### Special Case: Basic Blocks

<table>
<thead>
<tr>
<th>( L )</th>
<th>( x, y, z )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_1 = x + y; )</td>
<td>( x, y, z )</td>
</tr>
<tr>
<td>( M[A_1] = z; )</td>
<td>( x, z )</td>
</tr>
<tr>
<td>( x = x + 1; )</td>
<td>( x )</td>
</tr>
<tr>
<td>( z = M[A_1]; )</td>
<td>( x, z )</td>
</tr>
<tr>
<td>( t = M[x]; )</td>
<td>( x, z, t )</td>
</tr>
<tr>
<td>( A_2 = x + t; )</td>
<td>( x, z, t )</td>
</tr>
<tr>
<td>( M[A_2] = z; )</td>
<td>( x, t )</td>
</tr>
<tr>
<td>( y = M[x]; )</td>
<td>( y, t )</td>
</tr>
<tr>
<td>( M[y] = t; )</td>
<td>( y, t )</td>
</tr>
</tbody>
</table>

The live ranges of \( x \) and \( z \) can be split:
The live ranges of $x$ and $z$ can be split:

| $A_1 = x + y$; $M[A_1] = z$; $x_1 = x + 1$; $z_1 = M[A_1]$; $t = M[x_1]$; $A_2 = x_1 + t$; $M[A_2] = z_1$; $y_1 = M[x_1]$; $M[y_1] = t$ | \[ x, y, z \]
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$x_1$</td>
</tr>
<tr>
<td>$z_1$</td>
<td>$z_1$</td>
</tr>
<tr>
<td>$t$</td>
<td>$t$</td>
</tr>
<tr>
<td>$x_1 + t$</td>
<td>$x_1 + t$</td>
</tr>
<tr>
<td>$z_1$</td>
<td>$z_1$</td>
</tr>
<tr>
<td>$y_1$</td>
<td>$y_1$</td>
</tr>
<tr>
<td>$t$</td>
<td>$t$</td>
</tr>
</tbody>
</table>

Interference graphs for minimal live ranges on basic blocks are known as interval graphs:

vertex — interval
edge — joint vertex
The covering number of a vertex is given by the number of incident intervals.

**Theorem:**

maximal covering number

- size of the maximal clique
- minimally necessary number of colors

Graphs with this property (for every sub-graph) are called perfect ... A minimal coloring can be found in polynomial time ...)

**Idea:**

- Conceptually iterate over the vertices \(0, \ldots, m-1\)!
- Maintain a list of currently free colors.
- If an interval starts, allocate the next free color.
- If an interval ends, free its color.

This results in the following algorithm:

```
free = [1, \ldots, k];
for (i = 0; i < m; i++) {
    init[i] = []; exit[i] = [];
}
forall (I = [u, v] \in Intervals) {
    init[u] = (I :: init[u]); exit[v] = (I :: exit[v]);
}
for (i = 0; i < m; i++) {
    forall (I \in init[i]) {
        color[I] = hd free; free = tl free;
    }
    forall (I \in exit[i]) free = color[I] :: free;
}
```
Discussion:

→ For arbitrary programs, we thus may apply some heuristics for graph coloring ...
→ If the number of real register does not suffice, the remaining variables are spilled into a fixed area on the stack.
→ Generally, variables from inner loops are preferably held in registers.
→ For basic blocks we have succeeded to derive an optimal register allocation :-)

The number of required registers could even be determined beforehand !
→ This works only once live ranges have been split ...

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 \( x = x_i \) at the end of each incoming edge.
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

Reaching Definitions

<table>
<thead>
<tr>
<th>$R$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$(x, 0), (y, 0)$</td>
</tr>
<tr>
<td>1</td>
<td>$(x, 1), (y, 0)$</td>
</tr>
<tr>
<td>2</td>
<td>$(x, 1), (x, 5), (y, 2), (y, 4)$</td>
</tr>
<tr>
<td>3</td>
<td>$(x, 1), (x, 5), (y, 2), (y, 4)$</td>
</tr>
<tr>
<td>4</td>
<td>$(x, 1), (x, 5), (y, 4)$</td>
</tr>
<tr>
<td>5</td>
<td>$(x, 5), (y, 4)$</td>
</tr>
<tr>
<td>6</td>
<td>$(x, 1), (x, 5), (y, 2), (y, 4)$</td>
</tr>
<tr>
<td>7</td>
<td>$(x, 1), (x, 5), (y, 2), (y, 4)$</td>
</tr>
</tbody>
</table>

Example

Reaching Definitions

<table>
<thead>
<tr>
<th>$R$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$(x, 0), (y, 0)$</td>
</tr>
<tr>
<td>1</td>
<td>$(x, 1), (y, 0)$</td>
</tr>
<tr>
<td>2</td>
<td>$(x, 1), (x, 5), (y, 2), (y, 4)$</td>
</tr>
<tr>
<td>3</td>
<td>$(x, 1), (x, 5), (y, 2), (y, 4)$</td>
</tr>
<tr>
<td>4</td>
<td>$(x, 1), (x, 5), (y, 4)$</td>
</tr>
<tr>
<td>5</td>
<td>$(x, 5), (y, 4)$</td>
</tr>
<tr>
<td>6</td>
<td>$(x, 1), (x, 5), (y, 2), (y, 4)$</td>
</tr>
<tr>
<td>7</td>
<td>$(x, 1), (x, 5), (y, 2), (y, 4)$</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:

$$\mathbb{R} = 2^{Def}$$

where

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

Then:

$$[(\_ , x = r , v)]^{\mathbb{R}} = R \setminus Def(x) \cup \{(x , v)\}$$

$$[(\_ , x = x \mid x \in L , v)]^{\mathbb{R}} = R \setminus \bigcup_{x \in L} Def(x) \cup \{(x , v) \mid x \in L\}$$

The ordering on $\mathbb{R}$ is given by subset inclusion $\subseteq$, where the value at program start is given by $I_0 = \{(x , \text{start}) \mid x \in \text{Vars}\}$.

Assumption:

No join point is the endpoint of several definitions of the same variable.

The Transformation SSA, Step 1:

where $k \geq 2$.

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

$$\psi = \{x = x \mid x \in L[v] , \#(R[v] \cap Defs(x)) > 1\}$$
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.):**

![Diagram showing the transformation SSA, Step 1](image)

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

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

---

**Discussion**

- Program start is interpreted as (the end point of) a definition of every variable \( x : \cdot \)
- 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 control flow graph is reachable from \( \text{start} \) and that every left-hand side of a definition is live.

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 \( \alpha \), the set \( \mathcal{R}[\alpha] \) has exactly one definition for every variable \( x \) which is live at \( \alpha \).
Theorem

Assume that every program point in the control flow 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 $R[u]$ 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:
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 \((u, lab, v)\) is replaced with \((u, T_{v, o}[lab], v)\) where \(\phi x = x'\) if \((x, x') \in R[u]\) and:

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

Remark

The multiple assignments:

\[
p_u = x^{(i)}_n = x^{(i)}_1 \mid \ldots \mid x^{(k)}_n = x^{(k)}_n
\]

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

\[
[p_\mu](\rho, \mu) = (\rho \oplus \{x^{(i)}_n \mapsto \rho(x^{(i)}_n) \mid i = 1, \ldots, k\}, \mu)
\]

Example

![Diagram](image)

Theorem

Assume that every program point is reachable from \textcolor{red}{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.
- Clearly, always $\lambda \leq \omega(G) \leq \chi(G) \quad :)$
  Therefore, it suffices to color the interference graph with $\lambda$ colors.
- Instead, we provide an algorithm which directly operates on the cfg ... 

Example

\[
\begin{align*}
\text{Neg}(x > 1) & : M[a] = y; \\
\text{Pos}(x > 1) & : y = x \times y; \\
\text{Neg}(x < 1) & : x = M[i]; \\
\text{Pos}(x < 1) & : y = x \times y;
\end{align*}
\]

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 $L[x]$ forms a tree fragment of $T$ $!$
- A tree fragment is a subtree from which some subtrees have been removed ...

Example

\[
\begin{align*}
\text{Neg}(x > 1) & : M[a] = y; \\
\text{Pos}(x > 1) & : y = x \times y; \\
\text{Neg}(x < 1) & : x = M[i]; \\
\text{Pos}(x < 1) & : y = x \times y;
\end{align*}
\]
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 ...

The Greedy Algorithm

```plaintext
forall (u ∈ Nodes) visited[u] = false;
forall (x ∈ L[start]) Γ(x) = extract(free);
alloc(start);

void alloc (Node u) {
    visited[u] = true;
    forall ((lab, v) ∈ edges[u])
        if (!visited[v]) {
            forall (x ∈ L[u] \ L[v]) insert(free, Γ(x));
            forall (x ∈ L[v] \ L[u]) Γ(x) = extract(free);
            alloc(v);
        }
}
```

Example

```
0: read();
1: x = M[A];
2: y = x + 1;
3: t = -y - y;
4: M[A] = t;
5: Neg(y)
6: Pos(y)
7: M[A] = z;
8: R1 = R1 - R2 - R2;
9: R1 = R1 - R1 - R1;
```

Example

```
0: read();
1: x = M[A];
2: y = x + 1;
3: t = -y - y;
4: M[A] = t;
5: Neg(y)
6: Pos(y)
7: M[A] = z;
8: R1 = R1 - R2 - R2;
9: R1 = R1 - R1 - R1;
```
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 :-(