A Compositional Semantics for Verified Separate Compilation & Linking

Tahina Ramananandro
Zhong Shao
Shu-Chun Weng
Jérémie Koenig
Yuchen Fu

January 13\textsuperscript{th}, 2015
Certified Programs and Proofs
TIFR, Mumbai, Maharashtra, India
Vertical vs. horizontal composition (Hur et al. POPL 2012)

Proof

Spec1

Horizontal compositionality

Source1

Compiler

Target1

LINK??

Spec2

Proof

Source2

Compiler

Target2

Vertical compositionality
How to preserve component proofs by compilation and linking?

CompCert (Leroy POPL 2006) cannot preserve proofs by linking

- Works only for whole programs
- No correctness statement for open modules
Our unified approach: compositional semantics + refinement

Based on a compositional semantics

Research challenges

- What is the semantics of an open module?
- How to generalize compiler correctness to open modules?
- How to connect to compositional program logics?
Our contributions

1. Semantics of open modules
2. Semantic linking operator
3. Linking theorem
4. Compositional refinement
5. Refinement for memory-changing passes
Our contributions

1. Semantics of open modules
2. Semantic linking operator
3. Linking theorem
4. Compositional refinement
5. Refinement for memory-changing passes
Reminder: Operational small-step semantics

- Usual way to describe the machine semantics of the executable

  $s \xrightarrow{\text{eventList}} s'$

- Not suitable to describe compiler correctness at the higher level
  - Too fine-grained
  - Optimizations can change intermediate states

- Not compositional for linking purposes
  - Only for whole programs
Observable program behaviors

- Big-step the small-step semantics
- \([\text{Prog}] \subseteq \{\text{Terminates(eventList)}, \text{Stuck(eventList)}, \text{Diverges(eventList)}, \text{Reacts(eventStream)}\}\)

- Compiler correctness: program behavior refinement:
  \([\text{Compiler(Prog)}] \subseteq [\text{Prog}]\)
## Examples

<table>
<thead>
<tr>
<th>The C program...</th>
<th>... has the behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>int main () {</td>
<td></td>
</tr>
<tr>
<td>printf('a');</td>
<td></td>
</tr>
<tr>
<td>return 2;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>OUT (a) . Terminates(2)</td>
</tr>
<tr>
<td>int main () {</td>
<td></td>
</tr>
<tr>
<td>printf('a');</td>
<td></td>
</tr>
<tr>
<td>3/0;</td>
<td></td>
</tr>
<tr>
<td>return 4;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>OUT (a) . Stuck</td>
</tr>
<tr>
<td>int main () {</td>
<td></td>
</tr>
<tr>
<td>printf('a');</td>
<td></td>
</tr>
</tbody>
</table>
|     while (1) {};
|     return 5;   |
| }               | OUT (a) . Diverges |
| int main () {    |
|     while (1) { printf('b'); };
|     return 6;   |
| }               | OUT (b) :: ... :: OUT (b) :: ... (Reacts) |
Big-stepping
the small-step semantics

Terminates($l_1 ++ l_2 ++ \ldots ++ l_n$)

\[
\begin{array}{c}
s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n \text{ final state}
\end{array}
\]

Stuck($l_1 ++ l_2 ++ \ldots ++ l_n$)

\[
\begin{array}{c}
s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n \text{ not final}
\end{array}
\]

Diverges($l_1 ++ l_2 ++ \ldots ++ l_n$)

\[
\begin{array}{c}
s_0 \rightarrow \ldots \rightarrow s_n = s'0 \rightarrow s'1 \rightarrow \ldots \text{ indefinitely}
\end{array}
\]

Reacts($l_1 +++ l_2 +++ \ldots$)

\[
\begin{array}{c}
s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \text{ indefinitely}
\end{array}
\]
How to deal with input?

Provide a behavior for each possible input value.

The remaining behavior depends on that input value.

```c
int main() {
    char x = getchar();
    printf("%c", x);
    return 0;
}
```

```plaintext
{ 
IN(a) :: OUT(a) . Terminates(0),
IN(b) :: OUT(b) . Terminates(0),
IN(c) :: OUT(c) . Terminates(0),
...}
```
From Operational to Compositional Semantics

Whole-program small-step semantics

big-stepping

Whole-program big-step semantics

Events for external function calls

Semantic linking

Compositional small-step semantics

big-stepping

Compositional big-step semantics
Compositional semantics

- Events for external function calls
  - Provide behaviors for each possible return value and return memory state (like input)

\[
\begin{align*}
m &= \{ f \mapsto \text{int } x=18; \text{ int } y=g(&x); \text{ printf("\%d \%d", y, x);}\} \\
\langle m \rangle (f) &= \{ \\
\text{Extcall}(g, [x->18], &x, 0, [x->0]) :: \text{OUT 0 :: OUT 0 . Terminates}, \\
\text{Extcall}(g, [x->18], &x, 0, [x->1]) :: \text{OUT 0 :: OUT 1 . Terminates}, \\
... \\
\text{Extcall}(g, [x->18], &x, 1, [x->0]) :: \text{OUT 1 :: OUT 0 . Terminates}, \\
\text{Extcall}(g, [x->18], &x, 1, [x->1]) :: \text{OUT 1 :: OUT 1 . Terminates}, \\
... \}\}
\end{align*}
\]
Compositional semantics

• Events for external function calls
  – Provide behaviors for each possible return value and return memory state (like input)

\[
m = \{ f \mapsto \text{int } x=18; \text{ int } y=g(&x); \text{ printf("%d %d", y, x); } \}
\]

\[
\{ m \} (f) = \{
\text{Extcall}(g, [x->18], &x, 0, [x->0]) :: \text{OUT 0 :: OUT 0 . Terminates,}
\text{Extcall}(g, [x->18], &x, 0, [x->1]) :: \text{OUT 0 :: OUT 1 . Terminates,}
...
\text{Extcall}(g, [x->18], &x, 1, [x->0]) :: \text{OUT 1 :: OUT 0 . Terminates,}
\text{Extcall}(g, [x->18], &x, 1, [x->1]) :: \text{OUT 1 :: OUT 1 . Terminates,}
\}\n\]
Compositional semantics

- Events for external function calls
  - Provide behaviors for each possible return value and return memory state (like input)

\[
m = \{ f \mapsto \begin{align*}
&\text{int } x = 18; \\
&\text{int } y = g(&x); \\
&\text{printf(“%d %d”, y, x);}
\end{align*}\}
\]

\[
\langle m \rangle (f) = \{
\begin{align*}
\text{Extcall}(g, [x \mapsto 18], &x, 0, [x \mapsto 0]) &:: \text{OUT 0 :: OUT 0 . Terminates,} \\
\text{Extcall}(g, [x \mapsto 18], &x, 0, [x \mapsto 1]) &:: \text{OUT 0 :: OUT 1 . Terminates,} \\
\end{align*}
\]

...
Compositional semantics

- Function semantics parameterized on arguments and memory state before call
- Terminating behaviors also bear return value and return memory state

\[
m' = \{g(\text{int}^* x) \mapsto \text{int } y=\*x; \text{ printf("\%d", y-1); } \*x=y+1; \text{ return } y; \} \]

\[
\langle m' \rangle (g)(p)[p->n] = \{
\text{OUT } (n-1) . \text{ Terminates}(n, [p->n+1])
\}
\]
Our contributions

1. Semantics of open modules
2. Semantic linking operator
3. Linking theorem
4. Compositional refinement
5. Refinement for memory-changing passes
Semantic Linking

\[ m_1 = \{ f_1 \mapsto \text{call } f_2 \} \]

\[ \langle m_1 \rangle (f_1) = \text{Extcall}(f_2) \text{ Terminates} \]

\[ m_2 = \{ f_2 \mapsto \text{call } f_1 \} \]

\[ \langle m_2 \rangle (f_2) = \text{Extcall}(f_1) \text{ Terminates} \]

\[ \langle m_1 \rangle \gg \langle m_2 \rangle (f_1) = \{ \text{Diverges} \} \]

\[ \langle m_1 \rangle \gg \langle m_2 \rangle (f_2) = \{ \text{Diverges} \} \]

\[ m_1 \cup m_2 = \{ f_1 \mapsto \text{call } f_2, f_2 \mapsto \text{call } f_1 \} \]

\[ \langle m_1 \cup m_2 \rangle = \langle m_1 \rangle \gg \langle m_2 \rangle \]

- Defined at the semantic level of big-step behaviors
  - No need for the underlying small-step semantics
  - Can link semantics of modules of different languages

- Main technical challenge (mechanized Coq proof)
The linking operator

- “Replace” each external function call event with a behavior of the callee
- Linking based on a resolution operator $R$ performing those replacements:

$$\psi_1 \bowtie \psi_2 = R(\psi_1 \psi \psi_2)$$
The resolution operator
Example #1: Terminating case

\[ \psi = \{ g \mapsto \text{Terminates} ; \\
    f \mapsto \text{Extcall}(g) \cdot \text{Terminates} \} \]

\[ \mathcal{R}(\psi) = \{ g \mapsto \text{Terminates} ; \\
    f \mapsto \text{Terminates} \} \]
The resolution operator
Example #2: Diverging case

\[ \psi = \{ g \mapsto \text{Diverges} ; \\ f \mapsto \text{Extcall}(g) :: \text{Print 2 . Terminates} \} \]

\[ R(\psi) = \{ g \mapsto \text{Diverges} ; \\ f \mapsto \text{Diverges} \} \]
The resolution operator

Example #3: Infinitely many externals

\[ \psi = \{ g \mapsto \text{Print}(e) \cdot \text{Terminates} ;
    f \mapsto \text{Extcall}(g) :: \text{Extcall}(g) :: \ldots :: \text{Extcall}(g) :: \ldots \} \]

\[ R(\psi) = \{ g \mapsto \text{Print}(e) \cdot \text{Terminates} ;
    f \mapsto \text{Print}(e) :: \text{Print}(e) :: \ldots :: \text{Print}(e) :: \ldots \} \]
The resolution operator
Example #4: Infinitely many externals

\[ \psi = \{ g \mapsto \text{Terminates} ; \\
    f \mapsto \text{Extcall}(g) :: \text{Extcall}(g) :: \ldots :: \text{Extcall}(g) :: \ldots \} \]

\[ R(\psi) = \{ g \mapsto \text{Terminates} ; \\
    f \mapsto \text{Diverges} \} \]

Eager replacement will fail.
The resolution operator
Example #5: (mutual) recursion

$$\psi = \{ g \mapsto \text{Extcall}(f) :: \text{Print}(a) \cdot \text{Terminates} ;$$
$$f \mapsto \text{Extcall}(g) :: \text{Print}(b) \cdot \text{Terminates} \}$$

$$R(\psi) = \{ g \mapsto \text{Diverges} ;$$
$$f \mapsto \text{Diverges} \}$$
How resolution works

- **Behavior simulation** small-step semantics
- Big-step this small-step semantics

\[
\begin{align*}
\text{(a)} & \quad \triangletriangle e \quad \quad \rightarrow \quad \triangletriangle e \\
\text{(b)} & \quad \triangletriangle \text{Extcall}(f, m_1, m_2) \quad \rightarrow \quad \triangletriangle \text{Extcall}(f, m_1, m_2) \\
\text{(c)} & \quad \triangletriangle \text{Extcall}(f, m_1, m_2) \quad \rightarrow \quad \triangletriangle \circ\circ\circ\circ\circ \quad \rightarrow \quad \triangletriangle \circ\circ\circ\circ\circ
\end{align*}
\]

Fig. 1. Three cases in behavior simulation: (a) regular event; (b) \( f \notin \text{dom}(\psi) \); (c) \( \circ\circ\circ\circ\circ \in \psi(f) \).
How resolution works

\[ \psi = \{ g \mapsto \text{Print}(e) \cdot \text{Terminates} ; f \mapsto \text{Extcall}(g) :: \text{Extcall}(g) \cdot \text{Terminates} \} \]

How to compute \( R(\psi)(f) \)?
How resolution works

\[ \psi = \{g \mapsto \text{Print}(e) \ . \ \text{Terminates} ; \ f \mapsto \text{Extcall}(g) :: \text{Extcall}(g) \ . \ \text{Terminates} \} \]

How to compute \( R(\psi)(f) \)?

\[ \text{Extcall}(g) :: \text{Extcall}(g) \ . \ \text{Terminates} , [] \]
How resolution works

\[ \psi = \{ g \mapsto \text{Print}(e) \cdot \text{Terminates} ; \]
\[ f \mapsto \text{Extcall}(g) :: \text{Extcall}(g) \cdot \text{Terminates} \} \]

How to compute \( R(\psi)(f) \) ?

\[ \text{Extcall}(g) :: \text{Extcall}(g) \cdot \text{Terminates} , [] \]
How resolution works

\[ \psi = \{ g \mapsto \text{Print(e) . Terminates} ; \\
       f \mapsto \text{Extcall(g) :: Extcall(g) . Terminates} \} \]

How to compute \( R(\psi)(f) \) ?

\[
\text{Print(e) . Terminates} , [\text{Extcall(g) . Terminates}] 
\]
How resolution works

\[ \psi = \{ g \mapsto \text{Print}(e) \cdot \text{Terminates} ; f \mapsto \text{Extcall}(g) :: \text{Extcall}(g) \cdot \text{Terminates} \} \]

How to compute \( R(\psi)(f) \) ?

\[ \text{Print}(e) \cdot \text{Terminates} , [\text{Extcall}(g) \cdot \text{Terminates}] \]
How resolution works

\[ \psi = \{ \text{g} \mapsto \text{Print(e) . Terminates} ; \text{f} \mapsto \text{Extcall(g) :: Extcall(g) . Terminates} \} \]

How to compute \( R(\psi)(f) \) ?

\[ \text{Terninates , [Extcall(g) . Terminates] Print(e)} \]
How resolution works

\[ \psi = \{ g \mapsto \text{Print}(e) \cdot \text{Terminates} ; \\
 f \mapsto \text{Extcall}(g) :: \text{Extcall}(g) \cdot \text{Terminates} \} \]

How to compute \( R(\psi)(f) \) ?

\[ \text{Terminates} , \text{[Extcall}(g) \cdot \text{Terminates}] \]

\[ \text{Print}(e) . \]
How resolution works

\[ \psi = \{ g \mapsto \text{Print(e)} \cdot \text{Terminates} ; \]
\[ f \mapsto \text{Extcall(g)} :: \text{Extcall(g)} \cdot \text{Terminates} \} \]

How to compute \( R(\psi)(f) \) ?

\[ \text{Extcall(g)} \cdot \text{Terminates} , [] \]

\[ \text{Print(e)} . \]
How resolution works

\[ \psi = \{ g \mapsto \text{Print}(e) \ . \ \text{Terminates} \ ; \\
 f \mapsto \text{Extcall}(g) :: \text{Extcall}(g) \ . \ \text{Terminates} \} \]

How to compute \( \mathcal{R}(\psi)(f) \) ?

\[ \text{Extcall}(g) \ . \ \text{Terminates}, [] \]

\[ \text{Print}(e) \ . \]

How resolution works

\[ \psi = \{ g \mapsto \text{Print}(e) \ . \ \text{Terminates} \ ; \ f \mapsto \text{Extcall}(g) :: \text{Extcall}(g) \ . \ \text{Terminates} \} \]

How to compute \( R(\psi)(f) \)?

\[ \text{Print}(e) \ . \ \text{Terminates} \ , \ [\text{Terminates}] \]

\[ \text{Print}(e) \ . \]
How resolution works

\[ \psi = \{ g \mapsto \text{Print(e)} \cdot \text{Terminates} ; \\
    f \mapsto \text{Extcall(g)} :: \text{Extcall(g)} \cdot \text{Terminates} \} \]

How to compute \( R(\psi)(f) \) ?

\text{Print(e)} \cdot \text{Terminates} , \ [\text{Terminates}] \text{Print(e)} .
How resolution works

\[ \psi = \{ g \mapsto \text{Print}(e) . \text{Terminates} ; f \mapsto \text{Extcall}(g) :: \text{Extcall}(g) . \text{Terminates} \} \]

How to compute \( R(\psi)(f) \) ?

Terminates, [Terminates]

Print(e) :: Print(e) .
How resolution works

\[ \psi = \{ g \mapsto \text{Print(e)} \cdot \text{Terminates} ; \\
    f \mapsto \text{Extcall(g)} :: \text{Extcall(g)} \cdot \text{Terminates} \} \]

How to compute \( R(\psi)(f) \) ?

\textbf{Terminates} , \ [\textbf{Terminates}] \\
\text{Print(e)} :: \text{Print(e)} .
How resolution works

\[ \psi = \{ g \mapsto \text{Print}(e) \cdot \text{Terminates} ; \\
\quad f \mapsto \text{Extcall}(g) :: \text{Extcall}(g) \cdot \text{Terminates} \} \]

How to compute \( R(\psi)(f) \) ?

\[ \text{Terminates} , [] \]

\text{Print}(e) :: \text{Print}(e) . \]
How resolution works

\[ \psi = \{ g \mapsto \text{Print}(e) \ . \ \text{Terminates} \ ; \\
    f \mapsto \text{Extcall}(g) :: \text{Extcall}(g) \ . \ \text{Terminates} \} \]

How to compute \( R(\psi)(f) \) ?

\[ \text{Print}(e) :: \text{Print}(e) \ . \ \text{Terminates} \]
Resolution and (mutual) recursion

\[ \psi = \{ g \mapsto \text{Extcall}(f) :: \text{Print}(a) . \text{Terminates} ; \\
\quad f \mapsto \text{Extcall}(g) :: \text{Print}(b) . \text{Terminates} \} \]

How to compute \( R(\psi)(g) \) ?
Resolution and (mutual) recursion

\[ \psi = \{ g \mapsto \text{Extcall}(f) :: \text{Print}(a) . \text{Terminates} ; \\ f \mapsto \text{Extcall}(g) :: \text{Print}(b) . \text{Terminates} \} \]

How to compute \( R(\psi)(g) \) ?

\[ \text{Extcall}(f) :: \text{Print}(a) . \text{Terminates} , \]

[]
Resolution and (mutual) recursion

\[ \psi = \{ g \mapsto \text{Extcall}(f) :: \text{Print}(a) \cdot \text{Terminates} ; \\
               f \mapsto \text{Extcall}(g) :: \text{Print}(b) \cdot \text{Terminates} \} \]

How to compute \( R(\psi)(g) \) ?

\[ \text{Extcall}(g) :: \text{Print}(b) \cdot \text{Terminates} , \\
    [\text{Print}(a) \cdot \text{Terminates}] \]
Resolution and (mutual) recursion

\[ \psi = \{ g \mapsto \text{Extcall}(f) :: \text{Print}(a) . \text{Terminates} ; f \mapsto \text{Extcall}(g) :: \text{Print}(b) . \text{Terminates} \} \]

How to compute \( R(\psi)(g) \) ?

\[ \text{Extcall}(f) :: \text{Print}(a) . \text{Terminates} , [\text{Print}(b) . \text{Terminates} ; \text{Print}(a) . \text{Terminates}] \]
Resolution and (mutual) recursion

\[ \psi = \{ g \mapsto \text{Extcall}(f) :: \text{Print}(a) \cdot \text{Terminates} ; \\
     f \mapsto \text{Extcall}(g) :: \text{Print}(b) \cdot \text{Terminates} \} \]

How to compute \( R(\psi)(g) \) ?

\[ \text{Extcall}(g) :: \text{Print}(b) \cdot \text{Terminates} , \\
    [ \text{Print}(a) \cdot \text{Terminates} ; \\
    \text{Print}(b) \cdot \text{Terminates} ; \\
    \text{Print}(a) \cdot \text{Terminates} ] \]
Resolution and (mutual) recursion

\[ \psi = \{ g \mapsto \text{Extcall}(f) :: \text{Print}(a) . \text{Terminates} ; f \mapsto \text{Extcall}(g) :: \text{Print}(b) . \text{Terminates} \} \]

How to compute \( R(\psi)(g) \) ?

… and so on: it will diverge

[ \text{Print}(b) . \text{Terminates} \]
\[ \text{Print}(a) . \text{Terminates} ; \]
\[ \text{Print}(b) . \text{Terminates} ; \]
\[ \text{Print}(a) . \text{Terminates} \]
Relation with denotational semantics

- Parameterization needs care (intension, or coinductive local vs. global knowledge) to make fixpoints work
- Our work needs no such “fixpoint” thing besides regular big-stepping of small-step semantics
Higher-order functions

- \( \text{iter } f \ (x_1 :: ... :: x_n :: \text{nil}) \)

- Semantics depends on symbol resolution and module in which iter is defined:
  - If \( f \) is an external function, then:
    \( \text{Extcall}(f) :: ... :: \text{Extcall}(f) . \) Terminates
  - If \( f \) is a function defined in the same module as iter, then no such events
Challenges

• How to cope with return values / return memory state?
Reminder: compositional semantics

- Provide behaviors for each possible return value and each possible return memory state

\[ m_1 = \{ f \mapsto \text{int } x=18; \text{int } y=g(\&x); \text{printf}("%d %d", y, x) \} \]

\[ \langle m_1 \rangle (f) = \{ \]
\[ \text{Extcall}(g, [x\rightarrow 18], \&x, 0, [x\rightarrow 0]) :: \text{print} 0 :: \text{print} 0 . \text{Terminates}, \]
\[ \text{Extcall}(g, [x\rightarrow 18], \&x, 0, [x\rightarrow 1]) :: \text{print} 0 :: \text{print} 1 . \text{Terminates}, \]
\[ \ldots \]
\[ \text{Extcall}(g, [x\rightarrow 18], \&x, 1, [x\rightarrow 0]) :: \text{print} 1 :: \text{print} 0 . \text{Terminates}, \]
\[ \text{Extcall}(g, [x\rightarrow 18], \&x, 1, [x\rightarrow 1]) :: \text{print} 1 :: \text{print} 1 . \text{Terminates}, \]
\[ \ldots \} \]
Behavior simulation with return value and memory state

\[
\psi_1(f) = \{ \\
\text{Extcall}(g, [x\mapsto 18], \&x, 0, [x\mapsto 0]) :: \text{print } 0 :: \text{print } 0 . \text{Terminates}(x\mapsto 0), \\
\text{Extcall}(g, [x\mapsto 18], \&x, 0, [x\mapsto 1]) :: \text{print } 0 :: \text{print } 1 . \text{Terminates}(x\mapsto 1), \\
\ldots \\
\text{Extcall}(g, [x\mapsto 18], \&x, 1, [x\mapsto 0]) :: \text{print } 1 :: \text{print } 0 . \text{Terminates}(x\mapsto 0), \\
\text{Extcall}(g, [x\mapsto 18], \&x, 1, [x\mapsto 1]) :: \text{print } 1 :: \text{print } 1 . \text{Terminates}(x\mapsto 1) \\
\ldots \}
\]

\[
m_2 = \{g \mapsto (\text{int* px}) \ast px = 1729; \text{ return } 42;\} \\
\langle m_2 \rangle (g) = \{ \text{Terminates}(42, (x\mapsto 1729)) \}
\]
Behavior simulation with return value and memory state

\[ \psi_2(g) = \{ \text{Terminates}(42, (x\rightarrow1729)) \} \]
Behavior simulation with return value and memory state

\[
\psi_2(g) = \{ \text{Terminates}(42, (x\mapsto1729)) \}
\]
Behavior simulation with return value and memory state

Terminates(42, (x↦1729)) ,
[ (1, [x↦0]) . print 1 :: print 0 . Terminates(x↦0) ]
Behavior simulation with return value and memory state

Terminates(42, (x↦1729)) ,
[ (1, [x↦0]) . print 1 :: print 0 . Terminates(x↦0) ]

- The result expected by the caller is not the result returned by the callee.

- How to choose in advance a behavior of the caller in accordance with all its callees' behaviors?
  - What if the caller performs infinitely many external function calls?
Behavior simulation with return value and memory state

\[
\text{Terminates}(42, (x \mapsto 1729)) ,
\text{[ (1, [x \mapsto 0]) . print 1 :: print 0 . Terminates(x \mapsto 0) ]}
\]

**SPURIOUS**

This behavior will be removed from the behavior simulation big-step semantics

(consider it as a terminating behavior with result SPURIOUS).
Our contributions

1. Semantics of open modules
2. Semantic linking operator
3. Linking theorem
4. Compositional refinement
5. Refinement for memory-changing passes
Does resolution make sense?

- How to relate behavior simulation with actual program linking?
Yes, resolution makes sense!

- How to relate behavior simulation with actual program linking?
  - Linking two modules written in the same language

\[
\langle m_1 \cup m_0 \rangle = \langle m_1 \rangle \bowtie \langle m_0 \rangle
\]

Mechanized Coq proof
\[
\langle \mathbf{m}_1 \rangle \bowtie \langle \mathbf{m}_0 \rangle \subseteq \langle \mathbf{m}_1 \cup \mathbf{m}_0 \rangle
\]

• Simulation diagram

\[
\begin{align*}
(b, \chi) & \quad \rightarrow \quad (b', \chi') \\
(h, l, p) & \text{ behaves } b \text{ in } \langle \mathbf{m}_1 \cup \mathbf{m}_0 \rangle \\
\chi & \text{ similarly matches } q
\end{align*}
\]

Invariant INV

\[
\text{Stack: } k = p ++ q
\]

\[
\begin{align*}
(h, l, k) & \quad \rightarrow \quad (h', l', k')
\end{align*}
\]
\( \langle m_1 \cup m_0 \rangle \subseteq \langle m_1 \rangle \bowtie \langle m_0 \rangle \)

- Easy induction for finite (terminating/stuck) behaviors
- For infinite (diverging/reacting) behaviors, distinguish between 3 cases:
  - Finitely many external function calls that all terminate
  - Infinitely many external function calls that all terminate
  - Some external function call does not return
Summary

- Resolution and linking operator at the semantic level
- Independent of the underlying language
- Allows linking behaviors of modules written in different languages
Our contributions

1. Semantics of open modules
2. Semantic linking operator
3. Linking theorem
4. Compositional refinement
5. Refinement for memory-changing passes
Compositional refinement

- Expressed at the semantic level
- CompCert behavior improvement (cf. Dockins' PhD thesis)
  - \( \text{Beh1} \text{ improves } \text{Beh2} \text{ iff } \text{Beh1} = \text{Beh2} \text{ or } \text{Beh2} \text{ is a stuck prefix of } \text{Beh1} \)
- **Vertical composition**: Already known to be transitive
- **Horizontal composition**: Compatible with linking
  \[
  \psi_1 \sqsubseteq \psi_2 \implies \psi \trianglerighteq \psi_1 \sqsubseteq \psi \trianglerighteq \psi_2
  \]
- Mechanized (Coq) proof
Compositional compiler correctness

\[ \langle \text{Compiler}(\text{Prog}) \rangle \subseteq \langle \text{Prog} \rangle \]

- Refinement with compositional semantics
- No deep change in CompCert proofs
  - External function call events treated like ordinary events
  - We have instantiated our framework with a CompCert optimization proof
    - common subexpression elimination with value numbering
Vertical and horizontal composition

User proves:  \( \langle P \rangle \bowtie L_s \subseteq S \) and  \( \langle \{ L_i \} \rangle \subseteq L_s \)
Vertical and horizontal composition

User proves:  $\langle P \rangle \bowtie L_s \subseteq S$ and $\langle Li \rangle \subseteq L_s$

Horizontal composition:  $\langle P \rangle \bowtie \langle Li \rangle \subseteq \langle P \rangle \bowtie L_s$
Vertical and horizontal composition

User proves: \[\langle P \rangle \bowtie L_s \subseteq S\] and \[\langle Li \rangle \subseteq L_s\]

Horizontal composition: \[\langle P \rangle \bowtie \langle Li \rangle \subseteq \langle P \bowtie L_s \rangle\]

Vertical composition: \[\langle P \rangle \bowtie \langle Li \rangle \subseteq S\]
Vertical and horizontal composition

User proves:  $\langle P \rangle \bowtie Ls \subseteq S$ and $\langle Li \rangle \subseteq Ls$

Horizontal composition:  $\langle P \rangle \bowtie \langle Li \rangle \subseteq \langle P \rangle \bowtie Ls$

Vertical composition:  $\langle P \rangle \bowtie \langle Li \rangle \subseteq S$

Linking theorem:  $\langle P \cup Li \rangle \subseteq S$
Our contributions

1. Semantics of open modules
2. Semantic linking operator
3. Linking theorem
4. Compositional refinement
5. Refinement for memory-changing passes
Memory-changing transformations: \( \alpha \)-refinement

- CompCert refinement relation works for all passes that do not change memory.
- For other passes (e.g. C\#\text{minor-to-C\text{minor}}), argue that compilation “preserves information” through some bijection \( \alpha \)

Diagram:

```
Msource \[\text{Compiler pass}\] Mtarget
```
Memory-changing transformations: \( \alpha \)-refinement

- CompCert refinement relation works for all passes that do not change memory.
- For other passes (e.g. C#minor-to-Cminor), argue that compilation “preserves information” through some bijection \( \alpha \).
Memory-changing transformations: \( \alpha \)-refinement

\[(T_{\text{target}}, M_{\text{target}}) = \alpha(T_{\text{source}}, M_{\text{source}})\]

- For other passes (e.g. C\# minor-to-C minor), argue that compilation “preserves information” through some bijection \( \alpha \)
Memory-changing transformations: $\alpha$-refinement

$(T_{target}, M_{target}) = \alpha(T_{source}, M_{source})$

Source-level run-time invariant $J$ assumed to hold also for external modules

$\alpha$ injective from $
\{ (T_{source}, M_{source}) \mid J(T_{source}, M_{source}) \}$
Implementation

- Instantiated with CSE optimization and its CompCert proof
- Memory-changing transformation: C#minor to Cminor (local variable layout)
- Proofs available on the Internet
  - http://flint.cs.yale.edu/publications/vscl.html
Related work

- Hur et al. (POPL 2012)
  - coined “horizontal vs. vertical composition” problem
  - Based on parameterized operational semantics without intension
  - Local vs. global knowledge
- Stewart et al. (POPL 2015)
  - Based on operational small-step semantics: focus on simulation diagrams
  - Full-scale CompCert
  - We provide a linking theorem
  - Potential for simpler, less redundant proofs?
  - Go attend their talk on Friday at 10am, HBA!
Related work

• Perconti et al. (ESOP 2014)
  – Devices to unify different languages and compose their semantics
  – Can extend our linking theorem to cross-language linking

• Ghica et al. (MFPS 2012)
  – Based on game semantics
  – Opponent = unknown module to link with
Conclusion

- Language-independent compositional semantics and semantic linking operator
- Semantic vs. syntactic linking theorem
- Generalization of CompCert's event-based semantics
  - Point out minimal proof changes (at least in simpler settings)