Code analysis and transformation

More DFAs

Simone Campanoni
simonec@eecs.northwestern.edu
Simone Campanoni
Tenure-track assistant professor
Department of Electrical Engineering and Computer Science
Northwestern University

Welcome!

I am a tenure-track assistant professor at the Electrical E Northwestern University.
My research focuses on code compilation challenges for both processors. I address these challenges by co-designing compilers they target, and programming languages.

I live in Evanston, Illinois with my wife, Noelle, our son, Marco,

Research: Our work has been accepted for a PACT 2017,

LinkedIn Twitter

Code Analysis and Transformation

Description
Fast, highly sophisticated code analysis and code transformation tools are essential for modern software development. Before releasing its mobile apps, Facebook submits them to a tool called Infer that finds bugs by static analysis, i.e., without even having to run the code, and guides developers in fixing them. Google Chrome and Mozilla Firefox analyze and optimize JavaScript code to make browsers acceptably responsive. Performance-critical systems and application software would be impossible to build and evolve without compilers that derive highly optimized machine code from high-level source code that humans can understand. Understanding what modern code analysis and transformation techniques can and can’t do is a prerequisite for research on both software engineering and computer architecture since hardware relies on software to realize its potential. In this class, you will learn the fundamentals of code analysis and transformation, and you will apply them by extending LLVM, a compiler framework now in production use by Apple, Adobe, Intel and other industrial and academic enterprises.

Syllabus

Material
This class takes materials from three different books (listed in the syllabus) as well as a few research papers. The result is a set of slides, notes, and code. Some lectures rely on code and notes (not slides). Next you can find only slides; the rest of the material is available only on Canvas.

<table>
<thead>
<tr>
<th>Week number</th>
<th>First lecture</th>
<th>Second lecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 0</td>
<td>Welcome</td>
<td>Introduction to LLVM</td>
</tr>
<tr>
<td>Week 1</td>
<td>Control Flow Analysis</td>
<td>CFA in LLVM</td>
</tr>
<tr>
<td>Week 2</td>
<td>Data Flow Analysis</td>
<td>Static Single Assignment form</td>
</tr>
<tr>
<td>Week 3</td>
<td>Data Flow Analysis and their uses</td>
<td>Foundations of Data Flow Analysis</td>
</tr>
<tr>
<td>Week 4</td>
<td>Dependencies</td>
<td>Dependencies</td>
</tr>
<tr>
<td>Week 5</td>
<td>Dependencies</td>
<td>Introduction to inter-procedural CAT</td>
</tr>
<tr>
<td>Week 6</td>
<td>Inter-procedural CAT</td>
<td>Inter-procedural analysis example: VLLPA</td>
</tr>
<tr>
<td>Week 7</td>
<td>Introduction to loops</td>
<td>Loops</td>
</tr>
<tr>
<td>Week 8</td>
<td>Introduction to loop transformations</td>
<td>Loop transformations</td>
</tr>
<tr>
<td>Week 9</td>
<td>State-of-the-art CAT</td>
<td>Competition</td>
</tr>
</tbody>
</table>

Hall of Fame
Students extend the industrial-strength compiler clang using their own advanced code analyses and transformations developed during this class. At the end of the class, the resulting compilers compete and the names of the student(s) that designed and built the best compiler are reported below.

<table>
<thead>
<tr>
<th>Year</th>
<th>Name</th>
<th>Year</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017</td>
<td>TBD</td>
<td>TBD</td>
<td>TBD</td>
</tr>
</tbody>
</table>
Outline

• Reaching definition and constant propagation

• More DFAs and related transformations

• DFAs without assumptions

• Other uses of DFA
Constant propagation: problem definition

Given a program, we would like to know for every point in that program, which variables have constant values, and which ones do not.

A variable has a constant value at a certain point in the CFG if every execution that reaches that point sees that variable holding the same constant value.

\begin{align*}
i: x &= 0 \\
j: N &= N + 1 \\
k: Z &= x + N
\end{align*}
Reaching definition summary

• Reaching definition data-flow analysis computes IN[i] and OUT[i] for every instruction i

• IN[i] (OUT[i]) includes definitions that reach just before (just after) instruction i

• Each IN/OUT set contains a mapping for every variable in the program to a “value”
Constant propagation

- For a use of variable v in statement n, n: x = ... v ...
- If the definitions of v that reach n are all of the form d: v = c [c a constant]
- then replace the use of v in n with c

Do you see any problem?
What about in The CAT language?

**Constant propagation problem?**

1: int x,y
3: y = 0
4: If (a > b)
5: x=5
6: If (b > N)
7: return y
8: return 5

**IN[3]={}**
**IN[4]=\{3\}**
**IN[5]=\{3\}**
**IN[6]=\{3,5\}**
**IN[7]=\{3,5\}**
**IN[8]=\{3,5\}**

**Better solutions?**
- New analysis
- Customize reaching definitions

**Is this correct?**

```c
CATData CAT_create_signed_value (int64_t value);
```
Copy propagation: problem definition

Given a CFG, we would like to know for every point in the program, if a variable contains always the same value of another one.

1: \( x = y \)
2: \( a = 5 \)
3: \( b = x + 3 \)

Copy propagation

1: \( x = y \)
2: \( a = 5 \)
3: \( b = y + 3 \)

How can we implement this transformation?
Reaching definition summary

- Reaching definition data-flow analysis computes IN[i] and OUT[i] for every instruction i.

- IN[i] (OUT[i]) includes definitions that reach just before (just after) instruction i.

- Each IN/OUT set contains a mapping for every variable in the program to a “value”;
Copy propagation

• For a use of variable v in statement n, n: x = ... v ...
• If the definitions of v that reach n are all of the form d: v = z [z is another variable]
• then replace the use of v in n with z

Do you see any problem?
Thinking about what we have done

• What’s the value of these propagations?
  • Constant propagation: less variable uses
    **Redundant use of variables**
  • Copy propagation: less variable uses
    **Redundant use of variables**

• Redundancy operations are the principal source of optimization in compilers
Outline

• Reaching definition and constant propagation

• More DFAs and related transformations

• DFAs without assumptions

• Other uses of DFA
Dead code elimination: problem definition

Given a program, we would like to know statements/instructions that do not influence the program at all (i.e., dead code)

How can we identify dead code?
Liveness analysis

A variable is **live** at a particular point in the program if its value at that point will be used in the future (dead, otherwise)

• To compute liveness at a given point of a CFG, we need to look at instructions that will be executed next

• How to use variable liveness information for eliminating dead-code?
Example of variable liveness and dead-code elimination

What are in IN/OUT sets?

IN[0]  = {}
OUT[0] = {a}
IN[1]  = {a}
OUT[1] = {a, b}
IN[2]  = {a, b}
OUT[2] = {b}
IN[3]  = {b}
OUT[3] = {b}
IN[4]  = {b}
OUT[4] = {}

Is there dead-code?
Liveness analysis

A variable $v$ is live at a given point of a program $p$ if

- Exist a directed path from $p$ to a use of $v$ and
- that path does not contain any definition of $v$

- Is liveness data-flow analysis forward or backward?
  - Liveness flows backwards through the CFG, because the behavior at future nodes determines liveness at a given node

- What are the elements in data flow values? variables

$\text{GEN}[i]$ = variables used by $i$ $\quad \text{KILL}[i]$ = variable defined by $i$

$\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] – \text{KILL}[i])$

$\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]$
Liveness analysis

A variable is **live** at a particular point in the program if its value at that point will be used in the future (dead, otherwise)

- Another use: register allocation
- A program contains an unbounded number of variables
  - Must execute on a machine with a bounded number of registers
  - Two variables can use the same register if they are never in use at the same time
- EECS 322 Compiler Construction
Common sub-expression elimination: problem definition

Given a program, we would like to know for every point in the program, which expressions are available.

1: \( y = x + 3 \)
2: \( b = x + 3 \)
Available expressions

• What are the elements in data-flow sets?
• GEN and KILL?
• Forward or backward?
• IN and OUT?

\[
\text{IN}[i] = \bigcap_{p \text{ a predecessor of } i} \text{OUT}[p]
\]

\[
\text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] - \text{KILL}[i])
\]

• How to use available expressions for eliminating redundant code?

1: \( y = x + 3 \)
2: \( b = x + 3 \)

i: \( y = x + 3 \)
j: \( z = x + 3 \)
k: \ldots
So far ...

- Reaching definitions
- Variable liveness
- Available expressions

  - Constant propagation
  - Copy propagation
  - Dead-code elimination
  - Common sub-expression elimination
Dominators

**Definition:** a basic block $d$ dominates $n$ 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 elements for data flow values? GEN ? KILL ? IN ? OUT?
Outline

• Reaching definition and constant propagation

• More DFAs and related transformations

• DFAs without assumptions

• Other uses of DFA
What about function parameters?

... let’s compute the reaching definition analysis
Which information is missing?

```c
int myFunction(int a, int b)
{
    if (a > b)
    {
        a = 5;
    }
    return a;
}
```

IN[0a] : { }
OUT[0a] : {0a}

IN[0b] : {0a}
OUT[0b] : {0a,0b}

IN[1] : {0a,0b}

IN[2] : {0a,0b}
OUT[2] : {2,0b}

IN[3] = {2,0a,0b}

CP algorithm replaces “a” with “5” in instruction 3!

Can we exploit SSA properties?
What about escaped variables?

... let’s compute the reaching definition analysis
Which information is missing?

int myFunction (void){
    int a;
    int *p = f(&a);
    if (a > b){
        a = 5;
    } else {
        *p = 6;
    }
    return a;
}
What about memory references?

```c
int myFunction (void){
    int a;
    int *p = f();
    a = *p;
    return a;
}
```
Outline

• Reaching definition and constant propagation

• More DFAs and related transformations

• DFAs without assumptions

• Other uses of DFA
Identifying software bugs

1: int x, y
2: y = 0
3: If (a > b)

4: x=5

5: If (b > N)

6: return y

7: return x

• “x” can be undefined at instruction 7
• Can we design an analysis to identify this problem and notify a developer about this bug?
• Let’s define precisely the problem
  • Conservativeness
• What are the data flow values?
  • GEN[i] = ?
  • KILL[i] = ?
  • IN[i] and OUT[i] ?
Identifying software bugs (2)

1: int x
2: call f(&x)
3: If (a > b)
4: x=5
5: return x

• What about now?
• Let’s define precisely the problem
  • Conservativeness
  • Warnings vs. errors
Data-flow analysis: food for thought

- Correctness: is the answer ALWAYS correct?
- Meaning: what is exactly the meaning of the answer?
- Precision: how good is the answer?
- Convergence:
  - Will the analysis ALWAYS terminate?
  - Under what conditions does the iterative algorithm converge?
- Speed: how long does it take to converge in the worst case?