Control Flow Graphs and Call Graphs

Wei Le

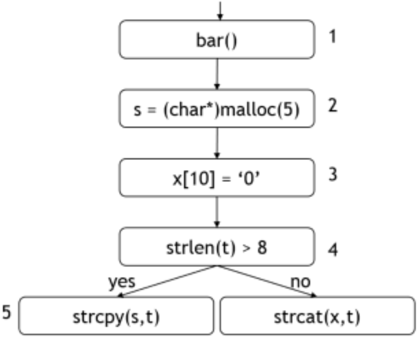
January 24, 2020

Program Representations

1. syntactic information – functions, loops, branches, identifiers: *abstract syntax tree*
2. semantic information (related to actual executions and output values): 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

Control Flow Graphs

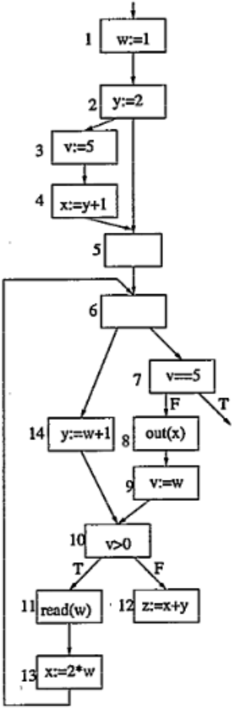
```
bar();
s = (char*)malloc(5);
x[10] = '\0';
if(strlen(t)>8)
   strcpy(s,t);
else
    strcat(x,t);
```



See more real-world examples

Control Flow Graphs

1. History: 1970, Frances Allen's papers: "Control Flow Analysis" and "A Basis for Program Optimization" for analyzing programs for optimizations
2. Each function has a CFG, single entry and single exit
3. Goal: sequence the statements, make the paths available
4. *Path*: a sequence of node on the CFG, including an entry node and an exit node
5. *Trace*: a sequence of instructions performed during execution
6. *Infeasible paths*: paths never can be executed
7. *Path segment*: a subsequence of nodes along the path


1
w:=1
2
y:=2
3
v:=5
4
x:=y+1
5
6
7
v==5
F
T
14
y:=w+1
8
out(x)
9
v:=w
10
v>0
T
F
11
read(w)
12
z:=x+y
13
x:=2*w

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)

Loops [dragon book p.531]

- 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.
- How to find loops in CFG: 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

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*
5. CFG is *reducible* if every loop is a natural loop

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++

Call Graph

- *Call graphs* – representing calling relations, there is an edge from caller to callee
- Challenge: function pointers, virtual functions, event-driven and framework based architecture like Android: callbacks, synchronous and asynchronous execution

Call Graph Construction - Virtual Functions

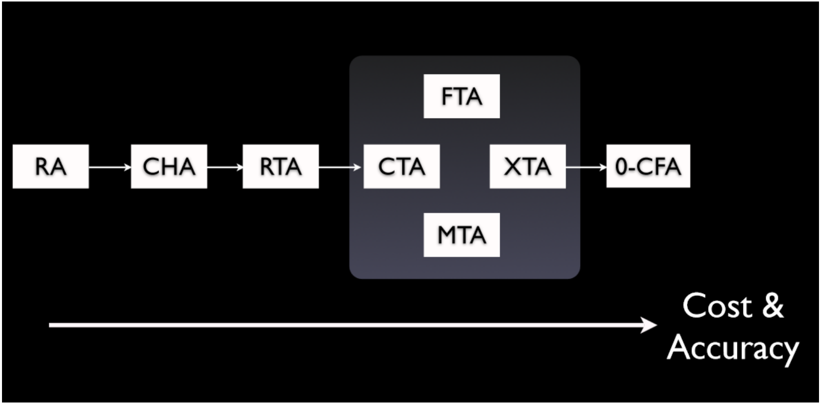
```
class A {
public:
   virtual void f();
   ...
};
```

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

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

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

Algorithms for Handling Virtual Functions



Class Hierarchy Analysis: CHA

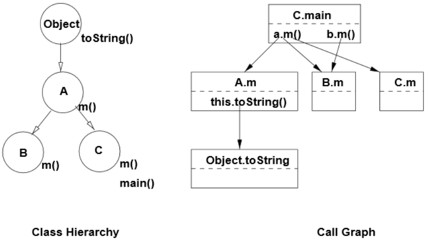
```
class A extends Object {
  String m() {
    return(this.toString());
  }
}

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 within a function
- *k-CFA*: *k* number of calls are considered when resolving pointers

Relations of Type Inference, Alias Analysis, Call Graph Construction

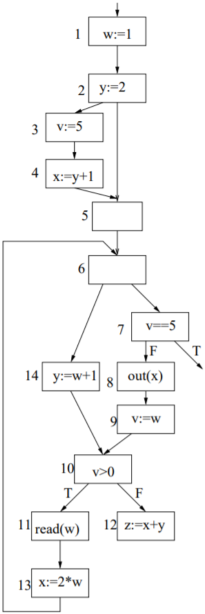
- Call graph construction needs to know the type of the object receivers for the virtual functions
- Determine types of the set of relevant variables: type inferences – infer types of program variables
- Object receivers may alias to a set of reference variables so we need to perform alias analysis

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