Constant propagation is an example of the FORWARD/FALL/PATHS problem. Intuitively, data is propagated forward in CFG, and value is constant at a point p only if it is the same constant for all paths from start to p.

General classification of dataflow problems:

<table>
<thead>
<tr>
<th>BACKWARD</th>
<th>FORWARD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY PATH</td>
<td>ALL PATHS</td>
</tr>
<tr>
<td>Live variables</td>
<td>Very busy expressions</td>
</tr>
<tr>
<td>Reaching definitions</td>
<td>Available expressions</td>
</tr>
<tr>
<td>Constant propagation</td>
<td></td>
</tr>
</tbody>
</table>

Constant propagation is an example of the BACKWARD/FALL/PATHS problem.
Available expressions: FORWARDED FLOW, ALL PATHS

Definition: An expression \( x \) of \( y \) is available at every point \( p \) if every path from START to \( p \) contains an evaluation of \( p \) after which there are no assignments to \( x \) or \( y \).

Lattice: Powerset of all expressions in program ordered by containment.

There are no assignments to \( x \) or \( y \).
compute greatest solution

confluence operator: meet (intersection)

\( E_0 \) = { }  

\( E_1 = \{ y \ op \ x \} \ U ( E_0 - E_0 ) \)  

\( E_0 \)  

\( x := y \ op \ z \)

\( \exists x \ op \ y \) = \( \{ \} = E_0 \)

EQUATIONS:
Lattice: powerset of all expressions in procedure

merge

\[ I + x =: z \]
\[ I + x =: y \]
**Reaching Definitions: Forward Flow**

Any path from START to p which contains d (and which does not contain any definitions of a variable v) is said to reach a point p if there is a definition d of a variable v as defined in procedure.

**Lattice:** powerset of definitions in procedure.
Many intermediate representations record reaching definitions in graphical form. Def-use chain: edge whose source is a definition of variable \( v \), and whose destination is a use reached by that definition. Use-def chain: reverse of def-use chain whose destination is a use of \( v \) that definition reached. Def-use chains represent information in graphical form.
A variable \( x \) is said to be live at a point \( p \) if \( x \) is used before being assigned on some path from \( p \) to END (used in register allocation).

Live variable analysis: Backward flow, Any Path
Very busy expressions: FORWARD FLOW, ALL PATHS

Lattice: powerset of expressions ordered by containment

\[ \text{Equations:} \]

\[
\begin{align*}
\text{END} & = \{\} \\
\{x := y \text{ op } z\} & \cup (\text{END} - \text{Ex})
\end{align*}
\]

Confluence operator: meet (intersection)

\[
\begin{align*}
\text{Ex} & \text{ is set of expressions containing } x \\
\{z \text{ op } y\} & \cup (\text{END} - \text{Ex})
\end{align*}
\]

Compute greatest solution

Very busy expression \( e \) is said to be \( \text{very busy} \) at a point \( p \) if it is evaluated on every path from \( p \) to \( \text{END} \) before an assignment to \( y \) or \( z \).
Pragmatics of data flow analysis

Two approaches:

- Exploit sparsity
- Exploit structure in control flow graph

Question: Can we speed up data flow analysis?

- Use bit vectors to represent sets.
- Compute and store information at basic block level.

Pragmatics of data flow analysis
Optimizing Data Analysis
Two approaches to speeding up dataflow analysis:

\[ \exists V \forall E : O \text{ available expressions on CFG} \]
\[ \exists V \forall E : O \text{ reaching definitions on CFG} \]
\[ \exists V \forall E : O \text{ constant propagation on CFG} \]
Exploiting program structure

- Work-list algorithm did not enforce any particular order for processing equations.

- Should exploit program structure to avoid revisiting equations unnecessarily.

- If this is within a loop nest, can be a big win.

- Otherwise, $e_2$ will have to be done twice.

- We should schedule $e_2$ after we have processed $e_1$ and $e_3$.

$$x = 2 \quad x = 3$$

$$\cdots y = \ldots$$
Structured Programs: Limit in which no iteration is required

- ...•
- Intervals
- basic-blocks, if-then-else, loops
- basic-blocks
- ...

What should be a region?

Interpolate dataflow solution into collapsed regions.
Solve dataflow equations iteratively on the collapsed graph.

as region

region into a single node with the same input-output behavior

Identity regions of CFG that can be preprocessed by collapsing

General approach to exploiting structure elimination
Example: reaching definitions in structured language

To summarize the effect of a region, compute $\text{gen}$ and $\text{kill}$ for each region,

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

exit or not even if they reach the beginning of $R$

$\text{Kill}[R]$: set of definitions in program that do not reach exit of $R$ even if they reach the beginning of the same variable $R$

$\text{gen}[R]$: set of definitions in $R$ from which there is a path to exit, i.e., if other definitions of the same variable exit of $R$, even if they reach the beginning of $R$

Dataflow equation for region can be written using $\text{gen}$ and $\text{kill}$ for each region.

$\text{out} = \text{gen}[R] \cup (\text{in} - \text{kill}[R])$
\[ \text{gen}[R] = \{ d \} \]

\[ \text{kill}[R] = \text{Da} \text{ (all definitions of } a) \]

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

\[ \text{in}[R2] = \text{gen}[R1] \cup (\text{in}[R] - \text{kill}[R1]) \]

\[ \text{in}[R1] = \text{in}[R] = \text{in}[R2] = \text{in}[R] \]

\[ \text{in}[R1] = \text{in}[R] \cup \text{gen}[R] \]

\[ a = b + c \]

\[ d \]
For structured programs (%like reducible programs%), we can even solve the dataflow problem directly on the abstract syntax tree (no need to build the control flow graph). We don't need to iterate.

For less structured programs (%like reducible programs%), we must build the control flow graph to identify regions like intervals, but there is still no need to iterate.

Any reaching definitions purely by elimination (without any iteration) has complexity $\mathcal{O}(\Sigma \Sigma)$.

For structured programs, we can solve dataflow problems like

Observations:
Exploiting sparsity to speed up dataflow analysis

Subtle point: In what order should we process variables?

Skipping over irrelevant portions of control flow graph

- Propagate information directly from definitions to uses.
- Do constant propagation for each variable separately.

Solution:

used only at bottom (consider a variable that is defined at top of procedure and not at bottom).

- Propagate information for all variables in lock-step forces a graph to propagate state vectors.
- CGC algorithm for constant propagation used control flow

Example: constant propagation