Simone Campanoni
simonec@eecs.northwestern.edu

CFA

de analysis
and
transformation
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
Let us start by looking at how to iterate over instructions of a function in LLVM
Functions and instructions

```cpp
bool runOnFunction (Function &F) override {
    errs() << "Hello LLVM World at \"runOnFunction\"\n";
    return false;
}
```

runOnFunction’s job is to analyze/transform a function F
... by analyzing/transforming its instructions
Functions and instructions

```
#include "llvm/IR/InstIterator.h"

for (auto& inst : instructions(F)){
  errs() << inst << "\n";
}
```

runOnFunction’s job is to analyze/transform a function F ... by analyzing/transforming its instructions

What is the instruction that will be executed after `inst`?

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

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

int x = a + 1
int x = a + 1

tmp = a > 5
branch_ifnot tmp L1
tmp = a > 5
branch_ifnot tmp L1

x++
branch L1
x++
branch L1

What is the next instruction executed?

L1: x--
L1: x--

L2: return x
L2: return x

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

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
Storing order ≠ executing order

Control Flow Analyses are designed to understand the possible execution paths

To improve/transform the code, we need to analyze the execution paths
This is the job of Control Flow Analysis
• To further see the need of CFAs, we can look at their uses (e.g., code transformations)
  • Constant propagation

• Before further showing the need of CFAs
  • let me introduce a few concepts,
  • then we’ll further motivate CFAs using a code transformation,
  • and then we’ll talk about CFAs
Code transformation

**Code transformation:**
An algorithm that takes code as input and it generates new code as output

![Diagram showing code transformation from version A to B]

**Semantically-preserving code transformation:**
A code transformation that always generates code that is guaranteed to have the same semantics of the code given as input.
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: 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
int main (int argc, char *argv[])
{
    int y = argc + 2;
    printf("%d", 2*argc + 2);
    return 1;
}
```

```
$ ./myprog 2
6
$ echo $?
```

```
$ ./myprog 2
6
$ echo $?
```
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 = 42;
    return y;
}
```

Our new code transformation:

```c
int main (int argc, char *argv[])
{
    int y = 42;
    return y;
}
```

*We have preserved the semantics of the original code!*

```c
int main (int argc, char *argv[])
{
    int y = 42;
    return 42;
}
```
Program semantic

Program semantic: Input -&gt; 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 = 42;
    int x = y;
    if (argc &gt; 20)
        y = 81;
    return x + y;
}

Our transformation needs to understand how the execution flows through the instructions to preserve the semantics!

int main (int argc, char *argv[])
{
    int y = 42;
    int x = 42;
    if (argc &gt; 20)
        y = 81;
    return x + 42;
}

Our new code transformation

We haven’t preserved the semantics of the original code

This is ok!

When this is executed
Control flows

**Control flow**: sequence of instructions in a program that may execute in that order

*(common simplification: we ignore data values and arithmetic operations)*

Control flow:

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

Control flow:

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

Understanding the control flows is the job of the Control Flow Analyses
Let us go deeper in the need for control flow analysis for code transformation.

Let us introduce an actual code transformation implemented by all compilers: constant propagation.

... but first, we need to introduce a few definitions.
Variables and constants

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

Constants

Variable definitions

Variable uses
Code transformation example: constant propagation

```c
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;
}
```

Replace a variable use with a constant while preserving the original code semantics.
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 and
  • 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

• Control Flow Analysis discovers facts about control flows
A few concepts before our first 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

What about calls?
- Program exits
- Exceptions

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
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
void @Z8functionv() #8 {
  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)
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 $x$ ($I_x$)
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();

ENTRY

Basic Block 1
1: int a = 1;
2: int b = 2;
T: if (b == 2)

Basic Block 2
1: ++b

Basic Block 3
1: int c = 3;
2: int d = 4;

Basic Block 4
T: while (a < 5)

Basic Block 5
1: ++a

Basic Block 6
1: int e = 5;
2: int f = 6;

[EXIT]
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";
    }
}
```
Output of the LLVM pass of the previous slide:
It is now the time to introduce your first control flow analysis
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 ($d \ dom \ n$) iff 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 is the relation between instructions in different basic blocks?

It depends on the CFG
In other words, dominators depend on the control flows
Dominators

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

What are the dominators of basic blocks 1 and 2?

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

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

What are now the dominators of basic blocks 1, 2, and 3?
Now that we know what we want to obtain (the dominance binary relation between basic blocks),

let us define an algorithm (a CFA) that computes it
A CFA to find dominators

Consider a block $n$ with $k$ predecessors $p_1, \ldots, p_k$

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 $p_i$

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

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

This is your first CFA

Notice: this CFA does not depend on values and/or operations/operators
Dominance

CFG

1

2

3

Dominators

1

2

3
We can now introduce new concepts based on the dominator relation.
Strict dominance

Definition:
a node $d$ strictly dominates $n$ iff
\begin{itemize}
  \item $d$ dominates $n$ and
  \item $d$ is not $n$
\end{itemize}
Immediate dominators

**Definition:** the immediate dominator of a node $n$ is the unique node that strictly dominates $n$ but does not strictly dominate another node that strictly dominates $n$.
Immediate dominators

**Definition:** the immediate dominator of a node $n$ is the unique node that strictly dominates $n$ but does not strictly dominate another node that strictly dominates $n$.

![CFG](image1.png)
![Strict dominators](image2.png)
![Immediate dominators](image3.png)

Dominator tree
#include "llvm/IR/Dominators.h"

bool runOnFunction (Function &F) override {
    errs() << "=== Dominators\n";

    DominatorTree &DT = getAnalysis<DominatorTreeWrapperPass>().getDomTree();

    for (auto & bb : F){
        auto inst = bb.begin();
        errs() << *inst << "\n";

        auto instNode = DT.getNode(&bb);

        for (auto child : instNode->getChildren()){
            auto dominatedBB = child->getBlock();
            auto dominatedInst = dominatedBB->begin();
            errs() << "-" << " " << dominatedInst << "\n";
        }
    }

    return false;
}

void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.addRequired<DominatorTreeWrapperPass>();
    AU.setPreservesAll();
}
Dominators in LLVM

```c
#include "llvm/IR/Dominators.h"

bool runOnFunction (Function &F) override {
  errs() << "== Dominators\n";
  DominatorTree &DT = getAnalysis<DominatorTreeWrapperPass>().getDomTree();
  for (auto &bb : F) {
    auto inst = bb.begin();
    errs() << *inst << "\n";
    auto instNode = DT.getNode(&bb);
    for (auto child : instNode->getChildren()){
      auto dominatedBB = child->getBlock();
      auto dominatedInst = dominatedBB->begin();
      errs() << " -> " << "dominatedInst " << "\n";
    }
  }
  return false;
}

void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.addRequired<DominatorTreeWrapperPass>();
  AU.setPreservesAll();
}
```
Dominators in LLVM: example 2

```cpp
#include "llvm/IR/Dominators.h"

bool runOnFunction (Function &F) override {
  errs() << "=== Dominators\n";

  DominatorTree &DT = getAnalysis<DominatorTreeWrapperPass>().getDomTree();

  for (auto & bb : F){
    auto inst = bb.begin();
    errs() << *inst << "\n";

    auto instNode = DT.getNode(&bb);
    for (auto child : instNode->getChildren()){
      auto dominatedBB = child->getBlock();
      auto dominatedInst = dominatedBB->begin();
      errs() << " -> " << *dominatedInst << "\n";
    }

  return false;
}

void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.addRequired<DominatorTreeWrapperPass>();
  AU.setPreservesAll();
}

3 int main(int argc, char *argv[]) {
  printf("START\n");

  if (argc > 0){
    printf("THEN\n");

    if (argc > 20){
      printf("Inside THEN\n");

      if (argc > 40){
        printf("Inside the inside of THEN\n");
      }
    }

  printf("END\n");
  return 1;
}
```
#include "llvm/IR/Dominators.h"

bool runOnFunction (Function &F) override {
  errs() << "--- Dominators\n";
  DominatorTree DT = getAnalysis<

DominatorTreeWrapperPass>().getDomTree();

  for (auto & bb : F){
    auto inst = bb.begin();
    errs() << "\n";
    auto instNode = DT.getNode(&bb);

    for (auto child : instNode->getChildren()){
      auto dominatedBB = child->getBlock();
      auto dominatedInst = dominatedBB->begin();
      errs() << "\n";
    }
  }
  return false;
}

void getAnalysisUsage (AnalysisUsage &AU) const override {
  AU.addRequired<

DominatorTreeWrapperPass>();
  AU.setPreservesAll();
}
LLVM-specific notes for dominators

• bool DominatorTree::dominates (...)
  • bool dominates (Instruction *i, Instruction *j)
    Return true if the basic block that includes i is an immediate dominator of the basic block that includes j
  • bool dominates (Instruction *i, BasicBlock *b)
    Return true if the basic block that includes i is an immediate dominator of b

• If the first argument (either instruction or basic block) is not reachable from the entry point of the function, return false
• If the second argument (either instruction or basic block) is not reachable from the entry point of the function, return true
It is now the time to introduce your second control flow analysis
Post-dominators

**Assumption:** Single exit node in CFG

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

$$
\begin{align*}
\text{B: } & \text{if (par1 > 5)} \\
\text{C: } & \text{varX = par1 + 1} \\
\text{D: } & \text{print(varX)}
\end{align*}
$$

How to compute post-dominators?
Post-dominators

**Assumption:** Single exit node in CFG

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

---

**Diagram:**

```
  B
   ↓
  C
   ↓
C2
  ↓
D
```

**Immediate post-dominator tree:**

```
  D
   /\  \\
  C2  B
      /\  \\
  C   C
      /\  \\
 C2  D
```

**Code snippets:**

- B: if (par1 > 5)
- C: varX = par1 + 1
- C2: ...
- D: print(varX)
Post dominators in LLVM

```c
#include "llvm/Analysis/PostD dominators.h"

bool runOnFunction (Function &F) override {
  errs() << "--- Post dominators\n";

  PostDominatorTree& PDT = getAnalysis<PostDominatorTreeWrapperPass>().getPostDTree();

  for (auto& bb : F) {
    auto inst = bb.begin();
    errs() << *inst << "\n";

    auto instNode = PDT.getNode(&bb);

    for (auto child : instNode->getChildren()) {
      auto dominatedBB = child->getBlock();
      auto dominatedInst = dominatedBB->begin();
      errs() << " -> " << *dominatedInst << "\n";
    }

  }

  return false;
}

void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.addRequired<PostDominatorTreeWrapperPass>();
  AU.setPreservesAll();
}

int main(int argc, const char *argv[]) {
  printf("START\n");

  if (argc > 0) {
    printf("THEN\n");
  } else {
    printf("ELSE\n");
  }

  printf("END\n");

  if (argc == 0) {
    return 0;
  }

  return 1;
}
Post dominators in LLVM

```c
#include "llvm/Analysis/PostDominators.h"

bool runOnFunction (Function &F) override {
  errs() << "=== Post dominators\n";

  PostDominatorTree& PDT = getAnalysis<PostDominatorTreeWrapperPass>().getPostDomTree();

  for (auto& bb : F){
    auto inst = bb.begin();
    errs() << inst << "\n";

    auto instNode = PDT.getNode(&bb);

    for (auto child : instNode->getChildren()){,
      auto dominatedBB = child->getBlock();
      auto dominatedInst = dominatedBB->begin();
      errs() << "-> " << *dominatedInst << "\n";
    }
  }

  return false;
}

void getAnalysisUsage(AnalysisUsage &AU) const override {  
  AU.addRequired<PostDominatorTreeWrapperPass>();
  AU.setPreservesAll();
}
LLVM-specific notes for post dominators

• bool PostDominatorTree::dominates (…)
  • bool dominates (Instruction *i, Instruction *j)
    Return true if the basic block that includes i is an immediate post-dominator of the basic block that includes j
  • bool dominates (Instruction *i, BasicBlock *b)
    Return true if the basic block that includes i is an immediate post-dominator of b

• If the first argument (either instruction or basic block) is not reachable from the entry point of the function, return false

• If the second argument (either instruction or basic block) is not reachable from the entry point of the function, return true
LLVM-specific notes for *dominators

DominatorTreeBase
::bool dominates(...)
Another example of CFA (and CFT)

A homework of this class could be the following one: design and implement an algorithm to implement this CFA

- CFA: it says whether it is safe to merge two basic blocks
- CFT: it merges only the basic block pairs identified by the CFA

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

Existing LLVM pass: simplifycfg
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

```c
... If (b == 2){
    return;
}
... b == 2
```

```bash
clang myfile.c -DCRAZY -o myprog
```

```c
#include <stdio.h>

#define CRAZY 1

int main() {
    if (b == 2)
        printf("Yep");
    return;
}
```
Now that you know CFA and LLVM

• It’s time for the homework H1