Compositional Compiler Verification for a Multi-Language World

Amal Ahmed

Northeastern University & Inria Paris

Compiler Verification

One of the "big problems" of computer science

- since *McCarthy and Painter 1967:*
  Correctness of a Compiler for Arithmetic Expressions

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.

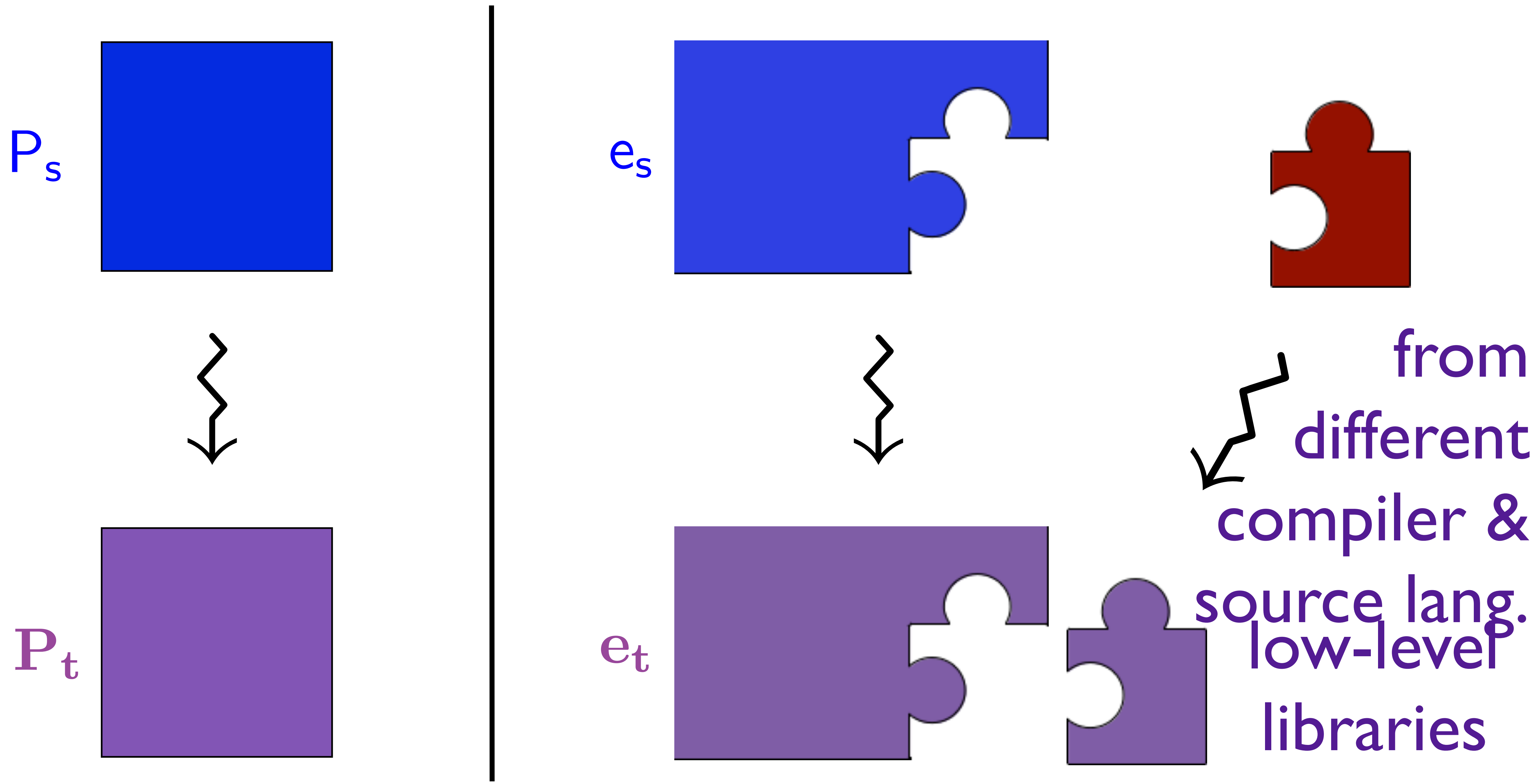
Zhao et al.'13 : Formal verification of SSA-based optimizations for LLVM

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

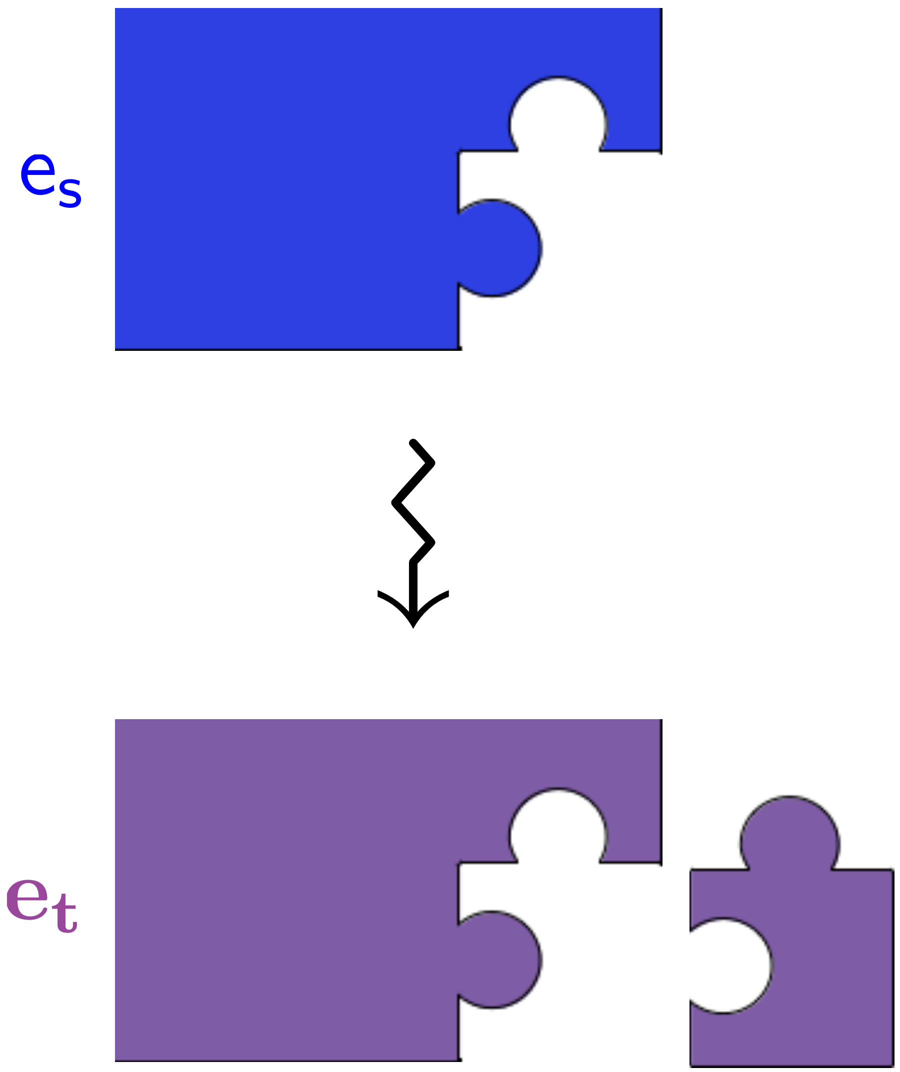
⋮

Problem: Whole-Program Assumption

Correct compilation guarantee only applies to whole programs!



“Compositional” Compiler Verification



This Talk…

- why specifying compositional compiler correctness is hard
- survey recent results
- generic CCC theorem
- lessons for formalizing linking & verifying multi-pass compilers
- language design & control over extra-linguistic features

Compiler Correctness

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

compiles to

same behavior

Compiler Correctness

$$s \rightsquigarrow t \qquad \Longrightarrow \qquad s \approx t$$

expressed how?

Whole-Program Compiler Correctness

$$P_s \rightsquigarrow P_t \quad \Longrightarrow \quad P_s \approx P_t$$

expressed how?
“closed” simulations

CompCert

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

Whole-Program Compiler Correctness

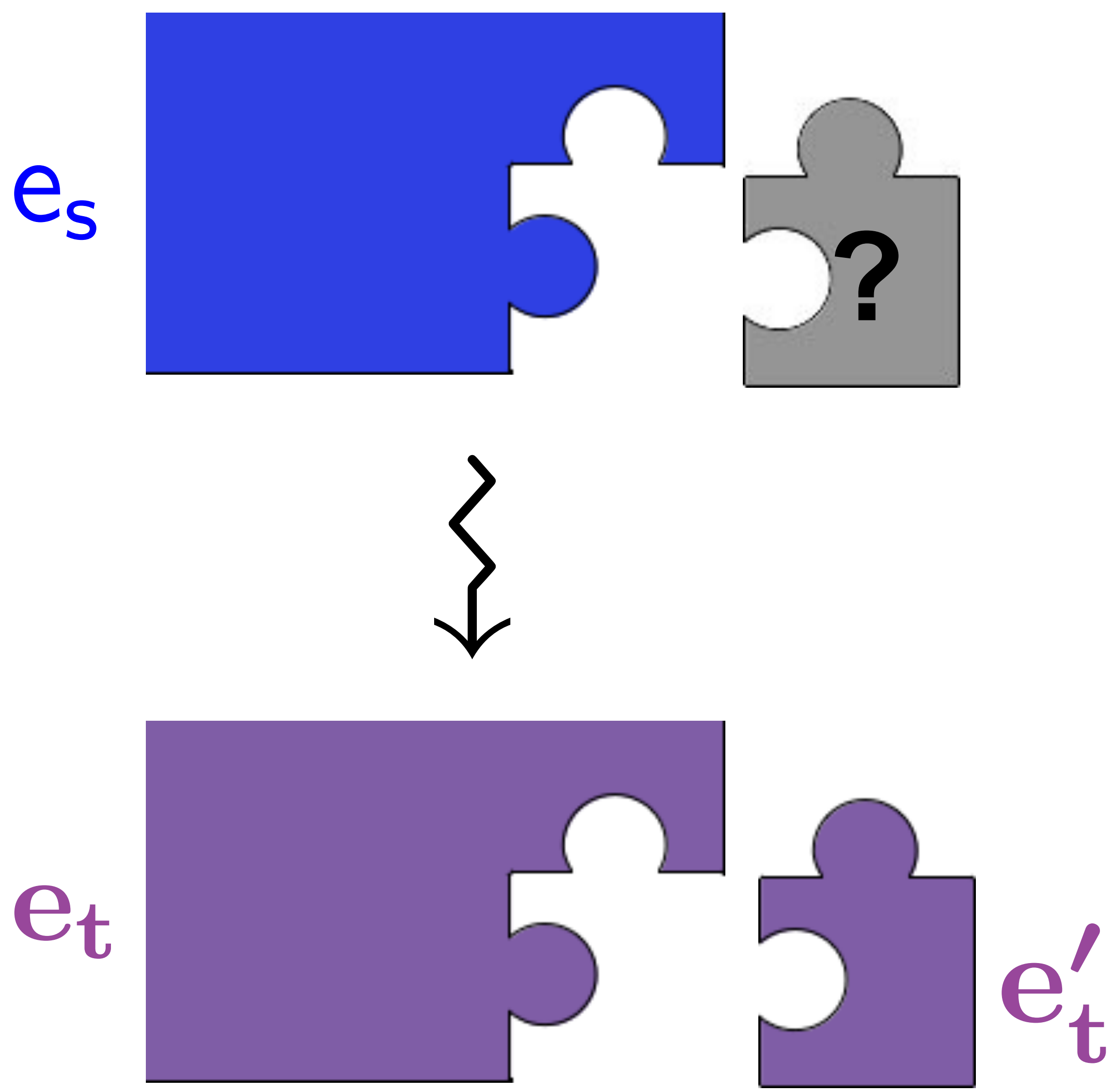
$$P_s \rightsquigarrow P_t \quad \Longrightarrow \quad P_t \sqsubseteq P_s$$

behavior refinement

$$\forall n. \quad P_t \overset{T_t}{\longmapsto}{}^{n} P_t' \implies$$

$$\exists m. \quad P_s \overset{T_s}{\longmapsto}{}^{m} P_s' \quad \wedge \quad T_t \simeq T_s$$

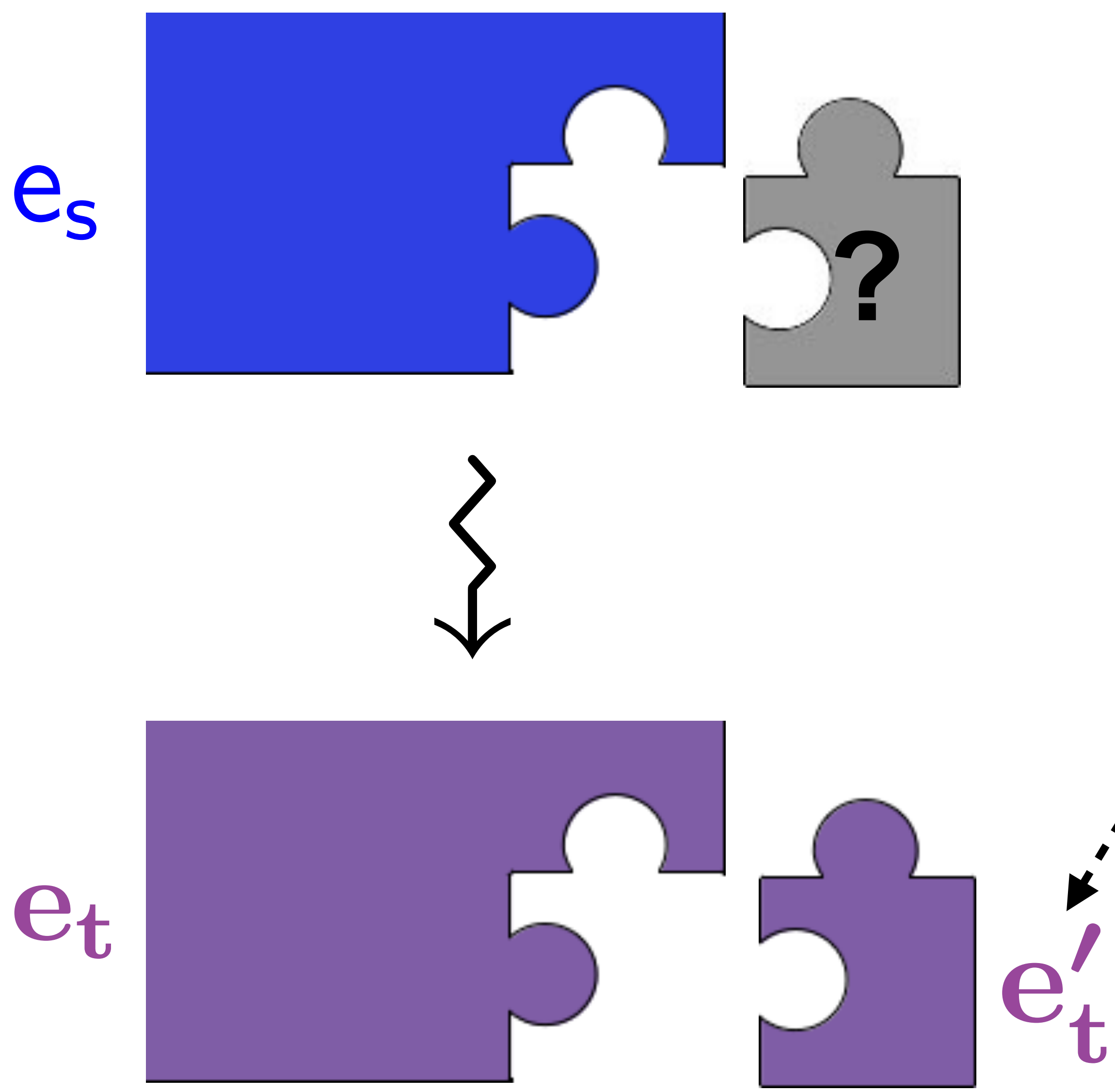
Correct Compilation of Components?



$$e_S \approx e_T$$

expressed how?

"Compositional" Compiler Correctness



$e_S \approx e_T$

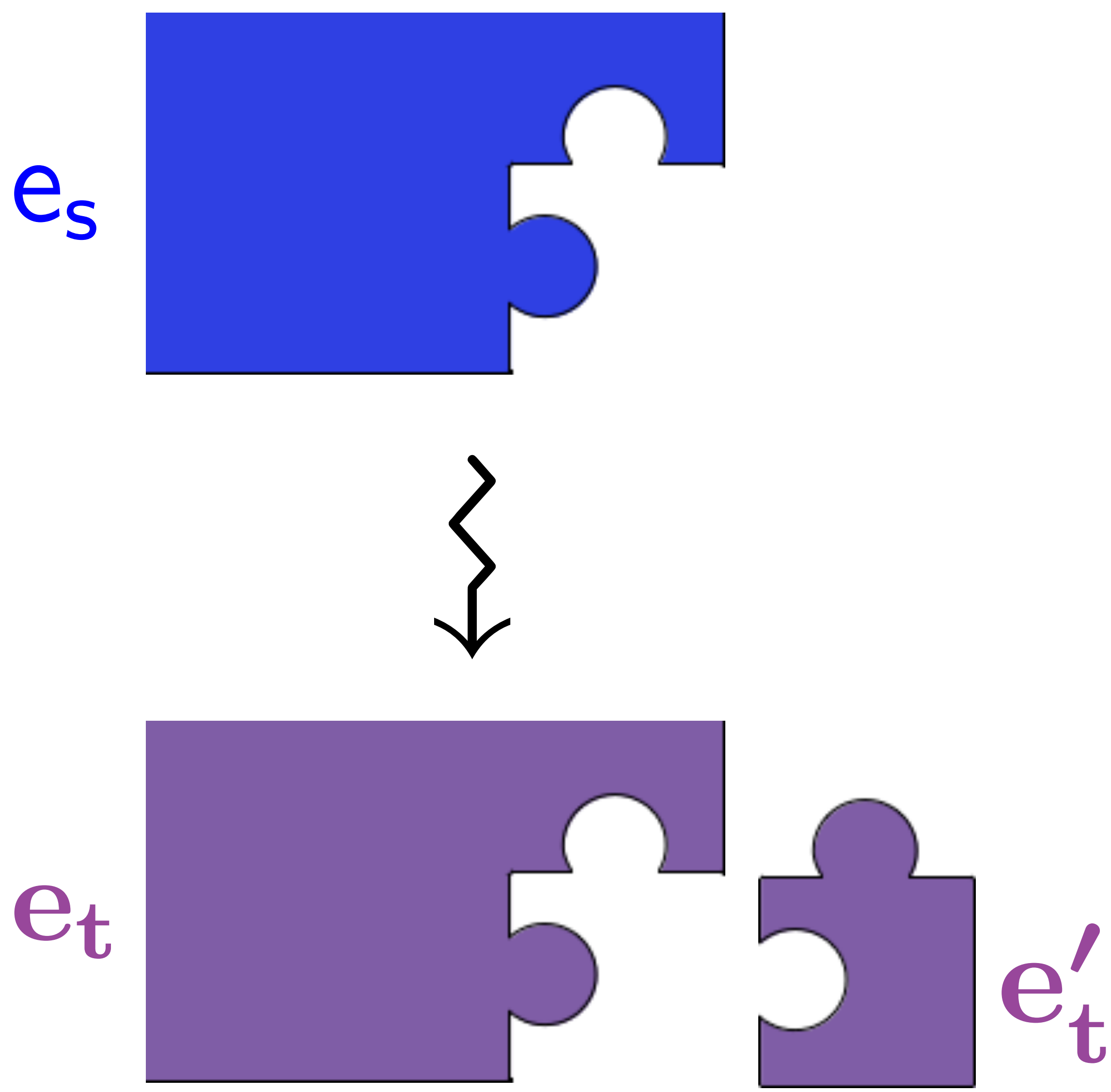
expressed how?

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

Behavior expressible in S?

“Compositional” Compiler Correctness

If we want to verify realistic compilers…



$$e_S \approx e_T$$

Definition should:

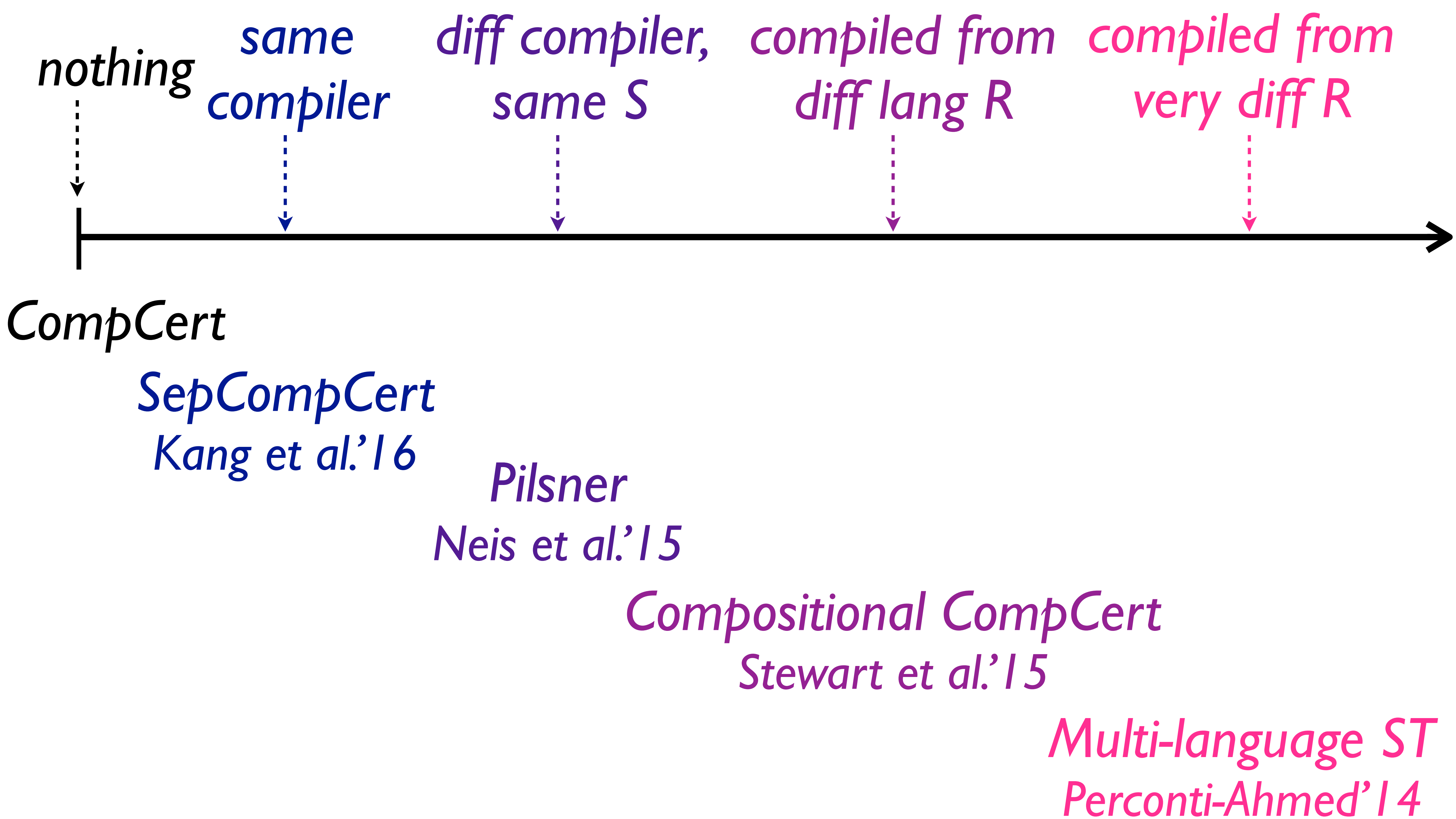
- permit **linking** with target code of arbitrary provenance
- support verification of **multi-pass** compilers

Next: Survey of State of the Art

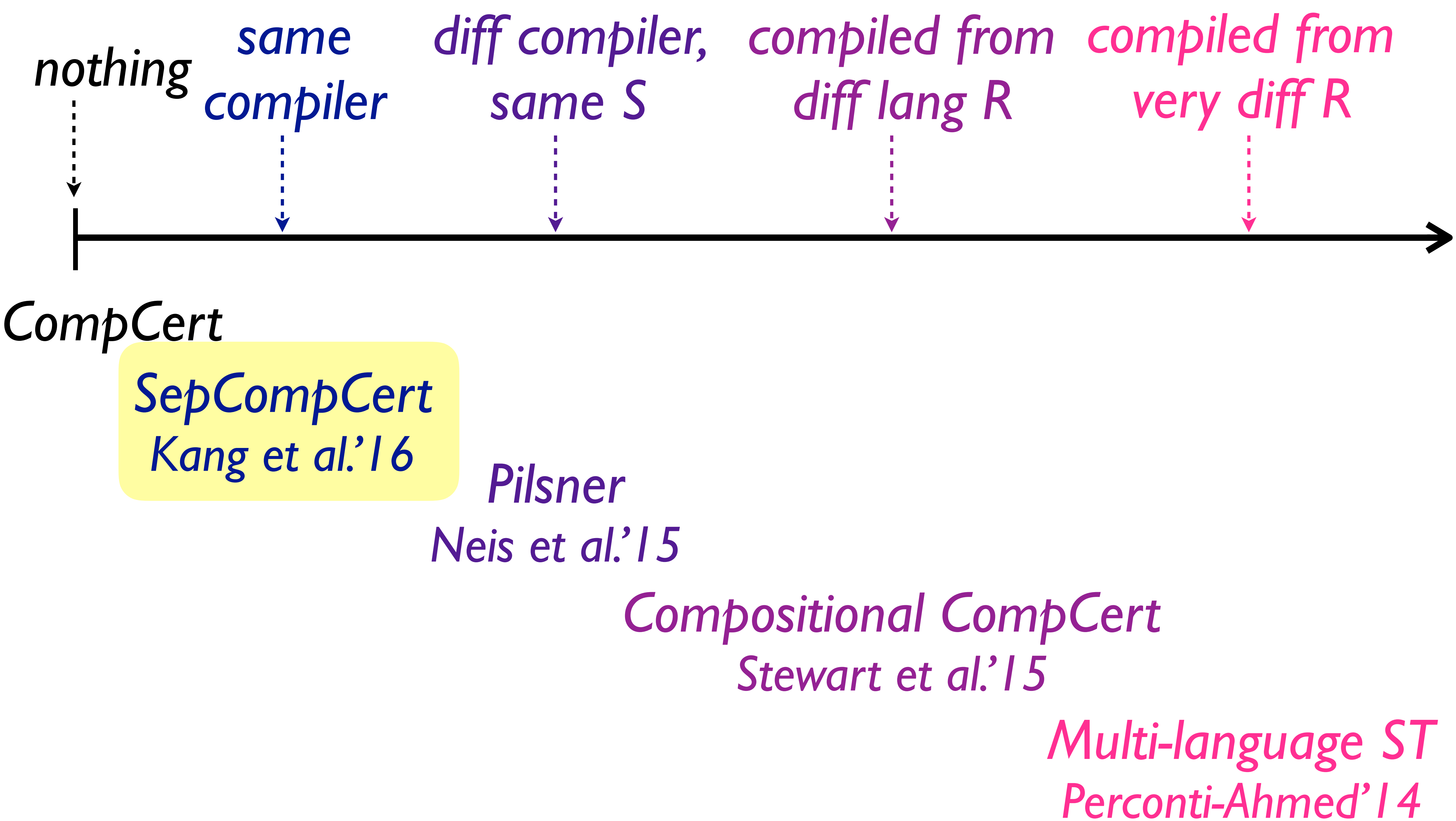
- Survey of "compositional" compiler correctness results
  - how to express $e_S \approx \mathbf{e_T}$

- How does the choice affect:
  - what we can **link** with *(horizontal compositionality)*
  - how we check if some $\mathbf{e'_t}$ is okay to link with
  - effort required to prove *transitivity* for **multi-pass** compilers *(vertical compositionality)*
  - effort required to have confidence in theorem statement

What we can link with



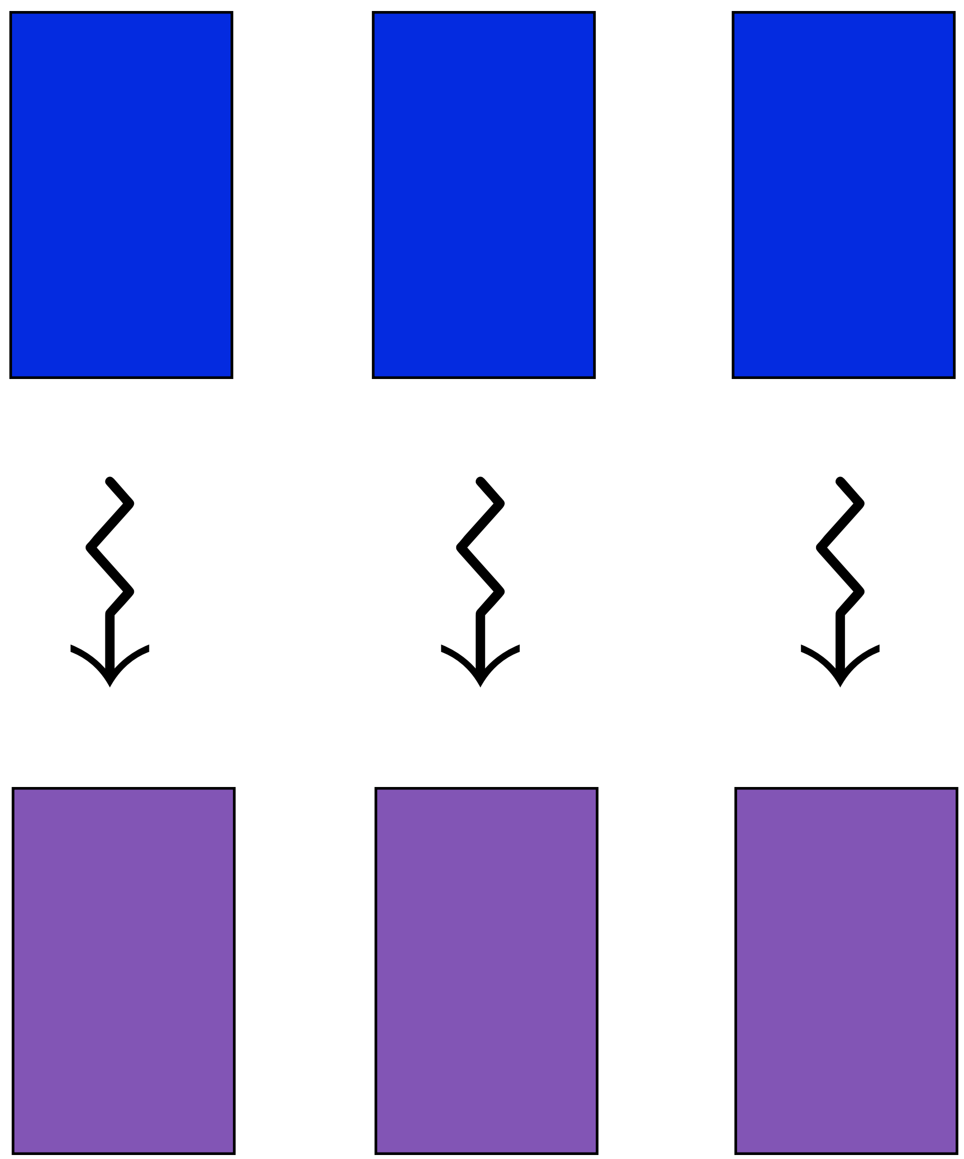
What we can link with



Approach: Separate Compilation

SepCompCert

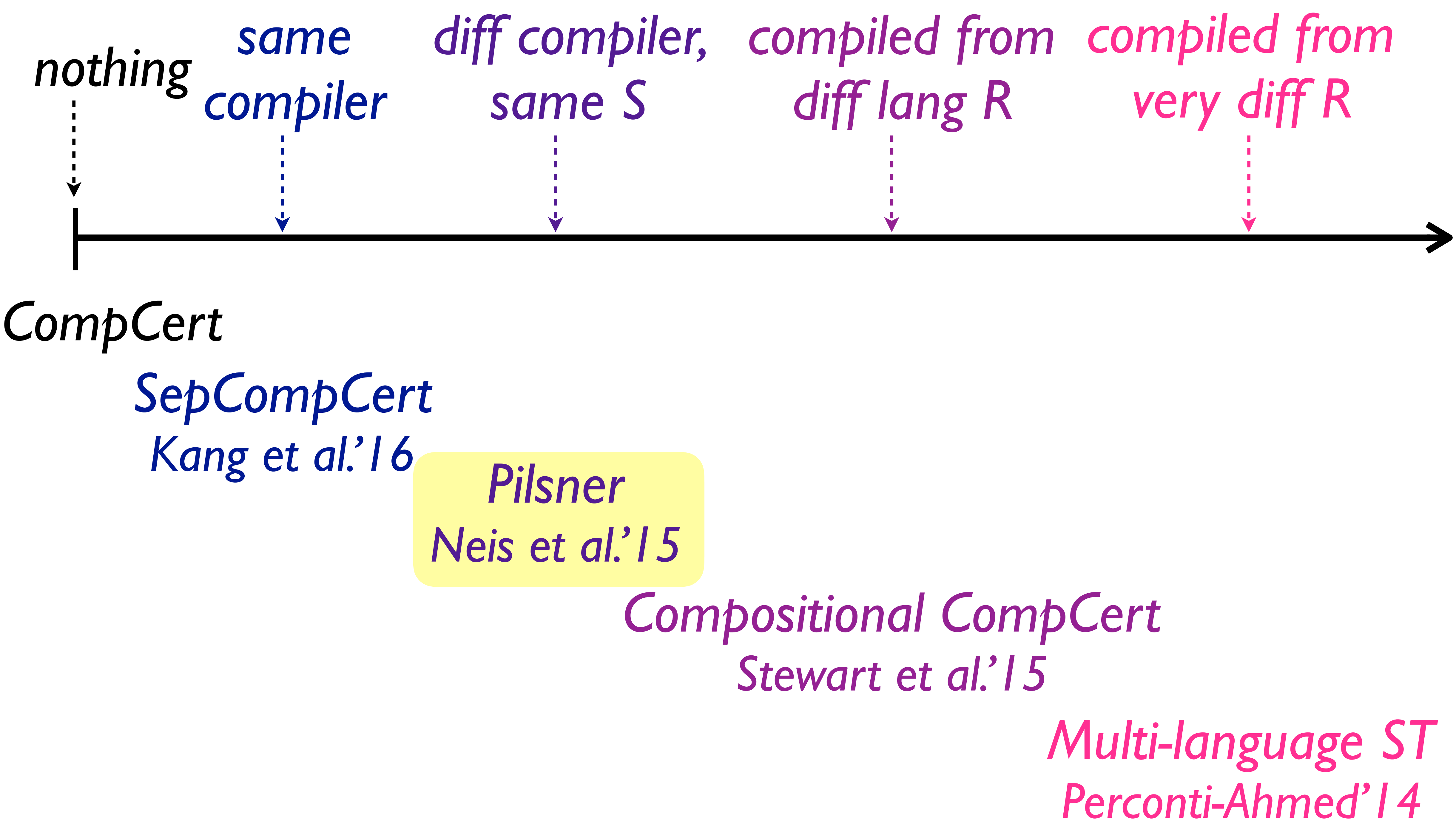
[Kang et al. '16]



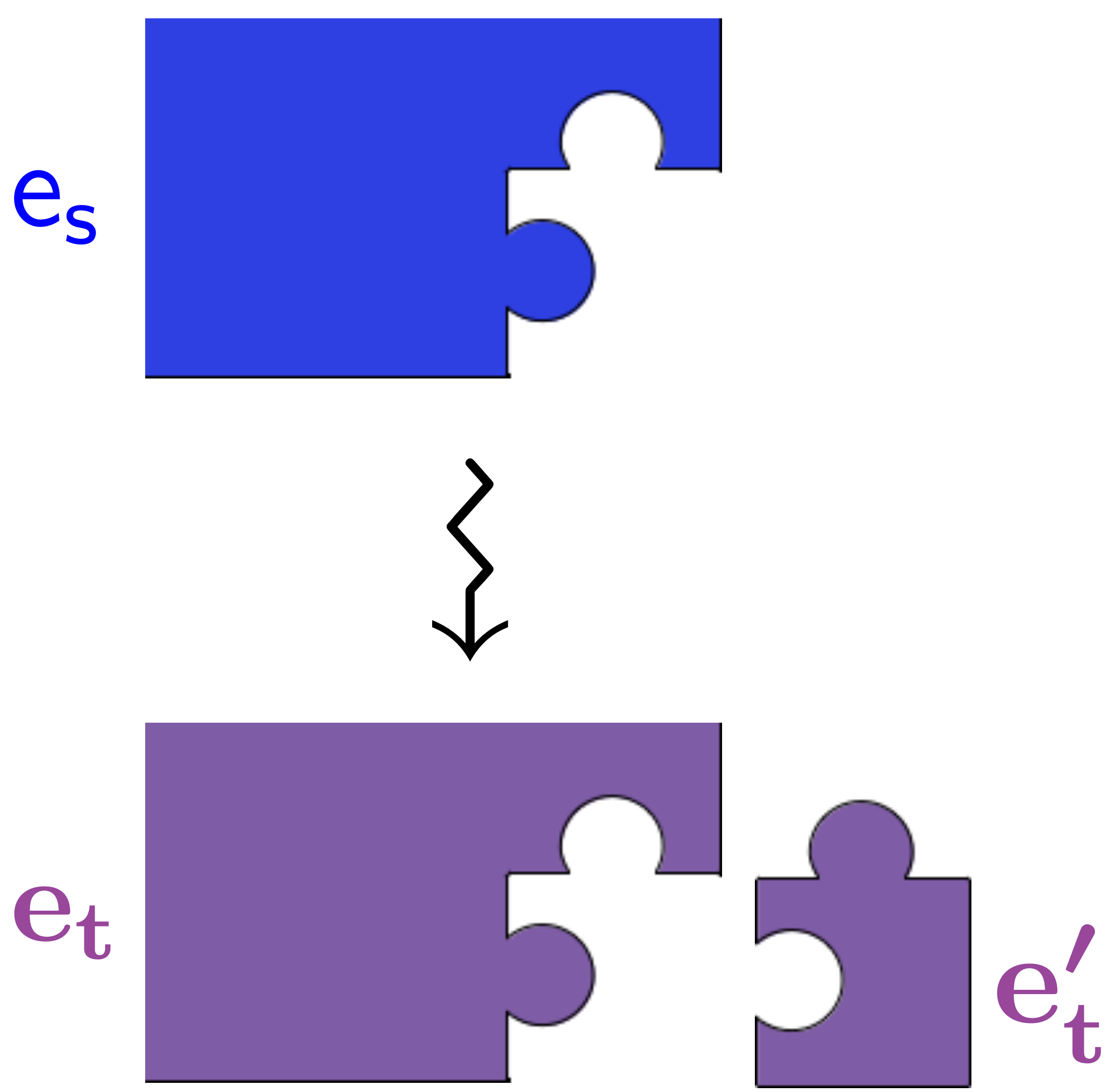
Level A correctness:
exactly same compiler

Level B correctness:
can omit some intra-language (RTL) optimizations

What we can link with



Approach: Cross-Language Relations



Cross-language relation

$$e_S \approx e_T$$

Compiling ML-like langs:

Logical relations

-

-

No transitivity!

Parametric inter-language simulations (PILS)

-

Prove transitivity, but requires effort!

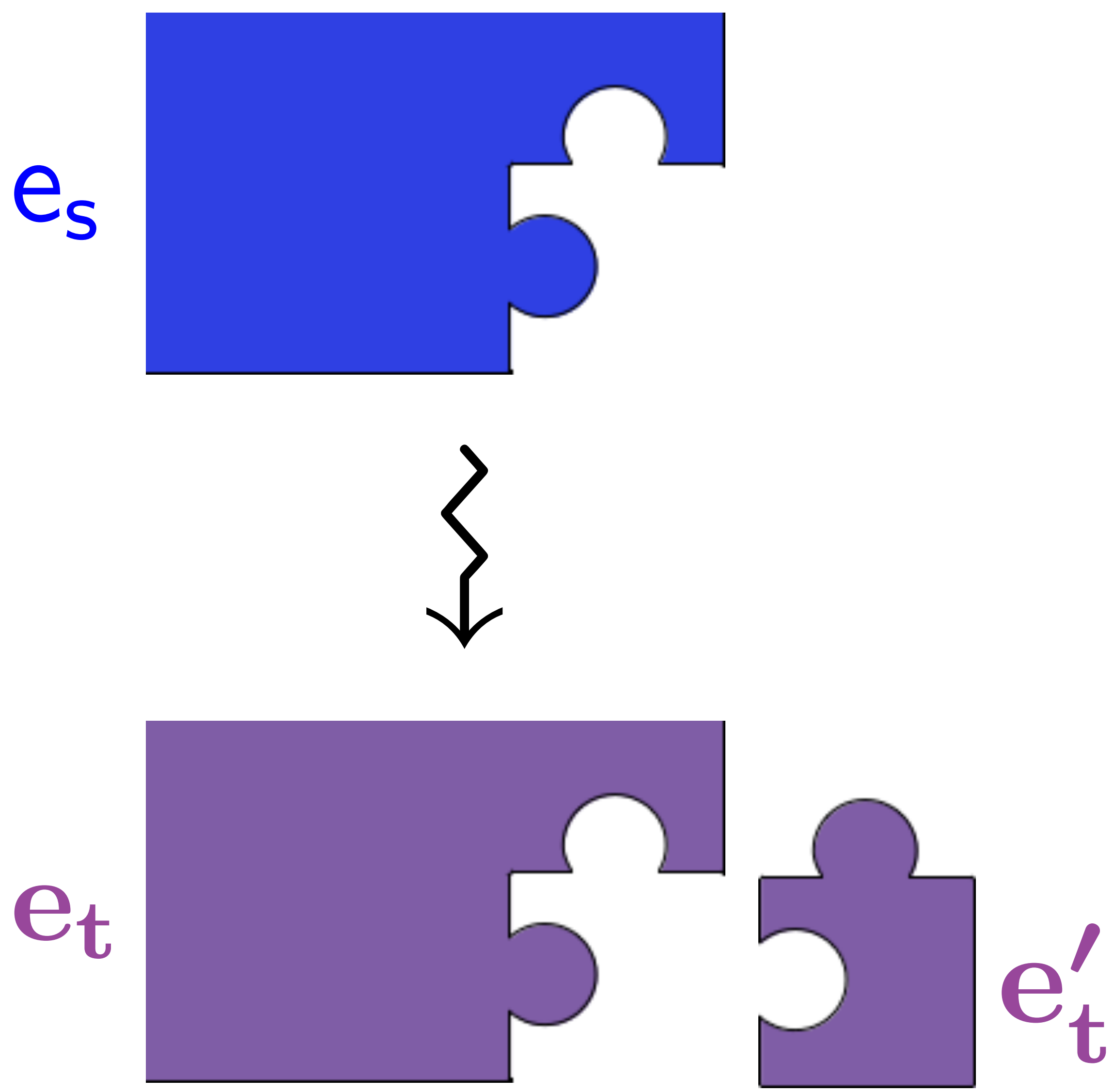
Cross-Language Relation (Pilsner)

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

cross-language relation

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

Cross-Language Relation (Pilsner)

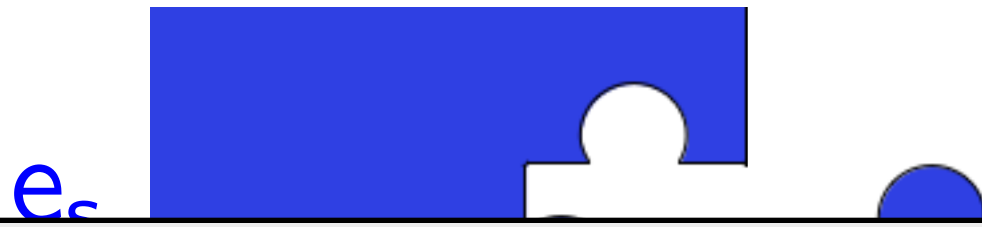


Have $x : \tau' \vdash e_s \simeq \mathbf{e_t} : \tau$

Does the compiler correctness theorem permit linking with $\mathbf{e'_t}$ *?*

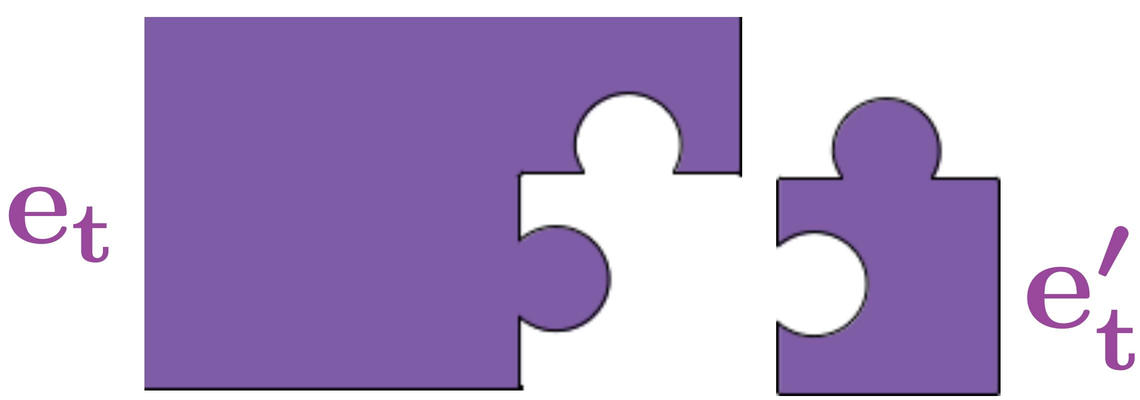
Cross-Language Relation (Pilsner)

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



cross-language relation

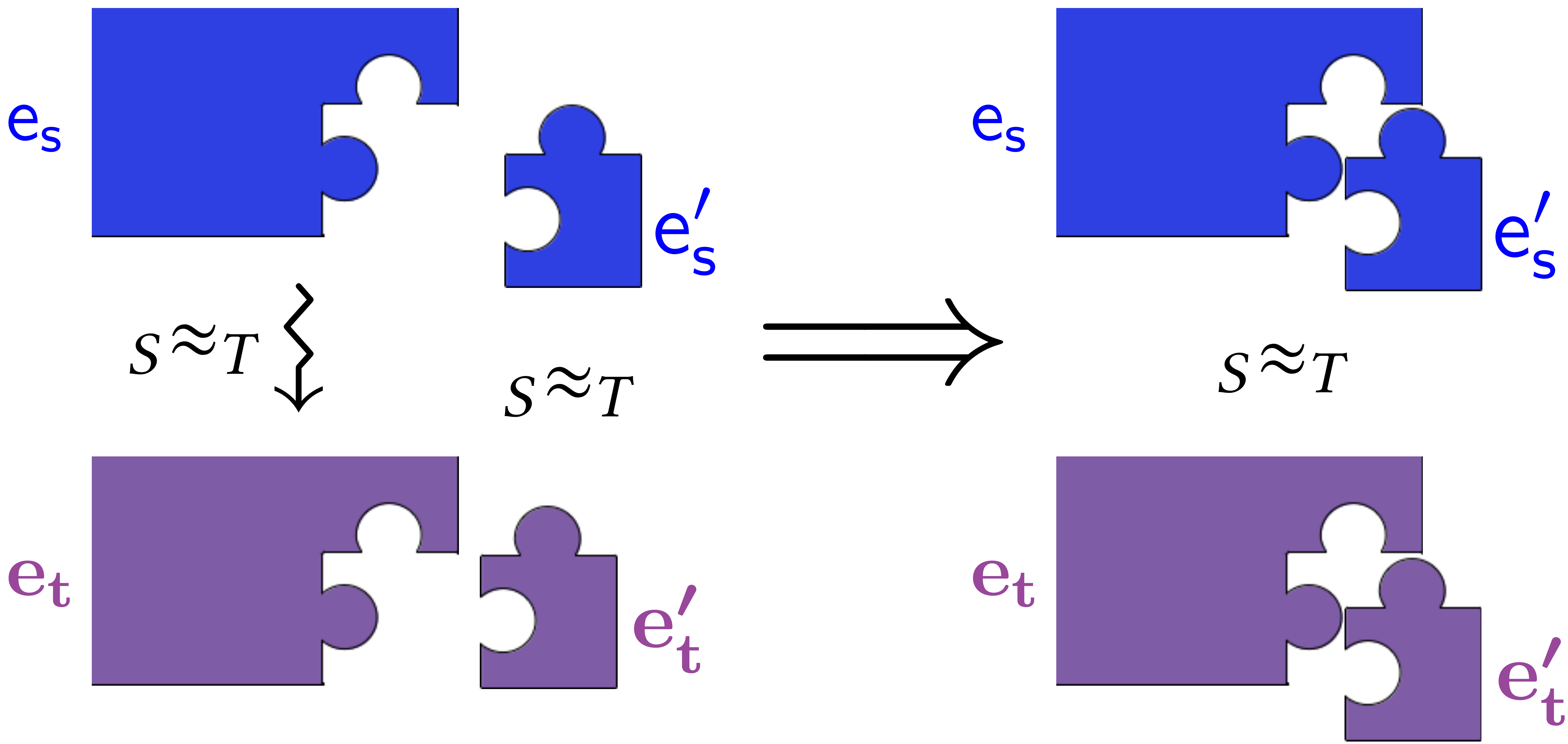
$$\forall e_s', e_t'.\ \vdash e_s' \simeq e_t' : \tau' \implies \vdash e_s[e_s'/x] \simeq e_t[e_t'/x] : \tau$$



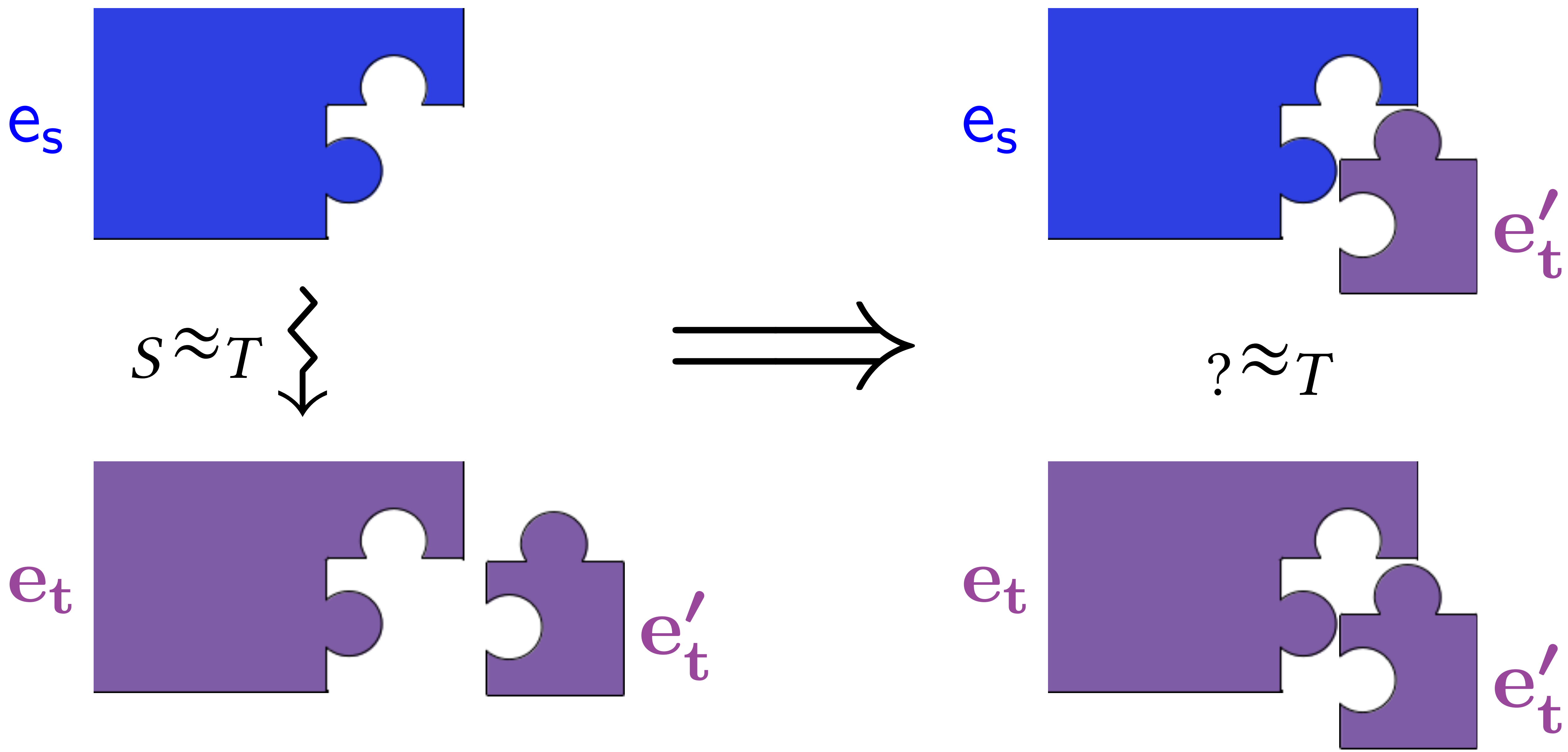
- Need to come up with e_s' -- not feasible in practice!
- Cannot link with e_t' whose behavior cannot be expressed in source.

Horizontal Compositionality

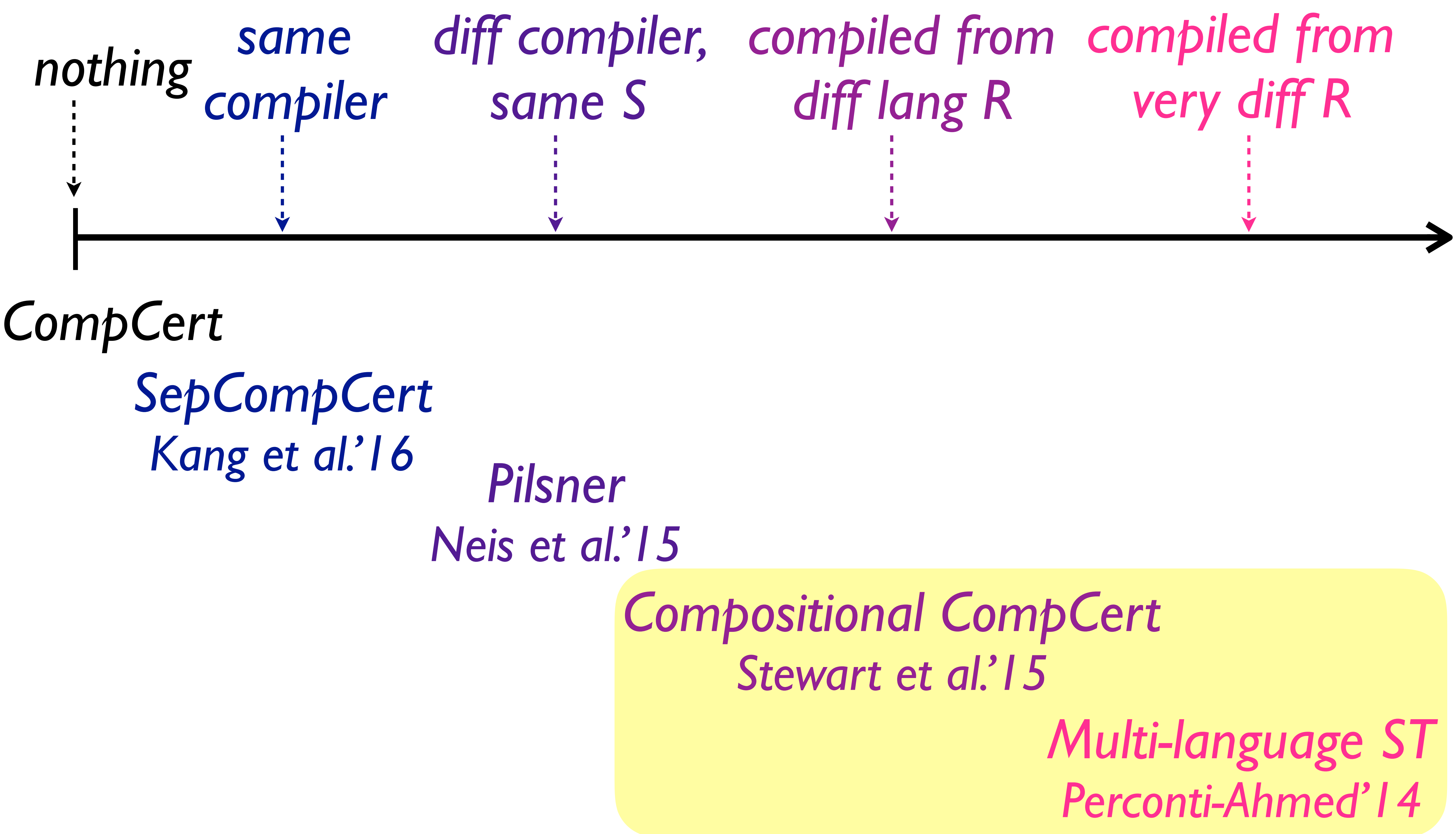
Linking



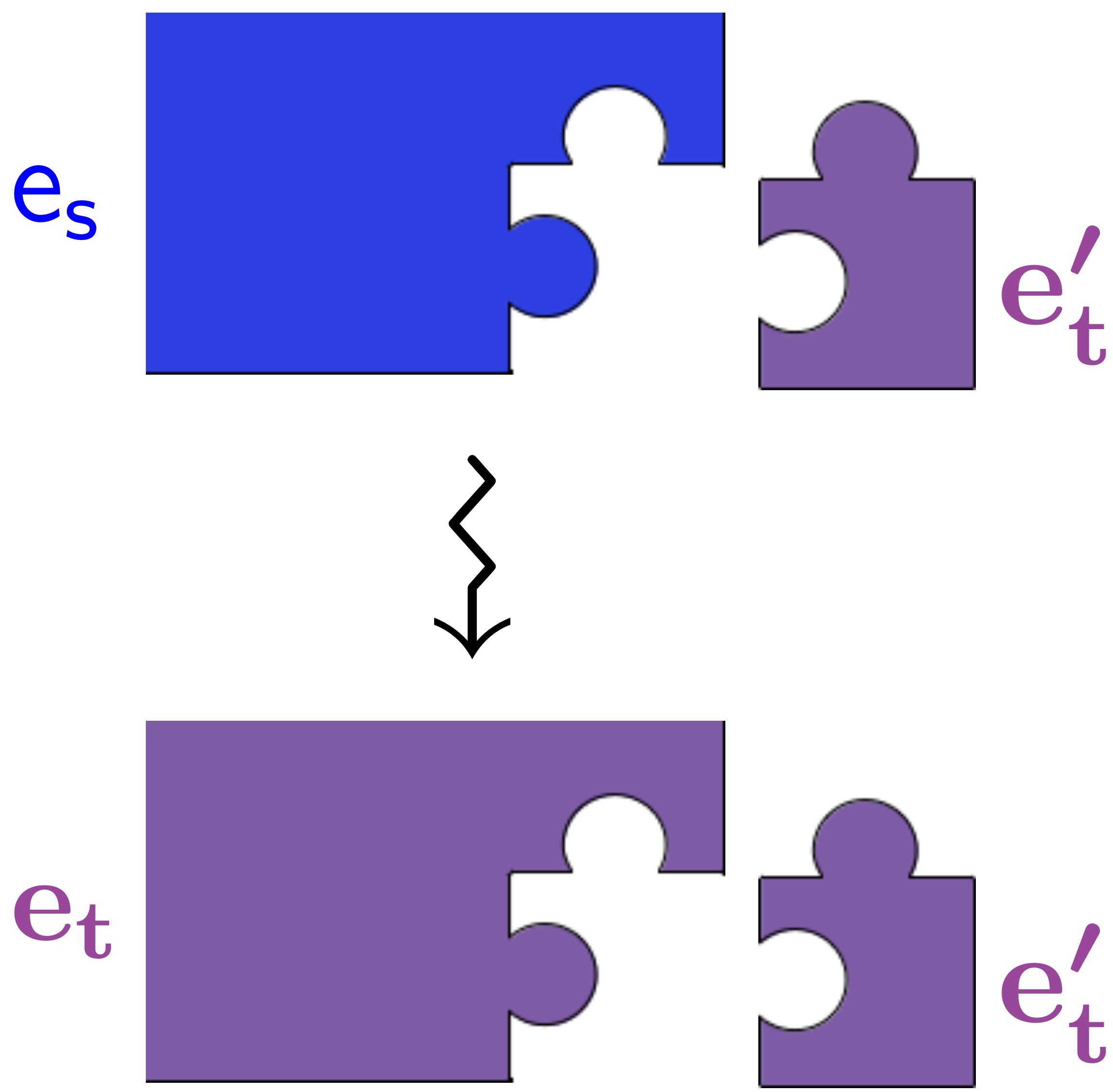
Linking



What we can link with



Correct Compilation of Components?



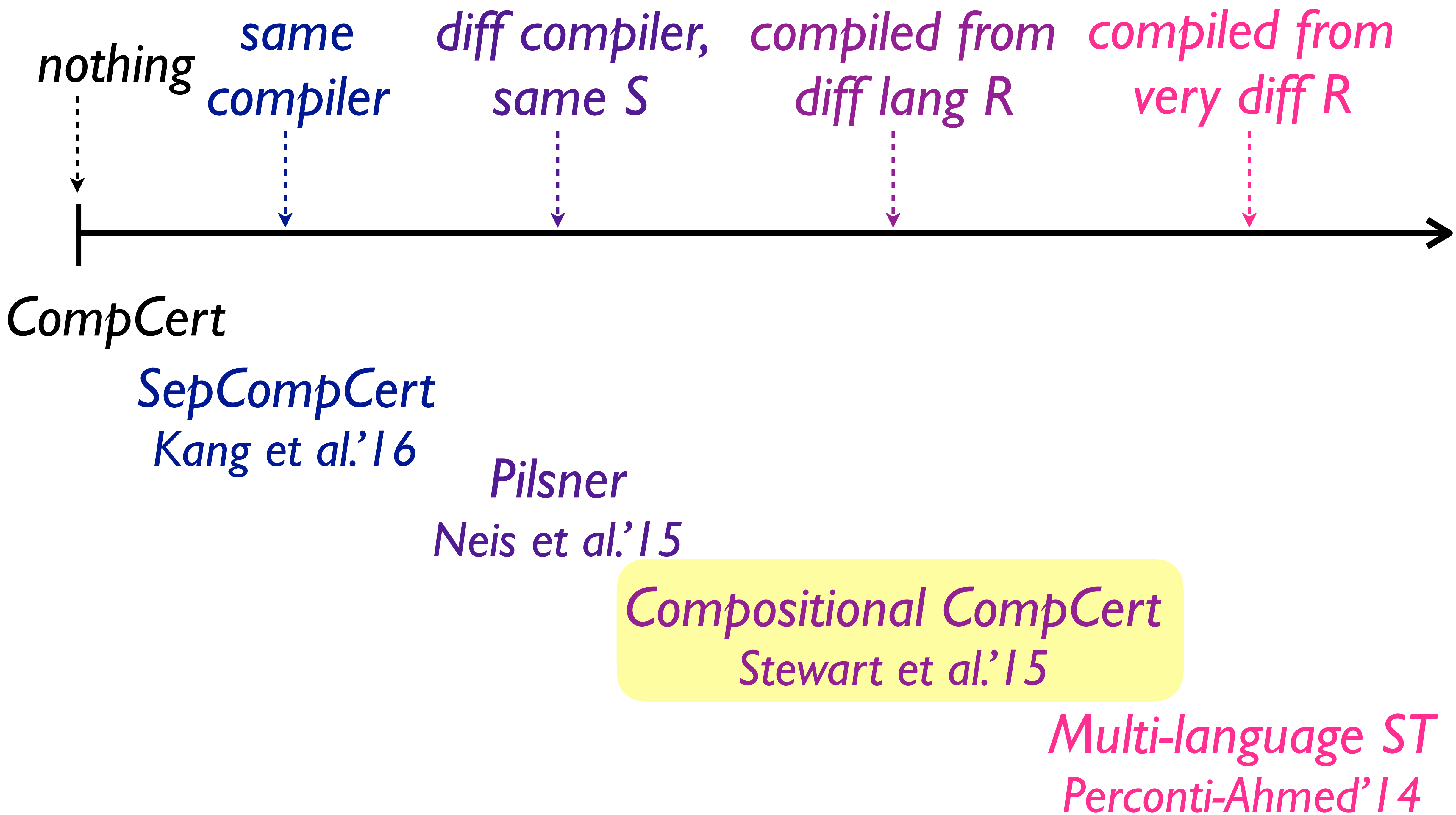
$e_S \approx e_T$

expressed how?

Need a semantics of source-target interoperability:

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

What we can link with

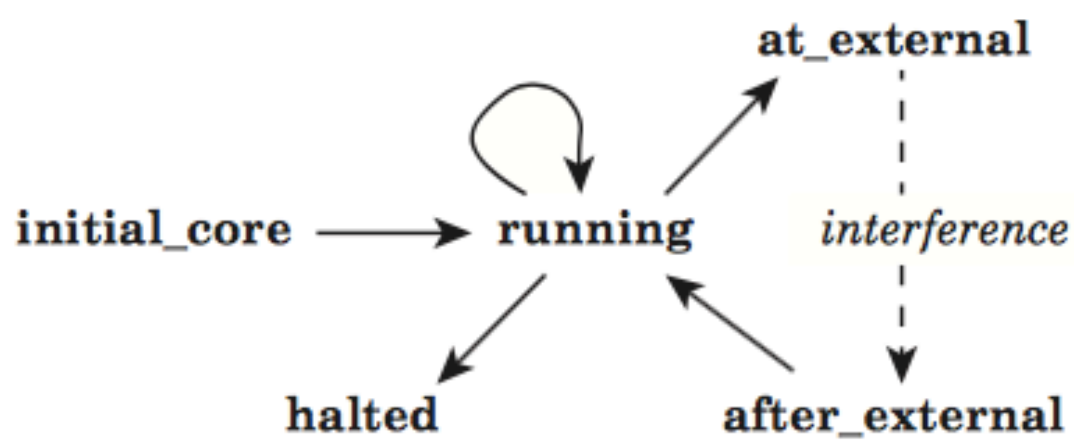


Approach: Interaction Semantics

Compositional CompCert

[Stewart et al. '15]

- Language-independent linking



$$\text{Semantics}\,(G\ C\ M : \text{Type}) : \text{Type} \triangleq
\begin{cases}
\text{initial_core} & : \quad G \to \mathcal{V} \to \text{list}\ \mathcal{V} \to \text{option}\ C \\
\text{at_external} & : \quad C \to \text{option}\ (\mathcal{F} \times \text{list}\ \mathcal{V}) \\
\text{after_external} & : \quad \text{option}\ \mathcal{V} \to C \to \text{option}\ C \\
\text{halted} & : \quad C \to \text{option}\ \mathcal{V} \\
\text{corestep} & : \quad G \to C \to M \to C \to M \to \text{Prop}
\end{cases}$$

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.

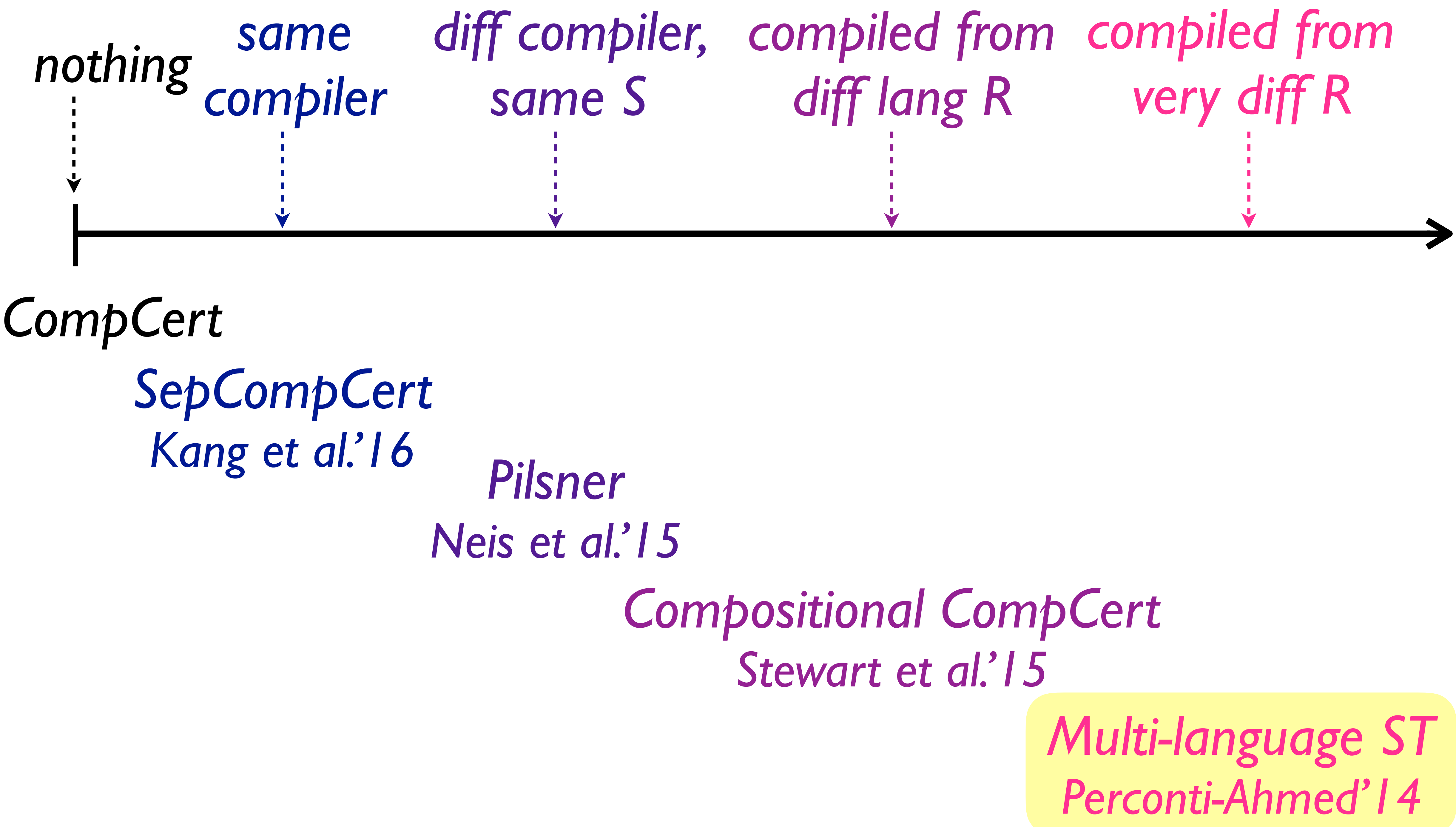
Approach: Interaction Semantics

Compositional CompCert
[Stewart et al. '15]

- Language-independent linking

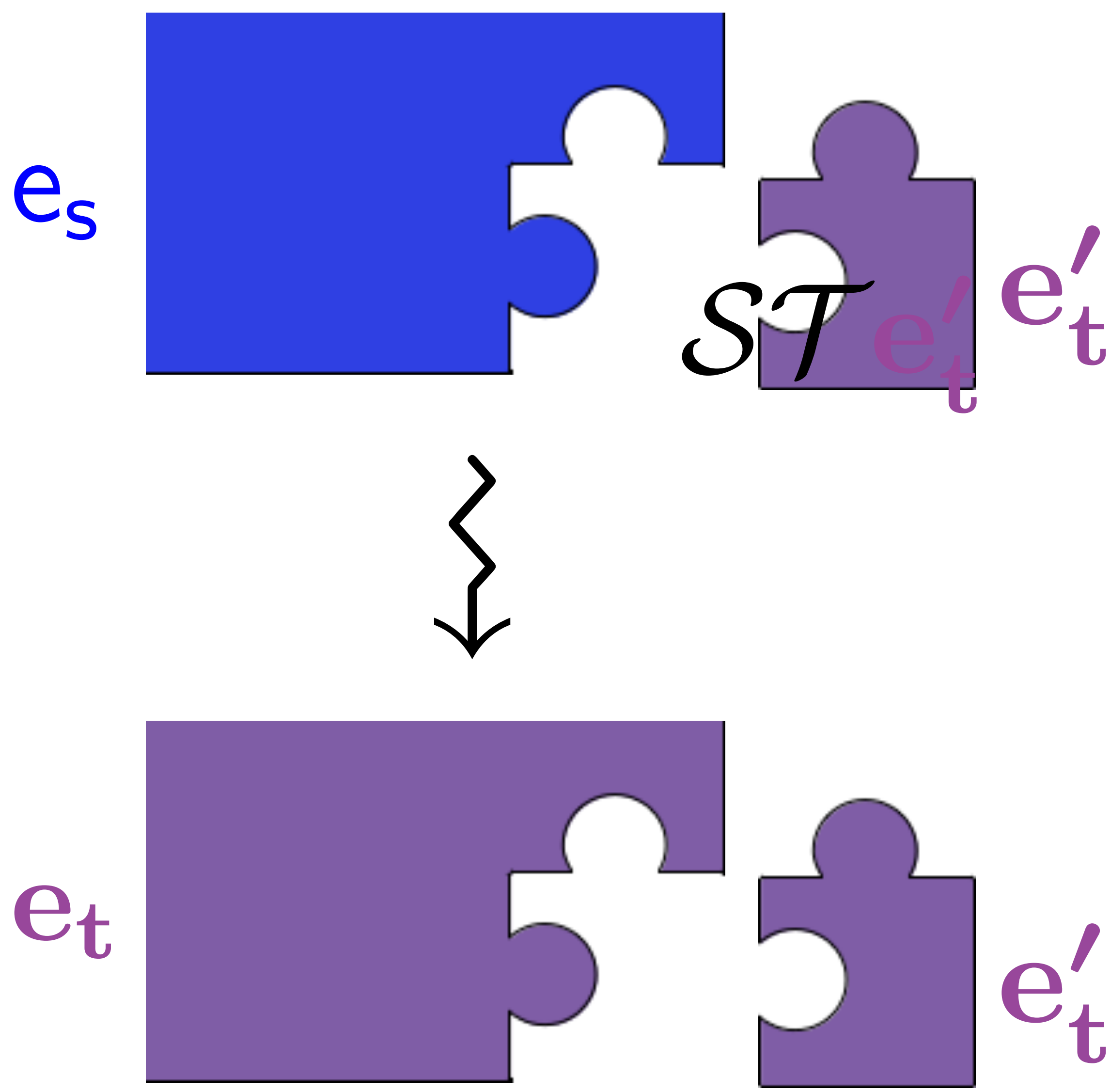
- Structured simulation: support rely-guarantee relationship between the different languages while retaining vertical compositionality

What we can link with



Approach: Source-Target Multi-lang.

[Perconti-Ahmed'14]



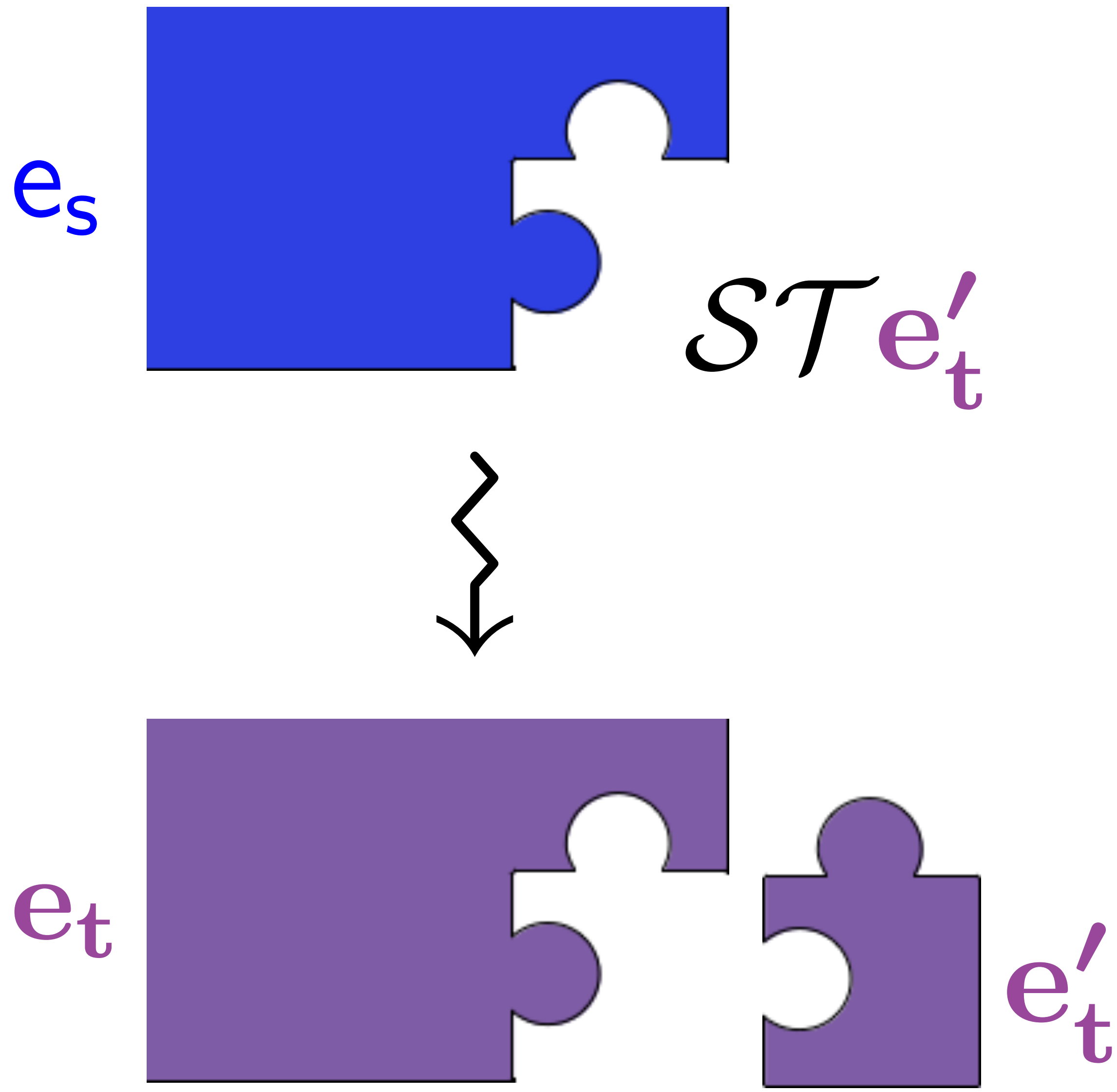
Specify semantics of source-target interoperability:

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

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

Approach: Source-Target Multi-lang.

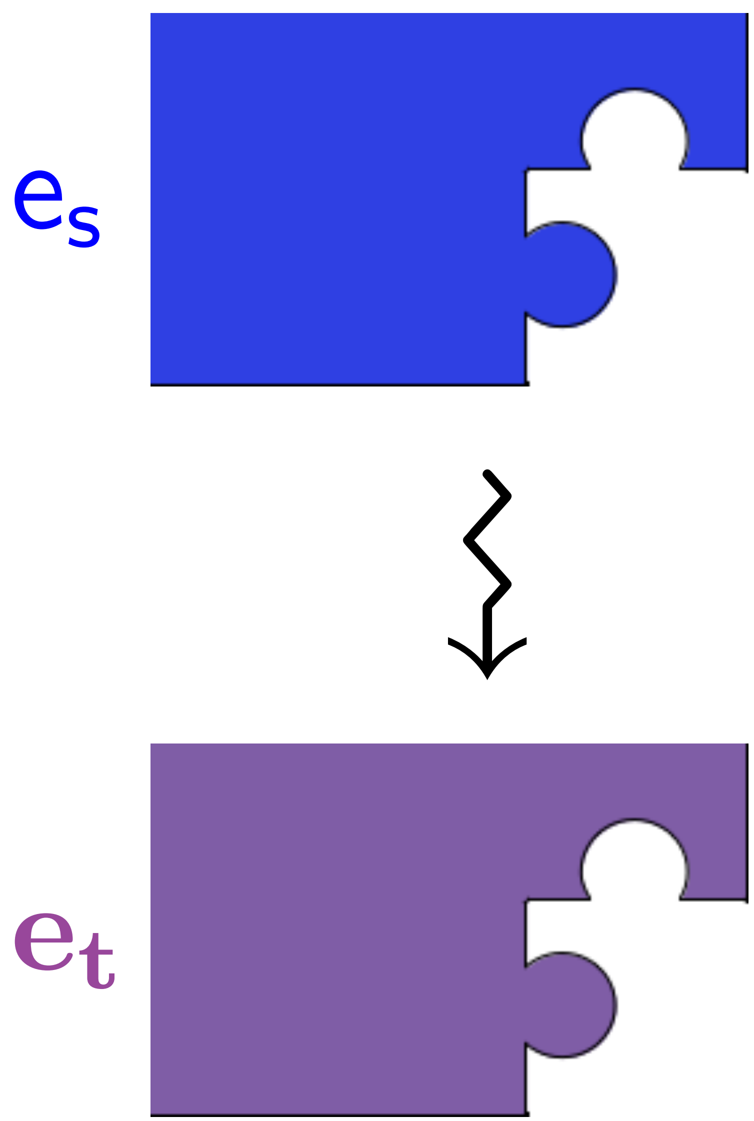
[Perconti-Ahmed'14]



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

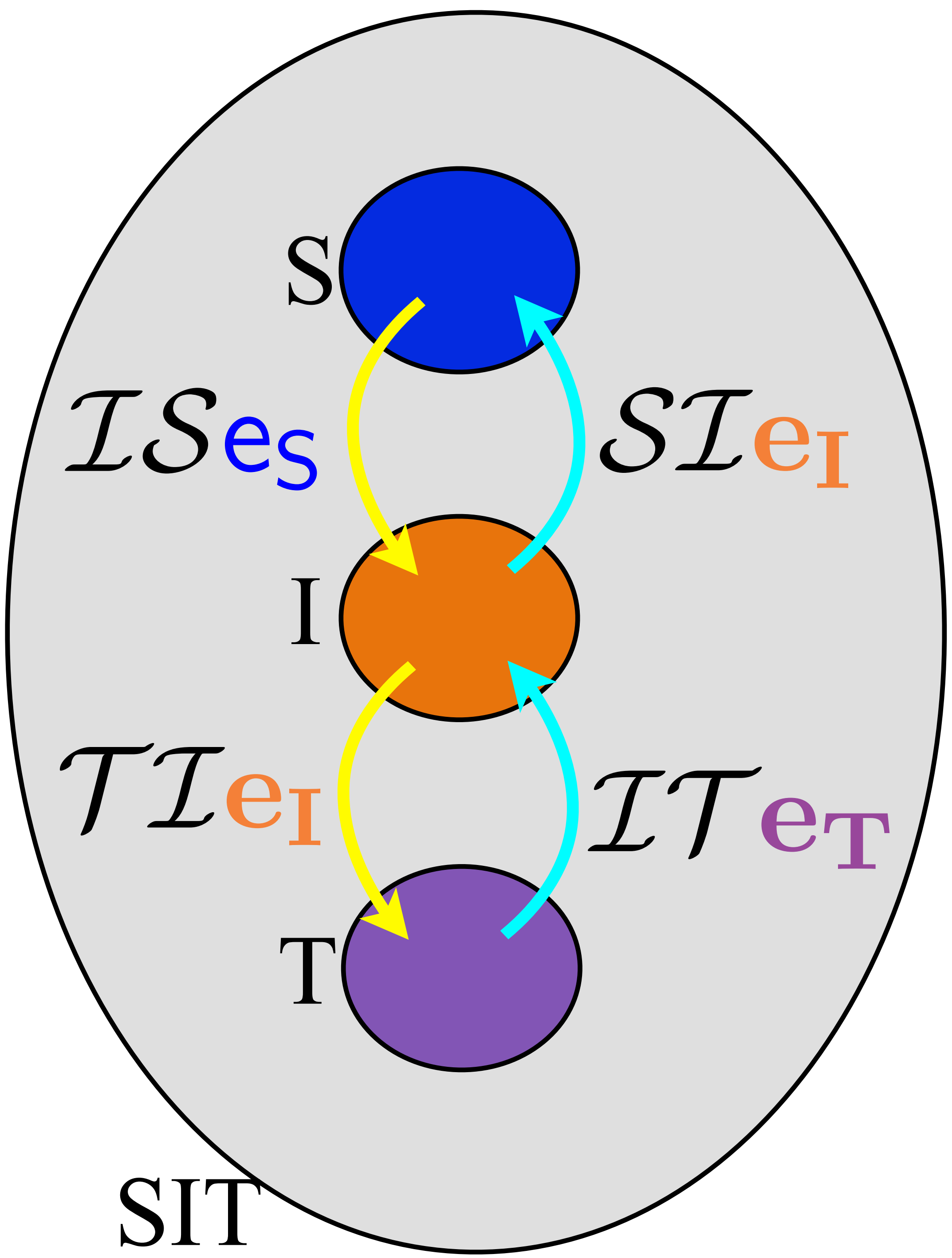
Approach: Source-Target Multi-lang.

[Perconti-Ahmed'14]

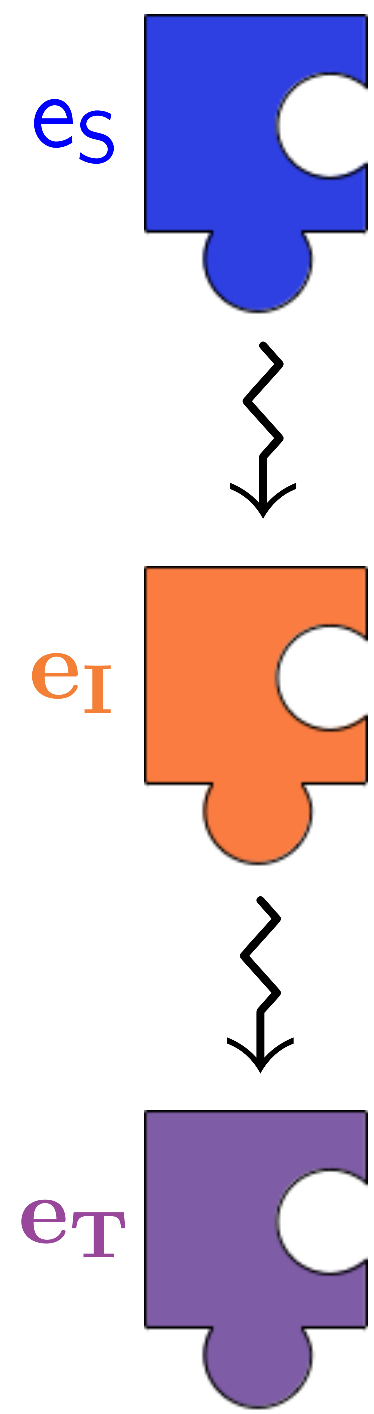


$$\boxed{\begin{aligned} & e_S \approx \mathbf{e_T} \stackrel{\text{def}}{=} \\ & \qquad e_S \approx^{ctx} \mathcal{ST}\,\mathbf{e_T} \end{aligned}}$$

Multi-Language Semantics Approach



Compiler Correctness

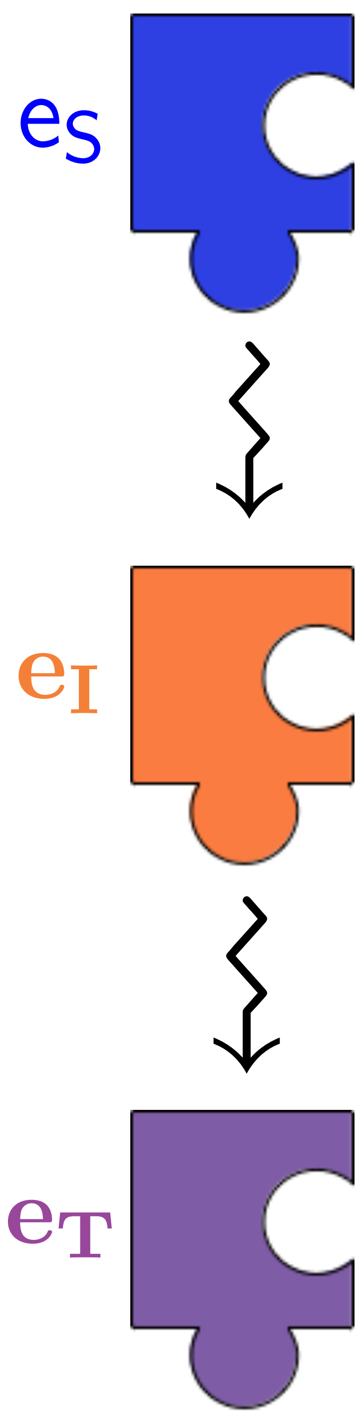


$e_S \approx^{ctx} \mathcal{SI}\,e_I$

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

Multi-Lang. Approach: Multi-pass ✓

Compiler Correctness

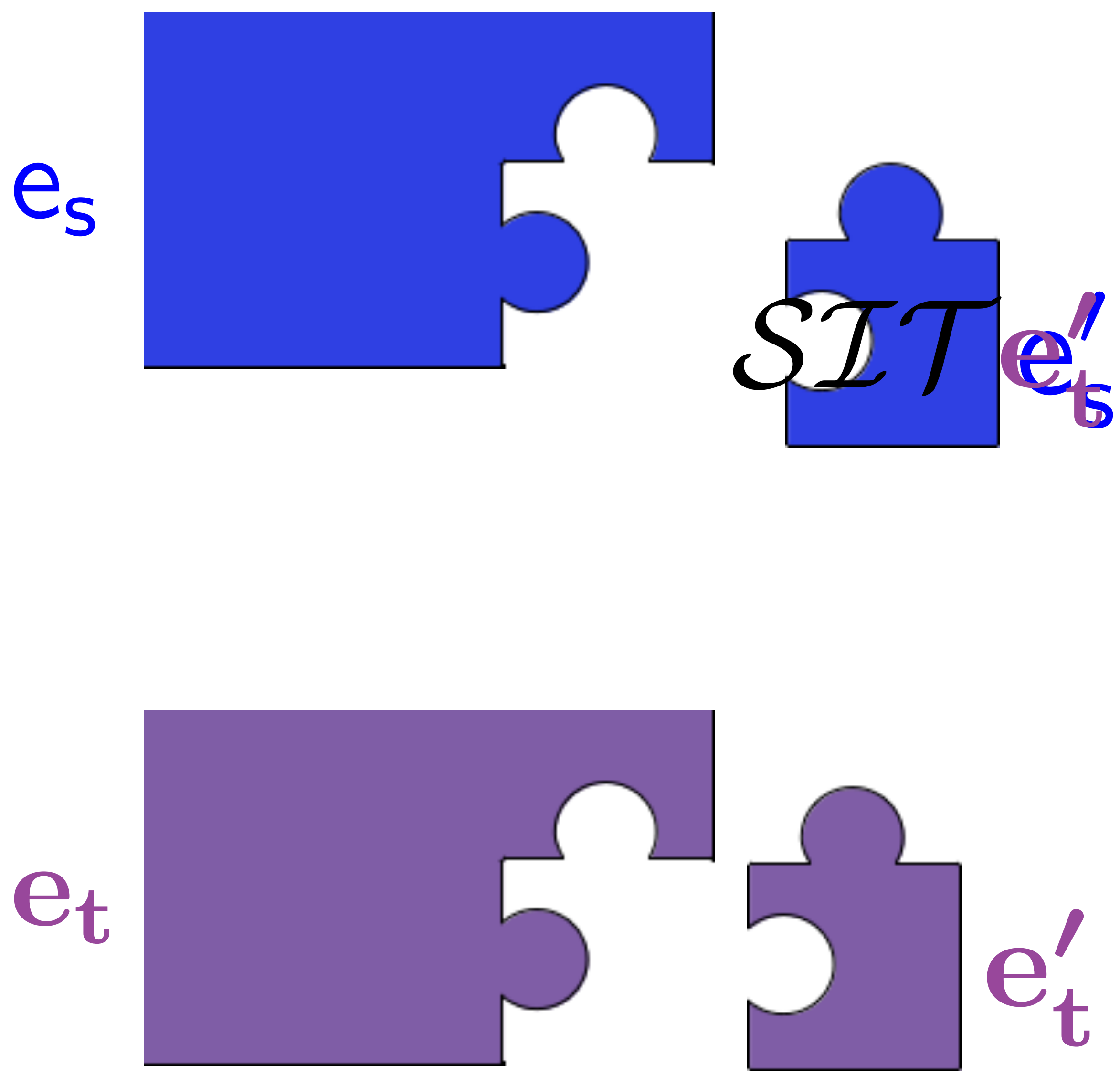


$$e_S \approx^{ctx} \mathcal{SI}\,e_I$$

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

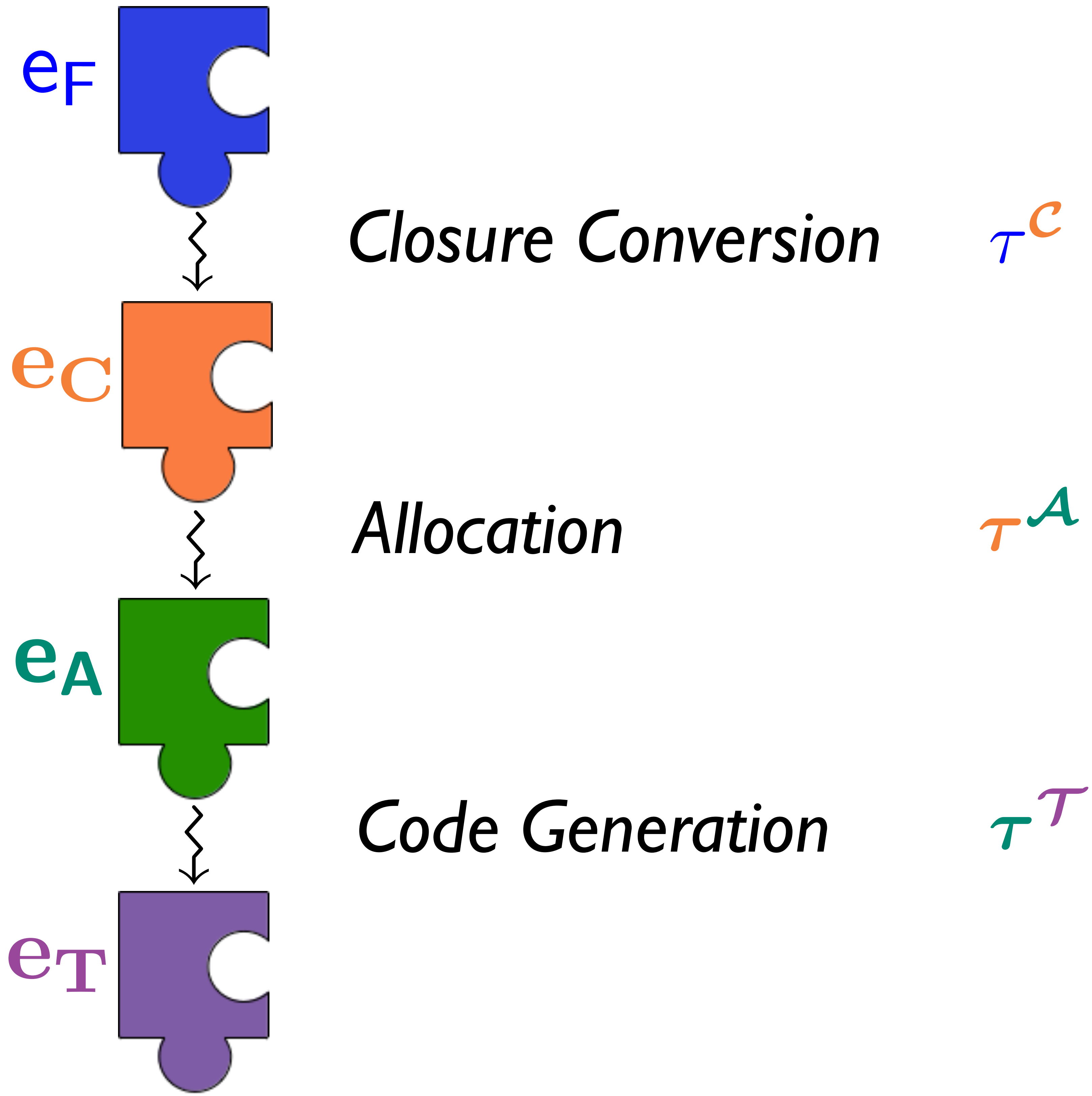
$$e_S \approx^{ctx} \mathcal{SIT}\,e_T$$

Multi-Lang. Approach: Linking ✔

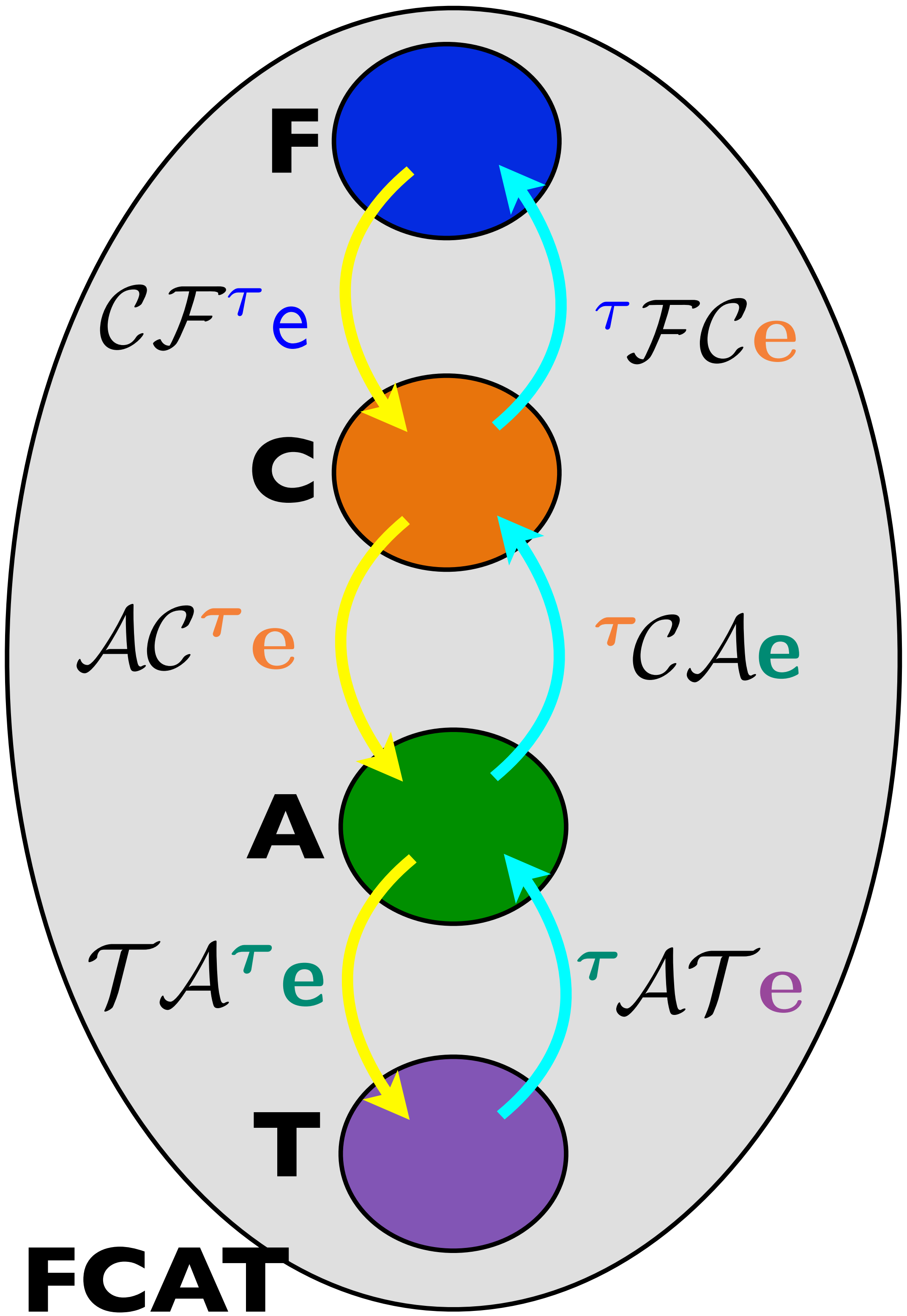


$$\mathcal{TIS}(\mathbf{e_s}\ (\mathcal{SIT}\,\mathbf{e'_t}))$$
$$\approx^{ctx} \mathbf{e_t}\ \mathbf{e'_t}$$

Compiler Correctness: F to TAL



Combined language **FCAT**

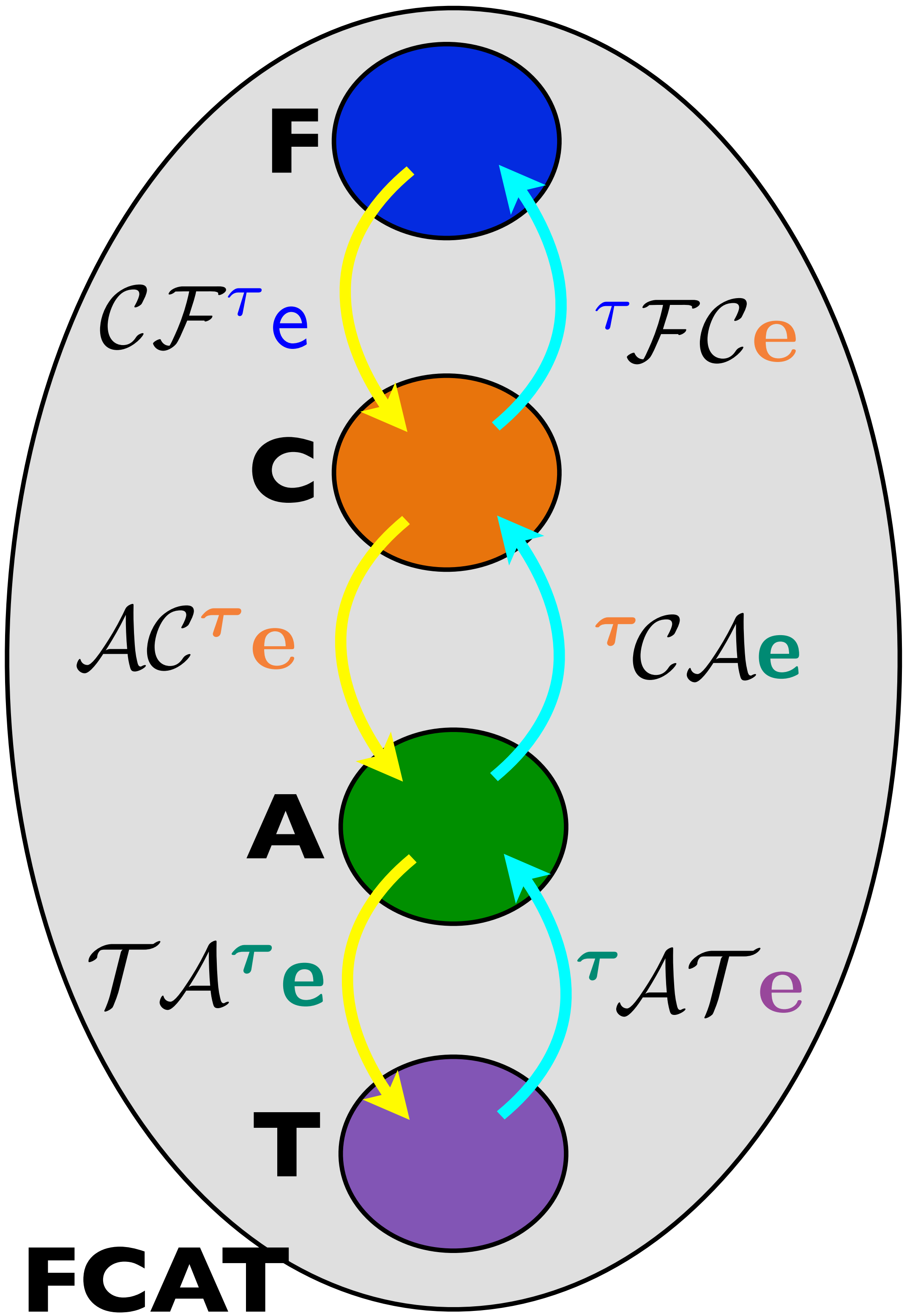


[Perconti-Ahmed'14]
[Patterson et al.'17]

- Boundaries mediate between

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

Challenges

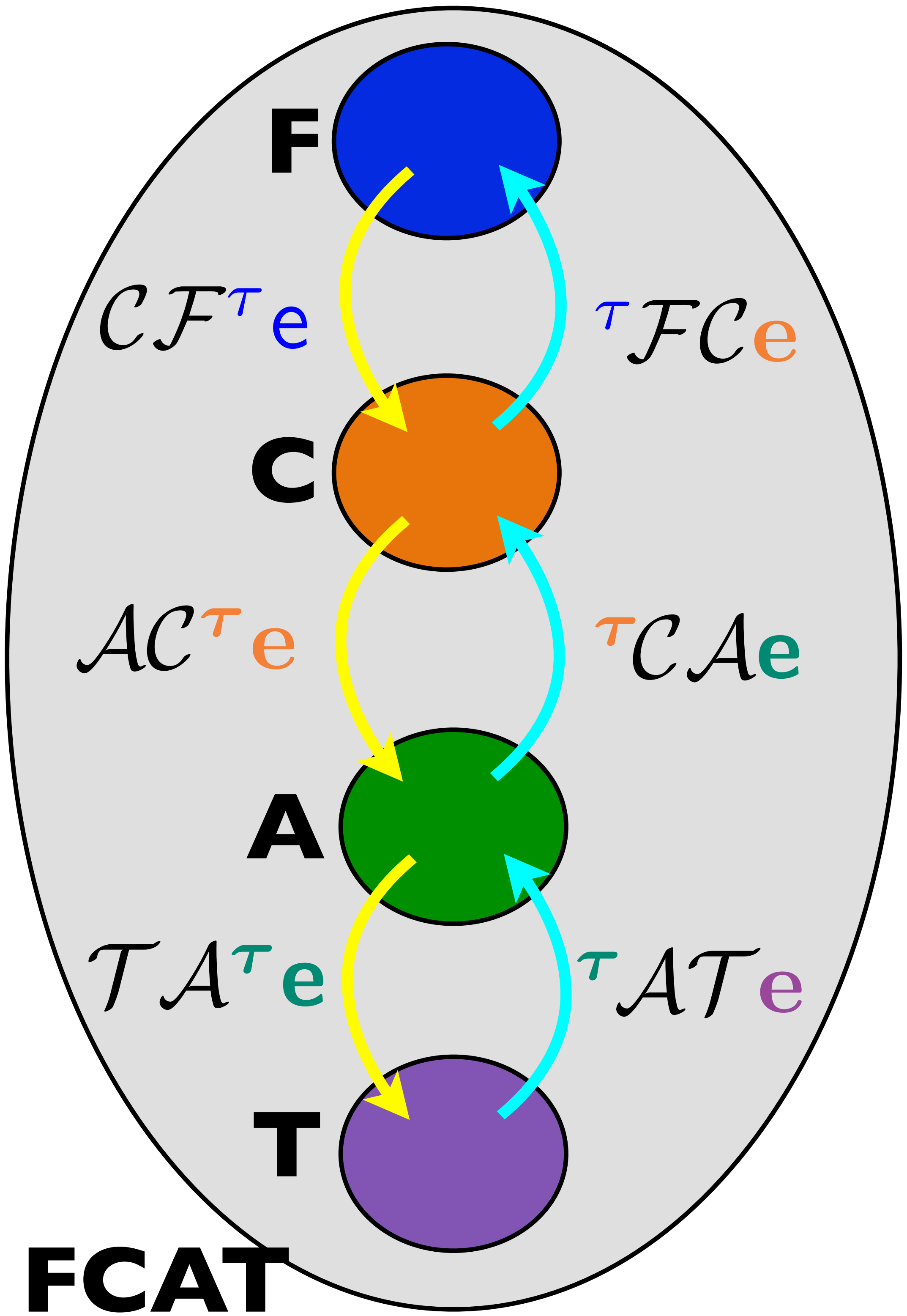


F+C: Interoperability semantics with type abstraction in both languages

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

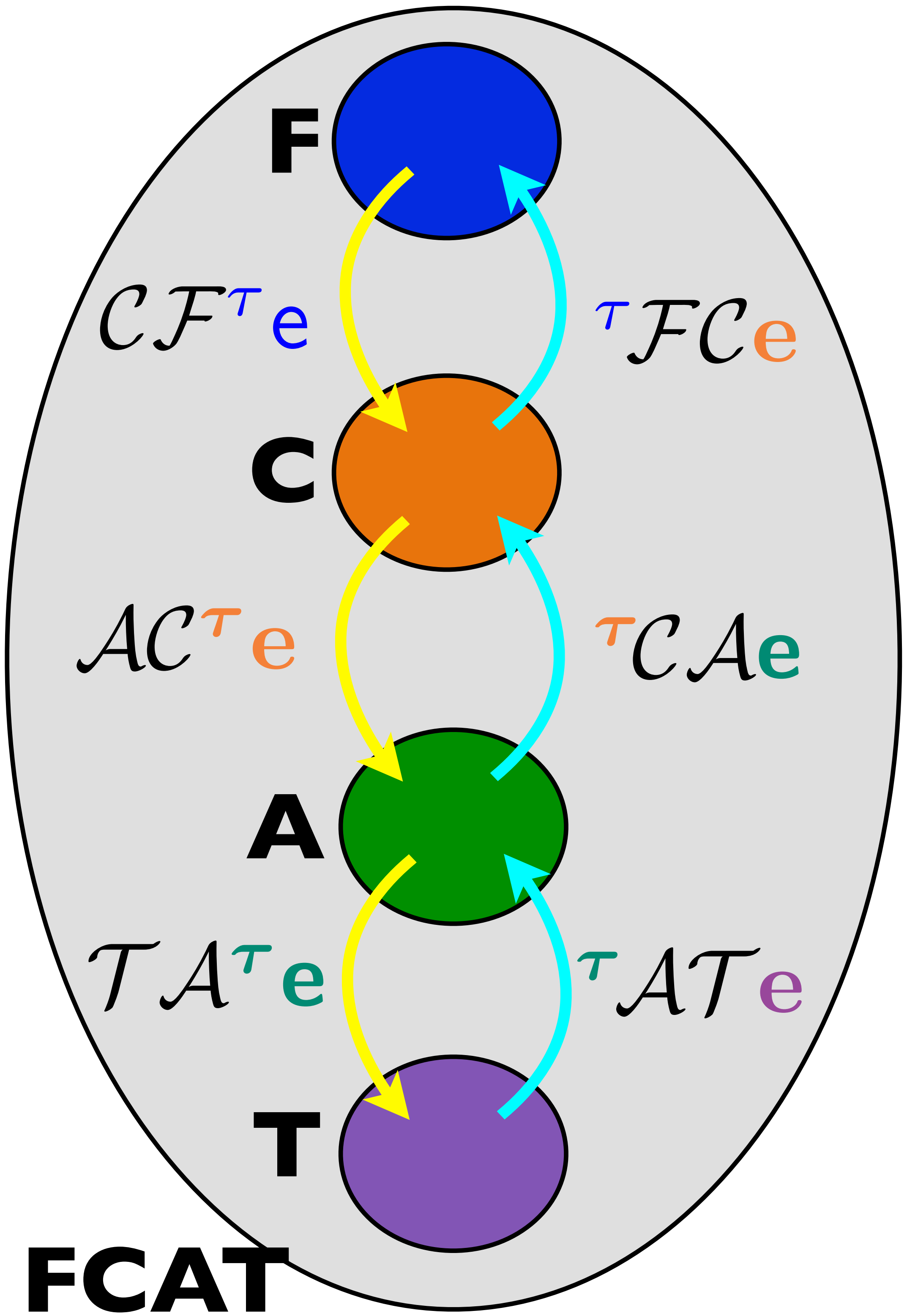


F+C: Interoperability semantics with type abstraction in both languages

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

A+T: What is e? What is 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 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. PLDI'17]*

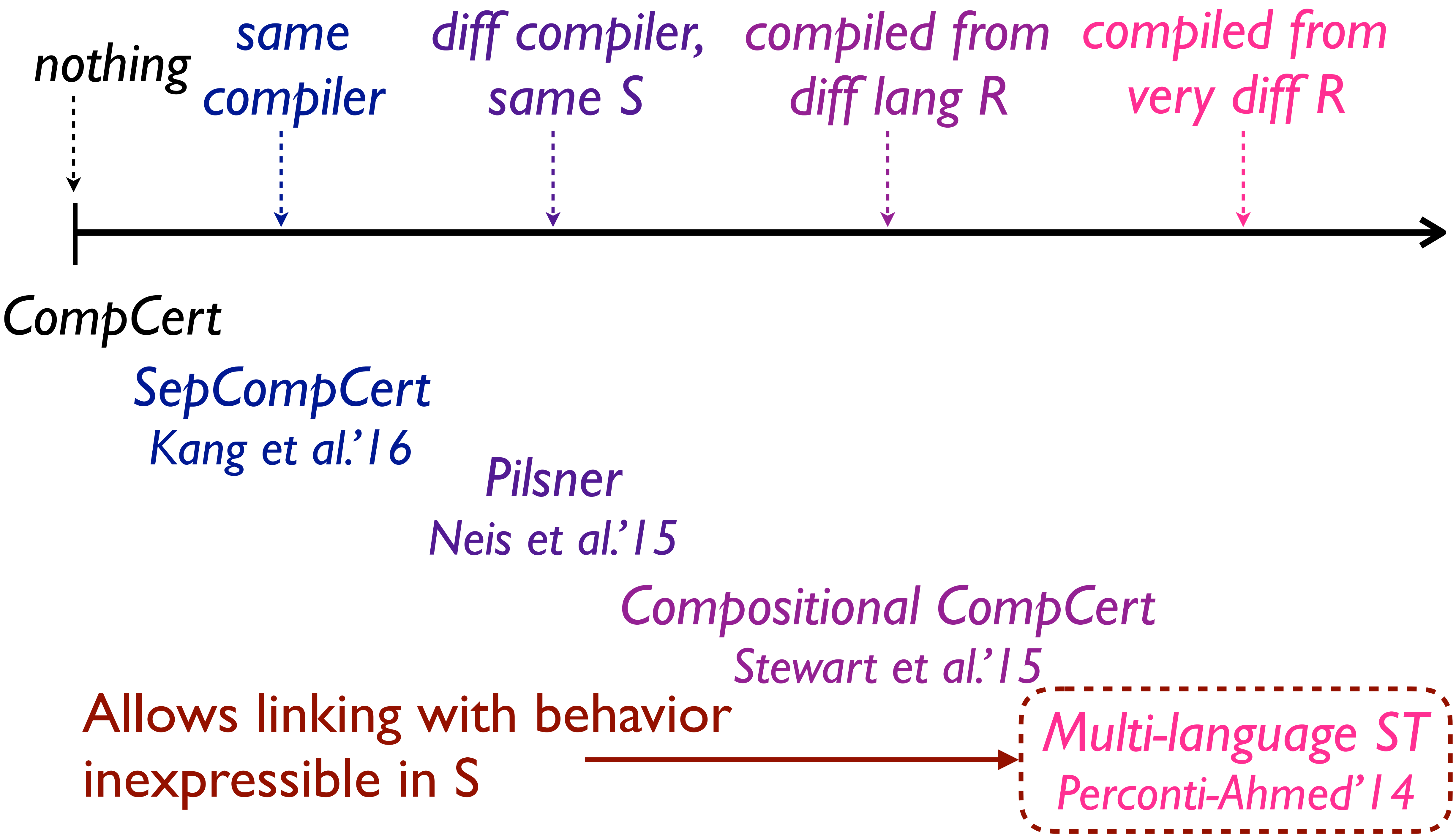
CompCompCert vs. Multi-language

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

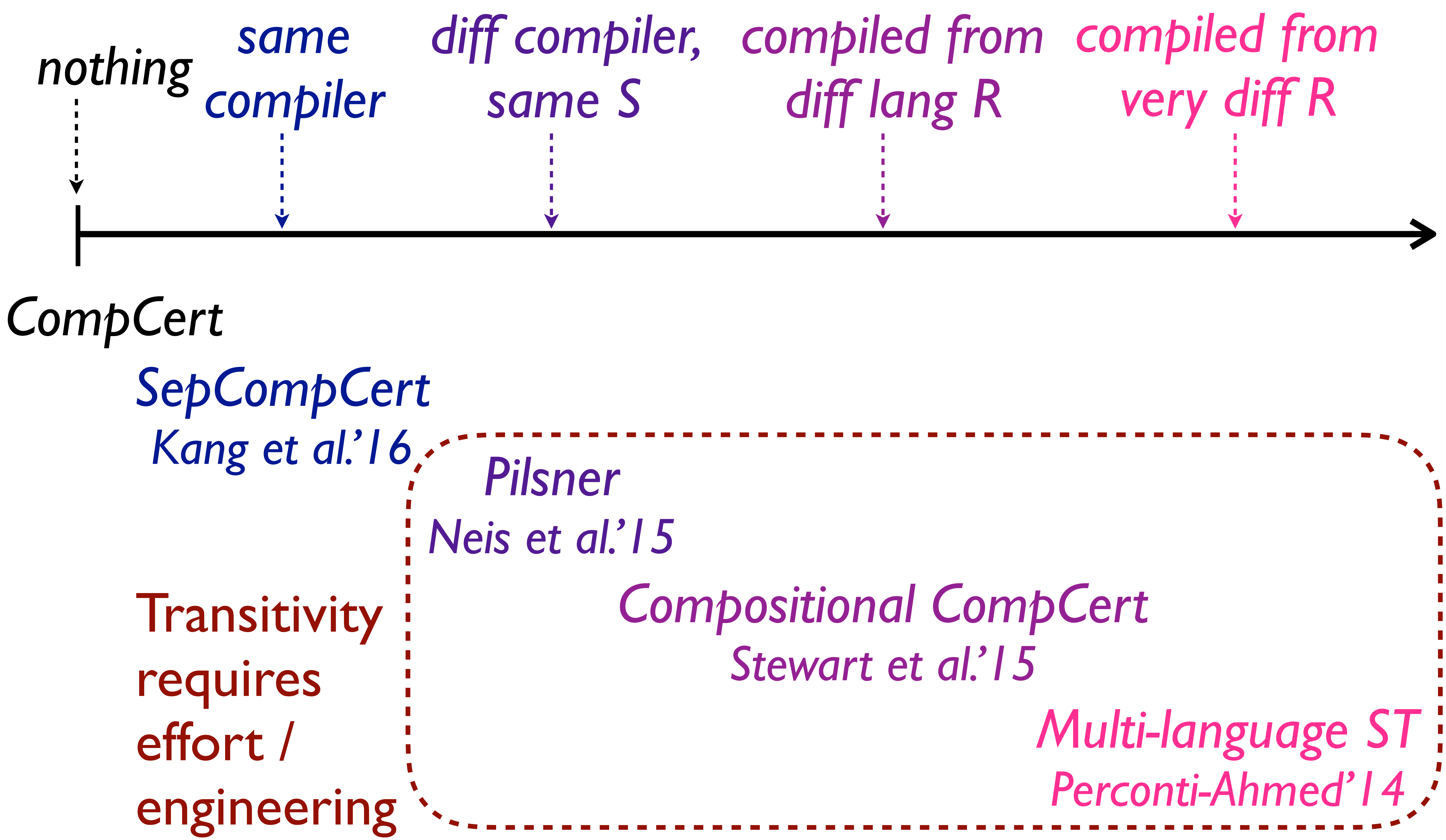
Requires uniform memory model across compiler IRs?

- yes	- no

What we can link with

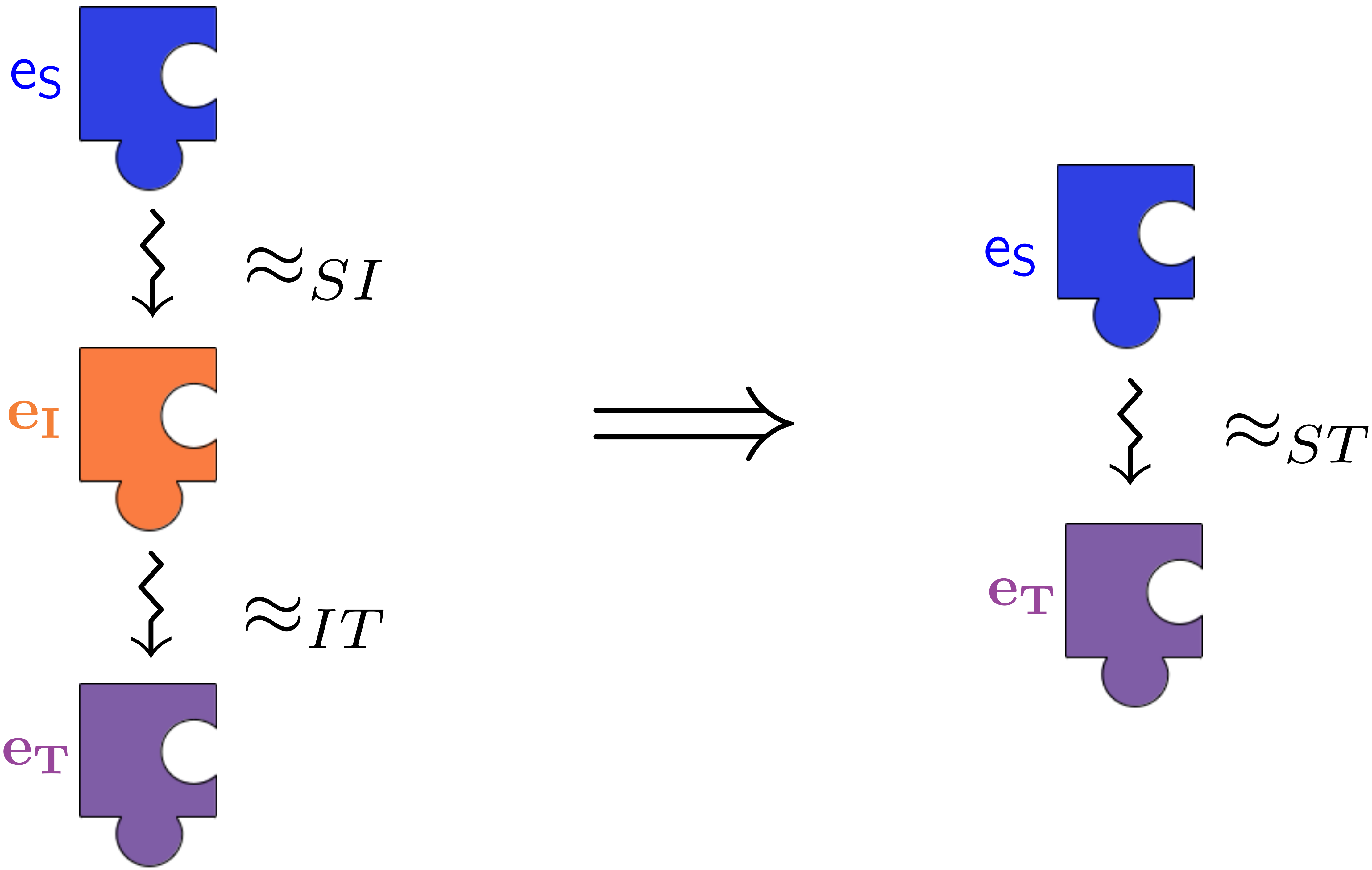


Proving Transitivity



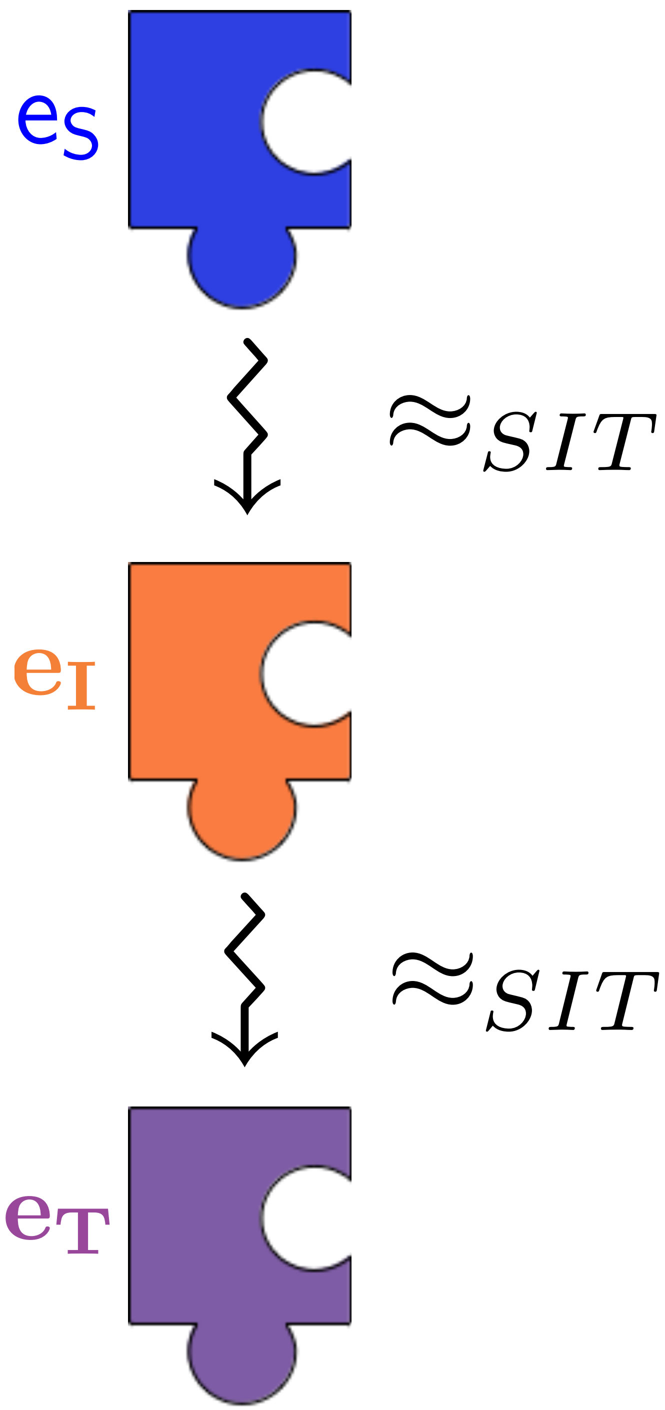
Vertical Compositionality

Transitivity

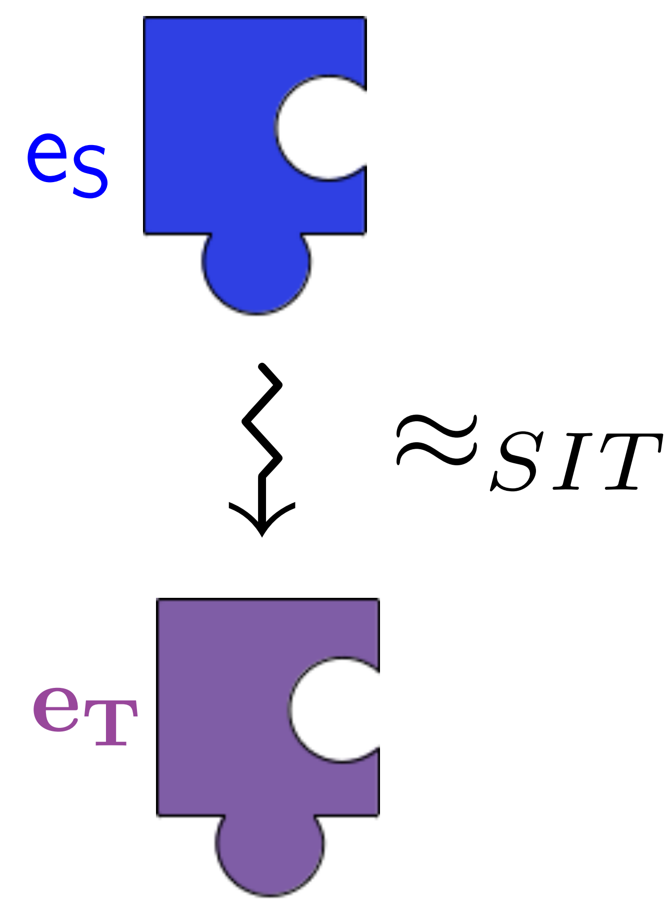


Transitivity

CompCompCert & Multi-lang



$\Longrightarrow$



Horizontal Compositionality

Pilsner
Neis et al.'15

Source-Independent Linking

Compositional CompCert
Stewart et al.'15

Multi-language ST
Perconti-Ahmed'14

Vertical Compositionality

Pilsner
Neis et al.'15

Transitivity

Compositional CompCert
Stewart et al.'15

Multi-language ST
Perconti-Ahmed'14

To Understand if Theorem is Correct…

Pilsner
Neis et al.'15

- source-target PILS

Compositional CompCert
Stewart et al.'15

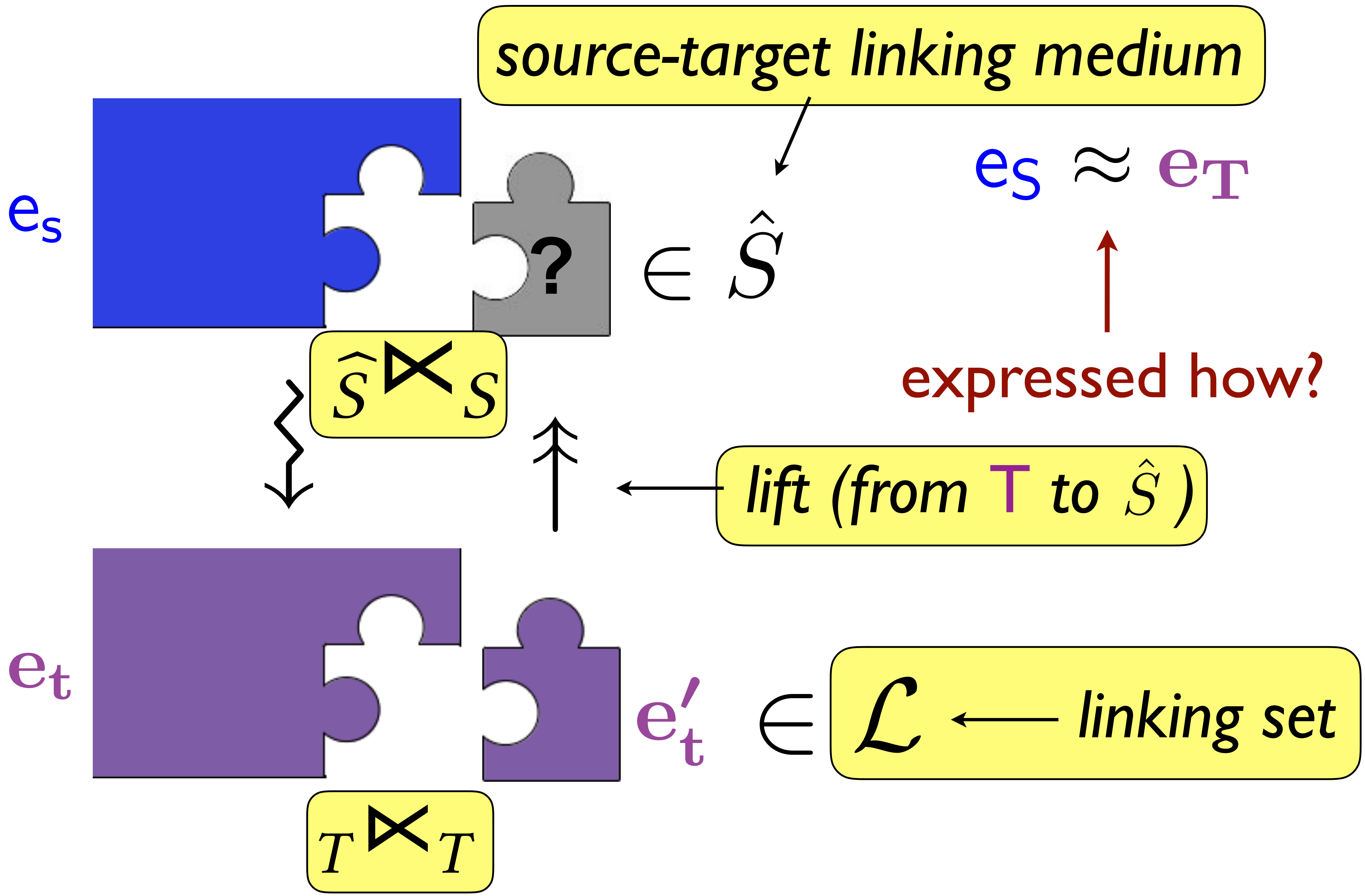
- interaction semantics
  & structured simulations

Multi-language ST
Perconti-Ahmed'14

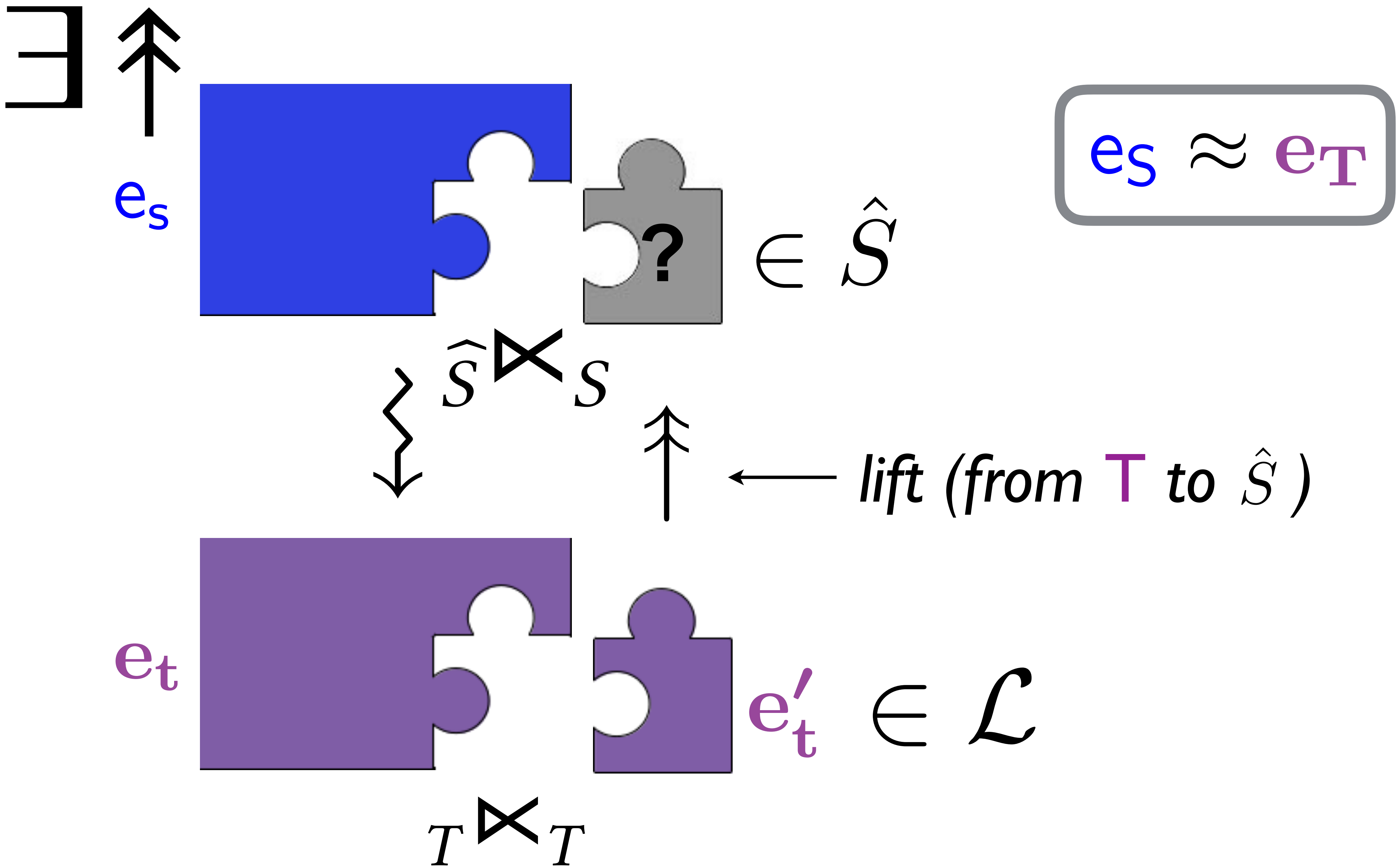
- source-target multi-language

Is there a generic CCC theorem?

Generic CCC Theorem?



Generic CCC Theorem



Generic CCC Theorem

$$\exists \Uparrow .\ \forall e_S, e_T .\ e_S \rightsquigarrow e_T \implies$$

$$\forall (e'_T, \varphi) \in \mathcal{L}.\ ok_{\ltimes}(e'_T, e_T) \implies$$

$$e'_T \ {}_T{\ltimes}_T\ e_T \ \ {}_T{\sqsubseteq}_{\widehat{S}} \ \Uparrow(e'_T, \varphi)\ {}_{\widehat{S}}{\ltimes}_S\ e_S$$

Generic CCC Theorem

$$\exists \twoheaduparrow.\ \forall e_S, e_T.\ e_S \rightsquigarrow e_T \implies$$

$$\forall (e'_T, \varphi) \in \mathcal{L}.\ ok_{\ltimes}(e'_T, e_T) \implies$$

$$e'_T\ {}_T{\ltimes}_T\ e_T\ \ {}_T{\sqsubset}_{\widehat{S}}\ \ \twoheaduparrow(e'_T, \varphi)\ {}_{\widehat{S}}{\ltimes}_S\ e_S$$

…and "lift" is inverse of "compile" on compiler output

$$\forall (e'_T, \varphi) \in \mathcal{L}.\ \forall e_S, e_T.\ e_S \rightsquigarrow e_T \implies$$

$$e'_T\ \ {}_T{\sqsubseteq}_T\ \ e_T \implies \twoheaduparrow(e'_T, \varphi)\ \ {}_{\widehat{S}}{\sqsubseteq}_S\ \ e_S$$

CCC Properties

Implies whole-program compiler correctness & correct separate compilation

Can be instantiated with different formalisms…

CCC with Pilsner

$\mathcal{L}$	$\{(e_T, \varphi) \mid \varphi =$ source component e_S & proof that $e_S \simeq e_T\}$
$\widehat{S}$	unchanged source language S
${}_{\widehat{S}}\ltimes_S$	unchanged source language linking
${}_{\widehat{S}}\sqsubseteq_S$	source language (whole program) observational equivalence
$\Uparrow(\cdot)$	$\Uparrow(e_T, (e_S, \_)) = e_S$

CCC with Multi-language

$\mathcal{L}$	$\{(e_T, \_) \mid$ where e_T is any target component $\}$
$\widehat{S}$	source-target multi-language ST
$_{\widehat{S}}\ltimes_S$	$e\ {}_{ST}\ltimes_{ST}\ e_S$
$_{\widehat{S}}\sqsubset_S$	run $\widehat{S}$ according to multi-lang ST, compare with running S
$\Uparrow(\cdot)$	$\Uparrow(e_T, \_) = \mathcal{ST}(e_T)$

Vertical Compositionality for Free

when $\Uparrow_{ST} = \Uparrow_{SI} \circ \Uparrow_{IT}$

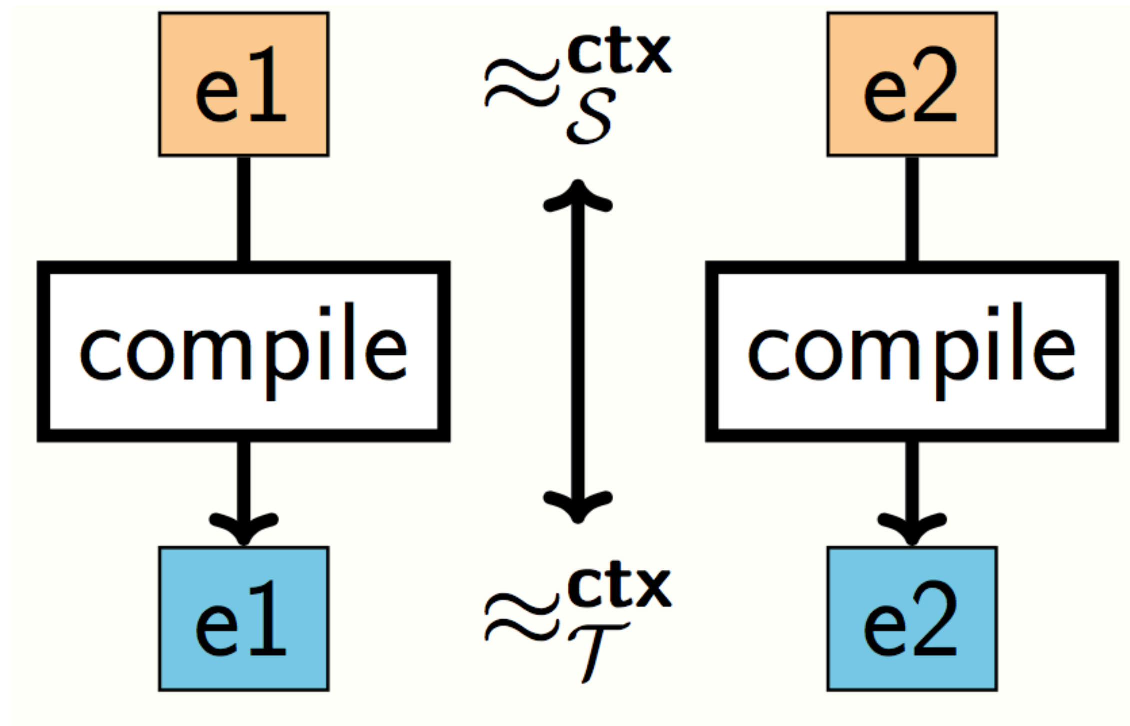
i.e., when lift $\Uparrow$ is a back-translation that maps every $e_T \in \mathcal{L}$ to some e_S (or an approximate back-translation that takes the interaction between e_T and some compiled e_S into account).

Fully abstract compilers have such back-translations!

Bonus of vertical comp: can verify different passes using different formalisms to instantiate CCC

Fully Abstract Compilers

preserve equivalence



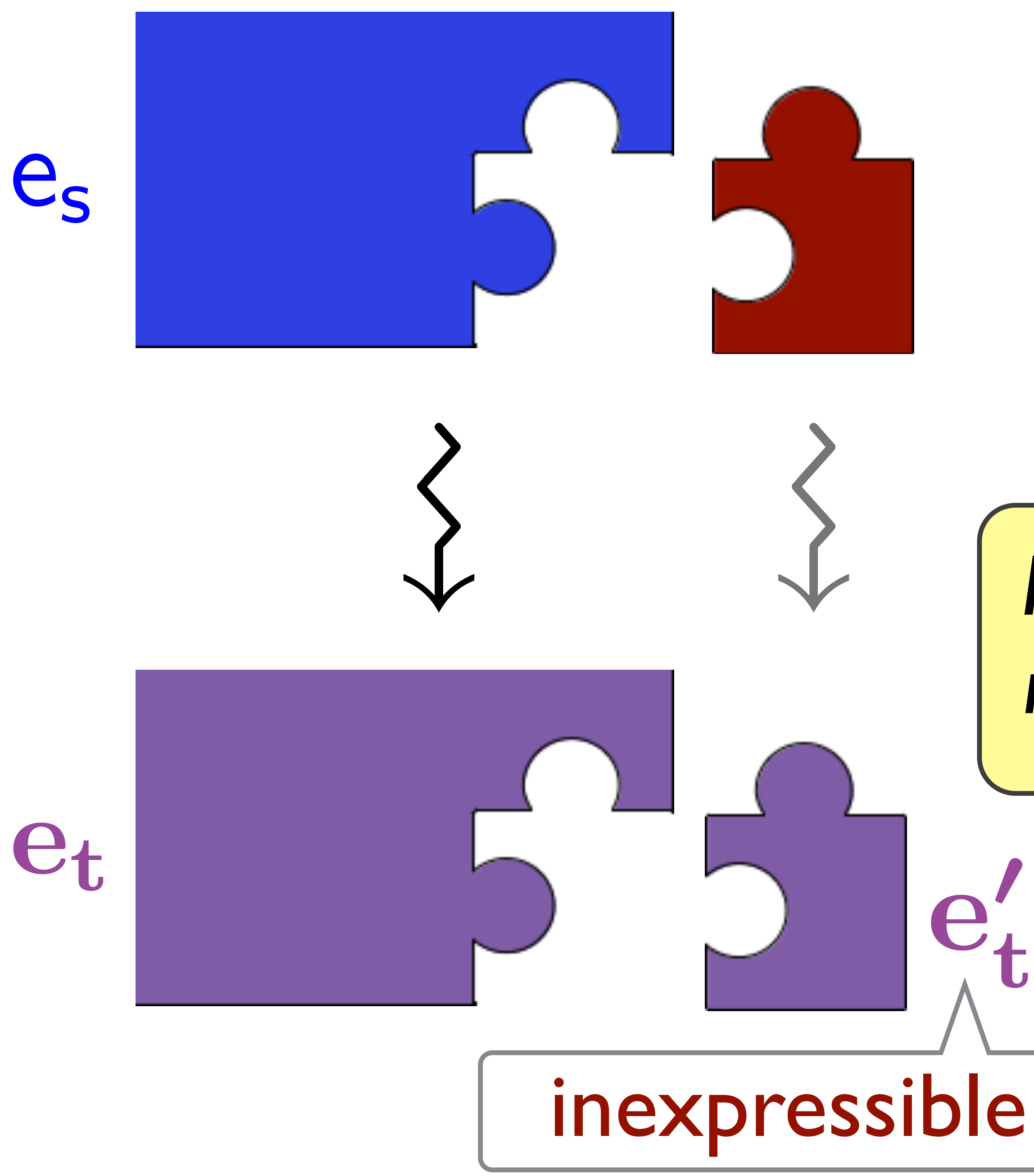
- ensure a compiled component does not interact with any target behavior that is inexpressible in S
- Do we want to link with behavior inexpressible in S? Or do we want fully abstract compilers?
  - We want both!

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]

Stepping back…

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

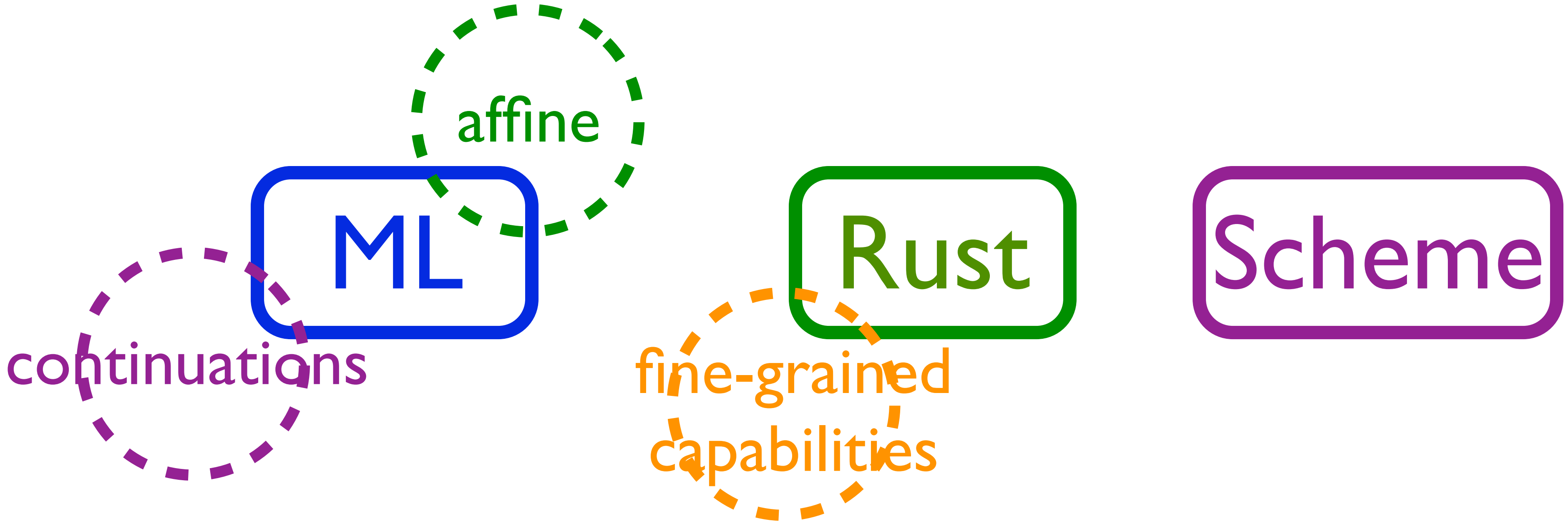
ML — C FFI

Rust — unsafe

Java — JNI

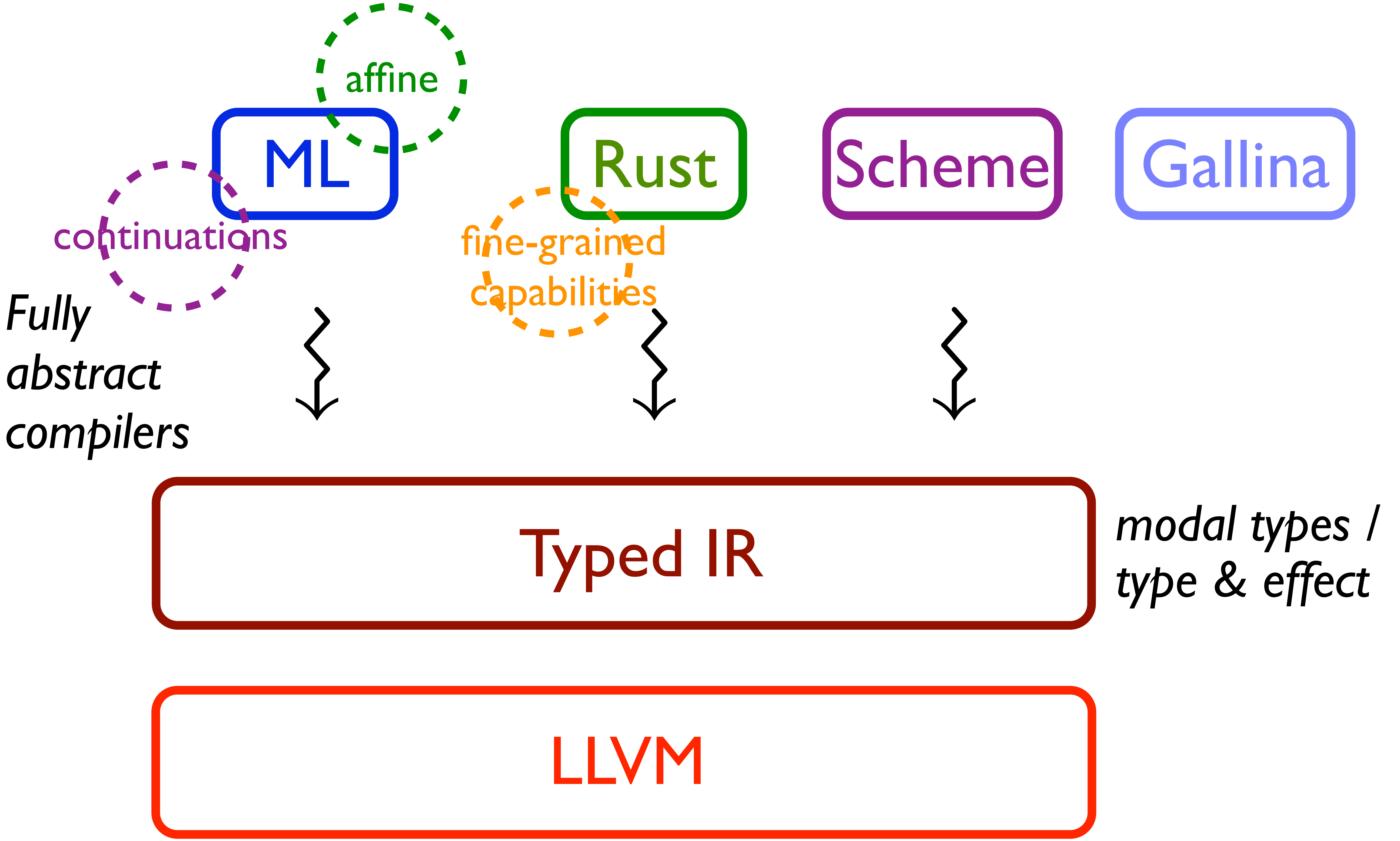
Design linking types extensions that support safe interoperability with other languages

PL Design, Linking Types

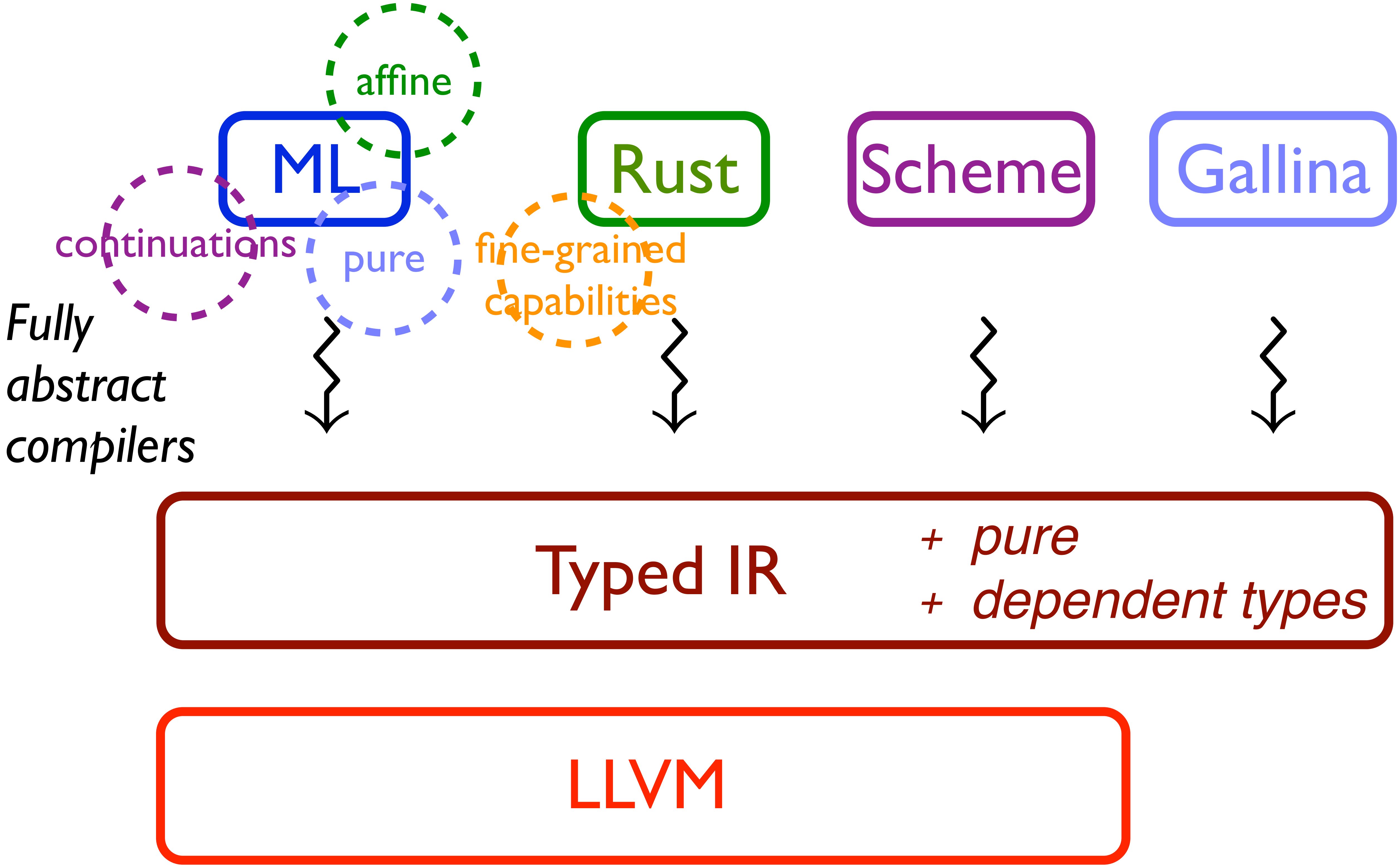


Only need linking types extensions to interact with behavior inexpressible in your language.

PL Design, Linking Types, Compilers



PL Design, Linking Types, Compilers



Linking Types

- 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

Compositional Compiler Verification

- CompCert started a renaissance in compiler verification
  - major advances in mechanized proof
- Now we need: Compositional Compiler Correctness
  - that applies to world of multi-language software…
  - but source-independent linking and vertical compositionality are at odds
  - fully abstract compilation and linking types could help improve multi-language software toolchains