More DFAs

Simone Campanoni
simonec@eecs.northwestern.edu
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.

```
i: x = 0
j: N = N + 1
k: Z = x + N
```
Reaching definition summary

• Reaching definition data-flow analysis computes \(\text{IN}[i]\) and \(\text{OUT}[i]\) for every instruction \(i\)

• \(\text{IN}[i]\) (\(\text{OUT}[i]\)) 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 by instruction 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?
Constant propagation problem?

What about in The CAT language?

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

IN[3]={}
IN[4]={3}

5: x=5

5: x=5
IN[5]={3}

6: If (b > N)

IN[6]={3,5}

7: return y
IN[7]={3,5}

8: return 5
IN[8]={3,5}

CATData CAT_new (int64_t value);

Is this correct?

Better solutions?
- New analysis
- Customize reaching definitions
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: \( b = x + 3 \)

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? How can we fix it? (3 points)
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

Front-end → ... → Back-end
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?

<table>
<thead>
<tr>
<th>IN[0]</th>
<th>OUT[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{a}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IN[1]</th>
<th>OUT[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a}</td>
<td>{a, b}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{a, b}</td>
<td>{b}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{b}</td>
<td>{b}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{b}</td>
<td>{}</td>
</tr>
</tbody>
</table>

Is there dead-code?

0: \(a = 0\)
1: \(b = a + 1\)
2: \(a = a + b\)
3: \(d = b * 2\)
4: \(\text{return } b\)
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 $i$  $\text{KILL}[i]=$ variable defined by $i$

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

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

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

Do you see any redundancy?
Available expressions

• What are the elements in data-flow sets?
• GEN and KILL (2 points)?
• Forward or backward?
• IN and OUT (2 points)?
  \[
  \text{IN}[i] = \bigcap_{p \text{ a predecessor of } i} \text{OUT}[p]
  \]
  \[
  \text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] - \text{KILL}[i])
  \]
• How to use available expressions for eliminating redundant code?

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

- Reaching definitions
  - Constant propagation
- Variable liveness
  - Copy propagation
- Available expressions
  - Dead-code elimination
  - Common sub-expression elimination
**Dominator**

**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? (1 point)
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?

```
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}

Can we exploit SSA properties?

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

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

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

CP algorithm replaces “a” with “5” in instruction 3!
What about escaped variables?

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

```c
int myFunction (void){
    int a;
    int *p = f(&a);
    if (a > b){
        a = 5;
    } else {
        *p = 6;
    }
    return a;
}
```

IN[6] = {2, 3}
OUT[6] = {2, 3, 6}

5: a = 5
OUT[5] = {2, 3, 5}

IN[7] = {2, 3, 5, 6}

CP algorithm replaces “a” with “5” in instruction 7!
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?