Generalization: Static Single Assignment Form

We proceed in two phases:

**Step 1:**
Transform the program such that each program point \( v \) is reached by at most one definition of a variable \( x \) which is live at \( v \).

**Step 2:**
- Introduce a separate variant \( x_i \) for every occurrence of a definition of a variable \( x \)!
- Replace every use of \( x \) with the use of the reaching variant \( x_h \) ...
Implementing Step 1:

- Determine for every program point the set of reaching definitions.
- If the join point \( v \) is reached by more than one definition for the same variable \( x \) which is live at program point \( v \), insert definitions \( x = x \); at the end of each incoming edge.
Example

```latex
\text{Reaching Definitions}

\begin{tabular}{|c|c|}
\hline
\text{\(R\)} & \\
\hline
0 & \(\langle x, 0 \rangle, \langle y, 0 \rangle\) \\
1 & \(\langle x, 1 \rangle, \langle y, 0 \rangle\) \\
2 & \(\langle x, 1 \rangle, \langle x, 5 \rangle, \langle y, 2 \rangle, \langle y, 4 \rangle\) \\
3 & \(\langle x, 1 \rangle, \langle x, 5 \rangle, \langle y, 2 \rangle, \langle y, 4 \rangle\) \\
4 & \(\langle x, 1 \rangle, \langle x, 5 \rangle, \langle y, 4 \rangle\) \\
5 & \(\langle x, 5 \rangle, \langle y, 4 \rangle\) \\
6 & \(\langle x, 1 \rangle, \langle x, 5 \rangle, \langle y, 2 \rangle, \langle y, 4 \rangle\) \\
7 & \(\langle x, 1 \rangle, \langle x, 5 \rangle, \langle y, 2 \rangle, \langle y, 4 \rangle\) \\
\hline
\end{tabular}
```

\[x = M[I];\]
\[y = 1;\]
\[M[R] = y;\]
\[\text{Neg}(x > 1);\]
\[\text{Pos}(x > 1);\]
\[x = x - 1;\]
Example

Reaching Definitions

<table>
<thead>
<tr>
<th></th>
<th>$\mathcal{R}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\langle x, 0 \rangle, \langle y, 0 \rangle$</td>
</tr>
<tr>
<td>1</td>
<td>$\langle x, 1 \rangle, \langle y, 0 \rangle$</td>
</tr>
<tr>
<td>2</td>
<td>$\langle x, 1 \rangle, \langle x, 5 \rangle, \langle y, 2 \rangle, \langle y, 4 \rangle$</td>
</tr>
<tr>
<td>3</td>
<td>$\langle x, 1 \rangle, \langle x, 5 \rangle, \langle y, 2 \rangle, \langle y, 4 \rangle$</td>
</tr>
<tr>
<td>4</td>
<td>$\langle x, 1 \rangle, \langle x, 5 \rangle, \langle y, 4 \rangle$</td>
</tr>
<tr>
<td>5</td>
<td>$\langle x, 5 \rangle, \langle y, 4 \rangle$</td>
</tr>
<tr>
<td>6</td>
<td>$\langle x, 1 \rangle, \langle x, 5 \rangle, \langle y, 2 \rangle, \langle y, 4 \rangle$</td>
</tr>
<tr>
<td>7</td>
<td>$\langle x, 1 \rangle, \langle x, 5 \rangle, \langle y, 2 \rangle, \langle y, 4 \rangle$</td>
</tr>
</tbody>
</table>

where $\psi \equiv x = x \mid y = y$
Reaching Definitions

The complete lattice $\mathbb{R}$ for this analysis is given by:

$$\mathbb{R} = 2^{\text{Defs}}$$

where

$$\text{Defs} = \text{Vars} \times \text{Nodes} \quad \text{Defs}(x) = \{x\} \times \text{Nodes}$$

Then:

$$\llbracket (\_, x = r; v) \rrbracket^\#_R = R \setminus \text{Defs}(x) \cup \{\langle x, v \rangle\}$$
$$\llbracket (\_, x = x \mid x \in L, v) \rrbracket^\#_R = R \setminus \bigcup_{x \in L} \text{Defs}(x) \cup \{\langle x, v \rangle \mid x \in L\}$$

The ordering on $\mathbb{R}$ is given by subset inclusion $\subseteq$ where the value at program start is given by $R_0 = \{\langle x, \text{start} \rangle \mid x \in \text{Vars}\}$. 
The Transformation SSA, Step 1:

where \( k \geq 2 \).

The label \( \psi \) of the new in-going edges for \( v \) is given by:

\[
\psi \equiv \{ x = x \mid x \in \mathcal{L}[v], \#(\mathcal{R}[v] \cap \text{Defs}(x)) > 1 \}
\]
If the node $v$ is the start point of the program, we add auxiliary edges whenever there are further ingoing edges into $v$:

The Transformation SSA, Step 1 (cont.):

where $k \geq 1$ and $\psi$ of the new in-going edges for $v$ is given by:

$$\psi \equiv \{ x = x \mid x \in \mathcal{L}[v], \#(\mathcal{R}[v] \cap \text{Defs}(x)) > 1 \}$$
Discussion

- Program start is interpreted as (the end point of) a definition of every variable $x$.
- At some edges, parallel definitions $\psi$ are introduced.
- Some of them may be useless :-(

616
Discussion

- Program start is interpreted as (the end point of) a definition of every variable \( x \) :-)
- At some edges, parallel definitions \( \psi \) are introduced !
- Some of them may be useless :-(

Improvement:

- We introduce assignments \( x = x \) before \( v \) only if the sets of reaching definitions for \( x \) at incoming edges of \( v \) differ !
- This introduction is repeated until every \( v \) is reached by exactly one definition for each variable live at \( v \).
Theorem

Assume that every program point in the controlflow graph is reachable from \texttt{start} and that every left-hand side of a definition is live. Then:

1. The algorithm for inserting definitions \( x = x \) terminates after at most \( n \cdot (m + 1) \) rounds were \( m \) is the number of program points with more than one in-going edges and \( n \) is the number of variables.

2. After termination, for every program point \( u \), the set \( \mathcal{R}[u] \) has exactly one definition for every variable \( x \) which is live at \( u \).
Discussion

The efficiency crucially depends on the number of iterations. If the cfg is well-structured, it terminates already after one iteration!
Discussion

The efficiency crucially depends on the number of iterations. If the cfg is well-structured, it terminates already after one iteration!

A well-structured cfg can be reduced to a single vertex or edge by:
Discussion

The efficiency crucially depends on the number of iterations. If the cfg is well-structured, it terminates already after one iteration!

A well-structured cfg can be reduced to a single vertex or edge by:
Discussion (cont.)

- Reducible cfgs are not the exception — but the rule :-) 
- In Java, reducibility is only violated by loops with breaks/continues.
- If the insertion of definitions does not terminate after \( k \) iterations, we may immediately terminate the procedure by inserting definitions \( x = x \) before all nodes which are reached by more than one definition of \( x \).

Assume now that every program point \( u \) is reached by exactly one definition for each variable which is live at \( u \) ...
The Transformation SSA, Step 2:

Each edge \((u, \text{lab}, v)\) is replaced with \((u, T_v, \phi[\text{lab}], v)\) where 
\[\phi x = x_{u'}\] if \(\langle x, u' \rangle \in R[u]\) and:

\[T_{v, \phi}[;] = ;\]
\[T_{v, \phi}[\text{Neg}(e)] = \text{Neg}(\phi(e))\]
\[T_{v, \phi}[\text{Pos}(e)] = \text{Pos}(\phi(e))\]
\[T_{v, \phi}[x = e] = x_v = \phi(e)\]
\[T_{v, \phi}[x = M[e]] = x_v = M[\phi(e)]\]
\[T_{v, \phi}[M[e_1] = e_2] = M[\phi(e_1)] = \phi(e_2)\]
\[T_{v, \phi}[\{x = x \mid x \in L\}] = \{x_v = \phi(x) \mid x \in L\}\]
Remark

The multiple assignments:

\[ pa = x_v^{(1)} = x_{v1}^{(1)} | \ldots | x_v^{(k)} = x_{vk}^{(k)} \]

in the last row are thought to be executed in parallel, i.e.,

\[ \left[ pa \right] (\rho, \mu) = (\rho \oplus \{ x_v^{(i)} \mapsto \rho(x_v^{(i)}_i) \mid i = 1, \ldots, k \}, \mu) \]
Example

\[ x_1 = M[I]; \]
\[ y_1 = 1; \]

Neg(\(x_3 > 1\))

Pos(\(x_3 > 1\))

\[ M[R] = y_3; \]

\[ y_2 = x_3 \times y_3; \]

\[ x_2 = x_3 - 1; \]

\[ \psi_1 = x_3 = x_1 \mid y_3 = y_1 \]
\[ \psi_2 = x_3 = x_2 \mid y_3 = y_2 \]
Theorem

Assume that every program point is reachable from \texttt{start} and the program is in SSA form without assignments to dead variables.

Let \( \lambda \) denote the maximal number of simultaneously live variables and \( G \) the interference graph of the program variables. Then:

\[
\lambda = \omega(G) = \chi(G)
\]

where \( \omega(G), \chi(G) \) are the maximal size of a clique in \( G \) and the minimal number of colors for \( G \), respectively.

A minimal coloring of \( G \), i.e., an optimal register allocation can be found in polynomial time.
Discussion

- By the theorem, the number $\lambda$ of required registers can be easily computed :-)

- Thus variables which are to be spilled to memory, can be determined ahead of the subsequent assignment of registers !

- Thus here, we may, e.g., insist on keeping iteration variables from inner loops.
Discussion

• By the theorem, the number $\lambda$ of required registers can be easily computed :-)

• Thus variables which are to be spilled to memory, can be determined ahead of the subsequent assignment of registers !

• Thus here, we may, e.g., insist on keeping iteration variables from inner loops.

• Clearly, always $\lambda \leq \omega(G) \leq \chi(G')$ :-)

Therefore, it suffices to color the interference graph with $\lambda$ colors.

• Instead, we provide an algorithm which directly operates on the cfg ...
Observation

- Live ranges of variables in programs in SSA form behave similar to live ranges in basic blocks!
- Consider some dfs spanning tree $T$ of the cfg with root $\text{start}$. 
- For each variable $x$, the live range $\mathcal{L}[x]$ forms a tree fragment of $T$!
- A tree fragment is a subtree from which some subtrees have been removed ...
Example

\[ x = M[i]; \]
\[ y = 1; \]
\[ M[a] = y; \]
\[ y = x \times y; \]
\[ x = x - 1; \]

Neg(\( x > 1 \))

Pos(\( x > 1 \))
Discussion

- Although the example program is not in SSA form, all live ranges still form tree fragments :-) 

- The intersection of tree fragments is again a tree fragment!

- A set $C$ of tree fragments forms a clique iff their intersection is non-empty !!!

- The greedy algorithm will find an optimal coloring ...
Proof of the Intersection Property

(1) Assume \( I_1 \cap I_2 \neq \emptyset \) and \( v_i \) is the root of \( I_i \). Then:

\[
v_1 \in I_2 \quad \text{or} \quad v_2 \in I_1
\]

(2) Let \( C \) denote a clique of tree fragments. Then there is an enumeration \( C = \{I_1, \ldots, I_r\} \) with roots \( v_1, \ldots, v_r \) such that

\[
v_i \in I_j \quad \text{for all} \quad j \leq i
\]

In particular, \( v_r \in I_i \) for all \( i \). :-)}
The Greedy Algorithm

\[
\text{forall}\ (u \in \text{Nodes}) \ \text{visited}[u] = \text{false};
\]

\[
\text{forall}\ (x \in \mathcal{L}[\text{start}]) \ \Gamma(x) = \text{extract}(\text{free});
\]

\text{alloc}(\text{start});

\[
\text{void alloc (Node } u) \ {\}
\]

\[
\text{visited}[u] = \text{true};
\]

\[
\text{forall}\ ((\text{lab}, v) \in \text{edges}[u])
\]

\[
\text{if } (\neg \text{visited}[v]) \ {\}
\]

\[
\text{forall}\ (x \in \mathcal{L}[u]\setminus\mathcal{L}[v]) \ \text{insert}(\text{free}, \Gamma(x));
\]

\[
\text{forall}\ (x \in \mathcal{L}[v]\setminus\mathcal{L}[u]) \ \Gamma(x) = \text{extract}(\text{free});
\]

\text{alloc}(v);

};
Example

0
read();

1
x = M[A];

2
y = x + 1;

3
Neg(y)  Pos(y)

4
t = −y · y;

5
M[A] = t;

6
z = x · x

7
M[A] = z;

8
Example

read();

\( x = M[A]; \)

\( y = x + 1; \)

\( \text{Neg}(y) \)

\( \text{Pos}(y) \)

\( t = -y \cdot y; \)

\( z = x \cdot x \)

\( M[A] = t; \)

\( M[A] = z; \)

read();

\( R_1 = M[A]; \)

\( R_2 = R_1 + 1; \)

\( \text{Neg}(R_2) \)

\( \text{Pos}(R_2) \)

\( R_1 = -R_2 \cdot R_2; \)

\( R_1 = R_1 \cdot R_1 \)

\( M[A] = R_1; \)

\( M[A] = R_1; \)
Remark:

- Intersection graphs for tree fragments are also known as **cordal graphs** ...
- A cordal graph is an undirected graph where every cycle with more than three nodes contains a **cord** :-)
- Cordal graphs are another sub-class of **perfect graphs** :-))

- Cheap register allocation comes at a price:

  when transforming into **SSA** form, we have introduced parallel register-register moves  :-(

636
Problem

The parallel register assignment:

\[ \psi_1 = R_1 = R_2 | R_2 = R_1 \]

is meant to exchange the registers \( R_1 \) and \( R_2 \) :-(

There are at least two ways of implementing this exchange ...
Problem

The parallel register assignment:

\[ \psi_1 = R_1 = R_2 \mid R_2 = R_1 \]

is meant to exchange the registers \( R_1 \) and \( R_2 \) :-)

There are at least two ways of implementing this exchange ...

(1) Using an auxiliary register:

\[
\begin{align*}
R &= R_1; \\
R_1 &= R_2; \\
R_2 &= R;
\end{align*}
\]
(2) XOR:

\[
R_1 = R_1 \oplus R_2; \\
R_2 = R_1 \oplus R_2; \\
R_1 = R_1 \oplus R_2;
\]
(2) XOR:

\[
R_1 = R_1 \oplus R_2; \\
R_2 = R_1 \oplus R_2; \\
R_1 = R_1 \oplus R_2;
\]

But what about cyclic shifts such as:

\[
\psi_k = R_1 = R_2 \mid \ldots \mid R_{k-1} = R_k \mid R_k = R_1
\]

for \( k > 2 \) ??
(2) XOR:

\[ R_1 = R_1 \oplus R_2; \]
\[ R_2 = R_1 \oplus R_2; \]
\[ R_1 = R_1 \oplus R_2; \]

But what about cyclic shifts such as:

\[ \psi_k = R_1 = R_2 | \ldots | R_{k-1} = R_k | R_k = R_1 \]

for \( k > 2 \)?

Then at most \( k - 1 \) swaps of two registers are needed:

\[ \psi_k = R_1 \leftrightarrow R_2; \]
\[ R_2 \leftrightarrow R_3; \]
\[ \ldots \]
\[ R_{k-1} \leftrightarrow R_k; \]