Abstract Interpretation (2/2)

Martin Kellogg

Reading quiz: abstract interpretation

Reading quiz: abstract interpretation

- today's quiz is on paper, and also covers the topics of last week's class
- you have 15 minutes to complete it. When you're finished, bring it to Kazi in the back.
- you may use any **hand-written** notes that you took during last class
    - this includes notes on a tablet or similar, if you wrote them with a stylus
        - but I will be looking over your shoulder if you do :)

Agenda: abstract interpretation, part 2

- review and clarifications from last week
- more on soundness
- refinement and branching
- widening
- Stein’s algorithm example
- analysis implementation demo

Agenda: abstract interpretation, part 2

- **review and clarifications from last week**
- more on soundness
- refinement and branching
- widening
- Stein’s algorithm example
- analysis implementation demo

Review: definitions

Review: definitions

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

Review: definitions

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

- an **abstract domain** over which to reason, which is composed of:

Review: definitions

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

- an **abstract domain** over which to reason, which is composed of:
    - a set of **abstract values**

Review: definitions

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

- an **abstract domain** over which to reason, which is composed of:
    - a set of **abstract values**
    - an **ordering operation** (e.g., LUB)

Review: definitions

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

- an **abstract domain** over which to reason, which is composed of:
  - a set of **abstract values**
  - an **ordering operation** (e.g., LUB)
  - together these form a **lattice**

Review: definitions

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

- an **abstract domain** over which to reason, which is composed of:
  - a set of **abstract values**
  - an **ordering operation** (e.g., LUB)
  - together these form a **lattice**
- a set of **transfer functions** that tell the abstract interpreter how to reason over that abstract domain

Review: definitions

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

- an **abstract domain** over which to reason, which is composed of:
  - a set of **abstract values**
  - an **ordering operation** (e.g., LUB)
  - together these form a **lattice**
- a set of **transfer functions** that tell the abstract interpreter how to reason over that abstract domain
  - one for each kind of operation in the underlying programming language (e.g., one for +, one for -, etc.)

Review: definitions

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

- an **abstract domain** over which to reason, which is composed of:
  - a set of **abstract values**
  - an **ordering operation** (e.g., LUB)
  - together these form a **lattice**
- a set of **transfer functions** that tell the abstract interpreter how to reason over that abstract domain
  - one for each kind of operation in the underlying programming language (e.g., one for +, one for -, etc.)
  - usually represented as tables

Concrete vs abstract domains

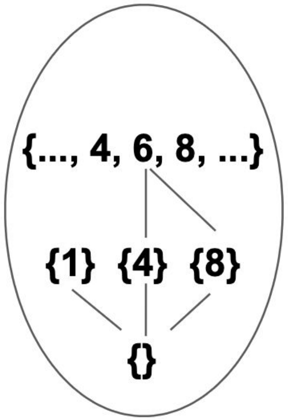
Concrete vs abstract domains

- the ***concrete domain*** of a variable is the set of values that the variable might actually take on during execution

Concrete vs abstract domains

- the *concrete domain* of a variable is the set of values that the variable might actually take on during execution

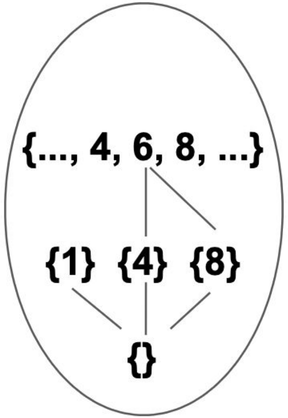
concrete domain



Concrete vs abstract domains

- the *concrete domain* of a variable is the set of values that the variable might actually take on during execution
- an *abstract domain* is a layer of indirection on top of the concrete domain that splits it into a smaller number of sets

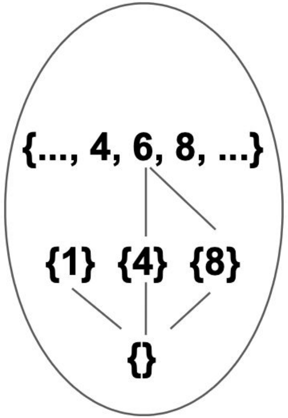
concrete domain

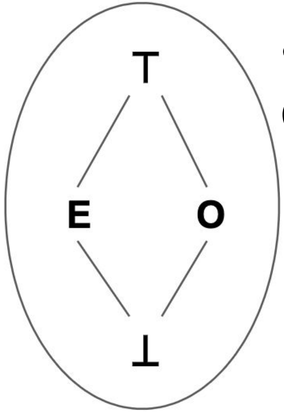


Concrete vs abstract domains

- the *concrete domain* of a variable is the set of values that the variable might actually take on during execution
- an *abstract domain* is a layer of indirection on top of the concrete domain that splits it into a smaller number of sets

concrete domain



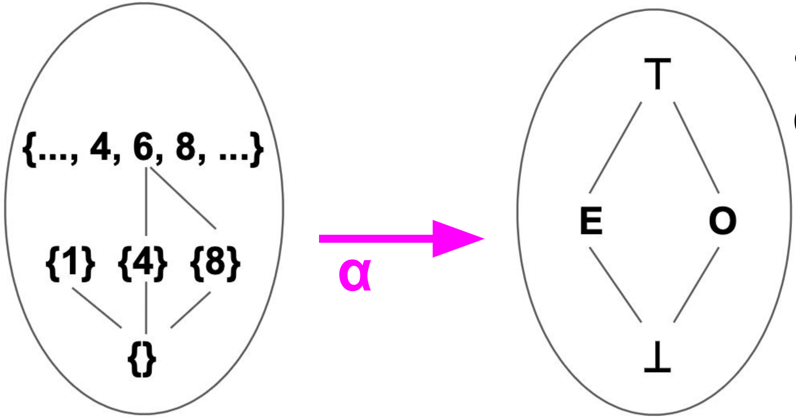


abstract domain

Concrete vs abstract domains

- the *concrete domain* of a variable is the set of values that the variable might actually take on during execution
- an ***abstract domain*** is a layer of indirection on top of the concrete domain that splits it into a smaller number of sets

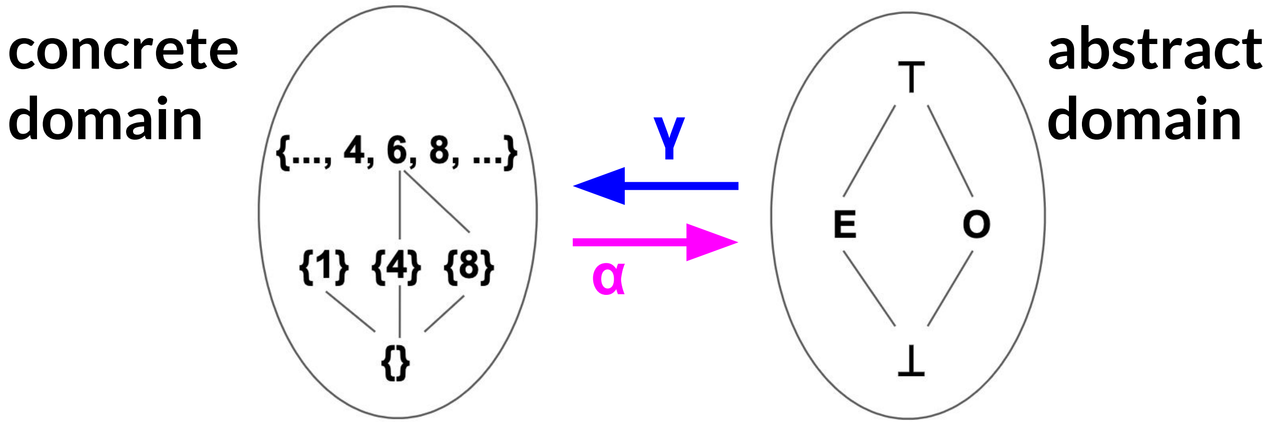
concrete domain



abstract domain

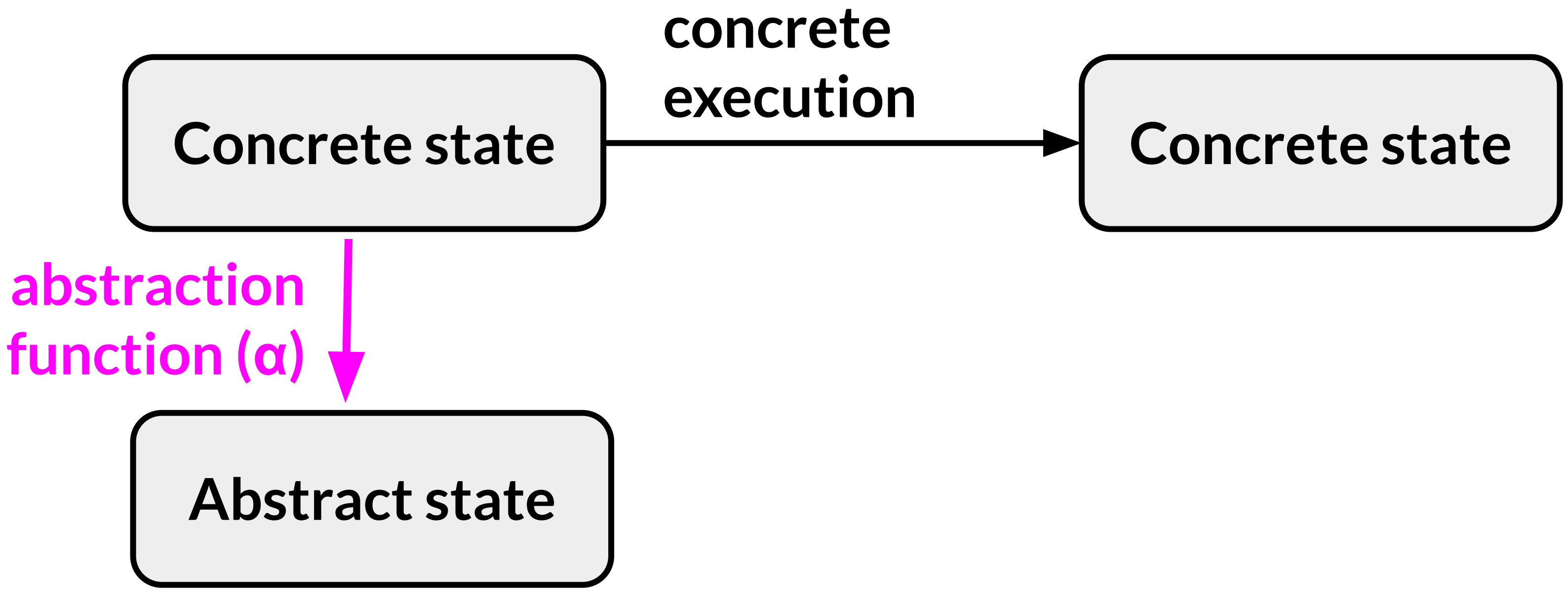
Concrete vs abstract domains

- the *concrete domain* of a variable is the set of values that the variable might actually take on during execution
- an *abstract domain* is a layer of indirection on top of the concrete domain that splits it into a smaller number of sets

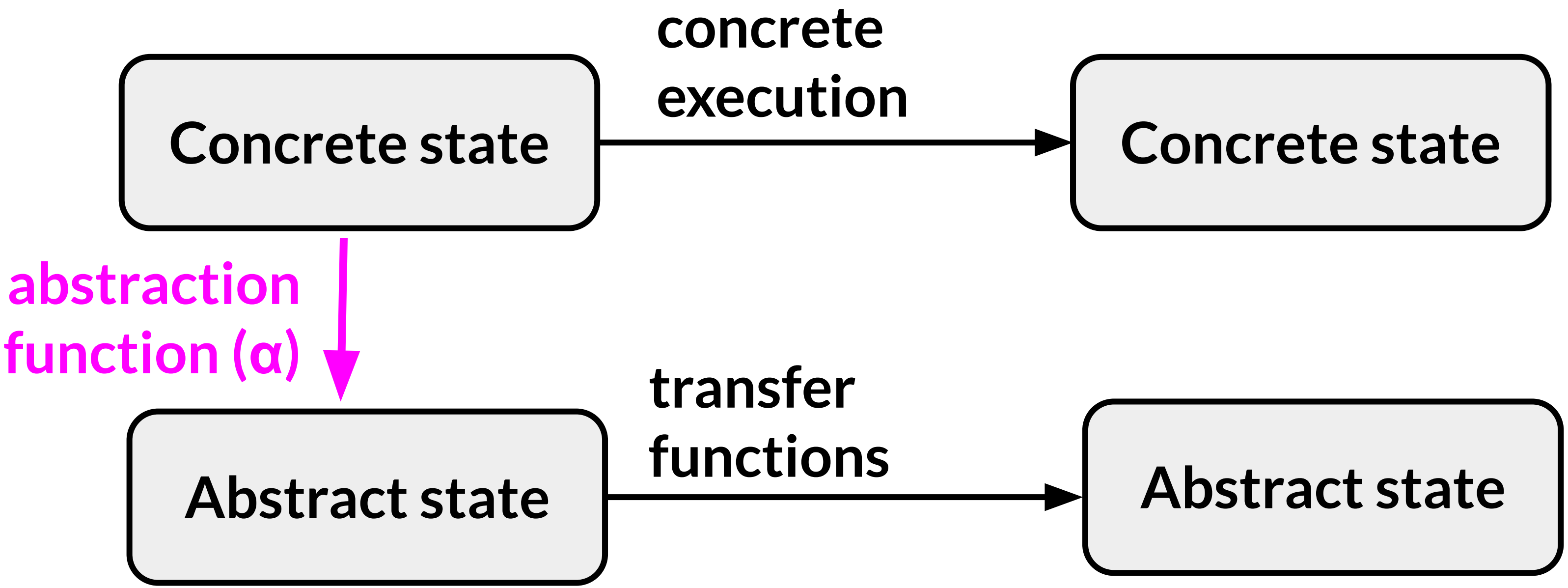


Review: abstract vs concrete interpretation

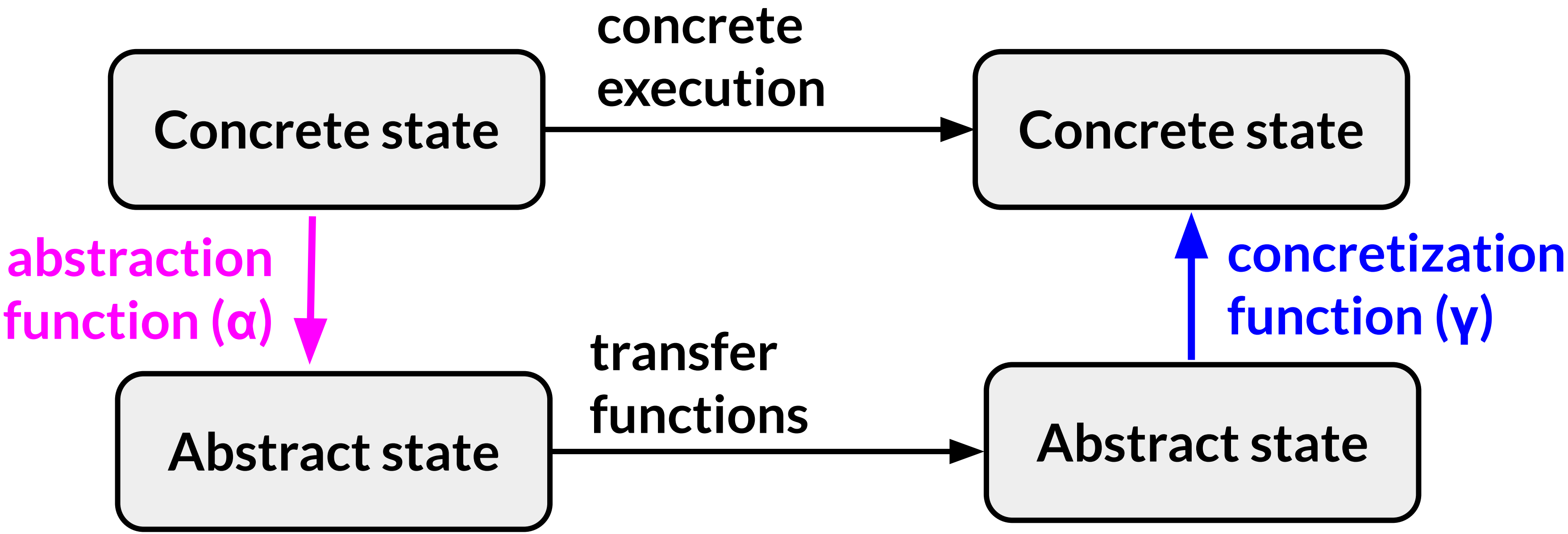
Review: abstract vs concrete interpretation



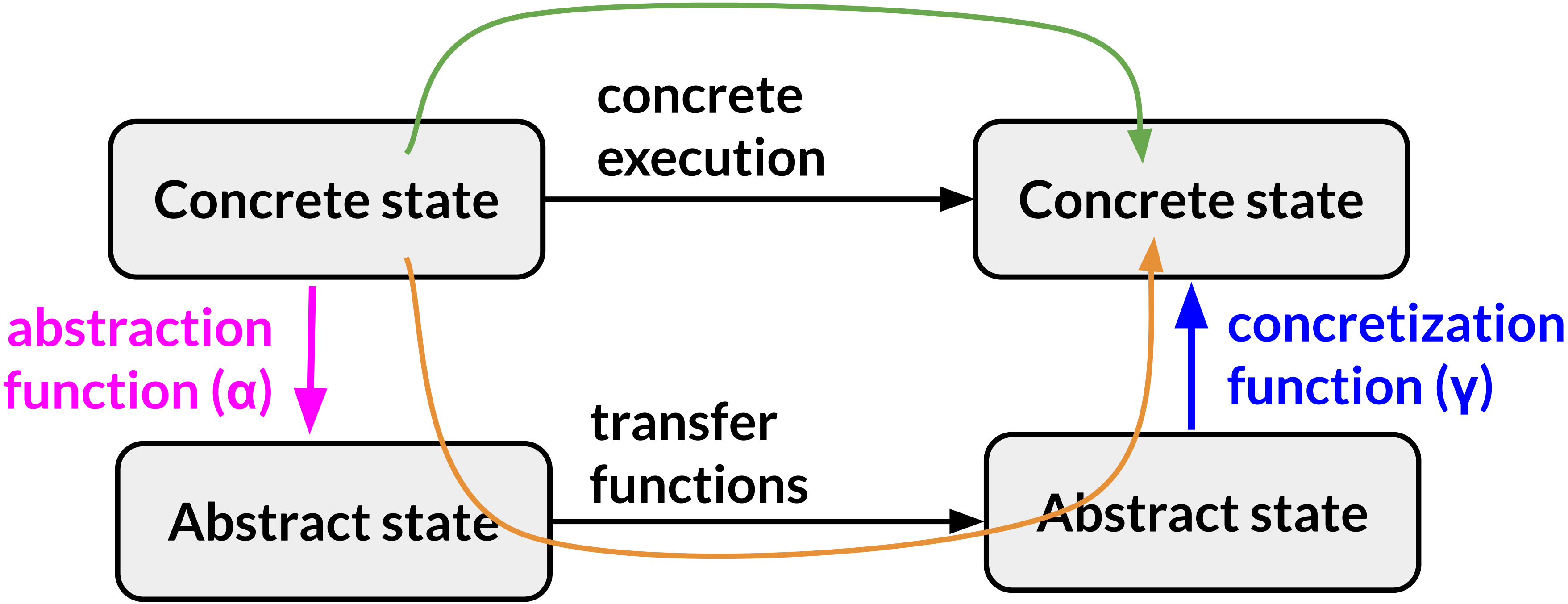
Review: abstract vs concrete interpretation



Review: abstract vs concrete interpretation



Review: abstract vs concrete interpretation



soundness means that the **green path** is a subset of the **orange path**

Review: clarifications

Review: clarifications

- last week, I went through an extended example of how to get a parity analysis to work on **one program**

Review: clarifications

- last week, I went through an extended example of how to get a parity analysis to work on **one program**
    - however, that was just an example!
        - an abstract interpretation is applicable to **any program**

Review: clarifications

- last week, I went through an extended example of how to get a parity analysis to work on **one program**
  - however, that was just an example!
    - an abstract interpretation is applicable to **any program**
  - one of the **key challenges** in abstract interpretation design is figuring out the **right set of examples** to handle precisely

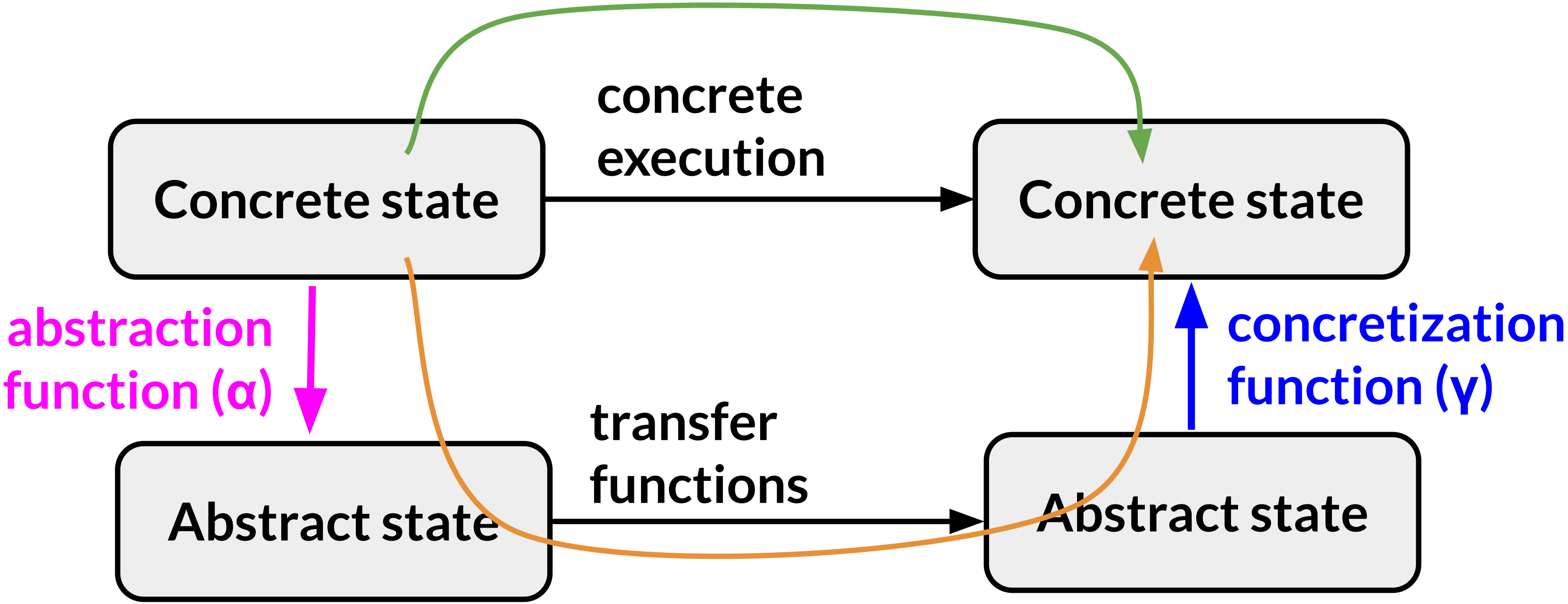
Review: clarifications

- last week, I went through an extended example of how to get a parity analysis to work on **one program**
  - however, that was just an example!
    - an abstract interpretation is applicable to **any program**
  - one of the **key challenges** in abstract interpretation design is figuring out the **right set of examples** to handle precisely
    - when you're implementing your divide-by-zero analysis, I **strongly recommend** that you write out some examples as test cases!
      - you can just add them to the existing test

Agenda: abstract interpretation, part 2

- review and clarifications from last week
- **more on soundness**
- refinement and branching
- widening
- Stein’s algorithm example
- analysis implementation demo

More on soundness: using a Galois connection



soundness means that the **green path** is a subset of the **orange path**

More on soundness: using a Galois connection

- how would we actually show that a particular abstract interpretation is **sound**?

More on soundness: using a Galois connection

- how would we actually show that a particular abstract interpretation is **sound**?
- here's an algorithm for doing so:

More on soundness: using a Galois connection

- how would we actually show that a particular abstract interpretation is **sound**?
- here's an algorithm for doing so:
  - for each transfer function T_{op} for some operation *op:*

More on soundness: using a Galois connection

- how would we actually show that a particular abstract interpretation is **sound**?
- here's an algorithm for doing so:
  - for each transfer function T_{op} for some operation *op:*
    - prove that for all concrete states *c*:

More on soundness: using a Galois connection

- how would we actually show that a particular abstract interpretation is **sound**?
- here's an algorithm for doing so:
  - for each transfer function T_{op} for some operation *op:*
    - prove that for all concrete states *c*:

$$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$$

More on soundness: using a Galois connection

- how would we actually show that a particular abstract interpretation is **sound**?
- here's an algorithm for doing so:
  - for each transfer function T_{op} for some operation *op:*
    - prove that for all concrete states *c*:

$$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$$

possible results of concrete execution (**green line**)

More on soundness: using a Galois connection

- how would we actually show that a particular abstract interpretation is **sound**?
- here's an algorithm for doing so:
  - for each transfer function T_{op} for some operation *op*:
    - prove that for all concrete states *c*:

$$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$$

concretization

More on soundness: using a Galois connection

- how would we actually show that a particular abstract interpretation is **sound**?
- here's an algorithm for doing so:
    - for each transfer function T_{op} for some operation *op*:
        - prove that for all concrete states *c*:

$$op(c) \subseteq \gamma(\boldsymbol{T_{op}}(\alpha(c)))$$

concretization of the result of applying the **transfer function**

More on soundness: using a Galois connection

- how would we actually show that a particular abstract interpretation is **sound**?
- here's an algorithm for doing so:
  - for each transfer function T_{op} for some operation *op:*
    - prove that for all concrete states *c*:

$$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$$

concretization of the result of applying the **transfer function** to the **abstraction** of the original concrete state

More on soundness: using a Galois connection

- how would we actually show that a particular abstract interpretation is **sound**?
- here's an algorithm for doing so:
  - for each transfer function T_{op} for some operation *op:*
    - prove that for all concrete states *c*:

$$op(c) \subseteq \boxed{\gamma(T_{op}(\alpha(c)))}$$

concretization of the result of applying the **transfer function** to the **abstraction** of the original concrete state (**orange line**)

More on soundness: example proof

- let’s carry out an example proof using this technique ourselves on the **plus transfer function** from our simple parity analysis

$\boxed{op(c) \subseteq \gamma(T_{op}(\alpha(c)))}$

More on soundness: example proof

- let's carry out an example proof using this technique ourselves on the **plus transfer function** from our simple parity analysis

$\boxed{op(c) \subseteq \gamma(T_{op}(\alpha(c)))}$

More on soundness: example proof

- let's carry out an example proof using this technique ourselves on the **plus transfer function** from our simple parity analysis

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

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- let's carry out an example proof using this technique ourselves on the **plus transfer function** from our simple parity analysis

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

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

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- Let's first dispense with the easy cases:

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- Let's first dispense with the easy cases:
    - if the transfer function entry is **top**, then it's easy:

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- Let's first dispense with the easy cases:
  - if the transfer function entry is **top**, then it's easy:
    - $\forall\ c.\ op(c) \subseteq$ { all integers } is trivially true!

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- Let's first
  - if the … en it's easy:
    - ∀ … ly true!

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

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- Let's first
  - if the … en it's easy:
    - ∀ … ly true!

+	⊤	even	odd	⊥
⊤	$\mp$	$\mp$	$\mp$	⊥
even	$\mp$	even	odd	⊥
odd	$\mp$	odd	even	⊥
⊥	⊥	⊥	⊥	⊥

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- Let's first dispense with the easy cases:
  - if the transfer function entry is **top**, then it's easy:
    - $\forall$ c. $op(c) \subseteq$ { all integers } is trivially true!
  - if the transfer function entry is **bottom**, it's still pretty easy:

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- Let's first dispense with the easy cases:
  - if the transfer function entry is **top**, then it's easy:
    - $\forall\ c.\ op(c) \subseteq$ { all integers } is trivially true!
  - if the transfer function entry is **bottom**, it's still pretty easy:
    - for every entry in our transfer function that's bottom, one of the inputs is **also bottom**

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- Let's first dispense with the easy cases:
  - if the transfer function entry is **top**, then it's easy:
    - $\forall$ *c. op(c)* $\subseteq$ { all integers } is trivially true!
  - if the transfer function entry is **bottom**, it's still pretty easy:
    - for every entry in our transfer function that's bottom, one of the inputs is **also bottom**
    - *op({})* is always the empty set (it **can't be executed**)

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- Let's first dispense with the easy cases:
  - if the transfer function entry is **top**, then it's easy:
    - $\forall\ c.\ op(c) \subseteq$ { all integers } is trivially true!
  - if the transfer function entry is **bottom**, it's still pretty easy:
    - for every entry in our transfer function that's bottom, one of the inputs is **also bottom**
    - $op(\{\})$ is always the empty set (it **can't be executed**)
    - $\{\} \subseteq \{\}$

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- Let's first dispense with the easy cases:
  - if the transfer function entry is **top**, then it's easy:
    - $\forall$ *c. op(c)* $\subseteq$ { all integers } is trivially true!
  - if the transfer function entry is **bottom**, it's still pretty easy:
    - for every entry in our transfer function that's bottom, one of the inputs is **also bottom**
    - *op({})* is always the empty set (it **can't be executed**)
    - {} $\subseteq$ {}
    - QED

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- Let's first
  - if the … en it's easy:
    - ∀ … ly true!
  - if the … n, it's still pretty easy:
    - fo … ction that's bottom, one of
    - *op* … n't be executed)
    - {}
    - Q

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

$\boxed{op(c) \subseteq \gamma(T_{op}(\alpha(c)))}$

More on soundness: example proof

- Let's first ...
  - if the ... en it's easy:
    - ∀ ... ly true!
  - if the ... , it's still pretty easy:
    - fo ... ction that's bottom, one of
    - *op* ... **n't be executed**)
    - {}
    - Q

+	⊤	even	odd	⊥
⊤	∓	∓	∓	±
even	∓	even	odd	±
odd	∓	odd	even	±
⊥	±	±	±	±

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- Now we need to handle the more **complex cases** in the middle

$\boxed{op(c) \subseteq \gamma(T_{op}(\alpha(c)))}$

More on soundness: example proof

- Now we need to handle the more **complex cases** in the middle
  - we could do them one-by-one…

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- Now we need to handle the more **complex cases** in the middle
  - we could do them one-by-one…
  - but we can skip some because addition is commutative
    - so we don’t need to worry about order

$$\boxed{op(c) \subseteq \gamma(T_{op}(\alpha(c)))}$$

More on soundness: example proof

- Now we n… **cases** in the middle
  - we co…
  - but w… is commutative
    - so… rder

+	$\top$	even	odd	$\bot$
$\top$	$\cancel{\top}$	$\cancel{\top}$	$\cancel{\top}$	$\cancel{\bot}$
even	$\cancel{\top}$	even	**odd**	$\cancel{\bot}$
odd	$\cancel{\top}$	**odd**	even	$\cancel{\bot}$
$\bot$	$\cancel{\bot}$	$\cancel{\bot}$	$\cancel{\bot}$	$\cancel{\bot}$

in other words, the two **orange** cases are the same!

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- Now we need to handle the more **complex cases** in the middle
  - we could do them one-by-one…
  - but we can skip some because addition is commutative
    - so we don't need to worry about order
- So, we have three cases to deal with:

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- Now we need to handle the more **complex cases** in the middle
  - we could do them one-by-one…
  - but we can skip some because addition is commutative
    - so we don’t need to worry about order
- So, we have three cases to deal with:
  1. even integer + even integer is an even integer
  2. odd integer + odd integer is an even integer
  3. odd integer + even integer is an odd integer

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- Now we need to handle the more complex cases in the middle
  - we could do them one-by-one…
  - but we can skip some because addition is commutative
    - so we don't need to worry about order
- So, we have three cases to deal with:
  1. even integer + even integer is an even integer
  2. odd integer + odd integer is an even integer
  3. odd integer + even integer is an odd integer
- we dispatch these three by considering each case individually
  - they're all basically the same, so we're only going to do one

$$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$$

More on soundness: example proof

- Now we need to handle the more complex cases in the middle
  - we could do them one-by-one…
  - but we can skip some because addition is commutative
    - so we don't need to worry about order
- So, we have three cases to deal with:
  1. even integer + even integer is an even integer
  2. odd integer + odd integer is an even integer
  3. **odd integer + even integer is an odd integer**
- we dispatch these three by considering each case individually
  - they're all basically the same, so we're only going to do one

$\boxed{op(c) \subseteq \gamma(T_{op}(\alpha(c)))}$

More on soundness: example proof

- c is some addition statement $x + y$

$\boxed{op(c) \subseteq \gamma(T_{op}(\alpha(c)))}$

More on soundness: example proof

- *c* is some addition statement $x + y$
  - we know that concretely x is odd and y is even (why?)
    - formally, we would state this as $x \% 2 = 1$ and $y \% 2 = 0$

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- c is some addition statement $x + y$
  - we know that concretely x is odd and y is even (why?)
    - formally, we would state this as $x \% 2 = 1$ and $y \% 2 = 0$
- what is $op(c)$?

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- *c* is some addition statement $x + y$
  - we know that concretely *x* is odd and *y* is even (why?)
    - formally, we would state this as $x \% 2 = 1$ and $y \% 2 = 0$
- what is *op*(*c*)?
  - represent *x* as $2a + 1$ and *y* as $2b$ for some *a*, *b* (how?)

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- *c* is some addition statement $x + y$
  - we know that concretely *x* is odd and *y* is even (why?)
    - formally, we would state this as $x \% 2 = 1$ and $y \% 2 = 0$
- what is *op*(*c*)?
  - represent *x* as $2a + 1$ and *y* as $2b$ for some *a*, *b* (how?)
  - $2a + 1 + 2b = 2(a+b) + 1$, which we can easily prove is the set of all odd integers

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- *c* is some addition statement $x + y$
    - we know that concretely *x* is odd and *y* is even (why?)
        - formally, we would state this as $x \% 2 = 1$ and $y \% 2 = 0$
- what is $op(c)$?
    - represent *x* as $2a + 1$ and *y* as $2b$ for some *a*, *b* (how?)
    - $2a + 1 + 2b = 2(a+b) + 1$, which we can easily prove is the set of all odd integers
- what's $\alpha(c)$?

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- *c* is some addition statement $x + y$
  - we know that concretely *x* is odd and *y* is even (why?)
    - formally, we would state this as $x \% 2 = 1$ and $y \% 2 = 0$
- what is *op(c)*?
  - represent *x* as $2a + 1$ and *y* as $2b$ for some *a*, *b* (how?)
  - $2a + 1 + 2b = 2(a+b) + 1$, which we can easily prove is the set of all odd integers
- what's $\alpha(c)$?
  - $\alpha(x)$ is **odd** (the abstract value), and $\alpha(y)$ is **even** (the AV)

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- *c* is some addition statement $x + y$
  - we know that concretely *x* is odd and *y* is even (why?)
    - formally, we would state this as $x \% 2 = 1$ and $y \% 2 = 0$
- what is $op(c)$?
  - represent *x* as $2a + 1$ and *y* as $2b$ for some *a*, *b* (how?)
  - $2a + 1 + 2b = 2(a+b) + 1$, which we can easily prove is the set of all odd integers
- what's $\alpha(c)$?
  - $\alpha(x)$ is **odd** (the abstract value), and $\alpha(y)$ is **even** (the AV)
- $T_+(\alpha(c))$ is just applying our transfer function: result is the **odd** AV

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- *c* is some addition statement $x + y$
  - we know that concretely *x* is odd and *y* is even (why?)
    - formally, we would state this as $x \% 2 = 1$ and $y \% 2 = 0$
- what is *op*(*c*)?
  - represent *x* as $2a + 1$ and *y* as $2b$ for some *a*, *b* (how?)
  - $2a + 1 + 2b = 2(a+b) + 1$, which we can easily prove is the set of all odd integers
- what's $\alpha(c)$?
  - $\alpha(x)$ is **odd** (the abstract value), and $\alpha(y)$ is **even** (the AV)
- $T_+(\alpha(c))$ is just applying our transfer function: result is the **odd** AV
- $\gamma($**odd**$)$ is the set of all odd integers, which does contain itself

$op(c) \subseteq \gamma(T_{op}(\alpha(c)))$

More on soundness: example proof

- *c* is some addition statement $x + y$
  - we know that concretely *x* is odd and *y* is even (why?)
    - formally, we would state this as $x \% 2 = 1$ and $y \% 2 = 0$
- what is *op*(*c*)?
  - represent *x* as $2a + 1$ and *y* as $2b$ for some *a*, *b* (how?)
  - $2a + 1 + 2b = 2(a+b) + 1$, which we can easily prove is the set of all odd integers
- what's $\alpha(c)$?
  - $\alpha(x)$ is **odd** (the abstract value), and $\alpha(y)$ is **even** (the AV)
- $T_+(\alpha(c))$ is just applying our transfer function: result is the **odd** AV
- γ(**odd**) is the set of all odd integers, which does contain itself $\square$

Agenda: abstract interpretation, part 2

- review and clarifications from last week
- more on soundness
- **refinement and branching**
- widening
- Stein’s algorithm example
- analysis implementation demo

Refinement

Consider the following program:

```
x = 0
while (x < 3):
  x = x + 1
print x
```

Refinement

Consider the following program:

```
x = 0
while (x < 3):
  x = x + 1
print x
```

What value is printed?

Refinement

Consider the following program:

```
x = 0
while (x < 3):
  x = x + 1
print x
```

What value is printed?
How do you know?

Refinement

Consider the following program:

```
x = 0
while (x < 3):
  x = x + 1
print x
```

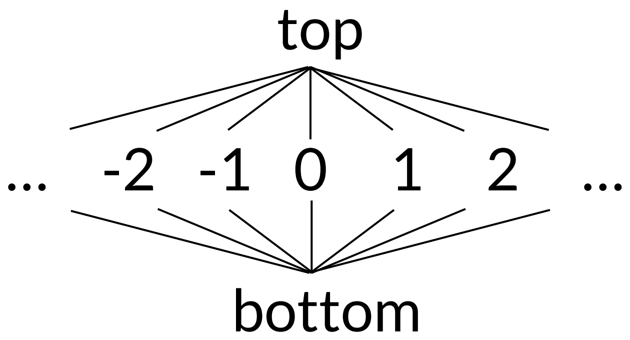
What value is printed?
How do you know?

Insight: *anything* you can figure out by reasoning through the program by hand, an abstract interpretation can do too!

Refinement

Consider the following program:

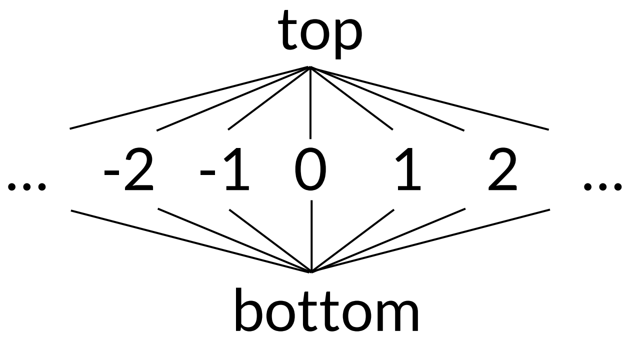
```
x = 0
while (x < 3):
  x = x + 1
print x
```



Refinement

Consider the following program:

```
x = 0
while (x < 3):
  x = x + 1
print x
```



not enough! need sets

Refinement

draw in the correct lattice here:

Consider the following program:

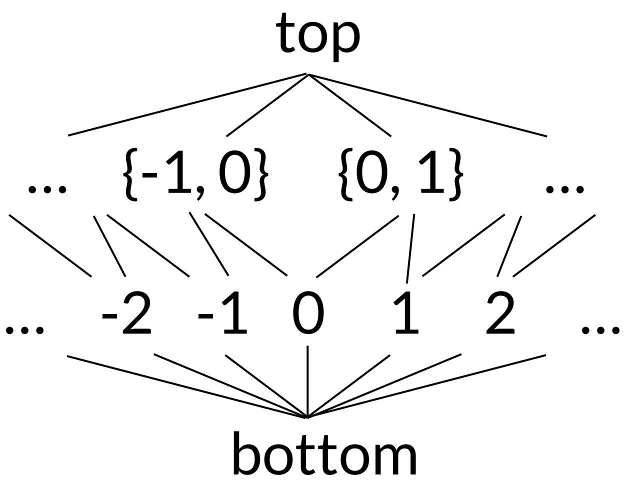
```
x = 0
while (x < 3):
  x = x + 1
print x
```

Refinement

draw in the correct
lattice here:

Consider the following program:

```
x = 0
while (x < 3):
  x = x + 1
print x
```



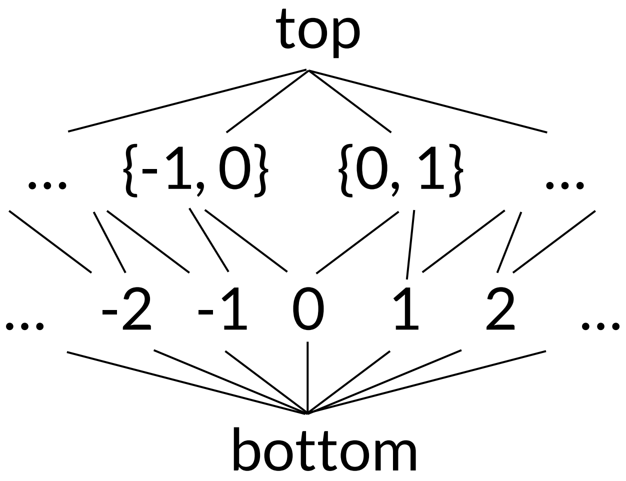
(actually need to extend this to 4 layers,
but there's not room on the slide)

Refinement

draw in the correct lattice here:

Consider the following program:

```
x = 0
while (x < 3):
  x = x + 1
print x
```



Does this permit us to prove the value of x at the end?

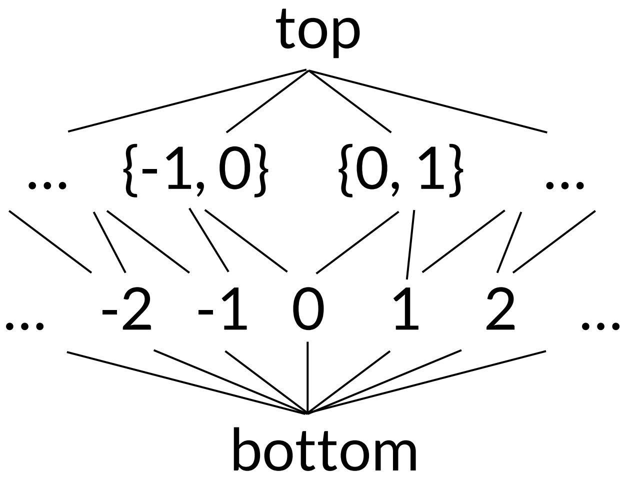
(actually need to extend this to 4 layers, but there's not room on the slide)

Refinement

draw in the correct lattice here:

Consider the following program:

```
x = 0
while (x < 3):
  x = x + 1
print x
```



Does this permit us to prove the value of x at the end?
NO (need transfer function)

(actually need to extend this to 4 layers, but there's not room on the slide)

Refinement

- We need a transfer function for **branching**

Refinement

- We need a transfer function for **branching**
  - when we exit the while loop, we know the loop guard is **false**

Refinement

- We need a transfer function for **branching**
  - when we exit the while loop, we know the loop guard is **false**
- These transfer functions are called ***refinements*** because they typically involve a greatest lower bound

Refinement

- We need a transfer function for **branching**
  - when we exit the while loop, we know the loop guard is **false**
- These transfer functions are called ***refinements*** because they typically involve a greatest lower bound
  - a refinement **rules out** some possible states

Refinement

- We need a transfer function for **branching**
  - when we exit the while loop, we know the loop guard is **false**
- These transfer functions are called ***refinements*** because they typically involve a greatest lower bound
  - a refinement **rules out** some possible states
- Refinements are defined over the **boolean operators** of the language

Refinement

- We need a transfer function for **branching**
    - when we exit the while loop, we know the loop guard is **false**
- These transfer functions are called ***refinements*** because they typically involve a greatest lower bound
    - a refinement **rules out** some possible states
- Refinements are defined over the **boolean operators** of the language
    - for our example, we need a refinement for **>=**

Refinement

- We need a transfer function for **branching**
  - when we exit the while loop, we know the loop guard is **false**
- These transfer functions are called ***refinements*** because they typically involve a greatest lower bound
  - a refinement **rules out** some possible states
- Refinements are defined over the **boolean operators** of the language
  - for our example, we need a refinement for **>=**
  - why >= and not < ?

Refinement

- We need a transfer function for **branching**
  - when we exit the while loop, we know the loop guard is **false**
- These transfer functions are called ***refinements*** because they typically involve a greatest lower bound
  - a refinement **rules out** some possible states
- Refinements are defined over the **boolean operators** of the language
  - for our example, we need a refinement for **>=**
  - why >= and not < ?
    - loop guard is false, so we invert the operator

Refinement

Consider the following program:

```
x = 0
while (x < 3):
  x = x + 1
print x
```

(on the whiteboard. Start by drawing a CFG, then execute the algorithm. Put the CFG to the side and don't erase it.)

Widening

- What if we want to build a **bigger** constant propagation lattice?

Widening

- What if we want to build a **bigger** constant propagation lattice?
  - the previous example only worked because we knew that we only needed **at most 4 values** at a time

Widening

- What if we want to build a **bigger** constant propagation lattice?
  - the previous example only worked because we knew that we only needed **at most 4 values** at a time
  - in the real world, we don't know **how many values** we'll need for any given program!

Widening

- What if we want to build a **bigger** constant propagation lattice?
    - the previous example only worked because we knew that we only needed **at most 4 values** at a time
    - in the real world, we don’t know **how many values** we’ll need for any given program!
    - it would be nice if we could have sets of **arbitrary size**

Widening

- What if we want to build a **bigger** constant propagation lattice?
  - the previous example only worked because we knew that we only needed **at most 4 values** at a time
  - in the real world, we don't know **how many values** we'll need for any given program!
  - it would be nice if we could have sets of **arbitrary size**
    - and we shouldn't need to **reimplement** our analysis each time we need to reason about differently-sized sets

Widening

- What if we want to build a **bigger** constant propagation lattice?
  - the previous example only worked because we knew that we only needed **at most 4 values** at a time
  - in the real world, we don't know **how many values** we'll need for any given program!
  - it would be nice if we could have sets of **arbitrary size**
    - and we shouldn't need to **reimplement** our analysis each time we need to reason about differently-sized sets
  - do you think that's possible?

Widening

- What if we want to build a **bigger** constant propagation lattice?
  - the previous example only worked because we knew that we only needed **at most 4 values** at a time
  - in the real world, we don't know **how many values** we'll need for any given program!
  - it would be nice if we could have sets of **arbitrary size**
    - and we shouldn't need to **reimplement** our analysis each time we need to reason about differently-sized sets
  - do you think that's possible?
    - We can use ***widening operators*** to allow this (sort of)

Widening

Definition: a ***widening operator*** is a predefined policy to take a particular upper bound if the abstract value at a particular location has changed too many times

Widening

Definition: a ***widening operator*** is a predefined policy to take a particular upper bound if the abstract value at a particular location has changed too many times

- effectively, this guarantees termination by **bounding** the number of times that a particular value can change, even if the lattice is of infinite size

Widening

Definition: a ***widening operator*** is a predefined policy to take a particular upper bound if the abstract value at a particular location has changed too many times

- effectively, this guarantees termination by **bounding** the number of times that a particular value can change, even if the lattice is of infinite size
- this is safe because the analysis isn't required to take **the least** upper bound so long as it chooses **an** upper bound

Widening

Definition: a ***widening operator*** is a predefined policy to take a particular upper bound if the abstract value at a particular location has changed too many times

- effectively, this guarantees termination by **bounding** the number of times that a particular value can change, even if the lattice is of infinite size
- this is safe because the analysis isn't required to take **the least** upper bound so long as it chooses **an** upper bound
- example widening operator for constant propagation:
  - if an abstract value has changed at least five times, go to top

Widening

Let’s return to the previous example:

```
x = 0
while (x < 3):
  x = x + 1
print x
```

Widening

Let's return to the previous example:

```
x = 0
while (x < 3 10):
  x = x + 1
print x
```

Widening

- The main advantage of widening is that it permits lattices with **infinite height**

Widening

- The main advantage of widening is that it permits lattices with **infinite height**
- The downside is that it introduces additional **imprecision**
  - but abstract interpretation was always imprecise, so that’s okay

Widening

- The main advantage of widening is that it permits lattices with **infinite height**
- The downside is that it introduces additional **imprecision**
  - but abstract interpretation was always imprecise, so that's okay
- A nice fact about implementing an abstract interpretation is that it is **always safe** to apply a widening operator

Widening

- The main advantage of widening is that it permits lattices with **infinite height**
- The downside is that it introduces additional **imprecision**
    - but abstract interpretation was always imprecise, so that's okay
- A nice fact about implementing an abstract interpretation is that it is **always safe** to apply a widening operator
    - this means it's easy to support complex language features: just immediately widen any values that they impact
        - "go to top" is a sound policy in all situations

Agenda: abstract interpretation, part 2

- review and clarifications from last week
- more on soundness
- refinement and branching
- widening
- **Stein's algorithm example**
- analysis implementation demo

Another example: Stein's algorithm

```
def gcd(int a, int b):
  if a == 0 or b == 0:
    return 0
  int expt = 0
  while a is even and b is even:
    a = a / 2
    b = b / 2
    expt = expt + 1
  while a != b:
    if a is even: a = a / 2
    elif b is even: b = b / 2
    elif a > b: a = (a - b) / 2
    else: b = (b - a) / 2
  return a * 2^expt
```

Another example: Stein's algorithm

```
def gcd(int a, int b):
  if a == 0 or b == 0:
    return 0
  int expt = 0
  while a is even and b is even:
    a = a / 2
    b = b / 2
    expt = expt + 1
  while a != b:
    if a is even: a = a / 2
    elif b is even: b = b / 2
    elif a > b: a = (a - b) / 2
    else: b = (b - a) / 2
  return a * 2^expt
```

First question: does this program ever **divide by zero**? Take a moment and discuss.

Another example: Stein's algorithm

```
def gcd(int a, int b):
  if a == 0 or b == 0:
    return 0
  int expt = 0
  while a is even and b is even:
    a = a / 2
    b = b / 2
    expt = expt + 1
  while a != b:
    if a is even: a = a / 2
    elif b is even: b = b / 2
    elif a > b: a = (a - b) / 2
    else: b = (b - a) / 2
  return a * 2^expt
```

First question: does this program ever **divide by zero**? Take a moment and discuss.

Answer: **definitely not**!

Another example: Stein's algorithm

```
def gcd(int a, int b):
  if a == 0 or b == 0:
    return 0
  int expt = 0
  while a is even and b is even:
    a = a / 2
    b = b / 2
    expt = expt + 1
  while a != b:
    if a is even: a = a / 2
    elif b is even: b = b / 2
    elif a > b: a = (a - b) / 2
    else: b = (b - a) / 2
  return a * 2^expt
```

First question: does this program ever **divide by zero**? Take a moment and discuss.

Answer: **definitely not**!

- all divisions are by 2
  - 2 != 0

Another example: Stein's algorithm

```
def gcd(int a, int b):
  if a == 0 or b == 0:
    return 0
  int expt = 0
  while a is even and b is even:
    a = a / 2
    b = b / 2
    expt = expt + 1
  while a != b:
    if a is even: a = a / 2
    elif b is even: b = b / 2
    elif a > b: a = (a - b) / 2
    else: b = (b - a) / 2
  return a * 2^expt
```

First question: does this program ever **divide by zero**? Take a moment and discuss.

Answer: **definitely not**!

- all divisions are by 2
  - 2 != 0
- "constant propagation" can prove no divisions by zero!

Another example: Stein's algorithm

```
def gcd(int a, int b):
  if a == 0 or b == 0:
    return 0
  int expt = 0
  while a is even and b is even:
    a = a / 2
    b = b / 2
    expt = expt + 1
  while a != b:
    if a is even: a = a / 2
    elif b is even: b = b / 2
    elif a > b: a = (a - b) / 2
    else: b = (b - a) / 2
  return a * 2^expt
```

Next question: does this program **terminate on all inputs**? Take a moment and discuss. (Hint: draw a CFG.)

Another example: Stein's algorithm

```
def gcd(int a, int b):
  if a == 0 or b == 0:
    return 0
  int expt = 0
  while a is even and b is even:
    a = a / 2
    b = b / 2
    expt = expt + 1
  while a != b:
    if a is even: a = a / 2
    elif b is even: b = b / 2
    elif a > b: a = (a - b) / 2
    else: b = (b - a) / 2
  return a * 2^expt
```

Next question: does this program **terminate on all inputs**? Take a moment and discuss. (Hint: draw a CFG.)

To prove termination, we need to show that both while loop guards are **eventually false**.

Another example: Stein's algorithm

```
def gcd(int a, int b):
  if a == 0 or b == 0:
    return 0
  int expt = 0
  while a is even and b is even:
    a = a / 2
    b = b / 2
    expt = expt + 1
  while a != b:
    if a is even: a = a / 2
    elif b is even: b = b / 2
    elif a > b: a = (a - b) / 2
    else: b = (b - a) / 2
  return a * 2^expt
```

Next question: does this program **terminate on all inputs**? Take a moment and discuss. (Hint: draw a CFG.)

To prove termination, we need to show that both while loop guards are **eventually false**.

- 1st: a is odd or b is odd

Another example: Stein's algorithm

```
def gcd(int a, int b):
  if a == 0 or b == 0:
    return 0
  int expt = 0
  while a is even and b is even:
    a = a / 2
    b = b / 2
    expt = expt + 1
  while a != b:
    if a is even: a = a / 2
    elif b is even: b = b / 2
    elif a > b: a = (a - b) / 2
    else: b = (b - a) / 2
  return a * 2^expt
```

Next question: does this program **terminate on all inputs**? Take a moment and discuss. (Hint: draw a CFG.)

To prove termination, we need to show that both while loop guards are **eventually false**.

- 1st: a is odd or b is odd
- 2nd: a eventually equals b

Another example: Stein's algorithm: parity

```
def gcd(int a, int b):
  if a == 0 or b == 0:
    return 0
  int expt = 0
  while a is even and b is even:
    a = a / 2
    b = b / 2
    expt = expt + 1
  while a != b:
    if a is even: a = a / 2
    elif b is even: b = b / 2
    elif a > b: a = (a - b) / 2
    else: b = (b - a) / 2
  return a * 2^expt
```

Fortunately, we already know an analysis for parity. Let's use it (on the board; requires a CFG).

Another example: Stein's algorithm: parity

```
def gcd(int a, int b):
  if a == 0 or b == 0:
    return 0
  int expt = 0
  while a is even and b is even:
    a = a / 2
    b = b / 2
    expt = expt + 1
  while a != b:
    if a is even: a = a / 2
    elif b is even: b = b / 2
    elif a > b: a = (a - b) / 2
    else: b = (b - a) / 2
  return a * 2^expt
```

Fortunately, we already know an analysis for parity. Let's use it (on the board; requires a CFG).

- we ran into a problem: we **can't prove** that a and b are eventually odd!
  - the transfer function for even / is2 returns T

Another example: Stein's algorithm: parity

```
def gcd(int a, int b):
  if a == 0 or b == 0:
    return 0
  int expt = 0
  while a is even and b is even:
    a = a / 2
    b = b / 2
    expt = expt + 1
  while a != b:
    if a is even: a = a / 2
    elif b is even: b = b / 2
    elif a > b: a = (a - b) / 2
    else: b = (b - a) / 2
  return a * 2^expt
```

Fortunately, we already know an analysis for parity. Let's use it (on the board; requires a CFG).

- we ran into a problem: we **can't prove** that a and b are eventually odd!
  - the transfer function for even / is2 returns T
- in this case, that's actually correct!
  - the program does not terminate on all inputs
  - -1, 1 is a counterexample

Agenda: abstract interpretation, part 2

- review and clarifications from last week
- more on soundness
- refinement and branching
- widening
- Stein’s algorithm example
- **analysis implementation demo**

Course announcements

- This week's homework is **individual** (you may not work with a partner)
  - this is a difference from previous homeworks!
- early next week I will send out a survey (via Discord) about what topic we should cover in the last week of class (April 25)
  - please give this some serious thought!
  - the survey will be open until next week's class, and I will announce the result during class