Control Flow Graphs and Call Graphs

Wei Le

February 1, 2019

Program Representations

1. syntactic information – functions, loops, branches, identifiers: *abstract syntax tree*
2. semantic information (related to actual executions and output): program paths – *control flow graphs*, *call graphs*; data relations – *dependency graphs*
3. *Control Flow Graph (CFG)* is a directed graph where node is a statement/instruction/a *basic block* (a sequence of instructions that do not have branches), and an edge indicates the order of the two statements/instructions/basic blocks
   - loop
   - branch (if, switch, goto)
   - call
   - exception handling routines

Control Flow Graphs

1. History: 1970, Frances Allen's papers: "Control Flow Analysis" and "A Basis for Program Optimization" established "intervals" as the context for efficient and effective dataflow analysis and optimization
2. Each function has a CFG, single entry and single exit
3. *Interprocedural Control Flow Graphs (ICFG)*: connect CFGs for all the functions in the code considering their calling relations
4. Goal: sequence the statements, make the paths available
5. *Path*: a sequence of node on the CFG (static), including an entry node and an exit node; *Infeasible paths*: paths never can be executed; *Path segment*: a subsequence of nodes along the path; *Trace*: a sequence of instructions performed during execution (dynamic)

Constructing CFG

1. build *abstract syntax tree (AST)*: parse tree in an abstract form
2. convert AST to CFG
3. There are many off-the-shelf tools: llvm for c/c++; soot for java; Boa, Helium cfg (source level)

see some examples (Constructing CFG from AST, paths, infeasible paths, path segments, basic block, ICFG)

Challenges

- Many analyses traverse the CFG, the analyses need to know if they are visiting a loop, then how to identify loops on a CFG?
- *Call graphs* – representing calling relations, there is a node from caller to callee: how to resolve function pointers, virtual functions (see examples)
- special software, e.g., event-driven and framework based architecture like Android: callbacks, synchronous and asynchronous execution

Loops

- Most of the execution time is spent in loops - the 90/10 law, which states that 90% of the time is spent in 10% of the code, and only 10% of the time in the remaining 90% of the code.
- Node d of a CFG *dominates* node n if every path from the entry node of the graph to n passes through d, noted as d *dom* n
- *(optional materials) properties of dominance relations, strictly dominate, immediate dominate, post dominator, dominator tree*

Loops [dragon book p.531]

A set of nodes L in CFG is a loop if, L contains a node e, called *loop entry* or *head*

- e is not an entry of the entire flow graph
- No node in L besides e has a predecessor outside L, (e dominates all the nodes in the loop)
- every node in L has a nonempty path completely within L, to e

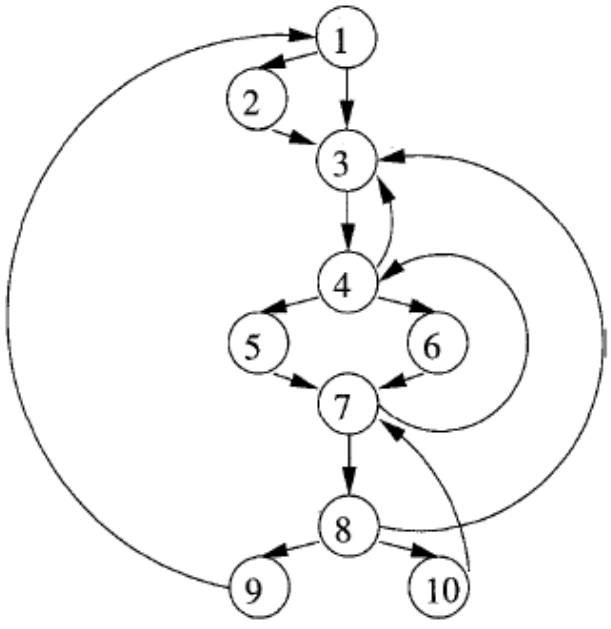
Natural Loops

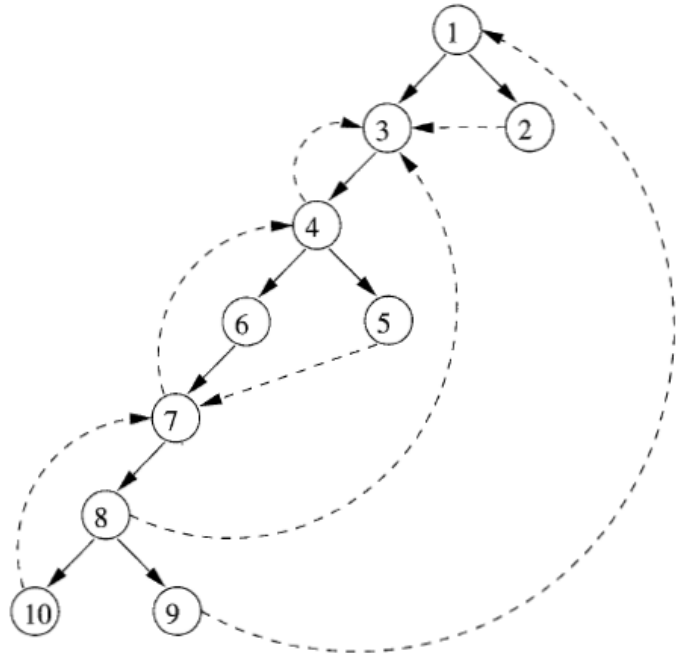
1. Single entry node (d)
   - no jumps into middle of loop
   - d dominates all nodes in loop
2. Requires back edge into loop header ($n \rightarrow d$)
3. *back edge* – head (ancestor) dominates its tail (decedent), any edge from tail to head is a back edge
4. Loop terminologies: *single loop*, *nested loop*, *inner loop*, *outer loop*

CFG Reducibility

- CFG is reducible if every loop is a natural loop
- Graph reducibility:
  - A graph traversal visits each node via edges
  - A *depth-first traversal* of a graph visits all the nodes in the graph once, by starting at the entry node and visiting the nodes as far away from the entry node as quickly as possible before backtracking.
  - In a depth-first presentation of the flow graph, the parent is the ancestor and the children is the decedent. *Retreating edge*: connects a decedent to its ancestor [dragon book p.662]
  - A graph is *reducible* if every retreating edge in a graph is a back edge: remove the back edges, the result should be acyclic

Example



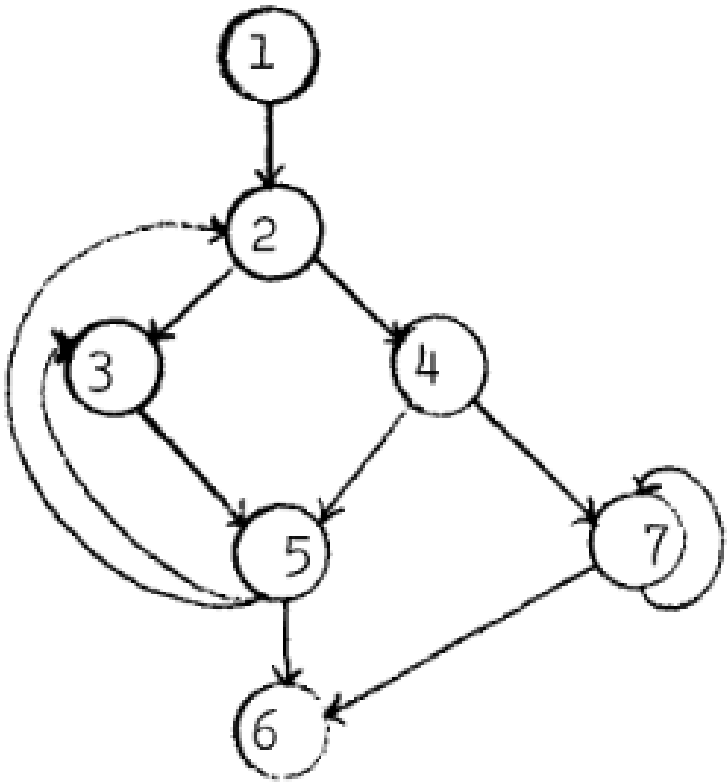


Reducibility in Practice

- If you use only while-loops, for-loops, repeat-loops, if-then(-else), break, and continue, then your flow graph is reducible.
- Some languages only permit procedures with reducible flowgraphs (e.g., Java)
- "GOTO Considered Harmful": can introduce irreducibility
  - FORTRAN
  - C
  - C++

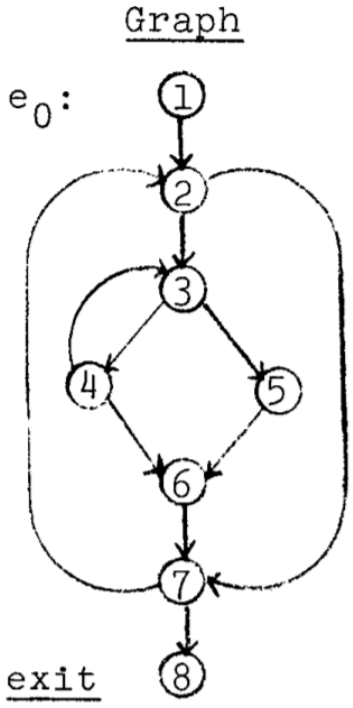
Intervals on CFG [1970:Allen]

- Path P = (b1,...,bn)
- A *closed path or circuit* is a path in which bn = b1, e.g.,(3,5,3), (5,3,5), (2,3,5,2), (3,5,2,3), (5,2,3,5), (4,5,2,4), (2,3,5,3,5,2).



Intervals on CFG [1970:Allen]

- Given a node h, an interval I(h) is the maximal, single entry subgraph for which h is the entry node and in which all closed paths contain h



Intervals

I(1) = 1

I(2) = 2

I(3) = 3,4,5,6

I(7) = 7,8

(the naming of the nodes is, as usual, arbitrary)

Call Graph Construction - Virtual Functions

```
class A {
public:
   virtual void f();
   ...
};
```

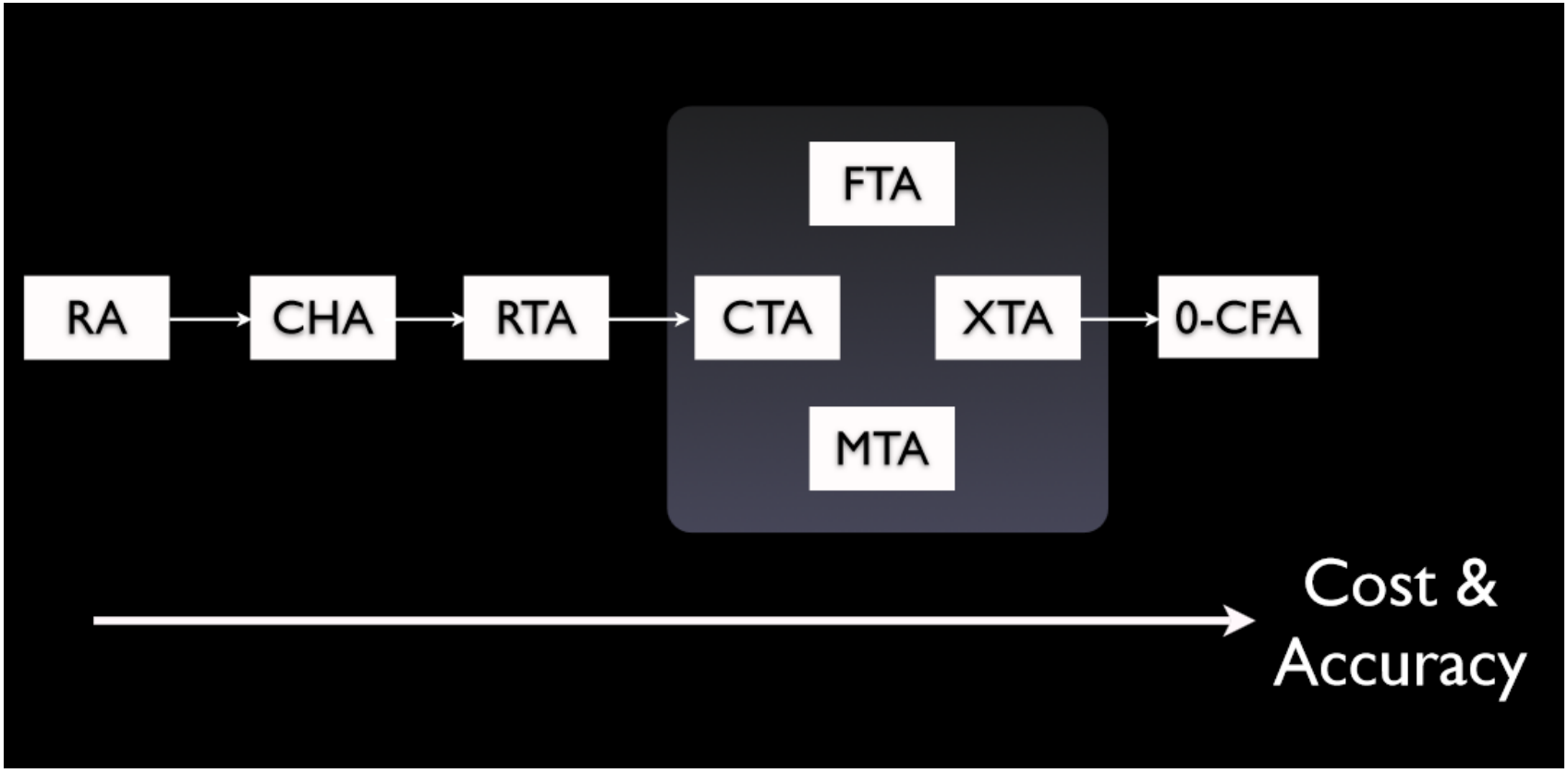
```
class B: public A {
public:
   virtual void f();
...
};
```

```
int main()
{
   A *pa = new B();

   pa->f();
   ...
}
```

Real challenge: handling Anroid framework

Call Graph Construction - Handling Virtual Functions



Class Hierarchy Analysis: CHA

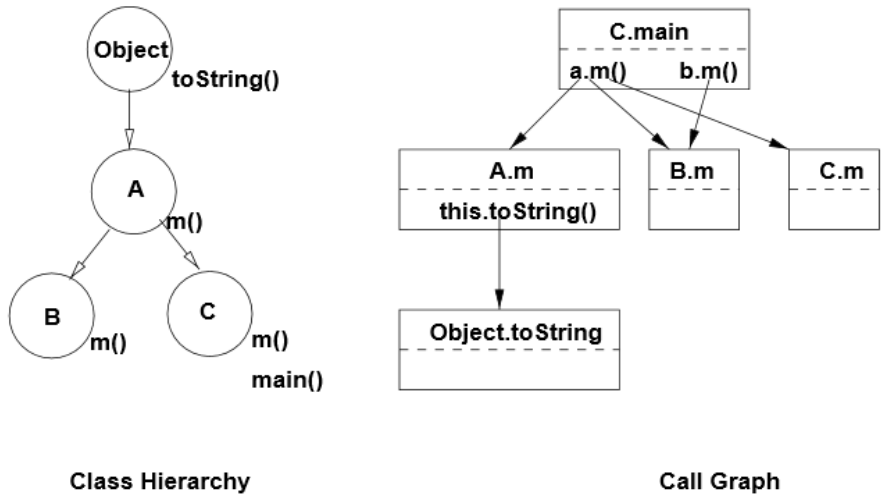
```
class A extends Object {
  String m() {
    return(this.toString());
  }
}

class B extends A {
  String m() { ... }
}

class C extends A {
  String m() { ... }
  public static void main(...) {
   A a = new A();
   B b = new B();
   String s;

   ...
   s = a.m();
   s = b.m();
  }
}
```

(a) Example Program



(b) Class Hierarchy and Call Graph

k-CFA [1988:Shivers] (k-Control Flow analysis)

Resolving the value of pointers and references:

- 0-CFA: *context-insensitive* pointer analysis
- *k-CFA*: k number of calls are considered

Type Inference, Alias Analysis, Call Graph Construction for OO programs

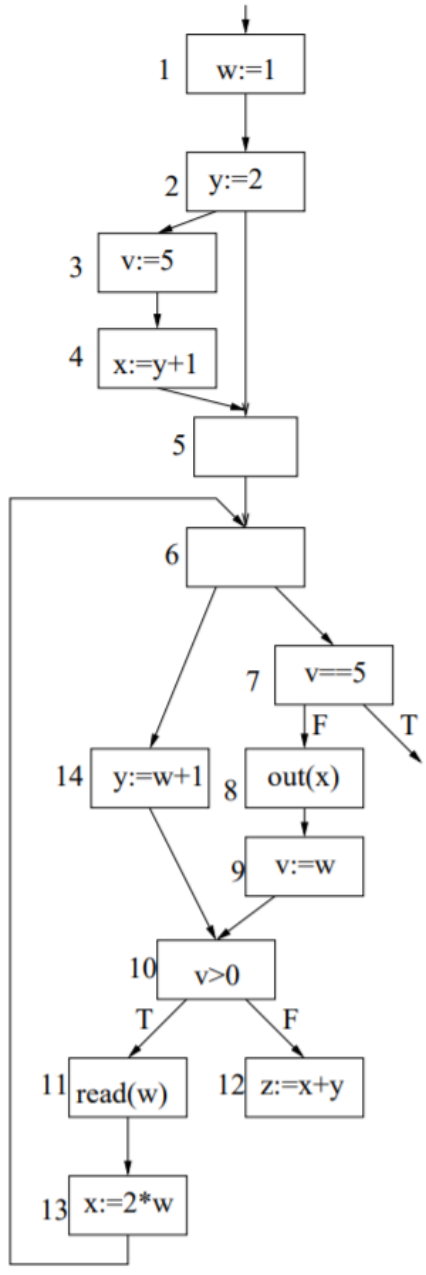
- Call graph construction needs to know the type of the object receivers for the virtual functions
- Determine types of the set of relevant variables: type inferences – infer types of program variables
- Object receivers may alias to a set of reference variables so we need to perform alias analysis

Function Pointers [2004:atkinson]

Resolving function pointers based on the *type signature*

```
int (*q) ()
int main() { ...
  char *x = "a";
  int *y = 1;
  (*q) (2, x); ...
  (*q) (3, y);
}
char q1(int x, int *p) { ... }
int q2(int x, int *p) { ... }
int q3(int x, char *p) { ... }
```

Infeasible Paths Detection [1997:Bodik]



Further Reading

1. Control Flow Analysis by Fran Allen
2. Scalable Propagation-Based Call Graph Construction Algorithms by Frank Tip and Jens Palsberg
3. Refining Data Flow Information using Infeasible Paths by Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa