Call Graph Construction in Object-Oriented Languages

D. Grove, G. DeFouw, J. Dean, G. Chambers
University of Washington
OOPSLA'97

Overview

- Motivation
- Theoretical framework
- Experimental results
- Conclusions

Call Graph Construction

- Optimizing compilers
  - Interprocedural analyses
  - Need call graph
  - Difficulty: dynamic dispatch
- Solution: construct the call graph simultaneously with interprocedural class analysis
  - Class analysis: set of classes for each variable
- Subsequent interprocedural analyses

Context-sensitivity

- Context-insensitive call graph
  - One node per procedure
  - Set of call sites per node
  - Set of callees per call site
- Context-sensitive call graph
  - Different copies (*contours*) for different calling contexts
  - One dimension of context-sensitivity

Other Dimensions

- Instance variable contours
  - One or more contours per source-level instance variable declaration
  - Most analyses are context-insensitive
  - Example: separate contours for each subclass
- Class contours
  - One or more contours per source-level class
  - Different instantiation sites create different contours
  - Compute a set of class contours for each variable

Call Graph Domain

ClassContour = 2Tuple(Class,ClassKey)

ClassContourSet = Pow(ClassContour)

InstVarContour = 3Tuple(InstVariable,InstVarKey,
ClassContourSet)

InstVarContourSet = Pow(InstVarContour)

Call Graph Domain (cont)

ProcContour = 7Tuple(Procedure,ProcKey,
ProcContour,
Map(Variable,ClassContourSet),
Map(CallSite,ProcContourSet),
Map(LoadSite,InstVarContourSet),
Map(StoreSite,InstVarContourSet))

ProcContourSet = Pow(ProcContour)

CallGraph = 2Tuple(ProcContourSet,InstVarContourSet)

Examples of Algorithms

- Context-insensitive (0-CFA)
- Based on the dynamic call chain
  - *k* enclosing calling contours (call sites?): *k*-CFA
  - *l* contours around class instantiation site: *k*-*l*-CFA
  - Examples: *k=1* and *l=1*
- Based on the classes of the actuals
  - Cartesian Product Algorithm
  - Simple Class Sets
  - Bounded versions

Less precise algorithms

- $G_{selector}$: incompatible names/number of arguments
- G_{static}: CHA for statically-typed languages
- G_{intra}: flow-sensitive intraprocedural analysis
- G_{RTA}: Rapid Type Analysis
- G_{unif}: unification-based, Steensgaard-like analysis

Impact on performance

- Base version: intraprocedural analysis and optimizations
- Optimized version: additional interprocedural analyses improve intraprocedural optimizations
  - Most important: interprocedural class analysis for better devirtualization
  - Treeshaking: removal of unreachable methods
- Disappointing speedups for Java

Conclusions

- General framework for a family of analyses
- Several dimensions in the design space
- Are there benefits from context-sensitivity?
  - Did not use procedure specialization
- Scalability may become a problem for flow-sensitive analyses
- Is it worth it?