Interprocedural Analysis

Last time

– Alias analysis

Today

– Interprocedural analysis

Using Alias Information

Example: reaching definitions

– Compute at each point in the program a set of (s,v) pairs, indicating that statement s may define variable v

Flow functions

– s: `*p = x;`
$out_{reach}[s] = \{(s,\mathbf{z}) \mid (\mathbf{p}\rightarrow\mathbf{z}) \in in_{may\text{-}pt}[s]\} \cup$
$(in_{reach}[s] - \{(t,\mathbf{y})\ \forall t \mid (\mathbf{p}\rightarrow\mathbf{y}) \in in_{must\text{-}pt}[s]\}$

– s: `x = *p;`
$out_{reach}[s] = \{(s,\mathbf{x})\} \cup (in_{reach}[s] - \{(t,\mathbf{x})\ \forall t\}$

– . . .

What happens with Points-to information at function calls

Question

– How do function calls affect our points-to sets?

e.g.,
```
p1 = &x;
p2 = &p1;
...
foo();
```

{(**p1**→**x**), (**p2**→**p1**)}

???

Be conservative

– Assume that any reachable pointer may be changed
– Pointers can be "reached" via globals and parameters
  – May pass through objects in the heap
– Can be changed to anything reachable or something else
– Can we prune aliases using types?

Problem

– Lose a lot of information

General Need for Interprocedural Analysis

Procedural abstraction

– Cornerstone of programming
– Introduces barriers to analysis

Example
```
x = 7;
foo(p);
y = x+3;
```

Does **foo()** modify **x**?

Example
```
void f(int x)
{
   if (x)
      foo();
   else
      bar();
}
. . .
f(0);
f(1);
```

What is the calling context of **f()**?

Interprocedural Analysis

Goal

- Avoid making conservative assumptions about the effects of procedures and the state at call sites

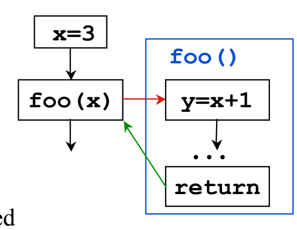
Terminology

```
int a, e;                  // Globals
void foo(int &b, &c)       // Formal parameters (passed
{                          //   by reference)
   b = c;
}
main()
{
   int d;                  // Local variables
   foo(a, d);              // Actual parameters
}
```


Naive Approach: ICFG

Compose the CFGs for all procedures

- Connect call nodes to entry nodes of callees
- Connect return nodes of callees back to return node in caller
- Called **interprocedural control-flow graph**



Pros

- Simple
- Intraprocedural analysis algorithms can work unchanged
- Reasonably effective

Cons

- Accuracy?
- Performance?
- No separate compilation
- Problematic

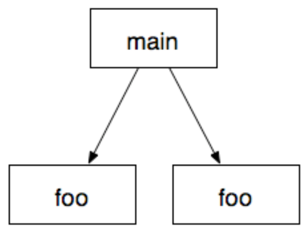
Smears information from different contexts.
IDFA converges in d+2 iterations, where d is the Number of nested loops [Kam & Ullman '76].
Graphs will have many cycles (one for each callsite).
Graphs will often be huge.

Brute Force: Full Context-Sensitive Interprocedural Analysis

Invocation Graph [Emami94]

- Re-analyze callee for all distinct calling paths
- Pro: precise
- Cons: exponentially expensive, recursion is tricky

```
int a, e;
void foo(int &b, &c)
   b = c;
}
main()
{
   int d;
   foo(a, d);
   foo(e, a);
}
```

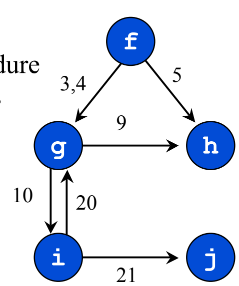


Middle Ground: Use Call Graph and Compute Summaries

```
1   procedure f()
2   begin
3       call g()
4       call g()
5       call h()
6   end
7   procedure g()
8   begin
9       call h()
10      call i()
11  end
12  procedure h()
13  begin
14  end
15  procedure i()
16      procedure j()
17      begin
18      end
19  begin
20      call g()
21      call j()
22  end
```

Goal

- Represent procedure call relationships



Definition

- If program P consists of n procedures: $p_1, \ldots, p_n$
- Static **call graph** of P is $G_P = (N,S,E,r)$
  - $N = \{p_1, \ldots, p_n\}$
  - $S = \{\text{call-site labels}\}$
  - $E \subseteq N \times N \times S$
  - $r \in N$ is **start node**

Interprocedural Analysis: Summaries

Compute summary information for each procedure

- Summarize effect/result of called procedure for callers
- Summarize effect/input of callers for called procedure

Store summaries in database

- Use when optimizing procedures later

Pros

- Concise
- Can be fast to compute/use
- Separate compilation practical

Cons

- Imprecise if only summarize per procedure

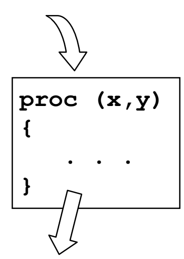
Two Types of Information

Track information that flows into a procedure

- Sometimes known as **propagation problems**
  e.g., What formals are constant?
  e.g., Which formals are aliased to globals?

Track information that flows out of a procedure

- Sometimes known as **side effect problems**
  e.g., Which globals are def'd/used by a procedure?
  e.g., Which locals are def'd/used by a procedure?
  e.g., Which actual parameters are def'd by a procedure?



Examples

Propagation Summaries

- MAY-ALIAS: The set of formals that may be aliased to globals and each other
- MUST-ALIAS: The set of formals that are definitely aliased to globals and each other
- CONSTANT: The set of formals that must be constant

Side-effect Summaries

- MOD: The set of variables possibly modified (defined) by a call to a procedure
- REF: The set of variables possibly read (used) by a call to a procedure
- KILL: The set of variables that are definitely killed by a procedure (*e.g.*, in the liveness sense)

Computing Interprocedural Summaries

Top-down

- Summarize information about the caller (MAY-ALIAS, MUST-ALIAS)
- Use this information inside the procedure body

```
int a;
void foo(int &b, &c){
    . . .
}
foo(a,a);
```

Bottom-up

- Summarize the effects of a call (MOD, REF, KILL)
- Use this information around procedure calls

```
x = 7;
foo(x);
y = x + 3;
```

Context-Sensitivity of Summaries

None (zero levels of the call path)

- Forward propagation: Meet (or smear) information from all callers to particular callee
- Side-effects: Use side-effect information for callee at all callsites

Callsite (one level of the call path)

- Forward propagation: Label data-flow information with callsite
- Side-effects: Affects alias analysis, which in turn affects side-effects

k levels of call path

- Forward propagation: Label data-flow information with k levels of the call path
- Side-effects: Affects alias analysis, which in turn affects side-effects

Bi-Directional Interprocedural Summaries

Interprocedural Constant Propagation (ICP)

- Information flows from caller to callee and back (CONSTANT)

```
int a,b,c,d;
void foo(e){
   a = b + c;
   d = e + 2;
}
foo(3);
```

The calling context tells us that the formal **e** is bound to the constant 3, which enables constant propagation within **foo()**

After calling **foo()** we know that the constant 5 (3+2) propagates to the global **d**

Interprocedural Alias Analysis

- forward propagation: aliasing due to reference parameters
- side-effects: points-to relationships due to multi-level pointers

Improving the Efficiency of the Iterative Algorithm

Jump Functions and Return Jump Functions for ICP

$$J_{callsite}^{formal} = f(actuals, globals, constants)$$

$$R_{function}^{global \text{ or } refparam} = f(formals, globals, constants)$$

```
int a,b,c,d;
void foo(e){
    a = b + c;
    d = e + 2;
}
foo(3);
```

$$J_{foo(3)}^{e} = 3$$

$$R_{foo}^{d} = e + 2$$

$$R_{foo}^{a} = b + c$$

Partial Transfer Functions for Interprocedural Alias Analysis

- funcOutput = PTF(funcInput)
- use memoization
- PTF lazily computed for each input pattern that occurs

Partial Transfer Function [Wilson et. al. 95]

Example [http://www.cs.princeton.edu/~jqwu/Memory/survey.html]

```
main() {
  int *a,*b,c,d;
  a = &c;
  b = &d;
  for (i = 0; i<2; i++) {
    foo(&a,&a);
    foo(&b,&b);
    foo(&a,&b);
    foo(&b,&a);
  }
}
void foo(int** x, int **y){
  int *temp = *x;
  *x = *y;
  *y = temp;
}
```

Concepts

Call graphs

Analysis versus optimization

Approaches

– ICFG, flow-sensitive but not context-sensitive

– Invocation graph, fully context sensitive except for recursion

– Call Graph: Bottom-up, top-down, bi-directional summaries

Context-sensitivity options when using the call graph

Propagation versus side-effect problems

Next Time

Next lecture

– Interprocedural optimization