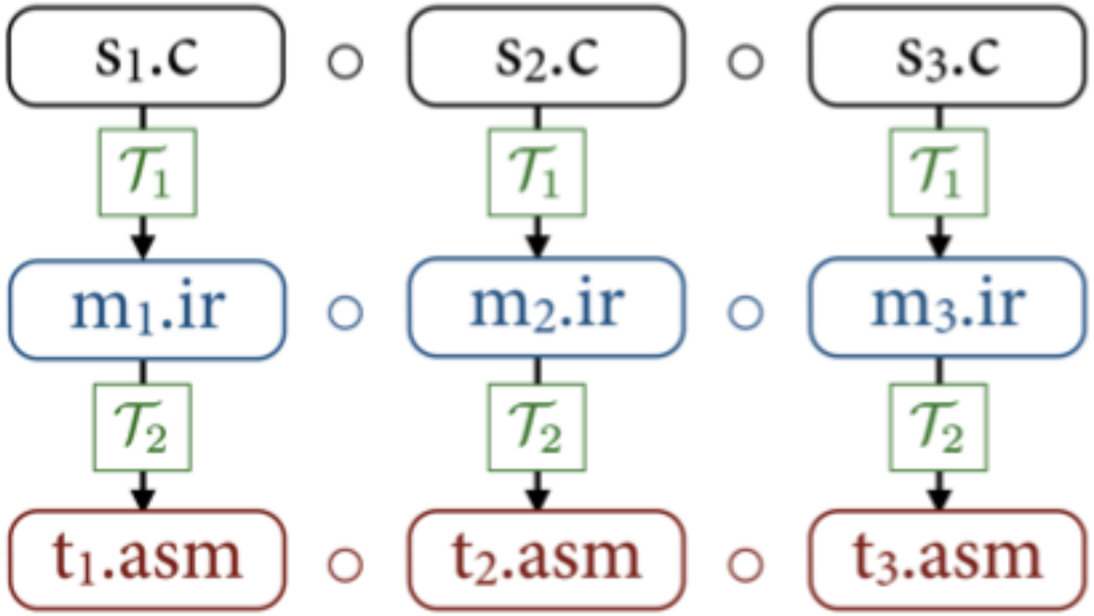


Approach: Separate Compilation (C)

SepCompCert

[Kang et al. '16]

Level A correctness



s_1.c ∘ s_2.c ∘ s_3.c

$\mathcal{T}_1$ $\mathcal{T}_1$ $\mathcal{T}_1$

m_1.ir ∘ m_2.ir ∘ m_3.ir

$\mathcal{T}_2$ $\mathcal{T}_2$ $\mathcal{T}_2$

t_1.asm ∘ t_2.asm ∘ t_3.asm

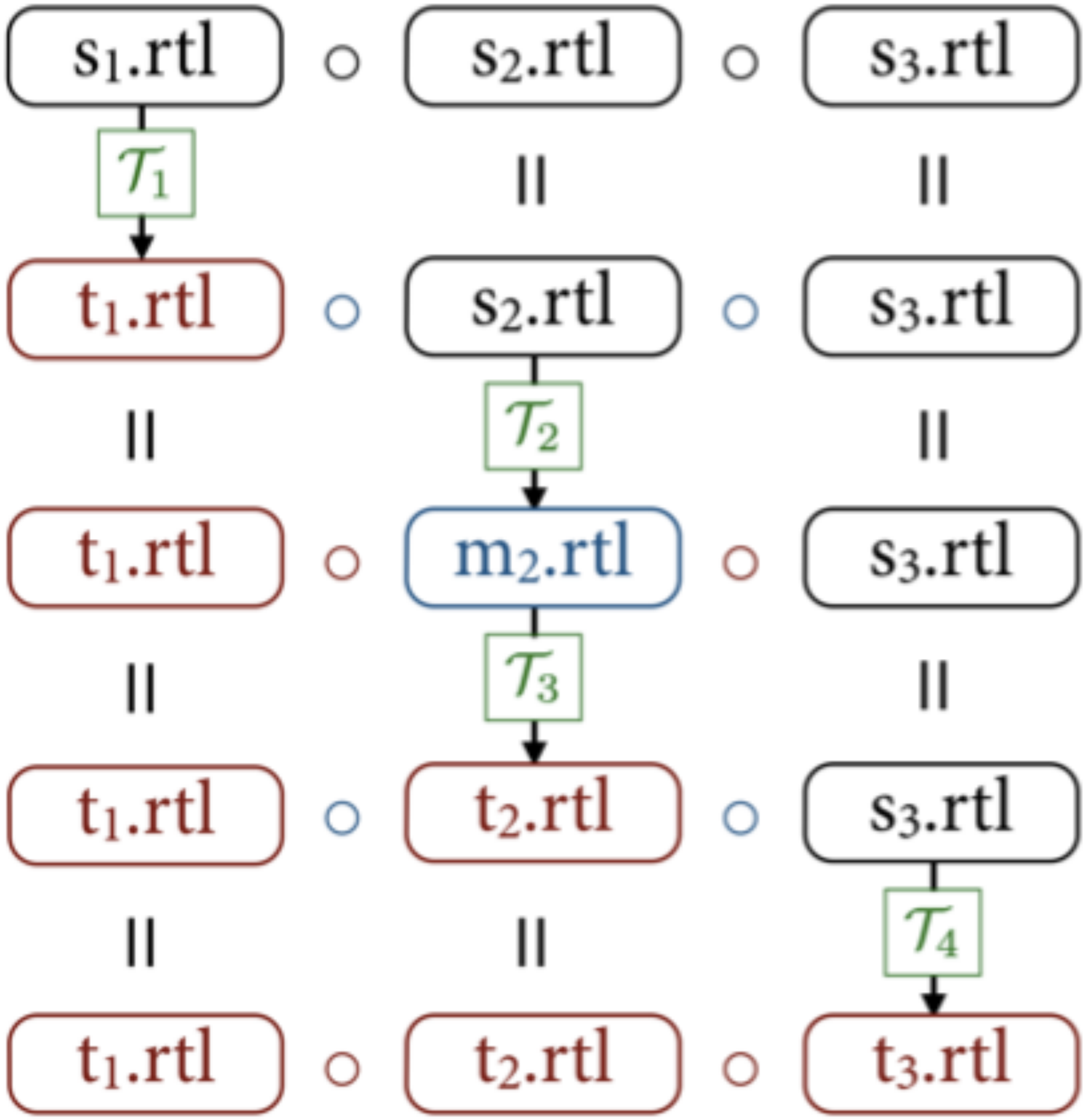
End-to-end

$$\frac{\begin{array}{c}\forall i \in \{1 \ldots n\}.\ \mathcal{C}(\mathtt{s}_i\mathtt{.c}) = \mathtt{t}_i\mathtt{.asm} \\ s = \mathrm{load}(\mathtt{s}_1\mathtt{.c} \circ \ldots \circ \mathtt{s}_n\mathtt{.c}) \qquad t = \mathrm{load}(\mathtt{t}_1\mathtt{.asm} \circ \ldots \circ \mathtt{t}_n\mathtt{.asm})\end{array}}{\mathrm{Behav}(s) \supseteq \mathrm{Behav}(t)}$$

Approach: Separate Compilation (C)

SepCompCert
[Kang et al. '16]

Level B correctness: omit some RTL optimizations



Approach: Separate Compilation (C)

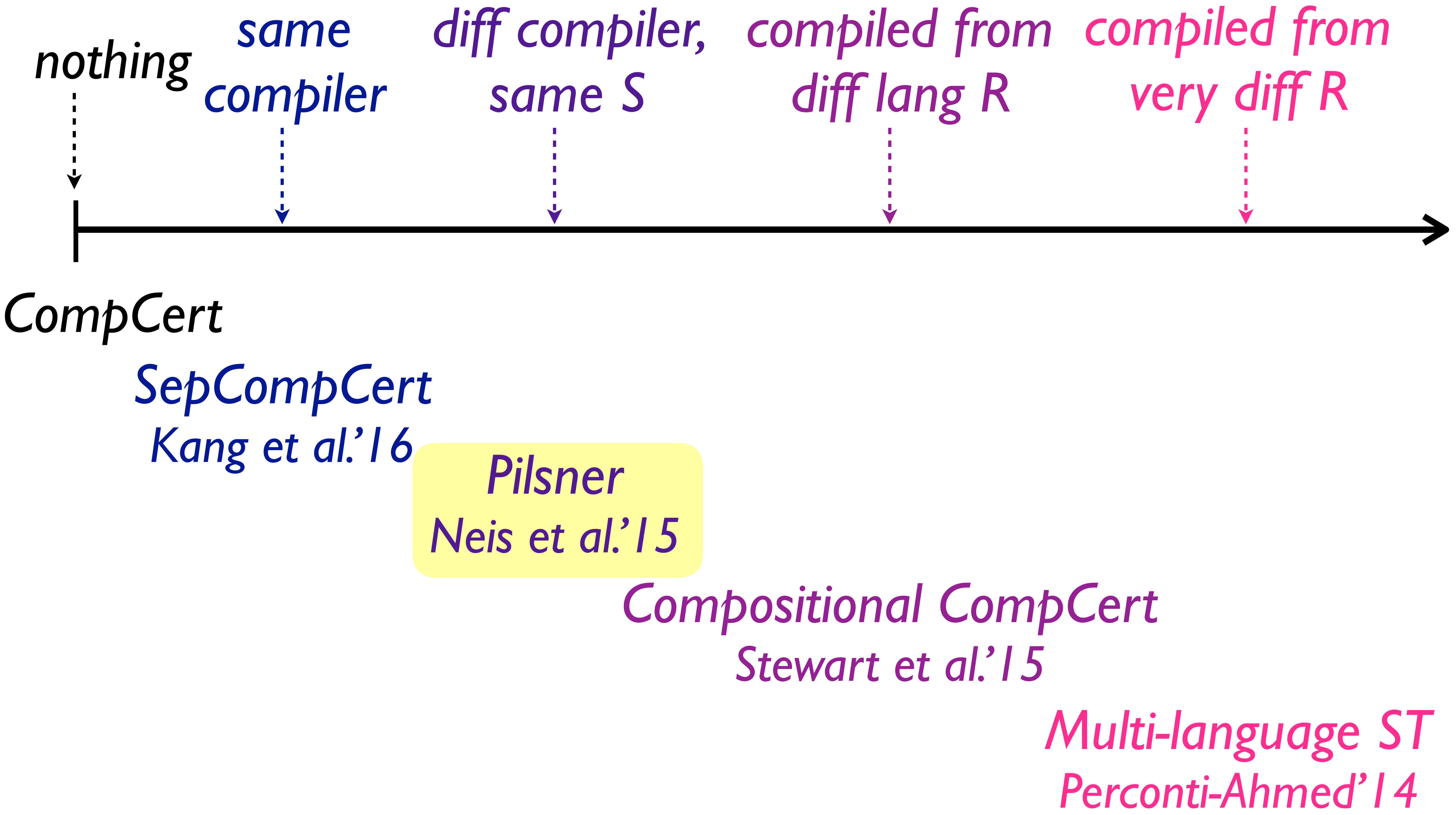
SepCompCert
[Kang et al. '16]

Level B correctness: omit some RTL optimizations

End-to-end

$$\frac{\forall i \in \{1 \ldots n\}.\ \mathcal{C}_i(\mathbf{s}_i.\mathbf{c}) = \mathbf{t}_i.\mathbf{asm} \qquad s = \mathrm{load}(\mathbf{s}_1.\mathbf{c} \circ \ldots \circ \mathbf{s}_n.\mathbf{c}) \qquad t = \mathrm{load}(\mathbf{t}_1.\mathbf{asm} \circ \ldots \circ \mathbf{t}_n.\mathbf{asm})}{\mathrm{Behav}(s) \supseteq \mathrm{Behav}(t)}$$

What we can link with



nothing

same compiler

diff compiler, same S

compiled from diff lang R

compiled from very diff R

CompCert

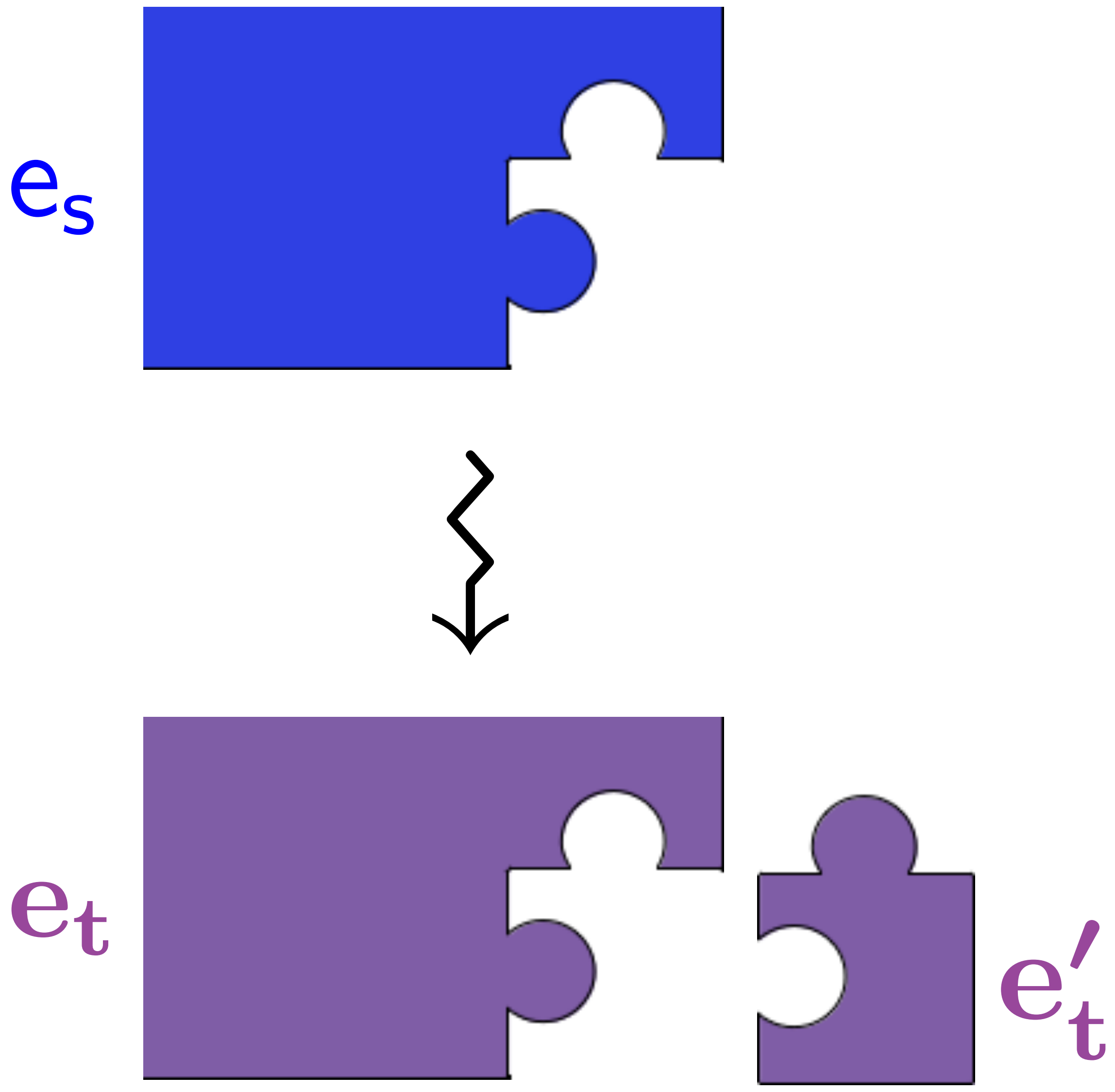
SepCompCert
Kang et al.'16

Pilsner
Neis et al.'15

Compositional CompCert
Stewart et al.'15

Multi-language ST
Perconti-Ahmed'14

Approach: Cross-Language Relations



Cross-language relation

$$e_S \approx e_T$$

Compiling ML-like langs:

Logical relations

- *[Benton-Hur ICFP'09]*
- *[Hur-Dreyer POPL'11]*

Parametric inter-language simulations (PILS)

- *[Neis et al. ICFP'15]*

Case Study: Closure Conversion

- Typed Closure Conversion

- Correctness of closure conversion
  using a cross-language logical relation...

[on board]

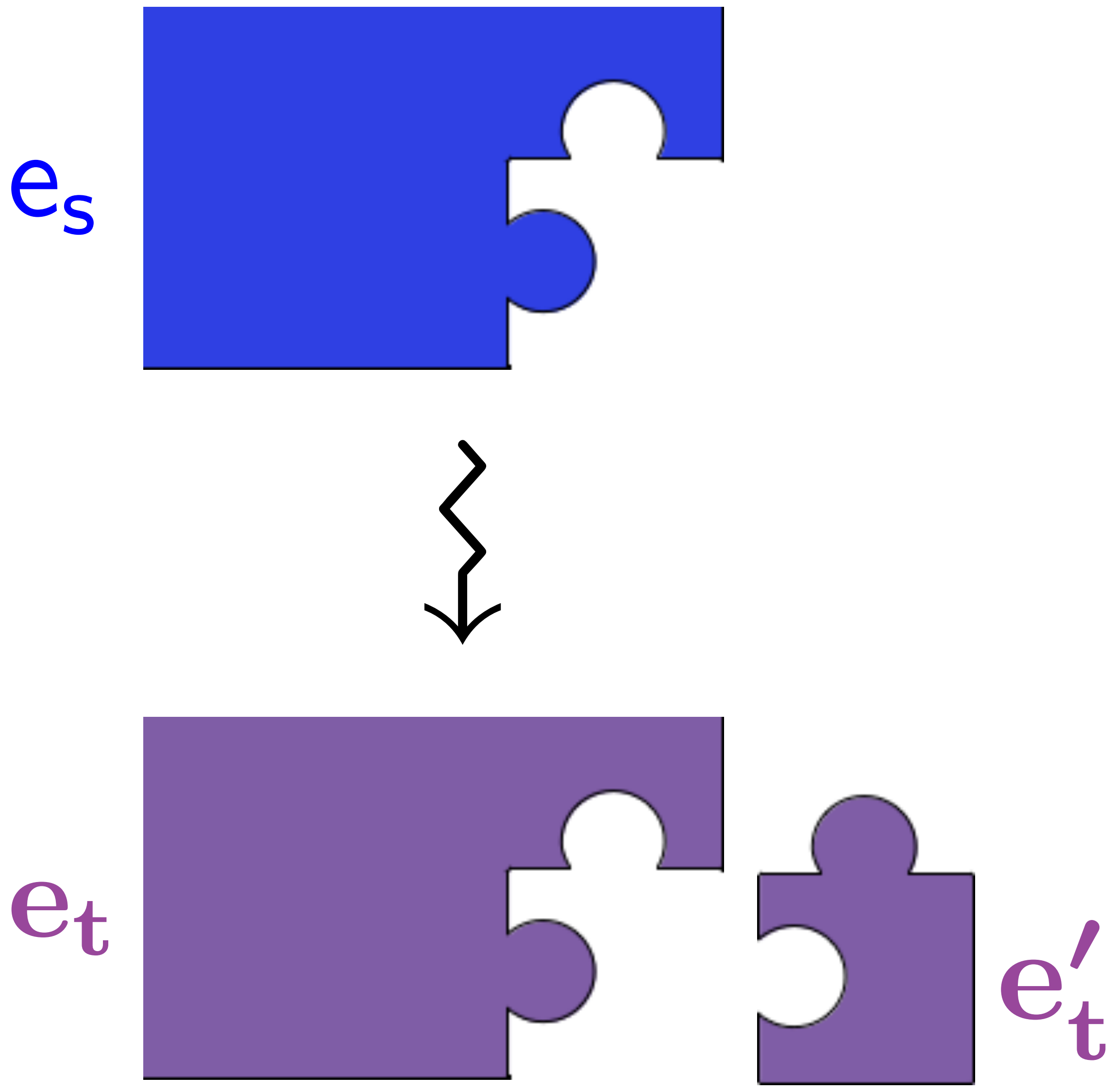
Cross-Language Relation: Problem 1

$$\mathsf{x} : \tau' \vdash \mathsf{e_s} : \tau \rightsquigarrow \mathbf{e_t} \quad \Longrightarrow \quad \mathsf{x} : \tau' \vdash \mathsf{e_s} \simeq \mathbf{e_t} : \tau$$

cross-language logical relation

$$\forall \mathsf{e_s'}, \mathbf{e_s'}.\ \vdash \mathsf{e_s'} \simeq \mathbf{e_t'} : \tau' \quad \Longrightarrow \quad \vdash \mathsf{e_s}[\mathsf{e_s'}/\mathsf{x}] \simeq \mathbf{e_t}[\mathbf{e_t'}/\mathbf{x}] : \tau$$

Cross-Language Relation: Problem 1



Have $x : \tau' \vdash e_s \simeq e_t : \tau$

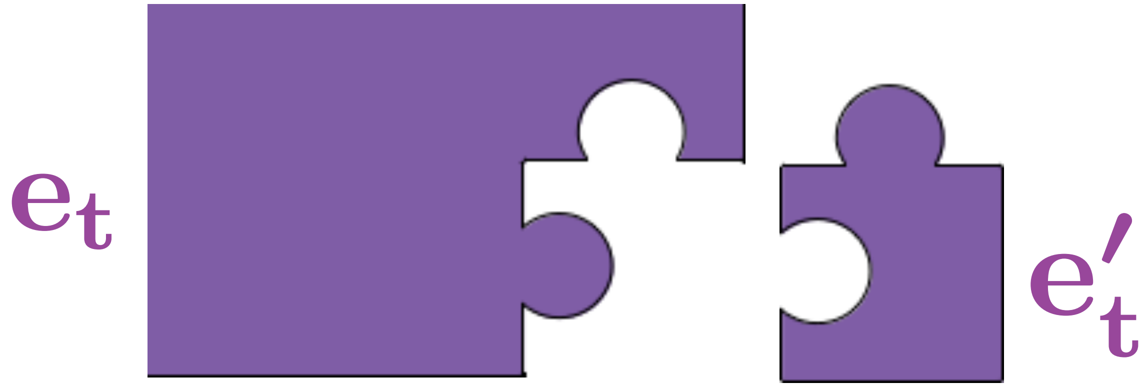
Does the compiler correctness theorem permit linking with e'_t *?*

Cross-Language Relation: Problem 1

Have $x : \tau' \vdash e_s \simeq e_t : \tau$

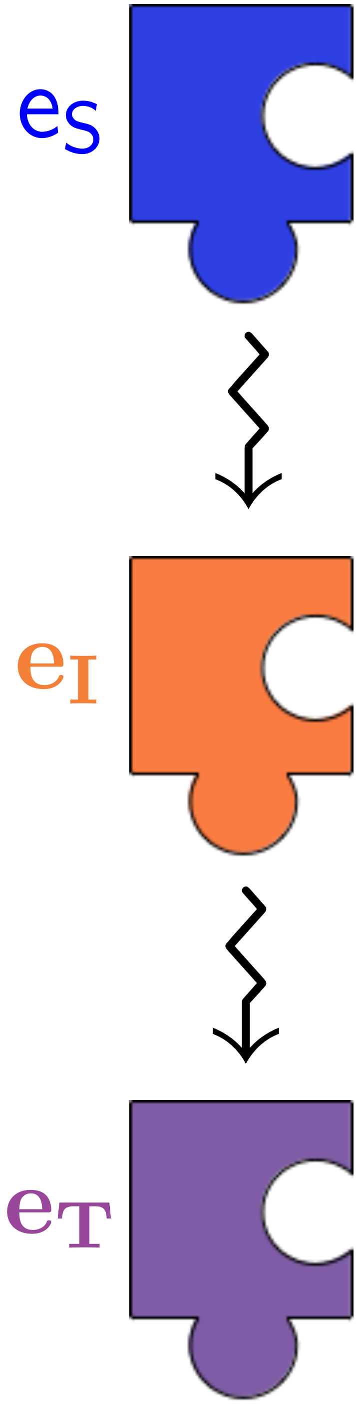
cross-language logical relation

$$\forall e_s', e_s'.\ \vdash e_s' \simeq e_t' : \tau' \implies \vdash e_s[e_s'/x] \simeq e_t[e_t'/x] : \tau$$



- Need to come up with e_s' -- not feasible in practice!
- Cannot link with e_t' whose behavior cannot be expressed in source.

Cross-Language LR: Problem 2

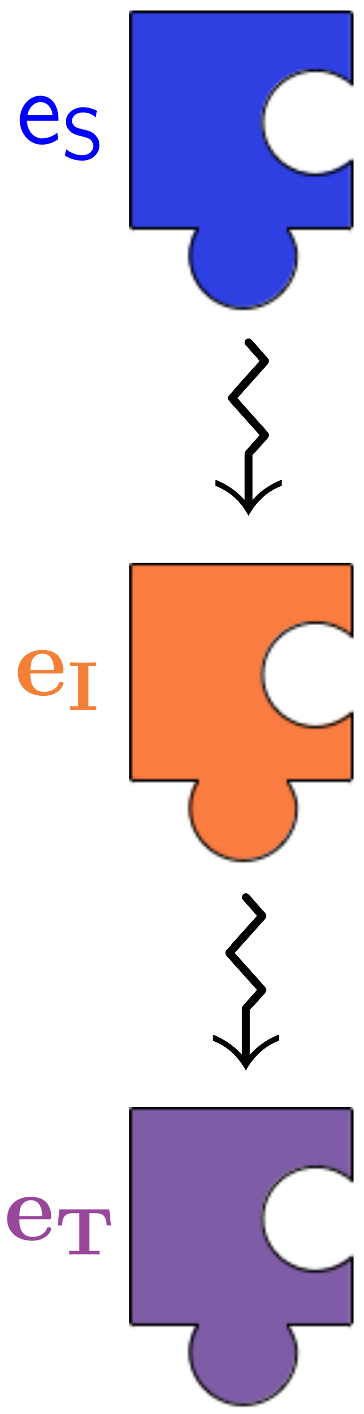


$$\left.\begin{array}{l} e_S \simeq e_I \\ \\ e_I \simeq e_T \end{array}\right\} \overset{?}{\Longrightarrow} \quad e_S \simeq e_T$$

Transitivity for single-lang. logical relation?

$$\left.\begin{array}{l} e_1 \approx e_2 \\ \\ e_2 \approx e_3 \end{array}\right\} \overset{?}{\Longrightarrow} \quad e_1 \approx e_3$$

Cross-Language LR: Problem 2



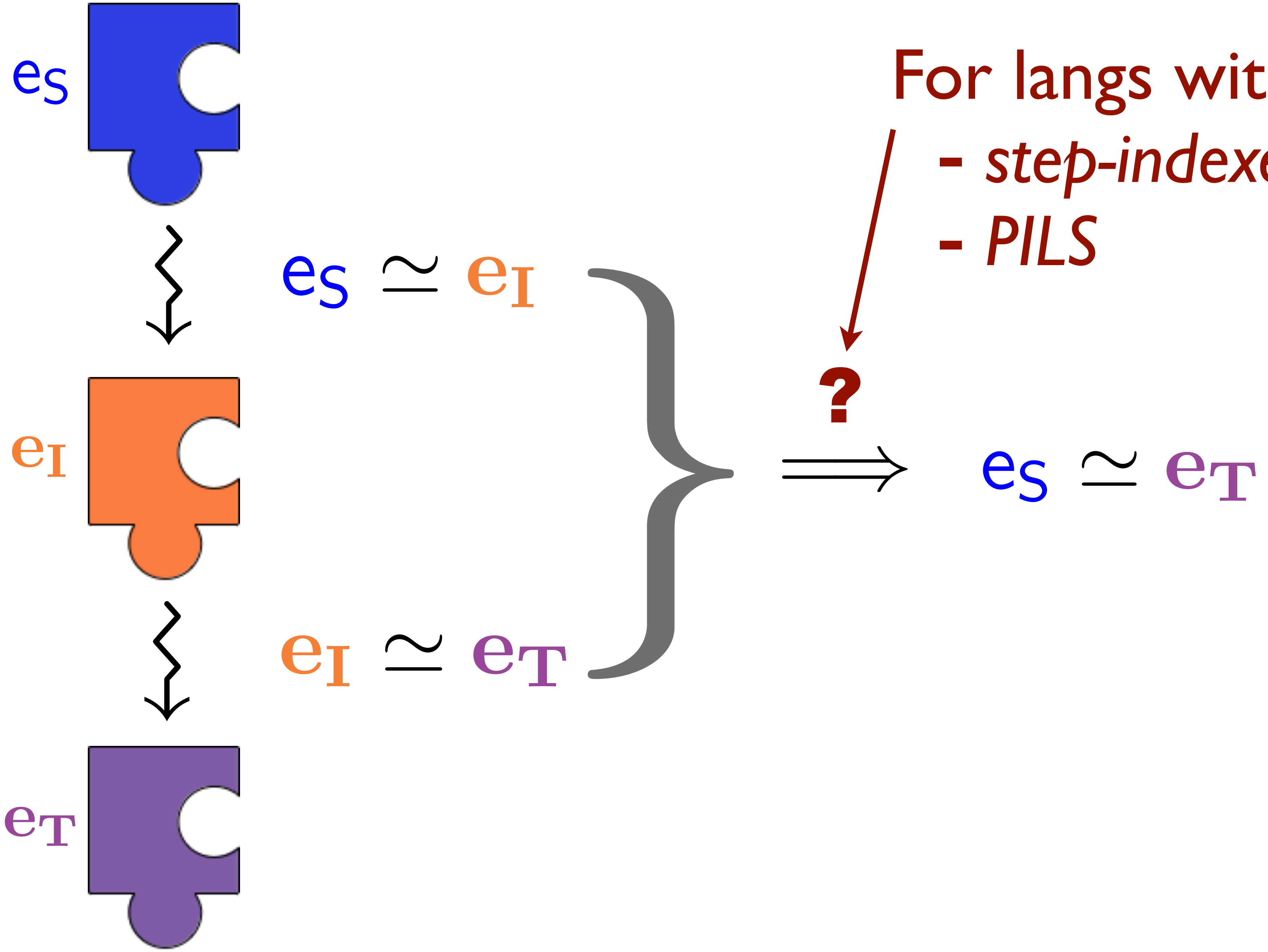
$$\left.\begin{array}{l} e_S \simeq e_I \\ \\ e_I \simeq e_T \end{array}\right\} \overset{?}{\Longrightarrow} \quad e_S \simeq e_T$$

cross-language relation;
no definition of ctx. equiv

Transitivity for single-lang. logical relation:

$$e \approx e' \iff e \approx^{ctx} e'$$

Cross-Language LR: Problem 2

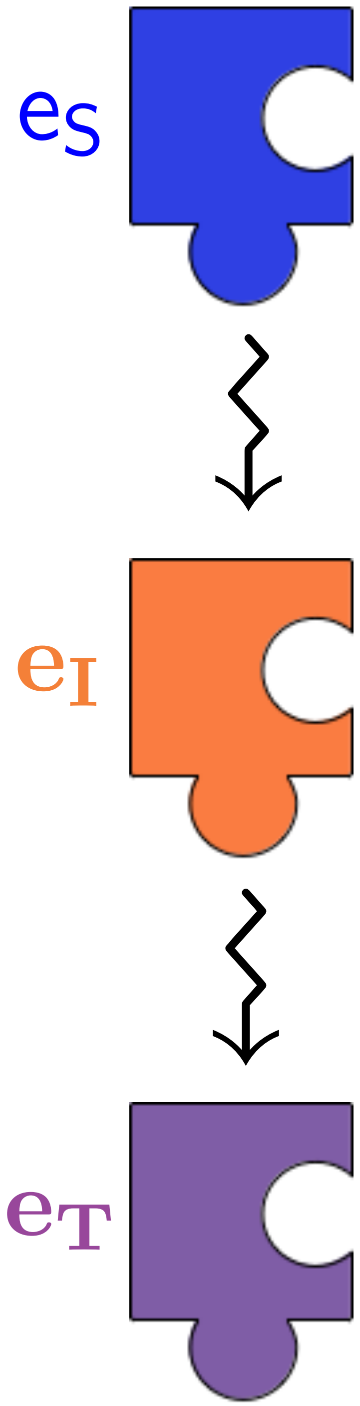


$$\left.\begin{array}{l} e_S \simeq e_I \\ \\ e_I \simeq e_T \end{array}\right\} \overset{?}{\Longrightarrow} \quad e_S \simeq e_T$$

For langs with refs:

- *step-indexed Kripke LR*
- *PILS*

Cross-Language LR: Problem 2

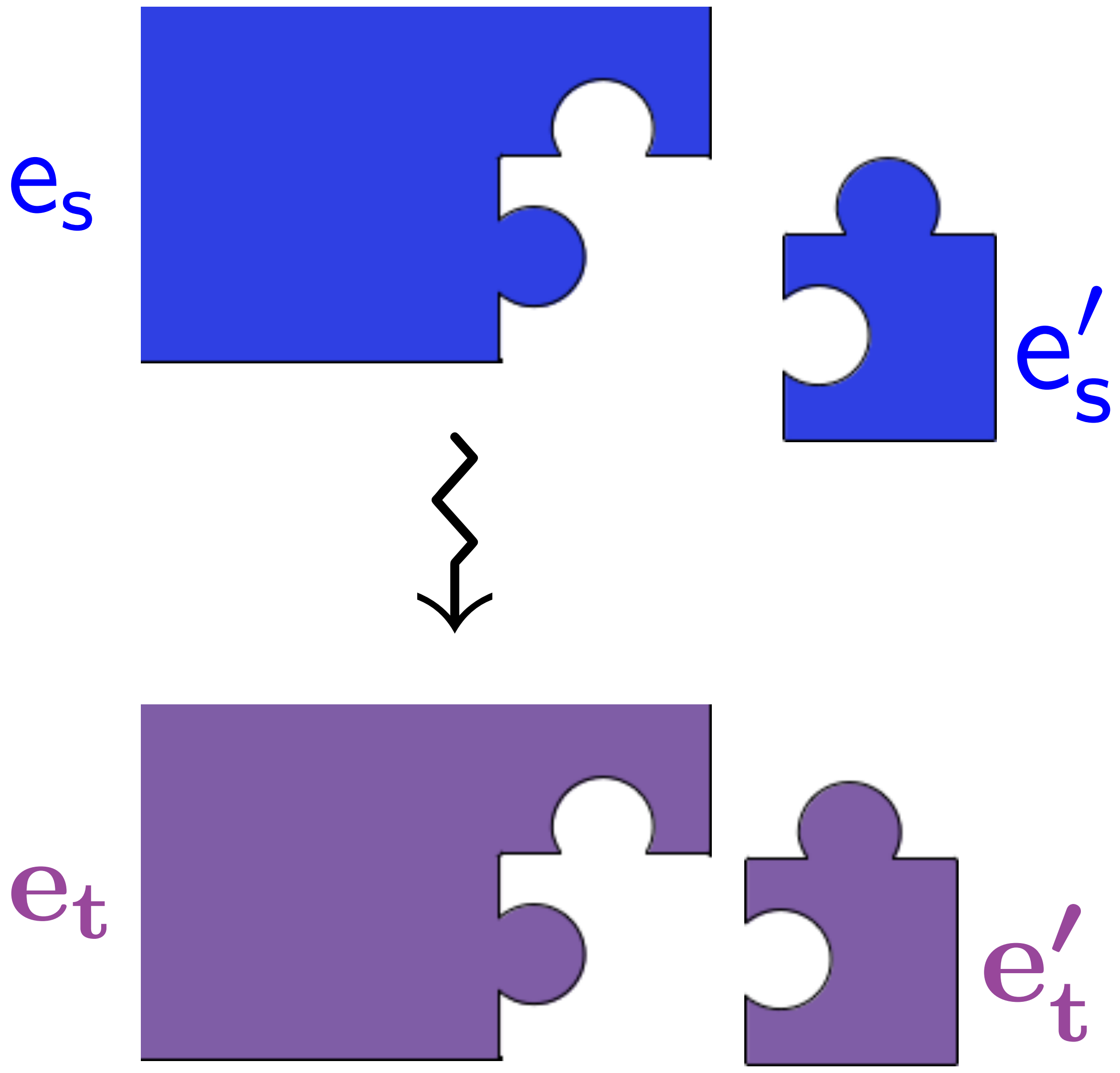


$$\left.\begin{array}{l} e_S \simeq e_I \\ \\ e_I \simeq e_T \end{array}\right\} \overset{?}{\Longrightarrow} \quad e_S \simeq e_T$$

For langs with refs:

- *step-indexed Kripke LR* ✗
- *PILS* ✓ *but lots of effort*

PILS: Problem 1 remains



Have $x : \tau' \vdash e_s \underset{\uparrow}{\simeq} e_t : \tau$

PILS

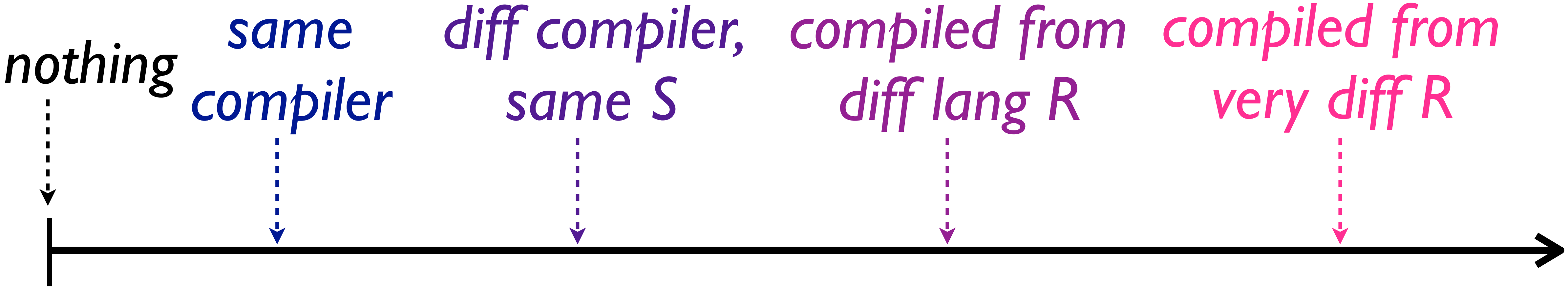
$\vdash e'_s \simeq e'_t : \tau'$

- Need to come up with e'_s -- not feasible in practice!
- Cannot link with e'_t whose behavior cannot be expressed in source.

Need a New Approach...

- that works for multi-pass compilers
- that allows linking with target code of arbitrary provenance

What we can link with



nothing

same compiler

diff compiler, same S

compiled from diff lang R

compiled from very diff R

CompCert

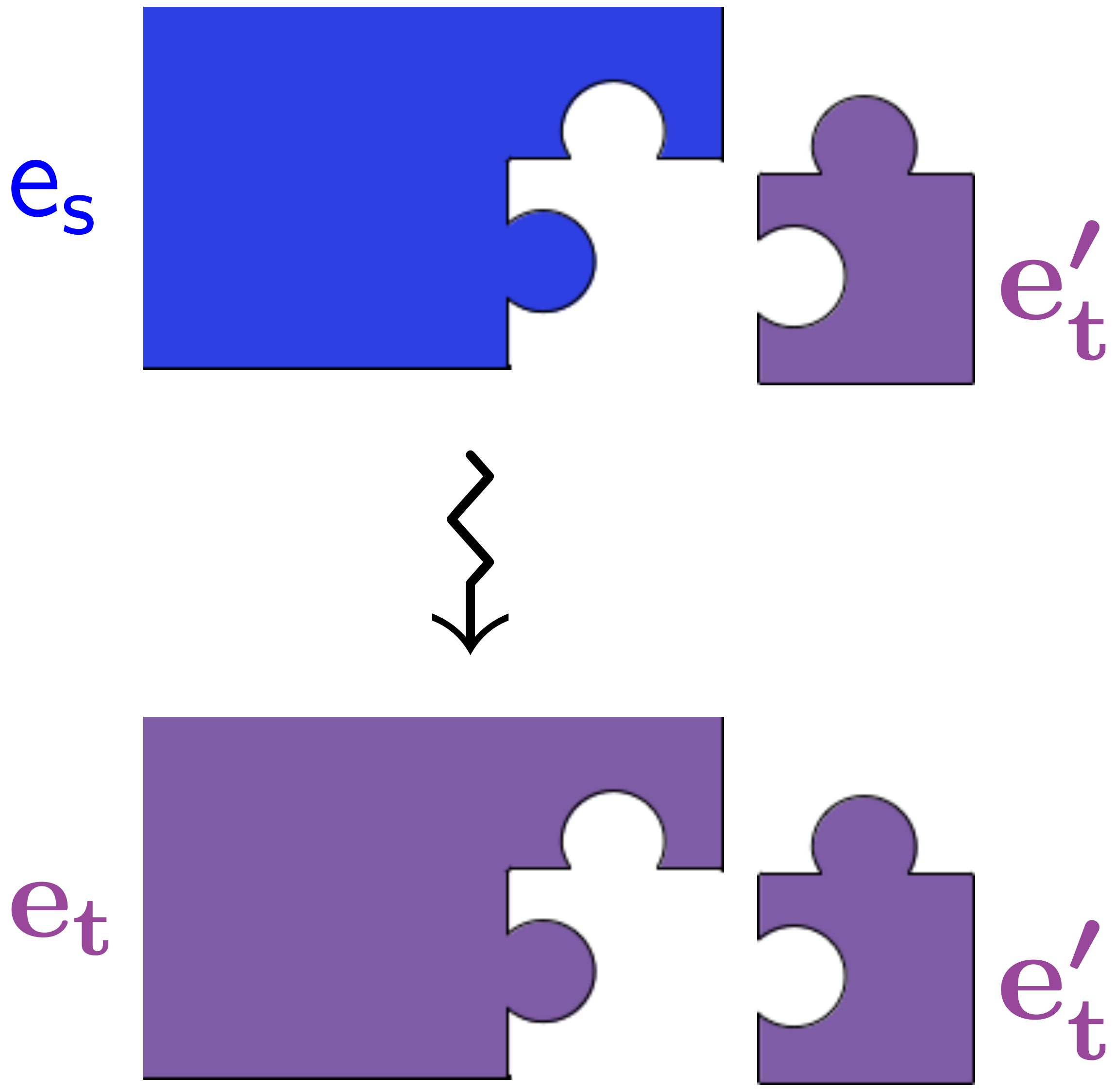
SepCompCert
Kang et al.'16

Pilsner
Neis et al.'15

Compositional CompCert
Stewart et al.'15

Multi-language ST
Perconti-Ahmed'14

Correct Compilation of Components?



$$e_S \approx e_T$$

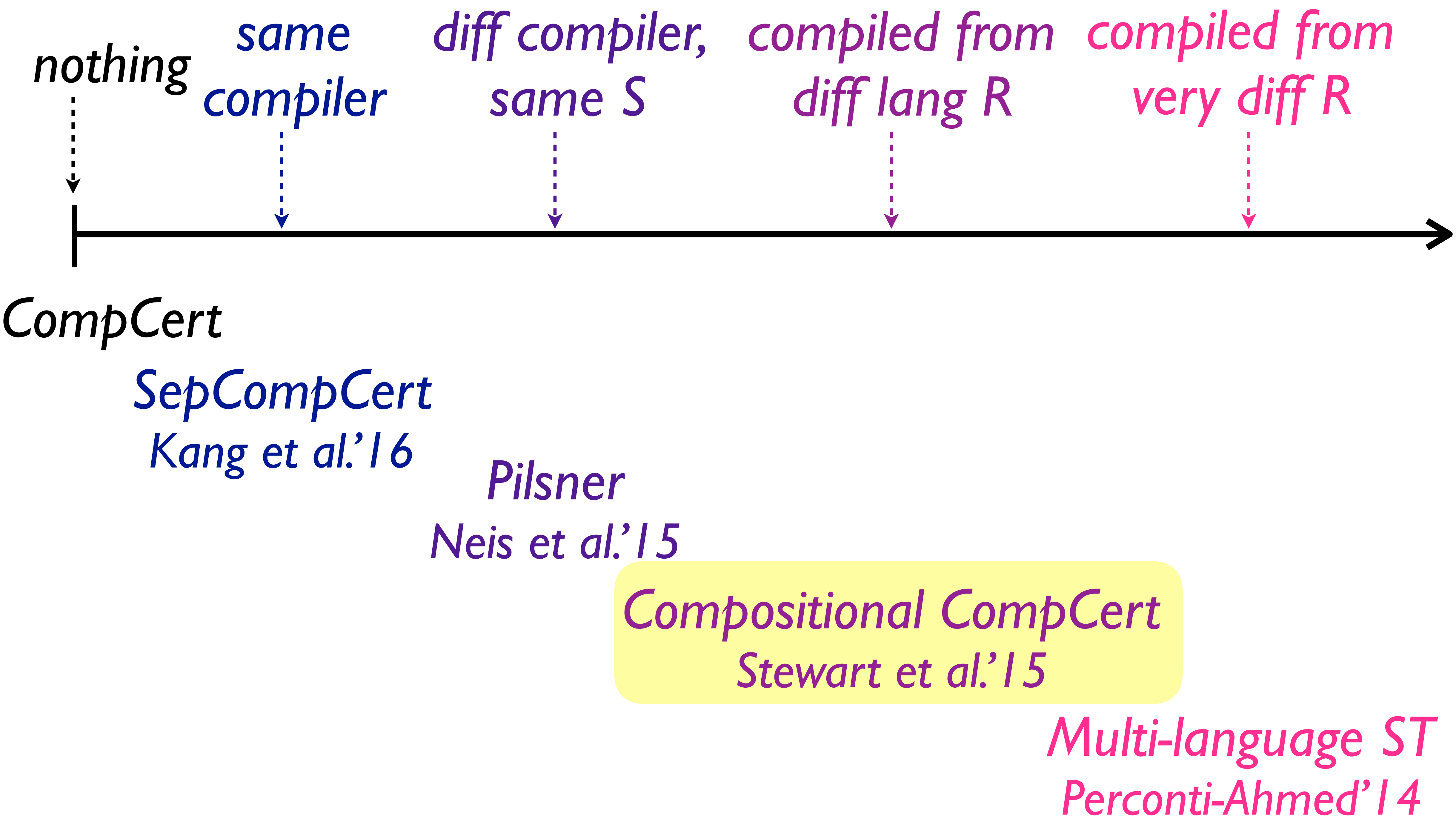
$\uparrow$

expressed how?

Need a semantics of source-target interoperability:

- *interaction semantics*
- *source-target multi-language*

What we can link with



nothing

same compiler

diff compiler, same S

compiled from diff lang R

compiled from very diff R

CompCert

SepCompCert
Kang et al.'16

Pilsner
Neis et al.'15

Compositional CompCert
Stewart et al.'15

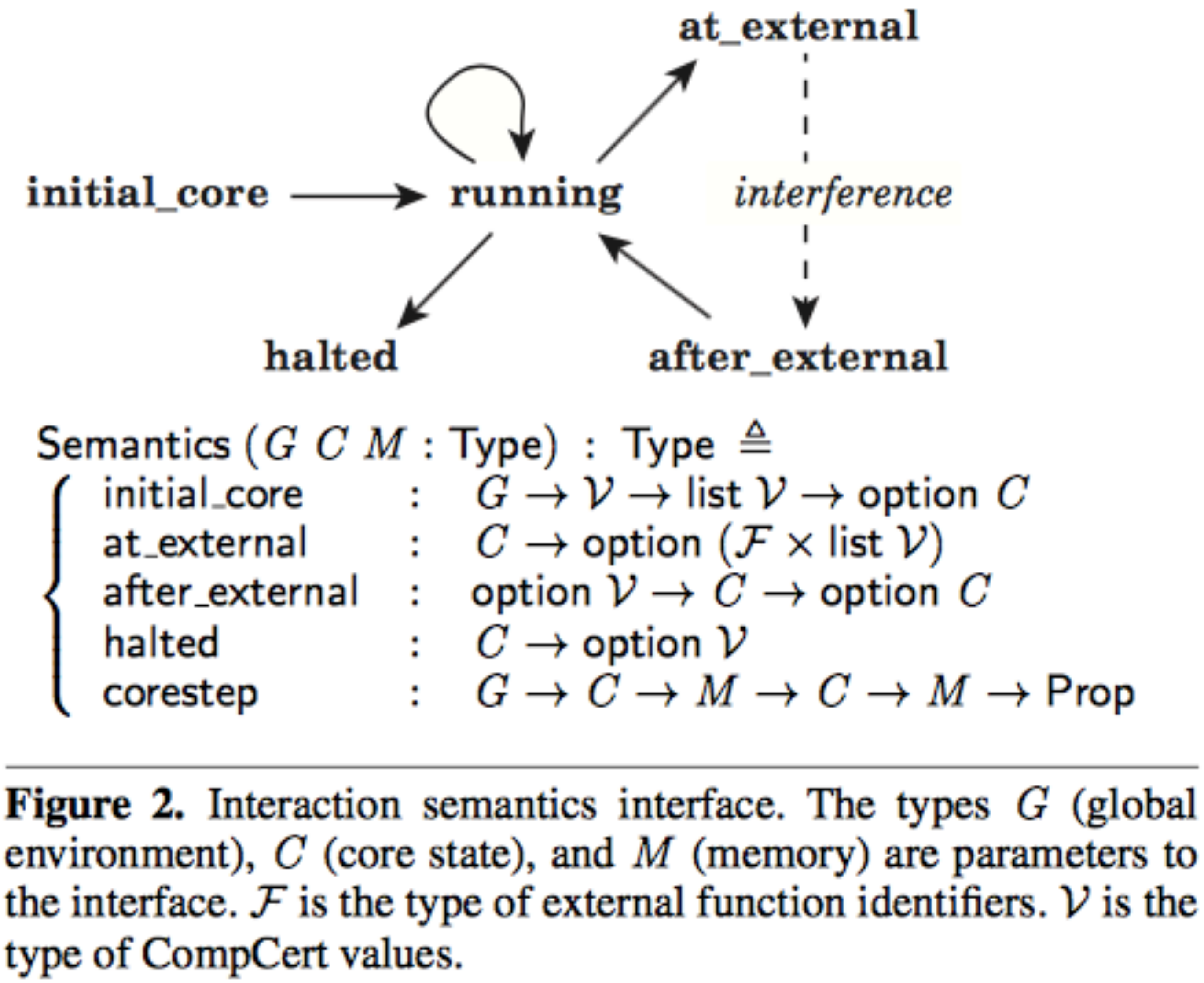
Multi-language ST
Perconti-Ahmed'14

Approach: Interaction Semantics

Compositional CompCert

[Stewart et al. '15]

- Language-independent linking



$$\mathsf{Semantics}\ (G\ C\ M : \mathsf{Type})\ :\ \mathsf{Type} \triangleq \left\{ \begin{array}{lcl} \mathsf{initial\_core} & : & G \to \mathcal{V} \to \mathsf{list}\ \mathcal{V} \to \mathsf{option}\ C \\ \mathsf{at\_external} & : & C \to \mathsf{option}\ (\mathcal{F} \times \mathsf{list}\ \mathcal{V}) \\ \mathsf{after\_external} & : & \mathsf{option}\ \mathcal{V} \to C \to \mathsf{option}\ C \\ \mathsf{halted} & : & C \to \mathsf{option}\ \mathcal{V} \\ \mathsf{corestep} & : & G \to C \to M \to C \to M \to \mathsf{Prop} \end{array} \right.$$

Figure 2. Interaction semantics interface. The types G (global environment), C (core state), and M (memory) are parameters to the interface. $\mathcal{F}$ is the type of external function identifiers. $\mathcal{V}$ is the type of CompCert values.

Approach: Interaction Semantics

Compositional CompCert
[Stewart et al. '15]

- Language-independent linking
- Structured simulation: support rely-guarantee relationship between the different languages while retaining vertical compositionality

Semantic representation of contexts code can link with.

Approach: Interaction Semantics

Compositional CompCert

[Stewart et al. '15]

- Language-independent linking

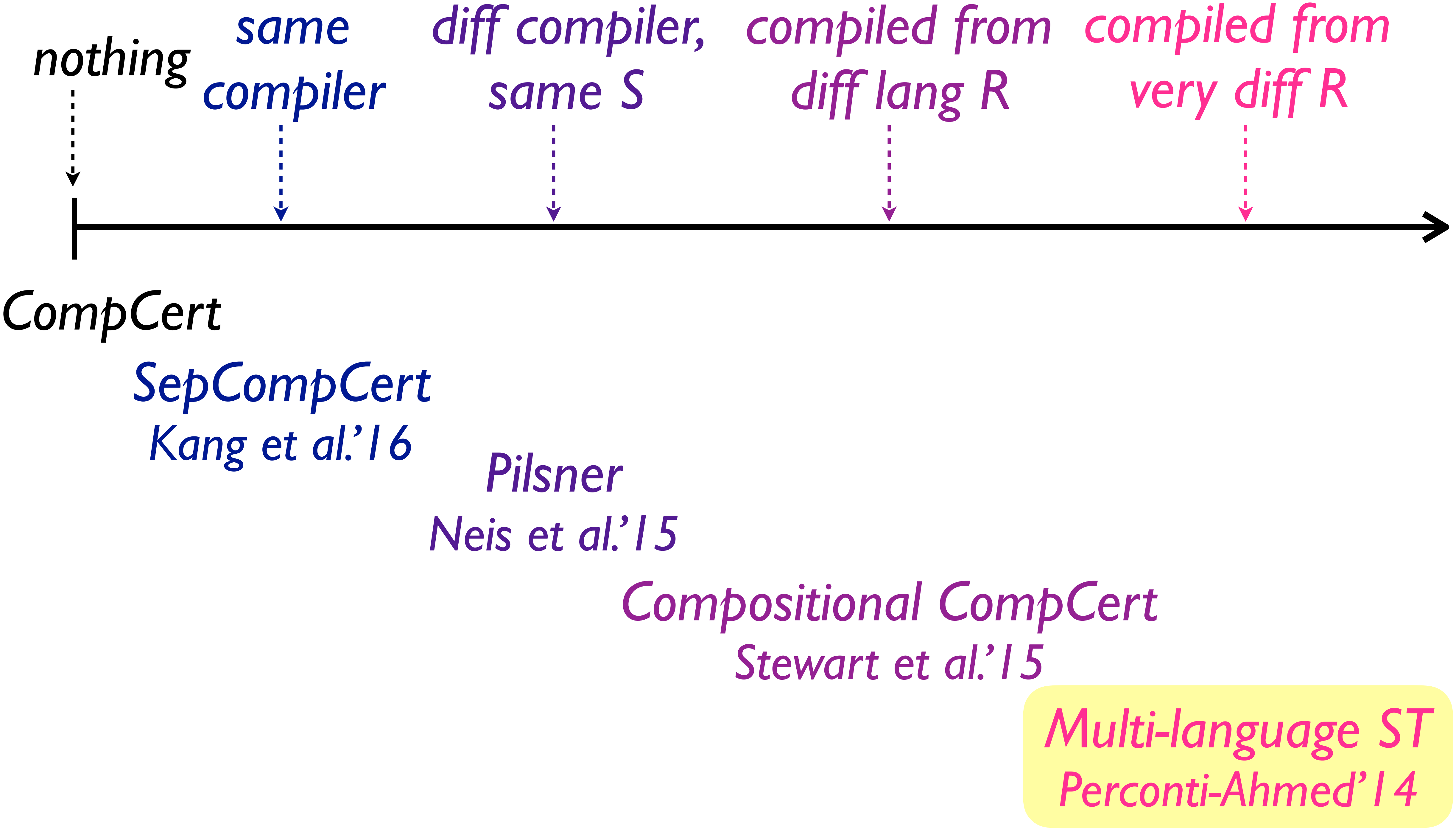
  - uniform CompCert memory model across all languages

  - not clear how to scale to richer source langs (e.g., ML), compilers with different source/target memory models

- Structured simulation: support rely-guarantee relationship between the different languages while retaining vertical compositionality

  - transitivity relies on compiler passes performing restricted set of memory transformations

What we can link with



nothing

same compiler

diff compiler, same S

compiled from diff lang R

compiled from very diff R

CompCert

SepCompCert
Kang et al.'16

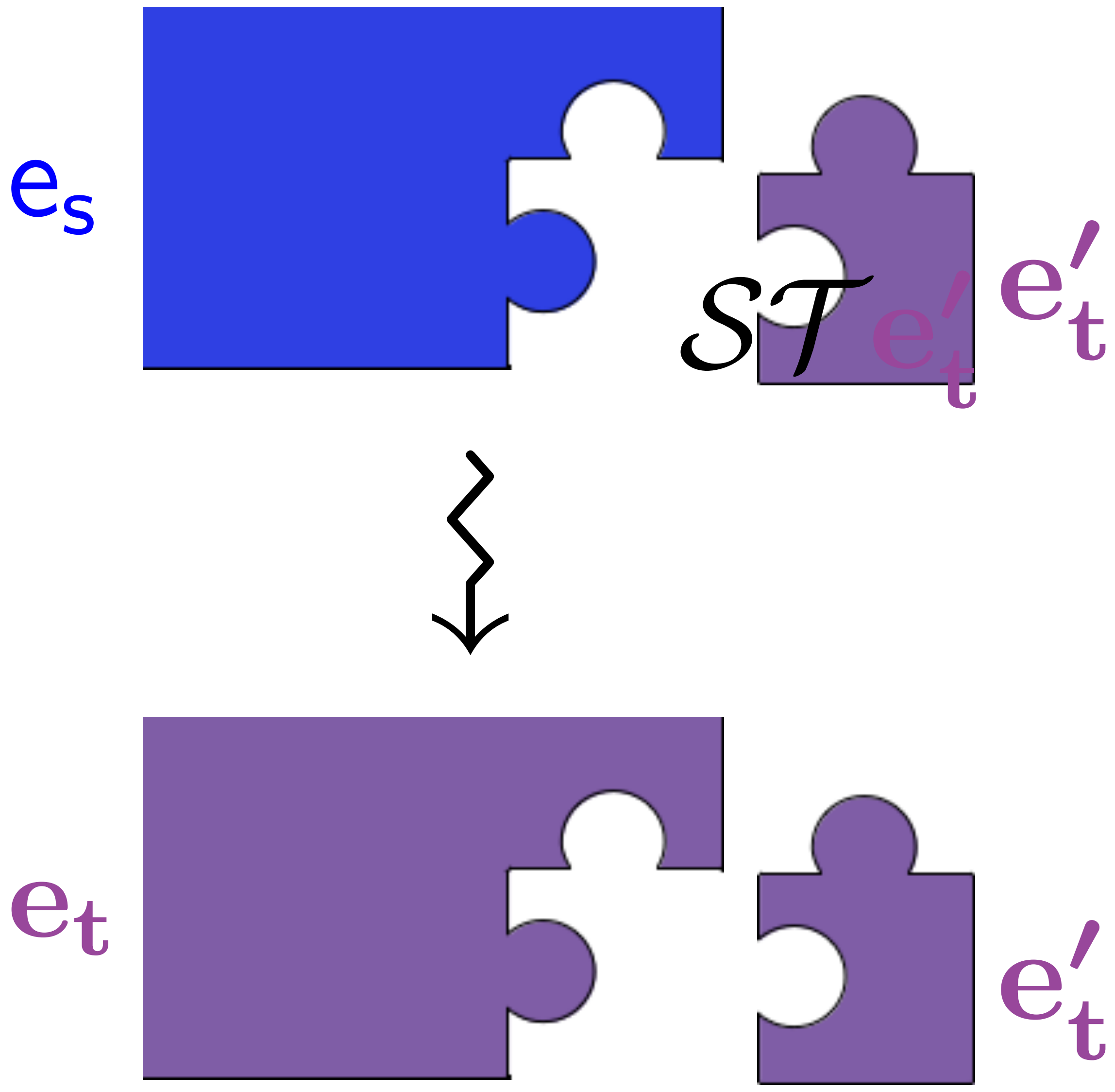
Pilsner
Neis et al.'15

Compositional CompCert
Stewart et al.'15

Multi-language ST
Perconti-Ahmed'14

Approach: Source-Target Multi-lang.

[Perconti-Ahmed'14]



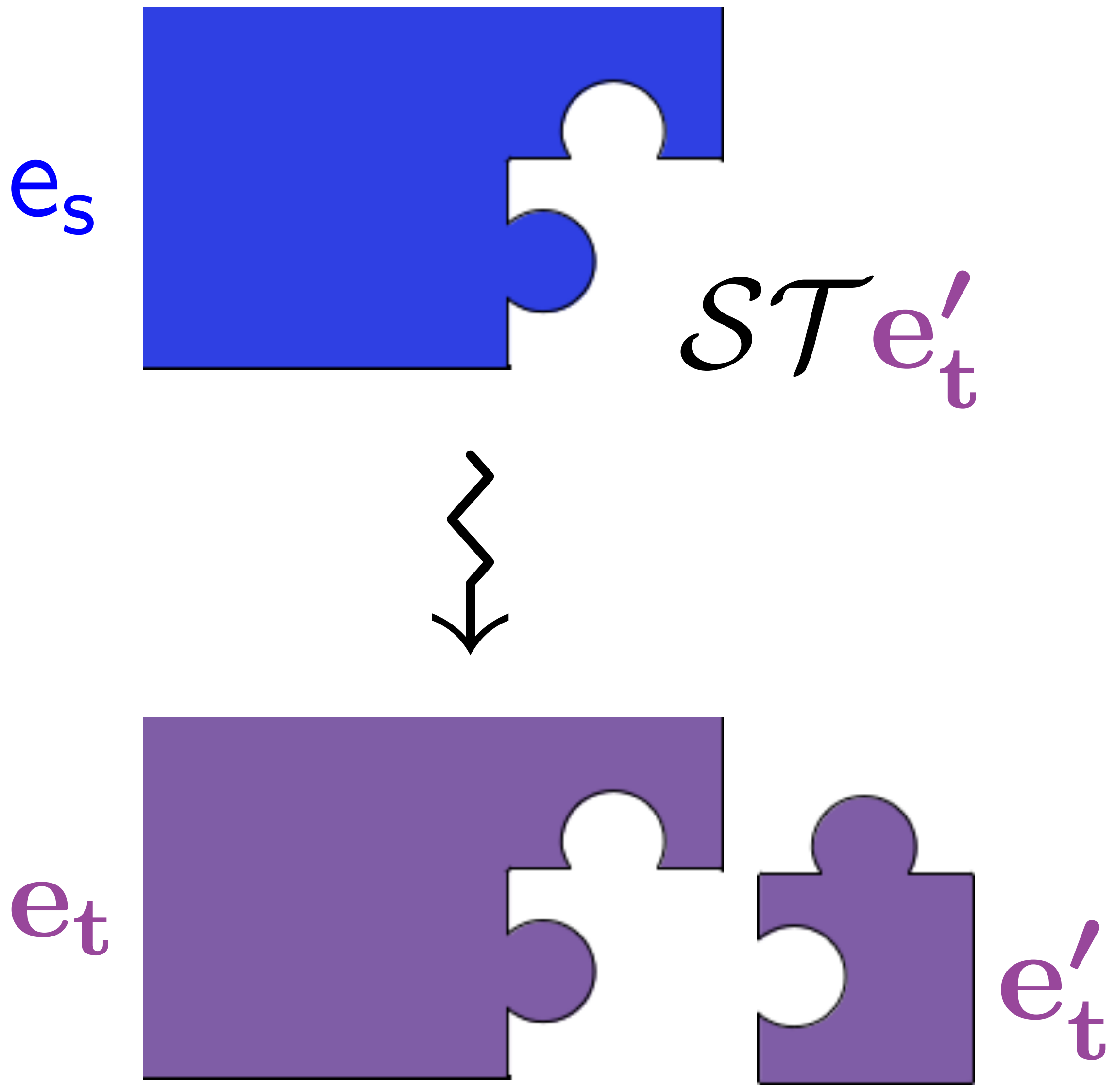
Specify semantics of source-target interoperability:

$\mathcal{ST}e_t$ $\mathcal{TS}e_s$

Multi-language semantics: a la Matthews-Findler '07

Approach: Source-Target Multi-lang.

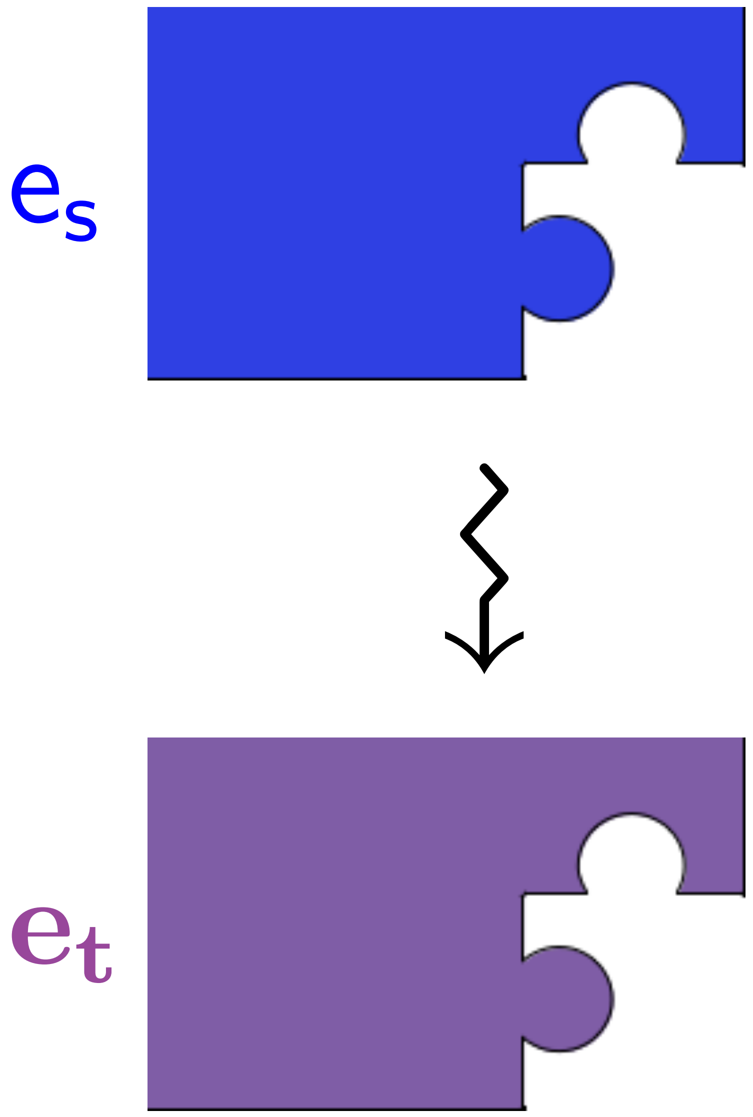
[Perconti-Ahmed'14]



$$\mathcal{T}\mathcal{S}(\mathrm{e_s}\ (\mathcal{S}\mathcal{T}\,\mathrm{e}'_{\mathrm{t}})) \approx^{ctx} \mathrm{e_t}\ \mathrm{e}'_{\mathrm{t}}$$

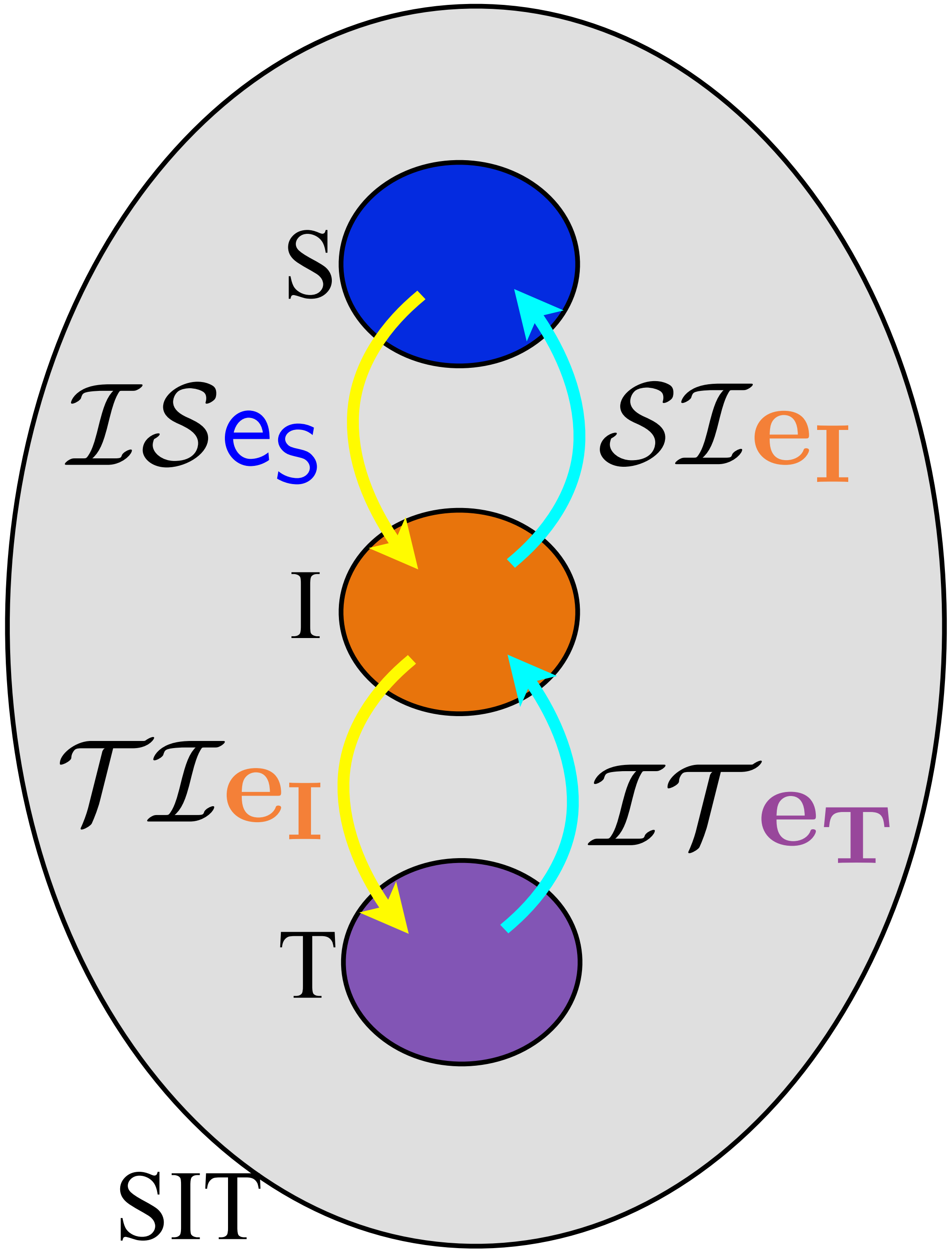
Approach: Source-Target Multi-lang.

[Perconti-Ahmed'14]

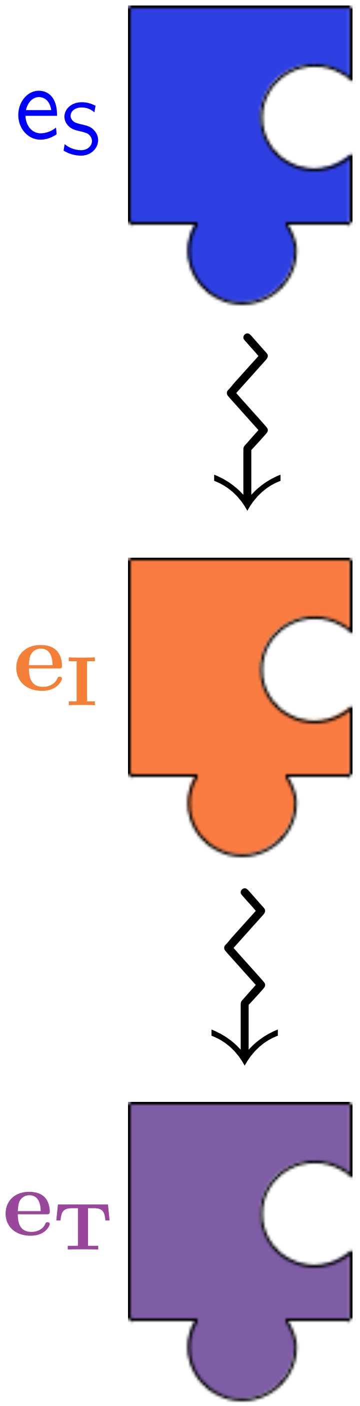


$$\begin{aligned} e_S \approx \mathbf{e}_T &\stackrel{\text{def}}{=} \\ &e_S \approx^{ctx} \mathcal{S}\mathcal{T}\,\mathbf{e}_T \end{aligned}$$

Multi-Language Semantics Approach



Compiler Correctness

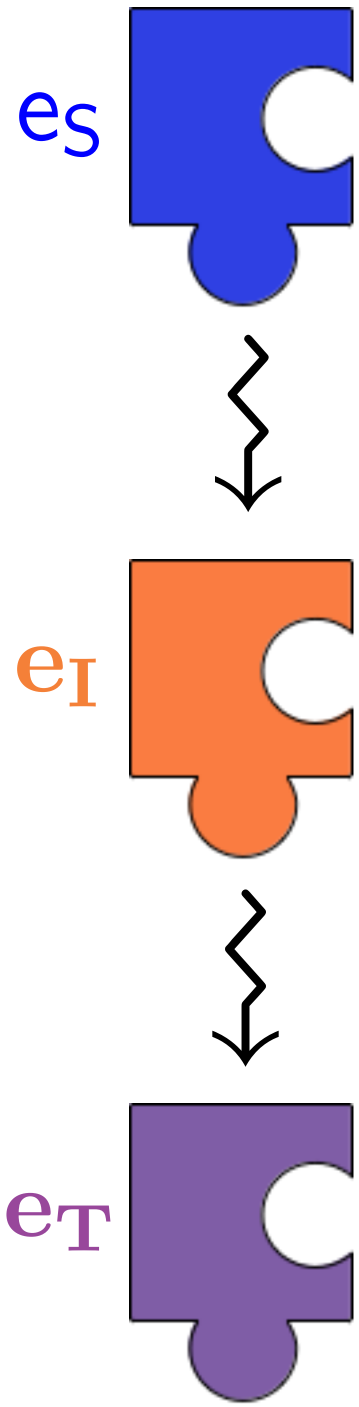


$e_S \approx^{ctx} \mathcal{SI}e_I$

$e_I \approx^{ctx} \mathcal{IT}e_T$

Multi-Lang. Approach: Multi-pass ✓

Compiler Correctness

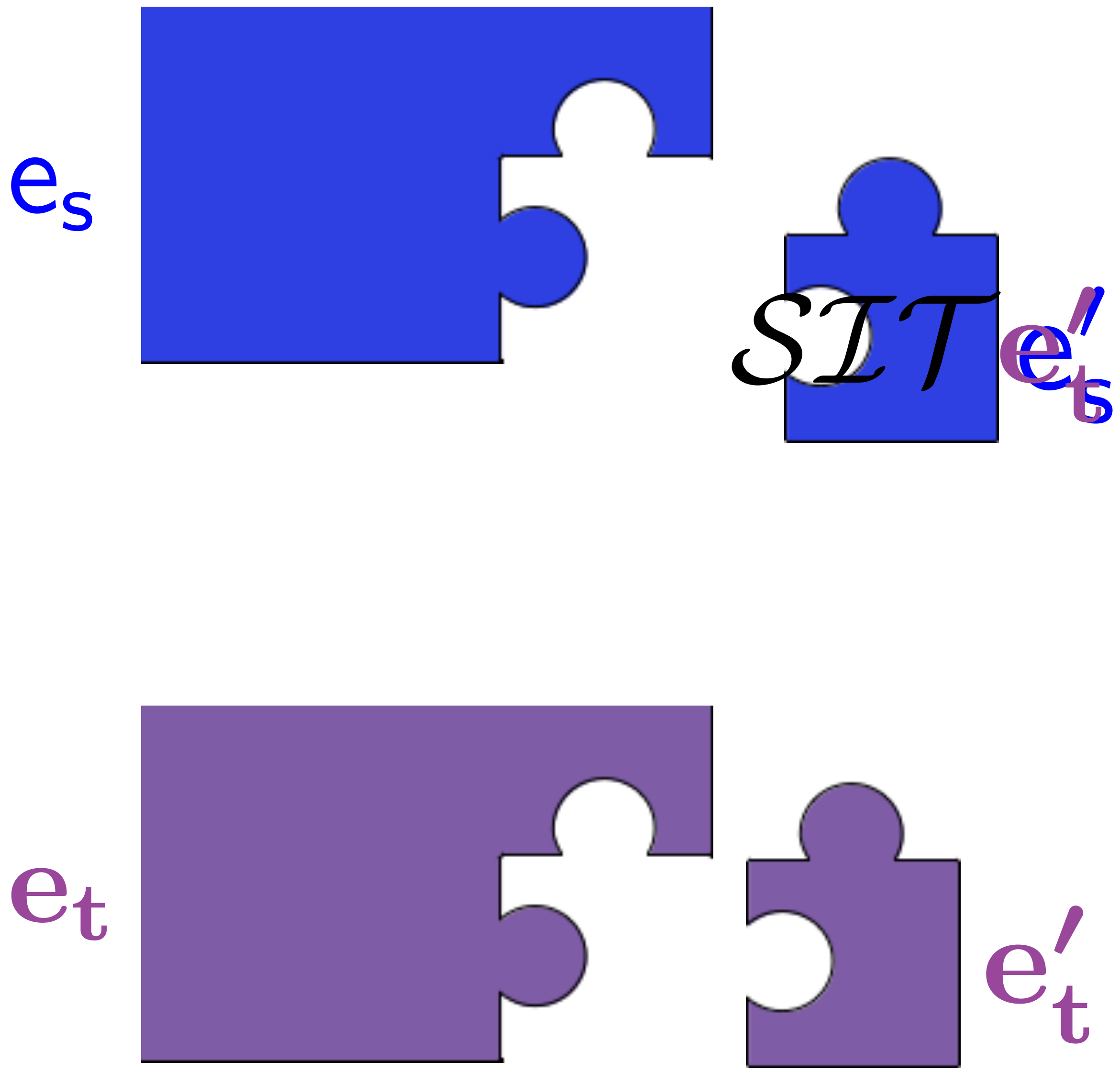


$e_S \approx^{ctx} \mathcal{SI}\,\mathbf{e_I}$

$\mathcal{SI}\,\mathbf{e_I} \approx^{ctx} \mathcal{SI}(\mathcal{IT}\,\mathbf{e_T})$

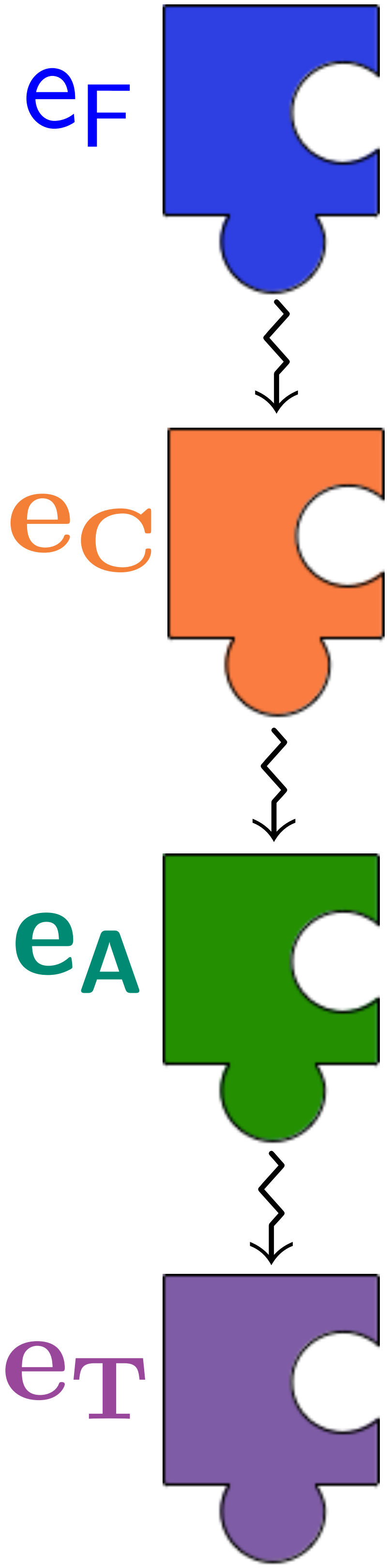
$e_S \approx^{ctx} \mathcal{SIT}\,\mathbf{e_T}$

Multi-Lang. Approach: Linking ✓



$$\mathcal{TIS}(e_s\,(\mathcal{SIT}\,e'_t)) \approx^{ctx} e_t\;e'_t$$

Compiler Correctness: F to TAL

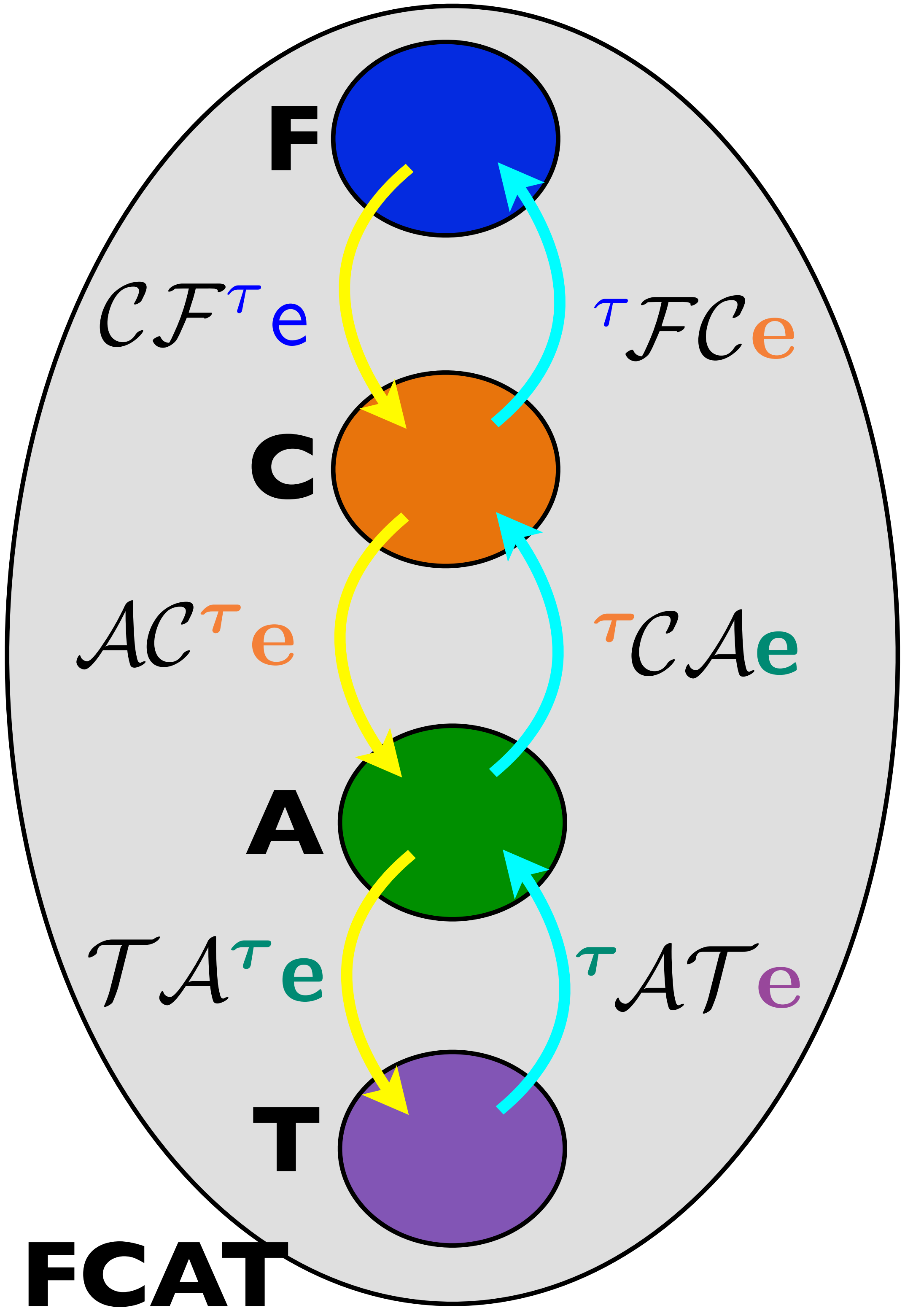


Closure Conversion $\tau^{\mathcal{C}}$

Allocation $\tau^{\mathcal{A}}$

Code Generation $\tau^{\mathcal{T}}$

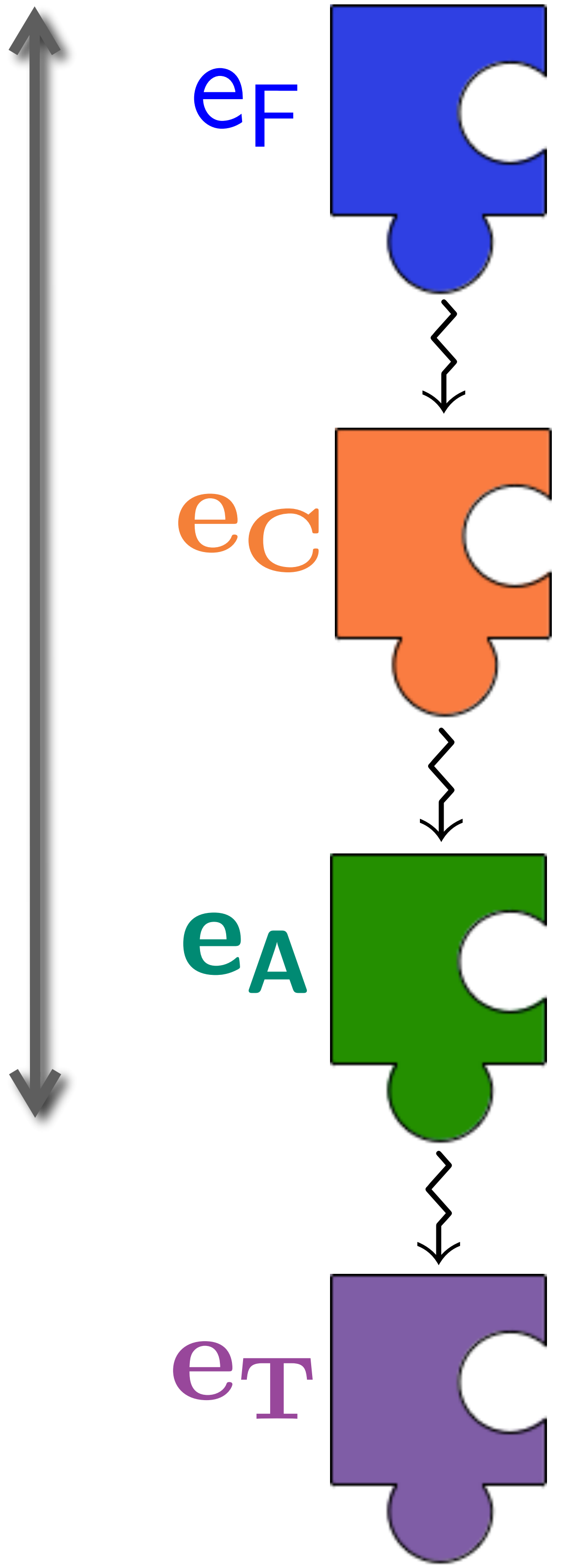
Combined language **FCAT**



- Boundaries mediate between

  τ & $\tau^{\mathcal{C}}$ τ & $\tau^{\mathcal{A}}$ τ & $\tau^{\mathcal{T}}$

Compiler Correctness: F to TAL



[Perconti-Ahmed ESOP'14]

e_F

Closure Conversion $\tau^{\mathcal{C}}$

e_C

Allocation $\tau^{\mathcal{A}}$

e_A

Code Generation $\tau^{\mathcal{T}}$

e_T

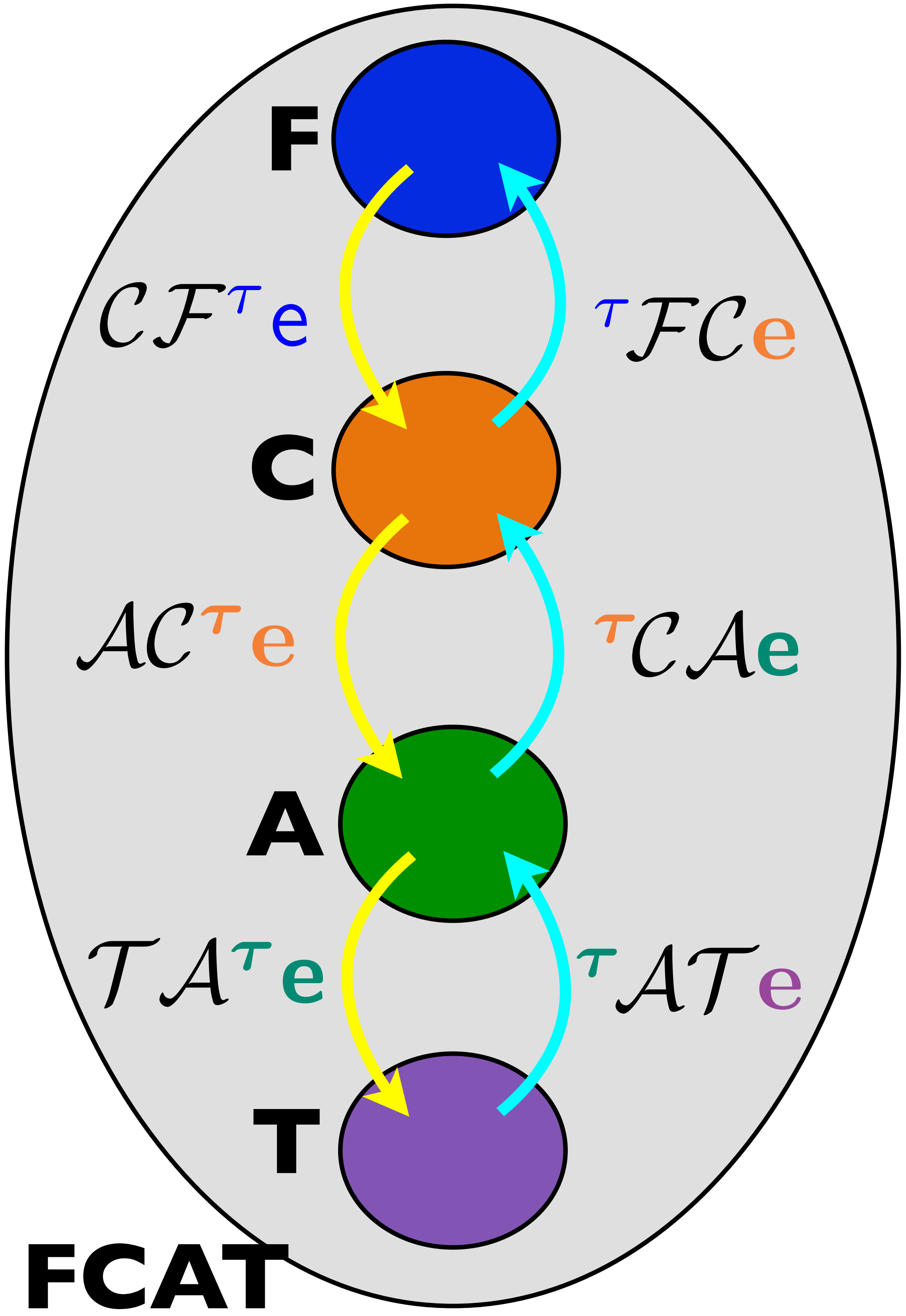
CompCompCert vs. Multi-language

Transitivity:	
- structured simulations	- all passes use multi-lang $\approx^{ctx}$
Check okay-to-link-with:	
- satisfies CompCert memory model	- satisfies expected type (translation of source type)
Contexts:	
- semantic representation	- syntactic representation
Requires uniform memory model across compiler IRs?	
- yes	- no

Case Study: Closure Conversion

Correctness of typed closure conversion using multi-language semantics... [on board]

Challenges

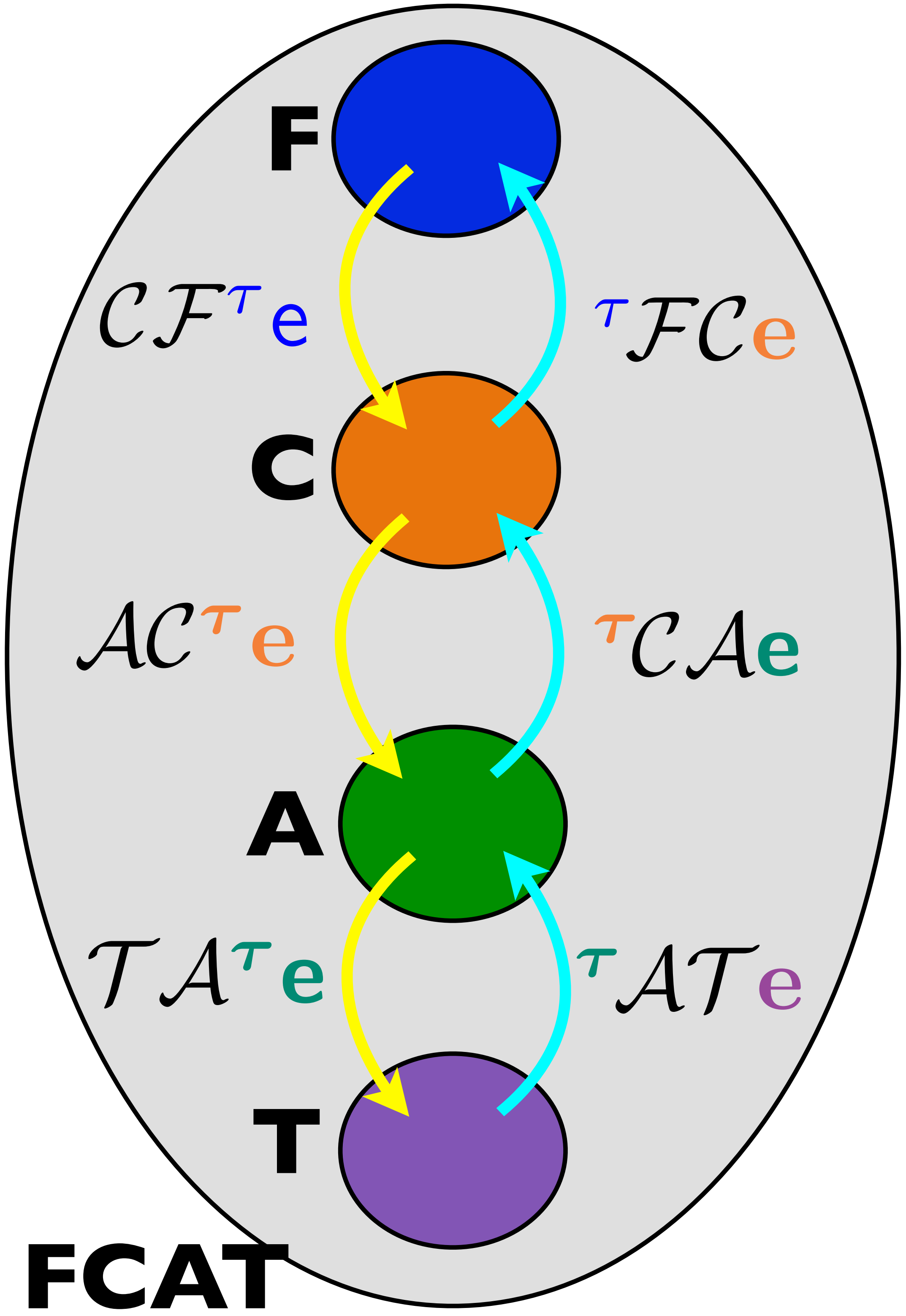


F+C: Interoperability semantics with type abstraction in both languages

C+A: Interoperability when compiler pass allocates code & tuples on heap

A+T: What is e? What is v? How to define contextual equiv. for TAL *components*? How to define logical relation?

Challenges



F+C: Interoperability semantics with type abstraction in both languages

C+A: Interoperability when compiler pass allocates code & tuples on heap

A+T: What is e ? What is $\mathbf{v}$? How to define contextual equiv. for TAL *components*? How to define logical relation?

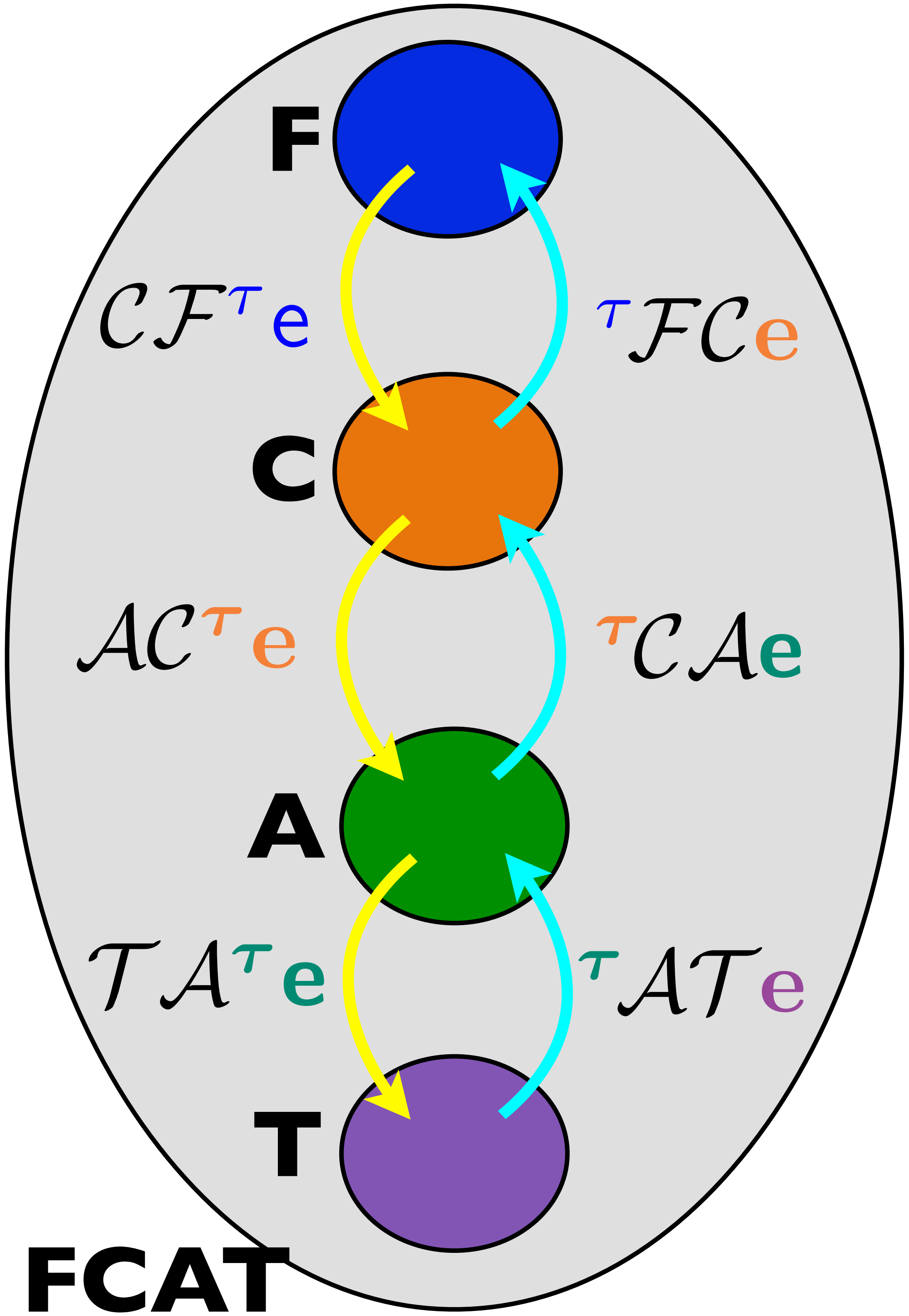
Interoperability: **C** and **A**

$$\mathbf{H}, \ell \mapsto \langle \mathbf{v} \rangle ; {}^{\langle \tau \rangle}\mathcal{C}\mathcal{A}\, \ell \longmapsto \mathbf{H}, \ell \mapsto \langle \mathbf{v} \rangle ; \langle \mathbf{v} \rangle$$

$$\mathbf{H} ; \mathcal{A}\mathcal{C}^{\langle \tau \rangle} \langle \mathbf{v} \rangle \longmapsto \mathbf{H}, \ell \mapsto \langle \mathbf{v} \rangle ; \ell$$

Allocate a new
location for tuple

Challenges



F+C: Interoperability semantics with type abstraction in both languages

C+A: Interoperability when compiler pass allocates code & tuples on heap

A+T: What is $\mathbf{e}$? What is $\mathbf{v}$? How to define contextual equiv. for TAL *components*? How to define logical relation?

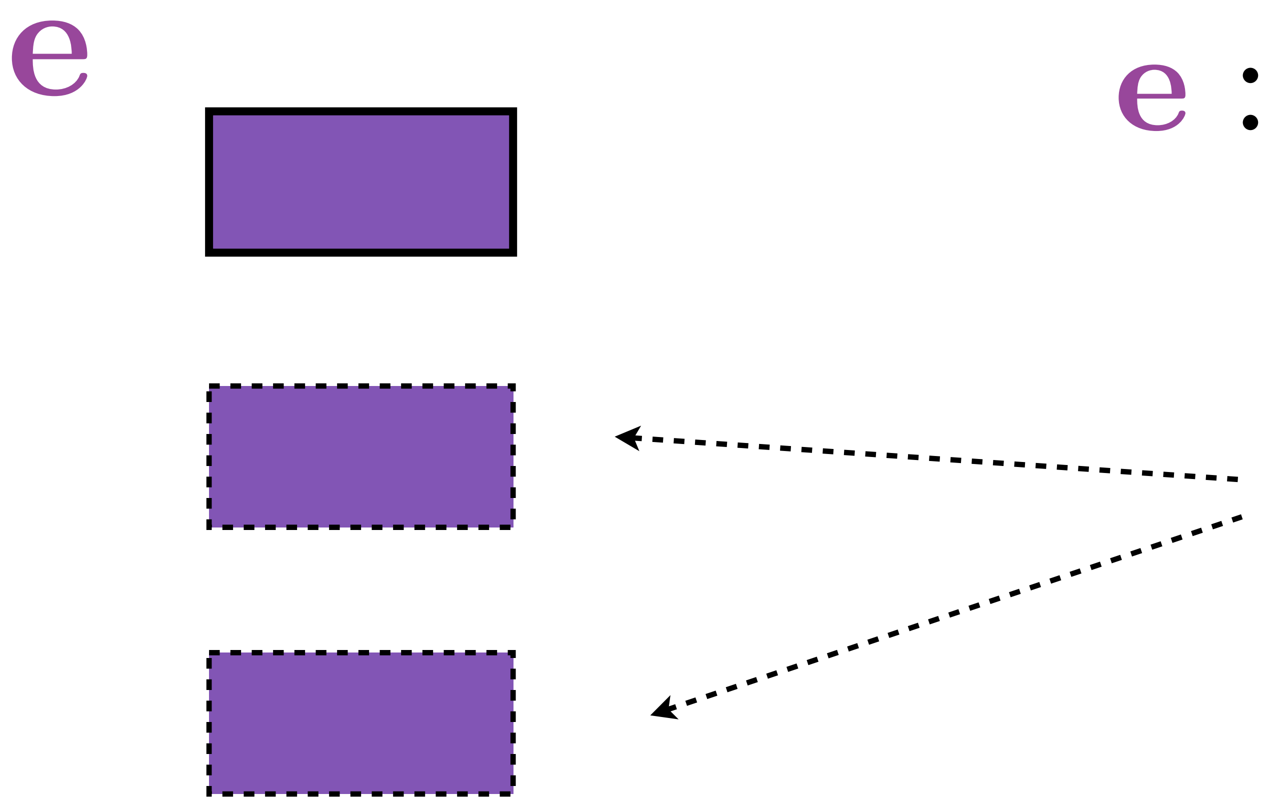
Central Challenge: interoperability between high-level (direct-style) language & assembly (continuation style)

FunTAL: Reasonably Mixing a Functional Language with Assembly *[Patterson et al. '17]*

What is a component in TAL?

$$\mathbf{e} : \tau \rightsquigarrow \mathbf{e}$$

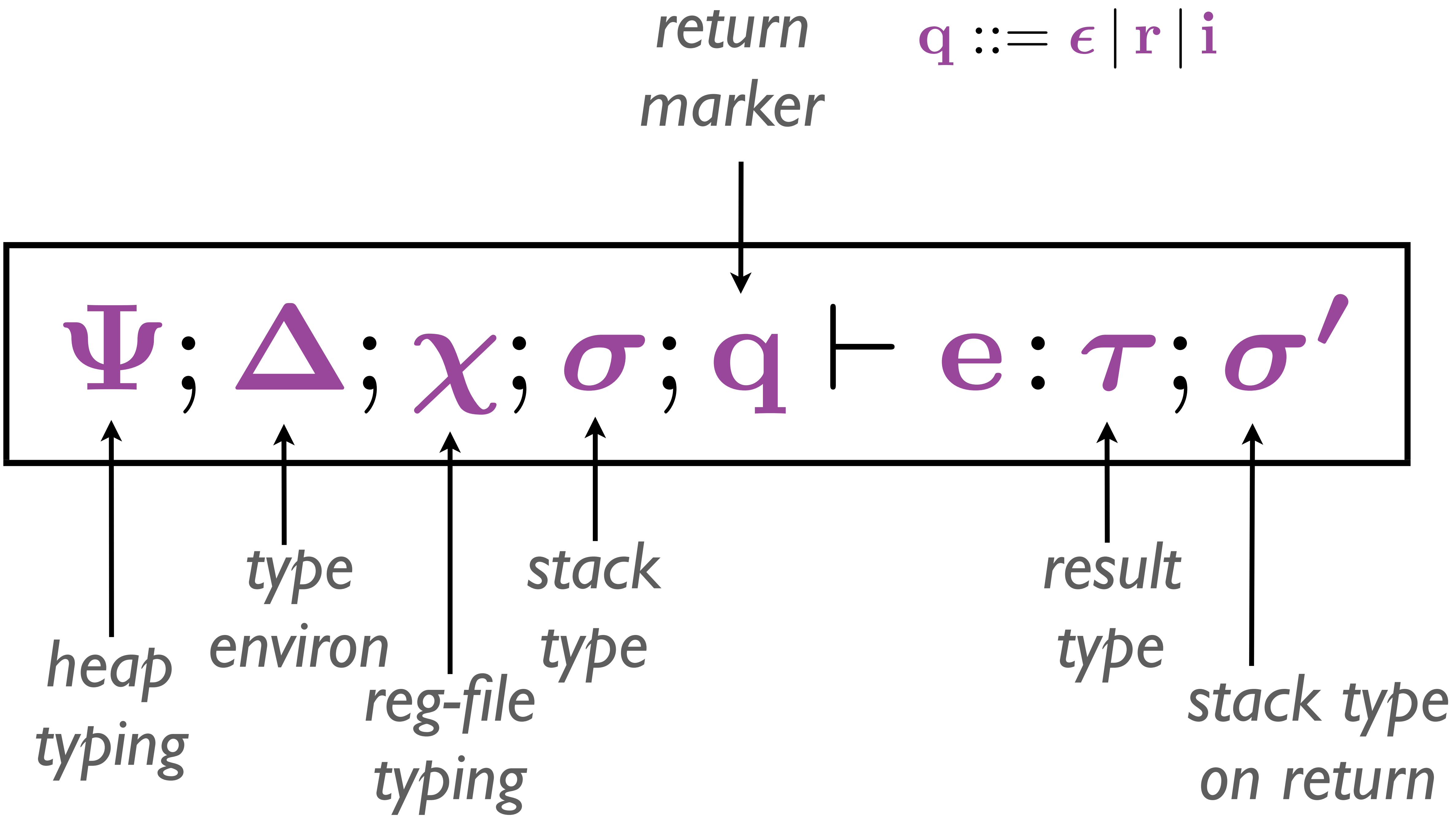
Instruction Sequence



$$\mathbf{e} ::= (\mathbf{I}, \mathbf{H})$$

Heap with basic blocks

Typing TAL Components



$\mathbf{q} ::= \epsilon \mid \mathbf{r} \mid \mathbf{i}$

return marker

$$\Psi; \Delta; \chi; \sigma; \mathbf{q} \vdash \mathbf{e} : \tau; \sigma'$$

heap typing

type environ

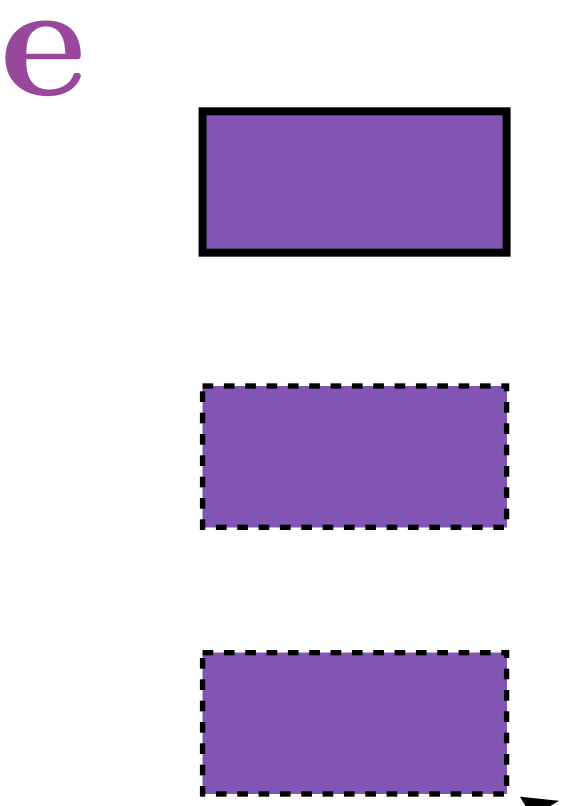
reg-file typing

stack type

result type

stack type on return

Basic blocks



$$\mathbf{e} ::= (\mathbf{I}, \mathbf{H})$$

$$\mathbf{code}[\Delta]\{\chi;\sigma\}^{\mathrm{q}}.\mathbf{I} \;:\; \forall[\Delta].\{\chi;\sigma\}^{\mathrm{q}}$$

Equivalence of **T** Components: Tricky!

Logical relations: related inputs to related outputs

$$\mathcal{V}[\![\tau_1 \rightarrow \tau_2]\!] = \{(W, \lambda \mathsf{x}.\mathsf{e}_1, \lambda \mathsf{x}.\mathsf{e}_1) \mid \ldots\}$$

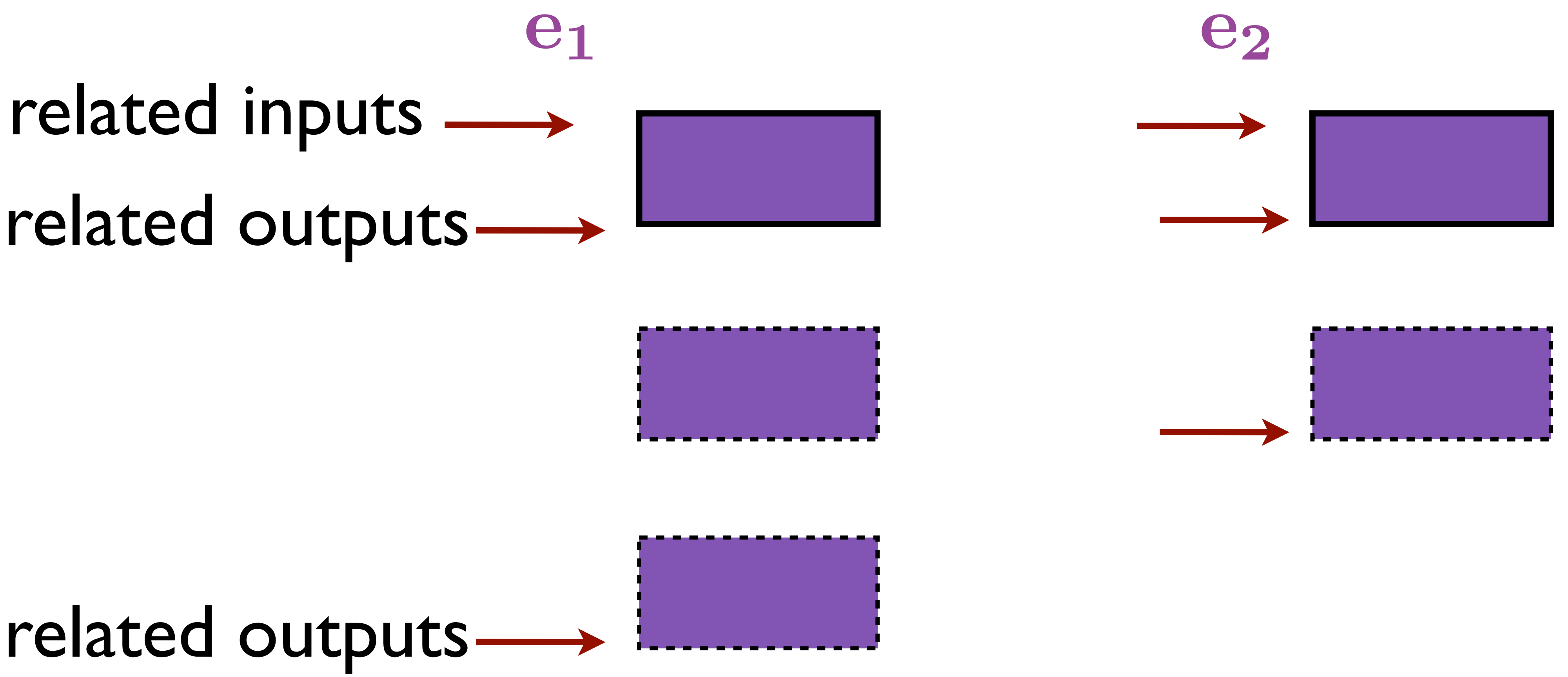
$$\mathcal{HV}[\![\forall[\boldsymbol{\Delta}].\{\boldsymbol{\chi}; \boldsymbol{\sigma}\}^{\mathrm{q}}]\!] = \{(W, \mathbf{code}[\boldsymbol{\Delta}]\{\boldsymbol{\chi}; \boldsymbol{\sigma}\}^{\mathrm{q}}.\mathbf{I}_1, \mathbf{code}[\boldsymbol{\Delta}]\{\boldsymbol{\chi}; \boldsymbol{\sigma}\}^{\mathrm{q}}.\mathbf{I}_2) \mid \ldots\}$$

Equivalence of **T** Components

Logical relations: related inputs to related outputs

$$\mathcal{V}[\![\tau_1 \rightarrow \tau_2]\!] = \{(W, \lambda \mathsf{x}.\mathsf{e}_1, \lambda \mathsf{x}.\mathsf{e}_1) \mid \ldots\}$$

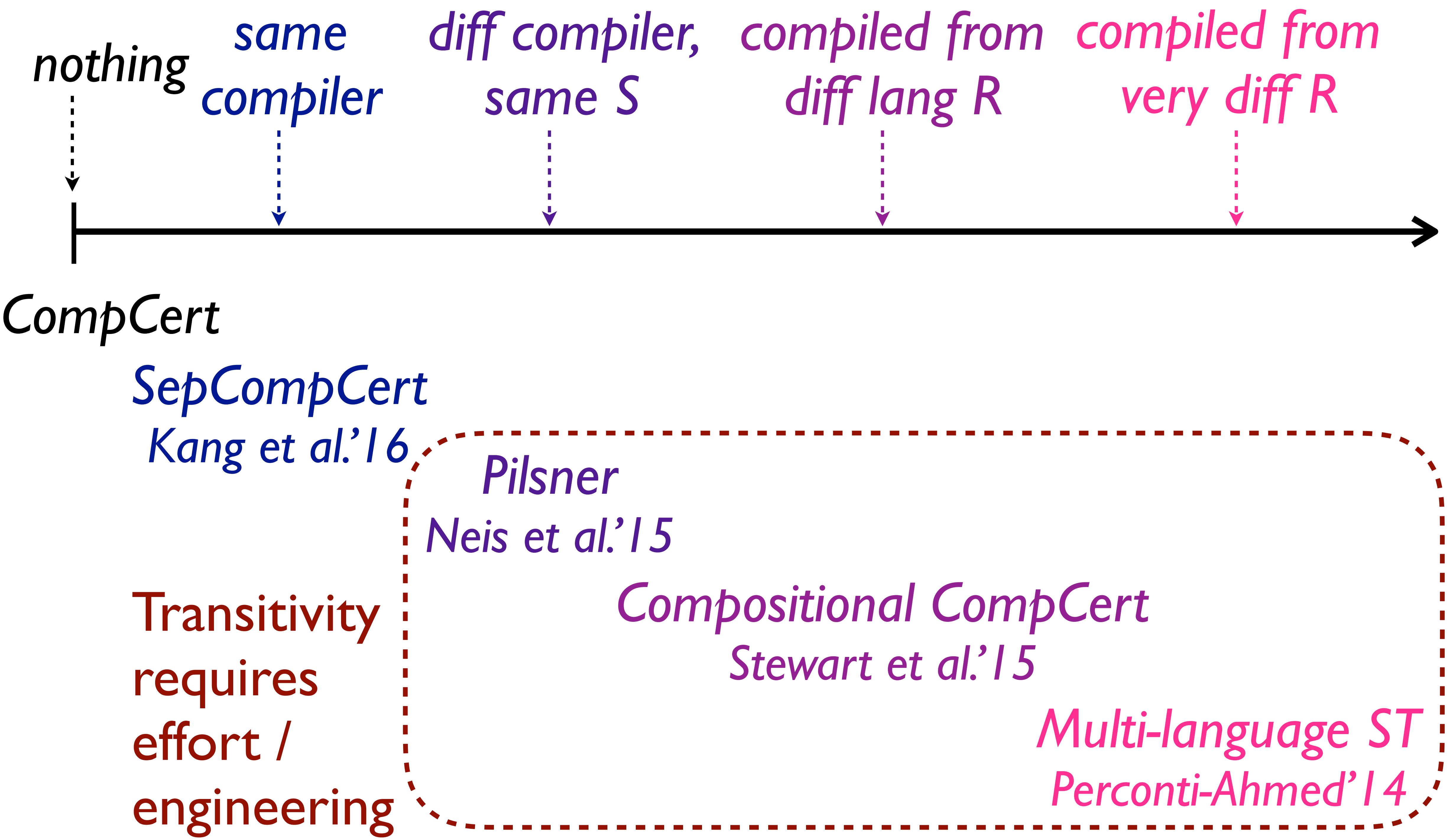
$$\mathcal{HV}[\![\forall[\boldsymbol{\Delta}].\boldsymbol{\{\chi;\sigma\}^{\mathrm{q}}}]\!] = \{(W, \mathbf{code}[\boldsymbol{\Delta}]\boldsymbol{\{\chi;\sigma\}^{\mathrm{q}}}.\mathbf{I}_1, \mathbf{code}[\boldsymbol{\Delta}]\boldsymbol{\{\chi;\sigma\}^{\mathrm{q}}}.\mathbf{I}_2) \mid \ldots\}$$



Ongoing: Multi-lang. Approach

- Underway: Code Generation pass to TAL
- Working on simplifying multi-language design to support easier proofs when multiple embedded languages have polymorphism & refs
  - *Matthew Kolosick, Dustin Jamner, Max New, AA*

Horizontal / Vertical Compositionality



nothing

same compiler

diff compiler, same S

compiled from diff lang R

compiled from very diff R

CompCert

SepCompCert
Kang et al.'16

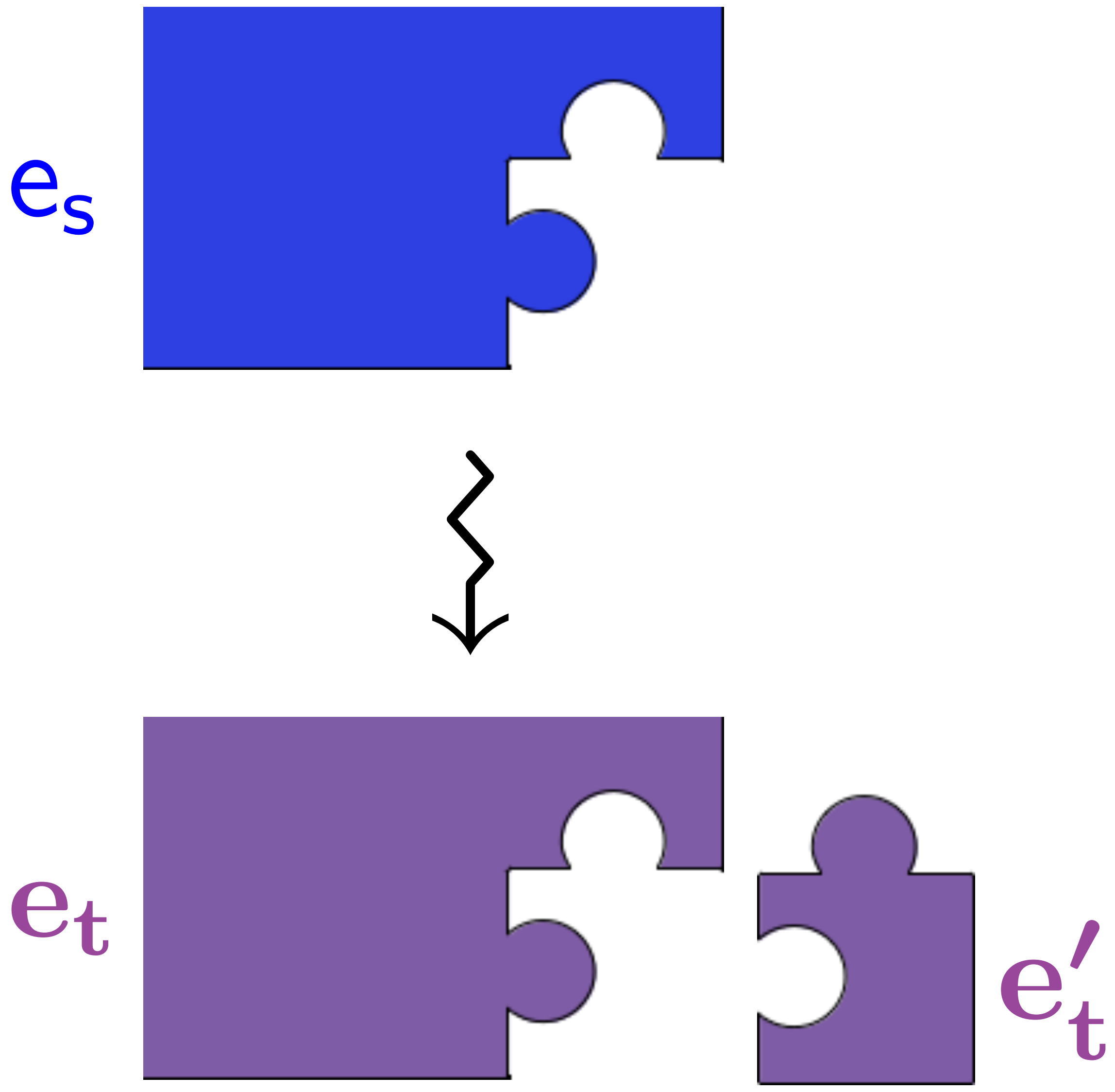
Pilsner
Neis et al.'15

Compositional CompCert
Stewart et al.'15

Multi-language ST
Perconti-Ahmed'14

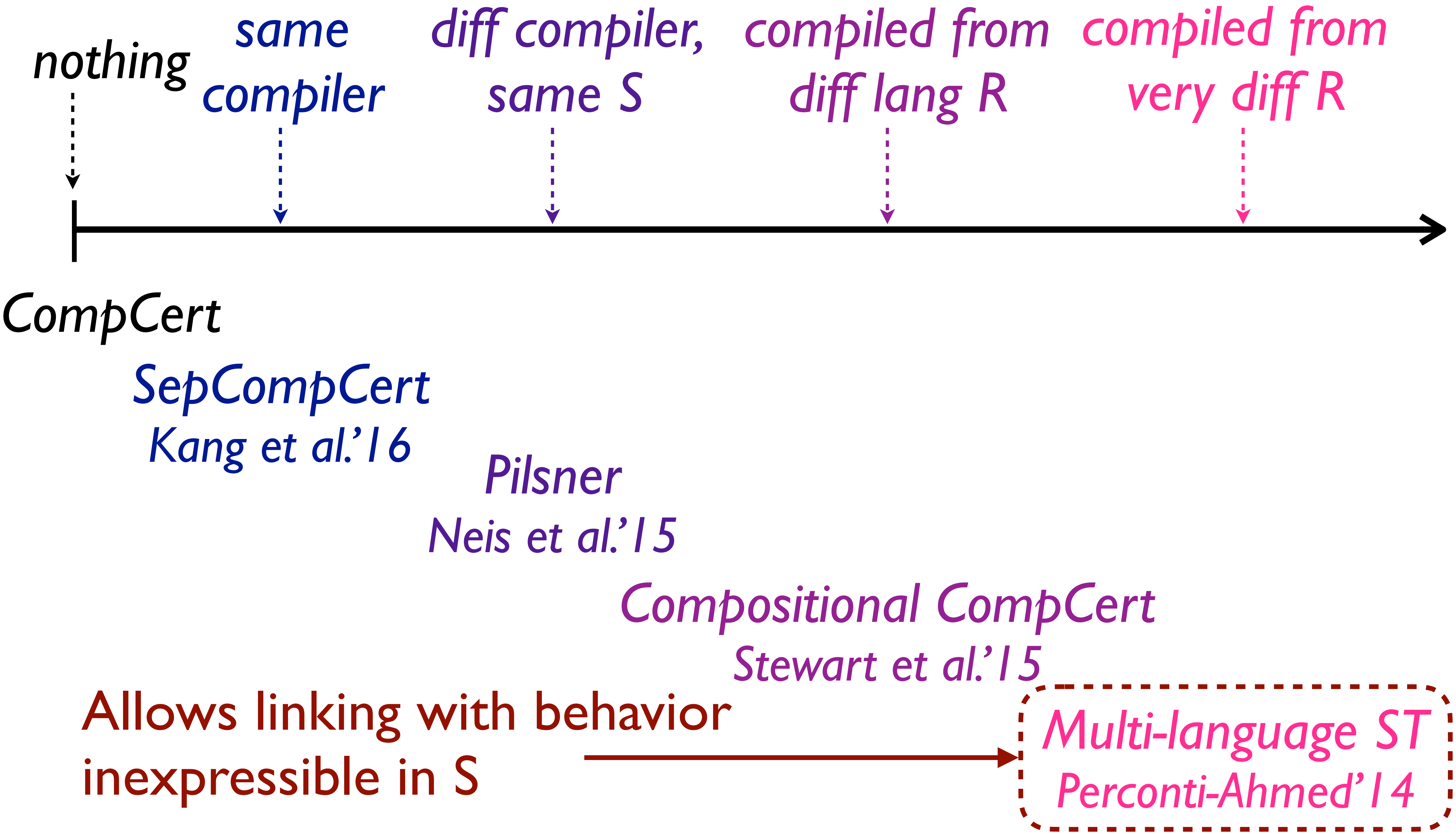
Transitivity requires effort / engineering

Horizontal Compositionality



Behavior inexpressible in S ?

Horizontal / Vertical Compositionality



nothing

same compiler

diff compiler, same S

compiled from diff lang R

compiled from very diff R

CompCert

SepCompCert
Kang et al.'16

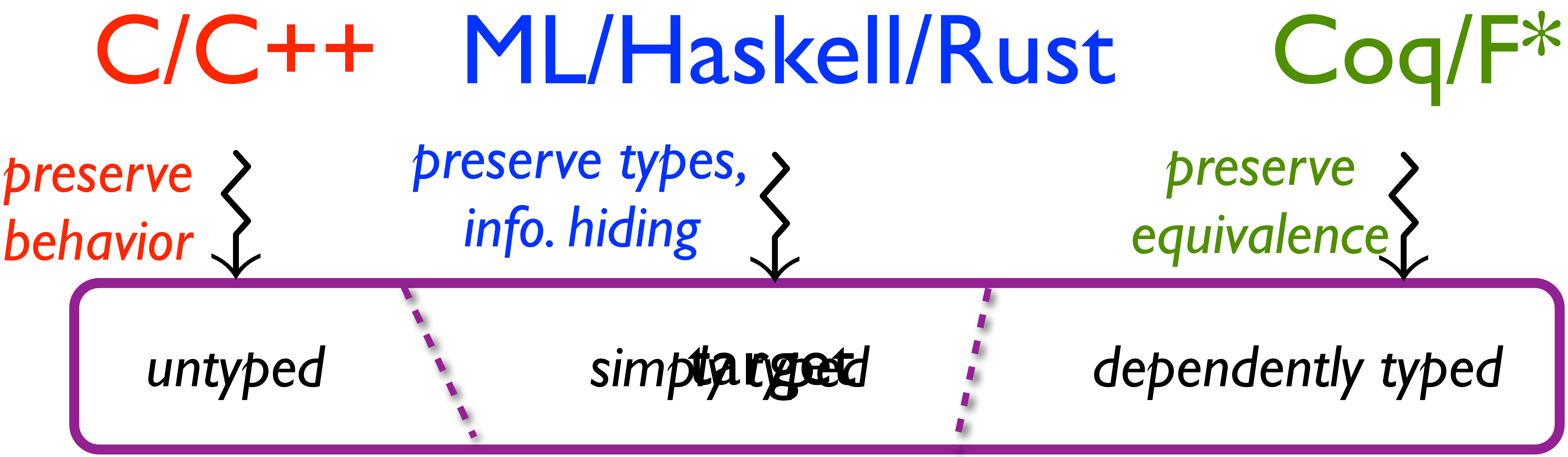
Pilsner
Neis et al.'15

Compositional CompCert
Stewart et al.'15

Allows linking with behavior inexpressible in S

Multi-language ST
Perconti-Ahmed'14

Verified Compilers for Multi-lang. World



C/C++ ML/Haskell/Rust Coq/F*

preserve behavior

preserve types, info. hiding

preserve equivalence

untyped

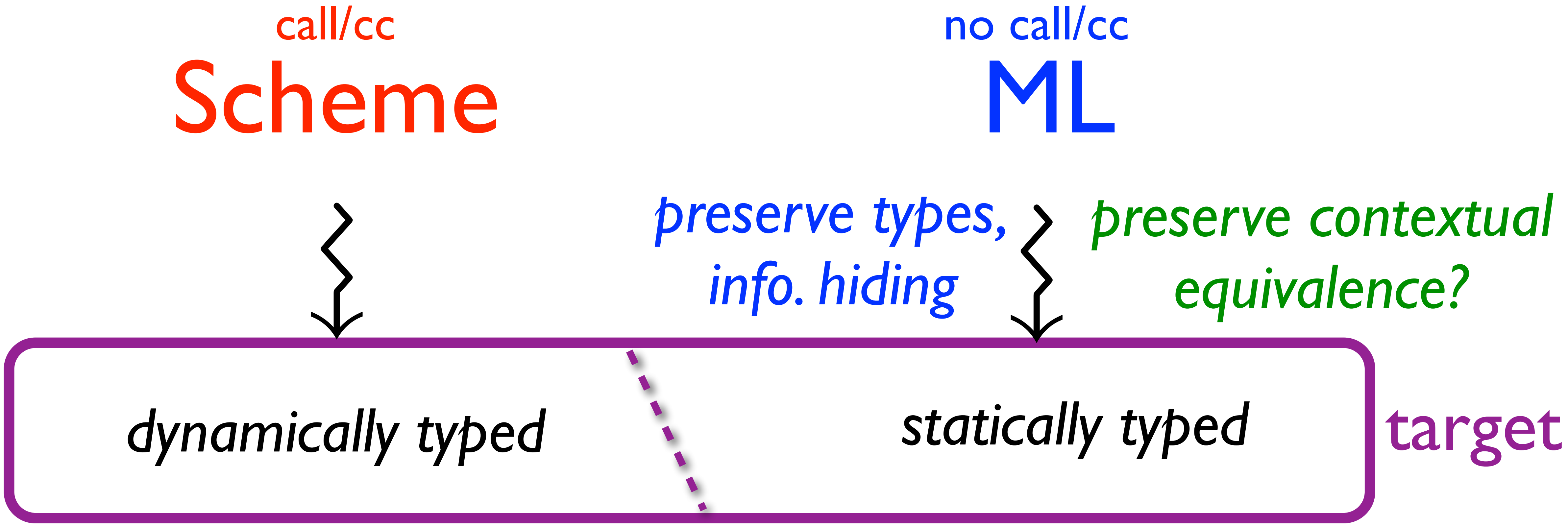
simply typed

target

dependently typed

- It's about principled language interoperability!

“Principled” Language Interoperability?



call/cc

Scheme

no call/cc

ML

preserve types, info. hiding

preserve contextual equivalence?

dynamically typed

statically typed

target

- Compiler can preserve different properties through choice of type-translation: a spectrum of linking options

Next…

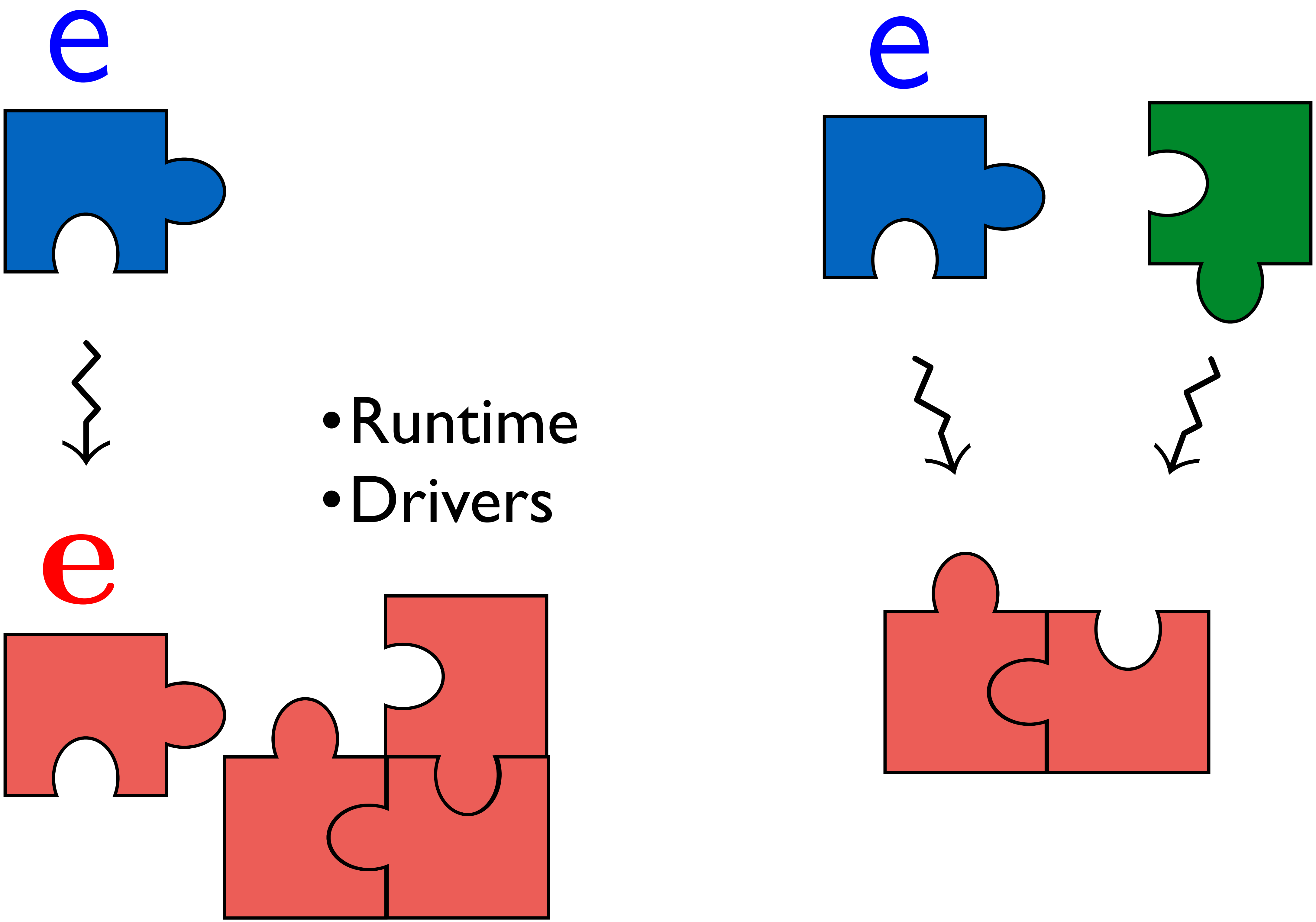
- Fully Abstract Compilation / Secure Compilation
  - compiler is equivalence-preserving
  - ensures a compiled component does not interact with any target behavior that is inexpressible in S
  - Recent results on fully abstract compilation

Next…

- Fully Abstract Compilation / Secure Compilation
  - compiler is equivalence-preserving
  - ensures a compiled component does not interact with any target behavior that is inexpressible in S
  - Recent results on fully abstract compilation

- Do we want to link with behavior inexpressible in S? Or do we want fully abstract compilers?
  - Answer: we want both!
  - How to get there? Languages should let programmers specify what behavior they want to link with

Verified Compilers for Multi-lang. World

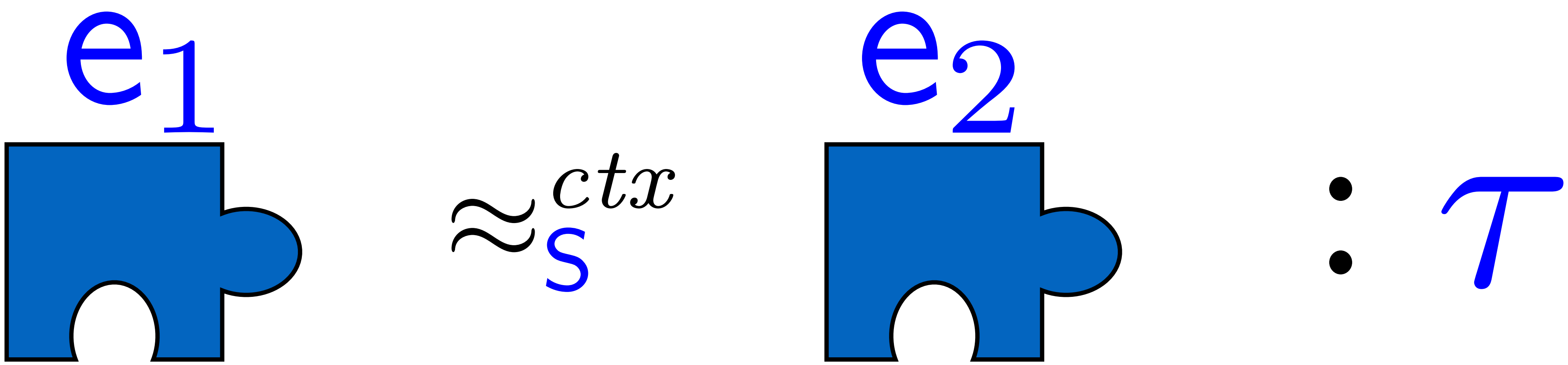


- Runtime
- Drivers

Source programmers should be able to reason in the **source** language!

Source Language Reasoning

Contextual Equivalence

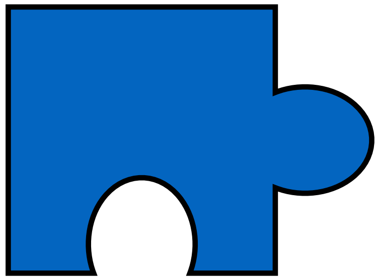


$$\mathsf{e}_1 \approx_{\mathsf{S}}^{ctx} \mathsf{e}_2 : \tau$$

Source Language Reasoning

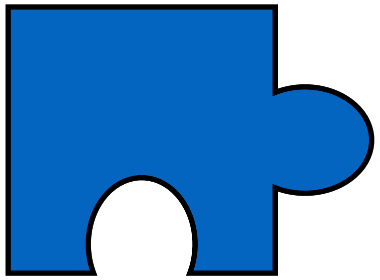
Contextual Equivalence

e_1



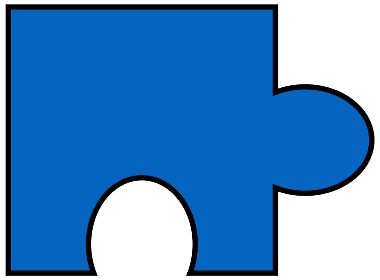
$\approx_{S}^{ctx}$

e_2



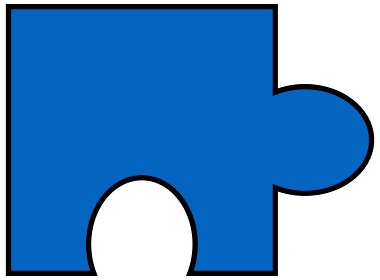
$: \tau$

e_1



is indistinguishable from

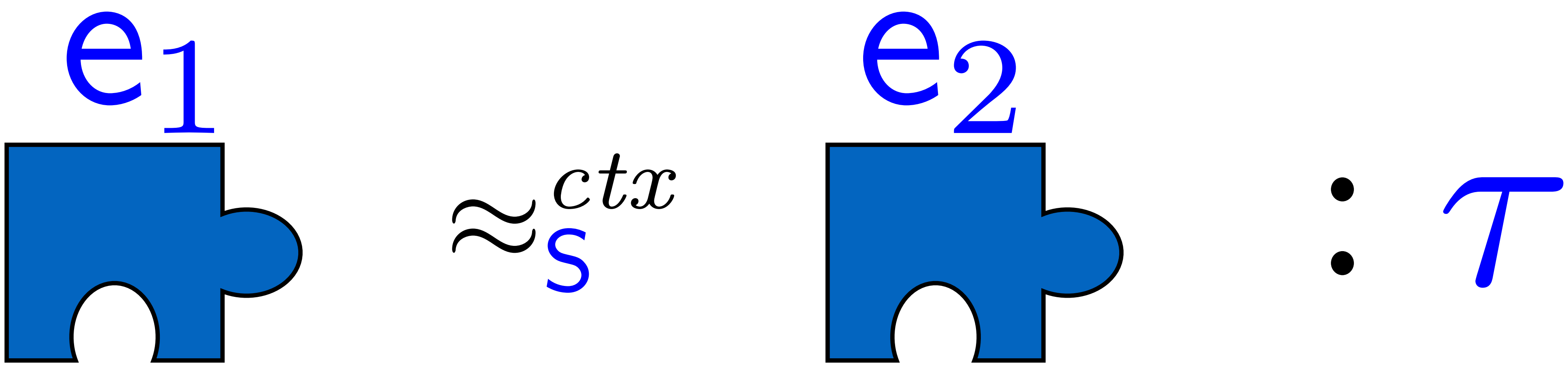
e_2



by contexts at type τ

Source Language Reasoning

Contextual Equivalence



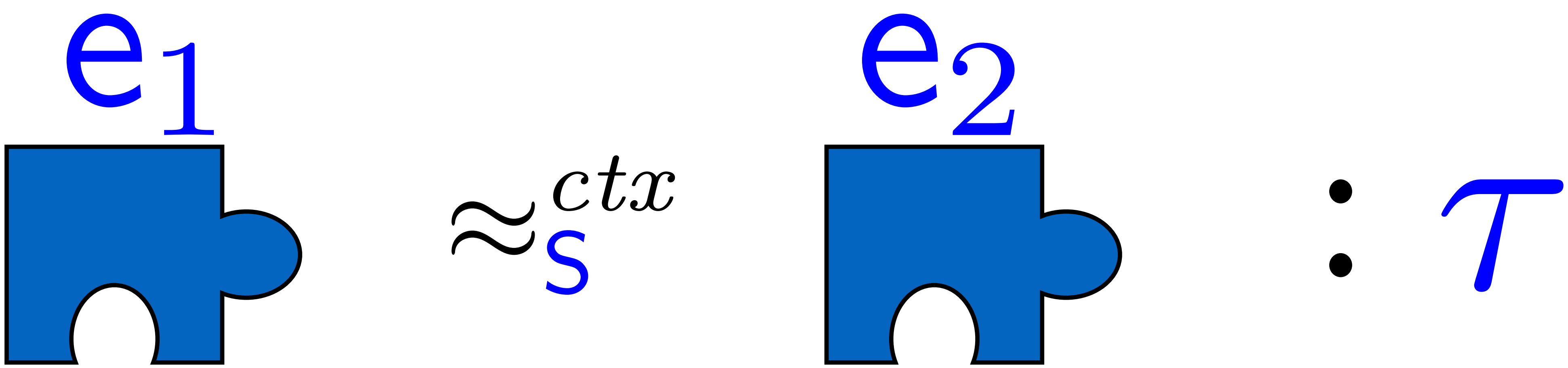
$$e_1 \approx^{ctx}_{S} e_2 : \tau$$

Formal basis for

- Refactoring

Source Language Reasoning

Contextual Equivalence



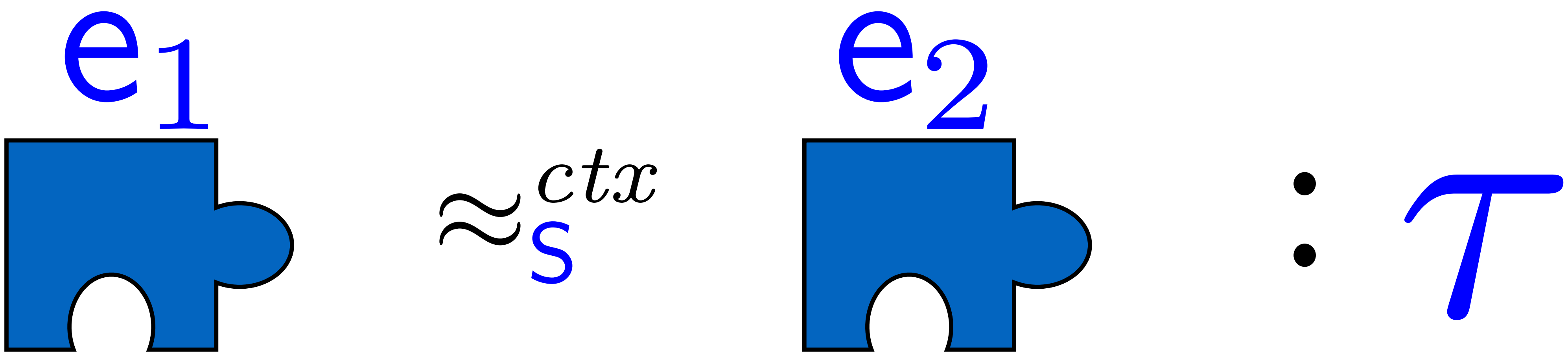
$$e_1 \approx^{ctx}_{S} e_2 : \tau$$

Formal basis for

- Refactoring
- Data abstraction

Source Language Reasoning

Contextual Equivalence



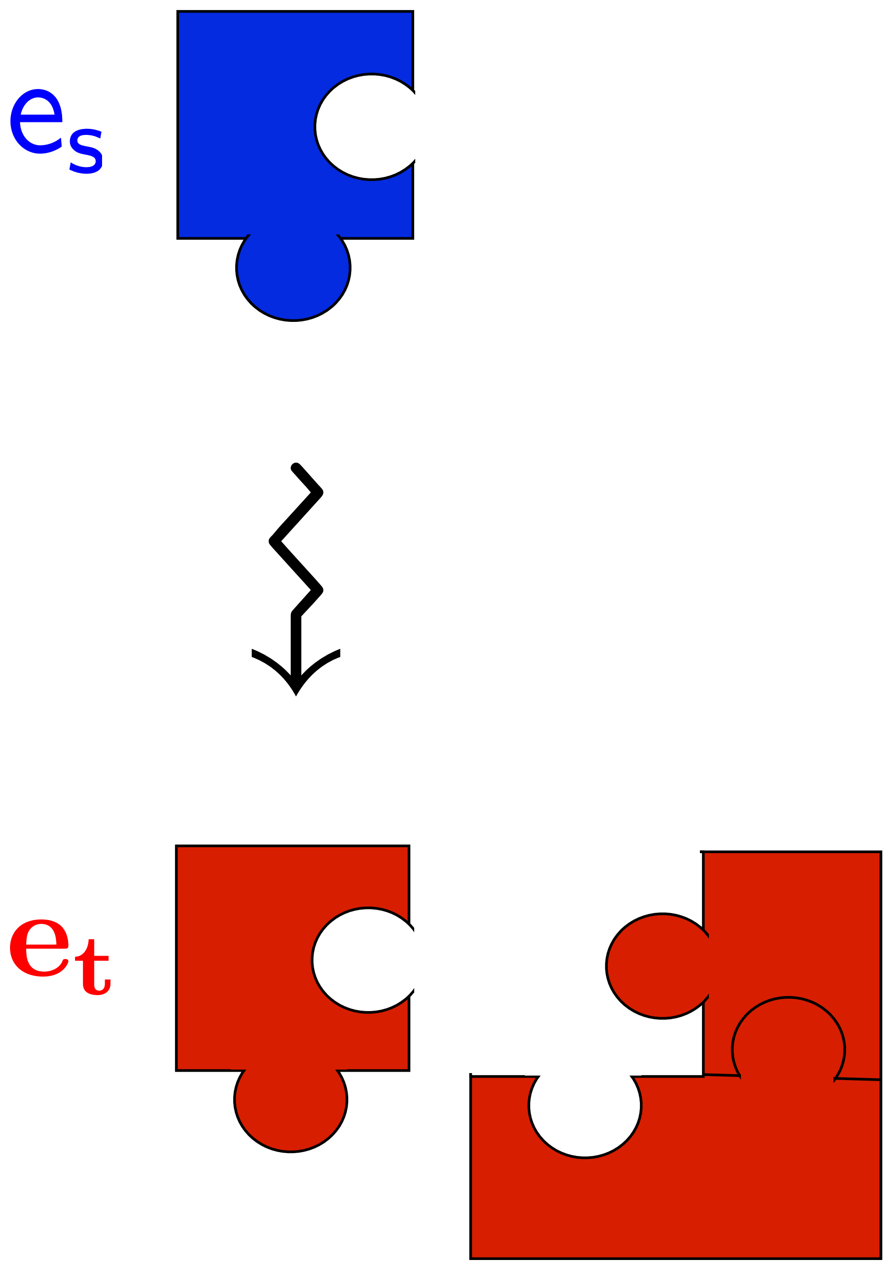
$$\mathsf{e}_1 \approx^{ctx}_{\mathsf{S}} \mathsf{e}_2 : \tau$$

Formal basis for

- Refactoring
- Data abstraction
- Security

Fully Abstract / Secure Compilation

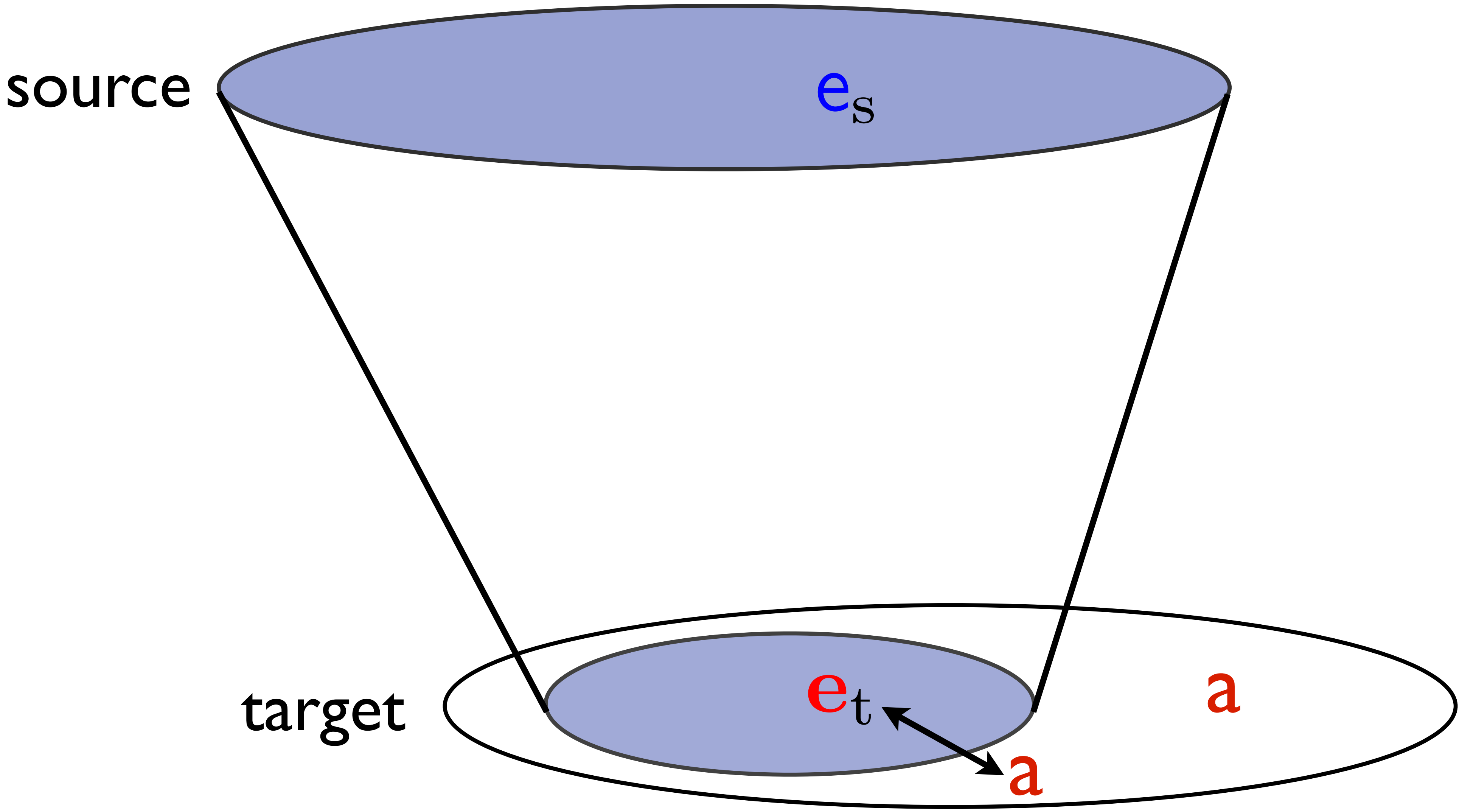
Secure compilation of components:



Want guarantee that e_t will remain as secure as e_s when executed in arbitrary target-level contexts

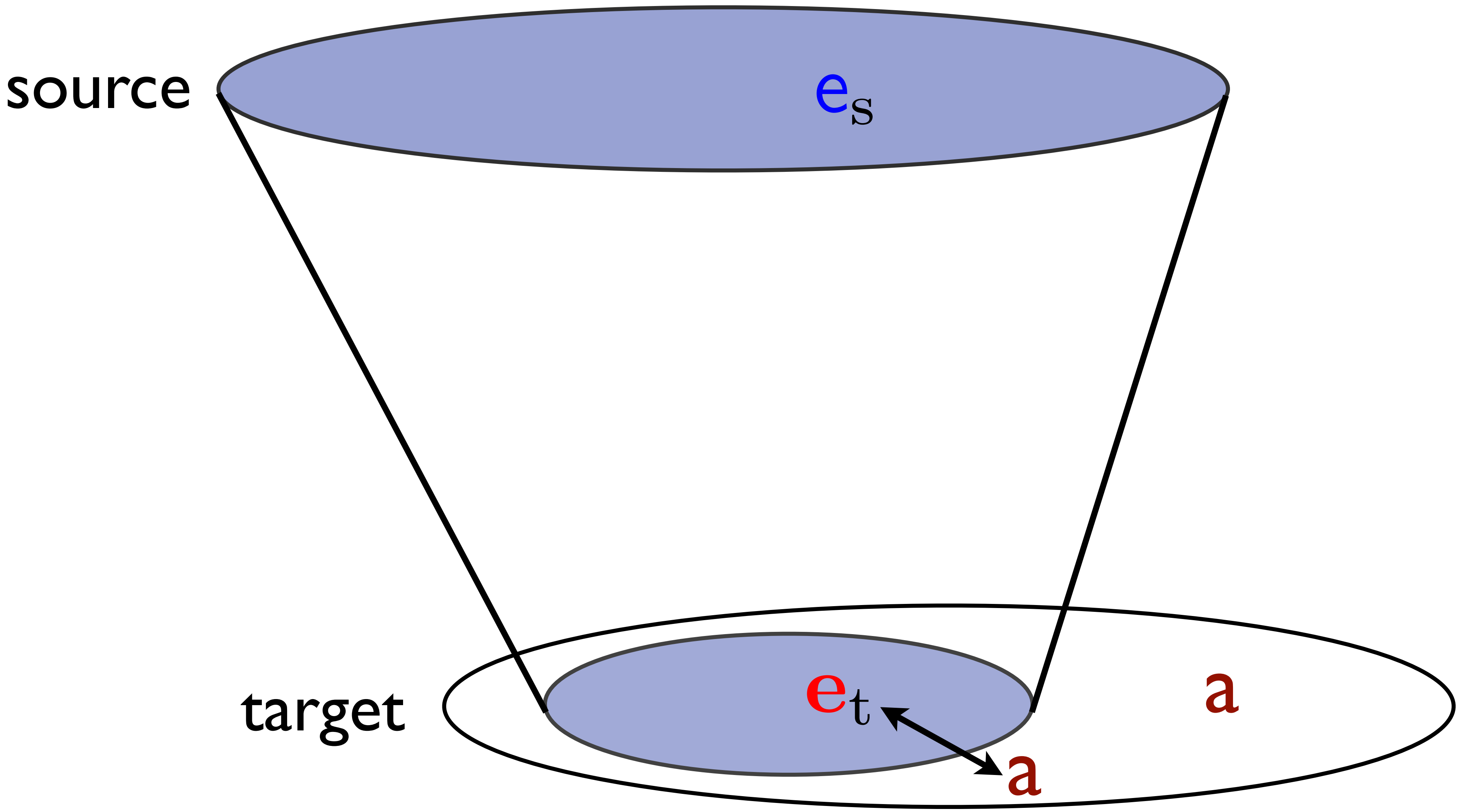
i.e. target contexts (attackers!) can make no more observations about e_t than a source context can make about e_s

Ensuring Secure Compilation



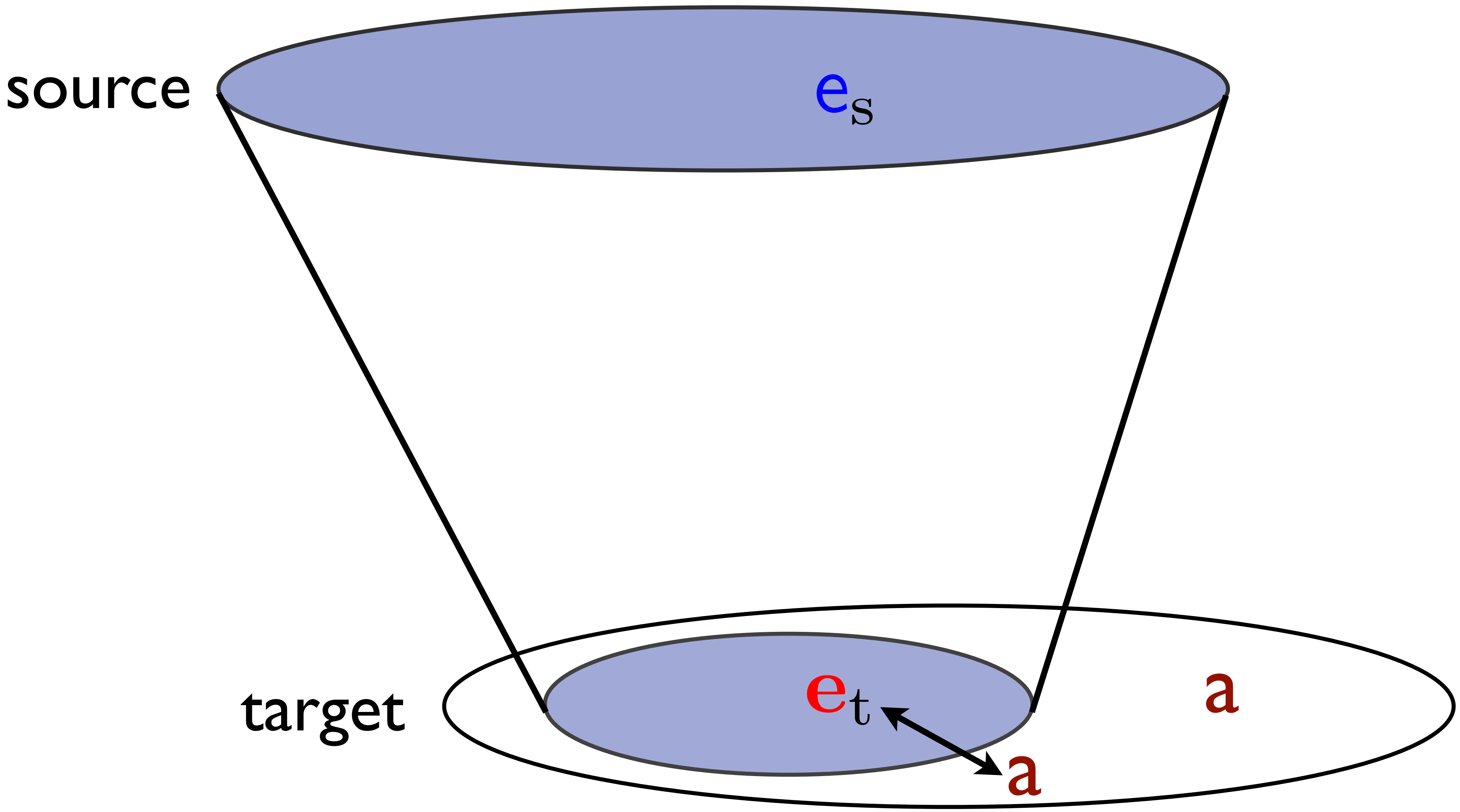
Must ensure that any a we link with behaves like some source context

Ensuring Secure Compilation



1. Add target features to the source language. **Bad!**

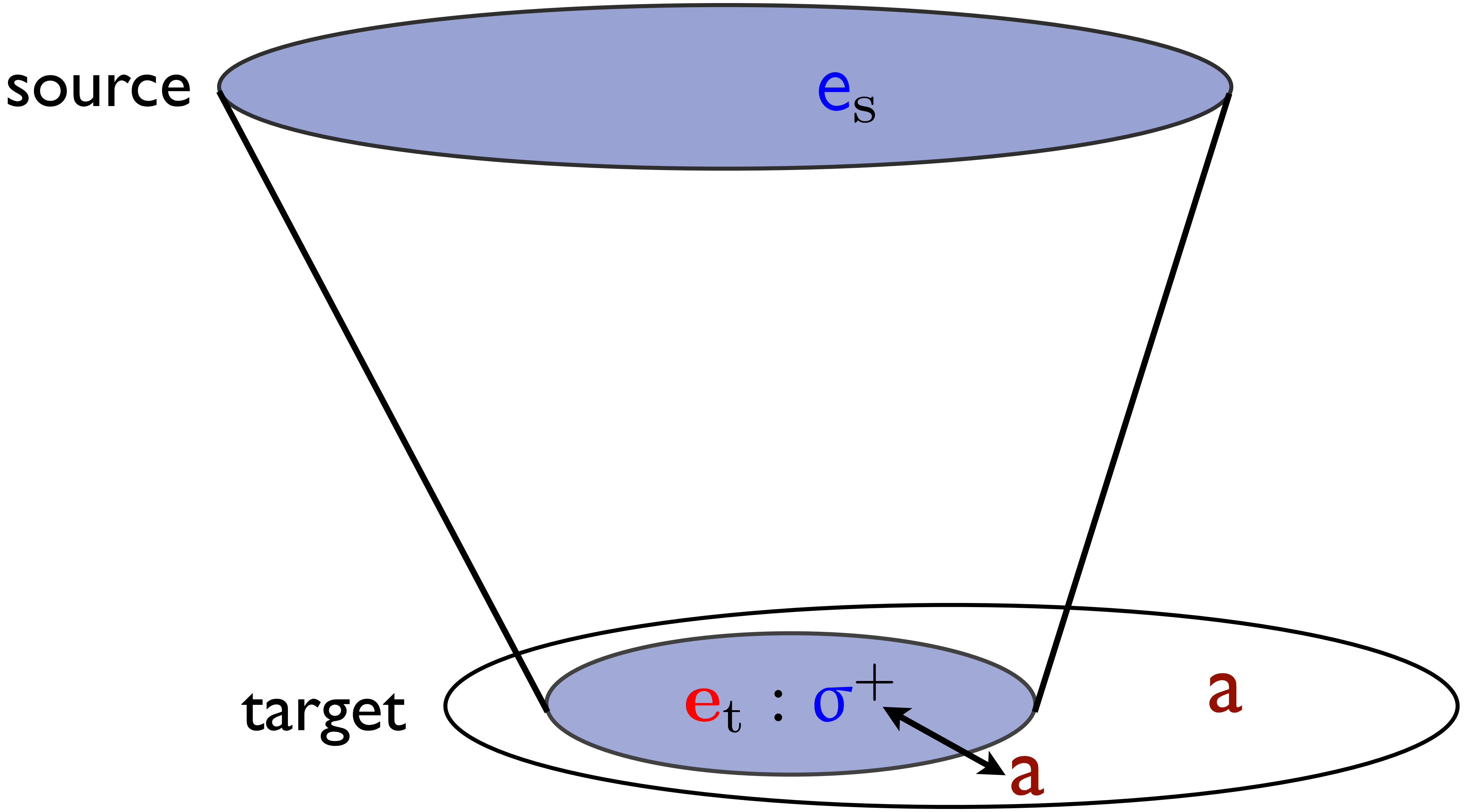
Ensuring Secure Compilation



1. Add target features to the source language. **Bad!**
2. Dynamics checks: catch badly behaved code in the act.

Performance cost

Ensuring Secure Compilation



1. Add target features to the source language. **Bad!**
2. Dynamics checks: catch badly behaved code in the act. **Performance cost**
3. Static checks: rule our badly behaved code in the first place **Verification**

Type-preserving compilation

$$\mathbf{e} : \tau \rightsquigarrow \mathbf{e} : \tau^{+}$$

Equivalence-preserving compilation

If $\mathsf{e}_1 : \tau \rightsquigarrow \mathbf{e}_1 : \tau^+$ and $\mathsf{e}_2 : \tau \rightsquigarrow \mathbf{e}_2 : \tau^+$ then:

$$\mathsf{e}_1 \approx^{ctx}_{\mathrm{S}} \mathsf{e}_2 : \tau \implies \mathbf{e}_1 \approx^{ctx}_{\mathrm{T}} \mathbf{e}_2 : \tau^+$$

Fully abstract compilation

If $\mathsf{e}_1 : \tau \rightsquigarrow \mathbf{e}_1 : \tau^+$ and $\mathsf{e}_2 : \tau \rightsquigarrow \mathbf{e}_2 : \tau^+$ then:

$$\mathsf{e}_1 \approx^{ctx}_{\mathrm{S}} \mathsf{e}_2 : \tau \iff \mathbf{e}_1 \approx^{ctx}_{\mathrm{T}} \mathbf{e}_2 : \tau^+$$

preserves & reflects equivalence

Challenge of proving full abstraction

Suppose $\Gamma \vdash \mathsf{e}_1 : \tau \rightsquigarrow \mathbf{e}_1$ and $\Gamma \vdash \mathsf{e}_2 : \tau \rightsquigarrow \mathbf{e}_2$.

$$\begin{tikzcd}
\Gamma \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{S}} \mathsf{e}_2 : \tau \arrow[d] \\
\Gamma^{+} \vdash \mathbf{e}_1 \approx^{ctx}_{\mathrm{T}} \mathbf{e}_2 : \tau^{+}
\end{tikzcd}$$

Challenge of proving full abstraction

Suppose $\Gamma \vdash \mathsf{e}_1 : \tau \rightsquigarrow \mathbf{e}_1$ and $\Gamma \vdash \mathsf{e}_2 : \tau \rightsquigarrow \mathbf{e}_2$.

$$\begin{tikzcd}
\Gamma \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{S}} \mathsf{e}_2 : \tau \arrow[d] \\
\Gamma^{+} \vdash \mathbf{e}_1 \approx^{ctx}_{\mathrm{T}} \mathbf{e}_2 : \tau^{+}
\end{tikzcd}$$

Given:

No C_{S} can distinguish e_1, e_2

Challenge: Back-translation

1. If target is not more expressive than source, use the same language: back-translation can be avoided in lieu of *wrappers* between τ and τ^+

   - Closure conversion: System F with recursive types
     [Ahmed-Blume ICFP'08]

   - f* (STLC with refs) to js* (encoding of JavaScript in f*)
     [Fournet et al. POPL'13]

Challenge: Back-translation

2. If target is more expressive than source

   (a) Both terminating: use back-translation by partial evaluation

   - Equivalence-preserving CPS from STLC to System F *[Ahmed-Blume ICFP'11]*
   - Noninterference for Free (DCC to Fω) *[Bowman-Ahmed ICFP'15]*

   (b) Both nonterminating: use ??
   back-trans by partial evaluation is not well-founded!

   Observation: our source lang. has recursive types,
   can write interpreter for target lang. in source lang.

Fully Abstract Closure Conversion

[New et al.'16]

Source: STLC + μ types

Target: System F + $\exists$ types + μ types + exceptions

First full abstraction result where target has exceptions but source does not.

Earlier work, due to lack of sufficiently powerful back-translation techniques, adds features from target to source.

Novel proof technique — **Universal Embedding**

- Untyped embedding of target in source
- Mediate between strongly typed source and untyped back-translation

Fully Abstract Closure Conversion

Source: STLC + μ types

Target: System F + $\exists$ types + μ types + exceptions

Equivalent source terms, inequivalent in lang. with exceptions:

$$\mathrm{e}_1 = \lambda \mathrm{f}.\, (\mathrm{f}\ \mathrm{true}; \mathrm{f}\ \mathrm{false}; \langle\rangle) \qquad \mathrm{e}_2 = \lambda \mathrm{f}.\, (\mathrm{f}\ \mathrm{false}; \mathrm{f}\ \mathrm{true}; \langle\rangle)$$

$$C = \mathrm{catch}\ \mathrm{y} = ([\cdot]\ (\lambda \mathrm{x}.\, \mathrm{raise}\, \mathrm{x}))\ \mathrm{in}\ \mathrm{y}$$

$$C[\mathrm{e}_1] \Downarrow \mathrm{true} \qquad C[\mathrm{e}_2] \Downarrow \mathrm{false}$$

Idea: use modal type system at target to rule out linking with code that throws unhandled exceptions

Ensuring Full Abstraction

$$\mathsf{e}_1 \approx_{\mathsf{S}}^{ctx} \mathsf{e}_2 : (\mathsf{bool} \rightarrow 1) \rightarrow 1$$

$$\mathbf{(bool \rightarrow E\,0\,1) \rightarrow E\,0\,1}$$

$$\neq$$

$$\mathbf{C} : \mathbf{(bool \rightarrow E\,bool\,1) \rightarrow E\,bool\,1}$$

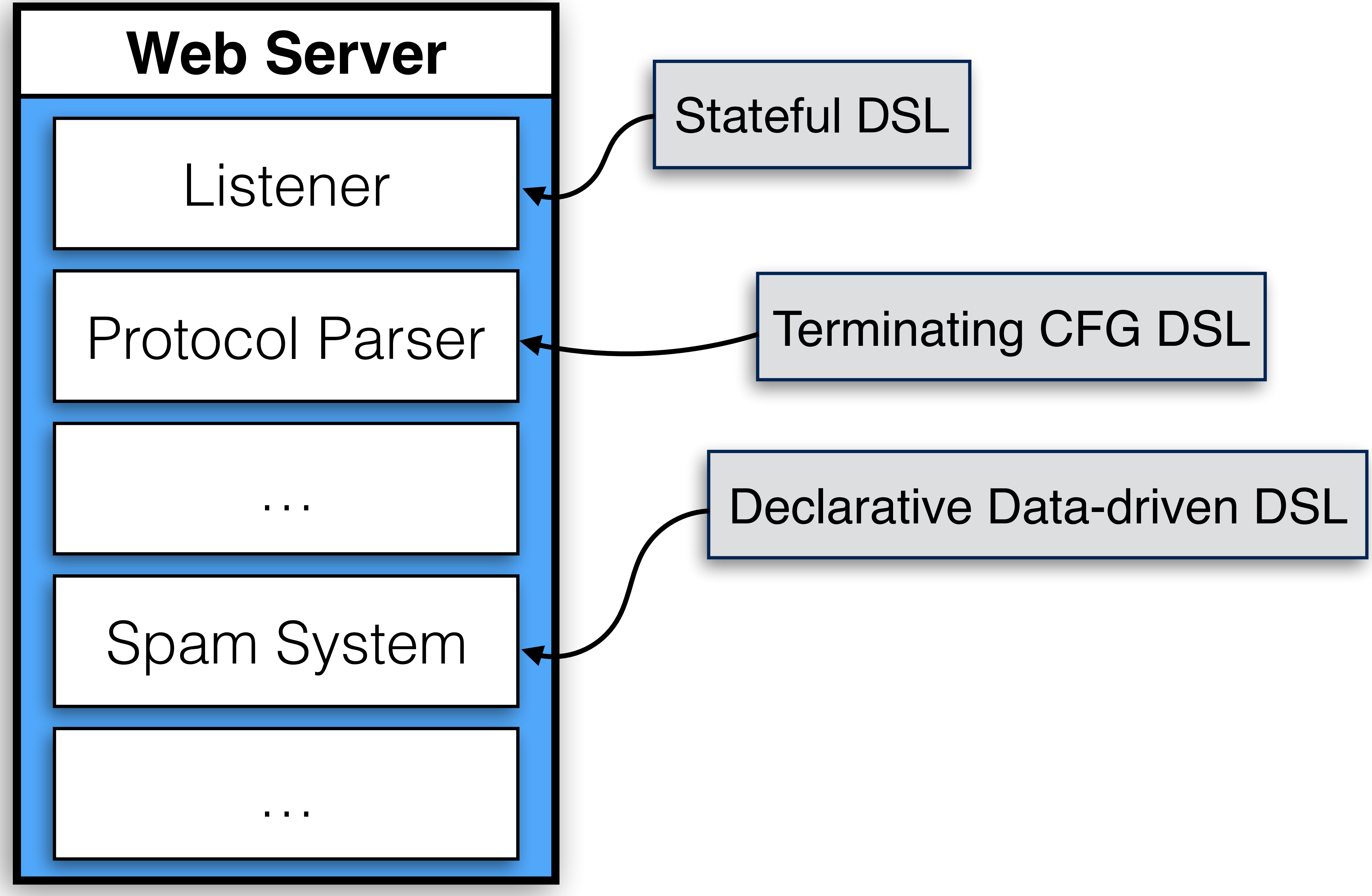
$$\mathbf{C} = (\,[\cdot]\ \boldsymbol{\lambda}\mathbf{(x : bool).\,raise\ x})$$

Static Fully Abstract Compilation

- Type checking ensures that we never link with target code whose (extensional) behavior does not match *some* source behavior

- But what if we want to link with behaviors unavailable in the source?
  - Surely, we want that when building multi-language software!

Multi-Language System



Web Server

Listener

Protocol Parser

. . .

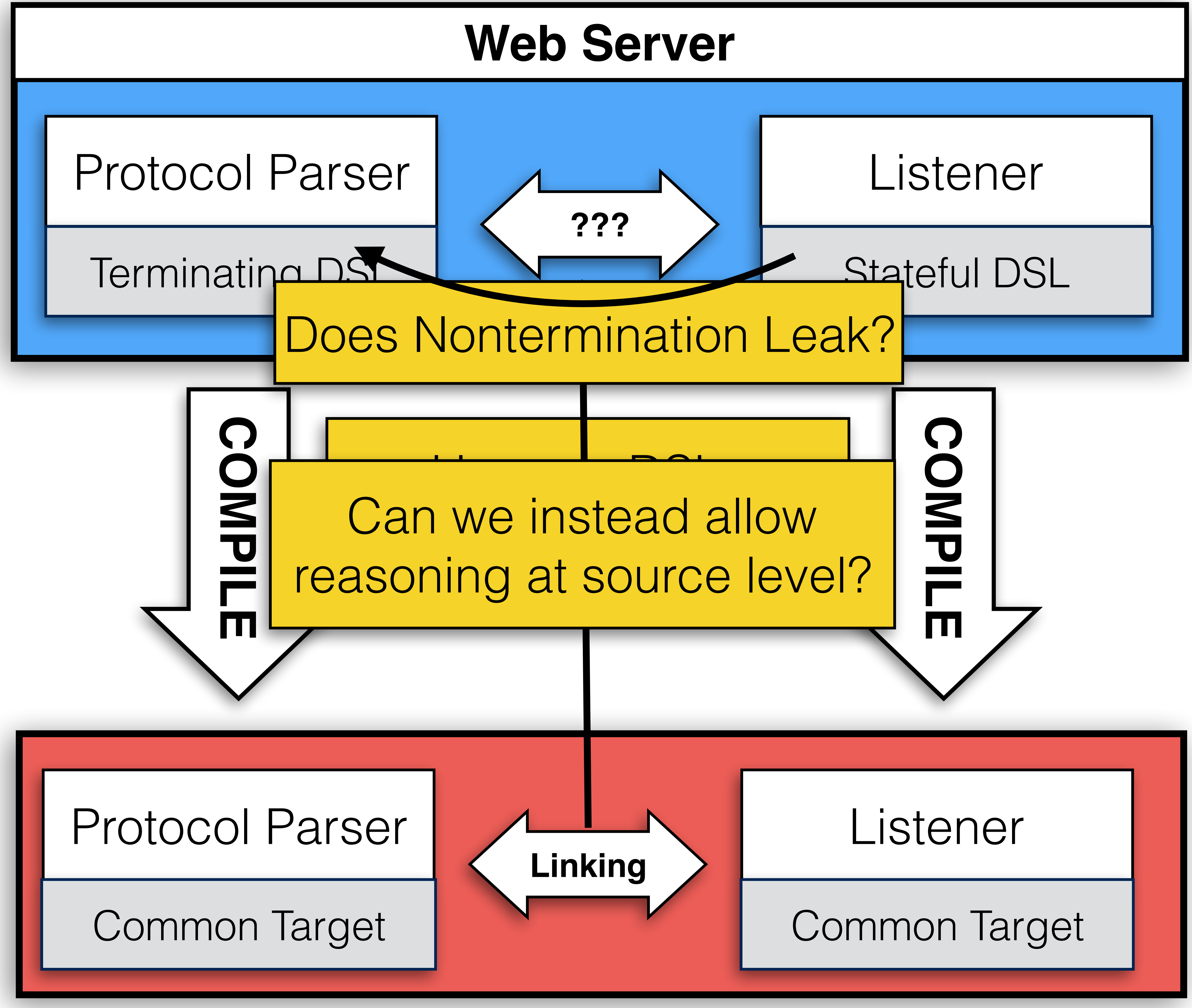
Spam System

. . .

Stateful DSL

Terminating CFG DSL

Declarative Data-driven DSL



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

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

In a Simpler Setting

λ (simply-typed lambda calculus)

λ^{ref} (extended with ML references)

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

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

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

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

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

Refactoring is reasoning about equivalence

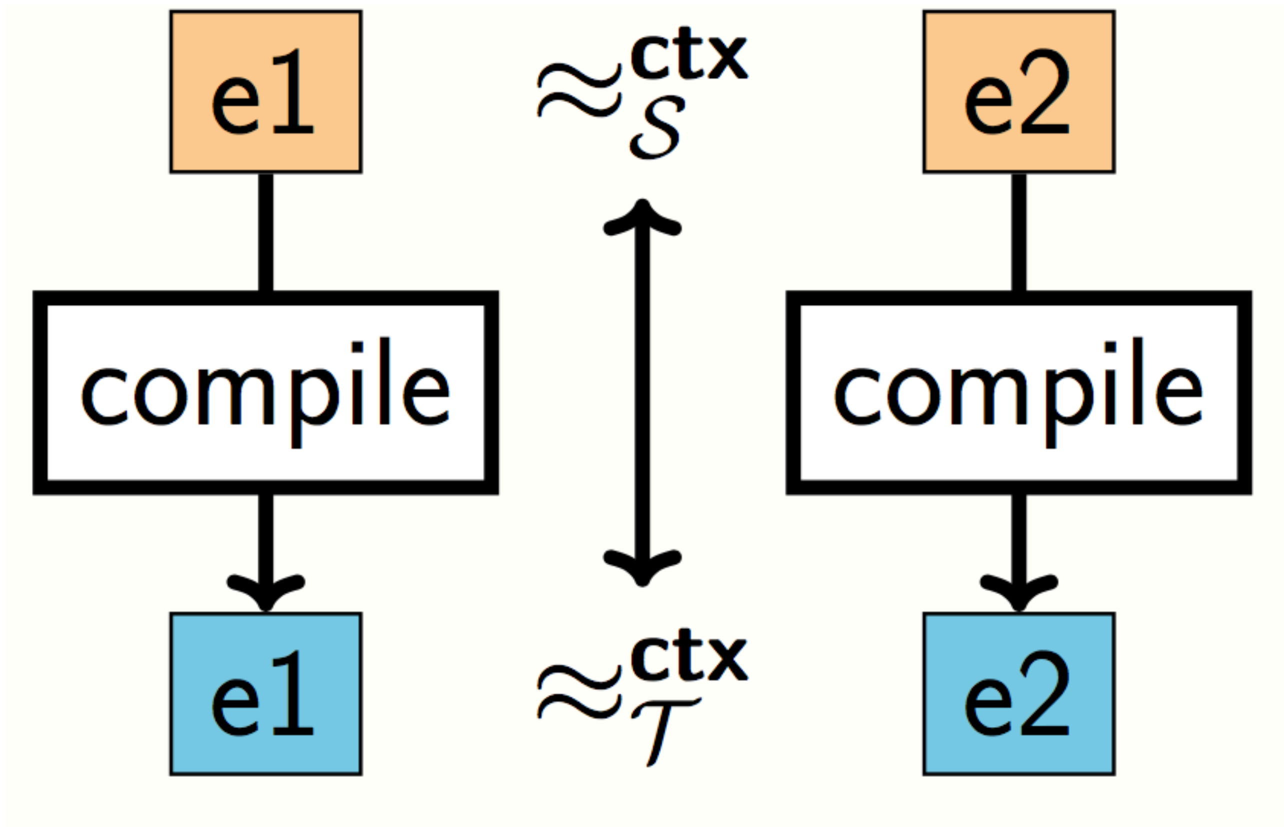
Reasoning About Refactoring

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

Should be okay because

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

Fully abstract compilers preserve equivalences



What about linking with λ^{ref}?

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

but

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

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!

$\mathbf{unit} \to \mathbf{int}$ ✓ $\mathsf{unit} \to \mathsf{int}$ ✗

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

λ with linking types extension

$$\tau \quad ::= \quad \textbf{unit} \mid \textbf{int} \mid \tau \rightarrow \tau$$

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

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

λ^{κ} Allows Programmers To Write Both

$\mathbf{unit} \rightarrow \mathbf{int}$

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

$\mathsf{unit} \rightarrow \mathsf{int}$

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

Refactoring: Pure Inputs

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

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

Ill-typed, since `f` requires pure code

Refactoring: Impure Inputs

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

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

Well-typed, since `f` accepts impure code

Minimal Annotation Burden

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

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

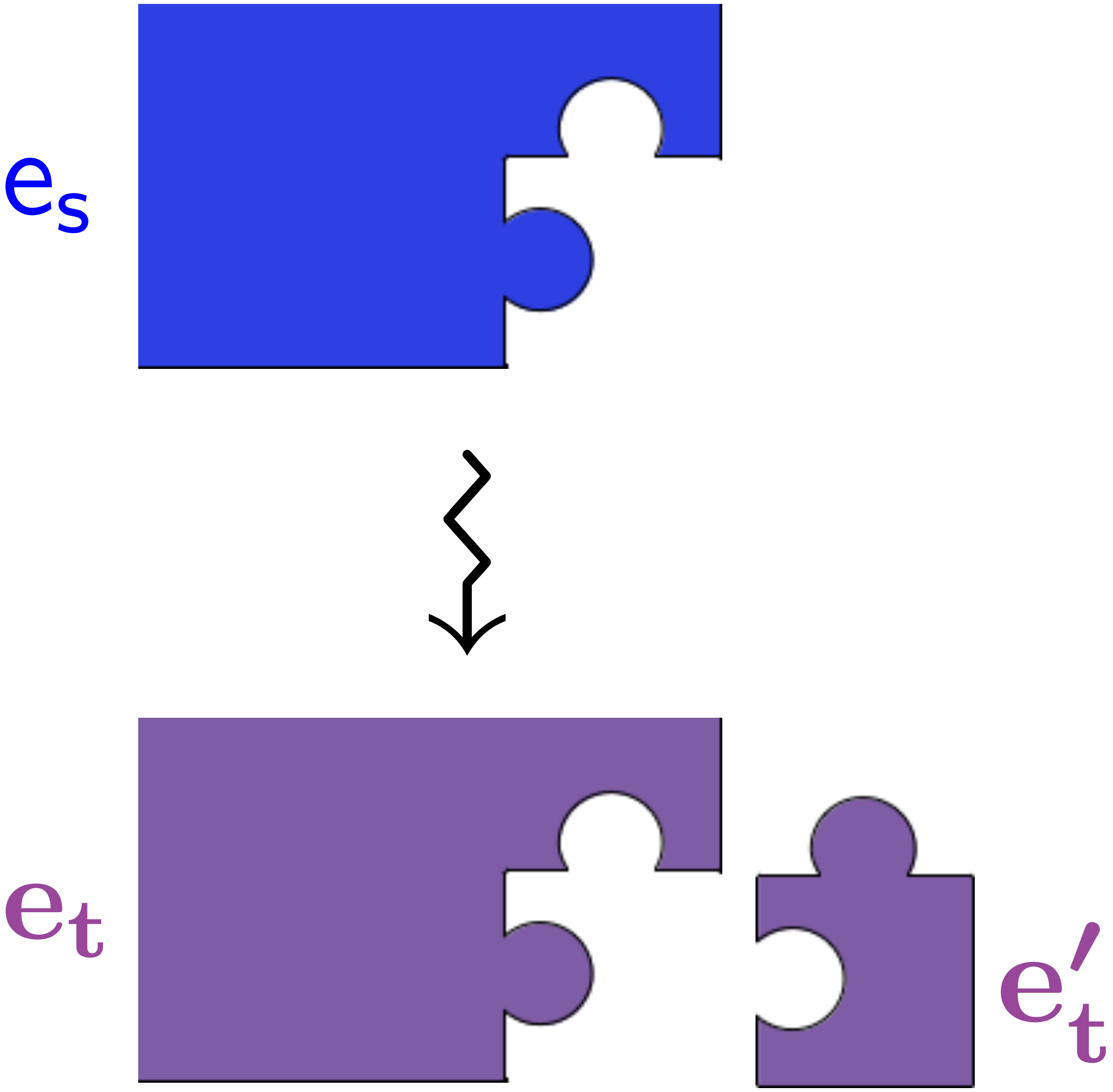
λ^{κ} must provide default translation

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

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

Stepping Back…

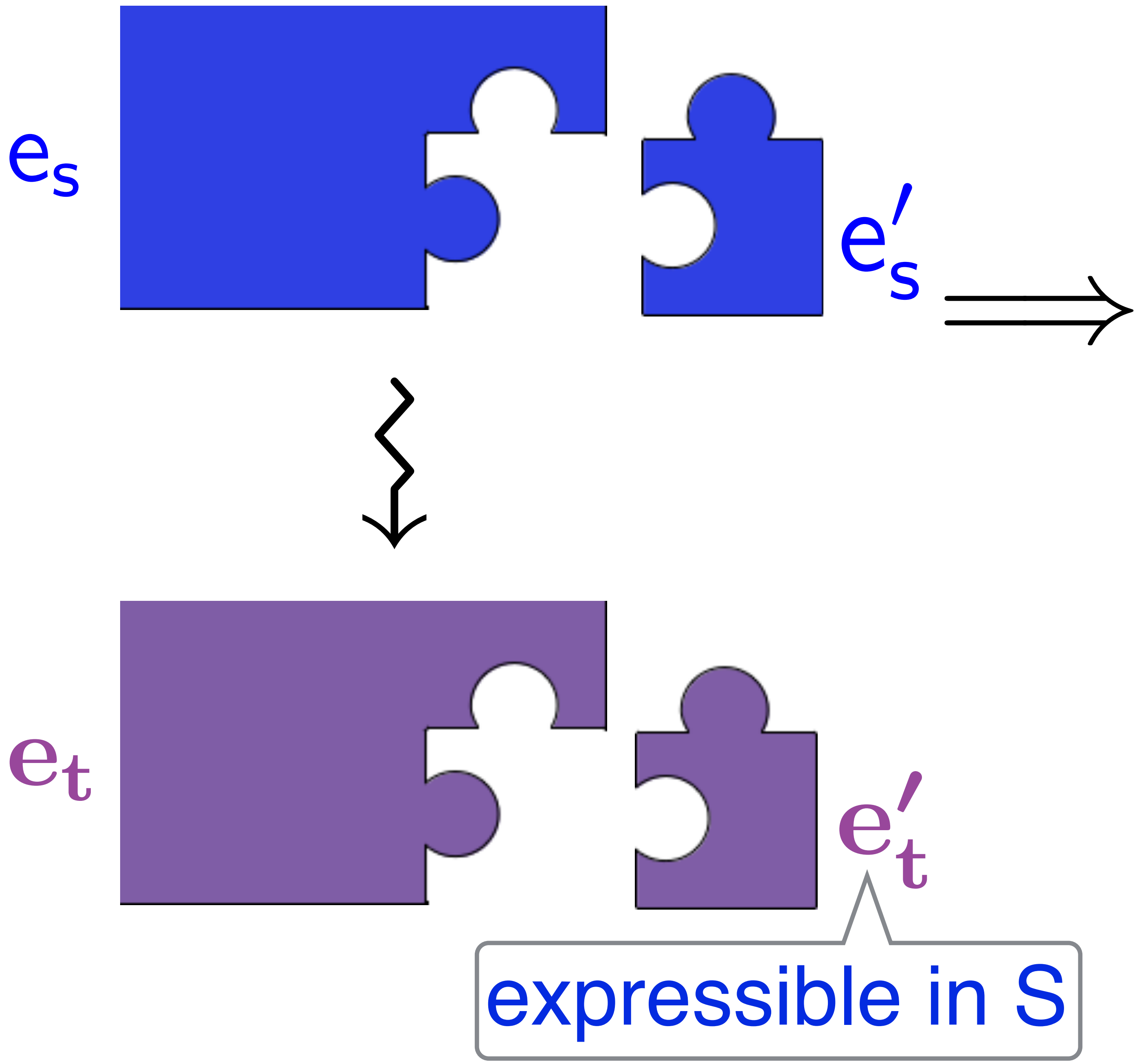
Correct Compilation of Components



specifies behaviors compiled code may be linked with

$$\Longrightarrow \qquad e_S \approx e_T$$

Correct Compilation of Components



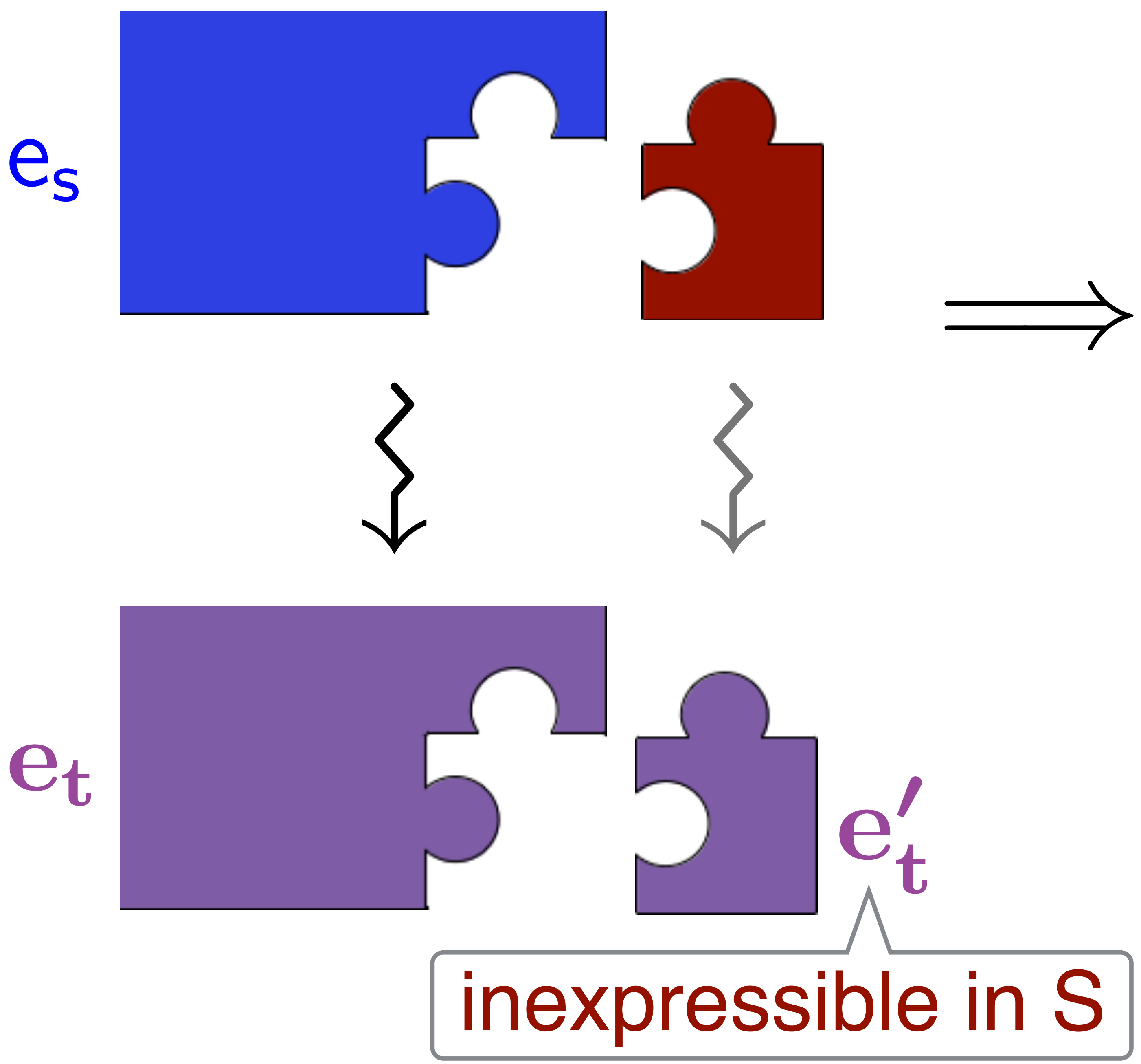
expressible in S

specifies behaviors
compiled code may
be linked with

$$e_S \approx e_T$$

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

Correct Compilation: Multi-Language



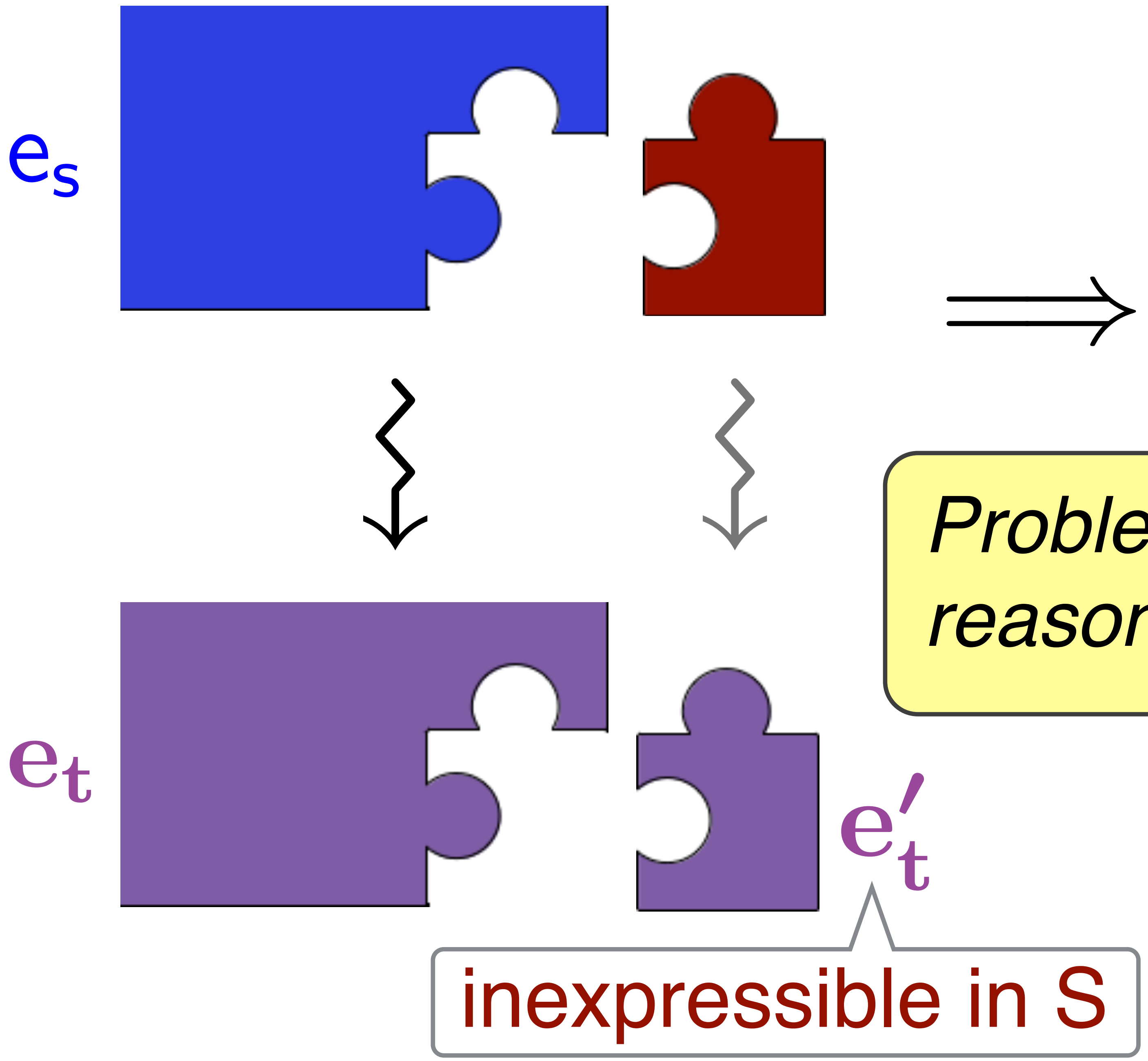
inexpressible in S

specifies behaviors compiled code may be linked with

$$e_S \approx e_T$$

- *[Perconti-Ahmed'14]*
- *Verified Compilers for a Multi-Language World [Ahmed SNAPL'15]*

Correct Compilation: Multi-Language



$$\Longrightarrow \quad e_S \approx e_T$$

Problem: programmer cannot reason at source level!

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

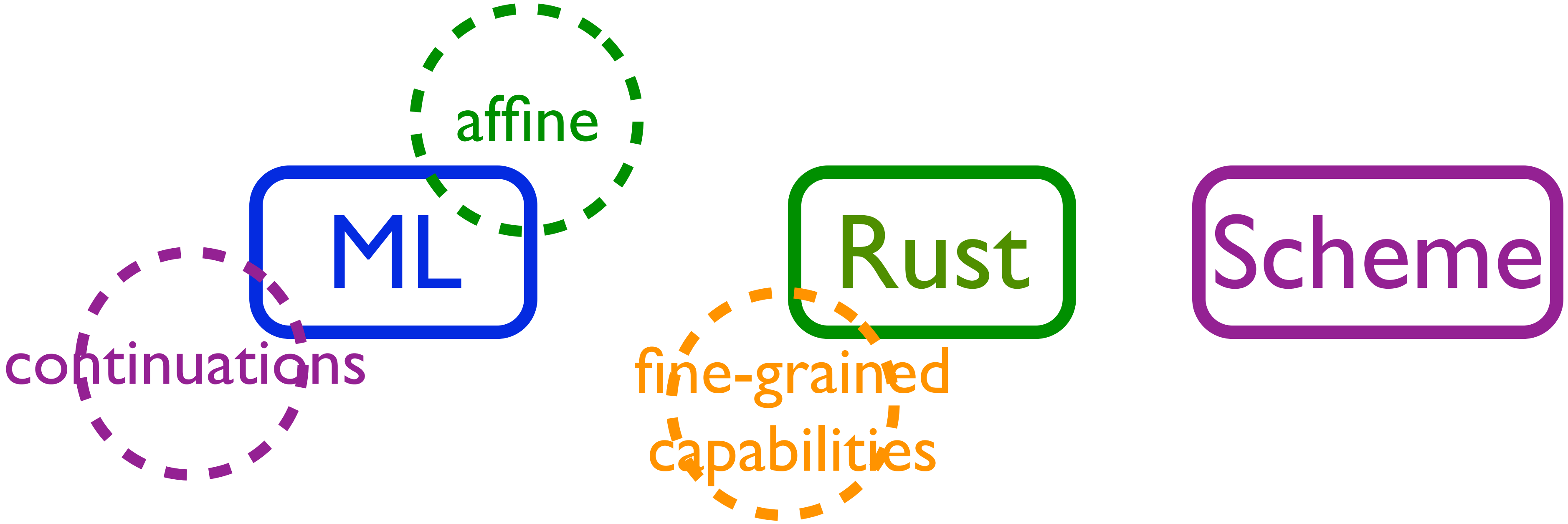
ML — C FFI

Rust — unsafe

Java — JNI

Design linking types extensions that support safe interoperability with other languages

PL Design, Linking Types



affine

ML

continuations

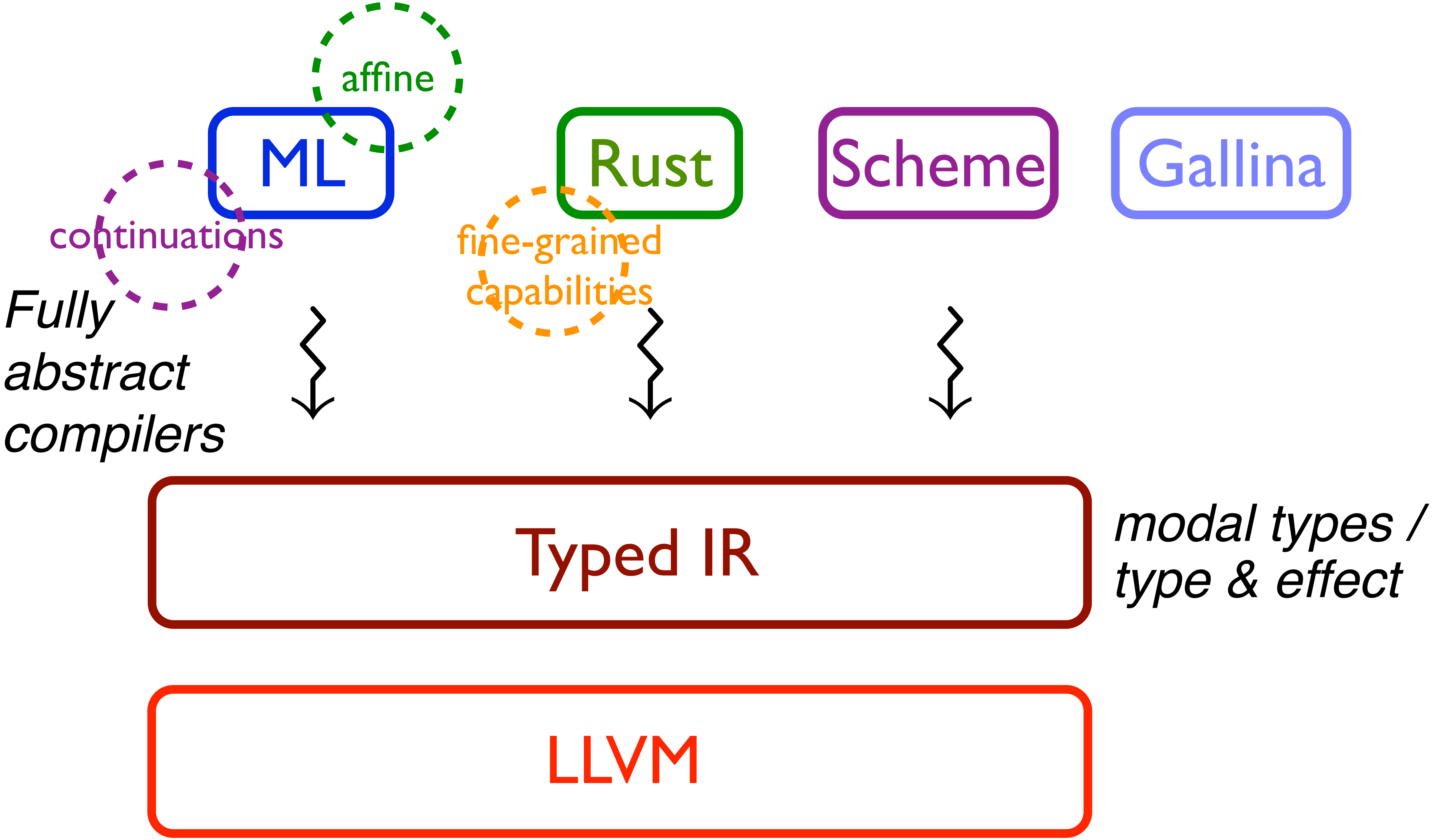
Rust

fine-grained capabilities

Scheme

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

PL Design, Linking Types, Compilers



affine

ML

Rust

Scheme

Gallina

continuations

fine-grained capabilities

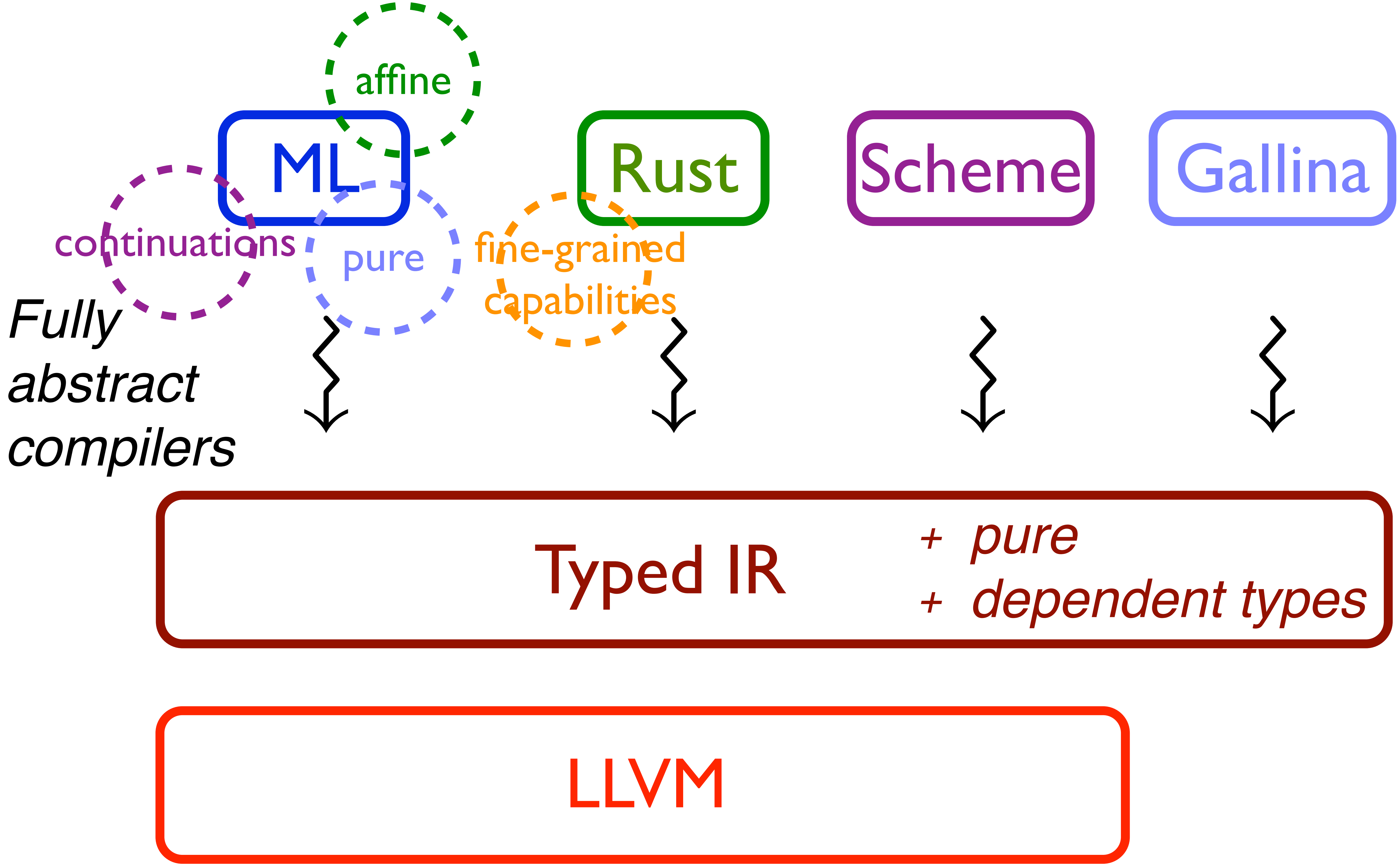
Fully abstract compilers

Typed IR

modal types / type & effect

LLVM

PL Design, Linking Types, Compilers



ML

Rust

Scheme

Gallina

affine

continuations

pure

fine-grained capabilities

Fully abstract compilers

Typed IR

\+ *pure*

\+ *dependent types*

LLVM

Linking Types

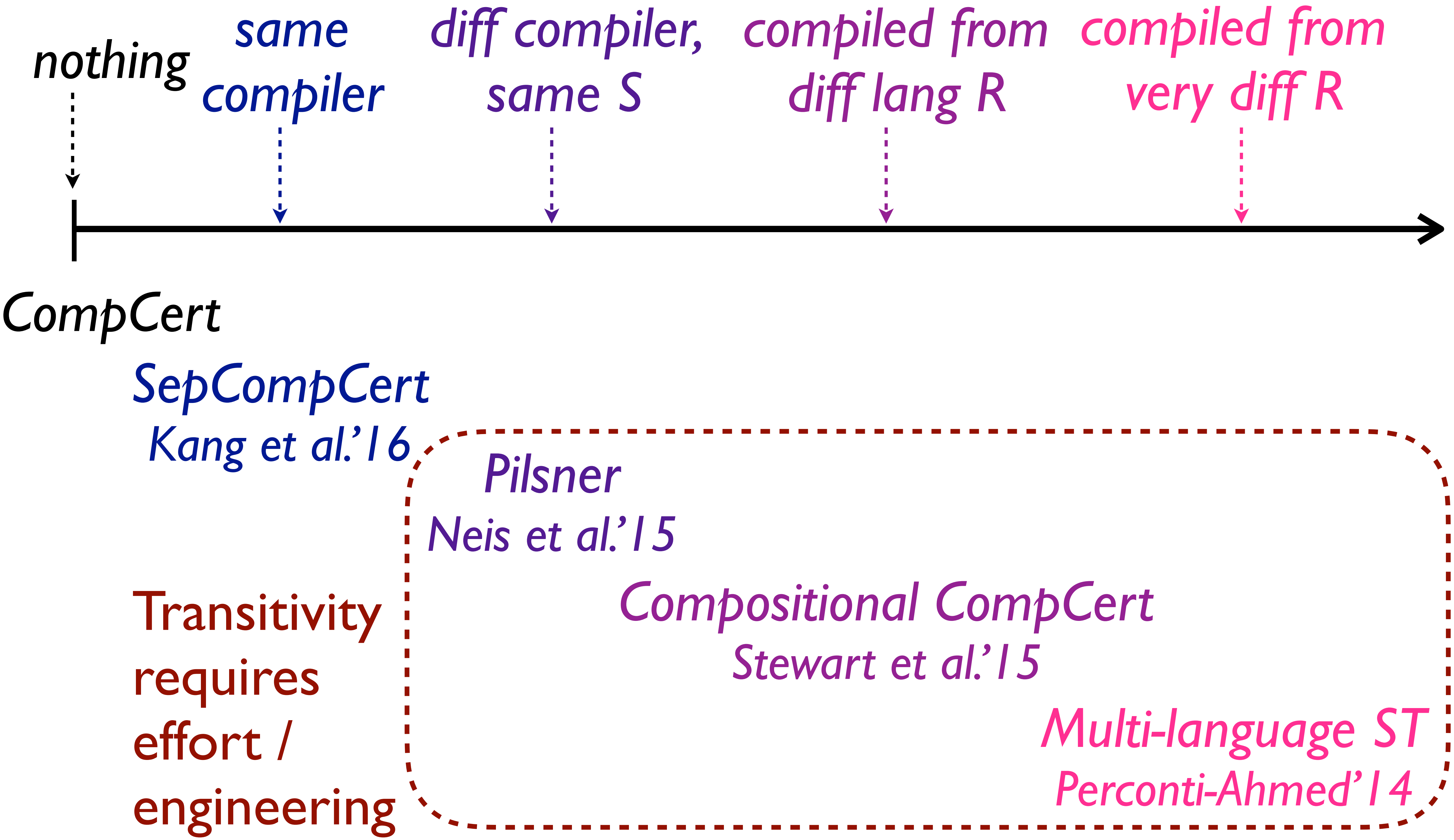
- Allow programmers to reason in *almost* their own source languages, even when building multi-language software
- Allow compilers to be fully abstract, yet support multi-language linking

Conclusion

Compiler Verif. for Multi-Lang. World

- Compositional Compiler Correctness
  - horizontal and vertical compositionality

Horizontal / Vertical Compositionality



nothing

same compiler

diff compiler, same S

compiled from diff lang R

compiled from very diff R

CompCert

SepCompCert
Kang et al.'16

Pilsner
Neis et al.'15

Compositional CompCert
Stewart et al.'15

Multi-language ST
Perconti-Ahmed'14

Transitivity requires effort / engineering

Compiler Verif. for Multi-Lang. World

- CompCert started a renaissance in compiler verification
  - major advances in mechanized proof
- Now we need: Compositional Compiler Correctness
  - but horizontal and vertical compositionality at odds
- Need to rethink proof architectures for compiler verification to support linking with code of arbitrary provenance. But want transitivity to be easier!

Verification of *realistic* compilers for a multi-language world demands formal techniques and language design

- compositional equational reasoning
- formal semantics of language interoperability
- types and logics to enforce sensible (safe, secure) linking
- extending our language designs with *principled extensions* to replace unprincipled escape hatches