Program Equivalence

& Compositional Compiler Correctness

Chung-kil Hur

joint work with Nick Benton

22 Mar 2010

MPI-SWS
• Program Equivalence within the same languages
  • Compiler Correctness
  • Our approach:
    - Program Equivalence between different languages
    - Compositional Compiler Correctness
  • Formalization in Cog
• Future work
Program Equivalence

Question
Which programs behave observationally the same in all valid contexts?

Main area of application
Formal verification of program transformations in optimizing compilers
Contextual Equivalence

\[ M_1 \approx_{\text{ctx}} M_2 \text{ if } \forall C[-], C[M_1] \downarrow \Leftrightarrow C[M_2] \downarrow \]

makes sense for languages with sound type system without input-output
Contextual Equivalence

Example

\[ \text{int minus (x:int) (y:int) := return (x-y)} \]
\[ \text{int minus' (x:int) (y:int) := return (y-x)} \]

\[ \text{minus} \not\subseteq_{ctx} \text{minus'} \text{ because} \]
\[ \text{if } [\text{minus}] \ 10 = 1 \text{ then } 0 \text{ else while(true) do skip od ; 0 ↓} \]
\[ \text{if } [\text{minus'}] \ 10 = 1 \text{ then } 0 \text{ else while(true) do skip od ; 0 ↑} \]
Contextual Equivalence

But, how do we prove

\[ \text{plus} \approx_{\text{ctx}} \text{plus}' \]

for

\[
\begin{align*}
\text{int plus} \; (x: \text{int}) \; (y: \text{int}) & \; := \; \text{return} \; x + y \\
\text{int plus'} \; (x: \text{int}) \; (y: \text{int}) & \; := \; \text{return} \; y + 2x
\end{align*}
\]
Contextual Equivalence

More difficult with

- recursion \( \leadsto \) fact \( n : = \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1) \)
- recursive types \( \leadsto \) List \( T : = \text{nil} \mid \text{cons of } T \times \text{List } T \)
- Polymorphic types \( \leadsto \) cons : \( \forall T. \ T \times \text{List } T \rightarrow \text{List } T \)
- Existential types \( \leadsto \) HashTable : \( \exists T. \ \text{create} : \text{void} \rightarrow T \)
  \begin{align*}
    \text{insert} : & \ T \times \text{int} \times \text{int} \rightarrow T \\
    \text{delete} : & \ T \times \text{int} \rightarrow T \\
    \text{lookup} : & \ T \times \text{int} \rightarrow \text{Option } \text{int}
  \end{align*}
- Higher-order mutable store
- Exceptions, Continuations, input-output, …
Operationally-based logical relation

polymorphic types $\leadsto$ existential types

Logical relation $+$ (Reynolds 1983)

recursion $\leadsto$ Biorthogonality $+$ (Krivine 1994; Pitts & Stark 1998)

first-order mutable state $\leadsto$ possible world model $+$ (Pitts & Stark 1998)

recursive types $\leadsto$ step-indexing $+$ (Appell & McAllester 2001; Ahmed 2006)

higher-order mutable state $\leadsto$ more ideas (Ahmed, Dreyer & Rossberg 2009)

Others: bisimulation based technique, denotationally-based log.rel.
Basic ideas behind the techniques

- Logical relation
  \[ n \xrightarrow{\text{val}} n \]
  \[ f \xrightarrow{A \to B} f' \text{ if } \forall a \xrightarrow{A} a', f(a) \xrightarrow{B} f(a') \]

- Biorthogonality
  \[ \xrightarrow{T} \text{ val comp } \xrightarrow{T} \text{ ctx s.t. } M \xrightarrow{T} M' \text{ if } \forall C \xrightarrow{T} C', C[M] \downarrow \iff C'[M'] \downarrow \]

- Step-indexing
  \[ \xrightarrow{k \in \mathbb{N}} \text{ val comp } \xrightarrow{k \in \mathbb{N}} \text{ for } k \in \mathbb{N} \]
  \[ s.t. M \xrightarrow{T} M' \text{ if } (M \downarrow_j V \text{ for } j < k \Rightarrow M' \downarrow V' \text{ s.t. } V \xrightarrow{T} V') \]
  \[ \wedge (M' \downarrow_j V' \text{ for } j < k \Rightarrow M \downarrow V \text{ s.t. } V \xrightarrow{T} V') \]
- Program Equivalence within the same languages
- Compiler Correctness
- Our approach:
  - Program Equivalence between different languages
  - Compositional Compiler Correctness
- Formalization in Cog
- Future work
Compiler Correctness

\[ L \Downarrow : \forall M \in S, L^M \Downarrow \text{behaves observationally the same as } M. \]

- state-of-the-art verified compiler: CompCert at INRIA

\[ L \Downarrow : \text{Clight} \rightarrow \text{PowerPC Assembly} \rightarrow \text{realistic optimizing compiler} \]

- complete program
- no polymorphic
- no existential
- no malloc

\[ \forall P \in \text{Clight}, \ (P \Downarrow \Rightarrow LP \Downarrow \Downarrow t) \]

\[ \neg (P \Downarrow \Rightarrow LP \Downarrow \Downarrow T) \]

\[ \neg \text{input/output traces} \]

N.B.
- no context considered
Non-compositionality of compiler correctness

only complete programs under empty context are considered
⇒ nothing is guaranteed about separate compilation
and linking.

Example

\[
P := \text{plusone (x:int) \{ return x+1 \}} \quad \text{main ( ) \{ return plusone(3) \}}
\]

we know that \( P \simeq LPJ \) but not \( P \simeq \text{link(}LFJ,LMJ\text{)} \)
because \( LPJ \) may be different from \( \text{link(}LFJ,LMJ\text{)} \)
• Program Equivalence within the same languages

• Compiler Correctness

• Our approach:
  - Program Equivalence between different languages
  - Compositional Compiler Correctness

• Formalization in Cog

• Future work
Compositional Compiler Correctness

How about

$$\forall c, m, \ C[M] \approx \text{link}(LC_J, LM_J)$$

Correctness of linking of compiled code with code from elsewhere is not guaranteed.

Example

$$\forall c, m, \ C[M] \approx \text{link}(LC_{J_1}, LM_{J_1})$$

$$\forall c, m, \ C[M] \approx \text{link}(LC_{J_2}, LM_{J_2})$$

$$\not\exists \forall c, m, \ C[M] \approx \text{link}(LC_{J_1}, LM_{J_2})$$
Our approach: Compiler-independent notion of equivalence

We generalize the idea of operationally-based logical relation to relate high-level and low-level programs.

\[
\text{extended sysF} \quad \overset{\sim}{\Rightarrow} \quad \text{H}
\]

with recursion, polymorphic types, existential types

\[
\sim \quad L
\]

\[
\text{low-level abstract machine with a syntactic equality test}
\]

without recursive types, mutable state, input/output

\[
\text{not assembly language}
\]

\[
\text{Future work}
\]

\[
\text{without possible world model}
\]
Difficulties in dealing with \( L \)

- \( L \) is an untyped low-level language
  - only \( P \downarrow, P \uparrow \) for complete configurations \( P \) makes sense
  - we use biorthogonality.

- not every low-level program is valid, so one should rule out invalid programs.
  - step-indexing plays a crucial role.
Program equivalence between H and L

\[
H : \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})
\]

\[
L : \text{log.rel+biur.} \rightarrow \text{log.rel+biur.} \rightarrow \text{step-indexing} \rightarrow \text{good}
\]

due to recursion + the syntactic equality test
Compositional Compiler Correctness

- Computational adequacy
  \[ p \sim p \Rightarrow (p \downarrow \Leftrightarrow p' \downarrow) \]

- Compositional locality
  \[ C \sim c \land M \sim m \Rightarrow C[M] \sim \text{link}(c, m) \]

- Compiler correctness
  \[ \forall M \quad M \sim LMI \]

- Linking
  \[ C \sim LCJ_1, M \sim LMJ_2 \Rightarrow C[M] \sim \text{link}(LCJ_1, LMJ_2) \Rightarrow (C[M] \downarrow \Leftrightarrow \text{link}(LCJ_1, LMJ_2) \downarrow) \]
Application

1. for a toy compiler L-J doing tail-call optimization
   \( \forall m \in H, M \sim LMJ \)

2. Correctness of handwritten code
   - polymorphic list module
   - fixpoint combinator
Specification and Implementation

ML ~ C

Permutation sort ~ quick sort

↑

Clear specification    efficient implementation
- Program Equivalence within the same languages
- Compiler Correctness
- Our approach:
  - Program Equivalence between different languages
  - Compositional Compiler Correctness
- Formalization in Cog
- Future work
What is Coq?

Coq: implementation of Calculus of inductive & coinductive Construction

- program: as a programming language (type theory)
- type: as a set theory
- element: as a logic
- set
- proposition
- proof
- proposition
- proof
- checking: fast decidable
- construction: by hand, but many automations
Our formalization in Coq

- encoding of sys-F
  - strongly typed representation
    - I developed a library \texttt{Heq} (ver 0.91)

- encoding of binding - PopLMark Challenge
  - de Bruijn index

  + kind of explicit substitution
• Program Equivalence within the same languages
• Compiler Correctness
• Our approach:
  - Program Equivalence between different languages
  - Compositional Compiler Correctness
• Formalization in Cog
• Future work
Summary & future work

- **Summary**
  
  logical relation + biorthogonality + step-indexing
  
  $\Rightarrow$ program equivalence between sys-F and SECD
  
  $\Rightarrow$ compositional compiler correctness
  
  + correctness of handwritten code

- **Future work**
  
  - recursive type + higher-order mutable store
  
  - realistic assembly language
  
  - other languages: C, Java, ...
  
  - as a verification technique
  
  - better formalization of languages in Coq