CSC2/455 Software Analysis and Improvement
Interprocedural Analyses - II

Sreepathi Pai

URCS

April 10, 2019

Outline

Outline

Strategies for handling effects of functions

Can we reuse our CFG-based analyses to handle function calls?

- Inlining, perhaps?
- What are the limitations of inlining?

Call graphs

- Basic graph structure for analysis
  - Nodes for each procedure (and/or call site)
  - Edge connect nodes from caller to callee
- Various definitions possible
  - Your three textbooks have three different definitions
- Primary Purpose
  - Capture how control flows at the procedure level

Context-insensitive analysis

- Treat function calls as control flow
  - Functions form node in CFG
  - Invocations are treated as "`gotos`" to entry of function
  - The `returns` as `goto` to location after function call
- This results in CFG with:
  - multi-entry nodes
  - multi-exit nodes that are not branches (example in a few slides)

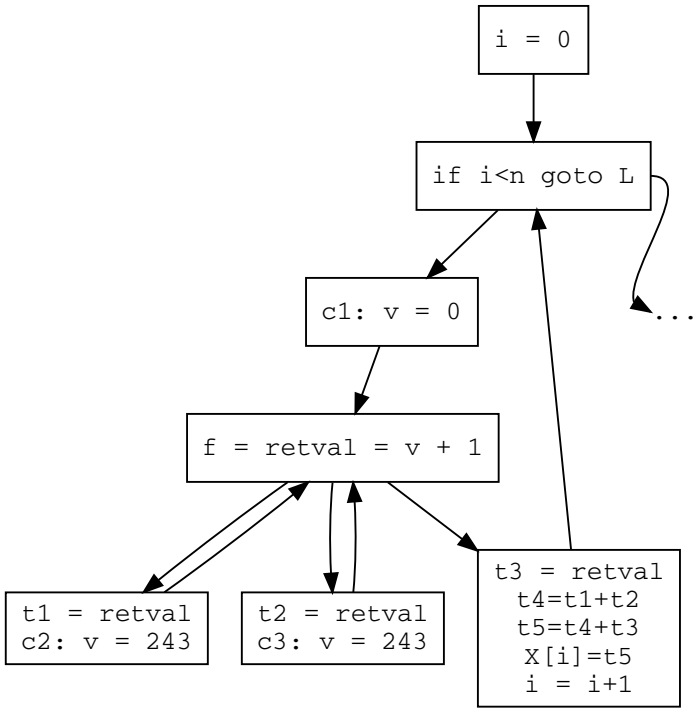
Example

```
   for(i = 0; i < n; i++) {
c1:       t1 = f(0);
c2:       t2 = f(243);
c3:       t3 = f(243);
          X[i] = t1 + t2 + t3;
   }

   int f(int v) {
       return (v+1);
   }
```

Is the value of X[i] constant?

CFG for Example



- values for:
  - `v` into `f`
  - `retval` out of `f`
  - `t1, t2 and t3`?
  - `X[i]`?

Context-sensitivity

- In treating function calls as control-flow, we lost the ability to detect *context*
- Context is the call stack for the function
  - Can be treated as a string, called a call string
  - If `c1` calls `c2`, which then calls `c3`, the context for `c3` is `(c1, c2)`
- How many contexts can there be?
  - consider indirect function calls
  - consider recursive functions
- *k*-limiting context sensitivity
  - Limit context to *k* immediate call *sites* (important!)
  - 0 is context-insensitive

Cloning-based Context-Sensitive Analysis

```
    for(i = 0; i < n; i++) {
c1:     t1 = f1(0);
c2:     t2 = f2(243);
c3:     t3 = f3(243);
        X[i] = t1 + t2 + t3;
    }

    int f1(int v) {
        return (v+1);
    }

    int f2(int v) {
        return (v+1);
    }

    int f3(int v) {
        return (v+1);
    }
```

- Create a clone for each unique calling context and then apply context-insensitive analysis
- Is this the same as inlining?
  - See textbook for a differentiating example

k-level Context-Sensitive Analysis

```
   for(i = 0; i < n; i++) {
c1:      t1 = g(0);
c2:      t2 = g(243);
c3:      t3 = g(243);
         X[i] = t1 + t2 + t3;
   }

   int g(int v) {
       if(v > 1)
          return f(v);
       else
          return (v+1);
   }

   int f(int v) {
       return (v+2);
   }
```

To what depth shall we clone functions?

k-level Context-sensitive analysis

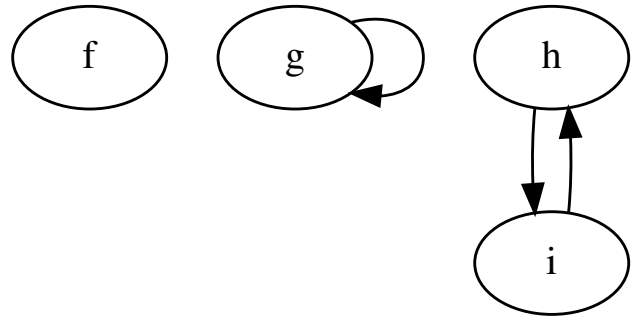
- A function call may be distinguished by its context
  - Calling functions or
  - Call-sites (i.e. call stack)
- If we do not distinguish contexts,
  - context-insensitive
  - $k = 0$
- Different values of k may yield different precision
- No value of k may be sufficient
  - recursive function calls
  - indirect function calls

Some numbers

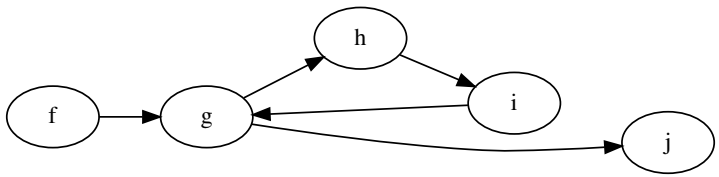
- If there are N functions in a program, how many calling contexts are possible
  - if no recursion is involved?
  - if recursion is involved?

Handling Recursion in Contexts

- Consider nodes in a call graph
  - non-recursive functions
  - self-recursive functions
  - mutually recursive functions
- Look for strongly-connected components
  - trivial (non-recursive)
  - non-trivial (the latter two)



Methods to “finitize” Recursion



- Model them using regular expressions
  - `f(g h i)*j`
- Eliminate all call information within SCC
  - `f g j`

Have contexts, will analyze!

- Cloning-based analysis
  - Clone functions, once per context
  - Followed by context-insensitive analysis
- Summary-based analysis
  - (Bottom-up phase) Compute summaries of each function for an analysis (e.g. constant propagation) in terms of input parameters
  - (Top-down phase) Pass inputs to summaries, one per context OR merge contexts using meet operator
  - See textbook for details

Dynamic Call Graph Construction

```
class t {
    t n() { return new r(); }  /* call site g */
}

class s extends t {
    t n() { return new s(); }  /* call site h */
}

class r extends s {
    t n() { return new r(); }  /* call site i */
}

main() {
    t a = new t();             /* call site j */
    a = a.n();
}
```

What is a potential call graph for `a.n()` from the points-to relationships?

Outline

Recall

Recall how we compute and update pointsTo sets from last class...

Flavours

- Flow-sensitive/Flow-insensitive
- Context-insensitive
- Context-sensitive
  - Cloning-based
  - Summary-based

What the textbook describes

- Flow-insensitive
- Context-sensitive
  - With non-trivial SCCs treated as a single node
- Cloning-based

Additionally, the Dragon book formulates the points-to analysis as a (datalog) logical formula to be solved.

Outline

References

- Chapter 12 of the Dragon Book
- Paper recommended:
  - Reps et al. "Precise interprocedural dataflow analysis via graph reachability"