Compositional Compiler Verification & Secure Compilation

Amal Ahmed

Northeastern University

Compiler Correctness

= semantics-preserving compilation

$$s \leadsto t \quad \Longrightarrow \quad s \approx t$$

compiles to

same behavior

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. CompCert

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. CompCertTSO

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

Vellvm verified LLVM

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

CAKEML

⋮

Why CompCert had such impact…

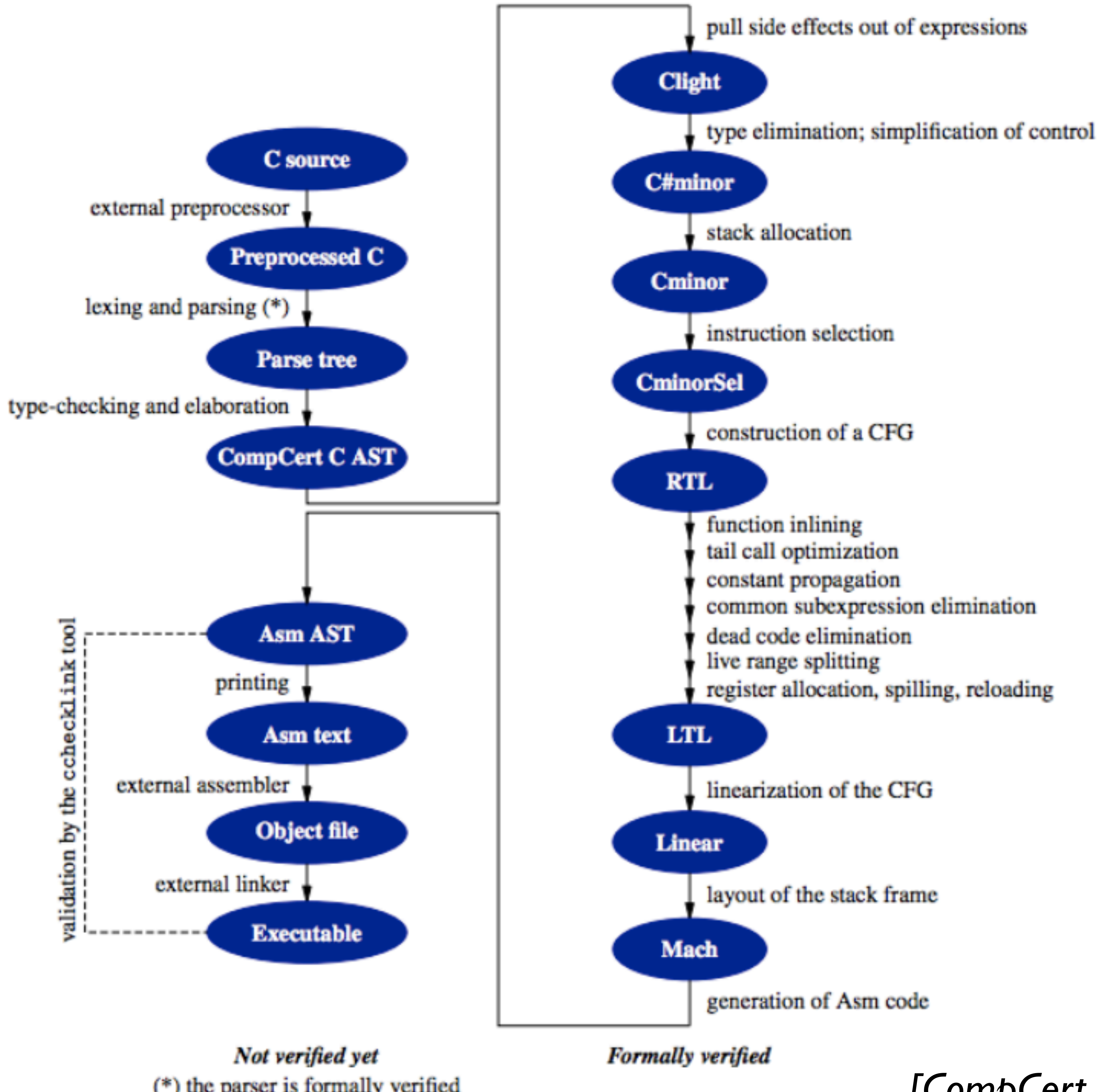
- Demonstrated that realistic verified compilers are both *feasible* and bring *tangible benefits*

> 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…

- Demonstrated that realistic verified compilers are both *feasible* and bring *tangible benefits [Yang et al. PLDI'11]*

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



[CompCert manual 2015]

Why CompCert had such impact…

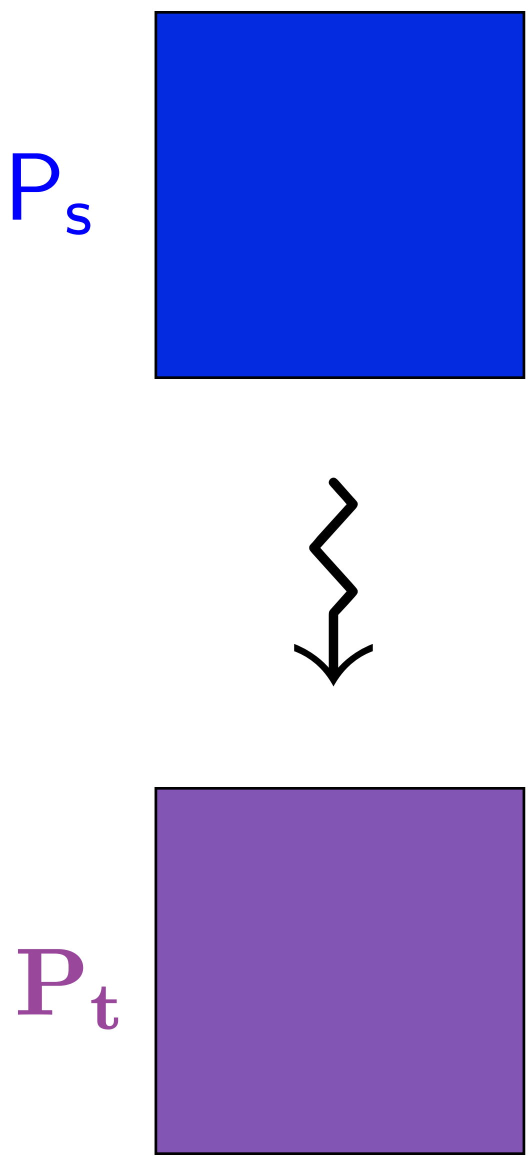
- Demonstrated that realistic verified compilers are both *feasible* and bring *tangible benefits [Yang et al. PLDI'11]*

- 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

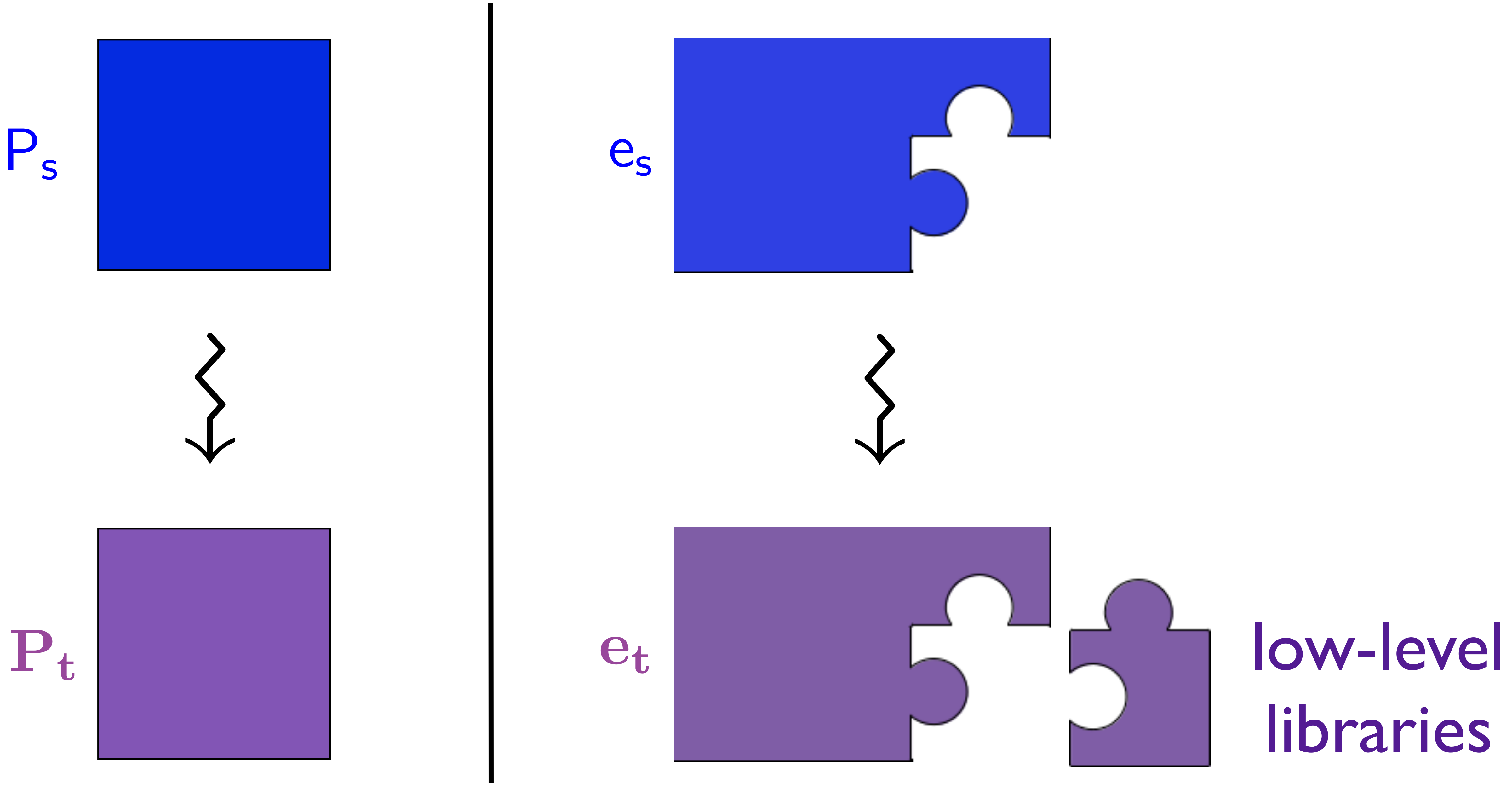
Correct compilation guarantee only applies to whole programs!



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)

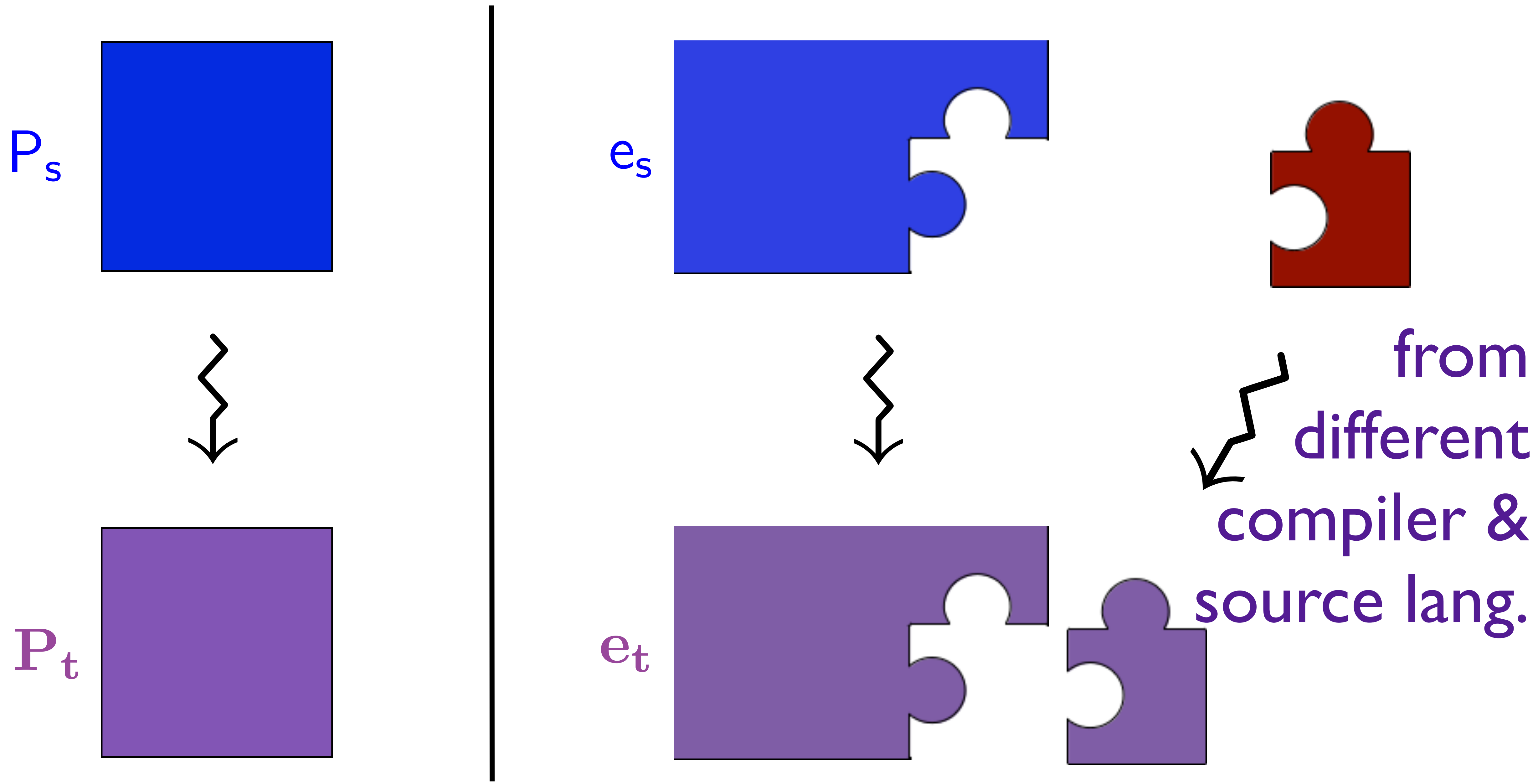
Problem: Whole-Program Assumption

Correct compilation guarantee only applies to whole programs!

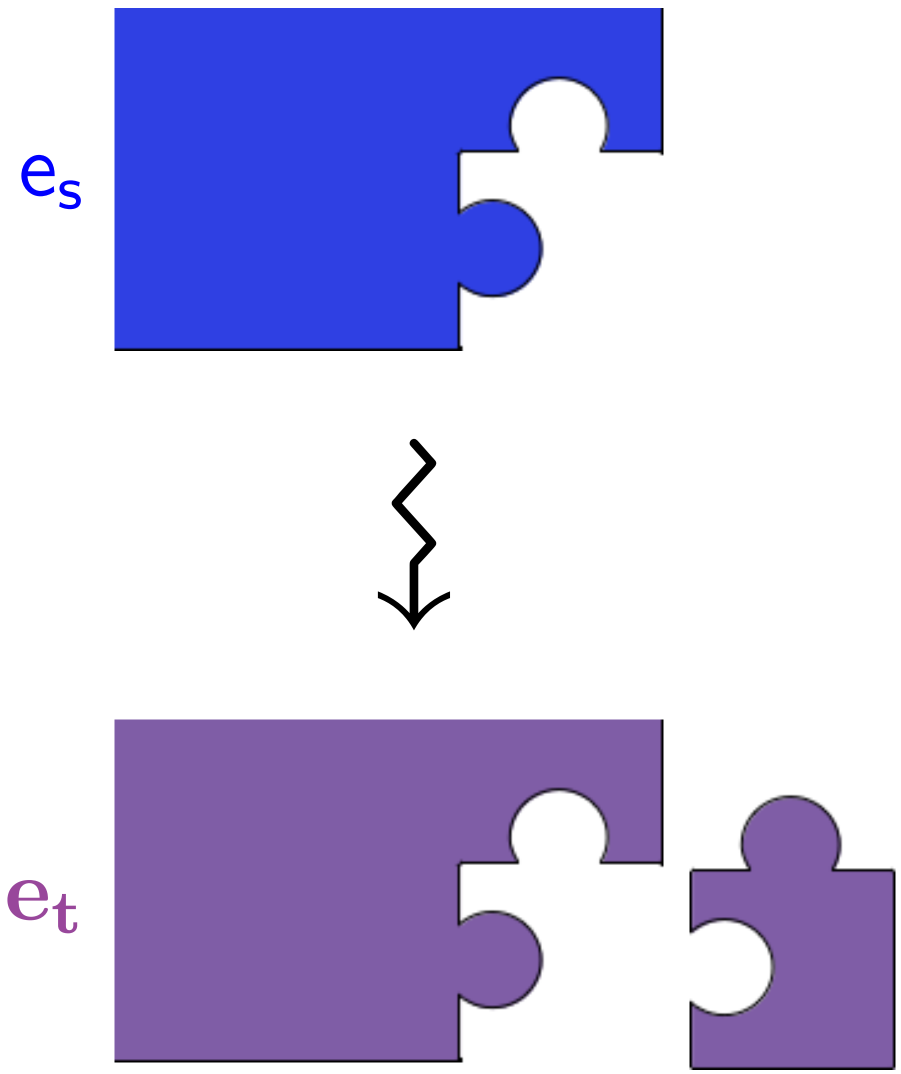


Problem: Whole-Program Assumption

Correct compilation guarantee only applies to whole programs!



"Compositional" Compiler Verification



This Lecture…

- why specifying compositional compiler correctness theorems is hard
- survey recent results
- generic CCC theorem to guide future compiler correctness theorems
- lessons for formalizing linking & verifying multi-pass compilers

Compiler Correctness

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

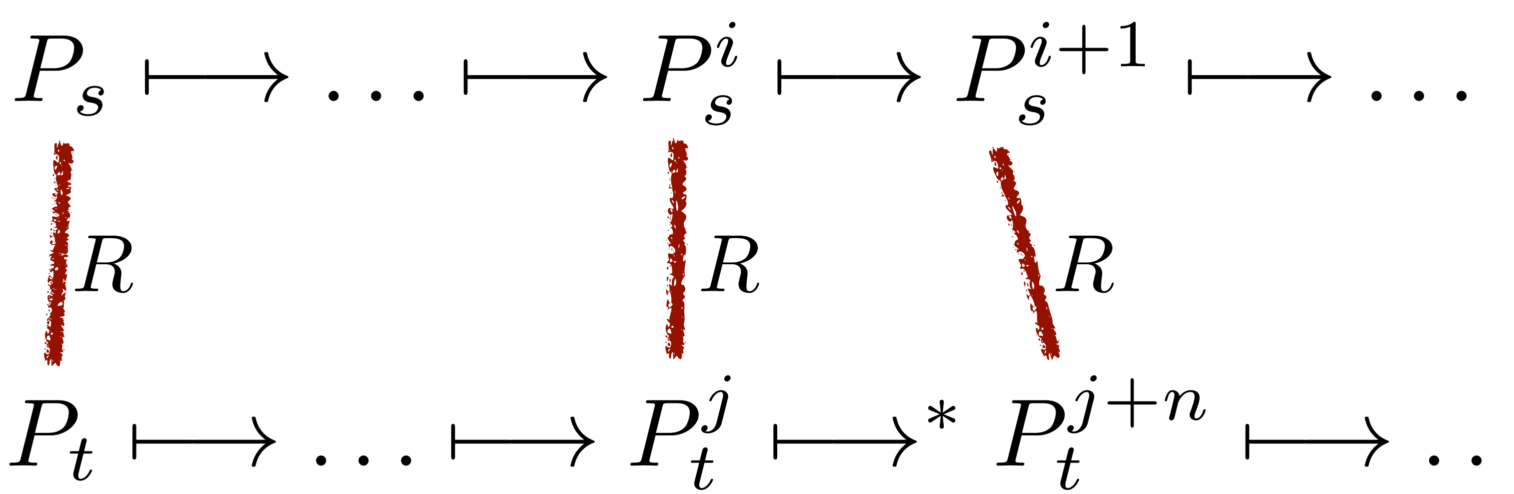
↑ expressed how?

Whole-Program Compiler Correctness

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

expressed how?
"closed" simulations

CompCert



Whole-Program Compiler Correctness

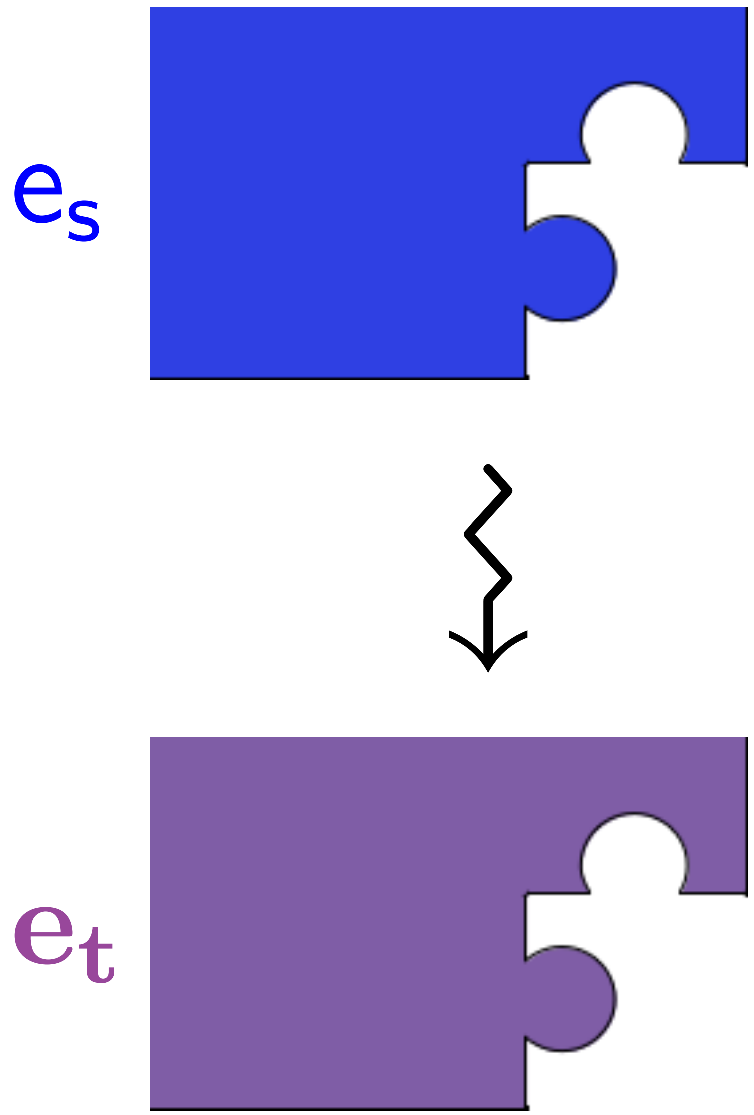
$$P_s \rightsquigarrow P_t \implies P_t \sqsubseteq P_s$$

behavior refinement

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

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

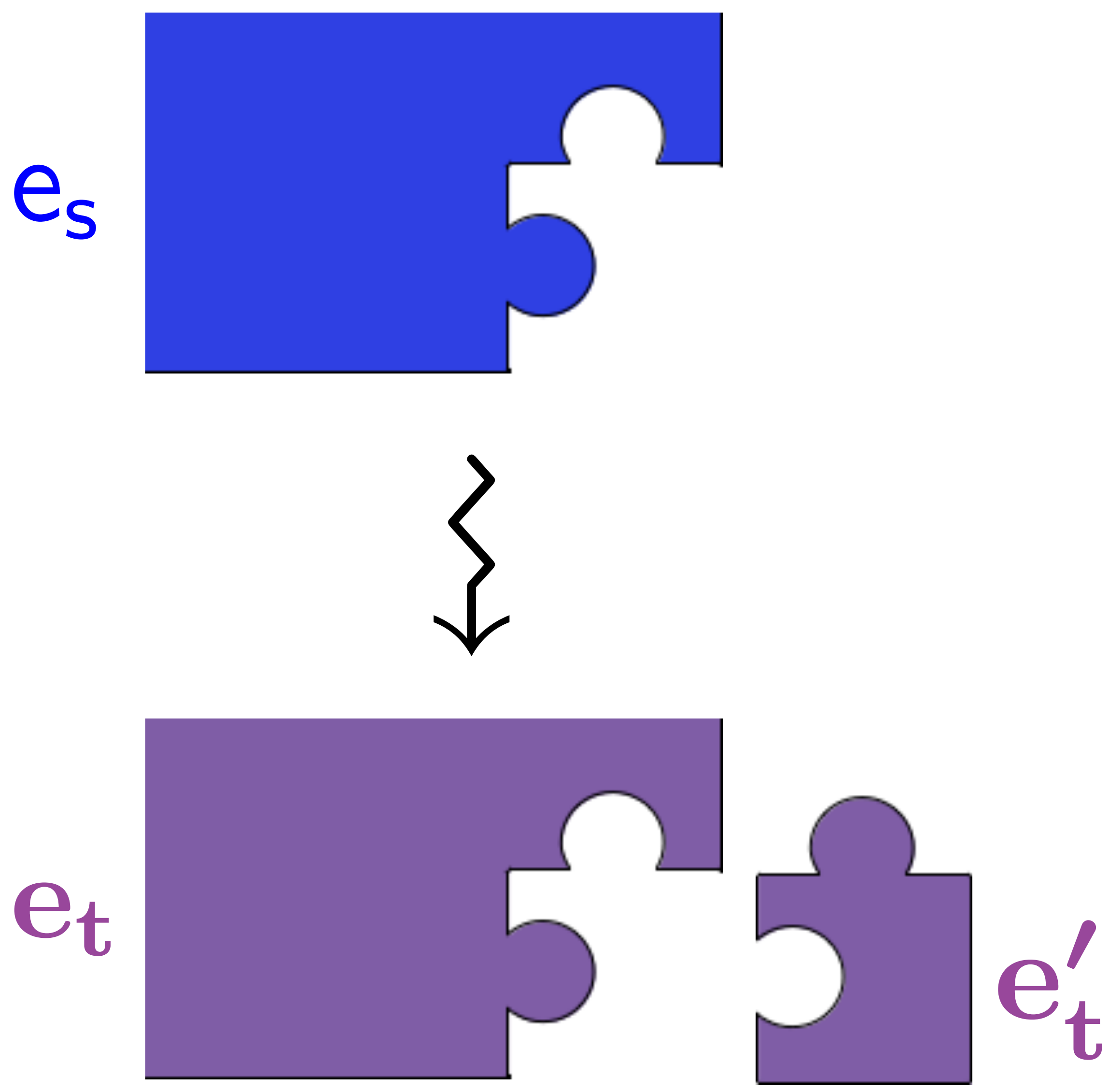
Correct Compilation of Components?



$$e_S \approx e_T$$

expressed how?

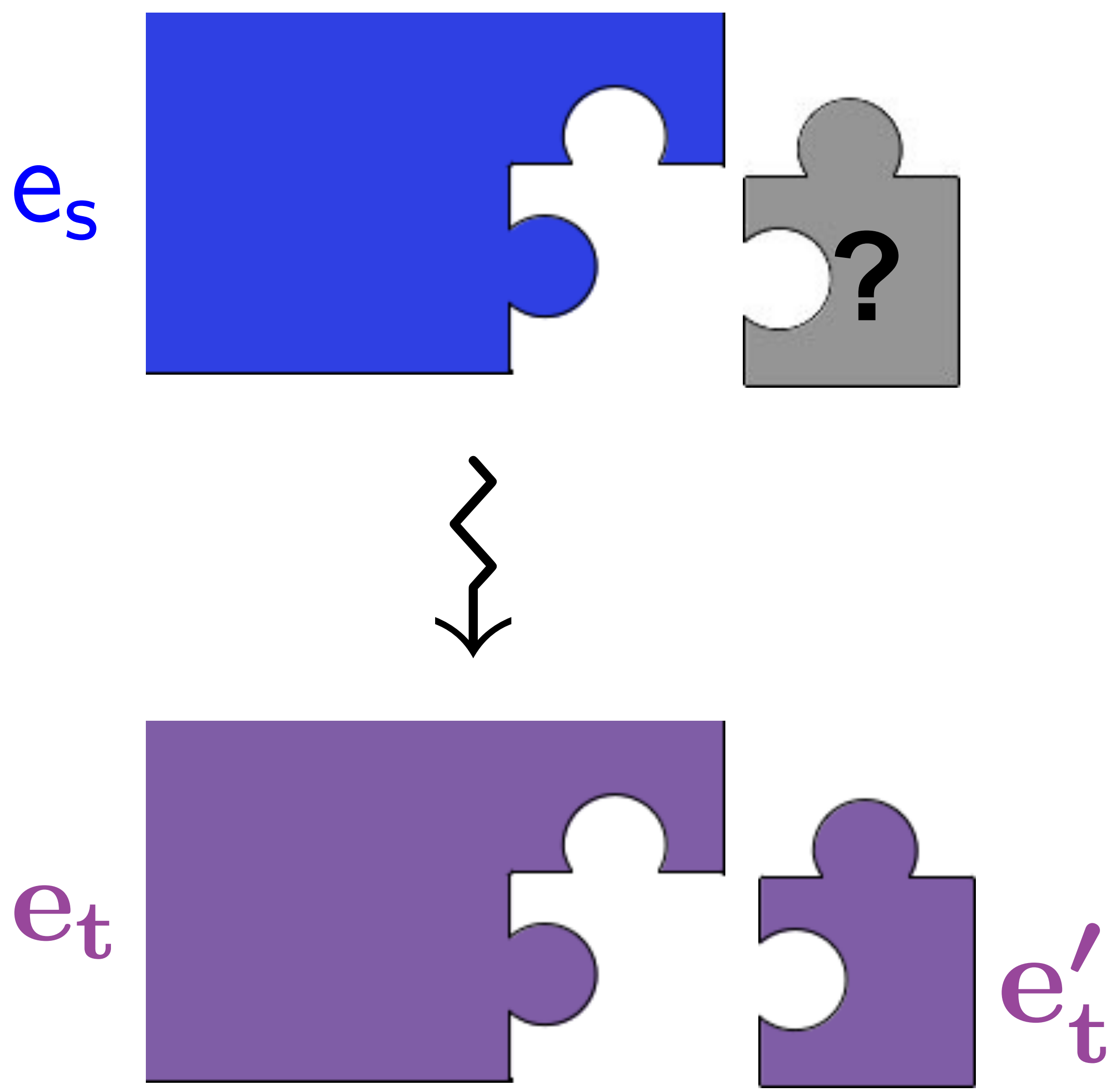
Correct Compilation of Components?



$$e_S \approx e_T$$

expressed how?

Correct Compilation of Components?

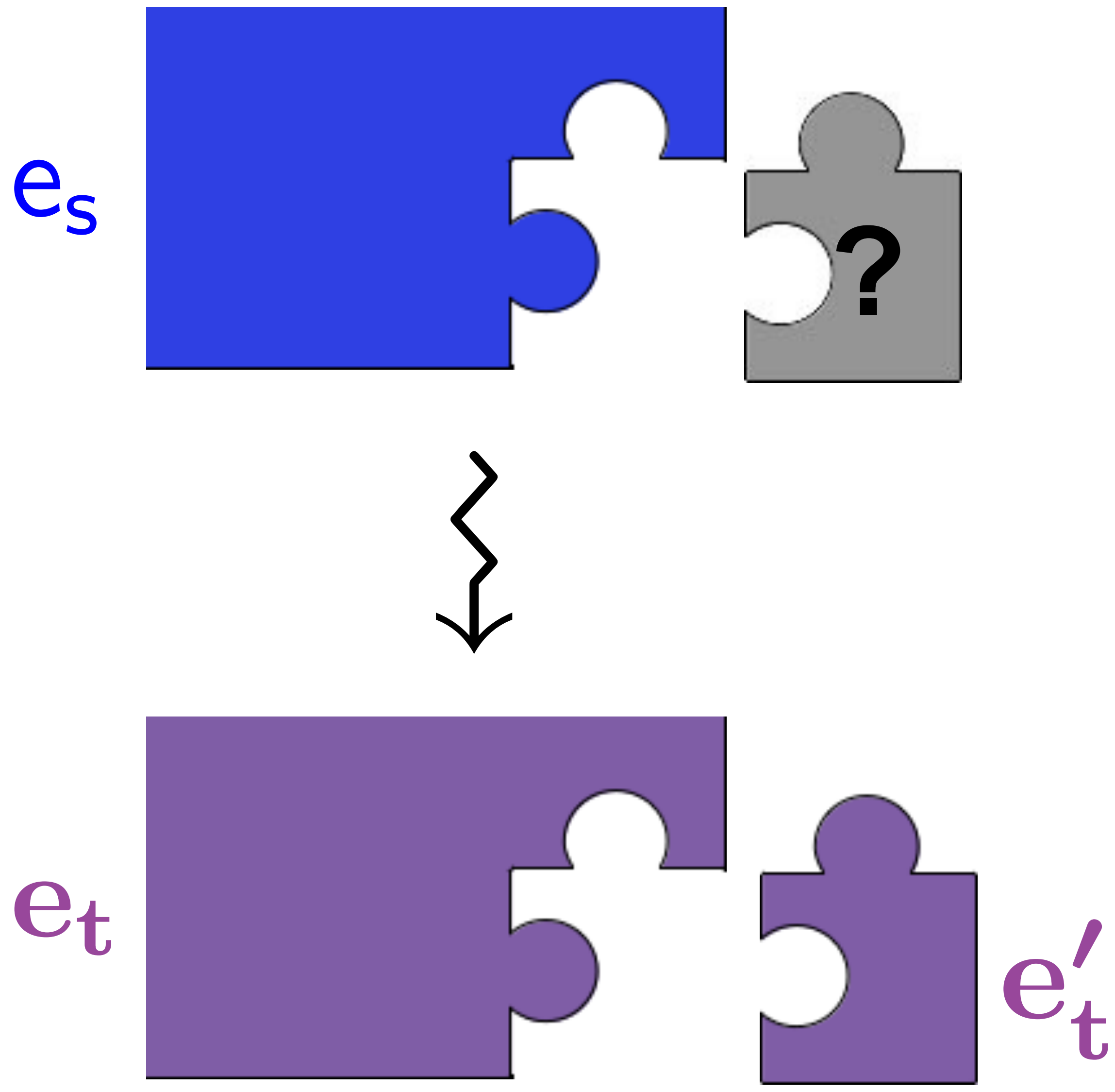


$$e_S \approx e_T$$

↑

expressed how?

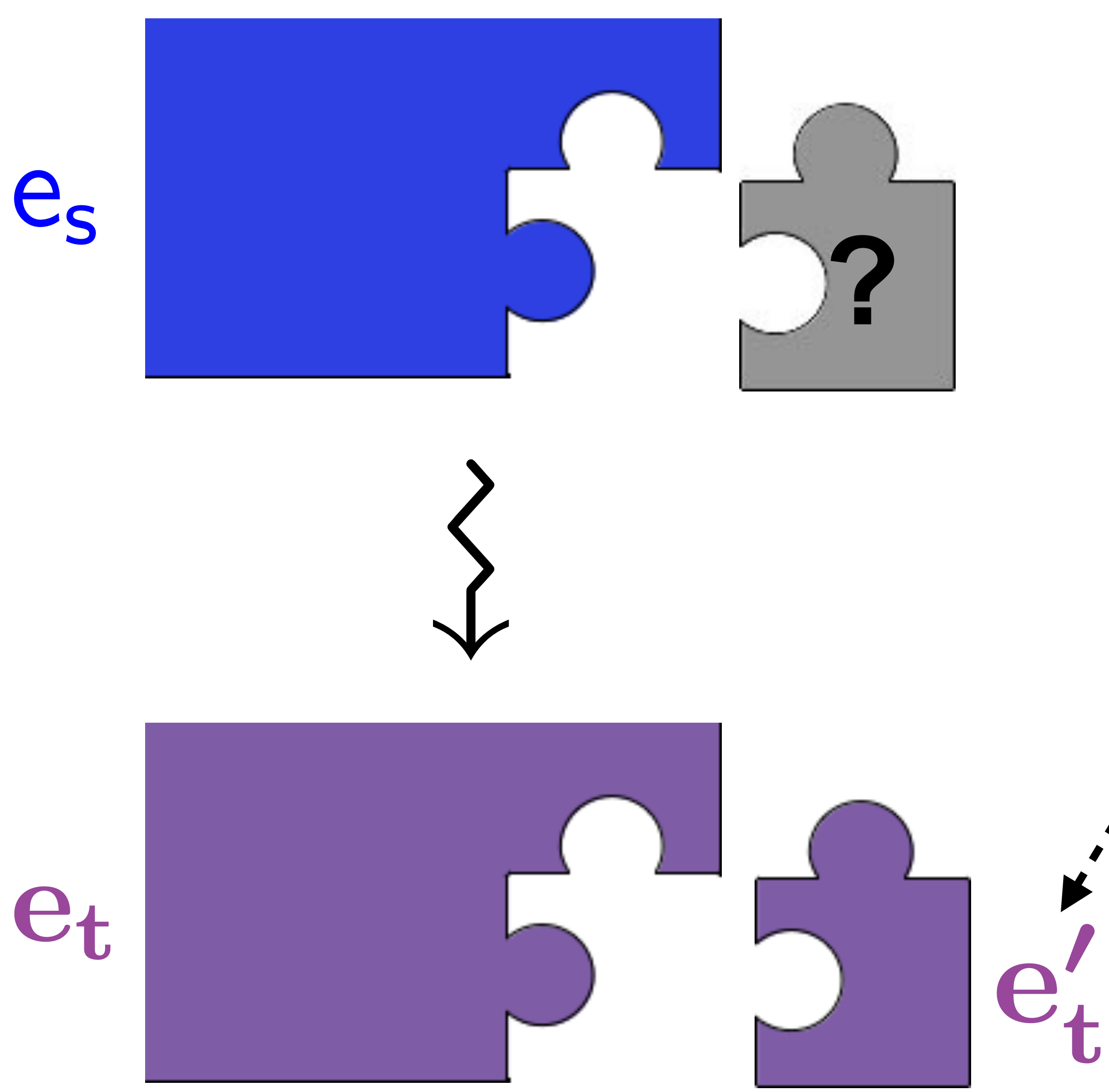
"Compositional" Compiler Correctness



$e_S \approx e_T$

expressed how?

“Compositional” Compiler Correctness



$e_S \approx e_T$

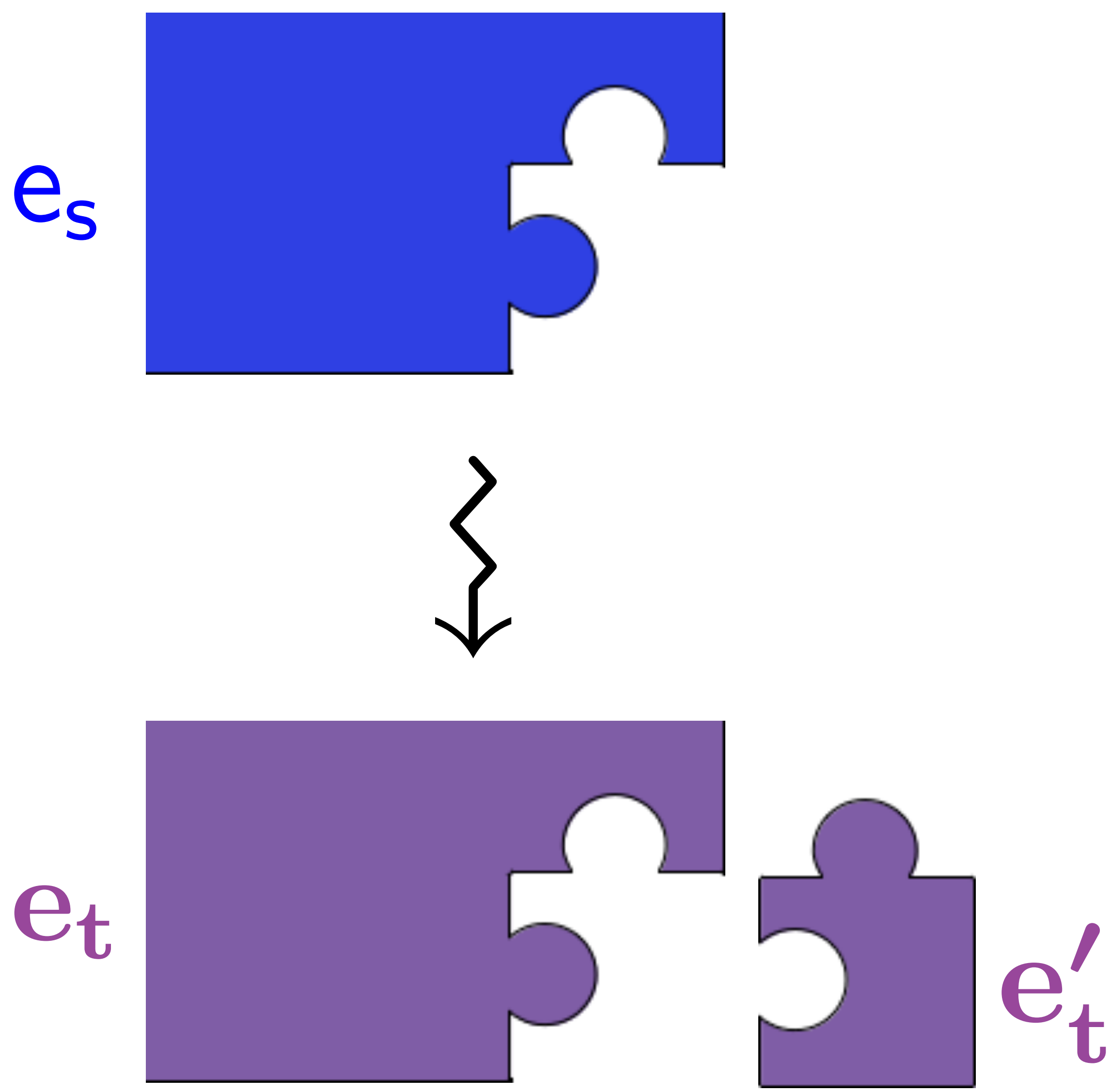
expressed how?

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

Is behavior of e'_t *expressible in S?*

"Compositional" Compiler Correctness

If we want to verify realistic compilers…



$e_S \approx e_T$

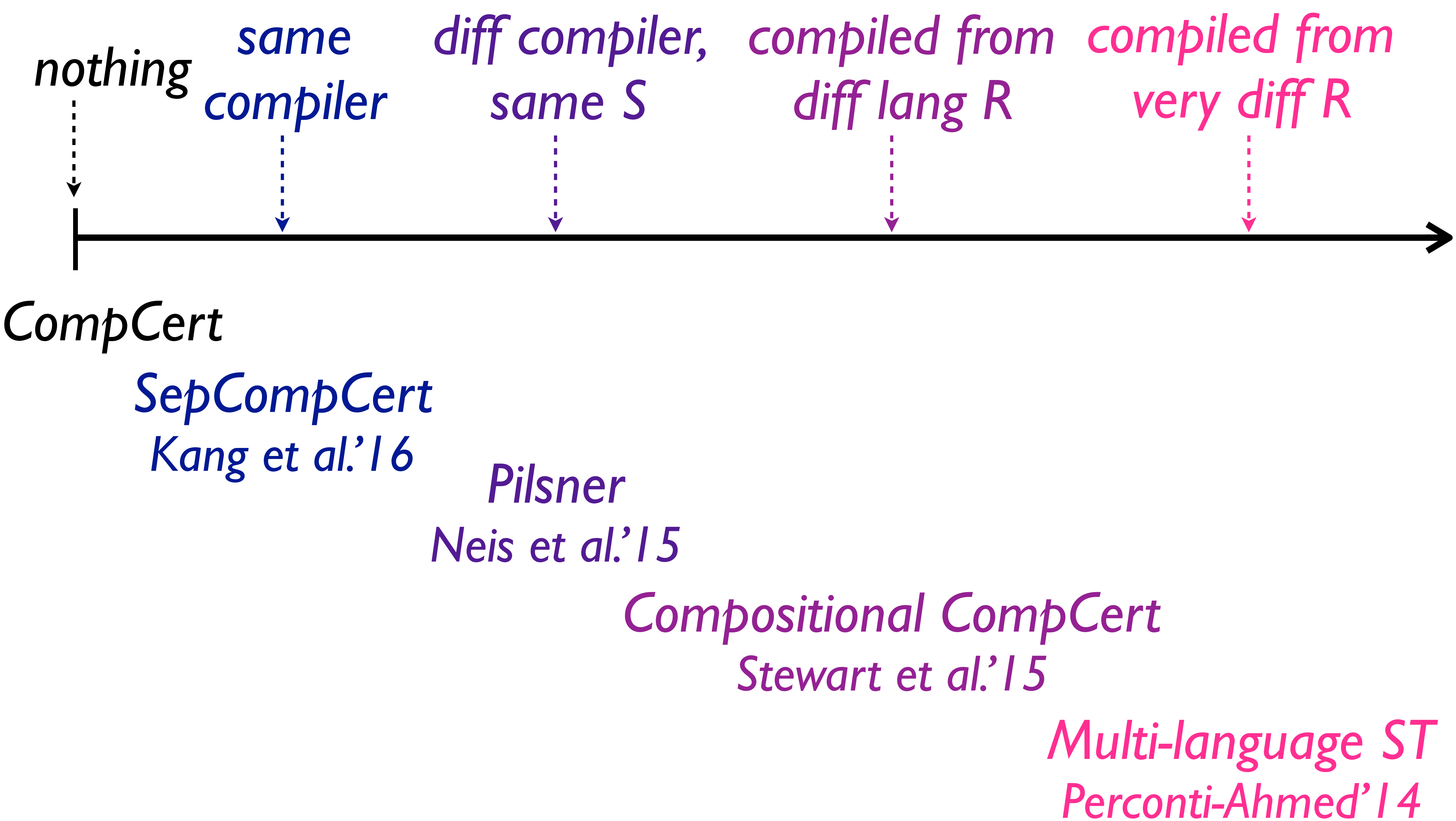
Definition should:

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

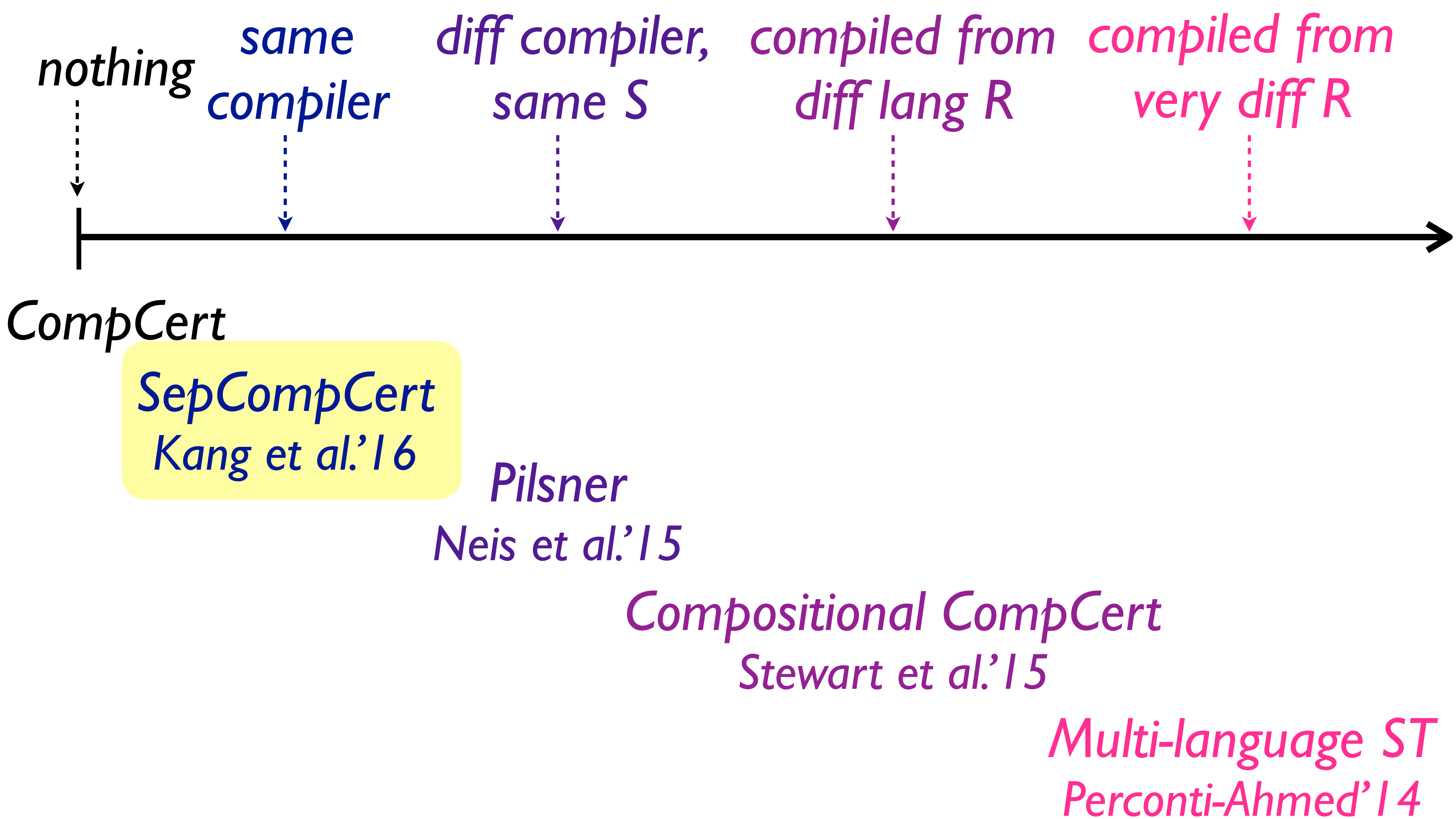
Next

- Survey of "compositional" compiler correctness results
  - how to express $e_S \approx e_T$
- How does the choice affect:
  - what we can **link** with *(horizontal compositionality)*
  - how we check if some 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



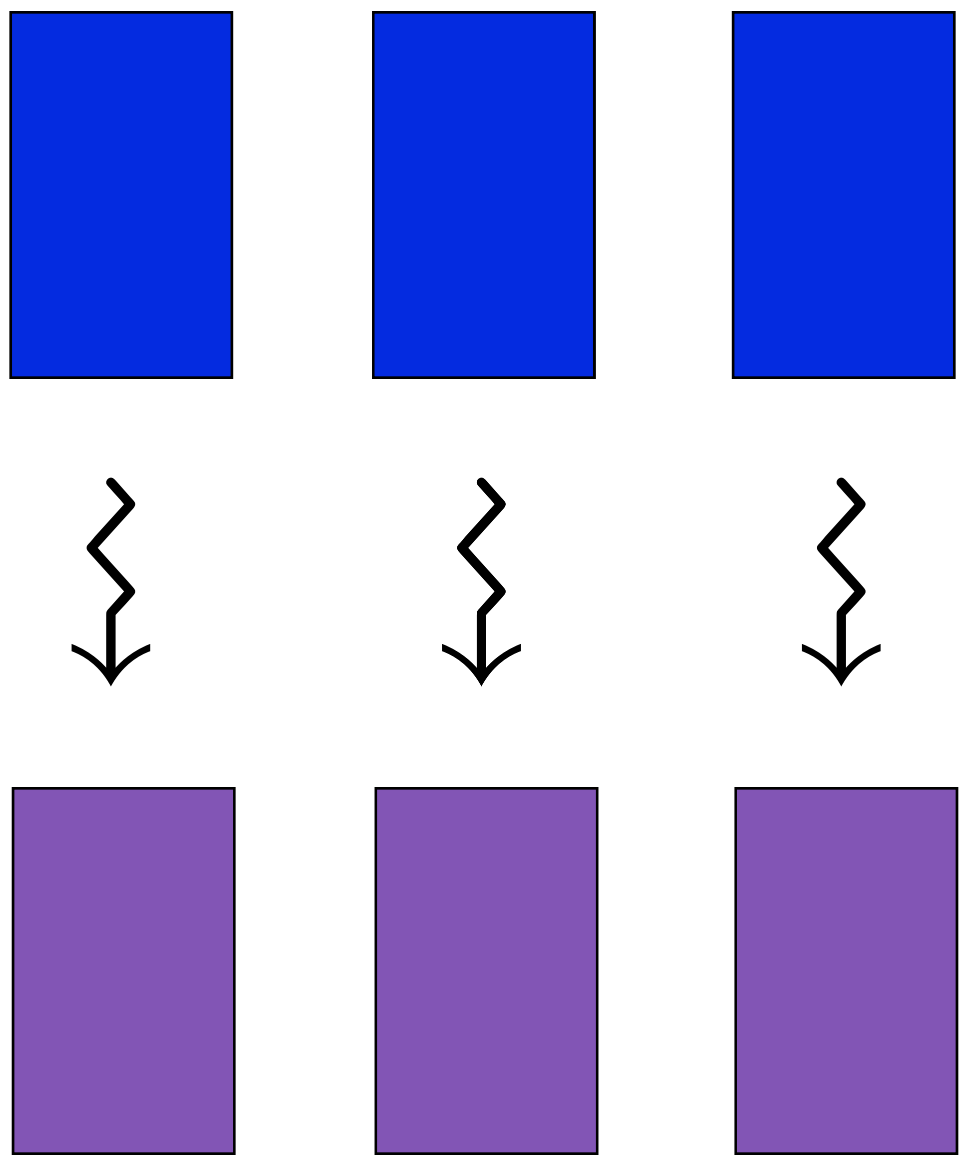
What we can link with



Approach: Separate Compilation

SepCompCert

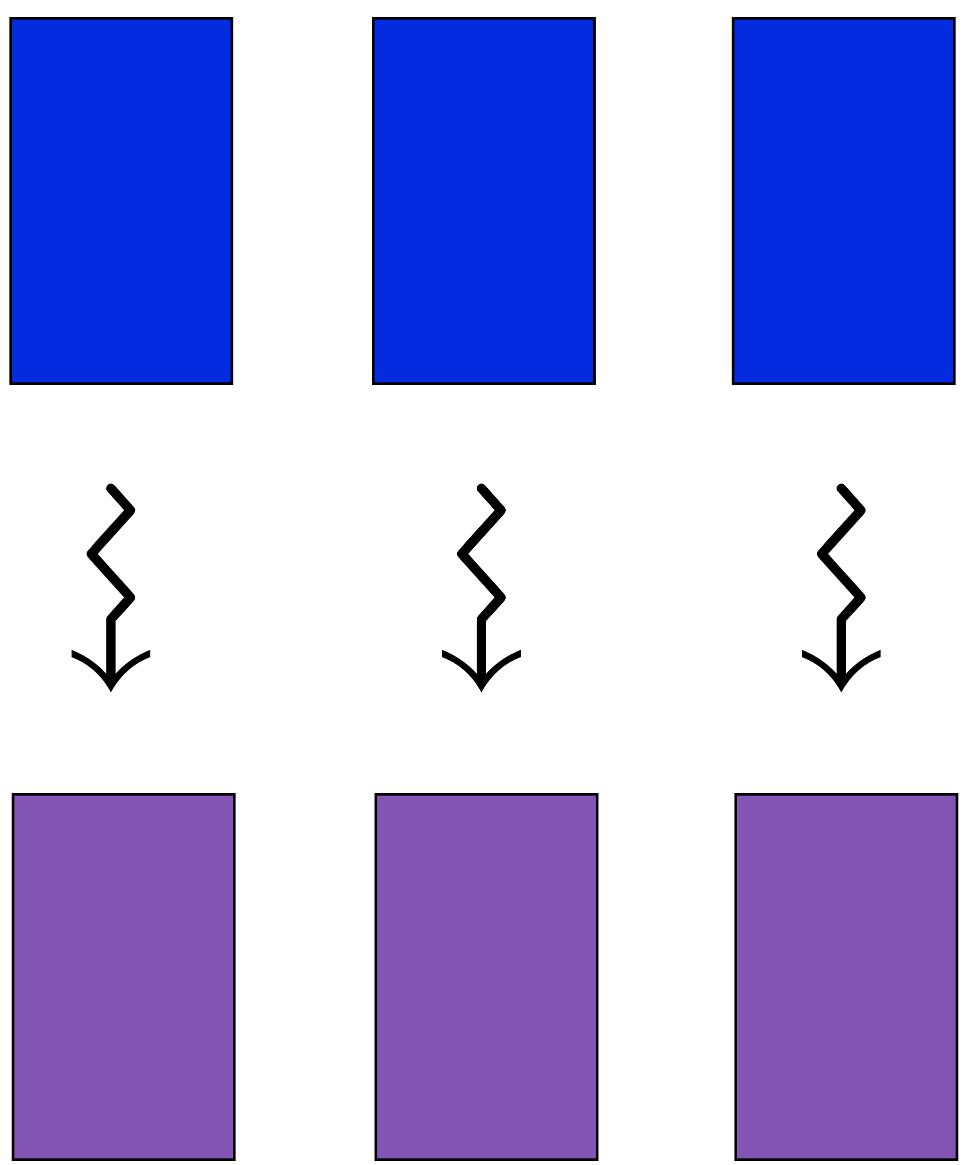
[Kang et al. '16]



Approach: Separate Compilation

SepCompCert

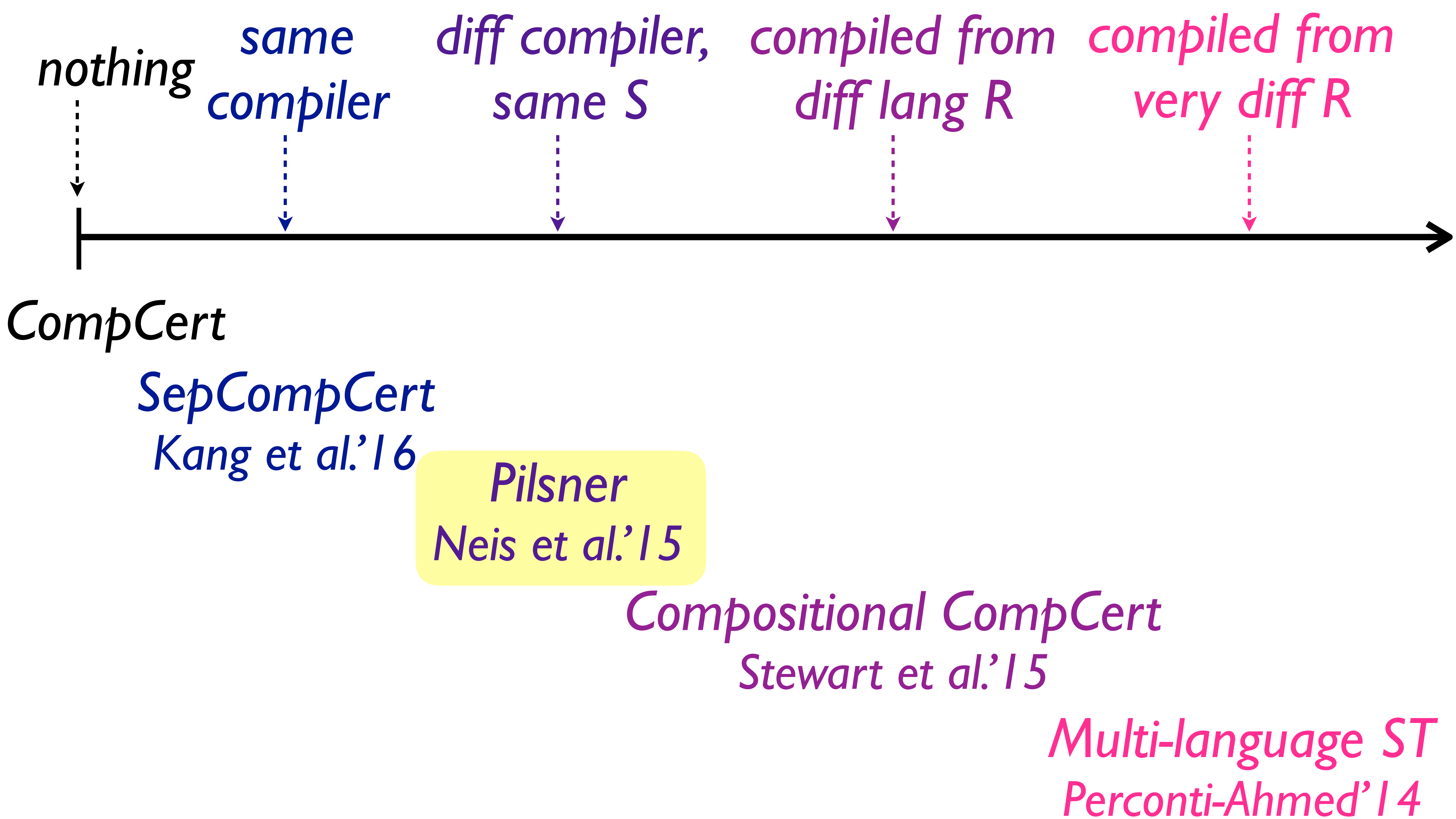
[Kang et al. '16]



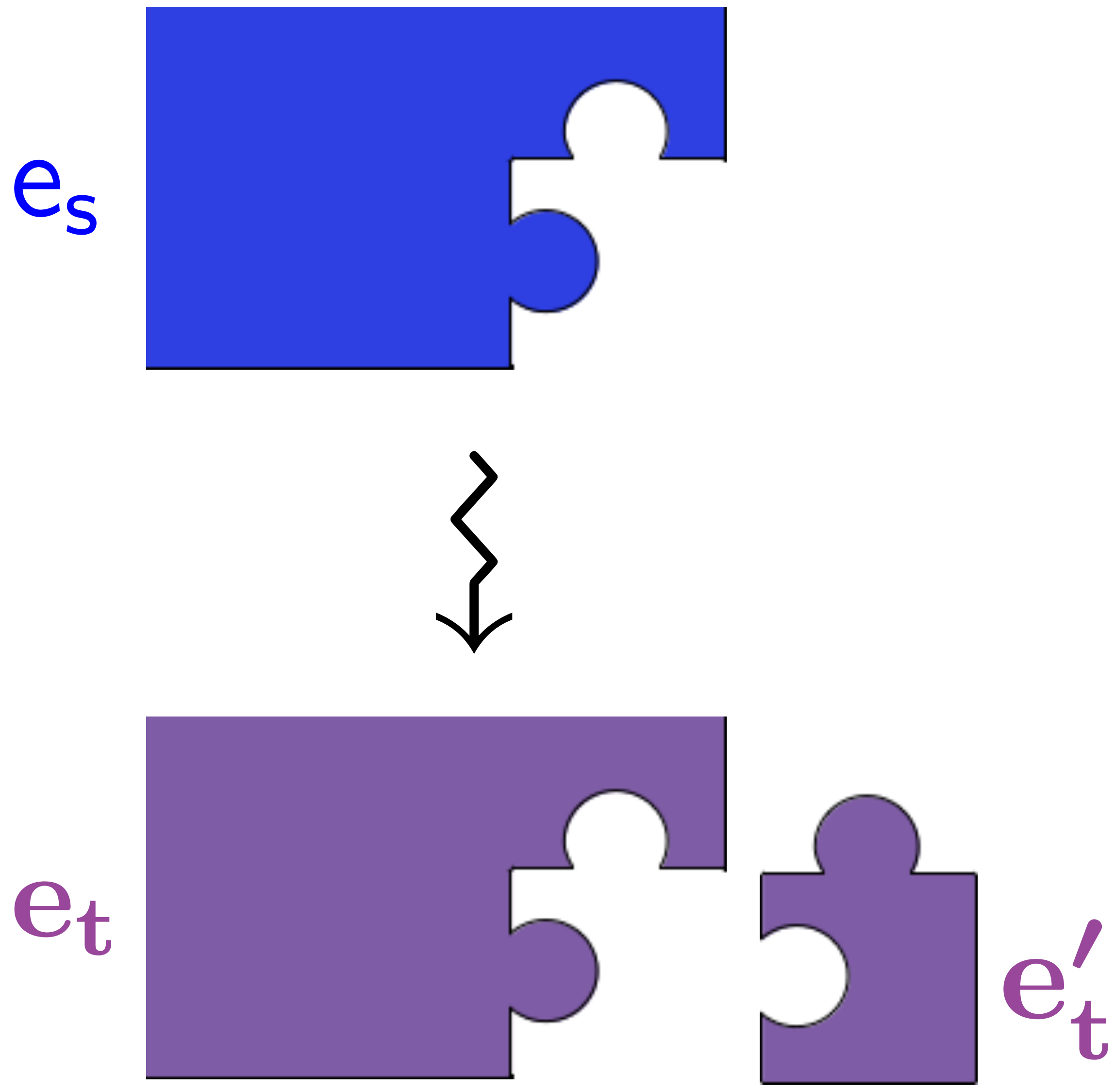
Level A correctness:
exactly same compiler

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

What we can link with



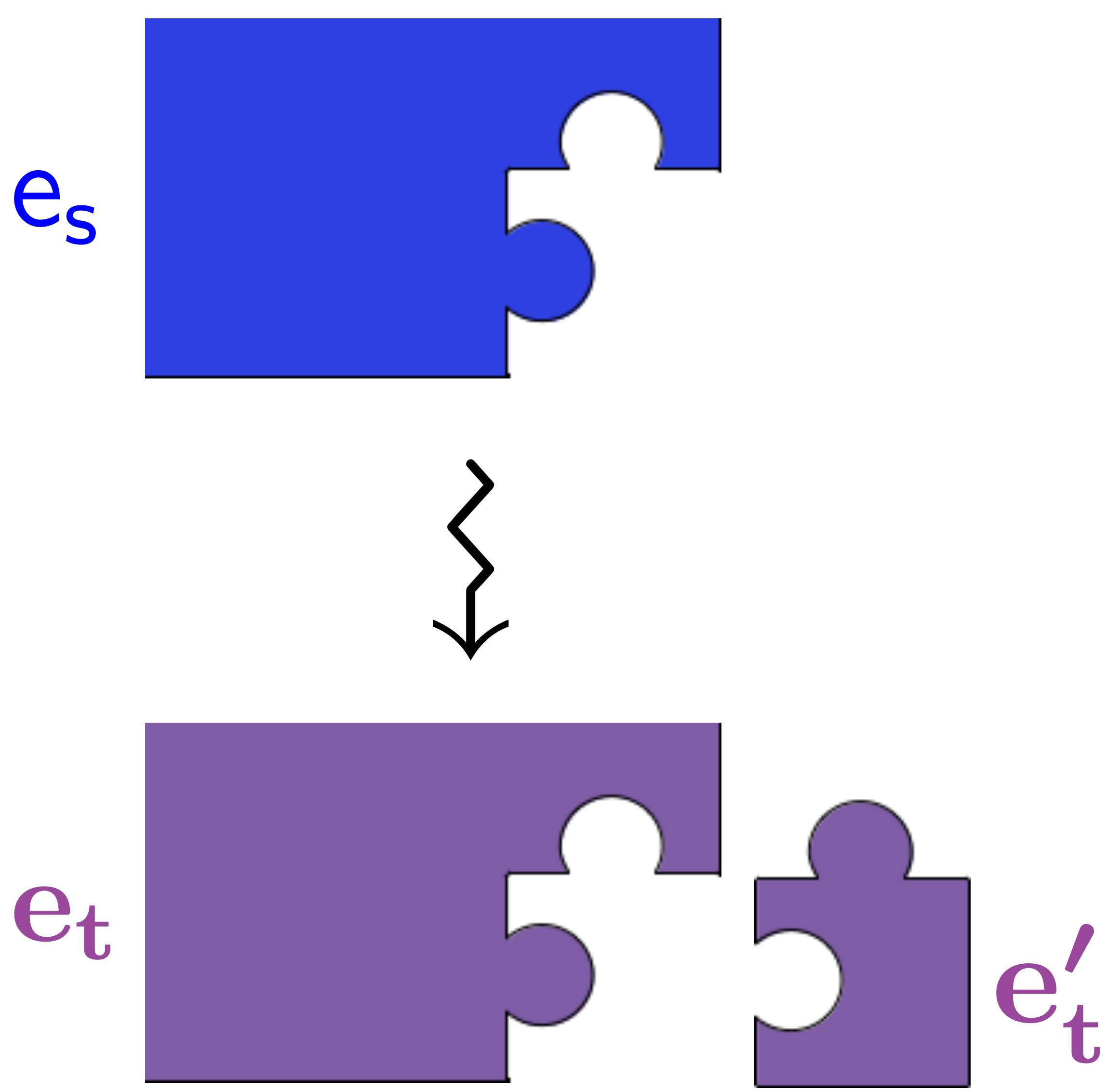
Approach: Cross-Language Relations



Cross-language relation

$e_S \approx e_T$

Approach: Cross-Language Relations



Cross-language relation

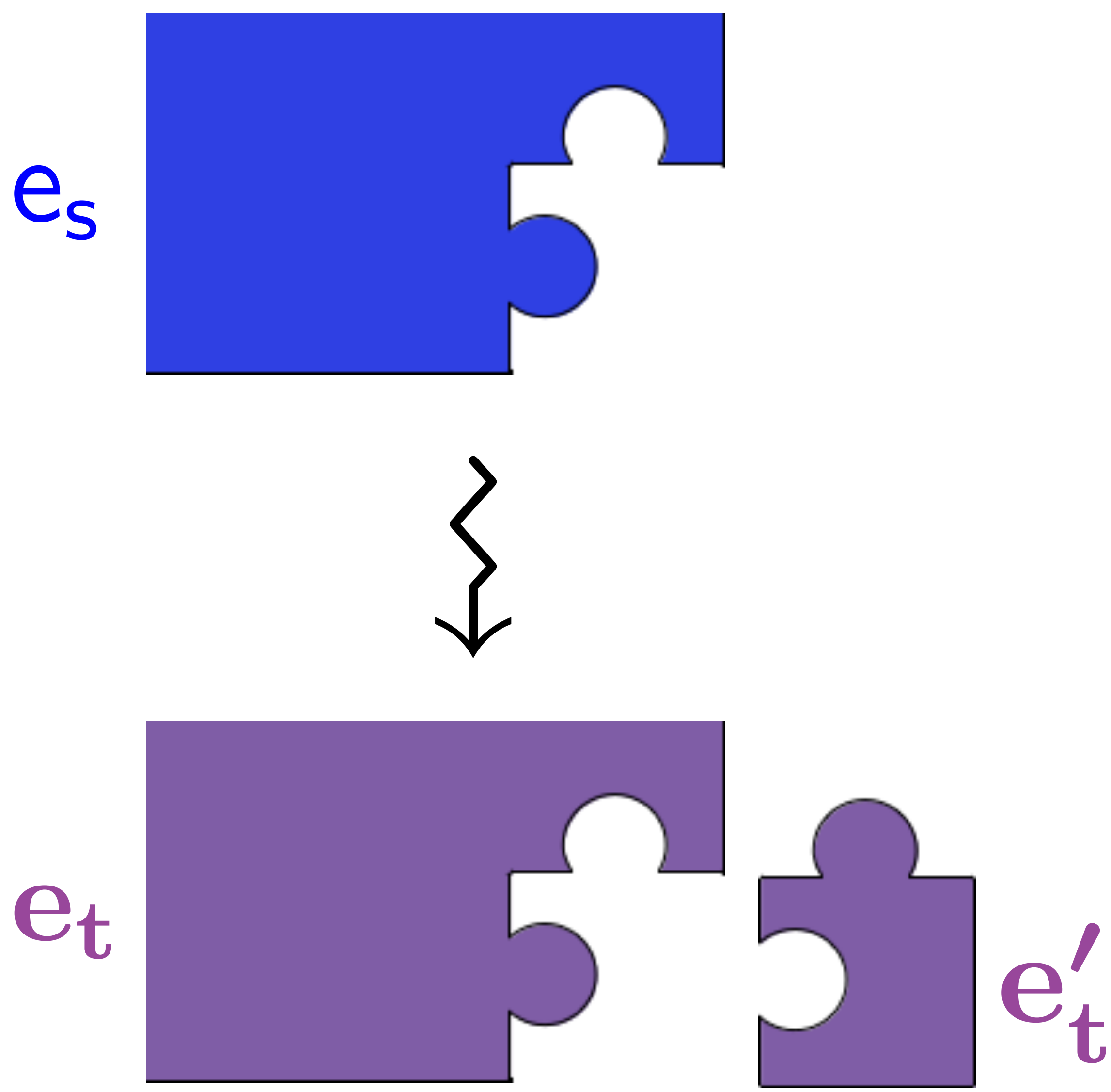
$$e_S \approx e_T$$

Compiling ML-like langs:

Logical relations

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

Approach: Cross-Language Relations



Cross-language relation

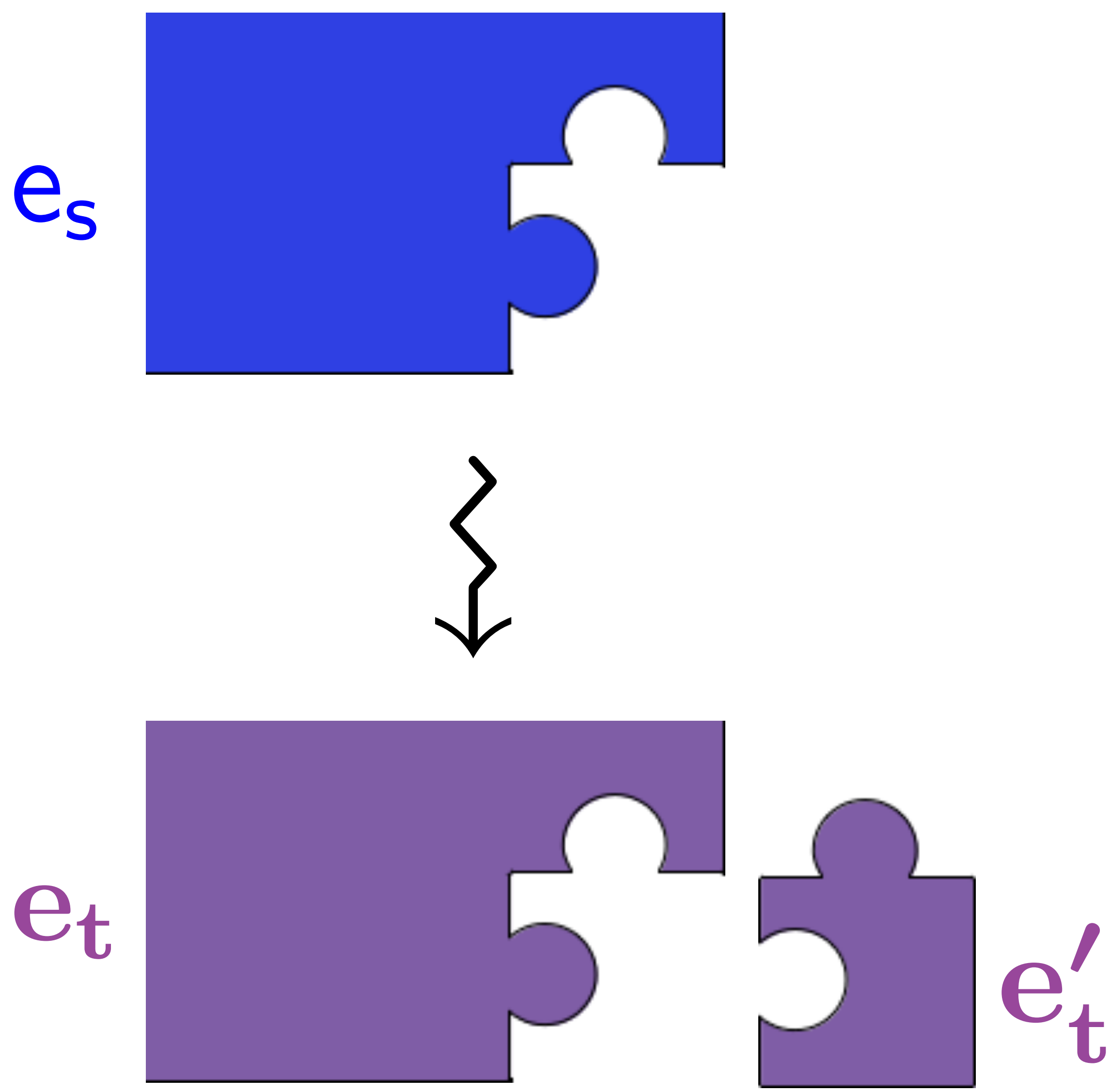
$$e_S \approx e_T$$

Compiling ML-like langs:

Logical relations

No transitivity!

Approach: Cross-Language Relations



Cross-language relation

$$e_S \approx e_T$$

Compiling ML-like langs:

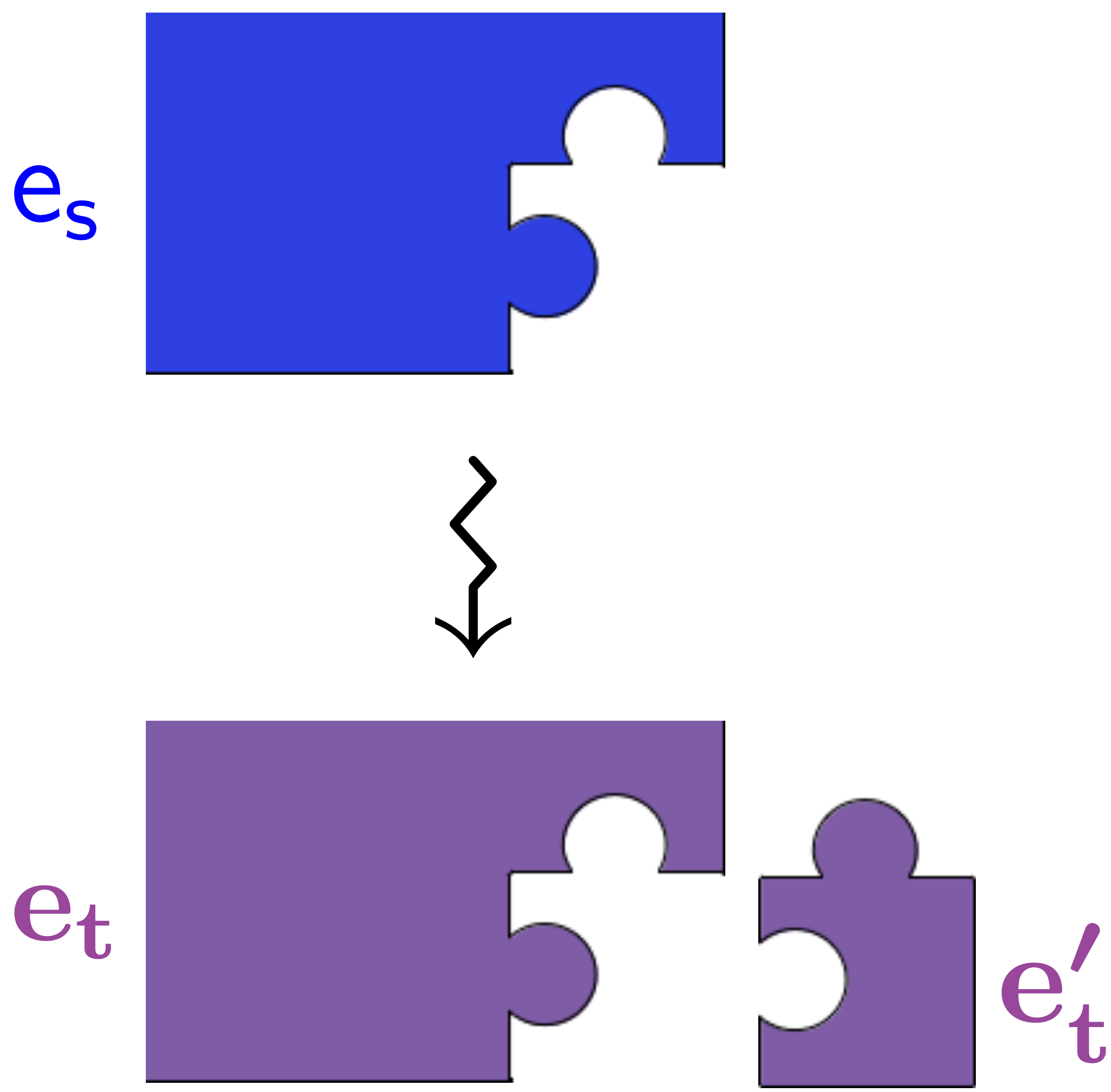
Logical relations

No transitivity!

Parametric inter-language simulations (PILS)

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

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)

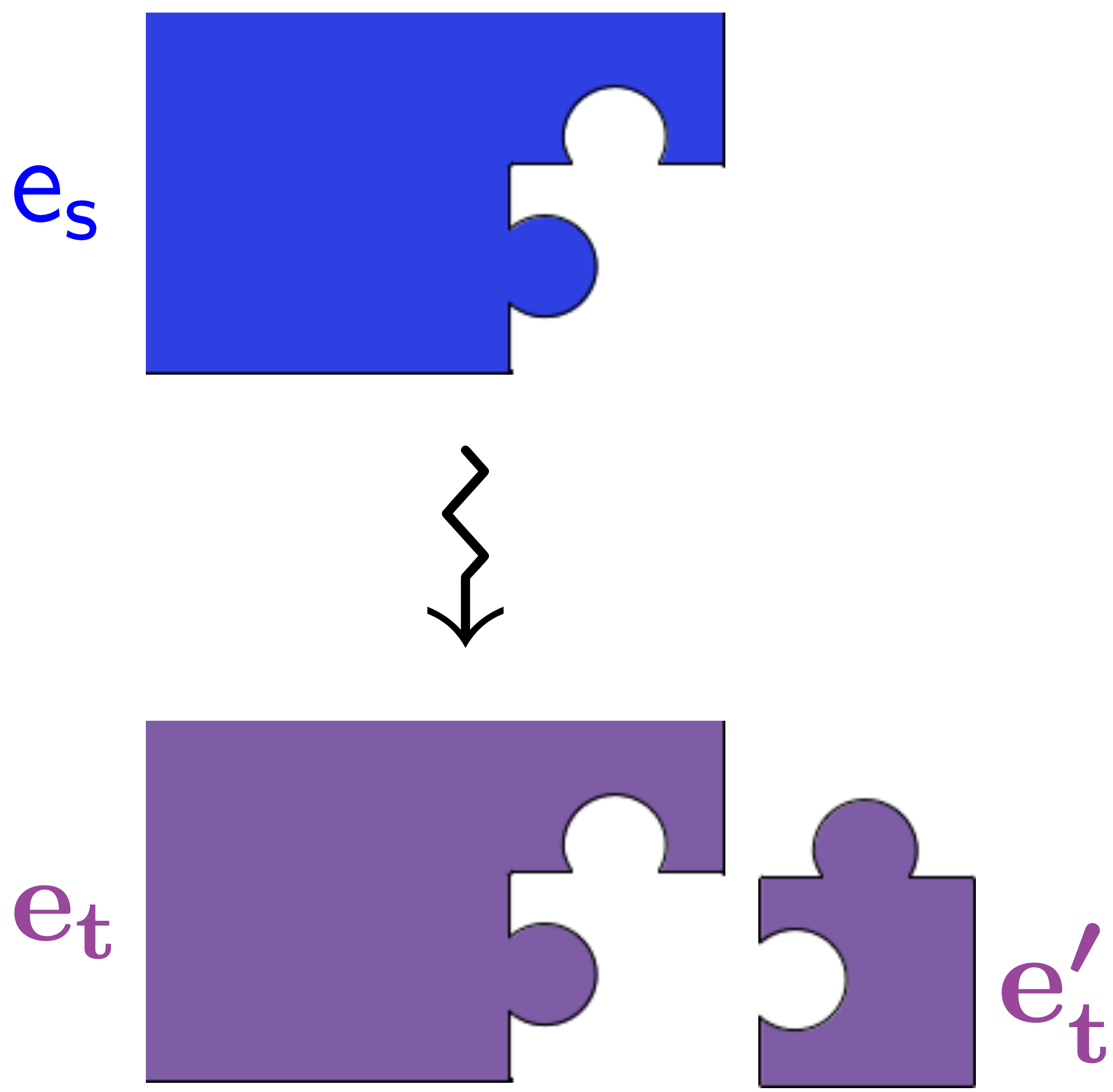
$$x : \tau' \vdash e_s : \tau \leadsto \mathbf{e_t} \quad \Longrightarrow \quad x : \tau' \vdash e_s \simeq \mathbf{e_t} : \tau$$

cross-language relation

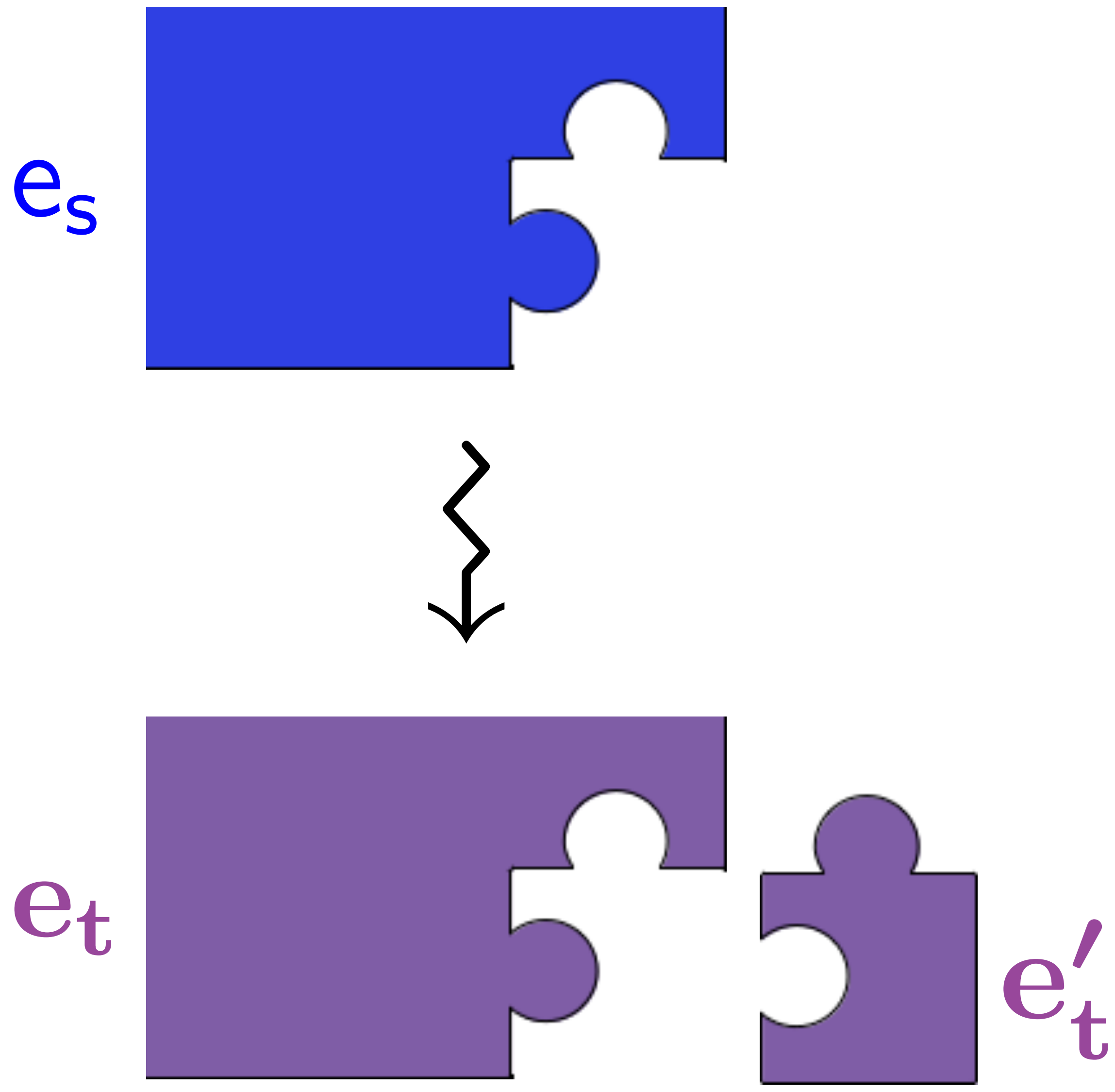
$$\forall e_s', \mathbf{e_t'}.\ \vdash e_s' \simeq \mathbf{e_t'} : \tau' \quad \Longrightarrow \quad \vdash e_s[e_s'/x] \simeq \mathbf{e_t}[\mathbf{e_t'}/\mathbf{x}] : \tau$$

Cross-Language Relation (Pilsner)

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



Cross-Language Relation (Pilsner)



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

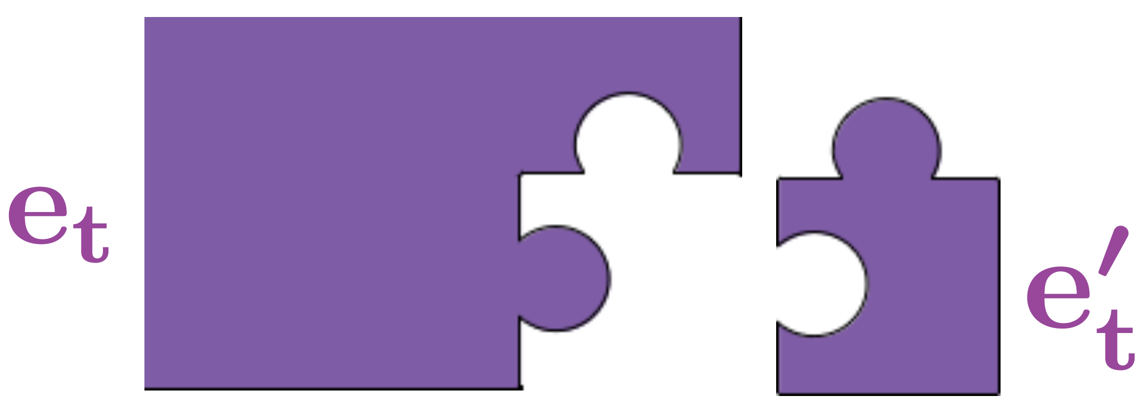
Does the compiler correctness theorem permit linking with e'_t *?*

Cross-Language Relation (Pilsner)

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

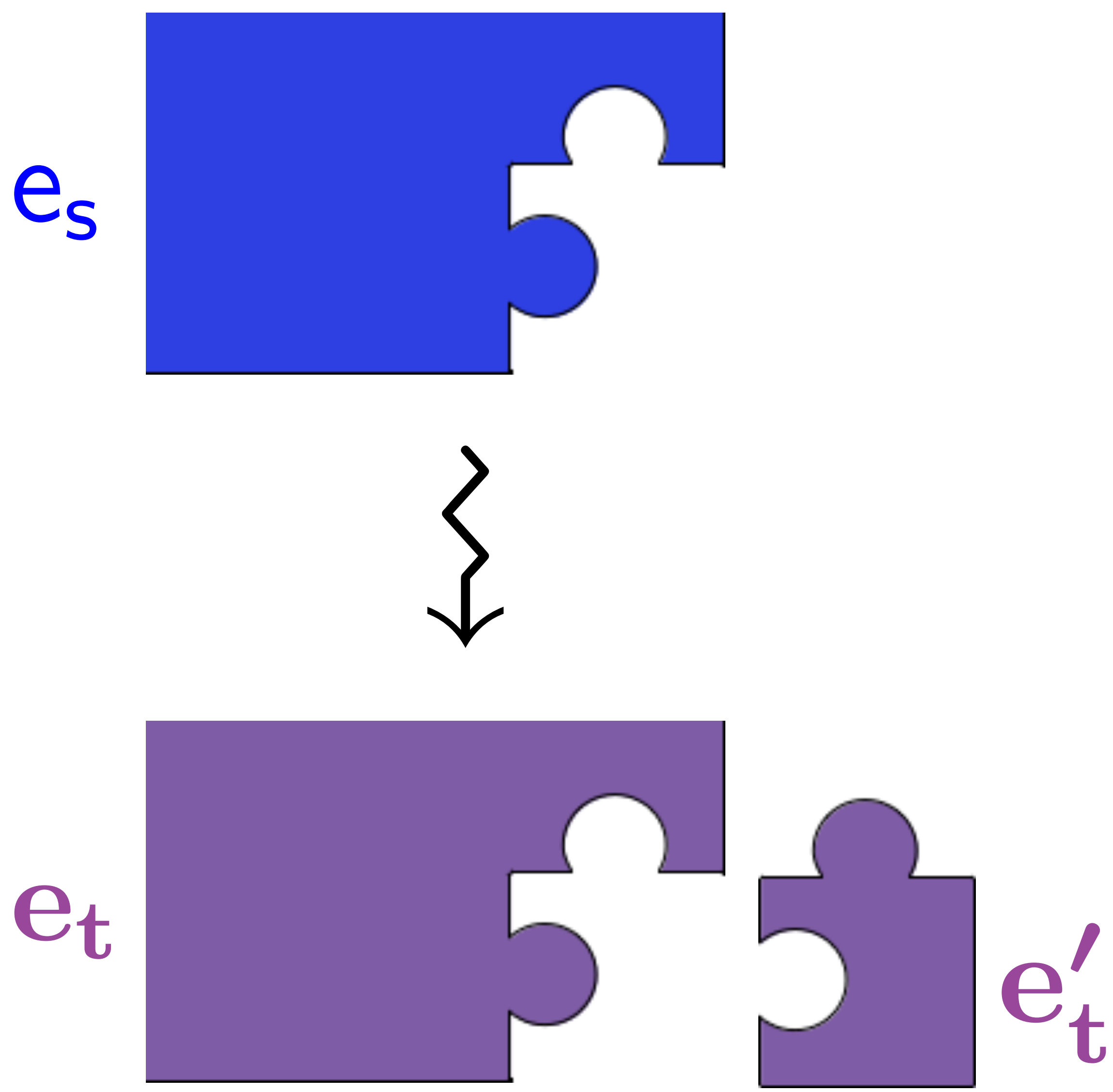
cross-language relation

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

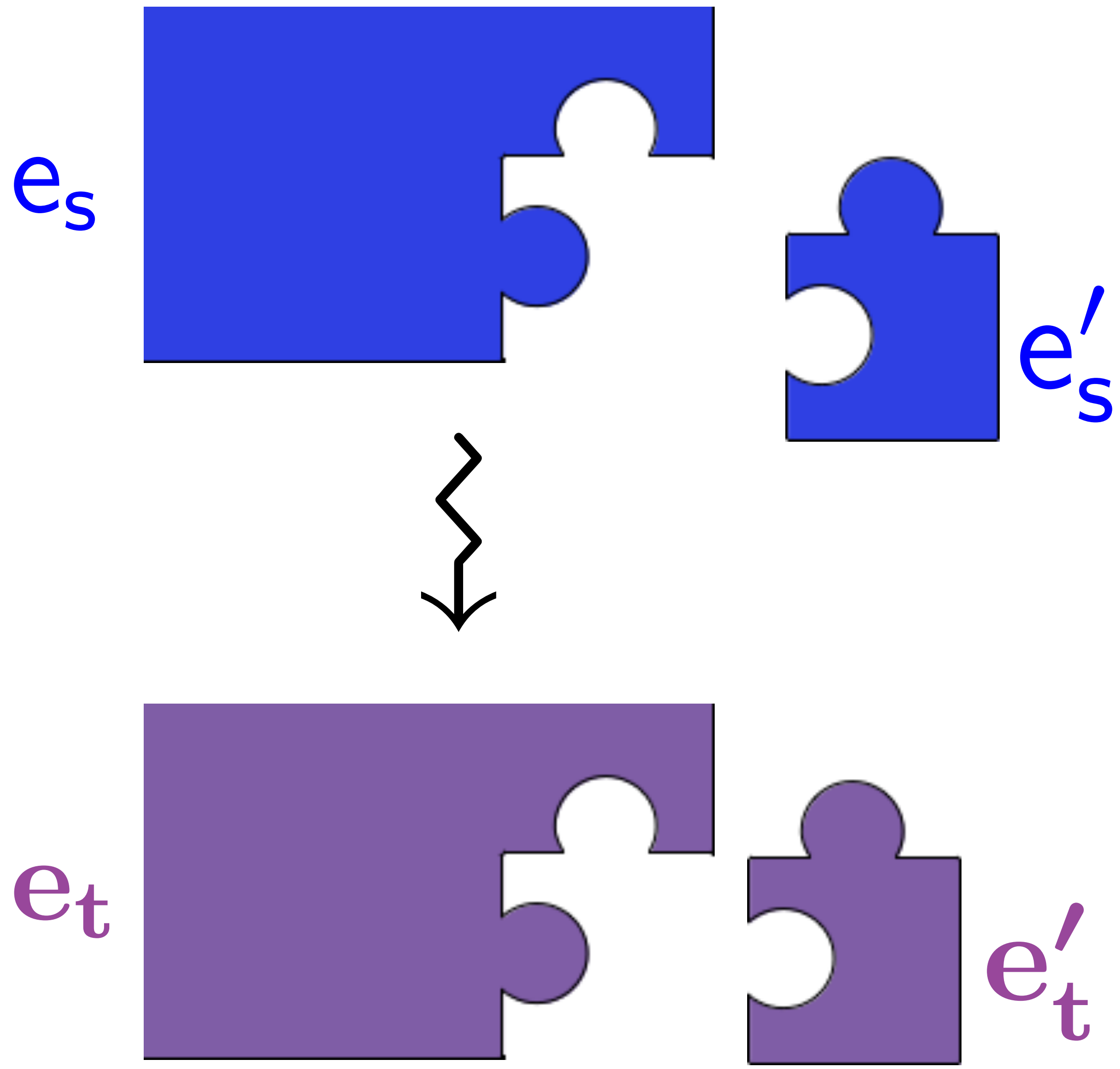


Cross-Language Relation (Pilsner)

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

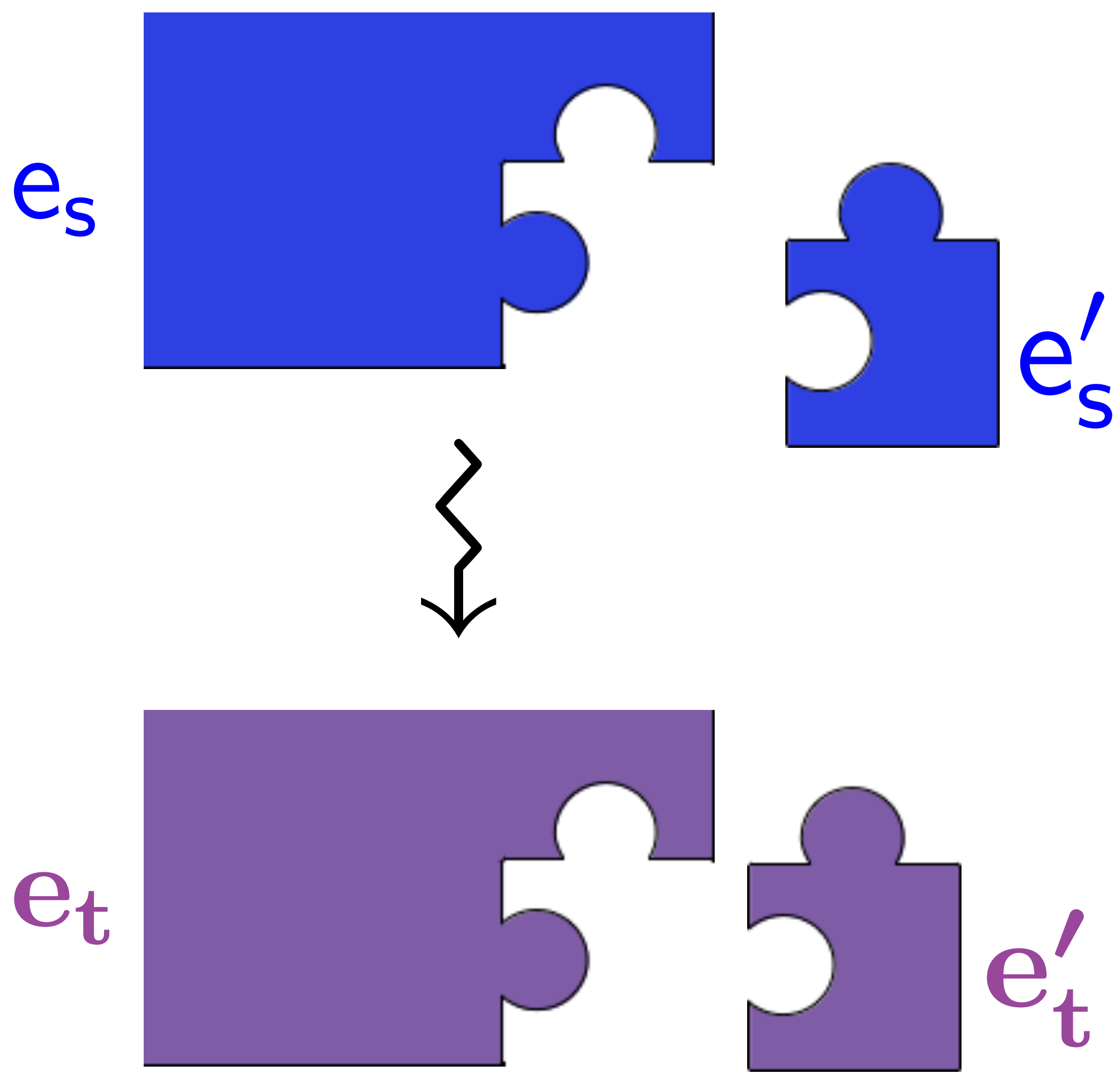


Cross-Language Relation (Pilsner)



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

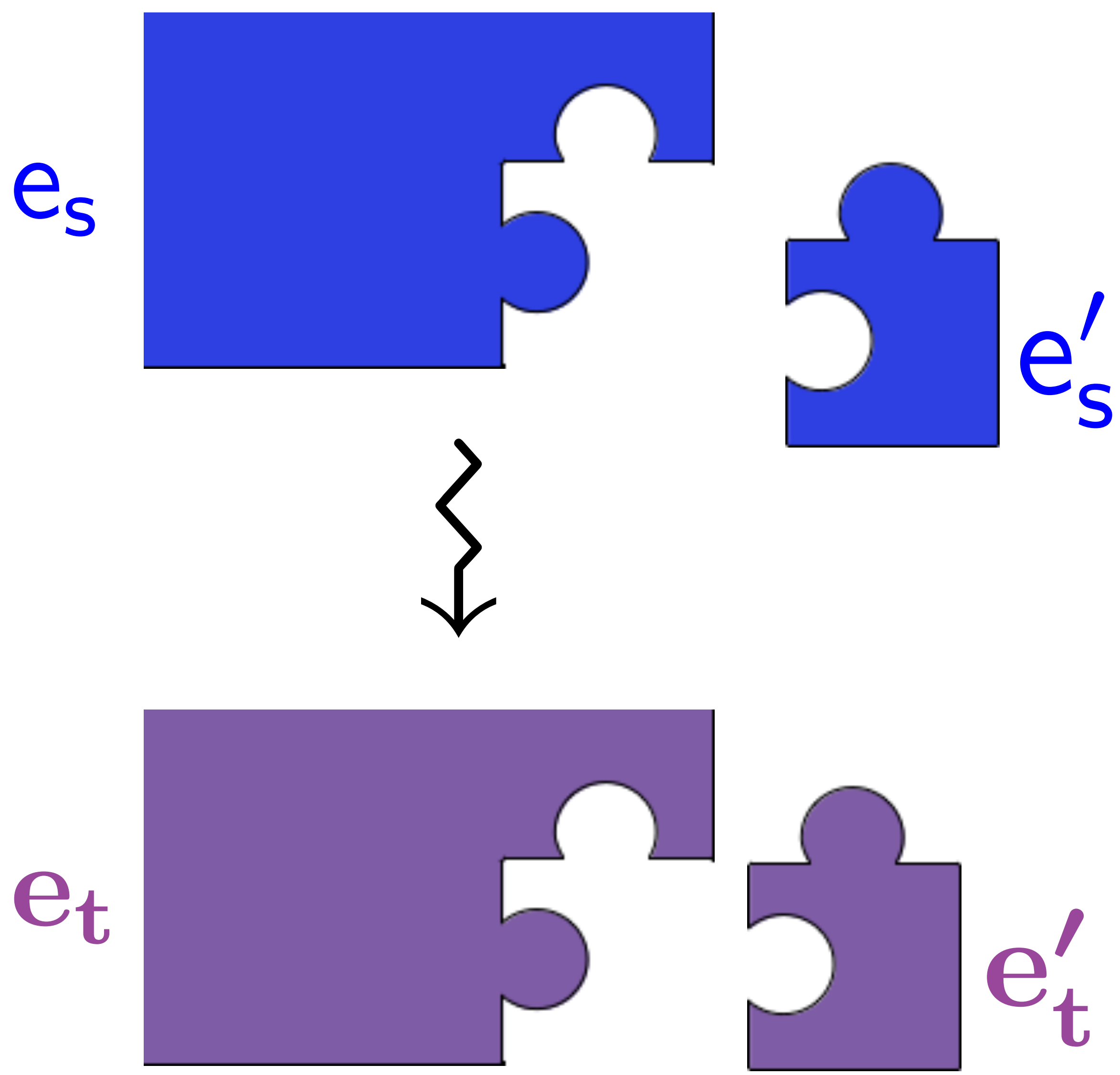
Cross-Language Relation (Pilsner)



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

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

Cross-Language Relation (Pilsner)

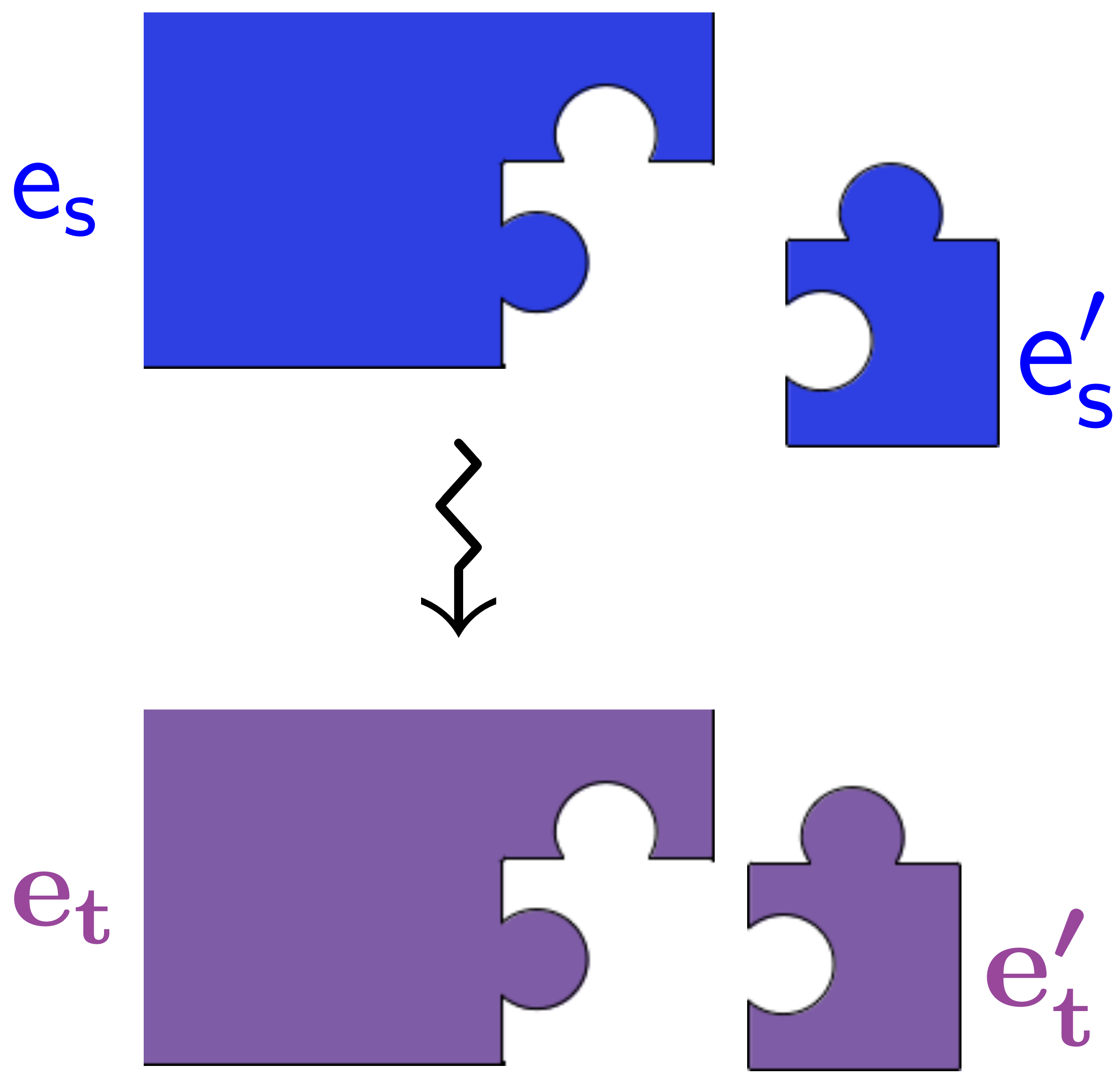


Have $x : \tau' \vdash e_s \simeq e_t : \tau$

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

$\therefore \vdash e_s[e'_s/x] \simeq e_t[e'_t/x] : \tau$

Cross-Language Relation (Pilsner)



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

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

$\therefore \vdash e_s[e'_s/x] \simeq e_t[e'_t/x] : \tau$

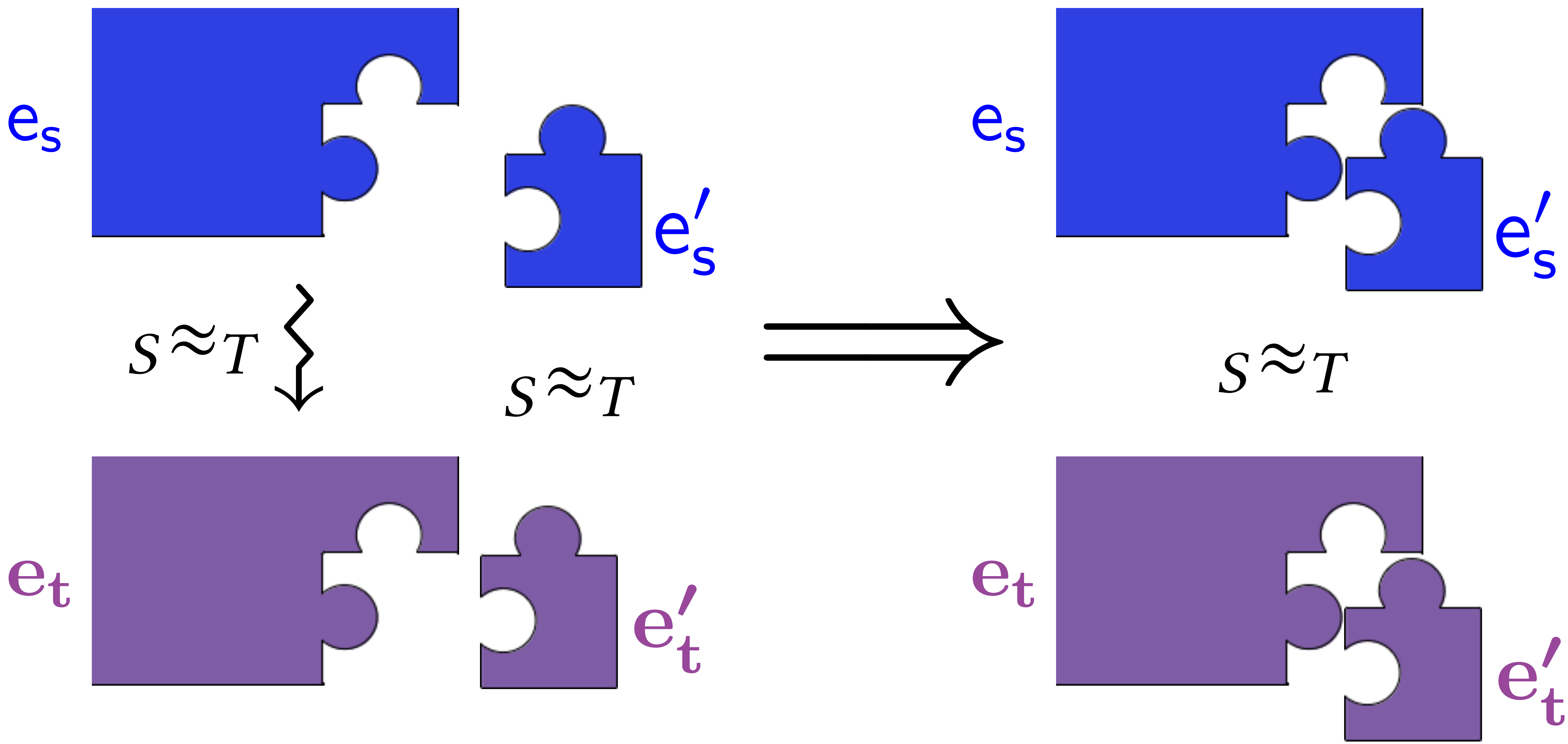
- Need to come up with e'_s -- not feasible in practice!
- Cannot link with e'_t whose behavior cannot be expressed in source.

Horizontal Compositionality

Linking

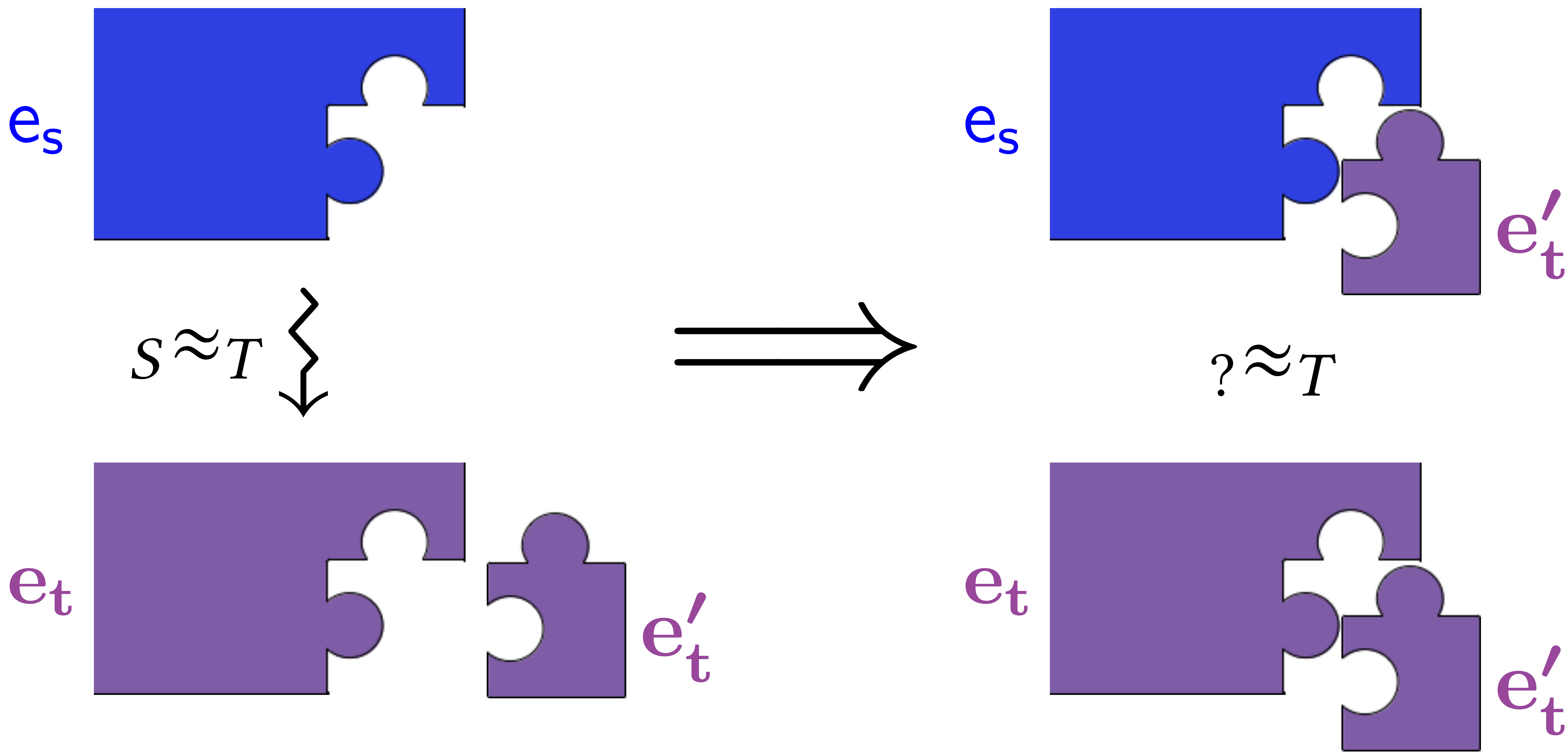
Horizontal Compositionality

Horizontal Compositionality

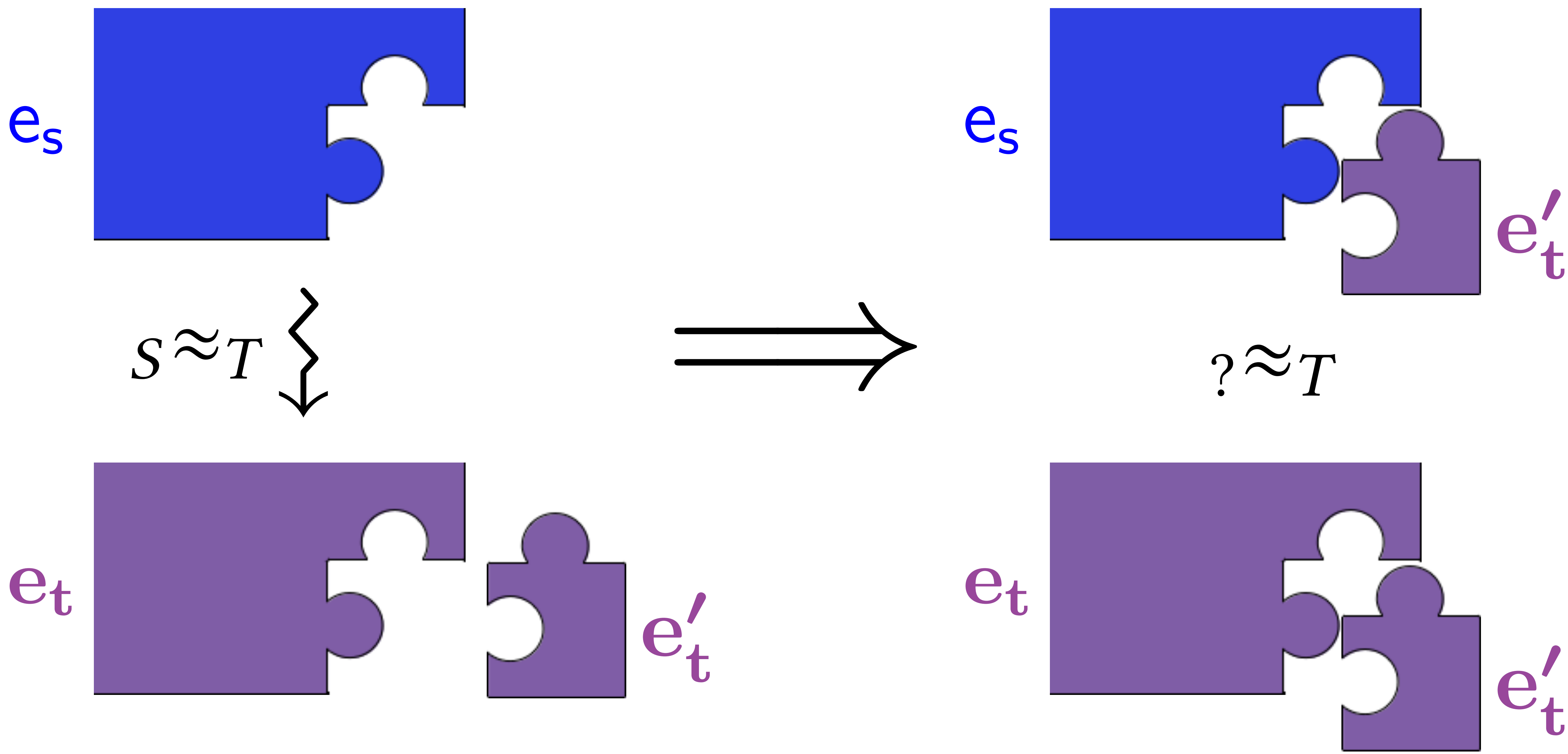


Linking

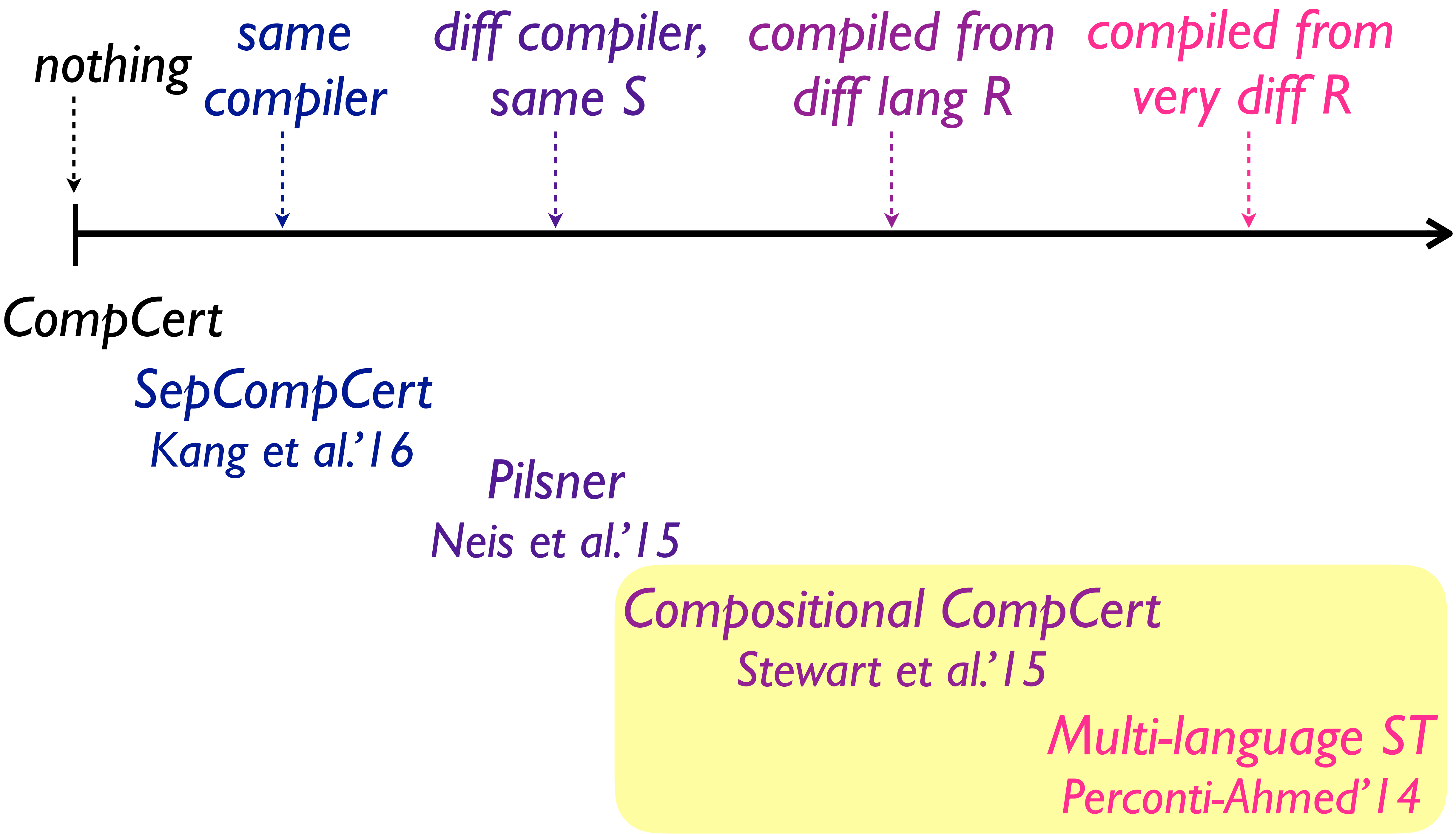
Linking



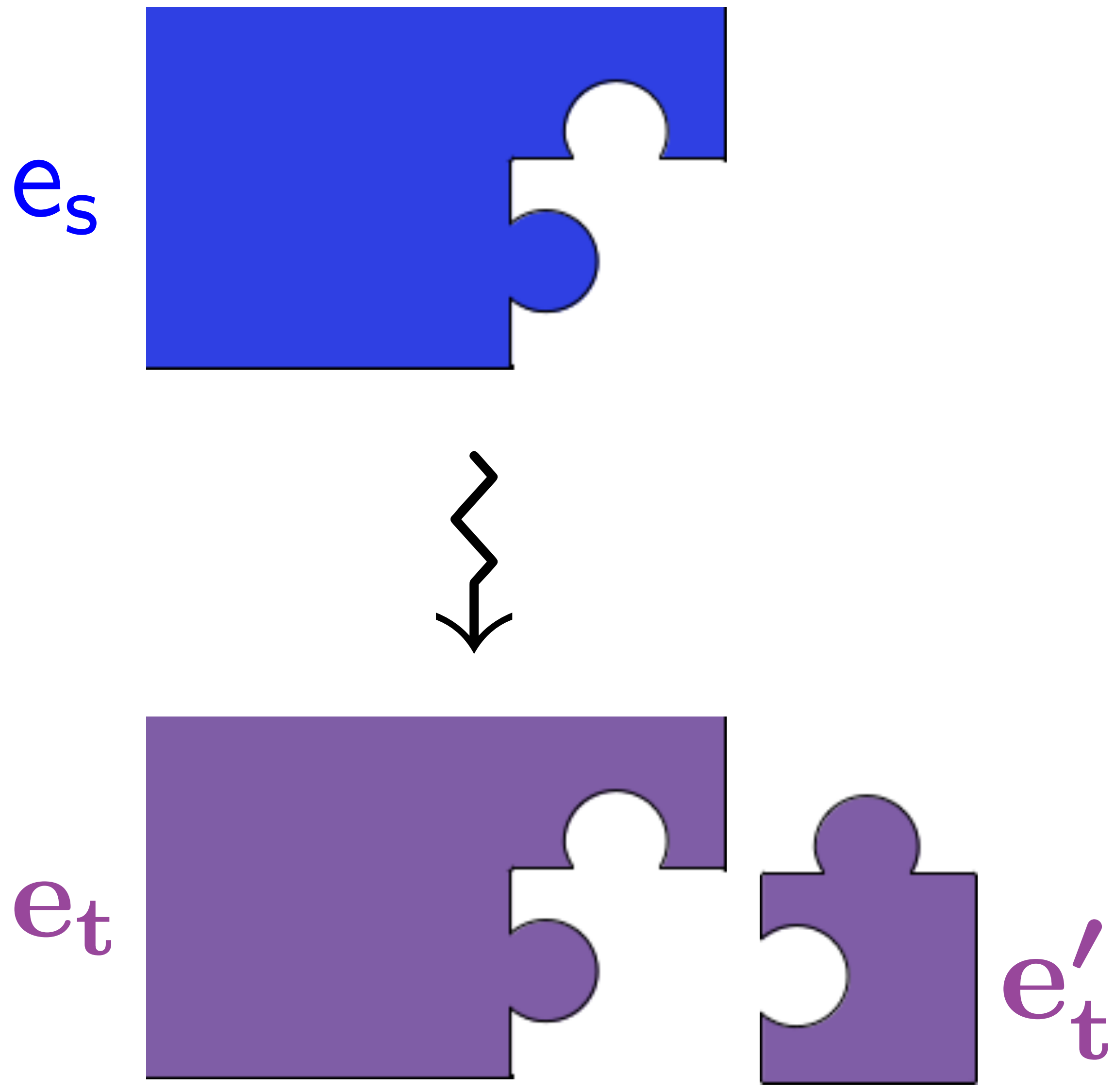
Source-Independent Linking



What we can link with



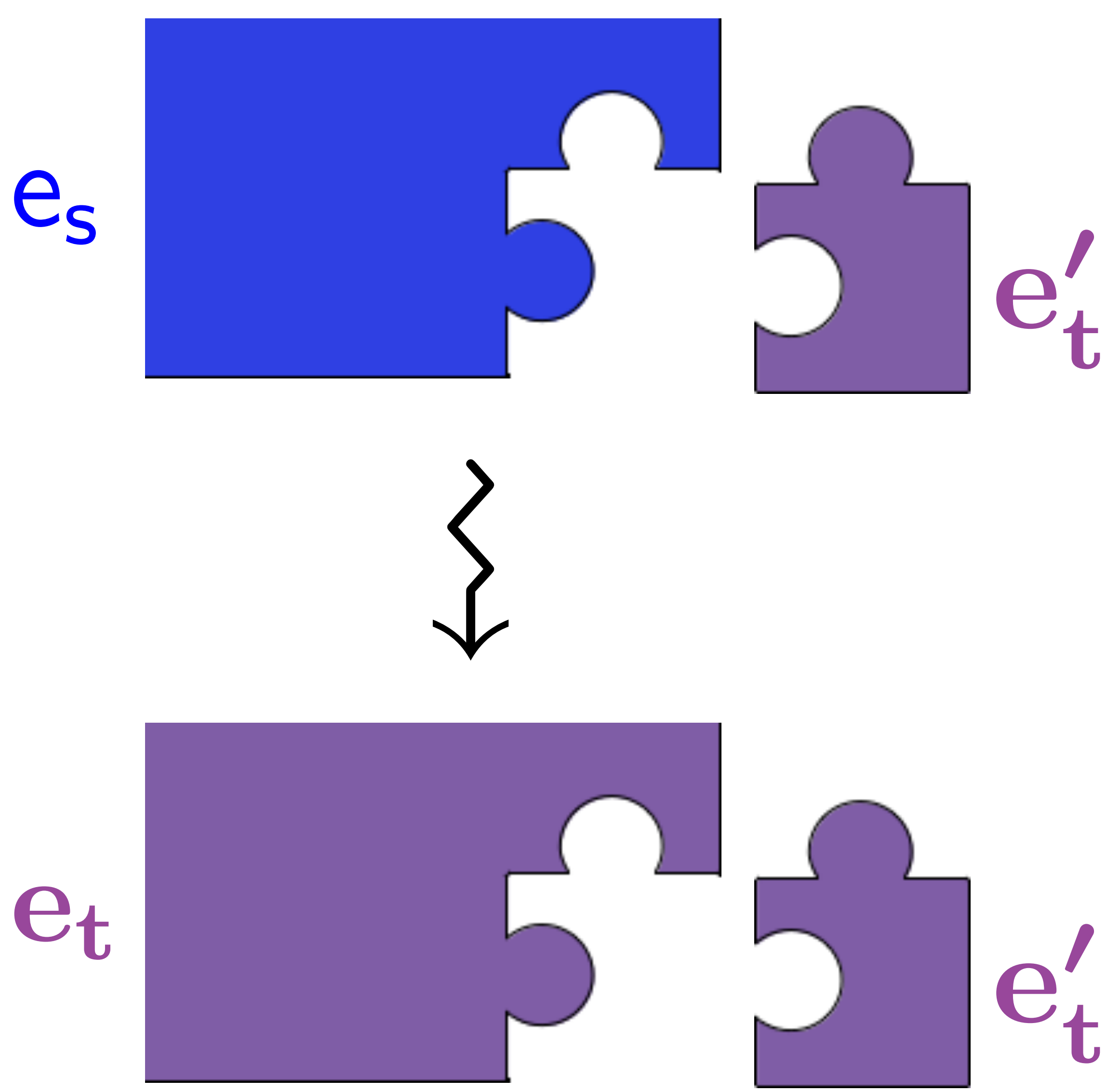
Correct Compilation of Components?



$$e_S \approx e_T$$

expressed how?

Correct Compilation of Components?

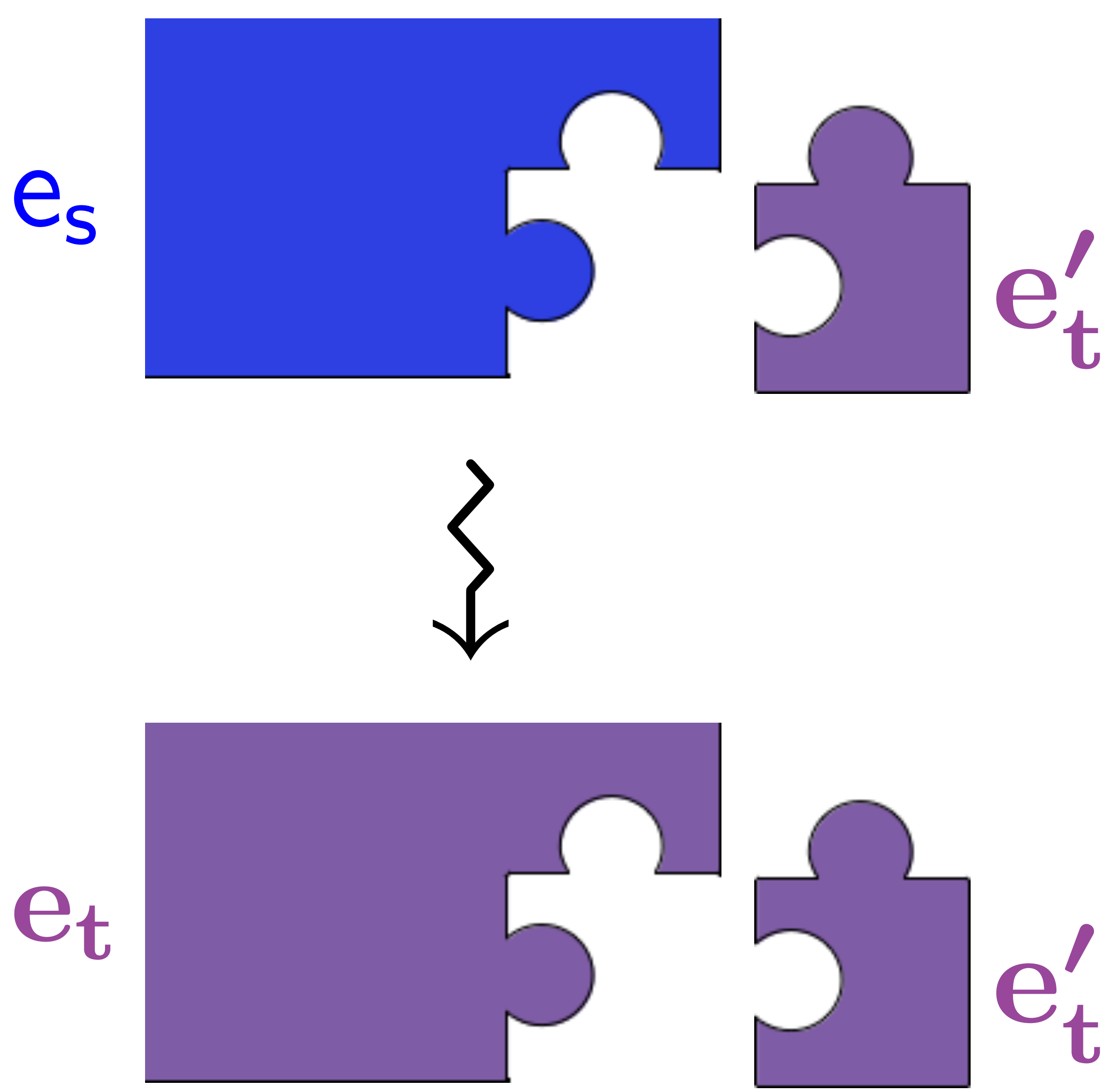


$e_S \approx e_T$

↑

expressed how?

Correct Compilation of Components?



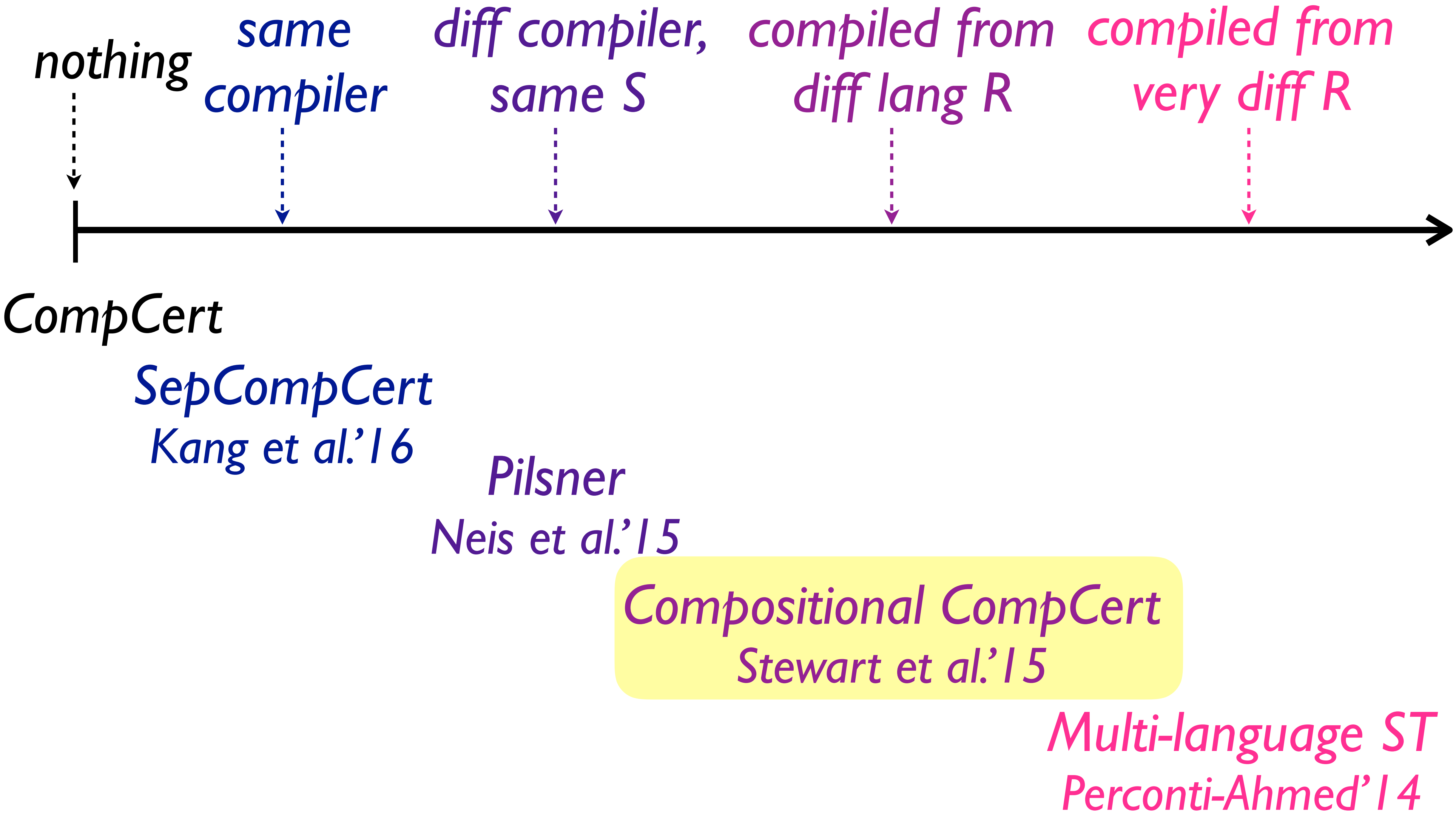
$e_S \approx e_T$

expressed how?

Need a semantics of source-target interoperability:

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

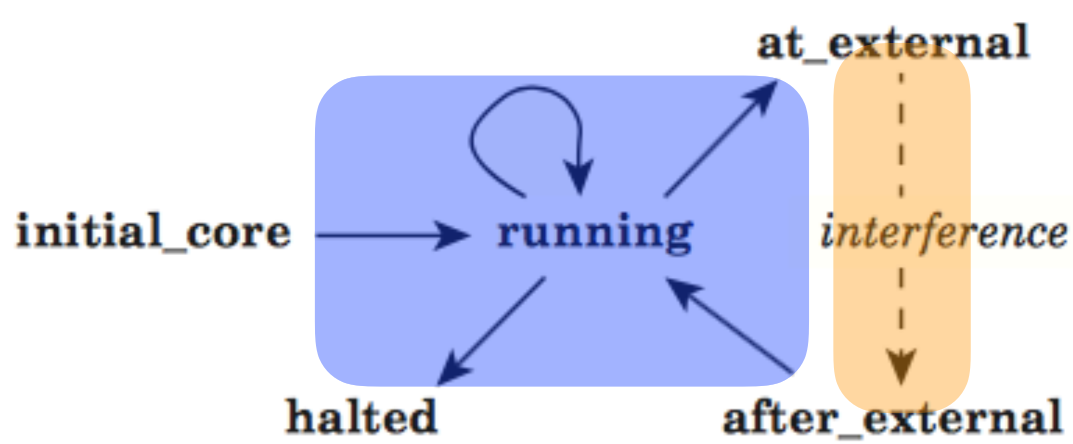
What we can link with



Approach: Interaction Semantics

Compositional CompCert *[Stewart et al. POPL'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. POPL'15]*

- Language-independent linking

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

Approach: Interaction Semantics

Compositional CompCert *[Stewart et al. POPL'15]*

- Language-independent linking

- 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

Approach: Interaction Semantics

Compositional CompCert *[Stewart et al. POPL'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

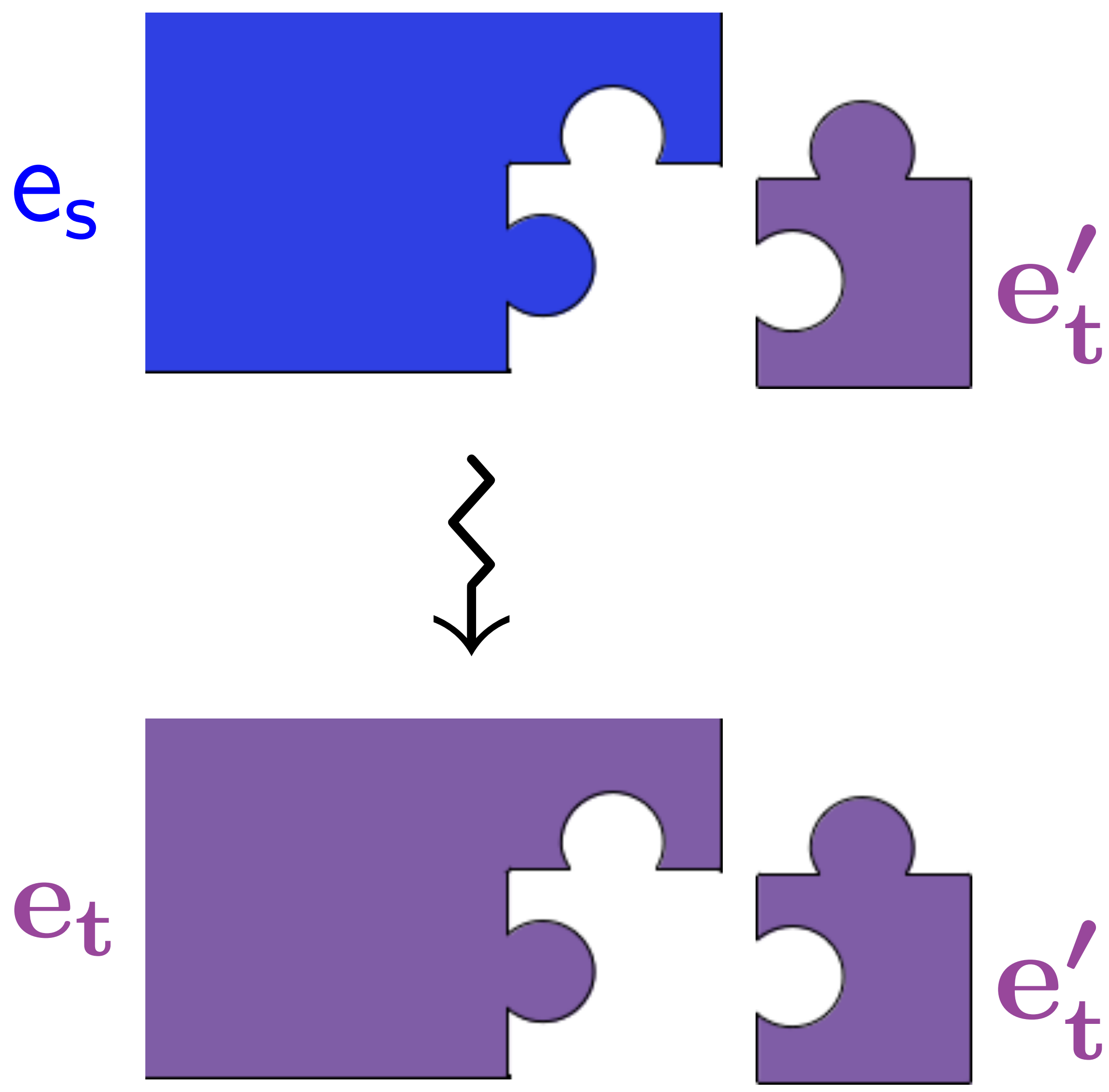
Pilsner
Neis et al.'15

Compositional CompCert
Stewart et al.'15

Multi-language ST
Perconti-Ahmed'14

Approach: Source-Target Multi-lang.

[Perconti-Ahmed ESOP'14]



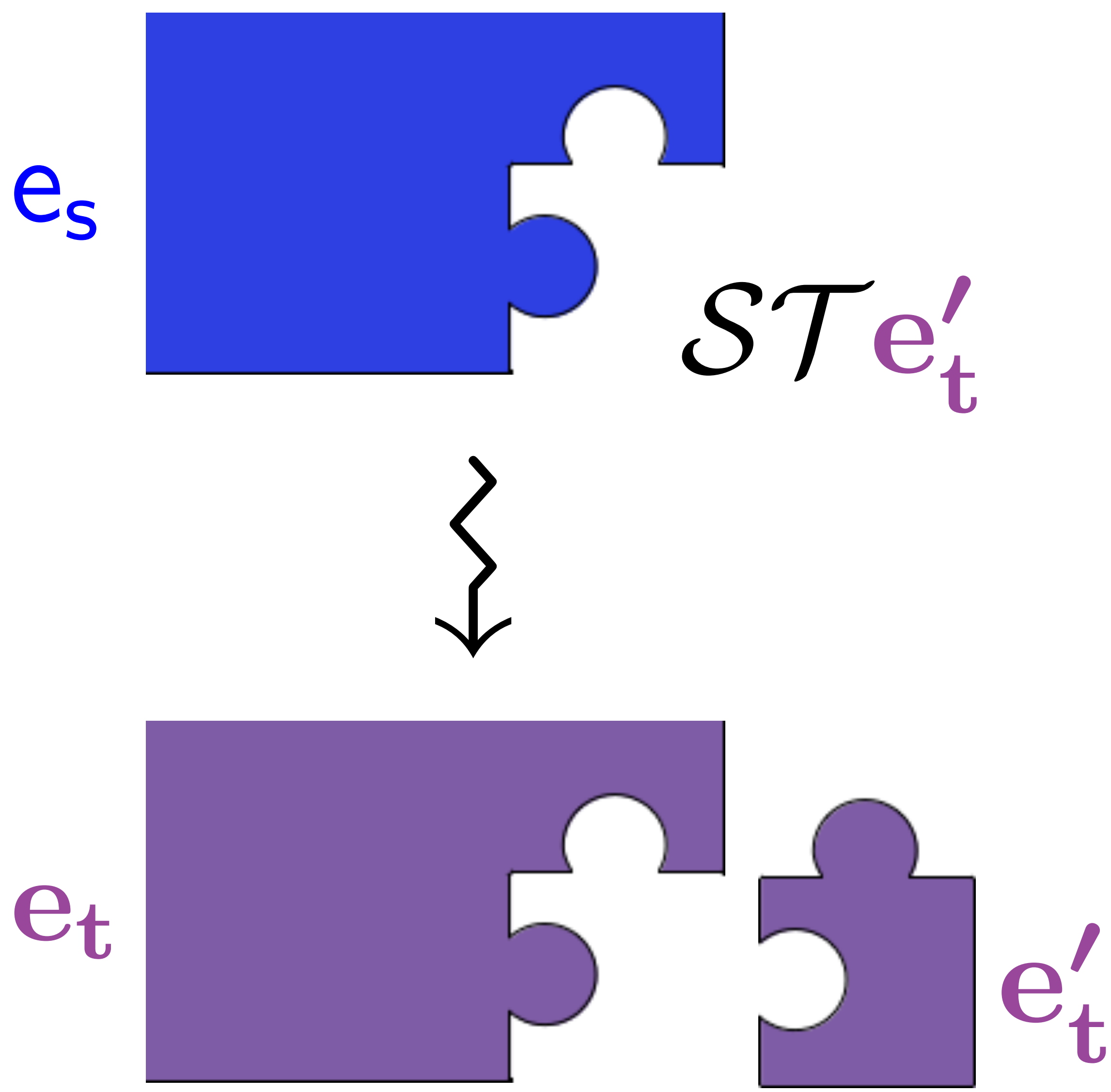
Specify semantics of source-target interoperability:

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

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

Approach: Source-Target Multi-lang.

[Perconti-Ahmed ESOP'14]



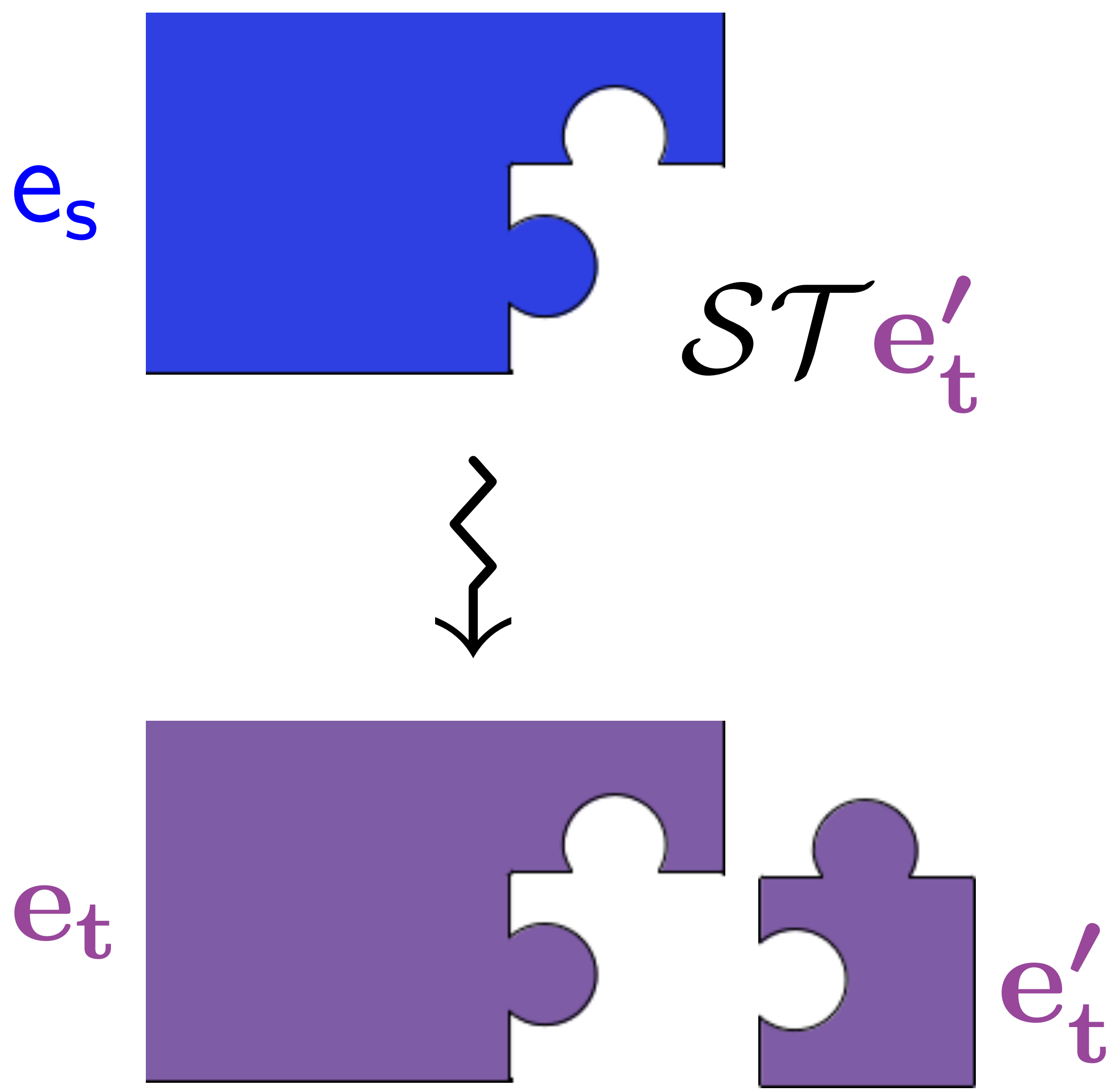
Specify semantics of source-target interoperability:

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

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

Approach: Source-Target Multi-lang.

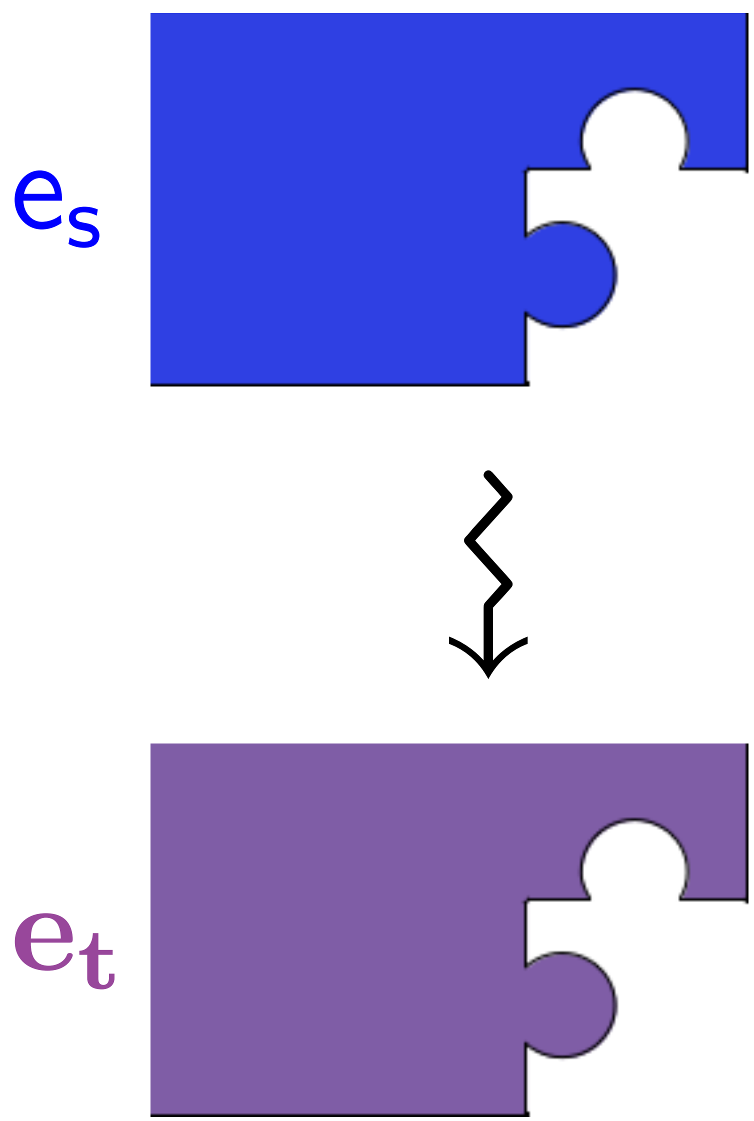
[Perconti-Ahmed ESOP'14]



$$\mathcal{TS}(e_s\ (\mathcal{ST}\,e'_t)) \approx^{ctx} e_t\ e'_t$$

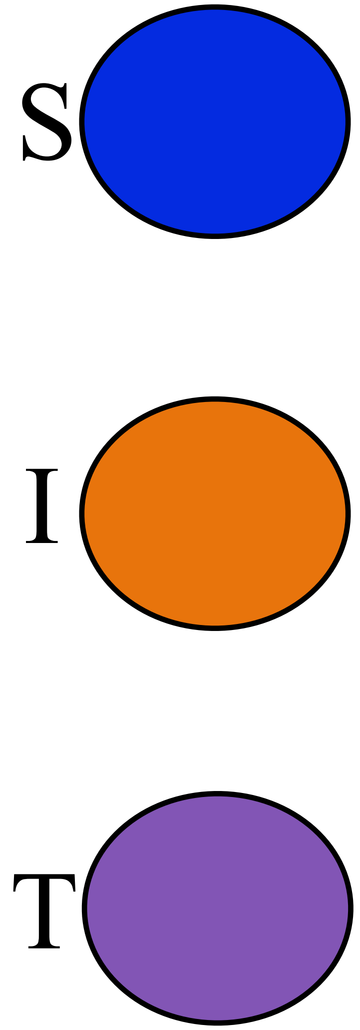
Approach: Source-Target Multi-lang.

[Perconti-Ahmed ESOP'14]

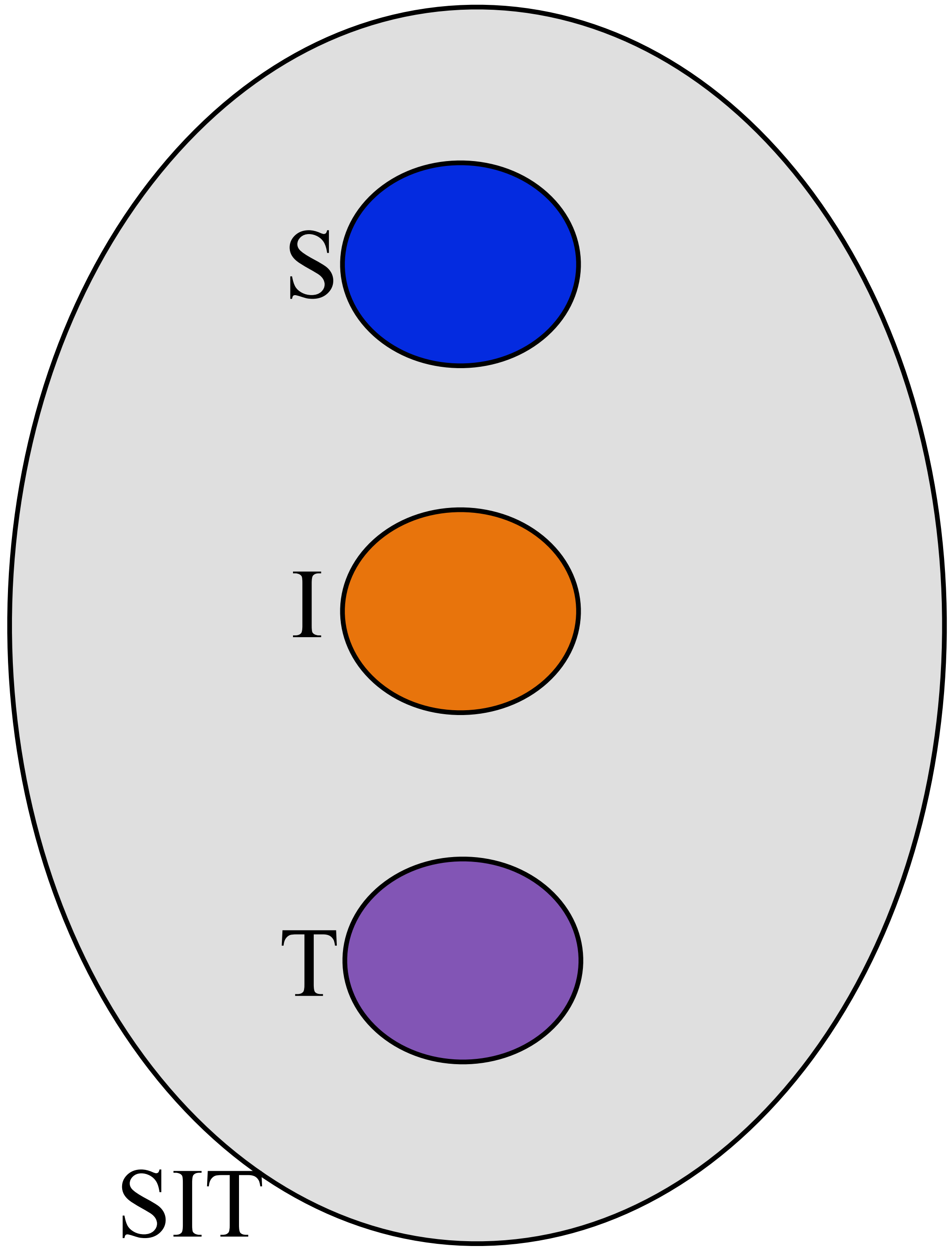


$$e_S \approx e_T \stackrel{\text{def}}{=} e_S \approx^{ctx} \mathcal{ST}\, e_T$$

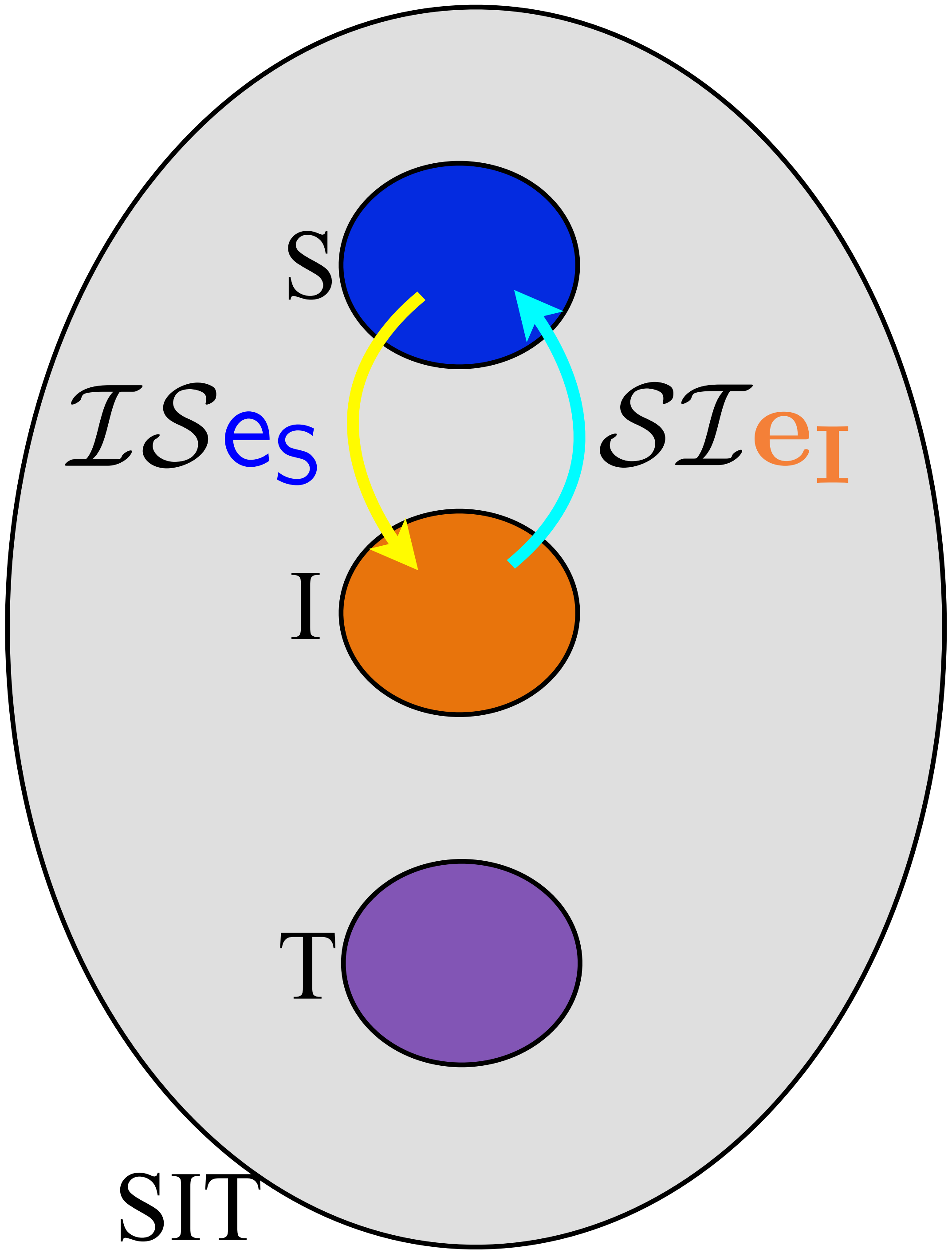
Multi-Language Semantics Approach



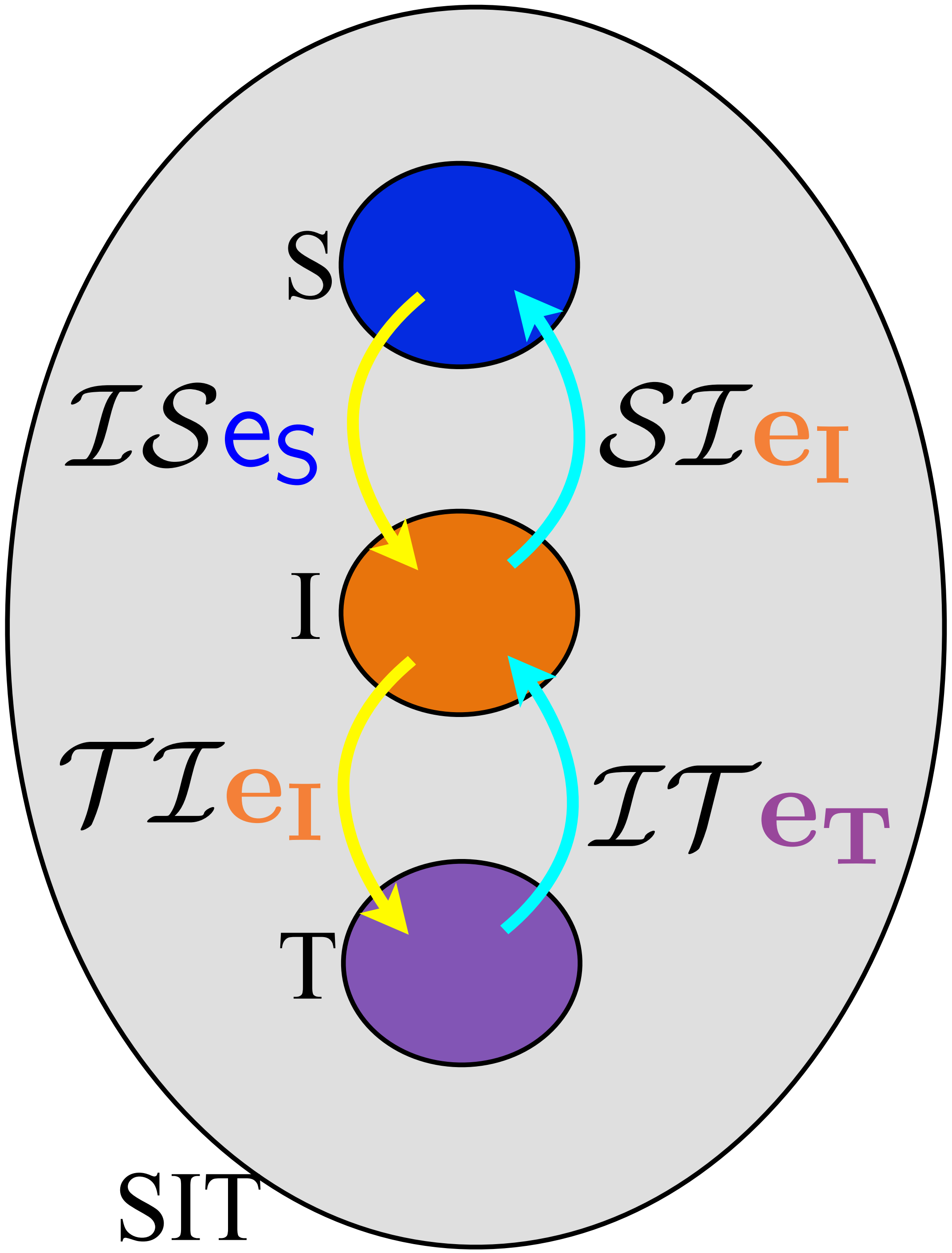
Multi-Language Semantics Approach



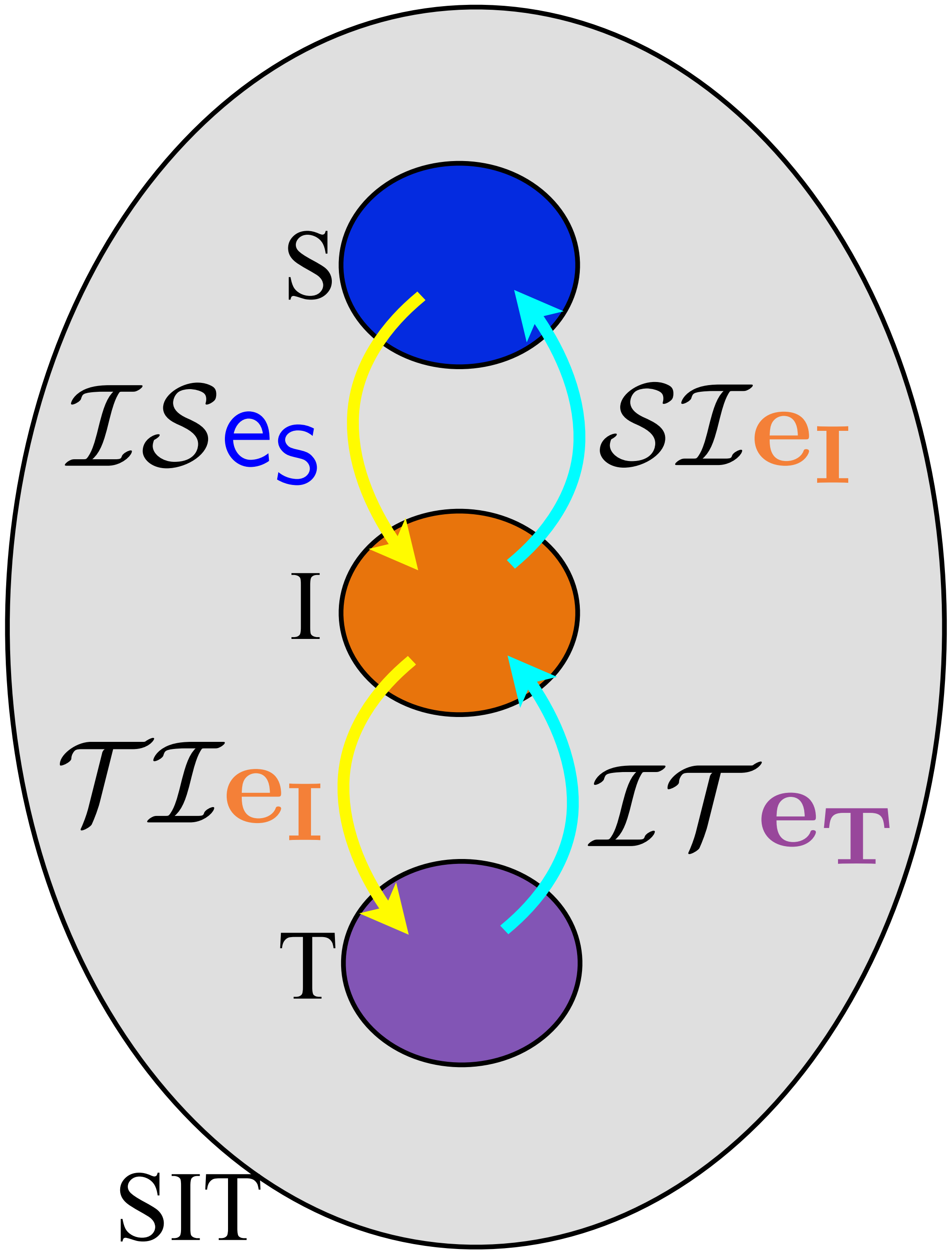
Multi-Language Semantics Approach



Multi-Language Semantics Approach

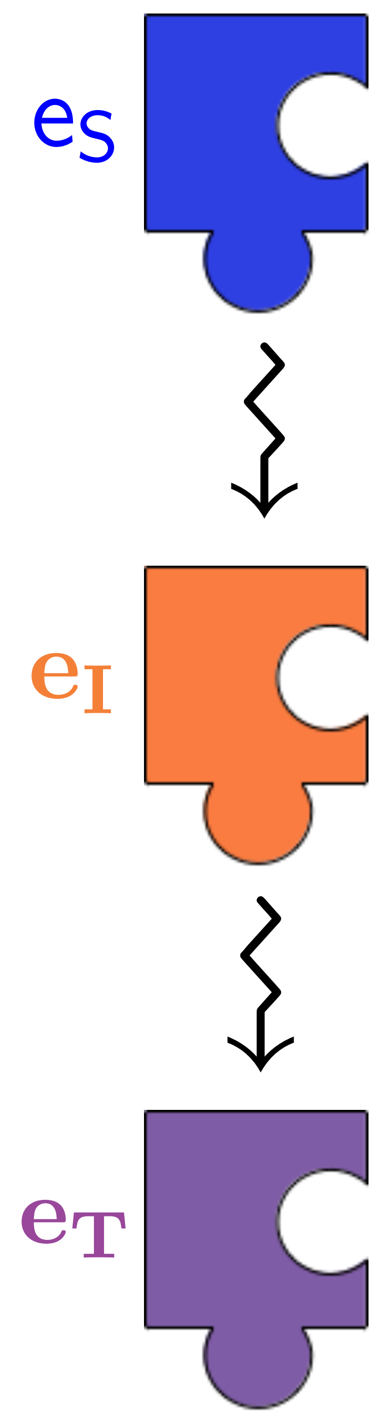


Multi-Language Semantics Approach



$\mathcal{IS}e_S$ $\mathcal{SI}e_I$ $\mathcal{TI}e_I$ $\mathcal{IT}e_T$

Compiler Correctness

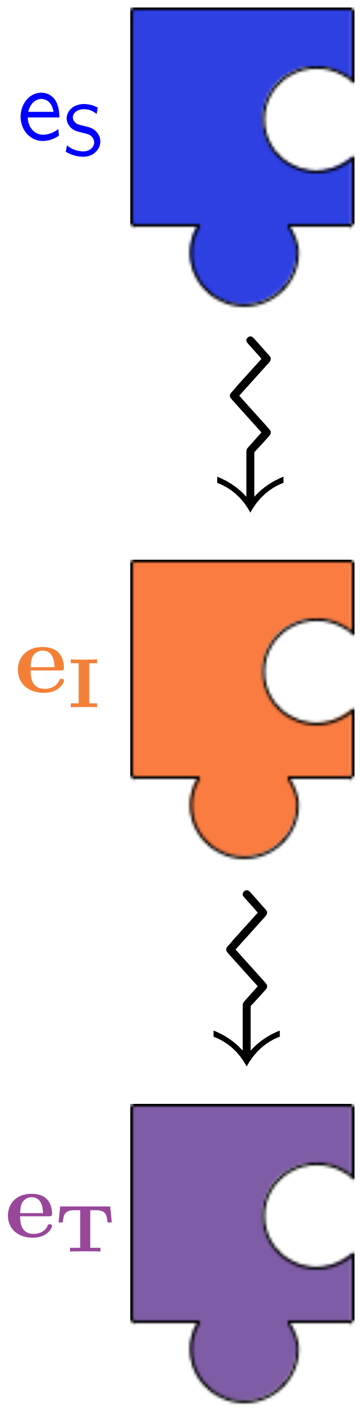


$e_S \approx^{ctx} \mathcal{SI}e_I$

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

Multi-Lang. Approach: Multi-pass ✓

Compiler Correctness

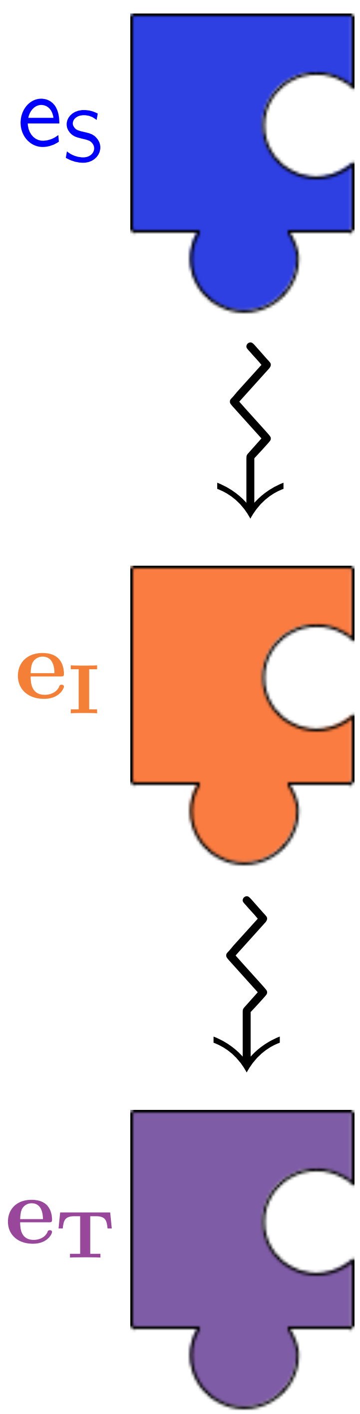


$$e_S \approx^{ctx}_{\mathcal{SI}} e_I$$

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

Multi-Lang. Approach: Multi-pass ✓

Compiler Correctness

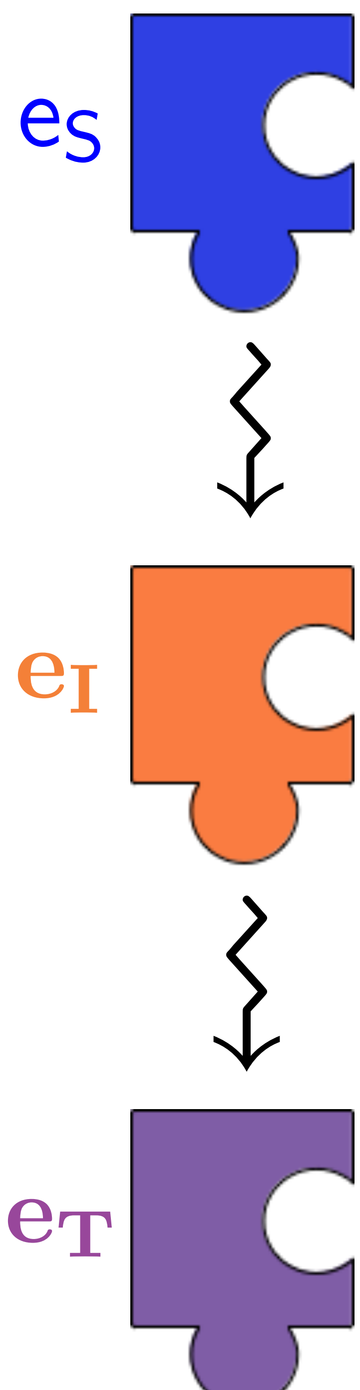


$$e_S \approx^{ctx} \mathcal{SI}\, \mathbf{e_I}$$

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

Multi-Lang. Approach: Multi-pass ✓

Compiler Correctness

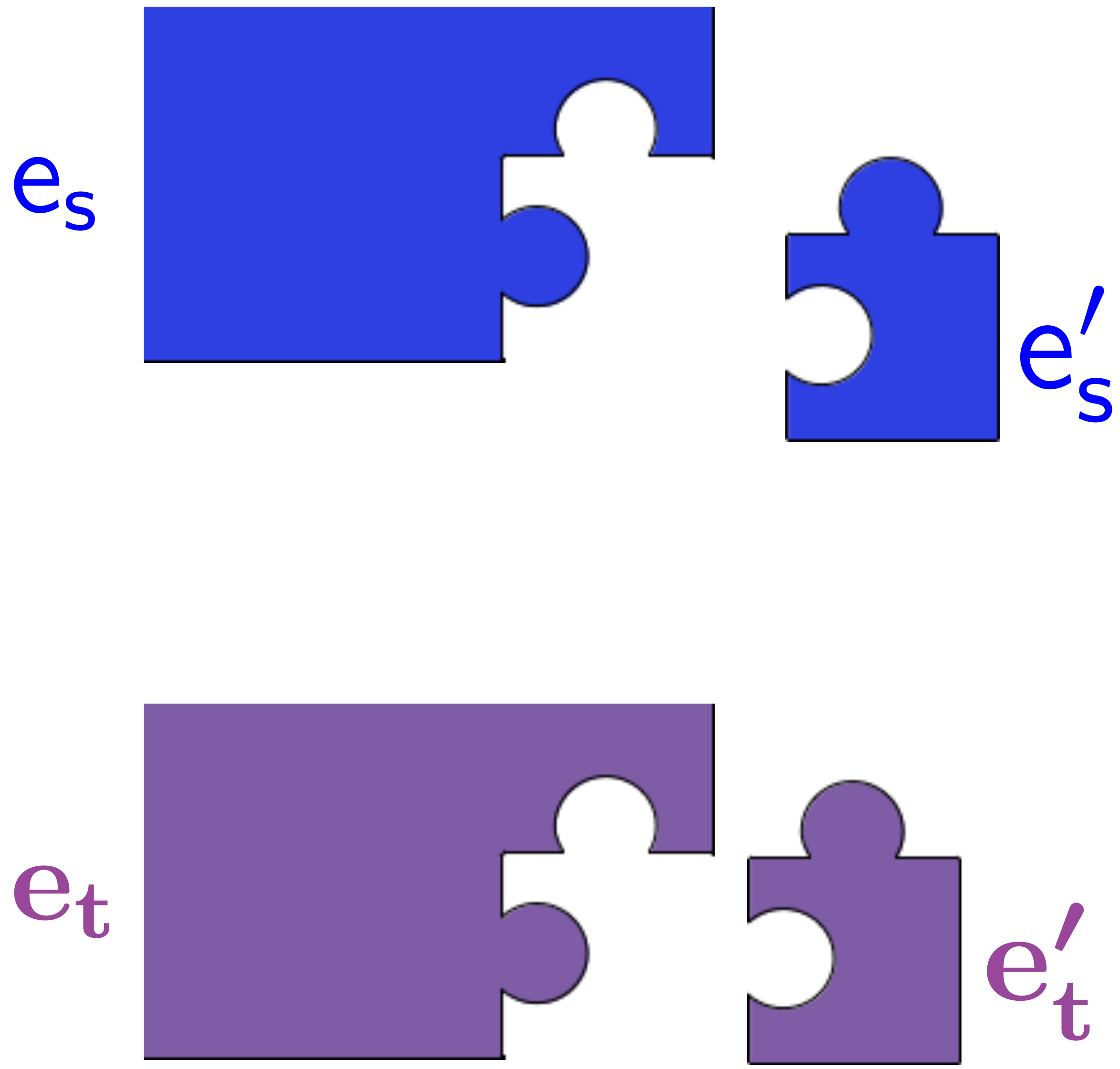


$$e_S \approx^{ctx} \mathcal{S}\mathcal{I}\, e_I$$

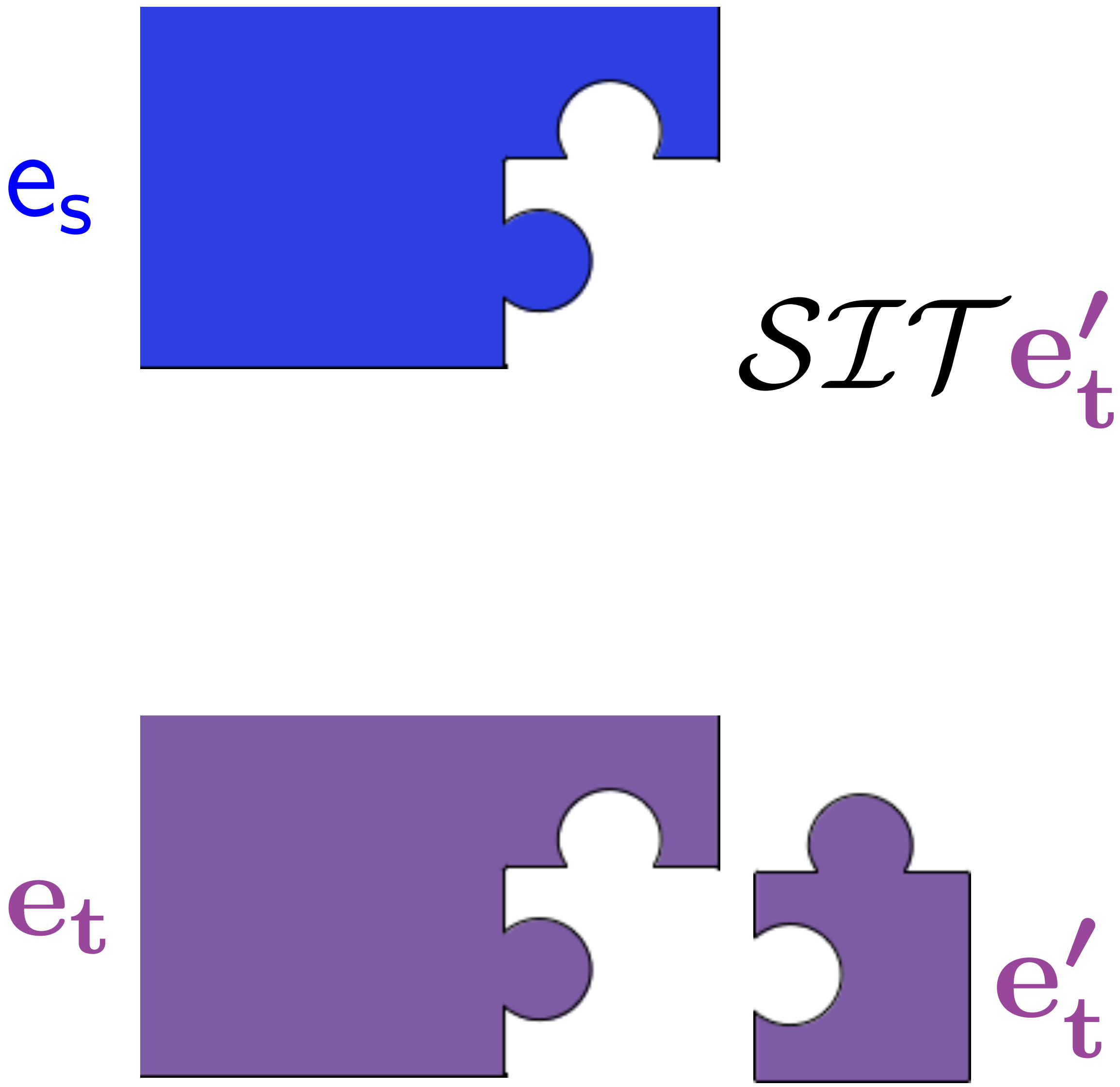
$$\mathcal{S}\mathcal{I}\, e_I \approx^{ctx} \mathcal{S}\mathcal{I}(\mathcal{I}\mathcal{T}\, e_T)$$

$$\Bigg\} \quad e_S \approx^{ctx} \mathcal{S}\mathcal{I}\mathcal{T}\, e_T$$

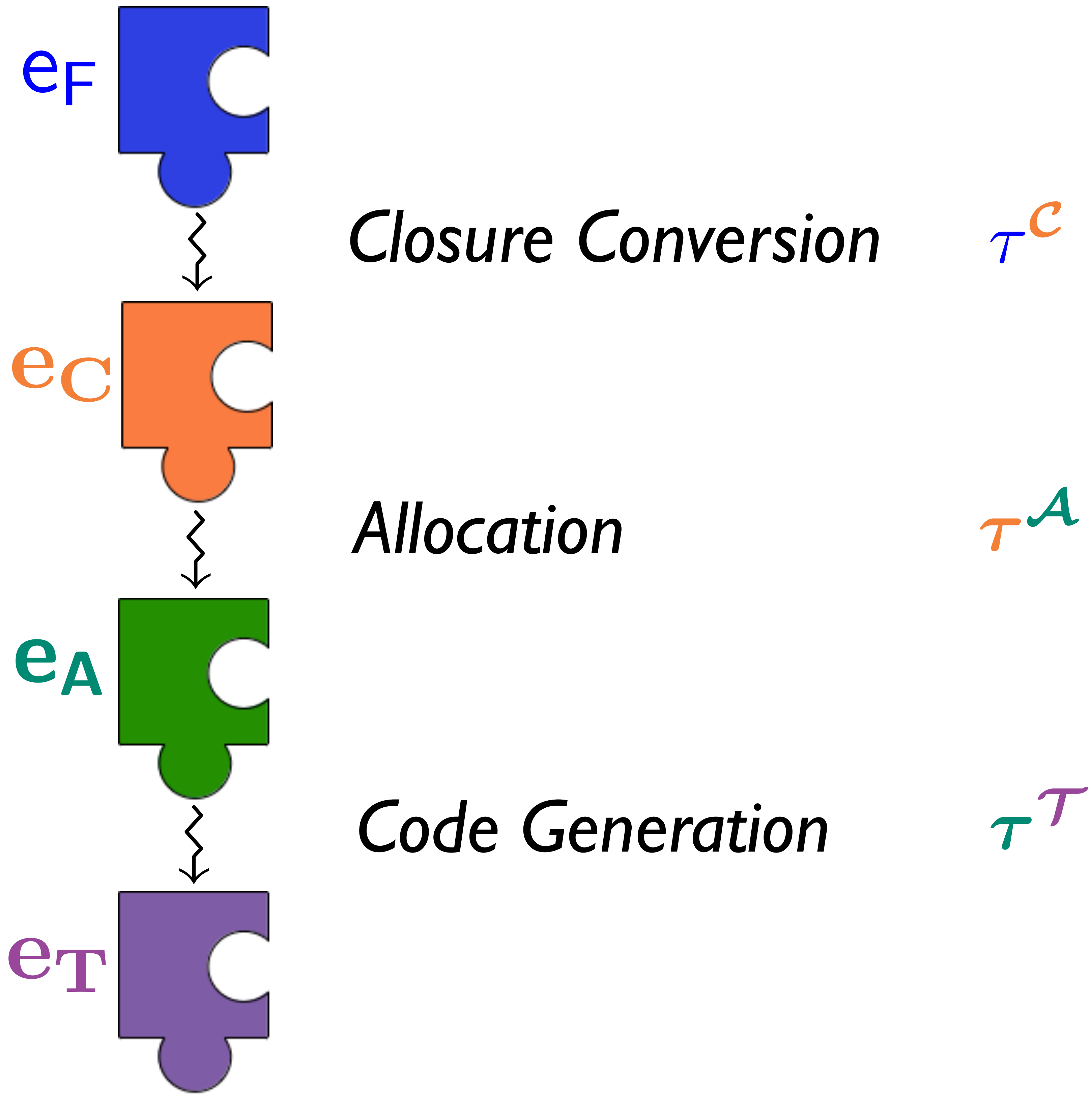
Multi-Lang. Approach: Linking ✓



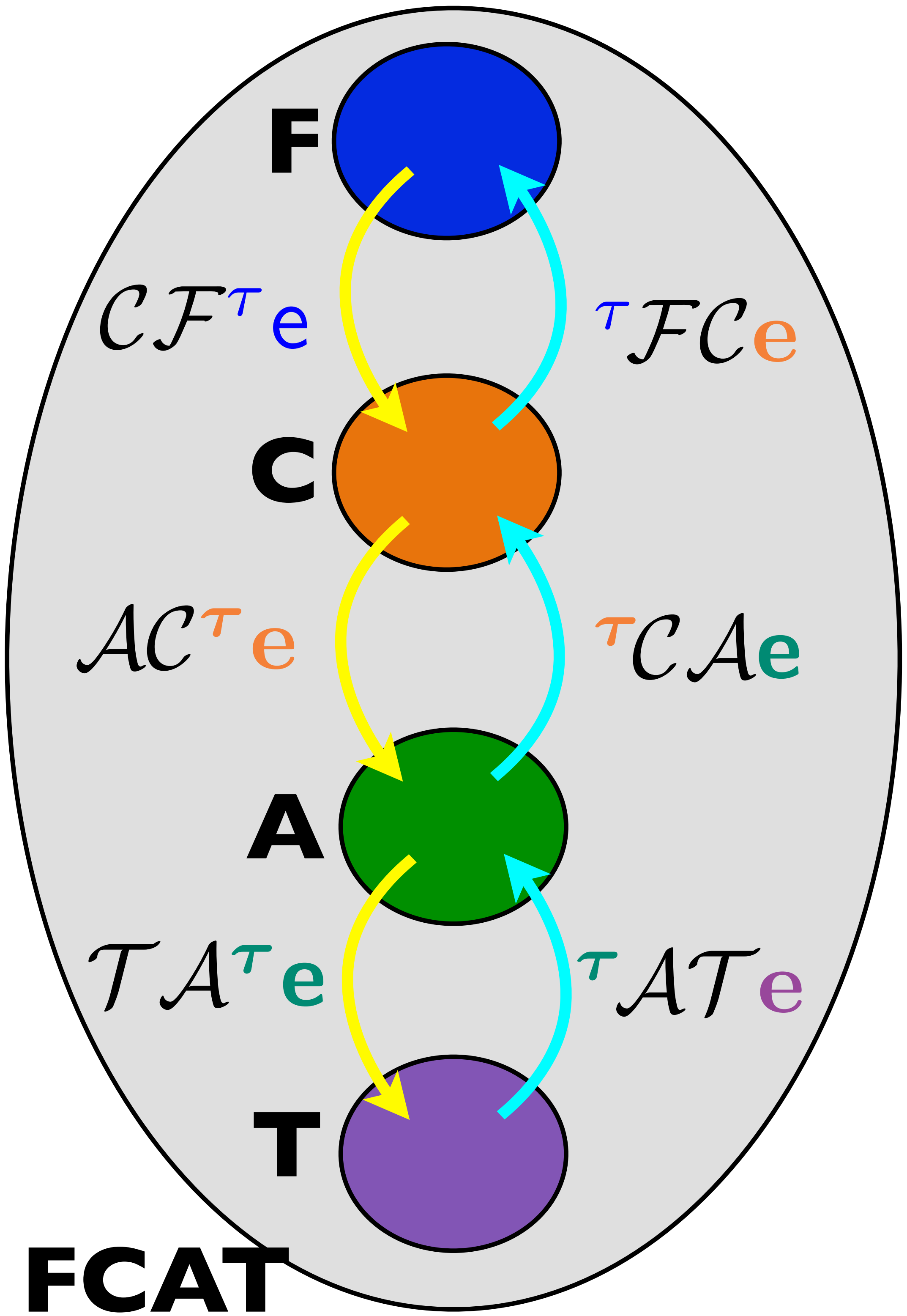
Multi-Lang. Approach: Linking ✓



Compiler Correctness: F to TAL



Combined language **FCAT**



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

- Boundaries mediate between

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

- Operational semantics

  $$\mathcal{CF}^{\tau}\mathrm{e} \longmapsto^{*} \mathcal{CF}^{\tau}\mathrm{v} \longmapsto \mathbf{v}$$

  $${}^{\tau}\mathcal{FC}\mathbf{e} \longmapsto^{*} {}^{\tau}\mathcal{FC}\mathbf{v} \longmapsto \mathrm{v}$$

- Boundary cancellation

  $${}^{\tau}\mathcal{FC}\mathcal{CF}^{\tau}\mathrm{e} \approx^{ctx} \mathrm{e} : \tau$$

  $$\mathcal{CF}^{\tau}\,{}^{\tau}\mathcal{FC}\mathbf{e} \approx^{ctx} \mathbf{e} : \tau^{\mathcal{C}}$$

Interoperability: **F** and **C**

$$\mathcal{CF}^{\mathsf{int}}(\mathsf{n}) \longmapsto \mathbf{n}$$

$${}^{\mathsf{int}}\mathcal{FC}(\mathbf{n}) \longmapsto \mathsf{n}$$

Interoperability: **F** and **C**

$$(\tau \to \tau')^{\mathcal{C}} = \exists \beta. \langle ((\beta, \tau^{\mathcal{C}}) \to \tau'^{\mathcal{C}}), \beta \rangle$$

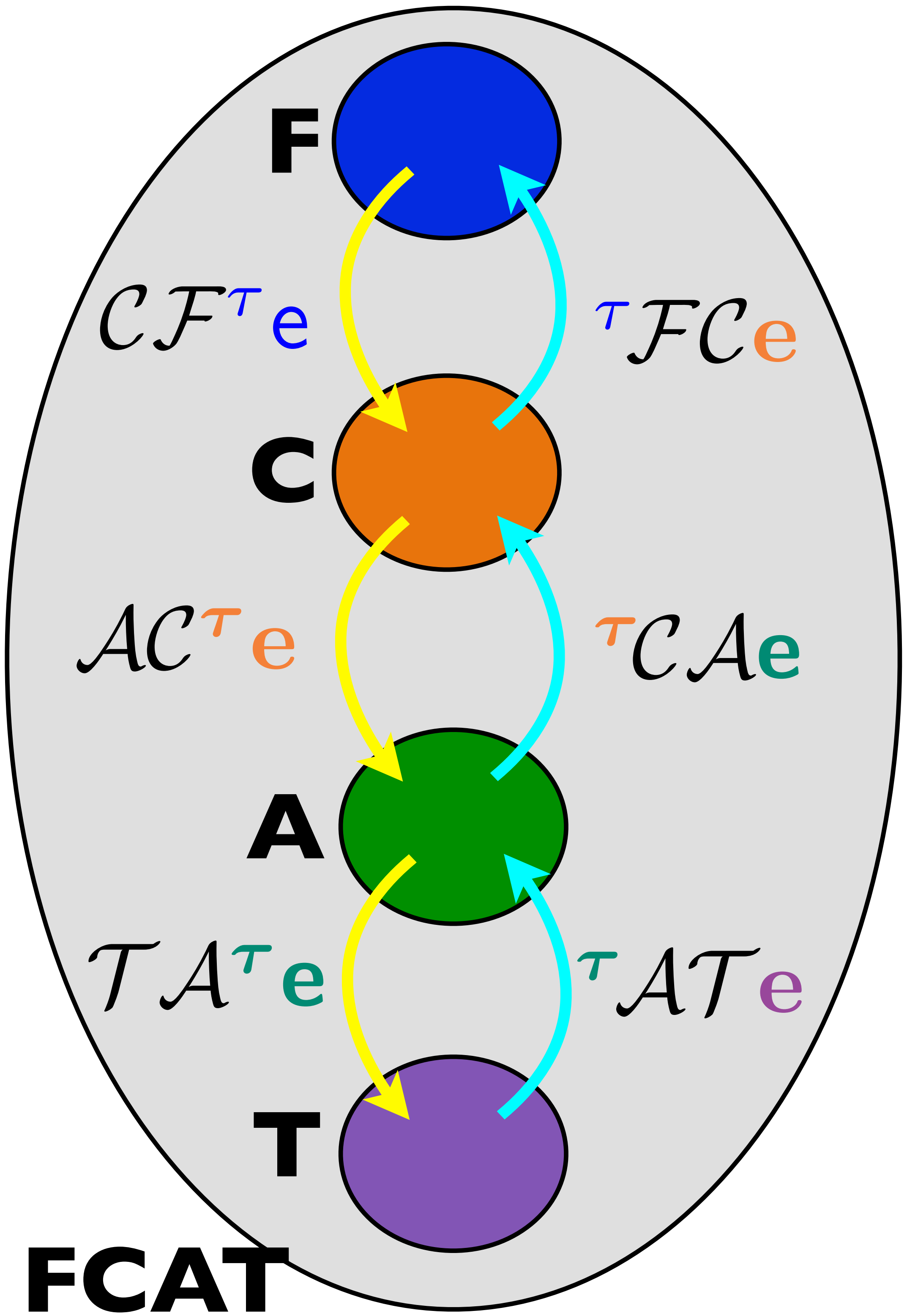
$$\mathcal{CF}^{\tau \to \tau'} \mathsf{v} \longmapsto \mathbf{pack} \langle \mathbf{unit}, \langle \mathbf{v}, () \rangle \rangle \ \mathbf{as} \ \exists \beta. \langle ((\beta, \tau^{\mathcal{C}}) \to \tau'^{\mathcal{C}}), \beta \rangle$$

$$\text{where } \mathbf{v} = \boldsymbol{\lambda}(\mathbf{z} : \mathbf{unit}, \mathbf{x} : \tau^{\mathcal{C}}). \mathcal{CF}^{\tau'} (\mathsf{v} \ {}^{\tau}\mathcal{FC} \ \mathbf{x})$$

$$^{\tau \to \tau'}\mathcal{FC}\mathbf{v} \longmapsto \lambda(\mathsf{x} : \tau). {}^{\tau'}\mathcal{FC}(\mathbf{unpack} \ \langle \beta, \mathbf{y} \rangle = \mathbf{v}$$

$$\mathbf{in} \ \boldsymbol{\pi}_1(\mathbf{y}) \ \boldsymbol{\pi}_2(\mathbf{y}) \ \mathcal{CF}^{\tau} \mathsf{x})$$

Challenges

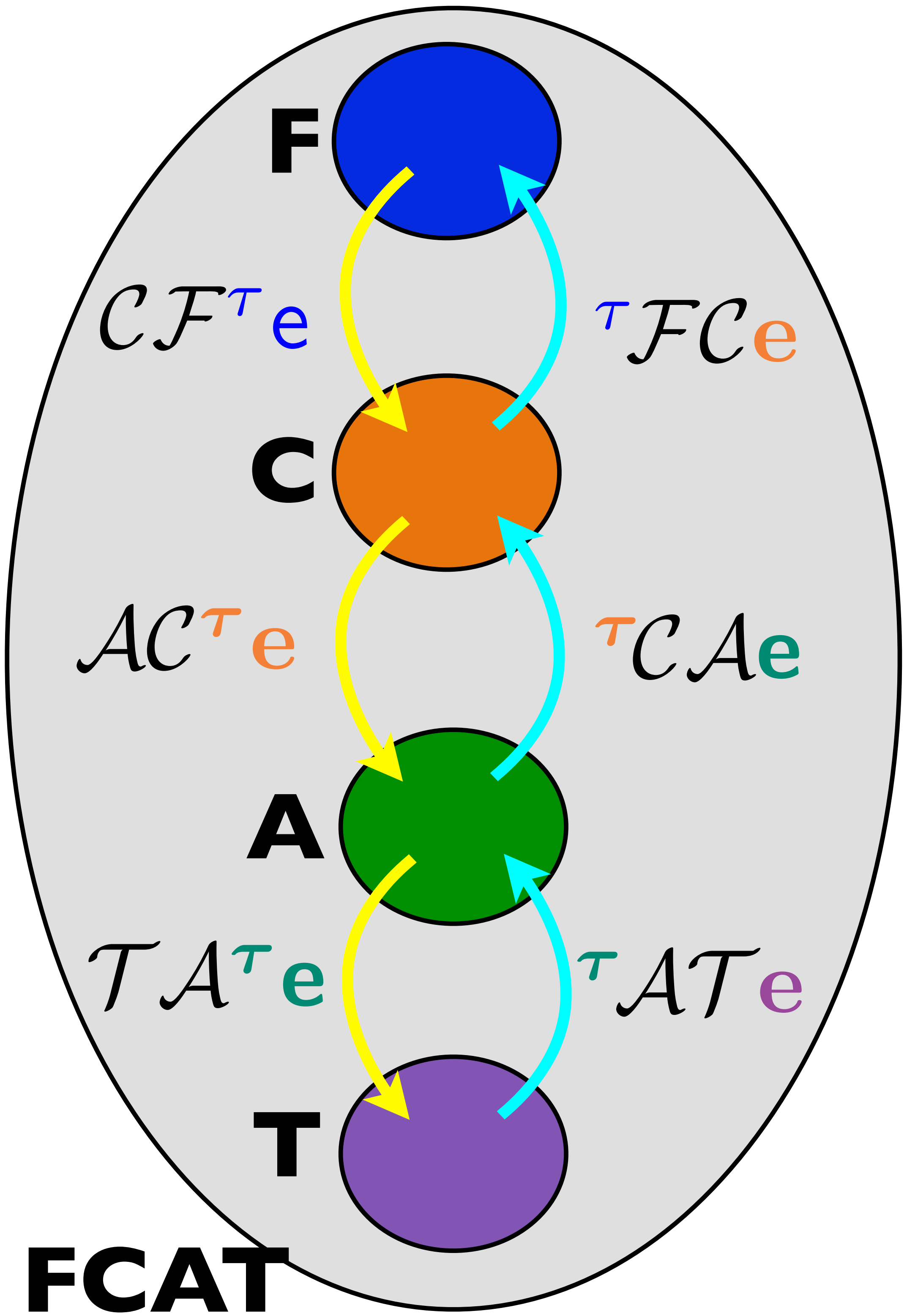


F+C: Interoperability semantics with type abstraction in both languages

C+A: Interoperability when compiler pass allocates code & tuples on heap, e.g., $\mathcal{AC}\langle \mathbf{v_1}, \mathbf{v_2} \rangle$

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

Challenges

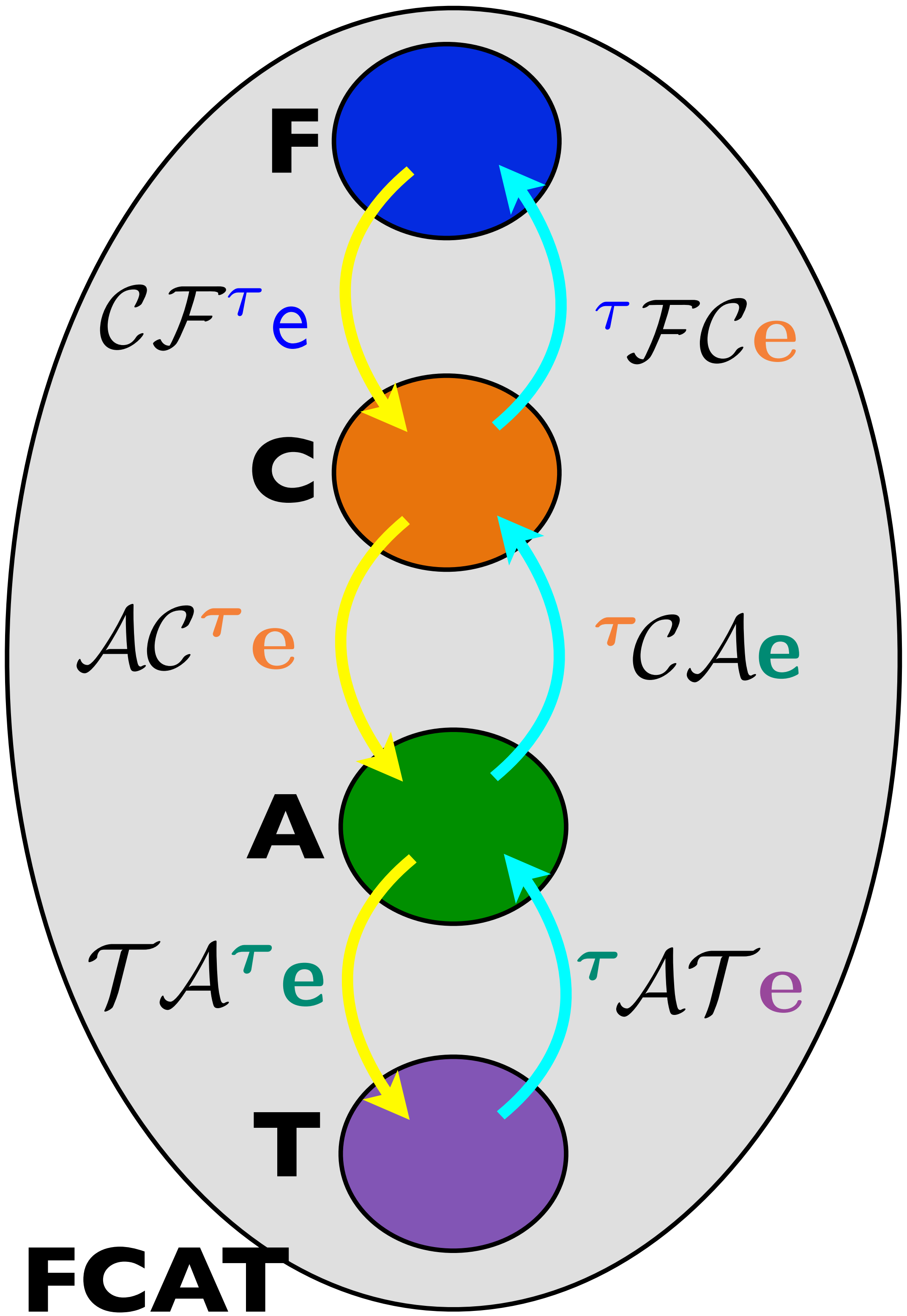


F+C: Interoperability semantics with type abstraction in both languages

C+A: Interoperability when compiler pass allocates code & tuples on heap, e.g., $\mathcal{AC}\langle \mathbf{v_1}, \mathbf{v_2} \rangle$

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

Challenges



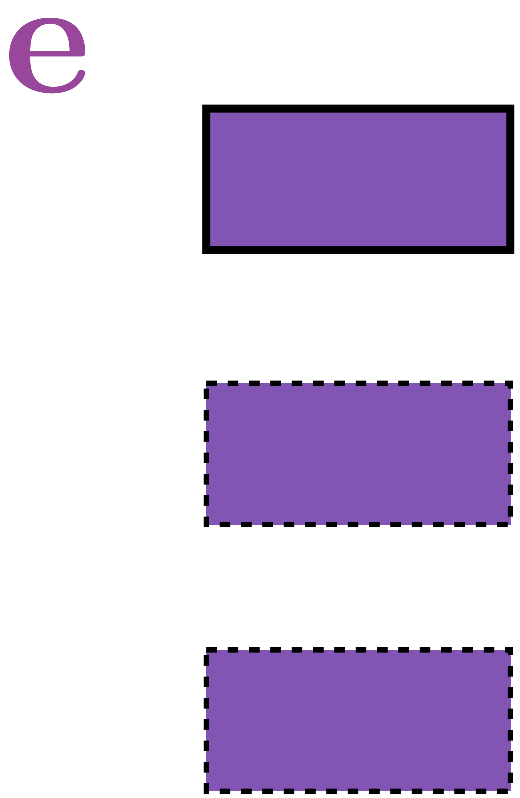
F+C: Interoperability semantics with type abstraction in both languages

C+A: Interoperability when compiler pass allocates code & tuples on heap, e.g., $\mathcal{AC}\langle \mathbf{v_1}, \mathbf{v_2} \rangle$

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

What is a component in TAL?

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

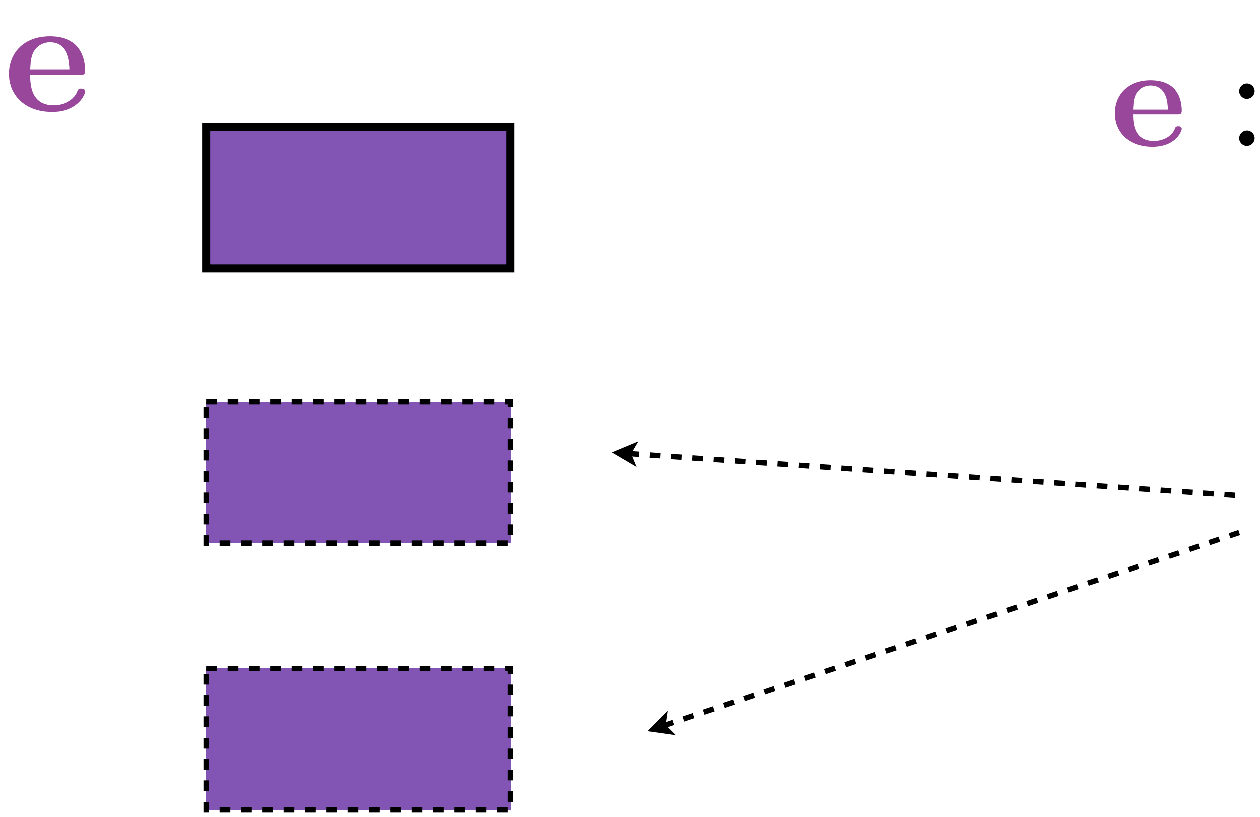


What is a component in TAL?

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

basic block = instruction sequence

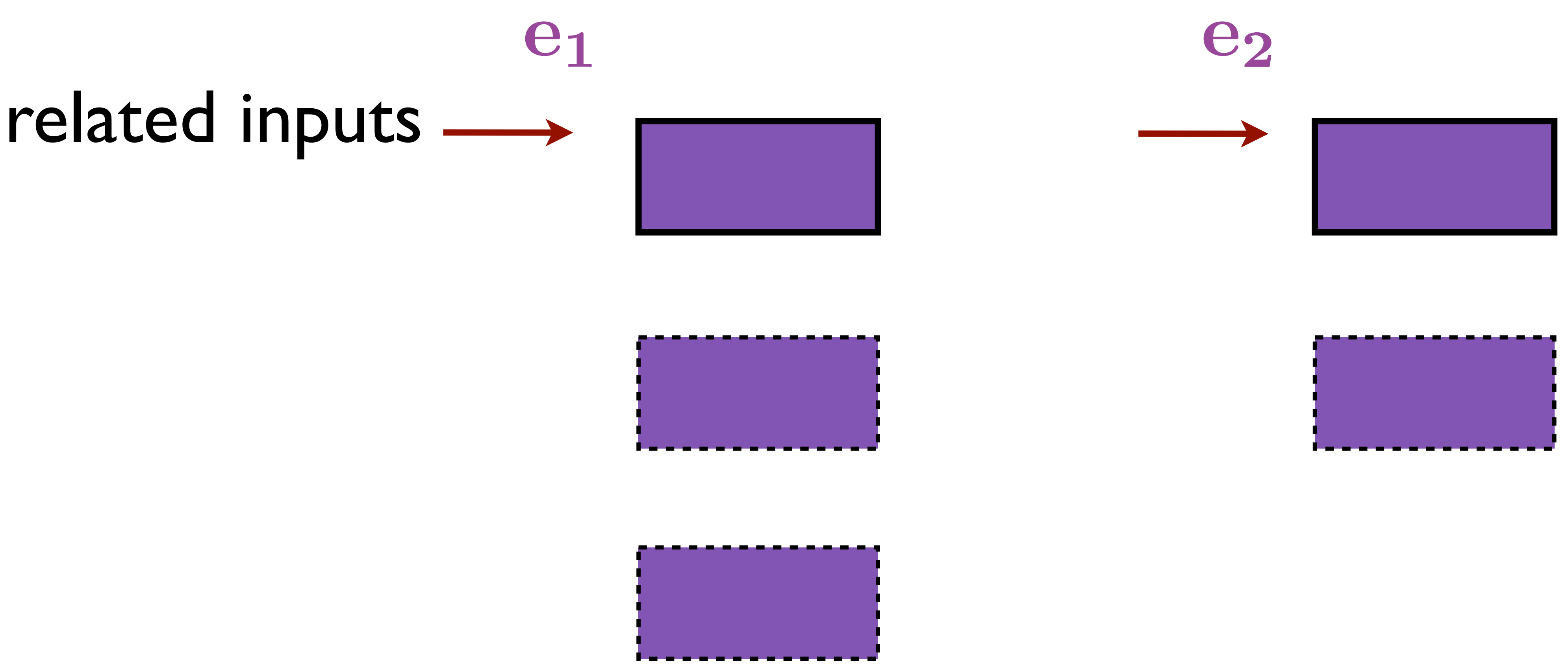
$$\mathbf{e} ::= (\mathbf{I}, \mathbf{H})$$



Heap with basic blocks

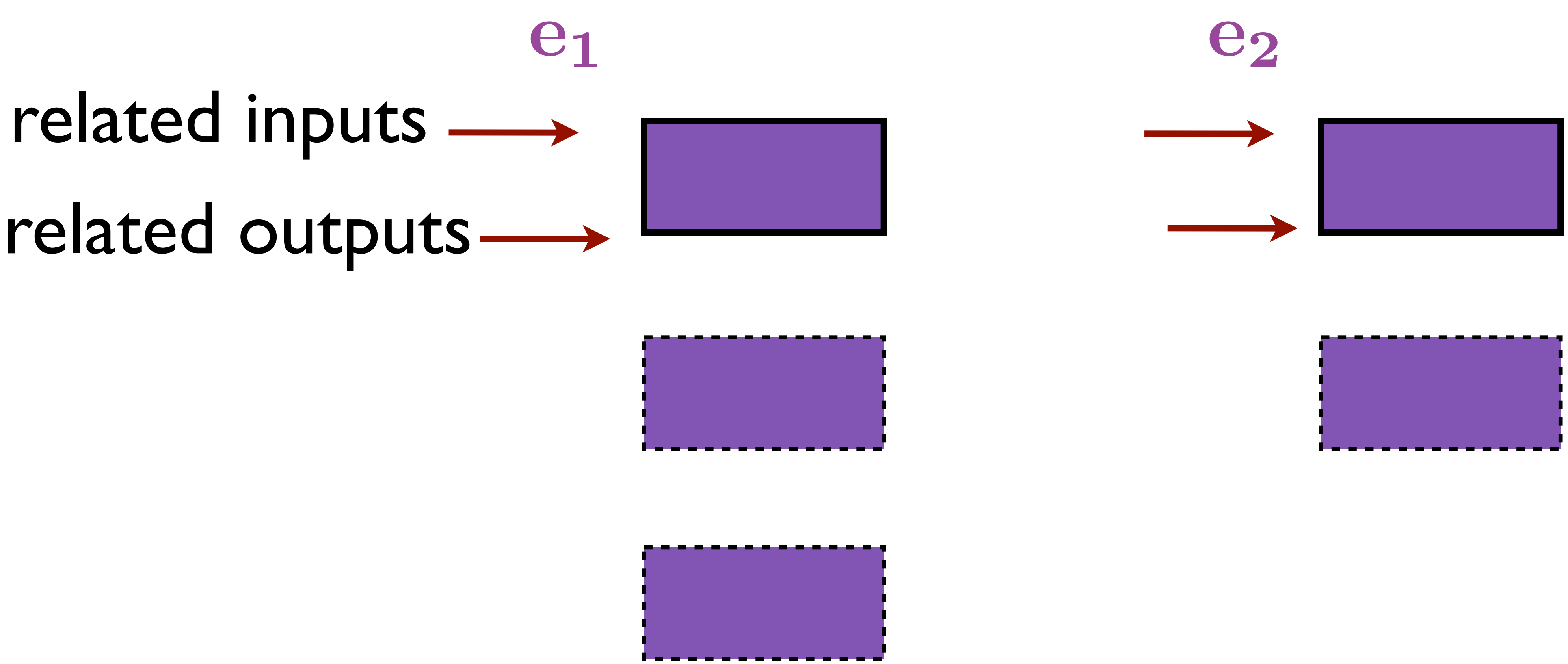
Equivalence of components in TAL?

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



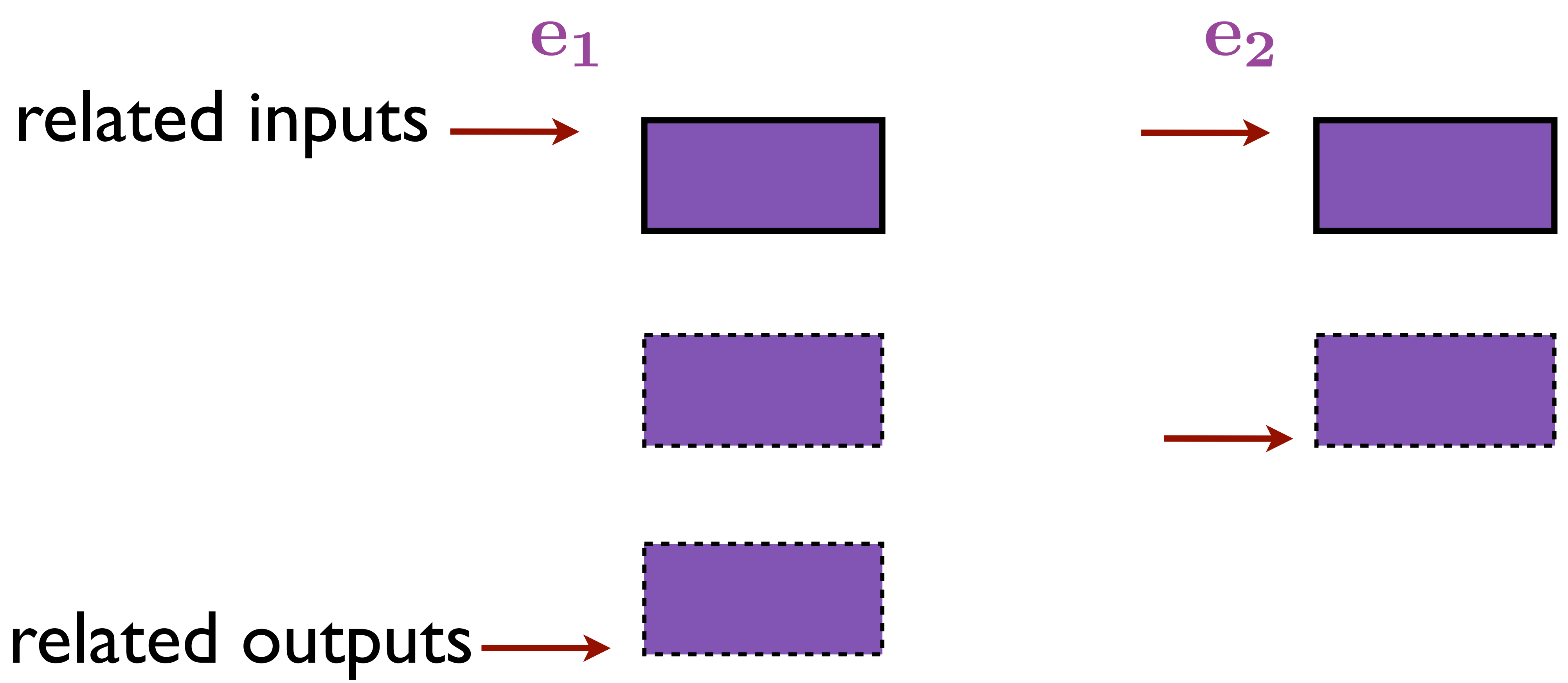
Equivalence of components in TAL?

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



Equivalence of components in TAL?

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



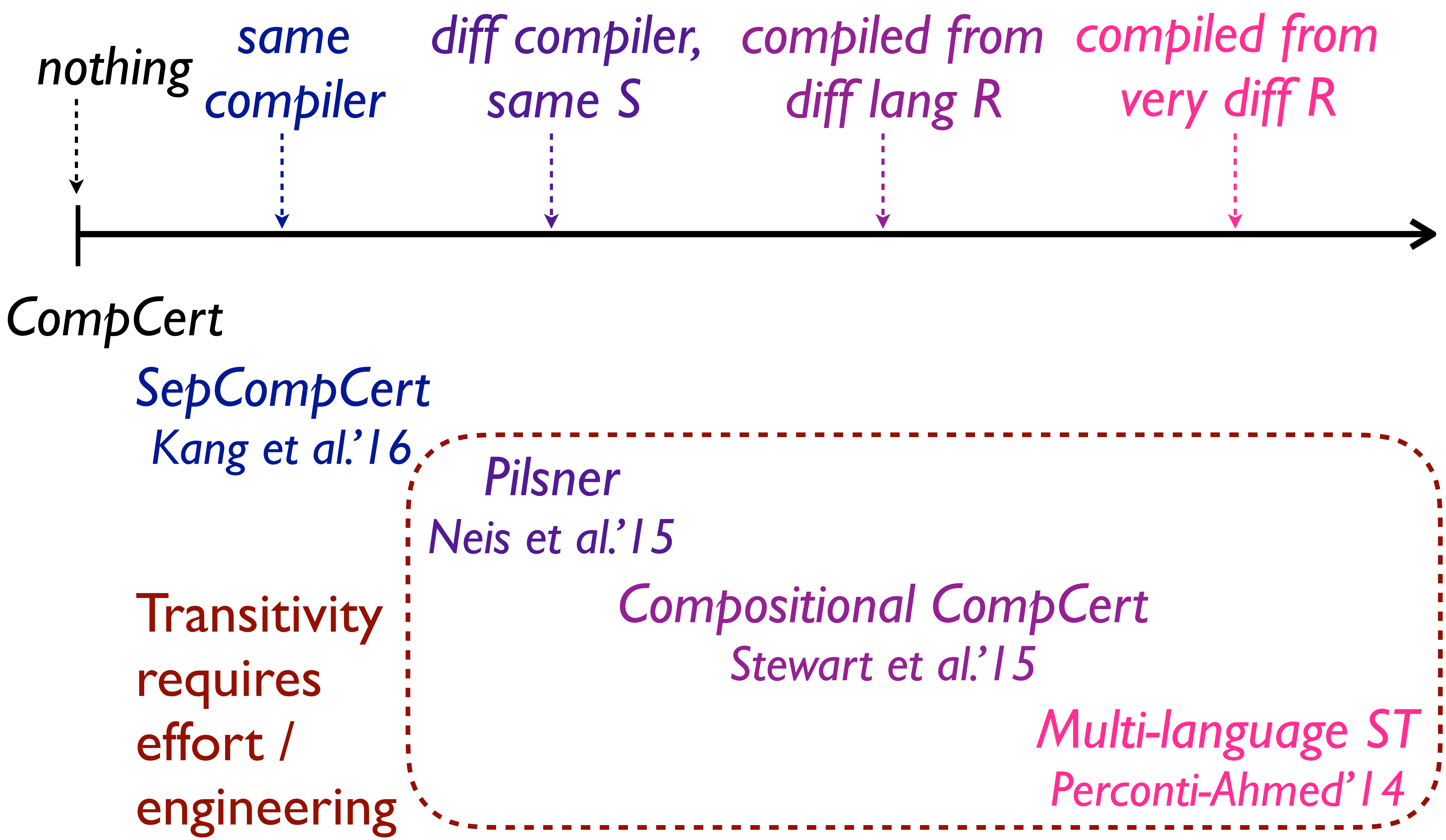
central challenge: interoperability between high-level (direct-style) language & assembly (continuation style)

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

CompCompCert vs. Multi-language

Transitivity	structured simulation	all passes use multi-lang $\approx^{ctx}$
Check okay-to-link-with	satisfies CompCert memory model	satisfies expected type (translation of source type)
Requires uniform memory model across compiler IRs	yes	no
Allows linking with behavior inexpressible in S	no	yes

Proving Transitivity

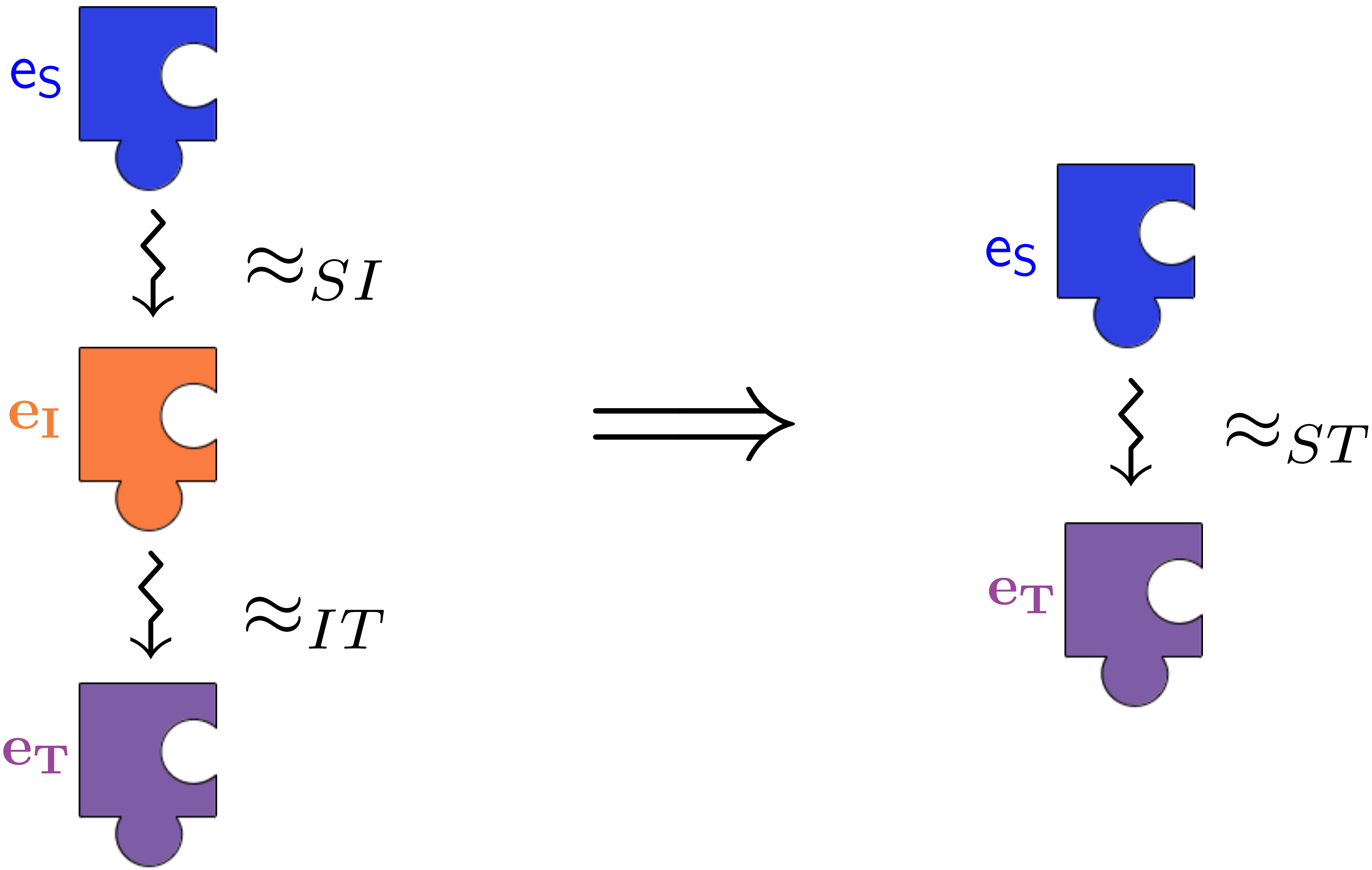


Vertical Compositionality

Transitivity

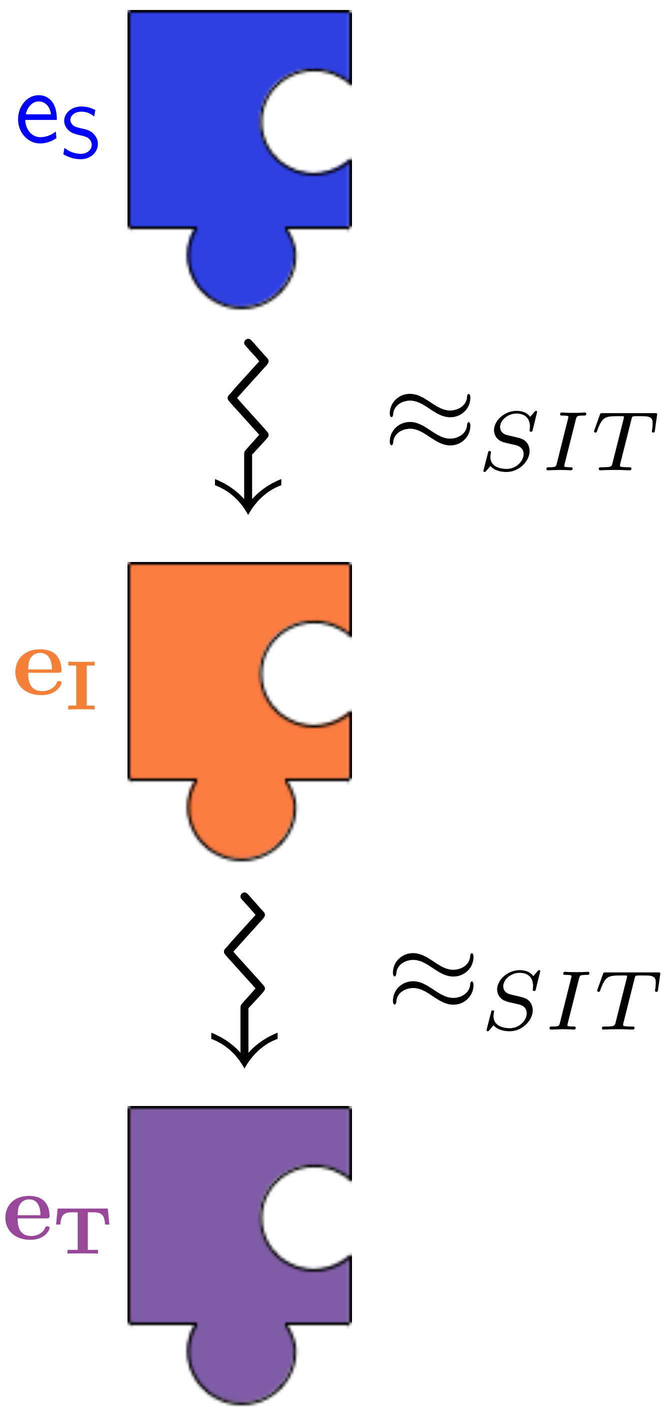
Vertical Compositionality

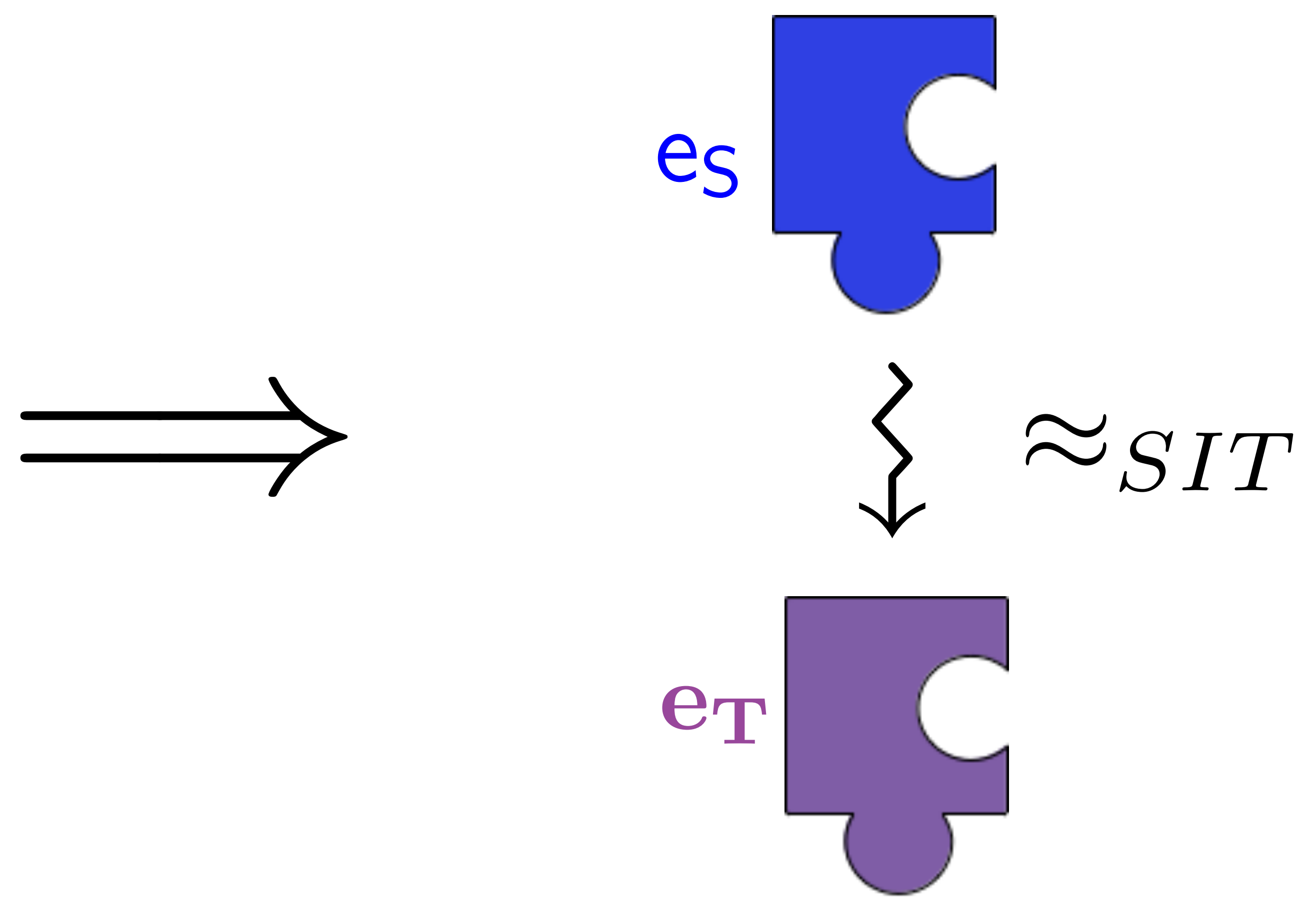
Vertical Compositionality



Transitivity

CompCompCert & Multi-lang





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

To Understand if Theorem is Correct…

Pilsner
Neis et al.'15

- source-target PILS

Compositional CompCert
Stewart et al.'15

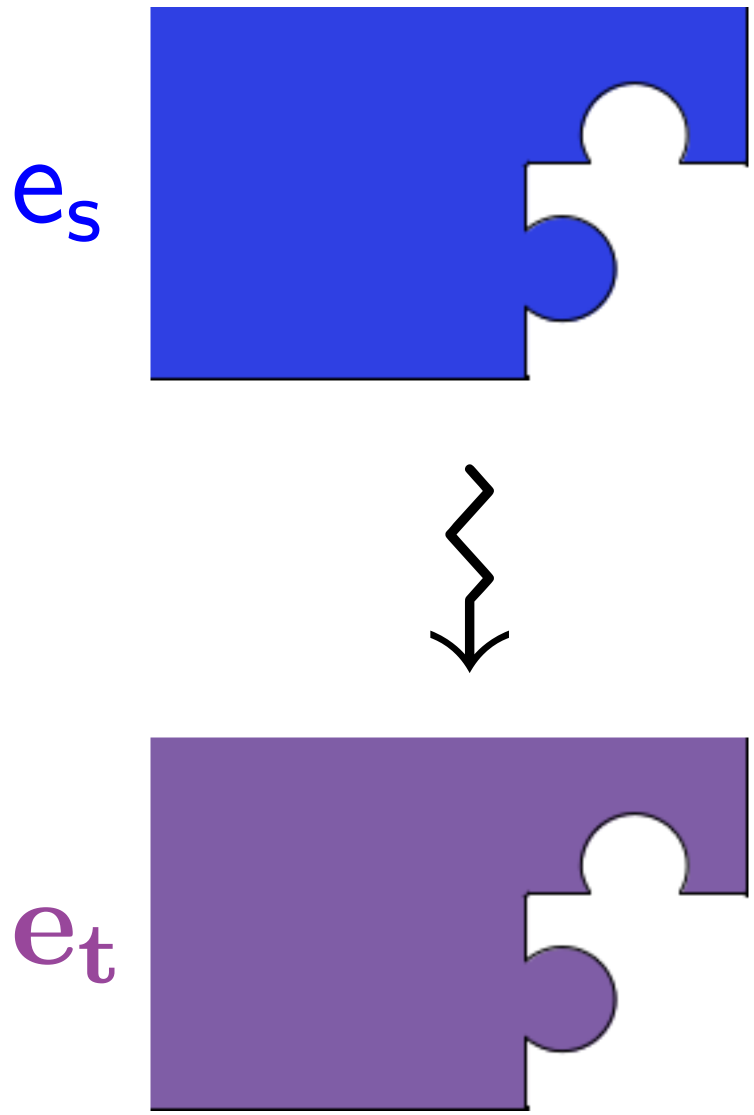
- interaction semantics & structured simulations

Multi-language ST
Perconti-Ahmed'14

- source-target multi-language

Is there a generic CCC theorem?

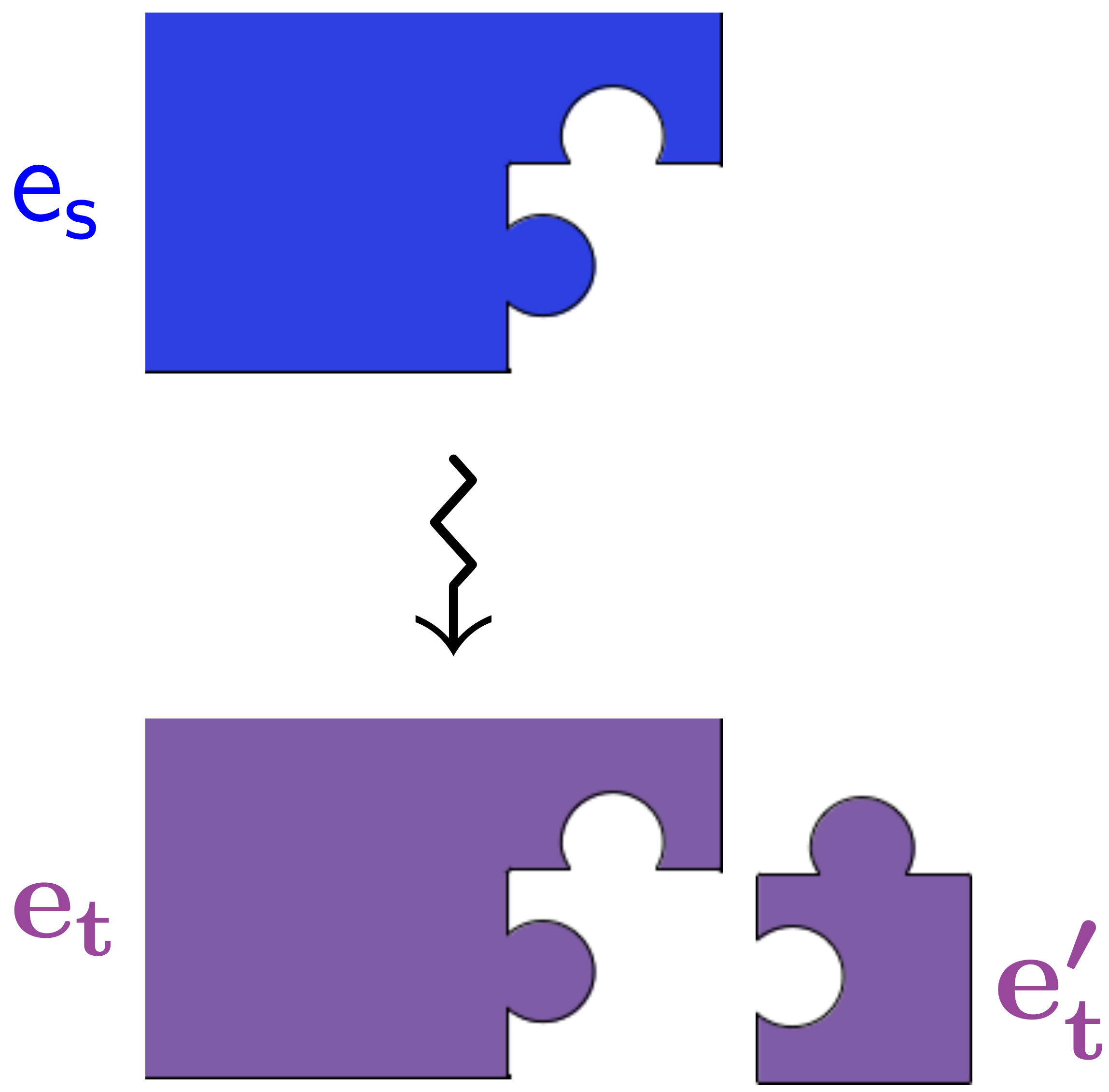
Generic CCC Theorem?



$e_S \approx e_T$

expressed how?

Generic CCC Theorem?



$e_S \approx e_T$

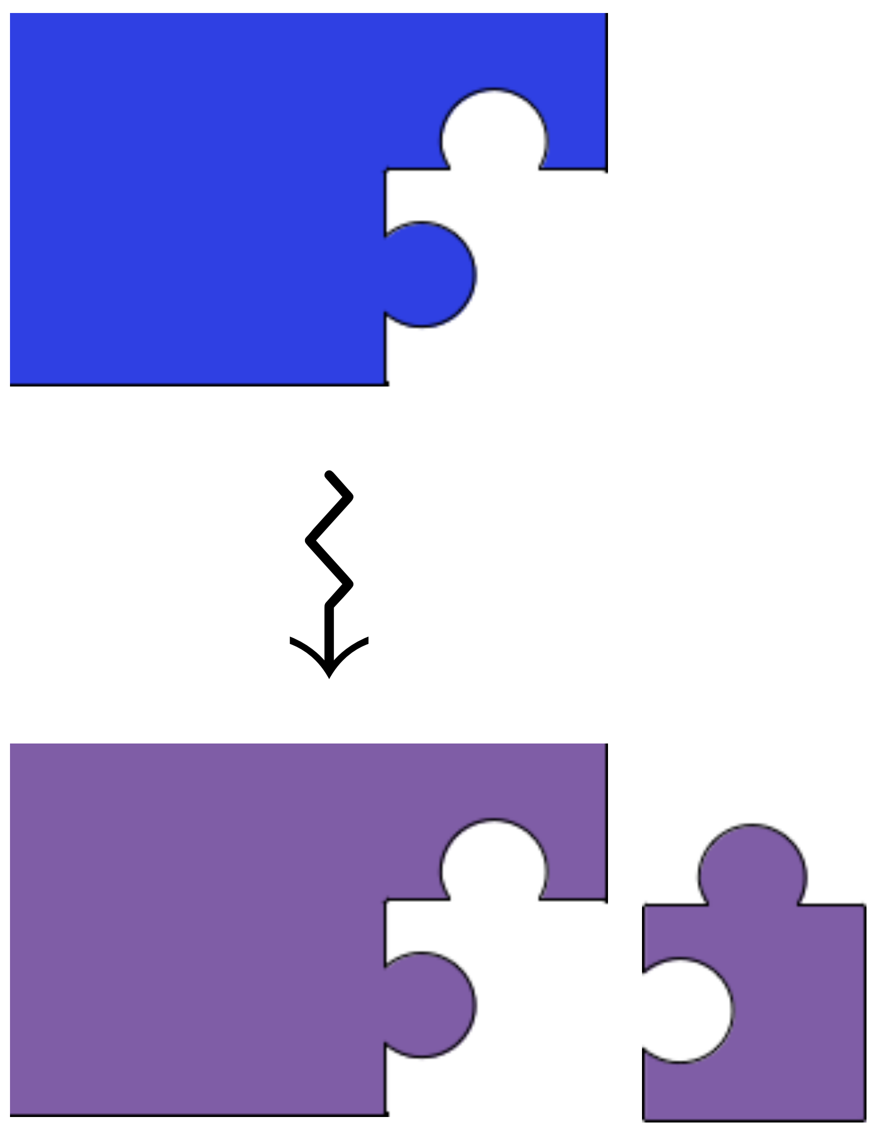
expressed how?

Generic CCC Theorem?

$e_S \approx e_T$

↑ expressed how?

e_s

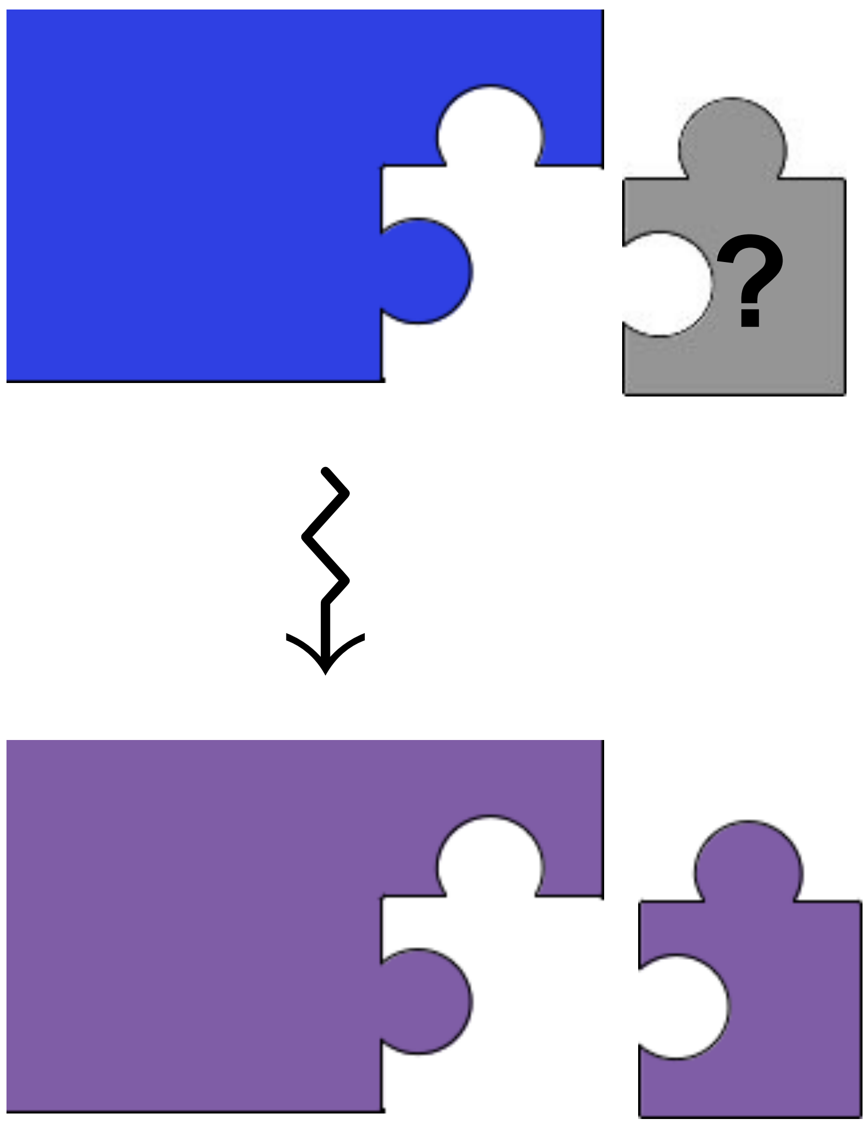


e_t

$e'_t, \varphi \in \mathcal{L} \longleftarrow$ *linking set*

Generic CCC Theorem?

$\mathbf{e_s}$



$\mathbf{e_t}$

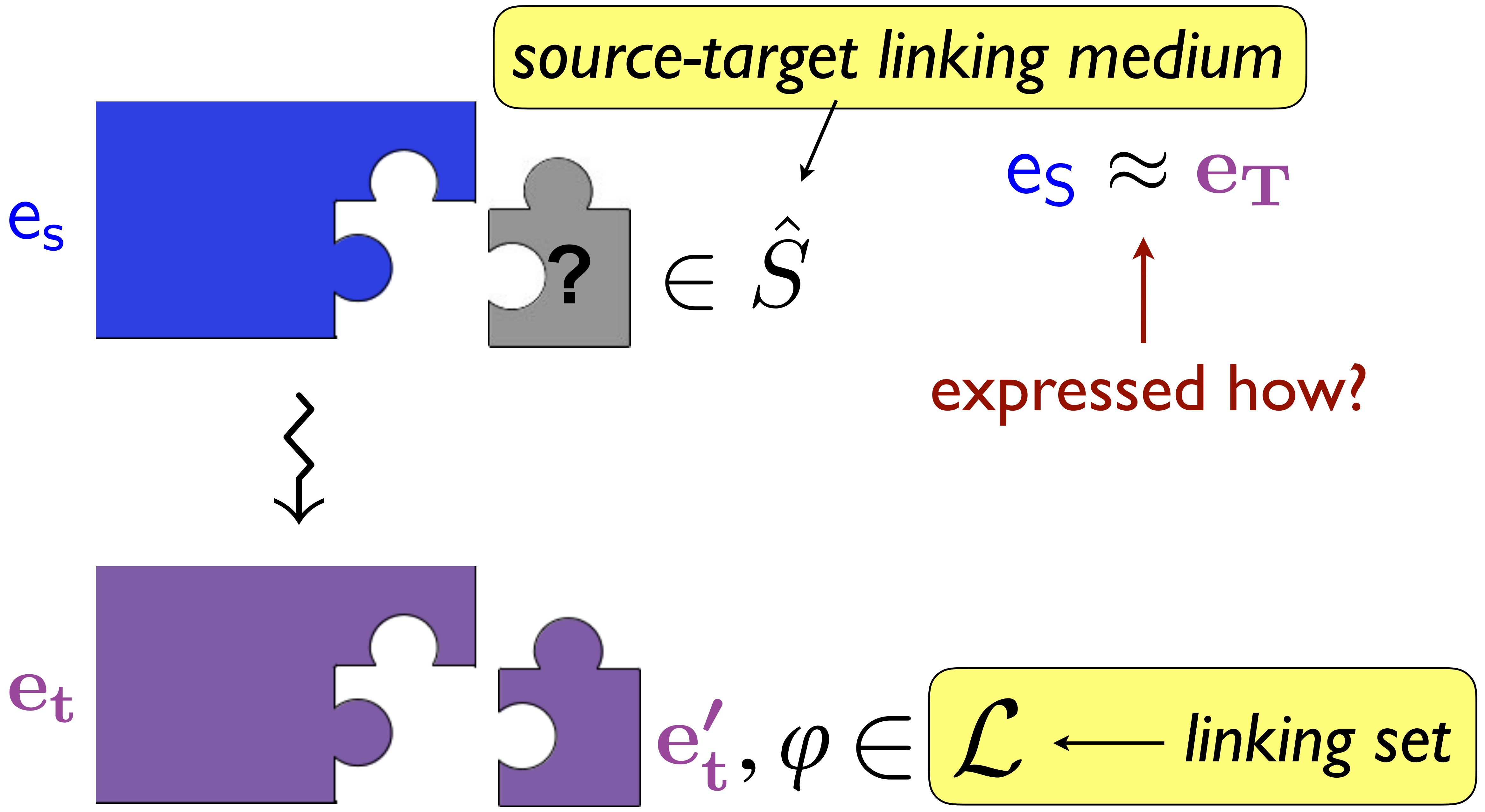
$\mathbf{e'_t}, \varphi \in \mathcal{L} \longleftarrow$ *linking set*

$\mathbf{e_S} \approx \mathbf{e_T}$

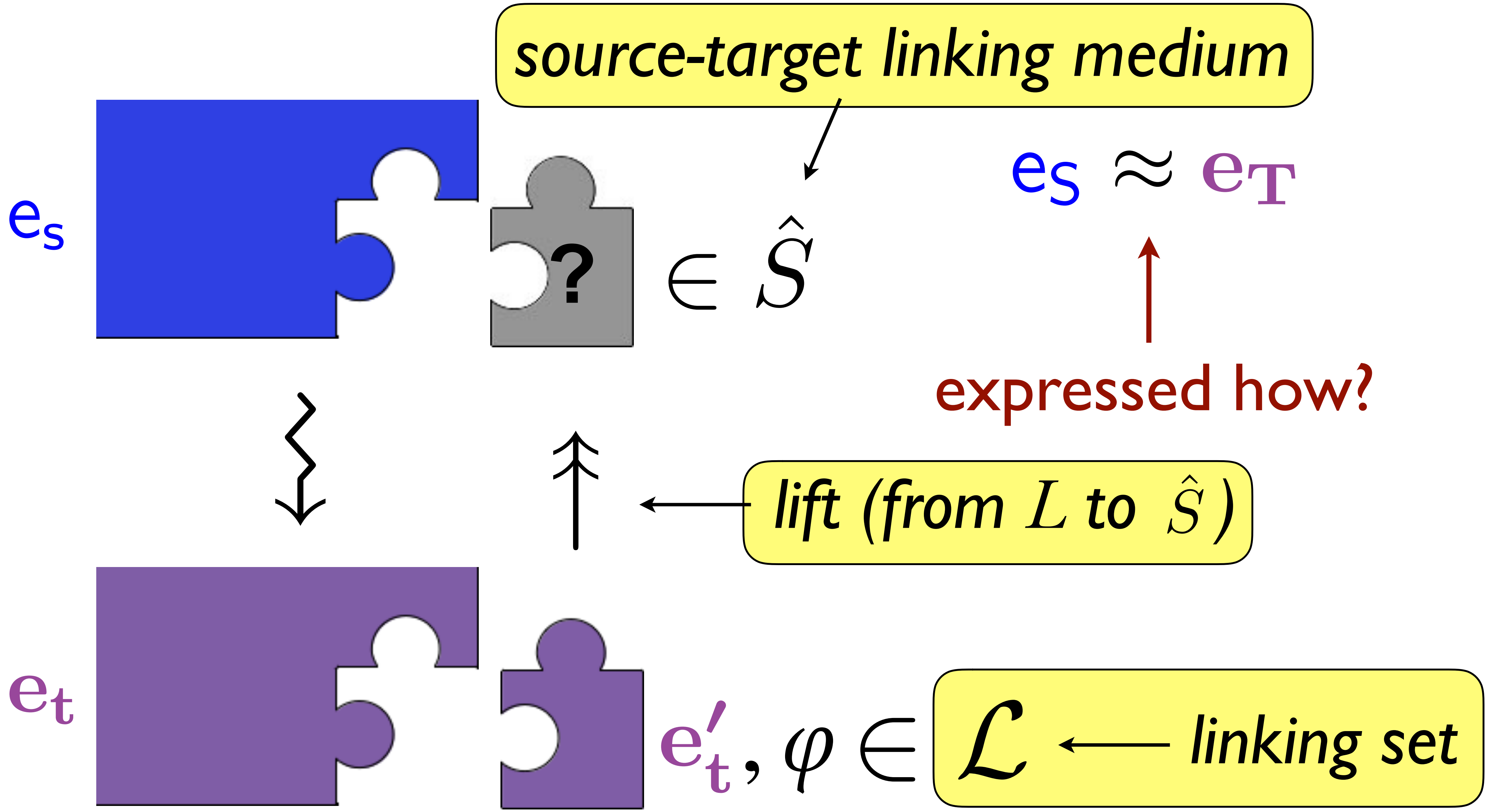
$\uparrow$

expressed how?

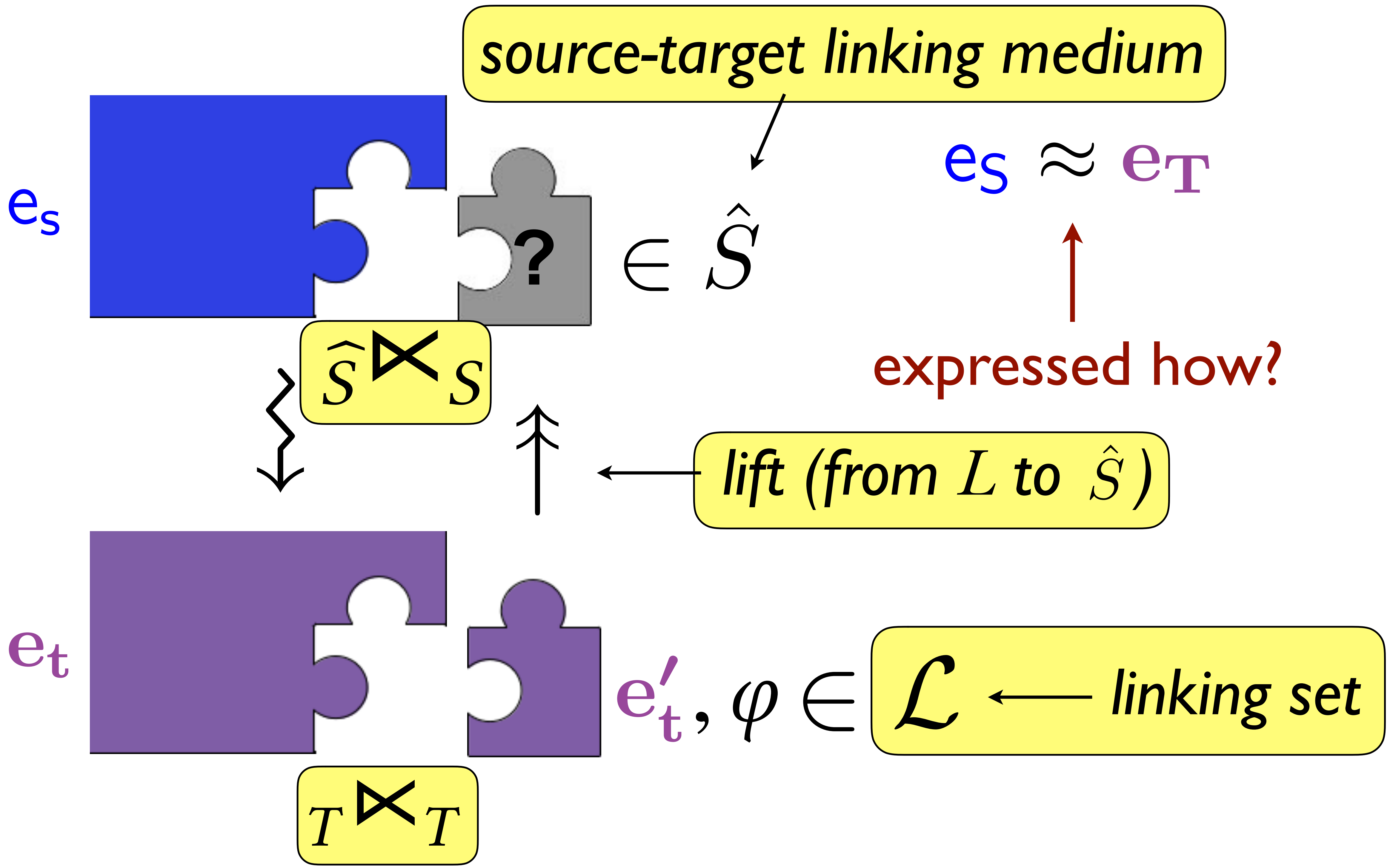
Generic CCC Theorem?



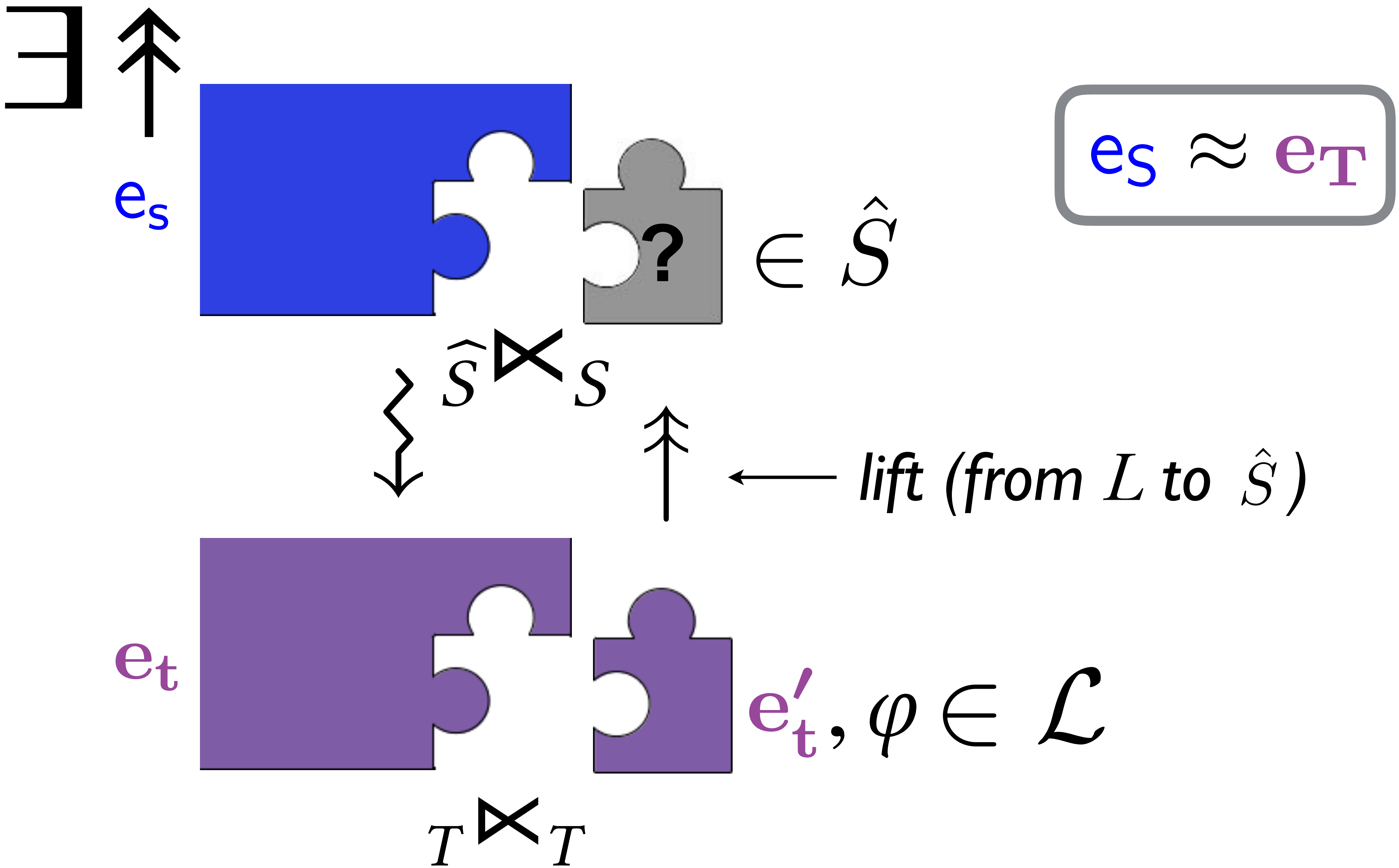
Generic CCC Theorem?



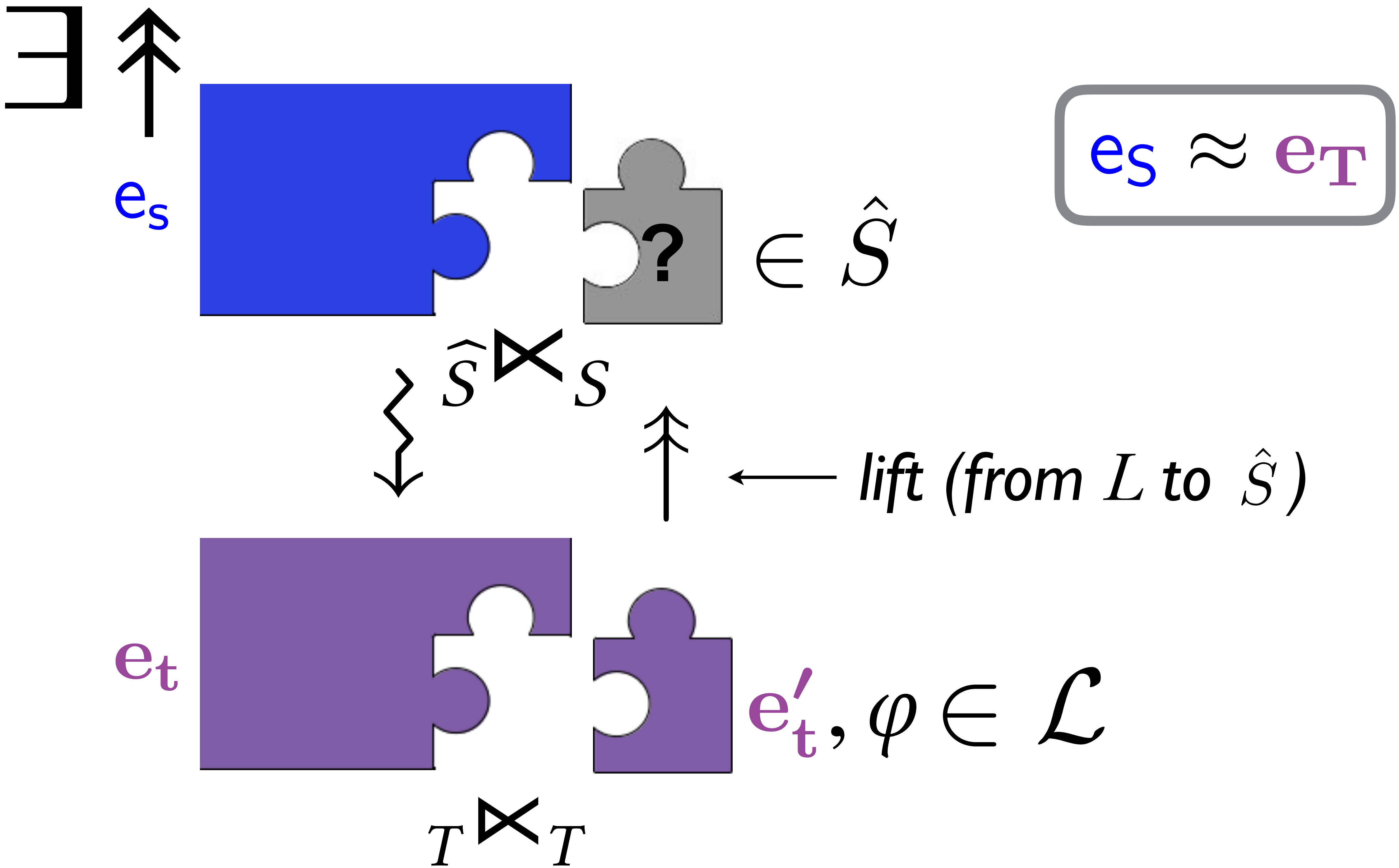
Generic CCC Theorem?



Generic CCC Theorem



Generic CCC Theorem



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

Generic CCC Theorem (Formally)

$$\exists \Uparrow.\ \forall e_S \in S.\ \forall (e_T, \varphi) \in \mathcal{L}.$$

$$e_T \ {}_T\!\bowtie_T\ C_T^S(e_S) \ {}_T\!\sqsubseteq_{\widehat{S}}\ \Uparrow(e_T, \varphi) \ {}_{\widehat{S}}\!\bowtie_S\ e_S$$

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

$$\forall (e_T, \varphi) \in \mathcal{L}.\ \forall e_S.$$

$$(\forall c_T.\ c_T \ {}_T\!\bowtie_T\ e_T \ {}_T\!\sqsubseteq_T\ c_T \ {}_T\!\bowtie_T\ C_T^S(e_S)) \implies$$

$$(\forall c_S.\ c_S \ {}_S\!\bowtie_{\widehat{S}}\ \Uparrow(e_T, \varphi) \ {}_{\widehat{S}}\!\sqsubseteq_S\ c_S \ {}_S\!\bowtie_S\ e_S)$$

The Next 700 Compiler Correctness Theorems (Functional Pearl)
[Patterson-Ahmed, ICFP 2019]

CCC Properties

Implies whole-program compiler correctness & separate compilation correctness

Can be instantiated with different formalisms…

CCC Properties

Implies whole-program compiler correctness & separate compilation correctness

Can be instantiated with different formalisms…

Pilsner

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

CCC Properties

Implies whole-program compiler correctness & separate compilation correctness

Can be instantiated with different formalisms…

Multi-language ST

$\mathcal{L}$ $\{(e_T, \_) \mid$ where e_T is any target component $\}$

$\widehat{S}$ source-target multi-language ST

$\widehat{S}\ltimes_S\ e\ {}_{ST}\ltimes_{ST}\ e_S$

$\Uparrow(\cdot)$ $\Uparrow(e_T, \_) = \mathcal{ST}(e_T)$

CCC Properties

Implies whole-program compiler correctness & separate compilation correctness

Can be instantiated with different formalisms…

Compositional CompCert

$\mathcal{L}$	$\{(e_T, \_) \mid$ where e_T is any target component $\}$
$\widehat{S}$	semantics that embeds source and target, equipped with interaction semantics
$_{\widehat{S}}\ltimes_S$	adding another module to combined semantics
$\Uparrow(\cdot)$	$\Uparrow(e_T, \_) = e_T$

Benefits of CCC for the Next 700...

- Sheds light on pros & cons of compiler correctness formalisms
- Is a compositional compiler correctness theorem right? Instantiate CCC with your compiler correctness formalism & show that CCC follows as a corollary
- What's needed for better vertical compositionality / easier transitivity? ...

Vertical Compositionality for Free

$$\text{CCC}(S,I) \text{ and } \text{CCC}(I,T) \implies \text{CCC}(S,T)$$

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

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

Vertical Compositionality for Free

$\mathsf{CCC}(S,I)$ and $\mathsf{CCC}(I,T) \implies \mathsf{CCC}(S,T)$

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

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

Vertical Compositionality for Free

$$\text{CCC}(S,I) \text{ and } \text{CCC}(I,T) \implies \text{CCC}(S,T)$$

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

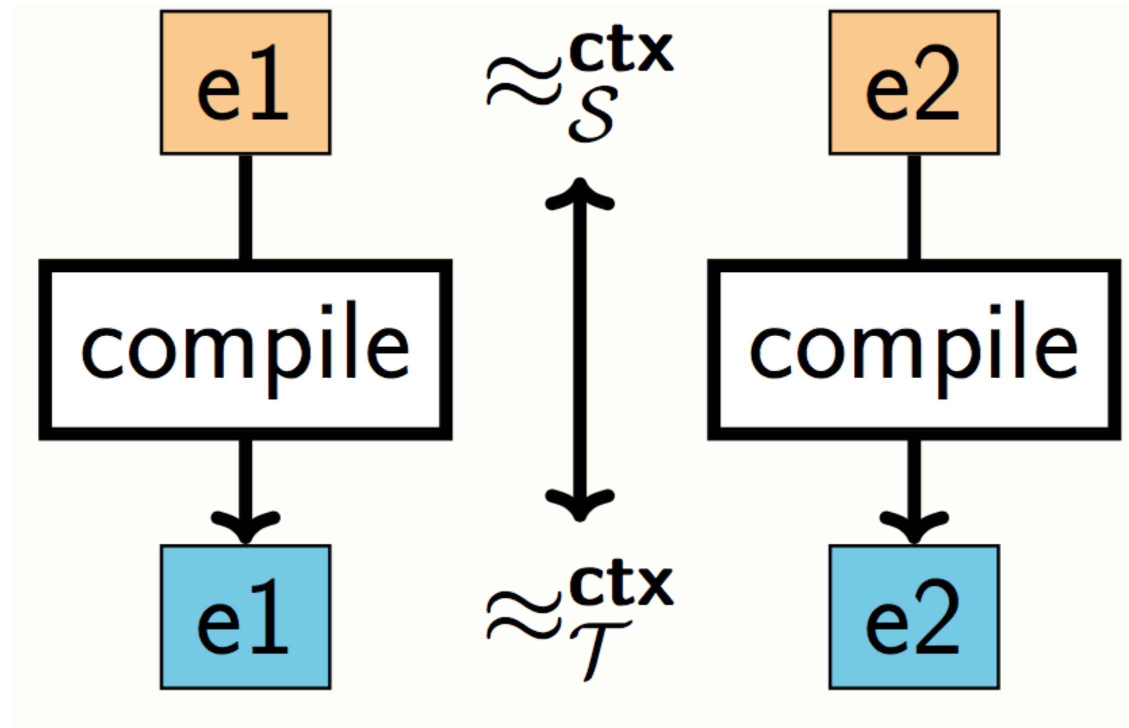
i.e., when lift $\Uparrow$ is a back-translation that maps every $e_T \in \mathcal{L}$ to some e_S

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

Fully abstract compilers have such back-translations!

Fully Abstract Compilers

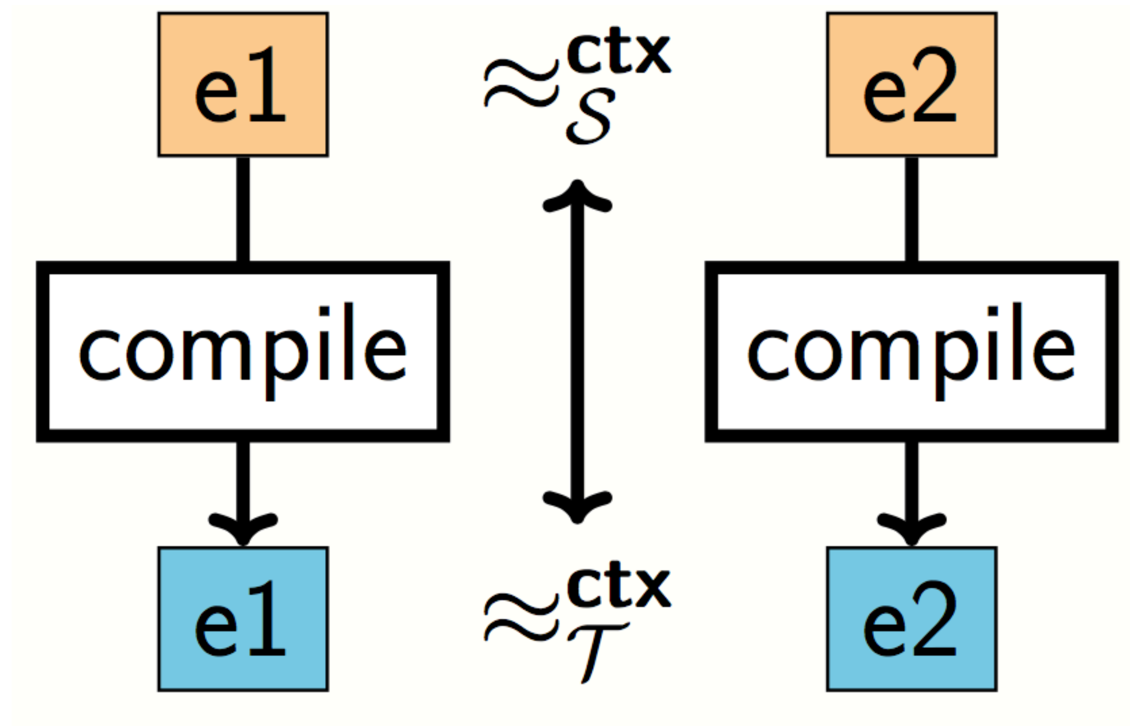
preserve equivalence



- ensure a compiled component does not interact with any target behavior that is inexpressible in S
  - *this guarantees programmer can reason at source level*

Fully Abstract Compilers

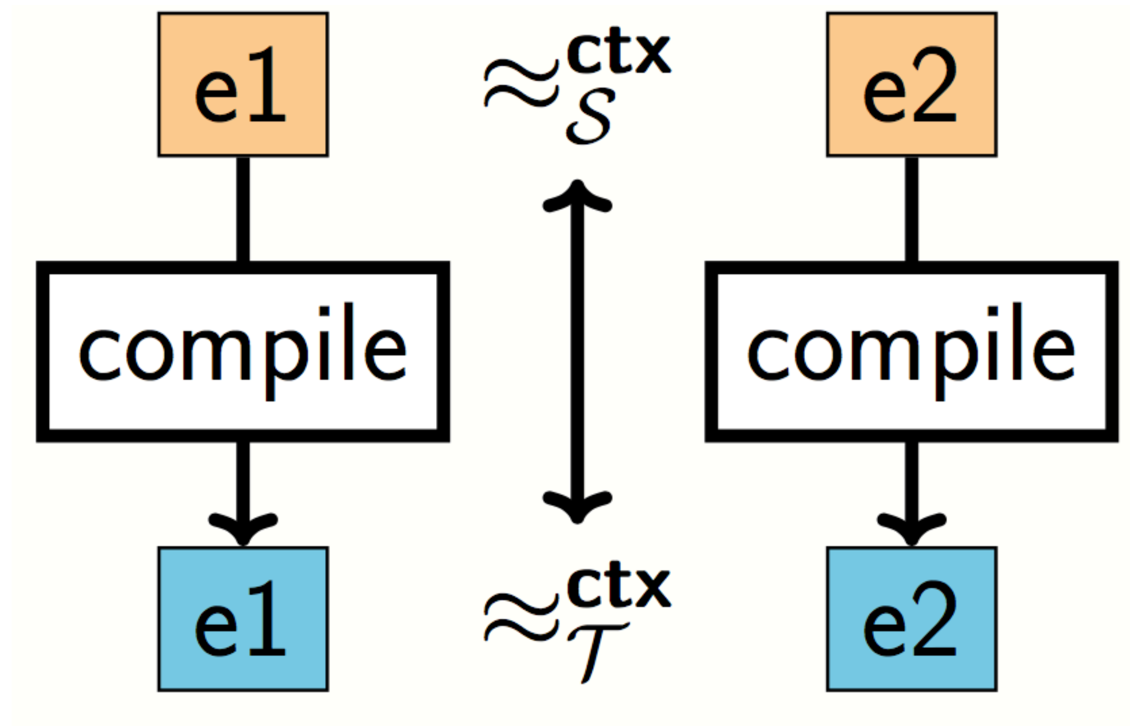
preserve equivalence



- ensure a compiled component does not interact with any target behavior that is inexpressible in S
  - *this guarantees programmer can reason at source level*
- Do we want to link with behavior inexpressible in S? Or do we want fully abstract compilers?

Fully Abstract Compilers

preserve equivalence



- ensure a compiled component does not interact with any target behavior that is inexpressible in S

  - *this guarantees programmer can reason at source level*

- Do we want to link with behavior inexpressible in S? Or do we want fully abstract compilers?

  We want both!

Stepping Back…

Current State of PL Design

ML

Rust

Java

Target

Current State of PL Design

ML

Rust

Java

Language specifications are incomplete!
Don't account for linking

Target

Current State of PL Design

escape hatches

ML
C FFI

Rust
unsafe

Java
JNI

Language specifications are incomplete!
Don’t account for linking

Target

The Way Forward…

Rethink PL Design with *Linking Types*

escape hatches

ML — C FFI

Rust — unsafe

Java — JNI

Rethink PL Design with *Linking Types*

escape hatches

ML — C FFI

Rust — unsafe

Java — JNI

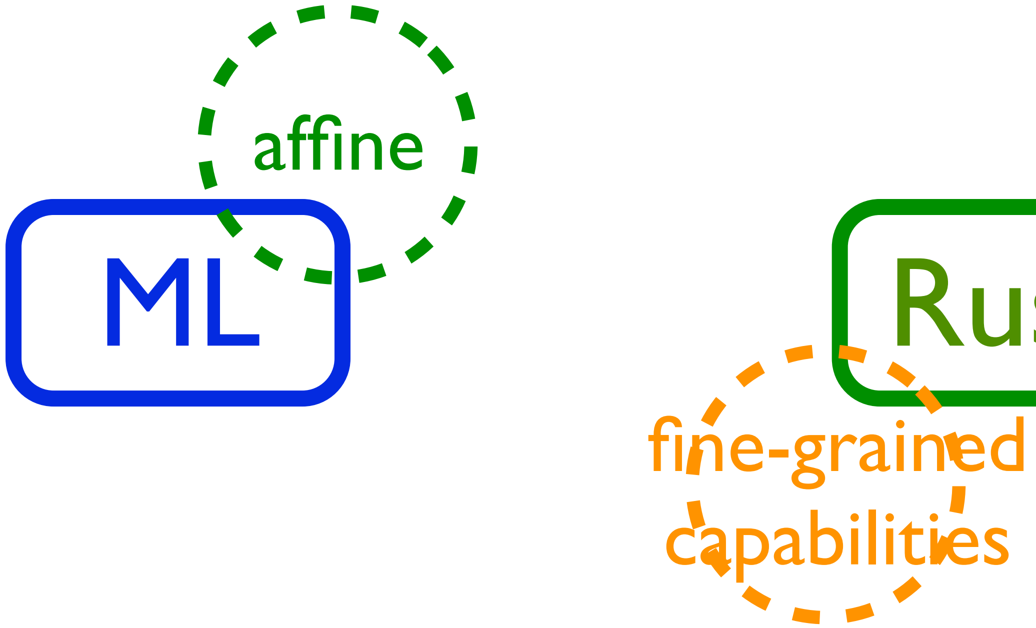
Design **linking types** extensions that support safe interoperability with other languages

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

PL Design, Linking Types

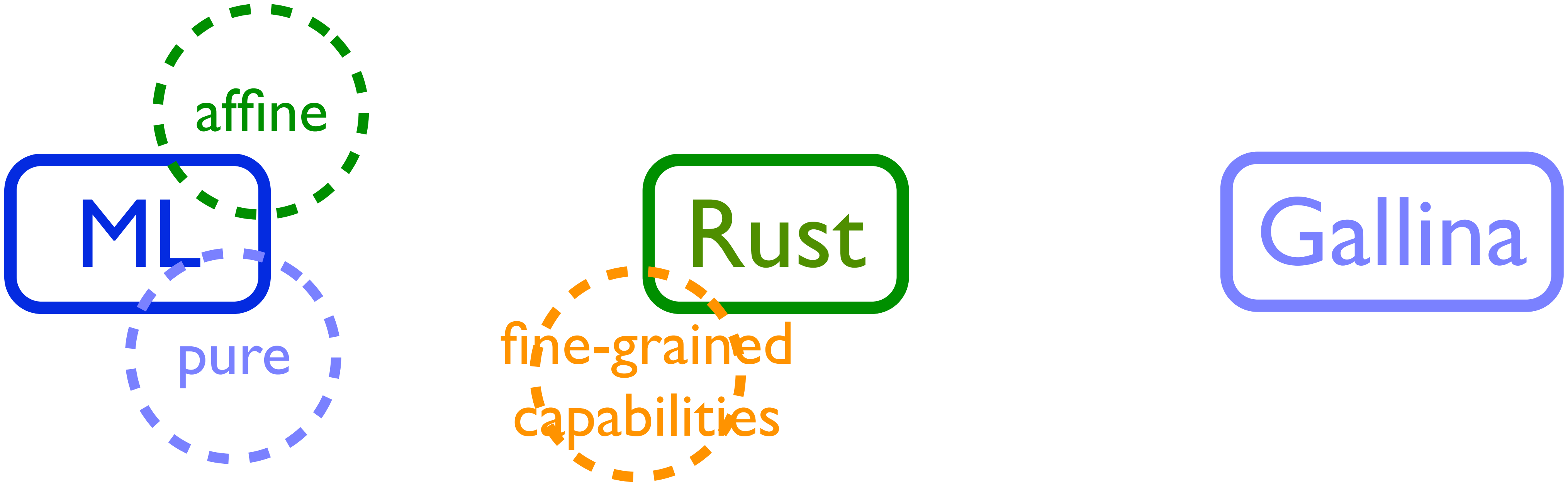
Only need linking types extensions to interact with behavior inexpressible in your language

PL Design, Linking Types



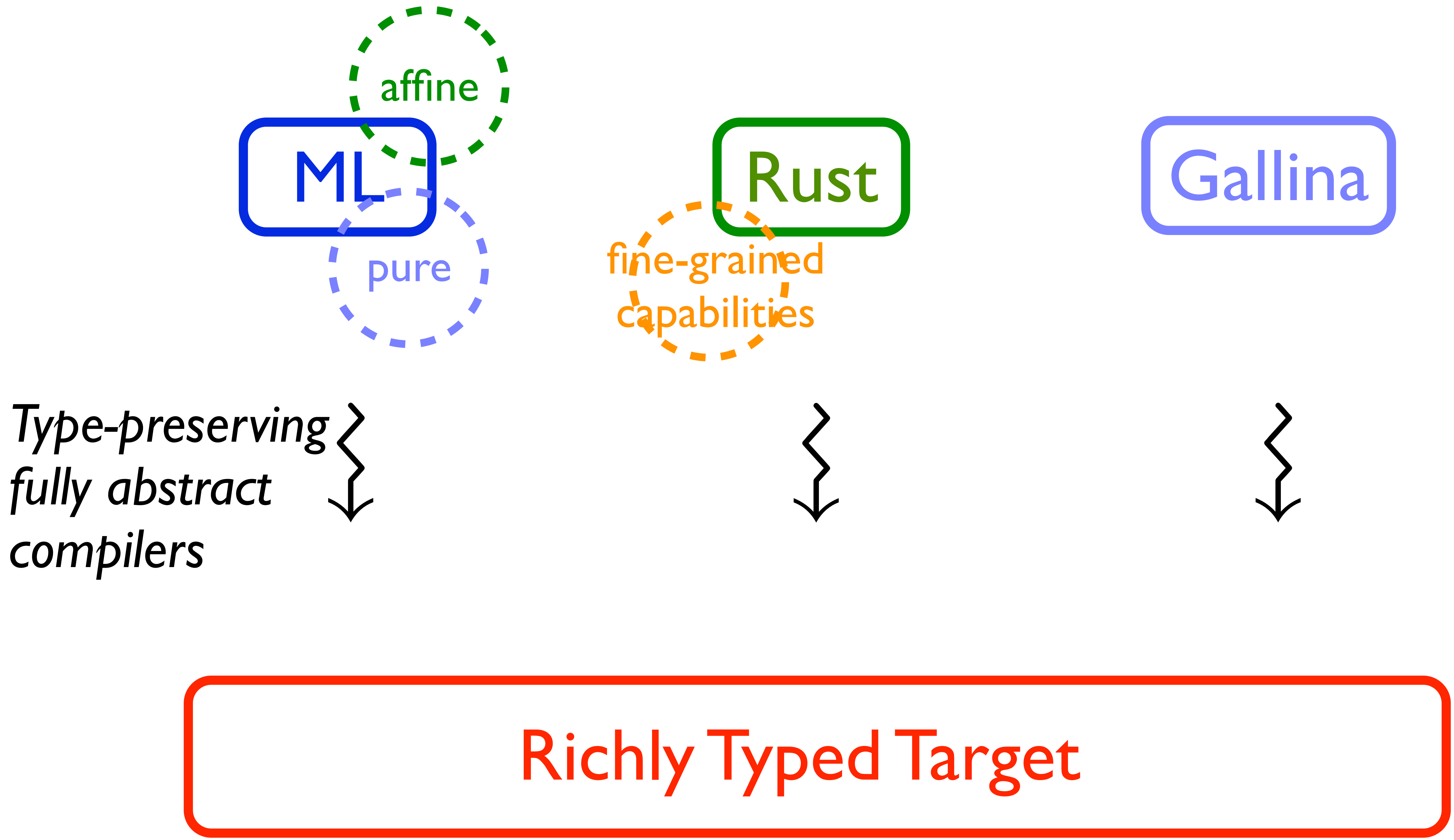
Only need linking types extensions to interact with behavior inexpressible in your language

PL Design, Linking Types



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

PL Design, Linking Types



Linking Types

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

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

Final Thoughts on Correct Compilation

- CompCert started a renaissance in compiler verification
  - major advances in mechanized proof
- Next challenge: Compositional Compiler Correctness
  - that applies to world of multi-language software
  - but source-independent linking and vertical compositionality are at odds
  - generic CCC theorem sheds light on current/future results

Secure Compilation

References & Future Directions

Formal Approaches to Secure Compilation:
A Survey of Fully Abstract Compilation
[Patrignani--Ahmed-Clarke, ACM Computing Surveys 2019]

Challenge: Proving Full Abstraction

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

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

$$\downarrow$$

$$\Gamma^+ \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{T}} \mathsf{e}_2 : \tau^+$$

Given:
No C_{S} can distinguish $\mathsf{e}_1, \mathsf{e}_2$

Show:
Given arbitrary C_{T} it cannot distinguish $\mathsf{e}_1, \mathsf{e}_2$

Need to be able to "back-translate" C_{T} to an equivalent C_{S}

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, exceptions) 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!

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

Fully Abstract Closure Conversion

[New et al. ICFP'16]

Source: STLC + μ types

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

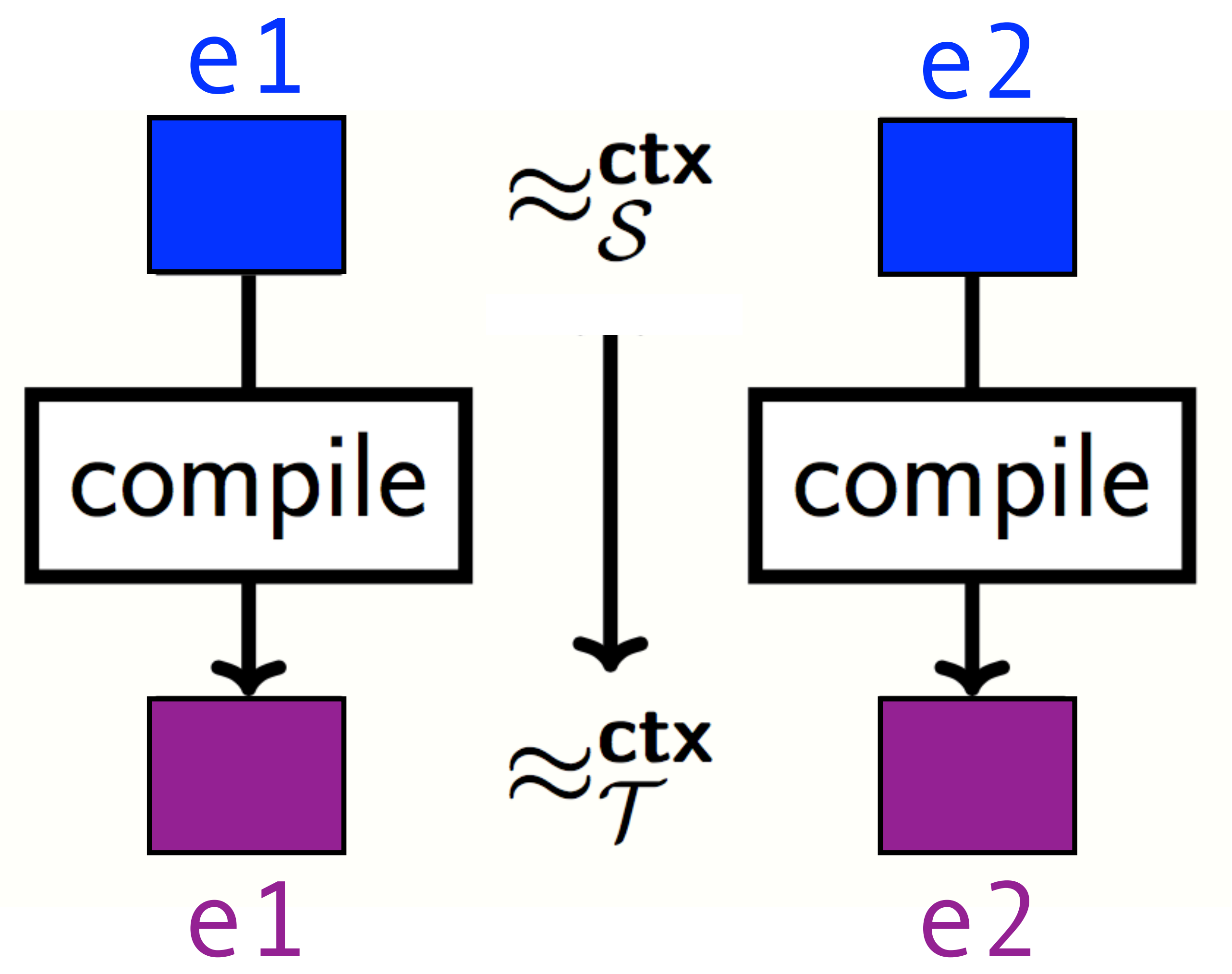
First full abstraction result where target has exceptions but source does not.

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

Proof technique: **Universal Embedding**

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

Dynamic Secure Compilation



Dynamic Secure Compilation

1. Cryptographically enforced: concurrent, distributed langs.
   - Join calculus to Sjoin with crypto primitives, preserves and reflect weak bisimulation *[Abadi et al. S&P'99, POPL'00, I&C'02]*
   - Pi-calculus to Spi-calculus *[Bugliesi and Giunti, POPL'07]*
   - F# with session types to F# with crypto primitives *[Corin et al., J. Comp. Security'08]*
   - Distributed WHILE lang. with security levels to WHILE with crypto and distributed threads *[Fournet et al, CCS'09]*
   - *TINYLINKS* distributed language to *F7* (ML w. refinement types), preserves data and control integrity *[Baltopoulos and Gordon, TLDI'09]*

Dynamic Secure Compilation

2. Dynamic Checks / Runtime Monitoring
   - STLC with recursion to untyped lambda-calc, proved fully abstract using *approximate back-translation*. Types erased and replaced w. dynamic checks. *[Devriese et al. POPL'16]*
   - f* (STLC with refs, exceptions) to js* (encoding of JavaScript in f*). Defensive wrappers perform dynamic type checks on untyped js* *[Fournet et al. POPL'13]*
   - Lambda-calc to VHDL digital circuits, run-time monitors check that external code respects expected communication protocol *[Ghica and Al-Zobaidi ICE'12]*

Dynamic Secure Compilation

3. Memory Protection Techniques
   (a) Address space layout randomization (ASLR)
      - STLC w. abstract memory, to target with concrete memory; show probabilistic full abstraction for large memory *[Abadi-Plotkin TISSEC'12]*
      - Added dynamic alloc, h.o. refs, call/cc, testing hash of reference, to target with probref to reverse hash *[Jagadeesan et al. CSF'11]*

Dynamic Secure Compilation

3. Memory Protection Techniques

    (b) Protected Module Architectures (PMAs) (e.g., Intel SGX)
    protected memory with code and data sections, and unprotected memory

    - Secure compilation of an OO language (with dynamic allocation, exceptions, inner classes) to PMA; proved fully abstract using trace semantics. Objects allocated in secure memory partition *[Patrignani et al. TOPLAS'15]*

Dynamic Secure Compilation

3. Memory Protection Techniques
   (c) PUMP Machine architecture tracks meta-data, registers and memory locations have tags, checked during execution
      - Secure compartmentalizing compiler with mutually distrustful compartments that can be compromised by attacker. OO lang to RISC with micro policies *[Juglaret et al. 2015]*

Dynamic Secure Compilation

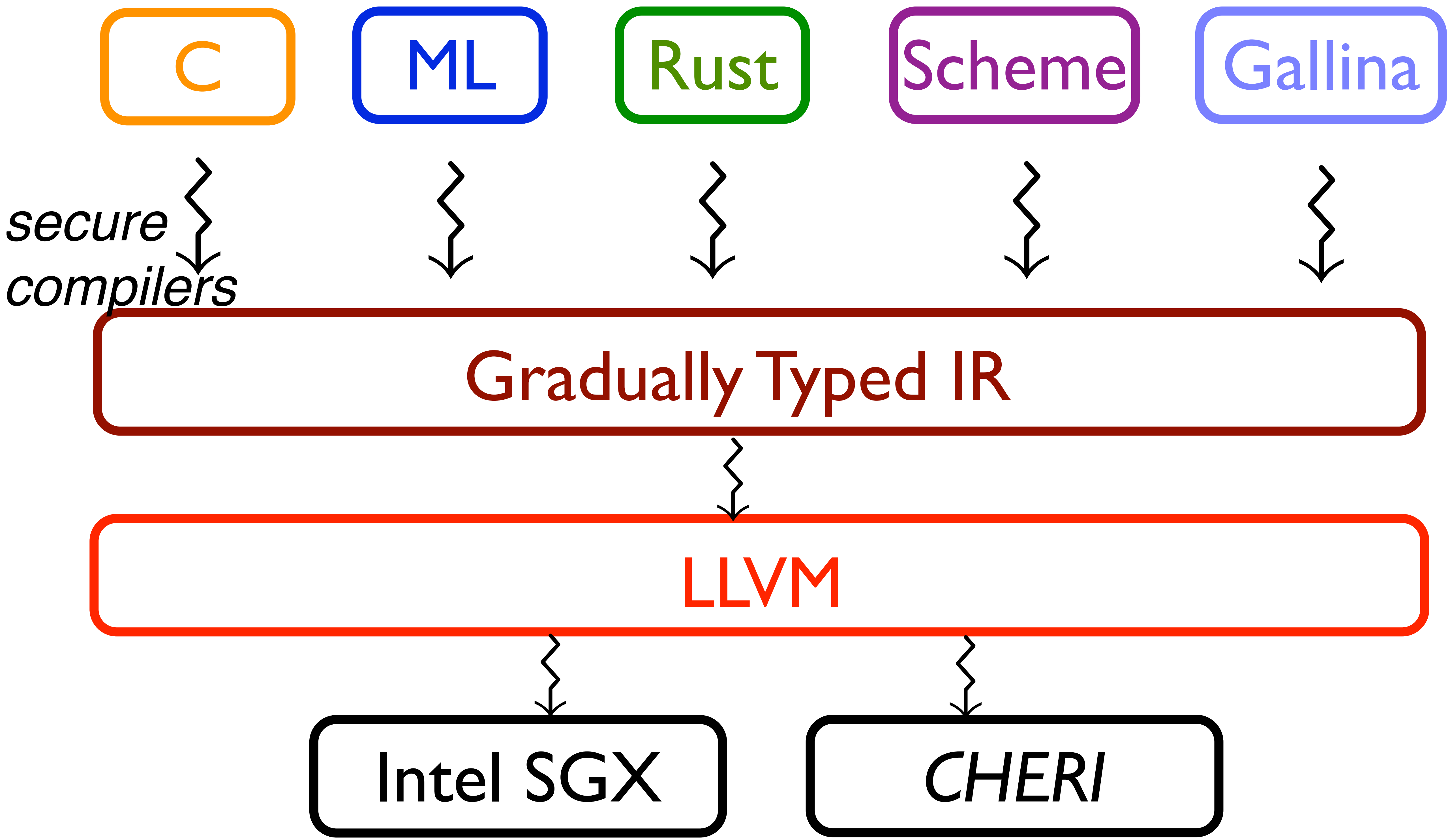
4. Capability Machines
   - C to CHERI-like capability machine: give calling convention that enforces well-bracketed control-flow and encapsulation of stack frames using local capabilities (subsequent work: linear capabilities); proved using logical relation *[Skorstengaard et al. ESOP'18, POPL'19]*

Secure Compilation: Open Problems

Secure Compilation: Open Problems

1. Need languages / DSLs that allow programmers to easily express security intent.
   - Compilers need to know programmer intent so they can *preserve* that intent (e.g., FaCT, a DSL for constant-time programming *[Cauligi et al. SecDev'17]*
2. Performant secure compilers
   - Static enforcement avoids performance overhead, could run on stock hardware; need richly typed compiler IRs
   - Dynamic enforcement when code from static/dynamic and safe/unsafe languages interoperates (e.g., h/w support)
   - Better integration of static and dynamic enforcement...

- Better integration of static and dynamic enforcement...



Secure Compilation: Open Problems

3. Preserve (weaker) security properties than contextual equiv.
   - Full abstraction may preserve too many incidental/ unimportant equivalences and has high overhead for dynamic enforcement
4. Security against side-channel attacks
   - Requires reasoning about side channels in source language, which is cumbersome. Can DSLs help?
   - *Correctness-Security Gap in Compiler Optimizations [D'Silva et al. LangSec'15]*. Make compilers aware of programmers' security intent to take into account for optimizations.

Secure Compilation: Open Problems

5. Cryptographically enforced secure compilation
    - e.g., Obliv-C ensures memory-trace obliviousness using garbled circuits, but no formal proof that it is secure
6. Concurrency (beyond message-passing, targeting untyped multi-threaded assembly)
7. Easier proof techniques and reusable proof frameworks (trace-based techniques, back-translation, logical relations, bisimulation)

Final Thoughts

It's an exciting time to be working on secure compilation!

- Numerous advances in the last decade, in PL/formal methods and systems/security.

- For performant secure compilers, will need to integrate static and dynamic enforcement techniques, and provide programmers with better languages for communicating their security intent to compilers.