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 Coq
- 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

$M_1 \approx_{ctx} M_2$ 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

int minus $(x:\text{int})$ $(y:\text{int})$:= return $(x-y)$

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

minus $\not\simeq_{ctx}$ minus' because

if [minus] 1 0 = 1 then 0 else while (true) do skip od ; 0 $\downarrow$

if [minus'] 1 0 = 1 then 0 else while (true) do skip od ; 0 $\uparrow$

Contextual Equivalence

But, how do we prove

$$\text{plus} \approx_{ctx} \text{plus}'$$

for

```
int plus (x:int) (y:int)  := return x+y
int plus' (x:int) (y:int)  := return y+x
```

?

Contextual Equivalence

More difficult with

- recursion $\rightsquigarrow$ $\text{fact } n := \text{if } n=0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1)$
- recursive types $\rightsquigarrow$ $\text{List } T := \underline{\text{nil}} \mid \underline{\text{cons}} \text{ of } T \times \text{List } T$
- Polymorphic types $\rightsquigarrow$ $\underline{\text{cons}} : \forall T.\ T \times \text{List } T \to \text{List } T$
- Existential types $\rightsquigarrow$ $\text{HashTable} : \exists T.\ \text{create} : \text{void} \to T$
  $\text{insert} : T \times \text{int} \times \text{int} \to T$
  $\text{delete} : T \times \text{int} \to T$
  $\text{lookup} : T \times \text{int} \to \text{Option int}$
- higher-order mutuable store
- Exceptions, Continuations, input-output, ...

Operationally-based logical relation

polymorphic types ⇐ Logical relation (Reynolds 1983)
existential types

+

recursion ⇐ Biorthogonality (Krivine 1994; Pitts & Stark 1998)

+

first-order mutuable state ⇐ possible world model (Pitts & Stark 1998)

+

recursive types ⇐ step-indexing (Appell & McAllester 2001; Ahmed 2006)

+

higher-order mutuable state ⇐ more ideas (Ahmed, Dreyer & Rossberg 2009)

- others : bisimulation based technique, denotationally-based log.rel.

Basic ideas behind the techniques

- Logical relation

$$n \overset{val}{\sim}_{int} n$$

$$f \overset{val}{\sim}_{A\to B} f' \quad \text{if } \forall\, a \overset{val}{\sim}_{A} a',\ f(a) \overset{comp}{\sim}_{B} f'(a')$$

$$\overset{\wedge}{\wr}$$

$$\Big(\overset{\forall V}{f(a) \Downarrow V} \Rightarrow \overset{\exists V'}{f'(a') \Downarrow V'} \text{ s.t. } V \overset{val}{\underset{B}{\sim}} V'\Big)$$

$$\wedge \Big(\underset{\forall V'}{f'(a') \Downarrow V'} \Rightarrow \underset{\exists V}{f(a) \Downarrow V} \text{ s.t. } V \overset{val}{\underset{B}{\sim}} V'\Big)$$

$\vdots$

- Biorthogonality

$$\overset{val}{\sim}_T \quad \overset{comp}{\sim}_T \quad \overset{CTX}{\sim}_T \quad \text{s.t. } M \overset{comp}{\sim}_T M' \text{ if } \forall\, C \overset{CTX}{\sim}_T C',\ C[M]\downarrow \Leftrightarrow C'[M']\downarrow$$

$$\overset{\wedge}{\wr} \quad \forall\, V \overset{val}{\sim}_T V',\ C[V]\downarrow \Leftrightarrow C'[V']\downarrow$$

- Step-indexing

$$\overset{val}{\underset{k}{\sim}}_{T} \quad \overset{comp}{\underset{k}{\sim}}_{T} \quad \text{for } k \in \mathbb{N}$$

$$\text{s.t. } M \overset{comp}{\underset{k}{\sim}}_{T} M' \text{ if } \Big(M \Downarrow_j V \text{ for } j<k \Rightarrow M' \Downarrow V' \text{ s.t. } V \overset{val}{\underset{k-j}{\sim}}_{T} V'\Big)$$

$$\wedge \Big(M' \Downarrow_j V' \text{ for } j<k \Rightarrow M \Downarrow V \text{ s.t. } V \overset{val}{\underset{k-j}{\sim}}_{T} V'\Big)$$

- Program Equivalence within the same languages
- ➡ Compiler Correctness
- Our approach:
    - Program Equivalence between different languages
    - Compositional Compiler Correctness
- Formalization in Coq
- Future work

Compiler Correctness

$$\lfloor - \rfloor : S \to T$$

$\forall M \in S,\ \underset{\substack{\rotatebox{-90}{$\in$} \\ T}}{\lfloor M \rfloor}$ behaves observationally the same as $\underset{\substack{\rotatebox{-90}{$\in$} \\ S}}{M}$.

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

$\lfloor - \rfloor$: Clight → PowerPC Assembly → realistic optimizing compiler

complete program

$$\left.\begin{aligned} &\forall P \in \text{Clight},\ (P \downarrow t \Rightarrow \lfloor P \rfloor \downarrow t) \\ &\qquad \wedge (P \uparrow T \Rightarrow \lfloor P \rfloor \uparrow T) \end{aligned}\right\} \to \boxed{\text{denote by } P \approx \lfloor P \rfloor}$$

no polymorphic ↤
no existential
no malloc

input/output traces

N.B.
no context considered

Non-compositionality of compiler correctness

only complete programs under empty context are considered

$\Rightarrow$ nothing is guaranteed about seperate compilation and linking.

Example

$$P := \underbrace{\text{plusone (x:int) \{return x+1\}}}_{F} \quad \underbrace{\text{main () \{return plusone(3)\}}}_{M}$$

we know that $P \approx \lfloor P \rfloor$ but not $P \approx \text{link}(\lfloor F \rfloor, \lfloor M \rfloor)$

because $\lfloor P \rfloor$ may be different from $\text{link}(\lfloor F \rfloor, \lfloor M \rfloor)$

- Program Equivalence within the same languages
- Compiler Correctness
- ⇨ Our approach:
  - Program Equivalence between different languages
  - Compositional Compiler Correctness
- Formalization in Coq
- Future work

Compositional Compiler Correctness

How about

$$\forall C, M,\quad C[M] \approx \mathrm{link}(\lfloor C \rfloor, \lfloor M \rfloor)\ ?$$

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

Example

$$\forall C, M,\quad C[M] \approx \mathrm{link}(\lfloor C \rfloor_1, \lfloor M \rfloor_1)$$

$$\forall C, M,\quad C[M] \approx \mathrm{link}(\lfloor C \rfloor_2, \lfloor M \rfloor_2)$$

$$\not\Rightarrow \forall C, M,\quad C[M] \approx \mathrm{link}(\lfloor C \rfloor_1, \lfloor M \rfloor_2)$$

Our approach : Compiler-independent notion of equivalence

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

extended sysF

$H \sim L$

extended SECD

with
recursion
polymorphic types
extistential types

log. rel.
+
step-indexing
+
biorthogonality

low-level abstract machine
with a syntactic equality test

without
recursive types
mutable state
input/output

without possible world model

not assembly language

Future work

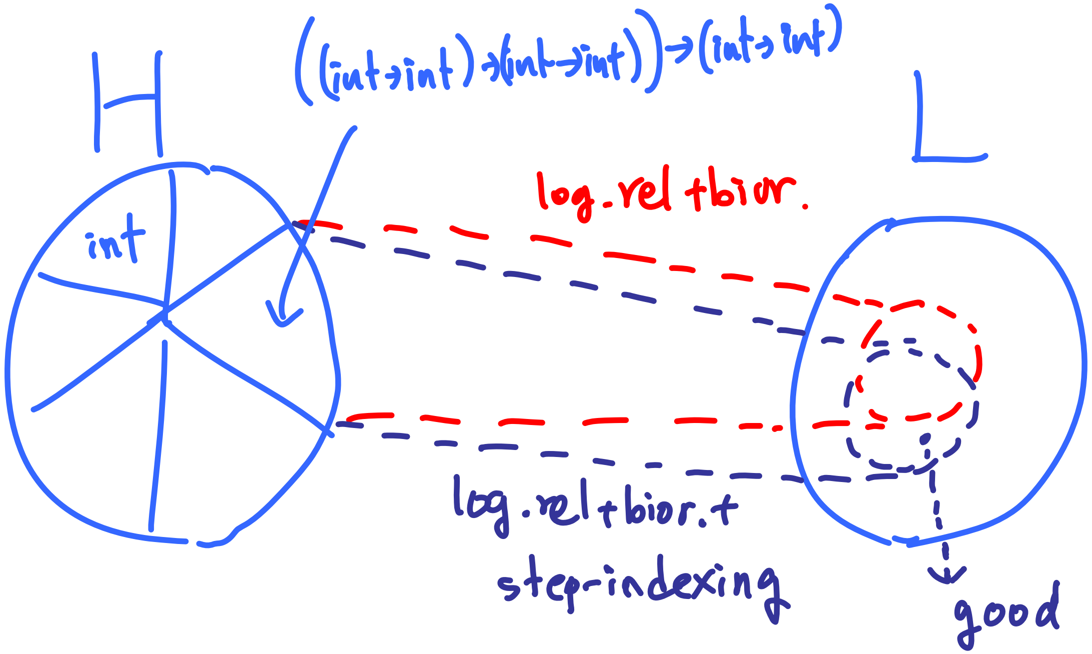
Difficulties in dealing with L

- 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



due to recursion
+ the syntactic equality test

Compositional Compiler Correctness

- Computational adequacy

$$P \sim p \Rightarrow (P{\downarrow} \Leftrightarrow p{\downarrow})$$

- Compositionality

$$C \sim c \wedge M \sim m \Rightarrow C[M] \sim \mathrm{link}(c, m)$$

- Compiler correctness

$$\forall M \quad M \sim \lfloor M \rfloor$$

- Linking

$$C \sim \lfloor C \rfloor_1 \,,\; M \sim \lfloor M \rfloor_2 \Rightarrow C[M] \sim \mathrm{link}(\lfloor C \rfloor_1, \lfloor M \rfloor_2)$$

$$\Rightarrow \left(C[M]{\downarrow} \Leftrightarrow \mathrm{link}(\lfloor C \rfloor_1, \lfloor M \rfloor_2){\downarrow}\right)$$

Application

① for a toy compiler $\llcorner \cdot \lrcorner$ doing tail-call optimization

$\forall M \in H, \quad M \sim \llcorner M \lrcorner$

② Correctness of Handwritten code

- polymorphic list module
- fixpoint combinator

Specification and Implementation

ML $\sim$ C

Permutation sort $\sim$ 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 Coq
- Future work

What is Coq?

Coq : implementation of Calculus of inductive & coinductive Construction

program	type	as a programming language (type theory)
element	set	as a set theory
proof	proposition	as a logic

We can do general mathematics inside Coq.

- proof checking : fast decidable
- proof construction : by hand, but many automations

Our formalization in Coq

- encoding of sys-F
  - strongly typed representation
    : I developed a library Heq_ (ver 0.91)
  - encoding of binding – POPLMARK Challenge
    : debruijn 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 Coq
- ➡ Future work

Summary & future work

- Summary

  logical relation + biorthogonality + step-indexing

  ⇝ program equivalence between sys-F and SECD

  ⇝ compositional compiler correctness
  + correctness of handwritten code

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