Bit Vector Data Flow Frameworks

Uday Khedker
(www.cse.iitb.ac.in/~uday)

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

Jul 2017

Part 1

About These Slides

Copyright

These slides constitute the lecture notes for CS618 Program Analysis course at IIT Bombay and have been made available as teaching material accompanying the book:

  (Indian edition published by Ane Books in 2013)

Apart from the above book, some slides are based on the material from the following books


These slides are being made available under GNU FDL v1.2 or later purely for academic or research use.

Outline

• Live Variables Analysis
• Observations about Data Flow Analysis
• Available Expressions Analysis
• Anticipable Expressions Analysis
• Reaching Definitions Analysis
• Common Features of Bit Vector Frameworks
• Partial Redundancy Elimination
Defining Live Variables Analysis

A variable \( v \) is live at a program point \( p \), if some path from \( p \) to program exit contains an r-value occurrence of \( v \) which is not preceded by an l-value occurrence of \( v \).

Path based specification

\[
\begin{align*}
&v \text{ is live at } p \\
&a = v + 2 \\
&v = v + 2
\end{align*}
\]

Basic Blocks \( \equiv \) Single statements or Maximal groups of sequentially executed statements

Local Data Flow Properties

Control Transfer
Local Data Flow Properties for Live Variables Analysis

- \( r \)-value occurrence: Value is only read, e.g. \( x, y, z \) in \( x.\text{sum} = y.\text{data} + z.\text{data} \)
- \( l \)-value occurrence: Value is modified e.g. \( y \) in \( y = x.\text{lptr} \)

\[
\begin{align*}
\text{Gen}_n &= \{ v \mid \text{variable } v \text{ is used in basic block } n \text{ and is not preceded by a definition of } v \} \\
\text{Kill}_n &= \{ v \mid \text{basic block } n \text{ contains a definition of } v \}
\end{align*}
\]

Defining Data Flow Analysis for Live Variables Analysis

- \( \text{In}_n = \text{Out}_n - \text{Kill}_n \cup \text{Gen}_n \)
- \( \text{Out}_n = \bigcup_{s \in \text{succ}(n)} \text{In}_s \) otherwise

Global Data Flow Properties

- Specifications based on Edge

Data Flow Equations for Our Example

1. \( w = x \)
2. \( \text{while } (x.\text{data} < \text{max}) \)
3. \( y = x.\text{lptr}, x = x.\text{rptr} \)
4. \( z = \text{New class of } z \) (Cyclic Dependence)
5. \( y = y.\text{lptr} \)
6. \( z.\text{sum} = x.\text{data} + y.\text{data} \)

\[
\begin{align*}
\text{In}_1 &= (\text{Out}_1 - \text{Kill}_1) \cup \text{Gen}_1 \\
\text{Out}_1 &= \text{In}_2 \\
\text{In}_2 &= (\text{Out}_2 - \text{Kill}_2) \cup \text{Gen}_2 \\
\text{Out}_2 &= \text{In}_3 \cup \text{In}_4 \\
\text{In}_3 &= (\text{Out}_3 - \text{Kill}_3) \cup \text{Gen}_3 \\
\text{Out}_3 &= \text{In}_2 \\
\text{In}_4 &= (\text{Out}_4 - \text{Kill}_4) \cup \text{Gen}_4 \\
\text{Out}_4 &= \text{In}_5 \\
\text{In}_5 &= (\text{Out}_5 - \text{Kill}_5) \cup \text{Gen}_5 \\
\text{Out}_5 &= \text{In}_6 \\
\text{In}_6 &= (\text{Out}_6 - \text{Kill}_6) \cup \text{Gen}_6 \\
\text{Out}_6 &= \text{In}_7 \\
\text{In}_7 &= (\text{Out}_7 - \text{Kill}_7) \cup \text{Gen}_7 \\
\text{Out}_7 &= \emptyset
\end{align*}
\]
Performing Live Variables Analysis

Initialization

\[ \text{Gen} = \{x\}, \text{Kill} = \{w\} \]
\[ w = x \]
\[ \text{Gen} = \{x\}, \text{Kill} = \{y\} \]
\[ y = x.lptr \]
\[ \text{Gen} = \{x\}, \text{Kill} = \{y\} \]
\[ y = y.lptr \]
\[ \text{Gen} = \{x, y, z\}, \text{Kill} = \{w\} \]
\[ z = \text{New class}\_\text{of}\_z \]
\[ z = \text{New class}\_\text{of}\_z \]
\[ z = \text{New class}\_\text{of}\_z \]
\[ z = \text{New class}\_\text{of}\_z \]
\[ z = \text{New class}\_\text{of}\_z \]
\[ z \text{.sum} = x\text{.data} + y\text{.data} \]

\[ \text{Gen} = \{x\}, \text{Kill} = \{y\} \]
\[ y = x.lptr \]
\[ \text{Gen} = \{x\}, \text{Kill} = \{y\} \]
\[ y = y.lptr \]
\[ \text{Gen} = \{x, y, z\}, \text{Kill} = \{w\} \]
\[ z = \text{New class}\_\text{of}\_z \]
\[ z = \text{New class}\_\text{of}\_z \]
\[ z = \text{New class}\_\text{of}\_z \]
\[ z = \text{New class}\_\text{of}\_z \]
\[ z = \text{New class}\_\text{of}\_z \]
\[ z \text{.sum} = x\text{.data} + y\text{.data} \]
Performing Live Variables Analysis

Gen = \{x\}, Kill = \{w\}
\[
\begin{align*}
    w &= x \\
    \text{Gen} &= \{x\}, \text{Kill} = \emptyset \\
    \text{while} \ (x.\text{data} < \text{max}) \\
    \text{Gen} &= \{x\}, \text{Kill} = \{y\} \\
    y &= x.\text{lptr} \\
    \text{Gen} &= \{x, y\}, \text{Kill} = \emptyset \\
    \text{z} &= \text{New class of} \ y \\
    \text{Gen} &= \{x, y, z\}, \text{Kill} = \emptyset \\
    \text{z.\text{sum}} &= x.\text{data} + y.\text{data}
\end{align*}
\]

Traversal

Iteration #2

Local Data Flow Properties for Live Variables Analysis

\[\text{In}_n = \text{Gen}_n \cup (\text{Out}_n \setminus \text{Kill}_n)\]

- Gen\(_n\) : Use not preceded by definition
  - Upwards exposed use

- Kill\(_n\) : Definition anywhere in a block
  - Stop the effect from being propagated across a block

<table>
<thead>
<tr>
<th>Case</th>
<th>Local Information</th>
<th>Example</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>v \notin Gen(_n) \notin Kill(_n)</td>
<td>a = b + c \ b = c * d</td>
<td>liveness of v is unaffected by the basic block</td>
</tr>
<tr>
<td>2</td>
<td>v \in Gen(_n) \notin Kill(_n)</td>
<td>a = b + c \ b = v * d</td>
<td>v becomes live before the basic block</td>
</tr>
<tr>
<td>3</td>
<td>v \notin Gen(_n) v \in Kill(_n)</td>
<td>a = b + c \ v = c * d</td>
<td>v ceases to be live before the basic block</td>
</tr>
<tr>
<td>4</td>
<td>v \in Gen(_n) v \in Kill(_n)</td>
<td>a = v + c \ v = c * d</td>
<td>liveness of v is killed but v becomes live before the basic block</td>
</tr>
</tbody>
</table>
Using Data Flow Information of Live Variables Analysis

- Used for register allocation
  If variable \( x \) is live in a basic block \( b \), it is a potential candidate for register allocation

- Used for dead code elimination
  If variable \( x \) is not live after an assignment \( x = \ldots \), then the assignment is redundant and can be deleted as dead code

Tutorial Problem 1: Round #2 of Dead Code Elimination

Local Data Flow Information

<table>
<thead>
<tr>
<th>Iteration 1</th>
<th>Iteration 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gen</td>
<td>Kill</td>
</tr>
<tr>
<td>( n1 )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( n2 )</td>
<td>( {a, n} )</td>
</tr>
<tr>
<td>( n3 )</td>
<td>( {a} )</td>
</tr>
<tr>
<td>( n4 )</td>
<td>( {a} )</td>
</tr>
<tr>
<td>( n5 )</td>
<td>( {a, b, c} )</td>
</tr>
<tr>
<td>( n6 )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

Global Data Flow Information

<table>
<thead>
<tr>
<th>Iteration 1</th>
<th>Iteration 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out</td>
<td>In</td>
</tr>
<tr>
<td>( n6 )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( n5 )</td>
<td>( {a, b, c} )</td>
</tr>
<tr>
<td>( n4 )</td>
<td>( {a, b} )</td>
</tr>
<tr>
<td>( n3 )</td>
<td>( {a} )</td>
</tr>
<tr>
<td>( n2 )</td>
<td>( {a, c, n} )</td>
</tr>
<tr>
<td>( n1 )</td>
<td>( {a, b, c, n} )</td>
</tr>
</tbody>
</table>

print "Hello" n6

Tutorial Problem 1: Round #3 of Dead Code Elimination

Local Data Flow Information

<table>
<thead>
<tr>
<th>Iteration 1</th>
<th>Iteration 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gen</td>
<td>Kill</td>
</tr>
<tr>
<td>( n1 )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( n2 )</td>
<td>( {a, n} )</td>
</tr>
<tr>
<td>( n3 )</td>
<td>( {a} )</td>
</tr>
<tr>
<td>( n4 )</td>
<td>( {a} )</td>
</tr>
<tr>
<td>( n5 )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( n6 )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

Global Data Flow Information

<table>
<thead>
<tr>
<th>Iteration 1</th>
<th>Iteration 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out</td>
<td>In</td>
</tr>
<tr>
<td>( n6 )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( n5 )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( n4 )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( n3 )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( n2 )</td>
<td>( {a, n} )</td>
</tr>
<tr>
<td>( n1 )</td>
<td>( {a, n} )</td>
</tr>
</tbody>
</table>

print "Hello" n6
Part 3

Some Observations

What Does Data Flow Analysis Involve?

• Defining the analysis. Define the properties of execution paths
• Formulating the analysis. Define data flow equations
  ▶ Linear simultaneous equations on sets rather than numbers
  ▶ Later we will generalize the domain of values
• Performing the analysis. Solve data flow equations for the given program flow graph
• Many unanswered questions

A Digression: Iterative Solution of Linear Simultaneous Equations

• Simultaneous equations represented in the form of the product of a matrix of coefficients (A) with the vector of unknowns (x)
  \[ Ax = b \]
• Start with approximate values
• Compute new values repeatedly from old values
• Two classical methods
  ▶ Gauss-Seidel Method (Gauss: 1823, 1826), (Seidel: 1874)
  ▶ Jacobi Method (Jacobi: 1845)

A Digression: An Example of Iterative Solution of Linear Simultaneous Equations

<table>
<thead>
<tr>
<th>Equations</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>4w = x + y + 32</td>
<td>w = x = y = z = 16</td>
</tr>
<tr>
<td>4x = y + z + 32</td>
<td></td>
</tr>
<tr>
<td>4y = z + w + 32</td>
<td></td>
</tr>
<tr>
<td>4z = w + x + 32</td>
<td></td>
</tr>
</tbody>
</table>

• Rewrite the equations to define w, x, y, and z
  \[ w = 0.25x + 0.25y + 8 \]
  \[ x = 0.25y + 0.25z + 8 \]
  \[ y = 0.25z + 0.25w + 8 \]
  \[ z = 0.25w + 0.25x + 8 \]
• Assume some initial values of \( w_0, x_0, y_0, \) and \( z_0 \)
• Compute \( w_t, x_t, y_t, \) and \( z_t \) within some margin of error
A Digression: Gauss-Seidel Method

<table>
<thead>
<tr>
<th>Equations</th>
<th>Initial Values</th>
<th>Error Margin</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w = 0.25x + 0.25y + 8 )</td>
<td>( w_0 = 24 )</td>
<td>( w_{i+1} - w_i \leq 0.35 )</td>
</tr>
<tr>
<td>( x = 0.25y + 0.25z + 8 )</td>
<td>( x_0 = 24 )</td>
<td>( x_{i+1} - x_i \leq 0.35 )</td>
</tr>
<tr>
<td>( y = 0.25z + 0.25w + 8 )</td>
<td>( y_0 = 24 )</td>
<td>( y_{i+1} - y_i \leq 0.35 )</td>
</tr>
<tr>
<td>( z = 0.25w + 0.25x + 8 )</td>
<td>( z_0 = 24 )</td>
<td>( z_{i+1} - z_i \leq 0.35 )</td>
</tr>
</tbody>
</table>

Iteration 1  | Iteration 2  | Iteration 3  \\
\( w_1 = 6 + 6 + 8 = 20 \)  | \( w_2 = 5 + 5 + 8 = 18 \)  | \( w_3 = 4.5 + 4.5 + 8 = 17 \) \\
\( x_1 = 6 + 6 + 8 = 20 \)  | \( x_2 = 5 + 5 + 8 = 18 \)  | \( x_3 = 4.5 + 4.5 + 8 = 17 \) \\
\( y_1 = 6 + 6 + 8 = 20 \)  | \( y_2 = 5 + 5 + 8 = 18 \)  | \( y_3 = 4.5 + 4.5 + 8 = 17 \) \\
\( z_1 = 6 + 6 + 8 = 20 \)  | \( z_2 = 5 + 5 + 8 = 18 \)  | \( z_3 = 4.5 + 4.5 + 8 = 17 \) \\

Iteration 4  | Iteration 5  \\
\( w_4 = 4.25 + 4.25 + 8 = 16.5 \)  | \( w_5 = 4.125 + 4.125 + 8 = 16.25 \) \\
\( x_4 = 4.25 + 4.25 + 8 = 16.5 \)  | \( x_5 = 4.125 + 4.125 + 8 = 16.25 \) \\
\( y_4 = 4.25 + 4.25 + 8 = 16.5 \)  | \( y_5 = 4.125 + 4.125 + 8 = 16.25 \) \\
\( z_4 = 4.25 + 4.25 + 8 = 16.5 \)  | \( z_5 = 4.125 + 4.125 + 8 = 16.25 \) \\

A Digression: Jacobi Method

Use values from the current iteration wherever possible

<table>
<thead>
<tr>
<th>Equations</th>
<th>Initial Values</th>
<th>Error Margin</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w = 0.25x + 0.25y + 8 )</td>
<td>( w_0 = 24 )</td>
<td>( w_{i+1} - w_i \leq 0.35 )</td>
</tr>
<tr>
<td>( x = 0.25y + 0.25z + 8 )</td>
<td>( x_0 = 24 )</td>
<td>( x_{i+1} - x_i \leq 0.35 )</td>
</tr>
<tr>
<td>( y = 0.25z + 0.25w + 8 )</td>
<td>( y_0 = 24 )</td>
<td>( y_{i+1} - y_i \leq 0.35 )</td>
</tr>
<tr>
<td>( z = 0.25w + 0.25x + 8 )</td>
<td>( z_0 = 24 )</td>
<td>( z_{i+1} - z_i \leq 0.35 )</td>
</tr>
</tbody>
</table>

Iteration 1  | Iteration 2  \\
\( w_1 = 6 + 6 + 8 = 20 \)  | \( w_2 = 5 + 5 + 8 = 18 \)  \\
\( x_1 = 6 + 6 + 8 = 20 \)  | \( x_2 = 5 + 5 + 8 = 18 \)  \\
\( y_1 = 6 + 5 + 8 = 19 \)  | \( y_2 = 5 + 5 + 8 = 18 \)  \\
\( z_1 = 5 + 5 + 8 = 18 \)  | \( z_2 = 4.375 + 4.375 + 8 = 16.8125 \) \\

Iteration 3  | Iteration 4  \\
\( w_3 = 4.3125 + 4.23375 + 8 = 16.54625 \) | \( w_4 = 16.20172 \) \\
\( x_3 = 4.23375 + 4.23375 + 8 = 16.436875 \) | \( x_4 = 16.17844 \) \\
\( y_3 = 4.23375 + 4.1365625 + 8 = 16.370 \) | \( y_4 = 16.13637 \) \\
\( z_3 = 4.1365625 + 4.11 + 8 = 16.34375 \) | \( z_4 = 16.09504 \) \\

Our Method of Performing Data Flow Analysis

- Round robin iteration
- Essentially Jacobi method
- Unknowns are the data flow variables In; and Out;
- Domain of values is not numbers
- Computation in a fixed order
  - either forward (reverse post order) traversal, or
  - backward (post order) traversal
over the control flow graph

Tutorial Problem 2 for Liveness Analysis

Draw the control flow graph and perform live variables analysis

```c
int f(int m, int n, int k)
{
    int a, i;
    for (i=m-1; i<k; i++)
    {
        if (i>=n)
            a = n;
        a = a+i;
    }
    return a;
}
```
The Semantics of Return Statement for Live Variables Analysis

"return a" is modelled by the statement "return value, in stack = a"

- If we assume that the statement is executed within the block
  \[ a \in BI \]

- If we assume that the statement is executed outside of the block and
  along the edge connecting the procedure to its caller
  \[ a \in BI \]

Solution of Tutorial Problem 2

<table>
<thead>
<tr>
<th>Block</th>
<th>Local Information</th>
<th>Iteration # 1</th>
<th>Global Information</th>
<th>Change in iteration # 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n_6 )</td>
<td>{ a }</td>
<td>\∅</td>
<td>\∅</td>
<td>{ a }</td>
</tr>
<tr>
<td>( n_5 )</td>
<td>{ a, i }</td>
<td>{ a, i }</td>
<td>\∅</td>
<td>{ a, i, k, n }</td>
</tr>
<tr>
<td>( n_4 )</td>
<td>{ i }</td>
<td>{ a }</td>
<td>{ i, n }</td>
<td>{ a, i, k, n }</td>
</tr>
<tr>
<td>( n_3 )</td>
<td>{ i, n }</td>
<td>\∅</td>
<td>{ a, i, n }</td>
<td>{ a, i, k, n }</td>
</tr>
<tr>
<td>( n_2 )</td>
<td>{ i, k }</td>
<td>\∅</td>
<td>{ a, i, n }</td>
<td>{ a, i, k, n }</td>
</tr>
<tr>
<td>( n_1 )</td>
<td>{ m }</td>
<td>{ i }</td>
<td>{ a, i, k, n }</td>
<td>{ a, k, m, n }</td>
</tr>
</tbody>
</table>

Interpreting the Result of Liveness Analysis for Tutorial Problem 2

- Is a live at the exit of \( n_5 \) at the end of iteration 1? Why?
  (We have used post order traversal)

- Is a live at the exit of \( n_5 \) at the end of iteration 2? Why?
  (We have used post order traversal)

- Show an execution path along which \( a \) is live at the exit of \( n_5 \)

- Show an execution path along which \( a \) is live at the exit of \( n_3 \)
  \( n_1 \rightarrow n_2 \rightarrow n_3 \rightarrow n_5 \rightarrow n_2 \rightarrow \ldots \)

- Show an execution path along which \( a \) is not live at the exit of \( n_3 \)
  \( n_1 \rightarrow n_2 \rightarrow n_3 \rightarrow n_4 \rightarrow n_2 \rightarrow \ldots \)

Tutorial Problem 3 for Liveness Analysis

Also write a C program for this CFG without using goto or break
Solution of Tutorial Problem 3

<table>
<thead>
<tr>
<th>Block</th>
<th>Local Information</th>
<th>Global Information</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gen_n</td>
<td>Kill_n</td>
</tr>
<tr>
<td>n6</td>
<td>{x}</td>
<td>{z}</td>
</tr>
<tr>
<td>n5</td>
<td>{c}</td>
<td>{z}</td>
</tr>
<tr>
<td>n4</td>
<td>{y, z, d}</td>
<td>{x, y}</td>
</tr>
<tr>
<td>n3</td>
<td>{y, z, d}</td>
<td>{x, y}</td>
</tr>
<tr>
<td>n2</td>
<td>{c}</td>
<td>∅</td>
</tr>
<tr>
<td>n1</td>
<td>∅</td>
<td>{x, y}</td>
</tr>
</tbody>
</table>

Interpreting the Result of Liveness Analysis for Tutorial Problem 3

- Why is z live at the exit of n5?
- Why is z not live at the entry of n5?
- Why is x live at the exit of n3 inspite of being killed in n4?
- Identify the instance of dead code elimination z = x in n6
- Would the first round of dead code elimination cause liveness information to change? Yes
- Would the second round of liveness analysis lead to further dead code elimination? Yes

Choice of Initialization

What should be the initial value of internal nodes?

- Confluence is ∪
- Identity of ∪ is ∅
- We begin with ∅ and let the sets at each program point grow
  A revisit to a program point
    ▶ may consider a new execution path
    ▶ more variables may be found to be live
    ▶ a variable found to be live earlier does not become dead

The role of boundary info BI explained later in the context of available expressions analysis

How Does the Initialization Affect the Solution?

```
Init.  Iter. #1  Iter. #2
a = b = 5  0  0  0
          0  \{b\}  \{b\}
          0  0  0
print b   0  0  0
          0  0  0
```
Soundness and Precision of Live Variables Analysis

Consider dead code elimination based on liveness information

- Spurious inclusion of a non-live variable
  - A dead assignment may not be eliminated
  - Solution is sound but may be imprecise
- Spurious exclusion of a live variable
  - A useful assignment may be eliminated
  - Solution is unsound
- Given $L_2 \supseteq L_1$ representing liveness information
  - Using $L_2$ in place of $L_1$ is sound
  - Using $L_1$ in place of $L_2$ may not be sound
- The smallest set of all live variables is most precise
  - Since liveness sets grow (confluence is $\cup$), we choose $\emptyset$ as the initial conservative value

Termination, Convergence, and Complexity

- For live variables analysis,
  - The set of all variables is finite, and
  - the confluence operation (i.e. meet) is union, hence
  - the set associated with a data flow variable can only grow
  \[ \Rightarrow \text{Termination is guaranteed} \]
- Since initial value is $\emptyset$, live variables analysis converges on the smallest set
- How many iterations do we need for reaching the convergence?
- Going beyond live variables analysis
  - Do the sets always grow for other data flow frameworks?
  - What is the complexity of round robin analysis for other analyses?
  \[ \text{Answered formally in module 2 (Theoretical Abstractions)} \]

Conservative Nature of Analysis (1)

- $\text{abs}(n)$ returns the absolute value of $n$
- Is $y$ live on entry to block $b_2$?
- By execution semantics, NO
  - Path $b_1 \rightarrow b_2 \rightarrow b_3$ is an infeasible execution path
- A compiler makes conservative assumptions:
  - All branch outcomes are possible
  - Consider every path in CFG as a potential execution path
- Our analysis concludes that $y$ is live on entry to block $b_2$

Conservative Nature of Analysis (2)

- Is $b$ live on entry to block $b_2$?
- By execution semantics, NO
  - Path $b_1 \rightarrow b_2 \rightarrow b_4 \rightarrow b_6$ is an infeasible execution path
- Is $c$ live on entry to block $b_3$?
  - Path $b_1 \rightarrow b_3 \rightarrow b_4 \rightarrow b_6$ is a feasible execution path
- A compiler make conservative assumptions:
  - Our analysis is path insensitive
  - Note: It is flow sensitive (i.e. information is computed for every control flow points)
- Our analysis concludes that $b$ is live at the entry of $b_2$
- Is $c$ live at the entry of $b_3$?
Conservative Nature of Analysis at Intraprocedural Level

- We assume that all paths are potentially executable
- Our analysis is path insensitive
  - The data flow information at a program point $p$ is path insensitive
    - Information at $p$ is merged along all paths reaching $p$
  - The data flow information reaching $p$ is computed path insensitively
    - Information is merged at all shared points in paths reaching $p$
    - May generate spurious information due to non-distributive flow functions

More about it in module 2

Conservative Nature of Analysis at Interprocedural Level

- Context insensitivity
  - Merges of information across all calling contexts
- Flow insensitivity
  - Disregards the control flow

More about it in module 4

What About Soundness of Analysis Results?

- No compromises
- We will study it in module 2

Part 4
Available Expressions Analysis
Defining Available Expressions Analysis

An expression $e$ is available at a program point $p$, if every path from program entry to $p$ contains an evaluation of $e$ which is not followed by a definition of any operand of $e$.

$\begin{align*}
a \ast b \text{ is available at } p & \\
& \text{ and } a \ast b \text{ is not available at } p
\end{align*}$

Local Data Flow Properties for Available Expressions Analysis

Gen

$\begin{align*}
\text{Gen}_n &= \{ e \mid \text{expression } e \text{ is evaluated in basic block } n \text{ and this evaluation is not followed by a definition of any operand of } e \}
\end{align*}$

Kill

$\begin{align*}
\text{Kill}_n &= \{ e \mid \text{basic block } n \text{ contains a definition of an operand of } e \}
\end{align*}$

Entity Manipulation Exposition

<table>
<thead>
<tr>
<th>Entity</th>
<th>Manipulation</th>
<th>Exposition</th>
</tr>
</thead>
</table>
| Gen
  | Expression   | Use        | Downwards  |
| Kill
  | Expression   | Modification | Anywhere   |

Data Flow Equations For Available Expressions Analysis

$\begin{align*}
\text{In}_n &= \left\{ \begin{array}{l}
\text{BI} \\
\bigcap_{p \in \text{pred}(n)} \text{Out}_p
\end{array} \right. \\
& \quad \text{n is Start block}
\end{align*}$

$\begin{align*}
\text{Out}_n &= \text{Gen}_n \cup (\text{In}_n - \text{Kill}_n)
\end{align*}$

Alternatively,

$\begin{align*}
\text{Out}_n &= f_n(\text{In}_n), \quad \text{where}
\text{Out}_n &= \text{Gen}_n \cup (\text{X} - \text{Kill}_n)
\end{align*}$

- $\text{In}_n$ and $\text{Out}_n$ are sets of expressions
- BI is $\emptyset$ for expressions involving a local variable

Using Data Flow Information of Available Expressions Analysis

- Common subexpression elimination
  - If an expression is available at the entry of a block $n$ ($\text{In}_n$) and a computation of the expression exists in $n$ such that it is not preceded by a definition of any of its operands ($\text{AntGen}_n$)
  - Then the expression is redundant

$\begin{align*}
\text{Redundant}_n &= \text{In}_n \cap \text{AntGen}_n
\end{align*}$

- A redundant expression is upwards exposed whereas the expressions in $\text{Gen}_n$ are downwards exposed
### An Example of Available Expressions Analysis

#### Initialisation

<table>
<thead>
<tr>
<th>Node</th>
<th>Gen</th>
<th>Kill</th>
<th>Available</th>
<th>Redund.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>0</td>
<td>0010</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0000</td>
<td>0110</td>
<td>1010</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0000</td>
<td>0011</td>
<td>1010</td>
</tr>
<tr>
<td>5</td>
<td>0001</td>
<td>0</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>6</td>
<td>0001</td>
<td>0</td>
<td>0000</td>
<td>0010</td>
</tr>
</tbody>
</table>

#### Iteration #1

<table>
<thead>
<tr>
<th>Node</th>
<th>Gen</th>
<th>Kill</th>
<th>Available</th>
<th>Redund.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>0</td>
<td>0010</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0000</td>
<td>0110</td>
<td>1010</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0000</td>
<td>0011</td>
<td>1010</td>
</tr>
<tr>
<td>5</td>
<td>0001</td>
<td>0</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>6</td>
<td>0001</td>
<td>0</td>
<td>0000</td>
<td>0010</td>
</tr>
</tbody>
</table>

#### Let $e_1 \equiv a \times b$, $e_2 \equiv b \times c$, $e_3 \equiv c \times d$, $e_4 \equiv d \times e$

#### Iteration #2

<table>
<thead>
<tr>
<th>Node</th>
<th>Gen</th>
<th>Kill</th>
<th>Available</th>
<th>Redund.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>0</td>
<td>0010</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0000</td>
<td>0110</td>
<td>1010</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0000</td>
<td>0011</td>
<td>1010</td>
</tr>
<tr>
<td>5</td>
<td>0001</td>
<td>0</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>6</td>
<td>0001</td>
<td>0</td>
<td>0000</td>
<td>0010</td>
</tr>
</tbody>
</table>

#### Let $e_1 \equiv a \times b$, $e_2 \equiv b \times c$, $e_3 \equiv c \times d$, $e_4 \equiv d \times e$
An Example of Available Expressions Analysis

Final Result

Let $e_1 = a \times b$, $e_2 = b \times c$, $e_3 = c \times d$, $e_4 = d \times e$

<table>
<thead>
<tr>
<th>Node</th>
<th>Gen</th>
<th>Kill</th>
<th>Available</th>
<th>Redund.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${e_1, e_2}$</td>
<td>1100</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
<tr>
<td>2</td>
<td>${e_1}$</td>
<td>0010</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
<tr>
<td>3</td>
<td>$\emptyset$</td>
<td>0000</td>
<td>${e_2, e_3}$</td>
<td>0110</td>
</tr>
<tr>
<td>4</td>
<td>$\emptyset$</td>
<td>0000</td>
<td>${e_3, e_4}$</td>
<td>0011</td>
</tr>
<tr>
<td>5</td>
<td>${e_1, e_3}$</td>
<td>1001</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
<tr>
<td>6</td>
<td>${e_4}$</td>
<td>0001</td>
<td>$\emptyset$</td>
<td>0000</td>
</tr>
</tbody>
</table>

Tutorial Problem 2 for Available Expressions Analysis

$$\text{Expr} = \{ a \times b, b + c \}$$

Solution of the Tutorial Problem 2

<table>
<thead>
<tr>
<th>Node</th>
<th>Local Information</th>
<th>Global Information</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gen</td>
<td>Kill</td>
</tr>
<tr>
<td></td>
<td>In$_n$</td>
<td>Out$_n$</td>
</tr>
<tr>
<td>$n_1$</td>
<td>11</td>
<td>00</td>
</tr>
<tr>
<td>$n_2$</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>$n_3$</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>$n_4$</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>$n_5$</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>$n_6$</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

Tutorial Problem 3 for Available Expressions Analysis

$$\text{Expr} = \{ a \times b, b + c, a + b \}$$
Solution of the Tutorial Problem 3

Bit vector \(a + b + b + c + a + b\)

<table>
<thead>
<tr>
<th>Node</th>
<th>Local Information</th>
<th>Global Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gen (_n)</td>
<td>Kill (_n)</td>
<td>AntGen (_n)</td>
</tr>
<tr>
<td>(n_2)</td>
<td>110</td>
<td>010</td>
</tr>
<tr>
<td>(n_3)</td>
<td>010</td>
<td>000</td>
</tr>
<tr>
<td>(n_4)</td>
<td>001</td>
<td>101</td>
</tr>
<tr>
<td>(n_5)</td>
<td>000</td>
<td>010</td>
</tr>
<tr>
<td>(n_6)</td>
<td>001</td>
<td>000</td>
</tr>
</tbody>
</table>

Why do we need 3 iterations as against 2 for previous problems?

The Effect of BI and Initialization on a Solution

This represents the expected availability information leading to elimination of \(a + c\) in node 3 (\(a \ast c\) is not redundant in node 3)

This misses the availability of \(a + c\) in node 3
Some Observations

- Data flow equations do not require a particular order of computation
  - **Specification.** Data flow equations define what needs to be computed and not how it is to be computed
  - **Implementation.** Round robin iterations perform the actual computation
    - Specification and implementation are distinct
- Initialization governs the quality of solution found
  - Only precision is affected, soundness is guaranteed
  - Associated with “internal” nodes
- **BI** depends on the semantics of the calling context
  - May cause unsoundness
  - Associated with “boundary” node (specified by data flow equations)
    - Does not vary with the method or order of traversal
A New Data Flow Framework: Partially available expressions analysis

- Expressions that are computed and remain unmodified along some path reaching \( p \)
- The data flow equations are same as that of available expressions analysis except that the confluence is changed to \( \cup \)

Perform partially available expressions analysis for the example program used for available expressions analysis.
Defining Reaching Definitions Analysis

- A definition \( d_x : x = e \) reaches a program point \( p \) if it appears (without a redefinition of \( x \)) on some path from program entry to \( p \)
  \((x \text{ is a variable and } e \text{ is an expression})\)

- Application: Copy Propagation
  A use of a variable \( x \) at a program point \( p \) can be replaced by \( y \) if \( d_x : x = y \) is the only definition which reaches \( p \) and \( y \) is not modified between the point of \( d_x \) and \( p \).

Using Reaching Definitions for Def-Use and Use-Def Chains

Def-Use Chains

1. \( a_1: a = 4 \)
2. \( b_1: b = 2 \)
3. \( c_1: c = 3 \)
4. \( n_1: n = c+2 \)
5. \( a: a = a+1 \)
6. \( a: a = a+1 \)

Use-Def Chains

1. \( a_1: a = 4 \)
2. \( b_1: b = 2 \)
3. \( c_1: c = 3 \)
4. \( n_1: n = c+2 \)
5. \( t_1 = a+b \)
6. \( t_1 = a+b \)
7. \( a: a = a+1 \)
8. \( a: a = a+1 \)
**Defining Data Flow Analysis for Reaching Definitions Analysis**

Let $d_v$ be a definition of variable $v$

$$Gen_n = \{ d_v | \text{variable } v \text{ is defined in basic block } n \text{ and this definition is not followed (within } n) \text{ by a definition of } v\}$$

$$Kill_n = \{ d_v | \text{basic block } n \text{ contains a definition of } v\}$$

<table>
<thead>
<tr>
<th>Entity</th>
<th>Manipulation</th>
<th>Exposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Gen_n$</td>
<td>Definition</td>
<td>Occurrence</td>
</tr>
<tr>
<td>$Kill_n$</td>
<td>Definition</td>
<td>Occurrence</td>
</tr>
</tbody>
</table>

**Data Flow Equations for Reaching Definitions Analysis**

$$In_n = \begin{cases}BL & n \text{ is Start block} \\ \bigcup_{p \in \text{pred}(n)} \text{Out}_p & \text{otherwise} \end{cases}$$

$$Out_n = Gen_n \cup (In_n - Kill_n)$$

$$BI = \{ d_x : x = \text{undef} | x \in \text{Var} \}$$

$In_n$ and $Out_n$ are sets of definitions.
**Tutorial Problem for Copy Propagation**

1. \( a_1: a = 4 \)
   \( b_1: b = 2 \)
   \( c_1: c = 3 \)
   \( n_1: n = 6 \)

2. if \((a > n)\)
   \( F \)
   \( T \)
   \( a_2: a = a + 1 \)

3. if \((a \geq 12)\)
   \( F \)
   \( T \)
   \( t_11: t_1 = a + b \)
   \( a_3: a = t_1 + c \)

4. if \((a \geq 12)\)
   \( F \)
   \( T \)
   \( t_11: t_1 = a + b \)
   \( a_3: a = t_1 + c \)

5. \( \) \( \)

6. print \( a \)

---

**Gen**

<table>
<thead>
<tr>
<th>n1</th>
<th>{a_1, b_1, c_1, n_1}</th>
<th>{a_0, b_0, b_0, a_1, a_2, a_3, b_0, a_1, b_0, a_0, b_0, a_0, n_0, n_1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>n2</td>
<td>{a_2}</td>
<td>{a_0, a_1, a_2, a_3}</td>
</tr>
<tr>
<td>n3</td>
<td>{a_3}</td>
<td>{a_0, a_1, a_2, a_3}</td>
</tr>
<tr>
<td>n4</td>
<td>{a_0, a_1, a_2, a_3}</td>
<td>{a_0, a_1, a_2, a_3}</td>
</tr>
<tr>
<td>n6</td>
<td>{a_0, a_1, a_2, a_3}</td>
<td>{a_0, a_1, a_2, a_3}</td>
</tr>
</tbody>
</table>

**Kill**

<table>
<thead>
<tr>
<th>n1</th>
<th>{a_0, b_0, c_0, n_0}</th>
<th>{a_1, b_1, c_1, n_1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>n2</td>
<td>{a_1, b_1, c_1, n_1}</td>
<td>{a_0, a_2, b_1, c_1, n_1}</td>
</tr>
<tr>
<td>n3</td>
<td>{a_1, b_1, c_1, n_1}</td>
<td>{a_1, b_1, c_1, n_1}</td>
</tr>
<tr>
<td>n4</td>
<td>{a_1, b_1, c_1, n_1}</td>
<td>{a_1, b_1, c_1, n_1}</td>
</tr>
<tr>
<td>n5</td>
<td>{a_1, b_1, c_1, n_1}</td>
<td>{a_1, b_1, c_1, n_1}</td>
</tr>
<tr>
<td>n6</td>
<td>{a_1, a_2, b_1, c_1, n_1}</td>
<td>{a_1, a_2, b_1, c_1, n_1}</td>
</tr>
</tbody>
</table>

---

**Iteration #1**

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>n1</td>
<td>{a_0, b_0, c_0, n_0}</td>
</tr>
<tr>
<td>n2</td>
<td>{a_1, b_1, c_1, n_1}</td>
</tr>
<tr>
<td>n3</td>
<td>{a_1, b_1, c_1, n_1}</td>
</tr>
<tr>
<td>n4</td>
<td>{a_1, b_1, c_1, n_1}</td>
</tr>
<tr>
<td>n5</td>
<td>{a_1, b_1, c_1, n_1}</td>
</tr>
<tr>
<td>n6</td>
<td>{a_1, a_2, b_1, c_1, n_1}</td>
</tr>
</tbody>
</table>

---

**Iteration #2**

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>n1</td>
<td>{a_0, b_0, c_0, n_0}</td>
</tr>
<tr>
<td>n2</td>
<td>{a_1, b_1, c_1, n_1}</td>
</tr>
<tr>
<td>n3</td>
<td>{a_1, b_1, c_1, n_1}</td>
</tr>
<tr>
<td>n4</td>
<td>{a_1, b_1, c_1, n_1}</td>
</tr>
<tr>
<td>n5</td>
<td>{a_1, b_1, c_1, n_1}</td>
</tr>
<tr>
<td>n6</td>
<td>{a_1, a_2, a_3, b_1, c_1, n_1}</td>
</tr>
</tbody>
</table>
### Tutorial Problem for Copy Propagation

1. \(a_1: a = 4\)
   \(b_1: b = 2\)
   \(c_1: c = 3\)
   \(n_1: n = 6\)

2. If \((a > n)\)
   \(T\)
   \(F\)
   \(\{a_1, a_2, b_1, c_1, n_1\}\)

3. \(a_2: a = a + 1\)

4. If \((a \geq 12)\)
   \(T\)
   \(F\)
   \(\{t_1: t_1 = a + b_2\}
   \(\{a_3: a = t_1 + c_3\}\}
   \(\{a_1, a_2, a_3, b_1, c_1, n_1\}\)

5. \(t_1: t_1 = a + 2\)
   \(a_3: a = t_1 + 3\)

6. \(\text{print } a\)

* So what is the advantage?

**Dead Code Elimination**

- Only \(a\) is live at the exit of 1
- Assignments of \(b, c,\) and \(n\) are dead code
- Can be deleted

### Defining Anticipable Expressions Analysis

- An expression \(e\) is anticipable at a program point \(p\), if every path from \(p\) to the program exit contains an evaluation of \(e\) which is not preceded by a redefinition of any operand of \(e\).
- Application: Safety of Code Placement
Safety of Code Placement

Placing \( a/b \) at the exit of 1 is unsafe (\( \equiv \) can change the behaviour of the optimized program)

A guarded computation of an expression should not be converted to an unguarded computation

Defining Data Flow Analysis for Anticipable Expressions Analysis

\[ \begin{align*}
Gen_n &= \{ e \mid \text{expression } e \text{ is evaluated in basic block } n \text{ and this evaluation is not preceded (within } n \text{) by a definition of any operand of } e \} \\
Kill_n &= \{ e \mid \text{basic block } n \text{ contains a definition of an operand of } e \}
\end{align*} \]

Entity | Manipulation | Exposition
---|---|---
Gen | Expression | Use | Upwards
Kill | Expression | Modification | Anywhere

Data Flow Equations for Anticipable Expressions Analysis

\[ \begin{align*}
In_n &= Gen_n \cup (Out_n - Kill_n) \\
Out_n &= \begin{cases} 
BI & n \text{ is End block} \\
\bigcap_{s \in succ(n)} In_s & \text{otherwise}
\end{cases}
\end{align*} \]

\( In_n \) and \( Out_n \) are sets of expressions

Tutorial Problem 1 for Anticipable Expressions Analysis

\[ \begin{align*}
\text{Expr} &= \{ a + b, b + c, b - c \}
\end{align*} \]
Solution of Tutorial Problem 1

<table>
<thead>
<tr>
<th>Block</th>
<th>Local Information</th>
<th>Global Information</th>
<th>Iteration # 1</th>
<th>Change in iteration # 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gen_6</td>
<td>Kill_6</td>
<td>Out_6</td>
<td>In_6</td>
</tr>
<tr>
<td>n_6</td>
<td>110</td>
<td>000</td>
<td>000</td>
<td>110</td>
</tr>
<tr>
<td>n_5</td>
<td>001</td>
<td>000</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>n_4</td>
<td>010</td>
<td>011</td>
<td>111</td>
<td>110</td>
</tr>
<tr>
<td>n_3</td>
<td>010</td>
<td>100</td>
<td>110</td>
<td>010</td>
</tr>
<tr>
<td>n_2</td>
<td>001</td>
<td>011</td>
<td>010</td>
<td>001</td>
</tr>
<tr>
<td>n_1</td>
<td>000</td>
<td>111</td>
<td>001</td>
<td>000</td>
</tr>
</tbody>
</table>

Solution of Tutorial Problem 2

<table>
<thead>
<tr>
<th>Block</th>
<th>Local Information</th>
<th>Global Information</th>
<th>Iteration # 1</th>
<th>Change in iteration # 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gen_6</td>
<td>Kill_6</td>
<td>Out_6</td>
<td>In_6</td>
</tr>
<tr>
<td>n_6</td>
<td>10</td>
<td>00</td>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td>n_5</td>
<td>01</td>
<td>11</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>n_4</td>
<td>00</td>
<td>00</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>n_3</td>
<td>10</td>
<td>01</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>n_2</td>
<td>10</td>
<td>10</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>n_1</td>
<td>10</td>
<td>01</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Part 7

Common Features of Bit Vector Data Flow Frameworks
Defining Local Data Flow Properties

- Live variables analysis

<table>
<thead>
<tr>
<th>Entity</th>
<th>Manipulation</th>
<th>Exposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Gen_n$</td>
<td>Variable Use</td>
<td>Upwards</td>
</tr>
<tr>
<td>$Kill_n$</td>
<td>Variable Modification</td>
<td>Anywhere</td>
</tr>
</tbody>
</table>

- Analysis of expressions

<table>
<thead>
<tr>
<th>Entity</th>
<th>Manipulation</th>
<th>Exposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Gen_n$</td>
<td>Expression Use</td>
<td>Downwards Upwards</td>
</tr>
<tr>
<td>$Kill_n$</td>
<td>Expression Modification</td>
<td>Anywhere Anywhere</td>
</tr>
</tbody>
</table>

Common Form of Data Flow Equations

$$X_i = f(Y_i)$$
$$Y_i = \cap X_j$$

Confluence

- So far we have seen $\cup$ and $\cap$. Could be other operations.

Data Flow Paths Discovered by Data Flow Analysis

- Liveness
- Anticipability
- Availability
- Partial Availability

A Taxonomy of Bit Vector Data Flow Frameworks

- Any Path
- All Paths

<table>
<thead>
<tr>
<th></th>
<th>Union</th>
<th>Intersection</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>Anticipable Expressions</td>
</tr>
<tr>
<td>Bidirectional (limited)</td>
<td></td>
<td>Partial Redundancy Elimination (Original M-R Formulation)</td>
</tr>
</tbody>
</table>
Sequence of blocks \( (n_1, n_2, \ldots, n_k) \) which is a prefix of some potential execution path starting at \( n_1 \) such that:

- \( n_k \) contains an upwards exposed use of \( v \), and
- no other block on the path contains an assignment to \( v \).

**Liveness**

Sequence of blocks \( (n_1, n_2, \ldots, n_k) \) which is a prefix of some potential execution path starting at \( n_1 \) such that:

- \( n_k \) contains an upwards exposed use of \( a \ast b \), and
- no other block on the path contains an assignment to \( a \) or \( b \), and
- every path starting at \( n_1 \) is an anticipability path of \( a \ast b \).

**Anticipability**

Sequence of blocks \( (n_1, n_2, \ldots, n_k) \) which is a prefix of some potential execution path starting at \( n_1 \) such that:

- \( n_1 \) contains a downwards exposed use of \( a \ast b \), and
- no other block on the path contains an assignment to \( a \) or \( b \).

**Availability**

Sequence of blocks \( (n_1, n_2, \ldots, n_k) \) which is a prefix of some potential execution path starting at \( n_1 \) such that:

- \( n_1 \) contains a downwards exposed use of \( a \ast b \), and
- no other block on the path contains an assignment to \( a \) or \( b \).
Data Flow Paths Discovered by Data Flow Analysis

Liveness

Anticipability

Availability

Partial Availability

Part 9

Partial Redundancy Elimination

Precursor: Common Subexpression Elimination

Code Fragment | Flow Graph | Remarks
--- | --- | ---
if (...)  
c = a*b;  
else  
d = a*b;  
e = a*b;  | 1 if (...)  
2 c = a*b  
3 d = a*b  
4 e = a*b | • a and b are not modified along paths 1 → 2 → 4 and 1 → 3 → 4  
• Computation of a*b in 4 is redundant  
• Previous value can be used

if (...)  
c = a*b;  
else  
d = a*b;  
e = a*b;  | 1 if (...)  
2 t = a*b  
\[c = t\]  
\[d = t\]  
4 e = t | • a and b are not modified along paths 1 → 2 → 4 and 1 → 3 → 4  
• Computation of a*b in 4 is redundant  
• Previous value can be used
Partial Redundancy Elimination

- Computation of $a \times b$ in 4 is
  - redundant along path $1 \to 2 \to 4$, but ...
  - not redundant along path $1 \to 3 \to 4$

Code Hoisting for Partial Redundancy Elimination

- Computation of $a \times b$ in 3 becomes totally redundant
- Can be deleted

PRE Subsumes Loop Invariant Movement

What’s that?
**PRE Subsumes Loop Invariant Movement**

1. \( a = b \times c \)
2. \( t = b \times c \)
3. \( a = b \times c \)

**PRE Can be Used for Strength Reduction**

- Replace \( \times \) by + in the loop
- Delete \( i = i + 1 \) in the loop
- Expression \( i \times 4 \) becomes loop invariant
- Hoist it and increment \( t_1 \) in the loop
**PRE Can be Used for Strength Reduction**

- $i = 0$
- $a = A[t1]$
- $t1 = t1 + 4$

- $\times$ in the loop has been replaced by $+$
- $i = i + 1$ in the loop has been eliminated

**Performing Partial Redundancy Elimination**

1. Identify partial redundancies
2. Identify program points where computations can be inserted
3. Insert expressions
4. Partial redundancies become total redundancies
   $\implies$ Delete them.

Morel-Renvoise Algorithm (*CACM*, 1979.)

**Defining Hoisting Criteria**

- An expression can be safely inserted at a program point $p$ if it is:
  - Available at $p$
  - Anticipable at $p$

- If it is available at $p$, then there is no need to insert it at $p$.
- If it is anticipable at $p$, then all such occurrences should be hoisted to $p$.
- An expression should be hoisted to $p$ provided it can be hoisted to $p$ along all paths from $p$ to exit.

**Safety of Hoisting an Expression**

- Predecessor Blocks
- Successor Blocks
- Entry
- Exit

- Basic Block
Safety of Hoisting an Expression

- **Safety of hoisting to the exit of a block**
  - S.1 Hoist only if it can be hoisted out of the entries of all successor blocks

- **Safety of hoisting to the entry of a block**
  - S.2 Hoist only if
    - S.2.a it is upwards exposed, or
    - S.2.b it can be hoisted to its exit and is transparent in the block

- **Safety of hoisting out of the entry of a block**
  - S.3 Hoist only if for each predecessor
    - S.3.a it can be hoisted to its exit, or
    - S.3.b it is available at its exit.

Anticipability and Code Hoisting

- What is the meaning of the assertion
  - "\(a \times b\) is anticipable at program point \(p\)"
    - \(a \times b\) is computed along every path from \(p\) to \(End\) before \(a\) or \(b\) are modified
    - The value computed at \(p\) would be same as the next value computed on any path
    - \(a \times b\) can be safely inserted at \(p\)

- It does not say that the subsequent computations of \(a \times b\) can be deleted
  (Expression may not be available at the subsequent points)

- Hoisting involves
  - making the expressions available and
  - deleting their subsequent computations

A Comparison of Anticipability and Hoistability

- **Anticipability**
  - \(a = 5\)
  - \(a \times b\)

- **Hoistability**
  - \(a = 5\)
  - \(a \times b\)

Characterises safety of placement but not safety of hoisting
A Comparison of Anticipability and Hoistability

Anticipability

Hoistability

Hoist an expression to the entry of a block only if it can be hoisted out of the block into all predecessor blocks.

Revised Safety Criteria of Hoisting an Expression

- Safety of hoisting to the exit of a block
  - S.1 Hoist only if it can be hoisted out of the entries of all successor blocks.

- Safety of hoisting to the entry of a block
  - S.2 Hoist only if
    - S.2.a it is upwards exposed, or
    - S.2.b it can be hoisted to its exit and is transparent in the block.

- Safety of hoisting out of the entry of a block
  - S.3 Hoist only if for each predecessor
    - S.3.a it can be hoisted to its exit, or
    - S.3.b it is available at its exit.

Desirability of Hoisting an Expression

- Desirability of hoisting to the entry of a block
  - D.1 Hoist only if it is partially available.
Final Hoisting Criteria

- Safety of hoisting to the exit of a block
  S.1 Hoist only if it can be hoisted out of the entries of all successor blocks

- Safety of hoisting to the entry of a block
  S.2 Hoist only if
    S.2.a it is upwards exposed, or
    S.2.b it can be hoisted to its exit and is transparent in the block
  S.3 Hoist only if for each predecessor
    S.3.a it can be hoisted to its exit, or
    S.3.b it is available at its exit.

- Desirability of hoisting to the entry of a block
  D.1 Hoist only if it is partially available

From Hoisting Criteria to Data Flow Equations (1)

First Level Global Data Flow Properties in PRE

- Partial Availability.
  \[
  \begin{align*}
  PavIn_n &= \begin{cases} 
  BI & n \text{ is Start block} \\
  \bigcup_{p \in \text{pred}(n)} PawOut_p & \text{otherwise}
  \end{cases} \\
  PavOut_n &= Gen_n \cup (PavIn_n - Kill_n)
  \end{align*}
  \]

- Total Availability.
  \[
  \begin{align*}
  AvIn_n &= \begin{cases} 
  BI & n \text{ is Start block} \\
  \bigcap_{p \in \text{pred}(n)} AvOut_p & \text{otherwise}
  \end{cases} \\
  AvOut_n &= Gen_n \cup (AvIn_n - Kill_n)
  \end{align*}
  \]

From Hoisting Criteria to Data Flow Equations (2)

- Safety of hoisting to the exit of a block
  S.1 Hoist only if it can be hoisted out of the entries of all successor blocks

- Safety of hoisting to the entry of a block
  S.2 Hoist only if
    S.2.a it is upwards exposed, or
    S.2.b it can be hoisted to its exit and is transparent in the block
  S.3 Hoist only if for each predecessor
    S.3.a it can be hoisted to its exit, or
    S.3.b it is available at its exit.

- Desirability of hoisting to the entry of a block
  D.1 Hoist only if it is partially available

\[
\begin{align*}
\forall s \in \text{succ}(n), \\
Out_n \subseteq In_s
\end{align*}
\]

\[
\begin{align*}
In_n &\subseteq \text{AntGen}_n \cup (Out_n - Kill_n) \\
\forall p \in \text{pred}(n), \\
In_n &\subseteq AvOut_p \cup Out_p
\end{align*}
\]

\[
\begin{align*}
In_n &\subseteq PavIn_n \\
In_n &\subseteq PavOut_n
\end{align*}
\]

Find out the largest such set

From Hoisting Criteria to Data Flow Equations (3)

- Safety of hoisting to the exit of a block
  S.1 Hoist only if it can be hoisted out of the entries of all successor blocks

- Safety of hoisting to the entry of a block
  S.2 Hoist only if
    S.2.a it is upwards exposed, or
    S.2.b it can be hoisted to its exit and is transparent in the block
  S.3 Hoist only if for each predecessor
    S.3.a it can be hoisted to its exit, or
    S.3.b it is available at its exit.

- Desirability of hoisting to the entry of a block
  D.1 Hoist only if it is partially available

\[
\begin{align*}
\forall s \in \text{succ}(n), \\
Out_n \subseteq In_s
\end{align*}
\]

\[
\begin{align*}
In_n &\subseteq \text{AntGen}_n \cup (Out_n - Kill_n) \\
\forall p \in \text{pred}(n), \\
In_n &\subseteq AvOut_p \cup Out_p
\end{align*}
\]

\[
\begin{align*}
In_n &\subseteq PavIn_n \\
In_n &\subseteq PavOut_n
\end{align*}
\]

Find out the largest such set
From Hoisting Criteria to Data Flow Equations (3)

∀s ∈ succ(n),
Out_n ⊆ In_s

In_n ⊆ AntGen_n ∪ (Out_n − Kill_n)

∀p ∈ pred(n),
In_n ⊆ AvOut_p ∪ Out_p

In_n ⊆ PavIn_n

Expressions should be partially available, and

Desirability: D.1

In_n = PavIn_n

From Hoisting Criteria to Data Flow Equations (3)

∀s ∈ succ(n),
Out_n ⊆ In_s

In_n = PavIn_n ∩ (AntGen_n ∪ (Out_n − Kill_n))

∀p ∈ pred(n),
In_n ⊆ AvOut_p ∪ Out_p

In_n ⊆ PavIn_n

For every predecessor, expressions can be hoisted to its exit, or

Safety: S.2.b

Expressions should be upwards exposed, or

In_n ⊆ PavIn_n

From Hoisting Criteria to Data Flow Equations (3)

∀s ∈ succ(n),
Out_n ⊆ In_s

In_n ⊆ AntGen_n ∪ (Out_n − Kill_n)

∀p ∈ pred(n),
In_n ⊆ AvOut_p ∪ Out_p

In_n ⊆ PavIn_n

Expressions can be hoisted to the exit and are transparent in the block

Safety: S.2.a

Expressions should be upwards exposed, or

In_n = PavIn_n ∩ (AntGen_n ∪ (Out_n − Kill_n))

∀p ∈ pred(n),
In_n ⊆ AvOut_p ∪ Out_p

In_n ⊆ PavIn_n

For every predecessor, expressions can be hoisted to its exit, or

Safety: S.3.b

From Hoisting Criteria to Data Flow Equations (3)

∀s ∈ succ(n),
Out_n ⊆ In_s

In_n = PavIn_n ∩ (AntGen_n ∪ (Out_n − Kill_n))

∀p ∈ pred(n),
In_n ⊆ AvOut_p ∪ Out_p

In_n ⊆ PavIn_n

For every predecessor, expressions can be hoisted to its exit, or

Safety: S.3.b

Expressions can be hoisted to the exit and are transparent in the block
∀s ∈ succ(n),
Out_n ⊆ In_s

\[ In_n = PavIn_n \cap \left( AntGen_n \cup (Out_n - Kill_n) \right) \]
\[ \bigcap_{p \in \text{pred}(n)} \left( Out_p \cup \text{AvOut}_p \right) \]

∀p ∈ pred(n),
In_n ⊆ AvOut_p ∪ Out_p

\[ In_n ⊆ PavIn_n \]

Safety: S.3.a

Expressions should be hoisted to the exit of a block
if they can be hoisted to the entry of all successors

Jul 2017
IIT Bombay

∀s ∈ succ(n),
Out_n ⊆ In_s

\[ In_n = PavIn_n \cap \left( AntGen_n \cup (Out_n - Kill_n) \right) \]
\[ \bigcap_{p \in \text{pred}(n)} \left( Out_p \cup \text{AvOut}_p \right) \]

∀p ∈ pred(n),
In_n ⊆ AvOut_p ∪ Out_p

\[ In_n = \text{Kill}_n \bigcup \text{Out}_p \bigcup \text{AvOut}_p \bigcup \text{PavIn}_n \]

Boundary condition
Anticipability and PRE (Hoistability) Data Flow Equations

PRE Hoistability
\[ \text{PRE Hoistability} = \text{PavIn}_n \cap (\text{AntGen}_n \cup (\text{Out}_n - \text{Kill}_n)) \]
\[ \bigcap_{p \in \text{pred}(n)} (\text{Out}_p \cup \text{AvOut}_p) \]

Anticipability
\[ \text{Anticipability} = \text{In}_n = \text{AntGen}_n \cup (\text{Out}_n - \text{Kill}_n) \]
\[ \text{Out}_n = \begin{cases} \text{BL} & \text{n is End block} \\ \bigcap_{s \in \text{succ}(n)} \text{In}_s & \text{otherwise} \end{cases} \]

\[ \text{Out}_n = \begin{cases} \text{BL} & \text{n is End block} \\ \bigcap_{s \in \text{succ}(n)} \text{In}_s & \text{otherwise} \end{cases} \]

\[ \text{Redundant}_n = \text{In}_n \cap \text{AntGen}_n \]

Deletion Criteria in PRE
- An expression is redundant in node \( n \) if
  - it can be placed at the entry (i.e. can be "hoisted" out) of \( n \), AND
  - it is upwards exposed in node \( n \).

- A hoisting path for an expression \( e \) begins at \( n \) if \( e \in \text{Redundant}_n \)
- This hoisting path extends against the control flow.

Insertion Criteria in PRE
- An expression is inserted at the exit of node \( n \) if
  - it can be placed at the exit of \( n \), AND
  - it is not available at the exit of \( n \), AND
  - it cannot be hoisted out of \( n \), OR it is modified in \( n \).

\[ \text{Insert}_n = \text{Out}_n \cap (\neg \text{AvOut}_n) \cap (\neg \text{In}_n \cup \text{Kill}_n) \]

- A hoisting path for an expression \( e \) ends at \( n \) if \( e \in \text{Insert}_n \)

Performing PRE by Computing \( \text{In}/\text{Out} \): Simple Cases (1)

- First Level Values

<table>
<thead>
<tr>
<th>Node</th>
<th>First Level Values</th>
<th>Init.</th>
<th>Iter. 1</th>
<th>Iter. 2</th>
<th>Redund.</th>
<th>Insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Performing PRE by Computing **In/Out**: Simple Cases (1)

\[ c = a \ast b \]
\[ d = a \ast b \]
\[ t = c \ast b \]
\[ a = t \]
\[ d = t \]

This is an instance of **Common Subexpression Elimination**

Performing PRE by Computing **In/Out**: Simple Cases (2)

\[ c = a \ast b \]
\[ d = a \ast b \]
\[ t = a \ast b \]
\[ a = 5 \]
\[ d = t \]

Redundancy
Insertion

Node | First Level Values | Init. | Iter. 1 | Iter. 2 | Redund. | Insert
---|---|---|---|---|---|---
| AntGen | Kill | PavIn | AvOut | Out | In | Out | In | Out | In | Redund. | Insert
1 | | | | | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0
2 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0
3 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0

Performing PRE by Computing **In/Out**: Simple Cases (3)

\[ a = b \ast c \]
\[ t = b \ast c \]
\[ a = t \]
\[ a = 5 \]
\[ a = 5 \]

Insertion
Redundancy

Node | First Level Values | Init. | Iter. 1 | Iter. 2 | Redund. | Insert
---|---|---|---|---|---|---
| AntGen | Kill | PavIn | AvOut | Out | In | Out | In | Out | In | Redund. | Insert
2 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0
3 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0

Tutorial Problems for PRE

(a) \[ a \ast b \]
(b) \[ a \ast b \]
(c) \[ a \ast b \]
(d) \[ a \ast b \]
(e) \[ a \ast b \]
**Tutorial Problems for PRE**

(a) \[ a \times b \]
(b) \[ a = 5 \]
(c) \[ a \times b \]
(d) \[ a = 5 \]
(e) \[ a \times b \]

**Further Tutorial Problem for PRE**

Let \( \{a \times b, b \times c\} \equiv \text{bit string 11} \)

<table>
<thead>
<tr>
<th>Node ( n )</th>
<th>( \text{Kill}_n )</th>
<th>( \text{AntGen}_n )</th>
<th>( \text{PavIn}_n )</th>
<th>( \text{AvOut}_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>10</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>00</td>
<td>11</td>
<td>00</td>
</tr>
<tr>
<td>4</td>
<td>00</td>
<td>00</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>00</td>
<td>01</td>
<td>11</td>
<td>01</td>
</tr>
<tr>
<td>6</td>
<td>00</td>
<td>00</td>
<td>11</td>
<td>01</td>
</tr>
</tbody>
</table>

- Compute \( \text{ln}_n / \text{Out}_n / \text{Redundant}_n / \text{Insert}_n \)
- Identify hoisting paths

**Result of PRE Data Flow Analysis of the Running Example**

<table>
<thead>
<tr>
<th>Block</th>
<th>Constant Information</th>
<th>Iteration #1</th>
<th>Changes in iteration #2</th>
<th>Changes in iteration #3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \text{PavIn}_n )</td>
<td>( \text{AvOut}_n )</td>
<td>( \text{Out}_n )</td>
<td>( \text{ln}_n )</td>
</tr>
<tr>
<td>( n_8 )</td>
<td>11111</td>
<td>00001</td>
<td>00000</td>
<td>00001</td>
</tr>
<tr>
<td>( n_7 )</td>
<td>11101</td>
<td>11000</td>
<td>00011</td>
<td>01001</td>
</tr>
<tr>
<td>( n_6 )</td>
<td>11101</td>
<td>11001</td>
<td>01001</td>
<td>01001</td>
</tr>
<tr>
<td>( n_5 )</td>
<td>11101</td>
<td>11000</td>
<td>01001</td>
<td>01001</td>
</tr>
<tr>
<td>( n_4 )</td>
<td>11100</td>
<td>10100</td>
<td>01001</td>
<td>11100</td>
</tr>
<tr>
<td>( n_3 )</td>
<td>11101</td>
<td>10000</td>
<td>01000</td>
<td>01001</td>
</tr>
<tr>
<td>( n_2 )</td>
<td>10001</td>
<td>00010</td>
<td>00011</td>
<td>00000</td>
</tr>
<tr>
<td>( n_1 )</td>
<td>00000</td>
<td>10001</td>
<td>00000</td>
<td>00000</td>
</tr>
</tbody>
</table>
Hoisting Paths for Some Expressions in the Running Example

\[ n_1 \]
\[ b = 4; \]
\[ a = b + c; \]
\[ d = a \times b; \]

\[ c = b + c; \]

\[ d = a + b; \]

\[ f(a - c); \]
\[ f(b + c); \]

\[ g(a + b); \]
\[ h(a - c); \]
\[ f(b + c); \]

Optimized Version of the Running Example

\[ n_1 \]
\[ b = 4; \]
\[ t_2 = b + c; \]
\[ a = t_2; \]
\[ t_0 = a \times b; \]
\[ d = t_0; \]

\[ c = t_2 \]
\[ t_1 = a + b; \]

\[ d = t_1; \]
\[ t_2 = b + c; \]

Jul 2017