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}
\text{\(P\)} & 0 \\
\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:

```

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

```
Example:

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

![Diagram showing edge relationships between nodes u and v]

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:

- \(v\) pre-dominates \(u_1\);
- \(u_1\) pre-dominates \(v_1\);
- \(v_1\) predominates \(u\).
Assume that \((u, _, v)\) is the back edge.

Then edges \(k = (u_1, _, v_1)\) could be selected such that:

- \(v\) pre-dominates \(u_1\);
- \(u_1\) pre-dominates \(v_1\);
- \(v_1\) pre-dominates \(u\).

On the level of source programs, this is trivial:

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

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., 42), or slightly more liberal, an expression which only depends on variables which are not modified during the loop :-)

495
(3) Differences for Sets

Consider the fixpoint computation:

\[
x = \emptyset;
\]

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

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

\[
x = \emptyset;
\]

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

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:
\[
\Delta_{i+1} = F(F^i(\emptyset)) \setminus F^i(\emptyset)
= F(\Delta_i) \setminus (\Delta_1 \cup \ldots \cup \Delta_i)
\]
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>Method</th>
<th>Costs</th>
<th>( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>( 1 + 2 + \ldots + n + n )</td>
<td>( \frac{1}{2}n(n+3) )</td>
</tr>
<tr>
<td>semi-naive</td>
<td>( 2n )</td>
<td></td>
</tr>
</tbody>
</table>

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

\[ \Longrightarrow \quad \text{A linear factor is saved} \quad :-) \]
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:

\[ y = M[x]; \ x = x + 1; \quad \implies \quad y = M[x++]; \]
// given that there is a specific post-increment instruction  :-)

\[ z = y - a + a; \quad \implies \quad z = y; \]
// algebraic simplifications  :-)

\[ x = x; \quad \implies \quad ; \]
\[ x = 0; \quad \implies \quad x = x \oplus x; \]
\[ x = 2 \cdot x; \quad \implies \quad x = x + x; \]
If \((v_1, ;, v)\) is an edge, \(v_1\) has no further out-going edge.

Consequently, we can identify \(v_1\) and \(v\) :-)

The ordering of the identifications does not matter :-))
Implementation:

• We construct a function $\text{next} : \text{Nodes} \rightarrow \text{Nodes}$ with:

$$\text{next } u = \begin{cases} 
\text{next } v & \text{if } (u, ;, v) \text{ edge} \\
\text{next } v & \text{otherwise}
\end{cases}$$

Warning: This definition is only recursive if there are $;$-loops

• We replace every edge:

$$(u, lab, v) \implies (u, lab, \text{next } v)$$

... whenever $lab \neq ;$

• All $;$-edges are removed $;-$)
Example:

next 1 = 1
next 3 = 4
next 5 = 6
Example:

\[\begin{align*}
\text{next } 1 & = 1 \\
\text{next } 3 & = 4 \\
\text{next } 5 & = 6
\end{align*}\]
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 !!!
Example:

0: if \( e_1 \) goto 2;

2: Rumpf

3: if \( e_2 \) goto 4;
goto 1;

Bad: The loop body is jumped into :-(

0:
1: if \( e_1 \) goto 2;
4: halt
2: Rumpf
3: if \( e_2 \) goto 4;
goto 1;
Example:

0:
1: if (!\(e_1\)) goto 4;
2: Rumpf
3: if (!\(e_2\)) goto 1;
4: halt

// better cache behavior :-)
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 ...
... in the Example:

The sub-tree with back edge is hotter ...
... in the Example:
More Complicated Example:
More Complicated Example:
More Complicated Example:
Our definition of Loop implies that (detected) loops are necessarily nested :-)

Is is also meaningful for do-while-loops with breaks ...
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:

\[
\begin{align*}
f() & \{ \text{stmt}* \}
\end{align*}
\]

Additionally, we distinguish between global and local variables.

Program execution starts with the call of a procedure \( \text{main()} \).
Example:

```c
int  a, ret;
main () {
    a = 3;
    f();
    M[17] = ret;
    ret = 0;
}
```

```c
f () {
    int  b;
    if (a <= 1) {ret = 1; goto exit; }
    b = a;
    a = b - 1;
    f();
    ret = b * ret;
}
exit :
}
```

Such programs can be represented by a set of CFGs: one for each procedure ...
... in the Example:

```
main()

0
a = 3;

1
f();

2
M[17] = ret;

3
ret = 0;

4

5

6
Neg (a \leq 1)

7
b = a;

8
a = b - 1;

9
f();

10
Pos (a \leq 1)

11
ret = 1;

f()

ret = b * ret;

ret = 0;
```

518