Interprocedural Analysis

Yu Zhang

Most content comes from http://cs.au.dk/~amoeller/spa/

1

Interprocedural Analysis

- Analyzing the body of a single function
  - ***intra***procedural analysis
- Analyzing the whole program with function calls
  - ***inter***procedural analysis
- For now, we consider TIP without function pointers and indirect calls
- A naive approach:
  - analyze each function in isolation
  - be maximally pessimistic about results of function calls
  - rarely sufficient precision…

2

CFG for Whole Programs

The idea:
- Construct a CFG for each function
- Then glue them together to reflect function calls and returns

We need to take care of:
- parameter passing
- return values
- values of local variables across calls (including recursive functions, so not enough to assume unique variable names)

3

A Simplifying Assumption

- Assume that all function calls are of the form

$X = f(E_1, \ldots, E_n);$

- This can always be obtained by normalization

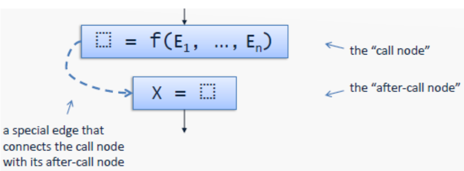
4

Interprocedural CFGs (1/3)

Split each original call node

```
X = f(E1, …, En)
```

into two nodes:



5

Interprocedural CFGs (2/3)

Change each return node

```
return E
```

into an assignment:

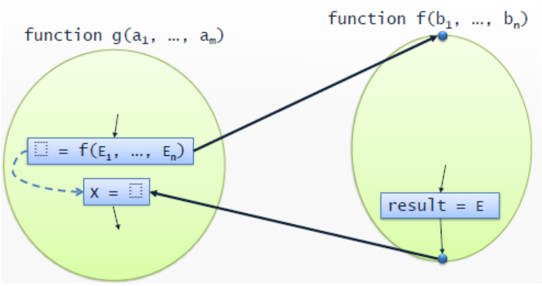
```
result = E
```

(where result is a fresh variable)

6

Interprocedural CFGs (3/3)

Add call edges and return edges:

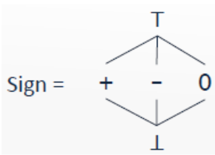


Constraints

- For call/entry nodes:
  - be careful to model evaluation of *all* the actual parameters before binding them to the formal parameter names (otherwise, it may fail for recursive functions)
- For after-call/exit nodes:
  - like an assignment: X = **result**
  - but also restore local variables from before the call using the call↷after-call edge
- The details depend on the specific analysis…

Example: Interprocedural Sign Analysis

- Recall the intraprocedural sign analysis…
- Lattice for abstract values:



- Lattice for abstract states:

  Vars→*Sign*

Example: Interprocedural Sign Analysis

- Constraint for entry node v of function f($b_1, \ldots, b_n$):

$$[\![v]\!] = \bigsqcup_{w\in pred(v)} \bot[b_1\rightarrow eval([\![w]\!],E_1^w), \ldots, b_n\rightarrow eval([\![w]\!],E_n^w)]$$

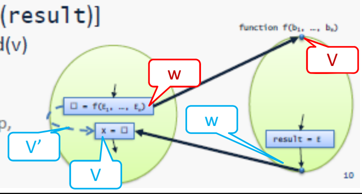
where E_i^w is i'th argument at w

- Constraint for after-call node v labeled $X = □$, with call node v':

$$[\![v]\!] = [\![v']\!][X\rightarrow[\![w]\!](\mathtt{result})]$$

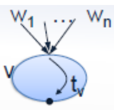
where w∈pred(v)

(Recall: no global variables, no heap, and no higher-order functions)



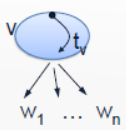
Alternative Formulations

1) $[\![v]\!] = t_v(\bigsqcup_{w\in pred(v)} [\![w]\!])$

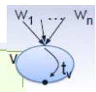


2) $\forall w\in succ(v)$: $t_v([\![v]\!]) \sqsubseteq [\![w]\!]$

   – recall "solving inequations"
   – may require fewer join operations if there are many CFG edges
   – more suitable for *inter*procedural flow



The Worklist Algorithm (original version)

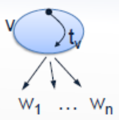



```
x1 = ⊥; ... xn = ⊥
W = {v1, ..., vn}
while (W≠∅) {
  vi = W.removeNext()
  y = fi(x1, ..., xn)
  if (y≠xi) {
    for (vj ∈ dep(vi)) {
      W.add(vj)
    }
    xi = y
  }
}
```

如果CFG结点vi 的语义值发生变化，则将计算vi语义值所依赖的结点vj加入到worklist

The Worklist Algorithm (alternative version)

```
x1 = ⊥; ... xn = ⊥
W = {v1, ..., vn}
while (W≠∅) {
  vi = W.removeNext()
  y = ti(xi)
  for (vj ∈ dep(vi)) {
    propagate(y,vj)
  }
}
```

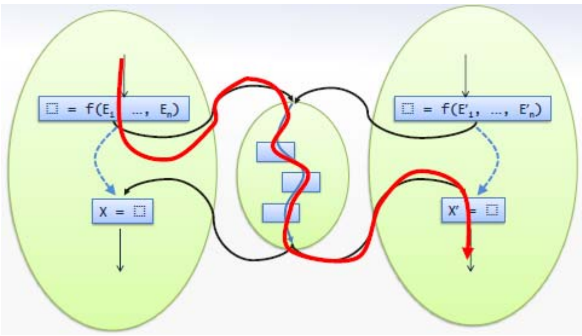



```
propagate(y,vj) {
  z = xj ⊔ y
  if (z≠xj) {
    xj = z
    W.add(vj)
  }
}
```

Implementation: WorklistFixpointPropagationSolver

- Interprocedural analysis
- **Context-sensitive interprocedural analysis**

Interprocedurally Invalid Paths



Example

What is the sign of the return value of g?

```
f(z) {
  return z*42;
}

g() {
  var x,y;
  x = f(0);
  y = f(87);
  return x + y;
}
```

Our current analysis says "⊤"

Function Cloning (alternatively, function inlining)

- Clone functions such that each function has only one callee
- Can avoid interprocedurally invalid paths ☺
- For high nesting depths, gives exponential blow-up☹
- Doesn't work on (mutually) recursive functions ☹
- Use heuristics to determine when to apply (trade-off between CFG size and precision)

Example, with cloning

- What is the sign of the return value of g?

```
f1(z1) {
  return z1*42;
}

f2(z2) {
  return z2*42;
}

g() {
  var x,y;
  x = f1(0);
  y = f2(87);
  return x + y;
}
```

Context Sensitive Analysis

- Function cloning provides a kind of context sensitivity (also called poly-variant analysis)
- Instead of physically copying the function CFGs, do it *logically*
- Replace the lattice for abstract states, States, by
  Contexts → lift(States)

  where Contexts is a set of ***call contexts***
  - the contexts are abstractions of the state at function entry
  - Contexts must be finite to ensure finite height of the lattice
  - the bottom element of lift(States) represents “unreachable” contexts
- Different strategies for choosing the set Contexts…

One-level Cloning

- Let $c_1,\ldots,c_n$ be the call nodes in the program
- Define Contexts=$\{c_1,\ldots,c_n\}\cup\{\varepsilon\}$
  - each call node now defines its own “call context”(using ε to represent the call context at the main function)
  - the context is then like the return address of the top-most stack frame in the call stack
- Same effect as one-level cloning, but without actually copying the function CFGs
- Usually straightforward to generalize the constraints for a context insensitive analysis to this lattice
- (Example: context-sensitive sign analysis –later…)

The Call String Approach

- Let $c_1,\ldots,c_n$ be the call nodes in the program
- Define Contexts as the set of strings over $\{c_1,\ldots,c_n\}$ of length $\leq$k
  - such a string represents the top-most k call locations on the call stack
  - the empty string ε again represents the call context at the main function
- For k=1 this amounts to one-level cloning

Example:
interprocedural sign analysis with call strings (k=1)

Lattice for abstract states: Contexts → lift(Vars → Sign)
where Contexts=$\{\varepsilon, c_1, c_2\}$

```
f(z) {
  var t1,t2;
  t1 = z*6;
  t2 = t1*7;
  return t2;
}
...
x = f(0);  // c1
y = f(87); // c2
...
```

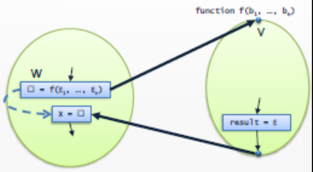
[ε ↦ unreachable,
c1 ↦ ⊥[z↦0, t1↦0, t2↦0],
c2 ↦ ⊥[z↦+, t1↦+, t2↦+]]

What is an example program that requires k=2 to avoid loss of precision?

Context Sensitivity with Call Strings
function entry nodes, for k=1

Constraint for entry node v of function $f(b_1,\ldots,b_n)$:
(if not ‘main’)

$$[\![v]\!](c) = \bigsqcup_{\substack{w\in pred(v)\ \wedge\\ c = w\ \wedge\\ c'\in Contexts}} s_w^{c'}$$

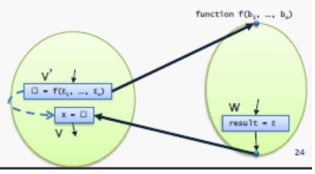


$$s_w^{c'} = \begin{cases} \text{unreachable} & \text{if } [\![w]\!](c') = \text{unreachable} \\ \bot[b_1\to eval([\![w]\!](c'),E_1^w), \ldots, b_n\to eval([\![w]\!](c'),E_n^w)] & \text{otherwise} \end{cases}$$

Context Sensitivity with Call Strings
after-call nodes, for k=1

Constraint for after-call node v labeled $X = \square$,
with call node v′ and exit node w∈pred(v):

$$[\![v]\!](c) = \begin{cases} \text{unreachable} & \text{if } [\![v']\!](c) = \text{unreachable} \vee [\![w]\!](v') = \text{unreachable} \\ [\![v']\!](c)[X\to[\![w]\!](v')(\texttt{result})] & \text{otherwise} \end{cases}$$



The Functional Approach

- The call string approach considers *control flow*
  - but why distinguish between two different call sites if their abstract states are the same?
- The functional approach instead considers *data*
- In the most general form, choose
  Contexts = States
  (requires States to be finite)
- Each element of the lattice States → lift(States) is now a map m that provides an element m(x) from States (or “unreachable”) for each possible x where x describes the state at function entry

25

Example:

interprocedural sign analysis with the functionalapproach

Lattice for abstract states: Contexts → lift(Vars → Sign)
where Contexts = Vars → Sign

```
f(z) {
  var t1,t2;
  t1 = z*6;
  t2 = t1*7;
  return t2;
}
...
x = f(0);
y = f(87);
...
```

[⊥[z↦0] ↦ ⊥[z↦0, t1↦0, t2↦0],
⊥[z↦+] ↦ ⊥[z↦+, t1↦+, t2↦+],
all other contexts ↦ unreachable]

26

The Functional Approach

- The lattice element for a function exit node is thus a ***function summary*** that maps abstract function input to abstract function output
- This can be exploited at call nodes!
- When entering a function with abstract state x:
  - consider the function summary s for that function
  - if s(x) already has been computed, use that to model the entire function body, then proceed directly to the after-call node
- Avoids the problem with interprocedurally invalid paths!
- …but may be expensive if States is large

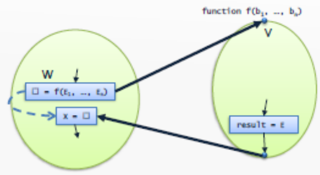
27

Context sensitivity with the functional approach

function entry nodes

Constraint for entry node v of function f(b_1, …, b_n):
(if not ‘main’)

$$[\![v]\!](c) = \bigsqcup_{\substack{w \in pred(v) \,\wedge \\ c = s_w^{c'} \,\wedge \\ c' \in Contexts}} s_w^{c'}$$



where $s_w^{c'}$ is defined as before

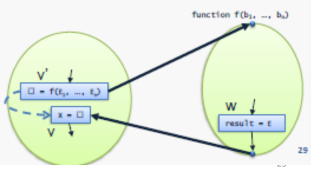
28

Context sensitivity with the functional approach

after-call nodes

Constraint for after-call node v labeled X = ⬚,
with call node v’ and exit node w∊pred(v):

$$[\![v]\!](c) = \begin{cases} \text{unreachable} & \text{if } [\![v']\!](c) = \text{unreachable} \vee [\![w]\!](s_{v'}^{c}) = \text{unreachable} \\ [\![v']\!](c)[X \mapsto [\![w]\!](s_{v'}^{c})(\texttt{result})] & \text{otherwise} \end{cases}$$



29