Script **generated by TTT**

Title: Seidl: Programmoptimierung (07.01.2016)

Date: Thu Jan 07 08:36:31 CET 2016

Duration: 74:01 min

Pages: 41

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 !!!

$\Longrightarrow$ Static Single Assignment Form

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_h ...

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.

Implementing Step 1

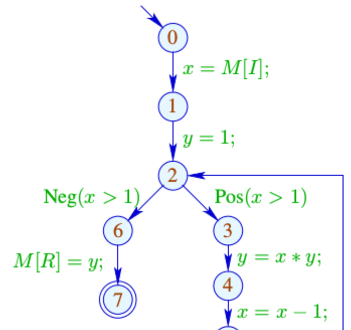
- Determine for every program point the set of reaching definitions.
- **Assumption**

  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

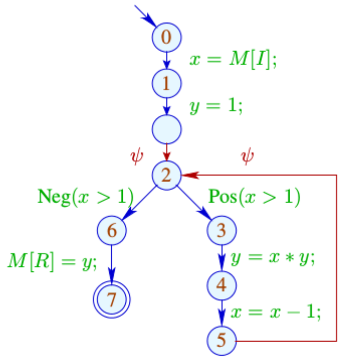
Reading Definitions



	$\mathcal{R}$
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	$\langle x,1\rangle,\langle x,5\rangle,\langle y,2\rangle,\langle y,4\rangle$

Example

Reading Definitions



	$\mathcal{R}$
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	$\langle x,1\rangle,\langle x,5\rangle,\langle y,2\rangle,\langle y,4\rangle$

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

Reaching Definitions

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

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

where

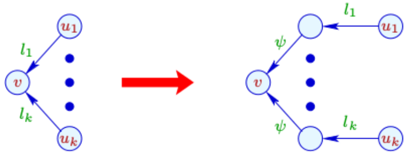
$$\mathit{Defs} = \mathit{Vars} \times \mathit{Nodes} \qquad \mathit{Defs}(x) = \{x\} \times \mathit{Nodes}$$

Then:

$$\begin{array}{lcl} [\![(\_, x = r;, v)]\!]^\sharp R & = & R \backslash \mathit{Defs}(x) \cup \{\langle x, v\rangle\} \\ [\![(\_, x = x \mid x \in L, v)]\!]^\sharp R & = & R \backslash \bigcup_{x \in L} \mathit{Defs}(x) \cup \{\langle x, v\rangle \mid x \in L\} \end{array}$$

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

The Transformation SSA, Step 1



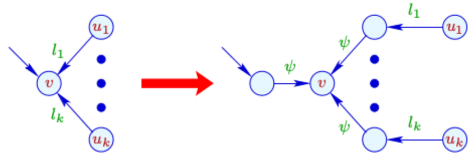
where $k \geq 2$.

The label ψ of the new in-going edges for v is given by:

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

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.)

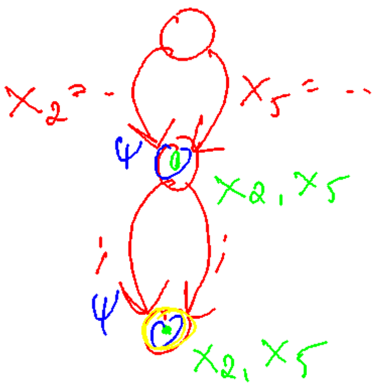


where $k \geq 1$ and ψ of the new in-going edges for v is given by:

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

Discussion

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



Discussion

- Program start is interpreted as (the end point of) a definition of every variable x.
- At some edges, parallel definitions ψ 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 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

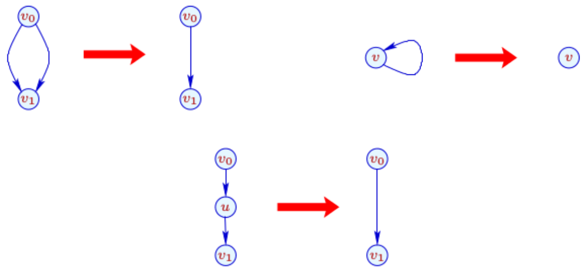
The efficiency crucially depends on the number of iterations. If the cfg is well-structured, it terminates already after one iteration !

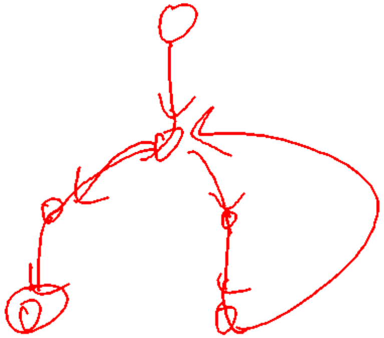
A well-structured cfg can be reduced to a single vertex or edge by:

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:

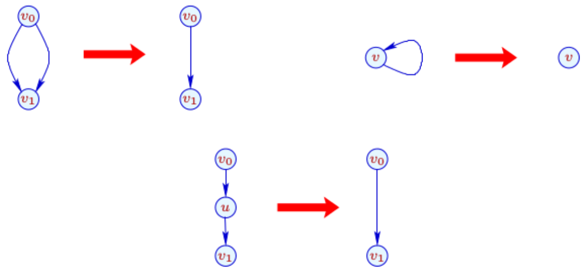




Discussion

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 $(u, \mathcal{T}_{v,\phi}[lab], v)$ where $\phi\, x = x_{u'}$ if $\langle x, u' \rangle \in \mathcal{R}[u]$ and:

$$
\begin{array}{lll}
\mathcal{T}_{v,\phi}[\,;\,] & = & ; \\
\mathcal{T}_{v,\phi}[\text{Neg}(e)] & = & \text{Neg}(\phi(e)) \\
\mathcal{T}_{v,\phi}[\text{Pos}(e)] & = & \text{Pos}(\phi(e)) \\
\mathcal{T}_{v,\phi}[x = e] & = & x_v = \phi(e) \\
\mathcal{T}_{v,\phi}[x = M[e]] & = & x_v = M[\phi(e)] \\
\mathcal{T}_{v,\phi}[M[e_1] = e_2] & = & M[\phi(e_1)] = \phi(e_2)] \\
\mathcal{T}_{v,\phi}[\{x = x \mid x \in L\}] & = & \{x_v = \phi(x) \mid x \in L\}
\end{array}
$$

Remark

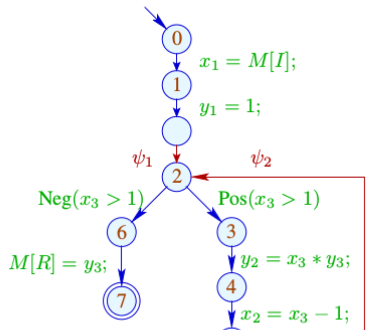
The multiple assignments:

$$pa = x_v^{(1)} = x_{v_1}^{(1)} \mid \ldots \mid x_v^{(k)} = x_{v_k}^{(k)}$$

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

$$[\![pa]\!]\,(\rho, \mu) = (\rho \oplus \{{x^{(i)}}_v \mapsto \rho({x^{(i)}}_{v_i}) \mid i = 1, \ldots, k\}, \mu)$$

Example



$$\psi_1 = x_3 = x_1 \mid y_3 = y_1$$
$$\psi_2 = x_3 = x_2 \mid y_3 = y_2$$

Theorem

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

Let λ 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

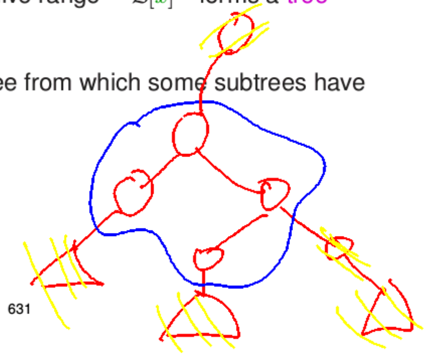
- By the theorem, the number λ of required registers can be easily computed.
- Thus variables which are to be spilled to memory, can be determined ahead of the subsequent assignment of registers.
- Thus here, we may, e.g., insist on keeping iteration variables from inner loops.

Discussion

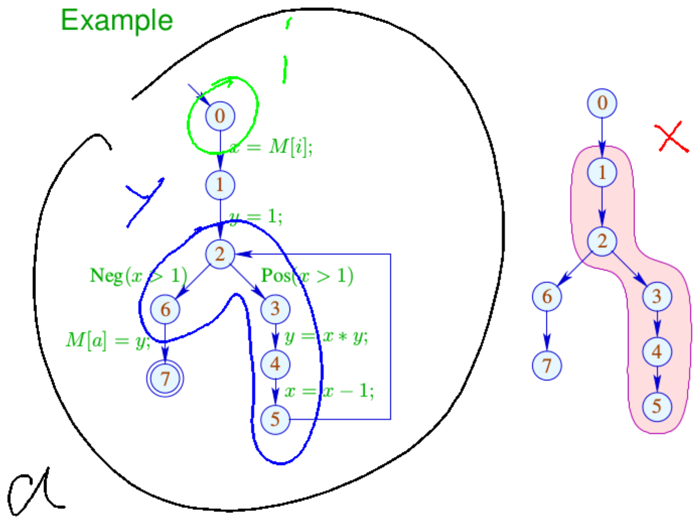
- By the theorem, the number λ of required registers can be easily computed.
- Thus variables which are to be spilled to memory, can be determined ahead of the subsequent assignment of registers.
- Thus here, we may, e.g., insist on keeping iteration variables from inner loops.
- Clearly, always $\lambda \leq \omega(G) \leq \chi(G)$.
  Therefore, it suffices to color the interference graph with λ 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 $\mathcal{L}[x]$ forms a tree fragment of T.
- A tree fragment is a subtree from which some subtrees have been removed ...



Example



Proof of the Intersection Property

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

$$v_1 \in I_2 \text{ or } v_2 \in I_1$$

(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

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

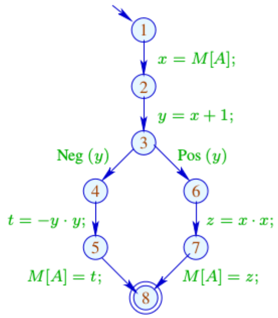
In particular, $v_r \in I_i$ for all i.

The Greedy Algorithm

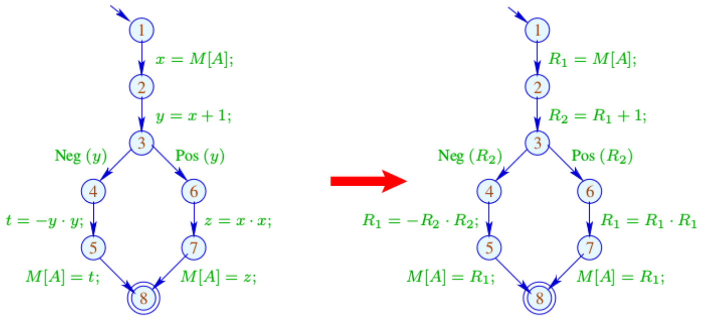
$\textbf{forall}\ (u \in \mathit{Nodes})\ \mathit{visited}[u] = \textbf{false};$
$\textbf{forall}\ (x \in \mathcal{L}[\mathit{start}])\ \Gamma(x) = \mathsf{extract}(\mathit{free});$
$\mathsf{alloc}(\mathit{start});$

$\textbf{void}\ \mathsf{alloc}\ (\textbf{Node}\ u)\ \{$
$\quad \mathit{visited}[u] = \textbf{true};$
$\quad \textbf{forall}\ ((\mathit{lab}, v) \in \mathit{edges}[u])$
$\qquad \textbf{if}\ (\neg \mathit{visited}[v])\ \{$
$\qquad\quad \textbf{forall}\ (x \in \mathcal{L}[u] \backslash \mathcal{L}[v])\ \mathsf{insert}(\mathit{free}, \Gamma(x));$
$\qquad\quad \textbf{forall}\ (x \in \mathcal{L}[v] \backslash \mathcal{L}[u])\ \Gamma(x) = \mathsf{extract}(\mathit{free});$
$\qquad\quad \mathsf{alloc}\ (v);$
$\qquad\quad \}$
$\}$

Example



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 ...

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:

$$\begin{aligned} R &= R_1; \\ R_1 &= R_2; \\ R_2 &= R; \end{aligned}$$

(2) XOR:

$$\begin{aligned} R_1 &= R_1 \oplus R_2; \\ R_2 &= R_1 \oplus R_2; \\ R_1 &= R_1 \oplus R_2; \end{aligned}$$

(2) XOR:

$$\begin{aligned} R_1 &= R_1 \oplus R_2; \\ R_2 &= R_1 \oplus R_2; \\ R_1 &= R_1 \oplus R_2; \end{aligned}$$

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:

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

for $k > 2$??

Then at most $k-1$ swaps of two registers are needed:

$$\begin{aligned} \psi_k \quad = \quad & R_1 \leftrightarrow R_2; \\ & R_2 \leftrightarrow R_3; \\ & \ldots \\ & R_{k-1} \leftrightarrow R_k; \end{aligned}$$

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 ...

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

$$\psi = R_1 = R_2 \mid R_2 = R_5 \mid R_3 = R_4 \mid R_4 = R_3 \mid R_5 = R_1$$

consists of the cycles (R_1, R_2, R_5) and (R_3, R_4). Therefore:

$$\begin{aligned} \psi \quad = \quad & R_1 \leftrightarrow R_2; \\ & R_2 \leftrightarrow R_5; \\ & R_3 \leftrightarrow R_4; \end{aligned}$$

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

$$\psi = R_1 = R_2 \mid R_2 = R_4 \mid R_3 = R_5 \mid R_5 = R_3$$

The parallel assignment realizes the linear register moves for R_1, R_2 and R_4 together with the cyclic shift for R_3 and R_5:

$$\begin{aligned} \psi \quad = \quad & R_1 = R_2; \\ & R_2 = R_4; \\ & R_3 \leftrightarrow R_5; \end{aligned}$$

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

$$\psi = R_1 = R_2 \mid R_2 = R_4 \mid R_3 = R_5 \mid R_5 = R_3$$

The parallel assignment realizes the linear register moves for R_1, R_2 and R_4 together with the cyclic shift for R_3 and R_5:

$$\begin{aligned} \psi \quad = \quad & R_1 = R_2; \\ & R_2 = R_4; \\ & R_3 \leftrightarrow R_5; \end{aligned}$$

Interprocedural Register Allocation

→ For every local variable, there is an entry in the stack frame.

→ Before calling a function, the locals must be saved into the stack frame and be restored after the call.

→ Sometimes there is hardware support.
Then the call is transparent for all registers.

→ If it is our responsibility to save and restore, we may ...

- save only registers which are over-written;
- restore overwritten registers only.

→ Alternatively, we save only registers which are still live after the call — and then possibly into different registers ⟹ reduction of life ranges