Helmut Seidl

Program Optimization

TU München

Winter 2008/09

Organization

Dates:	Lecture:	Monday, 12-14
		Tuesday, 12-14
	Tutorials:	Friday, 12-14
		Vesal Vojdani: `vojdanig@in.tum.de`
	Material:	slides, recording :-)
		simulator environment

Grades:

- Bonus for homeworks
- written exam

Proposed Content:

1. Avoiding redundant computations
   - → available expressions
   - → constant propagation/array-bound checks
   - → code motion
2. Replacing expensive with cheaper computations
   - → peep hole optimization
   - → inlining
   - → reduction of strength

     ...

3. Exploiting Hardware

→ Instruction selection

→ Register allocation

→ Scheduling

→ Memory management

0 Introduction

Observation 1: Intuitive programs often are inefficient.

Example:

```
void swap (int i, int j) {
    int t;
    if (a[i] > a[j]) {
        t = a[j];
        a[j] = a[i];
        a[i] = t;
       }
    }
```

Inefficiencies:

- Addresses `a[i]`, `a[j]` are computed three times :-(
- Values `a[i]`, `a[j]` are loaded twice :-(

Improvement:

- Use a pointer to traverse the array `a`;
- store the values of `a[i]`, `a[j]`!

```
void swap (int *p, int *q) {
     int t, ai, aj;
     ai = *p; aj = *q;
     if (ai > aj) {
         t = aj;
         *q = ai;
         *p = t;    // t can also be
        }           // eliminated!
     }
```

Observation 2:

Higher programming languages (even C :-) abstract from hardware and efficiency.

It is up to the compiler to adapt intuitively written program to hardware.

Examples:

... Filling of delay slots;

... Utilization of special instructions;

... Re-organization of memory accesses for better cache behavior;

... Removal of (useless) overflow/range checks.

Observation 3:

Programm-Improvements need not always be correct :-(

Example:

```
y = f() + f();
```

$\Longrightarrow$

```
y = 2 * f();
```

Idea: Save second evaluation of `f()` ...

Observation 3:

Programm-Improvements need not always be correct :-(

Example:

```
y = f() + f();     ⟹     y = 2 * f();
```

Idea: Save the second evaluation of `f()` ???

Problem: The second evaluation may return a result different from the first; (e.g., because `f()` reads from the input :-)

Consequences:

$\Longrightarrow$ Optimizations have assumptions.

$\Longrightarrow$ The assumption must be:

- formalized,
- checked :-)

$\Longrightarrow$ It must be proven that the optimization is correct, i.e., preserves the semantics !!!

Observation 4:

Optimization techniques depend on the programming language:

→ which inefficiencies occur;

→ how analyzable programs are;

→ how difficult/impossible it is to prove correctness ...

Example: Java

Unavoidable Inefficiencies:

- $*$ Array-bound checks;
- $*$ Dynamic method invocation;
- $*$ Bombastic object organization ...

Analyzability:

- $+$ no pointer arithmetic;
- $+$ no pointer into the stack;
- $-$ dynamic class loading;
- $-$ reflection, exceptions, threads, ...

Correctness proofs:

+ more or less well-defined semantics;

− features, features, features;

− libraries with changing behavior ...

... in this course:

a simple imperative programming language with:

- variables // registers
- $R = e;$ // assignments
- $R = M[e];$ // loads
- $M[e_1] = e_2;$ // stores
- if (e) s_1 else s_2 // conditional branching
- goto $L;$ // no loops :-)

Note:

- For the beginning, we omit procedures :-)
- External procedures are taken into account through a statement $f()$ for an unknown procedure f.

  $\Longrightarrow$ intra-procedural

  $\Longrightarrow$ kind of an intermediate language in which (almost) everything can be translated.

Example: `swap()`

```
0 :   $A_1 = A_0 + 1 * i;$          //   $A_0 == \&a$
1 :   $R_1 = M[A_1];$               //   $R_1 == a[i]$
2 :   $A_2 = A_0 + 1 * j;$
3 :   $R_2 = M[A_2];$               //   $R_2 == a[j]$
4 :   if $(R_1 > R_2)$ $\{$
5 :       $A_3 = A_0 + 1 * j;$
6 :       $t = M[A_3];$
7 :       $A_4 = A_0 + 1 * j;$
8 :       $A_5 = A_0 + 1 * i;$
9 :       $R_3 = M[A_5];$
10 :      $M[A_4] = R_3;$
11 :      $A_6 = A_0 + 1 * i;$
12 :      $M[A_6] = t;$
      $\}$
```

Optimization 1: $1 * R \Longrightarrow R$

Optimization 2: Reuse of subexpressions

$$A_1 == A_5 == A_6$$
$$A_2 == A_3 == A_4$$

$$M[A_1] == M[A_5]$$
$$M[A_2] == M[A_3]$$

$$R_1 == R_3$$

By this, we obtain:

$$\begin{array}{lll}
A_1 & = & A_0 + i; \\
R_1 & = & M[A_1]; \\
A_2 & = & A_0 + j; \\
R_2 & = & M[A_2]; \\
\multicolumn{3}{l}{\textsf{if } (R_1 > R_2)\ \{} \\
 & \begin{array}{lll}
t & = & R_2; \\
M[A_2] & = & R_1; \\
M[A_1] & = & t; \\
\}
\end{array} &
\end{array}$$

Optimization 3: Contraction of chains of assignments :-)

Gain:

	before	after
$+$	6	2
$*$	6	0
load	4	2
store	2	2
$>$	1	1
$=$	6	2

1 Removing superfluous computations

1.1 Repeated computations

Idea:

If the same value is computed repeatedly, then

→ store it after the first computation;

→ replace every further computation through a look-up!

$\Longrightarrow$ Availability of expressions

$\Longrightarrow$ Memoization

Problem: Identify repeated computations!

Example:

$$\begin{array}{lrcl} & z & = & 1; \\ & y & = & M[17]; \\ A: & x_1 & = & \boxed{y+z}; \\ & & & \ldots \\ B: & x_2 & = & \boxed{y+z}; \end{array}$$

Note:

B is is a repeated computation of the value of $\boxed{y + z}$, if:

(1) A is always executed before B; and

(2) y and z at B have the same values as at A :-)

$\Longrightarrow$ We need:

$\rightarrow$ an operational semantics :-)

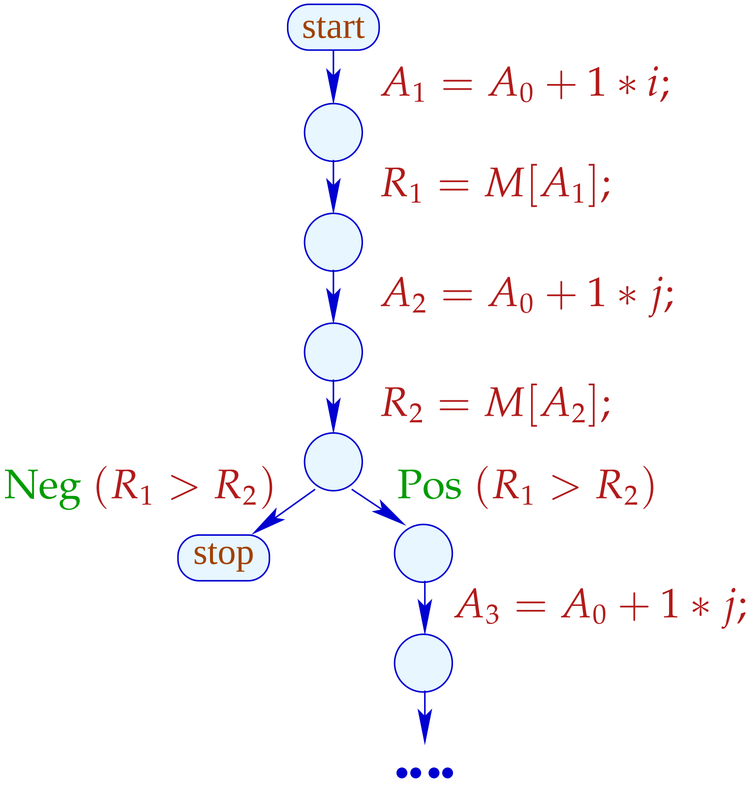
$\rightarrow$ a method which identifies at least some repeated computations ...

Background 1: An Operational Semantics

we choose a small-step operational approach.

Programs are represented as control-flow graphs.

In the example:



Thereby, represent:

vertex	program point
start	programm start
stop	program exit
edge	step of computation

Thereby, represent:

vertex	program point
start	programm start
stop	program exit
edge	step of computation

Edge Labelings:

Test :	Pos (e) or Neg (e)
Assignment :	$R = e;$
Load :	$R = M[e];$
Store :	$M[e_1] = e_2;$
Nop :	$;$

Computations follow paths.

Computations transform the current state

$$s = (\rho, \mu)$$

where:

$\rho : \mathit{Vars} \rightarrow \textbf{int}$	contents of registers
$\mu : \mathbb{N} \rightarrow \textbf{int}$	contents of storage

Every edge $k = (u, \mathit{lab}, v)$ defines a partial transformation

$$[\![k]\!] = [\![\mathit{lab}]\!]$$

of the state:

$$\llbracket ; \rrbracket\,(\rho, \mu) \quad = \quad (\rho, \mu)$$

$$\begin{aligned}
\llbracket \mathrm{Pos}\,(e) \rrbracket\,(\rho, \mu) &= (\rho, \mu) && \text{if } \llbracket e \rrbracket\,\rho \neq 0 \\
\llbracket \mathrm{Neg}\,(e) \rrbracket\,(\rho, \mu) &= (\rho, \mu) && \text{if } \llbracket e \rrbracket\,\rho = 0
\end{aligned}$$

$$
\begin{array}{lcll}
[\![;]\!]\,(\rho,\mu) & = & (\rho,\mu) & \\
\\
[\![\mathrm{Pos}\,(e)]\!]\,(\rho,\mu) & = & (\rho,\mu) & \text{if } [\![e]\!]\,\rho \neq 0 \\
[\![\mathrm{Neg}\,(e)]\!]\,(\rho,\mu) & = & (\rho,\mu) & \text{if } [\![e]\!]\,\rho = 0
\end{array}
$$

// $[\![e]\!]$: evaluation of the expression e, e.g.

// $[\![x+y]\!]\,\{x \mapsto 7, y \mapsto -1\} = 6$

// $[\![!(x == 4)]\!]\,\{x \mapsto 5\} = 1$

$$[\![;]\!]\,(\rho,\mu) \quad = \quad (\rho,\mu)$$

$$\begin{array}{lcll} [\![\mathrm{Pos}\,(e)]\!]\,(\rho,\mu) & = & (\rho,\mu) & \text{if } [\![e]\!]\,\rho \neq 0 \\ [\![\mathrm{Neg}\,(e)]\!]\,(\rho,\mu) & = & (\rho,\mu) & \text{if } [\![e]\!]\,\rho = 0 \end{array}$$

// $[\![e]\!]$: evaluation of the expression e, e.g.

// $[\![x+y]\!]\,\{x \mapsto 7, y \mapsto -1\} = 6$

// $[\![!(x == 4)]\!]\,\{x \mapsto 5\} = 1$

$$[\![R = e;]\!]\,(\rho,\mu) \quad = \quad (\boxed{\rho \oplus \{R \mapsto [\![e]\!]\,\rho\}}, \mu)$$

// where “$\oplus$” modifies a mapping at a given argument

$$
\begin{aligned}
[\![R = M[e];]\!]\,(\rho,\mu) &= (\boxed{\rho \oplus \{R \mapsto \mu([\![e]\!]\,\rho))\}}\,, \mu) \\
[\![M[e_1] = e_2;]\!]\,(\rho,\mu) &= (\rho, \boxed{\mu \oplus \{[\![e_1]\!]\,\rho \mapsto [\![e_2]\!]\,\rho\}}\,)
\end{aligned}
$$

Example:

$[\![x = x + 1;]\!]\,(\{x \mapsto 5\}, \mu) = (\rho, \mu)$ where:

$$
\begin{aligned}
\rho &= \{x \mapsto 5\} \oplus \{x \mapsto [\![x+1]\!]\,\{x \mapsto 5\}\} \\
&= \{x \mapsto 5\} \oplus \{x \mapsto 6\} \\
&= \{x \mapsto 6\}
\end{aligned}
$$

A path $\pi = k_1 k_2 \ldots k_m$ is a computation for the state s if:

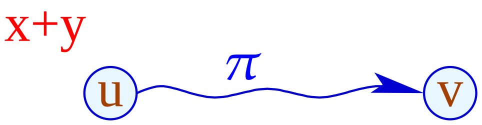
$$s \in \mathit{def}\,(\llbracket k_m \rrbracket \circ \ldots \circ \llbracket k_1 \rrbracket)$$

The result of the computation is:

$$\llbracket \pi \rrbracket\, s = (\llbracket k_m \rrbracket \circ \ldots \circ \llbracket k_1 \rrbracket)\, s$$

Application:

Assume that we have computed the value of red$x + y$ at program point u:



We perform a computation along path π and reach v where we evaluate again $x + y$ ...

Idea:

If x and y have not been modified in π, then evaluation of $x + y$ at v must return the same value as evaluation at u :-)

We can check this property at every edge in π :-}

Idea:

If x and y have not been modified in π, then evaluation of $x + y$ at v must return the same value as evaluation at u :-)

We can check this property at every edge in π :-}

More generally:

Assume that the values of the expressions $A = \{e_1, \ldots, e_r\}$ are available at u.

Idea:

If x and y have not been modified in π, then evaluation of $x + y$ at v must return the same value as evaluation at u :-)

We can check this property at every edge in π :-}

More generally:

Assume that the values of the expressions $A = \{e_1, \ldots, e_r\}$ are available at u.

Every edge k transforms this set into a set $[\![k]\!]^\sharp\, A$ of expressions whose values are available after execution of k ...

... which transformations can be composed to the effect of a path $\pi = k_1 \ldots k_r$:

$$[\![\pi]\!]^\sharp = [\![k_r]\!]^\sharp \circ \ldots \circ [\![k_1]\!]^\sharp$$

... which transformations can be composed to the effect of a path $\pi = k_1 \ldots k_r$:

$$[\![\pi]\!]^\sharp = [\![k_r]\!]^\sharp \circ \ldots \circ [\![k_1]\!]^\sharp$$

The effect $[\![k]\!]^\sharp$ of an edge $k = (u, lab, v)$ only depends on the label lab, i.e., $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$

... which transformations can be composed to the effect of a path $\pi = k_1 \ldots k_r$:

$$[\![\pi]\!]^\sharp = [\![k_r]\!]^\sharp \circ \ldots \circ [\![k_1]\!]^\sharp$$

The effect $[\![k]\!]^\sharp$ of an edge $k = (u, lab, v)$ only depends on the label lab, i.e., $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ where:

$$\begin{array}{lllll}
[\![;]\!]^\sharp\, A & = & A & & \\
[\![Pos(e)]\!]^\sharp\, A & = & [\![Neg(e)]\!]^\sharp\, A & = & A \cup \{e\} \\
[\![x = e;]\!]^\sharp\, A & = & (A \cup \{e\}) \backslash Expr_x & & \text{where} \\
 & & \multicolumn{3}{l}{Expr_x \text{ all expressions which contain } x}
\end{array}$$

$$\begin{array}{lll} [\![x = M[e];]\!]^\sharp\, A & = & (A \cup \{e\}) \backslash Expr_x \\ [\![M[e_1] = e_2;]\!]^\sharp\, A & = & A \cup \{e_1, e_2\} \end{array}$$

$$\begin{array}{lcl} [\![x = M[e];]\!]^\sharp\, A & = & (A \cup \{e\}) \backslash Expr_x \\ [\![M[e_1] = e_2;]\!]^\sharp\, A & = & A \cup \{e_1, e_2\} \end{array}$$

By that, every path can be analyzed :-)

A given program may admit several paths :-(

For any given input, another path may be chosen :-((

$$\begin{array}{lll} [\![x = M[e];]\!]^\sharp\, A & = & (A \cup \{e\})\backslash \mathit{Expr}_x \\ [\![M[e_1] = e_2;]\!]^\sharp\, A & = & A \cup \{e_1, e_2\} \end{array}$$

By that, every path can be analyzed :-)

A given program may admit several paths :-(

For any given input, another path may be chosen :-((

$\Longrightarrow$ We require the set:

$$\mathcal{A}[v] \quad = \quad \bigcap \{ [\![\pi]\!]^\sharp \emptyset \mid \pi : \mathit{start} \rightarrow^* v \}$$

Concretely:

$\longrightarrow$ We consider all paths π which reach v.

$\longrightarrow$ For every path π, we determine the set of expressions which are available along π.

$\longrightarrow$ Initially at program start, nothing is available :-)

$\longrightarrow$ We compute the intersection $\Longrightarrow$ safe information

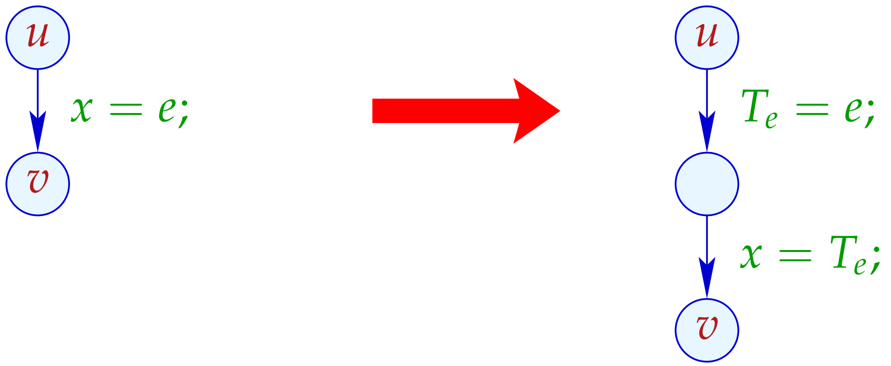
Concretely:

- $\longrightarrow$ We consider all paths π which reach v.
- $\longrightarrow$ For every path π, we determine the set of expressions which are available along π.
- $\longrightarrow$ Initially at program start, nothing is available :-)
- $\longrightarrow$ We compute the intersection $\Longrightarrow$ safe information

How do we exploit this information ???

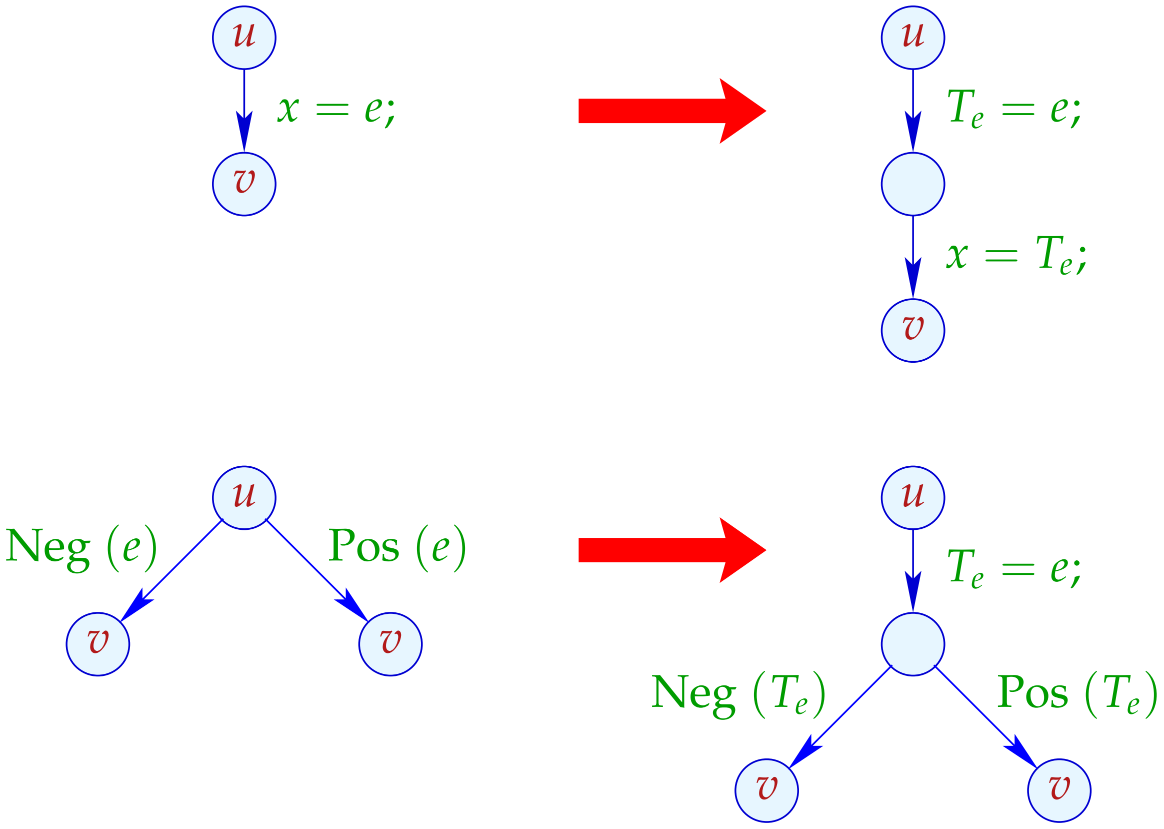
Transformation 1.1:

We provide novel registers T_e as storage for the e:



Transformation 1.1:

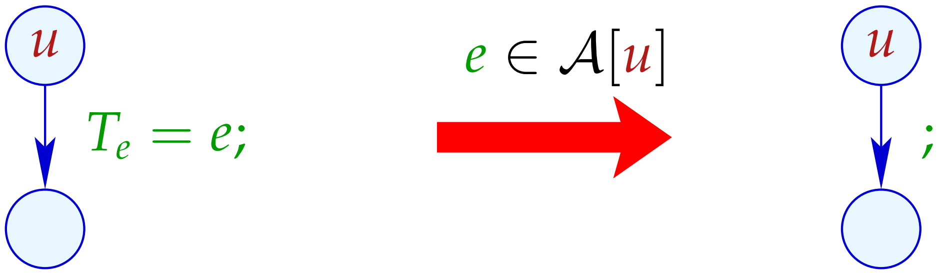
We provide novel registers T_e as storage for the e:



... analogously for $R = M[e];$ and $M[e_1] = e_2;$.

Transformation 1.2:

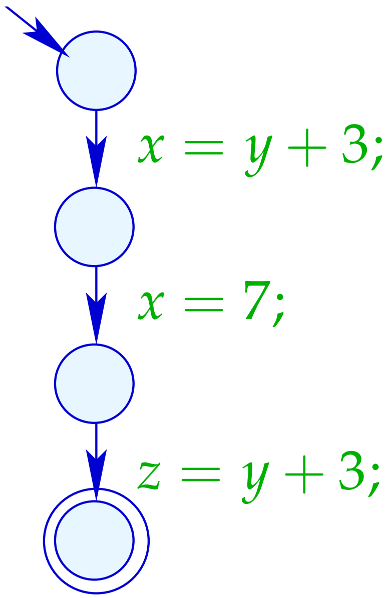
If e is available at program point u, then e need not be re-evaluated:



We replace the assignment with Nop :-)

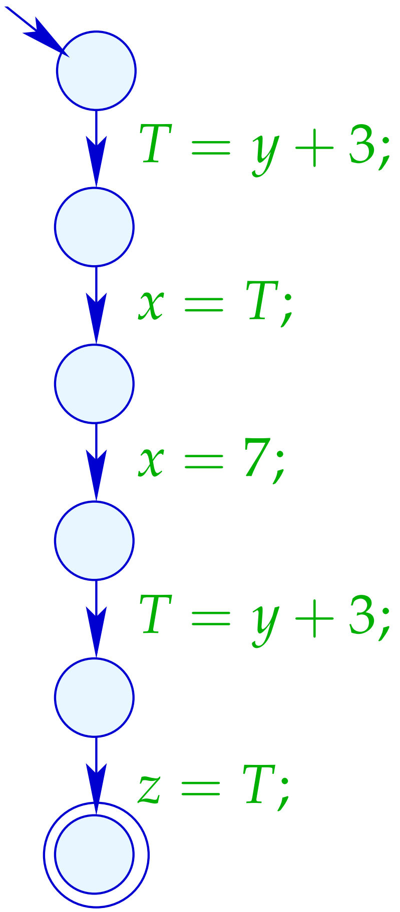
Example:

$$\begin{aligned} x &= y+3; \\ x &= 7; \\ z &= y+3; \end{aligned}$$



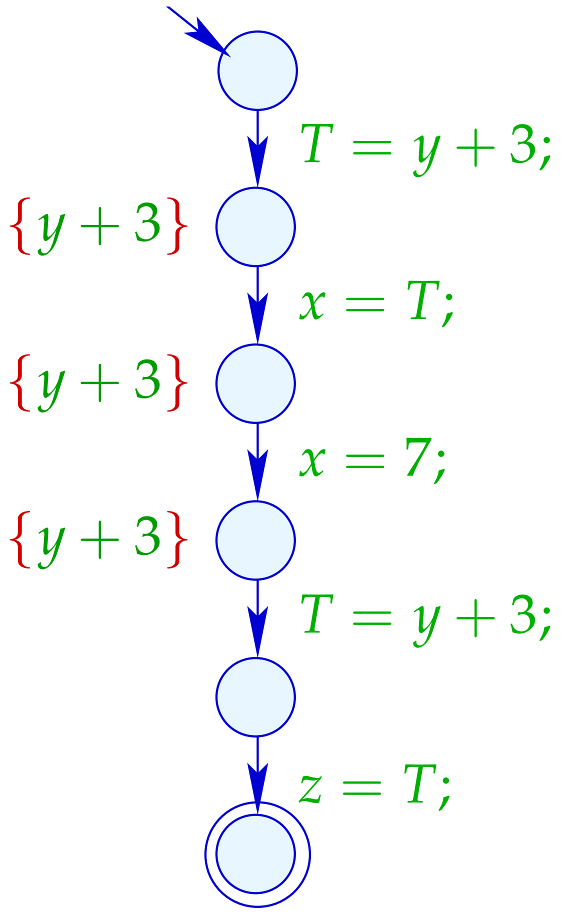
Example:

$$\begin{aligned} x &= y+3; \\ x &= 7; \\ z &= y+3; \end{aligned}$$



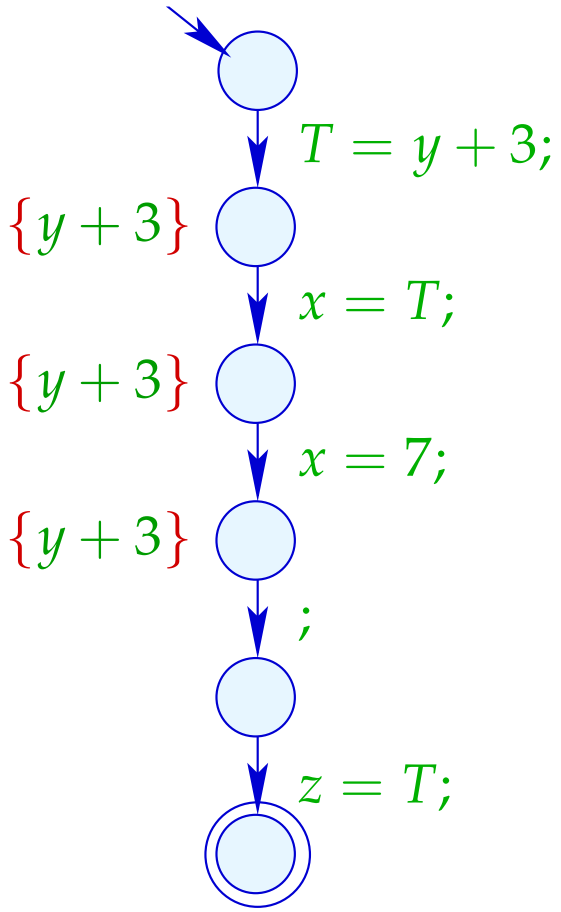
Example:

$$\begin{aligned} x &= y+3; \\ x &= 7; \\ z &= y+3; \end{aligned}$$



Example:

$$\begin{aligned} x &= y+3; \\ x &= 7; \\ z &= y+3; \end{aligned}$$



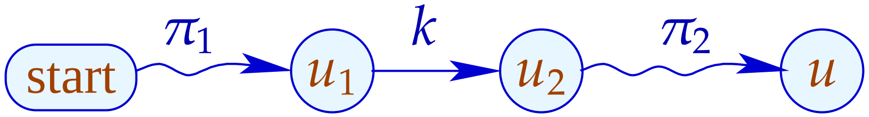
Correctness: (Idea)

Transformation 1.1 preserves the semantics and $\mathcal{A}[u]$ for all program points u :-)

Assume $\pi : \mathit{start} \to^* u$ is the path taken by a computation.

If $e \in \mathcal{A}[u]$, then also $e \in [\![\pi]\!]^\sharp\,\emptyset$.

Therefore, π can be decomposed into:



with the following properties:

- The expression e is evaluated at the edge k;
- The expression e is not removed from the set of available expressions at any edge in π_2, i.e., no variable of e receives a new value :-)

- The expression e is evaluated at the edge k;
- The expression e is not removed from the set of available expressions at any edge in π_2, i.e., no variable of e receives a new value :-)

$$\Longrightarrow$$

The register T_e contains the value of e whenever u is reached :-))

Warning:

Transformation 1.1 is only meaningful for assignments $x = e;$ where:

→ $x \notin Vars(e);$

→ $e \notin Vars;$

→ the evaluation of e is non-trivial :-}

Warning:

Transformation 1.1 is only meaningful for assignments $x = e;$ where:

- $\to$ $x \not\in \mathit{Vars}(e);$
- $\to$ $e \not\in \mathit{Vars};$
- $\to$ the evaluation of e is non-trivial :-}

Which leaves us with the following question ...

Question:

How do we compute $\mathcal{A}[u]$ for every program point u ??

Question:

How can we compute $\mathcal{A}[u]$ for every program point u ??

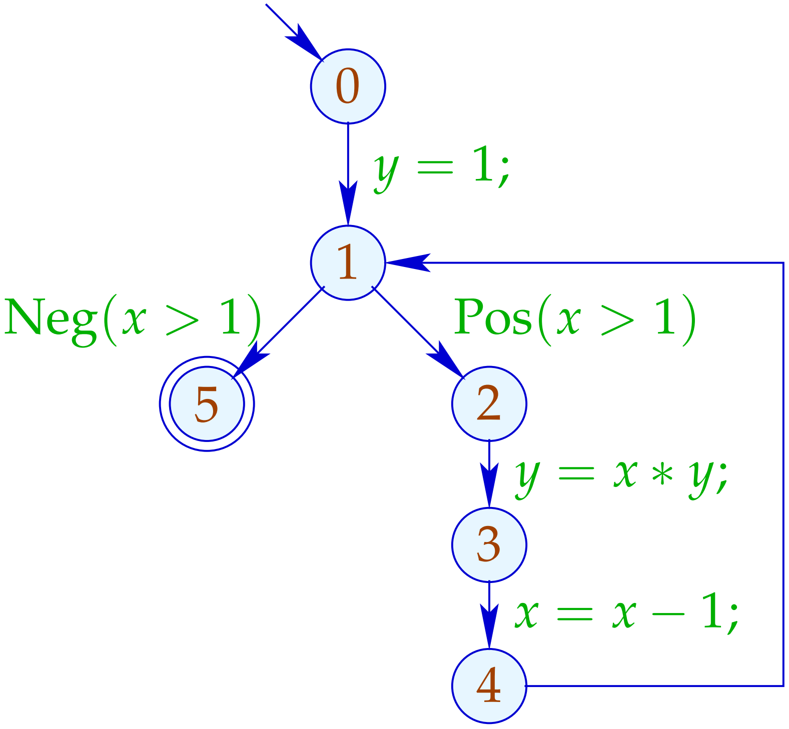
We collect all restrictions to the values of $\mathcal{A}[u]$ into a system of constraints:

$$\begin{array}{lll} \mathcal{A}[start] & \subseteq & \emptyset \\ \mathcal{A}[v] & \subseteq & [\![k]\!]^\sharp\,(\mathcal{A}[u]) \qquad\qquad k = (u,\_,v) \quad \text{edge} \end{array}$$

Wanted:

- a maximally large solution (??)
- an algorithm which computes this :-)

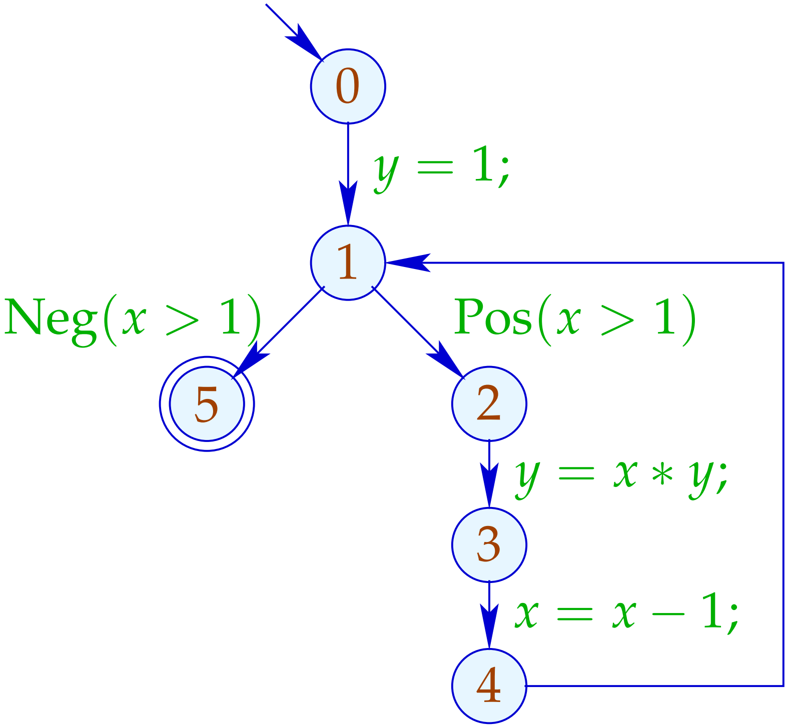
Example:



Wanted:

- a maximally large solution (??)
- an algorithm which computes this :-)

Example:

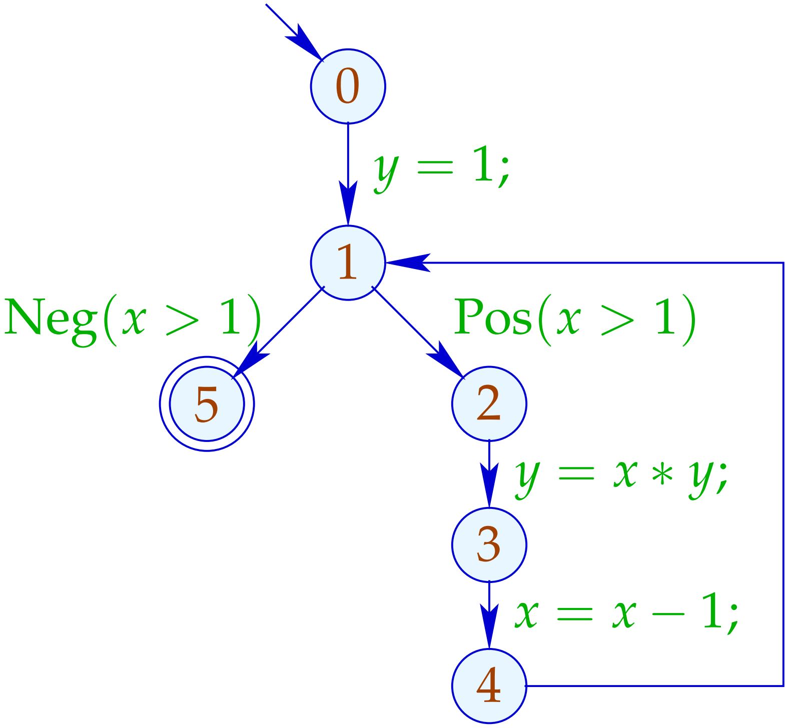


$$\mathcal{A}[0] \quad \subseteq \quad \emptyset$$

Wanted:

- a maximally large solution (??)
- an algorithm which computes this :-)

Example:

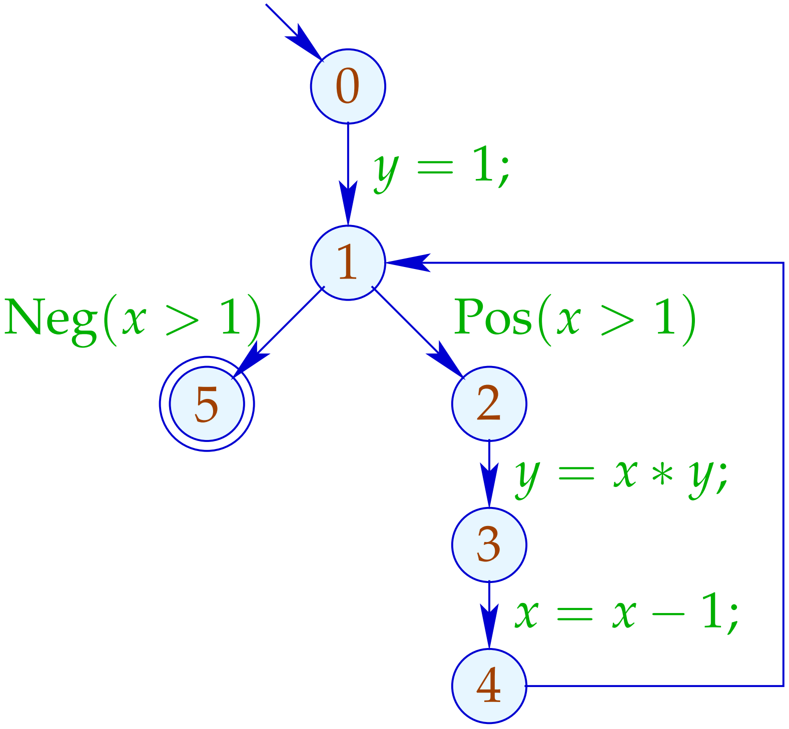


$$\begin{array}{lll} \mathcal{A}[0] & \subseteq & \emptyset \\ \mathcal{A}[1] & \subseteq & (\mathcal{A}[0] \cup \{1\}) \backslash Expr_y \\ \mathcal{A}[1] & \subseteq & \mathcal{A}[4] \end{array}$$

Wanted:

- a maximally large solution (??)
- an algorithm which computes this :-)

Example:

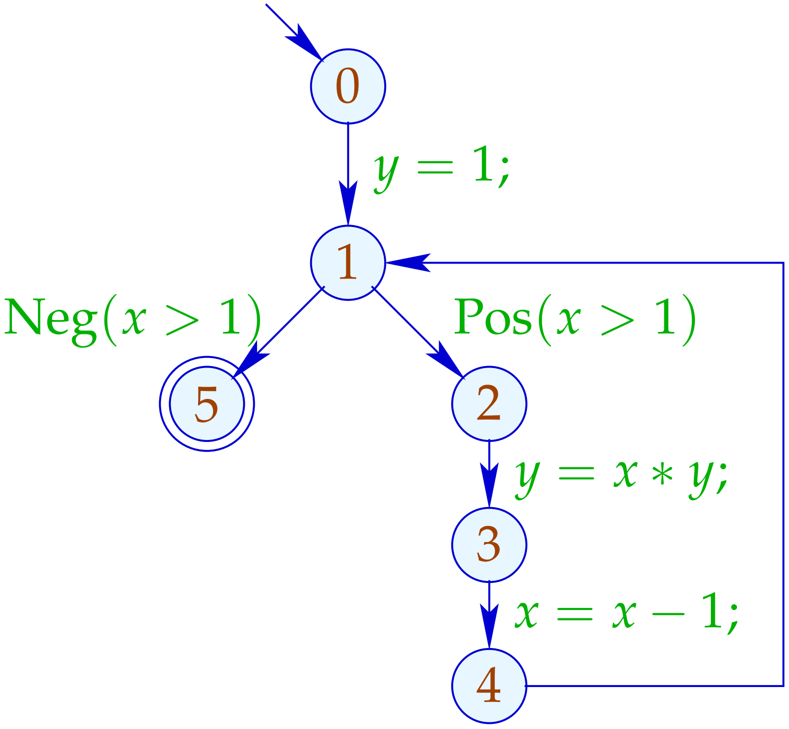


$$
\begin{array}{lll}
\mathcal{A}[0] & \subseteq & \emptyset \\
\mathcal{A}[1] & \subseteq & (\mathcal{A}[0] \cup \{1\}) \backslash \mathit{Expr}_y \\
\mathcal{A}[1] & \subseteq & \mathcal{A}[4] \\
\mathcal{A}[2] & \subseteq & \mathcal{A}[1] \cup \{x > 1\}
\end{array}
$$

Wanted:

- a maximally large solution (??)
- an algorithm which computes this :-)

Example:

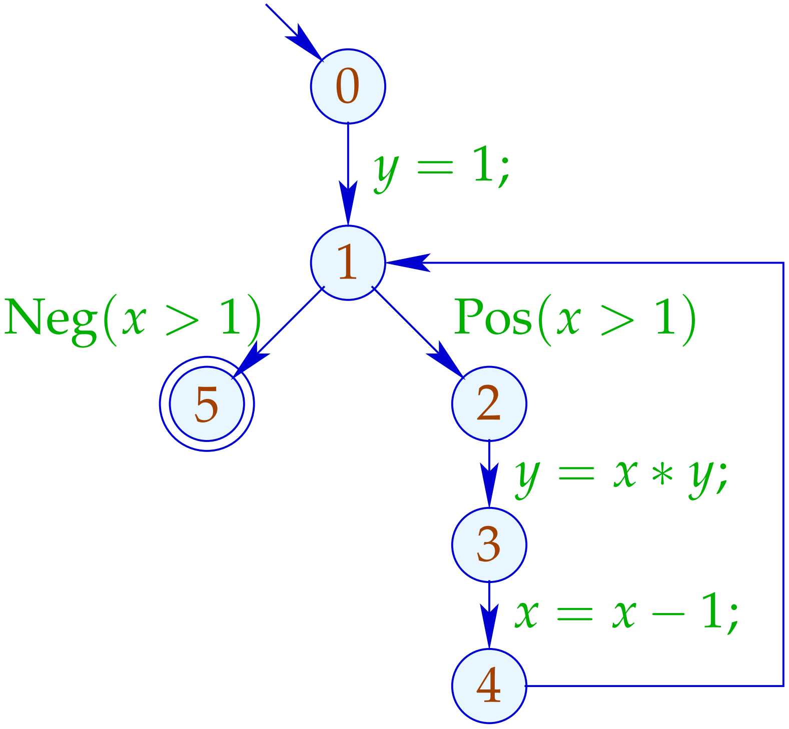


$$\begin{array}{lll}
\mathcal{A}[0] & \subseteq & \emptyset \\
\mathcal{A}[1] & \subseteq & (\mathcal{A}[0] \cup \{1\}) \backslash Expr_y \\
\mathcal{A}[1] & \subseteq & \mathcal{A}[4] \\
\mathcal{A}[2] & \subseteq & \mathcal{A}[1] \cup \{x > 1\} \\
\mathcal{A}[3] & \subseteq & (\mathcal{A}[2] \cup \{x * y\}) \backslash Expr_y
\end{array}$$

Wanted:

- a maximally large solution (??)
- an algorithm which computes this :-)

Example:

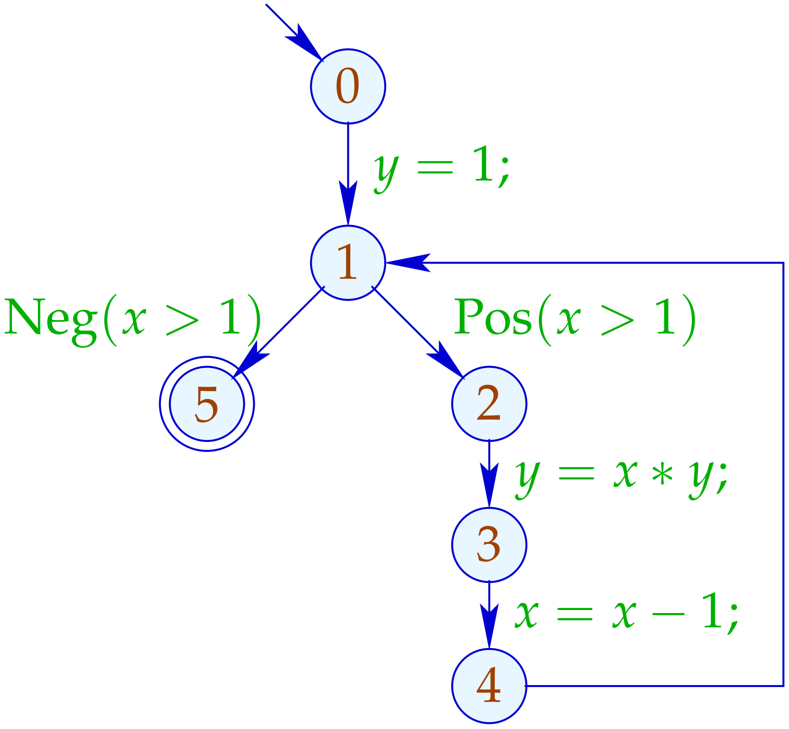


$$
\begin{array}{lcl}
\mathcal{A}[0] & \subseteq & \emptyset \\
\mathcal{A}[1] & \subseteq & (\mathcal{A}[0] \cup \{1\}) \backslash Expr_y \\
\mathcal{A}[1] & \subseteq & \mathcal{A}[4] \\
\mathcal{A}[2] & \subseteq & \mathcal{A}[1] \cup \{x > 1\} \\
\mathcal{A}[3] & \subseteq & (\mathcal{A}[2] \cup \{x * y\}) \backslash Expr_y \\
\mathcal{A}[4] & \subseteq & (\mathcal{A}[3] \cup \{x - 1\}) \backslash Expr_x
\end{array}
$$

Wanted:

- a maximally large solution (??)
- an algorithm which computes this :-)

Example:

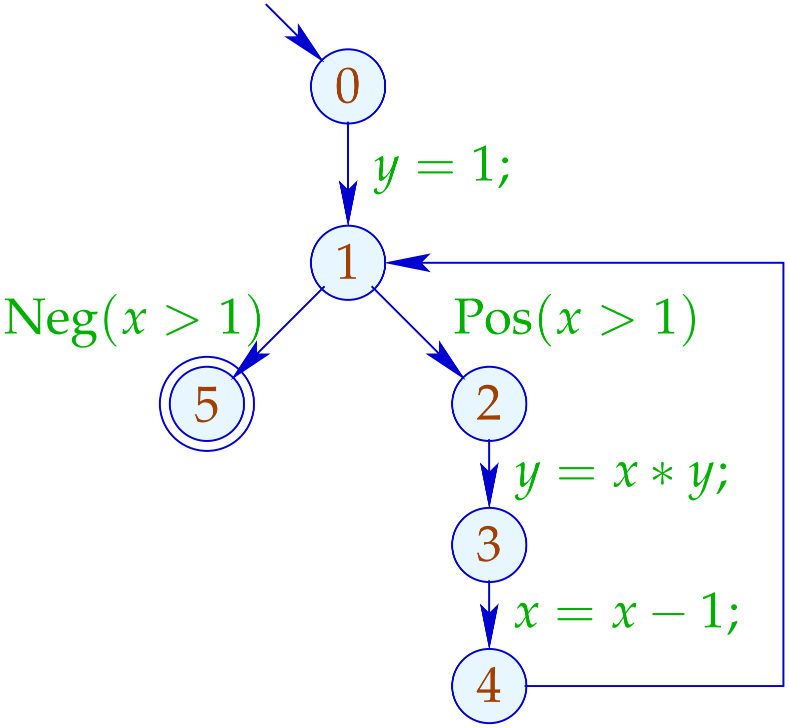


$$\begin{array}{lcl}
\mathcal{A}[0] & \subseteq & \emptyset \\
\mathcal{A}[1] & \subseteq & (\mathcal{A}[0] \cup \{1\}) \backslash Expr_y \\
\mathcal{A}[1] & \subseteq & \mathcal{A}[4] \\
\mathcal{A}[2] & \subseteq & \mathcal{A}[1] \cup \{x > 1\} \\
\mathcal{A}[3] & \subseteq & (\mathcal{A}[2] \cup \{x * y\}) \backslash Expr_y \\
\mathcal{A}[4] & \subseteq & (\mathcal{A}[3] \cup \{x - 1\}) \backslash Expr_x \\
\mathcal{A}[5] & \subseteq & \mathcal{A}[1] \cup \{x > 1\}
\end{array}$$

Wanted:

- a maximally large solution (??)
- an algorithm which computes this :-)

Example:



Solution:

$$
\begin{aligned}
\mathcal{A}[0] &= \emptyset \\
\mathcal{A}[1] &= \{1\} \\
\mathcal{A}[2] &= \{1, x > 1\} \\
\mathcal{A}[3] &= \{1, x > 1\} \\
\mathcal{A}[4] &= \{1\} \\
\mathcal{A}[5] &= \{1, x > 1\}
\end{aligned}
$$

Observation:

- The possible values for $\mathcal{A}[u]$ form a complete lattice:

$$\mathbb{D} = 2^{Expr} \quad \text{with} \quad B_1 \sqsubseteq B_2 \quad \text{iff} \quad B_1 \supseteq B_2$$

Observation:

- The possible values for $\mathcal{A}[u]$ form a complete lattice:

$$\mathbb{D} = 2^{Expr} \quad \text{with} \quad B_1 \sqsubseteq B_2 \quad \text{iff} \quad B_1 \supseteq B_2$$

- The functions $\quad [\![k]\!]^\sharp : \mathbb{D} \to \mathbb{D} \quad$ are monotonic, i.e.,

$$[\![k]\!]^\sharp(B_1) \sqsubseteq [\![k]\!]^\sharp(B_2) \quad \text{iff} \quad B_1 \sqsubseteq B_2$$

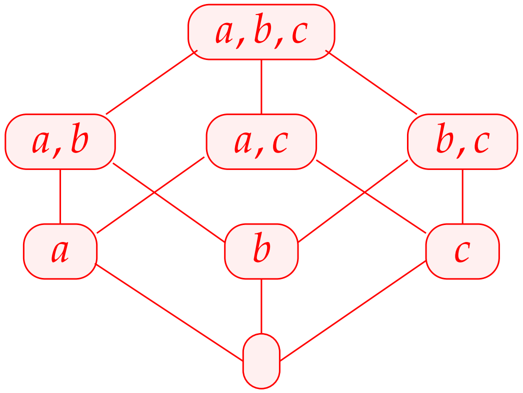
Background 2: complete Lattices

A set $\mathbb{D}$ together with a relation $\sqsubseteq \; \subseteq \; \mathbb{D} \times \mathbb{D}$ is a partial order if for all $a, b, c \in \mathbb{D}$,

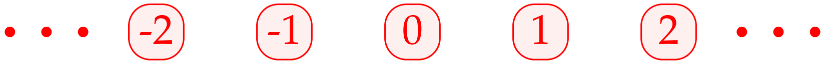
$$\begin{array}{lll} a \sqsubseteq a & & \textit{reflexivity} \\ a \sqsubseteq b \wedge b \sqsubseteq a & \Longrightarrow \quad a = b & \textit{anti-symmetry} \\ a \sqsubseteq b \wedge b \sqsubseteq c & \Longrightarrow \quad a \sqsubseteq c & \textit{transitivity} \end{array}$$

Examples:

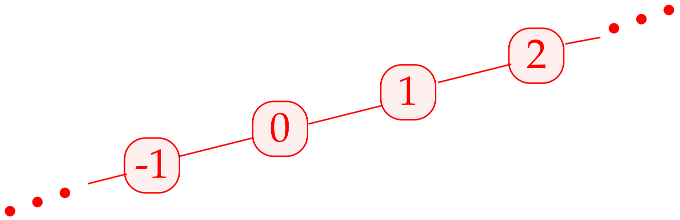
1. $\mathbb{D} = 2^{\{a,b,c\}}$ with the relation "$\subseteq$" :



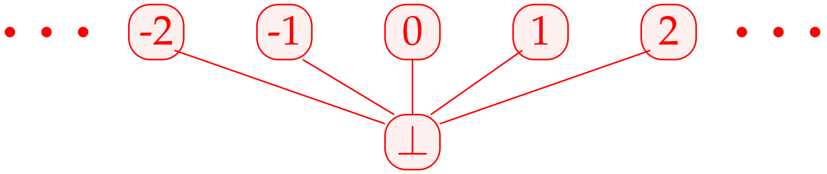
3. $\mathbb{Z}$ with the relation "$=$" :



3. $\mathbb{Z}$ with the relation "$\leq$" :



4. $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$ with the ordering:



$d \in \mathbb{D}$ is called upper bound for $X \subseteq \mathbb{D}$ if

$$x \sqsubseteq d \qquad \text{for all } x \in X$$

$d \in \mathbb{D}$ is called upper bound for $X \subseteq \mathbb{D}$ if

$$x \sqsubseteq d \qquad \text{for all } x \in X$$

d is called least upper bound (lub) if

1. d is an upper bound and
2. $d \sqsubseteq y$ for every upper bound y of X.

$d \in \mathbb{D}$ is called upper bound for $X \subseteq \mathbb{D}$ if

$$x \sqsubseteq d \qquad \text{for all } x \in X$$

d is called least upper bound (lub) if

1. d is an upper bound and
2. $d \sqsubseteq y$ for every upper bound y of X.

Warning:

- $\{0, 2, 4, \ldots\} \subseteq \mathbb{Z}$ has no upper bound!
- $\{0, 2, 4\} \subseteq \mathbb{Z}$ has the upper bounds $4, 5, 6, \ldots$

A complete lattice (cl) $\mathbb{D}$ is a partial ordering where every subset $X \subseteq \mathbb{D}$ has a least upper bound $\bigsqcup X \in \mathbb{D}$.

Note:

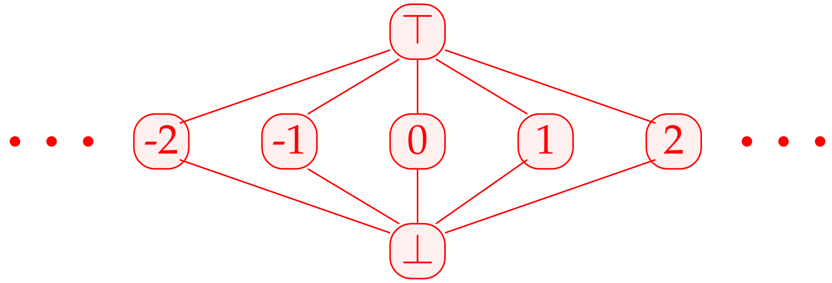
Every complete lattice has

→ a least element $\bot = \bigsqcup \emptyset \in \mathbb{D}$;

→ a greatest element $\top = \bigsqcup \mathbb{D} \in \mathbb{D}$.

Examples:

1. $\mathbb{D} = 2^{\{a,b,c\}}$ is a cl :-)
2. $\mathbb{D} = \mathbb{Z}$ with "$=$" is not.
3. $\mathbb{D} = \mathbb{Z}$ with "$\leq$" is neither.
4. $\mathbb{D} = \mathbb{Z}_\perp$ is also not :-(
5. With an extra element $\top$, we obtain the flat lattice $\mathbb{Z}_\perp^\top = \mathbb{Z} \cup \{\perp, \top\}$:



We have:

Theorem:

If $\mathbb{D}$ is a complete lattice, then every subset $X \subseteq \mathbb{D}$ has a greatest lower bound $\bigsqcap X$.

We have:

Theorem:

If $\mathbb{D}$ is a complete lattice, then every subset $X \subseteq \mathbb{D}$ has a greatest lower bound $\bigsqcap X$.

Proof:

Construct $U = \{u \in \mathbb{D} \mid \forall\, x \in X : \; u \sqsubseteq x\}$.

// the set of all lower bounds of X :-)

We have:

Theorem:

If $\mathbb{D}$ is a complete lattice, then every subset $X \subseteq \mathbb{D}$ has a greatest lower bound $\bigsqcap X$.

Proof:

Construct $U = \{u \in \mathbb{D} \mid \forall\, x \in X : \; u \sqsubseteq x\}$.

// the set of all lower bounds of X :-)

Set: $g := \bigsqcup U$

Claim: $g = \bigsqcap X$

(1) g is a lower bound of X :

Assume $x \in X$. Then:

$u \sqsubseteq x$ for all $u \in U$

$\Longrightarrow$ x is an upper bound of U

$\Longrightarrow$ $g \sqsubseteq x$:-)

(1) g is a lower bound of X :

Assume $x \in X$. Then:

$u \sqsubseteq x$ for all $u \in U$

$\Longrightarrow$ x is an upper bound of U

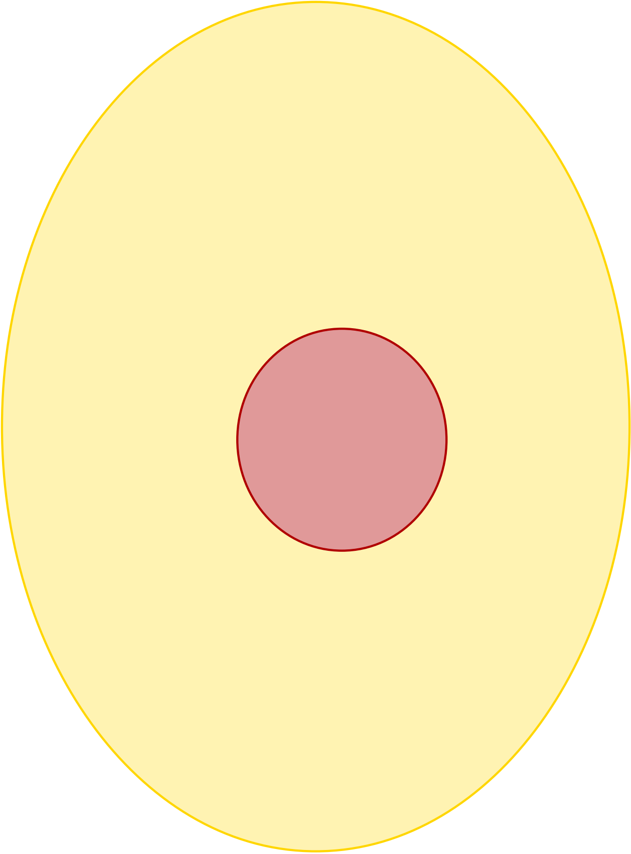
$\Longrightarrow$ $g \sqsubseteq x$:-)

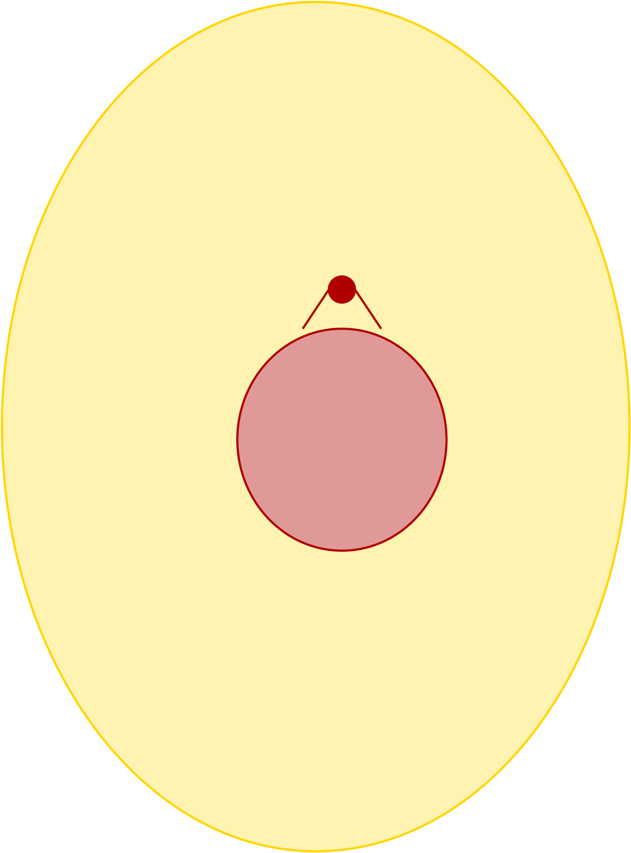
(2) g is the greatest lower bound of X :

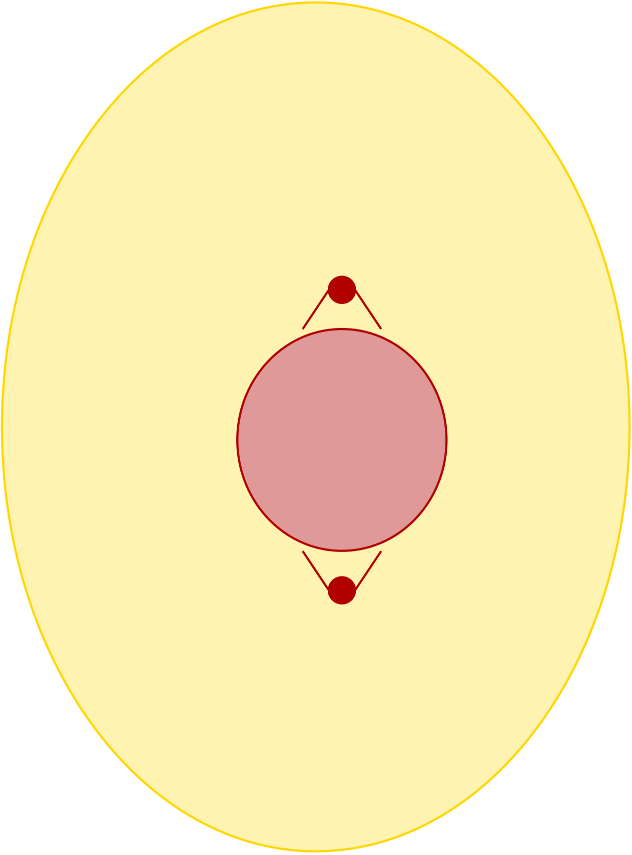
Assume u is a lower bound of X. Then:

$u \in U$

$\Longrightarrow$ $u \sqsubseteq g$:-))







We are looking for solutions for systems of constraints of the form:

$$x_i \quad \sqsupseteq \quad f_i(x_1, \ldots, x_n) \qquad\qquad (*)$$

We are looking for solutions for systems of constraints of the form:

$$x_i \quad \sqsupseteq \quad f_i(x_1, \ldots, x_n) \qquad\qquad (*)$$

where:

x_i	unknown	here:	$\mathcal{A}[u]$
$\mathbb{D}$	values	here:	2^{Expr}
$\sqsubseteq \quad \subseteq \quad \mathbb{D} \times \mathbb{D}$	ordering relation	here:	$\supseteq$
$f_i \colon \mathbb{D}^n \longrightarrow \mathbb{D}$	constraint	here:	...

We are looking for solutions for systems of constraints of the form:

$$x_i \quad \sqsupseteq \quad f_i(x_1, \ldots, x_n) \qquad\qquad (*)$$

where:

x_i	unknown	here:	$\mathcal{A}[u]$
$\mathbb{D}$	values	here:	2^{Expr}
$\sqsubseteq \;\subseteq\; \mathbb{D} \times \mathbb{D}$	ordering relation	here:	$\supseteq$
$f_i\colon \mathbb{D}^n \to \mathbb{D}$	constraint	here:	...

Constraint for $\mathcal{A}[v]$ $(v \neq start)$:

$$\mathcal{A}[v] \quad \subseteq \quad \bigcap\{[\![k]\!]^\sharp\,(\mathcal{A}[u]) \mid k = (u,\_,v) \;\text{ edge}\}$$

We are looking for solutions for systems of constraints of the form:

$$x_i \quad \sqsupseteq \quad f_i(x_1, \ldots, x_n) \qquad\qquad (*)$$

where:

x_i	unknown	here:	$\mathcal{A}[u]$
$\mathbb{D}$	values	here:	2^{Expr}
$\sqsubseteq \;\subseteq\; \mathbb{D} \times \mathbb{D}$	ordering relation	here:	$\supseteq$
$f_i \colon \mathbb{D}^n \to \mathbb{D}$	constraint	here:	...

Constraint for $\mathcal{A}[v]$ $(v \neq start)$:

$$\mathcal{A}[v] \quad \subseteq \quad \bigcap\{[\![k]\!]^\sharp\,(\mathcal{A}[u]) \mid k = (u, \_, v) \;\text{ edge}\}$$

Because:

$$x \sqsupseteq d_1 \wedge \ldots \wedge x \sqsupseteq d_k \quad \text{iff} \quad x \sqsupseteq \bigsqcup\{d_1, \ldots, d_k\} \qquad :\text{-})$$

A mapping $f : \mathbb{D}_1 \to \mathbb{D}_2$ is called monotonic, if $f(a) \sqsubseteq f(b)$ for all $a \sqsubseteq b$.

A mapping $f : \mathbb{D}_1 \to \mathbb{D}_2$ is called monotonic, if $f(a) \sqsubseteq f(b)$ for all $a \sqsubseteq b$.

Examples:

(1) $\mathbb{D}_1 = \mathbb{D}_2 = 2^U$ for a set U and $f\,x = (x \cap a) \cup b$.

Obviously, every such f is monotonic :-)

A mapping $f : \mathbb{D}_1 \to \mathbb{D}_2$ is called monotonic, is $f(a) \sqsubseteq f(b)$ for all $a \sqsubseteq b$.

Examples:

(1) $\mathbb{D}_1 = \mathbb{D}_2 = 2^U$ for a set U and $f\,x = (x \cap a) \cup b$.

Obviously, every such f is monotonic :-)

(2) $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{Z}$ (with the ordering "$\leq$"). Then:

- $\mathsf{inc}\,x = x + 1$ is monotonic.
- $\mathsf{dec}\,x = x - 1$ is monotonic.

A mapping $f : \mathbb{D}_1 \to \mathbb{D}_2$ is called monotonic, is $f(a) \sqsubseteq f(b)$ for all $a \sqsubseteq b$.

Examples:

(1) $\mathbb{D}_1 = \mathbb{D}_2 = 2^U$ for a set U and $f\, x = (x \cap a) \cup b$.

Obviously, every such f is monotonic :-)

(2) $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{Z}$ (with the ordering "$\leq$"). Then:

- $\textsf{inc}\, x = x + 1$ is monotonic.
- $\textsf{dec}\, x = x - 1$ is monotonic.
- $\textsf{inv}\, x = -x$ is not monotonic :-)

Theorem:

If $\quad f_1 : \mathbb{D}_1 \longrightarrow \mathbb{D}_2 \quad$ and $\quad f_2 : \mathbb{D}_2 \longrightarrow \mathbb{D}_3 \quad$ are monotonic, then also $f_2 \circ f_1 : \mathbb{D}_1 \longrightarrow \mathbb{D}_3$:-)

Theorem:

If $f_1 : \mathbb{D}_1 \to \mathbb{D}_2$ and $f_2 : \mathbb{D}_2 \to \mathbb{D}_3$ are monotonic, then also $f_2 \circ f_1 : \mathbb{D}_1 \to \mathbb{D}_3$:-)

Theorem:

If $\mathbb{D}_2$ is a complete lattice, then the set $[\mathbb{D}_1 \to \mathbb{D}_2]$ of monotonic functions $f : \mathbb{D}_1 \to \mathbb{D}_2$ is also a complete lattice where

$$f \sqsubseteq g \quad \text{iff} \quad f\, x \sqsubseteq g\, x \quad \text{for all } x \in \mathbb{D}_1$$

Theorem:

If $f_1 : \mathbb{D}_1 \to \mathbb{D}_2$ and $f_2 : \mathbb{D}_2 \to \mathbb{D}_3$ are monotonic, then also $f_2 \circ f_1 : \mathbb{D}_1 \to \mathbb{D}_3$:-)

Theorem:

If $\mathbb{D}_2$ is a complete lattice, then the set $[\mathbb{D}_1 \to \mathbb{D}_2]$ of monotonic functions $f : \mathbb{D}_1 \to \mathbb{D}_2$ is also a complete lattice where

$$f \sqsubseteq g \quad \text{iff} \quad f\,x \sqsubseteq g\,x \quad \text{for all } x \in \mathbb{D}_1$$

In particular for $F \subseteq [\mathbb{D}_1 \to \mathbb{D}_2]$,

$$\bigsqcup F = f \quad \text{mit} \quad f\,x = \bigsqcup \{g\,x \mid g \in F\}$$

For functions $f_i\, x = a_i \cap x \cup b_i$, the operations "$\circ$", "$\sqcup$" and "$\sqcap$" can be explicitly defined by:

$$\begin{aligned}
(f_2 \circ f_1)\, x &= \boxed{a_1 \cap a_2} \cap x \cup \boxed{a_2 \cap b_1 \cup b_2} \\
(f_1 \sqcup f_2)\, x &= \boxed{(a_1 \cup a_2)} \cap x \cup \boxed{b_1 \cup b_2} \\
(f_1 \sqcap f_2)\, x &= \boxed{(a_1 \cup b_1) \cap (a_2 \cup b_2)} \cap x \cup \boxed{b_1 \cap b_2}
\end{aligned}$$

Wanted: minimally small solution for:

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (*)$$

where all $f_i : \mathbb{D}^n \to \mathbb{D}$ are monotonic.

Wanted: minimally small solution for:

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (*)$$

where all $f_i : \mathbb{D}^n \to \mathbb{D}$ are monotonic.

Idea:

- Consider $F : \mathbb{D}^n \longrightarrow \mathbb{D}^n$ where

  $F(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$ with $y_i = f_i(x_1, \ldots, x_n)$.

Wanted: minimally small solution for:

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad\qquad (*)$$

where all $f_i : \mathbb{D}^n \to \mathbb{D}$ are monotonic.

Idea:

- Consider $F : \mathbb{D}^n \longrightarrow \mathbb{D}^n$ where

  $F(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$ with $y_i = f_i(x_1, \ldots, x_n)$.
- If all f_i are monotonic, then also F :-)

Wanted: minimally small solution for:

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (*)$$

where all $f_i : \mathbb{D}^n \to \mathbb{D}$ are monotonic.

Idea:

- Consider $F : \mathbb{D}^n \longrightarrow \mathbb{D}^n$ where
  $$F(x_1, \ldots, x_n) = (y_1, \ldots, y_n) \quad \text{with} \quad y_i = f_i(x_1, \ldots, x_n).$$
- If all f_i are monotonic, then also F :-)
- We successively approximate a solution. We construct:
  $$\underline{\perp}, \quad F\,\underline{\perp}, \quad F^2\,\underline{\perp}, \quad F^3\,\underline{\perp}, \quad \ldots$$

Hope: We eventually reach a solution ... ???

Example: $\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq \; = \; \subseteq$

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a,b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

Example: $\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq \, = \, \subseteq$

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a,b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

The Iteration:

	0	1	2	3	4
x_1	$\emptyset$				
x_2	$\emptyset$				
x_3	$\emptyset$				

Example: $\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq \,=\, \subseteq$

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

The Iteration:

	0	1	2	3	4
x_1	$\emptyset$	$\{a\}$			
x_2	$\emptyset$	$\emptyset$			
x_3	$\emptyset$	$\{c\}$			

Example: $\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq = \subseteq$

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a,b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

The Iteration:

	0	1	2	3	4
x_1	$\emptyset$	$\{a\}$	$\{a,c\}$		
x_2	$\emptyset$	$\emptyset$	$\emptyset$		
x_3	$\emptyset$	$\{c\}$	$\{a,c\}$		

Example: $\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq \, = \, \subseteq$

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

The Iteration:

	0	1	2	3	4
x_1	$\emptyset$	$\{a\}$	$\{a, c\}$	$\{a, c\}$	
x_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{a\}$	
x_3	$\emptyset$	$\{c\}$	$\{a, c\}$	$\{a, c\}$	

Example: $\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq = \subseteq$

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

The Iteration:

	0	1	2	3	4
x_1	$\emptyset$	$\{a\}$	$\{a, c\}$	$\{a, c\}$	dito
x_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{a\}$	
x_3	$\emptyset$	$\{c\}$	$\{a, c\}$	$\{a, c\}$	

Theorem

- $\underline{\perp}, F\,\underline{\perp}, F^2\,\underline{\perp}, \ldots$ form an ascending chain :

$$\underline{\perp} \quad \sqsubseteq \quad F\,\underline{\perp} \quad \sqsubseteq \quad F^2\,\underline{\perp} \quad \sqsubseteq \quad \ldots$$

- If $F^k\,\underline{\perp} = F^{k+1}\,\underline{\perp}$, a solution is obtained which is the least one :-)
- If all ascending chains are finite, such a k always exists.

Theorem

- $\underline{\perp}, F\,\underline{\perp}, F^2\,\underline{\perp}, \ldots$ form an ascending chain :
$$\underline{\perp} \quad \sqsubseteq \quad F\,\underline{\perp} \quad \sqsubseteq \quad F^2\,\underline{\perp} \quad \sqsubseteq \quad \ldots$$
- If $F^k\,\underline{\perp} = F^{k+1}\,\underline{\perp}$, a solution is obtained which is the least one :-)
- If all ascending chains are finite, such a k always exists.

Proof

The first claim follows by complete induction:

Foundation: $F^0\,\underline{\perp} = \underline{\perp} \sqsubseteq F^1\,\underline{\perp}$:-)

Step: Assume $F^{i-1}\,\underline{\perp} \;\sqsubseteq\; F^{i}\,\underline{\perp}$. Then

$$F^{i}\,\underline{\perp} = F\,(F^{i-1}\,\underline{\perp}) \sqsubseteq F\,(F^{i}\,\underline{\perp}) = F^{i+1}\,\underline{\perp}$$

since F monotonic :-)

Step: Assume $F^{i-1}\,\underline{\perp} \sqsubseteq F^{i}\,\underline{\perp}$. Then

$$F^{i}\,\underline{\perp} = F\,(F^{i-1}\,\underline{\perp}) \sqsubseteq F\,(F^{i}\,\underline{\perp}) = F^{i+1}\,\underline{\perp}$$

since F monotonic :-)

Conclusion:

If $\mathbb{D}$ is finite, a solution can be found which is definitely the least :-)

Question:

What, if $\mathbb{D}$ is not finite ???

Theorem Knaster – Tarski

Assume $\mathbb{D}$ is a complete lattice. Then every monotonic function $f : \mathbb{D} \to \mathbb{D}$ has a least fixpoint $d_0 \in \mathbb{D}$.

Let $P = \{d \in \mathbb{D} \mid f\,d \sqsubseteq d\}$.

Then $d_0 = \bigsqcap P$.

Bronisław Knaster (1893-1980), topology

Theorem Knaster – Tarski

Assume $\mathbb{D}$ is a complete lattice. Then every monotonic function $f : \mathbb{D} \to \mathbb{D}$ has a least fixpoint $d_0 \in \mathbb{D}$.

Let $P = \{d \in \mathbb{D} \mid f\, d \sqsubseteq d\}$.

Then $d_0 = \bigsqcap P$.

Proof:

(1) $d_0 \in P$:

Theorem Knaster – Tarski

Assume $\mathbb{D}$ is a complete lattice. Then every monotonic function $f : \mathbb{D} \to \mathbb{D}$ has a least fixpoint $d_0 \in \mathbb{D}$.

Let $P = \{d \in \mathbb{D} \mid f\, d \sqsubseteq d\}$.

Then $d_0 = \bigsqcap P$.

Proof:

(1) $d_0 \in P$:

$$f\, d_0 \sqsubseteq f\, d \sqsubseteq d \quad \text{for all } d \in P$$

$$\Longrightarrow \quad f\, d_0 \quad \text{is a lower bound of } P$$

$$\Longrightarrow \quad f\, d_0 \sqsubseteq d_0 \quad \text{since } d_0 = \bigsqcap P$$

$$\Longrightarrow \quad d_0 \in P$$

:-)

(2) $f\,d_0 = d_0 :$

(2) $\quad f\, d_0 = d_0$:

$$\begin{array}{lll} & f\, d_0 \sqsubseteq d_0 & \text{by} \quad (1) \\ \Longrightarrow & f\,(f\, d_0) \sqsubseteq f\, d_0 & \text{by monotonicity of } f \\ \Longrightarrow & f\, d_0 \in P & \\ \Longrightarrow & d_0 \sqsubseteq f\, d_0 & \text{and the claim follows} \quad \text{:-)} \end{array}$$

(2) $f\, d_0 = d_0$:

$$
\begin{array}{ll}
 & f\, d_0 \sqsubseteq d_0 \quad \text{by} \quad (1) \\
\Longrightarrow & f\,(f\, d_0) \sqsubseteq f\, d_0 \quad \text{by monotonicity of } f \\
\Longrightarrow & f\, d_0 \in P \\
\Longrightarrow & d_0 \sqsubseteq f\, d_0 \qquad \text{and the claim follows} \quad \text{:-)}
\end{array}
$$

(3) d_0 is least fixpoint:

(2) $f\,d_0 = d_0$:

$f\,d_0 \sqsubseteq d_0$ by (1)

$\Longrightarrow$ $f\,(f\,d_0) \sqsubseteq f\,d_0$ by monotonicity of f

$\Longrightarrow$ $f\,d_0 \in P$

$\Longrightarrow$ $d_0 \sqsubseteq f\,d_0$ and the claim follows :-)

(3) d_0 is least fixpoint:

$f\,d_1 = d_1 \sqsubseteq d_1$ an other fixpoint

$\Longrightarrow$ $d_1 \in P$

$\Longrightarrow$ $d_0 \sqsubseteq d_1$:-))

Remark:

The least fixpoint d_0 is in P and a lower bound :-)

$\Longrightarrow$ d_0 is the least value x with $x \sqsupseteq f\,x$

Remark:

The least fixpoint d_0 is in P and a lower bound :-)

$\Longrightarrow$ d_0 is the least value x with $x \sqsupseteq f\, x$

Application:

Assume $x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n$ $(*)$

is a system of constraints where all $f_i : \mathbb{D}^n \to \mathbb{D}$ are monotonic.

Remark:

The least fixpoint d_0 is in P and a lower bound :-)

$\Longrightarrow$ d_0 is the least value x with $x \sqsupseteq f\,x$

Application:

Assume $x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (*)$

is a system of constraints where all $f_i : \mathbb{D}^n \to \mathbb{D}$ are monotonic.

$\Longrightarrow$ least solution of $(*)$ $==$ least fixpoint of F :-)

Example 1: $\mathbb{D} = 2^U, \quad f\,x = x \cap a \cup b$

Example 1: $\mathbb{D} = 2^U, \quad f\,x = x \cap a \cup b$

f	$f^k\,\bot$	$f^k\,\top$
0	$\emptyset$	U

Example 1: $\mathbb{D} = 2^U, \quad f\,x = x \cap a \cup b$

f	$f^k\,\bot$	$f^k\,\top$
0	$\emptyset$	U
1	b	$a \cup b$

Example 1: $\mathbb{D} = 2^U, \quad f\,x = x \cap a \cup b$

f	$f^k\,\bot$	$f^k\,\top$
0	$\emptyset$	U
1	b	$a \cup b$
2	b	$a \cup b$

Example 1: $\mathbb{D} = 2^U, \quad f\,x = x \cap a \cup b$

f	$f^k\,\perp$	$f^k\,\top$
0	$\emptyset$	U
1	b	$a \cup b$
2	b	$a \cup b$

Example 2: $\mathbb{D} = \mathbb{N} \cup \{\infty\}$

Assume $f\,x = x + 1$. Then

$$f^i\,\perp = f^i\,0 = i \quad \sqsubset \quad i + 1 = f^{i+1}\,\perp$$

Example 1: $\mathbb{D} = 2^U, \quad f\,x = x \cap a \cup b$

f	$f^k\,\bot$	$f^k\,\top$
0	$\emptyset$	U
1	b	$a \cup b$
2	b	$a \cup b$

Example 2: $\mathbb{D} = \mathbb{N} \cup \{\infty\}$

Assume $f\,x = x + 1$. Then

$$f^i\,\bot = f^i\,0 = i \quad \sqsubset \quad i + 1 = f^{i+1}\,\bot$$

$\Longrightarrow$ Ordinary iteration will never reach a fixpoint :-(

$\Longrightarrow$ Sometimes, transfinite iteration is needed :-)

Conclusion:

Systems of inequations can be solved through fixpoint iteration, i.e., by repeated evaluation of right-hand sides :-)

Conclusion:

Systems of inequations can be solved through fixpoint iteration, i.e., by repeated evaluation of right-hand sides :-)

Warning: Naive fixpoint iteration is rather inefficient :-(

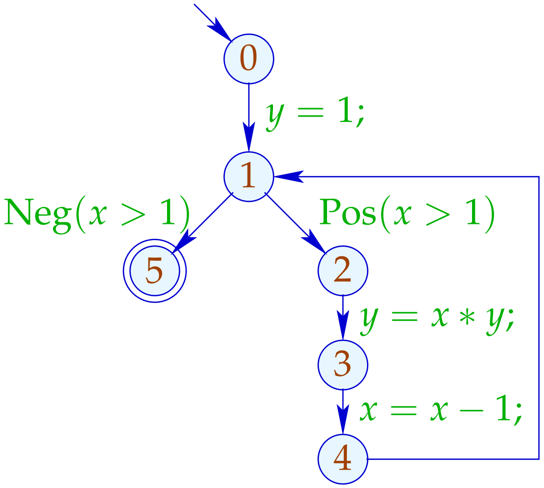
Conclusion:

Systems of inequations can be solved through fixpoint iteration, i.e., by repeated evaluation of right-hand sides :-)

Warning:

Naive fixpoint iteration is rather inefficient :-(

Example:



0
1
2
3
4
5

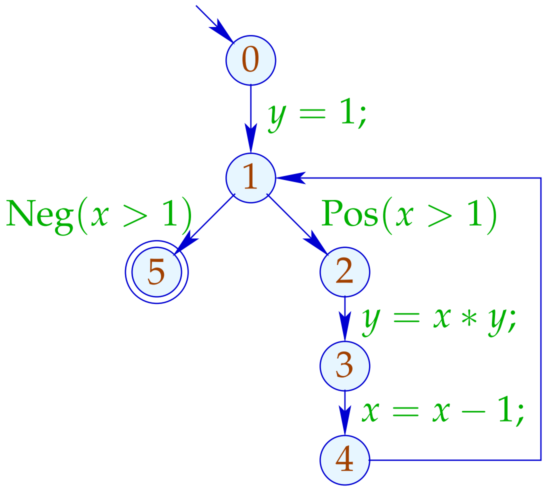
Conclusion:

Systems of inequations can be solved through fixpoint iteration, i.e., by repeated evaluation of right-hand sides :-)

Warning:

Naive fixpoint iteration is rather inefficient :-(

Example:



	1
0	$\emptyset$
1	$\{1, x > 1, x - 1\}$
2	*Expr*
3	$\{1, x > 1, x - 1\}$
4	$\{1\}$
5	*Expr*

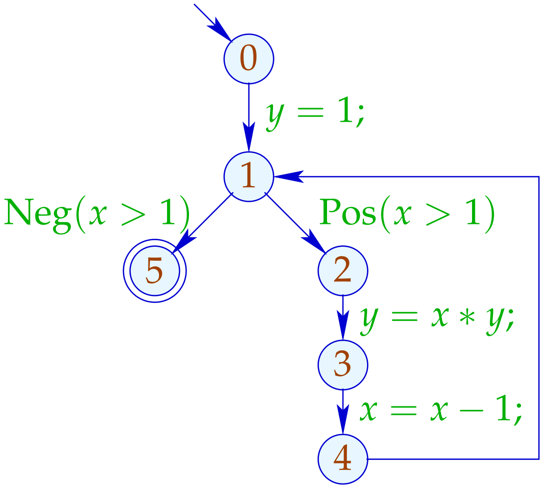
Conclusion:

Systems of inequations can be solved through fixpoint iteration, i.e., by repeated evaluation of right-hand sides :-)

Warning:

Naive fixpoint iteration is rather inefficient :-(

Example:



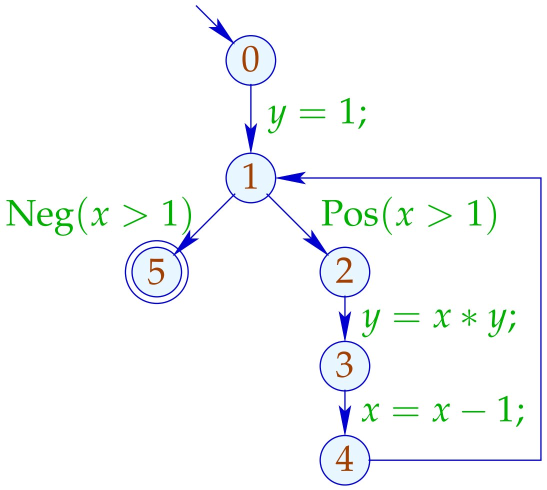
	1	2
0	$\emptyset$	$\emptyset$
1	$\{1, x > 1, x - 1\}$	$\{1\}$
2	*Expr*	$\{1, x > 1, x - 1\}$
3	$\{1, x > 1, x - 1\}$	$\{1, x > 1, x - 1\}$
4	$\{1\}$	$\{1\}$
5	*Expr*	$\{1, x > 1, x - 1\}$

Conclusion:

Systems of inequations can be solved through fixpoint iteration, i.e., by repeated evaluation of right-hand sides :-)

Warning: Naive fixpoint iteration is rather inefficient :-(

Example:



	1	2	3
0	$\emptyset$	$\emptyset$	$\emptyset$
1	$\{1, x > 1, x - 1\}$	$\{1\}$	$\{1\}$
2	*Expr*	$\{1, x > 1, x - 1\}$	$\{1, x > 1\}$
3	$\{1, x > 1, x - 1\}$	$\{1, x > 1, x - 1\}$	$\{1, x > 1, x - 1\}$
4	$\{1\}$	$\{1\}$	$\{1\}$
5	*Expr*	$\{1, x > 1, x - 1\}$	$\{1, x > 1\}$

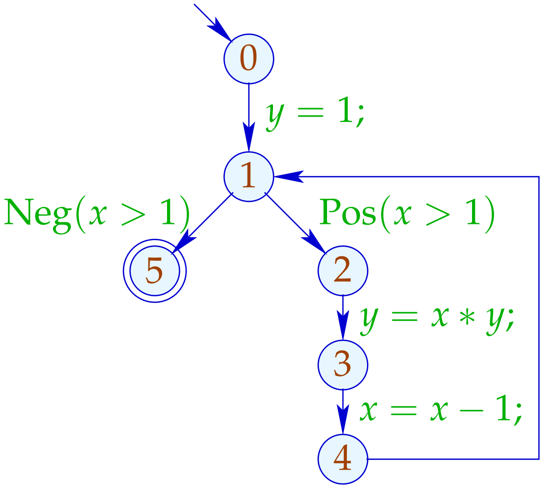
Conclusion:

Systems of inequations can be solved through fixpoint iteration, i.e., by repeated evaluation of right-hand sides :-)

Warning:

Naive fixpoint iteration is rather inefficient :-(

Example:



	1	2	3	4
0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\{1, x > 1, x - 1\}$	$\{1\}$	$\{1\}$	$\{1\}$
2	*Expr*	$\{1, x > 1, x - 1\}$	$\{1, x > 1\}$	$\{1, x > 1\}$
3	$\{1, x > 1, x - 1\}$	$\{1, x > 1, x - 1\}$	$\{1, x > 1, x - 1\}$	$\{1, x > 1\}$
4	$\{1\}$	$\{1\}$	$\{1\}$	$\{1\}$
5	*Expr*	$\{1, x > 1, x - 1\}$	$\{1, x > 1\}$	$\{1, x > 1\}$

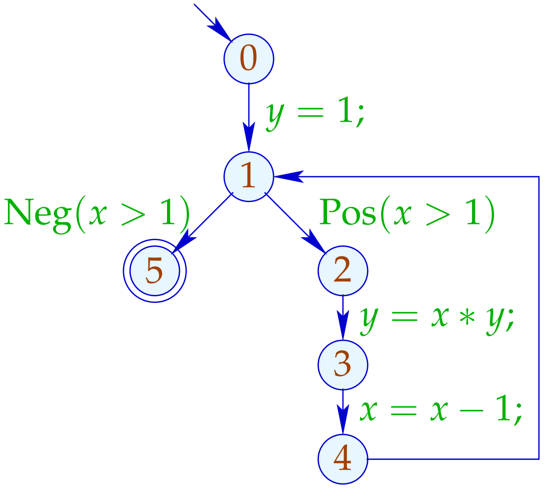
Conclusion:

Systems of inequations can be solved through fixpoint iteration, i.e., by repeated evaluation of right-hand sides :-)

Warning:

Naive fixpoint iteration is rather inefficient :-(

Example:



	1	2	3	4	5
0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	
1	$\{1, x > 1, x - 1\}$	$\{1\}$	$\{1\}$	$\{1\}$	
2	*Expr*	$\{1, x > 1, x - 1\}$	$\{1, x > 1\}$	$\{1, x > 1\}$	
3	$\{1, x > 1, x - 1\}$	$\{1, x > 1, x - 1\}$	$\{1, x > 1, x - 1\}$	$\{1, x > 1\}$	dito
4	$\{1\}$	$\{1\}$	$\{1\}$	$\{1\}$	
5	*Expr*	$\{1, x > 1, x - 1\}$	$\{1, x > 1\}$	$\{1, x > 1\}$	

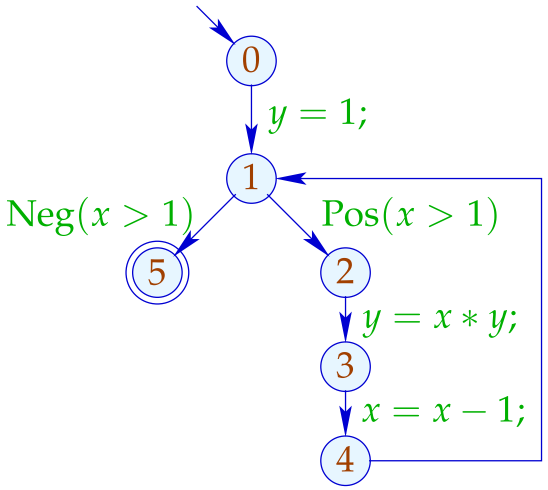
Idea: Round Robin Iteration

Instead of accessing the values of the last iteration, always use the current values of unknowns :-)

Idea: Round Robin Iteration

Instead of accessing the values of the last iteration, always use the current values of unknowns :-)

Example:

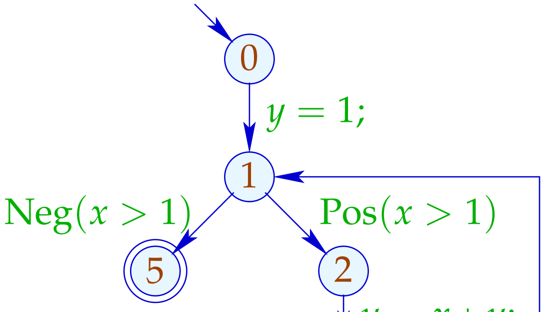


0
1
2
3
4
5

Idea: Round Robin Iteration

Instead of accessing the values of the last iteration, always use the current values of unknowns :-)

Example:

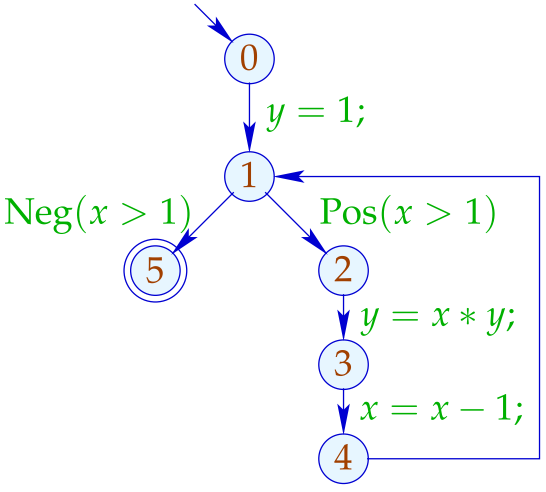


	1
0	$\emptyset$
1	$\{1\}$
2	$\{1, x > 1\}$
3	$\{1, x > 1\}$
4	$\{1\}$
5	$\{1, x > 1\}$

Idea: Round Robin Iteration

Instead of accessing the values of the last iteration, always use the current values of unknowns :-)

Example:



	1	2
0	$\emptyset$	dito
1	$\{1\}$	
2	$\{1, x > 1\}$	
3	$\{1, x > 1\}$	
4	$\{1\}$	
5	$\{1, x > 1\}$	

The code for Round Robin Iteration in Java looks as follows:

```
for $(i = 1; i \leq n; i{+}{+})\ x_i = \bot;$
do {
        $finished = \text{true};$
        for $(i = 1; i \leq n; i{+}{+})$ {
                $new = f_i(x_1, \ldots, x_n);$
                if $(!(x_i \sqsupseteq new))$ {
                        $finished = \text{false};$
                        $x_i = x_i \sqcup new;$
                }
        }
} while $(!finished);$
```

Correctness:

Assume $y_i^{(d)}$ is the i-th component of $F^d \underline{\perp}$.

Assume $x_i^{(d)}$ is the value of x_i after the d-th RR-iteration.

Correctness:

Assume $y_i^{(d)}$ is the *i*-th component of $F^d \, \underline{\perp}$.

Assume $x_i^{(d)}$ is the value of x_i after the *i*-th RR-iteration.

One proves:

(1) $y_i^{(d)} \sqsubseteq x_i^{(d)}$:-)

Correctness:

Assume $y_i^{(d)}$ is the *i*-th component of $F^d \, \underline{\perp}$.

Assume $x_i^{(d)}$ is the value of x_i after the *i*-th RR-iteration.

One proves:

(1) $y_i^{(d)} \sqsubseteq x_i^{(d)}$:-)

(2) $x_i^{(d)} \sqsubseteq z_i$ for every solution $(z_1, \ldots, z_n)$:-)

Correctness:

Assume $y_i^{(d)}$ is the *i*-th component of $F^d \underline{\perp}$.

Assume $x_i^{(d)}$ is the value of x_i after the *i*-th RR-iteration.

One proves:

(1) $y_i^{(d)} \sqsubseteq x_i^{(d)}$:-)

(2) $x_i^{(d)} \sqsubseteq z_i$ for every solution $(z_1, \ldots, z_n)$:-)

(3) If RR-iteration terminates after *d* rounds, then
$(x_1^{(d)}, \ldots, x_n^{(d)})$ is a solution :-))

Warning:

The efficiency of RR-iteration depends on the ordering of the unknowns !!!

Warning:

The efficiency of RR-iteration depends on the ordering of the unknowns !!!

Good:

→ u before v, if $u \longrightarrow^* v$;

→ entry condition before loop body :-)

Warning:

The efficiency of RR-iteration depends on the ordering of the unknowns !!!

Good:

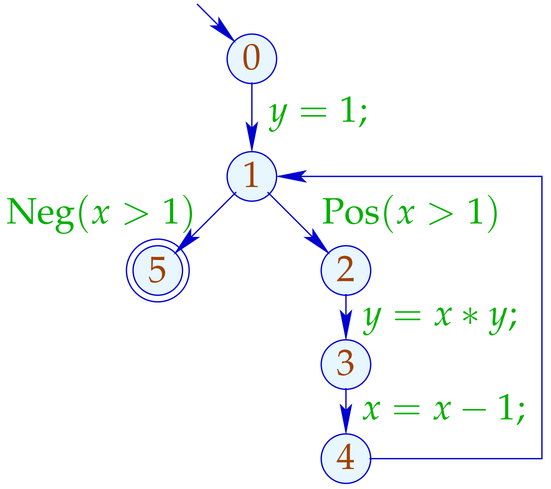
→ u before v, if $u \longrightarrow^* v$;

→ entry condition before loop body :-)

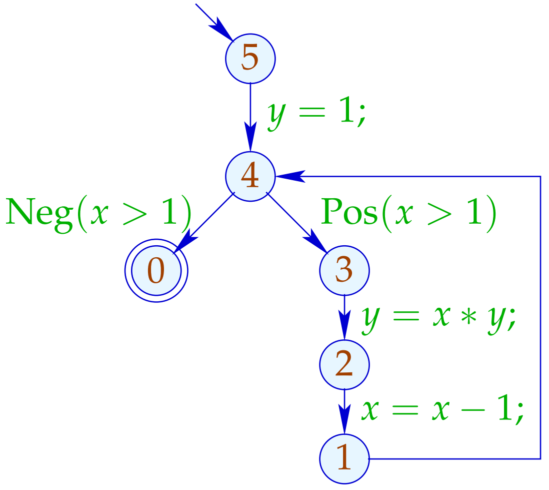
Bad:

e.g., post-order DFS of the CFG, starting at start :-)

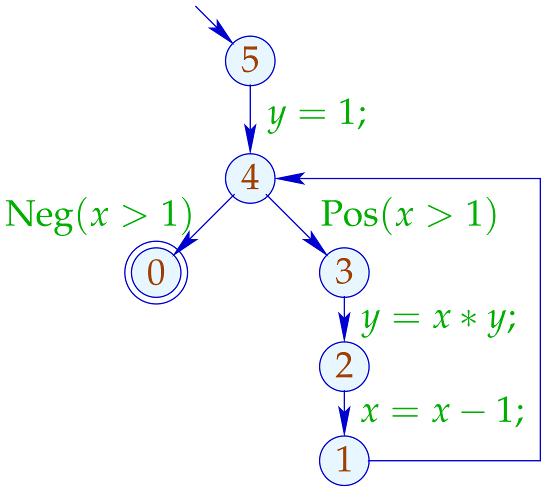
Good:


0
$y = 1;$
1
$\mathrm{Neg}(x > 1)$
$\mathrm{Pos}(x > 1)$
5
2
$y = x * y;$
3
$x = x - 1;$
4

Bad:

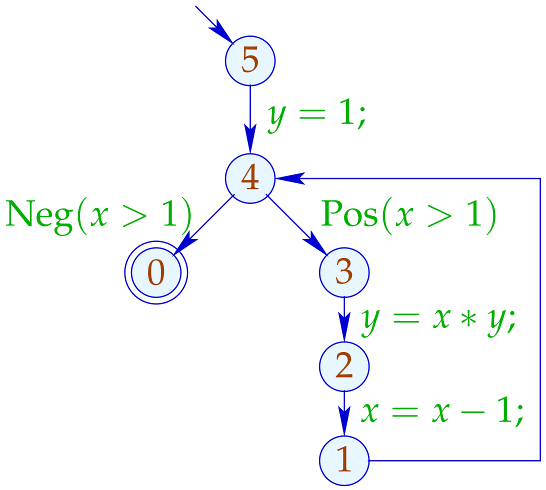

5
$y = 1;$
4
$\mathrm{Neg}(x > 1)$
$\mathrm{Pos}(x > 1)$
0
3
$y = x * y;$
2
$x = x - 1;$
1

Inefficient Round Robin Iteration:



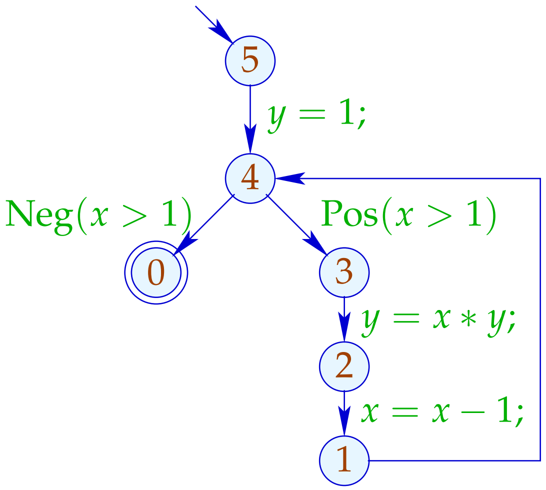
0
1
2
3
4
5

Inefficient Round Robin Iteration:



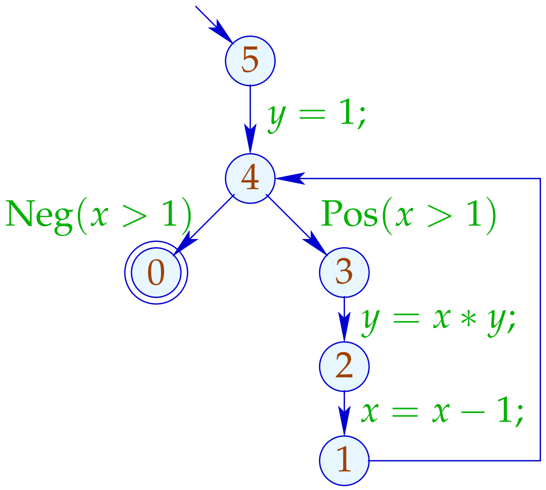
	1
0	*Expr*
1	$\{1\}$
2	$\{1, x - 1, x > 1\}$
3	*Expr*
4	$\{1\}$
5	$\emptyset$

Inefficient Round Robin Iteration:



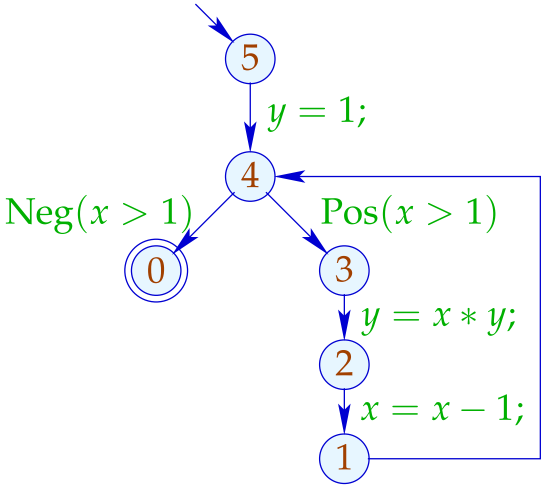
	1	2
0	*Expr*	$\{1, x > 1\}$
1	$\{1\}$	$\{1\}$
2	$\{1, x - 1, x > 1\}$	$\{1, x - 1, x > 1\}$
3	*Expr*	$\{1, x > 1\}$
4	$\{1\}$	$\{1\}$
5	$\emptyset$	$\emptyset$

Inefficient Round Robin Iteration:



	1	2	3
0	*Expr*	$\{1, x > 1\}$	$\{1, x > 1\}$
1	$\{1\}$	$\{1\}$	$\{1\}$
2	$\{1, x-1, x > 1\}$	$\{1, x-1, x > 1\}$	$\{1, x > 1\}$
3	*Expr*	$\{1, x > 1\}$	$\{1, x > 1\}$
4	$\{1\}$	$\{1\}$	$\{1\}$
5	$\emptyset$	$\emptyset$	$\emptyset$

Inefficient Round Robin Iteration:



	1	2	3	4
0	*Expr*	$\{1, x > 1\}$	$\{1, x > 1\}$	
1	$\{1\}$	$\{1\}$	$\{1\}$	
2	$\{1, x - 1, x > 1\}$	$\{1, x - 1, x > 1\}$	$\{1, x > 1\}$	dito
3	*Expr*	$\{1, x > 1\}$	$\{1, x > 1\}$	
4	$\{1\}$	$\{1\}$	$\{1\}$	
5	$\emptyset$	$\emptyset$	$\emptyset$	

$\Longrightarrow$ significantly less efficient :-)

... end of background on: Complete Lattices

... end of background on: Complete Lattices

Final Question:

Why is a (or the least) solution of the constraint system usefull ???

... end of background on: Complete Lattices

Final Question:

Why is a (or the least) solution of the constraint system usefull ???

For a complete lattice $\mathbb{D}$, consider systems:

$$\begin{array}{lcll} \mathcal{I}[\mathit{start}] & \sqsupseteq & d_0 & \\ \mathcal{I}[v] & \sqsupseteq & [\![k]\!]^\sharp\,(\mathcal{I}[u]) & \qquad k = (u, \_, v) \quad \text{edge} \end{array}$$

where $d_0 \in \mathbb{D}$ and all $[\![k]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ are monotonic ...

... end of background on: Complete Lattices

Final Question:

Why is a (or the least) solution of the constraint system usefull ???

For a complete lattice $\mathbb{D}$, consider systems:

$$\begin{array}{lcll} \mathcal{I}[start] & \sqsupseteq & d_0 & \\ \mathcal{I}[v] & \sqsupseteq & [\![k]\!]^\sharp\,(\mathcal{I}[u]) & \qquad k = (u, \_, v) \quad \text{edge} \end{array}$$

where $d_0 \in \mathbb{D}$ and all $[\![k]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ are monotonic ...

$\Longrightarrow$ Monotonic Analysis Framework

Wanted: MOP (Merge Over all Paths)

$$\mathcal{I}^*[v] = \bigsqcup \{ [\![\pi]\!]^\sharp\, d_0 \mid \pi : \mathit{start} \to^* v \}$$

Wanted: MOP (Merge Over all Paths)

$$\mathcal{I}^*[v] = \bigsqcup\{[\![\pi]\!]^\sharp\, d_0 \mid \pi : \mathit{start} \to^* v\}$$

Theorem

Kam, Ullman 1975

Assume $\mathcal{I}$ is a solution of the constraint system. Then:

$$\mathcal{I}[v] \;\sqsupseteq\; \mathcal{I}^*[v] \qquad\qquad \text{for every} \quad v$$

Jeffrey D. Ullman, Stanford

Wanted: MOP (Merge Over all Paths)

$$\mathcal{I}^*[v] = \bigsqcup\{[\![\pi]\!]^\sharp\, d_0 \mid \pi : \mathit{start} \to^* v\}$$

Theorem Kam, Ullman 1975

Assume $\mathcal{I}$ is a solution of the constraint system. Then:

$$\mathcal{I}[v] \;\sqsupseteq\; \mathcal{I}^*[v] \qquad \text{for every} \quad v$$

In particular: $\mathcal{I}[v] \;\sqsupseteq\; [\![\pi]\!]^\sharp\, d_0$ for every $\pi : \mathit{start} \to^* v$

Proof: Induction on the length of π.

Proof: Induction on the length of π.

Foundation: $\pi = \epsilon$ (empty path)

Proof: Induction on the length of π.

Foundation: $\pi = \epsilon$ (empty path)

Then:

$$[\![\pi]\!]^\sharp\, d_0 = [\![\epsilon]\!]^\sharp\, d_0 = d_0 \sqsubseteq \mathcal{I}[\mathit{start}]$$

Proof: Induction on the length of π.

Foundation: $\pi = \epsilon$ (empty path)

Then:

$$[\![\pi]\!]^\sharp\, d_0 = [\![\epsilon]\!]^\sharp\, d_0 = d_0 \sqsubseteq \mathcal{I}[start]$$

Step: $\pi = \pi' k$ for $k = (u, \_, v)$ edge.

Proof: Induction on the length of π.

Foundation: $\pi = \epsilon$ (empty path)

Then:

$$[\![\pi]\!]^\sharp\, d_0 = [\![\epsilon]\!]^\sharp\, d_0 = d_0 \sqsubseteq \mathcal{I}[\mathit{start}]$$

Step: $\pi = \pi' k$ for $k = (u, \_, v)$ edge.

Then:

$$\begin{array}{llll} & [\![\pi']\!]^\sharp\, d_0 & \sqsubseteq & \mathcal{I}[u] & \text{by I.H. for } \pi \\ \Longrightarrow & [\![\pi]\!]^\sharp\, d_0 & = & [\![k]\!]^\sharp\,([\![\pi']\!]^\sharp\, d_0) & \\ & & \sqsubseteq & [\![k]\!]^\sharp\,(\mathcal{I}[u]) & \text{since } [\![k]\!]^\sharp \text{ monotonic} \\ & & \sqsubseteq & \mathcal{I}[v] & \text{since } \mathcal{I} \text{ solution} \quad :\text{-))} \end{array}$$

Disappointment:

Are solutions of the constraint system just upper bounds ???

Disappointment:

Are solutions of the constraint system just upper bounds ???

Answer:

In general: yes :-(

Disappointment:

Are solutions of the constraint system just upper bounds ???

Answer:

In general: yes :-(

With the notable exception when all functions $[\![k]\!]^\sharp$ are distributive ... :-)

The function $f : \mathbb{D}_1 \longrightarrow \mathbb{D}_2$ is called

- distributive, if $f\,(\bigsqcup X) = \bigsqcup\{f\,x \mid x \in X\}$ for all $\emptyset \neq X \subseteq \mathbb{D}$;
- strict, if $f\,\bot = \bot$.
- totally distributive, if f is distributive and strict.

The function $f : \mathbb{D}_1 \to \mathbb{D}_2$ is called

- distributive, if $f\,(\bigsqcup X) = \bigsqcup\{f\,x \mid x \in X\}$ for all $\emptyset \neq X \subseteq \mathbb{D}$;
- strict, if $f\,\bot = \bot$.
- totally distributive, if f is distributive and strict.

Examples:

- $f\,x = x \cap a \cup b$ for $a, b \subseteq U$.

The function $f : \mathbb{D}_1 \to \mathbb{D}_2$ is called

- distributive, if $f\,(\bigsqcup X) = \bigsqcup\{f\,x \mid x \in X\}$ for all $\emptyset \neq X \subseteq \mathbb{D}$;
- strict, if $f\,\bot = \bot$.
- totally distributive, if f is distributive and strict.

Examples:

- $f\,x = x \cap a \cup b$ for $a, b \subseteq U$.

  Strictness: $f\,\emptyset = a \cap \emptyset \cup b = b = \emptyset$ whenever $b = \emptyset$:-(

The function $f : \mathbb{D}_1 \to \mathbb{D}_2$ is called

- distributive, if $f\,(\bigsqcup X) = \bigsqcup\{f\,x \mid x \in X\}$ for all $\emptyset \neq X \subseteq \mathbb{D}$;
- strict, if $f\,\bot = \bot$.
- totally distributive, if f is distributive and strict.

Examples:

- $f\,x = x \cap a \cup b$ for $a, b \subseteq U$.

Strictness: $f\,\emptyset = a \cap \emptyset \cup b = b = \emptyset$ whenever $b = \emptyset$:-(

Distributivity:

$$\begin{array}{lll} f\,(x_1 \cup x_2) & = & a \cap (x_1 \cup x_2) \cup b \\ & = & a \cap x_1 \cup a \cap x_2 \cup b \\ & = & f\,x_1 \cup f\,x_2 \qquad\qquad \text{:-)} \end{array}$$

- $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}$, $\quad \mathsf{inc}\, x = x + 1$

- $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}$, $\quad \mathsf{inc}\, x = x + 1$

  Strictness: $\quad f\, \bot = \mathsf{inc}\, 0 = 1 \quad \neq \quad \bot \quad$:-(

- $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad \mathsf{inc}\, x = x + 1$

  Strictness: $\quad f\, \bot = \mathsf{inc}\, 0 = 1 \quad \neq \quad \bot$:-(

  Distributivity: $\quad f\, (\bigsqcup X) \quad = \quad \bigsqcup\{x + 1 \mid x \in X\}$ for $\emptyset \neq X$:-)

- $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad \mathsf{inc}\, x = x + 1$

  Strictness: $f\, \bot = \mathsf{inc}\, 0 = 1 \quad \neq \quad \bot$:-(

  Distributivity: $f\, (\bigsqcup X) \quad = \quad \bigsqcup\{x + 1 \mid x \in X\}$ for $\emptyset \neq X$:-)

- $\mathbb{D}_1 = (\mathbb{N} \cup \{\infty\})^2, \quad \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad f\,(x_1, x_2) = x_1 + x_2$

- $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad \mathsf{inc}\, x = x + 1$

  Strictness: $f\,\bot = \mathsf{inc}\, 0 = 1 \quad \neq \quad \bot$:-(

  Distributivity: $f\,(\bigsqcup X) \quad = \quad \bigsqcup\{x+1 \mid x \in X\}$ for $\emptyset \neq X$:-)

- $\mathbb{D}_1 = (\mathbb{N} \cup \{\infty\})^2, \quad \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad f\,(x_1, x_2) = x_1 + x_2 :$

  Strictness: $f\,\bot = 0 + 0 \quad = \quad 0$:-)

- $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad \mathsf{inc}\, x = x + 1$

  Strictness: $\quad f\,\bot = \mathsf{inc}\, 0 = 1 \quad \neq \quad \bot \quad$:-(

  Distributivity: $\quad f\,(\bigsqcup X) \quad = \quad \bigsqcup\{x + 1 \mid x \in X\}$ for $\emptyset \neq X$:-)

- $\mathbb{D}_1 = (\mathbb{N} \cup \{\infty\})^2, \quad \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad f(x_1, x_2) = x_1 + x_2:$

  Strictness: $\quad f\,\bot = 0 + 0 \quad = \quad 0 \quad$:-)

  Distributivity:

$$\begin{array}{lllll} f\,((1,4) \sqcup (4,1)) & = & f\,(4,4) & = & 8 \\ & \neq & 5 & = & f\,(1,4) \sqcup f\,(4,1) \end{array}$$

:-)

Remark:

If $f : \mathbb{D}_1 \longrightarrow \mathbb{D}_2$ is distributive, then also monotonic :-)

Remark:

If $f : \mathbb{D}_1 \longrightarrow \mathbb{D}_2$ is distributive, then also monotonic :-)

Obviously: $a \sqsubseteq b$ iff $a \sqcup b = b$.

Remark:

If $f : \mathbb{D}_1 \to \mathbb{D}_2$ is distributive, then also monotonic :-)

Obviously: $a \sqsubseteq b$ iff $a \sqcup b = b$.

From that follows:

$$\begin{array}{rrcl} & f\,b & = & f\,(a \sqcup b) \\ & & = & f\,a \sqcup f\,b \\ \Longrightarrow & f\,a & \sqsubseteq & f\,b \qquad\qquad :\text{-)} \end{array}$$

Assumption: all v are reachable from start .

Assumption: all v are reachable from *start* .
Then:

Theorem Kildall 1972

If all effects of edges $[\![k]\!]^\sharp$ are distributive, then: $\mathcal{I}^*[v] = \mathcal{I}[v]$
for all v .

Gary A. Kildall (1942-1994).

Has developed the operating system CP/M and GUIs for PCs.

Assumption: all v are reachable from *start* .
Then:

Theorem Kildall 1972

If all effects of edges $[\![k]\!]^\sharp$ are distributive, then: $\mathcal{I}^*[v] = \mathcal{I}[v]$
for all v .

Assumption: all v are reachable from *start* .
Then:

Theorem Kildall 1972

If all effects of edges $[\![k]\!]^\sharp$ are distributive, then: $\mathcal{I}^*[v] = \mathcal{I}[v]$ for all v .

Proof:

It suffices to prove that $\mathcal{I}^*$ is a solution :-)

For this, we show that $\mathcal{I}^*$ satisfies all constraints :-))

(1) We prove for *start* :

$$\begin{aligned} \mathcal{I}^*[\mathit{start}] &= \bigsqcup\{[\![\pi]\!]^\sharp\, d_0 \mid \pi : \mathit{start} \to^* \mathit{start}\} \\ &\sqsupseteq [\![\epsilon]\!]^\sharp\, d_0 \\ &\sqsupseteq d_0 \qquad \text{:-)} \end{aligned}$$

(1) We prove for $start$:

$$\begin{array}{lll}
\mathcal{I}^*[start] & = & \bigsqcup\{[\![\pi]\!]^\sharp\, d_0 \mid \pi : start \rightarrow^* start\} \\
& \sqsupseteq & [\![\epsilon]\!]^\sharp\, d_0 \\
& \sqsupseteq & d_0 \qquad \text{:-)}
\end{array}$$

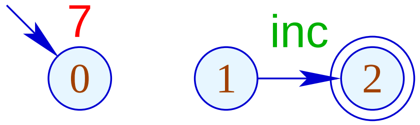
(2) For every $k = (u, \_, v)$ we prove:

$$\begin{array}{lll}
\mathcal{I}^*[v] & = & \bigsqcup\{[\![\pi]\!]^\sharp\, d_0 \mid \pi : start \rightarrow^* v\} \\
& \sqsupseteq & \bigsqcup\{[\![\pi' k]\!]^\sharp\, d_0 \mid \pi' : start \rightarrow^* u\} \\
& = & \bigsqcup\{[\![k]\!]^\sharp\,([\![\pi']\!]^\sharp\, d_0) \mid \pi' : start \rightarrow^* u\} \\
& = & [\![k]\!]^\sharp\,(\bigsqcup\{[\![\pi']\!]^\sharp\, d_0 \mid \pi' : start \rightarrow^* u\}) \\
& = & [\![k]\!]^\sharp\,(\mathcal{I}^*[u])
\end{array}$$

since $\{\pi' \mid \pi' : start \rightarrow^* u\}$ is non-empty :-)

Warning:

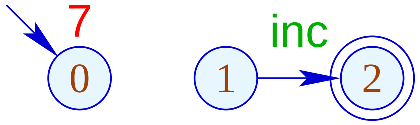
- Reachability of all program points cannot be abandoned! Consider:



where $\mathbb{D} = \mathbb{N} \cup \{\infty\}$

Warning:

- Reachability of all program points cannot be abandoned! Consider:



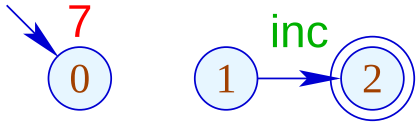
where $\mathbb{D} = \mathbb{N} \cup \{\infty\}$

Then:

$$\begin{array}{lllll} \mathcal{I}[2] & = & \mathsf{inc}\, 0 & = & 1 \\ \mathcal{I}^*[2] & = & \bigsqcup \emptyset & = & 0 \end{array}$$

Warning:

- Reachability of all program points cannot be abandoned! Consider:



where $\mathbb{D} = \mathbb{N} \cup \{\infty\}$

Then:

$$\begin{array}{lllll} \mathcal{I}[2] & = & \mathsf{inc}\,0 & = & 1 \\ \mathcal{I}^*[2] & = & \bigsqcup \emptyset & = & 0 \end{array}$$

- Unreachable program points can always be thrown away :-)

Summary and Application:

→ The effects of edges of the analysis of availability of expressions are distributive:

$$\begin{aligned}(a \cup (x_1 \cap x_2))\backslash b &= ((a \cup x_1) \cap (a \cup x_2))\backslash b \\ &= ((a \cup x_1)\backslash b) \cap ((a \cup x_2)\backslash b)\end{aligned}$$

Summary and Application:

→ The effects of edges of the analysis of availability of expressions are distributive:

$$\begin{aligned}(a \cup (x_1 \cap x_2))\backslash b &= ((a \cup x_1) \cap (a \cup x_2))\backslash b \\ &= ((a \cup x_1)\backslash b) \cap ((a \cup x_2)\backslash b)\end{aligned}$$

→ If all effects of edges are distributive, then the MOP can be computed by means of the constraint system and RR-iteration. :-)

Summary and Application:

→ The effects of edges of the analysis of availability of expressions are distributive:

$$\begin{aligned}(a \cup (x_1 \cap x_2)) \backslash b &= ((a \cup x_1) \cap (a \cup x_2)) \backslash b \\ &= ((a \cup x_1) \backslash b) \cap ((a \cup x_2) \backslash b)\end{aligned}$$

→ If all effects of edges are distributive, then the MOP can be computed by means of the constraint system and RR-iteration. :-)

→ If not all effects of edges are distributive, then RR-iteration for the constraint system at least returns a safe upper bound to the MOP :-)

1.2 Removing Assignments to Dead Variables

Example:

$$\begin{array}{ll} 1: & x = y + 2; \\ 2: & y = 5; \\ 3: & x = y + 3; \end{array}$$

The value of x at program points 1, 2 is over-written before it can be used.

Therefore, we call the variable x dead at these program points :-)

Note:

→ Assignments to dead variables can be removed ;-)

→ Such inefficiencies may originate from other transformations.

Note:

→ Assignments to dead variables can be removed ;-)

→ Such inefficiencies may originate from other transformations.

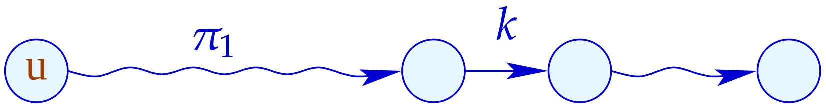
Formal Definition:

The variable x is called live at u along the path π starting at u relative to a set X of variables either:

if $x \in X$ and π does not contain a definition of x; or:

if π can be decomposed into: $\pi = \pi_1 \, k \, \pi_2$ such that:

- k is a use of x ; and
- π_1 does not contain a definition of x.

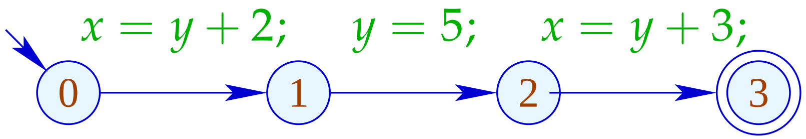


Thereby, the set of all defined or used variables at an edge $k = (\_, lab, \_)$ is defined by:

lab	used	defined
$;$	$\emptyset$	$\emptyset$
$\text{Pos}\,(e)$	$Vars\,(e)$	$\emptyset$
$\text{Neg}\,(e)$	$Vars\,(e)$	$\emptyset$
$x = e;$	$Vars\,(e)$	$\{x\}$
$x = M[e];$	$Vars\,(e)$	$\{x\}$
$M[e_1] = e_2;$	$Vars\,(e_1) \cup Vars\,(e_2)$	$\emptyset$

A variable x which is not live at u along π (relative to X) is called dead at u along π (relative to X).

Example:



where $X = \emptyset$. Then we observe:

	live	dead
0	$\{y\}$	$\{x\}$
1	$\emptyset$	$\{x, y\}$
2	$\{y\}$	$\{x\}$
3	$\emptyset$	$\{x, y\}$

The variable x is live at u (relative to X) if x is live at u along some path to the exit (relative to X). Otherwise, x is called dead at u (relative to X).

The variable x is live at u (relative to X) if x is live at u along some path to the exit (relative to X). Otherwise, x is called dead at u (relative to X).

Question:

How can the sets of all dead/live variables be computed for every u ???

The variable x is live at u (relative to X) if x is live at u along some path to the exit (relative to X). Otherwise, x is called dead at u (relative to X).

Question:

How can the sets of all dead/live variables be computed for every u ???

Idea:

For every edge $k = (u, \_, v)$, define a function $[\![k]\!]^\sharp$ which transforms the set of variables which are live at v into the set of variables which are live at u ...

Let $\quad \mathbb{L} = 2^{Vars}$.

For $\quad k = (\_, lab, \_)$, define $\quad [\![k]\!]^\sharp = [\![lab]\!]^\sharp \quad$ by:

$$
\begin{array}{lcl}
[\![;]\!]^\sharp \, L & = & L \\
[\![\text{Pos}(e)]\!]^\sharp \, L & = & [\![\text{Neg}(e)]\!]^\sharp \, L \quad = \quad L \cup Vars(e) \\
[\![x = e;]\!]^\sharp \, L & = & (L \backslash \{x\}) \cup Vars(e) \\
[\![x = M[e];]\!]^\sharp \, L & = & (L \backslash \{x\}) \cup Vars(e) \\
[\![M[e_1] = e_2;]\!]^\sharp \, L & = & L \cup Vars(e_1) \cup Vars(e_2)
\end{array}
$$

Let $\mathbb{L} = 2^{Vars}$.

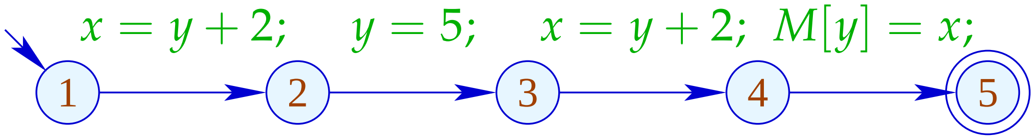
For $k = (\_, lab, \_)$, define $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ by:

$$\begin{array}{lcl}
[\![;]\!]^\sharp\, L & = & L \\
[\![\text{Pos}(e)]\!]^\sharp\, L & = & [\![\text{Neg}(e)]\!]^\sharp\, L \quad = \quad L \cup Vars(e) \\
[\![x = e;]\!]^\sharp\, L & = & (L \backslash \{x\}) \cup Vars(e) \\
[\![x = M[e];]\!]^\sharp\, L & = & (L \backslash \{x\}) \cup Vars(e) \\
[\![M[e_1] = e_2;]\!]^\sharp\, L & = & L \cup Vars(e_1) \cup Vars(e_2)
\end{array}$$

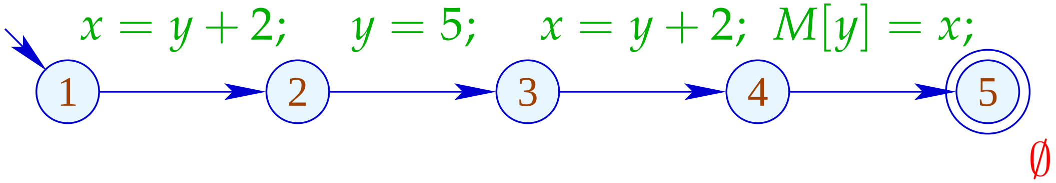
$[\![k]\!]^\sharp$ can again be composed to the effects of $[\![\pi]\!]^\sharp$ of paths $\pi = k_1 \ldots k_r$ by:

$$[\![\pi]\!]^\sharp = [\![k_1]\!]^\sharp \circ \ldots \circ [\![k_r]\!]^\sharp$$

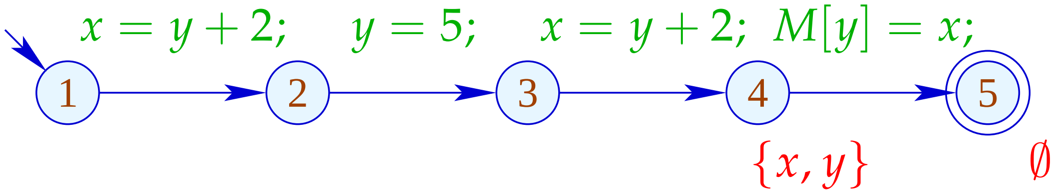
We verify that these definitions are meaningful :-)



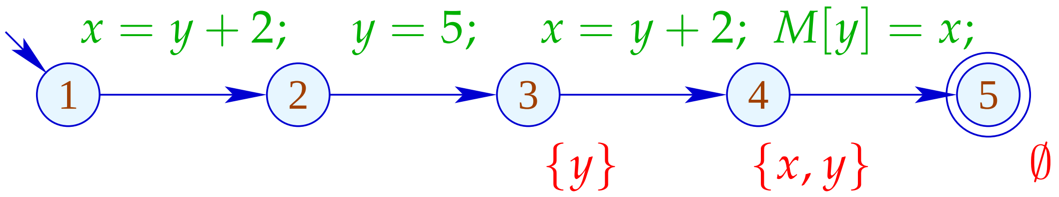
We verify that these definitions are meaningful :-)



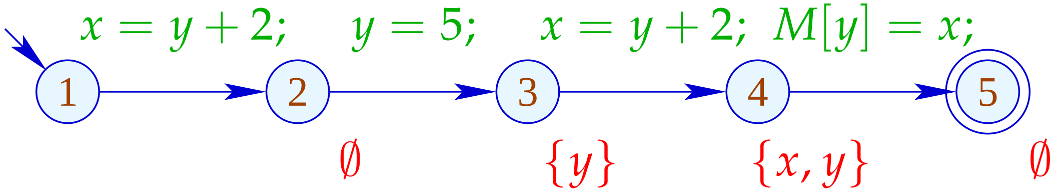
We verify that these definitions are meaningful :-)



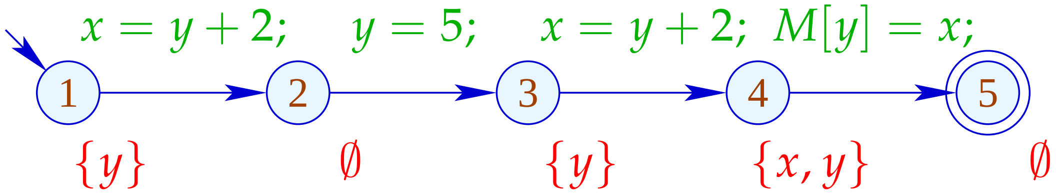
We verify that these definitions are meaningful :-)



We verify that these definitions are meaningful :-)



We verify that these definitions are meaningful :-)



The set of variables which are live at u then is given by:

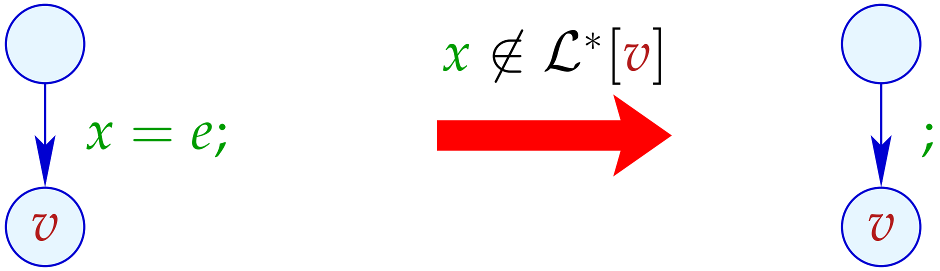
$$\mathcal{L}^*[u] \;=\; \bigcup\{[\![\pi]\!]^\sharp\, X \mid \pi : u \rightarrow^* stop\}$$

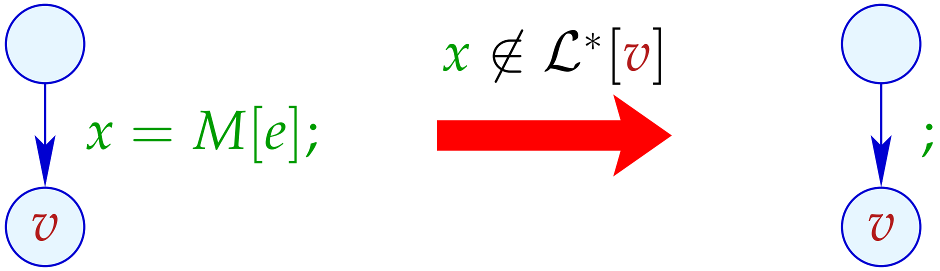
... literally:

- The paths start in u :-)

  $\Longrightarrow$ As partial ordering for $\mathbb{L}$ we use $\sqsubseteq \,=\, \subseteq$.
- The set of variables which are live at program exit is given by the set X :-)

Transformation 2:





Correctness Proof:

→ Correctness of the effects of edges: If L is the set of variables which are live at the exit of the path π , then $[\![\pi]\!]^\sharp L$ is the set of variables which are live at the beginning of π :-)

→ Correctness of the transformation along a path: If the value of a variable is accessed, this variable is necessarily live. The value of dead variables thus is irrelevant :-)

→ Correctness of the transformation: In any execution of the transformed programs, the live variables always receive the same values :-))

Computation of the sets $\mathcal{L}^*[u]$:

(1) Collecting constraints:

$$\begin{array}{lll} \mathcal{L}[stop] & \supseteq & X \\ \mathcal{L}[u] & \supseteq & [\![k]\!]^\sharp\,(\mathcal{L}[v]) \qquad k = (u, \_, v) \quad \text{edge} \end{array}$$

(2) Solving the constraint system by means of RR iteration.

Since $\mathbb{L}$ is finite, the iteration will terminate :-)

(3) If the exit is (formally) reachable from every program point, then the smallest solution $\mathcal{L}$ of the constraint system equals $\mathcal{L}^*$ since all $[\![k]\!]^\sharp$ are distributive :-))

Computation of the sets $\mathcal{L}^*[u]$:

(1) Collecting constraints:

$$\begin{array}{lll} \mathcal{L}[stop] & \supseteq & X \\ \mathcal{L}[u] & \supseteq & [\![k]\!]^\sharp\,(\mathcal{L}[v]) \qquad k = (u, \_, v) \quad \text{edge} \end{array}$$

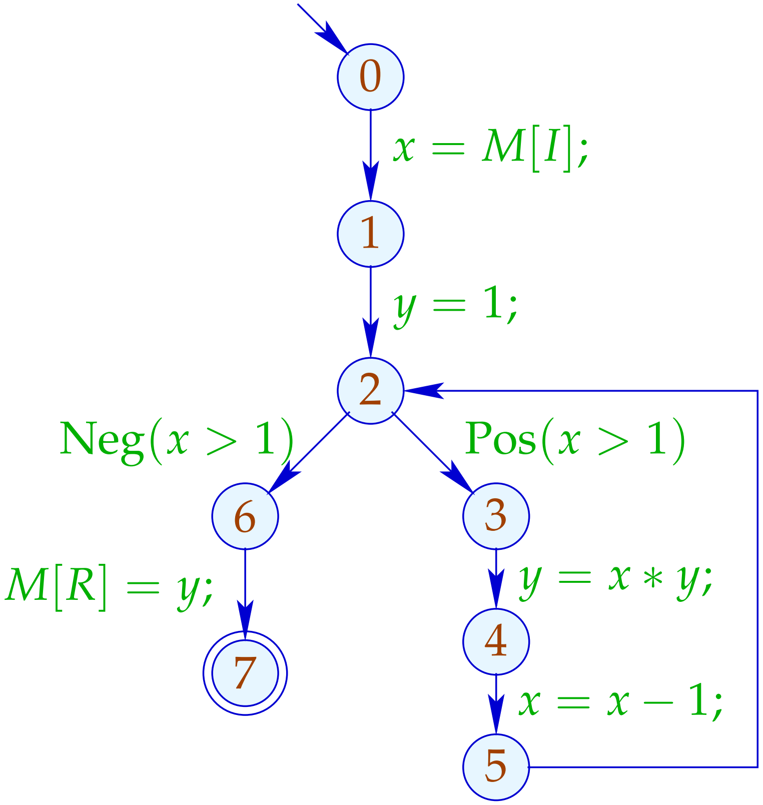
(2) Solving the constraint system by means of RR iteration.

Since $\mathbb{L}$ is finite, the iteration will terminate :-)

(3) If the exit is (formally) reachable from every program point, then the smallest solution $\mathcal{L}$ of the constraint system equals $\mathcal{L}^*$ since all $[\![k]\!]^\sharp$ are distributive :-))

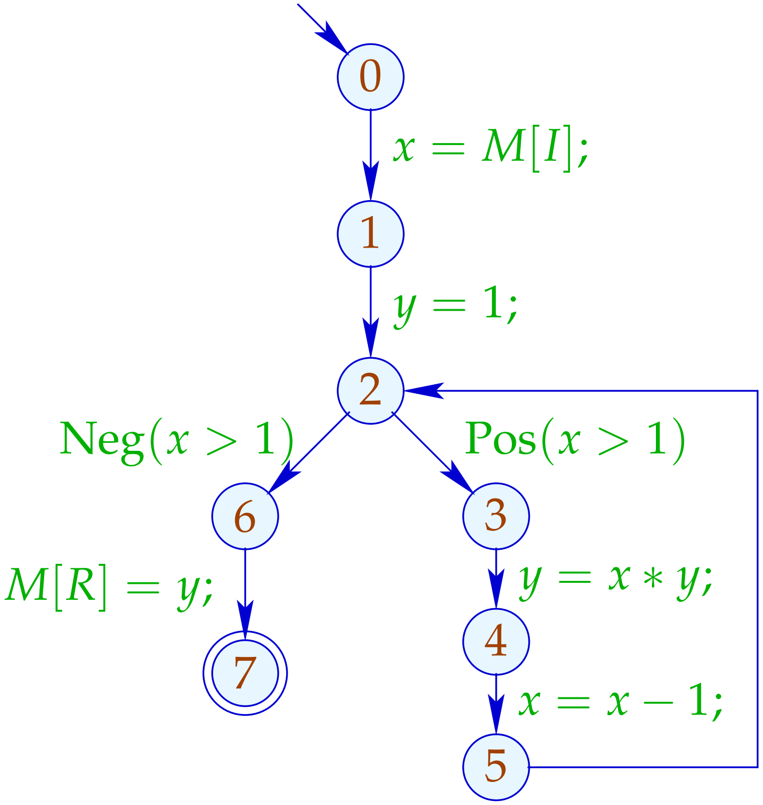
Warning: The information is propagated backwards !!!

Example:



$$
\begin{array}{lll}
\mathcal{L}[0] & \supseteq & (\mathcal{L}[1] \backslash \{x\}) \cup \{I\} \\
\mathcal{L}[1] & \supseteq & \mathcal{L}[2] \backslash \{y\} \\
\mathcal{L}[2] & \supseteq & (\mathcal{L}[6] \cup \{x\}) \cup (\mathcal{L}[3] \cup \{x\}) \\
\mathcal{L}[3] & \supseteq & (\mathcal{L}[4] \backslash \{y\}) \cup \{x, y\} \\
\mathcal{L}[4] & \supseteq & (\mathcal{L}[5] \backslash \{x\}) \cup \{x\} \\
\mathcal{L}[5] & \supseteq & \mathcal{L}[2] \\
\mathcal{L}[6] & \supseteq & \mathcal{L}[7] \cup \{y, R\} \\
\mathcal{L}[7] & \supseteq & \emptyset
\end{array}
$$

Example:

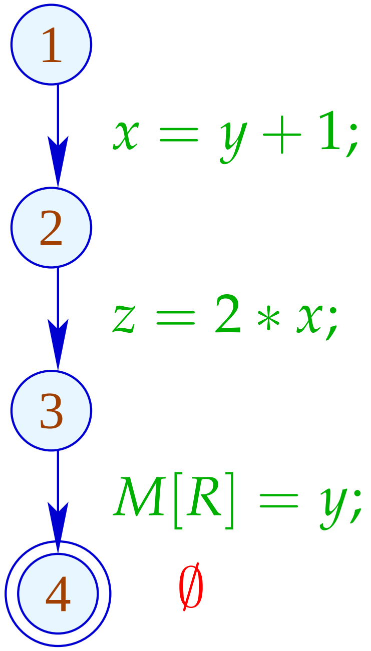


	1	2
7	$\emptyset$	
6	$\{y, R\}$	
2	$\{x, y, R\}$	dito
5	$\{x, y, R\}$	
4	$\{x, y, R\}$	
3	$\{x, y, R\}$	
1	$\{x, R\}$	
0	$\{I, R\}$	

The left-hand side of no assignment is dead :-)

Warning:

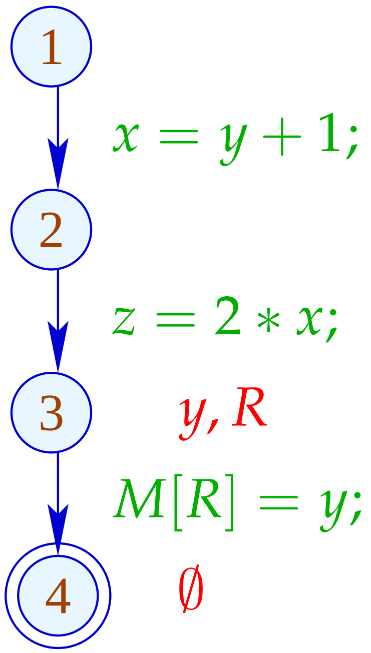
Removal of assignments to dead variables may kill further variables:



The left-hand side of no assignment is dead :-)

Warning:

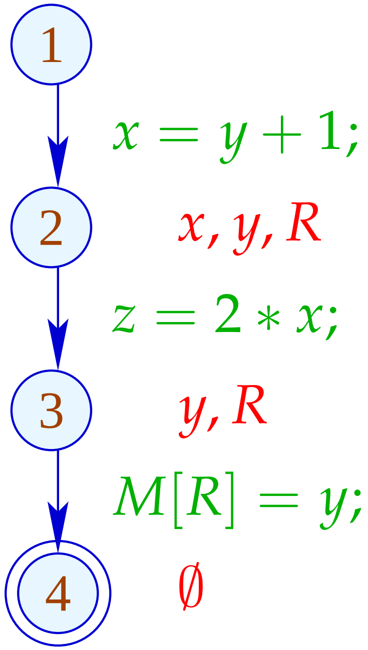
Removal of assignments to dead variables may kill further variables:



The left-hand side of no assignment is dead :-)

Warning:

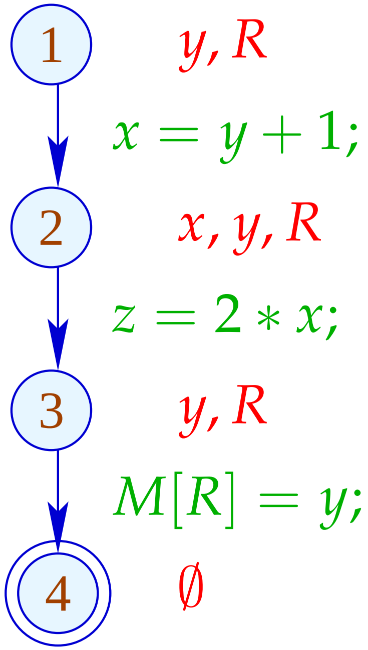
Removal of assignments to dead variables may kill further variables:



The left-hand side of no assignment is dead :-)

Warning:

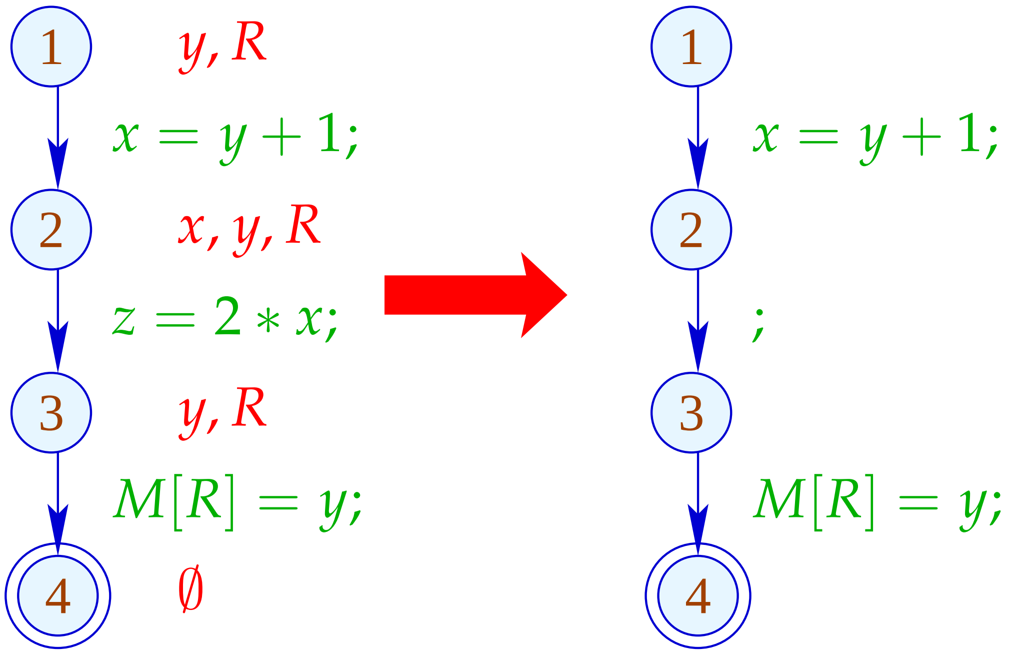
Removal of assignments to dead variables may kill further variables:



The left-hand side of no assignment is dead :-)

Warning:

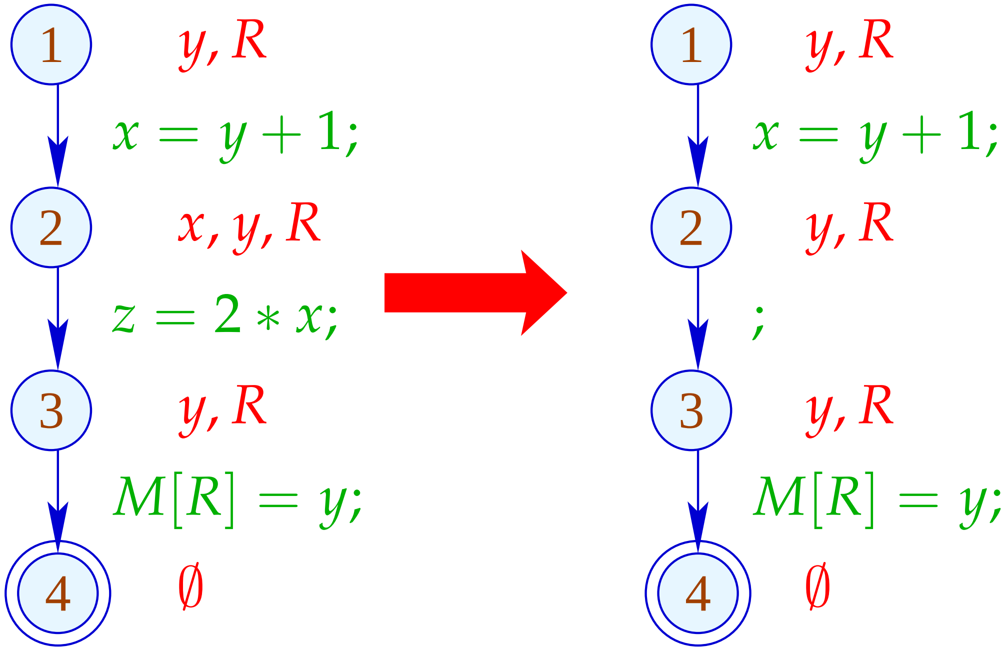
Removal of assignments to dead variables may kill further variables:



The left-hand side of no assignment is dead :-)

Warning:

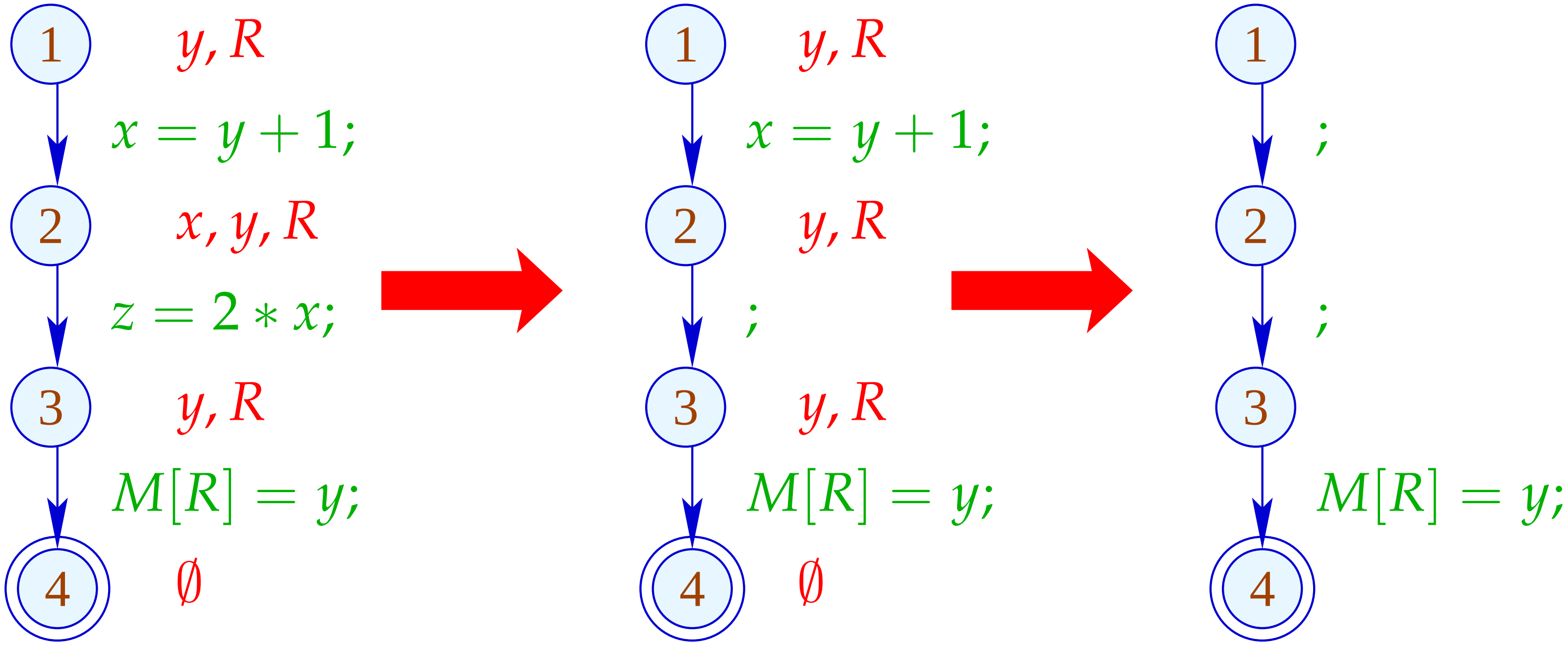
Removal of assignments to dead variables may kill further variables:



The left-hand side of no assignment is dead :-)

Warning:

Removal of assignments to dead variables may kill further variables:



Re-analyzing the program is inconvenient :-(

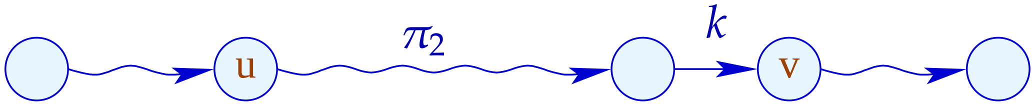
Idea: Analyze true liveness!

x is called truely live at u along a path π (relative to X), either

if $x \in X$, π does not contain a definition of x; or

if π can be decomposed into $\pi = \pi_1\, k\, \pi_2$ such that:

- k is a true use of x ;
- π_1 does not contain any definition of x.

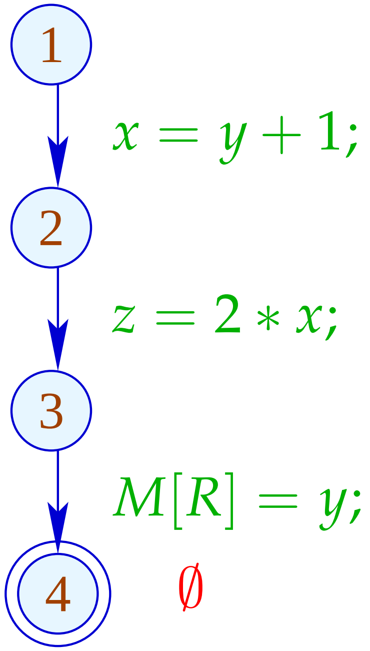


The set of truely used variables at an edge $k = (\_, lab, v)$ is defined as:

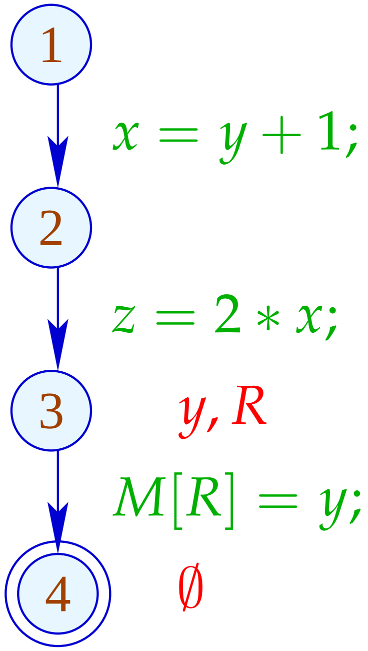
lab	truely used
;	$\emptyset$
Pos (e)	$Vars\,(e)$
Neg (e)	$Vars\,(e)$
$x = e;$	$Vars\,(e)$ $(*)$
$x = M[e];$	$Vars\,(e)$ $(*)$
$M[e_1] = e_2;$	$Vars(e_1) \cup Vars(e_2)$

$(*)$ – given that x is truely live at v :-)

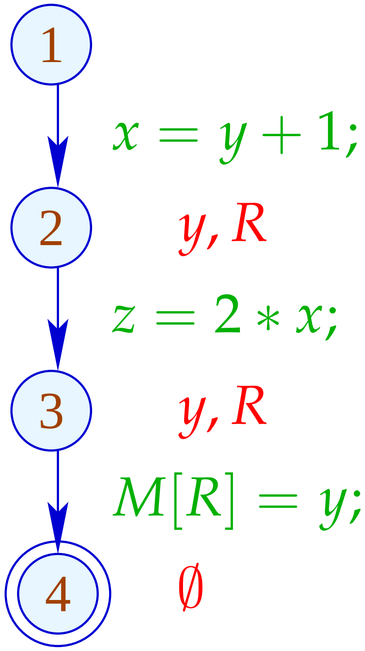
Example:



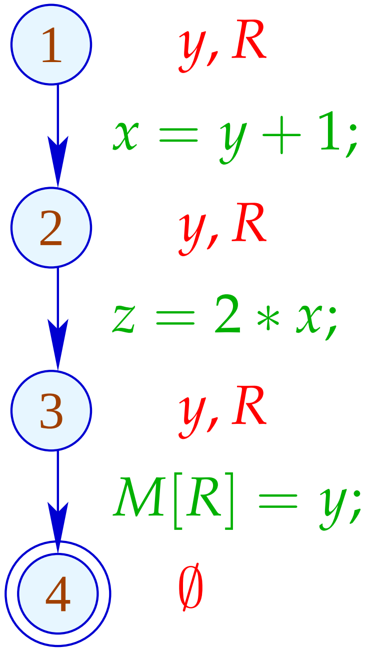
Example:



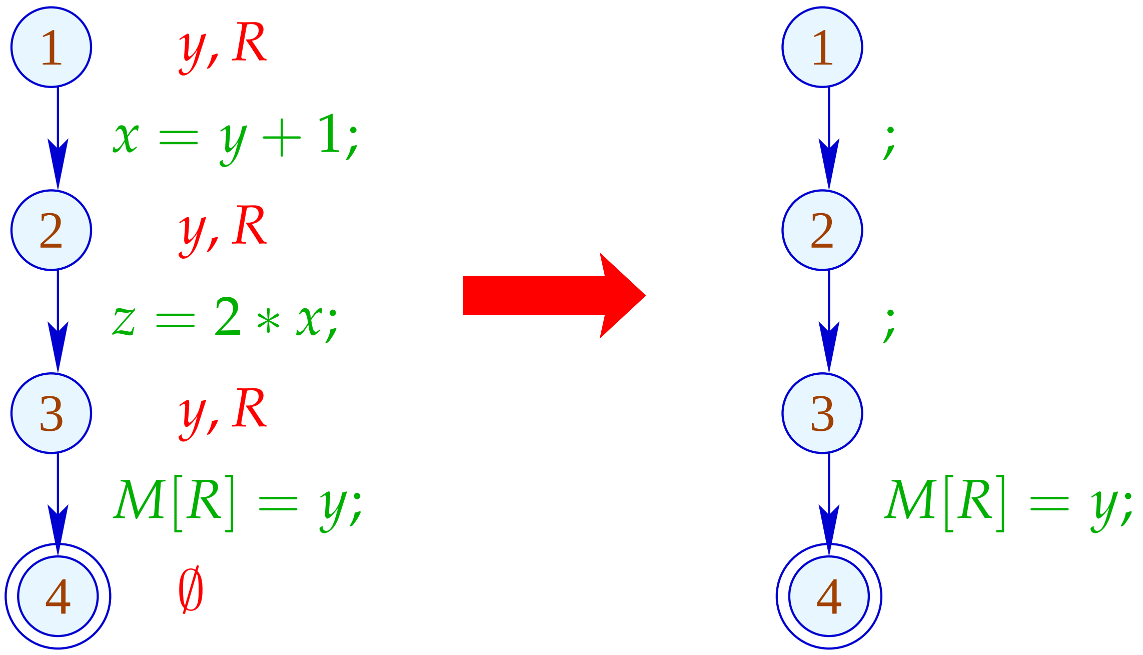
Example:



Example:



Example:



The Effects of Edges:

$$\begin{array}{lclcl}
[\![;]\!]^\sharp\, L & = & L & & \\
[\![\mathrm{Pos}(e)]\!]^\sharp\, L & = & [\![\mathrm{Neg}(e)]\!]^\sharp\, L & = & L \cup \mathit{Vars}(e) \\
[\![x = e;]\!]^\sharp\, L & = & (L \backslash \{x\}) \cup & & \mathit{Vars}(e) \\
[\![x = M[e];]\!]^\sharp\, L & = & (L \backslash \{x\}) \cup & & \mathit{Vars}(e) \\
[\![M[e_1] = e_2;]\!]^\sharp\, L & = & L \cup \mathit{Vars}(e_1) \cup \mathit{Vars}(e_2) & &
\end{array}$$

The Effects of Edges:

$$\begin{array}{lll}
[\![;]\!]^\sharp\, L & = & L \\
[\![\mathrm{Pos}(e)]\!]^\sharp\, L & = & [\![\mathrm{Neg}(e)]\!]^\sharp\, L \quad = \quad L \cup \mathit{Vars}(e) \\
[\![x = e;]\!]^\sharp\, L & = & (L \backslash \{x\}) \cup \; (x \in L)\, ?\, \mathit{Vars}(e) : \; \emptyset \\
[\![x = M[e];]\!]^\sharp\, L & = & (L \backslash \{x\}) \cup \; (x \in L)\, ?\, \mathit{Vars}(e) : \; \emptyset \\
[\![M[e_1] = e_2;]\!]^\sharp\, L & = & L \cup \mathit{Vars}(e_1) \cup \mathit{Vars}(e_2)
\end{array}$$

Note:

- The effects of edges for truely live variables are more complicated than for live variables :-)
- Nonetheless, they are distributive !!

Note:

- The effects of edges for truely live variables are more complicated than for live variables :-)
- Nonetheless, they are distributive !!

  To see this, consider for $\mathbb{D} = 2^U$, $f\,y = (u \in y)\,?\,b : \emptyset$ We verify:

$$
\begin{aligned}
f\,(y_1 \cup y_2) &= (u \in y_1 \cup y_2)\,?\,b : \emptyset \\
&= (u \in y_1 \vee u \in y_2)\,?\,b : \emptyset \\
&= (u \in y_1)\,?\,b : \emptyset \cup (u \in y_2)\,?\,b : \emptyset \\
&= f\,y_1 \cup f\,y_2
\end{aligned}
$$

Note:

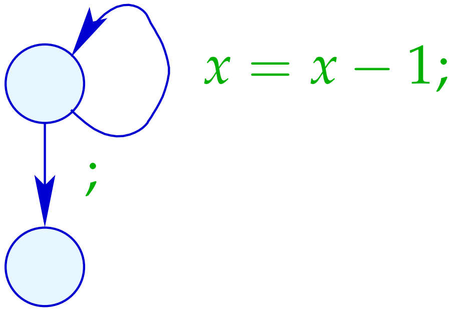
- The effects of edges for truely live variables are more complicated than for live variables :-)
- Nonetheless, they are distributive !!

  To see this, consider for $\mathbb{D} = 2^U$, $f\,y = (u \in y)\,?\,b : \emptyset$ We verify:

$$\begin{array}{lll} f\,(y_1 \cup y_2) & = & (u \in y_1 \cup y_2)\,?\,b : \emptyset \\ & = & (u \in y_1 \vee u \in y_2)\,?\,b : \emptyset \\ & = & (u \in y_1)\,?\,b : \emptyset \cup (u \in y_2)\,?\,b : \emptyset \\ & = & f\,y_1 \cup f\,y_2 \end{array}$$

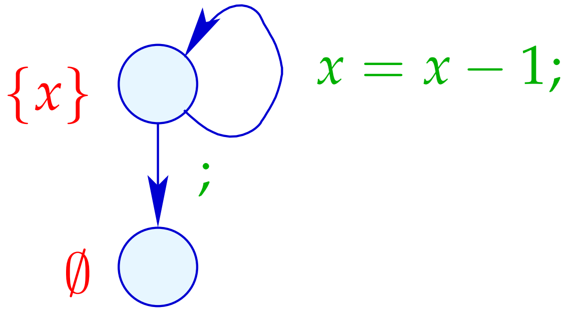
$\Longrightarrow$ the constraint system yields the MOP :-))

- True liveness detects more superfluous assignments than repeated liveness !!!



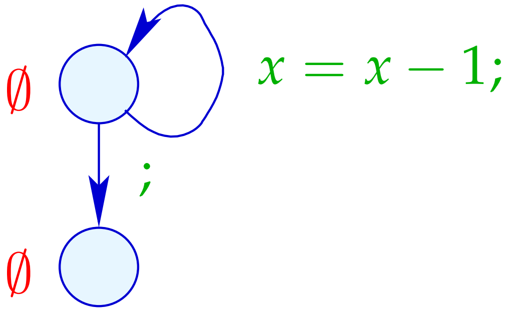
- True liveness detects more superfluous assignments than repeated liveness !!!

Liveness:



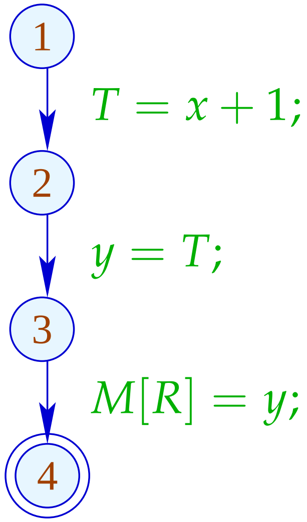
- True liveness detects more superfluous assignments than repeated liveness !!!

True Liveness:



1.3 Removing Superfluous Moves

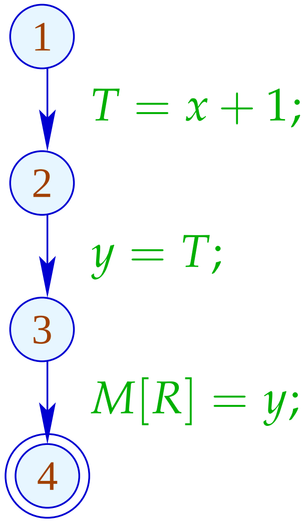
Example:



This variable-variable assignment is obviously useless :-(

1.3 Removing Superfluous Moves

Example:

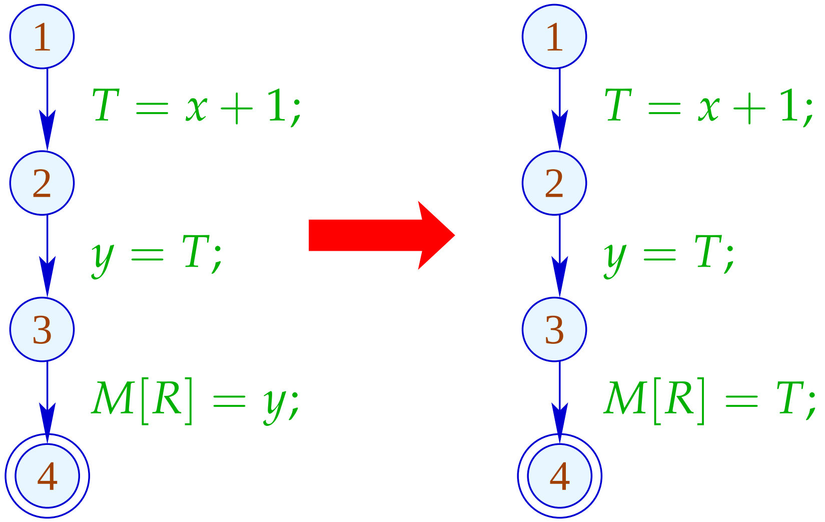


This variable-variable assignment is obviously useless :-(

Instead of y, we could also store T :-)

1.3 Removing Superfluous Moves

Example:

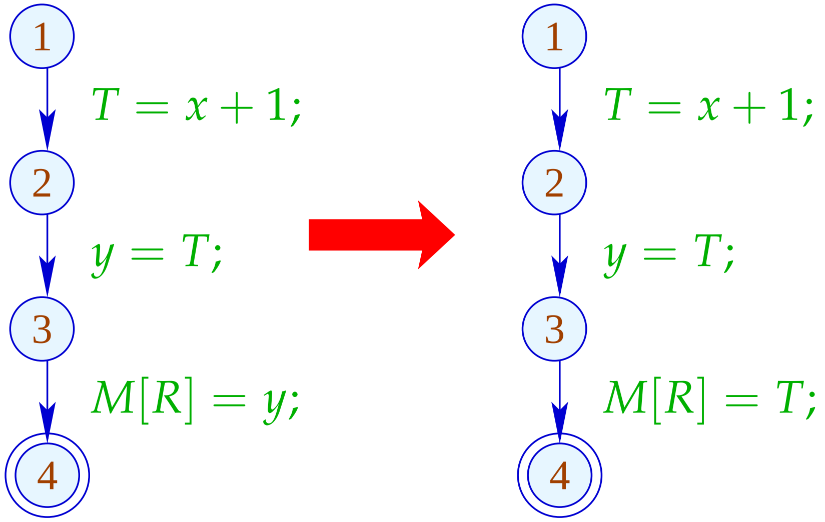


This variable-variable assignment is obviously useless :-(

Instead of y, we could also store T :-)

1.3 Removing Superfluous Moves

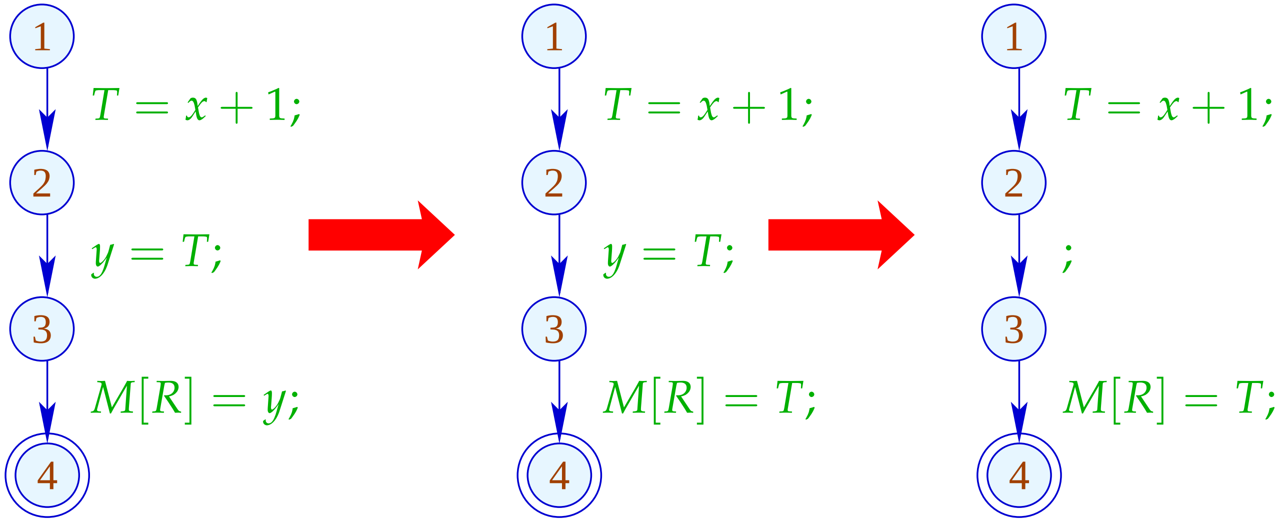
Example:



Advantage: Now, y has become dead :-))

1.3 Removing Superfluous Moves

Example:



Advantage: Now, y has become dead :-))

Idea:

For each expression, we record the variable which currently contains its value :-)

We use: $\mathbb{V} = \mathit{Expr} \to 2^{\mathit{Vars}}$ ...

Idea:

For each expression, we record the variable which currently contains its value :-)

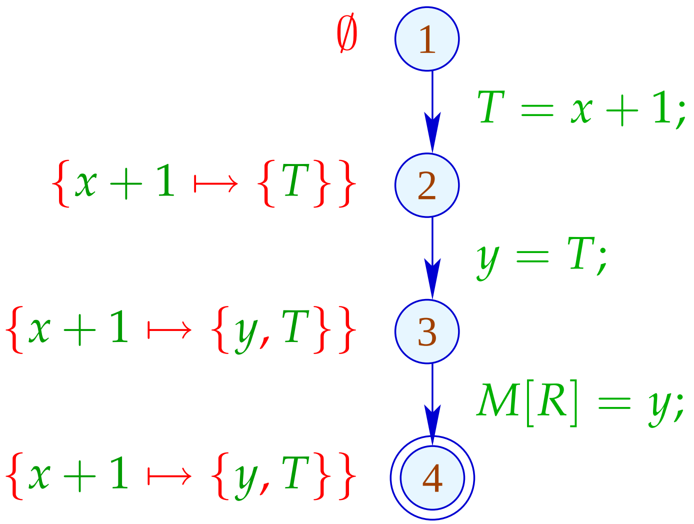
We use: $\mathbb{V} = \mathit{Expr} \to 2^{\mathit{Vars}}$ and define:

$$\begin{array}{lll}
[\![;]\!]^\sharp\, V & = & V \\
[\![\mathrm{Pos}(e)]\!]^\sharp\, V\, e' & = & [\![\mathrm{Neg}(e)]\!]^\sharp\, V\, e' \quad = \quad \begin{cases} \emptyset & \text{if } e' = e \\ V\, e' & \text{otherwise} \end{cases} \\
\ldots & &
\end{array}$$

$$
\begin{array}{lll}
[\![x = c;]\!]^\sharp\, V\, e' & = & \begin{cases} (V\, c) \cup \{x\} & \text{if} \quad e' = c \\ (V\, e')\backslash\{x\} & \text{otherwise} \end{cases} \\
[\![x = y;]\!]^\sharp\, V\, e & = & \begin{cases} (V\, e) \cup \{x\} & \text{if} \quad y \in V\, e \\ (V\, e)\backslash\{x\} & \text{otherwise} \end{cases} \\
[\![x = e;]\!]^\sharp\, V\, e' & = & \begin{cases} \{x\} & \text{if} \quad e' = e \\ (V\, e')\backslash\{x\} & \text{otherwise} \end{cases} \\
[\![x = M[c];]\!]^\sharp\, V\, e' & = & (V\, e')\backslash\{x\} \\
[\![x = M[y];]\!]^\sharp\, V\, e' & = & (V\, e')\backslash\{x\} \\
[\![x = M[e];]\!]^\sharp\, V\, e' & = & \begin{cases} \emptyset & \text{if } e' = e \\ (V\, e')\backslash\{x\} & \text{otherwise} \end{cases}
\end{array}
$$

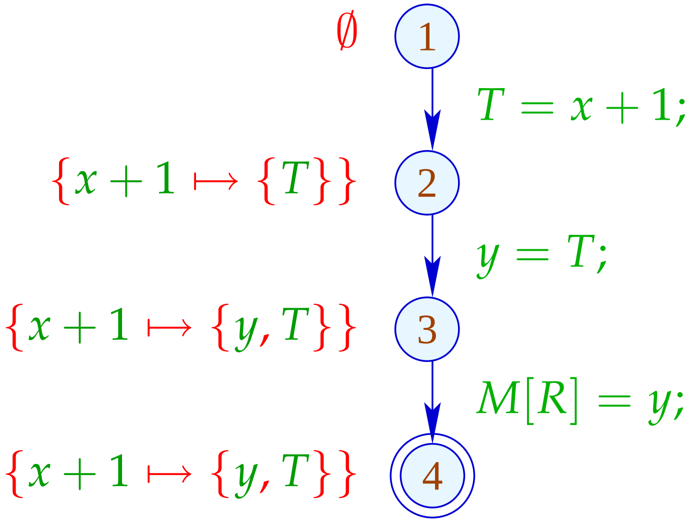
// analogously for the diverse stores

In the Example:



$\emptyset$
1
$T = x + 1;$
$\{x + 1 \mapsto \{T\}\}$
2
$y = T;$
$\{x + 1 \mapsto \{y, T\}\}$
3
$M[R] = y;$
$\{x + 1 \mapsto \{y, T\}\}$
4

In the Example:



$\emptyset$
1
$T = x + 1;$
$\{x + 1 \mapsto \{T\}\}$
2
$y = T;$
$\{x + 1 \mapsto \{y, T\}\}$
3
$M[R] = y;$
$\{x + 1 \mapsto \{y, T\}\}$
4

→ We propagate information in forward direction :-)

At *start* , $V_0\, e = \emptyset$ for all e;

→ $\sqsubseteq\; \subseteq\; \mathbb{V} \times \mathbb{V}$ is defined by:

$$V_1 \sqsubseteq V_2 \quad \text{iff} \quad V_1\, e \;\supseteq\; V_2\, e \qquad \text{for all} \quad e$$

Observation:

The new effects of edges are distributive:

To show this, we consider the functions:

(1) $f_1^x \, V \, e = (V \, e) \backslash \{x\}$

(2) $f_2^{e,a} \, V = V \oplus \{e \mapsto a\}\}$

(3) $f_3^{x,y} \, V \, e = (y \in V \, e) \, ? \, (V \, e \cup \{x\}) : ((V \, e) \backslash \{x\})$

Obviously, we have:

$$\begin{array}{lcl} [\![x = e;]\!]^\sharp & = & f_2^{e,\{x\}} \circ f_1^x \\ [\![x = y;]\!]^\sharp & = & f_3^{x,y} \\ [\![x = M[e];]\!]^\sharp & = & f_2^{e,\emptyset} \circ f_1^x \end{array}$$

By closure under composition, the assertion follows :-))

(1) For $f\, V\, e = (V\, e) \backslash \{x\}$, we have:

$$\begin{array}{lll} f\,(V_1 \sqcup V_2)\, e & = & ((V_1 \sqcup V_2)\, e) \backslash \{x\} \\ & = & ((V_1\, e) \cap (V_2\, e)) \backslash \{x\} \\ & = & ((V_1\, e) \backslash \{x\}) \cap ((V_2\, e) \backslash \{x\}) \\ & = & (f\, V_1\, e) \cap (f\, V_2\, e) \\ & = & (f\, V_1 \sqcup f\, V_2)\, e \qquad \text{:-)} \end{array}$$

(2) For $f\,V = V \oplus \{e \mapsto a\}$, we have:

$$\begin{array}{lll} f\,(V_1 \sqcup V_2)\,e' & = & ((V_1 \sqcup V_2) \oplus \{e \mapsto a\})\,e' \\ & = & (V_1 \sqcup V_2)\,e' \\ & = & (f\,V_1 \sqcup f\,V_2)\,e' \qquad \text{given that} \quad e \neq e' \end{array}$$

$$\begin{array}{lll} f\,(V_1 \sqcup V_2)\,e & = & ((V_1 \sqcup V_2) \oplus \{e \mapsto a\})\,e \\ & = & a \\ & = & ((V_1 \oplus \{e \mapsto a\})\,e) \sqcap ((V_2 \oplus \{e \mapsto a\})\,e) \\ & = & (f\,V_1 \sqcup f\,V_2)\,e \qquad \text{:-)} \end{array}$$

(3) For $f\,V\,e = (y \in V\,e)\,?\,(V\,e \cup \{x\}) : ((V\,e)\backslash\{x\})$, we have:

$$\begin{array}{lll}
f\,(V_1 \sqcup V_2)\,e & = & (((V_1 \sqcup V_2)\,e)\backslash\{x\}) \cup (y \in (V_1 \sqcup V_2)\,e)\,?\,\{x\} : \emptyset \\
 & = & ((V_1\,e \cap V_2\,e)\backslash\{x\}) \cup (y \in (V_1\,e \cap V_2\,e))\,?\,\{x\} : \emptyset \\
 & = & ((V_1\,e \cap V_2\,e)\backslash\{x\}) \cup \\
 & & ((y \in V_1\,e)\,?\,\{x\} : \emptyset) \cap ((y \in V_2\,e)\,?\,\{x\} : \emptyset) \\
 & = & (((V_1\,e)\backslash\{x\}) \cup (y \in V_1\,e)\,?\,\{x\} : \emptyset) \cap \\
 & & (((V_2\,e)\backslash\{x\}) \cup (y \in V_2\,e)\,?\,\{x\} : \emptyset) \\
 & = & (f\,V_1 \sqcup f\,V_2)\,e \qquad\qquad :\text{-})
\end{array}$$

We conclude:

→ Solving the constraint system returns the MOP solution :-)

→ Let $\mathcal{V}$ denote this solution.

If $x \in \mathcal{V}[u]\, e$, then x at u contains the value of e — which we have stored in T_e

$\Longrightarrow$

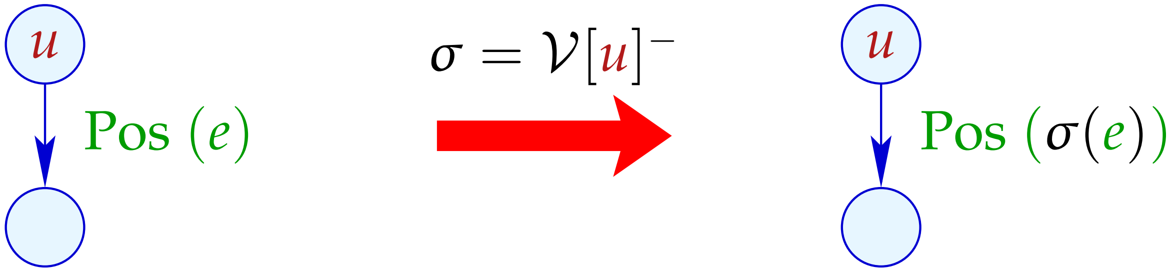
the access to x can be replaced by the access to T_e :-)

For $V \in \mathbb{V}$, let V^- denote the variable substitution with:

$$V^- \, x = \begin{cases} T_e & \text{if } x \in V\, e \\ x & \text{otherwise} \end{cases}$$

if $V\, e \cap V\, e' = \emptyset$ for $e \neq e'$. Otherwise: $V^- \, x = x$:-)

Transformation 3:



u

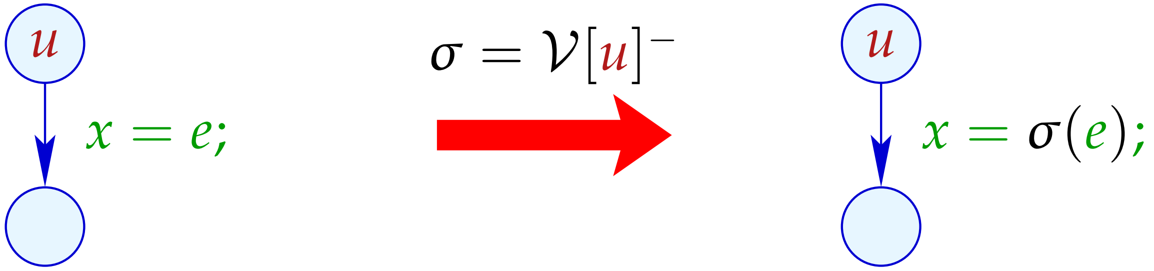
Pos (e)

$\sigma = \mathcal{V}[u]^{-}$

u

Pos $(\sigma(e))$

... analogously for edges with Neg (e)



u

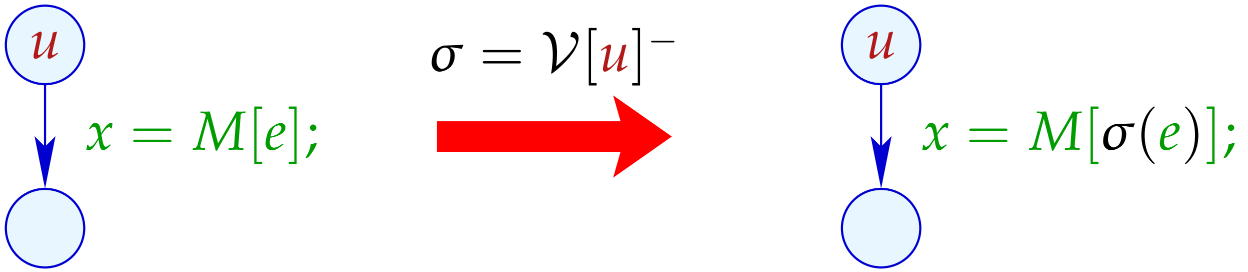
$x = e;$

$\sigma = \mathcal{V}[u]^{-}$

u

$x = \sigma(e);$

Transformation 3 (cont.):



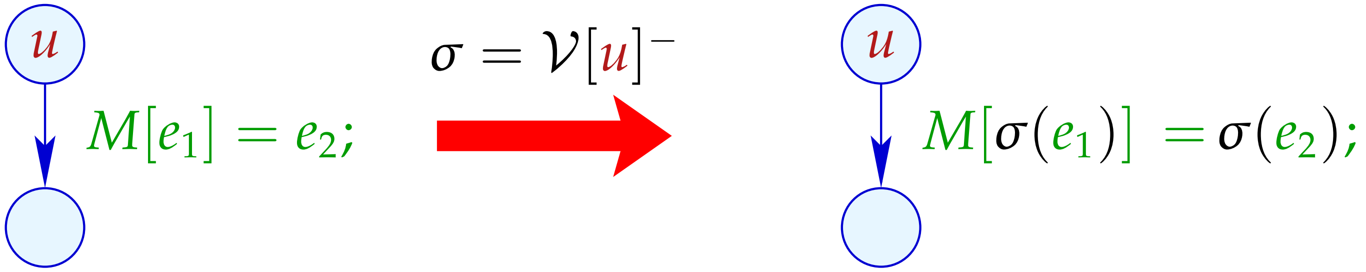
u

$x = M[e];$

$\sigma = \mathcal{V}[u]^{-}$

u

$x = M[\sigma(e)];$



u

$M[e_1] = e_2;$

$\sigma = \mathcal{V}[u]^{-}$

u

$M[\sigma(e_1)] = \sigma(e_2);$

Procedure as a whole:

(1) Availability of expressions: T1

- + removes arithmetic operations
- − inserts superfluous moves

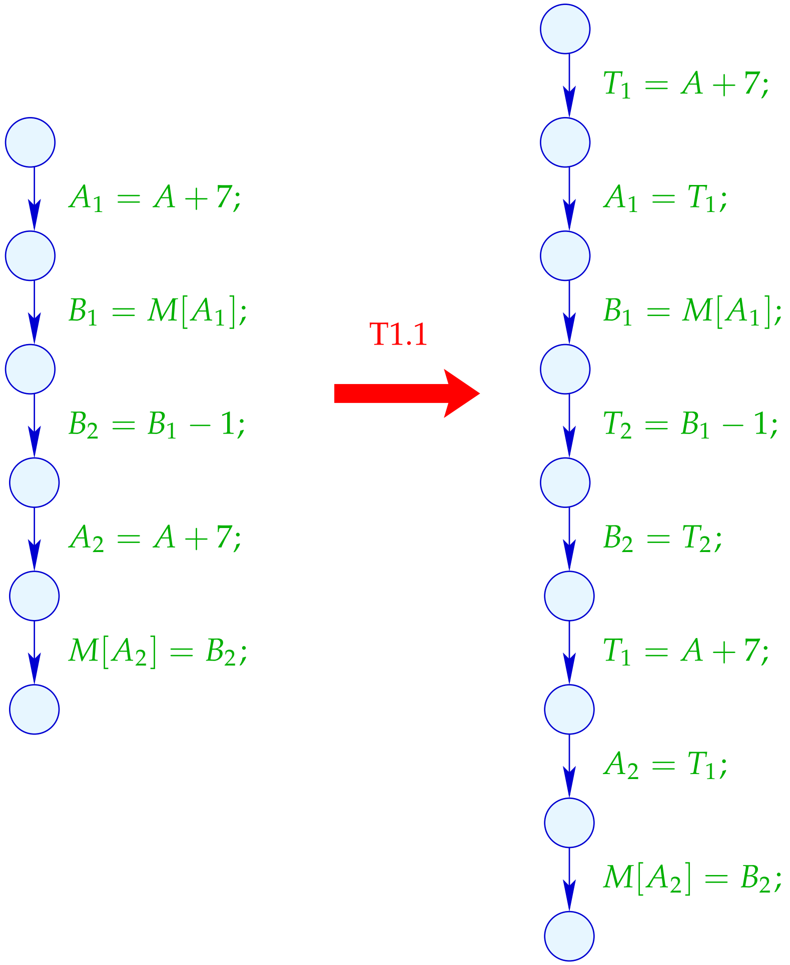
(2) Values of variables: T3

- + creates dead variables

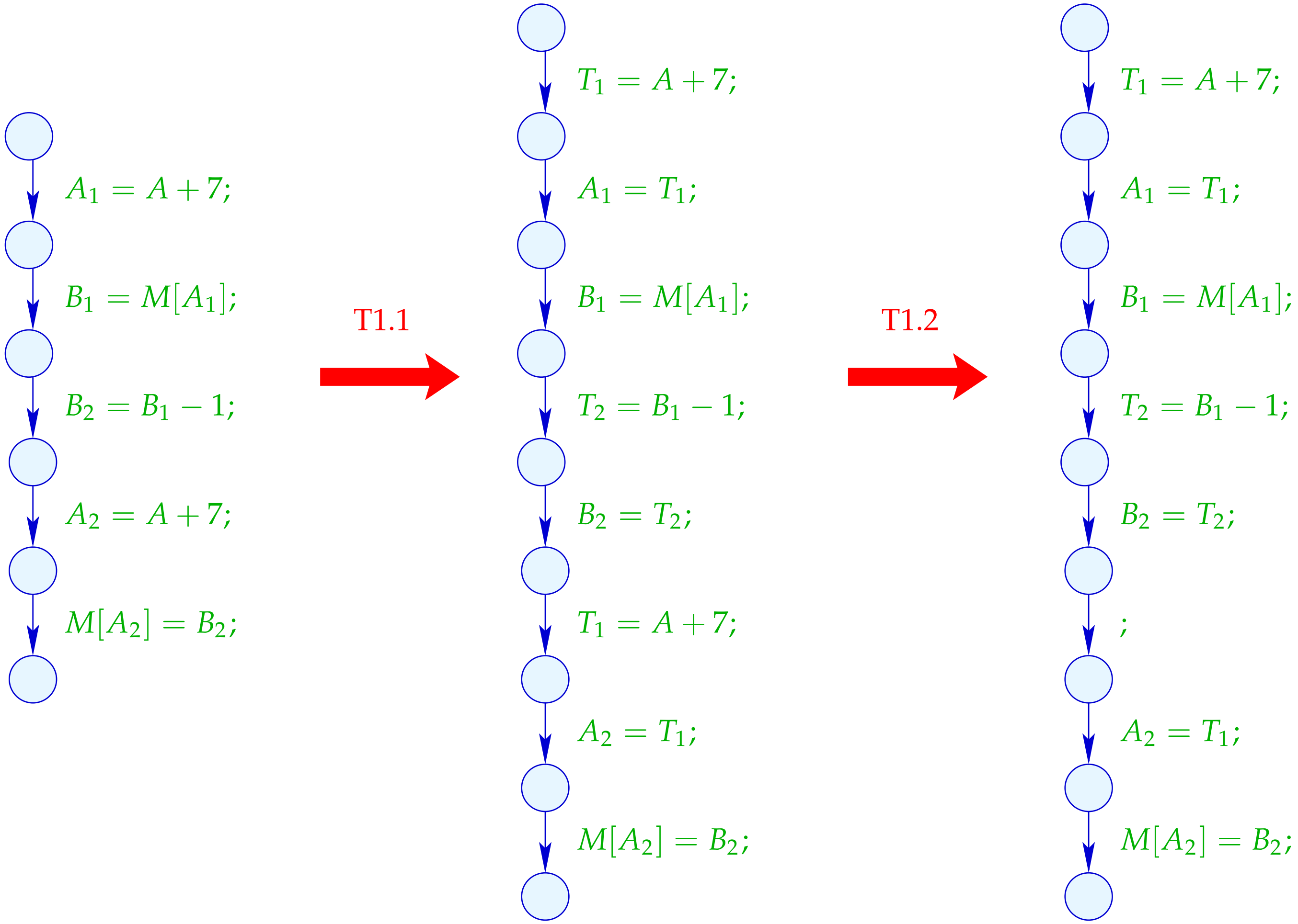
(3) (true) liveness of variables: T2

- + removes assignments to dead variables

Example: `a[7]--;`


$A_1 = A + 7;$
$B_1 = M[A_1];$
$B_2 = B_1 - 1;$
$A_2 = A + 7;$
$M[A_2] = B_2;$
T1.1
$T_1 = A + 7;$
$A_1 = T_1;$
$B_1 = M[A_1];$
$T_2 = B_1 - 1;$
$B_2 = T_2;$
$T_1 = A + 7;$
$A_2 = T_1;$
$M[A_2] = B_2;$

Example: `a[7]--;`



$A_1 = A + 7;$

$B_1 = M[A_1];$

$B_2 = B_1 - 1;$

$A_2 = A + 7;$

$M[A_2] = B_2;$

T1.1

$T_1 = A + 7;$

$A_1 = T_1;$

$B_1 = M[A_1];$

$T_2 = B_1 - 1;$

$B_2 = T_2;$

$T_1 = A + 7;$

$A_2 = T_1;$

$M[A_2] = B_2;$

T1.2

$T_1 = A + 7;$

$A_1 = T_1;$

$B_1 = M[A_1];$

$T_2 = B_1 - 1;$

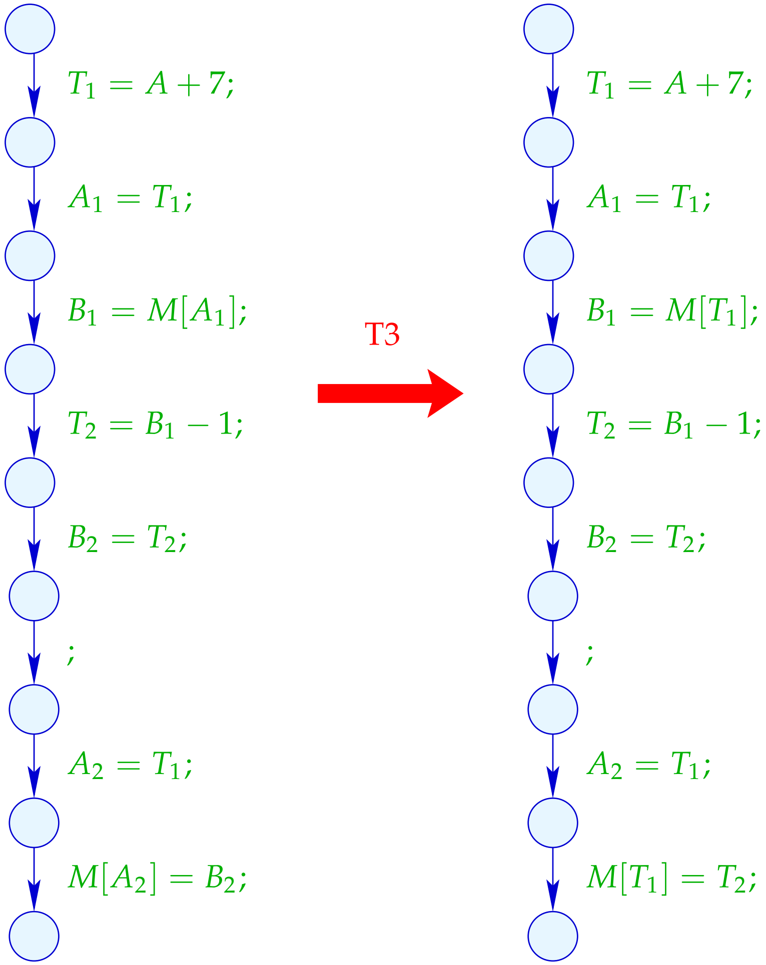
$B_2 = T_2;$

$;$

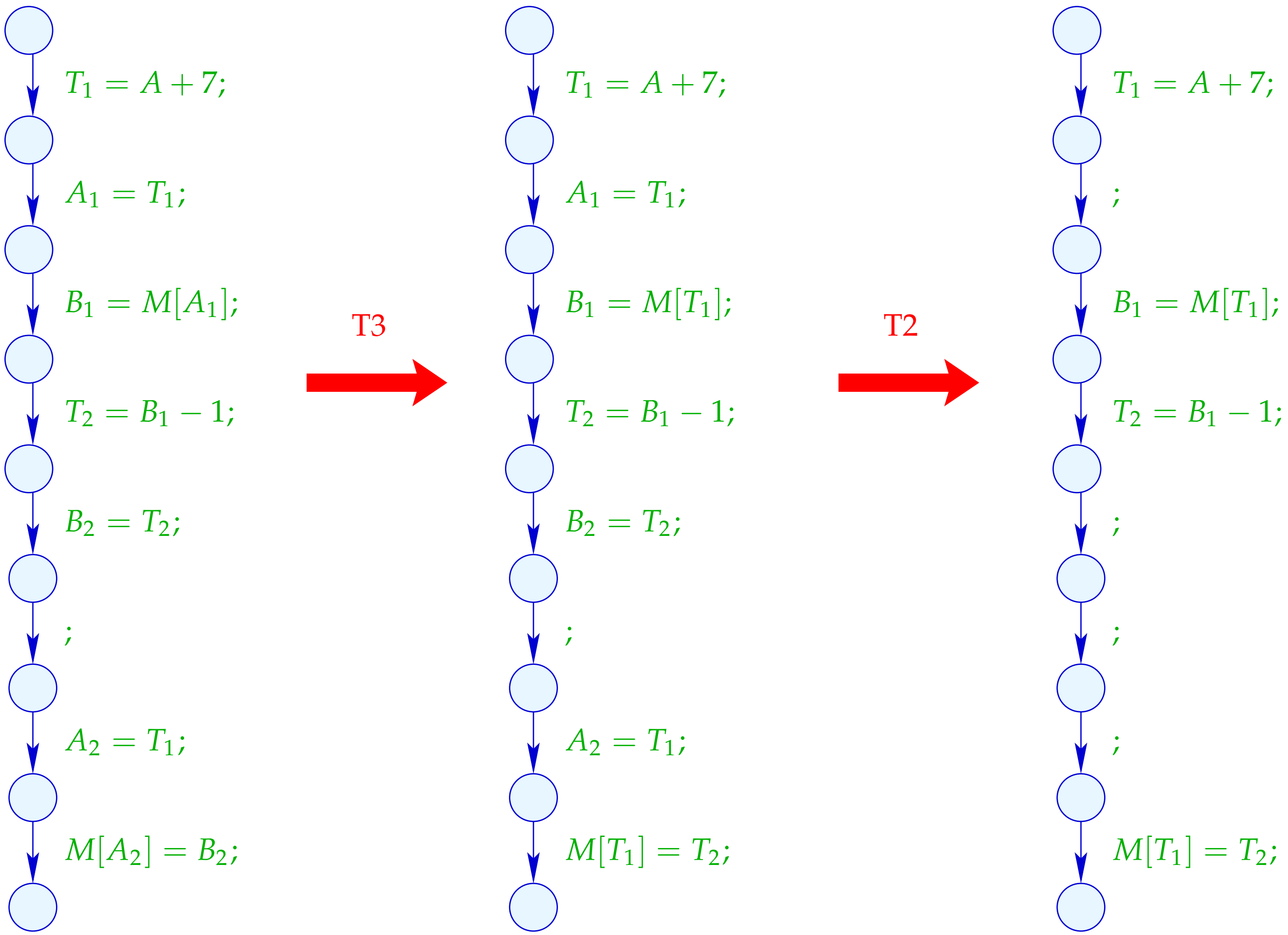
$A_2 = T_1;$

$M[A_2] = B_2;$

Example (cont.): `a[7]--;`



Example (cont.): `a[7]--;`



$T_1 = A + 7;$

$A_1 = T_1;$

$B_1 = M[A_1];$

$T_2 = B_1 - 1;$

$B_2 = T_2;$

$;$

$A_2 = T_1;$

$M[A_2] = B_2;$

T3

$T_1 = A + 7;$

$A_1 = T_1;$

$B_1 = M[T_1];$

$T_2 = B_1 - 1;$

$B_2 = T_2;$

$;$

$A_2 = T_1;$

$M[T_1] = T_2;$

T2

$T_1 = A + 7;$

$;$

$B_1 = M[T_1];$

$T_2 = B_1 - 1;$

$;$

$;$

$;$

$M[T_1] = T_2;$

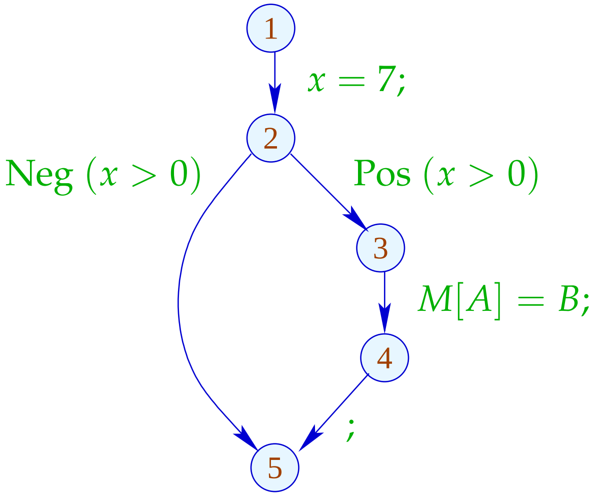
1.4 Constant Propagation

Idea:

Execute as much of the code at compile-time as possible!

Example:

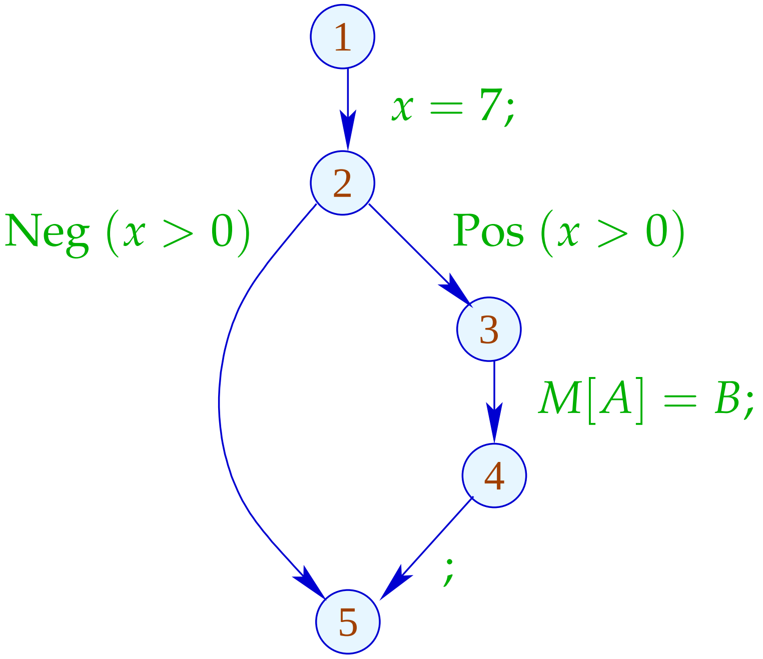
$$x = 7;$$
$$\text{if } (x > 0)$$
$$\qquad M[A] = B;$$



Obviously, x has always the value 7 :-)

Thus, the memory access is always executed :-))

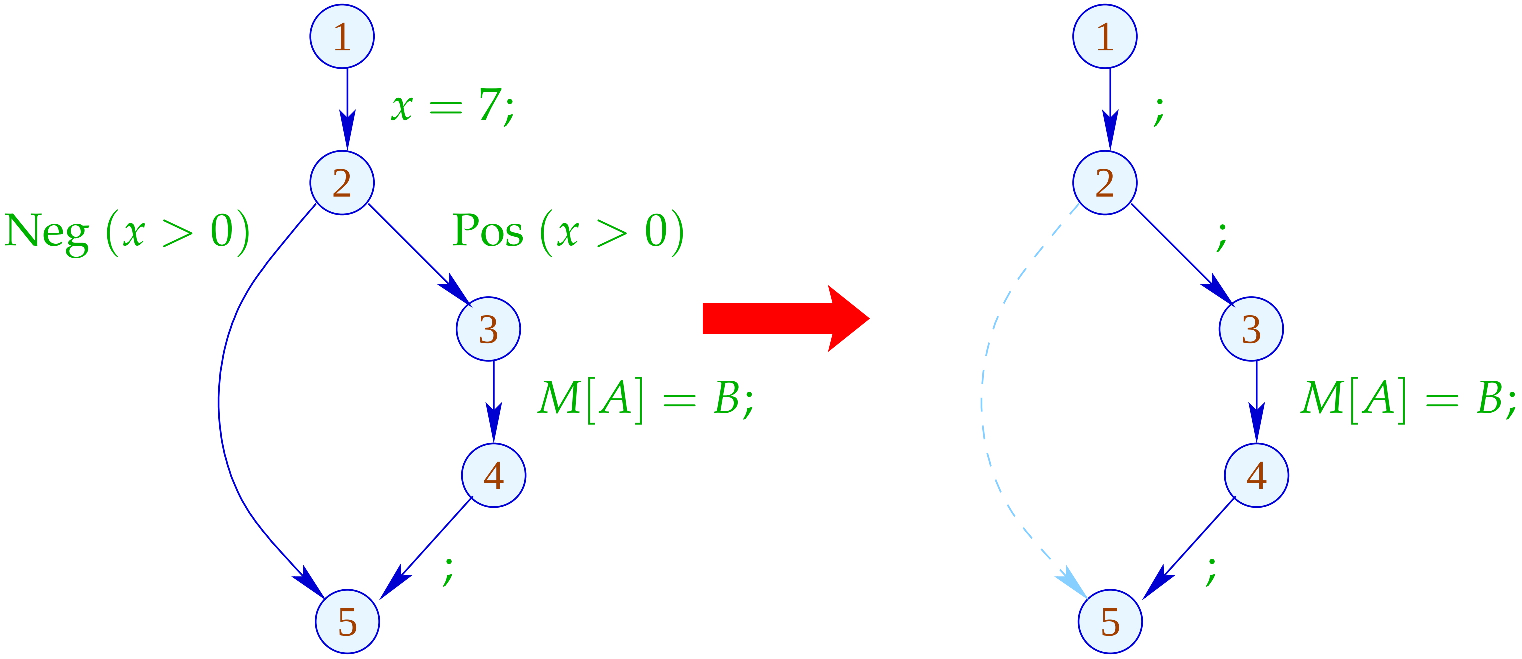
Goal:



Obviously, x has always the value 7 :-)

Thus, the memory access is always executed :-))

Goal:



Generalization: Partial Evaluation

Neil D. Jones, DIKU, Kopenhagen

Idea:

Design an analysis which for every u,

- determines the values which variables definitely have;
- tells whether u can be reached at all :-)

Idea:

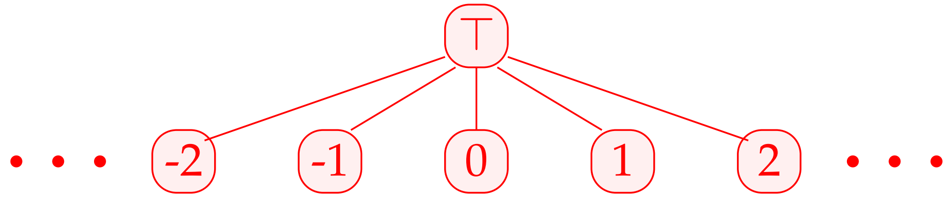
Design an analysis which for every u,

- determines the values which variables definitely have;
- tells whether u can be reached at all :-)

The complete lattice is constructed in two steps.

(1) The potential values of variables:

$$\mathbb{Z}^\top = \mathbb{Z} \cup \{\top\} \qquad \text{with} \quad x \sqsubseteq y \quad \text{iff } y = \top \text{ or } x = y$$



Warning: $\mathbb{Z}^\top$ is not a complete lattice in itself :-(

(2) $\mathbb{D} = (\mathit{Vars} \to \mathbb{Z}^\top)_\perp = (\mathit{Vars} \to \mathbb{Z}^\top) \cup \{\perp\}$

// $\perp$ denotes: "not reachable" :-))

with $D_1 \sqsubseteq D_2$ iff $\perp = D_1$ or

$D_1\, x \sqsubseteq D_2\, x \quad (x \in \mathit{Vars})$

Remark: $\mathbb{D}$ is a complete lattice :-)

Warning: $\mathbb{Z}^\top$ is not a complete lattice in itself :-(

(2) $\mathbb{D} = (\mathit{Vars} \to \mathbb{Z}^\top)_\perp = (\mathit{Vars} \to \mathbb{Z}^\top) \cup \{\perp\}$

// $\perp$ denotes: “not reachable” :-))

with $D_1 \sqsubseteq D_2$ iff $\perp = D_1$ or

$D_1\, x \sqsubseteq D_2\, x \quad (x \in \mathit{Vars})$

Remark: $\mathbb{D}$ is a complete lattice :-)

Consider $X \subseteq \mathbb{D}$. W.l.o.g., $\perp \not\in X$.

Then $X \subseteq \mathit{Vars} \to \mathbb{Z}^\top$.

If $X = \emptyset$, then $\bigsqcup X = \perp \in \mathbb{D}$:-)

If $X \neq \emptyset$, then $\bigsqcup X = D$ with

$$\begin{array}{lll} D\,x & = & \bigsqcup\{f\,x \mid f \in X\} \\ & = & \begin{cases} z & \text{if} \quad f\,x = z \quad (f \in X) \\ \top & \text{otherwise} \end{cases} \end{array}$$

:-))

If $X \neq \emptyset$, then $\bigsqcup X = D$ with

$$\begin{array}{lll} D\,x & = & \bigsqcup\{f\,x \mid f \in X\} \\ & = & \left\{\begin{array}{lll} z & \text{if} & f\,x = z \quad (f \in X) \\ \top & \text{otherwise} & \end{array}\right. \end{array}$$

:-))

For every edge $k = (\_, lab, \_)$, construct an effect function $[\![k]\!]^\sharp = [\![lab]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ which simulates the concrete computation.

Obviously, $[\![lab]\!]^\sharp \perp = \perp$ for all lab :-)

Now let $\perp \neq D \in \mathit{Vars} \to \mathbb{Z}^\top$.

Idea:

- We use D to determine the values of expressions.

Idea:

- We use D to determine the values of expressions.
- For some sub-expressions, we obtain $\top$:-)

Idea:

- We use D to determine the values of expressions.
- For some sub-expressions, we obtain $\top$:-)

$\Longrightarrow$

We must replace the concrete operators $\Box$ by abstract operators $\Box^\sharp$ which can handle $\top$:

$$a \,\Box^\sharp\, b = \begin{cases} \top & \text{if} \quad a = \top \text{ or } b = \top \\ a \,\Box\, b & \text{otherwise} \end{cases}$$

Idea:

- We use D to determine the values of expressions.
- For some sub-expressions, we obtain $\top$:-)

  $\Longrightarrow$

  We must replace the concrete operators $\Box$ by abstract operators $\Box^\sharp$ which can handle $\top$:

$$a \Box^\sharp b = \begin{cases} \top & \text{if} \quad a = \top \text{ or } b = \top \\ a \Box b & \text{otherwise} \end{cases}$$

- The abstract operators allow to define an abstract evaluation of expressions:

$$[\![e]\!]^\sharp \; : \; (\mathit{Vars} \to \mathbb{Z}^\top) \to \mathbb{Z}^\top$$

Abstract evaluation of expressions is like the concrete evaluation — but with abstract values and operators. Here:

$$\begin{array}{lll} [\![c]\!]^\sharp\, D & = & c \\ [\![e_1 \,\Box\, e_2]\!]^\sharp\, D & = & [\![e_1]\!]^\sharp\, D \,\Box^\sharp\, [\![e_2]\!]^\sharp\, D \end{array}$$

... analogously for unary operators :-)

Abstract evaluation of expressions is like the concrete evaluation — but with abstract values and operators. Here:

$$\begin{array}{lll} [\![c]\!]^\sharp\, D & = & c \\ [\![e_1 \,\Box\, e_2]\!]^\sharp\, D & = & [\![e_1]\!]^\sharp\, D \,\Box^\sharp\, [\![e_2]\!]^\sharp\, D \end{array}$$

... analogously for unary operators :-)

Example: $D = \{x \mapsto 2, y \mapsto \top\}$

$$\begin{array}{lll} [\![x+7]\!]^\sharp\, D & = & [\![x]\!]^\sharp\, D +^\sharp [\![7]\!]^\sharp\, D \\ & = & 2 +^\sharp 7 \\ & = & 9 \\ [\![x-y]\!]^\sharp\, D & = & 2 -^\sharp \top \\ & = & \top \end{array}$$

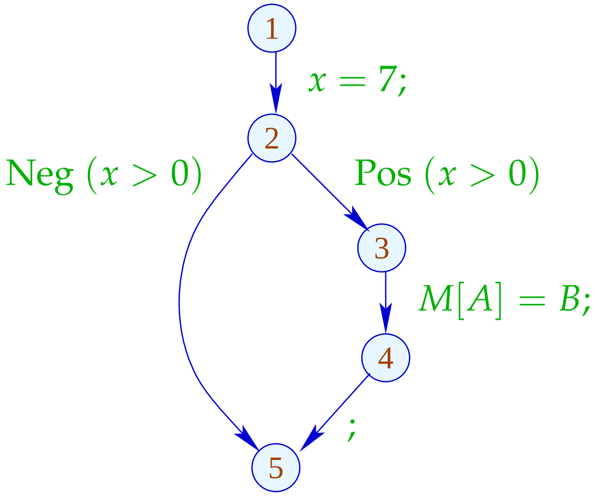
Thus, we obtain the following effects of edges $[\![lab]\!]^\sharp$:

$$\begin{array}{lll}
[\![;]\!]^\sharp\, D & = & D \\
[\![\text{Pos}\,(e)]\!]^\sharp\, D & = & \left\{\begin{array}{ll} \bot & \text{if} \quad 0 = [\![e]\!]^\sharp\, D \\ D & \text{otherwise} \end{array}\right. \\
[\![\text{Neg}\,(e)]\!]^\sharp\, D & = & \left\{\begin{array}{ll} D & \text{if} \quad 0 \sqsubseteq [\![e]\!]^\sharp\, D \\ \bot & \text{otherwise} \end{array}\right. \\
[\![x = e;]\!]^\sharp\, D & = & D \oplus \{x \mapsto [\![e]\!]^\sharp\, D\} \\
[\![x = M[e];]\!]^\sharp\, D & = & D \oplus \{x \mapsto \top\} \\
[\![M[e_1] = e_2;]\!]^\sharp\, D & = & D
\end{array}$$

... whenever $D \neq \bot$:-)

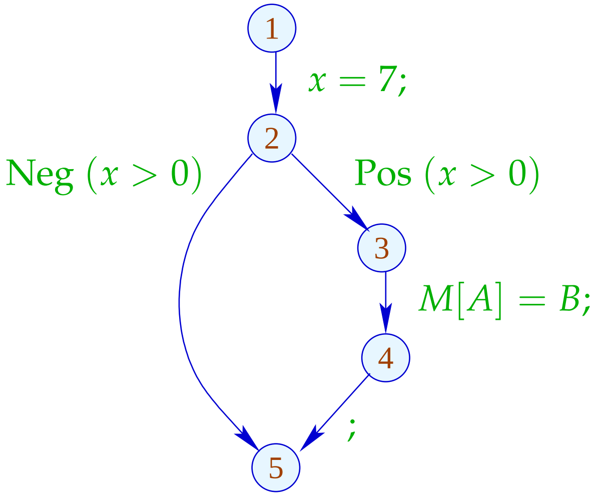
At *start*, we have $D_\top = \{x \mapsto \top \mid x \in \mathit{Vars}\}$.

Example:



At *start*, we have $D_\top = \{x \mapsto \top \mid x \in \mathit{Vars}\}$.

Example:



1	$\{x \mapsto \top\}$
2	$\{x \mapsto 7\}$
3	$\{x \mapsto 7\}$
4	$\{x \mapsto 7\}$
5	$\bot \sqcup \{x \mapsto 7\} = \{x \mapsto 7\}$

The abstract effects of edges $[\![k]\!]^\sharp$ are again composed to the effects of paths $\pi = k_1 \ldots k_r$ by:

$$[\![\pi]\!]^\sharp = [\![k_r]\!]^\sharp \circ \ldots \circ [\![k_1]\!]^\sharp \quad : \mathbb{D} \to \mathbb{D}$$

Idea for Correctness: Abstract Interpretation

Cousot, Cousot 1977

Patrick Cousot, ENS, Paris

The abstract effects of edges $[\![k]\!]^\sharp$ are again composed to the effects of paths $\pi = k_1 \ldots k_r$ by:

$$[\![\pi]\!]^\sharp = [\![k_r]\!]^\sharp \circ \ldots \circ [\![k_1]\!]^\sharp \quad : \mathbb{D} \to \mathbb{D}$$

Idea for Correctness: Abstract Interpretation

Cousot, Cousot 1977

Establish a description relation Δ between theconcrete values and their descriptions with:

$$x \,\Delta\, a_1 \quad \wedge \quad a_1 \sqsubseteq a_2 \quad \Longrightarrow \quad x \,\Delta\, a_2$$

Concretization: $\gamma\, a = \{x \mid x \,\Delta\, a\}$

// returns the set of described values :-)

(1) Values: $\Delta \subseteq \mathbb{Z} \times \mathbb{Z}^\top$

$$z \,\Delta\, a \quad \text{iff} \quad z = a \vee a = \top$$

Concretization:

$$\gamma\, a = \begin{cases} \{a\} & \text{if} \quad a \sqsubset \top \\ \mathbb{Z} & \text{if} \quad a = \top \end{cases}$$

(1) Values: $\Delta \subseteq \mathbb{Z} \times \mathbb{Z}^\top$

$$z \, \Delta \, a \quad \text{iff} \quad z = a \vee a = \top$$

Concretization:

$$\gamma \, a = \begin{cases} \{a\} & \text{if} \quad a \sqsubset \top \\ \mathbb{Z} & \text{if} \quad a = \top \end{cases}$$

(2) Variable Assignments: $\Delta \subseteq (\mathit{Vars} \to \mathbb{Z}) \times (\mathit{Vars} \to \mathbb{Z}^\top)_\perp$

$$\rho \, \Delta \, D \quad \text{iff} \quad D \neq \perp \wedge \rho \, x \sqsubseteq D \, x \quad (x \in \mathit{Vars})$$

Concretization:

$$\gamma \, D = \begin{cases} \emptyset & \text{if} \quad D = \perp \\ \{\rho \mid \forall \, x : \; (\rho \, x) \, \Delta \, (D \, x)\} & \text{otherwise} \end{cases}$$

Example: $\{x \mapsto 1, y \mapsto -7\} \;\Delta\; \{x \mapsto \top, y \mapsto -7\}$

(3) States:

$$\Delta \;\subseteq\; ((\mathit{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z})) \times (\mathit{Vars} \to \mathbb{Z}^{\top})_{\perp}$$

$$(\rho, \mu) \;\Delta\; D \qquad \text{iff} \qquad \rho \;\Delta\; D$$

Concretization:

$$\gamma\, D = \begin{cases} \emptyset & \text{if} \quad D = \perp \\ \{(\rho, \mu) \mid \forall\, x :\; (\rho\, x) \;\Delta\; (D\, x)\} & \text{otherwise} \end{cases}$$

We show:

(∗) If $s \,\Delta\, D$ and $[\![\pi]\!]\, s$ is defined, then:

$$([\![\pi]\!]\, s) \;\Delta\; ([\![\pi]\!]^\sharp\, D)$$

$$\begin{tikzcd}
\boxed{s} \arrow[r, "{[\![\pi]\!]}"] \arrow[d, dash, "\Delta"'] & \boxed{s_1} \arrow[d, dash, dotted, "\Delta"] \\
\boxed{D} \arrow[r, "{[\![\pi]\!]^\sharp}"'] & \boxed{D_1}
\end{tikzcd}$$

The abstract semantics simulates the concrete semantics :-)

In particular:

$$[\![\pi]\!]\, s \in \gamma\,([\![\pi]\!]^{\sharp}\, D)$$

The abstract semantics simulates the concrete semantics :-)

In particular:

$$[\![\pi]\!]\, s \in \gamma\,([\![\pi]\!]^\sharp\, D)$$

In practice, this means, e.g., that $D\,x = -7$ implies:

$$\begin{array}{rrcll} & \rho'\,x & = & -7 & \text{for all} \quad \rho' \in \gamma\, D \\ \Longrightarrow & \rho_1\,x & = & -7 & \text{for} \quad (\rho_1, \_) = [\![\pi]\!]\, s \end{array}$$

To prove $(*)$, we show for every edge k :

$(**)$

$$\begin{tikzcd} \boxed{s} \arrow[r, "[\![k]\!]"] \arrow[d, dash, "\Delta"'] & \boxed{s_1} \arrow[d, dash, dotted, "\Delta"] \\ \boxed{D} \arrow[r, "[\![k]\!]^\sharp"'] & \boxed{D_1} \end{tikzcd}$$

Then $(*)$ follows by induction :-)

To prove $(**)$, we show for every expression e :

$$(***) \quad ([\![e]\!]\,\rho) \;\; \Delta \;\; ([\![e]\!]^\sharp\, D) \quad \text{whenever} \quad \rho \,\Delta\, D$$

To prove $(**)$, we show for every expression e :

$(***) \quad ([\![e]\!]\,\rho) \;\; \Delta \;\; ([\![e]\!]^\sharp\, D)$ whenever $\rho \,\Delta\, D$

To prove $(***)$, we show for every operator $\Box$:

$$(x \,\Box\, y) \;\; \Delta \;\; (x^\sharp \,\Box^\sharp\, y^\sharp) \qquad \text{whenever} \quad x \,\Delta\, x^\sharp \wedge y \,\Delta\, y^\sharp$$

To prove $(**)$, we show for every expression e :

$(***)$ $\quad ([\![e]\!]\,\rho) \;\Delta\; ([\![e]\!]^\sharp\, D)$ $\quad$ whenever $\quad \rho \,\Delta\, D$

To prove $(***)$, we show for every operator $\square$:

$$(x \,\square\, y) \;\Delta\; (x^\sharp \,\square^\sharp\, y^\sharp) \qquad \text{whenever} \quad x \,\Delta\, x^\sharp \wedge y \,\Delta\, y^\sharp$$

This precisely was how we have defined the operators $\square^\sharp$:-)

Now, $(**)$ is proved by case distinction on the edge labels lab .

Let $s = (\rho, \mu) \;\Delta\; D$. In particular, $\bot \neq D \;:\; Vars \to \mathbb{Z}^\top$

Case $\boxed{x = e;}$:

$$\begin{array}{lll} \rho_1 & = & \rho \oplus \{x \mapsto [\![e]\!]\,\rho\} \qquad \mu_1 \;=\; \mu \\ D_1 & = & D \oplus \{x \mapsto [\![e]\!]^\sharp\, D\} \end{array}$$

$$\Longrightarrow \qquad (\rho_1, \mu_1) \;\Delta\; D_1$$

Case $\boxed{x = M[e];}$:

$$\rho_1 = \rho \oplus \{x \mapsto \mu\,(\llbracket e \rrbracket^\sharp \rho)\} \qquad \mu_1 = \mu$$

$$D_1 = D \oplus \{x \mapsto \top\}$$

$$\Longrightarrow \qquad (\rho_1, \mu_1)\ \Delta\ D_1$$

Case $\boxed{M[e_1] = e_2;}$:

$$\rho_1 = \rho \qquad \mu_1 = \mu \oplus \{\llbracket e_1 \rrbracket^\sharp \rho \mapsto \llbracket e_2 \rrbracket^\sharp \rho\}$$

$$D_1 = D$$

$$\Longrightarrow \qquad (\rho_1, \mu_1)\ \Delta\ D_1$$

Case $\boxed{\mathrm{Neg}(e)}$:

$$(\rho_1, \mu_1) = s \quad \text{where:}$$

$$\begin{array}{rlll} & 0 & = & [\![e]\!]\,\rho \\ & & \Delta & [\![e]\!]^\sharp\, D \\ \Longrightarrow & 0 & \sqsubseteq & [\![e]\!]^\sharp\, D \\ \Longrightarrow & \bot & \neq & D_1 \;=\; D \\ \Longrightarrow & \multicolumn{3}{l}{(\rho_1, \mu_1)\;\Delta\; D_1} \end{array}$$

Case $\boxed{\mathrm{Pos}(e)}$: $(\rho_1, \mu_1) = s$ where:

$$\begin{array}{lccc} & 0 & \neq & [\![e]\!]\,\rho \\ & & \Delta & [\![e]\!]^\sharp\, D \\ \Longrightarrow & 0 & \neq & [\![e]\!]^\sharp\, D \\ \Longrightarrow & \bot & \neq & D_1 \;=\; D \\ \Longrightarrow & \multicolumn{3}{l}{(\rho_1, \mu_1)\;\Delta\; D_1} \end{array}$$

:-)

We conclude: The assertion $(*)$ is true :-))

The MOP-Solution:

$$\mathcal{D}^*[v] = \bigsqcup\{[\![\pi]\!]^\sharp\, D_\top \mid \pi : \mathit{start} \to^* v\}$$

where $D_\top\, x = \top \qquad (x \in \mathit{Vars})$.

We conclude: The assertion $(*)$ is true :-))

The MOP-Solution:

$$\mathcal{D}^*[v] \;=\; \bigsqcup\{[\![\pi]\!]^\sharp\, D_\top \mid \pi : \mathit{start} \to^* v\}$$

where $D_\top\, x = \top \qquad (x \in \mathit{Vars})$.

By $(*)$, we have for all initial states s and all program executions π which reach v :

$$([\![\pi]\!]\, s) \;\;\Delta\;\; (\mathcal{D}^*[v])$$

We conclude: The assertion $(*)$ is true :-))

The MOP-Solution

$$\mathcal{D}^*[v] \;=\; \bigsqcup\{[\![\pi]\!]^\sharp\, D_\top \mid \pi : start \to^* v\}$$

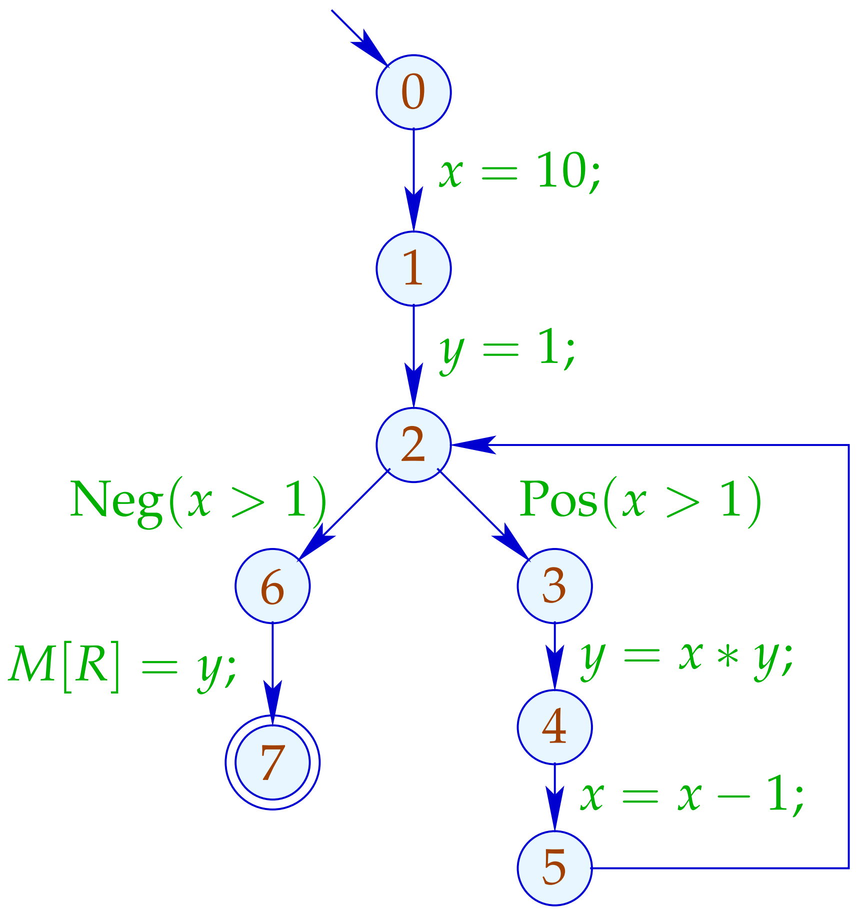
where $D_\top\, x = \top \qquad (x \in Vars)$.

By $(*)$, we have for all initial states s and all program executions π which reach v :

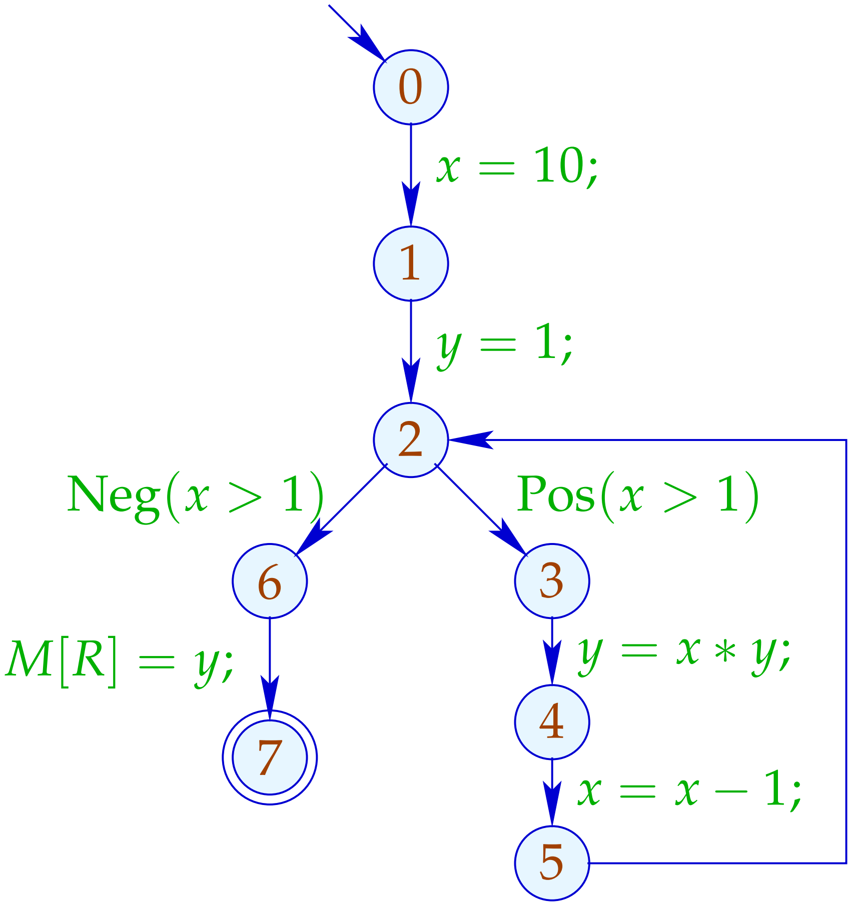
$$([\![\pi]\!]\, s) \;\;\Delta\;\; (\mathcal{D}^*[v])$$

In order to approximate the MOP, we use our constraint system :-))

Example:

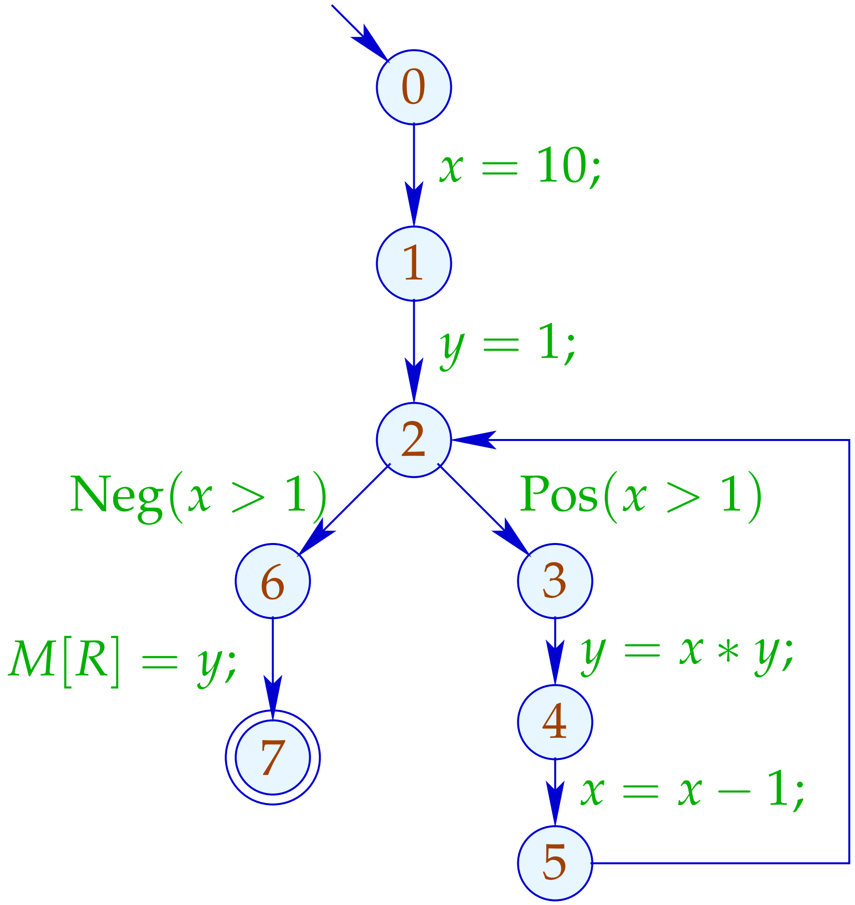


Example:



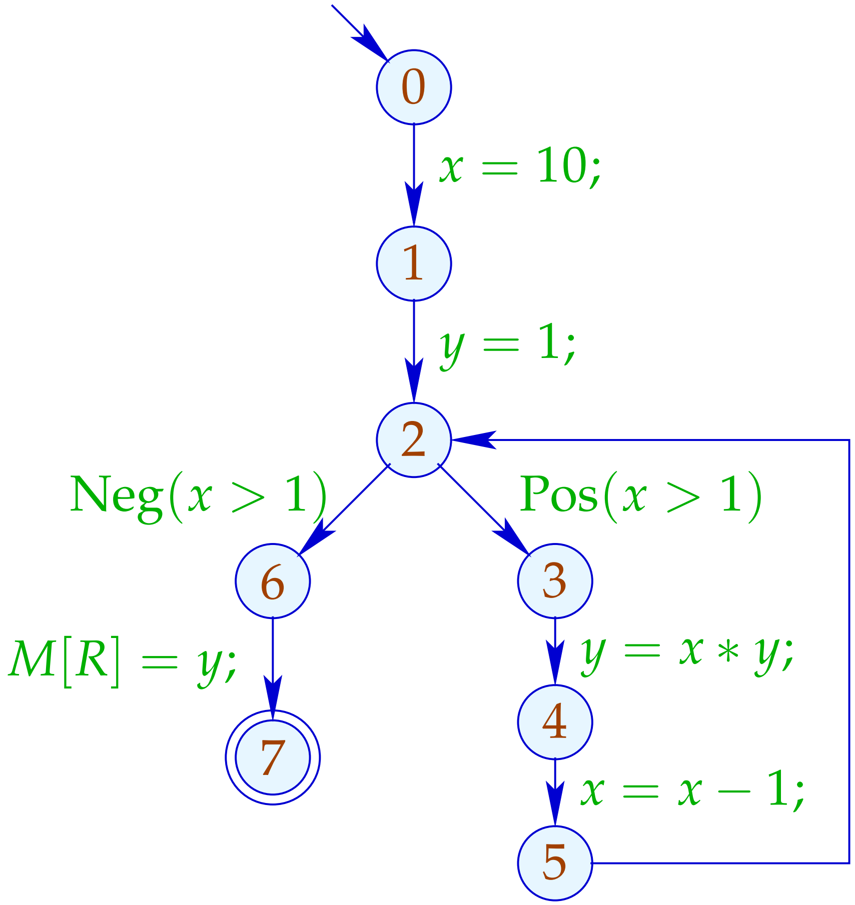
	1	
	x	y
0	$\top$	$\top$
1	10	$\top$
2	10	1
3	10	1
4	10	10
5	9	10
6	$\bot$	
7	$\bot$	

Example:



	1		2	
	x	y	x	y
0	$\top$	$\top$	$\top$	$\top$
1	10	$\top$	10	$\top$
2	10	1	$\top$	$\top$
3	10	1	$\top$	$\top$
4	10	10	$\top$	$\top$
5	9	10	$\top$	$\top$
6	$\bot$		$\top$	$\top$
7	$\bot$		$\top$	$\top$

Example:



	1		2		3	
	x	y	x	y	x	y
0	$\top$	$\top$	$\top$	$\top$	dito	
1	10	$\top$	10	$\top$		
2	10	1	$\top$	$\top$		
3	10	1	$\top$	$\top$		
4	10	10	$\top$	$\top$		
5	9	10	$\top$	$\top$		
6	$\bot$		$\top$	$\top$		
7	$\bot$		$\top$	$\top$		

Conclusion:

Although we compute with concrete values, we fail to compute everything :-(

The fixpoint iteration, at least, is guaranteed to terminate:

For n program points and m variables, we maximally need: $n \cdot (m + 1)$ rounds :-)

Warning:

The effects of edge are not distributive !!!

Counter Example: $\quad f = [\![x = x + y;]\!]^\sharp$

Let
$$\begin{aligned} D_1 &= \{x \mapsto 2, y \mapsto 3\} \\ D_2 &= \{x \mapsto 3, y \mapsto 2\} \end{aligned}$$

Dann
$$\begin{aligned} f\,D_1 \sqcup f\,D_2 &= \{x \mapsto 5, y \mapsto 3\} \sqcup \{x \mapsto 5, y \mapsto 2\} \\ &= \{x \mapsto 5, y \mapsto \top\} \\ &\neq \{x \mapsto \top, y \mapsto \top\} \\ &= f\,\{x \mapsto \top, y \mapsto \top\} \\ &= f\,(D_1 \sqcup D_2) \end{aligned}$$

:-((

We conclude:

The least solution $\mathcal{D}$ of the constraint system in general yields only an upper approximation of the MOP, i.e.,

$$\mathcal{D}^*[v] \quad \sqsubseteq \quad \mathcal{D}[v]$$

We conclude:

The least solution $\mathcal{D}$ of the constraint system in general yields only an upper approximation of the MOP, i.e.,

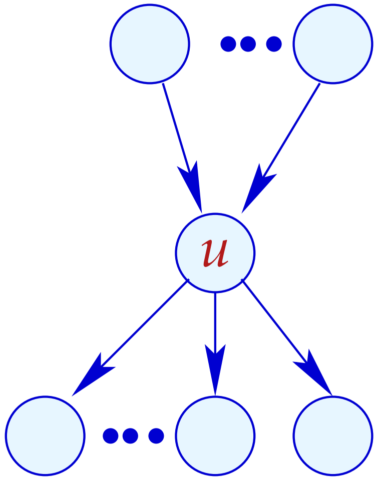
$$\mathcal{D}^*[v] \quad \sqsubseteq \quad \mathcal{D}[v]$$

As an upper approximation, $\mathcal{D}[v]$ nonetheless describes the result of every program execution π which reaches v :

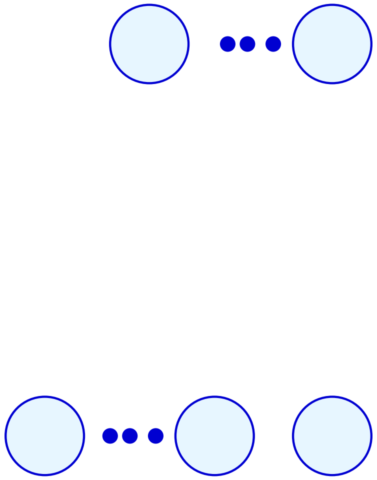
$$(\llbracket\pi\rrbracket\,(\rho, \mu)) \;\; \Delta \;\; (\mathcal{D}[v])$$

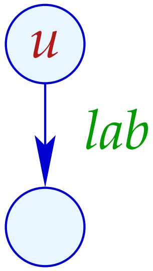
whenever $\llbracket\pi\rrbracket\,(\rho, \mu)$ is defined ;-))

Transformation 4: Removal of Dead Code

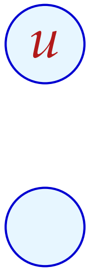


$\mathcal{D}[u] = \bot$

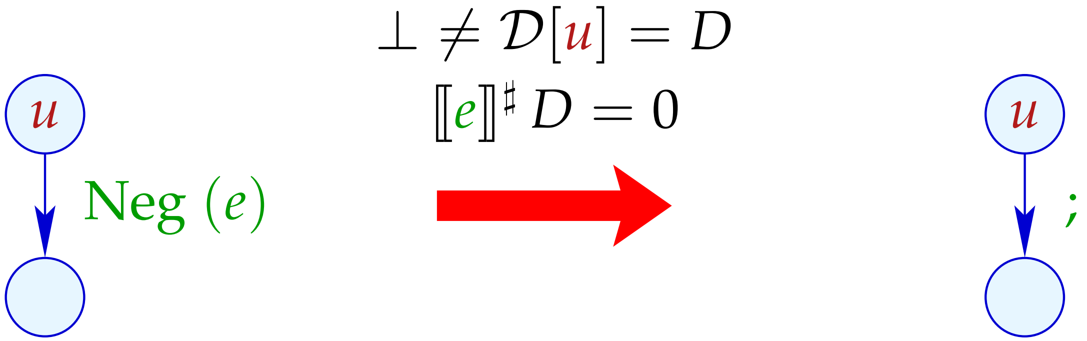




$[\![lab]\!]^\sharp(\mathcal{D}[u]) = \bot$

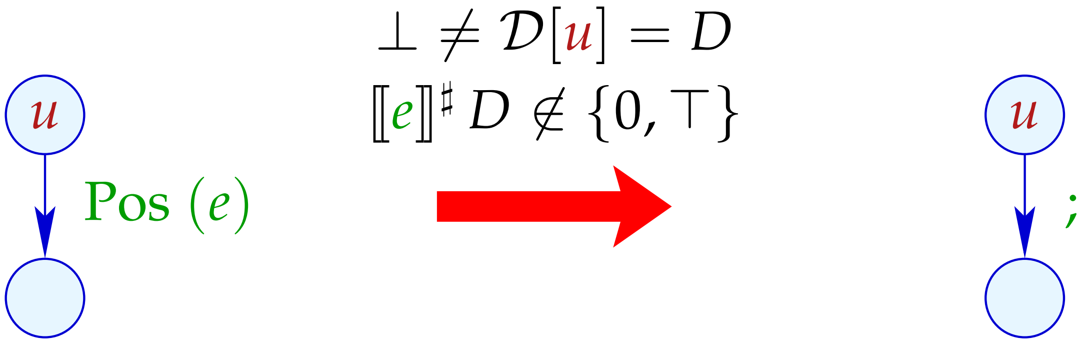


Transformation 4 (cont.): Removal of Dead Code



$$\perp \neq \mathcal{D}[u] = D$$

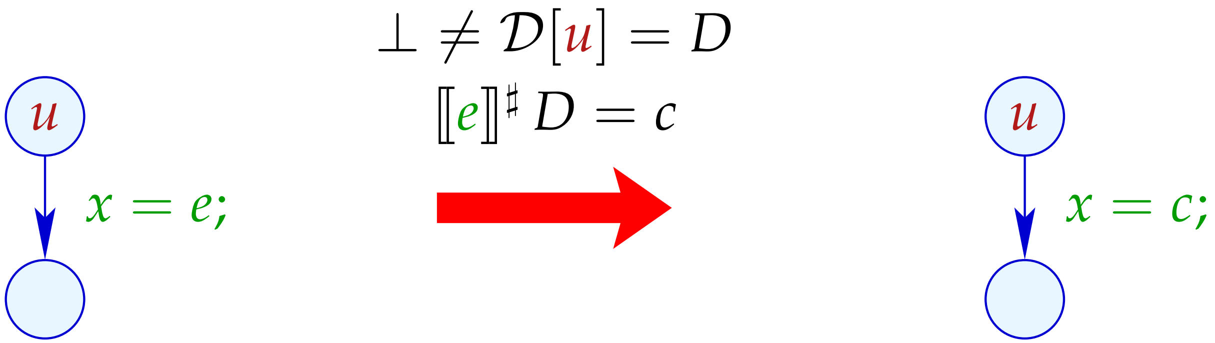
$$[\![e]\!]^{\sharp}\, D = 0$$



$$\perp \neq \mathcal{D}[u] = D$$

$$[\![e]\!]^{\sharp}\, D \notin \{0, \top\}$$

Transformation 4 (cont.): Simplified Expressions



$$\perp \neq \mathcal{D}[u] = D$$

$$[\![e]\!]^\sharp\, D = c$$

Extensions:

- Instead of complete right-hand sides, also subexpressions could be simplified:

$$x + (3 * y) \quad \xRightarrow{\{x \mapsto \top, y \mapsto 5\}} \quad x + 15$$

... and further simplifications be applied, e.g.:

$$\begin{array}{rcl} x * 0 & \Longrightarrow & 0 \\ x * 1 & \Longrightarrow & x \\ x + 0 & \Longrightarrow & x \\ x - 0 & \Longrightarrow & x \\ & \ldots & \end{array}$$

- So far, the information of conditions has not yet be optimally exploited:

$$\textsf{if } (x == 7)$$
$$y = x + 3;$$

Even if the value of x before the if statement is unknown, we at least know that x definitely has the value 7 — whenever the then-part is entered :-)

Therefore, we can define:

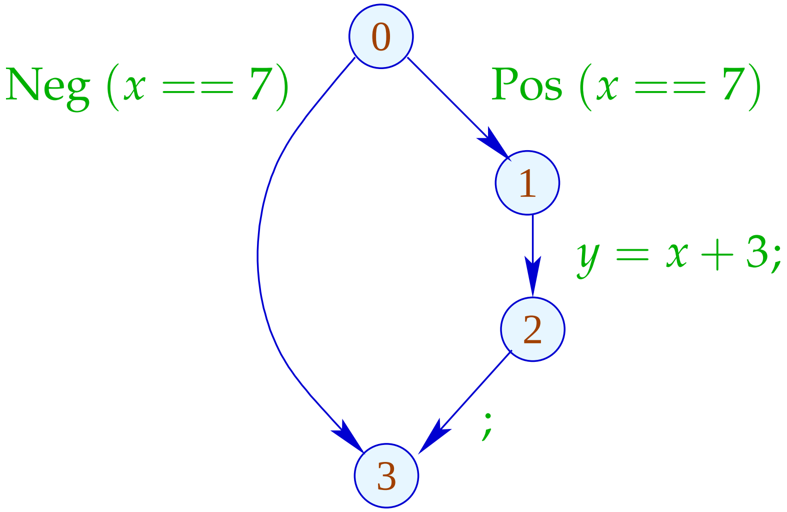
$$[\![\text{Pos}\,(x == e)]\!]^\sharp\, D = \begin{cases} D & \text{if} \quad [\![x == e]\!]^\sharp\, D = 1 \\ \bot & \text{if} \quad [\![x == e]\!]^\sharp\, D = 0 \\ D_1 & \text{otherwise} \end{cases}$$

where

$$D_1 = D \oplus \{x \mapsto (D\,x \sqcap [\![e]\!]^\sharp\, D)\}$$

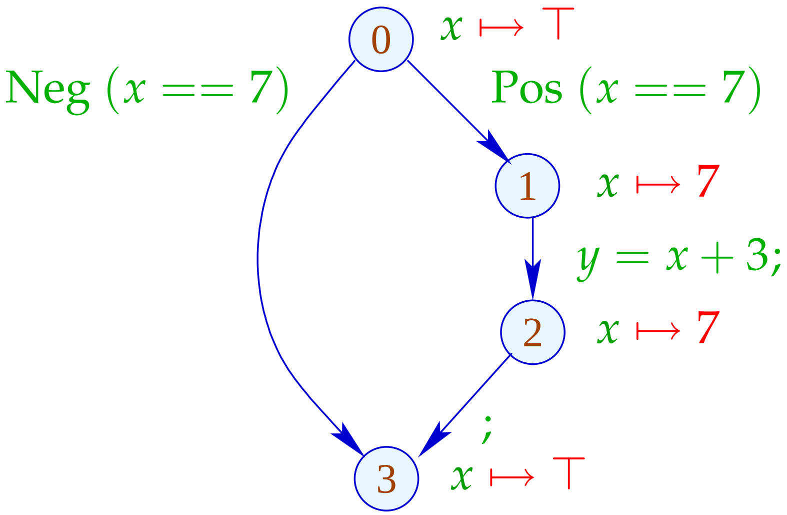
The effect of an edge labeled $\text{Neg}\,(x \neq e)$ is analogous :-)

Our Example:



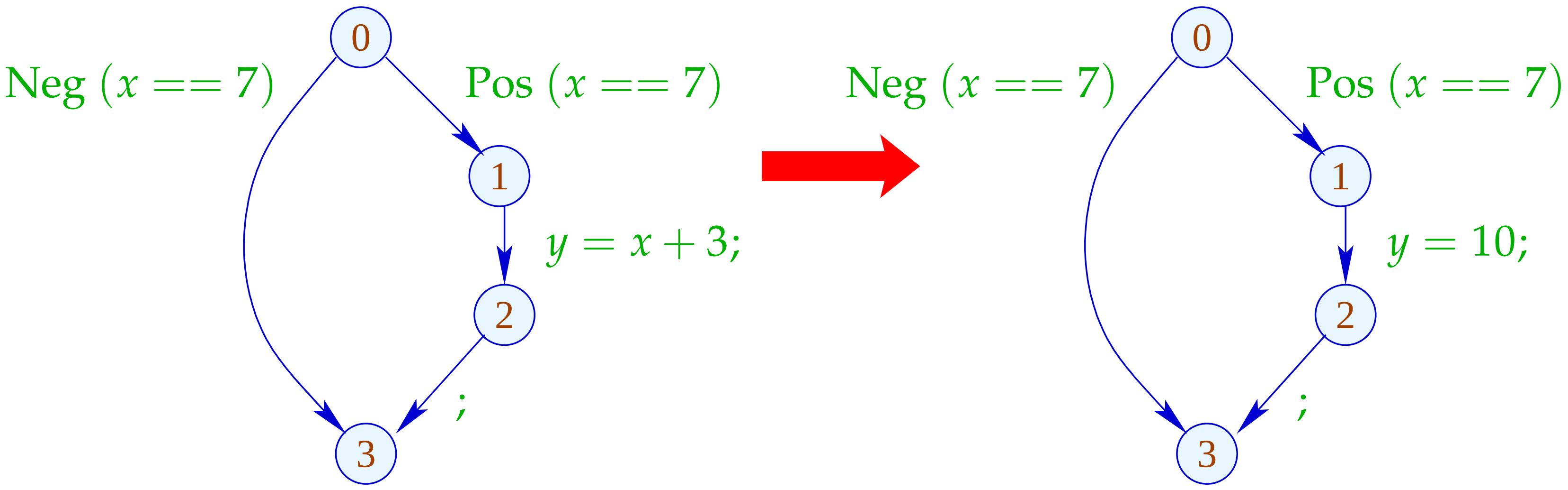
The effect of an edge labeled $\text{Neg}\,(x \neq e)$ is analogous :-)

Our Example:



The effect of an edge labeled $\text{Neg}\,(x \neq e)$ is analogous :-)

Our Example:



1.5 Interval Analysis

Observation:

- Programmers often use global constants for switching debugging code on/off.

  $\Longrightarrow$

  Constant propagation is useful :-)

- In general, precise values of variables will be unknown — perhaps, however, a tight interval !!!

Example:

```
for ($i = 0; i < 42; i{+}{+}$)
    if ($0 \leq i \wedge i < 42$){
        $A_1 = A + i;$
        $M[A_1] = i;$
    }
  //   $A$ start address of an array
  //   if the array-bound check
```

Obviously, the inner check is superfluous :-)

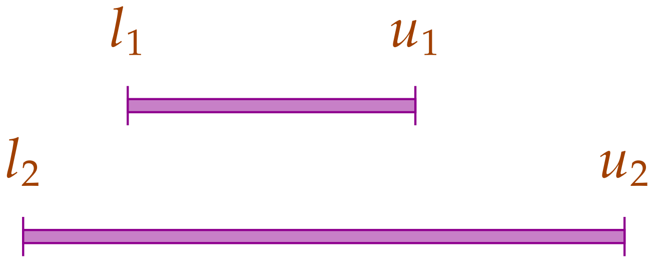
Idea 1:

Determine for every variable x an (as tight as possible :-) interval of possible values:

$$\mathbb{I} = \{[l, u] \mid l \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{+\infty\}, l \leq u\}$$

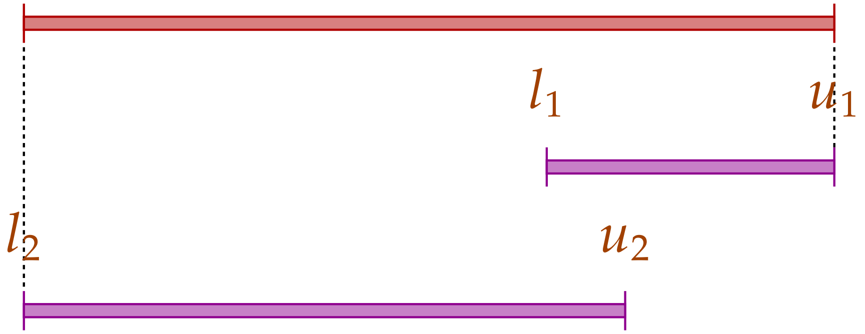
Partial Ordering:

$$[l_1, u_1] \sqsubseteq [l_2, u_2] \qquad \text{iff} \qquad l_2 \leq l_1 \wedge u_1 \leq u_2$$



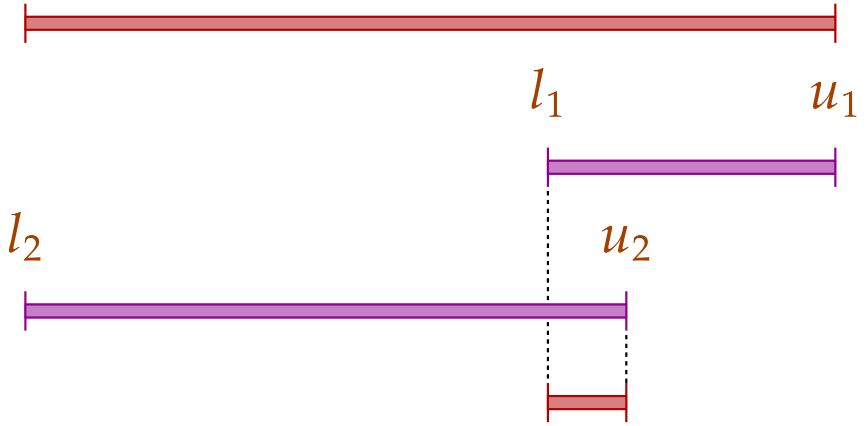
Thus:

$$[l_1, u_1] \sqcup [l_2, u_2] \quad = \quad [l_1 \sqcap l_2, u_1 \sqcup u_2]$$



Thus:

$$
\begin{array}{lcll}
[l_1, u_1] \sqcup [l_2, u_2] & = & [l_1 \sqcap l_2, u_1 \sqcup u_2] & \\
[l_1, u_1] \sqcap [l_2, u_2] & = & [l_1 \sqcup l_2, u_1 \sqcap u_2] & \text{whenever } (l_1 \sqcup l_2) \leq (u_1 \sqcap u_2)
\end{array}
$$



Warning:

→ $\mathbb{I}$ is not a complete lattice :-)

→ $\mathbb{I}$ has infinite ascending chains, e.g.,

$$[0,0] \sqsubset [0,1] \sqsubset [-1,1] \sqsubset [-1,2] \sqsubset \ldots$$

Warning:

→ $\mathbb{I}$ is not a complete lattice :-)

→ $\mathbb{I}$ has infinite ascending chains, e.g.,

$$[0,0] \sqsubset [0,1] \sqsubset [-1,1] \sqsubset [-1,2] \sqsubset \ldots$$

Description Relation:

$$z \;\Delta\; [l,u] \qquad \text{iff} \qquad l \leq z \leq u$$

Concretization:

$$\gamma\,[l,u] = \{z \in \mathbb{Z} \mid l \leq z \leq u\}$$

Example:

$$\begin{array}{rcl} \gamma\,[0,7] & = & \{0,\ldots,7\} \\ \gamma\,[0,\infty] & = & \{0,1,2,\ldots,\} \end{array}$$

Computing with intervals:

Interval Arithmetic :-)

Addition:

$$\begin{array}{rcl} [l_1, u_1] +^\sharp [l_2, u_2] & = & [l_1 + l_2, u_1 + u_2] \qquad \text{where} \\ -\infty + \_ & = & -\infty \\ +\infty + \_ & = & +\infty \\ & // & -\infty + \infty \quad \text{cannot occur} \end{array}$$

:-)

Negation:

$$-^{\sharp}\,[l,u] \quad = \quad [-u,-l]$$

Multiplication:

$$\begin{array}{rcl} [l_1,u_1] \;*^{\sharp}\; [l_2,u_2] & = & [a,b] \qquad \text{where} \\ a & = & l_1 l_2 \sqcap l_1 u_2 \sqcap u_1 l_2 \sqcap u_1 u_2 \\ b & = & l_1 l_2 \sqcup l_1 u_2 \sqcup u_1 l_2 \sqcup u_1 u_2 \end{array}$$

Example:

$$\begin{array}{rcl} [0,2] \;*^{\sharp}\; [3,4] & = & [0,8] \\ [-1,2] \;*^{\sharp}\; [3,4] & = & [-4,8] \\ [-1,2] \;*^{\sharp}\; [-3,4] & = & [-6,8] \\ [-1,2] \;*^{\sharp}\; [-4,-3] & = & [-8,4] \end{array}$$

Division: $[l_1, u_1] \,/^\sharp\, [l_2, u_2] \;=\; [a, b]$

- If 0 is not contained in the interval of the denominator, then:

$$
\begin{array}{lll}
a & = & l_1/l_2 \sqcap l_1/u_2 \sqcap u_1/l_2 \sqcap u_1/u_2 \\
b & = & l_1/l_2 \sqcup l_1/u_2 \sqcup u_1/l_2 \sqcup u_1/u_2
\end{array}
$$

- If: $l_2 \leq 0 \leq u_2$, we define:

$$[a, b] \;=\; [-\infty, +\infty]$$

Equality:

$$[l_1, u_1] ==^\sharp [l_2, u_2] \;=\; \begin{cases} [1,1] & \text{if} \quad l_1 = u_1 = l_2 = u_2 \\ [0,0] & \text{if} \quad u_1 < l_2 \vee u_2 < l_1 \\ [0,1] & \text{otherwise} \end{cases}$$

Equality:

$$[l_1, u_1] ==^\sharp [l_2, u_2] \;=\; \begin{cases} [1,1] & \text{if} \quad l_1 = u_1 = l_2 = u_2 \\ [0,0] & \text{if} \quad u_1 < l_2 \vee u_2 < l_1 \\ [0,1] & \text{otherwise} \end{cases}$$

Example:

$$\begin{array}{rcl} [42,42] ==^\sharp [42,42] & = & [1,1] \\ [0,7] ==^\sharp [0,7] & = & [0,1] \\ [1,2] ==^\sharp [3,4] & = & [0,0] \end{array}$$

Less:

$$[l_1, u_1] <^{\sharp} [l_2, u_2] = \begin{cases} [1,1] & \text{if} \quad u_1 < l_2 \\ [0,0] & \text{if} \quad u_2 \leq l_1 \\ [0,1] & \text{otherwise} \end{cases}$$

Less:

$$[l_1, u_1] \;<^\sharp\; [l_2, u_2] \;=\; \begin{cases} [1,1] & \text{if} \quad u_1 < l_2 \\ [0,0] & \text{if} \quad u_2 \leq l_1 \\ [0,1] & \text{otherwise} \end{cases}$$

Example:

$$\begin{array}{rcl} [1,2] <^\sharp [9,42] & = & [1,1] \\ [0,7] \;<^\sharp\; [0,7] & = & [0,1] \\ [3,4] \;<^\sharp\; [1,2] & = & [0,0] \end{array}$$

By means of $\mathbb{I}$ we construct the complete lattice:

$$\mathbb{D}_{\mathbb{I}} = (\mathit{Vars} \to \mathbb{I})_{\perp}$$

Description Relation:

$$\rho \;\Delta\; D \qquad \text{iff} \qquad D \neq \perp \quad \wedge \quad \forall\, x \in \mathit{Vars} : \; (\rho\, x) \;\Delta\; (D\; x)$$

The abstract evaluation of expressions is defined analogously to constant propagation. We have:

$$([\![e]\!]\; \rho) \;\Delta\; ([\![e]\!]^{\sharp}\; D) \qquad \text{whenever} \qquad \rho \;\Delta\; D$$

The Effects of Edges:

$$\begin{array}{lcl}
[\![;]\!]^\sharp\, D & = & D \\
[\![x = e;]\!]^\sharp\, D & = & D \oplus \{x \mapsto [\![e]\!]^\sharp\, D\} \\
[\![x = M[e];]\!]^\sharp\, D & = & D \oplus \{x \mapsto \top\} \\
[\![M[e_1] = e_2;]\!]^\sharp\, D & = & D \\
[\![\mathrm{Pos}\,(e)]\!]^\sharp\, D & = & \begin{cases} \bot & \text{if} \quad [0,0] = [\![e]\!]^\sharp\, D \\ D & \text{otherwise} \end{cases} \\
[\![\mathrm{Neg}\,(e)]\!]^\sharp\, D & = & \begin{cases} D & \text{if} \quad [0,0] \sqsubseteq [\![e]\!]^\sharp\, D \\ \bot & \text{otherwise} \end{cases}
\end{array}$$

... given that $D \neq \bot$:-)

Better Exploitation of Conditions:

$$[\![\mathrm{Pos}\,(e)]\!]^\sharp\,D \quad = \quad \begin{cases} \bot & \text{if} \quad [0,0] = [\![e]\!]^\sharp\,D \\ D_1 & \text{otherwise} \end{cases}$$

where :

$$D_1 \quad = \quad \begin{cases} D \oplus \{x \mapsto (D\,x) \sqcap ([\![e_1]\!]^\sharp\,D)\} & \text{if } e \equiv x == e_1 \\ D \oplus \{x \mapsto (D\,x) \sqcap [-\infty, u]\} & \text{if } e \equiv x \;\leq\; e_1, [\![e_1]\!]^\sharp\,D = [\_, u] \\ D \oplus \{x \mapsto (D\,x) \sqcap [l, \infty]\} & \text{if } e \equiv x \;\geq\; e_1, [\![e_1]\!]^\sharp\,D = [l, \_] \end{cases}$$

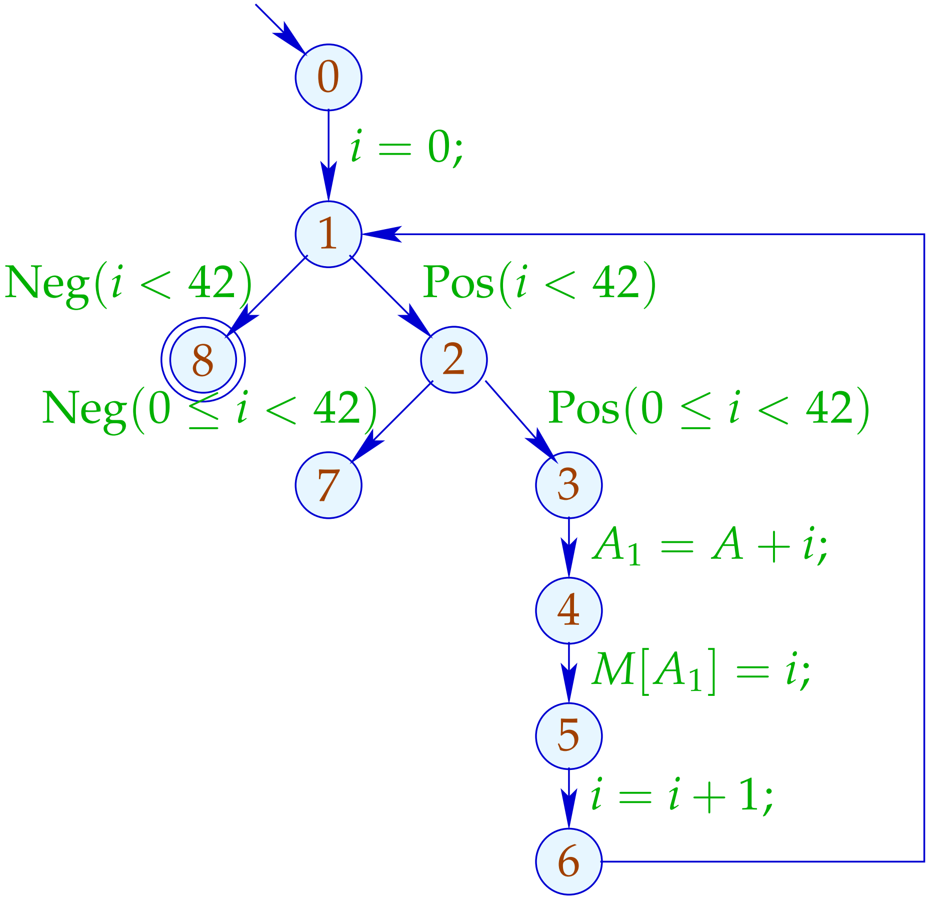
Better Exploitation of Conditions (cont.):

$$[\![\text{Neg}\,(e)]\!]^\sharp\,D \quad = \quad \begin{cases} \bot & \text{if} \quad [0,0] \not\sqsubseteq [\![e]\!]^\sharp\,D \\ D_1 & \text{otherwise} \end{cases}$$

where :

$$D_1 \qquad = \quad \begin{cases} D \oplus \{x \mapsto (D\,x) \sqcap ([\![e_1]\!]^\sharp\,D)\} & \text{if } e \equiv x \ \neq \ e_1 \\ D \oplus \{x \mapsto (D\,x) \sqcap [-\infty, u]\} & \text{if } e \equiv x \ > \ e_1, [\![e_1]\!]^\sharp\,D = [\_, u] \\ D \oplus \{x \mapsto (D\,x) \sqcap [l, \infty]\} & \text{if } e \equiv x \ < \ e_1, [\![e_1]\!]^\sharp\,D = [l, \_] \end{cases}$$

Example:



	i	
	l	u
0	$-\infty$	$+\infty$
1	0	42
2	0	41
3	0	41
4	0	41
5	0	41
6	1	42
7	$\bot$	
8	42	42

Problem:

→ The solution can be computed with RR-iteration — after about 42 rounds :-(

→ On some programs, iteration may never terminate :-((

Idea 1: Widening

- Accelerate the iteration — at the prize of imprecision :-)
- Allow only a bounded number of modifications of values !!!

... in the Example:

- dis-allow updates of interval bounds in $\mathbb{Z}$ ...

$\Longrightarrow$ a maximal chain:

$$[3, 17] \sqsubset [3, +\infty] \sqsubset [-\infty, +\infty]$$

Formalization of the Approach:

Let $\quad x_i \sqsupseteq f_i\,(x_1,\ldots,x_n)\;, \quad i = 1,\ldots,n \qquad\qquad (1)$

denote a system of constraints over $\mathbb{D}$ where the f_i are not necessarily monotonic.

Nonetheless, an accumulating iteration can be defined. Consider the system of equations:

$$x_i \;=\; x_i \sqcup f_i\,(x_1,\ldots,x_n)\;, \quad i = 1,\ldots,n \qquad\qquad (2)$$

We obviously have:

(a) $\underline{x}$ is a solution of (1) iff $\underline{x}$ is a solution of (2).

(b) The function $G : \mathbb{D}^n \to \mathbb{D}^n$ with
$G\,(x_1,\ldots,x_n) = (y_1,\ldots,y_n)\;, \quad y_i = x_i \sqcup f_i\,(x_1,\ldots,x_n)$
is increasing, i.e., $\quad \underline{x} \sqsubseteq G\,\underline{x}$ for all $\underline{x} \in \mathbb{D}^n$.

(c) The sequence $G^k\,\underline{\bot}\,, \quad k \geq 0,$ is an ascending chain:

$$\underline{\bot} \;\sqsubseteq\; G\,\underline{\bot} \;\sqsubseteq\; \ldots \;\sqsubseteq\; G^k\,\underline{\bot} \;\sqsubseteq\; \ldots$$

(d) If $G^k\,\underline{\bot} = G^{k+1}\,\underline{\bot} = \underline{y}$, then $\underline{y}$ is a solution of (1).

(e) If $\mathbb{D}$ has infinite strictly ascending chains, then (d) is not yet sufficient ...

but: we could consider the modified system of equations:

$$x_i = x_i \,\underline{\sqcup}\, f_i(x_1, \ldots, x_n)\,, \quad i = 1, \ldots, n \tag{3}$$

for a binary operation widening:

$$\underline{\sqcup} : \mathbb{D}^2 \to \mathbb{D} \qquad \text{with} \qquad v_1 \sqcup v_2 \;\sqsubseteq v_1 \,\underline{\sqcup}\, v_2$$

(RR)-iteration for (3) still will compute a solution of (1) :-)

... for Interval Analysis:

- The complete lattice is: $\mathbb{D}_{\mathbb{I}} \;=\; (\mathit{Vars} \to \mathbb{I})_\perp$
- the widening $\sqcup\!\!\!\!\!\!\sqcup$ is defined by:

$$\perp \sqcup\!\!\!\!\sqcup D \;=\; D \sqcup\!\!\!\!\sqcup \perp \;=\; D \qquad \text{and for} \quad D_1 \neq \perp \neq D_2:$$

$$\begin{array}{rcl}
(D_1 \sqcup\!\!\!\!\sqcup D_2)\; x & = & (D_1\; x) \sqcup\!\!\!\!\sqcup (D_2\; x) \qquad \text{where} \\
{[l_1, u_1]} \sqcup\!\!\!\!\sqcup [l_2, u_2] & = & [l, u] \qquad \text{with} \\
l & = & \begin{cases} l_1 & \text{if} \quad l_1 \leq l_2 \\ -\infty & \text{otherwise} \end{cases} \\
u & = & \begin{cases} u_1 & \text{if} \quad u_1 \geq u_2 \\ +\infty & \text{otherwise} \end{cases}
\end{array}$$

$\Longrightarrow$ $\sqcup\!\!\!\!\sqcup$ is not commutative !!!

Example:

$$\begin{array}{rcl} [0,2] \sqcup [1,2] & = & [0,2] \\ [1,2] \sqcup [0,2] & = & [-\infty,2] \\ [1,5] \sqcup [3,7] & = & [1,+\infty] \end{array}$$

→ Widening returns larger values more quickly.

→ It should be constructed in such a way that termination of iteration is guaranteed :-)

→ For interval analysis, widening bounds the number of iterations by:

$$\#\mathit{points} \cdot (1 + 2 \cdot \#\mathit{Vars})$$

Conclusion:

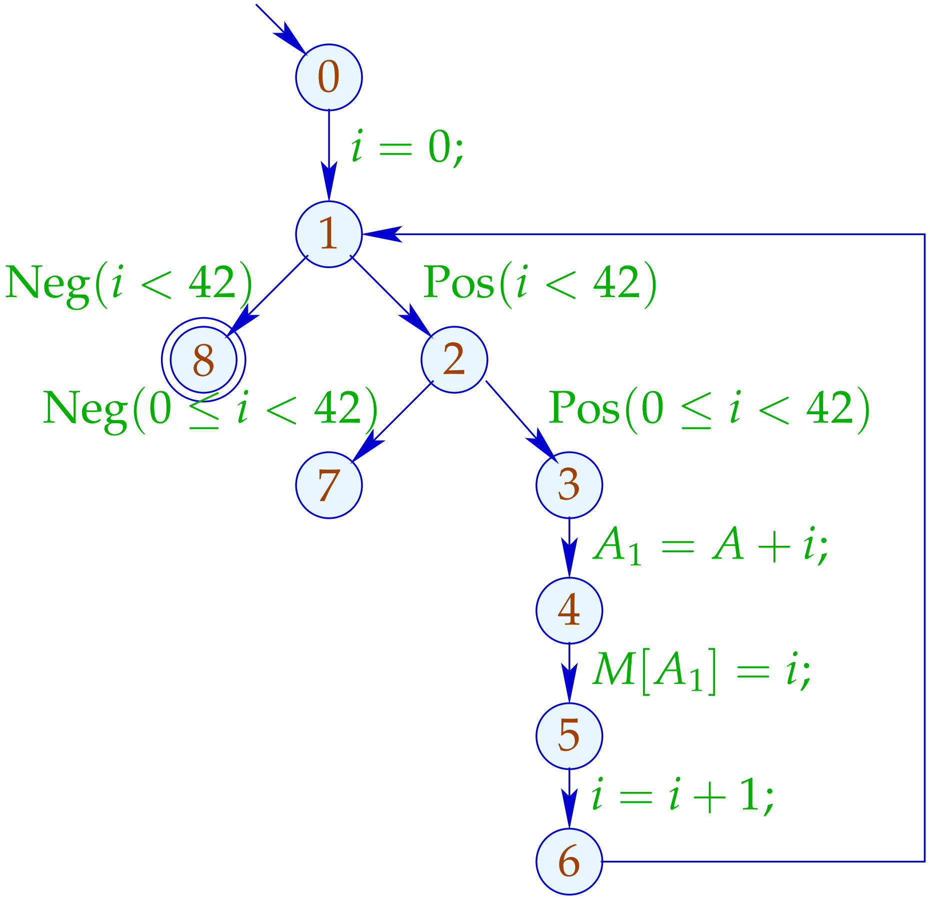
- In order to determine a solution of (1) over a complete lattice with infinite ascending chains, we define a suitable widening and then solve (3) :-)
- Warning: The construction of suitable widenings is a dark art !!!

  Often $\sqcup$ is chosen dynamically during iteration such that

  → the abstract values do not get too complicated;

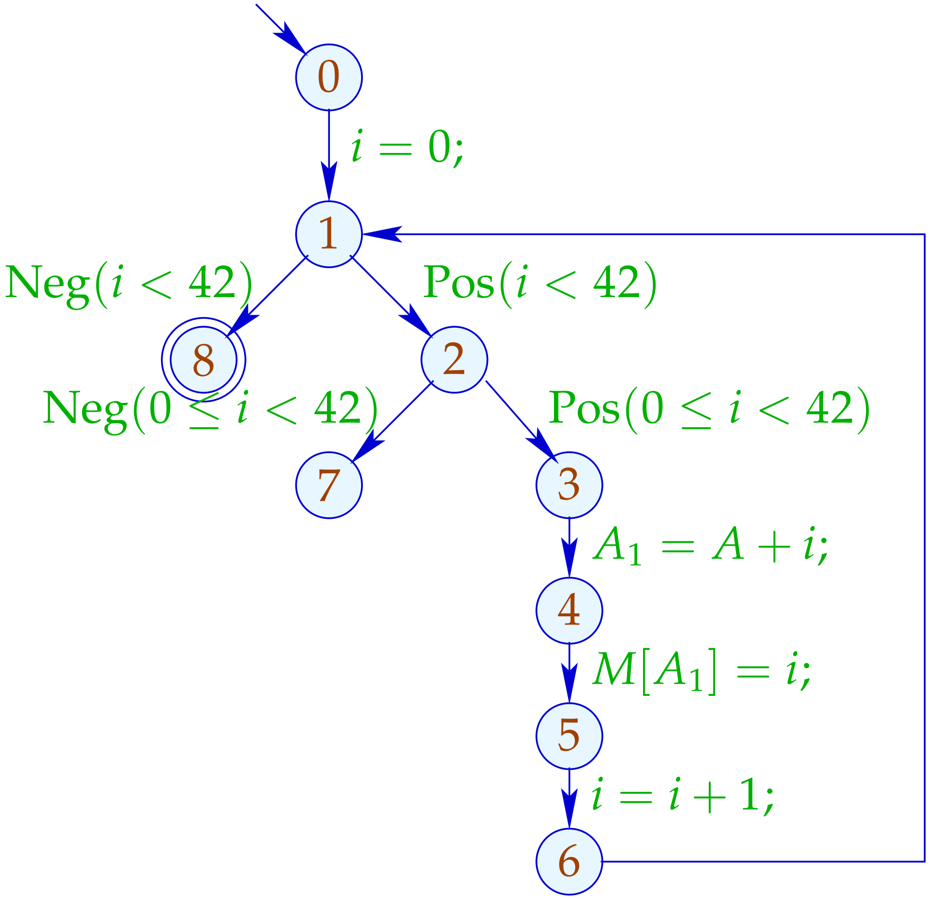
  → the number of updates remains bounded ...

Our Example:



	1	
	l	u
0	$-\infty$	$+\infty$
1	0	0
2	0	0
3	0	0
4	0	0
5	0	0
6	1	1
7	$\perp$	
8	$\perp$	

Our Example:



	1		2		3	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$	dito	
1	0	0	0	$+\infty$		
2	0	0	0	$+\infty$		
3	0	0	0	$+\infty$		
4	0	0	0	$+\infty$		
5	0	0	0	$+\infty$		
6	1	1	1	$+\infty$		
7	$\perp$		42	$+\infty$		
8	$\perp$		42	$+\infty$		

... obviously, the result is disappointing :-(

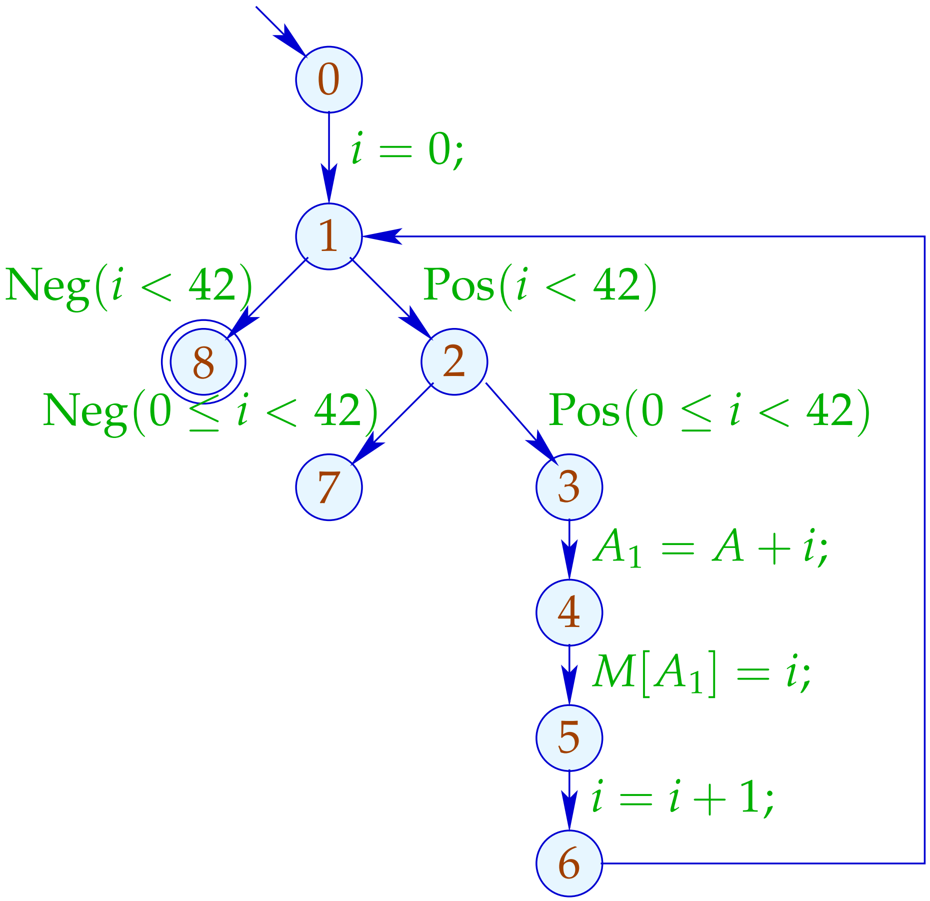
Idea 2:

In fact, acceleration with $\sqcup$ need only be applied at sufficiently many places!

A set I is a loop separator, if every loop contains at least one point from I :-)

If we apply widening only at program points from such a set I, then RR-iteration still terminates !!!

In our Example:

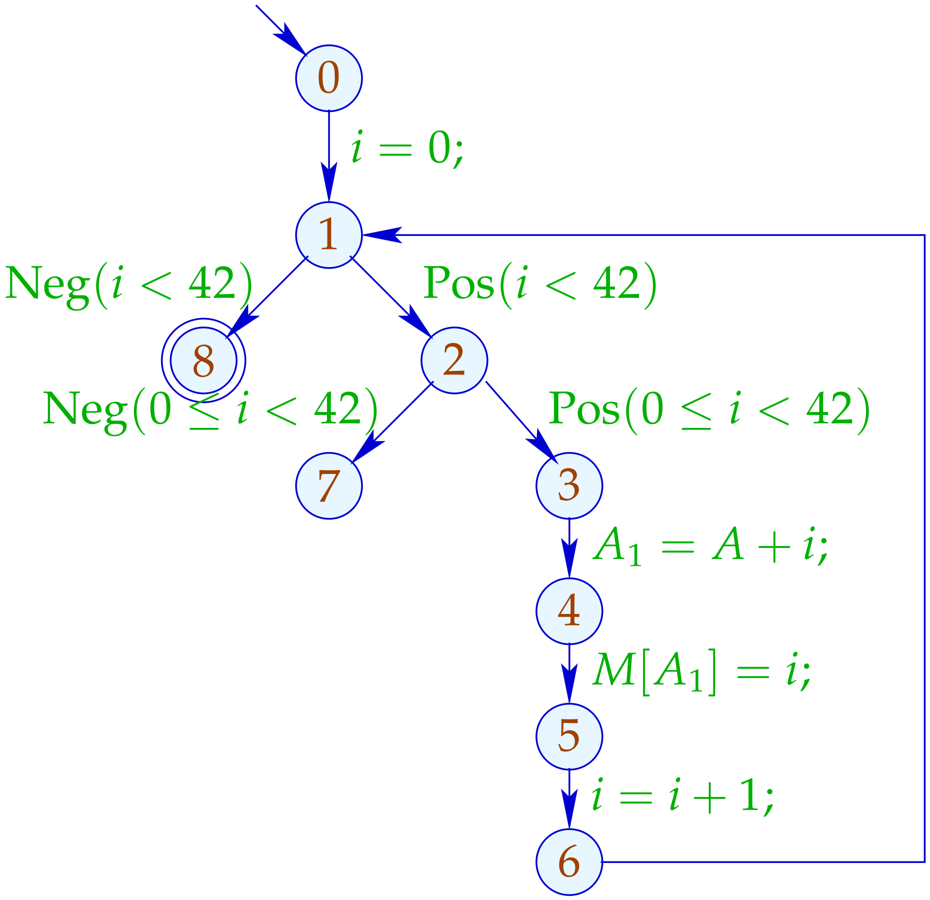


$$I_1 = \{1\} \quad \text{or:}$$

$$I_2 = \{2\} \quad \text{or:}$$

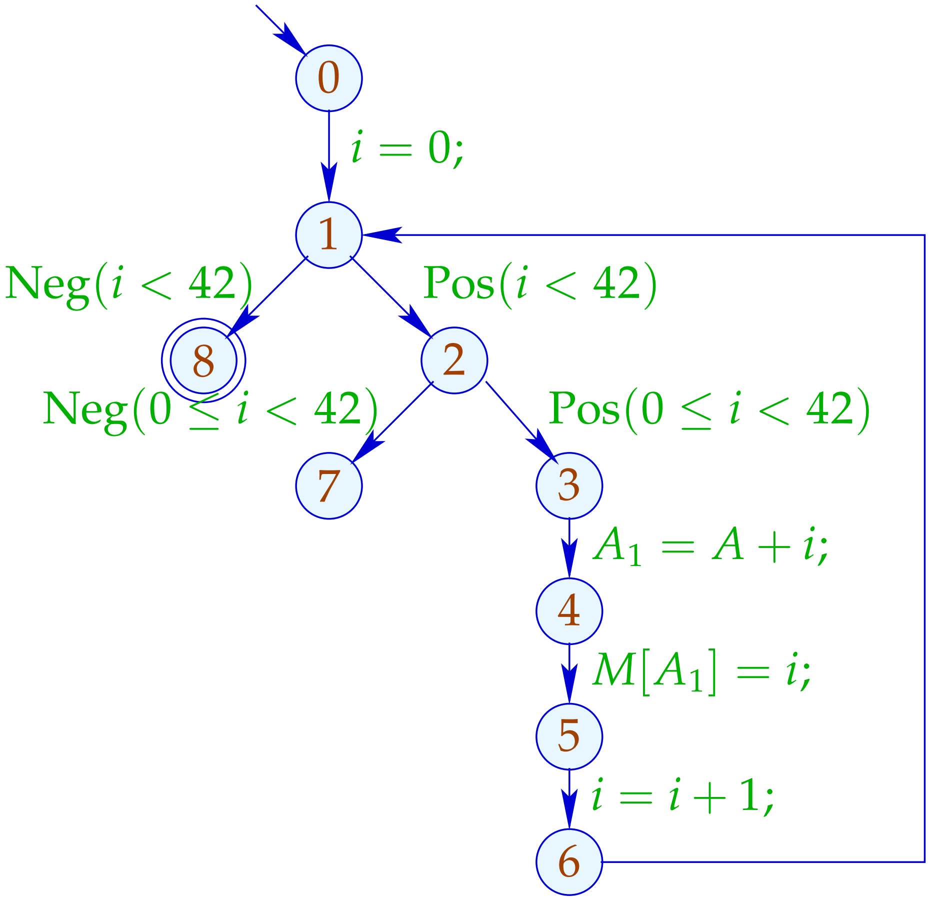
$$I_3 = \{3\}$$

The Analysis with $I = \{1\}$:



	1		2		3	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$	dito	
1	0	0	0	$+\infty$		
2	0	0	0	41		
3	0	0	0	41		
4	0	0	0	41		
5	0	0	0	41		
6	1	1	1	42		
7	$\perp$		$\perp$			
8	$\perp$		42	$+\infty$		

The Analysis with $I = \{2\}$:



	1		2		3	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$	dito	
1	0	0	0	42		
2	0	0	0	$+\infty$		
3	0	0	0	41		
4	0	0	0	41		
5	0	0	0	41		
6	1	1	1	42		
7	$\perp$		42	$+\infty$		
8	$\perp$		42	42		

Discussion:

- Both runs of the analysis determine interesting information :-)
- The run with $I = \{2\}$ proves that always $i = 42$ after leaving the loop.
- Only the run with $I = \{1\}$ finds, however, that the outer check makes the inner check superfluous :-(

How can we find a suitable loop separator I ???

Idea 3: Narrowing

Let $\underline{x}$ denote any solution of (1) , i.e.,

$$x_i \sqsupseteq f_i\,\underline{x} \;, \qquad i = 1, \ldots, n$$

Then for monotonic f_i ,

$$\underline{x} \sqsupseteq F\,\underline{x} \sqsupseteq F^2\,\underline{x} \sqsupseteq \ldots \sqsupseteq F^k\,\underline{x} \sqsupseteq \ldots$$

// Narrowing Iteration

Idea 3: Narrowing

Let $\underline{x}$ denote any solution of (1) , i.e.,

$$x_i \sqsupseteq f_i\, \underline{x} \;, \qquad i = 1, \ldots, n$$

Then for monotonic f_i ,

$$\underline{x} \sqsupseteq F\, \underline{x} \sqsupseteq F^2\, \underline{x} \sqsupseteq \ldots \sqsupseteq F^k\, \underline{x} \sqsupseteq \ldots$$

// Narrowing Iteration

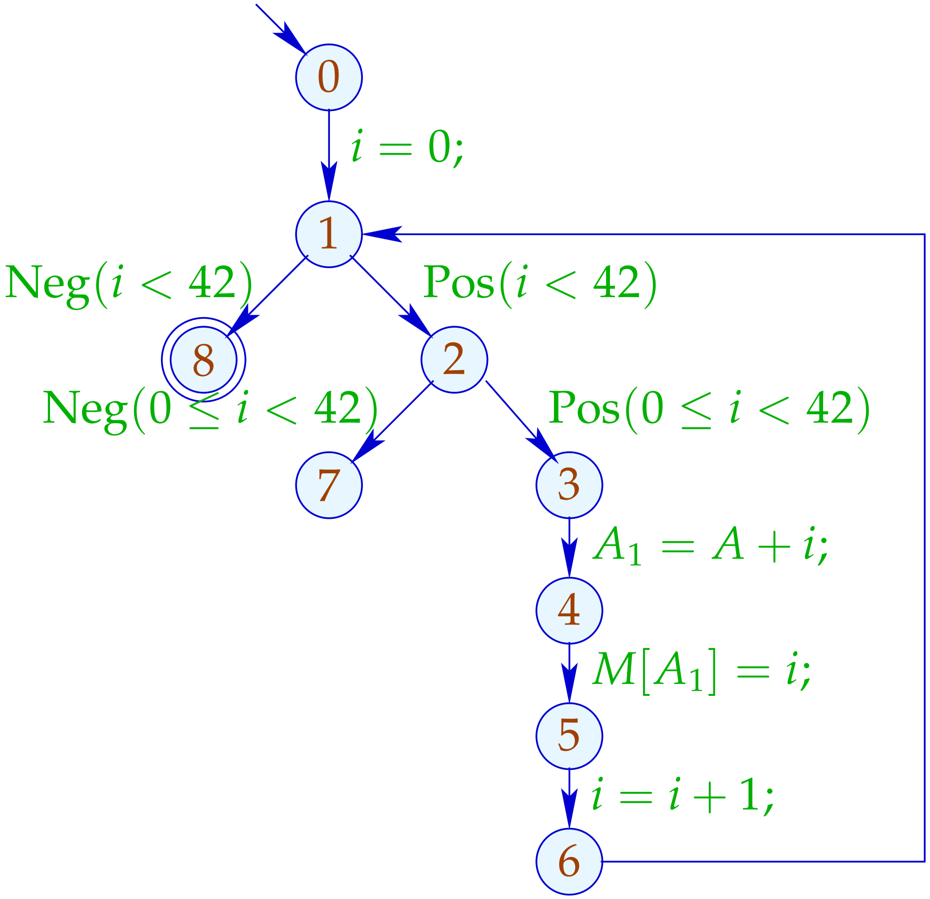
Every tuple $F^k\, \underline{x}$ is a solution of (1) :-)

⟹

Termination is no problem anymore:
we stop whenever we want :-))

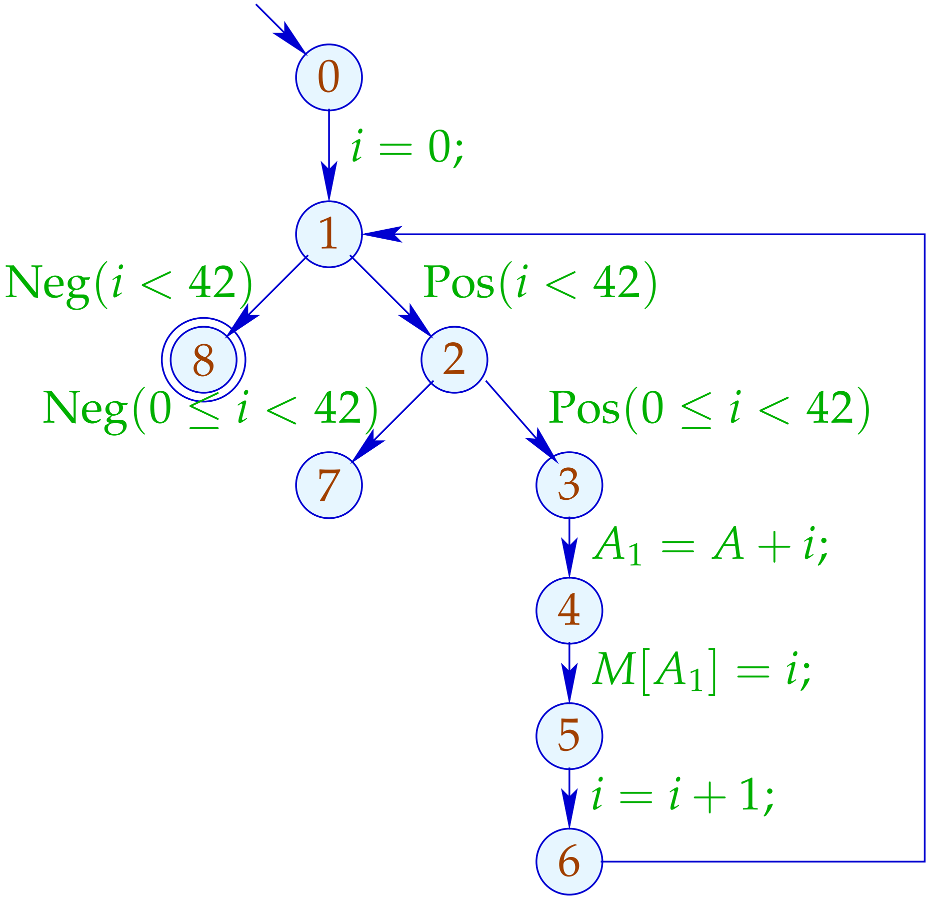
// The same also holds for RR-iteration.

Narrowing Iteration in the Example:



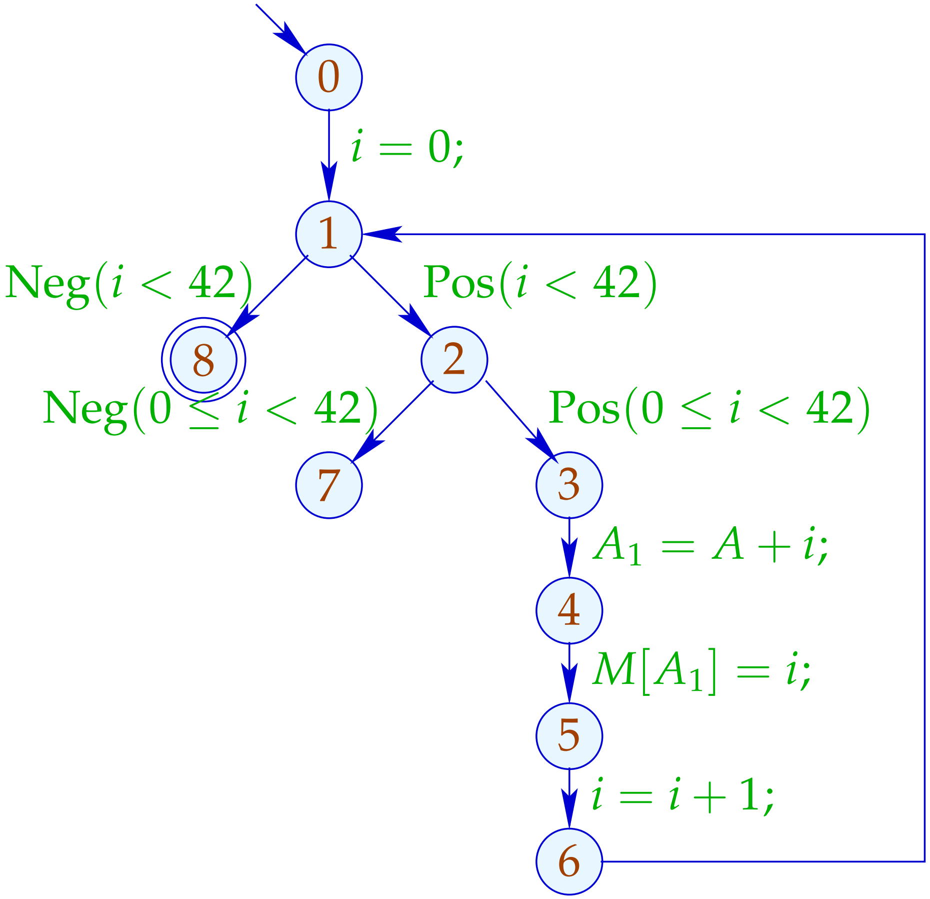
	0	
	l	u
0	$-\infty$	$+\infty$
1	0	$+\infty$
2	0	$+\infty$
3	0	$+\infty$
4	0	$+\infty$
5	0	$+\infty$
6	1	$+\infty$
7	42	$+\infty$
8	42	$+\infty$

Narrowing Iteration in the Example:



	0		1	
	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$
1	0	$+\infty$	0	$+\infty$
2	0	$+\infty$	0	41
3	0	$+\infty$	0	41
4	0	$+\infty$	0	41
5	0	$+\infty$	0	41
6	1	$+\infty$	1	42
7	42	$+\infty$	$\perp$	
8	42	$+\infty$	42	$+\infty$

Narrowing Iteration in the Example:



	0		1		2	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$
1	0	$+\infty$	0	$+\infty$	0	42
2	0	$+\infty$	0	41	0	41
3	0	$+\infty$	0	41	0	41
4	0	$+\infty$	0	41	0	41
5	0	$+\infty$	0	41	0	41
6	1	$+\infty$	1	42	1	42
7	42	$+\infty$	$\perp$		$\perp$	
8	42	$+\infty$	42	$+\infty$	42	42

Discussion:

→ We start with a safe approximation.

→ We find that the inner check is redundant :-)

→ We find that at exit from the loop, always $i = 42$:-))

→ It was not necessary to construct an optimal loop separator :-)))

Last Question:

Do we have to accept that narrowing may not terminate ???

4. Idea: Accelerated Narrowing

Assume that we have a solution $\underline{x} = (x_1, \ldots, x_n)$ of the system of constraints:

$$x_i \;\sqsupseteq\; f_i\,(x_1, \ldots, x_n)\;, \quad i = 1, \ldots, n \qquad (1)$$

Then consider the system of equations:

$$x_i \;=\; x_i \sqcap f_i\,(x_1, \ldots, x_n)\;, \quad i = 1, \ldots, n \qquad (4)$$

Obviously, we have for monotonic f_i : $H^k\,\underline{x} \;=\; F^k\,\underline{x}$:-)

where $H\,(x_1, \ldots, x_n) = (y_1, \ldots, y_n)\;, \quad y_i = x_i \sqcap f_i\,(x_1, \ldots, x_n)$.

In (4) , we replace $\sqcap$ durch by the novel operator $\bar{\sqcap}$ where:

$$a_1 \sqcap a_2 \;\sqsubseteq\; a_1 \bar{\sqcap}\, a_2 \;\sqsubseteq\; a_1$$

... for Interval Analysis:

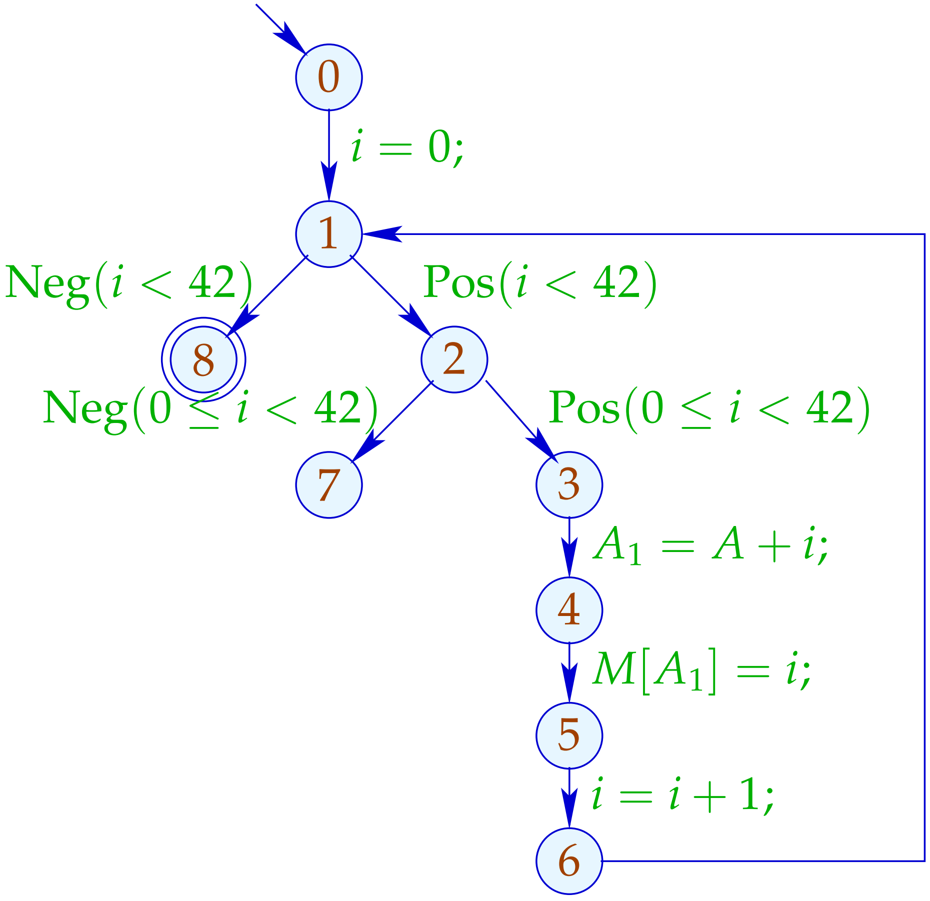
We preserve finite interval bounds   :-)

Therefore, $\quad \bot \mathbin{\overline{\sqcap}} D \;=\; D \mathbin{\overline{\sqcap}} \bot \;=\; \bot \quad$ and for $\quad D_1 \neq \bot \neq D_2$:

$$\begin{array}{rcl}
(D_1 \mathbin{\overline{\sqcap}} D_2)\, x & = & (D_1\, x) \mathbin{\overline{\sqcap}} (D_2\, x) \qquad \text{where} \\
[l_1, u_1] \mathbin{\overline{\sqcap}} [l_2, u_2] & = & [l, u] \qquad \text{with} \\
l & = & \begin{cases} l_2 & \text{if} \quad l_1 = -\infty \\ l_1 & \text{otherwise} \end{cases} \\
u & = & \begin{cases} u_2 & \text{if} \quad u_1 = \infty \\ u_1 & \text{otherwise} \end{cases}
\end{array}$$

$\Longrightarrow \quad \overline{\sqcap} \quad$ is not commutative !!!

Accelerated Narrowing in the Example:



	0		1		2	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$
1	0	$+\infty$	0	$+\infty$	0	42
2	0	$+\infty$	0	41	0	41
3	0	$+\infty$	0	41	0	41
4	0	$+\infty$	0	41	0	41
5	0	$+\infty$	0	41	0	41
6	1	$+\infty$	1	42	1	42
7	42	$+\infty$	$\perp$		$\perp$	
8	42	$+\infty$	42	$+\infty$	42	42

Discussion:

→ Warning: Widening also returns for non-monotonic f_i a solution. Narrowing is only applicable to monotonic f_i !!

→ In the example, accelerated narrowing already returns the optimal result :-)

→ If the operator $\sqcap\!\!\!\!\!\!\;\overline{}$ only allows for finitely many improvements of values, we may execute narrowing until stabilization.

→ In case of interval analysis these are at most:

$$\#points \cdot (1 + 2 \cdot \#Vars)$$

1.6 Pointer Analysis

Questions:

→ Are two addresses possibly equal?

→ Are two addresses definitively equal?

1.6 Pointer Analysis

Questions:

- → Are two addresses possibly equal? May Alias
- → Are two addresses definitively equal? Must Alias

⟹ Alias Analysis

The analyses so far without alias information:

(1) Available Expressions:

- Extend the set *Expr* of expressions by occurring loads $M[e]$.

- Extend the Effects of Edges:

$$\begin{array}{lcl} [\![x = e;]\!]^\sharp\, A & = & (A \cup \{e\})\backslash \mathit{Expr}_x \\ [\![x = M[e];]\!]^\sharp\, A & = & (A \cup \{e, M[e]\})\backslash \mathit{Expr}_x \\ [\![M[e_1] = e_2;]\!]^\sharp\, A & = & (A \cup \{e_1, e_2\})\backslash \mathit{Loads} \end{array}$$

(2) Values of Variables:

- Extend the set *Expr* of expressions by occurring loads $M[e]$.

- Extend the Effects of Edges:

$$[\![x = M[e];]\!]^\sharp\, V\, e' \quad = \begin{cases} \{x\} & \text{if} \quad e' = M[e] \\ \emptyset & \text{if} \quad e' = e \\ V\, e' \backslash \{x\} & \text{otherwise} \end{cases}$$

$$[\![M[e_1] = e_2;]\!]^\sharp\, V\, e' \quad = \begin{cases} \emptyset & \text{if} \quad e' \in \{e_1, e_2\} \\ V\, e' & \text{otherwise} \end{cases}$$

(3) Constant Propagation:

- Extend the abstract state by an abstract store M

- Execute accesses to known memory locations!

$$[\![x = M[e];]\!]^\sharp\,(D, M) \quad = \quad \begin{cases} (D \oplus \{x \mapsto M\,a\}, M) & \text{if} \\ & [\![e]\!]^\sharp\,D = a \sqsubset \top \\ (D \oplus \{x \mapsto \top\}, M) & \text{otherwise} \end{cases}$$

$$[\![M[e_1] = e_2;]\!]^\sharp\,(D, M) \quad = \quad \begin{cases} (D, M \oplus \{a \mapsto [\![e_2]\!]^\sharp D\}) & \text{if} \\ & [\![e_1]\!]^\sharp\,D = a \sqsubset \top \\ (D, \underline{\top}) & \text{otherwise} \end{cases} \qquad \text{where}$$

$$\underline{\top}\,a \quad = \quad \top \qquad\qquad (a \in \mathbb{N})$$

Problems:

- Addresses are from $\mathbb{N}$:-(

  There are no infinite strictly ascending chains, but ...
- Exact addresses at compile-time are rarely known :-(
- At the same program point, typically different addresses are accessed ...
- Storing at an unknown address destroys all information M :-(

$\Longrightarrow$ constant propagation fails :-(

$\Longrightarrow$ memory accesses/pointers kill precision :-(

Simplification:

- We consider pointers to the beginning of blocks A which allow indexed accesses $A[i]$:-)
- We ignore well-typedness of the blocks.
- New statements:

$$
\begin{array}{lll}
x = \mathsf{new}(); & // & \text{allocation of a new block} \\
x = y[e]; & // & \text{indexed read access to a block} \\
y[e_1] = e_2; & // & \text{indexed write access to a block}
\end{array}
$$

- Blocks are possibly infinite :-)
- For simplicity, all pointers point to the beginning of a block.

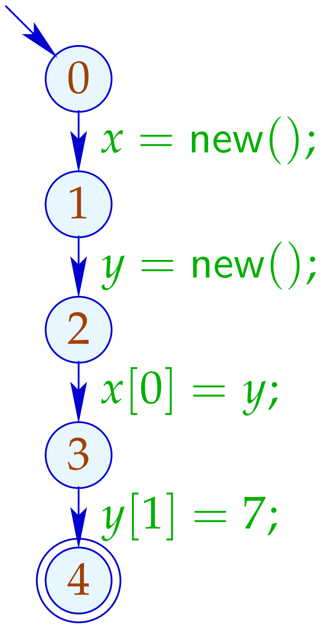
Simple Example:

$x = \mathsf{new}();$

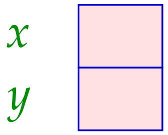
$y = \mathsf{new}();$

$x[0] = y;$

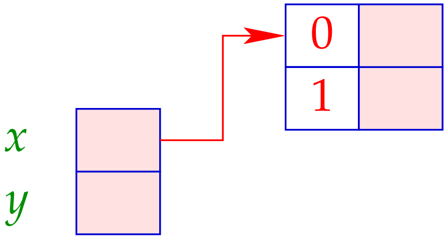
$y[1] = 7;$



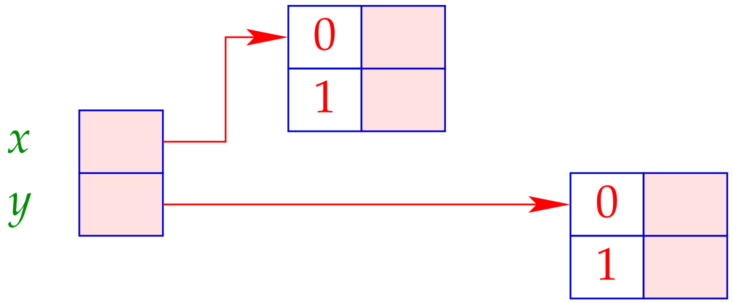
The Semantics:



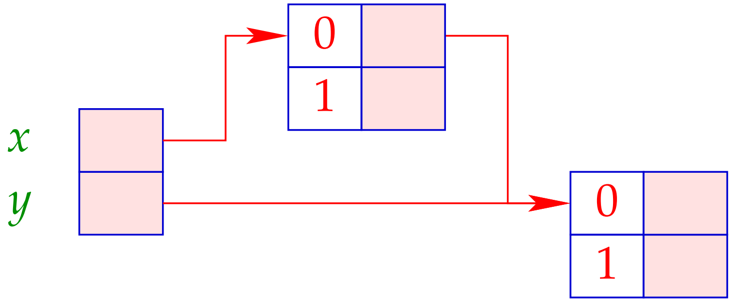
The Semantics:



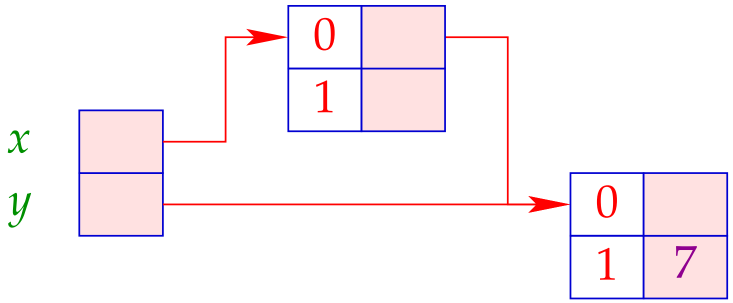
The Semantics:



The Semantics:

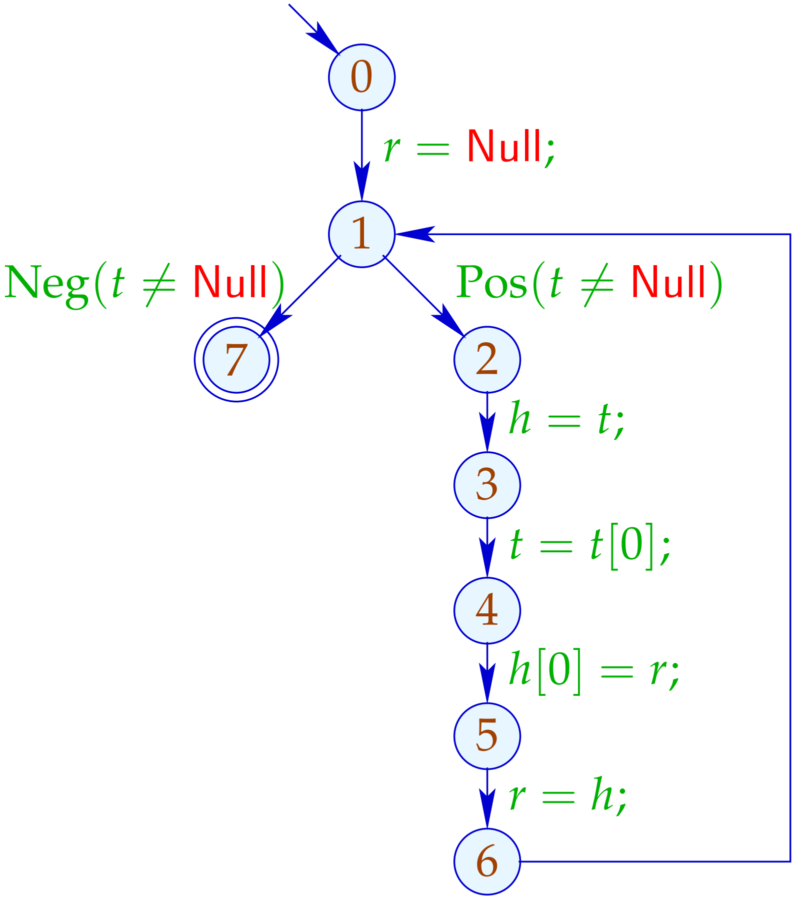


The Semantics:



More Complex Example:

```
$r = \text{Null};$
while $(t \neq \text{Null})$ {
        $h = t;$
        $t = t[0];$
        $h[0] = r;$
        $r = h;$
}
```


0
$r = \text{Null};$
1
$\text{Neg}(t \neq \text{Null})$
$\text{Pos}(t \neq \text{Null})$
7
2
$h = t;$
3
$t = t[0];$
4
$h[0] = r;$
5
$r = h;$
6

Concrete Semantics:

A store consists of a finite collection of blocks.

After h new-operations we obtain:

$$\begin{array}{lcll} \mathit{Addr}_h & = & \{\mathsf{ref}\ a \mid 0 \leq a < h\} & // \quad \text{addresses} \\ \mathit{Val}_h & = & \mathit{Addr}_h \cup \mathbb{Z} & // \quad \text{values} \\ \mathit{Store}_h & = & (\mathit{Addr}_h \times \mathbb{N}_0) \to \mathit{Val}_h & // \quad \text{store} \\ \mathit{State}_h & = & (\mathit{Vars} \to \mathit{Val}_h) \times \mathit{Store}_h & // \quad \text{states} \end{array}$$

For simplicity, we set: $0 = \mathsf{Null}$

Let $(\rho, \mu) \in State_h$. Then we obtain for the new edges:

$$\begin{array}{lll}
[\![x = \mathsf{new}();]\!]\,(\rho,\mu) & = & (\rho \oplus \{x \mapsto \mathsf{ref}\, h\}, \\
& & \;\mu \oplus \{(\mathsf{ref}\, h, i) \mapsto 0, (i \in \mathbb{N}_0) \\
[\![x = y[e];]\!]\,(\rho,\mu) & = & (\rho \oplus \{x \mapsto \mu\,(\rho\, y, [\![e]\!]\,\rho)\}, \mu) \\
[\![y[e_1] = e_2;]\!]\,(\rho,\mu) & = & (\rho, \mu \oplus \{(\rho\, y, [\![e_1]\!]\,\rho) \mapsto \rho\,[\![e_2]\!]\,\rho\})
\end{array}$$

Warning:

This semantics is too detailled in that it computes with absolute Addresses. Accordingly, the two programs:

$$\begin{array}{lcl} x = \mathsf{new}(); & \qquad\qquad & y = \mathsf{new}(); \\ y = \mathsf{new}(); & & x = \mathsf{new}(); \end{array}$$

are not considered as equivalent !!?

Possible Solution:

Define equivalence only up to permutation of addresses :-)

Alias Analysis 1. Idea:

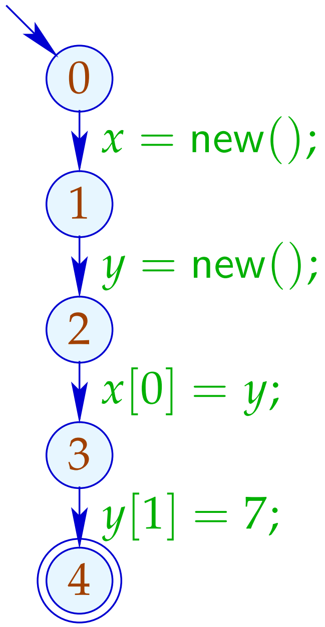
- Distinguish finitely many classes of blocks.
- Collect all addresses of a block into one set!
- Use sets of addresses as abstract values!

$\Longrightarrow$ Points-to-Analysis

$$\begin{array}{lcllll}
\mathit{Addr}^\sharp & = & \mathit{Edges} & & // & \text{creation edges} \\
\mathit{Val}^\sharp & = & 2^{\mathit{Addr}^\sharp} & & // & \text{abstract values} \\
\mathit{Store}^\sharp & = & \mathit{Addr}^\sharp \to \mathit{Val}^\sharp & & // & \text{abstract store} \\
\mathit{State}^\sharp & = & (\mathit{Vars} \to \mathit{Val}^\sharp) \times \mathit{Store}^\sharp & & // & \text{abstract states}
\end{array}$$

// complete lattice !!!

... in the Simple Example:



	x	y	$(0,1)$
0	$\emptyset$	$\emptyset$	$\emptyset$
1	$\{(0,1)\}$	$\emptyset$	$\emptyset$
2	$\{(0,1)\}$	$\{(1,2)\}$	$\emptyset$
3	$\{(0,1)\}$	$\{(1,2)\}$	$\{(1,2)\}$
4	$\{(0,1)\}$	$\{(1,2)\}$	$\{(1,2)\}$

The Effects of Edges:

$$
\begin{array}{lcl}
[\![(\_,;,\_)]\!]^\sharp\,(D,M) & = & (D,M) \\
[\![(\_,\mathrm{Pos}(e),\_)]\!]^\sharp\,(D,M) & = & (D,M) \\
[\![(\_,x=y;,\_)]\!]^\sharp\,(D,M) & = & (D\oplus\{x\mapsto D\,y\},M) \\
[\![(\_,x=e;,\_)]\!]^\sharp\,(D,M) & = & (D\oplus\{x\mapsto\emptyset\},M) \qquad , \qquad e\notin \mathit{Vars} \\
\\
[\![(u,x=\mathsf{new}();,v)]\!]^\sharp\,(D,M) & = & (D\oplus\{x\mapsto\{(u,v)\}\},M) \\
[\![(\_,x=y[e];,\_)]\!]^\sharp\,(D,M) & = & (D\oplus\{x\mapsto\bigcup\{M(f)\mid f\in D\,y\}\},M) \\
[\![(\_,y[e_1]=x;,\_)]\!]^\sharp\,(D,M) & = & (D,M\oplus\{f\mapsto(M\,f\cup D\,x)\mid f\in D\,y\})
\end{array}
$$

Warning:

- The value Null has been ignored. Dereferencing of Null or negative indices are not detected :-(
- Destructive updates are only possible for variables, not for blocks in storage!

  $\Longrightarrow$ no information, if not all block entries are initialized before use :-((
- The effects now depend on the edge itself.

  The analysis cannot be proven correct w.r.t. the reference semantics :-(

  In order to prove correctness, we first instrument the concrete semantics with extra information which records where a block has been created.

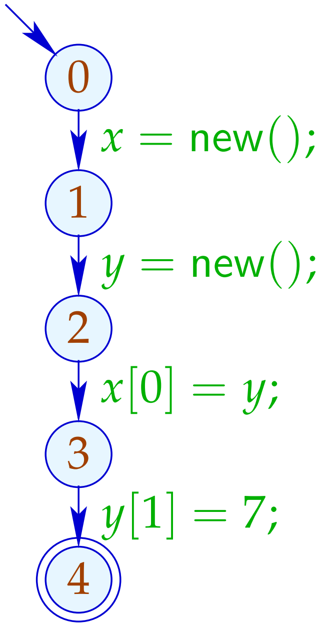
...

- We compute possible points-to information.
- From that, we can extract may-alias information.
- The analysis can be rather expensive — without finding very much :-(
- Separate information for each program point can perhaps be abandoned ??

Alias Analysis   2. Idea:

Compute for each variable and address a value which safely approximates the values at every program point simultaneously !

... in the Simple Example:



$x = \mathsf{new}();$

$y = \mathsf{new}();$

$x[0] = y;$

$y[1] = 7;$

x	$\{(0,1)\}$
y	$\{(1,2)\}$
$(0,1)$	$\{(1,2)\}$
$(1,2)$	$\emptyset$

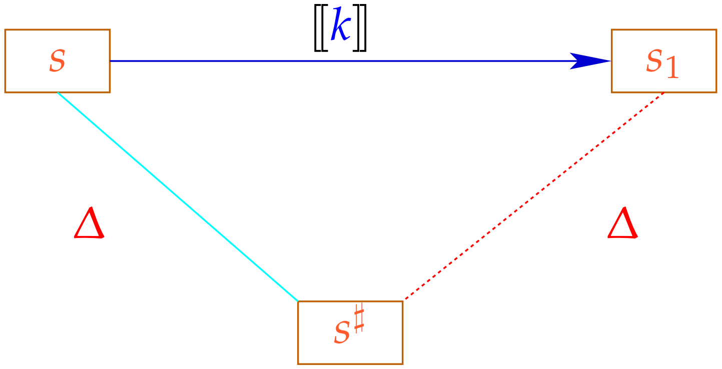
Each edge (u, lab, v) gives rise to constraints:

lab	$Constraint$
$x = y;$	$\mathcal{P}[x] \supseteq \mathcal{P}[y]$
$x = \mathsf{new}();$	$\mathcal{P}[x] \supseteq \{(u, v)\}$
$x = y[e];$	$\mathcal{P}[x] \supseteq \bigcup\{\mathcal{P}[f] \mid f \in \mathcal{P}[y]\}$
$y[e_1] = x;$	$\mathcal{P}[f] \supseteq (f \in \mathcal{P}[y])\,?\,\mathcal{P}[x] : \emptyset$ for all $f \in Addr^\sharp$

Other edges have no effect :-)

Discussion:

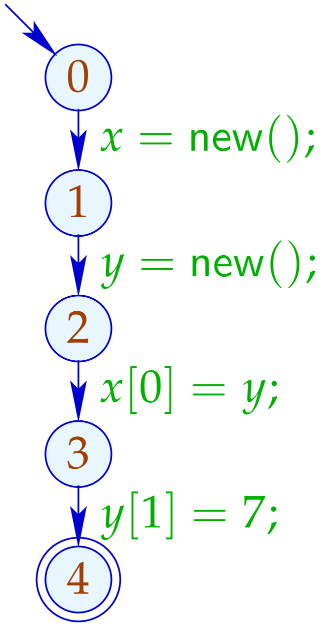
- The resulting constraint system has size $\mathcal{O}(k \cdot n)$ for k abstract addresses and n edges :-(
- The number of necessary iterations is $\mathcal{O}(k)$ ...
- The computed information is perhaps still too zu precise !!?
- In order to prove correctness of a solution $s^\sharp \in States^\sharp$ we show:



Alias Analysis 3. Idea:

Determine one equivalence relation $\equiv$ on variables x and memory accesses $y[\,]$ with $s_1 \equiv s_2$ whenever s_1, s_2 may contain the same address at some u_1, u_2

... in the Simple Example:



$$\equiv \;=\; \{\{x\},\\ \{y, x[\,]\},\\ \{y[\,]\}\}$$

Discussion:

→ We compute a single information fo the whole program.

→ The computation of this information maintains partitions $\pi = \{P_1, \ldots, P_m\}$:-)

→ Individual sets P_i are identified by means of representatives $p_i \in P_i$.

→ The operations on a partition π are:

$$\begin{array}{lcl} \textsf{find}\,(\pi, p) & = & p_i \qquad \text{if } p \in P_i \\ & // & \text{returns the representative} \\ \textsf{union}\,(\pi, p_{i_1}, p_{i_2}) & = & \{P_{i_1} \cup P_{i_2}\} \cup \{P_j \mid i_1 \neq j \neq i_2\} \\ & // & \text{unions the represented classes} \end{array}$$

→ If $x_1, x_2 \in \mathit{Vars}$ are equivalent, then also $x_1[\,]$ and $x_2[\,]$ must be equivalent :-)

→ If $P_i \cap \mathit{Vars} \neq \emptyset$, then we choose $p_i \in \mathit{Vars}$. Then we can apply union recursively :

```
union* (π, q_1, q_2)  =  let  $p_{i_1}$  =  find (π, q_1)
                               $p_{i_2}$  =  find (π, q_2)
                         in  if $p_{i_1} == p_{i_2}$ then π
                             else  let π  =  union (π, $p_{i_1}$, $p_{i_2}$)
                                   in  if $p_{i_1}, p_{i_2} \in \mathit{Vars}$ then
                                           union* (π, $p_{i_1}[\,]$, $p_{i_2}[\,]$)
```

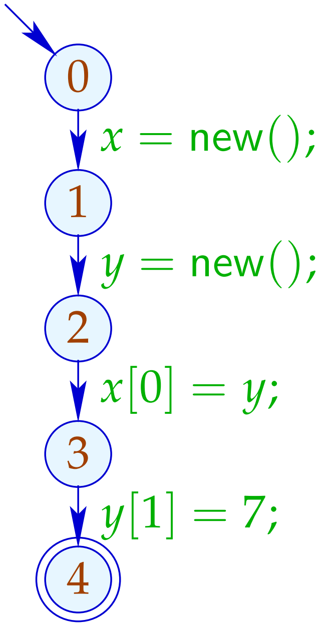
The analysis iterates over all edges once:

```
$\pi = \{\{x\}, \{x[\,]\} \mid x \in \mathit{Vars}\};$
forall   $k = (\_, lab, \_)$   do   $\pi = [\![lab]\!]^\sharp\,\pi;$
```

where:

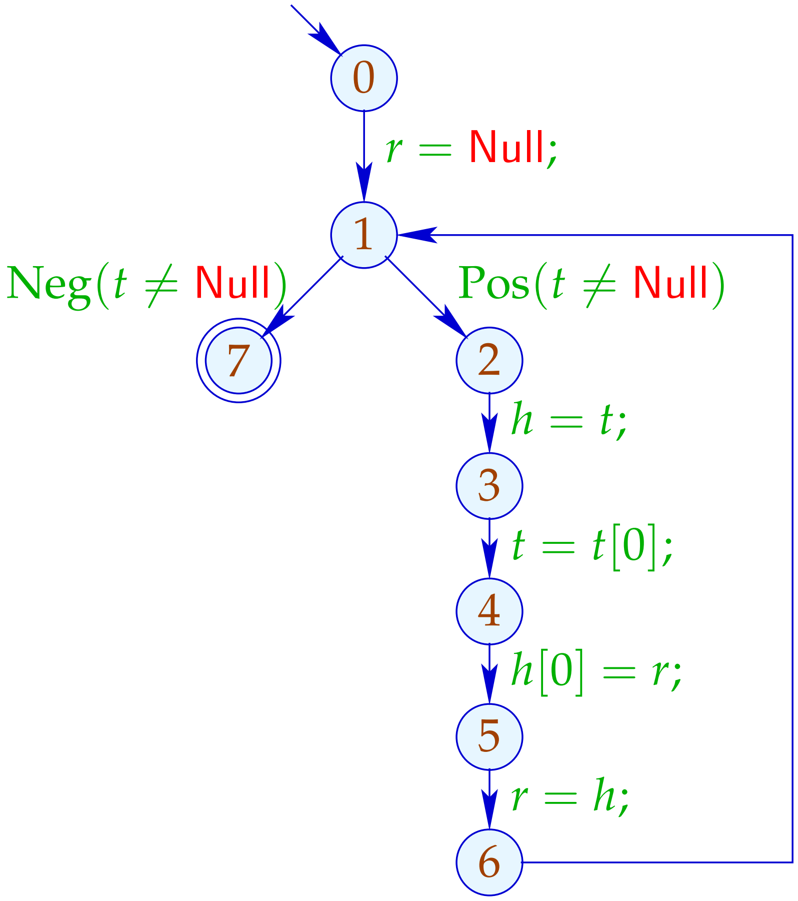
$$\begin{array}{lcll}
[\![x = y;]\!]^\sharp\,\pi & = & \mathsf{union}^*\,(\pi, x, y) & \\
[\![x = y[e];]\!]^\sharp\,\pi & = & \mathsf{union}^*\,(\pi, x, y[\,]) & \\
[\![y[e] = x;]\!]^\sharp\,\pi & = & \mathsf{union}^*\,(\pi, x, y[\,]) & \\
[\![lab]\!]^\sharp\,\pi & = & \pi & \text{otherwise}
\end{array}$$

... in the Simple Example:



	$\{\{x\},\{y\},\{x[\,]\},\{y[\,]\}\}$
$(0,1)$	$\{\{x\},\{y\},\{x[\,]\},\{y[\,]\}\}$
$(1,2)$	$\{\{x\},\{y\},\{x[\,]\},\{y[\,]\}\}$
$(2,3)$	$\{\{x\},\boxed{\{y,x[\,]\}},\{y[\,]\}\}$
$(3,4)$	$\{\{x\},\{y,x[\,]\},\{y[\,]\}\}$

... in the More Complex Example:



	$\{\{h\},\{r\},\{t\},\{h[\,]\},\{t[\,]\}\}$
$(2,3)$	$\{\boxed{\{h,t\}},\{r\},\boxed{\{h[\,],t[\,]\}}\}$
$(3,4)$	$\{\boxed{\{h,t,h[\,],t[\,]\}},\{r\}\}$
$(4,5)$	$\{\boxed{\{h,t,r,h[\,],t[\,]\}}\}$
$(5,6)$	$\{\{h,t,r,h[\,],t[\,]\}\}$

Warning:

In order to find something, we must assume that variables / addresses always receive a value before they are accessed.

Complexity:

we havve:

$\mathcal{O}(\#\,edges + \#\,Vars)$	calls of	union*
$\mathcal{O}(\#\,edges + \#\,Vars)$	calls of	find
$\mathcal{O}(\#\,Vars)$	calls of	union

$\Longrightarrow$ We require efficient Union-Find data-structure :-)

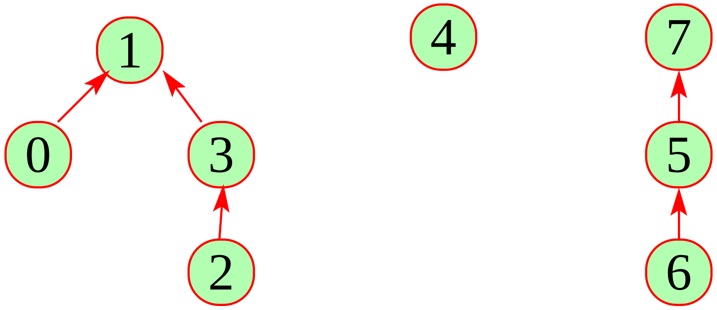
Idea:

Represent partition of U as directed forest:

- For $u \in U$ a reference $F[u]$ to the father is maintained;
- Roots are elements u with $F[u] = u$.

Single trees represent equivalence classes.

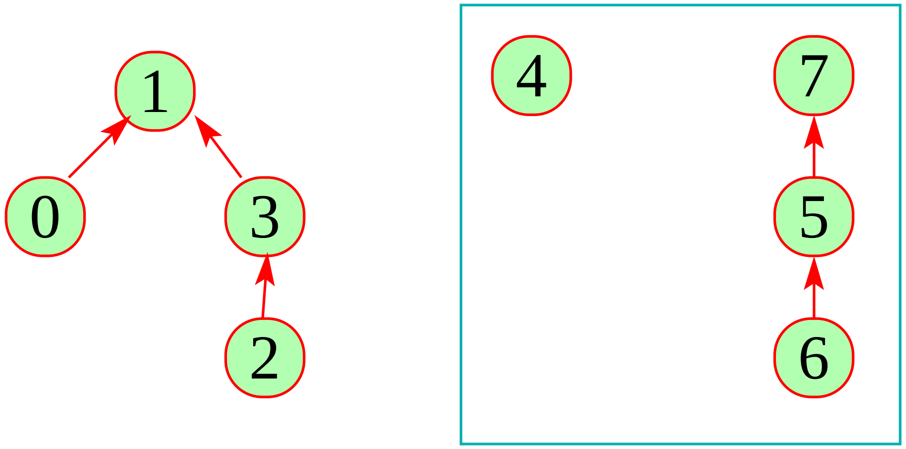
Their roots are their representatives ...



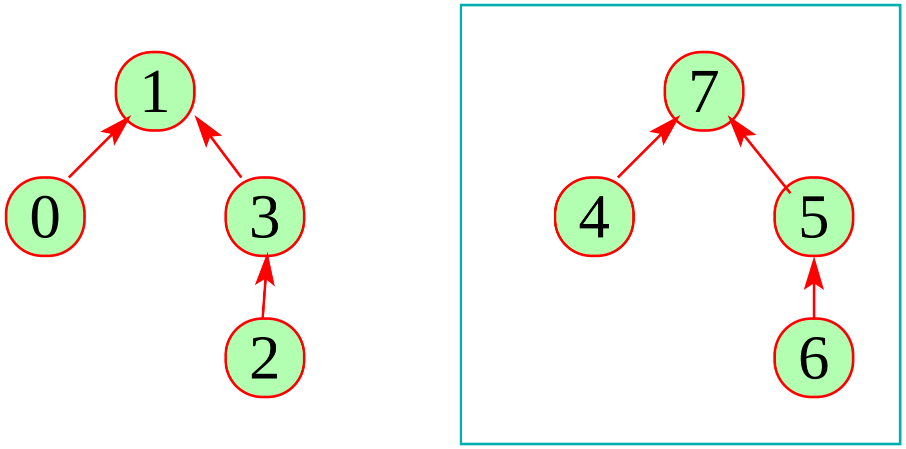
0	1	2	3	4	5	6	7
1	1	3	1	4	7	5	7

→ find (π, u) follows the father references :-)

→ union (π, u_1, u_2) re-directs the father reference of one u_i ...



0	1	2	3	4	5	6	7
1	1	3	1	4	7	5	7



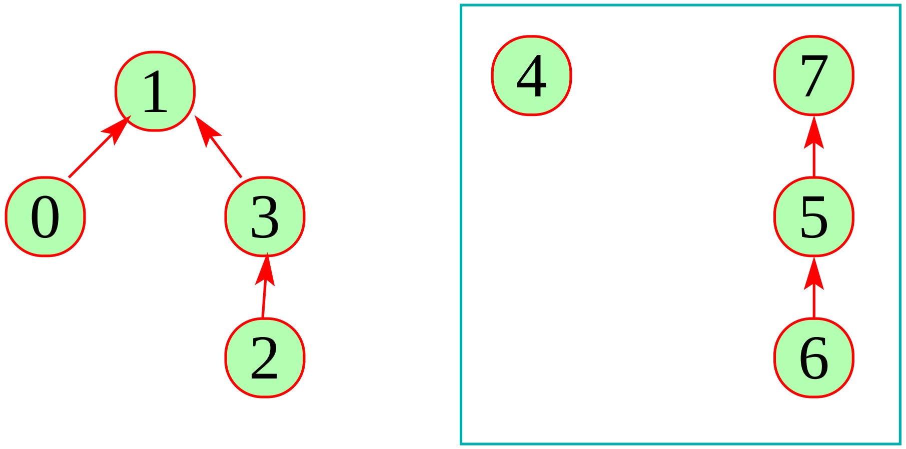
0	1	2	3	4	5	6	7
1	1	3	1	7	7	5	7

The Costs:

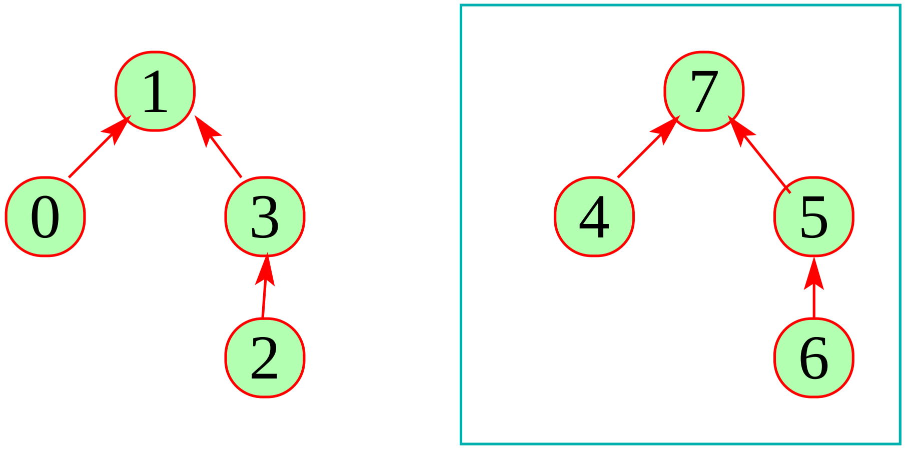
union	:	$\mathcal{O}(1)$	:-)
find	:	$\mathcal{O}(depth(\pi))$	:-(

Strategy to Avoid Deep Trees:

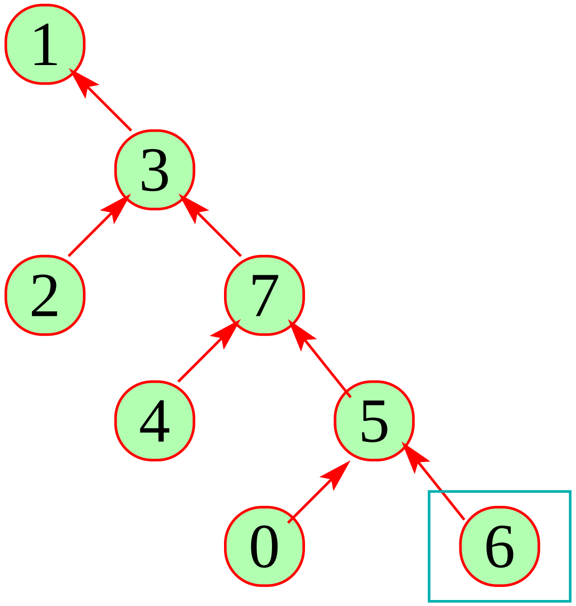
- Put the smaller tree below the bigger !
- Use find to compress paths ...



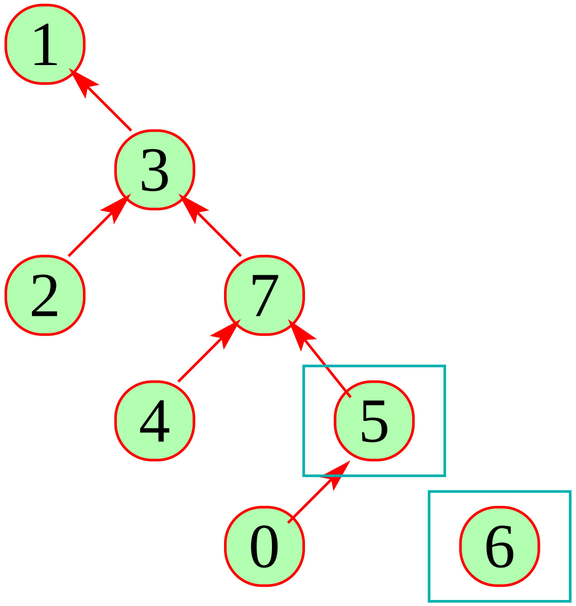
0	1	2	3	4	5	6	7
1	1	3	1	4	7	5	7



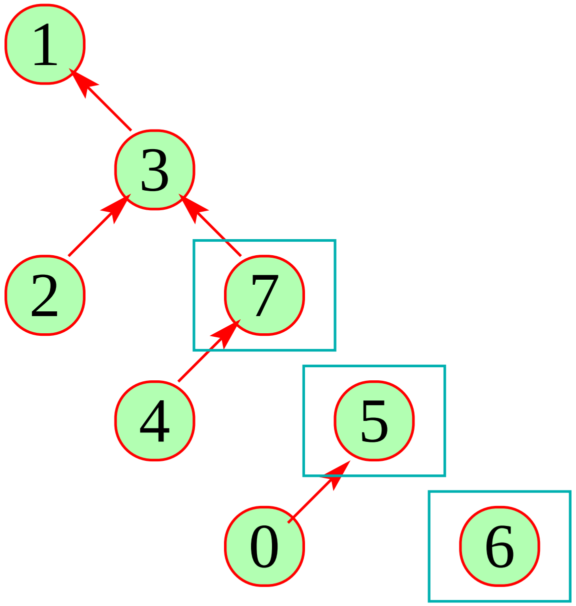
0	1	2	3	4	5	6	7
1	1	3	1	7	7	5	7



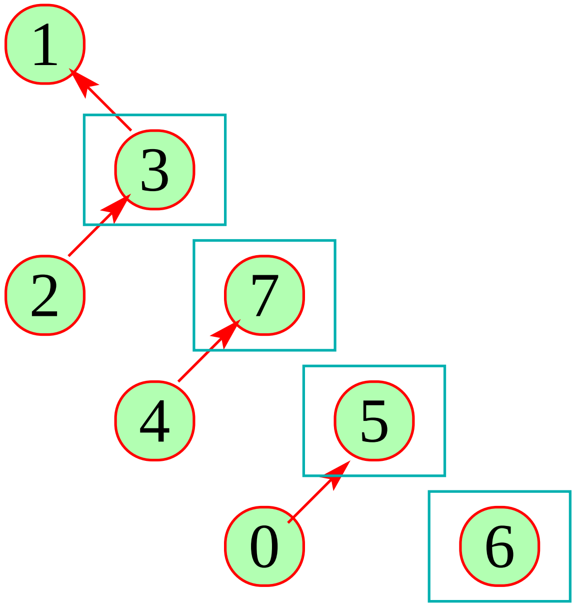
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



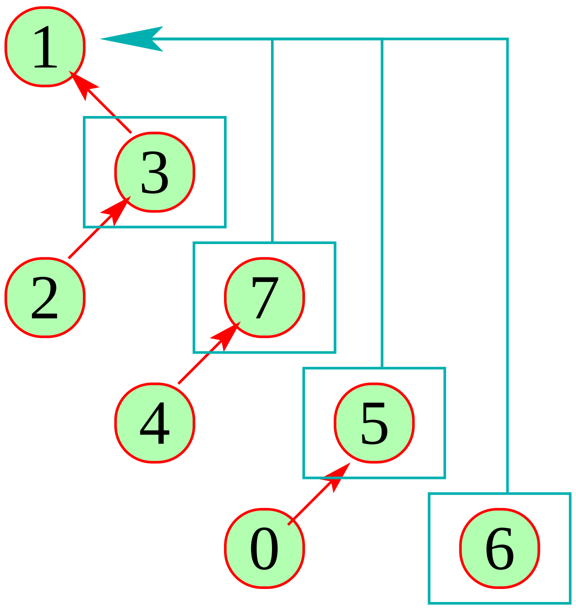
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



0	1	2	3	4	5	6	7
5	1	3	1	1	7	1	1

Robert Endre Tarjan, Princeton

Note:

- By this data-structure, n union- und m find operations require time $\mathcal{O}(n + m \cdot \alpha(n, n))$

  // α the inverse Ackermann-function :-)
- For our application, we only must modify union such that roots are from *Vars* whenever possible.
- This modification does not increase the asymptotic run-time. :-)

Summary:

The analysis is extremely fast — but may not find very much.

Background 3: Fixpoint Algorithms

Consider: $x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n$

Observation:

RR-Iteration is inefficient:

→ We require a complete round in order to detect termination :-(

→ If in some round, the value of just one unknown is changed, then we still re-compute all :-(

→ The practical run-time depends on the ordering on the variables :-(

Idea: Worklist Iteration

If an unknown x_i changes its value, we re-compute all unknowns which depend on x_i . Technically, we require:

- → the lists $Dep\ f_i$ of unknowns which are accessed during evaluation of f_i. From that, we compute the lists:

  $$I[x_i] = \{x_j \mid x_i \in Dep\ f_j\}$$

  i.e., a list of all x_j which depend on the value of x_i ;
- → the values $D[x_i]$ of the x_i where initially $D[x_i] = \bot$;
- → a list W of all unknowns whose value must be recomputed ...

The Algorithm:

```
$W = [x_1, \ldots, x_n];$
while $(W \neq [\,])$ {
        $x_i = \text{extract } W;$
        $t = f_i \text{ eval};$
        if $(t \not\sqsubseteq D[x_i])$ {
                $D[x_i] = D[x_i] \sqcup t;$
                $W = \text{append } I[x_i] \; W;$
        }
}
```

where :

$$eval \;\; x_j = D[x_j]$$

Example:

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

	I
x_1	$\{x_3\}$
x_2	$\emptyset$
x_3	$\{x_1, x_2\}$

Example:

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

	I
x_1	$\{x_3\}$
x_2	$\emptyset$
x_3	$\{x_1, x_2\}$

$D[x_1]$	$D[x_2]$	$D[x_3]$	W
$\emptyset$	$\emptyset$	$\emptyset$	$\boxed{x_1}, x_2, x_3$
$\{a\}$	$\emptyset$	$\emptyset$	$\boxed{x_2}, x_3$
$\{a\}$	$\emptyset$	$\emptyset$	$\boxed{x_3}$
$\{a\}$	$\emptyset$	$\{a, c\}$	$\boxed{x_1}, x_2$
$\{a, c\}$	$\emptyset$	$\{a, c\}$	$\boxed{x_3}, x_2$
$\{a, c\}$	$\emptyset$	$\{a, c\}$	$\boxed{x_2}$
$\{a, c\}$	$\{a\}$	$\{a, c\}$	$[\,]$

Theorem

Let $x_i \sqsupseteq f_i\,(x_1, \ldots, x_n)\;,\quad i = 1, \ldots, n$ denote a constraint system over the complete lattice $\mathbb{D}$ of hight $h > 0$.

(1) The algorithm terminates after at most $h \cdot N$ evaluations of right-hand sides where

$$N = \sum_{i=1}^{n} (1 + \#\,(\mathit{Dep}\; f_i)) \qquad // \quad \text{size of the system} \quad \text{:-)}$$

(2) The algorithm returns a solution.
If all f_i are monotonic, it returns the least one.

Proof:

Ad (1):

Every unknown x_i may change its value at most h times :-)

Each time, the list $I[x_i]$ is added to W .

Thus, the total number of evaluations is:

$$\begin{array}{ll} \leq & n + \sum_{i=1}^{n} (h \cdot \#\,(I[x_i])) \\ = & n + h \cdot \sum_{i=1}^{n} \#\,(I[x_i]) \\ = & n + h \cdot \sum_{i=1}^{n} \#\,(Dep\ f_i) \\ \leq & h \cdot \sum_{i=1}^{n} (1 + \#\,(Dep\ f_i)) \\ = & h \cdot N \end{array}$$

Ad (2):

We only consider the assertion for monotonic f_i .

Let D_0 denote the least solution. We show:

- $D_0[x_i] \sqsupseteq D[x_i]$ (all the time)
- $D[x_i] \not\sqsupseteq f_i\,\mathsf{eval} \Longrightarrow x_i \in W$ (at exit of the loop body)
- On termination, the algo returns a solution :-))

Discussion:

- In the example, fewer evaluations of right-hand sides are required than for RR-iteration :-)
- The algo also works for non-monotonic f_i :-)
- For monotonic f_i, the algo can be simplified:

$$\boxed{D[x_i] = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{D[x_i] = \qquad\qquad t;}$$

- In presence of widening, we replace:

$$\boxed{D[x_i] = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{D[x_i] = D[x_i] \underline{\sqcup}\, t;}$$

- In presence of Narrowing, we replace:

$$\boxed{D[x_i] = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{D[x_i] = D[x_i] \overline{\sqcap}\, t;}$$

Warning:

- The algorithm relies on explicit dependencies among the unknowns.

  So far in our applications, these were obvious. This need not always be the case :-(

- We need some strategy for extract which determines the next unknown to be evaluated.

- It would be ingenious if we always evaluated first and then accessed the result ... :-)

$\Longrightarrow$ recursive evaluation ...

Idea:

→ If during evaluation of f_i , an unknown x_j is accessed, x_j is first solved recursively. Then x_i is added to $I[x_j]$:-)

$$\begin{array}{lll} \mathsf{eval}\ x_i\ x_j & = & \mathsf{solve}\, x_j; \\ & & I[x_j] = I[x_j] \cup \{x_i\}; \\ & & D[x_j]; \end{array}$$

→ In order to prevent recursion to descend infinitely, a set *Stable* of unknown is maintained for which solve just looks up their values :-)

Initially, $\mathit{Stable} = \emptyset$ ...

The Function solve :

```
$\text{solve}\ x_i$ = if $(x_i \notin Stable)$ {
        $Stable = Stable \cup \{x_i\};$
        $t = f_i\,(\text{eval}\ \ x_i);$
        if $(t \not\sqsubseteq D[x_i])$ {
            $W = I[x_i]; \quad I[x_i] = \emptyset;$
            $D[x_i] = D[x_i] \sqcup t;$
            $Stable = Stable \backslash W;$
            app solve $W$;
        }
    }
```

Helmut Seidl, TU München ;-)

Example:

Consider our standard example:

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

A trace of the fixpoint algorithm then looks as follows:

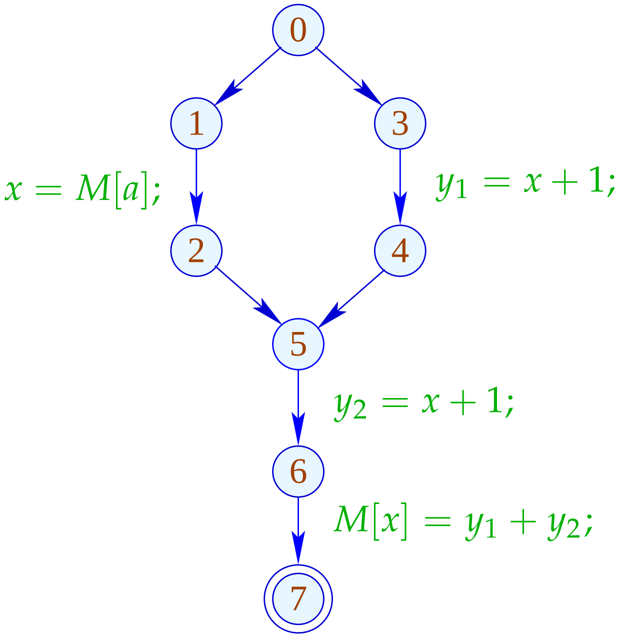
solve x_2	eval x_2 x_3	solve x_3	eval x_3 x_1	solve x_1	eval x_1 x_3	solve x_3
						stable!
					$I[x_3] = \{x_1\}$	
					$\Rightarrow \quad \emptyset$	
				$\boxed{D[x_1] = \{a\}}$		
			$I[x_1] = \{x_3\}$			
			$\Rightarrow \quad \{a\}$			
		$\boxed{D[x_3] = \{a, c\}}$				
		$I[x_3] = \emptyset$				
		solve x_1	eval x_1 x_3	solve x_3		
				stable!		
			$I[x_3] = \{x_1\}$			
			$\Rightarrow \quad \{a, c\}$			
		$\boxed{D[x_1] = \{a, c\}}$				
		$I[x_1] = \emptyset$				
		solve x_3	eval x_3 x_1	solve x_1		
				stable!		
			$I[x_1] = \{x_3\}$			
			$\Rightarrow \quad \{a, c\}$			
		$\boxed{\text{ok}}$				
	$I[x_3] = \{x_1, x_2\}$					
	$\Rightarrow \quad \{a, c\}$					
$\boxed{D[x_2] = \{a\}}$						

- → Evaluation starts with an interesting unknown x_i (e.g., the value at *stop*)
- → Then automatically all unknowns are evaluated which influence x_i :-)
- → The number of evaluations is often smaller than during worklist iteration ;-)
- → The algorithm is more complex but does not rely on pre-computation of variable dependencies :-))
- → It also works if variable dependencies during iteration change !!!

$\Longrightarrow$ interprocedural analysis

1.7 Eliminating Partial Redundancies

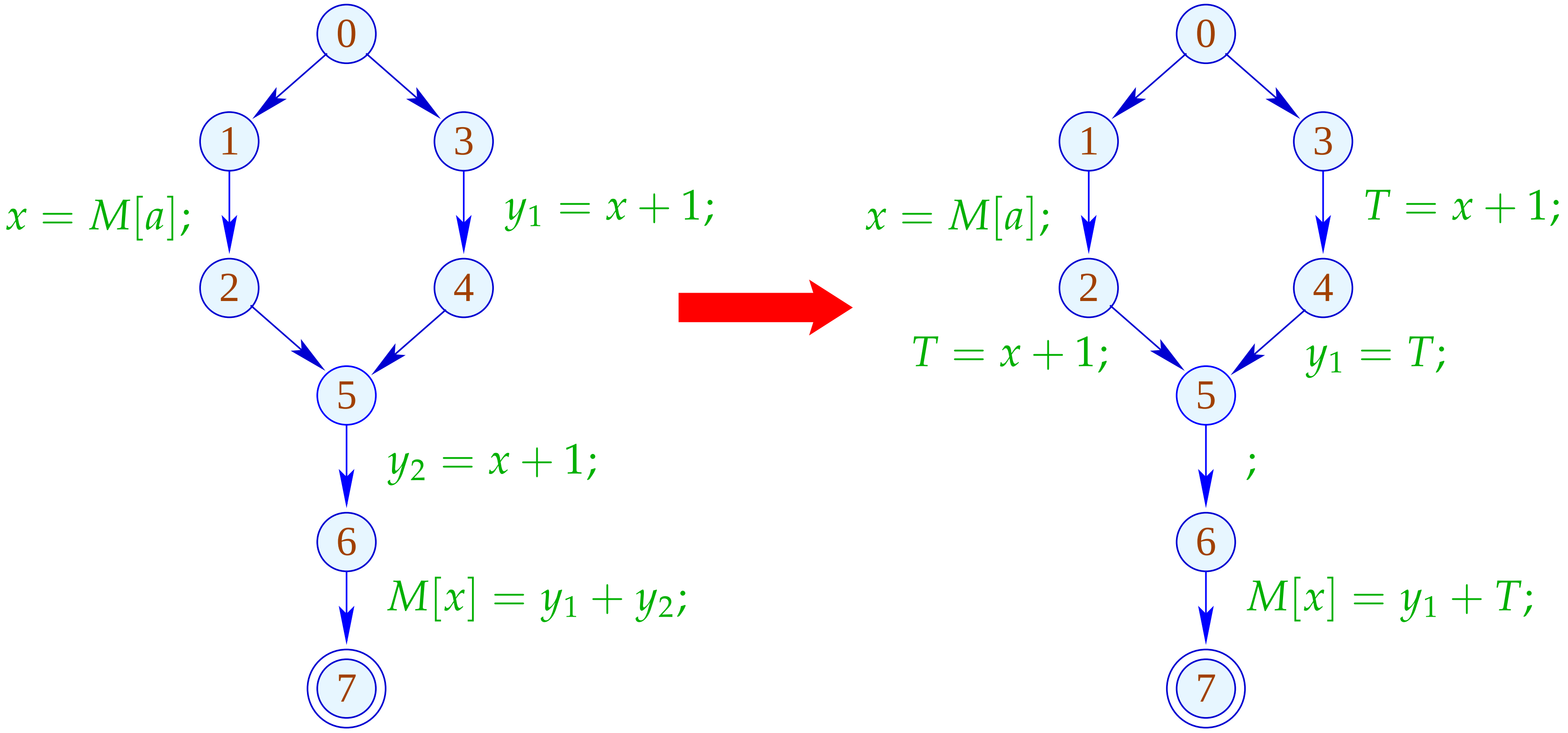
Example:



// $x + 1$ is evaluated on every path ...

// on one path, however, even twice :-(

Goal:


0
1
3
$x = M[a];$
$y_1 = x + 1;$
2
4
5
$y_2 = x + 1;$
6
$M[x] = y_1 + y_2;$
7
0
1
3
$x = M[a];$
$T = x + 1;$
2
4
$T = x + 1;$
$y_1 = T;$
5
;
6
$M[x] = y_1 + T;$
7

Idea:

(1) Insert assignments $T_e = e;$ such that e is available at all points where the value of e is required.

(2) Thereby spare program points where e either is already available or will definitely be computed in future.

Expressions with the latter property are called very busy.

(3) Replace the original evaluations of e by accesses to the variable T_e.

$\Longrightarrow$ we require a novel analysis :-))

An expression e is called busy along a path π , if the expression e is evaluated before any of the variables $x \in \mathit{Vars}(e)$ is overwritten.

$//$ backward analysis!

e is called very busy at u , if e is busy along every path $\pi : u \longrightarrow^{*} \mathit{stop}$.

An expression e is called busy along a path π , if the expression e is evaluated before any of the variables $x \in \mathit{Vars}(e)$ is overwriten.

$//$ backward analysis!

e is called very busy at u , if e is busy along every path $\pi : u \to^* \mathit{stop}$.

Accordingly, we require:

$$\mathcal{B}[u] \;=\; \bigcap\{[\![\pi]\!]^\sharp\,\emptyset \mid \pi : u \to^* \mathit{stop}\}$$

where for $\pi = k_1 \ldots k_m$:

$$[\![\pi]\!]^\sharp \;=\; [\![k_1]\!]^\sharp \circ \ldots \circ [\![k_m]\!]^\sharp$$

Our complete lattice is given by:

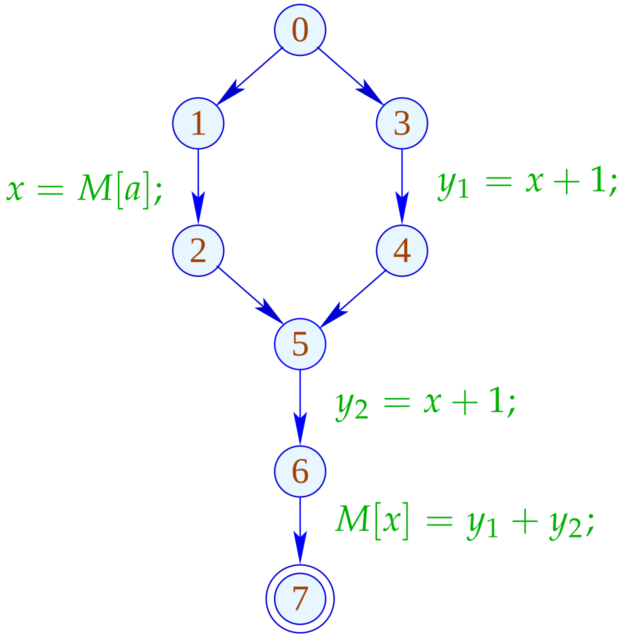
$$\mathbb{B} = 2^{Expr \setminus Vars} \qquad \text{with} \qquad \sqsubseteq \;=\; \supseteq$$

The effect $[\![k]\!]^\sharp$ of an edge $k = (u, lab, v)$ only depends on lab , i.e., $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ where:

$$\begin{array}{lll}
[\![;]\!]^\sharp\, B & = & B \\
[\![Pos(e)]\!]^\sharp\, B & = & [\![Neg(e)]\!]^\sharp\, B \qquad = \quad B \cup \{e\} \\
[\![x = e;]\!]^\sharp\, B & = & (B \setminus Expr_x) \cup \{e\} \\
[\![x = M[e];]\!]^\sharp\, B & = & (B \setminus Expr_x) \cup \{e\} \\
[\![M[e_1] = e_2;]\!]^\sharp\, B & = & B \cup \{e_1, e_2\}
\end{array}$$

These effects are all distributive. Thus, the least solution of the constraint system yields precisely the MOP — given that *stop* is reachable from every program point :-)

Example:



7	$\emptyset$
6	$\emptyset$
5	$\{x+1\}$
4	$\{x+1\}$
3	$\{x+1\}$
2	$\{x+1\}$
1	$\emptyset$
0	$\emptyset$

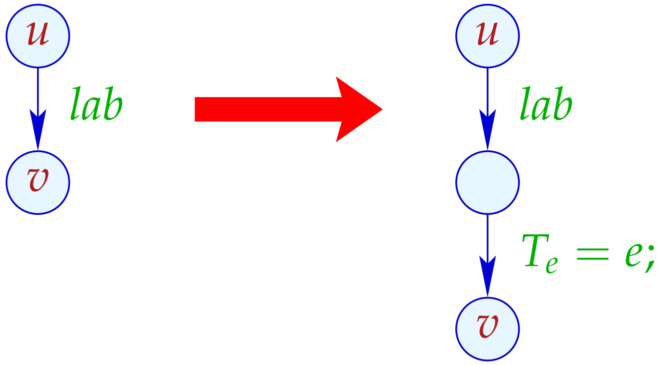
A point u is called safe for e, if $e \in \mathcal{A}[u] \cup \mathcal{B}[u]$, i.e., e is either available or very busy.

Idea:

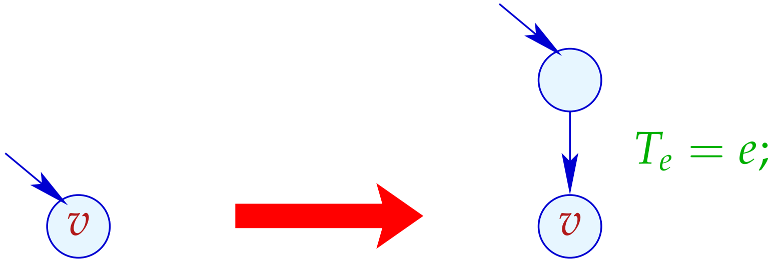
- We insert computations of e such that e becomes available at all safe program points :-)
- We insert $T_e = e;$ after every edge (u, lab, v) with

$$e \in \mathcal{B}[v] \backslash [\![lab]\!]^\sharp_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u])$$

Transformation 5.1:

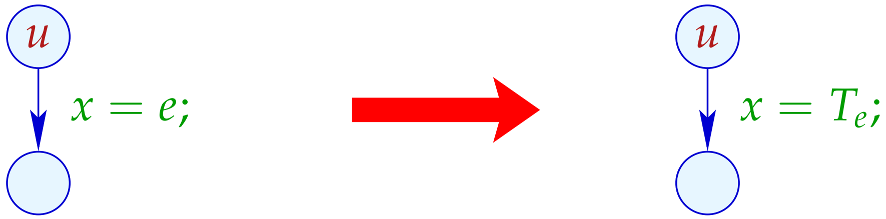


$(e \in \mathcal{B}[v] \backslash [\![lab]\!]^\sharp_{\mathcal{A}} (\mathcal{A}[u] \cup \mathcal{B}[u]))$



$(e \in \mathcal{B}[v])$

Transformation 5.2:



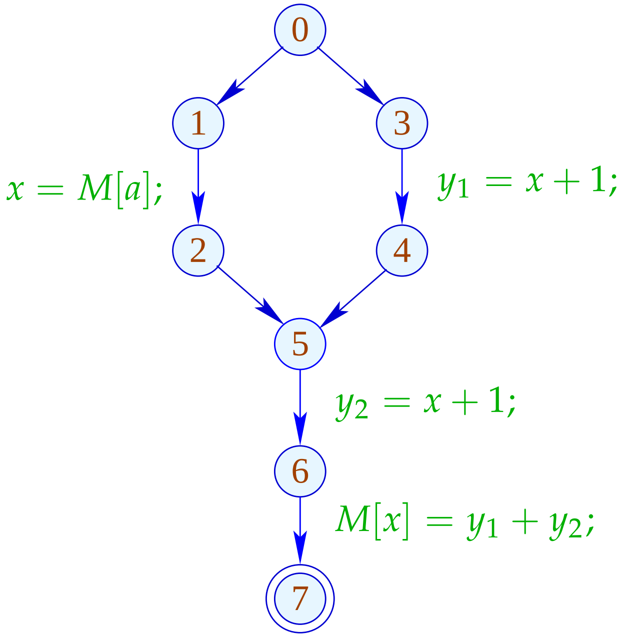
// analogously for the other uses of e

// at old edges of the program.

Bernhard Steffen, Dortmund

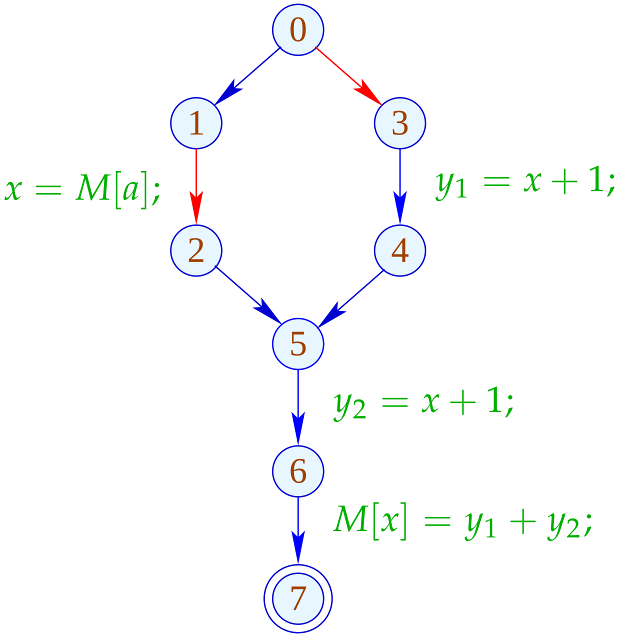
Jens Knoop, Wien

In the Example:



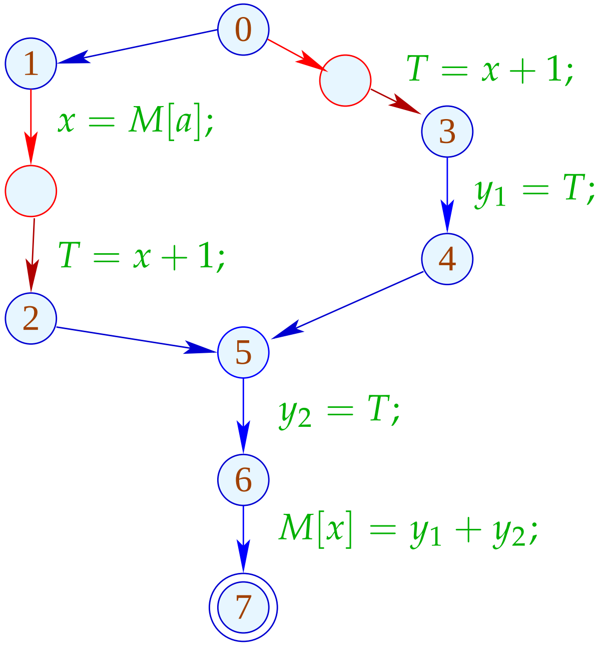
	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x+1\}$
3	$\emptyset$	$\{x+1\}$
4	$\{x+1\}$	$\{x+1\}$
5	$\emptyset$	$\{x+1\}$
6	$\{x+1\}$	$\emptyset$
7	$\{x+1\}$	$\emptyset$

In the Example:



	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x+1\}$
3	$\emptyset$	$\{x+1\}$
4	$\{x+1\}$	$\{x+1\}$
5	$\emptyset$	$\{x+1\}$
6	$\{x+1\}$	$\emptyset$
7	$\{x+1\}$	$\emptyset$

Im Example:



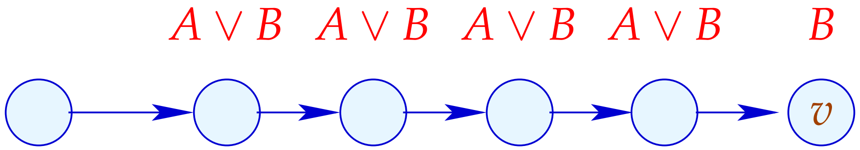
	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x+1\}$
3	$\emptyset$	$\{x+1\}$
4	$\{x+1\}$	$\{x+1\}$
5	$\emptyset$	$\{x+1\}$
6	$\{x+1\}$	$\emptyset$
7	$\{x+1\}$	$\emptyset$

Correctness:

Let π denote a path reaching v after which a computation of an edge with e follows.

Then there is a maximal suffix of π such that for every edge $k = (u, lab, u')$ in the suffix:

$$e \in [\![lab]\!]^{\sharp}_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u])$$



Correctness:

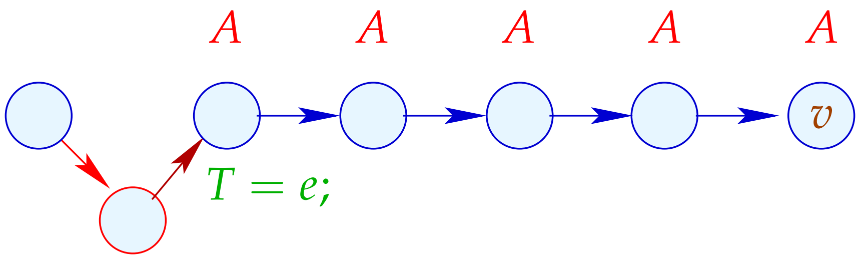
Let π denote a path reaching v after which a computation of an edge with e follows.

Then there is a maximal suffix of π such that for every edge $k = (u, lab, u')$ in the suffix:

$$e \in [\![lab]\!]^\sharp_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u])$$

In particular, no variable in e receives a new value :-)

Then $T_e = e;$ is inserted before the suffix :-))



We conclude:

- Whenever the value of e is required, e is available :-)

  $\Longrightarrow$ correctness of the transformation

- Every $T = e;$ which is inserted into a path corresponds to an e which is replaced with T :-))

  $\Longrightarrow$ non-degradation of the efficiency

1.8 Application: Loop-invariant Code

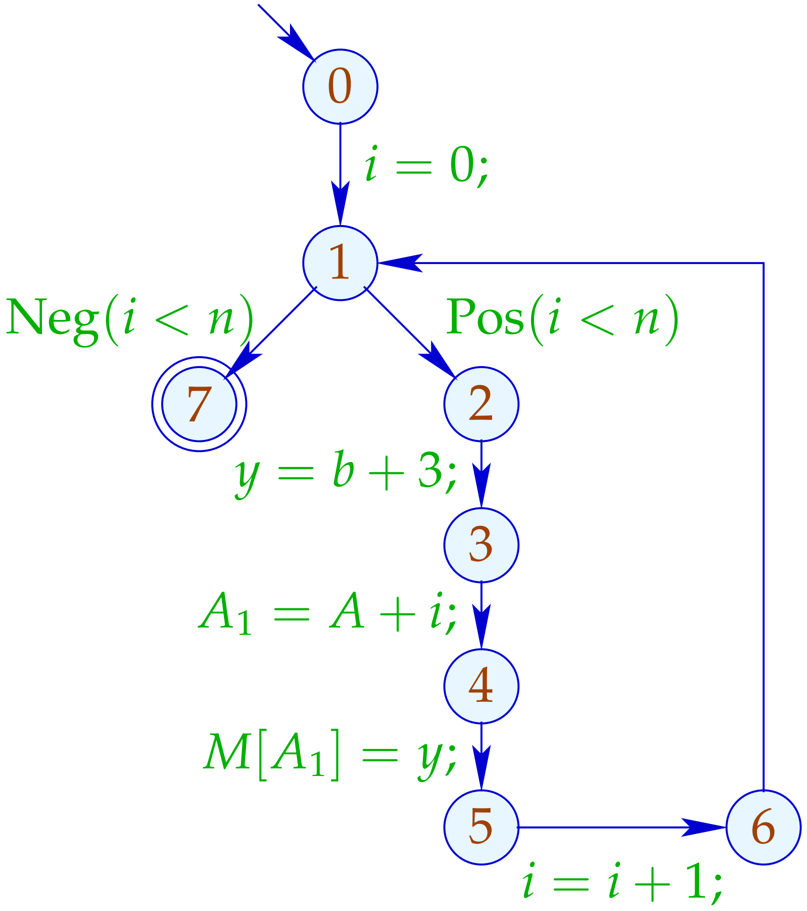
Example:

$$\text{for } (i = 0; i < n; i{+}{+})$$
$$a[i] = b + 3;$$

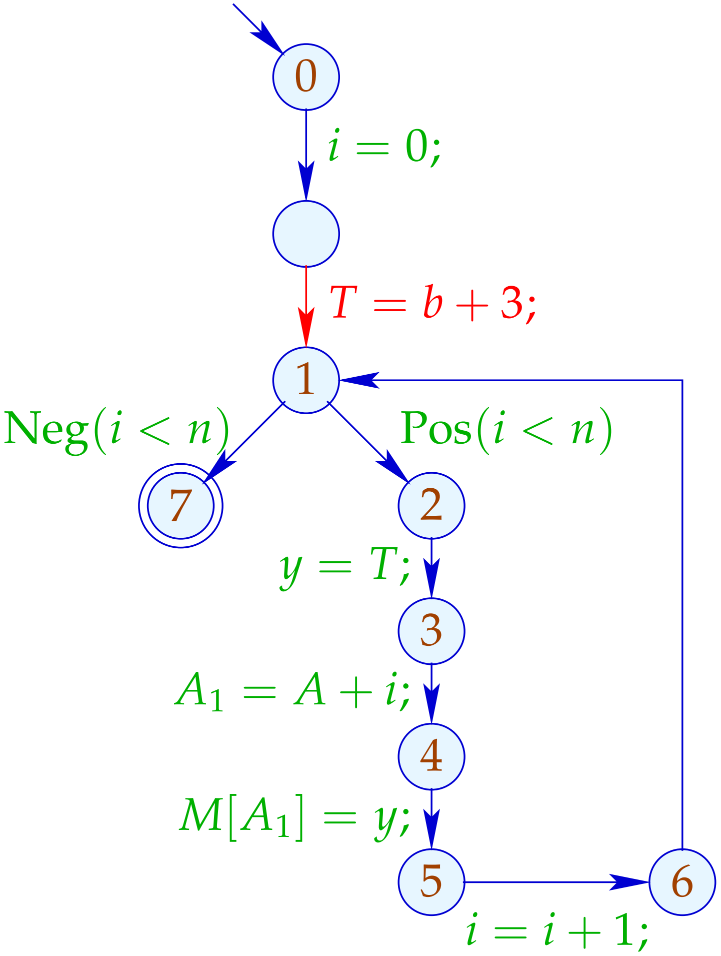
$//$ The expression $b + 3$ is recomputed in every iteration :-(

$//$ This should be avoided :-)

The Control-flow Graph:

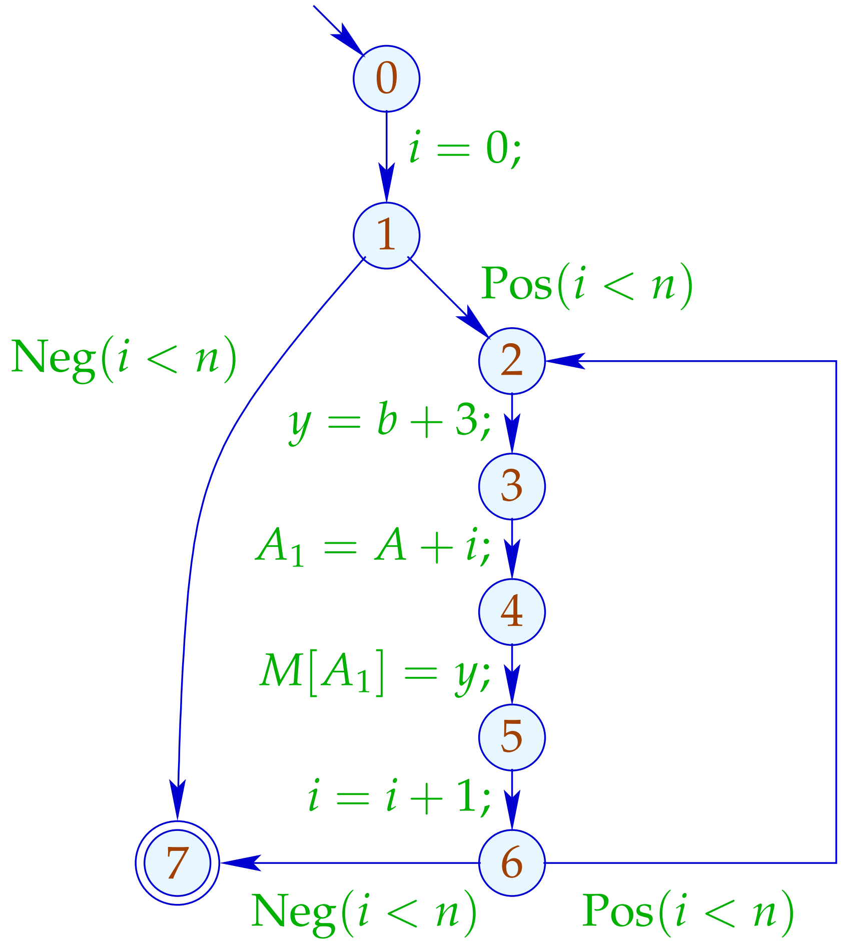


Warning: $T = b + 3;$ may not be placed before the loop :

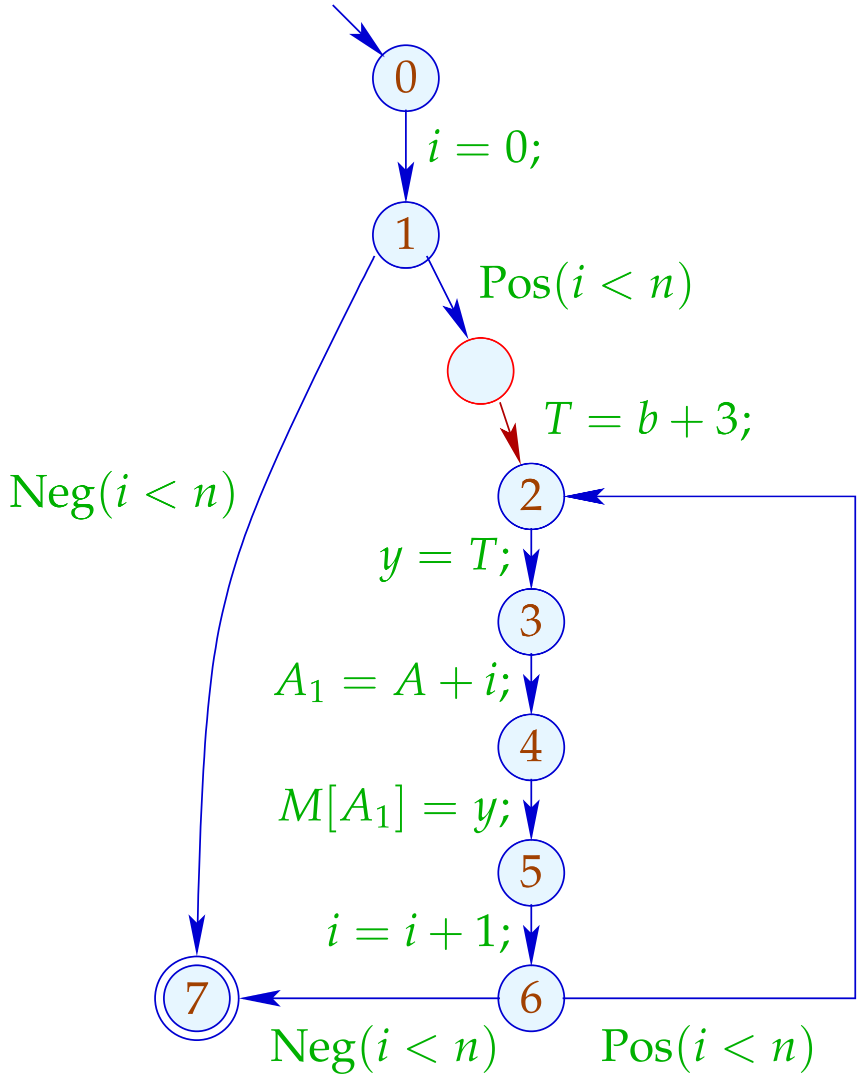


$\Longrightarrow$ There is no decent place for $T = b + 3;$:-(

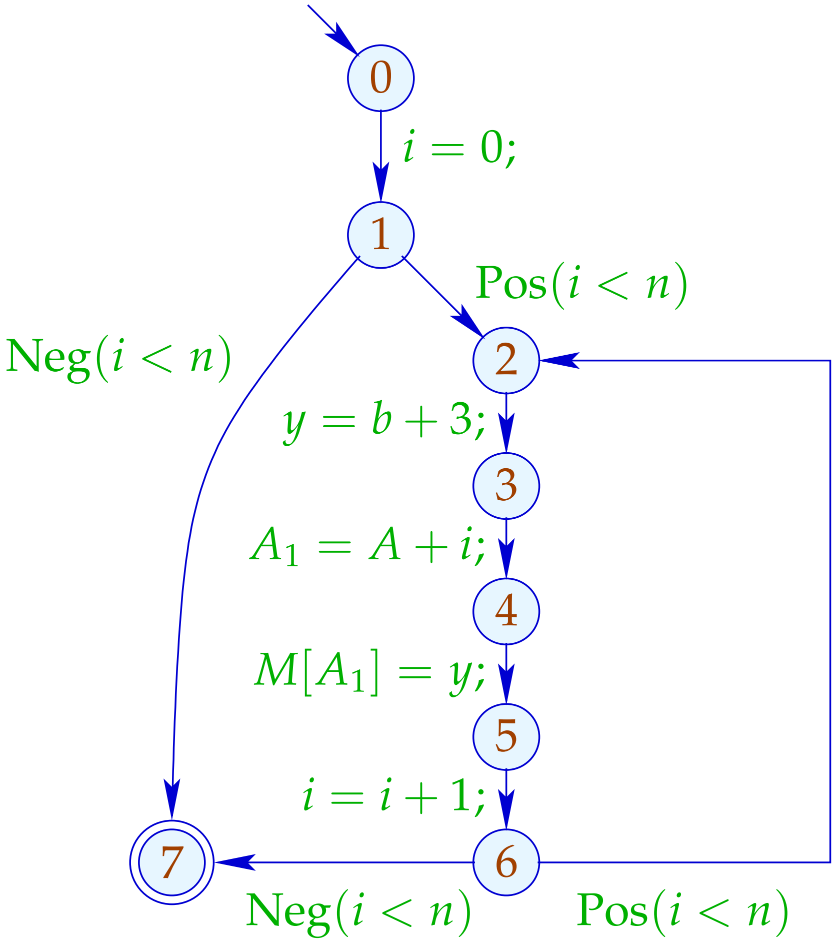
Idea: Transform into a do-while-loop ...



... now there is a place for $T = e;$:-)

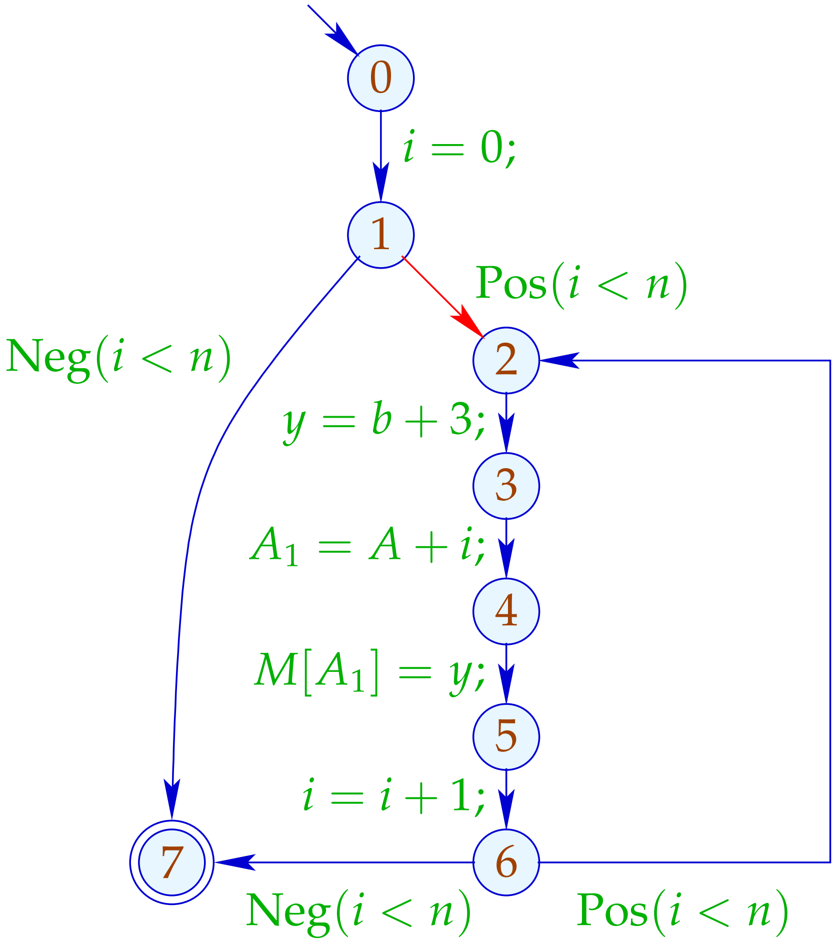


Application of T5 (PRE) :



	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{b + 3\}$
3	$\{b + 3\}$	$\emptyset$
4	$\{b + 3\}$	$\emptyset$
5	$\{b + 3\}$	$\emptyset$
6	$\{b + 3\}$	$\emptyset$
6	$\emptyset$	$\emptyset$
7	$\emptyset$	$\emptyset$

Application of T5 (PRE) :



	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{b + 3\}$
3	$\{b + 3\}$	$\emptyset$
4	$\{b + 3\}$	$\emptyset$
5	$\{b + 3\}$	$\emptyset$
6	$\{b + 3\}$	$\emptyset$
6	$\emptyset$	$\emptyset$
7	$\emptyset$	$\emptyset$

Conclusion:

- Elimination of partial redundancies may move loop-invariant code out of the loop :-))
- This only works properly for do-while-loops :-(
- To optimize other loops, we transform them into do-while-loops before-hand:

```
while (b) stmt    ==>    if (b)
                               do stmt
                               while (b);
```

$\Longrightarrow$ Loop Rotation

Problem:

If we do not have the source program at hand, we must re-construct potential loop headers ;-)

$\Longrightarrow$ Pre-dominators

u pre-dominates v, if every path $\pi : \mathit{start} \rightarrow^* v$ contains u.
We write: $u \Rightarrow v$.

"$\Rightarrow$" is reflexive, transitive and anti-symmetric :-)

Computation:

We collect the nodes along paths by means of the analysis:

$$\mathbb{P} = 2^{Nodes} \quad , \qquad \sqsubseteq \; = \; \supseteq$$

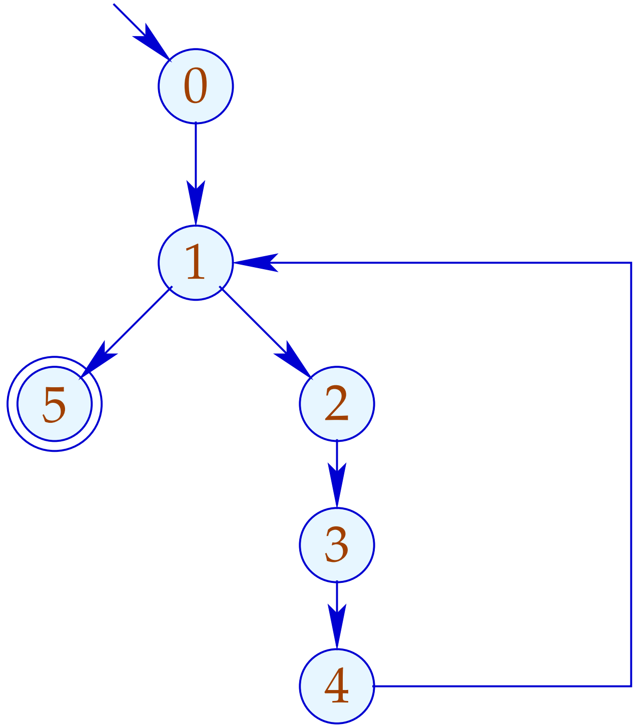
$$[\![(\_,\_,v)]\!]^\sharp\, P \quad = \quad P \cup \{v\}$$

Then the set $\mathcal{P}[v]$ of pre-dominators is given by:

$$\mathcal{P}[v] = \bigcap\{[\![\pi]\!]^\sharp\, \{start\} \mid \pi : start \rightarrow^* v\}$$

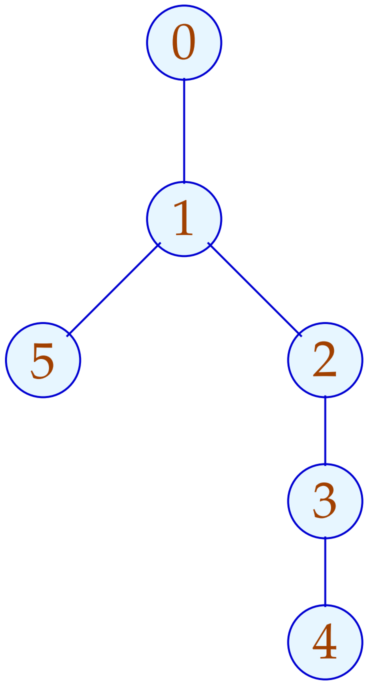
Since $[\![k]\!]^\sharp$ are distributive, the $\mathcal{P}[v]$ can computed by means of fixpoint iteration :-)

Example:



	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

The partial ordering “$\Rightarrow$” in the example:



	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

Apparently, the result is a tree :-)

In fact, we have:

Theorem:

Every node v has at most one immediate pre-dominator.

Proof:

Assume:

there are $u_1 \neq u_2$ which immediately pre-dominate v.

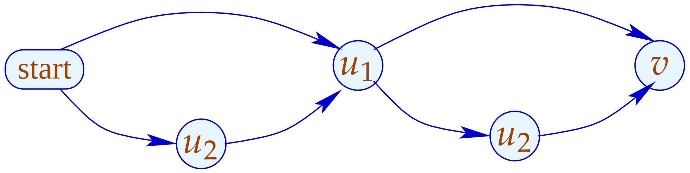
If $u_1 \Rightarrow u_2$ then u_1 not immediate.

Consequently, u_1, u_2 are incomparable :-)

Now for every $\pi : start \longrightarrow^* v$:

$$\pi = \pi_1 \; \pi_2 \qquad \text{with} \qquad \begin{array}{l} \pi_1 : start \longrightarrow^* u_1 \\ \pi_2 : u_1 \longrightarrow^* v \end{array}$$

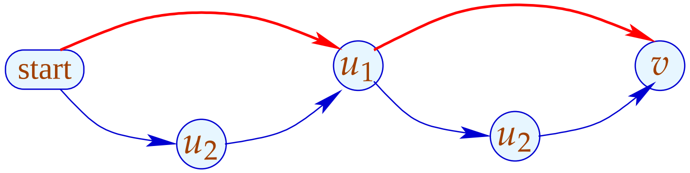
If, however, u_1, u_2 are incomparable, then there is path:
$start \longrightarrow^* v$ avoiding u_2 :



Now for every $\pi : start \longrightarrow^* v$:

$$\pi = \pi_1\ \pi_2 \qquad \text{with} \qquad \begin{array}{l} \pi_1 : start \longrightarrow^* u_1 \\ \pi_2 : u_1 \longrightarrow^* v \end{array}$$

If, however, u_1, u_2 are incomparable, then there is path:
$start \longrightarrow^* v$ avoiding u_2 :



Observation:

The loop head of a while-loop pre-dominates every node in the body.

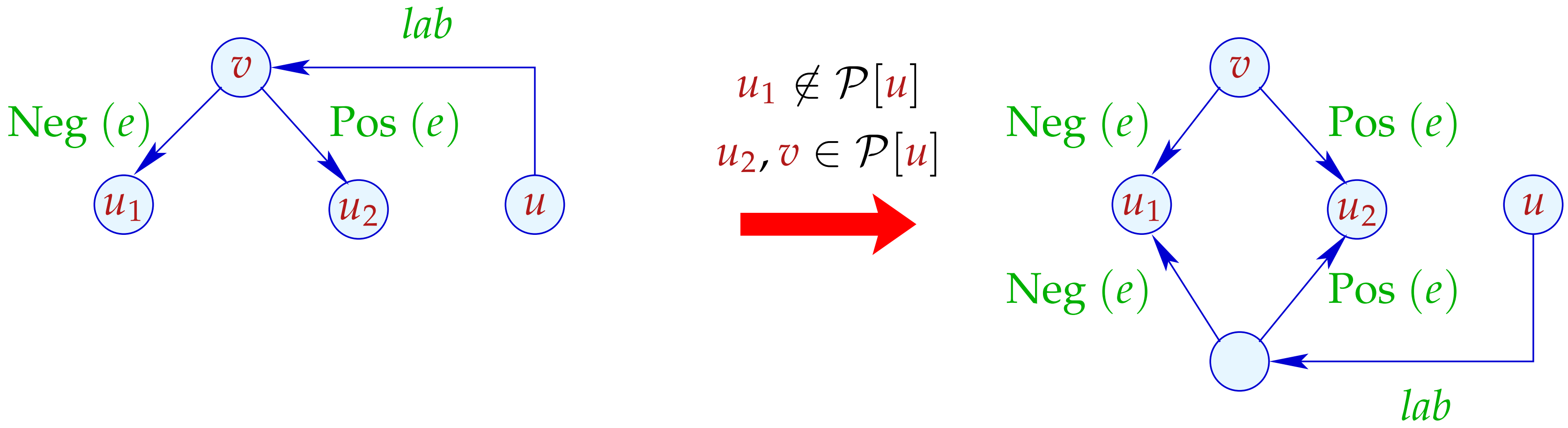
A back edge from the exit u to the loop head v can be identified through

$$v \in \mathcal{P}[u]$$

:-)

Accordingly, we define:

Transformation 6:

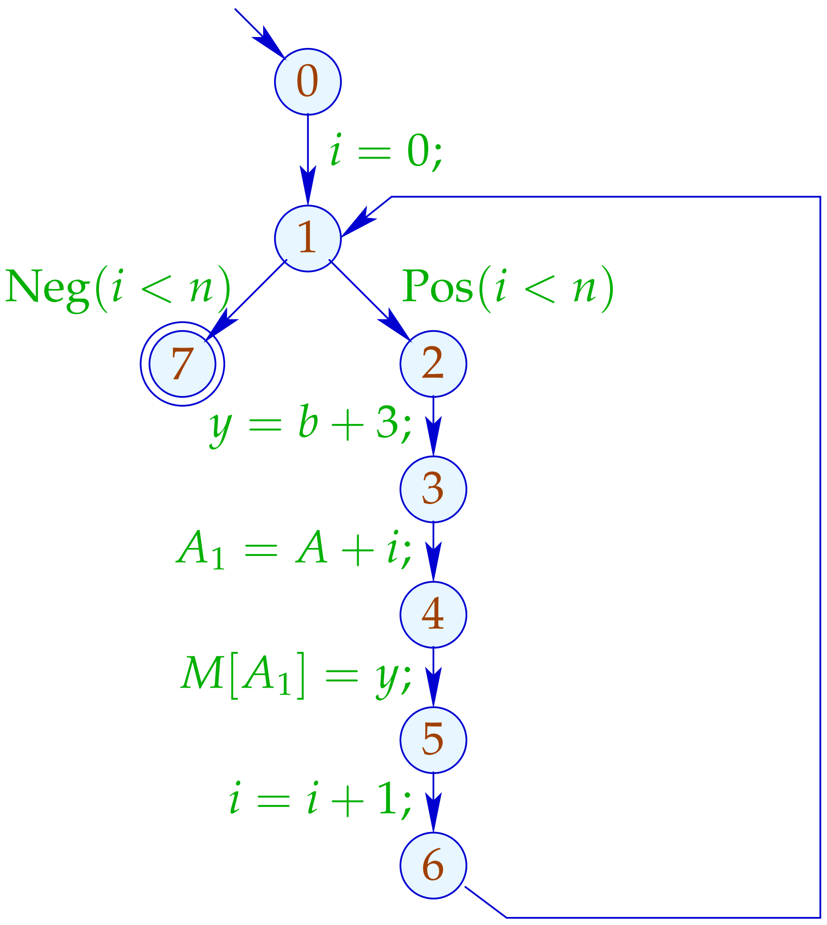


$$u_1 \notin \mathcal{P}[u]$$

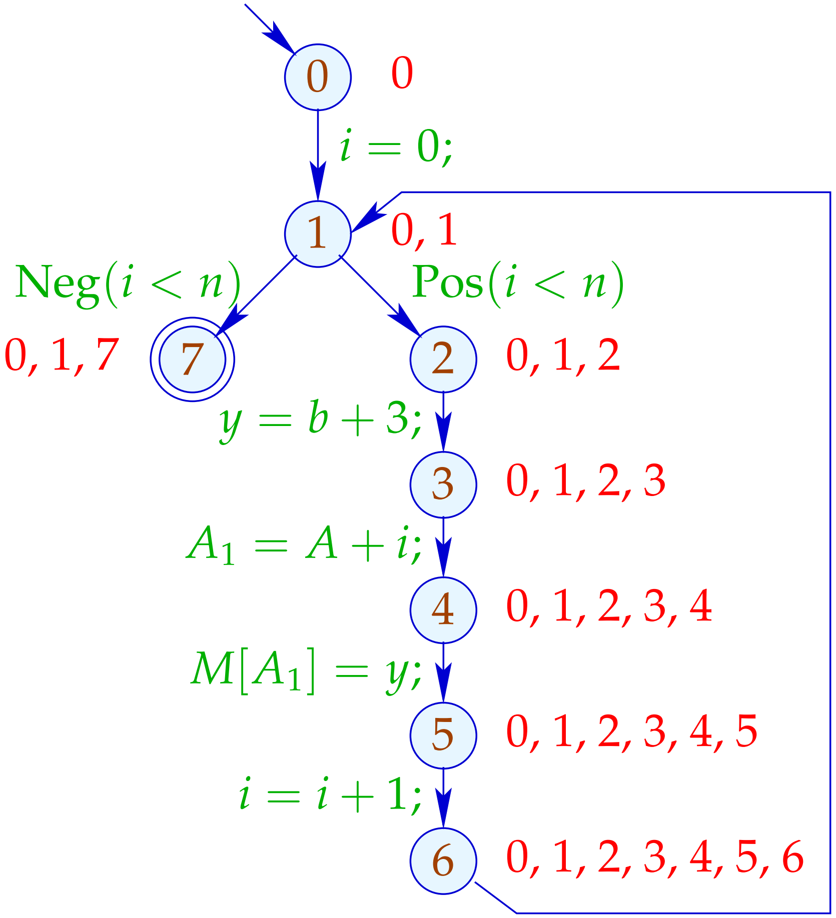
$$u_2, v \in \mathcal{P}[u]$$

We duplicate the entry check to all back edges :-)

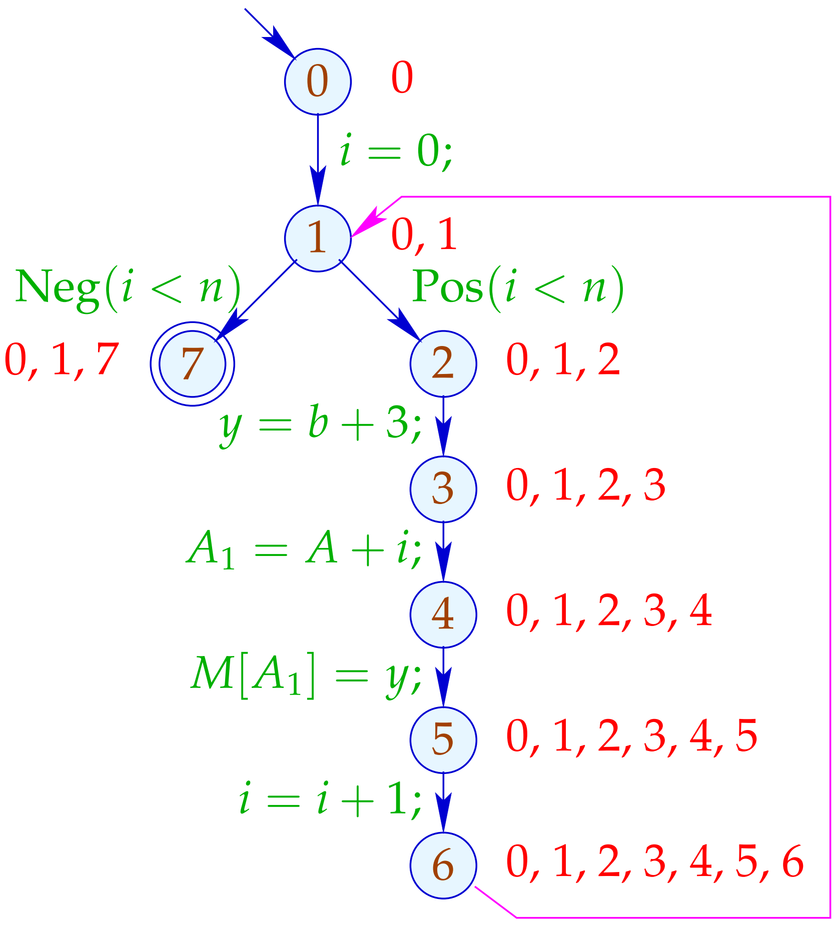
... in the Example:



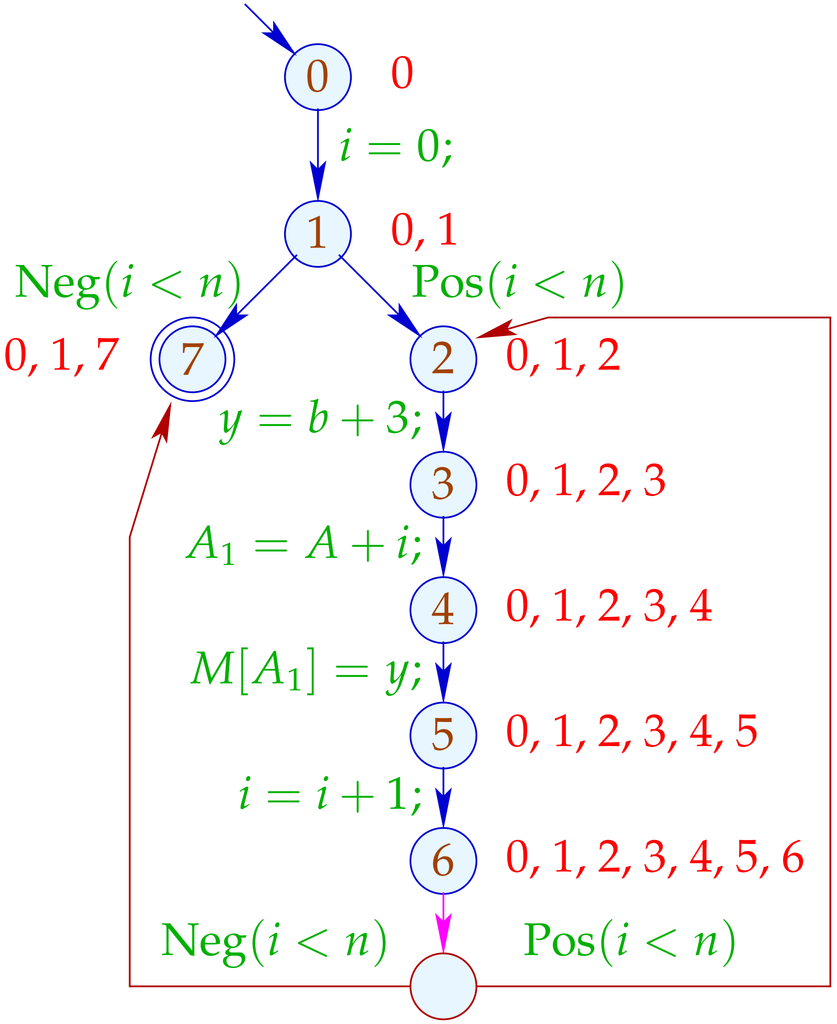
... in the Example:



... in the Example:

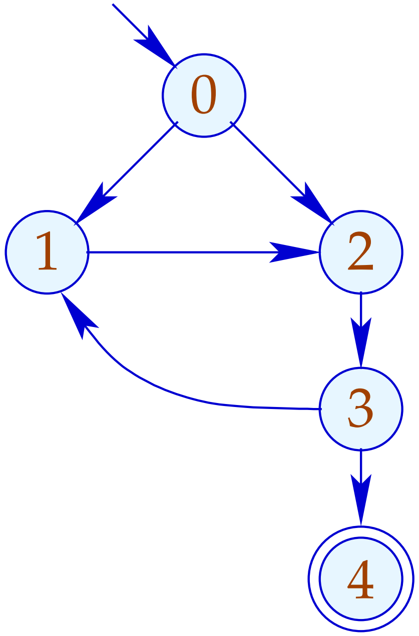


... in the Example:

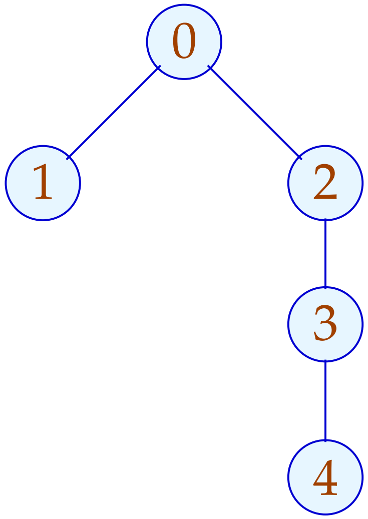


Warning:

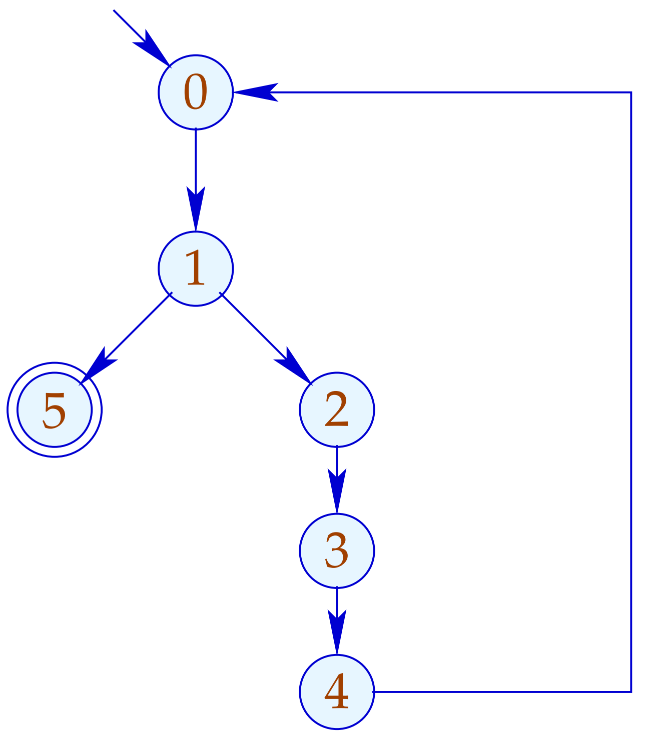
There are unusual loops which cannot be rotated:

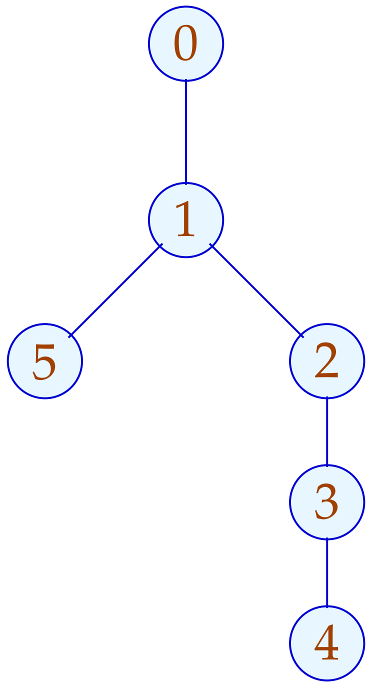


Pre-dominators:



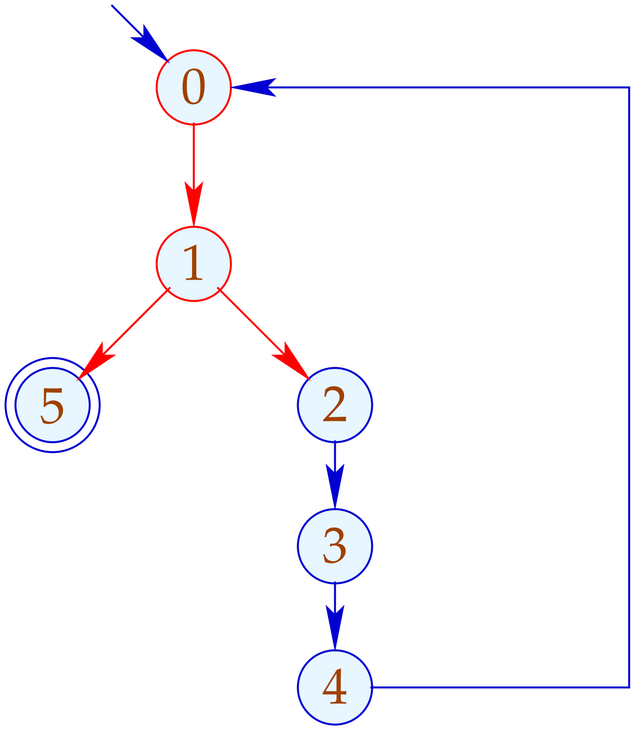
... but also common ones which cannot be rotated:

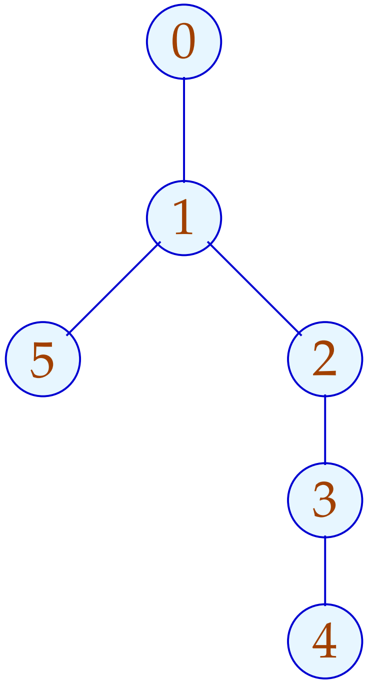




Here, the complete block between back edge and conditional jump should be duplicated :-(

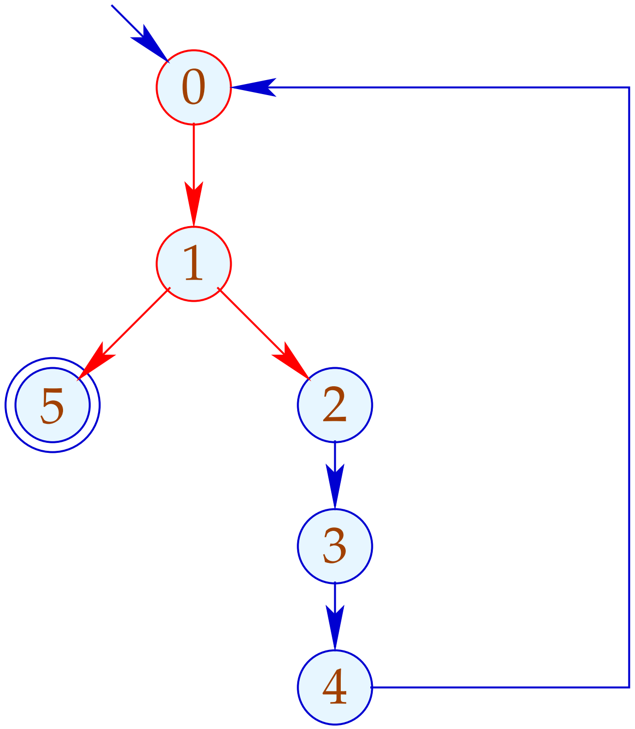
... but also common ones which cannot be rotated:

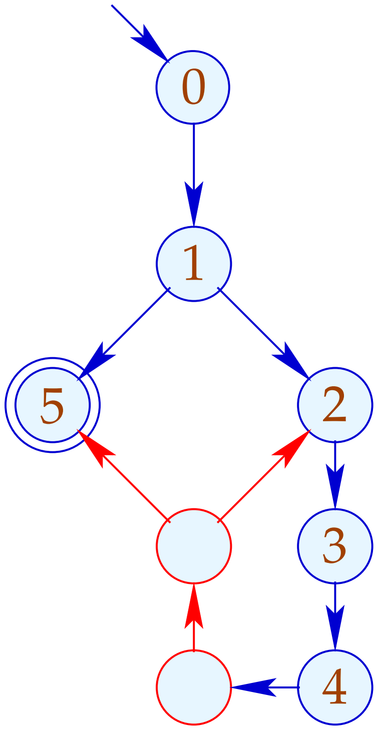




Here, the complete block between back edge and conditional jump should be duplicated :-(

... but also common ones which cannot be rotated:

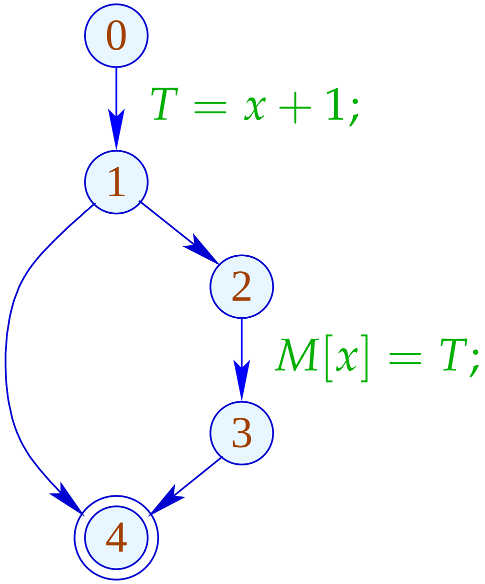




Here, the complete block between back edge and conditional jump should be duplicated :-(

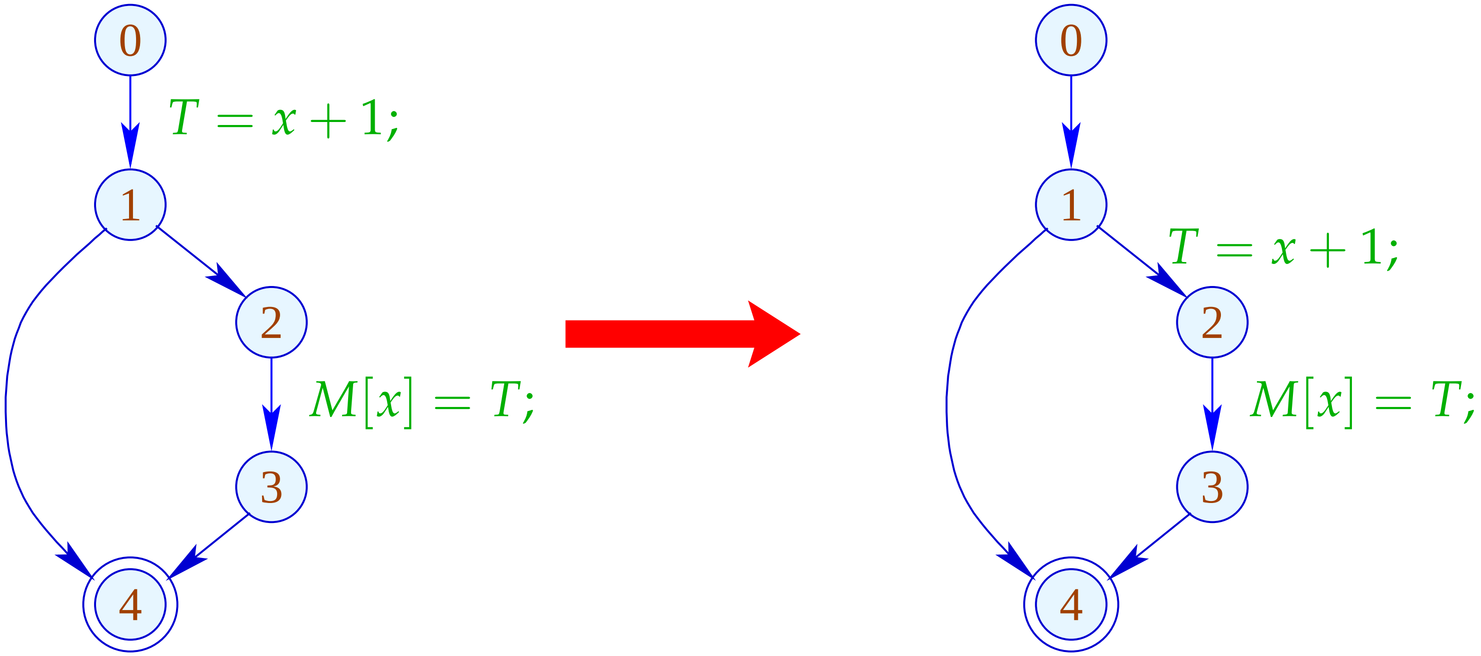
1.9 Eliminating Partially Dead Code

Example:



$x + 1$ need only be computed along one path ;-(

Idea:



Problem:

- The definition $x = e;$ $(x \notin Vars_e)$ may only be moved to an edge where e is safe ;-)
- The definition must still be available for uses of x ;-)

$$\Longrightarrow$$

We define an analysis which maximally delays computations:

$$\begin{array}{lll} [\![;]\!]^\sharp\, D & = & \\ [\![x = e;]\!]^\sharp\, D & = & \begin{cases} D \backslash (Use_e \cup Def_x) \cup \{x = e;\} & \text{if} \quad x \notin Vars_e \\ D \backslash (Use_e \cup Def_x) & \text{if} \quad x \in Vars_e \end{cases} \end{array}$$

... where:

$$
\begin{aligned}
\mathit{Use}_{e} &= \{y = e'; \mid y \in \mathit{Vars}_{e}\} \\
\mathit{Def}_{x} &= \{y = e'; \mid y \equiv x \vee x \in \mathit{Vars}_{e'}\}
\end{aligned}
$$

... where:

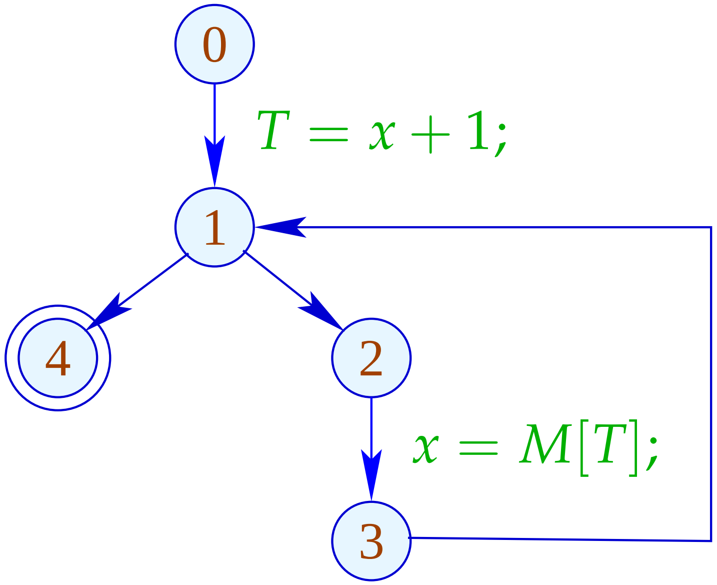
$$\begin{array}{lll} \mathit{Use}_e & = & \{y = e'; \mid y \in \mathit{Vars}_e\} \\ \mathit{Def}_x & = & \{y = e'; \mid y \equiv x \vee x \in \mathit{Vars}_{e'}\} \end{array}$$

For the remaining edges, we define:

$$\begin{array}{lllll} [\![x = M[e];]\!]^\sharp\, D & = & D \backslash (\mathit{Use}_e \cup \mathit{Def}_x) \\ [\![M[e_1] = e_2;]\!]^\sharp\, D & = & D \backslash (\mathit{Use}_{e_1} \cup \mathit{Use}_{e_2}) \\ [\![\mathsf{Pos}(e)]\!]^\sharp\, D & = & [\![\mathsf{Neg}(e)]\!]^\sharp\, D & = & D \backslash \mathit{Use}_e \end{array}$$

Warning:

We may move $y = e;$ beyond a join only if $y = e;$ can be delayed along all joining edges:



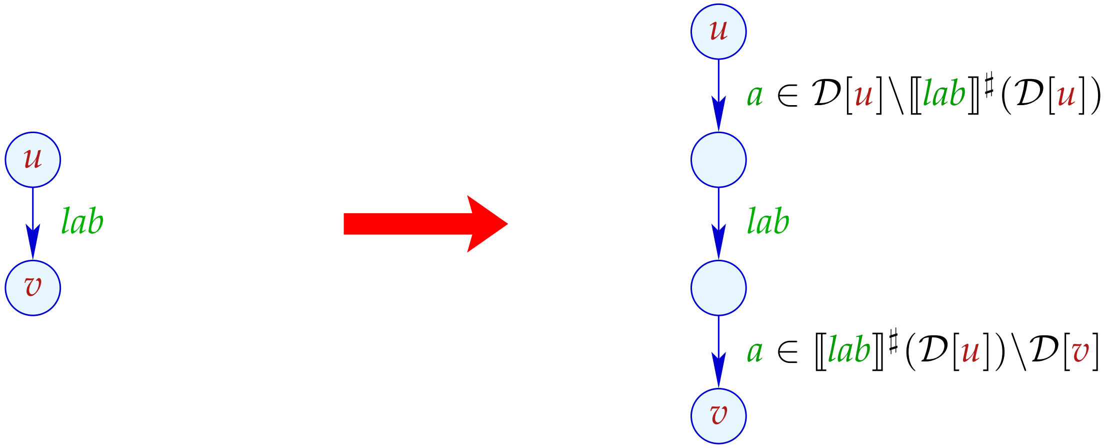
Here, $T = x + 1;$ cannot be moved beyond 1 !!!

We conclude:

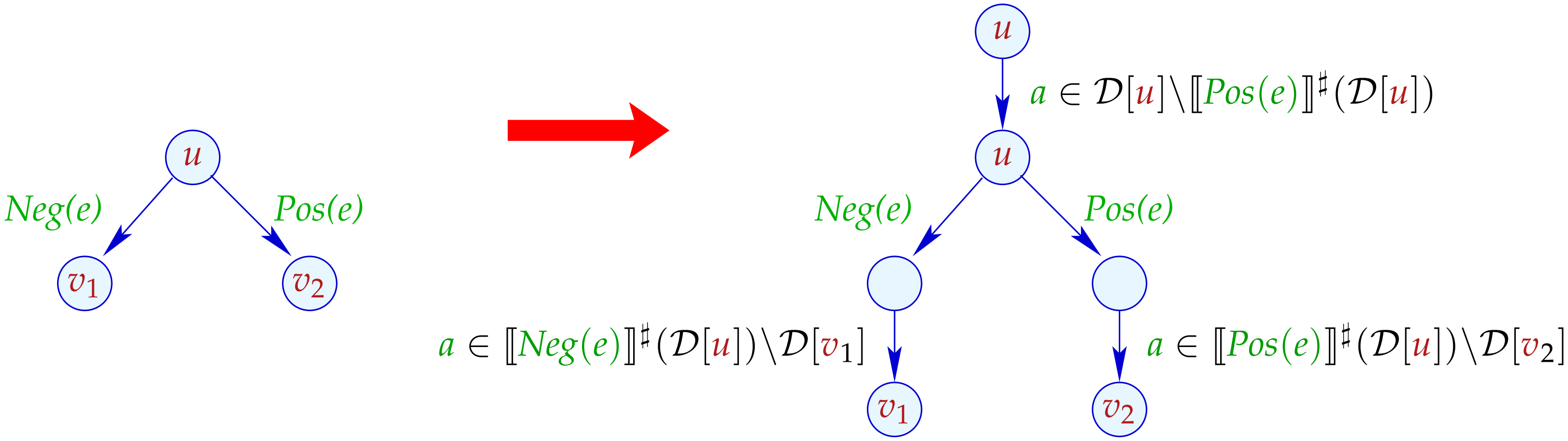
- The partial ordering of the lattice for delayability is given by "$\supseteq$".
- At program start: $D_0 = \emptyset$.

  Therefore, the sets $\mathcal{D}[u]$ of at u delayable assignments can be computed by solving a system of constraints.
- We delay only assignments a where $a\ a$ has the same effect as a alone.
- The extra insertions render the original assignments as assignments to dead variables ...

Transformation 7:



u
lab
v
u
$a \in \mathcal{D}[u] \backslash [\![lab]\!]^\sharp(\mathcal{D}[u])$
lab
$a \in [\![lab]\!]^\sharp(\mathcal{D}[u]) \backslash \mathcal{D}[v]$
v

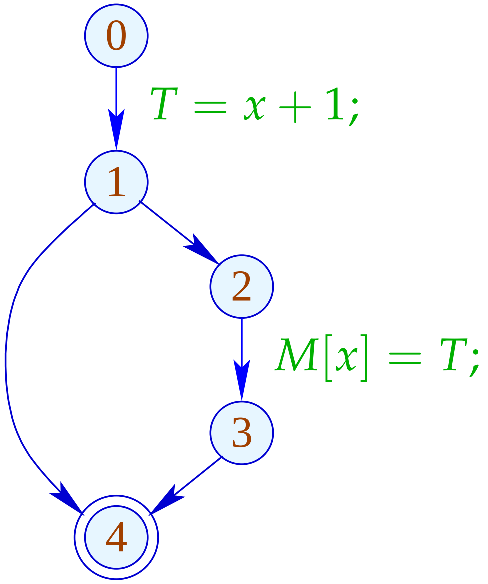


u
$Neg(e)$
$Pos(e)$
v_1
v_2
u
$a \in \mathcal{D}[u] \backslash [\![Pos(e)]\!]^\sharp(\mathcal{D}[u])$
u
$Neg(e)$
$Pos(e)$
$a \in [\![Neg(e)]\!]^\sharp(\mathcal{D}[u]) \backslash \mathcal{D}[v_1]$
$a \in [\![Pos(e)]\!]^\sharp(\mathcal{D}[u]) \backslash \mathcal{D}[v_2]$
v_1
v_2

Note:

Transformation T7 is only meaningful, if we subsequently eliminate assignments to dead variables by means of transformation T2 :-)

In the example, the partially dead code is eliminated:

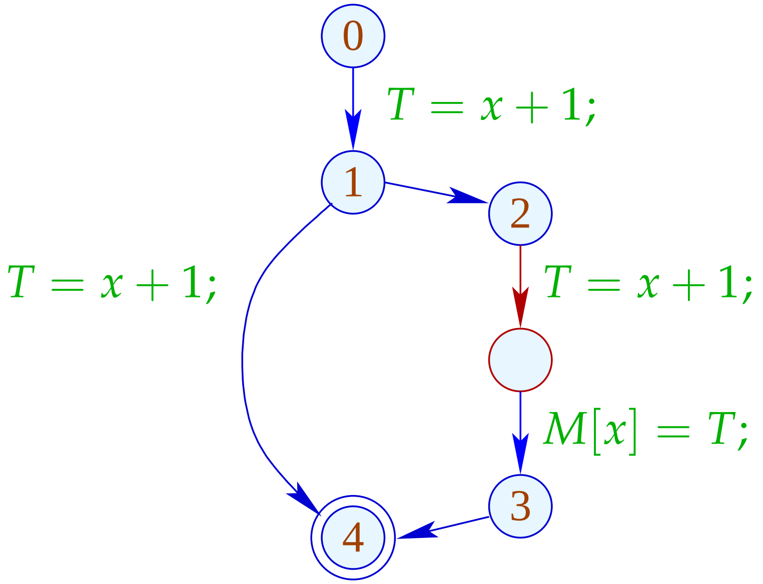


	$\mathcal{D}$
0	$\emptyset$
1	$\{T = x + 1;\}$
2	$\{T = x + 1;\}$
3	$\emptyset$
4	$\emptyset$

Note:

Transformation T7 is only meaningful, if we subsequently eliminate assignments to dead variables by means of transformation T2 :-)

In the example, the partially dead code is eliminated:

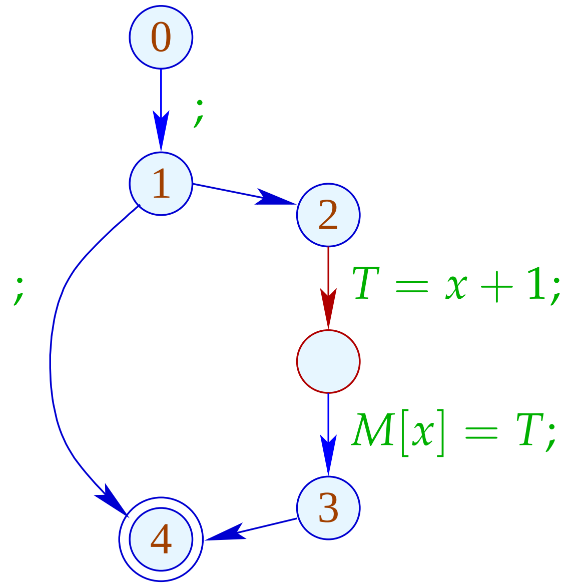


	$\mathcal{D}$
0	$\emptyset$
1	$\{T = x + 1;\}$
2	$\{T = x + 1;\}$
3	$\emptyset$
4	$\emptyset$

Note:

Transformation T7 is only meaningful, if we subsequently eliminate assignments to dead variables by means of transformation T2 :-)

In the example, the partially dead code is eliminated:



	$\mathcal{L}$
0	$\{x\}$
1	$\{x\}$
2	$\{x\}$
$2'$	$\{x, T\}$
3	$\emptyset$
4	$\emptyset$

Remarks:

- After $T7$, all original assignments $y = e;$ with $y \not\in Vars_e$ are assignments to dead variables and thus can always be eliminated :-)
- By this, it can be proven that the transformation is guaranteed to be non-degradating efficiency of the code :-))
- Similar to the elimination of partial redundancies, the transformation can be repeated :-}

Conclusion:

- → The design of a meaningful optimization is non-trivial.
- → Many transformations are advantageous only in connection with other optimizations :-)
- → The ordering of applied optimizations matters !!
- → Some optimizations can be iterated !!!

... a menaingful ordering:

T4	Constant Propagation Interval Analysis Alias Analysis
T6	Loop Rotation
T1, T3, T2	Available Expressions
T2	Dead Variables
T7, T2	Partially Dead Code
T5, T3, T2	Partially Redundant Code

2 Replacing Expensive Operations by Cheaper Ones

2.1 Reduction of Strength

(1) Tabulation of Polynomials

$$f\,(x) \;=\; a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_1 \cdot x + a_0$$

	Multiplications	Additions
naive	$\frac{1}{2}n(n+1)$	n
re-use	$2n-1$	n
Horner-Schema	n	n

Idea:

$$f(x) = (\ldots((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2}) \ldots) \cdot x + a_0$$

(2) Tabulation of a polynomial $f(x)$ of degree n :

→ To recompute $f(x)$ for every argument x is too expensive :-)

→ Luckily, the n-th differences are constant !!!

Example: $f(x) = 3x^3 - 5x^2 + 4x + 13$

n	$f(n)$	Δ	Δ^2	Δ^3
0	13	2	8	$\boxed{18}$
1	15	10	$\boxed{26}$	
2	25	$\boxed{36}$		
3	$\boxed{61}$			
4	...			

Here, the n-th difference is always

$$\Delta_h^n(f) = n! \cdot a_n \cdot h^n \qquad (h \text{ step width})$$

Costs:

- n times evaluation of f ;
- $\frac{1}{2} \cdot (n-1) \cdot n$ subtractions to determine the Δ^k ;
- $2n-2$ multiplications for computing $\Delta_h^n(f)$;
- n additions for every further value :-)

$\Longrightarrow$

Number of multiplications only depends on n :-))

Simple Case: $f(x) = a_1 \cdot x + a_0$

- ... naturally occurs in many numerical loops :-)
- The first differences are already constant:

$$f(x+h) - f(x) = a_1 \cdot h$$

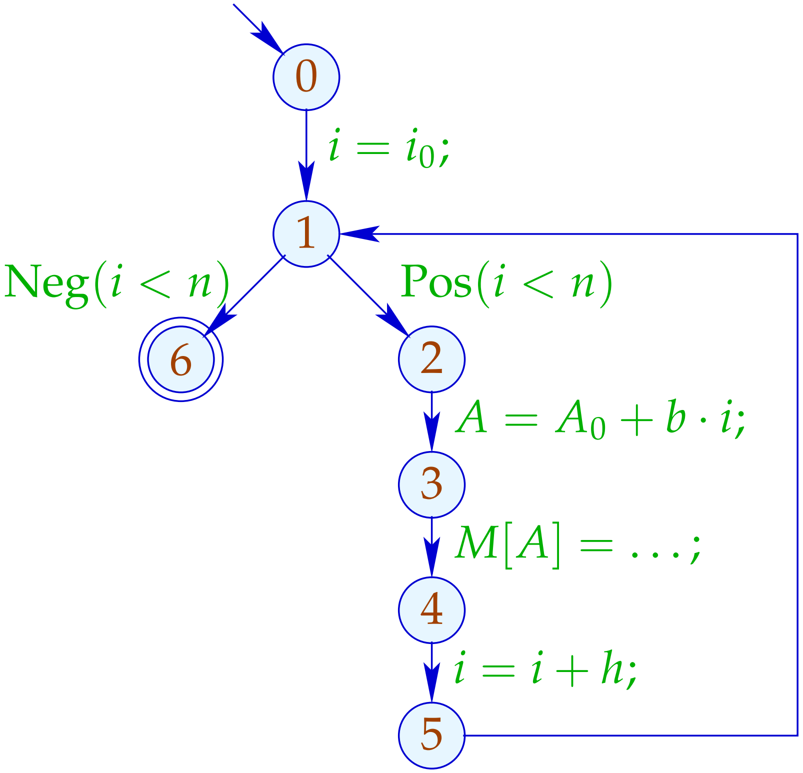
- Instead of the sequence: $y_i = f(x_0 + i \cdot h), \quad i \geq 0$

  we compute: $y_0 = f(x_0), \quad \Delta = a_1 \cdot h$

  $y_i = y_{i-1} + \Delta, \quad i > 0$

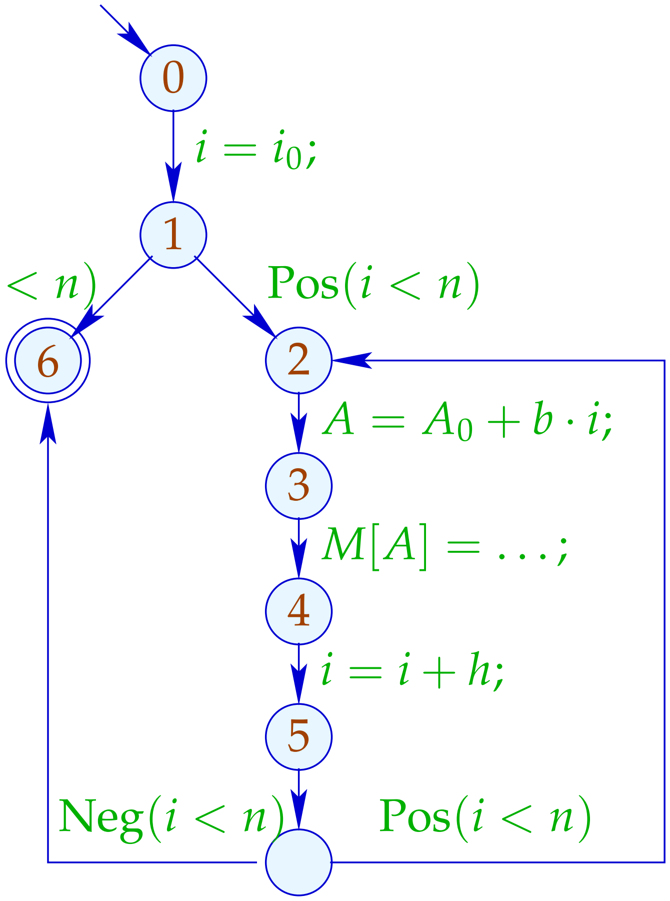
Example:

$$
\begin{aligned}
&\text{for } (i = i_0; i < n; i = i + h) \ \{ \\
&\qquad\qquad A = A_0 + b \cdot i; \\
&\qquad\qquad M[A] = \ldots; \\
&\}
\end{aligned}
$$



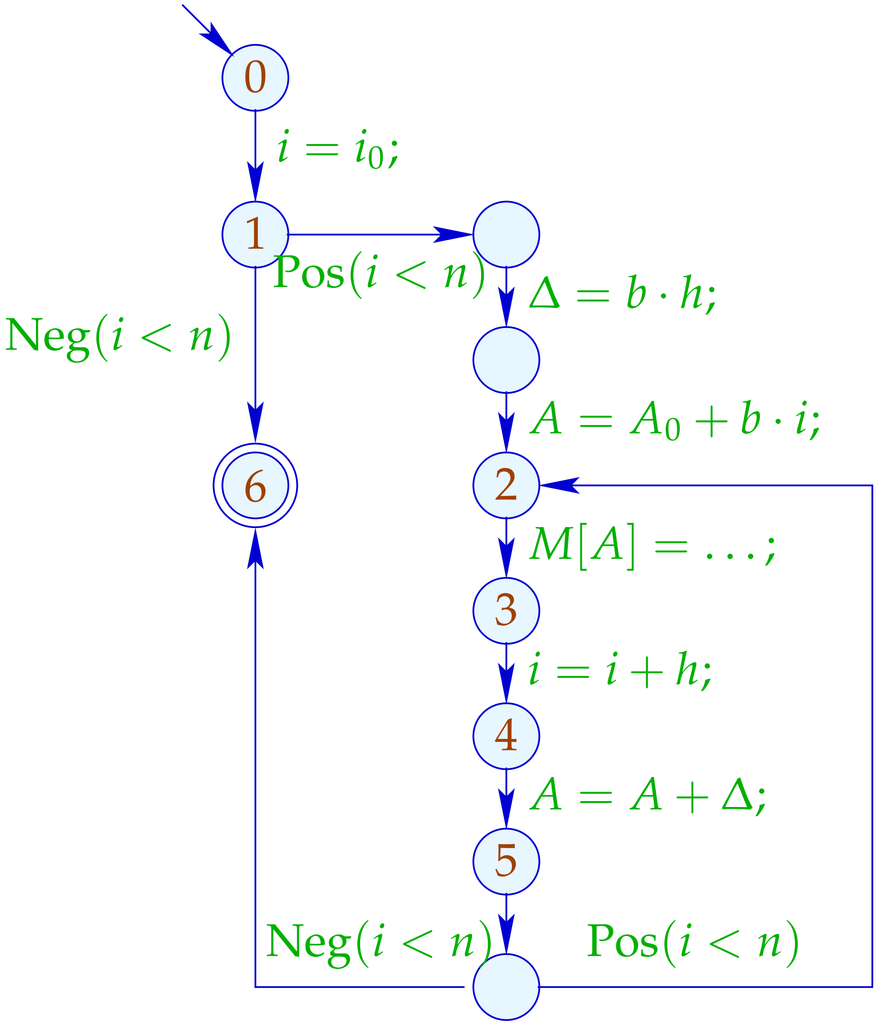
... or, after loop rotation:

```
$i = i_0;$
if $(i < n)$ do {
        $A = A_0 + b \cdot i;$
        $M[A] = \ldots;$
        $i = i + h;$
    } while $(i < n);$
```



... and reduction of strength:

```
$i = i_0;$
if $(i < n)$ {
        $\Delta = b \cdot h;$
        $A = A_0 + b \cdot i_0;$
        do {
                $M[A] = \ldots;$
                $i = i + h;$
                $A = A + \Delta;$
        } while $(i < n);$
}
```



Warning:

- The values b, h, A_0 must not change their values during the loop.
- i, A may be modified at exactly one position in the loop :-(
- One may try to eliminate the variable i altogether :

  → i may not be used else-where.

  → The initialization must be transformed into: $A = A_0 + b \cdot i_0$.

  → The loop condition $i < n$ must be transformed into: $A < N$ for $N = A_0 + b \cdot n$.

  → b must always be different from zero !!!

Approach:

Identify

. . . loops;

. . . iteration variables;

. . . constants;

. . . the matching use structures.

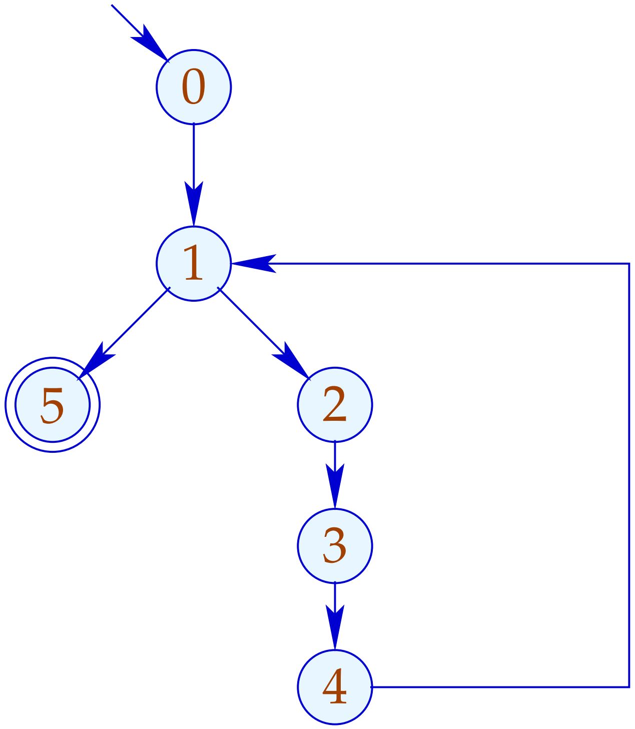
Loops:

... are identified through the node v with back edge $(\_, \_, v)$:-)

For the sub-graph G_v of the cfg on $\{w \mid v \Rightarrow w\}$, we define:

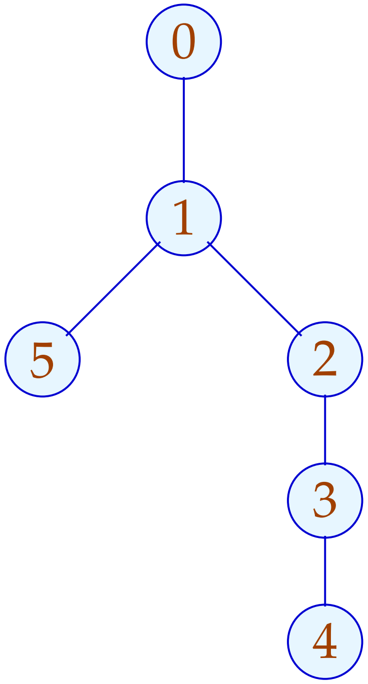
$$\mathsf{Loop}[v] \quad = \quad \{w \mid w \longrightarrow^* v \;\; \text{in} \;\; G_v\}$$

Example:



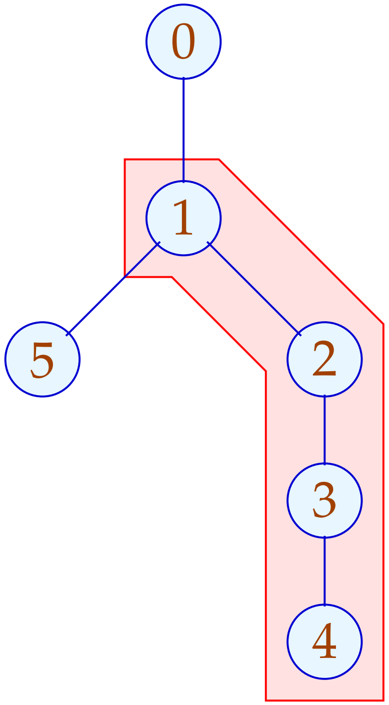
	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

Example:



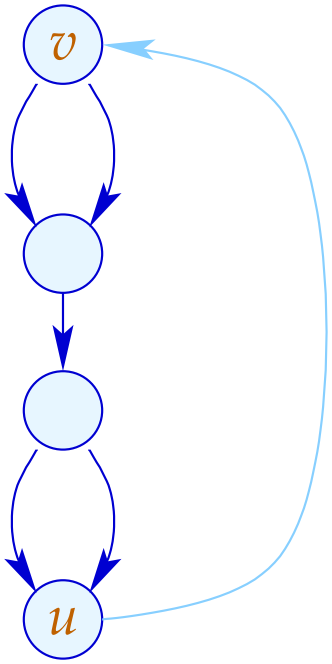
	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

Example:



	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

We are interested in edges which during each iteration are executed exactly once:



Graph-theoretically, this is noot easily expressible :-(

Edges k could be selected such that:

- the sub-graph $G = \mathsf{Loop}[v] \backslash \{(\_, \_, v)\}$ is connected;
- the graph $G \backslash \{k\}$ is split into two unconnected sub-graphs.

Edges k could be selected such that:

- the sub-graph $G = \mathsf{Loop}[v]\backslash\{(\_,\_,v)\}$ is connected;
- the graph $G\backslash\{k\}$ is split into two unconnected sub-graphs.

On the level of source programs, this is trivial:

```
do { s_1 ... s_k
   } while (e);
```

The desired assignments must be among the s_i :-)

Iteration Variable:

i is an iteration variable if the only definition of i inside the loop occurs at an edge which separates the body and is of the form:

$$i = i + h;$$

for some loop constant h .

A loop constant is simply a constant (e.g., 42), or slightly more libaral, an expression which only depends on variables which are not modified during the loop :-)

(3) Differences for Sets

Consider the fixpoint computation:

```
$x = \emptyset;$
for $(t = F\,x; t \not\subseteq x; \boxed{t = F\,x;})$
        $x = x \cup t;$
```

If F is distributive, it could be replaced by:

```
$x = \emptyset;$
for $(\Delta = F\,x; \Delta \neq \emptyset; \boxed{\Delta = (F\,\Delta) \setminus x;})$
        $x = x \cup \Delta;$
```

The function F must only be computed for the smaller sets Δ

:-) semi-naive iteration

Instead of the sequence: $\emptyset \quad \subseteq \quad F\,(\emptyset) \quad \subseteq \quad F^2\,(\emptyset) \quad \subseteq \quad \ldots$

we compute: $\Delta_1 \quad \cup \quad \Delta_2 \quad \cup \quad \ldots$

where:

$$\begin{aligned} \Delta_{i+1} &= F\,(F^i(\emptyset))\backslash F^i(\emptyset) \\ &= F\,(\Delta_i)\backslash(\Delta_1 \cup \ldots \cup \Delta_i) \quad \text{with} \quad \Delta_0 = \emptyset \end{aligned}$$

Assume that the costs of $F\ x$ is $1 + \#x$.

Then the costs sum up to:

naive	$1 + 2 + \ldots + n + n \quad = \quad \frac{1}{2}n(n+3)$
semi-naive	$2n$

where n is the cardinality of the result.

$\Longrightarrow$ A linear factor is saved :-)

2.2 Peephole Optimization

Idea:

- Slide a small window over the program.
- Optimize agressively inside the window, i.e.,

  → Eliminate redundancies!

  → Replace expensive operations inside the window by cheaper ones!

Examples:

$x = x + 1;$ $\Longrightarrow$ $x{+}{+};$

// given that there is a specific increment instruction :-)

$z = y - a + a;$ $\Longrightarrow$ $z = y;$

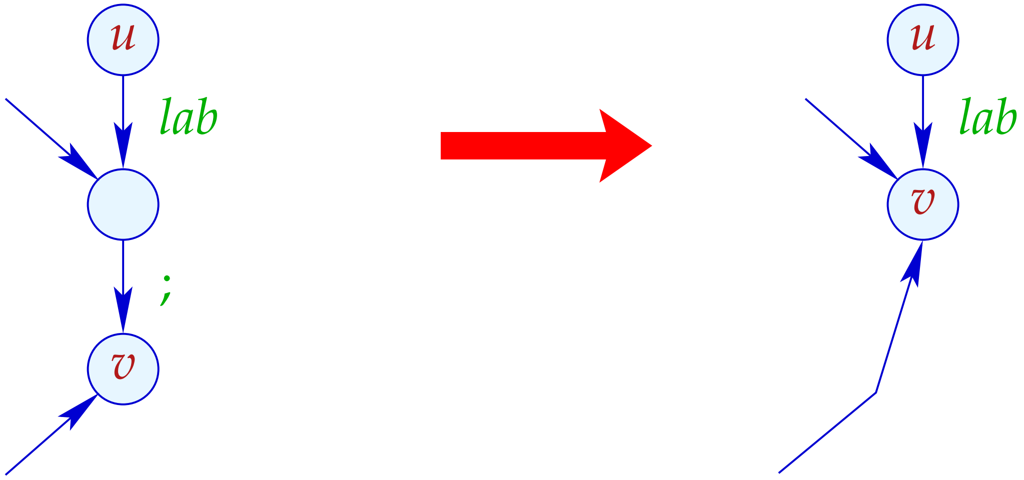
// algebraic simplifications :-)

$x = x;$ $\Longrightarrow$ $;$

$x = 0;$ $\Longrightarrow$ $x = x \oplus x;$

$x = 2 \cdot x;$ $\Longrightarrow$ $x = x + x;$

Important Subproblem: *nop*-Optimization



- → If $(v_1, ;, v)$ is an edge, v_1 has no further out-going edge.
- → Consequently, we can identify v_1 and v :-)
- → The ordering of the identifications does not matter :-))

Implementation:

- We construct a function $\mathsf{next} : \mathit{Nodes} \to \mathit{Nodes}$ with:

$$\mathsf{next}\ u = \begin{cases} \mathsf{next}\ v & \text{if} \quad (u, ;, v) \quad \text{edge} \\ u & \text{otherwise} \end{cases}$$

  Warning: This definition is only recursive if there are ;-loops ???

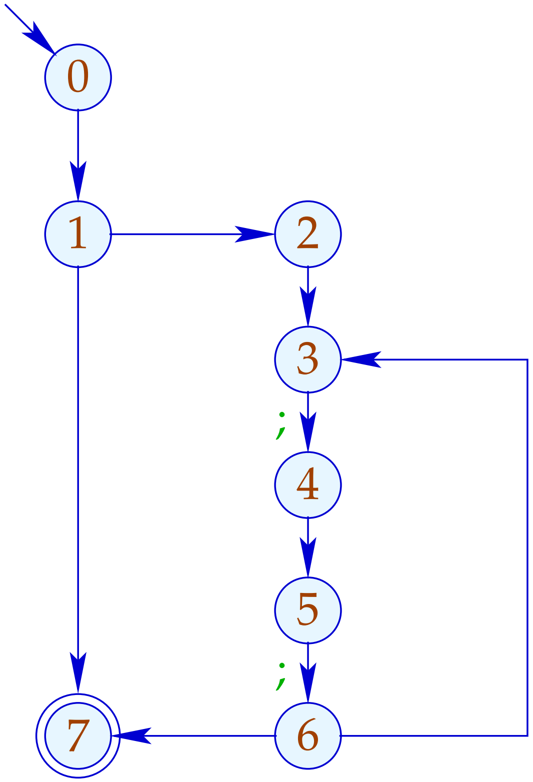
- We replace every edge:

$$(u, \mathit{lab}, v) \qquad \Longrightarrow \qquad (u, \mathit{lab}, \mathsf{next}\ v)$$

  ... whenever $\mathit{lab} \neq ;$

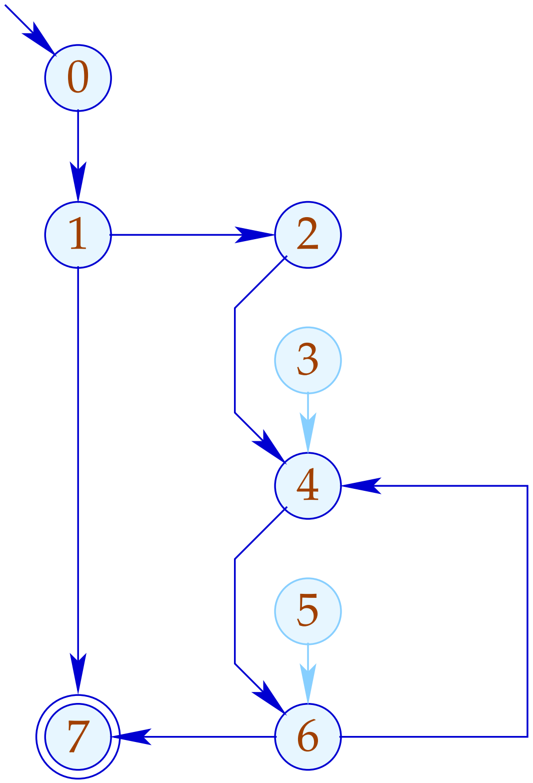
- All ;-edges are removed ;-)

Example:



$$\begin{aligned} \text{next } 1 &= 1 \\ \text{next } 3 &= 4 \\ \text{next } 5 &= 6 \end{aligned}$$

Example:



$$\text{next } 1 = 1$$
$$\text{next } 3 = 4$$
$$\text{next } 5 = 6$$

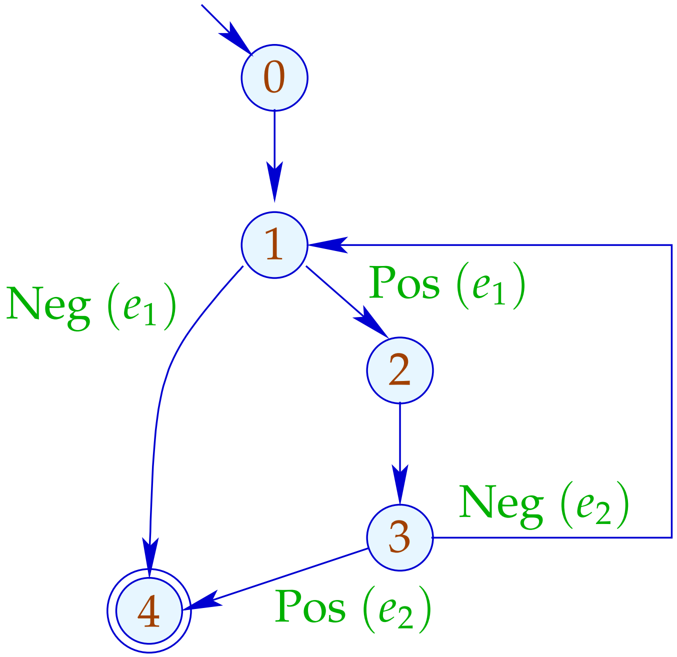
2. Subproblem: Linearization

After optimization, the CFG must again be brought into a linearly arrangement of instructions :-)

Warning:

Not every linearization is equally efficient !!!

Example:



0:

1: if (e_1) goto 2;

4: halt

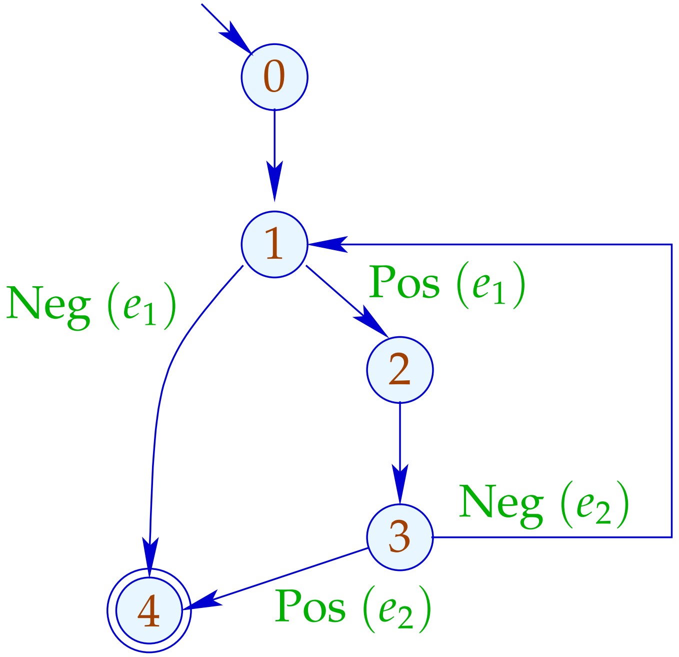
2: $\boxed{\text{Rumpf}}$

3: if (e_2) goto 4;

goto 1;

Bad: The loop body is jumped into :-(

Example:



```
0:
1:    if (!e1) goto 4;
2:    Rumpf
3:    if (!e2) goto 1;
4:    halt
```

// better cache behavior :-)

Idea:

- Assign to each node a temperature!
- always jumps to

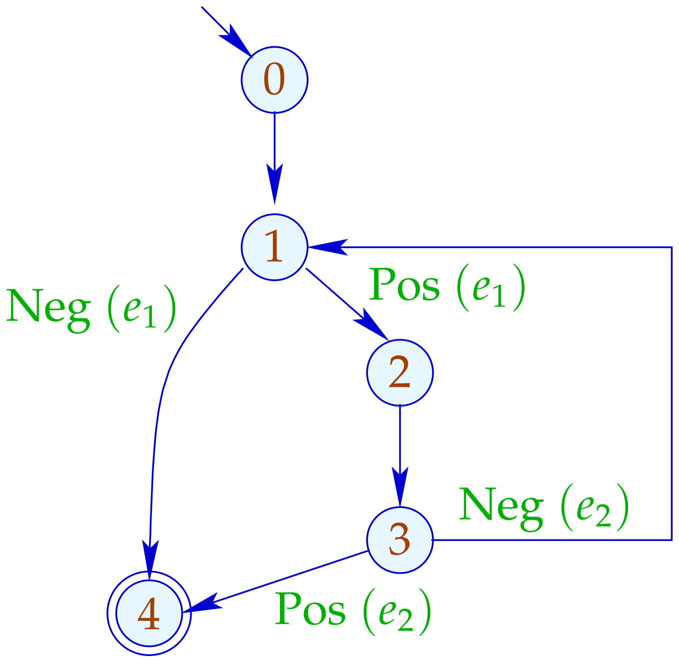
  (1) nodes which have already been handled;

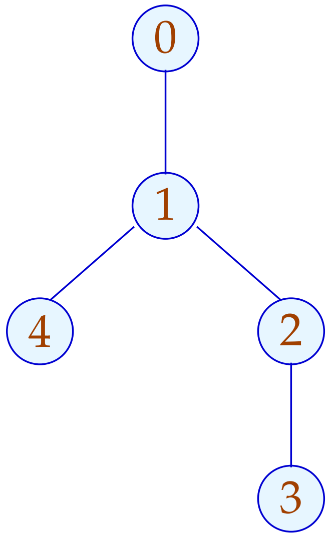
  (2) colder nodes.

- Temperature $\approx$ nesting-depth

For the computation, we use the pre-dominator tree and strongly connected components ...

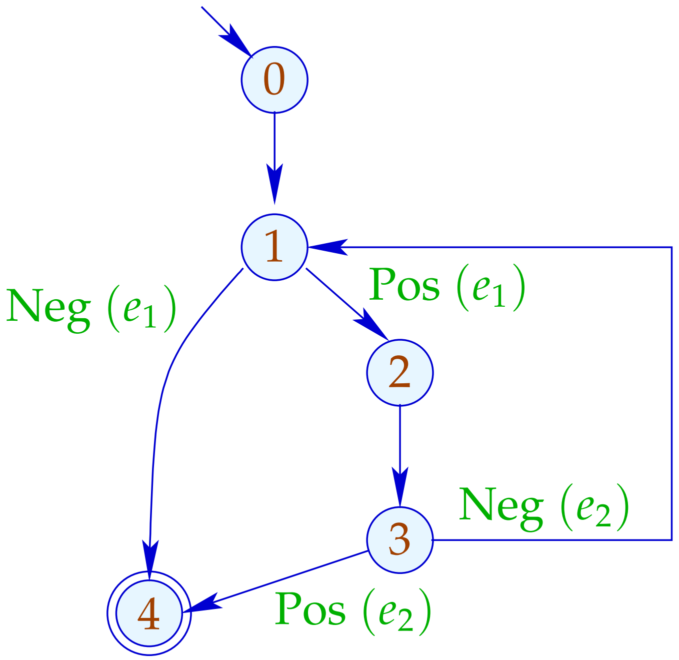
... in the Example:

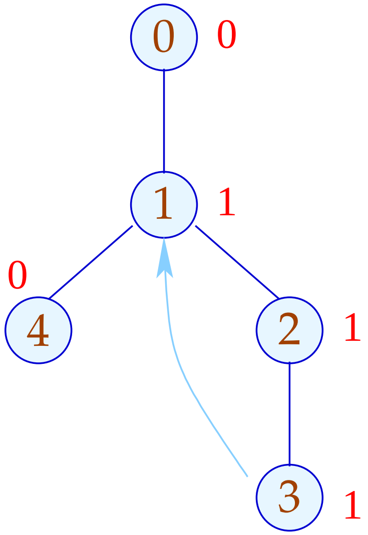




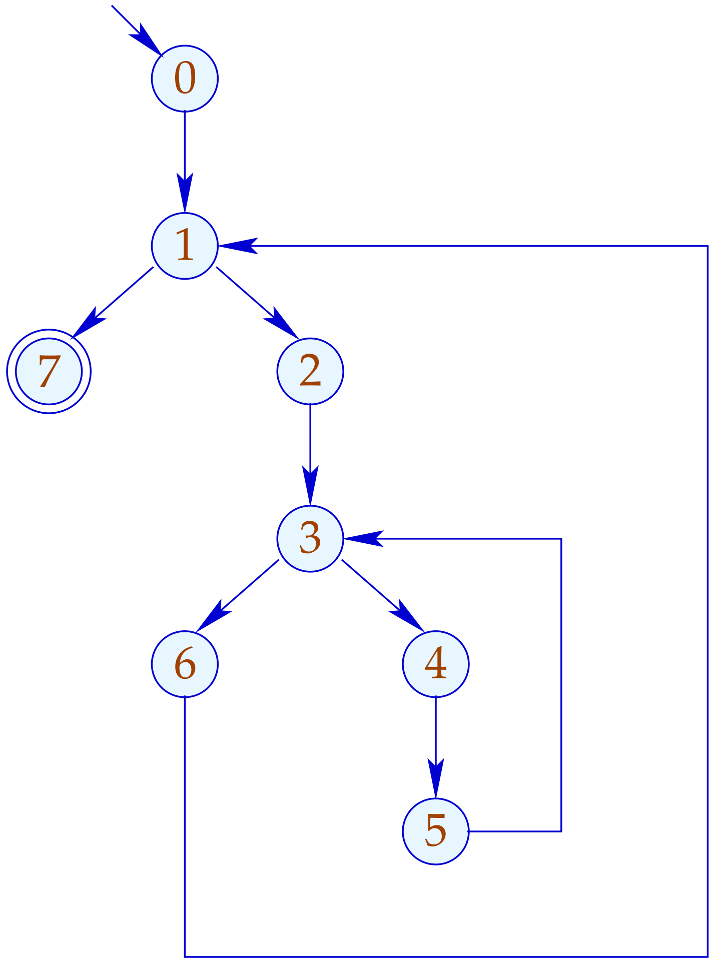
The sub-tree with back edge is hotter ...

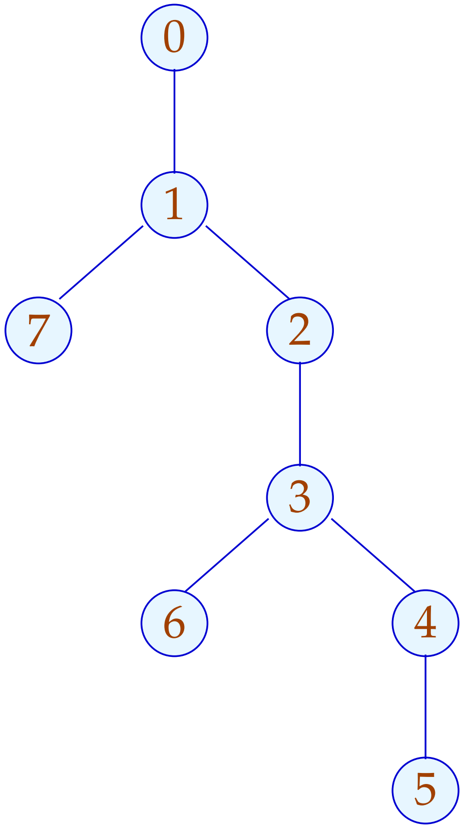
... in the Example:



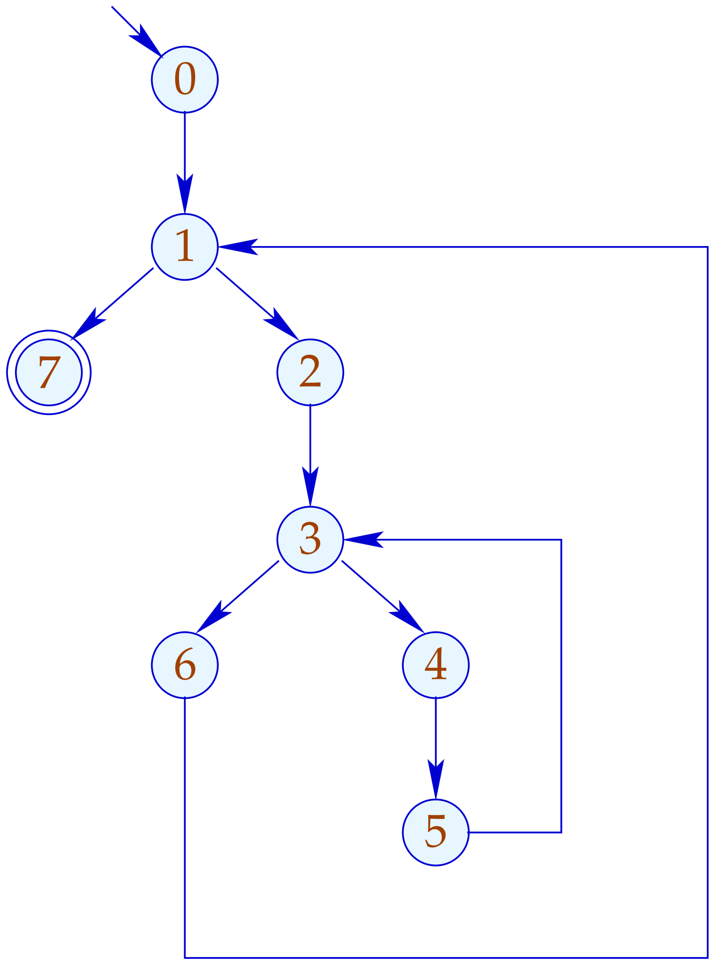


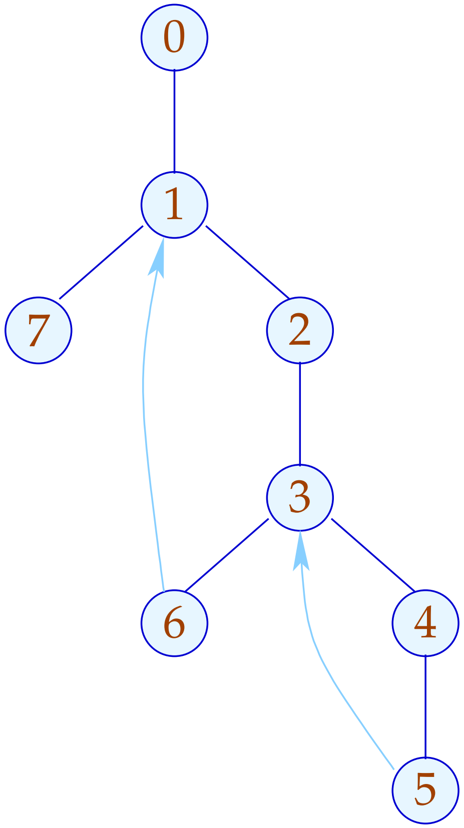
More Complicated Example:



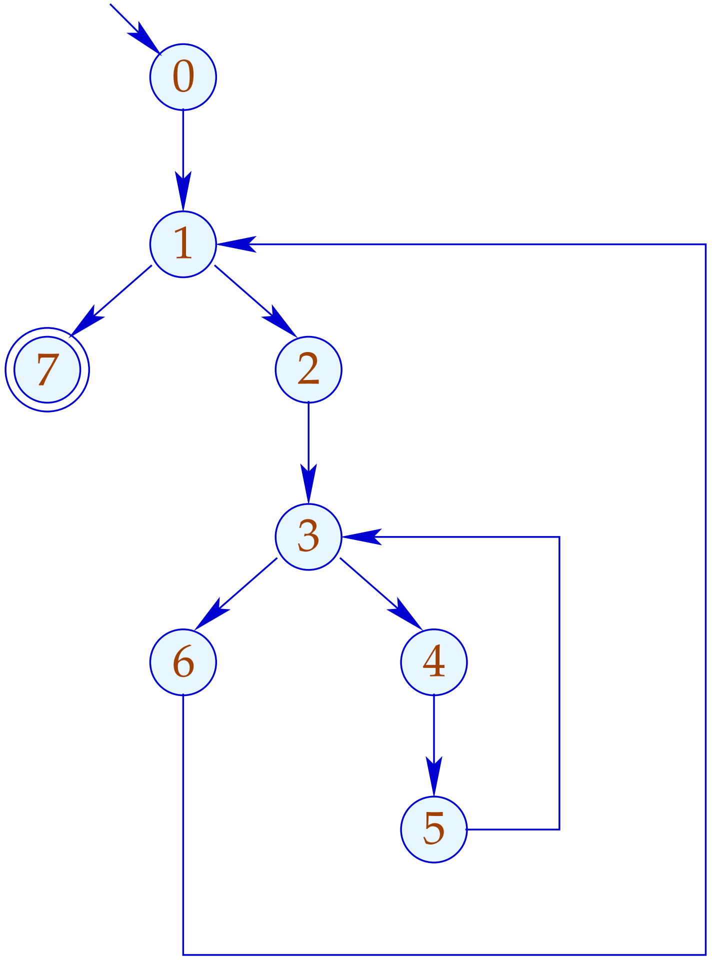


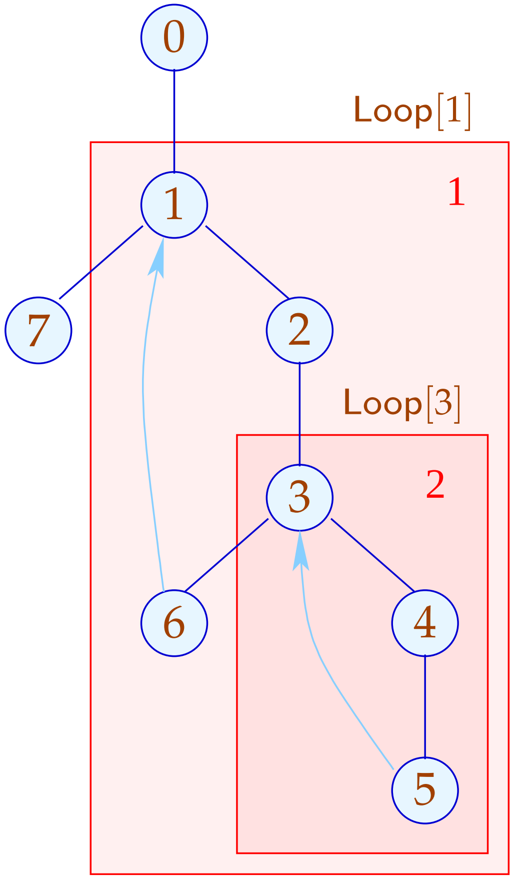
More Complicated Example:





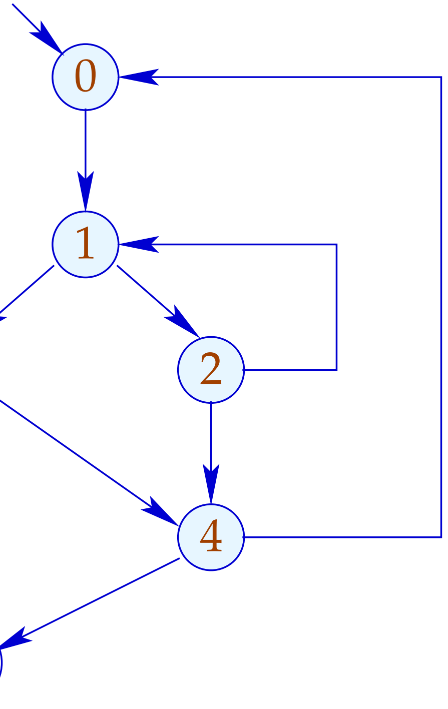
More Complicated Example:

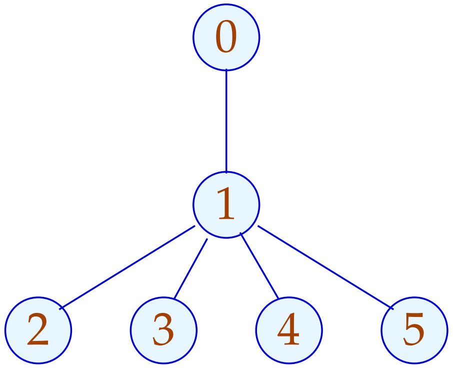




Our definition of Loop implies that (detected) loops are necessarily nested :-)

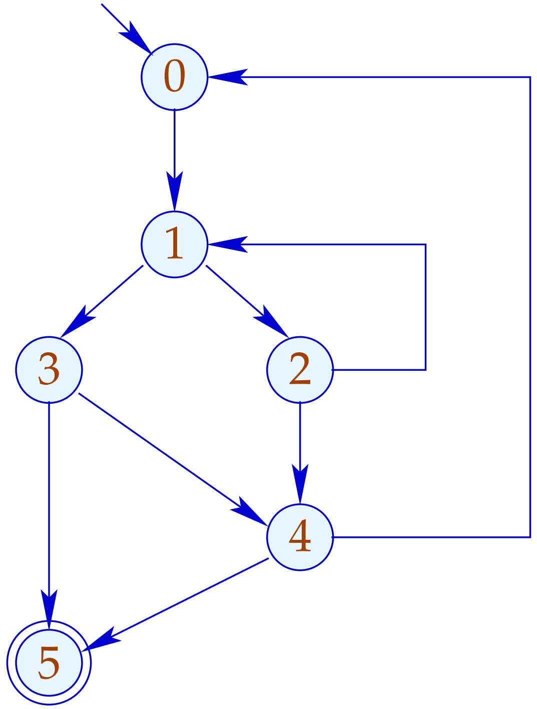
Is is also meaningful for do-while-loops with breaks ...

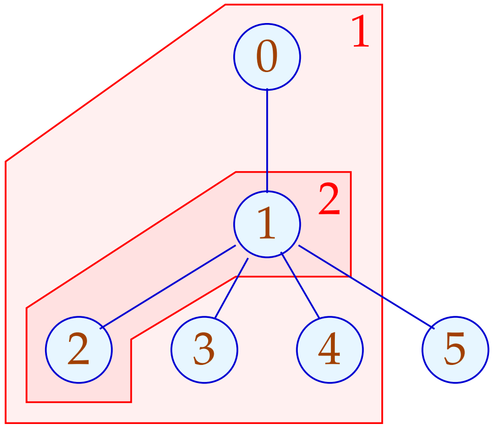




Our definition of Loop implies that (detected) loops are necessarily nested :-)

Is is also meaningful for do-while-loops with breaks ...





Summary: The Approach

(1) For every node, determine a temperature;

(2) Pre-order-DFS over the CFG;

→ If an edge leads to a node we already have generated code for, then we insert a jump.

→ If a node has two successors with different temperature, then we insert a jump to the colder of the two.

→ If both successors are equally warm, then it does not matter ;-)

2.3 Procedures

We extend our mini-programming language by procedures without parameters and procedure calls.

For that, we introduce a new statement:

$$f();$$

Every procedure f has a definition:

$$f\,()\ \{\ stmt^*\ \}$$

Additionally, we distinguish between global and local variables.

Program execution starts with the call of a procedure $\mathsf{main}\,()$.

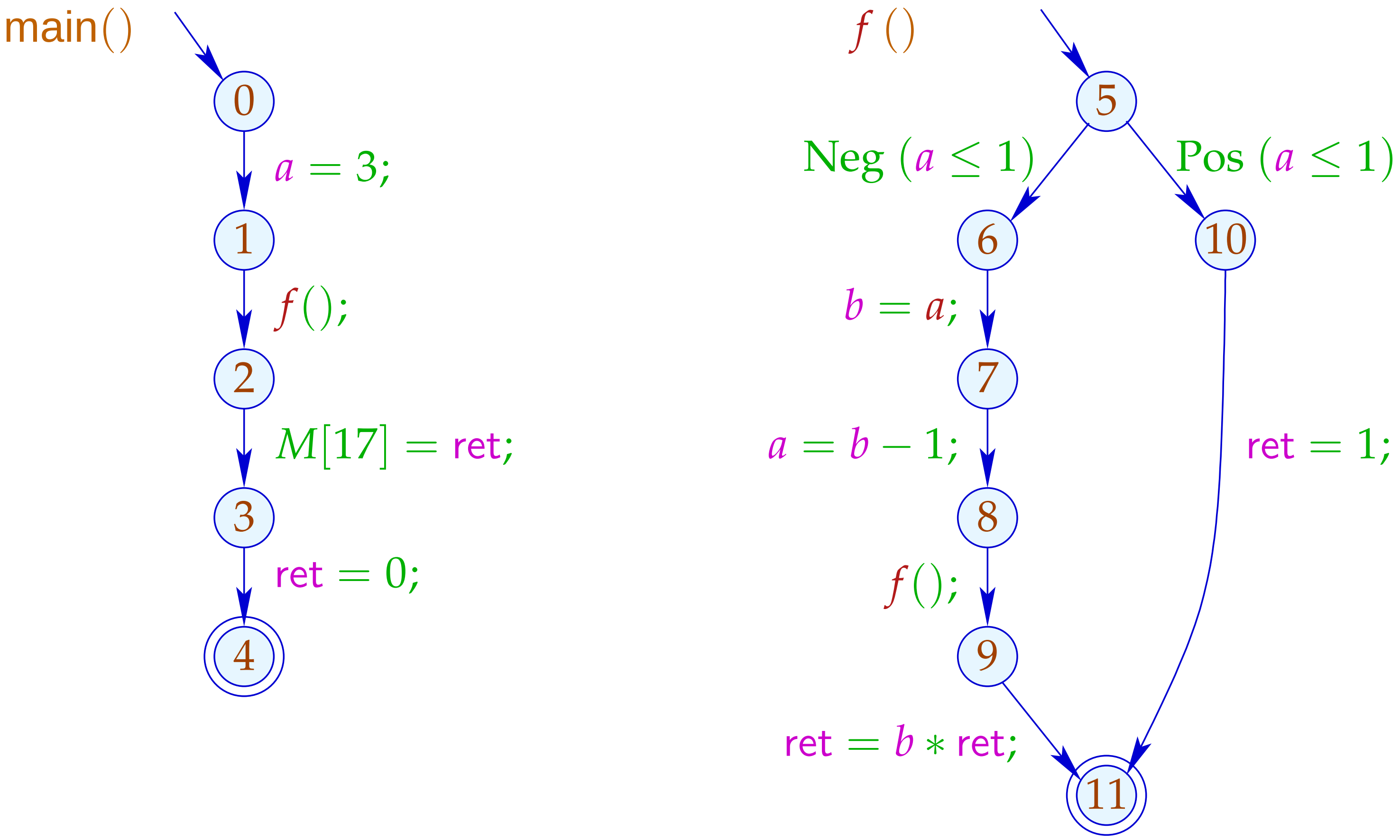
Example:

```
int  a, ret;
main ()  {
        a = 3;
        f ();
        M[17] = ret;
        ret = 0;
}
```

```
f ()  {
        int  b;
        if (a ≤ 1) {ret = 1; goto exit; }
        b = a;
        a = b − 1;
        f ();
        ret = b · ret;
exit :
}
```

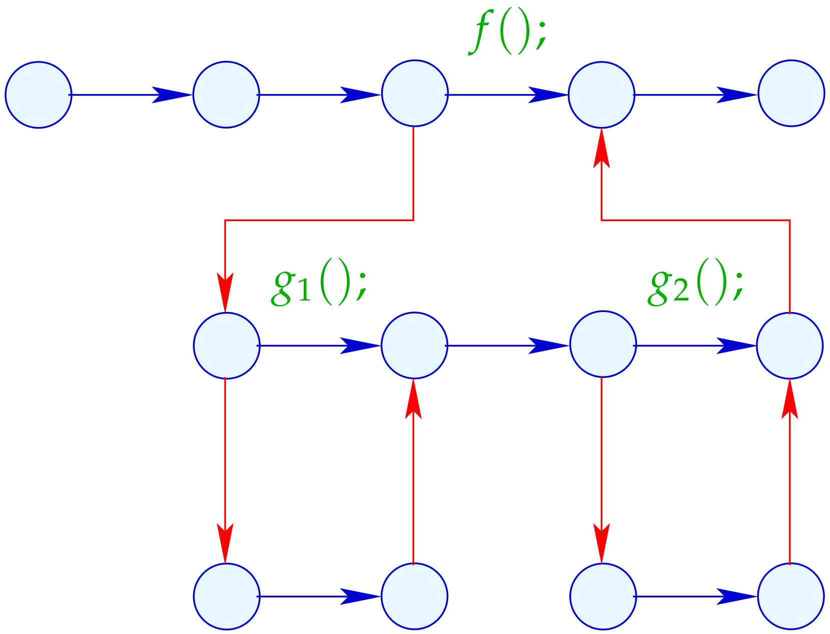
Such programs can be represented by a set of CFGs: one for each procedure ...

... in the Example:

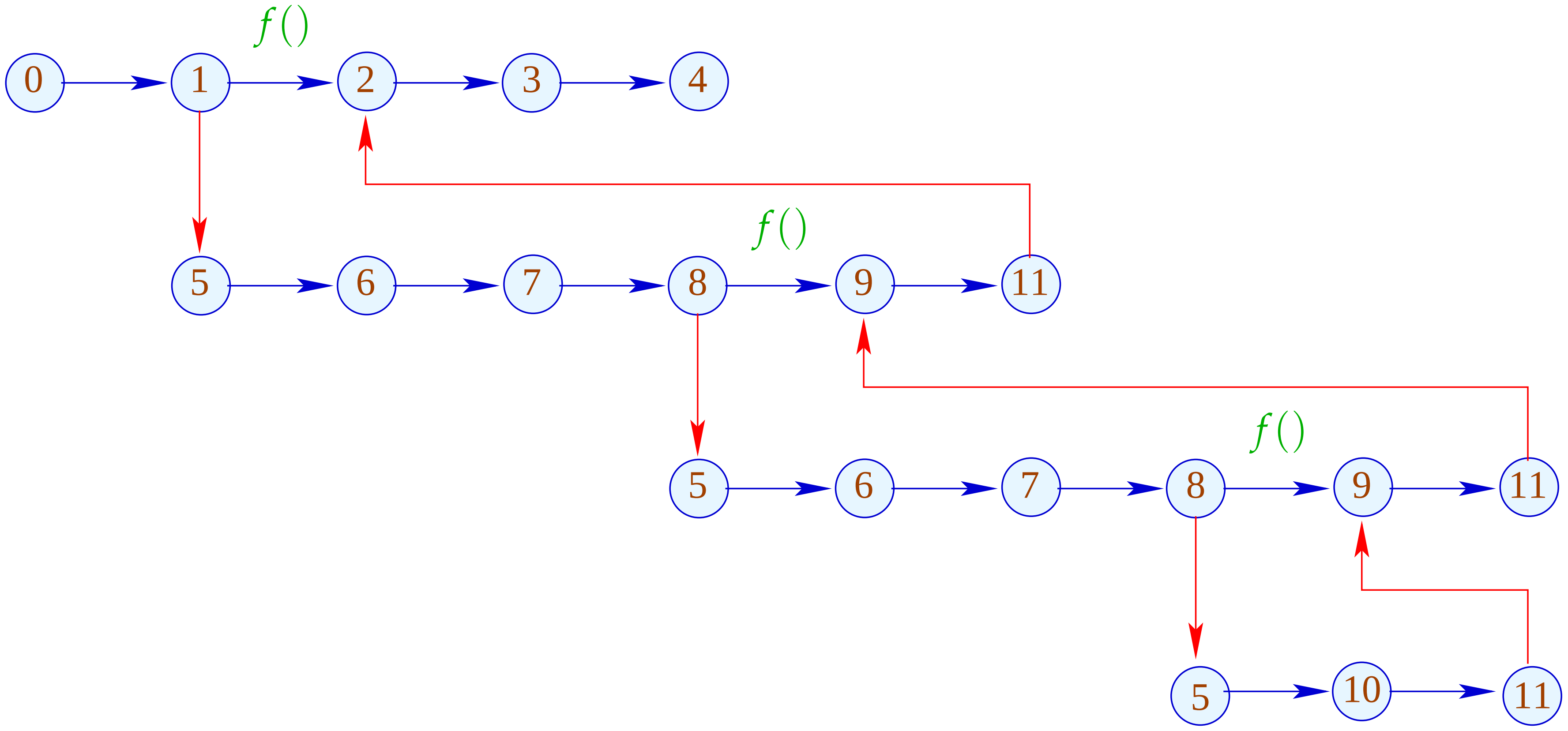


In order to optimize such programs, we require an extended operational semantics ;-)

Program executions are no longer paths, but forests:



... in the Example:



The function $[\![.]\!]$ is extended to computation forests: w :

$$[\![w]\!] : (\mathit{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z}) \to (\mathit{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z})$$

For a call $k = (u, f();, v)$ we must:

- determine the initial values for the locals:

$$\mathsf{enter}\ \rho = \{x \mapsto 0 \mid x \in \mathit{Locals}\} \oplus (\rho|_{\mathit{Globals}})$$

- ... combine the new values for the globals with the old values for the locals:

$$\mathsf{combine}\ (\rho_1, \rho_2) = (\rho_1|_{\mathit{Locals}}) \oplus (\rho_2|_{\mathit{Globals}})$$

- ... evaluate the computation forest inbetween:

$$\begin{array}{lll} [\![k\ \langle w \rangle]\!]\ (\rho, \mu) & = & \mathsf{let}\ \ (\rho_1, \mu_1) = [\![w]\!]\ (\mathsf{enter}\ \rho, \mu) \\ & & \mathsf{in}\ \ \ (\mathsf{combine}\ (\rho, \rho_1), \mu_1) \end{array}$$

Warning:

- In general, $[\![w]\!]$ is only partially defined :-)
- Dedicated global/local variables a_i, b_i, ret can be used to simulate specific calling conventions.
- The standard operational semantics relies on configurations which maintain a call stack.
- Computation forests are better suited for the construction of analyses and correctness proofs :-)
- It is an awkward (but useful) exercise to prove the equivalence of the two approaches ...

Configurations:

$$\begin{aligned}
\mathit{configuration} \quad & == \quad \mathit{stack} \times \mathit{store} \\
\mathit{store} \quad & == \quad \mathit{globals} \times \mathbb{N} \to \mathbb{Z} \\
\mathit{locals} \quad & == \quad (\mathit{Globals} \to \mathbb{Z}) \\
\mathit{stack} \quad & == \quad \mathit{frame} \cdot \mathit{frame}^{*} \\
\mathit{frame} \quad & == \quad \mathit{point} \times \mathit{locals} \\
\mathit{locals} \quad & == \quad (\mathit{Locals} \to \mathbb{Z})
\end{aligned}$$

A *frame* specifies the local state of computation inside a procedure call :-)

The leftmost frame corresponds to the current call.

Computation steps refer to the current call :-)

The novel kinds of steps:

call $k = (u, f\,();, v)$:

$$(\boxed{(u, \rho)} \cdot \sigma, \langle \gamma, \mu \rangle) \quad \Longrightarrow \quad (\boxed{(u_f, \{x \to 0 \mid x \in Locals\}) \cdot (v, \rho)} \cdot \sigma, \langle \gamma, \mu \rangle)$$

u_f entry point of f

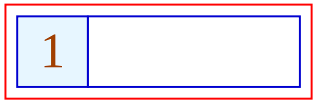
return:

$$(\boxed{(r_f, \_)} \cdot \sigma, \langle \gamma, \mu \rangle) \quad \Longrightarrow \quad (\sigma, \langle \gamma, \mu \rangle)$$

r_f return point of f

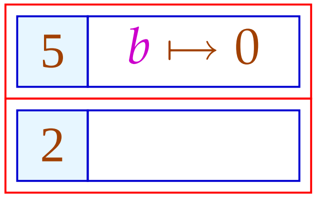
The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:



The call stack explicitly implements the DFS traversal through the computation forest :-)

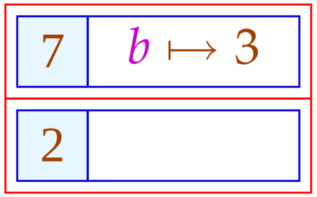
... in the Example:



5	$b \mapsto 0$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

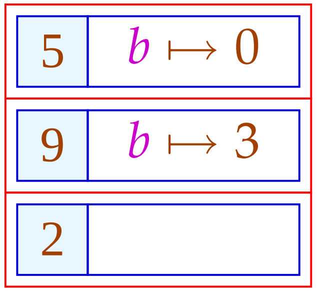
... in the Example:



7	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

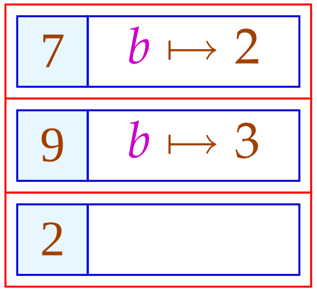
... in the Example:



5	$b \mapsto 0$
9	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:



7	$b \mapsto 2$
9	$b \mapsto 3$
2	

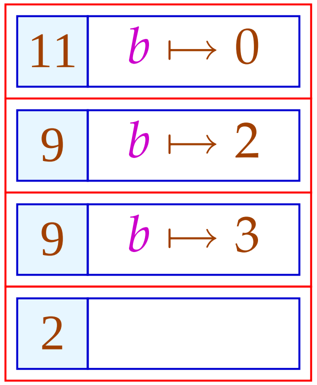
The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:

5	$b \mapsto 0$
9	$b \mapsto 2$
9	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

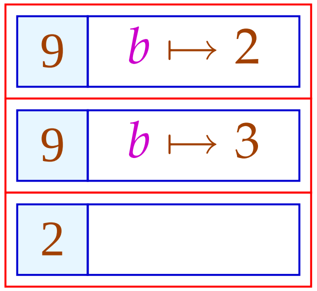
... in the Example:



11	$b \mapsto 0$
9	$b \mapsto 2$
9	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

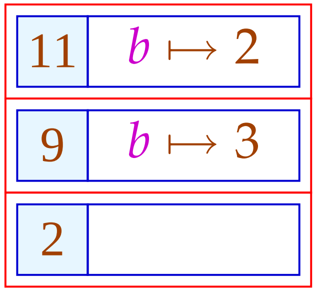
... in the Example:



9	$b \mapsto 2$
9	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

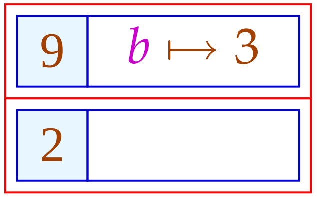
... in the Example:



11	$b \mapsto 2$
9	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest   :-)

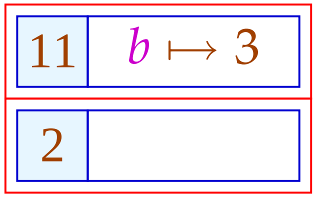
... in the Example:



9	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

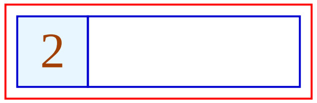
... in the Example:



11	$b \mapsto 3$
2	

The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:



This operational semantics is quite realistic :-)

Costs for a Procedure Call:

Before entering the body:

- Creating a stack frame;
- assing of the parameters;
- Saving the registers;
- Saving the return address;
- Jump to the body.

At procedure exit:

- Freeing the stack frame.
- Restoring the registers.
- Passing of the result.
- Return behind the call.

$\Longrightarrow$ ... quite expensive !!!

1. Idea: Inlining

Copy the procedure body at every call site !!!

Example:

```
abs () {
        $a_2 = -a_1$;
        max ();
}

max () {
        if ($a_1 < a_2$) { ret = $a_2$; goto _exit; }
        ret = $a_1$;
_exit :
}
```

... yields:

```
abs () {
$a_2 = -a_1$;
        if ($a_1 < a_2$) { ret = $a_2$; goto _exit; }
        ret = $a_1$;
  _exit :
}
```

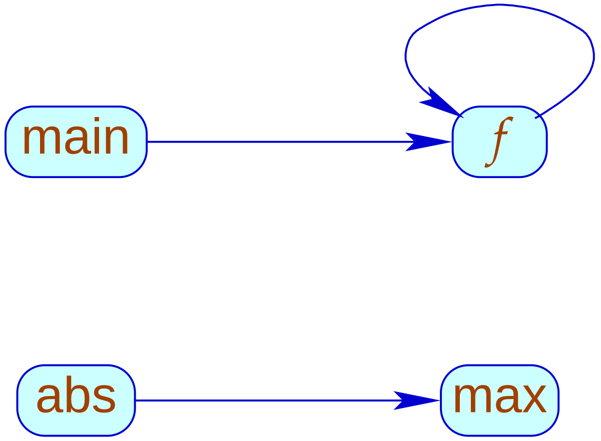
Problems:

- The copied block may modify the locals of the calling procedure ???
- More general: Multiple use of local variable names may lead to errors.
- Multiple calls of a procedure may lead to code duplication :-((
- How can we handle recursion ???

Detection of Recursion:

We construct the call-graph of the program.

In the Examples:



Call-Graph:

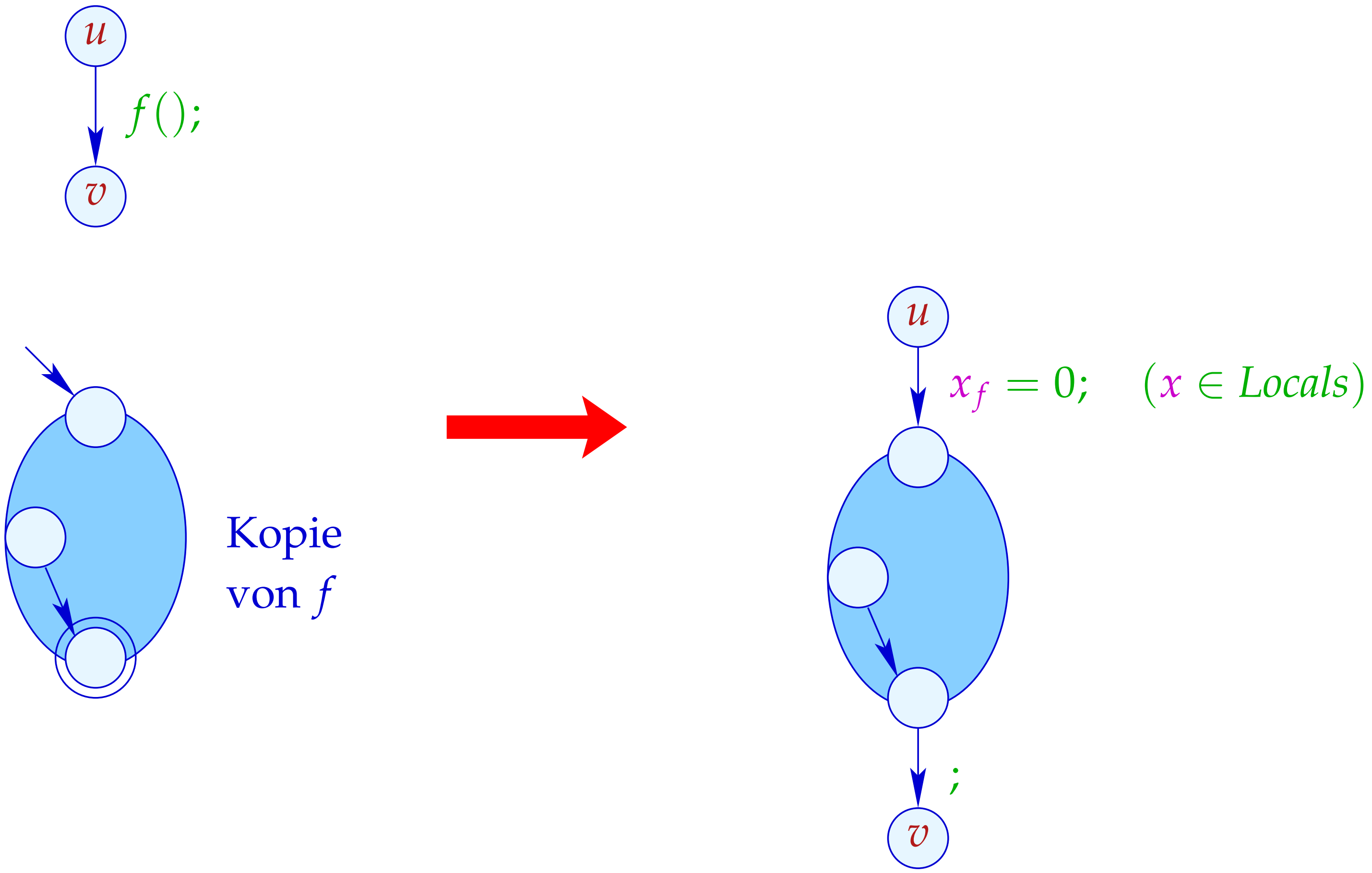
- The nodes are the procedures.
- An edge connexts g with h, whenever the body of g contains a call of h.

Strategies for Inlining:

- Just copy nur leaf-procedures, i.e., procedures without further calls :-)
- Copy all non-recursive procedures!

... here, we consider just leaf-procedures ;-)

Transformation 9:



Note:

- The Nop-edge can be eliminated if the *stop*-node of f has no out-going edges ...
- The x_f are the copies of the locals of the procedure f.
- According to our semantics of procedure calls, these must be initialized with 0 :-)

2. Idea: Elimination of Tail Recursion

```
f () {   int  b;
         if  (a_2 ≤ 1)  {  ret = a_1; goto _exit;  }
         b = a_1 · a_2;
         a_2 = a_2 − 1;
         a_1 = b;
         f ();
_exit :
}
```

After the procedure call, nothing in the body remains to be done.

$\Longrightarrow$ We may directly jump to the beginning :-)

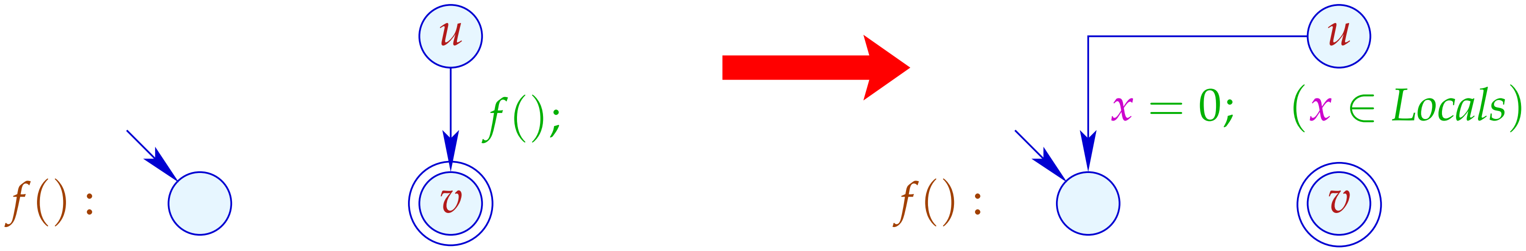
... after having reset the locals to 0.

... this yields in the Example:

```
$f$ () {   int $b$;
 _f :      if ($a_2 \leq 1$) { ret $= a_1$; goto _exit; }
           $b = a_1 \cdot a_2$;
           $a_2 = a_2 - 1$;
           $a_1 = b$;
           $b = 0$;   goto _f;
 _exit :
}
```

// It works, since we have ruled out references to variables!

Transformation 11:



Warning:

→ This optimization is crucial for programming languages without iteration constructs !!!

→ Duplication of code is not necessary :-)

→ No variable renaming is necessary :-)

→ The optimization may also be profitable for non-recursive tail calls :-)

→ The corresponding code may contain jumps from the body of one procedure into the body of another ???

Background 4: Interprocedural Analysis

So far, we can analyze each procedure separately.

→ The costs are moderate :-)

→ The methods also work in presence of separate compilation :-)

→ At procedure calls, we must assume the worst case :-(

→ Constant propagation only works for local constants :-((

Question:

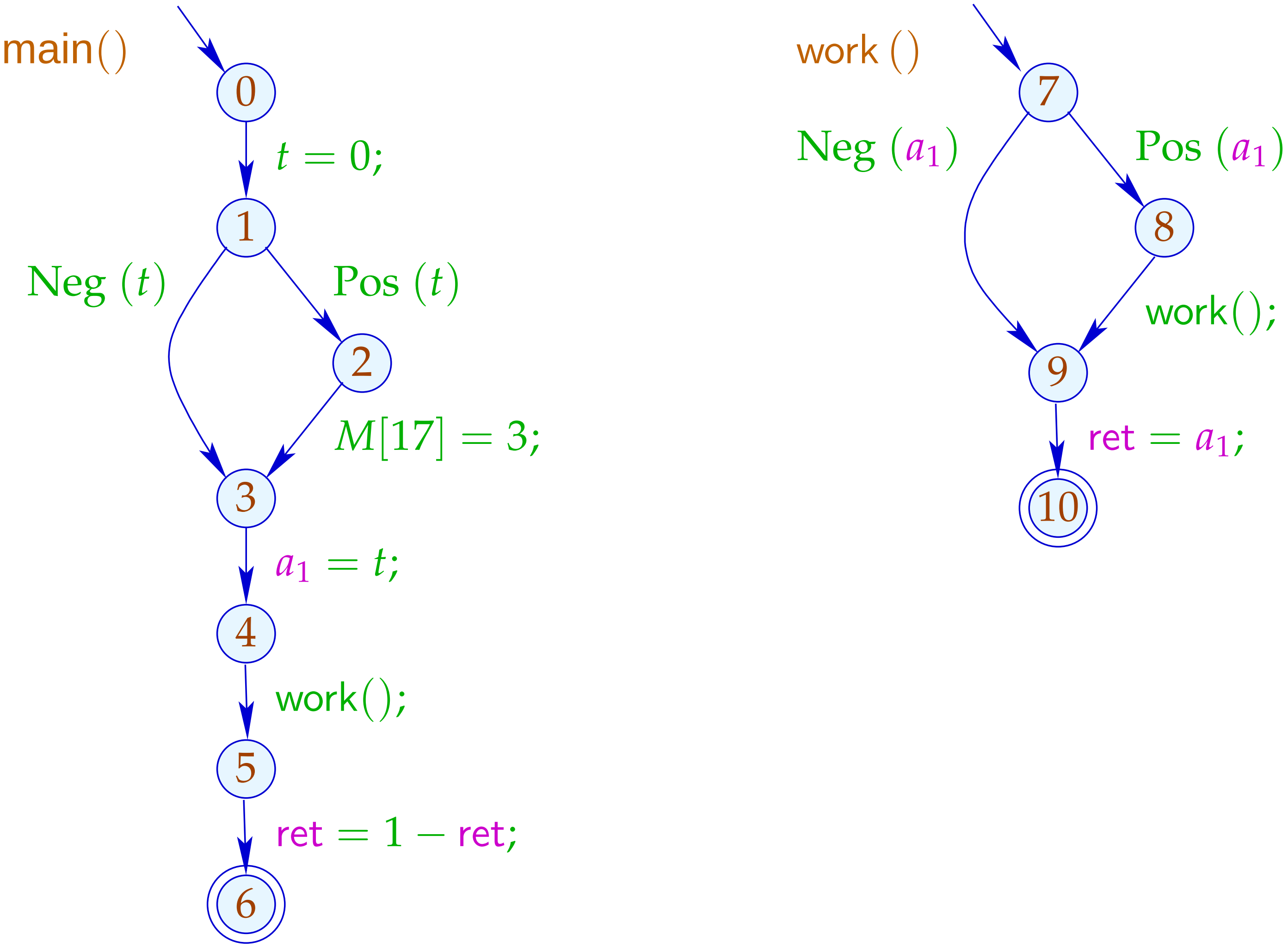
How can recursive programs be analyzed ???

Example: Constant Propagation

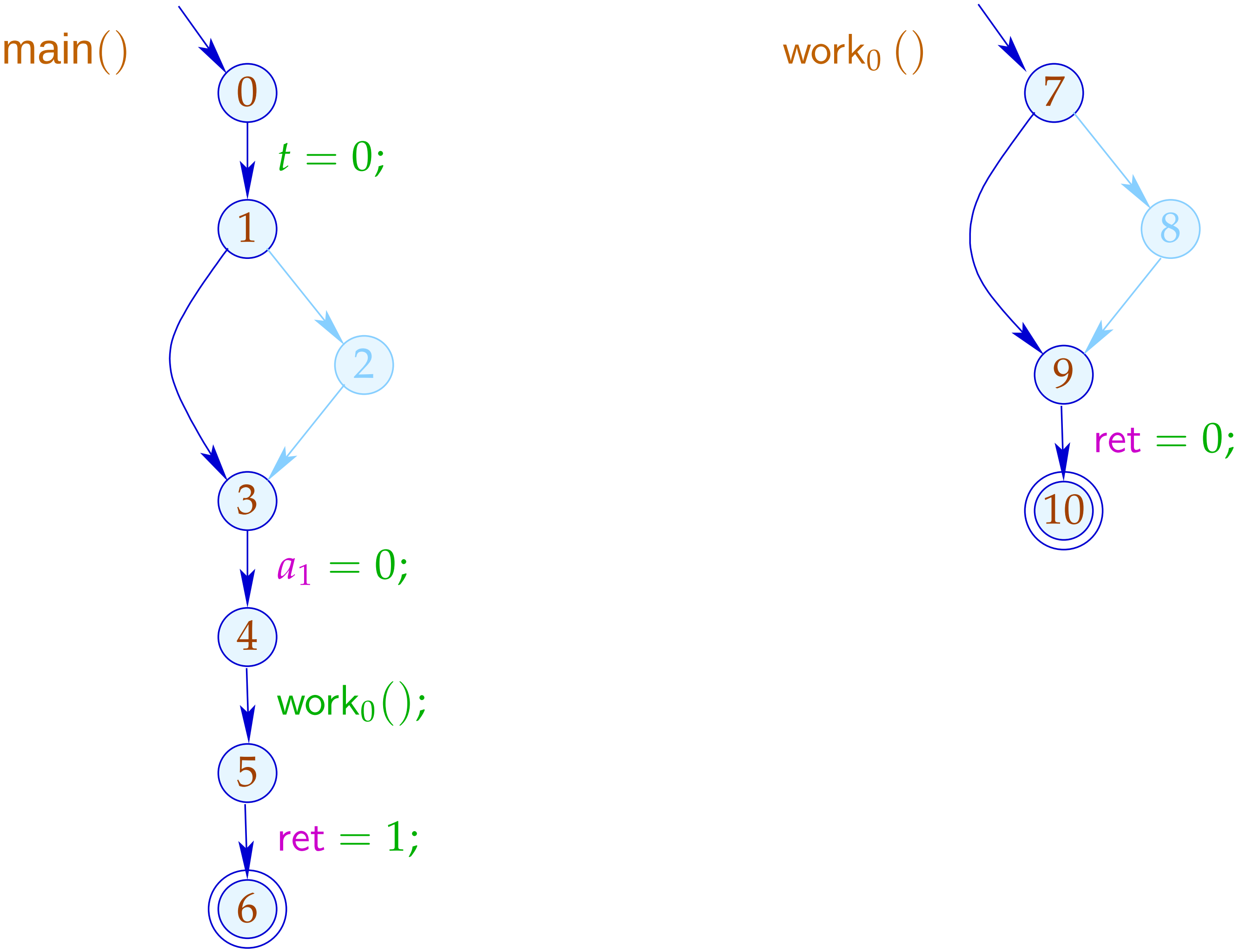
```
main() { int $t$;
        $t = 0$;
        if ($t$) $M[17] = 3$;
        $a_1 = t$;
        work ();
        ret $= 1 -$ ret;
}
```

```
work() {
        if ($a_1$) work();
        ret $= a_1$;
}
```

Example: Constant Propagation



Example: Constant Propagation



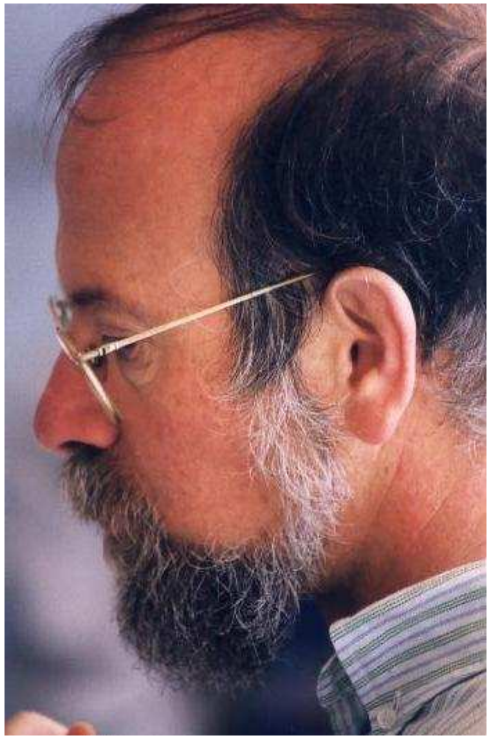
(1) Functional Approach:

Let $\mathbb{D}$ denote a complete lattice of (abstract) states.

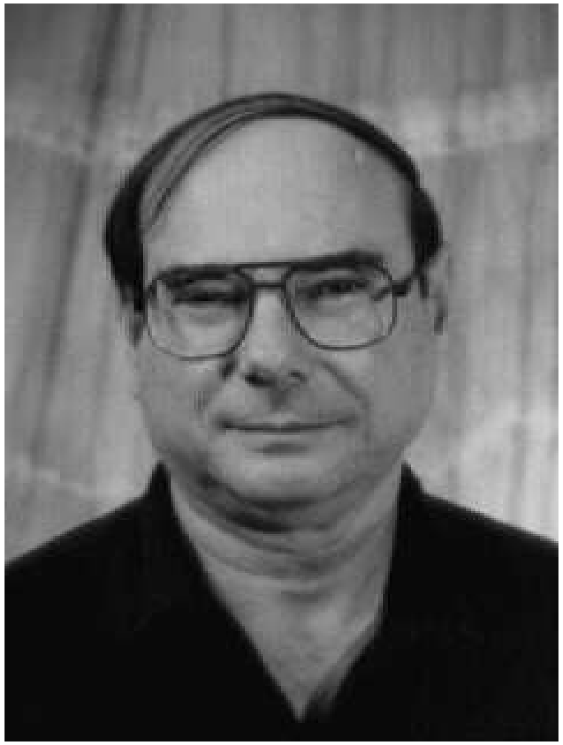
Idea:

Represent the effect of $f()$ by a function:

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



Micha Sharir, Tel Aviv University



Amir Pnueli, Weizmann Institute

In order to determine the effect of a call edge $k = (u, f\,();, v)$ we require abstract functions:

$$\begin{array}{lll} \mathsf{enter}^\sharp & : & \mathbb{D} \longrightarrow \mathbb{D} \\ \mathsf{combine}^\sharp & : & \mathbb{D}^2 \longrightarrow \mathbb{D} \end{array}$$

Then we define:

$$[\![k]\!]^\sharp\, D \quad = \quad \mathsf{combine}^\sharp\, (D, [\![f]\!]^\sharp\, (\mathsf{enter}^\sharp\, D))$$

... for Constant Propagation:

$$\mathbb{D} = (\mathit{Vars} \to \mathbb{Z}^\top)_\bot$$

$$\mathsf{enter}^\sharp\, D = \begin{cases} \bot & \text{if} \quad D = \bot \\ D|_{\mathit{Globals}} \oplus \{x \mapsto 0 \mid x \in \mathit{Locals}\} & \text{otherwise} \end{cases}$$

$$\mathsf{combine}^\sharp\, (D_1, D_2) = \begin{cases} \bot & \text{if} \quad D_1 = \bot \vee D_2 = \bot \\ D_1|_{\mathit{Locals}} \oplus D_2|_{\mathit{Globals}} & \text{otherwise} \end{cases}$$

The effects $[\![f]\!]^\sharp$ then can be determined by a system of constraints over the complete lattice $\mathbb{D} \to \mathbb{D}$:

$$\begin{array}{lllll} [\![v]\!]^\sharp & \sqsupseteq & \mathsf{Id} & v & \text{Eintrittspunkt} \\ [\![v]\!]^\sharp & \sqsupseteq & [\![k]\!]^\sharp \circ [\![u]\!]^\sharp & k = (u,\_,v) & \text{edge} \\ [\![f]\!]^\sharp & \sqsupseteq & [\![\mathit{stop}_f]\!]^\sharp & \mathit{stop}_f & \text{end point of } f \end{array}$$

$[\![v]\!]^\sharp \quad : \quad \mathbb{D} \to \mathbb{D}$ describes the effect of all prefixes of computation forests w of a procedure which lead from the entry point to v
:-)

Problems:

- How can we represent functions $f \;:\; \mathbb{D} \to \mathbb{D}$???
- If $\#\mathbb{D} = \infty$, then $\mathbb{D} \to \mathbb{D}$ has infinite strictly increasing chains :-(

Simplification: Copy-Constants

→ Conditions are interpreted as ; :-)

→ Only assignments $x = e;$ with $e \in \mathit{Vars} \cup \mathbb{Z}$ are treated exactly :-)

Observation:

→ The effects of assignments are:

$$[\![x = e;]\!]^\sharp\, D \;=\; \begin{cases} D \oplus \{x \mapsto c\} & \text{if} \quad e = c \in \mathbb{Z} \\ D \oplus \{x \mapsto (D\; y)\} & \text{if} \quad e = y \in \mathit{Vars} \\ D \oplus \{x \mapsto \top\} & \text{otherwise} \end{cases}$$

→ Let $\mathbb{V}$ denote the (finite !!!) set of constant right-hand sides. Then variables may only take values from $\mathbb{V}^\top$:-))

→ The occurring effects can be taken from

$$\mathbb{D}_f \to \mathbb{D}_f \qquad \text{with} \qquad \mathbb{D}_f = (\mathit{Vars} \to \mathbb{V}^\top)_\perp$$

→ The complete lattice is huge, but finite !!!

Improvement:

→ Not all functions from $\mathbb{D}_f \to \mathbb{D}_f$ will occur :-)

→ All occurring functions $\lambda D.\bot \;\neq\; M$ are of the form:

$$\begin{array}{lcll} M & = & \{x \mapsto (b_x \sqcup \bigsqcup_{y\in I_x} y) \mid x \in \mathit{Vars}\} & \text{where:} \\ M\;D & = & \{x \mapsto (b_x \sqcup \bigsqcup_{y\in I_x} D\;y) \mid x \in \mathit{Vars}\} & \text{für} \quad D \neq \bot \end{array}$$

→ Let $\mathbb{M}$ denote the set of all these functions. Then for $M_1, M_2 \in \mathbb{M}$ $\quad (M_1 \neq \lambda\, D.\,\bot \neq M_2):$

$$(M_1 \sqcup M_2)\; x \quad = \quad (M_1\; x) \sqcup (M_2\; x)$$

→ For $k = \#\mathit{Vars}$, $\mathbb{M}$ has height $\mathcal{O}(k^2)$:-)

Improvement (Cont.):

→ Also, composition can be directly implemented:

$$\begin{array}{rcll} (M_1 \circ M_2)\ x & = & b' \sqcup \bigsqcup_{y \in I'} y & \qquad \text{with} \\ b' & = & b \sqcup \bigsqcup_{z \in I} b_z & \\ I' & = & \bigcup_{z \in I} I_z & \quad \text{where} \\ M_1\ x & = & b \sqcup \bigsqcup_{y \in I} y & \\ M_2\ z & = & b_z \sqcup \bigsqcup_{y \in I_z} y & \end{array}$$

→ The effects of assignments then are:

$$[\![x = e;]\!]^\sharp \quad = \quad \begin{cases} \mathsf{Id}_{Vars} \oplus \{x \mapsto c\} & \text{if} \quad e = c \in \mathbb{Z} \\ \mathsf{Id}_{Vars} \oplus \{x \mapsto y\} & \text{if} \quad e = y \in Vars \\ \mathsf{Id}_{Vars} \oplus \{x \mapsto \top\} & \text{otherwise} \end{cases}$$

... in the Example:

$$\begin{array}{lll} [\![t = 0;]\!]^\sharp & = & \{a_1 \mapsto a_1, \mathsf{ret} \mapsto \mathsf{ret}, \boxed{t \mapsto 0}\} \\ [\![a_1 = t;]\!]^\sharp & = & \{\boxed{a_1 \mapsto t}, \mathsf{ret} \mapsto \mathsf{ret}, t \mapsto t\} \end{array}$$

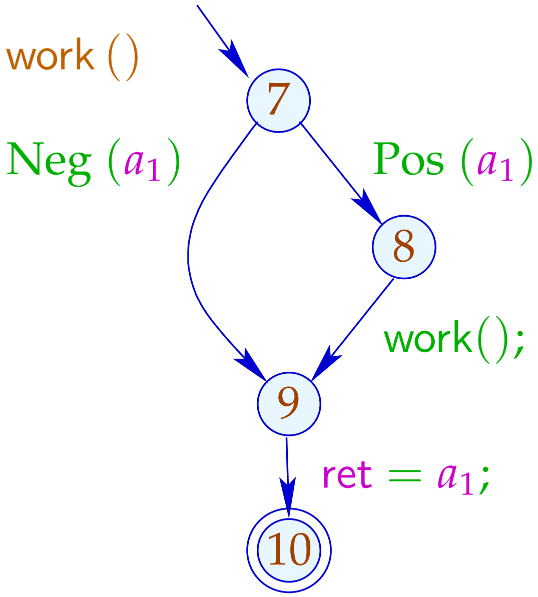
In order to implement the analysis, we additionally must construct the effect of a call $k = (\_, f\,();, \_)$ from the effect of a procedure f :

$$\begin{array}{lll} [\![k]\!]^\sharp & = & H\,([\![f]\!]^\sharp) \qquad\qquad \text{where:} \\ H\,(M) & = & \mathsf{Id}|_{Locals} \oplus \{\mathsf{x} \mapsto (M \circ \mathsf{enter}^\sharp)|_{Globals} \\ \mathsf{enter}^\sharp\ x & = & \left\{ \begin{array}{ll} x & \text{if} \quad x \in Globals \\ 0 & \text{otherwise} \end{array} \right. \end{array}$$

... in the Example:

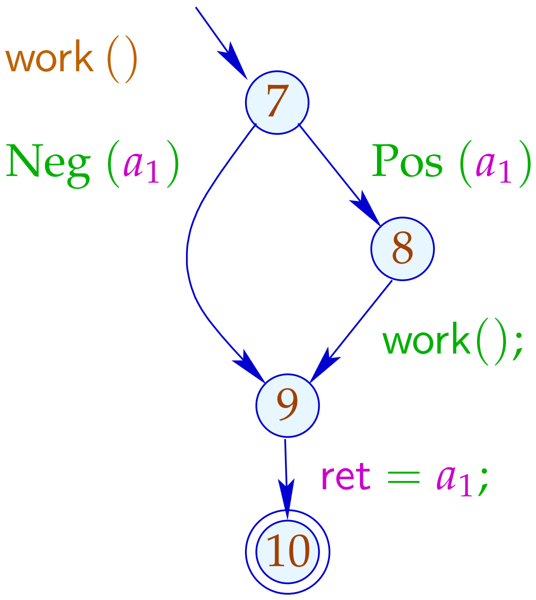
$$\begin{array}{llcl} \text{If} & [\![\mathsf{work}]\!]^\sharp & = & \{a_1 \mapsto a_1, \mathsf{ret} \mapsto a_1, t \mapsto t\} \\ \text{then} & H\,[\![\mathsf{work}]\!]^\sharp & = & \mathsf{Id}_{\{t\}} \oplus \{a_1 \mapsto a_1, \mathsf{ret} \mapsto a_1\} \\ & & = & \{a_1 \mapsto a_1, \mathsf{ret} \mapsto a_1, t \mapsto t\} \end{array}$$

Now we can perform fixpoint iteration :-)



	1
7	$\{a_1 \mapsto a_1, \text{ret} \mapsto \text{ret}, t \mapsto t\}$
9	$\{a_1 \mapsto a_1, \text{ret} \mapsto \text{ret}, t \mapsto t\}$
10	$\{a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t\}$
8	$\{a_1 \mapsto a_1, \text{ret} \mapsto \text{ret}, t \mapsto t\}$

$$\begin{aligned} [\![(8,\ldots,9)]\!]^\sharp \circ [\![8]\!]^\sharp &= \{a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t\} \circ \\ &\quad \{a_1 \mapsto a_1, \text{ret} \mapsto \text{ret}, t \mapsto t\} \\ &= \{a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t\} \end{aligned}$$



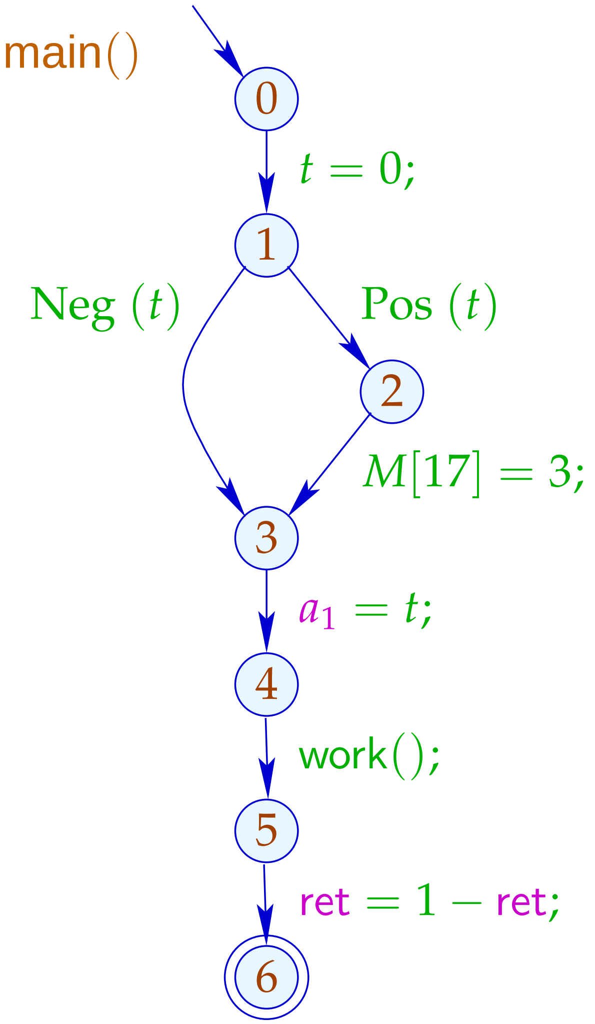
	2
7	$\{a_1 \mapsto a_1, \mathsf{ret} \mapsto \mathsf{ret}, t \mapsto t\}$
9	$\boxed{\{a_1 \mapsto a_1, \mathsf{ret} \mapsto a_1 \sqcup \mathsf{ret}, t \mapsto t\}}$
10	$\boxed{\{a_1 \mapsto a_1, \mathsf{ret} \mapsto a_1, t \mapsto t\}}$
8	$\{a_1 \mapsto a_1, \mathsf{ret} \mapsto \mathsf{ret}, t \mapsto t\}$

$$\begin{aligned}[\![(8,\ldots,9)]\!]^\sharp \circ [\![8]\!]^\sharp &= \{a_1 \mapsto a_1, \mathsf{ret} \mapsto a_1, t \mapsto t\} \;\circ \\ &\quad\ \{a_1 \mapsto a_1, \mathsf{ret} \mapsto \mathsf{ret}, t \mapsto t\} \\ &= \{a_1 \mapsto a_1, \mathsf{ret} \mapsto a_1, t \mapsto t\}\end{aligned}$$

If we know the effects of procedure calls, we can put up a constraint system for determining the abstract state when reaching a program point:

$$\begin{array}{llll}
\mathcal{R}[\mathsf{main}] & \sqsupseteq & \mathsf{enter}^\sharp\; d_0 & \\
\mathcal{R}[f] & \sqsupseteq & \mathsf{enter}^\sharp\; (\mathcal{R}[u]) & k = (u, f\,();, \_) \quad \text{call} \\
\mathcal{R}[v] & \sqsupseteq & \mathcal{R}[f] & v \quad \text{entry point of} \quad f \\
\mathcal{R}[v] & \sqsupseteq & [\![k]\!]^\sharp\; (\mathcal{R}[u]) & k = (u, \_, v) \quad \text{edge}
\end{array}$$

... in the Example:



0	$\{a_1 \mapsto \top, \mathsf{ret} \mapsto \top, t \mapsto 0\}$
1	$\{a_1 \mapsto \top, \mathsf{ret} \mapsto \top, t \mapsto 0\}$
2	$\{a_1 \mapsto \top, \mathsf{ret} \mapsto \top, t \mapsto 0\}$
3	$\{a_1 \mapsto \top, \mathsf{ret} \mapsto \top, t \mapsto 0\}$
4	$\{a_1 \mapsto 0, \mathsf{ret} \mapsto \top, t \mapsto 0\}$
5	$\{a_1 \mapsto 0, \mathsf{ret} \mapsto 0, t \mapsto 0\}$
6	$\{a_1 \mapsto 0, \mathsf{ret} \mapsto \top, t \mapsto 0\}$

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 $\mathbb{M} \subseteq \mathbb{D} \to \mathbb{D}$ 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:

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

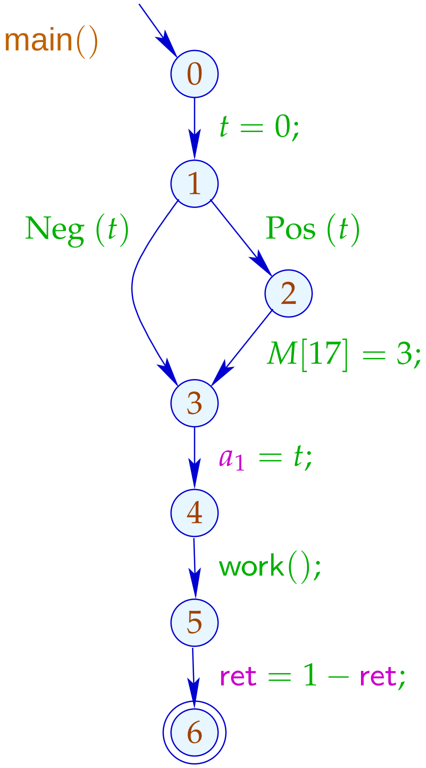
// $[\![v,a]\!]^\sharp$ $==$ value for the argument a .

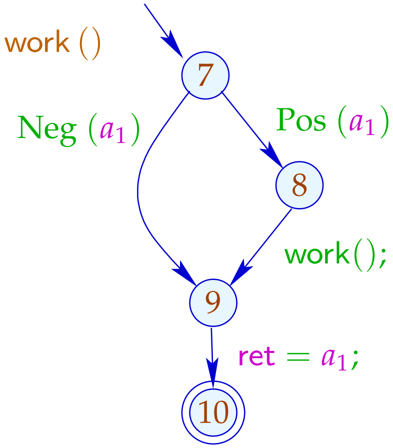
Discussion:

- 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 $[\![\text{main}(), a_0]\!]^\sharp$ $\Longrightarrow$ 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	$\top$	$\top$	$\top$	$\top$
1	$\top$	$\top$	$\top$	$\top$
2	$\top$	$\top$	$\bot$	
3	$\top$	$\top$	$\top$	$\top$
4	$\top$	$\top$	0	$\top$
7	0	$\top$	0	$\top$
8	0	$\top$	$\bot$	
9	0	$\top$	0	$\top$
10	0	$\top$	0	0
5	$\top$	$\top$	0	0
main()	$\top$	$\top$	0	1

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!

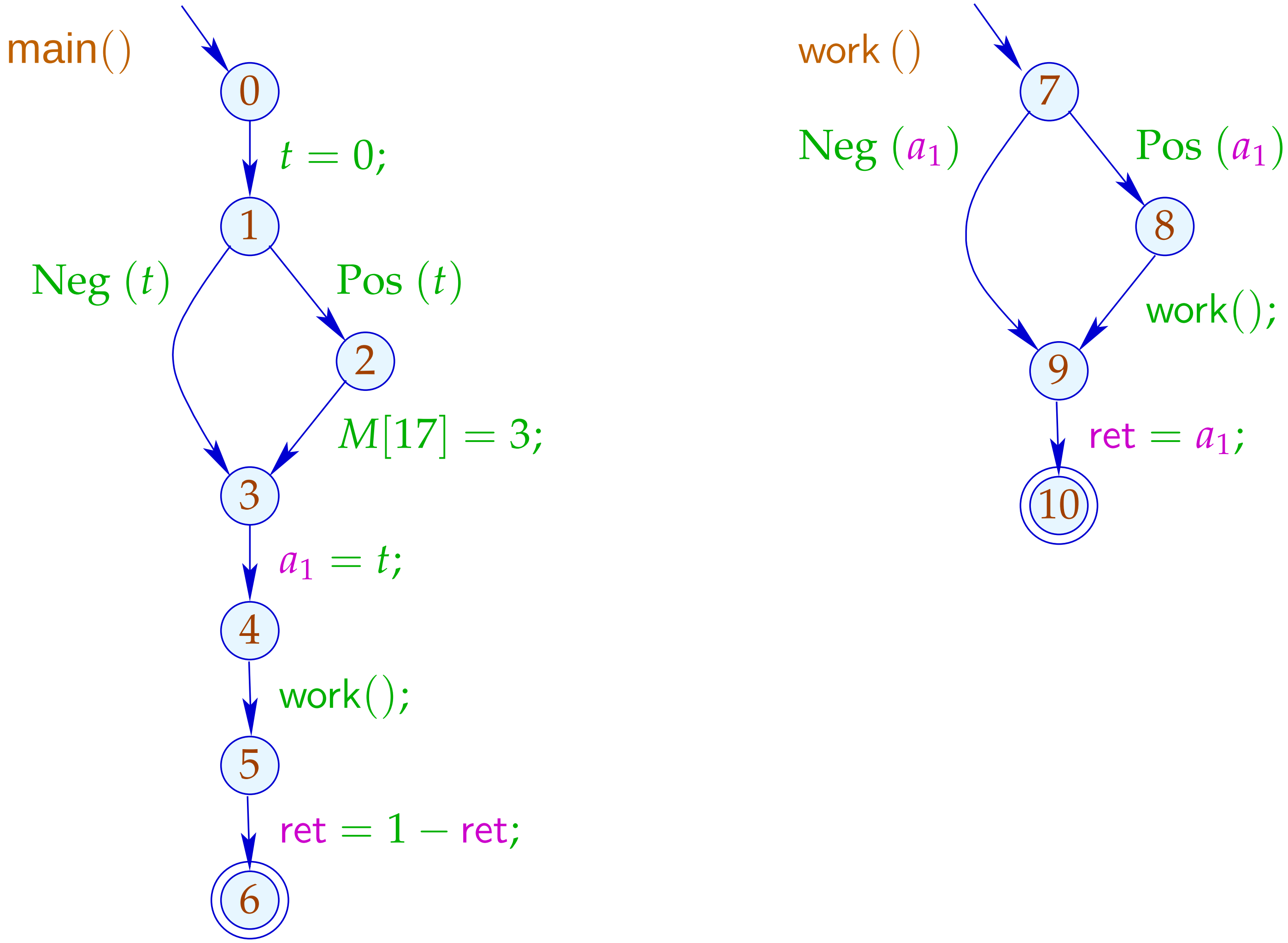
→ In general, this is infinite :-(

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

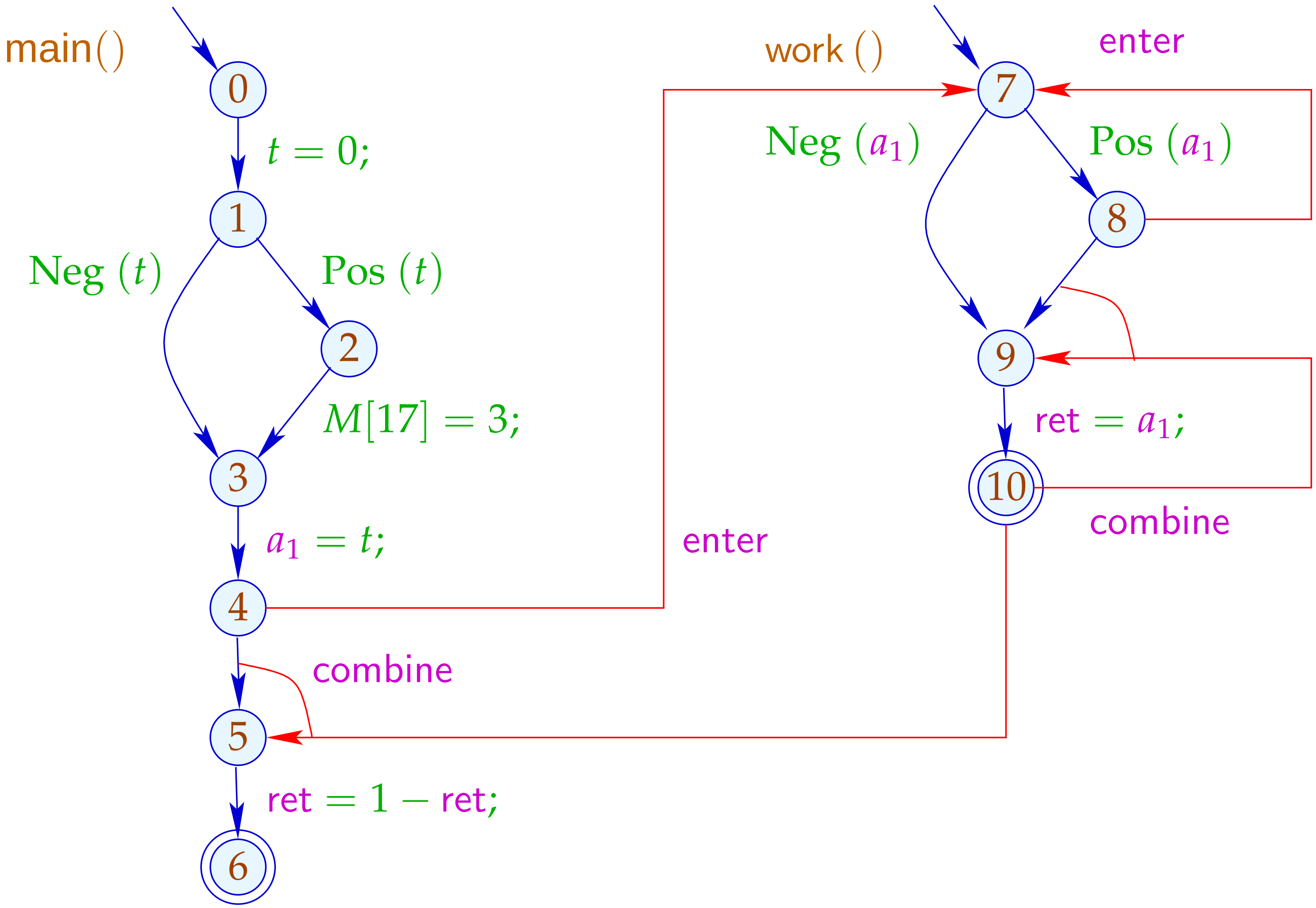
→ Important special case: $d = 0$.

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

... in the Example:



... in the Example:



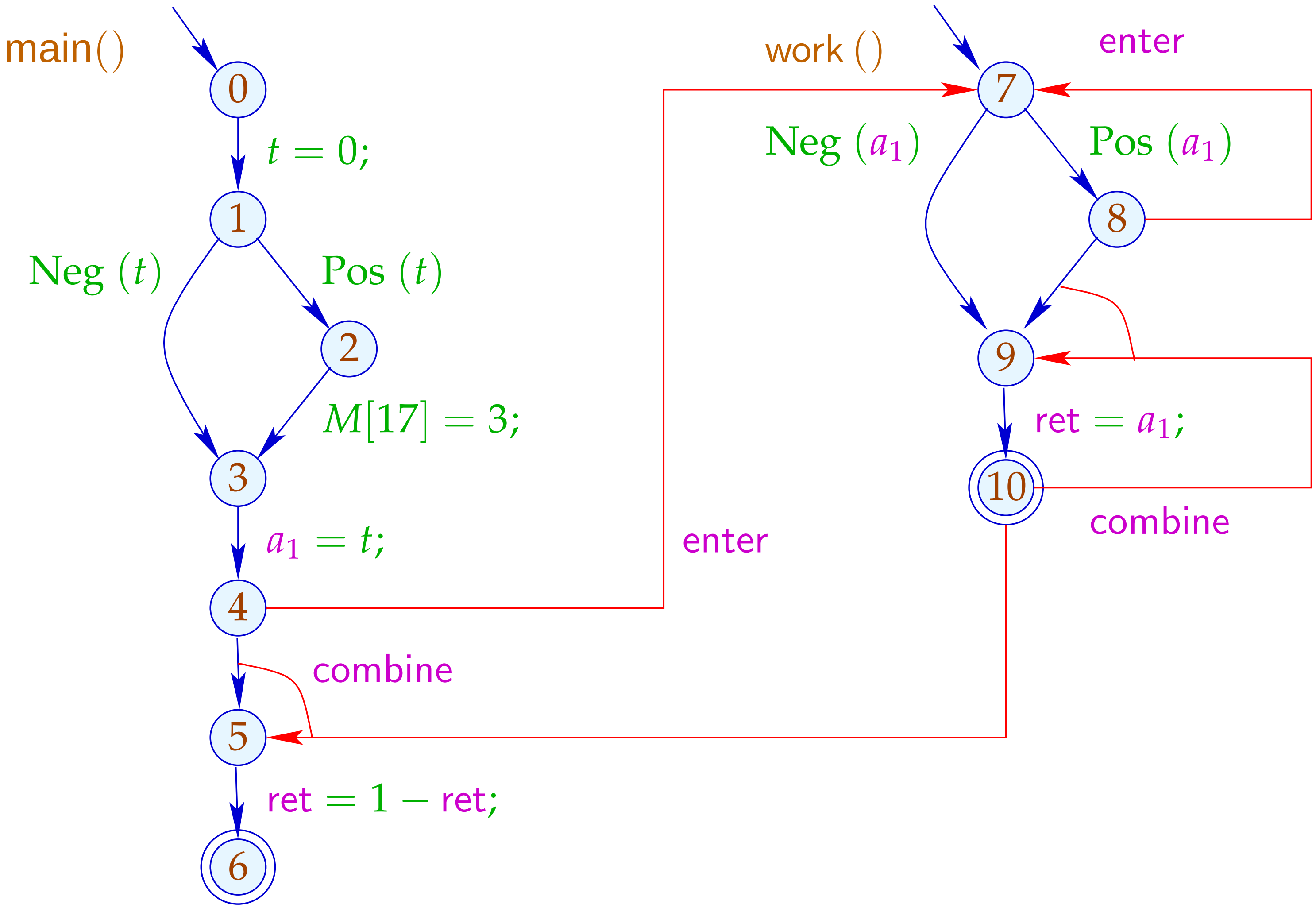
The conditions for $5, 7, 10$, e.g., are:

$$\begin{array}{lll} \mathcal{R}[5] & \sqsupseteq & \textsf{combine}^\sharp\,(\mathcal{R}[4], \mathcal{R}[10]) \\[1ex] \mathcal{R}[7] & \sqsupseteq & \textsf{enter}^\sharp\,(\mathcal{R}[4]) \\ \mathcal{R}[7] & \sqsupseteq & \textsf{enter}^\sharp\,(\mathcal{R}[8]) \\[1ex] \mathcal{R}[9] & \sqsupseteq & \textsf{combine}^\sharp\,(\mathcal{R}[8], \mathcal{R}[10]) \end{array}$$

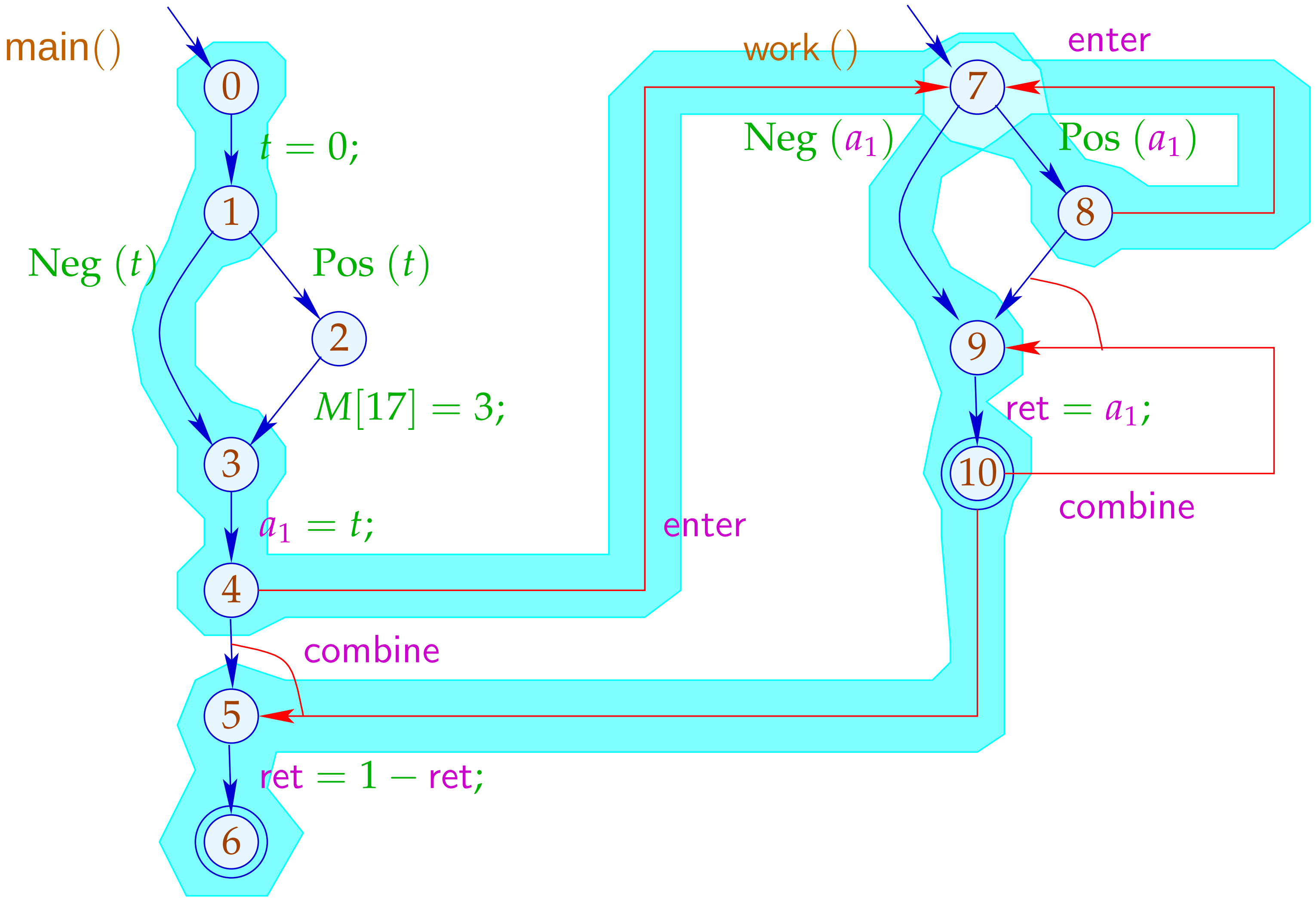
Warning:

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 :-)

3 Exploiting Hardware Features

Question: How can we optimally use:

... Registers

... Pipelines

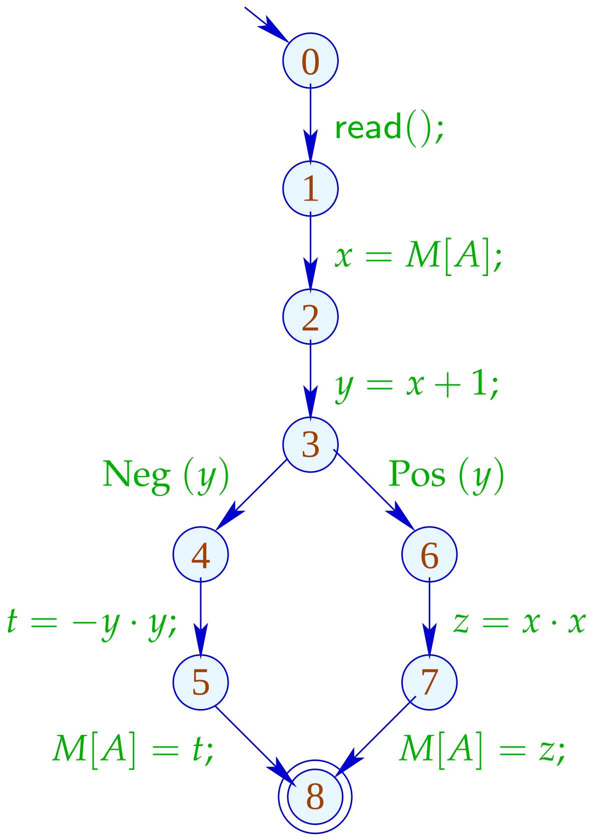
... Caches

... Processors ???

3.1 Registers

Example:

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



The program uses 5 variables ...

Problem:

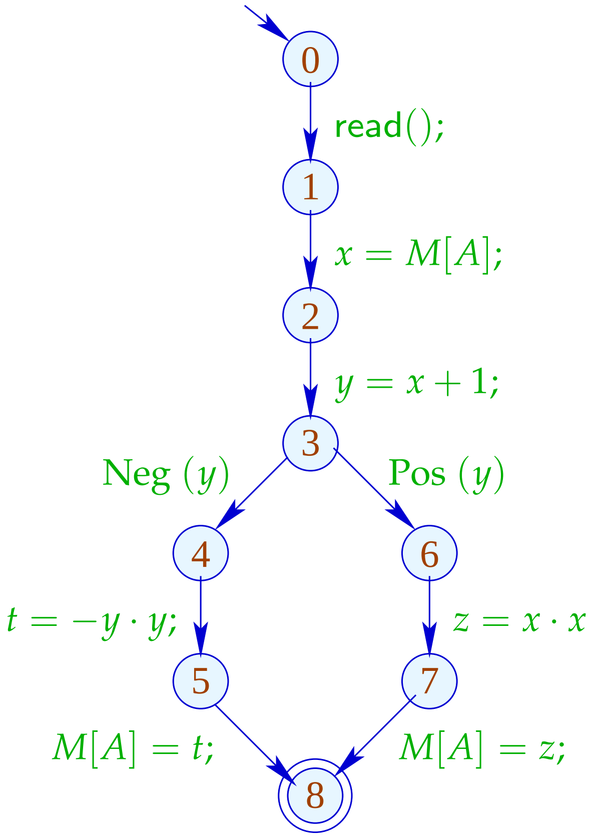
What if the program uses more variables than there are registers :-(

Idea:

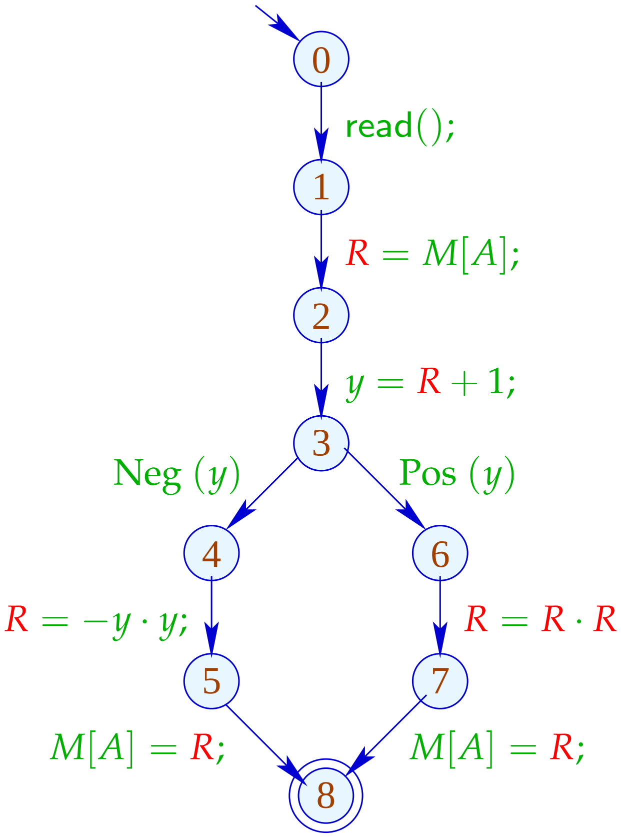
Use one register for several variables :-)

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

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



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



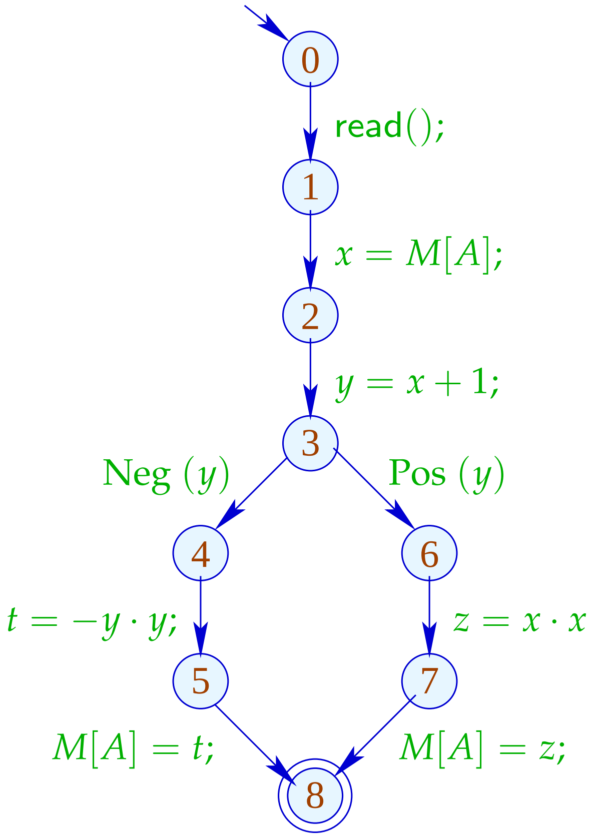
Warning:

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

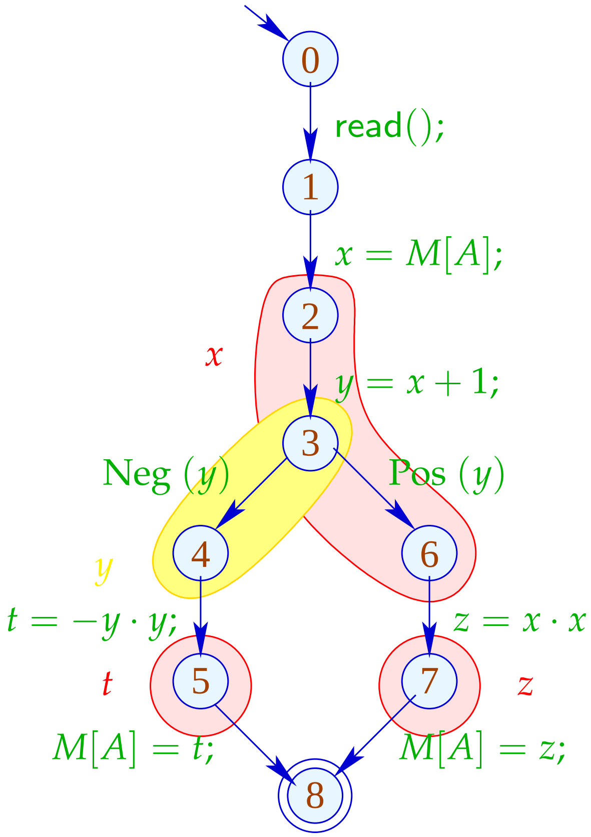
The (true) live range of x is defined by:

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

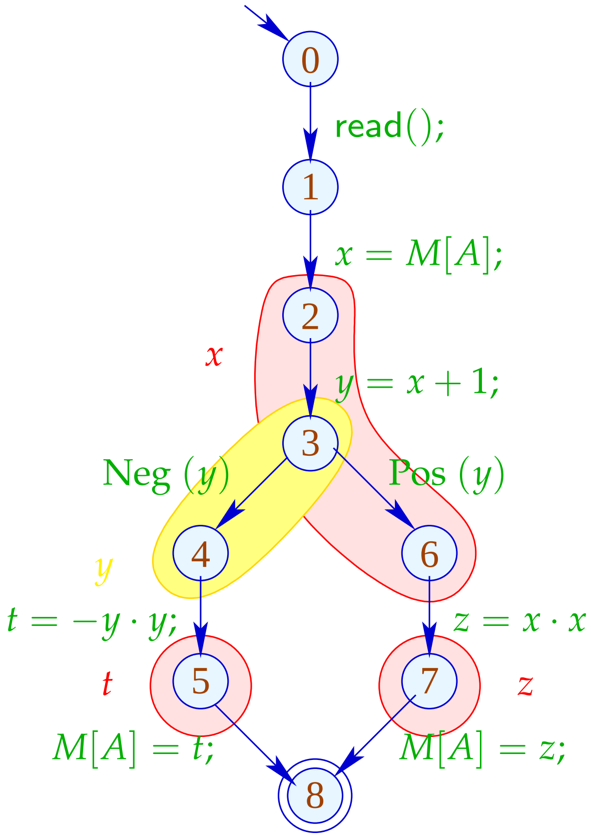
... in the Example:



	$\mathcal{L}$
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	$\emptyset$



Live Ranges:

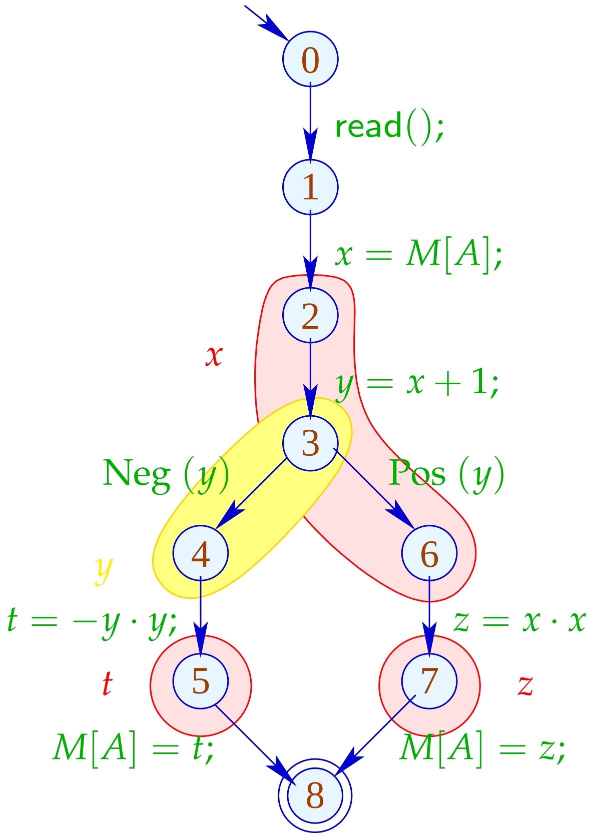
A	$\{1, \ldots, 7\}$
x	$\{2, 3, 6\}$
y	$\{2, 4\}$
t	$\{5\}$
z	$\{7\}$

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

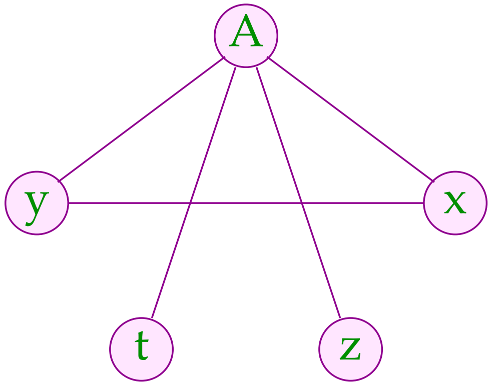
$$E_I = \{\{x, y\} \mid x \neq y, \mathcal{L}[x] \cap \mathcal{L}[y] \neq \emptyset\}$$

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

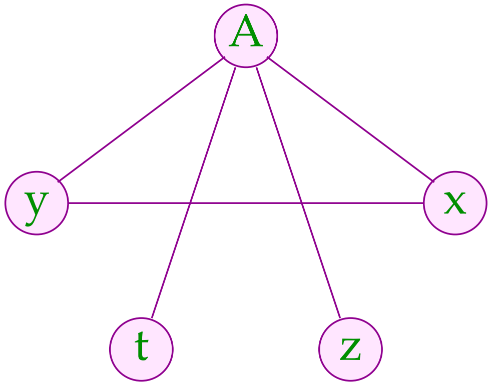
... in the Example:



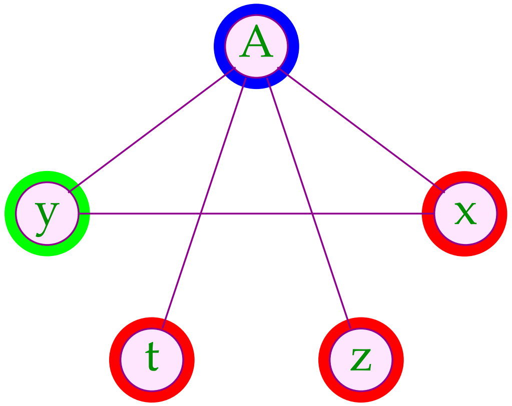
Interference Graph:



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

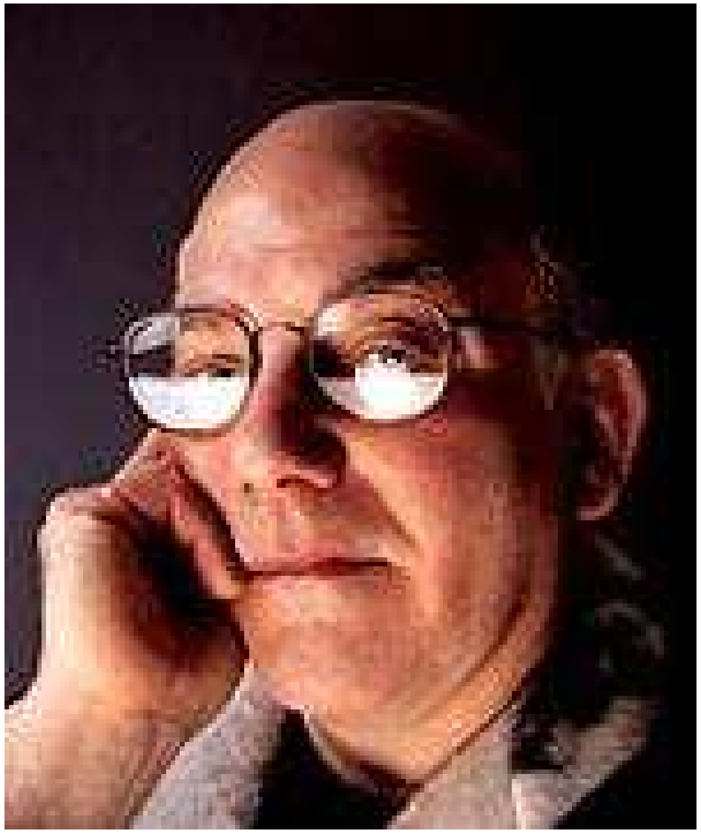


Variables which are not connected with an edge can be assigned to the same register :-)



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 :-)

→ ... Warning: differen variants have beenpatented !!!

Discussion:

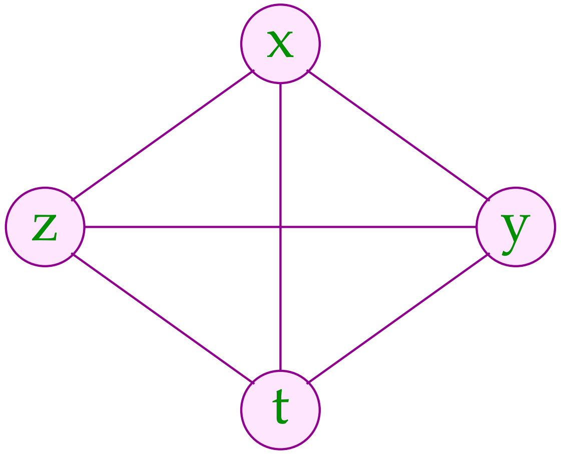
- → Essentially, this is a Pre-order DFS :-)
- → In theory, the result may arbitrarily far from the optimum :-(
- → ... in practice, it may not be as bad :-)
- → ... Warning: differen variants have beenpatented !!!

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

Idea: Life Range Splitting

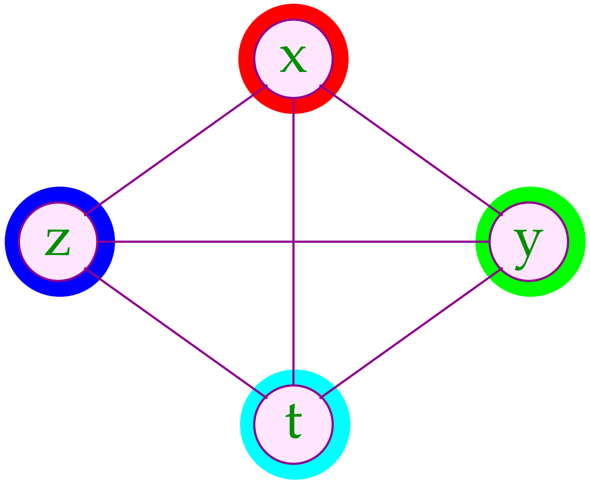
Special Case: Basic Blocks

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



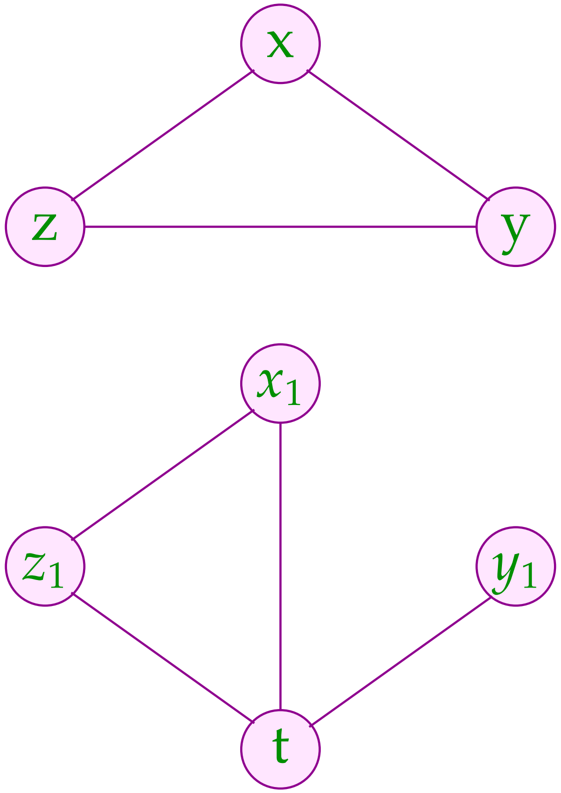
Special Case: Basic Blocks

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



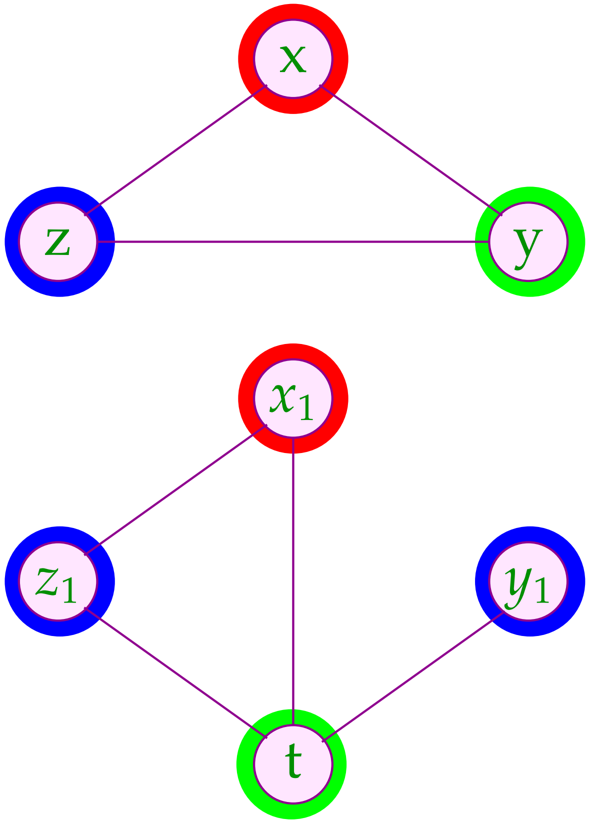
The live ranges of x and z can be split:

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

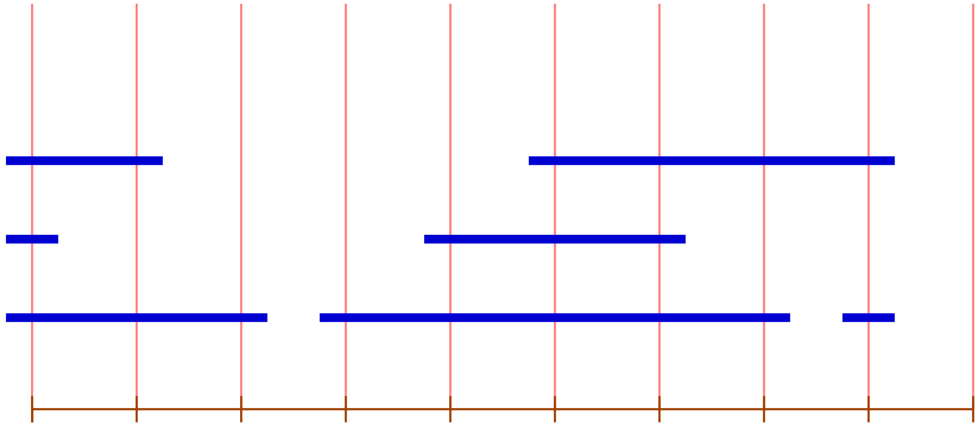


The live ranges of x and z can be split:

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



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:

$$
\begin{aligned}
&\text{maximal covering number} \\
&\qquad\qquad = \text{size of the maximal clique} \\
&\qquad\qquad = \text{maximally necessary number of colors} \quad \text{:-)}
\end{aligned}
$$

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$[i] = (I ::$ 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 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.

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.

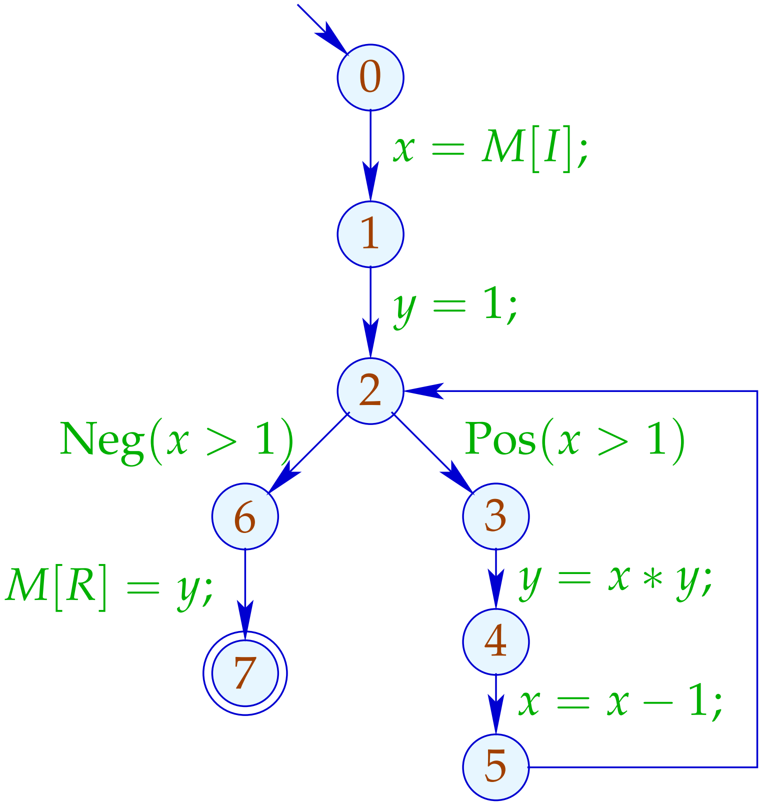
Step 2:

- Introduce a separate variant x_i for every ocurrence 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 $x = x;$ at the end of each incoming edge.

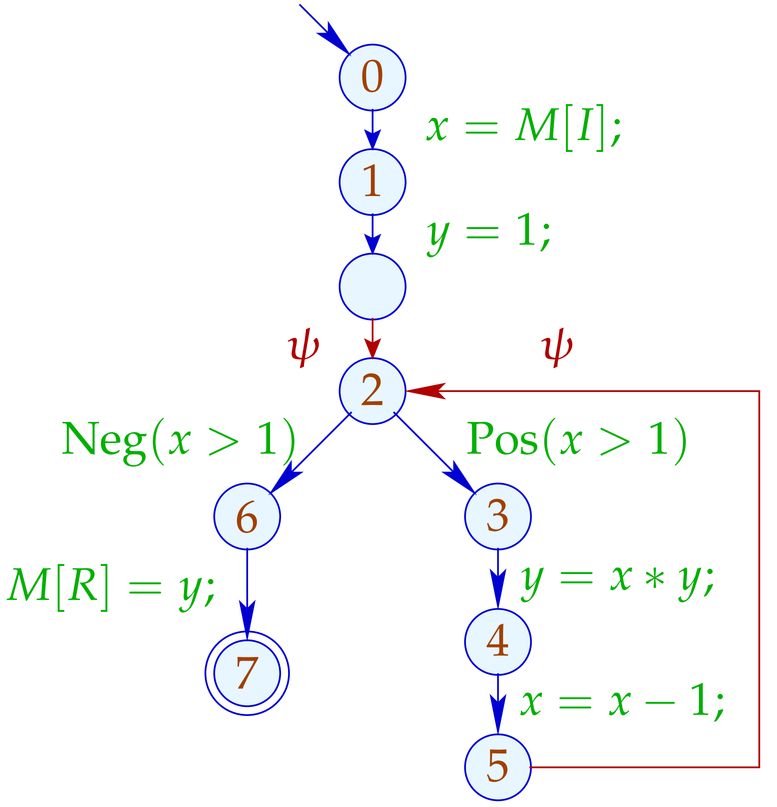
Example



Reaching 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



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

Reaching 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	$\langle x, 1\rangle, \langle x, 5\rangle, \langle y, 2\rangle, \langle y, 4\rangle$

Reaching Definitions

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

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

where

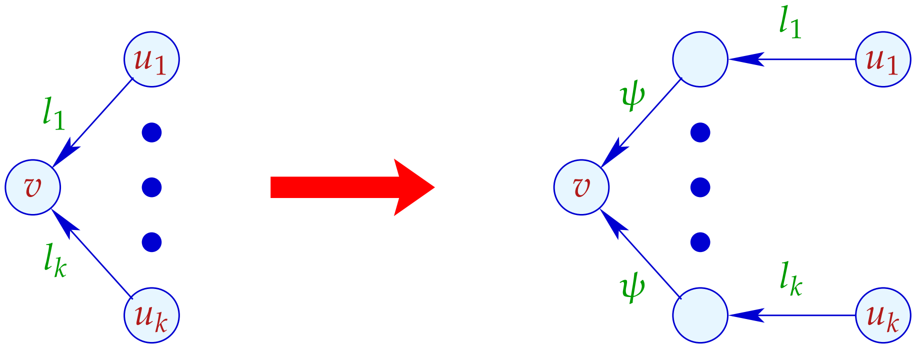
$$Defs = Vars \times Nodes \qquad Defs(x) = \{x\} \times Nodes$$

Then:

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

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

The Transformation SSA, Step 1:



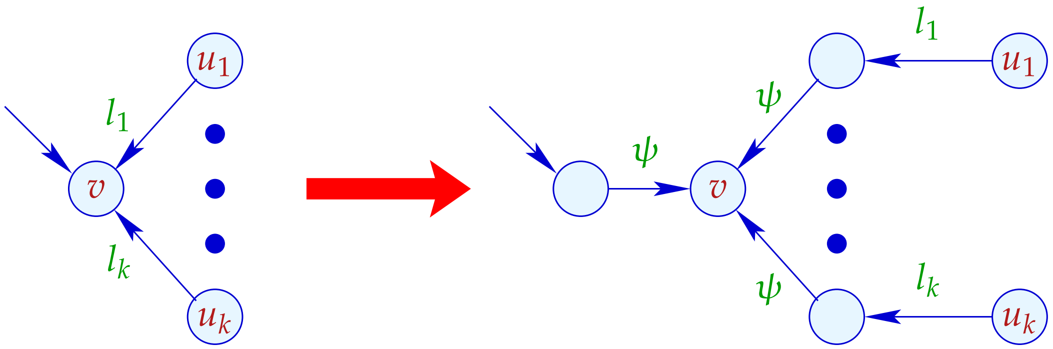
where $k \geq 2$.

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.

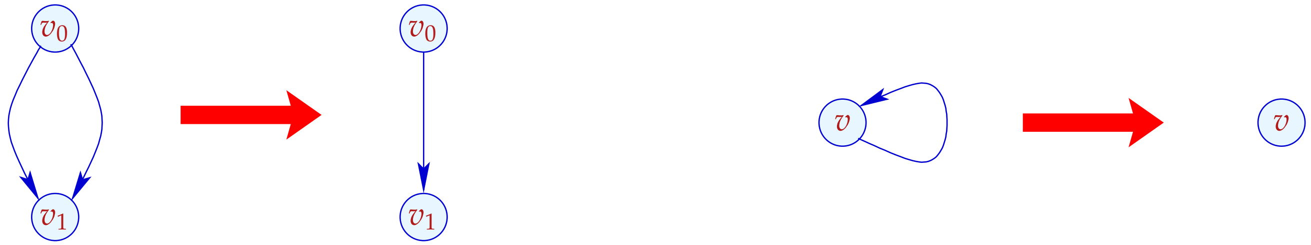
Discussion

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 !

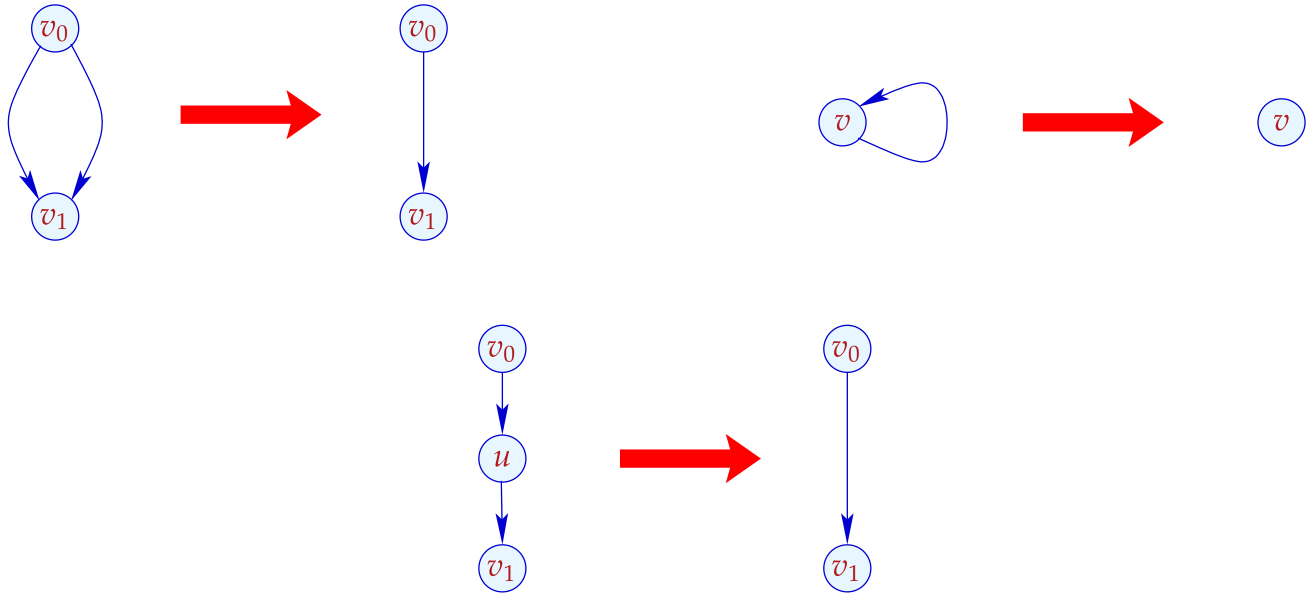
A well-structured cfg can be reduced to a single vertex or edge by:



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 (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:

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

Remark

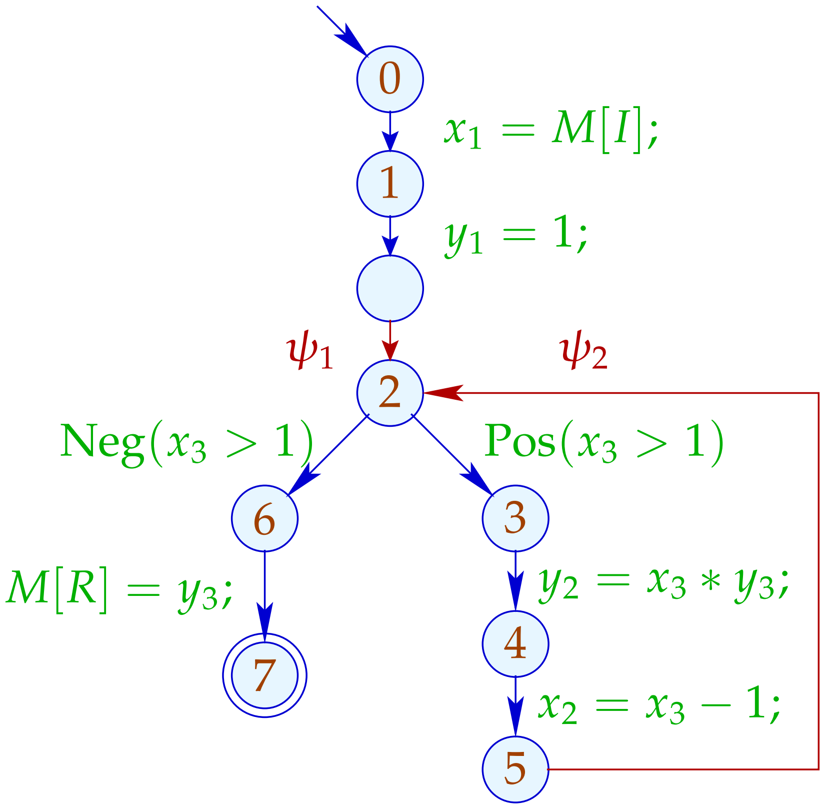
The multiple assignments:

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

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

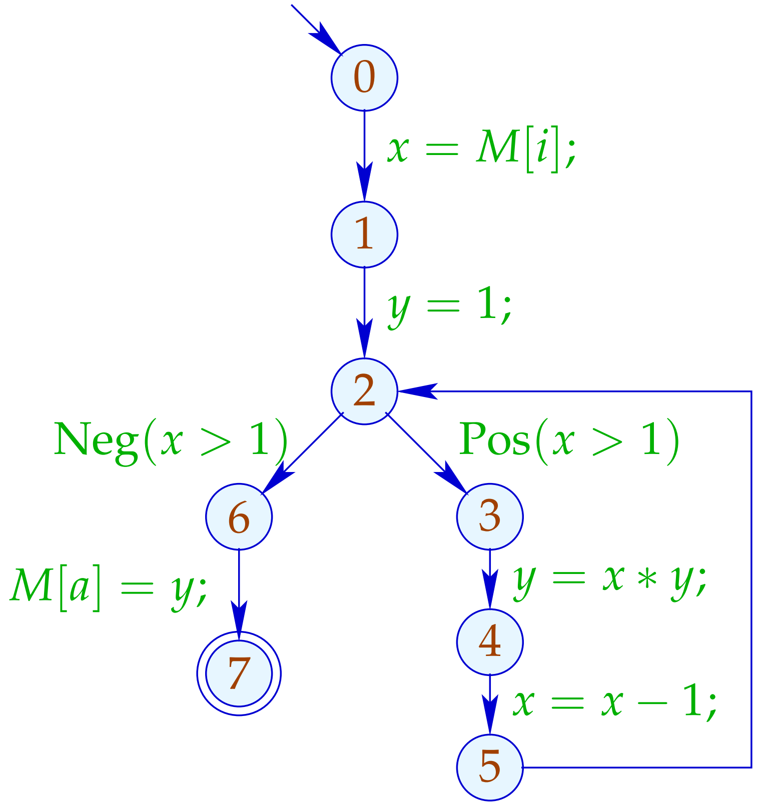
- By the theorem, the number λ of required registers can be easily computed :-)
- Thus variables which are to be spilled to memory, can be determined ahead of the subsequent assignment of registers !
- Thus here, we may, e.g., insist on keeping iteration variables from inner loops.
- Clearly, always $\lambda \leq \omega(G) \leq \chi(G)$:-)

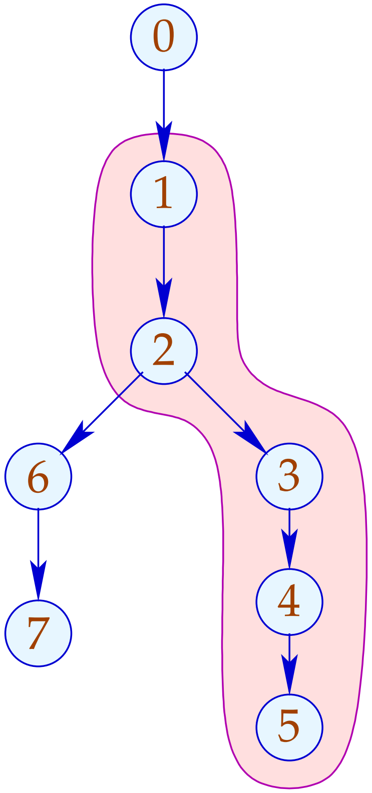
  Therefore, it suffices to color the interference graph with λ colors.
- Instead, we provide an algorithm which directly operates on the cfg ...

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





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 ...

Proof of the Intersection Property

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

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

(2) Let C denote a clique of tree fragments.
Then there is an enumeration $C = \{I_1, \ldots, I_r\}$ with roots v_1,...,v_r such that

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

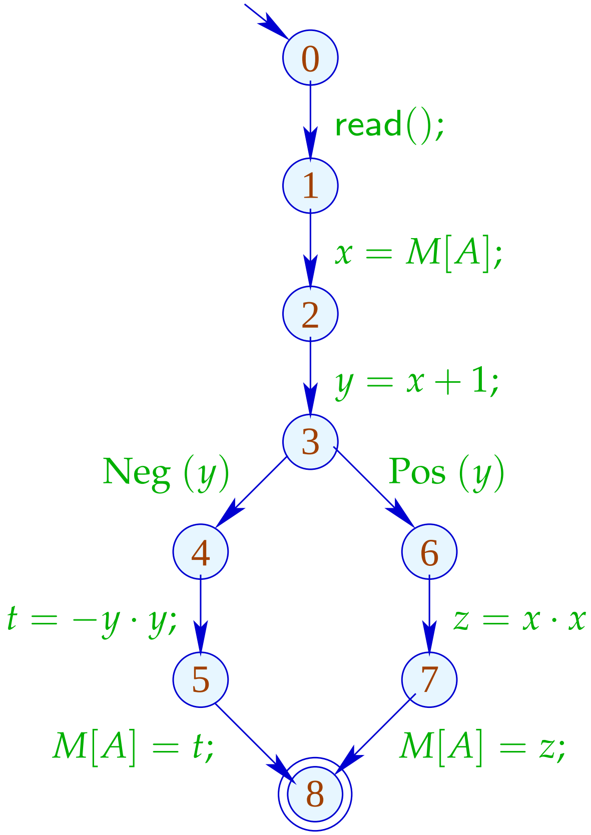
In particular, $v_r \in I_i$ for all i. :-)

The Greedy Algorithm

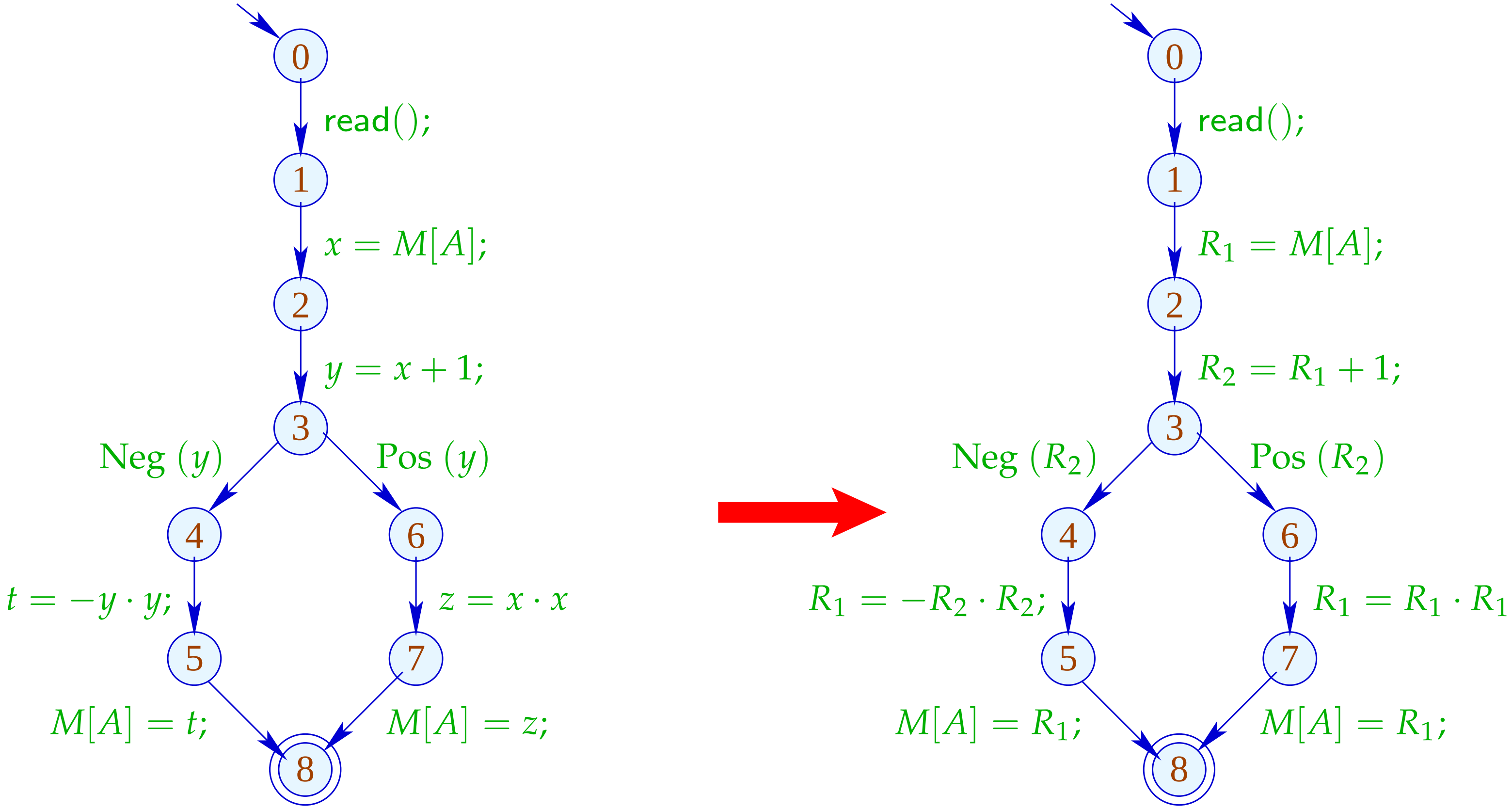
```
**forall** $(u \in \mathit{Nodes})$ $\mathit{visited}[u] = \textbf{false};$
**forall** $(x \in \mathcal{L}[\mathit{start}])$ $\Gamma(x) = \mathsf{extract}(\mathit{free});$
$\mathsf{alloc}(\mathit{start});$

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

Example



Example


0
read();
1
$x = M[A];$
2
$y = x + 1;$
3
Neg (y)
Pos (y)
4
6
$t = -y \cdot y;$
$z = x \cdot x$
5
7
$M[A] = t;$
$M[A] = z;$
8
0
read();
1
$R_1 = M[A];$
2
$R_2 = R_1 + 1;$
3
Neg (R_2)
Pos (R_2)
4
6
$R_1 = -R_2 \cdot R_2;$
$R_1 = R_1 \cdot R_1$
5
7
$M[A] = R_1;$
$M[A] = R_1;$
8

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:

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

(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:

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

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:

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

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 $k > 2$??

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

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

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:

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

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:

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

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 $\Longrightarrow$ reduction of life ranges :-)

3.2 Instruction Level Parallelism

Modern processors do not execute one instruction after the other strictly sequentially.

Here, we consider two approaches:

(1) VLIW (Very Large Instruction Words)
(2) Pipelining

VLIW:

One instruction simultaneously executes up to k (e.g., 4:-) elementary Instructions.

Pipelining:

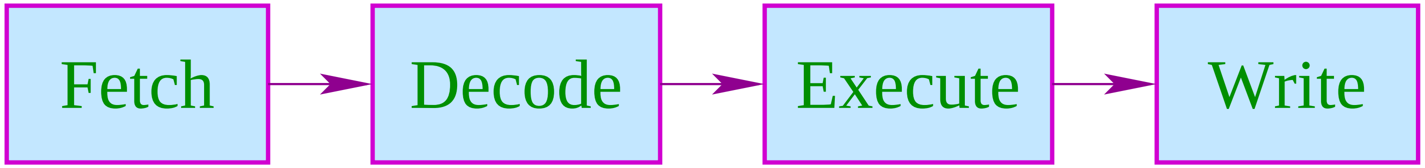
Instruction execution may overlap.

Example:

$$w = (R_1 = R_2 + R_3 \mid D = D_1 * D_2 \mid R_3 = M[R_4])$$

Warning:

- Instructions occupy hardware ressources.
- Instructions may access the same busses/registers ⟹ hazards
- Results of an instruction may be available only after some delay.
- During execution, different parts of the hardware are involved:



- During Execute and Write different internal registers/busses/alus may be used.

We conclude:

Distributing the instruction sequence into sequences of words is amenable to various constraints ...

In the following, we ignore the phases Fetch und Decode :-)

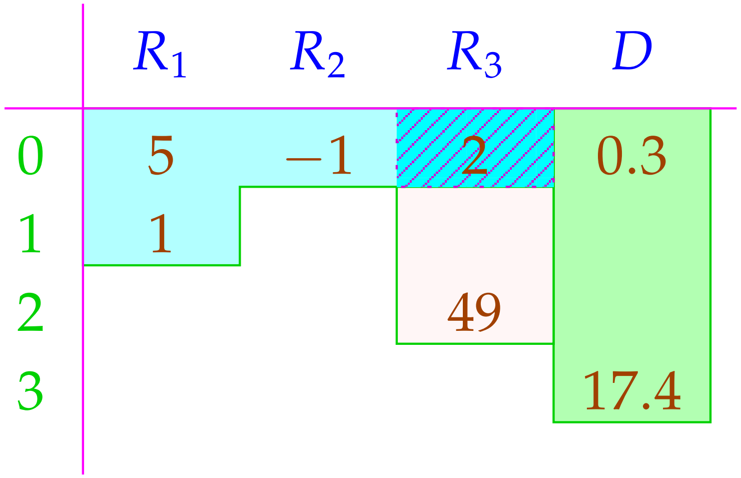
Examples for Constraints:

(1) at most one load/store per word;
(2) at most one jump;
(3) at most one write into the same register.

Example Timing:

Floating-point Operation	3
Load/Store	2
Integer Arithmetic	1

Timing Diagram:



	R_1	R_2	R_3	D
0	5	-1	2	0.3
1	1			
2			49	
3				17.4

R_3 is over-written, after the addition has fetched 2 :-)

If a register is accessed simultaneously (here: R_3), a strategy of conflict solving is required ...

Conflicts:

Read-Read: A register is simulatneously read.

$\Longrightarrow$ in general, unproblematic :-)

Read-Write: A register is simultaneously read and written.

Conflict Resolution:

- ... ruled out!
- Read is delayed (stalls), until write has terminated!
- Read before write returns old value!

Write-Write: A register is simultaneously written to.

⟹ in general, unproblematic :-)

Conflict Resolutions:

- ... ruled out!
- ...

In Our Examples ...

- simultaneous read is permitted;
- simultaneous write/read and write/write is ruled out;
- no stalls are injected.

We first consider basic blocks only, i.e., linear sequences of assignments ...

Idea: Data Dependence Graph

Vertices	Instructions
Edges	Dependencies

Example:

(1) $x = x + 1;$

(2) $y = M[A];$

(3) $t = z;$

(4) $z = M[A + x];$

(5) $t = y + z;$

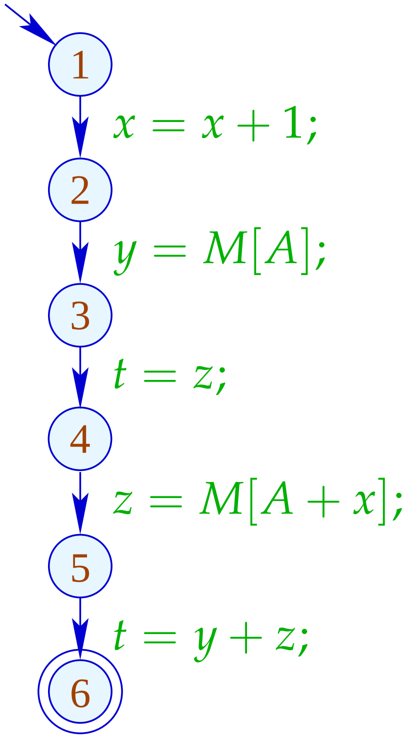
Possible Dependencies:

Definition	→	Use	//	Reaching Definitions
Use	→	Definition	//	???
Definition	→	Definition	//	Reaching Definitions

Reaching Definitions:

Determine for each u which definitions may reach $\Longrightarrow$ can be determined by means of a system of constraints :-)

... in the Example:



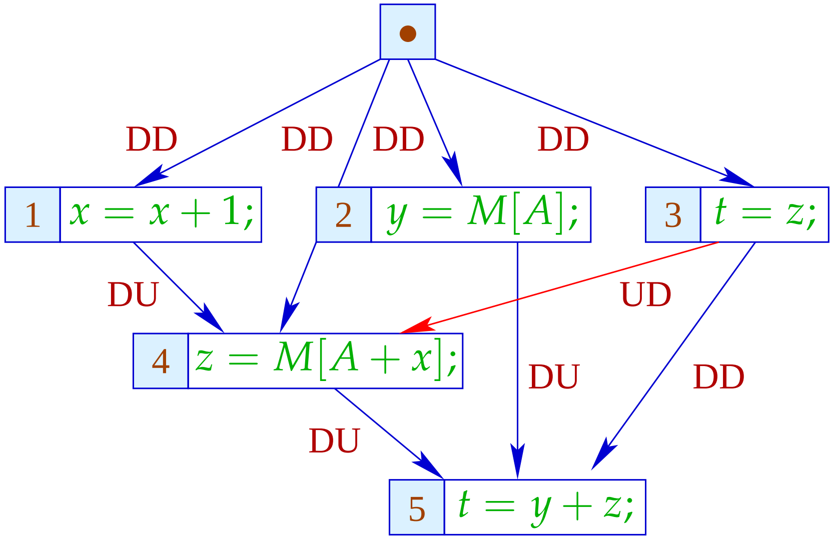
	$\mathcal{R}$
1	$\{\langle x, 1\rangle, \langle y, 1\rangle, \langle z, 1\rangle, \langle t, 1\rangle\}$
2	$\{\langle x, 2\rangle, \langle y, 1\rangle, \langle z, 1\rangle, \langle t, 1\rangle\}$
3	$\{\langle x, 2\rangle, \langle y, 3\rangle, \langle z, 1\rangle, \langle t, 1\rangle\}$
4	$\{\langle x, 2\rangle, \langle y, 3\rangle, \langle z, 1\rangle, \langle t, 4\rangle\}$
5	$\{\langle x, 2\rangle, \langle y, 3\rangle, \langle z, 5\rangle, \langle t, 4\rangle\}$
6	$\{\langle x, 2\rangle, \langle y, 3\rangle, \langle z, 5\rangle, \langle t, 6\rangle\}$

Let $U_i, \; D_i$ denote the sets of variables which are used or defined at the edge outgoing from u_i . Then:

$$
\begin{array}{lll}
(u_1, u_2) \in DD & \text{if} & u_1 \in \mathcal{R}[u_2] \wedge D_1 \cap D_2 \neq \emptyset \\
(u_1, u_2) \in DU & \text{if} & u_1 \in \mathcal{R}[u_2] \wedge D_1 \cap U_2 \neq \emptyset
\end{array}
$$

... in the Example:

		Def	*Use*
1	$x = x + 1;$	$\{x\}$	$\{x\}$
2	$y = M[A];$	$\{y\}$	$\{A\}$
3	$t = z;$	$\{t\}$	$\{z\}$
4	$z = M[A + x];$	$\{z\}$	$\{A, x\}$
5	$t = y + z;$	$\{t\}$	$\{y, z\}$



The UD-edge $(3, 4)$ has been inserted to exclude that z is over-written before use :-)

In the next step, each instruction is annotated with its (required ressources, in particular, its) execution time.

Our goal is a maximally parallel correct sequence of words.

For that, we maintain the current system state:

$$\Sigma : \mathit{Vars} \to \mathbb{N}$$

$$\Sigma(x) \;\hat{=}\; \text{expected delay until } x \text{ is available}$$

Initially:

$$\Sigma(x) = 0$$

As an invariant, we guarantee on entry of the basic block, that all operations are terminated :-)

Then the slots of the word sequence are successively filled:

- We start with the minimal nodes in the dependence graph.
- If we fail to fill all slots of a word, we insert ; :-)
- After every inserted instruction, we re-compute Σ .

Warning:

→ The execution of two VLIWs can overlap !!!

→ Determining an optimal sequence, is NP-hard ...

Example: Word width $k = 2$

Word		State			
1	2	x	y	z	t
		0	0	0	0
$x = x + 1$	$y = M[A]$	0	1	0	0
$t = z$	$z = M[A + x]$	0	0	1	0
		0	0	0	0
$t = y + z$		0	0	0	0

In each cycle, the execution of a new word is triggered.

The state just records the number of cycles still to be waited for the result :-)

Note:

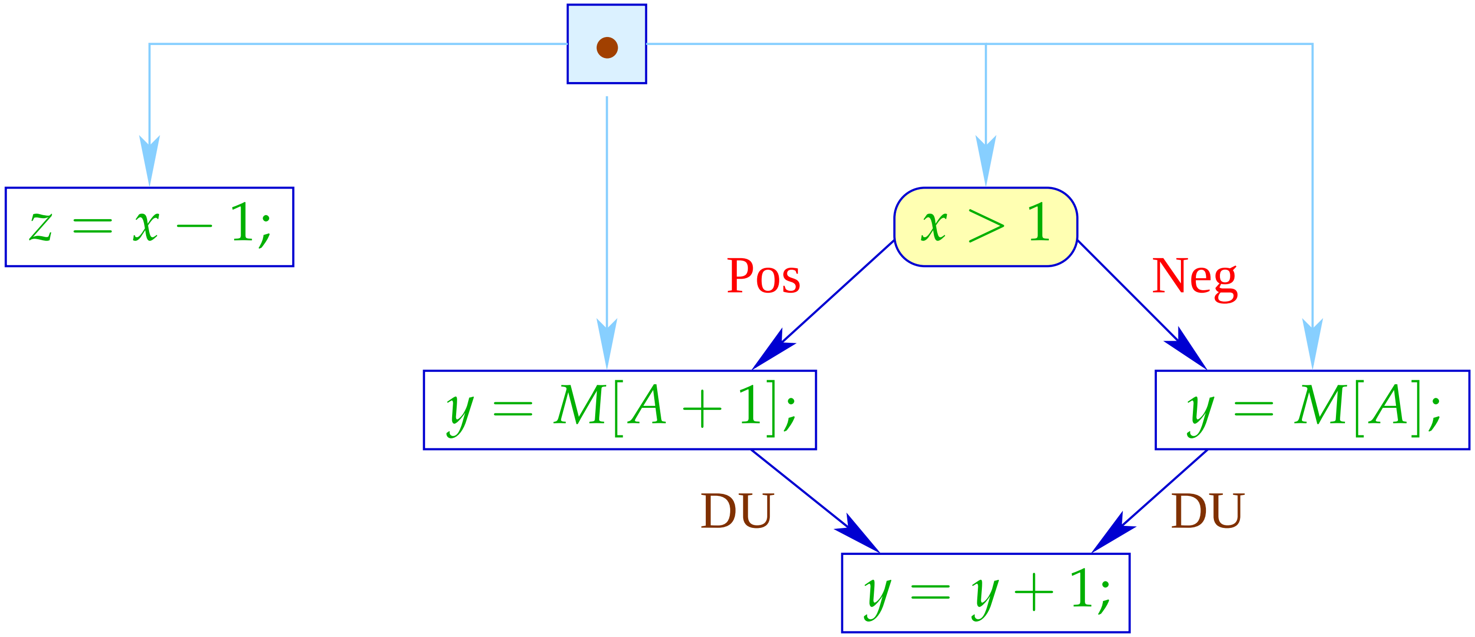
- If instructions put constraints on future selection, we also record these in Σ .
- Overall, we still distinuish just finitely many system states :-)
- The computation of the effect of a VLIW onto Σ can be compiled into a finite automaton !!!
- This automaton, though, could be quite huge :-(
- The challenge of making choices still remains :-(
- Basic blocks usually are not very large

$\Longrightarrow$ opportunities for parallelization are limited :-((

Extension 1: Acyclic Code

```
if ($x > 1$) {
        $y = M[A];$
        $z = x - 1;$
} else {
        $y = M[A + 1];$
        $z = x - 1;$
}
$y = y + 1;$
```

The dependence graph must be enriched with extra control-dependencies ...



The statement $z = x - 1;$ is executed with the same arguments in both branches and does not modify any of the remaining variables :-)

We could have moved it before the if anyway :-))

The following code could be generated:

	$z = x - 1$	if $(!(x > 0))$ goto A
	$y = M[A]$	
	goto B	
$A :$	$y = M[A + 1]$	
$B :$	$y = y + 1$	

At every jump target, we guarantee the invariant :-(

If we allow several (known) states on entry of a sub-block, we can generate code which complies with all of these.

... in the Example:

	$z = x - 1$	if $(!(x > 0))$ goto A
	$y = M[A]$	goto B
$A:$	$y = M[A + 1]$	
$B:$		
	$y = y + 1$	

If this parallelism is not yet sufficient, we could try to speculatively execute possibly useful tasks ...

For that, we require:

- an idea which alternative is executed more frequently;
- the wrong execution may not end in a catastrophy, i.e., run-time errors such as, e.g., division by 0;
- the wrong execution must allow roll-back (e.g., by delaying a commit) or may not have any observational effects ...

... in the Example:

	$z = x - 1$	$y = M[A]$	if $(x > 0)$ goto B
	$y = M[A+1]$		
$B:$			
	$y = y + 1$		

In the case $x \leq 0$ we have $y = M[A]$ executed in advance.

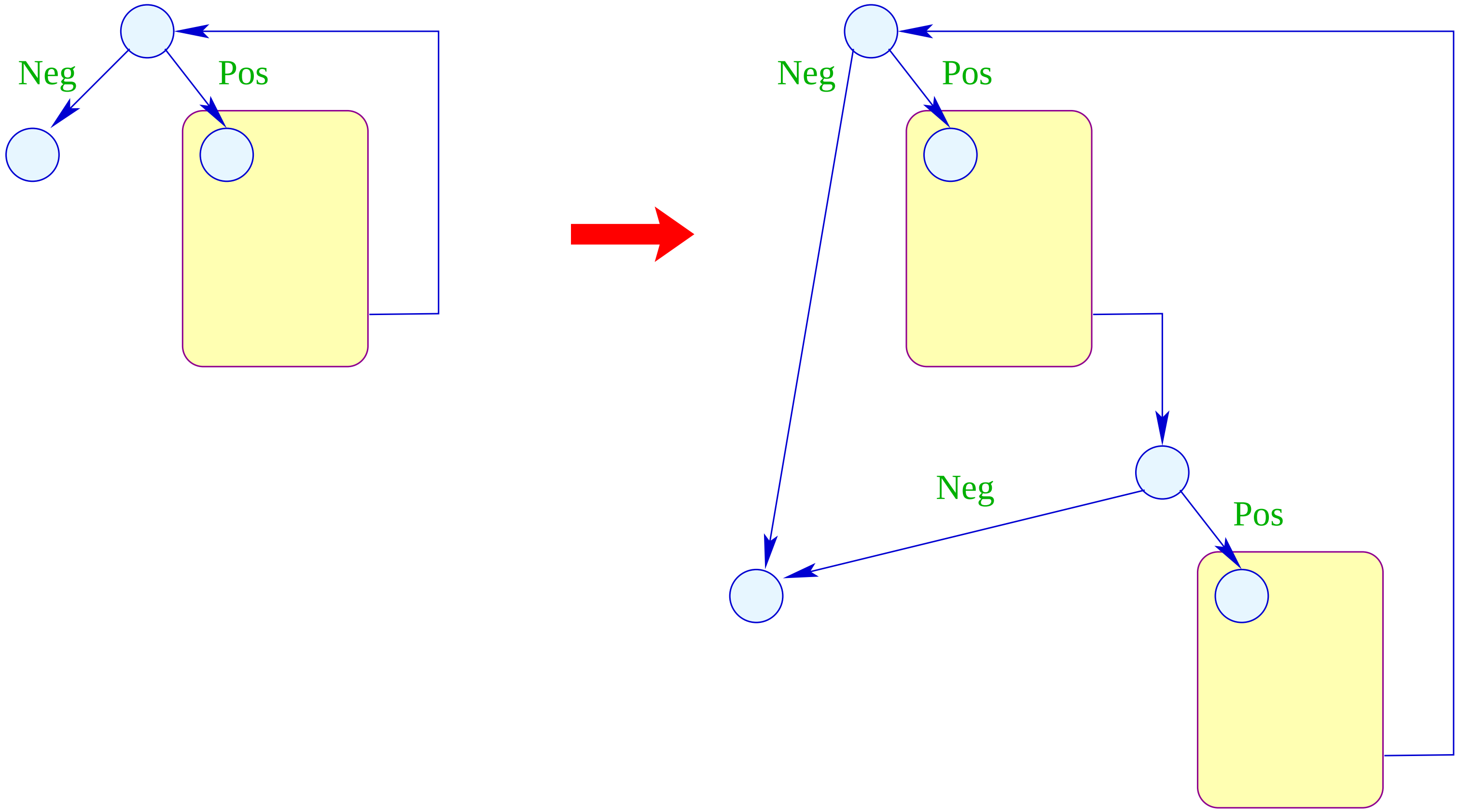
This value, however, is overwritten in the next step :-)

In general:

$x = e;$ has no observable effect in a branch if x is dead in this branch :-)

Extension 2: Unrolling of Loops

We may unrole important, i.e., inner loops several times:



Now it is clear which side of tests to prefer:

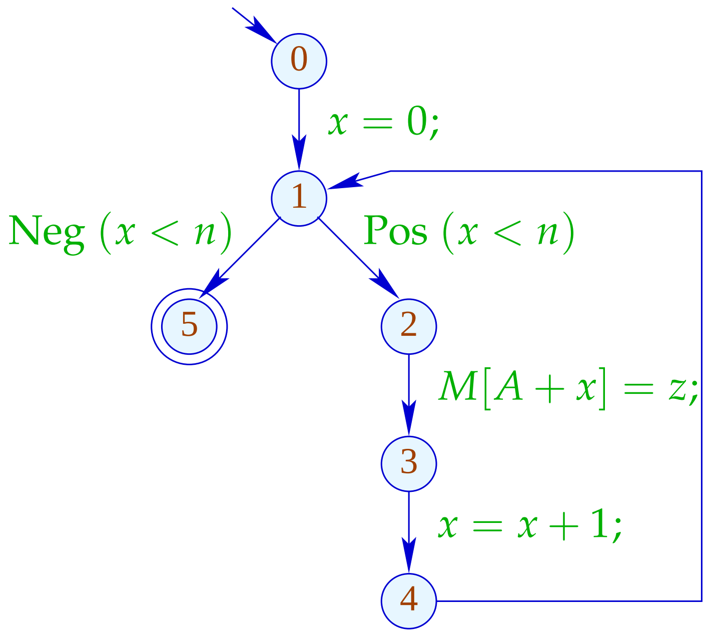
the side which stays within the unroled body of the loop :-)

Warning:

- The different instances of the body are translated relative to possibly different initial states :-)
- The code behind the loop must be correct relative to the exit state corresponding to every jump out of the loop!

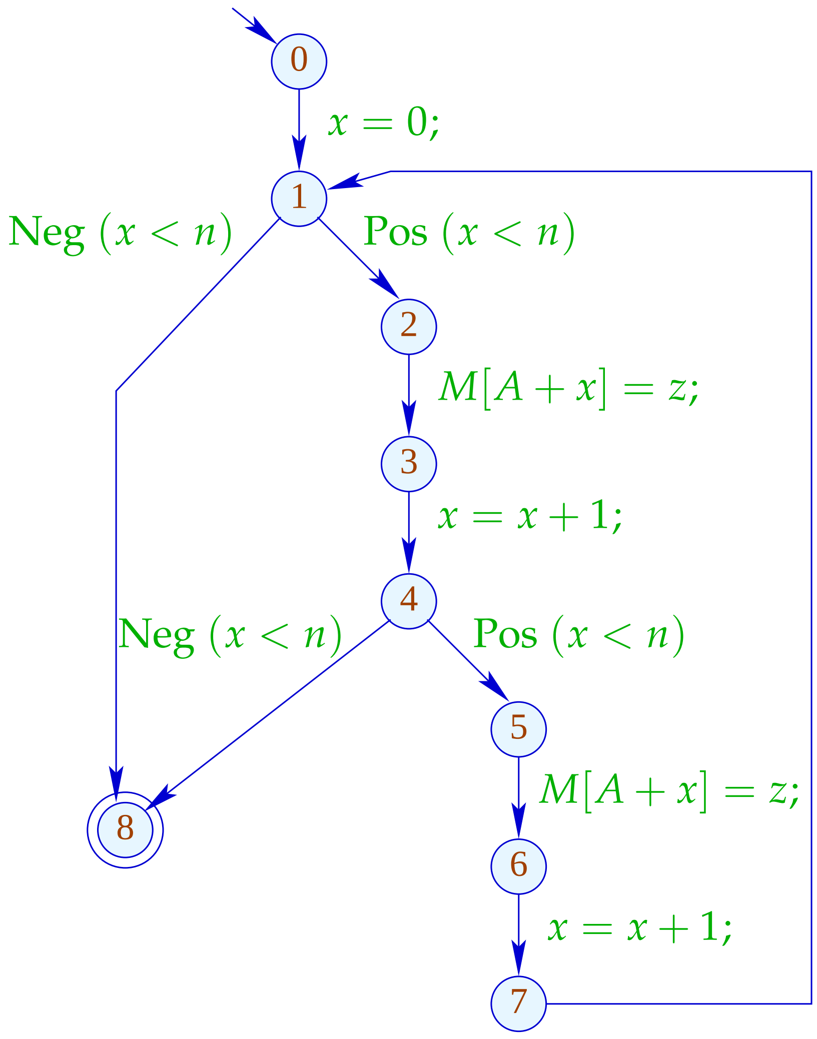
Example:

$$\text{for} \quad (x = 0; x < n; x{+}{+})$$
$$M[A + x] = z;$$



Duplication of the body yields:

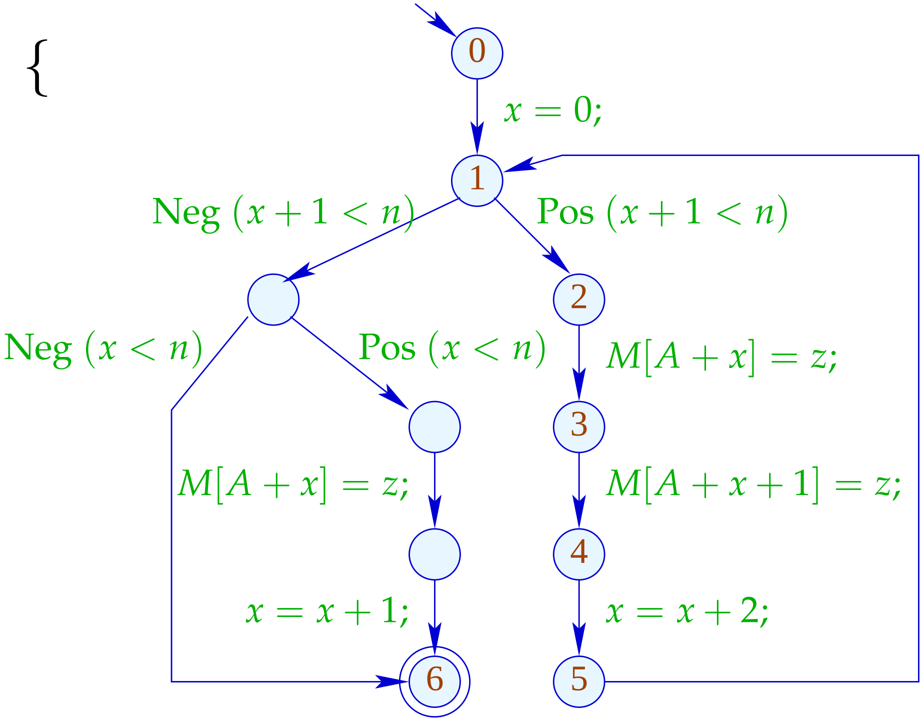
```
for ($x = 0; x < n; x$++) {
        $M[A + x] = z;$
        $x = x + 1;$
        if $(!(x < n))$ break;
        $M[A + x] = z;$
        }
```



It would be better if we could remove the assignment $x = x + 1;$ together with the test in the middle — since these serialize the execution of the copies !!

This is possible if we substitute $x + 1$ for x in the second copy, transform the condition and add a compensation code:

```
for  $(x = 0; x + 1 < n; x = x + 2)$  {
          $M[A + x] = z;$
          $M[A + x + 1] = z;$
          }
if  $(x < n)$  {
          $M[A + x] = z;$
          $x = x + 1;$
          }
```



Discussion:

- Elimination of the intermediate test together with the the fusion of all increments at the end reveals that the different loop iterations are in fact independent :-)
- Nonetheless, we do not gain much since we only allow one store per word :-(
- If right-hand sides, however, are more complex, we can interleave their evaluation with the stores :-)

Extension 3:

Sometimes, one loop alone does not provide enough opportunities for parallelization :-(

... but perhaps two successively in a row :-)

Example:

```
for (x = 0; x < n; x++) {                for (x = 0; x < n; x++) {
        R = B[x];                                R = B[x];
        S = C[x];                                S = C[x];
        T_1 = R + S;                             T_2 = R - S;
        A[x] = T_1;                              C[x] = T_2;
        }                                        }
```

In order to fuse two loops into one, we require that:

- the iteration schemes coincide;
- the two loops access different data.

In case of individual variables, this can easily be verified.

This is more difficult in presence of arrays.

Taking the source program into account, accesses to distinct statically allocated arrays can be identified.

An analysis of accesses to the same array is significantly more difficult ...

Assume that the blocks A, B, C are distinct.

Then we can combine the two loops into:

```
for $(x = 0; x < n; x{+}{+})$ {
        $R = B[x];$                      $R = B[x];$
        $S = C[x];$                      $S = C[x];$
        $T_1 = R + S;$                   $T_2 = R - S;$
        $A[x] = T_1;$                    $C[x] = T_2;$
                                         }
```

The first loop may in iteration x not read data which the second loop writes to in iterations $< x$.

The second loop may in iteration x not read data which the first loop writes to in iterations $> x$.

If the index expressions of jointly accessed arrays are linear, the given constraints can be verified through integer linear programming ...

$$\begin{array}{rcl} i & \geq & 0 \\ i & \leq & x - 1 \end{array} \qquad \begin{array}{rcl} x_{\text{write}} & = & i \\ x_{\text{read}} & = & x \\ x_{\text{read}} & = & x_{\text{write}} \end{array}$$

// x_{read} read access to C by 1st loop

// x_{write} write access to C by 2nd loop

... obviously has no solution :-)

General Form:

$$\begin{aligned} i &\geq t_1 \\ t_2 &\geq i \\ y_1 &= s_1 \\ y_2 &= s_2 \\ y_1 &= y_2 \end{aligned}$$

for linear expressions s, t_1, t_2, s_1, s_2 over i and the iteration variables.

This can be simplified to:

$$0 \leq s - t_1 \qquad 0 \leq t_2 - s \qquad 0 = s_1 - s_2$$

What should we do with it ???

Simple Case:

The two inequations have no solution over $\mathbb{Q}$.

Then they also have no solution over $\mathbb{Z}$:-)

... in Our Example:

$$\begin{array}{rclcl} x & = & i & & \\ 0 & \leq & i & = & x \\ 0 & \leq & x-1-i & = & -1 \end{array}$$

The second inequation has no solution :-)

Equal Signs:

If a variable x occurs in all inequations with the same sign, then there is always a solution :-(

Example:

$$\begin{array}{rcl} 0 & \leq & 13 + 7 \cdot x \\ 0 & \leq & -1 + 5 \cdot x \end{array}$$

The variable x may, e.g., be chosen as:

$$x \geq \mathsf{max}(-\frac{13}{7}, \frac{1}{5}) = \frac{1}{5}$$

Unequal Signs:

A variable x occurs in one inequation negative, in all others positive (if at all). Then a system can be constructed without x ...

Example:

$$\begin{array}{rcl} 0 & \leq & 13 - 7 \cdot x \\ 0 & \leq & -1 + 5 \cdot x \end{array} \quad \Longleftrightarrow \quad \begin{array}{rcl} x & \leq & \frac{13}{7} \\ 0 & \leq & -1 + 5 \cdot x \end{array}$$

Since $0 \leq -1 + 5 \cdot \frac{13}{7}$ the system has at least a rational solution ...

One Variable:

The inequations where x occurs positive, provide lower bounds.

The inequations where x occurs negative, provide upper bounds.

If G, L are the greatest lower and the least upper bound, respectively, then all (integer) solution are in the interval $[G, L]$:-)

Example:

$$\begin{array}{lll} 0 & \leq & 13 - 7 \cdot x \\ 0 & \leq & -1 + 5 \cdot x \end{array} \quad \Longleftrightarrow \quad \begin{array}{lll} x & \leq & \frac{13}{7} \\ x & \geq & \frac{1}{5} \end{array}$$

The only integer solution of the system is $x = 1$:-)

Discussion:

- Solutions only matter within the bounds to the iteration variables.
- Every integer solution there provides a conflict.
- Fusion of loops is possible if no conflicts occur :-)
- The given secial cases suffice to solve the case of two variables over $\mathbb{Q}$ and of one variable over $\mathbb{Z}$:-)
- The number of variables in the inequations corresponds to the nesting-depth of for-loops $\Longrightarrow$ in general, is quite small :-)

Discussion:

- Integer Linear Programming (ILP) can decide satisfiability of a finite set of equations/inequations over $\mathbb{Z}$ of the form:

$$\sum_{i=1}^{n} a_i \cdot x_i = b \quad \text{bzw.} \quad \sum_{i=1}^{n} a_i \cdot x_i \geq b\,, \qquad a_i \in \mathbb{Z}$$

- Moreover, a (linear) cost function can be optimized :-)
- Warning: The decision problem is in general, already NP-hard !!!
- Notwithstanding that, surprisingly efficient implementations exist.
- Not just loop fusion, but also other re-organizations of loops yield ILP problems ...

Background 5: Presburger Arithmetic

Many problems in computer science can be formulated without multiplication :-)

Let us first consider two simple special cases ...

1. Linear Equations

$$\begin{array}{rcrcrcr} 2x & + & 3y & & & = & 24 \\ x & - & y & + & 5z & = & 3 \end{array}$$

Question:

- Is there a solution over $\mathbb{Q}$?
- Is there a solution over $\mathbb{Z}$?
- Is there a solution over $\mathbb{N}$?

Let us reconsider the equations:

$$\begin{array}{rcrcrcr} 2x & + & 3y & & & = & 24 \\ x & - & y & + & 5z & = & 3 \end{array}$$

Answers:

- Is there a solution over $\mathbb{Q}$? Yes
- Is there a solution over $\mathbb{Z}$? No
- Is there a solution over $\mathbb{N}$? No

Complexity:

- Is there a solution over $\mathbb{Q}$? Polynomial
- Is there a solution over $\mathbb{Z}$? Polynomial
- Is there a solution over $\mathbb{N}$? NP-hard

Solution Method for Integers:

Observation 1:

$$a_1 x_1 + \ldots + a_k x_k = b \qquad\qquad (\forall\, i : \; a_i \neq 0)$$

has a solution iff

$$\mathrm{gcd}\{a_1, \ldots, a_k\} \quad \mid \quad b$$

Example:

$$5y - 10z = 18$$

has no solution over $\mathbb{Z}$:-)

Example:

$$5y - 10z = 18$$

has no solution over $\mathbb{Z}$:-)

Observation 2:

Adding a multiple of one equation to another does not change the set of solutions :-)

Example:

$$\begin{array}{rcrcrcr} 2x & + & 3y & & & = & 24 \\ x & - & y & + & 5z & = & 3 \end{array}$$

Example:

$$\begin{array}{rcrcrcr} 2x & + & 3y & & & = & 24 \\ x & - & y & + & 5z & = & 3 \end{array}$$

$$\Longrightarrow$$

$$\begin{array}{rcrcrcr} & & 5y & - & 10z & = & 18 \\ x & - & y & + & 5z & = & 3 \end{array}$$

Observation 3:

Adding multiples of columns to another column is an invertible transformation which we keep track of in a separate matrix ...

$$\begin{array}{ccc|rrrrrrr}
1 & 0 & 0 & & & 5y & - & 10z & = & 18 \\
0 & 1 & 0 & x & - & y & + & 5z & = & 3 \\
0 & 0 & 1 & & & & & & &
\end{array}$$

$$\Longrightarrow$$

$$\begin{array}{ccc|rrrrrrr}
1 & 0 & 0 & & & 5y & & & = & 18 \\
0 & 1 & 2 & x & - & y & + & 3z & = & 3 \\
0 & 0 & 1 & & & & & & &
\end{array}$$

Observation 3:

Adding multiples of columns to another column is an invertible transformation which we keep track of in a separate matrix ...

$$\begin{array}{ccc|ccccccc}
1 & 0 & 0 & & & 5y & & & = & 18 \\
0 & 1 & 2 & x & - & y & + & 3z & = & 3 \\
0 & 0 & 1 & & & & & & &
\end{array}$$

$$\Longrightarrow$$

$$\begin{array}{ccc|ccccccc}
1 & 0 & -3 & & & 5y & & & = & 18 \\
0 & 1 & 2 & x & - & y & & & = & 3 \\
0 & 0 & 1 & & & & & & &
\end{array}$$

$\Longrightarrow$ triangular form !!

Observation 4:

- A special solution of a triangular system can be directly read off :-)
- All solutions of a homogeneous triangular system can be directly read off :-)
- All solutions of the original system can be recovered from the solutions of the triangular system by means of the accumulated transformation matrix:-))

Example

$$\begin{array}{rrr|rcrcr} 1 & 0 & -3 & & & 5y & = & 15 \\ 0 & 1 & 2 & x & - & y & = & 3 \\ 0 & 0 & 1 & & & & & \end{array}$$

One special solution:

$$[6, 3, 0]^\top$$

All solutions of the homogeneous system are spanned by:

$$[0, 0, 1]^\top$$

Solving over $\mathbb{N}$

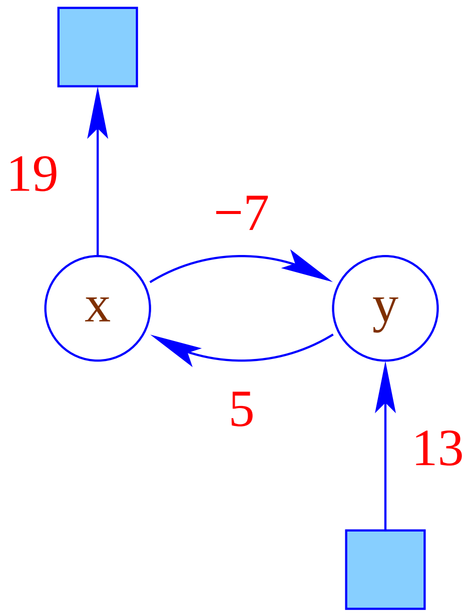
- ... is of major practical importance;
- ... has led to the development of many new techniques;
- ... easily allows to encode NP-hard problems;
- ... remains difficult if just three variables are allowed per equation.

2. One Polynomial Special Case:

$$\begin{aligned} x &\geq y + 5 \\ 19 \geq x & \\ y &\geq 13 \\ y &\geq x - 7 \end{aligned}$$

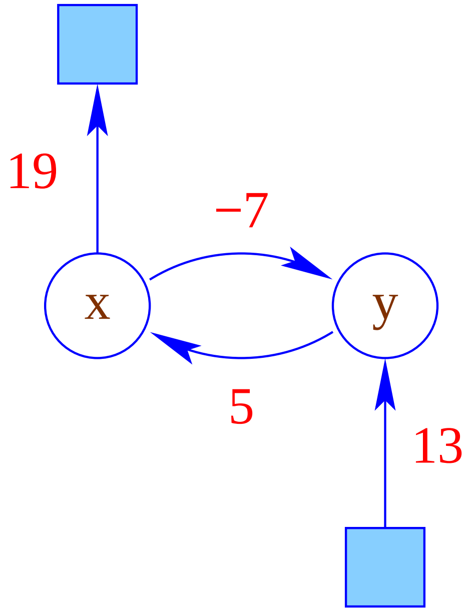
- There are at most 2 variables per in-equation;
- no scaling factors.

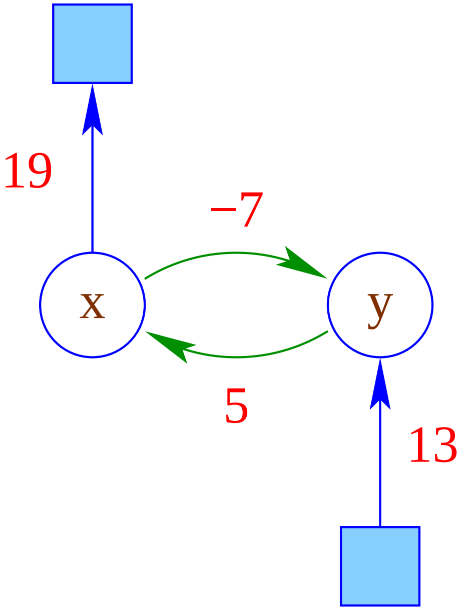
Idea: Represent the system by a graph:

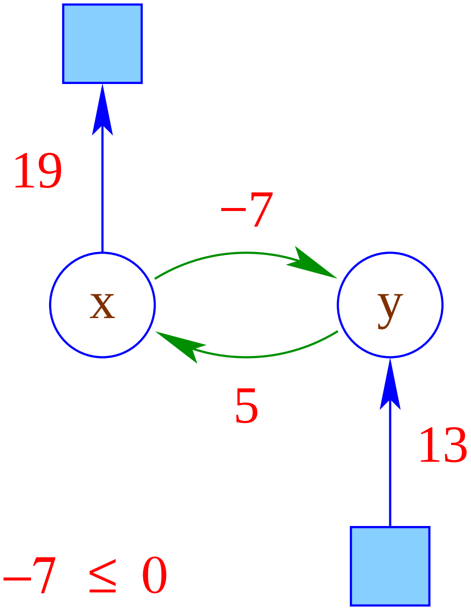


The in-equations are satisfiable iff

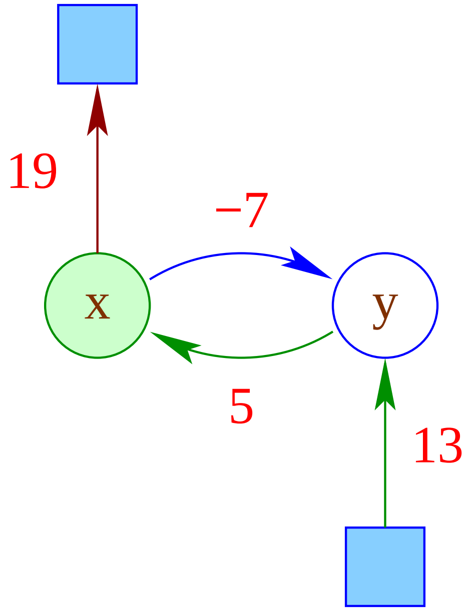
- the weight of every cycle are at most 0;
- the weights of paths reaching x are bounded by the weights leaving x.

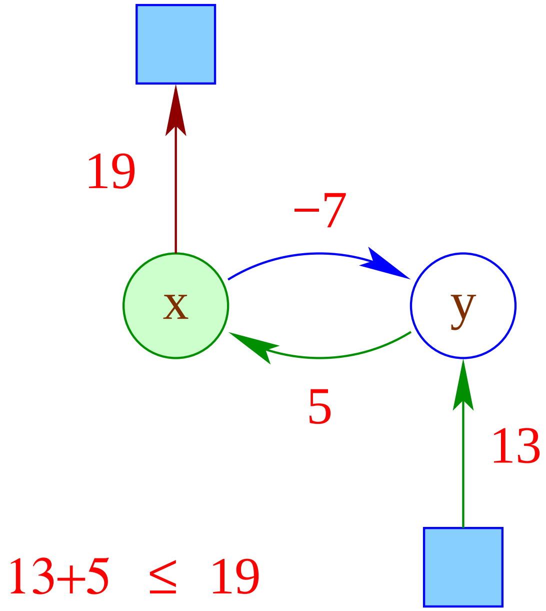

19
−7
x
y
5
13


19
−7
x
y
5
13



$5-7 \leq 0$


19
−7
x
y
5
13



$13+5 \leq 19$

The in-equations are satisfiable iff

- the weight of every cycle are at most 0;
- the weights of paths reaching x are bounded by the weights leaving x.

$\Longrightarrow$

Compute the reflexive and transitive closure of the edge weights!

3. A General Solution Method:

Idea: Fourier-Motzkin Elimination

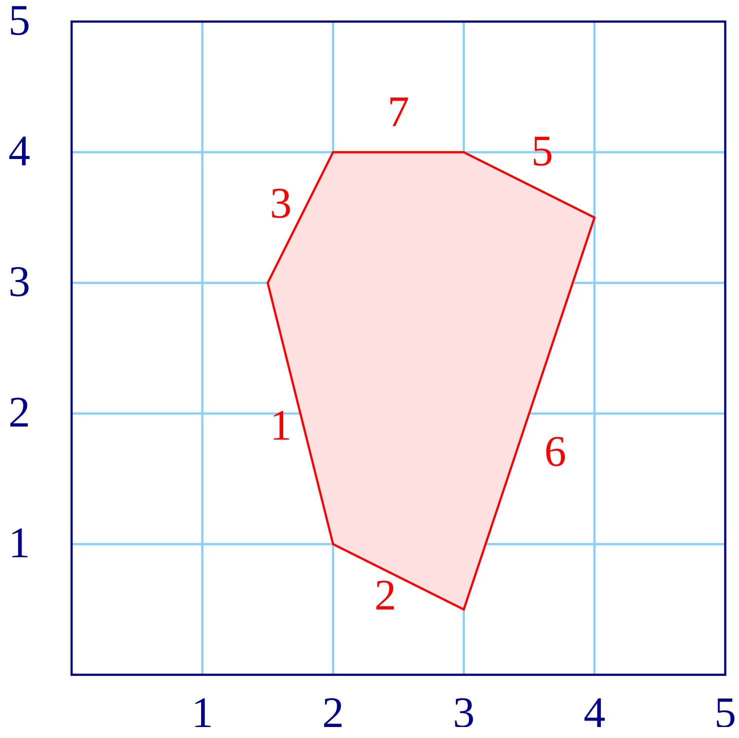
- Successively remove individual variables x !
- All in-equations with positive occurrences of x yield lower bounds.
- All in-equations with negative occurrences of x yield upper bounds.
- All lower bounds must be at most as big as all upper bounds ;-))

J. Boilly del.
Julien Sculp.

Jean Baptiste Joseph Fourier,   1768–1830

Example:

$$\begin{array}{rcll}
9 & \leq & 4x_1 + x_2 & (1) \\
4 & \leq & x_1 + 2x_2 & (2) \\
0 & \leq & 2x_1 - x_2 & (3) \\
6 & \leq & x_1 + 6x_2 & (4) \\
-11 & \leq & -x_1 - 2x_2 & (5) \\
-17 & \leq & -6x_1 + 2x_2 & (6) \\
-4 & \leq & -x_2 & (7)
\end{array}$$



For x_1 we obtain:

$$\begin{array}{rcll}
9 & \leq & 4x_1 + x_2 & (1) \\
4 & \leq & x_1 + 2x_2 & (2) \\
0 & \leq & 2x_1 - x_2 & (3) \\
6 & \leq & x_1 + 6x_2 & (4) \\
-11 & \leq & -x_1 - 2x_2 & (5) \\
-17 & \leq & -6x_1 + 2x_2 & (6) \\
-4 & \leq & -x_2 & (7)
\end{array}
\qquad
\begin{array}{lcll}
\frac{9}{4} - \frac{1}{4}x_2 & \leq & x_1 & (1) \\
4 - 2x_2 & \leq & x_1 & (2) \\
\frac{1}{2}x_2 & \leq & x_1 & (3) \\
6 - 6x_2 & \leq & x_1 & (4) \\
x_1 & \leq & 11 - 2x_2 & (5) \\
x_1 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (6) \\
-4 & \leq & -x_2 & (7)
\end{array}$$

If such an x_1 exists, all lower bounds must be bounded by all upper bounds, i.e.,

$$
\begin{array}{rcll}
\frac{9}{4} - \frac{1}{4}x_2 & \leq & 11 - 2x_2 & (1,5) \\
\frac{9}{4} - \frac{1}{4}x_2 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (1,6) \\
4 - 2x_2 & \leq & 11 - 2x_2 & (2,5) \\
4 - 2x_2 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (2,6) \\
\frac{1}{2}x_2 & \leq & 11 - 2x_2 & (3,5) \\
\frac{1}{2}x_2 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (3,6) \\
6 - 6x_2 & \leq & 11 - 2x_2 & (4,5) \\
6 - 6x_2 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (4,6) \\
-4 & \leq & -x_2 & (7)
\end{array}
\quad \text{or} \quad
\begin{array}{rcll}
-35 & \leq & -7x_2 & (1,5) \\
-\frac{7}{12} & \leq & \frac{7}{12}x_2 & (1,6) \\
-7 & \leq & 0 & (2,5) \\
\frac{7}{6} & \leq & \frac{7}{3}x_2 & (2,6) \\
-22 & \leq & -5x_2 & (3,5) \\
-\frac{17}{6} & \leq & -\frac{1}{6}x_2 & (3,6) \\
-5 & \leq & 4x_2 & (4,5) \\
\frac{19}{6} & \leq & \frac{19}{3}x_2 & (4,6) \\
-4 & \leq & -x_2 & (7)
\end{array}
$$

$$\begin{array}{rclccrcll}
\frac{9}{4}-\frac{1}{4}x_2 & \leq & 11-2x_2 & (1,5) & & -5 & \leq & -x_2 & (1,5)\\
\frac{9}{4}-\frac{1}{4}x_2 & \leq & \frac{17}{6}+\frac{1}{3}x_2 & (1,6) & & -1 & \leq & x_2 & (1,6)\\
4-2x_2 & \leq & 11-2x_2 & (2,5) & & -7 & \leq & 0 & (2,5)\\
4-2x_2 & \leq & \frac{17}{6}+\frac{1}{3}x_2 & (2,6) & & \frac{1}{2} & \leq & x_2 & (2,6)\\
\frac{1}{2}x_2 & \leq & 11-2x_2 & (3,5) & \text{or} & -\frac{22}{5} & \leq & -x_2 & (3,5)\\
\frac{1}{2}x_2 & \leq & \frac{17}{6}+\frac{1}{3}x_2 & (3,6) & & -17 & \leq & -x_2 & (3,6)\\
6-6x_2 & \leq & 11-2x_2 & (4,5) & & -\frac{5}{4} & \leq & x_2 & (4,5)\\
6-6x_2 & \leq & \frac{17}{6}+\frac{1}{3}x_2 & (4,6) & & \frac{1}{2} & \leq & x_2 & (4,6)\\
-4 & \leq & -x_2 & (7) & & -4 & \leq & -x_2 & (7)
\end{array}$$

This is the one-variable case which we can solve exactly:

$$\max\,\{-1, \boxed{\tfrac{1}{2}}, -\tfrac{5}{4}, \tfrac{1}{2}\} \quad \leq \quad x_2 \quad \leq \quad \min\,\{5, \tfrac{22}{5}, 17, \boxed{4}\}$$

From which we conclude: $x_2 \in [\frac{1}{2}, 4]$:-)

In General:

- The original system has a solution over $\mathbb{Q}$ iff the system after elimination of one variable has a solution over $\mathbb{Q}$:-)
- Every elimination step may square the number of in-equations $\Longrightarrow$ exponential run-time :-((
- It can be modified such that it also decides satisfiability over $\mathbb{Z}$ $\Longrightarrow$ Omega Test

William Worthington Pugh, Jr.
University of Maryland, College Park

Idea:

- We successively remove variables. Thereby we omit division ...
- If x only occurs with coeffient ± 1, we apply Fourier-Motzkin elimination :-)
- Otherwise, we provide a bound for a positive multiple of x ...

Consider, e.g., (1) and (6) :

$$\begin{array}{rcl} 6 \cdot x_1 & \leq & 17 + 2x_2 \\ 9 - x_2 & \leq & 4 \cdot x_1 \end{array}$$

W.l.o.g., we only consider strict in-equations:

$$\begin{array}{rcl} 6 \cdot x_1 & < & 18 + 2x_2 \\ 8 - x_2 & < & 4 \cdot x_1 \end{array}$$

... where we always divide by gcds:

$$\begin{array}{rcl} 3 \cdot x_1 & < & 9 + x_2 \\ 8 - x_2 & < & 4 \cdot x_1 \end{array}$$

This implies:

$$3 \cdot (8 - x_2) \quad < \quad 4 \cdot (9 + x_2)$$

We thereby obtain:

- If one derived in-equation is unsatisfiable, then also the overall system :-)
- If all derived in-equations are satisfiable, then there is a solution which, however, need not be integer :-(
- An integer solution is guaranteed to exist if there is sufficient separation between lower and upper bound ...
- Assume $\alpha < a \cdot x \qquad b \cdot x < \beta$.

  Then it should hold that:

  $$b \cdot \alpha < a \cdot \beta$$

  and moreover:

  $$\boxed{a \cdot b} < a \cdot \beta - b \cdot \alpha$$

... in the Example:

$$12 \quad < \quad 4 \cdot (9 + x_2) - 3 \cdot (8 - x_2)$$

or:

$$12 \quad < \quad 12 + 7x_2$$

or:

$$0 \quad < \quad x_2$$

In the example, also these strengthened in-equations are satisfiable

$\Longrightarrow$ the system has a solution over $\mathbb{Z}$:-)

Discussion:

- If the strengthened in-equations are satisfiable, then also the original system. The reverse implication may be wrong :-(
- In the case where upper and lower bound are not sufficiently separated, we have:

$$a \cdot \beta \leq b \cdot \alpha + \boxed{a \cdot b}$$

or:

$$b \cdot \alpha < ab \cdot x < b \cdot \alpha + \boxed{a \cdot b}$$

Division with b yields:

$$\alpha < a \cdot x < \alpha + \boxed{a}$$

$\Longrightarrow$ $\boxed{\alpha + i = a \cdot x}$ for some $i \in \{1, \ldots, a - 1\}$!!!

Discussion (cont.):

- → Fourier-Motzkin Elimination is not the best method for rational systems of in-equations.
- → The Omega test is necessarily exponential :-)

  If the system is solvable, the test generally terminates rapidly.

  It may have problems with unsolvable systems :-(
- → Also for ILP, there are other/smarter algorithms ...
- → For programming language problems, however, it seems to behave quite well :-)

4. Generalization to a Logic

Disjunction:

$$\begin{array}{llll} (x - 2y = 15 & \wedge & x + y = 7) & \vee \\ (x + y = 6 & \wedge & 3x + z = -8) & \end{array}$$

Quantors:

$$\exists\, x : \quad z - 2x = 42 \quad \wedge \quad z + x = 19$$

4. Generalization to a Logic

Disjunction:

$$\begin{array}{lcll}(x - 2y = 15 & \wedge & x + y = 7) & \vee \\ (x + y = 6 & \wedge & 3x + z = -8) & \end{array}$$

Quantors:

$$\exists\, x : \quad z - 2x = 42 \quad \wedge \quad z + x = 19$$

$\Longrightarrow$ Presburger Arithmetic

Mojzesz Presburger, 1904–1943 (?)

Presburger Arithmetic $=$ full arithmetic without multiplication

Presburger Arithmetic == full arithmetic
without multiplication

Arithmetic : highly undecidable :-(
even incomplete :-((

Presburger Arithmetic $=$ full arithmetic
without multiplication

Arithmetic : highly undecidable :-(
even incomplete :-((

$\Longrightarrow$ Hilbert’s 10th Problem

$\Longrightarrow$ Gödel’s Theorem

Presburger Formulas over $\mathbb{N}$:

$$\begin{array}{lll}\phi & ::= & x + y = z \quad | \quad x = n \quad | \\ & & \phi_1 \wedge \phi_2 \quad | \quad \neg\phi \quad | \\ & & \exists\, x : \ \phi\end{array}$$

Presburger Formulas over $\mathbb{N}$:

$$\begin{array}{lll}\phi & ::= & x + y = z \quad | \quad x = n \quad | \\ & & \phi_1 \land \phi_2 \quad | \quad \neg\phi \quad | \\ & & \exists\, x : \quad \phi\end{array}$$

Goal: PSAT

Find values for the free variables in $\mathbb{N}$ such that ϕ holds ...

Idea: Code the values of the variables as Words :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Idea: Code the values of the variables as Words :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Idea: Code the values of the variables as Words :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Idea: Code the values of the variables as Words :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Idea: Code the values of the variables as Words :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Idea: Code the values of the variables as Words :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Idea: Code the values of the variables as Words :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Idea: Code the values of the variables as Words :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Idea: Code the values of the variables as Words :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Idea: Code the values of the variables as Words :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Observation:

The set of satisfying variable assignments is regular :-))

Observation:

The set of satisfying variable assignments is regular :-))

$$\phi_1 \wedge \phi_2 \quad \Longrightarrow \quad \mathcal{L}(\phi_1) \cap \mathcal{L}(\phi_2) \quad \text{(Intersection)}$$

$$\neg\phi \quad \Longrightarrow \quad \overline{\mathcal{L}(\phi)} \quad \text{(Complement)}$$

$$\exists\, x : \quad \phi \quad \Longrightarrow \quad \pi_x(\mathcal{L}(\phi)) \quad \text{(Projection)}$$

Projecting away the *x*-component:

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

Projecting away the *x*-component:

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0

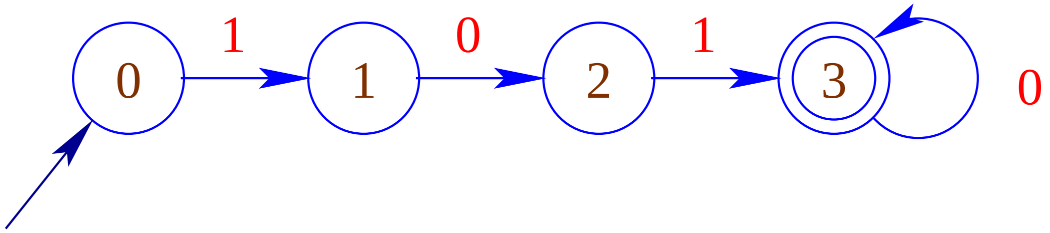
Warning:

- Our representation of numbers is not unique: 011101 should be accepted iff every word from $011101 \cdot 0^*$ is accepted!
- This property is preserved by union, intersection and complement :-)
- It is lost by projection !!!

$\Longrightarrow$ The automaton for projection must be enriched such that the property is re-established !!

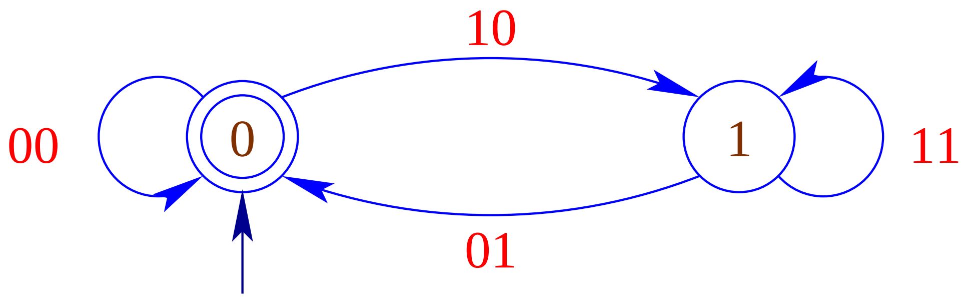
Automata for Basic Predicates:

$$x = 5$$



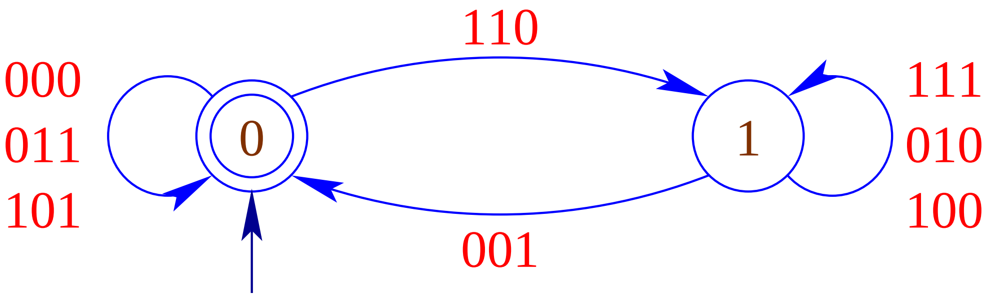
Automata for Basic Predicates:

$x+x = y$



Automata for Basic Predicates:

$x+y = z$



Results:

Ferrante, Rackoff,1973 : $\text{PSAT} \leq \text{DSPACE}(2^{2^{c \cdot n}})$

Results:

Ferrante, Rackoff,1973 : $\text{PSAT} \leq \text{DSPACE}(2^{2^{c \cdot n}})$

Fischer, Rabin,1974 : $\text{PSAT} \geq \text{NTIME}(2^{2^{c \cdot n}})$

3.3 Improving the Memory Layout

Goal:

- Better utilization of caches

  $\Longrightarrow$ reduction of the number of cache misses

- Reduction of allocation/de-allocation costs

  $\Longrightarrow$ replacing heap allocation by stack allocation

  $\Longrightarrow$ support to free superfluous heap objects

- Reduction of access costs

  $\Longrightarrow$ short-circuiting indirection chains (Unboxing)

1. Cache Optimization:

Idea: local memory access

- Loading from memory fetches not just one byte but fills a complete cache line.
- Access to neighbored cells become cheaper.
- If all data of an inner loop fits into the cache, the iteration becomes maximally memory-efficient ...

Possible Solutions:

→ Reorganize the data accesses !

→ Reorganize the data !

Such optimizations can be made fully automatic only for arrays :-(

Example:

```
for (j = 1; j < n; j++)
    for (i = 1; i < m; i++)
        a[i][j] = a[i-1][j-1] + a[i][j];
```

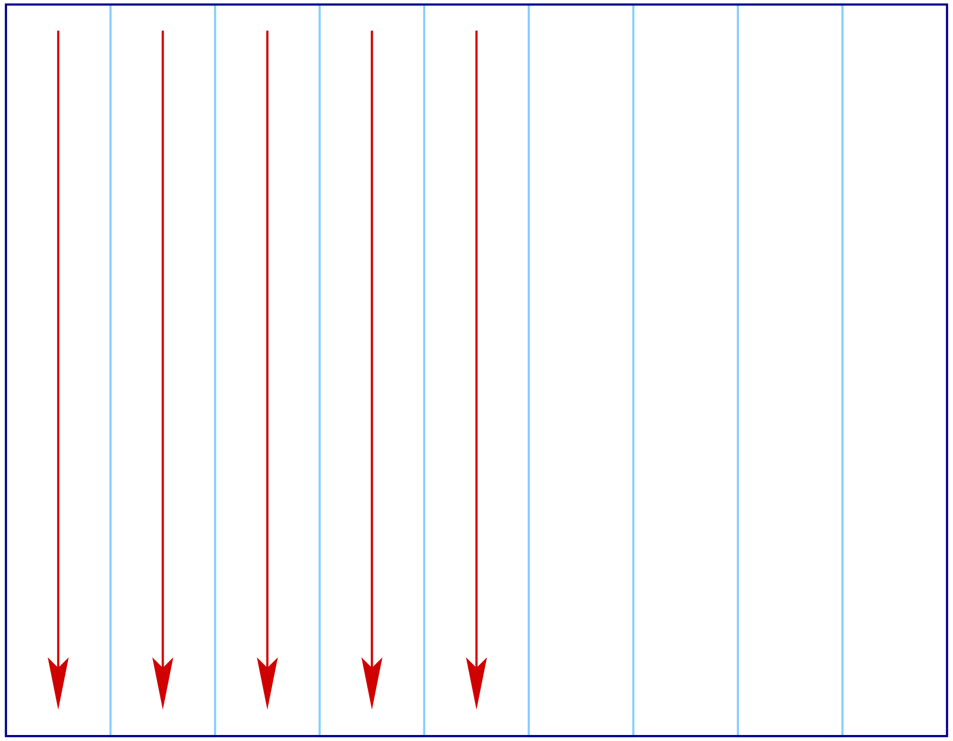
⟹ At first, always iterate over the rows!

⟹ Exchange the ordering of the iterations:

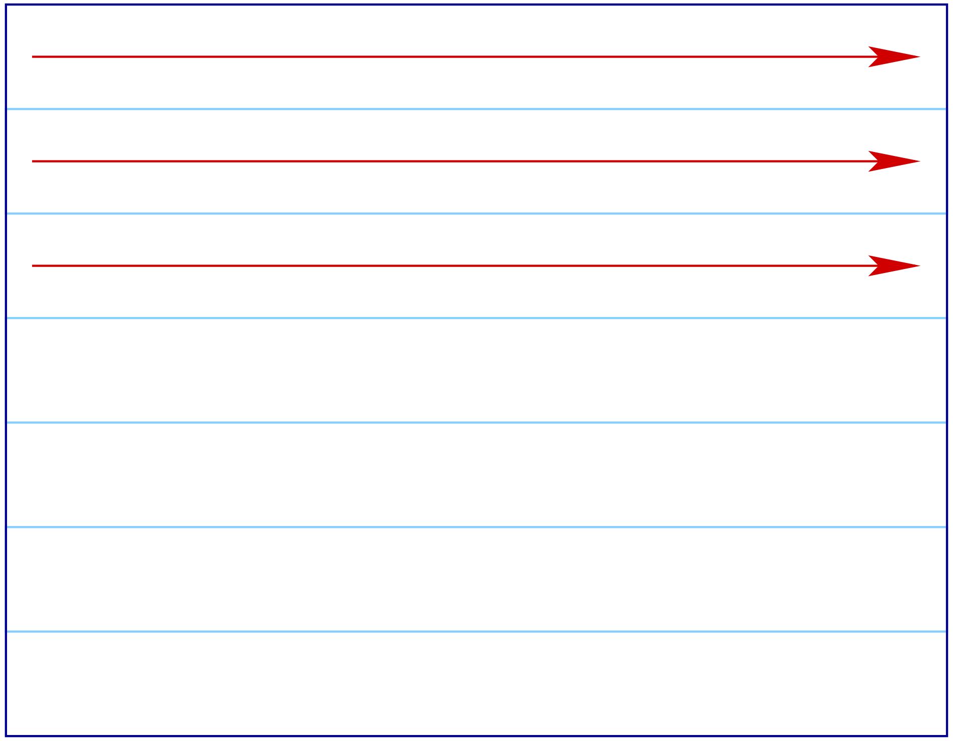
```
for $(i = 1; i < m; i{+}{+})$
    for $(j = 1; j < n; j{+}{+})$
        $a[i][j] = a[i-1][j-1] + a[i][j];$
```

When is this permitted???

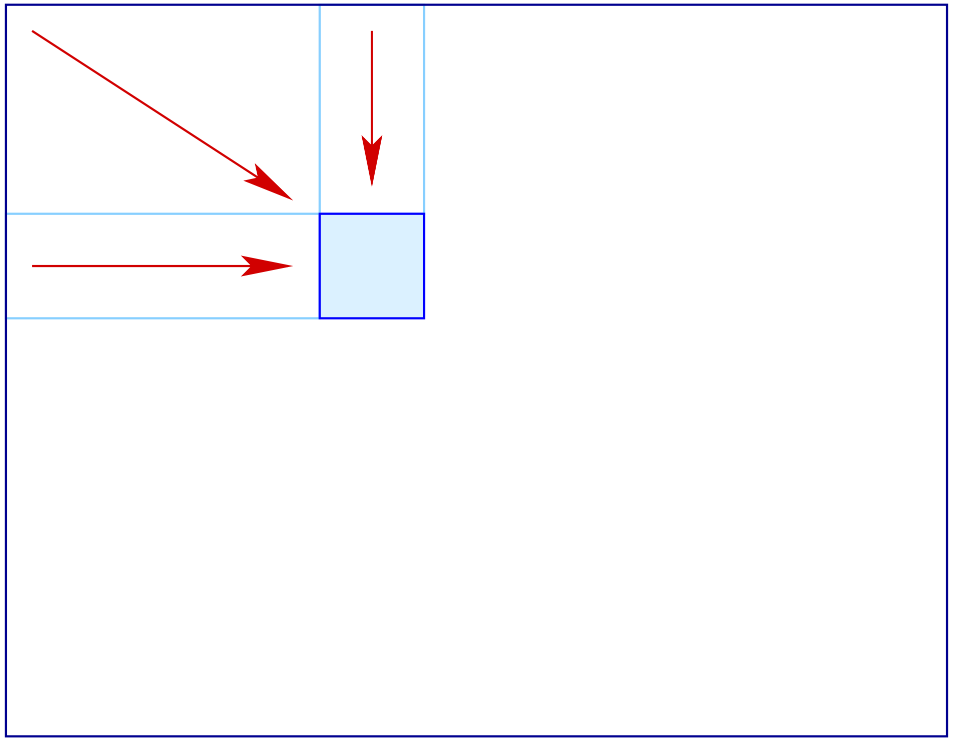
Iteration Scheme: before:



Iteration Scheme: after:



Iteration Scheme: allowed dependencies:



In our case, we must check that the following equation systems have no solution:

Write		Read
(i_1, j_1)	$=$	$(i_2 - 1, j_2 - 1)$
i_1	$\leq$	i_2
j_2	$\leq$	j_1
(i_1, j_1)	$=$	$(i_2 - 1, j_2 - 1)$
i_2	$\leq$	i_1
j_1	$\leq$	j_2

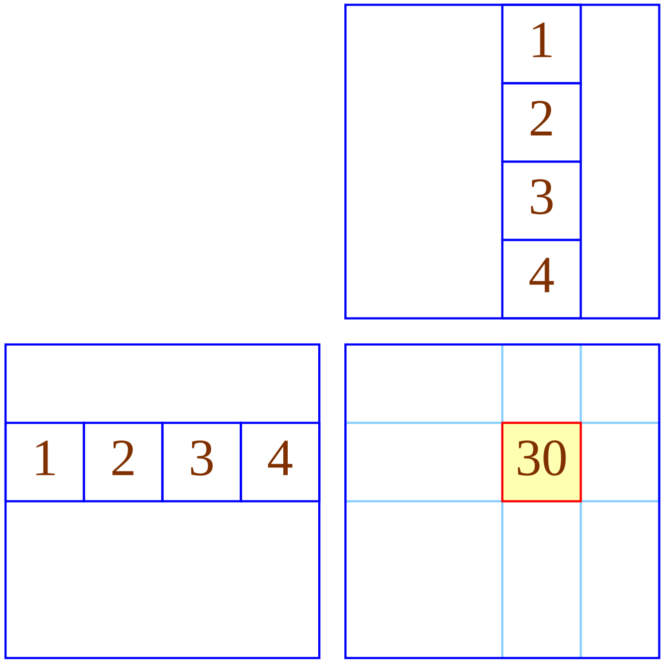
The first implies: $j_2 \leq j_2 - 1$ Hurra!

The second implies: $i_2 \leq i_2 - 1$ Hurra!

Example: Matrix-Matrix Multiplication

```
for $(i = 0; i < N; i{+}{+})$
    for $(j = 0; j < M; j{+}{+})$
        for $(k = 0; k < K; k{+}{+})$
            $c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];$
```

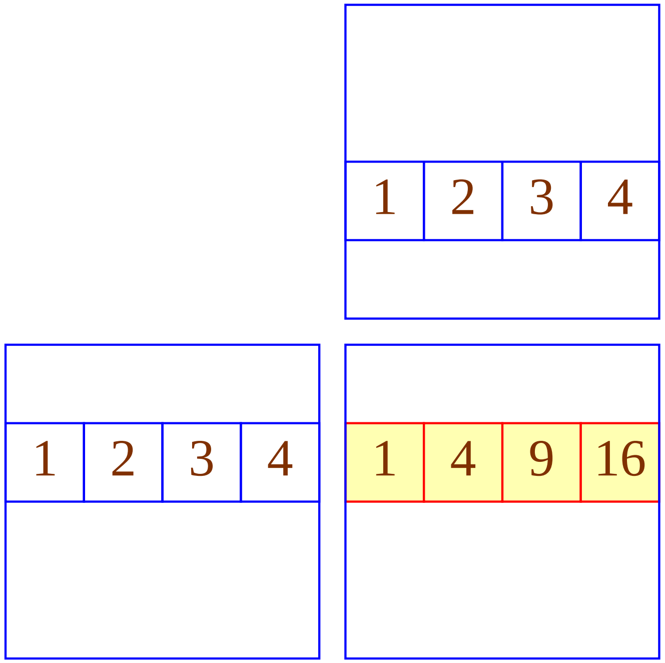
Over $b[][]$ the iteration is columnwise :-(


1
2
3
4
1 2 3 4
30

Exchange the two inner loops:

```
for $(i = 0; i < N; i{+}{+})$
    for $(k = 0; k < K; k{+}{+})$
        for $(j = 0; j < M; j{+}{+})$
            $c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];$
```

Is this permitted ???


1 2 3 4
1 2 3 4
1 4 9 16

Discussion:

- Correctness follows as before :-)
- A similar idea can also be used for the implementation of multiplication for row compressed matrices :-))
- Sometimes, the program must be massaged such that the transformation becomes applicable :-(
- Matrix-matrix multiplication perhaps requires initialization of the result matrix first ...

```
for $(i = 0; i < N; i{+}{+})$
    for $(j = 0; j < M; j{+}{+})$ {
        $c[i][j] = 0;$
        for $(k = 0; k < K; k{+}{+})$
            $c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];$
    }
```

- Now, the two iterations can no longer be exchanged :-(
- The iteration over j, however, can be duplicated ...

```
for $(i = 0; i < N; i{+}{+})$ {
    for $(j = 0; j < M; j{+}{+})$ $c[i][j] = 0;$
    for $(j = 0; j < M; j{+}{+})$
        for $(k = 0; k < K; k{+}{+})$
            $c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];$
}
```

Correctness:

⟹ The read entries (here: no) may not be modified in the remaining body of the loop !!!

⟹ The ordering of the write accesses to a memory cell may not be changed :-)

We obtain:

```
for $(i = 0; i < N; i{+}{+})$ {
    for $(j = 0; j < M; j{+}{+})$ $c[i][j] = 0;$
    for $(k = 0; k < K; k{+}{+})$
        for $(j = 0; j < M; j{+}{+})$
            $c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];$
}
```

Discussion:

- Instead of fusing several loops, we now have distributed the loops :-)
- Accordingly, conditionals may be moved out of the loop
  ⟹ if-distribution ...

Warning:

Instead of using this transformation, the inner loop could also be optimized as follows:

```
for  $(i = 0; i < N; i{+}{+})$
    for  $(j = 0; j < M; j{+}{+})$  $\{$
        $t = 0;$
        for  $(k = 0; k < K; k{+}{+})$
            $t = t + a[i][k] \cdot b[k][j];$
        $c[i][j] = t;$
    $\}$
```

Idea:

If we find heavily used array elements $a[e_1]\ldots[e_r]$ whose index expressions stay constant within the inner loop, we could instead also provide auxiliary registers :-)

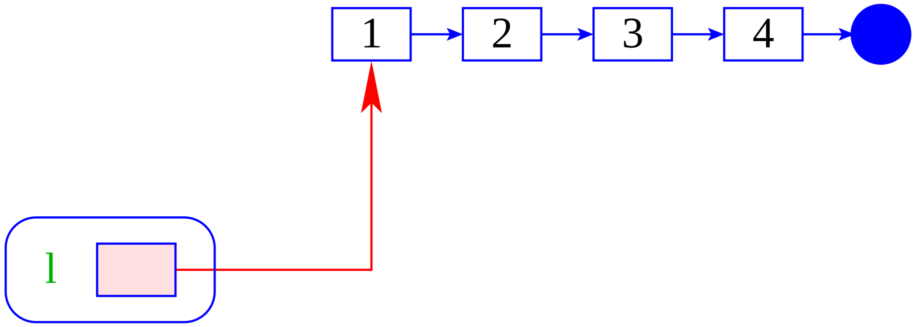
Warning:

The latter optimization prohibits the former and vice versa ...

Discussion:

- so far, the optimizations are concerned with iterations over arrays.
- Cache-aware organization of other data-structures is possible, but in general not fully automatic ...

Example: Stacks



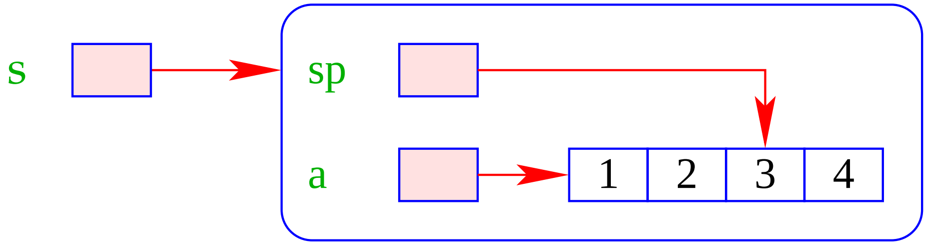
Advantage:

+ The implementation is simple :-)
+ The operations push / pop require constant time :-)
+ The data-structure may grow arbitrarily :-)

Disadvantage:

− The individual list objects may be arbitrarily dispersed over the memory :-(

Alternative:



Advantage:

- + The implementation is also simple :-)
- + The operations push / pop still require constant time :-)
- + The data are consequtively allocated; stack oscillations are typically small

$\Longrightarrow$ better Cache behavior !!!

Disadvantage:

– The data-structure is bounded :-(

Improvement:

- If the array is full, replace it with another of double size !!!
- If the array drops empty to a quarter, halve the array again !!!

⟹ The extra amortized costs are constant :-)

⟹ The implementation is no longer so trivial :-}

Discussion:

→ The same idea also works for queues :-)

→ Other data-structures are attempted to organize blockwise.

Problem: how can accesses be organized such that they refer mostly to the same block ???

$\Longrightarrow$ Algorithms for external data

2. Stack Allocation instead of Heap Allocation

Problem:

- Programming languages such as Java allocate all data-structures in the heap — even if they are only used within the current method :-(
- If no reference to these data survives the call, we want to allocate these on the stack :-)

$\Longrightarrow$ Escape Analysis

Idea:

Determine points-to information.

Determine if a created object is possibly reachable from the out side ...

Example: Our Pointer Language

$$x = \mathsf{new}();$$
$$y = \mathsf{new}();$$
$$x[A] = y;$$
$$z = y;$$
$$\mathsf{ret} = z;$$

... could be a possible method body ;-)

Accessible from the outside world are memory blocks which:

- are assigned to a global variable such as ret; or
- are reachable from global variables.

... in the Example:

$$x = \mathsf{new}();$$
$$y = \mathsf{new}();$$
$$x[A] = y;$$
$$z = y;$$
$$\mathsf{ret} = \boxed{z};$$

Accessible from the outside world are memory blocks which:

- are assigned to a global variable such as ret; or
- are reachable from global variables.

... in the Example:

$$x = \mathsf{new}();$$
$$y = \mathsf{new}();$$
$$x[A] = y;$$
$$z = \boxed{y};$$
$$\mathsf{ret} = \boxed{z};$$

Accessible from the outside world are memory blocks which:

- are assigned to a global variable such as ret; or
- are reachable from global variables.

... in the Example:

$$x = \mathsf{new}();$$

$$y = \mathsf{new}();$$

$$\boxed{x[A]} = y;$$

$$z = \boxed{y};$$

$$\mathsf{ret} = \boxed{z};$$

Accessible from the outside world are memory blocks which:

- are assigned to a global variable such as ret; or
- are reachable from global variables.

... in the Example:

$$
\begin{array}{l}
x = \mathsf{new}();\\
y = \boxed{\mathsf{new}()};\\
\boxed{x[A]} = y;\\
z = \boxed{y};\\
\mathsf{ret} = \boxed{z};
\end{array}
$$

We conclude:

- The objects which have been allocated by the first new() may never escape.
- They can be allocated on the stack :-)

Warning:

This is only meaningful if only few such objects are allocated during a method call :-(

If a local new() occurs within a loop, we still may allocate the objects in the heap ;-)

Extension: Procedures

- We require an interprocedural points-to analysis :-)
- We know the whole program, we can, e.g., merge the control-flow graphs of all procedures into one and compute the points-to information for this.
- Warning: If we always use the same global variables $y_1, y_2, \ldots$ for (the simulation of) parameter passing, the computed information is necessarily imprecise :-(
- If the whole program is not known, we must assume that each reference which is known to a procedure escapes :-((

3.4 Wrap-Up

We have considered various optimizations for improving hardware utilization.

Arrangement of the Optimizations:

- First, global restructuring of procedures/functions and of loops for better memory behavior ;-)
- Then local restructuring for better utilization of the instruction set and the processor parallelism :-)
- Then register allocation and finally,
- Peephole optimization for the final kick ...

Procedures:	Tail Recursion + Inlining Stack Allocation
Loops:	Iteration Reordering → if-Distribution → for-Distribution Value Caching
Bodies:	Life-Range Splitting (SSA) Instruction Selection Instruction Scheduling with → Loop Unrolling → Loop Fusion
Instructions:	Register Allocation Peephole Optimization

4 Optimization of Functional Programs

Example:

$$\begin{array}{lll} \textbf{let rec } \textsf{fac}\ x & = & \textbf{if }\ x \leq 1\ \textbf{ then }\ 1 \\ & & \textbf{else }\ x \cdot \textsf{fac}\ (x-1) \end{array}$$

- There are no basic blocks :-(
- There are no loops :-(
- Virtually all functions are recursive :-((

Strategies for Optimization:

$\Longrightarrow$ Improve specific inefficiencies such as:

- Pattern matching
- Lazy evaluation (if supported ;-)
- Indirections — Unboxing / Escape Analysis
- Intermediate data-structures — Deforestation

$\Longrightarrow$ Detect and/or generate loops with basic blocks :-)

- Tail recursion
- Inlining
- **let**-Floating

Then apply general optimization techniques

... e.g., by translation into C ;-)

Warning:

Novel analysis techniques are needed to collect information about functional programs.

Example: Inlining

$$\begin{array}{lll} \textbf{let}\ \ \textsf{max}\ (x, y) & = & \textbf{if}\ \ x > y\ \ \textbf{then}\ \ x \\ & & \textbf{else}\ \ y \\ \textbf{let}\ \ \textsf{abs}\ z & = & \textsf{max}\ (z, -z) \end{array}$$

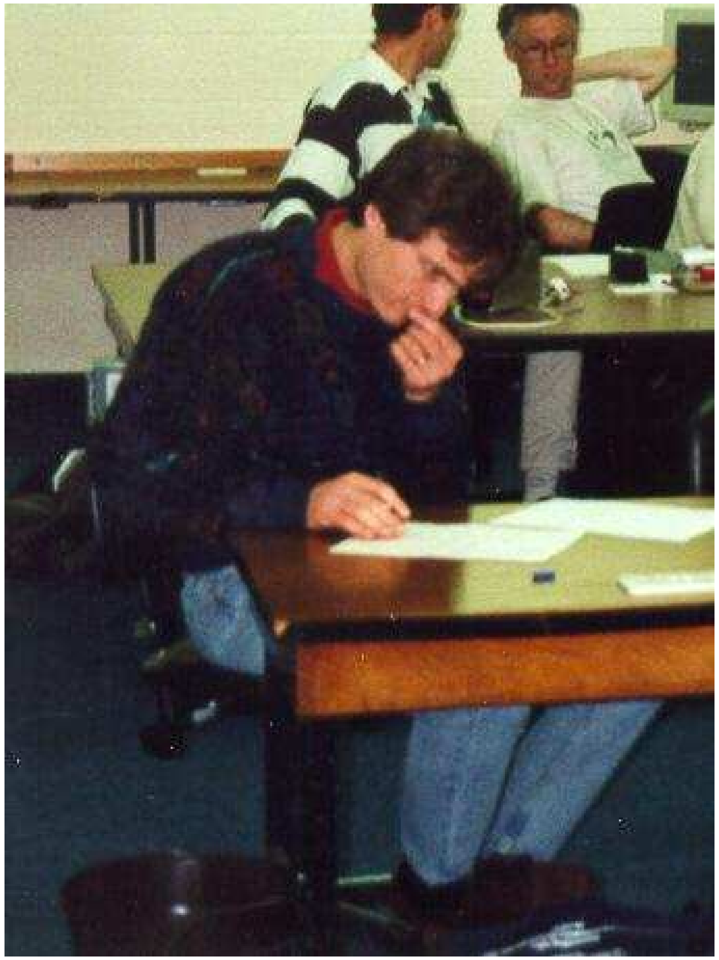
As result of the optimization we expect ...

$$\begin{array}{lll}
\textbf{let}\ \ \mathsf{max}\ (x,y) & = & \textbf{if}\ \ x > y\ \ \textbf{then}\ \ x \\
 & & \textbf{else}\ \ y \\
\textbf{let}\ \ \mathsf{abs}\ z & = & \textbf{let}\qquad x = z \\
 & & \textbf{and}\quad\ y = -z \\
 & & \textbf{in}\qquad \boxed{\begin{array}{l}\textbf{if}\ \ x > y\ \ \textbf{then}\ \ x \\ \textbf{else}\ \ y\end{array}} \\
 & & \textbf{end}
\end{array}$$

Discussion:

For the beginning, max is just a name. We must find out which value it takes at run-time

$\Longrightarrow$ Value Analysis required !!



Nevin Heintze in the Australian team
of the Prolog-Programming-Contest, 1998

The complete picture:

4.1 A Simple Functional Language

For simplicity, we consider:

$$\begin{array}{lcl} e & ::= & b \mid (e_1, \ldots, e_k) \mid c\ e_1\ \ldots\ e_k \mid \textbf{fun}\ x \rightarrow e \\ & & \mid (e_1\ e_2) \mid (\Box_1\ e) \mid (e_1\ \Box_2\ e_2) \mid \\ & & \textbf{let}\ x_1 = e_1\ \textbf{and} \ldots \textbf{and}\ x_k = e_k\ \textbf{in}\ e_0 \mid \\ & & \textbf{match}\ e_0\ \textbf{with}\ p_1 \rightarrow e_1\ \mid \ldots \mid\ p_k \rightarrow e_k \\ p & ::= & b \mid x \mid c\ x_1 \ldots x_k \mid (x_1, \ldots, x_k) \\ t & ::= & \textbf{let rec}\ x_1 = e_1\ \textbf{and} \ldots \textbf{and}\ x_k = e_k\ \textbf{in}\ e \end{array}$$

where b is a constant, x is a variable, c is a (data-)constructor and $\Box_i$ are i-ary operators.

Discussion:

- **let rec** only occurs on top-level.
- Constructors and functions are always unary.
  Instead, there are explicit tuples :-)
- **if**-expressions and case distinction in function definitions is reduced to **match**-expressions.
- In case distinctions, we allow just simple patterns.

  $\Longrightarrow$ Complex patterns must be decomposed ...
- **let**-definitions correspond to basic blocks :-)
- Type-annotations at variables, patterns or expressions could provide further useful information
  — which we ignore :-)

... in the Example:

A definition of max may look as follows:

$$
\begin{array}{llll}
\textbf{let}\ \mathsf{max} = \textbf{fun}\ x \rightarrow & \textbf{match}\ x\ \textbf{with}\ (x_1, x_2) \rightarrow (& & \\
 & \quad \textbf{match}\ x_1 < x_2 & & \\
 & \quad\ \textbf{with} & \mathsf{True} : & x_2 \\
 & \quad\quad \mid & \mathsf{False} : & x_1 \\
 & \quad) & &
\end{array}
$$

Accordingly, we have for abs :

$$\begin{array}{lll} \textbf{let } \mathsf{abs} & = & \textbf{fun } x \rightarrow \quad \textbf{let } z = (x, -x) \\ & & \qquad\qquad \textbf{in } \mathsf{max}\ z \end{array}$$

4.2 A Simple Value Analysis

Idea:

For every subexpression e we collect the set $[\![e]\!]^\sharp$ of possiblevalues of e ...

Let V denote the set of occurring (classes of) constants, applications of constructors and functions. As our lattice, we choose:

$$\mathbb{V} \;=\; 2^V$$

As usual, we put up a constraint system:

- If e is a value, i.e., of the form: $b, c\,e, (e_1, \ldots, e_k)$, an operator application or $\textbf{fun}\;x \;\rightarrow\; e$ we generate the constraint:

$$[\![e]\!]^\sharp \supseteq \{e\}$$

- If $e \equiv (e_1\; e_2)$ and $f \equiv \textbf{fun}\;x \;\rightarrow\; e'$, then

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & (f \in [\![e_1]\!]^\sharp)\,?\,[\![e']\!]^\sharp \;:\; \emptyset \\ [\![x]\!]^\sharp & \supseteq & (f \in [\![e_1]\!]^\sharp)\,?\,[\![e_2]\!]^\sharp \;:\; \emptyset \end{array}$$

...

- int-values returned by operators are described by the unevaluated expression;

  Operator applications which return Boolean values, e.g., by $\{$True, False$\}$:-)

- If $e \equiv \textbf{let } x_1 = e_1 \textbf{ and} \ldots \textbf{and } x_k = e_k \textbf{ in } e_0$, then we generate:

$$\begin{array}{lll} [\![x_i]\!]^\sharp & \supseteq & [\![e_i]\!]^\sharp \\ [\![e]\!]^\sharp & \supseteq & [\![e_0]\!]^\sharp \end{array}$$

- Assume $e \equiv \textbf{match}\ e_0\ \textbf{with}\ p_1 \ \to\ e_1 \mid \ldots \mid p_k \ \to\ e_k$.
  Then we generate for $p_i \equiv b$,

$$[\![e]\!]^\sharp \supseteq (b \in [\![e]\!]^\sharp)\ ?\ [\![e_i]\!]^\sharp\ :\ \emptyset$$

If $p_i \equiv c\, y$ and $v \equiv c\, e'$ is a value, then

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & (v \in [\![e_0]\!]^\sharp)\ ?\ [\![e_i]\!]^\sharp\ :\ \emptyset \\ [\![y]\!]^\sharp & \supseteq & (v \in [\![e_0]\!]^\sharp)\ ?\ [\![e']\!]^\sharp\ :\ \emptyset \end{array}$$

If $p_i \equiv (y_1, \ldots, y_k)$ and $v \equiv (e'_1, \ldots, e'_k)$ is a value, then

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & (v \in [\![e_0]\!]^\sharp)\ ?\ [\![e_i]\!]^\sharp\ :\ \emptyset \\ [\![y_j]\!]^\sharp & \supseteq & (v \in [\![e_0]\!]^\sharp)\ ?\ [\![e'_j]\!]^\sharp\ :\ \emptyset \end{array}$$

If $p_i \equiv y$, then

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & [\![e_i]\!]^\sharp \\ [\![y]\!]^\sharp & \supseteq & [\![e_0]\!]^\sharp \end{array}$$

Example The append-Function

Consider the concatenation of two lists. In Ocaml, we would write:

$$\begin{array}{llll}
\textbf{let rec } \mathsf{app} & = & \textbf{fun } x \;\to & \textbf{match } x \textbf{ with} \\
 & & & \quad [\,] \quad \to \quad \textbf{fun } y \;\to\; y \\
 & & & \;|\; h :: t \quad \to \quad \textbf{fun } y \;\to\; h :: \mathsf{app}\; t\; y \\
\textbf{in } \mathsf{app}\; [1;2]\; [3] & & &
\end{array}$$

The analysis then results in:

$$\begin{array}{lll}
[\![\mathsf{app}]\!]^\sharp & = & \{\textbf{fun } x \to \textbf{match} \ldots\} \\
[\![x]\!]^\sharp & = & \{[1;2], [1], [\,]\} \\
[\![\textbf{match} \ldots]\!]^\sharp & = & \{\textbf{fun } y \to y, \textbf{fun } y \to x :: \mathsf{app} \ldots\} \\
[\![y]\!]^\sharp & = & \{[3]\} \\
\ldots & &
\end{array}$$

$$\begin{array}{lcl}
\ldots & & \\
[\![h]\!]^\sharp &=& \{1,2\} \\
[\![t]\!]^\sharp &=& \{[2],[\,]\} \\
[\![\mathsf{app}\ t]\!]^\sharp &=& \\
[\![\mathsf{app}\ [1;2]]\!]^\sharp &=& \{\mathbf{fun}\ y \to y, \mathbf{fun}\ y \to x :: \mathsf{app} \ldots\} \\
[\![\mathsf{app}\ t\ y]\!]^\sharp &=& \\
[\![\mathsf{app}\ [1;2]\ [3]]\!]^\sharp &=& \{[3], h :: \mathsf{app} \ldots\}
\end{array}$$

Values $c\,e$ or $(e_1, \ldots, e_k)$ now are interpreted as recursive calls $c\,[\![e]\!]^\sharp$ or $([\![e_1]\!]^\sharp, \ldots, [\![e_k]\!]^\sharp)$, respectively.

$\Longrightarrow$ regular tree grammar

... in the Example:

We obtain for $A = [\![\mathsf{app}\; t\; y]\!]^\sharp$:

$$\begin{array}{lcl} A & \to & [3] \quad | \quad [\![h]\!]^\sharp :: A \\ [\![h]\!]^\sharp & \to & 1 \quad | \quad 2 \end{array}$$

Let $\mathcal{L}(e)$ denote the set of terms derivable from $[\![e]\!]^\sharp$ w.r.t. the regular tree grammar. Thus, e.g.,

$$\begin{array}{lcl} \mathcal{L}(h) & = & \{1,2\} \\ \mathcal{L}(\mathsf{app}\; t\; y) & = & \{[a_1; \ldots, a_r; 3] \mid r \geq 0, a_i \in \{1,2\}\} \end{array}$$

4.3 An Operational Semantics

Idea:

We construct a Big-Step operational semantics which evaluates expressions w.r.t. an environment :-)

Values are of the form:

$$v ::= b \mid c\,v \mid (v_1, \ldots, v_k) \mid (\mathbf{fun}\,x \to e, \eta)$$

Examples for Values:

$$\begin{array}{l} c\,1 \\ [1;2] = ::\,1\,(::\,2\,[\,]) \\ (\mathbf{fun}\,x \to x{::}y, \{y \mapsto [5]\}) \end{array}$$

Expressions are evaluated w.r.t. an environment $\eta : \mathit{Vars} \to \mathit{Values}$.

The Big-Step operational semantics provides rules to infer the value to which an expression is evaluated w.r.t. a given environment...

Values:

$$(b, \eta) \Longrightarrow b$$

$$(\mathbf{fun}\, x \ \to\ e, \eta) \Longrightarrow (\mathbf{fun}\, x \ \to\ e, \eta)$$

$$\frac{(e, \eta) \Longrightarrow v}{(c\, e, \eta) \Longrightarrow c\, v}$$

$$\frac{(e_1, \eta) \Longrightarrow v_1 \quad \ldots \quad (e_k, \eta) \Longrightarrow v_k}{((e_1, \ldots, e_k), \eta) \Longrightarrow (v_1, \ldots, v_k)}$$

Global Definition:

$$\frac{\begin{array}{l}\textbf{let rec } \ldots\ x = e\ \ldots\ \textbf{in } \ldots \\ (e, \emptyset) \Longrightarrow v\end{array}}{(x, \eta) \Longrightarrow v}$$

Function Application:

$$\frac{(e_1, \eta) \Longrightarrow (\textbf{fun } x \;\rightarrow\; e, \eta_1) \qquad (e_2, \eta) \Longrightarrow v_2 \qquad (e, \eta_1 \oplus \{x \mapsto v_2\}) \Longrightarrow v_3}{(e_1\ e_2, \eta) \Longrightarrow v_3}$$

Case Distinction 1:

$$\frac{(e, \eta) \Longrightarrow b \qquad (e_i, \eta) \Longrightarrow v_i}{(\textbf{match } e \textbf{ with } p_1 \;\rightarrow\; e_1 \mid \ldots \mid p_k \;\rightarrow\; e_k, \eta) \Longrightarrow v_i}$$

if $p_i \equiv b$ is the first pattern which matches b :-)

Case Distinction 2:

$$\frac{\begin{array}{l}(e, \eta) \Longrightarrow c\, v \\ (e_i, \eta \oplus \{z \mapsto v\}) \Longrightarrow v_i\end{array}}{(\textbf{match } e \textbf{ with } p_1 \;\to\; e_1 \mid \ldots \mid p_k \;\to\; e_k, \eta) \Longrightarrow v_i}$$

if $p_i \equiv c\ z$ is the first pattern which matches $c\, v$:-)

Case Distinction 3:

$$\frac{\begin{array}{l}(e, \eta) \Longrightarrow (v_1, \ldots, v_k) \\ (e_i, \eta \oplus \{y_1 \mapsto v_1, \ldots, y_1 \mapsto v_k\}) \Longrightarrow v_i\end{array}}{(\textbf{match } e \textbf{ with } p_1 \rightarrow e_1 \mid \ldots \mid p_k \rightarrow e_k, \eta) \Longrightarrow v_i}$$

if $\quad p_i \equiv (y_1, \ldots, y_k) \quad$ is the first pattern which matches $(v_1, \ldots, v_k)$:-)

Case Distinction 4:

$$\frac{(e, \eta) \Longrightarrow v \qquad (e_i, \eta \oplus \{x \mapsto v\}) \Longrightarrow v_i}{(\textbf{match } e \textbf{ with } p_1 \;\to\; e_1 \mid \ldots \mid p_k \;\to\; e_k, \eta) \Longrightarrow v_i}$$

if $p_i \equiv x$ is the first pattern which matches v :-)

Local Definitions:

$$
\frac{
\begin{array}{l}
(e_1, \eta) \Longrightarrow v_1 \\
(e_2, \eta \oplus \{x_1 \mapsto v_1\}) \Longrightarrow v_2 \\
\ldots \\
(e_k, \eta \oplus \{x_1 \mapsto v_1, \ldots, x_{k-1} \mapsto v_{k-1}\}) \Longrightarrow v_k \\
(e_0, \eta \oplus \{x_1 \mapsto v_1, \ldots, x_k \mapsto v_k\}) \Longrightarrow v_0
\end{array}
}{
(\textbf{let}\ x_1 = e_1\ \textbf{and} \ldots \textbf{and}\ x_k = e_k\ \textbf{in}\ e_0, \eta) \Longrightarrow v_0
}
$$

Correctness of the Analysis:

For every (e, η) occurring in a proof for the program, it should hold:

- If $\eta\,(x) = v$, then $[v] \in \mathcal{L}(x)$.
- If $(e, \eta) \Longrightarrow v$, then $[v] \in \mathcal{L}(e)$ ...

- where $[v]$ is the stripped expression corresponding to v, i.e., obtained by removing all environments.

Conclusion:

$\mathcal{L}(e)$ returns a superset of the values to which e is evaluated :-)

4.4 Application: Inlining

Problem:

- global variables. The program:

$$\begin{array}{lllll}
\mathbf{let} & x = 1 & & & \\
 & f = & \mathbf{let} & x = 2 & \\
 & & \mathbf{in} & \mathbf{fun}\ y & \rightarrow\ y + x \\
\mathbf{in} & f\ x & & &
\end{array}$$

... computes something else than:

$$\begin{array}{llll} \textbf{let} & x = 1 & & \\ & f = & \textbf{let} & x = 2 \\ & & \textbf{in} & \textbf{fun}\ y \quad \to \quad y + x \\ \textbf{in} & \multicolumn{3}{l}{\boxed{\begin{array}{ll}\textbf{let} & y = x \\ \textbf{in} & y + x\end{array}}} \end{array}$$

- recursive functions. In the definition:

$$\mathsf{foo} \quad = \quad \textbf{fun}\ y \quad \to \quad \mathsf{foo}\ y$$

foo should better not be substituted :-)

Idea 1:

→ First, we introduce unique variable names.

→ Then, we only substitute functions which are staticly within the scope of the same global variables as the application :-)

→ For every expression, we determine all function definitions with this property :-)

Let $D = D[e]$ denote the set of definitions which staticly arrive at e.

- ●● If $e \equiv \mathbf{let}\; x_1 = e_1 \;\mathbf{and} \ldots \mathbf{and}\; x_k = e_k \;\mathbf{in}\; e_0$ then:

$$\begin{array}{lll} D[e_1] & = & D \\ & \ldots & \\ D[e_k] & = & D \cup \{x_1, \ldots, x_{k-1}\} \\ D[e_0] & = & D \cup \{x_1, \ldots, x_k\} \end{array}$$

- ●● In all other cases, D is propagated to the sub-expressions unchanged :-)

  E.g., if $e \equiv \mathbf{fun}\; x \to e_1$ then:

$$D[e_1] = D$$

... in the Example:

$$
\begin{array}{lll}
\textbf{let} & x = 1 & \\
 & f = & \textbf{let} \quad x_1 = 2 \\
 & & \textbf{in} \quad \textbf{fun}\ y \ \rightarrow \ y + x_1 \\
\textbf{in} & f\ x &
\end{array}
$$

... the application $f\ x$ is not in the scope of x_1

$\Longrightarrow$ we first duplicate the definition of x_1 :

$$
\begin{array}{lllll}
\textbf{let} & x = 1 & & & \\
& x_1 = 2 & & & \\
& f = & \textbf{let} & x_1 = 2 & \\
& & \textbf{in} & \textbf{fun}\ y & \rightarrow\ y + x_1 \\
\textbf{in} & f\ x & & &
\end{array}
$$

$\Longrightarrow$ the inner definition becomes redundant !!!

$$\begin{array}{ll} \textbf{let} & x = 1 \\ & x_1 = 2 \\ & f = \textbf{fun}\ y \ \longrightarrow \ y + x_1 \\ \textbf{in} & f\ x \end{array}$$

$\Longrightarrow$ now we can apply inlining :

$$\begin{array}{ll}\textbf{let} & x = 1 \\ & x_1 = 2 \\ & f = \textbf{fun}\ y \ \longrightarrow\ y + x_1 \\ \textbf{in} & \boxed{\begin{array}{ll}\textbf{let} & y = x \\ \textbf{in} & y + x_1\end{array}}\end{array}$$

Removing variable-variable-assignments, we arrive at:

$$\begin{array}{ll} \textbf{let} & x = 1 \\ & x_1 = 2 \\ & f = \textbf{fun}\ y \longrightarrow y + x_1 \\ \textbf{in} & \boxed{x + x_1} \end{array}$$

Idea 2:

→ We apply our value analysis.

→ We ignore global variables :-)

→ We only substitute functions without free variables :-))

Example: The map-Function

```
let rec f   = fun x  →  x · x
        map = fun g  →  fun x  →  match x
                          with  [] →  []
                            |    :: z →  match z with (x1, x2)
                                         in  x1 :: map g x2
in  map  f  list
```

- The actual parameter f in the application map f is always $\textbf{fun}\ x \ \rightarrow\ x \cdot x$:-)
- Therefore, map f can be specialized to a new function h defined by:

$$
\begin{array}{lll}
\mathsf{h} & = & \textbf{let}\ g = \boxed{\textbf{fun}\ x \ \rightarrow\ x \cdot x} \\
 & & \textbf{in}\ \ \textbf{fun}\ x \ \rightarrow\ \textbf{match}\ x \\
 & & \qquad\quad \textbf{with}\ \ [\,] \ \rightarrow\ \quad [\,] \\
 & & \qquad\qquad \mid \quad\ ::\ z \ \rightarrow\ \textbf{match}\ z\ \textbf{with}\ (x_1, x_2) \\
 & & \qquad\qquad\qquad\qquad\quad \rightarrow\ \ g\ x_1 :: \boxed{\mathsf{map}\ g}\ x_2
\end{array}
$$

The inner occurrence of $\mathsf{map}\ g$ can be replaced with h

$\Longrightarrow$ fold-Transformation :-)

$$
\begin{array}{lll}
\mathsf{h} & = & \textbf{let } g = \textbf{fun } x \rightarrow x \cdot x \\
 & & \textbf{in fun } x \rightarrow \textbf{match } x \\
 & & \qquad \textbf{with } [\,] \rightarrow [\,] \\
 & & \qquad \mid\ \ :: z \rightarrow \textbf{match } z \textbf{ with } (x_1, x_2) \\
 & & \qquad\qquad\qquad \rightarrow g\ x_1 :: \mathsf{h}\ x_2
\end{array}
$$

Inlining the function g yields:

$$
\begin{array}{lll}
\mathsf{h} & = & \mathbf{let}\ g = \mathbf{fun}\ x\ \to\ x \cdot x \\
 & & \mathbf{in}\ \ \mathbf{fun}\ x\ \to\ \mathbf{match}\ x \\
 & & \qquad\quad \mathbf{with}\ \ [\,]\ \to\ \ [\,] \\
 & & \qquad\qquad\ \mid\ \quad ::\ z\ \to\ \mathbf{match}\ z\ \mathbf{with}\ (x_1, x_2) \\
 & & \qquad\qquad\qquad\qquad\qquad\ \to\ \big(\,\mathbf{let}\ x = x_1 \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\ \mathbf{in}\ \ x * x\,\big)\ ::\ \mathsf{h}\ x_2
\end{array}
$$

Removing useless definitions and variable-variable assignments yields:

$$
\begin{array}{lllll}
\mathsf{h} & = & \textbf{fun}\ x \to \textbf{match}\ x & & \\
& & \quad\ \textbf{with} & [\,] \to & [\,] \\
& & \quad\ \ \ \ \ | & ::\ z \to & \textbf{match}\ z\ \textbf{with}\ (x_1, x_2) \\
& & & & \to x_1 * x_1 :: \mathsf{h}\ x_2
\end{array}
$$

4.5 Deforestation

- Functional programmers love to collect intermediate results in lists which are processed by higher-order functions.
- Examples of such higher-order functions are:

$$
\begin{array}{lllllll}
\mathsf{map} & = & \mathbf{fun}\ f \ \rightarrow & \mathbf{fun}\ l \ \rightarrow & \mathbf{match}\ l\ \mathbf{with}\ [\,] & \rightarrow & [\,] \\
 & & & & \mid\ \ ::\ z & \rightarrow & (\mathbf{match}\ z\ \mathbf{with}\ (x, xs)\ \rightarrow \\
 & & & & & & \quad f\ x :: \mathsf{map}\ f\ xs)
\end{array}
$$

$$
\begin{aligned}
\textsf{filter} \quad = \quad & \textbf{fun}\ p \rightarrow \quad \textbf{fun}\ l \rightarrow \textbf{match}\ l\ \textbf{with}\ [\,] \rightarrow [\,] \\
& \qquad\qquad\quad \mid\ ::z \rightarrow (\textbf{match}\ z\ \textbf{with}\ (x, xs) \rightarrow \\
& \qquad\qquad\qquad\qquad\qquad \textbf{if}\ p\ x\ \textbf{then}\ x :: \textsf{filter}\ p\ xs \\
& \qquad\qquad\qquad\qquad\qquad \textbf{else}\ \textsf{filter}\ p\ xs)
\end{aligned}
$$

$$
\begin{aligned}
\textsf{foldl} \quad = \quad & \textbf{fun}\ f \rightarrow \quad \textbf{fun}\ a \rightarrow \textbf{fun}\ l \rightarrow \textbf{match}\ l\ \textbf{with}\ [\,] \rightarrow a \\
& \qquad\qquad\quad \mid ::z \rightarrow (\textbf{match}\ z\ \textbf{with}\ (x, xs) \rightarrow \\
& \qquad\qquad\qquad\qquad\quad \textsf{foldl}\ f\ (f\ a\ x)\ xs)
\end{aligned}
$$

$$\mathsf{id} \quad = \quad \textbf{fun}\ x \ \rightarrow\ x$$

$$\mathsf{comp} \quad = \quad \textbf{fun}\ f \ \rightarrow\ \textbf{fun}\ g \ \rightarrow\ \textbf{fun}\ x \ \rightarrow\ f\ (g\ x)$$

$$\begin{aligned} \mathsf{comp}_1 \quad = \quad & \textbf{fun}\ f \ \rightarrow\ \textbf{fun}\ g \ \rightarrow\ \textbf{fun}\ x_1 \ \rightarrow\ \textbf{fun}\ x_2 \ \rightarrow \\ & \quad f\ (g\ x_1)\ x_2 \end{aligned}$$

$$\begin{aligned} \mathsf{comp}_2 \quad = \quad & \textbf{fun}\ f \ \rightarrow\ \textbf{fun}\ g \ \rightarrow\ \textbf{fun}\ x_1 \ \rightarrow\ \textbf{fun}\ x_2 \ \rightarrow \\ & \quad f\ x_1\ (g\ x_2) \end{aligned}$$

Example:

$$\begin{array}{lcl}
\mathsf{sum} & = & \mathsf{foldl}\ (+)\ 0 \\
\mathsf{length} & = & \begin{array}{l} \textbf{let}\ \ f \ = \ \mathsf{map}\ (\textbf{fun}\ x \to 1) \\ \textbf{in}\ \ \mathsf{comp}\ \mathsf{sum}\ f \end{array} \\
\mathsf{dev} & = & \textbf{fun}\ l \ \to \ \begin{array}{l} \textbf{let}\ \begin{array}{lcl} s_1 & = & \mathsf{sum}\ l \\ n & = & \mathsf{length}\ l \\ mean & = & s_1 / n \\ l_1 & = & \mathsf{map}\ (\textbf{fun}\ x \to x - mean)\ l \\ l_2 & = & \mathsf{map}\ (\textbf{fun}\ x \to x \cdot x)\ l_1 \\ s_2 & = & \mathsf{sum}\ l_2 \end{array} \\ \textbf{in}\ \ s_2 / n \end{array}
\end{array}$$

Observations:

- Explicit recursion does no longer occur!
- The implementation creates unnecessary intermediate data-structures!

  length could also be implemented as:

$$\begin{array}{lll} \mathsf{length} & = & \textbf{let}\ f\ =\ \textbf{fun}\ a \rightarrow \textbf{fun}\ x \rightarrow a+1 \\ & & \textbf{in}\ \ \mathsf{foldl}\ f\ 0 \end{array}$$

- This implementation avoids to create intermediate lists !!!

Simplification Rules:

$$\begin{array}{lcl}
\mathsf{comp}\ \mathsf{id}\ f & = & \mathsf{comp}\ f\ \mathsf{id} \quad = \quad f \\
\mathsf{comp}_1\ f\ \mathsf{id} & = & \mathsf{comp}_2\ f\ \mathsf{id} \quad = \quad f \\
\mathsf{map}\ \mathsf{id} & = & \mathsf{id} \\
\mathsf{comp}\ (\mathsf{map}\ f)\ (\mathsf{map}\ g) & = & \mathsf{map}\ (\mathsf{comp}\ f\ g) \\
\mathsf{comp}\ (\mathsf{foldl}\ f\ a)\ (\mathsf{map}\ g) & = & \mathsf{foldl}\ (\mathsf{comp}_2\ f\ g)\ a
\end{array}$$

Simplification Rules:

$$
\begin{array}{lcl}
\mathsf{comp}\;\mathsf{id}\;f & = & \mathsf{comp}\;f\;\mathsf{id} \quad = \quad f \\
\mathsf{comp}_1\;f\;\mathsf{id} & = & \mathsf{comp}_2\;f\;\mathsf{id} \quad = \quad f \\
\mathsf{map}\;\mathsf{id} & = & \mathsf{id} \\
\mathsf{comp}\;(\mathsf{map}\;f)\;(\mathsf{map}\;g) & = & \mathsf{map}\;(\mathsf{comp}\;f\;g) \\
\mathsf{comp}\;(\mathsf{foldl}\;f\;a)\;(\mathsf{map}\;g) & = & \mathsf{foldl}\;(\mathsf{comp}_2\;f\;g)\;a \\
\mathsf{comp}\;(\mathsf{filter}\;p_1)\;(\mathsf{filter}\;p_2) & = & \mathsf{filter}\;(\mathbf{fun}\;x \;\to\; \begin{array}[t]{l}\mathbf{if}\;p_2\;x\;\mathbf{then}\;p_1\;x \\ \mathbf{else}\;\mathsf{false})\end{array} \\
\mathsf{comp}\;(\mathsf{foldl}\;f\;a)\;(\mathsf{filter}\;p) & = & \mathbf{let}\;h = \mathbf{fun}\;a \to \mathbf{fun}\;x \to \begin{array}[t]{l}\mathbf{if}\;p\;x\;\mathbf{then}\;f\;a\;x \\ \mathbf{else}\;a\end{array} \\
 & & \mathbf{in}\;\;\mathsf{foldl}\;h\;a
\end{array}
$$

Warning:

Function compositions also could occur as nested function calls ...

$$
\begin{array}{lcl}
\mathsf{id}\ x & = & x \\
\mathsf{map}\ \mathsf{id}\ l & = & l \\
\mathsf{map}\ f\ (\mathsf{map}\ g\ l) & = & \mathsf{map}\ (\mathsf{comp}\ f\ g)\ l \\
\mathsf{foldl}\ f\ a\ (\mathsf{map}\ g\ l) & = & \mathsf{foldl}\ (\mathsf{comp}_2\ f\ g)\ a\ l \\
\mathsf{filter}\ p_1\ (\mathsf{filter}\ p_2\ l) & = & \mathsf{filter}\ (\mathbf{fun}\ x\ \to\ p_1\ x \wedge p_2\ x)\ l \\
\mathsf{foldl}\ f\ a\ (\mathsf{filter}\ p\ l) & = & \mathbf{let}\ h = \mathbf{fun}\ a \to \mathbf{fun}\ x \to\ \begin{array}{l}\mathbf{if}\ p\ x\ \mathbf{then}\ f\ a\ x \\ \mathbf{else}\ a\end{array} \\
& & \mathbf{in}\ \ \mathsf{foldl}\ h\ a\ l
\end{array}
$$

Example, optimized:

$$
\begin{array}{lcl}
\mathsf{sum} & = & \mathsf{foldl}\ (+)\ 0 \\
\mathsf{length} & = & \ \ \textbf{let}\ \ f\ \ =\ \ \mathsf{comp}_2\ (+)\ (\textbf{fun}\ x \to 1) \\
 & & \ \ \textbf{in}\ \ \mathsf{foldl}\ f\ 0 \\
\mathsf{dev} & = & \textbf{fun}\ l\ \ \longrightarrow\ \ \begin{array}[t]{l} \textbf{let}\ \begin{array}[t]{lcl} s_1 & = & \mathsf{sum}\ l \\ n & = & \mathsf{length}\ l \\ mean & = & s_1/n \\ f & = & \mathsf{comp}\ \ (\textbf{fun}\ x \longrightarrow x \cdot x) \\ & & \qquad\quad\ (\textbf{fun}\ x \longrightarrow x - mean) \\ g & = & \mathsf{comp}_2\ (+)\ f \\ s_2 & = & \mathsf{foldl}\ g\ 0\ l \end{array} \\ \textbf{in}\ \ s_2/n \end{array}
\end{array}
$$

Remarks:

- All intermediate lists have disappeared :-)
- Only foldl remain — i.e., loops :-))
- Compositions of functions can be further simplified in the next step by Inlining.
- Inside dev, we then obtain:

$$\begin{array}{lllllllll} g & = & \textbf{fun}\ a & \rightarrow & \textbf{fun}\ x & \rightarrow & \textbf{let} & x_1 & = & x - \mathit{mean} \\ & & & & & & & x_2 & = & x_1 \cdot x_1 \\ & & & & & & \textbf{in} & \multicolumn{3}{l}{a + x_2} \end{array}$$

- The result is a sequence of **let**-definitions !!!

Extension: Tabulation

If the list has been created by tabulation of a function, the creation of the list sometimes can be avoided ...

$$\begin{array}{lcl} \mathsf{tabulate}' & = & \textbf{fun}\ j \ \rightarrow\ \ \textbf{fun}\ f\ \rightarrow\ \textbf{fun}\ n\ \rightarrow \\ & & \qquad\qquad\ \textbf{if}\ j \geq n\ \textbf{then}\ [\,] \\ & & \qquad\qquad\ \textbf{else}\ (f\ j) :: \mathsf{tabulate}'\ (j+1)\ f\ n \\ \mathsf{tabulate} & = & \mathsf{tabulate}'\ 0 \end{array}$$

Then we have:

$$\begin{array}{lcl}
\mathsf{comp}\ (\mathsf{map}\ f)\ (\mathsf{tabulate}\ g) & = & \mathsf{tabulate}\ (\mathsf{comp}\ f\ g) \\
\mathsf{comp}\ (\mathsf{foldl}\ f\ a)\ (\mathsf{tabulate}\ g) & = & \mathsf{loop}\ (\mathsf{comp}_2\ f\ g)\ a
\end{array}$$

where:

$$\begin{array}{lcll}
\mathsf{loop}' & = & \textbf{fun}\ j\ \rightarrow & \textbf{fun}\ f\ \rightarrow\ \textbf{fun}\ a\ \rightarrow\ \textbf{fun}\ n\ \rightarrow \\
 & & & \textbf{if}\ j \geq n\ \textbf{then}\ a \\
 & & & \textbf{else}\ \mathsf{loop}'\ (j+1)\ f\ (f\ a\ j))\ n \\
\mathsf{loop} & = & \mathsf{loop}'\ 0 &
\end{array}$$

Extension (2): List Reversals

Sometimes, the ordering of lists or arguments is reversed:

$$
\begin{array}{lcl}
\mathsf{rev}' & = & \textbf{fun}\, a \;\rightarrow\; \textbf{fun}\, l \;\rightarrow \\
 & & \qquad\qquad \textbf{match}\; l\; \textbf{with}\; [\,] \;\rightarrow\; a \\
 & & \qquad\qquad \mid\;\; ::z \;\rightarrow\; \big(\textbf{match}\; z\; \textbf{with}\; (x, xs) \;\rightarrow \\
 & & \qquad\qquad\qquad\qquad\quad \mathsf{rev}'\; (x :: a)\; xs\big) \\
\mathsf{rev} & = & \mathsf{rev}'\; [\,] \\
 & & \\
\mathsf{comp\ rev\ rev} & = & \mathsf{id} \\
 & & \\
\mathsf{swap} & = & \textbf{fun}\; f \;\rightarrow\; \textbf{fun}\; x \;\rightarrow\; \textbf{fun}\; y \;\rightarrow\; f\; y\; x \\
 & & \\
\mathsf{comp\ swap\ swap} & = & \mathsf{id}
\end{array}
$$

$$\mathsf{foldr}\ f\ a \quad = \quad \mathsf{comp}\ (\mathsf{foldl}\ (\mathsf{swap}\ f)\ a)\ \mathsf{rev}$$

Discussion:

- The standard implementation of foldr is not tail-recursive.
- The last equation decomposes a foldr into two tail-recursive functions — at the price that an intermediate list is created.
- Therefore, the standard implementation is probably faster :-)
- Sometimes, the operation rev can also be optimized away ...

We have:

$$\begin{aligned}
\mathsf{comp}\ \mathsf{rev}\ (\mathsf{map}\ f) &= \mathsf{comp}\ (\mathsf{map}\ f)\ \mathsf{rev} \\
\mathsf{comp}\ \mathsf{rev}\ (\mathsf{filter}\ p) &= \mathsf{comp}\ (\mathsf{filter}\ p)\ \mathsf{rev} \\
\mathsf{comp}\ \mathsf{rev}\ (\mathsf{tabulate}\ f) &= \mathsf{rev\_tabulate}\ f
\end{aligned}$$

Here, rev_tabulate tabulates in reverse ordering. This function has properties quite analogous to tabulate:

$$\begin{aligned}
\mathsf{comp}\ (\mathsf{map}\ f)\ (\mathsf{rev\_tabulate}\ g) &= \mathsf{rev\_tabulate}\ (\mathsf{comp}_2\ f\ g) \\
\mathsf{comp}\ (\mathsf{foldl}\ f\ a)\ (\mathsf{rev\_tabulate}\ g) &= \mathsf{rev\_loop}\ (\mathsf{comp}_2\ f\ g)\ a
\end{aligned}$$

Extension (3): Dependencies on the Index

- Correctness is proven by induction on the lengthes of occurring lists.
- Similar composition results also hold for transformations which take the current indices into account:

$$\begin{aligned}
\mathsf{mapi}' \quad &= \quad \textbf{fun}\ i \rightarrow \quad \textbf{fun}\ f \ \rightarrow\ \textbf{fun}\ l \ \rightarrow\ \textbf{match}\ l\ \textbf{with}\ [\,] \ \rightarrow \quad [\,] \\
&\qquad\qquad\qquad \mid\ \ ::\ z \ \rightarrow\ (\textbf{match}\ z\ \textbf{with}\ (x, xs)\ \rightarrow \\
&\qquad\qquad\qquad\qquad (f\ i\ x) :: \mathsf{mapi}'\ (i+1)\ f\ \ xs) \\
\mathsf{mapi} \quad &= \quad \mathsf{mapi}'\ 0
\end{aligned}$$

Analogously, there is index-dependent accumulation:

$$\begin{array}{lcl}
\mathsf{foldli}' & = & \textbf{fun } i \;\to\; \textbf{fun } f \;\to\; \textbf{fun } a \;\to\; \textbf{fun } l \;\to \\
& & \qquad\qquad\quad \textbf{match } l \textbf{ with } [\,] \;\to\; a \\
& & \qquad\qquad\quad \mid\; :: z \;\to\; (\textbf{match } z \textbf{ with } (x, xs) \;\to \\
& & \qquad\qquad\qquad\quad \mathsf{foldli}'\; (i+1)\; f\; (f\; i\; a\; x)\; xs) \\
\mathsf{foldli} & = & \mathsf{foldli}'\; 0
\end{array}$$

For composition, we must take care that always the same indices are used. This is achieved by:

$$\mathsf{compi} \;=\; \textbf{fun}\ f \rightarrow \textbf{fun}\ g \rightarrow \textbf{fun}\ i \rightarrow \textbf{fun}\ x \rightarrow f\ i\ (g\ i\ x)$$

$$\mathsf{compi}_1 \;=\; \textbf{fun}\ f \rightarrow \textbf{fun}\ g \rightarrow \textbf{fun}\ i \rightarrow \textbf{fun}\ x_1 \rightarrow \textbf{fun}\ x_2 \rightarrow f\ i\ (g\ i\ x_1)\ x_2$$

$$\mathsf{compi}_2 \;=\; \textbf{fun}\ f \rightarrow \textbf{fun}\ g \rightarrow \textbf{fun}\ i \rightarrow \textbf{fun}\ x_1 \rightarrow \textbf{fun}\ x_2 \rightarrow f\ i\ x_1\ (g\ i\ x_2)$$

$$\mathsf{cmp}_1 \;=\; \textbf{fun}\ f \rightarrow \textbf{fun}\ g \rightarrow \textbf{fun}\ i \rightarrow \textbf{fun}\ x_1 \rightarrow \textbf{fun}\ x_2 \rightarrow f\ i\ x_1\ (g\ x_2)$$

$$\mathsf{cmp}_2 \;=\; \textbf{fun}\ f \rightarrow \textbf{fun}\ g \rightarrow \textbf{fun}\ i \rightarrow \textbf{fun}\ x_1 \rightarrow \textbf{fun}\ x_2 \rightarrow f\ x_1\ (g\ i\ x_2)$$

Then:

$$\begin{array}{lcl}
\mathsf{comp}\,(\mathsf{mapi}\ f)\ (\mathsf{map}\ g) & = & \mathsf{mapi}\,(\mathsf{comp}_2\ f\ g) \\
\mathsf{comp}\,(\mathsf{map}\ f)\ (\mathsf{mapi}\ g) & = & \mathsf{mapi}\,(\mathsf{comp}\ f\ g) \\
\mathsf{comp}\,(\mathsf{mapi}\ f)\ (\mathsf{mapi}\ g) & = & \mathsf{mapi}\,(\mathsf{compi}\ f\ g) \\
\mathsf{comp}\,(\mathsf{foldli}\ f\ a)\ (\mathsf{map}\ g) & = & \mathsf{foldli}\,(\mathsf{cmp}_1\ f\ g)\ a \\
\mathsf{comp}\,(\mathsf{foldl}\ f\ a)\ (\mathsf{mapi}\ g) & = & \mathsf{foldli}\,(\mathsf{cmp}_2\ f\ g)\ a \\
\mathsf{comp}\,(\mathsf{foldli}\ f\ a)\ (\mathsf{mapi}\ g) & = & \mathsf{foldli}\,(\mathsf{compi}_2\ f\ g)\ a \\
\mathsf{comp}\,(\mathsf{foldli}\ f\ a)\ (\mathsf{tabulate}\ g) & = & \mathbf{let}\ h = \ \mathbf{fun}\ a\ \rightarrow\ \mathbf{fun}\ i\ \rightarrow \\
 & & \qquad\quad f\ i\ a\ (g\,i) \\
 & & \mathbf{in}\ \ \mathsf{loop}\ h\ a
\end{array}$$

Discussion:

- Warning: index-dependent transformations may not commute with rev or filter.
- All our rules can only be applied if the functions id, map, mapi, foldl, foldli, filter, rev, tabulate, rev_tabulate, loop, rev_loop, ... are provided by a standard library: Only then the algebraic properties can be guaranteed !!!
- Similar simplification rules can be derived for any kind of tree-like data-structure tree α .
- These also provide operations map, mapi and foldl, foldli with corresponding rules.
- Further opportunities are opened up by functions to_list and from_list ...

Example

$$
\begin{array}{lcl}
\mathbf{type}\ \mathsf{tree}\ \alpha & = & \mathsf{Leaf} \mid \mathsf{Node}\ \alpha\ (\mathsf{tree}\ \alpha)\ (\mathsf{tree}\ \alpha) \\
\mathsf{map} & = & \mathbf{fun}\ f \rightarrow \mathbf{fun}\ t \rightarrow \mathbf{match}\ t\ \mathbf{with}\ \mathsf{Leaf} \rightarrow \mathsf{Leaf} \\
 & & \qquad\qquad \mid\ \mathsf{Node}\ x\ l\ r \rightarrow \mathbf{let}\ l' = \mathsf{map}\ f\ l \\
 & & \qquad\qquad\qquad\qquad\qquad\quad r' = \mathsf{map}\ f\ r \\
 & & \qquad\qquad\qquad\qquad\quad \mathbf{in}\ \mathsf{Node}\ (f\ x)\ l'\ r'
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{foldl} & = & \mathbf{fun}\ f \rightarrow \mathbf{fun}\ a \rightarrow \mathbf{fun}\ t \rightarrow \mathbf{match}\ t\ \mathbf{with}\ \mathsf{Leaf} \rightarrow a \\
 & & \qquad\qquad \mid \mathsf{Node}\ x\ l\ r \rightarrow \mathbf{let}\ a' = \mathsf{foldl}\ f\ a\ l \\
 & & \qquad\qquad\qquad\qquad\quad \mathbf{in}\ \mathsf{foldl}\ f\ (f\ a'\ x)\ r
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{to\_list}' & = & \mathbf{fun}\, a \;\rightarrow\; \mathbf{fun}\, t \;\rightarrow\; \mathbf{match}\; t \;\mathbf{with}\; \mathsf{Leaf} \;\rightarrow\; a \\
 & & \mid\; \mathsf{Node}\; x\; t_1\; t_2 \;\rightarrow\; \mathbf{let}\;\; a' \;=\; \mathsf{to\_list}'\; a\; t_2 \\
 & & \qquad\qquad\qquad\qquad\quad \mathbf{in}\;\; \mathsf{to\_list}'\; (x :: a')\; t_1 \\
\mathsf{to\_list} & = & \mathsf{to\_list}'\; [\,]
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{from\_list} & = & \mathbf{fun}\, l \;\rightarrow \\
 & & \mathbf{match}\; l \;\mathbf{with}\; [\,] \;\rightarrow\; \mathsf{Leaf} \\
 & & \mid\;\; ::\, z \;\rightarrow\; (\mathbf{match}\; z \;\mathbf{with}\; (x, xs) \;\rightarrow \\
 & & \qquad\qquad\quad \mathsf{Node}\; x\; \mathsf{Leaf}\; (\mathsf{from\_list}\; xs))
\end{array}
$$

Warning:

Not every natural equation is valid:

$$\begin{aligned}
\mathsf{comp\ to\_list\ from\_list} &= \mathsf{id}\\
\mathsf{comp\ from\_list\ to\_list} &\neq \mathsf{id}\\
\mathsf{comp\ to\_list}\ (\mathsf{map}\ f) &= \mathsf{comp}\ (\mathsf{map}\ f)\ \mathsf{to\_list}\\
\mathsf{comp\ from\_list}\ (\mathsf{map}\ f) &= \mathsf{comp}\ (\mathsf{map}\ f)\ \mathsf{from\_list}\\
\mathsf{comp}\ (\mathsf{foldl}\ f\ a)\ \mathsf{to\_list} &= \mathsf{foldl}\ f\ a\\
\mathsf{comp}\ (\mathsf{foldl}\ f\ a)\ \mathsf{from\_list} &= \mathsf{foldl}\ f\ a
\end{aligned}$$

In this case, there is even a rev:

$$\begin{array}{lcl}
\mathsf{rev} & = & \textbf{fun}\ t\ \rightarrow \\
 & & \textbf{match}\ t\ \textbf{with}\ \mathsf{Leaf}\ \rightarrow\ \mathsf{Leaf} \\
 & & \mid\ \mathsf{Node}\ x\ t_1\ t_2\ \rightarrow\ \begin{array}[t]{ll} \textbf{let} & \begin{array}[t]{lcl} s_1 & = & \mathsf{rev}\ t_1 \\ s_2 & = & \mathsf{rev}\ t_2 \end{array} \\ \textbf{in} & \mathsf{Node}\ x\ s_2\ s_1 \end{array}
\end{array}$$

$$\begin{array}{lcl}
\mathsf{comp\ to\_list\ rev} & = & \mathsf{comp\ rev\ to\_list} \\
\mathsf{comp\ from\_list\ rev} & \neq & \mathsf{comp\ rev\ from\_list}
\end{array}$$

4.6 CBN vs. CBV: Strictness Analysis

Problem:

- Programming languages such as Haskell evaluate expressions for **let**-defined variables and actual parameters not before their values are accessed.
- This allows for an elegant treatment of (possibly) infinite lists of which only small initial segments are required for computing the result :-)
- Delaying evaluation by default incures, though, a non-trivial overhead ...

Example

$$\mathsf{from} \;=\; \textbf{fun}\ n \;\to\; n :: \mathsf{from}\ (n+1)$$

$$\begin{array}{ll} \mathsf{take} \;=\; \textbf{fun}\ k \to \textbf{fun}\ s \to & \textbf{if}\ k \leq 0\ \textbf{then}\ [\,] \\ & \textbf{else}\ \ \textbf{match}\ s\ \textbf{with}\ [\,] \to [\,] \\ & \qquad \mid\ :: z \to \ \textbf{match}\ z\ \textbf{with}\ (x, xs) \to \\ & \qquad\qquad x :: \mathsf{take}\ (k-1)\ xs \end{array}$$

Then CBN yields:

$$\mathsf{take}\ 5\ (\mathsf{from}\ 0) = [0, 1, 2, 3, 4]$$

— whereas evaluation with CBV does not terminate !!!

Then CBN yields:

$$\mathsf{take}\ 5\ (\mathsf{from}\ 0) = [0, 1, 2, 3, 4]$$

— whereas evaluation with CBV does not terminate !!!

On the other hand, for CBN, tail-recursive functions may require non-constant space ???

$$\begin{array}{lllllll} \mathsf{fac2} & = & \mathbf{fun}\ x & \rightarrow & \mathbf{fun}\ a & \rightarrow & \mathbf{if}\ x \leq 0\ \mathbf{then}\ a \\ & & & & & & \mathbf{else}\ \ \mathsf{fac2}\ (x-1)\ (a \cdot x) \end{array}$$

Discussion:

- The multiplications are collected in the accumulating parameter through nested closures.
- Only when the value of a call fac2 x 1 is accessed, this dynamic data structure is evaluated.
- Instead, the accumulating parameter should have been passed directly by-value !!!
- This is the goal of the following optimization ...

Simplification:

- At first, we rule out data structures, higher-order functions, and local function definitions.
- We introduce an unary operator # which forces the evaluation of a variable.
- Goal of the transformation is to place # at as many places as possible ...

Simplification:

- At first, we rule out data structures, higher-order functions, and local function definitions.
- We introduce an unary operator # which forces the evaluation of a variable.
- Goal of the transformation is to place # at as many places as possible ...

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid e_1 \,\Box_2\, e_2 \mid \Box_1\, e \mid f\ e_1\ \ldots\ e_k \mid \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \\
 & & \mid \textbf{let}\ r_1 = e_1\ \textbf{in}\ e \\
r & ::= & x \mid \#x \\
d & ::= & f\ x_1\ \ldots\ x_k = e \\
p & ::= & \textbf{letrec and}\ d_1\ \ldots\ \textbf{and}\ d_n\ \textbf{in}\ e
\end{array}
$$

Idea:

- Describe a k-ary function

$$f : \textbf{int} \to \ldots \to \textbf{int}$$

by a function

$$[\![f]\!]^\sharp : \mathbb{B} \to \ldots \to \mathbb{B}$$

- 0 means: evaluation does definitely not terminate.
- 1 means: evaluation may terminate.
- $[\![f]\!]^\sharp\ 0 = 0$ means: If the function call returns a value, then the evaluation of the argument must have terminated and returned a value.

$\Longrightarrow$ f is strict.

Idea (cont.):

- We determine the abstract semantics of all functions :-)
- For that, we put up a system of equations ...

Auxiliary Function:

$$
\begin{array}{lcl}
[\![e]\!]^\sharp & : & (\mathit{Vars} \to \mathbb{B}) \to \mathbb{B} \\
[\![c]\!]^\sharp\ \rho & = & 1 \\
[\![x]\!]^\sharp\ \rho & = & \rho\, x \\
[\![\square_1\ e]\!]^\sharp\ \rho & = & [\![e]\!]^\sharp\ \rho \\
[\![e_1\ \square_2\ e_2]\!]^\sharp\ \rho & = & [\![e_1]\!]^\sharp\ \rho \wedge [\![e_2]\!]^\sharp\ \rho \\
[\![\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!]^\sharp\ \rho & = & [\![e_0]\!]^\sharp\ \rho \wedge ([\![e_1]\!]^\sharp\ \rho \vee [\![e_2]\!]^\sharp\ \rho) \\
[\![f\ e_1\ \ldots\ e_k]\!]^\sharp\ \rho & = & [\![f]\!]^\sharp\ ([\![e_1]\!]^\sharp\ \rho)\ \ldots\ ([\![e_k]\!]^\sharp\ \rho) \\
\ldots & &
\end{array}
$$

$$
\begin{array}{lcl}
[\![\textbf{let } x_1 = e_1 \textbf{ in } e]\!]^\sharp\,\rho & = & [\![e]\!]^\sharp\,(\rho \oplus \{x_1 \mapsto [\![e_1]\!]^\sharp\,\rho\}) \\
[\![\textbf{let } \#x_1 = e_1 \textbf{ in } e]\!]^\sharp\,\rho & = & ([\![e_1]\!]^\sharp\,\rho) \wedge ([\![e]\!]^\sharp\,(\rho \oplus \{x_1 \mapsto 1\}))
\end{array}
$$

System of Equations:

$$[\![f_i]\!]^\sharp b_1 \;\ldots\; b_k = [\![e_i]\!]^\sharp\,\{x_j \mapsto b_j \mid j = 1,\ldots,k\}, \qquad i = 1,\ldots,n, b_1,\ldots,b_k \in \mathbb{B}$$

- The unkowns of the system of equations are the functions $[\![f_i]\!]^\sharp$ or the individual entries $[\![f_i]\!]^\sharp b_1 \;\ldots\; b_k$ in the value table.
- All right-hand sides are monotonic!
- Consequently, there is a least solution :-)
- The complete lattice $\mathbb{B} \to \ldots \to \mathbb{B}$ has height $\mathcal{O}(2^k)$:-(

Example:

For fac2, we obtain:

$$\begin{array}{lll} \llbracket \mathsf{fac2} \rrbracket^\sharp\; b_1\; b_2 & = & b_1 \wedge (b_2 \vee \\ & & \llbracket \mathsf{fac2} \rrbracket^\sharp\; b_1\; (b_1 \wedge b_2)) \end{array}$$

Fixpoint iteration yields:

0	$\mathbf{fun}\, x \to \mathbf{fun}\, a \to 0$
1	$\mathbf{fun}\, x \to \mathbf{fun}\, a \to x \wedge a$
2	$\mathbf{fun}\, x \to \mathbf{fun}\, a \to x \wedge a$

We conclude:

- The function fac2 is strict in both arguments, i.e., if evaluation terminates, then also the evaluation of its arguments.
- Accordingly, we transform:

$$
\begin{array}{lllllll}
\text{fac2} & = & \textbf{fun } x & \rightarrow & \textbf{fun } a & \rightarrow & \textbf{if } x \leq 0 \textbf{ then } a \\
 & & & & & & \textbf{else let } \begin{array}{lll} \# x' & = & x - 1 \\ \# a' & = & x \cdot a \end{array} \\
 & & & & & & \textbf{in } \text{fac2 } x' \; a'
\end{array}
$$

Correctness of the Analysis:

- The system of equations is an abstract denotational semantics.
- The denotational semantics characterizes the meaning of functions as least solution of the corresponding equations for the concrete semantics.
- For values, the denotational semantics relies on the complete partial ordering $\mathbb{Z}_\perp$.
- For complete partial orderings, Kleene's fixpoint theorem is applicable :-)
- As description relation Δ we use:

$$\perp \Delta 0 \quad \text{and} \quad z \Delta 1 \quad \text{für } z \in \mathbb{Z}$$

Extension: Data Structures

- Functions may vary in the parts which they require from a data structure ...

$$\begin{array}{lllll} \mathsf{hd} & = & \textbf{fun}\ l & \rightarrow & \textbf{match}\ l\ \textbf{with}\ ::z\ \rightarrow \\ & & & & \textbf{match}\ z\ \textbf{with}\ (x, xs)\ \rightarrow\ x \end{array}$$

- hd only accesses the first element of a list.
- length only accesses the backbone of its argument.
- rev forces the evaluation of the complete argument — given that the result is required completely ...

Extension of the Syntax:

We additionally consider expression of the form:

$$\begin{array}{lllll} e & ::= & \ldots & \mid & [\,] \mid\, ::\, e \mid \textbf{match}\ e_0\ \textbf{with}\ [\,]\ \rightarrow\ e_1\ \mid\ ::\, z\ \rightarrow\ e_2 \\ & & & \mid & (e_1, e_2) \mid \textbf{match}\ e_0\ \textbf{with}\ (x_1, x_2)\ \rightarrow\ e_1 \end{array}$$

Top Strictness

- We assume that the program is well-typed.
- We are only interested in top constructors.
- Again, we model this property with (monotonic) Boolean functions.
- For **int**-values, this coincides with strictness :-)
- We extend the abstract evaluation $[\![e]\!]^\sharp\ \rho$ with rules for case-distinction ...

$$\begin{aligned}
\llbracket \textbf{match } e_0 \textbf{ with } [\,] \;\to\; e_1 \;\mid\; ::z \;\to\; e_2 \rrbracket^\sharp\,\rho \;&=\; \\
& \llbracket e_0 \rrbracket^\sharp\,\rho \wedge (\llbracket e_1 \rrbracket^\sharp\,\rho \vee \llbracket e_2 \rrbracket^\sharp\,(\rho \oplus \{z \mapsto 1\})) \\
\llbracket \textbf{match } e_0 \textbf{ with } (x_1, x_2) \;\to\; e_1 \rrbracket^\sharp\,\rho \;&=\; \\
& \llbracket e_0 \rrbracket^\sharp\,\rho \wedge \llbracket e_1 \rrbracket^\sharp\,(\rho \oplus \{x_1, x_2 \mapsto 1\}) \\
\llbracket [\,] \rrbracket^\sharp\,\rho \;=\; \llbracket ::\, e \rrbracket^\sharp\,\rho \;=\; \llbracket (e_1, e_2) \rrbracket^\sharp\,\rho \;&=\; 1
\end{aligned}$$

- The rules for **match** are analogous to those for **if**.
- In case of ::, we know nothing about the values beneath the constructor; therefore $\{z \mapsto 1\}$.
- We check our analysis on the function app ...

Example:

$$\begin{array}{lllllll} \mathsf{app} & = & \mathbf{fun}\ x & \rightarrow & \mathbf{fun}\ y & \rightarrow & \mathbf{match}\ x\ \mathbf{with}\ [\,] \ \rightarrow \ y \\ & & & \mid & ::z & \rightarrow & \mathbf{match}\ z\ \mathbf{with}\ (x, xs) \ \rightarrow :: (x, \mathsf{app}\ xs\ y) \end{array}$$

Abstract interpretation yields the system of equations:

$$\begin{array}{lll} [\![\mathsf{app}]\!]^\sharp\ b_1\ b_2 & = & b_1 \wedge (b_2 \vee 1) \\ & = & b_1 \end{array}$$

We conclude that we may conclude for sure only for the first argument that its top constructor is required :-)

Total Strictness

Assume that the result of the function application is totally required. Which arguments then are also totally required ?

We again refer to Boolean functions ...

$$
\begin{array}{lll}
[\![\textbf{match}\ e_0\ \textbf{with}\ [\,]\ \rightarrow\ e_1\ \mid\ ::z\ \rightarrow\ e_2]\!]^\sharp\,\rho & = & [\![e_0]\!]^\sharp\,\rho \wedge [\![e_1]\!]^\sharp\,\rho \\
 & & \vee\ [\![e_2]\!]^\sharp\,(\rho \oplus \{z \mapsto [\![e_0]\!]^\sharp\,\rho\}) \\
[\![\textbf{match}\ e_0\ \textbf{with}\ (x_1, x_2)\ \rightarrow\ e_1]\!]^\sharp\,\rho & = & \textbf{let}\ b = [\![e_0]\!]^\sharp\,\rho \\
\multicolumn{3}{r}{\textbf{in}\ [\![e_1]\!]^\sharp\,(\rho \oplus \{x_1 \mapsto 1, x_2 \mapsto b\}) \vee [\![e_1]\!]^\sharp\,(\rho \oplus \{x_1 \mapsto b, x_2 \mapsto 1\})} \\
[\![[\,]]\!]^\sharp\,\rho & = & 1 \\
[\![::e]\!]^\sharp\,\rho & = & [\![e]\!]^\sharp\,\rho \\
[\![(e_1, e_2)]\!]^\sharp\,\rho & = & [\![e_1]\!]^\sharp\,\rho \wedge [\![e_2]\!]^\sharp\,\rho
\end{array}
$$

Discussion:

- The rules for constructor applications have changed.
- Also the treatment of **match** now involves the components z and x_1, x_2.
- Again, we check the approach for the function app.

Example:

Abstract interpretation yields the system of equations:

$$\begin{array}{lll} [\![\mathsf{app}]\!]^\sharp\; b_1\; b_2 & = & b_1 \wedge b_2 \vee b_1 \wedge [\![\mathsf{app}]\!]^\sharp\; 1\; b_2 \vee 1 \wedge [\![\mathsf{app}]\!]^\sharp\; b_1\; b_2 \\ & = & b_1 \wedge b_2 \vee b_1 \wedge [\![\mathsf{app}]\!]^\sharp\; 1\; b_2 \vee [\![\mathsf{app}]\!]^\sharp\; b_1\; b_2 \end{array}$$

This results in the following fixpoint iteration:

0	$\textbf{fun}\ x \to \textbf{fun}\ y \to 0$
1	$\textbf{fun}\ x \to \textbf{fun}\ y \to x \wedge y$
2	$\textbf{fun}\ x \to \textbf{fun}\ y \to x \wedge y$

We deduce that both arguments are definitely totally required if the result is totally required :-)

Warning:

Whether or not the result is totally required, depends on the context of the function call!

In such a context, a specialized function may be called ...

$$
\begin{array}{lllll}
\mathsf{app\#} & = & \textbf{fun } x \to \textbf{fun } y \to & \multicolumn{2}{l}{\textbf{let } \#x' = x \textbf{ and } \#y' = y \textbf{ in}} \\
 & & & \multicolumn{2}{l}{\textbf{match } {}'x \textbf{ with } [\,] \;\to\; y'} \\
 & & & \quad \mid \; ::z \;\to & \textbf{match } z \textbf{ with } (x, xs) \;\to \\
 & & & & \textbf{let } \# r = ::\, (x, \mathsf{app\#}\; xs\;\, y) \\
 & & & & \textbf{in } r
\end{array}
$$

Discussion:

- Both strictness analyses employ the same complete lattice.
- Results and application, though, are quite different :-)
- Thereby, we use the following description relations:

  Top Strictness : $\bot \;\Delta\; 0$

  Total Strictness : $z \;\Delta\; 0$ if $\bot$ occurs in z.
- Both analyses can also be combined to an a joint analysis ...

Combined Strictness Analysis

- We use the complete lattice:

$$\mathbb{T} = \{0 \sqsubset 1 \sqsubset 2\}$$

- The description relation is given by:

$$\perp \,\Delta\, 0 \qquad z \,\Delta\, 1 \;(z \text{ contains } \perp) \qquad z \,\Delta\, 2 \;(z \text{ value})$$

- The lattice is more informative, the functions, though, are no longer as efficiently representable, e.g., through Boolean expressions :-(

- We require the auxiliary functions:

$$(i \sqsubseteq x); \; y = \begin{cases} y & \text{if } i \sqsubseteq x \\ 0 & \text{otherwise} \end{cases}$$

The Combined Evaluation Function:

$$\begin{array}{lll}
[\![\textbf{match}\, e_0\, \textbf{with}\, [\,] \;\rightarrow\; e_1 \;|\; ::z \;\rightarrow\; e_2]\!]^\sharp\, \rho & = & \\
\qquad (2 \sqsubseteq [\![e_0]\!]^\sharp\, \rho)\,;\, [\![e_1]\!]^\sharp\, \rho \sqcup (1 \sqsubseteq [\![e_0]\!]^\sharp\, \rho)\,;\, [\![e_2]\!]^\sharp\, (\rho \oplus \{z \mapsto [\![e_0]\!]^\sharp\, \rho\}) & & \\
[\![\textbf{match}\, e_0\, \textbf{with}\, (x_1, x_2) \;\rightarrow\; e_1]\!]^\sharp\, \rho & = & \textbf{let}\; b = [\![e_0]\!]^\sharp\, \rho \\
\qquad \textbf{in}\; (1 \sqsubseteq [\![e_0]\!]^\sharp\, \rho)\,;\, ([\![e_1]\!]^\sharp\, (\rho \oplus \{x_1 \mapsto 2, x_2 \mapsto b\}) & & \\
\qquad \sqcup\, [\![e_1]\!]^\sharp\, (\rho \oplus \{x_1 \mapsto b, x_2 \mapsto 2\})) & & \\
[\![\,[\,]\,]\!]^\sharp\, \rho & = & 2 \\
[\![::\, e]\!]^\sharp\, \rho & = & 1 \sqcup [\![e]\!]^\sharp\, \rho \\
[\![(e_1, e_2)]\!]^\sharp\, \rho & = & 1 \sqcup ([\![e_1]\!]^\sharp\, \rho \sqcap [\![e_2]\!]^\sharp\, \rho)
\end{array}$$

Example:

For our beloved function app, we obtain:

$$
\begin{array}{lll}
[\![\mathsf{app}]\!]^\sharp\; d_1\; d_2 & = & (2 \sqsubseteq d_1)\;;\; d_2 \;\sqcup \\
 & & (1 \sqsubseteq d_1)\;;\; (1 \;\sqcup\; [\![\mathsf{app}]\!]^\sharp\; d_1\; d_2 \;\sqcup\; d_1 \sqcap [\![\mathsf{app}]\!]^\sharp\; 2\; d_2) \\
 & = & (2 \sqsubseteq d_1)\;;\; d_2 \;\sqcup \\
 & & (1 \sqsubseteq d_1)\;;\; 1 \;\sqcup \\
 & & (1 \sqsubseteq d_1)\;;\; [\![\mathsf{app}]\!]^\sharp\; d_1\; d_2 \;\sqcup \\
 & & d_1 \sqcap [\![\mathsf{app}]\!]^\sharp\; 2\; d_2
\end{array}
$$

this results in the fixpoint computation:

0	$\textbf{fun}\ x \to \textbf{fun}\ y \to\ 0$
1	$\textbf{fun}\ x \to \textbf{fun}\ y \to\ (2 \sqsubseteq x);\ y \sqcup (1 \sqsubseteq x);\ 1$
2	$\textbf{fun}\ x \to \textbf{fun}\ y \to\ (2 \sqsubseteq x);\ y \sqcup (1 \sqsubseteq x);\ 1$

We conclude

- that both arguments are totally required if the result is totally required; and
- that the root of the first argument is required if the root of the result is required :-)

Remark:

The analysis can be easily generalized such that it guarantees evaluation up to a depth d ;-)

Further Directions:

- Our Approach is also applicable to other data structures.
- In principle, also higher-order (monomorphic) functions can be analyzed in this way :-)
- Then, however, we require higher-order abstract functions — of which there are many :-(
- Such functions therefore are approximated by:

$$\textbf{fun } x_1 \rightarrow \ldots \textbf{ fun } x_r \rightarrow \top$$

:-)

- For some known higher-order functions such as map, foldl, loop, ... this approach then should be improved :-))

5 Optimization of Logic Programs

We only consider the mini language PuP (“Pure Prolog”). In particular, we do not consider:

- arithmetic;
- the cut-operator.
- Self-modification by means of assert and retract.

Example:

$$
\begin{array}{lcl}
\mathsf{bigger}(X,Y) & \leftarrow & X = \mathit{elephant}, Y = \mathit{horse} \\
\mathsf{bigger}(X,Y) & \leftarrow & X = \mathit{horse}, Y = \mathit{donkey} \\
\mathsf{bigger}(X,Y) & \leftarrow & X = \mathit{donkey}, Y = \mathit{dog} \\
\mathsf{bigger}(X,Y) & \leftarrow & X = \mathit{donkey}, Y = \mathit{monkey} \\
\mathsf{is\_bigger}(X,Y) & \leftarrow & \mathsf{bigger}(X,Y) \\
\mathsf{is\_bigger}(X,Y) & \leftarrow & \mathsf{bigger}(X,Z), \mathsf{is\_bigger}(Z,Y) \\
 & \leftarrow & \mathsf{is\_bigger}(\mathit{elephant}, \mathit{dog})
\end{array}
$$

A more realistic Example:

$$\begin{array}{rcl} \mathsf{app}(X, Y, Z) & \leftarrow & X = [\,],\; Y = Z \\ \mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'],\; Z = [H|Z'],\; \mathsf{app}(X', Y, Z') \\ & \leftarrow & \mathsf{app}(X, [Y, c], [a, b, Z]) \end{array}$$

A more realistic Example:

$$\begin{array}{rcl} \mathsf{app}(X, Y, Z) & \leftarrow & X = [\,], \; Y = Z \\ \mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'], \; Z = [H|Z'], \; \mathsf{app}(X', Y, Z') \\ & \leftarrow & \mathsf{app}(X, [Y, c], [a, b, Z]) \end{array}$$

Remark:

$[\,]$	$==$	the atom empty list
$[H\|Z]$	$==$	binary constructor application
$[a, b, Z]$	$==$	Abbreviation for: $[a\|[b\|[Z\|[\,]]]]$

Accordingly, a program p is constructed as follows:

$$\begin{array}{lcl} t & ::= & a \mid X \mid \_ \mid f(t_1, \ldots, t_n) \\ g & ::= & p(t_1, \ldots, t_k) \mid X = t \\ c & ::= & p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_r \\ q & ::= & \leftarrow g_1, \ldots, g_r \\ p & ::= & c_1 \ldots c_m q \end{array}$$

- A term t either is an atom, a (possibly anonymous) variable or a constructor application.
- A goal g either is a literal, i.e., a predicate call, or a unification.
- A clause c consists of a head $p(X_1, \ldots, X_k)$ together with body consisting of a sequence of goals.
- A program consists of a sequence of clauses together with a sequence of goals as query.

Procedural View of PuP-Programs:

literal	$==$	procedure call
predicate	$==$	procedure
definition	$==$	body
term	$==$	value
unification	$==$	basic computation step
binding of variables	$==$	side effect

Warning: Predicate calls ...

- do not return results!
- modify the caller solely through side effects :-)
- may fail. Then, the following definition is tried $\Longrightarrow$ backtracking

Inefficiencies:

Backtracking:

- The matching alternative must be searched for $\Longrightarrow$ Indexing
- Since a successful call may still fail later, the stack can only be cleared if there are no pending alternatives.

Unification:

- The translation possibly must switch between build and check several times.
- In case of unification with a variable, an Occur Check must be performed.

Type Checking:

- Since Prolog is untyped, it must be checked at run-time whether or not a term is of the desired form.
- Otherwise, ugly errors could show up.

Some Optimizations:

- Replacing last calls with jumps;
- Compile-time type inference;
- Identification of deterministic predicates ...

Example:

$$\begin{array}{rcl}
\mathsf{app}(X, Y, Z) & \leftarrow & X = [\,],\; Y = Z \\
\mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'],\; Z = [H|Z'],\; \mathsf{app}(X', Y, Z') \\
 & \leftarrow & \mathsf{app}([a, b], [Y, c], Z)
\end{array}$$

Observation:

- In PuP, functions must be simulated through predicates.
- These then have designated input- and output parameters.
- Input parameters are those which are instantiated with a variable-free term whenever the predicate is called.

  These are also called ground.
- In the example, the first parameter of app is an input parameter.
- Unification with such a parameter can be implemented as pattern matching !
- Then we see that app in fact is deterministic !!!

5.1 Groundness Analysis

A variable X is called ground w.r.t. a program execution π starting program entry and entering a program point v, if X is bound to a variable-free term.

Goal:

- Find all variables which are ground whenever a particular program point is reached !
- Find all arguments of a predicate which are ground whenever the predicate is called !

Idea:

- Describe groundness by values from $\mathbb{B}$:

  $1 \quad == \quad$ variable-free term;

  $0 \quad == \quad$ term which contains variables.

- A set of variable assignments is described by Boolean functions :-)

  $X \leftrightarrow Y \quad == \quad$ X is ground iff Y is ground.

  $X \wedge Y \quad == \quad$ X and Y are ground.

Idea (cont.):

- The constant function 0 denotes an unreachable program point.
- Occurring sets of variable assignments are closed under substitution.

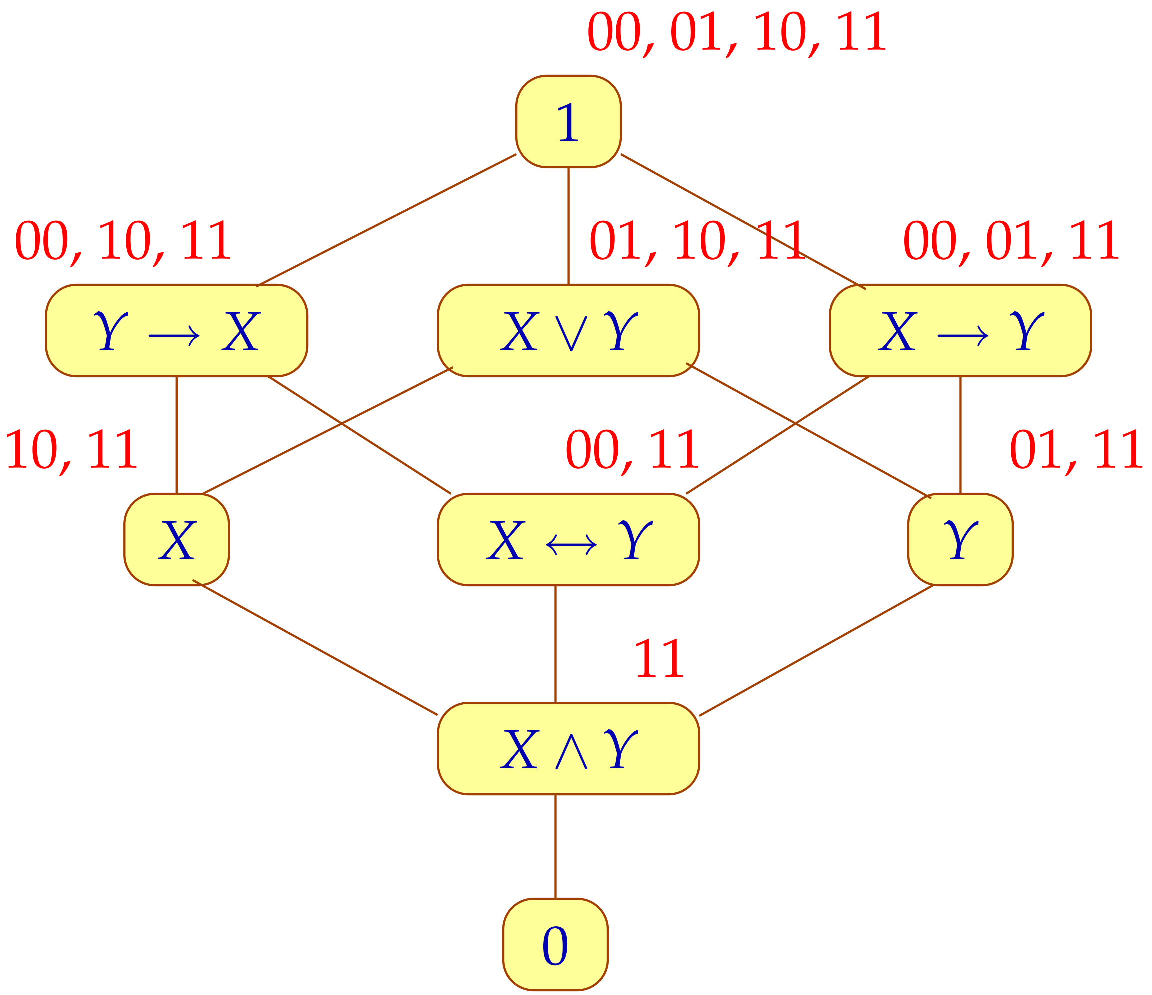
  This means that for every occurring function $\phi \neq 0$,

  $$\phi(1, \ldots, 1) = 1$$

  These functions are called positive.
- The set of all positive functions is called Pos.

  Ordering: $\phi_1 \sqsubseteq \phi_2$ if $\phi_1 \Rightarrow \phi_2$.
- In particular, the least element is 0 :-)

Example:


00, 01, 10, 11
1
00, 10, 11
01, 10, 11
00, 01, 11
$Y \rightarrow X$
$X \vee Y$
$X \rightarrow Y$
10, 11
00, 11
01, 11
X
$X \leftrightarrow Y$
Y
11
$X \wedge Y$
0

Remarks:

- Not all positive functions are monotonic !!!
- For k variables, there are $2^{2^k-1}+1$ many functions.
- The height of the complete lattice is 2^k.
- We construct an interprocedural analysis which for every predicate p determines a (monotonic) transformation

$$[\![p]\!]^\sharp \;:\; \mathsf{Pos} \to \mathsf{Pos}$$

- For every clause, $p(X_1, \ldots, X_k) \Leftarrow g_1, \ldots, g_n$ we obtain the constraint:

$$[\![p]\!]^\sharp\,\psi \quad \sqsupseteq \quad \exists\, X_{k+1}, \ldots, X_m.\; [\![g_n]\!]^\sharp\,(\ldots([\![g_1]\!]^\sharp\,\psi)\ldots)$$

// m number of clause variables

Abstract Unification:

$$\begin{array}{lll} [\![X = t]\!]^\sharp \psi & = & \psi \wedge (X \leftrightarrow X_1 \wedge \ldots \wedge X_r) \\ & & \text{if} \quad \mathit{Vars}(t) = \{X_1, \ldots, X_r\}. \end{array}$$

Abstract Literal:

$$[\![q(s_1, \ldots, s_k)]\!]^\sharp \psi \quad = \quad \mathsf{combine}^\sharp_{s_1,\ldots,s_k}(\psi, [\![q]\!]^\sharp (\mathsf{enter}^\sharp_{s_1,\ldots,s_k} \psi))$$

// analogous to procedure call !!

Thereby:

$$\begin{array}{rcl}
\mathsf{enter}^{\sharp}_{s_1,\ldots,s_k}\,\psi & = & \mathsf{ren}\,(\exists\, X_1,\ldots,X_m.\; [\![\bar{X}_1 = s_1,\ldots,\bar{X}_k = s_k]\!]^{\sharp}\psi) \\[2ex]
\mathsf{combine}^{\sharp}_{s_1,\ldots,s_k}(\psi,\psi_1) & = & \exists\,\bar{X}_1,\ldots,\bar{X}_r.\; \psi \wedge [\![\bar{X}_1 = s_1,\ldots,\bar{X}_k = s_k]\!]^{\sharp}(\overline{\mathsf{ren}}\,\psi_1)
\end{array}$$

where

$$\begin{array}{rcl}
\exists\, X.\,\phi & = & \phi[0/X] \vee \phi[1/X] \\
\mathsf{ren}\,\phi & = & \phi[X_1/\bar{X}_1,\ldots,X_k/\bar{X}_k] \\[2ex]
\overline{\mathsf{ren}}\,\phi & = & \phi[\bar{X}_1/X_1,\ldots,\bar{X}_r/X_r]
\end{array}$$

Example:

$$\begin{array}{lcl} \mathsf{app}(X, Y, Z) & \leftarrow & X = [\,],\; Y = Z \\ \mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'],\; Z = [H|Z'],\; \mathsf{app}(X', Y, Z') \end{array}$$

Then

$$\begin{array}{lcl} [\![\mathsf{app}]\!]^\sharp(X) & \sqsupseteq & X \wedge (Y \leftrightarrow Z) \\ [\![\mathsf{app}]\!]^\sharp(X) & \sqsupseteq & \textbf{let } \psi = X \wedge H \wedge X' \wedge (Z \leftrightarrow Z') \\ & & \textbf{in } \exists\, H, X', Z'.\ \mathsf{combine}^\sharp_{\dots}(\psi, [\![\mathsf{app}]\!]^\sharp(\mathsf{enter}^\sharp_{\dots}(\psi))) \end{array}$$

where for $\psi = X \wedge H \wedge X' \wedge (Z \leftrightarrow Z')$:

$$\begin{array}{lcl} \mathsf{enter}^\sharp_{\dots}(\psi) & = & X \\ \mathsf{combine}^\sharp_{\dots}(\psi, X \wedge (Y \leftrightarrow Z)) & = & (X \wedge H \wedge X' \wedge (Z \leftrightarrow Z') \wedge (Y \leftrightarrow Z') \end{array}$$

Example (Cont.):

Furthermore,

$$\begin{array}{lll} [\![\mathsf{app}]\!]^\sharp(Z) & \sqsupseteq & X \wedge Y \wedge Z \\ [\![\mathsf{app}]\!]^\sharp(Z) & \sqsupseteq & \textbf{let } \psi = X \wedge H \wedge X' \wedge Z \wedge Z' \\ & & \textbf{in } \exists\, H, X', Z'.\ \mathsf{combine}^\sharp_{\ldots}(\psi, [\![\mathsf{app}]\!]^\sharp(\mathsf{enter}^\sharp_{\ldots}(\psi))) \end{array}$$

where for $\psi = Z \wedge H \wedge Z' \wedge (X \leftrightarrow X')$:

$$\begin{array}{lll} \mathsf{enter}^\sharp_{\ldots}(\psi) & = & Z \\ \mathsf{combine}^\sharp_{\ldots}(\psi, X \wedge Y \wedge Z) & = & X \wedge H \wedge X' \wedge Y \wedge Z \wedge Z' \end{array}$$

Fixpoint iteration therefore yields:

$$[\![\mathsf{app}]\!]^\sharp\,(X) \;=\; X \wedge (Y \leftrightarrow Z) \qquad [\![\mathsf{app}]\!]^\sharp\,(Z) \;=\; X \wedge Y \wedge Z$$

Discussion:

- Exhaustive tabulation of the transformation $[\![\mathsf{app}]\!]^\sharp$ is not feasible.
- Therefore, we rely on demand-driven fixpoint iteration !
- The evaluation starts with the evaluation of the query g, i.e., with the evaluation of $[\![g]\!]^\sharp\,1$.
- The set of inspected fixpoint variables $[\![p]\!]^\sharp\,\psi$ yields a description of all possible calls :-))
- For an efficient representation of functions $\psi \in \mathsf{Pos}$ we rely on binary decision diagrams (BDDs).

Background 6: Binary Decision Diagrams

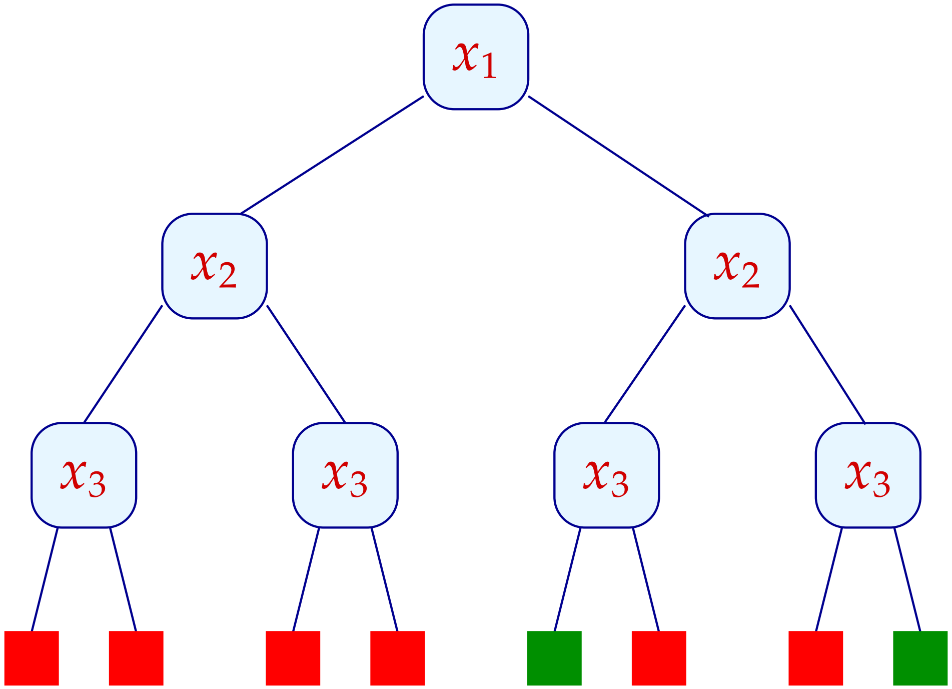
Idea (1):

- Choose an ordering $x_1, \ldots, x_k$ on the arguments ...
- Represent the function $f : \mathbb{B} \to \ldots \to \mathbb{B}$ by $[f]_0$ where:

$$\begin{array}{lll} [b]_k & = & b \\ [f]_{i-1} & = & \textbf{fun } x_i \to \textbf{if } x_i \textbf{ then } [f\ 1]_i \\ & & \qquad\qquad \textbf{else } [f\ 0]_i \end{array}$$

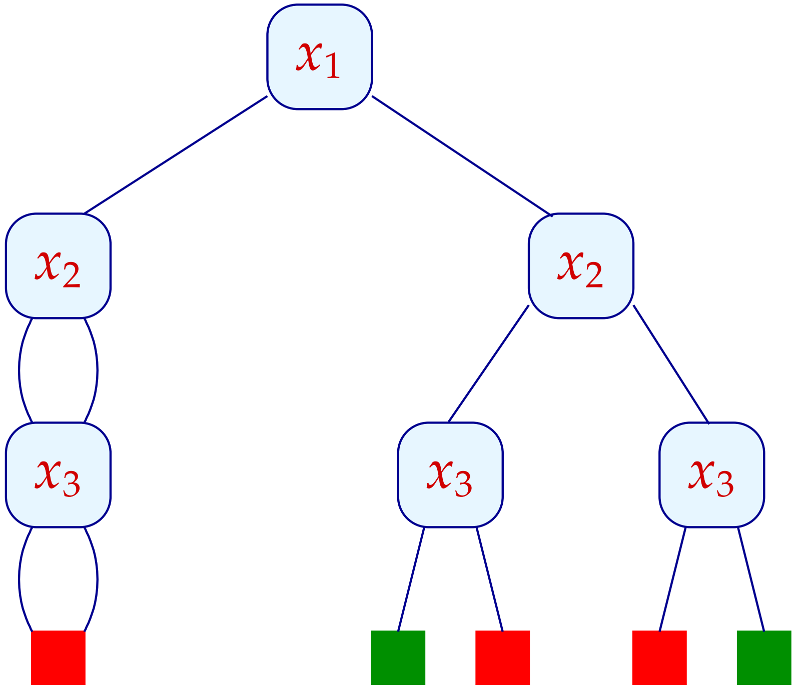
Example: $f\ x_1\ x_2\ x_3 = x_1 \wedge (x_2 \leftrightarrow x_3)$

... yields the tree:



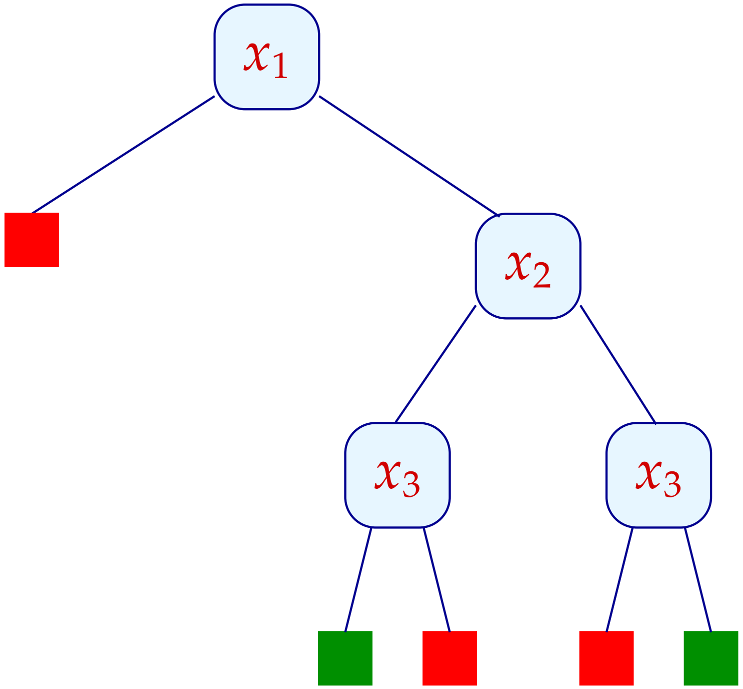
Idea (2):

- Decision trees are exponentially large :-(
- Often, however, many sub-trees are isomorphic :-)
- Isomorphic sub-trees need to be represented only once ...



Idea (3):

- Nodes whose test is irrelevant, can also be abandoned ...



Discussion:

- This representation of the Boolean function f is unique !

  $\Longrightarrow$

  Equality of functions is efficiently decidable !!

- For the representation to be useful, it should support the basic operations: $\wedge, \vee, \neg, \Rightarrow, \exists\, x_j$ ...

$$\begin{array}{lcl} [b_1 \wedge b_2]_k & = & b_1 \wedge b_2 \\ [f \wedge g]_{i-1} & = & \textbf{fun } x_i \;\to\; \textbf{if } x_i \textbf{ then } [f\,1 \wedge g\,1]_i \\ & & \qquad\qquad\qquad \textbf{else } [f\,0 \wedge g\,0]_i \\ & & // \quad \text{analogous for the remaining operators} \end{array}$$

$$
\begin{array}{lcll}
[\exists\, x_j.\, f]_{i-1} & = & \textbf{fun}\ x_i \ \rightarrow\ \textbf{if}\ x_i\ \textbf{then}\ [\exists\, x_j.\, f\ 1]_i & \\
 & & \qquad\qquad \textbf{else}\ [\exists\, x_j.\, f\ 0]_i & \text{if } i < j \\
[\exists\, x_j.\, f]_{j-1} & = & [f\ 0 \vee f\ 1]_j &
\end{array}
$$

- Operations are executed bottom-up.
- Root nodes of already constructed sub-graphs are stored in a unique-table

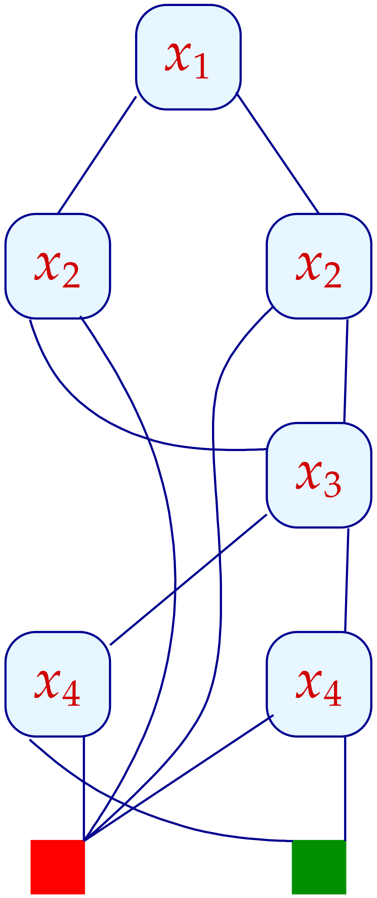
  $\Longrightarrow$

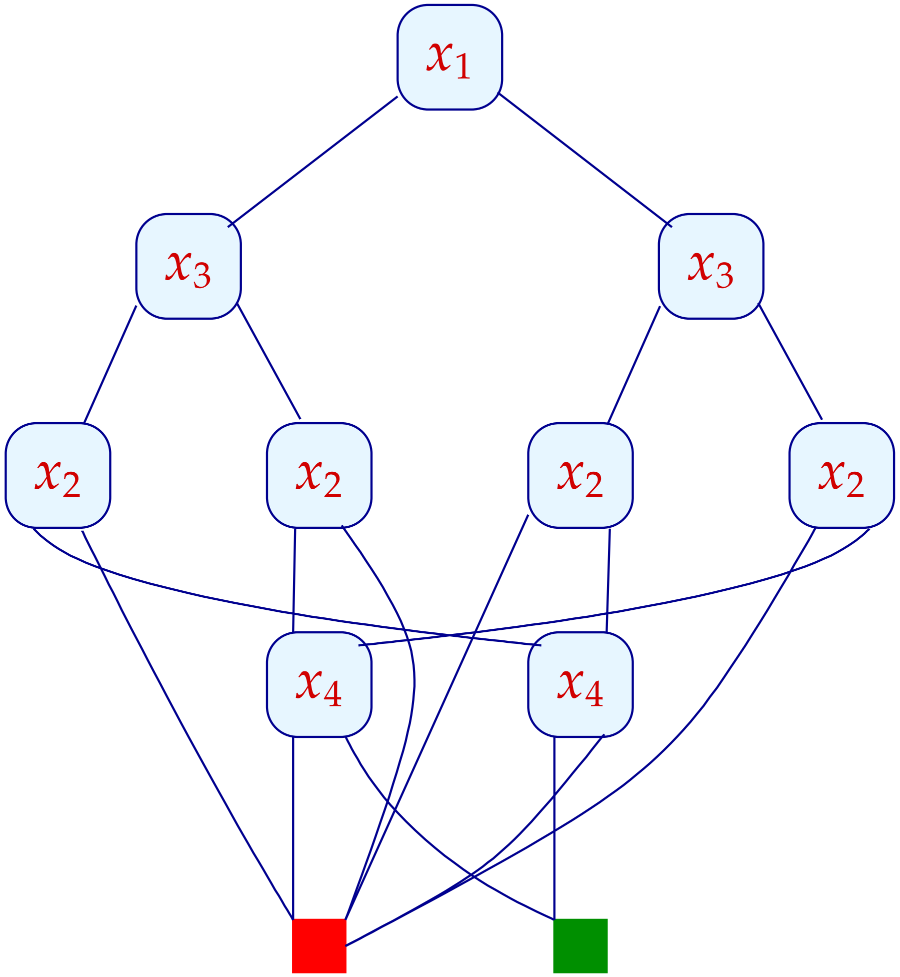
  Isomorphy can be tested in constant time !
- The operations thus are polynomial in the size of the input BDDs :-)

Discussion:

- Originally, BDDs have been developped for circuit verification.
- Today, they are also applied to the verification of software ...
- A system state is encoded by a sequence of bits.
- A BDD then describes the set of all reachable system states.
- Warning: Repeated application of Boolean operations may increase the size dramatically !
- The variable ordering may have a dramatic impact ...

Example: $(x_1 \leftrightarrow x_2) \wedge (x_3 \leftrightarrow x_4)$





Discussion (2):

- In general, consider the function:

  $$(x_1 \leftrightarrow x_2) \wedge \ldots \wedge (x_{2n-1} \leftrightarrow x_{2n})$$

  W.r.t. the variable ordering:

  $$x_1 < x_2 < \ldots < x_{2n}$$

  the BDD has $3n$ internal nodes.

  W.r.t. the variable ordering:

  $$x_1 < x_3 < \ldots < x_{2n-1} < x_2 < x_4 < \ldots < x_{2n}$$

  the BDD has more than 2^n internal nodes !!

- A similar result holds for the implementation of Addition through BDDs.

Discussion (3):

- Not all Boolean functions have small BDDs :-(
- Difficult functions:
  - □ multiplication;
  - □ indirect addressing ...

$\Longrightarrow$ data-intensive programs cannot be analyzed in this way :-(

Perspectives: Further Properties of Programs

Freeness: Is X_i possibly/always unbound ?

$\Longrightarrow$

If X_i is always unbound, no indexing for X_i is required :-)

If X_i is never unbound, indexing for X_i is complete :-)

Pair Sharing: Are X_i, X_j possibly bound to terms t_i, t_j with

$$Vars(t_i) \cap Vars(t_j) \neq \emptyset \quad ?$$

$\Longrightarrow$

Literals without sharing can be executed in parallel :-)

Remark:

Both analyses may profit from Groundness !

5.2 Types for Prolog

Example:

$$\begin{array}{lll}
\mathsf{nat}(X) & \leftarrow & X = 0 \\
\mathsf{nat}(X) & \leftarrow & X = s(Y), \mathsf{nat}(Y) \\
\mathsf{nat\_list}(X) & \leftarrow & X = [] \\
\mathsf{nat\_list}(X) & \leftarrow & X = [H|T], \mathsf{nat}(H), \mathsf{nat\_list}(T)
\end{array}$$

Discussion

- In Prolog, a type is a set of ground terms with a simple description.
- There is no common agreement what simple means :-)
- One possibility are (non-deterministic) finite tree automata or normal Horn clauses:

$$\begin{array}{lcll}
\mathsf{nat\_list}([H|T]) & \leftarrow & \mathsf{nat}(H), \mathsf{nat\_list}(T) & \text{normal} \\
\mathsf{bin}(node(T, T)) & \leftarrow & \mathsf{bin}(T) & \text{nicht normal} \\
\mathsf{tree}(node(T_1, T_2)) & \leftarrow & \mathsf{tree}(T_1), \mathsf{tree}(T_2) & \text{normal}
\end{array}$$

Comparison:

Normal clauses	Tree automaton
unary predicate	state
normal clause	transition
constructor in the head	input symbol
body	pre-condition

General Form:

$$\begin{array}{lll} p(a(X_1, \ldots, X_k)) & \leftarrow & p_1(X_1), \ldots, p_k(X_k) \\ p(X) & \leftarrow & \\ p(b) & \leftarrow & \end{array}$$

Properties:

- Types then are in fact regular tree languages ;-)
- Types are closed under intersection:

$$
\begin{array}{lll}
\langle p,q\rangle(a(X_1,\ldots,X_k)) & \leftarrow & \langle p_1,q_1\rangle(X_1),\ldots,\langle p_k,q_k\rangle(X_k) \qquad \text{if} \\
p(a(X_1,\ldots,X_k)) & \leftarrow & p_1(X_1),\ldots,p_k(X_k) \qquad \text{and} \\
q(a(X_1,\ldots,X_k)) & \leftarrow & q_1(X_1),\ldots,q_k(X_k)
\end{array}
$$

- Types are also closed under union :-)
- Queries $p(X)$ and $p(t)$ can be decided in polynomial time
  but:
- ... only in presence of tabulation !

- Or the program is topdown deterministic ...

Example: Topdown vs. Bottom-up

$$\begin{array}{lll} p(a(X_1, X_2)) & \leftarrow & p_1(X_1), p_2(X_2) \\ p(a(X_1, X_2)) & \leftarrow & p_2(X_1), p_1(X_2) \\ p_1(b) & \leftarrow & \\ p_2(c) & \leftarrow & \end{array}$$

... is bottom-up, but not topdown deterministic.

There is no topdown deterministic program for this type !

$\Longrightarrow$

Topdown deterministic types are closed under intersection, but not under union !!!

For a set T of terms, we define the set $\Pi(T)$ of paths in terms from T:

$$
\begin{array}{lcl}
\Pi(T) & = & \bigcup\{\Pi(t) \mid t \in T\} \\
\\
\Pi(b) & = & \{b\} \\
\Pi(a(t_1,\ldots,t_k)) & = & \{a_j w \mid w \in \Pi(t_j)\} \qquad (k > 0) \\
 & & // \quad \text{for new unary constructors } a_j
\end{array}
$$

Example

$$
\begin{array}{lcl}
T & = & \{a(b,c), a(c,b)\} \\
\Pi(T) & = & \{a_1 b, a_2 c, a_1 c, a_2 b\}
\end{array}
$$

Vice versa from a set P of paths, a set $\Pi^-(P)$ of terms can be recovered:

$$\Pi^-(P) \quad = \quad \{t \mid \Pi(t) \subseteq P\}$$

Example (Cont.):

$$\begin{array}{lll} P & = & \{a_1 b, a_2 c, a_1 c, a_2 b\} \\ \Pi^-(P) & = & \{a(b,b), a(b,c), a(c,b), a(c,c)\} \end{array}$$

The set has become larger !!

Theorem:

Assume that T is a regular set of terms. Then:

- $\Pi(T)$ is regular :-)
- $T \subseteq \Pi^-(\Pi(T))$:-)
- $T = \Pi^-(\Pi(T))$ iff T is topdown deterministic :-)
- $\Pi^-(\Pi(T))$ is the smallest superset of T which is topdown deterministic. :-)

Consequence:

If we are interested in topdown deterministic types, it suffices to determine the set of paths in terms !!!

Example (Cont.):

$$
\begin{array}{lcl}
\mathsf{add}(X, Y, Z) & \leftarrow & X = 0, \mathsf{nat}(Y), Y = Z \\
\mathsf{add}(X, Y, Z) & \leftarrow & \mathsf{nat}(X), X = s(X'), Z = s(Z'), \mathsf{add}(X', Y, Z') \\
\mathsf{mult}(X, Y, Z) & \leftarrow & X = 0, \mathsf{nat}(Y), Z = 0 \\
\mathsf{mult}(X, Y, Z) & \leftarrow & \mathsf{nat}(X), X = s(X'), \mathsf{mult}(X', Y, Z'), \mathsf{add}(Z', Y, Z)
\end{array}
$$

Question:

Which run-time checks are necessary?

Idea:

- Approximate the semantics of predicates by means of topdown-deterministic regular tree languages !
- Alternatively: Approximate the set of paths in the semantics of predicates by regular word languages !

Idea:

- All predicates $p/k, k > 0$, are split into predicates $p_1/1, \ldots, p_k/1$.

Semantics:

Let $\mathcal{C}$ denote a set of clauses.

The set $[\![p]\!]_{\mathcal{C}}$ is the set of tuples of ground terms $(s_1, \ldots, s_k)$, for which $p(s_1, \ldots, s_k)$ is provable :-)

$[\![p]\!]_{\mathcal{C}}$ (p predicate) thus is the smallest collection of sets of tuples for which:

$$\sigma(\underline{t}) \in [\![p]\!]_{\mathcal{C}} \quad \text{when ever} \quad \forall\, i.\ \sigma(\underline{t}_i) \in [\![p_i]\!]_{\mathcal{C}}$$

for clauses $p(\underline{t}) \leftarrow p_1(\underline{t}_1), \ldots, p_n(\underline{t}_n) \in \mathcal{C}$ and ground substitutions σ.

Approximation of Paths:

Every clause

$$p(t_1, \ldots, t_k) \;\leftarrow\; \alpha$$

is approximated by the clauses:

$$\begin{array}{lcll} p_j(w) & \leftarrow & \bigwedge \Pi(\alpha) & \quad \text{where} \\ \Pi(g_1, \ldots, g_m) & = & \Pi(g_1) \cup \ldots \cup \Pi(g_m) & \\ \Pi(q(s_1, \ldots, s_n)) & = & \{q_i(w) \mid w \in \Pi(s_i)\} & \end{array}$$

$(j = 1, \ldots, k, w \in \Pi(t_j))$.

Example:

$$\begin{array}{lcl} \mathsf{add}(0, Y, Y) & \leftarrow & \mathsf{nat}(Y) \\ \mathsf{add}(s(X), Y, s(Z)) & \leftarrow & \mathsf{add}(X, Y, Z) \end{array}$$

yields:

$$\begin{array}{lcl}
\mathsf{add}_1(0) & \leftarrow & \mathsf{nat}_1(Y) \\
\mathsf{add}_2(Y) & \leftarrow & \mathsf{nat}_1(Y) \\
\mathsf{add}_3(Y) & \leftarrow & \mathsf{nat}_1(Y) \\
\\
\mathsf{add}_1(s_1\, X) & \leftarrow & \mathsf{add}_1(X), \mathsf{add}_2(Y), \\
 & & \mathsf{add}_3(Z) \\
\mathsf{add}_2(Y) & \leftarrow & \mathsf{add}_1(X), \mathsf{add}_2(Y), \\
 & & \mathsf{add}_3(Z) \\
\mathsf{add}_3(s_1\, Z) & \leftarrow & \mathsf{add}_1(X), \mathsf{add}_2(Y), \\
 & & \mathsf{add}_3(Z)
\end{array}$$

Discussion:

- Every literal has at most one occurrence of a variable.
- The literals $q_j(w_j Y)$ where the variable Y does not occur in the head, represent tests:

  If there is a w with $w_j w \in [\![q_j]\!]_{C^\sharp}$ for all such j, then we can cancel these literals.

  If there is no such w, then we can cancel the clause ...

... in the Example:

The literals:

$$\mathsf{add}_1(X), \mathsf{add}_2(Y), \mathsf{add}_3(Z)$$

are all satisfiable :-)

We conclude:

$$\begin{array}{lll}
\mathsf{add}_1(0) & \leftarrow & \\
\mathsf{add}_2(Y) & \leftarrow & \mathsf{nat}_1(Y) \\
\mathsf{add}_3(Y) & \leftarrow & \mathsf{nat}_1(Y) \\
\\
\mathsf{add}_1(s_1\,X) & \leftarrow & \mathsf{add}_1(X) \\
\mathsf{add}_2(Y) & \leftarrow & \mathsf{add}_2(Y) \\
\mathsf{add}_3(s_1\,Z) & \leftarrow & \mathsf{add}_3(Z)
\end{array}$$

We conclude:

$$\begin{array}{lcl}
\mathsf{add}_1(0) & \leftarrow & \\
\mathsf{add}_2(Y) & \leftarrow & \mathsf{nat}_1(Y) \\
\mathsf{add}_3(Y) & \leftarrow & \mathsf{nat}_1(Y) \\
\\
\mathsf{add}_1(s_1\ X) & \leftarrow & \mathsf{add}_1(X) \\
\\
\\
\mathsf{add}_3(s_1\ Z) & \leftarrow & \mathsf{add}_3(Z)
\end{array}$$

We verify:

Theorem

Assume that $\mathcal{C}$ is a set of clauses.

Let $\mathcal{C}^\sharp$ denote the corresponding set of clauses for the paths.

Then for all predicates p/k:

$$\Pi(\llbracket p \rrbracket_{\mathcal{C}}) \;\subseteq\; \llbracket p_1 \rrbracket_{\mathcal{C}^\sharp} \cup \ldots \cup \llbracket p_k \rrbracket_{\mathcal{C}^\sharp}$$

Proof:

Induction on the approximations of the respective fixpoints :-)

A set of clauses with unary predicates and unary constructors is called Alternating Pushdown System (APS).

Theorem

- Every APS is equivalent to a simple APS of the form:

$$\begin{array}{lcl} p(a\,X) & \leftarrow & p_1(X), \ldots, p_r(X) \\ p(X) & \leftarrow & \\ p(b) & \leftarrow & \end{array}$$

- Every APS is equivalent to a normal APS of the form:

$$\begin{array}{lcl} p(a\,X) & \leftarrow & p_1(X) \\ p(X) & \leftarrow & \\ p(b) & \leftarrow & \end{array}$$

Step 1: Removal of complicated heads:

For $w = a^{(1)} \dots a^{(m)}$ $(m > 1)$ we replace

$$\begin{array}{lcl} p(w\,X) & \leftarrow & \mathit{rhs} \qquad \text{with:} \\[1em] p(a^{(1)}\,X) & \leftarrow & p_2(X) \\ p_2(a^{(2)}\,X) & \leftarrow & p_3(X) \\ & \dots & \\ p_{m-1}(a^{(m-1)}\,X) & \leftarrow & p_m(X) \\ p_m(a^{(m)}\,X) & \leftarrow & \mathit{rhs} \\ & & // \qquad p_j \text{ all new} \end{array}$$

Step 1 (Cont.): Removal of complicated heads:

For $w = a^{(1)} \dots a^{(m)} b \quad (m > 0)$ we replace

$$\begin{array}{lll}
p(w) & \leftarrow & \mathit{rhs} \qquad \text{with:} \\[1ex]
p(a^{(1)}\, X) & \leftarrow & p_2(X) \\
p_2(a^{(2)}\, X) & \leftarrow & p_3(X) \\
 & \dots & \\
p_{m-1}(a^{(m-1)}\, X) & \leftarrow & p_m(X) \\
p_m(a^{(m)}\, X) & \leftarrow & p_{m+1}(X) \\
p_{m+1}(b) & \leftarrow & \mathit{rhs} \\
 & & // \qquad p_j \text{ all new}
\end{array}$$

Step 2: Splitting

We separate independent parts of pre-conditions into auxiliary predicates:

$$\begin{array}{lll} \mathit{head} & \leftarrow & \mathit{rest},\; p_1(w_1\,X), \ldots, p_m(w_m\,X) \\ & & \qquad (X \text{ does not occur in } \mathit{head}, \mathit{rest}) \end{array}$$

is replaced with:

$$\begin{array}{lll} \mathit{head} & \leftarrow & \mathit{rest}, q() \\ q() & \leftarrow & p_1(w_1\,X), \ldots, p_m(w_m\,X) \end{array}$$

for a new predicate $q/0$.

Step 3: Normalization

We add simpler derived clauses:

$$\begin{array}{lcl}
\mathit{head} & \leftarrow & p(a\,w), \mathit{rest} \\
p(a\,X) & \leftarrow & p_1(X), \ldots, p_r(X) \\
 & & \qquad\text{implies:} \\
\mathit{head} & \leftarrow & p_1(w), \ldots, p_r(w), \mathit{rest}
\end{array}$$

$$\begin{array}{lcl}
p(X) & \leftarrow & p_1(X), \ldots, p_m(X) \\
p_i(a\,X) & \leftarrow & p_{i1}(X), \ldots, p_{ir_i}(X) \\
 & & \qquad\text{implies:} \\
p(a\,X) & \leftarrow & p_{11}(X), \ldots, p_{mr_m}(X)
\end{array}$$

Step 3 (Cont.): Normalization

$$\begin{array}{lcl} head & \leftarrow & p(w), rest \\ p(X) & \leftarrow & \qquad \text{implies:} \\ head & \leftarrow & rest \end{array}$$

$$\begin{array}{lcl} head & \leftarrow & p(b), rest \\ p(b) & \leftarrow & \qquad \text{implies:} \\ head & \leftarrow & rest \end{array}$$

$$\begin{array}{lcl} p(()) & \leftarrow & p_1(X), \ldots, p_m(X) \\ p_i(a\, X) & \leftarrow & p_{i1}(X), \ldots, p_{ir_i}(X) \\ & & \qquad \text{implies:} \\ p(()) & \leftarrow & p_{11}(X), \ldots, p_{mr_m}(X) \end{array}$$

Example:

$$\begin{array}{lcl} \mathsf{add}_1(X) & \leftarrow & \mathsf{add}_0(X) \\ \mathsf{add}_0(0) & \leftarrow & \\ \mathsf{add}_1(X) & \leftarrow & \mathsf{add}_1(X) \\ \mathsf{add}_1(s_1\,X) & \leftarrow & \mathsf{add}_1(X) \end{array}$$

... results in the new clause:

$$\mathsf{add}_1(0) \quad \leftarrow$$

Theorem

Assume that $\mathcal{C}$ is a finite set of clauses for which steps 1 and 2 have been executed and which then has been saturated according to step 3.

Assume that $\mathcal{C}_0 \subseteq \mathcal{C}$ is the subset of normal clauses of $\mathcal{C}$. Then for all occurring predicates p,

$$[\![p]\!]_{\mathcal{C}_0} = [\![p]\!]_{\mathcal{C}}$$

Proof:

Induction on the depth of terms in $[\![p]\!]_{\mathcal{C}}$:-)

... in the Example:

For $\mathsf{add}_1(X)$ we obtain the following clauses:

$$\begin{array}{lll} \mathsf{add}_1(0) & \leftarrow & \\ \mathsf{add}_1(s_1\ X) & \leftarrow & \mathsf{add}_1(X) \end{array}$$

These clauses are already normal :-)

Transforming into Normal Clauses:

Introduce new predicates for conjunctions of predicates.

Assume that $A = \{p_1, \ldots, p_m\}$. Then:

$$[A](b) \leftarrow \qquad \text{whenever} \quad p_i(b) \leftarrow \quad \text{for all } i.$$

$$[A](a\,X) \leftarrow [B](X) \qquad \text{whenever} \quad B = \{p_{ij} \mid i = 1, \ldots, m\} \quad \text{for}$$

$$p_i(a\,X) \leftarrow p_{i1}(X), \ldots, p_{ir_i}(X)$$

Last Step: Transformation into a Type

- First, the automaton is determinized ...

Last Step: Transformation into a Type

- First, the automaton is determinized ...

- Then transitions for the components of constructors a:

$$p(a_j\,X) \leftarrow p^{(j)}(X)$$

  are joined into a transition for a:

$$p(a(X_1, \ldots, X_k)) \leftarrow p^{(1)}(X_1), \ldots, p^{(k)}(X_k)$$

- Finally, the predicates p_j for the components of the predicate p/k are joined to a transition:

$$p(X_1, \ldots, x_k) \leftarrow p_1(X_1), \ldots, p_k(X_k)$$

In the Example we find:

$$\begin{array}{lcll}
\mathsf{add}(X,Y,Z) & \leftarrow & \mathsf{add}_1(X), \mathsf{nat}(Y), q'(Z) & \qquad \text{where} \\
q'(0) & \leftarrow & & \\
q'(s\,X) & \leftarrow & q'(X) & \\
q' & = & \{\mathsf{nat}, \mathsf{add}_2\} &
\end{array}$$

In the Example we find:

$$\begin{array}{lcll} \mathsf{add}(X, Y, Z) & \leftarrow & \mathsf{add}_1(X), \mathsf{nat}(Y), q'(Z) & \quad \text{where} \\ q'(0) & \leftarrow & & \\ q'(s\,X) & \leftarrow & q'(X) & \\ q' & = & \{\mathsf{nat}, \mathsf{add}_2\} & \end{array}$$

The types $\mathsf{add}_1, q', \mathsf{nat}$ are all equivalent :-)

Discussion:

- For type-checking, it suffices to check for every predicate p/k that

$$[\![p_i]\!]_{\mathcal{C}^\sharp} \subseteq \Pi(T_i)$$

- Since the T_i are topdown deterministic, we have a deterministic automaton for $\Pi(T_i)$:-)
- Therefore, we can easily construct a DFA for the complement $\overline{\Pi(T_i)}$!!
- Then we check whether

$$[\![p_i]\!]_{\mathcal{C}^\sharp} \cap \overline{\Pi(T_i)} = \emptyset$$

$\Longrightarrow$ this saves us determinization :-))

Warning:

- The emptiness probelm for APS is DEXPTIME-complete !
- In many cases, though, our method terminates quickly ;-)

Warning:

- The emptiness probelm for APS is DEXPTIME-complete !
- In many cases, though, our method terminates quickly ;-)

- Inferred types can also be used to understand legacy code.
- Then, however, they are only useful if they are not too complicated !
- Our type inference provides very precise information :-)
- In practical applications, further widenings are applied to accelerate the analysis, e.g., by reducing the number of occurring sets.

5.3 Goal-directed Type Inference

Prolog programs explore predicates only insofar as they contribute to answer a query.

Example: append

$$\begin{array}{lcl} \mathsf{app}([\,], Y, Y) & \leftarrow & \\ \mathsf{app}([H|T], Y, [H|Z]) & \leftarrow & \mathsf{app}(T, Y, Z) \\ & \leftarrow & \mathsf{app}([1,2], [3], Z) \end{array}$$

... results in:

The APS-Approximation

$$
\begin{array}{lcl}
\mathsf{app}_1([|]_1(H)) & \leftarrow & \mathsf{app}_1(T), \mathsf{app}_2(Y), \mathsf{app}_3(Z). \\
\mathsf{app}_1([|]_2(T)) & \leftarrow & \mathsf{app}_1(T), \mathsf{app}_2(Y), \mathsf{app}_3(Z). \\
\mathsf{app}_2(Y) & \leftarrow & \mathsf{app}_1(T), \mathsf{app}_2(Y), \mathsf{app}_3(Z). \\
\mathsf{app}_3([|]_1(H)) & \leftarrow & \mathsf{app}_1(T), \mathsf{app}_2(Y), \mathsf{app}_3(Z). \\
\mathsf{app}_3([|]_2(Z)) & \leftarrow & \mathsf{app}_1(T), \mathsf{app}_2(Y), \mathsf{app}_3(Z). \\
\mathsf{app}_1([\,]) & \leftarrow & \\
\mathsf{app}_2(X) & \leftarrow & \\
\mathsf{app}_3(X)) & \leftarrow & \\
 & \leftarrow & \mathsf{app}_1([|]_1(1)), \mathsf{app}_1([|]_2([|]_1(2))), \mathsf{app}_1([|]_2([|]_2([\,]))), \\
 & & \mathsf{app}_2([|]_1(3)), \mathsf{app}_2([|]_2([\,])), \mathsf{app}_3(X)
\end{array}
$$

Ignoring the query, we find via normalization:

$$\begin{array}{lll}
\mathsf{app}_2(X) & \leftarrow & \\
\mathsf{app}_3(X) & \leftarrow & \\
\mathsf{app}_1([]) & \leftarrow & \\
\mathsf{app}_1([|]_2 X) & \leftarrow & q_0(X) \\
\mathsf{app}_1([|]_2 X) & \leftarrow & q_1(X) \\
[\mathsf{app}_1]([|]_2 X) & \leftarrow & q_2(X) \\
\mathsf{app}_1([|]_1 X) & \leftarrow & \\
q_0([]) & \leftarrow & \\
q_1([|]_2 X) & \leftarrow & q_0(X) \\
q_1([|]_2 X) & \leftarrow & q_1(X) \\
q_1([|]_2 X) & \leftarrow & q_2(X) \\
q_2([|]_1 X) & \leftarrow &
\end{array}$$

Discussion

- The second and third argument can be arbitrary.
- The first argument is a list where nothing is known about the elements :-)
- Ignoring the query, this result is the best we can hope for :-(
- Better results can be obtained if additionally call patterns are tracked !

$\Longrightarrow$ Magic Set Transformation

Magic Sets

- For every predicate p/k, we introduce a new predicate called_p/k with the clauses

$$\mathsf{called}_p(\underline{t}) \leftarrow \quad \text{for the query} \quad \leftarrow \mathsf{p}(\underline{t})$$

- $$\begin{aligned} \mathsf{called}_{p_i}(\underline{t_i}) &\leftarrow \mathsf{called}_p(\underline{t}), \mathsf{p}_1(\underline{t_1}), \ldots, \mathsf{p}_{i-1}(\underline{t_{i-1}}) \\ p_i(\underline{t}) &\leftarrow \mathsf{called}_p(\underline{t}), \mathsf{p}_1(\underline{t_1}), \ldots, \mathsf{p}_{i-1}(\underline{t_m}) \end{aligned}$$

  for every clause:

$$\mathsf{p}(\underline{t}) \leftarrow \mathsf{p}_1(\underline{t_1}), \ldots, \mathsf{p}_m(\underline{t_m})$$

Example: append (Cont.)

$$\begin{array}{lcl}
\mathsf{app}([\,], Y, Y) & \leftarrow & \mathsf{called}([\,], Y, Y) \\
\mathsf{app}([H|T], Y, [H|Z]) & \leftarrow & \mathsf{called}([H|T], Y, [H|Z]), \\
 & & \mathsf{app}(T, Y, Z) \\
\mathsf{called}(T, Y, Z) & \leftarrow & \mathsf{called}([H|T], Y, [H|Z]) \\
\mathsf{called}([1, 2], [3], Z) & \leftarrow &
\end{array}$$

The APS-Approximation:

$$
\begin{array}{lcl}
\mathsf{app}_1([\,]) & \leftarrow & \mathsf{called}_1([\,]), \mathsf{called}_2(X), \mathsf{called}_3(X) \\
\mathsf{app}_2(X) & \leftarrow & \mathsf{called}_1([\,]), \mathsf{called}_2(X), \mathsf{called}_3(X) \\
\mathsf{app}_3(X) & \leftarrow & \mathsf{called}_1([\,]), \mathsf{called}_2(X), \mathsf{called}_3(X) \\
\mathsf{app}_1([|]_1 H) & \leftarrow & \mathsf{called}_1([|]_1 H), \mathsf{called}_1([|]_2 T), \mathsf{called}_2(Y), \mathsf{called}_3([|]_1 H), \mathsf{called}_3([|]_2 Z), \\
 & & \mathsf{app}_1(T), \mathsf{app}_2(Y), \mathsf{app}_3(Z) \\
\mathsf{app}_1([|]_2 T) & \leftarrow & \mathsf{called}_1([|]_1 H), \mathsf{called}_1([|]_2 T), \mathsf{called}_2(Y), \mathsf{called}_3([|]_1 H), \mathsf{called}_3([|]_2 Z), \\
 & & \mathsf{app}_1(T), \mathsf{app}_2(Y), \mathsf{app}_3(Z) \\
\mathsf{app}_2(Y) & \leftarrow & \mathsf{called}_1([|]_1 H), \mathsf{called}_1([|]_2 T), \mathsf{called}_2(Y), \mathsf{called}_3([|]_1 H), \mathsf{called}_3([|]_2 Z), \\
 & & \mathsf{app}_1(T), \mathsf{app}_2(Y), \mathsf{app}_3(Z) \\
\mathsf{app}_3([|]_1 H) & \leftarrow & \mathsf{called}_1([|]_1 H), \mathsf{called}_1([|]_2 T), \mathsf{called}_2(Y), \mathsf{called}_3([|]_1 H), \mathsf{called}_3([|]_2 Z), \\
 & & \mathsf{app}_1(T), \mathsf{app}_2(Y), \mathsf{app}_3(Z) \\
\mathsf{app}_3([|]_2 Z) & \leftarrow & \mathsf{called}_1([|]_1 H), \mathsf{called}_1([|]_2 T), \mathsf{called}_2(Y), \mathsf{called}_3([|]_1 H), \mathsf{called}_3([|]_2 Z), \\
 & & \mathsf{app}_1(T), \mathsf{app}_2(Y), \mathsf{app}_3(Z)
\end{array}
$$

$$
\begin{array}{lcl}
 & \ldots & \\
\mathsf{called}_1(T) & \leftarrow & \mathsf{called}_1([|]_1 H), \mathsf{called}_1([|]_2 T), \mathsf{called}_2(Y), \mathsf{called}_3([|]_1 H), \mathsf{called}_3([|]_2 Z) \\
\mathsf{called}_2(Y) & \leftarrow & \mathsf{called}_1([|]_1 H), \mathsf{called}_1([|]_2 T), \mathsf{called}_2(Y), \mathsf{called}_3([|]_1 H), \mathsf{called}_3([|]_2 Z) \\
\mathsf{called}_3(Z) & \leftarrow & \mathsf{called}_1([|]_1 H), \mathsf{called}_1([|]_2 T), \mathsf{called}_2(Y), \mathsf{called}_3([|]_1 H), \mathsf{called}_3([|]_2 Z) \\
\mathsf{called}_1([|]_1 1) & \leftarrow & \\
\mathsf{called}_1([|]_2([|]_1 2) & \leftarrow & \\
\mathsf{called}_1([|]_2 [|]_2 []) & \leftarrow & \\
\mathsf{called}_2([|]_1 3) & \leftarrow & \\
\mathsf{called}_2([|]_2 []) & \leftarrow & \\
\mathsf{called}_3(X) & \leftarrow &
\end{array}
$$

The Normalized APS-Approximation (Cont.)

$$\begin{array}{lcl}
\mathsf{app}_1([||]_1X) & \leftarrow & q_1(X) \\
\mathsf{app}_1([||]_1X) & \leftarrow & q_2(X) \\
\mathsf{app}_1([]) & \leftarrow & \\
\mathsf{app}_1([||]_2X) & \leftarrow & q_4(X) \\
\mathsf{app}_1([||]_2X) & \leftarrow & q_0(X) \\
\mathsf{app}_1([||]_2X) & \leftarrow & q_5(X) \\
\mathsf{app}_2([||]_1X) & \leftarrow & q_3(X) \\
\mathsf{app}_2([||]_2X) & \leftarrow & q_0(X) \\
\mathsf{app}_3([||]_1X) & \leftarrow & q_1(X) \\
\mathsf{app}_3([||]_1X) & \leftarrow & q_2(X)
\end{array}$$

$$\begin{array}{lcl}
\mathsf{app}_3([||]_1X) & \leftarrow & q_3(X) \\
\mathsf{app}_3([||]_2X) & \leftarrow & q_0(X) \\
\mathsf{app}_3([||]_2X) & \leftarrow & q_4(X) \\
\mathsf{app}_3([||]_2X) & \leftarrow & q_6(X) \\
\mathsf{app}_3([||]_2X) & \leftarrow & q_7(X) \\
\mathsf{app}_3([||]_2X) & \leftarrow & q_8(X) \\
q_0([]) & \leftarrow & \\
q_1(1) & \leftarrow & \\
q_2(2) & \leftarrow & \\
q_3(3) & \leftarrow &
\end{array}$$

$$\begin{array}{lcl}
q_4([||]_2X) & \leftarrow & q_0(X) \\
q_5([||]_1X) & \leftarrow & q_2(X) \\
q_6([||]_1X) & \leftarrow & q_3(X) \\
q_7([||]_1X) & \leftarrow & q_1(X) \\
q_7([||]_1X) & \leftarrow & q_2(X) \\
q_8([||]_2X) & \leftarrow & q_4(X) \\
q_8([||]_2X) & \leftarrow & q_7(X) \\
q_8([||]_2X) & \leftarrow & q_8(X) \\
q_8([||]_2X) & \leftarrow & q_6(X)
\end{array}$$

Discussion

- The result now is amazingly precise !!
- The correct values for the second parameter is inferred.
- For the result parameter, a list containing 1,2 and 3 is inferred.

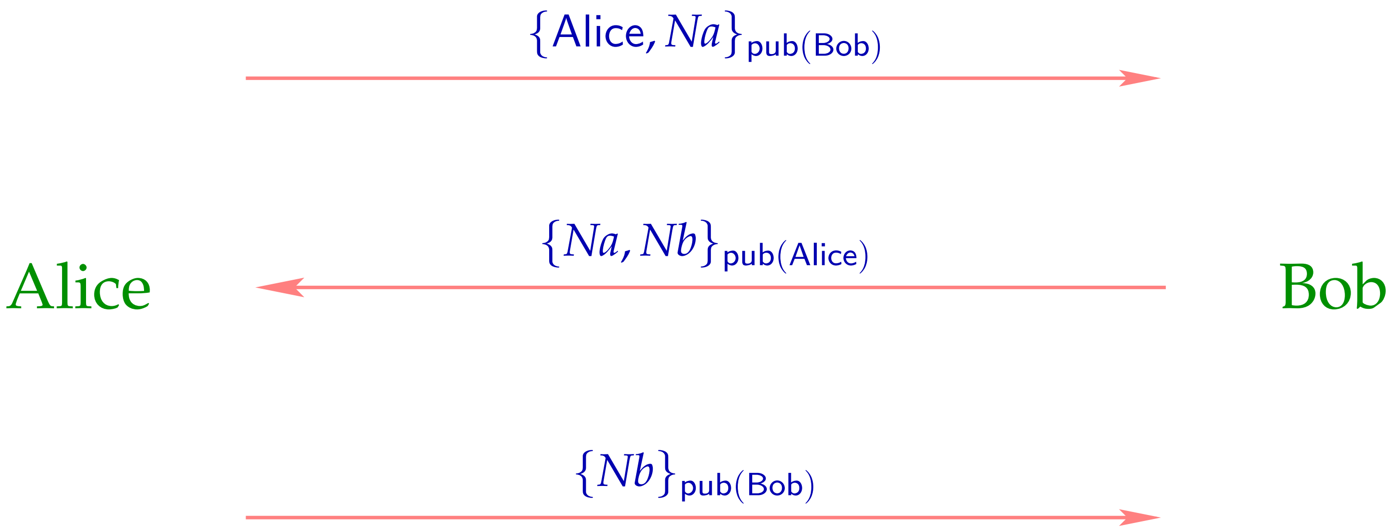
- It only fails to infer that this list is finite and of length 3 :-)

Perspective: Normal Horn Clauses

- Prolog may no longer be the sexiest programming language :-)
- Horn clauses, though, are very well suited for the specification of analysis problems.
- It is a separate problem then to solve the stated analysis problem :-)
- If the least solution cannot be computed exactly, approximate solutions may at least yield approximative answers ...

Example: Cryptographic Protocols

Rules for the Exchange of Messages:



$\{\mathsf{Alice}, Na\}_{\mathsf{pub(Bob)}}$

Alice

$\{Na, Nb\}_{\mathsf{pub(Alice)}}$

Bob

$\{Nb\}_{\mathsf{pub(Bob)}}$

Properties to be verified:

secrecy, authenticity, ...

The Dolev-Yao Model:

- Messages are terms:

	Representation
$\{m\}_k$	$\texttt{encrypt}(m, k)$
$\langle m_1, m_2 \rangle$	$\texttt{pair}(m_1, m_2)$

⟹ Distinct terms represent distinct messages :-)

⟹ perfect cryptography. Therefore, we have:

$\{m\}_k = \{m'\}_{k'}$ iff $m = m'$ and $k = k'$

- The attacker has full control over the network:

  All messages are exchanged with the attacker.

Example: The Needham-Schroeder Protocol

1. $A \longrightarrow B : \{a, n_a\}_{k_b}$
2. $B \longrightarrow A : \{n_a, n_b\}_{k_a}$
3. $A \longrightarrow B : \{n_b\}_{k_b}$

Abstraction:

- Unbounded number of sessions !!
- Nonces sind not necessarily fresh ??

Idea:

Characterize the knowledge of the attacker by means of Horn clauses ...

$$\begin{array}{lll}
1. & A \longrightarrow B : \{a, n_a\}_{k_b} & \mathsf{known}(\{a, n_a\}_{k_b}) \leftarrow \\
2. & B \longrightarrow A : \{n_a, n_b\}_{k_a} & \mathsf{known}(\{X, n_b\}_{k_a}) \leftarrow \mathsf{known}(\{a, X\}_{k_b}) \\
3. & A \longrightarrow B : \{n_b\}_{k_b} & \mathsf{known}(\{X\}_{k_b}) \leftarrow \mathsf{known}(\{n_a, X\}_{k_a})
\end{array}$$

Secrecy of N_b : $\leftarrow \mathsf{known}(n_b)$.

Discussion:

- We have abstracted all nonces with finitely many.
- Less restrictive (though still correct) abstractions are still possible ...

$$
\begin{array}{lll}
1. & A \longrightarrow B : \{a, n_a\}_{k_b} & \ldots \\
2. & B \longrightarrow A : \{n_a, n_b\}_{k_a} & \mathsf{known}(\{X, n_b(X)\}_{k_a}) \leftarrow \mathsf{known}(\{a, X\}_{k_b}) \\
3. & A \longrightarrow B : \{n_b\}_{k_b} & \ldots
\end{array}
$$

The fresh nonce is a function of the received nonce :-)

Blanchet 2001

Further capabilities of the attacker:

$$
\begin{array}{lcl}
\mathsf{known}(\{X\}_Y) & \leftarrow & \mathsf{known}(X), \mathsf{known}(Y) \\
 & & // \quad \text{The attacker can encode} \\
\mathsf{known}(\langle X, Y\rangle) & \leftarrow & \mathsf{known}(X), \mathsf{known}(Y) \\
 & & // \quad \text{The attacker can construct pairs} \\
\mathsf{known}(X) & \leftarrow & \mathsf{known}(\{X\}_Y), \mathsf{known}(Y) \\
 & & // \quad \text{The attacker can decode} \\
\mathsf{known}(X) & \leftarrow & \mathsf{known}(\langle X, Y\rangle) \\
\mathsf{known}(Y) & \leftarrow & \mathsf{known}(\langle X, Y\rangle) \\
 & & // \quad \text{The attacker can project}
\end{array}
$$

Discussion

- Type inference for Prolog computed a regular abstraction of the set of paths of the denotational semantics.
- Sometimes, this is too imprecise :-(
- Instead, we now approximate the denotational semantics directly :-)

- This, however, can be quite expensive

  $\Longrightarrow$ not well suited for compilers :-(

  $\Longrightarrow$ in general, much more precise :-)

Simplification:

We only consider clauses whose heads are of the form:

$$p(f(X_1, \ldots, X_k)) \qquad \text{or} \qquad p(b) \qquad \text{or} \qquad p(X_1, \ldots, X_k)$$

Such clauses are called H1.

Theorem

- Every finite set of H1-clauses is equivalent to a finite set of simple H1-clauses of the form:

$$\begin{array}{lll} p(f(X_1, \ldots, X_k)) & \leftarrow & p_1(X_{i_1}), \ldots, p_r(X_{i_1}) \\ p(X_1, \ldots, X_k) & \leftarrow & p_1(X_{i_1}), \ldots, p_r(X_{i_1}) \\ p(b) & \leftarrow & \end{array}$$

- ... or even to a finite set of normal H1-clauses.

Idea:

We successively introduce simper clauses until the complicated ones become superfluous ...

Rule 1: Splitting

We separate independent parts from the pre-conditions:

$$head \leftarrow rest,\ p_1(X), \ldots, p_m(X)$$

$$(X \text{ does not occur in } head, rest\ )$$

is replaced with:

$$\begin{array}{lcl} head & \leftarrow & rest, q() \\ q() & \leftarrow & p_1(X), \ldots, p_m(X) \end{array}$$

for a new predicate $q/0$.

Rule 2: Simplification

We introduce simpler derived clauses:

$$\begin{array}{lcl} \mathit{head} & \leftarrow & p(f(t_1,\ldots,t_k)), \mathit{rest} \\ p(f(X_1,\ldots,X_k)) & \leftarrow & p_1(X_{i_1}),\ldots,p_r(X_{i_r}) \end{array}$$

implies:

$$\begin{array}{lcl} \mathit{head} & \leftarrow & p_1(t_{i_1}),\ldots,p_r(t_{i_r}), \mathit{rest} \end{array}$$

$$\begin{array}{lcl} \mathit{head} & \leftarrow & p(t_1,\ldots,t_k), \mathit{rest} \\ p(X_1,\ldots,X_k) & \leftarrow & p_1(X_{i_1}),\ldots,p_r(X_{i_r}) \end{array}$$

implies:

$$\begin{array}{lcl} \mathit{head} & \leftarrow & p_1(t_{i_1}),\ldots,p_r(t_{i_r}), \mathit{rest} \end{array}$$

Rule 3 (Cont.): Simplification

$$
\begin{array}{lll}
p(X) & \leftarrow & p_1(X), \ldots, p_m(X) \\
p_i(f(X_1, \ldots, X_k)) & \leftarrow & p_{i1}(X_{i1}), \ldots, p_{ir_i}(X_{ir_i})
\end{array}
$$

implies:

$$
p(f(X_1, \ldots, X_k))) \leftarrow p_{11}(X_{11}), \ldots, p_{mr_m}(X_{mr_m})
$$

$$
\begin{array}{lll}
\mathit{head} & \leftarrow & p(b), \mathit{rest} \\
p(b) & \leftarrow &
\end{array}
$$

implies:

$$
\mathit{head} \leftarrow \mathit{rest}
$$

Rule 4: Guard Simplification

$$\begin{array}{lcl}
p() & \leftarrow & p_1(X), \ldots, p_m(X) \\
p_i(f(X_1, \ldots, X_k)) & \leftarrow & p_{i1}(X_{i1}), \ldots, p_{ir_i}(X_{ir_i}) \\
 & & \qquad \text{implies:} \\
p() & \leftarrow & p_{11}(X_{11}), \ldots, p_{mr_m}(X_{mr_m})
\end{array}$$

$$\begin{array}{lcl}
p() & \leftarrow & p_1(X), \ldots, p_m(X) \\
p_i(b) & \leftarrow & \qquad \text{implies:} \\
p() & \leftarrow &
\end{array}$$

Theorem

Assume that $\mathcal{C}$ is finite set of clauses which is closed under splitting and simplification and guard simplification.

Let $\mathcal{C}_0 \subseteq \mathcal{C}$ denote the subset of simple clauses of $\mathcal{C}$. Then for all occurring predicates p,

$$[\![p]\!]_{\mathcal{C}_0} = [\![p]\!]_{\mathcal{C}}$$

Proof:

Induction on the depth of terms in tuples of $[\![p]\!]_{\mathcal{C}}$:-)

Transformation into normal clauses:

Introduce fresh predicates for conjunctions of unary predicates.

Assume $A = \{p_1, \ldots, p_m\}$. Then:

$$\begin{array}{lcl}
[A](b) & \leftarrow & \qquad \text{whenever} \quad p_i(b) \leftarrow \quad \text{for all } i. \\
[A](f(X_1, \ldots, X_k)) & \leftarrow & [B_1](X_1), \ldots, [B_k](X_k) \\
 & & \qquad \text{whenever} \quad B_i = \{p_{jl} \mid X_{i_{jl}} = X_i\} \quad \text{for} \\
 & & p_j(f(X_1, \ldots, X_k)) \leftarrow p_{j1}(X_{i_{j1}}), \ldots, p_{jr_j}(X_{i_{jr_j}})
\end{array}$$

Warning:

- The emptiness problem for Horn clauses in H1 is DEXPTIME-complete !
- In many cases, our method still terminates quickly ;-)

- Not all Horn clauses are in H1 :-(

$\Longrightarrow$ an approximation technique is required ...

Approximation of Horn Clauses

Step 1:

Simplification of pre-conditions by splitting, simplification and guard simplification (as before :-)

Step 2:

Introduction of copies of variables X. Every copy receives all literals of X as pre-condition.

$$
\begin{array}{lll}
p(f(X,X)) & \leftarrow & q(X) \qquad \text{yields :} \\[2ex]
p(f(X,X')) & \leftarrow & q(X), q(X')
\end{array}
$$

Step 3:

Introduction of an auxiliary predicate for every non-variable subterm of the head.

$$p(f(g(X,Y),Z)) \quad \leftarrow \quad q_1(X), q_2(Y), q_3(Z) \qquad \text{yields :}$$

$$\begin{array}{lcl} p_1(g(X,Y)) & \leftarrow & q_1(X), q_2(Y), q_3(Z) \\ p(f(H,Z)) & \leftarrow & p_1(H), q_1(X), q_2(Y), q_3(Z) \end{array}$$