... 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:

\[ x + 1 \text{ need only be computed along one path} \]
Idea:

\[ T = x + 1; \]

\[ M[x] = T; \]

Problem:

- The definition \( x = e; \) (\( x \not\in \text{Vars}_e \)) may only be moved to an edge where \( e \) is safe \( \rightarrow \)
- The definition must still be available for \( x \):

\[ \rightarrow \]

We define an analysis which maximally delays computations:

\[
\begin{align*}
[\cdot]^D & = D \\
[x = e]^D & = \begin{cases} 
D \setminus (\text{Use}_e \cup \text{Def}_x) \cup \{x = e\} & \text{if } x \not\in \text{Vars}_e \\
D \setminus (\text{Use}_e \cup \text{Def}_x) & \text{if } x \in \text{Vars}_e 
\end{cases}
\end{align*}
\]

... where:

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

For the remaining edges, we define:

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

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

\[
T = x + 1;
\]

Here, \( T = x + 1 \) cannot be moved beyond 1 !!!

We conclude:

- The partial ordering of the lattice for delayability is given by \( \geq \).
- At program start: \( D_0 = \emptyset \).
- Therefore, the sets \( D[a] \) of at \( a \) delayable assignments can be computed by solving a system of constraints.
- We delay only assignments \( a \) where \( a a \) has the same effect as \( a \) alone.
- The extra insertions render the original assignments as assignments to dead variables ...

Transformation 7:

\[
\begin{align*}
\text{Neg}(a) & \quad \text{Pos}(a) \\
\Delta \text{Neg}(a) & \quad \Delta \text{Pos}(a) \\
\end{align*}
\]

Note:

Transformation 7 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:
Remarks:

- After $T7$, all original assignments $y = c$; with $y \not\in \text{Vars}_e$ 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.

- After $T7$, all original assignments $y = c$; with $y \not\in \text{Vars}_e$ 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:

| T4 | Constant Propagation
|    | Interval Analysis
|    | Alias Analysis
| T6 | Loop Rotation
| T1, T3, T2 | Available Expressions
| T2 | Dead Variables
| T7, T2 | Partially Dead Code
| T5, T3, T2 | Partially Redundant Code

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{3}{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>
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>Method</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>

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>8</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>10</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>61</td>
<td>10</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>134</td>
<td>10</td>
<td>8</td>
<td>18</td>
</tr>
</tbody>
</table>

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

\[
 \Delta^k_n(f) = n! \cdot a_n \cdot h^n \quad (h \text{ step width})
\]
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 \( \text{:-)} \)

Number of multiplications only depends on \( n \) \( \text{:-)) \)

Simple Case:

\[ f(x) = a_1 \cdot x + a_0 \]

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

Example:

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

... or, after loop rotation:

\[ i = i_0; \]
\[ \text{if} (i < n) \; \text{do} \{ \]
  \( A = A_0 + b \cdot i \);
  \( M[A] = \ldots \);
  \( i = i + h \);  
\} \; \text{while} (i < n); \]
... and reduction of strength:

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

... or, after loop rotation:

\[ i = i_0; \]
if \((i < n)\) do {
    \[ A = A_0 + b \cdot i; \]
    \[ M[A] = \ldots; \]
    \[ i = i + h_i; \]
} 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:
    \[ A < N \quad \text{for} \quad N = A_0 + b \cdot n. \]
  - \( b \) must always be different from zero !!!

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:
    \[ A < N \quad \text{for} \quad N = A_0 + b \cdot n. \]
  - \( b \) must always be different from zero !!!
... or, after loop rotation:

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

... and reduction of strength:

\[ i = i_0; \]
\[ \text{if} \ (i < n) \ \text{do} \{
\begin{align*}
\Delta &= b \cdot h; \\
A &= A_0 + b \cdot i_0; \\
d\text{o} \{ \\
M[A] &= \ldots; \\
i &= i + h; \\
A &= A + \Delta; \\
\} \ \text{while} \ (i < n); \\
\end{align*}
\}

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:
    \[ A < N \quad \text{for} \quad N = A_0 + b \cdot n. \]
  - \( b \) must always be different from zero !!!

Approach:

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

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

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

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

Example:

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

Assume that \((u_{i-1}, v)\) is the back edge. Then edges \(k = (u_{i-1}, v_i)\) could be selected such that:

- \(v\) pre-dominates \(u_i\);
- \(u_i\) pre-dominates \(v_i\);
- \(v_i\) predominate\(s\) \(u_i\).
Assume that \((u, \ldots, v)\) is the back edge. Then edges \(k = (u_1, \ldots, v_1)\) could be selected such that:

- \(v\) pre-dominate\(s\) \(u_1\);
- \(u_1\) pre-dominate\(s\) \(v_1\);
- \(v_1\) predominate\(s\) \(u\).

On the level of source programs, this is trivial:

\[
\text{do } \{ s_1 \ldots s_k \} \text{ while } (c);
\]

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

---

**Iteration Variable:**

\(i\) is an iteration variable if the only definition of \(i\) inside the loop occurs at an edge which separates the body and is of the form:

\[
i = i + h;
\]

for some loop constant \(h\).

A loop constant is simply a constant (e.g., \(i\)), or slightly more liberal, an expression which only depends on variables which are not modified during the loop :-)

---

**Differences for Sets**

Consider the fixpoint computation:

\[
x = \emptyset; \\
\text{for } (t = F x; t \not\subseteq x; x = x \cup t);
\]

If \(F\) is distributive, it could be replaced by:

\[
x = \emptyset; \\
\text{for } (\Delta = F \Delta; \Delta \neq \emptyset; x = x \cup \Delta);
\]

The function \(F\) must only be computed for the smaller sets \(\Delta\) :-)

semi-naive iteration
Instead of the sequence:  $\emptyset \subseteq F(\emptyset) \subseteq F^2(\emptyset) \subseteq \ldots$
we compute:  \[\Delta_1 \cup \Delta_2 \cup \ldots\]
where:
\[
\Delta_{i+1} = F(F(\emptyset))(\emptyset) \\
= F(\Delta_i) \setminus (\Delta_i \cup \ldots \cup \Delta_i) \quad \text{with } \Delta_0 = \emptyset
\]

Assume that the costs of  $F \ x$  is  $1 + \#x$ .
Then the costs may sum up to:

<table>
<thead>
<tr>
<th>naive</th>
<th>$1 + 2 + \ldots + n + n$</th>
<th>$\frac{1}{2}n(n + 3)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>semi-naive</td>
<td>$2n$</td>
<td></td>
</tr>
</tbody>
</table>

where  $n$  is the cardinality of the result.

$\implies$ A linear factor is saved  $\therefore$

### 2.2 Peephole Optimization

**Idea:**
- Slide a small window over the program.
- Optimize aggressively inside the window, i.e.,
  - Eliminate redundancies!
  - Replace expensive operations inside the window by cheaper ones!

#### Examples:

- $y = M[x]; x = x + 1; \implies y = M[x++]$;  
  // given that there is a specific post-increment instruction  $\therefore$
- $z = y - a + a; \implies z = y$;  
  // algebraic simplifications  $\therefore$
- $x = x; \implies ;$
- $x = 0; \implies x = x \oplus x;$
- $x = 2 \cdot x; \implies x = x + x;$

### Examples:

- $y = M[x]; x = x + 1; \implies y = M[x++]$;  
  // given that there is a specific post-increment instruction  $\therefore$
- $z = y - a + a; \implies z = y$;  
  // algebraic simplifications  $\therefore$
- $x = x; \implies ;$
- $x = 0; \implies x = x \oplus x;$
- $x = 2 \cdot x; \implies x = x + x;$
- $y = a \times x + c$
**Important Subproblem: nop-Optimization**

1. If \((v_1, ;, v)\) is an edge, \(v_1\) has no further out-going edge.
2. Consequently, we can identify \(v_1\) and \(v\) :-)
3. The ordering of the identifications does not matter :-))

**Implementation:**

- We construct a function \(\text{next} : \text{Nodes} \rightarrow \text{Nodes}\) with:
  
  \[
  \text{next } u = \begin{cases} 
  \text{next } v & \text{if } (u, ;, v) \text{ edge} \\
  u & \text{otherwise}
  \end{cases}
  \]

  **Warning:** This definition is only recursive if there are \(\;\)-loops.

- We replace every edge:
  
  \((u, lab, v) \rightarrow (u, lab, \text{next } v)\)

  ... whenever \(lab \neq ;\)

- All \(\;\)-edges are removed :-)

**Example:**

```
next 1 = 1
next 3 = 4
next 5 = 6
```
2. Subproblem: Linearization

After optimization, the CFG must again be brought into a linearly arranged arrangement of instructions.

Warning:

Not every linearization is equally efficient!!!