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 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*
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 “$\subseteq$” = “$\supseteq$".

```
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 \( \mathbb{D} = (\text{Vars} \to Signs)_{\bot} \), 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?
The analysis should "bind" program variables to elements in \textit{Signs}.

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

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

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?
How is a solution found?
Iterating until a fixed-point is reached

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

- **0**: \( x = 0 \)
- **1**: \( y = 1 \)
- **2**: \( \text{true}(y > 0) \) and \( \text{false}(y > 0) \)
- **4**: \( y = y + x \)
- **5**: \( x = x + (-1) \)
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 $\square$ working on values by abstract operators $\square\#$ 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 \( \diamond \) working on values by abstract operators \( \diamond \# \) working on signs:
- The abstract operators allow to define an abstract evaluation of expressions:

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

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

\[ \times \# \]

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

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

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

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

It depends only on the label $lab$:

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

... whenever $D \neq \bot$

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 $v$:

→ 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 $\pi$.

→ Initially at program start, no information about signs is available.

→ The analysis computes a superset of the set of signs as safe information.

⇒ For each node $v$, we need the set:

$$S[v] = \bigsqcup \{ [[\pi]]^T \mid \pi : start \rightarrow^* v \}$$
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:

$$
S[start] \equiv \top
$$
$$
S[v] \equiv \[#k\# (S[u]) \quad k = (u, _, v) \quad \text{edge}
$$
Wanted:

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

Example:
\( S[0] \supseteq \top \)
\( S[1] \supseteq S[0] \oplus \{ x \mapsto \{0\} \} \)
\( S[2] \supseteq S[1] \oplus \{ y \mapsto \{+\} \} \)
\( S[2] \supseteq S[5] \oplus \{ x \mapsto [x + (-1)]^{\#} S[5] \} \)
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>

Edge labels:
- Test : $\text{Pos}(e)$ or $\text{Neg}(e)$
- Assignment : $R = e$
- Load : $R = M[e]$
- Store : $M[e_1] = e_2$
- Nop : $\cdot$

34
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**:  \( \text{Pos} \ (e) \) or \( \text{Neg} \ (e) \)

**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:

| $\rho : Vars \rightarrow \text{int}$ | 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:
\[ [;] (\rho, \mu) = (\rho, \mu) \]

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

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

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

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

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

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

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

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

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

// \ [e] \ : \ \text{evaluation of the expression} \ e, \ \text{e.g.}

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

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

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

// \ \text{where} \ \text{“} \oplus \text{”} \ \text{modifies a mapping at a given argument}
\[ [R = M[e];] \ (\rho, \mu) \ = \ (\rho \oplus \{ R \mapsto \mu([e] \rho) \}, \mu) \]

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

Example:

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

\[ \rho \ = \ \{ x \mapsto 5 \} \oplus \{ x \mapsto [x + 1] \{ x \mapsto 5 \} \} \]
\[ = \ \{ x \mapsto 5 \} \oplus \{ x \mapsto 6 \} \]
\[ = \ \{ x \mapsto 6 \} \]

A path \( \pi = k_1 k_2 \ldots k_m \) defines a computation in the state \( s \) if

\[ s \in \text{def} \left( [k_m] \circ \ldots \circ [k_1] \right) \]

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 $a, b, c \in \mathbb{D}$,

- $a \sqsubseteq a$ \hspace{1cm} \text{reflexivity}
- $a \sqsubseteq b \land b \sqsubseteq a \implies a = b$ \hspace{1cm} \text{anti–symmetry}
- $a \sqsubseteq b \land b \sqsubseteq c \implies a \sqsubseteq c$ \hspace{1cm} \text{transitivity}

\textbf{Intuition:} $\sqsubseteq$ represents \textit{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. \( \mathbb{Z}_\perp = \mathbb{Z} \cup \{ \perp \} \) 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

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

$d$ is called least upper bound (lub) if

1. $d$ is an upper bound and
2. $d \subseteq y$ 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

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

$\rightarrow$ 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
   
   $\mathbb{Z}_\perp^\top = \mathbb{Z} \cup \{\bot, \top\}$ :

   ![Lattice Diagram](image)
Theorem:

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

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

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

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

Correct because:

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

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

(\star)

Relation to the running example:

<table>
<thead>
<tr>
<th>( x_i )</th>
<th>unknown</th>
<th>here: ( S[u] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D )</td>
<td>values</td>
<td>here: ( Signs )</td>
</tr>
<tr>
<td>( \sqsubseteq \subseteq D \times D )</td>
<td>ordering relation</td>
<td>here: ( \subseteq )</td>
</tr>
<tr>
<td>( f_i: D^n \rightarrow D )</td>
<td>constraint</td>
<td>here: ( \ldots )</td>
</tr>
</tbody>
</table>
A mapping \( f : \mathbb{D}_1 \rightarrow \mathbb{D}_2 \) is called \textit{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
\[ f(a) \sqsubseteq f(b) \quad \text{for all} \quad a \sqsubseteq b. \]

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 \( f(a) \subseteq f(b) \) 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.
A mapping \( f : \mathbb{D}_1 \rightarrow \mathbb{D}_2 \) is called **monotonic (order preserving)** if \( f(a) \preceq f(b) \) for all \( a \preceq b \).

**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 “\( \preceq \)“). Then:

- \( \text{inc} \) \( x = x + 1 \) is monotonic.
- \( \text{dec} \) \( x = x - 1 \) is monotonic.
- \( \text{inv} \) \( x = -x \) is not monotonic
Theorem:

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

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

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

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

(*)

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

Idea:

- Consider \( F : \mathbb{D}^n \rightarrow \mathbb{D}^n \) 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:

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

(\ast)

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

Idea:

- Consider \( F : \mathbb{D}^n \rightarrow \mathbb{D}^n \) 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

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

(*)

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

Idea:

- Consider \( F : \mathbb{D}^n \rightarrow \mathbb{D}^n \) 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:
  \[ \perp, \quad F \perp, \quad F^2 \perp, \quad F^3 \perp, \quad \ldots \]

Intuition: This iteration eliminates unjustified assumptions.

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

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

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

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>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>(x_2)</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>(x_3)</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</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></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: \[ \mathcal{D} = 2\{a, b, c\}, \quad \subseteq = \subseteq \]

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

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></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: \[ \mathbb{D} = 2^{\{a,b,c\}}, \quad \subseteq = \subseteq \]

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

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></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>
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>\text{dito}</td>
</tr>
<tr>
<td>( x_2 )</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>{a}</td>
<td>\text{dito}</td>
</tr>
<tr>
<td>( x_3 )</td>
<td>\emptyset</td>
<td>{c}</td>
<td>{a, c}</td>
<td>{a, c}</td>
<td>\text{dito}</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

- \( \bot, F \bot, F^2 \bot, \ldots \) 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 \( \mathbb{D} \) is finite, a solution can be found that is definitely the least solution.

**Question:** What, if \( \mathbb{D} \) is not finite?
Theorem

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

Application:

Assume $x_i \supseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n$ is a system of constraints where all $f_i : \mathcal{D}^n \to \mathcal{D}$ are monotonic.

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

\[
\begin{array}{c|c|c}
  f & f^k \perp & f^k \top \\
  \hline
  0 & \emptyset & U \\
\end{array}
\]
Example 1: \( \mathcal{D} = 2^U, \quad f(x) = x \cap a \cup b \)

<table>
<thead>
<tr>
<th></th>
<th>( f^k \bot )</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: \( \mathcal{D} = 2^U, \ f(x) = x \cap a \cup b \)

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

<table>
<thead>
<tr>
<th></th>
<th>( f^k \bot )</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 \textit{fixed-point iteration}, i.e., by repeated evaluation of right-hand sides.
Caveat: Naive fixed-point iteration is rather inefficient

Example:
Idea: Round Robin Iteration

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

Example:
<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<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>

Diagram:

- **Node 0**: x = 0
- **Node 1**: y = 1
- **Node 2**: true(y>0) -> 4, false(y>0) -> 3
- **Node 4**: y = y+x
- **Node 5**: x = x+(-1)
The code for Round Robin Iteration in Java looks as follows:

```java
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:

\[
\mathcal{I}[\text{start}] \sqsupseteq d_0 \\
\mathcal{I}[v] \sqsupseteq \ [k]^\# (\mathcal{I}[u]) \quad k = (u, -, v) \quad \text{edge}
\]

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

Wanted: MOP (Merge Over all Paths)

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

**Theorem** Kam, Ullman 1975

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

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

In particular: $\mathcal{I}[v] \sqsupseteq [\pi]^\# d_0$ \quad for every $\pi : \text{start} \rightarrow^* v$
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

$$f (\bigsqcup X) = \bigsqcup \{ f x \mid x \in X \} \text{ for all } \emptyset \neq X \subseteq D;$$

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

Theorem Kildall 1972

Assume all $v$ are reachable from $\text{start}$.

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:

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

\(\text{Neg (}x > 0)\)
\(\text{Pos (}x > 0)\)
\(M[A] = B;\)
Obviously, \( x \) has always the value 7
Thus, the memory access is always executed

Goal:

1. \( x = 7; \)
2. Neg \((x > 0)\)  
3. Pos \((x > 0)\)  
4. \( M[A] = B; \)  
5. ;
Obviously, \( x \) has always the value 7
Thus, the memory access is always executed

**Goal:**

\[
\begin{align*}
1 & \rightarrow 2 : x = 7; \\
2 & \rightarrow 3 : \\
3 & \rightarrow 4 : M[A] = B; \\
4 & \rightarrow 5 : \\
5 & \rightarrow 1 : \\
1 & \rightarrow 2 : \\
2 & \rightarrow 3 : \\
3 & \rightarrow 4 : M[A] = B; \\
4 & \rightarrow 5 : \\
5 & \rightarrow 1 :
\end{align*}
\]
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 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:

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

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

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

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

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

Obviously, $[lab]^\# \bot = \bot$ 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 $\top$

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

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

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

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:

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

\[
\begin{align*}
\llbracket c \rrbracket^\sharp D &= c \\
\llbracket e_1 \square e_2 \rrbracket^\sharp D &= \llbracket e_1 \rrbracket^\sharp D \square^\sharp \llbracket e_2 \rrbracket^\sharp D
\end{align*}
\]

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

\[
\begin{align*}
\llbracket c \rrbracket_D & = c \\
\llbracket e_1 \mathbin{\square} e_2 \rrbracket_D & = \llbracket e_1 \rrbracket_D \mathbin{\boxplus} \llbracket e_2 \rrbracket_D \\
\end{align*}
\]

... analogously for unary operators

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

\[
\begin{align*}
\llbracket x + 7 \rrbracket_D & = \llbracket x \rrbracket_D \mathbin{\boxplus} \llbracket 7 \rrbracket_D \\
& = 2 \mathbin{\boxplus} 7 \\
& = 9 \\
\llbracket x - y \rrbracket_D & = 2 \mathbin{\boxminus} \top \\
& = \top
\end{align*}
\]
Thus, we obtain the following abstract edge effects \([lab]^\#:\):

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

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

Example:
At \textit{start}, we have \( D_{\top} = \{ x \mapsto \top \mid x \in \text{Vars} \} \).

Example:

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

\[
[\pi]^\# = [k_r]^\# \circ \ldots \circ [k_1]^\# : \mathbb{D} \rightarrow \mathbb{D}
\]
Idea for Correctness: Abstract Interpretation
Cousot, Cousot 1977

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

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

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: \[ \Delta \subseteq \mathbb{Z} \times \mathbb{Z}^\top \]

\[ z \Delta a \iff z = a \lor a = \top \]

Concretization:

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

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

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

Concretization:

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

(3) States:

\[
\Delta \subseteq ((\text{Vars} \rightarrow \mathbb{Z}) \times (\mathbb{N} \rightarrow \mathbb{Z})) \times (\text{Vars} \rightarrow \mathbb{Z}^\top) \perp
\]

\( (\rho, \mu) \triangle D \) \iff \( \rho \triangle D \)

Concretization:

\[
\gamma D = \begin{cases} 
\emptyset & \text{if } D = \perp \\
\{(\rho, \mu) \mid \forall x : (\rho x) \triangle (D x)\} & \text{otherwise}
\end{cases}
\]
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:

$$\left[\pi\right] s \in \gamma\left(\left[\pi\right]^D\right)$$
The abstract semantics simulates the concrete semantics

In particular:

\[ [\pi] s \in \gamma ([\pi] \# D) \]

In practice, this means for example that \( Dx = -7 \) 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:

\[ D^*\![v] = \bigcup \{\pi\# \mid D_\top x : \pi : \text{start} \rightarrow^* v \} \]

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

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

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

```
\begin{align*}
M[R] &= y; \\
\text{Pos}(x > 1) &\quad y = x \ast y; \\
\text{Neg}(x > 1) &
\end{align*}
```

```
\begin{align*}
x &= 10; \\
y &= 1;
\end{align*}
```

```
\begin{array}{c|c|c}
\text{Pos}(x > 1) & 10 & 10 \\
\text{Neg}(x > 1) & 10 & 1 \\
\text{Pos}(x = 10) & 10 & 10 \\
\text{Neg}(x = 10) & 9 & 10 \\
\text{Pos}(x = 9) & 9 & 10 \\
\text{Neg}(x = 9) & 9 & 10 \\
\end{array}
```
Example:

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

\[ y = x \ast y; \]
\[ x = x - 1; \]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( x )</td>
<td>( y )</td>
</tr>
<tr>
<td>0</td>
<td>( \top )</td>
<td>( \top )</td>
</tr>
<tr>
<td>1</td>
<td>( 10 )</td>
<td>( \top )</td>
</tr>
<tr>
<td>2</td>
<td>( 10 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>3</td>
<td>( 10 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>4</td>
<td>( 10 )</td>
<td>( 10 )</td>
</tr>
<tr>
<td>5</td>
<td>( 9 )</td>
<td>( 10 )</td>
</tr>
<tr>
<td>6</td>
<td>( \bot )</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>( \bot )</td>
<td></td>
</tr>
</tbody>
</table>
Example:

\[ x = 10; \]

\[ y = 1; \]

\[ M[R] = y; \]

\[ y = x \times y; \]

\[ x = x - 1; \]

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

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

\[
\begin{array}{|c|c|c|c|}
\hline
 & 1 & 2 & 3 \\
\hline
 x & y & x & y & x & y \\
\hline
 0 & T & T & T & T & \\
 1 & 10 & T & 10 & T & \\
 2 & 10 & 1 & T & T & \\
 3 & 10 & 1 & T & T & \\
 4 & 10 & 10 & T & T & \\
 5 & 9 & 10 & T & T & \\
 6 & \bot & T & T & dito & \\
 7 & \bot & T & T & & \\
\hline
\end{array}
\]
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: \[ f = \llbracket x = x + y; \rrbracket \# \]

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

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

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

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

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

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:

```plaintext
for (i = 0; i < 42; i++)
    if (0 ≤ i ∧ i < 42){
        A_1 = A + i;
        M[A_1] = i;
    }

A start address of an array
if-statement does array-bound 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:

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

\[ l_1 \quad l_2 \]
\[ \text{---} \quad \text{---} \]
\[ u_1 \quad u_2 \]
Thus:

\[[l_1, u_1] \sqcup [l_2, u_2] = [l_1 \cap l_2, u_1 \sqcup u_2]\]
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.,

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

→ \( I \) is not a complete lattice,
→ \( I \) has infinite ascending chains, e.g.,

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

Description Relation:

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

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:

\[ [l_1, u_1] +_\# [l_2, u_2] = [l_1 + l_2, u_1 + u_2] \]
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:

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

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

Example:

\[
\begin{align*}
[42, 42] & ==^* [42, 42] = \text{true} \\
[0, 7] & ==^* [0, 7] = \top \\
[1, 2] & ==^* [3, 4] = \text{false}
\end{align*}
\]
Less:

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

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

Example:

\[
\begin{align*}
[1, 2] <^\# [9, 42] &= \text{true} \\
[0, 7] <^\# [0, 7] &= \top \\
[3, 4] <^\# [1, 2] &= \text{false}
\end{align*}
\]
By means of $\mathbb{I}$ we construct the complete lattice:

$$
\mathbb{D}_\mathbb{I} = (\text{Vars} \to \mathbb{I})_\bot
$$

Description Relation:

$$
\rho \Delta D \quad \text{iff} \quad D \neq \bot \land \forall x \in \text{Vars} : (\rho x) \Delta (D x)
$$

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

$$
(\llbracket e \rrbracket \rho) \Delta (\llbracket e \rrbracket^\Delta D) \quad \text{whenever} \quad \rho \Delta D
$$
The Effects of Edges:

\[
[[;]]^\# D = D \\
[[x = e;]]^\# D = D \oplus \{ x \mapsto [[e]]^\# D \} \\
[[x = M[e];]]^\# D = D \oplus \{ x \mapsto \top \} \\
[[M[e_1] = e_2;]]^\# D = D \\
[[\text{Pos}(e)]]^\# D = \begin{cases} 
\bot & \text{if } \text{false} = [[e]]^\# D \\
D & \text{otherwise}
\end{cases} \\
[[\text{Neg}(e)]]^\# D = \begin{cases} 
D & \text{if } \text{false} \sqsubseteq [[e]]^\# D \\
\bot & \text{otherwise}
\end{cases}
\]

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

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

where:

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

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

where:

\[
D_1 = \begin{cases} 
D \oplus \{x \mapsto (D x) \cap ([e_1]^\# D)\} & \text{if } e \equiv x \not= e_1 \\
D \oplus \{x \mapsto (D x) \cap [-\infty, u]\} & \text{if } e \equiv x > e_1, [e_1]^\# D = [\_, u] \\
D \oplus \{x \mapsto (D x) \cap [l, \infty]\} & \text{if } e \equiv x < e_1, [e_1]^\# D = [l, \_] 
\end{cases}
\]
Example:

\begin{align*}
&i = 0; \\
&i = i + 1; \\
&A_1 = A + i; \\
&M[A_1] = i;
\end{align*}

\begin{table}[h]
\begin{tabular}{|c|c|c|}
\hline
\textbf{i} & \textbf{l} & \textbf{u} \\
\hline
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 \\
\hline
\end{tabular}
\end{table}
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 \( x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \) denote a system of constraints over \( \mathbb{D} \).

Define an accumulating iteration:

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

(2)

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., \( x \sqsubseteq Gx \) for all \( x \in \mathbb{D}^n \).
(c) The sequence \( G^k \downarrow, \quad k \geq 0, \) is an ascending chain:
\[
\downarrow \subseteq G \downarrow \subseteq \ldots \subseteq G^k \downarrow \subseteq \ldots
\]

(d) If \( G^k \downarrow = G^{k+1} \downarrow = y \), 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:
\[
x_i = x_i \sqcup f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n
\]
(3)

for a binary operation widening:
\[
\sqcup : \mathbb{D}^2 \rightarrow \mathbb{D} \quad \text{with} \quad v_1 \sqcup v_2 \sqsubseteq v_1 \sqcup v_2
\]

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

- The complete lattice is: \( \mathbb{D}_\Pi = (\text{Vars} \to \Pi \downarrow) \)
- the widening \( \sqcup \) is defined by:

\[
\bot \sqcup D = D \sqcup \bot = D
\]

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

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

where

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

with

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

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

\[\Rightarrow \quad \sqcup \quad \text{is not commutative} \]
Example:

\[ [0, 2] \uplus [1, 2] = [0, 2] \]
\[ [1, 2] \uplus [0, 2] = [-\infty, 2] \]
\[ [1, 5] \uplus [3, 7] = [1, +\infty] \]

→ 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
  
  $\rightarrow$ the abstract values do not get too complicated;
  
  $\rightarrow$ the number of updates remains bounded ...
Our Example:

\[ i = 0; \]

\[ \text{Neg}(i < 42) \]

\[ \text{Pos}(i < 42) \]

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

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

\[ A_1 = A + i; \]

\[ M[A_1] = i; \]

\[ i = i + 1; \]

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

```
0

\( i = 0; \)

1

Neg(\( i < 42 \))

Pos(\( i < 42 \))

Neg(0 \( \leq i < 42 \))

2

Pos(0 \( \leq i < 42 \))

3

\( A_1 = A + i; \)

4

\( M[A_1] = i; \)

5

\( i = i + 1; \)

6

```

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th></th>
<th>2</th>
<th></th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>l</td>
<td>-( \infty )</td>
<td>+( \infty )</td>
<td>-( \infty )</td>
<td>+( \infty )</td>
<td></td>
</tr>
<tr>
<td>u</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>+( \infty )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>+( \infty )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>+( \infty )</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>+( \infty )</td>
<td>dito</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>+( \infty )</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>+( \infty )</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>42</td>
<td>+( \infty )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>42</td>
<td>+( \infty )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>