Abstract Interpretation (1/2)

Martin Kellogg

Agenda: abstract interpretation

- Today: definitions, examples, soundness (?)
- Next class: more theory and examples

Agenda: abstract interpretation

- Today: **definitions**, examples, soundness
- Next class: more theory and examples

What is an abstract interpretation (formally)?

What is an abstract interpretation (formally)?

An abstract interpretation formally has **two components**:

What is an abstract interpretation (formally)?

An abstract interpretation formally has **two components**:

- an **abstract domain** over which to reason

What is an abstract interpretation (formally)?

An abstract interpretation formally has **two components**:

- an **abstract domain** over which to reason
- a set of **transfer functions** that tell the abstract interpreter how to reason over that abstract domain

What is an abstract interpretation (formally)?

An abstract interpretation formally has **two components**:

- an **abstract domain** over which to reason
- a set of **transfer functions** that tell the abstract interpreter how to reason over that abstract domain

A concrete interpreter for a real programming language (e.g., CPython, Node.js) also has these two components:

What is an abstract interpretation (formally)?

An abstract interpretation formally has **two components**:

- an **abstract domain** over which to reason
- a set of **transfer functions** that tell the abstract interpreter how to reason over that abstract domain

A concrete interpreter for a real programming language (e.g., CPython, Node.js) also has these two components:

- the "domain" is the concrete values that the machine can represent, like "64-bit integers"

What is an abstract interpretation (formally)?

An abstract interpretation formally has **two components**:

- an **abstract domain** over which to reason
- a set of **transfer functions** that tell the abstract interpreter how to reason over that abstract domain

A concrete interpreter for a real programming language (e.g., CPython, Node.js) also has these two components:

- the “domain” is the concrete values that the machine can represent, like “64-bit integers”
- the “transfer functions” are the concrete semantics of the programming language, such as what “+” actually means (“dispatch the operators to the ALU”)

What is an abstract interpretation (formally)?

An abstract interpretation formally has **two components**:

- an **abstract domain** over which to re
- a set of **transfer functions** that tell
  to reason over that abstract domai

A concrete interpreter for a real progra
CPython, Node.js) also has these two c

- the "domain" is the concrete values that the machine can represent, like "64-bit integers"
- the "transfer functions" are the concrete semantics of the programming language, such as what "+" actually means ("dispatch the operators to the ALU")

When dealing with a concrete language, we don't usually get to **choose** the domain or the semantics. But in abstract interpretation, we do!

Domains

Definition: a *domain* is a set of possible values

Domains

Definition: a ***domain*** is a set of possible values

- e.g., you might have heard the terms “domain” and “range” applied to functions in your previous math classes

Domains

Definition: a ***domain*** is a set of possible values

- e.g., you might have heard the terms “domain” and “range” applied to functions in your previous math classes
- we are interested in two kinds of domains:

Domains

Definition: a *domain* is a set of possible values

- e.g., you might have heard the terms "domain" and "range" applied to functions in your previous math classes
- we are interested in two kinds of domains:
  - the ***concrete domain*** of a variable is the set of values that the variable might actually take on during execution
    - probably familiar to you already
    - this is what the computer computes

Domains

Definition: a ***domain*** is a set of possible values

- e.g., you might have heard the terms “domain” and “range” applied to functions in your previous math classes
- we are interested in two kinds of domains:
    - the ***concrete domain*** of a variable is the set of values that the variable might actually take on during execution
        - probably familiar to you already
        - this is what the computer computes
    - an ***abstract domain*** is a layer of indirection on top of the concrete domain that splits the concrete domain into a smaller number of sets

Domains: concrete vs abstract example

Domains: concrete vs abstract example

- concrete domain = **natural numbers**:

Domains: concrete vs abstract example

- concrete domain = **natural numbers**:
    - { 0, 1, 2, 3, 4, … }

Domains: concrete vs abstract example

- concrete domain = **natural numbers**:
    - { 0, 1, 2, 3, 4, … }
- abstract domains:

Domains: concrete vs abstract example

- concrete domain = **natural numbers**:
  - { 0, 1, 2, 3, 4, … }
- abstract domains:
  - **even/odd**
  - **prime/composite**
  - **positive/nonnegative**
  - many more!

Domains: concrete vs abstract example

- concrete domain = **natural numbers**:
  - { 0, 1, 2, 3, 4, … }
- abstract domains:
  - **even/odd**
  - **prime/composite**
  - **positive/nonnegative**
  - many more!

Important property of an abstract domain: it must **completely cover** the concrete domain

Domains: concrete vs abstract

- More formally:

Domains: concrete vs abstract

- More formally:
  - let **C** be the **concrete domain** of interest (e.g., natural numbers)

Domains: concrete vs abstract

- More formally:
  - let **C** be the **concrete domain** of interest (e.g., natural numbers)
  - an ***abstract domain*** ***A*** **=** ***{A₁, A₂, ..., Aₙ}*** is a set of subsets of ***C*** that fulfills the following properties:

Domains: concrete vs abstract

- More formally:
  - let **C** be the **concrete domain** of interest (e.g., natural numbers)
  - an ***abstract domain*** $\boldsymbol{A = \{A_1, A_2, \ldots, A_n\}}$ is a set of subsets of **C** that fulfills the following properties:
    - $\forall\, \boldsymbol{A_i} \in \boldsymbol{A}, \boldsymbol{A_i} \subseteq \boldsymbol{C}$

Domains: concrete vs abstract

- More formally:
  - let **C** be the **concrete domain** of interest (e.g., natural numbers)
  - an ***abstract domain*** $\boldsymbol{A = \{A_1, A_2, \ldots, A_n\}}$ is a set of subsets of ***C*** that fulfills the following properties:
    - $\forall\, A_i \in A, A_i \subseteq C$
    - $A_1 \cup A_2 \cup \ldots \cup A_n = C$

Domains: concrete vs abstract

- More formally:
  - let C be the **concrete domain** of interest (e.g., natural numbers)
  - an ***abstract domain*** $A = \{A_1, A_2, \ldots, A_n\}$ is a set of subsets of C that fulfills the following properties:
    - $\forall\, A_i \in A, A_i \subseteq C$
    - $A_1 \cup A_2 \cup \ldots \cup A_n = C$
  - each A_i represents an ***abstract value***

Domains: concrete vs abstract

- More formally:
  - let ***C*** be the **concrete domain** of interest (e.g., natural numbers)
  - an ***abstract domain*** $\boldsymbol{A = \{A_1, A_2, \ldots, A_n\}}$ is a set of subsets of ***C*** that fulfills the following properties:
    - $\forall\, \boldsymbol{A_i} \in \boldsymbol{A}, \boldsymbol{A_i} \subseteq \boldsymbol{C}$
    - $\boldsymbol{A_1} \cup \boldsymbol{A_2} \cup \ldots \cup \boldsymbol{A_n} = \boldsymbol{C}$
  - each $\boldsymbol{A_i}$ represents an ***abstract value***
    - e.g., “odd integers”, “Strings that match my regular expression”, etc.

Domains: orderings and lattices

- An abstract domain is incomplete without an **ordering**: that is, a way to tell how the abstract values are related to each other
  - an abstract domain with an ordering is called a **lattice**

Domains: orderings and lattices

- An abstract domain is incomplete without an **ordering**: that is, a way to tell how the abstract values are related to each other
  - an abstract domain with an ordering is called a **lattice**
- There are two ways to express the ordering:

Domains: orderings and lattices

- An abstract domain is incomplete without an **ordering**: that is, a way to tell how the abstract values are related to each other
    - an abstract domain with an ordering is called a **lattice**
- There are two ways to express the ordering:
    - define a **less than relation** (usually denoted by $\sqsubset$), or

Domains: orderings and lattices

- An abstract domain is incomplete without an **ordering**: that is, a way to tell how the abstract values are related to each other
  - an abstract domain with an ordering is called a **lattice**
- There are two ways to express the ordering:
  - define a **less than relation** (usually denoted by $\sqsubset$), or
  - define a **least upper bound operator** (usually denoted by $\sqcup$)

Domains: orderings and lattices

- An abstract domain is incomplete without an **ordering**: that is, a way to tell how the abstract values are related to each other
  - an abstract domain with an ordering is called a **lattice**
- There are two ways to express the ordering:
  - define a **less than relation** (usually denoted by $\sqsubset$), or
  - define a **least upper bound operator** (usually denoted by $\sqcup$)
- These two approaches are **equivalent**: you can derive the LUB from the less than relation and vice-versa

Domains: ordering: less than relation

- Review: informally, a ***relation*** on a set may, or may not, hold between two given members of the set

Domains: ordering: less than relation

- Review: informally, a ***relation*** on a set may, or may not, hold between two given members of the set
  - formally, we define a relation as a set of ordered pairs

Domains: ordering: less than relation

- Review: informally, a ***relation*** on a set may, or may not, hold between two given members of the set
    - formally, we define a relation as a set of ordered pairs
- If $x \sqsubset y$, then we say that *x* is lower or less, and that *y* is higher or greater

Domains: ordering: less than relation

- Review: informally, a ***relation*** on a set may, or may not, hold between two given members of the set
  - formally, we define a relation as a set of ordered pairs
- If $x \sqsubset y$, then we say that *x* is lower or less, and that *y* is higher or greater
- The less-than relation **need not be total**
  - for two points *e1* and *e2*, it is possible that neither $e1 \sqsubset e2$ nor $e2 \sqsubset e1$ is true

Domains: ordering: least upper bound

- While the less than relation is in some ways better for doing a proof, it can be unwieldy when thinking about programs

Domains: ordering: least upper bound

- While the less than relation is in some ways better for doing a proof, it can be unwieldy when thinking about programs
- The least upper bound is often more useful, because it directly models the **join operator**

Domains: ordering: least upper bound

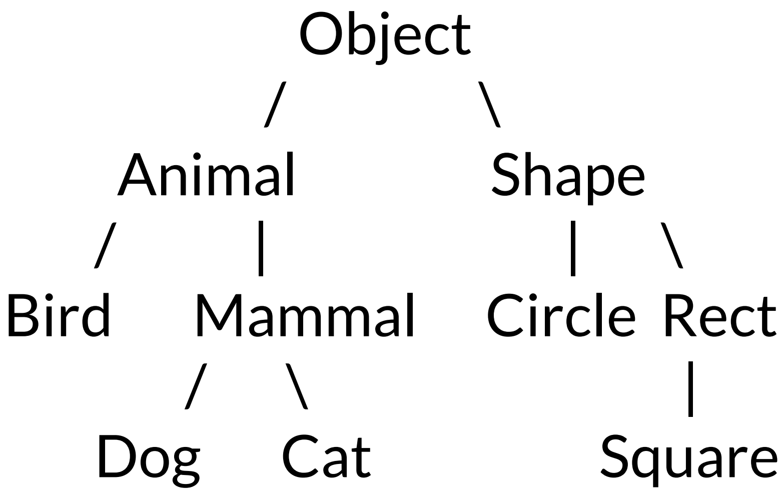
- While the less than relation is in some ways better for doing a proof, it can be unwieldy when thinking about programs
- The least upper bound is often more useful, because it directly models the **join operator**
  - that is, it models what happens when two possible abstract values flow to the same location (e.g., the then and else branches of an if)

Least upper bound: relationship to types

- You are already familiar with the LUB operator from our discussion of type systems and your experience with **object-oriented programming**

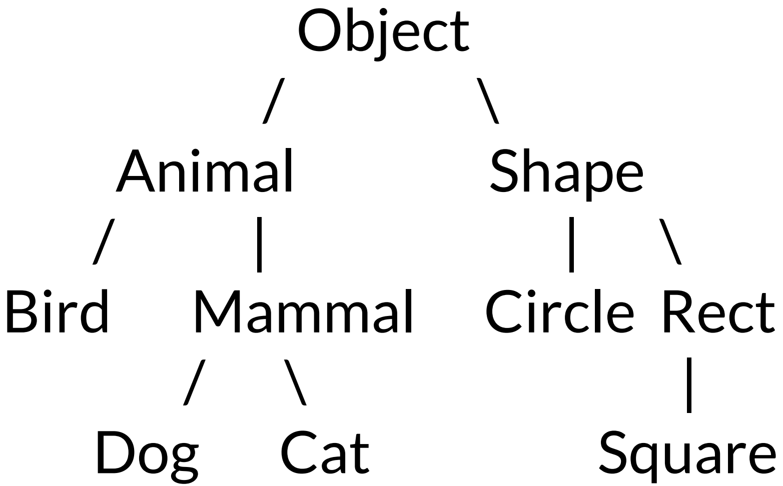
Least upper bound: relationship to types

- You are already familiar with the LUB operator from our discussion of type systems and your experience with **object-oriented programming**



Least upper bound: relationship to types

- You are already familiar with the LUB operator from our discussion of type systems and your experience with **object-oriented programming**
  - any time that you've answered the question "what is the closest supertype that these two types share", you're doing a LUB



Domains: ordering: least upper bound

- There are two important requirements on the LUB operator:

Domains: ordering: least upper bound

- There are two important requirements on the LUB operator:
    - it must be **complete**: that is, $\forall X, Y \in A \,.\, X \sqcup Y$ must be defined

Domains: ordering: least upper bound

- There are two important requirements on the LUB operator:
    - it must be **complete**: that is, $\forall X, Y \in A \,.\, X \sqcup Y$ must be defined
    - it must be **monotonic**: that is, it preserves the ordering relationship.

Domains: ordering: least upper bound

- There are two important requirements on the LUB operator:
    - it must be **complete**: that is, $\forall$ X, Y $\in$ A . X $\sqcup$ Y must be defined
    - it must be **monotonic**: that is, it preserves the ordering relationship.
        - LUB is a binary function; for a binary function f, monotonicity is defined as
            - $\forall$ a, b, c, d . a $\sqsubseteq$ b $\land$ c $\sqsubseteq$ d $\Rightarrow$ f(a, c) $\sqsubseteq$ f(b,d)

Domains: ordering: least upper bound

- There are two important requirements on the LUB operator:
  - it must be **complete**: that is, $\forall\ X, Y \in A\ .\ X \sqcup Y$ must be defined
  - it must be **monotonic**: that is, it preserves the ordering relationship.
    - LUB is a binary function; for a binary function f, monotonicity is defined as
      - $\forall\ a, b, c, d\ .\ a \sqsubseteq b \land c \sqsubseteq d \Rightarrow f(a, c) \sqsubseteq f(b,d)$
    - Note that this is not the same as:
      - $\forall\ x, y\ .\ f(x, y) \sqsupseteq x \land f(x, y) \sqsupseteq y$!
      - though this property is also true of the LUB operator

Domains: ordering: least upper bound

- There are two important requirements on the LUB operator:
    - it must be **complete**: that is, $\forall X, Y \in A . X \sqcup Y$ must be defined
    - it must be **monotonic**: that is, it preserves the ordering relationship.
        - LUB is a binary function; for a binary function f, monotonicity is defined as
            - $\forall a, b, c, d . a \sqsubseteq b \wedge c \sqsubseteq d =$
        - Note that this is not the same
            - $\forall x, y . f(x, y) \sqsupseteq x \wedge f(x, y) \sqsupseteq$
            - though this property is als

Hint: I like to ask exam questions like “why is this property required?” or “what would happen if it weren’t true?”

Domains: lattices = abstract domain + order

- A *lattice* formally has two components:
    - the abstract domain
    - the ordering relation

Domains: lattices = abstract domain + order

- A *lattice* formally has two components:
  - the abstract domain
  - the ordering relation
- That is, a lattice is a *partially-ordered* set

Domains: lattices = abstract domain + order

- A *lattice* formally has two components:
  - the abstract domain
  - the ordering relation
- That is, a lattice is a *partially-ordered* set

A set is ***partially ordered*** iff ∃ a binary relationship ≤ that is:

- ***reflexive***: $x \leq x$
- ***anti-symmetric***: $x \leq y \land y \leq x \Rightarrow x = y$
- ***transitive***: $x \leq y \land y \leq z \Rightarrow x \leq z$

Domains: lattices = abstract domain + order

- A *lattice* formally has two components:
  - the abstract domain
  - the ordering relation
- That is, a lattice is a *partially-ordered* set
  - *join semilattices* and *meet semilattices* are special kinds of partially-ordered sets

Domains: lattices = abstract domain + order

- A ***lattice*** formally has two components:
  - the abstract domain
  - the ordering relation
- That is, a lattice is a ***partially-ordered*** set
  - ***join semilattices*** and ***meet semilattices*** are special kinds of partially-ordered sets
    - join semilattices have a unique top element

Domains: lattices nain + order

- A ***lattice*** formally has
  - the abstract doma
  - the ordering relat
- That is, a lattice is a ***po***
  - ***join semilattices*** ar are special kinds of partially-ordered s
    - join semilattices have a unique top element

Join semilattice example:

```
  T
 / \
A   B
|   |
C   D
```

Domains: lattices = abstract domain + order

- A ***lattice*** formally has two components:
  - the abstract domain
  - the ordering relation
- That is, a lattice is a ***partially-ordered*** set
  - ***join semilattices*** and ***meet semilattices*** are special kinds of partially-ordered sets
    - join semilattices have a unique top element
    - meet semilattices have a unique bottom element

Domains: lattices ain + order

Meet semilattice example:

```
A   B
|   |
C   D
 \ /
  ⊥
```

- A ***lattice*** formally has
  - the abstract doma
  - the ordering relat
- That is, a lattice is a ***p***
  - ***join semilattices*** ar e special kinds of partially-ordered s
    - join semilattices have a unique top element
    - meet semilattices have a unique bottom element

Domains: lattices = abstract domain + order

- A *lattice* formally has two components:
  - the abstract domain
  - the ordering relation
- That is, a lattice is a *partially-ordered* set
  - ***join semilattices*** and ***meet semilattices*** are special kinds of partially-ordered sets
    - join semilattices have a unique top element
    - meet semilattices have a unique bottom element
  - a lattice formally is both a join and a meet semilattice

AI = Lattice + Transfer functions

AI = Lattice + Transfer functions

- the goal of the transfer functions are to encode the ***abstract semantics*** of the operations in the programming language

AI = Lattice + Transfer functions

- the goal of the transfer functions are to encode the ***abstract semantics*** of the operations in the programming language
  - that is, the transfer function for an operation answers the question “what does this operation ***mean*** in the context of the abstract domain”?

AI = Lattice + Transfer functions

- the goal of the transfer functions are to encode the ***abstract semantics*** of the operations in the programming language
  - that is, the transfer function for an operation answers the question “what does this operation ***mean*** in the context of the abstract domain”?
- formally, an abstract interpretation requires a transfer function for **each language construct**

AI = Lattice + Transfer functions

- the goal of the transfer functions are to encode the ***abstract semantics*** of the operations in the programming language
    - that is, the transfer function for an operation answers the question “what does this operation ***mean*** in the context of the abstract domain”?
- formally, an abstract interpretation requires a transfer function for **each language construct**
    - in practice, though, we usually assume that most are obvious and focus on the ones that might be interesting, which is what I’ll do in the examples on the next few slides

Example AI: even/odd integers

Example AI: even/odd integers

Example lattice:

Example AI: even/odd integers

Example lattice:

```
{ even, odd } = top
     /        \
 {even}     {odd}
     \         /
     {} = bottom
```

Example AI: even/odd integers

Example lattice:

```
{ even, odd } = top
     /           \
 {even}       {odd}
     \           /
      {} = bottom
```

A note about top:

- top represents ***no constraints*** on the possible values
- equivalently, ***every value*** is a member of top

Example AI: even/odd integers

Example lattice:

```
{ even, odd } = top
      /         \
  {even}      {odd}
      \         /
     {} = bottom
```

Similarly for bottom:

- bottom represents ***all possible constraints at once*** on values
- equivalently, ***no values*** are members of bottom

Example AI: even/odd integers

Example lattice:

```
{ even, odd } = top
     /            \
{even}       {odd}
     \            /
      {} = bottom
```

Example transfer function:

+	T	even	odd	⊥
T				
even				
odd				
⊥				

Example AI: even/odd integers

Example lattice:

{ even, odd } = top
/ \
{even} {odd}
\ /
{} = bottom

Example transfer function:

+	T	even	odd	⊥
T	T	T	T	⊥
even	T	even	odd	⊥
odd	T	odd	even	⊥
⊥	⊥	⊥	⊥	⊥

Example AI: even/odd integers

Let’s apply this AI to an example:

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Example AI: even/odd integers

Let’s apply this AI to an example:

```
x = 0;
y = read_even()
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Concrete execution

```
{x=0;  y=undef}
{x=0;  y=8}
{x=9;  y=8}
{x=9;  y=18}
{x=16; y=18}
{x=16; y=8}
```

Example AI: even/odd integers

Let's apply this AI to an example:

```
x = 0;
y = read_even()
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Concrete execution

```
{x=0;  y=undef}
{x=0;  y=8}
{x=9;  y=8}
{x=9;  y=18}
{x=16; y=18}
{x=16; y=8}
```

Abstract interpr.

```
{x=?;  y=?}
{x=?;  y=?}
{x=?;  y=?}
{x=?;  y=?}
{x=?;  y=?}
{x=?;  y=?}
```

Example AI: even/odd integers

Let’s apply this AI to an example:

```
x = 0;
y = read_even()
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Concrete execution

```
{x=0;  y=undef}
{x=0;  y=8}
{x=9;  y=8}
{x=9;  y=18}
{x=16; y=18}
{x=16; y=8}
```

Abstract interpr.

```
{x=e;  y=?}
{x=?;  y=?}
{x=?;  y=?}
{x=?;  y=?}
{x=?;  y=?}
{x=?;  y=?}
```

Abstraction function

- How did we know that 0 was even?

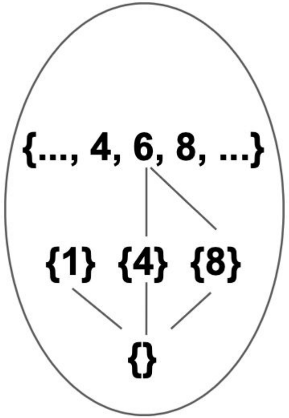
Abstraction function

- How did we know that 0 was even?
    - an ***abstraction function*** (typically denoted by α) tells us which abstract domain a particular concrete element belongs to

Abstraction function

- How did we know that 0 was even?
  - an ***abstraction function*** (typically denoted by α) tells us which abstract domain a particular concrete element belongs to

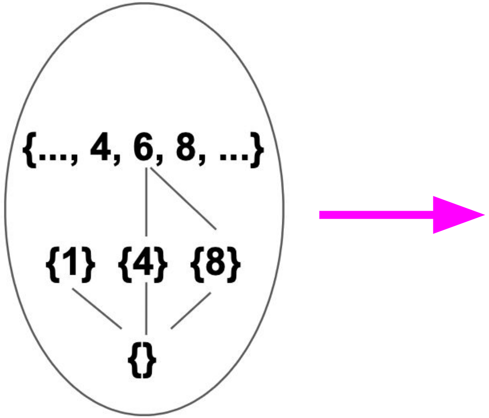
concrete domain



Abstraction function

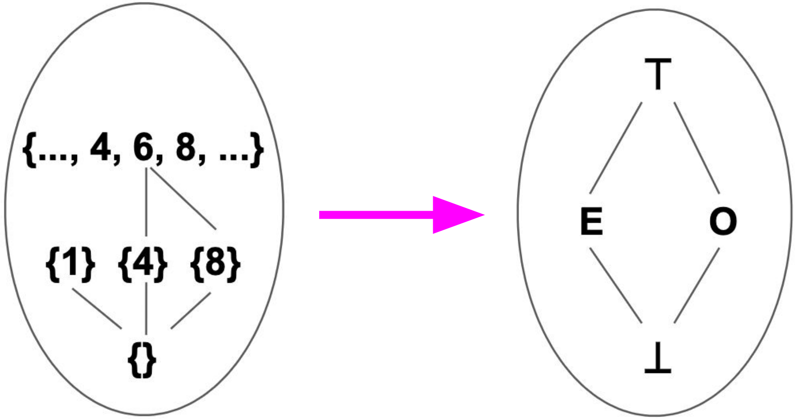
- How did we know that 0 was even?
  - an ***abstraction function*** (typically denoted by α) tells us which abstract domain a particular concrete element belongs to

concrete domain



Abstraction function

- How did we know that 0 was even?
  - an ***abstraction function*** (typically denoted by α) tells us which abstract domain a particular concrete element belongs to



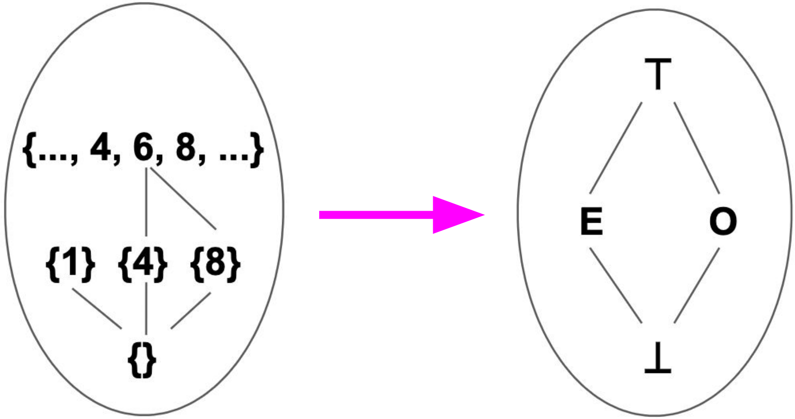
concrete domain

abstract domain

Abstraction function

- How did we know that 0 was even?
  - an ***abstraction function*** (typically denoted by α) tells us which abstract domain a particular concrete element belongs to

concrete domain



abstract domain

e.g.:
α(4) = even
α({}) = bottom

Concretization function

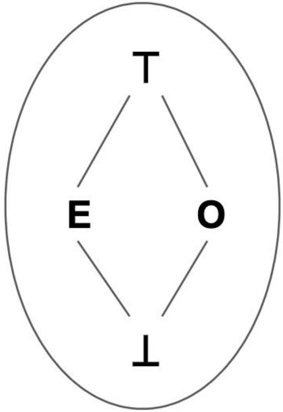
- What about going the other way?

Concretization function

- What about going the other way?
    - an ***concretization function*** (typically denoted by γ) tells us which concrete elements are associated with an abstract value

Concretization function

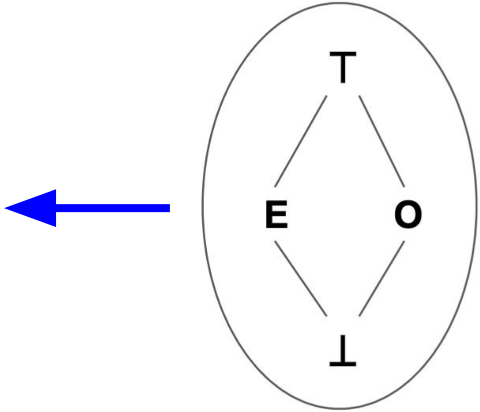
- What about going the other way?
  - an ***concretization function*** (typically denoted by γ) tells us which concrete elements are associated with an abstract value



abstract domain

Concretization function

- What about going the other way?
  - an ***concretization function*** (typically denoted by γ) tells us which concrete elements are associated with an abstract value

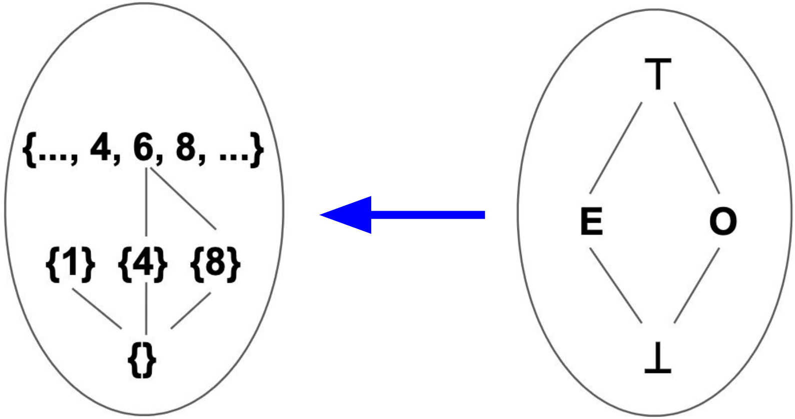


abstract domain

Concretization function

- What about going the other way?
  - an ***concretization function*** (typically denoted by γ) tells us which concrete elements are associated with an abstract value

concrete domain

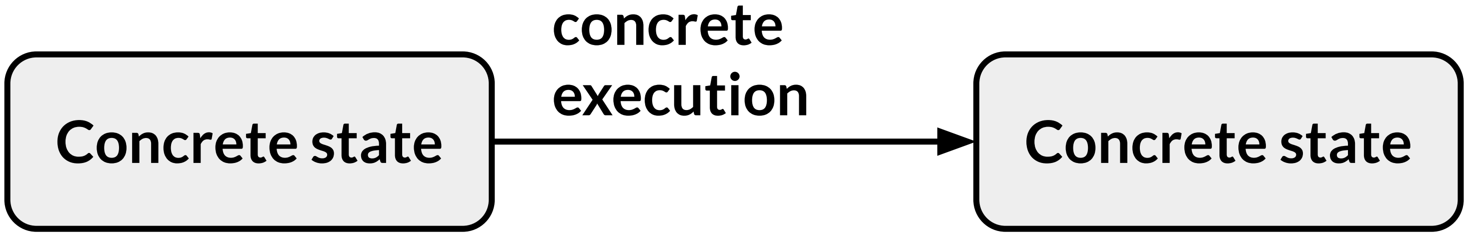


abstract domain

Role of abstr., concr., and transfer fcns.

Concrete state

Role of abstr., concr., and transfer fcns.

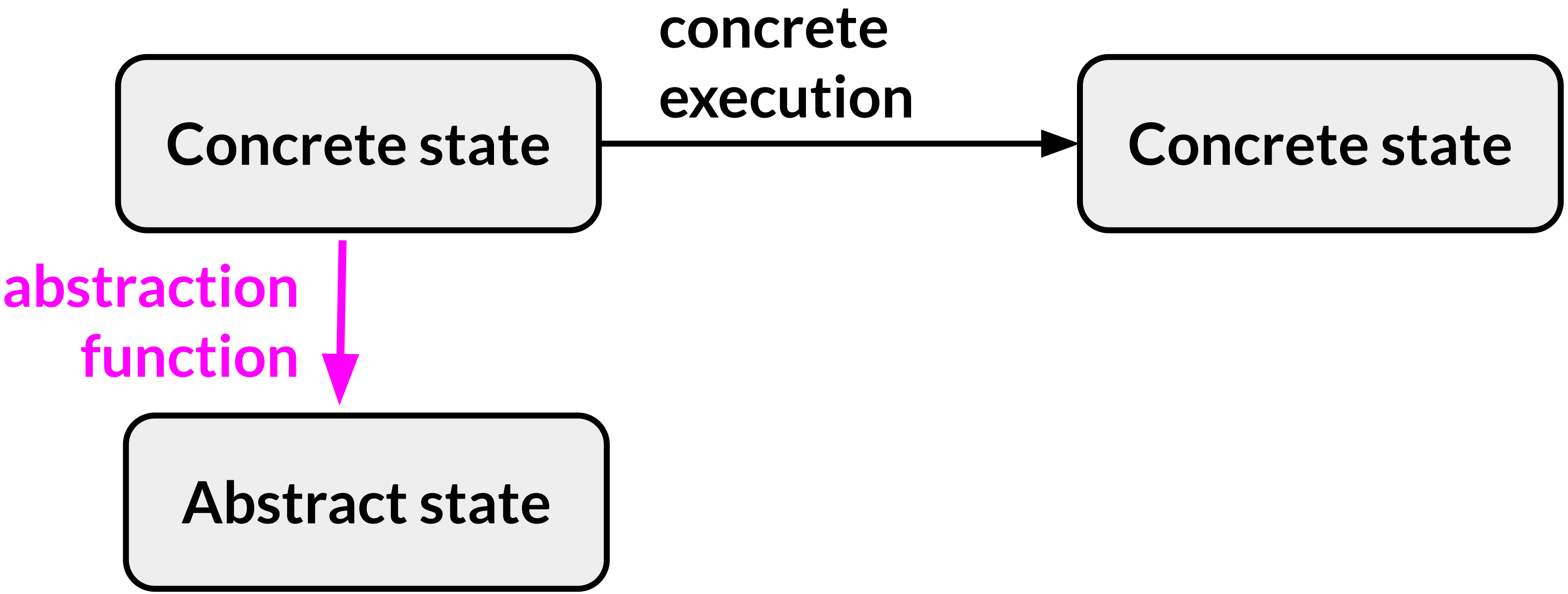


Concrete state

concrete execution

Concrete state

Role of abstr., concr., and transfer fcns.



Concrete state

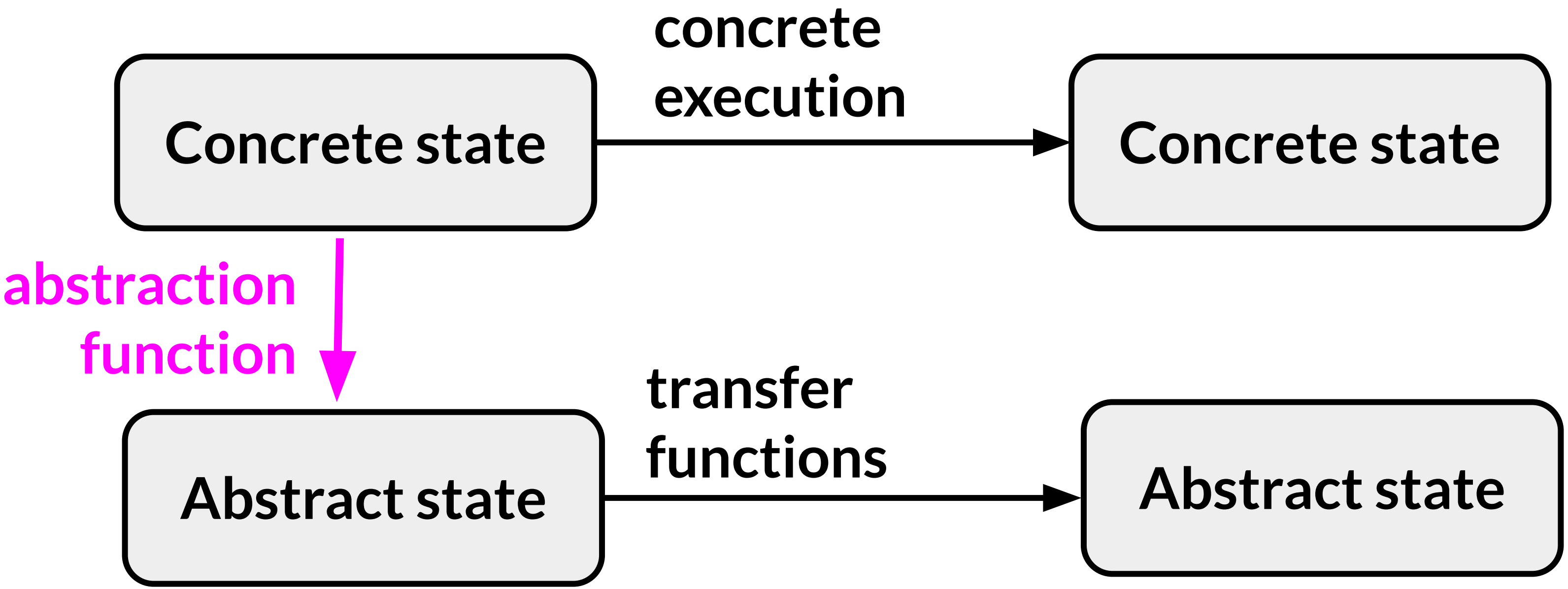
concrete execution

Concrete state

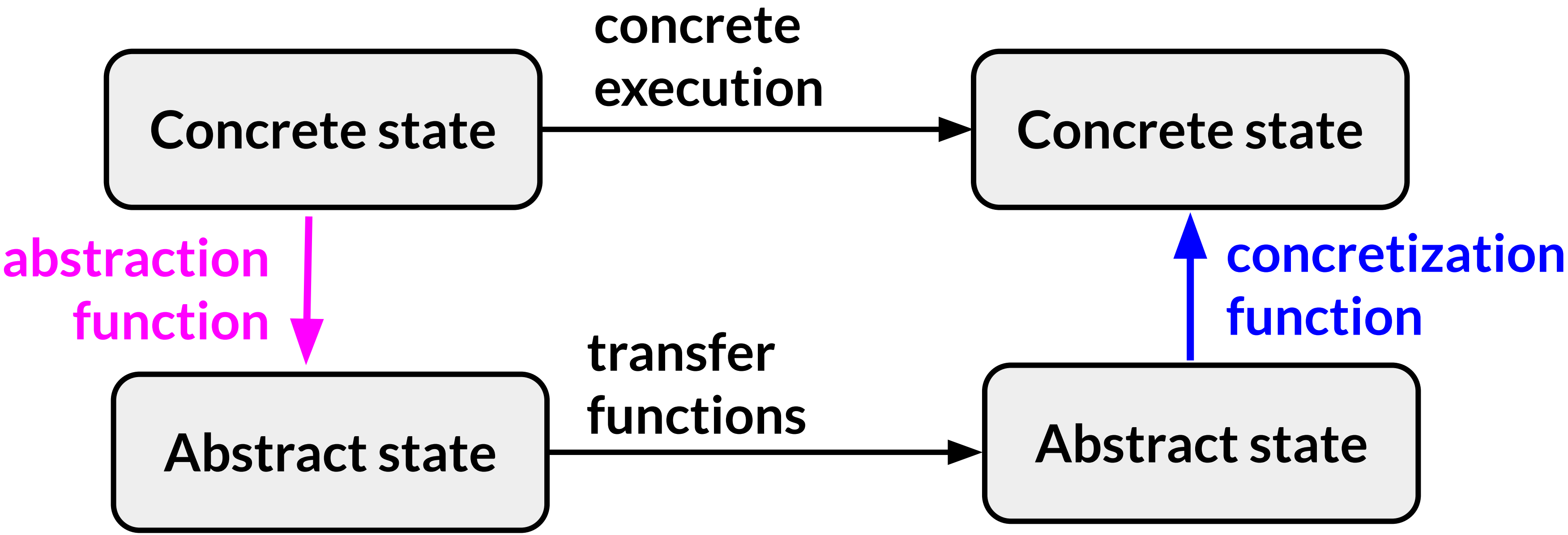
abstraction function

Abstract state

Role of abstr., concr., and transfer fcns.



Role of abstr., concr., and transfer fcns.



Example AI: even/odd integers

Let’s apply this AI to an example:

```
x = 0;
y = read_even()
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Concrete execution

```
{x=0;  y=undef}
{x=0;  y=8}
{x=9;  y=8}
{x=9;  y=18}
{x=16; y=18}
{x=16; y=8}
```

Abstract interpr.

```
{x=e;  y=?}
{x=?;  y=?}
{x=?;  y=?}
{x=?;  y=?}
{x=?;  y=?}
{x=?;  y=?}
```

Example AI: even/odd integers

Let’s apply this AI to an example:

```
x = 0;
y = read_even()
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Concrete execution

```
{x=0;  y=undef}
{x=0;  y=8}
{x=9;  y=8}
{x=9;  y=18}
{x=16; y=18}
{x=16; y=8}
```

Abstract interpr.

```
{x=e; y=⊥}
{x=?; y=?}
{x=?; y=?}
{x=?; y=?}
{x=?; y=?}
{x=?; y=?}
```

Example AI: even/odd integers

Let’s apply this AI to an example:

```
x = 0;
y = read_even()
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

<u>**Concrete execution**</u>	<u>**Abstract interpr.**</u>
`{x=0;   y=undef}`	{x=**e**; y=⊥}
`{x=0;   y=8}`	{x=**e**; y=**e**}
`{x=9;   y=8}`	`{x=?;  y=?}`
`{x=9;   y=18}`	`{x=?;  y=?}`
`{x=16;  y=18}`	`{x=?;  y=?}`
`{x=16;  y=8}`	`{x=?;  y=?}`

Example AI: even/odd integers

Let’s apply this AI to an example:

transfer function for +!

```
x = 0;
y = read_even()
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Concrete execution	Abstract interpr.
`{x=0;  y=undef}`	{x=**e**; y=⊥}
`{x=0;  y=8}`	{x=**e**; y=**e**}
`{x=9;  y=8}`	{x=**o**; y=**e**}
`{x=9;  y=18}`	`{x=?;  y=?}`
`{x=16; y=18}`	`{x=?;  y=?}`
`{x=16; y=8}`	`{x=?;  y=?}`

Example AI: even/odd integers

Let’s apply this AI to an example:

```
x = 0;
y = read_even()
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Concrete execution	**Abstract interpr.**
{x=0; y=undef}	{x=**e**; y=⊥}
{x=0; y=8}	{x=**e**; y=**e**}
{x=9; y=8}	{x=**o**; y=**e**}
{x=9; y=18}	{x=**o**; y=**e**}
{x=16; y=18}	{x=?; y=?}
{x=16; y=8}	{x=?; y=?}

Example AI: even/odd integers

Let’s apply this AI to an example:

```
x = 0;
y = read_even()
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Concrete execution

```
{x=0;  y=undef}
{x=0;  y=8}
{x=9;  y=8}
{x=9;  y=18}
{x=16; y=18}
{x=16; y=8}
```

Abstract interpr.

```
{x=e;  y=⊥}
{x=e;  y=e}
{x=o;  y=e}
{x=o;  y=e}
{x=e;  y=e}
{x=?;  y=?}
```

Example AI: even/odd integers

Let’s apply this AI to an example:

```
x = 0;
y = read_even()
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Concrete execution	Abstract interpr.
{x=0; y=undef}	{x=e; y=⊥}
{x=0; y=8}	{x=e; y=e}
{x=9; y=8}	{x=o; y=e}
{x=9; y=18}	{x=o; y=e}
{x=16; y=18}	{x=e; y=e}
{x=16; y=8}	{x=e; y=e?}

Example AI: even/odd integers

Let’s apply this AI to an example:

```
x = 0;
y = read_even()
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Concrete execution	**Abstract interpr.**
`{x=0;   y=undef}`	`{x=`**e**`;   y=⊥}`
`{x=0;   y=8}`	`{x=`**e**`;   y=`**e**`}`
`{x=9;   y=8}`	`{x=`**o**`;   y=`**e**`}`
`{x=9;   y=18}`	`{x=`**o**`;   y=`**e**`}`
`{x=16; y=18}`	`{x=`**e**`;   y=`**e**`}`
`{x=16; y=8}`	`{x=`**e**`;   y=`**e**`?}`

Example AI: even/odd integers

What’s the transfer function for division?

↓/→	⊤	even	odd	⊥
⊤				
even				
odd				
⊥				

Example AI: even/odd integers

What’s the transfer function for division?

↓/→	T	even	odd	⊥
T	T	T	T	⊥
even	T	T	T	⊥
odd	T	T	T	⊥
⊥	⊥	⊥	⊥	⊥

Notes for online readers:

- even/even is top:
  - 6/2 = 3
  - 8/2 = 4
- odd/odd is top:
  - 5/5 = 1
  - 11/5 = 2
    - integer division!

Example AI: even/odd integers

Let’s apply this AI to an example:

```
x = 0;
y = read_even()
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Concrete execution

```
{x=0;  y=undef}
{x=0;  y=8}
{x=9;  y=8}
{x=9;  y=18}
{x=16; y=18}
{x=16; y=8}
```

Abstract interpr.

```
{x=e;  y=⊥}
{x=e;  y=e}
{x=o;  y=e}
{x=o;  y=e}
{x=e;  y=e}
{x=e;  y=T}
```

Example AI: even/odd integers

Let's apply this AI to an example:

```
x = 0;
y = read_even()
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

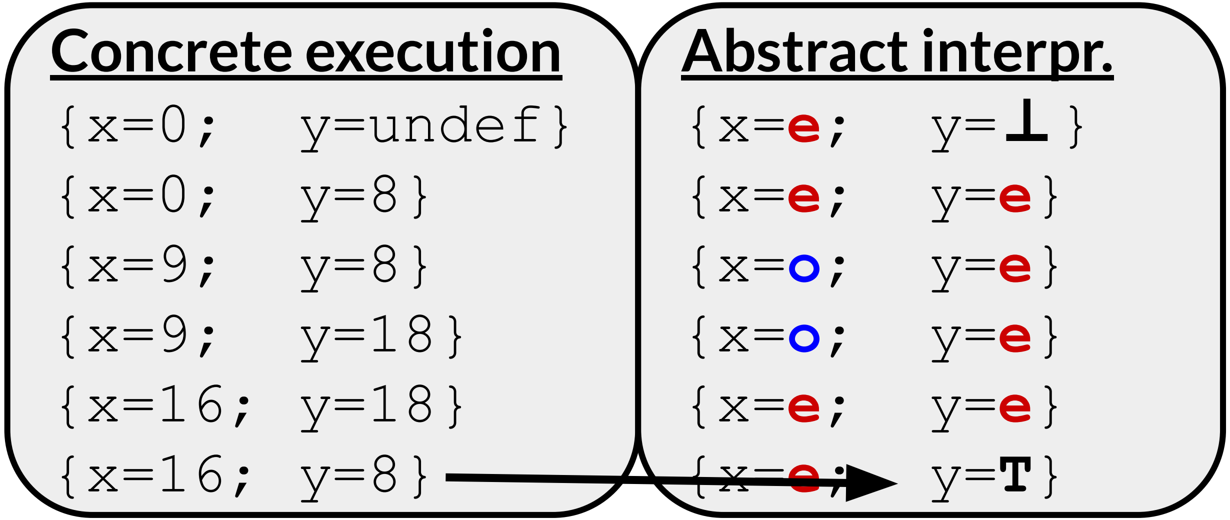
Concrete execution	**Abstract interpr.**
{x=0; y=undef}	{x=e; y=⊥}
{x=0; y=8}	{x=e; y=e}
{x=9; y=8}	{x=o; y=e}
{x=9; y=18}	{x=o; y=e}
{x=16; y=18}	{x=e; y=e}
{x=16; y=8}	{x=e; y=⊤}

for x, our abstraction was precise

Example AI: even/odd integers

Let’s apply this AI to an example:

```
x = 0;
y = read_even()
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

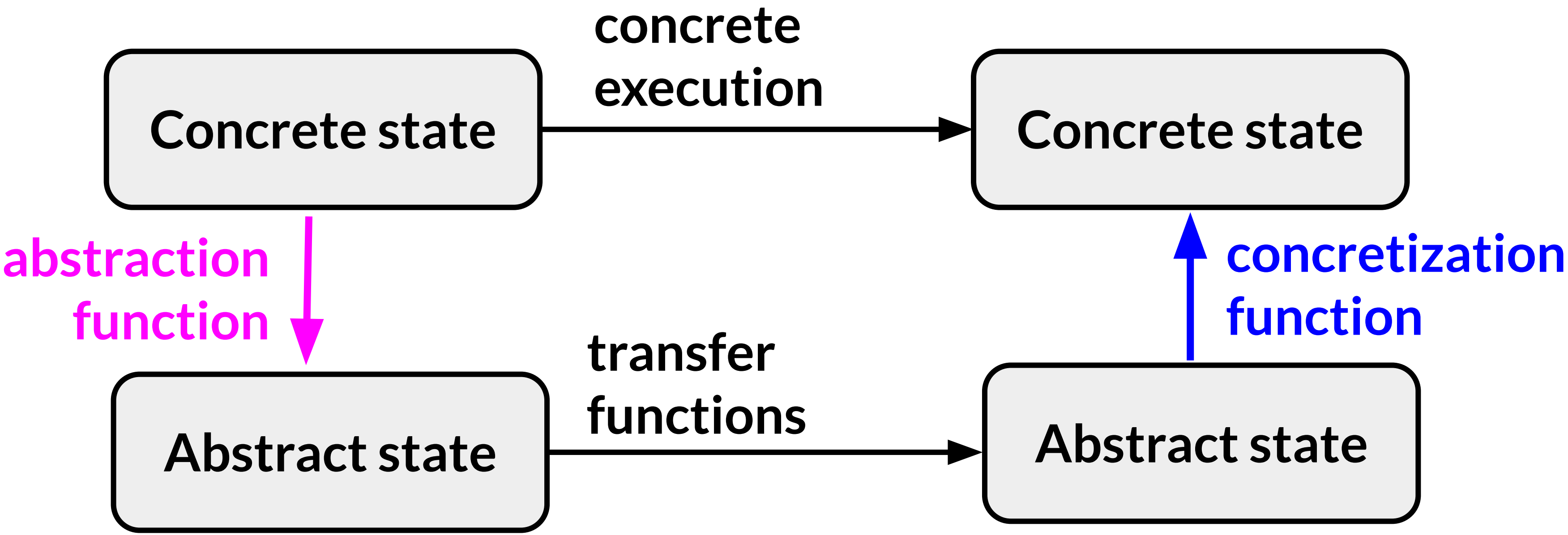


Concrete execution	Abstract interpr.
{x=0; y=undef}	{x=e; y=⊥}
{x=0; y=8}	{x=e; y=e}
{x=9; y=8}	{x=o; y=e}
{x=9; y=18}	{x=o; y=e}
{x=16; y=18}	{x=e; y=e}
{x=16; y=8}	{x=e; y=T}

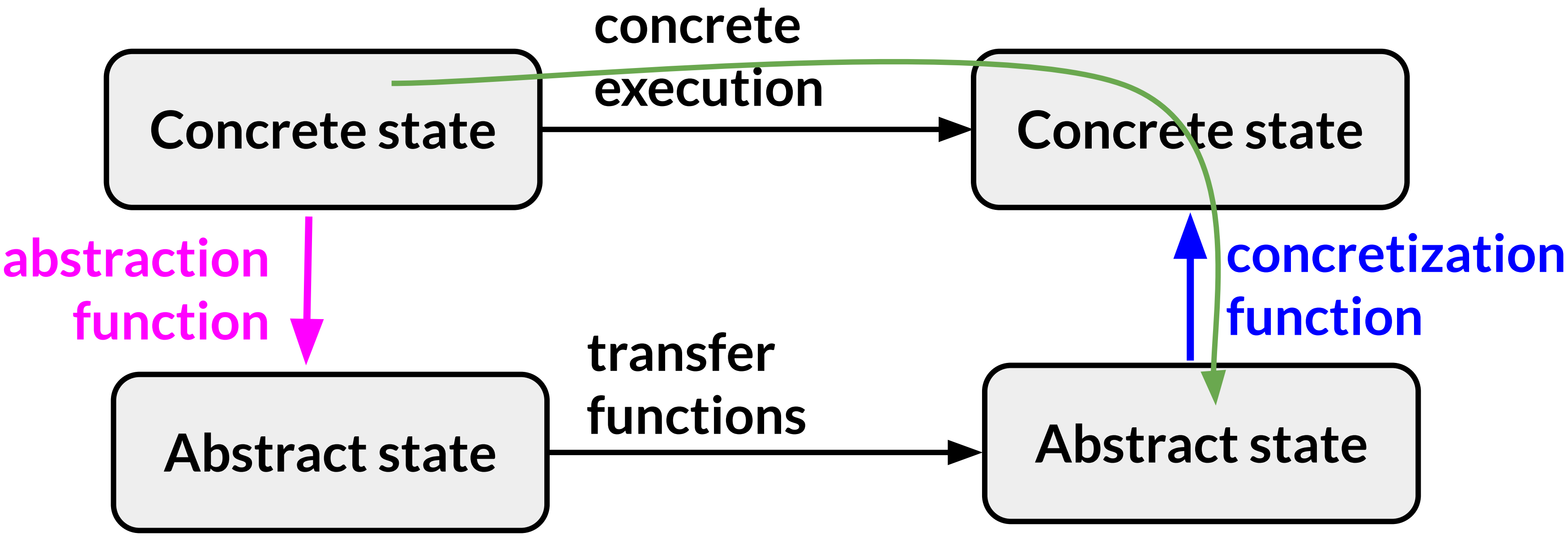
for x, our abstraction was precise

but for y, it was not

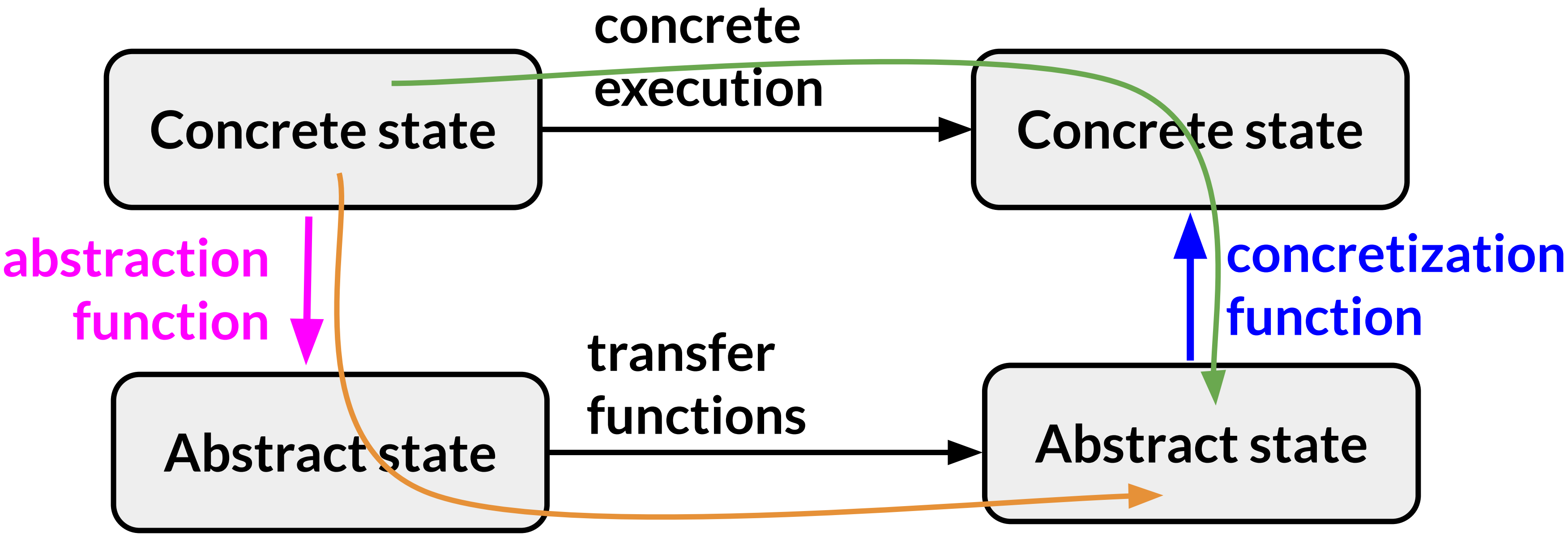
Approximation!



Approximation!



Approximation!



Concrete state

concrete execution

Concrete state

abstraction function

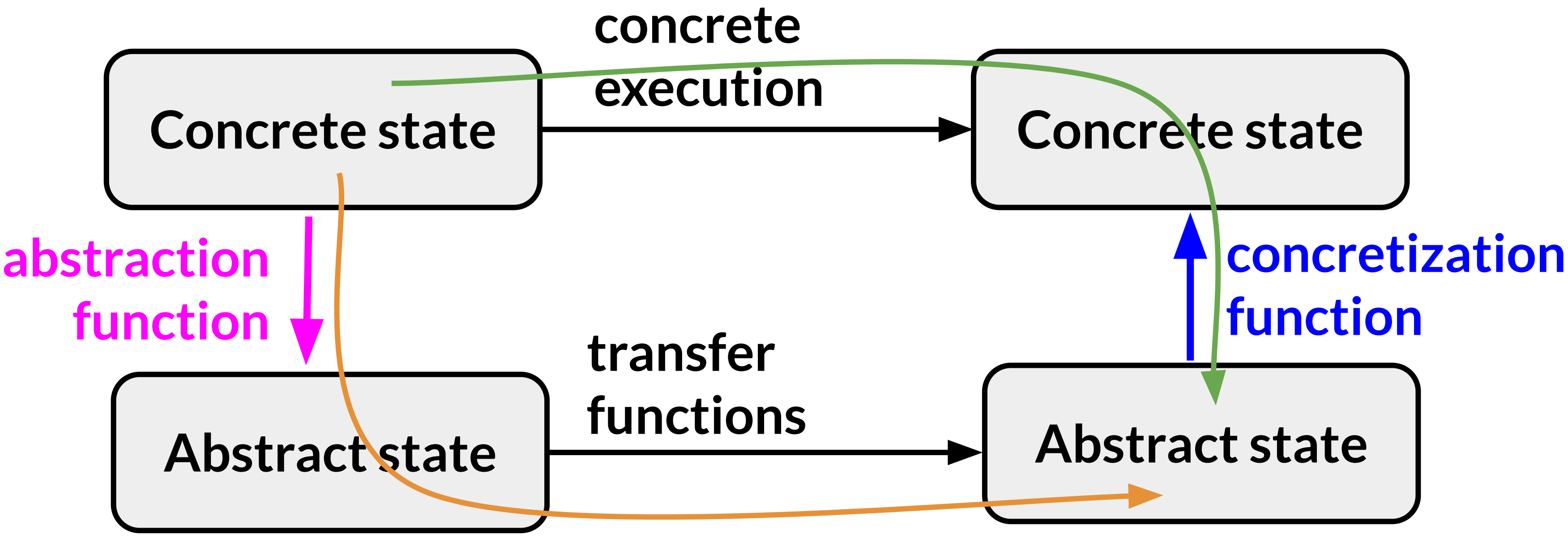
concretization function

Abstract state

transfer functions

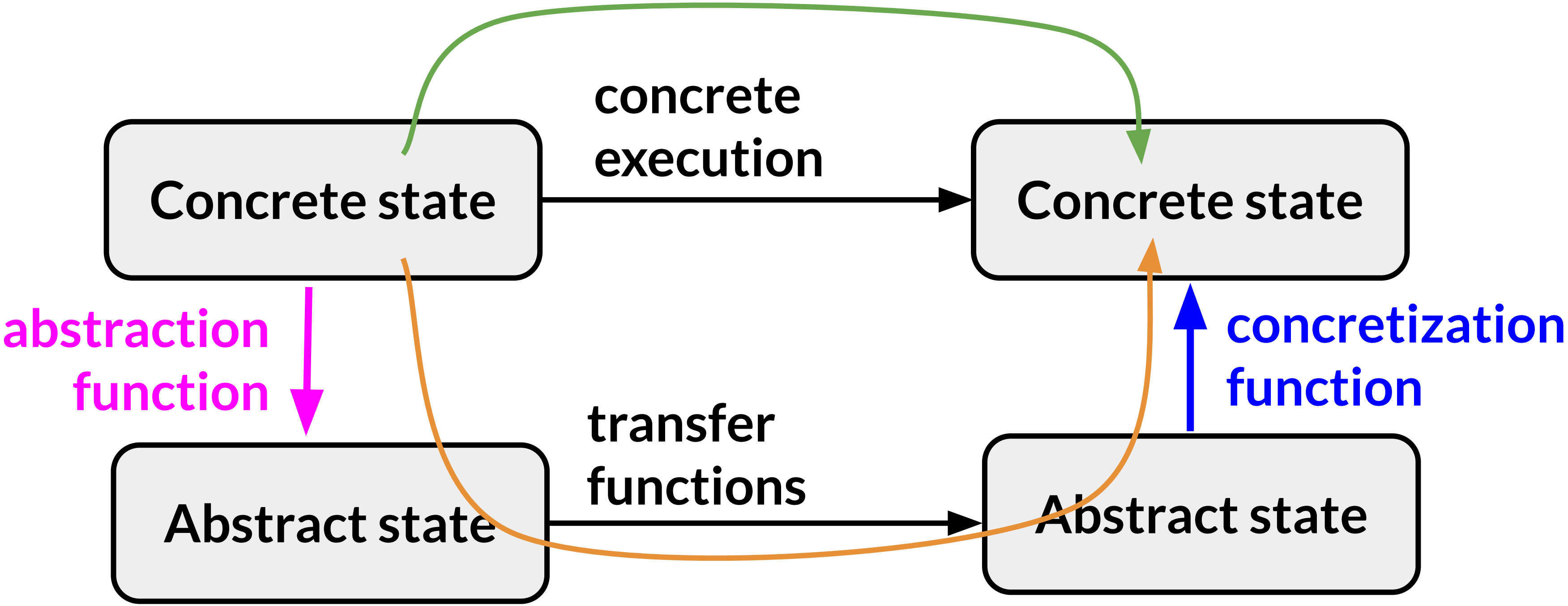
Abstract state

Approximation!



Do the green and orange paths always lead to the same abstract state?

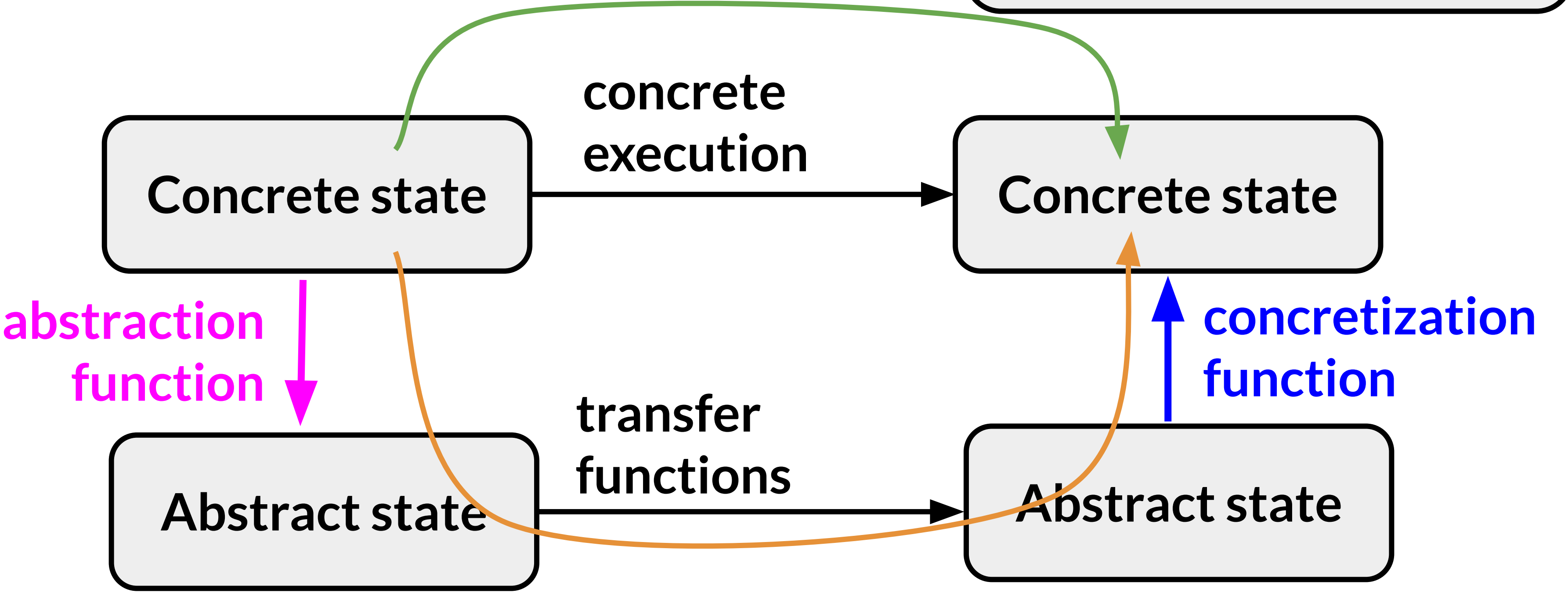
Approximation!



Do the green and orange paths always lead to the same concrete state?

Approximation!

We'll come back to this question when we discuss **soundness**



Do the green and orange paths always lead to the same concrete state?

Alternative example AI: even/odd integers

Is there an **alternative** AI that we can use to conclude that y is even after we analyze the example?

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Alternative example AI: even/odd integers

Is there an alternative AI that we can use to conclude that y is even after we analyze the example?

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

In-class exercise: with a partner, ***design an alternative*** abstract interpretation that can conclude that y is even.

Alternative example AI: even/odd integers

Key property that we need to conclude is that `x / 2` is even.

Alternative example AI: even/odd integers

Key property that we need to conclude is that `x / 2` is even.

- ask yourself: “**for what x is that true?**”

Alternative example AI: even/odd integers

Key property that we need to conclude is that `x / 2` is even.

- ask yourself: "**for what x is that true?**"
  - simplest answer: `x . x%4 = 0` - that is, all `x`s such that x is **divisible by 4**

Alternative example AI: even/odd integers

Key property that we need to conclude is that `x / 2` is even.

- ask yourself: “**for what x is that true?**”
  - simplest answer: `x . x%4 = 0` - that is, all `x`s such that x is **divisible by 4**
  - alternative answer: abstract value tracks the number of 2s in the prime factorization

Alternative example AI: even/odd integers

Key property that we need to conclude is that `x / 2` is even.

- ask yourself: “**for what x is that true?**”
  - simplest answer: `x . x%4 = 0` - that is, all `x`s such that x is **divisible by 4**
  - alternative answer: abstract value tracks the number of 2s in the prime factorization
- cunning plan: add a “divisible by 4” abstract value (**mod4**) to our lattice, then rebuild our transfer functions

Alternative example AI: even/odd integers

Next question: where does “divisible by 4” go in the **lattice**?

```
{ even, odd } = top
      /          \
  {even}      {odd}
      \          /
      {} = bottom
```

Alternative example AI: even/odd integers

Next question: where does “divisible by 4” go in the lattice?

all mod4 integers are also even!

```
{ even, odd } = top
     /          \
 {even}      {odd}
    |           |
 {mod4}         |
    \          /
    {} = bottom
```

Alternative example AI: even/odd integers

How to change our **transfer functions**? Let’s do two examples (+ and /):

Alternative example AI: even/odd integers

How to change our **transfer functions**? Let’s do two examples (+ and /):

recall our **original** transfer function for +:

+	T	even	odd	⊥
T	T	T	T	⊥
even	T	even	odd	⊥
odd	T	odd	even	⊥
⊥	⊥	⊥	⊥	⊥

Alternative example AI: even/odd integers

How to change our **transfer functions**? Let’s do two examples (+ and /):

recall our **original** transfer function for +:

we need to add a row and a column for **mod4**:

+	⊤	even	odd	mod4	⊥
⊤	⊤	⊤	⊤		⊥
even	⊤	even	odd		⊥
odd	⊤	odd	even		⊥
mod4					
⊥	⊥	⊥	⊥		⊥

Alternative example AI: even/odd integers

How to change our **transfer functions**? Let’s do two examples (+ and /):

recall our **original** transfer function for +:

we need to add a row and a column for **mod4**:

+	⊤	even	odd	mod4	⊥
⊤	⊤	⊤	⊤	⊤	⊥
even	⊤	even	odd	even	⊥
odd	⊤	odd	even	odd	⊥
mod4	⊤	even	odd	mod4	⊥
⊥	⊥	⊥	⊥	⊥	⊥

Alternative example AI: even/odd integers

How to change our **transfer functions**? Let’s do two examples (+ and /):

same thing for **division**:

↓/→	⊤	even	odd	mod4	⊥
⊤	⊤	⊤	⊤		⊥
even	⊤	⊤	⊤		⊥
odd	⊤	⊤	⊤		⊥
mod4					
⊥	⊥	⊥	⊥		⊥

Alternative example AI: even/odd integers

How to change our **transfer functions**? Let’s do two examples (+ and /):

same thing for **division**:

oh no! why is mod4 divided by even top?

- 4/4 = 1 :(
- we need **another** lattice element to make this work!

↓/→	⊤	even	odd	mod4	⊥
⊤	⊤	⊤	⊤	⊤	⊥
even	⊤	⊤	⊤	⊤	⊥
odd	⊤	⊤	⊤	⊤	⊥
mod4	⊤	⊤	⊤	⊤	⊥
⊥	⊥	⊥	⊥	⊥	⊥

Alternative example AI: even/odd integers

Another lattice element: “**is2**”

Alternative example AI: even/odd integers

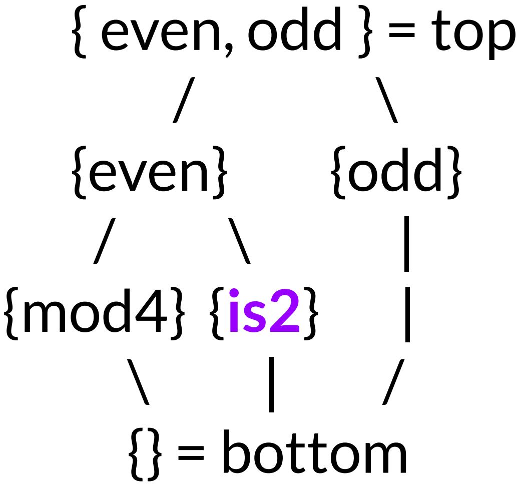
Another lattice element: “**is2**”

- **sibling** of mod4 in the lattice

Alternative example AI: even/odd integers

Another lattice element: “**is2**”

- **sibling** of mod4 in the lattice



Alternative example AI: even/odd integers

Another lattice element: “**is2**”

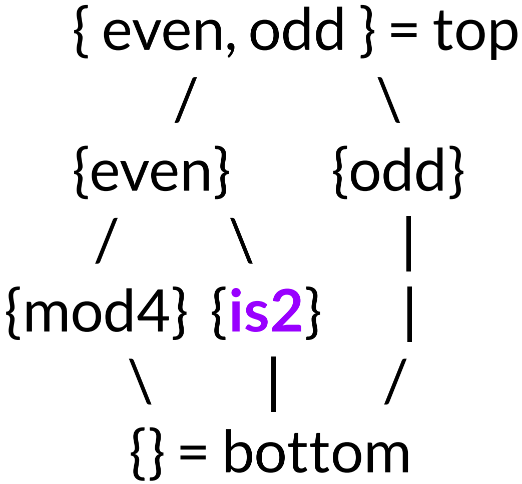
- **sibling** of mod4 in the lattice
- its only purpose is to be treated specially in the **division transfer function**

```
   { even, odd } = top
     /           \
   {even}      {odd}
    /     \        |
{mod4} {is2}      |
      \     |    /
      {} = bottom
```

Alternative example AI: even/odd integers

Another lattice element: “is2”

- sibling of mod4 in the lattice
- its only purpose is to be treated specially in the division transfer function
  - in particular, we add the rule “**mod4 / is2 -> even**”
  - full transfer functions left as an exercise



```
        { even, odd } = top
             /          \
        {even}      {odd}
       /      \          |
{mod4} {is2}       |
      \        |       /
        {} = bottom
```

Alternative example AI: let’s try it

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Abstract interpr.

```
{x=?;      y=?}
{x=?;      y=?}
{x=?;      y=?}
{x=?;      y=?}
{x=?;      y=?}
{x=?;      y=?}
```

Alternative example AI: let’s try it

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Abstract interpr.

```
{x=e;     y=⊥}
{x=?;     y=?}
{x=?;     y=?}
{x=?;     y=?}
{x=?;     y=?}
{x=?;     y=?}
```

Alternative example AI: let’s try it

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Abstract interpr.

```
{x=e;       y=⊥}
{x=e;       y=e}
{x=?;       y=?}
{x=?;       y=?}
{x=?;       y=?}
{x=?;       y=?}
```

Alternative example AI: let’s try it

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Abstract interpr.

```
{x=e;     y=⊥}
{x=e;     y=e}
{x=o;     y=e}
{x=?;     y=?}
{x=?;     y=?}
{x=?;     y=?}
```

Alternative example AI: let’s try it

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Abstract interpr.

```
{x=e;     y=⊥}
{x=e;     y=e}
{x=o;     y=e}
{x=o;     y=e}
{x=?;     y=?}
{x=?;     y=?}
```

Alternative example AI: let’s try it

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Abstract interpr.

```
{x=e;     y=⊥}
{x=e;     y=e}
{x=o;     y=e}
{x=o;     y=e}
{x=?;     y=?}
{x=?;     y=?}
```

what should the transfer function for even - is2 be?

Alternative example AI: let’s try it

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Abstract interpr.

```
{x=e;       y=⊥}
{x=e;       y=e}
{x=o;       y=e}
{x=o;       y=e}
{x=?;       y=?}
{x=?;       y=?}
```

what should the transfer function for even - is2 be?

- **even! why not mod4?**

Alternative example AI: let’s try it

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Abstract interpr.

```
{x=e;      y=⊥}
{x=e;      y=e}
{x=o;      y=e}
{x=o;      y=e}
{x=?;      y=?}
{x=?;      y=?}
```

what should the transfer function for even - is2 be?

- **even! why not mod4? counterexample: 8 - 2 = 6**

Alternative example AI: let’s try it

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Abstract interpr.

```
{x=e;     y=⊥}
{x=e;     y=e}
{x=o;     y=e}
{x=o;     y=e}
{x=e;     y=e}
{x=?;     y=?}
```

Alternative example AI: let’s try it

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Abstract interpr.

```
{x=e;     y=⊥}
{x=e;     y=e}
{x=o;     y=e}
{x=o;     y=e}
{x=e;     y=e}
{x=e;     y=T}
```

Alternative example AI: even/odd integers

- Why did adding **is2** and **mod4** fail to fix the approximation problem in the example?

Alternative example AI: even/odd integers

- Why did adding is2 and mod4 fail to fix the approximation problem in the example?
  - the example relies on the fact that for all X, (X + 1) * 2 - 2 = 2X
    - and if X is initially even, then this means that the result is divisible by 4

Alternative example AI: even/odd integers

- Why did adding is2 and mod4 fail to fix the approximation problem in the example?
  - the example relies on the fact that for all X, (X + 1) * 2 - 2 = 2X
    - and if X is initially even, then this means that the result is divisible by 4
- **lesson from this example**: most programs rely on complex invariants, and designing an abstract domain that can capture those invariants is **hard**!

Alternative example AI: even/odd integers

- Why did adding is2 and mod4 fail to fix the approximation problem in the example?
    - the example relies on the fact that for all X, (X + 1) * 2 - 2 = 2X
        - and if X is initially even, then this means that the result is divisible by 4
- **lesson from this example**: most programs rely on complex invariants, and designing an abstract domain that can capture those invariants is **hard**!
- how could we get the right answer on this example?

Alternative example AI: even/odd integers

- Why did adding **is2** and **mod4** fail to fix the approximation problem in the example?
    - the example relies on the fact that for all X, (X + 1) * 2 - 2 = 2X
        - and if X is initially even, then this means that the result is divisible by 4
- **lesson from this example**: most programs rely on complex invariants, and designing an abstract domain that can capture those invariants is **hard**!
- how could we get the right answer on this example?
    - more complex abstract values, e.g., oddTimes2?
    - store the mathematical expression for each variable?

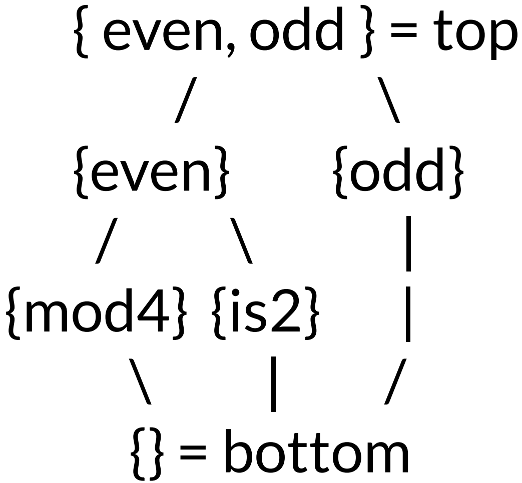
Alternative example AI: even/odd integers

- Why did adding **is2** and **mod4** fail to fix the approximation problem in the example?
  - the example relies on the fact that for all X, (X + 1) * 2 - 2 = 2X
    - and if X is initially even, then this means that the result is divisible by 4
- **lesson from this example**: most programs rely on complex invariants, and designing an abstract domain that can capture those invariants is **hard**!
- how could we get the right answer on this example?
  - more complex abstract values, e.g., oddTimes2?
  - store the mathematical expression for each variable?

one more try…

Alternative example AI: even/odd integers

Yet another lattice element: “**odd2**”

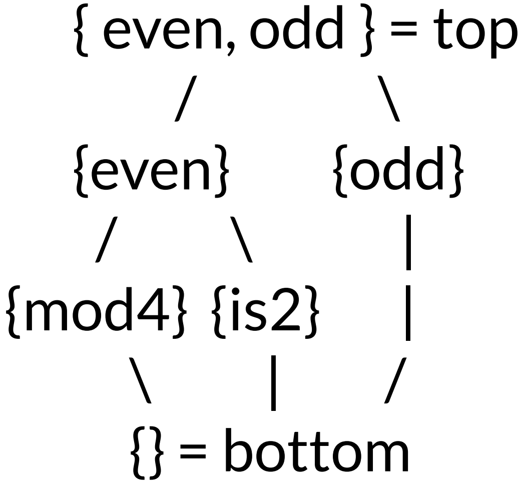


```
      { even, odd } = top
           /          \
      {even}       {odd}
       /      \          |
{mod4} {is2}      |
       \       |       /
         {} = bottom
```

Alternative example AI: even/odd integers

Yet another lattice element: “**odd2**”

- produced by multiplying an odd number by 2 (i.e., transfer fcn for odd * is2 -> odd2)



```
        { even, odd } = top
           /            \
       {even}        {odd}
       /      \           |
{mod4} {is2}       |
       \        |       /
        {} = bottom
```

Alternative example AI: even/odd integers

Yet another lattice element: “**odd2**”

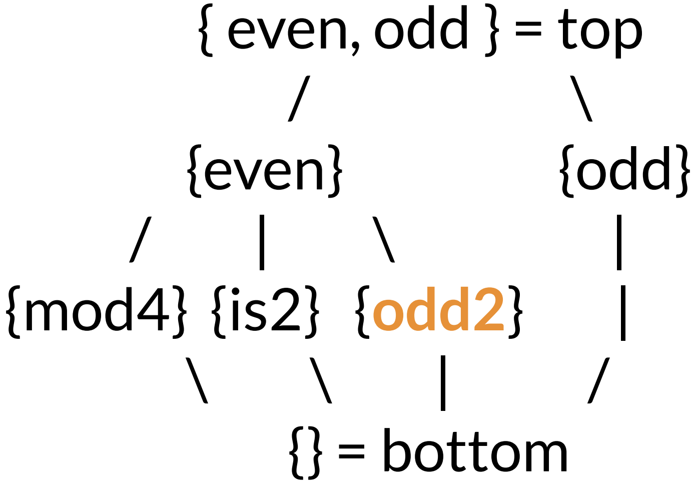
- produced by multiplying an odd number by 2 (i.e., transfer fcn for odd * is2 -> odd2)
- **where does it go** in the lattice?

```
   { even, odd } = top
        /        \
   {even}      {odd}
    /     \        |
{mod4} {is2}       |
      \     |     /
      {} = bottom
```

Alternative example AI: even/odd integers

Yet another lattice element: “odd2”

- produced by multiplying an odd number by 2 (i.e., transfer fcn for odd * is2 -> odd2)
- **where does it go** in the lattice?
  - a sibling of is2 and mod4?



{ even, odd } = top

{even}

{odd}

{mod4}

{is2}

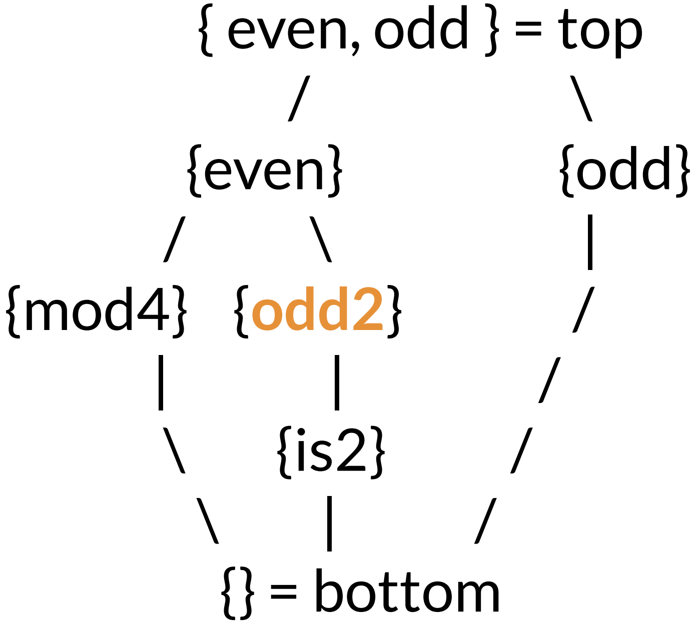
{odd2}

{} = bottom

Alternative example AI: even/odd integers

Yet another lattice element: “odd2”

- produced by multiplying an odd number by 2 (i.e., transfer fcn for odd * is2 -> odd2)
- **where does it go** in the lattice?
    - ~~a sibling of is2 and mod4?~~
    - between even and is2!

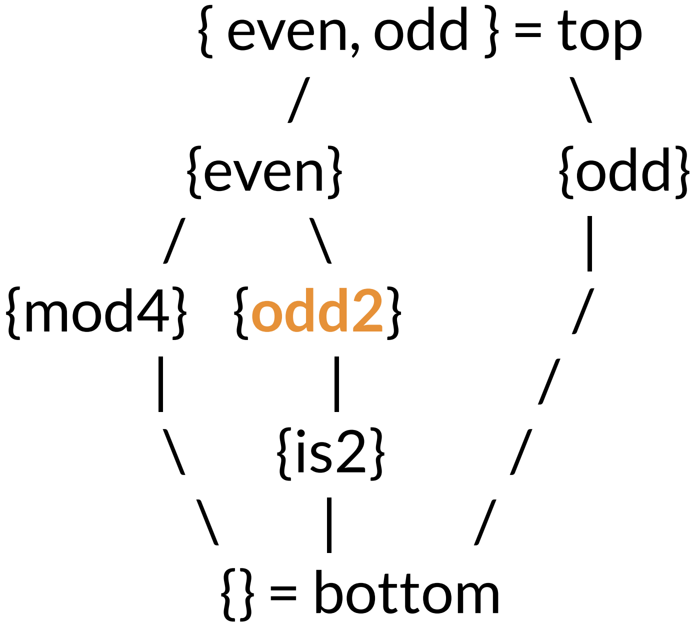


```
          { even, odd } = top
              /              \
          {even}              {odd}
         /      \               |
 {mod4}   {odd2}               /
    |        |                /
    \      {is2}             /
     \       |              /
        {} = bottom
```

Alternative example AI: even/odd integers

Yet another lattice element: “odd2”

- produced by multiplying an odd number by 2 (i.e., transfer fcn for odd * is2 -> odd2)
- **where does it go** in the lattice?
  - ~~a sibling of is2 and mod4?~~
  - between even and is2!
  - now we can add a new rule:



{ even, odd } = top

{even}

{odd}

{mod4}

{odd2}

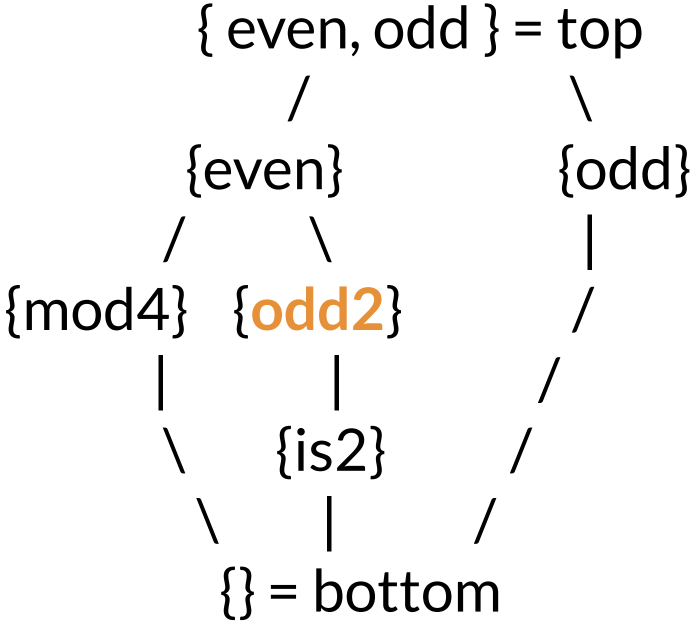
{is2}

{} = bottom

Alternative example AI: even/odd integers

Yet another lattice element: "odd2"

- produced by multiplying an odd number by 2 (i.e., transfer fcn for odd * is2 -> odd2)
- **where does it go** in the lattice?
    - ~~a sibling of is2 and mod4?~~
    - between even and is2!
    - now we can add a new rule:
        - odd2 - is2 -> mod4



{ even, odd } = top

{even}

{odd}

{mod4}

{odd2}

{is2}

{} = bottom

Alternative example AI: another attempt

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Abstract interpr.

```
{x=?;      y=?}
{x=?;      y=?}
{x=?;      y=?}
{x=?;      y=?}
{x=?;      y=?}
{x=?;      y=?}
```

Alternative example AI: another attempt

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Abstract interpr.

```
{x=e;        y=⊥}
{x=?;        y=?}
{x=?;        y=?}
{x=?;        y=?}
{x=?;        y=?}
{x=?;        y=?}
```

Alternative example AI: another attempt

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Abstract interpr.

```
{x=e;     y=⊥}
{x=e;     y=e}
{x=?;     y=?}
{x=?;     y=?}
{x=?;     y=?}
{x=?;     y=?}
```

Alternative example AI: another attempt

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Abstract interpr.

```
{x=e;     y=⊥}
{x=e;     y=e}
{x=o;     y=e}
{x=?;     y=?}
{x=?;     y=?}
{x=?;     y=?}
```

Alternative example AI: another attempt

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Abstract interpr.

```
{x=e;       y=⊥}
{x=e;       y=e}
{x=o;       y=e}
{x=o;       y=odd2}
{x=?;       y=?}
{x=?;       y=?}
```

Alternative example AI: another attempt

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Abstract interpr.

```
{x=e;    y=⊥}
{x=e;    y=e}
{x=o;    y=e}
{x=o;    y=odd2}
{x=mod4; y=odd2}
{x=?;    y=?}
```

Alternative example AI: another attempt

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Abstract interpr.

```
{x=e;    y=⊥}
{x=e;    y=e}
{x=o;    y=e}
{x=o;    y=odd2}
{x=mod4; y=odd2}
{x=mod4; y=e}
```

Success!

Formalizing the AI algorithm

- the core algorithm for abstract interpretation is the same one we saw last week for dataflow analysis:

Formalizing the AI algorithm

- the core algorithm for abstract interpretation is the same one we saw last week for dataflow analysis:

1. convert the program to a CFG

Formalizing the AI algorithm

- the core algorithm for abstract interpretation is the same one we saw last week for dataflow analysis:

1. convert the program to a CFG
2. start with an initial estimate at every program point (usually ⊥)

Formalizing the AI algorithm

- the core algorithm for abstract interpretation is the same one we saw last week for dataflow analysis:

1. convert the program to a CFG
2. start with an initial estimate at every program point (usually ⊥)
3. put each program point in a worklist

Formalizing the AI algorithm

- the core algorithm for abstract interpretation is the same one we saw last week for dataflow analysis:

1. convert the program to a CFG
2. start with an initial estimate at every program point (usually $\perp$)
3. put each program point in a worklist
4. until the worklist is empty, choose an item from the worklist and:

Formalizing the AI algorithm

- the core algorithm for abstract interpretation is the same one we saw last week for dataflow analysis:

1. convert the program to a CFG
2. start with an initial estimate at every program point (usually $\bot$)
3. put each program point in a worklist
4. until the worklist is empty, choose an item from the worklist and:
    a. if the item is a basic block, abstractly execute it using the transfer functions (and abstraction function, if applicable)

Formalizing the AI algorithm

- the core algorithm for abstract interpretation is the same one we saw last week for dataflow analysis:

1. convert the program to a CFG
2. start with an initial estimate at every program point (usually ⊥)
3. put each program point in a worklist
4. until the worklist is empty, choose an item from the worklist and:
    a. if the item is a basic block, abstractly execute it using the transfer functions (and abstraction function, if applicable)
    b. if the item is a join point, use the LUB to combine its inputs

Formalizing the AI algorithm

- the core algorithm for abstract interpretation is the same one we saw last week for dataflow analysis:

1. convert the program to a CFG
2. start with an initial estimate at ⊥)
3. put each program point in a wo
4. until the worklist is empty, choo and:
   a. if the item is a basic block, a transfer functions (and abstraction function, if applicable)
   b. if the item is a join point, use the LUB to combine its inputs

Using LUB at join points models the fact that the program may **take either branch** of an if statement.

Formalizing the AI algorithm

- the core algorithm for abstract interpretation is the same one we saw last week for dataflow analysis:

1. convert the program to a CFG
2. start with an initial estimate at every program point (usually ⊥)
3. put each program point in a worklist
4. until the worklist is empty, choose an item from the worklist and:
    a. if the item is a basic block, abstractly execute it using the transfer functions (and abstraction function, if applicable)
    b. if the item is a join point, use the LUB to combine its inputs
    c. if either a. or b. caused a change, re-add dependent blocks to the worklist

What about loops?

What about loops?

- this algorithm terminates for the same reasons that any dataflow algorithm does:

What about loops?

- this algorithm terminates for the same reasons that any dataflow algorithm does:
    - the lattice is of **finite size**
    - LUB is **monotonic**

What about loops?

- this algorithm terminates for the same reasons that any dataflow algorithm does:
  - the lattice is of **finite size**
  - LUB is **monotonic**

You may be surprised that it is possible to build an abstract interpretation using (some) infinite-height lattices. Next week, we'll discuss ***widening***, which is the technique for this.

What about loops?

- this algorithm terminates for the same reasons that any dataflow algorithm does:
    - the lattice is of **finite size**
    - LUB is **monotonic**
- that is, each loop will be analyzed at most *k-1* times for each variable in the loop, where *k* is the height of the lattice

What about loops?

- this algorithm terminates for the same reasons that any dataflow algorithm does:
  - the lattice is of **finite size**
  - LUB is **monotonic**
- that is, each loop will be analyzed at most *k-1* times for each variable in the loop, where *k* is the height of the lattice
- otherwise, loops are just a join point and a back-edge in the CFG

Why start with bottom?

Why start with bottom?

- the abstract interpretations we've considered so far are **optimistic**: they start with $\perp$ and then go upwards in the lattice

Why start with bottom?

- the abstract interpretations we’ve considered so far are **optimistic**: they start with ⊥ and then go upwards in the lattice
    - these algorithms get the **most precise answer**

Why start with bottom?

- the abstract interpretations we’ve considered so far are optimistic: they start with ⊥ and then go upwards in the lattice
    - these algorithms get the **most precise answer**
    - but their downside is that they **must run to fixpoint** - they cannot be stopped early (the result might still be unsound)!

Why start with bottom?

- the abstract interpretations we've considered so far are **optimistic**: they start with ⊥ and then go upwards in the lattice
    - these algorithms get the **most precise answer**
    - but their downside is that they **must run to fixpoint** - they cannot be stopped early (the result might still be unsound)!
- **pessimistic** algorithms are also possible

Why start with bottom?

- the abstract interpretations we've considered so far are **optimistic**: they start with ⊥ and then go upwards in the lattice
  - these algorithms get the **most precise answer**
  - but their downside is that they **must run to fixpoint** - they cannot be stopped early (the result might still be unsound)!
- **pessimistic** algorithms are also possible
  - start with T everywhere and move downwards in the lattice

Why start with bottom?

- the abstract interpretations we’ve considered so far are optimistic: they start with ⊥ and then go upwards in the lattice
  - these algorithms get the **most precise answer**
  - but their downside is that they **must run to fixpoint** - they cannot be stopped early (the result might still be unsound)!
- **pessimistic** algorithms are also possible
  - start with T everywhere and move downwards in the lattice
  - can be stopped at any time (e.g., when a budget is reached), but answer may not be precise

Another example

Another example

- Consider an abstract interpretation for ***constant propagation***

Another example

- Consider an abstract interpretation for ***constant propagation***
  - the goal of constant propagation is to determine whether, for each variable, its value can be known at compile time

Another example

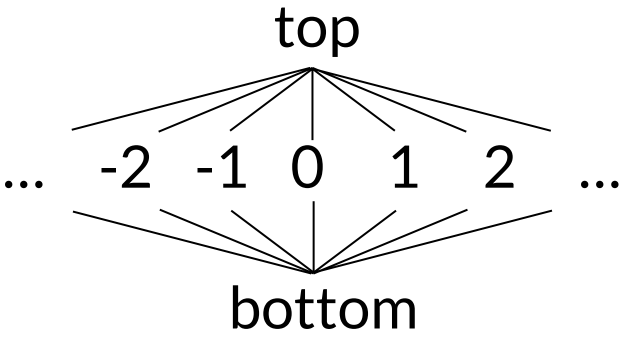
- Consider an abstract interpretation for ***constant propagation***
  - the goal of constant propagation is to determine whether, for each variable, its value can be known at compile time
  - constant propagation is a standard compiler optimization

Another example

- Consider an abstract interpretation for ***constant propagation***
    - the goal of constant propagation is to determine whether, for each variable, its value can be known at compile time
    - constant propagation is a standard compiler optimization
    - lattice:

Another example

- Consider an abstract interpretation for ***constant propagation***
  - the goal of constant propagation is to determine whether, for each variable, its value can be known at compile time
  - constant propagation is a standard compiler optimization
  - lattice:



Another example

Consider the following program:

```
w = 5
x = read()
if (x is even)
  y = 5
  w = w + y
else
  y = 10
  w = y
z = y + 1
x = 2 * w
```

Correctness of Abstract Interpretation

Correctness of Abstract Interpretation

- I’ve claimed several times that it is possible to use abstract interpretation to produce **sound** program analyses

Correctness of Abstract Interpretation

- I’ve claimed several times that it is possible to use abstract interpretation to produce **sound** program analyses
    - that is, analyses without false negatives

Correctness of Abstract Interpretation

- I’ve claimed several times that it is possible to use abstract interpretation to produce **sound** program analyses
    - that is, analyses without false negatives
- The key idea to demonstrate that an abstract interpretation is sound is the ***galois connection*** between a concrete value and the concretization of its abstraction function

Correctness of Abstract Interpretation

- I’ve claimed several times that it is possible to use abstract interpretation to produce **sound** program analyses
  - that is, analyses without false negatives
- The key idea to demonstrate that an abstract interpretation is sound is the ***galois connection*** between a concrete value and the concretization of its abstraction function
  - ideally, we’d like $\forall$ x, $\gamma(\alpha(x)) = x$

Correctness of Abstract Interpretation

- I’ve claimed several times that it is possible to use abstract interpretation to produce **sound** program analyses
  - that is, analyses without false negatives
- The key idea to demonstrate that an abstract interpretation is sound is the ***galois connection*** between a concrete value and the concretization of its abstraction function
  - ideally, we’d like $\forall$ x, $\gamma(\alpha(x)) = x$
  - but this is too strong: approximation may cause us to lose information! So, the standard formalism is:
    - $\forall$ x, $x \in \gamma(\alpha(x))$

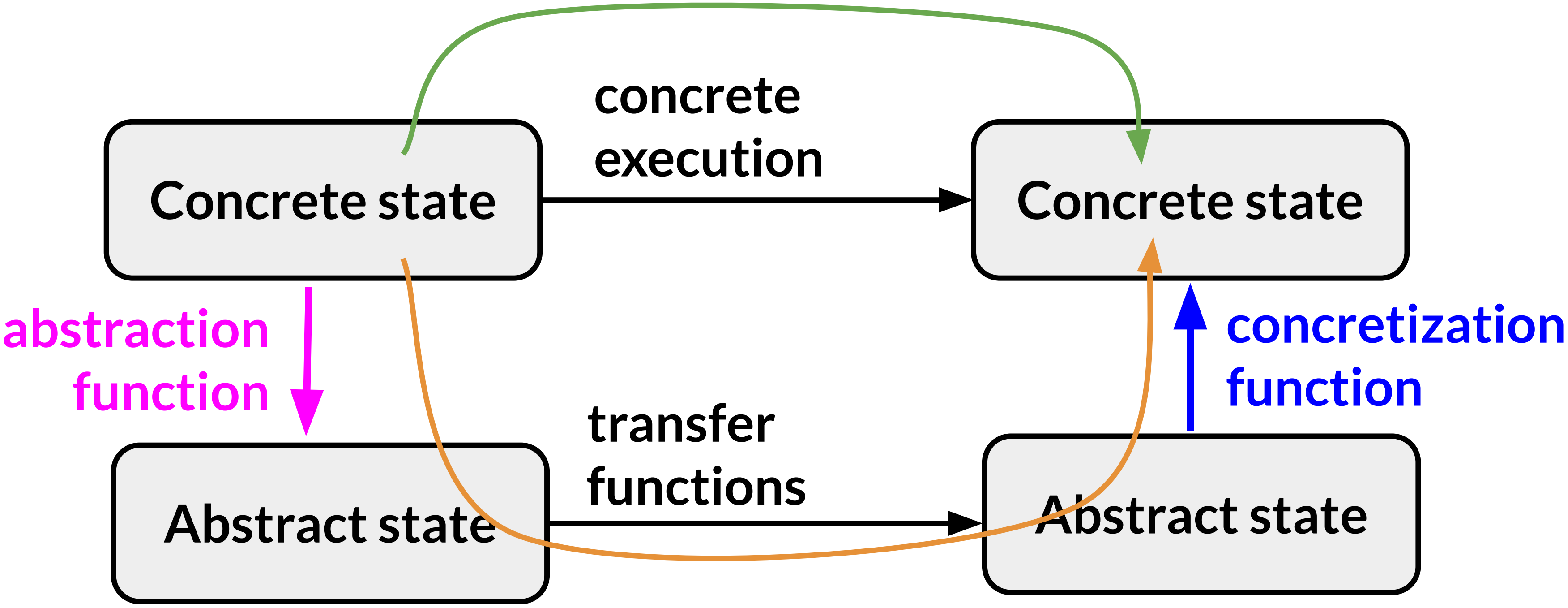
Correctness of Abstract Interpretation

- I’ve claimed several times that it is possible to use abstract interpretation to produce **sound** program analyses
  - that is, analyses without false negatives
- The key idea to demonstrate that an abstract interpretation is sound is the ***galois connection*** between a concrete value and the concretization of its abstraction
  - ideally, we’d like $\forall$ x, $\gamma(\alpha(x))$
  - but this is too strong: appro information! So, the standa
    - $\forall x, x \in \gamma(\alpha(x))$

And, it’s also necessary to show that the Galois connection holds for the **transfer functions**!

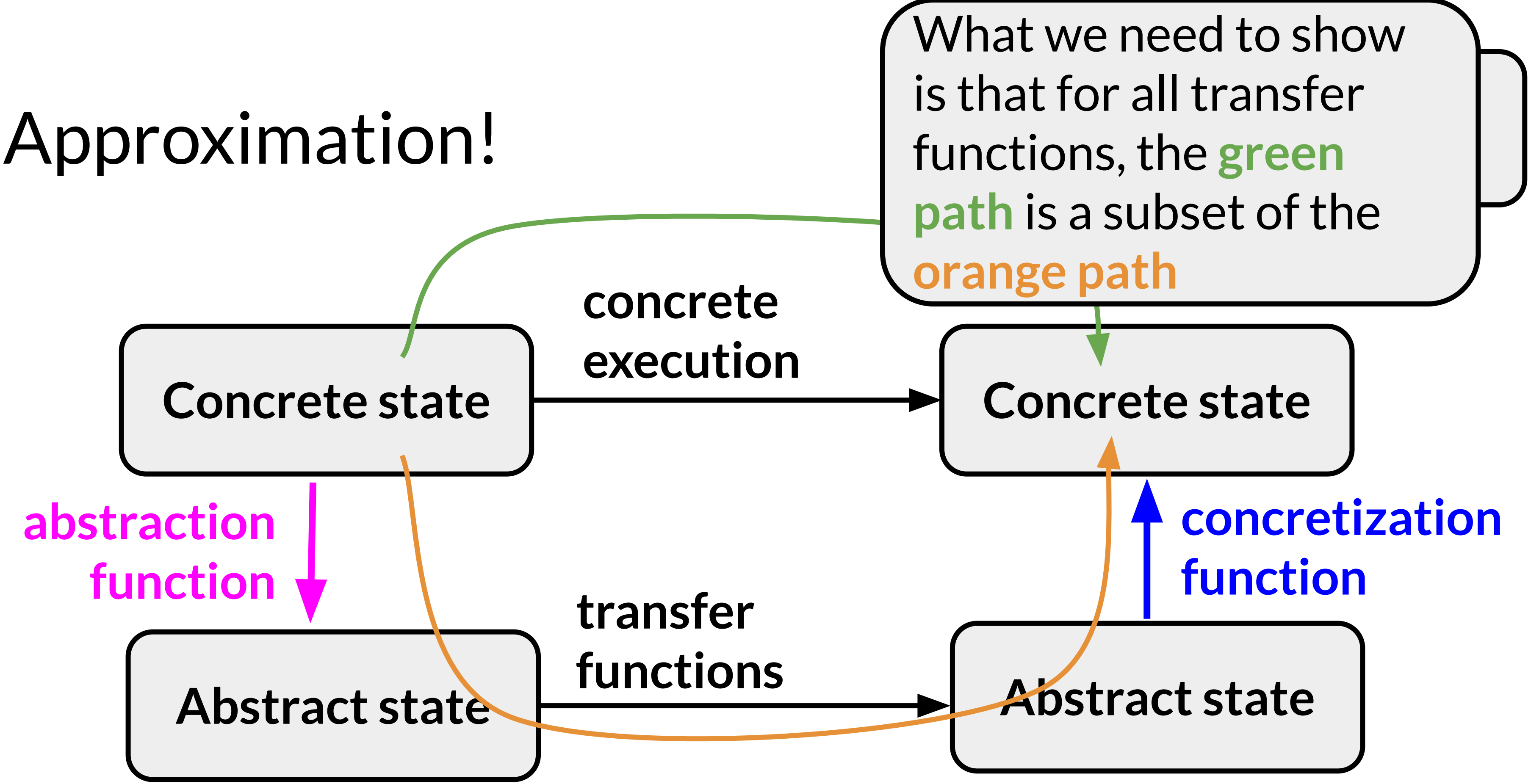
Approximation!

Remember this diagram from earlier?



Do the green and orange paths always lead to the same concrete state?

Approximation!



Do the green and orange paths always lead to the same concrete state?