Script **generated by TTT**

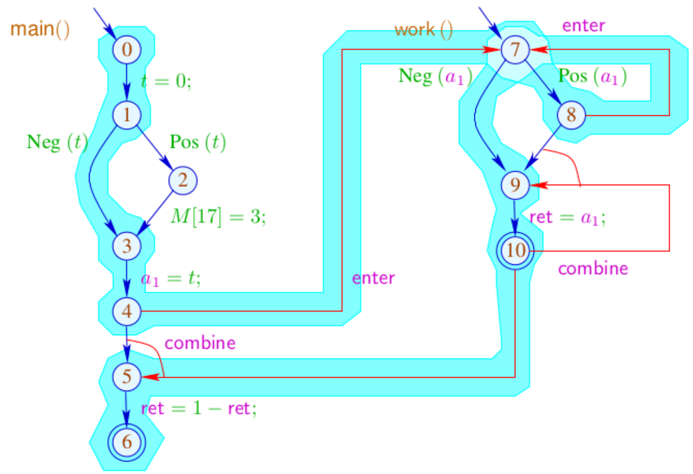
Title: Seidl: Programmoptimierung (08.01.2014)

Date: Wed Jan 08 08:31:12 CET 2014

Duration: 88:52 min

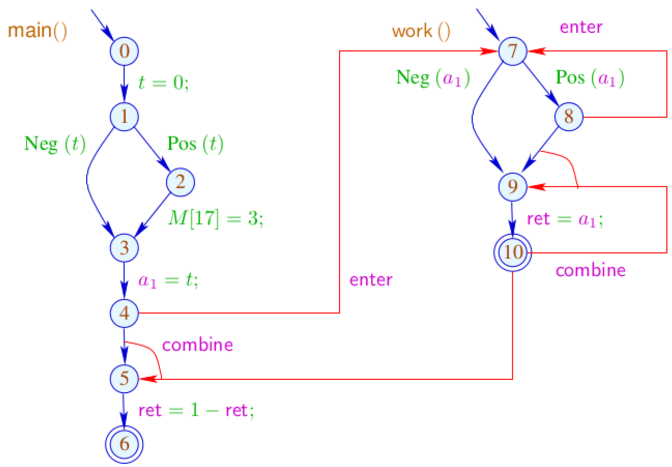
Pages: 41

... in the Example this is:



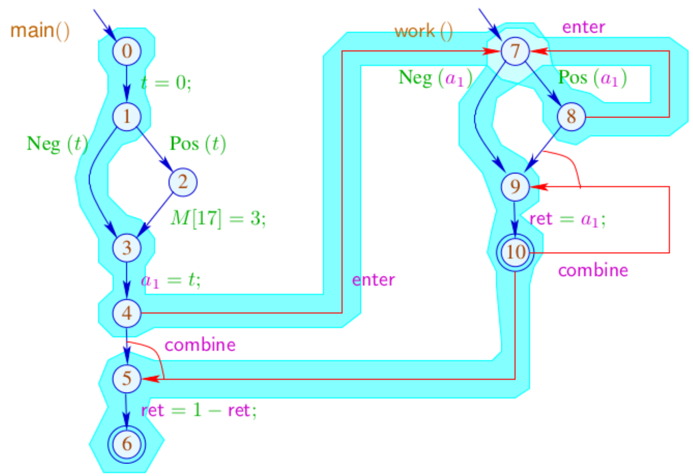
... in the Example this is:

$[\![f]\!]^\#: \mathbb{D} \to \mathbb{D}$

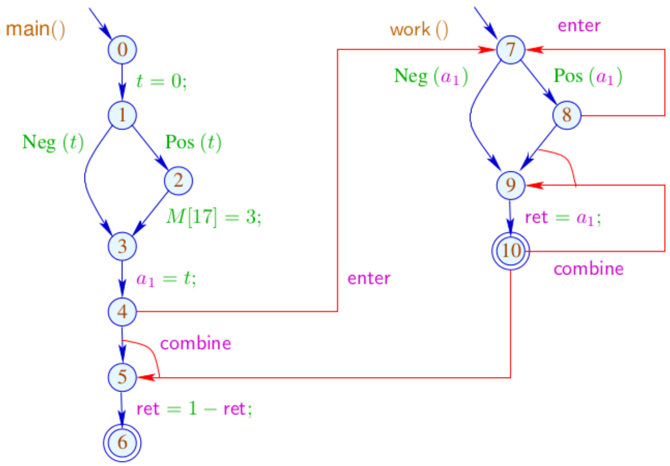


... in the Example this is:

$[\![f]\!]^\#: \mathbb{D} \to \mathbb{D}$



... in the Example this is:



The conditions for $5, 7, 10$, e.g., are:

$$\mathcal{R}[5] \sqsupseteq \textsf{combine}^\sharp\,(\mathcal{R}[4], \mathcal{R}[10])$$

$$\mathcal{R}[7] \sqsupseteq \textsf{enter}^\sharp\,(\mathcal{R}[4])$$

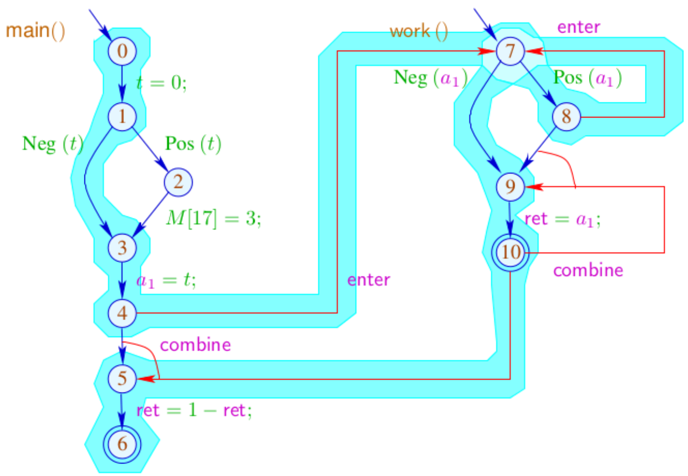
$$\mathcal{R}[7] \sqsupseteq \textsf{enter}^\sharp\,(\mathcal{R}[8])$$

$$\mathcal{R}[9] \sqsupseteq \textsf{combine}^\sharp\,(\mathcal{R}[8], \mathcal{R}[10])$$

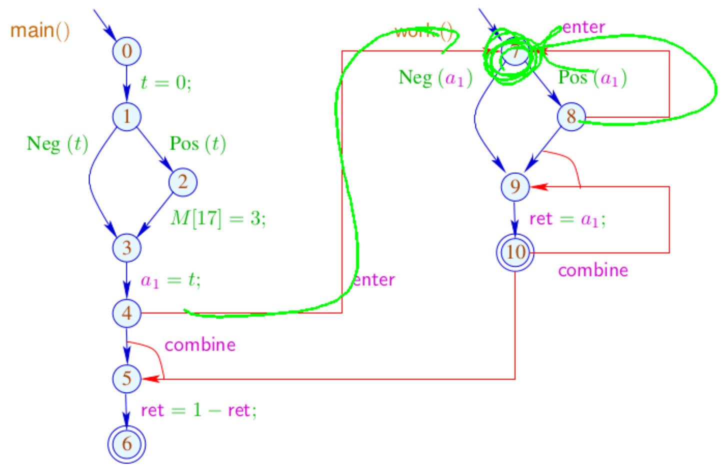
Warning:

The resulting super-graph contains obviously impossible paths ...

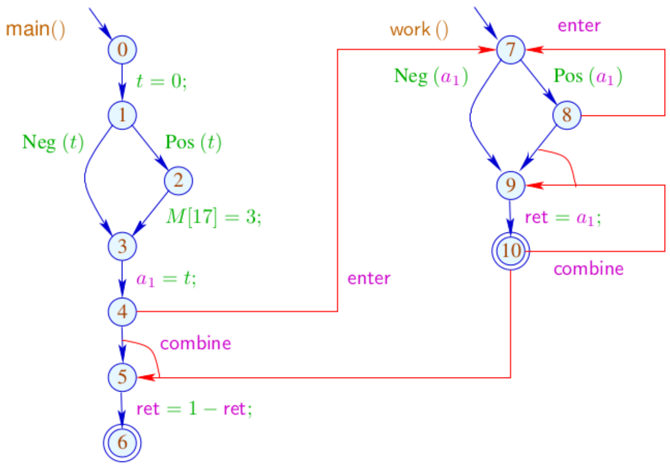
... in the Example this is:



... in the Example this is:



... in the Example this is:



3 Exploiting Hardware Features

Question: How can we optimally use:

... Registers

... Pipelines

... Caches

... Processors ???

3 Exploiting Hardware Features

Question: How can we optimally use:

... Registers

... Pipelines

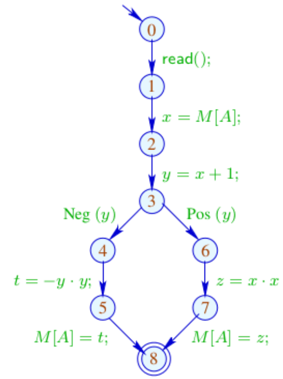
... Caches

... Processors ???

3.1 Registers

Example:

```
read();
x = M[A];
y = x + 1;
if (y) {
        z = x · x;
        M[A] = z;
} else {
        t = −y · y;
        M[A] = t;
}
```



The program uses 5 variables ...

Problem:

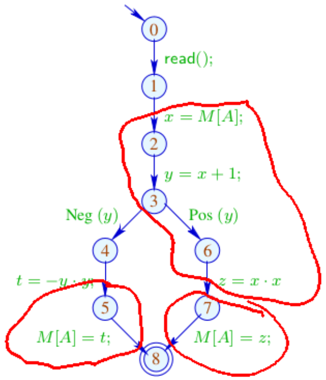
What if the program uses more variables than there are registers :-(

Idea:

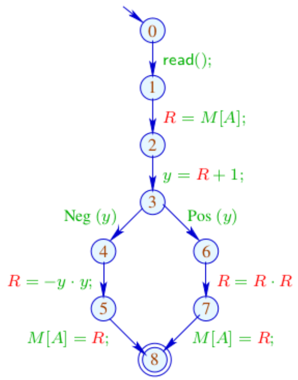
Use one register for several variables :-)

In the example, e.g., one for x, t, z ...


```
read();
x = M[A];
y = x + 1;
if (y) {
        z = x · x;
        M[A] = z;
} else {
        t = −y · y;
        M[A] = t;
}
```




```
read();
R = M[A];
y = R + 1;
if (y) {
        R = R · R;
        M[A] = R;
} else {
        R = −y · y;
        M[A] = R;
}
```



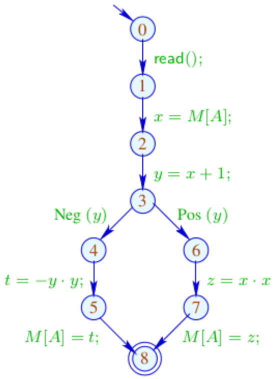
Warning:

This is only possible if the live ranges do not overlap :-)

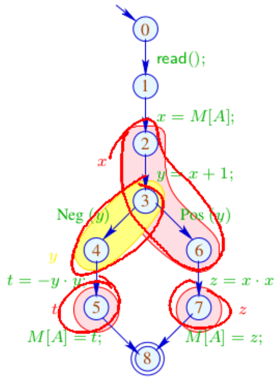
The (true) live range of x is defined by:

$$\mathcal{L}[x] \;=\; \{u \mid x \in \mathcal{L}[u]\}$$

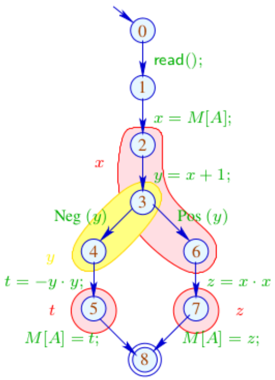
... in the Example:



	$\mathcal{L}$
8	$\emptyset$
7	$\{A, z\}$
6	$\{A, x\}$
5	$\{A, t\}$
4	$\{A, y\}$
3	$\{A, x, y\}$
2	$\{A, x\}$
1	$\{A\}$
0	$\emptyset$



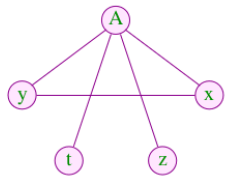
	$\mathcal{L}$
8	$\emptyset$
7	$\{A, z\}$
6	$\{A, x\}$
5	$\{A, t\}$
4	$\{A, y\}$
3	$\{A, x, y\}$
2	$\{A, x\}$
1	$\{A\}$
0	$\{A\}$



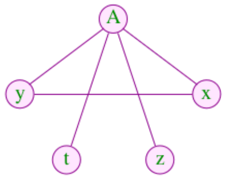
Live Ranges:

A	$\{0, \ldots, 7\}$
x	$\{2, 3, 6\}$
y	$\{2, 4\}$
t	$\{5\}$
z	$\{7\}$

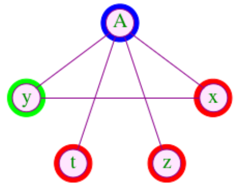
Variables which are not connected with an edge can be assigned to the same register :-)



Variables which are not connected with an edge can be assigned to the same register :-)

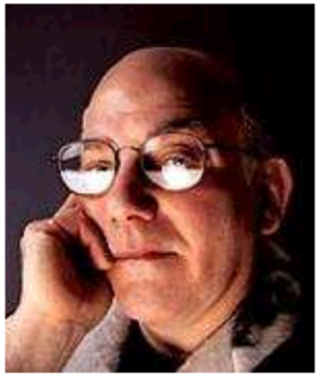


Variables which are not connected with an edge can be assigned to the same register :-)

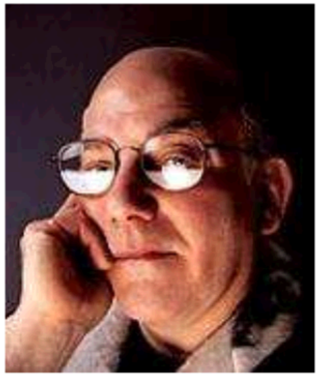


Color == Register

Sviatoslav Sergeevich Lavrov,
Russian Academy of Sciences (1962)



Gregory J. Chaitin, University of Maine (1981)



Gregory J. Chaitin, University of Maine (1981)

Abstract Problem:

Given: Undirected Graph (V, E) .

Wanted: Minimal coloring, i.e., mapping $c : V \to \mathbb{N}$ mit

(1) $c(u) \neq c(v)$ for $\{u, v\} \in E$;

(2) $\bigsqcup\{c(u) \mid u \in V\}$ minimal!

- In the example, 3 colors suffice :-) But:
- In general, the minimal coloring is not unique :-(
- It is NP-complete to determine whether there is a coloring with at most k colors :-((

$\Longrightarrow$

We must rely on heuristics or special cases :-)

Greedy Heuristics:

- Start somewhere with color 1;
- Next choose the smallest color which is different from the colors of all already colored neighbors;
- If a node is colored, color all neighbors which not yet have colors;
- Deal with one component after the other ...

... more concretely:

forall $(v \in V)$ $c[v] = 0$;
forall $(v \in V)$ color (v);

void color (v) {
  if $(c[v] \neq 0)$ return;
  neighbors $= \{u \in V \mid \{u, v\} \in E\}$;
  $c[v] = \bigsqcap\{k > 0 \mid \forall\, u \in$ neighbors $:\ k \neq c(u)\}$;
  forall $(u \in$ neighbors$)$
    if $(c(u) == 0)$ color (u);
}

The new color can be easily determined once the neighbors are sorted according to their colors :-)

Discussion:

- → Essentially, this is a Pre-order DFS :-)
- → In theory, the result may arbitrarily far from the optimum :-(
- → ... in practice, it may not be as bad :-)
- → ... Anecdote: different variants have been patented !!!

Discussion:

- → Essentially, this is a Pre-order DFS :-)
- → In theory, the result may arbitrarily far from the optimum :-(
- → ... in practice, it may not be as bad :-)
- → ... Anecdote: different variants have been patented !!!

602

Discussion:

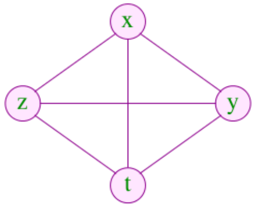
- → Essentially, this is a Pre-order DFS :-)
- → In theory, the result may arbitrarily far from the optimum :-(
- → ... in practice, it may not be as bad :-)
- → ... Anecdote: different variants have been patented !!!

The algorithm works the better the smaller life ranges are ...

Idea: Life Range Splitting

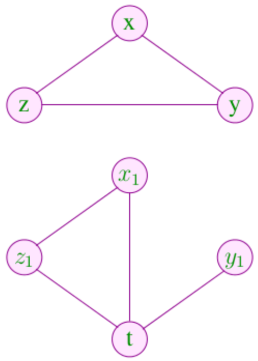
Special Case: Basic Blocks

	$\mathcal{L}$
	x, y, z
$A_1 = x + y;$	x, z
$M[A_1] = z;$	x
$x = x + 1;$	x
$z = M[A_1];$	x, z
$t = M[x];$	x, z, t
$A_2 = x + t;$	x, z, t
$M[A_2] = z;$	x, t
$y = M[x];$	y, t
$M[y] = t;$	



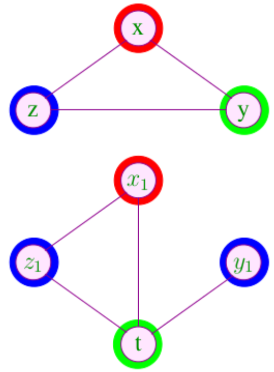
The live ranges of x and z can be split:

	$\mathcal{L}$
	x, y, z
$A_1 = x + y;$	x, z
$M[A_1] = z;$	x
$x_1 = x + 1;$	x_1
$z_1 = M[A_1];$	x_1, z_1
$t = M[x_1];$	x_1, z_1, t
$A_2 = x_1 + t;$	x_1, z_1, t
$M[A_2] = z_1;$	x_1, t
$y_1 = M[x_1];$	y_1, t
$M[y_1] = t;$	

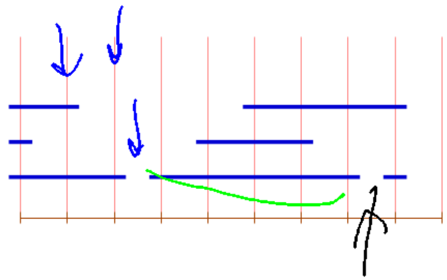


The live ranges of x and z can be split:

	$\mathcal{L}$
	x, y, z
$A_1 = x + y;$	x, z
$M[A_1] = z;$	x
$x_1 = x + 1;$	x_1
$z_1 = M[A_1];$	x_1, z_1
$t = M[x_1];$	x_1, z_1, t
$A_2 = x_1 + t;$	x_1, z_1, t
$M[A_2] = z_1;$	x_1, t
$y_1 = M[x_1];$	y_1, t
$M[y_1] = t;$	



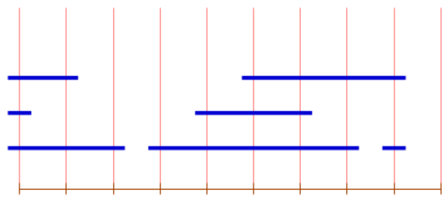
Interference graphs for minimal live ranges on basic blocks are known as interval graphs:



vertex $\equiv$ interval

edge $\equiv$ joint vertex

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

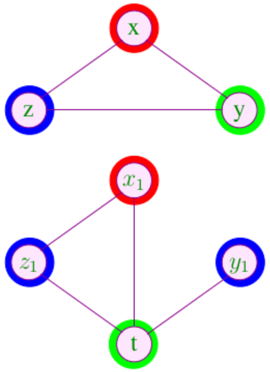


vertex $\equiv$ interval

edge $\equiv$ joint vertex

The live ranges of x and z can be split:

	$\mathcal{L}$
	x, y, z
$A_1 = x + y;$	x, z
$M[A_1] = z;$	x
$x_1 = x + 1;$	x_1
$z_1 = M[A_1];$	x_1, z_1
$t = M[x_1];$	x_1, z_1, t
$A_2 = x_1 + t;$	x_1, z_1, t
$M[A_2] = z_1;$	x_1, t
$y_1 = M[x_1];$	y_1, t
$M[y_1] = t;$	



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 :-))

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, ..., k];
for (i = 0; i < m; i++) {
      init[i] = [];  exit[i] = [];
}
forall (I = [u, v] ∈ Intervals) {
      init[u] = (I :: init[u]);  exit[v] = (I :: exit[v]);
}
for (i = 0; i < m; i++) {
      forall (I ∈ init[i]) {
            color[I] = hd free;  free = tl free;
      }
      forall (I ∈ 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 before-hand !

→ This works only once live ranges have been split.

→ Splitting of live ranges for full programs results programs in static single assignment form ...

SSA