(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: $d = 0$.
   ⟷ Just track the current stack frame ...
... in the Example:

**main()**

0: \( t = 0; \)

1: \( \text{Neg} (t) \)

2: \( \text{Pos} (t) \)

3: \( M[17] = 3; \)

4: \( a_1 = t; \)

5: \( \text{work}(); \)

6: \( \text{ret} = 1 - \text{ret}; \)

7: \( \text{work}(); \)

8: \( \text{Neg} (a_1) \)

9: \( \text{Pos} (a_1) \)

10: \( \text{ret} = a_1; \)
... in the Example:

```
main()

0
    t = 0;

1
    Neg (t)
    Pos (t)

2
    M[17] = 3;

3
    a_1 = t;

4

5
    ret = 1 - ret;

6

7
    work ()
    enter

8
    Neg (a_1)
    Pos (a_1)

9

10
    ret = a_1;
    combine

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

$$R[5] \sqsubseteq \text{combine}^\#(R[4], R[10])$$

$$R[7] \sqsubseteq \text{enter}^\#(R[4])$$

$$R[7] \sqsubseteq \text{enter}^\#(R[8])$$

$$R[9] \sqsubseteq \text{combine}^\#(R[8], R[10])$$

**Warning:**

The resulting super-graph contains obviously impossible paths ...
... in the Example this is:

main()

0

\( t = 0; \)

1

Neg \( t \)

Pos \( t \)

2

\( M[17] = 3; \)

3

4

\( a_1 = t; \)

combine

5

6

\( \text{return} = 1 - \text{return}; \)

work()

0

1

7

8

9

10

\( \text{Neg} \ (a_1) \)

\( \text{Pos} \ (a_1) \)

\( \text{return} = a_1; \)

combine

enter

enter

576
... in the Example this is:

main()

0

\[ t = 0; \]

1

Pos (t)

2

Neg (t)

3

\[ M[17] = 3; \]

\[ a_1 = t; \]

4

5

\[ \text{ret} = 1 - \text{ret}; \]

6

7

Pos (a_1)

8

Neg (a_1)

9

10

\[ \text{ret} = a_1; \]

work ()

combine

enter

combine

enter
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 :(-)
3 Exploiting Hardware Features

**Question:** How can we optimally use:

... Registers
... Pipelines
... Caches
... Processors ??
3.1 Registers

Example:

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

Diagram:

```
0 → read()
   ↓
1 → read()
   ↓
2 → x = M[A];
   ↓
3 → y = x + 1;
   ↓
4 → Neg (y)
   ↓
5 → t = -y \cdot y;
   ↓
6 → Pos (y)
   ↓
7 → t = -y \cdot y;
   ↓
8 → z = x \cdot x
   ↓
0 → read()
   ↓
1 → x = M[A];
   ↓
2 → y = x + 1;
   ↓
3 → Neg (y)
   ↓
4 → t = -y \cdot y;
   ↓
5 → Pos (y)
   ↓
6 → t = -y \cdot y;
   ↓
7 → z = x \cdot x
   ↓
8 → M[A] = t;
   ↓
0 → read()
   ↓
1 → x = M[A];
   ↓
2 → y = x + 1;
   ↓
3 → Neg (y)
   ↓
4 → t = -y \cdot y;
   ↓
5 → Pos (y)
   ↓
6 → t = -y \cdot y;
   ↓
7 → z = x \cdot x
   ↓
8 → M[A] = z;
   ↓
0 → read()
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();

x = M[A];

y = x + 1;

Neg(y) Pos(y)

z = x · x

t = −y · y;

M[A] = t;

M[A] = z;
read();
\( R = M[A] \);
\( y = R + 1; \)
if \( (y) \) {
  \( R = R \cdot R; \)
  \( M[A] = R; \)
} else {
  \( R = -y \cdot y; \)
  \( M[A] = R; \)
}
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:
\begin{align*}
\text{read();} \\
x &= M[A]; \\
y &= x + 1; \\
\text{Neg}(y) &\quad \text{Pos}(y) \\
t &= -y \cdot y; \\
z &= x \cdot x \\
M[A] &= t; \\
M[A] &= z;
\end{align*}

\begin{tabular}{|c|c|}
\hline
\text{L} & \text{ } \\
\hline
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 \\
\hline
\end{tabular}
read();

\[ x = M[A]; \]

\[ y = x + 1; \]

\[ z = x \cdot x \]

\[ t = -y \cdot y; \]

\[ M[A] = t; \]

\[ M[A] = z; \]

\[ \text{Neg}(y) \]

\[ \text{Pos}(y) \]

\begin{array}{|c|}
\hline
| \ell | \{ | \ predator | \} |
\hline
| 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 |
\hline
\end{array}
read();

\[ x = M[A]; \]

\[ y = x + 1; \]

\[ z = x \cdot x; \]

\[ t = -y \cdot y; \]

\[ M[A] = t; \]

\[ M[A] = z; \]

**Live Ranges:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>( {1, \ldots, 7} )</td>
</tr>
<tr>
<td>( x )</td>
<td>( {2, 3, 6} )</td>
</tr>
<tr>
<td>( y )</td>
<td>( {2, 4} )</td>
</tr>
<tr>
<td>( t )</td>
<td>( {5} )</td>
</tr>
<tr>
<td>( z )</td>
<td>( {7} )</td>
</tr>
</tbody>
</table>
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:
read();

\[ x = M[A]; \]
\[ y = x + 1; \]
\[ \text{Neg}(y) \]
\[ \text{Pos}(y) \]
\[ t = -y \cdot y; \]
\[ z = x \cdot x \]
\[ M[A] = t; \]
\[ M[A] = z; \]

Interference Graph:

\( A \)
\( y \)
\( x \)
\( t \)
\( z \)
Variables which are *not* connected with an edge can be assigned to the same register  :-)

![Diagram with nodes and edges]

- A
- y
- x
- t
- z
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)
Abstract Problem:

Given: Undirected Graph \((V, E)\).

Wanted: Minimal coloring, i.e., mapping \(c : V \rightarrow \mathbb{N}\) mit

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

\(2\) \(\cup\{c(u) \mid u \in V\}\) 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 :-(

\[\Rightarrow\]

We must rely on heuristics or special cases :-)

594
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 \((v)\);

```c
void color (v) {
    if \((c[v] \neq 0)\) return;
    neighbors = \{u \in V | \{u, v\} \in E\};
    c[v] = \cap \{k > 0 | \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 :-)

→ ... Warning: differen 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 :-)
→ ... Warning: differsen variants have been patented !!!

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

Idea: Life Range Splitting
Special Case:

Basic Blocks

\[ A_1 = x + y; \]
\[ M[A_1] = z; \]
\[ x = x + 1; \]
\[ z = M[A_1]; \]
\[ t = M[x]; \]
\[ A_2 = x + t; \]
\[ M[A_2] = z; \]
\[ y = M[x]; \]
\[ M[y] = t; \]

\[ \mathcal{L} \]

\[ x, y, z \]
\[ x, z \]
\[ x \]
\[ x, z \]
\[ x, z, t \]
\[ x, z, t \]
\[ x, t \]
\[ y, t \]
Special Case:  Basic Blocks

<table>
<thead>
<tr>
<th>Equation</th>
<th>( \mathcal{L} )</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></td>
</tr>
</tbody>
</table>
The live ranges of $x$ and $z$ can be split:

<table>
<thead>
<tr>
<th></th>
<th>$\mathcal{L}$</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>$x_1 = x + 1$;</td>
<td>$x_1$</td>
</tr>
<tr>
<td>$z_1 = M[A_1]$;</td>
<td>$x_1, z_1$</td>
</tr>
<tr>
<td>$t = M[x_1]$;</td>
<td>$x_1, z_1, t$</td>
</tr>
<tr>
<td>$A_2 = x_1 + t$;</td>
<td>$x_1, z_1, t$</td>
</tr>
<tr>
<td>$M[A_2] = z_1$;</td>
<td>$x_1, t$</td>
</tr>
<tr>
<td>$y_1 = M[x_1]$;</td>
<td>$y_1, t$</td>
</tr>
<tr>
<td>$M[y_1] = t$;</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram](image-url)
The live ranges of $x$ and $z$ can be split:

<table>
<thead>
<tr>
<th>$A_1 = x + y$;</th>
<th>$\mathcal{L}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M[A_1] = z$;</td>
<td>$x, y, z$</td>
</tr>
<tr>
<td>$x_1 = x + 1$;</td>
<td>$x, z$</td>
</tr>
<tr>
<td>$z_1 = M[A_1]$;</td>
<td>$x$</td>
</tr>
<tr>
<td>$t = M[x_1]$;</td>
<td>$x_1$</td>
</tr>
<tr>
<td>$A_2 = x_1 + t$;</td>
<td>$x_1, z_1$</td>
</tr>
<tr>
<td>$M[A_2] = z_1$;</td>
<td>$x_1, z_1, t$</td>
</tr>
<tr>
<td>$y_1 = M[x_1]$;</td>
<td>$x_1, t$</td>
</tr>
<tr>
<td>$M[y_1] = t$;</td>
<td>$y_1, t$</td>
</tr>
</tbody>
</table>
Interference graphs for minimal live ranges on basic blocks are known as **interval graphs**:
The **covering number** of a vertex is given by the number of incident intervals.

**Theorem:**

maximal covering number

\[ \text{size of the maximal clique} \]

\[ \text{minimally necessary number of colors} \quad :) \]

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:
\[ \text{free} = [1, \ldots, k]; \]

\[
\text{for} \quad (i = 0; i < m; i++) \quad \{ \\
\quad \text{init}[i] = []; \quad \text{exit}[i] = []; \\
\}
\]

\[
\text{forall} \quad (I = [u, v] \in \text{Intervals}) \quad \{ \\
\quad \text{init}[u] = (I :: \text{init}[u]); \quad \text{exit}[i] = (I :: \text{exit}[v]); \\
\}
\]

\[
\text{for} \quad (i = 0; i < m; i++) \quad \{ \\
\quad \text{forall} \quad (I \in \text{init}[i]) \quad \{ \\
\quad \text{color}[I] = \text{hd free}; \quad \text{free} = \text{tl free}; \\
\quad \}
\quad \text{forall} \quad (I \in \text{exit}[i]) \quad \text{free} = \text{color}[I] :: \text{free}; \\
\}
\]
Discussion:

→ For basic blocks we have succeeded to derive an optimal register allocation :-)

→ The same problem for simple loops (circular arc graphs) is already NP-hard :-(

→ For arbitrary programs, we thus may apply some heuristics for graph coloring ...

→ which always works better the less live ranges overlap :-)

→ 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.