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
Part 2

Live Variables Analysis

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

\[ v \text{ is live at } p \]
\[ v \text{ is not live at } p \]
\[ v \text{ is live at } p \]

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

Control Transfer

Local Data Flow Properties

Local Data Flow Properties
Local Data Flow Properties for Live Variables Analysis

- **r-value occurrence**
  Value is only read, e.g. x, y, z in
  \[ x \text{.} \text{sum} = y \text{.} \text{data} + z \text{.} \text{data} \]

- **l-value occurrence**
  Value is modified, e.g. y in
  \[ y = x \text{.} \text{lptr} \]

\[ \text{Gen}_n = \{ v \mid \text{variable } v \text{ used in basic block } n \text{ and is not preceded by a definition of } v \} \]

\[ \text{Kill}_n = \{ v \mid \text{basic block } n \text{ contains a definition of } v \} \]

Global Data Flow Properties

- Edge based specifications

Data Flow Equations for Our Example

1. \[ w = x \]
2. \[ \text{while} (x \text{.} \text{data} < \text{max}) \]
3. \[ y = x \text{.} \text{lptr} \]
4. \[ x = x \text{.} \text{rptr} \]
5. \[ z = \text{New class of } z \]
6. \[ y = y \text{.} \text{lptr} \]
7. \[ z \text{.} \text{sum} = x \text{.} \text{data} + y \text{.} \text{data} \]
Performing Live Variables Analysis

Gen = {x}, Kill = {w}

w = x

Gen = {x}, Kill = {y}
y = x.lptr

Gen = {x}, Kill = {y}
z = New class_of_x

Gen = {y}, Kill = {y}
y = y.lptr

Gen = {x, y, z}, Kill = {z}
z.sum = x.data + y.data

Gen = {x, y, z}, Kill = {z}
z.sum = x.data + y.data

Gen = {x, y, z}, Kill = {z}
z.sum = x.data + y.data

Gen = {x, y, z}, Kill = {z}
z.sum = x.data + y.data

Gen = {x}, Kill = {w}

x = x.rptr

x = x.rptr

x = x.rptr

x = x.rptr

x, y, z are considered to be used based purely on local use even if the value of z is not used later. A different analysis called strongly live variables analysis improves on this.

Initialization

Traversal

Iteration #1

Gen = {x}, Kill = {w}

w = x

Gen = {x}, Kill = {y}
y = x.lptr

Gen = {x}, Kill = {y}
z = New class_of_x

Gen = {y}, Kill = {y}
y = y.lptr

Gen = {x, y, z}, Kill = {z}
z.sum = x.data + y.data

Gen = {x, y, z}, Kill = {z}
z.sum = x.data + y.data

Gen = {x, y, z}, Kill = {z}
z.sum = x.data + y.data

Gen = {x, y, z}, Kill = {z}
z.sum = x.data + y.data

Gen and Kill need not be mutually exclusive

z is an r-value occurrence and not an l-value occurrence
Performing Live Variables Analysis

Local Data Flow Properties for Live Variables Analysis

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

- \( \text{Gen}_n \): Use not preceded by definition
  - Upwards exposed use
- \( \text{Kill}_n \): 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 \text{Gen}_n ) ( v \notin \text{Kill}_n )</td>
<td>( a = b + c ) ( b = c \times d )</td>
<td>liveness of ( v ) is unaffected by the basic block</td>
</tr>
<tr>
<td>2</td>
<td>( v \in \text{Gen}_n ) ( v \notin \text{Kill}_n )</td>
<td>( a = b + c ) ( b = v \times d )</td>
<td>( v ) becomes live before the basic block</td>
</tr>
<tr>
<td>3</td>
<td>( v \notin \text{Gen}_n ) ( v \in \text{Kill}_n )</td>
<td>( a = b + c ) ( v = c \times d )</td>
<td>( v ) ceases to be live before the basic block</td>
</tr>
<tr>
<td>4</td>
<td>( v \in \text{Gen}_n ) ( v \in \text{Kill}_n )</td>
<td>( a = v + c ) ( v = c \times 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 #3 of Dead Code Elimination**

<table>
<thead>
<tr>
<th>Iteration #1</th>
<th>Iteration #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out</td>
<td>In</td>
</tr>
<tr>
<td>$n_0$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$n_5$</td>
<td>${a, b, c}$</td>
</tr>
<tr>
<td>$n_4$</td>
<td>${a, b, c}$</td>
</tr>
<tr>
<td>$n_3$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$n_2$</td>
<td>${a, b, c}$</td>
</tr>
<tr>
<td>$n_1$</td>
<td>${a, b, c}$</td>
</tr>
</tbody>
</table>

**Tutorial Problem 1: Perform Dead Code Elimination**

<table>
<thead>
<tr>
<th>Iteration #1</th>
<th>Iteration #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out</td>
<td>In</td>
</tr>
<tr>
<td>$n_6$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$n_5$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$n_4$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$n_3$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$n_2$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$n_1$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
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

\[
\begin{align*}
4w &= x + y + 32 \\
4x &= y + z + 32 \\
4y &= z + w + 32 \\
4z &= w + x + 32
\end{align*}
\]

\[
\begin{align*}
\text{Solution} & \quad w = x = y = z = 16
\end{align*}
\]

- Rewrite the equations to define \(w, x, y,\) and \(z\)
  \[
  \begin{align*}
  w &= 0.25x + 0.25y + 8 \\
  x &= 0.25y + 0.25z + 8 \\
  y &= 0.25z + 0.25w + 8 \\
  z &= 0.25w + 0.25x + 8
  \end{align*}
  \]
- Assume some initial values of \(w_0, x_0, y_0,\) and \(z_0\)
- Compute \(w_i, x_i, y_i,\) and \(z_i\) within some margin of error
**A Digression: Gauss-Seidel 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**
- $w_1 = 6 + 6 + 8 = 20$
- $x_1 = 6 + 6 + 8 = 20$
- $y_1 = 6 + 6 + 8 = 20$
- $z_1 = 6 + 6 + 8 = 20$

**Iteration 2**
- $w_2 = 5 + 5 + 8 = 18$
- $x_2 = 5 + 5 + 8 = 18$
- $y_2 = 5 + 5 + 8 = 18$
- $z_2 = 5 + 5 + 8 = 18$

**Iteration 3**
- $w_3 = 4.5 + 4.5 + 8 = 17$
- $x_3 = 4.5 + 4.5 + 8 = 17$
- $y_3 = 4.5 + 4.5 + 8 = 17$
- $z_3 = 4.5 + 4.5 + 8 = 17$

**Iteration 4**
- $w_4 = 4.25 + 4.25 + 8 = 16.5$
- $x_4 = 4.25 + 4.25 + 8 = 16.5$
- $y_4 = 4.25 + 4.25 + 8 = 16.5$
- $z_4 = 4.25 + 4.25 + 8 = 16.5$

**Iterate 5**
- $w_5 = 4.125 + 4.125 + 8 = 16.25$
- $x_5 = 4.125 + 4.125 + 8 = 16.25$
- $y_5 = 4.125 + 4.125 + 8 = 16.25$
- $z_5 = 4.125 + 4.125 + 8 = 16.25$

**Our Method of Performing Data Flow Analysis**

- Round robin iteration
- Essentially Jacobi method
- Unknowns are the data flow variables $ln_i$ and $out_i$
- 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

**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**
- $w_1 = 6 + 6 + 8 = 20$
- $x_1 = 6 + 6 + 8 = 20$
- $y_1 = 6 + 6 + 8 = 20$
- $z_1 = 6 + 6 + 8 = 20$

**Iteration 2**
- $w_2 = 5 + 5 + 8 = 18$
- $x_2 = 5 + 5 + 8 = 18$
- $y_2 = 5 + 5 + 8 = 18$
- $z_2 = 5 + 5 + 8 = 18$

**Iteration 3**
- $w_3 = 4.5 + 4.5 + 8 = 17$
- $x_3 = 4.5 + 4.5 + 8 = 17$
- $y_3 = 4.5 + 4.5 + 8 = 17$
- $z_3 = 4.5 + 4.5 + 8 = 17$

**Iteration 4**
- $w_4 = 4.25 + 4.25 + 8 = 16.5$
- $x_4 = 4.25 + 4.25 + 8 = 16.5$
- $y_4 = 4.25 + 4.25 + 8 = 16.5$
- $z_4 = 4.25 + 4.25 + 8 = 16.5$

**Iteration 5**
- $w_5 = 4.125 + 4.125 + 8 = 16.25$
- $x_5 = 4.125 + 4.125 + 8 = 16.25$
- $y_5 = 4.125 + 4.125 + 8 = 16.25$
- $z_5 = 4.125 + 4.125 + 8 = 16.25$

**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;
}
```

```
 Jul 2017

```
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
  \[⇒ BI \text{ can be } ∅\]

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

Solution of Tutorial Problem 2

<table>
<thead>
<tr>
<th>Block</th>
<th>Local Information</th>
<th>Global Information</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iteration # 1</td>
<td>Iteration # 2</td>
</tr>
<tr>
<td></td>
<td>Gen_a</td>
<td>Kill_a</td>
</tr>
<tr>
<td>n_6</td>
<td>{a}</td>
<td>∅</td>
</tr>
<tr>
<td>n_5</td>
<td>{a, i}</td>
<td>{a, i}</td>
</tr>
<tr>
<td>n_4</td>
<td>{n}</td>
<td>{a}</td>
</tr>
<tr>
<td>n_3</td>
<td>{i, n}</td>
<td>∅</td>
</tr>
<tr>
<td>n_2</td>
<td>{i, k}</td>
<td>∅</td>
</tr>
<tr>
<td>n_1</td>
<td>{m}</td>
<td>{i}</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
  \[n_1 → n_2 → n_3 → n_5 → n_2 → \ldots\]

- Show an execution path along which a is not live at the exit of n_3
  \[n_1 → n_2 → n_3 → n_4 → n_2 → \ldots\]

Tutorial Problem 3 for Liveness Analysis

Also write a C program for this CFG without using goto or break

```c
void f()
{
    int x, y, z;
    int c, d;
    x = 1;
    y = 2;
    if (c)
    {
        do
        {
            x = y+1;
            y = 2*z;
            if (d)
            {
                x = y+z;
                z = 1;
            }
        } while (c < 20);
    }
    return a;
}
```
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&lt;sub&gt;n&lt;/sub&gt;</td>
<td>KIll&lt;sub&gt;n&lt;/sub&gt;</td>
</tr>
<tr>
<td>n&lt;sub&gt;0&lt;/sub&gt;</td>
<td>{x}</td>
<td>{z}</td>
</tr>
<tr>
<td>n&lt;sub&gt;1&lt;/sub&gt;</td>
<td>{y, z, d}</td>
<td>{x, y}</td>
</tr>
<tr>
<td>n&lt;sub&gt;2&lt;/sub&gt;</td>
<td>{c}</td>
<td>∅</td>
</tr>
<tr>
<td>n&lt;sub&gt;3&lt;/sub&gt;</td>
<td>{y, z, d}</td>
<td>{x, y}</td>
</tr>
<tr>
<td>n&lt;sub&gt;4&lt;/sub&gt;</td>
<td>{c}</td>
<td>{x}</td>
</tr>
<tr>
<td>n&lt;sub&gt;5&lt;/sub&gt;</td>
<td>{c}</td>
<td>{z}</td>
</tr>
</tbody>
</table>

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

Interpreting the Result of Liveness Analysis for Tutorial Problem 3

- Why is z live at the exit of n<sub>5</sub>?
- Why is z not live at the entry of n<sub>5</sub>?
- Why is x live at the exit of n<sub>3</sub> inspite of being killed in n<sub>4</sub>?
- Identify the instance of dead code elimination z = x in n<sub>5</sub>
- 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

How Does the Initialization Affect the Solution?

<table>
<thead>
<tr>
<th>Init. Iter. #1</th>
<th>Init. Iter. #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b = 5</td>
<td>a = b = 5</td>
</tr>
<tr>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>{a, b}</td>
<td>∅</td>
</tr>
</tbody>
</table>

a is spuriously marked live
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?
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:
  \[ \text{All branch outcomes are possible} \]
  \[ \Rightarrow \text{Consider every path in CFG as a potential execution path} \]
- 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 makes conservative assumptions
  \[ \Rightarrow \text{our analysis is path insensitive} \]
  Note: It is \textit{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 $e$.

Local Data Flow Properties for Available Expressions Analysis

$$Gen_n = \{ e \mid \text{expression } e \text{ is evaluated in basic block } n \text{ and this evaluation is not followed by a definition of any operand of } e \}$$

$$Kill_n = \{ e \mid \text{basic block } n \text{ contains a definition of an operand of } e \}$$

Entity | Manipulation | Exposition
--- | --- | ---
$Gen_n$ | Expression | Use | Downwards
$Kill_n$ | Expression | Modification | Anywhere

Data Flow Equations For Available Expressions Analysis

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

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

Alternatively,

$$Out_n = f_n(In_n), \quad \text{where}$$

$$f_n(X) = Gen_n \cup (X - Kill_n)$$

- $In_n$ and $Out_n$ are sets of expressions
- $BL$ 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 $n$ ($In_n$) and a computation of the expression exists in $n$ such that it is not preceded by a definition of any of its operands ($AntGen_n$)
  - Then the expression is redundant

$$Redundant_n = In_n \cap AntGen_n$$

- A redundant expression is upwards exposed whereas the expressions in $Gen_n$ are downwards exposed
An Example of Available Expressions Analysis

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 #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>{e_1, e_2}</td>
<td>1110</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>2</td>
<td>{e_2}</td>
<td>0010</td>
<td>0000</td>
<td>{e_1}</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_2, e_4}</td>
<td>0011</td>
</tr>
<tr>
<td>5</td>
<td>{e_1, e_4}</td>
<td>1001</td>
<td>0000</td>
<td>{e_2}</td>
</tr>
<tr>
<td>6</td>
<td>{e_2}</td>
<td>0001</td>
<td>0000</td>
<td>{e_1, e_2}</td>
</tr>
</tbody>
</table>

An Example of Available Expressions Analysis

**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>{e_1, e_2}</td>
<td>1110</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>2</td>
<td>{e_2}</td>
<td>0010</td>
<td>0000</td>
<td>{e_1}</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_2, e_4}</td>
<td>0011</td>
</tr>
<tr>
<td>5</td>
<td>{e_1, e_4}</td>
<td>1001</td>
<td>0000</td>
<td>{e_2}</td>
</tr>
<tr>
<td>6</td>
<td>{e_2}</td>
<td>0001</td>
<td>0000</td>
<td>{e_1, e_2}</td>
</tr>
</tbody>
</table>
An Example of Available Expressions Analysis

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

<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>${e_2}$</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>

Final Result

```
0000
1100
1000
0010
1010
1001
0001
```

Tutorial Problem 2 for Available Expressions Analysis

```
d = a \times b
e = b + c

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

Solution of the Tutorial Problem 2

```
Bit vector $a \times b \mid b + c$

<table>
<thead>
<tr>
<th>Node</th>
<th>Local Information</th>
<th>Global Information</th>
<th>Iteration # 1</th>
<th>Changes in iteration # 2</th>
<th>Redundant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gen$_n$</td>
<td>Kill$_n$</td>
<td>AntGen$_n$</td>
<td>In$_n$</td>
<td>Out$_n$</td>
<td>In$_n$</td>
</tr>
<tr>
<td>$n_1$</td>
<td>11</td>
<td>00</td>
<td>11</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>$n_2$</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>$n_3$</td>
<td>01</td>
<td>10</td>
<td>01</td>
<td>11</td>
<td>01</td>
</tr>
<tr>
<td>$n_4$</td>
<td>00</td>
<td>11</td>
<td>10</td>
<td>11</td>
<td>00</td>
</tr>
<tr>
<td>$n_5$</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>$n_6$</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
```

Tutorial Problem 3 for Available Expressions Analysis

```
c = a \times b
d = b + c

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

Bit vector $a \oplus b \oplus b \oplus c \oplus a \oplus b$

<table>
<thead>
<tr>
<th>Node</th>
<th>Local Information</th>
<th>Global Information</th>
<th>Iteration # 1</th>
<th>Iteration # 2</th>
<th>Iteration # 3</th>
<th>Redundant_n</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_1$</td>
<td>110</td>
<td>Kill_n</td>
<td>AntGen_n</td>
<td>In_n</td>
<td>Out_n</td>
<td>In_n</td>
</tr>
<tr>
<td>$n_2$</td>
<td>001</td>
<td>000</td>
<td>001</td>
<td>110</td>
<td>111</td>
<td>100</td>
</tr>
<tr>
<td>$n_3$</td>
<td>010</td>
<td>000</td>
<td>010</td>
<td>111</td>
<td>111</td>
<td>001</td>
</tr>
<tr>
<td>$n_4$</td>
<td>001</td>
<td>101</td>
<td>000</td>
<td>111</td>
<td>011</td>
<td>011</td>
</tr>
<tr>
<td>$n_5$</td>
<td>000</td>
<td>010</td>
<td>000</td>
<td>111</td>
<td>111</td>
<td>001</td>
</tr>
<tr>
<td>$n_6$</td>
<td>001</td>
<td>000</td>
<td>001</td>
<td>101</td>
<td>101</td>
<td>001</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 makes $a \ast c$ available in node 3 although its computation in node 3 is not redundant.

Bit Vector

\[
\begin{array}{c|c|c}
\text{Node} & \text{Initialization } U & \text{Initialization } \emptyset \\
\hline
1 & 00 & 10 \\
2 & 10 & 11 \\
3 & 10 & 11 \\
\emptyset & 11 & 11 \\
\hline
\end{array}
\]

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

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

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.

Solution of the Tutorial Problem 2 for Partial Availability Analysis

Bit vector \( a \times b \text{ or } b + c \)

<table>
<thead>
<tr>
<th>Node</th>
<th>Local Information</th>
<th>Global Information</th>
<th>Iteration # 1</th>
<th>ParRedund ( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n_1 )</td>
<td>11 00 11</td>
<td>00 11 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( n_2 )</td>
<td>00 00 00</td>
<td>11 11 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( n_3 )</td>
<td>01 10 01</td>
<td>11 01 01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( n_4 )</td>
<td>00 11 10</td>
<td>11 00 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( n_5 )</td>
<td>00 00 00</td>
<td>01 01 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( n_6 )</td>
<td>00 00 00</td>
<td>01 01 00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Solution of the Tutorial Problem 3 for Partial Availability Analysis

Bit vector \( a \times b \text{ or } b + c \text{ or } a + b \)

<table>
<thead>
<tr>
<th>Node</th>
<th>Local Information</th>
<th>Global Information</th>
<th>Iteration # 1</th>
<th>Changes in # 2</th>
<th>ParRedund ( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n_1 )</td>
<td>110 010 100</td>
<td>000 110 110</td>
<td>010 111 111</td>
<td>000 110 110</td>
<td>000</td>
</tr>
<tr>
<td>( n_2 )</td>
<td>001 000 001</td>
<td>110 111 111</td>
<td>010 111 111</td>
<td>001 111 111</td>
<td>001</td>
</tr>
<tr>
<td>( n_3 )</td>
<td>010 000 010</td>
<td>111 111 111</td>
<td>010 111 111</td>
<td>000 111 111</td>
<td>000</td>
</tr>
<tr>
<td>( n_4 )</td>
<td>001 101 000</td>
<td>111 011 011</td>
<td>000 011 011</td>
<td>000 011 011</td>
<td>000</td>
</tr>
<tr>
<td>( n_5 )</td>
<td>000 010 000</td>
<td>111 101 101</td>
<td>000 101 101</td>
<td>000 101 101</td>
<td>000</td>
</tr>
<tr>
<td>( n_6 )</td>
<td>001 000 001</td>
<td>101 101 101</td>
<td>001 101 101</td>
<td>001 101 101</td>
<td>001</td>
</tr>
</tbody>
</table>

Part 5

Reaching Definitions 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 \( p \) (\( x \) is a variable and \( e \) 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

- \( a_1: a = 4 \)
- \( b_1: b = 2 \)
- \( c_1: c = 3 \)
- \( n_1: n = c+2 \)

- \( 2: \) if (\( a > n \))
  - \( 3: a_2: a = a+1 \)

- \( 4: \) if (\( a \geq 12 \))
  - \( 5: t_l_1: t_1 = a+b \)
  - \( a_3: a = t_1+c \)

- \( 6: \) print \( a \)

Use-Def Chains

- \( a_1: a = 4 \)
- \( b_1: b = 2 \)
- \( c_1: c = 3 \)
- \( n_1: n = c+2 \)

- \( 2: \) if (\( a > n \))
  - \( 3: a_2: a = a+1 \)

- \( 4: \) if (\( a \geq 12 \))
  - \( 5: t_l_1: t_1 = a+b \)
  - \( a_3: a = t_1+c \)

- \( 6: \) print \( a \)
Defining Data Flow Analysis for Reaching Definitions Analysis

Let \( d_v \) be a definition of variable \( v \)

\[
\text{Gen}_n = \{ \ d_v \mid \text{variable v is defined in basic block n and this definition is not followed (within n) by a definition of v} \}\]

\[
\text{Kill}_n = \{ \ d_v \mid \text{basic block n contains a definition of v} \}\]

<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>Anywhere</td>
</tr>
</tbody>
</table>

Data Flow Equations for Reaching Definitions Analysis

\[
\text{In}_n = \begin{cases} \text{BI} & n \text{ is Start block} \\ \bigcup_{p \in \text{pred}(n)} \text{Out}_p & \text{otherwise} \end{cases}
\]

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

\[
\text{BI} = \{ d_x : x = \text{undef} \mid x \in \text{Var} \}
\]

\( \text{In}_n \) and \( \text{Out}_n \) are sets of definitions.
Temporary variable t1 is ignored

Local copy propagation and constant folding

- Temporary variable t1 is ignored
- For variable v, v0 denotes the definition v = ?
  This is used for defining BI
### Tutorial Problem for Copy Propagation

```
1 a1: a = 4
   b1: b = 2
c1: c = 3
n1: n = 6

2 if (a > n1)
   F \{a1, a2, b1, c1, n1\}
        \{a1, a2, b1, c1, n1\}
3 a2: a = a+1
   T \{a1, a2, b1, c1, n1\}
4 if (a ≥ 12)
   F \{t1, a1, a2, b1, c1, n1\}
        T \{t1, a1, a2, b1, c1, n1\}
        \{t1, a1, a2, b1, c1, n1\}
5 t1: t1 = a + b2
   a3: a = t1 + c3
   T \{t1, a1, a2, b1, c1, n1\}
6 print a
        \{a1, a2, a3, b1, c1, n1\}
```

- RHS of \( n_1 \) is constant and hence cannot change
- In block 2, \( n \) can be replaced by 6
- RHS of \( b_1 \) and \( c_1 \) are constant and hence cannot change
- In block 5, \( b \) can be replaced by 2 and \( c \) can be replaced by 3

---

### 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

1 if (b == 0)
   True
   2 c = a/b
   False
   3

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

\[Gen_n = \{e \mid \text{expression } e \text{ is evaluated in basic block } n \text{ and this evaluation is not preceded (within } n) \text{ by a definition of any operand of } e\}\]

\[Kill_n = \{e \mid \text{basic block } n \text{ contains a definition of an operand of } e\}\]

<table>
<thead>
<tr>
<th>Entity</th>
<th>Manipulation</th>
<th>Exposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Gen_n)</td>
<td>Expression</td>
<td>Use</td>
</tr>
<tr>
<td>(Kill_n)</td>
<td>Expression</td>
<td>Modification</td>
</tr>
</tbody>
</table>

Data Flow Equations for Anticipable Expressions Analysis

\[In_n = Gen_n \cup (Out_n - Kill_n)\]

\[Out_n = \begin{cases} \bigcap_{s \in \text{succ}(n)} In_s & \text{if } BI_n \text{ is End block} \\ \text{otherwise} & \end{cases}\]

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

Tutorial Problem 1 for Anticipable Expressions Analysis

\(a = 5;\)  
\(b = 10;\)

\(e = b - c;\)  
\(c = 6;\)

\(d = b + c;\)  
\(a = 10;\)

\(d = b + c;\)  
\(c = 2;\)

\(d = b - c;\)

\(d = b + c;\)  
\(e = a + b;\)

\(Expr = \{a + b, b + c, b - c\}\)
### 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_a</td>
<td>Kill_a</td>
<td>Out_a</td>
<td>In_a</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>

### Tutorial Problem 2 for Anticipable Expressions Analysis

```
Expr = { a + b, c + d }
```

```
da = a * b;
if (d)
    c = a + b;
nde = c + d;
a = 5;
print a * b;
```

### 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_a</td>
<td>Kill_a</td>
<td>Out_a</td>
<td>In_a</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>
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 (a)</td>
<td>Variable</td>
<td>Use</td>
</tr>
<tr>
<td>Kill (a)</td>
<td>Variable</td>
<td>Modification</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 (a)</td>
<td>Expression</td>
<td>Use</td>
</tr>
<tr>
<td>Kill (a)</td>
<td>Expression</td>
<td>Modification</td>
</tr>
</tbody>
</table>

Common Form of Data Flow Equations

So far we have seen sets (or bit vectors). Could be entities other than sets.

\[ X_i = f(Y_i) \]
\[ Y_i = \bigcap X_j \]

Flow Function

Confluence

So far we have seen \( \cup \) and \( \cap \). Could be other operations.

A Taxonomy of Bit Vector Data Flow Frameworks

<table>
<thead>
<tr>
<th>Path Type</th>
<th>Confluence</th>
<th>Intersection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>Reached 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>Partial Redundancy Elimination (Original M-R Formulation)</td>
<td></td>
</tr>
</tbody>
</table>

Data Flow Paths Discovered by Data Flow Analysis

- Liveness
- Anticipability
- Availability
- Partial Availability
Data Flow Paths Discovered by Data Flow Analysis

**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 \(v\), and
- no other block on the path contains an assignment to \(v\).

**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:
- \(n_k\) 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 \(a \ast b\).

**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 \(b\), and
- every path ending at \(n_k\) is an availability path of \(a \ast b\).

**Partial 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