The spirit of my lectures
a.k.a. my teaching philosophy

• I’ll describe problems/opportunities
• I’ll describe concepts required to solve these problems
  (take advantage of these opportunities)
• I’ll describe their solutions that are based on these concepts

• I’ll describe new problems/opportunities
  • You’ll apply concepts/solutions learned in these lectures
    to solve the new problems/opportunities
    • Required to pass the homework
Data Flow Analysis outline

• 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 ad-hoc DFAs
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 $2$?
Constant propagation

• Find an instruction \( i \) that defines a variable with a constant expression

\( \text{Instruction } i: \quad b = \text{CONSTANT\_EXPRESSION} \)

• 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

CFA

DFA
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;
    }
    return x;
}
```

Data-flow analysis is a collection of techniques for compile-time reasoning about the run-time values.

We need to know the “data-flow” of the program.
Understanding the data-flow requires understanding the control-flow.
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
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)
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)

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?

\[
i: \quad b = 2
\]

\[
... \quad ...
\]

\[
j: \quad ... = b
\]

Data flow value
Data Flow Analysis outline

• 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.

What are the possible values of $b$?

```
Data flow analysis (DFA):
traverse the CFGs collecting information about
what may happen at run time
(Conservative approximation)
```
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., the data-flow value of reaching definition DFA is all definitions in the program that might have been executed before that point
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: $\text{IN}[i]$ and $\text{OUT}[i]$ 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

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

• In a backward data-flow problem

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

$f$s is DFA-specific
Transfer function internals: $Y[i] = fs(X[i])$

- It relies on information that reaches $i$

- It transforms such information to propagate the result to the rest of the CFG
  
  $GEN[i] = \text{data flow value added by } i$
  
  $KILL[i] = \text{data flow value removed because of } i$

- To do so, it relies on information specific to $i$
  - Encoded in $GEN[i]$, $KILL[i]$
  - $fs$ 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

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

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

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

• 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.
Optimization example: constant propagation

```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;}
    return x;
}
```

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

达 \[ \text{IN[return } y \text{]} = \{ y=0 \} \]
达 \[ \text{IN[return } x \text{]} = \{ x=0, x = x + N \} \]
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.

```
D: v = 0
J: call printf(...)
X: ... = v ...
```

- $D \text{ reaches } X$
- $\text{IN}[X] = \{D\}$
- $\text{GEN}[D] = \{D\}$
- $\text{GEN}[i] = \text{data flow value added by } i$
- killed

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

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

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

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

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

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

Example:

$D: v = 0$

$J: v = v + n$

$X: \ldots = v \ldots$

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

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

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

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

$J$ 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.

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

GEN[0] = {}  
GEN[1] = {1}  
GEN[2] = {2}  
GEN[3] = {}  

\[
\begin{align*}
\text{IN} &= \{ \} \\
0: \text{int } x,y \\
1: x = 0 \\
2: y = 0 \\
3: \text{If } (a > b)
\end{align*}
\]

\[
\text{OUT} = \{ x = 0 \}
\]

- Forward or backward?
  \[
  \text{OUT}[i] = fs(\text{IN}[i])
\]

- \( \text{GEN}[i] = \text{what } i \text{ generates} \)
- \( \text{KILL}[i] = \text{what } i \text{ kills (invalidates)} \)

- \( fs \) within a basic block?
- Let \( i \) be an instruction and \( p \) be its only predecessor
  \[
  \begin{align*}
  \text{IN}[i] &= \text{OUT}[p] \\
  \text{OUT}[i] &= \text{GEN}[i] \cup \text{(IN}[i] - \text{KILL}[i])
  \end{align*}
  \]

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.

\[
\text{IN}[i] = \bigcup_{\text{a predecessor of } i} \text{OUT}[\rho]
\]

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

\[
\text{IN}[\text{entry}] = \{ \}
\]
Data Flow Analysis outline

• 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
    \[ \text{IN}[i] = \bigcup_{\text{a predecessor of } i} \text{OUT}[p] \]
  - Output of equation OUT depends on input of equation IN
    \[ \text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] \setminus \text{KILL}[i]) \]
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
  • 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] = { }

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

**GEN**

<table>
<thead>
<tr>
<th>i</th>
<th>GEN[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{}</td>
</tr>
<tr>
<td>1</td>
<td>{1}</td>
</tr>
<tr>
<td>2</td>
<td>{2}</td>
</tr>
<tr>
<td>3</td>
<td>{}</td>
</tr>
<tr>
<td>4</td>
<td>{4}</td>
</tr>
<tr>
<td>5</td>
<td>{}</td>
</tr>
</tbody>
</table>

**KILL**

<table>
<thead>
<tr>
<th>i</th>
<th>KILL[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{}</td>
</tr>
<tr>
<td>1</td>
<td>{4}</td>
</tr>
<tr>
<td>2</td>
<td>{}</td>
</tr>
<tr>
<td>3</td>
<td>{}</td>
</tr>
<tr>
<td>4</td>
<td>{1}</td>
</tr>
<tr>
<td>5</td>
<td>{}</td>
</tr>
</tbody>
</table>

**IN**

<table>
<thead>
<tr>
<th>i</th>
<th>IN[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{}</td>
</tr>
<tr>
<td>1</td>
<td>{}</td>
</tr>
<tr>
<td>2</td>
<td>{1}</td>
</tr>
<tr>
<td>3</td>
<td>{}</td>
</tr>
<tr>
<td>4</td>
<td>{1,2}</td>
</tr>
<tr>
<td>5</td>
<td>{1,2,4}</td>
</tr>
</tbody>
</table>

**OUT**

<table>
<thead>
<tr>
<th>i</th>
<th>OUT[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{}</td>
</tr>
<tr>
<td>1</td>
<td>{1}</td>
</tr>
<tr>
<td>2</td>
<td>{1,2}</td>
</tr>
<tr>
<td>3</td>
<td>{}</td>
</tr>
<tr>
<td>4</td>
<td>{2,4}</td>
</tr>
<tr>
<td>5</td>
<td>{1,2,4}</td>
</tr>
</tbody>
</table>

**Algorithm**

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

**Query**

Why do we need to reach a fixed point?  

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

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

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

\(\text{do} \{\)

\(\text{for (each instruction i) } \{\)

\(\text{IN}[i] = \bigcup_{p \text{ a predecessor of } i} \text{OUT}[p];\)

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

\} \}

\(\text{while (changes to any OUT occur)}\)

- Memory representation of data flow values
- Operations performed on them
- What is an element in a set?
Can we optimize the analysis?

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

... that’s a lot of iterations repeated for each while iteration

Is this always necessary?
for (each basic block B) \ IN[B] = OUT[B] = \{ \};

\textbf{do \{} \\
\quad for (each basic block B) \{ \\
\quad\quad \text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]; \\
\quad\quad \text{OUT}[B] = \text{GEN}[B] \cup (\text{IN}[B] \setminus \text{KILL}[B]); \\
\quad\}\}

\textbf{\}} while (changes to any OUT occur)

\textbf{Optimization 1: basic blocks}

Contains \textbf{all} definitions in block B that are \textbf{visible} immediately after B

Contains \textbf{all} definitions killed by instructions in block B
for (each basic block B) IN[B] = OUT[B] = { };
do {
  for (each basic block B) {
    IN[B] = \bigcup_{P \text{ a predecessor of } B} OUT[P];
    OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B]);
  }
} while (changes to any OUT occur)
Optimization 2: bit-sets

• Assign a bit to each element that might be in the set
  • Union: bitwise OR
  • Intersection: bitwise AND
  • Subtraction: bitwise NEGATE and AND

• Fast implementation
  • 64 elements packed to each word on today’s commodity processors
  • AND and OR are single machine code instructions (single cycle latency)
Optimization 3: work list

for (each basic block B) \( \text{IN}[B] = \text{OUT}[B] = \{ \} \); do {
for (each basic block B) {
\( \text{IN}[B] = \bigcup_{\text{a predecessor of } B} \text{OUT}[P]; \)
\( \text{OUT}[B] = \text{GEN}[B] \cup (\text{IN}[B] \setminus \text{KILL}[B]); \)
}
} while (changes to any OUT occur)
Optimization 3: work list

OUT[ENTRY] = { };
for (each basic block B other than ENTRY)  OUT[B] = { };
workList = all basic blocks
while (workList isn’t empty)
    B = pick and remove a block from workList
    oldOUT = OUT[B]
    IN[B] = \bigcup_{p \text{ a predecessor of } B} OUT[p];
    OUT[B] = GEN[B] \cup (IN[B] − KILL[B]);
    if (oldOut != OUT[B]) workList = workList U {all successors of B}
Optimization 4: block order

\[ \text{OUT}[\text{ENTRY}] = \{ \}; \]

for (each basic block \( B \) other than \( \text{ENTRY} \)) \( \text{OUT}[B] = \{ \} \);

workList = all basic blocks

while (workList isn’t empty)

\[ \text{B} = \text{pick and remove a block from workList} \]

oldOUT = OUT[B]

\[ \text{IN}[B] = \bigcup_{\text{p a predecessor of } B} \text{OUT}[p]; \]

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

if (oldOut \(!=\) OUT[B]) \( \text{workList} = \text{workList} \cup \{ \text{all successors of } B \} \)

}
• 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?

\[ \text{IN}[s] = fs(\text{OUT}[s]) \]
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?
Now that you know reaching definition

• It’s time for the homework H2