Script generated by TTT

Title: Seidl: Programmoptimierung (12.12.2012)

Date: Wed Dec 12 08:34:00 CET 2012

Duration: 86:36 min

Pages: 43

Discussion:

- At least copy-constants can be determined interprocedurally.
- For that, we had to ignore conditions and complex assignments :-(
- In the second phase, however, we could have been more precise :-)
- The extra abstractions were necessary for two reasons:
  - (1) The set of occurring transformers $\boxed{\mathbb{M}} \subseteq \boxed{\mathbb{D}} \to \boxed{\mathbb{D}}$ must be finite;
  - (2) The functions $M \in \mathbb{M}$ must be efficiently implementable :-)
- The second condition can, sometimes, be abandoned ...

Observation:

Sharir/Pnueli, Cousot

→ Often, procedures are only called for few distinct abstract arguments.

→ Each procedure need only to be analyzed for these :-)

→ Put up a constraint system:

$$
\begin{array}{llll}
[\![v,a]\!]^\sharp & \sqsupseteq & a & v \quad \text{entry point} \\
[\![v,a]\!]^\sharp & \sqsupseteq & \mathsf{combine}^\sharp([\![u,a]\!]^\sharp, [\![f, \mathsf{enter}^\sharp\, [\![u,a]\!]^\sharp]\!]^\sharp) & (u, f();, v) \quad \text{call} \\
[\![v,a]\!]^\sharp & \sqsupseteq & [\![lab]\!]^\sharp\, [\![u,a]\!]^\sharp & k = (u, lab, v) \quad \text{edge} \\
[\![f,a]\!]^\sharp & \sqsupseteq & [\![stop_f, a]\!]^\sharp & stop_f \quad \text{end point of} \quad f
\end{array}
$$

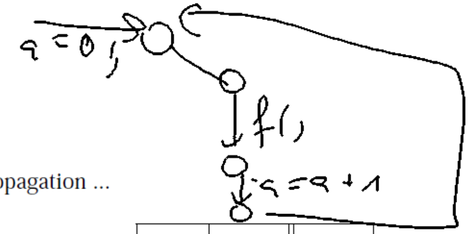
// $[\![v,a]\!]^\sharp$ == value for the argument a .

Discussion:

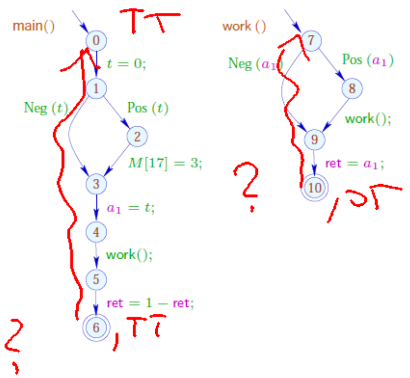
$f, u_f \quad [u_f, a_1], \; [u_f, a_2]$

- This constraint system may be huge :-(
- We do not want to solve it completely!!!
- It is sufficient to compute the correct values for all calls which occur, i.e., which are necessary to determine the value $[\![\mathsf{main}(), a_0]\!]^\sharp$ $\Longrightarrow$ We apply our local fixpoint algorithm :-))
- The fixpoint algo provides us also with the set of actual parameters $a \in \mathbb{D}$ for which procedures are (possibly) called and all abstract values at their program points for each of these calls :-)

... in the Example:



Let us try a full constant propagation ...

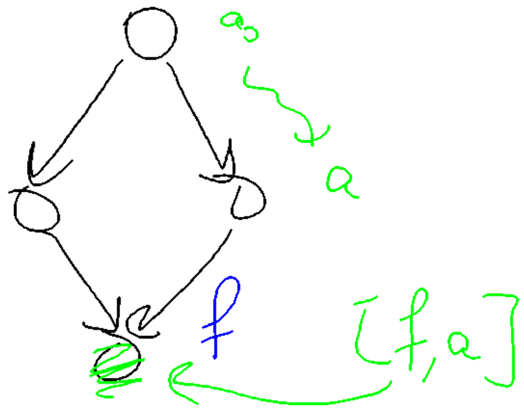


	a_1	ret	a_1	ret
0	⊤	⊤	⊤	⊤
1	⊤	⊤	⊤	⊤
2	⊤	⊤	⊥	
3	⊤	⊤	⊤	⊤
4	⊤	⊤	0	⊤
7	0	⊤	0	⊤
8	0	⊤	⊥	
9	0	⊤	0	⊤
10	0	⊤	0	0
5	⊤	⊤	0	0
main()	⊤	⊤	0	1

Classic fp iteration

Discussion:

- In the Example, the analysis terminates quickly :-)
- If $\mathbb{D}$ has finite height, the analysis terminates if each procedure is only analyzed for finitely many arguments :-))
- Analogous analysis algorithms have proved very effective for the analysis of Prolog :-)
- Together with a points-to analysis and propagation of negative constant information, this algorithm is the heart of a very successful race analyzer for C with Posix threads :-)



Discussion:

- In the Example, the analysis terminates quickly :-)
- If $\mathbb{D}$ has finite height, the analysis terminates if each procedure is only analyzed for finitely many arguments :-))
- Analogous analysis algorithms have proved very effective for the analysis of Prolog :-)
- Together with a points-to analysis and propagation of negative constant information, this algorithm is the heart of a very successful race analyzer for C with Posix threads :-)

(2) The Call-String Approach:

Idea:

→ Compute the set of all reachable call stacks!

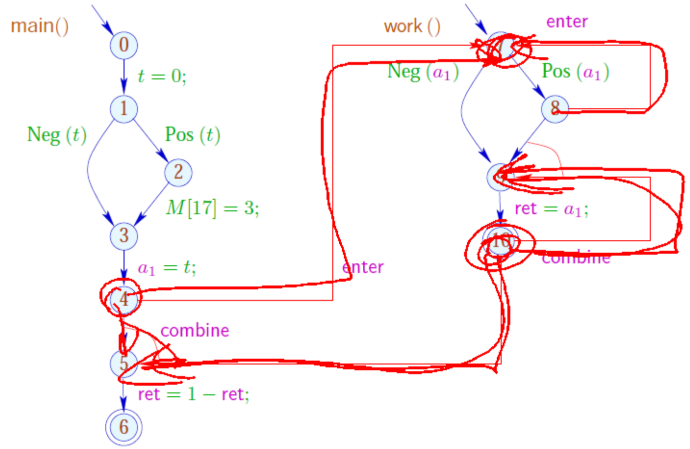
→ In general, this is infinite :-(

→ Only treat stacks up to a fixed depth d precisely! From longer stacks, we only keep the upper prefix of length d :-)

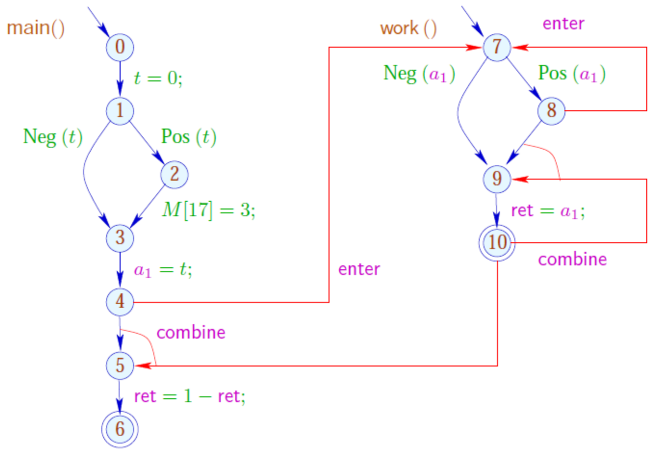
→ Important special case: $d = 0$.

$\Longrightarrow$ Just track the current stack frame ...

... in the Example:



... in the Example:



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

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

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

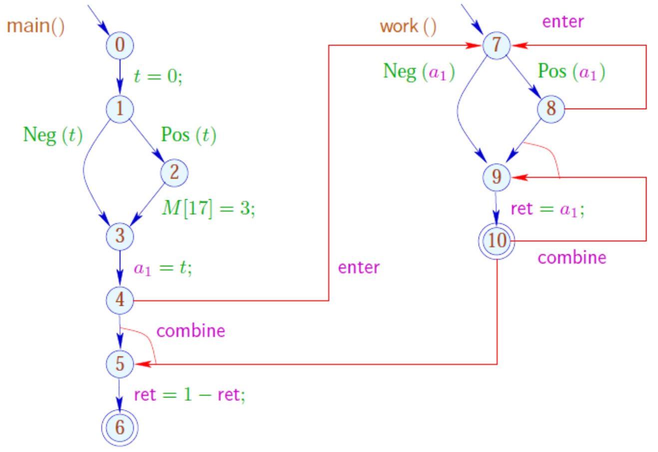
$$\mathcal{R}[7] \sqsupseteq \text{enter}^\sharp(\mathcal{R}[8])$$

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

Warning:

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

... in the Example:



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

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

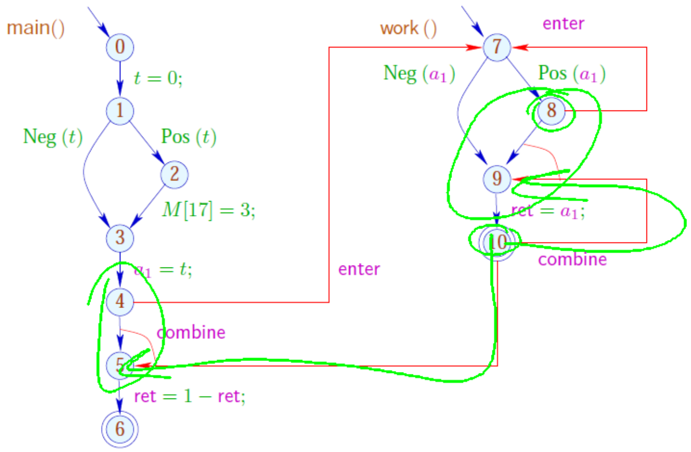
$$\mathcal{R}[7] \sqsupseteq \text{enter}^\sharp\,(\mathcal{R}[4])$$
$$\mathcal{R}[7] \sqsupseteq \text{enter}^\sharp\,(\mathcal{R}[8])$$

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

Warning:

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

... in the Example:



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

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

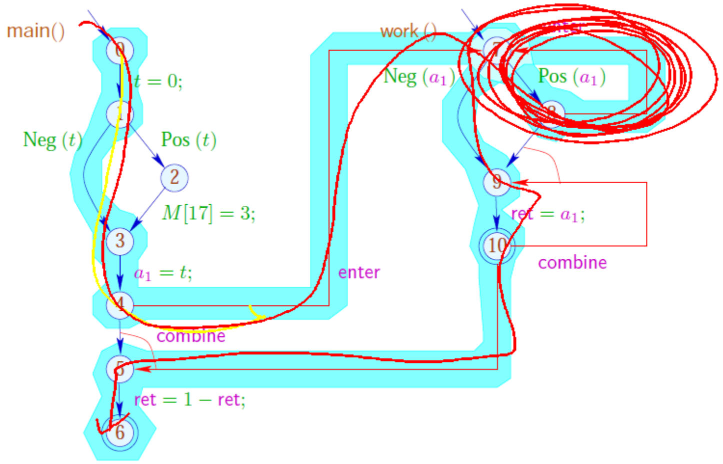
$$\mathcal{R}[7] \sqsupseteq \text{enter}^\sharp\,(\mathcal{R}[4])$$
$$\mathcal{R}[7] \sqsupseteq \text{enter}^\sharp\,(\mathcal{R}[8])$$

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

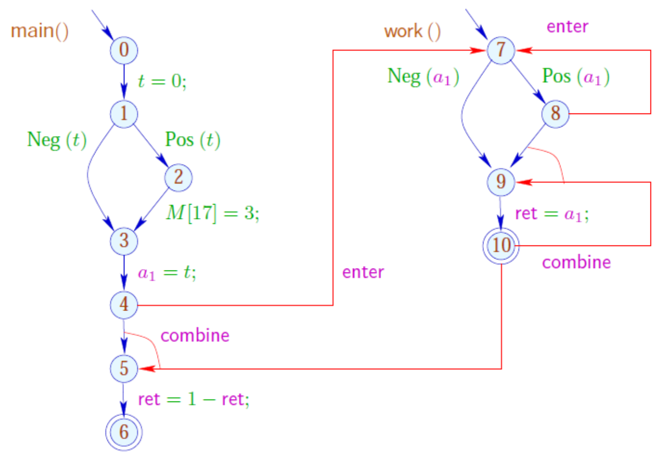
Warning:

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

... in the Example this is:



... in the Example this is:



Note:

→ In the example, we find the same results: more paths render the results less precise.

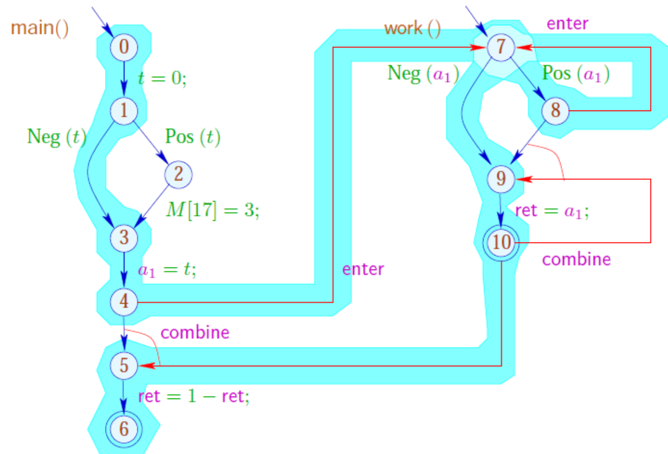
In particular, we provide for each procedure the result just for one (possibly very boring) argument :-(

→ The analysis terminates — whenever $\mathbb{D}$ has no infinite strictly ascending chains :-)

→ The correctness is easily shown w.r.t. the operational semantics with call stacks.

→ For the correctness of the functional approach, the semantics with computation forests is better suited :-)

... in the Example this is:



3 Exploiting Hardware Features

Question: How can we optimally use:

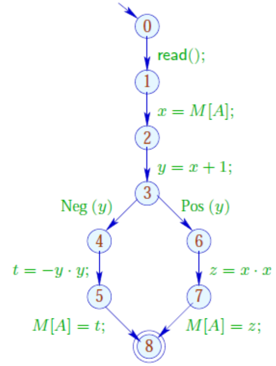
... Registers
... Pipelines
... Caches
... ~~Processors ???~~

GPUs

3.1 Registers

Example:

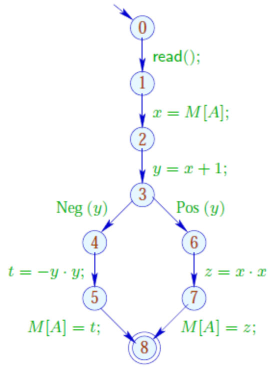
read();
$x = M[A];$
$y = x + 1;$
if (y) {
    $z = x \cdot x;$
    $M[A] = z;$
} else {
    $t = -y \cdot y;$
    $M[A] = t;$
}



3.1 Registers

Example:

read();
$x = M[A];$
$y = x + 1;$
if (y) {
    $z = x \cdot x;$
    $M[A] = z;$
} else {
    $t = -y \cdot y;$
    $M[A] = t;$
}



The program uses 5 variables ...

Problem:

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

Idea:

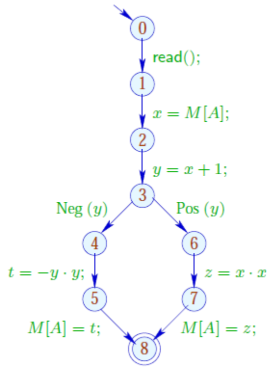
Use one register for several variables :-)

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

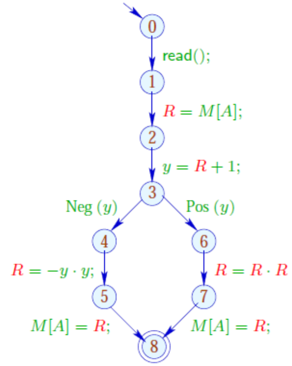
3.1 Registers

Example:

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




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



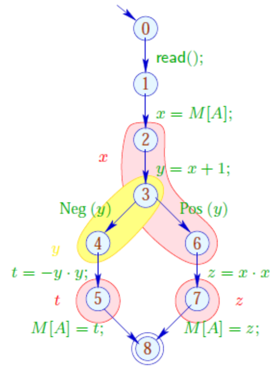
Warning:

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:



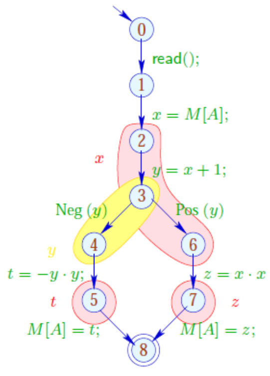
	$\mathcal{L}$
8	$\emptyset$
7	$\{A, z\}$
6	$\{A, x\}$
5	$\{A, t\}$
4	$\{A, y\}$
3	$\{A, x, y\}$
2	$\{A, x\}$
1	$\{A\}$
0	$\{A\}$

In order to determine sets of compatible variables, we construct the Interference Graph $I = (Vars, E_I)$ where:

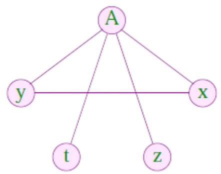
$$E_I = \{\{x, y\} \mid x \neq y, \mathcal{L}[x] \cap \mathcal{L}[y] \neq \emptyset\}$$

E_I has an edge for $x \neq y$ iff x, y are jointly live at some program point :-)

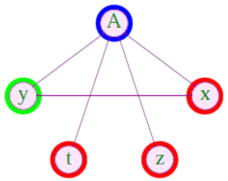
... in the Example:



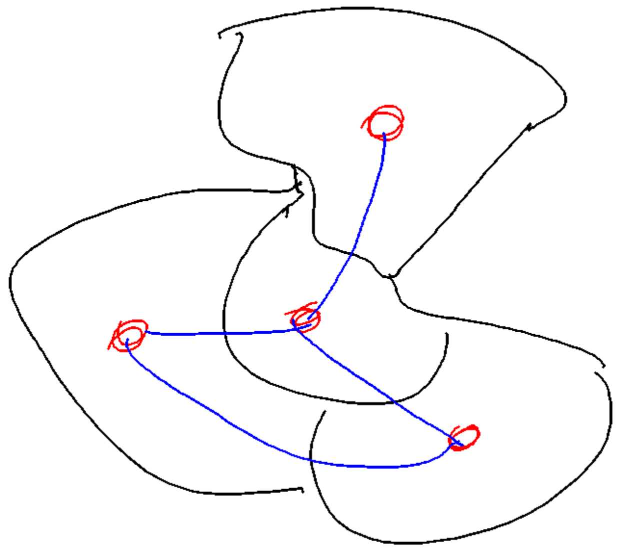
Interference Graph:



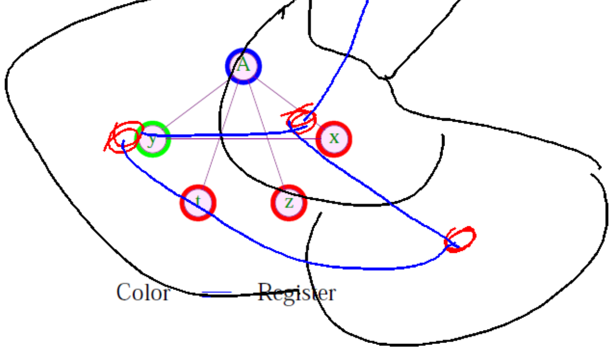
Variables which are not connected with an edge can be assigned to the same register :-)



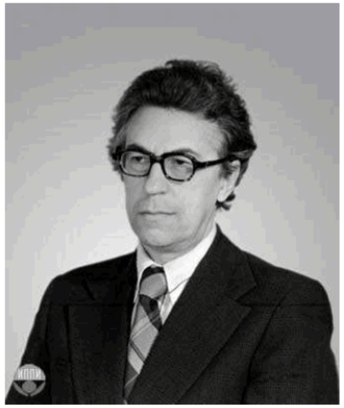
Color $=$ Register



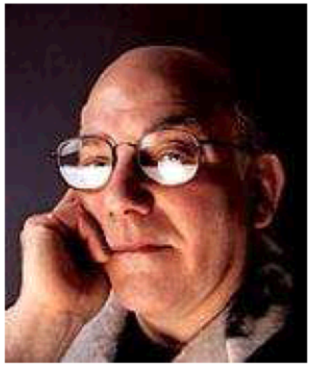
Variables which are not connected with an edge can be assigned to the same register :-)



Color — Register



Sviatoslav Sergeevich Lavrov,
Russian Academy of Sciences (1962)



Gregory J. Chaitin, University of Maine (1981)

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:

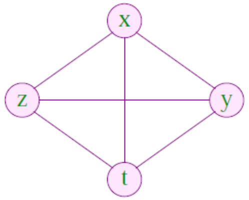
→ Essentially, this is a Pre-order DFS :-)
→ In theory, the result may arbitrarily far from the optimum :-(
→ ... in practice, it may not be as bad :-)
→ ... Anecdote: different variants have been patented !!!

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

Idea: Life Range Splitting

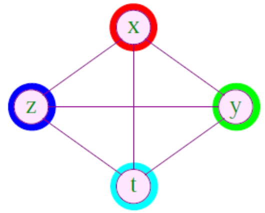
Special Case: Basic Blocks

	$\mathcal{L}$
	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
$M[y] = t;$	



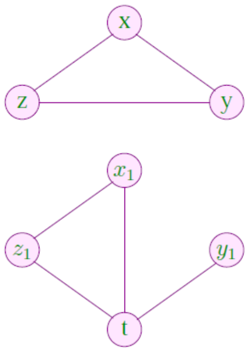
Special Case: Basic Blocks

	$\mathcal{L}$
	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
$M[y_1] = t;$