Semantics: Application to C Programs

Lecture

Matthias Kettl

© SoSy-Lab, LMU Munich, Germany

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

Organization

Lecture and Exercise

Lecture
Feb 27, 2025, 12:30 – 16:00

Exercise
Feb 28, 2025, 10:00 – 16:00

 © SoSy-Lab, LMU Munich, Germany

Course Material

https://www.sosy-lab.org/Teaching/2024-WS-Semantik/

Required software:

- Linux
- Java 17
- CPAchecker 4.0
- Python >= 3.12
- pip (usually comes with python)

 © SoSy-Lab, LMU Munich, Germany

Introduction

Bingo

C

Use-Def

Specification

Invariant

State Space

Formal Verification

Dead Code

Model Checking

Taint Analysis

Least Upper Bound

Constant Propagation

Partial Order

Program Syntax

CPAchecker

SMT

Predicate Abstraction

Program Path

Operational Semantics

Software Analysis

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

Applications

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

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;
}
```

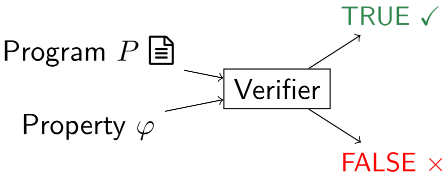
Software Verification

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.

 © SoSy-Lab, LMU Munich, Germany

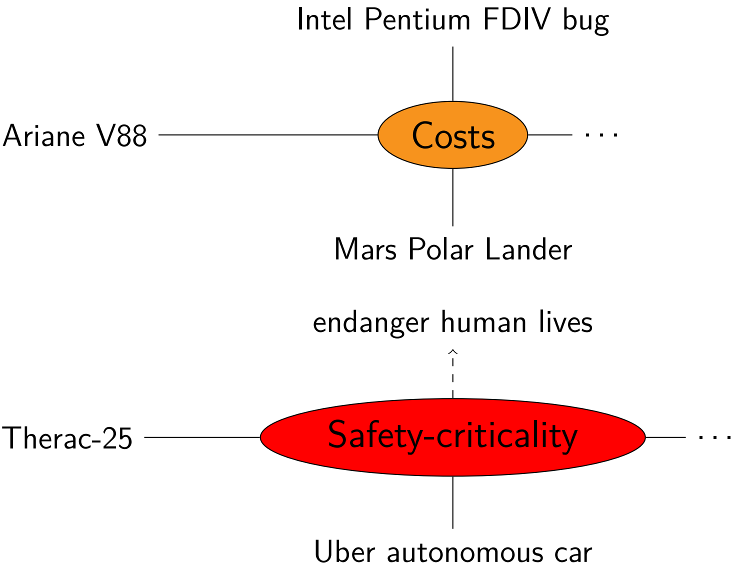
Does This Code Work?

```
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

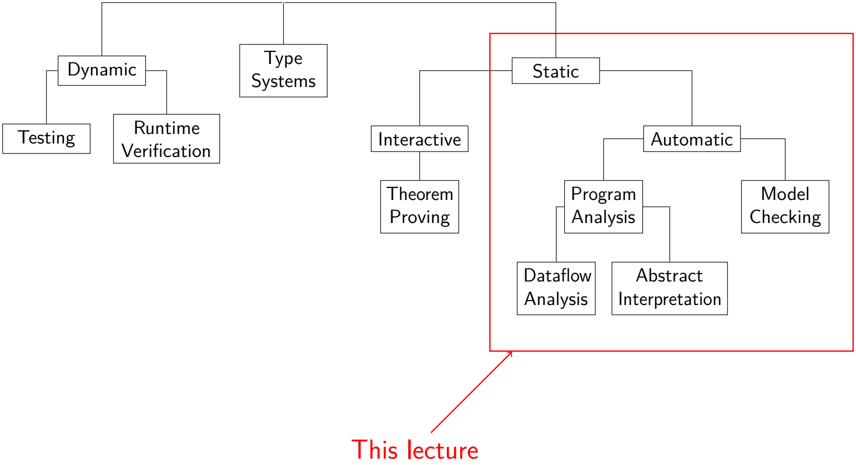
Sapienz Klee PeX SymCC

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.

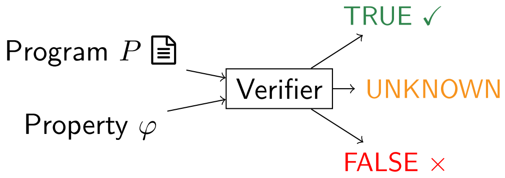
Consequences

Techniques are

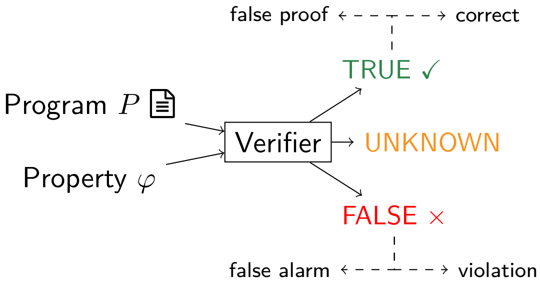
- incomplete, e.g. answer UNKNOWN, or
- unsound, i.e., report
  - false alarms (non-existing bugs),
  - false proofs (miss bugs).

Verifier Design Space

Ideal verifier

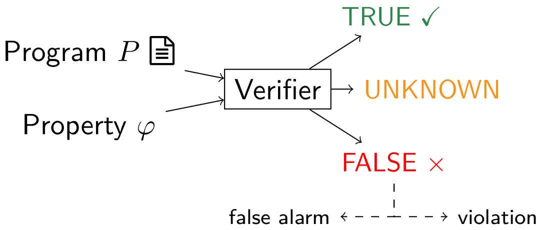


Unreliable verifier



Verifier Design Space

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



- Underapproximating verifier (subset of program behavior)

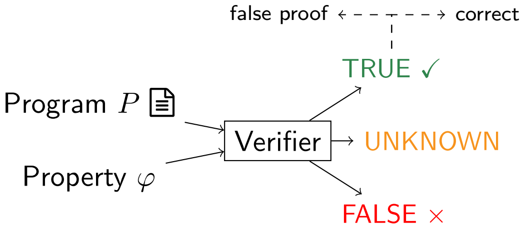


Illustration Underapproximation

Consider the following program (assume **int** $= \mathbb{Z}$):

```
int sign(int y) {
  int signVar = 0;
  if (y < 0)
    signVar = −1;
  if (y > 0)
    signVar = −1; // copy error
  if (y ∗ signVar < 0)
    ERROR: ;
  return signVar;
}
```

Assume that our verifier **underapproximates** with $y > 0$.

Q: What's the verdict?
A: FALSE

Q: Can we be sure that there is indeed an error?
A: Yes

Illustration Underapproximation

Consider the following program (assume **int** $= \mathbb{Z}$):

```
int sign(int y) {
  int signVar = 0;
  if (y < 0)
    signVar = -1;
  if (y > 0)
    signVar = -1; // copy error
  if (y * signVar < 0)
    ERROR: ;
  return signVar;
}
```

Assume that our verifier **underapproximates** with $y < 0$.

Q: What's the verdict?
A: TRUE

Q: Can we be sure that there is no error?
A: No!

Illustration Overapproximation

Consider the following program (assume **int** $= \mathbb{Z}$):

```
void foo(int y) {
  if (y > 100) {
    if (y < 10) {
      // unreachable
      ERROR: ;
    }
  }
}
```

Assume that our verifier can only track $y > 0$ and $y \leq 0$ and **overapproximates** with $y > 0$ within the first if statement.

Q: What's the verdict?
A: FALSE

Q: Can we be sure that there is an error?
A: No!

Illustration Overapproximation

Consider the following program (assume **int** $= \mathbb{Z}$):

```
void foo(int y) {
  if (y > 100) {
    if (y < 10) {
      // unreachable
      ERROR: ;
    }
  }
}
```

Assume that our verifier can only track $y > 10$ and $y \leq 10$ and **overapproximates** with $y > 10$ within the first if statement.

Q: What's the verdict?
A: TRUE

Q: Can we be sure that the program is safe?
A: Yes!

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

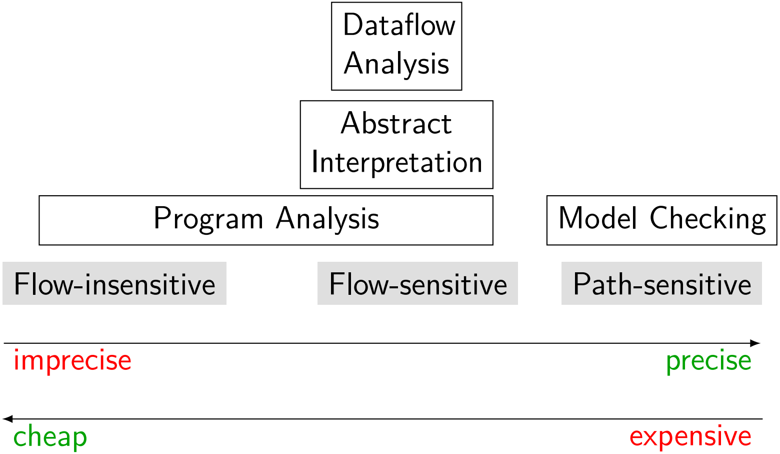
```
if (x>0)
    y=1;
else
    y=0;
if (x>0)
    y=y+2;
else
    y=y+1;
```

To detect that y has value 1 or 3

- must exclude infeasible, syntactic path along first else-branch and second then-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 ($\texttt{var} \in V,\ n \in \mathrm{Num},\ a_i \in \texttt{AExp}$):
  $\texttt{AExp} := n \mid \texttt{var} \mid -a_0 \mid a_1\ op_{\mathrm{a}}\ a_2$
  op_{a} standard arithmetic operation like $+, -, /, \%, \ldots$
- Boolean expressions ($a_i \in \texttt{AExp},\ b_i \in \texttt{BExp}$):
  $\texttt{BExp} := a_0 \mid a_1\ op_{\mathrm{c}}\ a_2 \mid \texttt{!}b_0 \mid b_1\ op_{\mathrm{b}}\ b_2$
  - integer value $0 \equiv f\!f$, remaining values represent true
  - op_c comparison operator like `<, <=, >=, >, ==, !=`
  - op_b logic connective like `&&, ||, ^`
- Program ($\mathsf{a} \in \texttt{AExp}, \mathsf{b} \in \texttt{BExp}$):

S := `var` = a; | `while` (b) S | `if` (b) S `else` S | `if` (b) S | S;S

How to Represent a Program?

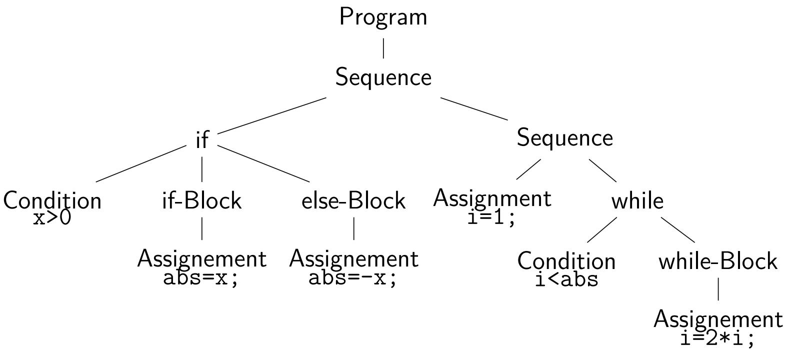
1. Source code

```
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?

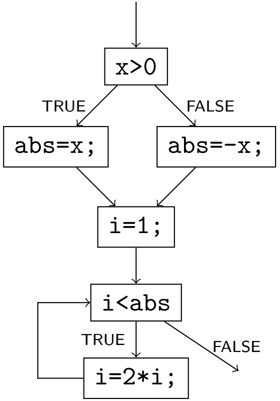
2. Abstract-syntax tree (AST)



- Hierarchical representation
- Flow, paths hard to detect

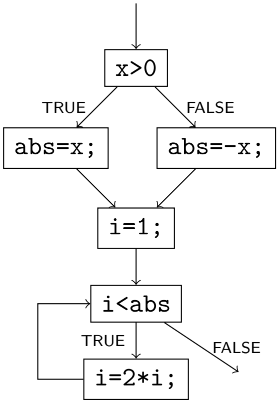
How to Represent a Program?

3. Control-flow graph

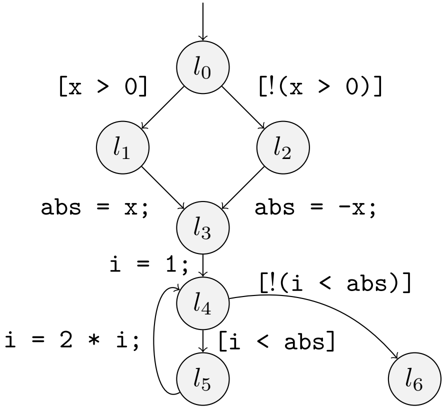


How to Represent a Program?

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;`)

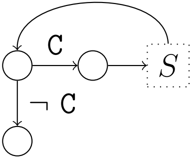
From Source Code to Control-Flow Automaton

Assignment `var=expr;`

From Source Code to Control-Flow Automaton

Assignment `var=expr;`

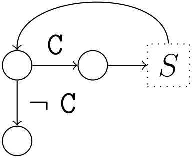
While-Statement `while` (C) S



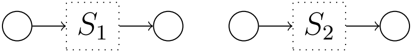
From Source Code to Control-Flow Automaton

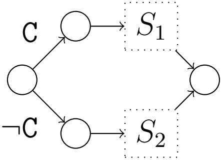
Assignment `var=expr;`

While-Statement `while` (C) S



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

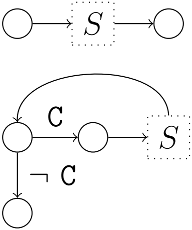




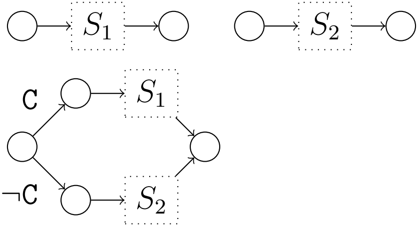
From Source Code to Control-Flow Automaton

Assignment `var=expr;`

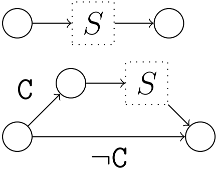
While-Statement `while` (C) S



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



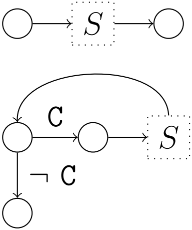
If-Statement `if` (C) S



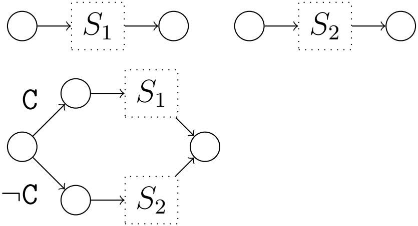
From Source Code to Control-Flow Automaton

Assignment `var=expr;`

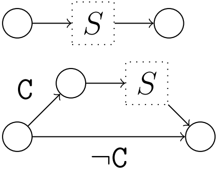
While-Statement `while` (C) S



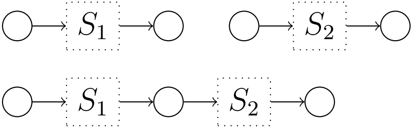
If-Statement `if` (C) S_1 `else` S_2



If-Statement `if` (C) S



Sequential Composition $S_1; S_2$



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

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

Semantics of Arithmetic Expressions

Evaluation function $\mathcal{A}[\![\, - \,]\!]\sigma : \texttt{AExp} \to (\Sigma \to \mathbb{Z})$

Defined recursively on structure

- $\mathcal{A}[\![n]\!]\sigma = \mathcal{N}[\![n]\!]$
- $\mathcal{A}[\![\texttt{var}]\!]\sigma = \sigma(\texttt{var})$
- $\mathcal{A}[\![t_1 \;\; op_{\mathrm{a}} \;\; t_2]\!]\sigma = \mathcal{A}[\![t_1]\!]\sigma \;\; op_{\mathrm{a}} \;\; \mathcal{A}[\![t_2]\!]\sigma$
- $\mathcal{A}[\![-t]\!]\sigma = -\mathcal{A}[\![t]\!]\sigma$

Semantics of Boolean Expressions

Evaluation function $\mathcal{B}[\![\, - \,]\!]\sigma : \texttt{BExp} \rightarrow (\Sigma \rightarrow \mathbb{B})$

Defined recursively on structure

- arithmetic expression:

$$\mathcal{B}[\![a]\!]\sigma = \begin{cases} tt & \text{if } \mathcal{A}[\![a]\!]\sigma \neq 0 \\ \mathit{ff} & \text{otherwise} \end{cases}$$

- comparison: $\mathcal{B}[\![a_1 \;\; op_{\mathrm{c}} \;\; a_2]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma \;\; op_{\mathrm{c}} \;\; \mathcal{A}[\![a_2]\!]\sigma$
- logic connection: $\mathcal{B}[\![b_1 \;\; op_{\mathrm{b}} \;\; b_2]\!]\sigma = \mathcal{B}[\![b_1]\!]\sigma \;\; op_{\mathrm{b}} \;\; \mathcal{B}[\![b_2]\!]\sigma$

State Update

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

$$\sigma[\texttt{var} = a] = \sigma'$$

with $\sigma'(v) = \begin{cases} \sigma(v) & \text{if } v \neq \texttt{var} \\ \mathcal{A}[\![a]\!]\sigma & \text{else} \end{cases}$

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` $=$ a; (correct update) :
     ... $\wedge \ c'(d) = c(d)[\texttt{var} = \text{a;}]$
   - op assume `bexpr` (feasible, no side effects) :
     ... $\wedge \ \mathcal{B}[\![\texttt{bexpr}]\!]c(d) = tt \ \wedge \ c(d) = c'(d)$

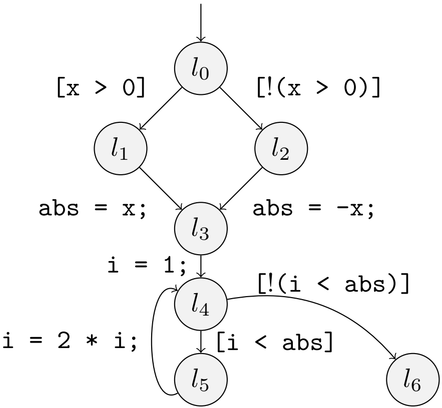
Program Paths

Defined inductively

- every concrete state c with $c(pc) = l_0$ is a program path
- if $c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n$ 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 $paths(P)$.

Examples for Program Paths



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

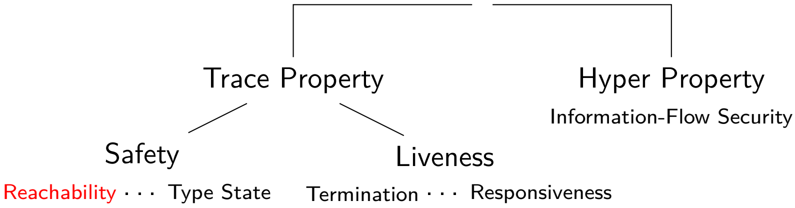
Solves: $\exists n \in N : \; 2^n - |x| \geq 0 \;\wedge\; \forall m < n : \; 2^m - |x| < 0$

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 that a set $\varphi_R \subseteq C$ of concrete states 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

© SoSy-Lab, LMU Munich, Germany

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

 © SoSy-Lab, LMU Munich, Germany

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 $\forall e_1, e_2, e_3 \in E : (e_1 \sqsubseteq e_2 \wedge e_2 \sqsubseteq e_3) \Rightarrow e_1 \sqsubseteq e_3$,
- antisymmetric
  $\forall e_1, e_2 \in E : (e_1 \sqsubseteq e_2 \wedge e_2 \sqsubseteq e_1) \Rightarrow e_1 = e_2$.

 © SoSy-Lab, LMU Munich, Germany

Examples for Partial Orders

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

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 $e' \sqsubseteq e$.

 © SoSy-Lab, LMU Munich, Germany

Computing Upper Bounds

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

Computing Upper Bounds

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

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.

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 $\bot = \sqcap E = \sqcup \emptyset$

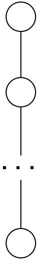
Note:

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

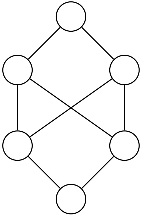
Which Partial Orders Are Lattices?

(a)

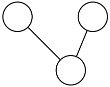
(b)



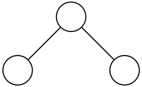
(c)



(d)



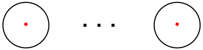
(e)



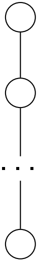
(f)

Which Partial Orders Are Lattices?

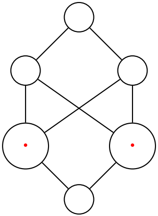
(a) ✓



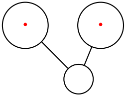
(b) ×



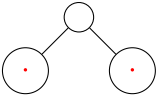
(c) ✓



(d) ×



(e) ×



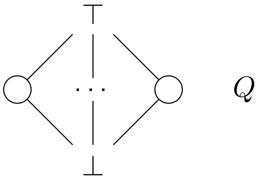
(f) ×

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 $\bot \sqsubseteq \top$
- $\sqcup = \begin{cases} \bot & X = \emptyset \vee X = \{\bot\} \\ q & X = \{q\} \vee X = \{\bot, q\} \\ \top & \text{else} \end{cases}$
- $\sqcap = \begin{cases} \top & X = \emptyset \vee X = \{\top\} \\ q & X = \{q\} \vee X = \{\top, q\} \\ \bot & \text{else} \end{cases}$



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.

 © SoSy-Lab, LMU Munich, Germany

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)
  - $[\![\top]\!] = C$
  - $\forall E_{\text{sub}} \subseteq E : \bigcup_{e \in E_{\text{sub}}} [\![e]\!] \subseteq [\![\sqcup E_{\text{sub}}]\!]$
    (join operator overapproximates, e.g., for flat lattices,)

Example Concretization

Given a semi-lattice $(2^Q, \subseteq, \cup, Q)$ where Q is the set of all predicates, e.g., $\{x > 0, x < 0, x = 0\} \subset Q$.
What does $[\![\{x > 0\}]\!]$ mean?

Example Concretization

Given a semi-lattice $(2^Q, \subseteq, \cup, Q)$ where Q is the set of all predicates, e.g., $\{x > 0, x < 0, x = 0\} \subset Q$.
What does $[\![\{x > 0\}]\!]$ mean?

Answer: $[\![\{x > 0\}]\!] = \{c \in C \mid x = c(d)(x) \Rightarrow x > 0\}$

Abstract Semantics

Abstract interpretation of a program:

- Abstract domain with abstract states E
- CFA $P = (L, l_0, G)$
  with control-flow edges $(l, op, l') = g \in G$

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

Recap: Elements of Abstraction

1. Abstract domain
   - Join-semi lattice $\mathcal{E}$ on set of abstract states E
   - Concretization of abstract states $[\![\cdot]\!]$
2. Abstract semantics $\rightsquigarrow$

Example Abstractions

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 $(\ell = l \vee \ell = \top)$ and $\ell' = l'$

Value Domain

Assigns values to (some) variables.

- Domain elements are partial functions $f : Var \to \mathbb{Z}_\top$
- $f \sqsubseteq f'$ if $dom(f') \subseteq dom(f)$
  and $\forall v \in dom(f') : f(v) = f'(v)$
- $\sqcup F = \bigcap F$ (only keep identical variable-value pairs)
- $\top = \{\}$
- $[\![f]\!] = C \setminus \{c \mid \forall v \in dom(f) : c(d)(v) \neq f(v)\}$

Value Abstraction $\mathbb{V}$

Uses variable-separated domain

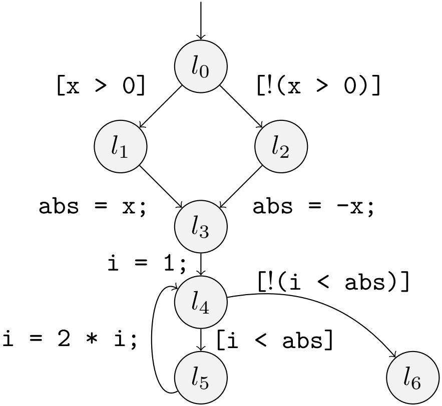
- Base domain flat lattice of $\mathbb{Z}$, $\top$ means any value
- Notation: $\phi(expr, f) := expr \;\wedge \bigwedge_{v \in dom(f) \wedge f(v) \neq \top} v = f(v)$
- Assignment: $(f, (\cdot, w = a;, \cdot), f') \in \rightsquigarrow_{\mathbb{V}}$ if

$$f'(v) = \begin{cases} f(v) & \text{if } v \neq w \\ c & \text{if } v = w \text{ and } c \text{ is the only satisfying} \\ & \text{assignment for } v' \text{ in } \phi(v' = a, f) \\ \top & \text{otherwise} \end{cases}$$

- Assume: $(f, (\cdot, expr, \cdot), f') \in \rightsquigarrow_{\mathbb{V}}$ if
  $\phi(expr, f)$ is satisfiable and

$$f'(v) = \begin{cases} c & \text{if } c \text{ is the only satisfying assignment} \\ & \text{for } v \text{ in } \phi(expr, f) \\ f(v) & \text{otherwise} \end{cases}$$

Example Abstract Transitions



Start with

$f_0 : x \mapsto 2$

$f_0' : \{\}$

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

© SoSy-Lab, LMU Munich, Germany

Cartesian Predicate Abstraction

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

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

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

Example Abstract Transitions

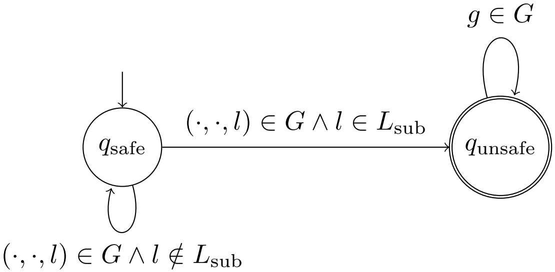
Consider set of predicates {i>0, x=10}

On the board:

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

Property Encoding

An *observer automaton* observes violations of the reachability property $\varphi_{L_{\text{sub}}}$



Property Abstraction $\mathbb{R}$

Represent observer automaton-encoding of property $\varphi_{L_{\mathrm{sub}}}$ 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 $q' = q \wedge l' \notin L_{\mathrm{sub}}$

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

Two Prominent Combinations

- Value analysis $\mathbb{L} \times \mathbb{V} \times \mathbb{R}$
- Predicate analysis $\mathbb{L} \times \mathbb{P} \times \mathbb{R}$

Configurable Program Analysis

Starting Position

Two main analysis techniques: dataflow analysis and model checking

- dataflow analysis: path-insensitive, flow-sensitive
- model checking: path-sensitive
- differences only in behavior when control-flow meets and in termination check

Use synergies → combine into one configurable analysis

Comparing Analysis Algorithms

	Dataflow analysis	Model checking	CPA
	program	program	program
input	abstraction	abstraction	abstraction
	widening operator ∇		e_0, new operators
exploration	reached, waitlist	reached, waitlist	reached, waitlist
	pop from waitlist	pop from waitlist	pop from waitlist
	all successors	all successors	all successors
combination	upper bound (∇) (same location)	never	**merge operator**
coverage	same location, $\sqsubseteq$	same location, $\sqsubseteq$	**stop operator**
termination	empty waitlist	empty waitlist	empty waitlist

Note: flow-insensitive analyses can also be expressed with CPA.

Merge Operator

Defines when and how to combine abstract states

$$\mathsf{merge} : E \times E \to E$$

Correctness criterion:
Must consume second parameter (already explored element)

$$\forall e, e' \in E : e' \sqsubseteq \mathsf{merge}(e, e')$$

 © SoSy-Lab, LMU Munich, Germany

Examples for Merge Operator

- Flow-insensitive: $\mathsf{merge}(e, e') = \sqcup\{e, e'\}$
- Dataflow analysis:
  $\mathsf{merge}((l, e), (l', e')) = \begin{cases} \sqcup\{(l, e), (l', e')\} & \text{if } l = l' \\ (l', e') & \text{else} \end{cases}$
- Model checking: $\mathsf{merge}(e, e') = e'$

Stop Operator

Defines when to stop exploration (termination check)

$$\mathsf{stop} : E \times 2^E \rightarrow \{true, false\}$$

Correctness criterion:
Must be covered by second parameter (set of explored elements)

$$\forall e \in E, E_{\text{sub}} \subseteq E : \mathsf{stop}(e, E_{\text{sub}}) \Rightarrow (\llbracket e \rrbracket \subseteq \bigcup_{e' \in E_{\text{sub}}} \llbracket e' \rrbracket)$$

Examples for Stop Operator

- $\mathsf{stop}(e, E_{\text{sub}}) = false$
- Flow-insensitive: $\mathsf{stop}(e, E_{\text{sub}}) = e \in E_{\text{sub}}$
- Dataflow analysis:
  $\mathsf{stop}((l, e), E_{\text{sub}}) = \exists (l, e') \in E_{\text{sub}} : (l, e) \sqsubseteq (l, e')$
- Model checking: $\mathsf{stop}(e, E_{\text{sub}}) = \exists e' \in E_{\text{sub}} : e \sqsubseteq e'$

Configurable Program Analysis (CPA)

Abstraction plus merge and stop operator

A CPA $\mathbb{C} = ((C, (E, \sqsubseteq, \sqcup, \top), [\![\cdot]\!]), \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$ consists of

- abstract domain $(C, (E, \sqsubseteq, \sqcup, \top), [\![\cdot]\!])$
  - join-semilattice $(E, \sqsubseteq, \sqcup, \top)$
  - $[\![\cdot]\!] : E \to 2^C$ with
    - $[\![\top]\!] = C$
    - $\forall E_{\text{sub}} \subseteq E : \bigcup_{e \in E_{\text{sub}}} [\![e]\!] \subseteq [\![\sqcup E_{\text{sub}}]\!]$
- transfer relation $\rightsquigarrow \subseteq E \times G \times E$ $\forall e \in E, g \in G :$ $\bigcup_{c \in [\![e]\!]} \{c' \mid (c, g, c') \in \mathcal{T}\} \subseteq \bigcup_{(e,g,e') \in \rightsquigarrow} [\![e']\!]$
- merge operator $\mathsf{merge} : E \times E \to E$

$$\forall e, e' \in E : e' \sqsubseteq \mathsf{merge}(e, e')$$

- stop operator $\mathsf{stop} : E \times 2^E \to \{true, false\}$

$$\forall e \in E, E_{\text{sub}} \subseteq E : \mathsf{stop}(e, E_{\text{sub}}) \Rightarrow ([\![e]\!] \subseteq \bigcup_{e' \in E_{\text{sub}}} [\![e']\!])$$

Value Dataflow Analyses as CPA

- abstract domain $\mathbb{L} \times \mathbb{V}$
- transfer relation: product transfer relation $\rightsquigarrow_{\mathbb{L}\times\mathbb{V}}$
- merge operator
  $\text{merge}((l, v), (l', v')) = \begin{cases} \sqcup\{(l, v), (l', v')\} & \text{if } l = l' \\ (l', v') & \text{else} \end{cases}$
- stop operator
  $\text{stop}((l, v), E_{\text{sub}}) = \exists (l, v') \in E_{\text{sub}} : (l, v) \sqsubseteq (l, v')$

© SoSy-Lab, LMU Munich, Germany

Predicate Model Checking as CPA

- abstract domain $\mathbb{L} \times \mathbb{P}$
- transfer relation: product transfer relation $\rightsquigarrow_{\mathbb{L}\times\mathbb{P}}$
- merge operator $\mathsf{merge}(e, e') = e'$
- stop operator
  $\mathsf{stop}((l, p), E_{\mathrm{sub}}) = \exists (l, p') \in E_{\mathrm{sub}} : (l, p) \sqsubseteq (l, p')$

CPA Algorithm

Input: program $P = (L, \ell_0, G)$
CPA $((C, (E, \sqsubseteq, \sqcup, \top), [\![\cdot]\!]), \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$
initial abstract state $e_0 \in E$

reached$=\{e_0\}$; waitlist$=\{e_0\}$;
while (waitlist $\neq \emptyset$) **do**
  pop e from waitlist;
  for each $e \rightsquigarrow e'$ **do**
    for each $e_r \in$ reached **do**
      $e_m =$ merge(e', e_r)
      if $(e_m \neq e_r)$ **then**
        reached$=$(reached $\setminus \{e_r\}) \cup \{e_m\}$;
        waitlist$=$(waitlist $\setminus \{e_r\}) \cup \{e_m\}$;
    if $(\neg$stop$(e', reached))$ **then**
      reached$=$reached$\cup\{e'\}$;
      waitlist$=$waitlist$\cup\{e'\}$;
return reached

Termination of CPA Algorithm

- Generally not guaranteed (inherited from model checking)
- Depends on configuration
  (even for loop-free programs may not terminate, e.g.
  $\text{stop}(e, E_{\text{sub}}) = false$)
- Guarantees for individual techniques (flow-insensitive, dataflow analysis, etc.) still apply

Soundness

Final set $\mathrm{reached}$ overapproximates all reachable states if the initial abstract state e_0 covers all initial states, i.e.,

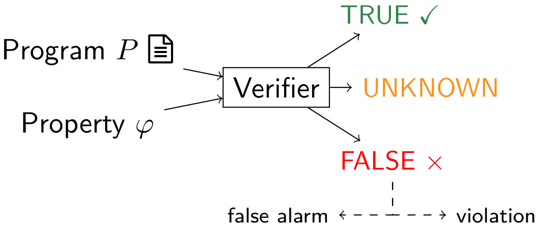
$$\{c \mid c(pc) = l_0\} \subseteq [\![e_0]\!] \Rightarrow reach(P) \subseteq \bigcup_{e \in reached} [\![e]\!]$$

Reasons

- Explore all successors of states in reached (always add state to waitlist if added to reached)
- Transfer relation overapproximates
- Replace state by more abstract (merge property), never only delete
- Must add abstract successor to reached if not covered (stop property)

Classifying Configurable Program Analysis

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

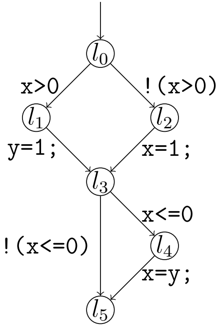


 © SoSy-Lab, LMU Munich, Germany

Counterexample-Guided Abstraction Refinement (CEGAR)

Why CEGAR for Predicate Abstraction?

Let $\varphi_R = \{l_4\}$



Which predicates to use

- {x=1} too coarse
- {y=1;x=1;x>0} too precise
  $\Rightarrow$ inefficient
- {x>0} best candidate

Use CEGAR to determine required set of predicates

Idea of CEGAR

Find good trade-off between precision and efficiency automatically

- Start with efficient abstraction
- Refine if fails to (dis)prove property (spurious counterexample found)
- Often, works automatically

Typically, used for model checking

Counterexample

Syntactical program paths from initial to error location

Definition

Let $P = (L, l_0, G)$ be a CFA and
$\varphi_{L_{\mathrm{sub}}}$ with $L_{\mathrm{sub}} \subseteq L$ be a reachability property.

A sequence $g_1 g_2 \dots g_n \in G^*$ is a counterexample if

- $g_1 = (l_0, \cdot, \cdot)$
- $g_n = (\cdot, \cdot, l_e)$ s.t. $l_e \in L_{\mathrm{sub}}$
- $\forall 1 \leq i < n : g_i = (\cdot, \cdot, l) \Rightarrow g_{i+1} = (l, \cdot, \cdot)$

Feasibility of Counterexamples

A counterexample is feasible if

$$\exists c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n \in paths(P) \ .$$

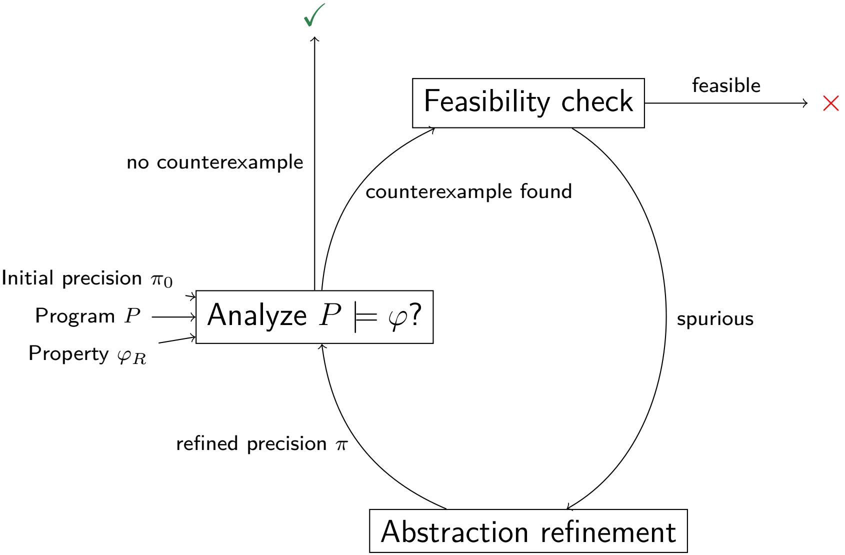
A counterexample is spurious if it is not feasible.

Which Counterexamples are Feasible/Spurious?

Consider reachability property $\varphi = \{l_4\}$.

- $l_0 \stackrel{x>0}{\rightarrow} l_1 \stackrel{y=1;}{\rightarrow} l_3 \stackrel{x<=0}{\rightarrow} l_4$
- $l_0 \stackrel{x>0}{\rightarrow} l_1 \stackrel{y=2;}{\rightarrow} l_3 \stackrel{x>0}{\rightarrow} l_4$
- $l_0 \stackrel{!(x>0)}{\rightarrow} l_2 \stackrel{y=-1;}{\rightarrow} l_3 \stackrel{y==0}{\rightarrow} l_4$
- $l_0 \stackrel{y=0;}{\rightarrow} l_1 \stackrel{x>0}{\rightarrow}\stackrel{y=5;}{\rightarrow} l_3 \stackrel{x>10}{\rightarrow} l_4$

CEGAR Overview



Adapting Model Checking Algorithm for CEGAR

Input: program $P = (L, \ell_0, G)$
abstraction $((L \times E \times R, \sqsubseteq, \sqcup, \top), [\![]\!], \rightsquigarrow)$
reached$\subseteq E$ and waitlist$\subseteq E$

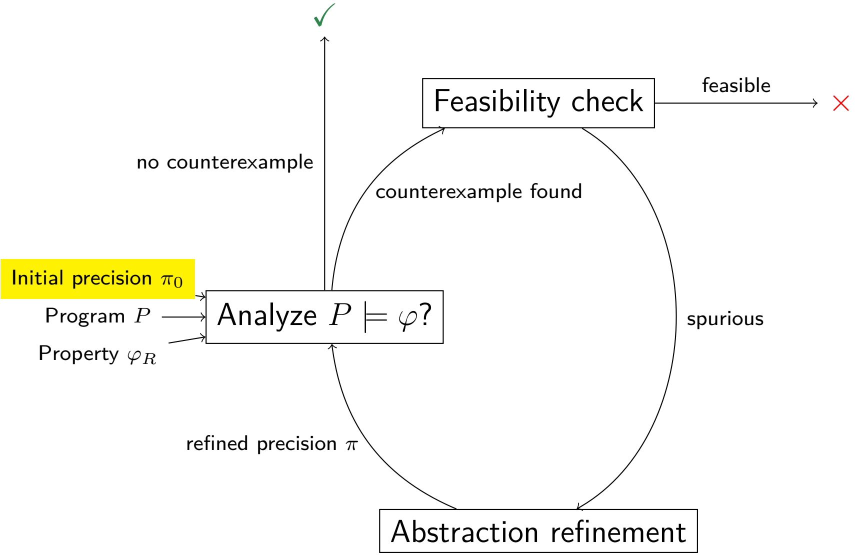
reached=waitlist=$\{(l_0, \top, q_{\text{safe}})\}$;
while (waitlist $\neq \emptyset$) **do**
  pop $(l, e, q))$ from waitlist;
  for each $((l, e, q), g, (l', e', q')) \in \rightsquigarrow_\pi$ **do**
    if $q' = q_{\text{unsafe}}$ **then**
      reached=reached$\cup\{(l', e', q')\}$;
      waitlist=waitlist$\cup\{(l, e, q)\}$;
      return (reached, waitlist)
    if $(\neg\exists(l', e'', q'') \in \text{ reached} : (l', e', q') \sqsubseteq (l', e'', q''))$ **then**
      reached=reached$\cup\{(l', e', q')\}$;
      waitlist=waitlist$\cup\{(l', e', q')\}$;
return (reached, waitlist)

CEGAR Algorithm

Input: program $P = (L, \ell_0, G)$
abstraction $\mathbb{A} = ((L \times E \times R, \sqsubseteq, \sqcup, \top), [\![]\!], \rightsquigarrow)$

reached=waitlist=$\{(l_0, \top, q_{\text{safe}})\}$;
$\pi = \emptyset$;
while (waitlist $\neq \emptyset$) **do**
    modelcheck(P, $\mathbb{A}_\pi$, reached, waitlist);
    if $\exists(\cdot, \cdot, q_{\text{unsafe}}) \in$ reached **then**
        cex = extractErrorPath(reached);
        if isFeasible(cex) **then**
            return unsafe
        $\pi =$ refine(cex);
        reached=waitlist=$\{(l_0, \top, q_{\text{safe}})\}$;
return safe

CEGAR Overview

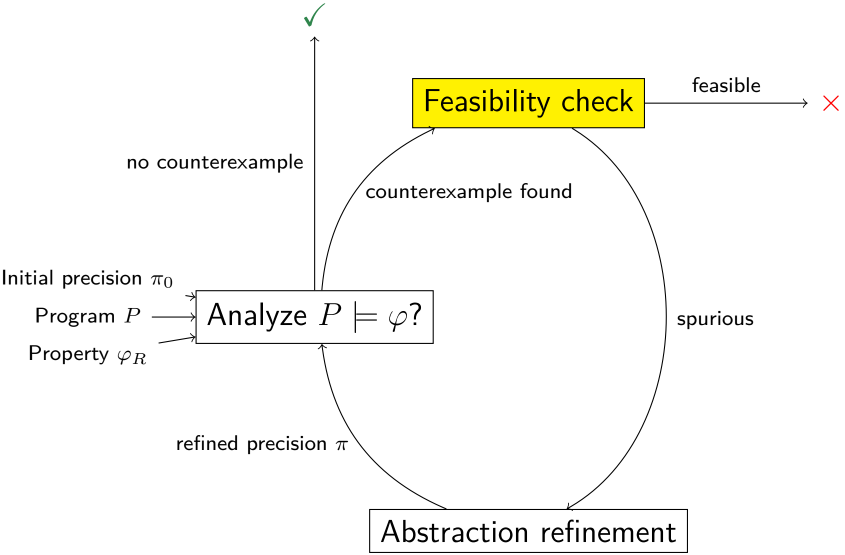


Initial Abstraction

Should be efficient, may be coarse
$\Rightarrow$ Use coarsest abstraction possible, i.e., empty set of predicates for predicate analysis.

© SoSy-Lab, LMU Munich, Germany

CEGAR Overview



Checking Feasibility of Counterexample

Two (dual) possibilities

1. Forward approach
   - Compute strongest postcondition starting with true
   - If unsatisfiable, counterexample spurious
2. Backward approach
   - Compute weakest precondition starting from false
   - If tautology, counterexample spurious

Strongest Postcondition (Recap)

Known from predicate abstraction
Smallest set of successors

For operations:

- Assume b: $sp(b, f) = f \wedge b$
- Assignment v=a;:

$$sp(v = a;, f) = \exists v' : f[v \rightarrow v'] \wedge v = a[v \rightarrow v']$$

Extension to sequences

- $sp(\varepsilon, f) = f$
- $sp(op_1 \ldots op_n, f) = sp(op_2 \ldots op_n, sp(op_1, f))$

© SoSy-Lab, LMU Munich, Germany

Examples for Strongest Postcondition

Consider following sequences of edges G_i. Compute strongest post condition $sp(G_i, true)$.

- $G_1 := x > 0, y = 1; , x <= 0$
- $G_2 := x > 0, y = 2; , x > 0$
- $G_3 :=!(x > 0), y = -1; , y == 0$
- $G_4 := y = 0; , x > 0, y = 5; , x > 10$

Which formulae are satisfiable?

 © SoSy-Lab, LMU Munich, Germany

Skolemization

Eliminate existential quantifiers
(only consider formuala without universal quantifiers ($\forall$))

General schema
$\exists x : f$ replaced by $f[x \to c]$ such that c does not occur in f

In this lecture:
Use static single assignment (SSA) like skolemization

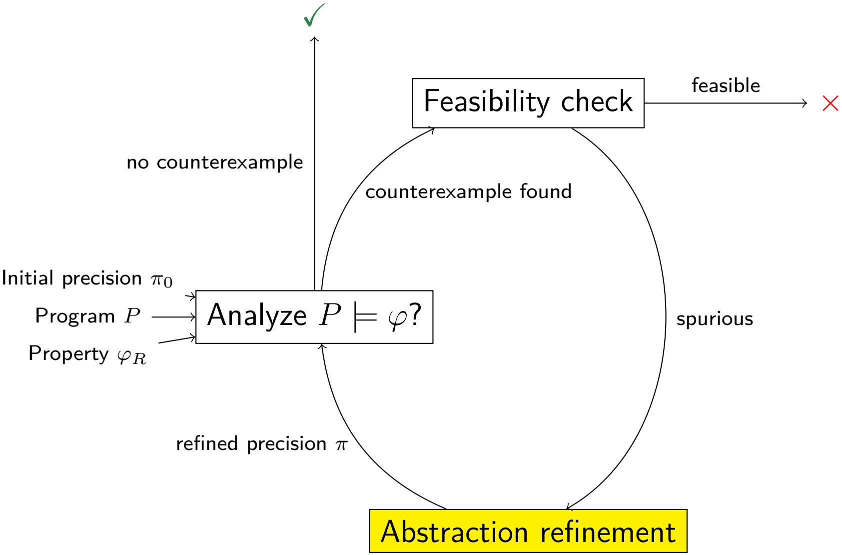
SSA-based Skolemization

- Use indexed variables, i.e., x_i for x
- Start with index 0
- Expressions (assume or right hand side of assignments) use highest index seen so far
- Increase index in assignments, i.e., the index of the assigned variable is maximum seen so far plus 1

Example

$$
\begin{aligned}
& sp(y = 0;, x > 0, y = 5;, x > 10, \mathit{true}) \\
= \; & sp(x > 0, y = 5;, x > 10, y_1 = 0) \\
= \; & sp(y = 5;, x > 10, y_1 = 0 \wedge x_0 > 0) \\
= \; & sp(x > 10, y_1 = 0 \wedge x_0 > 0 \wedge y_2 = 5) \\
= \; & sp(\varepsilon, y_1 = 0 \wedge x_0 > 0 \wedge y_2 = 5 \wedge x_0 > 10) \\
= \; & y_1 = 0 \wedge x_0 > 0 \wedge y_2 = 5 \wedge x_0 > 10
\end{aligned}
$$

CEGAR Overview



Abstraction Refinement

Increases precision (more precise abstraction)
Based on counterexample

Goal
New exploration excludes counterexample
(ensures progress)

In this lecture: refinement via interpolation

Craig Interpolant

Let f_1 and f_2 be two formulas such that $f_1 \wedge f_2$ unsatisfiable.

A formula f is an interpolant if

- $f_1 \Rightarrow f$
- $f \wedge f_2$ unsatisfiable
- $var(f) \subseteq (var(f_1) \cap var(f_2))$

Which Formulae are Craig Interpolants?

Consider the formulae

$$f_1 = x_0 > 10 \wedge y_1 = 1$$

$$f_2 = x_0 \leq 0 \wedge z_0 = 0$$

Which of the following formulae are interpolants for f_1 and f_2?

- f_1
- $x_0 > 0 \wedge (z_0 > 0 \vee z_0 \leq 0)$
- false
- $x_0 \neq 0$
- $x_0 > 0$

Computing New Predicates via Interpolation

Let $op_1 \ldots op_n$ be sequence of operations on spurious counterexample

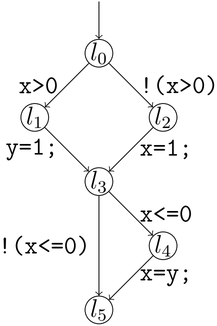
For all $1 \leq k < n$ compute

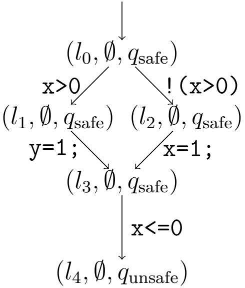
1. Compute $f_1 = sp(op_1 \ldots op_k, true)$ and $f_2 = sp(op_{k+1} \ldots op_n, true)$ (index shift!) (can reuse strongest postcondition from feasibility check, only need to split appropriately)
2. Compute interpolant f for f_1 and f_2
3. Use all literals from f after removing indices as new predicates

In practice:
solver computes interpolants from unsatisfiability proof

CEGAR with Predicate Abstraction

Let $\varphi_R = \{l_4\}$ and initial set of predicates $\{\}$





Two counterexamples:

- $l_0 \overset{x>0}{\longrightarrow} l_1 \overset{y=1;}{\longrightarrow} l_3 \overset{x<=0}{\longrightarrow} l_4$
- $l_0 \overset{!(x>0)}{\longrightarrow} l_2 \overset{x=1;}{\longrightarrow} l_3 \overset{x<=0}{\longrightarrow} l_4$

Compute Strongest Postcondition

Consider counterexample $l_0 \stackrel{x>0}{\rightarrow} l_1 \stackrel{y=1;}{\rightarrow} l_3 \stackrel{x<=0}{\rightarrow} l_4$

Strongest post $x_0 > 0 \land y_1 = 1 \land x_0 <= 0$ infeasible
$\Rightarrow$ counterexample spurious
$\Rightarrow$ refine

Computing New Predicates via Interpolation

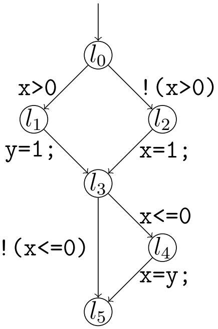
Split formula $x_0 > 0 \wedge y_1 = 1 \wedge x_0 <= 0$ at program locations

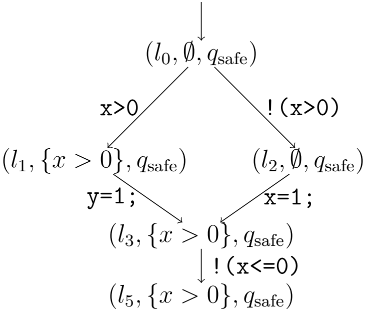
1. $f_1 = x_0 > 0$ and $f_2 = y_1 = 1 \wedge x_0 <= 0$
   interpolant $f = x_0 > 0$
2. $f_1 = x_0 > 0 \wedge y_1 = 1$ and $f_2 = x_0 <= 0$
   interpolant $f = x_0 > 0$

Consider all literals (predicates) occurring in interpolants
Remove SSA indices from these literals and then add them to set of predicates
$\Rightarrow$ new predicates x>0

Restart Predicate Model Checking

Let $\varphi_R = \{l_4\}$ and set of predicates {x>0}





Property proved.

CEGAR for Value Analysis

What would be the precision for value analysis?