1.9 Eliminating Partially Dead Code

Example:

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

\[ x + 1 \] need only be computed along one path :-(

Problem:

- The definition \( x = e; \quad (x \notin \text{Vars}_e) \) 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:

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

\[
\text{Use}_{e_{i}} = \{ y = e' : y \in \text{Vars}_{e_{i}} \}
\]
\[
\text{Def}_{x} = \{ y = e' : y \equiv x \lor x \in \text{Vars}_{e_{i}} \}
\]

Problem:

- The definition \( x = e_{i} \ (x \not\in \text{Vars}_{e_{i}}) \) 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:

\[
\begin{align*}
[x = e_{i}]^{D} & = D \\
[x = e_{i}]^{D} & = \begin{cases} 
D \setminus (\text{Use}_{e_{i}} \cup \text{Def}_{x}) \cup \{x = e_{i}\} & \text{if } x \not\in \text{Vars}_{e_{i}} \\
D \setminus (\text{Use}_{e_{i}} \cup \text{Def}_{x}) & \text{if } x \in \text{Vars}_{e_{i}}
\end{cases}
\end{align*}
\]

... where:

\[
\text{Use}_{e_{i}} = \{ y = e' : y \in \text{Vars}_{e_{i}} \}
\]
\[
\text{Def}_{x} = \{ y = e' : y \equiv x \lor x \in \text{Vars}_{e_{i}} \}
\]

... where:

\[
\text{Use}_{e_{i}} = \{ y = e' : y \in \text{Vars}_{e_{i}} \}
\]
\[
\text{Def}_{x} = \{ y = e' : y \equiv x \lor x \in \text{Vars}_{e_{i}} \}
\]

For the remaining edges, we define:

\[
\begin{align*}
[x = M[e_{i}]]^{D} & = D \setminus (\text{Use}_{e_{i}} \cup \text{Def}_{x}) \\
[M[e_{1}] = e_{2}]^{D} & = D \setminus (\text{Use}_{e_{1}} \cup \text{Use}_{e_{2}}) \\
[\text{Pos}(e)]^{D} & = [\text{Neg}(e)]^{D} = D \setminus \text{Use}_{e_{i}}
\end{align*}
\]
Warning:

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

\[
\begin{align*}
T &= x + 1; \\
x &= M[T];
\end{align*}
\]

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

- The partial ordering of the lattice for delayability is given by “≥”.
- At program start: \( D_0 = 0 \).

Therefore, the sets \( D_u \) of delayable assignments can be computed by solving a system of constraints.

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

... where:

\[
\begin{align*}
\text{Use}_e &= \{ y = e' : y \in \text{Vars}_e \} \\
\text{Def}_x &= \{ y = e' : y \equiv x \lor x \in \text{Vars}_e \}
\end{align*}
\]

For the remaining edges, we define:

\[
\begin{align*}
[x = M[e]] & D = D \setminus (\text{Use}_e \cup \text{Def}_x) \\
[M[e_1] = e_2] & D = D \setminus (\text{Use}_{e_1} \cup \text{Use}_{e_2}) \\
[\text{Pos}(e)] & D = [\text{Neg}(e)] D = D \setminus \text{Use}_e
\end{align*}
\]

Transformation 7:
Note:
Transformation $T_7$ is only meaningful, if we subsequently eliminate assignments to dead variables by means of transformation $T_2$.
In the example, the partially dead code is eliminated:
Remarks:

- After $T^+$, all original assignments $y = e$ with $y \not\in \text{Vars}$, 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-degrading efficiency of the code $:-))$
- Similar to the elimination of partial redundancies, the transformation can be repeated $:-)$

Transformation 7:

\[ e \in D[b] \setminus \{a\} \setminus \{D[e] \setminus \{e\}\} \]

\[ e \in \{a\} \setminus \{a\} \setminus \{D[e] \setminus \{e\}\} \]
Remarks:

- After $T^7$, 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-degrading 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>

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

→ To recompute \( f(x) \) for every argument \( x \) is too expensive :-(
→ Luckily, the \( n \)-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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>0</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>0</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here, the \( n \)-th difference is always:

\[ \Delta_n^n(f) = n! \cdot a_n \cdot h^n \]  
(h step width)

Costs:

- \( n \) times evaluation of \( f \);
- \( \frac{1}{3} \cdot (n - 1) \cdot n \) subtractions to determine \( \Delta^3 \);
- \( n \) 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:
  \[ y_0 = f(x_0), \quad \Delta = a_1 \cdot h, \quad y_i = y_{i-1} + \Delta, \quad i > 0 \]
  we compute:
  \[ y_0 = f(x_0), \quad y_i = y_0 + i \cdot h, \quad i > 0 \]

Example:

\[
\begin{align*}
&\text{Neg}(i < n) \quad \text{Pos}(i \geq n) \\
&\text{for } (i = k_0; i < n; i = i + h) \{ \\
&\quad A = A_0 + b \cdot i; \\
&\quad M[A] = \ldots; \\
&\}
\end{align*}
\]
... or, after loop rotation:

\[
i = i_0;
\]

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

... and reduction of strength:

\[
i = i_0;
\]

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

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:
  
  \[
  \rightarrow i \quad \text{may not be used else-where.}
  \]
  
  \[
  \rightarrow \quad \text{The initialization must be transformed into:}
  A = A_0 + b \cdot i_0.
  \]
  
  \[
  \rightarrow \quad \text{The loop condition } i < n \quad \text{must be transformed into:}
  A < N \quad \text{for } N = A_0 + b \cdot n.
  \]
  
  \[
  \rightarrow b \quad \text{must always be different from zero !!!}
  \]
... 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 else-where.
  - The initialization must be transformed into:
    \( A = A_0 + b \cdot i_0 \).
  - The loop condition \( i < n \) must be transformed into:
    \( A < N \) for \( N = A_0 + b \cdot n \).
  - \( b \) must always be different from zero.

... 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 else-where.
  - The initialization must be transformed into:
    \( A = A_0 + b \cdot i_0 \).
  - The loop condition \( i < n \) must be transformed into:
    \( A < N \) for \( N = A_0 + b \cdot n \).
  - \( b \) must always be different from zero.
... and reduction of strength:

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

... or, after loop rotation:

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

... and reduction of strength:

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

Approach:

Identify

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

... are identified through the node $v$ with back edge $(\ldots, v)$.

For the sub-graph $G_v$ of the CFG on \{w | v \rightarrow w\}, we define:

$$\text{Loop}[v] = \{w | w \rightarrow^* v \text{ in } G_v\}$$

Example:

<table>
<thead>
<tr>
<th>$\mathcal{P}$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>${0}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>${0, 1}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>${0, 1, 2}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>${0, 1, 2, 3}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>${0, 1, 2, 3, 4}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>${0, 1, 5}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We are interested in edges which during each iteration are executed exactly once:

This property can be expressed by means of the pre-dominator relation...