Lab: CS 194/294-280: Advanced LLM Agents, Spring 2025

Background

The rise of large language models (LLMs) has led to a surge in the development of advanced coding assistants. Companies specializing in code generation, such as Cursor and Codeium, have become some of the fastest-growing in recent history. The power of these assistants has given rise to a phenomenon called "vibe coding," where users with little programming experience can build small to medium-sized applications simply by prompting an LLM.

However, despite this progress, a major challenge remains: hallucinations – subtle, hard-to-detect bugs that accumulate over time and can eventually break an entire codebase.

To address this issue, our lab focuses on **verifiable code generation**, ensuring that programs are not just functional but provably correct. Lean 4, a language built on dependent type theory, allows code to be accompanied by formal proofs, verifying correctness before execution.

Our goal is to leverage Lean 4's capabilities to develop a code-generation agent that writes solutions and generates proofs to verify their correctness. By integrating formal verification into the AI-driven coding process, we aim to reduce hallucinations and improve the reliability of AI-generated code.

Task Overview

This lab involves developing a code-generation agent that can generate solutions to algorithm-related coding problems in Lean 4. Alongside each function, the agent must also produce corresponding proofs to verify key properties of the generated code.

You will be provided with the following inputs:

- **Natural Language Problem Description (description.txt)** – Contains a description of the coding task, the inputs given into the program, and information about the outputs. This will be given to the LLM.
- **Signature Template (signature.json)** – This is a JSON file that specifies the function signature of the Lean 4 program you will be implementing. This will be given to the LLM.
- **Unit Tests (test.json)** – This is a JSON file that contains the unit tests that your code is expected to run/pass. This will **not be given to the LLM.**
- **Lean Code (task.lean)** – This is a skeleton of the code that your LLM coding agent will need to fill out. It contains the blank function implementation (`def` keyword) and the theorems, which correspond to some constraint/specification of the implementation function. This will be given to the LLM.

Your task is to build an agent that **replaces** the placeholders in both the function and theorem templates, producing correct and verifiable Lean 4 implementations.

Introduction to Lean 4

Lean 4 has a steep learning curve, but you don't need to be an expert to complete this lab. However, we recommend building a solid foundation by working through the following topics.

Step 1: Learn the basics of programming in Lean 4.

- Resource: Functional Programming in Lean. Read chapters 1, 2, 3, 9.

Checkpoint: At this stage, you should be able to:

- Write functions, involving pattern matching, conditionals, and recursion.
- Understand Lean's type system at a basic level.
- Understand fundamental syntax and semantics of functional programming in Lean 4.

Here are some code snippets of Lean 4 that you should be comfortable reading/understanding:

```
/-- Subtracts 1 from a natural number -/
def sub1 (n : Nat) : Nat :=
  match n with
  | 0 => 0
  | n+1 => n

/-- Doubles a natural number -/
def double (n : Nat) : Nat := n + n
```

```
/-- Checks if a natural number is zero -/
def isZero (n : Nat) : Bool :=
  match n with
  | 0 => true
  | _ => false

#eval sub1 10 -- 9
#eval double 11 -- 22
#eval isZero 0 -- true

/-- Returns greeting for a name -/
def greet (name : String) : String := "Hello, " ++ name ++ "!"

/-- Checks if a string is empty -/
def isEmpty (s : String) : Bool :=
  match s with
  | "" => true
  | _ => false

#eval greet "justin" -- "Hello, justin!"
#eval isEmpty "" -- true
```

Step 2: Learn the basics of proof writing in Lean

- Resource: Theorem Proving in Lean 4. Read chapters 1, 2, 3, 5, 8.

Checkpoint: At this stage, you should be able to:

- Write proofs in Lean 4 using tactic style (not term style).
- Use basic tactics to manipulate and structure proofs.
- Understand theorem statements, applying strategies such as induction and rewriting.

Here's an example of an induction style proof that just proves that n < n + 1 for all natural numbers n.

```
theorem nat_induction (n : Nat) : n < n + 1 := by
  induction n with
  | zero => simp -- Base case: 0 < 1
  | succ n ih => simp_all [Nat.succ_eq_add_one] -- Inductive step: n < n + 1 → n + 1 < n + 2
```

Step 3: Writing Proofs in Lean 4.

To deepen your understanding, we strongly recommend examining the public test suite and creating reference implementations with corresponding proofs for each problem. This will help debug your coding agent's implementation. **Pro Tip:** Leverage coding assistants (especially Reasoning Models, etc.) to help build a reference implementation for each question associated with the public test. Make sure your reference implementations compile in Lean 4.

Checkpoint: At this stage, you will be able to:

- Understand the test suite.
- Understand what constitutes a correct function implementation and what it means to have a verifiably accurate function implementation via theorems.
- Apply inductive reasoning and basic tactics (casework, rewriting, simplification, induction) to prove function specifications confidently.

Additional Resources:

- All Tactics in Lean 4 [link]
  - **Most Relevant Tactics:** rfl, rw [h], simp, intro/intros, exact, apply, cases, induction, exfalso, contradiction.
- Lean 4, Natural Numbers Proof Game [link]
  - This is an interactive resource to learn the basics of proving, in the context of number theory.
  - **Warning:** Not all examples are relevant. Look at the complexity of the test cases to get a better sense of the difficulty level we require (our test-suite is less challenging than what you may encounter online).

Deliverable 1: Contribute to a Dataset

Group Size: You are allowed to do this task in groups containing up to two members. Each group must submit a total of 2 data points, regardless of group size.

Now that you have a better understanding of Lean 4, we invite you to contribute to an ongoing dataset for verifiable code generation. This dataset consists of programming problems paired with formal specifications, reference implementations, and unit test cases.

Your Task

Create a programming task in Lean 4 and provide the reference implementation and specification for it. Your submission will consist of four files that define a programming problem and its solution using the following template.

File 1: `description.txt`

Requirement: Write a detailed task description of the program, including the overall method functionality and the input/output relationship.

Example:

```
-----Description-----
This task requires writing a Lean 4 method that finds the minimum among three given integers. The method should retu

-----Input-----
The input consists of three integers:
a: The first integer.
b: The second integer.
c: The third integer.

-----Output-----
The output is an integer:
Returns the minimum of the three input numbers, assuring that the returned value is less than or equal to a, b, and
```

File 2: `signature.json`

Requirement: Write the definition of the function signature.

Example:

```
{
    "name": "minOfThree",
    "parameters": [
      {
        "param_name": "a",
        "param_type": "Int"
      },
      {
        "param_name": "b",
        "param_type": "Int"
      },
      {
        "param_name": "c",
        "param_type": "Int"
      }
    ],
    "return_type": "Int"
}
```

File 3: `test.json`

Requirement: Write unit tests for the function to be implemented. `unexpected` field should contain an array of values that should not be returned by the function. You are required to write at least **5** unit tests, each of which should at least contain one `unexpected` value. Your test cases should thoroughly cover all possible execution paths and include edge cases.

Example:

```
[
    {
      "input": { "a": 3, "b": 2, "c": 1 },
      "expected": 1,
      "unexpected": [2, 3, -1]
    },
    {
      "input": { "a": 5, "b": 5, "c": 5 },
      "expected": 5,
      "unexpected": [-1]
    },
```

```
    {
      "input": { "a": 10, "b": 20, "c": 15 },
      "expected": 10,
      "unexpected": [5]
    },
    {
      "input": { "a": -1, "b": 2, "c": 3 },
      "expected": -1,
      "unexpected": [2]
    },
    {
      "input": { "a": 2, "b": -3, "c": 4 },
      "expected": -3,
      "unexpected": [4]
    },
    {
      "input": { "a": 2, "b": 3, "c": -5 },
      "expected": -5,
      "unexpected": [2]
    },
    {
      "input": { "a": 0, "b": 0, "c": 1 },
      "expected": 0,
      "unexpected": [1]
    },
    {
      "input": { "a": 0, "b": -1, "c": 1 },
      "expected": -1,
      "unexpected": [0]
    },
    {
      "input": { "a": -5, "b": -2, "c": -3 },
      "expected": -5,
      "unexpected": [1, -2]
    }
]
```

File 4: `task.lean`

Requirements:

1. Update the template signature part as your defined signature
2. Fill in and , you can write multiple specifications for one implementation
3. DO NOT REMOVE comments like `-- << CODE START >>`, these labels will be used to extract your submission content.
4. Please stick to the template, separate each variable in the signature even if they share the same type. For example, do not write `(a b : Int)`, please write `(a : Int) (b : Int)` instead
5. When creating the specification, please try your best to provide a complete specification that fully constrains the program behavior, capturing all essential properties and invariants that your implementation must satisfy.

Template*:*

```
def task_code (input1 : type1) (input2 : type2) (...) : type_out :=
  -- << CODE START >>

  -- << CODE END >>

def task_sepc_1 (input1 : type1) (input2 : type2) (...) (result : type_out) : Prop :=
  -- << SPEC START >>

  -- << SPEC END >>
```

Example*:*

```
def minOfThree (a : Int) (b : Int) (c : Int) : Int :=
  -- << CODE START >>
```

```
  if a <= b && a <= c then a
  else if b <= a && b <= c then b
  else c
  -- << CODE END >>

def minOfThree_spec_isMin (a : Int) (b : Int) (c : Int) (result : Int) : Prop :=
  -- << SPEC START >>
  (result <= a ∧ result <= b ∧ result <= c) ∧ (result = a ∨ result = b ∨ result = c)
  -- << SPEC END >>

-- You can use the following to do unit tests, you don't need to submit the following code

#guard minOfThree 1 2 3 = 1
```

Guidelines

1. The requirement is only submitting 2 problems. We outline an extra credit opportunity that involves submitting additional problems in the next section.
2. Your problem must be non-trivial. We will require your fully implemented program to be >= 30 lines of code (may be subject to change). We will verify the quality of all student data point submissions.
3. You can create the programming tasks based on the following datasets, or use your own idea. Please note that you don't need to create overly complex questions - simple but well-defined problems work best. You can draw inspiration from common programming benchmarks like LiveCodeBench, APPS, MBPP, or coding platforms like LeetCode, HackerRank, or CodeForces for suitable task ideas.

Extra Credit Opportunity

To qualify for extra credit, you must submit an additional 2 data points for each member of your group. For instance, if you are a single person in a group, submit 2 additional data points to qualify for extra credit, and if you have two members, submit 4 additional data points. Each member of the group may earn **up to 20% extra credit points** on the assignment.

The guidelines are the same as the previous – we will **only** accept high quality submissions and will be less lenient on problem quality for extra credit submissions compared to the required single problem submitted in the previous section.

Deliverable 2: Peer Review

To ensure the quality of the datapoint submitted, after the submission deadline, we will assign 4 data points from other groups to you to help do peer reviews. We will provide the following three files for each datapoint:

- `description.txt`
- `signature.json`
- `task.lean`

Your Task

1. Write 5 unit tests for each datapoint being peer reviewed. We will be verifying all peer reviews to ensure they are consistent and accurate. If your peer review is significantly different than what we deem the quality of the data point to be, you may experience point deductions.
2. Carefully read each datapoint description and lean file, give an assessment of the quality of each datapoint using the following format:

```
{
        "description": {
                "is_trivial": true / false,
                "has_sufficient_details": "low" / "medium" / "high"
        },
        "code": {
                "is_correct": true / false,
                "quality": "low" / "medium" / high"
        },
        "spec": {
                "is_correct": true / false,
                "quality": "low" / "medium" / high",
                "match_full_intent_of_description": "low" / "medium" / high"
        }
}
```