

Interprocedural Analysis

Today

- Interprocedural analysis

Next

- Alias/Pointer Analysis

Motivation

Procedural abstraction

- Cornerstone of programming
- Introduces barriers to analysis

Example

```
x = 5;  
foo(p);  
y = x+1;
```

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 variable
    foo(a, d);    // Actual parameters
}
```

Interprocedural Analysis vs. Interprocedural Optimization

Interprocedural analysis

- Gather information across multiple procedures (typically across the entire program)
- Can use this information to improve intraprocedural analyses and optimization (*e.g.*, CSE)

Interprocedural optimizations

- Optimizations that involve multiple procedures
e.g., Inlining, procedure cloning, interprocedural register allocation
- Optimizations that use interprocedural analysis

Dimensions of Interprocedural Analysis

Flow-sensitive vs. flow-insensitive

Context-sensitive vs. context-insensitive

Path-sensitive vs. path-insensitive

Flow Sensitivity

Flow-sensitive analysis

- Computes one answer for every program point
- Requires iterative data-flow analysis or similar technique

Flow-insensitive analysis

- Ignores control flow
- Computes one answer for every procedure
- Can compute in linear time
- Less accurate than flow-sensitive

Flow Sensitivity Example

Is x constant?

```
void f(int x)
{
    x = 4;
    . . .
    x = 5;
}
```

Flow-sensitive analysis

- Computes an answer at every program point:
 - x is 4 after the first assignment
 - x is 5 after the second assignment

Flow-insensitive analysis

- Computes one answer for the entire procedure:
 - x is not constant

Context Sensitivity

Context-sensitive analysis

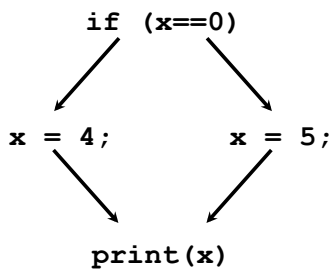
- Re-analyzes callee for each caller
- Also known as [polyvariant](#) analysis

Context-insensitive analysis

- Perform one analysis independent of callers
- Also known as [monovariant](#) analysis

Path Sensitivity Example

Is x constant?



Path-sensitive analysis

- Computes an answer for every path:
 - x is 4 at the end of the left path
 - x is 5 at the end of the right path

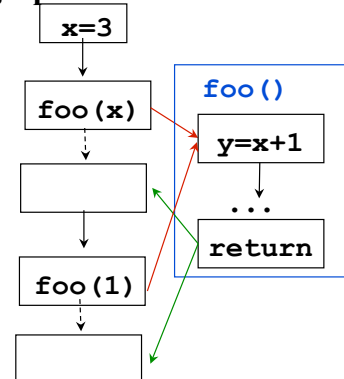
Path-insensitive analysis

- Computes one answer for all path:
 - x is not constant

Interprocedural Analysis: Supergraphs/ICFGs

Compose the CFGs for all procedures via the call graph

- Connect call nodes to **entry** nodes of callees
- Connect **return** nodes of callees back to calls
- Called **control-flow supergraph or ICFG**



Pros

- Simple
- Intraprocedural analysis algorithms work relatively unchanged
- Reasonably effective
- Flow-sensitive

Example

```
{  
    int x, y, a;  
    int *p;  
  
    p = &a;  
    x = 5;  
    foo (&x);  
    y = x + 1;  
}
```

```
foo (int *p)  
{  
    return p;  
}
```

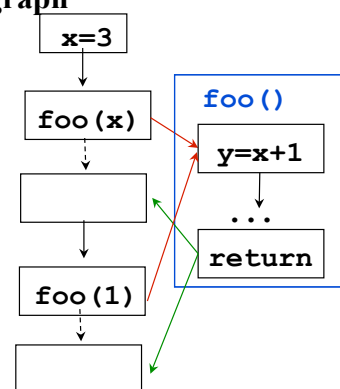
Is x constant?

- With a supergraph, run our same IDFA algorithm
- Determine that $x = 5$

Supergraphs/ICFGs (cont)

Compose the CFGs for all procedures via the call graph

- Connect call nodes to **entry** nodes of callees
- Connect **return** nodes of callees back to calls
- Called **control-flow supergraph or ICFG**



Cons

- Accuracy?
- Performance?
- No separate compilation

Smears information from different contexts.

IDFA is $O(\text{depth} * n^2)$, graphs can be huge

Brute Force: Full Context-Sensitive Interprocedural Analysis

Invocation Graph [Emami94]

- Use an **invocation graph**, which distinguishes all calling chains
- Re-analyze callee for all distinct calling paths
- Pro: precise
- Cons: exponentially expensive, recursion is tricky

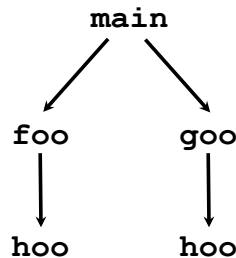
```

void foo(int b)
{ hoo(b); }

void goo(int c)
{ hoo(c); }

main()
{
    int x, y;
    foo(x);
    goo(y);
}

```



Middle Ground: Use Call Graph and Compute Summaries

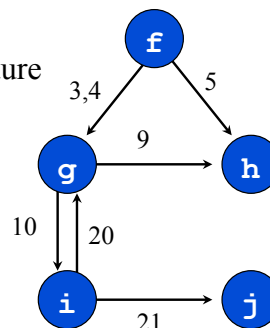
```

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, \dots, p_n
- Static **call graph** of P is $G_P = (N, S, E, r)$
 - $N = \{p_1, \dots, p_n\}$
 - $S = \{\text{call-site labels}\}$
 - $E \subseteq N \times N \times S$
 - $r \in N$ is **start node**

Interprocedural Analysis: Summaries

Compute summary information for each procedure

- Summarize effect of called procedure for callers
- Summarize effect of callers for called procedure

Store summaries in database

- Use later when optimizing procedures

Pros

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

Cons

- Imprecise if only have one summary per procedure

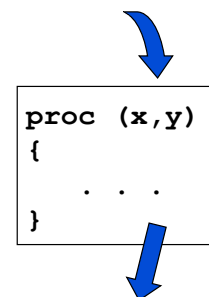
Two Types of Information

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
- DEF: The set of variables that are definitely defined by a procedure (*e.g.*, killed 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, DEF)
- Use this information around procedure calls

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

Context-Sensitivity of Summaries

None (zero levels of the call path)

- Top-down: Meet (or smear) information from all callers to particular callee
- Bottom-up: Use side-effect information for callee at all callsites

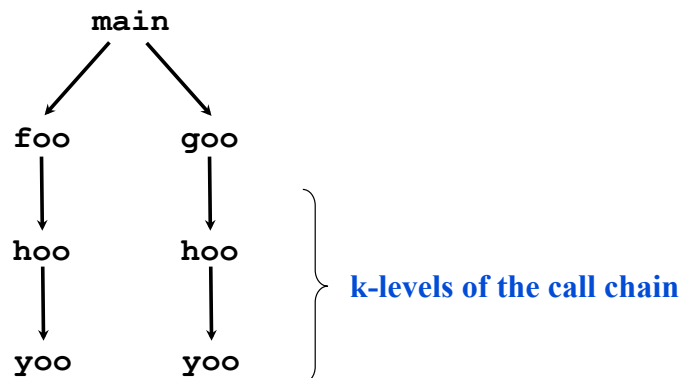
Callsite (one level of the call path)

- Top-down: Label data-flow information with callsite
- Bottom-up: Affects alias analysis, which in turn affects side-effects

Context-Sensitivity of Summaries (cont)

k levels of call path (**k-limiting**)

- Forward propagation: Label data-flow information with k levels of the call path
- Side-effects: label side-effects with k levels of call path



Bi-Directional Interprocedural Summaries

Interprocedural Constant Propagation (ICP)

- Information flows from caller to callee and back

```
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

- Top-down: aliasing due to reference parameters
- Bottom-up: points-to relationships due to multi-level pointers

Interprocedural Constant Propagation [Callahan, et al '86]

Basic Idea

- Extend the global constant propagation by using an extension of the CFG, the **interprocedural value graph**, which represents the movement of values across procedure calls
- Can be either context-sensitive or context-insensitive

Jump Functions

- For a call of procedure q at site s , J_s^x determines the value of the formal parameter x based on the inputs to the procedure p that calls q
- The **support** of J_s^x is the set of inputs to p actually used to compute J_s^x

Interprocedural Value Graph

- Create one node for each jump function J_s^x
- If $x \in \text{support}(J_t^y)$ for some call site t in the procedure, then construct an edge from J_s^x to J_t^y

} Connect J_s^x with nodes that use the value of x

Jump Functions

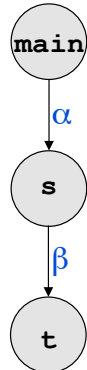
Example

```
main()
{
  int a, b;
  a = 1; b = 2;
 $\alpha$ : s(a,b);
}
```

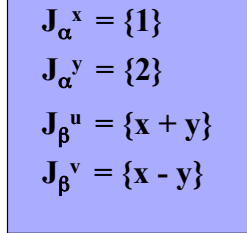
```
void s (int x, y)
{
  int z, w;
  z = x + y;
  w = x - y;
 $\beta$ : t(z,w);
}
```

```
void t(int u, v)
{
  print (u,v);
}
```

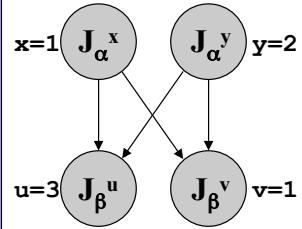
Call graph



Jump Functions



IP Value Graph



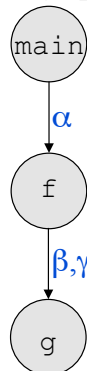
Exercise: Compute Interprocedural Value Graph

```
main()
{
  int x;
 $\alpha$ : f(x,1);
}
```

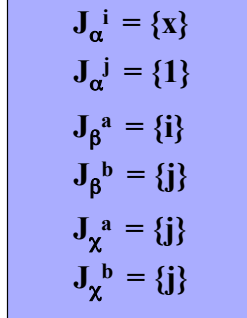
```
void f (int i, j)
{
 $\beta$ : g(i, j);
 $\gamma$ : g(j, j);
}
```

```
void g(int a, b)
{
  a = 2;
  b = b + a;
}
```

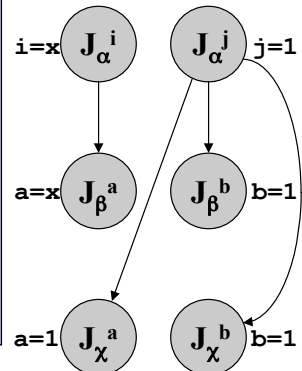
Call graph



Jump Functions



IP Value Graph



More About Jump Functions

How do we handle return values?

- Create **return jump functions** that pass values from callees to callers
- R_s^x determines the value of x upon return from procedure p
- Add nodes and edges for R_s^x to the interprocedural value graph

Types of jump functions

- Jump functions “determine the value of a parameter”
- How should they determine these values?
 - Symbolic interpretation
 - Conditional constants
 - Simple constants
- Simple constants seem to do well

Alternative to Interprocedural Analysis: Inlining

Idea

- Replace call with procedure body

Pros

- Reduces call overhead
- Exposes calling context to procedure body
- Exposes side effects of procedure to caller
- Simple!

Cons

- Code bloat (decrease efficacy of caches, branch predictor, etc)
- Can't always statically determine callee (*e.g.*, in OO languages)
- Library source is usually unavailable
- Can't always inline (recursion)

Inlining Policies

The hard question

- How do we decide which calls to inline?

Many possible heuristics

- Only inline small functions
- Let the programmer decide using an `inline` directive
- Use a code expansion budget [Ayers, et al '97]
- Use profiling or instrumentation to identify hot paths—inline along the hot paths [Chang, et al '92]
 - JIT compilers do this
- Use inlining trials for object oriented languages [Dean & Chambers '94]
 - Keep a database of functions, their parameter types, and the benefit of inlining
 - Keeps track of *indirect* benefit of inlining
 - Effective in an incrementally compiled language

} Oblivious to callsite

Alternative to Interprocedural Analysis: Cloning

Procedure Cloning/Specialization

- Create a customized version of procedure for particular call sites
- *Compromise* between inlining and interprocedural optimization

Pros

- Less code bloat than inlining
- Recursion is not an issue (as compared to inlining)
- Better caller/callee optimization potential (versus interprocedural analysis)

Cons

- Still some code bloat (versus interprocedural analysis)
- May have to do interprocedural analysis anyway
 - *e.g.* Interprocedural constant propagation can guide cloning

Evaluation

Many compilers avoid interprocedural analysis

- It's expensive and complex
- Not beneficial for most classical optimizations
- Separate compilation + interprocedural analysis requires [recompilation analysis](#) [Burke and Torczon'93]
- Can't analyze library code

When is it useful?

- Pointer analysis
- Constant propagation
- Object oriented class analysis
- Security and error checking
- Program understanding and re-factoring
- Code compaction
- Parallelization

} **Modern uses of compilers**

Other Trends

Cost of procedures is growing

- More of them and they're smaller (OO languages)
- Modern machines demand precise information (memory op aliasing)

Cost of inlining is growing

- Code bloat degrades efficacy of many modern structures
- Procedures are being used more extensively

Programs are becoming larger

Cost of interprocedural analysis is shrinking

- Faster machines
- Better methods

Concepts

Call graphs, invocation graphs

Analysis versus optimization

Characteristic of interprocedural analysis

- Flow sensitivity, context sensitivity, path sensitivity
- Smearing

Approaches

- Context sensitive, supergraph, summaries
- Bottom-up, top-down, bi-directional, iterative

Propagation versus side-effect problems

Alternatives to interprocedural analysis

- Inlining
- Procedure cloning

Next Time

Lecture

- Pointer analysis
- Look at pointer analysis as an important special case