Type-Based Verification of Assembly Language for Compiler Debugging

Bor-Yuh Evan Chang
George C. Necula

Adam Chlipala
Robert R. Schneck

University of California, Berkeley

TLDI 2005
Long Beach, California
January 10, 2005

Problem and Motivation

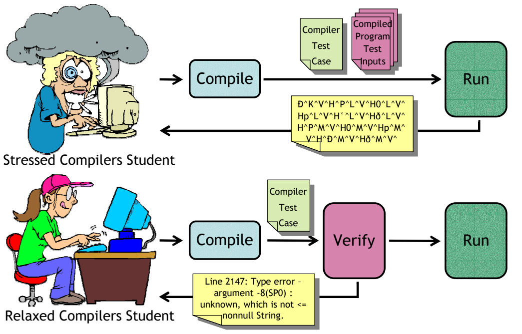
- Type check **assembly code** produced by **existing** compilers for a Java-like language
- Validate that certifying compilation aids compiler debugging
  - using undergraduate compilers course as a test bed
- Introduce undergraduates to using compiler technology for software quality

---

Problem and Motivation

- Starting Point: Bytecode verification
  - Effective, simple, and very accessible
  - 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() { … }
}

…
P p = new P();
P c = new C();
c.m();
…
```

invokevirtual P.m()

branch (= r_c 0) L_{abort}
r_{tmp} := m[r_c + 8]
r_{tmp} := m[r_{tmp} + 12]
r_{arg0} := r_c
r_{ra} := &L_{ret}
jump [r_{tmp}]
L_{ret}:

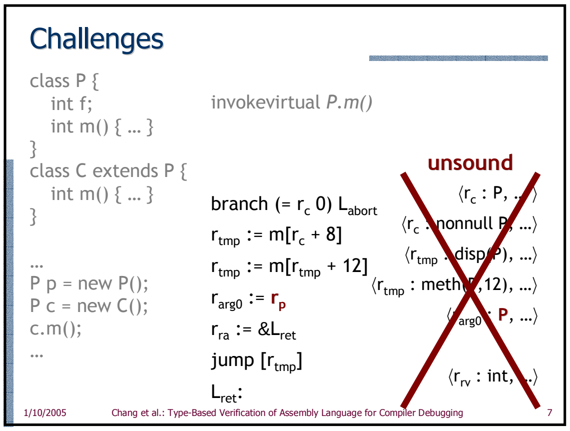
Challenges

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

…
P p = new P();
P c = new C();
c.m();
…
```

invokevirtual *P.m()*

⟨r_c : P, … ⟩
branch (= r_c 0) L_{abort}
⟨r_c : nonnull P, …⟩
r_{tmp} := m[r_c + 8]
⟨r_{tmp} : disp(P), …⟩
r_{tmp} := m[r_{tmp} + 12]
⟨r_{tmp} : meth(P,12), …⟩
r_{arg0} := r_c
⟨r_{arg0} : P, …⟩
r_{ra} := &L_{ret}
jump [r_{tmp}]
L_{ret}:
⟨r_{rv} : int, …⟩



Challenges

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

…
P p = new P();
P c = new C();
c.m();
…
```

invokevirtual *P.m()*

unsound

branch (= r_c 0) L_{abort} ⟨r_c : P, …⟩

r_{tmp} := m[r_c + 8] ⟨r_c : nonnull P, …⟩

r_{tmp} := m[r_{tmp} + 12] ⟨r_{tmp} : disp(P), …⟩

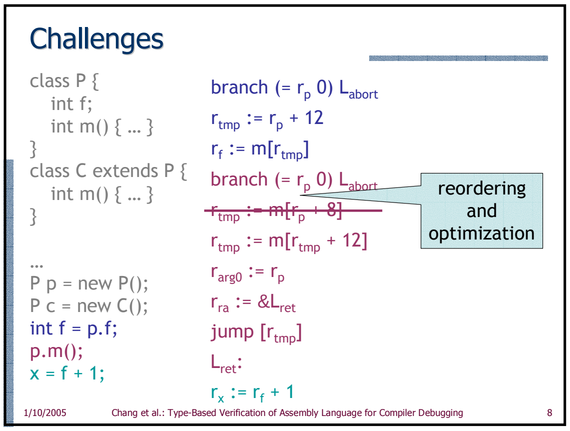
⟨r_{tmp} : meth(P,12), …⟩

r_{arg0} := **r_p** ⟨r_{arg0} : **P**, …⟩

r_{ra} := &L_{ret}

jump [r_{tmp}]

L_{ret}: ⟨r_{rv} : int, …⟩



Challenges

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

…
P p = new P();
P c = new C();
int f = p.f;
p.m();
x = f + 1;
```

branch (= r_p 0) L_{abort}

r_{tmp} := r_p + 12

r_f := m[r_{tmp}]

branch (= r_p 0) L_{abort}

~~r_{tmp} := m[r_p + 8]~~

r_{tmp} := m[r_{tmp} + 12]

r_{arg0} := r_p

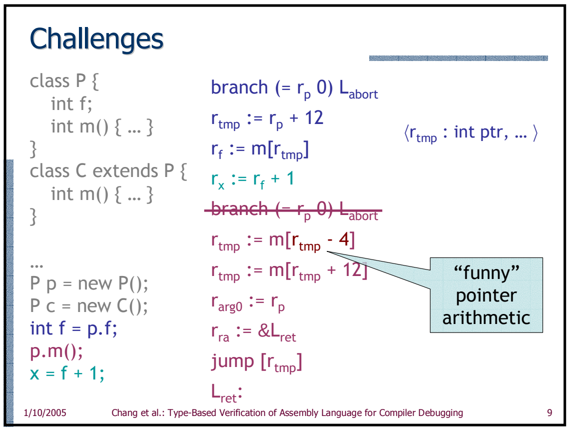
r_{ra} := &L_{ret}

jump [r_{tmp}]

L_{ret}:

r_x := r_f + 1

reordering
and
optimization



Challenges

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

…
P p = new P();
P c = new C();
int f = p.f;
p.m();
x = f + 1;
```

branch (= r_p 0) L_{abort}

r_{tmp} := r_p + 12 ⟨r_{tmp} : int ptr, … ⟩

r_f := m[r_{tmp}]

r_x := r_f + 1

~~branch (= r_p 0) L_{abort}~~

r_{tmp} := m[r_{tmp} - 4]

r_{tmp} := m[r_{tmp} + 12]

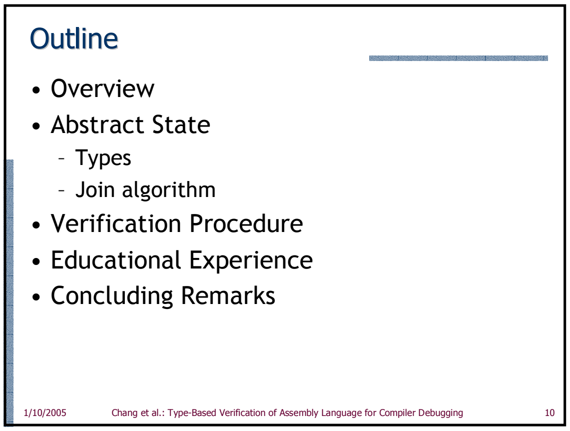
r_{arg0} := r_p

r_{ra} := &L_{ret}

jump [r_{tmp}]

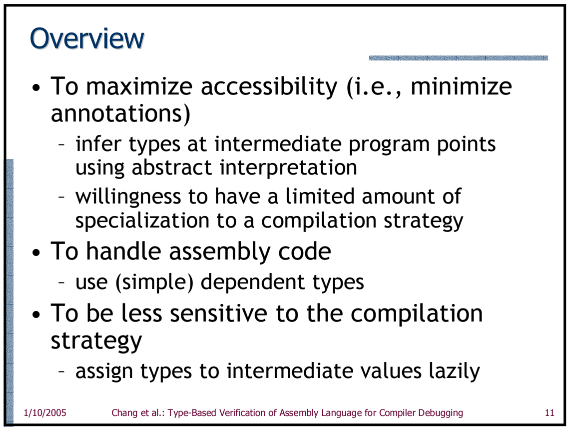
L_{ret}:

"funny"
pointer
arithmetic



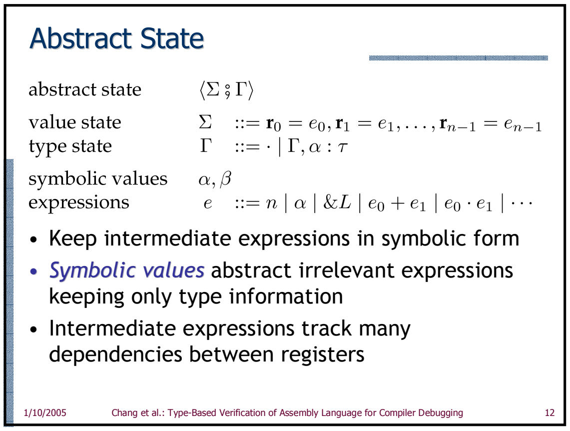
Outline

- 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



Abstract State

abstract state	$\langle \Sigma \fatsemi \Gamma \rangle$	
value state	Σ	$::= \mathbf{r}_0 = e_0, \mathbf{r}_1 = e_1, \ldots, \mathbf{r}_{n-1} = e_{n-1}$
type state	Γ	$::= \cdot \mid \Gamma, \alpha : \tau$
symbolic values	α, β	
expressions	e	$::= n \mid \alpha \mid \&L \mid e_0 + e_1 \mid e_0 \cdot e_1 \mid \cdots$

- Keep intermediate expressions in symbolic form
- *Symbolic values* abstract irrelevant expressions keeping only type information
- Intermediate expressions track many dependencies between registers

Why Symbolic Values?

- Used to track dependencies between registers

$$r_1 = r_2 \ \wedge \ r_1 = r_3 + 4$$

$$\Big\downarrow$$

$$r_1 = \alpha + 4,\ r_2 = \alpha + 4,\ r_3 = \alpha$$

- Value state form convenient for abstract interpretation

$$\Sigma$$

$$r_1 := r_2$$

$$\Sigma[r_1 \to \Sigma(r_2)]$$

Values

values $v ::= n_0 \cdot \&L + n_1 \cdot \alpha + n_2 \mid \alpha = n \mid \alpha \neq n \mid \alpha < n \mid \cdots$

- Define a normalization of expressions to values

$$\Gamma \vdash e \Downarrow v \triangleright \Gamma'$$

- 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

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

bounds $b ::= C$ — bounded above by C

$\mid$ exactly C — bounded above and below by C

$\mid$ classof(α) — same class as α

- Types for Run-time Structures

types $\tau ::= \ldots$

$\mid$ disp(α) — dispatch table of α

$\mid$ meth(α, n) — t n

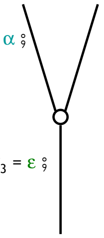
$\mid \ldots$

Tracks that the value is the dispatch table of object α

Join

$\langle r_1 = \alpha+4,\ r_2 = \beta+4,\ r_3 = \alpha \fatsemi$
$\alpha : \text{disp}(\beta),$
$\beta : \text{nonnull exactly C}\rangle$

$\langle r_1 = \gamma+4,\ r_2 = \delta+8,\ r_3 = \gamma \fatsemi$
$\gamma : \text{disp}(\delta),$
$\delta : \text{nonnull C}\rangle$



$\langle r_1 = \varepsilon+4,\ r_2 = \eta,\ r_3 = \varepsilon \fatsemi$
$\varepsilon : \text{disp}(\zeta),$
$\zeta : \text{nonnull C},$
$\eta : \top\rangle$

$(\alpha,\gamma) \to \varepsilon$

$(\beta,\delta) \to \zeta$

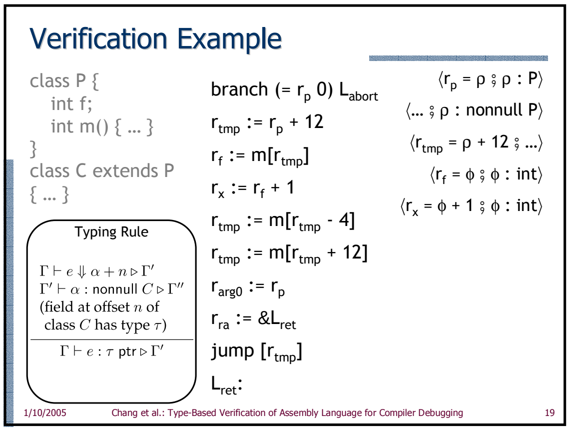
Join

- Joining of type state basically given by subtyping
  - subtyping lattice still fairly simple – dictated mostly by the class inheritance hierachy
- Values are (fancy) labels for equivalence classes of registers
  - lattice of conjunctions of equalities on registers
  - first normalize expressions to values for join
  - each distinct pair of values in the input state gives an equivalence class in the result state

---

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
{ … }
```

branch (= r_p 0) L_{abort} — $\langle r_p = \rho \fatsemi \rho : P\rangle$

r_{tmp} := r_p + 12 — $\langle \ldots \fatsemi \rho : \text{nonnull } P\rangle$

r_f := m[r_{tmp}] — $\langle r_{tmp} = \rho + 12 \fatsemi \ldots\rangle$

r_x := r_f + 1 — $\langle r_f = \phi \fatsemi \phi : \text{int}\rangle$

r_{tmp} := m[r_{tmp} - 4] — $\langle r_x = \phi + 1 \fatsemi \phi : \text{int}\rangle$

r_{tmp} := m[r_{tmp} + 12]

r_{arg0} := r_p

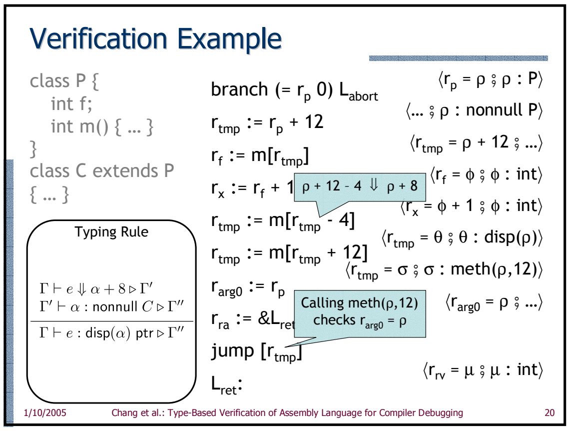
r_{ra} := &L_{ret}

jump [r_{tmp}]

L_{ret}:

Typing Rule

$$\frac{\Gamma \vdash e \Downarrow \alpha + n \triangleright \Gamma' \quad \Gamma' \vdash \alpha : \text{nonnull } C \triangleright \Gamma'' \quad \text{(field at offset } n \text{ of class } C \text{ has type } \tau)}{\Gamma \vdash e : \tau \text{ ptr} \triangleright \Gamma'}$$



Verification Example

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

branch (= r_p 0) L_{abort} — $\langle r_p = \rho \fatsemi \rho : P\rangle$

r_{tmp} := r_p + 12 — $\langle \ldots \fatsemi \rho : \text{nonnull } P\rangle$

r_f := m[r_{tmp}] — $\langle r_{tmp} = \rho + 12 \fatsemi \ldots\rangle$

r_x := r_f + 1 — $\langle r_f = \phi \fatsemi \phi : \text{int}\rangle$

r_{tmp} := m[r_{tmp} - 4] — $\langle r_x = \phi + 1 \fatsemi \phi : \text{int}\rangle$

r_{tmp} := m[r_{tmp} + 12] — $\langle r_{tmp} = \theta \fatsemi \theta : \text{disp}(\rho)\rangle$

r_{arg0} := r_p — $\langle r_{tmp} = \sigma \fatsemi \sigma : \text{meth}(\rho,12)\rangle$

r_{ra} := &L_{ret} — $\langle r_{arg0} = \rho \fatsemi \ldots\rangle$

jump [r_{tmp}]

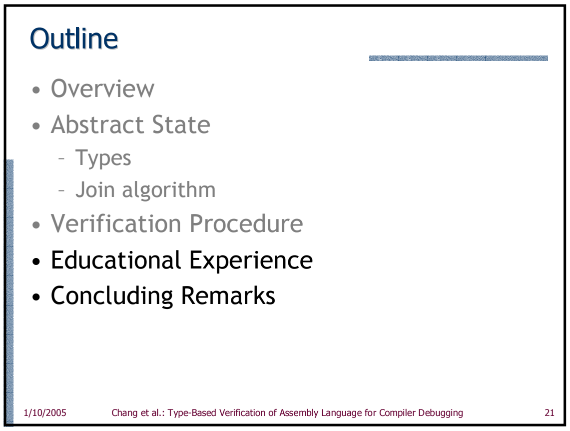
L_{ret}: — $\langle r_{rv} = \mu \fatsemi \mu : \text{int}\rangle$

$\rho + 12 - 4 \Downarrow \rho + 8$

Calling meth(ρ,12) checks $r_{arg0} = \rho$

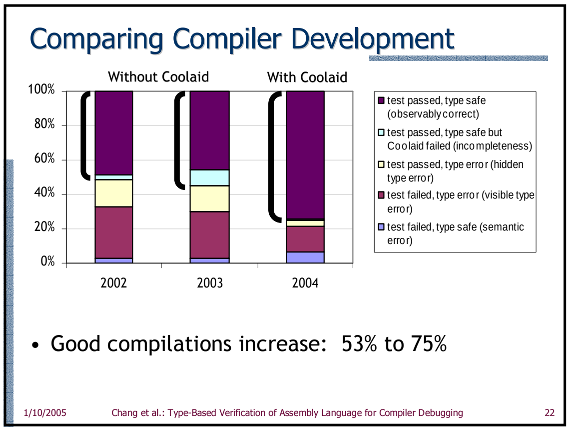
Typing Rule

$$\frac{\Gamma \vdash e \Downarrow \alpha + 8 \triangleright \Gamma' \quad \Gamma' \vdash \alpha : \text{nonnull } C \triangleright \Gamma''}{\Gamma \vdash e : \text{disp}(\alpha) \text{ ptr} \triangleright \Gamma''}$$



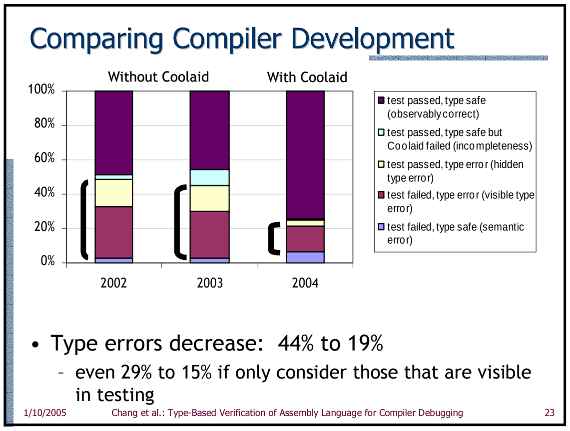
Outline

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



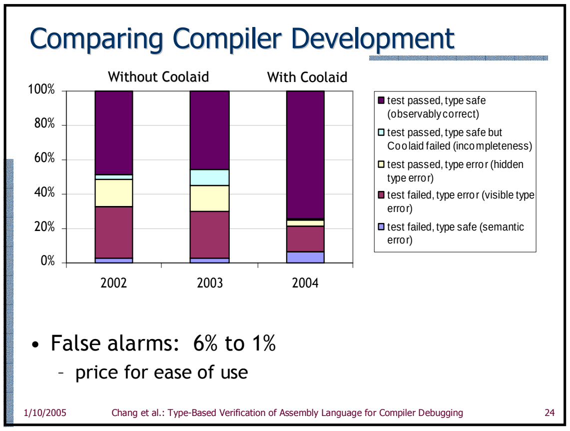
Comparing Compiler Development

- Good compilations increase: 53% to 75%



Comparing Compiler Development

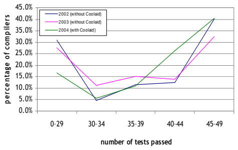
- Type errors decrease: 44% to 19%
  - even 29% to 15% if only consider those that are visible in testing

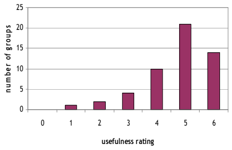


Comparing Compiler Development

- 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 (69%) in 2003
  - 39 (79%) in 2004

- 0: “counter-productive”
  6: “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 really understand stack 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