Coded analysis and transformation

DFA

Simone Campanoni
simone.campanoni@northwestern.edu
Data Flow Analysis outline

• Concepts needed by most code analyses

• Why do we need DFA? (opportunities)

• Introduction to DFA (concepts)

• A DFA example: reaching definitions (concept application)

• Implementation of DFA (actual implementation)
Variables and constants

x = 0;

y = x + 1;

Constants

Variable definitions

Variable uses
Now that we know variables, we can talk about how data stored in them could evolve through the code flows.
Now that we know variables, we can talk about how **data flows** through the code
Data flows

```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;
}
```

Data flows from a definition to its uses
Data flow examples

```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;
}
```

Understanding data flows require understanding the possible sequence of instructions that could be executed at run-time control flows.
Control flows

**Control flow:** sequence of instructions in a program that may execute at run-time in that order.

*(common simplification: we ignore 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++;
}
```
How can we automatically identify and represent the control flows?

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

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

The iteration order of instructions isn’t the execution one
We cannot use iteration order to analyze data flows
Storing order ≠ executing order

int myF (int a){
    int x = a + 1;
    if (a > 5){
        x++;  // A storing order
    } else {
        x--;  // Another (correct) storing order
    }
    return x;  // Another (correct) storing order
}

int x = a + 1
int tmp = a > 5
branch ifnot tmp L1  // A storing order
x++  // Another (correct) storing order
branch L2  // Another (correct) storing order
L1: x--  // Another (correct) storing order
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
How can we automatically identify and represent the control flows?

We could represent the control flows using a directed graph:
- Node: instruction
- Direct edge: points to the possible next instruction that could be executed at run-time
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
- The storing sequence is the execution order in a basic block
Basic blocks in compilers

• Automatically identified
  • Algorithm:

```java
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 compilers

• 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
Basic blocks in compilers

• 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
- Infinite loops in callees
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
Basic blocks in LLVM

• 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)
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

- All function variables are declared at the beginning of the function
- A variable access becomes a memory access
Basic blocks in LLVM in action

function() {
    int a = 1; //Sequential instructions
    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;
}

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
How can we automatically identify and represent the control flows?

We could represent the control flows using a directed graph:
- **Node**: instruction
- **Direct edge**: points to the possible next instruction that could be executed at run-time
Control Flow Graph (CFG)

• A CFG is a graph G = <Nodes, Edges>
• Nodes: Basic blocks
• Edges: \((x, y) \in \text{Edges}\) if and only if after executing the last instruction of basic block \(x\) \((l_x)\) the first instruction of the basic block \(x\) \((l_x)\) may execute.
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 instructions within a basic block

```cpp
auto nextInstruction = i->getNextNode();
auto prevInstruction = i->getPrevNode();
```
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 an instruction to its successors

Let’s say we want to iterate over the successors of $i$ so from $i$ to $j$ and $k$

How can we do it?

```cpp
for (auto succBB : successors(bb)){
```
Navigating the CFG in LLVM: From an instruction to its successors

```c++
for (auto &b : F){
    auto i = b.getTerminator();
    errs() << *i << "\n";
    for (auto succBB : successors(b)){
        auto firstInstOfSuccBB = succBB->front();
    }
}
```
H0/tests

Output of the LLVM pass of the previous slide:
Now that we know how to traverse over the CFG, we can introduce the first code transformation
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;
}
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); // Replace a variable use with a constant
        x = x + b*y; // while preserving the original code semantics
    }
    return x;
}
```
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*0;
    }
    return x;
}

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

Understanding requires understanding how the value of a variable could evolve over time—data flows, and this is the job of data flow analyses.
Data Flow Analysis outline

• Concepts needed by most code analyses

• Why do we need DFA? (opportunities)

• Introduction to DFA (concepts)

• A DFA example: reaching definitions (concept application)

• Implementation of DFA (actual implementation)
The need for DFAs

• We constantly need to improve programs (e.g., speed, energy efficiency, memory requirements)
• We constantly need to identify opportunities
• After having found an opportunity (e.g., propagating constants), you need to ask yourself:
  • What do I need to know to take advantage of this opportunity? (e.g., I need to know the possible values a given variable might have at a given point in the program)
  • How can I automatically compute this information? Often the solution relies on understanding how data flows through the code. This is often done by designing custom DFAs
Let us go deeper in the need for data flow analysis for code transformation

Let us introduce an actual code transformation implemented by all compilers: constant propagation
Transformation: constant propagation
Analysis: reaching definition DFA

• Opportunity: this code is “better” compared to this
  Which information do I need to know if it is safe to replace \( b \) with 2
  Among all possible run time control flows, what are the latest definitions of \( b \)?
  What are the possible values \( b \) can have at run time?

Which information do I need to know if it is safe to replace \( b \) with 2

Among all possible run time control flows, what are the latest definitions of \( b \)?

What are the possible values \( b \) can have at run time?
Constant propagation

- Find an instruction $i$ that defines a variable with a constant expression
  
  *Instruction $i$: $b = \text{CONSTANT\_EXPRESSION}$*

- Replace an use of $b$ in an instruction $j$ with that $\text{CONSTANT\_EXPRESSION}$ if
  
  - All control flows to $j$ includes $i$
  - There are no intervening definition of that variable
Constant propagation: code example

```c
int sumcalc (int a, int b, int N){
int x,y;
x = 0;
y = 0;
if (a > b){
   x = x + N;
}
if (b > N){  return y;}  // Data-flow analysis is a collection of techniques for compile-time reasoning about the run-time values
return x;
}
```

We need to analyze the “data-flows” of a program and represent them explicitly.
But constant propagation (CP) has been done already ...  

• CP has been already designed and implemented  

• Why should we study it?  
  Why don’t we design and implement all possible transformations and analyses in a compiler and move on?  

• It is always possible to invent new/better transformations  

  Full employment theorem for compiler writers
Since it is always possible to improve transformations, let us learn the typical approach to create new data-flow analyses that will drive the innovation.
New transformations and analyses

• New transformations (often) need to understand specific and new code properties related to how data might change through the code
  • So we need to know how to design a new data flow analysis that identifies these new code properties

• Generic recipe
  Data flow analysis (DFA):
  traverse the CFGs collecting information about what may happen at run time (Conservative approximation)
  Transformation:
  Modify the code based on the result of data flow analysis (Correctness guaranteed by the conservative approximation of DFA)

Data flow value
New transformations and analyses

• Generic recipe

**Data flow analysis (DFA):**
traverse the CFGs collecting information about
what may happen at run time (Conservative approximation)

**Transformation:**
Modify the code based on the result of data flow analysis
(Correctness guaranteed by the conservative approximation of DFA)
Data Flow Analysis outline

• Concepts needed by most code analyses

• Why do we need DFA? (opportunities)

• Introduction to DFA (concepts)

• A DFA example: reaching definitions (concept application)

• Implementation of DFA (actual implementation)
Concepts

• Static and dynamic control flows

• Data flow abstraction

• Data flow values

• Transfer functions

• GEN, KILL, IN, OUT sets
Static program vs. dynamic execution

- **Static:**
  Finite program

- **Dynamic:**
  Can have infinitely many possible control flows

- **Data flow analysis abstraction:**
  For each point in a program:
  combine information about all possible run-time instances of the same program point.

```
... = b
```

```
b = 1
```

```
b = 2
```

```
If (b > N)
```

```
b = b + 1
```

**Data flow analysis (DFA):**
traverse the CFGs collecting information about what may happen at run time
(Conservative approximation)

**What are the possible values of b?**
Example of data-flow questions

• What are the possible values of b just before an instruction “... = b”?
• Which instruction defines the value used in “... = b”?
Example of data-flow questions

• What are the possible values of b just before an instruction “... = b”?
• Which instruction defines the value used in “... = b”?
• Has the expression “a * b” been computed before another instruction? (“... = a * b”)
• What are the instructions that might read the value produced by an instruction “b = ...”?
• What are the instructions that will (must) read the value produced by an instruction “b = ...”?
• ...

Data-flow expressed in CFG

**Data-flow value:**
set of all possible program states that can be observed at a given program point
e.g., all definitions in the program that might have been executed before that point

**Data-flow analysis**
computes IN and OUT sets by computing the DFA-specific transfer functions
Transfer functions

- Let $i$ be an instruction: $IN[i]$ and $OUT[i]$ are the set of data-flow values before and after the instruction $i$ of a program.
- A transfer function $fs$ relates the data-flow values before and after an instruction $i$.
- In a forward data-flow problem
  \[ OUT[i] = fs(IN[i]) \]
- In a backward data-flow problem
  \[ IN[i] = fs(OUT[i]) \]

$fs$ is DFA-specific.
Transfer function internals: $Y[i] = \text{fs}(X[i])$

- It relies on information that reaches $i$.
- It transforms such information to propagate the result to the rest of the CFG.
- To do so, it relies on information specific to $i$.
  - Encoded in GEN[$i$], KILL[$i$].
  - $\text{fs}$ uses GEN[$i$] and KILL[$i$] to compute its output.
- GEN[$i$] and KILL[$i$] are DFA-specific and (typically) data/control flow independent!
DFA steps

1) Define the DFA-specific sets GEN[i] and KILL[i], for all i and without looking at the control flows

2) Implement the DFA-specific transfer function $fs$

3) Compute all IN[i] and OUT[i] following a DFA-generic algorithm

$\text{OUT}[i] = fs ( \text{IN}[i] )$

$\text{IN}[i] = fs ( \text{OUT}[i] )$

*Compilers typically have a data flow framework/engine to help developing new DFAs (we will not rely on such framework/engine for this class)*
Data Flow Analysis outline

• Concepts needed by most code analyses

• Why do we need DFA? (opportunities)

• Introduction to DFA (concepts)

• A DFA example: reaching definitions (concept application)

• Implementation of DFA (actual implementation)
Before introducing the reaching definition DFA, let us go back to the previous example to formalize new terminology
Optimization example: constant propagation

int sumcalc (int a, int b, int N){
    int x,y;
    x = 0;
    y = 0;
    if (a > b){
        x = x + N;
    }
    if (b > N){ return y;}
    return x;
}

Information needed just before an instruction $i$:
what are the definitions that might execute before $i$?

$IN[return y] = \{y=0\}$
$IN[return x] = \{x=0, x = x + N\}$
Let us define the concept of “reaching” more formally
Data-flow example: reaching definitions

• A definition $D$ reaches a program point $X$ if there is a control flow from $D$ to $X$ such that the variable defined by $D$ is not redefined along that path.

$\text{D: } v = 0$
$\text{J: call printf(...)}$
$\text{X: } ... = v ...$

$\text{GEN}[D] = \{D\}$

$\text{IN}[X] = \{D\}$

$D$ reaches $X$
Data-flow example: reaching definitions

• A definition $D$ reaches a program point $X$ if there is a control flow from $D$ to $X$ such that $D$ is not killed along that path.

$$D: \text{v} = 0$$
$$J: \text{call printf(...)}$$
$$X: \ldots = \text{v} \ldots$$

$D$ reaches $X$

$IN[X] = \{D\}$

$GEN[D] = \{D\}$
Data-flow example: reaching definitions

• A definition $D$ reaches a program point $X$ if there is a control flow from $D$ to $X$ such that $D$ is not killed along that path.

\[\begin{align*}
D & : v = 0 \\
J & : v = v + n \\
X & : ... = v ...
\end{align*}\]

$D$ does not reach $X$
$D$ is not in $\text{IN}[X]$

$\text{GEN}[D] = \{D\}$ \hspace{1cm} $\text{KILL}[D] = \{J\}$

$\text{KILL}[J] = \{D\}$

$\text{KILL}[i] = \text{data flow value removed because of } i$

$\text{GEN}[i] = \text{data flow value added by } i$
Data-flow example: reaching definitions

• A definition $D$ reaches a program point $X$ if there is a control flow from $D$ to $X$ such that $D$ is not killed along that path.

\begin{align*}
\ldots
\mathcal{D}: v &= 0 \\
\ldots
\mathcal{J}: v &= v + n \\
\ldots
\mathcal{X}: \ldots &= v
\end{align*}

- $\text{IN}[X] = \{J\}$
- $\text{GEN}[J] = \{J\}$
- $\text{KILL}[J] = \{D\}$
- $\text{KILL}[D] = \{J\}$
- $\text{GEN}[D] = \{D\}$
Data-flow example: reaching definitions

• A definition $D$ reaches a program point $X$ if there is a control flow from $D$ to $X$ such that $D$ is not killed along that path.

• The reaching definition data-flow problem for a flow graph is to compute all definitions that reach an instruction $i$ (i.e., $\text{IN}[i]$, $\text{OUT}[i]$) for all $i$ in that graph.
Computing INs and OUTs

GEN[0] = {}  IN = {}  { x=0 } = OUT
GEN[1] = {1}  0: int x,y  1: x = 0  2: y = 0  3: return x
GEN[2] = {2}  0: int x,y  1: x = 0  2: y = 0  3: return x
GEN[0] = {}  IN = {}  { x=0 } = OUT

- Forward or backward?
  OUT[i] = fs (IN[i])

- GEN[i] = what i generates
- KILL[i] = what i kills (invalidates)

- fs within a basic block?
  Let i be an instruction and p be its only predecessor
  IN[i] = OUT[p]
  OUT[i] = GEN[i] U (IN[i] – KILL[i])

Local reaching definitions
Data-flow example: reaching definitions

• A definition \( d \) reaches a program point \( X \) if there is a path from \( d \) to \( X \) such that \( d \) is not killed along that path

• The data-flow problem for a flow graph is to compute \( \text{IN}[i] \) and \( \text{OUT}[i] \) for all \( i \) in that graph

\[
\text{IN}[i] = \bigcup_{p \text{ a predecessor of } i} \text{OUT}[p]
\]
\[
\text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] - \text{KILL}[i])
\]
Data Flow Analysis outline

• Concepts needed by most code analyses

• Why do we need DFA? (opportunities)

• Introduction to DFA (concepts)

• A DFA example: reaching definitions (concept application)

• Implementation of DFA (actual implementation)
• So far, we have defined **data-flow equations** (i.e., IN and OUT equations)

• How can we actually compute them?

• Main problem:
  ➢ input of equation IN depends on output of equation OUT
    \[ \text{IN}[i] = \bigcup_{p \text{ a predecessor of } i} \text{OUT}[p] \]
  ➢ Output of equation OUT depends on input of equation IN
    \[ \text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] - \text{KILL}[i]) \]
We break all possible dependence cycles by iteratively computing all IN and OUT sets until a fixed point is reached.
Steps for iterative algorithm

• Compute GEN and KILL sets for all instructions without using the CFG
  • GEN and KILL sets will not change anymore

• Compute IN and OUT sets with an iterative algorithm
  
do{
    Compute IN and OUT sets for all instructions
  } while (any IN or OUT set changes from the previous iteration)
Iterative algorithm for reaching definitions

- Given GEN[i], KILL[i] for all instructions i, we compute IN[i] and OUT[i] for all i

```
for (each instruction i)  IN[i] = OUT[i] = { }
```

do {
    for (each instruction i) {
        IN[i] = \cup \text{p a predecessor of i} \text{ OUT}[p];
        OUT[i] = GEN[i] \cup (IN[i] - KILL[i]);
    }
} while (changes to any OUT occur)
Reaching definition in action

Why do we need to reach a fixed point?

IN[0] = {} \rightarrow \quad \text{OUT}[0] = {} \quad \text{IN}[1] = {} \rightarrow \quad \text{OUT}[1] = 1 \quad \text{IN}[2] = \{1\} \rightarrow \quad \text{OUT}[2] = \{1,2\} \quad \text{IN}[3] = \{1,2\} \rightarrow \quad \text{OUT}[3] = \{1,2\} \quad \text{IN}[4] = \{1,2\} \rightarrow \quad \text{OUT}[4] = \{2,4\} \quad \text{IN}[5] = \{1,2,4\} \rightarrow \quad \text{OUT}[5] = \{1,2,4\}

KILL[0] = {} \rightarrow \quad \text{KILL}[1] = \{4\} \quad \text{KILL}[2] = {} \rightarrow \quad \text{KILL}[3] = {} \quad \text{KILL}[4] = \{1\} \quad \text{KILL}[5] = {}

IN[i] = \bigcup \text{a predecessor} \quad \text{OUT}[p]

\text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] - \text{KILL}[i])
Now that you know reaching definition

• It’s time for the homework H1
• What we learned was for forward data-flow analysis

\[ \text{OUT}[s] = fs(\text{IN}[s]) \]

for (each instruction \(i\)) \(\text{IN}[i] = \text{OUT}[i] = \{\}\); 
do{} 
  for (each instruction \(i\)) 
    \[ \text{IN}[i] = \bigcup_{p \text{ a predecessor of } i} \text{OUT}[p]; \]
    \[ \text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] - \text{KILL}[i]); \]
  \} while (changes to any \text{OUT} occur)

• What about backward data-flow analysis?

\[ \text{IN}[s] = fs(\text{OUT}[s]) \]
for (each instruction $i$) \( \text{IN}[i] = \text{OUT}[i] = \{ \} \);

do {
    for (each instruction $i$) {
        \( \text{IN}[i] = \text{fs}_p \text{ a predecessor of } i \) (\( \text{OUT}[p] \))
        \( \text{OUT}[i] = \text{fs}(\text{IN}[i]) \)
    }
} while (changes to any \( \text{OUT} \) occur)
Backward DFA

for (each instruction \(i\)) \(\text{IN}[i] = \text{OUT}[i] = \{\}\);

do {
  for (each instruction \(i\)) {
    \(\text{OUT}[i] = fs\) a successor of \(i\) (\(\text{IN}[s]\))
    \(\text{IN}[i] = fs(\text{OUT}[i])\)
  }
}
while (changes to any \(\text{IN}\) occur)
Always have faith in your ability

Success will come your way eventually

Best of luck!