Compositional Compiler Verification & Secure Compilation

Amal Ahmed
Northeastern University
Compiler Correctness

\[ s \leadsto t \implies s \approx t \]

- \( s \leadsto t \): compiles to
- \( s \approx t \): 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.

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

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

https://cakeml.org/
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
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!

\[ P_s \rightarrow e_s \rightarrow e_t \rightarrow \text{low-level libraries} \]
Problem: Whole-Program Assumption

Correct compilation guarantee only applies to whole programs!

$P_s \leadsto e_s \leadsto P_t \leadsto e_t$

from different compiler & source lang.
“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

\[ s \rightsquigarrow t \iff s \approx t \]

expressed how?
Whole-Program Compiler Correctness

\[ P_s \rightsquigarrow P_t \implies P_s \approx P_t \]

CompCert

\[
\begin{align*}
P_s & \rightsquigarrow \ldots \rightsquigarrow P_s^i \rightsquigarrow P_s^{i+1} \rightsquigarrow \ldots \\
\mid & \ R \quad \mid \ R \quad \mid \ R \\
P_t & \rightsquigarrow \ldots \rightsquigarrow P_t^j \rightsquigarrow \ast P_t^{j+n} \rightsquigarrow \ldots
\end{align*}
\]
Whole-Program Compiler Correctness

\[ P_s \leadsto P_t \implies P_t \sqsubseteq P_s \]

\[ \forall n. \quad P_t \xrightarrow{n} P_t' \implies \]

\[ \exists m. \quad P_s \xrightarrow{m} P_s' \land T_t \simeq T_s \]
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

expressed how?

$e_s \approx e_T$
“Compositional” Compiler Correctness

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?

$e_S \approx e_T$

expressed how?
"Compositional" Compiler Correctness

If we want to verify realistic compilers…

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

- 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
What we can link with

- Nothing
- Same compiler
- Different compiler, same S
- Compiled from different language R
- Compiled from very different language 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

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

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

$e_s \approx e_T$
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)

\[ x : \tau' \vdash e_s : \tau \rightsquigarrow e_t \quad \implies \quad x : \tau' \vdash e_s \simeq e_t : \tau \]

\[ \forall e'_s, e'_t. \vdash e'_s \simeq e'_t : \tau' \quad \implies \quad \vdash e_s[e'_s/x] \simeq e_t[e'_t/x] : \tau \]
Cross-Language Relation (Pilsner)

Have $x : \tau' \vdash e_s \simeq e_t : \tau$
Cross-Language Relation (Pilsner)

Have \( x : \tau' \vdash e_s \simeq e_t : \tau \)

Does the compiler correctness theorem permit linking with \( e'_t \)?
Cross-Language Relation (Pilsner)

Have \( x : \tau' \vdash e_s \simeq e_t : \tau \)

\[ \forall e'_s, e'_t. \vdash e'_s \simeq e'_t : \tau' \quad \longrightarrow \quad \vdash e_s[e'_s/x] \simeq e_t[e'_t/x] : \tau \]
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

A compiler that supports horizontal compositionality, in the sense depicted in the top left of Figure 1, should be related to the composition of $S \approx T$.

But how do we characterize the type of linking that correct compilers support?

5.1 Source-independent Linking

We draw a distinction between horizontal compositionality for compilers and what we are to specify compiler correctness between a nonterminating STLC and an SECD machine.

Should support became popular after [Benton and Hur 2009] used a source-target logical relation to be a useful means of proving correctness of compiler transformations (e.g., [Minamide et al].

The type of compositionality supported by logical relations, a technique that has long been known to a tiny fraction of the modules, and inappropriate for most changes to programs will only a tiny fraction of the modules be linked with a target term $e'$.

All real compilers must support linking. Compiling whole programs is both wildly ineffectual for high-level languages which nearly universally link against libraries written in lower-level languages.

What the phrase, Horizontal Compositionality, should mean, in a semantic sense is what the top left quadrant in Figure 1 depicts: $S \approx T$.

Source-independent Linking. This is in the top half of Figure 1.
Linking
All real compilers must support linking. Compiling whole programs is both widely ine...
Source-Independent Linking

\[ S \simeq T \]

\[ e_s \]

\[ e_t \]

\[ e'_t \]

\[ e' \]

A compiler that supports horizontal compositionality, in the sense depicted in top left in Figure 1,

\[ T \]

\[ S \]

\[ \Rightarrow \]

\[ 0 \]

\[ S \]

\[ T \]

\[ ? \approx T \]

\[ e_t \]

\[ e'_t \]

should support became popular after [Benton and Hur 2009] used a source-target logical relation

\[ \text{Vertical Compositionality} \]

\[ \text{Transitivity} \]

that if we have a source component

\[ \text{Horizontal Compositionality} \]

\[ \text{Source-independent Linking} \]

But how do we characterize the type of linking that correct compilers support?

\[ e_s \rightarrow e'_t \]

\[ ? \approx T \]

\[ e_t \]

\[ e'_t \]

We draw a distinction between Horizontal Compositionality for compilers and what we are

\[ \text{Hole in compositionality} \]

\[ \text{Transitivity} \]

\[ T \]

\[ S \]

\[ 0 \]

\[ S \]

\[ T \]

\[ ? \approx T \]

\[ e_t \]

\[ e'_t \]

most changes to programs will only a

\[ \Rightarrow \]

\[ 0 \]

\[ S \]

\[ T \]

\[ e_t \]

\[ e'_t \]

\[ ? \approx T \]

\[ e_t \]

\[ e'_t \]

Inappropriate for

\[ e_t \]

\[ e'_t \]

All real compilers must support linking. Compiling whole programs is both wildly ine

\[ S \]

\[ T \]

\[ 0 \]

\[ S \]

\[ T \]

\[ ? \approx T \]

\[ e_t \]

\[ e'_t \]

\[ e_t \]

\[ e'_t \]

We believe that the

\[ T \]

\[ S \]

\[ 0 \]

\[ S \]

\[ T \]

\[ ? \approx T \]

\[ e_t \]

\[ e'_t \]

\[ e_t \]

\[ e'_t \]
What we can link with

nothing  \(\rightarrow\) same compiler  \(\rightarrow\) diff compiler, same S  \(\rightarrow\) compiled from diff lang R  \(\rightarrow\) compiled from very diff R

CompCert

SepCompCert  \(\rightarrow\) Kang et al.'16

Pilsner  \(\rightarrow\) Neis et al.'15

Compositional CompCert  \(\rightarrow\) Stewart et al.'15

Multi-language ST  \(\rightarrow\) Perconti-Ahmed'14
Correct Compilation of Components?

expressed how?

\[ e_s \approx e_T \]
Correct Compilation of Components?

Expressed how?

$e_s \approx e_T$
Correct Compilation of Components?

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. POPL’15]

• Language-independent linking

Figure 2. Interaction semantics interface. The types $G$ (global environment), $C$ (core state), and $M$ (memory) are parameters to the interface. $F$ 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  \[Stewart\ \textit{et al.}\ POPL'15\]

• Language-independent linking

• \textbf{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.

Specify semantics of source-target interoperability:

\[ \mathcal{ST} e_t \quad \mathcal{T S} e_s \]

Multi-language semantics: *a la* Matthews-Findler ’07
Approach: Source-Target Multi-lang.

Specify semantics of source-target interoperability:

$\mathcal{ST} e_t \gg \mathcal{T} S e_s$

Multi-language semantics: 
ala Matthews-Findler '07

[Perconti-Ahmed ESOP'14]
Approach: Source-Target Multi-lang.

\[ TS(e_s (ST e'_t)) \approx_{ctx} e_t e'_t \]
Approach: Source-Target Multi-lang. [Perconti-Ahmed ESOP’14]

\[ e_s \approx e_T \quad \text{def} \quad e_s \approx^{\text{ctx}} ST e_T \]
Multi-Language Semantics Approach
Multi-Language Semantics Approach
Multi-Language Semantics Approach
Multi-Language Semantics Approach
Multi-Language Semantics Approach

Compiler Correctness

\( e_S \approx^{ctx} S I e_I \)

\( e_I \approx^{ctx} I T e_T \)
Multi-Lang. Approach: Multi-pass

Compiler Correctness

\[ e_S \approx^{\text{ctx}} SI e_I \]

\[ e_I \approx^{\text{ctx}} IT e_T \]
Multi-Lang. Approach: Multi-pass ✔

Compiler Correctness

\[ e_S \approx_{ctx} S I e_I \]

\[ S I e_I \approx_{ctx} S I (I T e_T) \]
Multi-Lang. Approach: Multi-pass ✓

Compiler Correctness

\[ e_S \approx^{ctx} SI e_I \]

\[ SI e_I \approx^{ctx} SI(II e_T) \]

\[ e_S \approx^{ctx} SI(II e_T) e_T \]
Multi-Lang. Approach: Linking
Multi-Lang. Approach: Linking

\[ e_s \quad \text{and} \quad SIT e'_t \]

\[ e_t \quad \text{and} \quad e'_t \]
Compiler Correctness: F to TAL
Combined language **FCAT**

- Boundaries mediate between $\tau$ & $\tau^C$, $\tau$ & $\tau^A$, $\tau$ & $\tau^T$

- Operational semantics
  
  $\text{CF}^\tau e \longrightarrow^* \text{CF}^\tau v \longrightarrow v$

  $\tau\text{FC}^e \longrightarrow^* \tau\text{FC}^v \longrightarrow v$

- Boundary cancellation
  
  $\tau\text{FCCF}^\tau e \approx^{ctx} e : \tau$

  $\text{CF}^\tau\tau\text{FC}^e \approx^{ctx} e : \tau^C$

[Perconti-Ahmed ESOP’14]

[Patterson et al. PLDI’17]
Interoperability: \( \textbf{F} \) and \( \textbf{C} \)

\[ \mathcal{C} \mathcal{F}^{\text{int}}(n) \quad \leftrightarrow \quad n \]

\[ \text{int} \mathcal{F} \mathcal{C}(n) \quad \leftrightarrow \quad n \]
Interoperability: \( \textbf{F} \) and \( \textbf{C} \)

\[
(\tau \rightarrow \tau')^\text{C} = \exists \beta. \langle ((\beta, \tau^\text{C}) \rightarrow \tau'^\text{C}), \beta \rangle
\]

\[
\text{CF}^\tau \rightarrow \tau' \text{v} \quad \mapsto \quad \text{pack}\langle \text{unit}, \langle \text{v}, () \rangle \rangle \text{ as } \exists \beta. \langle ((\beta, \tau^\text{C}) \rightarrow \tau'^\text{C}), \beta \rangle
\]

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

\[
\tau \rightarrow \tau' \text{FC} \text{v} \quad \mapsto \quad \lambda(x: \tau). \tau' \text{FC}(\text{unpack } \langle \beta, y \rangle = \text{v} \text{ in } \pi_1(y) \pi_2(y) \text{CF}^{\tau} x)
\]
Challenges

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

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

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, e.g., $AC\langle v_1, v_2 \rangle$

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, e.g., $AC\langle v_1, v_2 \rangle$

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

\[ e : \mathcal{T} \sim \rightarrow e \]
What is a component in TAL?

\[ e : \mathcal{T} \sim \sim e \]  

basic block = instruction sequence

\[ e ::= (I, H) \]

Heap with basic blocks
Equivalence of components in TAL?

\[ e : \tau \sim\rightarrow e \]

Related inputs: \( e_1 \) → \( e_2 \)
Equivalence of components in TAL?

\[ \varepsilon : \mathcal{T} \sim \varepsilon \]

related inputs \[\rightarrow\] related outputs

\[ e_1 \]

\[ e_2 \]
Equivalence of components in TAL?

\[ e : \tau \sim \Rightarrow e \]

related inputs → \( e_1 \) → related outputs

related outputs → \( e_2 \) → related inputs
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]
# CompCert vs. Multi-language

<table>
<thead>
<tr>
<th>Transitivity</th>
<th>structured simulation</th>
<th>all passes use multi-lang $\approx_{ctx}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check okay-to-link-with</td>
<td>satisfies CompCert memory model</td>
<td>satisfies expected type (translation of source type)</td>
</tr>
<tr>
<td>Requires uniform memory model across compiler IRs</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Allows linking with behavior inexpressible in S</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Proving Transitivity

nothing  \(\Downarrow\) same compiler

\textit{CompCert}

\textit{SepCompCert}  
\textit{Kang et al.'16}

Transitivity requires effort / engineering

\textit{Pilsner}  
\textit{Neis et al.'15}

\textit{Compositional CompCert}  
\textit{Stewart et al.'15}

\textit{Multi-language ST}  
\textit{Perconti-Ahmed'14}
Vertical
Compositionality

Transitivity
Vertical Compositionality
Vertical Compositionality

e_S \approx SI \approx IT \approx ST

e_T
Transitivity

CompCompCert & Multi-lang

$e_S \Rightarrow \approx SIT \Rightarrow e_I \Rightarrow \approx SIT \Rightarrow e_T \Rightarrow \approx SIT$
<table>
<thead>
<tr>
<th>Horizontal Compositionality</th>
<th>Source-Independent Linking</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Pilsner</em> Neis et al.’15</td>
<td>Compositional CompCert</td>
</tr>
<tr>
<td></td>
<td>Stewart et al.’15</td>
</tr>
<tr>
<td></td>
<td>Multi-language ST</td>
</tr>
<tr>
<td></td>
<td>Perconti-Ahmed’14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vertical Compositionality</th>
<th>Transitivity</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Pilsner</em> Neis et al.’15</td>
<td>Compositional CompCert</td>
</tr>
<tr>
<td></td>
<td>Stewart et al.’15</td>
</tr>
<tr>
<td></td>
<td>Multi-language ST</td>
</tr>
<tr>
<td></td>
<td>Perconti-Ahmed’14</td>
</tr>
</tbody>
</table>
To Understand if Theorem is Correct...

Pilsner
Neis et al.’15

Compositional CompCert
Stewart et al.’15

Multi-language ST
Perconti-Ahmed’14

- source-target PILS

- interaction semantics & structured simulations

- 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?

\[ e_t', \varphi \in \mathcal{L} \leftarrow \text{linking set} \]
Generic CCC Theorem?

\[
e_S \approx e_T
\]

expressed how?

\[
e'_t, \varphi \in \mathcal{L} \leftarrow \text{linking set}
\]
Generic CCC Theorem?

We explicitly define a source-target linking medium.

\( e_s \approx e_T \)

expressed how?

\( e_s \in \hat{S} \)

linking set

\( e_t, \varphi \in \mathcal{L} \leftarrow \text{linking set} \)
Generic CCC Theorem?

- $e_s$ (source term)
- $e_t$ (target term)
- $\hat{S}$ (source-target linking medium)
- $L$ (linking set)

$e_s \approx e_T$

expressed how?

lift (from $L$ to $\hat{S}$)

\[ e_s \in \hat{S} \]

\[ e_t \in L \]

\[ e'_t, \phi \in L \]

linking set
What is left is our lifting function, but this too becomes simple, because of the SepCompCert theorem. There are several existing approaches to compositional compiler correctness, all of which aim to give meaning to the execution of...
There are several existing approaches to compositional compiler correctness, all of which aim to do this than was previously possible. These are not the only pieces of prior work on compositional this section, we will show how

4 EXISTING APPROACHES

\[ e_S \approx e_T \]

\( e_t', \varphi \in L \)

\( e_t \Rightarrow e_T \)

\( \hat{S} \Rightarrow S \)

\( \exists \uparrow \)

\( e_s \)

\( \hat{S} \times S \)

\( \hat{S} \times S \)

\( \hat{S} \times S \)

\( \hat{S} \times S \)
We can see that in this case, the language will also show how further approaches relate to this result.

The approach taken by [1], Vol. 1, No. 1, Article 1. Publication date: September 2017.

We describe in detail each point, and

\[ e_s \approx e_T \]

…the lift is inverse of “compile” on compiler output
Generic CCC Theorem (Formally)

\[ \exists \uparrow. \forall e_S \in S. \forall (e_T, \varphi) \in \mathcal{L}. \]

\[ e_T \quad T \bowtie_T \quad C_T^S(e_S) \quad T \sqsubseteq \tilde{S} \quad \uparrow(e_T, \varphi) \quad \tilde{S} \bowtie_S \quad e_S \]

...and “lift” is inverse of “compile” on compiler output

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

\[ (\forall c_T. \quad c_T \quad T \bowtie_T \quad e_T \quad T \sqsubseteq_T \quad c_T \quad T \bowtie_T \quad C_T^S(e_S)) \quad \Rightarrow \]

\[ (\forall c_S. \quad c_S \quad S \bowtie_S \quad \uparrow(e_T, \varphi) \quad \tilde{S} \sqsubseteq_S \quad c_S \quad S \bowtie_S \quad e_S) \]

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

\[ \mathcal{L} = \{(e_T, \varphi) \mid \varphi = \text{source component } e_S \text{ & proof that } e_S \simeq e_T\} \]

\[ \overline{\text{unchanged source language } S} \]

\[ \text{unchanged source language linking} \]

\[ \uparrow(\cdot) \uparrow(e_T, (e_S, _)) = e_S \]
CCC Properties

Implies whole-program compiler correctness & separate compilation correctness
Can be instantiated with different formalisms…

Multi-language ST

\[ \mathcal{L} = \{(e_T, _) \mid \text{where } e_T \text{ is any target component}\} \]

\( \hat{S} \) source-target multi-language ST

\( \hat{\times}_S \ e \ ST \times_{ST} \ e_S \)

\( \uparrow(\cdot) \ \uparrow(e_T, _) = ST(e_T) \)
### CCC Properties

Implies **whole-program compiler correctness & separate compilation correctness**

Can be instantiated with different formalisms…

<table>
<thead>
<tr>
<th>Compositional CompCert</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{L} )</td>
</tr>
<tr>
<td>( \widehat{\mathcal{S}} )</td>
</tr>
<tr>
<td>( \widehat{\mathcal{S}} \triangleright_{S} )</td>
</tr>
<tr>
<td>( \Uparrow(\cdot) )</td>
</tr>
</tbody>
</table>
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 \( \text{lift} \uparrow \) 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
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 \( \uparrow \) 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

- 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

- 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

- 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

Language specifications are incomplete!
Don’t account for linking

Target
The Way Forward...
Rethink PL Design with *Linking Types*

- **ML** (C FFI)
- **Rust** (unsafe)
- **Java** (JNI)

*escape hatches*
Rethink PL Design with *Linking Types*

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

**escape hatches**

- ML
- Rust
- Java

*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.
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

ML → Rust → Gallina

Type-preserving
fully abstract compilers

Richly Typed Target
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 $\Gamma \vdash e_1 : \tau \leadsto e_1$ and $\Gamma \vdash e_2 : \tau \leadsto e_2$

Given:
- No $C_S$ can distinguish $e_1$, $e_2$

Show:
- Given arbitrary $C_T$
- It cannot distinguish $e_1$, $e_2$

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 $\tau$ and $\tau^+$

   - 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 F\(\omega\))
        [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 $F_\omega$)
        [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

Source: STLC + $\mu$ types

Target: System F + $\exists$ types + $\mu$ 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

[New et al. ICFP’16]
Dynamic Secure Compilation

e1

\[
\sim_{ctx}^{S}
\]

\[
\sim_{ctx}^{T}
\]

e2

compile

e1

compile

e2
Dynamic Secure Compilation

   • 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]
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...

- C
- ML
- Rust
- Scheme
- Gallina

Gradually Typed IR

LLVM

Intel SGX

CHERI
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.