Interprocedural Analysis

This article is the summary of chapter 8 of `static program analysis`(by Anders Moller and Michael I. Schwartzbach)

Interprocedural CFG

For a procedural call `X = g(a1,...,a_n)`, in procedural `f` we have

- call node of `g`
- entry node of `g`
- exit node of `g`
- after-call node of `g`

```
call node of g
        \
         \
          \
            entry node of g
                |
                |
                |
            exit node of g
              /
             /
            /
after-call node of g
```

Note that the connection between the call node and the after-call node is represented by a special edge(not $succ$ or $pred$).

Context Insensitive

We take Sign Analysis as an example.

In **intraprocedural** Sign Analysis, the lattice is a map lattice: $States = Vars \rightarrow Sign$.

For **interprocedural** Sign Analysis, we can make some change.

Call Node

If v is a call node, we can view it as an no-op node, so we can define

$$[\![v]\!] = JOIN(v) = \cup_{w \in pred(v)} [\![w]\!]$$

Entry Node

If v is an entry node of a function $f(b_1, \ldots, b_n)$, then we can define

$$[\![v]\!] = \cup_{w \in pred(v)} s_w$$

where

$$s_w = \perp[b_1 \to eval([\![w]\!], E_1^w), \ldots, b_n \to eval([\![w]\!], E_n^w)]$$

$\perp$ is a map that maps every variable to the bottom element $\perp$ of $Sign$ and E_i^w is the i'th argument at the call node w.

Equivalently, for a function entry node v and every $w \in pred(v)$ as a caller node, we have

$$s_w \subseteq [\![v]\!]$$

Specially, for $main$ function, we have

$$[\![v]\!] = \perp[b_1 \to \top, \ldots, b_n \to \top]$$

since arguments $b_1, \ldots b_n$ are unknown.

Intuitively, this shows how information flows from the call node to function entry node.

Exit Node

If v is an exit node of a function, we can also view it as a no-op node and define

$$[\![v]\!] = JOIN(v) = \cup_{w \in pred(v)} [\![w]\!]$$

After-call Node

If v is an after-call node and suppose

- v stores return value $result$ in variable X
- the accompanying call node of v is v'
- $w \in pred(v)$ is the function exit node

Then we have the constraint

$$[\![v]\!] = [\![v']\!][X \to [\![w]\!](result)]$$

It does the propagation of the return value.

Actually, there is a hint that it only consider passing value in a parameter, and it does not care about passing reference in a parameter and global variables.

Context Sensitive

Interprocedurally invalid paths: dataflow from one call node propagates through the function body and returns **not only** at the matching after-call node.

A naive approach is to use **function-cloning**, but it will increase the program size significantly. Actually, we can instead encode the relevant information to distinguish the different calls by the use of more expressive lattices.

Context-insensitive dataflow analysis can be expressed as a lattice $States^n$ where $States = Var \to Sign$ and $n = |Nodes|$.

However, context-sensitive dataflow analysis uses a lattice of the form

$$(Contexts \to lift(States))^n$$

where $Contexts$ is a set of call contexts.

Since some call contexts are not feasible, they can only map to the bottom element of $lift(States)$, which is denoted as $unreachable$.

Note that if $Contexts$ is a singleton set, then it becomes a context-insensitive analysis. If we set $Contexts = States$, then it allows full context-sensitivity.

Take assignment statement `X = E` as an example,

- In intraprocedural analysis, we have

  $$[\![v]\!] = JOIN(v)[X \to eval(JOIN(v), E)]$$

  where

  $$JOIN(v) = \cup_{w \in pred(v)} [\![w]\!]$$

- In interprocedural analysis, we set the calling context of the method that `X = E` is in as c.
  - if $s = JOIN(v, c) \in States$, then $[\![v]\!](c) = JOIN(v)[X \to eval(s, E)]$
  - if $s = JOIN(v, c) = unreachable$, then $[\![v]\!](c) = unreachable$

  where

  $$JOIN(v, c) = \cup_{w \in pred(v)} [\![w]\!](c)$$

  Therefore, the it recognizes different contexts and eliminates the information flow of different contexts.

Here we introduce two context sensitive approaches: `Call Strings` and `Functional Approach`.

Call Strings

Let $Calls$ be the set of call nodes in the CFG, then we can define $contexts$

$$Contexts = Calls^{\leq k}$$

where k is a pre-defined positive integer, meaning the maximum length of call chain.

Here, a tuple $(c_1, c_2, \dots, c_m) \in Calls^{\leq k}$ identifies the topmost m call sites on the call stack.

For lattice element $(e_1, e_2, \dots, e_n) \in (Contexts \to States)^n$, $e_i(c_1, c_2, \dots, c_m)$ provides an abstract state that approximates the runtime states that may appear at the i'th CFG node, assuming that the sequence of call site is $c_1 \leftarrow c_2 \leftarrow \dots \leftarrow c_m$.

Since k is finite, m as the length of tuple is also finite, which ensures that $Contexts$ is also finite.

Entry Node

If v is an entry node of a function $f(b_1, \ldots, b_n)$, then it needs to take the call context c(obvious that $c = w$) at v and the call context c' at each call node w into account:

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

where $s_w^{c'}$ denotes the abstract state created from the call at node w in context c'

- if $[\![w]\!](c') = unreachable$, then $s_w^{c'} = unreachable$.
- otherwise, $s_w^{c'} = \perp[b_1 \to eval([\![w]\!](c'), E_1^w), \ldots, b_n \to eval([\![w]\!](c'), E_n^w)]$

It implies that no new dataflow can appear from call node w in context c' if that combination of node and context is unreachable.

To be more intuitive, we can let

$$s_w^{c'} \subseteq [\![v]\!](w)$$

for v's every call node w and w's corresponding context c'.

After-call Node

If v is an after-call node and suppose

- v stores return value $result$ in variable X
- the accompanying call node of v is v'
- $w \in pred(v)$ is the function exit node

Then the constraint rule for v merges the abstract state from v' and $result$ from w. Note that we set the context of v' as c and obviously the context of w is v'.

- if $[\![v']\!](c) = unreachable \vee [\![w]\!](v') = unreachable$, then $[\![v]\!](c) = unreachable$.
- otherwise, $[\![v]\!](c) = [\![v']\!](c)[X \to [\![w]\!](v')(result)]$

Actually, there is a hint that it only consider passing value in a parameter, and it does not care about passing reference in a parameter and global variables.

In summary, the call string approach distinguishes calls to the same procedure based on the call sites that appear in the call stack. The value of k is the for approximation as a trade-off between precision and performance.

Functional Approach

As we can see, the `Call String` method analyzes the method l times where l is the number of call sites of that method.

Rather than distinguishing calls based on **control flow information from the call stack**, we can use `Functional Approach` to distinguish calls based on data from **the abstract states at the calls**. That is, we can set

$$Contexts = States$$

And the lattice becomes

$$(States \to lift(States))^n$$

For each CFG node v, its lattice element is a map $m : States \rightarrow lift(States)$. If the current function containing v is entered in the state that matches s, then $m(s)$ is the possible states at v. If there is no execution of the program where the function is entered in a state that matches s and reaches v, then $m(s) = unreachable$ at v.

Entry Node

If v is an exit node of function f, the map m is a summary of f that maps abstract entry states to abstract exit states.

The constraint rule for an entry node v of function $f(b_1, , \ldots, b_n)$ is almost the same as the call strings approach:

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

where $s_w^{c'}$ denotes the abstract state created from the call at node w in context c'(Same as the definition above)

- if $[\![w]\!](c') = unreachable$, then $s_w^{c'} = unreachable$.
- otherwise, $s_w^{c'} = \bot[b_1 \rightarrow eval([\![w]\!](c'), E_1^w), \ldots, b_n \rightarrow eval([\![w]\!](c'), E_n^w)]$

Here, the abstract state computed for the call context c at the entry node v only includes the information of parameters at the call site.

If we use inequations to express, we can have

$$s_w^{c'} \subseteq [\![v]\!](s_w^{c'})$$

It shows that at call w in context c', the abstract state $s_w^{c'}$ is propagated to the function entry node v in a context that is identical to $s_w^{c'}$.

After-Call Node

If v is an after-call node and suppose

- v stores return value $result$ in variable X
- the accompanying call node of v is v'
- $w \in pred(v)$ is the function exit node

Then the constraint rule for v merges the abstract state from v' and $result$ from w. Note that we set the context of v' as c and obviously the context of w is $s_{v'}^c$.

- if $[\![v']\!](c) = unreachable \vee [\![w]\!](s_{v'}^c) = unreachable$, then $[\![v]\!](c) = unreachable$.
- otherwise, $[\![v]\!](c) = [\![v']\!](c)[X \rightarrow [\![w]\!](s_{v'}^c)(result)]$

In summary, Context sensitivity with `functional approach` gives the optimal precision for interprocedural analysis. However, it is at the cost of higher cost.