A Compositional Semantics for Verified Separate Compilation & Linking

Tahina Ramananandro
Zhong Shao

Shu-Chun Weng
Jérémie Koenig

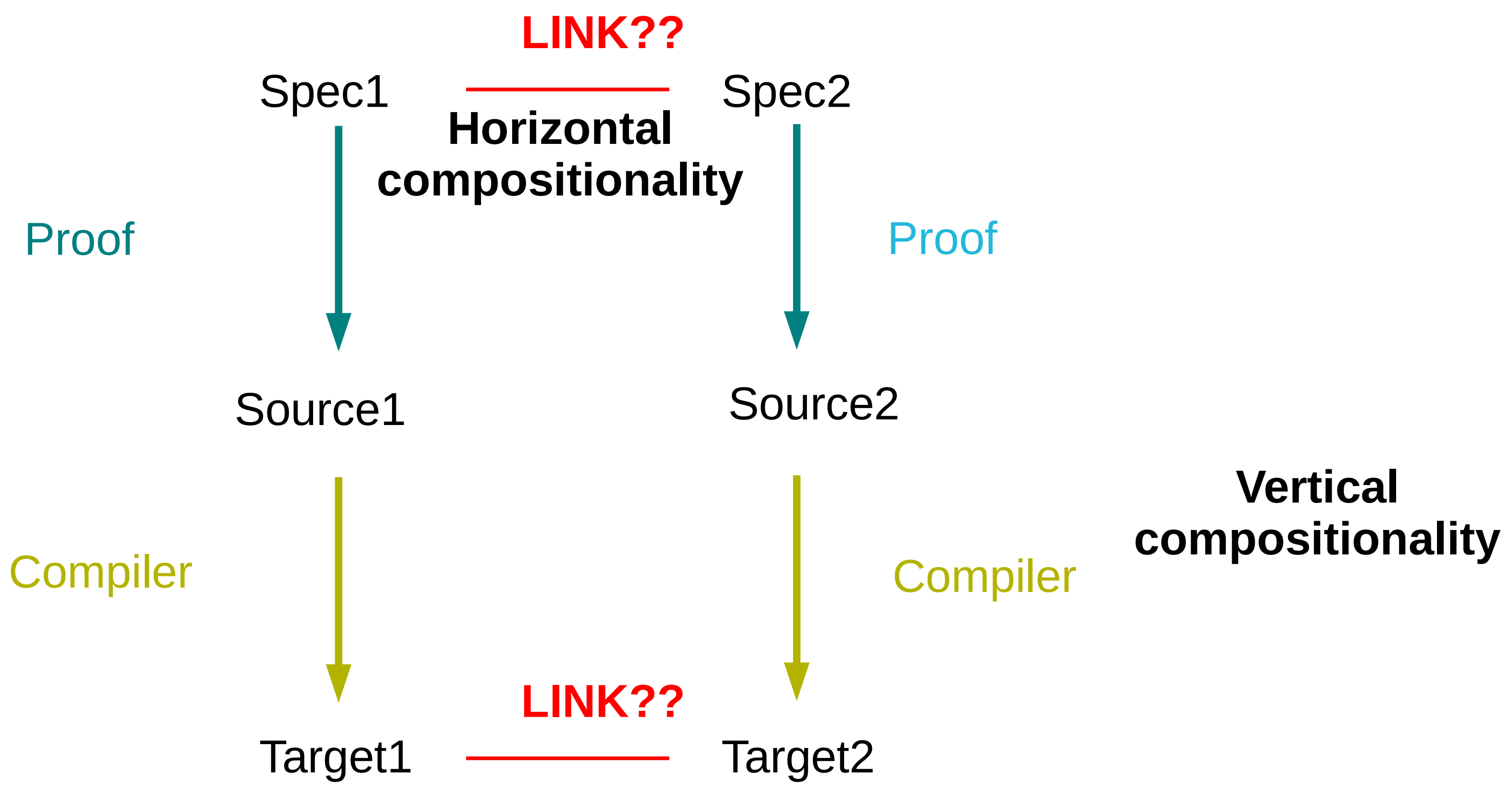
Yuchen Fu

Yale

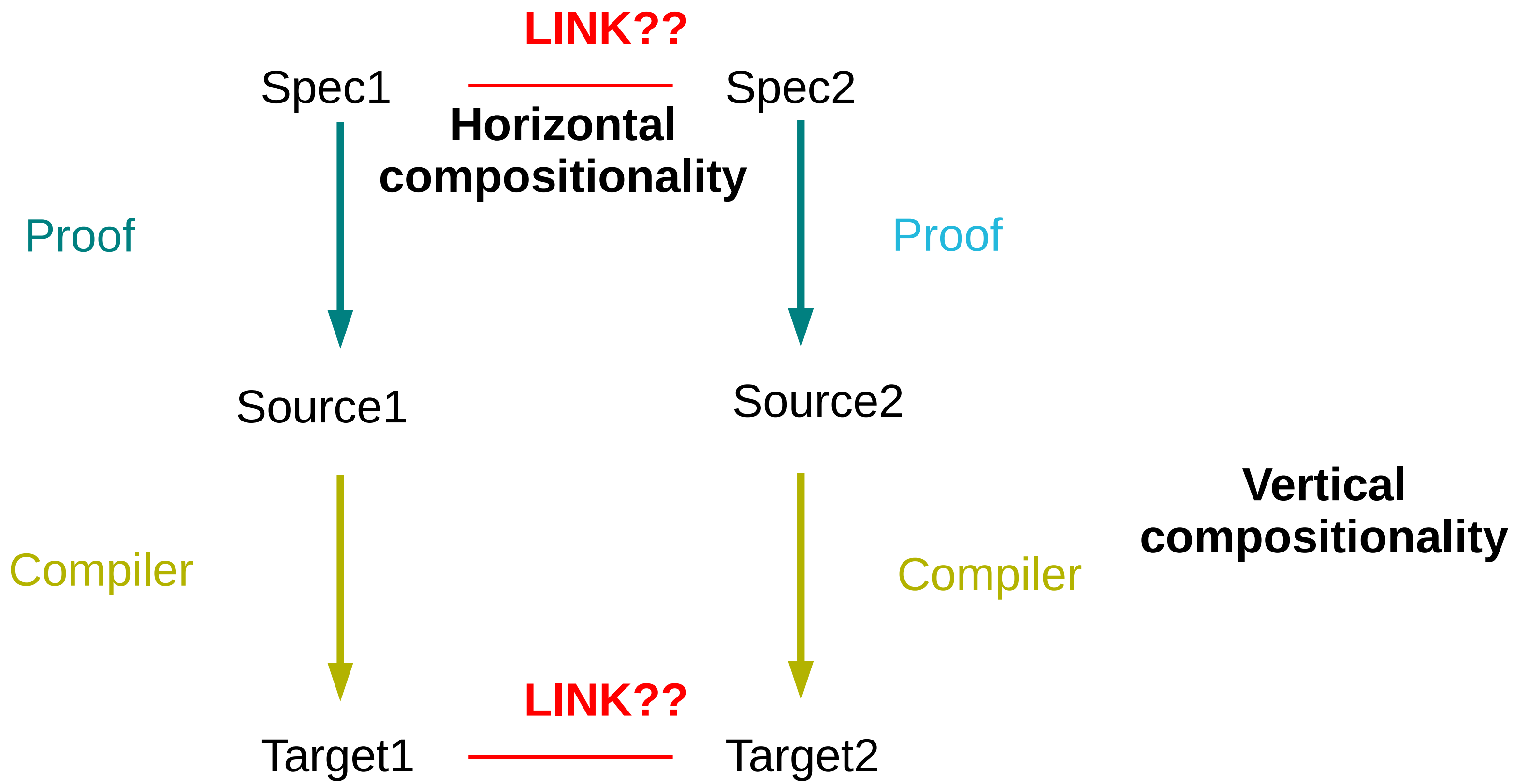
MIT

January 13th, 2015
Certified Programs and Proofs
TIFR, Mumbai, Maharashtra, India

Vertical vs. horizontal composition (Hur et al. POPL 2012)



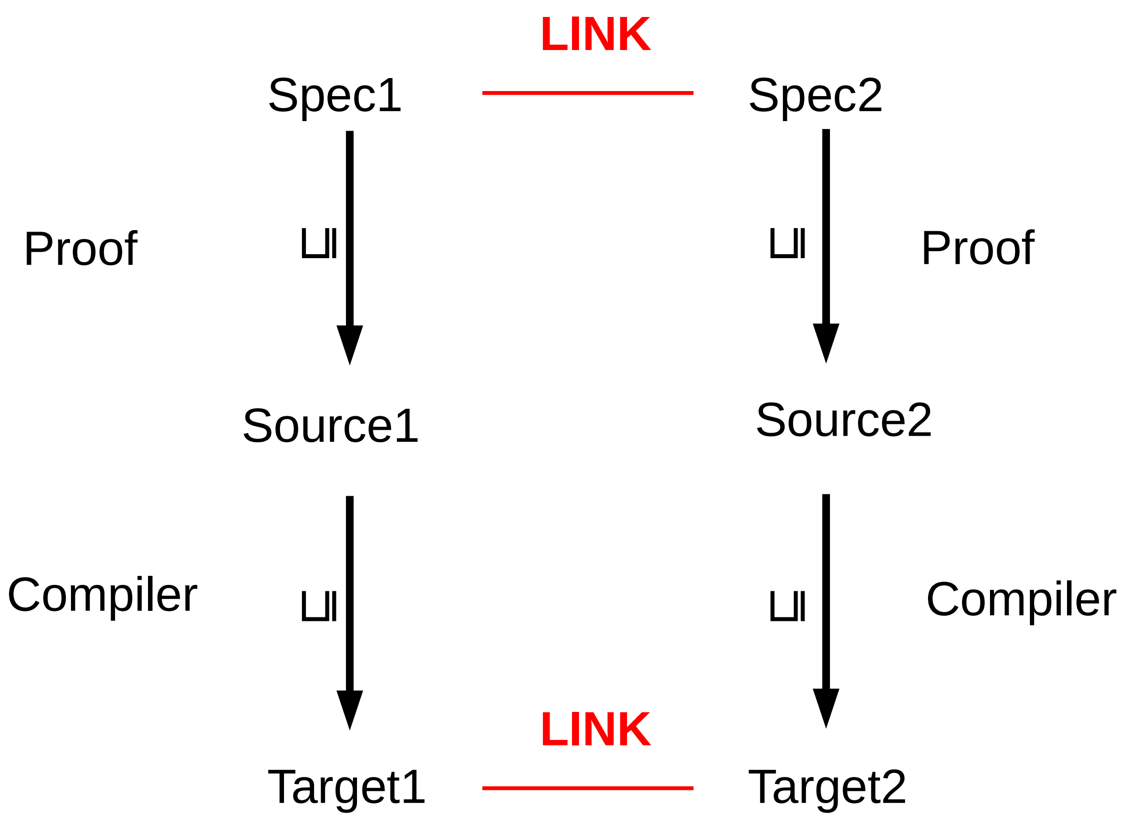
How to preserve component proofs by compilation and linking?

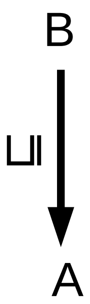


CompCert (Leroy POPL 2006) cannot preserve proofs by linking

- Works only for whole programs
- No correctness statement for open modules

Our unified approach: compositional semantics +refinement





A ⊑ B, reads “A refines B”
Based on a compositional semantics

<u>Research challenges</u>

- 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
- **[**Prog**]** ⊆

  {Terminates(eventList), Stuck(eventList), Diverges(eventList), Reacts(eventStream)}
- Compiler correctness: program behavior refinement:

  [Compiler(Prog)**]** ⊑ **[**Prog**]**

Examples

The C program...	… has the behavior

`int main () {`
`   printf('a');`
`   return 2;`
`}`
	OUT (a) . Terminates(2)

`int main () {`
`   printf('a');`
`   3/0;`
`   return 4;`
`}`
	OUT (a) . Stuck

`int main () {`
`   printf('a');`
`   while (1) {};`
`   return 5;`
`}`
	OUT (a) . Diverges

`int main () {`
`   while (1) { printf('b'); };`
`   return 6;`
`}`
	OUT (b) :: … :: OUT (b) :: … (Reacts)

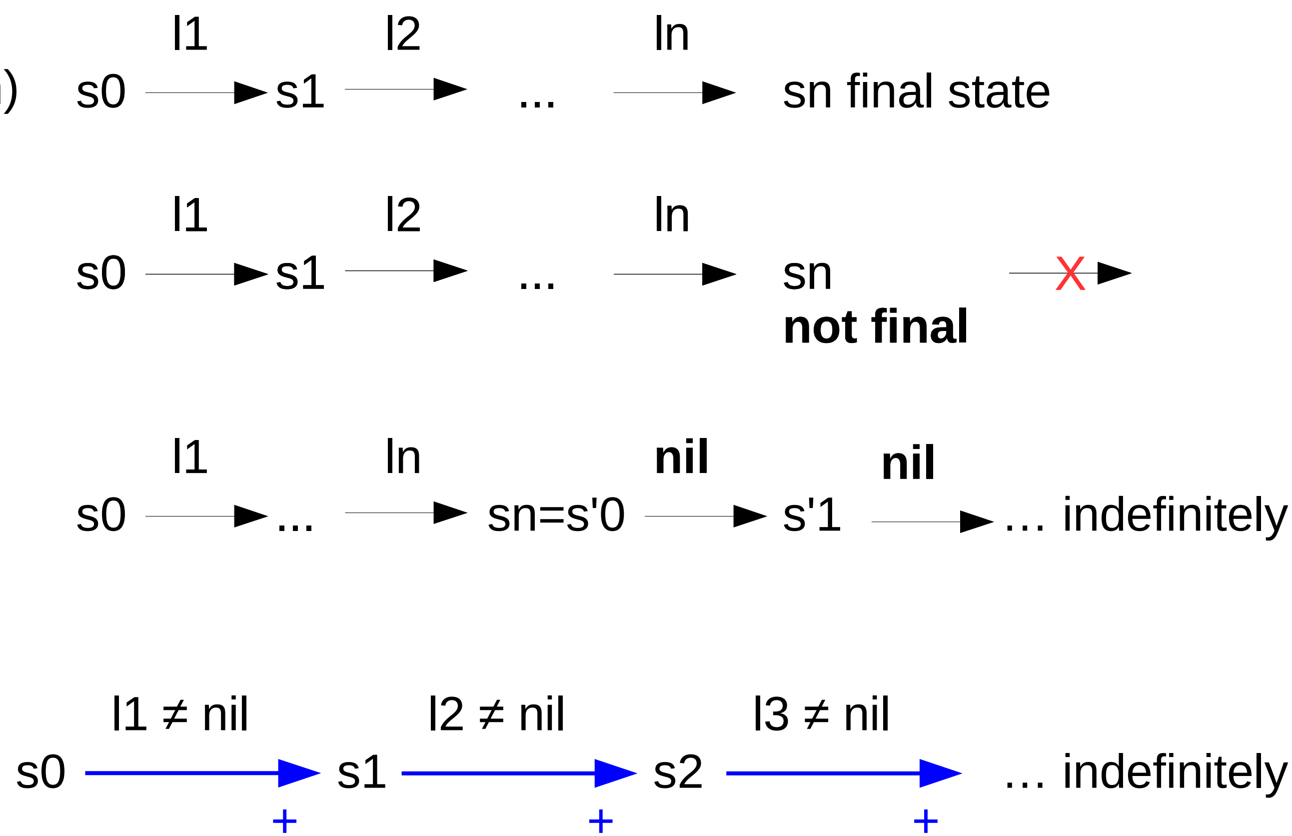
Big-stepping
the small-step semantics

Terminates(l1 ++ l2 ++ … ++ ln)

Stuck(l1 ++ l2 ++ … ++ ln)

Diverges(l1 ++ l2 ++ … ++ ln)

Reacts(l1 +++ l2 +++ ...)



How to deal with input?

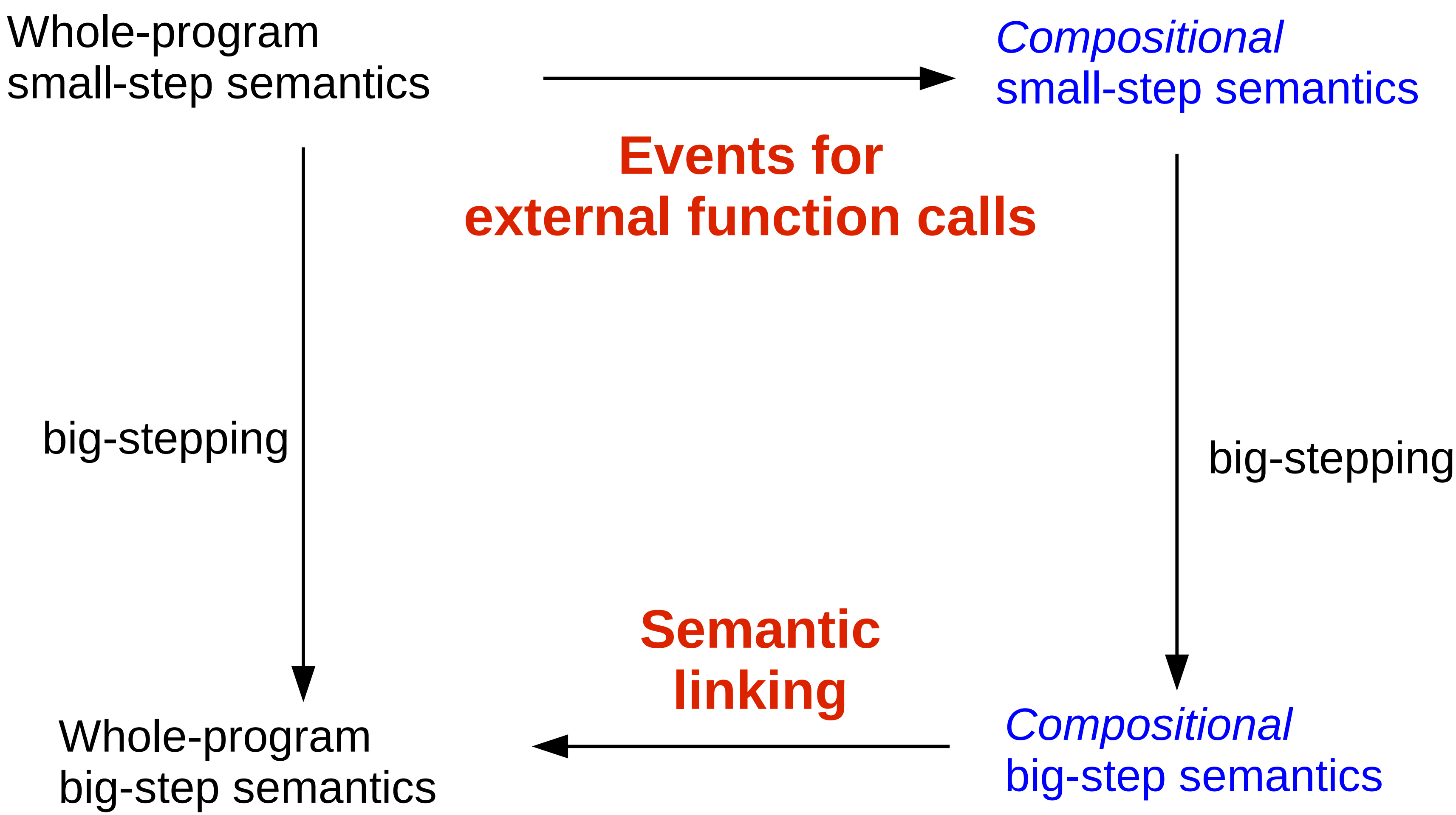
Provide a behavior for each possible input value.

The remaining behavior depends on that input value.

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

{
IN(a) :: OUT(a) . Terminates(0),
IN(b) :: OUT(b) . Terminates(0),
IN(c) :: OUT(c) . Terminates(0),
...}

From Operational to Compositional Semantics



Compositional semantics

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

m = {f ↦ `int x=18; int y=g(&x); printf("%d %d", y, x);` **}**

《 m 》 (f) = {
Extcall(g, [x->18], &x, 0, [x->0]) :: OUT 0 :: OUT 0 . Terminates,
Extcall(g, [x->18], &x, 0, [x->1]) :: OUT 0 :: OUT 1 . Terminates,
…
Extcall(g, [x->18], &x, 1, [x->0]) :: OUT 1 :: OUT 0 . Terminates,
Extcall(g, [x->18], &x, 1, [x->1]) :: OUT 1 :: OUT 1 . Terminates,
…}

Compositional semantics

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

m = {f ↦ `int x=18; int y=g(&x); printf("%d %d", y, x);` **}**

《 m 》 (f) = {

Extcall(g, [x->18], &x, 0, [x->0]) :: OUT 0 :: OUT 0 . Terminates,
Extcall(g, [x->18], &x, 0, [x->1]) :: OUT 0 :: OUT 1 . Terminates,
…
Extcall(g, [x->18], &x, 1, [x->0]) :: OUT 1 :: OUT 0 . Terminates,
Extcall(g, [x->18], &x, 1, [x->1]) :: OUT 1 :: OUT 1 . Terminates,
}

Callee

Memory state before call

Arguments

Compositional semantics

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

```
m = {f ↦ int x=18; int y=g(&x); printf("%d %d", y, x);}
```

《 m 》 (f) = {
Extcall(g, [x->18], &x, 0, [x->0]) :: OUT 0 :: OUT 0 . Terminates,
Extcall(g, [x->18], &x, 0, [x->1]) :: OUT 0 :: OUT 1 . Terminates,
…
Extcall(g, [x->18], &x, 1, [x->0]) :: OUT 1 :: OUT 0 . Terminates,
Extcall(g, [x->18], &x, 1, [x->1]) :: OUT 1 :: OUT 1 . Terminates,
}

Return value

Memory state after return

Compositional semantics

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

m' = {g(int* x**) ↦** `int y=*x; printf("%d", y-1); *x=y+1; return y;` **}**

《 m' 》 (g)(p)[p->*n*] = {
OUT (*n*-1) . 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

m1 = {f1 ↦ call f2}

《 m1 》 (f1) = Extcall(f2) Terminates

⋈

m2 = {f2 ↦ call f1}

《 m2 》 (f2) = Extcall(f1) Terminates

《 m1 》⋈《 m2 》 (f1) = {Diverges}
《 m1 》⋈《 m2 》 (f2) = {Diverges}

m1 ⊎ m2 = {f1 ↦ call f2, f2 ↦ call f1}

《 m1 ⊎ m2 》 = 《 m1 》⋈《 m2 》

- 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)

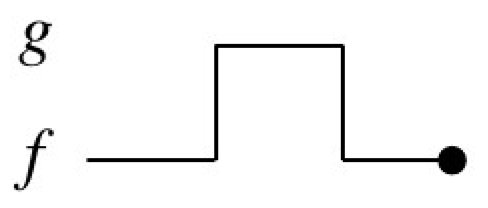
The linking operator

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

$$\psi 1 \bowtie \psi 2 = R(\psi 1 \uplus \psi 2)$$

The resolution operator
Example #1: Terminating case

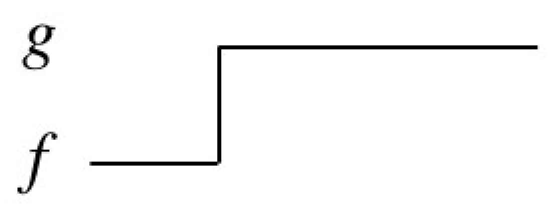
ψ = {g ↦ Terminates ;
f ↦ Extcall(g) . Terminates }



R(ψ) = {g ↦ Terminates ;
f ↦ Terminates }

The resolution operator
Example #2: Diverging case

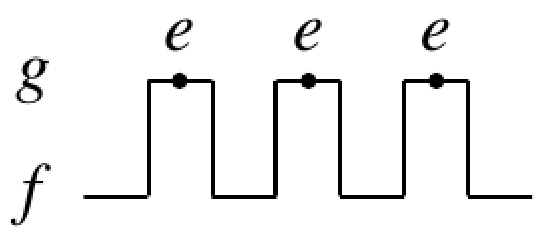
ψ = {g ↦ Diverges ;
f ↦ Extcall(g) :: Print 2 . Terminates }



R(ψ) = {g ↦ Diverges ;
f ↦ Diverges }

The resolution operator
Example #3: Infinitely many externals

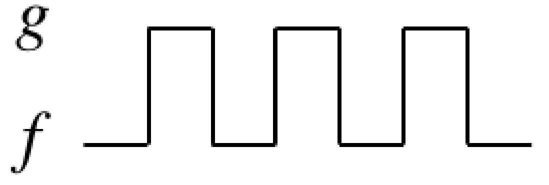
ψ = {g ↦ Print(e) . Terminates ;
f ↦ Extcall(g) :: Extcall(g) :: … :: Extcall(g) :: ... }



R(ψ) = {g ↦ Print(e) . Terminates ;
f ↦ Print(e) :: Print(e) :: … :: Print(e) :: ... }

The resolution operator
Example #4: Infinitely many externals

ψ = {g ↦ Terminates ;
f ↦ Extcall(g) :: Extcall(g) :: … :: Extcall(g) :: ... }

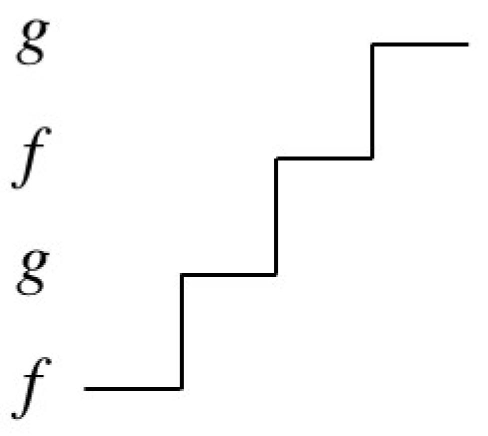


R(ψ) = {g ↦ Terminates ;
f ↦ Diverges }

Eager replacement will fail.

The resolution operator Example #5: (mutual) recursion

ψ = {g ↦ Extcall(f) :: Print(a) . Terminates ;
f ↦ Extcall(g) :: Print(b) . Terminates }



R(ψ) = {g ↦ Diverges ;
f ↦ Diverges }

How resolution works

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

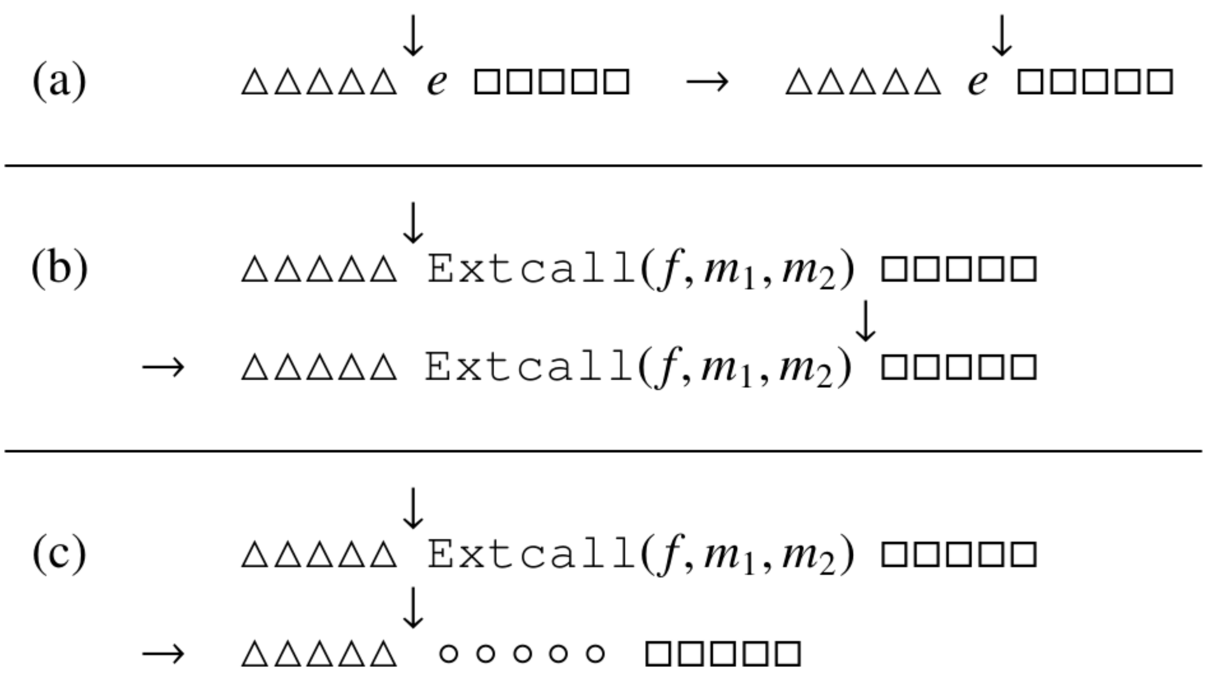


Fig. 1. Three cases in behavior simulation: (a) regular event; (b) $f \notin \mathrm{dom}(\psi)$; (c) $\circ\circ\circ\circ\circ \in \psi(f)$.

How resolution works

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

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

How resolution works

ψ = {g ↦ Print(e) . Terminates ;
f ↦ Extcall(g) :: Extcall(g) . Terminates }

How to compute R(ψ)(f) ?

Extcall(g) :: Extcall(g) . Terminates , []

How resolution works

ψ = {g ↦ Print(e) . Terminates ;
f ↦ Extcall(g) :: Extcall(g) . Terminates }

How to compute R(ψ)(f) ?

Extcall(g) :: Extcall(g) . Terminates , []

How resolution works

ψ = {g ↦ Print(e) . Terminates ;
f ↦ Extcall(g) :: Extcall(g) . Terminates }

How to compute R(ψ)(f) ?

Print(e) . Terminates , [Extcall(g) . Terminates]

How resolution works

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

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

Print(e) . Terminates , [Extcall(g) . Terminates]

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 , [Extcall(g) . Terminates]

Print(e) .

How resolution works

ψ = {g ↦ Print(e) . Terminates ;
f ↦ Extcall(g) :: Extcall(g) . Terminates }

How to compute R(ψ)(f) ?

Terminates , [Extcall(g) . Terminates]

Print(e) .

How resolution works

ψ = {g ↦ Print(e) . Terminates ;
f ↦ Extcall(g) :: Extcall(g) . Terminates }

How to compute R(ψ)(f) ?

Extcall(g) . Terminates , []

Print(e) .

How resolution works

ψ = {g ↦ Print(e) . Terminates ;
f ↦ Extcall(g) :: Extcall(g) . Terminates }

How to compute R(ψ)(f) ?

Extcall(g) . Terminates , []

Print(e) .

How resolution works

ψ = {g ↦ Print(e) . Terminates ;
f ↦ Extcall(g) :: Extcall(g) . Terminates }

How to compute R(ψ)(f) ?

Print(e) . Terminates , [Terminates]

Print(e) .

How resolution works

ψ = {g ↦ Print(e) . Terminates ;
f ↦ Extcall(g) :: Extcall(g) . Terminates }

How to compute R(ψ)(f) ?

Print(e) . Terminates , [Terminates]

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

ψ = {g ↦ Print(e) . Terminates ;
f ↦ Extcall(g) :: Extcall(g) . Terminates }

How to compute R(ψ)(f) ?

Terminates , [Terminates]

Print(e) :: Print(e) .

How resolution works

ψ = {g ↦ Print(e) . Terminates ;
f ↦ Extcall(g) :: Extcall(g) . Terminates }

How to compute R(ψ)(f) ?

Terminates , []

Print(e) :: 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

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

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 $R(\psi)(g)$?

Extcall(f) :: Print(a) . Terminates ,
[]

Resolution and (mutual) recursion

ψ = {g ↦ Extcall(f) :: Print(a) . Terminates ;
f ↦ Extcall(g) :: Print(b) . Terminates }

How to compute R(ψ)(g) ?

Extcall(g) :: Print(b) . Terminates ,
[Print(a) . Terminates]

Resolution and (mutual) recursion

ψ = {g ↦ Extcall(f) :: Print(a) . Terminates ;
f ↦ Extcall(g) :: Print(b) . Terminates }

How to compute R(ψ)(g) ?

Extcall(f) :: Print(a) . Terminates ,
[Print(b) . Terminates ;
Print(a) . Terminates]

Resolution and (mutual) recursion

ψ = {g ↦ Extcall(f) :: Print(a) . Terminates ;
f ↦ Extcall(g) :: Print(b) . Terminates }

How to compute R(ψ)(g) ?

Extcall(g) :: Print(b) . Terminates ,
[Print(a) . Terminates ;
Print(b) . Terminates ;
Print(a) . Terminates]

Resolution and (mutual) recursion

ψ = {g ↦ Extcall(f) :: Print(a) . Terminates ;
f ↦ Extcall(g) :: Print(b) . Terminates }

How to compute R(ψ)(g) ?

… and so on : it will diverge
[Print(b) . Terminates
Print(a) . Terminates ;
Print(b) . Terminates ;
Print(a) . 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

- iter f (x1 :: … :: xn :: nil)
- Semantics depends on symbol resolution and module in which iter is defined:
  - If f is an external function, then: Extcall(f) :: … :: 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

m1 = {f ↦ `int x=18; int y=g(&x); printf("%d %d", y, x);`**}**

《 m1 》 (f) = {
Extcall(g, [x↦18], &x, **0, [x↦0]**) :: print 0 :: print 0 . Terminates,
Extcall(g, [x↦18], &x, **0, [x↦1]**) :: print 0 :: print 1 . Terminates,
…
Extcall(g, [x↦18], &x, **1, [x↦0]**) :: print 1 :: print 0 . Terminates,
Extcall(g, [x↦18], &x, **1, [x↦1]**) :: print 1 :: print 1 . Terminates,
…}

Behavior simulation with return value and memory state

ψ1(f) = {
Extcall(g, [x↦18], &x, 0, [x↦0]) :: print 0 :: print 0 . Terminates(x↦0),
Extcall(g, [x↦18], &x, 0, [x↦1]) :: print 0 :: print 1 . Terminates(x↦1),
…
Extcall(g, [x↦18], &x, 1, [x↦0]) :: print 1 :: print 0 . Terminates(x↦0),
Extcall(g, [x↦18], &x, 1, [x↦1]) :: print 1 :: print 1 . Terminates(x↦1)
…}

$\bowtie$

m2 = {g ↦ `(int* px) *px = 1729; return 42;`**}**

《 m2 》 (g) = { Terminates(42, (x↦1729)) }

Behavior simulation with return value and memory state

Extcall(g, [x↦18], &x, 1, [x↦0]) :: print 1 :: print 0 . Terminates(x↦0), []

$\bowtie$

ψ2(g) = { Terminates(42, (x↦1729)) }

Behavior simulation with return value and memory state

Extcall(g, [x↦18], &x, 1, [x↦0]) :: print 1 :: print 0 . Terminates(x↦0), []

⋈

ψ2(g) = { Terminates(42, (x↦1729)) }

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

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

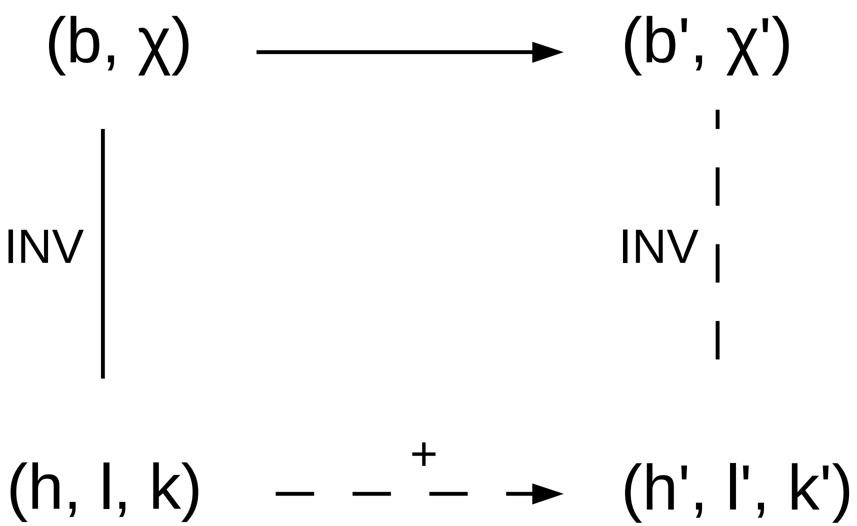
- How to relate behavior simulation with actual program linking?
  - Linking two modules written in the same language

$$\langle\langle\, \mathbf{m1} \uplus \mathbf{m0} \,\rangle\rangle = \langle\langle\, \mathbf{m1} \,\rangle\rangle \bowtie \langle\langle\, \mathbf{m0} \,\rangle\rangle$$

Mechanized Coq proof

《 m1 》 ⋈ 《 m0 》 ⊆ 《 m1 ⊎ m0 》

- Simulation diagram



Invariant INV

Stack: k = p ++ q

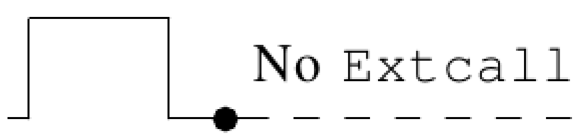
(h, l, p) behaves b in 《 m1 ⊎ m0 》

χ similarly matches q

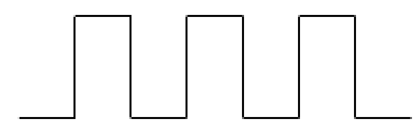
$\langle\!\langle\, m1 \uplus m0 \,\rangle\!\rangle \subseteq \langle\!\langle\, m1 \,\rangle\!\rangle \bowtie \langle\!\langle\, m0 \,\rangle\!\rangle$

- Easy induction for finite (terminating/stuck) behaviors
- For infinite (diverging/reacting) behaviors, distinguish between 3 cases:

Some external function call does not return



Finitely many external function calls that all terminate



Infinitely many external function calls that all terminate

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)
  - Beh1 improves Beh2 iff
    Beh1 = Beh2 or Beh2 is a stuck prefix of Beh1
- **Vertical composition:** Already known to be transitive
- **Horizontal composition:** Compatible with linking

$$\psi 1 \sqsubseteq \psi 2 \ \Rightarrow\ \psi \bowtie \psi 1 \sqsubseteq \psi \bowtie \psi 2$$

- **Mechanized (Coq) proof**

Compositional compiler correctness

$$\langle\!\langle \text{Compiler(Prog)} \rangle\!\rangle \sqsubseteq \langle\!\langle \text{Prog} \rangle\!\rangle$$

- 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: $\langle\langle P \rangle\rangle \bowtie Ls \sqsubseteq S$ and $\langle\langle Li \rangle\rangle \sqsubseteq Ls$

Vertical and horizontal composition

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

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

Vertical and horizontal composition

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

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

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

Vertical and horizontal composition

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

Horizontal composition: $\langle\!\langle P \rangle\!\rangle \bowtie \langle\!\langle Li \rangle\!\rangle \sqsubseteq \langle\!\langle P \rangle\!\rangle \bowtie Ls$

Vertical composition: $\langle\!\langle P \rangle\!\rangle \bowtie \langle\!\langle Li \rangle\!\rangle \sqsubseteq S$

$\parallel$

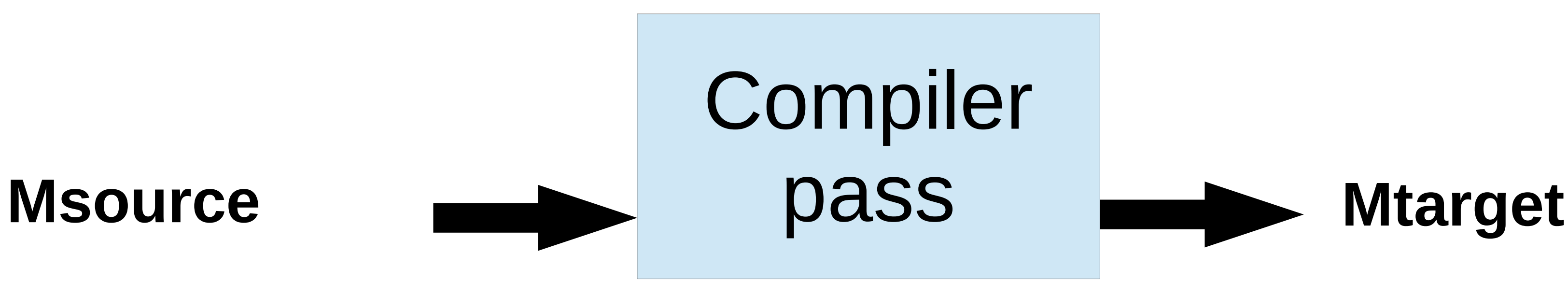
Linking theorem: $\langle\!\langle P \uplus Li \rangle\!\rangle \sqsubseteq S$

Our contributions

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

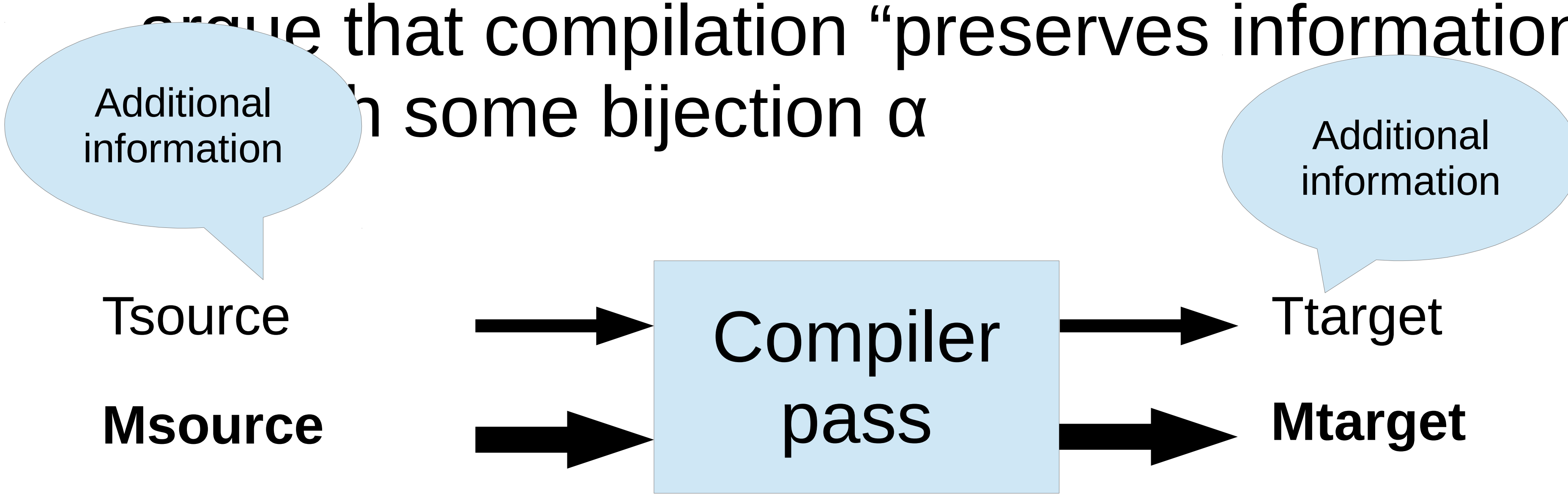
Memory-changing transformations: α-refinement

- 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 α



Memory-changing transformations: α-refinement

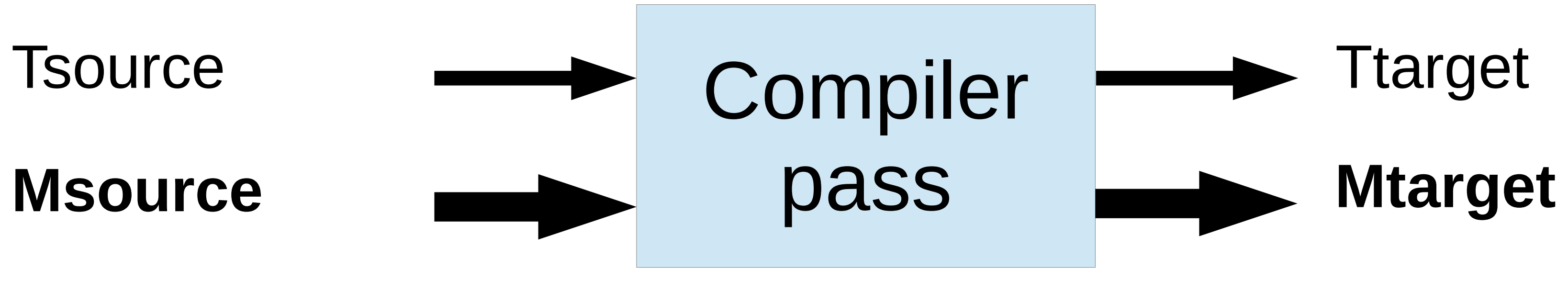
- CompCert refinement relation works for all passes that do not change memory
- For other passes (e.g. C#minor-to-Cminor), argue that compilation “preserves information” [illegible]h some bijection α



Memory-changing transformations: α-refinement

(Ttarget, Mtarget) = α(Tsource, Msource)

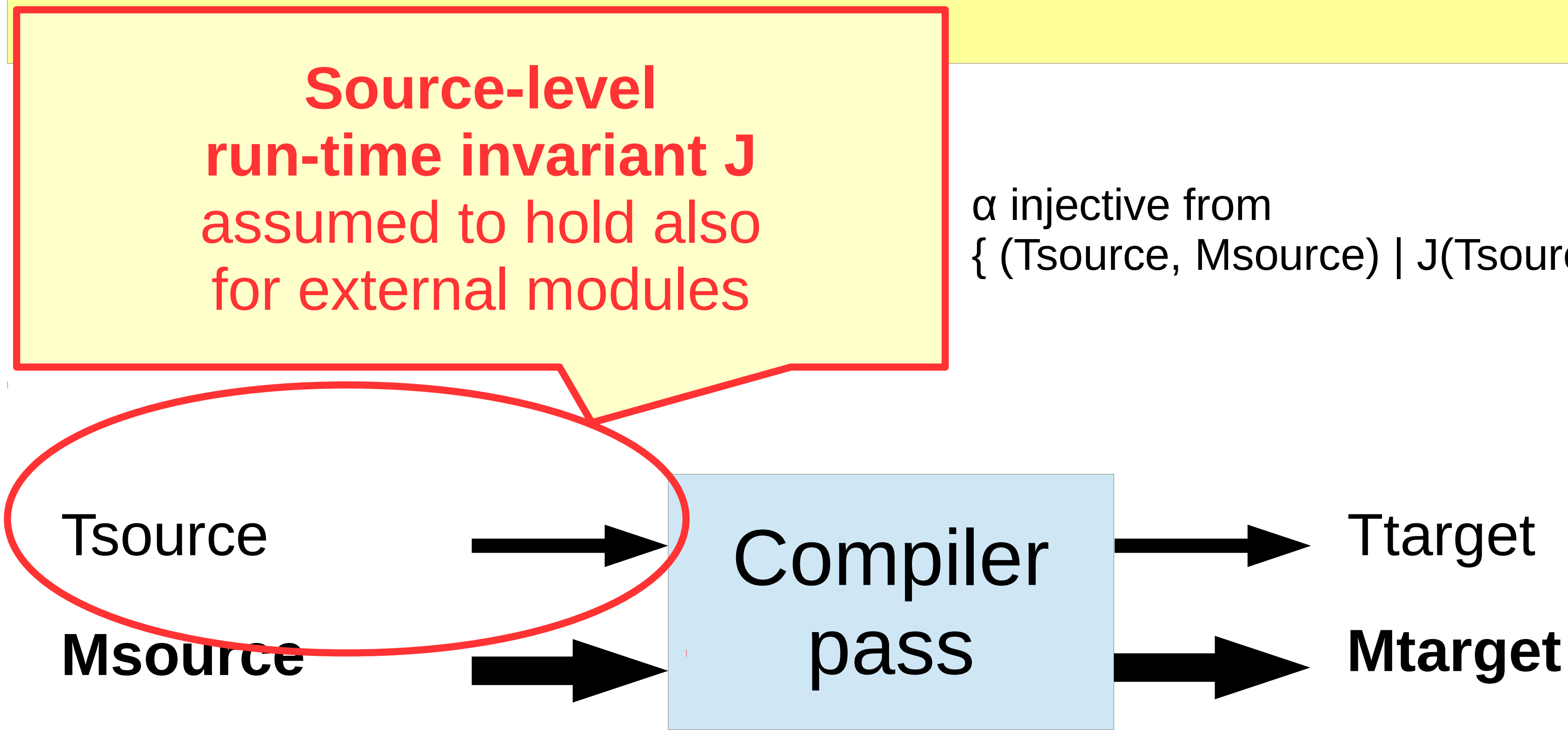
- For other passes (e.g. C#minor-to-Cminor), argue that compilation “preserves information” through some bijection α



Memory-changing transformations: α-refinement

(Ttarget, Mtarget) = α(Tsource, Msource)

α injective from
{ (Tsource, Msource) | J(Tsource, Msource)}



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)