자바 언어를 위한 정적 분석
(Static Analyses for Java)

'99 한국정보과학회 가을학술발표회 튜토리얼

1999. 10. 23

창 병 모
숙명여대 전산학과
http://cs.sookmyung.ac.kr/~chang

목차

- Java Overview
- Why static analyses ?
- Static analyses
  - Class analysis
  - Exception analysis
  - Escape analysis
- Conclusions

Java overview

- Object oriented
  - classes (single inheritance)
  - object types (interfaces)
  - objects
- C-like syntax
- Static typing
  - type soundness by Eisenbach in ECOOP'97
  - by Nipkow in ACM POPL'98

Java overview

- Dynamic binding
  - inheritance and overriding
  - class analysis (object type inference)
- Exception handling
  - uncaught exception analysis
- Automatic memory management
  - escape analysis
- Concurrency

Why Static Analyses for Java

- Class analysis
  - for call graph and fast method dispatch in compiler
- Exception analysis
  - for verification and programming environments
- Escape analysis
  - for efficient memory management and synchronization optimization

Higher-order CFA and Class Analyses

Higher-order Flow Analysis

- Call graphs
  - Many program analyses rely on a call-graph.
  - There is an edge (f,g) if function f calls function g.
  - Call graphs are easy to compute in FORTRAN.
- Not so easy in higher-order languages
  - functional (ML)
  - object-oriented (Java, C++)
  - pointer-based (C)

Higher-order Flow Analysis

- In a functional language

  $e_1\ e_2$ closure analysis [Shivers88]
- In an object oriented language

  e.m() class analysis
- In a pointer language

  (*p)() pointer analysis
- In each case it is unclear which function is called.

Class Analyses for Java

- The goal is
  - to approximate the classes of the objects, to which an expression refer
- Also gives an approximation of the call graph

Class Analyses for Java

- Domain
  - Sets of class names.
- A set variable [e] for each expression e.
- Set up set-constraints of the form :

$$[e] \supseteq se$$

- Analysis assigns possible classes of e to [e].
  - Solution of the constraints yield the information

Sample Constraints for Class Analyses

Suppose e is each of the following expressions

- new C | $[e] \supseteq \{C\}$
- if e_0 then e_1 else e_2 | $[e] \supseteq [e_1] \cup [e_2]$
- id = e_1 | $[id] \supseteq [e_1]$
- Method application $e_0.m(e_1)$

  for each class C in $[e_0]$ with a method $m(x_1)$ = return e_m

  $C \in [e_0] \Rightarrow ([x_1] \supseteq [e_1])$

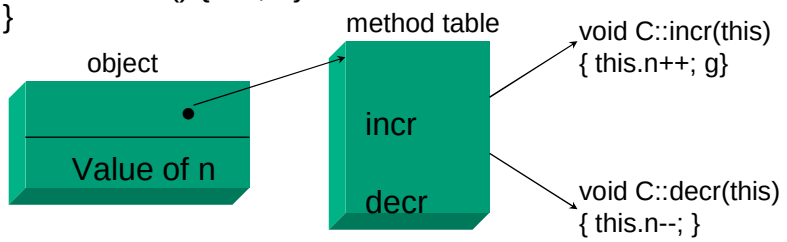
  $[e] \supseteq [e_m]$

History in Class Analyses

- Constraint resolution in $O(n^3)$ time.
- This analysis was discovered
  - by Palsberg and Schwartzbach in 1991.
  - closely related to closure analysis for functional programs (Jones,Shivers).
- Fast interprocedural class analysis
  - by node(set variable) merging
  - Grove and Chambers in ACM POPL'98

Objects and Methods in Java

```
class C {
    int n;
    void incr()  { n++; }
    void decr() { n--;  }
}
```



- Method invocation:

  [[e.m(arg)]] =

  object * o = [[e]]; lookup(o→vtable, m)(o, [[arg]])

Objects and Methods in Java

- Layout of method tables attached to objects
  - based on inheritance hierarchies
- Transformation of method invocations into

  method lookups + calls.
- We can generate a direct call c.m using analysis
  - if that set is a singleton {c} or
  - if all elements in that set have the same implementation of method *m*

Uncaught Exception Analysis

Exceptions in Java

- Every exception is declared as
  - a subclass of “Exception” class
- Throw exceptions

  throw e
- Exception handling

  try { … } catch (E x) { … }
- Specify uncaught exceptions in method definition

  m(...) throws … { …}

Uncaught Exception Analysis in JDK

- Intraprocedural analysis
  - Based on programmer's specifications.
- Not elaborate enough to
  - suggest for specialized handling nor
  - remove unnecessary handlers

Uncaught Exception Analysis

- We need an interprocedural analysis to
  - estimate Java program's exception flows
  - independently of the programmer's specs.
- Approximate all possible uncaught exceptions
  - for every expression and every method
- Exception analysis after class analysis

Deriving Set Constraints

- Domain
  - Sets of exception class names.
- A set variable P_e for every expression e
  - Deriving set constraints of the form :

$$P_e \supseteq se$$

- Analysis assigns classes of possible uncaught exceptions of e to P_e

Deriving Set Constraints

Suppose e is each of the following expressions

- id = e_1 | $P_e \supseteq P_{e1}$
- if e_0 then e_1 else e_2 | $P_e \supseteq P_{e0} \cup P_{e1} \cup P_{e2}$
- throw e_1 | $P_e \supseteq [e_1] \cup P_{e1}$
- try e_0 catch (c_1 x_1) e_1 | $P_e \supseteq (P_{e0} - \{c_1\}^*) \cup P_{e1}$
- Method invocation $e_0.m(e_1)$

  - $P_e \supseteq P_{e0} \cup P_{e1}$

  - for each class C in $[e_0]$ with a method m(x1) = e_m

$$C \in [e_0] \Rightarrow P_e \supseteq P_{em}$$

Method-level Exception Analysis

- Cost-Effective?
  - Too Many Set Variables
- Observations
  - exceptions are sparse objects
  - exceptions are usually explicit
  - methods are usually explicit

Set Variables for Method-level Analysis

- P_f for each method f
  - class names of uncaught exceptions during the call to f
- P_g for try expressions e_g in

  try e_g catch (c_1 x_1) e_1

- Assume that [e] represents
  - classes that are ``available'' at an expression e

Method-level Set Constraints

Suppose each expression is in a method f

- id = e_1 |
- if e_0 then e_1 else e_2 |
- throw e_1 | $P_f \supseteq [e_1] \cap ExnClasses$
- try e_g catch (c_1 x_1) e_1 | $P_f \supseteq P_g - \{c_1\}^*$
- Method invocation $e_0.m(e_1)$

  - for each class C in $[e_0]$ with a method $m(x_1) = e_m$

$$C \in [e_0] \Rightarrow P_f \supseteq P_{c.m}$$

Exception Analyses for Java

- Exception analysis for Java
  - by Yi and Chang in ECOOP'99 Workshop
  - Expression-level and Method-level
- We are currently devising
  - a general framework for method-level analysis
- Jex
  - A tool for a view of the exception flow
  - by Robillard and Murphy in 1999

Applications of Exception Analysis

- A kind of program verification
  - Provide programmers information on all possible uncaught exceptions
- Can be incorporated in Java programming environment

Escape Analysis

Escape Analysis

- Escape analysis is basically
  - lifetime analysis of objects
- An object escapes a method if it is
  - passed as a parameter
  - returned
- Basic idea of applications:
  - Basically all objects are allocated in a heap.
  - If an object doesn't escape a method(or region), it can be allocated on stack

Escape Graph for Escape Analysis

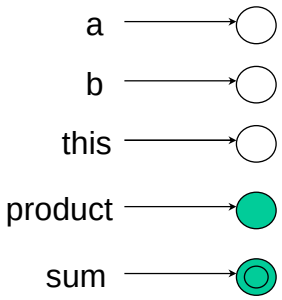
- inside node
  - object created inside the currently analyzed region and accessed via inside edges.
- outside node
  - object created outside the currently analyzed region or accessed via outside edges.
- inside edge
  - references created inside the current region
- outside edge
  - references created outside the current region

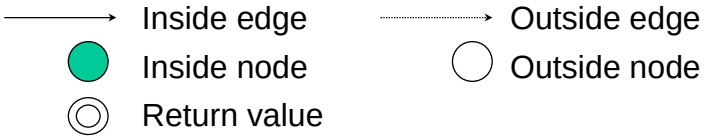
Example

```
Class complex {
    double x, y;
    complex (double a, double b) { x = a; y = b;}
    complex multiply(complex a) {
       complex product = new complex(x*a.x - y*a.y, x*a.y+y*a.x);
       return product;
    }
    complex add(complex a) {
        complex sum = new complex(x+a.x,y+a.y);
        return sum;
    }
    complex multiplyAdd(complex a, complex b) {
        complex product = a.multiply(b);
        complex sum = this.add(product);
        return sum;
    }
}
```


Example

Analysis Result for mutiplyAdd





Escape Analysis

- Intraprocedural analysis
  - Construction of escape graph following the control-flow
- Interprocedural analysis
  - For every method invocation cite, mapping between caller and callee.
  - To simulate parameter passing and returning

Escape Analysis

- OOPSLA'99
  - Compositional Pointer and Escape Analysis for Java Programs by J. Whaley and M. Rinard
  - Escape analysis for object-oriented languages: application to Java by B Blanchet
  - Escape analysis for Java by J.D. Choi et al.

Conclusions

- We surveyed three major analyses for Java
  - class analysis, exception analysis, escape analysis
- Further research topics
  - generalize method-level analysis
  - static analysis in connection with verification
  - analyses of Java bytecode