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 and instructions

• 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 myFun(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,
  • then we’ll motivate CFAs using a code transformation,
  • and then we’ll talk about CFAs
Variables and constants

\[
x = 0; \\
y = x + 1;
\]

Constants

Variable definitions

Variable uses
Control flows

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

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

```c
x = a;
y = x + 1;
if (y > 5){
x--;
} else {
x++;
}
```
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 x = argc;
    int y = x + 2;
    y++;
    printf("%d", y);
    return 0;
}
```

```c
int main (int argc, char *argv[])
{
    int x = argc;
    int y = argc + 2;
    y++;
    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.

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

```c
$ ./myprog 2
6
$ echo $?
1
```
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
  
  Instruction i: varX = CONSTANT_EXPRESSION

• Replace an use of varX with CONSTANT_EXPRESSION in an instruction j if
  • All control flows that reach j pass i
  • There are no intervening definition of that variable

We need to know the control flows of a program

Control flow: sequence of instructions in a program that might execute in that order ignoring data values and arithmetic operations

• Control Flow Analysis discovers facts about control flows
But before CFA

• Before diving into control flows and control flow analysis
  • We need to introduce the concept of basic blocks and how it is implemented in LLVM
  • We also need to talk about instructions in LLVM

• Then, we’ll look at the most common control flow analysis
CFA Outline

• Why do we need Control Flow Analysis?

• Basic blocks and instructions

• 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

```
Inst = F.entryPoint()
B = new BasicBlock()
While (Inst){
  if Inst is Label {
    B = new BasicBlock()
  }
  B.add(Inst)
  if Inst is Branch/Jump{
    B = new BasicBlock()
  }
  Inst = F.nextInst(Inst)
}
```

- Add missing labels
- Add explicit jumps
- Delete empty basic blocks

What about calls?
- Program exits
- Exception
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 (BasicBlock)
  • A function is a container of basic blocks (Function)
  • A module is a container of functions (Module)

Given an object `Module &M`
Function *sqrtF = M.getFunction(“sqrt”)
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;
    // Sequential instructions
    if (b == 2)
    {
        ++b;
        // Jump instruction
    }
    int c = 3;
    int d = 4;
    // Sequential instructions
    while (a < 5)
    {
        ++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
    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
}
Instructions in LLVM

• Each instruction sub-class has extra methods for this type of instructions
  • E.g., Function * CallInst::getCalledFunction()

• You need to cast Instruction objects to access instruction-specific methods
  • LLVM redefined casting
  • bool isa<CLASS>(objectPointer)

  • CLASS *ptrCasted = cast<CLASS>(objectPointer)

  • CLASS *ptrCasted = dyn_cast<CLASS>(objectPointer)
Now that you know LLVM

• It’s time for the homework H1
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 and instructions

• 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 of basic block $y$ ($I_y$) may be executed
  just after the last instruction of the basic block $x$ ($I_x$)

Predecessor

Successor
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

![Diagram of Control Flow Graph](attachment:control_flow_graph.png)
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

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

Successors of a basic block

```cpp
for (auto succBB : successors(bb)){
```

Predecessors of a basic block

```cpp
for (auto predBB : predecessors(bb)){
```
Navigating the CFG in LLVM: from a basic block to another (the old way)

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: From an instruction to basic blocks

```c
for (auto &b : F){
    TerminatorInst *i = b.getTerminator();
    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:

```
define i32 @CAT_execution() #0 {
  entry:
    %res = alloca i32, align 4
    %i = alloca i32, align 4
    %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds (i8*, i8** %call, i32 0, i32 0), i32 0, i32 %res, align 4)
    store i32 0, i32* %res, align 4
    store i32 0, i32* %i, align 4
    br label %for.cond

  for.cond:
    %0 = load i32, i32* %i, align 4
    %cmp = icmp slt i32 %0, 10000
    br i1 %cmp, label %for.body, label %for.end

  for.body:
    %1 = load i32, i32* %res, align 4
    %inc = add nsw i32 %1, 1
    store i32 %inc, i32* %res, align 4
    br label %for.inc

  for.inc:
    %2 = load i32, i32* %i, align 4
    %inc1 = add nsw i32 %2, 1
    store i32 %inc1, i32* %i, align 4
    br label %for.cond

  for.end:
    %3 = load i32, i32* %res, align 4
    ret i32 %3
}
```
Sometimes “may” 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 ($d \text{ dom } n$) if every control flow from the start node to $n$ goes through $d$. Every node dominates itself.

What is the relation between instructions within a basic block?

What are the dominators of basic blocks 1 and 2?

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 p1, ..., pk

**Observation 1:** if d dominates each pi (1<=i<=k), then d dominates n

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

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

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 \( n \).
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 $d$

```
B: if (par1 > 5)
C: varX = par1 + 1
D: print(varX)
```

Immediate post-dominator tree

How to compute post-dominators?
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 $d$

```
B: if (par1 > 5)
C: varX = par1 + 1
C2: ...
D: print(varX)
```
Another example of CFA (and CFT)

goto L1
L1: call printf()
return

call printf()
return

The two basic blocks can be merged

call printf()
return

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;
}
#endif CRAZY
printf(“Yep”);
#endif
return;

clang myfile.c –DCRAZY –o myprog