Reinhard Wilhelm

Universität des Saarlandes

Development of Safety-Critical Embedded Systems
Static Program Analysis

Winter Semester 2012/2013

Slides based on:

- R. Wilhelm, B. Wachter: Abstract Interpretation with Applications to Timing Validation. CAV 2008: 22-36
- Helmut Seidl’s slides
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 (30s), 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. Further static analyses in compilers: Elimination of superfluous computations
   → available expressions
   → live variables
   → array-bounds checks
7. timing (WCET) analysis
8. analysis for runtime errors
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

An intermediate language into which (almost) everything can be translated.
In particular, no procedures. So, only \textit{intra-procedural analyses}!
2 Example — Rules-of-Sign Analysis

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

Example program:

1: x = 0;
2: y = 1;
3: while (y > 0) do
4: y = y + x;
5: x = x + (-1);
Program representation as *control-flow graphs*

\[
\begin{align*}
0: & \quad x = 0 \\
1: & \quad y = 1 \\
2: & \quad \text{true}(y > 0) \quad \text{false}(y > 0) \\
4: & \quad y = y + x \\
5: & \quad x = x + (-1) \\
3: & \quad \text{true}(y > 0) \quad \text{false}(y > 0)
\end{align*}
\]
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, “⊑”, 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 $\text{Signs} = 2^{\{-,0,+,\}}$ with the relation "\(\subseteq\)" ="\(\supseteq\)".
The analysis should ”bind” program variables to elements in \( Signs \).
So, the abstract domain is \( D = (Vars \rightarrow Signs)_\bot \), a Sign-environment.
\( \bot \in D \) is the function mapping all arguments to \( \{\} \).
The partial order on \( D \) is \( D_1 \sqsubseteq D_2 \) iff 
\[
D_1 = \bot \quad \text{or} \\
D_1 x \supseteq D_2 x \quad (x \in Vars)
\]
Intuition?
The analysis should ”bind” program variables to elements in \( \text{Signs} \).

So, the abstract domain is \( \mathbb{D} = (\text{Vars} \rightarrow \text{Signs})\bot \). a Sign-environment.

\( \bot \in \mathbb{D} \) is the function mapping all arguments to \( \{\} \).

The partial order on \( \mathbb{D} \) is \( D_1 \sqsubseteq D_2 \) iff

\[
D_1 = \bot \quad \text{or} \quad D_1 x \supseteq D_2 x \quad (x \in \text{Vars})
\]

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?

\[
\begin{align*}
0 & : x = 0 \\
1 & : y = 1 \\
2 & : \text{true}(y>0) \\
4 & : \text{false}(y>0) \\
5 & : y = y+x \\
3 & : x = x+(-1)
\end{align*}
\]
How is a solution found?

Iterating until a fixed-point is reached

How is a solution found?

Iterating until a fixed-point is reached
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 □ working on values by abstract operators □♯ 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 \[\square\] working on values by abstract operators \[\square^\#\] working on signs:
• The abstract operators allow to define an abstract evaluation of expressions:

\[
[e]^\# : (\text{Vars} \rightarrow \text{Signs}) \rightarrow \text{Signs}
\]
Determining the sign of expressions in a Sign-environment works as follows:

\[
\begin{align*}
[c] \# D &= \begin{cases} 
+ & \text{if } c > 0 \\
- & \text{if } c < 0 \\
0 & \text{if } c = 0
\end{cases} \\
[v] \# &= D(v) \\
[e_1 \Box e_2] \# D &= [e_1] \# D \Box [e_2] \# D \\
[\square e] \# D &= \square [e] \# D
\end{align*}
\]
Abstract operators working on signs (Addition)

<table>
<thead>
<tr>
<th></th>
<th>{0}</th>
<th>{+}</th>
<th>{-}</th>
<th>{-, 0}</th>
<th>{-, +}</th>
<th>{0, +}</th>
<th>{-, 0, +}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0}</td>
<td>{0}</td>
<td>{+}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{+}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{-}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{-, 0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{-, +}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{0, +}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{-, 0, +}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Abstract operators working on signs (Multiplication)

<table>
<thead>
<tr>
<th>$\times#$</th>
<th>${0}$</th>
<th>${+}$</th>
<th>${-}$</th>
<th>${-, 0}$</th>
<th>${-, +}$</th>
<th>${0, +}$</th>
<th>${-, 0, +}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${0}$</td>
<td>${0}$</td>
<td>${0}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>${+}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>${-}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>${-, 0}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>${-, +}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>${0, +}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>${-, 0, +}$</td>
<td>${0}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Abstract operators working on signs (unary minus)

<table>
<thead>
<tr>
<th>$-#$</th>
<th>${0}$</th>
<th>${+}$</th>
<th>${-}$</th>
<th>${-, 0}$</th>
<th>${-, +}$</th>
<th>${0, +}$</th>
<th>${-, 0, +}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${0}$</td>
<td>${0}$</td>
<td>${-}$</td>
<td>${+, 0}$</td>
<td>${-, +}$</td>
<td>${0, -}$</td>
<td>${-, 0, +}$</td>
<td></td>
</tr>
</tbody>
</table>
Working an example:

\[ D = \{ x \mapsto \{+\}, y \mapsto \{+\} \} \]

\[
[x + 7]\# D = [x]\# D +\# [7]\# D \\
= \{+\} +\# \{+\} \\
= \{+\}
\]

\[
[x + (-y)]\# D = \{+\} +\# (-\#[y]\# D) \\
= \{+\} +\# (-\#\{+\}) \\
= \{+\} +\# \{-\} \\
= \{+,-,0\}
\]
$\text{[lab]}^\#$ is the abstract edge effects associated with edge $k$.

It depends only on the label $\text{lab}$:

\[
\begin{align*}
[;]^\#D &= D \\
[\text{true (e)}]^\#D &= D \\
[\text{false (e)}]^\#D &= D \\
[x = e;]^\#D &= D \oplus \{x \mapsto [e]^\#D\} \\
[x = M[e];]^\#D &= D \oplus \{x \mapsto \{+, -, 0\}\} \\
[M[e_1] = e_2;]^\#D &= D
\end{align*}
\]

... whenever $D \neq \perp$

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

\[
[\pi]^\# = [k_r]^\# \circ \ldots \circ [k_1]^\#
\]
Consider a program node $v$:

$\rightarrow$ For every path $\pi$ 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 $\pi$.

$\rightarrow$ Initially at program start, no information about signs is available.

$\rightarrow$ The analysis computes a superset of the set of signs as safe information.

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

$$S[v] = \bigcup\{[\pi]^* \downarrow \mid \pi : start \rightarrow^* v\}$$
Question:

How do we compute $S[u]$ for every program point $u$?
Question:

How can we compute $S[u]$ for every program point $u$?

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

$$S[start] \supseteq \bot$$

$$S[v] \supseteq \lfloor k \rfloor^\sharp (S[u])$$

$k = (u, _, v)$ edge

Why $\supseteq$?
Wanted:

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

Example:
\[
\begin{align*}
S[0] & \supseteq \bot \\
S[1] & \supseteq S[0] \oplus \{x \mapsto \{0\}\} \\
S[2] & \supseteq S[1] \oplus \{y \mapsto \{+\}\} \\
S[2] & \supseteq S[5] \oplus \{x \mapsto [x + (-1)] \# S[5]\} \\
\end{align*}
\]
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;
    }
}

A_1 = A_0 + 1 \times i;
R_1 = M[A_1];
A_2 = A_0 + 1 \times j;
R_2 = M[A_2];

\text{Pos} (R_1 > R_2)
\text{Neg} (R_1 > R_2)

A_3 = A_0 + 1 \times j;

....
Thereby, represent:

<table>
<thead>
<tr>
<th>vertex</th>
<th>program point</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>program start</td>
</tr>
<tr>
<td>stop</td>
<td>program exit</td>
</tr>
<tr>
<td>edge</td>
<td>labeled with a statement or a condition</td>
</tr>
</tbody>
</table>
Thereby, represent:

<table>
<thead>
<tr>
<th>vertex</th>
<th>program point</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>program start</td>
</tr>
<tr>
<td>stop</td>
<td>program exit</td>
</tr>
<tr>
<td>edge</td>
<td>step of computation</td>
</tr>
</tbody>
</table>

**Edge Labelings:**

- **Test**: Pos ($e$) or Neg ($e$)
- **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:

| \( \rho : \text{Vars} \rightarrow \text{int} \) | values of variables (contents of symbolic registers) |
| \( \mu : \mathbb{N} \rightarrow \text{int} \) | contents of memory |

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

\[
[k] = [lab]
\]

of the state:
\[
[;] (\rho, \mu) = (\rho, \mu)
\]

\[
[\text{Pos } (e)] (\rho, \mu) = (\rho, \mu) \quad \text{if } \lbrack e \rbrack \rho \neq 0
\]

\[
[\text{Neg } (e)] (\rho, \mu) = (\rho, \mu) \quad \text{if } \lbrack e \rbrack \rho = 0
\]
\[ (;) (\rho, \mu) = (\rho, \mu) \]

\[ \text{Pos} (e) \] \[ (\rho, \mu) = (\rho, \mu) \quad \text{if} \quad [e] \rho \neq 0 \]
\[ \text{Neg} (e) \] \[ (\rho, \mu) = (\rho, \mu) \quad \text{if} \quad [e] \rho = 0 \]

// \[ e \] : evaluation of the expression \( e \), e.g.

// \[ [x + y]\{x \mapsto 7, y \mapsto -1\} = 6 \]
// \[ ![x == 4]\{x \mapsto 5\} = 1 \]
\[;\] (\(\rho, \mu\)) = (\(\rho, \mu\))

\[[\text{Pos} (e)]\] (\(\rho, \mu\)) = (\(\rho, \mu\)) \quad \text{if} \ [e] \rho \neq 0

\[[\text{Neg} (e)]\] (\(\rho, \mu\)) = (\(\rho, \mu\)) \quad \text{if} \ [e] \rho = 0

// \[[e]\] : \ evaluation \ of \ the \ expression \ e, \ \text{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
$$\begin{align*}
[R = M[e];] (\rho, \mu) &= \left( \rho \oplus \{ R \mapsto \mu([e] \rho) \}, \mu \right) \\
[M[e_1] = e_2;] (\rho, \mu) &= (\rho, \mu \oplus \{ [e_1] \rho \mapsto [e_2] \rho \})
\end{align*}$$

Example:
$$[x = x + 1;] (\{ x \mapsto 5 \}, \mu) = (\rho, \mu) \quad \text{where}$$
\begin{align*}
\rho &= \{ x \mapsto 5 \} \oplus \{ x \mapsto [x + 1] \{ x \mapsto 5 \} \} \\
&= \{ x \mapsto 5 \} \oplus \{ x \mapsto 6 \} \\
&= \{ x \mapsto 6 \}
\end{align*}

A path \( \pi = k_1 k_2 \ldots k_m \) defines a computation in the state \( s \) if
\[
\begin{align*}
s &\in \text{def} (\llbracket k_m \rrbracket \circ \ldots \circ \llbracket k_1 \rrbracket)
\end{align*}
\]
The result of the computation is
\[
\llbracket \pi \rrbracket s = (\llbracket k_m \rrbracket \circ \ldots \circ \llbracket k_1 \rrbracket) s
\]
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 \( a, b, c \in \mathbb{D} \),

\[
\begin{align*}
& a \sqsubseteq a & \text{reflexivity} \\
& a \sqsubseteq b \land b \sqsubseteq a \implies a = b & \text{anti−symmetry} \\
& a \sqsubseteq b \land b \sqsubseteq c \implies a \sqsubseteq c & \text{transitivity}
\end{align*}
\]

**Intuition:** \( \sqsubseteq \) represents **precision**.

By convention: \( a \sqsubseteq b \) means \( a \) is **at least as precise as** \( b \).
Examples:

1. $\mathbb{D} = 2^{\{a,b,c\}}$ with the relation “$\subseteq$”:
2. The rules-of-sign analysis uses the following lattice \( \mathbb{D} = 2^{-,0,+} \) with the relation "\( \subseteq \)":
3. $\mathbb{Z}$ with the relation “$\leq$”:

4. $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\bot\}$ with the ordering:
$d \in \mathbb{D}$ is called **upper bound** for $X \subseteq \mathbb{D}$ if

$$x \sqsubseteq d \quad \text{for all } x \in X$$
$d \in \mathbb{D}$ is called upper bound for $X \subseteq \mathbb{D}$ if

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

$d$ 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

\[ x \sqsubseteq d \quad \text{for all } x \in X \]

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 $\bigcup X \in \mathbb{D}$.

**Note:**

Every complete lattice has

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

→ a greatest element $\top = \bigcup \mathbb{D} \in \mathbb{D}$. 
Examples:

1. $\mathcal{D} = 2\{a, b, c\}$ is a complete lattice.

2. $\mathcal{D} = \mathbb{Z}$ with “$\leq$” is not a complete lattice.

3. $\mathcal{D} = \mathbb{Z}_\perp$ is also not a complete lattice.

4. With an extra element $\top$, we obtain the flat lattice:
   
   $$\mathbb{Z}_\top = \mathbb{Z} \cup \{\bot, \top\}$$

\[\text{Diagram}\]
Theorem:

If $\mathcal{D}$ is a complete lattice, then every subset $X \subseteq \mathcal{D}$ has a greatest lower bound $\bigcap X$. 
Back to the system of constraints for Rules-of-Signs Analysis!

\[
S[start] \supseteq \top
\]
\[
S[v] \supseteq [k]^{\#}(S[u]) \quad k = (u, _, v) \quad \text{edge}
\]

Combine all constraints for each variable \(v\) by applying the least-upper-bound operator \(\sqcup\):

\[
S[v] \supseteq \biguplus\{[k]^{\#}(S[u]) \mid k = (u, _, v) \quad \text{edge}\}
\]

Correct because:

\[
x \supseteq d_1 \land \ldots \land x \supseteq d_k \quad \text{iff} \quad x \supseteq \biguplus\{d_1, \ldots, d_k\}
\]
Our generic form of the systems of constraints:

\[ x_i \supseteq f_i(x_1, \ldots, x_n) \]  

(\text{(*)})

Relation to the running example:

| \(x_i\) | unknown \(S[u]\) |
| \(D\) | values \(Signs\) |
| \(\subseteq \subseteq D \times D\) | ordering relation \(\subseteq\) |
| \(f_i: D^n \rightarrow D\) | constraint \(\ldots\) |
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:

- $\text{inc } x = x + 1$ is monotonic.
- $\text{dec } x = x - 1$ is monotonic.
A mapping $f : \mathbb{D}_1 \to \mathbb{D}_2$ is called monotonic (order preserving) if $f(a) \subseteq f(b)$ for all $a \subseteq b$.

**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:

   - $\text{inc } x = x + 1$ is monotonic.
   - $\text{dec } x = x - 1$ is monotonic.
   - $\text{inv } x = -x$ 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 \( f_1 : D_1 \to D_2 \) and \( f_2 : D_2 \to D_3 \) are monotonic, then also
\( f_2 \circ f_1 : D_1 \to D_3 \).
Theorem:

If $f_1 : \mathbb{D}_1 \rightarrow \mathbb{D}_2$ and $f_2 : \mathbb{D}_2 \rightarrow \mathbb{D}_3$ are monotonic, then also $f_2 \circ f_1 : \mathbb{D}_1 \rightarrow \mathbb{D}_3$
Wanted: least solution for:

\[ x_i \supseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \quad (\ast) \]

where all \( f_i : D^n \rightarrow D \) are monotonic.
Wanted: least solution for:

\[ x_i \equiv f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \]  

(*)

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:

\[ x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \]  

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

Idea:

- Consider \( F : \mathbb{D}^n \to \mathbb{D}^n \) where
  \[ F(x_1, \ldots, x_n) = (y_1, \ldots, y_n) \text{ with } y_i = f_i(x_1, \ldots, x_n). \]
- If all \( f_i \) are monotonic, then also \( F \)
Wanted: least solution for

\[ x_i \supseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \quad (\star) \]

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:

\[ \bot, \quad F \bot, \quad F^2 \bot, \quad F^3 \bot, \quad \ldots \]

Intuition: This iteration eliminates unjustified assumptions.

Hope: We eventually reach a solution!
Example: \[ \mathbb{D} = 2^{\{a,b,c\}}, \quad \subseteq = \subseteq \]

\[ x_1 \supseteq \{a\} \cup x_3 \]
\[ x_2 \supseteq x_3 \cap \{a, b\} \]
\[ x_3 \supseteq x_1 \cup \{c\} \]
Example: \( \mathbb{D} = 2^{\{a,b,c\}} \), \( \subseteq = \subseteq \)

\[
\begin{align*}
x_1 & \supseteq \{a\} \cup x_3 \\
x_2 & \supseteq x_3 \cap \{a, b\} \\
x_3 & \supseteq x_1 \cup \{c\}
\end{align*}
\]

The Iteration:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_1)</td>
<td>(\emptyset)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(x_2)</td>
<td>(\emptyset)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(x_3)</td>
<td>(\emptyset)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: \[ \mathcal{D} = 2\{a,b,c\}, \quad \subseteq = \subseteq \]

\[
\begin{align*}
x_1 & \supseteq \{a\} \cup x_3 \\
x_2 & \supseteq x_3 \cap \{a, b\} \\
x_3 & \supseteq x_1 \cup \{c\}
\end{align*}
\]

The Iteration:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_1)</td>
<td>(\emptyset)</td>
<td>({a})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(x_2)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(x_3)</td>
<td>(\emptyset)</td>
<td>({c})</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: \( \mathcal{D} = 2^{\{a,b,c\}} \), \( \subseteq = \subseteq \)

\[
\begin{align*}
x_1 & \supseteq \{a\} \cup x_3 \\
x_2 & \supseteq x_3 \cap \{a, b\} \\
x_3 & \supseteq x_1 \cup \{c\}
\end{align*}
\]

The Iteration:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_1)</td>
<td>\emptyset</td>
<td>{a}</td>
<td>{a, c}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(x_2)</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(x_3)</td>
<td>\emptyset</td>
<td>{c}</td>
<td>{a, c}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: \[ \mathcal{D} = 2\{a,b,c\}, \quad \subseteq = \subseteq \]

\[ x_1 \supseteq \{a\} \cup x_3 \]
\[ x_2 \supseteq x_3 \cap \{a, b\} \]
\[ x_3 \supseteq x_1 \cup \{c\} \]

The Iteration:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 )</td>
<td>( \emptyset )</td>
<td>{a}</td>
<td>{a, c}</td>
<td>{a, c}</td>
<td>{a, c}</td>
</tr>
<tr>
<td>( x_2 )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>{a}</td>
<td>{a}</td>
</tr>
<tr>
<td>( x_3 )</td>
<td>( \emptyset )</td>
<td>{c}</td>
<td>{a, c}</td>
<td>{a, c}</td>
<td>{a, c}</td>
</tr>
</tbody>
</table>
Example:

\[ \mathcal{D} = 2\{a,b,c\}, \quad \subseteq = \subseteq \]

\[
\begin{align*}
\mathcal{D} &\supseteq \{a\} \cup \mathcal{D} \\
\mathcal{D} &\supseteq \mathcal{D} \cap \{a, b\} \\
\mathcal{D} &\supseteq \mathcal{D} \cup \{c\}
\end{align*}
\]

The Iteration:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_1)</td>
<td>\emptyset</td>
<td>{a}</td>
<td>{a, c}</td>
<td>{a, c}</td>
<td>ditto</td>
</tr>
<tr>
<td>(x_2)</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>{a}</td>
<td></td>
</tr>
<tr>
<td>(x_3)</td>
<td>\emptyset</td>
<td>{c}</td>
<td>{a, c}</td>
<td>{a, c}</td>
<td></td>
</tr>
</tbody>
</table>
Theorem

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

  \[ \bot \subseteq F \bot \subseteq F^2 \bot \subseteq \ldots \]

- If $F^k \bot = F^{k+1} \bot$, $F^k$ is the least solution.

- If all ascending chains are finite, such a $k$ always exists.
Theorem

- ⊥, F ⊥, F₂ ⊥, ... form an ascending chain:
  \[ \bot \subseteq F \subseteq F^2 \subseteq \ldots \]

- If \( F^k \bot = F^{k+1} \bot \), a solution is obtained, which is the least one.

- If all ascending chains are finite, such a \( k \) always exists.

If \( D \) is finite, a solution can be found that is definitely the least solution.

Question: What, if \( D \) 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 $x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n$ (**$\ast$**) is a **system of constraints** where all $f_i : \mathbb{D}^n \to \mathbb{D}$ are monotonic. 

$\implies$ least solution of (**$\ast$**) $\iff$ least fixed point of $F$
Example 1: \( D = 2^U, \quad f(x) = x \cap a \cup b \)
Example 1: \( \mathbb{D} = 2^U, \; \ f x = x \cap a \cup b \)

<table>
<thead>
<tr>
<th></th>
<th>( f )</th>
<th>( f^k \bot )</th>
<th>( f^k \top )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \emptyset )</td>
<td>( U )</td>
<td></td>
</tr>
</tbody>
</table>
Example 1:  \( \mathbb{D} = 2^U, \quad f(x) = x \cap a \cup b \)

<table>
<thead>
<tr>
<th></th>
<th>( f^k \perp )</th>
<th>( f^k \top )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \emptyset )</td>
<td>( U )</td>
</tr>
<tr>
<td>1</td>
<td>( b )</td>
<td>( a \cup b )</td>
</tr>
</tbody>
</table>
Example 1:  \( \mathcal{D} = 2^U, \quad f(x) = x \cap a \cup b \)

<table>
<thead>
<tr>
<th></th>
<th>( f^k \perp )</th>
<th>( f^k \top )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \emptyset )</td>
<td>( U )</td>
</tr>
<tr>
<td>1</td>
<td>( b )</td>
<td>( a \cup b )</td>
</tr>
<tr>
<td>2</td>
<td>( b )</td>
<td>( a \cup b )</td>
</tr>
</tbody>
</table>
Example 1: \( \mathcal{D} = 2^U, \quad f x = x \cap a \cup b \)

<table>
<thead>
<tr>
<th>( f )</th>
<th>( f^k \perp )</th>
<th>( f^k \top )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \emptyset )</td>
<td>( U )</td>
</tr>
<tr>
<td>1</td>
<td>( b )</td>
<td>( a \cup b )</td>
</tr>
<tr>
<td>2</td>
<td>( b )</td>
<td>( a \cup b )</td>
</tr>
</tbody>
</table>
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:

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
```

Table:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>x</td>
<td>y</td>
<td>x</td>
<td>y</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Idea: Round Robin Iteration

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

Example:
\[ y = 1 \]
The code for **Round Robin Iteration** in **Java** looks as follows:

```java
for (i = 1; i ≤ n; i++) xᵢ = ⊥;
do {
    finished = true;
    for (i = 1; i ≤ n; i++) {
        new = fᵢ(x₁, . . . , xₙ);
        if (!xᵢ ⊒ new) {
            finished = false;
            xᵢ = xᵢ ⊔ 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:

\[
\begin{align*}
\mathcal{I}[\text{start}] & \supseteq d_0 \\
\mathcal{I}[v] & \supseteq [k]^{\#}(\mathcal{I}[u]) \quad k = (u, \_ , v) \quad \text{edge}
\end{align*}
\]

where \( d_0 \in \mathbb{D} \) and all \([k]^{\#} : \mathbb{D} \rightarrow \mathbb{D}\) are monotonic ...

**Wanted:**  \textbf{MOP}  (Merge Over all Paths)

\[
\mathcal{I}^*[v] = \bigsqcup \{ [\pi]^{\#} d_0 \mid \pi : \text{start} \rightarrow^* v \}
\]

**Theorem**  \hspace{1cm} Kam, Ullman 1975

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

\[
\mathcal{I}[v] \supseteq \mathcal{I}^*[v] \quad \text{for every} \quad v
\]

In particular: \( \mathcal{I}[v] \supseteq [\pi]^{\#} d_0 \quad \text{for every} \quad \pi : \text{start} \rightarrow^* v \)
**Disappointment:** Are solutions of the constraint system just upper bounds?

**Answer:** In general: yes

Notable exception, all functions $[[k]^{\#}]$ are distributive.

The function $f : D_1 \to D_2$ is called distributive, if $f (\bigcup X) = \bigcup \{f x \mid x \in X\}$ for all $\emptyset \neq X \subseteq D$;

**Remark:** If $f : D_1 \to D_2$ is distributive, then it is also monotonic

**Theorem**

Assume all $v$ are reachable from $start$.

Then: If all effects of edges $[[k]^{\#}]$ 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:

\[
x = 7;
\]

if \((x > 0)\)

\[M[A] = B;\]
Obviously, \( x \) has always the value 7

Thus, the memory access is always executed

Goal:

\[ x = 7; \]

\[ \text{Neg} \ (x > 0) \quad \text{Pos} \ (x > 0) \]

\[ M[A] = B; \]

;
Obviously, $x$ has always the value 7

Thus, the memory access is always executed

**Goal:**

\[
\begin{align*}
1 & \quad x = 7; \\
2 & \quad \text{Neg } (x > 0) \quad \text{Pos } (x > 0) \\
3 & \quad M[A] = B; \\
4 & \quad ; \\
5 & \quad ;
\end{align*}
\]
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 \textit{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:

$$\mathbb{Z}^\top = \mathbb{Z} \cup \{\top\} \quad \text{with} \quad x \sqsubseteq y \quad \text{iff} \quad y = \top \quad \text{or} \quad x = y$$
Caveat: \( \mathbb{Z}^\top \) is not a complete lattice in itself

\[
(2) \quad \mathbb{D} = (\text{Vars} \rightarrow \mathbb{Z}^\top)_\perp = (\text{Vars} \rightarrow \mathbb{Z}^\top) \cup \{\perp\}
\]

// \( \perp \) denotes: “not reachable”

with \( D_1 \sqsubseteq D_2 \) iff \( \perp = D_1 \) or \( D_1 x \sqsubseteq D_2 x \) \( (x \in \text{Vars}) \)

Remark: \( \mathbb{D} \) is a complete lattice
For every edge $k = (_, lab, _)$, construct an effect function $[k]# = [lab]# : \mathbb{D} \to \mathbb{D}$ which simulates the concrete computation.

Obviously, $[lab]# \perp = \perp$ for all $lab$.

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

We must replace the concrete operators $\boxtimes$ by abstract operators $\boxtimes#$ which can handle $\top$:

$$a \boxtimes# b = \begin{cases} 
\top & \text{if } a = \top \text{ or } b = \top \\
\hat{a} \boxtimes b & \text{otherwise}
\end{cases}$$
Idea:

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

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

$$a \square^\# b = \begin{cases} \top & \text{if } a = \top \text{ or } b = \top \\ a \square b & \text{otherwise} \end{cases}$$

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

$$[e]^\# : (Vars \to \mathbb{Z}^\top) \to \mathbb{Z}^\top$$
Abstract evaluation of expressions is like the concrete evaluation — but with abstract values and operators. Here:

\[
\begin{align*}
[c]^\sharp D &= c \\
[e_1 \Box e_2]^\sharp D &= [e_1]^\# D \Box^\# [e_2]^\# D
\end{align*}
\]

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

\[ [c]^\# D = c \]
\[ [e_1 \boxtimes e_2]^\# D = [e_1]^\# D \boxtimes [e_2]^\# D \]

... analogously for unary operators

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

\[ = 2 + [7]^\# D \]
\[ = 2 +\# 7 \]
\[ = 9 \]

\[ [x - y]^\# D = 2 - [y]^\# \top \]
\[ = \top \]
Thus, we obtain the following abstract edge effects $[\text{lab}]^\#$:

\[
\begin{align*}
[;]^\# D &= D \\
[\text{true (} e \text{)}]^\# D &= \begin{cases} 
\bot & \text{if } 0 = [e]^\# D \\
D & \text{otherwise}
\end{cases} \quad \text{definitely false} \\
[\text{false (} e \text{)}]^\# D &= \begin{cases} 
D & \text{if } 0 \sqsubseteq [e]^\# D \\
\bot & \text{otherwise}
\end{cases} \quad \text{definitely true} \\
[x = e;]^\# D &= D \oplus \{x \mapsto [e]^\# D\} \\
[x = M[e];]^\# D &= D \oplus \{x \mapsto \top\} \\
[M[e_1] = e_2;]^\# D &= D
\end{align*}
\]

... whenever $D \neq \bot$
At *start*, we have \( D_{\top} = \{ x \mapsto \top \mid x \in Vars \} \).

**Example:**

\[
\begin{align*}
1 & \quad x = 7; \\
2 & \quad \text{Neg} (x > 0) \quad \text{Pos} (x > 0) \\
3 & \quad M[A] = B; \\
4 & \quad ; \\
5 &
\end{align*}
\]
At \textit{start}, we have \( D_T = \{ x \mapsto \top \mid x \in \text{Vars} \} \).

Example:

\begin{align*}
1 & \quad x = 7; \\
2 & \quad \text{Neg } (x > 0) \\
3 & \quad \text{Pos } (x > 0) \\
4 & \quad M[A] = B; \\
5 & \quad ;
\end{align*}

\begin{align*}
1 & \quad \{ x \mapsto \top \} \\
2 & \quad \{ x \mapsto 7 \} \\
3 & \quad \{ x \mapsto 7 \} \\
4 & \quad \{ x \mapsto 7 \} \\
5 & \quad \bot \sqcup \{ x \mapsto 7 \} = \{ x \mapsto 7 \}
\end{align*}

The abstract effects of edges \([k] \uparrow\) are again composed to form the effects of paths \( \pi = k_1 \ldots k_r \) by:

\[ [\pi] \uparrow = [k_r] \uparrow \circ \ldots \circ [k_1] \uparrow : \mathbb{D} \rightarrow \mathbb{D} \]
Idea for Correctness: Abstract Interpretation

Cousot, Cousot 1977

Establish a description relation $\Delta$ between the concrete values and their descriptions with:

$$x \Delta a_1 \land a_1 \sqsubseteq a_2 \implies x \Delta a_2$$

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

// returns the set of described values
Values: \[ \Delta \subseteq \mathbb{Z} \times \mathbb{Z}^\top \]

\( z \Delta a \iff z = a \lor a = \top \)

Concretization:

\[
\gamma a = \begin{cases} 
\{a\} & \text{if } a \sqsubseteq \top \\
\mathbb{Z} & \text{if } a = \top
\end{cases}
\]
(1) Values: \[ \Delta \subseteq \mathbb{Z} \times \mathbb{Z}^\top \]

\[ z \Delta a \quad \text{iff} \quad z = a \lor a = \top \]

Concretization:

\[ \gamma a = \begin{cases} \{a\} & \text{if } a \sqsubseteq \top \\ \mathbb{Z} & \text{if } a = \top \end{cases} \]

(2) Variable Bindings: \[ \Delta \subseteq (Vars \rightarrow \mathbb{Z}) \times (Vars \rightarrow \mathbb{Z}^\top)_\perp \]

\[ \rho \Delta D \quad \text{iff} \quad D \neq \perp \land \rho x \sqsubseteq D x \quad (x \in Vars) \]

Concretization:

\[ \gamma D = \begin{cases} \emptyset & \text{if } D = \perp \\ \{\rho \mid \forall x : (\rho x) \Delta (D x)\} & \text{otherwise} \end{cases} \]
Example: \( \{x \mapsto 1, y \mapsto -7\} \triangleq \{x \mapsto \top, y \mapsto -7\} \)

(3) States:

\[
\Delta \subseteq ((\text{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z})) \times (\text{Vars} \to \mathbb{Z}^\top)_{\bot}
\]

\[(\rho, \mu) \Delta D \quad \text{iff} \quad \rho \Delta D\]

Concretization:

\[
\gamma D = \begin{cases} 
\emptyset & \text{if } D = \bot \\
\{(\rho, \mu) | \forall x : (\rho x) \Delta (D x)\} & \text{otherwise}
\end{cases}
\]
We show correctness:

\((\ast)\) If \(s \triangle D\) and \([\pi]s\) is defined, then:

\[
([\pi]s) \triangle ([\pi]\# D)
\]
The abstract semantics simulates the concrete semantics

In particular:

$$\boxed{\pi} S \in \gamma (\boxed{\pi}^\# D)$$
The abstract semantics simulates the concrete semantics
In particular:

\[ [\pi] s \in \gamma ([\pi]_D) \]

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

\[
\rho' x = -7 \quad \text{for all} \quad \rho' \in \gamma D
\]

\[ \implies \rho_1 x = -7 \quad \text{for} \quad (\rho_1, _) = [\pi] s \]
The MOP-Solution:

$$D^*[v] = \bigsqcup \{ [[\pi]]^\# \ D_{\top} \mid \pi : start \rightarrow^* v \}$$

where $$D_{\top} x = \top \quad (x \in Vars)$$.

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

\[ x = 10; \]
\[ y = 1; \]
\[ M[R] = y; \]
\[ y = x \ast y; \]
\[ x = x - 1; \]
Example:

\[ x = 10; \]
\[ y = 1; \]
\[ M[R] = y; \]
\[ y = x \times y; \]
\[ x = x - 1; \]

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
```

Table:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

116
Example:

\[ x = 10; \]

\[ y = 1; \]

\[ \text{Neg}(x > 1) \]

\[ \text{Pos}(x > 1) \]

\[ M[R] = y; \]

\[ y = x \cdot y; \]

\[ x = x - 1; \]

\[ 0 \]

\[ 1 \]

\[ 2 \]

\[ 3 \]

\[ 4 \]

\[ 5 \]

\[ 6 \]

\[ 7 \]

\[ 1 \]

\[ 2 \]

\[ \begin{array}{|c|c|c|c|}
\hline
x & y & x & y \\
\hline
0 & \top & \top & \top & \top \\
1 & 10 & \top & 10 & \top \\
2 & 10 & 1 & \top & \top \\
3 & 10 & 1 & \top & \top \\
4 & 10 & 10 & \top & \top \\
5 & 9 & 10 & \top & \top \\
6 & \bot & & \top & \top \\
7 & \bot & & \top & \top \\
\hline
\end{array} \]
Example:

\[ x = 10; \]
\[ y = 1; \]
\[ \text{Neg}(x > 1) \]
\[ M[R] = y; \]
\[ y = x \ast y; \]
\[ x = x - 1; \]

\[
\begin{array}{c|c|c|c|c|c}
 & 1 & 2 & 3 \\
\hline
x & y & x & y & x & y \\
\hline
0 & \top & \top & \top & \top & \top \\
1 & 10 & \top & 10 & \top & \top \\
2 & 10 & 1 & \top & \top & \top \\
3 & 10 & 1 & \top & \top & \top \\
4 & 10 & 10 & \top & \top & \top \\
5 & 9 & 10 & \top & \top & \top \\
6 & \bot & \top & \top & \top & \top \\
7 & \bot & \top & \top & \top & \top \\
\end{array}
\]

dito
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:
\[ n \cdot (m + 1) \text{ rounds} \]

**Observation:** The effects of edges are **not distributive**!
Counterexample: $f = [x = x + y;]$. 

Let $D_1 = \{x \mapsto 2, y \mapsto 3\}$
$D_2 = \{x \mapsto 3, y \mapsto 2\}$

Then $f D_1 \sqcup f D_2 = \{x \mapsto 5, y \mapsto 3\} \sqcup \{x \mapsto 5, y \mapsto 2\}$
$= \{x \mapsto 5, y \mapsto \top\}$
$\neq \{x \mapsto \top, y \mapsto \top\}$
$= f \{x \mapsto \top, y \mapsto \top\}$
$= f (D_1 \sqcup D_2)$
We conclude:

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

$$\mathcal{D}^*[v] \subseteq \mathcal{D}[v]$$
We conclude:

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

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

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

$$([\pi](\rho, \mu)) \triangle (\mathcal{D}[v])$$

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:

\[
\text{for } (i = 0; i < 42; i++) \\
\quad \text{if } (0 \leq i \land i < 42)\{ \\
\quad\quad A_1 = A + i; \\
\quad\quad M[A_1] = i; \\
\quad\} \\
\]

// 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:

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

Partial order:

$$[l_1, u_1] \sqsubseteq [l_2, u_2] \quad \text{iff} \quad l_2 \leq l_1 \land u_1 \leq u_2$$
Thus:

\[
[l_1, u_1] \sqcup [l_2, u_2] = [l_1 \sqcap l_2, u_1 \sqcup u_2]
\]
Thus:

\[
[l_1, u_1] \sqcup [l_2, u_2] = [l_1 \sqcap l_2, u_1 \sqcup u_2] \\
[l_1, u_1] \sqcap [l_2, u_2] = [l_1 \sqcup l_2, u_1 \sqcap u_2] \quad \text{whenever } (l_1 \sqcup l_2) \leq (u_1 \sqcap u_2)
\]
Caveat:

→ $\mathbb{I}$ is not a complete lattice,

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

$$[0, 0] \sqsubseteq [0, 1] \sqsubseteq [-1, 1] \sqsubseteq [-1, 2] \sqsubseteq \ldots$$
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:

\[
z \Delta [l, u] \quad \text{iff} \quad l \leq z \leq u
\]

Concretization:

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

\[ \gamma [0, 7] = \{0, \ldots, 7\} \]
\[ \gamma [0, \infty] = \{0, 1, 2, \ldots, \} \]

Computing with intervals: Interval Arithmetic.

Addition:

\[ [l_1, u_1] +^\# [l_2, u_2] = [l_1 + l_2, u_1 + u_2] \quad \text{where} \]
\[ -\infty + _- = -\infty \]
\[ +\infty + _- = +\infty \]

// \(-\infty + \infty\) cannot occur
Negation:

$$-\# [l, u] = [-u, -l]$$

Multiplication:

$$[l_1, u_1] \ast\# [l_2, u_2] = [a, b] \quad \text{where}$$

$$a = l_1 l_2 \sqcap l_1 u_2 \sqcap u_1 l_2 \sqcap u_1 u_2$$

$$b = l_1 l_2 \sqcup l_1 u_2 \sqcup u_1 l_2 \sqcup u_1 u_2$$

Example:

$$[0, 2] \ast\# [3, 4] = [0, 8]$$

$$[-1, 2] \ast\# [3, 4] = [-4, 8]$$

$$[-1, 2] \ast\# [-3, 4] = [-6, 8]$$

$$[-1, 2] \ast\# [-4, -3] = [-8, 4]$$
Division: \[ [l_1, u_1] /\# [l_2, u_2] = [a, b] \]

- If 0 is not contained in the interval of the denominator, then:
  \[
  a = \frac{l_1}{l_2} \cap \frac{l_1}{u_2} \cap \frac{u_1}{l_2} \cap \frac{u_1}{u_2}
  \]
  \[
  b = \frac{l_1}{l_2} \cup \frac{l_1}{u_2} \cup \frac{u_1}{l_2} \cup \frac{u_1}{u_2}
  \]

- If: \( l_2 \leq 0 \leq u_2 \), we define:
  \[
  [a, b] = [-\infty, +\infty]
  \]
Equality:

\[ [l_1, u_1] = \# [l_2, u_2] = \begin{cases} 
true & \text{if } l_1 = u_1 = l_2 = u_2 \\
false & \text{if } u_1 < l_2 \lor u_2 < l_1 \\
\top & \text{otherwise}
\end{cases} \]
Equality:

\[
[l_1, u_1] == ^# [l_2, u_2] = \begin{cases} 
true & \text{if } l_1 = u_1 = l_2 = u_2 \\
false & \text{if } u_1 < l_2 \lor u_2 < l_1 \\
\top & \text{otherwise}
\end{cases}
\]

Example:

\[
[42, 42] == ^# [42, 42] = true \\
[0, 7] == ^# [0, 7] = \top \\
[1, 2] == ^# [3, 4] = false
\]
Less:

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

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

Example:

\[
[1, 2] \prec^# [9, 42] = true \\
[0, 7] \prec^# [0, 7] = \top \\
[3, 4] \prec^# [1, 2] = false
\]
By means of \( I \) we construct the complete lattice:

\[
\mathbb{D}_I = (Vars \rightarrow I)_\perp
\]

**Description Relation:**

\[
\rho \triangle D \quad \text{iff} \quad D \neq \perp \quad \land \quad \forall x \in Vars : (\rho x) \triangle (D x)
\]

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

\[
([e] \rho) \triangle ([e]^{\sharp} D) \quad \text{whenever} \quad \rho \triangle D
\]
The Effects of Edges:

\[
\begin{align*}
[;]^\# D &= D \\
[x = e;]^\# D &= D \oplus \{x \mapsto [e]^\# D\} \\
[x = M[e];]^\# D &= D \oplus \{x \mapsto \top\} \\
[M[e_1] = e_2;]^\# D &= D \\
[\text{true } (e)]^\# D &= \begin{cases} 
\bot & \text{if definitely false} \\
D & \text{otherwise possibly true} 
\end{cases} \\
[\text{false } (e)]^\# D &= \begin{cases} 
D & \text{possibly false} \\
\bot & \text{definitely true} 
\end{cases}
\end{align*}
\]

... given that \( D \neq \bot \)
Better Exploitation of Conditions:

\[
[\text{Pos}(e)]^\# D = \begin{cases} 
\bot & \text{if } \text{false} = [e]^\# D \\
D_1 & \text{otherwise}
\end{cases}
\]

where:

\[
D_1 = \begin{cases} 
D \oplus \{x \mapsto (D x) \cap ([e_1]^\# D)\} & \text{if } e \equiv x = e_1 \\
D \oplus \{x \mapsto (D x) \cap [-\infty, u]\} & \text{if } e \equiv x \leq e_1, [e_1]^\# D = [\_, u] \\
D \oplus \{x \mapsto (D x) \cap [l, \infty]\} & \text{if } e \equiv x \geq e_1, [e_1]^\# D = [l, \_]
\end{cases}
\]
Better Exploitation of Conditions (cont.):

\[
\lbrack \text{Neg}(e) \rbrack \# D = \begin{cases} 
\bot & \text{if } \text{false} \not\sqsubseteq \lbrack e \rbrack \# D \\
D_1 & \text{otherwise}
\end{cases}
\]

where:

\[
D_1 = \begin{cases} 
D \oplus \{x \mapsto (D_x) \cap \lbrack \lbrack e_1 \rbrack \# D \rbrack \} & \text{if } e \equiv x \neq e_1 \\
D \oplus \{x \mapsto (D_x) \cap [\infty, u] \} & \text{if } e \equiv x > e_1, \lbrack e_1 \rbrack \# D = [\_, u] \\
D \oplus \{x \mapsto (D_x) \cap [l, \infty] \} & \text{if } e \equiv x < e_1, \lbrack e_1 \rbrack \# D = [l, \_] \end{cases}
\]
Example:

\[ i = 0; \]

\[ i = i + 1; \]

\[ M[A_1] = i; \]

\[ A_1 = A + i; \]

\[
\begin{array}{c|cc|c|c}
     i   & l &  u \\
\hline
     0   & -\infty & +\infty \\
     1   & 0   &  42 \\
     2   & 0   &  41 \\
     3   & 0   &  41 \\
     4   & 0   &  41 \\
     5   & 0   &  41 \\
     6   & 1   &  42 \\
     7   & + &  42 \\
     8   & 42 &  42 \\
\end{array}
\]
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 \( x_i \sqsubseteq f_i (x_1, \ldots, x_n) \), \( i = 1, \ldots, n \) denote a system of constraints over \( \mathbb{D} \)

Define an accumulating iteration:

\[
x_i = x_i \sqcup f_i (x_1, \ldots, x_n), \quad i = 1, \ldots, n
\]

We obviously have:

(a) \( x \) is a solution of (1) iff \( x \) 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., \( x \sqsubseteq G x \) for all \( x \in \mathbb{D}^n \).
(c) The sequence $G^k \perp$, $k \geq 0$, is an ascending chain:

\[
\perp \subseteq G \perp \subseteq \ldots \subseteq G^k \perp \subseteq \ldots
\]

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

\[
x_i = x_i \sqcup f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n
\]  \hspace{1cm} (3)

for a binary operation **widening**:

\[
\sqcup : \mathbb{D}^2 \rightarrow \mathbb{D} \quad \text{with} \quad v_1 \sqcup v_2 \subseteq v_1 \sqcup v_2
\]

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

- The complete lattice is: \( \mathbb{D}_I = (\text{Vars} \rightarrow \mathbb{I})_\perp \)
- the widening \( \sqcup \) is defined by:

\[
\perp \sqcup D = D \sqcup \perp = D
\]

and for \( D_1 \neq \perp \neq D_2 \):

\[
(D_1 \sqcup D_2) x = (D_1 x) \sqcup (D_2 x)
\]

where

\[
[l_1, u_1] \sqcup [l_2, u_2] = [l, u]
\]

with

\[
l = \begin{cases} 
  l_1 & \text{if } l_1 \leq l_2 \\
  -\infty & \text{otherwise}
\end{cases}
\]

\[
u = \begin{cases} 
  u_1 & \text{if } u_1 \geq u_2 \\
  +\infty & \text{otherwise}
\end{cases}
\]

\( \Rightarrow \) \( \sqcup \) is not commutative !!!
Example:

\[
[0, 2] \sqcup [1, 2] = [0, 2] \\
[1, 2] \sqcup [0, 2] = \left[ -\infty, 2 \right] \\
[1, 5] \sqcup [3, 7] = [1, +\infty]
\]

→ 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:

\[
\#points \cdot (1 + 2 \cdot \#Vars)
\]
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 ⊔ is chosen dynamically during iteration such that
  → the abstract values do not get too complicated;
  → the number of updates remains bounded ...
Our Example:

\[
i = 0; \\
\text{Neg}(i < 42) \\
\text{Neg}(0 \leq i < 42) \\
\text{Pos}(i < 42) \\
\text{Pos}(0 \leq i < 42) \\
\]

\[
A_1 = A + i; \\
M[A_1] = i; \\
i = i + 1;
\]

\[
\begin{array}{|c|c|c|}
\hline
l & u \\
\hline
0 & -\infty & +\infty \\
1 & 0 & 0 \\
2 & 0 & 0 \\
3 & 0 & 0 \\
4 & 0 & 0 \\
5 & 0 & 0 \\
6 & 1 & 1 \\
7 & \perp \\
8 & \perp \\
\hline
\end{array}
\]
Our Example:

- Neg\(i < 42\) \(\implies\) Pos\(i < 42\)
- Neg\(0 \leq i < 42\) \(\implies\) Pos\(0 \leq i < 42\)

\[
i = 0;
\]

\[
M[A_1] = i;
\]

\[
i = i + 1;
\]

\[
A_1 = A + i;
\]

\[
l \quad u
\]

\[
\begin{array}{|c|c|c|c|}
\hline
\text{l} & \text{u} & \text{l} & \text{u} \\
\hline
0 & -\infty & -\infty & +\infty \\
1 & 0 & 0 & +\infty \\
2 & 0 & 0 & +\infty \\
3 & 0 & 0 & +\infty \\
4 & 0 & 0 & +\infty \\
5 & 0 & 0 & +\infty \\
6 & 1 & 1 & +\infty \\
7 & \perp & 42 & +\infty \\
8 & \perp & 42 & +\infty \\
\hline
\end{array}
\]
7 Removing superfluous computations

A computation may be superfluous because

- the result is already available, → available-expression analysis, or
- the result is not needed → 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

→ store this value after the first computation;

→ 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:

\[
\begin{align*}
z &= 1; \\
y &= M[17]; \\
A: \quad x_1 &= y + z; \\
\quad \ldots \\
B: \quad x_2 &= y + z;
\end{align*}
\]

\textit{B} is a \textbf{redundant} computation of the value of \(y + z\), 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 \( u \)

\[
\begin{array}{c}
\text{x+y} \\
u \xrightarrow{\pi} v
\end{array}
\]

and a computation along path \( \pi \) reaches \( v \) where it evaluates again \( x + y \)

....

If \( x \) and \( y \) have not been modified in \( \pi \), then evaluation of \( x + y \) at \( v \) returns the same value as evaluation at \( u \).

This property can be checked at every edge in \( \pi \).
**Situation:** The value of $x + y$ is computed at program point $u$

![](x+y)

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

... If $x$ and $y$ have not been modified in $\pi$, 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 $\pi$.

**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 \( u \)

\[
x + y
\]

\[ u \xrightarrow{\pi} v \]

and a computation along path \( \pi \) reaches \( v \) where it evaluates again \( x + y \)

.... If \( x \) and \( y \) have not been modified in \( \pi \), 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 \( \pi \).

**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 \( [[k]]^\# A \) of expressions whose values are available after execution of \( k \).

\( [[k]]^\# A \) is the (abstract) edge effect associated with \( k \).
These edge effects can be composed to the effect of a path $\pi = k_1 \ldots k_r$:

$$[\pi]^\# = [k_r]^\# \circ \ldots \circ [k_1]^\#$$
These edge effects can be composed to the effect of a path \( \pi = k_1 \ldots k_r \):

\[
[\pi]^{\#} = [k_r]^{\#} \circ \ldots \circ [k_1]^{\#}
\]

The effect \([k]^{\#}\) of an edge \(k = (u, \text{lab}, v)\) only depends on the label \(\text{lab}\), i.e.,

\[
[k]^{\#} = [\text{lab}]^{\#}
\]
These edge effects can be composed to the effect of a path $\pi = k_1 \ldots k_r$:

$$[\pi]^{\#} = [k_r]^{\#} \circ \ldots \circ [k_1]^{\#}$$

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

$$[;]^{\#} A = A$$

$$[Pos(e)]^{\#} A = [Neg(e)]^{\#} A = A \cup \{e\}$$

$$[x = e;]^{\#} A = (A \cup \{e\}) \setminus \text{Expr}_x$$

where $\text{Expr}_x$ all expressions that contain $x$

$$[x = M[e];]^{\#} A = (A \cup \{e\}) \setminus \text{Expr}_x$$

$$[M[e_1] = e_2;]^{\#} A = A \cup \{e_1, e_2\}$$
→ An expression is available at $v$ if it is available along all paths $\pi$ to $v$.

→ For every such path $\pi$, the analysis determines the set of expressions that are available along $\pi$.

→ Initially at program start, nothing is available.

→ The analysis computes the intersection of the availability sets as safe information.

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

$$ \mathcal{A}[v] = \bigcap \{ [[\pi]]^\# \emptyset \mid \pi : \text{start} \rightarrow^* v \} $$
How does a compiler exploit this information?

Transformation UT (unique temporaries):

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

```
\begin{align*}
  u & \quad x = e; \\
  v & \\
  u & \quad T_e = e; \\
  v & \quad x = T_e;
\end{align*}
```
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:

\[
x = y + 3;
x = 7;
z = y + 3;
\]
Example:

\[ x = y + 3; \]
\[ x = 7; \]
\[ z = y + 3; \]
Example:

\[ x = y + 3; \]
\[ x = 7; \]
\[ z = y + 3; \]

\[ T = y + 3; \]
\[ x = T; \]
\[ \{y + 3\} \]

\[ T = y + 3; \]
\[ x = 7; \]
\[ \{y + 3\} \]

\[ T = y + 3; \]
\[ \{y + 3\} \]

\[ z = T; \]
\[ \{y + 3\} \]
Example:

\[
x = y + 3;
\]
\[
x = 7;
\]
\[
z = y + 3;
\]
\[
T = y + 3;
\]
\[
\{y + 3\}
\]
\[
\{y + 3\}
\]
\[
\{y + 3\}
\]
\[
\{y + 3\}
\]
\[
\{y + 3\}
\]
Warning:

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

$\rightarrow x \not\in \text{Vars}(e)$; why?

$\rightarrow e \not\in \text{Vars}$; why?

$\rightarrow$ the evaluation of $e$ is non-trivial; why?
Warning:

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

$\rightarrow x \not\in Vars(e)$; otherwise $e$ is not available afterwards.

$\rightarrow e \not\in Vars$; 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{align*}
\mathcal{A}[\text{start}] & \subseteq \emptyset \\
\mathcal{A}[v] & \subseteq [k]^\#(\mathcal{A}[u]) \quad k = (u, -, v) \text{ edge}
\end{align*}

Why $\subseteq$?
Question:

How can we compute $A[u]$ for every program point? $u$

Idea:

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

$$
\begin{align*}
A[start] & \subseteq \emptyset \\
A[v] & \subseteq \llbracket k \rrbracket^\#(A[u]) & k = (u, _, v) \text{\ edge}
\end{align*}
$$

Why $\subseteq$?

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

$$
A[v] \subseteq \bigcap\{\llbracket k \rrbracket^\#(A[u]) \mid k = (u, _, v) \text{\ edge}\}
$$
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:

\[
\begin{align*}
\text{Neg}(x > 1) & \quad \text{Pos}(x > 1) \\
5 & \quad 2 \\
0 & \quad 1 \\
& \quad y = x \ast y; \\
& \quad x = x - 1; \\
& \quad \mathcal{A}[0] \subseteq \emptyset
\end{align*}
\]
Wanted:

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

Example:

\[ A[0] \subseteq \emptyset \]
\[ A[1] \subseteq (A[0] \cup \{1\}) \backslash \text{Expr}_y \]
Wanted:

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

Example:

\[
\begin{align*}
A[0] & \subseteq \emptyset \\
A[1] & \subseteq (A[0] \cup \{1\}) \setminus Expr_y \\
\end{align*}
\]
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:

\[ A[0] \subseteq \emptyset \]
\[ A[1] \subseteq (A[0] \cup \{1\}) \setminus Expr_y \]
\[ A[2] \subseteq A[1] \cup \{x > 1\} \]
\[ A[3] \subseteq (A[2] \cup \{x \times y\}) \setminus Expr_y \]
Wanted:

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

Example:

\[
\begin{align*}
\mathcal{A}[0] & \subseteq \emptyset \\
\mathcal{A}[1] & \subseteq (\mathcal{A}[0] \cup \{1\}) \backslash \text{Expr}_y \\
\mathcal{A}[1] & \subseteq \mathcal{A}[4] \\
\mathcal{A}[2] & \subseteq \mathcal{A}[1] \cup \{x > 1\} \\
\mathcal{A}[3] & \subseteq (\mathcal{A}[2] \cup \{x \cdot y\}) \backslash \text{Expr}_y \\
\mathcal{A}[4] & \subseteq (\mathcal{A}[3] \cup \{x - 1\}) \backslash \text{Expr}_x \\
\mathcal{A}[5] & \subseteq \mathcal{A}[1] \cup \{x > 1\}
\end{align*}
\]
Wanted:

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

Example:

Solution:

\[
\begin{align*}
A[0] &= \emptyset \\
A[1] &= \{1\} \\
A[2] &= \{1, x > 1\} \\
A[3] &= \{1, x > 1\} \\
A[4] &= \{1\} \\
A[5] &= \{1, x > 1\}
\end{align*}
\]
Observation:

- Again, the possible values for $\mathcal{A}[u]$ form a complete lattice:
  \[ \mathbb{D} = 2^\text{Expr} \text{ with } B_1 \subseteq B_2 \text{ iff } B_1 \supseteq B_2 \]

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

- Again, the possible values for $A[u]$ form a complete lattice:

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

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

- The functions $\lbrack k \rbrack^\# : \mathbb{D} \to \mathbb{D}$ have the form $f_i x = a_i \cap x \cup b_i$. They are called *gen/kill* functions—$\cap$ kills, $\cup$ generates.

- They are monotonic, i.e.,

$$\lbrack k \rbrack^\#(B_1) \subseteq \lbrack k \rbrack^\#(B_2) \quad \text{iff} \quad B_1 \subseteq B_2$$
The operations “◦”, “⊔” and “⊓” can be explicitly defined by:

\[(f_2 \circ f_1)x = a_1 \cap a_2 \cap x \cup a_2 \cap b_1 \cup b_2\]

\[(f_1 \sqcup f_2)x = (a_1 \cup a_2) \cap x \cup b_1 \cup b_2\]

\[(f_1 \sqcap f_2)x = (a_1 \cup b_1) \cap (a_2 \cup b_2) \cap x \cup b_1 \cap b_2\]
7.2 Removing Assignments to Dead Variables

Example:

1:  \( x = y + 2; \)
2:  \( y = 5; \)
3:  \( x = y + 3; \)

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:

→ 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 a path $\pi$ starting at $u$ if $\pi$ can be decomposed into $\pi = \pi_1 k \pi_2$ such that:

- $k$ is a use of $x$ and
- $\pi_1$ does not contain a definition of $x$. 
Thereby, the set of all defined or used variables at an edge $k = (_-, lab, _-)$ is defined by

<table>
<thead>
<tr>
<th>$lab$</th>
<th>used</th>
<th>defined</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>Pos($e$)</td>
<td>$\mathit{Vars}(e)$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>Neg($e$)</td>
<td>$\mathit{Vars}(e)$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$x = e;$</td>
<td>$\mathit{Vars}(e)$</td>
<td>${x}$</td>
</tr>
<tr>
<td>$x = M[e]$;</td>
<td>$\mathit{Vars}(e)$</td>
<td>${x}$</td>
</tr>
<tr>
<td>$M[e_1] = e_2;$</td>
<td>$\mathit{Vars}(e_1) \cup \mathit{Vars}(e_2)$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
A variable $x$ which is not live at $u$ along $\pi$ is called dead at $u$ along $\pi$.

Example:

\[ x = y + 2; \quad y = 5; \quad x = y + 3; \]

Then we observe:

<table>
<thead>
<tr>
<th></th>
<th>live</th>
<th>dead</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>${y}$</td>
<td>${x}$</td>
</tr>
<tr>
<td>1</td>
<td>$\emptyset$</td>
<td>${x, y}$</td>
</tr>
<tr>
<td>2</td>
<td>${y}$</td>
<td>${x}$</td>
</tr>
<tr>
<td>3</td>
<td>$\emptyset$</td>
<td>${x, y}$</td>
</tr>
</tbody>
</table>
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 $[k]^\dagger$ 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 $L = 2^{\text{Vars}}$.

For $k = (\_, \text{lab}, \_)$, define $[k]^\# = [\text{lab}]^\#$ by:

$$
\begin{align*}
[\_]^\# L &= L \\
[\text{Pos}(e)]^\# L &= [\text{Neg}(e)]^\# L = L \cup \text{Vars}(e) \\
[x = e;]^\# L &= (L \setminus \{x\}) \cup \text{Vars}(e) \\
[x = M[e];]^\# L &= (L \setminus \{x\}) \cup \text{Vars}(e) \\
[M[e_1] = e_2;]^\# L &= L \cup \text{Vars}(e_1) \cup \text{Vars}(e_2)
\end{align*}
$$
Let \( \mathbb{L} = 2^{\text{Vars}} \).

For \( k = (_, \text{lab}, _) \), define \( [k]^\# = [\text{lab}]^\# \) by:

\[
\begin{align*}
[;]^\# L & = L \\
[\text{Pos}(e)]^\# L & = [\text{Neg}(e)]^\# L = L \cup \text{Vars}(e) \\
[x = e;]^\# L & = (L \setminus \{x\}) \cup \text{Vars}(e) \\
[x = M[e];]^\# L & = (L \setminus \{x\}) \cup \text{Vars}(e) \\
[M[e_1] = e_2;]^\# L & = L \cup \text{Vars}(e_1) \cup \text{Vars}(e_2)
\end{align*}
\]

\( [k]^\# \) can again be composed to the effects of \( [\pi]^\# \) of paths
\( \pi = k_1 \ldots k_r \) by:

\[
[\pi]^\# = [k_1]^\# \circ \ldots \circ [k_r]^\#
\]
We verify that these definitions are meaningful

\[ x = y + 2; \quad y = 5; \quad x = y + 2; \quad M[y] = x; \]
We verify that these definitions are meaningful.

\[ M[y] = x; \]

\[ y = 5; \]

\[ x = y + 2; \]

\[ x = y + 2; \]

\[ 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \]

\( \emptyset \)
We verify that these definitions are meaningful

\[ x = y + 2; \quad y = 5; \quad x = y + 2; \quad M[y] = x; \]

\[ \emptyset \{x, y\} \]
We verify that these definitions are meaningful.
We verify that these definitions are meaningful
We verify that these definitions are meaningful

\[ x = y + 2; \quad y = 5; \quad x = y + 2; \quad M[y] = x; \]

1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5

\{y\} \quad \emptyset \quad \{y\} \quad \{x, y\} \quad \emptyset
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:

$$\mathcal{L}^*[u] = \bigcup \{ [[\pi]] \#\emptyset \mid \pi : u \rightarrow^* \text{stop} \}$$

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):

\[ x = e; \quad x \not\in L^\ast[v] \]

\[ x = M[e]; \quad x \not\in L^\ast[v] \]
Correctness Proof:

\[ \rightarrow \text{Correctness of the effects of edges: If } L \text{ is the set of variables which are live at the exit of the path } \pi, \text{ then } \lbrack \pi \rbrack^\# L \text{ is the set of variables which are live at the beginning of } \pi \]

\[ \rightarrow \text{Correctness of the transformation along a path: If the value of a variable is accessed, this variable is necessarily live. The value of dead variables thus is irrelevant} \]

\[ \rightarrow \text{Correctness of the transformation: In any execution of the transformed programs, the live variables always receive the same values} \]
Computation of the sets $L^*[u]$ :

(1) Collecting constraints:

\[ L[\text{stop}] \supseteq \emptyset \]
\[ L[u] \supseteq [k]^\#(L[v]) \quad k = (u, _, v) \quad \text{edge} \]

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

Since $L$ is finite, the iteration will terminate.

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

(1) Collecting constraints:

$$
\begin{align*}
\mathcal{L}[\text{stop}] & \supseteq \emptyset \\
\mathcal{L}[u] & \supseteq [k]^\#(\mathcal{L}[v])\quad k = (u, \_, v) \text{ edge}
\end{align*}
$$

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

Since $\mathcal{L}$ 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 $[k]^\#$ are distributive.

Note: The information is propagated backwards!
Example:

\[ x = M[I]; \]
\[ y = 1; \]
\[ M[R] = y; \]
\[ x = x - 1; \]
\[ y = x \cdot y; \]
\[ \text{Neg}(x > 1) \]
\[ \text{Pos}(x > 1) \]

\[ \mathcal{L}[0] \supset (\mathcal{L}[1]\{x\}) \cup \{I\} \]
\[ \mathcal{L}[1] \supset \mathcal{L}[2]\{y\} \]
\[ \mathcal{L}[2] \supset (\mathcal{L}[6] \cup \{x\}) \cup (\mathcal{L}[3] \cup \{x\}) \]
\[ \mathcal{L}[3] \supset (\mathcal{L}[4]\{y\}) \cup \{x, y\} \]
\[ \mathcal{L}[4] \supset (\mathcal{L}[5]\{x\}) \cup \{x\} \]
\[ \mathcal{L}[5] \supset \mathcal{L}[2] \]
\[ \mathcal{L}[6] \supset \mathcal{L}[7] \cup \{y, R\} \]
\[ \mathcal{L}[7] \supset \emptyset \]
Example:

$$x = M[I];$$

$$y = 1;$$

Neg($$x > 1$$)

$$M[R] = y;$$

Pos($$x > 1$$)

$$y = x \times y;$$

$$x = x - 1;$$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>${y, R}$</td>
<td>dito</td>
</tr>
<tr>
<td>2</td>
<td>${x, y, R}$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>${x, y, R}$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>${x, y, R}$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>${x, y, R}$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>${x, R}$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>${I, R}$</td>
<td></td>
</tr>
</tbody>
</table>
The left-hand side of no assignment is **dead**

**Caveat:**

Removal of assignments to dead variables may kill further variables:

\[
x = y + 1;
\]

\[
z = 2 \times x;
\]

\[
M[R] = y;
\]

\[
\emptyset
\]
The left-hand side of no assignment is dead

Caveat:

Removal of assignments to dead variables may kill further variables:

\[ x = y + 1; \]
\[ z = 2 \times x; \]
\[ M[R] = y; \]
\[ \emptyset \]
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:

1. \( y, R \)
   
   \( x = y + 1; \)

2. \( x, y, R \)
   
   \( z = 2 \times x; \)

3. \( y, R \)
   
   \( M[R] = y; \)

4. \( \emptyset \)
The left-hand side of no assignment is **dead**

**Caveat:**

Removal of assignments to dead variables may kill further variables:

\[
\begin{align*}
1 & : y, R \\
2 & : x = y + 1; x, y, R \\
3 & : z = 2 \times x; y, R \\
4 & : M[R] = y; \emptyset
\end{align*}
\]

\[
\begin{align*}
1 & : x = y + 1; \\
2 & : ; \\
3 & : y, R \\
4 & : M[R] = y;
\end{align*}
\]
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:

\[
\begin{align*}
1 & : y, R \\
2 & : x = y + 1; \\
3 & : x, y, R \\
4 & : z = 2 \times x; \\
5 & : y, R \\
6 & : M[R] = y; \\
7 & : \emptyset
\end{align*}
\]

\[
\begin{align*}
1 & : y, R \\
2 & : x = y + 1; \\
3 & : y, R \\
4 & : M[R] = y; \\
5 & : \emptyset
\end{align*}
\]
Re-analyzing the program is inconvenient

**Idea:** Analyze **true** liveness!

$x$ is called **truly live** at $u$ along a path $\pi$, either if $\pi$ can be decomposed into $\pi = \pi_1 \ k \ \pi_2$ such that:

- $k$ is a **true** use of $x$;
- $\pi_1$ does not contain any **definition** of $x$. 
The set of truly used variables at an edge $k = (\_ , \text{lab}, v)$ is defined as:

<table>
<thead>
<tr>
<th>lab</th>
<th>truly used</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>Pos ($e$)</td>
<td>$\text{Vars} (e)$</td>
</tr>
<tr>
<td>Neg ($e$)</td>
<td>$\text{Vars} (e)$</td>
</tr>
<tr>
<td>$x = e;$</td>
<td>$\text{Vars} (e)$ ($\ast$)</td>
</tr>
<tr>
<td>$x = M[e];$</td>
<td>$\text{Vars} (e)$ ($\ast$)</td>
</tr>
<tr>
<td>$M[e_1] = e_2;$</td>
<td>$\text{Vars}(e_1) \cup \text{Vars}(e_2)$</td>
</tr>
</tbody>
</table>

($\ast$) – given that $x$ is truly live at $v$
Example:

\[ x = y + 1; \]
\[ z = 2 \ast x; \]
\[ M[R] = y; \]
\[ \emptyset \]
Example:

\[ x = y + 1; \]
\[ z = 2 \times x; \]
\[ y, R \]
\[ M[R] = y; \]
\[ \emptyset \]
Example:

\[ x = y + 1; \]

\[ y, R \]

\[ z = 2 \ast x; \]

\[ y, R \]

\[ M[R] = y; \]

\[ \emptyset \]
Example:

\[
\begin{align*}
1 & : \quad y, R \\
   & \quad x = y + 1; \\
2 & : \quad y, R \\
   & \quad z = 2 \times x; \\
3 & : \quad y, R \\
   & \quad M[R] = y; \\
4 & : \quad \emptyset
\end{align*}
\]
Example:

\[ \begin{align*}
1 & \quad y, R \\
2 & \quad x = y + 1; \\
3 & \quad y, R \\
4 & \quad z = 2 \times x; \\
3 & \quad y, R \\
4 & \quad M[R] = y; \\
4 & \quad \emptyset
\end{align*} \]
The Effects of Edges:

\[\text{[;] }^\# L = L\]
\[\text{[Pos}(e)\text{]}^\# L = [\text{Neg}(e)]^\# L = L \cup \text{Vars}(e)\]
\[\text{[x = e;]}^\# L = (L\{x\}) \cup \text{Vars}(e)\]
\[\text{[x = M[e];]}^\# L = (L\{x\}) \cup \text{Vars}(e)\]
\[\text{[M[e_1] = e_2;]}^\# L = L \cup \text{Vars}(e_1) \cup \text{Vars}(e_2)\]
The Effects of Edges:

\[
\begin{align*}
[;]\#L & = L \\
[\text{Pos}(e)]\#L & = [\text{Neg}(e)]\#L = L \cup \text{Vars}(e) \\
[x = e;]\#L & = (L \setminus \{x\}) \cup (x \in L) ? \text{Vars}(e) : \emptyset \\
[x = M[e];]\#L & = (L \setminus \{x\}) \cup (x \in L) ? \text{Vars}(e) : \emptyset \\
[M[e_1] = e_2;]\#L & = L \cup \text{Vars}(e_1) \cup \text{Vars}(e_2)
\end{align*}
\]
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:

$$f(y_1 \cup y_2) = (u \in y_1 \cup y_2) ? b : \emptyset$$

$$= (u \in y_1 \lor u \in y_2) ? b : \emptyset$$

$$= (u \in y_1) ? b : \emptyset \cup (u \in y_2) ? b : \emptyset$$

$$= f(y_1) \cup f(y_2)$$
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:

$$f (y_1 \cup y_2) = (u \in y_1 \cup y_2)? b : \emptyset$$

$$= (u \in y_1 \lor u \in y_2)? b : \emptyset$$

$$= (u \in y_1)? b : \emptyset \cup (u \in y_2)? b : \emptyset$$

$$= f y_1 \cup f y_2$$

$\implies$ the constraint system yields the MOP.
• True liveness detects more superfluous assignments than repeated liveness !!!

\[ x = x - 1; \]
True liveness detects more superfluous assignments than repeated liveness !!!

Liveness:

\[
\{x\} \xrightarrow{\phantom{\{}x\phantom{\}}} x = x - 1;
\]

\[
\emptyset \xrightarrow{\phantom{\{}x\phantom{\}}} x = x - 1;
\]
True liveness detects more superfluous assignments than repeated liveness !!!

True Liveness:

\[
x = x - 1;
\]