Compiler Verification meets Cross-Language Linking via Data Abstraction

Peng Wang,
Santiago Cuellar,
Adam Chlipala,
MIT CSAIL

MIT CSAIL
Princeton University
Verified Compiler

Program Verification Techniques

Source Program Semantics
Verified Compiler

Program Verification Techniques

Source Program Semantics

Semantic Preserving Compiler

Target Program Semantics
Cross-Language Development

- iPhoto.swift
- matrix.c

- iPhoto.S
- matrix.S
- malloc.S

- iPhoto.exe
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

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

Cito

Bedrock IL
Cito Syntax

Notation:

Optional \([\cdot]\) List Of \((\cdot)^*\)
Product Type \(\times\) Sum Type \(+\)
Machine Word \(W\) String \(S\)

Syntax:

Constant \(w \in W\)
Label \(l \in S_{\text{module} \times S_{\text{fun}}}\)
Variable \(x \in S\)
Binary Op \(o ::= + \mid - \mid \times \mid = \mid \neq \mid < \mid \leq\)
Expression \(e ::= x \mid w \mid e \circ e\)
Statement \(s ::= \text{skip} \mid s; s \mid \text{if } e \{s\} \text{ else } \{s\} \mid \text{while } e \{s\} \mid x := \text{call } e\ (e^*) \mid x := e \mid x := \text{label } l\)

Function \(f \in S_{\text{arg}}^* \times S_{\text{ret}} \times S\)
Module \(m \in S_{\text{name}} \times (S_{\text{fname}} \times f)^*\)

State:

Machine State \((\Sigma) = E \times H\)
Variable Assignment \((\sigma) E = S \rightarrow W\)
Heap \((\mu) H = W \rightarrow |A|\)
ADT Domain \(A = \text{[parameter of theory]}\)
Bedrock IL Syntax

Syntax:

- Constants $c ::= \text{[fixed-width bitvectors]}$
- Code labels $\ell ::= \ldots$
- Registers $r ::= \text{Sp | Rp | Rv}$
- 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 $i ::= L \leftarrow R \mid L \leftarrow R \circ R$
- Jumps $j ::= \text{goto } R \mid \text{if } (R \circ R) \text{ then } \ell \text{ else } \ell$
- Blocks $B ::= \ell : \{\lambda \gamma. \phi\} \text{ i* ; j}$
- Modules $M ::= B^*$

State:

- Machine State $= \text{Memory} \times \text{Registers} \times \text{ProgramCounter}$
- Memory $= W \rightarrow W$
- Registers $= r \rightarrow W$
- ProgramCounter $= Pc \rightarrow W$
Bedrock IL Syntax

Syntax:

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

State:

Machine State = Memory \times Registers \times ProgramCounter
Memory = \( W \rightarrow W \)
Registers = \( r \rightarrow W \)
ProgramCounter = \( Pc \rightarrow W \)
• **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

\[ \Psi(\mathbb{e}_f) = \text{spec of } f \]

\[ \Psi \vdash (\Sigma, x := \text{call } e_f (e^*)) \Downarrow \Sigma'' \]

- Environment (\(\Psi\)) : Function address \(\rightarrow\) Function specification

- Function specification:
  - **Operational**: callee’s body
  - **Axiomatic**: relation of pre-call and post-call state
# Operational vs. Axiomatic

<table>
<thead>
<tr>
<th>Operational Specification</th>
<th>Axiomatic Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>☹ Language dependent</td>
<td>☺ Language independent</td>
</tr>
<tr>
<td>☺ No annotation burden</td>
<td>☹ Need to be provided</td>
</tr>
<tr>
<td>Suitable for <strong>intra</strong>-language calls</td>
<td>Suitable for <strong>inter</strong>-language calls</td>
</tr>
</tbody>
</table>
## Operational vs. Axiomatic

<table>
<thead>
<tr>
<th>Operational Specification</th>
<th>Axiomatic Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>😞 Language dependent</td>
<td>😊 Language independent</td>
</tr>
<tr>
<td>😇 No annotation burden</td>
<td>😞 Need to be provided</td>
</tr>
<tr>
<td>Suitable for <strong>intra</strong>-language calls</td>
<td>Suitable for <strong>inter</strong>-language calls</td>
</tr>
</tbody>
</table>

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

  \[
  \text{side-conditions} \quad \{\text{pre-cond}\} \text{ code } \{\text{post-cond}\}
  \]
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

\[
\text{side-conditions}\]
\[
\{\text{pre-cond}\} \text{ code} \{\text{post-cond}\}
\]
\[
\text{compile}(s) = t
\]
\[
\{\text{safe to run } s\} t \{\text{state could result from running } s\}
\]
\[ \text{compile}(s) = t \]

{safe to run \( s \)} \( t \) {state could result from running \( s \)}
\( \forall A, B, a, b, s, t. \) safe(A, s) \( \Rightarrow \)

\[
\text{compile}(s) = t
\]

\{safe to run s\} \( t \) \{state could result from running s\}

\[
\exists \text{ state } b
\]
∀ A,B,a,b,s,t. safe(A,s) ⇒

\[ \text{compile}(s) = t \]

\{safe to run \( s \)\} \( t \) \{state could result from running \( s \)\}

The main compiler correctness theorem
∀ A, B, a, b, s, t. safe(A, s) ⇒

\[ \text{compile}(s) = t \]
\{safe to run s\} t \{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;
}
Abstract Data Types (ADTs) are a natural interface between languages.

**CountUnique.hl:**

```java
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
    
    \[ A = FSET(P) + \cdots \quad P = \mathcal{P}(\mathbb{W}) \] (* sets of machine intears *)
  
  \[
  \{ \}
  \text{new}() \{ \text{return set } \emptyset \}
  
  \{ \lambda I. I = [] \} \text{ new } \{ \lambda (O, R). O = [] \land R = \text{ADT}(FSET(\emptyset)) \}
  
  \{ x \text{ is a set} \} \text{ delete}(x) \{ x \text{ is deleted} \}
  
  \{ \lambda I. I = [\text{ADT}(FSET(\cdot))] \} \text{ delete } \{ \lambda (O, R). O = [(\text{ADT}(FSET(\cdot)), \bot)] \land R = SCA(\cdot) \}
  
  \{ x \text{ is set } s \} \text{ size}(x) \{ x \text{ is still set } s, \text{return } |s| \}
  
  \{ \lambda I. I = [\text{ADT}(FSET(\cdot))] \} \text{ size } \{ \lambda (O, R). \exists s. O = [(\text{ADT}(FSET(s)), FSET(s))] \land R = SCA(|s|) \}
  
  \{ x \text{ is set } s \} \text{ add}(x, w) \{ x \text{ is set } s \cup \{w\} \}
  
  \{ \lambda I. I = [\text{ADT}(FSET(\cdot)), SCA(\cdot)] \} \text{ add } \{ \lambda (O, R). \exists s, w. O = [(\text{ADT}(FSET(s)), FSET(s \cup \{w\})), (SCA(w), \bot)] \land R = 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:

```ocaml
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:

```ocaml
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).
```

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

\[
\text{compile}(s) = t
\]
\[
\{\text{safe to run } s\} \ t \ \{\text{state could result from running } s\}
\]

- Induction on statement \textit{s}
- Strengthen the theorem with a \textit{continuation} and an \textit{invARIANT}:

\[
\text{compile}(s, k) = t
\]
\[
\{\text{inv}(s; k)\} \ t \ \{\text{inv}(k)\}
\]

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

\[
\text{compile}(s) = t
\]

\[
\{\text{safe to run } s\} \quad t \quad \{\text{state could result from running } s\}
\]

- Induction on **statement** \( s \)

- Strengthen the theorem with a **continuation** and an **invariant**:

  \[
  \text{compile}(s, k) = t
  \]

  \[
  \{\text{inv}(s; k)\} \quad t \quad \{\text{inv}(k)\}
  \]

  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