1 Control flow analysis

1. Dominator sets:
   \[ D(1) = \{1\} \]
   \[ D(2) = \{1,2\} \]
   \[ D(3) = \{1,2,3\} \]
   \[ D(4) = \{1,2,4\} \]
   \[ D(5) = \{1,2,3,5\} \]
   \[ D(6) = \{1,2,3,5,6\} \]
   \[ D(7) = \{1,2,3,5,7\} \]
   \[ D(8) = \{1,2,3,5,6,8\} \]

   Dominator tree:
2. There are three loops:

(a) For back edge $3 \rightarrow 2$, loop is $\{2, 3\}$;
(b) For back edge $8 \rightarrow 5$, loop is $\{5, 6, 7, 8\}$;
(c) For back edge $8 \leftrightarrow 2$, loop is $\{2, 3, 5, 6, 7, 8\}$.

Therefore, the first two loops are nested in the third loop.

3. This control flow graph is not reducible. Because after we have removed all back edges (i.e., $3 \rightarrow 2$, $8 \rightarrow 5$ and $8 \rightarrow 2$), there is still a cycle $\{6, 7\}$ in the graph.

2 Depth first numbering

1. Depth first search: $1 \rightarrow 2 \rightarrow 3 \rightarrow 8 \rightarrow 8 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 5 \rightarrow 4 \rightarrow 6 \rightarrow 6 \rightarrow 2 \rightarrow 2$.

   Depth first numbering: $124657389$.

2. Back edges: $5 \rightarrow 2$ and $7 \rightarrow 4$.

   Loops: $\{2, 3, 4, 5, 6, 7, 8\}$ and $\{4, 5, 6, 7\}$.
Sample Problems: Data Flow Framework

1. Reachable uses – for each definition identify the set of uses reachable by the definition (used for computing def-use chains);

   - Data flow equations.
     
     This problem is similar to live variables. A use is reachable if there is a X-clear path from the definition to it.
     
     IN[B]: uses live at B's entry.
     OUT[B]: uses live at B’s exit.
     \[
     \begin{align*}
     \{ & \text{OUT}[B] = \bigcup_{s \in \text{succ}(B)} \text{IN}[s] \\
     \{ & \text{IN}[B] = \text{GEN}[B] \cup (\text{OUT}[B] - \text{KILL}[B])
     \end{align*}
     \]
     
     GEN[B]: uses in B prior to any definition of the same variable in B.
     KILL[B]: uses whose variables are redefined in B.

   - Lattice value. We can use USES to denote the set of all uses. So the lattice value \( \mathcal{L} = \mathcal{P}(\text{USES}) \).
   - Meet operator \( \wedge: \bigcup \)
   - Top element: \( \emptyset \); bottom elements: USES.
   - Partial order relation: \( \supseteq \)
   - Pictorial representation of the partial order. USES = \{u1, u2, u3\}.

   \[
   \begin{array}{c}
   \top \\
   (u_1) \downarrow \quad (u_2) \downarrow \quad (u_3) \\
   (u_1, u_2) \downarrow \quad (u_1, u_3) \quad \quad (u_2, u_3) \\
   (u_1, u_2, u_3) \downarrow \\
   \bot
   \end{array}
   \]

   - Transfer functions.
     \[
     f_n(x) = \text{GEN}[n] \cup (x - \text{KILL}[n])
     \]
     
     \[
     \begin{align*}
     \text{GEN}[n] = \{ u \in \text{USES} | \text{contains use } u \text{ whose variable is not redefined later in } n \} \\
     \text{KILL}[n] = \{ u \in \text{USES} | \text{defines the variable used by } u \}
     \end{align*}
     \]

2. Reaching uses – given a definition of variable x, identify the set of uses of x that are encountered prior to reaching the definition and there is no other definitions of x that intervene the use and the definition (used for computing antidependences);
• Data flow equations.
  \[ \begin{align*}
  \text{IN}[B] & = \bigcup_{s \in \text{pred}(B)} \text{OUT}[s] \\
  \text{OUT}[B] & = \text{GEN}[B] \cup (\text{IN}[B] - \text{KILL}[B])
  \end{align*} \]

  \text{GEN}[B]: \text{uses within } B \text{ that reach the end of } B.
  \text{KILL}[B]: \text{uses of the variables that are redefined in } B.

• Lattice value. We can use USES to denote the set of all uses. So the lattice value \( \mathcal{L} = \mathcal{P}(\text{USES}) \).
  \text{Meet operator } \wedge: \bigcup
  \text{Top element}: \emptyset; \text{bottom elements}: \text{USES}.
  \text{Partial order relation}: \supseteq
  \text{Pictorial representation of the partial order. USES} = \{u_1, u_2, u_3\}.

\[ \begin{array}{c}
  \top \\
  \downarrow \\
  \downarrow \\
  \downarrow \\
  \downarrow \\
  \downarrow \\
  \downarrow
\end{array} \]

\begin{align*}
  \{u_1\} & \leftarrow \{u_2\} & \leftarrow \{u_3\} \\
  \{u_1, u_2\} & \leftarrow \{u_1, u_3\} & \leftarrow \{u_2, u_3\}
\end{align*}

• Transfer functions.
  \[ f_n(x) = \text{GEN}[n] \cup (x - \text{KILL}[n]) \]

  \text{GEN}[n] = \{u \in \text{USES}|n \text{ contains use } u \text{ which reach the end of } n\} \\
  \text{KILL}[n] = \{u \in \text{USES}|n \text{ defines the variable used by } u\}

3. Classify the value of each program variable at each program point into one of the following categories: (a) the value is a \textit{unique constant} – you must also identify this constant value; (b) the value is \textit{one-of-many} constants – you are not actually compute the identities of these constants as part of your solution; and (c) the value is \textit{not-a-constant}, that is, it is neither a unique constant nor a one-of-many constants.

Single Variable.

  • Lattice for a single variable.
Meet operator for a single variable (using M to denote one-of-many):

\[
\begin{align*}
\text{any } \land \top &= \text{any} \\
\text{any } \land \bot &= \bot \\
c \land c &= c \\
c_1 \land c_2 &= M \quad (c_1 \neq c_2) \\
c \land M &= M \\
M \land M &= M
\end{align*}
\]

Now, we consider all variables.

- \(\text{VAR}\)- set of all variables
- \(\text{VAL}\)- set of all values (all constants, \(M, \top, \bot\))

Lattice for all variables \((L, \land')\).

- Lattice value.
  \[\mathcal{L} = \{\beta | \text{total function } \beta : \text{VAR} \rightarrow \text{VAL}\}\]

- Meet operator: \(\land'\).
  Definition of \(\land'\):
  - \(\beta_{\top}\) maps each variable to \(\top\)
  - \(\beta_{\bot}\) maps each variable to \(\bot\)
  - \((\beta_1 \land' \beta_2)(v) = \beta_1(v) \land \beta_2(v)\)

- Top element: \(\beta_{\top}\); bottom elements: \(\beta_{\bot}\).

- Partial order relation: \(\leq\)
  Definition of \(\leq\): \(\beta_1 \leq \beta_2\) iff \(\forall v \in \text{VAR}, \beta_1(v) \leq \beta_2(v)\)

- Transfer functions.
  For statement \(S : A = B \text{ op } C\):

\[
f_s(\beta) = \begin{cases} 
\beta(v) & \forall v \in \text{VAR} - \{A\} \\
\beta(A) = \bot & \text{if } \beta(B) = \bot \text{ or } \beta(C) = \bot \quad \text{(no change)} \\
\beta(A) = \top & \text{else if } \beta(B) = \top \text{ or } \beta(C) = \top \quad \text{(still undefined)} \\
\beta(A) = \bot & \text{else if } \beta(B) = M \text{ or } \beta(C) = M \quad \text{(not a constant)} \\
\beta(A) = c_B \text{ op } c_C & \text{else if } \beta(B) = c_B \text{ and } \beta(C) = c_C \quad \text{(constant)} 
\end{cases}
\]
• Data flow equations.

IN[B]: the mapping of variables at B’s entry.

OUT[B]: the mapping of variables at B’s exit.

\[
\begin{align*}
\text{IN}[\text{B}] &= \bigwedge'_{s \in \text{pred}(\text{B})} \text{OUT}[s] \\
\text{OUT}[\text{B}] &= f_B(\text{IN}[\text{B}])
\end{align*}
\]

where \( f \) is the transfer function of block B

4. Postdominator set of a node is the set of nodes that are encountered along all paths from the node to the end node of the control flow graph;

• Data flow equations.

\[
\begin{align*}
\text{OUT}[\text{B}] &= \bigcap_{s \in \text{succ}(\text{B})} \text{IN}[s] \\
\text{IN}[\text{B}] &= \{B\} \cup \text{OUT}[\text{B}]
\end{align*}
\]

• Lattice value. We can use N to denote the set of nodes. So the lattice value \( \mathcal{L} = \mathbb{P}(N) \).

• Meet operator \( \wedge: \cap \)

• Top element: \( N \); bottom elements: \( \emptyset \).

• Partial order relation: \( \subseteq \)

• Pictorial representation of the partial order. \( N = \{n_1, n_2, n_3\} \).

\[
\begin{array}{c}
\top \quad \{n_1, n_2, n_3\} \\
\{n_1, n_2\} \quad \{n_1, n_3\} \quad \{n_2, n_3\} \\
\{n_1\} \quad \{n_2\} \quad \{n_3\} \\
\bot
\end{array}
\]

• Transfer functions.

\[ f_n(x) = \{n\} \cup x \]
1. For the code segment below:
   a) Compute the dominance frontier for all nodes;
   b) Identify the placement points of phi nodes for each variable; and
   c) Provide the final SSA-form.

```
P = 1
Q = 2

If ()

P = P + Q
K = 1

K = P + 1

If ()

P = K + P
Q = K + Q

If ()

K = 2
Q = Q + 1

If ()
```
CS 201  Homework 2.

1. (a) 

(b) Variable Node where assigned value Node where Φ should be placed

<table>
<thead>
<tr>
<th>Variable</th>
<th>1,3,7</th>
<th>2,5,6</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td>1,4,7</td>
<td>2,5,6</td>
</tr>
<tr>
<td>K</td>
<td>3,4,5</td>
<td>2,5</td>
</tr>
</tbody>
</table>
(c)

\[ P_1 = 1 \]
\[ Q_1 = 2 \]

\[ P_2 = \varnothing (P_1, P_5) \]
\[ Q_2 = \varnothing (Q_1, Q_5) \]
\[ I_P() \]

\[ P_3 = P_2 + Q_2 \]
\[ k_1 = 1 \]

\[ K_2 = 2 \]
\[ Q_3 = Q_2 + 1 \]

\[ P_4 = \varnothing (P_3, P_2) \]
\[ Q_4 = \varnothing (Q_2, Q_3) \]
\[ k_3 = \varnothing (k_1, k_2) \]
\[ K_4 = P_4 + 1 \]

\[ P_5 = \varnothing (P_4, P_6) \]
\[ Q_5 = \varnothing (Q_4, Q_6) \]
\[ I_P() \]

\[ P_6 = K_4 + P_5 \]
\[ Q_6 = K_4 + Q_5 \]

\[ I_P() \]

-k could have a \( \varnothing \) rule here, but it doesn't really matter.
Q1. We would like to construct a constant propagation algorithm that identifies if the value of each variable at each program point is one of the following constants: -1, 0, +1. Which of the two lattices will you use in solving this problem? Provide a clear justification for your choice. Hint: Note that in the second lattice all constants other than -1, 0, +1 are treated as \textit{not-a-constant}.

\begin{center}
\begin{tikzpicture}[scale=0.8]
    \node (undef) at (0,0) {undef};
    \node (two) at (-2,-2) {-2};
    \node (minus) at (-1,-2) {-1};
    \node (zero) at (0,-2) {0};
    \node (plus) at (1,-2) {+1};
    \node (plus2) at (2,-2) {+2};
    \draw (undef) -- (two);
    \draw (undef) -- (minus);
    \draw (undef) -- (zero);
    \draw (undef) -- (plus);
    \draw (undef) -- (plus2);
    \draw (two) -- (zero);
    \draw (two) -- (plus);
    \draw (two) -- (plus2);
    \draw (minus) -- (zero);
    \draw (minus) -- (plus);
    \draw (minus) -- (plus2);
    \node (notaconstant) at (4,0) {not-a-constant};
    \node (minus1) at (6,-2) {-1};
    \node (zero1) at (7,-2) {0};
    \node (plus1) at (8,-2) {+1};
    \draw (notaconstant) -- (minus1);
    \draw (notaconstant) -- (zero1);
    \draw (notaconstant) -- (plus1);
    \draw (minus1) -- (zero1);
    \draw (minus1) -- (plus1);
    \draw (zero1) -- (plus1);
\end{tikzpicture}
\end{center}

Q2. Consider the restricted class of programs in which any expression on the right hand side of an assignment statement involves no more than one variable, i.e. other operands if present are constants. The data flow analysis for the \textit{constant propagation and folding} problem for general programs is known to be non-distributive. If this also the case for the above restricted class of programs? Explain.
\textbf{Soln. #3}

\begin{align*}
\text{def} & : \quad \land : \quad \text{def} \land \text{def} = \text{def} \\
\text{undef} & : \quad \text{undef} \land \text{def} = \text{undef} \\
& : \quad \text{undef} \land \text{undef} = \text{undef} \\

f_n(v): & \quad w \neq v \\
\text{fn}(v) = \text{def} & \quad f_n(v) = \text{identity} \\
\text{Initialization:} & \\
\forall u, \; \text{IN}[s, u] = \text{undef} \quad \text{where } s \text{ is the start node} \\
\text{IN}[\text{def}, u] = \text{def} \\

\text{Equations:} & \\
\text{IN}[n, u] &= \land_{p \in \text{pred}(n)} f_p(u)(\text{IN}[p, u]) \\

\text{Detection:} & \\
\forall n, \; \text{if } n \text{ uses } u \text{ and } \text{IN}[n, u] = \text{undef} \\
\text{then this use of } u \text{ in } n \text{ is a potentially undefined variable}.
\end{align*}

Q3. Develop an algorithm for identifying potential uses of uninitialized variables, i.e., uses of variables that are not always preceded by their definition.
Sample Problem

For the given code segment perform conditional constant propagation and folding:

(a) Using data flow based algorithm; and
(b) Using SSA-form based algorithm.

A = 1;  W = 1;  B = 2;
if (A != B) then Z = B + A
else Z = B - A  endif
do
    W = 2;
    if (Z < 0) then V = Z + 1
    else V = Z - 1  endif;
    B = B - 1;
while (B <= 0)
Print V+W;
Sample Problem

X = A + 1; Y = B + 1;
A = A + 1; B = B + 1;
else V = A + Z;
end;
W = A + 1; C = Z + 1;
if ( ... ) then
W = Z; R = Z + 1;
Z = 1; P = Z + 1;
while ( ... ) do
X = A + B; Y = B + A;
input A; input B;

variables
classes of
congruence
value numbering
perform global
code segment
For the given
input A1
input B1
X1 = A1 + B1
Y1 = B1 + A1
while (...) do
  A3 = φ(A1, A2)
  B3 = φ(B1, B2)
  X3 = φ(X1, X2)
  Y3 = φ(Y1, Y2)
  Z1 = 1
  P1 = Z1 + 1
  W1 = 2
  R1 = Z1 + 1
  if (...) then
    W2 = A3 + 1
    Q1 = Z1 + 1
  else
    V1 = A3 + Z1
  end if
  W3 = φ(W1, W2)
A2 = A3 + 1
B2 = B3 + 1
X2 = A2 + 1
Y2 = B2 + 1
end while
4.4. Code Optimization

Develop an algorithm that eliminates those fully redundant evaluations of an expression exp, that can be eliminated without introducing new temporary variables in the program. The example below illustrates the optimization. Note that \( D = P + Q \) is replaced by \( D = Z \) eliminating redundancy without introducing a temporary variable; however, redundancy due to \( X + Y \) cannot be eliminated.

Before Optimization

After Optimization
\[ \text{undef} \]
\[ \text{v}_1, \text{v}_2, \ldots, \text{v}_n \quad \text{(variables)} \]
\[ \text{not-avail} \quad \text{(not available in any variable)} \]

\[ \wedge \quad \text{any} \wedge \text{undef} = \text{any} \]
\[ \text{v} \wedge \text{v} = \text{v} \]
\[ \text{v}_1 \wedge \text{v}_2 = \text{not-avail} \]
\[ \text{any} \wedge \text{not-avail} = \text{not-avail} \]

\[ f(n, \text{exp}) : \]
\[ \text{generates} \]
\[ \text{kills} \]
\[ \text{uses variable v} \]
\[ f(n, \text{exp}) = \text{not-avail} \]
\[ f(n, \text{exp}) = \text{identity} \]

\[ \text{Instructions:} \]
\[ \forall \text{exp}, \quad \text{IN}[s, \text{exp}] = \text{not-avail} \quad \text{where} \quad s \quad \text{is the start node} \]
\[ \text{IN}[n \neq s, \text{exp}] = \text{undef} \]

\[ \text{Equations:} \]
\[ \text{IN}[n, \text{exp}] = \wedge_{p \in \text{pred}(n)} f(p, \text{exp}) \quad (\text{IN}[p, \text{exp}]) \]

\[ \text{Transformation:} \]
\[ \text{for each assignment} \quad n : \quad \text{v} = \text{exp} \]
\[ \quad \text{if} \quad \text{IN}[n, \text{exp}] = w \quad \text{then} \]
\[ \quad \text{replace} \quad \text{v} = \text{exp} \quad \text{by} \quad \text{v} = w. \]
3. Develop a general version of live variable analysis that classifies a variable as being: (a) **definitely live** - the value of variable is definitely live at a program point if it going to be used later no matter what path is taken; (b) **potentially live** - the value of a variable is possibly live at a program point if it may be used later; or (c) **not live**. Since this is a partitionable problem, please just present the analysis for a single variable.

*Your solution must specify the following:* $L$, the meet operator, the partial order, transfer function, data flow equations, and initialization of data flow values.

- $\text{not-live}$, $\text{pot-live}$, $\text{def-live}$, $\text{undef}$, $\text{any}$

- $\text{any} \land \text{undef} = \text{any}$
- $\text{any} \land \text{pot-live} = \text{pot-live}$
- $\text{not-live} \land \text{def-live} = \text{pot-live}$

- $f_S(s) = \begin{cases} \text{def-live} & \text{if } S \text{ uses variable } v \\ \text{not-live} & \text{else if } S \text{ defines variable } v \\ x & \text{otherwise} \end{cases}$

- $IN[S,v] = f_S(\bigwedge \{IN[n,v] \mid n \in \text{succ}(S)\})$

*Initialization:*  
- $IN[e,v] = \text{not-live}$ \quad (e - exit node)
- $IN[n,v] = \text{undef}$ \quad (\forall n \in N - \{e\}) \quad (i.e. all other nodes)
\[ \text{IN}(B, v) = \bigwedge_{p \in \text{pred}(B)} \text{OUT}(p, v) \wedge \bigwedge_{p \in T-\text{pred}(B)} \text{OUT-T}(p, v) \wedge \bigwedge_{p \in F-\text{pred}(B)} \text{OUT-F}(p, v) \]

**B contains predicate** \( v \leq c \):

\[ \text{OUT-T}(B, v) = \lambda \leq v \leq \min(c, u) \text{ st } \lambda \leq v \leq u \in \text{IN}(B, v) \]

\[ \text{OUT-F}(B, v) = \max(l, c+1) \leq v \leq u \text{ st } \lambda \leq v \leq u \in \text{IN}(B, v) \]

**B contains copy assignment** \( v = w \):

\[ \text{OUT}(B, v) = \text{IN}(B, w) \]

**B contains statement that redefines \( v \) (e.g., \text{Read} v)\:**

\[ \text{OUT}(B, v) = \min \leq v \leq \max \]

You are given a program that uses integer variables – the values of integer variables can vary between MIN and MAX. Let \( \text{Range}(v, p) = (lb, ub) \) denote the range of \( v \)'s value at program point \( p \). In other words, at program point \( p \), \( v \)'s value satisfies the following constraints \( lb \leq \text{Value}(v) \leq ub \).

If a program contains branch predicates of the form \( v \leq \text{constant} \), then along the true and false paths of the branches the range of \( v \)'s value can be further refined (i.e., narrowed further from \((MIN, MAX)\)) as illustrated in the example below.

![Data Flow Diagram](image)

Present data flow analysis that exploits the above observation to compute the range of a variable's value at various program points as illustrated in the above example. For simplicity, you can assume that there are no loops in the program.