Systems and Internet
Infrastructure Security

Network and Security Research Center
Department of Computer Science and Engineering
Pennsylvania State University, University Park PA

Static Analysis

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University

September 12, 2011

Outline

- Static Analysis Goals
- Static Analysis Concepts
- Abstract Interpretation
- Interprocedural Dataflow Analysis

Our Goal

- In this course, we want to develop techniques to detect vulnerabilities and fix them automatically
- What's a vulnerability?
- How to fix them?

- *Today we will start to develop some of the techniques that we will use*

Vulnerability

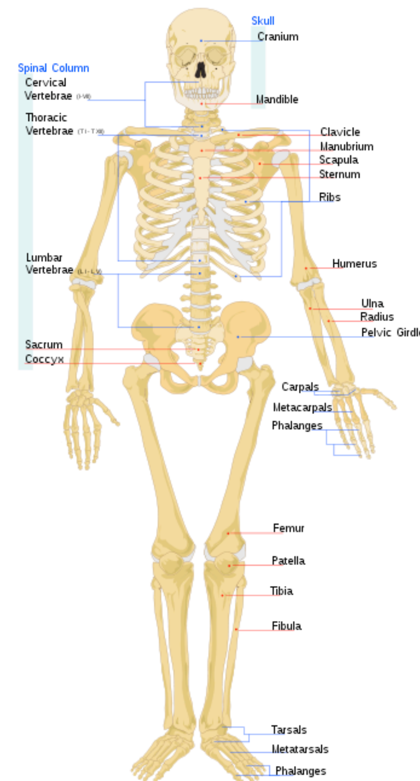
- How do you define computer ‘vulnerability’?
  - *Flaw*
  - *Accessible to adversary*
  - *Adversary has ability to exploit*

Vulnerability

- How do you define computer 'vulnerability'?
  - *Flaw – Can we find flaws in source code?*
  - *Accessible to adversary – Can we find what is accessible?*
  - *Adversary has ability to exploit – Can we find how to exploit?*

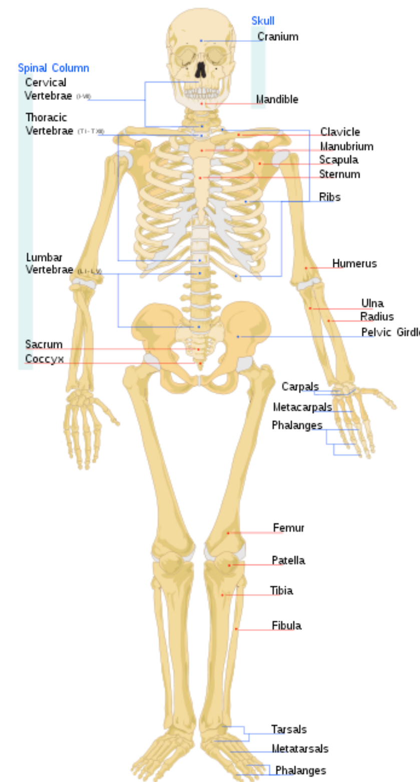
Anatomy of Control Flow Attacks

- Two steps
- First, the attacker changes the control flow of the program
  - In buffer overflow, overwrite the return address on the stack
  - What are the ways that this can be done?
- Second, the attacker uses this change to run code of their choice
  - In buffer overflow, inject code on stack
  - What are the ways that this can be done?



Anatomy of Control Flow Attacks

- Two steps
- First, the attacker changes the control flow of the program
  - In buffer overflow, overwrite the return address on the stack
  - *How can an adversary change control?*
- Second, the attacker uses this change to run code of their choice
  - In buffer overflow, inject code on stack
  - *How can we prevent this? ROP conclusions*



Static Analysis

- Explore all possible executions of a program
  - All possible inputs
  - All possible states

A Form of Testing

- Static analysis is an alternative to runtime testing
- Runtime
  - Select concrete inputs
  - Obtain a sequence of states given those inputs
  - Apply many concrete inputs (i.e., run many tests)
- Static
  - Select abstract inputs with common properties
  - Obtain sets of states created by executing abstract inputs
  - One run

Static Analysis

- Provides an approximation of behavior
- “Run in the aggregate”
  - Rather than executing on ordinary states
  - Finite-sized descriptors representing a collection of states
- “Run in non-standard way”
  - Run in fragments
  - Stitch them together to cover all paths
- Runtime testing is inherently incomplete, but static analysis can cover all paths

Static Analysis

- Provides an approximation of behavior
- "Run in the aggregate"
  - Rather than executing on ordinary states
  - Finite-sized descriptors representing a collection of states
- "Run in non-standard way"
  - Run in fragments
  - Stitch them together to cover all paths
- Runtime testing is inherently incomplete, but static analysis can cover all paths

Static Analysis Example

- Descriptors represent the sign of a value
  - Positive, negative, zero, unknown
- For instruction, *c = a * b*
  - If *a* has a descriptor *pos*
  - And *b* has a descriptor *neg*
- What is the descriptor for *c* after that instruction?
- How might this help?

Descriptors

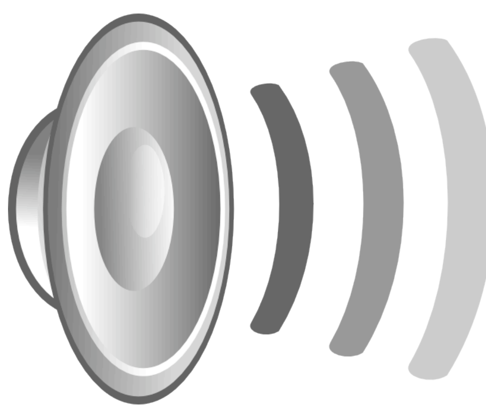
- Choose a set of descriptors that
  - Abstracts away details to make analysis tractable
  - Preserves enough information that key properties hold
    - Can determine interesting results
- Using *sign* as a descriptor
  - Abstracts away specific integer values (billions to four)
  - Guarantees when *a*b = 0* it will be zero in all executions
- Choosing descriptors is one key step in static analysis

Precision

- Abstraction loses some precision
- Enables run in aggregate, but may result in executions that are not possible in the program
  - *(a <= b)* when both are *pos*
  - If *b* is equal to *a* at that point, then false branch is never possible in concrete executions
- Results in false positives

Soundness

- The use of descriptors "over-approximates" a program's possible executions
- Abstraction must include all possible legal values
  - May include some values that are not actually possible
- The run-in-aggregate must preserve such abstractions
  - Thus, must propagate values that are not really possible



Implications of Soundness

- Enables proof that a class of vulnerabilities are completely absent
  - No false negatives in a sound analysis
- Comes at a price
  - Ensuring soundness can be complex, expensive, cautious
- Thus, unsound analyses have gained in popularity
  - Find bugs quickly and simply
  - Such analyses have both false positives and false negatives

What Is Static Analysis?

- Abstract Interpretation
  - Execute the system on a simpler data domain
    - Descriptors of the *abstract domain*
  - Rather than the *concrete domain*
- Elements in an abstract domain represent sets of concrete states
  - Execution mimics all concrete states at once
- Abstract domain provides an over-approximation of the concrete domain

Abstract Domain Example

- Use interval as abstract domain
  - *b* = [40, 41]
- *a* = 2**b*
  - a = [x, y]?
- What are the possible concrete values represented?
  - Which concrete states are possible?

Joins

- A join combines states from multiple paths
  - Approximates set-union as either path is possible
- Use Interval as abstract domain
  - *a* = [36, 39], *b* = [40, 41]
- *If (a >= 38) a=2*b; /* join */*
  - a = [x, y], b=[40, 41] – what are x and y?
- What's the impact of over-approximation?

Impact of Abstract Domain

- The choice of abstract domain must preserve the over-approximation to be sound (no false negatives)
- Integer arithmetic vs 2's-complement arithmetic
- *a = [126, 127], b = [10, 12]*
  - What is c = *a+b* in an 32-bit machine?
  - What is c = *a+b* in an 8-bit machine?

Successive Approximation

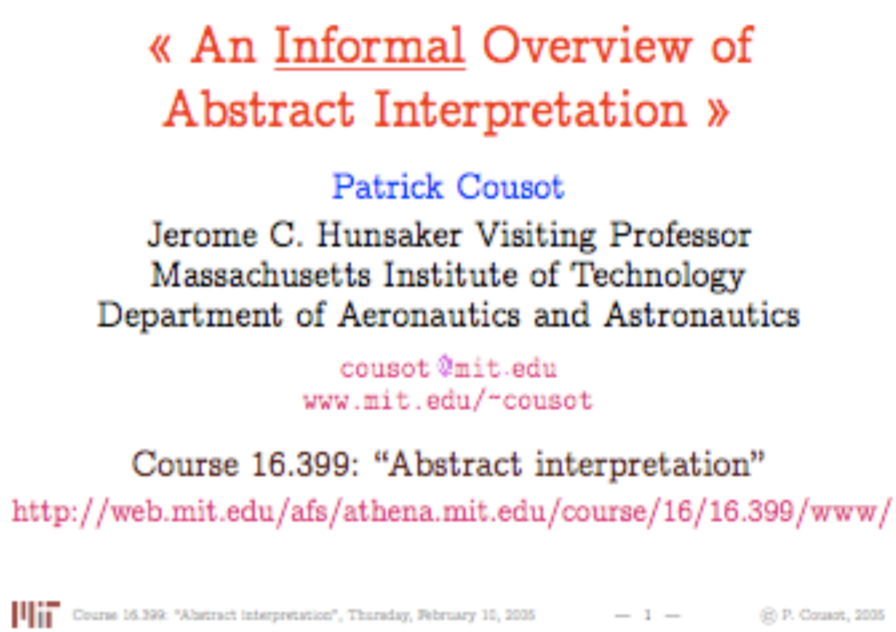
- The abstract execution of a system can often be cast as a problem of solving a set of equations by means of successive approximation.
- If constructed correctly, the execution of the system in the abstract domain over-approximates the semantics of the original system
    - Any behavior not exhibited by the abstract domain cannot be exhibited during concrete system execution.

Abstract Interpretation

- Patrick Cousot
  - Class slides/notes from MIT
  - http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/

Abstract Interpretation

- Patrick Cousot
  - Class slides/notes from MIT
  - http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/



« An Informal Overview of Abstract Interpretation »

Patrick Cousot

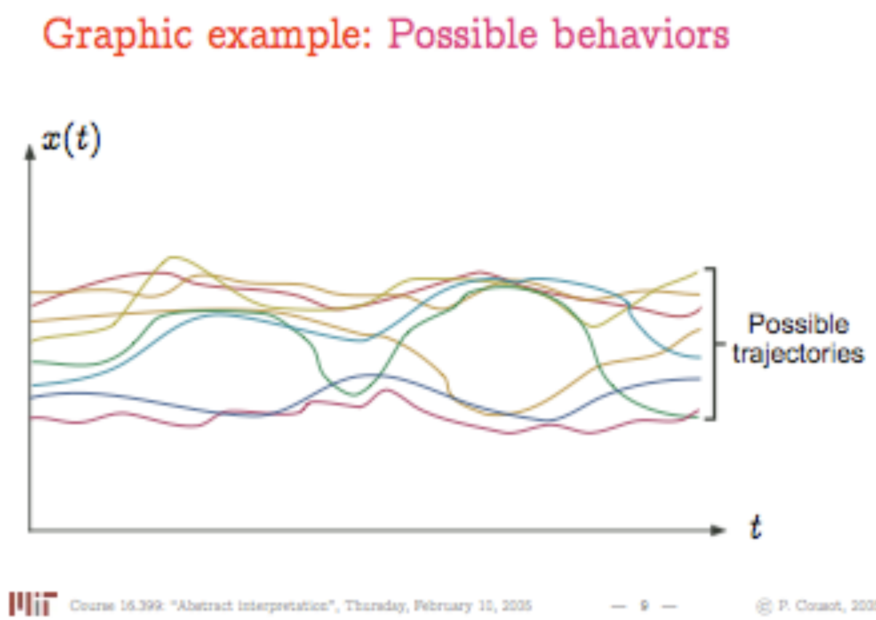
Jerome C. Hunsaker Visiting Professor
Massachusetts Institute of Technology
Department of Aeronautics and Astronautics

cousot@mit.edu
www.mit.edu/~cousot

Course 16.399: "Abstract interpretation"
http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/

Course 16.399: "Abstract interpretation", Thursday, February 10, 2005 — 1 — © P. Cousot, 2005

Abstract Interpretation



Abstract Interpretation

Undecidability

- The concrete mathematical semantics of a program is an "infinite" mathematical object, *not computable*;
- All non trivial questions on the concrete program semantics are *undecidable*.

Example: Kurt Gödel argument on termination

- Assume `termination(P)` would always terminates and returns `true` iff `P` always terminates on all input data;
- The following program yields a contradiction

$$\texttt{P} \equiv \texttt{while termination(P) do skip od.}$$

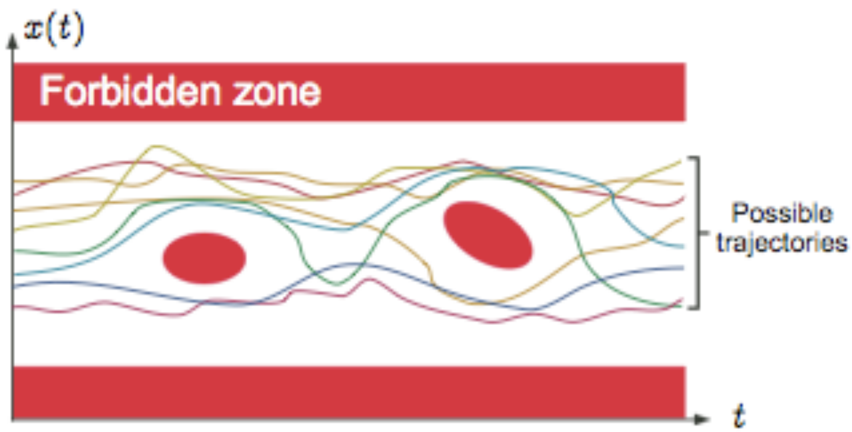
Course 16.399: "Abstract interpretation", Thursday, February 10, 2005 — 18 — © P. Cousot, 2005

Abstract Interpretation

Graphic example: Safety properties

The *safety properties* of a program express that no possible execution in any possible execution environment can reach an erroneous state.

Graphic example: Safety property



Safety proofs

- A safety proof consists in proving that the intersection of the program concrete semantics and the forbidden zone is empty;
- Undecidable problem (the concrete semantics is not computable);
- Impossible to provide completely automatic answers with finite computer resources and neither human interaction nor uncertainty on the answer [2].

[2] e.g. probabilistic answer.

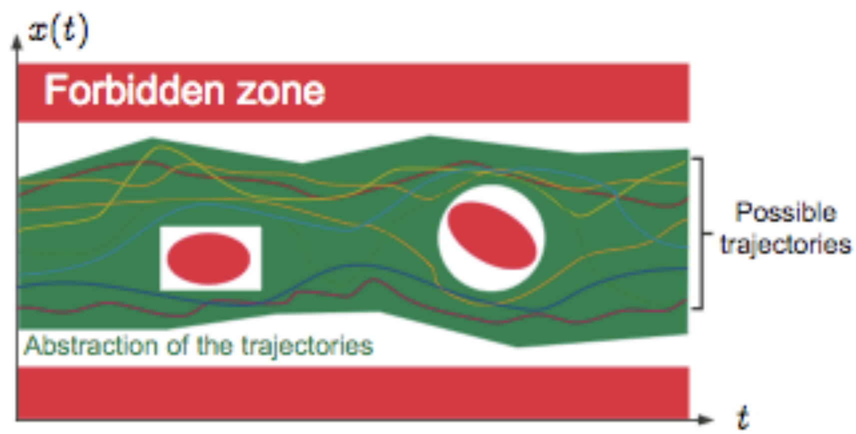
Abstract Interpretation

Abstract interpretation

– consists in considering an *abstract semantics*, that is to say a superset of the concrete semantics of the program;
– hence the abstract semantics covers all possible concrete cases;
– correct: if the abstract semantics is safe (does not intersect the forbidden zone) then so is the concrete semantics.

Course 16.399: "Abstract interpretation", Thursday, February 10, 2005 — 16 — © P. Cousot, 2005

Graphic example: Abstract interpretation



Course 16.399: "Abstract interpretation", Thursday, February 10, 2005 — 17 — © P. Cousot, 2005

Abstract Interpretation

PENNSTATE 1855

Formal methods

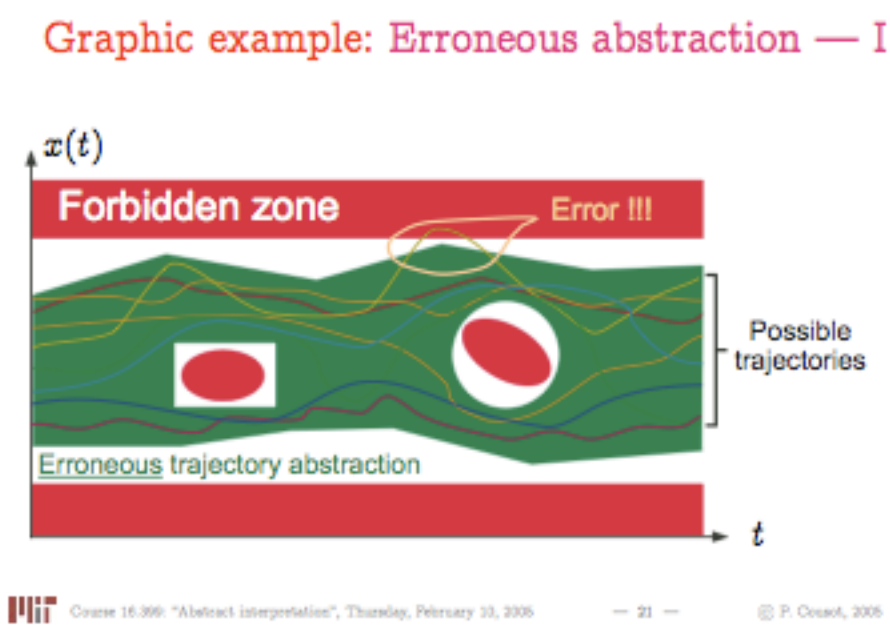
Formal methods are abstract interpretations, which differ in the way to obtain the abstract semantics:

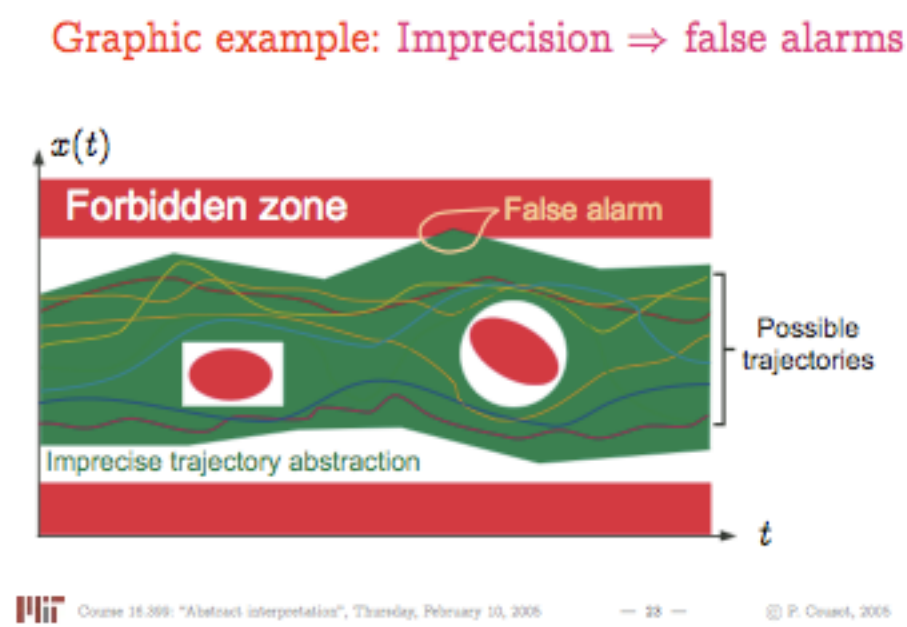
– "*model checking*":
  - the abstract semantics is given manually by the user;
  - in the form of a finitary model of the program execution;
  - can be computed automatically, by techniques relevant to static analysis.

– "*deductive methods*":
  - the abstract semantics is specified by verification conditions;
  - the user must provide the abstract semantics in the form of inductive arguments (e.g. invariants);
  - can be computed automatically by methods relevant to static analysis.

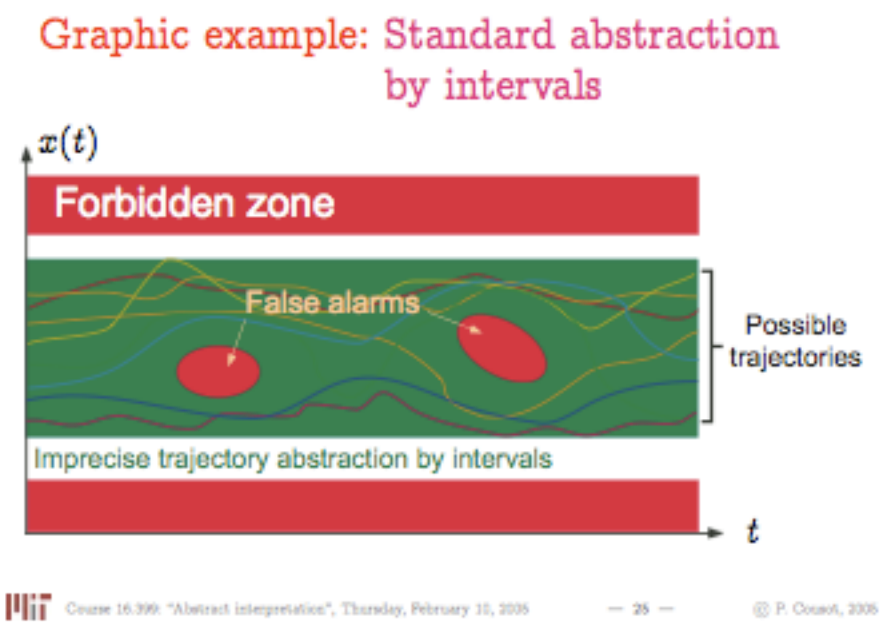
– "*static analysis*": the abstract semantics is computed automatically from the program text according to predefined abstractions (that can sometimes be tailored automatically/manually by the user).

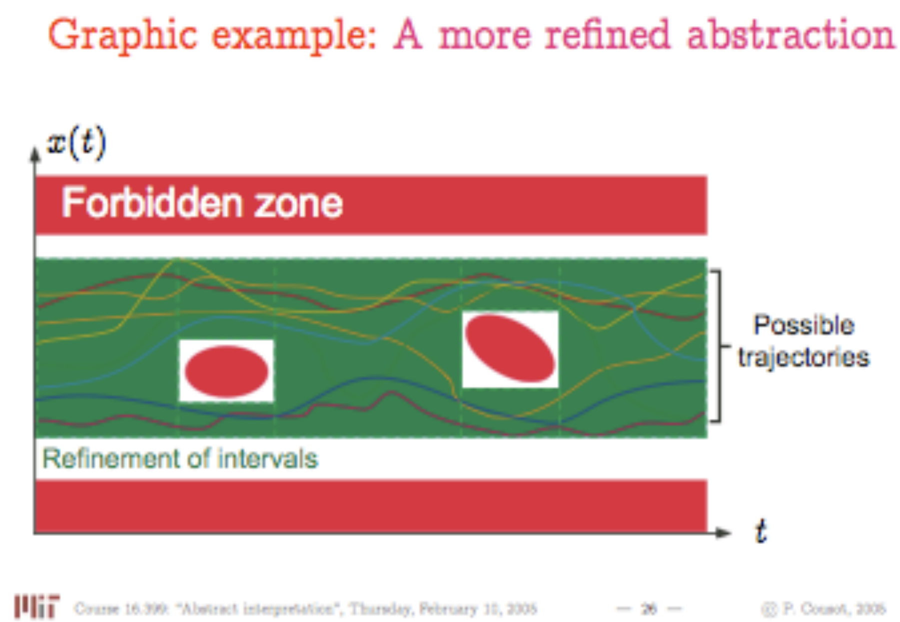
Abstract Interpretation





Abstract Interpretation





Abstract Interpretation

Abstraction by Galois connections

Abstracting sets (i.e. properties)

- Choose an abstract domain, replacing sets of objects (states, traces, ...) S by their abstraction $\alpha(S)$
- The abstraction function α maps a set of concrete objects to its abstract interpretation;
- The inverse concretization function γ maps an abstract set of objects to concrete ones;
- Forget no concrete objects: (abstraction from above) $S \subseteq \gamma(\alpha(S))$.

Abstract Interpretation

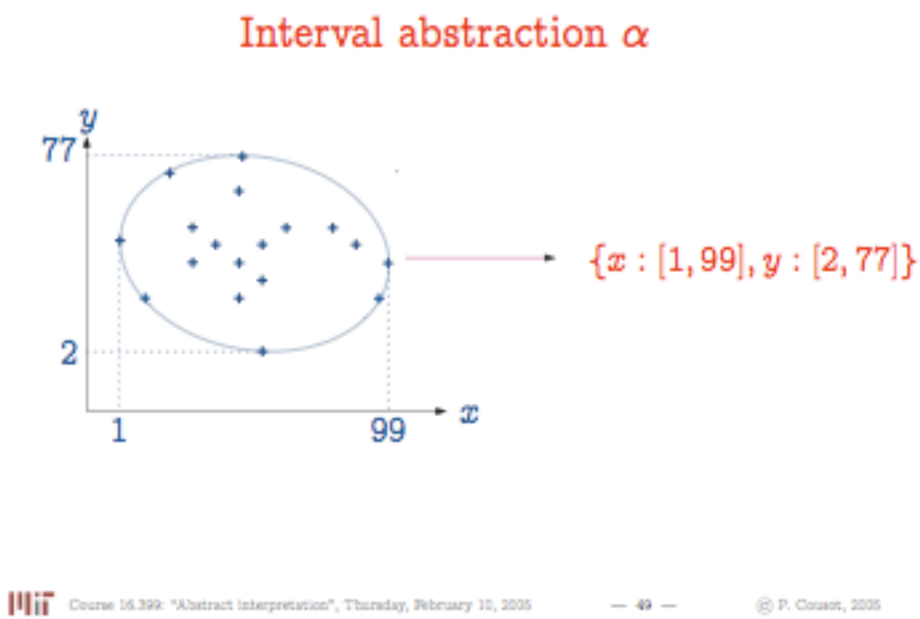
PENNSTATE 1855

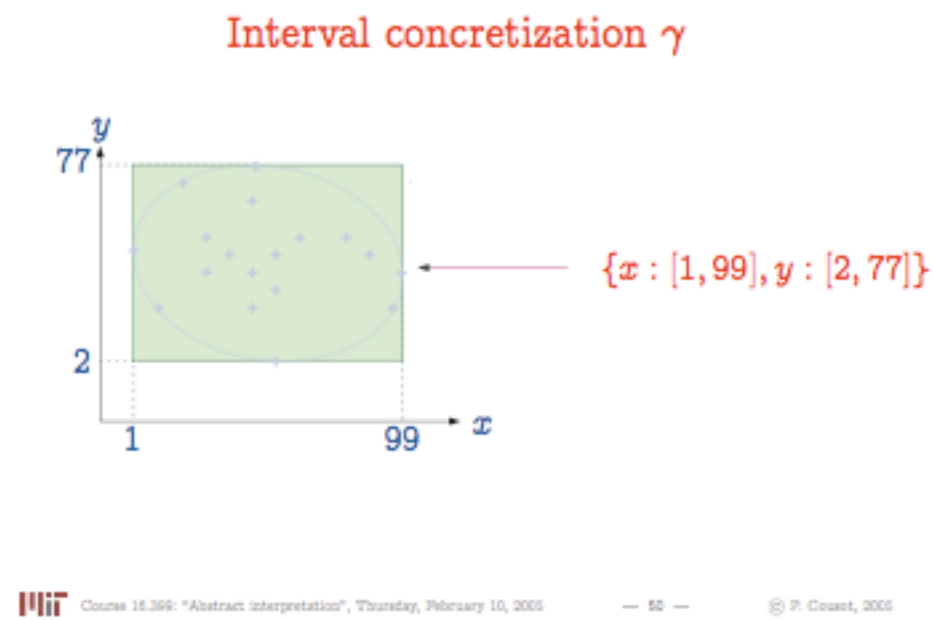
Abstraction by Galois connections

Abstracting sets (i.e. properties)

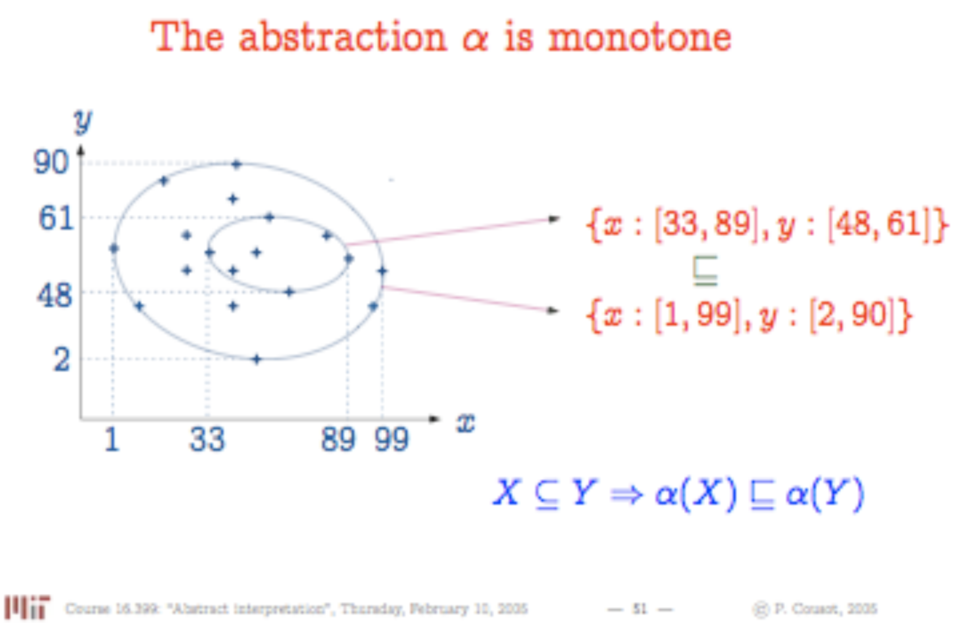
- Choose an abstract domain, replacing sets of objects (states, traces, ...) S by their abstraction $\alpha(S)$
- The abstraction function α maps a set of concrete objects to its abstract interpretation;
- The inverse concretization function γ maps an abstract set of objects to concrete ones;
- Forget no concrete objects: (abstraction from above) $S \subseteq \gamma(\alpha(S))$.

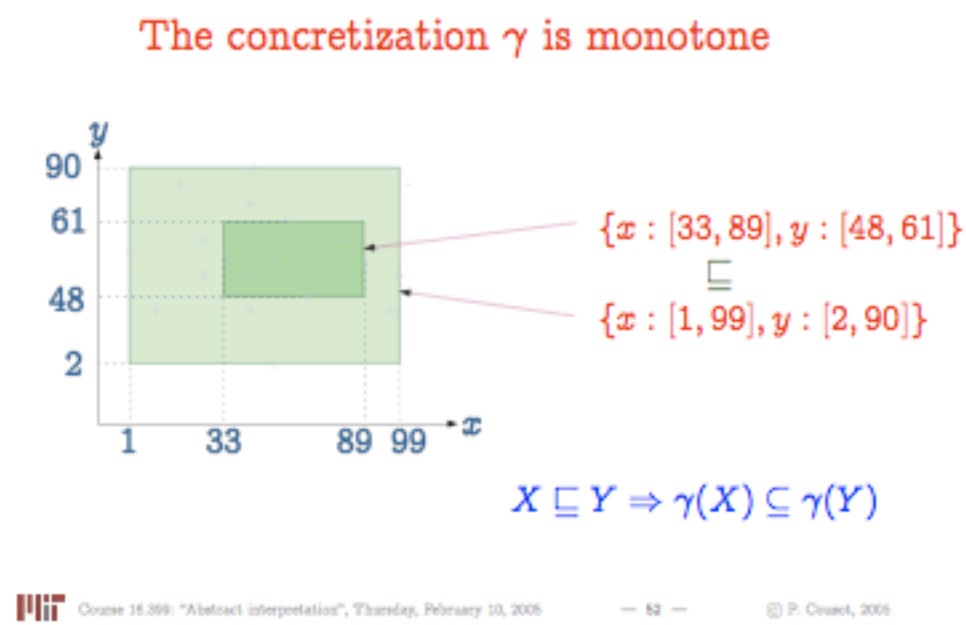
Abstract Interpretation



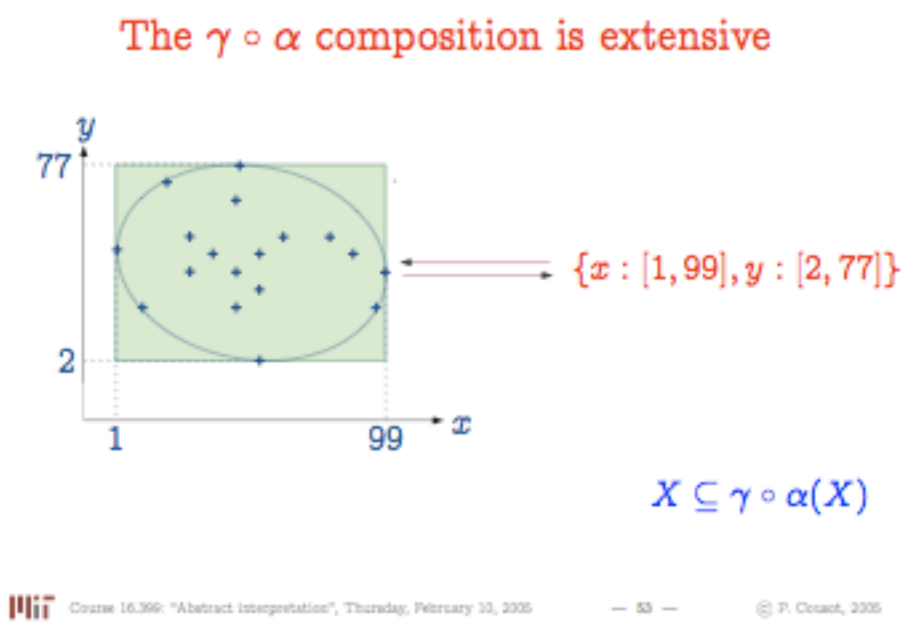


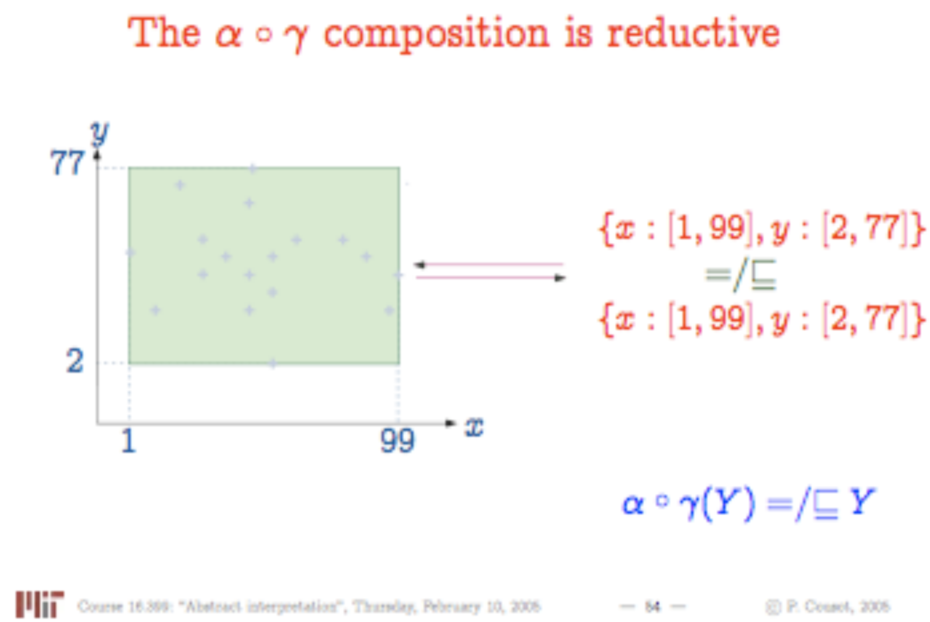
Abstract Interpretation





Abstract Interpretation





Abstract Interpretation

Galois connection

$$\langle \mathcal{D}, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\longleftrightarrow}} \langle \overline{\mathcal{D}}, \sqsubseteq \rangle$$

iff $\forall x, y \in \mathcal{D} : x \subseteq y \Longrightarrow \alpha(x) \sqsubseteq \alpha(y)$

$\wedge\ \forall \overline{x}, \overline{y} \in \overline{\mathcal{D}} : \overline{x} \sqsubseteq \overline{y} \Longrightarrow \gamma(\overline{x}) \subseteq \gamma(\overline{y})$

$\wedge\ \forall x \in \mathcal{D} : x \subseteq \gamma(\alpha(x))$

$\wedge\ \forall \overline{y} \in \overline{\mathcal{D}} : \alpha(\gamma(\overline{y})) \sqsubseteq \overline{x}$

iff $\forall x \in \mathcal{D}, \overline{y} \in \overline{\mathcal{D}} : \alpha(x) \sqsubseteq y \Longleftrightarrow x \subseteq \gamma(y)$

Course 16.399: "Abstract interpretation", Thursday, February 10, 2005 — 56 — © P. Cousot, 2005

Lattices

PENNSTATE 1855

- A partially ordered set (poset) in which any two elements have a
  - Greatest lower bound (meet)
  - Least upper bound (join)
- Semilattice has one or the other (join or meet)
- Claim: any abstract interpretation must express at least a join semilattice

Lattices

Generalizing to complete lattices

- The reasoning on abstractions of concrete properties $\langle \wp(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap, \neg \rangle$ to an abstract domain which, in case of best abstraction is a Moore family, whence a complete lattice, can be generalized to an arbitrary concrete complete lattice $\langle L, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$
- This allow a compositional approach where $\langle L, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ is abstracted to $\langle A_1, \sqsubseteq_1, \bot_1, \top_1, \sqcup_1, \sqcap_1 \rangle$ which itself can be further abstracted to $\langle A_2, \sqsubseteq_2, \bot_2, \top_2, \sqcup_2, \sqcap_2 \rangle, \ldots$

Course 16.399: "Abstract interpretation", Tuesday, April 12, 2005 — 98 — © P. Cousot, 2005

Lattices

Why are abstract domains complete lattices in the presence of best abstractions?

- The abstractions start from the complete lattice of concrete properties $\langle \wp(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap, \neg \rangle$ where objects in Σ represent program computations and the elements of $\wp(\Sigma)$ represent properties of these program computations
- We have defined abstract domains with best approximations in three equivalent different ways (more are considered in [3])
  - As a Moore family;
  - As a closure operator (which fixpoints form the abstract domain);
  - As the image of the concrete domain by a Galois surjection.

- In all cases, it follows that the abstract domain is a complete lattice, since we have seen that:
  - A Moore family of a complete lattice is a complete lattice;
  - The image of a complete lattice by an upper closure operator is a complete lattice (Ward);
  - The image of a complete latticc by the surjective abstraction of a Galois connection is a complete lattice.
- In general this property does not hold in absence of best abstraction or if arbitrary points are added to the abstract domain as shown next.

Reference

[3] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY, U.S.A.

Lattices Too Limiting?

- Does the requirement for an abstract interpretation that is a lattice too restrictive?
  - How can we build a lattice for a set of values?
  - How do we combine two sets of values representing two properties into a lattice?
  - What are the pros/cons of these results?

Dataflow Analysis

- Interprocedural Control Flow Graph (ICFG)
  - Possible flow paths in system
- Join Semilattice for an Abstract Interpretation
  - How to combine values on joins
- Initial Configuration for the Abstract Interpretation
  - Starting values for system
- Dataflow Transfer Function over edges in ICFG
  - How values are changed by operations in system

Intraprocedural CFG

- Statements
  - Nodes
  - One successor and one predecessor
- Basic Blocks
  - Multiple successors to the join (multiple predecessors)
  - Examples?
- Unique Enter and Exit
  - All start nodes are successors of enter
  - All return nodes are predecessors of exit

Legal and Illegal Paths

- Interprocedurally, connect CFGs
  - Calls → Enter
  - Exit → Return-Site
- Want to represent only legal paths
  - In particular, calls must match returns
    - Will discuss the implications of this later
- Example…

Path Function Problem

- A path of length $j >= 1$ from node m to node n is a (non-empty) sequence of j edges,
- denoted by $[e_1, e_2, \ldots, e_j]$, such that
  - the source of e_1 is m,
  - the target of e_j is n,
  - and for all i, $1 <= i <= j-1$, the target of edge e_i is the source of edge e_{i+1}.

Intraprocedural Dataflow Analysis

- The path function pf_q for path $q = [e_1, e_2, \ldots, e_j]$ is the composition, in order, of q's transfer functions
  - $pf_q = M(e_j) \circ \ldots \circ M(e_2) \circ M(e_1)$
- In intraprocedural dataflow analysis, the goal is to determine, for each node *n*, the "join-over-all-paths" solution
  - $JOP_n = join(q \text{ in } Paths(enter, n))\ pf_q(v_0)$
    - *Paths(enter, n)* denotes the set of paths in the CFG from enter node to *n*
    - *v_0 is the possible memory configurations at the start of the procedure*
- Soundness depends on the abstract interpretation

Abstract Interpretation

- As discussed above, a sound *JOP_n* solution requires
  - A Galois connection is established between concrete states and abstract states
  - Each dataflow transfer function *M(e)* is shown to overapproximate the transfer function for the concrete semantics of e

Example

Interprocedural Dataflow Analysis

- Find join-over-all-valid-paths
- What is a valid path?
  - Is a matched or valid path
    - Where a valid path has an open call
    - Where a matched path has a matching return for each call
    - Or consists only of edges without calls and returns
- Be able to use the grammar on your own

Join Over All Valid Paths

- Solution is said to be "context-sensitive"
  - A context-sensitive analysis captures the fact that the results propagated back to each return site r should depend only on the memory configurations that arise at the call site that corresponds to r.
- Formal definition
  - $JOVP_n = join(q \text{ in } VPaths(enter_{main}, n))\ pf_q(v_0)$
- $VPaths(enter_{main}, n)$ denotes the set of valid paths from the main entry point to *n*

Summary

- To find and fix bugs, we need to understand how programs and systems work
  - Testing – time-consuming and incomplete
  - Validation – find all bugs
- Static analysis
  - Key concepts: concrete to abstract domains
  - Soundness – No false negatives
- OK, so what do you do with static analysis?
  - E.g., Interprocedural Dataflow Analysis