Towards Path-sensitive Points-to Analysis

Tobias Gutzmann, Jonas Lundberg, and Welf Löwe

Växjö University, Sweden

September 30, 2007

Motivation

- Points-to analysis: (Static) dataflow analysis
  - Which objects can variable v possibly reference during program execution?
  - Compute the *points-to set* $pt(v)$ = set of abstract objects v may reference
  - Abstraction: Map possible *runtime objects* $\rightarrow$ *abstract objects*
- One variation point that affects precision and execution time is *Path-sensitivity*:
  - Run analysis for different execution paths separately
  - Expensive (possibly very large number of possible paths)
- Idea: Obtain additional dataflow information for different paths from control flow statements

Our Approach

Based on control-flow conditions, introduce *filter operations* for different branches

Java Example

```
// A, B types in no subtype-relation
// assume pt(v) = {o1 : A, o2 : B}
if (v instanceof A) {
    // v unfiltered
    // pt(v) = {o1 : A, o2 : B}
    ...
} else {
    // v unfiltered
    // pt(v) = {o1 : A, o2 : B}
    ...
}
```

Our Approach

Based on control-flow conditions, introduce *filter operations* for different branches

Java Example

// A, B types in no subtype-relation
// assume $pt(v) = \{o_1 : A, o_2 : B\}$
if (v instanceof A) {
    v = $\{ o \in pt(v) \mid o \text{ instanceof } A\}$
    // $pt(v) = \{o_1 : A, o_2 : B\}$
    ...
} else {
    v = $\{ o \in pt(v) \mid \neg(o \text{ instanceof } A)\}$
    // $pt(v) = \{o_1 : A, o_2 : B\}$
    ...
}

Our Approach

Based on control-flow conditions, introduce *filter operations* for different branches

Java Example

// A, B types in no subtype-relation
// assume $pt(v) = \{o_1 : A, o_2 : B\}$
if (v instanceof A) {
    v = $\{ o \in pt(v) \mid o \text{ instanceof } A\}$
    // $pt(v) = \{o_1 : A, \sout{o_2 : B}\}$
    ...
} else {
    v = $\{ o \in pt(v) \mid \neg(o \text{ instanceof } A)\}$
    // $pt(v) = \{\sout{o_1 : A}, o_2 : B\}$
    ...
}

Filter Operations for Java

Instance-Of Filter

- Condition: v instanceof T
- *true* branch: $pt(v) = \{o \in pt(v) \mid o \text{ instanceof } T\}$
- *false* branch: $pt(v) = \{o \in pt(v) \mid \neg(o \text{ instanceof } T)\}$

Further Filter Operations

- object equality
- *null*-checks
- $x.equals(y)$
- $x.getClass() == T.class$

Note: If filter operations were executed during an actual program run, they would never change the program state

Evaluation

Safe Casts: % of cast operations in a program that never fail
Dead Code: % of accesses not reached by any dataflow value
Safe Accesses: % of accesses never raising a NullPointerException
Time: Time factor t_{sens}/t_{insens}

	Safe Casts		Dead Code		Safe Accesses		Time
Program	Insens	Sens	Insens	Sens	Insens	Sens	
AntLR	43.1	49.0	19.3	19.5	84.1	85.3	1.12
JavaCC	10.4	33.9	2.5	2.7	68.6	70.9	1.06
Jython	64.6	77.6	28.3	29.5	90.1	93.0	0.91
...							
Average	34.5	45.4	15.0	15.7	81.0	83.7	1.09

- *Safe Casts* metric shows substantial improvements
- *Safe Accesses* shows improvements on an already high level
- A little more *Dead Code* is discovered
- Computation time is affected, but not in a harmful way

Conclusion

- Introduce filter operations to achieve Path-sensitivity
- We do not compute more paths than in a path-insensitive analysis
- Analysis metrics show improvements
- On average, only a slight performance slowdown is observed
- We target Java; however, the concept can also be adapted to other programming languages