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 fixpoint solution

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);
}
```

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

e.g. {x → {Int}, y → {Vector,String}}

Flow functions:
- `x := const`: Succ = Pred[x→classof(*const*)]
- `x := new class`: Succ = Pred[x→*class*]
- `x := y`: Succ = Pred[x→Pred(y)]
- `x := ...`: Succ = Pred[x→⊥]
- **if** `x.class == 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.class = 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

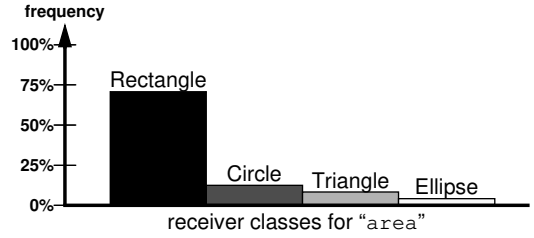
Improve information by:
- looking at dynamic profiles
- specializing methods for particular receiver/argument classes
- performing interprocedural class analysis
  - "flow-insensitive" methods
  - "flow-sensitive" methods

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
- in Self: on-line [Hölzle & Ungar, PLDI'94 & OOPSLA'94]
- in Cecil: off-line [Grove *et al.*, OOPSLA'95]



Before:

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

After:

```
i := s.class == R ? s.Rect::area() : 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 {
  ...
  virtual int area() {
    return length() * width(); }
  virtual int length() { ... }
  virtual int width() { ... }
};

class Square : Rectangle {
  int size;
  virtual int length() { return size; }
  virtual int width() { return size; }
};
```

If specialize `Rectangle::area` as `Square::area`, can inline-expand `length()` & `width()` virtual fn calls

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

[Chambers *et al.*, PLDI'89 & OOPSLA'89]

In Cecil: 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

[Dean *et al.*, PLDI'95]

Flow-insensitive interprocedural static class analysis

Simple idea: examine complete class hierarchy, put upper limit of possible callees of all messages

Example:

```
class Shape {
  virtual int area() = 0;
};
class Rectangle : Shape {
  ...
  virtual int area() {
    return length() * width(); }
  virtual int length() { ... }
  virtual int width() { ... }
};
class Square : Rectangle {
  int size;
  virtual int length() { return size; }
  virtual 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, OOPSLA'96]: in C++

Add intraprocedural analysis

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

Type-inference-style analysis

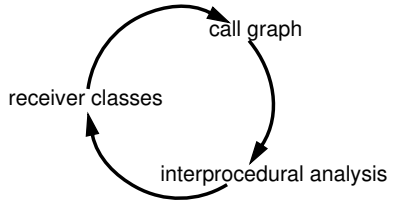
- compute set of classes for each "type variable"
- use unification to merge type variables

[DeFouw *et al.*, POPL'98]: in Cecil & Java, plus flow-sensitive analysis

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

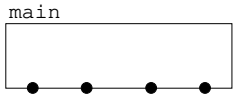
Use worklist-based algorithm, with procedures on the worklist

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

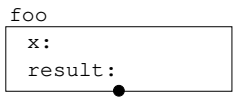
To process procedure off worklist:
- analyze, given class sets for formals:
  - perform method lookup at call sites
  - add call graph edges based on lookup
  - update callee(s) formals' sets based on actuals' class sets
- add callee method(s) to worklist, if their argument sets changed
- add caller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

Example

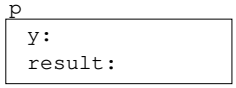
```
main {
  print(foo(3));
  print(foo(5.6));
}
```




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



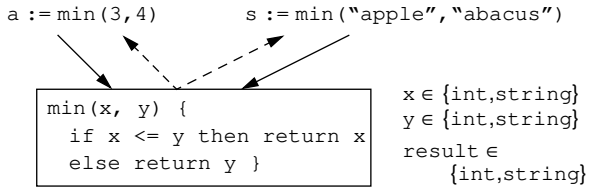

```
p(y) {
  return y;
}
```



A problem

Simple approach smears together effects of polymorphic methods

E.g. `min` function



Similar smearing for polymorphic data structures
- clients of e.g. `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.*, ECOOP'92]: *k* = 1, for toy language

Adaptive expansion

[Plevyak & Chien, OOPSLA'94]

Fixed *k*: too little work in some places, too much in others

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

- start with *k* = 0
- analyze program, building data flow graph
- identify 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
– didn't deal with first-class closures

Results

Language: Concurrent Aggregates
dynamically-typed pure OO language with first-class selectors & continuations, but no closures

Benchmarks:
< 1K lines, scientific data-parallel codes

Result:
interprocedural analysis plus specialization removes virtually all dynamic dispatches

How OO are benchmarks?

How will algorithm scale to larger programs?

An alternative

Cartesian Product Algorithm [Agesen, OOPSLA'95]

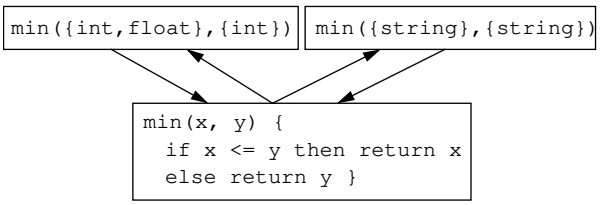
Idea: analyze methods for each tuple of singleton classes of arguments

- cache results and reuse at other call sites

A partial-transfer-function-based context-sensitive analysis

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

Example of CPA



Analyze & cache:

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


Results

Language: Self
a dynamically-typed pure OO language with closures

Benchmarks:
mostly <2K line programs, one 10K line program

Results:

- good speed-ups
- demonstrated tools to extract self-contained application from development environment

Benchmarks definitely OO

Scalability?

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?

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

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

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)
```