... but also **common ones** which cannot be rotated:

Here, the complete block between back edge and conditional jump should be duplicated  :-(
1.9 Eliminating Partially Dead Code

Example:

\[ T = x + 1; \]
\[ M[x] = T; \]

\( x + 1 \) need only be computed along one path
Idea:

\[ T = x + 1; \]

\[ M[x] = T; \]
Problem:

- The definition $x = e; \ (x \notin \text{Vars}_e)$ may only be moved to an edge where $e$ is safe ;-

- The definition must still be available for uses of $x$ ;-

We define an analysis which maximally delays computations:

\[
\boxed{[;] D \quad = \quad D}
\]

\[
[x = e;] D \quad = \quad \begin{cases} 
D \setminus (\text{Use}_e \cup \text{Def}_x) \cup \{x = e;\} & \text{if } x \notin \text{Vars}_e \\
D \setminus (\text{Use}_e \cup \text{Def}_x) & \text{if } x \in \text{Vars}_e
\end{cases}
\]
... where:

\[
\begin{align*}
Use_e &= \{ y = e'; \mid y \in Vars_e \} \\
Def_x &= \{ y = e'; \mid y \equiv x \lor x \in Vars_{e'} \}
\end{align*}
\]
... where:

\[
\text{Use}_e = \{ y = e'; \mid y \in \text{Vars}_e \}
\]
\[
\text{Def}_x = \{ y = e'; \mid y \equiv x \lor x \in \text{Vars}_{e'} \}
\]

For the remaining edges, we define:

\[
[x = M[e];] D = D\setminus(\text{Use}_e \cup \text{Def}_x)
\]
\[
[M[e_1] = e_2;] D = D\setminus(\text{Use}_{e_1} \cup \text{Use}_{e_2})
\]
\[
[\text{Pos}(e)] D = [\text{Neg}(e)] D = D\setminus\text{Use}_e
\]
Warning:

We may move \( y = e; \) beyond a join only if \( y = e; \) can be delayed along all joining edges:

\[
T = x + 1;
\]

\[
x = M[T];
\]

Here, \( T = x + 1; \) cannot be moved beyond \( 1 \) !!!
We conclude:

- The partial ordering of the lattice for delayability is given by “⊇”.
- At program start: \( D_0 = \emptyset \).

Therefore, the sets \( D[u] \) of at \( u \) delayable assignments can be computed by solving a system of constraints.

- We delay only assignments \( a \) where \( a \) has the same effect as \( a \) alone.
- The extra insertions render the original assignments as assignments to dead variables ...
Transformation 7:

\[ u \rightarrow v \]

\[ v \leftarrow lab \]

\[ a \in D[u] \backslash [lab]^{\#}(D[u]) \]

\[ u \rightarrow v \]

\[ v \leftarrow lab \]

\[ a \in [lab]^{\#}(D[u]) \backslash D[v] \]

\[ a \in D[u] \backslash [Pos(e)]^{\#}(D[u]) \]

\[ v \leftarrow Pos(e) \]

\[ v \leftarrow Neg(e) \]

\[ a \in [Neg(e)]^{\#}(D[u]) \backslash D[v_1] \]

\[ a \in [Pos(e)]^{\#}(D[u]) \backslash D[v_2] \]
Note:

Transformation \( T7 \) is only meaningful, if we subsequently eliminate assignments to dead variables by means of transformation \( T2 \) :-)

In the example, the partially dead code is eliminated:
\[ T = x + 1; \]

\[ M[x] = T; \]

<table>
<thead>
<tr>
<th></th>
<th>( D )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>1</td>
<td>( { T = x + 1; } )</td>
</tr>
<tr>
<td>2</td>
<td>( { T = x + 1; } )</td>
</tr>
<tr>
<td>3</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>4</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
\[ M[x] = T; \]

\[ T = x + 1; \]

\[ T = x + 1; \]

\[ \{ T = x + 1; \} \]

\[ \{ T = x + 1; \} \]

\[ \emptyset \]

\[ \emptyset \]

\[ \emptyset \]

\[ \emptyset \]
\[ M[x] = T; \]

\[ T = x + 1; \]

\[
\begin{array}{c|c}
\mathcal{L} & \\
0 & \{x\} \\
1 & \{x\} \\
2 & \{x\} \\
2' & \{x, T\} \\
3 & \emptyset \\
4 & \emptyset \\
\end{array}
\]
Remarks:

- After $T7$, all original assignments $y = e; \text{ with } y \notin Vars_e$ are assignments to dead variables and thus can always be eliminated :-) 
- By this, it can be proven that the transformation is guaranteed to be non-degradating efficiency of the code :-))
- Similar to the elimination of partial redundancies, the transformation can be repeated :-}
Conclusion:

→ The design of a meaningful optimization is non-trivial.

→ Many transformations are advantageous only in connection with other optimizations  :-(

→ The ordering of applied optimizations matters !!

→ Some optimizations can be iterated !!!
... a meaningful ordering:

<table>
<thead>
<tr>
<th>T4</th>
<th>Constant Propagation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Interval Analysis</td>
</tr>
<tr>
<td></td>
<td>Alias Analysis</td>
</tr>
<tr>
<td>T6</td>
<td>Loop Rotation</td>
</tr>
<tr>
<td>T1, T3, T2</td>
<td>Available Expressions</td>
</tr>
<tr>
<td>T2</td>
<td>Dead Variables</td>
</tr>
<tr>
<td>T7, T2</td>
<td>Partially Dead Code</td>
</tr>
<tr>
<td>T5, T3, T2</td>
<td>Partially Redundant Code</td>
</tr>
</tbody>
</table>
2 Replacing Expensive Operations by Cheaper Ones

2.1 Reduction of Strength

(1) Evaluation of Polynomials

\[ f(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_1 \cdot x + a_0 \]

<table>
<thead>
<tr>
<th></th>
<th>Multiplications</th>
<th>Additions</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>(\frac{1}{2}n(n+1))</td>
<td>(n)</td>
</tr>
<tr>
<td>re-use</td>
<td>(2n - 1)</td>
<td>(n)</td>
</tr>
<tr>
<td>Horner-Scheme</td>
<td>(n)</td>
<td>(n)</td>
</tr>
</tbody>
</table>
Idea:

\[ f(x) = \ldots ((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2}) \ldots) \cdot x + a_0 \]

(2) Tabulation of a polynomial \( f(x) \) of degree \( n \):

\[ \rightarrow \quad \text{To recompute } f(x) \text{ for every argument } x \text{ is too expensive } :-( \]

\[ \rightarrow \quad \text{Luckily, the } n\text{-th differences are constant } !!! \]
Example: \[ f(x) = 3x^3 - 5x^2 + 4x + 13 \]

<table>
<thead>
<tr>
<th>( n )</th>
<th>( f(n) )</th>
<th>( \Delta )</th>
<th>( \Delta^2 )</th>
<th>( \Delta^3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13</td>
<td>2</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>10</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>61</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here, the \( n \)-th difference is always
\[
\Delta^n_h(f) = n! \cdot a_n \cdot h^n \quad (h \text{ step width})
\]
Costs:

- $n$ times evaluation of $f$;
- $\frac{1}{2} \cdot (n - 1) \cdot n$ subtractions to determine the $\Delta^k$;
- $n$ additions for every further value :-) 

Number of multiplications only depends on $n$ :-))
Simple Case: $f(x) = a_1 \cdot x + a_0$

- ... naturally occurs in many numerical loops :-)
- The first differences are already constant:
  $$f(x + h) - f(x) = a_1 \cdot h$$
- Instead of the sequence: $y_i = f(x_0 + i \cdot h), \ i \geq 0$
  we compute:
  $y_0 = f(x_0), \ \Delta = a_1 \cdot h$
  $y_i = y_{i-1} + \Delta, \ i > 0$
Example:

```
for (i = i_0; i < n; i = i + h) {
    A = A_0 + b \cdot i;
    M[A] = \ldots;
}
```
... or, after loop rotation:

\[
i = i_0;
\]

if \((i < n)\) do  
\[
\begin{align*}
  A &= A_0 + b \cdot i; \\
  M[A] &= \ldots; \\
  i &= i + h;
\end{align*}
\]

\} while \((i < n)\);
... and reduction of strength:

\[ i = i_0; \]
\[
\text{if } (i < n) \ {\}
\]
\[ \Delta = b \cdot h; \]
\[ A = A_0 + b \cdot i_0; \]
\[
\text{do } \{ \]
\[ M[A] = \ldots; \]
\[ i = i + h; \]
\[ A = A + \Delta; \]
\[
\} \text{ while } (i < n); \]
Warning:

- The values $b, h, A_0$ must not change their values during the loop.
- $i, A$ may be modified at exactly one position in the loop :-(
- One may try to eliminate the variable $i$ altogether:
  - $i$ may not be used elsewhere.
  - The initialization must be transformed into:
    $$ A = A_0 + b \cdot i_0 . $$
  - The loop condition $i < n$ must be transformed into:
    $$ A < N \text{ for } N = A_0 + b \cdot n . $$
  - $b$ must always be different from zero !!!
Approach:

Identify

... loops;
... iteration variables;
... constants;
... the matching use structures.
Loops:

... are identified through the node \( v \) with back edge \((\_, \_, v) \) \(-)\)

For the sub-graph \( G_v \) of the cfg on \( \{ w \mid v \Rightarrow w \} \), we define:

\[
\text{Loop}[^v] = \{ w \mid w \rightarrow^* v \text{ in } G_v \}
\]
Example:

\[ \begin{array}{c|c}
\mathcal{P} & \\
\hline
0 & \{0\} \\
1 & \{0, 1\} \\
2 & \{0, 1, 2\} \\
3 & \{0, 1, 2, 3\} \\
4 & \{0, 1, 2, 3, 4\} \\
5 & \{0, 1, 5\} \\
\end{array} \]
Example:

\begin{itemize}
\item \textbf{0} \rightarrow \textbf{1}
\item \textbf{1} \rightarrow \textbf{2} \rightarrow \textbf{3} \rightarrow \textbf{4}
\item \textbf{5}
\end{itemize}

\begin{tabular}{|c|c|}
\hline
\textbf{P} & \textbf{0} \\
\hline
0 & \{0\} \\
\hline
1 & \{0, 1\} \\
\hline
2 & \{0, 1, 2\} \\
\hline
3 & \{0, 1, 2, 3\} \\
\hline
4 & \{0, 1, 2, 3, 4\} \\
\hline
5 & \{0, 1, 5\} \\
\hline
\end{tabular}
Example:

\[
\begin{array}{|c|}
\hline
\mathcal{P} \\
\hline
0 & \{0\} \\
1 & \{0, 1\} \\
2 & \{0, 1, 2\} \\
3 & \{0, 1, 2, 3\} \\
4 & \{0, 1, 2, 3, 4\} \\
5 & \{0, 1, 5\} \\
\hline
\end{array}
\]
We are interested in edges which during each iteration are executed exactly once:

Graph-theoretically, this is not easily expressible  :-(
Edges $k$ could be selected such that:

- the sub-graph $G = \text{Loop}[v] \setminus \{(\_, \_, v)\}$ is connected;
- the graph $G \setminus \{k\}$ is split into two unconnected sub-graphs.
Edges \( k \) could be selected such that:

- the sub-graph \( G = \text{Loop}[v] \setminus \{(\_, \_, v)\} \) is connected;
- the graph \( G \setminus \{k\} \) is split into two unconnected sub-graphs.

On the level of source programs, this is trivial:

```latex
do \{ s_1 \ldots s_k \\
    \} \text{ while (e);}
```

The desired assignments must be among the \( s_i \) :-)}