Compiling Object-Oriented Languages

Key features:

- inheritance (possibly multiple)
- subtyping & subtype polymorphism
- message passing, dynamic binding

Subtype polymorphism is the key problem

- support uniform representation of data
  (analogous to boxing for polymorphic data)
  - e.g. every object has a class pointer, or a virtual fn table pointer, at a known offset
- organize layout of data to make instance variable access and method lookup & invocation fast
  - multiple inheritance complicates this
- perform static analysis to bound polymorphism
- perform transformations to reduce polymorphism

Static analysis of OO programs

If knew exact class of receiver at compile-time,
no subtype polymorphism would remain

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

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

Particularly important in dynamically-typed and/or pure OOLs

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

In Self, Cecil:

– w/o static analysis (and other tricks):
  10-100x slower than optimized C

+ w/ static analysis: 50% of optimized C
  (for small benchmarks)

Intraprocedural class analysis

Propagate 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:

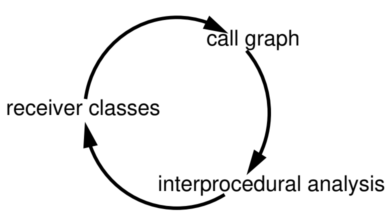
```
class Shape {
  virtual int area() = 0;
};
class Rectangle : Shape {
  ...
  virtual int area() {
    return length() * width(); }
  virtual int length() { ... }
  virtual int width() { ... }
};
class Square : Rectangle {
  int size;
  virtual int length() { return size; }
  virtual int width() {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

Use worklist-based algorithm, with procedures on the worklist

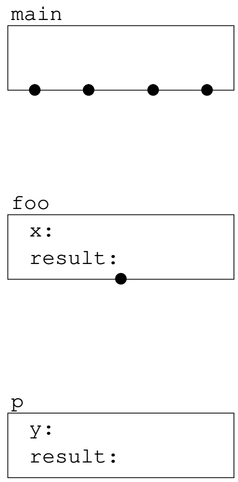
Initialize call graph & class sets to empty
Initialize worklist to `main`

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

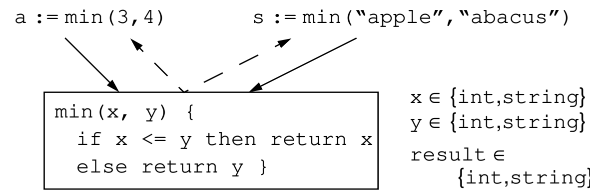
```
main {
  print(foo(3));
  print(foo(5.6));
}

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 [Grove 95]

Context-sensitive analysis

Avoid smearing through context-sensitive analysis

Approaches:

- *k*-CFA-style analysis
- iterated *k*-CFA analysis, adaptively identifying *k* in different parts of program [Plevyak & Chien 94]
- using argument class sets as context key [Agesen 95]

Flow-insensitive interprocedural analysis

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

[Diwan *et al.*]: in Modula-3

An improvement: tree shaking

- optimistically 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

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

An even better improvement: type-inference-style analysis

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

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?

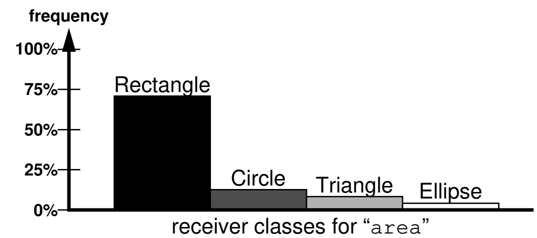
How does interprocedural analysis interact with separate compilation? rapid program development?

Profile-guided receiver class prediction

Can exploit dynamic profile information if static info lacking

Monitor receiver class distributions for each send

- on-line [Self] or off-line [Vortex]



Recompile program, inserting run-time class tests for common receiver classes

Before:

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

After:

```
i := s.class == R ? s.Rect::area() : s.area();
```


Some experimental results