CSC2/455 Software Analysis and Improvement
Program Analysis/Abstract Interpretation

Sreepathi Pai

Mar 28, 2022

URCS

Outline

And now for something (not so completely) different

Program Analysis in Industry

Basic Notions

Illustrating Abstract Interpretation

Postscript

Outline

And now for something (not so completely) different

Program Analysis in Industry

Basic Notions

Illustrating Abstract Interpretation

Postscript

So far

- Data flow analysis
  - Iterative data flow analysis
  - Region-based analysis
- Loop analysis
- Type Checking
- What next?

Compilers are not the only program analyzers

- Compilers are probably the most used program analyzers
- But are severely time constrained
- Finding *program* errors is not primary goal
  - Syntax errors, type errors
  - Code generation primary goal

Program/Software Analysis

- Software is increasingly mission-critical
- Can kill people!
  - Boeing 737 MAX(?)
  - Therac-25 (X-ray)
  - Industrial Robotics
- (less extreme?) Can lose money
  - Software crashes
  - Data loss
- Can we analyze programs for *functional* correctness?
  - Topic of the next few lectures

Outline

And now for something (not so completely) different

Program Analysis in Industry

Basic Notions

Illustrating Abstract Interpretation

Postscript

SLAM (Microsoft, early 2000s)

- MS isolated most crashes to buggy drivers
- Static Driver Verifier project
  - Would verify driver code (in C) for correctness
- Used *model checking*
  - Models programs as finite-state machines
  - I used a similar tool (CBMC) to check your assignments

Infer (Facebook)

- Checks C, C++, Objective C, Java and Android code
- Used for checking Facebook's mobile apps
- Open source, `https://fbinfer.com/`
  - Used by Amazon, Mozilla, Uber and Facebook and its affiliates, JD.com, etc.
- Comes with its own language AL (OCaml-derivative?) to describe analyses
  - Analyzes programs in SIL ("Smallfoot Intermediate Language")
- Uses abstract interpretation + *separation logic*
  - Abstract interpretation very similar to data flow analysis frameworks
- CACM Article: Scaling Static Analyses at Facebook
- Good video: Getting the most out of static analyzers

SPARTA (Facebook)

- Language-independent analyzer
  - a C++ framework
- Open source, `https://code.fb.com/open-source/sparta/`
- Used in FB's RedEx tools
  - for analyzing Android binary code (`.dex`)
- Also uses *abstract interpretation*

Other efforts

- Stanford Checker
  - commercialized by Coverity, late 2000s
  - CACM article, "A few billion lines of code later: using static analysis to find bugs in the real world"
- Google's static analysis tools
  - Checker Framework for Java programs
  - Shipshape (abandoned?) (Google Tricorder)
  - CACM article, "Lessons from Building Static Analysis Tools At Google"
- Oracle's Soufflé
  - Soufflé: Logic Defined Static Analysis

Outline

And now for something (not so completely) different

Program Analysis in Industry

Basic Notions

Illustrating Abstract Interpretation

Postscript

Limitations

- None of these frameworks and tools can escape the fact that analysis is an undecidable problem
- All compute approximations
  - Or risk ending up intractable
- Must be designed to be *sound*
  - Approximations are conservative/safe
- Leads to imprecision (i.e. *incomplete*)
  - May model behaviour not in original programs
  - (recall IDEAL vs MOP vs MFP)
  - leads to false positives

States and Transitions

- A program's state is a mapping of variables to values
- Programs move from one state to another
  - begin execution in subset of (initial) states
- Notions of state *before* a program point (i.e. a statement) and *after* a program point
  - Also sometimes known as *pre-condition* and *post-condition* respectively.
- Relation that maps before-states to after-states is called a *transition* relation (t)
  - $\langle x, y \rangle$ (x is before-state, y is after-state)

Traces

- An execution trace of a program is a sequence of states
  - $s_0 s_1 s_2 \ldots s_n$
- An execution trace may be finite or infinite
  - $s_0 s_1 s_2 \ldots$
- The collection of partial traces can actually happen (i.e. state transitions obey the transition relation) is called the *collecting semantics*
  - I.e. for all $s_i s_j$ in trace, $\langle s_i, s_j \rangle \in t$

Example

```
x = 0
while(x < 100)
   x = x + 1;
```

- states are $\mathbb{Z}$
- initial state is 0
- transition relation is $\{\langle x, x' \rangle | x < 100 \wedge x' = x + 1\}$
- is 0 1 2 3 part of the collecting semantics?
- is 0 2 4 6 part of the collecting semantics?

Patrick Cousot and Radhia Cousot, *Basic Concepts of Abstract Interpretation*

What can we do with the concrete semantics?

- By examining the (concrete) collecting semantics, we can check various "properties"
  - We'll formalize "property" later.
- Problems with using the concrete semantics:
  - The previous example had a single state in initial set $\{0\}$, this is not always true. Consider `x = randint()`
  - How do we deal with infinite loops?
  - How do we deal with alternate paths (i.e. conditionals)?
  - How do we get the concrete semantics for all programs statically?
- Implementation issues:
  - Even if we could get a concrete semantics, how large would it be?

Outline

And now for something (not so completely) different

Program Analysis in Industry

Basic Notions

Illustrating Abstract Interpretation

Postscript

A Graphical Programming Language[1]

Consider a simple language with the following constructs:

- `init(R)` where `R` is a region in 2D space
  - e.g. `init(`$\{(x, y)|(x, y) \in R\}$`)`
  - This chooses a single point in R non-deterministically
  - A program must always start with `init`
- `translate(dx, dy)` moves the point by dx in the X-direction, and dy in the Y-direction
  - e.g. `translate(1.0, 0.5)` moves the point to the right and up in the Cartesian plane
- `rotate(angle)` rotates the point by `angle` about the origin
  - e.g. `rotate(90)` will move a point on the X-axis to a point on the Y-axis

[1] This exposition is based on Chapter 2 of Rival and Yi (see Postscript)

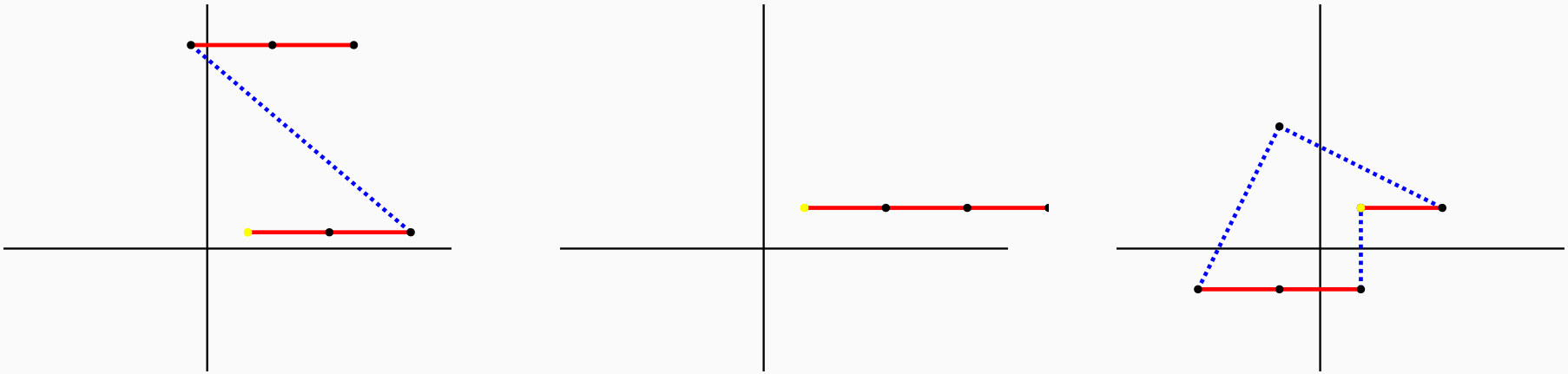
Grammar

```
P =   init(R) ';' Rest
Rest = translate(dx, dy)
       | rotate (a)
       | '{' Rest '}' or '{' Rest '}'
       | iter '{' Rest '}'
       | Rest ';' Rest
       | Empty
Empty =
```

- A program begins with `init` and is followed by statements in *Rest*
- The `or` construct is a non-deterministic choice
  - It executes the block (delimited by braces) on the right or the block on the left
  - Simulates a conditional
- The `iter` either executes the code inside the block or moves to the next statement
  - Simulates a loop
  - Note, `iter` can execute the block forever!

An Example Program

```
init([0, 1]x[0, 1]);
 translate(1, 0);
 iter {
     {
         translate(1, 0);
     } or {
         rotate(90);
     }
 }
```



Property we're interested in

Can x ever become negative?

- Can be represented by the set $x_{neg} = \{(x, y) | x < 0\}$
- Problems with concrete execution approach:
  - Infinite initial states ($R = \{(x, y) | 0 \leq x \leq 1 \wedge 0 \leq y \leq 1\}$
  - Conditionals that perform translation or rotation
  - Loop may be infinite
- But if we could obtain the set of states in all possible concrete executions, say x_{conc}, we need to show
  - $x_{conc} \cap x_{neg} = \emptyset$

Abstract Execution: Approximating Concrete Executions

- Abstract interpretation is a framework for performing program analysis
- Key ideas:
  - Abstract domain: Set of properties we're interested in
  - Abstraction function: Converts a concrete state to an element of the abstract domain
  - Transfer functions: Transforms an abstract state before a statement to an abstract state after the statement
  - Union/Join: Combines abstract states from alternate paths
  - Widen (∇): Combines abstract state across loop iterations

Abstract Domains

- For the property we're interested in, we only need the sign of x
- Potential abstract domain, *signs-x*
  - Only tracks $x < 0$, $x \in R$
  - Can only answer questions about this property, and about x
- Another abstract domain, track signs of all state variables
  - Tracks $\{x < 0, x \geq 0, x \in R\} \times \{y < 0, y \geq 0, y \in R\}$
- Abstract domain for the rest of the lecture: Intervals
  - For each state variable, track l_v and h_v such that $l_v \leq v \leq h_v$
  - For the graphical language, $l_x \leq x \leq h_x$, $l_y \leq y \leq h_y$
  - Thus, our abstraction approximates states using rectangles
  - The sides of the rectangle are parallel to the axes
  - l_v and h_v can be $-\infty$ and ∞ respectively to represent unbounded "rectangles"

Abstract Execution Using Intervals: `init`

```
init([0, 1]x[0, 1])
```

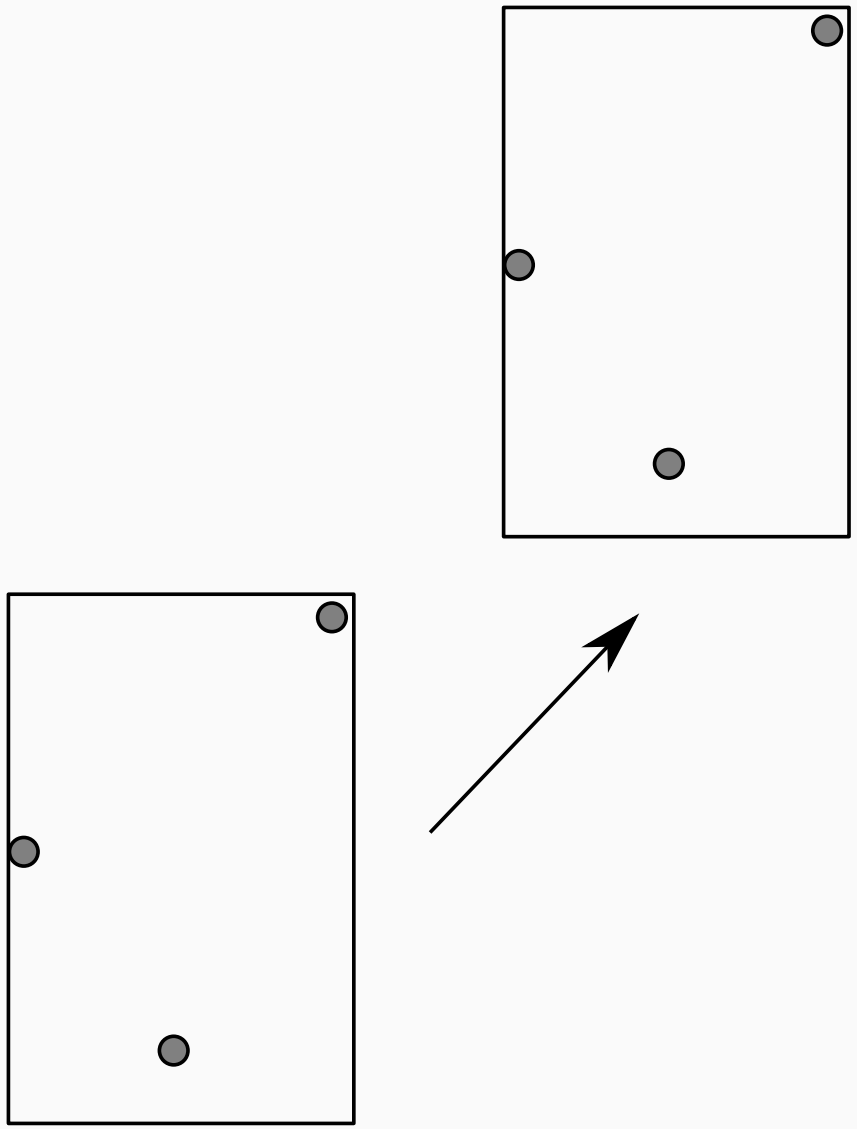
Concrete execution will give us a point in that rectangular region.

- Our abstract state after `init` will be:
  - $l_x = 0$
  - $h_x = 1$
  - $l_y = 0$
  - $h_y = 1$
- The transfer function for `init` computes the rectangle that covers the region
- In this case, the abstraction is precise

Abstract Execution Using Intervals: `translate`

```
translate(1.0, 0.5)
```

- Our abstract state after `translate` will be:
  - $l_x = l_x + 1.0 = 0.5$
  - $h_x = h_x + 1.0 = 2$
  - $l_y = l_y + 0.5 = 0.5$
  - $h_y = h_y + 0.5 = 1.5$
- The transfer function for `translate` shifts the current abstract state
  - The resulting abstract state is still precise



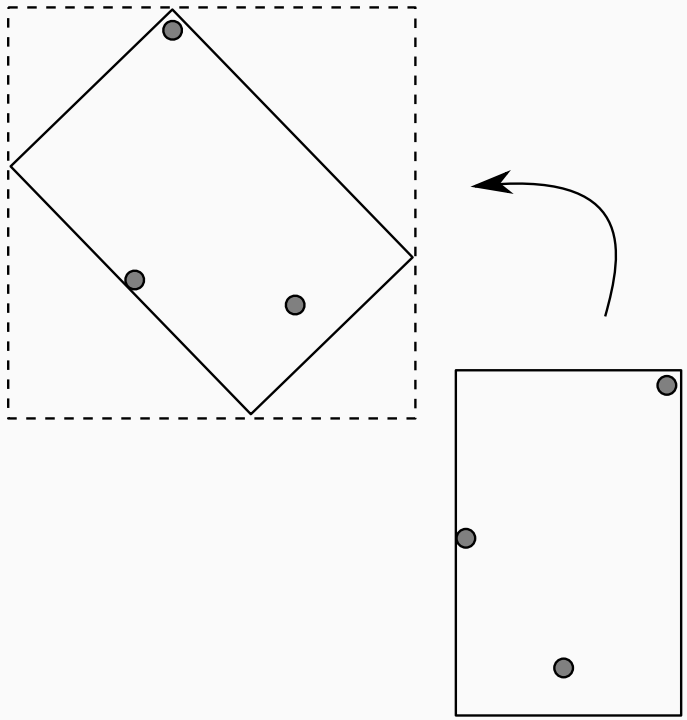
Abstract Execution Using Intervals: `rotate`

```
rotate(45)
```

- Our rectangle after `rotate` will be:
  - $l_{x_1}, l_{y_1} = rotate((l_x, l_y), 45)$
  - $h_{x_1}, l_{y_2} = rotate((h_x, l_y), 45)$
  - $l_{x_2}, h_{y_1} = rotate((l_x, h_y), 45)$
  - $h_{x_2}, h_{y_2} = rotate((h_x, h_y), 45)$
- The transfer function for `rotate` rotates the corners of the rectangle
  - The result is still a rectangle, but cannot always be represented using intervals
  - So it is not an abstract state

Finding a new interval in the abstract domain

- Let c' be a co-ordinate after `rotate` computed as in the previous slide, then
  - $l_x = min(l_{x_1}, l_{x_2}, h_{x_1}, h_{x_2})$
  - $h_x = max(l_{x_1}, l_{x_2}, h_{x_1}, h_{x_2})$
  - $l_y = min(l_{y_1}, l_{y_2}, h_{y_1}, h_{y_2})$
  - $h_y = max(l_{y_1}, l_{y_2}, h_{y_1}, h_{y_2})$
- Obviously, in general, this new interval contains more states than the rotated rectangle
  - We have lost precision
- But, the new interval/rectangle we have calculated is the "best fit"



A note on domains

- Intervals (and signs) are non-relational domains
  - They can't capture relations between x and y
  - e.g., a property that $x > y$
- Intervals also can't capture complex regions
- A more complicated abstract domain: *convex polyhedra*
  - A list of linear inequalities
  - Region is the feasible region (i.e. points that satisfy all the inequalities)
  - Convex polyhedra support relational properties, e.g. $x - y < 2$
- During program analysis you must choose the domain that most efficiently captures the property of interest

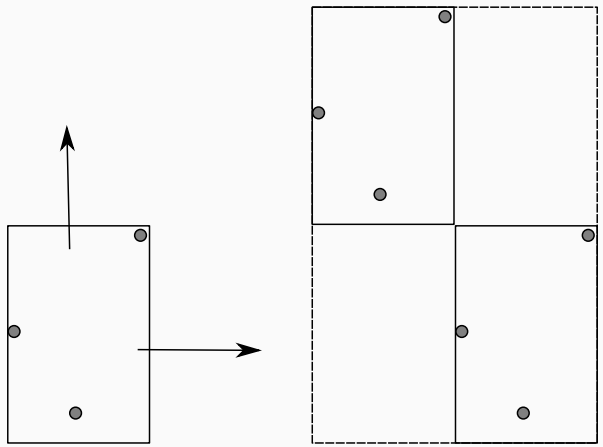
Handling Compound Statements

- Let the transfer function be called `analysis(stmt, pre-condition)`
- So far, we've defined:
  - `analysis(init, I)` (where *I* is the state on entry)
  - `analysis(translate, a)` where *a* is the state before translate
  - `analysis(rotate, a)` where *a* is the state before rotate
- Let's now define `analysis(p1 ; p2, a)`
  - The analysis of two statements `p1` and `p2`, with pre-condition `a`
  - Note that pre-condition for p2 is the post-condition of p1, so
  - `analysis(p1 ; p2, a)` is then `analysis(p2, analysis(p1, a))`

Handling conditionals: Union (Join)

```
{translate(1.0, 0)} or {translate(0, 1.0)}
```

- Program could execute either left part or right part
- Two possible abstract states, disjoint
- Resulting abstract state must incorporate both
  - *Union* the two abstract states
- `analysis(p1 or p2, a)` is:
  - `analysis(p1, a)` $\cup$ `analysis(p2, a)`
- The resulting abstract state is also, in general, imprecise



Union for Intervals

- For $a_1 \cup a_2$
  - $l_x = min(l_{x_1}, l_{x_2})$
  - $h_x = max(h_{x_1}, h_{x_2})$
  - $l_y = min(l_{y_1}, l_{y_2})$
  - $h_y = max(h_{y_1}, h_{y_2})$

Handling Loops

```
iter {b}
```

can be written as:

```
{}                                  # loop executes 0 times
{} or {b}                           # loop executes 1 time
{} or {b} or {b;b}                  # loop executes 2 times
{} or {b} or {b;b} or {b;b;b} # loop executes 3 times
 ...
```

This can be written (recursively) as: $p_{k+1} = p_k \text{or} \{p_k; \text{b}\}$, where:

- p_0 is $\{\}$
- p_1 is $\{\}$ `or` $\{\text{b}\}$
- p_2 is $\{\}$ `or` $\{\text{b}\}$ `or` $\{\text{b; b}\}$ and so on...

Iterating

```
analysis(iter {b}, a)
```

can be defined as an interative algorithm:

```
R = a
do
   T = R
   R = union(R, analysis(b, R)) # analysis for or
while R != T
```

Will this always terminate?

Example

```
init([0, 1] x [0, 1])
translate(1, 0)
iter { translate(0, 1) }
```

- The analysis of this code will not terminate!
- Abstract state before `iter`
  - $1 \leq x \leq 2,\ 0 \leq y \leq 1$
- After first union:
  - $1 \leq x \leq 2,\ 0 \leq y \leq 1 \cup 1 \leq x \leq 2,\ 1 \leq y \leq 2$
  - Result: $1 \leq x \leq 2,\ 0 \leq y \leq 2$
- After second union: $1 \leq x \leq 2,\ 0 \leq y \leq 3$
- And so on ...
  - the h_y bound keeps increasing without bound

The Widen operator

- We note that the interval $0 \leq y \leq \infty$ would overapproximate $0 \leq y \leq n$ where n is not ∞
- If we obtained this interval in our abstract state, we could terminate because h_y would "stop increasing"
  - $0 \leq y \leq \infty$ already includes all possible abstract states of the form $0 \leq y \leq n$
  - Union would no longer return a different result ensuring termination
- So widen (∇) is an operator that overapproximates unions
  - Its primary purpose is to ensure convergence

Analyzing loops using widen

```
R = a
do
   T = R
   R = widen(R, analysis(b, R))
until inclusion(R, T)  # i.e. R is included in T
return T
```

Widen for Intervals

```
def widen(a, b):
   out = union(a, b)

   if a.lx != b.lx:
      out.lx = -inf

   if a.hx != b.hx:
      out.hx = inf

   if a.ly != b.ly:
      out.ly = -inf

   if a.hy != b.hy:
      out.hy = inf
```

Putting it all together

- Each time `analysis` is applied, we obtain an abstract state a
- If a overlaps with x_{neg}, then we have detected a violation of our property

Outline

And now for something (not so completely) different

Program Analysis in Industry

Basic Notions

Illustrating Abstract Interpretation

Postscript

References

- This lecture follows the exposition in Chapter 2 of the book "An Introduction to Static Analysis: An Abstract Interpretation Perspective" by Xavier Rival and Kwangkeun Yi, MIT Press, 2020
  - The library has print copies
  - Suggest buying this book – it is self-contained and reasonably priced
- Prof. Cousot has a number of tutorials, I'll post links to them in the next lecture
  - Prof. Patrick Cousot and Prof. Radhia Cousot invented abstract interpretation