Interprocedural Analysis with Data-Dependent Calls

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

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 address taken, & escape (if known)

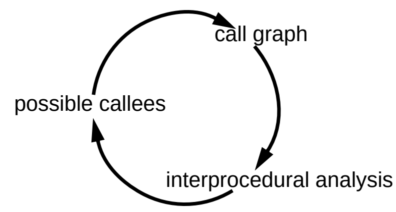
Could do analysis to compute possible callees/receiver classes

- intraprocedural analysis OK
- interprocedural analysis better
- context-sensitive interprocedural analysis even better

Circularity dilemma

Problem:

1. to compute possible callees,
   decide to do interprocedural analysis
2. to do interprocedural analysis,
   need a call graph
3. to construct a call graph,
   need to compute possible callee functions
1. to compute possible callees, ...



How to break vicious cycle?

A solution: optimistic iterative analysis

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 flow-insensitive, 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

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

Example

```
proc main() {
  proc p(pa) { return pa(d); }
  return b(p);
}

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

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

proc d(da) {
  proc r(ra) { return da; }
  return c(r);
}
```

Context-sensitive analyses

Can get more precision through
context-sensitive interprocedural analysis

k-CFA (**c**ontrol **f**low **a**nalysis) [Shivers 88 etc.]

- analyze Scheme programs
- context key: sequence of *k* enclosing call sites
- k=0 ⇒ context-insensitive
- k=1 ⇒ reanalyze for each call site (but not transitively)
- – loses precision beyond *k* recursive calls
- – cost is exponential in *k*, even if no gain in precision

An alternative:

- context key: set of possible functions for arguments
- + avoid weaknesses of *k*-CFA:
  - only expend effort if possibly beneficial
  - never hits an arbitrary cut-off
- worst-case cost proportional to $|\text{Functions}|^{\text{MaxNumberOfArgs}}$
  - how big can this get?

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

Solution: **static class analysis**

- compute set of possible classes of expressions

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

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

Intraprocedural class analysis

Propagate sets of bindings of
variables to sets of classes through CFG

e.g. {x → {Int}, y → {Vector,String}}

Flow functions:

- `x := `**`new`** *C*:
  Succ = Pred – {x→∗} ∪ {x→{*C*}}
- `x := y`:
  Succ = Pred – {x→∗} ∪ {x→Pred(y)}
- `x := ...`:
  Succ = Pred – {x→∗} ∪ {x→⊥}
- **`if`** `x` **`instanceof`** *C* **`goto`** `L1` **`else`** `L2`:
  Succ_{L1} = Pred – {x→∗} ∪
  {x → {$D \in$ Pred(x) | $D \in$ *Subclasses*(*C*)}}
  Succ_{L2} = Pred – {x→∗} ∪
  {x → {$D \in$ Pred(x) | $D \notin$ *Subclasses*(*C*)}}

Use info at sends, type tests

- `x := y.foo(z)`
- **`if`** `x` **`instanceof`** *class* **`goto`** `L1` **`else`** `L2`

Compose inlining of statically bound sends with class analysis

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
⇒ can't learn much from static type declarations

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

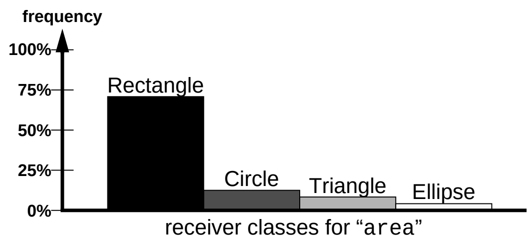
Profile-guided class prediction

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]) or off-line (e.g. in Vortex)



Before:

```
i := s.area();
```

After:

```
i := (if s.class == R
      then s.Rect::area()
      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 $\times$ 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

Example:

```
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 arc to any method whose class isn't reachable
- fill in skipped arcs as classes become reachable
- O(*N*)

[Bacon & Sweeney 96]: in C++

Add intraprocedural analysis

[Diwan *et al.* 96]: in Modula-3, w/o optimistic pruning,
w/ flow-sensitive interprocedural 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

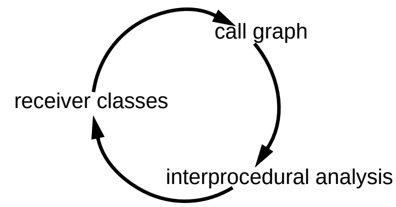
[DeFouw *et al.* 98, Grove & Chambers 01]: in Vortex

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

The standard problem:
constructing the interprocedural call graph



And the standard solution: optimistic iteration

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:

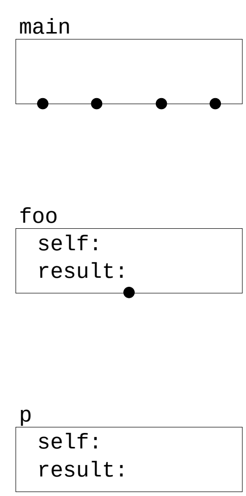
- analyze, given class sets for formals:
  - perform method lookup at call sites
  - add call graph edges based on lookup
  - update callee(s) formals' sets based on actuals' class sets
- if a callee method's argument set changes, add it to worklist
- if result set changes, add caller methods to worklist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

Example

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

self.foo() {
  return self.p();
}

self.p() {
  return self;
}
```



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) {                           x ∈ {int,string}
  if x <= y then return x             y ∈ {int,string}
  else return y }                     result ∈
                                          {int,string}
```

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*

[Oxhøj *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 transfer 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

+ precise analysis of methods
+ fairly simple
– combinatorial blow-up (but polymorphic, not exponential)
– doesn't address polymorphic data structures

Example of CPA

```
min({int,float},{int})    min({string},{string})

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

Analyze & cache:

```
min({int},{int}) ⇒ {int}
min({float},{int}) ⇒ {int,float}
min({string},{string}) ⇒ {string}
```

Open questions

How do algorithms scale to large, heavily-OO programs?

How much practical benefit is interprocedural analysis?

How appropriate are algorithms for different kinds of languages?

[Grove *et al.* 97]:
looked at first three questions for propagation-based analyses, with mostly negative results
(couldn't get both scalability and usefulness)

[Defouw *et al.* 98]:
looked at blend of unification & propagation, with encouraging results

[Grove & Chambers 01]:
journal summary of previous two papers

How does interprocedural analysis interact with separate compilation? rapid program development?

Vortex interprocedural analysis framework

Vortex allows construction of interprocedural analyses from intraprocedural ones

- assumes call graph already built
- doesn't support transformations w/ interprocedural analysis

User invokes `ip_traverse` to perform analysis,
compute table mapping procedures to summary functions:

```
call_graph.ip_traverse(
  initial_input_analysis_info,
  initial_output_analysis_info,  (T, for recursion)
  λ(proc, info, callback){
      let intra_info := ...(info, callback)...;
      proc.cfg.traverse(...) },
  context_sensitivity_strategy_fn)
→ (proc→(input→output))
```

Interprocedural framework & user's intraprocedural analysis call each other to traverse call graph (`callback`)

User's context sensitivity strategy specified by function:

```
λ(prevs:set[input_info],new:input_info)
→(dropped:set[input_info],added:new_info)
```