Static Analysis by Abstract Interpretation

Aloysius Goo

Orcacode, August 18, 2026

Table of Contents

Table of Contents

What is static analysis?

Static analysis (specifically static **program** analysis) aims to analyse and figure out properties about a program without running it (hence "static").

What is static analysis?

Static analysis (specifically static **program** analysis) aims to analyse and figure out properties about a program without running it (hence "static").

Rice's Theorem

All non-trivial semantic properties of programs are undecidable.

What is static analysis?

Static analysis (specifically static **program** analysis) aims to analyse and figure out properties about a program without running it (hence "static").

Rice's Theorem

All non-trivial semantic properties of programs are undecidable.

Static analysis "cheats" by approximating the property, e.g. giving more than one possible outcome

Table of Contents

Why static analysis?

Static analysis can provide solutions to some issues programmers are interested in:

Why static analysis?

Static analysis can provide solutions to some issues programmers are interested in:

- Not writing bugs

Why static analysis?

Static analysis can provide solutions to some issues programmers are interested in:

- Not writing bugs
- Rich IDE features

Why static analysis?

Static analysis can provide solutions to some issues programmers are interested in:

- Not writing bugs
- Rich IDE features
- Type checking and inference

Why static analysis?

Static analysis can provide solutions to some issues programmers are interested in:

- Not writing bugs
- Rich IDE features
- Type checking and inference
- Faster compiled code

Why static analysis?

Static analysis can provide solutions to some issues programmers are interested in:

- Not writing bugs
- Rich IDE features
- Type checking and inference
- Faster compiled code
- Solve the halting problem?

Example: Halt analysis

```
x = input()
if x == "67":
  print("haha")
else:
  print("tung tung tung sahur")
```

Example: Halt analysis

```
x = input()
if x == "67":
  print("haha")
else:
  print("tung tung tung sahur")
```

A simple static analysis can decide that both code paths will halt.

Example: Halt analysis

```
x = 100
while x > 67:
  x = x + 1
```

Example: Halt analysis

```
x = 100
while x > 67:
  x = x + 1
```

A simple static analysis can decide that the while loop is unescapable

Example: Halt analysis

```
def collatz(n):
  if n == 1:
    return n

  if n % 2 == 0:
    return collatz(n // 2)
  else:
    return collatz(3 * n + 1)

collatz(int(input()))
```

Example: Halt analysis

```
def collatz(n):
  if n == 1:
    return n

  if n % 2 == 0:
    return collatz(n // 2)
  else:
    return collatz(3 * n + 1)

collatz(int(input()))
```

No analysis can figure out whether the collatz conjecture is true or false (as of today), hence the analysis can only say the program will **maybe** halt.

Soundness and completeness

One way of achieving decidability is by over approximating on the set of possible outcomes. e.g. if a program has 3 possible outcomes $\{O_1, O_2, O_3\}$, an analysis may over approximate and give $\{O_1, O_2, O_3, O_4, O_5\}$

Soundness and completeness

One way of achieving decidability is by over approximating on the set of possible outcomes. e.g. if a program has 3 possible outcomes $\{O_1, O_2, O_3\}$, an analysis may over approximate and give $\{O_1, O_2, O_3, O_4, O_5\}$

Soundness

An analysis method is sound if it never excludes an actual possible behaviour of the program

Soundness and completeness

One way of achieving decidability is by over approximating on the set of possible outcomes. e.g. if a program has 3 possible outcomes $\{O_1, O_2, O_3\}$, an analysis may over approximate and give $\{O_1, O_2, O_3, O_4, O_5\}$

Soundness

An analysis method is sound if it never excludes an actual possible behaviour of the program

Completeness

An analysis method is complete if it does not output a behaviour that can never happen*

Soundness and completeness

One way of achieving decidability is by over approximating on the set of possible outcomes. e.g. if a program has 3 possible outcomes $\{O_1, O_2, O_3\}$, an analysis may over approximate and give $\{O_1, O_2, O_3, O_4, O_5\}$

Soundness

An analysis method is sound if it never excludes an actual possible behaviour of the program

Completeness

An analysis method is complete if it does not output a behaviour that can never happen*

An analysis method can not be both sound and complete, typically most analysis choose soundness at the cost of completeness.

Table of Contents

How to do static analysis

In general, a static analysis will step through the program and mimic its execution based on the semantics of the language.

How to do static analysis

In general, a static analysis will step through the program and mimic its execution based on the semantics of the language.
Some things we need to consider:

How to do static analysis

In general, a static analysis will step through the program and mimic its execution based on the semantics of the language.
Some things we need to consider:

- How to model the semantics of the language

How to do static analysis

In general, a static analysis will step through the program and mimic its execution based on the semantics of the language.
Some things we need to consider:

- How to model the semantics of the language
- How to represent state while stepping through the language

How to do static analysis

In general, a static analysis will step through the program and mimic its execution based on the semantics of the language.
Some things we need to consider:

- How to model the semantics of the language
- How to represent state while stepping through the language
- How to guarantee the analysis will find a correct approximation

How to do static analysis

In general, a static analysis will step through the program and mimic its execution based on the semantics of the language.
Some things we need to consider:

- How to model the semantics of the language
- How to represent state while stepping through the language
- How to guarantee the analysis will find a correct approximation
- How to guarantee the analysis will terminate

Table of Contents

Partial order

A partially ordered set (poset) is a set with a partial order ($\leq$) on it.

Partial order

A partially ordered set (poset) is a set with a partial order ($\leq$) on it.

A partial order on some set S has 3 properties for any $a, b, c \in S$:

1. Reflexivity: $a \leq a$

Partial order

A partially ordered set (poset) is a set with a partial order ($\leq$) on it.

A partial order on some set S has 3 properties for any $a, b, c \in S$:

1. Reflexivity: $a \leq a$
2. Anti-symmetry: $a \leq b$ and $b \leq a \implies a = b$

Partial order

A partially ordered set (poset) is a set with a partial order ($\leq$) on it.

A partial order on some set S has 3 properties for any $a, b, c \in S$:

1. Reflexivity: $a \leq a$
2. Anti-symmetry: $a \leq b$ and $b \leq a \implies a = b$
3. Transitivity: $a \leq b$ and $b \leq c \implies a \leq c$

Partial order

A partially ordered set (poset) is a set with a partial order ($\leq$) on it.

A partial order on some set S has 3 properties for any $a, b, c \in S$:

1. Reflexivity: $a \leq a$
2. Anti-symmetry: $a \leq b$ and $b \leq a \implies a = b$
3. Transitivity: $a \leq b$ and $b \leq c \implies a \leq c$

For example, $\mathbb{Z}$ is partially ordered by $\leq$.

Partial order

A partially ordered set (poset) is a set with a partial order ($\leq$) on it.

A partial order on some set S has 3 properties for any $a, b, c \in S$:

1. Reflexivity: $a \leq a$
2. Anti-symmetry: $a \leq b$ and $b \leq a \implies a = b$
3. Transitivity: $a \leq b$ and $b \leq c \implies a \leq c$

For example, $\mathbb{Z}$ is partially ordered by $\leq$.
However, $\leq$ on $\mathbb{Z}$ is also a total order, as any two elements of $\mathbb{Z}$ can be compared. In contrast, any two elements in a set with a partial order may not necessarily be comparable.

Powerset example

The powerset of any set $\mathcal{P}(S)$ is partially ordered by inclusion $\subseteq$.

Powerset example

The powerset of any set $\mathcal{P}(S)$ is partially ordered by inclusion $\subseteq$.

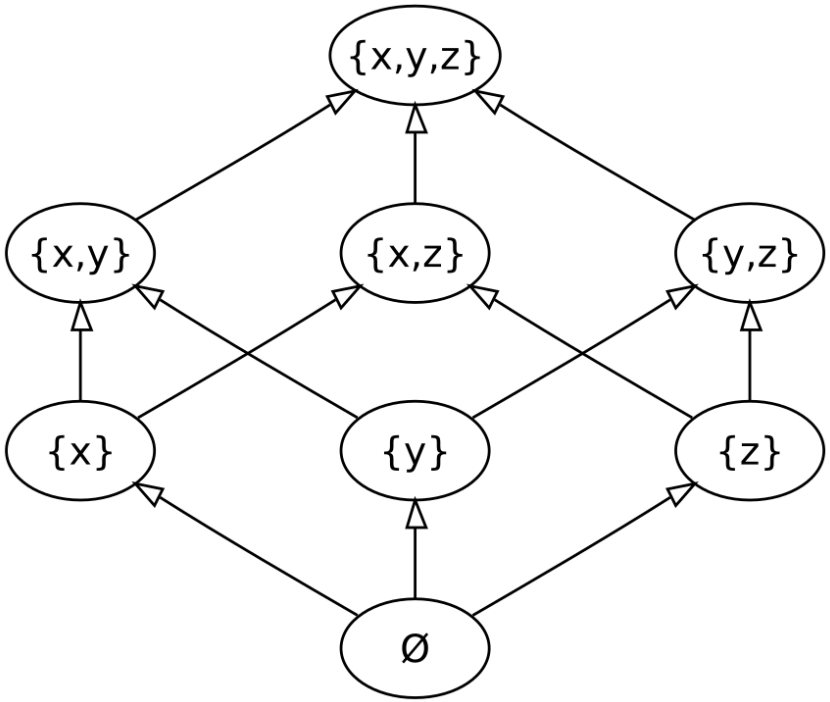


Table of Contents

Lattices

A lattice is a **mathematical structure** consisting of a **partially ordered set**, and a binary **join** and **meet** function for any pair of elements.

Lattices

A lattice is a **mathematical structure** consisting of a **partially ordered set**, and a binary **join** and **meet** function for any pair of elements.

Join

A join is a binary function that returns the **least upper bound** between the two arguments. Formally, j is a join of x and y if $x \leq j$ and $y \leq j$, and for any $w \in S$, $x \leq w$ and $y \leq w \implies j \leq w$

Lattices

A lattice is a **mathematical structure** consisting of a **partially ordered set**, and a binary **join** and **meet** function for any pair of elements.

Join

A join is a binary function that returns the **least upper bound** between the two arguments. Formally, j is a join of x and y if $x \leq j$ and $y \leq j$, and for any $w \in S$, $x \leq w$ and $y \leq w \implies j \leq w$

Meet

A meet is a binary function that returns the **greatest lower bound** between the two arguments. Formally, m is a meet of x and y if $m \leq x$ and $m \leq y$, and for any $w \in S$, $w \leq x$ and $w \leq y \implies w \leq m$

Top and bottom

Remark

Every finite lattice has a maximal top element ($\top$) and a minimal bottom element ($\bot$).

Top and bottom

Remark

Every finite lattice has a maximal top element ($\top$) and a minimal bottom element ($\bot$).

$\bot$ acts as the identity for join and $\top$ acts as the identity for meet.

Table of Contents

Abstract interpretation

Abstract interpretation is a mathematical theory that guarantees **sound and optimal approximation** of the semantics of a language.

Abstract interpretation

Abstract interpretation is a mathematical theory that guarantees **sound and optimal approximation** of the semantics of a language.
The general idea is:

Abstract interpretation

Abstract interpretation is a mathematical theory that guarantees **sound and optimal approximation** of the semantics of a language.
The general idea is:

- It is too expensive to keep track of every possible state an analysis is interested in

Abstract interpretation

Abstract interpretation is a mathematical theory that guarantees **sound and optimal approximation** of the semantics of a language.
The general idea is:

- It is too expensive to keep track of every possible state an analysis is interested in
- We define an abstraction that maps the concrete states to an **abstract domain**

Abstract interpretation

Abstract interpretation is a mathematical theory that guarantees **sound and optimal approximation** of the semantics of a language.
The general idea is:

- It is too expensive to keep track of every possible state an analysis is interested in
- We define an abstraction that maps the concrete states to an **abstract domain**
- We model the **concrete semantics** of the program in the abstract domain as **abstract semantics**

Abstract interpretation

Abstract interpretation is a mathematical theory that guarantees **sound and optimal approximation** of the semantics of a language.
The general idea is:

- It is too expensive to keep track of every possible state an analysis is interested in
- We define an abstraction that maps the concrete states to an **abstract domain**
- We model the **concrete semantics** of the program in the abstract domain as **abstract semantics**
- Our analysis implements the abstract semantics on the abstract domain, hence an "abstract interpreter"

Concrete and abstract domains

The concrete domain represents the actual possible states of a program.

Concrete and abstract domains

The concrete domain represents the actual possible states of a program.

```
for (int i = 1000000; i >= 0; i -= 10)
  print(i);
```

Concrete and abstract domains

The concrete domain represents the actual possible states of a program.

```
for (int i = 1000000; i >= 0; i -= 10)
  print(i);
```

Within the loop body, the concrete state of i can be represented by the set $\{i : 0 <= i <= 1000000, 10 | i\}$

Concrete and abstract domains

The concrete domain represents the actual possible states of a program.

```
for (int i = 1000000; i >= 0; i -= 10)
  print(i);
```

Within the loop body, the concrete state of i can be represented by the set $\{i : 0 <= i <= 1000000, 10 | i\}$

Far too expensive to keep track of all possible values of an integer! Sets could be as large as 2^{32} elements

Concrete and abstract domains

The abstract domain approximates the concrete domain by providing an **abstraction** on concrete states.

Concrete and abstract domains

The abstract domain approximates the concrete domain by providing an **abstraction** on concrete states.

```
for (int i = 1000000; i >= 0; i -= 10)
  print(i);
```

Concrete and abstract domains

The abstract domain approximates the concrete domain by providing an **abstraction** on concrete states.

```
for (int i = 1000000; i >= 0; i -= 10)
  print(i);
```

Instead of tracking every possible integer i could be, we could use an interval instead, representing i as $[0, 1000000]$

Concrete and abstract domains

The abstract domain approximates the concrete domain by providing an **abstraction** on concrete states.

```
for (int i = 1000000; i >= 0; i -= 10)
  print(i);
```

Instead of tracking every possible integer i could be, we could use an interval instead, representing i as $[0, 1000000]$

We lose precision, but now we only need to store 2 integers.

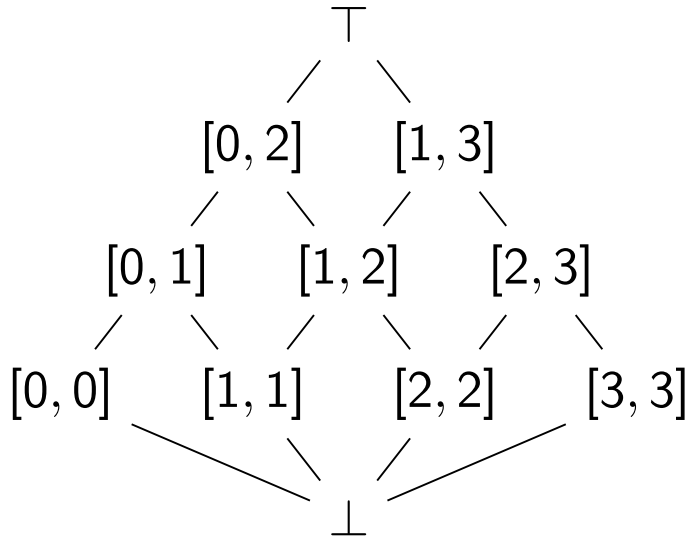
Lattices as domains

It turns out lattices are very useful tools to represent these domains.

Lattices as domains

It turns out lattices are very useful tools to represent these domains.

A point on the lattice can be thought of as a specific program state

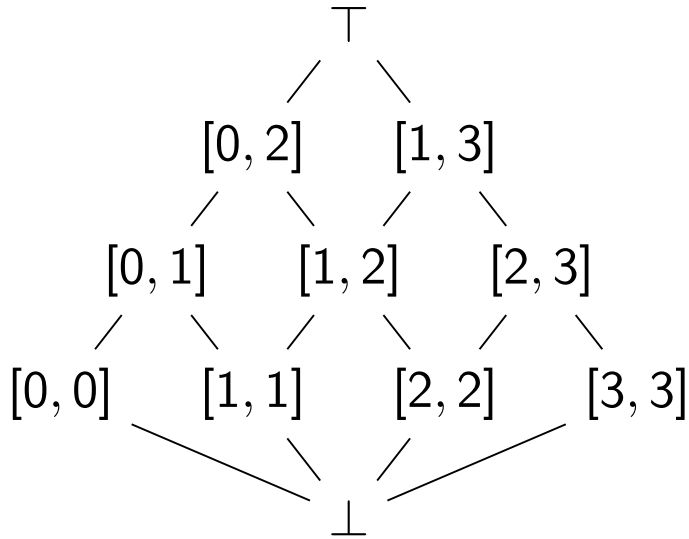


Lattices as domains

It turns out lattices are very useful tools to represent these domains.

A point on the lattice can be thought of as a specific program state

As we go up the lattice, we have less precise information about the program state



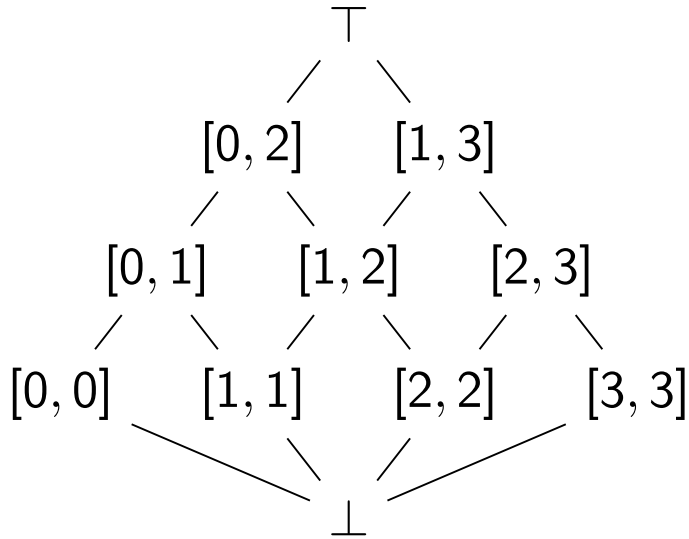
Lattices as domains

It turns out lattices are very useful tools to represent these domains.

A point on the lattice can be thought of as a specific program state

As we go up the lattice, we have less precise information about the program state

We can use the join to represent two program states merging.



Lattices as domains

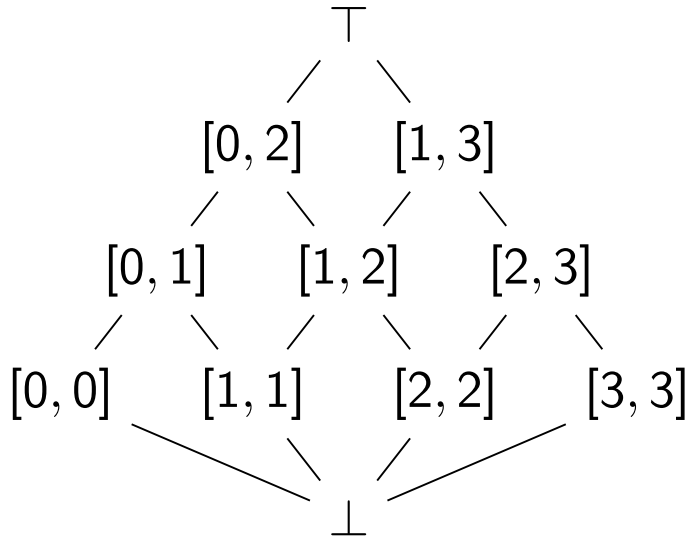
It turns out lattices are very useful tools to represent these domains.

A point on the lattice can be thought of as a specific program state

As we go up the lattice, we have less precise information about the program state

We can use the join to represent two program states merging.

We can also use the meet to "refine" program states when new constraints are introduced



Concrete vs abstract semantics

The concrete semantics of a language describes exactly how the program will execute on the actual values

Concrete vs abstract semantics

The concrete semantics of a language describes exactly how the program will execute on the actual values

Concrete semantics don't work on our abstract domains, we need to define a new **abstract semantics**

Transfer functions

Transfer functions are what implement the abstract semantics.

Transfer functions

Transfer functions are what implement the abstract semantics.

For example, if working with the interval domain, then
$[l_1, u_1] + [l_2, u_2] = [l_1 + l_2, u_1 + u_2]$

Transfer functions

Transfer functions are what implement the abstract semantics.

For example, if working with the interval domain, then
$[l_1, u_1] + [l_2, u_2] = [l_1 + l_2, u_1 + u_2]$

Another example: $[l_1, u_1] < [l_2, u_2]$

Transfer functions

Transfer functions are what implement the abstract semantics.

For example, if working with the interval domain, then $[l_1, u_1] + [l_2, u_2] = [l_1 + l_2, u_1 + u_2]$

Another example: $[l_1, u_1] < [l_2, u_2]$

1. If $u_1 < l_2$, then result is True

Transfer functions

Transfer functions are what implement the abstract semantics.

For example, if working with the interval domain, then $[l_1, u_1] + [l_2, u_2] = [l_1 + l_2, u_1 + u_2]$

Another example: $[l_1, u_1] < [l_2, u_2]$

1. If $u_1 < l_2$, then result is True
2. If $u_2 \leq l_1$, then result is False

Transfer functions

Transfer functions are what implement the abstract semantics.

For example, if working with the interval domain, then $[l_1, u_1] + [l_2, u_2] = [l_1 + l_2, u_1 + u_2]$

Another example: $[l_1, u_1] < [l_2, u_2]$

1. If $u_1 < l_2$, then result is True
2. If $u_2 \leq l_1$, then result is False
3. Otherwise, the result is $\top$

Transfer functions

Transfer functions are what implement the abstract semantics.

For example, if working with the interval domain, then
$[l_1, u_1] + [l_2, u_2] = [l_1 + l_2, u_1 + u_2]$

Another example: $[l_1, u_1] < [l_2, u_2]$

1. If $u_1 < l_2$, then result is True
2. If $u_2 \leq l_1$, then result is False
3. Otherwise, the result is $\top$

In the end, we just have to make sure the transfer functions are sound

Interval analysis example

```
x = 1
while x < 11:
  x += 2

if x == 11:
  x = 12

print(x)
```

Interval analysis example

Before analysis:

```
# x: ⊥
x = 1
while x < 11:
  x += 2

if x == 11:
  x = 12

print(x)
```

Interval analysis example

Before iteration:

```
x = 1
# x: [1, 1]
while x < 11:
  x += 2

if x == 11:
  x = 12

print(x)
```

Interval analysis example

At the end of iteration 1:

```
x = 1
# x: [1, 1]
while x < 11:
  x += 2
  # x: [3, 3]

if x == 11:
  x = 12

print(x)
```

Interval analysis example

Before iteration 2:

```
x = 1
# x: [1, 1] ∪ [3, 3]
while x < 11:
  x += 2

if x == 11:
  x = 12

print(x)
```

Interval analysis example

Before iteration 2:

```
x = 1
# x: [1, 3]
while x < 11:
  x += 2

if x == 11:
  x = 12

print(x)
```

Interval analysis example

At the end of iteration 2:

```
x = 1
# x: [1, 3]
while x < 11:
  x += 2
  # x: [3, 5]

if x == 11:
  x = 12

print(x)
```

Interval analysis example

Before iteration 3:

```
x = 1
# x: [1, 3] ∪ [3, 5]
while x < 11:
  x += 2

if x == 11:
  x = 12

print(x)
```

Interval analysis example

Before iteration 3:

```
x = 1
# x: [1, 5]
while x < 11:
  x += 2

if x == 11:
  x = 12

print(x)
```

Interval analysis example

Before iteration 4:

```
x = 1
# x: [1, 7]
while x < 11:
  x += 2

if x == 11:
  x = 12

print(x)
```

Interval analysis example

Before iteration 5:

```
x = 1
# x: [1, 9]
while x < 11:
  x += 2

if x == 11:
  x = 12

print(x)
```

Interval analysis example

Before iteration 6:

```
x = 1
# x: [1, 11]
while x < 11:
  x += 2

if x == 11:
  x = 12

print(x)
```

Interval analysis example

During interation 6:

```
x = 1
# x: [1, 11]
while x < 11:
  # x: [1, 11] ∩ [-inf, 10]
  x += 2

if x == 11:
  x = 12

print(x)
```

Interval analysis example

During interation 6:

```
x = 1
# x: [1, 11]
while x < 11:
  # x: [1, 10]
  x += 2

if x == 11:
  x = 12

print(x)
```

Interval analysis example

After iteration 6:

```
x = 1
# x: [1, 11]
while x < 11:
  x += 2
  # x: [3, 12]

if x == 11:
  x = 12

print(x)
```

Interval analysis example

After iteration 6:

```
x = 1
# x: [1, 12]
while x < 11:
  x += 2

if x == 11:
  x = 12

print(x)
```

Interval analysis example

After the loop:

```
x = 1
# x: [1, 12]
while x < 11:
  x += 2

# x: [1, 12] ∩ [11, inf]
if x == 11:
  x = 12

print(x)
```

Interval analysis example

After the loop:

```
x = 1
while x < 11:
  x += 2

# x: [11, 12]
if x == 11:
  x = 12

print(x)
```

Interval analysis example

After the loop:

```
x = 1
while x < 11:
  x += 2

# x: [11, 12]
if x == 11:
  x = 12
  # x: [12, 12]

print(x)
```

Interval analysis example

After the loop:

```
x = 1
while x < 11:
  x += 2

# x: [11, 12]
if x == 11:
  x = 12
  # x: [12, 12]
else:
  pass
  # x: [12, 12]

print(x)
```

Interval analysis example

After iteration 6:

```
x = 1
while x < 11:
  x += 2

if x == 11:
  x = 12

# x: [12, 12]
print(x)
```

Interval analysis example

After iteration 6:

```
x = 1
while x < 11:
  x += 2

if x == 11:
  x = 12

# x: [12, 12]
print(x)
```

Our analysis stops when there is nothing else we can update, i.e. we have reached a **fixed point**.

Interval analysis example

After iteration 6:

```
x = 1
while x < 11:
  x += 2

if x == 11:
  x = 12

# x: [12, 12]
print(x)
```

Our analysis stops when there is nothing else we can update, i.e. we have reached a **fixed point**.

Fixed points

Abstract interpretation aims to find fixed points in the presence of loops or recursion

Table of Contents

Abstraction and concretization functions

Let L_C be our concrete domain and L_A be the abstract domain.

Abstraction and concretization functions

Let L_C be our concrete domain and L_A be the abstract domain.

Let α be a function that maps elements from L_C to L_A, known as the **abstraction function**

Abstraction and concretization functions

Let L_C be our concrete domain and L_A be the abstract domain.

Let α be a function that maps elements from L_C to L_A, known as the **abstraction function**

Let γ be a function that maps elements from L_A to L_C, known as the **concretization function**

Abstraction and concretization functions

Let L_C be our concrete domain and L_A be the abstract domain.

Let α be a function that maps elements from L_C to L_A, known as the **abstraction function**

Let γ be a function that maps elements from L_A to L_C, known as the **concretization function**

α approximates our concrete values, γ converts the approximations back to real world values.

Soundness

Soundness

An analysis is sound if it never excludes an actual possible behaviour of the program

Soundness

Soundness

An analysis is sound if it never excludes an actual possible behaviour of the program

Our approximation is sound if for any concrete set S, $S \subseteq \gamma(\alpha(S))$.

Soundness

Soundness

An analysis is sound if it never excludes an actual possible behaviour of the program

Our approximation is sound if for any concrete set S, $S \subseteq \gamma(\alpha(S))$.

Abstraction never loses any information but it may include some extra "false" information.

Galois connections

Galois connection

For any concrete set S and any abstract value a, if our abstraction and concretization functions satisfy:

$$\alpha(S) \leq a \iff S \subseteq \gamma(a),$$

then we have a **galois connection**.

Galois connections

Galois connection

For any concrete set S and any abstract value a, if our abstraction and concretization functions satisfy:

$$\alpha(S) \leq a \iff S \subseteq \gamma(a),$$

then we have a **galois connection**.

Galois connections gives soundness and optimality.

Galois connections

Galois connection

For any concrete set S and any abstract value a, if our abstraction and concretization functions satisfy:

$$\alpha(S) \leq a \iff S \subseteq \gamma(a),$$

then we have a **galois connection**.

Galois connections gives soundness and optimality.

Let $a = \alpha(S)$, since $\alpha(S) \leq \alpha(S)$, by the galois connection we have $S \subseteq \gamma(\alpha(S))$

Galois connections

Galois connection

For any concrete set S and any abstract value a, if our abstraction and concretization functions satisfy:

$$\alpha(S) \leq a \iff S \subseteq \gamma(a),$$

then we have a **galois connection**.

Galois connections gives soundness and optimality.

Let $a = \alpha(S)$, since $\alpha(S) \leq \alpha(S)$, by the galois connection we have $S \subseteq \gamma(\alpha(S))$

Suppose $S \subseteq \gamma(a)$, then a is a valid approximation of S. By the galois connection, $\alpha(S)$ is always at least as precise as a.

Monotonicity

A monotonic function preserves the ordering on an ordered set

$$x \leq y \implies f(x) \leq f(y).$$

Monotonicity

A monotonic function preserves the ordering on an ordered set

$$x \leq y \implies f(x) \leq f(y).$$

Our transfer functions are monotonic if they preserve the ordering of the lattice.

Least fixed point

It turns out using monotonic transfer functions on our abstract domain, we are guaranteed a **least fixed point** exists (see Knaster–Tarski theorem).

Least fixed point

It turns out using monotonic transfer functions on our abstract domain, we are guaranteed a **least fixed point** exists (see Knaster–Tarski theorem).

If we iteratively apply our monotonic transfer functions on $\bot$, we are guaranteed to find the least fixed point (given some other nice conditions).

Least fixed point

It turns out using monotonic transfer functions on our abstract domain, we are guaranteed a **least fixed point** exists (see Knaster–Tarski theorem).

If we iteratively apply our monotonic transfer functions on $\bot$, we are guaranteed to find the least fixed point (given some other nice conditions).

The least fixed point represents the most precise possible answer in our abstract domain.

Least fixed point

It turns out using monotonic transfer functions on our abstract domain, we are guaranteed a **least fixed point** exists (see Knaster–Tarski theorem).

If we iteratively apply our monotonic transfer functions on $\bot$, we are guaranteed to find the least fixed point (given some other nice conditions).

The least fixed point represents the most precise possible answer in our abstract domain.

Since we are guaranteed to find this fixed point, we are guaranteed termination!

Wrapping things up

Abstract interpretation models execution of a program in an abstract domain

Wrapping things up

Abstract interpretation models execution of a program in an abstract domain

Galois connections gives us soundness and optimality

Wrapping things up

Abstract interpretation models execution of a program in an abstract domain

Galois connections gives us soundness and optimality

Monotonic transfer functions guarantees finding the most precise possible analysis in the abstract domain