Static Program Analysis

Seidl/Wilhelm/Hack: Compiler Design – Analysis and Transformation, Springer Verlag, 2012

Reinhard Wilhelm

Universität des Saarlandes

A Short History of Static Program Analysis

- Early high-level programming languages were implemented on very small and very slow machines.
- Compilers needed to generate executables that were extremely efficient in space and time.
- Compiler writers invented efficiency-increasing program transformations, wrongly called optimizing transformations.
- Transformations must not change the semantics of programs.
- Enabling conditions guaranteed semantics preservation.
- Enabling conditions were checked by static analysis of programs.

Theoretical Foundations of Static Program Analysis

- Theoretical foundations for the solution of recursive equations: Kleene (1930s), Tarski (1955)
- Gary Kildall (1972) clarified the lattice-theoretic foundation of data-flow analysis.
- Patrick Cousot (1974) established the relation to the programming-language semantics.

Static Program Analysis as a Verification Method

- Automatic method to derive invariants about program behavior, answers questions about program behavior:
  - will index always be within bounds at program point p?
  - will memory access at p always hit the cache?
- answers of sound static analysis are correct, but approximate: don't know is a valid answer!
- analyses proved correct wrt. language semantics,

Proposed Lectures Content:

1. Introductory example: rules-of-sign analysis
2. theoretical foundations: lattices
3. an operational semantics of the language
4. another example: constant propagation
5. relating the semantics to the analysis—correctness proofs
6. some further static analyses in compilers: Elimination of superfluous computations
   - $\rightarrow$ available expressions
   - $\rightarrow$ live variables
   - $\rightarrow$ array-bounds checks

1 Introduction

... in this course and in the Seidl/Wilhelm/Hack book:

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

Intermediate language into which (almost) everything can be compiled.

However, no procedures. So, only intra-procedural analyses!

2 Example: Rules-of-Sign Analysis

Starting Point: Questions about a program, mostly at a particular program point:

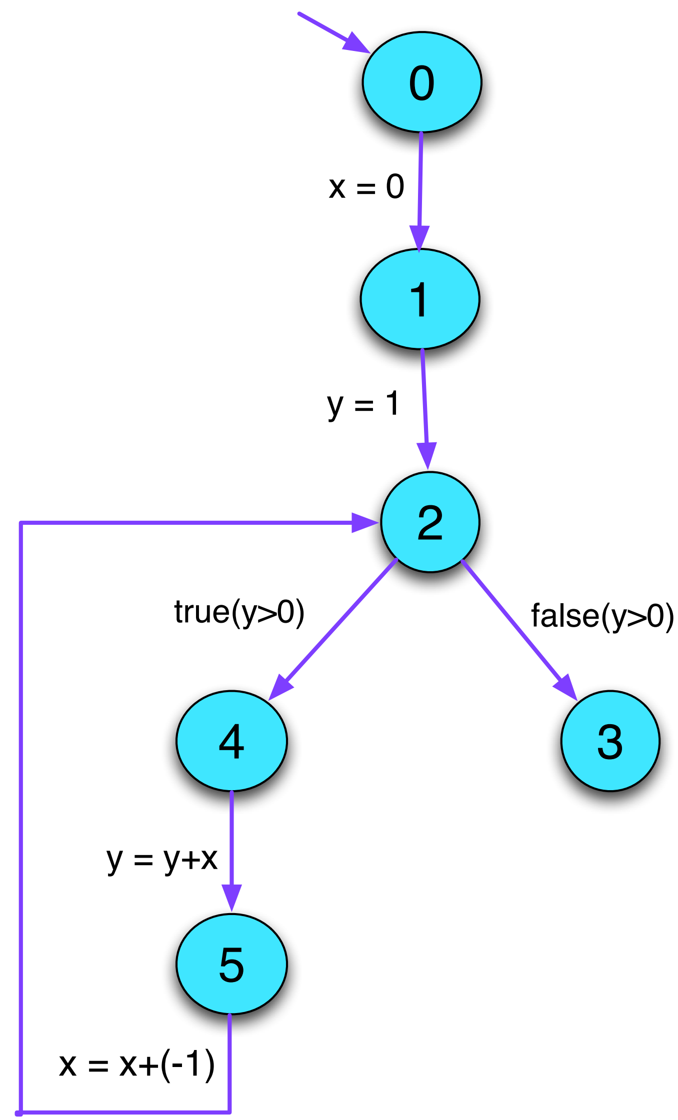
- May variable x have value 0 when program execution reaches this program point? $\longrightarrow$ Attempt to exclude division by 0.
- May x have a negative value? $\longrightarrow$ Attempt to exclude sqrt of a negative number.

Solution: Determine at each program point the sign of the values of all variables of numeric type.

Determines a sound, but maybe approximate answer.

Example program represented as *control-flow graph*

```
x = 0;
y = 1;
while (y > 0) do
      y = y + x;
      x = x + (-1);
```



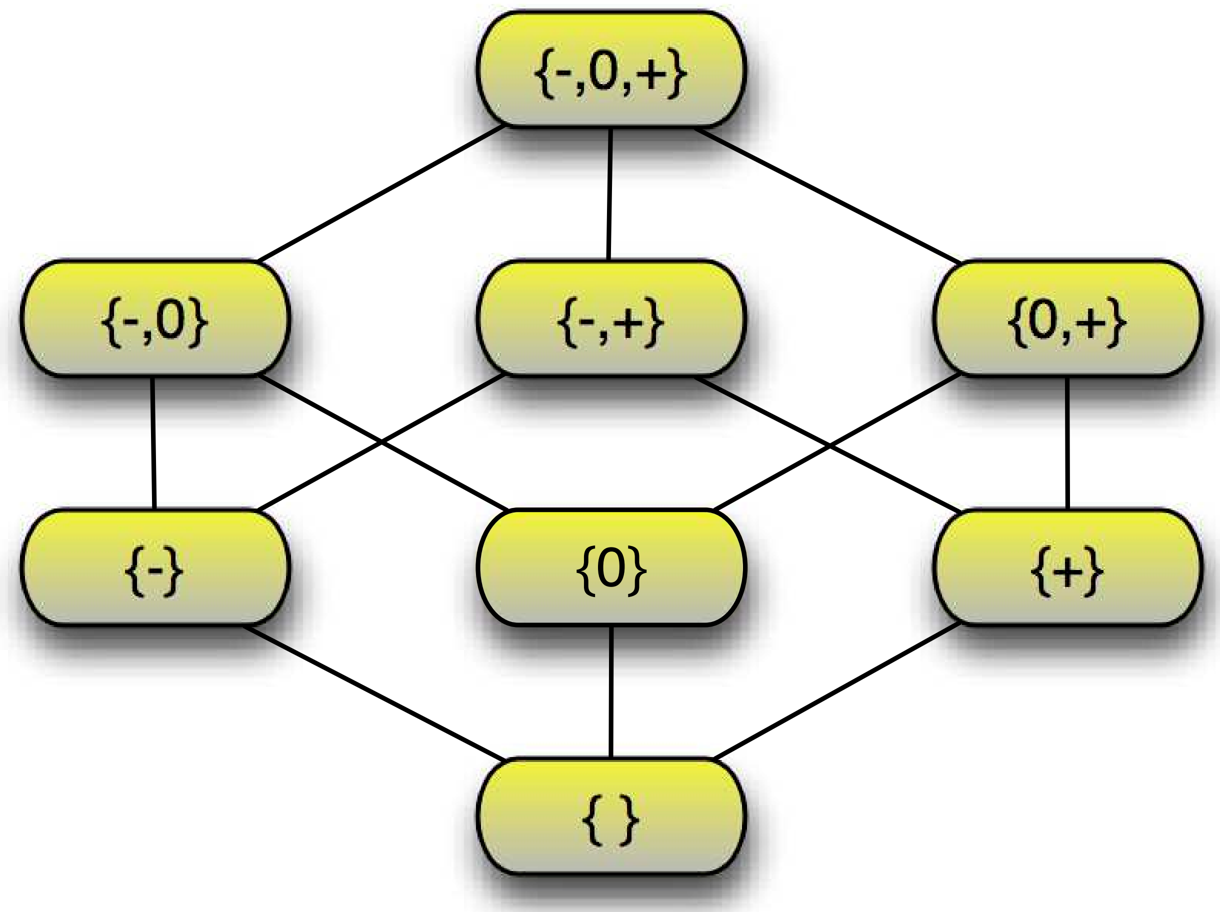
What are the ingredients that we need?

More ingredients?

All the ingredients:

- a set of information elements, each a set of possible signs,
- a partial order, “$\sqsubseteq$”, on these elements, specifying the ”relative strength” of two information elements,
- these together form the abstract domain, a lattice,
- functions describing how signs of variables change by the execution of a statement, abstract edge effects,
- these need an abstract arithmetic, an arithmetic on signs.

We construct the abstract domain for single variables starting with the lattice $Signs = 2^{\{-,0,+\}}$ with the relation "$\sqsubseteq$" ="$\supseteq$".



The analysis should ”bind” program variables to elements in $Signs$.

So, the abstract domain is $\mathbb{D} = (\mathit{Vars} \to \mathit{Signs})_\perp$, a Sign-environment.

$\perp \in \mathbb{D}$ is the function mapping all arguments to $\{\}$.

The partial order on $\mathbb{D}$ is $\quad D_1 \sqsubseteq D_2 \quad$ iff

$$D_1 = \perp \qquad \text{or}$$

$$D_1\, x \supseteq D_2\, x \qquad (x \in \mathit{Vars})$$

Intuition?

The analysis should ”bind” program variables to elements in $Signs$.

So, the abstract domain is $\mathbb{D} = (\,Vars \to Signs)_\perp$. a Sign-environment.

$\perp \in \mathbb{D}$ is the function mapping all arguments to $\{\}$.

The partial order on $\mathbb{D}$ is $\quad D_1 \sqsubseteq D_2 \quad$ iff

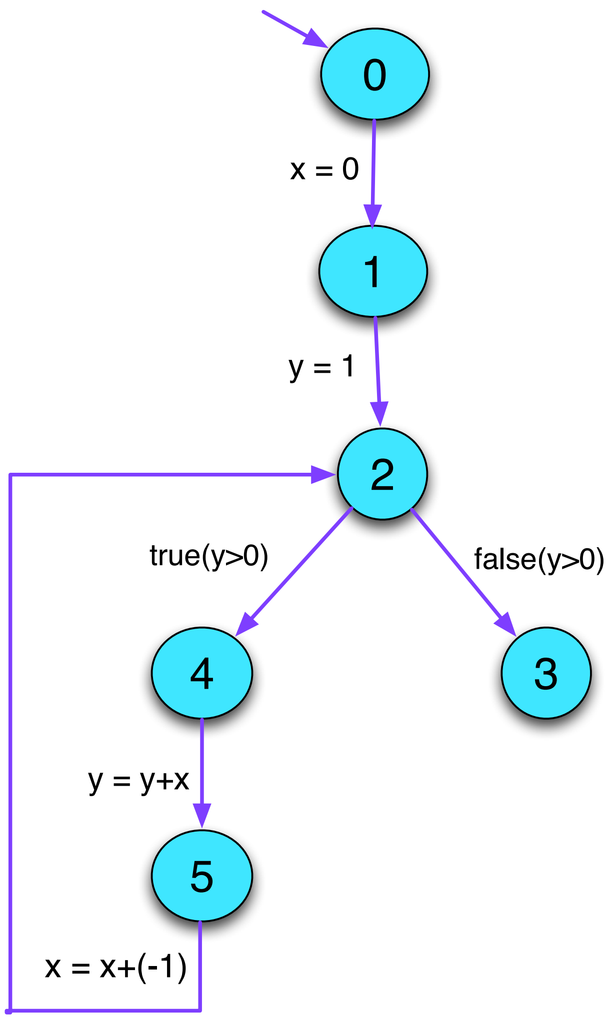
$D_1 = \perp \quad$ or

$D_1\, x \supseteq D_2\, x \quad (x \in Vars)$

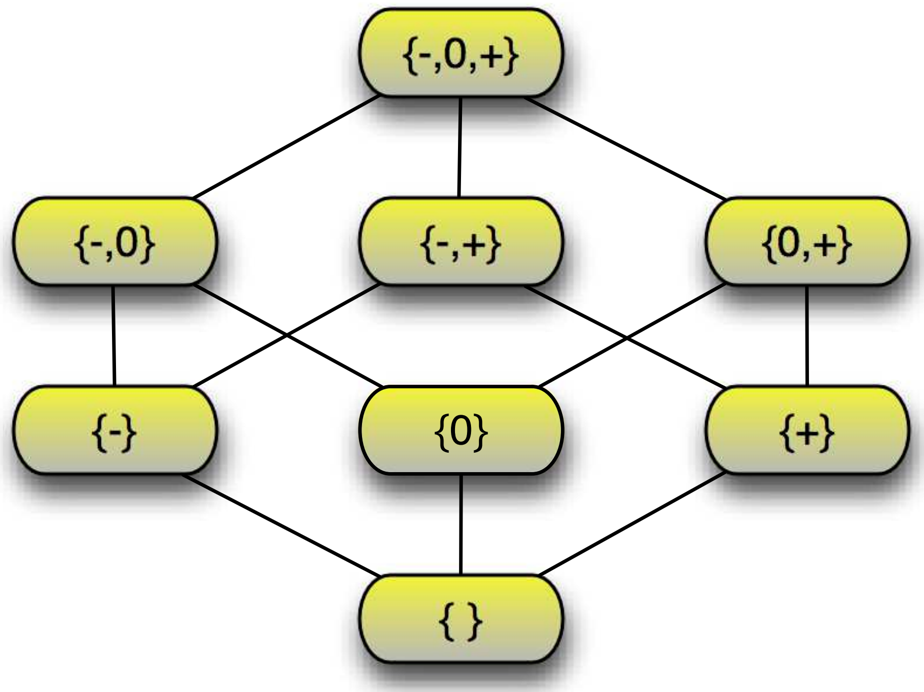
Intuition?

D_1 is at least as precise as D_2 since D_2 admits at least as many signs as D_1

How did we analyze the program?

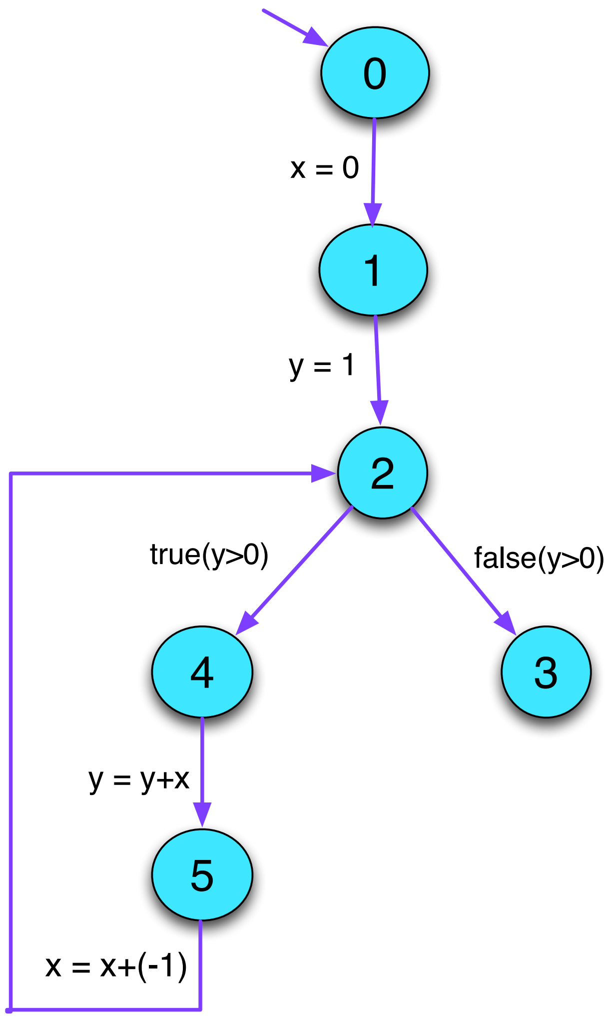


In particular, how did we walk the lattice for y at program point 5?



How is a solution found?

Iterating until a fixed-point is reached



0		1		2		3		4		5	
x	y	x	y	x	y	x	y	x	y	x	y

Idea:

- We want to determine the sign of the values of expressions.

Idea:

- We want to determine the sign of the values of expressions.
- For some sub-expressions, the analysis may yield $\{+, -, 0\}$, which means, it couldn't find out.

Idea:

- We want to determine the signs of the values of expressions.
- For some sub-expressions, the analysis may yield $\{+, -, 0\}$, which means, it couldn't find out.
- We replace the concrete operators $\Box$ working on values by abstract operators $\Box^\sharp$ working on signs:

Idea:

- We want to determine the signs of the values of expressions.
- For some sub-expressions, the analysis may yield $\{+, -, 0\}$, which means, it couldn't find out.
- We replace the concrete operators $\Box$ working on values by abstract operators $\Box^\sharp$ working on signs:
- The abstract operators allow to define an abstract evaluation of expressions:

$$[\![e]\!]^\sharp \; : \; (Vars \to Signs) \to Signs$$

Determining the sign of expressions in a Sign-environment is defined by the function $[\![\ ]\!] : Exp \times SignEnv \rightarrow Val$

$$
\begin{array}{lcl}
[\![c]\!]^{\sharp}\, D & = & \begin{cases} \{+\} & \text{if c > 0} \\ \{-\} & \text{if c < 0} \\ \{0\} & \text{if c = 0} \end{cases} \\
[\![v]\!]^{\sharp} & = & D(v) \\
[\![e_1 \,\Box\, e_2]\!]^{\sharp}\, D & = & [\![e_1]\!]^{\sharp}\, D \,\Box^{\sharp}\, [\![e_2]\!]^{\sharp}\, D \\
[\![\Box e]\!]^{\sharp}\, D & = & \Box^{\sharp} [\![e]\!]^{\sharp}\, D
\end{array}
$$

A remark about the notation:
$[\![\ ]\!]$ is given in a ”distributed” form; its first argument appears between the brackets, the second follows the brackets.

Abstract operators working on signs (Addition)

$+^{\#}$	{0}	{+}	{-}	{-, 0}	{-, +}	{0, +}	{-, 0, +}
{0}	{0}	{+}					
{+}							
{-}							
{-, 0}							
{-, +}							
{0, +}							
{-, 0, +}	{-, 0, +}						

Abstract operators working on signs (Multiplication)

$\times^{\#}$	$\{0\}$	$\{+\}$	$\{-\}$	$\cdots$
$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	
$\{+\}$	$\{0\}$	$\{+\}$	$\{-\}$	
$\{-\}$	$\{0\}$	$\{-\}$	$\{+\}$	
$\{-, 0\}$	$\{0\}$	$\{-, 0\}$	$\{0, +\}$	
$\{-, +\}$	$\{0\}$	$\{-, +\}$	$\{-, +\}$	
$\{0, +\}$	$\{0\}$	$\{0, +\}$	$\{-, 0\}$	
$\{-, 0, +\}$	$\{0\}$	$\{-, 0, +\}$	$\{-, 0, +\}$	

Abstract operators working on signs (unary minus)

$-^{\#}$	{0}	{+}	{-}	{-, 0}	{-, +}	{0, +}	{-, 0, +}
	{0}	{-}	{+}	{+, 0}	{-, +}	{0, -}	{-, 0, +}

Working an example: $D = \{x \mapsto \{+\}, y \mapsto \{+\}\}$

$$
\begin{array}{lcl}
[\![x + 7]\!]^\sharp\, D & = & [\![x]\!]^\sharp\, D \;+^\sharp\; [\![7]\!]^\sharp\, D \\
 & = & \{+\} \;+^\sharp\; \{+\} \\
 & = & \{+\} \\[1em]
[\![x + (-y)]\!]^\sharp\, D & = & \{+\} \;+^\sharp\; (-^\sharp [\![y]\!]^\sharp\, D\,) \\
 & = & \{+\} \;+^\sharp\; (-^\sharp \{+\}) \\
 & = & \{+\} \;+^\sharp\; \{-\} \\
 & = & \{+, -, 0\}
\end{array}
$$

$[\![lab]\!]^\sharp$ is the abstract edge effects associated with edge k.

It depends only on the label *lab*:

$$
\begin{array}{lll}
[\![;]\!]^\sharp\, D & = & D \\
[\![\text{true}\,(e)]\!]^\sharp\, D & = & D \\
[\![\text{false}\,(e)]\!]^\sharp\, D & = & D \\
[\![x = e;]\!]^\sharp\, D & = & D \oplus \{x \mapsto [\![e]\!]^\sharp\, D\} \\
[\![x = M[e];]\!]^\sharp\, D & = & D \oplus \{x \mapsto \{+, -, 0\}\} \\
[\![M[e_1] = e_2;]\!]^\sharp\, D & = & D
\end{array}
$$

... whenever $\qquad D \neq \bot$

These edge effects can be composed to the effect of a path $\pi = k_1 \ldots k_r$:

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

Consider a program node v:

- $\rightarrow$ For every path π from program entry *start* to v the analysis should determine for each program variable x the set of all signs that the values of x may have at v as a result of executing π.
- $\rightarrow$ Initially at program start, no information about signs is available.
- $\rightarrow$ The analysis computes a superset of the set of signs as safe information.

$\Longrightarrow$ For each node v, we need the set:

$$\mathcal{S}[v] \quad = \quad \bigcup\{[\![\pi]\!]^\sharp \top \mid \pi : \mathit{start} \rightarrow^* v\}$$

where $\top$ is the function bindig all variables to $\{-, 0, +\}$.

This function describes that we don't know the sign of any variable at program entry.

Question:

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

Question:

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

Collect all constraints on the values of $\mathcal{S}[u]$ into a system of constraints:

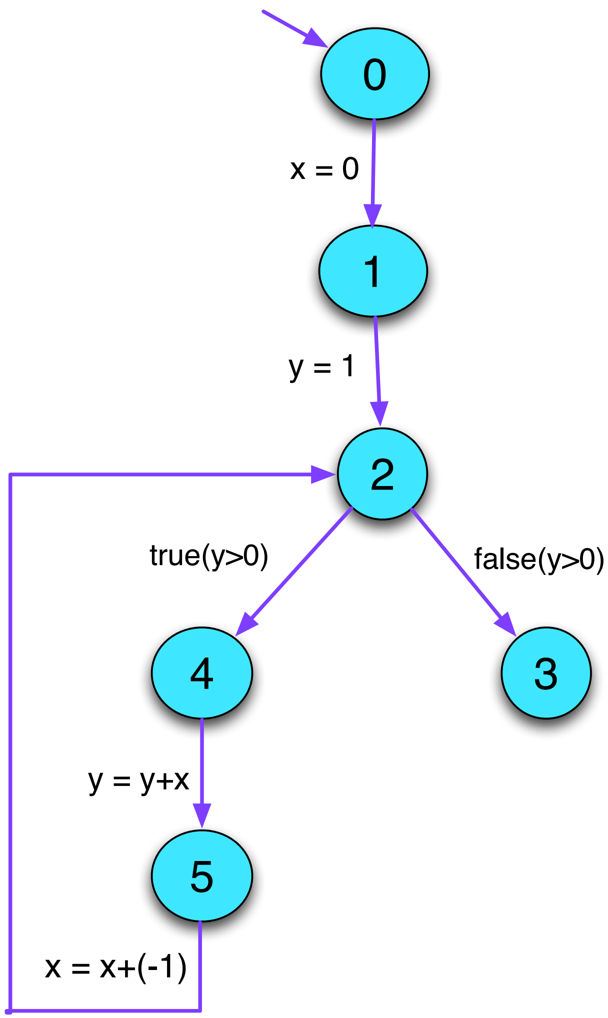
$$\begin{array}{lll} \mathcal{S}[start] & \supseteq & \top \\ \mathcal{S}[v] & \supseteq & [\![k]\!]^\sharp\,(\mathcal{S}[u]) \qquad\qquad k = (u, \_, v) \quad \text{edge} \end{array}$$

Why $\supseteq$?

Wanted:

- a least solution (why least?)
- an algorithm that computes this solution

Example:



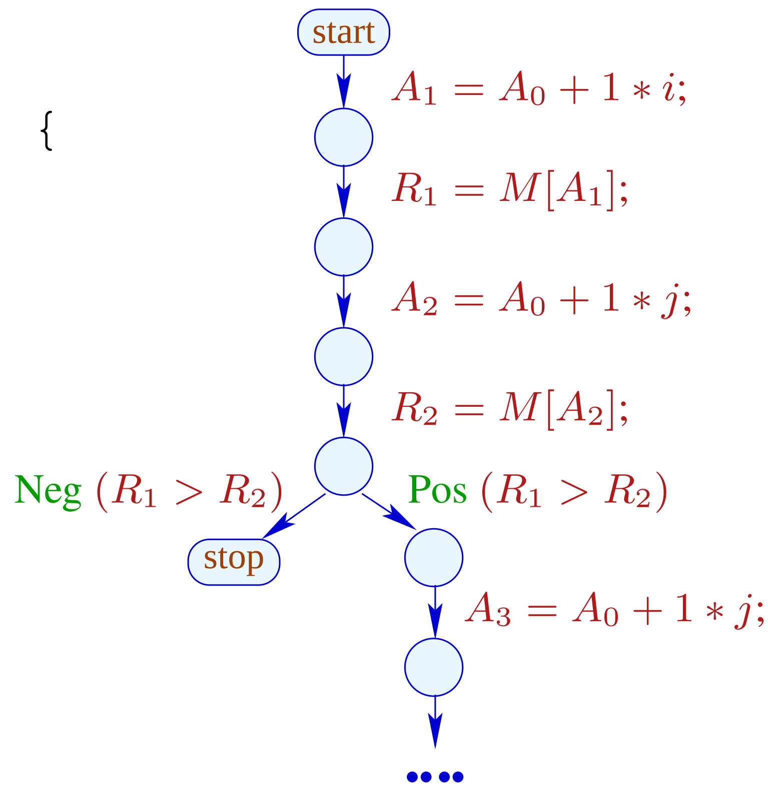
$$\begin{array}{lcl}
\mathcal{S}[0] & \supseteq & \top \\
\mathcal{S}[1] & \supseteq & \mathcal{S}[0] \oplus \{x \mapsto \{0\}\} \\
\mathcal{S}[2] & \supseteq & \mathcal{S}[1] \oplus \{y \mapsto \{+\}\} \\
\mathcal{S}[2] & \supseteq & \mathcal{S}[5] \oplus \{x \mapsto [\![x + (-1)]\!]^\sharp\, \mathcal{S}[5]\} \\
\mathcal{S}[3] & \supseteq & \mathcal{S}[2] \\
\mathcal{S}[4] & \supseteq & \mathcal{S}[2] \\
\mathcal{S}[5] & \supseteq & \mathcal{S}[4] \oplus \{y \mapsto [\![y + x]\!]^\sharp\, \mathcal{S}[4]\}
\end{array}$$

3 An Operational Semantics

Programs are represented as control-flow graphs.

Example:

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



Thereby, represent:

vertex	program point
start	program start
stop	program exit
edge	labeled with a statement or a condition

Thereby, represent:

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

Edge Labelings:

Test : Pos (e) or Neg (e) (better true(e) or false(e))

Assignment : $R = e;$

Load : $R = M[e];$

Store : $M[e_1] = e_2;$

Nop : ;

Execution of a path is a computation.

A computation transforms a state $s = (\rho, \mu)$ where:

$\rho : \mathit{Vars} \to \textbf{int}$	values of variables (contents of symbolic registers)
$\mu : \mathbb{N} \to \textbf{int}$	contents of memory

Every edge $k = (u, lab, v)$ defines a partial transformation

$$[\![k]\!] = [\![lab]\!]$$

of the state:

$$
\begin{aligned}
[\![;]\!]\,(\rho,\mu) &= (\rho,\mu) \\
\\
[\![\mathrm{true}\,(e)]\!]\,(\rho,\mu) &= (\rho,\mu) && \text{if } [\![e]\!]\,\rho \neq 0 \\
[\![\mathrm{false}\,(e)]\!]\,(\rho,\mu) &= (\rho,\mu) && \text{if } [\![e]\!]\,\rho = 0
\end{aligned}
$$

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

$$[\![\text{true}\,(e)]\!]\,(\rho, \mu) \quad = \quad (\rho, \mu) \qquad \textbf{if } [\![e]\!]\,\rho \neq 0$$

$$[\![\text{false}\,(e)]\!]\,(\rho, \mu) \quad = \quad (\rho, \mu) \qquad \textbf{if } [\![e]\!]\,\rho = 0$$

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

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

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

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

$$[\![\text{true}\,(e)]\!]\,(\rho,\mu) = (\rho,\mu) \qquad \textbf{if } [\![e]\!]\,\rho \neq 0$$

$$[\![\text{false}\,(e)]\!]\,(\rho,\mu) = (\rho,\mu) \qquad \textbf{if } [\![e]\!]\,\rho = 0$$

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

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

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

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

// where "$\oplus$" modifies a mapping at a given argument

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

$$[\![M[e_1] = e_2;]\!]\,(\rho, \mu) \quad = \quad (\rho, \boxed{\mu \oplus \{[\![e_1]\!]\,\rho \mapsto [\![e_2]\!]\,\rho\}})$$

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 $\pi = k_1 k_2 \dots k_m$ defines a computation in the state s if

$$s \in \mathit{def}\,([\![k_m]\!] \circ \ldots \circ [\![k_1]\!])$$

The result of the computation is $[\![\pi]\!]\,s = ([\![k_m]\!] \circ \ldots \circ [\![k_1]\!])\,s$

The approach:

A static analysis needs to collect correct and hopefully precise information about a program in a terminating computation.

Concepts:

- partial orders relate information for their contents/quality/precision,
- least upper bounds combine information in the best possible way,
- monotonic functions preserve the order, prevent loss of collected information, prevent oscillation.

4 Complete Lattices

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

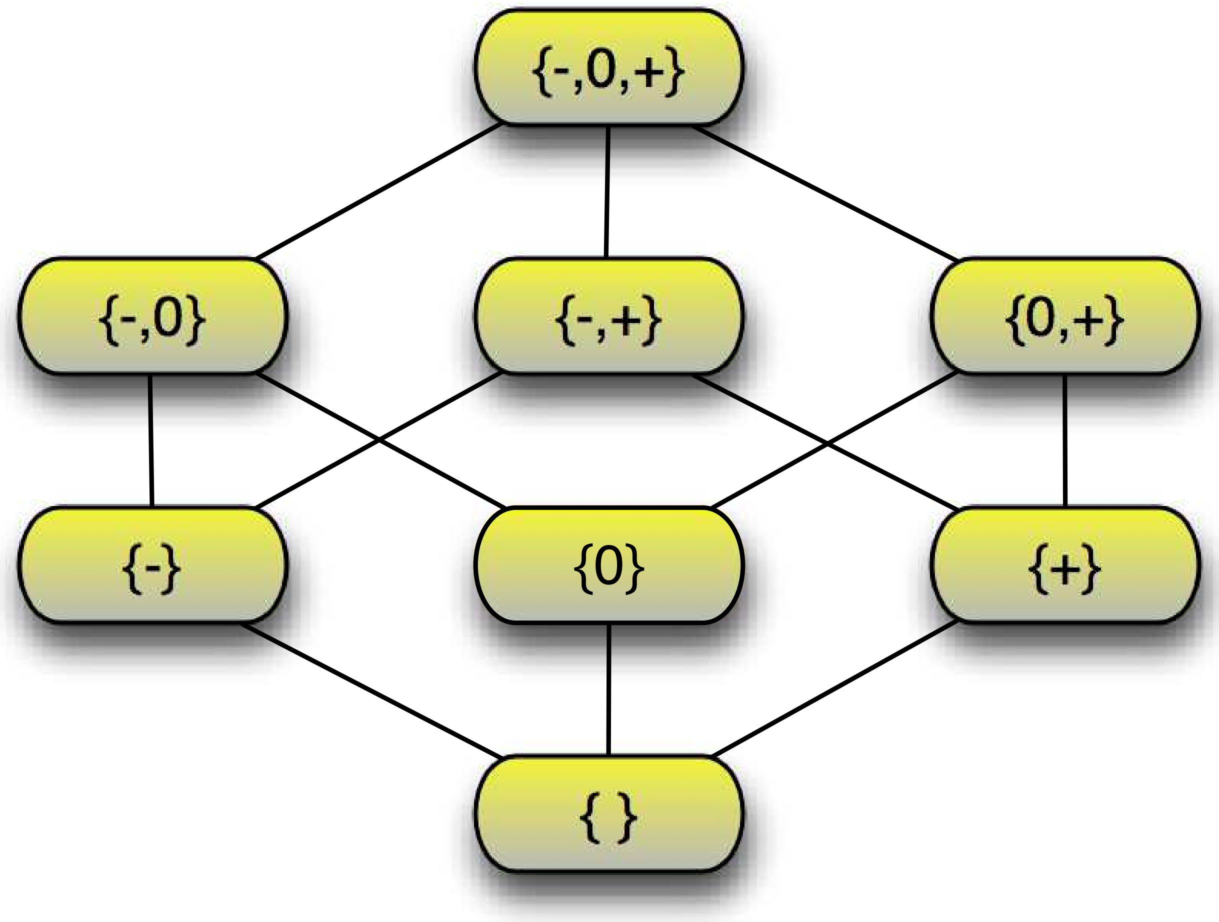
$$
\begin{array}{lll}
a \sqsubseteq a & & \textit{reflexivity} \\
a \sqsubseteq b \wedge b \sqsubseteq a & \Longrightarrow \quad a = b & \textit{anti-symmetry} \\
a \sqsubseteq b \wedge b \sqsubseteq c & \Longrightarrow \quad a \sqsubseteq c & \textit{transitivity}
\end{array}
$$

Intuition: $\sqsubseteq$ represents precision.

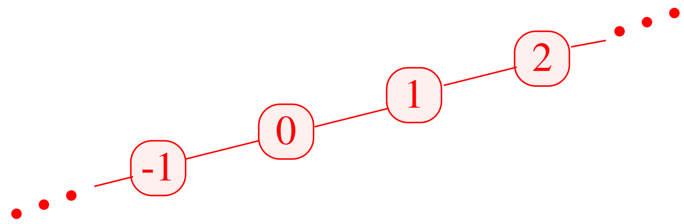
By convention: $a \sqsubseteq b$ means a is at least as precise as b.

Examples:

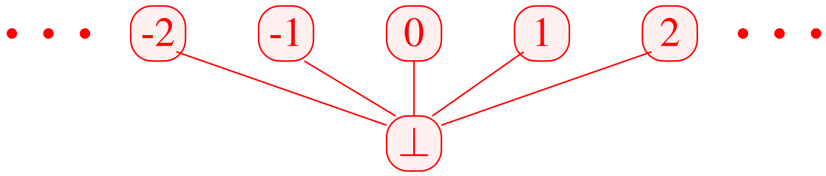
1. The rules-of-sign analysis uses the following lattice $\mathbb{D} = 2^{\{-,0,+\}}$ with the relation "$\subseteq$" :



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



3. $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$ 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.

The least upper bound is the youngest common ancestor in the pictorial representation of lattices.

Intuition: It is the best combined information for X.

Caveat:

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

A partially ordered set $\mathbb{D}$ is a complete lattice (cl) if every subset $X \subseteq \mathbb{D}$ has a least upper bound $\bigsqcup X \in \mathbb{D}$.

Note:

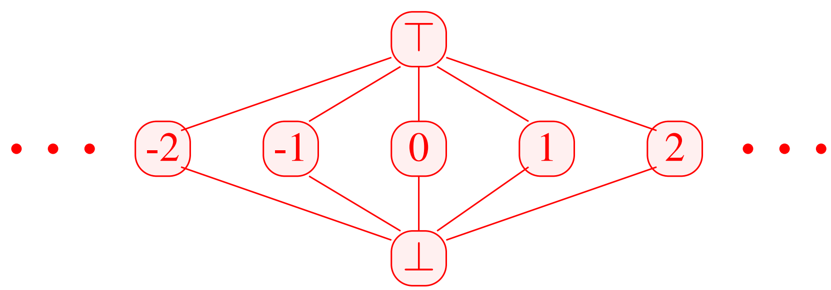
Every complete lattice has

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

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

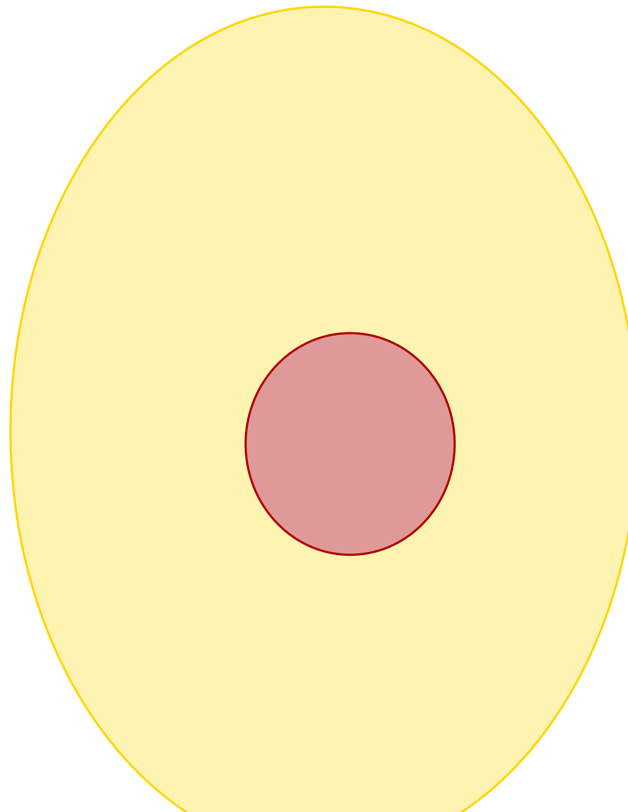
Examples:

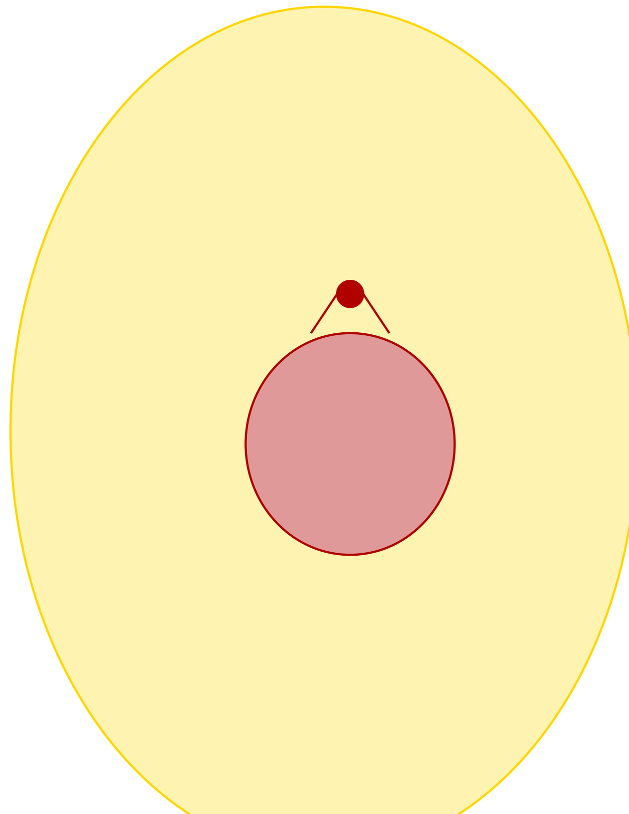
1. $\mathbb{D} = 2^{\{-,0,+\}}$ with $\sqsubseteq$ is a complete lattice
2. $\mathbb{D} = \mathbb{Z}$ with "$\leq$" is not a complete lattice.
3. $\mathbb{D} = \mathbb{Z}_\bot$ is also not a complete lattice
4. With an extra element $\top$, we obtain the flat lattice $\mathbb{Z}_\bot^\top = \mathbb{Z} \cup \{\bot, \top\}$:

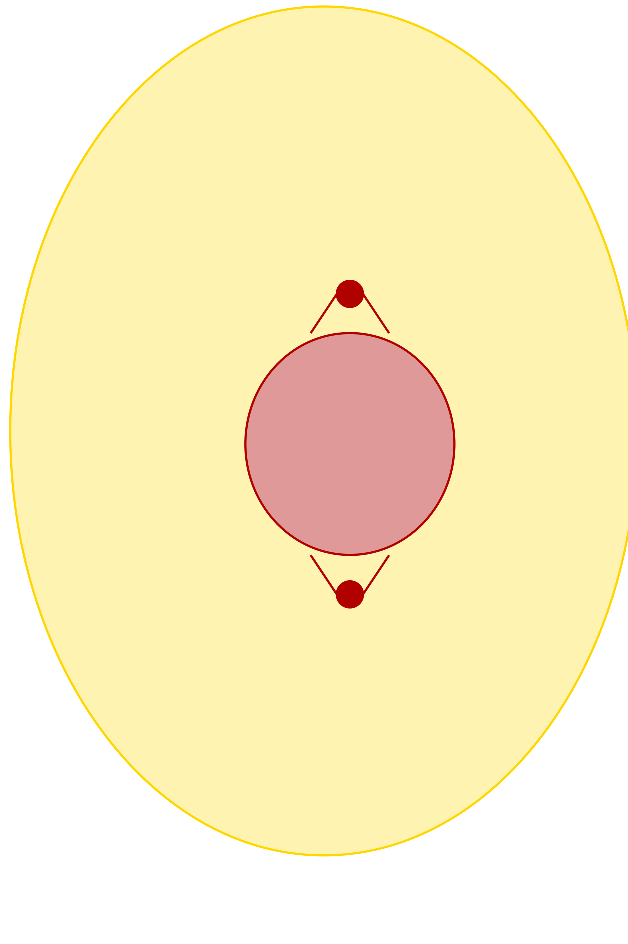


Theorem:

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







Back to the system of constraints for Rules-of-Signs Analysis!

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

Combine all constraints for each variable v by applying the least-upper-bound operator $\bigsqcup$:

$$\mathcal{S}[v] \quad \sqsupseteq \quad \bigsqcup\{[\![k]\!]^\sharp\,(\mathcal{S}[u]) \mid k = (u, \_, v) \ \text{ edge}\}$$

Correct because:

$$x \sqsupseteq d_1 \wedge \ldots \wedge x \sqsupseteq d_k \quad \text{iff} \quad x \sqsupseteq \bigsqcup\{d_1, \ldots, d_k\}$$

Our generic form of the systems of constraints:

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

Relation to the running example:

x_i	unknown	here:	$\mathcal{S}[u]$
$\mathbb{D}$	values	here:	$Signs$
$\sqsubseteq \quad \subseteq \quad \mathbb{D} \times \mathbb{D}$	ordering relation	here:	$\subseteq$
f_i: $\mathbb{D}^n \to \mathbb{D}$	constraint	here:	...

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

A mapping $f : \mathbb{D}_1 \to \mathbb{D}_2$ is called monotonic (order preserving) 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 (order preserving) 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

(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 (order preserving) 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

(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.
- $\mathsf{inv}\,x = -x$ is not monotonic

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

Wanted: least 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: least 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 $F : \mathbb{D}^n \to \mathbb{D}^n$ where

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

Wanted: least 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 $F : \mathbb{D}^n \to \mathbb{D}^n$ 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

Wanted: least 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 $F : \mathbb{D}^n \to \mathbb{D}^n$ 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 from below. We construct:

  $$\underline{\perp}, \quad F\,\underline{\perp}, \quad F^2\,\underline{\perp}, \quad F^3\,\underline{\perp}, \quad \ldots$$

Intuition: This iteration eliminates unjustified assumptions.

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

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

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

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

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

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

Theorem

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

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

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

If $\mathbb{D}$ is finite, a solution can be found that is definitely the least solution.

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 fixed point $d_0 \in \mathbb{D}$.

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

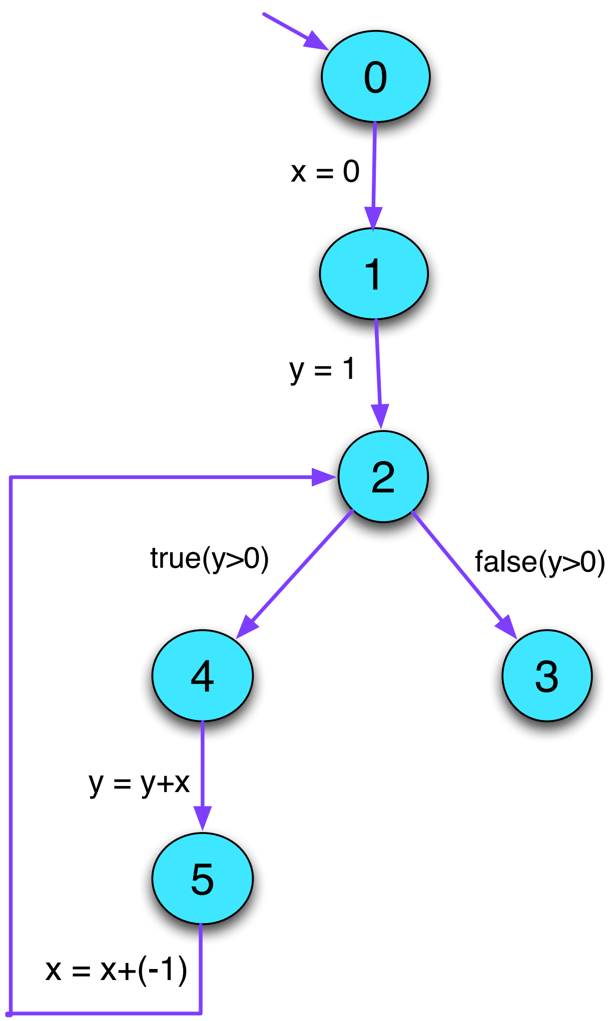
f	$f^k\,\bot$	$f^k\,\top$
0	$\emptyset$	U
1	b	$a \cup b$
2	b	$a \cup b$

Conclusion:

Systems of inequalities can be solved through fixed-point iteration, i.e., by repeated evaluation of right-hand sides

Caveat: Naive fixed-point iteration is rather inefficient

Example:

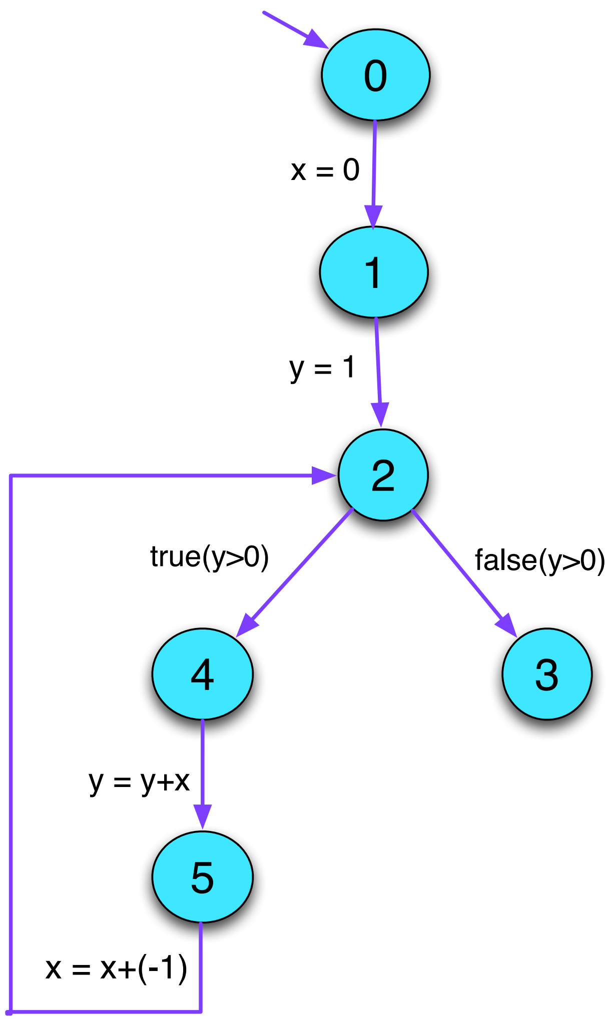


0		1		2		3		4		5	
x	y	x	y	x	y	x	y	x	y	x	y

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	
x	y	x	y	x	y	x	y	x	y	x	y

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

```
for (i = 1; i ≤ n; i++) x_i = ⊥;
do {
        finished = true;
        for (i = 1; i ≤ n; i++) {
                new = f_i(x_1, ..., x_n);
                if (!(x_i ⊒ new)) {
                        finished = false;
                        x_i = x_i ⊔ new;
                }
        }
} while (!finished);
```

What we have learned:

- The information derived by static program analysis is partially ordered in a complete lattice.
- the partial order represents information content/precision of the lattice elements.
- least upper-bound combines information in the best possible way.
- Monotone functions prevent loss of information.

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

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

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

Wanted: MOP (Merge Over all Paths)

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

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 $\pi : start \to^* v$

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

Answer: In general: yes

Notable exception, all functions $[\![k]\!]^\sharp$ are distributive.

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 $\emptyset \neq X \subseteq \mathbb{D}$;

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

Theorem Kildall 1972

Assume all v are reachable from *start*.

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

Question: Are the edge effects of the Rules-of-Sign analysis distributive?

5 Constant Propagation

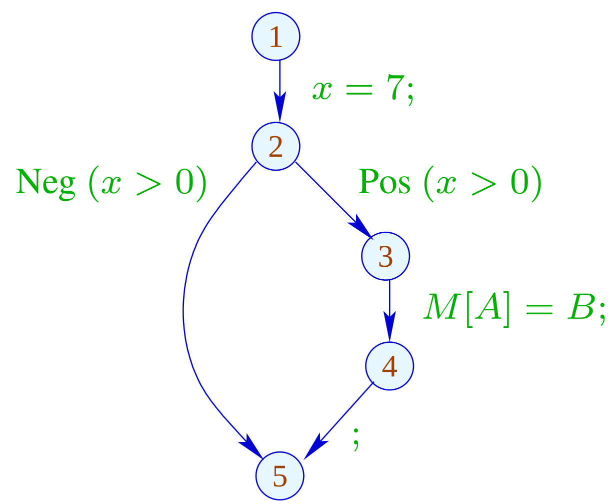
Goal: Execute as much of the code at compile-time as possible!

Example:

$x = 7;$

$\text{if } (x > 0)$

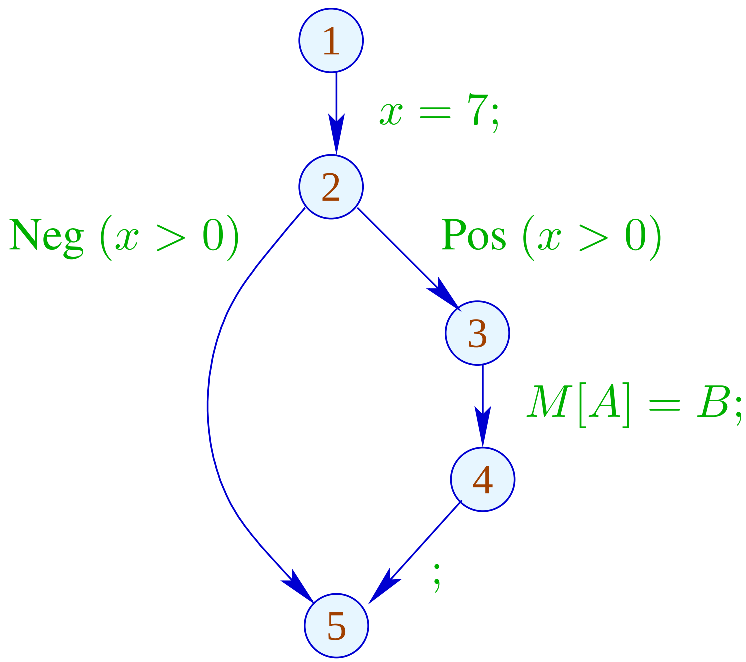
$\quad M[A] = B;$



Obviously, x has always the value 7

Thus, the memory access is always executed

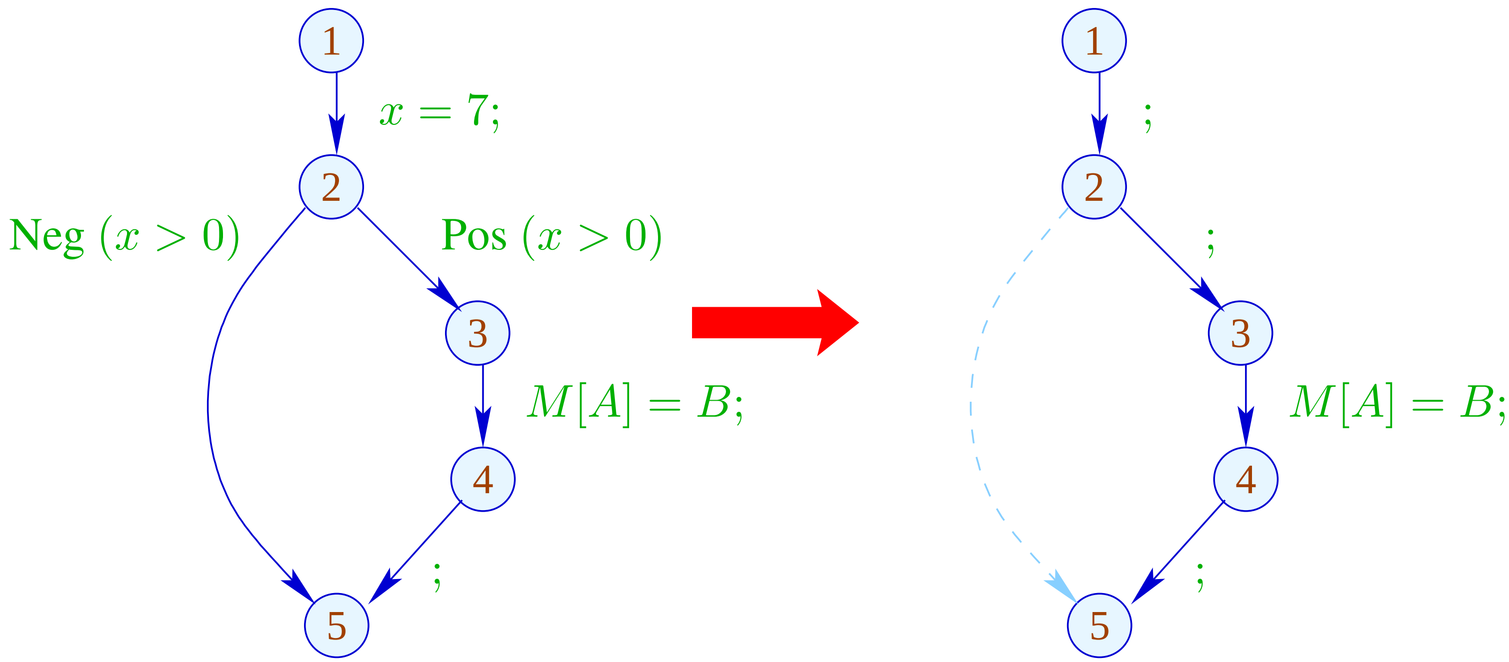
Goal:



Obviously, x has always the value 7

Thus, the memory access is always executed

Goal:



Idea:

Design an analysis that for every program point u determines the values that variables definitely have at u;

As a side effect, it also tells whether u can be reached at all

Idea:

Design an analysis that for every program point u, determines the values that variables definitely have at u;

As a side effect, it also tells whether u can be reached at all

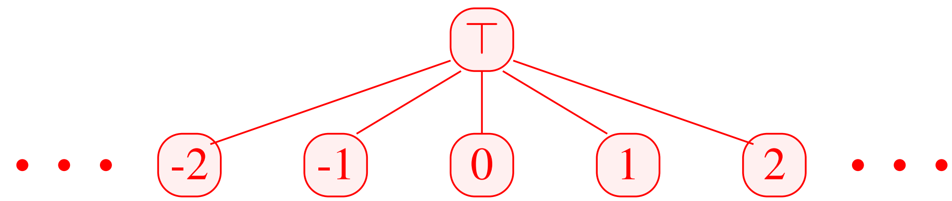
We need to design a complete lattice for this analysis.

It has a nice relation to the operational semantics of our tiny programming language.

As in the case of the Rules-of-Signs analysis the complete lattice is constructed in two steps.

(1) The potential values of variables:

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



Caveat: $\mathbb{Z}^\top$ 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

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

Obviously, $[\![lab]\!]^\sharp \bot = \bot$ for all lab

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

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:

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

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

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

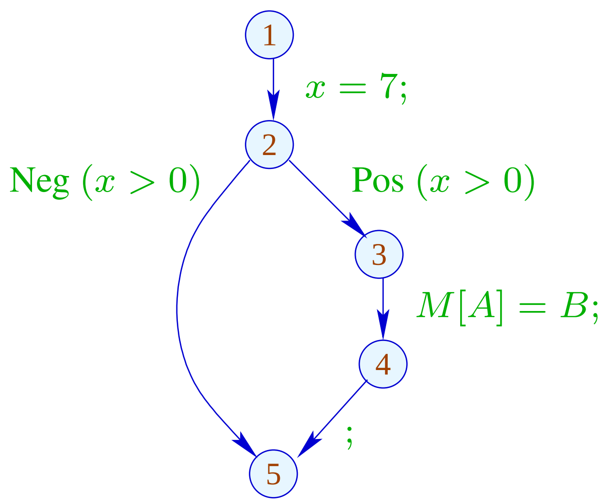
Thus, we obtain the following abstract edge effects $[\![lab]\!]^\sharp$:

$$
\begin{array}{lll}
[\![;]\!]^\sharp\, D & = & D \\
[\![\text{true}\,(e)]\!]^\sharp\, D & = & \begin{cases} \bot & \text{if} \quad 0 = [\![e]\!]^\sharp\, D \quad \text{definitely false} \\ D & \text{otherwise} \qquad\qquad\;\; \text{possibly true} \end{cases} \\
[\![\text{false}\,(e)]\!]^\sharp\, D & = & \begin{cases} D & \text{if} \quad 0 \sqsubseteq [\![e]\!]^\sharp\, D \quad \text{possibly false} \\ \bot & \text{otherwise} \qquad\qquad\;\; \text{definitely true} \end{cases} \\
[\![x = e;]\!]^\sharp\, D & = & D \oplus \{x \mapsto [\![e]\!]^\sharp\, D\} \\
[\![x = M[e];]\!]^\sharp\, D & = & D \oplus \{x \mapsto \top\} \\
[\![M[e_1] = e_2;]\!]^\sharp\, D & = & D
\end{array}
$$

... whenever $D \neq \bot$

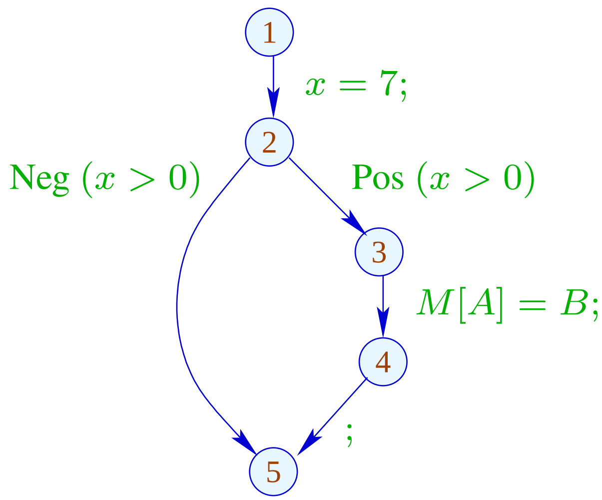
At *start*, we have $D_\top = \{x \mapsto \top \mid x \in \mathit{Vars}\}$.

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 form 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 the concrete 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 Bindings: $\Delta \subseteq (\mathit{Vars} \to \mathbb{Z}) \times (\mathit{Vars} \to \mathbb{Z}^\top)_\perp$

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

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:

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

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

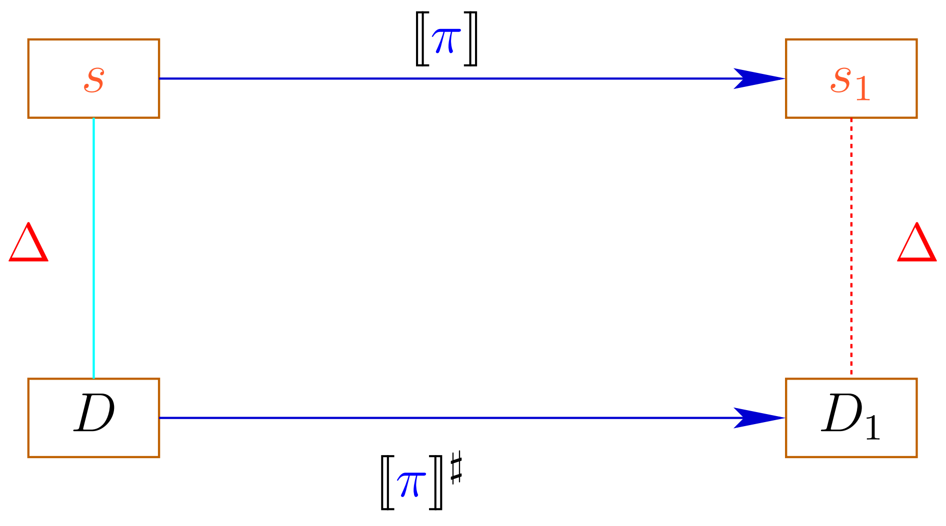
Concretization:

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

We show correctness:

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

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



The abstract semantics simulates the concrete semantics

In particular:

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

The abstract semantics simulates the concrete semantics

In particular:

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

In practice, this means for example that $D\, x = -7$ implies:

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

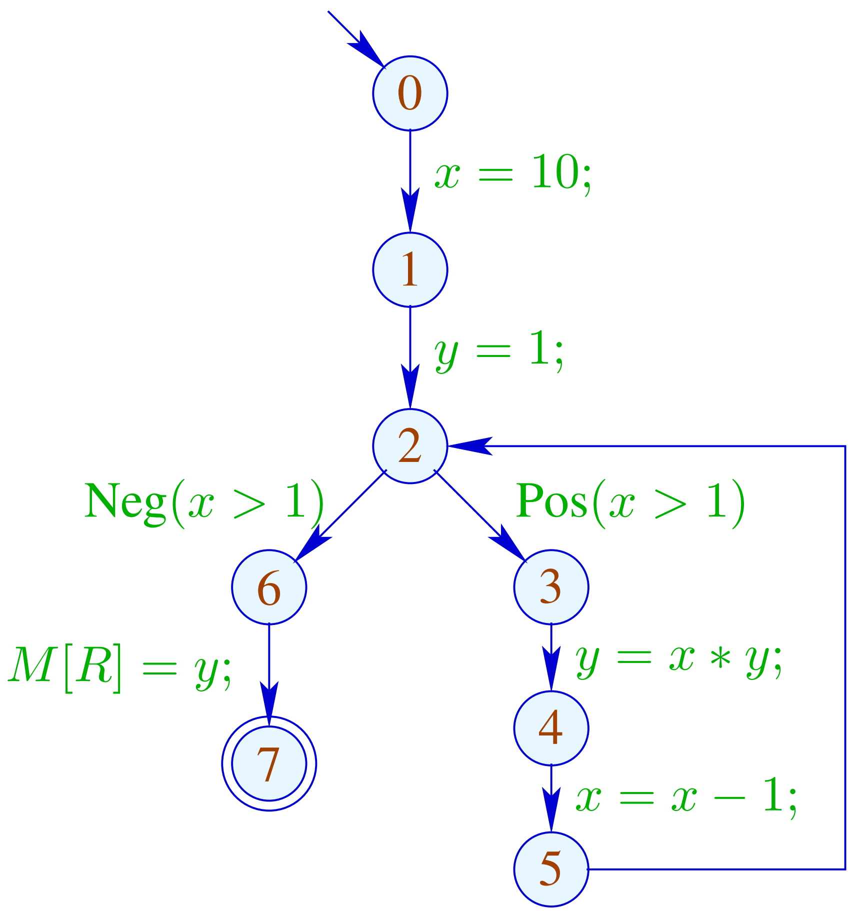
The MOP-Solution:

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

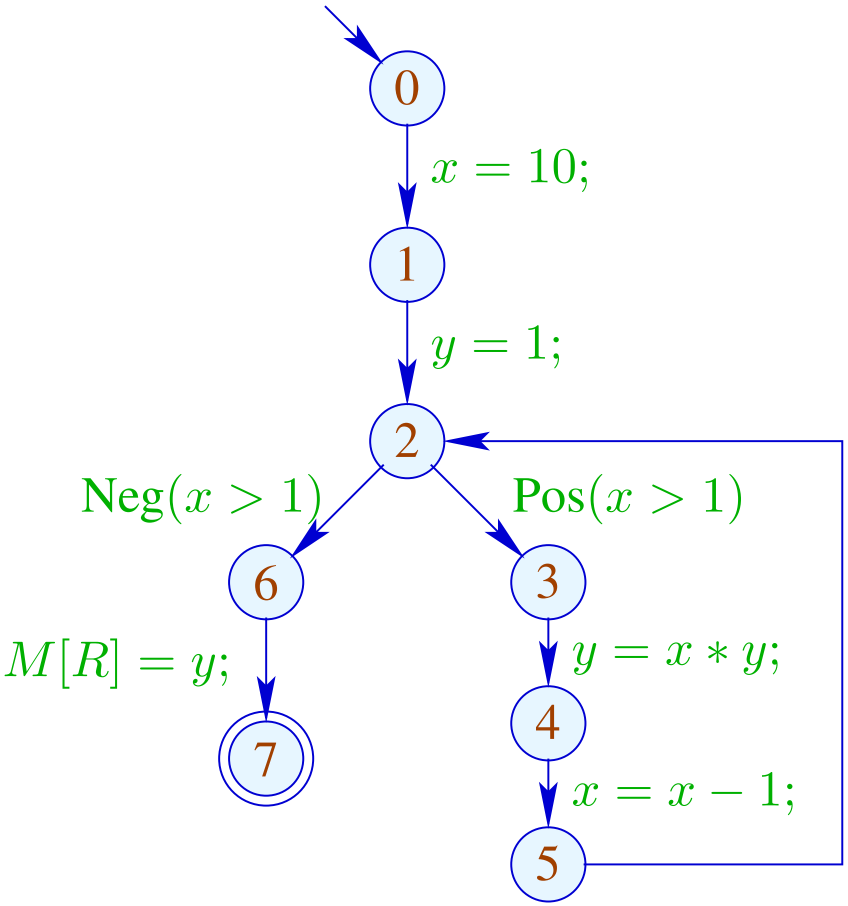
where $\quad D_\top\, x = \top \qquad (x \in Vars)$.

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

Example:

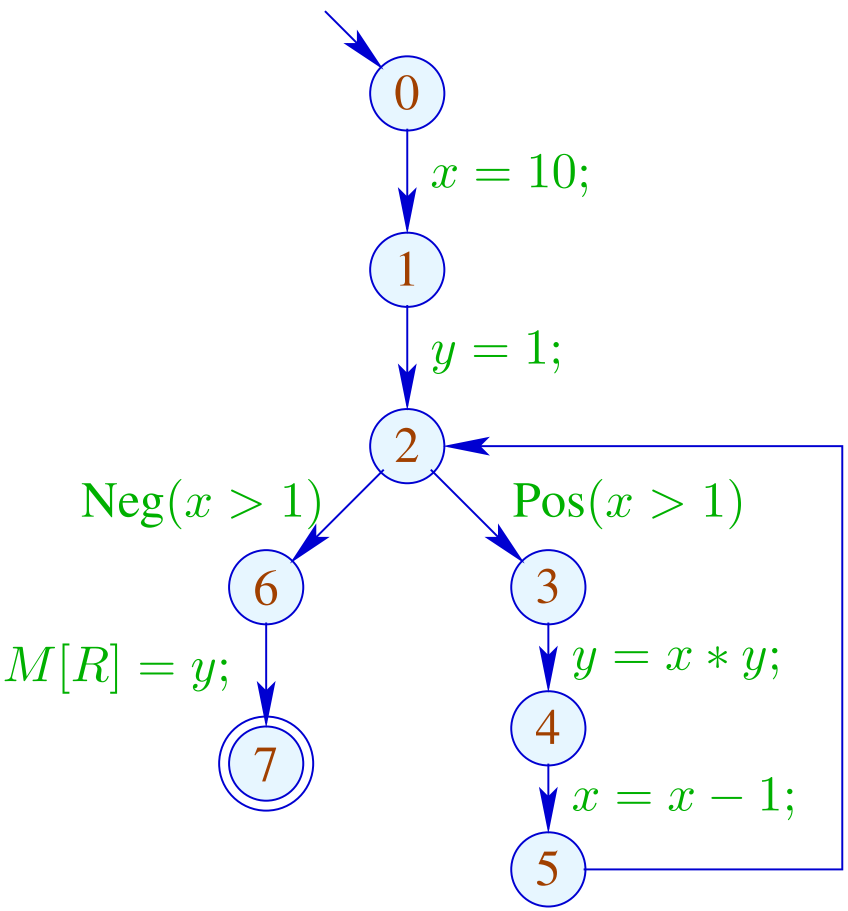


Example:



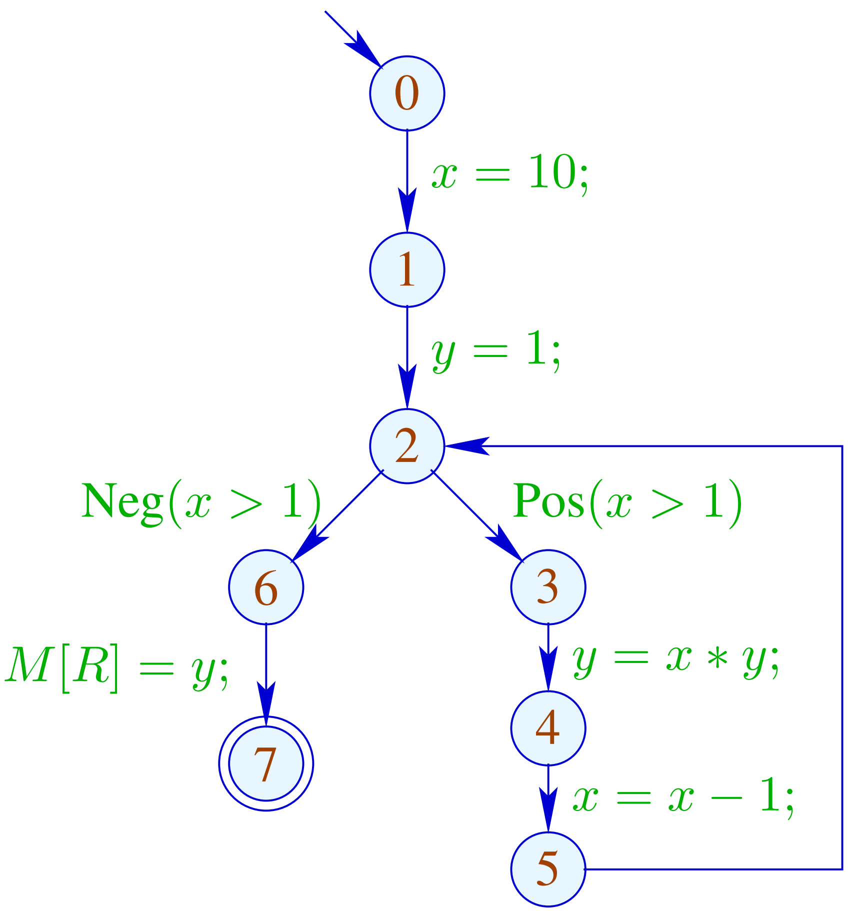
	1	
	x	y
0	$\top$	$\top$
1	10	$\top$
2	10	1
3	10	1
4	10	10
5	9	10
6	$\bot$	
7	$\bot$	

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	$\perp$		$\top$	$\top$
7	$\perp$		$\top$	$\top$

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	$\bot$		$\top$	$\top$		

Concrete vs. Abstract Execution:

Although program and all initial values are given, abstract execution does not compute the result!

On the other hand, fixed-point iteration is guaranteed to terminate:

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

Observation: The effects of edges are not distributive!

Counterexample: $f = [\![x = x + y;]\!]^\sharp$

$$\begin{array}{lrcl} \text{Let} & D_1 & = & \{x \mapsto 2, y \mapsto 3\} \\ & D_2 & = & \{x \mapsto 3, y \mapsto 2\} \\ \text{Then} & f\,D_1 \sqcup f\,D_2 & = & \{x \mapsto 5, y \mapsto 3\} \sqcup \{x \mapsto 5, y \mapsto 2\} \\ & & = & \{x \mapsto 5, y \mapsto \top\} \\ & & \neq & \{x \mapsto \top, y \mapsto \top\} \\ & & = & f\,\{x \mapsto \top, y \mapsto \top\} \\ & & = & f\,(D_1 \sqcup D_2) \end{array}$$

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.,

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

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

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

whenever $[\![\pi]\!] (\rho, \mu)$ is defined

6 Interval Analysis

Constant propagation attempts to determine values of variables.

However, variables may take on several values during program execution.

So, *the value* of a variable will often be unknown.

Next attempt: determine an interval enclosing all possible values that a variable may take on during program execution at a program point.

Example:

$$
\begin{array}{l}
\textsf{for } (i = 0; i < 42; i{+}{+}) \\
\quad \textsf{if } (0 \leq i \wedge i < 42)\{ \\
\quad\quad A_1 = A + i; \\
\quad\quad M[A_1] = i; \\
\quad \}
\end{array}
$$

// A start address of an array

// if-statement does array-bounds check

Obviously, the inner check is superfluous.

Idea 1:

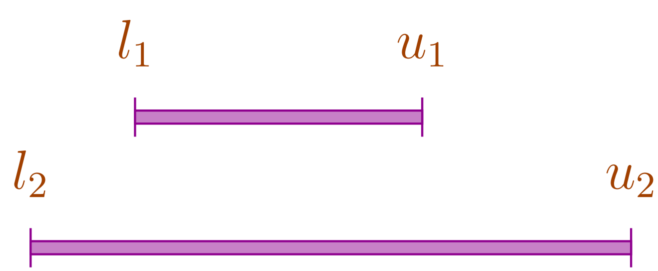
Determine for every variable x the tightest possible interval of potential values.

Abstract domain:

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

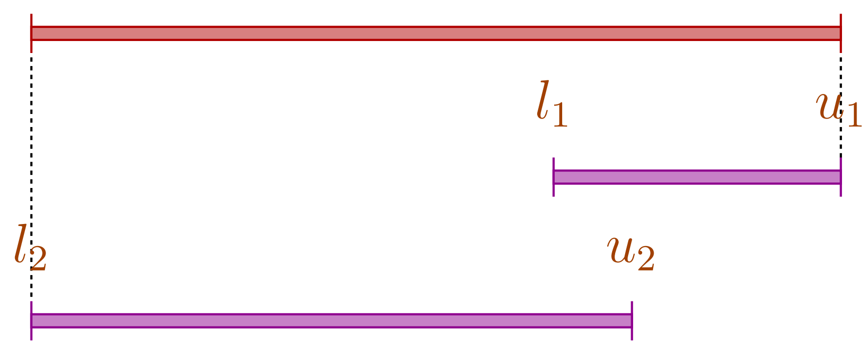
Partial order:

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



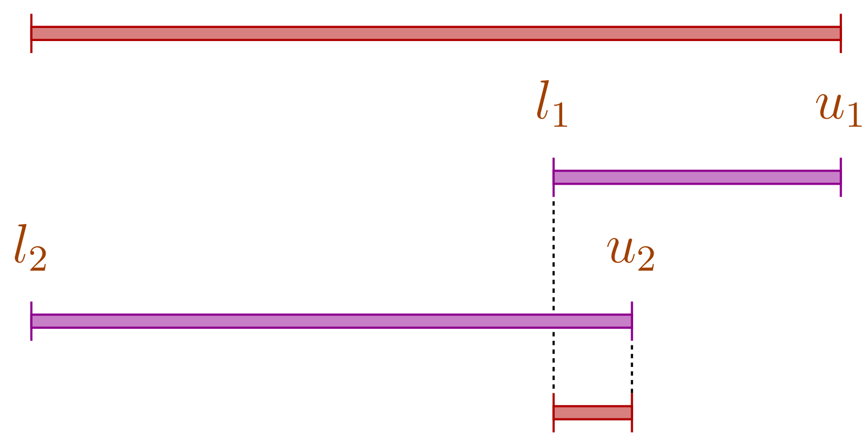
Thus:

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



Thus:

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



Caveat:

- $\rightarrow$ $\mathbb{I}$ is not a complete lattice,
- $\rightarrow$ $\mathbb{I}$ has infinite ascending chains, e.g.,

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

Caveat:

→ $\mathbb{I}$ 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:

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

Computing with intervals: Interval Arithmetic.

Addition:

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

Negation:

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

Multiplication:

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

Example:

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

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

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

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

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

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

Equality:

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

Equality:

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

Example:

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

Less:

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

Less:

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

Example:

$$\begin{aligned} [1, 2] <^\sharp [9, 42] &= true \\ [0, 7] \ <^\sharp \ [0, 7] &= \top \\ [3, 4] \ <^\sharp \ [1, 2] &= false \end{aligned}$$

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:

$$
\begin{array}{lcl}
[\![;]\!]^\sharp\, D & = & D \\
[\![x = e;]\!]^\sharp\, D & = & D \oplus \{x \mapsto [\![e]\!]^\sharp\, D\} \\
[\![x = M[e];]\!]^\sharp\, D & = & D \oplus \{x \mapsto \top\} \\
[\![M[e_1] = e_2;]\!]^\sharp\, D & = & D \\
[\![\text{true}\,(e)]\!]^\sharp\, D & = & \begin{cases} \bot & \text{if} \qquad \text{definitely false} \\ D & \text{otherwise} \qquad \text{possibly true} \end{cases} \\
[\![\text{false}\,(e)]\!]^\sharp\, D & = & \begin{cases} D & \text{if} \qquad \text{possibly false} \\ \bot & \text{otherwise} \qquad \text{definitely true} \end{cases}
\end{array}
$$

... given that $\quad D \neq \bot$

Better Exploitation of Conditions:

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

where :

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

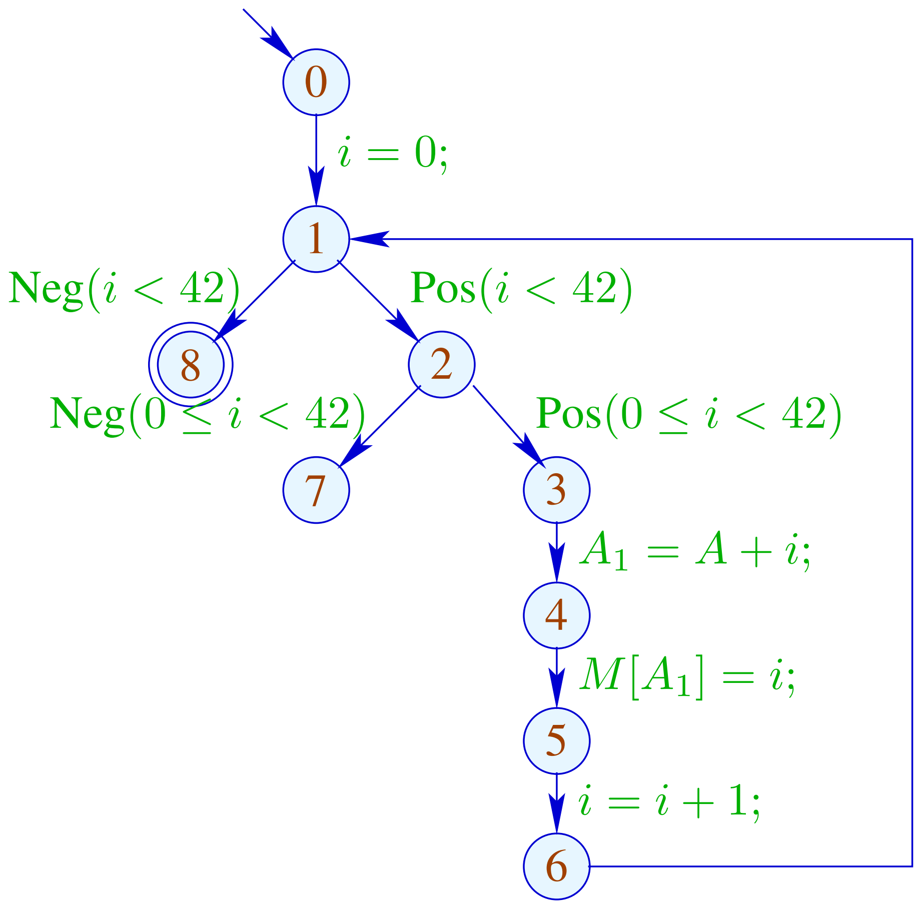
Better Exploitation of Conditions (cont.):

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

where :

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

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	$\perp$	
8	42	42

Problem:

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

→ On some programs, iteration may never terminate.

Idea: Widening

Accelerate the iteration — at the cost of precision

Formalization of the Approach:

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

denote a system of constraints over $\mathbb{D}$

Define an accumulating iteration:

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

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., $\underline{x} \sqsubseteq G\,\underline{x}$ for all $\underline{x} \in \mathbb{D}^n$.

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

$$\underline{\perp} \sqsubseteq G \underline{\perp} \sqsubseteq \ldots \sqsubseteq G^k \underline{\perp} \sqsubseteq \ldots$$

(d) If $G^k \underline{\perp} = G^{k+1} \underline{\perp} = \underline{y}$, 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:

$$x_i = x_i \sqcup f_i(x_1, \ldots, x_n) \;, \quad i = 1, \ldots, n \qquad (3)$$

for a binary operation widening:

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

(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 $\sqcup$ is defined by:

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

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

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

Example:

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

→ 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:

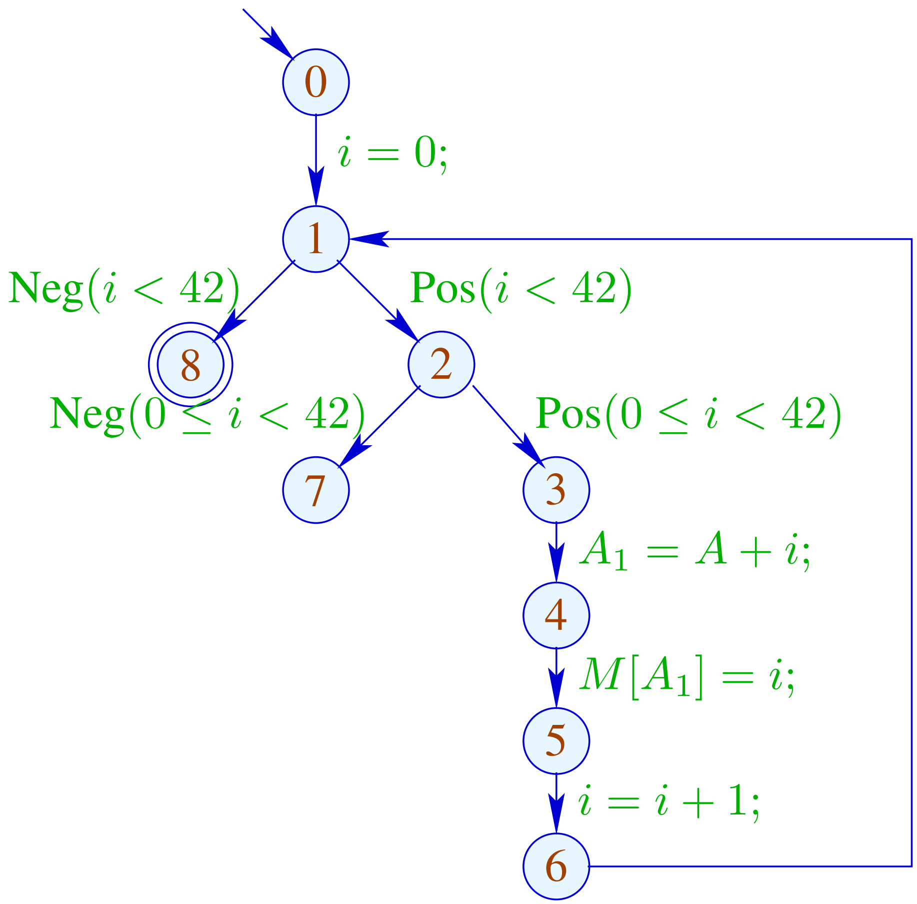
- In order to determine a solution of (1) over a complete lattice with infinite ascending chains, we define a suitable widening and then solve (3)
- Caveat: The construction of suitable widenings is a dark art !!!

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

  $\rightarrow$ the abstract values do not get too complicated;

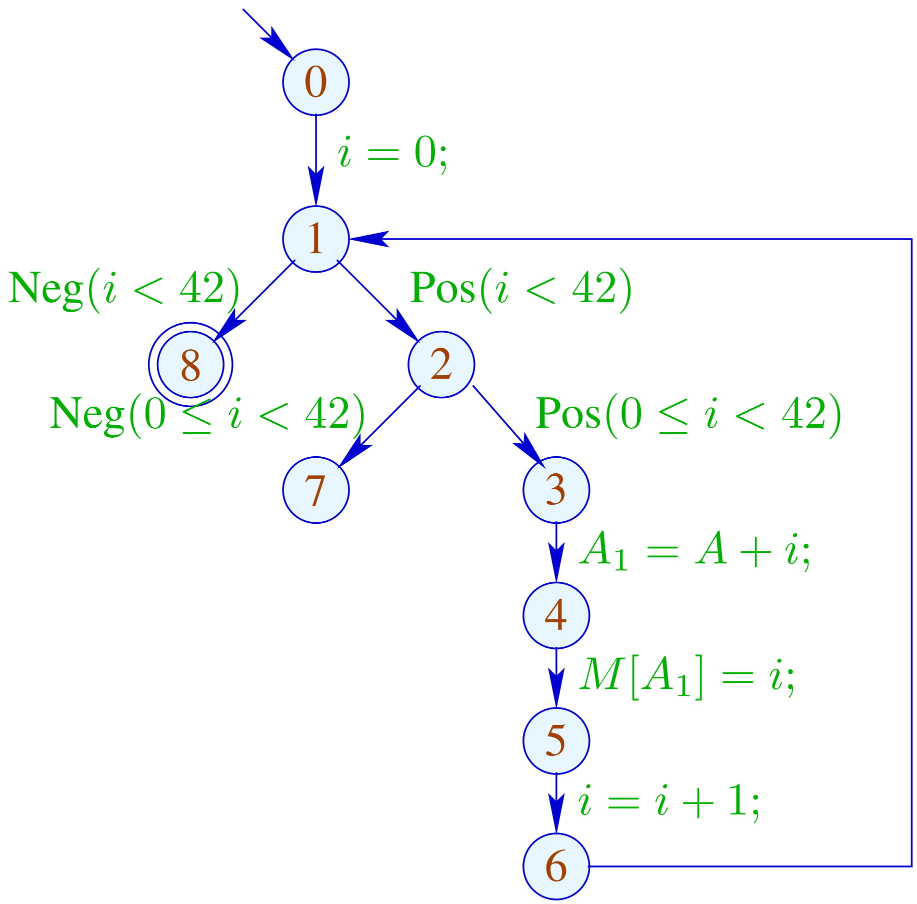
  $\rightarrow$ the number of updates remains bounded ...

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	$\bot$		42	$+\infty$		
8	$\bot$		42	$+\infty$		

7 Removing superfluous computations

A computation may be superfluous because

- the result is already available, $\longrightarrow$ available-expression analysis, or
- the result is not needed $\longrightarrow$ live-variable analysis.

7.1 Redundant computations

Idea:

If an expression at a program point is guaranteed to be computed to the value it had before, then

$\rightarrow$ store this value after the first computation;

$\rightarrow$ replace every further computation through a look-up

Question to be answered by static analysis: Is an expression available?

Problem: Identify sources of redundant computations!

Example:

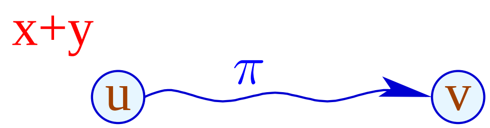
$$\begin{array}{lrcl} & z & = & 1; \\ & y & = & M[17]; \\ A: & x_1 & = & \boxed{y+z}; \\ & & \dots & \\ B: & x_2 & = & \boxed{y+z}; \end{array}$$

B is a redundant 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

Situation: The value of $x + y$ is computed at program point u



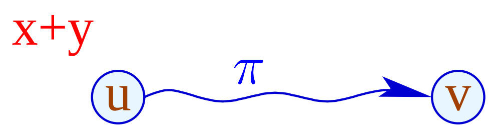
and a computation along path π reaches v where it evaluates again $x + y$

....

If x and y have not been modified in π, then evaluation of $x + y$ at v returns the same value as evaluation at u.

This property can be checked at every edge in π.

Situation: The value of $x + y$ is computed at program point u



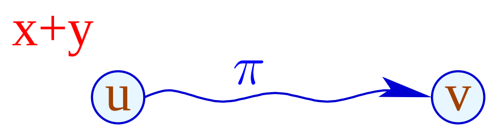
and a computation along path π reaches v where it evaluates again $x + y$ .... If x and y have not been modified in π, then evaluation of $x + y$ at v is known to return the same value as evaluation at u

This property can be checked at every edge in π.

More efficient: Do this check for all expressions occurring in the program in parallel.

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

Situation: The value of $x + y$ is computed at program point u



and a computation along path π reaches v where it evaluates again $x + y$ .... 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.

This property can be checked at every edge in π.

More efficient: Do this check for all expressions occurring in the program in parallel.

Assume that 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.

$[\![k]\!]^\sharp\, A$ is the (abstract) edge effect associated with k

These edge effects can be composed to the effect of a path $\pi = k_1 \ldots k_r$:

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

These edge effects 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., $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$

These edge effects 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:

$$\begin{array}{lll} [\![;]\!]^\sharp\, A & = & A \\ [\![true(e)]\!]^\sharp\, A & = & [\![false(e)]\!]^\sharp\, A \quad = \quad A \cup \{e\} \\ [\![x = e;]\!]^\sharp\, A & = & (A \cup \{e\}) \backslash Expr_x \qquad \text{where} \end{array}$$

$Expr_x$ all expressions that contain x

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

- $\rightarrow$ An expression is available at v if it is available along all paths π to v.
- $\rightarrow$ For every such path π, the analysis determines the set of expressions that are available along π.
- $\rightarrow$ Initially at program start, nothing is available.
- $\rightarrow$ The analysis computes the intersection of the availability sets as safe information.

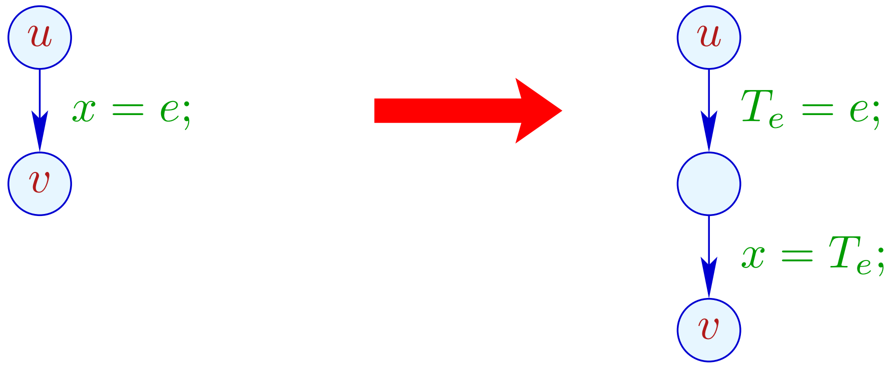
$\Longrightarrow$ For each node v, we need the set:

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

How does a compiler exploit this information?

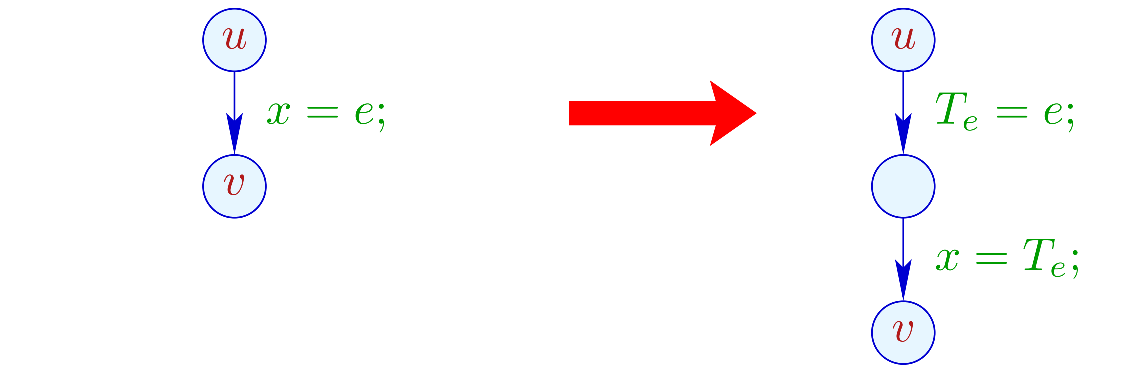
Transformation UT (unique temporaries):

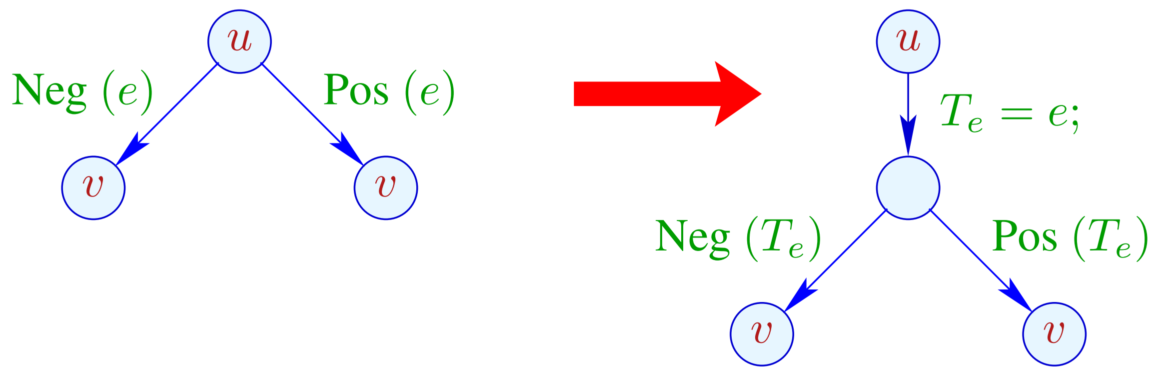
We provide a novel register T_e as storage for the values of e:



Transformation UT (unique temporaries):

We provide novel registers T_e as storage for the value of e:

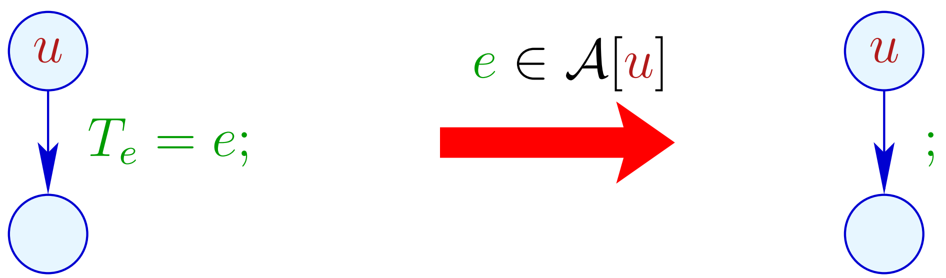




... analogously for $R = M[e];$ and $M[e_1] = e_2;$.

Transformation AEE (available expression elimination):

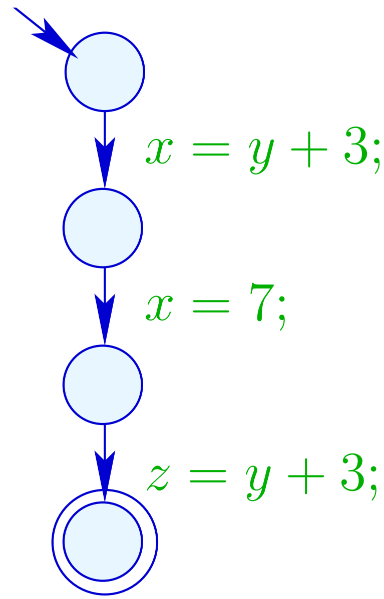
If e is available at program point u, then e need not be re-evaluated:



We replace the assignment with Nop.

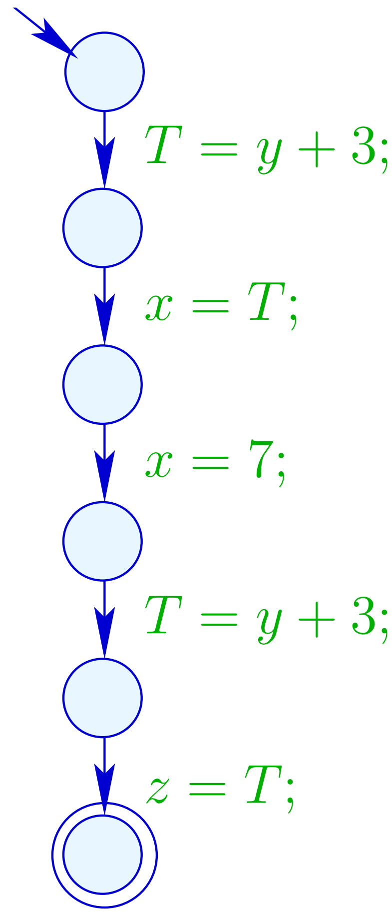
Example:

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



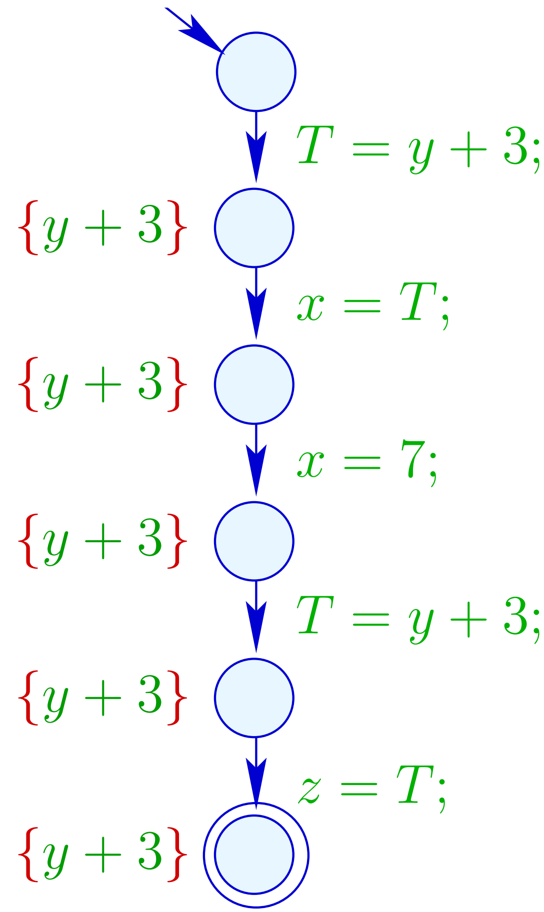
Example:

$$x = y + 3;$$
$$x = 7;$$
$$z = y + 3;$$



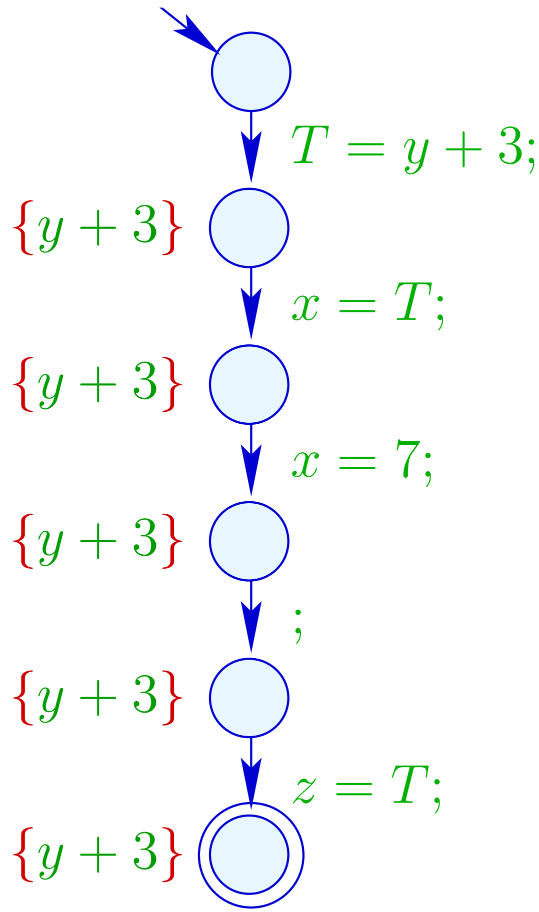
Example:

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



Example:

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



Warning:

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

- $\rightarrow$ $x \notin Vars(e)$; why?
- $\rightarrow$ $e \notin Vars$; why?
- $\rightarrow$ the evaluation of e is non-trivial; why?

Warning:

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

- $\rightarrow$ $x \notin Vars(e)$; otherwise e is not available afterwards.
- $\rightarrow$ $e \notin Vars$; otherwise values are shuffled around
- $\rightarrow$ the evaluation of e is non-trivial; otherwise the efficiency of the code is decreased.

Open 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 constraints on the values of $\mathcal{A}[u]$ into a system of constraints:

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

Why $\subseteq$?

Question:

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

Idea:

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

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

Why $\subseteq$?

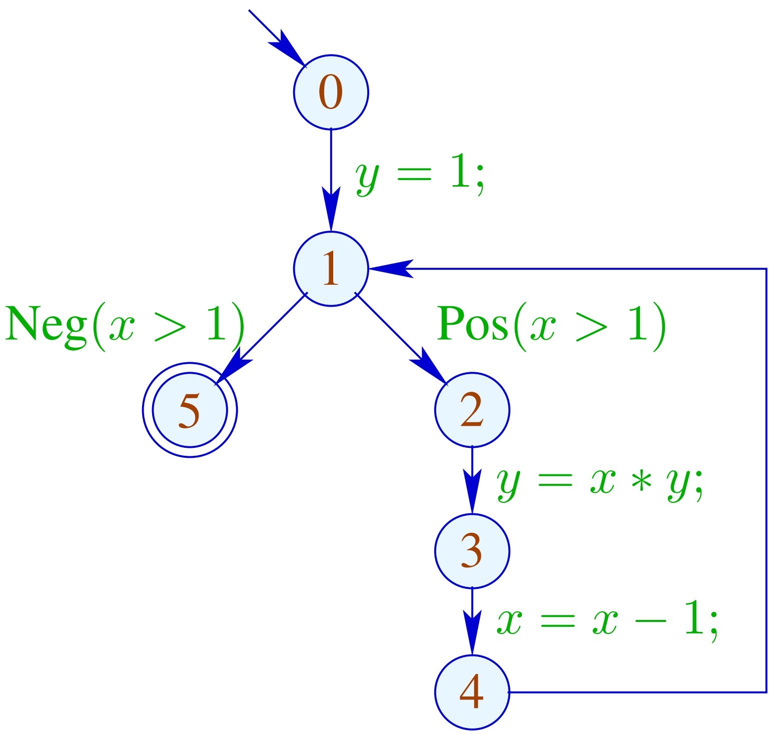
Then combine all constraints for each variable v by applying the least-upper-bound operator $\longrightarrow$

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

Wanted:

- a greatest solution (why greatest?)
- an algorithm that computes this solution

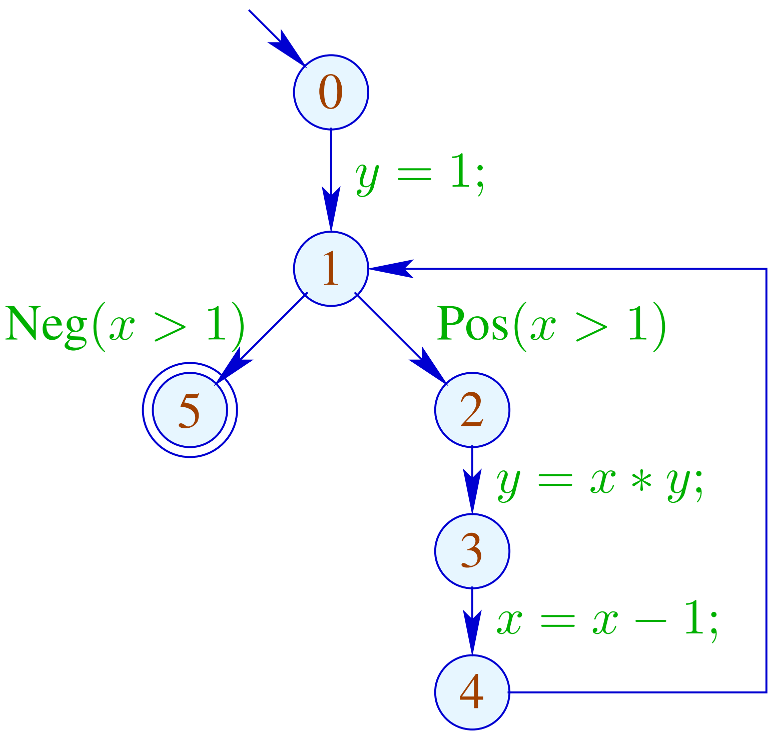
Example:



Wanted:

- a greatest solution (why greatest?)
- an algorithm that computes this solution

Example:

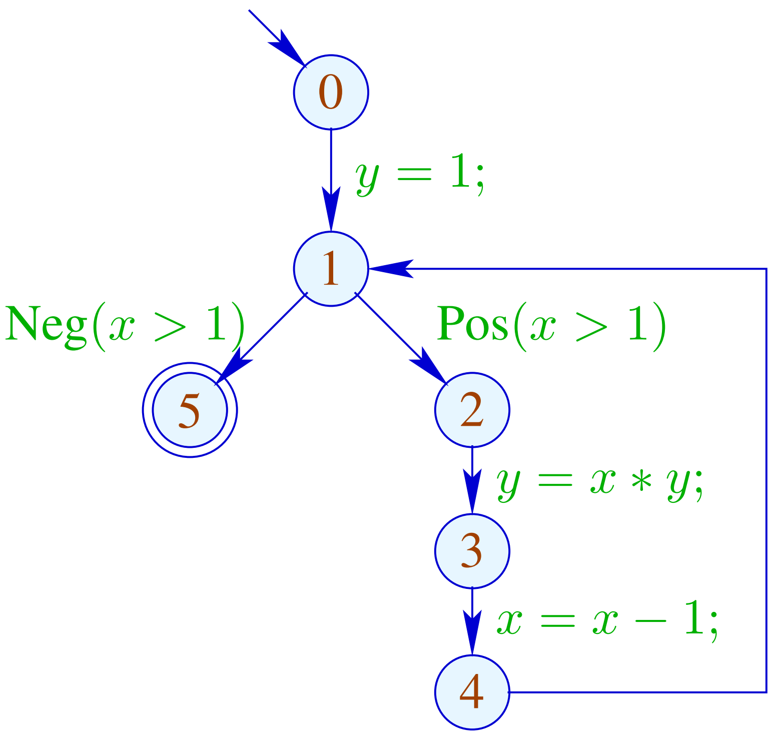


$$\mathcal{A}[0] \quad \subseteq \quad \emptyset$$

Wanted:

- a greatest solution (why greatest?)
- an algorithm that computes this solution

Example:

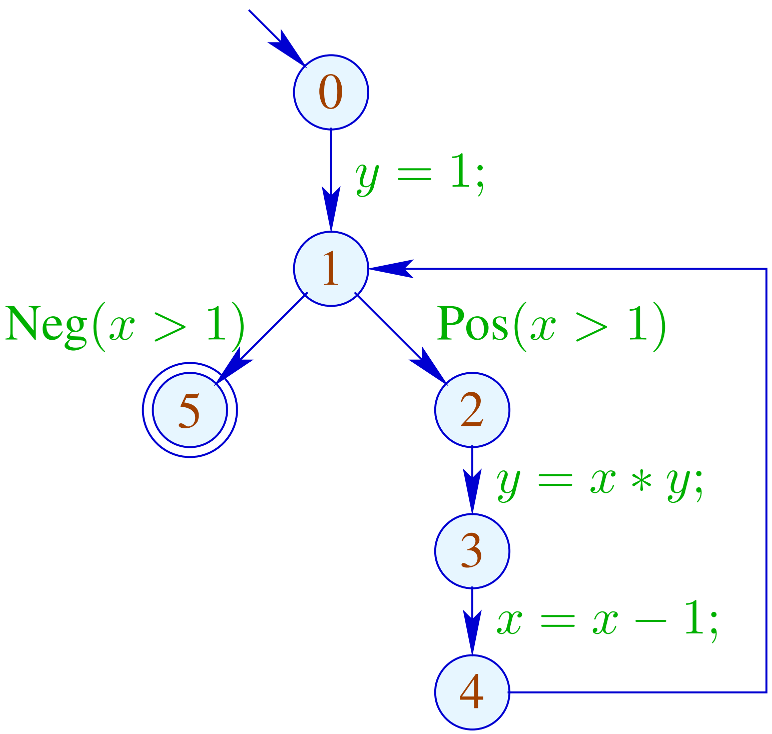


$$\begin{array}{lll} \mathcal{A}[0] & \subseteq & \emptyset \\ \mathcal{A}[1] & \subseteq & (\mathcal{A}[0] \cup \{1\}) \backslash Expr_y \\ \mathcal{A}[1] & \subseteq & \mathcal{A}[4] \end{array}$$

Wanted:

- a greatest solution (why greatest?)
- an algorithm that computes this solution

Example:

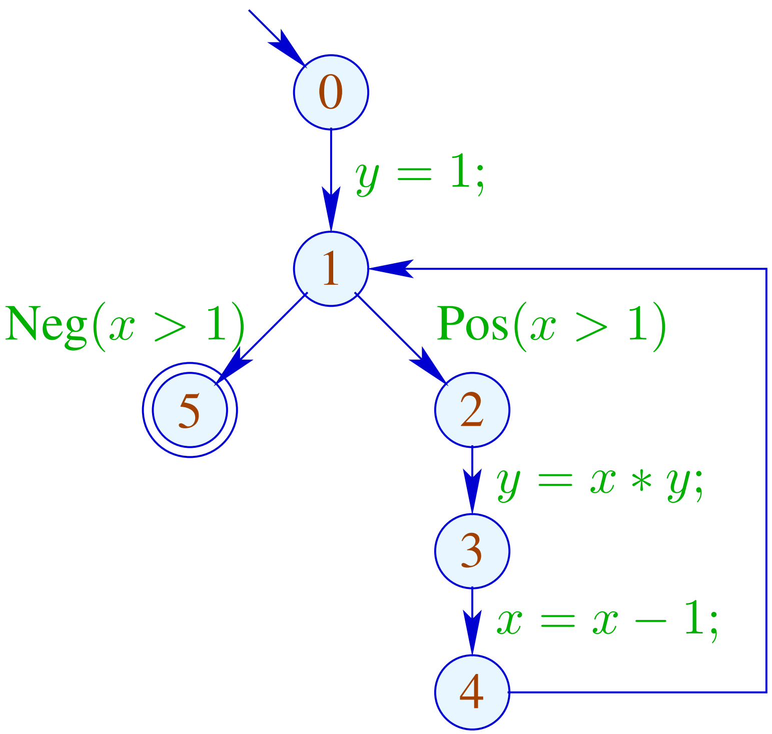


$$\begin{array}{lcl}
\mathcal{A}[0] & \subseteq & \emptyset \\
\mathcal{A}[1] & \subseteq & (\mathcal{A}[0] \cup \{1\}) \backslash Expr_y \\
\mathcal{A}[1] & \subseteq & \mathcal{A}[4] \\
\mathcal{A}[2] & \subseteq & \mathcal{A}[1] \cup \{x > 1\}
\end{array}$$

Wanted:

- a greatest solution (why greatest?)
- an algorithm that computes this solution

Example:

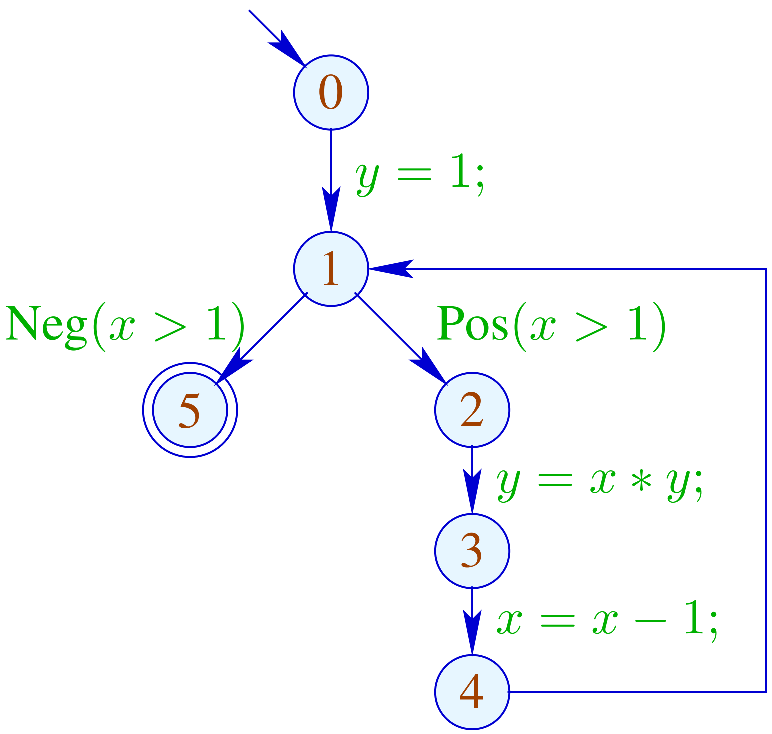


$$
\begin{array}{lcl}
\mathcal{A}[0] & \subseteq & \emptyset \\
\mathcal{A}[1] & \subseteq & (\mathcal{A}[0] \cup \{1\}) \backslash Expr_y \\
\mathcal{A}[1] & \subseteq & \mathcal{A}[4] \\
\mathcal{A}[2] & \subseteq & \mathcal{A}[1] \cup \{x > 1\} \\
\mathcal{A}[3] & \subseteq & (\mathcal{A}[2] \cup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

Wanted:

- a greatest solution (why greatest?)
- an algorithm that computes this solution

Example:

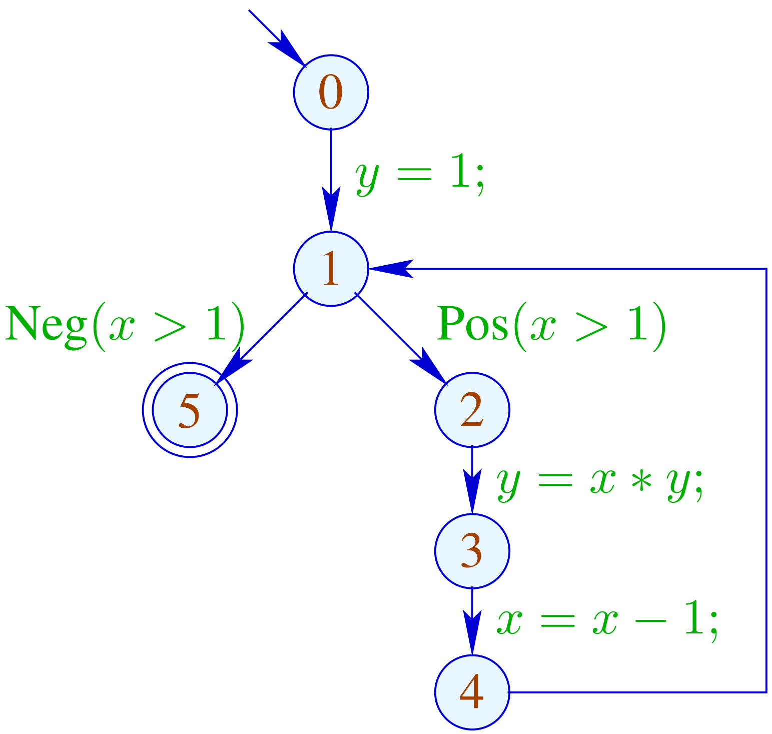


$$
\begin{array}{lcl}
\mathcal{A}[0] & \subseteq & \emptyset \\
\mathcal{A}[1] & \subseteq & (\mathcal{A}[0] \cup \{1\}) \backslash Expr_y \\
\mathcal{A}[1] & \subseteq & \mathcal{A}[4] \\
\mathcal{A}[2] & \subseteq & \mathcal{A}[1] \cup \{x > 1\} \\
\mathcal{A}[3] & \subseteq & (\mathcal{A}[2] \cup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

Wanted:

- a greatest solution (why greatest?)
- an algorithm that computes this solution

Example:

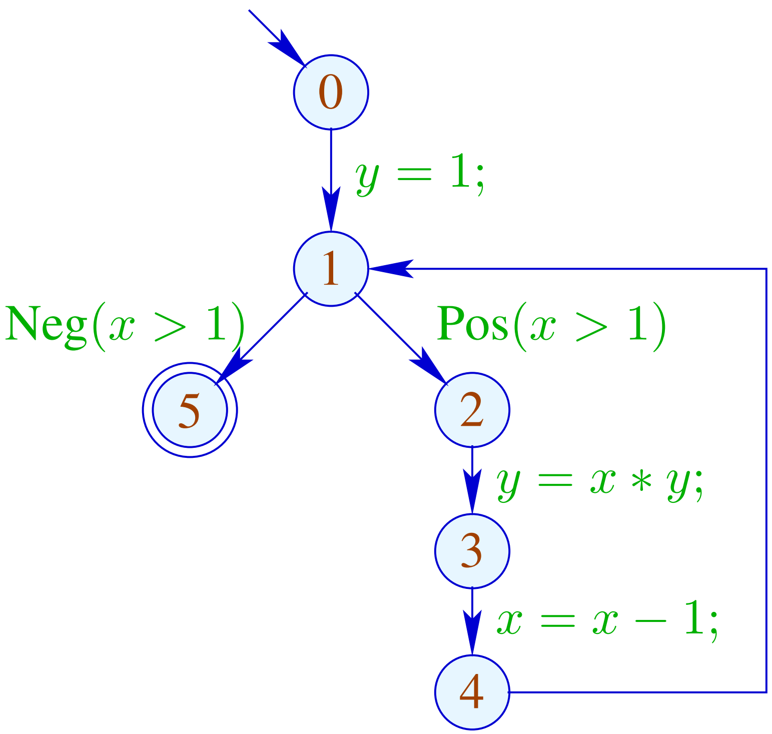


$$
\begin{array}{lll}
\mathcal{A}[0] & \subseteq & \emptyset \\
\mathcal{A}[1] & \subseteq & (\mathcal{A}[0] \cup \{1\}) \backslash Expr_y \\
\mathcal{A}[1] & \subseteq & \mathcal{A}[4] \\
\mathcal{A}[2] & \subseteq & \mathcal{A}[1] \cup \{x > 1\} \\
\mathcal{A}[3] & \subseteq & (\mathcal{A}[2] \cup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

Wanted:

- a greatest solution,
- an algorithm that computes this solution.

Example:



Solution:

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

Observation:

- Again, 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 order on the lattice elements indicates what is better information,
  more available expressions may allow more optimizations

Observation:

- Again, 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 order on the lattice elements indicates what is better information,
  more available expressions may allow more optimizations
- The functions $[\![k]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ have the form $\quad f_i\, x = a_i \cap x \cup b_i$.
  They are called *gen/kill* functions—$\cap$ kills, $\cup$ generates.
- they are monotonic, i.e.,

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

The operations "$\circ$", "$\sqcup$" and "$\sqcap$" can be explicitly defined by:

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

7.2 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 overwritten 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:

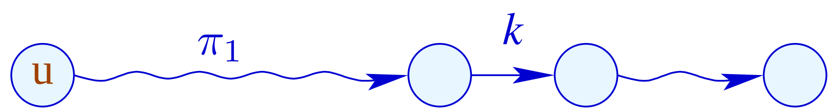
$\rightarrow$ Assignments to dead variables can be removed.

$\rightarrow$ Such inefficiencies may originate from other transformations.

Formal Definition:

The variable x is called live at u along a path π starting at u 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.

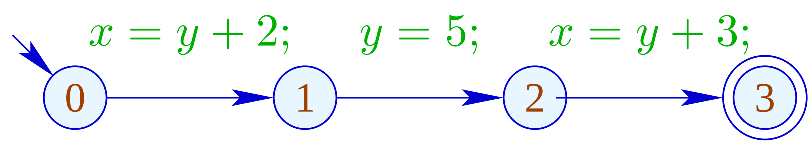


Thereby, the set of all defined or used variables at an edge $k = (\_, lab, \_)$ is defined by

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

A variable x which is not live at u along π is called dead at u along π.

Example:



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 if x is live at u along some path to the exit . Otherwise, x is called dead at u .

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

Question:

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

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

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 that are live at v into the set of variables that are live at u.

Note: Edge transformers go "backwards"!

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

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

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

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

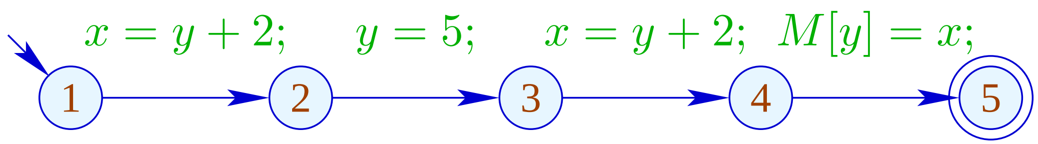
For $k = (\_, lab, \_)$, define $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ by:

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

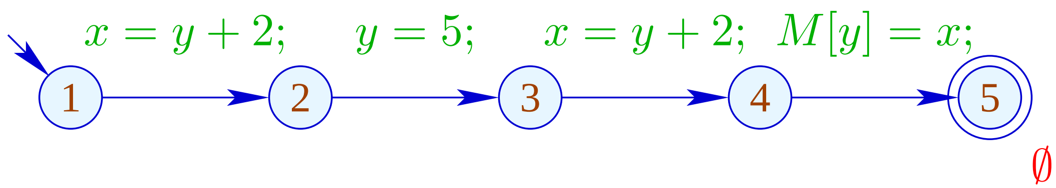
$[\![k]\!]^\sharp$ can again be composed to the effects of $[\![\pi]\!]^\sharp$ of paths $\pi = k_1 \ldots k_r$ by:

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

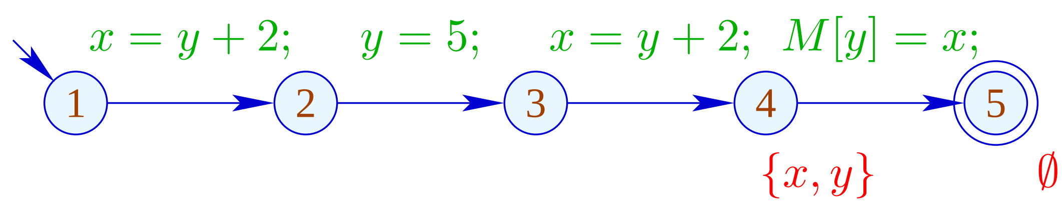
We verify that these definitions are meaningful



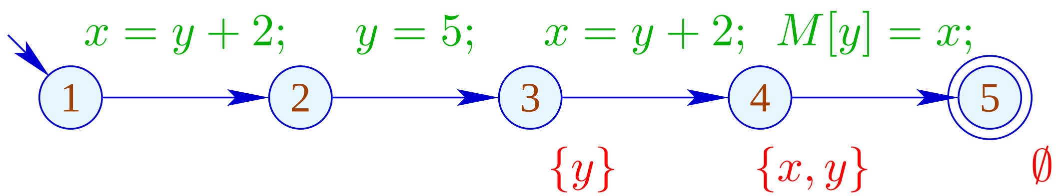
We verify that these definitions are meaningful



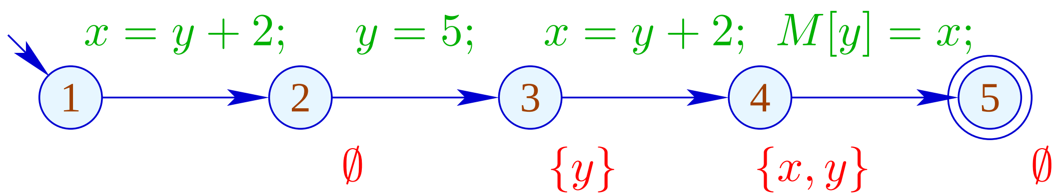
We verify that these definitions are meaningful



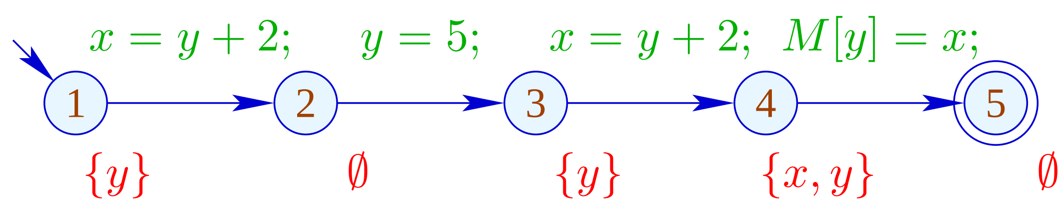
We verify that these definitions are meaningful



We verify that these definitions are meaningful



We verify that these definitions are meaningful



A variable is live at a program point u if there is at least one path from u to program exit on which it is live.

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

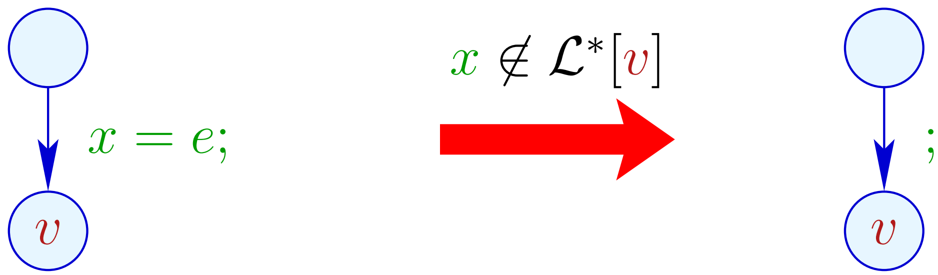
$$\mathcal{L}^*[u] \;=\; \bigcup\{[\![\pi]\!]^\sharp\emptyset \mid \pi : u \rightarrow^* \mathit{stop}\}$$

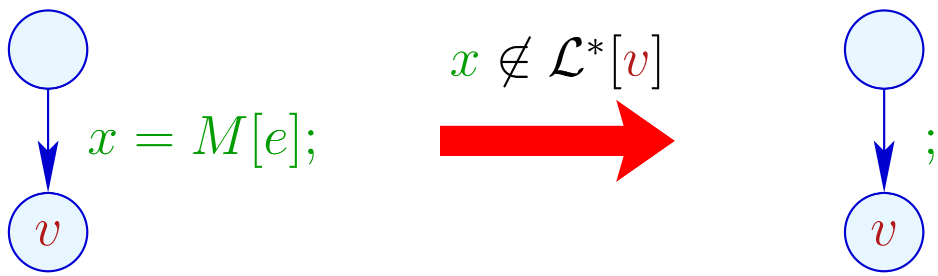
No variables are assumed to be live at program exit.

As partial order for $\mathbb{L}$ we use $\sqsubseteq \, = \, \subseteq$. why?

So, the least upper bound is $\bigcup$. why?

Transformation DE (Dead assignment Elimination):





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:

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

(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 least 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:

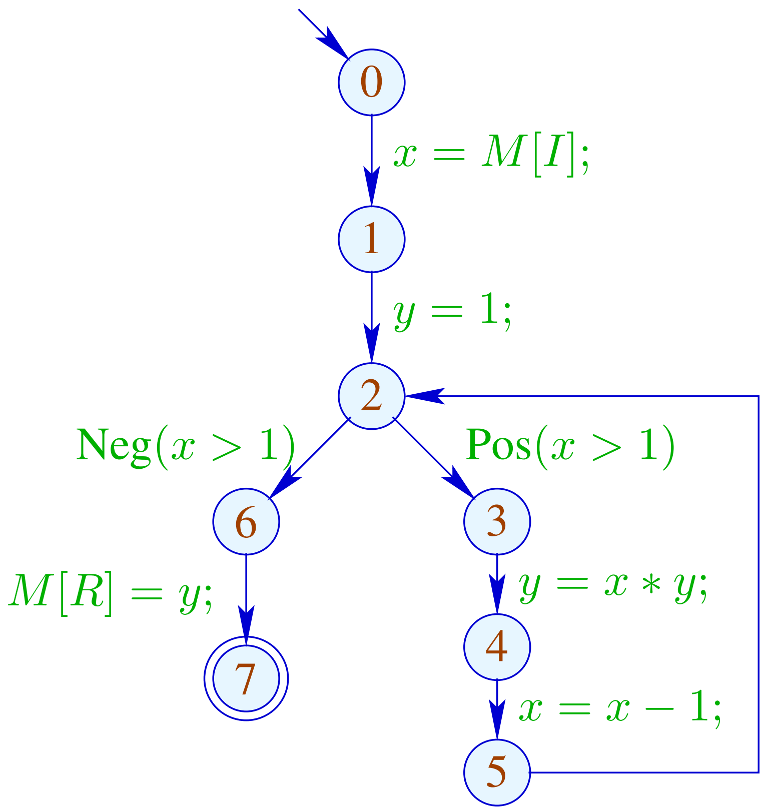
$$\begin{array}{lll} \mathcal{L}[stop] & \supseteq & \emptyset \\ \mathcal{L}[u] & \supseteq & [\![k]\!]^\sharp\,(\mathcal{L}[v]) \qquad k = (u, \_, v) \quad \text{edge} \end{array}$$

(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 least solution $\mathcal{L}$ of the constraint system equals $\mathcal{L}^*$ since all $[\![k]\!]^\sharp$ are distributive.

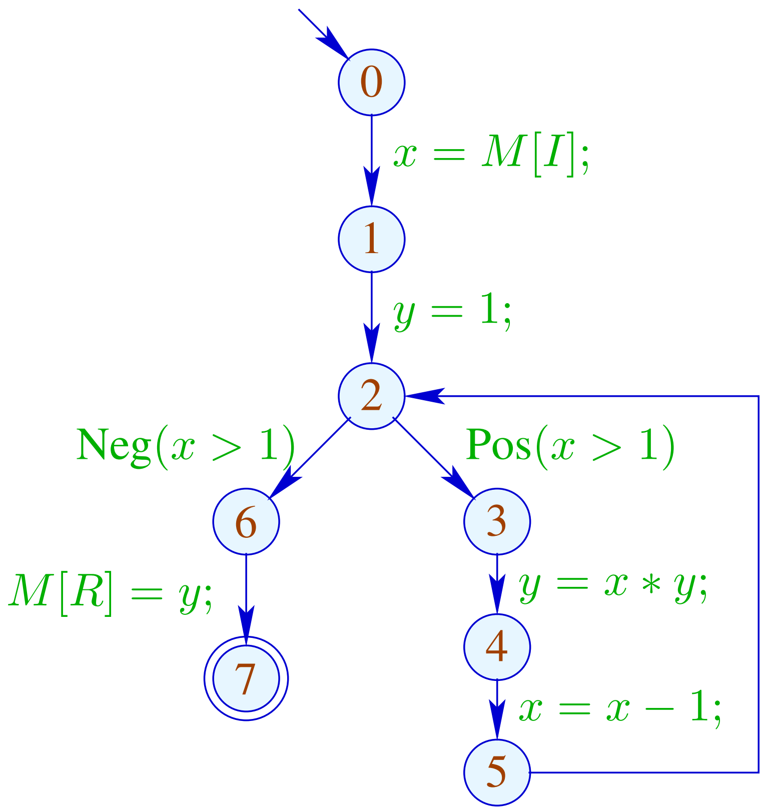
Note: The information is propagated backwards!

Example:



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

Example:

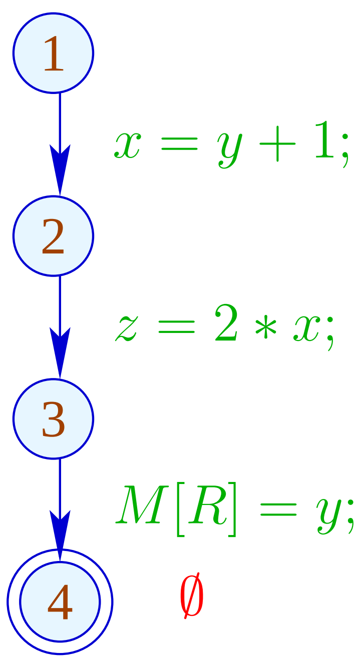


	1	2
7	$\emptyset$	
6	$\{y, R\}$	
2	$\{x, y, R\}$	dito
5	$\{x, y, R\}$	
4	$\{x, y, R\}$	
3	$\{x, y, R\}$	
1	$\{x, R\}$	
0	$\{I, R\}$	

The left-hand side of no assignment is dead

Caveat:

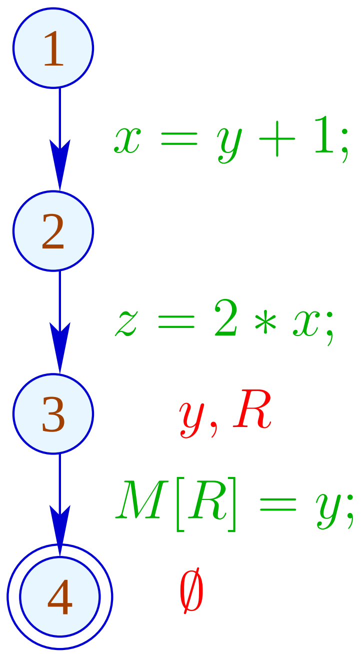
Removal of assignments to dead variables may kill further variables:



The left-hand side of no assignment is dead

Caveat:

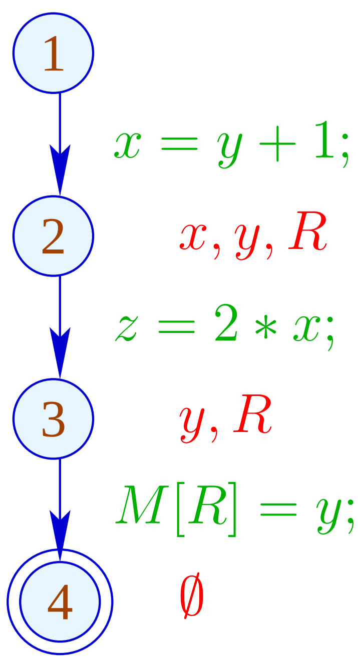
Removal of assignments to dead variables may kill further variables:



The left-hand side of no assignment is dead

Caveat:

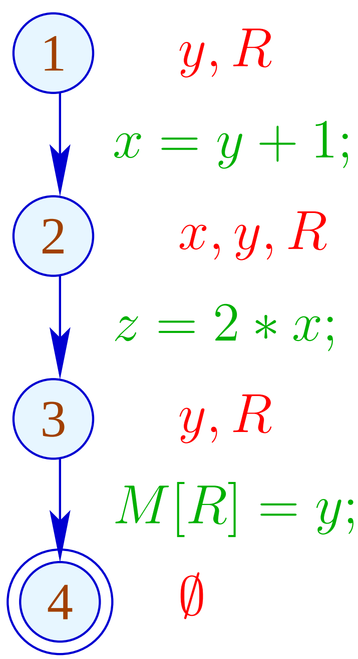
Removal of assignments to dead variables may kill further variables:



The left-hand side of no assignment is dead

Caveat:

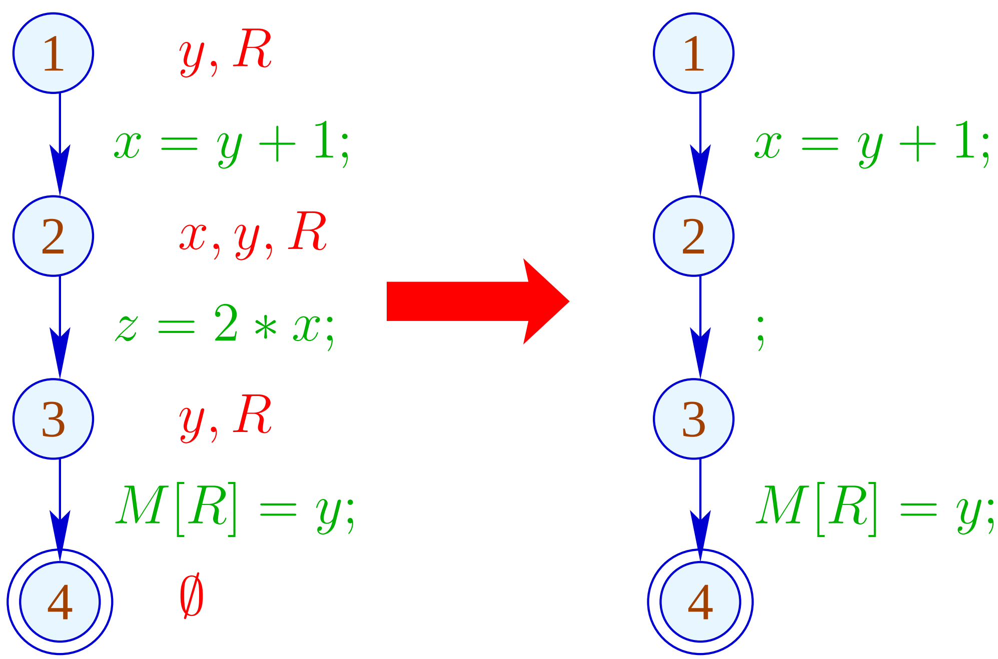
Removal of assignments to dead variables may kill further variables:



The left-hand side of no assignment is dead

Caveat:

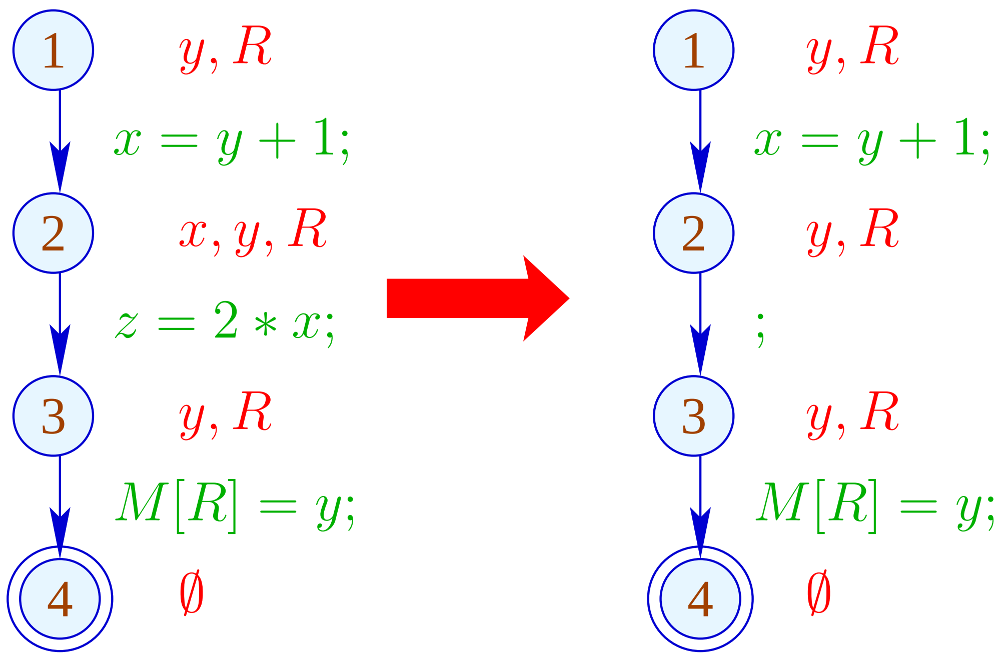
Removal of assignments to dead variables may kill further variables:



The left-hand side of no assignment is dead

Caveat:

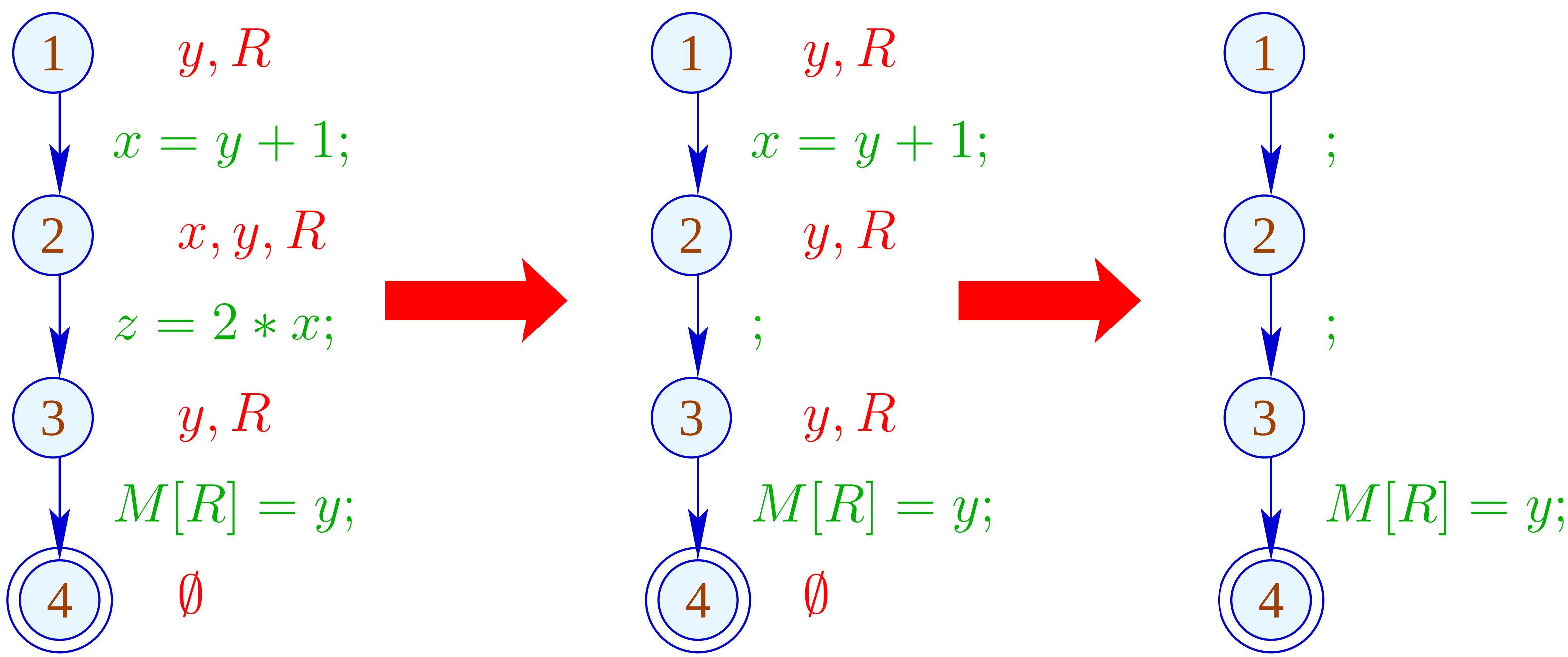
Removal of assignments to dead variables may kill further variables:



The left-hand side of no assignment is dead

Caveat:

Removal of assignments to dead variables may kill further variables:



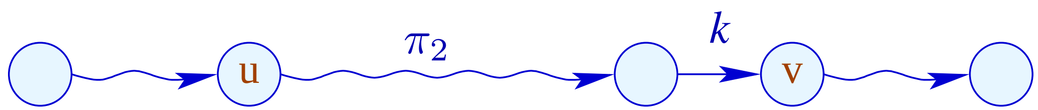
Re-analyzing the program is inconvenient

Idea: Analyze **true** liveness!

x is called **truly live** at u along a path π, either

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.

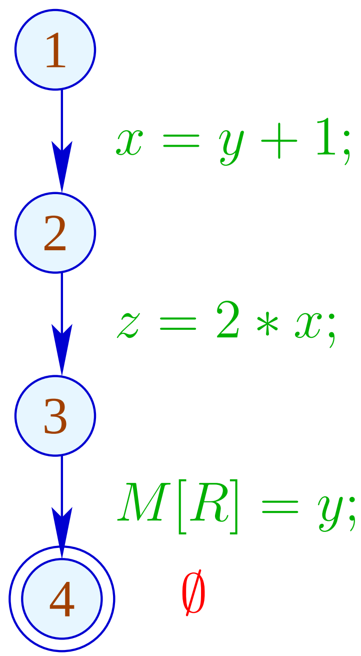


The set of truly used variables at an edge $k = (\_, lab, v)$ is defined as:

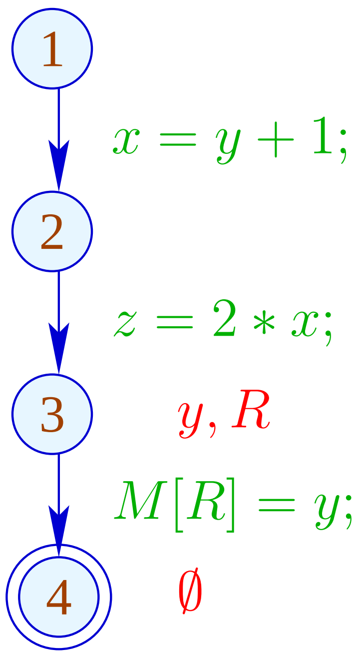
lab	truly used
$;$	$\emptyset$
$\text{true}\,(e)$	$Vars\,(e)$
$\text{false}\,(e)$	$Vars\,(e)$
$x = e;$	$Vars\,(e)$ $(*)$
$x = M[e];$	$Vars\,(e)$ $(*)$
$M[e_1] = e_2;$	$Vars(e_1) \cup Vars(e_2)$

$(*)$ – given that x is truly live at v

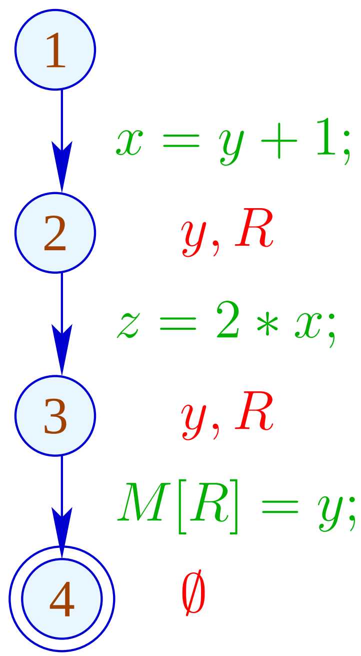
Example:



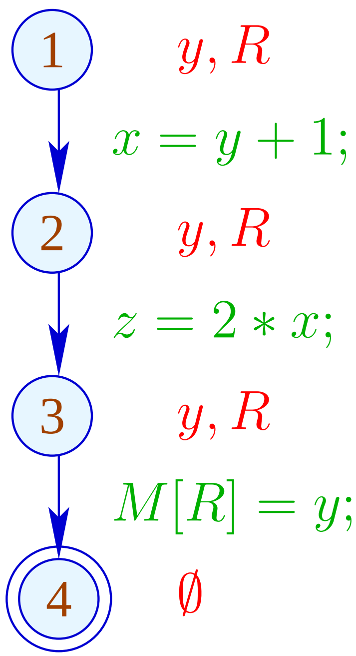
Example:



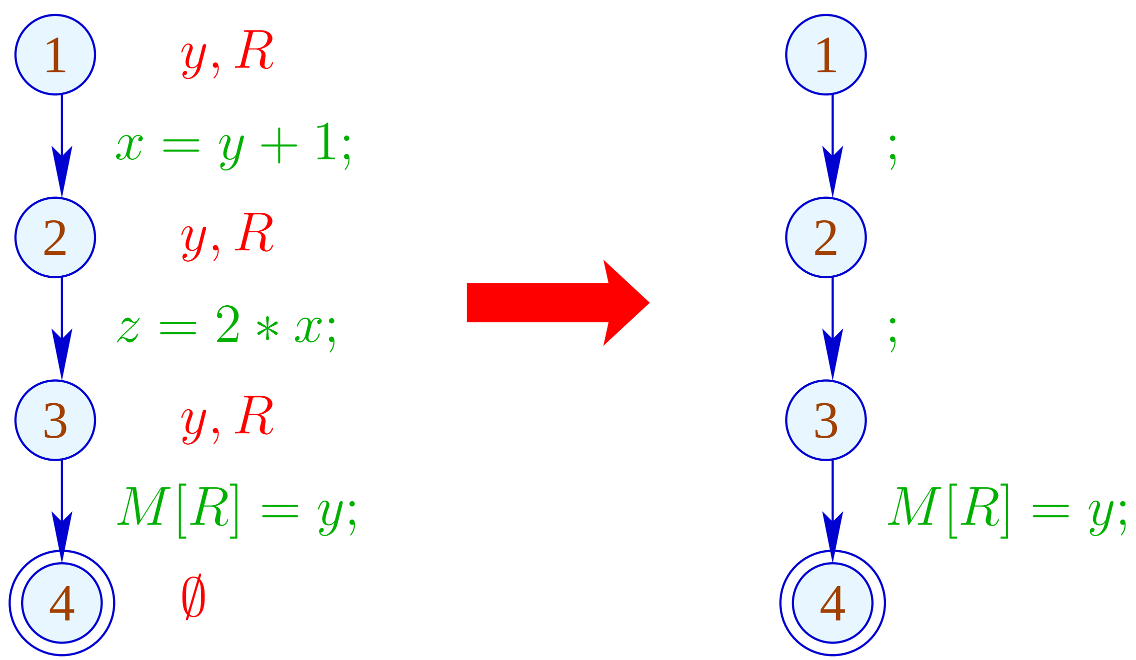
Example:



Example:



Example:



The Effects of Edges:

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

The Effects of Edges:

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

Note:

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

Note:

- The effects of edges for truly 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:

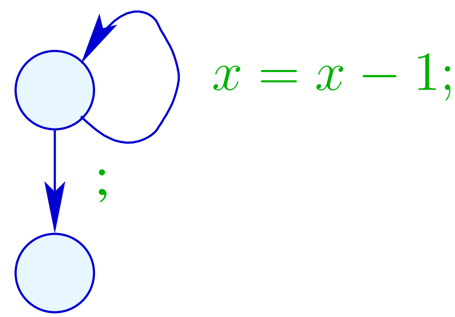
- The effects of edges for truly live variables are more complicated than for live variables
- Nonetheless, they are distributive !!

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

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

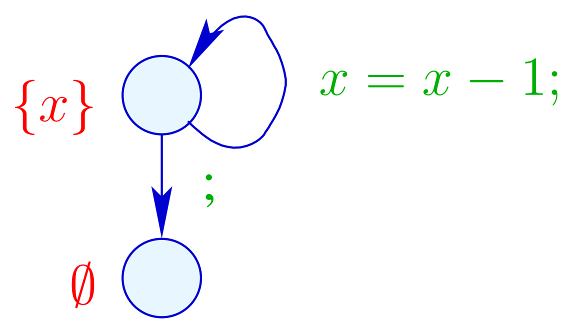
$\Longrightarrow$ the constraint system yields the MOP

- 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: