Quick recap of abstract interpretation

Approximation

- Want to show:

$$\alpha\left(\bigsqcup_{i=0}^{\infty} F_c^i(\bot_c)\right) \sqsubseteq_a \bigsqcup_{i=0}^{\infty} F_a^i(\bot_a) \quad (1)$$

$$\bigsqcup_{i=0}^{\infty} F_c^i(\bot_c) \sqsubseteq_c \gamma\left(\bigsqcup_{i=0}^{\infty} F_a^i(\bot_a)\right) \quad (2)$$

Cousot and Cousot 77

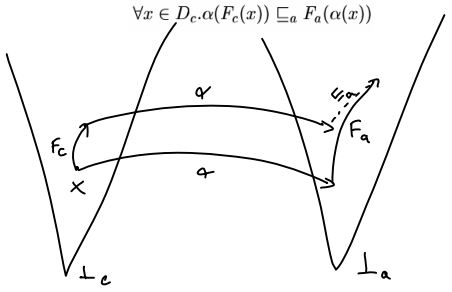
- Cousot and Cousot show that the following conditions are sufficient for proving (1) and (2):

$$\alpha \text{ and } \gamma \text{ are continuous} \quad (3)$$

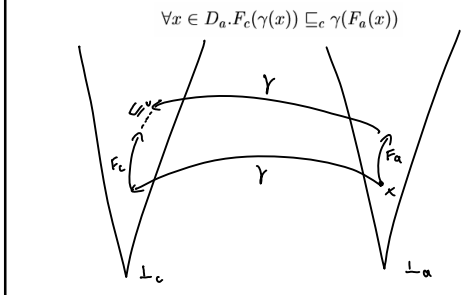
$$\forall x \in D_c.\alpha(F_c(x)) \sqsubseteq_a F_a(\alpha(x)) \quad (4)$$

$$\forall x \in D_a.F_c(\gamma(x)) \sqsubseteq_c \gamma(F_a(x)) \quad (5)$$

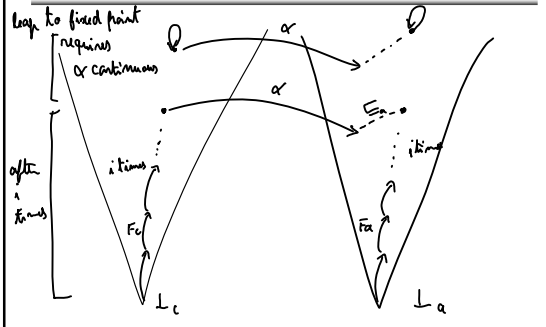
Let's look at the α condition

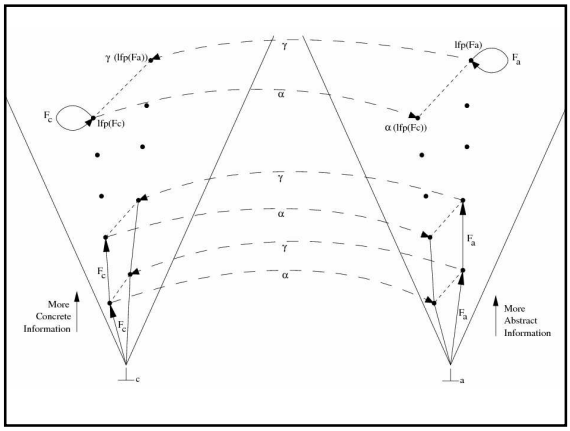


Let's look at the γ condition



Link between local and global



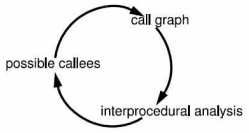


Optimizing OO languages

Interproc analysis with data-dependent calls

Previously, assumed call graph prior to interprocedural analysis

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



How to break the cycle?

How to break the cycle?

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 addresses taken, & escape (if known)

Better solution

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 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

Algorithm

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

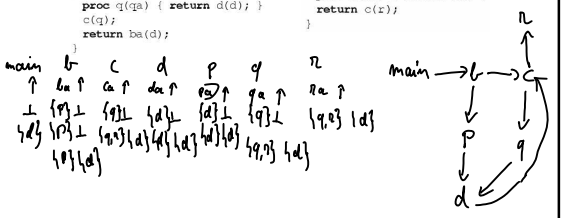
Example

```
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);
}
```



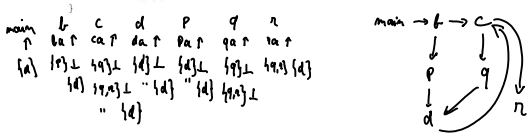
Example

```
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);
}
```



In the context 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

Class analysis

Solution to both problems: **static class analysis**

- compute set of possible classes of objects computed by each expression

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

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

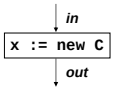
Intraprocedural class analysis

- Domain: Map [vars, 2^(classes)]

May Analysis ↳ ⊥ = ∅

⊤ = classes

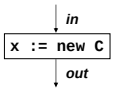
Flow functions



out = in[X ↦ {C}]

out = in[X ↦ in[Y]]

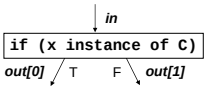
Flow functions

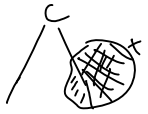


out = in[x → { C }]

out = in[x → in[y]]

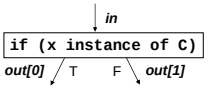
Flow functions





out[0] = in[x →
in[x] ∩ SubClasses(C)]
out[1] = in[x →
in[x] − SC(C)]

Flow functions



out[0] = in[x → in[x] ∩ subclasses(C)]

out[1] = in[x → in[x] – subclasses(C)]

Limitations of intraproc 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

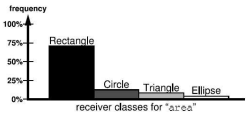
Can improve information by:
- looking at dynamic profiles
- specializing methods for particular receiver/argument classes
- performing interprocedural class analysis
  - flow-sensitive & -insensitive methods
  - context-sensitive & -insensitive methods

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



Before: b := new Rect

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

After:

```
i := (if s.class == Rectangle
      then Rectangle::area(s)
      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

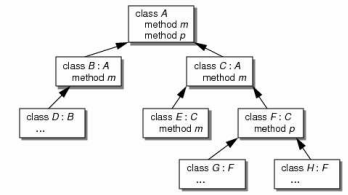
Flow insensitive interproc class analysis

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

- can now benefit from type declarations, instanceof's

Class Hierarchy Analysis (CHA) [Dean *et al.* 96, ...]

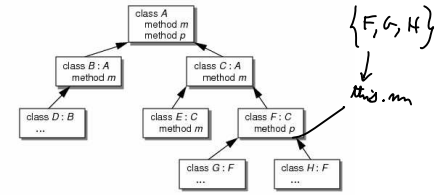
Example



CHA algorithm

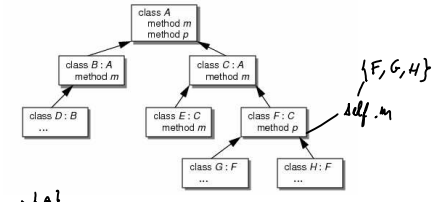
- Compute applies-to-set: for each method, determine the set of classes the method applies to.
- At a message send a.m, take the set of classes inferred for a, and check if this set overlaps with the applies-to sets of all methods that implement m.
- If only one set overlaps, then the message send should go to that method.

Example



A :: m → {A}
B :: m → {B, D}
C :: m → {C, F, G, H}
E :: m → {E}

Example



A :: m → {A}
B :: m → {B, D}
C :: m → {C, F, G, H}
E :: m → {E}

Improvements

Add optimistic pruning of unreachable classes

- optimistically track which classes are instantiated during analysis
- don't make call edge to any method not inherited by an instantiated class
- fill in skipped edges as classes become reachable
- O(*N*)

Rapid Type Analysis [Bacon & Sweeney 96]: in C++

Flow-sensitive interproc 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

Compute call graph and class sets simultaneously, through optimistic iterative refinement

Use worklist-based algorithm, with procedures on the worklist

Algorithm

Initialize call graph & class sets to empty

Initialize worklist to `main`

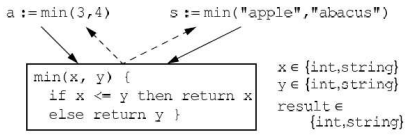
To process procedure off worklist:
- analyze, given class sets for formals:
  - perform method lookup at call sites
  - add call graph edges based on lookup
  - update callee(s) formals' sets based on actuals' class sets
- if a callee method's argument set changes, add it to worklist
- if result set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

Problem

Simple context-insensitive approach smears together effects of polymorphic methods

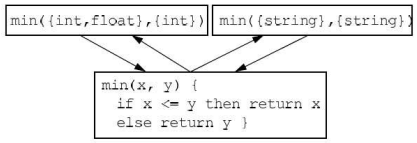
E.g. `foo` in example

E.g. `min` function:



Partial transfer functions

Idea: analyze methods for each tuple of singleton classes of arguments
- cache results and reuse at other call sites



Analyze & cache:

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