Control Flow Analysis

Introduction (approx. 2 minutes)

Good morning. Today I will be presenting Chapter 10 on Control Flow Analysis. So far in our course, constructing a Control Flow Graph (CFG) has been straightforward because, in simple languages, we can see exactly which function is called just by looking at the AST.

However, complications arise when we introduce **functions as values**—also known as higher-order functions. If a function can be passed as an argument or returned as a result, we cannot trivially determine which code is executed at a call site.

Write on the table:

$$\text{let } f = \ldots \text{ in } f(x)$$

In this example, to know what $f(x)$ calls, we need to know the value of f. This creates a dependency: we need dataflow information to build the CFG, but we usually need the CFG to compute dataflow. The goal of Control Flow Analysis is to resolve this by approximating the call graph. We aim for a **conservative over-approximation**, meaning our graph may have too many edges, but never too few.

CFA for the λ-Calculus (approx. 4 minutes)

Let us start with the purest form of this problem: the λ-calculus. The syntax consists of abstractions ($\lambda Id.Exp$), applications ($Exp_1 Exp_2$), and identifiers. To analyze the control flow, we need to determine which **closures** an expression may evaluate to.

Write on the table:

$$[\![v]\!] \subseteq \text{Closures}$$

$$\text{Constraint 1: } \lambda X \in [\![\lambda X.E]\!]$$

For every AST node v, we introduce a constraint variable $[\![v]\!]$. The first constraint is simple: An abstraction $\lambda X.E$ evaluates to itself. Therefore, the token representing this abstraction is always in the set $[\![\lambda X.E]\!]$.

The second constraint handles function application $E_1 E_2$.
Write on the table:

$$\text{Constraint 2 (Application):}$$

$$\lambda X \in [\![E_1]\!] \Rightarrow ([\![E_2]\!] \subseteq [\![X]\!] \wedge [\![E]\!] \subseteq [\![E_1 E_2]\!])$$

This is a conditional constraint. It states: If the operator E_1 evaluates to the closure $\lambda X.E$, then two things must hold:

1. The argument E_2 must flow into the formal parameter X (represented by subset inclusion).
2. The result of the function body E must flow into the result of the application $E_1 E_2$.

CFA for TIP (approx. 3 minutes)

Now, let's move to the TIP language. Here we have assignments and computed function calls. Our lattice is the powerset of all function names in the program.

We generate three main types of constraints.

Write on the table:

$$1.\ f \in [\![f]\!]$$

$$2.\ [\![E]\!] \subseteq [\![X]\!] \quad (\text{for } X = E)$$

First, for every function definition named f, we have the constraint $f \in [\![f]\!]$. Second, for assignments $X = E$, the values of E flow into X.

Finally, for computed calls $E(E_1, \dots, E_n)$.

Write on the table:

$$3.\ f \in [\![E]\!] \Rightarrow ([\![E_1]\!] \subseteq [\![a_f^1]\!] \wedge \cdots \wedge [\![E'_f]\!] \subseteq [\![E(E_1 \dots)]\!])$$

This mirrors the λ-calculus rule. If E evaluates to function f, then the arguments flow to f's parameters, and f's return expression flows to the call site. Note that for direct calls like $f(\dots)$, we can simplify this to an unconditional constraint since we know exactly which function is called.

The Cubic Algorithm (approx. 4 minutes)

We have defined the constraints, but how do we solve them? These constraints are instances of a general problem that can be solved in cubic time, $\mathcal{O}(n^3)$.

We use a graph where nodes correspond to constraint variables.

Write on the table:

Graph State:

$$x.sol \subseteq T \quad (\text{Solution set})$$

$$x.succ \subseteq V \quad (\text{Successor edges})$$

$$x.cond(t) \subseteq V \times V \quad (\text{Waiting constraints})$$

The solver maintains three data structures for each variable x:

1. $x.sol$: The set of tokens found so far.
2. $x.succ$: The edges representing subset inclusion $x \subseteq y$.
3. $x.cond(t)$: A list of conditional constraints waiting for token t.

The algorithm processes constraints as follows: If we see a subset constraint $x \subseteq y$, we add an edge. If we see a conditional constraint $t \in x \Rightarrow y \subseteq z$, we check if t is in $x.sol$.

Write on the table:

Algorithm Logic:

$$\text{If } t \in x.sol \rightarrow \text{AddEdge}(y, z)$$

$$\text{Else } \rightarrow \text{Add to } x.cond(t)$$

If the token is already present, we trigger the constraint immediately. If not, we save it in $x.cond(t)$. Whenever a new token is added to a node, we "propagate" it to all successors and check any satisfied conditional constraints.

The complexity is dominated by the edges. In the worst case, we check $\mathcal{O}(n^2)$ edges, and propagate $\mathcal{O}(n)$ tokens, leading to $\mathcal{O}(n^3)$.

Object-Oriented Languages (approx. 2 minutes)

Finally, we briefly touch upon Object-Oriented languages. The challenge there is dynamic dispatch: $x.m(a)$. While we could use the full cubic analysis, the class hierarchy allows for faster approximations.

Write on the table:

1. CHA (Class Hierarchy Analysis)
2. RTA (Rapid Type Analysis)

CHA assumes that any method m in the subclasses of the declared type of x can be called. RTA refines this by only considering classes that are actually instantiated in the program.

Conclusion

In summary, Control Flow Analysis enables us to construct call graphs for languages with first-class functions by solving a set of inclusion constraints. This call graph is the necessary foundation for subsequent interprocedural dataflow analyses.