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
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.
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 \( v \) 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?

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

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?
Undefined behavior: a funny interpretation

• **Undefined behavior** is the result of executing a program whose behavior is unpredictable

• Undefined behavior results in whatever compilers want the program being compiled to do
  even *to make demons fly out of your nose*
  • Undefined behavior is often referred to as *nasal demons*
Constant propagation in SSA (in LLVM)

If you want the conventional CP semantics
- Skip undef

IN[\%R2]=\{\%3, \%6, \%7\}
Constant propagation in SSA (in LLVM)

%E
%3 = icmp sgt %a, %b
br %3 %T %F

%T
br %F

%F
%6 = phi [undef, %E] [5, %T]
%7 = icmp sgt %b, %N
br %7 %R1 %R2

IN[%R2]={%3, %6, %7}

%R2 return 5

%R1 return 0
Constant propagation for CAT

- Undefined values enable optimizations
- What about in the CAT language?
- CATData CAT_new (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.

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 $n$, $n$: $x = \ldots v \ldots$
• 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
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)

1: y = ...
2: x = y
3: return x

Copy propagation

1: y = ...
2: x = y
3: return y

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?

```
0: a = 0
1: b = a + 1
2: a = a + b
3: d = b * 2
4: return b
```

```
0: a = 0
1: b = 0 + 1
4: return 1
```
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

  GEN[i] = variables used by i
  KILL[i] = variable defined by i

  IN[i] = GEN[i] ∪ (OUT[i] − KILL[i])
  OUT[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]

```
i: a = 5
...
j: a = v + 1
...
k: x = a + 1
```
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
- CS 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

\begin{align*}
1: & \ y = x + 3 \\
2: & \ b = x + 3
\end{align*}

\begin{align*}
1: & \ y = x + 3 \\
2: & \ b = y
\end{align*}

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

- Variable liveness
  - Copy propagation
  - Dead-code elimination
  - Common sub-expression elimination

- Available expressions
**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? (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}
IN[1] = {0a,0b}

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

Can we exploit SSA properties?

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

IN[6] = {2, 3}  OUT[6] = {2, 3, 6}
IN[7] = {2, 3, 5, 6}  7: return a
IN[5] = {2, 3}  OUT[5] = {2, 3, 5}
1: int a
2: int *q = &a
3: int *p = f(q)
4: If (a > b)
5: a = 5
6: *p = 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?