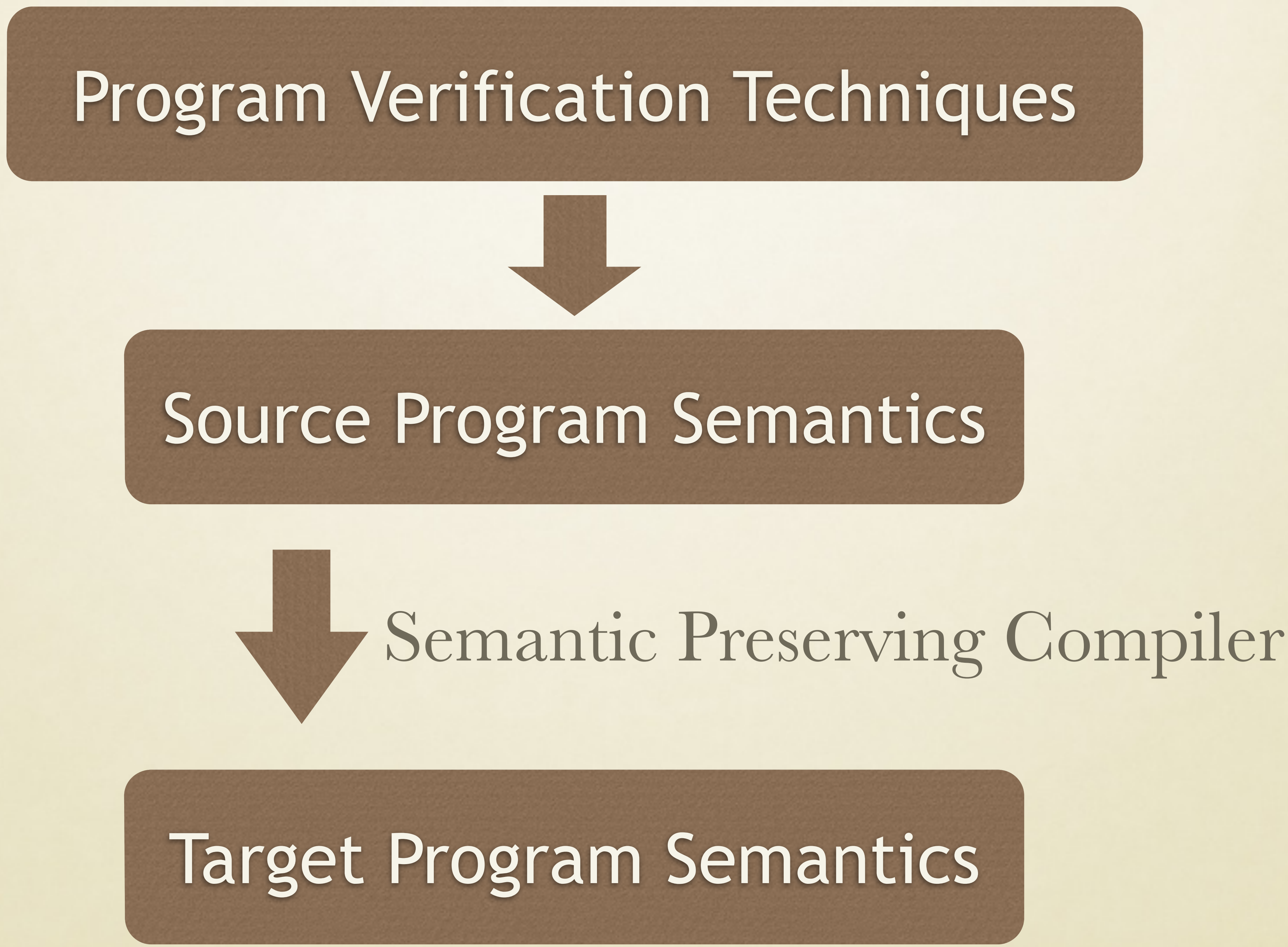
# Compiler Verification meets Cross-Language Linking via Data Abstraction

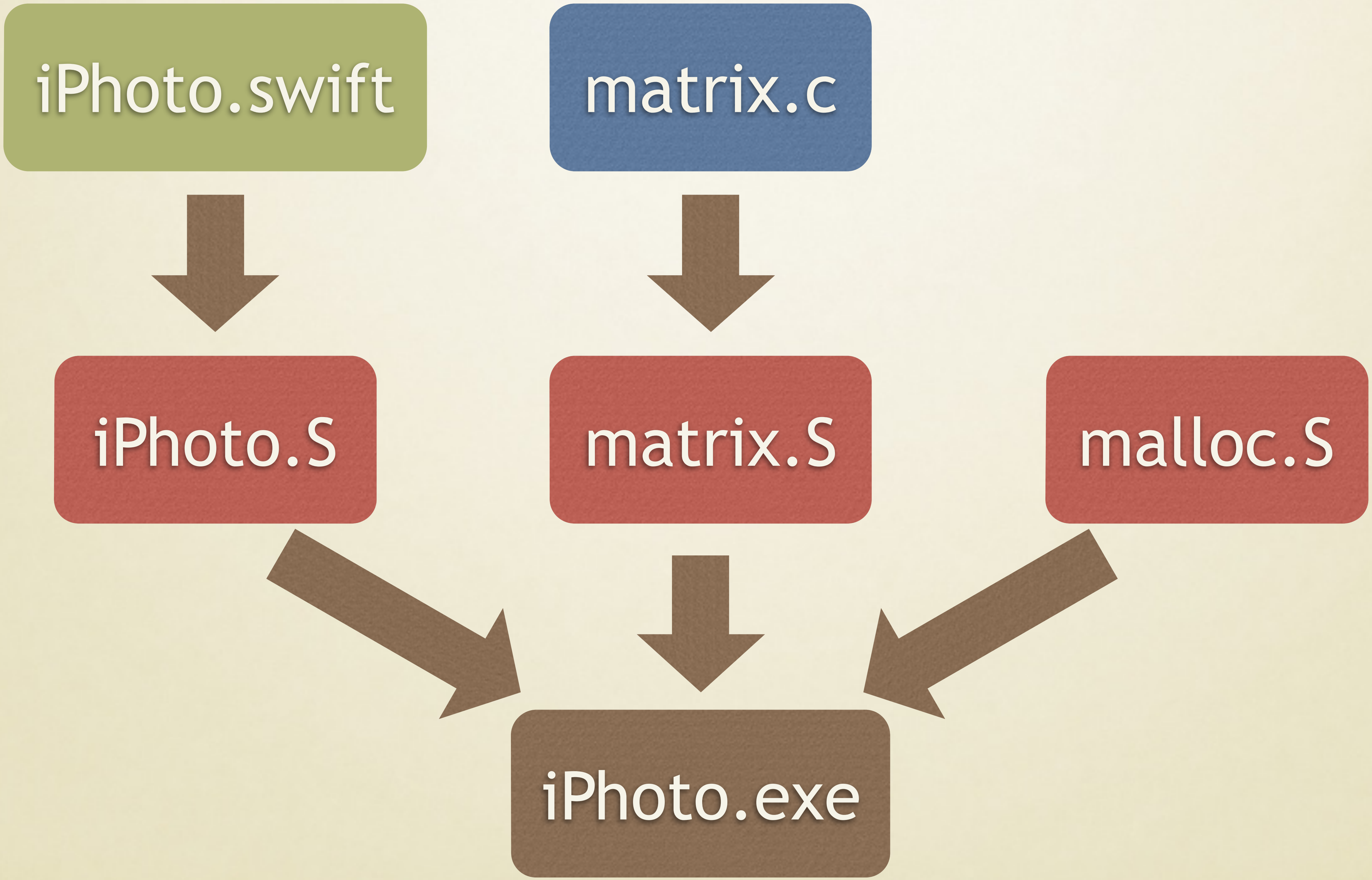
Peng Wang, MIT CSAIL
Santiago Cuellar, Princeton University
Adam Chlipala, MIT CSAIL

# Verified Compiler

# Verified Compiler



# Cross-Language Development



# ListSet.ll:

```
typedef /* ... */ ListSet;
ListSet ListSet_new() { /* ... */ }
void ListSet_delete(ListSet this) { /* ... */ }
void ListSet_add(ListSet this, int key) { /* ... */ }
int ListSet_size(ListSet this) { /* ... */ }
```

# CountUnique.hl:

```
int countUnique(int[] arr) {
  Set set = new ListSet();
  for (int i = 0; i < arr.length(); ++i)
    set.add(arr[i]);
  int ret = set.size();
  delete set;
  return ret;
}
```

- **Higher-level language:**
    - Memory of ADTs
    - Can call externally defined functions
    - Java/C++ like
- **Lower-level language:**
    - Memory of machine words
    - Assembly like

- **Higher-level language:**
  - Memory of ADTs
  - Can call externally defined functions
  - Java/C++ like
  - Expr/If/While/Call
  - Function pointers
- **Lower-level language:**
  - Memory of machine words
  - Assembly like

- **Higher-level language:**
  - Memory of ADTs
  - Can call externally defined functions
  - Java/C++ like
  - Expr/If/While/Call
  - Function pointers

**Cito**

- **Lower-level language:**
  - Memory of machine words
  - Assembly like

**Bedrock IL**

# Cito Syntax

**Notation:**

| | | | |
|---|---|---|---|
| Optional | $\lfloor \cdot \rfloor$ | List Of | $(\cdot)^*$ |
| Product Type | $\times$ | Sum Type | $+$ |
| Machine Word | $\mathbb{W}$ | String | $\mathbb{S}$ |

**Syntax:**

| | | | |
|---|---|---|---|
| Constant | $w$ | $\in$ | $\mathbb{W}$ |
| Label | $l$ | $\in$ | $\mathbb{S}_{\text{module}} \times \mathbb{S}_{\text{fun}}$ |
| Variable | $x$ | $\in$ | $\mathbb{S}$ |
| Binary Op | $o$ | $::=$ | $+ \mid - \mid \times \mid = \mid \neq \mid < \mid \leq$ |
| Expression | $e$ | $::=$ | $x \mid w \mid e\ o\ e$ |
| Statement | $s$ | $::=$ | $\mathsf{skip} \mid s; s \mid \mathsf{if}\ e\ \{s\}\ \mathsf{else}\ \{s\}$ $\mid \mathsf{while}\ e\ \{s\} \mid x := \mathsf{call}\ e\ (e^*)$ $\mid x := e \mid x := \mathsf{label}\ l$ |
| Function | $f$ | $\in$ | $\mathbb{S}^*_{\text{arg}} \times \mathbb{S}_{\text{ret}} \times s$ |
| Module | $m$ | $\in$ | $\mathbb{S}_{\text{name}} \times (\mathbb{S}_{\text{fname}} \times f)^*$ |

**State:**

| | | | |
|---|---|---|---|
| Machine State ($\Sigma$) | | $=$ | $E \times H$ |
| Variable Assignment ($\sigma$) | $E$ | $=$ | $\mathbb{S} \to \mathbb{W}$ |
| Heap ($\mu$) | $H$ | $=$ | $\mathbb{W} \to \lfloor A \rfloor$ |
| ADT Domain | $A$ | $=$ | [parameter of theory] |

# Bedrock IL Syntax

**Syntax:**

| | | | |
|---|---|---|---|
| Constants | $c$ | $::=$ | [fixed-width bitvectors] |
| Code labels | $\ell$ | $::=$ | ... |
| Registers | $r$ | $::=$ | $\mathsf{Sp} \mid \mathsf{Rp} \mid \mathsf{Rv}$ |
| Addresses | $a$ | $::=$ | $r \mid c \mid r + c$ |
| Lvalues | $L$ | $::=$ | $r \mid [a]_8 \mid [a]_{32}$ |
| Rvalues | $R$ | $::=$ | $L \mid c \mid \ell$ |
| Binops | $o$ | $::=$ | $+ \mid - \mid \times$ |
| Tests | $t$ | $::=$ | $= \mid \neq \mid < \mid \leq$ |
| Instructions | $i$ | $::=$ | $L \leftarrow R \mid L \leftarrow R\ o\ R$ |
| Jumps | $j$ | $::=$ | $\mathsf{goto}\ R \mid \mathsf{if}\ (R\ t\ R)\ \mathsf{then}\ \ell\ \mathsf{else}\ \ell$ |
| Blocks | $B$ | $::=$ | $\ell : \{\lambda\gamma.\ \phi\}\ i^*; j$ |
| Modules | $M$ | $::=$ | $B^*$ |

**State:**

| | | |
|---|---|---|
| Machine State | $=$ | Memory $\times$ Registers $\times$ ProgramCounter |
| Memory | $=$ | $\mathbb{W} \to \mathbb{W}$ |
| Registers | $=$ | $r \to \mathbb{W}$ |
| ProgramCounter | $=$ | $\mathsf{Pc} \to \mathbb{W}$ |

# Bedrock IL Syntax

Syntax:

| | | | |
|---|---|---|---|
| Constants | $c$ | $::=$ | [fixed-width bitvectors] |
| Code labels | $\ell$ | $::=$ | ... |
| Registers | $r$ | $::=$ | $\mathsf{Sp} \mid \mathsf{Rp} \mid \mathsf{Rv}$ |
| Addresses | $a$ | $::=$ | $r \mid c \mid r + c$ |
| Lvalues | $L$ | $::=$ | $r \mid [a]_8 \mid [a]_{32}$ |
| Rvalues | $R$ | $::=$ | $L \mid c \mid \ell$ |
| Binops | $o$ | $::=$ | $+ \mid - \mid \times$ |
| Tests | $t$ | $::=$ | $= \mid \neq \mid < \mid \leq$ |
| Instructions | $i$ | $::=$ | $L \leftarrow R \mid L \leftarrow R \; o \; R$ |
| Jumps | $j$ | $::=$ | $\mathsf{goto}\ R \mid \mathsf{if}\ (R\ t\ R)\ \mathsf{then}\ \ell\ \mathsf{else}\ \ell$ |
| Blocks | $B$ | $::=$ | $\ell : \{\lambda\gamma.\ \phi\}\ i^*; j$ |
| Modules | $M$ | $::=$ | $B^*$ |

State:

| | | |
|---|---|---|
| Machine State | $=$ | Memory $\times$ Registers $\times$ ProgramCounter |
| Memory | $=$ | $\mathbb{W} \rightarrow \mathbb{W}$ |
| Registers | $=$ | $r \rightarrow \mathbb{W}$ |
| ProgramCounter | $=$ | $\mathsf{Pc} \rightarrow \mathbb{W}$ |

- **Higher-level language:**
  - Memory of ADTs
  - Can call externally defined functions
  - Java/C++ like
  - Expr/If/While/Call
  - Function pointers

**Cito**

- **Lower-level language:**
  - Memory of machine words
  - Assembly like

**Bedrock IL**

- **Higher-level language:**
  - Memory of ADTs
  - Can call externally defined functions
  - Java/C++ like
  - Expr/If/While/Call
  - Function pointers

**Cito**

- **Lower-level language:**
  - Memory of machine words
  - Assembly like

**Bedrock IL**

# Semantics of Call

$$\frac{\Psi(\llbracket e_f \rrbracket) = \text{spec of } f}{\Psi \vdash (\Sigma, x := \mathsf{call}\ e_f\ (e^*)) \Downarrow \Sigma''}$$

- Environment ($\Psi$) : Function address → Function specification
- Function specification:
  - **Operational**: callee's body
  - **Axiomatic**: relation of pre-call and post-call state

# Operational vs. Axiomatic

| Operational Specification | Axiomatic Specification |
|---|---|
| ☹ Language dependent | ☺ Language independent |
| ☺ No annotation burden | ☹ Need to be provided |
| Suitable for **intra**-language calls | Suitable for **inter**-language calls |

# Operational vs. Axiomatic

| Operational Specification | Axiomatic Specification |
|---|---|
| ☹ Language dependent | ☺ Language independent |
| ☺ No annotation burden | ☹ Need to be provided |
| Suitable for **intra**-language calls | Suitable for **inter**-language calls |

We allow both!

# Prove Semantic Preservation

- **Method 1**: Prove simulation between source's and target's operational semantics
- **Method 2**: Express semantic preservation in a program logic for the **target** language

# Prove Semantic Preservation

- **Method 1**: Prove simulation between source's and target's operational semantics
- **Method 2**: Express semantic preservation in a program logic for the **target** language

$$\frac{\text{side-conditions}}{\{\text{pre-cond}\}\ \text{code}\ \{\text{post-cond}\}}$$

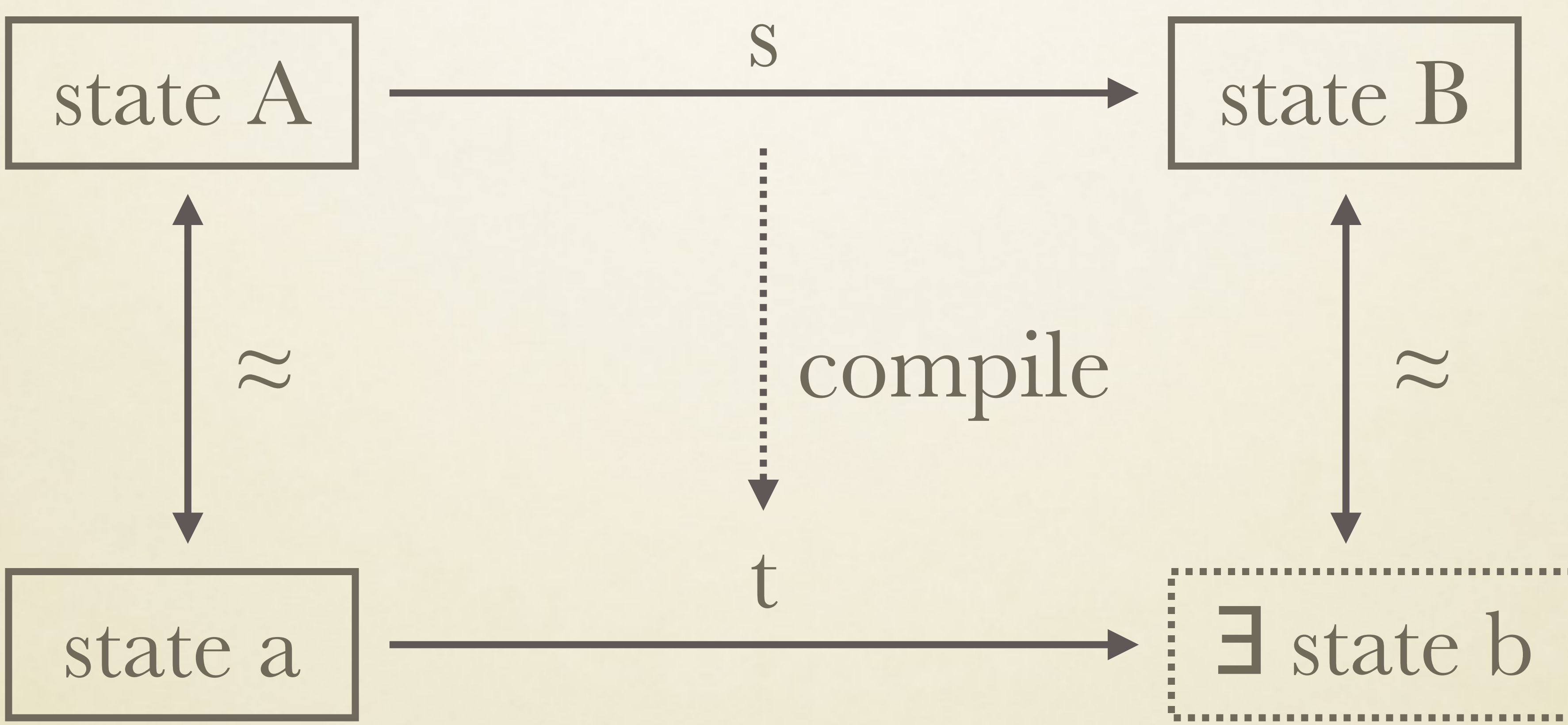
# Prove Semantic Preservation

- **Method 1**: Prove simulation between source's and target's operational semantics
- **Method 2**: Express semantic preservation in a program logic for the **target** language

$$\frac{\text{side-conditions}}{\{\text{pre-cond}\}\ \text{code}\ \{\text{post-cond}\}}$$

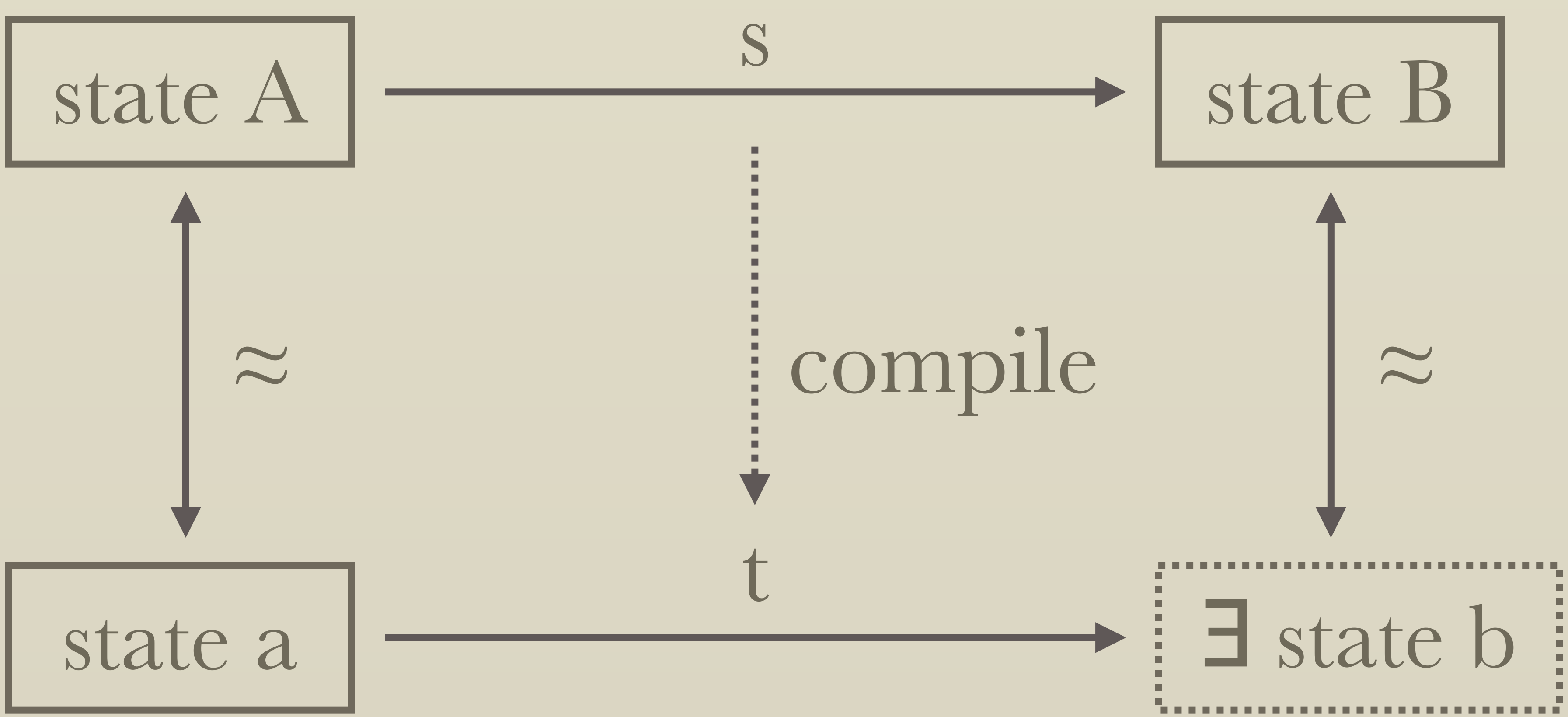
$$\frac{\mathsf{compile}(s) = t}{\{\text{safe to run } s\}\ t\ \{\text{state could result from running } s\}}$$

$$\frac{\mathsf{compile}(s) = t}{\{\text{safe to run } s\}\ t\ \{\text{state could result from running } s\}}$$



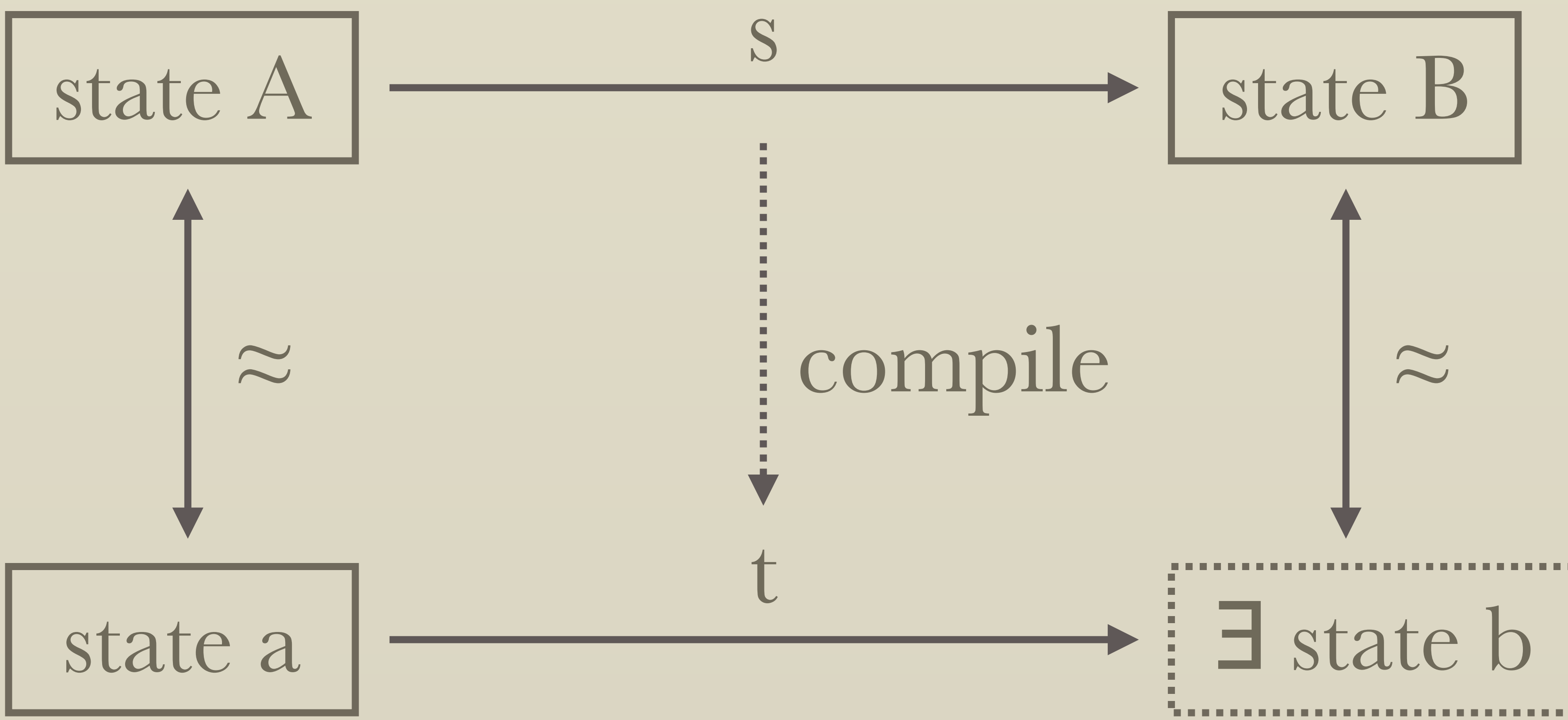
$$\frac{\mathsf{compile}(s) = t}{\{\text{safe to run } s\}\ t\ \{\text{state could result from running } s\}}$$

∀ A,B,a,b,s,t. safe(A,s) ⇒



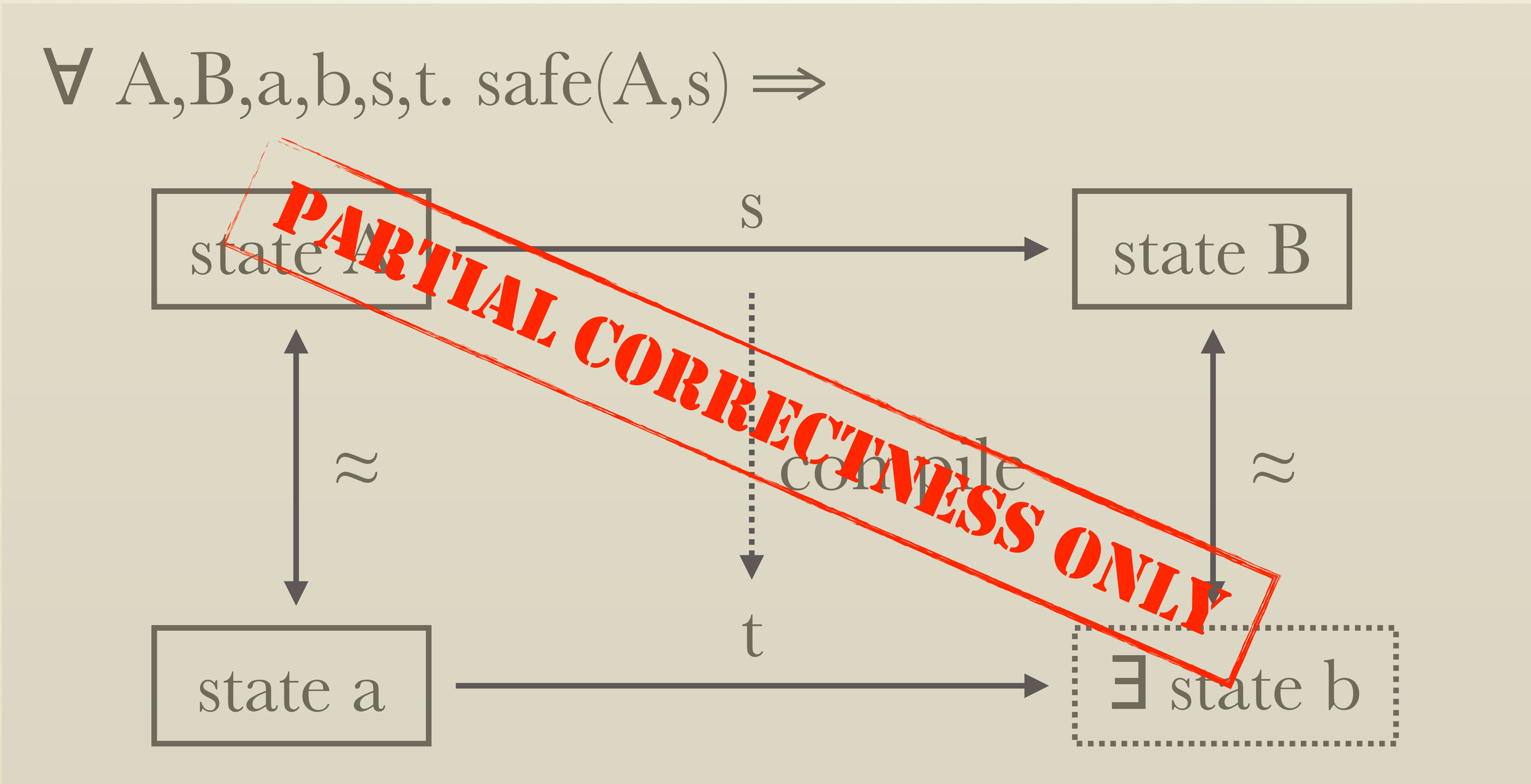
$$\frac{\textsf{compile}(s) = t}{\{\text{safe to run } s\}\ t\ \{\text{state could result from running } s\}}$$

∀ A,B,a,b,s,t. safe(A,s) ⇒




The main **compiler correctness theorem**

$$\frac{\mathsf{compile}(s) = t}{\{\text{safe to run } s\}\ t\ \{\text{state could result from running } s\}}$$




The main **compiler correctness theorem**

## ListSet.ll:

```
typedef /* ... */ ListSet;
ListSet ListSet_new() { /* ... */ }
void ListSet_delete(ListSet this) { /* ... */ }
void ListSet_add(ListSet this, int key) { /* ... */ }
int ListSet_size(ListSet this) { /* ... */ }
```

## CountUnique.hl:

```
int countUnique(int[] arr) {
  Set set = new ListSet();
  for (int i = 0; i < arr.length(); ++i)
    set.add(arr[i]);
  int ret = set.size();
  delete set;
  return ret;
}
```

## ListSet.ll:

```
typedef /* ... */ ListSet;
ListSet ListSet_new() { /* ... */ }
void ListSet_delete(ListSet this) { /* ... */ }
void ListSet_add(ListSet this, int key) { /* ... */ }
int ListSet_size(ListSet this) { /* ... */ }
```

**Abstract Data Types** (ADTs) are a natural interface between languages

## CountUnique.hl:

```
int countUnique(int[] arr) {
  Set set = new ListSet();
  for (int i = 0; i < arr.length(); ++i)
    set.add(arr[i]);
  int ret = set.size();
  delete set;
  return ret;
}
```

# ADT

- ADT objects are blackboxes, assessed only by axiomatically specified methods
- Object state is specified by a functional(mathematical) model
- Methods can:
  - **return** new object
  - in-place **modify** arguments
  - **delete** arguments
- Example: Set
  - **model**: mathematical set of integers
  - {} `new`() {return set ∅}
  - {x is a set} `delete`(x) {x is deleted}
  - {x is set *s*} `size`(x) {x is **still** set *s*, return |*s*|}
  - {x is set *s*} `add`(x, w) {x is set *s* ∪ {w} }

# ADT

- ADT objects are blackboxes, assessed only by axiomatically specified methods
- Object state is specified by a functional(mathematical) model
- Methods can:
  - **return** new object
  - in-place **modify** arguments
  - **delete** arguments
- Example: Set
  - **model**: mathematical set of integers
    $A = \mathsf{FSET}(\mathbb{P}) + \cdots \qquad \mathbb{P} = \mathcal{P}(\mathbb{W})$ (* sets of machine inteers *)
  - $\{\}$ `new`() {return set $\varnothing$}
    $\{\lambda I.\ I = []\}$ `new` $\{\lambda(O,R).\ O = [] \wedge R = \mathsf{ADT}(\mathsf{FSET}(\varnothing))\}$
  - {x is a set} `delete`(x) {x is deleted}
    $\{\lambda I.\ I = [\mathsf{ADT}(\mathsf{FSET}(\cdot))]\}$ `delete` $\{\lambda(O,R).\ O = [(\mathsf{ADT}(\mathsf{FSET}(\cdot)), \bot)] \wedge R = \mathsf{SCA}(\cdot)\}$
  - {x is set $s$} `size`(x) {x is **still** set $s$, return $|s|$ }
    $\{\lambda I.\ I = [\mathsf{ADT}(\mathsf{FSET}(\cdot))]\}$ `size` $\{\lambda(O,R).\ \exists s.\ O = [(\mathsf{ADT}(\mathsf{FSET}(s)), \mathsf{FSET}(s))] \wedge R = \mathsf{SCA}(|s|)\}$
  - {x is set $s$} `add`(x, w) {x is set $s \cup$ {w} }
    $\{\lambda I.\ I = [\mathsf{ADT}(\mathsf{FSET}(\cdot)), \mathsf{SCA}(\cdot)]\}$ `add` $\{\lambda(O,R).\ \exists s,w.\ O = [(\mathsf{ADT}(\mathsf{FSET}(s)), \mathsf{FSET}(s \cup \{w\})),$
    $(\mathsf{SCA}(w), \bot)] \wedge R = \mathsf{SCA}(\cdot)\}$

CountUnique.v:

```
Definition count :=
 cmodule "count" {{
   cfunction "count"("arr", "len") return "ret"
     "set" <-- Call "ListSet"!"new"();; "i" <- 0;;



     While ("i" < "len") {
       "e" <-- Call "ArraySeq"!"read" ("arr", "i");;
       Call "ListSet"!"add"("set", "e");; "i" <- "i" + 1
     };;
     "ret" <-- Call "ListSet"!"size"("set");;
     Call "ListSet"!"delete"("set")
   end
 }}.
```

**Steps:**

1. Write a Cito program

ExampleADT.v:

```
Inductive ADTModel :=
  | Arr : list W -> ADTModel
  | FSet : MSet.t W -> ADTModel
  ...
Definition ListSet_addSpec :=
  PRE[I] exists s n, I = [ADT (FSet s), SCA n]
  POST[O, R] exists s n any, O = [(ADT (FSet s), Some (FSet (add n s))), (SCA n, None)] /\ R = SCA any.
```

CountUnique.v:

```
Definition imports := [
   ("ArraySeq"!"read", ArraySeq_readSpec),
   ("ListSet"!"add", ListSet_addSpec), ... ]
Definition count :=
 cmodule "count" {{
   cfunction "count"("arr", "len") return "ret"
     "set" <-- Call "ListSet"!"new"();; "i" <- 0;;



     While ("i" < "len") {
       "e" <-- Call "ArraySeq"!"read" ("arr", "i");;
       Call "ListSet"!"add"("set", "e");; "i" <- "i" + 1
     };;
     "ret" <-- Call "ListSet"!"size"("set");;
     Call "ListSet"!"delete"("set")
   end
 }}.
Definition count_compil := compile count imports.
Theorem count_ok : moduleOk count_compil.
  compile_ok.
Qed.
```

**Steps:**

1. Write a Cito program
2. Provide ADT specifications

Compiler already usable, no programmer annotation burden

ExampleADT.v:

```
Inductive ADTModel :=
  | Arr : list W -> ADTModel
  | FSet : MSet.t W -> ADTModel
  ...
Definition ListSet_addSpec :=
  PRE[I] exists s n, I = [ADT (FSet s), SCA n]
  POST[O, R] exists s n any, O = [(ADT (FSet s), Some (FSet (add n s))), (SCA n, None)] /\ R = SCA any.
```

CountUnique.v:

```
Definition count_spec :=
  PRE[I] exists arr len, I = [ADT (Arr arr), SCA len] /\ len = length arr
  POST[O, R] exists arr, O[0] = (ADT (Arr arr), Some (Arr arr)) /\ R = SCA (count_unique arr).
Definition imports := [
   ("ArraySeq"!"read", ArraySeq_readSpec),
   ("ListSet"!"add", ListSet_addSpec), ... ]
Definition count :=
 cmodule "count" {{ [count_spec]
   cfunction "count"("arr", "len") return "ret"
     "set" <-- Call "ListSet"!"new"();; "i" <- 0;;
     [INIT (V, H) NOW (V', H') exists arr fset,
        find (V "arr") H = Some (Arr arr) /\
        H' == H * (V' "set" -> FSet fset) /\
        fset == to_set (firstn (V' "i") arr)]
     While ("i" < "len") {
       "e" <-- Call "ArraySeq"!"read" ("arr", "i");;
       Call "ListSet"!"add"("set", "e");; "i" <- "i" + 1
     };;
     "ret" <-- Call "ListSet"!"size"("set");;
     Call "ListSet"!"delete"("set")
   end
 }}.
Definition count_compil := compile count imports.
Theorem count_ok : moduleOk count_compil.
  compile_ok.
Qed.
```

**Steps:**

1. Write a Cito program

2. Provide ADT specifications

3*. Prove some property of the program, using **any** verification technique (e.g. a program logic)

# Proof Sketch

$$\frac{\textsf{compile}(s) = t}{\{\text{safe to run } s\}\ t\ \{\text{state could result from running } s\}}$$

- Induction on **statement** $s$
- Strengthen the theorem with a **continuation** and an **invariant**:

$$\frac{\textsf{compile}(s, k) = t}{\{\textsf{inv}(s; k)\}\ t\ \{\textsf{inv}(k)\}}$$

inv(*s*): “safe to run *s*, and **when the current function returns**, that state could result from running *s*”

# Proof Sketch

$$\frac{\mathsf{compile}(s) = t}{\{\text{safe to run } s\}\ t\ \{\text{state could result from running } s\}}$$

- Induction on **statement** $s$
- Strengthen the theorem with a **continuation** and an **invariant**:

$$\frac{\mathsf{compile}(s, k) = t}{\{\mathsf{inv}(s; k)\}\ t\ \{\mathsf{inv}(k)\}}$$

Need a higher-order assertion logic to express this predicate

inv(*s*): "safe to run *s*, and **when the current function returns**, that state could result from running *s*"

# What's in the paper

- Formal operational semantics of Cito
- Compilation procedure
- Linking support by IL's program logic XCAP
- Detailed proof techniques
- Two optimization phases (const fold, dead-code elim) to demonstrate **vertical compositionality**
- Complete CountUnique example

***“no obvious deficiencies”***

***“obviously no deficiencies”***

-Tony Hoare

"… the formal guarantees of semantic preservation apply only to whole programs that have been compiled as a whole by CompCert C."

— http://compcert.inria.fr/compcert-C.html