Organization
Lecture, Prof. Dr. Dirk Beyer
Feb 18, 2021, 13:00 – 15:30
Online, Zoom

Tutorial, Prof. Dr. Dirk Beyer
Feb 19, 2021, 10:00 – 12:00
Online, Zoom

Übung, Thomas Lemberger
Feb 19, 2021, 13:00 – 15:30
Online, Zoom
Course Material

https://www.sosy-lab.org/Teaching/2021-SS-Semantik/

Required software (for tomorrow):

- Java 11
- CPAchecker 1.7.1M
- Python \( \geq 3.8 \)
- pip (usually comes with python)
Introduction
Computes an (over-)approximation of a program’s behavior.

Applications

- Optimization
- Correctness (i.e., whether program satisfies a given property)
Formally proves whether a program $P$ satisfies a property $\varphi$.

- Requires program semantics, i.e., meaning of program
- Relies on mathematical methods,
  - logic
  - induction
  - ...
Software Verification

**Formally** proves whether a program $P$ satisfies a property $\varphi$.

Disprove ($\times$) Find a program execution (counterexample) that violates the property $\varphi$

Prove ($\checkmark$) Show that every execution of the program satisfies the property $\varphi$. 
What Could an Analysis Find out?

double divTwiceCons(double y) {
    int cons = 5;
    int d = 2*cons;
    if (cons != 0)
        return y/(2*cons);
    else
        return 0;
}

double divTwiceCons(double y) {
    int cons = 5;
    // expression 2*cons has value 10
    // variable d not used
    int d = 2*cons;
    if (cons != 0)
        // expression 2*cons evaluated before
        return y/(2*cons);
    else
        // dead code
        return 0;
}
double divTwiceCons(double y) {
    int cons = 5;
    // expression 2*cons has value 10
    // variable d not used
    int d = 2*cons;
    if (cons != 0)
        // expression 2*cons evaluated before
        return y/(2*cons);
    else
        // dead code
        return 0;
}

double divTwiceConsOptimized(double y) {
    return y/10;
}
double avgUpTo(int[] numbers, int length) {
    double sum = 0;
    for (int i = 0; i < length; i++)
        sum += numbers[i];
    return sum / (double) length;
}
Problems With This Code

double avgUpTo(int[] numbers, int length) {
    double sum = 0;
    for (int i = 0; i < length; i++)
        // possible null pointer access (numbers==null)
        // index out of bounds (length>numbers.length)
        sum += numbers[i];
    // division by zero (length==0)
    return sum/(double) length;
}
Why Should One Care for Bugs?

Costs

Ariane V88

Intel Pentium FDIV bug

Mars Polar Lander

endanger human lives

Safety-criticality

Therac-25

Uber autonomous car
## Analysis and Verification Tools

<table>
<thead>
<tr>
<th>Sapienz</th>
<th>Klee</th>
<th>PeX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infer</td>
<td>Lint</td>
<td>Error Prone</td>
</tr>
<tr>
<td>CBMC</td>
<td>SpotBugs</td>
<td>UltimateAutomizer</td>
</tr>
<tr>
<td>CPAchecker</td>
<td>⋮</td>
<td>⋮</td>
</tr>
</tbody>
</table>
Overview on Analysis and Verification Techniques

Dynamic
- Testing
- Runtime Verification

Type Systems
- Interactive
  - Theorem Proving
- Automatic
  - Model Checking
  - Program Analysis
  - Abstract Interpretation
  - Static

Static
- Dynamic
- Runtime Verification

This lecture
Why Different Static, Automatic Techniques?

Theorem of Rice
Any non-trivial, semantic property of programs is undecidable.

Consequences
Techniques are

- incomplete, e.g. answer UNKNOWN, or
- unsound, i.e., report
  - false alarms (non-existing bugs),
  - false proofs (miss bugs).
Verifier Design Space

Ideal verifier

Unreliable verifier

Program $P$ 

Property $\varphi$

Verifier

FALSE $\times$

TRUE $\checkmark$

false proof $\leftrightarrow$ correct

false alarm $\leftrightarrow$ violation

FALSE $\times$

TRUE $\checkmark$

unknown

true
Verifier Design Space

- **Overapproximating verifier (superset of program behavior)** without precise counterexample check

  Program $P$ \[\rightarrow\] Verifier \[\rightarrow\] Unknown

  Property $\phi$ \[\rightarrow\] FALSE \[\times\]

  False alarm $\leftrightarrow$ violation

- **Underapproximating verifier (subset of program behavior)**

  False proof $\leftrightarrow$ correct

  Program $P$ \[\rightarrow\] Verifier \[\rightarrow\] Unknown

  Property $\phi$ \[\rightarrow\] FALSE \[\times\] TRUE \[\checkmark\]
Other Reasons to Use Different Static Techniques

- State space grows exponentially with number of variables
- (Syntactic) paths grow exponentially with number of branches

⇒ Precise techniques may require too many resources (memory, time,...)
⇒ Trade-off between precision and costs
Flow-Insensitivity

Order of statements not considered

E.g., does not distinguish between these two programs

\begin{align*}
x &= 0; \\
y &= x; \\
x &= x + 1;
\end{align*}

\begin{align*}
x &= 0; \\
x &= x + 1; \\
y &= x;
\end{align*}

\Rightarrow \text{very imprecise}
Flow-Sensitivity Plus Path-Insensitivity

- Takes order of statements into account
- Mostly, ignores infeasibility of syntactical paths
- Ignores branch correlations

E.g., does not distinguish between these two programs

```plaintext
if (x>0)              if (x>0)              if (x>0)
  y=1;                y=1;                y=1;
else                   else                   else
  y=0;                y=0;                y=0;
if (x>0)              if (x>0)              if (x>0)
  y=y+1;              y=y+2;              y=y+2;
else                   else                   else
  y=y+2;              y=y+1;              y=y+1;
```
Path-Sensitivity

- Takes (execution) paths into account
- Excludes infeasible, syntactic paths (not necessarily all infeasible ones)
- Covers flow-sensitivity

```c
if (x>0)
  y=1;
else
  y=0;
if (x>0)
  y=y+2;
else
  y=y+1;
```

To detect that $y$ has value 0, 1, or 3
- must exclude infeasible, syntactic path along first else-branch and second if-branch
- need to detect correlation between the if-conditions
- requires path-sensitivity

⇒ very precise
Precision vs. Costs

Program Analysis

Dataflow Analysis

Abstract Interpretation

Flow-insensitive

Flow-sensitive

Path-sensitive

imprecise

precise

cheap

expensive
Program Syntax and Semantics
Theory: simple while-programs

- Restriction to integer constants and variables
- Minimal set of statements (assignment, if, while)
- Techniques easier to teach/understand

Practice: C programs

- Widely-used language
- Tool support
While-Programs

- Arithmetic expressions
  \[ aexpr := \mathbb{Z} \mid var \mid -aexpr \mid aexpr \; op_a \; aexpr \]
  \( op_a \) standard arithmetic operation like +, −, /, %, ...

- Boolean expressions
  \[ bexpr := aexpr \mid aexpr \; op_c \; aexpr \mid !bexpr \mid bexpr \; op_b \; bexpr \]
  \( op_c \) comparison operator like <, <=, >, >=, ==, !=
  \( op_b \) logic connective like && (\&\&), || (\lor), ^ (xor), ...

- Program
  \[ S := \text{var}=aexpr; \mid \text{while} \; bexpr \; S \mid \text{if} \; bexpr \; S \; \text{else} \; S \mid \text{if} \; bexpr \; S \; \text{else} \; S;S \]
Syntax vs. Semantics

Syntax
Representation of a program

Semantics
Meaning of a program
How to Represent a Program?

1. Source code

```c
if (x > 0)
    abs = x;
else
    abs = -x;
i = 1;
while (i < abs)
    i = 2 * i;
```

- Basically sequence of characters
- No explicit information about the structure or paths of programs
How to Represent a Program?

2. Abstract-syntax tree (AST)

- Hierarchical representation
- Flow, paths hard to detect
How to Represent a Program?

3. Control-flow graph

4. Control-flow automaton

\[ x > 0 \]

\[ x > 0 \]

\[ !(x > 0) \]

\[ i = 1; \]

\[ i = 2 \times i; \]
Control-Flow Automaton

Definition
A control-flow automaton (CFA) is a three-tuple \( P = (L, l_0, G) \) consisting of

- the set \( L \) of program locations (domain of program counter)
- the initial program location \( l_0 \in L \), and
- the control-flow edges \( G \subseteq L \times Ops \times L \).
Operations $Ops$

Two types

- Assumes (boolean expressions)
- Assignments ($var=aexpr;$)
From Source Code to Control-Flow Automaton

Assignment \texttt{var=expr;}

While-Statement \texttt{while (C) S}

If-Statement \texttt{if (C) S_{1} else S_{2}}

If-Statement \texttt{if (C) S}

Sequential Composition \texttt{S_{1}; S_{2}}

Prof. Dr. Dirk Beyer

© SoSy-Lab, LMU Munich, Germany
Semantics

Remember: defines meaning of programs

Different types

- Axiomatic semantics: based on pre- and postconditions, e.g. \{true\}x=0;\{x=0\}
- Denotational semantics: function from inputs to outputs
- Operational semantics (✓): defines execution of program
Operational Semantics

Defines program meaning by fixing program execution

- Transitions describe single execution steps
  - Level of assignment or assume
  - Change states
  - Evaluate semantics of expressions in a state

- Execution: sequence of transitions
Concrete States

Pair of program counter and data state \((C = L \times \Sigma)\)

- **Program counter**
  - Where am I?
  - Location in CFA
  - \(c(pc) = l\) refers to program counter of concrete state

- **Data state** \(\sigma : V \rightarrow \mathbb{Z}\)
  - Maps variables to values
  - \(c(d) = \sigma\) refers to data state of concrete state
Semantics of Arithmetic Expressions

Evaluation function $S_a : aexpr \times \Sigma \rightarrow \mathbb{Z}$

Defined recursively on structure

- const $\in \mathbb{Z}$: $S_a(const, \sigma) = const$
- variable var: $S_a(var, \sigma) = \sigma(var)$
- unary operation: $S_a(-t, \sigma) = -S_a(t, \sigma)$
- binary operation:
  $$S_a(t_1 \; op_a \; t_2, \sigma) = S_a(t_1, \sigma) \; op_a \; S_a(t_2, \sigma)$$
Evaluation function $S_b : bexpr \times \Sigma \rightarrow \{true, false\}$

Defined recursively on structure

- arithmetic expression:
  $$S_b(t, \sigma) = \begin{cases} true & \text{if } S_a(t, \sigma) \neq 0 \\ false & \text{else} \end{cases}$$

- comparison:
  $$S_b(t_1 \ op_c t_2, \sigma) = S_a(t_1, \sigma) \ op_c S_a(t_2, \sigma)$$

- logic connection:
  $$S_b(b_1 \ op_b b_2, \sigma) = S_b(b_1, \sigma) \ op_b S_b(b_2, \sigma)$$
Examples for Expression Evaluation

Consider $\sigma : \text{abs} \mapsto 2; i \mapsto 0; x \mapsto -2$

Derivation of the values of

$\mathcal{S}_a(-x, \sigma)$
$\mathcal{S}_a(2 * i, \sigma)$
$\mathcal{S}_b(x > 0, \sigma)$
$\mathcal{S}_b(i < \text{abs}, \sigma)$

on the board.
State Update

\[ \Sigma \times Ops_{\text{assignment}} \rightarrow \Sigma \]

\[ \sigma[\text{var} = \text{aexpr};] = \sigma' \]

with \( \sigma'(v) = \begin{cases} 
\sigma(v) & \text{if } v \neq \text{var} \\
S_a(\text{aexpr}, \sigma) & \text{else}
\end{cases} \)
Examples for State Update

Consider \( \sigma : \text{abs} \mapsto 2; \ i \mapsto 0; \ x \mapsto -2 \)

Computation of the state updates

\[
\begin{align*}
&\sigma[i = 1;] \\
&\sigma[\text{abs} = -x;] \\
&\sigma[i = 2 \ast i;]
\end{align*}
\]
on the board.
Transitions \( \mathcal{T} \subseteq C \times G \times C \) with \((c, (l, \text{op}, l'), c') \in \mathcal{T}\) if

1. Respects control-flow, i.e.,
   \[
   c(pc) = l \land c'(pc) = l'
   \]

2. Valid data behavior
   - \textit{op} assignment \texttt{var=aexpr;}
     
     \[
     \ldots \land c'(d) = c(d)[\texttt{var = aexpr;}]
     \]
   - \textit{op} assume \texttt{bexpr}
     
     \[
     \ldots \land S_b(\texttt{bexpr}, c(d)) = \text{true} \land c(d) = c'(d)
     \]
Program Paths

Defined inductively

- every concrete state $c$ with $c(pc) = l_0$ is a program path
- if $c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n$ is a program path and $(c_n, g_{n+1}, c_{n+1}) \in \mathcal{T}$, then $c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n \xrightarrow{g_{n+1}} c_{n+1}$ is a program path

Set of all program paths of program $P = (L, G, l_0)$ denoted by $\text{paths}(P)$. 
Examples for Program Paths

On the board: Shortest and longest program path starting in state \((l_0, \sigma)\) with \(\sigma : \text{abs} \mapsto 2; \text{i} \mapsto 0; \text{x} \mapsto -2\)
Reachable States

\[ \text{reach}(P) := \{ c \mid \exists c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n \in \text{paths}(P) : c_n = c \} \]
Program Properties and Program Correctness
Program Properties

- Trace Property
  - Safety
    - Reachability
    - Type State
  - Liveness
    - Termination
    - Responsiveness
- Hyper Property
  - Information-Flow Security
Reachability Property $\varphi_R$

Defines which concrete states $\varphi_R \subseteq C$ must not be reached

In this lecture:

- Certain program locations must not be reached
- Denoted by $\varphi_{L_{\text{sub}}} := \{ c \in C \mid c(pc) \in L_{\text{sub}} \}$
Correctness

Definition
Program $P$ is correct wrt. reachability property $\varphi_R$ if

\[ \text{reach}(P) \cap \varphi_R = \emptyset. \]
Formalizing Verification Terms

- **False alarm**: \( v(P, \varphi_R) = \text{FALSE} \land \text{reach}(P) \cap \varphi_R = \emptyset \)
- **False proof**: \( v(P, \varphi_R) = \text{TRUE} \land \text{reach}(P) \cap \varphi_R \neq \emptyset \)
- **Verifier** \( v \) is **sound** if \( v \) does not produce false proofs and \( v \) is **complete** if \( v \) does not produce false alarms.
Abstract Domains
Problem With Program Semantics

- Infinitely many data states $\sigma$
  $\Rightarrow$ infinitely many reachable states

- Cannot analyze program paths individually
How to deal with infinite state space?

Idea: analyze set of program paths together

- Group concrete states ⇒ abstract states
- Define (abstract) semantics for abstract states

⇒ Abstract domain
Definition
Let $E$ be a set and $\subseteq \subseteq E \times E$ a binary relation on $E$. The structure $(E, \subseteq)$ is a partial order if $\subseteq$ is

- reflexive $\forall e \in E : e \subseteq e$,
- transitive $\forall e_1, e_2, e_3 \in E : (e_1 \subseteq e_2 \land e_2 \subseteq e_3) \implies e_1 \subseteq e_3$,
- antisymmetric $\forall e_1, e_2 \in E : (e_1 \subseteq e_2 \land e_2 \subseteq e_1) \implies e_1 = e_2$. 
Examples for Partial Orders

- $(\mathbb{Z}, \leq)$
- $(2^Q, \subseteq)$
- $(\Sigma^*, \text{lexicographic order})$
- $(\Sigma^*, \text{suffix})$
Let \((E, \sqsubseteq)\) be a partial order.

**Definition (Chain)**

A subset \(E_{\text{sub}} \subseteq E\) is a chain if it is totally ordered, i.e.

\[
\forall e, e' \in E_{\text{sub}} : e \sqsubseteq e' \lor e' \sqsubseteq e.
\]

A chain \(E_{\text{sub}}\) is finite if the subset \(E_{\text{sub}}\) is finite.
Ascending Chains

Let \((E, \sqsubseteq)\) be a partial order.

**Definition (Ascending Chain)**

A sequence \((e_i)_{n \in \mathbb{N}} \in E^\omega\) is an ascending chain if \(\forall m, m' \in \mathbb{N}: m \leq m' \Rightarrow e_m \sqsubseteq e_{m'}\).

**Definition (Stabilization)**

A sequence \((e_i)_{n \in \mathbb{N}} \in E^\omega\) eventually stabilizes if \(\exists n_0 \in \mathbb{N}: \forall n \in \mathbb{N}: n \geq n_0 : e_n = e_{n_0}\)

**Definition (Stabilizing Ascending Chain)**

A stabilizing ascending chain eventually stabilizes.
Examples for Chains

Consider \((\mathbb{Z}, =)\)

- Set \(\{1,2\}\) not a chain
- \((a_1, a_2, \ldots)\) with \(a_i = 1\) ascending and stabilizing
- Is a stabilizing ascending chain.

Consider \((\mathbb{Z}, \leq)\)

- Every subset of \(\mathbb{Z}\) is a chain.
- \((a_1, a_2, \ldots)\) with \(a_i = \begin{cases} 0 & \text{if } i \text{ even} \\ 1 & \text{else} \end{cases}\) not ascending
- \((a_1, a_2, \ldots)\) with \(a_i = i\) ascending, but not stabilizing
- \((a_1, a_2, \ldots)\) with \(a_i = \min(i, 10)\) ascending and stabilizing
- Is not a stabilizing ascending chain.
Height of Partial Order

Let \((E, \sqsubseteq)\) be a partial order.

- \((E, \sqsubseteq)\) has finite height if all chains are finite.
- \((E, \sqsubseteq)\) has height \(h\) if all chains contain at most \(h + 1\) elements and one chain contains \(h + 1\) elements.

**Note:** If \(E\) is finite than \((E, \sqsubseteq)\) has finite height, but not vice versa.

For example, \((\mathbb{Z}, =)\)
Heights of Example Partial Orders

<table>
<thead>
<tr>
<th>PO</th>
<th>finite height</th>
<th>height</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\mathbb{Z}, \leq))</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>((\mathbb{Z}, \geq))</td>
<td>✓</td>
<td>0</td>
</tr>
<tr>
<td>((\mathbb{Z}, =))</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>((2^Q, \subseteq), Q \text{ finite})</td>
<td>✓</td>
<td>(</td>
</tr>
<tr>
<td>((\Sigma^*, \text{lexicographic order}))</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>((\Sigma^*, \text{suffix}))</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
## Heights of Example Partial Orders

<table>
<thead>
<tr>
<th>PO</th>
<th>finite height</th>
<th>height</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\mathbb{Z}, \leq))</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>((\mathbb{Z}, \geq))</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>((\mathbb{Z}, =))</td>
<td>✓</td>
<td>0</td>
</tr>
<tr>
<td>((2^Q, \subseteq), Q \text{ finite})</td>
<td>✓</td>
<td>(</td>
</tr>
<tr>
<td>((\Sigma^*, \text{ lexicographic order}))</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>((\Sigma^*, \text{ suffix}))</td>
<td>✗</td>
<td></td>
</tr>
</tbody>
</table>
Upper Bound (Join)

Let \((E, \sqsubseteq)\) be a partial order.

**Definition (Upper Bound)**

An element \(e \in E\) is an upper bound of a subset \(E_{sub} \subseteq E\) if

\[
\forall e' \in E_{sub} : e' \sqsubseteq e.
\]

**Definition (Least Upper Bound (lub))**

An element \(e \in E\) is a least upper bound \(\sqcup\) of a subset \(E_{sub} \subseteq E\) if

- \(e\) is an upper bound of \(E_{sub}\) and
- for all upper bounds \(e'\) of \(E_{sub}\) it yields that \(e \sqsubseteq e'\).
Lower Bound (Meet)

Let \((E, \sqsubseteq)\) be a partial order.

**Definition (Lower Bound)**

An element \(e \in E\) is a lower bound of a subset \(E_{\text{sub}} \subseteq E\) if

\[
\forall e' \in E_{\text{sub}} : e \sqsubseteq e'.
\]

**Definition (Greatest Lower Bound (glb))**

An element \(e \in E\) is a greatest lower bound \(\sqcap\) of a subset \(E_{\text{sub}} \subseteq E\) if

- \(e\) is a lower bound of \(E_{\text{sub}}\) and
- for all lower bounds \(e'\) of \(E_{\text{sub}}\) it yields that \(e' \sqsubseteq e\).
Computing Upper Bounds

<table>
<thead>
<tr>
<th>PO</th>
<th>subset</th>
<th>(\square) (lub)</th>
<th>(\sqcap) (glb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\mathbb{Z}, \leq))</td>
<td>{1, 4, 7}</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>((\mathbb{Z}, \leq))</td>
<td>(\mathbb{Z})</td>
<td>(\times)</td>
<td>(\times)</td>
</tr>
<tr>
<td>((\mathbb{N}, \leq))</td>
<td>(\emptyset)</td>
<td>0</td>
<td>(\times)</td>
</tr>
<tr>
<td>((2^\mathbb{Q}, \subseteq))</td>
<td>(2^\mathbb{Q})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((2^\mathbb{Q}, \subseteq))</td>
<td>({\emptyset})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((2^\mathbb{Q}, \subseteq))</td>
<td>(Y \subseteq 2^\mathbb{Q})</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Computing Upper Bounds

<table>
<thead>
<tr>
<th>PO</th>
<th>subset</th>
<th>□ (lub)</th>
<th>□ (glb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\mathbb{Z}, \leq))</td>
<td>{1, 4, 7}</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>((\mathbb{Z}, \leq))</td>
<td>(\mathbb{Z})</td>
<td>(\times)</td>
<td>(\times)</td>
</tr>
<tr>
<td>((\mathbb{N}, \leq))</td>
<td>(\emptyset)</td>
<td>0</td>
<td>(\times)</td>
</tr>
<tr>
<td>((2^\mathbb{Q}, \subseteq))</td>
<td>(2^\mathbb{Q})</td>
<td>(\mathbb{Q})</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>((2^\mathbb{Q}, \subseteq))</td>
<td>({\emptyset})</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>((2^\mathbb{Q}, \subseteq))</td>
<td>(Y \subseteq 2^\mathbb{Q})</td>
<td>(\bigcup_{y \in Y} y)</td>
<td>(\bigcap_{y \in Y} y)</td>
</tr>
</tbody>
</table>
Facts About Upper and Lower Bounds

1. Least upper bounds and greatest lower bound do not always exist.
   For example,
   - $(\mathbb{Z}, \leq)$
   - $(\mathbb{N}, \leq)$
   - $(\mathbb{N}, \geq)$

2. The least upper bound and the greatest lower bound are unique if they exist.
   (Proof on the board)
Lattice

Definition
A structure $\mathcal{E} = (E, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ is a lattice if

- $(E, \sqsubseteq)$ is a partial order
- least upper bound $\sqcup$ and greater lower bound $\sqcap$ exist for all subsets $E_{\text{sub}} \subseteq E$
- $\top = \sqcup E = \sqcap \emptyset$ and $\bot = \sqcap E = \sqcup \emptyset$

Note:
For any set $Q$ the structure $(2^Q, \subseteq, \cup, \cap, Q, \emptyset)$ is a lattice.
Which Partial Orders Are Lattices?

(a)          (b)

(c)          (d)

(e)          (f)
Flat-Lattice

**Definition**

A flat lattice of set $Q$ consists of

- Extended set $Q^\perp = Q \cup \{\top, \bot\}$
- Flat ordering $\sqsubseteq$, i.e. $\forall q \in Q$ : $\bot \sqsubseteq q \sqsubseteq \top$ and $\bot \sqsubseteq \top$

\[\begin{align*}
\sqcup &= \begin{cases} 
\bot & X = \emptyset \lor X = \{\bot\} \\
q & X = \{q\} \lor X = \{\bot, q\} \\
\top & \text{else}
\end{cases} \\
\sqcap &= \begin{cases} 
\top & X = \emptyset \lor X = \{\top\} \\
q & X = \{q\} \lor X = \{\top, q\} \\
\bot & \text{else}
\end{cases}
\end{align*}\]
Let $\mathcal{E}_1 = (E_1, \sqsubseteq_1, \sqcup_1, \sqcap_1, \top_1, \bot_1)$ and $\mathcal{E}_2 = (E_2, \sqsubseteq_2, \sqcup_2, \sqcap_2, \top_2, \bot_2)$ be lattices.

The product lattice $\mathcal{E}_\times = (E_1 \times E_2, \sqsubseteq_\times, \sqcup_\times, \sqcap_\times, \top_\times, \bot_\times)$ with

- $(e_1, e_2) \sqsubseteq_\times (e'_1, e'_2)$ if $e_1 \sqsubseteq_1 e'_1 \land e_2 \sqsubseteq_2 e'_2$
- $\sqcup_\times E_{\text{sub}} = (\sqcup_1 \{e_1 \mid (e_1, \cdot) \in E_{\text{sub}}\}, \sqcup_2 \{e_2 \mid (\cdot, e_2) \in E_{\text{sub}}\})$
- $\sqcap_\times E_{\text{sub}} = (\sqcap_1 \{e_1 \mid (e_1, \cdot) \in E_{\text{sub}}\}, \sqcap_2 \{e_2 \mid (\cdot, e_2) \in E_{\text{sub}}\})$
- $\top_\times = (\top_1, \top_2)$ and $\bot_\times = (\bot_1, \bot_2)$

is a lattice.

Proof on the board.
Join-Semi-Lattice

Complete lattice not always required
⇒ remove unused elements

Definition
Join-Semi-Lattice A structure $\mathcal{E} = (E, \subseteq, \sqcup, \top)$ is a lattice if

- $(E, \subseteq)$ is a partial order
- least upper bound $\sqcup$ exists for all subsets $E_{\text{sub}} \subseteq E$
- $\top = \sqcup E$
Abstract Domain

Join-semi-lattice on set of abstract states
+ meaning of abstract states

Definition
An abstract domain \( D = (C, \mathcal{E}, [\cdot]) \) consists of

- a set \( C \) of concrete states
- a join-semi-lattice \( \mathcal{E} = (E, \subseteq, \sqcup, \top) \)
- a concretization function \([\cdot] : E \rightarrow 2^C\) (assigns meaning of abstract states)

\begin{itemize}
  \item \([\top] = C\)
  \item \(\forall E_{\text{sub}} \subseteq E : \bigcup_{e \in E_{\text{sub}}} [e] \subseteq [\sqcup E_{\text{sub}}]\)
\end{itemize}

(join operator overapproximates)
Abstraction

\[ \alpha : 2^C \rightarrow E \]

Here:

- Not defined separately
- Returns smallest abstract state that covers set of concrete states
Abstraction and concretization function fulfill the following connection

1. \( \forall C'_{\text{sub}} \subseteq C : C_{\text{sub}} \subseteq [\alpha(C_{\text{sub}})] \)
   (abstraction safe approximation, but may loose information/precision)

2. \( \forall e \in E : \alpha([e]) \sqsubseteq e \)
   (no loss in safety)
Abstract Semantics

Abstract interpretation of program, i.e., evaluation on abstract states

Transfer relation $\leadsto \subseteq E \times G \times E$

$\forall e \in E, g \in G :$

$\bigcup_{c \in [e]} \{ c' \mid (c, g, c') \in T \} \subseteq \bigcup_{(e,g,e') \in \leadsto} [e']$

(safe over-approximation)

- Depends on abstract domain
- In this lecture: restricted to functions
Properties of Transfer Functions

▶ Monotony

\[ \forall e, e' \in E, g \in G : e \sqsubseteq e' \Rightarrow \leadsto(e, g) \sqsubseteq \leadsto(e', g) \]

▶ Distributivity (optional)

\[ \forall e, e' \in E, g \in G : \leadsto(e, g) \sqcup \leadsto(e', g) = \leadsto(e \sqcup e', g) \]
Elements of Abstraction (Recap.)

1. Abstract domain
   - Join-semi lattice $\mathcal{E}$ on set of abstract states $E$
   - Meaning of abstract states $[]$

2. Abstract semantics $\sim$
Properties of Abstraction (Recap.)

- Join operator overapproximates

\[ \forall E_{\text{sub}} \subseteq E : \bigcup_{e \in E_{\text{sub}}} [e] \subseteq [ \sqcup E_{\text{sub}}] \]

- Monotony of transfer relation

\[ \forall e, e' \in E, g \in G : e \sqsubseteq e' \Rightarrow \leadsto (e, g) \sqsubseteq \leadsto (e', g) \]

- Distributivity of transfer relation

\[ \forall e, e' \in E, g \in G : \leadsto (e, g) \sqcup \leadsto (e', g) = \leadsto (e \sqcup e', g) \]
Location Abstraction $\mathcal{L}$

Tracks control-flow of program

- Uses flat lattice of set $L$ of location states

$$ [[\ell]] := \begin{cases} C & \text{if } \ell = \top \\ \emptyset & \text{if } \ell = \bot \\ \{ c \in C \mid c(pc) = \ell \} & \text{else} \end{cases}$$

(guarantees that join overapproximates)

- $(\ell, (l, op, l'), \ell') \in \sim_{\mathcal{L}}$ if $(\ell = l \lor \ell = \top)$ and $\ell' = l'$
Properties of Location Abstraction

Transfer relation $\sim\rightarrow_{\mathcal{L}}$

- overapproximates, i.e.,

$$\forall e \in E_\mathcal{L}, g \in G : \bigcup_{c \in [e]} \{ c' \mid (c, g, c') \in \mathcal{T} \} \subseteq \bigcup_{(e, g, e') \in \sim\rightarrow_\mathcal{L}} [e']$$

- monotone
- distributive
Automaton observing violation of reachability property $\varphi_{L_{\text{sub}}}$

Property Encoding

$(\cdot, \cdot, l) \in G \land l \in L_{\text{sub}}$

$(\cdot, \cdot, l) \in G \land l \notin L_{\text{sub}}$
Represent automaton encoding of property $\varphi_{L_{\text{sub}}}$ as abstraction

- Uses join-semilattice on set $\{q_{\text{safe}}, q_{\text{unsafe}}\}$ with $q_{\text{safe}} \sqsubseteq q_{\text{unsafe}}$

$$[q] := \begin{cases} C & \text{if } q = q_{\text{unsafe}} \\ \{c \in C \mid c(pc) \notin L_{\text{sub}}\} & \text{else} \end{cases}$$

- $(q, (l, \text{op}, l'), q') \in \sim_{\mathbb{R}}$
  - if $q' = q_{\text{unsafe}} \land l' \in L_{\text{sub}}$ or $q' = q \land l' \notin L_{\text{sub}}$
Properties of Property Abstraction

Transfer relation \( \leadsto_R \)
- overapproximates
- monotone
- distributive
Assigns values to (some) variables.

- Domain elements are partial functions $f : \text{Var} \rightarrow \mathbb{Z}$
- $f \sqsubseteq f'$ if $\text{dom}(f') \subseteq \text{dom}(f)$ and $\forall v \in \text{dom}(f') : f(v) = f'(v)$
- $\bigcup F = \bigcap F$
- $\top = \{\}$
- $\llbracket f \rrbracket = \{c | \forall v \in \text{dom}(f) : c(d)(v) = f(v)\}$
Value Abstraction $\bigvee$

Uses variable-separate domain

- Base domain flat lattice of $\mathbb{Z}$
- Abstract value $\top$ means any value
- Transfer relation

Notation: $\phi(expr, f) := expr \land \bigwedge_{v \in \text{dom}(f)} v = f(v)$

- Assume: $(f, (\cdot, expr, \cdot), f') \in \rightsquigarrow_V$ if $\phi(expr, f)$ is satisfiable and $(v, c) \in f'$ if $c$ is the only satisfying assignment for variable $v$ in $\phi(expr, f)$

- Assignment: $(f, (\cdot, w := expr; \cdot), f') \in \rightsquigarrow_V$ with $(v, c) \in f'$ if either
  - $v \neq w$ and $(v, c) \in f$, or
  - $v = w$ and $c$ is the only satisfying assignment for variable $v'$ in $\phi(v' = expr, f)$
Transfer relation

- overapproximates
- monotone
- not distributive, e.g.,

\[ f : x \mapsto 3; \ y \mapsto 2 \quad f' : x \mapsto 2; \ y \mapsto 3 \]
\[ \sim (f, \ x = x + y;) \sqcup \sim (f', \ x = x + y;) : x \mapsto 5; \ y \mapsto \top, \]

but \[ \sim (f \sqcup f', \ x = x + y;) : x \mapsto \top; \ y \mapsto \top \]
Example Abstract Transitions

On the board:

- \(\leadsto (i \mapsto \top; x \mapsto 3, (l, i = 1; , l'))\)
- \(\leadsto (i \mapsto \top; x \mapsto \top, (l, i = i * 2; , l'))\)
- \(\leadsto (i \mapsto \top; x \mapsto 5, (l, i = i * 2; , l'))\)
- \(\leadsto (i \mapsto 0; x \mapsto \top, (l, i && (x > 0), l'))\)
- \(\leadsto (i \mapsto \top; x \mapsto 10, (l, x > 10, l'))\)
Sign Abstraction

Variable-separate domain using base domain

\[
\begin{array}{c}
\top \\
0^+ \\
+ \\
\downarrow \\
\downarrow \\
\downarrow \\
0 \\
\downarrow \\
0^-
\end{array}
\]

\[
\begin{align*}
[\top] &= \mathbb{Z} \\
[+++] &= \mathbb{N}^+ \\
[-] &= \mathbb{Z} \setminus \mathbb{N}_0^+ \\
[0] &= \{0\} \\
[+-] &= \mathbb{Z} \setminus \{0\} \\
[0+] &= \mathbb{N}_0^+ \\
[-0] &= \mathbb{Z} \setminus \mathbb{N}^+ \\
[\bot] &= \emptyset
\end{align*}
\]
Transfer Relation of Sign Abstraction

Suggestion 1:

- $\sim (f, g) = f'$ with $\forall v \in Var : f'(v) = \top$
- sound, but not useful
Transfer Relation of Sign Abstraction

Suggestion 2:

- **Assume:** $\rightsquigarrow (f, expr) = f$
- **Assignment:** $\rightsquigarrow (f, expr) = f'$

\[
\begin{align*}
\text{v=} & \text{const}; \quad f'(v) = \begin{cases} 
+ & \text{const} \in \mathbb{N}^+ \\
0 & \text{const} = 0 \\
- & \text{else}
\end{cases} \\
\text{v=} & \text{w}; \quad f'(v) = f(w) \\
\text{v=} & \text{expr}; \quad f'(v) = \top
\end{align*}
\]

and $\forall u \in Var : u \neq v \Rightarrow f'(u) = f(u)$

sound, but could be more precise
Transfer Relation of Sign Abstraction (Incomplete)

More precise for special boolean expression like
\( \text{var} > 0, \text{var} == 0, \text{var} < 0, \text{var} >= 0, \text{var} <= 0 \)

- can be decided
- used to restrict successor of assume expressions

Abstract evaluation of arithmetic expressions, e.g.

- \( e + e = e \), for any abstract value \( e \) except \( ++- \)
- \( e + 0 = e \)
- \( e - 0 = e \)
- \( e * 0 = 0 \)
- \( \ldots \)
Interval Abstraction II

Variable-separate domain based on interval domain

- \( E = \mathbb{Z}^2 \cup \{\top, \bot\} \)

- \( \bot \sqsubseteq e, e \sqsubseteq \top \) and \( [a, b] \sqsubseteq [c, d] \) if \( c \leq a \land b \leq d \)

- \( \sqcup E_{\text{sub}} = \begin{cases} \top & \text{if } \top \in E_{\text{sub}} \\ \bot & \text{if } E_{\text{sub}} \subseteq \{\bot\} \\ \left[\min_{a,b} \in E_{\text{sub}} a, \max_{a,b} \in E_{\text{sub}} b\right] & \text{else} \end{cases} \)

- \( [[a, b]] = \{x \in \mathbb{Z} | a \leq x \leq b\} \) \( \top = \mathbb{Z} \) \( \bot = \emptyset \)

Note: There are ascending chains that are not stabilizing.
Transfer Relation of Interval Abstraction

Relies on abstract evaluation of expressions in state $f$

Arithmetic expressions

- **const**: $[\text{const, const}]$
- **var**: $f(\text{var})$
- $-[a,b] = [-b, -a]$
- $[a,b] \ op_a [c,d] = [\min(a \ op_a c, b \ op_a d), \max(a \ op_a c, b \ op_a d)]$
- special treatment of values $\bot, \top$
Transfer Relation of Interval Abstraction

Relies on abstract evaluation of expressions in state $f$

Boolean expression

- $[a, b] = \begin{cases} 
  \{true\} & a > 0 \lor b < 0 \\
  \{false\} & a = b = 0 \\
  \{true, false\} & \text{else}
\end{cases}$

- $[a, b] < [c, d] = \begin{cases} 
  \{true\} & b < c \\
  \{false\} & a \geq d \\
  \{true, false\} & \text{else}
\end{cases}$

- Other comparison operators similar

Define transfer relation analogous to transition
Cartesian Predicate Abstraction

Represent states by first order logic formulae

- Restricted to a set of predicates $\text{Pred}$
  (subset of boolean expressions without boolean connectors)
- Conjunction of predicates
Cartesian Predicate Abstraction

- **Power set lattice on predicates** $(2^{\text{Pred}}, \supseteq, \cap, \cup, \emptyset, \text{Pred})$

- $[[\top]] = [[\emptyset]] = C$
  for $p \neq \bot$:
  $[[p]] = \{c \in C \mid \forall \text{pred} \in p : S_b(\text{pred}, c(d)) = true\}$
  (guarantees that join overapproximates)

- **Transfer relation**
  - **Assignment**
    
    
    $(p, v = expr, p')$ with
    
    
    $p' = \{ t \in \text{Pred} \mid (\bigwedge_{t' \in p} t'[v \rightarrow v'] \land v = expr[v \rightarrow v']) \Rightarrow t \}$
  
  - **Assume**
    
    $(p, expr, p')$ if $\bigwedge_{t \in p} t \land expr$ is satisfiable and
    
    $p' = \{ t \in \text{Pred} \mid (\bigwedge_{t' \in p} t'[\land expr) \Rightarrow t \}$
Properties of Cartesian Predicate Abstraction

Transfer relation

- overapproximates
- monotone
- not distributive

(e.g., use value abstraction example and value assignments as predicates)
Example Abstract Transitions

Consider set of predicates \{i>0, \ x=10\}

On the board:

- $\leadsto (\{x = 10\}, (l, i = 1; l'))$
- $\leadsto (\{i > 0\}, (l, i = i \times 2; l'))$
- $\leadsto (\{i > 0\}, (l, i < \text{abs}; l'))$
- $\leadsto (\{x = 10, i > 0\}, (l, x > 10, l'))$
Composite Abstraction

Combines two abstractions

- Product (join-semi) lattice $E_1 \times E_2$

- $\llbracket (e_1, e_2) \rrbracket = \llbracket e_1 \rrbracket_1 \cap \llbracket e_2 \rrbracket_2$

- Product transfer relation

  $((e_1, e_2), g, (e'_1, e'_2)) \in \sim$ if $(e_1, g, e'_1) \in \sim_1$ and $(e_2, g, e'_2) \in \sim_2$

- More precise transfer relations possible
Properties of Composite Abstraction

Properties inherited from components

Transfer relation

▶ overapproximates
▶ monotone
▶ distributive

if respective property is fulfilled by both components.

Proof on the board
Two Prominent Combination

- Value analysis \( L \times V \times R \)
- Predicate analysis \( L \times P \times R \)