Semantics: Application to C Programs

Lecture

Thomas Lemberger

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

Organization

Lecture and Exercise

Lecture
Mar 4, 2022, 10:00 – 12:00

Exercise
Mar 4, 2022, 13:00 – 16:00

Course Material

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

Required software:

- Linux
- Java 11
- CPAchecker 2.1.1
- Python >= 3.8
- pip (usually comes with python)

Introduction

Bingo

C
Use-Def
Invariant
Specification
State Space
Formal Verification
Dead Code
Model Checking
Taint Analysis
Least Upper Bound
Constant Propagation
Partial Order
Program Syntax
CPAchecker
SMT
Axiomatic Semantics
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;
}
```

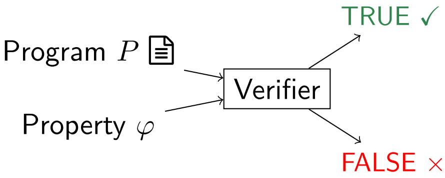
Software Verification

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.

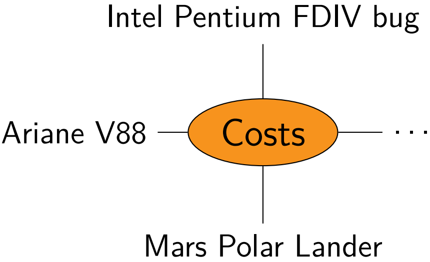
Does This Code Work?

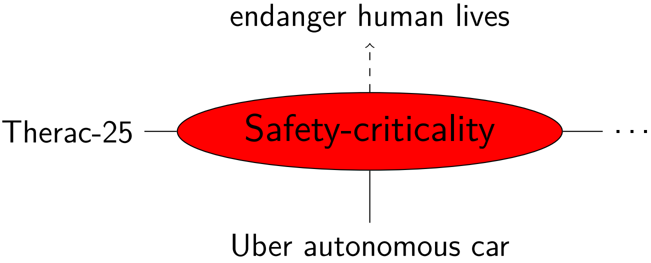
```
double avgUpTo(int[] numbers, int length) {
    double sum = 0;
    for(int i=0;i<length;i++)
        sum += numbers[i];
    return sum/(double)length;
}
```

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

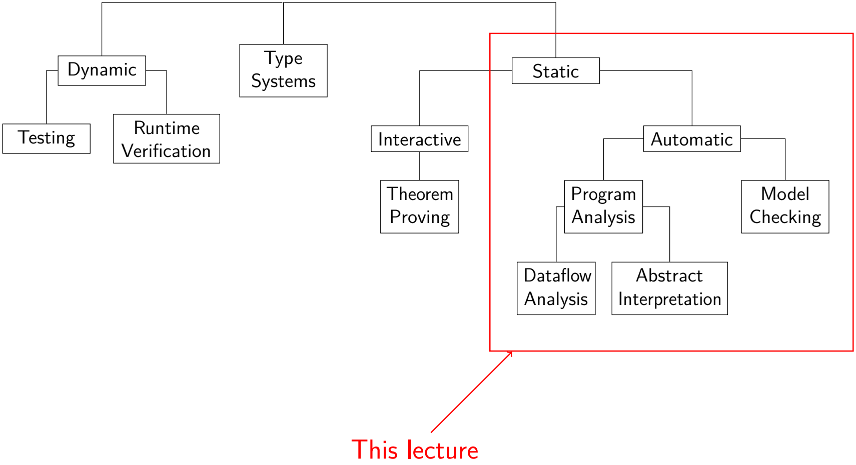
Sapienz Klee PeX SymCC

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.

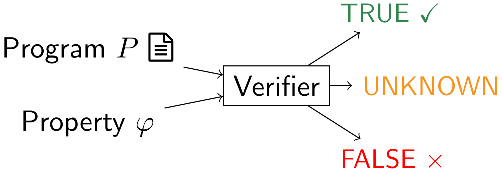
Consequences

Techniques are

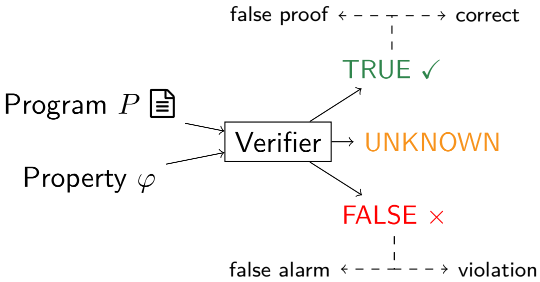
- incomplete, e.g. answer UNKNOWN, or
- unsound, i.e., report
  - false alarms (non-existing bugs),
  - false proofs (miss bugs).

Verifier Design Space

Ideal verifier

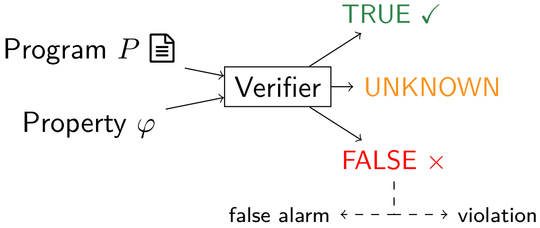


Unreliable verifier

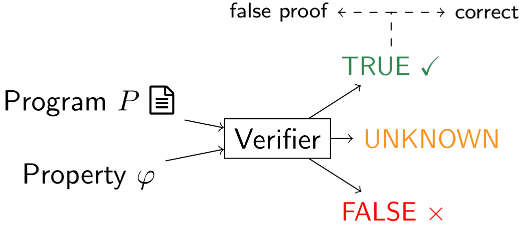


Verifier Design Space

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



- Underapproximating verifier (subset of program behavior)



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

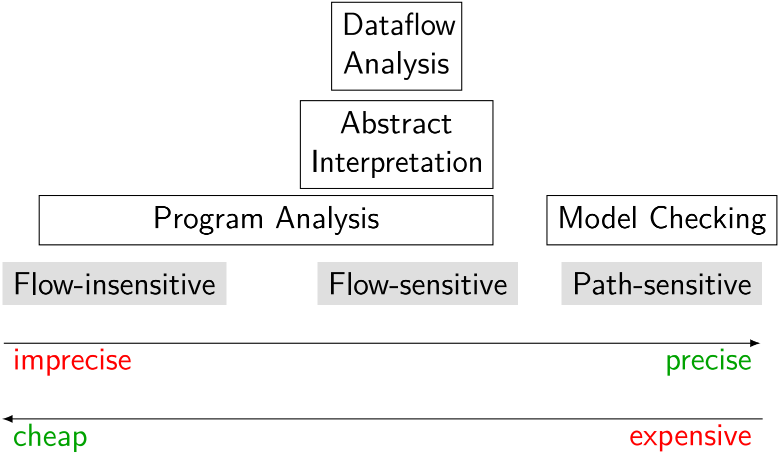
```
if (x>0)
    y=1;
else
    y=0;
if (x>0)
    y=y+2;
else
    y=y+1;
```

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 $\&\&(\wedge), || \ (\vee)$, ^ (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?

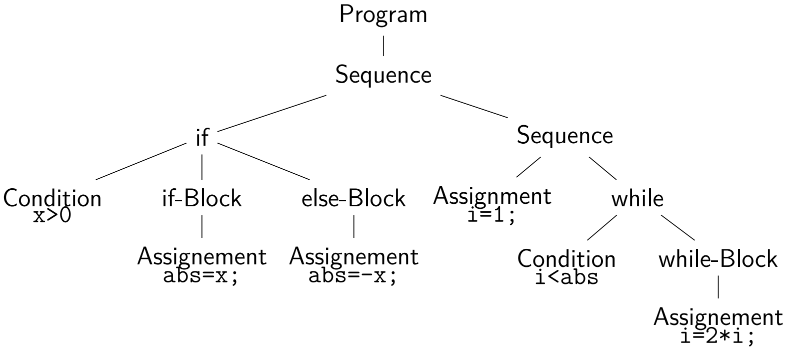
1. Source code

```
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?

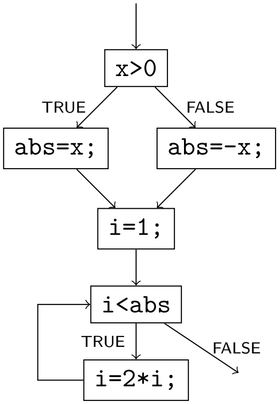
2. Abstract-syntax tree (AST)



- Hierarchical representation
- Flow, paths hard to detect

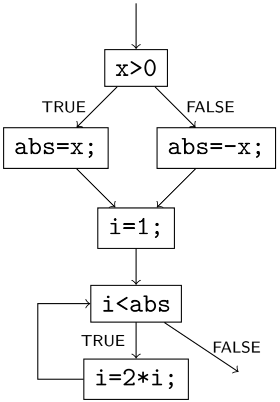
How to Represent a Program?

3. Control-flow graph

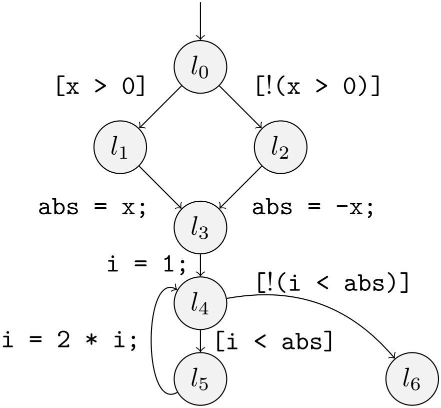


How to Represent a Program?

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;`)

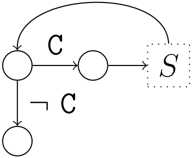
From Source Code to Control-Flow Automaton

Assignment `var=expr;`

From Source Code to Control-Flow Automaton

Assignment `var=expr;`

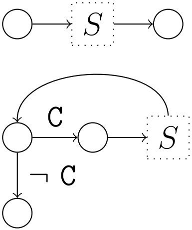
While-Statement `while` (C) S



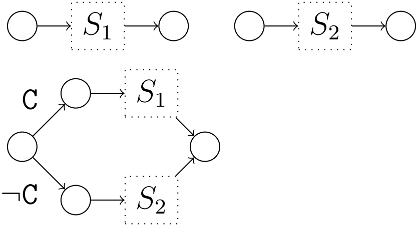
From Source Code to Control-Flow Automaton

Assignment `var=expr;`

While-Statement `while` (C) S



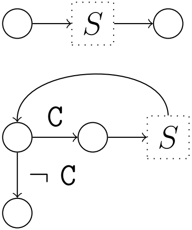
If-Statement `if` (C) S_1 `else` S_2



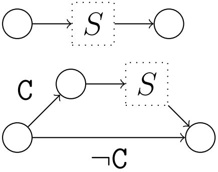
From Source Code to Control-Flow Automaton

Assignment `var=expr;`

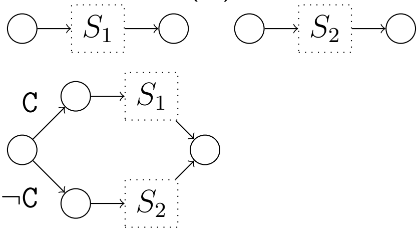
While-Statement `while` (C) S



If-Statement `if` (C) S



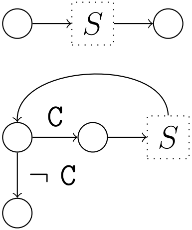
If-Statement `if` (C) S_1 `else` S_2



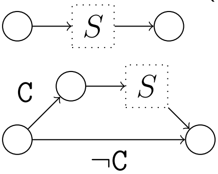
From Source Code to Control-Flow Automaton

Assignment `var=expr;`

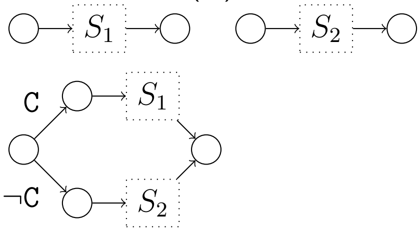
While-Statement `while` (C) S



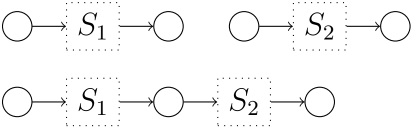
If-Statement `if` (C) S



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



Sequential Composition $S_1; S_2$



Semantics

Different types

- Axiomatic semantics: based on pre- and postconditions, e.g. $\{\text{true}\}$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 $\mathcal{S}_a : aexpr \times \Sigma \to \mathbb{Z}$

Defined recursively on structure

- $\text{const} \in \mathbb{Z} : \mathcal{S}_a(\text{const}, \sigma) = \text{const}$
- variable `var`: $\mathcal{S}_a(\texttt{var}, \sigma) = \sigma(\texttt{var})$
- 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 $\mathcal{S}_b : bexpr \times \Sigma \to \{true, false\}$

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

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

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

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` :
     ... $\wedge \ \mathcal{S}_b(\texttt{bexpr}, c(d)) = true \ \wedge \ c(d) = c'(d)$

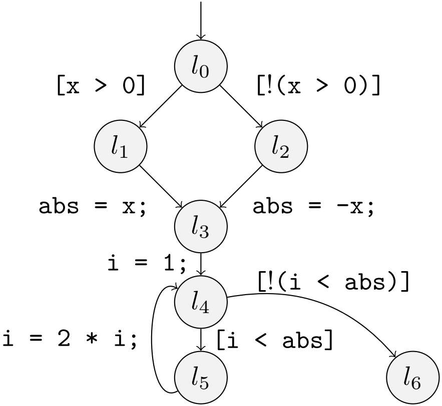
Program Paths

Defined inductively

- every concrete state c with $c(pc) = l_0$ is a program path
- if $c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n$ 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 $\sigma : \texttt{abs} \mapsto 2; \texttt{i} \mapsto 0; \texttt{x} \mapsto -2$

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

- Trace Property
  - Safety
    - Reachability · · · Type State
  - Liveness
    - Termination · · · Responsiveness
- Hyper Property
  - Information-Flow Security

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: $v(P, \varphi_R) = \textsf{FALSE} \land reach(P) \cap \varphi_R = \emptyset$
- False proof: $v(P, \varphi_R) = \textsf{TRUE} \land reach(P) \cap \varphi_R \neq \emptyset$
- 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 $\forall e_1, e_2, e_3 \in E : (e_1 \sqsubseteq e_2 \wedge e_2 \sqsubseteq e_3) \Rightarrow e_1 \sqsubseteq e_3$,
- antisymmetric
  $\forall e_1, e_2 \in E : (e_1 \sqsubseteq e_2 \wedge e_2 \sqsubseteq e_1) \Rightarrow e_1 = e_2$.

Examples for Partial Orders

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

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

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

Definition (Least Upper Bound (lub))

An element $e \in E$ is a least upper bound $\sqcup$ of a subset $E_{\mathrm{sub}} \subseteq E$ 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	$\sqcup$	$\sqcap$
$(\mathbb{Z}, \leq)$	$\{1, 4, 7\}$	7	1
$(\mathbb{Z}, \leq)$	$\mathbb{Z}$	$\times$	$\times$
$(\mathbb{N}, \leq)$	$\emptyset$	0	$\times$
$(2^Q, \subseteq)$	2^Q	Q	$\emptyset$
$(2^Q, \subseteq)$	$\{\emptyset\}$	$\emptyset$	$\emptyset$
$(2^Q, \subseteq)$	$Y \subseteq 2^Q$	$\bigcup_{y \in Y} y$	$\bigcap_{y \in Y} y$

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

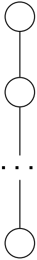
- $(E, \sqsubseteq)$ is a partial order
- least upper bound $\sqcup$ and greater lower bound $\sqcap$ exist for all subsets $E_{\text{sub}} \subseteq E$
- $\top = \sqcup E = \sqcap\emptyset$ and $\bot = \sqcap E = \sqcup\emptyset$

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

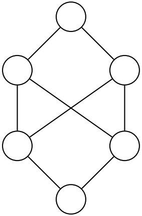
Which Partial Orders Are Lattices?

(a)

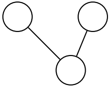
(b)



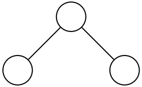
(c)



(d)



(e)



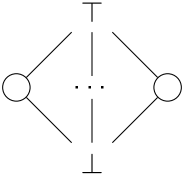
(f)

Flat-Lattice

Definition

A flat lattice of set Q consists of

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



Q

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.

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)

Abstraction

$\alpha : 2^C \rightarrow E$

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. $\forall C_{\mathrm{sub}} \subseteq C : C_{\mathrm{sub}} \subseteq [\![\alpha(C_{\mathrm{sub}})]\!]$
   (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 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

Properties of Transfer Relations

- Monotony

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

- Distributivity (optional)

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

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 $\ell' = l'$

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

Value Domain

Assigns values to (some) variables.

- Domain elements are partial functions $f : Var \rightharpoonup \mathbb{Z}$
- $f \sqsubseteq f'$ 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-separated domain

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

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

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

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

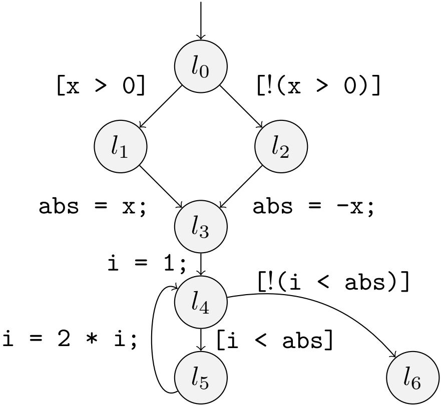
Properties of Value Abstraction $\mathbb{V}$

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



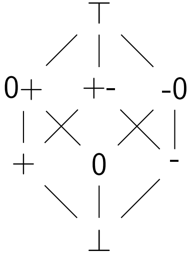
Start with

$f_0 : x \mapsto 2, abs \mapsto \top, i \mapsto \top$

$f_0' : x \mapsto \top, abs \mapsto \top, i \mapsto \top$

Sign Abstraction

Variable-separate domain using base domain



$$\llbracket \top \rrbracket = \mathbb{Z} \quad \llbracket + \rrbracket = \mathbb{N}^+ \quad \llbracket - \rrbracket = \mathbb{Z} \setminus \mathbb{N}_0^+ \quad \llbracket 0 \rrbracket = \{0\}$$
$$\llbracket + - \rrbracket = \mathbb{Z} \setminus \{0\} \quad \llbracket 0 + \rrbracket = \mathbb{N}_0^+ \quad \llbracket - 0 \rrbracket = \mathbb{Z} \setminus \mathbb{N}^+ \quad \llbracket \bot \rrbracket = \emptyset$$

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:

- Assignment: $\rightsquigarrow(f, aexpr) = f'$

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

$$\text{v=w;}\quad f'(v) = f(w)$$

$$\text{v=expr;}\quad f'(v) = \top$$

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

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

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 $+-$
- $e + 0 = e$
- $e - 0 = e$
- $e * 0 = 0$
- $\ldots$

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

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 $\bot, \top$

Transfer Relation of Interval Abstraction

Relies on abstract evaluation of expressions in state f

Boolean expression

- [a,b]= $\begin{cases} \{true\} & a > 0 \vee b < 0 \\ \{false\} & a = b = 0 \\ \{true, false\} & \text{else} \end{cases}$
- [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 $(2^{\mathrm{Pred}}, \supseteq, \cap, \cup, \emptyset, \mathrm{Pred})$

- $[\![\top]\!] = [\![\emptyset]\!] = C$
  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
    $(p, v = aexpr, p')$ with
    $p' = \left\{t \in \mathrm{Pred} \mid \left(\bigwedge_{t' \in p} t'[v \to v_{old}] \wedge v = aexpr[v \to v_{old}]\right) \Rightarrow t\right\}$
  - Assume
    $(p, bexpr, p')$ if $\bigwedge_{t \in p} t \wedge bexpr$ is satisfiable and
    $p' = \{t \in Pred \mid (\bigwedge_{t' \in p} t' \wedge bexpr) \Rightarrow t\}$

Properties of Cartesian Predicate Abstraction

Transfer relation

- overapproximates
- monotone
- not distributive

Example Abstract Transitions

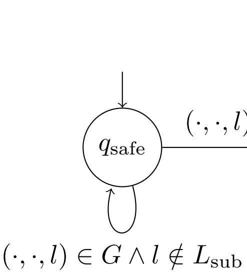
Consider set of predicates {i>0, x=10}

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 $\varphi_{L_{\text{sub}}}$ as abstraction

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

Properties of Property Abstraction

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

- overapproximates
- monotone
- distributive

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.

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

3 analysis techniques

- Often, similar
- But, not identical

Use synergies $\rightarrow$ combine into one configurable analysis

Comparing Analysis Algorithms

	Path-insensitive	Dataflow analysis	Model checking	
input	program abstraction initial state e_0	program abstraction widening operator ∇	program abstraction	program abstraction e_0, new operators
exploration	one element last state all successors	reached, waitlist pop from waitlist all successors	reached, waitlist pop from waitlist all successors	reached, waitlist pop from waitlist all successors
combination	least upper bound	upper bound (∇) (same location)	never	**merge operator**
coverage	identical	same location, $\sqsubseteq$	same location, $\sqsubseteq$	**stop operator**
termination	!changed	empty waitlist	empty waitlist	empty waitlist

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)

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

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)

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

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

- stop$(e, E_{\text{sub}}) = false$
- Flow-insensitive: stop$(e, E_{\text{sub}}) = e \in E_{\text{sub}}$
- Dataflow analysis and model checking:
  stop$((l, e), E_{\text{sub}}) = \exists (l, e') \in E_{\text{sub}} : (l, e) \sqsubseteq (l, e')$
  and
  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 $\mathsf{stop} : E \times 2^E \to \{true, false\}$

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

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
  $\mathsf{merge}((l, v), (l', v')) = \begin{cases} \sqcup\{(l, v), (l', v')\} & \text{if } l = l' \\ (l', v') & \text{else} \end{cases}$
- stop operator
  $\mathsf{stop}((l, v), E_{\text{sub}}) = \exists (l, v') \in E_{\text{sub}} : (l, v) \sqsubseteq (l, v')$

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 $\text{merge}(e, e') = e'$
- stop operator
  $\text{stop}((l, p), E_{\text{sub}}) = \exists (l, p') \in E_{\text{sub}} : (l, p) \sqsubseteq (l, p')$

CPA Algorithm

Input: program $P = (L, \ell_0, G)$
CPA $((C, (E, \sqsubseteq, \sqcup, \top), [\![\cdot]\!]), \rightsquigarrow, \text{merge}, \text{stop})$
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.
  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.,

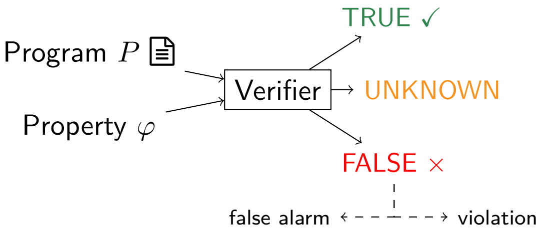
$$\{c \mid c(pc) = l_0\} \subseteq [\![e_0]\!] \Rightarrow reach(P) \subseteq \bigcup_{e \in reached} [\![e]\!]$$

Reasons

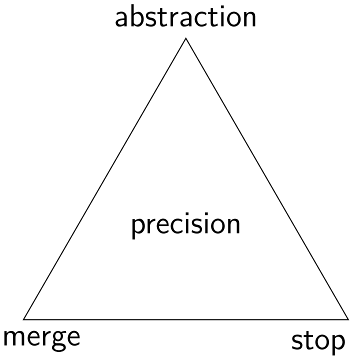
- Explore all successors of states in reached
  (always add state to waitlist if added to reached)
- Transfer relation overapproximates
- Replace state by more abstract (merge property), never only delete
- Must add abstract successor to reached if not covered (stop property)

Classifying Configurable Program Analysis

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



Exploring the Configuration Space



- Which set of concrete elements can be distinguished?
- merge: never $\leftrightarrow$ always $\top$
- stop: $[\![e]\!] \subseteq \bigcup_{e' \in E_{\text{sub}}} [\![e']\!] \leftrightarrow$ false

$\Rightarrow$ Relaxation to become more efficient