Interprocedural Analysis with Data-Dependent Calls

Previously, assumed call graph prior to interprocedural analysis

But in languages with function pointers, first-class functions, or dynamically dispatched messages, callee(s) at call site depend on (interprocedural) data flow

How to break the cycle?

Could make worst-case assumptions:
  call all possible functions/methods...
  • ... with matching name (if name is given at call site)
  • ... with matching type (if type is given & trustable)
  • ... that have had their addresses taken, & escape (if known)

Could do analysis to compute possible callees/receiver classes

Interprocedural analysis of sets of callee functions

Set up a standard optimistic interprocedural analysis, use iteration to relax initial optimistic solution into a sound fixed-point solution [e.g., for function ptrs/values]

A simple context-insensitive analysis:
  • for each (formal, local, result, global, instance) variable, maintain set of possible functions that could be there
  • initially: empty set for all variables
  • for each call site, set of callees derived from set associated with applied function expression
  • initially: no callees

\texttt{worklist := (main)}
\texttt{while worklist not empty}
\texttt{remove p from worklist}
\texttt{process p:}
  • perform intra analysis propagating fn sets from formals
  • foreach call site \texttt{s} in \texttt{p:}
    • add call edges for any new reachable callees
    • add \texttt{fns} of actuals to callees' \texttt{formals}
    • if new callees(s) reached or callees(s)' formals changed, put callees(s) back on worklist
    • if result changed, put caller(s) back on worklist

Example

\begin{verbatim}
proc main() {
  proc p(pa) { return pa(d); }
  return b(p);
}

proc b(ba) {
  proc q(qa) { return d(d); }
  c(q);
  return ba(c(d));
}

proc c(ca) {
  return ca(ca);
}

proc d(da) {
  proc r(ra) { return da; }
  return c(r);
}
\end{verbatim}

Context-sensitive analyses

Can get more precision through context-sensitive interprocedural analysis

\texttt{k-CFA (control flow analysis) [Shivers 88 etc.]}
  • analyze Scheme programs
  • context key: sequence of \texttt{k} enclosing call sites
  • \texttt{k=0} \Rightarrow context-insensitive
  • \texttt{k=1} \Rightarrow reanalyze for each call site (but not transitively)
    – loses precision beyond \texttt{k} recursive calls
    – cost is exponential in \texttt{k}, even if no gain in precision

An alternative:
  • context key: set of possible functions for arguments
  • avoid weaknesses of \texttt{k-CFA}:
    • only expend effort if possibly beneficial
    • never hits an arbitrary cut-off
    • worst-case cost proportional to \texttt{(functions)} \cdot \texttt{MaxNumberOfArgs}
**Static analysis of OO programs**

Problem: dynamically dispatched message sends
- direct cost: extra run-time checking to select target method
- indirect cost: hard to inline, construct call graph, do interprocedural analysis

Smaller problem: run-time class/subclass tests
(e.g. instanceof, checked casts)
- direct cost: extra tests

**Class analysis**

Solution to both problems: **static class analysis**
- compute set of possible classes of objects computed by each expression

Knowing set of possible classes of message receivers enables message lookup at compile-time
**static binding, devirtualization**

Benefits of knowing set of possible target methods:
- can construct call graph & do interprocedural analysis
- if single callee, then can inline, if profitable
- if small number of callees, then can insert type-case

Knowing classes of arguments to run-time class/subclass tests enables constant-folding of tests, plus cast checking tools

Many different algorithms for performing class analysis
- different trade-offs between precision and cost

**Intraprocedural class analysis**

Propagate sets of bindings of variables to sets of classes through CFG
(e.g. \(x \rightarrow \text{Int}, y \rightarrow \{	ext{Vector, String}\}\))

Flow functions:
- \(CA_X \leftarrow \text{new } c_1 \rightarrow \text{in} - (X \rightarrow \ast) \cup (X \rightarrow c_1)\)
- \(CA_X \leftarrow y \quad (\text{in}) \rightarrow \text{in} - (X \rightarrow \ast) \cup (X \rightarrow \text{in}(y))\)
- \(CA_X \leftarrow \ldots \quad (\text{in}) \rightarrow \text{in} - (X \rightarrow \ast) \cup (X \rightarrow \perp)\)
- \(\text{CA}_X \leftarrow \text{ instanceof c goto } ll \text{ else } ll (\text{in}) =\)
  - \(\text{in} - (X \rightarrow \ast) \cup (X \rightarrow \text{in}(c) \cap \text{Subclasses}(c))\) (for \(ll\))
  - \(\text{in} - (X \rightarrow \ast) \cup (X \rightarrow \text{in}(c) - \text{Subclasses}(c))\) (for \(ll\))

Use info at sends, subclass tests
- \(x \leftarrow y.f\text{oo}(z)\)
- \(\text{if } x \text{ instanceof } c \text{ goto } \text{ lab1 else } \text{ lab2}\)

Compose class analysis with inlining, etc.

**Limitations of intraprocedural analysis**

Don’t know classes of
- formals
- results of non-inlined messages
- contents of instance variables

Don’t know complete set of classes in program
\(\Rightarrow\) can’t learn much from static type declarations

Can improve information by:
- looking at dynamic profiles
- specializing methods for particular receiver/argument classes
- performing interprocedural class analysis
  - flow-sensitive & -insensitive methods
  - context-sensitive & -insensitive methods
Profile-guided class prediction

Can exploit dynamic profile information, if static info lacking

Monitor receiver class distributions for each send
Recompile program, inserting run-time class tests for common receiver classes
  • on-line (e.g. in Self [Hölzle & Ungar 96], Jikes, ...)
  • or off-line (e.g. in Vortex, ...)

Before:
  i := s.area();
After:
  i := (if s.class == Rectangle
         then Rectangle::area(s)
         else s.area());

Specialization

To get better static info,
specialize source method w.r.t. inheriting receiver class
+ compiler knows statically the class of the receiver formal

class Rectangle {
  ...
  int area() { return length() * width(); }
  int length() { ... }
  int width() { ... }
};

class Square extends Rectangle {
  int size;
  int length() { return size; }
  int width() { return size; }
};

If specialize Rectangle::area as Square::area, can inline-expand length() & width() sends

What to specialize?

In Sather, Trellis: specialize for all inheriting receiver classes
  • in Trellis, reuse superclass’s code if no change

In Self: same, but specialize at run-time
  • Self compiles everything at run-time, incrementally as needed
  • will only specialize for (classes x messages) actually used at run-time

In Vortex: use profile-derived weighted call graph to guide specialization
  • only specialize if high frequency & provides benefit
  • can specialize on args, too
  • can specialize for sets of classes w/ same behavior

Flow-insensitive interprocedural static class analysis

Simple idea: examine complete class hierarchy,
put upper limit of possible callees of all messages
  • can now benefit from type declarations, instanceof’s

Class Hierarchy Analysis (CHA) [Dean et al. 96, ...]

class Shape {
  abstract int area();
};

class Rectangle extends Shape {
  ...
  int area() { return length() * width(); }
  int length() { ... }
  int width() { ... }
};

class Square extends Rectangle {
  int size;
  int length() { return size; }
  int width() { return size; }
};

Rectangle r = ...;
... r.area() ...
Improvements

Add optimistic pruning of unreachable classes
- optimistically track which classes are instantiated during analysis
- don't make call edge to any method not inherited by an instantiated class
- fill in skipped edges as classes become reachable
- $O(N)$

Rapid Type Analysis [Bacon & Sweeney 96]: in C++

Add intraprocedural analysis
[Diwan et al. 96]: in Modula-3, w/o optimistic pruning, w/ flow-sensitive intraprocedural analysis after flow-insensitive call graph construction

Type-inference-style analysis à la Steensgaard
- compute set of classes for each "type variable"
- use unification to merge type variables
- can blend with propagation, too
[DeFouw et al. 98, Grove & Chambers 01]: in Vortex

Flow-sensitive interprocedural static class analysis

Extend static class analysis to examine entire program
- infer argument & result class sets for all methods
- infer contents of instance variables and arrays

Compute call graph and class sets simultaneously, through optimistic iterative refinement

Use worklist-based algorithm, with procedures on the worklist

Initialize call graph & class sets to empty
Initialize worklist to main

To process procedure off worklist:
- analyze, given class sets for formals:
  - perform method lookup at call sites
  - add call graph edges based on lookup
  - update callee(s) formals' sets based on actuals' class sets
  - if a callee method's argument set changes, add it to worklist
  - if result set changes, add caller methods to worklist
  - if contents of an instance variable or array changes, add all accessing methods to worklist

Example

```java
static void main() {
    3.foo().print();
    "hi".foo().print();
}

Object foo() {
    return this.p();
}

Object p() {
    return this;
}
```

A problem

Simple context-insensitive approach smears together effects of polymorphic methods

E.g. `foo` in example
E.g. `min` function:

```
  a := min(3, 4)  s := min("apple", "abacus")

  min(x, y) {
      if x <= y then return x
      else return y
  }
```

Similar smearing for polymorphic data structures
- readers of some array see all classes stored in any array

Smearing makes analysis slow for big programs

Solution: context-sensitive interprocedural analysis
**k-CFA-style analyses**

Idea: reanalyze method for each distinct stack of \( k \) callers
+ avoids smearing across some callers
- fails for polymorphic libraries of depth > \( k \)
- doesn’t address polymorphic data structures
- requires time exponential in \( k \)

[Oxhej et al. 92]: \( k = 1 \), for toy language

More sophisticated idea: iteratively reanalyze program,
expanding \( k \) in parts of program that matters
+ start with \( k = 0 \)
+ analyze program, building data flow graph
+ identify merges in graph that caused smearing, split apart
+ repeat, following splitting directives,
  till no more improvements possible
+ expend effort exactly where useful
+ works for polymorphic data structures, too
- complicated, particularly in recording confluences
- initial \( k=0 \) analysis can be expensive,
  iteration can be expensive

[Plevyak & Chien 94]

---

**Partial summary function-style analyses**

Cartesian Product Algorithm [Agesen 95]

Idea: analyze methods for each tuple of singleton classes of arguments
+ cache results and reuse at other call sites

\[
\begin{align*}
\text{min}((\text{int},\text{float}), (\text{int})) & \Rightarrow (\text{int}) \\
\text{min}((\text{float},\text{int}), (\text{int})) & \Rightarrow (\text{int},\text{float}) \\
\text{min}((\text{string},\text{string}), (\text{string})) & \Rightarrow (\text{string})
\end{align*}
\]

Analyze & cache:
- precise analysis of methods
- fairly simple
  - combinatorial blow-up (but polymorphic, not exponential)
  - doesn’t address polymorphic data structures