Overview of the Course

Lattice Theory for Parallel Programming

Pierre Talbot
pierre.talbot@uni.lu

16th September 2025

University of Luxembourg

Why Math?? Why Lattice??

Some cool computer science stuffs are highly inaccessible without the appropriate math background! Applications of **lattice theory** in C.S. include:

- Conflict-free replicated data type (field of distributed computing).
- Parallel lattice programming (field of parallel programming).
- Abstract interpretation (field of software verification).
- Abstract satisfaction (field of combinatorial optimization).
- Neural network verification (field of machine learning).
- Denotational semantics (field of programming languages).

This course gives you foundation in lattice theory and a broad overview of its applications to computer science.

Why?

Studying something only a few people know can unlock very interesting jobs:

@Meta with Sparta:

Abstract: Over 50% of the security vulnerabilities we found across Meta's family of apps (Facebook, Instagram, WhatsApp, Messenger, Oculus...) are detected automatically using Abstract Interpretation-based tools. In the talk, I will present the

@Redis

Why would you love this job?

You will be at the forefront of cutting-edge technology, working on the implementation and optimization
Types (CRDTs) within Redis, one of the most widely used NoSQL databases. This role offers a unique oppor
distributed systems challenges, ensuring high availability and consistency across multiple nodes, while co
dedicated to pushing the boundaries of database technology.

@Academia for a PhD

@Anywhere: solve complex problems in industry.

Lattice Theory in a Nutshell

Partially Ordered Set

A partially ordered set (poset) is essentially a set in which we order its elements:

- Age relation: *is-older-than*.
- Family tree and its *is-parent-of* relation.
- Inheritance relationship in C++.

What other examples of order?

Partially Ordered Set

A partially ordered set (poset) is essentially a set in which we order its elements:

- Age relation: *is-older-than*.
- Family tree and its *is-parent-of* relation.
- Inheritance relationship in C++.

What other examples of order?
What properties does an order should have?

Partially Ordered Set

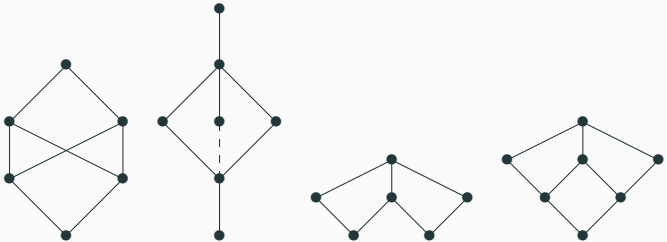
Definition

A partially ordered set (poset) is a tuple $\langle S, \leq \rangle$ where:

- S is a set
- $\leq$ is a binary relation (an *order*) such that $\forall x, y \in S$:
  - **Reflexive**: $x \leq x$.
  - **Antisymmetric**: $x \leq y$ and $y \leq x$ implies $x = y$.
  - **Transitive**: $x \leq y$ and $y \leq z$ implies

Examples

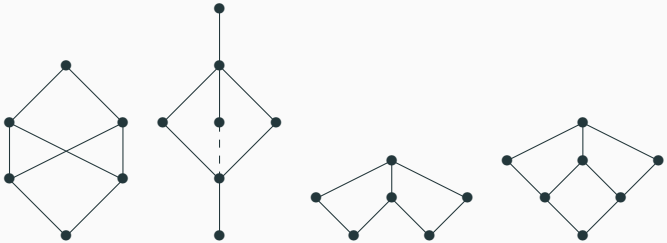
(dashed lines indicate infinite chains)



Lattice

A lattice is a poset with a little bit more structure.

Intuition: We want each pair of element x, y to have a "unique common ancestor" and "unique child".



Join and Meet

Formally, what is the "unique common ancestor" of x and y in a lattice $\langle S, \leq \rangle$?

Least upper bound

Let $U = \{z \in S \mid z \geq x, z \geq y\}$ the set of elements both greater than x and y.

The *least upper bound* (lub) is the smallest element of U, e.g. $s \in U$ such that $\forall t \in U, s \leq t$. The lub is denoted by $x \sqcup y$ (also $x \vee y$ depending on notation).

Join and Meet

Formally, what is the "unique common ancestor" of x and y in a lattice $\langle S, \leq \rangle$?

Least upper bound

Let $U = \{z \in S \mid z \geq x, z \geq y\}$ the set of elements both greater than x and y.
The *least upper bound* (lub) is the smallest element of U, e.g. $s \in U$ such that $\forall t \in U, s \leq t$. The lub is denoted by $x \sqcup y$ (also $x \vee y$ depending on notation).

Exercise: define the meet operation $x \sqcap y$ (or $x \wedge y$) which is defined similarly for the *greatest lower bound* (glb).

Join and Meet

Formally, what is the "unique common ancestor" of x and y in a lattice $\langle S, \leq \rangle$?

Least upper bound

Let $U = \{z \in S \mid z \geq x, z \geq y\}$ the set of elements both greater than x and y.
The *least upper bound* (lub) is the smallest element of U, e.g. $s \in U$ such that $\forall t \in U, s \leq t$. The lub is denoted by $x \sqcup y$ (also $x \vee y$ depending on notation).

Exercise: define the meet operation $x \sqcap y$ (or $x \wedge y$) which is defined similarly for the *greatest lower bound* (glb).

Lattice

A lattice $\langle L, \leq \rangle$ is a poset where the lub and glb exists for all pairs of elements $x, y \in L$.

Connection to CS

A lattice $\langle L, \leq \rangle$ is akind to *types* in CS, e.g., `struct T {...};` in C.

An element of a lattice is akind to an instantiation `T x = ...;`.

What about computation?

Connection to CS

A lattice $\langle L, \leq \rangle$ is akind to *types* in CS, e.g., `struct T {...};` in C.
An element of a lattice is akind to an instantiation `T x = ...;`.
What about computation? Functions over L!

But not any function: the *monotone functions* $(\forall x, y \in L, x \leq y \Rightarrow f(x) \leq f(y))$.
Why?

Connection to CS

A lattice $\langle L, \leq \rangle$ is akind to *types* in CS, e.g., `struct T {...};` in C.
An element of a lattice is akind to an instantiation `T x = ...;`.
What about computation? Functions over L!

But not any function: the *monotone functions* $(\forall x, y \in L, x \leq y \Rightarrow f(x) \leq f(y))$.
Why? By connecting the computation to the lattice order, we can prove some properties such as determinism and termination.

Connection to CS

A lattice $\langle L, \leq \rangle$ is akind to *types* in CS, e.g., `struct T {...};` in C.
An element of a lattice is akind to an instantiation `T x = ...;`.
What about computation? Functions over L!

But not any function: the *monotone functions* $(\forall x, y \in L, x \leq y \Rightarrow f(x) \leq f(y))$.
Why? By connecting the computation to the lattice order, we can prove some properties such as determinism and termination.

Ingredients of Lattice Theory for CS

- Lattice $\approx$ Type
- Element of lattice $\approx$ Value
- Computing fixpoint of monotone function $\approx$ Execution of program

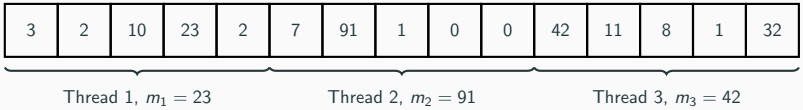
  Fixpoint: A fixpoint is an element $x \in L$ such that $f(x) = x$.

Parallel Lattice Programming

Pessimistic Parallel Programming

Running example: parallel maximum

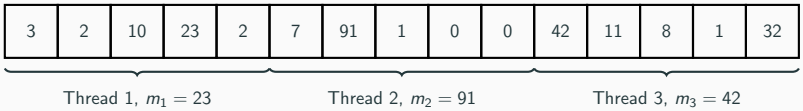
Each thread computes its local max (map), then we compute the max of all local max (reduce).



- Map:
- Reduce: $max([23, 91, 42]) = 91$.

Running example: parallel maximum

Each thread computes its local max (map), then we compute the max of all local max (reduce).



- Map:
- Reduce: $max([23, 91, 42]) = 91$.

Sequential bottleneck: With 100 elements (10 threads), the reduce step takes as much time as the map step.

How to program the reduce step in parallel?

Parallel max

```
/** Suppose as many threads as elements in 'data'. */
void max(int tid, const int* data, int* m) {
  if(data[tid] > *m) {
    *m = data[tid];
  }
}
```

Then you run:

```
*m = MIN_INT;
max(0, data, m) || ... || max(n-1, data, m)
```

where `p || q` is the parallel composition.

Parallel max

```
/** Suppose as many threads as elements in 'data'. */
void max(int tid, const int* data, int* m) {
  if(data[tid] > *m) {
    *m = data[tid];
  }
}
```

Then you run:

```
*m = MIN_INT;
max(0, data, m) || ... || max(n-1, data, m)
```

where `p || q` is the parallel composition.

Good? No! Data-race.

Parallel max fixed!?

```
/** Suppose as many threads as elements in 'data'. */
void max(int tid, const int* data, int* m) {
  if(data[tid] > *m) {
    lock(m) {
      *m = data[tid];
    }
  }
}
```

Parallel max fixed!?

```
/** Suppose as many threads as elements in 'data'. */
void max(int tid, const int* data, int* m) {
  if(data[tid] > *m) {
    lock(m) {
      *m = data[tid];
    }
  }
}
```

Good? No!

Can produce wrong results.

Parallel max fixed again!?

```
/** Suppose as many threads as elements in 'data'. */
void max(int tid, const int* data, int* m) {
  lock(m) {
    if(data[tid] > *m) {
      *m = data[tid];
    }
  }
}
```

Parallel max fixed again!?

```
/** Suppose as many threads as elements in 'data'. */
void max(int tid, const int* data, int* m) {
  lock(m) {
    if(data[tid] > *m) {
      *m = data[tid];
    }
  }
}
```

Good? Yes!

But our "parallel" algorithm is now sequential.

Atomics to the rescue (?)

C++26 atomics can unlock lock-free programming for better efficiency :)

```
void max(int tid, const int* data, std::atomic<int>& m) {
  m.fetch_max(data[tid]);
}
```

Atomics to the rescue (?)

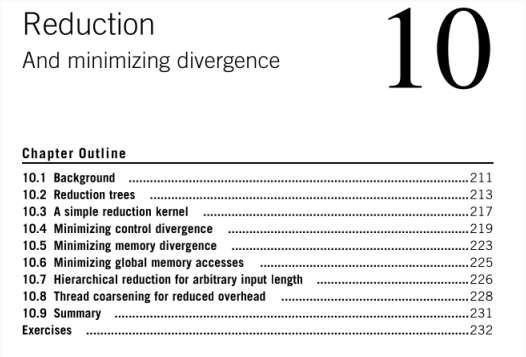
C++26 atomics can unlock lock-free programming for better efficiency :)

```
void max(int tid, const int* data, std::atomic<int>& m) {
  m.fetch_max(data[tid]);
}
```

Atomic operations are (much) slower than traditional operations.

Reduction in CUDA

Chapter 10 in book “Programming Massively Parallel Processors: A Hands-on Approach”.



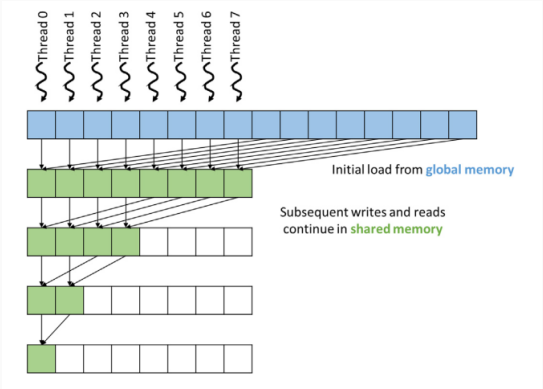
Reduction
And minimizing divergence

10

Chapter Outline

Reduction in CUDA

Chapter 10 in book “Programming Massively Parallel Processors: A Hands-on Approach”.



Not easy, and eventually, some threads inactive.

Multithreading programming is pessimistic.

For a data race that happens once in million instructions, this model:

- Makes parallel programming painful and difficult.
- Slows down computation.
- Prevents us from thinking with a true parallel mindset.

Optimistic Parallel Programming

Let's be optimistic

Instead of being afraid of data races, let's welcome them as part of the programming model itself.

```
void max(int tid, const int* data, int* m) {
  if(data[tid] > *m) {
    *m = data[tid];
  }
}
```

What happens in case of a data race?

- Suppose two threads with `data = [1, 2]`.
- If a data race occurs, `*m == 1`.
- But if we run `max` again, then we must obtain `*m == 2`.

Let's do extra work only when data races occur (optimistic)

In case of n data races, we run the algorithm $n + 2$ times:

```
int old = *m + 1;
while(old != *m) {
  old = *m;
  max(0, data, m) || ... || max(n-1, data, m);
}
```

This is called the *fixpoint loop*.

The Bigger Picture

We have computed a fixpoint over a lattice data structure!

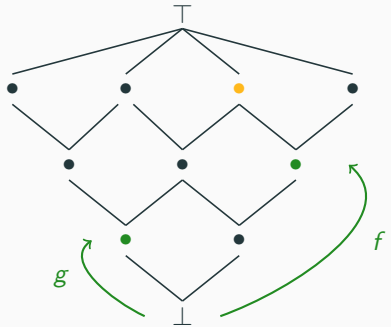
The Bigger Picture

We have computed a fixpoint over a lattice data structure!

- Lattice of increasing integers: $ZI = \langle \mathbb{Z}, \leq \rangle$, modelled by an `int` type.
- Fixpoint of the function $f \triangleq \max(data[0], m) \circ \ldots \circ \max(data[n-1], m)$ on the element $m \in ZI$.
- The fixpoint of f is the maximum of the array!

We will introduce a parallel model of computation over lattice!

Intuition: Lattice to Reconciliate Reduction



- Let f and g be two functions executed by two threads.
- The *join operator* $\sqcup$ acts as a sound reduction to obtain ●.
- Least fixpoint computation: `lfp (f || g)` = ●.

Application: Constraint Solving on GPU

Using this paradigm, we have built *Turbo*[1]:

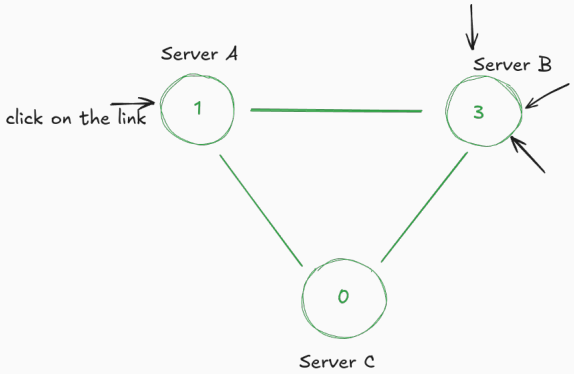
- **First general constraint solver fully executing on GPU (propagation + search).**
  ⇒ **General**: Support MiniZinc and XCSP3 constraint models.
  ⇒ **Simple**: interval-based constraint solving + backtracking search (no global constraints, learning, restart, event-based propagation, ...).
  ⇒ **Efficient?**: On-par with Choco.
  ⇒ **Open-source**: Publicly available on `https://github.com/ptal/turbo`.
- **Ternary constraint network**: representation of constraints suited for GPU architectures.

[1] Talbot P. et al., *A Variant of Concurrent Constraint Programming on GPU*, AAAI, 2022.

Conflict-free Replicated Data Type (CRDT)

Conflict-free Replicated Data Type (CRDT)

Same idea in the context of distributed systems.
For instance a replicated counter (e.g. “likes” on a picture).

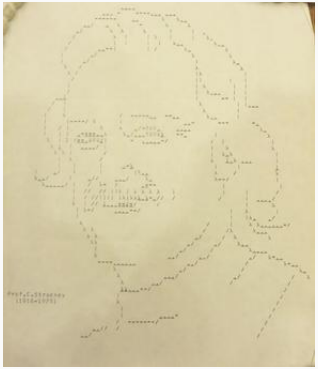


CRDT = Lattice + Monotone Functions

Denotational Semantics

Denotational Semantics

One of the first application of lattice theory was not to "compute with lattice" but to formally describe the meaning of a program. Developed by Christopher Strachey and Dana Scott in the 70s:



Denotational Semantics

One of the first application of lattice theory was not to "compute with lattice" but to formally describe the meaning of a program. Developed by Christopher Strachey and Dana Scott in the 70s:

Why?

- Without formalization, it is hard to reason on what is a program doing, and to prove it is actually doing it right!
- Capture the essence of a programming language, and to show its implementation is correct.
- Unlock formal static analysis: *abstract interpretation*.
  $\Rightarrow$ Proving the absence of bugs!

Attempt 1: Defining a Program Mathematically

How to assign to a program P a mathematical meaning?

What does `x := 3` means, mathematically?

[2] For simplicity, let's restrict us to integers

Attempt 1: Defining a Program Mathematically

How to assign to a program P a mathematical meaning?

What does `x := 3` means, mathematically?

Let's try: we define an *environment* from variables to values: a function[2] $Env \triangleq Var \to \mathbb{Z}$.

- The denotation of `x := 3` is the function $\{x \mapsto 3\} \in Env$.
- The denotation of `x := 3; y := 4` is the function

[2] For simplicity, let's restrict us to integers

Attempt 1: Defining a Program Mathematically

How to assign to a program P a mathematical meaning?

What does `x := 3` means, mathematically?

Let's try: we define an *environment* from variables to values: a function[2] $Env \triangleq Var \to \mathbb{Z}$.

- The denotation of `x := 3` is the function $\{x \mapsto 3\} \in Env$.
- The denotation of `x := 3; y := 4` is the function $\{x \mapsto 3, y \mapsto 4\}$.
- The denotation of `x := 3; x := 4` is the function

[2] For simplicity, let's restrict us to integers

Attempt 1: Defining a Program Mathematically

How to assign to a program P a mathematical meaning?

What does `x := 3` means, mathematically?

Let's try: we define an *environment* from variables to values: a function[2] $Env \triangleq Var \to \mathbb{Z}$.

- The denotation of `x := 3` is the function $\{x \mapsto 3\} \in Env$.
- The denotation of `x := 3; y := 4` is the function $\{x \mapsto 3, y \mapsto 4\}$.
- The denotation of `x := 3; x := 4` is the function $\{x \mapsto 4\}$.

[2] For simplicity, let's restrict us to integers

Attempt 1: Defining a Program Mathematically

How to assign to a program P a mathematical meaning?

What does `x := 3` means, mathematically?

Let's try: we define an *environment* from variables to values: a function[2] $Env \triangleq Var \to \mathbb{Z}$.

- The denotation of `x := 3` is the function $\{x \mapsto 3\} \in Env$.
- The denotation of `x := 3; y := 4` is the function $\{x \mapsto 3, y \mapsto 4\}$.
- The denotation of `x := 3; x := 4` is the function $\{x \mapsto 4\}$.

But what to do if the denotation *depends on the input of the program?*

[2] For simplicity, let's restrict us to integers

Attempt 2: Defining a Program Mathematically

Suppose the function `x := y + 1;` where `y` is an input of the program (e.g. a function's parameter or from a call to `scanf`).
What is its denotation?

Attempt 2: Defining a Program Mathematically

Suppose the function `x := y + 1;` where `y` is an input of the program (e.g. a function's parameter or from a call to `scanf`).
What is its denotation? $\{x \mapsto y + 1\}$? But we don't know y (function is $\mathit{Var} \to \mathbb{Z}$)

Attempt 2: Defining a Program Mathematically

Suppose the function `x := y + 1;` where `y` is an input of the program (e.g. a function's parameter or from a call to `scanf`).
What is its denotation? $\{x \mapsto y + 1\}$? But we don't know y (function is $\mathit{Var} \to \mathbb{Z}$)

We must lift everything to $\mathit{Env} \to \mathit{Env}$: the denotation is a function modifying an environment, called a *state transformer*.

Example State Transformer

Formally...

- Let $\rho \in \mathit{Env}$ be an environment, e.g. $\{y \mapsto 1\}$.
- Let $S[\![.]\!] \in \mathit{Program} \to (\mathit{Env} \to \mathit{Env})$ be a state transformer.
- Assignment: $S[\![x := e]\!]\rho = \rho[x \mapsto \mathit{eval}(e)]$.

Example State Transformer

Formally...

- Let $\rho \in \mathit{Env}$ be an environment, e.g. $\{y \mapsto 1\}$.
- Let $S[\![.]\!] \in \mathit{Program} \to (\mathit{Env} \to \mathit{Env})$ be a state transformer.
- Assignment: $S[\![x := e]\!]\rho = \rho[x \mapsto \mathit{eval}(e)]$.
- Evaluation:
  - $\mathit{eval}(x, \rho) = \rho(x)$.
  - $\mathit{eval}(c, \rho) = c$ where $c \in \mathbb{Z}$.
  - $\mathit{eval}(e_1 + e_2, \rho) = \mathit{eval}(e_1, \rho) + \mathit{eval}(e_2, \rho)$.

Example State Transformer

Formally...

- Let $\rho \in \mathit{Env}$ be an environment, e.g. $\{y \mapsto 1\}$.
- Let $S[\![.]\!] \in \mathit{Program} \to (\mathit{Env} \to \mathit{Env})$ be a state transformer.
- Assignment: $S[\![x := e]\!]\rho = \rho[x \mapsto \mathit{eval}(e)]$.
- Evaluation:
  - $\mathit{eval}(x, \rho) = \rho(x)$.
  - $\mathit{eval}(c, \rho) = c$ where $c \in \mathbb{Z}$.
  - $\mathit{eval}(e_1 + e_2, \rho) = \mathit{eval}(e_1, \rho) + \mathit{eval}(e_2, \rho)$.

Example

$$S[\![x := y + 1]\!]\rho = \rho[x \mapsto \mathit{eval}(y + 1, \rho)] = \{x \mapsto 2, y \mapsto 1\}$$

The denotation of `x := y + 1` is the function $S[\![x := y + 1]\!]$.

Compositionality

A principle of denotational semantics is *compositionality*, we can define the denotational semantics of atomic statement and build the semantics of compound statements from it.

Exercise: How to define the denotational semantics of the sequence operator $s_1; s_2$ where s_1 and s_2 are statements (either assignment or themselvers sequences)?

Going further: Fixpoint of state transformer can be used to describe the denotation of loop and recursive functions!

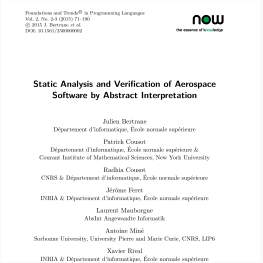
Abstract Interpretation

Abstract Interpretation

General, automated, incomplete and sound.

Success story: Astrée, prove absence of bugs in synchronous control/command aerospace software (Airbus).

Invented by Patrick and Radhia Cousot in the seventies.[3]



[3] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: POPL 77'.

Abstract Interpretation

Abstract interpretation answers precisely elementary questions:

- What is a program?
- What is a property of a program?
- What is the verification problem?

It builds on a theory of approximation:

- **Concrete semantics**: the mathematical denotation of the program.
- **Abstract semantics**: an approximation of the concrete semantics in order to design effective verification algorithm.
- The connection between the two is formalized by *Galois connection*.

Abstract Satisfaction

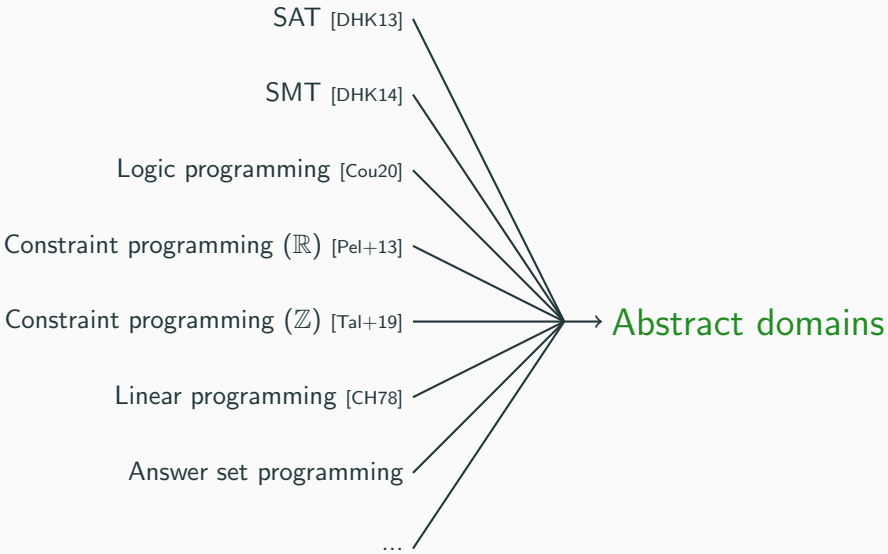
One Problem, Many Communities, Many Formalisms

Many communities emerged to solve the same problem: find ρ such that $A \models_\rho \varphi$.

BUT they (generally) focus on different fragments of FOL:

- Propositional fragment (SAT): $(a \vee b) \wedge (\neg b \vee c)$ with $a, b, c \in \{0, 1\}$.
- Pseudo-Boolean fragment: $\sum_{1 \leq i \leq n} c_i * a_i \leq c_0$ with $a_i \in \{0, 1\}$ and c_i some integers constants.
- Linear programming (LP): $\sum_{1 \leq i \leq n} c_i * b_i \leq b_0$ with $b_i \in \mathbb{R}$ and c_i some real constants.
- Integer linear programming (ILP): $\sum_{1 \leq i \leq n} c_i * b_i \leq b_0$ with $b_i \in \mathbb{Z}$ and c_i some integer constants.
- Mixed integer linear programming (MILP): $\sum_{1 \leq i \leq n} c_i * b_i \leq b_0$ with $b_i \in \mathbb{Z}$ or $b_i \in \mathbb{R}$ and c_i some integer or real constants.
- Uninterpreted fragment (logic programming).
- Discrete constraint programming: $\langle X, D, C \rangle$ with $D_i \in \mathcal{P}_f(\mathbb{Z})$.
- Continuous constraint programming: $\langle X, D, C \rangle$ with $D_i \in \mathcal{I}(\mathbb{R})$.
- Satisfiability modulo theories (SMT).
- ...

One Theory to Rule Them All?



Abstract Satisfaction

- **Concrete domain**: Solutions of a combinatorial problem.
- **Abstract domain**: Approximation of the set of solutions.
- **Fixpoint**: Computing the solutions of the problems.

Use the same theoretical framework for different methods.
Apply techniques from one field to another.

Neural Network Verification

Neural networks are widely used in many applications

- Public Safety and Security
- Image and Video Recognition
- Medical Diagnosis
- ...

But, neural networks are vulnerable to adversarial examples

An **adversarial example** is a correctly classified input with small noise that causes the neural networks to produce an incorrect result despite the modifed input appearing normal to humans.

stop sign
Confidence: 0.9153

Adversarial perturbation

flowerpot
Confidence: 0.8374

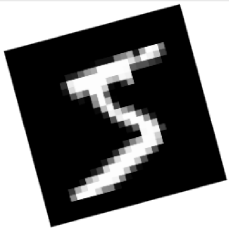
To ensure the reliability of neural networks

Definition: Preconditions

The preconditions in the input layer are defined by the set $\mathbf{\Phi}(\boldsymbol{x}_0, \epsilon) \triangleq \{\boldsymbol{x} \in \mathbb{R}^{d_{in}} \mid p(\boldsymbol{x}, \boldsymbol{x}_0) \leq 0\}$, where $p\colon \mathbb{R}^{d_{in}} \times \mathbb{R}^{d_{in}} \to \mathbb{R}$ is a function defining a perturbation and $\epsilon \in \mathbb{R}$ is the maximum perturbation.

Origin Image

L infinity

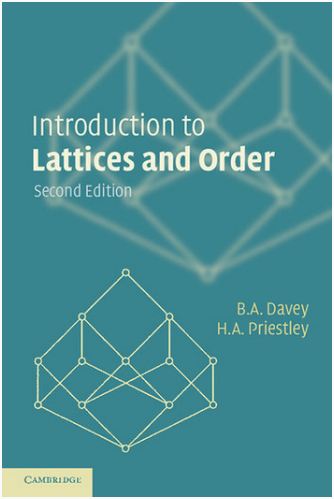


Rotation

- **Concrete domain**: Set of all perturbated inputs.
- **Abstract domain**: Approximation of the set.
- **Fixpoint**: Checking if the perturbated inputs satisfy a property (e.g. are correctly classified).

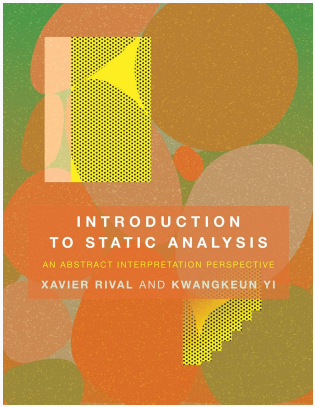
Resources

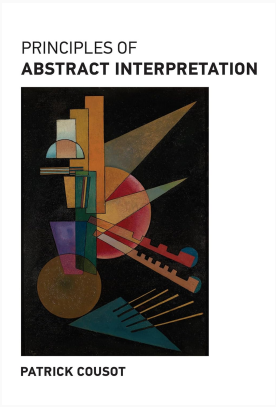
Lattice Theory



Abstract Interpretation

- MPRI class of Antoine Miné: `https://www-apr.lip6.fr/~mine/enseignement/mpri/2023-2024/` (two slides stolen from this class).
- Two recent books:





CRDT

A website with publications and infos: `https://crdt.tech/`

Nowpublisher, 2025

Foundations and Trends® in Programming Languages > Vol 8 > Issue 3-4

Safety and Trust in Artificial Intelligence with Abstract Interpretation

By **Gagandeep Singh**, University of Illinois Urbana-Champaign, USA, ggnds@illinois.edu ✉ | **Jacob Laurel**, Georgia Institute of Technology, USA, jlaurel6@gatech.edu ✉ | **Sasa Misailovic**, University of Illinois Urbana-Champaign, USA, misailo@illinois.edu ✉ | **Debangshu Banerjee**, University of Illinois Urbana-Champaign, USA, db21@illinois.edu ✉ | **Avaljot Singh**, University of Illinois Urbana-Champaign, USA, avaljot2@illinois.edu ✉ | **Changming Xu**, University of Illinois Urbana-Champaign, USA, cx23@illinois.edu ✉ | **Shubham Ugare**, University of Illinois Urbana-Champaign, USA, sugare2@illinois.edu ✉ | **Huan Zhang**, University of Illinois Urbana-Champaign, USA, huanz@illinois.edu ✉