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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>61</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

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:
  \[ y_i = f(x_0 + i \cdot h), \quad i \geq 0 \]
  \[ y_0 = f(x_0), \quad \Delta = a_1 \cdot h \]
  \[ y_i = y_{i-1} + \Delta, \quad i > 0 \]

Number of multiplications only depends on \( n \).
... or, after loop rotation:

```plaintext
i = i_0;

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

**Example**

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

... or, after loop rotation:

```plaintext
i = i_0;

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

\[ i = i_0; \]

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

Caveat

- 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 !!!

... and reduction of strength:

\[ i = i_0; \]

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

Caveat

- 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 !!!
... and reduction of strength:

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

---

Caveat

- 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:
    \( A < N \) for \( N = A_0 + b \cdot n \).
  - \( b \) must always be different from zero!!!

---

\[ A < A_0 + 6 \cdot n \]

Caveat

- 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:
    \( A < N \) for \( N = A_0 + b \cdot n \).
  - \( b \) must always be different from zero!!!
  \[ \epsilon = (A - A_0) / 6 \]
Caveat

- 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:
    $$A < N \quad \text{for} \quad N = A_0 + b \cdot n.$$ 
  - $b$ 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 $\langle \ldots, v \rangle$.

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{tabular}{|c|c|}
\hline
$p$ & \{ \ldots \} \\
\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{tabular}
We are interested in edges which during each iteration are executed exactly once:

![Diagram](image)

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

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

and is not contained in an inner loop.

On the level of source programs, this is trivial:

```plaintext
do \{ s_1 \ldots s_k \\
    \} while (c);
```

The desired assignments must be among the preceeding jumps.
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., 42), or slightly more literal, an expression which only depends on variables which are not modified during the loop.

(3) Differences for Sets

Consider the fixpoint computation:

\[
x = \emptyset;
\]

for \(( t = F x; t \not\in x; t = F x; \)\)

\[
x = x \cup t;
\]

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

\[
x = \emptyset;
\]

for \(( \Delta = F x; \Delta \neq \emptyset; \Delta = (F \Delta) \setminus x; \)\)

\[
x = x \cup \Delta;
\]

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

\[
\Delta_{i+1} = F(F^*(\emptyset)) \setminus F^*(\emptyset)
\]

\[= F((\Delta_i \setminus ((\Delta_1 \cup \ldots \cup \Delta_i) \setminus (\emptyset))) \text{ with } \Delta_0 = \emptyset \]

Assume that the costs of \( F \) is \( 1 + \#x \)

Then the costs may sum up to:

| naive | \( 1 + 2 + \ldots + n + n \) | \( \frac{1}{2} n(n + 3) \) |
| semi-naive | \( 2n \) |

where \( n \) is the cardinality of the result.

\[\Rightarrow\] A linear factor is saved.
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

\[
\begin{align*}
  y &= M[x]; x = x + 1; \quad \implies \quad y &= M[x+]; \\
  & \quad \text{// given that there is a specific post-increment instruction} \\
  z &= y - a + a; \quad \implies \quad z &= y; \\
  & \quad \text{// algebraic simplifications} \\
  x &= x; \quad \implies \quad ; \\
  x &= 0; \quad \implies \quad x &= x \oplus x; \\
  x &= 2 \cdot x; \quad \implies \quad x &= x + x;
\end{align*}
\]

Important Subproblem: \textit{nop}-Optimization

\[
\begin{align*}
  & \quad \text{If } (v_1, v) \text{ is an edge, } v_1 \text{ has no further out-going edge.} \\
  & \quad \text{Consequently, we can identify } v_1 \text{ and } v. \\
  & \quad \text{The ordering of the identifications does not matter.}
\end{align*}
\]
Implementation

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

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

- We replace every edge:
  \[
  (u, \text{lab}, v) \quad \Rightarrow \quad (u, \text{lab}, \text{next } v)
  \]

  ... whenever \( \text{lab} \neq ; \)

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

---

Example

2. Subproblem: Linearization

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

**Caveat**

Not every linearization is equally efficient !!!
Example

0: \( \text{if (c_1) goto 2;} \)
1: \( \text{Neg (c_1)} \)
2: \( \text{Pos (c_1)} \)
3: \( \text{if (c_2) goto 4; goto 1;} \)
4: \( \text{halt} \)
5: \( \text{Rumpl} \)

Bad: The loop body is jumped into.

Idea

- Assign to each node a temperature!
- always jumps to
  1. nodes which have already been handled;
  2. colder nodes.
- Temperature \( \approx \) nesting-depth

For the computation, we use the pre-dominator tree and strongly connected components ...
Our definition of Loop implies that (detected) loops are necessarily nested. Is is also meaningful for do-while-loops with breaks...
Summary: The Approach

(1) For every node, determine a temperature;
(2) Pre-order-DFS over the CFG;
   → If an edge leads to a node we already have generated code for, then we insert a jump.
   → If a node has two successors with different temperature, then we insert a jump to the colder of the two.
   → If both successors are equally warm, then it does not matter.

2.3 Procedures

We extend our mini-programming language by procedures without parameters and procedure calls.
For that, we introduce a new statement:

\[
f();
\]

Every procedure \( f \) has a definition:

\[
f() \{ \text{stmt}^* \}
\]

Additionally, we distinguish between global and local variables.
Program execution starts with the call of a procedure \( \text{main}() \).