Iterative Data-flow Analysis

Data Flow Analysis

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

- Global data flow analysis
  - Problems are trivial within a basic block
  - Works on control flow graphs
  - Collects information needed for code generation and optimization
    - Global register allocation, global (partial) redundancy elimination, copy propagation, etc.
DATA FLOW ANALYSIS

- Basic idea
  - Setting up and solving systems of equations that relate information at various points in a program
  - Iterative algorithms
  - Desired result is usually *meet over all paths* solution
    - “What is true on every path from the entry?”
    - “Can this happen on any path from the entry?”

ITERATIVE ALGORITHMS

- First, compute some local information within individual basic blocks
- Then, propagate local information along control flow edges
  - IN(B): some property on entry to basic block B
  - OUT(B): some property on exit from basic block B
  - Need to iterate until no changes

```plaintext
while change do
    change = false
    for each basic block
        apply equations updating IN or OUT
        if IN/OUT changes, set change to true
    end
end
```
DATA FLOW ANALYSIS

- Available expressions
- Reachability
- Liveness

- Dataflow analysis in general
  - Does it halt?
  - Does it produce the desired answer?
  - How fast does it converge?

COMPUTING AVAILABLE EXPRESSIONS

- For each block $b$
  - $Exprkill(b)$: set of expression killed in $b$
  - $DEExpr(b)$: set of downward exposed expressions
  - $AVAIL(b)$: set of expressions available on entry to $b$

$$AVAIL(b) = \bigcap_{x \in \text{pred}(b)} (DEExpr(x) \cup (AVAIL(x) - Exprkill(x)))$$

- $AVAIL(b_0) = \emptyset$

- This system of simultaneous equations forms a data-flow problem
- Solve it with a data-flow algorithm
**Iterative Algorithm for AVAIL**

$$\text{AVAIL}(b_0) = \emptyset$$

for i = 1 to k

$$\text{AVAIL}(b_i) = \text{set of all expressions}$$

changed = true

while (changed)

changed = false

for i = 0 to k

OldValue = AVAIL(b_i)

$$\text{AVAIL}(b_i) = \bigcap_{x \in \text{pred}(b_i)} (\text{DEExpr}(x) \cup (\text{AVAIL}(x) - \text{Exprkill}(x)))$$

If AVAIL(b_i) <> OldValue then changed = true

---

**A Special Example**

- **Initialize**
  - AVAIL(B_1) = AVAIL(B_2) = \emptyset

- **Iteration 1**
  - AVAIL(B_1) = \emptyset
  - AVAIL(B_2)
    - = (DEExpr(B_1) \cup (AVAIL(B_1) - Exprkill(B_1))) \cap (DEExpr(B_2) \cup (AVAIL(B_2) - Exprkill(B_2)))
    - = DEExpr(B_1) \cap DEExpr(B_2)

- However, expressions downward exposed by B_1 and not killed by B_2 should be available at the entry of B_2
A SPECIAL EXAMPLE

- Initialize
  - AVAIL(B₁) = AVAIL(B₂) = ∅

- Iteration 1
  - AVAIL(B₁) = ∅
  - AVAIL(B₂) = DEExpr(B₁) ∩ DEExpr(B₂)
    = {b+1} ∩ {a+1} = ∅

- No changes in AVAIL sets, iteration halts

- But, b+1 is available on entry to B₂!

A SPECIAL EXAMPLE

- Problem: initializing AVAIL to be ∅ is too restrictive
- Initialize
  - AVAIL(B₁) = ∅
  - AVAIL(B₂) = {a+1, b+1}

- Iteration 1
  - AVAIL(B₁) = ∅
  - AVAIL(B₂) = DEExpr(B₁)
    ∩ (DEExpr(B₂) ∪ (AVAIL(B₂) - Exprkill(B₂)))
    = {b+1} ∩ {a+1, b+1} = {b+1}

- Iteration 2
  - AVAIL(B₁) = ∅
  - AVAIL(B₂) = DEExpr(B₁)
    ∩ (DEExpr(B₂) ∪ (AVAIL(B₂) - Exprkill(B₂)))
    = {b+1} ∩ {a+1, b+1} = {b+1}

- Halts and gets the right result!
**Iterative Algorithm for AVAIL - 2**

\[
\text{AVAIL}_{\text{IN}}(b_0) = \emptyset \\
\text{for } i = 1 \text{ to } k \\
\quad \text{AVAIL}_{\text{IN}}(b_i) = \text{set of all expressions} \\
\text{changed} = \text{true} \\
\text{while } (\text{changed}) \\
\quad \text{changed} = \text{false} \\
\quad \text{for } i = 0 \text{ to } k \\
\quad \quad \text{OldValue} = \text{AVAIL}_{\text{IN}}(b_i) \\
\quad \quad \text{AVAIL}_{\text{IN}}(b_i) = \bigcap_{x \in \text{pred}(b_i)} (\text{AVAIL}_{\text{OUT}}(x)) \\
\quad \quad \text{AVAIL}_{\text{OUT}}(b_i) = \text{DEExpr}(b_i) \bigcup (\text{AVAIL}_{\text{IN}}(b_i) - \text{ExprKill}(b_i)) \\
\quad \quad \text{If } \text{AVAIL}_{\text{IN}}(b_i) \neq \text{OldValue} \text{ then } \text{changed} = \text{true}
\]

**Data-flow Analysis Algorithm**

- What can be generalized from AVAIL calculation?
  - Local information collected within basic blocks
    - AVAIL: DEExpr, ExprKill
  - Propagate information along CFG edges
    - AVAIL: ... from entry point to current point ...
    - Forward propagation: predecessor\(\rightarrow\)successor
    - Join nodes: how to deal with multiple predecessors
  - Initialization
    - AVAIL: entry node = empty set, other nodes = universal set
- Two more examples: REACH, LIVE
**Reachability**

- A *definition* of a variable $x$ is a statement that may assign a value to $x$.

- A definition may *reach* a program point $p$ if there exists some path from the point immediately following the definition to $p$ such that the assignment is not killed along that path.
  - A definition of a variable $x$ is *killed* if there is any other definition of $x$ anywhere along the path.

- Concept: relationship between definitions and uses.

**What blocks do definitions $d2$ and $d4$ reach?**

![Diagram](image_url)
**Reasoning about Basic Blocks**

Effect of single statement: \( a = b \ op \ c \)
- Uses variables \{b,c\}
- Kills all definitions of \{a\}
- Generates new definition (i.e. assigns a value) of \{a\}

Local Analysis:
- Analyze the effect of each instruction
- Compose these effects to derive information about the entire block

**Reachability Analysis: Step 1**

- For each block, compute local (block level) information
  - \( \text{DEDef}(B) \): the set of downward-exposed definitions in \( B \)
    - Those for which the defined name is not subsequently redefined by the exit from \( B \)
  - \( \text{DEFKill}(B) \): the set of definitions that are obscured by a definition of the same name in \( B \)
    - Also consider definition points outside \( B \)
- This information does not take control flow between blocks into account
EXAMPLE

Definition of a block B:
- \( d_1 \ i = m - 1 \)
- \( d_2 \ j = n \)
- \( d_3 \ a = u_1 \)

Definition of a block B2:
- \( d_4 \ i = i + 1 \)
- \( d_5 \ j = j - 1 \)

Definition of a block B3:
- \( d_6 \ a = u_2 \)

Definition of a block B4:
- \( d_7 \ i = u_2 \)

DEFKill need to consider the set of all definition points: \{1,2,3,4,5,6,7\}

REACHABILITY ANALYSIS: STEP 2

- Compute \textit{REACHES} set for each block in a forward direction
  - \( \text{REACHES}(b) \): the set of definitions that reach the entry to a block \( b \)
  - Start with \( \text{REACHES}(n_0) = \emptyset \)
  - \( \text{REACHES}(b) = \bigcup_{x \in \text{pred}(b)} (\text{DEDef}(x) \cup (\text{REACHES}(x) \setminus \text{DEFKill}(x))) \)

- Iterative algorithm: keep computing \text{REACHES} sets until a fixed point is reached
  - Locally defined in \( x \)
  - Propagated into \( x \) and not killed by any definition in \( x \)
**Reachability Analysis Equation**

- Compute **REACHES** set for each block in a forward direction
  - **REACHES**\( (b) \): the set of definitions that reach the entry to a block \( b \)
  - Start with **REACHES**\( (n_0) = REACHES(b) = \emptyset \)
  - **REACHES**\( (b) = \bigcup_{x \in \text{pred}(b)} OUT(x) \)
  - \( OUT(x) = \text{DEDef}(x) \cup (\text{REACHES}(x) - \text{DEFKill}(x)) \)

  - \( OUT(x) \) is the set of definitions that reach the exit from a block \( x \), which include definitions that are
    - Either generated within the block (\( \text{DEDef}(x) \)), or
    - Reach on entry to \( x \) and not killed by any definition in \( x \) (\( \text{REACHES}(x) - \text{DEFKill}(x) \))

**REACHING DEFINITIONS ALGORITHM**

**Input**: Flow graph with \( \text{DEDef} \) and \( \text{DEFKill} \) computed for each block  
**Output**: **REACHES**\( (B) \) for each block \( B \)

For each block \( B \) initialize \( \text{REACHES}(B) = \emptyset \)
change = true;
while change do begin
  change = false;
  for each block \( B \) do begin
    oldvalue = \( \text{REACHES}(B) \);
    \( \text{REACHES}(B) = \)
    \( \bigcup_{x \in \text{pred}(B)} (\text{DEDef}(x) \cup (\text{REACHES}(x) - \text{DEFKill}(x))) \)
    if \( \text{REACHES}(B) \neq \text{oldvalue} \) then change = true;
  end
end
**Example**

\[
\text{REACHES}(B) = \bigcup_{x \in \text{pred}(B)} (\text{DEDef}(x) \cup (\text{REACHES}(x) - \text{DEFKill}(x)))
\]

<table>
<thead>
<tr>
<th>Iter 0</th>
<th>Iter 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>B2</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>B3</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>B4</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>

**Example**

<table>
<thead>
<tr>
<th></th>
<th>DE-Def</th>
<th>DEF-Kill</th>
<th>Pred</th>
<th>Iter 0</th>
<th>Iter 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>1,2,3</td>
<td>4,5,6,7</td>
<td>-</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>B2</td>
<td>4,5</td>
<td>1,2,7</td>
<td>B1, B4</td>
<td>(\emptyset)</td>
<td>Def(1)+Def(4) = {1,2,3} + {7} = {1,2,3,7}</td>
</tr>
<tr>
<td>B3</td>
<td>6</td>
<td>3</td>
<td>B2</td>
<td>(\emptyset)</td>
<td>Def(2) + (REACHES(2) - Kill(2)) = {4,5} + ((1,2,3,7) - {1,2,7}) = {3,4,5}</td>
</tr>
<tr>
<td>B4</td>
<td>7</td>
<td>1,4</td>
<td>B2, B3</td>
<td>(\emptyset)</td>
<td>(3,4,5) + Def(3) + (REACHES(3)-Kill(3)) = (3,4,5) + (6) + ({3,4,5} - {3}) = {3,4,5,6}</td>
</tr>
</tbody>
</table>

\[
\text{REACHES}(B) = \bigcup_{x \in \text{pred}(B)} (\text{DEDef}(x) \cup (\text{REACHES}(x) - \text{DEFKill}(x)))
\]
Example

<table>
<thead>
<tr>
<th>DE-Def</th>
<th>DEF-Kill</th>
<th>Pred</th>
<th>Iter 0</th>
<th>Iter 1</th>
<th>Iter 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>1, 2, 3</td>
<td>4, 5, 6, 7</td>
<td>-</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>B2</td>
<td>4, 5</td>
<td>1, 2, 7</td>
<td>B1, B4</td>
<td>Ø</td>
<td>1, 2, 3, 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Def(1)+Def(4)+(REACHES(4)-Kill(4))={1, 2, 3, 7}+{(3, 4, 5, 6)-{1, 4}}={1, 2, 3, 5, 6, 7}</td>
</tr>
<tr>
<td>B3</td>
<td>6</td>
<td>3</td>
<td>B2</td>
<td>Ø</td>
<td>{3, 4, 5}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Def(2) + (REACHES(2) - Kill(2)) = {4, 5}+{(1, 2, 3, 5, 6, 7)-{1, 2, 7}} = {3, 4, 5, 6}</td>
</tr>
<tr>
<td>B4</td>
<td>7</td>
<td>1, 4</td>
<td>B2, B3</td>
<td>Ø</td>
<td>{3, 4, 5, 6}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>{3, 4, 5, 6}+Def(3)+(REACHES(3)-Kill(3))={3, 4, 5}+{6}+{(3, 4, 5, 6)-{3}}={3, 4, 5, 6}</td>
</tr>
</tbody>
</table>

\[ REACHES(B) = \bigcup_{x \in \text{pred}(B)} (\text{DEDef}(x) \cup (\text{REACHES}(x) - \text{DEFKill}(x))) \]

Example

<table>
<thead>
<tr>
<th>DE-Def</th>
<th>DEF-Kill</th>
<th>Pred</th>
<th>Iter 0</th>
<th>Iter 1</th>
<th>Iter 2</th>
<th>Iter 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>1, 2, 3</td>
<td>4, 5, 6, 7</td>
<td>-</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>B2</td>
<td>4, 5</td>
<td>1, 2, 7</td>
<td>B1, B4</td>
<td>Ø</td>
<td>1, 2, 3, 7</td>
<td>1, 2, 3, 5, 6, 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(1, 2, 3, 7)</td>
<td>(1, 2, 3, 5, 6, 7)</td>
</tr>
<tr>
<td>B3</td>
<td>6</td>
<td>3</td>
<td>B2</td>
<td>Ø</td>
<td>{3, 4, 5}</td>
<td>{3, 4, 5, 6}</td>
</tr>
<tr>
<td>B4</td>
<td>7</td>
<td>1, 4</td>
<td>B2, B3</td>
<td>Ø</td>
<td>{3, 4, 5, 6}</td>
<td>{3, 4, 5, 6}</td>
</tr>
</tbody>
</table>

\[ REACHES(B) = \bigcup_{x \in \text{pred}(B)} (\text{DEDef}(x) \cup (\text{REACHES}(x) - \text{DEFKill}(x))) \]

No more changes in REACHES sets, algorithm halts!
## Data-flow Problems

- **Examples we have discussed**

<table>
<thead>
<tr>
<th></th>
<th>AVAIL</th>
<th>REACHES</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain</strong></td>
<td>Set of expressions</td>
<td>Set of definitions</td>
</tr>
<tr>
<td><strong>Direction</strong></td>
<td>Forward</td>
<td>Forward</td>
</tr>
<tr>
<td><strong>Equation across blocks</strong></td>
<td>$\text{AVAIL}(B) = \bigcap_{x \in \text{pred}(B)} \text{OUT}(X)$</td>
<td>$\text{REACHES}(B) = \bigcup_{x \in \text{pred}(B)} \text{OUT}(X)$</td>
</tr>
<tr>
<td><strong>Equations within blocks</strong></td>
<td>$\text{OUT}(X) = \text{DEExpr}(X) \cup (\text{AVAIL}(X) - \text{ExprKill}(X))$</td>
<td>$\text{OUT}(X) = \text{DEDef}(X) \cup (\text{REACHES}(X) - \text{DEFKill}(X))$</td>
</tr>
<tr>
<td><strong>Initialize</strong></td>
<td>$\text{AVAIL}(B) = E$</td>
<td>$\text{REACHES}(B) = \emptyset$</td>
</tr>
<tr>
<td><strong>Boundary</strong></td>
<td>$\text{AVAIL}(B_0) = \emptyset$</td>
<td>$\text{REACHES}(B_0) = \emptyset$</td>
</tr>
</tbody>
</table>

## Live Variable Analysis

- A variable $x$ is **live** at a point $p$ if there is **some** path from $p$ where $x$ is used before it is redefined
  - Want to determine for some variable $x$ and point $p$ whether the value of $x$ could be used along some path starting at $p$

- Liveness analysis:
  - Identify values alive across blocks
  - Backwards: variables used in a successor block are live variables
  - May – live if used along **some** path starting at $p$
  - Useful in register allocation and efficient SSA generation
**LIVE Variable Analysis**

- **To collect:**
  - \( \text{LIVEout}(B) \) - set of variables live on exit from block \( B \)

- **Compute at the local (block) level:**
  - \( \text{VarKill}(B) \) - the set of variables assigned values in \( B \)
  - \( \text{UEVar}(B) \) - the set of upward-exposed variables
    - Variables used in \( B \) prior to any re-definition of that variable

---

**Example**

<table>
<thead>
<tr>
<th>Block</th>
<th>VarKill</th>
<th>UEVar</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>{a,b}</td>
<td>{}</td>
</tr>
<tr>
<td>B2</td>
<td>{c,d}</td>
<td>{a,b}</td>
</tr>
<tr>
<td>B3</td>
<td>{d}</td>
<td>{b,d}</td>
</tr>
<tr>
<td>B4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example relationship diagram:

- \( B_1 \):
  - \( d_1: a := 1 \)
  - \( d_2: b := 2 \)

- \( B_2 \):
  - \( d_3: c := a + b \)
  - \( d_4: d := c - a \)

- \( B_3 \):
  - \( d_5: d := b * d \)

- \( B_4 \):
  - \( d_6: d := a + b \)
  - \( d_7: e := e + 1 \)

- \( B_5 \):
  - \( d_8: b := a + b \)
  - \( d_9: e := e - a \)

- \( B_6 \):
  - \( d_{10}: a := b + d \)
  - \( d_{11}: b := a - d \)
Example

<table>
<thead>
<tr>
<th>Block</th>
<th>VarKill</th>
<th>UEVar</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>(a,b)</td>
<td>{}</td>
</tr>
<tr>
<td>B2</td>
<td>(c,d)</td>
<td>(a,b)</td>
</tr>
<tr>
<td>B3</td>
<td>{d}</td>
<td>(b,d)</td>
</tr>
<tr>
<td>B4</td>
<td>(d,e)</td>
<td>(a,b,e)</td>
</tr>
<tr>
<td>B5</td>
<td>(b,e)</td>
<td>(a,b,c)</td>
</tr>
<tr>
<td>B6</td>
<td>(a,b)</td>
<td>(b,d)</td>
</tr>
</tbody>
</table>

Example

**LIVE Variable Analysis**

- **Equation**
  - \( \text{LIVEout}(B) \) - variables live on exit from block \( B \):

\[
\text{LIVEout}(n_f) = \text{LIVEout}(B) = \emptyset \\
\text{LIVEout}(B) = \\
\cup_{x \in \text{SUCC}(B)}(\text{UEVar}(x) \cup (\text{LIVEout}(x) \cap \text{VarKill}(x)))
\]
**LIVE VARIABLE ANALYSIS**

- **Equation**
  - \( \text{LIVEout}(B) \) - variables live on exit from block B:

  \[
  \text{LIVEout}(n_f) = \text{LIVEout}(B) = \emptyset \\
  \text{LIVEout}(B) = \bigcup_{x \in \text{SUCC}(B)} \text{IN}(x)
  \]

  \[
  \text{IN}(x) = \text{UEVar}(x) \cup (\text{LIVEout}(x) \setminus \text{VarKill}(x))
  \]

- **Using the local information, compute iteratively over the CFG**
  - Initially, \( \text{LIVEout}(B) = \emptyset \) for all B
  - Re-evaluate \( \text{LIVEout} \) at each node repeatedly
  - Halts when the information stops changing
EXAMPLE

\[ \text{LIVEout}(B) = \bigcup_{x \in \text{SUCC}(B)} (\text{UEVar}(x) \cup (\text{LIVEout}(x) - \text{VarKill}(x))) \]

EXAMPLE

\[ \text{LIVEout}(B) = \bigcup_{x \in \text{SUCC}(B)} (\text{UEVar}(x) \cup (\text{LIVEout}(x) - \text{VarKill}(x))) \]
EXAMPLE

<table>
<thead>
<tr>
<th>Block</th>
<th>VarKill</th>
<th>UEVar</th>
<th>SUCC</th>
<th>Iter 0</th>
<th>Iter 1</th>
<th>Iter 2</th>
<th>Iter 3</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>(a,b)</td>
<td>{}</td>
<td>B2</td>
<td>{a,b}</td>
<td>{a,b}</td>
<td>{a,b}</td>
<td>{a,b}</td>
<td>…</td>
</tr>
<tr>
<td>B2</td>
<td>(c,d)</td>
<td>(a,b)</td>
<td>B3, B5</td>
<td>{a,b,c,d}</td>
<td>{a,b,c,d,e}</td>
<td>{a,b,c,d,e}</td>
<td>…</td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td>(d)</td>
<td>(b,d)</td>
<td>B4, B5</td>
<td>{a,b,c,e}</td>
<td>{a,b,c,d,e}</td>
<td>{a,b,c,d,e}</td>
<td>…</td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td>(d,e)</td>
<td>(a,b,e)</td>
<td>B3</td>
<td>{b,d,a,c,e}</td>
<td>{b,d,a,c,e}</td>
<td>{a,b,c,d,e}</td>
<td>…</td>
<td></td>
</tr>
<tr>
<td>B5</td>
<td>(b,e)</td>
<td>(a,b,c)</td>
<td>B2, B6</td>
<td>{a,b,d}</td>
<td>{a,b,d,e}</td>
<td>{a,b,d,e}</td>
<td>…</td>
<td></td>
</tr>
<tr>
<td>B6</td>
<td>(a,b)</td>
<td>(b,d)</td>
<td>-</td>
<td>{a,b}</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>…</td>
</tr>
</tbody>
</table>

- Iteration 4 confirms that we’ve reached a fixed point

\[ \text{LIVEout}(B) = \cup_{x \in \text{SUCC}(B)} (\text{UEVar}(x) \cup (\text{LIVEout}(x) - \text{VarKill}(x))) \]

Results

LIVEout(B)

- B1:
  d1: a := 1
  d2: b := 2
  {a,b,c,e}

- B2:
  d3: c := a + b
  d4: d := c - a
  {a,b,c,d,e}

- B3:
  d5: d := b \times d
  {a,b,c,d,e}

- B4:
  d6: d := a + b
  d7: e := e + 1
  {a,b,c,d,e}

- B5:
  d8: b := a + b
  d9: e := c - a
  {a,b,d,e}

- B6:
  d10: a := b * d
  d11: b := a - d
  {a,b,c,d,e}
### Types of Dataflow Analysis

- **Forward vs. backward dataflow analysis**
  - **Forward**
    - In a CFG, information is propagated from a basic block B's predecessors to B
    - Example: reachability, available expressions, constant propagation
  - **Backward**
    - In a CFG, information is propagated from a basic block B's successors to B
    - Example: live variable analysis

- **May vs. Must**
  - **Must** – true on **all paths** (set intersection)
    - Example: available expression – expression must be defined and not killed on all path
  - **May** – true on **some path** (set union)
    - Example: live variable analysis – a variable is live if it could be used on some path
**DATA FLOW ANALYSIS EXAMPLES**

<table>
<thead>
<tr>
<th>Domain</th>
<th>AVAIL</th>
<th>REACHES</th>
<th>LIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of expressions</td>
<td>Set of definitions</td>
<td>Set of variables</td>
<td></td>
</tr>
<tr>
<td>Direction</td>
<td>Forward</td>
<td>Forward</td>
<td>Backward</td>
</tr>
<tr>
<td>Equation across blocks</td>
<td>AVAIL(B) = ( \cap_{x \in \text{pred}(B)} \text{OUT}(X) )</td>
<td>REACHES(B) = ( \cup_{x \in \text{pred}(B)} \text{OUT}(X) )</td>
<td>LIVEout(B) = ( \cup_{x \in \text{succ}(B)} \text{IN}(X) )</td>
</tr>
<tr>
<td>Equation within blocks</td>
<td>( \text{OUT}(X) = \text{DEExpr}(X) \cup (\text{AVAIL}(X) - \text{ExprKill}(X)) )</td>
<td>( \text{OUT}(X) = \text{DEDef}(X) \cup (\text{REACHES}(X) - \text{DEFKill}(X)) )</td>
<td>( \text{IN}(X) = \text{UEVar}(X) \cup (\text{LIVEout}(X) - \text{VarKill}(X)) )</td>
</tr>
<tr>
<td>Initialize</td>
<td>AVAIL(B) = E</td>
<td>REACHES(B) = ( \emptyset )</td>
<td>LIVEout(B) = ( \emptyset )</td>
</tr>
<tr>
<td>Boundary</td>
<td>AVAIL(B(_0)) = ( \emptyset )</td>
<td>REACHES(B(_0)) = ( \emptyset )</td>
<td>LIVEout(B(_f)) = ( \emptyset )</td>
</tr>
</tbody>
</table>

**THEORETICAL FOUNDATIONS**

- **Formal model of iterative data-flow analysis**
- **Termination**
  - Does the iterative solver halt?
- **Correctness**
  - Does the iterative solver guarantee the right solution?
- **Speed**
  - How fast does the iterative solver converge?
- **References**
  - Dragon book Ch9.3
FORMAL MODEL

A data flow analysis framework (D, V, ∧, F)
- A direction of the data flow D
  - Forward or backward
- A meet-semi-lattice which includes a domain of values V and a meet operator ∧
- A family F of transfer functions V → V

Example 1: REACH analysis
- D: forward
- V: \(2^{\text{Def}}\), where \(\text{Def}\) is the set of all definitions, ∧: ∪
- For a block \(n\), \(f_n\) describes how information flows within \(n\): \(f_n\) has the form \(f_n(x) = a_n \cup (x - b_n)\)
  - Where \(a_n\) is \(\text{DEDef}(n)\) and \(b_n\) is \(\text{DEKILL}(n)\)

Example 2: LIVE analysis
- D: backwards
- V: \(2^{\text{Var}}\), where \(\text{Var}\) is the set of all variables, ∧: ∪
- For a block \(n\), \(f_n\) describes how information flows within \(n\): \(f_n\) has the form \(f_n(x) = a_n \cup (x - b_n)\)
  - Where \(a_n\) is \(\text{UEVar}(n)\) and \(b_n\) is \(\text{VARKILL}(n)\)

Example 3: AVAIL analysis
- D: forwards
- V: \(2^{\text{Exp}}\), where \(\text{Exp}\) is the set of all expressions, ∧: ∩
- For a block \(n\), \(f_n\) describes how information flows within \(n\): \(f_n\) has the form \(f_n(x) = a_n \cup (x - b_n)\)
  - Where \(a_n\) is \(\text{DEExpr}(n)\) and \(b_n\) is \(\text{EXPRKILL}(n)\)
SCALAR OPTIMIZATIONS

ROADMAP

- Constant propagation
- Copy propagation
- Code motion for loop invariants
- Partial redundancy elimination
CONSTANT PROPAGATION

\[ s: x := C \quad //\text{for some constant } C \]

\[ u: \ldots := x \]

- If statement \( s \) is the only definition of \( x \) reaching statement \( u \), we can replace \( x \) with constant \( C \)
  - Save a register
  - Enable constant folding and dead code elimination
  - Can potentially remove conditional branches
- What if more than one definition reaches \( u \)?
  - Data-flow analysis across basic blocks
- Replacement is iterative
  - One replacement may trigger more opportunities

USING DATAFLOW EQUATIONS

- \( \text{ConstIn}(b) \): pairs of \(<\text{variable}, \text{value}>\) that the compiler can prove to hold on entry to block \( b \)
  - One \(<\text{variable}, \text{value}>\) for each variable
  - \(<x, C> \in \text{ConstIn}(b)\): variable \( x \) is guaranteed to take a constant value \( C \) on entry to block \( b \)
  - \(<x, \text{NAC}>\): \( x \) is guaranteed not to be a constant
  - \(<x, \text{UNDEF}>\): we know nothing assertive about \( x \)
  - \( \text{ConstIn}(b) = \land \text{ConstOut}(j) \) for block \( j \in \text{Pred}(b) \)
- Meet operation for the pairs
  - \(<x, c> \land <x, c> = <x, c>\)
  - \(<x, c1> \land <x, c2> = <x, \text{NAC}> \) \((c1 \neq c2)\)
  - \(<x, c> \land <x, \text{NAC}> = <x, \text{NAC}>\)
  - \(<x, c> \land <x, \text{UNDEF}> = <x, c>\)
  - \(<x, \text{UNDEF}> \land <x, \text{NAC}> = <x, \text{NAC}>\)
**Using Dataflow Equations**

- **ConstOut(b):** pairs of \(<\text{variable}, \text{value}>\) on exit from block \(b\)
  - Initialized to be \(\text{ConstIn}(b)\) and modified by processing each statement \(s\) in block \(b\) in order.
  - \(s\) is a simple copy: \(x \leftarrow y\), the value of \(y\) decides \(x\).
  - \(s\) is a computation: \(x \leftarrow y \text{ op } z\), the values of \(y\) and \(z\) decide \(x\):
    - \(<x, c_1 \text{ op } c_2> \in \text{ConstOut} \text{ if } <y, c_1> \text{ and } <z, c_2> \in \text{ConstOut}\)
    - \(<x, \text{NAC}> \in \text{ConstOut} \text{ if either } <y, \text{NAC}> \text{ or } <z, \text{NAC}> \in \text{ConstOut}\)
    - \(<x, \text{UNDEF}> \in \text{ConstOut} \text{ otherwise}\)
  - \(s\) is a function call or assignment via a pointer: \(<x, \text{NAC}> \in \text{ConstOut}\)

- Optimization opportunity exists only for \(x\) s.t. \(<x, C> \in \text{ConstIn}(b)\) for some constant \(C\)

---

**Example**

\[
\begin{array}{c}
\{<X,\text{UNDEF}>, <Y,\text{UNDEF}>, \ldots\} \\
X = 2 \\
Y = 3 \\
\{<X,2>,<Y,3>,\ldots\} \\
\{<X,NAC>,<Y,NAC>,\ldots\} \\
Z = X + Y \\
\{<X,NAC>,<Y,NAC>,<Z,NAC>,\ldots\} \\
\end{array}
\]
**CONSTANT PROPAGATION w/ SSA**

- For statements $x_i := C$, for some constant $C$, replace all $x_i$ with $C$
- For $x_i := \phi(C,C,...,C)$, for some constant $C$, replace statement with $x_i := C$
- Iterate

**EXAMPLE: SSA**

```plaintext
a := 3
b := 2
f := a + b
g := 5
a := g - b
f <= g
```

```plaintext
f := g + 1
d := 2
```

```plaintext
da := 2
```

```plaintext
d3 = \phi(d2,d1)
a3 = \phi(a2,a1)
f1 := a3 + d3
g1 := 5
```

```plaintext
f1 <= g1
```

```plaintext
f2 := g1 + 1
g1 < a2
```

```plaintext
f3 := \phi(f2,f1)
d2 := 2
```

```plaintext
d2 := 2
```
EXAMPLE: SSA

```
a1 := 3
d1 := 2

\[ d_3 = \phi(d_2, d_1) \]
\[ a_3 = \phi(a_2, a_1) \]
\[ f_1 := a_3 + d_3 \]
\[ g_1 := 5 \]
\[ a_2 := g_1 - d_3 \]
\[ f_1 \leq g_1 \]

\[ f_2 := g_1 + 1 \]
\[ f_3 := \phi(f_2, f_1) \]
\[ d_2 := 2 \]
```

This may continue for a few steps ...

```
a1 := 3
d1 := 2

\[ d_3 = \phi(2, 2) \]
\[ a_3 = \phi(a_2, 3) \]
\[ f_1 := a_3 + d_3 \]
\[ g_1 := 5 \]
\[ a_2 := 5 - d_3 \]
\[ f_1 \leq 5 \]

\[ f_2 := 5 + 1 \]
\[ f_3 := \phi(f_2, f_1) \]
\[ d_2 := 2 \]
```

26
**Copy Propagation**

- Idea: use $v$ for $u$ wherever possible after the copy statement $u=v$
- Benefits
  - Can create dead code
  - Can enable algebraic simplifications

```
\begin{align*}
b &:= a \\
c &:= 4 \cdot b \\
c &> b \\
d &:= b + 2 \\
e &:= a + b
\end{align*}
```

```
\begin{align*}
b &:= a \\
c &:= 4 \cdot a \\
c &> a \\
d &:= a + 2 \\
e &:= a + a
\end{align*}
```

**Using Dataflow Analysis**

- Finding copies in blocks can be represented by a dataflow analysis framework similar to the one for constant propagation
  - A pair $<u,v>$ indicates that value is copied from $v$ to $u$
- Data flow direction?
  - Forward analysis
- Meet operator?
  - $\text{CopyIn}(b) = \cap \text{CopyOut}(j)$ for every predecessor $j$ of $b$
- Transfer function?
  - $\text{CopyOut}(b)$ is computed from $\text{CopyIn}(b)$ by processing each operations in $b$
- Similar to constant propagation
**Example 1**

```
CopyIn(b)
CopyOut(b)

b := a
l := b + 2
c := 4*b
d := c + 2
f := a + b

{<b,a>}
{<b,a>}
{<b,a>}
{<b,a>}
```
**Example 2**

CopyIn(b)
CopyOut(b)

{<d,c>}
{<d,c>}
{<d,c>,<g,e>}
{<d,c>,<g,e>}
{<d,c>,<g,e>}
{<d,c>,<g,e>}
{<d,c>,<g,e>}
{<d,c>,<g,e>}
{<d,c>,<g,e>}
{<g,e>}
{<d,c>,<g,e>}
{<g,e>}
{<d,c>}
{<d,c>}

**Loop Invariants & Code Motion**

- A loop invariant expression is a computation whose value does not change as long as control stays in the loop.
- Code motion is the optimization that finds loop invariants and moves them out of the loop.

```plaintext
while (i <= limit - 2) { ... }
⇒
t := limit - 2
while (i <= t) { ... }
```
**Part 1: Detecting Loop Invariants**

- Mark “invariant” all statements whose operands either are constants or have all reaching definitions outside the loop
  - How to know this?
- Iterate until there are no more “invariants” to mark
  - Iteratively marking all statements whose operands either are constants, have all reaching definitions outside the loop or have only “invariant” reaching definitions

```
L O O P  I N V A R I A N T S

\[ i = 1 \]
\[ i \leq 100 \]
\[ t1 = n + 2 \]
\[ k = i \times t1 \]
\[ j = i \]
\[ j \leq 100 \]
\[ t2 = 100 \times n \]
\[ t3 = 10 \times k \]
\[ t4 = t2 + t3 \]
\[ t5 = t4 + j \]
\[ j = j + 1 \]
```

**Loop Invariants**

- Inner loop
  - do i = 1, 100
  - do j = i, 100
  - \[ a[i,j] = 100 \times n + 10 \times k + j \]
- Outer loop

30
**Loop Invariants: SSA**

\[ i_1 = 1 \]

\[ i_2 = \phi(i_1, i_3) \]
\[ i_2 \leq 100 \]

\[ t_1 = n_1 \]
\[ k_1 = i_2 \times t_1 \]
\[ i_3 = i_2 + 1 \]

\[ t_2 = 100 \times n_1 \]
\[ t_3 = 10 \times k_1 \]
\[ t_4 = t_2 + t_3 \]
\[ t_5 = t_4 + j_2 \]

\[ j = j_2 + 1 \]

---

**Part 2: Code Motion**

- An invariant statement \( x := y + z \) can sometimes be moved out of the loop
  - Code can be moved just before the header
    - Will dominate the whole loop after code motion
  - Three conditions (following slides)
Condition 1: To move invariant $t := x \text{ op } y$, either the block that containing this invariant must dominate all loop exits, or $t$ must be not live-out of any loop exit.

Violation of Condition 1:

Condition 2: To move invariant $t := x \text{ op } y$, it must be the only definition of $t$ in the loop.

Violation of Condition 2:
Condition 3: To move invariant \( t := x \) or \( y \), no use of \( t \) in the loop is reached by any other definition of \( t \).

Violation of Condition 3

\[
\begin{align*}
x &:= 1 \\
u &< v \\
k &:= x \\
u &:= u + 1 \\
x &:= 2 \\
v &:= v - 1 \\
v &\leq 20
\end{align*}
\]

Assuming \( t_1 \) not live outside the loop-nest, this stmt is invariant and all three

\[
\begin{align*}
i_1 &= 1 \\
t_1 &= n_1 + 2 \\
k_1 &= t_2 \\
j_1 &= j_2 \\
t_2 &= 100 \cdot n_1 \\
t_3 &= 10 \cdot k_1 \\
t_4 &= t_2 + t_3 \\
t_5 &= t_4 + j_2 \\
j_3 &= j_2 + 1
\end{align*}
\]
CODE MOTION EXAMPLE

\begin{align*}
    i_1 &= 1 \\
    t_{10} &= n_1 + 2 \\
    i_2 &= \phi(i_1, j_3) \\
    i_2 &= i_1 \\
    k_1 &= i_2 \cdot t_{10} \\
    j_1 &= i_2 \\
    j_2 &= f(j_1, j_3) \\
    j_2 &= j_1 + 1 \\
    t_{20} &= 100 \cdot n_1 \\
    t_{30} &= 10 \cdot k_1 \\
    t_{40} &= t_{20} + t_{30} \\
    t_{50} &= t_{40} + i_2 \\
    i_3 &= i_2 + 1 \\
    j_3 &= j_2 + 1 \\
    t_{f} &= f(t_{20}, t_{30}) \\
    t_{f} &= t_{50}
\end{align*}

invariant and all conditions met, assuming t2, t3, t4 not live outside the loop-nest

CODE MOTION EXAMPLE

\begin{align*}
    i_1 &= 1 \\
    t_{10} &= n_1 + 2 \\
    i_2 &= \phi(i_1, j_3) \\
    i_2 &= i_1 \\
    k_1 &= i_2 \cdot t_{10} \\
    j_1 &= i_2 \\
    j_2 &= f(j_1, j_3) \\
    j_2 &= j_1 + 1 \\
    t_{20} &= 100 \cdot n_1 \\
    t_{30} &= 10 \cdot k_1 \\
    t_{40} &= t_{20} + t_{30} \\
    t_{50} &= t_{40} + j_2 \\
    i_3 &= i_2 + 1 \\
    j_3 &= j_2 + 1 \\
    t_{f} &= f(t_{20}, t_{30}) \\
    t_{f} &= t_{50}
\end{align*}

invariant and all conditions met
**REDUNDANT EXPRESSIONS**

- Expression E is redundant at point p if
  - On every path to p, E has been evaluated before reaching p and none of the constituent values of E has been redefined after the evaluation.

- Expression E is *partially redundant* at point p if
  - E is redundant along some but not all paths to p
  - To optimize: insert code to make it fully redundant.

**LOOP INVARIANTS**

- Loop invariant expressions are partially redundant
  - Available for all loop iterations except for the very first one
- Code motion works by making the expression fully redundant
**PARTIAL REDUNDANCY ELIMINATION**

- Uses standard data-flow techniques to figure out where to move the code
- Subsumes classical global common sub-expression elimination and code motion of loop invariants
- Used by many optimizing compilers
  - Traditionally applied to lexically equivalent expressions
  - With SSA support, applied to values as well

**PARTIAL REDUNDANCY ELIMINATION**

- May add a block to deal with *critical edges*
  - Critical edge – edge leading from a block with more than one successor to a block with more than one predecessor
**PARTIAL REDUNDANCY ELIMINATION**

- Code duplication to deal with redundancy

Can we find a way to deal with redundancy in general??

**LAZY CODE MOTION**

Redundancy: common expressions, loop invariant expressions, partially redundant expressions

Desirable Properties:
- All redundant computations of expressions that can be eliminated with code duplication are eliminated.
- The optimized program does not perform any computation that is not in the original program execution.
- Expressions are computed at the latest possible time.
LAZY CODE MOTION

- Solve four data-flow problems that reveal the limit of code motion
  - AVAIL: available expressions
  - ANTI: anticipated expression
  - EARLIEST: earliest placement for expressions
  - LATER: expressions that can be postponed

- Compute INSERT and DELETE sets based on the data-flow solutions for each basic block
  - They define how to move expressions between basic blocks

Can we make this better?
Lazy Code Motion

Locally Information

For each block b, compute the local sets:

- **DEExpr**: an expression is downward-exposed (locally generated) if it is computed in b and its operands are not modified after its last computation
- **UEExpr**: an expression is upward-exposed if it is computed in b and its operands are not modified before its first computation
- **NotKilled**: an expression is not killed if none of its operands is modified in b

\[
\begin{align*}
  f &= b + d \\
  a &= b + c \\
  d &= a + e
\end{align*}
\]

DEExpr = \{a + e, b + c\}
UEExpr = \{b + d, b + c\}
NotKilled = \{b + c\}
LOCAL INFORMATION

What do they imply?
- **DEExpr**: \( e \in \text{DEExpr}(b) \Rightarrow \) evaluating \( e \) at the end of \( b \) produces the same result as evaluating it at the original position in \( b \)
- **UEExpr**: \( e \in \text{UEExpr}(b) \Rightarrow \) evaluating \( e \) at the entry of \( b \) produces the same result as evaluating it at the original position in \( b \)
- **NotKilled**: \( e \in \text{NotKilled}(b) \Rightarrow \) evaluating \( e \) at either the start or end of \( b \) produces the same result as evaluating it at the original position

\[
\begin{align*}
f &= b + d \\
a &= b + c \\
d &= a + e
\end{align*}
\]

DEExpr = \{a + e, b + c\}
UEExpr = \{b + d, b + c\}
NotKilled = \{b + c\}

GLOBAL INFORMATION

Availability
- \( \text{AvailIn}(n_0) = \emptyset \)
- \( \text{AvailIn}(b) = \bigcap_{x \in \text{pred}(b)} \text{AvailOut}(x), b \neq n_0 \)
- \( \text{AvailOut}(b) = \text{DEExpr}(b) \cup (\text{AvailIn}(b) \cap \text{NotKilled}(b)) \)
- Initialize \( \text{AvailIn} \) and \( \text{AvailOut} \) to be the set of expressions for all blocks except for the entry block \( n_0 \)

Interpreting Avail sets
- \( e \in \text{AvailOut}(b) \Leftrightarrow \) evaluating \( e \) at end of \( b \) produces the same value for \( e \) as its most recent evaluation, no matter whether the most recent one is inside \( b \) or not
- \( \text{AvailOut} \) tells the compiler how far forward \( e \) can move
GLOBAL INFORMATION

- **Anticipability**
  - Expression e is anticipated at a point p if e is certain to be evaluated along all computation path leaving p before any re-computation of e’s operands
  - AntOut(nf) = Ø
  - AntOut(b) = $\bigcap x \in \text{succ}(b) \text{AntIn}(x)$, $b \neq nf$
  - AntIn(b) = UEExpr(b) $\cup$ (AntOut(b) $\cap$ NotKilled(b))
  - Initialize AntOut to be the set of expressions for all blocks except for the exit block nf

- **Interpreting Ant sets**
  - $e \in \text{AntIn}(b) \iff$ evaluating e at start of b produces the same value for e as evaluating it at the original position (later than start of b) with no additional overhead
  - AntIn tells the compiler how far backward e can move

---

**Example**

- **Block**
  - **Not-Killed**
  - **DE-Expr**
  - **UE-Expr**

<table>
<thead>
<tr>
<th>Block</th>
<th>Not-Killed</th>
<th>DE-Expr</th>
<th>UE-Expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>(x*y)</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>B2</td>
<td>(x*y)</td>
<td>(x*y)</td>
<td>(x*y)</td>
</tr>
<tr>
<td>B3</td>
<td>(x*y)</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>B4</td>
<td>(x*y)</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>B5</td>
<td>(x*y)</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>B6</td>
<td>(x*y)</td>
<td>(x*y)</td>
<td>(x*y)</td>
</tr>
<tr>
<td>B7</td>
<td>(x*y)</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>B8</td>
<td>(x*y)</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>B9</td>
<td>(x*y)</td>
<td>(x*y)</td>
<td>(x*y)</td>
</tr>
<tr>
<td>Exit</td>
<td>(x*y)</td>
<td>()</td>
<td>()</td>
</tr>
</tbody>
</table>
**Example: Avail**

\[ \mu \text{AvailIn}(b) = \bigcap_{x \in \text{pred}(b)} \mu \text{AvailOut}(x) \]

\[ \mu \text{AvailOut}(b) = \text{DEExpr}(b) \cup (\mu \text{AvailIn}(b) \cap \text{NotKilled}(b)) \]

**Example: Ant**

\[ \mu \text{AntOut}(b) = \bigcap_{x \in \text{succ}(b)} \mu \text{AntIn}(x) \]

\[ \mu \text{AntIn}(b) = \text{UEExpr}(b) \cup (\mu \text{AntOut}(b) \cap \text{NotKilled}(b)) \]
**Example: Avail and Ant**

```plaintext
z = a
x > 3

z = x * y
y < 5
z < 7

b = x * y

Interesting spots: Anticipated but not available
```

**Lazy Code Motion**

- A powerful algorithm
  - Finds different forms of redundancy in a unified framework
  - Subsumes loop invariant code motion and common expression elimination
- Data-flow analysis
  - Composes several simple data-flow analyses to produce a powerful result