Topic-4a

Dataflow Analysis
Global Dataflow Analysis

Motivation

We need to know variable `def` and `use` information between basic blocks for:

- constant folding
- dead-code elimination
- redundant computation elimination
- code motion
- induction variable elimination
- build data dependence graph (DDG)
- etc.
Topics of DataFlow Analysis

- Reaching definition
- Live variable analysis
- ud-chains and du-chains
- Available expressions
- Others ..
1. Definition & Use

\[ S : \quad V_1 = \ldots V_2 \]

- \( S \) is a “definition” of \( V_1 \)
- \( S \) is a “use” of \( V_2 \)
Compute Def and Use Information of a Program P?

- Case 1: P is a basic block?
- Case 2: P contains more than one basic blocks?
Points and Paths

**Points in a basic block:**
- between statements
- before the first statement
- after the last statement

In the example, how many points basic block B1, B2, B3, and B5 have?

B1 has four, B2, B3, and B5 have two points each.
A path is a sequence of points $p_1, p_2, \ldots, p_n$ such that either:
(i) if $p_i$ immediately precedes $S$, then $p_{i+1}$ immediately follows $S$.
(ii) or $p_i$ is the end of a basic block and $p_{i+1}$ is the beginning of a successor block.

In the example, is there a path from the beginning of block B5 to the beginning of block B6?

Yes, it travels through the end point of B5 and then through all the points in B2, B3, and B4.
Reach and Kill

Kill

a definition \( d_1 \) of a variable \( v \) is killed between \( p_1 \) and \( p_2 \) if *in every path* from \( p_1 \) to \( p_2 \) there is another definition of \( v \).

Reach

a definition \( d \) reaches a point \( p \) if \( \exists \) a path \( d \rightarrow p \), and \( d \) is not killed along the path.

In the example, do \( d_1 \) and \( d_2 \) reach the points ?

Both \( d_1, d_2 \) reach point but only \( d_1 \) reaches point.
Reach Example

The set of defs reaching the use of $N$ in S8:  
\{S2, S8\}

def S2 reach S11 along statement path:  
(S2, S3, S4, S5, S6, S7, S11)

S8 reach S11 along statement path:  
(S8, S9, S10, S5, S6, S7, S11)
Can $d_1$ reach point $p_1$?

$d_1$  $x := \text{exp1}$
$s_1$  if $p > 0$
$s_2$  $x := x + 1$
$s_3$  $a = b + c$
$s_4$  $e = x + 1$

It depends on what point $p_1$ represents!!!
Problem Formulation: Example 2

Can $d_1$ and $d_4$ reach point $p_3$?

$d_1$ $x := \text{exp1}$
$s_2$ while $y > 0$ do
$s_3$ $a := b + 2$
$d_4$ $x := \text{exp2}$
$s_5$ $c := a + 1$
end while

$p_3$
Structured programs have an useful property: there is a single point of entrance and a single exit point for each statement.

We will consider program statements that can be described by the following syntax:

```
Statement  →  id := Expression
             |  Statement ; Statement
             |  if Expression then Statement else Statement
             |  do Statement while Expression

Expression →  id + id
            |  id
```
Structured Programs

This restricted syntax results in the forms depicted below for flowgraphs:

\[
S ::= \text{id} ::= E \\
| S ; S \\
| \text{if } E \text{ then } S \text{ else } S \\
| \text{do } S \text{ while } E
\]

\[
E ::= \text{id} + \text{id} \\
| \text{id}
\]
1. Each program point associates with a data-flow value.
2. Data-flow value represents the possible program states that can be observed for that program point.
3. The data-flow value depends on the goal of the analysis.

Given a statement $S$, $\text{in}(S)$ and $\text{out}(S)$ denote the data-flow values before and after $S$, respectively.
Assume basic block $B$ consists of statement $s_1, s_2, \ldots, s_n$ ($s_1$ is the first statement of $B$ and $s_n$ is the last statement of $B$), the data-flow values immediately before and after $B$ is denoted as:

$$\text{in}(B) = \text{in}(s_1)$$
$$\text{out}(B) = \text{out}(s_n)$$
Instances of Data-Flow Problems

- Reaching Definitions
- Live-Variable Analysis
- DU Chains and UD Chains
- Available Expressions

To solve these problems we must take into consideration the data-flow and the control flow in the program. A common method to solve such problems is to create a set of data-flow equations.
Iterative Method for Dataflow Analysis

- Establish a set of dataflow relations for each basic block
- Establish a set dataflow equations between basic blocks
- Establish an initial solution
- Iteratively solve the dataflow equations, until a \textit{fixed point} is reached.
Generate set: \textit{gen}(S)

In general, \(d\) in \(\text{gen}(S)\) if \(d\) reaches the end of \(S\) independent of whether it reaches the beginning of \(S\).

We restrict \(\text{gen}(S)\) contains only the definition in \(S\).

If \(S\) is a basic block, \(\text{gen}(S)\) contains all the definitions inside the basic block that are “visible” immediately after the block.
Kill Set: kill(S)

\( d \) in \( \text{kill}(S) \) \( \Rightarrow d \) never reaches the end of \( S \).
This is equivalent to say:
\( d \) reaches end of \( S \) \( \Rightarrow d \) is not in \( \text{kill}(S) \)

\[
\begin{align*}
    \text{kill}(S) &= D_a - \{ dd \} \\
    \text{d1} &\ a := \\
    \text{d2} &\ a := \\
    \quad &\vdots \\
    \text{dk} &\ a := \\
    \text{dd} &\ a := \\
\end{align*}
\]

Of course the statements \( d_1, d_2, \ldots, d_k \) all get killed except \( dd \) itself.

A basic block’s kill set is simply the union of all the definitions killed by its individual statements!
Reaching Definitions

Problem Statement:

Determine the set of definitions reaching a point in a program.
Iterative Algorithm for Reaching Definitions

dataflow equations

The set of definitions reaching the entry of basic block $B$:

$$\text{in}[B] = \bigcup_{P \in \text{predecessor of } B} \text{out}[P]$$

The set of definitions reaching the exit of basic block $B$:

$$\text{out}[B] = \text{gen}[B] \cup \{\text{in}[B] - \text{kill}[B]\}$$

(AhoSethiUllman, pp. 606)
Iterative Algorithm for Reaching Definitions

**Algorithm**

1) \( \text{out}(\text{ENTRY}) = \emptyset \);  
2) for (each basic block \( B \) other than \( \text{ENTRY} \)) \( \text{out}(B) = \emptyset; \)  
3) while ( changes to any \( \text{out} \) occur)  
4) for ( each \( B \) other than \( \text{ENTRY} \)) \[
\{ \text{in}[B] = \bigcup_{P \in \text{predecessor of } B} \text{out}[P]; \\
\text{out}[B] = \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B]); \\
\}
\]

Need a flag to test if a \( \text{out} \) is changed! The initial value of the flag is true.

(AhoSethiUllman, pp. 607)
Dataflow Equations – a simple case

\[ d : a := b + c \]

\[ \text{gen}[S] = \{ d \} \]
\[ \text{kill } [S] = \text{Da} - \{ d \} \]

\[ \text{gen } [S] = \text{gen } [S_2] \cup (\text{gen } [S_1] - \text{kill } [S_2]) \]
\[ \text{kill } [S] = \text{kill } [S_2] \cup (\text{kill } [S_1] - \text{gen } [S_2]) \]

\[ \text{gen } [S] = \text{gen } [S_1] \cup \text{gen } [S_2] \]
\[ \text{kill } [S] = \text{kill } [S_1] \cap \text{kill } [S_2] \]

\[ \text{gen } [S] = \text{gen } [S_1] \]
\[ \text{kill } [S] = \text{kill } [S_1] \]

Da is the set of all definitions in the program for variable a!
Dataflow Equations

\[ d : a := b + c \]

\[ \text{out} [S] = \text{gen} [S] \cup (\text{in} [S] - \text{kill} [S]) \]

\[ \text{in} [S] = \text{in} [S_1] \]
\[ \text{in} [S_2] = \text{out} [S_1] \]
\[ \text{out} [S] = \text{out} [S_2] \]

\[ \text{in} [S] = \text{in} [S_1] = \text{in} [S_2] \]
\[ \text{out} [S] = \text{out} [S_1] \cup \text{out} [S_2] \]

\[ \text{in} [S] = \text{in} [S_1] \cup \text{out} [S_1] \]
\[ \text{out} [S] = \text{out} [S_1] \]
Dataflow Analysis: An Example

Using RD (reaching def) as an example:

Question:
What is the set of reaching definitions at the exit of the loop L?

\[ \text{in (L)} = \{d_1\} \cup \text{out(L)} \]
\[ \text{gen (L)} = \{d_2\} \]
\[ \text{kill (L)} = \{d_1\} \]
\[ \text{out [L]} = \text{gen [L]} \cup \{\text{in [L]} - \text{kill[L]}\} \]

\[ \text{in[L]} \text{ depends on out[L], and out[L]} \text{ depends on in[L]}! \]
Solution?

First iteration

\[ \text{out}(L) = \text{gen}(L) \cup (\text{in}(L) - \text{kill}(L)) \]
\[ = \{d_2\} \cup (\{d_1\} - \{d_1\}) \]
\[ = \{d_2\} \]

Second iteration

\[ \text{out}(L) = \text{gen}(L) \cup (\text{in}(L) - \text{kill}(L)) \]

but now:
\[ \text{in}(L) = \{d_1\} \cup \text{out}(L) = \{d_1\} \cup \{d_2\} \]
\[ = \{d_1, d_2\} \]

therefore:
\[ \text{out}(L) = \{d_2\} \cup (\{d_1, d_2\} - \{d_1\}) \]
\[ = \{d_2\} \cup \{d_2\} \]
\[ = \{d_2\} \]

So, we reached the fixed point!
Reaching Definitions: Another Example

Step 1: Compute gen and kill for each basic block

- gen[B1] = \{d_1, d_2, d_3\}
  kill[B1] = \{d_4, d_5, d_6, d_7\}
- gen[B2] = \{d_4, d_5\}
  kill[B2] = \{d_1, d_2, d_7\}
- gen[B3] = \{d_6\}
  kill[B3] = \{d_3\}
- gen[B4] = \{d_7\}
  kill[B4] = \{d_1, d_4\}
Step 2: For every basic block, make:
out[B] = ∅

Initialization:
out[B1] = ∅
out[B2] = ∅
out[B3] = ∅
out[B4] = ∅
Reaching Definitions: Another Example (Con’t)

To simplify the representation, the in[B] and out[B] sets are represented by bit strings. Assuming the representation $d_1d_2d_3d_4d_5d_6d_7$ we obtain:

**Initialization:**

- out[B1] = ∅
- out[B2] = ∅
- out[B3] = ∅
- out[B4] = ∅

<table>
<thead>
<tr>
<th>Block</th>
<th>Initial in[B]</th>
<th>Initial out[B]</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_1</td>
<td>000 0000</td>
<td></td>
</tr>
<tr>
<td>B_2</td>
<td>000 0000</td>
<td></td>
</tr>
<tr>
<td>B_3</td>
<td>000 0000</td>
<td></td>
</tr>
<tr>
<td>B_4</td>
<td>000 0000</td>
<td></td>
</tr>
</tbody>
</table>

Notation: $d_1d_2d_3d_4d_5d_6d_7$

**Pseudocode:**

1. **ENTRY**
   - B1: $d_1$: i := m-1
   - $d_2$: j := n
   - $d_3$: a := u1

2. **B2**
   - $d_4$: i := i+1
   - $d_5$: j := j - 1

3. **B3**
   - $d_6$: a := u2

4. **B4**
   - $d_7$: i := u3

**EXIT**
Teaching Definitions: Further Example (Con’t)

while a fixed point is not found:
in[B] = ∪ out[P] where P is a predecessor of B
out[B] = gen[B] ∪ (in[B]-kill[B])

<table>
<thead>
<tr>
<th>Block</th>
<th>Initial</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>in[B]</td>
</tr>
<tr>
<td>B1</td>
<td>000 0000</td>
</tr>
<tr>
<td>B2</td>
<td>000 0000</td>
</tr>
<tr>
<td>B3</td>
<td>000 0000</td>
</tr>
<tr>
<td>B4</td>
<td>000 0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block</th>
<th>First Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>in[B]</td>
</tr>
<tr>
<td>B1</td>
<td>000 0000</td>
</tr>
<tr>
<td>B2</td>
<td>000 0000</td>
</tr>
<tr>
<td>B3</td>
<td>000 0000</td>
</tr>
<tr>
<td>B4</td>
<td>000 0000</td>
</tr>
</tbody>
</table>

out(B) = gen(B)

ENTRY

B1
\[ d_1: i := m-1 \]
\[ d_2: j := n \]
\[ d_3: a := u1 \]

B2
\[ d_4: i := i+1 \]
\[ d_5: j := j - 1 \]

B3
\[ d_6: a := u2 \]

B4
\[ d_7: i := u3 \]

EXIT

Notation: \( d_1d_2d_3d_4d_5d_6d_7 \)
Teaching Definitions: Further Example (Con’t)

```
gen[B1] = \{d_1, d_2, d_3\}
kill[B1] = \{d_4, d_5, d_6, d_7\}
gen[B2] = \{d_4, d_5\}
kill [B2] = \{d_1, d_2, d_7\}
gen[B3] = \{d_6\}
kill [B3] = \{d_3\}
gen[B4] = \{d_7\}
kil[l[B4] = \{d_1, d_4\}
```

**ENTRY**

```
B1
\quad d_1: i := m-1
\quad d_2: j := n
\quad d_3: a := u1
```

```
B2
\quad d_4: i := i+1
\quad d_5: j := j - 1
```

```
B3
\quad d_6: a := u2
```

```
B4
\quad d_7: i := u3
```

**EXIT**

```
Notation: d_1d_2d_3d_4d_5d_6d_7
```

while a fixed point is not found:

\[ \text{in}[B] = \cup \text{out}[P] \quad \text{where } P \text{ is a predecessor of } B \]

\[ \text{out}[B] = \text{gen}[B] \cup (\text{in}[B]-\text{kill}[B]) \]

<table>
<thead>
<tr>
<th>Block</th>
<th>First Iteration</th>
<th>out[B]</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_1</td>
<td>000 0000</td>
<td>111 0000</td>
</tr>
<tr>
<td>B_2</td>
<td>000 0000</td>
<td>000 1100</td>
</tr>
<tr>
<td>B_3</td>
<td>000 0000</td>
<td>000 0010</td>
</tr>
<tr>
<td>B_4</td>
<td>000 0000</td>
<td>000 0001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block</th>
<th>Second Iteration</th>
<th>out[B]</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_1</td>
<td>000 0000</td>
<td>111 0000</td>
</tr>
<tr>
<td>B_2</td>
<td>111 0010</td>
<td>001 1110</td>
</tr>
<tr>
<td>B_3</td>
<td>000 1100</td>
<td>000 1110</td>
</tr>
<tr>
<td>B_4</td>
<td>000 1100</td>
<td>001 0111</td>
</tr>
</tbody>
</table>

2008/3/11
\course\cpeg421-08s\Topic4-a.ppt 31
Teaching Definitions: Other Example (Con’t)

ENTRY

B1

\[ d_1: i := m - 1 \]
\[ d_2: j := n \]
\[ d_3: a := u1 \]

B2

\[ d_4: i := i + 1 \]
\[ d_5: j := j - 1 \]

B3

\[ d_6: a := u2 \]

B4

\[ d_7: i := u3 \]

EXIT

Notation: \( d_1d_2d_3d_4d_5d_6d_7 \)

while a fixed point is not found:
\[ \text{in}[B] = \bigcup \text{out}[P] \text{ where } P \text{ is a} \]
\[ \text{predecessor of } B \]
\[ \text{out}[B] = \text{gen}[B] \cup (\text{in}[B] \setminus \text{kill}[B]) \]

<table>
<thead>
<tr>
<th>Block</th>
<th>Second Iteration</th>
<th>Third Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\text{in}[B]</td>
<td>\text{out}[B]</td>
</tr>
<tr>
<td>( B_1 )</td>
<td>000 0000</td>
<td>111 0000</td>
</tr>
<tr>
<td>( B_2 )</td>
<td>111 0010</td>
<td>001 1110</td>
</tr>
<tr>
<td>( B_3 )</td>
<td>000 1100</td>
<td>000 1110</td>
</tr>
<tr>
<td>( B_4 )</td>
<td>000 1100</td>
<td>001 0111</td>
</tr>
</tbody>
</table>

we reached the fixed point!
while a fixed point is not found:

\[ \text{in}[B] = \bigcup \text{out}[P] \quad \text{where } P \text{ is a predecessor of } B \]

\[ \text{out}[B] = \text{gen}[B] \cup (\text{in}[B] \setminus \text{kill}[B]) \]

<table>
<thead>
<tr>
<th>Block</th>
<th>Third Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>in[B]</td>
</tr>
<tr>
<td>B₁</td>
<td>000 0000</td>
</tr>
<tr>
<td>B₂</td>
<td>111 0010</td>
</tr>
<tr>
<td>B₃</td>
<td>000 1100</td>
</tr>
<tr>
<td>B₄</td>
<td>000 1100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block</th>
<th>Forth Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>in[B]</td>
</tr>
<tr>
<td>B₁</td>
<td>000 0000</td>
</tr>
<tr>
<td>B₂</td>
<td>111 1110</td>
</tr>
<tr>
<td>B₃</td>
<td>001 1110</td>
</tr>
<tr>
<td>B₄</td>
<td>001 1110</td>
</tr>
</tbody>
</table>

we reached the fixed point!

Notation: \( d_1 d_2 d_3 d_4 d_5 d_6 d_7 \)
Other Applications of Data flow Analysis

- Live Variable Analysis
- DU and UD Chains
- Available Expressions
- Constant Propagation
- Constant Folding
- Others ..
Live Variable Analysis: Another Example of Flow Analysis

- A variable $V$ is *live* at the exit of a basic block $n$, if there is a *def-free* path from $n$ to an outward exposed use of $V$ in a node $n'$.

“live variable analysis problem” - determine the set of variables which are live at the exit from each program point.

Live variable analysis is a "backwards must" analysis, that is the analysis is done in a backwards order.
Live Variable Analysis: Another Example of Flow Analysis

\[ L1: \; b := 3; \]
\[ L2: \; c := 5; \]
\[ L3: \; a := b + c; \]
\[ \textit{goto L1}; \]

The set of live variables at line L2 is \( \{b, c\} \), but the set of live variables at line L1 is only \( \{b\} \) since variable "c" is updated in line 2. The value of variable "a" is never used, so the variable is never live.

Copy from Wikipedia, the free encyclopedia
Live Variable Analysis: Def and use set

- **def[B]**: the set of variables defined in basic block B prior to any use of that variable in B
- **use[B]**: the set of variables whose values may be used in B prior to any definition of the variable.
Live Variable Analysis

dataflow equations

The set of variables live at the entry of basic block $B$:

$$\text{in}[B] = \text{use}[B] \cup \{\text{out}[B] - \text{def}[B] \}$$

The set of variables live at the exit of basic block $B$:

$$\text{out}[B] = \bigcup_{S \in \text{successor of } B} \text{in}[S]$$
Iterative Algorithm for Live Variable Analysis

**Algorithm**

1) $\text{out(EXIT)} = \emptyset$;
2) for (each basic block B other than EXIT)
   
   $\text{in(B)} = \emptyset$;
3) while (changes to any in occur)
4) for (each B other than EXIT)
   
   \[
   \begin{align*}
   \text{out}[B] &= \bigcup_{S \in \text{successor of } B} \text{in}[S]; \\
   \text{in}[B] &= \text{use}[B] \cup (\text{out}[B] - \text{def}[B]);
   \end{align*}
   \]

Need a flag to test if a in is changed! The initial value of the flag is true.
Live Variable Analysis: a Quiz

Calculate the live variable sets in(B) and out(B) for the program:

B1
- d1: i := m-1
- d2: j := n
- d3: a := u1

B2
- d4: i := i+1
- d5: j := j - 1

B3
- d6: a := u2

B4
- d7: i := u3

2008/3/11
D-U and U-D Chains

Many dataflow analyses need to find the use-sites of each defined variable or the definition-sites of each variable used in an expression.

Def-Use (D-U), and Use-Def (U-D) chains are efficient data structures that keep this information.

Notice that when a code is represented in Static Single-Assignment (SSA) form (as in most modern compilers) there is no need to maintain D-U and U-D chains.
An UD chain is a list of all definitions that can reach a given use of a variable.

A UD chain: \( \text{UD}(S_n, v) = (S'_1, \ldots, S'_m) \).
A DU chain is a list of all uses that can be reached by a given definition of a variable. DU Chain is a counterpart of a UD Chain.

A DU chain: \( DU(S_{n'}, v) = (S_1, ..., S_k) \).
Use of DU/UD Chains in Optimization/Parallelization

- Dependence analysis
- Live variable analysis
- Alias analysis
- Analysis for various transformations
Available Expressions

An expression \( x+y \) is *available* at a point \( p \) if:

(1) Every path from the *start* node to \( p \) evaluates \( x+y \).

(2) After the last evaluation prior to reaching \( p \), there are no subsequent assignments to \( x \) or to \( y \).

We say that a basic block kills expression \( x+y \) if it *may* assign \( x \) or \( y \), and does not subsequently recomputes \( x+y \).
Available Expression: Example

Yes. It is generated in all paths leading to B4 and it is not killed after its generation in any path. Thus the redundant expression can be eliminated.
Available Expression: Example

Yes. It is generated in all paths leading to B4 and it is not killed after its generation in any path. Thus the redundant expression can be eliminated.
Available Expression: Example

Yes. It is generated in all paths leading to B4 and it is not killed after its generation in any path. Thus the redundant expression can be eliminated.
Available Expressions: gen and kill set

Assume $U$ is the “universal” set of all expressions appearing on the right of one or more statements in a program.

$e_{\text{gen}}[B]$: the set of expressions generated by $B$

$e_{\text{kill}}[B]$: the set of expressions in $U$ killed in $B$. 
Calculate the Generate Set of Available Expressions

No generated expression

\[ x = y + z \]

\( p \quad S: \emptyset \)

\( q \quad S': \text{add } y+z \text{ to } S; \text{ delete expressions involving } x \text{ from } S \)

\[ S \]

\[ \emptyset \]

\[ a = b + c \]

\[ b = a - d \]

\[ c = b + c \]

\[ d = a - d \]

\[ b + c \]

\[ b - c, \quad a - d \]

\[ a - d, \quad b + c \]

\[ \emptyset - d \]
Iterative Algorithm for Available Expressions

**Dataflow equations**

The set of expressions available at the entry of basic block $B$:

$$\text{in}[B] = \bigcap_{P \in \text{predecessor of } B} \text{out}[P]$$

The set of expressions available at the exit of basic block $B$:

$$\text{out}[B] = \text{e_gen}[B] \cup (\text{in}[B] - \text{e_kill}[B])$$

(AhoSethiUllman, pp. 606)
Iterative Algorithm for Reaching Definitions

Algorithm

1) out(ENTRY) = \emptyset ;
2) for (each basic block B other than ENTRY)
   \hspace{1cm} out(B) = U;
3) while (changes to any out occur)
4) for (each B other than ENTRY)
   \hspace{1cm} \{ \hspace{1cm}
   \hspace{1cm} \hspace{1cm} in[B] = \bigcap_{P \in \text{predecessor of } B} \text{out}[P];
   \hspace{1cm} \hspace{1cm} out[B] = e_{\text{gen}}[B] \cup (in[B] - e_{\text{kill}}[B]);
   \hspace{1cm} \}
Use of Available Expressions

- Detecting global common subexpressions
More Useful Data-Flow Frameworks

**Constant propagation** is the process of substituting the values of known constants in expressions at compile time.

**Constant folding** is a compiler optimization technique where constant subexpressions are evaluated at compiler time.
**Constant Folding Example**

\[ i = 32 \times 48 - 1530 \quad \rightarrow \quad i = 6 \]

Constant folding can be implemented:

- In a compiler’s front end on the IR tree (before it is translated into three-address codes)
- In the back end, as an adjunct to constant propagation
Constant Propagation Example

```c
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

**Constant propagation**

```c
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

**Constant folding**

```c
int x = 14;
int y = 0;
return 0;
```
Summary

- Basic Blocks
- Control Flow Graph (CFG)
- Dominator and Dominator Tree
- Natural Loops
- Program point and path
- Dataflow equations and the iterative method
- Reaching definition
- Live variable analysis
- Available expressions
Remarks of Mathematical Foundations on Solving Dataflow Equations

- As long as the dataflow value domain is “nice” (e.g., semi-lattice)
- And each function specified by the dataflow equation is “nice” -- then iterative application of the dataflow equations at each node will eventually terminate with a stable solution (a fix point).
- For mathematical foundation -- read
  - Muchnik’s book: Section 8.2, pp 223

For a good discussion: also read 9.3 (pp 618-632) in new Dragon Book
Algorithm Convergence

Intuitively we can observe that the algorithm converges to a fix point because the out[B] set never decreases in size.

It can be shown that an upper bound on the number of iterations required to reach a fix point is the number of nodes in the flow graph.

Intuitively, if a definition reaches a point, it can only reach the point through a cycle free path, and no cycle free path can be longer than the number of nodes in the graph.

Empirical evidence suggests that for real programs the number of iterations required to reach a fix point is less than five.
More remarks

- If a data-flow framework meets “good” conditions then it has a unique fixed-point solution
- The iterative algorithm finds the (best) answer
- The solution does not depend on order of computation
- Algorithm can choose an order that converges quickly
Ordering the Nodes to Maximize Propagation

Postorder
Visit children first

Reverse Postorder
Visit parents first