形式化方法导引

第 6 章 案例分析

6.2 Software Analysis: Abstract Interpretation | CEGAR

黄文超

https://faculty.ustc.edu.cn/huangwenchao

$\longrightarrow$ 教学课程 $\longrightarrow$ 形式化方法导引

```
MODULE main
VAR
  y : 0..15000;
ASSIGN
  init(y) := 0;
TRANS
  case
    y=70  : next(y)=0;
    TRUE : next(y)=y+1;
  esac
LTLSPEC
  G (y in (0..70))
```

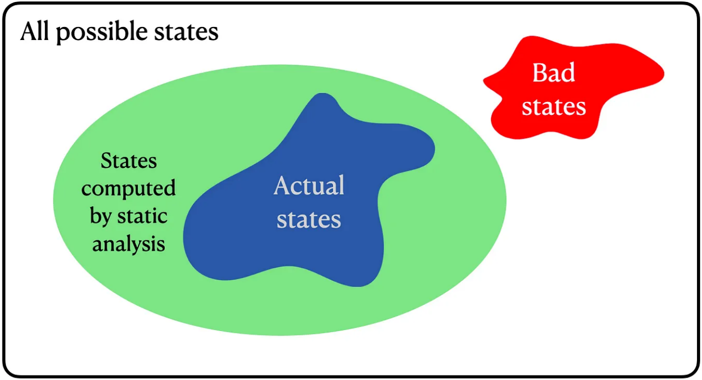
回顾:

- G (y in (0..69))
  - return false
- G (y in (0..71))

Deductive verifiers require annotations (e.g., *loop invariants*) from users

- Fortunately, many techniques that can automatically learn *loop invariants*
- A common framework for this purpose is *Abstract Interpretation* (AI)

- *Abstract interpretation* forms the *basis* of *most static analyzers*
  - A framework for computing ***over-approximations*** of program states



- Cons: *Cannot* reason about the *exact* program behavior
- Pros: It *can* be *enough* to prove program correctness

- Motivation Example: Insertion Sort

```
for i=1 to 99 do

    p := T[i]; j := i+1;

    while j <= 100 and T[j] < p do

        T[j-1] := T[j]; j := j+1;

    end;

    T[j-1] := p;
end;
```

问: Is there any out of bound array access?

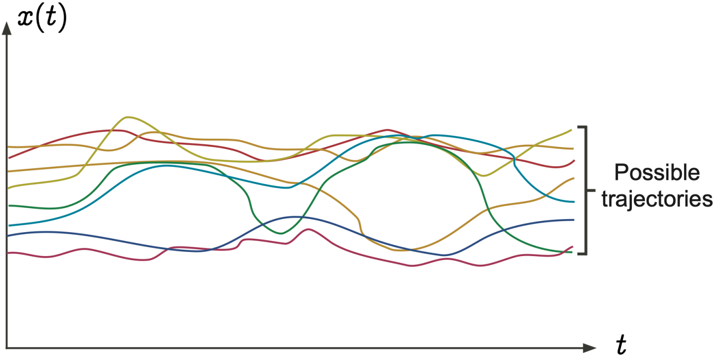
- Motivation Example: Insertion Sort

```
for i=1 to 99 do
    // i ∈ [1, 99]
    p := T[i]; j := i+1;
    // i ∈ [1, 99], j ∈ [2, 100]
    while j <= 100 and T[j] < p do
        // i ∈ [1, 99], j ∈ [2, 100]
        T[j-1] := T[j]; j := j+1;
        // i ∈ [1, 99], j ∈ [3, 101]
    end;
    // i ∈ [1, 99], j ∈ [2, 101]
    T[j-1] := p;
end;
```

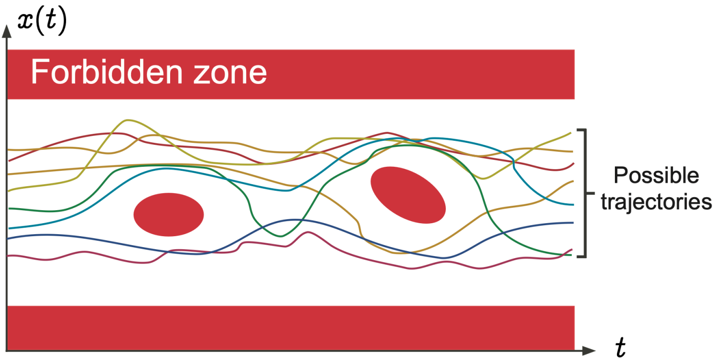
问: Is there any out of bound array access?
答: No

- by *interval analysis*, using an AI tool, e.g., *Apron*

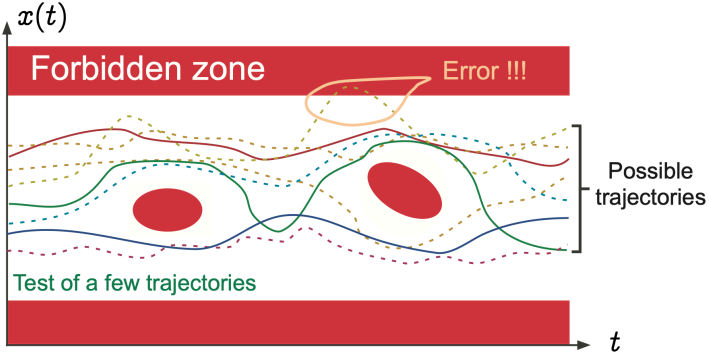
- Graphic Example:



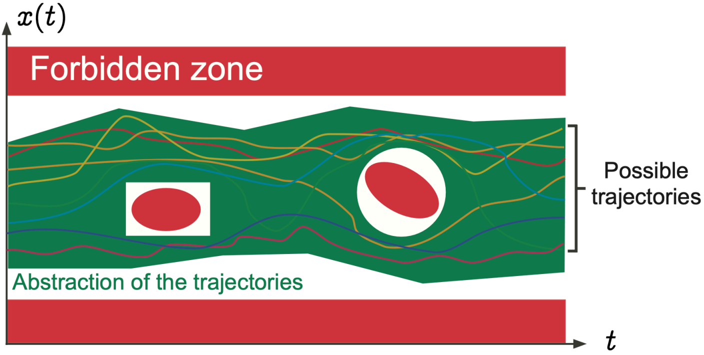
- Graphic Example: Specification



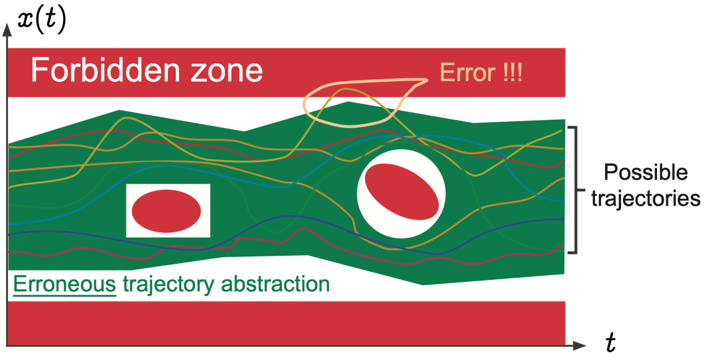
- Graphic Example: Test – main problem: absence of coverage



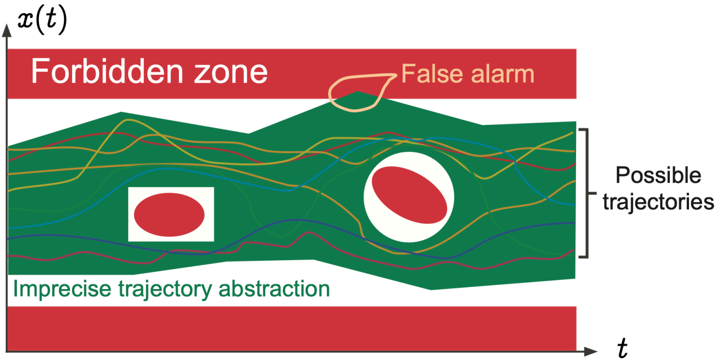
- Graphic Example: Abstract interpretation – Soundness



- Graphic Example: Erroneous abstraction – Unsound



- Graphic Example: Imprecision – False alarms



The AI Recipe

1. Define ***abstract domain*** - fixes "shape" of the invariants
   - e.g., $c_1 \leq x \leq c_2$ (*intervals*), or $\pm x \pm y \leq c$ (octagons)
2. Define ***abstract semantics (transformers)***
   - Define how to symbolically execute each statement in the chosen abstract domain
   - Must be sound wrt to concrete semantics
3. Iterate abstract transformers until ***fixed point***
   - The fixed-point is an over-approximation of program behavior

The AI Recipe

1. Define ***abstract domain*** - fixes "shape" of the invariants
   - e.g., $c_1 \le x \le c_2$ (*intervals*), or $\pm x \pm y \le c$ (octagons)
2. Define ***abstract semantics (transformers)***
   - Define how to symbolically execute each statement in the chosen abstract domain
   - Must be sound wrt to concrete semantics
3. Iterate abstract transformers until ***fixed point***
   - The fixed-point is an over-approximation of program behavior

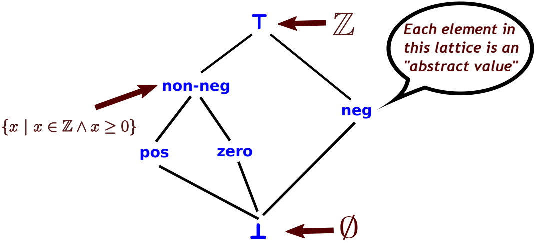
Abstract interpretation provides a recipe for computing over-approximations of program behavior

Simple Example: Sign Domain

Suppose we want to infer invariants of the form $x \bowtie 0$ where

- $\bowtie \in \{\geq, =, >, <\}$
- i.e., zero, non-negative, positive, negative

This corresponds to the following abstract domain represented as *lattice*:



Lattice is a *partially ordered set*

- i.e., $(S, \sqsubseteq)$, where
- each pair of elements has
  - a least upper bound
    - i.e., **join** $\sqcup$
  - a greatest lower bound
    - i.e., **meet** $\sqcap$

The "meaning" of abstract domain is given by ***abstraction*** and ***concretization*** functions that relate concrete and abstract values

Concretization function (γ)

It maps each *abstract value* to *sets of concrete elements*

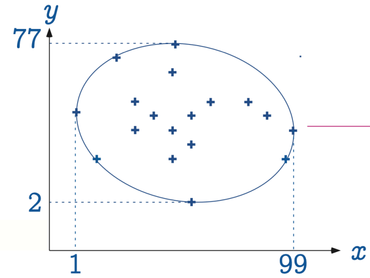
- $\gamma(\textbf{pos}) = \{x \mid x \in \mathbb{Z} \land x > 0\}$

Abstraction function (α)

It maps *sets of concrete elements* to the ***most precise*** *value* in the abstract domain

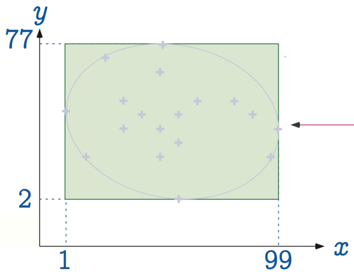
- $\alpha(\{2, 10, 0\}) = \textbf{non-neg}$
- $\alpha(\{3, 99\}) = \textbf{pos}$
- $\alpha(\{-3, 2\}) = \top$

- Interval abstraction α



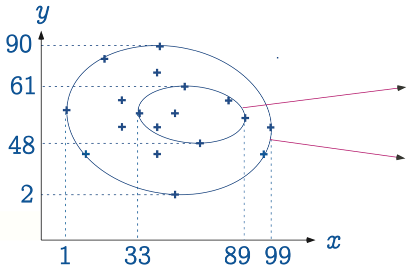
$\{x : [1, 99], y : [2, 77]\}$

- Interval concretization γ



$\{x : [1, 99], y : [2, 77]\}$

- Requirement 1: The abstraction α is monotone



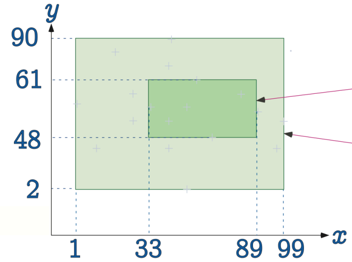
$\{x : [33, 89], y : [48, 61]\}$

$\sqsubseteq$

$\{x : [1, 99], y : [2, 90]\}$

$$X \subseteq Y \Rightarrow \alpha(X) \sqsubseteq \alpha(Y)$$

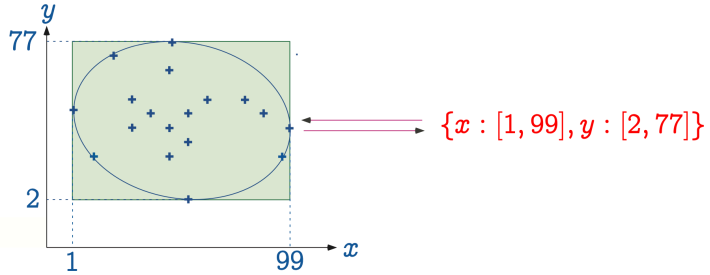
- Requirement 2: The concretization γ is monotone



$$\{x : [33, 89], y : [48, 61]\}$$
$$\sqsubseteq$$
$$\{x : [1, 99], y : [2, 90]\}$$

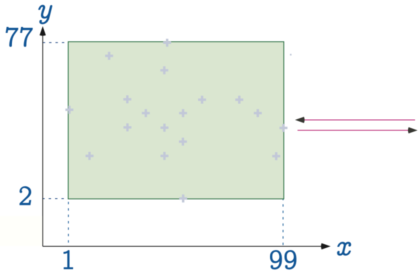
$$X \sqsubseteq Y \Rightarrow \gamma(X) \subseteq \gamma(Y)$$

- Requirement 3: The $\gamma \circ \alpha$ composition is extensive



$$X \subseteq \gamma \circ \alpha(X)$$

- Requirement 4: The $\alpha \circ \gamma$ composition is reductive



$\{x : [1, 99], y : [2, 77]\}$

$=/\sqsubseteq$

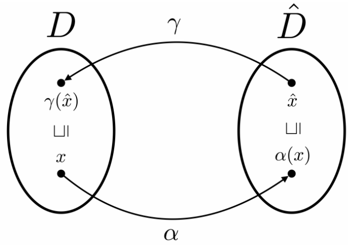
$\{x : [1, 99], y : [2, 77]\}$

$$\alpha \circ \gamma(Y) =/\sqsubseteq Y$$

Total requirement: concrete domain D and abstract domain $\hat{D}$ must be related through *Galois connection*:

Galois connection

$$\forall x \in D, \forall \hat{x} \in \hat{D}. \alpha(x) \sqsubseteq \hat{x} \Leftrightarrow x \sqsubseteq \gamma(\hat{x})$$

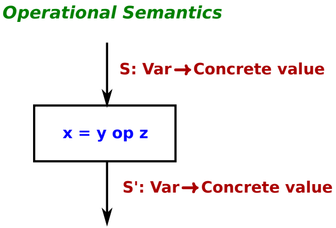


Intuitively, this says that α, γ respect the orderings of $D, \hat{D}$

Step 2: Abstract Semantics

Define *abstract* transformers (i.e., *semantics*) for each statement, given abstract domain, α, γ

- Describes how statements affect our abstraction
- Abstract counter-part of *operational semantics* rules



- Back to Our Example, we can define abstract transformer for $x = y + z$ as follows:

	pos	neg	zero	non-neg	$\top$	$\bot$
pos	pos	$\top$	pos	pos	$\top$	$\bot$
neg	$\top$	neg	neg	$\top$	$\top$	$\bot$
zero	pos	neg	zero	non-neg	$\top$	$\bot$
non-neg	pos	$\top$	non-neg	non-neg	$\top$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

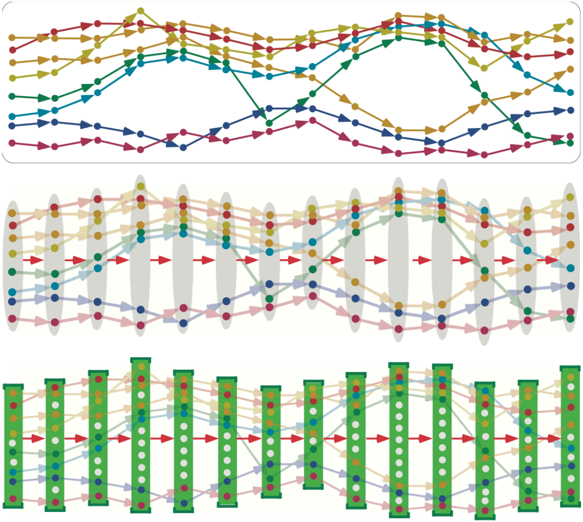
Set of traces

⇓

Traces of sets

⇓

Trace of Intervals



Soundness of Abstract Transformers

- *Total requirement*: Abstract semantics must be *sound* wrt (i.e., faithfully models) the concrete semantics

Soundness of $\hat{F}$

If F is the concrete transformer and $\hat{F}$ is its abstract counterpart, soundness of $\hat{F}$ means:

$$\forall x \in D, \forall \hat{x} \in \hat{D}. \alpha(x) \sqsubseteq \hat{x} \Rightarrow \alpha(F(x)) \sqsubseteq \hat{F}(\hat{x})$$

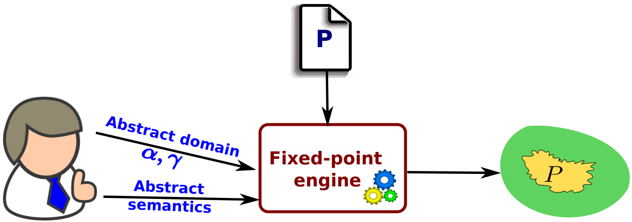
Note: recall Galois connection

- In other words, If $\hat{x}$ is an overapproximation of x, then $\hat{F}(\hat{x})$ is an over-approximation of $F(x)$

Step 3: Fixed-Point Computation

Repeated symbolic execution of the program using abstract semantics until our approximation of the program reaches an *equilibrium*:

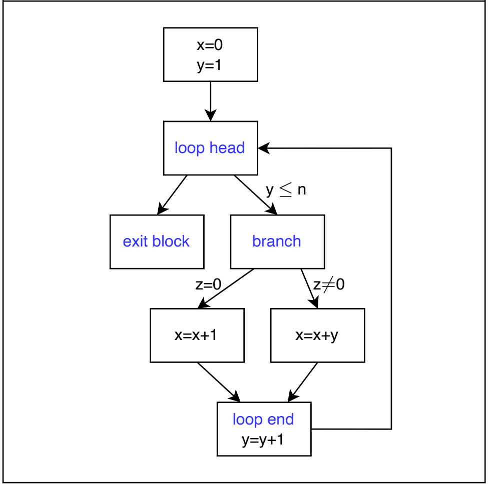
$$\bigsqcup_{i \in \mathbb{Z}} \hat{F}^i(\bot)$$



An example:

```
x = 0;
y =1;
while(y <= n) {
    if (z == 0) {
        x = x+1;
    }
    else {
        x = x + y;
    }
    y = y+1;
}
```

- Specification: Is x *always non-negative* inside the loop?



Control Flow Graph

Least fixed-point

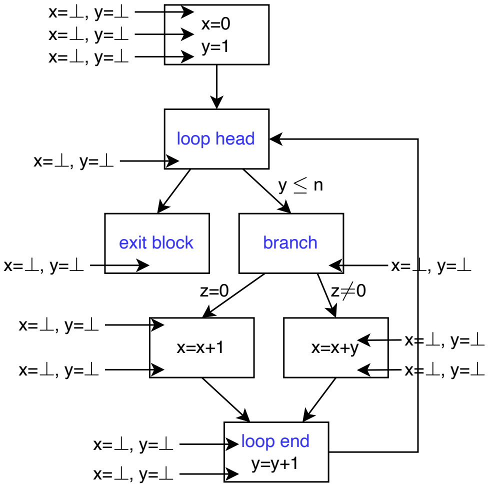
- *Start with underapproximation*
- Want to compute abstract values at every program point
- Grow the approximation until it stops growing

Least fixed-point

1. *Initialize* all abstract states to $\bot$
2. *Repeat* until no abstract state changes at any program point:
   - Compute abstract state on entry to *a basic block B* by taking the *join* of B's *predecessors*
   - *Symbolically execute* each basic block using abstract semantics

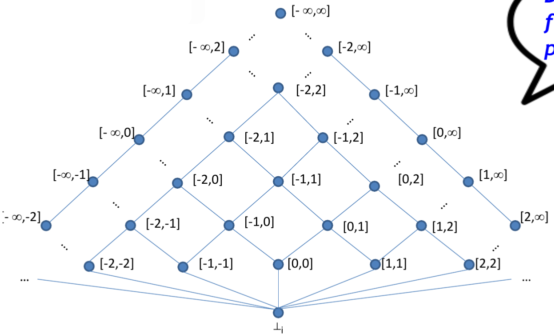
- 性质: *Assuming correctness* of your abstract semantics, the *least fixed point* is an *overapproximation* of the program!

Control Flow Graph - Initialization



Interval Analysis

- In the interval domain, abstract values are of the form $[c_1, c_2]$ where c_1 is a lower bound and c_2 has an upper bound
- If the abstract value for x is $[1, 3]$ at some program point P, this means $1 \leq x \leq 3$ is an *invariant* of P



Requirements on **widening** ∇ operator

1. $\forall a, b \in \hat{D}. a \sqcup b \sqsubseteq a \nabla b$
2. For all increasing chains $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$, the ascending chains $d_0^\nabla \sqsubseteq d_1^\nabla \sqsubseteq \ldots$ eventually *stabilizes* where $d_0^\nabla = d_0$ and

$$d_{i+1}^\nabla = d_i^\nabla \nabla d_{i+1}$$

性质

- Overapproximate least-fixed-point by using widening operator rather than *join*
  - Sound and guaranteed to *terminate*
- This is called *post-fixed-point*

Requirements on widening ∇ operator

1. $\forall a, b \in \hat{D}. a \sqcup b \sqsubseteq a \nabla b$
2. For all increasing chains $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$, the ascending chains $d_0^\nabla \sqsubseteq d_1^\nabla \sqsubseteq \ldots$ eventually *stabilizes* where $d_0^\nabla = d_0$ and

$$d_{i+1}^\nabla = d_i^\nabla \nabla d_{i+1}$$

性质

- Overapproximate least-fixed-point by using widening operator rather than *join*
  - Sound and guaranteed to *terminate*
- This is called *post-fixed-point*

If abstract domain does not have this property, we need a widening ∇ operator that forces convergence

例: **Widening** in **Interval Domain**

$$[a,b]\nabla\bot = [a,b]$$
$$\bot\nabla[a,b] = [a,b]$$
$$[a,b]\nabla[c,d] = [(c<a? -\infty : a), (b<d? +\infty : b)]$$

作业 1

- $[2,3]\nabla[1,3] =$
- $[1,4]\nabla[2,3] =$
- $[2,6]\nabla[2,6] =$
- $[3,4]\nabla[3,5] =$

例: Widening in Interval Domain

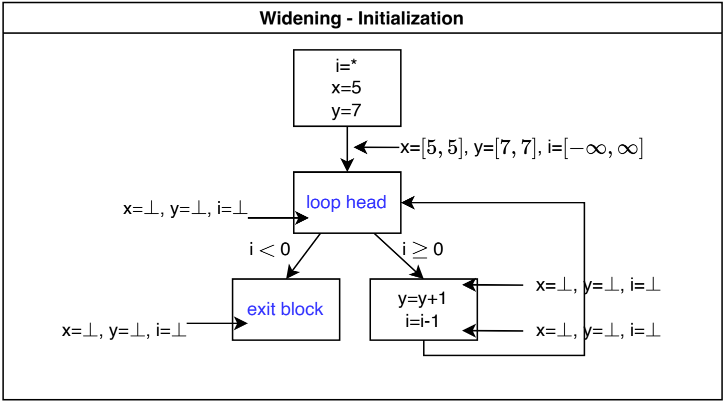
$$
\begin{aligned}
[a,b]\triangledown\bot &= [a,b] \\
\bot\triangledown[a,b] &= [a,b] \\
[a,b]\triangledown[c,d] &= [(c<a?-\infty:a),(b<d?+\infty:b)]
\end{aligned}
$$

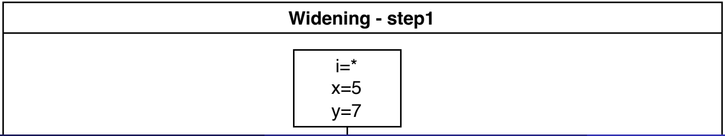
作业 1

- $[2,3]\triangledown[1,3] =$
- $[1,4]\triangledown[2,3] =$
- $[2,6]\triangledown[2,6] =$
- $[3,4]\triangledown[3,5] =$

For the interval domain, we *can* define the simple widening operator.

Example with widening





- 问题: In many cases, *widening overshoots* and generates imprecise results
- 例:

```
x=1;
while(*) {
    x = 2;
}
```

- After widening, x's abstract value will be $[1, \infty]$ after the loop; but more precise value is $[1, 2]$
- 解决方法: After finding a post-fixed-point (using widening), have a second pass using a *narrowing* operator

Requirements on **Narrowing** $\triangle$ operator (Recall Widening $\triangledown$)

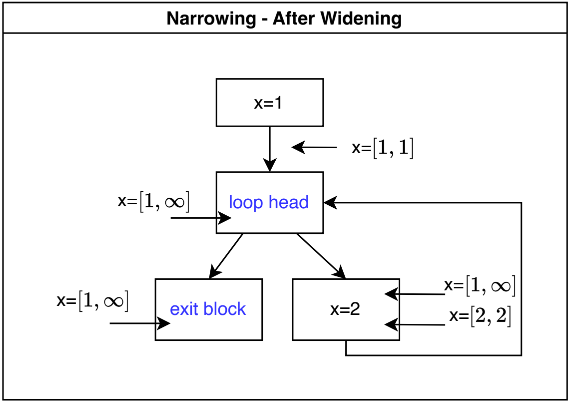
1. $\forall x, y \in \hat{D}.(y \sqsubseteq x) \Rightarrow y \sqsubseteq (x \triangle y) \sqsubseteq x$
2. For all decreasing chains $x_0 \sqsupseteq x_1 \sqsupseteq \ldots$, the chains $y_0, y_1, \ldots$ eventually *stabilizes* where $y_0 = x_0$ and

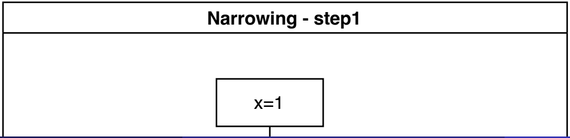
$$y_{i+1} = y_i \triangle x_{i+1}$$

例: **Narrowing** in **Interval Domain**

$$[a,b] \triangle \bot = [a,b]$$
$$\bot \triangle [a,b] = [a,b]$$
$$[a,b] \triangle [c,d] = [(a = -\infty ? c : a), (b = \infty ? d : b)]$$

Example with narrowing





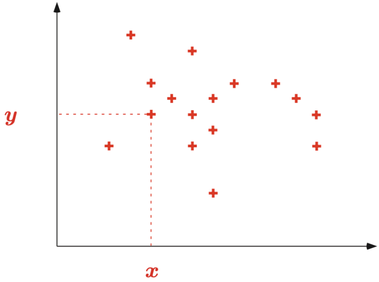
- Both the sign and interval domain are *non-relational domains*
  - i.e., do not relate different program variables
- *Relational domains* track relationships between variables
  - more powerful
- A motivating example

```
x=0; y=0;
while(*) {
  x = x+1; y = y+1;
}
assert(x=y);
```

- *Cannot* prove this assertion using *interval domain*

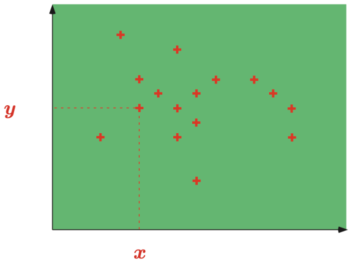
- *Karr's domain*: Tracks equalities between variables (e.g., $x = 2y + z$)
- *Octagon domain*: Constraints of the form $\pm x \pm y \leq c$
- *Polyhedra domain*: Constraints of the form $c_1 x_1 + \ldots c_n x_n \leq c$
- Polyhedra domain most precise among these, but can be expensive (exponential complexity)
- Octagons less precise but cubic time complexity

- Approximations of an [in]finite set of points



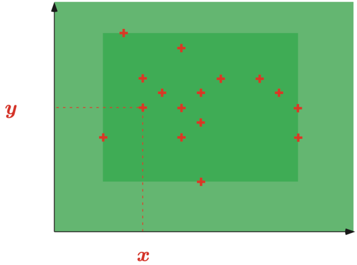
$$\{\ldots, \langle 19,\ 77\rangle, \ldots, \langle 20,\ 03\rangle, \ldots\}$$

- Effective computable approximations of an [in]finite set of points:
- *Signs*



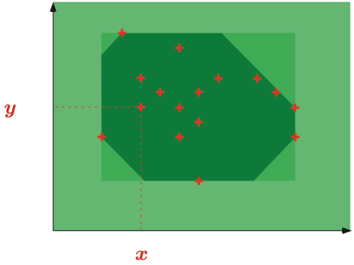
$$\begin{cases} x \geq 0 \\ y \geq 0 \end{cases}$$

- Effective computable approximations of an [in]finite set of points:
- *Intervals*



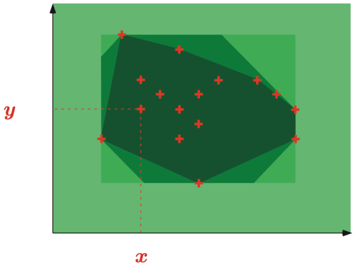
$$\begin{cases} x \in [19,\ 77] \\ y \in [20,\ 03] \end{cases}$$

- Effective computable approximations of an [in]finite set of points:
- *Octagons*



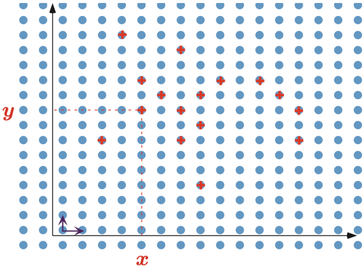
$$\begin{cases} 1 \le x \le 9 \\ x + y \le 77 \\ 1 \le y \le 9 \\ x - y \le 99 \end{cases}$$

- Effective computable approximations of an [in]finite set of points:
- *Polyhedra*



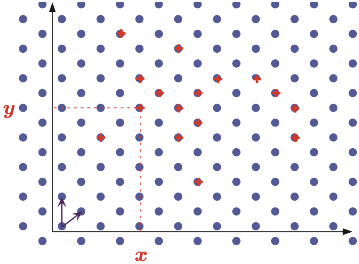
$$\begin{cases} 19x + 77y \leq 2004 \\ 20x + 03y \geq 0 \end{cases}$$

- Effective computable approximations of an [in]finite set of points:
- *Simple congruences*



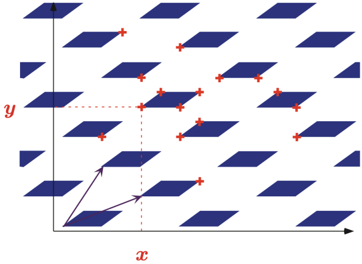
$$\begin{cases} x = 19 \bmod 77 \\ y = 20 \bmod 99 \end{cases}$$

- Effective computable approximations of an [in]finite set of points:
- *Linear congruences*



$$\begin{cases} 1x + 9y = 7 \bmod 8 \\ 2x - 1y = 9 \bmod 9 \end{cases}$$

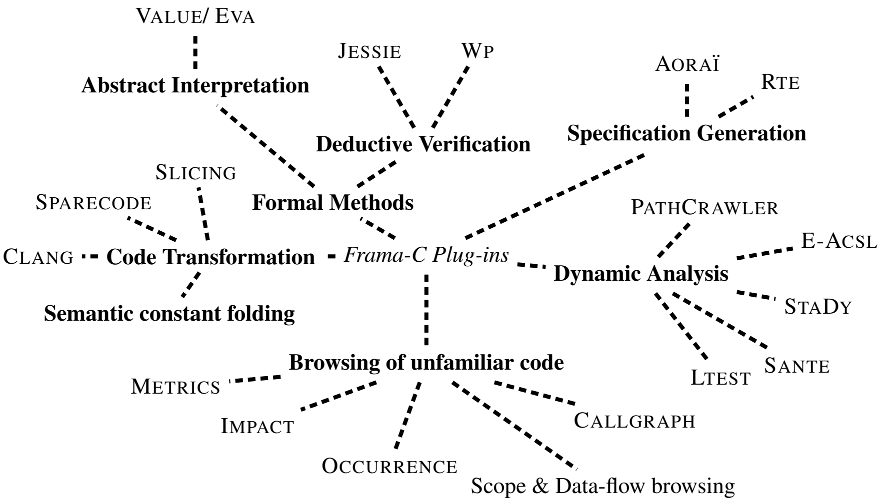
- Effective computable approximations of an [in]finite set of points:
- *Trapezoidal linearcongruences*



$$\begin{cases} 1x + 9y \in [0, 77] \bmod 10 \\ 2x - 1y \in [0, 99] \bmod 11 \end{cases}$$

Tools supporting Abstract Interpretation

- FRAMA-C
  - https://frama-c.com/
  - https://frama-c.com/download/frama-c-user-manual.pdf
  - *Eva, an Evolved Value Analysis*
    - https://frama-c.com/fc-plugins/eva.html
- Infer
  - https://fbinfer.com/
  - *Infer.AI framework*
    - https://fbinfer.com/docs/absint-framework/
- Apron
  - https://antoinemine.github.io/Apron/doc/


Value/ Eva
Abstract Interpretation
Jessie
Wp
Deductive Verification
Aoraï
Rte
Specification Generation
Slicing
Sparecode
Formal Methods
PathCrawler
E-Acsl
Clang
Code Transformation
Frama-C Plug-ins
Dynamic Analysis
StaDy
Semantic constant folding
Browsing of unfamiliar code
Sante
Ltest
Metrics
Impact
Callgraph
Occurrence
Scope & Data-flow browsing

Revisit the widening example:

```
wenchao@wenchao-Teaching:~/code/frama-c$ frama-c -eva loop.c
[kernel] Parsing loop.c (with preprocessing)
[eva] Analyzing a complete application starting at main
[eva:initial-state] Values of globals at initialization

[eva] loop.c:6: starting to merge loop iterations
[eva:alarm] loop.c:7: Warning: signed overflow. assert y + 1 ≤ 2147483647;
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function main:
  i ∈{-1}
  x ∈{5}
  y ∈[7..2147483647]
```

实验小作业 3： 自己编写个一程序，使用 frama-c 的 eva 功能来分析一下这个程序，给出实验报告。

A method of software model checking:

- Counterexample-Guided Abstraction Refinement (*CEGAR*)

Keywords:

- Predicate Abstraction
  - Predicate Abstraction Lattice
- Abstract Transformers
  - Strongest Postcondition
- Refinement

Example

```
x:=0;  y:=  0;
while(x<100)
{
      x  :=  x+1;
      y  :=  y+1;
}
assert(y  =  100);
```

- Predicate set $\mathcal{P} = \{x < 100, y = 100\}$

Predicate Abstraction

Given a set of predicates $\mathcal{P} = \{p_1, \ldots, p_n\}$, predicate abstraction computes for *every program location*, an abstract value $[b_1, \ldots, b_n]$ where:

- b_i indicates whether p_i holds or not at that location
- values of b_i drawn from the set $\{0, 1, *\}$ where $*$ indicates unknown

- In the example, at Line 1, $[b_1, b_2] = [1, 0]$
- In other words, we have an abstract domain where each element is a formula $\bigwedge_i l_i$ (sometimes called a ***cube***), where $l_i = p_i \mid \neg p_i$

性质: Predicate Abstraction Lattice

Given predicates $\mathcal{P}$, $(Cubes(\mathcal{P}), \Rightarrow)$ forms a complete lattice

- $Cubes(\mathcal{P})$ is any formula $\bigwedge_i p_i$ where p_i is a predicate or the negation of a predicate in $\mathcal{P}$

- In other words, we have $\phi_1 \sqsubseteq \phi_2$ iff $\phi_1 \Rightarrow \phi_2$
  - e.g., $p_1 \wedge p_2 \sqsubseteq p_1$
- 练习: How do we compute $\phi_1 \sqcup \phi_2$?,
  - $(p_1 \wedge p_2) \sqcup p_1$?
  - $(p_1 \wedge p_2) \sqcup \neg p_1$

Abstract Transformers

Given a statement S and cube ϕ, define abstract transformer *post*$^{\#}(S, \phi)$ to be the strongest cube ϕ' over $\mathcal{P}$ such that:

$$\mathrm{sp}(S, \phi) \Rightarrow \phi'$$

where sp is the *strongest post-condition* of S wrt to ϕ

Strongest Postcondition $\mathrm{sp}(S, \phi)$

Executing statement S on any state s_0 in the ϕ region must result in a state s in the $\mathrm{sp}(S, \phi)$ region

- $\mathrm{sp}(\mathrm{assume}\ c, \phi) \Leftrightarrow c \wedge \phi$
- $\mathrm{sp}(v := e[v], \phi[v]) \Leftrightarrow \exists v_0 . v = e[v_0] \wedge \phi[v_0]$

Abstract Transformers

Given a statement S and cube ϕ, define abstract transformer *post*$^{\#}(S, \phi)$ to be the strongest cube ϕ' over $\mathcal{P}$ such that:

$$\mathrm{sp}(S, \phi) \Rightarrow \phi'$$

where sp is the *strongest post-condition* of S wrt to ϕ

Strongest Postcondition $\mathrm{sp}(S, \phi)$

Executing statement S on any state s_0 in the ϕ region must result in a state s in the $\mathrm{sp}(S, \phi)$ region

- $\mathrm{sp}(\mathrm{assume}\ c, \phi) \Leftrightarrow c \land \phi$
- $\mathrm{sp}(v := e[v], \phi[v]) \Leftrightarrow \exists v_0 . v = e[v_0] \land \phi[v_0]$

If s is the current state and $s \vDash \mathrm{sp}(S, \phi)$, then there exists a state s_0 such that executing S on s_0 results in state s and $s_0 \vDash \phi$

Example: Given $\mathcal{P} = \{x = y, x \neq y, x \geq y\}$, compute $post^{\#}(x := x + 1, x = y)$?

The answer is $[b_1, b_2, b_3] = [0, 1, 1]$

- $\text{sp}(x := x + 1, x = y) \Leftrightarrow (\exists x_0 . x = x_0 + 1 \wedge x_0 = y) \Leftrightarrow (x = y + 1)$
- $\phi' \equiv (b_1 = 0 \wedge b_2 = 1 \wedge b_3 = 1)$
  - since $(x = y + 1) \Rightarrow \phi'$

作业 *4*: Practice in program, compute

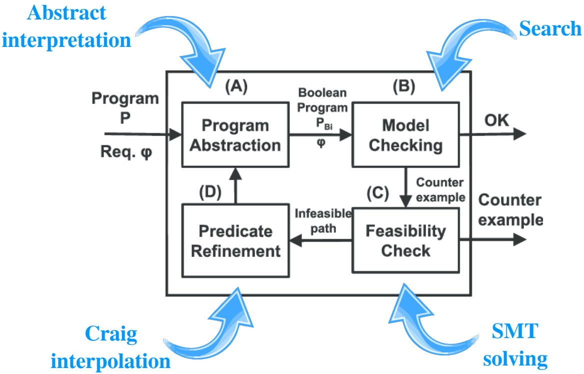
- $post^{\#}(x := x + 1, x < 100)$
- $post^{\#}(x := x + 1, x < 100 \wedge y = 100)$

Motivation for CEGAR

- Predicate abstraction is very *sensitive* to the set of predicates
  - If you choose the right set, verification succeeds; otherwise, it fails
- The *CEGAR* paradigm allows *automatically* and *iteratively discovering* the *right set* of predicates

Key Steps:

- Program Abstraction
- Model Checking
- Feasibility Check
- Refinement

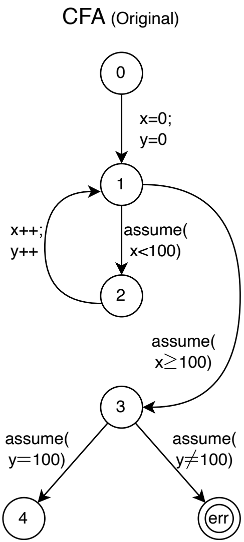


Motivation for *Program Abstraction*

- Given a program P, the state is a tuple $l, v_1, v_2, \ldots, v_n$, where
  - l is the ***control*** *location*
  - v_i denotes the value of ith variable
- The *state space* is large or even infinite!

Idea: construct a so-called ***boolean program*** via *predicate abstraction*

- Replace *concrete states* with *predicates*
- Operate over control-flow automaton (*CFA*)
  - Like CFG but nodes/edges are flipped + explicit error locations



Program Abstraction

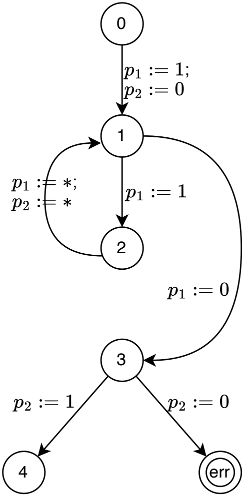
- state space
  - From $6 * 2^{32} * 2^{32}$ to $6 * 2 * 2$

问: How to translate the program into a boolean program?

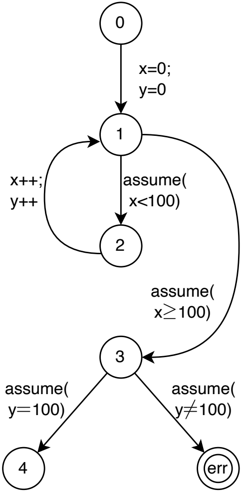
作业 5: Translate statements in CFA

- $1 \rightarrow 2$
- $2 \rightarrow 1$

CFA (Predicate Abstraction)



CFA (Original)



Translating Statements

Given statement S and boolean b representing predicate p,

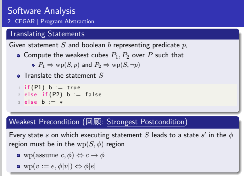
- Compute the weakest cubes P_1, P_2 over P such that
  - $P_1 \Rightarrow \text{wp}(S, p)$ and $P_2 \Rightarrow \text{wp}(S, \neg p)$
- Translate the statement S

```
if(P1) b := true
else if(P2) b := false
else b := *
```

Weakest Precondition (回顾: Strongest Postcondition)

Every state s on which executing statement S leads to a state s' in the ϕ region must be in the $\text{wp}(S, \phi)$ region

- $\text{wp}(\text{assume } c, \phi) \Leftrightarrow c \rightarrow \phi$
- $\text{wp}(v := e, \phi[v]) \Leftrightarrow \phi[e]$



If s is the current state and

$$s \vDash sp(S, \phi)$$

then there exists a state s_0 such that executing S on s_0 results in state s and

$$s \vDash \phi$$

- Example: Consider the predicates $\{x > 5, x < 5, y = 5\}$, how to translate the statement $x := y$ in the boolean program with variables b_1, b_2, b_3?

- $\text{wp}(x := y, x > 5) \Leftrightarrow (y > 5)$
- $\text{wp}(x := y, x < 5) \Leftrightarrow (y < 5)$
- $\text{wp}(x := y, y = 5) \Leftrightarrow (y = 5)$

- For b_1, since $b_3 \Rightarrow \neg(y > 5)$, translation: if (b_3) $b_1 := 0$, else $b_1 := *$
- For b_2, since $b_3 \Rightarrow \neg(y < 5)$, translation: if (b_3) $b_2 := 0$, else $b_2 := *$
- For b_3, since $b_3 \Rightarrow (y = 5)$ and $\neg b_3 \Rightarrow \neg(y = 5)$, translation: (Empty)

Totally, the translated statements of $x := y$ are

```
    if (b3) b1:=0, b2:=0
    else b1:=*, b2:=*
```

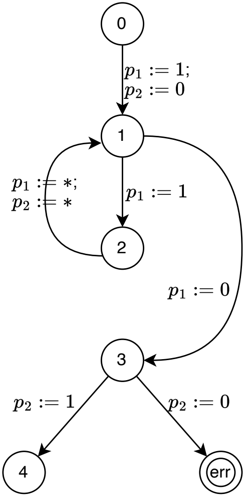
Model Checking

- *Initial states*:

$$(0, p_1, p_2), (0, \neg p_1, p_2), (0, p_1, \neg p_2), (0, \neg p_1, \neg p_2)$$

- There is a *transition* from $(l, b_1, \ldots, b_n)$ to $(l', b'_1, \ldots, b'_n)$ *iff*:
  - There must be a transition from l to l' labeled with S
  - The formula $\mathrm{sp}(S, \bigwedge_i b_i) \wedge \bigwedge_i b'_i$ must be satisfiable. (query SAT solver)

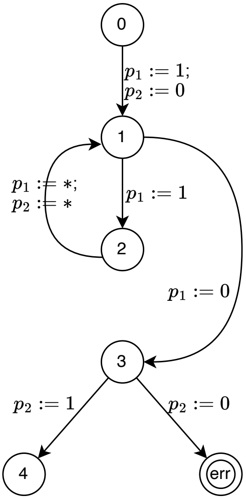
CFA (Predicate Abstraction)



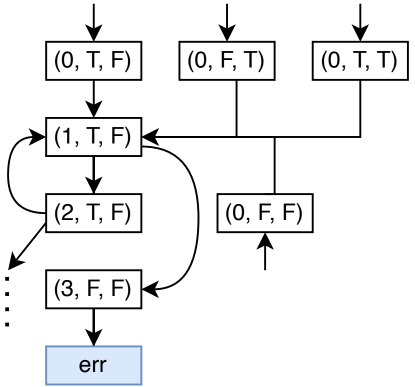
练习: Which of these transition exist in the state transition graph?

- $(1, p_1, p_2)$ to $(3, \neg p_1, p_2)$
  - Yes
- $(1, p_1, p_2)$ to $(3, p_1, p_2)$
  - No
- $(3, \neg p_1, p_2)$ to $(\mathrm{err}, \neg p1, \neg p_2)$
  - Yes

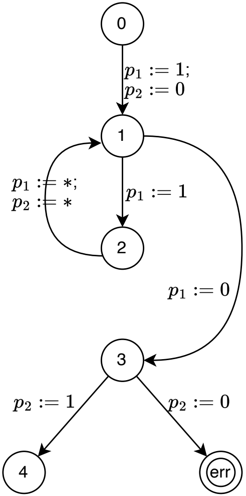
CFA (Predicate Abstraction)



Partial transition system:



CFA (Predicate Abstraction)

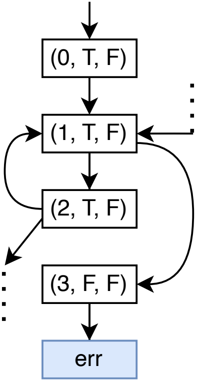


Verification outputs FALSE because *error state* is *reachable*!

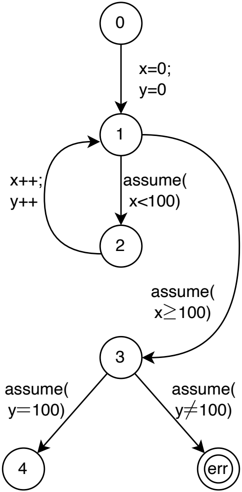
Feasibility Check

- *But* if the error state is reachable, this could be due to *imprecision* in the abstraction
  - i.e., current set of predicates may *not* be *fine-grain enough*
- To decide how to proceed, we need to check if the property is *actually* violated
- Fortunately, the model checker can provide a *counterexample* in the form of a *program trace*!

Transition System



CFA (Original)



Counterexample Trace

```
x := 0; y:=0;
assume(x>=100);
assume(y!=100);
```

Clearly spurious because the trace formula is *UNSAT*:

$$x = 0 \wedge$$
$$y = 0 \wedge$$
$$x \geq 100 \wedge$$
$$y \neq 100$$

Refinement

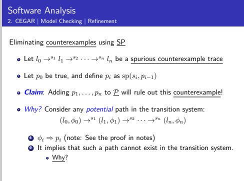
- Goal: *prevent* the model checker from giving the *same* counterexample trace as before
- 问: How do we find predicates that will rule out this spurious trace?
- *Most basic idea*: Compute strongest postcondition for each statement in the counterexample trace; add these to set of predicates!

Eliminating counterexamples using SP

- Let $l_0 \rightarrow^{s_1} l_1 \rightarrow^{s_2} \cdots \rightarrow^{s_n} l_n$ be a spurious counterexample trace
- Let p_0 be true, and define p_i as $\mathrm{sp}(s_i, p_{i-1})$
- ***Claim***: Adding $p_1, \ldots, p_n$ to $\mathcal{P}$ will rule out this counterexample!
- *Why?* Consider any *potential* path in the transition system:

$$(l_0, \phi_0) \rightarrow^{s_1} (l_1, \phi_1) \rightarrow^{s_2} \cdots \rightarrow^{s_n} (l_n, \phi_n)$$

  1. $\phi_i \Rightarrow p_i$ (note: See the proof in notes)
  2. It implies that such a path cannot exist in the transition system.
     - Why?



For any path $(l_0, \phi_0) \to^{s_1} (l_1, \phi_1) \to^{s_2} \cdots \to^{s_n} (l_n, \phi_n)$, we have $\phi_i \Rightarrow p_i$

- *Base case*: Trivial since p_0 is true
- *Induction*:
  - By the inductive hypothesis, we have $\phi_{i-1} \Rightarrow p_{i-1}$
  - By construction of transition systems, $(l_{i-1}, \phi_{i-1}) \to^{s_i} (l_i, \phi_i)$ exists if $\mathrm{sp}(s_i, \phi_{i-1}) \wedge \phi_i$ is satisfiable, which implies SAT$(\mathrm{sp}(s_i, p_{i-1}) \wedge \phi_i)$
  - Furthermore, we have either $\phi_i \Rightarrow p_i$ or $\phi_i \Rightarrow \neg p_i$ - why?
  - But if $\phi_i \Rightarrow \neg p_i$, we'd have UNSAT$(\mathrm{sp}(s_i, p_{i-1}) \wedge \phi_i)$
  - Thus, $\phi_i \Rightarrow p_i$

The *problem* of the most basic idea

- *Only* removes this counterexample trace
- Ideally, we want to learn predicates that allow us to remove *multiple* spurious traces
- ***Trick***: We can learn more general predicates using a technique called ***Craig interpolation***

Craig interpolation: Given an *unsatisfiable* formula $\phi_1 \wedge \phi_2$, a Craig interpolant is a formula ψ such that:

- $\phi_1 \Rightarrow \psi$
- UNSAT $(\phi_2 \wedge \psi)$
- ψ is over the common variables of ϕ_1 and ϕ_2

Interpolant Examples

$$\phi_1 \equiv x \le w \land y \ge w \land z = x$$
$$\phi_2 \equiv y < t \land t = z$$

- Which of the following formulas are interpolants for $\phi_1 \land \phi_2$?
  1. $y \ge z$
  2. $y \ge x \land z = x$
  3. $y > z$

How to learn by Craig interpolation?

- Let $l_0 \rightarrow^{s_1} l_1 \rightarrow^{s_2} \cdots \rightarrow^{s_n} l_n$ be a spurious counterexample trace
- For simplicity, suppose the trace is in SSA form and suppose $\mathrm{enc}(s_i)$ gives logical encoding of s_i's semantics
- Then, we know that the following formula is *UNSAT*:

$$\mathrm{enc}(s_1) \wedge \mathrm{enc}(s_2) \wedge \cdots \wedge \mathrm{enc}(s_n)$$

- Now let ϕ_i^- denote the trace formula *up to* statement i and ϕ_i^+ denote the formula after i
- Then, for each location l_i, we have UNSAT $(\phi_i^- \wedge \phi_i^+)$ and *the interpolant gives predicates that are useful to track at* l_i!

- Consider the following counterexample trace that corresponds to executing loop body *once*:

```
x0:=0; y0:=0;
assume(x0<100);
x1:=x0+1;
y1:=y1+1;
- - - - - - - - - - - - -
assume(x1>=100);
assume(y1!=100);
```

$$\phi_1: \quad x_0 = 0 \land y_0 = 0 \land x_0 < 100 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1$$

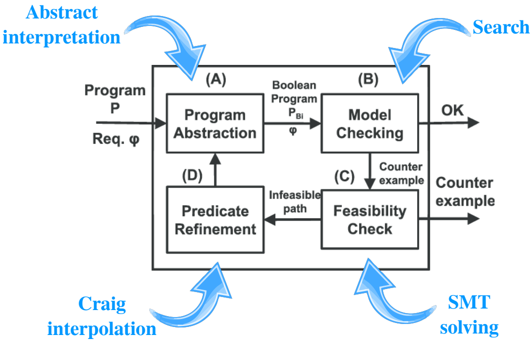
$$\phi_2: \quad x_1 \geq 100 \land y_1 \neq 100$$

- Interpolant: $x_1 = y_1 \land x_1 \leq 100$
- Using the predicates in the interpolant, we *can now verify* the correctness of this program!

Per-location Abstraction

- In the *basic form* of predicate abstraction, we have a global set of predicates that we " track" *everywhere*
  - But not all predicates are useful everywhere...
- ***Observation***: The interpolant tells us *which predicates are useful where!*
- Thus, rather than having a global set of predicates, we can have a *different predicate set* for each *different location*
  - Since the model checker is very *sensitive* to the *number* of predicates, this is really important for scalability

- Summary: (*CEGAR*)



- Can both verify and give counterexamples
- but no termination guarantees...

作业

- 作业 1
- 作业 2: 使用interval domain下的Least Fixed Point算法、Widening算法求出模型中 y 在各点的估计。
- 实验小作业 3
- 作业 4
- 作业 5

本章节大作业参考论文

大作业可参考论文 (但不限于下列论文):

- 经典
  - Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints
- 应用
  - AI^2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation
  - Extracting Protocol Format as State Machine via Controlled Static Loop Analysis
  - Rule-Based Static Analysis of Network Protocol Implementations
  - Precise Enforcement of Progress-Sensitive Security