Correct and Secure Compilation for Multi-Language Software

Amal Ahmed
Northeastern University
Compiler Correctness

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

- \( s \leadsto t \): Compiles to
- \( s \approx t \): Same behavior
Semantics-preserving compilation

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

compiles to

same behavior
Range of Compiler Properties…

- Type-preserving compilation (90s)
- Semantics-preserving compilation (00s…)
  \[= \textit{Correct compilation}\]
- Fully abstract compilation
  \[= \textit{Equivalence-preserving and -reflecting}\]
  \[= \textit{Secure compilation}\]
- Security-preserving compilation
  - preserving “security types” vs. preserving noninterference
Compiler Verification

One of the “big problems” of computer science

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

• see Dave 2003: Compiler Verification: A Bibliography
Compiler Verification since 2006...

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

Lochbihler ’10: Verifying a compiler for Java threads.

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

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

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

Kumar et al.’14: CakeML: A verified implementation of ML
Why CompCert had such impact...

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

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

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

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

But the simplicity of the proof architecture comes at a price…
Problem: Whole-Program Assumption

Correct compilation guarantee only applies to whole programs!

CompCert’s … “formal guarantees of semantics preservation apply only to whole programs that have been compiled as a whole by [the] CompCert C [compiler]” (Leroy 2014)
Problem: Whole-Program Assumption

Correct compilation guarantee only applies to whole programs!

\[ P_s \Rightarrow e_s \Rightarrow e_t \]

\[ P_t \Rightarrow \]

from different compiler & source lang. low-level libraries
Why Whole Programs?

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

expressed how?
Why Whole Programs?

\[ P_s \leadsto P_t \quad \implies \quad P_s \approx P_t \]

expressed how?

CompCert

\[ P_s \leftarrow \ldots \leftarrow P_s^i \leftarrow P_{s+1}^i \leftarrow \ldots \]

\[ P_t \leftarrow \ldots \leftarrow P_t^j \leftarrow P_{t+j}^n \leftarrow \ldots \]
Proof composes per-pass simulations

\[ P_s \approx P_t \implies P_s \rightarrowtail \ldots \rightarrowtail P_s \rightarrowtail P_s^{i+1} \rightarrowtail \ldots \]

\[ \quad \parallel R_1 \parallel R_1 \parallel R_1 \]

\[ P_t \rightarrowtail \ldots \rightarrowtail P_t^{j} \rightarrowtail \ast P_t^{j+n} \rightarrowtail \ldots \]

\[ P_t \approx P_u \implies P_t \rightarrowtail \ldots \rightarrowtail P_t^{i} \rightarrowtail P_t^{i+1} \rightarrowtail \ldots \]

\[ \quad \parallel R_2 \parallel R_2 \parallel R_2 \]

\[ P_u \rightarrowtail \ldots \rightarrowtail P_u^{j} \rightarrowtail \ast P_u^{j+n} \rightarrowtail \ldots \]

\[ P_s \approx P_u \implies P_s \approx P_u \]
Why Whole Programs?

\[ P_s \leadsto P_t \quad \implies \quad P_s \approx P_t \]

“closed” simulations

CompCert

\[ P_s \longleftarrow \ldots \longleftarrow P_s^i \longleftarrow P_s^{i+1} \longleftarrow \ldots \]

\[ P_t \longleftarrow \ldots \longleftarrow P_t^j \longleftarrow \ast P_t^{j+n} \longleftarrow \ldots \]
Correct Compilation of Components?

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

Behavior expressible in S?
Correct Compilers, Multi-language SW

Definition should:

• permit **linking** with target code of arbitrary provenance

• support verification of **multi-pass** compilers

\[ e_S \approx e_T \]
Plan

• Survey the literature: how to express $e_S \approx e_T$
  - "compositional" compiler correctness
    = correct compilation of components
Compositional Compiler Correctness

\[ e_S \approx e_T \]

Dictates:

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

• Survey the literature: how to express $e_S \approx e_T$

• How does the choice affect:
  - what we can link with
  - how we check if some $e_t'$ is okay to link with
  - effort required to prove transitivity
  - effort required to have confidence in theorem statement

• How to support linking with code from very different $R$
Plan

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

nothing

same compiler

diff compiler, same $S$

compiled from diff lang $R$

compiled from very diff $R$

CompCert

SepCompCert

Kang et al.'16

Pilsner

Neis et al.'15

Compositional CompCert

Stewart et al.'15

Multi-language ST

Perconti-Ahmed’14
What we can link with

nothing

same compiler

diff compiler, same S

compiled from diff lang R

compiled from very diff R

CompCert

SepCompCert

Kang et al.’16

Pilsner

Neis et al.’15

Compositional CompCert

Stewart et al.’15

Multi-language ST

Perconti-Ahmed’14
Approach: Separate Compilation (C)

SepCompCert
[Kang et al. ’16]
Approach: Separate Compilation (C)

SepCompCert
[Kang et al. ’16]

Level A correctness

End-to-end

\[
\forall i \in \{1 \ldots n\}. C(s_i.c) = t_i.asm \\
\quad s = \text{load}(s_1.c \circ \ldots \circ s_n.c) \quad t = \text{load}(t_1.asm \circ \ldots \circ t_n.asm) \\
\quad \text{Behav}(s) \supseteq \text{Behav}(t)
\]
Approach: Separate Compilation (C)

SepCompCert
[Kang et al. ’16]

Level B correctness: omit some RTL optimizations
Approach: Separate Compilation (C)

SepCompCert

[Kang et al. ’16]

Level B correctness: omit some RTL optimizations

End-to-end

\[\forall i \in \{1 \ldots n\}, C_i(s_i.c) = t_i.asm\]

\[
s = \text{load}(s_1.c \circ \ldots \circ s_n.c) \quad t = \text{load}(t_1.asm \circ \ldots \circ t_n.asm)
\]

\[\text{Behav}(s) \supseteq \text{Behav}(t)\]
What we can link with

nothing

same compiler

diff compiler, same S

compiled from diff lang R

compiled from very diff R

CompCert

SepCompCert

Kang et al.'16

Pilsner

Neis et al.'15

Compositional CompCert

Stewart et al.'15

Multi-language ST

Perconti-Ahmed'14
Approach: Cross-Language Relations

Cross-language relation

\[ e_S \approx e_T \]

Compiling ML-like langs:

- Logical relations
  - [Benton-Hur ICFP’09]
  - [Hur-Dreyer POPL’11]

- Parametric inter-language simulations (PILS)
  - [Neis et al. ICFP’15]
Case Study: Closure Conversion

- Typed Closure Conversion

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

[on board]
Cross-Language Relation: Problem 1

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

Cross-language logical relation

\[
\forall e'_s, e'_s. \vdash e'_s \simeq e'_t : \tau' \quad \implies \quad \vdash e_s[e'_s/x] \simeq e_t[e'_t/x] : \tau
\]
Cross-Language Relation: Problem 1

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

Does the compiler correctness theorem permit linking with \( e'_t \)?
Cross-Language Relation: Problem 1

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

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

- Need to come up with \( e'_s \)
  -- not feasible in practice!
- Cannot link with \( e'_t \)
  whose behavior cannot be expressed in source.
Cross-Language LR: Problem 2

\[ \begin{align*}
  e_S & \cong e_I \\
  e_I & \cong e_T \\
  e_I & \cong e_T \\
  e_S & \cong e_T
\end{align*} \]
Transitivity for single-lang. logical relation?

\[ e_1 \approx e_2 \]
\[ e_2 \approx e_3 \]
\[ e_1 \approx e_3 \]
Cross-Language LR: Problem 2

Transitivity for single-lang. logical relation:

\[ e \approx e' \iff e \approx^{ctx} e' \]
Cross-Language LR: Problem 2

For langs with refs:
- step-indexed Kripke LR
- PILS

\[ e_S \simeq e_I \]
\[ e_I \simeq e_T \]
\[ \Rightarrow e_S \simeq e_T \]
Cross-Language LR: Problem 2

For langs with refs:
- step-indexed Kripke LR
- PILS but lots of effort

\[ e_S \simeq e_I \]
\[ e_I \simeq e_T \]
\[ e_S \simeq e_T \]
PILS: Problem 1 remains

\[ \vdash e_s' \simeq e_t' : \tau' \]

- Need to come up with \( e_s' \) -- not feasible in practice!
- Cannot link with \( e_t' \) whose behavior cannot be expressed in source.
Need a New Approach...

- that works for multi-pass compilers
- that allows linking with target code of arbitrary provenance
What we can link with

nothing  ——>  same compiler  ——>  diff compiler, same S  ——>  compiled from diff lang R  ——>  compiled from very diff R

CompCert

SepCompCert  
Kang et al.’16

Pilsner  
Neis et al.’15

Compositional CompCert  
Stewart et al.’15

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

\[ e_s \approx e_T \]

expressed how?

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

nothing

same compiler

diff compiler, same $S$

compiled from diff lang $R$

compiled from very diff $R$

CompCert

SepCompCert

Kang et al.'16

Pilsner

Neis et al.'15

Compositional CompCert

Stewart et al.'15

Multi-language ST

Perconti-Ahmed'14
Approach: Interaction Semantics

Compositional CompCert

[Stewart et al. ‘15]

- Language-independent linking

![Diagram](image)

Semantics ($G, C, M : \text{Type}$) : Type $\triangleq$

- initial_core : $G \to V \to \text{list } V \to \text{option } C$
- at_external : $C \to \text{option } (F \times \text{list } V)$
- after_external : $\text{option } V \to C \to \text{option } C$
- halted : $C \to \text{option } V$
- corestep : $G \to C \to M \to C \to M \to \text{Prop}$

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

Semantic representation of contexts code can link with.
Approach: Interaction Semantics

Compositional CompCert

[Stewart et al. ’15]

- Language-independent linking
  - uniform CompCert memory model across all languages
  - not clear how to scale to richer source langs (e.g., ML), compilers with different source/target memory models

- Structured simulation: support rely-guarantee relationship between the different languages while retaining vertical compositionality
  - transitivity relies on compiler passes performing restricted set of memory transformations
What we can link with

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

CompCert

SepCompCert  Kang et al.’16

Pilsner  Neis et al.’15

Compositional CompCert  Stewart et al.’15

Multi-language ST  Perconti-Ahmed’14
Approach: Source-Target Multi-lang.

Specify semantics of source-target interoperability:

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

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

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

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

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

Compiler Correctness

\[ \text{es} \approx^{\text{ctx}} S\text{Ie}_{\text{I}} \]

\[ \text{e}_{\text{I}} \approx^{\text{ctx}} I\text{T}e_{\text{T}} \]
Multi-Lang. Approach: Multi-pass

Compiler Correctness

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

\[ S I e_I \approx_{ctx} S I (I T e_T e_T) \]

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

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

- Closure Conversion
- Allocation
- Code Generation
Combined language \textbf{FCAT}

- Boundaries mediate between $\tau$ & $\tau^C$, $\tau$ & $\tau^A$, $\tau$ & $\tau^T$
Compiler Correctness: F to TAL

Closure Conversion

Allocation

Code Generation

[Perconti-Ahmed ESOP’14]
CompCompCert vs. Multi-language

Transitivity:
- structured simulations
- all passes use multi-lang \( \approx^{ctx} \)

Check okay-to-link-with:
- satisfies CompCert memory model
- satisfies expected type (translation of source type)

Contexts:
- semantic representation
- syntactic representation

Requires uniform memory model across compiler IRs?
- yes
- no
Case Study: Closure Conversion

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

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

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

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

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

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

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

\[ H, \ell \mapsto \langle v \rangle; \langle \tau \rangle CA \ell \mapsto H, \ell \mapsto \langle v \rangle; \langle v \rangle \]

\[ H; AC \langle \tau \rangle \langle v \rangle \mapsto H, \ell \mapsto \langle v \rangle; \ell \]

Allocate a new location for tuple
Challenges

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

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

A+T: What is 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. ’17]
What is a component in TAL?

$e : \mathcal{T} \rightsquigarrow e$

Instruction Sequence

$e ::= (I, H)$

Heap with basic blocks
Typing TAL Components

\[
\Psi; \Delta; \chi; \sigma; q \vdash e : \tau; \sigma'
\]

return

marker

\[
q ::= \epsilon \mid r \mid i
\]
Basic blocks

\[ e \]

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

\[ \text{code}[\Delta]\{\chi; \sigma\}^q.I : \forall[\Delta].\{\chi; \sigma\}^q \]
Equivalence of $\mathbf{T}$ Components: Tricky!

Logical relations: related inputs to related outputs

$\mathcal{V}[\tau_1 \to \tau_2] = \{(W, \lambda x.e_1, \lambda x.e_1) | \ldots\}$

$\mathcal{H}^\mathcal{V}[\forall[\Delta].\{\chi; \sigma\}^q] = \{(W, \text{code}[\Delta]\{\chi; \sigma\}^q.I_1, \text{code}[\Delta]\{\chi; \sigma\}^q.I_2) | \ldots\}$
Equivalence of $\mathbf{T}$ Components

Logical relations: related inputs to related outputs

$$\forall[\tau_1 \rightarrow \tau_2] = \{(W, \lambda x. e_1, \lambda x. e_1) | \ldots\}$$

$$\forall[\Delta].\{(\chi; \sigma)^q\} = \{(W, \text{code}[\Delta]\{(\chi; \sigma)^q.I_1, \text{code}[\Delta]\{(\chi; \sigma)^q.I_2) | \ldots\}$$

related inputs $\rightarrow$ e_1

related outputs $\rightarrow$

related outputs $\rightarrow$ e_2
Ongoing: Multi-lang. Approach

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

nothing  

same compiler

diff compiler, same S  

compiled from diff lang R  

compiled from very diff R

CompCert

*SepCompCert*  
*Kang et al.’16*

Transitivity requires effort / engineering

Pilsner  
*Neis et al.’15*

Compositional CompCert  
*Stewart et al.’15*

Multi-language ST  
*Perconti-Ahmed’14*
Horizontal Compositionality

$e_s \hookrightarrow e'_t$ → Behavior inexpressible in S?
Horizontal / Vertical Compositionality

CompCert

- SepCompCert
  Kang et al.'16

- Pilsner
  Neis et al.'15

- Compositional CompCert
  Stewart et al.'15

- Multi-language ST
  Perconti-Ahmed’14

Vertical Compositionality

- same compiler
  compiled from diff lang R
  very diff R

- diff compiler, same S
  compiled from very diff R

- nothing
- It’s about principled language interoperability!
“Principled” Language Interoperability?

• Compiler can preserve different properties through choice of type-translation: a spectrum of linking options
• Fully Abstract Compilation / Secure Compilation
  • compiler is equivalence-preserving
  • ensures a compiled component does not interact with any target behavior that is inexpressible in S
  • Recent results on fully abstract compilation
Next...

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

• Do we want to link with behavior inexpressible in S? Or do we want fully abstract compilers?
  • Answer: we want both!
  • How to get there? Languages should let programmers specify what behavior they want to link with
Verified Compilers for Multi-lang. World

- Runtime
- Drivers
Source programmers should be able to reason in the source language!
Source Language Reasoning

Contextual Equivalence

\[ e_1 \approx_{\text{ctx}}^{\text{ctx}} e_2 \]

: \[ \top \]
Source Language Reasoning

Contextual Equivalence

\( e_1 \approx^\text{ctx} e_2 \)

is indistinguishable from

by contexts at type \( \mathcal{T} \)
Source Language Reasoning

Contextual Equivalence

Formal basis for

- Refactoring
Source Language Reasoning

Contextual Equivalence

Formal basis for

- Refactoring
- Data abstraction
Source Language Reasoning

Contextual Equivalence

Formal basis for
- Refactoring
- Data abstraction
- Security
Secure compilation of components:

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

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

Must ensure that any \( a \) we link with behaves like some source context.
Ensuring Secure Compilation

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

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

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

\[ e : \tau \sim e : \tau^+ \]
If $e_1 : \tau \leadsto e_1 : \tau^+$ and $e_2 : \tau \leadsto e_2 : \tau^+$ then:

$$e_1 \approx_{ctx}^S e_2 : \tau \implies e_1 \approx_{ctx}^T e_2 : \tau^+$$
Fully abstract compilation

If $e_1 : \tau \leadsto e_1 : \tau^+$ and $e_2 : \tau \leadsto e_2 : \tau^+$ then:

$$e_1 \approx_{ctx}^S e_2 : \tau \iff e_1 \approx_{ctx}^T e_2 : \tau^+$$

preserves & reflects equivalence
Suppose $\Gamma \vdash e_1 : \tau \leadsto e_1$ and $\Gamma \vdash e_2 : \tau \leadsto e_2$.

\[
\Gamma \vdash e_1 \approx_{S}^\text{ctx} e_2 : \tau
\]

\[
\Gamma^+ \vdash e_1 \approx_{T}^\text{ctx} e_2 : \tau^+
\]
Challenge of **proving** full abstraction

Suppose $\Gamma \vdash e_1 : \tau \leadsto e_1$ and $\Gamma \vdash e_2 : \tau \leadsto e_2$.

\[
\Gamma \vdash e_1 \approx^{ctx} e_2 : \tau
\]

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

\[
\Gamma^+ \vdash e_1 \approx^{ctx} e_2 : \tau^+
\]
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) 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!

Observation: our 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, adds features from target to source.

Novel proof technique — **Universal Embedding**
- Untyped embedding of target in source
- Mediate between strongly typed source and untyped back-translation

[New et al.’16]
Fully Abstract Closure Conversion

Source: STLC + $\mu$ types
Target: System F + $\exists$ types + $\mu$ types + exceptions

Equivalent source terms, inequivalent in lang. with exceptions:

\[ e_1 = \lambda f. (f \ true; f \ false; \langle \rangle) \quad e_2 = \lambda f. (f \ false; f \ true; \langle \rangle) \]

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

\[ C[e_1] \Downarrow \text{true} \quad C[e_2] \Downarrow \text{false} \]

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

\[ e_1 \approx^{ctx}_S e_2 : (\text{bool} \to 1) \to 1 \]

\[ (\text{bool} \to E\,0\,1) \to E\,0\,1 \neq \]

\[ C : (\text{bool} \to E\text{bool}1) \to E\text{bool}1 \]

\[ C = ([\cdot] \, \lambda(x : \text{bool}). \text{raise } x) \]
Static Fully Abstract Compilation

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

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

Can we instead allow reasoning at source level?
Linking types are about raising programmer reasoning back to the source level.

Linking Types for Multi-Language Software: Have Your Cake and Eat it Too
[Patterson-Ahmed SNAPL’17]
In a Simpler Setting

\( \lambda \) (simply-typed lambda calculus)

\[
\begin{align*}
\tau & ::= \text{unit} | \text{int} | \tau \rightarrow \tau \\
\epsilon & ::= () | n | x | \lambda x : \tau. \epsilon \\
& \quad | \epsilon \epsilon | \epsilon + \epsilon | \epsilon \times \epsilon
\end{align*}
\]

\( \lambda \text{ref} \) (extended with ML references)

\[
\begin{align*}
\tau & ::= \ldots | \text{ref} \tau \\
\epsilon & ::= \ldots | \text{ref} \epsilon | \epsilon := \epsilon | !\epsilon
\end{align*}
\]

How to reason in \( \lambda \) while linking with \( \lambda \text{ref} \)?

Refactoring is reasoning about equivalence
Reasoning About Refactoring

\[ \lambda c. \ c() ; c() \quad \xrightarrow{\text{\small\text{\uparrow}}} \quad \lambda c. \ c() : (\text{unit} \to \text{int}) \to \text{int} \]

Should be okay because

\[ \lambda c. \ c() ; c() \approx^{\text{ctx}}_{\lambda} \lambda c. \ c() \]

Fully abstract compilers preserve equivalences.
What about linking with $\lambda ref$?

let counter $f' = let v = ref 0 in$
let $c'() = v := !v + 1; !v in f' c'$

let $f = \lambda c: unit \rightarrow int. c(); c()$

in counter $f$

but

let counter $f' = let v = ref 0 in$
let $c'() = v := !v + 1; !v in f' c'$

let $f = \lambda c: unit \rightarrow int. c()$

in counter $f$

When linked with $\lambda ref$, no longer equivalent!
Is this refactoring correct?

\[ \lambda c. \ c(); \ c() \quad \Rightarrow \quad \lambda c. \ c() : (\text{unit} \rightarrow \text{int}) \rightarrow \text{int} \]

It depends on what it is linked with!

- \[ \text{unit} \rightarrow \text{int} \] (Correct)
- \[ \text{unit} \rightarrow \text{int} \] (Incorrect)

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

\[ \tau ::= \text{unit} | \text{int} | \tau \rightarrow \tau \]

\[ \lambda^k \tau ::= \text{unit} | \text{int} | \tau \rightarrow R^\emptyset \tau | \text{ref} \tau | \tau \rightarrow R^\dagger \tau \]

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

Allows Programmers To Write Both

unit \rightarrow \text{int}

\text{unit} \rightarrow R^\emptyset \text{int}

\text{unit} \rightarrow R^\frac{1}{2} \text{int}
Refactoring: Pure Inputs

$\lambda c : \text{unit} \rightarrow R^0_{\text{int.}}. c(); c() \approx^{\text{ctx}}_{\lambda \kappa} \lambda c : \text{unit} \rightarrow R^0_{\text{int.}}. c()$

```ocaml
let counter f' = let v = ref 0 in
              let c' () = v := !v + 1; !v in f' c'
let f = \lambda c : \text{unit} \rightarrow R^0_{\text{int.}}. c()
in counter f
```

Ill-typed, since $f$ requires pure code
Refactoring: Impure Inputs

\[ \lambda c : \text{unit} \rightarrow R^\downarrow \text{int. } c();c() \not\sim^\text{ctx} \lambda c : \text{unit} \rightarrow R^\downarrow \text{int. } c() \]

let counter \( f' \) = let \( v = \text{ref } 0 \) in
    let \( c'() = v := !v + 1; !v \) in \( f'c' \)

let \( f \) = \( \lambda c : \text{unit} \rightarrow R^\downarrow \text{int. } c() \)
in counter \( f \)

Well-typed, since \( f \) accepts impure code
Minimal Annotation Burden

\[ \lambda c : \text{unit} \rightarrow R^\emptyset \text{int. } c(); c() \]

\[ \lambda c : \text{unit} \rightarrow \text{int. } c(); c() \]

\[ \lambda \kappa \text{ must provide default translation} \]

\[ \kappa^+(\text{unit}) = \text{unit} \]
\[ \kappa^+(\text{int}) = \text{int} \]
\[ \kappa^+(\tau_1 \rightarrow \tau_2) = \kappa^+(\tau_1) \rightarrow R^\emptyset \kappa^+(\tau_2) \]

\[ \forall e_1, e_2. \ e_1 \approx^{\text{ctx}} e_2 : \tau \implies e_1 \approx^{\text{ctx}} e_2 : \kappa^+(\tau) \]
Stepping Back...
Correct Compilation of Components

\[ e_s \approx e_T \]

specifies behaviors compiled code may be linked with
Correct Compilation of Components

- Compositional CompCert
- SepCompCert
- Pilsner

es\rightarrow e_s'^{\prime} e_s \Leftrightarrow e_T\approx e_s

specifies behaviors compiled code may be linked with

expressible in S
Correct Compilation: Multi-Language

$e_s \approx e_T$

specifies behaviors compiled code may be linked with

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

inexpressible in S
Correct Compilation: Multi-Language

Problem: programmer cannot reason at source level!

inexpressible in S
Fully Abstract Compilation?

Language specifications are incomplete! Don’t account for linking

Target
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

Typed IR

LLVM

modal types / type & effect

continuations

affine

fine-grained capabilities
PL Design, Linking Types, Compilers

Fully abstract compilers

continuations

affine pure

fine-grained capabilities

Typed IR + pure + dependent types

LLVM
Linking Types

- Allow programmers to reason in *almost* their own source languages, even when building multi-language software
- Allow compilers to be fully abstract, yet support multi-language linking
Conclusion
Compiler Verif. for Multi-Lang. World

• Compositional Compiler Correctness
  - horizontal and vertical compositionality
Horizontal / **Vertical** Compositionality

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

---

**CompCert**

- **SepCompCert**
  - Kang et al.’16

- **Transitivity**
  - requires effort / engineering

**Pilsner**

- Neis et al.’15

**Compositional CompCert**

- Stewart et al.’15

**Multi-language ST**

- Perconti-Ahmed’14
Compiler Verif. for Multi-Lang. World

- CompCert started a renaissance in compiler verification
  - major advances in mechanized proof
- Now we need: Compositional Compiler Correctness
  - but horizontal and vertical compositionality at odds
- Need to rethink proof architectures for compiler verification to support linking with code of arbitrary provenance. But want transitivity to be easier!
Verification of realistic compilers for a multi-language world demands formal techniques and language design
- compositional equational reasoning
- formal semantics of language interoperability
- types and logics to enforce sensible (safe, secure) linking
- extending our language designs with principled extensions to replace unprincipled escape hatches