Static analysis of OO programs

If knew exact class of receiver at compile-time,
could replace dynamically-dispatched send
with statically-bound call

+ no run-time dispatching overhead
+ can apply inlining, interprocedural analysis

If knew receiver was one of small # of classes at compile-time,
could still do decent interprocedural analysis

Particularly important in dynamically-typed and/or pure OOLs

- message sends frequent
- static type declarations on object types don't help compiler

In Self:

– w/o static analysis (and other tricks):
10-1000x slower than optimized C

+ w/ static analysis: 50% of optimized C

[caveat: small integer/array benchmarks]

Intraprocedural class analysis

Propagate forward sets of bindings of
variables to sets of classes

e.g. {x → {Int}, y → {Vector,String}}

Base cases:

- `x := const;`
- `x := new class;`
- otherwise: bind variable to universal set of classes

Propagate through assignments

Exploit at sends, type tests

- `if x.class = Vector goto ...`
  - narrow class sets after type test
- `x := send "foo"(y,z)`

Weakness: don't know class of

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

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

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

Class hierarchy analysis

Even if don't know exact receiver class,
examining class hierarchy often enables static binding

Example:

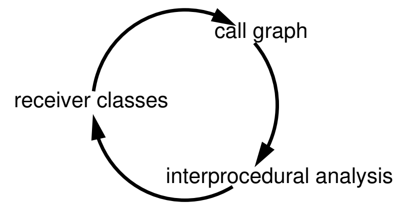
```
class Shape {
  virtual int area() = 0;
};
class Rectangle : Shape {
  ...
  virtual int area() {
    return length() * width(); }
  virtual int length() { ... }
  virtual int width() { ... }
};
class Square : Rectangle {
  int size;
  virtual int length() { return size; }
  virtual int width() { return size; }
};

Rectangle *r = ...;
... r->area() ...
```


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

A hard problem: constructing the interprocedural call graph



Solution

Compute call graph and class sets simultaneously,
through iterative refinement

Use worklist-based algorithm, with procedures on the worklist

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

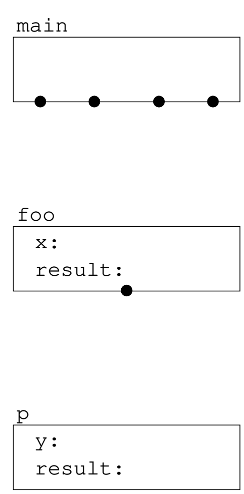
To process procedure off worklist:

- 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

Variations: depth-first vs. breadth-first vs. recursive analysis at calls

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

```
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

- clients of e.g. `array` see all classes stored in any array

Smearing makes analysis slow for big programs [Grove 95]

Simple interprocedural analysis is not **context-sensitive**

k-deep expansion approaches

Idea: reanalyze method for each call site

Fancier idea: reanalyze method for stack of *k* callers

+ avoids smearing across some callers
– fails for polymorphic libraries of depth > *k*
– requires (much) more analysis time
– polymorphic data structures?

Adaptive expansion

[Plevyak & Chien 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 improvement

Cartesian Product Algorithm [Agesen 95]

Idea: instead of (lazily) expanding by *k*,
eagerly reanalyze for each combination of singleton classes

- cache results and reuse at other call sites

+ precise analysis of methods
+ fairly simple
+ can be (is?) included in Plevyak & Chien's algorithm
– combinatorial blow-up (better worst-case; average case?)
– doesn't avoid smearing of 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:

$\texttt{min(\{int\},\{int\})} \Rightarrow \texttt{\{int\}}$
$\texttt{min(\{float\},\{int\})} \Rightarrow \texttt{\{int,float\}}$
$\texttt{min(\{string\},\{string\})} \Rightarrow \texttt{\{string\}}$

(Context-sensitive partial transfer function-based idea,
like Wilson & Lam's pointer analysis)

Results

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

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

Results:

- good speed-ups,
  competitive with profile-guided optimization
- demonstrated tools to extract self-contained application
  from development environment (**tree shaking**)

Benchmarks definitely OO
Scaling?

Interprocedural analysis of statically-typed OO L's

Problem is easier in e.g. C++ and Modula-3

- hybrid language, so precise analysis is less important
- can exploit static type declarations of arguments

Simple idea:
compute initial approximate call graph using only static type declarations and/or intraprocedural analysis,
then do interprocedural analysis with "head start"

- may not achieve as precise a solution as if started from scratch

An improvement: tree shaking

- track which classes are reachable during analysis
- don't make call arc to any method whose containing class isn't reachable
- may require reanalysis as classes become reachable

[Diwan *et al.*]: in Modula-3
[Bacon *et al.*]: in C++, with tree-shaking

Open questions

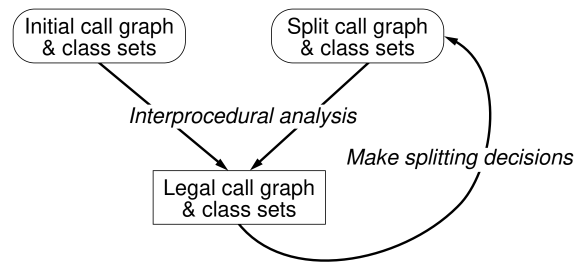
How do algorithms scale to large heavily-OO programs?

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?

Engineering a scalable, effective analysis



Possible initial call graphs:

- empty (a.k.a. optimistic)
- full (a.k.a. pessimistic)
  - different for statically and dynamically-typed languages
- after intraprocedural analysis
  - superset of optimal
- profile-derived
  - subset of optimal

Variations in analyis algorithm, w.r.t. splitting & tree-shaking

Variations in splitting decision making