Inter-procedural Analysis

Lecture 6: CPEN 400P

Karthik Pattabiraman, UBC

Outline

Why do we need inter-procedural(IP) analysis ?

Challenges in IP analysis

Call Graph construction

Interprocedural Constant Propagation

Motivation

Why consider analysis & optimization of whole programs?

1. **To produce better code around call sites**
   - **Avoid saves & restores**
   - **Understand cross-call data flow**
2. **To produce tailored copies of procedures**
   - **Often, full generality is unneeded**
   - **Constant-valued parameters & globals, aliases**
3. **To provide sharper global analysis**
   - **Improve on conservative assumptions**
   - **Particularly true for global variables**

Inter-procedural Analysis

Consider the following program. Let's say we want to perform const propagation.

```
int x = 5;

foo(&x);

y = x + 2;
```

Without knowing anything about what foo does, it's not possible to conclude whether or not y is a constant.

Function calls can also have side-effects (affecting global vars) and return values

What’s Interprocedural Analysis ?

Analyzing the program across procedure boundaries and functions

Running analysis on the **whole** program - more precise analysis results

Why not just create a single large graph and analyze it all ?

- Too expensive to do in practice
- Too many low-level details
- Doesn’t work where we don’t know calls in advance (e.g., recursive functions)
- Can be effective for some special cases though

Outline

Why do we need inter-procedural(IP) analysis ?

Challenges in IP analysis

Call Graph construction

Interprocedural Constant Propagation

Challenges in IP analysis

What happens at a procedure call?

- **Use worst case assumptions about side effects**
- **Leads to imprecise intraprocedural information**
- **Leads to explosion in intraprocedural def-use chains**

What are the problems?

```
procedure joe(i,j,k)
        l ← 2 * k
        if (j = 100)
                then m ← 10 * j
                else m ← i
        call ralph(l,m,k)
        o ← m * 2
        q ← 2
        call ralph(o,q,k)
        write q, m, o, l
```

```
procedure main
        call joe( 10, 100, 1000)

procedure ralph(a,b,c)
        b ← a * c / 2000
```

Since j = 100 this always executes the then clause

and always m has the value 1000

What value is printed for q?
Did ralph() change it?

With perfect knowledge, the compiler could replace this with write 1000, 1000, 2000, 2000, and the rest is dead !

*

Interprocedural Analysis

The compiler needs to understand call sites

- **Limit loss of information at calls**
- **Shrink intraprocedural data structures**
  - **Def-use chains in PFC**
- **Solve simple intraprocedural problems**

Interprocedural effects limit intraprocedural analysis

- **Grove & Torczon showed major impact of call sites on SCCP**
  - **Each call site killed many potential constants**
- **Knowledge about modifications eliminated most of it**

Definitions

- **May** problems describe events that might happen in a call
  - ***May Modify*** **sets include any name the call might define**
  - ***May Reference*** **sets include any name the call might use**

- **Must** problems describe events that always happen in a call
  - ***Must Modify*** **set describes KILLs**

Outline

Why do we need inter-procedural(IP) analysis ?

Challenges in IP analysis

Sensitivity of Analysis

Call Graph construction

Interprocedural Constant Propagation

Different Kinds of Sensitivity

Analysis can trade-off precision for cost depending on the types of sensitivity

1. Flow-sensitive
2. Path sensitive
3. Context sensitive

Flow-Sensitive Analysis

Takes into account the order of execution of program statements. Consider:

```
x = 1;

…

x = 2;
```

Is the value of x a constant (assume non-SSA form) ?

It depends on the analysis. Flow-sensitive: Yes (it can be 1 or 2)

Flow-insensitive: No (Meet of 1 and 2 is BOT)

Path-Sensitive Analysis

Path-sensitive: Takes into account specific paths in the CFG for analysis. Consider

```
If (...)
        x = 1;
else
        x = 2;
```

Is x a constant ? Path-sensitive analysis will say ‘Yes’, Path-insensitive says ‘No’

Number of paths in CFG can be exponential with no. of conditional branches

Context-Sensitive Analysis

Context-sensitive: Takes individual call-sites into account separately. Consider

x = foo(1)

x = foo(2)

int foo(x) { return x; }

Is x a constant ? It depends. Context-sensitive: yes, context-insensitive: no

- Context-sensitive will consider each call-site separately
- Context-sensitive will need to reanalyze the caller for each site (expensive !)

Class Activity

Consider the program below. What'd be the set of paths in the program considered by a context-sensitive analysis Vs a context insensitive analysis ?

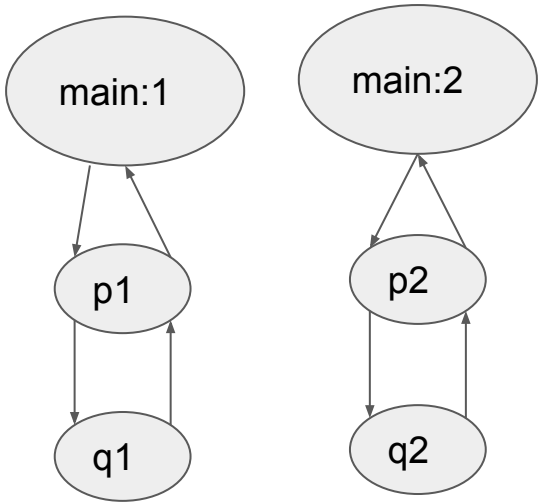
```
main() {
        1: p(7);
        2: p(42);
}

p(int n) {
        3: q(n);
}

q(int k) {
        return k;
}
```

Class Activity

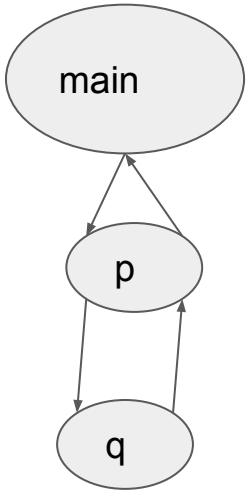
Consider the program below. What'd be the set of paths in the program considered by a context-sensitive analysis Vs a context insensitive analysis ?

```
main() {
        1: p(7);
        2: p(42);
}

p(int n) {
        3: q(n);
}

q(int k) {
        return k;
}
```

Context-sensitive



Context-insensitive



Outline

Why do we need inter-procedural(IP) analysis ?

Challenges in IP analysis

Sensitivity of Analysis

Call Graph construction

Interprocedural Constant Propagation

Call Graph

Represents the call of functions in the program

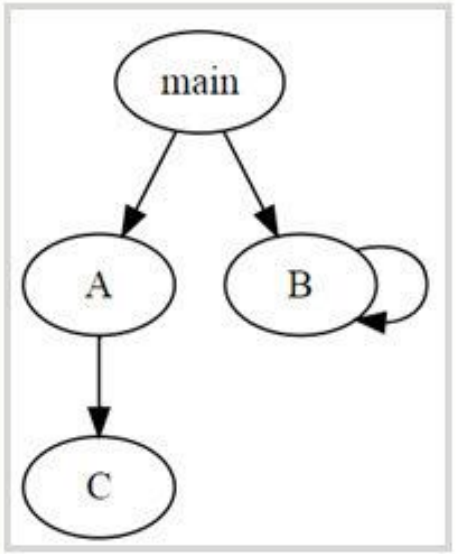
- Nodes are the functions in the program
- Edges go from call sites to functions
- If multiple calls are possible, they’re represented as parallel edges
- Recursive calls are represented as self-loops
- Typically **flow-insensitive (no execution order)**

Static call graph has all **possible function calls in the program** (Conservative)

- May lead to false-positives and over-approximations in the analysis

Call graph Example (Java)

```
class Main {
  public static void main(String[] args) {
    A();
    B();
  }
  public static int A(){
    C();
  }
  public static void B(){
      B();
  }
}
```



Source: [GraphEvo: Characterizing and Understanding Software Evolution using Call Graphs]

Constructing the Call Graph

Solution: Ryder, 1979 (non-recursive Fortran)

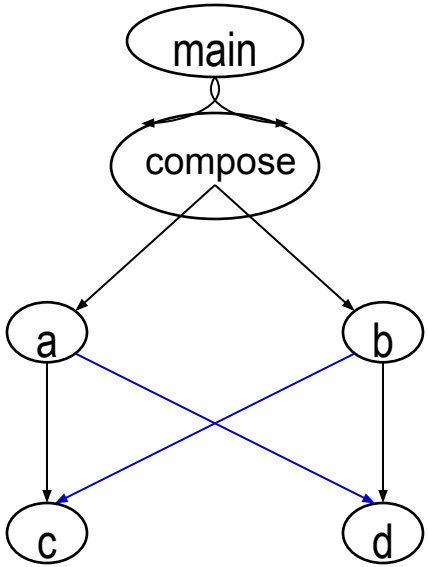
- **Build subgraph described by literal constants**
- **Propagate sets of values for procedure variables**
- **Complexity is linear in size of call graph**

Procedure-valued variables complicate the process

- **Must track values of variables (constant propagation)**
  - **Typically (& fortunately) no arithmetic**
- **Results can be approximate (overestimate) or precise**

Constructing the Call Graph

```
procedure main
        call compose(a,c)
        call compose(b,d)

procedure compose(x,y)
        call x(y)

procedure a(z)
        call z()

procedure b(z)
        call z()

procedure c()
        …

procedure d()
        …
```



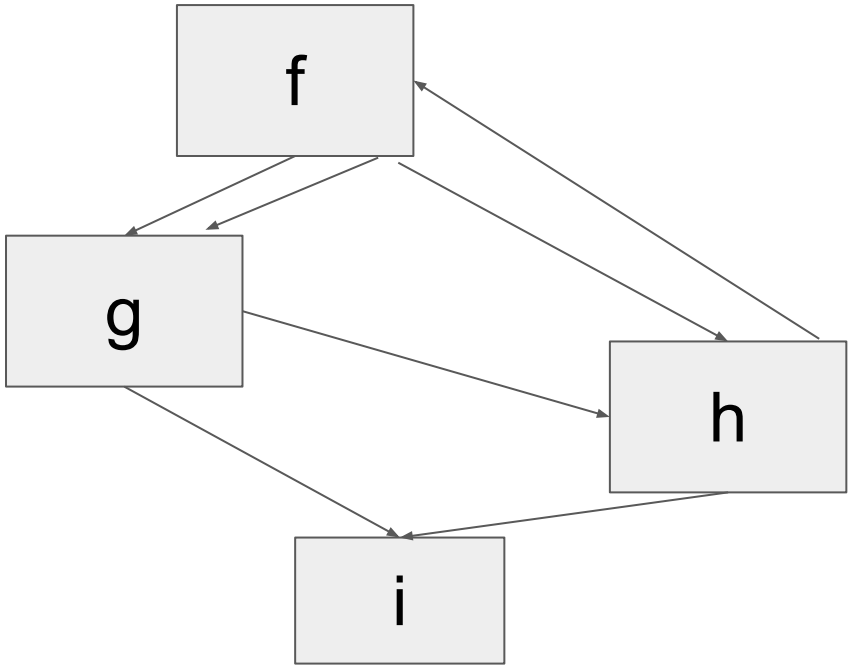
Imprecise call graph

Class Activity: Draw the call-graph for the code below

```
f() {
    1: g();
    2: g();
    3: h();
}

g() {
   4: h();
   5: i();
}

h() {
    6: f();
    7: i();
}

 i() { … }
```

Class Activity: Draw the call-graph for the code below

```
f() {
    1: g();
    2: g();
    3: h();
}

g() {
   4: h();
   5: i();
}

h() {
    6: f();
    7: i();
}

 i() { … }
```



Outline

Why do we need inter-procedural(IP) analysis ?

Challenges in IP analysis

Sensitivity of Analysis

Call Graph construction

Interprocedural Constant Propagation

What’s the main idea ?

Need to track whether or not something is a constant across **function calls**

Strawman approach:

- Duplicate the CFG of the called function inside the calling function (inlining)
- Too expensive in practice; doesn’t work with recursive functions

Instead, we construct the call graph of the program and use **jump and support** functions to abstract the effect of each function call on its formal parameters

- Simply apply the jump function rather than reanalyze the function each time
- May require multiple passes through the call graph for it to converge

Formal definition of Jump function

Jump function models the set of formal parameters passed to a function and whether they are constant (i.e., their value in the Const lattice): J_s^X

At each call site, there's a vector of support functions mapping each argument to a formal parameter of the callee, say (a, b, c, ..)

$J_s = [\ J_s^a\ J_s^b\ J_s^c\ ]$

Support functions are used to denote the mapping of the formal parameters to the variables of the function containing s: $Support(J_s^X)$

$Support(J_s^X)$ = T if Value(y) = T for any y ϵ $Support(J_s^X)$

Algorithm Sketch

Keep track of the value of each formal parameter x of a procedure p in *Value(x)*

Initialization Phase

- Set the values of all fields to T
- Iterate over each actual parameter *a* at each call-site
- Update the *Value* field of the *a*'s formal parameter *f* by Value(f) $\wedge$ J_s^F
- Add f to the Worklist

Second Phase

- Repeatedly select a formal parameter from the worklist 'x' and propagate it
- If there's any change in *Value(x)* then add x to the Worklist

Algorithm

```
// Phase 1: Initializations
Build all jump functions and Support mappings
Worklist ← ∅

for each procedure p in the program
    for each formal parameter f to p
        Value(f) ← ⊤                          // Optimistic initial value
        Worklist ← Worklist ∪ {f}

for each call site s in the program
    for each formal parameter f that receives a value at s
        Value(f) ← Value(f) ∧ J_s^f           // Initial constants factor in to J_s^f

// Phase 2: Iterate to a fixed point
while (Worklist ≠ ∅)
   pick parameter f from Worklist             // Pick an arbitrary parameter
   let p be the procedure declaring f

   // Update the Value of each parameter that depends on f
   for each call site s in p and parameter x such that f ∈ Support(J_s^x)
       t ← Value(x)
       Value(x) ← Value(x) ∧ J_s^x            // Compute new value
       if (Value(x) < t)
           then Worklist ← Worklist ∪ {x}

// Post-process Val sets to produce CONSTANTS
for each procedure p
    CONSTANTS(p) ← ∅
    for each formal parameter f to p
        if (Value(f) = ⊤)
            then Value(f) ← ⊥

        if (Value(f) ≠ ⊥)
            then CONSTANTS(p) ← CONSTANTS(p) ∪ {(f, Value(f))}
```

Outline

Why do we need inter-procedural(IP) analysis ?

Challenges in IP analysis

Sensitivity of Analysis

Call Graph construction

Interprocedural Constant Propagation