Type-Based Verification of Assembly Language for Compiler Debugging

Bor-Yuh Evan Chang  Adam Chlipala
George C. Necula  Robert R. Schneck
University of California, Berkeley
TLDI 2005
Long Beach, California
January 10, 2005

Problem and Motivation

- Starting Point: Bytecode verification
  - Effective, simple, and very verifiable
  - But does not work for assembly code
- Why assembly code?
  - Native code compilers are more complex
  - Also debug JITs
- Why existing compilers?
  - Force to make the verifier flexible
  - Better accessibility to students (i.e., require little or no annotations)
  - More test cases to validate “certified compilation for debugging”

Debugging Compilers with Verification

Challenges

class P {
    int f;
    int m() { ... }
}
class C extends P {
    int m() { ... }
}

branch (r0, 0) \text{Label}
rm := m[r0 + 8]
rm := m[rm + 12]
rm := \text{f}
rm := \text{null}
jump \text{Label}

\text{invokevirtual P.m()}

Challenges

class P {
    int f;
    int m() { ... }
}
class C extends P {
    int m() { ... }
}

branch (r0, 0) \text{Label}
r0 := m[r0 + 8]
r0 := m[r0 + 12]
r0 := \text{f}
r0 := \text{null}
jump \text{Label}

\text{invokevirtual P.m()}
Challenges

```java
class P {
    int f;
    int m() {...}
}

class C extends P {
    int m() {...}
    ...
    P p = new P();
    P c = new C();
    c.m();
    ...
    branch (= r_p 0) LAbort
    r_ops : m[f + 8] (r_p : nonnull P)
    r_ops := m[r_ops + 12] (r_p := P)
    r_ops := m[r_ops + 12] (r_ops : method P)
    r_p := &Lset
    jump [r_ops]
    (r_p := int)
```

Class P:
```java
branch (= r_p 0) LAbort
r_ops := m[f + 8] (r_p := nonnull P)
```

Class C:
```java
branch (= r_p 0) LAbort
r_ops := m[f + 8] (r_p := nonnull P)
```

Class P:
```java
branch (= r_p 0) LAbort
r_ops := m[f + 8] (r_p := nonnull P)
```

Class C:
```java
branch (= r_p 0) LAbort
r_ops := m[f + 8] (r_p := nonnull P)
```

Outline
- Overview
- Abstract State
  - Types
  - Join algorithm
- Verification Procedure
- Educational Experience
- Concluding Remarks

Overview
- To maximize accessibility (i.e., minimize annotations)
  - infer types at intermediate program points using abstract interpretation
  - willingness to have a limited amount of specialization to a compilation strategy
- To handle assembly code
  - use (simple) dependent types
- To be less sensitive to the compilation strategy
  - assign types to intermediate values lazily
Why Symbolic Values?

- Used to track dependencies between registers

\[ r_1 = r_2 \land r_1 = r_3 + 4 \]

\[ r_1 = \alpha + 4, r_2 = \alpha + 4, r_3 = \alpha \]

- Value state form convenient for abstract interpretation

\[ r_1 := r_2 \]

\[ \Sigma \]

\[ \Sigma[r_1 \rightarrow \Sigma(r_2)] \]

Values

\[ \text{values } v := n \odot L + n_1 \cdot n_2 \mid \alpha = n \mid \alpha \neq n \mid \alpha < n \mid \cdots \]

- Define a normalization of expressions to values

\[ \Gamma \vdash e \Downarrow \Gamma' \]

- Values give the expressions of interest

  - For our case, address computation and comparisons with constants
  - Would vary depending on the source language or compiler of interest
  - Not all equal expressions normalize to the same value, so value forms must be chosen carefully

- Registers are equal if they map to the same values (after normalization)

Types

- Primitive Types and Object References

\[ \text{types } \tau ::= T \mid \text{word} \mid \bot \mid \text{nonnull} \mid \tau \mid \text{ptr} \mid \ldots \]

\[ \text{bounds } b ::= C \mid \text{exactly } C \mid \text{bounded above by } C \mid \text{bounded above and below by } C \mid \text{same class as } \alpha \]

- Types for Run-time Structures

\[ \text{types } \tau ::= \ldots \]

\[ \text{dispatch of } \alpha \]

\[ \text{dispatch of } \alpha \]

\[ \text{method}(\alpha, n) \]

\[ \ldots \]

Join

\[ (r_1 = \alpha + 4, r_2 = \beta + 4, r_3 = \alpha) \]

\[ (r_1 = \gamma + 4, r_2 = \delta + 8, r_3 = \gamma) \]

\[ \alpha : \text{disp}(\beta), \beta : \text{nonnull exactly } C \]

\[ \gamma : \text{disp}(\delta), \delta : \text{nonnull } C \]

\[ (\alpha, \gamma) \rightarrow \varepsilon \]

\[ (\beta, \delta) \rightarrow \zeta \]

Outline

- Overview
  - Abstract State
    - Types
    - Join algorithm
  - Verification Procedure
  - Educational Experience
  - Concluding Remarks
Verification Example

class P {
  int f;
  int m() { ... }
}
class C extends P {
  ... 
}

Typing Rule

Γ ⊢ e : α + n → Γ'
Γ ⊢ e : α : nonnull C > Γ''
Γ ⊢ e : printf(e)

r_exp := r_p + 12
r_{m[r_exp]} := r_f
r_a := &m
jump [r_exp]
L_out;

Comparing Compiler Development

Without Coidlaid With Coidlaid

Type errors decrease: 44% to 19%  
- even 2% to 15% if only consider those that are visible in testing

Outline

- Overview
- Abstract State
  - Types
  - Join algorithm
- Verification Procedure
- Educational Experience
- Concluding Remarks

Comparing Compiler Development

Without Coidlaid With Coidlaid

• Good compilations increase: 53% to 75%

Comparing Compiler Development

Without Coidlaid With Coidlaid

• False alarms: 6% to 1%  
  - price for ease of use
**Student Perspective**

- Traditional testing procedure
  - 49 test cases / inputs
- Mean scores:
  - 33 (67%) in 2002
  - 34 (68%) in 2003
  - 39 (79%) in 2004

- “counter-productive”
- “can’t imagine being without it”

A favorite comment:
“**I would be totally lost without Coolaid. I learn best when I am using it hands-on... I was able to understand the exact functions, conventions, and optimizations and to appreciate them.**”

---

**Conclusion**

- Extended data-flow based type inference of intermediate languages to assembly code
  - Able to retrofit existing compilers
  - Minimal annotations for accessibility
- Provided experimental confirmation that certifying compilation helps early compiler debugging
- Introduced undergraduates to the idea of improving software quality with program analysis