CMSC 430
Introduction to Compilers
Fall 2018

Data Flow Analysis
Data Flow Analysis

• A framework for proving facts about programs

• Reasons about lots of little facts

• Little or no interaction between facts
  ▪ Works best on properties about how program computes

• Based on all paths through program
  ▪ Including infeasible paths

• Operates on control-flow graphs, typically
x := a + b;
y := a * b;
while (y > a) {
    a := a + 1;
    x := a + b
}
Control-Flow Graph w/Basic Blocks

\[
\begin{align*}
x & := a + b; \\
y & := a \times b; \\
\text{while } (y > a + b) \{ \\
    & \quad a := a + 1; \\
    & \quad x := a + b \\
\}
\end{align*}
\]

- Can lead to more efficient implementations
- But more complicated to explain, so...
  - We’ll use single-statement blocks in lecture today
Example with Entry and Exit

\[ x := a + b; \]
\[ y := a \times b; \]
\[ \text{while } (y > a) \{ \]
\[ \quad a := a + 1; \]
\[ \quad x := a + b \]
\[ \} \]

- All nodes without a (normal) predecessor should be pointed to by entry
- All nodes without a successor should point to exit
Notes on Entry and Exit

• Typically, we perform data flow analysis on a function body

• Functions usually have
  - A unique entry point
  - Multiple exit points

• So in practice, there can be multiple exit nodes in the CFG
  - For the rest of these slides, we’ll assume there’s only one
  - In practice, just treat all exit nodes the same way as if there’s only one exit node
Available Expressions
Available Expressions

• An expression $e$ is available at program point $p$ if
  ▪ $e$ is computed on every path to $p$, and
  ▪ the value of $e$ has not changed since the last time $e$ was computed on the paths to $p$
Available Expressions

• An expression $e$ is available at program point $p$ if
  ▪ $e$ is computed on every path to $p$, and
  ▪ the value of $e$ has not changed since the last time $e$ was computed on the paths to $p$

• Optimization
  ▪ If an expression is available, need not be recomputed
    - (At least, if it’s still in a register somewhere)
Data Flow Facts

• Is expression e available?
• Facts:
  ▪ a + b is available
  ▪ a * b is available
  ▪ a + 1 is available

entry

x := a + b

y := a * b

y > a

a := a + 1

x := a + b

exit
Gen and Kill

• What is the effect of each statement on the set of facts?

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := a + b</td>
<td>a + b</td>
<td></td>
</tr>
<tr>
<td>y := a * b</td>
<td>a * b</td>
<td></td>
</tr>
<tr>
<td>a := a + 1</td>
<td></td>
<td>a + 1, a + b, a * b</td>
</tr>
</tbody>
</table>

```
entry
x := a + b
y := a * b
y > a
a := a + 1
x := a + b
exit
```
Computing Available Expressions

entry

\[ x := a + b \]

\[ y := a \times b \]

\[ y > a \]

\[ a := a + 1 \]

\[ x := a + b \]
Computing Available Expressions

entry

x := a + b

y := a * b

y > a

a := a + 1

x := a + b

exit
Computing Available Expressions

∅

entry

x := a + b

y := a * b

y > a

y + a

x := a + b

{a + b}

exit
Computing Available Expressions

∅

entry

x := a + b

y := a * b

y > a

a := a + 1

x := a + b

{a + b}

exit
Computing Available Expressions

∅

entry

{x := a + b}

y := a * b

{y > a, y > a}

a := a + 1

x := a + b

exit
Computing Available Expressions

- \( a + b \)
- \( a + b, a \times b \)
- \( y > a \)
- \( a := a + 1 \)
- \( x := a + b \)
Computing Available Expressions

\[
\emptyset \rightarrow \text{entry} \\
\{a + b\} \rightarrow x := a + b \\
\{a + b, a \times b\} \rightarrow y := a \times b \\
\{a + b, a \times b\} \rightarrow y > a \\
\{a + b, a \times b\} \rightarrow a := a + 1 \\
\text{exit} \rightarrow x := a + b
\]
Computing Available Expressions

∅

entry

{x := a + b}

{a + b}

y := a * b

{a + b, a * b}

{a + b, a * b}

y > a

{a + b, a * b}

a := a + 1

y > a

exit

x := a + b

x := a + b
Computing Available Expressions

∅

entry

x := a + b

y := a * b

y > a

a := a + 1

x := a + b

∅

{a + b}

{a + b, a * b}

{a + b, a * b}

∅

exit
Computing Available Expressions

\[
\varnothing \quad \xrightarrow{\text{entry}} \quad \begin{cases} x := a + b \\ y := a \times b \end{cases} \quad \xrightarrow{\{a + b\}} \quad \{a + b, a \times b\} \quad \xrightarrow{\{a + b, a \times b\}} \quad \{a + b, a \times b\} \quad \xrightarrow{\varnothing} \quad \begin{cases} a := a + 1 \\ x := a + b \end{cases} \quad \xrightarrow{\text{exit}}
\]
Computing Available Expressions

∅

entry

x := a + b

{a + b}

y := a * b

{a + b, a * b}

y > a

{a + b, a * b}

∅

a := a + 1

{a + b, a * b}

x := a + b

{a + b}

exit
Computing Available Expressions

\[ \emptyset \]

entry

\[ x := a + b \]

\[ \{a + b\} \]

\[ y := a * b \]

\[ \{a + b, a * b\} \]

\[ y > a \]

\[ \{a + b, a * b\} \]

\[ a := a + 1 \]

\[ \emptyset \]

\[ x := a + b \]

\[ \{a + b\} \]

exit
Computing Available Expressions

∅

entry

x := a + b

{a + b}

y := a * b

{a + b, a * b}

y > a

{a + b, a * b}

∅

a := a + 1

{a + b, a * b}

x := a + b

{a + b}

exit
Computing Available Expressions

∅

entry

x := a + b

y := a * b

y > a

a := a + 1

x := a + b

∅

{a + b}

{a + b, a * b}

{a + b}

exit

{a + b}
Computing Available Expressions

- \( \emptyset \) to entry
- \( \{ a + b \} \) to \( x := a + b \)
- \( \{ a + b, a \times b \} \) to \( y := a \times b \)
- \( \emptyset \) to \( y > a \)
- \( \{ a + b \} \) to exit
- \( \{ a + b \} \) to \( a := a + 1 \)
- \( \{ a + b \} \) to \( x := a + b \)
Computing Available Expressions

∅

entry

{x := a + b}

{a + b}

{a + b, a * b}

{a + b, a * b}

y := a * b

y > a

{a + b, a * b}

∅

a := a + 1

{x := a + b}

{a + b, a * b}

exit

y := a * b

{a + b}

y > a

{a + b}

∅

a := a + 1

{x := a + b}

{a + b}
Computing Available Expressions

entry

\[ x := a + b \]

\[ y := a \times b \]

\[ y > a \]

\[ a := a + 1 \]

\[ x := a + b \]
**Terminology**

- A *joint point* is a program point where two branches meet.

- Available expressions is a *forward must* problem:
  - Forward = Data flow from *in* to *out*
  - Must = At join point, property must hold on all paths that are joined.
Data Flow Equations

- Let $s$ be a statement
  - $\text{succ}(s) = \{ \text{immediate successor statements of } s \}$
  - $\text{pred}(s) = \{ \text{immediate predecessor statements of } s \}$
  - $\text{in}(s) =$ program point just before executing $s$
  - $\text{out}(s) =$ program point just after executing $s$

- $\text{in}(s) = \bigcap_{s' \in \text{pred}(s)} \text{out}(s')$

- $\text{out}(s) = \text{gen}(s) \cup (\text{in}(s) - \text{kill}(s))$
  - Note: These are also called transfer functions
Liveness Analysis
Liveness Analysis

- A variable $v$ is *live* at program point $p$ if
  - $v$ will be used on some execution path originating from $p$...
  - before $v$ is overwritten
Liveness Analysis

• A variable $v$ is live at program point $p$ if
  ■ $v$ will be used on some execution path originating from $p$...
  ■ before $v$ is overwritten

• Optimization
  ■ If a variable is not live, no need to keep it in a register
  ■ If variable is dead at assignment, can eliminate assignment
Data Flow Equations

• Available expressions is a forward must analysis
  ▸ Data flow propagate in same dir as CFG edges
  ▸ Expr is available only if available on all paths

• Liveness is a backward may problem
  ▸ To know if variable live, need to look at future uses
  ▸ Variable is live if used on some path

• \( \text{out}(s) = \bigcup_{s' \in \text{succ}(s)} \text{in}(s') \)

• \( \text{in}(s) = \text{gen}(s) \cup (\text{out}(s) - \text{kill}(s)) \)
Gen and Kill

- What is the effect of each statement on the set of facts?

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := a + b</td>
<td>a, b</td>
<td>x</td>
</tr>
<tr>
<td>y := a * b</td>
<td>a, b</td>
<td>y</td>
</tr>
<tr>
<td>y &gt; a</td>
<td>a, y</td>
<td></td>
</tr>
<tr>
<td>a := a + 1</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>
Computing Live Variables

\[
\begin{align*}
x & := a + b \\
y & := a \times b \\
y & > a \\
a & := a + 1 \\
x & := a + b
\end{align*}
\]
Computing Live Variables

\[ x := a + b \]

\[ y := a \times b \]

\[ y > a \]

\[ a := a + 1 \]

\[ x := a + b \]
Computing Live Variables

\{x, y, a\}

\[x := a + b\]

\[y := a \times b\]

\[y > a\]

\[a := a + 1\]

\[x := a + b\]
Computing Live Variables

\[ x := a + b \]

\[ y := a \times b \]

\[ y > a \]

\[ a := a + 1 \]

\[ x := a + b \]
Computing Live Variables

```
x := a + b

y := a * b

y > a

a := a + 1

x := a + b
```
Computing Live Variables

\[
\begin{align*}
&x := a + b \\
y := a \times b \\
y > a \\
a := a + 1 \\
x := a + b
\end{align*}
\]
Computing Live Variables

1. \( x := a + b \)
2. \( y := a \times b \)
3. \( y > a \)
4. \( a := a + 1 \)
5. \( x := a + b \)
Computing Live Variables

\[ x := a + b \]
\[ y := a \times b \]
\[ y > a \]
\[ a := a + 1 \]
\[ x := a + b \]
Computing Live Variables

{x, y, a, b}

x := a + b

{y, a, b}
y := a * b

y > a

{y, a, b}
a := a + 1

{y, a, b}
x := a + b

{y, a, b}

{y, a, b}

{a}

{y, a, b}

{x}

{x, y, a, b}
Computing Live Variables

\[
x := a + b \\
y := a * b \\
y > a \\
a := a + 1 \\
x := a + b
\]
Computing Live Variables

\[
x := a + b
\]

\[
y := a \times b
\]

\[
y > a
\]

\[
a := a + 1
\]

\[
x := a + b
\]
Computing Live Variables

\{a, b\} → x := a + b

\{x, a, b\} → y := a * b

\{x, y, a, b\} → y > a

\{y, a, b\} → a := a + 1

\{y, a, b\} → x := a + b

\{x, y, a, b\} → \{x\}
**Very Busy Expressions**

- An expression \( e \) is very busy at point \( p \) if
  - On every path from \( p \), expression \( e \) is evaluated before the value of \( e \) is changed.

- Optimization
  - Can hoist very busy expression computation.

- What kind of problem?
  - Forward or backward?
  - May or must?
Very Busy Expressions

• An expression $e$ is very busy at point $p$ if
  ▪ On every path from $p$, expression $e$ is evaluated before the value of $e$ is changed

• Optimization
  ▪ Can hoist very busy expression computation

• What kind of problem?
  ▪ Forward or backward? backward
  ▪ May or must?
Very Busy Expressions

• An expression $e$ is **very busy** at point $p$ if
  - On every path from $p$, expression $e$ is evaluated before the value of $e$ is changed

• Optimization
  - Can hoist very busy expression computation

• What kind of problem?
  - Forward or backward? **backward**
  - May or must? **must**
Reaching Definitions

- A definition of a variable \( v \) is an assignment to \( v \)
- A definition of variable \( v \) reaches point \( p \) if
  - There is some path from the definition to \( p \) such that there is no intervening assignment to \( v \) on the path

- Also called def-use information

- What kind of problem?
  - Forward or backward?
  - May or must?
Reaching Definitions

• A definition of a variable $v$ is an assignment to $v$

• A definition of variable $v$ reaches point $p$ if
  - There is some path from the definition to $p$ such that there is no intervening assignment to $v$ on the path

• Also called def-use information

• What kind of problem? forward
  - Forward or backward?
  - May or must?
Reaching Definitions

- A *definition* of a variable $v$ is an assignment to $v$
- A definition of variable $v$ reaches point $p$ if
  - There is some path from the definition to $p$ such that there is no intervening assignment to $v$ on the path

- Also called def-use information

- What kind of problem?  
  - Forward or backward?  
  - May or must?
Most data flow analyses can be classified this way:
- A few don’t fit: bidirectional analysis
- Lots of literature on data flow analysis

<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>Reaching definitions</td>
<td>Available expressions</td>
</tr>
<tr>
<td>Backward</td>
<td>Live variables</td>
<td>Very busy expressions</td>
</tr>
</tbody>
</table>
Solving data flow equations

• Let’s start with forward may analysis
  - Dataflow equations:
    - \( in(s) = \bigcup_{s' \in \text{pred}(s)} out(s') \)
    - \( out(s) = gen(s) \cup (in(s) - kill(s)) \)
  
• Need algorithm to compute \( in \) and \( out \) at each stmt

• Key observation: \( out(s) \) is \textit{monotonic} in \( in(s) \)
  - \( gen(s) \) and \( kill(s) \) are fixed for a given \( s \)
  - If, during our algorithm, \( in(s) \) grows, then \( out(s) \) grows
  - Furthermore, \( out(s) \) and \( in(s) \) have max size

• Same with \( in(s) \)
  - in terms of \( out(s') \) for predecessors \( s' \)
Solving data flow equations (cont’d)

• Idea: fixpoint algorithm
  ▪ Set \(\text{out(entry)}\) to emptyset
    - E.g., we know no definitions reach the entry of the program
  ▪ Initially, assume \(\text{in(s)}, \text{out(s)}\) empty everywhere else, also
  ▪ Pick a statement \(s\)
    - Compute \(\text{in(s)}\) from predecessors’ \(\text{out(s)}\)
    - Compute new \(\text{out(s)}\) for \(s\)
  ▪ Repeat until nothing changes

• Improvement: use a worklist
  ▪ Add statements to worklist if their \(\text{in(s)}\) might change
  ▪ Fixpoint reached when worklist is empty
### Forward May Data Flow Algorithm

<table>
<thead>
<tr>
<th>out(entry) = ∅</th>
</tr>
</thead>
<tbody>
<tr>
<td>for all other statements s</td>
</tr>
<tr>
<td>\hspace{1cm} out(s) = ∅</td>
</tr>
<tr>
<td>W = all statements \hspace{1cm} // worklist</td>
</tr>
<tr>
<td>while W not empty</td>
</tr>
<tr>
<td>\hspace{1cm} take s from W</td>
</tr>
<tr>
<td>\hspace{1cm} in(s) = \cup_{s' \in \text{pred}(s)} out(s')</td>
</tr>
<tr>
<td>\hspace{1cm} temp = gen(s) \cup (in(s) - \text{kill}(s))</td>
</tr>
<tr>
<td>\hspace{1cm} if temp \neq out(s) then</td>
</tr>
<tr>
<td>\hspace{1cm} \hspace{1cm} out(s) = temp</td>
</tr>
<tr>
<td>\hspace{1cm} \hspace{1cm} W := W \cup \text{succ}(s)</td>
</tr>
<tr>
<td>\hspace{1cm} end</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>


## Generalizing

<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward</strong></td>
<td>in(s) = ( \bigcup_{s' \in \text{pred}(s)} \text{out}(s') )</td>
<td>in(s) = ( \bigcap_{s' \in \text{pred}(s)} \text{out}(s') )</td>
</tr>
<tr>
<td></td>
<td>out(s) = gen(s) ( \cup ) (in(s) - kill(s))</td>
<td>out(s) = gen(s) ( \cup ) (in(s) - kill(s))</td>
</tr>
<tr>
<td></td>
<td>out(entry) = ( \emptyset )</td>
<td>out(entry) = ( \emptyset )</td>
</tr>
<tr>
<td></td>
<td>initial out elsewhere = ( \emptyset )</td>
<td>initial out elsewhere = {all facts}</td>
</tr>
<tr>
<td><strong>Backward</strong></td>
<td>out(s) = ( \bigcup_{s' \in \text{succ}(s)} \text{in}(s') )</td>
<td>out(s) = ( \bigcap_{s' \in \text{succ}(s)} \text{in}(s') )</td>
</tr>
<tr>
<td></td>
<td>in(s) = gen(s) ( \cup ) (out(s) - kill(s))</td>
<td>in(s) = gen(s) ( \cup ) (out(s) - kill(s))</td>
</tr>
<tr>
<td></td>
<td>in(exit) = ( \emptyset )</td>
<td>in(exit) = ( \emptyset )</td>
</tr>
<tr>
<td></td>
<td>initial in elsewhere = ( \emptyset )</td>
<td>initial in elsewhere = {all facts}</td>
</tr>
</tbody>
</table>
Forward Analysis

\[
\text{out(entry)} = \emptyset \\
\text{for all other statements } s \quad \text{out(s)} = \emptyset \\
W = \text{all statements} \quad \text{// worklist} \\
\text{while W not empty} \\
\quad \text{take s from W} \\
\quad \text{in(s)} = \bigcup_{s' \in \text{pred(s)}} \text{out(s')} \\
\quad \text{temp} = \text{gen(s)} \cup (\text{in(s)} - \text{kill(s)}) \\
\quad \text{if temp} \neq \text{out(s)} \text{ then} \\
\quad \quad \text{out(s)} = \text{temp} \\
\quad \text{W := W} \cup \text{succ(s)} \\
\text{end} \\
\text{end}
\]

May

\[
\text{out(entry)} = \emptyset \\
\text{for all other statements } s \quad \text{out(s)} = \text{all facts} \\
W = \text{all statements} \\
\text{while W not empty} \\
\quad \text{take s from W} \\
\quad \text{in(s)} = \bigcap_{s' \in \text{pred(s)}} \text{out(s')} \\
\quad \text{temp} = \text{gen(s)} \cup (\text{in(s)} - \text{kill(s)}) \\
\quad \text{if temp} \neq \text{out(s)} \text{ then} \\
\quad \quad \text{out(s)} = \text{temp} \\
\quad \text{W := W} \cup \text{succ(s)} \\
\text{end} \\
\text{end}
\]

Must
Backward Analysis

in(exit) = ∅
for all other statements s
  in(s) = ∅
W = all statements
while W not empty
  take s from W
    out(s) = \bigcup_{s' \in \text{succ}(s)} \text{in}(s')
    temp = \text{gen}(s) \cup (\text{out}(s) - \text{kill}(s))
    if temp \neq \text{in}(s) then
      in(s) = temp
      W := W \cup \text{pred}(s)
    end
end

May

in(exit) = ∅
for all other statements s
  in(s) = all facts
W = all statements
while W not empty
  take s from W
    out(s) = \bigcap_{s' \in \text{succ}(s)} \text{in}(s')
    temp = \text{gen}(s) \cup (\text{out}(s) - \text{kill}(s))
    if temp \neq \text{in}(s) then
      in(s) = temp
      W := W \cup \text{pred}(s)
    end
end

Must
Practical Implementation

• Represent set of facts as bit vector
  ■ Fact_i represented by bit i
  ■ Intersection = bitwise and, union = bitwise or, etc

• “Only” a constant factor speedup
  ■ But very useful in practice
Basic Blocks

• Recall a basic block is a sequence of statements s.t.
  ■ No statement except the last in a branch
  ■ There are no branches to any statement in the block except the first

• In some data flow implementations,
  ■ Compute gen/kill for each basic block as a whole
    - Compose transfer functions
  ■ Store only in/out for each basic block
  ■ Typical basic block ~5 statements
    - At least, this used to be the case...
Order Matters

- Assume forward data flow problem
  - Let $G = (V, E)$ be the CFG
  - Let $k$ be the height of the lattice

- If $G$ acyclic, visit in topological order
  - Visit head before tail of edge

- Running time $O(|E|)$
  - No matter what size the lattice
Order Matters — Cycles

- If $G$ has cycles, visit in reverse postorder
  - Order from depth-first search
  - (Reverse for backward analysis)

- Let $Q = \text{max \# back edges on cycle-free path}$
  - Nesting depth
  - Back edge is from node to ancestor in DFS tree

- In common cases, running time can be shown to be $O((Q+1)|E|)$
  - Proportional to structure of CFG rather than lattice
Flow-Sensitivity

- Data flow analysis is *flow-sensitive*
  - The order of statements is taken into account
  - I.e., we keep track of facts per program point

- Alternative: *Flow-insensitive* analysis
  - Analysis the same regardless of statement order
  - Standard example: types
    - /* x : int */ x := ... /* x : int */
What happens at a function call?
- Lots of proposed solutions in data flow analysis literature

In practice, only analyze one procedure at a time

Consequences
- Call to function kills all data flow facts
- May be able to improve depending on language, e.g., function call may not affect locals
More Terminology

- An analysis that models only a single function at a time is *intraprocedural*
- An analysis that takes multiple functions into account is *interprocedural*
- An analysis that takes the whole program into account is *whole program*

- Note: *global* analysis means “more than one basic block,” but still within a function
  - Old terminology from when computers were slow...
Data Flow Analysis and The Heap

- Data Flow is good at analyzing local variables
  - But what about values stored in the heap?
  - Not modeled in traditional data flow

- In practice: \( *x := e \)
  - Assume all data flow facts killed (!)
  - Or, assume write through \( x \) may affect any variable whose address has been taken

- In general, hard to analyze pointers
Proebsting’s Law
Proebsting’s Law

• Moore’s Law: Hardware advances double computing power every 18 months.
Proebsting’s Law

• Moore’s Law: Hardware advances double computing power every 18 months.

• Proebsting’s Law: Compiler advances double computing power every 18 years.
Proebsting’s Law

• Moore’s Law: Hardware advances double computing power every 18 months.

• Proebsting’s Law: Compiler advances double computing power every 18 years.

  ▪ Not so much bang for the buck!
DFA and Defect Detection

- LCLint - Evans et al. (UVa)
- METAL - Engler et al. (Stanford, now Coverity)
- ESP - Das et al. (MSR)
- FindBugs - Hovemeyer, Pugh (Maryland)
  - For Java. The first three are for C.
- Many other one-shot projects
  - Memory leak detection
  - Security vulnerability checking (tainting, info. leaks)