

Program Representation for Control Flow

Wei Le

January 20, 2021

Outline

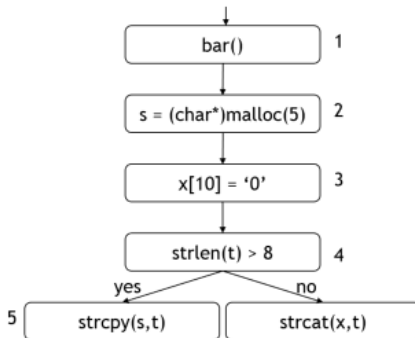
- ▶ Control flow graph (CFG)
- ▶ Call graph
- ▶ Inteprocedural control flow graph (ICFG)

What is program control flow?

- ▶ *Control flow*: the execution order of statements of a program
- ▶ The problem of identifying control flow: identifying the execution order of statements from program source code (sometimes from binary code)
- ▶ *Control flow graph*: represent such order in a graph representation, the program paths will be available by traversing the graph

Control flow graph: an example

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



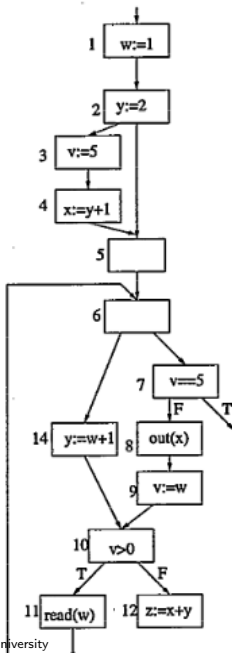
Control flow graph

- ▶ history: 1970, Frances Allen's papers: "Control Flow Analysis" and "A Basis for Program Optimization": using control flow graph to analyze programs for code optimizations
- ▶ it is a directed graph, we also call it CFG
- ▶ each function has a control flow graph
- ▶ has only one entry and one exit
- ▶ the node is a statement (source code) or an instruction (binary code); a node can also be a *basic block* (a sequence of statements/instructions that do not have branches),
- ▶ an edge indicates the order of the two statements/instructions/basic blocks

Other concepts related to program control flow

1. *Path*: a sequence of node on the CFG, including an entry node and an exit node
2. *Trace*: a sequence of instructions performed during execution
3. *Infeasible paths*: paths never can be executed
4. *Path segment*: a subsequence of nodes along the path

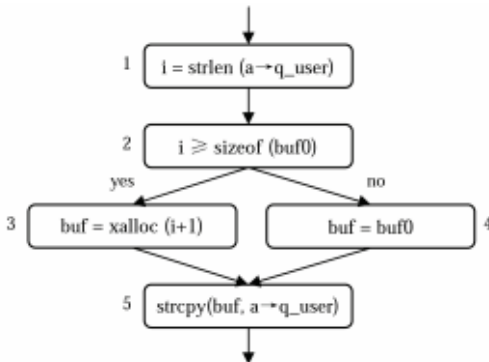
Paths and infeasible path: an example



Control flow graphs and bug finding

Bug detection:

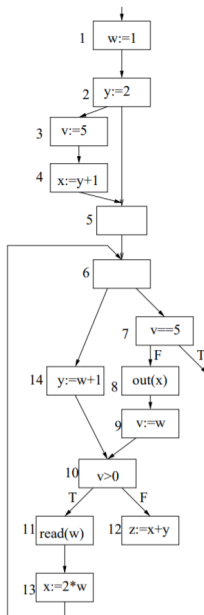
- ▶ using buggy code patterns (code smell) to scan new software: disadvantage is a large number of false positives
- ▶ finding erroneous/undesired state on paths



Automatically constructing CFG

- ▶ Step 1: build *abstract syntax tree (AST)*: parse tree in an abstract form
- ▶ Step 2: convert AST to CFG
- ▶ There are many off-the-shelf tools: llvm for c/c++; soot for java

Automatic infeasible paths detection [1997:Bodik]



Identifying loops from CFG

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.

A set of nodes L in CFG is a *natural loop* if, L contains a node e , called *loop entry* or *head* [dragon book p.531]:

- ▶ 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): if every path from the entry node of the graph to n passes through d , noted as $d \text{ dom } n$
- ▶ every node in L has a nonempty path, completely within L , to e

Reducible CFG and natural loops

1. Single entry node (e)
 - ▶ no jumps into middle of loop
 - ▶ e dominates all nodes in loop
2. Requires a back edge into loop header ($n \rightarrow e, n \in L$)
3. *back edge* – head (ancestor) dominates its tail (decendent), 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

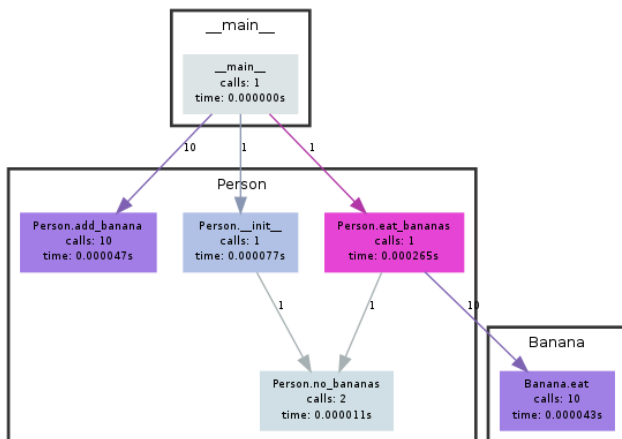
- ▶ If you use only while-loops, for-loops, repeat-loops, if-then(-else), break, and continue, then your control flow graph is reducible.
- ▶ Some languages only permit procedures with reducible control flow graphs (e.g., Java)
- ▶ “GOTO Considered Harmful”: can introduce irreducibility
 - ▶ FORTRAN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

Interprocedural control flow graph (ICFG)

- ▶ ICFG: representing control flow for the entire program
- ▶ what is a valid interprocedural path and how to represent it in ICFG?
- ▶ Combining: CFG and *call graph*: calling relationships of functions

Call graph

- *Call graph*: representing calling relations between functions, there is an edge from caller to callee See example from wiki ¹



Generated by Python Call Graph v1.0.0
<http://pycallgraph.slowchop.com>

¹https://en.wikipedia.org/wiki/Call_graph

Constructing call graphs

There are off-the-shelf tools: llvm for c/c++; soot for java, except that the following challenges are still under active research:

- ▶ function pointers
- ▶ virtual functions
- ▶ event-driven and framework based architecture like Android: callbacks (the library code calls user's functions)

Function pointers [2004:atkinson]

Resolving function pointers based on the types in function signatures

```
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) { ... }
```

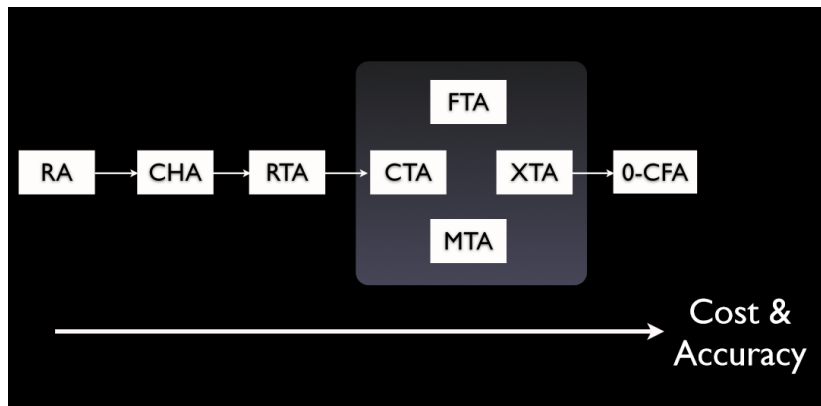
Virtual functions

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

```
int main()  
{  
    A *pa = new B();  
  
    pa->f();  
    ...  
}
```

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

Algorithms for handling virtual functions

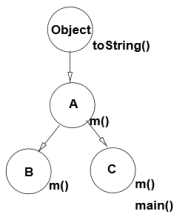


Details see paper: Scalable Propagation-Based Call Graph Construction Algorithms by Frank Tip and Jens Palsberg

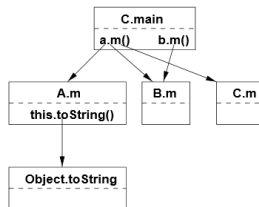
CHA: class hierarchy analysis

```
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



Class Hierarchy



Call Graph

(b) Class Hierarchy and Call Graph

Relations of type Inference, alias analysis, call graph construction in Java

- ▶ 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

Callbacks

```
class MyActivity extends Activity {
    void onCreate(...) {
        ...
    }
    void onStart() {
        Intent i = new Intent(this, S.class);
        startService(i);
        bindService(i, c);
    }
    void onResume() {
        ...
    }
    void onStop() {
        super.onStop();
        unbindService(c);
    }
}

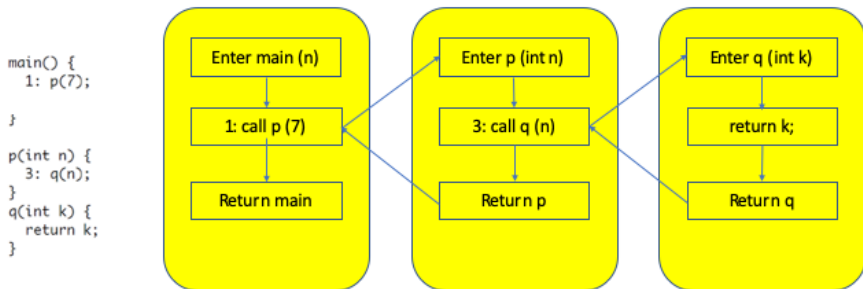
class MyService extends Service {
    void onCreate() {
        wifilock.acquire();
    }
    int onStartCommand(...) {
        ...
    }
    void onBind(...) {
        ...
    }
    boolean onUnbind(...) {
        ...
    }
    void onDestroy() {
        if (wifilock.isHeld())
            wifilock.release();
    }
}
```

dead code

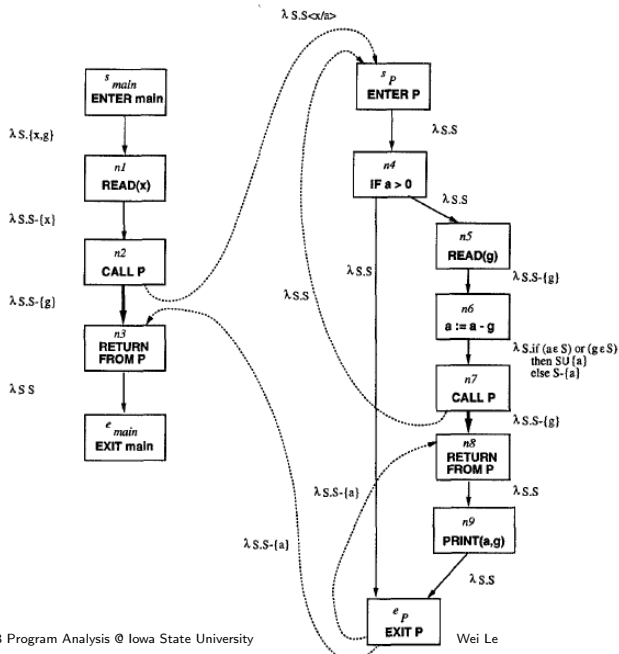
Interprocedural control flow graph (ICFG)

After ICFG and callgraph, let's revisit the concept of ICFG:

- ▶ ICFG: representing control flow for the entire program
- ▶ Finding the potentially valid execution paths for the entire program



ICFG with recursive calls



Getting paths from ICFG

Calling context-sensitivity and realizable paths:

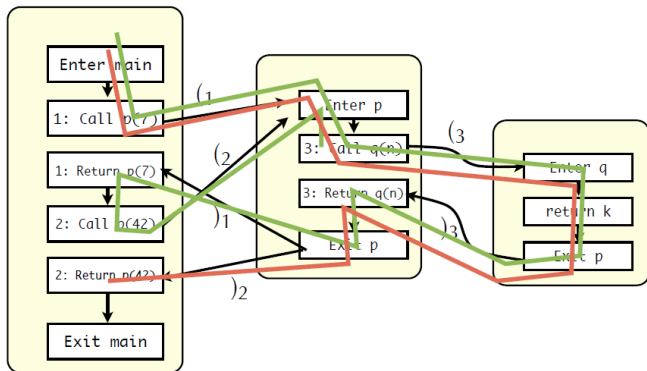
- ▶ *Context-sensitive*: when a call is invoked twice, do we distinguish at which callsite we should return from the callee (when traversing ICFG for paths)?
- ▶ *Realizable paths*: ICFG contains more paths than valid execution paths. Realizable paths are the paths on ICFG that represent potentially valid executions.

ICFG and realizable paths

```
main() {  
  1: p(7);  
  2: x:=p(42);  
}
```

```
p(int n) {  
  3: q(n);  
}
```

```
q(int k) {  
  return k;  
}
```



Realizable paths

Idea: restrict attention to **realizable paths**: paths that have proper nesting of procedure calls and exits

For each call site i , let's label the call edge " $(i$ " and the return edge " $)_i$ "

Define a grammar that represents balanced paren strings

$\text{matched} ::= \epsilon$ *empty string*
 $| e$ *anything not containing parens*
 $| \text{matched matched}$
 $| (i \text{ matched })_i$

- Corresponds to matching procedure returns with procedure calls

Define grammar of partially balanced parens (calls that have not yet returned)

$\text{realizable} ::= \epsilon$
 $| (i \text{ realizable}$
 $| \text{matched realizable}$

CFL (context-free language) reachability

Reducing to the CFL reachability problem: Let L be a context-free language over alphabet Σ

- ▶ Let G be graph with edges labeled from Σ
- ▶ Each path in G defines word over Σ
- ▶ A path in G is an L -path if its word is in L

CFL reachability problems

Computing *realizable paths*: $O(n^3)$

- ▶ All-pairs L-path problem: for all pairs of nodes $n1$ and $n2$, finding an L-path from $n1$ to $n2$
- ▶ Single-source L-path problem: for all nodes $n2$, finding an L-path from given node $n1$ to $n2$
- ▶ Single-target L-path problem: for all nodes $n1$, finding an L-path from $n1$ to given node $n2$
- ▶ Single-source single-target L-path problem: finding an L-path from given node $n1$ to given node $n2$

Bug detection on ICFG

- ▶ Enumerate all the paths: too many paths
- ▶ Demand-driven algorithms: new after 2000
- ▶ Summary based algorithms: summarize the effect of a callee and then use the summary for bug detection

Computing procedural summaries on ICFG

When call p is encountered in context C , with input D , check if procedure summary for p in context C exists.

- ▶ If not, process p in context C with input D
- ▶ If yes, with input D' and output E'
- ▶ If $D' \sqsubseteq D$, then use E'
- ▶ if $D' \not\sqsubseteq D$, then process p in context C with input $D' \sqcap D$ (merge the dataflow)
- ▶ If output of p in context C changes then may need to reprocess anything that called it
- ▶ Need to take care with recursive calls

A tool demo using Understand