Interprocedural Analysis

**Last time**
- Introduction to alias analysis

**Today**
- Interprocedural analysis

---

Motivation

**Procedural abstraction**
- Cornerstone of programming
- Introduces barriers to analysis

**Example**

```c
void f(int x)
{
    if (x)
        foo();
    else
        bar();
}

f(0);
f(1);
```

---

**Example**

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

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

Does \( foo() \) modify \( x \)?

---

March 4, 2015 Interprocedural Analysis

March 4, 2015 Interprocedural Analysis
Function Calls and Pointers

Recall
- Function calls can affect our points-to sets
e.g.,

```c
p1 = &x;
p2 = &p1;
...
foo();
```

Be conservative
- Lose a lot of information

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`
- `{b = c;`
- `main()
  {`
- `int d;
  // Local variables`
- `foo(a, d);
  // Actual parameters`
- `}`

March 4, 2015 Interprocedural Analysis 3

March 4, 2015 Interprocedural Analysis 4
Interprocedural Analysis vs. Interprocedural Optimization

**Interprocedural analysis**
- Gather information across multiple procedures (typically across the entire program)
- Use this information to improve intra-procedural 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
- Faster but less accurate than flow-sensitive analysis

Flow Sensitivity Example

**Is x constant?**

```c
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

**Where have we seen examples of flow-insensitive analysis?**
- Address Taken pointer analysis
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

---

**Context Sensitivity Example**

Is $x$ constant?

**Context-sensitive analysis**
- Computes an answer for every callsite:
  - $x$ is 4 in the first call
  - $x$ is 5 in the second call

**Context-insensitive analysis**
- Computes one answer for all callsites:
  - $x$ is not constant
- Suffers from unrealizable paths:
  - Can mistakenly conclude that $\text{id}(4)$ can return 5 because we merge (smear) information from all callsites

```plaintext
a = id(4); b = id(5);

id(x) { return x; }
```

---

```plaintext
a = id(4); b = id(5);

id(x) { return x; }
```
**Path Sensitivity**

**Path-sensitive analysis**
- Computes one answer for every execution path
- Subsumes flow-sensitivity and context-sensitivity
- Extremely expensive

**Path-insensitive**
- Not path-sensitive

---

**Path Sensitivity Example**

Is `x` constant?

```
if (x==0)
    x = 4;
    x = 5;
print(x)
4 5
```

**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 paths:
  - `x` is not constant
Dimensions of Interprocedural Analysis (cont)

Flow-insensitive context-insensitive (FICI)

```c
int** foo(int **p, **q)
{
    int **x;
    x = p;               p →
    ...                   q →
    x = q;               x →
    return x;
}
```

```c
int main()
{
    int **a, *b, *d, *f,
        c, e;
    d → {c, e}
    a = foo(&b, &f);
    f → {c, e}
    *a = &c;
    a = foo(&d, &g);
    g → {c, e}
    *a = &e;
}
```

Dimensions of Interprocedural Analysis (cont)

Flow-sensitive context-insensitive (FSCI)

```c
int** foo(int **p, **q)
{
    int **x;
    x = p;               p → {b, d}
    ...                   q → {f, g}
    x = q;               x → {b, d, f, g} x_1 →
    return x;
}
```

```c
int main()
{
    int **a, *b, *d, *f,
        c, e;
    d → {c, e} f_1 →
    a = foo(&b, &f);
    f → {c, e} g_1 →
    *a = &c;
    a = foo(&d, &g);
    g → {c, e}
    Weak update f_2 →
    *a = &e;
}
```
**Interprocedural Analysis: Supergraphs**

**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**

**Pros**
- Simple
- Intraprocedural analysis algorithms work unchanged
- Reasonably effective

---

**Monday’s Example Revisited**

```c
{ 
    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 (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

Cons
– Accuracy? Smears information from different contexts.
– Performance? IDFA is $O(n^4)$, graphs can be huge
– No separate compilation IDFA converges in $d+2$ iterations, where $d$ is the Number of nested loops [Kam & Ullman ’76]. Graphs will have many cycles (one per callsite)

Brute Force: Full Context-Sensitive Interprocedural Analysis

Invocation Graph [Emami’94]
– Use an invocation graph, which distinguishes all calling chains
– Re-analyze callee for each distinct calling paths

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

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

Goal
- Represent procedure call relationships

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 there’s 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?
- Useful for optimizing the body of a procedure

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?
- Useful for optimizing the code that calls a procedure

Examples

Propagation Summaries
- May-Alias: The set of formals that may be aliased to globals and to each other
- Must-Alias: The set of formals that are definitely aliased to globals and to each other
- Constant: The set of formals that have constant value

Side-effect Summaries
- Mod: The set of variables possibly modified (defined) by a call to a procedure
- Ref: The set of variables possibly read 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

```c
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

```c
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
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: Affects alias analysis, which in turn affects side-effects

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

Bi-Directional Interprocedural Summaries

**Interprocedural Constant Propagation (ICP)**
- Information flows from caller to callee and back

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
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 (decreases the 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
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

Most 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 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
- Flow-insensitive analysis
- Look at pointer analysis as an important special case