3. Motivation

3.1. A Toy Example

We envision the workflow of someone using LFORGE to be as follows:

1. The user specifies a model in interest, or they already have a completed model in Forge that they would like to formalize. Within Forge, the user writes relevant predicates and uses Forge's SAT backend to quickly check if they are satisfiable (and what satisfying instances look like), or if they are not satisfiable (that is, the negation is a theorem). Users might isolate specific predicates that they would like to prove in detail.

2. In a new Lean file, the user imports the LFORGE module and pastes their Forge specification in, verbatim. If they started the process working in the LFORGE subset of Forge, no changes need to be made. If they are working with an existing Forge file, our tool will prompt the user to make any modifications necessary to keep it compliant with the subset of syntax, including potential type annotations (see section 5.4). After this, all sigs, fields, predicates are available in Lean.

3. Optionally, a user might want to make *additional* claims or write predicates using Lean's syntax. They have the option to do so here (see Mix-ins, section 6.1).

4. Finally, the user will identify important predicates within their specification that they want to prove generally. They do so using the interactive theorem prover in Lean.

We start by providing a small example that might represent a Forge specification and the desired generated code in Lean as motivation behind this workflow. Bertrand Russell, in illustrating Russel's paradox on sets, poses the following paradox: "Let the barber be 'one who shaves all those, and those only, who do not shave themselves.' The question is, does the barber shave himself?" [9, p. 101].

We might want to construct a formal model for this village for which the barber shaves all those who do not shave themselves, and use Forge to prototype and quickly check properties regarding this model:

```
sig Person {
  shaver: one Person
}

pred shavesThemselves[p: Person] {
  p = p.shaver
}

pred existsBarber {
  some barber : Person | all p : Person | {
    not shavesThemselves[p] <=> p.shaver = barber
  }
}
```

Attempting to run this model for `existsBarber` will yield an `Unsatisfiable`. After all, if the barber doesn't shave themselves, yet they shave all those who don't, they they ought to shave themselves. To prove this statement (and not merely rely on the fact that Forge was not able to find a satisfiable instance within its bounds), we need to transition to a theorem prover.

LFORGE aids in porting the entire specification into Lean, producing a set of equivalent Lean definitions:

```
opaque Person : Type
opaque shaver : Person → Person

def shavesThemselves : Person → Prop :=
  fun p ↦ shaver p p

def existsBarber : Prop :=
  ∃ (barber : Person), ∀ (p : Person), ¬shavesThemselves p ↔ shaver p barber
```

At this point, we might want to continue to provide a mathematical proof, as we observed in Forge, that there cannot possibly be a barber in this town. That is,

```
theorem no_barber : ¬ existsBarber := by ...
```

The conclusion of this example, including the Lean proof of our property, is provided in section 6.2. Note that this was not possible working solely in Forge, which only checks for the existence of examples or counterexamples within the bounds that it knows.

While this was a simple example, it serves as a demonstration of what is possible under this dual framework. One might provide a specification of a protocol and prove that no vulnerabilities exist (or that it has all the desired properties). Lean would serve invaluable in proving the correctness of properties about any real-life system described and prototyped in the Forge specification language. All this stays true to the goal of providing better formal methods tools that are more universally applicable and practical.

Furthermore, the pedagogical implications of such a program is also worth noting. Forge, designed to be a pedagogical language, seeks to teach formal methods gradually and introduce students to formal methods tools enabling them to work better in the real world and industry [7]. Students who are interested in formal methods often take Logic for Systems in the Computer Science department at Brown, the course which Forge was developed for and taught in. Students continue on to take Formal Proof and Verification, a course that teaches the use of proof assistants via Lean. Both courses have open-ended research-style final projects that encourages students to explore and formalize topics that they are interested in. We believe LFORGE also provides an invaluable opportunity for students who have background with both formal methods tools (or who might merely want to explore beyond) to bridge their knowledge between automated reasoning and formalization via proof. While Forge instructs us that a specification or predicate is true, Lean reveals *why* the specification is true. Oftentimes, this makes it difficult to debug incorrect specifications without additional tools like visualizations [7]. By attempting to construct proofs of model properties, students are compelled to think about their modeling choices and justify each statement, seamlessly translating between the statements they are making and proofs of their correctness [1].

6. Results and Examples

6.1. Forge as a Lean DSL

One of the crucial benefits of working with Lean 4 as a metaprogramming language and a target for our translation is the rich support for DSL implementation and integration. Lean and its accompanying Language Server Protocol (LSP)[18] are designed to have highly flexible and extensiblie user interfaces that expose useful APIs for implementers of DSLs and custom UI to utilize [5, 6]. Furthermore, Lean's extensible syntax and macro system are simple yet remarkably powerful [10, 8].

It is as such that we justify framing our implementation of Forge in Lean as an honest-to-goodness domain-specific language (DSL). We do not treat user-experience of our tool as an afterthought, nor do we skimp over ensuring that LForge has a set of developer aids just as capable as those found in Forge or Lean themselves. As mentioned in section 4.1, the fact that we are able to interact with Lean's own implementation means that many of Lean's 'IDE-like' features are exposed to us and available for us to use in the Forge DSL without much overhead.

The following are some (non-exhaustive) examples of the user experience and interface of Forge within Lean.

Syntax Highlighting

Superficially, by virtue of defining our syntax as Lean objects and isolating our keywords (we piggyback off Lean's lexer), we get syntax highlighting of Forge code 'for free', on par with Forge's native solutions. In fig. 2, the syntax of a Forge specification is highlighted.

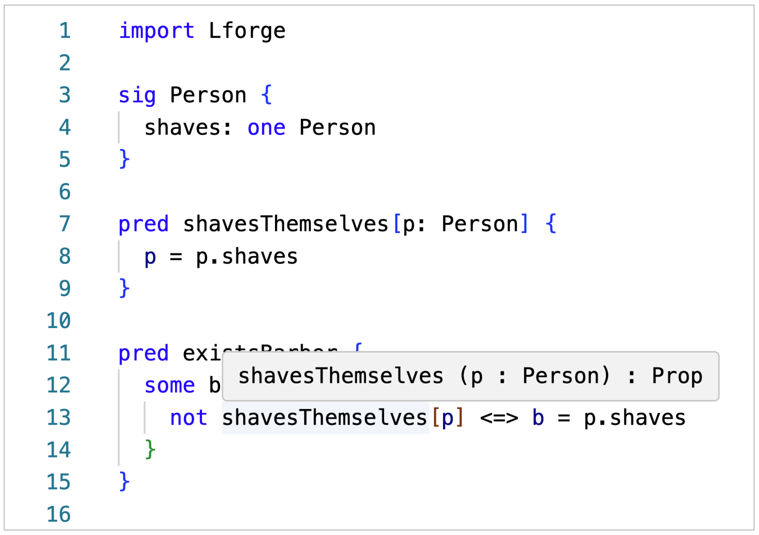
Types on Hover

Lean exposes an `addTermInfo'` method that allows us to attach declared names to pieces of syntax (nodes in Lean's concrete syntax tree), including custom syntax like ours for Forge. As such, we can annotate relevant pieces of syntax within our Forge specification to reflect names and types that are within scope. When the user hovers their mouse over a piece of syntax corresponding to a Forge expression, a hovering tooltip will display with the type of the expression. In fig. 2, the tooltip shows the type of a Forge predicate defined earlier in the file.

Documentation

As a pedagogical language, Forge has a focus on usability, learnability, and helpful feedback [7], especially when its parent language Alloy is far more permissive with errors. We follow in the same vein in reporting errors and missing features, and in addition we include documentation on Forge's syntax within Forge's on-hover features.

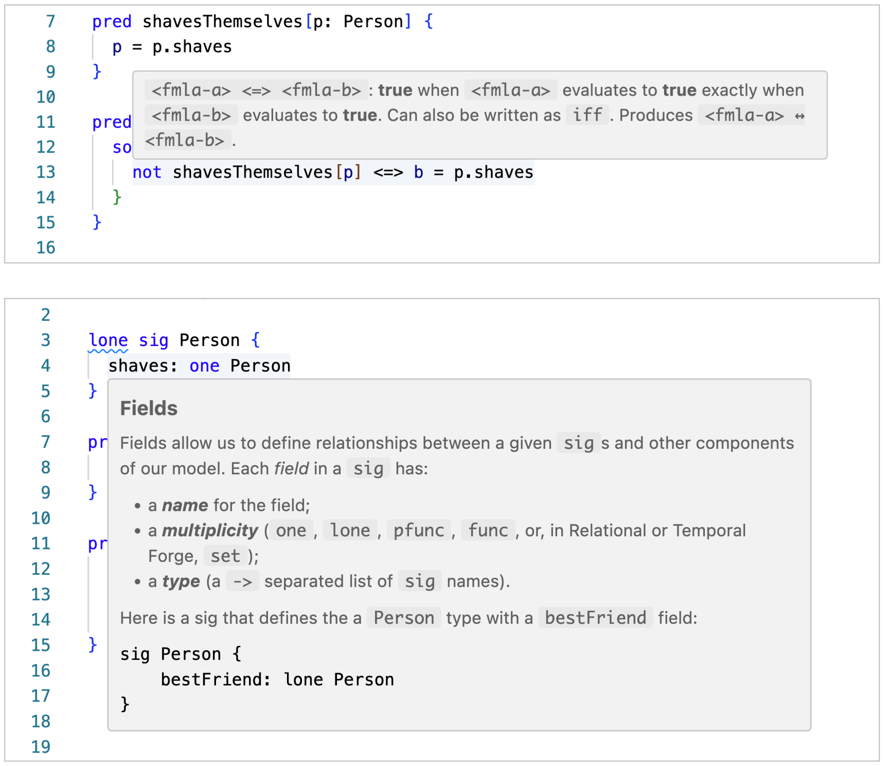
[18]This is the language server that processes Lean code and communicates with the code editor or integrated development environment. In this case, we use VS Code.



```
import Lforge

sig Person {
  shaves: one Person
}

pred shavesThemselves[p: Person] {
  p = p.shaves
}

pred exi
  some b
    not shavesThemselves[p] <=> b = p.shaves
  }
}

```

```
shavesThemselves (p : Person) : Prop
```

Figure 2.: Tooltips containing type information are available on hover. Forge syntax is automatically highlighted without any extra work.



```
pred shavesThemselves[p: Person] {
  p = p.shaves
}

pred
  so
    not shavesThemselves[p] <=> b = p.shaves
  }
}

```

`<fmla-a> <=> <fmla-b>` : **true** when `<fmla-a>` evaluates to **true** exactly when `<fmla-b>` evaluates to **true**. Can also be written as `iff`. Produces `<fmla-a> ↔ <fmla-b>`.

```

lone sig Person {
  shaves: one Person
}

pr

}

pr




}




```

Fields

Fields allow us to define relationships between a given `sig`s and other components of our model. Each *field* in a `sig` has:

- a ***name*** for the field;
- a ***multiplicity*** (`one`, `lone`, `pfunc`, `func`, or, in Relational or Temporal Forge, `set`);
- a ***type*** (a `->` separated list of `sig` names).

Here is a sig that defines the a `Person` type with a `bestFriend` field:

```
sig Person {
    bestFriend: lone Person
}
```

Figure 3.: We can define our syntax definitions to print with custom documentation text for users new to using Forge syntax.

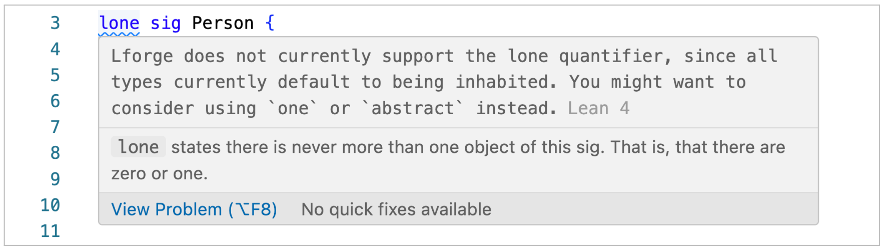
Forge documentation is included via docstrings that are inline with our syntax objects (see section 4.2), which is automatically included by Lean's LSP to display on the frontend. Figure 3 showcases docstrings of varying verbosity for operators as well as declaration syntax.

Additionally, we need to be clear and verbose about language features that are not supported in LFORGE. Since LFORGE includes a subset of relational Forge determined by compatibility with Lean's semantics, we prompts users attempting to use unsupported language features with clarification and a request to redefine their statements. Figure 4 showcases an example of a prompt that `lone` sig quantifier is unsupported and potentially ambiguous.

Error Checking

Compared to Forge or Racket, Lean (and consequently, LFORGE) provides a markedly better experience with error messages and prompting users when there are errors present in their source program. Since Lean runs in the background as a LSP, users immediately get the immediate feedback whether their source code parses and 'compiles'.[19]

Lean's error locality system allows its error monad to refer to any piece of syntax object to potentially throw an error. This allows us to prompt errors as soon and as granular as possible. Figure 5 illustrates error reporting at the level of specific identifiers.



```
lone sig Person {
```

```
Lforge does not currently support the lone quantifier, since all
types currently default to being inhabited. You might want to
consider using `one` or `abstract` instead. Lean 4
```

`lone` states there is never more than one object of this sig. That is, that there are zero or one.

Figure 4.: We can define custom error messages with our implementation to prompt users to change their specification if a piece of syntax is ambiguous or not supported.

Types

Lean's dependent type system is both a blessing and a curse when it comes to the task of translating a language with a foreign type system into Lean. While the type system is a highly-optimized algorithm that attempts to resolve type coercions, type classes, and reductions [2], it at often delicate and temperamental, especially when we are working at such a low level of emitting Lean `exprs`, which happens *after* type unification (see section 4.3). We discuss some of the downsides of such a strict type system in section 5.4, and introduced LFORGE features that circumvent Lean's restrictions and play into Lean's type system.

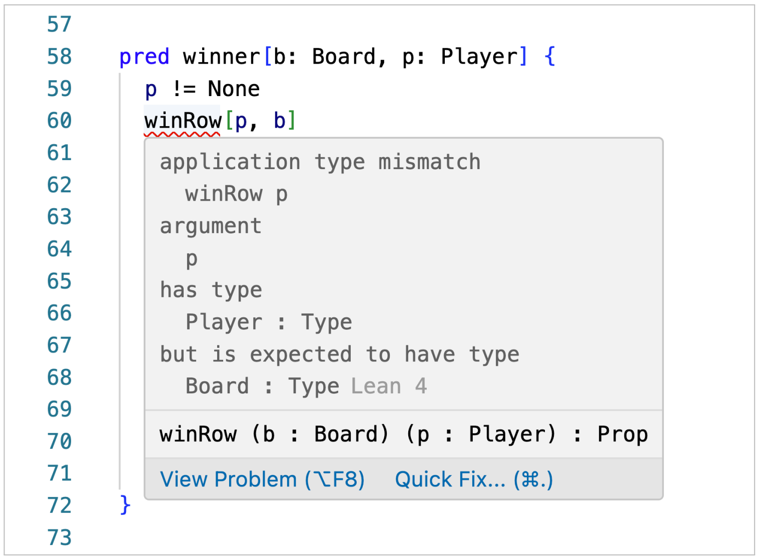
Here, we discuss some of the merits of implementing a DSL designed around Lean's extensive type system. One of the side effects of translating Forge into Lean is that we inherit Lean's powerful

[19]Forge translations in Lean are not really executable, so they provide their value in being interactive with the proof system.

type unification and checking system. This allows us, at specification-time, to check for type-errors within the specification. Alloy, on the other hand, is purposefully untyped [4] and only reports type errors at runtime when the successful evaluation of expressions results in the empty expression [3]. This proves difficult to debug and unwieldly for users to understand, as [7] observes. For students who are newly learning the idea of relations, sets, and units, lacking instantaneous feedback on the validity of types and expressions is immensely useful.

Since expressions in our Forge DSL need to translate to typed terms in Lean, we necessarily have to specify types (or use Lean metavariables awaiting unification in place of types) to the Lean expressions emitted. Fortunately, much of this process is abstracted away by the type inference system in place in Lean. For example, to make an application, say, a set union, we don't need to specify the type of set that are being used in the operands and `mkAppM` will complete that for us. However, if we specified two sets of different types, Lean would raise an error. This results in a type checking and inference system that is as powerful as Lean's with minimal overhead.

Since the Lean LSP provides type linting, our Forge DSL inherits this feature as well. In fig. 5, we pass the `Board` and `Player` arguments to `winRow` in the wrong order, which causes a type error. Lean is able to identify that the first input to `winRow`, `p`, has the wrong type and display an appropriate error message.



```
pred winner[b: Board, p: Player] {
  p != None
  winRow[p, b]
}
```

```
application type mismatch
  winRow p
argument
  p
has type
  Player : Type
but is expected to have type
  Board : Type Lean 4
```

```
winRow (b : Board) (p : Player) : Prop
```

Figure 5.: A Lean error message indicating a type mismatch in our Forge expression.

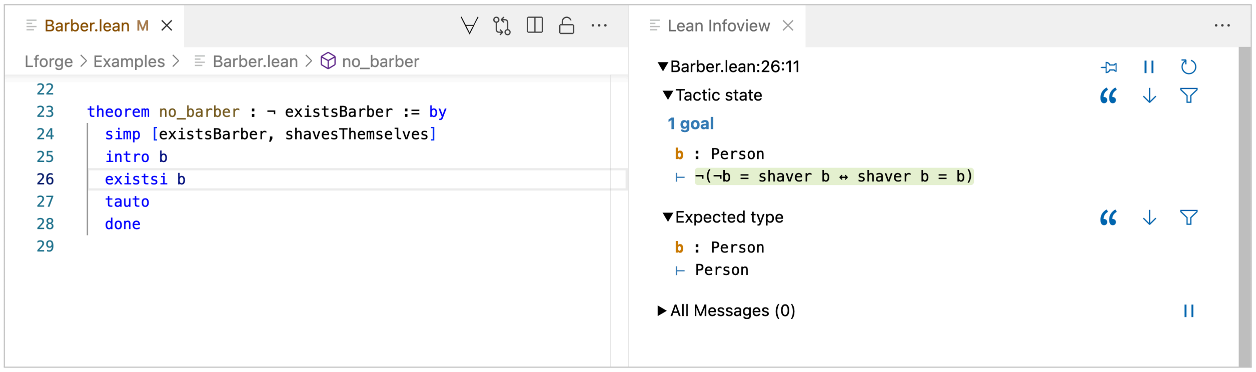
Mix-ins

complete

6.2. A Toy Example, *Continued*

We revisit our toy example from section 3.1 (the barber who "shaves all those, and only those, who do not shave themselves"). Recall that we had introduced a Forge specification for this problem, as well as an equivalent Lean specification of the paradox. We concluded earlier that while it was insightful for Forge to produce a result that this specification was `Unsatisfiable`, we might still desire for a general proof of this fact outside of Forge's finite and restricted search bounds.

Here's an example of what the last part of the modeling workflow—writing and completing the proof—would look like in Lean's interactive tactic mode:



```
22
23  theorem no_barber : ¬ existsBarber := by
24    simp [existsBarber, shavesThemselves]
25    intro b
26    existsi b
27    tauto
28    done
29
```

```
▼Barber.lean:26:11
▼Tactic state
1 goal
b : Person
⊢ ¬(¬b = shaver b ↔ shaver b = b)
▼Expected type
b : Person
⊢ Person
▶All Messages (0)
```

Figure 6.: The proof of nonexistence of a barber in the barber paradox, in Lean. The interactive proof state is on the right with the proof source on the left.

On line 25, we use the `simp` tactic (or alternatively, we can also use `simp only` or `rw`) to rewrite our Forge-defined predicates `existsBarber` and `shavesThemselves`. Due to the simplicity of the example, the goal can be closed using the `tauto` (tautology) tactic which repeatedly breaks down assumptions and splits goals with with logical connectives until it can close the goal.

While simple, this example demonstrates the expressiveness of Forge programs embedded in Lean, and the relative ease with which some proofs of translated properties can be executed. The following section, section 6.3, presents a more elaborative example of LFORGE.

6.3. A Mutual-Exclusion Protocol

- Circle back to the earlier example
- A more complicated example, potentially with Lean mixins?

Bibliography

[1] Jeremy Avigad. “Learning logic and proof with an interactive theorem prover”. In: *Proof technology in mathematics research and teaching* (2019), pp. 277–290.

[2] Leonardo De Moura et al. “The Lean theorem prover (system description)”. In: *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*. Springer International Publishing. 2015, pp. 378–388.

[3] Jonathan Edwards, Daniel Jackson, and Emina Torlak. “A type system for object models”. In: *ACM SIGSOFT Software Engineering Notes* 29.6 (2004), pp. 189–199.

[4] Daniel Jackson. “Alloy: a language and tool for exploring software designs”. In: *Communications of the ACM* 62.9 (2019), pp. 66–76.

[5] Leonardo de Moura and Sebastian Ullrich. “The lean 4 theorem prover and programming language”. In: *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*. Springer. 2021, pp. 625–635.

[6] Wojciech Nawrocki, Edward W Ayers, and Gabriel Ebner. “An extensible user interface for Lean 4”. In: *14th International Conference on Interactive Theorem Proving (ITP 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2023.

[7] Tim Nelson et al. “Forge: A Tool and Language for Teaching Formal Methods”. In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA1 (2024), pp. 1–31.

[8] Arthur Paulino et al. *Metaprogramming in Lean 4*. 2024. URL: https://leanprover-community.github.io/lean4-metaprogramming-book/.

[9] Bertrand Russell. *The philosophy of logical atomism*. Routledge, 2009.

[10] Sebastian Ullrich and Leonardo De Moura. “Beyond notations: Hygienic macro expansion for theorem proving languages”. In: *Logical Methods in Computer Science* 18 (2022).

Appendices

A. Barber Paradox Proof

The proof to the barber paradox annotated with the Lean tactic state after each tactic/step is provided below. This is provided in file in the corresponding software artifact. filepath

```
theorem no_barber : ¬ existsBarber := by
  /-
  ⊢ ¬existsBarber
  -/
  simp [existsBarber, shavesThemselves]
  /-
  ⊢ ∀ (x : Person), ∃ x_1, ¬(¬x_1 = shaver x_1 ↔ shaver x_1 = x)
  -/
  intro b
  /-
  b : Person
  ⊢ ∃ x, ¬(¬x = shaver x ↔ shaver x = b)
  -/
  existsi b
  /-
  b : Person
  ⊢ ¬(¬b = shaver b ↔ shaver b = b)
  -/
  tauto
  done
```