Interprocedural Analysis and Context Sensitivity

Static Program Analysis

Introduction (Approx. 2 Minutes)

Our goal is to analyze whole programs containing multiple functions and function calls. When we do this, we can no longer treat function calls as "black boxes" where we assume the worst-case scenario (like returning $\top$). We need to model the flow of data *into* functions (parameters) and *out* of functions (return values).

To keep things manageable, we rely on a specific normalization of the TIP language. We assume all function calls happen in isolation within an assignment.

Write on the board:

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

This normalization simplifies our Control Flow Graph construction significantly because we don't have to deal with nested calls inside expressions.

8.1 Interprocedural Control Flow Graphs (Approx. 4 Minutes)

Let's look at how we construct the Interprocedural CFG. We take the CFGs of individual functions and "glue" them together.

For every function call statement in the program, we split it into two distinct nodes.

Write on the board:
1. The Call Node: Represents the connection from the caller to the entry of f.
2. The After-Call Node: Where execution resumes after returning from f.

Conceptually, the flow looks like this:

1. The control flows from the **Call Node** to the distinct **Entry Node** of the callee function.
2. The function executes.
3. The control flows from the **Exit Node** of the callee back to the **After-Call Node**.

There is one crucial addition: we add a special edge directly connecting the Call Node to the After-Call Node.

Write on the board:
Draw a diagram:

Call Node (v') $\rightarrow$ *(Edge to Entry)*
$\downarrow$ *(Local edge)*
After-Call Node (v) $\leftarrow$ *(Edge from Exit)*

This local edge is vital. It preserves the values of local variables from the caller that are *not* involved in the function call. Without this, we would lose the state of local variables (like loop counters or temporary variables) while the function executes.

We also normalize return statements. A statement ‘return E;‘ is treated as an assignment to a special variable called ‘result‘.

Write on the board:

$$\text{result} = E$$

Dataflow Constraints (Context Insensitive)

Now, let’s look at the constraints for the Sign Analysis as our running example. This is the ”naive” or **Context Insensitive** approach.

For a function entry node v, we must collect abstract values from *all* possible call sites w.

Write on the board:
Constraint for Entry Node v:

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

where

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

Here, b_i are the formal parameters and E_i^w are the actual arguments at call site w. We join ($\bigsqcup$) the states from all predecessors because the function could be called from anywhere.

For the return flow—at the after-call node v—we need to combine two things: the result from the function, and the local variables we saved earlier.

Write on the board:
Constraint for After-Call Node v:

$$[[v]] = [[v']][X \mapsto [[w]](\text{result})]$$

(v' is the call node, w is the function exit node)

This formula says: take the state from the call node v' (restoring our locals), but update the variable X with the value of ‘result‘ coming from the exit node w.

8.2 The Problem: Context Insensitivity (Approx. 2 Minutes)

The approach I just described is **Context Insensitive**. It does not distinguish between different calls to the same function. This leads to the problem of **Interprocedurally Invalid Paths**.

Imagine a function ‘inc(x)‘ that returns ‘x+1‘.

- Call 1: ‘inc(0)‘ (Expects positive return)

- Call 2: ‘inc(-5)‘ (Expects negative return)

Since we join the inputs at the entry node, the analysis thinks the input is $\{0, -\}$. Consequently, it thinks the output is $\{+, 0, -\}$. When we return to Call 1, we report that the result could be negative. This is imprecise. We effectively allowed data from Call 2 to flow back into Call 1.

Context Sensitivity (Approx. 1 Minute)

To fix this, we need **Context Sensitivity**. We want to distinguish between different calls. We do this by lifting our lattice. Instead of just mapping nodes to states ($Node \to State$), we map nodes to a function from contexts to states.

Write on the board:

$$Context \to lift(State)$$

The choice of what *Context* is defines the precision and complexity of our analysis.

8.3 Call Strings Approach (Approx. 3 Minutes)

The first major strategy is the **Call Strings** approach (also known as k-CFA).

Here, a context is a tuple of call sites $(c_1, c_2, \dots)$ representing the call stack.

If we choose $k = 1$, the context is simply the most recent call site.

This is logically equivalent to **Function Cloning**, where we create a copy of the function for every distinct place it is called.

Let’s look at the constraint for a function entry node v using Call Strings ($k = 1$). We only merge information if the context matches the specific call node.

Write on the board:

$$[[v]](c) = \bigsqcup_{w \in pred(v), c = w} s_w^c$$

Instead of joining *all* predecessors, we only take the state from the predecessor w that matches the current context c. This keeps the data separate.

Similarly, at the after-call node, we only accept return values that match our current context, effectively filtering out the ”invalid paths.”

8.4 Functional Approach (Approx. 3 Minutes)

The second strategy is the **Functional Approach**. While Call Strings distinguish calls based on *where* they came from (control flow), the Functional Approach distinguishes calls based on *data*.

We define the Context to be the Abstract State itself.

Write on the board:

$$Context = State$$

Lattice: $State \rightarrow lift(State)$

This views a function as a mapping from input states to output states—a **Function Summary**.

If we call ‘inc(0)‘ and ‘inc(0)‘ from two different places, the Call String approach analyzes it twice. The Functional approach sees the input state is the same $\{0\}$ and analyzes it once, reusing the result.

The constraint for the entry node becomes:

Write on the board:

$$[[v]](c) = \bigsqcup_{w \in pred(v), c = s_w} s_w$$

Here, we effectively say: "If the state s_w being passed from the caller matches the context c we are currently analyzing, then propagate the data."

This approach offers optimal precision (equivalent to infinite inlining) but can be very expensive if the lattice of states is large.

Conclusion (Approx. 1 Minute)

To summarize:

- **Interprocedural Analysis** requires building a super-graph of all functions.
- **Context Insensitive** analysis is fast but suffers from data flowing along invalid paths (mixing up callers).
- **Call String Sensitivity (k-CFA)** separates calls based on the call stack (Control Flow).
- **Functional Sensitivity** separates calls based on input values (Data Flow).

In practice, we often use heuristics to choose k or select specific variables for the functional context to balance precision and performance.

Thank you.