How to arrive at SSA 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 \( x_i \).

Discussion
- Every live variable should be defined at most once ??
- Every live variable should have at most one definition ?
- All definitions of the same variable should have a common end point !!!

⇒ Static Single Assignment Form

Implementing Step 1:
- Determine for every program point the set of reaching definitions.
- **Assumption**
  All incoming edges of a join point \( v \) are labeled with the same parallel assignment \( x = x \mid x \in L_v \) for some set \( L_v \).
  Initially, \( L_v = \emptyset \) for all \( v \).
- 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 \( x \) into \( L_v \), i.e., add definitions \( x = x \) at the end of each incoming edge of \( v \).
Example

\[ x = M[f]; \]
\[ y = 1; \]
\[ M[R] = y; \]
\[ x = x - 1; \]
\[ \psi \]

\[ \neg(x > 1) \]
\[ M[R] = y; \]
\[ x = x - 1; \]

where \( \psi = x = x \mid y = y \)

Example

\[ x = M[f]; \]
\[ y = 1; \]
\[ M[R] = y; \]
\[ x = x - 1; \]
\[ \psi \]

\[ \neg(x > 1) \]
\[ M[R] = y; \]
\[ x = x - 1; \]

where \( \psi = x = x \mid y = y \)

Example

\[ x = M[f]; \]
\[ y = 1; \]
\[ M[R] = y; \]
\[ x = x - 1; \]

\[ \psi \]

\[ \neg(x > 1) \]
\[ M[R] = y; \]
\[ x = x - 1; \]

where \( \psi = x = x \mid y = y \)

Example

\[ x = M[f]; \]
\[ y = 1; \]
\[ M[R] = y; \]
\[ x = x - 1; \]
\[ \psi \]

\[ \neg(x > 1) \]
\[ M[R] = y; \]
\[ x = x - 1; \]

where \( \psi = x = x \mid y = y \)

Reaching Definitions

The complete lattice \( \mathcal{R} \) for this analysis is given by:

\[ \mathcal{R} = 2^{\text{Defs}} \]

where

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

Then:

\[ ([_., x = r; v])^R = \mathcal{R} \setminus \text{Defs}(x) \cup \{(x, v)\} \]
\[ ([_., x = x | x \in L; v])^R = \mathcal{R} \setminus \bigcup_{v \in L} \text{Defs}(x) \cup \{(x, v) | x \in L\} \]

The ordering on \( \mathcal{R} \) is given by subset inclusion \( \subseteq \) where the value at program start is given by \( R_0 = \{(x, \text{start}) | x \in \text{Vars}\} \).
The Transformation SSA, Step 1:

where $k \geq 2$.

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

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

---

Discussion

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

---

If the node $v$ is the start point of the program, we add auxiliary edges whenever there are further in-going 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 $x : x$
- At some edges, parallel definitions $\psi$ are introduced !!
- Some of them may be useless $\neg$

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 where \( m \) 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 \( R[u] \) 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 !

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

\begin{center}
\begin{tikzpicture}
\node (a) at (0,0) [circle,draw] {a};
\node (b) at (1,0) [circle,draw] {b};
\node (c) at (2,0) [circle,draw] {c};
\draw (a) -- (b);
\end{tikzpicture}
\end{center}

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:

\begin{center}
\begin{tikzpicture}
\node (a) at (0,0) [circle,draw] {a};
\node (b) at (1,0) [circle,draw] {b};
\node (c) at (2,0) [circle,draw] {c};
\draw (a) -- (b);
\end{tikzpicture}
\end{center}
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 \((u, lab, v)\) is replaced with \((u, T_{\phi}[lab], v)\) where \(\phi x = x_u\) if \((x, u') \in R[u]\) and:

\[
\begin{align*}
T_{\phi}[\cdot] &= : \\
T_{\phi}[\text{Neg}(e)] &= \text{Neg}(\phi(e)) \\
T_{\phi}[\text{Pos}(e)] &= \text{Pos}(\phi(e)) \\
T_{\phi}[x = e] &= x_e = \phi(e) \\
T_{\phi}[x = M(e)] &= x_e = M(\phi(e)) \\
T_{\phi}[M[e_1] = e_2] &= M[\phi(e_1)] = \phi(e_2) \\
T_{\phi}[[x = x \mid x \in L]] &= [x_v = \phi(x) \mid x \in L]
\end{align*}
\]

The Transformation SSA, Step 2:

Each edge \((u, lab, v)\) is replaced with \((u, T_{\phi}[lab], v)\) where \(\phi x = x_u\) if \((x, u') \in R[u]\) and:

\[
\begin{align*}
T_{\phi}[\cdot] &= : \\
T_{\phi}[\text{Neg}(e)] &= \text{Neg}(\phi(e)) \\
T_{\phi}[\text{Pos}(e)] &= \text{Pos}(\phi(e)) \\
T_{\phi}[x = e] &= x_e = \phi(e) \\
T_{\phi}[x = M(e)] &= x_e = M(\phi(e)) \\
T_{\phi}[M[e_1] = e_2] &= M[\phi(e_1)] = \phi(e_2) \\
T_{\phi}[[x = x \mid x \in L]] &= [x_v = \phi(x) \mid x \in L]
\end{align*}
\]
The Transformation SSA, Step 2:

Each edge \((u, lab, v)\) is replaced with \((u, T_{v, \phi}[lab], v)\) where \(\phi x = x_u'\) if \((x, u') \in \mathcal{R}[u]\) and:

\[
\begin{align*}
T_{v, \phi}[:] & = : \\
T_{v, \phi}[\text{Neg}(e)] & = \text{Neg}(\phi(e)) \\
T_{v, \phi}[\text{Pos}(e)] & = \text{Pos}(\phi(e)) \\
T_{v, \phi}[x = e] & = x_v = \phi(e) \\
T_{v, \phi}[x = M[e]] & = x_v = M[\phi(e)] \\
T_{v, \phi}[M[e_1] = e_2] & = M[\phi(e_1)] = \phi(e_2)] \\
T_{v, \phi}[x = x \mid x \in L]] & = \{x_v = \phi(x) \mid x \in L\}
\end{align*}
\]

Remark

The multiple assignments:

\(pa = x_v^{(1)} = x_v^{(3)} \cdots x_v^{(k)} = x_v^{(k)}\)

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

\([x]_v (\rho, \mu) : (\rho \oplus \{x_v^{(i)} \rightarrow \rho[v^{(i)}] \mid i = 1 \ldots k\}, \mu)\)
Example

![Diagram of SSA form]

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 start.
- For each variable $x$, the live range $L[x]$ forms a tree fragment of $T$!
- A tree fragment is a subtree from which some subtrees have been removed ...

Example

![Example diagram]

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 $I_r$. Then:

$$v_i \in I_2 \text{ or } v_i \in I_1$$

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

$$v_i \in I_j \text{ for all } j \leq i$$

In particular, $v_i \in I_i$ for all $i$. ☑

Example
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 :-(

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 ...

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 :-(

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:

\[
R = R_1; \\
R_1 = R_2; \\
R_2 = R_1 
\]
(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:

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

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

for \( k > 2 \)?

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

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

Next complicated case: permutations.

- Every permutation can be decomposed into a set of disjoint shifts.
- Any permutation of \( n \) registers with \( r \) shifts can be realized by \( n - r \) swaps ...

Example

\[ \psi = R_1 = R_2 \quad | \quad R_3 = R_2 \quad | \quad R_3 = R_4 \quad | \quad R_5 = R_1 \]

consists of the cycles \((R_1, R_2, R_3)\) and \((R_3, R_4)\). Therefore:

\[ \psi = R_1 \leftrightarrow R_2; \]
\[ R_2 \leftrightarrow R_3; \]
\[ R_3 \leftrightarrow R_4; \]

The general case:

- Every register receives its value at most once.
- The assignment therefore can be decomposed into a permutation together with tree-like assignments (directed towards the leaves) ...

Example

\[ \psi = R_1 = R_2 \quad | \quad R_3 = R_4 \quad | \quad R_5 = R_6 \quad | \quad R_5 = R_3 \]

The parallel assignment realizes the linear register moves for \( R_1, R_2 \) and \( R_4 \) together with the cyclic shift for \( R_5 \) and \( R_6 \):

\[ \psi = R_1 \leftrightarrow R_2; \]
\[ R_2 \leftrightarrow R_3; \]
\[ R_3 \leftrightarrow R_4; \]

\[ R_5 \leftrightarrow R_6; \]