Abstract Interpretation

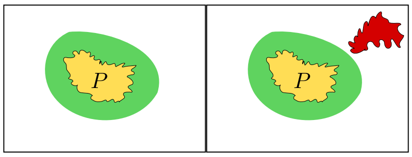
Işıl Dillig

Overview

- Deductive verifiers require annotations (e.g., loop invariants) from user
- 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

Key Idea: Over-approximation

- Abstract interpretation is a framework for computing **over-approximations** of program states



- Cannot reason about the **exact** program behavior due to undecidability (and also for scalability reasons)
- But we can obtain a conservative over-approximation and this can be enough to prove program correctness

Motivating Example

```
proc MC(n:int) returns (r:int)
var t1:int, t2:int;
begin
 if (n>100) then
  r = n-10;
 else
  t1 = n + 11;
  t2 = MC(t1);
  r  = MC(t2);
 endif;
end

var a:int, b:int;
begin
b = MC(a);
end
```

Invariants per program point (automatically computed):

```
top
n-101 ≥ 0
-n+r+10 = 0; n-101 ≥ 0
-n+100 ≥ 0
-n+t1-11 = 0; -n+100 ≥ 0
-n+t1-11 =0; -n+100 ≥ 0;
-n+t2-1 ≥ 0; t2-91  ≥ 0
-n+t1-11=0;-n+100 ≥ 0;-n+t2-1 ≥ 0;
t2-91 ≥ 0; r-t2+10 ≥ 0; r-91 ≥ 0
-n+r+10 ≥ 0; r-91 ≥ 0
top
-a+b+10 ≥ 0; b-91 ≥ 0
```

- What does this function do?
- Annotations computed automatically using an AI tool (Apron)

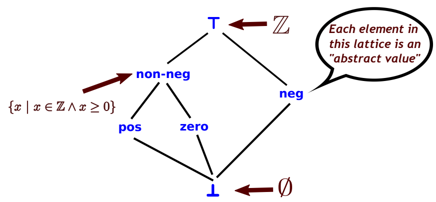
The AI Recipe

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

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

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 $(S, \sqsubseteq)$ where each pair of elements has a least upper bound or **join** ($\sqcup$)

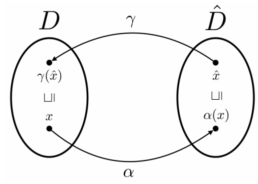
Concretization and Abstraction Functions

- The "meaning" of abstract domain is given by **abstraction** and **concretization** functions that relate concrete and abstract values
- **Concretization function (γ)** maps each abstract value to sets of concrete elements
  - $\gamma(\text{pos}) = \{x \mid x \in \mathbb{Z} \wedge x > 0\}$
- **Abstraction function (α)** maps sets of concrete elements to the most precise value in the abstract domain
  - $\alpha(\{2, 10, 0\}) =$ non-neg
  - $\alpha(\{3, 99\}) =$ pos
  - $\alpha(\{-3, 2\}) = \top$

Requirement: Galois Connection

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

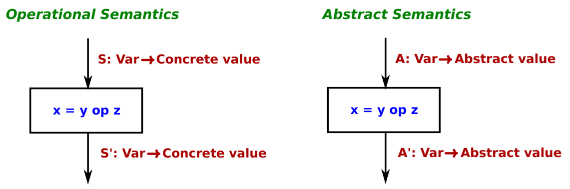
$$\forall x \in D, \forall \hat{x} \in \hat{D}.\ \alpha(x) \sqsubseteq \hat{x} \Leftrightarrow x \sqsubseteq \gamma(\hat{x})$$



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

Step 2: Abstract Semantics

- Given abstract domain, α, γ, need to define abstract transformers (i.e., semantics) for each statement
  - Describes how statements affect our abstraction
  - Abstract counter-part of operational semantics rules



Back to Our Example

- For our sign analysis, we can define abstract transformer for $\mathrm{x} = \mathrm{y} + \mathrm{z}$ 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	$\top$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

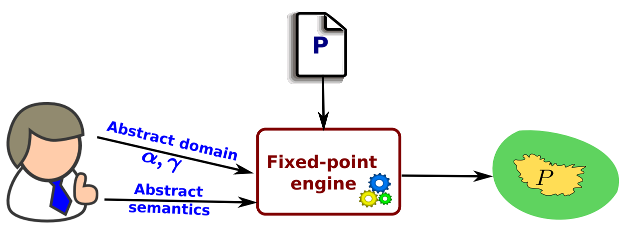
Soundness of Abstract Transformers

- **Important requirement:** Abstract semantics must be **sound** wrt (i.e., faithfully models) the concrete semantics
- If F is the concrete transformer and $\hat{F}$ is its abstract counterpart, soundness of $\hat{F}$ means:

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

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

Putting It All Together

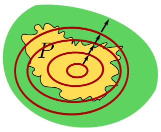


Fixed-point Computations

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

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

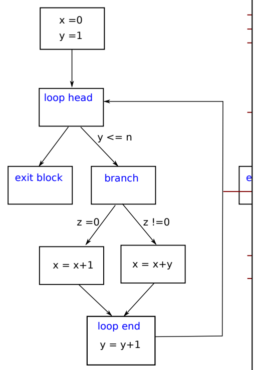
- **Least fixed-point:** Start with underapproximation and grow the approximation until it stops growing



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

Performing Least Fixed Point Computation

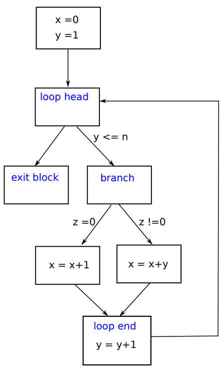
- Represent program as a **control-flow graph**
- Want to compute abstract values at every program point
- Initialize all abstract states to $\bot$
- Repeat until no abstract state changes at any program point:
  - Compute abstract state on entry to a basic block B by taking the **join** of B's predecessors
  - Symbolically execute each basic block using abstract semantics



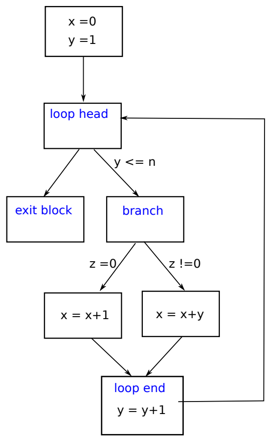
An Example

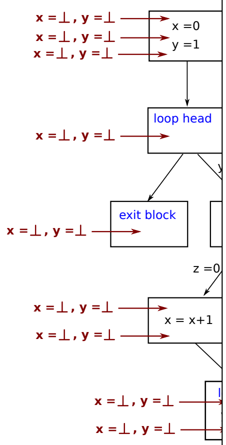
```
x = 0;
y =0;

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

Is x always non-negative inside the loop?



Fixed-Point Computation



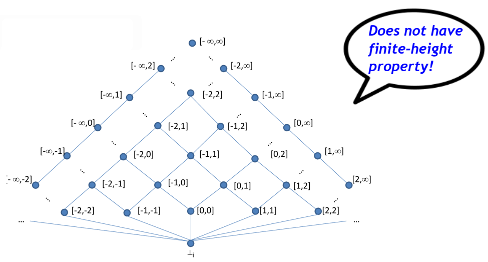


Termination of Fixed Point Computation

- In this example, we quickly reached least fixed point – but does this computation always terminate?
  - Yes if the lattice has finite height; otherwise, it might not
- Unfortunately, many interesting domains do not have this property, so we need widening operators for convergence.

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



Widening

- If abstract domain does not have this property, we need a **widening** ∇ operator that forces convergence
- Conditions on ∇:
  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 chaing $d_0^\nabla \sqsubseteq d_1^\nabla \sqsubseteq \ldots$ eventually stabilizes where $d_0^\nabla = d_0$ and

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

- Overapproximate lfp by using widening operator rather than join $\Rightarrow$ sound and guaranteed to terminate
- This is called **post-fixed-point**

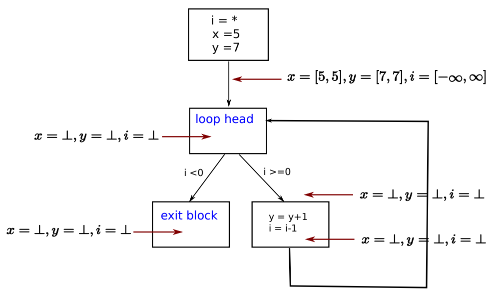
Widening in Interval Domain

- For the interval domain, we can define the following simple widening operator:

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

- $[1,2] \nabla [0,2] =$
- $[0,2] \nabla [1,2] =$
- $[1,5] \nabla [1,5] =$
- $[2,3] \nabla [2,4] =$

Example with Widening



Motivation for Narrowing

- In many cases, widening overshoots and generates imprecise results
- Consider this example:

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

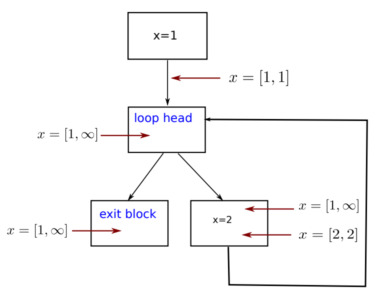
- After widening, x's abstract value will be $[1, \infty]$ after the loop; but more precise value is $[1, 2]$

Narrowing

- **Idea:** After finding a post-fixed-poing (using widening), have a second pass using a **narrowing** operator
- Narrowing operator $\triangle$ must satisfy the following conditions:
  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 sequence $y_0 = x_0, \ldots y_{i+1} = y_i \triangle x_{i+1}$ converges
- For interval domain, we can define $\triangle$ as follows:

$$\begin{array}{rcl} [a,b] \triangle \bot & = & \bot \\ \bot \triangle [a,b] & = & \bot \\ [a,b] \triangle [c,d] & = & [(a = -\infty ? c : a), (b = \infty ? d : b)] \end{array}$$

Example with Narrowing



Relational Abstract Domains

- Both the sign and interval domain are **non-relational** domains (i.e., do not relate different program variables)
- **Relational domains** track relationships between variables and are 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

Examples of Relational Domains

- **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

Message from Patrick Cousot