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*

```
x = 0
y = 1
true(y>0) false(y>0)
y = y+x
x = x+(-1)
```
What are the ingredients that we need?
More ingredients?
All the ingredients:

- a set of information elements, each a set of possible signs,
- a partial order, “$\sqsubseteq$”, on these elements, specifying the ”relative strength” of two information elements,
- these together form the abstract domain, a lattice,
- functions describing how signs of variables change by the execution of a statement, abstract edge effects,
- these need an abstract arithmetic, an arithmetic on signs.
We construct the abstract domain for single variables starting with the lattice \( \text{Signs} = 2\{-,0,\} \) with the relation “\( \subseteq \)” = “\( \supseteq \)".
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 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 \( \text{Signs} \).

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?
How is a solution found?

Iterating until a fixed-point is reached

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
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 \(\square\) working on values by abstract operators \(\square^\#\) 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:

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

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

\[
\begin{align*}
[x + 7]^D & = \left[ x \right]^D +^D \left[ 7 \right]^D \\
& = \{ + \} +^D \{ + \} \\
& = \{ + \}
\end{align*}
\]

\[
\begin{align*}
[x + (-y)]^D & = \{ + \} +^D (-^D \left[ y \right]^D) \\
& = \{ + \} +^D (-^D \{ + \}) \\
& = \{ + \} +^D \{ - \} \\
& = \{ +, -, 0 \}
\end{align*}
\]
[\textit{lab}]^\# is the abstract edge effects associated with edge $k$.

It depends only on the label \textit{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 $D \neq \bot$

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 $v$:

$\rightarrow$ 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:

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

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

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

Example:
\[ S[0] \supseteq \top \]
\[ S[1] \supseteq S[0] \oplus \{ x \mapsto \{0\} \} \]
\[ S[2] \supseteq S[1] \oplus \{ y \mapsto \{+\} \} \]
\[ S[2] \supseteq S[5] \oplus \{ x \mapsto \left[ x + (-1) \right] \} \]
\[ S[5] \supseteq S[4] \oplus \{ y \mapsto \left[ y + x \right] \} \]
Background An Operational Semantics

Programs are represented as control-flow graphs.

Example:

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

\[
\begin{align*}
A_1 &= A_0 + 1 \times i; \\
R_1 &= M[A_1]; \\
A_2 &= A_0 + 1 \times j; \\
R_2 &= M[A_2]; \\
\text{Neg (} R_1 > R_2 \text{)} &\quad \text{Pos (} R_1 > R_2 \text{)} \\
A_3 &= A_0 + 1 \times j; \\
\end{align*}
\]
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:

<table>
<thead>
<tr>
<th>$\rho : Vars \rightarrow \text{int}$</th>
<th>values of variables (contents of symbolic registers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu : \mathbb{N} \rightarrow \text{int}$</td>
<td>contents of memory</td>
</tr>
</tbody>
</table>

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

$$[k] = [lab]$$

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

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

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

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

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

// \ [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)\]
\[ \begin{align*}
[; ] (\rho, \mu) &= (\rho, \mu) \\
[\text{Pos} (e)] (\rho, \mu) &= (\rho, \mu) \quad \text{if } [e] \rho \neq 0 \\
[\text{Neg} (e)] (\rho, \mu) &= (\rho, \mu) \quad \text{if } [e] \rho = 0
\end{align*} \]

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

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

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

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

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

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

Example:

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

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

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

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

The result of the computation is \( \pi s = (\{k_m\} \circ \ldots \circ \{k_1\}) s \)
The approach:
A static analysis needs to collect correct and hopefully precise information about a program in a terminating computation.

Concepts:

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

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

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

Intuition: $\sqsubseteq$ represents precision.

By convention: $a \sqsubseteq b$ means $a$ is more precise than $b$. 
Examples:

1. \( \mathbb{D} = 2^{\{a,b,c\}} \) with the relation “\( \subseteq \)”: 

![Diagram showing the relation between subsets of \( \{a,b,c\} \)]
2. The rules-of-sign analysis uses the following lattice with the relation “⊆”:

\[ \mathbb{D} = 2^{\{-,0,+\}} \]
3. $\mathbb{Z}$ with the relation “$\leq$”:

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

Caveat:

• has no upper bound!
• has the upper bounds
$d \in \mathbb{D}$ 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 \( D \) is a complete lattice (cl) if every subset \( X \subseteq D \) has a least upper bound \( \bigcup X \in D \).

**Note:**

Every complete lattice has

\[ \rightarrow \text{ a least element } \bot = \bigcup \emptyset \in D; \]

\[ \rightarrow \text{ a greatest element } \top = \bigcup D \in D. \]
Examples:

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

\[
\begin{array}{ccccccc}
& & & & \top & & \\
& & & \downarrow & & & \\
& & \bot & & & & \bot \\
& \downarrow & & \downarrow & & \downarrow & \\
-2 & -1 & 0 & 1 & 2 & \cdots \\
& \downarrow & \downarrow & \downarrow & \cdots & & \\
& \cdots & \cdots & \cdots & \cdots & \cdots & \\
\end{array}
\]
Theorem:

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

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

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

\[ S[v] \sqsubseteq \bigsqcup \{[[k]]^\#(S[u]) \mid k = (u, _, v) \text{ edge} \} \]

Correct because:

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

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

Relation to the running example:

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

\textbf{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 \( f_1 : \mathbb{D}_1 \rightarrow \mathbb{D}_2 \) 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 \( f_2 \circ f_1 : \mathbb{D}_1 \to \mathbb{D}_3 \)
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:

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

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

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

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:

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

(*)

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

Idea:

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

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

- If all \( f_i \) are monotonic, then also \( F \)
Wanted: least solution for

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

(*)

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

Idea:

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

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

\[
x_1 \supseteq \{a\} \cup x_3
\]

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

\[
x_3 \supseteq x_1 \cup \{c\}
\]

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

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

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: \( \mathbb{D} = 2\{a, b, c\}, \quad \subseteq = \subseteq \)

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

The Iteration:

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
 & 0 & 1 & 2 & 3 & 4 \\
\hline
x_1 & \emptyset & \{a\} & \{a, c\} &  &  \\
x_2 & \emptyset & \emptyset & \emptyset & \emptyset &  \\
x_3 & \emptyset & \{c\} & \{a, c\} &  &  \\
\hline
\end{array}
\]
Example: \( \mathcal{D} = 2\{a, b, c\}, \; \subseteq = \subseteq \)

\[
x_1 \supseteq \{a\} \cup x_3
\]

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

\[
x_3 \supseteq x_1 \cup \{c\}
\]

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

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

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>dito</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**

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

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

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

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

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

\[
\begin{array}{c|c|c}
  f & f^k \bot & f^k \top \\
  \hline
  0 & \emptyset & U \\
  1 & b & a \cup b \\
  2 & b & a \cup b \\
\end{array}
\]
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:

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

x = 0
y = 1
true(y>0)
false(y>0)
y = y+x
x = x+(-1)
Idea: Round Robin Iteration

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

Example:
\[ y = 1 \]
\[ x = 0 \]

- \[ y = y + x \]
- \[ x = x + (-1) \]
- If \( y > 0 \) then \[ \text{true} \]
- If \( y \leq 0 \) then \[ \text{false} \]

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

Diagram:

- Node 0: \( x = 0 \)
- Node 1: \( y = 1 \)
- Node 2: \[ \text{true}(y>0) \] or \[ \text{false}(y>0) \]
- 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 $\mathcal{D}$, consider systems:

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

where $d_0 \in \mathcal{D}$ and all $\mathcal{I}[u]^{k} : \mathcal{D} \rightarrow \mathcal{D}$ are monotonic ...

**Wanted:** MOP (Merge Over all Paths)

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

**Theorem** 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: $\mathcal{I}[v] \supseteq \mathcal{I}[u]^{\pi}d_0 \quad \text{for every} \quad \pi : \text{start} \rightarrow^{*} v$
Disappointment: Are solutions of the constraint system just upper bounds?

Answer: In general: yes

With the notable exception when all functions $[k]^#$ are distributive ...

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

If $f : \mathbb{D}_1 \to \mathbb{D}_2$ is distributive, then also monotonic
Assumption: all $v$ are reachable from $start$.

Then:

**Theorem**

Theorem Kildall 1972

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

→ The abstract edge effects in the analysis of availability of expressions are distributive:

\[
(a \cup (x_1 \cap x_2)) \setminus b = ((a \cup x_1) \cap (a \cup x_2)) \setminus b = ((a \cup x_1) \setminus b) \cap ((a \cup x_2) \setminus b)
\]

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

→ The abstract edge effects in the analysis of availability of expressions are distributive:

\[
(a \cup (x_1 \cap x_2)) \setminus b = ((a \cup x_1) \cap (a \cup x_2)) \setminus b
\]
\[
= ((a \cup x_1) \setminus b) \cap ((a \cup x_2) \setminus b)
\]

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

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