An expression $e$ is called busy along a path $\pi$, if the expression $e$ is evaluated before any of the variables $x \in Vars(e)$ is overwritten.

// backward analysis!

e is called very busy at $u$, if $e$ is busy along every path $\pi : u \to^* \text{stop}$. 

Accordingly, we require:

$$B[u] = \bigcap \{[\pi]^2 \mid \pi : u \to^* \text{stop} \}$$

where for $\pi = k_1 \ldots k_m$:

$$[\pi]^2 = [k_1]^2 \circ \ldots \circ [k_m]^2$$

Our complete lattice is given by:

$$\mathcal{B} = 2^{\text{Expr} \setminus \text{Vars}}$$

with $\subseteq = \supseteq$

The effect $[k]^2$ of an edge $k = (u, lab, v)$ only depends on $lab$, i.e., $[k]^2 = [lab]^2$ where:

$$[\emptyset]^2 B = B$$

$$[\text{Pos}(e)]^2 B = [\text{Neg}(e)]^2 B = B \cup \{e\}$$

$$[x = e_1]^2 B = (B \setminus \text{Expr}_x) \cup \{e\}$$

$$[x = M[e]]^2 B = (B \setminus \text{Expr}_x) \cup \{e\}$$

$$[M[e_1] = e_2]^2 B = B \cup \{e_1, e_2\}$$
These effects are all **distributive**, Thus, the least solution of the constraint system yields precisely the MOP — given that \textit{stop} is reachable from every program point\:-)

**Example:**

![Diagram](image)

<table>
<thead>
<tr>
<th>\text{7}</th>
<th>\emptyset</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{6}</td>
<td>{y_1 + y_2}</td>
</tr>
<tr>
<td>\text{5}</td>
<td>{x + 1}</td>
</tr>
<tr>
<td>\text{4}</td>
<td>{x + 1}</td>
</tr>
<tr>
<td>\text{3}</td>
<td>{x + 1}</td>
</tr>
<tr>
<td>\text{2}</td>
<td>{x + 1}</td>
</tr>
<tr>
<td>\text{1}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>\text{0}</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>

A point \(u\) is called **safe** for \(e\), if \(e \in \mathcal{A}[u] \cup \mathcal{B}[u]\), i.e., \(e\) is either available or very busy.

**Idea:**

- We insert computations of \(e\) such that \(e\) becomes available at all safe program points \:-)
- We insert \(T_e = e;\) after every edge \((u, \text{\textit{lab}}, v)\) with

\[
e \in \mathcal{B}[v] \setminus \{\text{\textit{lab}}\}_A^\mathcal{A}(\mathcal{A}[u] \cup \mathcal{B}[u])
\]

**Transformation 5.1:**

- \(T_e = e;\)  \((e \in \mathcal{B}[v] \setminus \{\text{\textit{lab}}\}_A^\mathcal{A}(\mathcal{A}[u] \cup \mathcal{B}[u]))\)
Transformation 5.2:

\[ e = e; \]
\[ x = T_e; \]

// analogously for the other uses of \( e \)
// at old edges of the program.

In the Example:

\[
\begin{array}{c|c|c}
\text{A} & \text{B} \\
\hline
0 & 0 & 0 \\
1 & 0 & 0 \\
2 & 0 & \{x + 1\} \\
3 & 0 & \{x + 1\} \\
4 & \{x + 1\} & \{x + 1\} \\
5 & 0 & \{x + 1\} \\
6 & \{x + 1\} & \{y_1 + y_2\} \\
7 & \{x + 1, y_1 + y_2\} & 0 \\
\end{array}
\]

In the Example:

\[
\begin{array}{c|c|c}
\text{A} & \text{B} \\
\hline
0 & 0 & 0 \\
1 & 0 & 0 \\
2 & 0 & \{x + 1\} \\
3 & 0 & \{x + 1\} \\
4 & \{x + 1\} & \{x + 1\} \\
5 & 0 & \{x + 1\} \\
6 & \{x + 1\} & \{y_1 + y_2\} \\
7 & \{x + 1, y_1 + y_2\} & 0 \\
\end{array}
\]
Correctness:

Let $\pi$ denote a path reaching $v$ after which a computation of an edge with $e$ follows.

Then there is a maximal suffix of $\pi$ such that for every edge $k = (u, lab, u')$ in the suffix:

$$e \in \text{lab}_A^2(\mathcal{A}[u] \cup \mathcal{B}[v])$$

Correctness:

Let $\pi$ denote a path reaching $v$ after which a computation of an edge with $e$ follows.

Then there is a maximal suffix of $\pi$ such that for every edge $k = (u, lab, u')$ in the suffix:

$$e \in \text{lab}_A^2(\mathcal{A}[u] \cup \mathcal{B}[v])$$

In particular, no variable in $e$ receives a new value $\rightarrow$

Then $T_e = e$ is inserted before the suffix $\rightarrow$
We conclude:

- Whenever the value of $e$ is required, $e$ is available $\Rightarrow$
  $\implies$ correctness of the transformation

- Every $T = e$; which is inserted into a path corresponds to an $e$
  which is replaced with $T$ $\Rightarrow$
  $\implies$ non-degradation of the efficiency

Correctness:

Let $\pi$ denote a path reaching $v$ after which a computation of an edge with $e$ follows.

Then there is a maximal suffix of $\pi$ such that for every edge $k = (u, lab, u')$ in the suffix:

$$e \in [a\beta]\rho(A[u] \cup B[v])$$

1.8 Application: Loop-invariant Code

Example:

$$\text{for } (i = 0; i < n; i++)$$

$$a[i] = b + 3;$$

// The expression $b + 3$ is recomputed in every iteration $\Rightarrow$
// This should be avoided $\Rightarrow$
The Control-flow Graph:

... now there is a place for 

\[ T = c; \quad :-) \]

Application of TS (PRE):

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>1</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>2</td>
<td>\emptyset</td>
<td>{b + 3}</td>
</tr>
<tr>
<td>3</td>
<td>{b + 3}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>4</td>
<td>{b + 3}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>5</td>
<td>{b + 3}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>6</td>
<td>{b + 3}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>7</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>
Application of T5 (PRE):

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>{b + 3}</td>
</tr>
<tr>
<td>3</td>
<td>{b + 3}</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>{b + 3}</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>{b + 3}</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>{b + 3}</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Conclusion:

- Elimination of partial redundancies may move loop-invariant code out of the loop :-(
- This only works properly for do-while-loops :-(
- To optimize other loops, we transform them into do-while-loops beforehand:

while (b) stmt → if (b)

  do stmt

  while (b);

  Loop Rotation
Conclusion:

- Elimination of partial redundancies may move loop-invariant code out of the loop.
- This only works properly for do-while-loops.
- To optimize other loops, we transform them into do-while-loops before-hand:

\[
\text{while } (b) \text{ stmt } \quad \longrightarrow \quad \text{if } (b) \\
\quad \quad \quad \text{do stmt} \\
\quad \quad \quad \quad \text{while } (b); \\
\longrightarrow \quad \text{Loop Rotation}
\]

Problem:

If we do not have the source program at hand, we must re-construct potential loop headers:

\[ \quad \longrightarrow \quad \text{Pre-dominators} \]

\[ u \text{ pre-dominates } v, \text{ if every path } \pi : \text{start} \rightarrow^* v \text{ contains } u. \text{ We write: } u \Rightarrow v. \]

\[ \Rightarrow \quad \text{is reflexive, transitive and anti-symmetric} \quad \longrightarrow \]
Problem:

If we do not have the source program at hand, we must re-construct potential loop headers :-)

\[ u \text{ pre-dominates } v, \text{ if every path } \pi : \text{start} \rightarrow^* v \text{ contains } u. \text{ We write: } u \Rightarrow v. \]

\[ \Rightarrow \] is reflexive, transitive and anti-symmetric :-)

Computation:

We collect the nodes along paths by means of the analysis:

\[ \mathcal{P} = 2^{\text{Nodes}}, \quad \mathcal{C} = \mathcal{C} \]

\[ \left[ \left( \psi \downarrow \downarrow v \right)^{I} \right] \mathcal{P} = \mathcal{P} \cup \{v\} \]

Then the set \( \mathcal{P}[v] \) of pre-dominators is given by:

\[ \mathcal{P}[v] = \bigcap \{ \left[ \psi \right]^{I} \{ \text{start} \} \mid \psi : \text{start} \rightarrow^* v \} \]

Example:

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

Since \( [k]^I \) are distributive, the \( \mathcal{P}[v] \) can be computed by means of fixpoint iteration :-)

Example:

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

Since \( [k]^I \) are distributive, the \( \mathcal{P}[v] \) can be computed by means of fixpoint iteration :-)}
The partial ordering "\(\Rightarrow\)" in the 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>

Apparently, the result is a tree :-) In fact, we have:

**Theorem:**

Every node \(v\) has at most one immediate pre-dominator.

**Proof:**

Assume:

there are \(u_1 \neq u_2\) which immediately pre-dominate \(v\).

If \(u_1 \Rightarrow u_2\) then \(u_1\) not immediate.

Consequently, \(u_1, u_2\) are incomparable :-) 

Now for every \(\pi : \text{start} \rightarrow^* v\):

\[
\pi = \pi_1 \pi_2 \quad \text{with} \quad \pi_1 : \text{start} \rightarrow^* u_1
\]

\[
\pi_2 : u_1 \rightarrow^* v
\]

If, however, \(u_1, u_2\) are incomparable, then there is path: \(\text{start} \rightarrow^* v\) avoiding \(u_2\):

![Diagram showing paths]

Observation:

The loop head of a while-loop pre-dominates every node in the body.

A back edge from the exit \(u\) to the loop head \(v\) can be identified through

\(v \in \mathcal{P}[u]\) :-)

Accordingly, we define:
Transformation 6:

We duplicate the entry check to all back edges :-(

... in the Example:

... in the Example:

Warning:

There are unusual loops which cannot be rotated:

Pre-dominators:
... but also common ones which cannot be rotated:

Here, the complete block between back edge and conditional jump should be duplicated  :( 

... but also common ones which cannot be rotated:

Here, the complete block between back edge and conditional jump should be duplicated  :( 

... 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:

\[ T = x + 1; \]

\[ M(x) = T; \]

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

0

\[ T = x + 1; \]

1

\[ M[x] = T; \]

2

3

4

\[ T = x + 1; \]

\[ M[x] = T; \]

5