CFA

Simone Campanoni
simonec@eecs.northwestern.edu
Problems with Canvas?
Problems with slides?
Problems with H0?
Any problem?
CFA Outline

• Why do we need Control Flow Analysis?

• Basic blocks

• Control flow graph
Functions and instructions

runOnFunction’s job is to analyze/transform a function F ...

... by analyzing/transforming its instructions

What is the instruction that will be executed after I?

The order of instructions isn’t the execution one
Storing order ≠ executing order

When the storing order is chosen (compile time), the execution order isn’t known

```c
int myF(int a) {
    int x = a + 1;
    if (a > 5) {
        x++;
    } else {
        x--;
    }
    return x;
}
```

Common pitfall 1:
if instruction i1 has been stored before i2, then i2 is always executed after i1

Common pitfall 2:
if instruction i1 has been stored before i2, then i2 can execute after i1

we need to analyze the execution paths
This is the job of Control Flow Analysis
• The need of CFAs can be seen in their uses (e.g., code transformations)
  • Constant propagation

• Before showing the need of CFAs, let me introduce a few concepts
Variables and constants

\begin{align*}
x &= 0; \\
y &= x + 1;
\end{align*}

Constants

Variable definitions

Variable uses
Control flows

**Control flow:** sequence of instructions in a program ignoring data values and arithmetic operations

```plaintext
x = a;
y = x + 1;
x++;
return x + y;
```

```plaintext
x = a;
y = x + 1;
if (y > 5){
    x--;
} else {
    x++;
}
```
Program semantic

Program semantic: Input -> Output

Two programs, p1 and p2, are semantically equivalent if for a given input, p1 and p2 generate the same output for every possible input.

```c
int main (int argc, char *argv[])
{
    int x = argc;
    int y = x + 1;
    y++;
    printf("%d", x + y);
    return 0;
}
```

```c
int main (int argc, char *argv[])
{
    int y = argc + 2;
    printf("%d", argc + y);
    return 0;
}
```

```c
int main (int argc, char *argv[])
{
    int y = argc + 2;
    printf("%d", 2*argc + 2);
    return 0;
}
```
Program semantic

Program semantic: Input -> Output

Two programs, p1 and p2, are semantically equivalent if for a given input, p1 and p2 generate the same output for every possible input.

```
int main (int argc, char *argv[])
{
    int y = argc + 2;
    printf("%d", 2*argc + 2);
    return 1;
}
```

```
int main (int argc, char *argv[])
{
    int y = argc + 2;
    printf("%d", 2*argc + 2);
    return 0;
}
```

$ ./myprog 2
6
$ echo $?
Two programs, p1 and p2, are semantically equivalent if for a given input, p1 and p2 generate the same output for every possible input.

```c
int main ( int argc, char *argv[] )
{
    argc++;  
    printf("%d", 2*argc + 2);
    return 0;
}
```

```c
int main ( int argc, char *argv[] )
{
    int y = argc + 2;
    printf("%d", 2*argc + 2);
    return 0;
}
```
Code transformation example:
constant propagation

int sumcalc (int a, int b, int N){
    int x,y;
    x = 0;
y = 0;
    for (int i=0; i <= N; i++){
        x = x + a * b;
        x = x + b * y;
    }
    return x;
}
Constant propagation and CFA

• Find a constant expression
  \textit{Instruction i:} \ varX = \text{CONSTANT\_EXPRESSION}

• Replace the use of the variable defined in a constant expression with that constant if
  • All control flows to the use of \( \text{varX} \) pass \( i \)
  • There are no intervening definition of that variable

\textbf{We need to know the control flows of a program}

\textbf{Control flow:} sequence of instructions in a program ignoring data values and arithmetic operations

• Control Flow Analysis discovers facts about control flows
CFA Outline

• Why do we need Control Flow Analysis?

• Basic blocks

• Control flow graph
Representing the control flow of the program

- Most instructions
- Jump instructions
- Branch instructions
Representing the control flow of the program

A graph where nodes are instructions
- Very large
- Lot of straight-line connections
- Can we simplify it?

**Basic block**
Sequence of instructions that is always entered at the beginning and exited at the end
Basic blocks

A basic block is a maximal sequence of instructions such that
• Only the first one can be reached from outside this basic block

• All instructions within are executed consecutively if the first one get executed
  • Only the last instruction can be a branch/jump
  • Only the first instruction can be a label

• Is the storing sequence = execution order in a basic block?
Basic blocks

• Automatically identified
• Algorithm:
  • Code changes trigger the re-identification
  • Increase the compilation time

• Enforced by design
• Instruction exists only within the context of its basic block
• To define a function:
  • you define its basic blocks first
  • Then you define the instructions of each basic block
  • Add missing labels
  • Add explicit jumps
  • Delete empty basic blocks

What about calls?

- Program exits
- Exceptions
Basic blocks in LLVM

• Every basic block in LLVM must
  • Have a label associated to it
  • Have a “terminator” at the end of it

• The first basic block of LLVM (entry point) cannot have predecessors

• LLVM organizes “compiler concepts” in containers
  • A basic block is a container of ordered LLVM instructions
  • A function is a container of basic blocks
  • A module is a container of functions
Basic blocks in LLVM (2)

• LLVM C++ Class “BasicBlock”

• Uses:
  • BasicBlock *b = ... ;
  • Function *f = b.getParent();
  • Module *m = b.getModule();
  • Instruction *i = b.getTerminator();
  • Instruction *i = b.front();
  • size_t b.size();
Basic blocks in LLVM in action

```c
function()
{
    int a = 1;
    int b = 2;
    if (b == 2)
        // Jump instruction
        ++b;

    int c = 3;
    int d = 4;
    while (a < 5)
        // Jump instruction
        ++a;

    int e = 5;
    int f = 6;
}
```

```
void @_Z8functionv() #0 {
entry:
    %a = alloca i32, align 4
    %b = alloca i32, align 4
    %c = alloca i32, align 4
    %d = alloca i32, align 4
    while (while.cond):
        %2 = load i32* %a, align 4
        %cmp1 = icmp slt i32 %2, 5
        br i1 %cmp1, label %while.body, label %while.end
    while.body:
        %3 = load i32* %a, align 4
        %inc2 = add nsw i32 %3, 1
        store i32 %inc2, i32* %a, align 4
        br label %while.cond
    while.end:
        store i32 5, i32* %e, align 4
        store i32 6, i32* %f, align 4
        ret void
```
We need to identify all possible control flows between *instructions*

We need to identify all possible control flows between *basic blocks*

We need to know the control flows of a program

Control flow: sequence of instructions in a program ignoring data values and arithmetic operations

• Control Flow Analysis discovers facts about control flows
CFA Outline

• Why do we need Control Flow Analysis?

• Basic blocks

• Control flow graph
Control Flow Graph (CFG)

- A CFG is a graph $G = \langle \text{Nodes}, \text{Edges} \rangle$
- Nodes: Basic blocks
- Edges: $(x,y) \in \text{Edges}$ iff first instruction in basic block $y$ \textbf{might} be executed just after the last instruction of the basic block $x$
Control Flow Graph (CFG)

• Entry node: block with the first instruction of the function
• Exit nodes: blocks with the return instruction
  • Some compilers make a single exit node by adding a special node
function()
{
    int a = 1;  // Sequential instructions
    int b = 2;  // ------------------------
    if (b == 2) // Jump instruction
    {
      ++b;      // Jump target
    }
    int c = 3;  // Sequential instructions
    int d = 4;  // ------------------------
    while (a < 5)  // Jump instruction and jump target
    {
      ++a;      // Jump target
    }
    int e = 5;  // Sequential instructions
    int f = 6;  // ------------------------
}
CFG in LLVM

Differences?

Bitcode generation

```
opt -view-cfg
F.viewCFG();
```
Navigating the CFG in LLVM: from a basic block to another

Successors of a basic block

```c
for (auto sit = succ_begin(bb), set = succ_end(bb); sit != set; ++sit){
    BasicBlock *succBB = *sit;
}
```

Predecessors of a basic block

```c
for (auto it = pred_begin(bb), et = pred_end(bb); it != et; ++it){
    BasicBlock *predecessorBB = *it;
}
```
Navigating the CFG in LLVM (2)

```cpp
for (auto &b : F) {
    TerminatorInst *i = b.get Terminator();
    errs() << *i << "\n";
    for (auto index = 0; index < i->getNumSuccessors(); index++) {
        BasicBlock *succ = i->getSuccessor(index);
        errs() << " " << succ->front() << "\n";
    }
}
```
H0/tests/test1

Output of our latest pass:
Sometimes “might” isn’t enough

How to differentiate between the two situations by using only successor/predecessor relations?
Dominators

**Definition:** Node $d$ dominates node $n$ in a CFG ($d \ dom \ n$) if every control flow from the start node to $n$ goes through $d$. Every node dominates itself.

What are the dominators of basic block 1?

What are the dominators of basic blocks 1, 2, and 3?

What is the relation between instructions in different basic blocks?
A CFA to find dominators

Consider a block $n$ with $k$ predecessors $p_1$, ..., $p_k$

**Observation 1:** if $d$ dominates each $p_i$ ($1 \leq i \leq k$), then $d$ dominates $n$

**Observation 2:** if $d$ dominates $n$, then it must dominate all $p_i$

$$D[n] = \{n\} \cup (\cap_{p \in \text{predecessors}(n)} D[p])$$

To compute it:
- By iteration
- Initialize each $D[n]$ to include every one
Immediate dominators

**Definition**: the immediate dominator of a node $n$ is the unique node that strictly dominates $n$ (i.e., it isn’t $n$) but does not strictly dominate another node that strictly dominates $n$. 

![CFG](image1.png)

![Dominator tree](image2.png)
Post-dominators

**Assumption:** Single exit node in CFG

**Definition:** Node $d$ post-dominates node $n$ in a graph if every path from $n$ to the exit node goes through $d$
Post-dominators

**Assumption:** Single exit node in CFG

**Definition:** Node $d$ post-dominates node $n$ in a graph if every path from $n$ to the exit node goes through $d$

---

**Immediate post-dominator tree**

- **B:** if (par1 > 5)
- **C:** varX = par1 + 1
- **C2:** ...
- **D:** print(varX)

---

**CFG**
Another example of CFA (and CFT)

This is a simple CFA and CFG, but useful after applying several other code transformations.
Another example of CFA

• What are the possible equivalent CFGs the compiler can choose from?
• The compiler needs to be able to transform CFGs
  • CFAs tell the compiler what are the equivalent CFGs

... 
If (b == 2)
   return;
}
#elsedef CRAZY
printf(“Yep”);
#endif

clang myfile.c –DCRAZY –o myprog

... 
if (b == 2)
   ... 
else
   return

b == 2

return

return

return