Abstract Interpretation

Static Analysis section, **Lecture 19**

Pavel Panchekha

CS 6110, U of Utah

19 March 2020

Class Progress

Logical reasoning

Program logics

Static analysis

Online lecture for the rest of the term

Will try to continue as much as possible without change

Attendance cancelled: not taken, everyone marked present

Assignment 5 **cancelled**. Final presentation **virtual**.

Record final presentation; watch others' videos

Write and respond to written questions on presentation

Video Lecture

Download **Zoom app**, join this meeting at **normal time**:

232-421-488

Hold lectures, do exercises, Q&A as normal:

Stay muted; hold space bar to temporarily unmute

Raise hand to ask a question; I'll call on you } **Participants** panel

Technical support over chat with Manasij

Lectures **recorded**, re-watch / stream on Youtube

Project

Due dates **not changing**, project still on

Milestone II in class; final presentation recorded-only

- Prerecord as backup in case of connection issues
- Live Q&A in class for the Milestone II presentations
- Written Q&A for the final presentations (10% of grade)

Collaborate online for group projects

- Schedule a **1 hour weekly meeting** or more
- “Pair programming” for presentations works well

Schedule

Absences, extensions granted freely

- Still expecting achievement of most or all project goals
- Willing to do "**incomplete**"s for final projects, though

Some **reorganization of the schedule** expected

- Since final project presentations online, extra time

Your health is more important than class

- Please request absences, extensions in case of illness
- Child care, elderly care, and sick care all good reasons

A Note

No one knows what is going on or how to respond

- The administration is giving contradictory advice
- New guidance comes out daily

Currently, voting to make **grading pass/fail**

- May or may not pass; don’t get your hopes up

I’ll do what I can to make things right

- Please be flexible with me, but I’ll return the favor
- Don’t expect much coordination from the U as a whole

Another Note

My planning has been totally thrown off

- Let’s salvage what we can out of the semester
- Expect less prepared, less-polished lectures

Happily, **most essential content done**

- If you’ve learned how Dafny works and how to use it, I’m happy
- Learned **one** method of verification top-to-bottom

Refresher

Long spring break, huh?

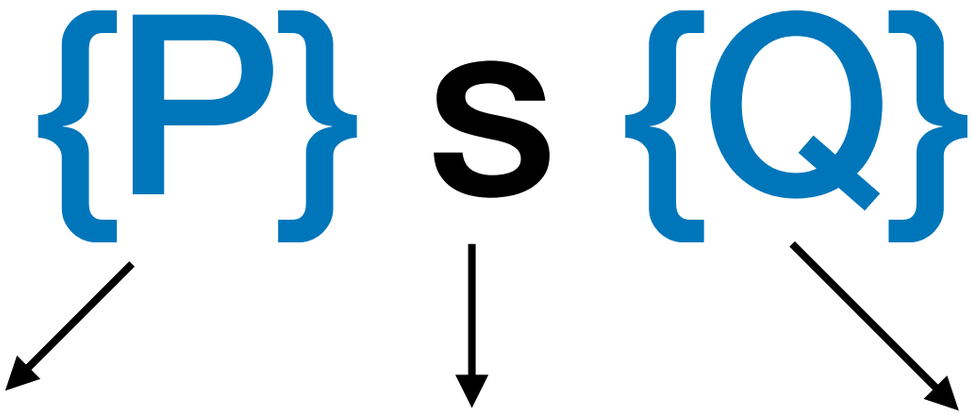
Verification

Consider verifying a **binary search**:

```
def bsearch(l, x):                          { sorted(l) }
   i, j = 0, len(l) - 1
   if j == -1 or x < l[i] or l[j] < x:
      return -1
   while j - i > 1:
      m = (i + j) / 2
      if l[m] < x: i = m
      elif l[m] > x: j = m
      else: return m
   if l[i] == x: return i
   elif l[j] == x: return j
   else: return -1
                                            { l[return] == x or return == -1 }
```

What We Learned

Add **pre-/post-conditions** to specify behavior



If **P** true of a state, and **s** executed, then **Q** true after

Each **{P} s {Q}** is a **logical statement**

Weakest preconditions systematically generate that statement

Verification

Consider verifying a **binary search**:

```
def bsearch(l, x):                          { sorted(l) }
   i, j = 0, len(l) - 1
   if j == -1 or x < l[i] or l[j] < x:
      return -1                             { 0 <= i <= j < len(l) }
while j - i > 1:                            { l[i] <= x <= l[j] }
   m = (i + j) / 2                          { DECREASES j - I }
   if l[m] < x: i = m
   elif l[m] > x: j = m
   else: return m                           { (l[i] == x and Q) or … }
if l[i] == x: return i
elif l[j] == x: return j
else: return -1                             { l[return] == x or return == -1 }
```

Statement Types

Loops as a form of **infinite statement**

- **Invariants** a short-hand for verifying that statement
- **Measures** for proving a loop terminates

Functions for **modular bits of code**

- **Reusing** function pre-/post-conditions at call sites
- **Measures** for proving recursive functions terminate

Verification

Consider verifying a **binary search**:

```
def bsearch(l, x):                          { sorted(l) }
   i, j = 0, len(l) - 1
   if j == -1 or x < l[i] or l[j] < x:
      return -1                               { 0 <= i <= j < len(l) }
while j - i > 1:                              { l[i] <= x <= l[j] }
   m = (i + j) / 2                            { DECREASES j - I }
   if l[m] < x: i = m
   elif l[m] > x: j = m
   else: return m                           { (l[i] == x and Q) or … }
if l[i] == x: return i
elif l[j] == x: return j
else: return -1                             { l[return] == x or return == -1 }
```

Verification

Reasoning about the verification condition

```
[redacted] ->
([redacted] or [redacted] or [redacted])
 and ([redacted] or [redacted])) or
(not ([redacted] or [redacted] or [redacted])
 and ([redacted] and [redacted])
 and (([redacted] and ([redacted] and [redacted] -> …)
```

Boolean structure uses DPLL(T) algorithm

Put into **conjunctive form**: AND of ORs

Guess boolean value, simplify and infer new ones

Result is a **query** containing true theory terms

Verification

Reasoning about the verification condition

```
[redacted] ->
([redacted] or [redacted] or [redacted])
 and ([redacted] or [redacted])) or
(not ([redacted] or [redacted] or [redacted])
 and ([redacted] and [redacted])
 and (([redacted] and ([redacted] and [redacted] -> …)
```

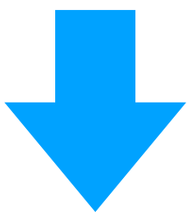
```
sorted(l) and ~X and ~Y and (X or ~A or ~B) and (~X or A) and …
```

```
sorted(l) and l[0] < x and 0 < len(l) and …
```

Verification

Reasoning about the query

```
sorted(l) and l[0] < x and 0 < len(l) and …
```



```
sorted(l) and l0 < x and l0 = l[0] and 0 < lenl and …
```

Integers

```
l0 < x and 0 < lenl and …
```

Arrays

```
sorted(l) and l0 = l[0] and …
```

Split the query into domain-specific queries

Verification

Reasoning about the domain-specific query

```
l0 < x and 0 < lenl and l0 >= x and …
```

Variable elimination for integer reasoning

Pick a variable x to eliminate

Group equations by x on the left or right

Form all pairs of equations, eliminating x

```
l0 <= l0 - 1
```

Staying Safe

Don’t get Coronavirus

Avoiding Illness

Keep a safe distance (6 feet) from other people

- Stay away from crowds; work from home if possible
- Shop, buy groceries less often (buy in bulk)
- Avoid large social outings (including for your children)

Keep clean; wash hands, wear gloves

- Virus spreads by **contact** (hand-shakes) or **coughing**
- Wash hands for **20 seconds** using soap
- Wear a mask or **cover your coughs** if you are sick

Symptoms

Mild cases (80%) have the usual flu symptoms

- Fever / high temperature
- Coughing, usually a dry cough
- Shortness of breath and general tiredness

Severe cases (20%): emergency medical attention (911)

- Difficulty breathing; bluish lips or face
- Pain or pressure in the chest
- Confusion and listlessness

Medical Care

Call your doctor if you think you have it

- You've recently travelled to Europe, China, Korea, or Iran
- You've been in contact with someone sick
- **Call before getting care** to protect others

Mild cases usually pass without issue

- Drink lots of liquids, don't skip meals, get rest
- Track and record your temperature regularly
- Isolate yourself for 3 days after the fever passes

More Information

coronavirus.gov

Scaling Up

What makes the process slow?

Abstract Interpretation

New goal of **scalability** for analyses

Approach: simple but specific over slow but general

Simple universe of predicates on program states

Loops analyzed by repeatedly re-analyzing

Lattice structure characterizes each analysis

Data structure for predicates, “or” method, finite height

Scalability

Process described so far can be **very slow**:

- Weakest preconditions **duplicate code**: exponential in code
- SAT solving is **exponential in formula**
- Result is **doubly-exponential**!

Process described so far can be **very general**:

- Limited only by **solver capabilities**
- Can describe **most correctness** and performance specs
- Writing specs is its own **time-consuming** task!

Alternative

Replace **slow, general** specs with **fast, specialized** ones

Array bounds

Nullness

Format validation

Invalid states

String escaping

Non-negativity

Replace **slow, general** specs with **fast, specialized** ones

Array bounds

Index non-negative

Index less than length

Non-negativity follows **simple rules**

Simplified conditions

Conditions: which variables **known to be non-negative**

Need to know: **which expressions** produce non-negative results

$$(\geq 0) + (\geq 0) \rightsquigarrow (\geq 0) \qquad (\geq 0) \times (\geq 0) \rightsquigarrow (\geq 0) \qquad (\geq 0) \div (\geq 0) \rightsquigarrow (\geq 0)$$

```
{ i ≥ 0 && j ≥ 0 }
m = (i + j) / 2
{ i ≥ 0 && j ≥ 0 && m ≥ 0 }
l[m]  ⟵ valid
```

Simple rules instead of solver queries: fast

Propagating Conditions

Simplified conditions **too general**; over-approximate

Make sure **rules for propagating** are also over-approximate

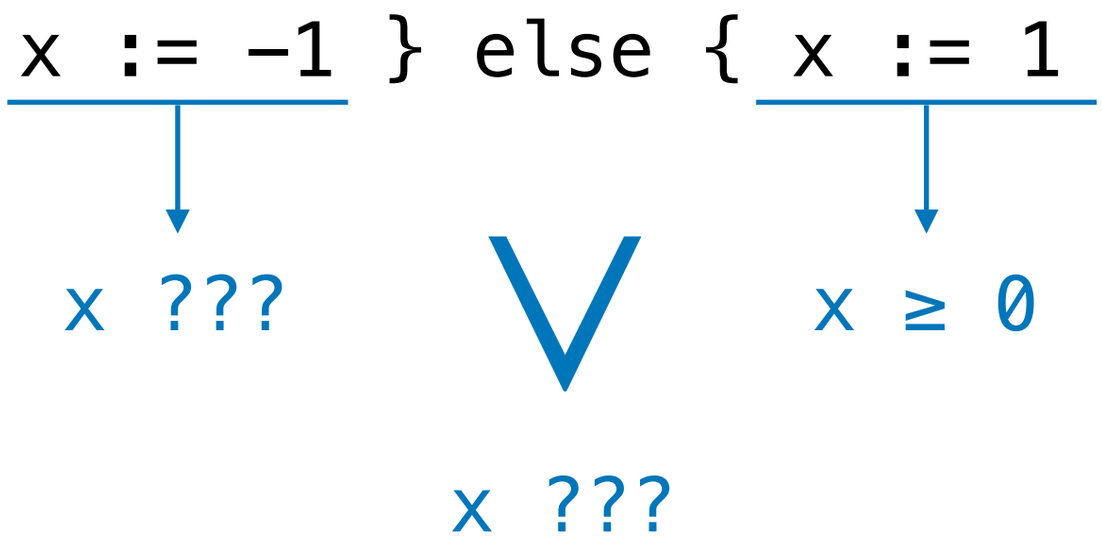
$$\{ P \}\ \texttt{skip}\ \{ P \}$$

$$\{ P \}\ s\ \{ Q \} \land \{ Q \}\ t\ \{ R \} \rightarrow \{ P \}\ s; t\ \{ R \}$$

$$e \geq 0 \rightarrow \{ P(x) \}\ x := e\ \{ x \geq 0 \}$$

Propagating Conditions

If statements require **“or”** for simplified conditions

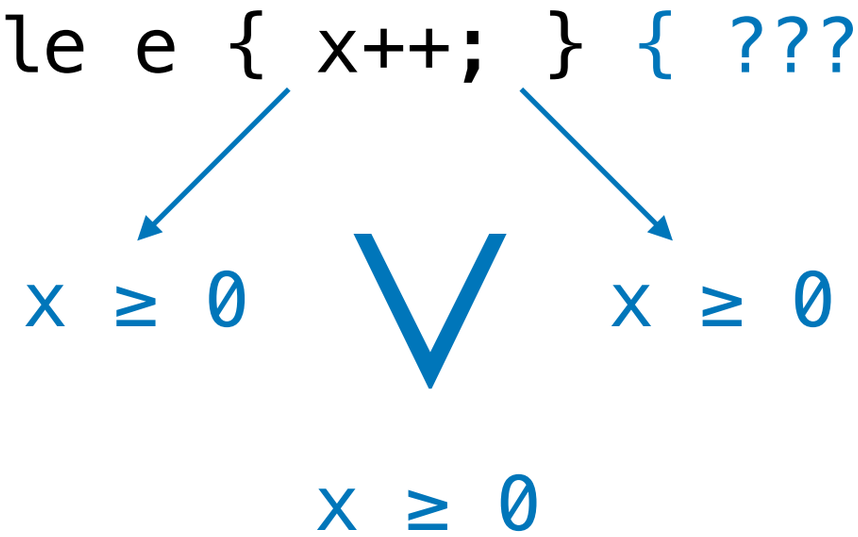


The conditional could also be over-approximated

Called “**flow-sensitive**” analysis; ignored here

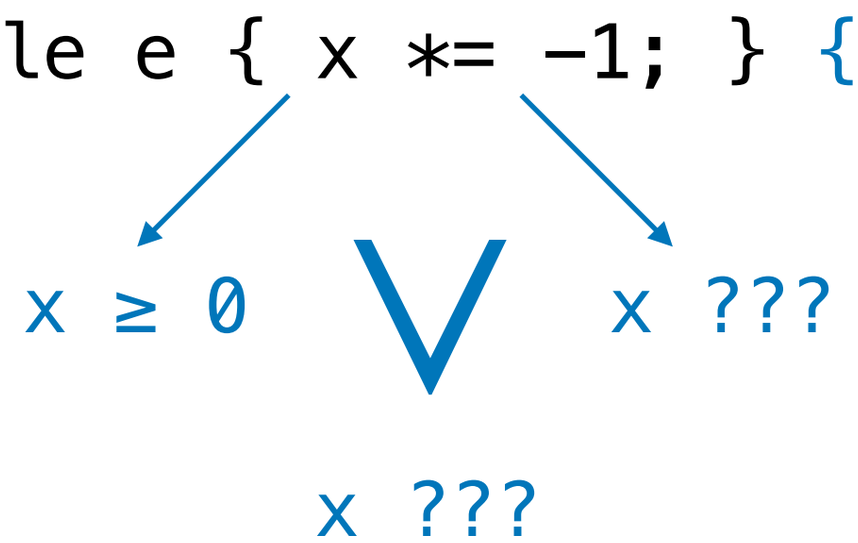
Loops

If statements require **“or”** for simplified conditions



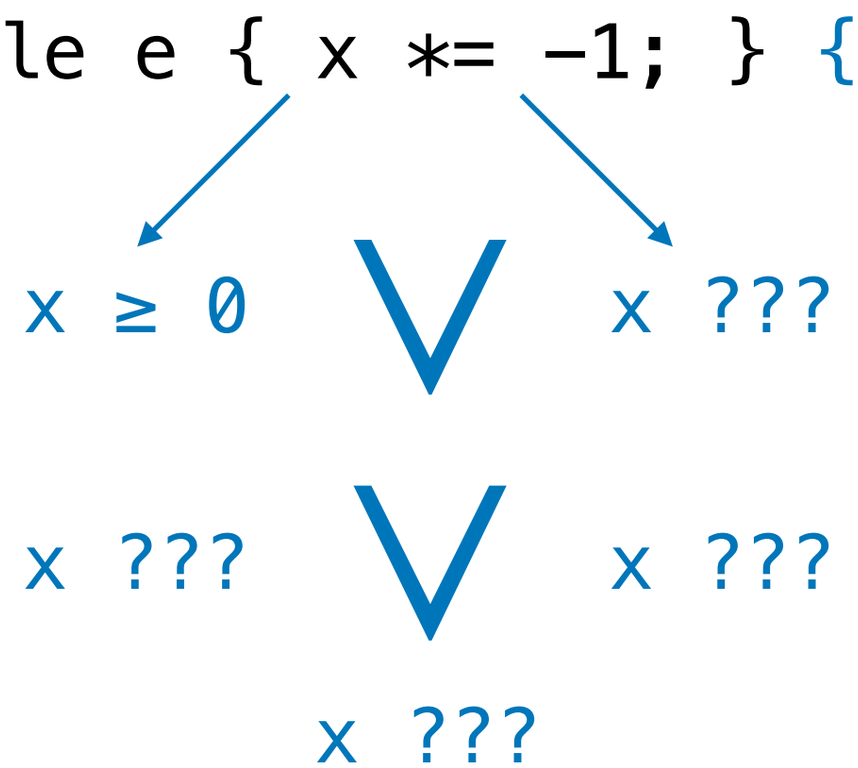
Loops

If statements require **“or”** for simplified conditions



Loops

If statements require **“or”** for simplified conditions



Repeat loop analysis until result stops changing

Why it works

Each step **over-approximates** program state

- If warning skipped, definitely no issue

However, **some specifications** cannot be stated

- Specifically restricted set of conditions

Analysis of loops **repeated** until fix-point

- Lattice must have **finite height** for analysis to terminate

Abstract Interpretation

A general approach to simplified conditionals

Generalizing

Create a **simple universe** of simplified conditions

Conditions for 1) **values**, and 2) **program states**

Non-negative

Unknown

Over-approximating **“or” operation** on conditions

Also need a least-specific “unknown” state

Example

Track **sign** of values & variables

Positive

Zero

Negative

Non-positive

Non-negative

Non-zero

Unknown

OR	+	0	-	!+	!-	!0	?
+							
0							
-							
!+							
!-							
!0							
?							

Example

Track **sign** of values & variables

Operation rules:

```
(+) + (+) = (+)
(+) + (0) = (+)
(+) + (−) = (?)
```

…

```
(+) * (+) = (+)
(−) * (−) = (+)
```

…

Constant rules:

```
0 is (0)
1, 2, … is (+)
−1, −2, … is (−)
```

Conditional rules:

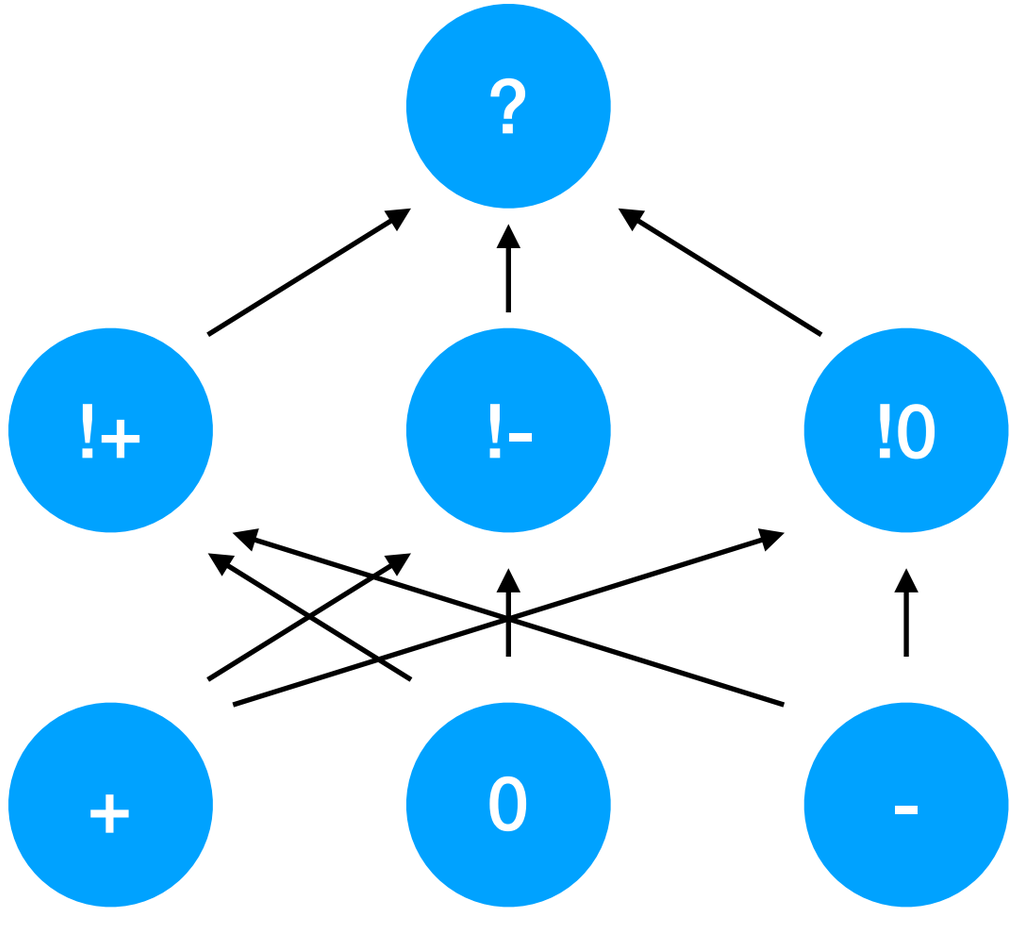
```
(+) < x → x is (+)
(−) < (+) = True
```

…

Example

OR operation produces a **graph structure**

A → B when (A OR B) = B; structure called a **lattice**



Exercise

Define an analysis for **evenness/oddness** of integers

Analysis

Represent predicates by a **data structure**

- The “or” function is a **method** on the structure

Define **rules for propagating** predicates over statements

- Also rules for assigning predicates to expressions
- Analysis of loops requires repeating until fix-point

Finally, **issue warnings** on certain operations

- Usually, check for certain predicates on function arguments

Next class:

Common Domains

To do:

- ☐ Course feedback
- ☐ Work on project

Abstract Interpretation

New goal of **scalability** for analyses

Approach: simple but specific over slow but general

Simple universe of predicates on program states

Loops analyzed by repeatedly re-analyzing

Lattice structure characterizes each analysis

Data structure for predicates, “or” method, finite height

RELATIONS

MULTIPLE

VARIABLES

INTERVALS

WIDENING

NARROWING

EXAMPLE

ARRAY

BOUNDS

Next class:

Common Domains

To do:

- ☐ Course feedback
- ☐ Work on project