Interprocedural Analysis

CIS410/510 Program Analysis and Transformation

Interprocedural analysis

- Last time
  - Alias (pointer) analysis
  - Single-function scope

- Today
  - Introduction to interprocedural analysis
Function calls

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

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

- Does the function call modify x?
  - With our intra-procedural analysis so far, we don’t know
  - Make worst case assumptions
    - Assume that any reachable pointer may be changed
    - Pointers can be “reached” via globals and parameters
      - May pass through objects in the heap
Representations of aliasing

- Points-to pairs [Emami94]
  - Pairs where the first member points to the second
    e.g., (a -> b), (b -> c)

- Alias pairs [Shapiro & Horwitz 97]
  - Pairs that refer to the same memory
    e.g., (*a, b), (*b, c), (**a, c)
  - Completely general
  - More concise than points-to pairs

- Equivalence sets
  - All memory references in the same set are aliases
    e.g., {*a, b}, {*b, c, **a}

---

**Introduction to interprocedural analysis**

- Procedural abstraction
  - Key to structured programming
  - Makes analysis harder

**Example:**

```c
int **a, *b, c, *d;
1: a = &b;
2: b = &c;
```

**Example:**

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

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

---

```c
What is the calling context of f()?
```
Function calls and pointers

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

\[
\begin{align*}
p1 &= \&x; \\
p2 &= \&p1; \\
\ldots \quad &\{(p1->x),(p2->p1)\} \\
\text{foo}(); &\quad \text{??}
\end{align*}
\]

- Be conservative \(\rightarrow\) lose a lot of information

Interprocedural analysis

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

Terminology:

```c
int a, e; \quad // Globals
void foo(int &b, int &c) \quad // Formal parameters (passed
{ \quad // by reference)
  b = c;
}
main()
{
  int d; \quad // Local variables
  foo(a, d); \quad // 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 optimizations (e.g., common subexpression elimination)

- Interprocedural optimizations
  - Optimization that involve multiple procedures, e.g., inlining, procedure cloning, interprocedural register allocation
  - Optimizations that use interprocedural analysis

Types 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
  - Required iterative dataflow analysis or similar techniques

- Flow-insensitive analysis
  - Ignores control flow, assume statements can execute in any order
  - Computes one answer for the whole program
  - Can compute in linear time
  - Less accurate than flow-sensitive

Flow sensitivity example

Is \( x \) constant?

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

- Flow-sensitive analysis
  - Computes one answer for every program point
    - \( x \) is 4 after the first assignment
    - \( x \) is 5 after the second assignment

- Flow-insensitive analysis
  - Computes one answer for entire program
    - \( x \) is not constant

- What examples of flow-insensitive analysis have you already seen?
  - “Address taken” pointer analysis
Context sensitivity

- Context-sensitive analysis
  - Reanalyzes 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**, i.e., can mistakenly conclude that \( \text{id}(4) \) can return 5 because we merge **smear** information from all callsites
Path sensitivity

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

- Path-insensitive
  - Not path-sensitive (same as flow-insensitive)

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 paths:
  - $x$ is not constant
Interprocedural analysis: Supergraphs

- Compose the CFGs for all procedures via the `call graph`
  - Connect call nodes to `entry` nodes of the callees
  - Connect `return` nodes of callees back to calls
  - Called control-flow supergraph

- Advantages
  - Simple
  - Intraprocedural algorithms work unchanged
  - Reasonably effective

Earlier example revisited

```c
int x, y, a;
int *p;

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

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

- Is `x` constant?
  - With a supergraph, run the same IDFA (iterative dataflow analysis) algorithm
  - Determine that `x=5`
Supergraphs (cont.)

- Compose the CFGs for all procedures via the call graph
  - Connect call nodes to entry nodes of the callees
  - Connect return nodes of callees back to calls
  - Called control-flow supergraph

- Disadvantages
  - Accuracy
  - Performance
  - No separate compilation

  Smears information from different contexts.
  IDFA is $O(n^4)$, graphs can be huge
  IDFA converges in 2 iterations, where $d$ is the number of nested loops [Kam & Ullman '76].
  Graphs will have many cycles (one per callsite).

Brute force approach: 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

```c
void foo (int b) {
    hoo(b);
}
void goo (int c) {
    hoo(c);
}
int 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. procedure i()
15. procedure j()
16. begin
17. call g()
18. call j()
19. end
20. end

Goal
- Represent procedure call relationships

Definition
- If program consists of n procedures \( p_1, \ldots, p_n \)
- Static call graph of \( P \) is \( G_p = (N, S, E, r) \)
  - \( N = \{p_1, \ldots, p_n\} \)
  - \( S = \{\text{callsite labels}\} \)
  - \( E \subseteq N \cdot N \cdot S \)
  - \( r \in N \) is the start node

Interprocedural analysis: Summaries

- First introduced by Sharir and Pnueli in 1978
  - Take a read if you want to see what a type-written CS paper looked like!
- 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 possible (and practical)
- Cons
  - Imprecise if only have one summary per procedure
Summaries (cont.)

What is a summary?
- Think of it as a succinct representation of the behavior of the procedure that is also parameterized by any information about its input variables

- Each procedure is analyzed once (or a few times in the case of recursive procedures) to build its summary

- Different approaches (less or more general) for constructing summaries exist

Two types of information

- Track information that flows into a procedure
  - Sometimes known as *propagation problems*, e.g.,
    - What formal parameters are constant?
    - Which formal parameters are aliased to globals?

- Track information that flows out of a procedure
  - Sometimes known as *side effect problems*, e.g.,
    - Which globals are defined/used by a procedure?
    - Which locals are defined/used by a procedure?
    - Which actual parameters are defined by a procedure?

```c
proc (x,y)
{
  ...
}
```
Examples

- **Propagation summaries**
  - MAY-ALIAS: the set of formal parameters that may be aliased to globals and each other
  - MUST-ALIAS: the set of formal parameters that are definitely aliased to globals and each other
  - CONSTANT: the set of formal parameters 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 (e.g., MAY-ALIAS, MUST-ALIAS)
  - Use this information inside the procedure body
    ```c
    int a;
    void foo (int &b, int &c) {
      ...
    }
    foo(a, a);
    ```

- **Bottom-up**
  - Summarize the effects of a call (MOD, REF, KILL)
  - Use this information around procedure calls
    ```c
    x = 8;
    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 dataflow information with callsite
  - Side effects: Affects alias analysis which in turn affects side effects

- **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
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
  - 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 (eliminates) call overhead
  - exposes calling context to procedure body
  - exposes side effects of procedure to caller
  - simple!

- Cons
  - Code bloat (instruction caching, branch predictor, etc.)
  - Can’t always statically determine callee
    - Runtime binding of virtual function calls in OO languages
    - Function pointers in C
  - 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 (e.g., using inline directive)
  - Use profiling or instrumentation to identify hot paths—inline along the hot paths [Chang, et al ’92]
    - JIT compilers do this
  - Use a code expansion budget [Ayers, et al ’97]
  - 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

Alternatives to interprocedural analysis: Cloning

- Procedure cloning/specialization
  - Create a customized version of procedure for particular call sites
  - A compromise between inlining and interprocedural optimizations

- Pros
  - Less code bloat (than inlining)
  - Recursion is not a problem (compared to inlining)
  - Better caller/callee optimization potential (vs interprocedural analysis)

- Cons
  - Still some code bloat (vs. 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
  - Separate compilation + interprocedural analysis requires *recompilation analysis*
  - can’t analyze library code

- When is it useful?
  - Pointer analysis
  - Constant propagation
  - Parallelization
  - Code compaction
  - OO class analysis
  - Program understanding and refactoring
  - Security and error checking
  - ...

- Some compilers increasingly support interprocedural optimizations (and analysis)
  - GCC, Intel (-ip for single file, -ipo for all files), Microsoft C, LLVM (Clang)

Other trends

- Cost of procedures is growing
  - More of them and each is smaller (OO languages)
  - On modern machines we require precise information (memory operation aliasing) to achieve good performance

- Cost of inlining is growing
  - Instruction caches are small, code bloat causes cache misses
  - Procedures are being used more extensively

- Programs are becoming larger and more complex

- Cost of interprocedural analysis is shrinking
  - faster machines
  - better methods
Concepts

- Call graphs, invocation graphs
- Analysis vs optimization
- Characteristics of interprocedural analysis
  - Flow sensitivity, context sensitivity, path sensitivity
  - Smearing of dataflow information
- Approaches
  - Context-sensitive, supergraph, summaries
  - Bottom-up, top-down, bi-directional, iterative
- Propagation vs side-effect problems
- Alternatives to interprocedural analysis
  - inlining
  - procedure cloning