What is this Course?

CAS CS 491: Type Theory and Mechanized Reasoning

January 22, 2023

Outline

Administrivia

(The entire first part of this meeting)

Greetings

My name is Nathan. I am a new(ish) lecturer at BU.

This course is called *Type Theory and Mechanized Reasoning*.

This is an advanced topics course, which means a fair amount of independent work.

This is a new course, and will (likely) a small group, so there will be some room for experimentation.

Course Structure

Two meetings a week ideally very interactive (bring a computer if you can).

Short weekly assignments and a final project (no exams). More details on the final project, to come.

Piazza for course communications and Gradescope for submissions.

Please read through the course webpage and the syllabus for more details.

Goals for Today

Motivate the topics of this course by looking at how they have affected computer science and mathematics.

Go over, at a high level, the topics we'll be covering and the context in which to understand them.

Take a brief tour of the tools that we will be using, along with their applications in real-world settings.

Practice Problem

None today, we haven't learned anything yet.

Outline

Three Recent Events

The Lean theorem prover was used to mechanically verify theorems in condensed mathematics, a new area of mathematics which earned Peter Scholze his Fields medal.

Github reported that roughly 40% of code created by users of Copilot is AI generated, and the CEO of GitHub estimates 80% in the near future.

OpenAI's Q∗ algorithms demonstrated the ability to do elementary mathematics and DeepMind's FunSearch aided the discovery of novel mathematical results.

Three Recent Events

The Lean theorem prover was used to mechanically verify theorems in condensed mathematics, a new area of mathematics which earned Peter Scholze his Fields medal.

Github reported that roughly 40% of code created by users of Copilot is AI generated, and the CEO of GitHub estimates 80% in the near future.

OpenAI's Q$*$ algorithms demonstrated the ability to do elementary mathematics and DeepMind's FunSearch aided the discovery of novel mathematical results.

Three Recent Events

The Lean theorem prover was used to mechanically verify theorems in condensed mathematics, a new area of mathematics which earned Peter Scholze his Fields medal.

Github reported that roughly 40% of code created by users of Copilot is AI generated, and the CEO of GitHub estimates 80% in the near future.

OpenAI's Q$*$ algorithms demonstrated the ability to do elementary mathematics and DeepMind's FunSearch aided the discovery of novel mathematical results.

Visions of a Formalized World

As LLMs become more powerful, more code will be written by algorithms.

As mathematics becomes more complicated, there is a greater need for computer verified certainty.

These two concerns are not that different.

A Though Experiment

Imagine you are given a very complex implementation of sorting produced by a machine learning model.

How can you be certain it is actually an implementation of sorting? How can you know it does what its supposed to do?

If the model produces OCaml code this might be difficult. But what if it used a programming language in which it was impossible to produce incorrect code?

A Though Experiment

Imagine you are given a very complex implementation of sorting produced by a machine learning model.

How can you be certain it is actually an implementation of sorting? How can you know it does what its supposed to do?

If the model produces OCaml code this might be difficult. But what if it used a programming language in which it was impossible to produce incorrect code?

A Though Experiment

Imagine you are given a very complex implementation of sorting produced by a machine learning model.

How can you be certain it is actually an implementation of sorting? How can you know it does what its supposed to do?

If the model produces OCaml code this might be difficult. But what if it used a programming language in which it was impossible to produce incorrect code?

A Thought Experiment (Continued)

Sorting in OCaml would be a function of the following type:

```
val sort : int list -> int list
```

What if the model had to produce a function with a type like:

```
val verified_sort : int list -> int list * proof
```

Given a list l, the function verified_sort returns

- a sorting of l
- a proof that the produced list is a sorting of l

A Thought Experiment (Continued)

Sorting in OCaml would be a function of the following type:

```
val sort : int list -> int list
```

What if the model had to produce a function with a type like:

```
val verified_sort : int list -> int list * proof
```

Given a list l, the function verified_sort returns

- a sorting of l
- a proof that the produced list is a sorting of l

A Thought Experiment (Continued)

Sorting in OCaml would be a function of the following type:

```
val sort : int list -> int list
```

What if the model had to produce a function with a type like:

```
val verified_sort : int list -> int list * proof
```

Given a list l, the function verified_sort returns

- a sorting of l
- a proof that the produced list is a sorting of l

The Burden/Power of Proof

If we have a proof that the algorithm *actually* sorts (which can be verified) then we can accept the code without knowing how it works.

This leaves open a lot of questions…

- What kind of things is this proof?
- How do we verify the proof?
- How do we ensure that the proof proves the right thing?

The Burden/Power of Proof

If we have a proof that the algorithm *actually* sorts (which can be verified) then we can accept the code without knowing how it works.

This leaves open a lot of questions…

- What kind of things is this proof?
- How do we verify the proof?
- How do we ensure that the proof proves the right thing?

Our Motivations

1. More things being automatically generated. It may be soon be more valuable to be able to determine that code is correct than to write correct code.
2. This is a topics course for advanced computer science students (and the motivated philosophy students). It’s possible you’ll end up in the work force after you graduate. This may be one of your last chances to have fun waxing philosophical. Let’s take advantage of that.

Our Motivations

1. More things being automatically generated. It may be soon be more valuable to be able to determine that code is correct than to write correct code.
2. This is a topics course for advanced computer science students (and the motivated philosophy students). It's possible you'll end up in the work force after you graduate. This may be one of your last chances to have fun waxing philosophical. Let's take advantage of that.

At a High Level

This course is about the relationship between logic and computation, played out on *three levels*:

1. **Theory.** formalisms for reasoning about logic (propositions, theories, higher-order type theory)
2. **Implementation.** software for logical formalisms (automated theorem provers, proof search algorithms)
3. **Practice.** techniques for *using* said software (practical dependent types, specifications, proof assistants)

We will focus on 1 and 3 with minor interludes to 2.

Outline

What is logic?

Logic is a framework for modeling of *reasoning* and *language*

Logic is the study of modeling *procedures of reasoning* (from the beginning, logic is about computation, albeit not necessarily with modern (silicon) computers).

Logic is about the trade-off between *expressivity* and *complexity*. The more complex your language and reasoning the harder it to formalize and mechanize.

What is logic?

Logic is a framework for modeling of *reasoning* and *language*

Logic is the study of modeling *procedures of reasoning* (from the beginning, logic is about computation, albeit not necessarily with modern (silicon) computers).

Logic is about the trade-off between *expressivity* and *complexity*. The more complex your language and reasoning the harder it to formalize and mechanize.

What is logic?

Logic is a framework for modeling of *reasoning* and *language*

Logic is the study of modeling *procedures of reasoning* (from the beginning, logic is about computation, albeit not necessarily with modern (silicon) computers).

Logic is about the trade-off between *expressivity* and *complexity*. The more complex your language and reasoning the harder it to formalize and mechanize.

Why study logic?

As computer scientists. We understand programs to be correct if they satisfy a *properties of correctness*. What is the language of those properties of correctness?

As philosophers. Logic plays a role in meta-mathematics and the foundations of mathematics. And constructivity has had a fascinating impact on the recent landscape of the philosophy of mathematics.

Why study logic?

As computer scientists. We understand programs to be correct if they satisfy a *properties of correctness*. What is the language of those properties of correctness?

As philosophers. Logic plays a role in meta-mathematics and the foundations of mathematics. And constructivity has had a fascinating impact on the recent landscape of the philosophy of mathematics.

Kinds of Logic

Propositional logic. the study of Boolean connectives (like and and or) which underlie all logical reasoning.

Predicate logic. the study of *predication*, the ability to organize and distinguish objects in a given domain (like the predicate ”is odd” over natural numbers).

Kinds of Logic

Propositional logic. the study of Boolean connectives (like and and or) which underlie all logical reasoning.

Predicate logic. the study of *predication*, the ability to organize and distinguish objects in a given domain (like the predicate ”is odd” over natural numbers).

Kinds of Logic (Continued)

First-order and Higher-order logic. The study of *quantification*, or the ability to reason uniformly over an entire domain (like the statement ”every natural number is even or is odd”)

Extensions like modal logic and linear logic. Modal logic is the study of *necessity* and linear logic is the study of *resource reasoning* (we likely won’t very this much).

Kinds of Logic (Continued)

First-order and Higher-order logic. The study of *quantification*, or the ability to reason uniformly over an entire domain (like the statement ”every natural number is even or is odd”)

Extensions like modal logic and linear logic. Modal logic is the study of *necessity* and linear logic is the study of *resource reasoning* (we likely won’t very this much).

Expressivity, Complexity, Computation

The previous examples *increase in expressivity and complexity*.

Greater expressivity brings more complex meta-reasoning, and consequently, more complex reasoning *procedures*.

The ”mechanized” part of mechanized reasoning gets more difficult.

Expressivity, Complexity, Computation

The previous examples *increase in expressivity and complexity*.

Greater expressivity brings more complex meta-reasoning, and consequently, more complex reasoning *procedures*.

The ”mechanized” part of mechanized reasoning gets more difficult.

Expressivity, Complexity, Computation

The previous examples *increase in expressivity and complexity*.

Greater expressivity brings more complex meta-reasoning, and consequently, more complex reasoning *procedures*.

The ”mechanized” part of mechanized reasoning gets more difficult.

What about type theory?

We're hopefully somewhat familiar with types, e.g, in OCaml:

```
val square : int -> int  (* a function type *)
(*------------*)
let square x = x * x
```

History has shown that type theory is the "correct" language for expressing complex reasoning with computers (in particular, higher-order reasoning and extensions like linearity).

Note. Type theory is always changing, we're not learning *one* type theory (i.e., one programming language), we're learning a framework and analogy for discussing and analyzing formal reasoning.

What about type theory?

We're hopefully somewhat familiar with types, e.g, in OCaml:

```
val square : int -> int  (* a function type *)
(*------------*)
let square x = x * x
```

History has shown that type theory is the "correct" language for expressing complex reasoning with computers (in particular, higher-order reasoning and extensions like linearity).

Note. Type theory is always changing, we're not learning *one* type theory (i.e., one programming language), we're learning a framework and analogy for discussing and analyzing formal reasoning.

What about type theory?

We're hopefully somewhat familiar with types, e.g, in OCaml:

```
val square : int -> int  (* a function type *)
(*------------*)
let square x = x * x
```

History has shown that type theory is the ”correct” language for expressing complex reasoning with computers (in particular, higher-order reasoning and extensions like linearity).

Note. Type theory is always changing, we're not learning *one* type theory (i.e., one programming language), we're learning a framework and analogy for discussing and analyzing formal reasoning.

The Curry–Howard isomorphism

This will be the central concept of this course.
Roughly speaking, it says:

Type theories define logics.

We can use types to represent *statements* and *programs* to represent *proofs* of those statements (we'll see an example of this in the next slide). This idea, sometimes called propositions as types, underlies modern proof assistants like Lean, Coq, and Agda.

Note. This is a very philosophically fraught topic. People have different views about what the CH isomorphism really says. Part of this course will be deciding yourself.

The Curry–Howard isomorphism

This will be the central concept of this course.
Roughly speaking, it says:

Type theories define logics.

We can use types to represent *statements* and *programs* to represent *proofs* of those statements (we'll see an example of this in the next slide). This idea, sometimes called propositions as types, underlies modern proof assistants like Lean, Coq, and Agda.

Note. This is a very philosophically fraught topic. People have different views about what the CH isomorphism really says. Part of this course will be deciding yourself.

The Curry–Howard isomorphism

This will be the central concept of this course.
Roughly speaking, it says:

Type theories define logics.

We can use types to represent *statements* and *programs* to represent *proofs* of those statements (we'll see an example of this in the next slide). This idea, sometimes called propositions as types, underlies modern proof assistants like Lean, Coq, and Agda.

Note. This is a very philosophically fraught topic. People have different views about what the CH isomorphism really says. Part of this course will be deciding yourself.

Another Thought Experiment

Can you come up with OCaml functions with the following types?

```
val f : 'a -> a
val g : ('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)
```

Another Thought Experiment (Continued)

These types are said to be inhabited.

```
let f x = x
let g a b = fun x => a (b x)
```

But if we read "->" and "implies" and we think of 'a, 'b and 'c as *arbitrary mathematical statements*, then the types read as new reasonable mathematical statements:

```
val f : 'a -> 'a
(* "A implies A", or "if A is true then A is true" *)
val g : ('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)
(* "if A imp. B and B imp. C , then A imp. C " *)
```

Another Thought Experiment (Continued)

These types are said to be inhabited.

```
let f x = x
let g a b = fun x => a (b x)
```

But if we read ”->” and ”implies” and we think of ’a, ’b and ’c as *arbitrary mathematical statements*, then the types read as new reasonable mathematical statements:

```
val f : 'a -> 'a
(* "A implies A", or "if A is true then A is true" *)
val g : ('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)
(* "if A imp. B and B imp. C , then A imp. C " *)
```

The Point

We’ve reduced the problem of proving a statement to writing a program.

Outline

What is mechanized reasoning?

There are (roughly) two classes of tools for mechanized reasoning:

- automated theorem provers
- proof assistants

Automated theorem provers build proofs for you.

Proof assistants are tools for helping you build proofs that you've written so that they can be verified by a computer.

What is mechanized reasoning?

There are (roughly) two classes of tools for mechanized reasoning:

- automated theorem provers
- proof assistants

Automated theorem provers build proofs for you.

Proof assistants are tools for helping you build proofs that you've written so that they can be verified by a computer.

What is mechanized reasoning?

There are (roughly) two classes of tools for mechanized reasoning:

- automated theorem provers
- proof assistants

Automated theorem provers build proofs for you.

Proof assistants are tools for helping you build proofs that you've written so that they can be verified by a computer.

Proof verification vs Proof Construction

A key tenet of logic is that proofs should be easy to verify.

No matter how complex the reasoning, anyone with sufficient time and patience should be able to analyze a proof and determine if it is correct. This is verification.

This does not mean that it is easy to come up with the proof (It is easier to read a solution to math problem then to come up with it). This is the process of construction or automation.

Proof verification vs Proof Construction

A key tenet of logic is that proofs should be easy to verify.

No matter how complex the reasoning, anyone with sufficient time and patience should be able to analyze a proof and determine if it is correct. This is verification.

This does not mean that it is easy to come up with the proof (It is easier to read a solution to math problem then to come up with it). This is the process of construction or automation.

Proof verification vs Proof Construction

A key tenet of logic is that proofs should be easy to verify.

No matter how complex the reasoning, anyone with sufficient time and patience should be able to analyze a proof and determine if it is correct. This is verification.

This does not mean that it is easy to come up with the proof (It is easier to read a solution to math problem then to come up with it). This is the process of construction or automation.

Tools for this Course

SAT solvers. SATisfiability is a fundamental problem in complexity theory. It is NP-hard: if it has a fast algorithm, then a lot of difficult problems have fast algorithms. SAT solvers are the fastest known algorithms for SAT and they can be leveraged to solve difficult problems.

SMT solvers. Satisfiability Modulo Theories is SAT with predicates. They are also very fast have greater expressivity for better problem modeling.

Proof assistants. Proof assistants use type theory as a framework for expressing and proving in a mechanically verifiable way, serious theorems in mathematics and correctness properties of large-scale software systems.

Tools for this Course

SAT solvers. SATisfiability is a fundamental problem in complexity theory. It is NP-hard: if it has a fast algorithm, then a lot of difficult problems have fast algorithms. SAT solvers are the fastest known algorithms for SAT and they can be leveraged to solve difficult problems.

SMT solvers. Satisfiability Modulo Theories is SAT with predicates. They are also very fast have greater expressivity for better problem modeling.

Proof assistants. Proof assistants use type theory as a framework for expressing and proving in a mechanically verifiable way, serious theorems in mathematics and correctness properties of large-scale software systems.

Tools for this Course

SAT solvers. SATisfiability is a fundamental problem in complexity theory. It is NP-hard: if it has a fast algorithm, then a lot of difficult problems have fast algorithms. SAT solvers are the fastest known algorithms for SAT and they can be leveraged to solve difficult problems.

SMT solvers. Satisfiability Modulo Theories is SAT with predicates. They are also very fast have greater expressivity for better problem modeling.

Proof assistants. Proof assistants use type theory as a framework for expressing and proving in a mechanically verifiable way, serious theorems in mathematics and correctness properties of large-scale software systems.

Tools for this Course (Specifically)

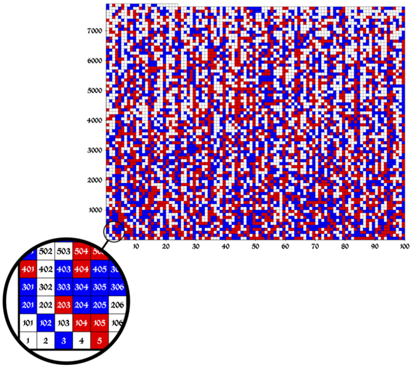
SAT solvers. Minisat via the python interface PySAT.

SMT solvers. Z3 via its python interface.

Proof assistants. Lean (and maybe Agda).

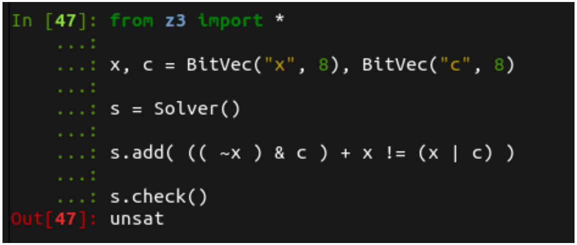
(this is all subject to change)

SAT Solvers in Practice



The Boolean Pythagorean triples problem was solved with a (massively parallel) SAT solver and produced a 200 terabyte large proof.

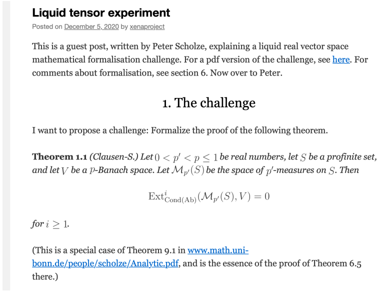
SMT Solvers in Practice



```
In [47]: from z3 import *
    ...:
    ...: x, c = BitVec("x", 8), BitVec("c", 8)
    ...:
    ...: s = Solver()
    ...:
    ...: s.add( (( ~x ) & c ) + x != (x | c) )
    ...:
    ...: s.check()
Out[47]: unsat
```

SMT solvers (like Z3) are used throughout industry for operations research, embedded systems, and critical systems like aviation.

Proof Assistants in Practices (Condensed Mathematics)


Liquid tensor experiment

Posted on December 5, 2020 by xenaproject

This is a guest post, written by Peter Scholze, explaining a liquid real vector space mathematical formalisation challenge. For a pdf version of the challenge, see here. For comments about formalisation, see section 6. Now over to Peter.

1. The challenge

I want to propose a challenge: Formalize the proof of the following theorem.

Theorem 1.1 *(Clausen-S.) Let $0 < p' < p \leq 1$ be real numbers, let S be a profinite set, and let V be a p-Banach space. Let $\mathcal{M}_{p'}(S)$ be the space of p'-measures on S. Then*

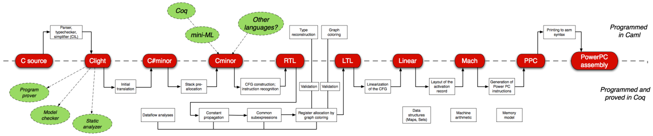
$$\mathrm{Ext}^i_{\mathrm{Cond(Ab)}}(\mathcal{M}_{p'}(S), V) = 0$$

for $i \geq 1$.

(This is a special case of Theorem 9.1 in www.math.uni-bonn.de/people/scholze/Analytic.pdf, and is the essence of the proof of Theorem 6.5 there.)

Lean was use to verify cutting-edge mathematics, allowing the mathematician Peter Scholze to know with (near) certainty that his reasoning was sound.

Proof Assistants in Practice (CompCert)



A different proof assistance (Coq) is being used to verify a C compiler, which in uncovered bugs in existing compilers.

Outline

Closing Remarks

There's a lot to cover in this course. The focus is on breadth over depth. We will skip a lot of details, and may miss of key ideas.

The final project is an opportunity to go deeper into the topic you find most interesting.

Its okay if the topics right now are foreign. Despite the emphasis on breadth we're going to start from the beginning. I want this course to be fairly exploratory.

Getting Set Up

If we have some time now, let’s try to get your system set up:

- Install VSCode if you haven’t already
- Download the Lean extension for VSCode
- Install Python and Pip if you haven’t already
- Use Pip to install `z3` and `python-sat`