Dataflow Analysis 2

Data-flow Analysis

• Algorithm
  1. Find basic blocks, build CFG
  2. Find propagation functions for basic blocks
  3. Propagate information around CFG
  4. Propagate information into basic blocks

• Example

  Code
  ```
  a = 1
  if (b) then
    c = a+b
  else
    b = 1
    c = a+b
  ...
  ```

  Control flow graph (CFG)
Live Variables

A use of a variable x is a statement where the value of x may be read.

A use of variable x is killed if x is reassigned a value; the "killing" definition must occur.

\[
LV(B) = \bigcup_{Bi \in SUC(B)} (GEN(Bi) \cup [ LV(Bi) - KILL(Bi) ]) 
\]

\(GEN(Bi)\): All uses in Bi that are not killed by preceding definitions in Bi
\(KILL(Bi)\): All uses of variables x that are defined in Bi

Live Variables

- Universe of facts?
  - All possible subsets of variables in the program

- GEN and KILL sets for each basic block?
  - GEN = variables used in basic block AND not killed before reaching beginning of basic block
  - KILL = variables defined in basic block

- Initial values of \(LV(B)\) before propagation starts?
  - All variables in the program
    - Assumes variable is live until proven otherwise
    - The conservative assumption
Live Variables Example

\[ \text{LV}(B) = \bigcup (\text{GEN}(B_i) \cup [\text{AVAIL}(B_i) - \text{KILL}(B_i)]) \]
\[ B_i \in \text{SUCC}(B) \]

Available Expressions

An expression e is **defined** if its value is computed.

An expression e is **killed** if the values of any of its operands may have been changed.

\[ \text{AVAIL}(B) = \bigcap (\text{GEN}(B_i) \cup [\text{AVAIL}(B_i) - \text{KILL}(B_i)]) \]
\[ B_i \in \text{PRED}(B) \]

**GEN(Bi)**: All definitions of expressions in Bi that are not subsequently killed in Bi

**KILL(Bi)**: All expressions with operand variables defined in Bi
Available Expressions

- Universe of facts?
  → All possible subsets of expressions in the program

- GEN and KILL sets for each basic block?
  → GEN = expressions defined in basic block AND 
    operands not defined before reaching end of basic block
  → KILL = expressions whose operands are defined in basic block

- Initial values of AVAIL(B) before propagation starts?
  → Ø
  - Assumes expression is not available until proven otherwise
  - The conservative assumption

Available Expressions Example

- Control flow graph

<table>
<thead>
<tr>
<th>Node</th>
<th>KILL</th>
<th>GEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>a+b</td>
<td>Ø</td>
</tr>
<tr>
<td>B</td>
<td>Ø</td>
<td>a+b</td>
</tr>
<tr>
<td>C</td>
<td>a+b</td>
<td>a+b</td>
</tr>
<tr>
<td>D</td>
<td>Ø</td>
<td>Ø</td>
</tr>
</tbody>
</table>

- Solution
  \[
  \begin{align*}
  \text{AVAIL}(A) &= \emptyset \\
  \text{AVAIL}(B) &= \text{GEN}(A) \cup (\text{AVAIL}(A) - \text{KILL}(A)) = \emptyset \cup (\emptyset - \{a+b\}) = \emptyset \\
  \text{AVAIL}(C) &= \text{GEN}(A) \cup (\text{AVAIL}(A) - \text{KILL}(A)) = \emptyset \cup (\emptyset - \{a+b\}) = \emptyset \\
  \text{AVAIL}(D) &= (\text{GEN}(B) \cup (\text{AVAIL}(B) - \text{KILL}(B))) \cap \\
  &\quad (\text{GEN}(C) \cup (\text{AVAIL}(C) - \text{KILL}(C))) \\
  &= (\{a+b\} \cup (\emptyset - \emptyset)) \cap (\{a+b\} \cup (\emptyset - \{a+b\})) = \{a+b\}
  \end{align*}
\]
Very Busy Expressions

An expression $e$ is **defined** if its value is used before any definition of its operands.

An expression $e$ is **killed** if the values of any of its operands may have been changed before it is used.

$$VBE(B) = \bigcap_{Bi \in SUCC(B)} (GEN(Bi) \cup \left( VBE(Bi) - KILL(Bi) \right))$$

**$GEN(Bi)$**: All definitions of expressions in $Bi$ that are used before they are killed in $Bi$

**$KILL(Bi)$**: All expressions with operand variables defined in $Bi$

- **Universe of facts?**
  - All possible subsets of expressions in the program

- **GEN and KILL sets for each basic block?**
  - $GEN = \text{expressions used in basic block AND}$
    - operands not defined before reaching beginning of basic block
  - $KILL = \text{expressions whose operands are defined in basic block}$

- **Initial values of $VBE(B)$ before propagation starts?**
  - $\emptyset$
    - Assumes expression is not busy until proven otherwise
    - The conservative assumption
Very Busy Expressions Example

\[
VBE(B) = \bigcap \left( \text{GEN}(B_i) \cup \left[ VBE(B_i) \setminus \text{KILL}(B_i) \right] \right)
\]

\(B_i \in \text{SUCC}(B)\)

Implementation Issues: Bit-vector Problems

The set of facts (universe of facts) can often be expressed as finite subsets of a finite base set. Such sets can be represented as bit-vectors.

The set of facts (universe of facts) can often be expressed as finite subsets of a finite base set. Such sets can be represented as bit-vectors.


\[
\begin{align*}
\text{B1:a, B2:c, B3:d} &= \underline{1 1 1} \\
\text{B1:a, B2:c} &= \underline{1 1 0} \\
\text{B1:a, B3:d} &= \underline{1 0 1} \\
\text{B2:c, B3:d} &= \underline{0 1 1} \\
\text{B1:a} &= \underline{1 0 0} \\
\text{B2:c} &= \underline{0 1 0} \\
\text{B3:d} &= \underline{0 0 1} \\
\text{T} &= \underline{0 0 0}
\end{align*}
\]

Meet operation is either bit-wise logical AND or bit-wise logical OR.

GEN and KILL sets can be expressed as single bit-vectors.

Bit-wise logical - is bit-wise negation followed by bit-wise AND.

Implementation steps:
1. bit-vector construction/interpretation
2. bit-vector CFG initialization (RD, GEN, and KILL vectors)
3. bit-vector CFG propagation
4. information post-processing (e.g.: DU/UD chains)
**Bit-vector Problem?**

B1: $i := a + b$

B2: $j := a + b$

B3: $k := a + b$

- What does lattice look like?
  - $a + b$

- GEN and KILL functions?
  - B1: $a + b$ in GEN, KILL is empty
  - B2: $a + b$ in GEN, KILL is empty
  - B3: $a + b$ in GEN, KILL is empty

- Confluence (meet) operator $\wedge$?
  - NOT bit-wise logical AND

- Initial value $AVAIL(B)$
  - $AVAIL(B) = ENTIRE SET (a + b)$

---

**Constant Propagation**

A constant for a variable $x$ is **generated** if it is assigned a constant value.

A constant for variable $x$ is **killed** if $x$ is reassigned a different value, even if it is a constant value.

$$CONST(B) = \bigcap (GEN(Bi) \cup [CONST(Bi) - KILL(Bi)])$$

$Bi \in PRED(B)$

$GEN(Bi)$: Set of immediate constants (static) and dynamically encountered constants (dynamic)

$KILL(Bi)$: All definitions of variables $x$ defined in $Bi$
Very Busy Expressions

- Universe of facts?
  → All possible subsets of variable * constant in the program

- GEN and KILL sets for each basic block?
  → GEN = variable assigned constant value in basic block AND
    variable not redefined before reaching end of basic block
  → KILL = variable defined to non-constant value in basic block

- Initial values of CONST(B) before propagation starts?
  → ∅
  - Assumes variable is not constant until proven otherwise
  - The conservative assumption

Constant Propagation Example

\[
\begin{align*}
B1: & \quad x = 1 \\
B2: & \quad x = 1 \\
B3: & \quad y = 2 + x \\
B4: & \quad x = 2 \\
\end{align*}
\]

\[
C(B) = \bigcap \left( \{ \text{GEN}(B) \} \cup \{ \text{KILL}(B) \} \right)
\]

B₁ ∈ PRED(B)
The set of facts (universe of facts) are set of pairs 
\((\text{variable}, \text{value})\)
where value is from the following lattice:

![Lattice Diagram]

Examples: \((a, \top)\), \((b, 5)\)

Note: Typically, constant propagation is only performed on integral values, not floating point values.

The confluence (meet) operator \(\land\) for the second component of a \((\text{variable}, \text{value})\) pair is defined as follows:

<table>
<thead>
<tr>
<th></th>
<th>(\top)</th>
<th>(c_2)</th>
<th>(\bot)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\land)</td>
<td>(\top)</td>
<td>(c_2)</td>
<td>(\bot)</td>
</tr>
<tr>
<td>(\top)</td>
<td>(\top)</td>
<td>(c_2)</td>
<td>(\bot)</td>
</tr>
<tr>
<td>(c_1)</td>
<td>(c_1)</td>
<td>If (c_1 = c_2) then (c_1) else (\bot)</td>
<td>(\bot)</td>
</tr>
<tr>
<td>(\bot)</td>
<td>(\bot)</td>
<td>(\bot)</td>
<td>(\bot)</td>
</tr>
</tbody>
</table>
Non Bit-vector Problem

i := 1

j := i

i := j

i := i + 1

Remarks:

• optimistic constant propagation

• constant propagation typically done on UD/DU chains, but similar principles apply

• pessimistic constant propagation does not find any constants in our example

• intermediate results are not safe in optimistic approach

Dead Code Elimination

How to use DU/UD chains for dead code elimination?

Terminology:

UD(S, v): set of statements that contain definitions for variable v that reach the use of v in statement S

DU(S, v): set of statements that contain used of variable v that are reached by the definition of v in statement S
Dead Code Elimination

Two flavors of dead code: A statement can be considered dead code if

(1) it will never be executed, or

(2) it may be executed, but the result will never be used (including side effects)

The discussed algorithm uses UD chains.

Terminology:

DEFS(S): set of variables that may be written in statement S.

USES(S): set of variables that may be read in statement S.

CONTROL(S): set of statements such that S is control dependent on these statements

S is control dependent on S': S' contains a control flow branch, and there are execution paths from S' to the program exit such that S is on one path, but not on the other.
Dead Code Elimination

Report all statements that are not on any path from the entry to exit node as dead code, and remove them.

Initialize all statements as not useful.

Initialize worklist with critical statements (print statements).

While worklist is not empty Do

Remove a statement from worklist, call it S; mark S useful.

Mark all S’ in CONTROL(S) useful and add them to worklist.

For all variables v in USES(S) add each S’ in UD(S,v) to worklist EndForall

EndWhile

Report all unmarked statements as dead code.