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

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 \(v_1\);
- \(v_1\) predominate \(u\).

This property can be expressed by means of the pre-dominator relation ...
We are interested in edges which during each iteration are executed exactly once:

Assume that \((u, v)\) is the back edge. Then edges \(k = (u_1, u_1)\) could be selected such that:

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

This property can be expressed by means of the pre-dominator relation ...
Assume that \((u, v, w)\) 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 \(u\).

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 libral, 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 \(t = F x; t \nsubseteq x; \{ t = F x \}\)

\[
x = x \cup t;
\]

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 \(\Delta\) during the 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>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 \( :-) \)

---

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!

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

\[ y = M[x]; \ x = x + 1; \quad \implies \quad y = M[x+1]; \]

// given that there is a specific post-increment instruction \( \implies \)
\[ z = y - a + a; \quad \implies \quad z = y; \]

// algebraic simplifications \( \implies \)
\[ x = x; \quad \implies \quad x; \]
\[ x = 0; \quad \implies \quad x = x \oplus x; \]
\[ x = 2 \cdot x; \quad \implies \quad x = x + x; \]

Important Subproblem: \( \textit{nop-Optimization} \)

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

Implementation:

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

  Warning: This definition is only recursive if there are \( \cdot \)-loops

- We replace every edge:
  \[
  (u, \text{lab}, v) \quad \implies \quad (u, \text{lab}, \text{next} \ v)
  \]
  ... whenever \( \text{lab} \neq \cdot \);

- All \( \cdot \)-edges are removed \( \implies \)

Example:

\[ \begin{array}{c}
  \text{next} \ 1 = 1 \\
  \text{next} \ 3 = 4 \\
  \text{next} \ 5 = 6
\end{array} \]
2. Subproblem: Linearization

After optimization, the CFG must again be brought into a linearly arrangement of instructions :-)

Warning:
Not every linearization is equally efficient !!!
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:

Our definition of \textbf{Loop} implies that (detected) loops are necessarily nested :-(
Is it also meaningful for \textbf{do-while-loops} with \textbf{breaks} ...

More Complicated Example:

Our definition of \textbf{Loop} implies that (detected) loops are necessarily nested :-(
Is it also meaningful for \textbf{do-while-loops} with \textbf{breaks} ...
Our definition of Loop implies that (detected) loops are necessarily nested.

Is it 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 \( \text{main}() \).
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 \( \text{main}() \).

Example:

```c
int a, ret;

main() {
    int b;
    a = 3;
    if (a ≤ 1) \{ ret = 1; goto exit; \}
    f();
    b = a;
    M[17] = ret;
    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:
In order to optimize such programs, we require an extended operational semantics.

Program executions are no longer paths, but forests:

The function \([\cdot]\) is extended to computation forests: \(w:\)

\[w : (\text{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z}) \to (\text{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z})\]

For a call \(k = (u, f(), v)\) we must:

- determine the initial values for the locals:
  \[\text{enter } \rho = \{x \mapsto 0 \mid x \in \text{Locals}\} \oplus (\rho|_{\text{Globals}})\]

- ... combine the new values for the globals with the old values for the locals:
  \[\text{combine } (\rho_1, \rho_2) = (\rho_1|_{\text{Locals}}) \oplus (\rho_2|_{\text{Globals}})\]

- ... evaluate the computation forest inbetween:
  \[\llbracket k (w) \rrbracket (\rho, \mu) = \text{let } (\rho_1, \mu_1) = \llbracket w \rrbracket (\text{enter } \rho, \mu)\]
  \[\text{in } \text{combine } (\rho, \rho_1, \mu_1)\]