Project: Theorem Proving in Lean and Type Safety

Due date: December 1 (Wednesday) at 11:59pm

Overview

This project consists of three parts:

1. Prove the *progress property* of a simple language on natural numbers and formalize the proof in Lean.
2. Prove the *totality property* of the language and formalize the proof in Lean.
3. Prove the *type preservation property* of an extension of the language and formalize the proof in Lean.

Important notes:

- **You will need to give yourself enough time to learn Lean.** Lean is the language you will use to complete this project, and you will probably be unfamiliar with it. There are good materials for learning Lean, but to take advantage of that, you will need to take enough time. Remember you will have more time on this project than on homework assignments; use the longer period of time wisely.
- **There will be no partial credit for each problem.** For each problem, you will get full credit if your proof typechecks, and get no credit otherwise.
- **Your final proofs should not contain any `sorry` expressions.** If your solution file `probN.lean` contains any `sorry` expressions, you will get no credit for Problem N and all the problems whose Lean file imports `probN.lean`.
- You should prove a statement by hand first, before writing any proofs in Lean. Writing a proof in Lean without having a hand-written proof will be quite difficult for later problems.
- You are optionally allowed to work with a partner for this project. One member of each pair should submit the project on Canvas (instructions at the bottom).

Lean

In this project, you will prove a few properties of two simple languages and "formalize" the proofs in the Lean language. Here "formalization" of a proof means writing a proof in some programming language so that the validity of the proof can be checked mechanically by the type system of the language. We choose to use Lean because it has very good tutorials and documentation.

You should install Lean 3.4.2; no other version is allowed. We recommend you to install it on your local machine, as you need to use an interactive editor (VSCode or Emacs) to complete the project. More details on installation:

- Download the **binary of Lean 3.4.2** at `https://leanprover.github.io/download/` and untar it at a directory, say `$HOME/abc/` (i.e., you can run `$HOME/abc/bin/lean --version`). Add the path `$HOME/abc/bin` to the PATH environment variable (e.g., add `export PATH="$HOME/abc/bin:$PATH"` to `$HOME/.profile` or `$HOME/.bashrc`). After this, running `lean --version` should output `Lean (version 3.4.2. ...)`.

- If you are using the Darwin binaries (macOS) and run into this error:

```
dyld:  Library not loaded:  /usr/local/opt/gmp/lib/libgmp.10.dylib
```

  then you need to install `gmp`, which can be installed via Homebrew with `brew install gmp`. If you don't have Homebrew, follow its installation instructions on `https://brew.sh/`.

- Install VSCode or Emacs and integrate it with Lean. For details, see `https://leanprover.github.io/reference/using_lean.html#using-lean-with-vscode` for VSCode, and `https://github.com/leanprover/lean-mode` for Emacs.

You can learn a few important concepts behind Lean (e.g., propositions-as-types) and the basics of Lean by reading the Lean tutorial ("Theorem Proving in Lean"): `https://leanprover.github.io/theorem_proving_in_lean/index.html`. To complete this project, you will need to read at most (not at least) the following sections from the tutorial: 2.1–2.4, 2.8, 3.1–3.4, 4.1–4.2, 4.4, 5.1–5.7, and 7.1–7.7. More suggestions:

- You should not try to understand everything in the Lean tutorial. The goal of this project is not to master Lean, but to formalize several proofs in Lean in a limited amount of time. So it is completely fine even if you do not understand some parts of the tutorial, as long as you can complete this project.
- To give you an idea of which tactics may be necessary to complete this project, we list the tactics used in the reference solution as follows: `intros`, `revert`, `show`, `have`, `exact`, `assumption`, `apply`, `split`, `left`, `right`, `existsi`, `cases`, `cases ... with ...`, `induction`, `case ... : ... {...}`, `injections`, `reflexivity`, `symmetry`, `transitivity`, and `simp *`.
- You should be familiar enough with Lean to solve the following problems. For this purpose, we recommend you to prove at least the propositions in `exercises.lean`, before starting Part 1.

Type Safety

Consider a language L equipped with expressions e, an evaluation judgement $e \mapsto e'$, and a typing judgement $\Gamma \vdash e : \tau$. Here the judgement $e \mapsto e'$ denotes that an expression e evaluates to another expression e', and the judgment $\Gamma \vdash e : \tau$ denotes that an expression e has type τ under the typing context Γ. That is, $e \mapsto e'$ defines the operational semantics of L and $\Gamma \vdash e : \tau$ defines the type system of L. Note that $e \mapsto e'$ is assumed to be non-reflexive (i.e., $e \mapsto e$ does not hold for all e). We say that the language L satisfies type safety iff it satisfies the following two properties which connect the operational semantics with the type system.

Theorem (Progress)**.** If $\cdot \vdash e : \tau$, then either e is a value of L or there exists e' such that $e \mapsto e'$.

Theorem (Type Preservation)**.** If $\Gamma \vdash e : \tau$ and $e \mapsto e'$, then $\Gamma \vdash e' : \tau$.

The progress property states that a well-typed expression never gets stuck (i.e., either the expression is already a value or it evaluates to another expression), where $\cdot$ denotes an empty typing context. Note that the progress property does not necessarily imply the termination of an expression e: even with the property, it is possible to have an infinite sequence of evaluation such as $(e \mapsto e_1) \wedge (e_1 \mapsto e_2) \wedge (e_2 \mapsto e_3) \wedge \cdots$. The type preservation property states that the type of an expression remains the same during evaluation. Type safety is one of the most basic properties of a typed language, and you will prove type safety of two simple languages using Lean.

Part 1: Progress

In the first and second parts of this project, we consider a simple language $\mathrm{L}_{\mathsf{nat}}$ on natural numbers. First, the abstract syntax of $\mathrm{L}_{\mathsf{nat}}$ is defined as follows.

$$e \ ::= \ n \ | \ e \circledast e$$

Here e denotes an expression in $\mathrm{L}_{\mathsf{nat}}$, $n \in \mathbb{N}$ denotes a natural number (including 0), and $\circledast \in \{\oplus, \ominus, \otimes, \oslash\}$ denotes an operation supported by $\mathrm{L}_{\mathsf{nat}}$. Second, the operational semantics of $\mathrm{L}_{\mathsf{nat}}$ (i.e., how an expression evaluates to another expression) is defined as follows.

$$\frac{e_1 \mapsto e_1'}{e_1 \circledast e_2 \mapsto e_1' \circledast e_2}\ \text{E-Left} \qquad \frac{e_1\ \mathsf{val} \qquad e_2 \mapsto e_2'}{e_1 \circledast e_2 \mapsto e_1 \circledast e_2'}\ \text{E-Right} \qquad \frac{}{n_1 \circledast n_2 \mapsto n_1 * n_2}\ \text{E-Op}$$

Here $*$ in the rule E-Op denotes the mathematical operation corresponding to $\circledast$, and the judgment e val in the rule E-Right is defined as follows.

$$\frac{}{n\ \mathsf{val}}\ \text{V-Num}$$

The judgement e val denotes that an expression e is a value in $\mathrm{L}_{\mathsf{nat}}$, and its single inference rule states that natural numbers are the only values in $\mathrm{L}_{\mathsf{nat}}$. The inference rules of $e \mapsto e'$ fixes the order of evaluation: to evaluate $e_1 \circledast e_2$, we should first evaluate e_1 to some value, then evaluate e_2 to some other value, and finally evaluate an operation on two obtained values. Third, the type system of $\mathrm{L}_{\mathsf{nat}}$ is very simple: every expression is well-typed and has a single type nat. We omit the inference rules of $\Gamma \vdash e : \mathsf{nat}$ as they are trivial.

The language $\mathrm{L}_{\mathsf{nat}}$ satisfies the progress and type preservation properties. The type preservation property is straightforward to prove, as all expressions have a single type. On the other hand, the progress property is not as obvious to prove as the type preservation property, because not every expression evaluates to another expression. The goal of this part is to prove the progress property of $\mathrm{L}_{\mathsf{nat}}$ (stated in the following theorem) and formalize the proof in Lean. You can find the formalization of $\mathrm{L}_{\mathsf{nat}}$ in `src/lnat.lean`.

Theorem (Progress)**.** For any expression e, either e val holds or there exists e' such that $e \mapsto e'$.

Problem 1: Inversion

To prove the progress theorem, you need to prove the following inversion lemma, which states that every value in $\mathrm{L}_{\mathsf{nat}}$ is a natural number. **Your task is to prove the lemma in `prob01.lean`.** As you may expect, the proof is short and indeed the reference solution is 4 lines.

Lemma (Inversion)**.** If e val, then there exists $n \in \mathbb{N}$ such that $e = n$.

Problem 2: Example for Progress

Before proving the progress theorem, let's get used to Lean and how $\mathrm{L}_{\mathsf{nat}}$ is encoded in Lean. **For this purpose, your task is to find an expression e_1 that satisfies the following two conditions and to prove $e_1 \mapsto e_2$ in `prob02.lean`.** The reference solution is 12 lines.

- Condition 1: There exists an expression e_2 such that $e_1 \mapsto e_2$.
- Condition 2: The proof of $e_1 \mapsto e_2$ uses all the three inference rules for $e \mapsto e'$.

To allow us to automatically check the condition 2, **all three terms `eval.ELeft`, `eval.ERight`, and `eval.EOp` should appear in `prob02.lean`.**

Problem 3: Progress

You are now ready to prove the progress theorem. **Your task is to prove the theorem in `prob03.lean`. For this and later problems, you will need to write proofs in tactic mode (not in proof term mode)**; otherwise, it will be very difficult to complete the proofs. The reference solution is 33 lines. Here are some tips:

- A proof in Lean will resemble the corresponding hand-written proof. Your proof of the progress theorem will start by introducing terms (`intros`), inducting on the variable of interest (`induction e`), then proceeding by cases (`case Expr.Num :  n { ...  }`).

- As you develop your proof, continually use the proof context displayed in the message window of VSCode or Emacs. Always be clear on what the current goal is and what facts are in scope.
- Use the `show` tactic to change the goal, and the `have` tactic to prove a new fact without changing the goal. Use the `exact` tactic to provide a term that has the type of the goal, and the `assumption` tactic to find a term in the proof context for the goal.
- Use the powerful `simp` tactic, whenever possible, which can reduce your work if it works. For example, `simp *` attempts to apply every fact in the context to prove the current goal.

Part 2: Totality

The language L$_{\mathsf{nat}}$ enjoys another nice property called totality. The totality property states that any expression e evaluates to some value in a finite number of steps. To formalize the property, we define the transitive closure of the evaluation relation $e \mapsto e'$ as follows.

$$\frac{}{e \mapsto^* e}\ \text{C-Refl} \qquad\qquad \frac{e \mapsto e' \qquad e' \mapsto^* e''}{e \mapsto^* e''}\ \text{C-Step}$$

The judgement $e \mapsto^* e'$ states that e evaluates to e' in a finite number of steps (including no steps). Using the new judgement, the totality property of L$_{\mathsf{nat}}$ can be expressed by the following theorem. The goal of this part is to prove the theorem and formalize the proof in Lean.

Theorem (Totality)**.** For any expression e, there exists e' such that e' val and $e \mapsto^* e'$.

Problems 4-5: Left- and Right-Transitivity

To prove the totality theorem, you need to prove the following two lemmas on the transitivity of $e \mapsto^* e'$. **Your task is to prove the two lemmas in `prob{04,05}.lean`.** The reference solution is 9 lines for each problem.

Lemma (Left-Transitivity)**.** If $e_1 \mapsto^* e_1'$, then $(e_1 \circledast e_2) \mapsto^* (e_1' \circledast e_2)$ for any expression e_2.

Lemma (Right-Transitivity)**.** If e_1 val and $e_2 \mapsto^* e_2'$, then $(e_1 \circledast e_2) \mapsto^* (e_1 \circledast e_2')$ for any expression e_1.

The reflexivity ($e \mapsto^* e$) and the transitivity ($e \mapsto^* e' \wedge e' \mapsto^* e'' \implies e \mapsto^* e''$) of $e \mapsto^* e'$ are already proven in `lnat.lean`. For this and later problems, you can apply the two properties by simply using the tactics `reflexivity` and `transitivity`.

Problem 6: Totality

You are now ready to prove the totality theorem. **Your task is to prove the theorem in `prob06.lean`.** The proof will proceed by induction on e. The reference solution is 31 lines.

Problem 7: Example for Totality

One way to better understand a theorem is to think about an instance of the theorem. Let's do that exercise for the totality theorem. **Your task is to find an expression e_1 that satisfies the following two conditions and to prove e_2 val and $e_1 \mapsto^* e_2$ in `prob07.lean`.** The reference solution is 18 lines.

- Condition 1: There exists an expression e_2 such that e_2 val and $e_1 \mapsto^* e_2$.
- Condition 2: The proof of $e_1 \mapsto^* e_2$ uses the rule C-Step **at least three times**.

To allow us to automatically check the condition 2, **the term `evals.CStep` should appear at least three times in `prob07.lean`.**

Part 3: Type Preservation

In the third part of this project, we focus on a new language $\mathrm{L}^{+}_{\mathsf{nat}}$ which extends the previous language $\mathrm{L}_{\mathsf{nat}}$ by adding constructs for variables and let bindings. The language $\mathrm{L}^{+}_{\mathsf{nat}}$ is defined as follows.

First, the abstract syntax of $\mathrm{L}^{+}_{\mathsf{nat}}$ is extended from that of $\mathrm{L}_{\mathsf{nat}}$ as follows.

$$e \ ::= \ \cdots \ | \ \widehat{i} \ | \ \mathsf{let}\ e\ \mathsf{in}\ e$$

Here $\widehat{i}$ denotes a variable and $\mathsf{let}\ e_1\ \mathsf{in}\ e_2$ denotes a let binding, where $i \in \mathbb{N}$ denotes a natural number. We put a hat over i to differentiate an expression denoting a variable from an expression denoting a natural number. The representation of variables in $\mathrm{L}^{+}_{\mathsf{nat}}$ is a bit different from that in the calculi we have seen in the lectures: variables in $\mathrm{L}^{+}_{\mathsf{nat}}$ cannot have arbitrary names, rather they are represented by particular natural numbers that are obtained from a rule to be explained below. We choose this unusual representation of variables, called *de Bruijn indices*, because the representation makes it easier to formalize in Lean the language $\mathrm{L}^{+}_{\mathsf{nat}}$ itself and the proofs of its properties.

To understand the semantics of variables and let bindings in $\mathrm{L}^{+}_{\mathsf{nat}}$, consider the following expression e.

$$5 \oplus \Big(\mathsf{let}\ 7\ \mathsf{in}\ \Big(\widehat{0} \otimes (\mathsf{let}\ 9\ \mathsf{in}\ (\widehat{0} \ominus \widehat{1}))\Big)\Big)$$

This expression can be translated into a language (not $\mathrm{L}^{+}_{\mathsf{nat}}$) with named let binding as follows.

$$5 \oplus \Big(\mathsf{let}\ x = 7\ \mathsf{in}\ \Big(x \otimes (\mathsf{let}\ y = 9\ \mathsf{in}\ (y \ominus x))\Big)\Big)$$

Here is a rule on how to interpret variables and (nameless) let bindings in $\mathrm{L}^{+}_{\mathsf{nat}}$: for a given variable reference $\widehat{i}$, its enclosing let bindings are numbered from $0, 1, \cdots$ (the closest enclosing binding is the 0th, the next closest is the 1st, and so on); then, this particular $\widehat{i}$ will take its value from the ith enclosing let binding. Let's see how this rule applies to the above expression e. First, 7 will be substituted for the first occurrence of $\widehat{0}$, as 7 appears in the let part of the 0th enclosing let binding (i.e., the closest let binding that encloses the first $\widehat{0}$). Similarly, 9 will be substituted for the second occurrence of $\widehat{0}$. More interestingly, 7 will be substituted for $\widehat{1}$, as 7 appears in the let part of the 1st enclosing let binding (i.e., the second closest let binding that encloses $\widehat{1}$)—remember that the numbering of enclosing let bindings starts from 0. As a result, the above expression e evaluates to $5 + (7 \times (9 - 7)) = 19$.

Based on this rule on how variables and let bindings evaluate, the operational semantics of $\mathrm{L}^{+}_{\mathsf{nat}}$ is defined by adding the following inference rule to the operational semantics of $\mathrm{L}_{\mathsf{nat}}$.

$$\frac{}{\mathsf{let}\ e_1\ \mathsf{in}\ e_2 \mapsto e_2\,[\widehat{0} := e_1]}\ \text{E-Let}$$

Here the substitution operator $e\,[\widehat{i} := e']$ is defined as follows.

$$\begin{aligned}
n\,[\widehat{i} := e'] &= n \\
\Big(e_1 \circledast e_2\Big)\,[\widehat{i} := e'] &= \Big(e_1\,[\widehat{i} := e']\Big) \circledast \Big(e_2\,[\widehat{i} := e']\Big) \\
\widehat{j}\,[\widehat{i} := e'] &= \begin{cases} e' & \text{if } j = i \\ \widehat{j} & \text{if } j \neq i \end{cases} \\
\Big(\mathsf{let}\ e_1\ \mathsf{in}\ e_2\Big)\,[\widehat{i} := e'] &= \mathsf{let}\ \Big(e_1\,[\widehat{i} := e']\Big)\ \mathsf{in}\ \Big(e_2\,[\widehat{i+1} := e']\Big)
\end{aligned}$$

The operator $e\,[\widehat{i} := e']$ substitutes the expression e' for all those variables in e that "correspond" to $\widehat{i}$. Note that the in part of $(\mathsf{let}\ e_1\ \mathsf{in}\ e_2)\,[\widehat{i} := e']$ is defined by $\mathsf{let} \cdots \mathsf{in}\ (e_2\,[\widehat{i+1} := e'])$, not by $\mathsf{let} \cdots \mathsf{in}\ (e_2\,[\widehat{i} := e'])$. This reflects the aforementioned rule on how variables in $\mathrm{L}^{+}_{\mathsf{nat}}$ behave. Using the substitution operator, the rule E-Let states that $\mathsf{let}\ e_1\ \mathsf{in}\ e_2$ evaluates to the expression obtained by substituting e_1 for $\widehat{0}$ in e_2.

Finally, the type system of $\mathrm{L}^{+}_{\mathsf{nat}}$ is defined as follows.

$$\frac{}{\Gamma \vdash n : \mathsf{nat}}\ \text{T-Num} \qquad\qquad \frac{\Gamma \vdash e_1 : \mathsf{nat} \qquad \Gamma \vdash e_2 : \mathsf{nat}}{\Gamma \vdash e_1 \circledast e_2 : \mathsf{nat}}\ \text{T-Op}$$

$$\frac{i < \Gamma}{\Gamma \vdash \widehat{i} : \mathsf{nat}}\ \text{T-Var} \qquad\qquad \frac{\Gamma \vdash e_1 : \mathsf{nat} \qquad \Gamma + 1 \vdash e_2 : \mathsf{nat}}{\Gamma \vdash \mathsf{let}\ e_1\ \mathsf{in}\ e_2 : \mathsf{nat}}\ \text{T-Let}$$

In the inference rules, a typing context Γ is represented as a natural number (i.e., $\Gamma \in \mathbb{N}$) with the following interpretation: $\Gamma = i$ (for some $i \in \mathbb{N}$) denotes the typing context $\widehat{0} : \mathsf{nat}, \widehat{1} : \mathsf{nat}, \cdots, \widehat{i-1} : \mathsf{nat}$. Given this interpretation of Γ, the part $i < \Gamma$ in the rule T-Var denotes that $\widehat{i} : \mathsf{nat}$ is in Γ, and the part $\Gamma + 1$ in the rule T-Let denotes the typing context $\widehat{0} : \mathsf{nat}, \cdots, \widehat{i} : \mathsf{nat}$, where $\Gamma = i$ for some $i \in \mathbb{N}$. Here is an example demonstrating how the typing rules work.

$$\frac{\dfrac{}{0 \vdash 7 : \mathsf{nat}}\ \text{T-Num} \qquad \dfrac{\dfrac{}{1 \vdash 9 : \mathsf{nat}}\ \text{T-Num} \qquad \dfrac{0 < 1}{1 \vdash \widehat{0} : \mathsf{nat}}\ \text{T-Var}}{1 \vdash 9 \oplus \widehat{0} : \mathsf{nat}}\ \text{T-Op}}{0 \vdash \mathsf{let}\ 7\ \mathsf{in}\ (9 \oplus \widehat{0}) : \mathsf{nat}}\ \text{T-Let}$$

Note that $\mathrm{L}^{+}_{\mathsf{nat}}$ has a single type `nat` as $\mathrm{L}_{\mathsf{nat}}$ does, but not every expression of $\mathrm{L}^{+}_{\mathsf{nat}}$ is well-typed unlike $\mathrm{L}_{\mathsf{nat}}$.

The language $\mathrm{L}^{+}_{\mathsf{nat}}$ satisfies the progress and type preservation properties as in $\mathrm{L}_{\mathsf{nat}}$. The goal of this part is to prove the type preservation property of $\mathrm{L}^{+}_{\mathsf{nat}}$ (stated in the following theorem) and formalize the proof in Lean. Note that in the theorem, 0 denotes an empty typing context. You can find the formalization of $\mathrm{L}^{+}_{\mathsf{nat}}$ in `src/lnatplus.lean`.

Theorem (Type Preservation)**.** If $0 \vdash e : \mathsf{nat}$ and $e \mapsto e'$, then $0 \vdash e' : \mathsf{nat}$.

Problem 8: Substitution

To prove the type preservation theorem, you need to prove the following substitution lemma, which states that a well-type expression remains well-typed after substitution under some conditions. **Your task is to prove the lemma in `prob08.lean`.** The reference solution is 89 lines in total.

Lemma (Substitution)**.** If $i \vdash e : \mathsf{nat}$ and $i + 1 \vdash e' : \mathsf{nat}$, then $i \vdash e' \left[\widehat{i} := e\right] : \mathsf{nat}$.

You may find it useful to define and prove one or more auxiliary lemmas, and use them in the course of proving the substitution lemma.

For this and the following problems, you will need to use a few basic properties of natural numbers (e.g., $\forall n \in \mathbb{N}. \neg(n < n)$). These properties can be found at `https://leanprover-community.github.io/mathlib_docs/init/data/nat/basic.html` and `https://leanprover-community.github.io/mathlib_docs/init/data/nat/lemmas.html`. The reference solution for Problems 8-11 makes use of the following theorems or lemmas listed in the above links: `nat.le_of_lt_succ`, `nat.le_succ_of_le`, `nat.lt_of_le_and_ne`, `nat.lt_of_lt_of_le`, `nat.lt_irrefl`, `nat.lt_succ_self`, and `nat.succ_le_succ`.

Problem 9: Type Preservation

You are now ready to prove the type preservation theorem. **Your task is to prove the theorem in `prob09.lean`.** The reference solution is 24 lines.

Problem 10: Non-Example for Type Preservation

One other way to better understand a theorem is to think about why each condition of the theorem is necessary. Let's do that exercise for the type preservation theorem. **Your task is to find expressions e_1 and e_2 that satisfy the following condition, and to prove in `prob10.lean` that e_1 and e_2 indeed satisfy the condition.** The reference solution is 18 lines.

- Condition: $e_1 \mapsto e_2$ holds, but $0 \vdash e_1 : \mathsf{nat}$ does not hold, thereby $0 \vdash e_2 : \mathsf{nat}$ does not hold.

Problem 11: Counterexample for Generalized Type Preservation

Another way to better understand a theorem is to think about whether the theorem can be generalized. Let's do that exercise for the type preservation theorem. You might guess that the theorem could be generalized as follows.

Theorem? For any $i \in \mathbb{N}$, if $i \vdash e : \texttt{nat}$ and $e \mapsto e'$, then $i \vdash e' : \texttt{nat}$.

However, it turns out that the above generalization is invalid and your task is to figure out why it is invalid. **Precisely, you need to find a natural number i and expressions e_1 and e_2 that satisfy the following condition, and to prove in `prob11.lean` that they indeed satisfy the condition.** The reference solution is 17 lines.

- Condition: $i \vdash e_1 : \texttt{nat}$ and $e_1 \mapsto e_2$ hold, but $i \vdash e_2 : \texttt{nat}$ does not hold.

Submission

- Regardless of whether you are working in a pair, navigate to Canvas → People, then click on the "proj-lean-groups" tab and sign up for a proj-lean group. If you are working with a partner, make sure you both sign up for the same group.
- Make sure you have used Lean 3.4.2: `lean --version` should output `Lean (version 3.4.2. ...)`.
- **Make sure `lean probN.lean --trust=0` gives no error messages for each Problem N you solved.**
- Edit `README.txt` to include your student ID number (the last 8-digit number) and the student ID number of your partner (if you are working in a pair).
- Generate `solution-lean.tar.gz` by running `python3 src/submit.py`, and upload the tarball file to Canvas.
- Make sure the script gives no errors or warnings.
- Make sure the tarball file consists precisely of `prob`$\{$`01,`$\cdots$`,11`$\}$`.lean` and `README.txt`.