Data-flow Analysis
Data-flow analysis (DFA)

- A framework for statically proving facts about program data.
  - Focuses on simple, finite facts about programs.
- Necessarily over- or under-approximate (may or must).
  - Conservatively considers all possible behaviors.
- Requires control-flow information; i.e., a control-flow graph.
  - If imprecise, DFA may consider infeasible code paths!
- Examples: reaching defs, available expressions, liveness,…
define fact(n : int)
{
    s := 1;
    while (n > 1)
    {
        s := s*n;
        n := n-1;
    }
    return s;
}
Control-flow graphs

- Intraprocedural CFGs may have a single entry/exit.
- Nodes can be a **basic block** or a single statement.
- Each block or statement has $1^+$ predecessor and $1^+$ successor (stmt or block).
- A **fork point** is where paths diverge and a **join point** is where paths come together.
Data-flow analysis

- Computed by propagating facts *forward* or *backward*.
- Computes *may* or *must* information.
- Reaching definitions/assignments (def-use): which assignments may reach each statement.
- Liveness: which variables are still needed at each point.
- Available expressions: which expressions are already stored.
- Very busy expressions: which expressions are computed down all possible paths forward.
Reaching defs, worklist algorithm

• Reaching definitions is a **forward may** analysis.

• gen(s) yields facts generated by a statement s.

• kill(s) yields facts invalidated by a statement s.

• pred(s) and succ(s) yield sets of statements preceding/succ.

• \(\text{entry}(s) = \bigcup_{s' \in \text{pred}(s)} \text{exit}(s')\); \(\text{exit}(s) = (\text{entry}(s) \setminus \text{kill}(s)) \cup \text{gen}(s)\)

• gen, kill, pred, succ, are fixed for each s; exit is monotonic in entry (if entry(s) grows, exit(s) grows); entry for exits of preds.

• We iterate rules for entry/exit until reaching a **fixed point**.
Reaching defs, worklist algorithm

• Main idea: worklist-based fixed-point algorithm.

• All entry(s) and exit(s) are initialized to be empty.

• Add all statements to the worklist; all must be considered.

• Until the wordlist is empty, remove an s from worklist:
  • Compute entry(s) from union of all preds’ exit(s’)
  • Compute exit(s) from gen(s), kill(s), and entry(s)
  • If exit(s) was increased, add all succ(s) to the worklist.
Reaching defs, worklist algorithm

```
entry(s) = ∅
exit(s) = ∅
W = all statements s //worklist
while W not empty:
    remove s from W
    entry(s) = ∪s'∈pred(s)exit(s')
    update = (entry(s) \ kill(s)) U gen(s)
    if update != exit(s):
        exit(s) = update
        W = W U succ(s)
```

Forward may analysis
## Reaching definitions analysis

<table>
<thead>
<tr>
<th>Stmt</th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s := 1;</code></td>
<td>(s,1)</td>
<td></td>
</tr>
<tr>
<td><code>n &gt; 1</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>s := s*n;</code></td>
<td>(s,3)</td>
<td>(s,1) (s,3)</td>
</tr>
<tr>
<td><code>n := n-1;</code></td>
<td>(n,4)</td>
<td>(n,0) (n,4)</td>
</tr>
<tr>
<td><code>return s;</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Diagram**

```
entry

s := 1;  

n > 1

s := s*n;  

n := n-1;  

return s;  

exit
```
Reaching definitions analysis

\( (n, 0) \)

\( (n, 0), (s, 1), (n, 4), (s, 3) \)

\( (n, 0), (s, 1), (n, 4), (s, 3) \)

\( (n, 0), (s, 3), (n, 4) \)

\( (n, 4), (s, 3) \)
all(x) = {(x, ℓ) | ∀ℓ}

RD_{entry}(1) = RD_{exit}(0)
RD_{entry}(2) = RD_{exit}(1) \cup RD_{exit}(4)
RD_{entry}(3) = RD_{exit}(2)
RD_{entry}(4) = RD_{exit}(3)
RD_{entry}(5) = RD_{exit}(2)

RD_{exit}(0) = {(n, 0)}
RD_{exit}(1) = RD_{entry}(1) \setminus \text{all}(s)
\quad \cup \{(s, 1)\}
RD_{exit}(2) = RD_{entry}(2)
RD_{exit}(3) = RD_{entry}(3) \setminus \text{all}(s)
\quad \cup \{(s, 3)\}
RD_{exit}(4) = RD_{entry}(4) \setminus \text{all}(n)
\quad \cup \{(n, 4)\}
RD_{exit}(5) = RD_{entry}(5)

fact(n)^0 \textbf{entry}

s := 1; \text{↑}

n > 1

s := s \times n; \text{↑}

n := n - 1; \text{↑}

return s; \text{↓}

fact(n) \textbf{exit}
Lattices

• Facts range over lattices: partial orders with joins (least upper bounds) and meets (greatest lower bounds).
Reaching definitions analysis

• The 11 sets \( RD_{\text{exit}}(0), RD_{\text{entry}}(1), RD_{\text{exit}}(1), \ldots RD_{\text{exit}}(5) \), are defined in terms of one another. Written as a vector of sets:

\[
\begin{align*}
&\text{RD} \\
&\text{RD}_0 \subseteq \text{RD}_1 \Rightarrow F(\text{RD}_0) \subseteq F(\text{RD}_1)
\end{align*}
\]

• Our set of equations can be turned into a monotonic \( F \):

\[
\begin{align*}
\text{RD} &= F(\text{RD}) = F^n(\bot) \text{ for some } n
\end{align*}
\]

• So that a satisfying vector of reachable defs is a fixed point:

\[
\begin{align*}
\text{RD} &= F(\text{RD}) = F^n(\bot) \text{ for some } n
\end{align*}
\]

• For example, the join point would end up encoded as:

\[
F(\ldots, RD_{\text{exit}}(1), \ldots, RD_{\text{exit}}(4), \ldots) = (\ldots, RD_{\text{exit}}(1) \cup RD_{\text{exit}}(4), \ldots)
\]
Very busy expressions analysis

- Computes a set of expressions that are computed down all paths forward before any subexpressions change value.

- Assignments represent GEN for the right hand side and KILL for expressions containing the right hand side (assigned var).

- Is a **backward must** data-flow analysis:
  - Propagates a set of computed expressions *backward*.
  - Computes the **meet** (GLB, intersection) of entry(s’) in s’∈succ(s) at each fork point to obtain exit(s).
Very busy expressions analysis

<table>
<thead>
<tr>
<th>Stmt</th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>s := 1;</td>
<td>1</td>
<td>s * n</td>
</tr>
<tr>
<td>n &gt; 1^2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s := s*n;</td>
<td>s*n</td>
<td>s * n</td>
</tr>
<tr>
<td>n := n-1;</td>
<td>n-1</td>
<td>n-1 * n</td>
</tr>
<tr>
<td>return s;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
fact(n) entry
s := 1;  

n > 1^2

s := s\*n;  
n := n-1;  
return s;  

fact(n) exit
```
Very busy expressions analysis

1
∅
s*n, n-1
n-1
∅

fact(n)\(\text{entry}\)

\[s := 1;^1\]

n > 1\(^2\)

\[s := s*n;^3\]

n := n-1\(^4\)

return s;\(^5\)

fact(n)\(\text{exit}\)
# May/Must & Forward/Backward

<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>Forward, computes exit(s)</td>
<td>Forward, computes exit(s)</td>
</tr>
<tr>
<td></td>
<td>join at join points</td>
<td>meet at join points</td>
</tr>
<tr>
<td></td>
<td>e.g., Reaching Defs</td>
<td>e.g., Available Expressions</td>
</tr>
<tr>
<td>Backward</td>
<td>Backward, computes entry(s)</td>
<td>Backward, computes entry(s)</td>
</tr>
<tr>
<td></td>
<td>join at join points</td>
<td>meet at join points</td>
</tr>
<tr>
<td></td>
<td>e.g., Live Variables</td>
<td>e.g., Very Busy Expressions</td>
</tr>
</tbody>
</table>