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

- e.g. call all possible functions
- e.g. all possible methods with matching name

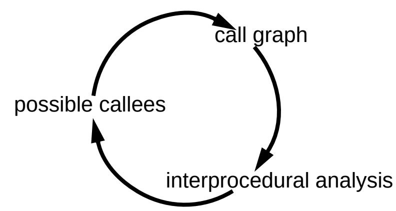
Could do analysis to compute possible callees/receiver classes

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

Circularity dilemma

Problem:

- to do interprocedural analysis, need a call graph
- to construct a call graph, need to know possible callee functions
- to know possible callee functions, need to do interprocedural analysis
- ...



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

A simple flow-insensitive, context-insensitive analysis:

- for each (formal, local, 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 ⇒ empty call graph

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

Example

```
proc main() {
  proc p(f1) { return f1(d); }
  return b(p);
}

proc b(fb) {
  proc q(f2) { return d(d); }
  c(q);
  return fb(d);
}

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

proc d(fd) {
  proc r(f3) { return fd; }
  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, using as context *k* enclosing call sites
- k=0 ⇒ context-insensitive

Could design transfer-function-based context, partial or total

+ avoid weaknesses of *k*-CFA
- not done by anyone?

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

- direct cost: extra tests

Solution: static class analysis

- compute set of possible classes of expressions

Classes of receiver enables compile-time method lookup

Given 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

Classes of argument to run-time class/subclass test
enables constant-folding of test

Intraprocedural class analysis

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

e.g. {x → {Int}, y → {Vector,String}}

- or single set of classes on edges of dataflow graph

Flow functions:

- `x := `**`new`**` `*`class`*:
  Succ = Pred[x→*`class`*]
- `x := y`:
  Succ = Pred[x→Pred(y)]
- `x := ...`:
  Succ = Pred[x→⊥]
- **`if`**` x `**`instanceof`**` `*`class`*` `**`goto`**` L1 `**`else`**` L2`:
  Succ_{L1} = Pred[x→*`class`*]
  Succ_{L2} = Pred[x→(Pred(x)–*`class`*)]

Use info at sends, type tests

- `x := `**`send`**` foo(y,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 class of

- formals
- results of non-inlined messages
- contents of instance variables

Improve information by:

- looking at dynamic profiles
- specializing methods for particular receiver/argument classes
- performing interprocedural class analysis
  - flow-insensitive methods
  - flow-sensitive methods

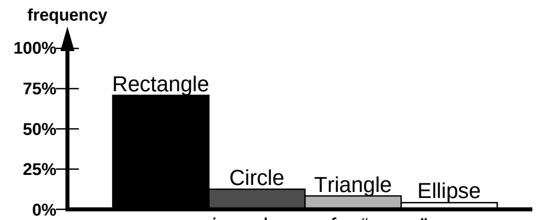
Profile-guided class prediction

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



receiver classes for “`area`”

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 × 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

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

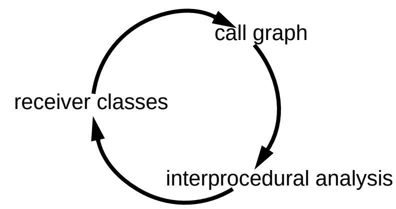
[DeFouw *et al.* 98]: in Vortex

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

Standard problem: constructing the interprocedural call graph



Standard solution

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

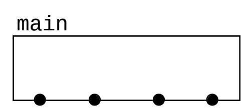
Initialize worklist to `main`

To process procedure off worklist:

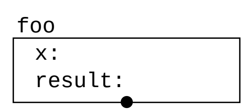
- analyze, given class sets for formals:
  - perform method lookup at call sites
  - add call graph edges based on lookup
  - update callee(s) formals' sets based on actuals' class sets
- add callee method(s) to worklist, if their argument sets changed
- add caller method(s) to worklist, if result set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

Example

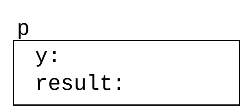
```
main {
  print(foo(3));
  print(foo(5.6));
}
```




```
foo(x) {
  return p(x);
}
```




```
p(y) {
  return y;
}
```



A problem

Simple context-insensitive approach smears together effects of polymorphic methods

E.g. `min` function

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

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

```
x ∈ {int,string}
y ∈ {int,string}
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 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

Idea: iteratively reanalyze program, expanding *k* in parts of program that matters

- start with *k* = 0
- analyze program, building data flow graph
- identify bad confluences in graph, 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

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)
```