

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

$$M_1 \underset{\text{ctx}}{\approx} M_2 \quad \text{if} \quad \forall C[-], C[M_1] \downarrow \Leftrightarrow C[M_2] \downarrow$$

makes sense for languages

with sound type system

without input-output

Contextual Equivalence

Example

int minus (x:int) (y:int) := return (x-y)

int minus' (x:int) (y:int) := return (y-x)

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

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

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

Contextual Equivalence

But, how do we prove

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

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 $\text{List } T := \underline{\text{nil}} \mid \underline{\text{cons}}$ of $T \times \text{List } T$
- Polymorphic types \rightsquigarrow $\underline{\text{cons}} : \forall T. T \times \text{List } T \rightarrow \text{List } T$
- Existential types \rightsquigarrow $\text{HashTable} : \exists T. \begin{array}{l} \text{create} : \text{void} \rightarrow T \\ \text{insert} : T \times \text{int} \times \text{int} \rightarrow T \\ \text{delete} : T \times \text{int} \rightarrow T \\ \text{lookup} : T \times \text{int} \rightarrow \text{Option int} \end{array}$
- higher-order mutable store
- Exceptions, Continuations, input-output, ...

Operationally-based logical relation

polymorphic types \leftarrow Logical relation (Reynolds 1983)
existential types

recursion \leftarrow Biorthogonality (Krivine 1994;
pitts & Stark 1998)

first-order
mutable state \leftarrow possible world model (Pitts & Stark 1998)

recursive types \leftarrow Step-indexing (Appell & McAllester 2001;
Ahmed 2006)

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

• Others : bisimulation based technique, denotationally-based log. rel.

Basic ideas behind the techniques

- Logical relation

$$\eta \underset{\text{int}}{\sim}^{\text{val}} \eta$$

$$f \underset{A \rightarrow B}{\sim}^{\text{val}} f' \text{ if } \forall a \underset{A}{\sim}^{\text{val}} a', f(a) \underset{B}{\sim}^{\text{comp}} f'(a')$$

⋮

- Biorthogonality

$$\underset{T}{\sim}^{\text{val}} \underset{T}{\sim}^{\text{comp}} \underset{T}{\sim}^{\text{ctx}} \text{ s.t. } M \underset{T}{\sim}^{\text{comp}} M' \text{ if } \forall C \underset{T}{\sim}^{\text{ctx}} C', C[M] \downarrow \Leftrightarrow C[M'] \downarrow$$

- Step-indexing

$$\underset{k}{\sim}^{\text{val}} \underset{T}{\sim} \underset{k}{\sim}^{\text{comp}} \underset{T}{\sim} \text{ for } k \in \mathbb{N}$$

$$\text{s.t. } M \underset{k}{\sim}^{\text{comp}} M' \text{ if } \left(M \Downarrow_j V \text{ for } j < k \Rightarrow M' \Downarrow V' \text{ s.t. } V \underset{k-j}{\sim}^{\text{val}} V' \right) \\ \wedge \left(M' \Downarrow_j V' \text{ for } j < k \Rightarrow M \Downarrow V \text{ s.t. } V \underset{k-j}{\sim}^{\text{val}} V' \right)$$

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

- 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

$$L \dashv \vdash : S \rightarrow T$$

$\forall M \in S, \underset{\substack{\mathbb{P} \\ T}}{LM \dashv \vdash}$ behaves observationally the same as $\underset{\substack{\mathbb{P} \\ S}}{M}$.

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

$L \dashv \vdash : \text{Clight} \rightarrow \text{PowerPC Assembly}$ \rightarrow realistic optimizing compiler

$\forall P \in \text{Clight}, (P \downarrow t \Rightarrow LP \dashv \vdash \downarrow t) \wedge (P \uparrow T \Rightarrow LP \dashv \vdash \uparrow T)$

$\left. \begin{array}{l} \leftarrow \text{complete program} \\ \leftarrow \text{no polymorphic} \\ \leftarrow \text{no existential} \\ \leftarrow \text{no malloc} \end{array} \right\} \text{denote by } P \approx LP \dashv \vdash$

$\left. \begin{array}{l} \leftarrow \text{input/output traces} \end{array} \right\} \text{N.B. no context considered}$


Non-compositionality of compiler correctness

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

Example

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

we know that $P \approx LP \downarrow$ but not $P \approx \text{link}(LF \downarrow, LM \downarrow)$
because $LP \downarrow$ may be different from $\text{link}(LF \downarrow, LM \downarrow)$

- 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

$$\forall C, M, C[M] \approx \text{link}(LC, LM) ?$$

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

Example

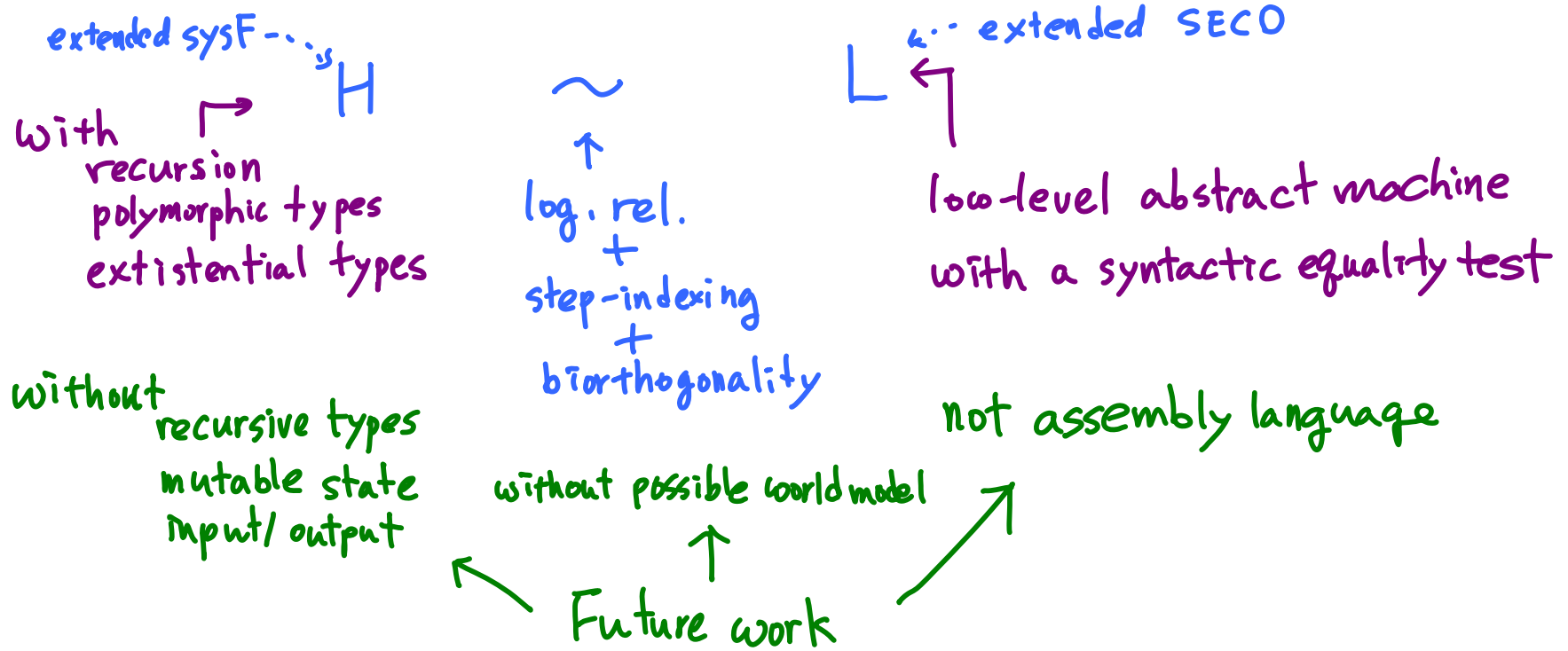
$$\forall C, M, C[M] \approx \text{link}(LC_1, LM_1)$$

$$\forall C, M, C[M] \approx \text{link}(LC_2, LM_2)$$

$$\nRightarrow \forall C, M, C[M] \approx \text{link}(LC_1, LM_2)$$

Our approach : Compiler-independent notion of equivalence

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



Difficulties in dealing with L

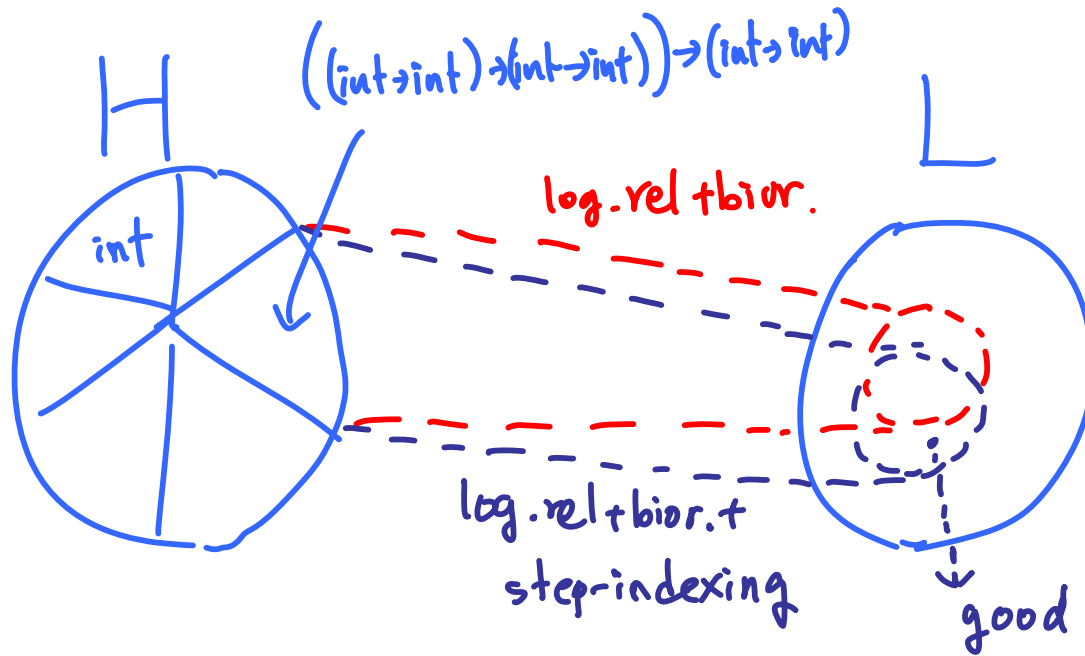
- L is an untyped low-level language
 - only $P \downarrow$, $P \uparrow$ for complete configurations P makes sense

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

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

- Compiler correctness

$$\forall M \quad M \sim \llbracket M \rrbracket$$

- Linking

$$\begin{aligned} C \sim \llbracket C \rrbracket_1, \quad M \sim \llbracket M \rrbracket_2 &\Rightarrow C[M] \sim \text{link}(\llbracket C \rrbracket_1, \llbracket M \rrbracket_2) \\ &\Rightarrow (C[M] \downarrow \Leftrightarrow \text{link}(\llbracket C \rrbracket_1, \llbracket M \rrbracket_2) \downarrow) \end{aligned}$$

Application

① for a toy compiler $L \rightarrow J$ doing tail-call optimization

$$\forall M \in H, M \sim LM$$

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 Heg (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 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 mutable store

- realistic assembly language

- other languages : C, Java, ...

- as a verification technique

- better formalization of languages in Coq