Semantics: Application to C Programs

Lecture

Prof. Dr. Dirk Beyer

Dirk.Beyer@sosy-lab.org

Slides and Material prepared by D. Beyer, M.-C. Jakobs, and M. Spießl

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

SoSy-Lab Software Systems

Organization

Lecture and Tutorial

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 >= 3.8
- pip (usually comes with python)

Introduction

Software Analysis

Computes an (over-)approximation of a program's **behavior**.

Applications

- Optimization
- Correctness
  (i.e., whether program satisfies a given property)

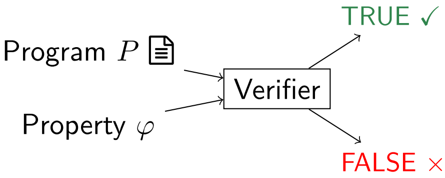
Software Verification

Formally proves whether a program P satisfies a property φ.

- 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 φ.



Disprove (×) Find a program execution (**counterexample**) that violates the property φ

Prove (✓) Show that **every** execution of the program satisfies the property φ.

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;
}
```

Some Analysis Results

```
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;
}
```

One Resulting Code Optimization

```
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;
}
```

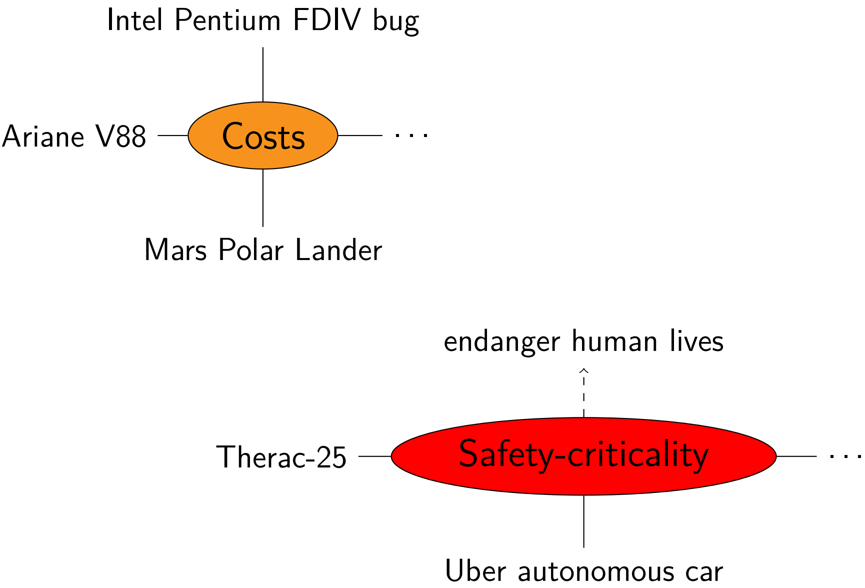
Does This Code Work?

```
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?



Analysis and Verification Tools

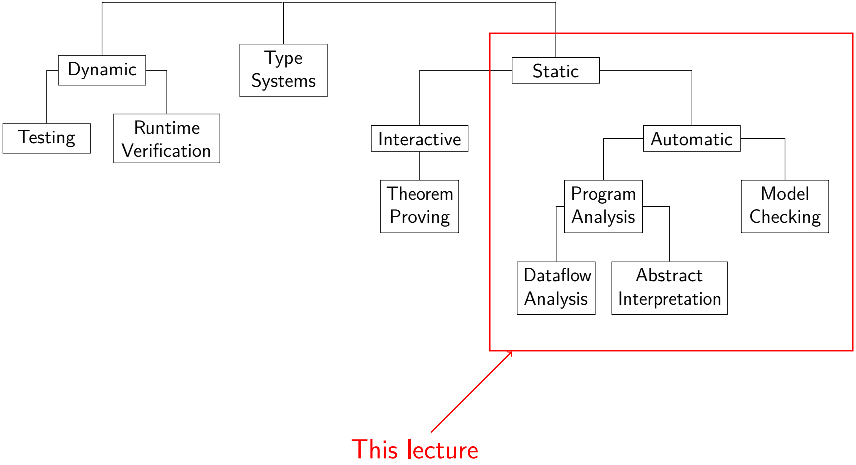
Sapienz Klee PeX

Infer Lint Error Prone SLAM

CBMC SpotBugs UltimateAutomizer

CPAchecker · · ·

Overview on Analysis and Verification Techniques



Why Different Static, Automatic Techniques?

Theorem of Rice

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

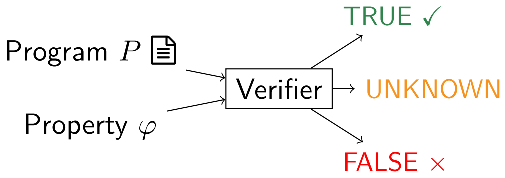
Consequences

Techniques are

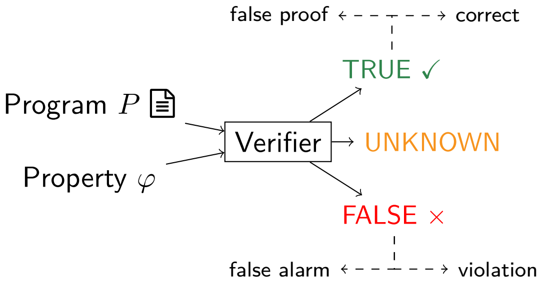
- incomplete, e.g. answer UNKNOWN, or
- unsound, i.e., report
  - false alarms (non-existing bugs),
  - false proofs (miss bugs).

Verifier Design Space

Ideal verifier

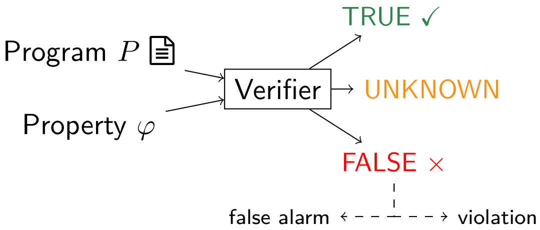


Unreliable verifier

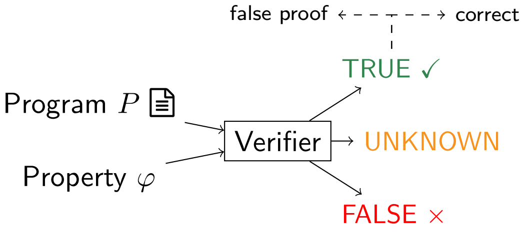


Verifier Design Space

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



- Underapproximating verifier (subset of program behavior)



Other Reasons to Use Different Static Techniques

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

$\Rightarrow$ Precise techniques may require too many resources (memory, time,...)

$\Rightarrow$ Trade-off between precision and costs

Flow-Insensitivity

Order of statements not considered

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

`x=0;`	`x=0;`
`y=x;`	`x=x+1;`
`x=x+1;`	`y=x;`

$\Rightarrow$ 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

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

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

Path-Sensitivity

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

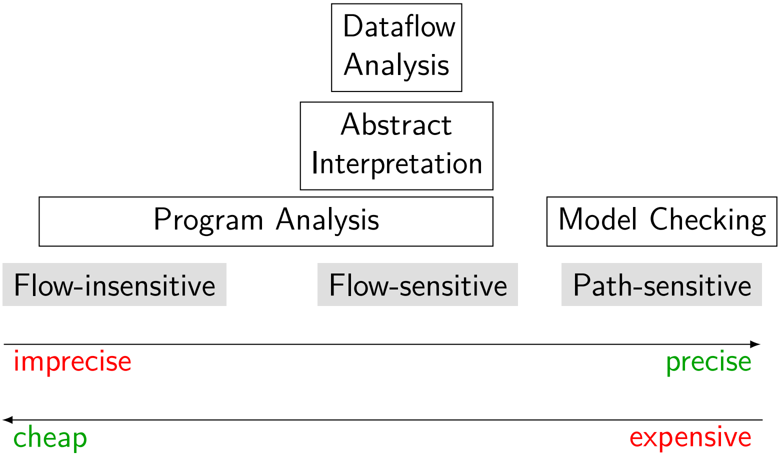
```
if (x>0)
    y=1;
else
    y=0;
if (x>0)
    y=y+2;
else
    y=y+1;
```

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

$\Rightarrow$ very precise

Precision vs. Costs



Program Syntax and Semantics

Programs

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 $:=$ $\mathbb{Z}$ | `var` | -aexpr | aexpr op_a aexpr
  op_a standard arithmetic operation like $+, -, /, \%, \ldots$
- Boolean expressions
  bexpr $:=$ aexpr | aexpr op_c aexpr | !bexpr | bexpr op_b bexpr
  - integer value 0 $\equiv$false, remaining values represent true
  - op_c comparison operator like $<, <=, >=, >, ==, ! =$
  - op_b logic connective like $\&\&(\wedge), || \ (\vee)$, ˆ (xor), $\ldots$
- Program
  S$:=$ `var=aexpr;` | `while` bexpr S | `if` bexpr S `else` S |
  `if` bexpr S | S;S

Syntax vs. Semantics

Syntax

Representation of a program

Semantics

Meaning of a program

How to Represent a Program?

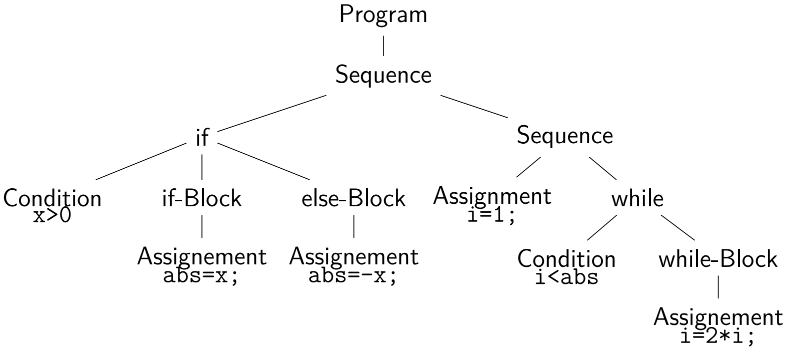
1. Source code

```
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?

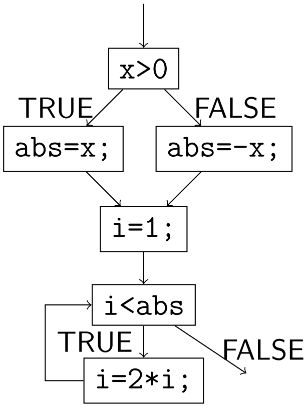
2. Abstract-syntax tree (AST)



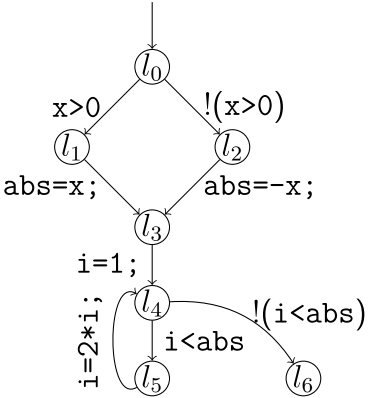
- Hierarchical representation
- Flow, paths hard to detect

How to Represent a Program?

3. Control-flow graph



4. Control-flow automaton



✓

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

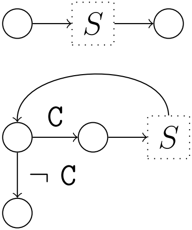
Two types

- Assumes (boolean expressions)
- Assignments (`var=aexpr;`)

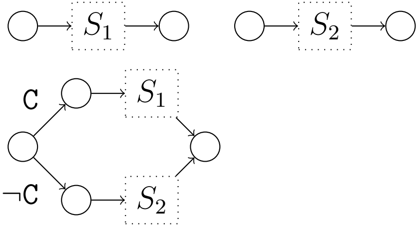
From Source Code to Control-Flow Automaton

Assignment `var=expr;`

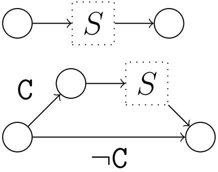
While-Statement `while` (C) S



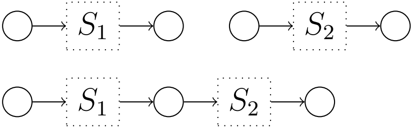
If-Statement `if` (C) S_1 `else` S_2



If-Statement `if` (C) S



Sequential Composition $S_1; S_2$



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 $\sigma : V \to \mathbb{Z}$
  - Maps variables to values
  - $c(d) = \sigma$ refers to data state of concrete state

Semantics of Arithmetic Expressions

Evaluation function $\mathcal{S}_a : aexpr \times \Sigma \rightarrow \mathbb{Z}$

Defined recursively on structure

- $\mathrm{const} \in \mathbb{Z} : \mathcal{S}_a(\mathrm{const}, \sigma) = \mathrm{const}$
- variable `var`: $\mathcal{S}_a(\mathtt{var}, \sigma) = \sigma(\mathtt{var})$
- unary operation: $\mathcal{S}_a(-t, \sigma) = -\mathcal{S}_a(t, \sigma)$
- binary operation:
  $\mathcal{S}_a(t_1 \ op_a \ t_2, \sigma) = \mathcal{S}_a(t_1, \sigma) \ op_a \ \mathcal{S}_a(t_2, \sigma)$

Semantics of Boolean Expressions

Evaluation function $\mathcal{S}_b : bexpr \times \Sigma \rightarrow \{true, false\}$

Defined recursively on structure

- arithmetic expression:
  $\mathcal{S}_b(t, \sigma) = \begin{cases} true & \text{if } \mathcal{S}_a(t, \sigma) \neq 0 \\ false & \text{else} \end{cases}$
- comparison: $\mathcal{S}_b(t_1\ op_c\ t_2, \sigma) = \mathcal{S}_a(t_1, \sigma)\ op_c\ \mathcal{S}_a(t_2, \sigma)$
- logic connection: $\mathcal{S}_b(b_1\ op_b\ b_2, \sigma) = \mathcal{S}_b(b_1, \sigma)\ op_b\ \mathcal{S}_b(b_2, \sigma)$

Examples for Expression Evaluation

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

Derivation of the values of

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

on the board.

State Update

$$\Sigma \times Ops_{\text{assignment}} \rightarrow \Sigma$$

$$\sigma[var = aexpr;] = \sigma'$$

with $\sigma'(v) = \begin{cases} \sigma(v) & \text{if } v \neq var \\ \mathcal{S}_a(aexpr, \sigma) & \text{else} \end{cases}$

Examples for State Update

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

Computation of the state updates

- $\sigma[i = 1;]$
- $\sigma[abs = -x;]$
- $\sigma[i = 2 * i;]$

on the board.

Transitions – Single Execution Steps

Transitions $\mathcal{T} \subseteq C \times G \times C$ with $(c, (l, op, l'), c') \in \mathcal{T}$ if

1. Respects control-flow, i.e.,

$$c(pc) = l \ \wedge \ c'(pc) = l'$$

2. Valid data behavior
   - op assignment `var=aexpr;` :
     ... $\wedge \ c'(d) = c(d)[var = aexpr;]$
   - op assume `bexpr` :
     ... $\wedge \ \mathcal{S}_b(\texttt{bexpr}, c(d)) = true \ \wedge \ c(d) = c'(d)$

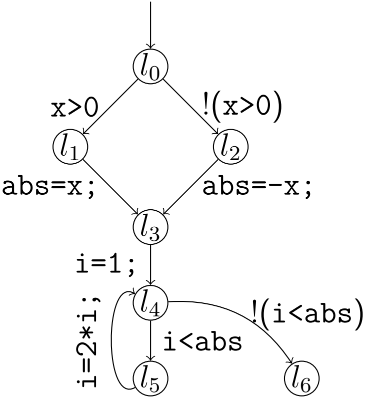
Program Paths

Defined inductively

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

Set of all program paths of program $P = (L, G, l_0)$ denoted by $paths(P)$.

Examples for Program Paths



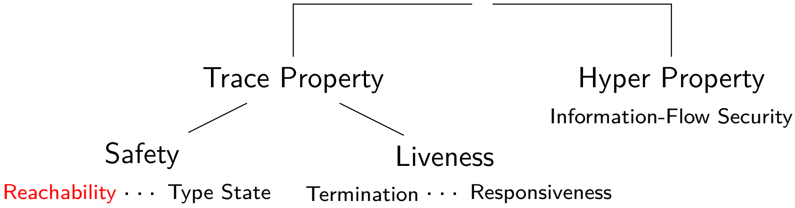
On the board: Shortest and longest program path starting in state (l_0, σ) with $\sigma : \texttt{abs} \mapsto 2; \texttt{i} \mapsto 0; \texttt{x} \mapsto -2$

Reachable States

$$reach(P) := \{c \mid \exists c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n \in paths(P) : c_n = c\}$$

Program Properties and Program Correctness

Program Properties



Reachability Property φ_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 φ_R if

$$reach(P) \cap \varphi_R = \emptyset.$$

Formalizing Verification Terms

- False alarm: $v(P, \varphi_R) = \text{FALSE} \wedge reach(P) \cap \varphi_R = \emptyset$
- False proof: $v(P, \varphi_R) = \text{TRUE} \wedge reach(P) \cap \varphi_R \neq \emptyset$
- 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 σ
  $\Rightarrow$ 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 $\Rightarrow$ abstract states
- Define (abstract) semantics for abstract states

$\Rightarrow$ Abstract domain

Partial Order (Recap)

Definition

Let E be a set and $\sqsubseteq \subseteq E \times E$ a binary relation on E. The structure $(E, \sqsubseteq)$ is a *partial order* if $\sqsubseteq$ is

- reflexive $\forall e \in E : e \sqsubseteq e$,
- transitive $\forall e_1, e_2, e_3 \in E : (e_1 \sqsubseteq e_2 \land e_2 \sqsubseteq e_3) \Rightarrow e_1 \sqsubseteq e_3$,
- antisymmetric
  $\forall e_1, e_2 \in E : (e_1 \sqsubseteq e_2 \land e_2 \sqsubseteq e_1) \Rightarrow e_1 = e_2$.

Examples for Partial Orders

- $(\mathbb{Z}, \leq)$
- $(2^Q, \subseteq)$
- $(\Sigma^*, \text{lexicographic order})$
- $(\Sigma^*, \text{suffix})$

Chains

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

Definition (Chain)

A subset $E_{\text{subset}} \subseteq E$ is a chain if it is totally ordered, i.e.
$\forall e, e' \in E_{\text{sub}} : e \sqsubseteq e' \vee e' \sqsubseteq e$.

A chain E_{subset} is finite if the subset E_{subset} 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
- $(a_1, a_2, \dots)$ with $a_i = 1$ ascending and stabilizing
- Is a stabilizing ascending chain.

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

- Every subset of $\mathbb{Z}$ is a chain.
- $(a_1, a_2, \dots)$ with $a_i = \begin{cases} 0 & \text{if } i \text{ even} \\ 1 & \text{else} \end{cases}$ not ascending
- $(a_1, a_2, \dots)$ with $a_i = i$ ascending, but not stabilizing
- $(a_1, a_2, \dots)$ 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

PO	finite height	height
$(\mathbb{Z}, \leq)$	$\times$	
$(\mathbb{Z}, \geq)$		
$(\mathbb{Z}, =)$	$\checkmark$	0
$(2^Q, \subseteq)$, Q finite	$\checkmark$	$\lvert Q \rvert$
$(\Sigma^*, \text{lexicographic order})$		
$(\Sigma^*, \text{suffix})$		

Heights of Example Partial Orders

PO	finite height	height
$(\mathbb{Z}, \leq)$	×	
$(\mathbb{Z}, \geq)$	×	
$(\mathbb{Z}, =)$	✓	0
$(2^Q, \subseteq)$, Q finite	✓	$\lvert Q\rvert$
$(\Sigma^*, \text{lexicographic order})$	×	
$(\Sigma^*, \text{suffix})$	×	

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 $E_{\text{sub}} \subseteq E$ if

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

Definition (Least Upper Bound (lub))

An element $e \in E$ is a least upper bound $\sqcup$ of a subset $E_{\text{sub}} \subseteq E$ 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 an 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_{sub} and
- for all lower bounds e' of E_{sub} it yields that $e' \sqsubseteq e$.

Computing Upper Bounds

PO	subset	$\sqcup$ (lub)	$\sqcap$ (glb)
$(\mathbb{Z}, \leq)$	$\{1, 4, 7\}$	7	1
$(\mathbb{Z}, \leq)$	$\mathbb{Z}$	$\times$	$\times$
$(\mathbb{N}, \leq)$	$\emptyset$	0	$\times$
$(2^Q, \subseteq)$	2^Q		
$(2^Q, \subseteq)$	$\{\emptyset\}$		
$(2^Q, \subseteq)$	$Y \subseteq 2^Q$		

Computing Upper Bounds

PO	subset	$\sqcup$ (lub)	$\sqcap$ (glb)
$(\mathbb{Z}, \leq)$	$\{1, 4, 7\}$	7	1
$(\mathbb{Z}, \leq)$	$\mathbb{Z}$	$\times$	$\times$
$(\mathbb{N}, \leq)$	$\emptyset$	0	$\times$
$(2^Q, \subseteq)$	2^Q	Q	$\emptyset$
$(2^Q, \subseteq)$	$\{\emptyset\}$	$\emptyset$	$\emptyset$
$(2^Q, \subseteq)$	$Y \subseteq 2^Q$	$\bigcup_{y \in Y} y$	$\bigcap_{y \in Y} y$

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 exists.
   (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 $\bot = \sqcap E = \sqcup\emptyset$

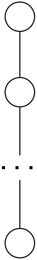
Note:

For any set Q the structure $(2^Q, \subseteq, \cup, \cap, Q, \emptyset)$ is a lattice.

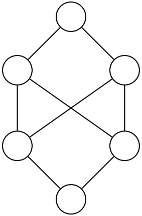
Which Partial Orders Are Lattices?

(a)

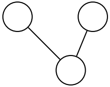
(b)



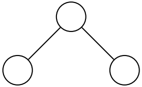
(c)



(d)



(e)



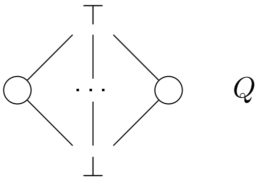
(f)

Flat-Lattice

Definition

A flat lattice of set Q consists of

- Extended set $Q_\bot^\top = Q \cup \{\top, \bot\}$
- Flat ordering $\sqsubseteq$, i.e. $\forall q \in Q : \bot \sqsubseteq q \sqsubseteq \top$ and $\bot \sqsubseteq \top$
- $\sqcup = \begin{cases} \bot & X = \emptyset \lor X = \{\bot\} \\ q & X = \{q\} \lor X = \{\bot, q\} \\ \top & \text{else} \end{cases}$
- $\sqcap = \begin{cases} \top & X = \emptyset \lor X = \{\top\} \\ q & X = \{q\} \lor X = \{\top, q\} \\ \bot & \text{else} \end{cases}$



Product Lattice

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

is a lattice.

Proof on the board.

Join-Semi-Lattice

Complete lattice not always required
$\Rightarrow$ remove unused elements

Definition

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

- $(E, \sqsubseteq)$ 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 $\mathcal{E} = (E, \sqsubseteq, \sqcup, \top)$
- a concretization function $[\![\cdot]\!] : E \to 2^C$
  (assigns meaning of abstract states)
  - $[\![\top]\!] = C$
  - $\forall E_{\mathrm{sub}} \subseteq E : \bigcup_{e \in E_{\mathrm{sub}}} [\![e]\!] \subseteq [\![\bigsqcup E_{\mathrm{sub}}]\!]$
    (join operator overapproximates)

Abstraction

$\alpha : 2^C \to E$

Here:

- Not defined separately
- Returns smallest abstract state that covers set of concrete states

Galois Connection

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 $\rightsquigarrow \subseteq E \times G \times E$

- $\forall e \in E, g \in G :$
  $\bigcup_{c \in [\![e]\!]} \{c' \mid (c, g, c') \in \mathcal{T}\} \subseteq \bigcup_{(e,g,e') \in \rightsquigarrow} [\![e']\!]$
  (safe over-approximation)
- Depends on abstract domain
- In this lecture: restricted to functions

Properties of Transfer Functions

- Monotony

$$\forall e, e' \in E, g \in G: \; e \sqsubseteq e' \;\Rightarrow\; \rightsquigarrow(e,g) \sqsubseteq \rightsquigarrow(e',g)$$

- Distributivity (optional)

$$\forall e, e' \in E, g \in G: \; \rightsquigarrow(e,g) \;\sqcup\; \rightsquigarrow(e',g) \;=\; \rightsquigarrow(e \sqcup e', g)$$

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 $\rightsquigarrow$

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

$$\forall e, e' \in E, g \in G : e \sqsubseteq e' \Rightarrow \rightsquigarrow (e, g) \sqsubseteq \rightsquigarrow (e', g)$$

- Distributivity of transfer relation

$$\forall e, e' \in E, g \in G : \rightsquigarrow (e, g) \sqcup \rightsquigarrow (e', g) = \rightsquigarrow (e \sqcup e', g)$$

Location Abstraction $\mathbb{L}$

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)
- $(\ell, (l, op, l'), \ell') \in \rightsquigarrow_{\mathbb{L}}$ if $(\ell = l \lor \ell = \top)$ and $\ell' = l'$

Properties of Location Abstraction

Transfer relation $\rightsquigarrow_{\mathbb{L}}$

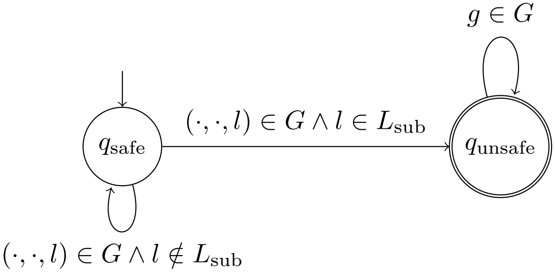
- overapproximates, i.e.,

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

- monotone
- distributive

Property Encoding

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



Property Abstraction $\mathbb{R}$

Represent automaton encoding of property φ_{L_sub} as abstraction

- Uses join-semilattice on set $\{q_\mathrm{safe}, q_\mathrm{unsafe}\}$ with $q_\mathrm{safe} \sqsubseteq q_\mathrm{unsafe}$
- $[\![q]\!] := \begin{cases} C & \text{if } q = q_\mathrm{unsafe} \\ \{c \in C \mid c(pc) \notin L_\mathrm{sub}\} & \text{else} \end{cases}$
- $(q, (l, op, l'), q') \in \rightsquigarrow_\mathbb{R}$ if $q' = q_\mathrm{unsafe} \wedge l' \in L_\mathrm{sub}$ or $q' = q \wedge l' \notin L_\mathrm{sub}$

Properties of Property Abstraction

Transfer relation $\rightsquigarrow_{\mathbb{R}}$

- overapproximates
- monotone
- distributive

Value Domain

Assigns values to (some) variables.

- Domain elements are partial functions $f : Var \twoheadrightarrow \mathbb{Z}$
- $f \sqsubseteq f'$ if $dom(f') \subseteq dom(f)$
  and $\forall v \in dom(f') : f(v) = f'(v)$
- $\sqcup F = \bigcap F$
- $\top = \{\}$
- $[\![f]\!] = \{c \mid \forall v \in dom(f) : c(d)(v) = f(v)\}$

Value Abstraction $\mathbb{V}$

Uses variable-separate domain

- Base domain flat lattice of $\mathbb{Z}$
- Abstract value $\top$ means any value
- Transfer relation
  Notation: $\phi(expr, f) := expr \;\wedge \bigwedge_{v \in dom(f)} v = f(v)$
  - Assume: $(f, (\cdot, expr, \cdot), f') \in \rightsquigarrow_{\mathbb{V}}$ 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: $(f, (\cdot, w := expr;, \cdot), f') \in \rightsquigarrow_{\mathbb{V}}$ 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 $\phi(v' = expr, f)$

Properties of Value Abstraction $\mathbb{V}$

Transfer relation

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

$$f : x \mapsto 3; y \mapsto 2 \qquad f' : x \mapsto 2; y \mapsto 3$$
$$\rightsquigarrow(f, x = x + y;) \sqcup \rightsquigarrow(f', x = x + y;) : x \mapsto 5; y \mapsto \top,$$
but $\rightsquigarrow(f \sqcup f', x = x + y;) : x \mapsto \top; y \mapsto \top$

Example Abstract Transitions

On the board:

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

Sign Abstraction

Variable-separate domain using base domain

$$
\begin{tikzcd}[arrows=-]
 & \top \arrow[dl] \arrow[d] \arrow[dr] & \\
0+ \arrow[d] \arrow[dr] & +- \arrow[dl] \arrow[dr] & -0 \arrow[dl] \arrow[d] \\
+ \arrow[dr] & 0 \arrow[d] & - \arrow[dl] \\
 & \bot &
\end{tikzcd}
$$

$$[\![\top]\!] = \mathbb{Z} \quad [\![+]\!] = \mathbb{N}^+ \quad [\![-]\!] = \mathbb{Z} \setminus \mathbb{N}_0^+ \quad [\![0]\!] = \{0\}$$
$$[\![+-]\!] = \mathbb{Z} \setminus \{0\} \quad [\![0+]\!] = \mathbb{N}_0^+ \quad [\![-0]\!] = \mathbb{Z} \setminus \mathbb{N}^+ \quad [\![\bot]\!] = \emptyset$$

Transfer Relation of Sign Abstraction

Suggestion 1:

- $\rightsquigarrow (f, g) = f'$ 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: $\rightsquigarrow(f, expr) = f'$

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

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
var>0, var==0, var<0, var>=0, 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 $+-$
- $e + 0 = e$
- $e - 0 = e$
- $e * 0 = 0$
- ...

Interval Abstraction $\mathbb{I}$

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 $c \leq a \wedge b \leq d$
- $\sqcup E_{\mathrm{sub}} = \begin{cases} \top & \text{if } \top \in E_{\mathrm{sub}} \\ \bot & \text{if } E_{\mathrm{sub}} \subseteq \{\bot\} \\ [\min_{[a,b] \in E_{\mathrm{sub}}} a, \max_{[a,b] \in E_{\mathrm{sub}}} b] & \text{else} \end{cases}$
- $[\![[a, b]]\!] = \{x \in \mathbb{Z} \mid a \leq x \leq b\} \quad [\![\top]\!] = \mathbb{Z} \quad [\![\bot]\!] = \emptyset$

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: [const,const]
- var: f(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 $\perp, \top$

Transfer Relation of Interval Abstraction

Relies on abstract evaluation of expressions in state f

Boolean expression

- [a,b]= $\begin{cases} \{true\} & a > 0 \vee b < 0 \\ \{false\} & a = b = 0 \\ \{true, false\} & \text{else} \end{cases}$
- [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 Pred
  (subset of boolean expressions without boolean connectors)
- Conjunction of predicates

Cartesian Predicate Abstraction

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

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

- Transfer relation
  - Assignment
    $(p, v = expr, p')$ with
    $p' = \left\{t \in Pred \;\middle|\; \left(\bigwedge_{t' \in p} t'[v \to v'] \wedge v = expr[v \to v']\right) \Rightarrow t\right\}$
  - Assume
    $(p, expr, p')$ if $\bigwedge_{t \in p} t \wedge expr$ is satisfiable and
    $p' = \{t \in Pred \mid (\bigwedge_{t' \in p} t' \wedge expr) \Rightarrow t\}$

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:

- $\rightsquigarrow (\{x = 10\}, (l, i = 1;, l'))$
- $\rightsquigarrow (\{i > 0\}, (l, i = i * 2;, l'))$
- $\rightsquigarrow (\{i > 0\}, (l, i < abs, l'))$
- $\rightsquigarrow (\{x = 10, i > 0\}, (l, x > 10, l'))$

Composite Abstraction

Combines two abstractions

- Product (join-semi) lattice $E_1 \times E_2$
- $[\![(e_1, e_2)]\!] = [\![e_1]\!]_1 \cap [\![e_2]\!]_2$
- Product transfer relation
  $((e_1, e_2), g, (e_1', e_2')) \in \rightsquigarrow$
  if $(e_1, g, e_1') \in \rightsquigarrow_1$ and $(e_2, g, e_2') \in \rightsquigarrow_2$
- 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 $\mathbb{L} \times \mathbb{V} \times \mathbb{R}$
- Predicate analysis $\mathbb{L} \times \mathbb{P} \times \mathbb{R}$

