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!

$P_s \rightarrow e_s \rightarrow e_t$ from different compiler & source lang. low-level libraries
“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

\[ s \sim s \implies t \implies s \approx t \]

- \( s \sim s \implies t \) compiles to
- \( s \approx t \) same behavior
Compiler Correctness

\[ s \sim t \quad \iff \quad s \approx t \]

expressed how?
Whole-Program Compiler Correctness

$P_s \leadsto P_t \quad \implies \quad P_s \approx P_t$

expressed how? “closed” simulations

CompCert

$P_s \quad \mapsto \ldots \mapsto P_s^i \mapsto P_s^{i+1} \mapsto \ldots$

$\vdash R \quad \vdash R \quad \vdash R$

$P_t \quad \mapsto \ldots \mapsto P_t^j \mapsto * P_t^{j+n} \mapsto \ldots$
Whole-Program Compiler Correctness

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

\[ \forall n. P_t \xrightarrow{n} P_t' \implies \exists m. P_s \xrightarrow{m} P_s' \land T_t \simeq T_s \]

behavior refinement
Correct Compilation of Components?

$e_s \approx e_T$

expressed how?
“Compositional” Compiler Correctness

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

Behavior expressible in S?

expressed how?

$e_s \approx e_T$
“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 $e_S \approx e_T$

• 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  \(\rightarrow\) 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
What we can link with

nothing

same compiler

diff compiler, same S

compiled from diff lang R

compiled from very diff R

CompCert

SepCompCert

Kang et al.’16

Pilsner

Neis et al.’15

Compositional CompCert

Stewart et al.’15

Multi-language ST

Perconti-Ahmed’14
Approach: Separate Compilation

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

\[ e_S \approx e_T \]

Compiling ML-like langs:

Logical relations

- No transitivity!

Parametric inter-language simulations (PILS)

- Prove transitivity, but requires effort!
Cross-Language Relation (Pilsner)

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

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

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

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

\[
\forall e'_s, e'_t \vdash e'_s \simeq e'_t : \tau' \quad \implies \quad \vdash e'_s[e'_s/x] \simeq e'_t[e'_t/x] : \tau
\]

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

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

Horizontal Compositionality

\[ S \approx T \]

\[ T \approx S \]

\[ S \approx T \]

Vertical Compositionality

\[ e_s \]

\[ e'_s \]

\[ e_t \]

\[ e'_t \]

Linking

\[ S \approx T \]
We believe that the idea that horizontal compositionality is the form of linking property that compositional compilers should support became popular after [Benton and Hur 2009] used a source-target logical relation to prove the correctness of closure conversion. We believe that the type of compositionality supported by logical relations, a technique that has long been known to be a useful means of proving correctness of compiler transformations (e.g., [Minamide et al.]), should mean, in a semantic sense is what the top left quadrant in Figure 1 depicts: Source-independent Linking. This is in the top half of Figure 1. What the phrase, Horizontal Compositionality, should mean, in a semantic sense is what the top left quadrant in Figure 1 depicts: Source-independent Linking. This is in the top half of Figure 1. What the phrase, Horizontal Compositionality, should mean, in a semantic sense is what the top left quadrant in Figure 1 depicts: Source-independent Linking. This is in the top half of Figure 1. What the phrase, Horizontal Compositionality, should mean, in a semantic sense is what the top left quadrant in Figure 1 depicts: Source-independent Linking. This is in the top half of Figure 1. What the phrase, Horizontal Compositionality, should mean, in a semantic sense is what the top left quadrant in Figure 1 depicts: Source-independent Linking. This is in the top half of Figure 1. What the phrase, Horizontal Compositionality, should mean, in a semantic sense is what the top left quadrant in Figure 1 depicts: Source-independent Linking. This is in the top half of Figure 1. What the phrase, Horizontal Compositionality, should mean, in a semantic sense is what the top left quadrant in Figure 1 depicts: Source-independent Linking. This is in the top half of Figure 1. What the phrase, Horizontal Compositionality, should mean, in a semantic sense is what the top left quadrant in Figure 1 depicts: Source-independent Linking. This is in the top half of Figure 1. What the phrase, Horizontal Compositionality, should mean, in a semantic sense is what the top left quadrant in Figure 1 depicts: Source-independent Linking. This is in the top half of Figure 1. What the phrase, Horizontal Compositionality, should mean, in a semantic sense is what the top left quadrant in Figure 1 depicts: Source-independent Linking. This is in the top half of Figure 1. What the phrase, Horizontal Compositionality, should mean, in a semantic sense is what the top left quadrant in Figure 1 depicts: Source-independent Linking. This is in the top half of Figure 1. What the phrase, Horizontal Compositionality, should mean, in a semantic sense is what the top left quadrant in Figure 1 depicts: Source-independent Linking. This is in the top half of Figure 1. What the phrase, Horizontal Compositionality, should mean, in a semantic sense is what the top left quadrant in Figure 1 depicts: Source-independent Linking. This is in the top half of Figure 1.
What we can link with

- nothing
- same compiler
- diff compiler, same S
- compiled from diff lang R
- compiled from very diff R

CompCert

- SepCompCert
  Kang et al.’16

Pilsner
Neis et al.’15

Compositional CompCert
Stewart et al.’15

Multi-language ST
Perconti-Ahmed’14
Correct Compilation of Components?

Need a semantics of source-target interoperability:
- interaction semantics
- source-target multi-language
What we can link with

nothing

same compiler

diff compiler, same S

compiled from diff lang R

compiled from very diff R

CompCert

SepCompCert

Kang et al.’16

Pilsner

Neis et al.’15

Compositional CompCert

Stewart et al.’15

Multi-language ST

Perconti-Ahmed’14
Approach: Interaction Semantics

Compositional CompCert

[Stewart et al. ’15]

- Language-independent linking

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. $V$ 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

- 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:

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

Multi-language semantics: *a la* Matthews-Findler ’07

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

\[ \text{TS}(e_s (\text{ST} e'_t)) \approx^{\text{ctx}} e_t e'_t \]
Approach: Source-Target Multi-lang.

\[ e_s \approx e_T \quad \text{def} \]

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

Compiler Correctness

\[ e_S \approx^{ctx} S \| e_I \]

\[ e_I \approx^{ctx} I \| e_T \]
Multi-Lang. Approach: Multi-pass ✓

Compiler Correctness

\[ e_S \approx^{ctx} S I e_I \]

\[ S I e_I \approx^{ctx} S I (e_T e_T) \]

\[ e_S \approx^{ctx} S I T e_T \]
Multi-Lang. Approach: Linking

\[
TIS(s_t \circ (SIT e'_t)) \approx_{ctx} e_t e'_t
\]
Compiler Correctness: F to TAL

- $e_F$ ➜ Closure Conversion $\tau^C$
- $e_C$ ➜ Allocation $\tau^A$
- $e_A$ ➜ Code Generation $\tau^T$
- $e_T$
Combined language **FCAT**

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

[Perconti-Ahmed’14]
[Patterson et al.’17]
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 $\Leftrightarrow^\text{ctx}$

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

- nothing
- same compiler
- diff compiler, same S
- compiled from diff lang R
- compiled from very diff R

CompCert

- SepCompCert
  - Kang et al.'16
- Pilsner
  - Neis et al.'15
- Compositional CompCert
  - Stewart et al.'15

Allows linking with behavior inexpressible in S

Multi-language ST
- Perconti-Ahmed’14
Proving Transitivity

- nothing
- same compiler
- diff compiler, same $S$
- compiled from diff lang $R$
- compiled from very diff $R$

CompCert

- SepCompCert
  - Kang et al.’16

- Transitivity requires effort / engineering

- Pilsner
  - Neis et al.’15

- Compositional CompCert
  - Stewart et al.’15

- Multi-language ST
  - Perconti-Ahmed’14
Vertical Compositionality

\[ e_S \Rightarrow S \approx I \Rightarrow e_I \approx I \Rightarrow e_T \approx T \]

Transitivity

\[ e_S \Rightarrow S \approx T \Rightarrow e_T \]
Transitivity

CompCompCert & Multi-lang

e_S \approx SIT

\Rightarrow

e_T \approx SIT

\Rightarrow

e_S \approx SIT
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?**
What is left is our lifting function, but this too becomes simple, because of the SepCompCert approach taken by D. This is the reason for much of the simplification scales down to separate compilation. We can see that in this case, the language restriction that we only link with modules that were produced by the SepCompCert compiler. We do this than was previously possible. These are not the only pieces of prior work on compositional correctness, but they represent a suitable broad cross-section. We will expose various features of the approaches. We will then compare and contrast each method, showing how the CCC framework allows us to more easily give meaning to the execution of the theorem. In doing so, we will satisfy our CCC Theorem. In order to lift (from $T$ to $\hat{S}$), as it means that much of the rest of the source component is actually just the source language...
There are several existing approaches to compositional compiler correctness, all of which aim to satisfy our CCC Theorem. In doing so, we will expose various features of the approaches. We will give meaning to the execution of

\[ e_s \approx e_T \]

This is the reason as it means that much of the rest of the

\[ \exists \uparrow \]

\[ e_s \]

\[ \tilde{S} \times S \]

\[ \in \hat{S} \]

\[ T \times T \]

\[ e_t' \in \mathcal{L} \]

lift (from T to \( \hat{S} \))
Generic CCC Theorem

\[ \exists \uparrow. \forall e_S, e_T. e_S \rightsquigarrow e_T \implies \]

\[ \forall (e'_T, \varphi) \in \mathcal{L}. \ ok\bigtriangleup (e'_T, e_T) \implies \]

\[ e'_T \bigtriangleup_T e_T \bigtriangledown \overset{\sim}{S} \uparrow (e'_T, \varphi) \bigtriangleup_S e_S \]
Generic CCC Theorem

\[ \exists \uparrow. \forall e_S, e_T. e_S \rightsquigarrow e_T \implies \]

\[ \forall (e'_T, \varphi) \in \mathcal{L}. \text{ok}(e'_T, e_T) \implies \]

\[ e'_T \; T \boxleftarrow \; e_T \; T \unlhd \; \widehat{S} \; \uparrow(e'_T, \varphi) \; \widehat{S} \boxleftarrow \; S \; e_S \]

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

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

\[ e'_T \; T \unlhd \; e_T \implies \uparrow(e'_T, \varphi) \; \widehat{S} \unlhd \; S \; e_S \]
CCC Properties

Implies whole-program compiler correctness & correct separate compilation

Can be instantiated with different formalisms…
CCC with Pilsner

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

\( \hat{S} \) unchanged source language \( S \)

\( \hat{S} \times_S \) unchanged source language linking

\( \hat{S} \sqcap_S \) source language (whole program) observational equivalence

\( \uparrow(\cdot) \uparrow(e_T, (e_S, _)) = e_S \)
- CCC with Multi-language

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

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

\[ \hat{S} \times_S e \stackrel{\_}{\times}_{ST} e_S \]

\( \hat{S} \sqsubset_S \) run \( \hat{S} \) according to multi-lang ST, compare with running \( S \)

\[ \hat{\uparrow}(\_ ) \quad \hat{\uparrow}(e_T, \_ ) = ST(e_T) \]
Vertical Compositionality for Free

\[ \uparrow_{ST} = \uparrow_{SI} \circ \uparrow_{IT} \]

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

- 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

\[ e_s \approx e_T \]

Problem: programmer cannot reason at source level!

inexpressible in S
Fully Abstract Compilation?

Language specifications are incomplete!
Don’t account for linking
Rethink PL Design with Linking Types

Design linking types extensions that support safe interoperability with other languages.
Only need linking types extensions to interact with behavior inexpressible in your language.
PL Design, Linking Types, Compilers

- ML
- Rust
- Scheme
- Gallina

Fully abstract compilers
continuations
affine
fine-grained capabilities
modal types / type & effect

Typed IR
LLVM
PL Design, Linking Types, Compilers

Fully abstract compilers

ML

affine

pure

continuations

Typed IR

+ pure

+ dependent types

Rust

fine-grained

capabilities

Scheme

Gallina

LLVM
Linking Types

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