Lforge: Extending Forge with an Interactive Theorem Prover

Embedding the Forge specification language via a machine translation
as a language-level feature of the Lean theorem prover.

Jiahua Chen

Advisor: Robert Lewis
Reader: Tim Nelson

A Thesis submitted in partial fulfillment of the requirements for Honors
in the Departments of Mathematics and Computer Science at Brown University

Department of Computer Science
Brown University
Providence, Rhode Island
April, 2024

Abstract

While formal methods are being increasingly applied in industry and are invaluable in allowing users to specify, model, and verify complex systems, the multitude of available tools offer little to no interoperability amongst each other. There are no common formal methods models nor specification languages, and each tool offers precisely what its specific logical framework allows. This creates a dilemma in the field that while users understand the strengths and capabilities of formal methods tools, they are often not able to select a tool appropriate for the task at hand. We present LFORGE, a tool that implements the Forge specification language via a translation process as a language-level feature of Lean. LFORGE offers a 'best-of-both-worlds' approach—allowing users to no longer be constrained by the bounded yet automatic resolving capabilities of Forge nor the tedious proof process of Lean. LFORGE serves as an example of interfacing between two drastically different tools, thus allowing users to harness the resolving capabilities of two frameworks: Forge, an offshoot of the Alloy specification language based on a relational logic solver that can automatically prove facts about bounded models; and Lean, an interactive theorem prover that allows users to prove generalized statements. In doing so, LFORGE serves as a model of what specification portability might look like across different classes of formal methods tools, as well as a model of user experience in such a translation. LFORGE is furthermore one of the first experimentations of Lean 4's rich metaprogramming capabilities. This allows LFORGE to be a full-fledged Forge DSL within Lean with a focus on usability-first despite operating within the constraints imposed by Forge and Lean's respective formal frameworks.

Acknowledgements

Write acknowledgements...

Contents

Notation

Code snippets and listings have been included in this paper to serve as examples, motivation, or to provide implementation details. Where they are included, the color of the code block denotes the source language and context.

The following is an example of a Lean implementation code block:

```
-- This is the code block for the Lean implementation of our translation
def forgeEnsureHasType (expectedType? : Option Expr) (e : Expr)
    (errorMsgHeader? : Option String := "Forge Type Error")
    (f? : Option Expr := none) : TermElabM Expr := do
  let some expectedType := expectedType? | return e
  if (← isDefEq (← inferType e) expectedType) then
    return e
  else
    mkCoe expectedType e f? errorMsgHeader?
```

This denotes code from the implementation of the translation from Forge to Lean. This encompasses the parsing and elaboration of Forge syntax within Lean and is most often the metaprogramming implementation of Forge in Lean.

The following is an example of a Forge code block:

```
-- This is the code block for a snippet of a model specification in Forge
sig Node {
  neighbors : set Node
}
pred connected[a : Node, b : Node] {
  b in a.neighbors
}
```

This denotes examples of a model specification (or a snippet of a model) in Forge.

The following is an example of a Lean translation code block:

```
-- This is the code block for the translated Lean equivalent of a Forge snippet
opaque Node : Type
opaque neighbors : Node → Node → Prop

def connected (a : Node) (b : Node) : Prop :=
  neighbors a b
```

This denotes examples of the translated version of a Forge model or snippet. This is oftentimes the translated Lean code that is emitted out of our program.

1. Introduction

Formal methods are increasingly being applied in industry. Domain-specific formal methods tools empower researchers to specify, model, analyze, and verify complex software and hardware systems that otherwise prove unfeasible to fully examine by hand [1, 28]. These applications prove invaluable when the functionality of an existing system needs to be verified to be correct, or when new systems need to be synthesized based on a set of logical constraints and specifications [60].

Yet, there is a multitude of tools that exist in the realm of formal methods: type-checked programming languages [21, 12], property-based testing frameworks [20, 31], modeling and specification languages [24, 23, 45], SMT solvers [15], proof assistants [38], etc. Each of these tools offers a tailored set of features and is based upon a specific yet different logical framework. An SMT solver based on the boolean satisfiability problem is different from a proof assistant which relies on dependent type theory. Due to operating under these diverse and different frameworks, few if any offer any form of interoperability—there is no common form of formal methods modeling or specification language. Each tool offers precisely what its framework allows.

As a result, there arise limitations as to what *can* and *cannot* be modeled by specific techniques. In a survey of applications of formal methods within industry, the majority of respondents repeatedly identified that formal methods are useful in projects but similarly agreed that they felt tools were incapable or often ill-suited to the particular task at hand [60]. This problem of selecting tools is so prevalent that there have been surveys and methodologies devised for this task [26, 13].

This paper introduces Lforge[1], a tool that implements the Forge specification language [45] (a pedagogical offshoot of Alloy [24], which we use due to its gradually featured nature as well as simpler syntax) via a translation process as a language-level feature of Lean, an interactive proof assistant [38].

In doing such, Lforge aims to be an example of interfacing between two drastically different tools in the larger realm of formal methods. The goal of this is to reduce the number of 'make-or-break' choices that researchers face within the field and allow users to harness the capabilities of multiple formal methods systems, picking and choosing the features from multiple feature sets that are important to them.

Forge and Lean work in fundamentally different ways. Forge, which is based on the Alloy relational model solver, uses the language of relational logic to specify and 'solve' systems. A system is specified as a collection of *signatures*, and model specifications are provided as a set of logical constraints on relations between signatures. Forge will then apply a SAT solver to the set of constraints to generate an *instance* of the specified model (with some finite instances of each signature) [23, 45]. This makes Forge suited for generating finite examples or counterexamples to provided specifications. Lean, on the other hand, is an interactive theorem prover that is based on dependent type theory[2] [5]. Systems are implemented within the functional programming language, and

[1] Unfortunately the superior option amongst contenders Flean, Fean, and Lorge.

[2] Specifically, the *calculus of (inductive) constructions*. This is the logical system first implemented in Coq [9].

theorems containing logical statements about said systems can be stated and proven. Lean verifies that said proof is correct—and that the system has claimed properties.

While Forge can automatically reason (via solving for satisfying instances) about finite instances and can solve for model existence, Lean allows the user to make general claims about a system, at the cost of requiring manual proving. Lean can assist in generalizing statements made in Forge, while Forge can easily disprove incorrect Lean statements via counterexample[3]. After a user utilizes Forge's automated reasoning tools to generate potential properties related to their model, they can input their Forge model directly into a Lean source program and write generalized proofs of the same properties directly in Lean. LFORGE recognizes the benefit of being able to interoperate between these two models for verifying systems can prove useful in checking real-world models, especially where a human translation between the two tools can be tedious and prone to errors.

The implication of this work and further hope is that the model specification syntax of Forge/Alloy can become a universal and portable specification language suitable for multiple tools based on multiple frameworks alike. While we do make compromises as to what is and isn't able to be brought over from Forge to Lean, LFORGE is explicit and clear about those assumptions and what is left for the user to specify or supplement. The larger objective is for LFORGE to serve as a model of what specification portability could look like across classes of formal methods tools, and what the user experience might be as specifications are being translated and utilized.

LFORGE builds on a long series of existing work implementing and embedding Alloy and similar specification languages into existing programming languages [35, 36, 25, 32, 33], theorem provers [2, 17, 27]. We propose a modern example of this protocol that focuses on user experience and select our source and target language around these constraints. Forge is designed to be learnable and a simpler adaptation of Alloy [45], and Lean provides an extensible [40, 58] and usable [6] framework for theorem proving.

LFORGE serves as one of the first experimentations with Lean 4's complex and rich metaprogramming capabilities [49], implementing Forge as a full-fledged DSL in Lean 4. At the same time, we harness Lean 4's out-of-the-box language server[4], interactive capabilities [40], and type unification system to suit Forge. The end product is a language experience that is definitely on par, if not more feature-rich than native Forge support in any IDEs from a purely user-experience perspective.

The hope is that LFORGE becomes a DSL that focuses on usability first despite operating within the translation constraints imposed by formal frameworks. We hope for it to become a tool (or, at the very least, serve as a blueprint for a tool) that is an essential utility in any researcher's formal methods toolbox.

[3]Lean will not indicate whether a statement is true or false.

[4]What powers Lean 4's autocomplete, interactive theorem prover, tooltip-on-hover, etc. in VS Code.

2. Background

We provide some necessary background relating to Forge, and Lean, as well as existing work translating and embedding Alloy and related tools in a multitude of target languages.

2.1. *Forge*, *Alloy*, and other relational specification languages

The Z language pioneered the formal specification of software systems, providing the first syntax and language for users to formally specify software specifications and behaviors [54]. Alloy builds on Z by introducing a simpler yet expressive relational logic whose models can be automatically decided in small scopes [23]. Alloy's automatic instance search is achieved using its backend Kodkod, which translates relational specifications into SAT problems to solve them [55]. While the search space was often small and scaled super-exponentially with respect to the size of the model [23], automated solvers provided what traditional theorem provers lacked—automation.

Both Alloy and Kodkod have been widely used and implemented in practice. [56] provides some broad use cases of Alloy: it is used to model software designs, verify program specifications, generate test cases, and synthesize examples and counterexamples. Alloy is further used across a diverse selection of fields ranging from cryptography to networking [56]. Kodkod, which provides a convenient specification API accessible outside of the Alloy language [55], is also widely utilized. Nitpick, proof assistant Isabelle's counterexample generator, relies on Kodkod as its automated search backend [11]. Bounded verifiers and domain-specific analyzers alike, such as Forge [45], a pedagogical offshoot of Alloy; and Margrave, an access-control policy analyzer [47], all utilize the Kodkod solver. Similar projects porting Alloy into other languages and environments (see below section 2.3) also similarly utilize Kodkod as well as Alloy's syntax to some degree.

Forge specifically, improves Alloy on the principles of learnability and usability, whilst maintaining core features of Alloy such as push-button automation and visualization of instances. We focus on Forge because the philosophy behind Forge's inception are core to our project as well: we want to focus on usability, we select appropriate and applicable subsets of the Forge specification language, and that its simpler syntax compared to Alloy's provides a suitable translation target. We furthermore find that Forge's focus on the Froglet language level—which restricts fully relational functionality—is also compatible with the capabilities and type system of Lean.

2.2. *Lean* and other proof assistants

Lean [38], along with other proof assistants such as Isabelle [50] and Coq [10], work in a drastically different way compared to Alloy. While they are not limited by the same bound constraints of Kodkod-based solvers, this comes at the cost of automation. Proof assistants require guidance from the user, just like in traditional mathematical or logical proof, to prove statements and theorems. Automation of varying kinds and capabilities are often provided as additional plugins separate from the core language [11, 19, 14, 30].

One of Lean's[5] particular focus is on extensibility and ease-of-use. Lean 4 is a theorem prover *and* programming language designed around metaprogramming, extensibility [38], and user experience [40]. Lean exposes its own implementation for users to extend, allowing us to implement additional features, and even syntax, into the language with relative ease. Furthermore, it's built on a platform that allows us to access first-class IDE features, which enables us to provide a DSL experience that is next-to-native. Lean's flexibility and extensibility suit it for both implementing a translation, as well as presenting such a translation to the end-user.

2.3. Related Work

Both Alloy and Lean have long histories of integration with other tools and programming languages.

Alloy (or the Kodkod solver) has been translated or embedded in: its predecessor model specification language Z [32]; object-oriented programming languages Java [35] and Ruby [36], which introduces 'mixed execution' between automated searches and imperative code; as well as the Athena [3, 39], B [33, 27], and Isabelle [11] theorem provers. [17] discusses a class of *solver-aided programming languages* where programming environments are built around the Alloy solver that allows data processing and software verification simultaneously. Several extensions of Alloy exist either to adapt it to a specific problem [46, 43] or to introduce new functionality [48, 37, 61].

There are also many examples of Lean's extensibility and interoperability. [57] describes Lean 4's extensive macro system that allows it to be extremely extensible. [22] implements a custom proof methodology, Small Scale Reflection, via Lean's metaprogramming framework into Lean's proofwriting mode, complete with custom syntax, macros, elaboration functions, and visual support. [29] implements an interface between Lean and computer algebra system Mathematica, allowing users to interact with declarations and harness the capabilities of both tools.

We believe that these serve as guiding foundations atop which we implement our translation.

[5] Specifically, Lean 4.

3. Motivation

As discussed in section 2.1 and section 2.3, there is a desire for more functionality out of specification languages like Alloy and Forge for them to be more usable and widely applicable in the real world.

There is a want for something beyond model-finding capabilities from tools like Alloy and Forge [36]. By adding the deductive 'power' of interactive theorem prover to Forge, we open up possibilities of using Forge for types of analyses not previously possible.

Furthermore, one of the pain points of model searching is that systems are constricted to checking small examples and models—model checkers of the likes of Forge scale poorly with large models which makes complex problems difficult to model and analyze [7, 52]. By embedding Forge within Lean, we can break Forge specifications free of the bounds imposed by the SAT solver backend and instead utilize Lean's size-agnostic logical framework for proof of model properties. Pivoting the same model into a different class of models—a theorem prover—we gain the capabilities of writing proofs at the cost of needing to guide the model through a written proof instead of an automated search.

3.1. A Toy Example

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

1. The user specifies a model of 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 6.5). After this, all sigs, fields, and 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 Mixed Execution, section 7.1).

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 some motivation. 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?" [53, 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 {
  shaved_by: one Person
}

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

pred existsBarber {
  some barber : Person | all p : Person | {
    not shavesThemselves[p] <=> p.shaved_by = 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 more generally (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[6].

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

```
opaque Person : Type
opaque shaved_by : Person → Person

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

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

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

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

[6] *An aside:* We are aware that this is not generally true, and that our example might in fact be *overly* simplified. The Bernays-Schönfinkel-Ramsey (or *effectively propositional*) class of first-order logic formulas—formulas written in the form $\exists x_1 \ldots \exists x_n \forall y_1 \ldots \forall y_m, \phi(x_1, \ldots, x_n, y_1, \ldots, y_m)$—are in fact decidable [8, 51]. Satisfiability for these formulas in a finite model with pre-determined size (as a function of the formula) *is* sufficient and necessary for general satisfiability of the formula. The statement of the barber paradox, interpreting `shaved_by` as a relation, is in such a form. One could imagine an analogous example that does not fall into such a specific class of first-order formulas, where Forge's bounded satisfiability search is not sufficient. The vast majority of model specifications and their properties predicate lie outside this class of formulas.

The conclusion of this example, including the Lean proof of our property, is provided in section 7.2. 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 is 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 are 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 [45]. Students who are interested in formal methods often take Logic for Systems in the Computer Science department at Brown, the course in which Forge was developed and taught. Students continue 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 encourage students to explore and formalize topics that they are interested in.

We believe LFORGE provides an opportunity for students who have a 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 *potentially has* certain properties, Lean reveals *why* the specification satisfies its properties. Forge provides a simple and easy avenue for students to construct examples and counterexamples that they can then visualize [45], but can then trade the automated (counter)example search functionalities of Forge in favor of a theorem prover that allows them to construct a formal proof now that they are motivated by examples and quick prototyping. By attempting to construct models and state (then prove) properties of said model, students are compelled to think about their modeling choices, first verify them on a first-order via an automated search via Forge, only then do they attempt to justify each statement within Lean, seamlessly translating between the statements they are making and proofs of their correctness [4].

4. Design

4.1. Design Summary

Lean 4 is a good target for our translation as well as a suitable language to implement our translation in because Lean 4 is mostly implemented in itself. As users, we can utilize and emit the same data structures used in Lean's implementation to extend the functionality of Lean [38]. These metaprogramming capabilities of Lean make our work implementing a Forge module in Lean much easier.

The Lean 4 compilation process is structured as in fig. 1.

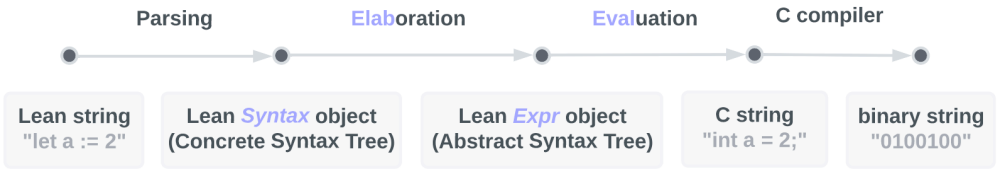


Figure 1. A diagram from [49] summarizing the Lean 4 compilation process.

Specifically, the parsing and elaboration steps are designed to be highly customizable and are provided as a 'first-class' feature of Lean 4. We approach the problem of translating Forge into Lean as a task of adding new language features to Lean itself. We define our own Lean syntax objects that correspond to a concrete syntax tree (CST) of Forge and implement a parser for Forge (see section 4.2), and then implement a custom elaboration function for our Forge syntax to translate it into native Lean expressions (see section 4.3). LFORGE's translation process, which is tightly integrated into the Lean compilation process, is outlined in fig. 2.

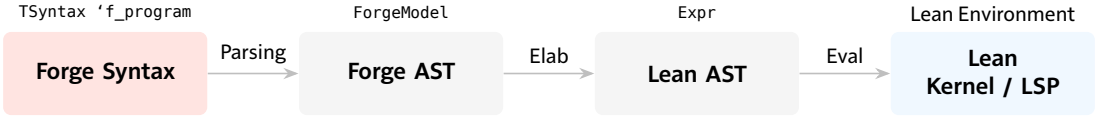


Figure 2. A diagram of LFORGE's translation process.

As a result, there is as little additional overhead as possible when translating a Forge specification in Lean. After users have imported our module, all Forge expressions *are* valid Lean expressions and the two languages can be used interchangeably[7]. Definitions can be passed to and from (see Mixed Execution, section 7.1), and most notably Forge predicates can be proven in Lean.

[7]We are very fortunate that there are few to no conflicts between Forge and Lean syntax that would hinder this. We provide a flag `#lang forge` following Forge's Racket `#lang` syntax when users want to explicitly denote Forge code, and the Lean parser will try to parse as many succeeding lines as Forge as possible.

4.2. Syntax, Parsing, and the Forge AST

In the case of parsing, by defining the Forge grammar in the same specification format that Lean defines its syntax in, we can rely on Lean's parser to parse Forge source code for us.

The benefits of this are two-sided:

1. We are provided `Syntax`-typed Lean objects at the end of this process, the same type that parsing a Lean program would produce. This enables us to treat our Forge implementation as an implementation of additional Lean language features, and we can also harness Lean metaprogramming libraries along the way. This is to say, we are implementing Forge in Lean the *same way* Lean is implemented in Lean, which greatly reduces our burden for additional implementation overhead.

2. By defining Forge 'blocks' as a Lean `command`—which is the top-level syntax category[8]—users of the tool can insert raw Forge, without any annotations that this is an "extension language", into Lean. With the addition of an import statement, every Forge program *is* a valid Lean program.

Lean allows us to create *syntax categories* for each nonterminal symbol in our grammar. At the top-level, we have defined `f_sig` (Sigs), `f_pred` (Predicates), `f_fun` (Functions). Terms are either `f_fmla` for formulas (evaluate to True or False) or `f_expr` for expressions (evaluate to a set, relation, int).

For example, the grammar[9] of Forge arguments and predicates is:

$$\langle arg \rangle \quad ::= \langle ident \rangle,+ \text{ ':' } \langle expr \rangle$$

$$\langle args \rangle \quad ::= \langle arg \rangle,*$$

$$\langle pred \rangle \quad ::= \text{'pred'} \langle ident \rangle \text{ ['['} \langle args \rangle \text{ ']'] '\{'} \langle fmla \rangle * \text{ '\}'}$$

Which we can translate into a corresponding syntax definition in Lean:

```
declare_syntax_cat f_arg
syntax ident,+ ":" f_expr : f_arg

declare_syntax_cat f_args
syntax f_arg,* : f_args

declare_syntax_cat f_pred
syntax "pred" ident ("[" f_args "]")? "{" f_fmla* "}" : f_pred
```

Following this blueprint, we can translate the entirety of the grammar of Forge[10] into Lean syntax definitions. This is provided in our package as the `Lforge.Ast` module (see appendix A).

[8] For example, the top-level definition in Lean "`def x: Int := 0`" is a 'command'.

[9] Where $,+$ and $,*$ denote one/zero or more comma-separated occurrences respectively. $+$ and $*$ denote one/zero or more repetitions.

[10] At least, a useful subset of the Forge language we care about. This is based on the grammar of Alloy [24, 23, 45].

What remains to be done is to convert syntax, almost one-to-one, into an AST for Forge, and then *elaborate* (see section 4.3) our AST into Lean expressions and declarations.

```
structure Predicate where
  name : Symbol
  name_tok : Syntax
  args : List (Symbol × Expression) -- (name, type) pairs
  body : Formula -- with args bound
  deriving Repr, Inhabited
```

The associated structure definitions of the Forge AST is a deep embedding of Forge into Lean. For example, the following Forge predicate:

```
pred ownerOwnsPet {
  all p: Person | all pet: Pet | { pet in p.pets <=> pet.owner = p }
}
```

yields the following deep embedding (AST) as the output of parsing:

```
{
  name := "ownerOwnsPet",
  args := [],
  body := quantifier all [("p", literal "Person")] (
    quantifier all [("pet", literal "Pet")] (
      iff
      -- pet ∈ p.pets
      (subset (literal "pet") (join (literal "p") (literal "pets")))
      -- pet.owner = p
      (eq (join (literal "pet") (literal "owner")) (literal "p"))))
  : Predicate
}
```

Our overall parser has type `TSyntax `f_program → MetaM ForgeModel`[11], where `f_program` is the top-level syntax category for Forge programs (lists of sigs, predicates, and functions), and `MetaM` is a metaprogramming monad that provides us with error reporting. Using this, we can then implement a translation of our Forge model into native Lean expressions and types.

4.3. Elaboration

Elaboration in Lean 4 processes Lean `Syntax` objects, which are the outputs of the Lean parser, into Lean `Expr` objects[12], which are Lean's low-level kernel representations [49]. Elaboration is responsible for Lean's type and metavariable unification[13], which provides all the type information to Lean.

[11] 'Sub-parsers', like the one that parses single predicate declarations, are typed `TSyntax `f_pred → MetaM Predicate`. A `ForgeModel` structure wraps sigs, predicates, and functions into a single structure.

[12] Technically, `Expr` objects wrapped in relevant monads that allow us to implement side-effects, like error and info reporting within the Lean LSP (see section 7.1) and interact with Lean's environment.

[13] That is, types are inferred, coerced, and type classes resolved at this step. Types, including implicit types, must be fully specified within `Expr` objects.

Analogously, LFORGE implements a Forge-specific custom elaboration function that takes our deep embeddings of type `ForgeMode` and returns a `CommandElabM Unit` type, where the `CommandElabM` monad allows us to add declarations to the environment (and `Unit` because we don't expect top-level declarations/commands to return values).

Our elaboration function takes care of elaborating sigs and fields into their corresponding opaque types (see section 6.4), and creates relevant definitions for predicates and functions, inserts the corresponding translations of formulas and expressions respectively (see below section 5), and adds said definitions into the working Lean environment.

5. The Forge Model in Lean: An Overview

We need to make careful choices of how we use elaboration (see above section 4.3) to translate Forge concepts into corresponding Lean equivalents. While many of our translations are self-evident, others have complex subtleties. Having the *means* to perform such a translation does not make the task of translation itself any easier. We need to ensure that the translation is coherent and interoperable[14].

Here we outline the Forge syntax and give an overview of their equivalents in Lean. When there are nuances or specific edge cases in a particular translation, we refer to the appropriate section that addresses them. Where translations have not been implemented, we justify this conscious decision.

This should serve as a birds-eye-view of the entire project, and point to specific instances of translations and implementations mentioned throughout this paper.

Sigs See section 6.4 for a discussion on Forge sigs, how they're translated, and how quantifiers (like `one` or `abstract`) are handled.

Formulas Formulas evaluate to some `True` or `False` value; see table 1 for translations of various Forge formulas.

Expressions Expressions evaluate to some set-typed expression; see table 2 for translations of the various Forge expressions. While we treat integers as expressions, they are detailed separately.

Predicates and functions Predicates and functions get mapped to top-level definitions in Lean, with the body being the translated formula or expression respectively. See code listings included in section 6.4 for an example with functions and section 6.1 for an example with predicates.

Operations with integers See table 3 for translations of integer expressions, and additionally section 6.6 on how integers are specifically handled and implemented.

[14] Informally, our translation should be like a homomorphism, preserving the structure of Forge models.

Table 1. A list of Forge formula syntax and their corresponding Lean implementations in LFORGE. `x`, `y`, and `z` represent formulas; `a` and `b` represent expressions; `T` represents the sig of expression `a`; and `x`, `y` represent integers.

Forge Syntax	**Lean Implementation**
`!x`	$\neg$`x`
`x && y`	`x` $\wedge$ `y`
`x || y`	`x` $\vee$ `y`
`x => y`	`x` $\rightarrow$ `y`
`x => y else z`	`x` $\rightarrow$ `y` $\wedge$ $\neg$`x` $\rightarrow$ `z`
`x <=> y`	`x` $\leftrightarrow$ `y`
`some a`	$\exists$ `x : T, a x`
`no a`	`a` $= \emptyset$
`one a`	$\exists!$ `x : T,` $a = \{x\}$
`lone a`	`one a` $\vee$ `no a`
`a in b`	*Usually* `a` $\in$ `b` *or* `a` $\subseteq$ `b`*, but varies, see section 6.2.*
`x = y`	`x` $=$ `y`
`a = b`	*Usually* `a` $=$ `b`*, but varies, see section 6.2.*
`n < m`	`n` $<$ `m`
`n <= m`	`n` $\leq$ `m`
`n > m`	`n` $>$ `m`
`n >= m`	`n` $\geq$ `m`
`all a : T | {`$\langle fmla \rangle$`}`	$\forall$ `a : T,` $\langle fmla \rangle$
`some a : T | {`$\langle fmla \rangle$`}`	$\exists$ `a : T,` $\langle fmla \rangle$
`one a : T | {`$\langle fmla \rangle$`}`	*Unimplemented*[15]
`no a : T | {`$\langle fmla \rangle$`}`	*Unimplemented*
`lone a : T | {`$\langle fmla \rangle$`}`	*Unimplemented*
`let a =` $\langle term \rangle$ `| ...`	`let a :=` $\langle term \rangle$ `in ...`
$\langle pred \rangle$`[a,...]`	$\langle pred \rangle$ `a ...`
`true`	`True`
`false`	`False`

[15] Due to the semantics of the 'complex quantifiers': their interactions with multiple binders and that they encode extra constraints invisibly [41], there is no suitable Lean equivalent. Users are suggested to rewrite their quantification statements using only `all` and `some`, which all complex quantifiers can be expressed using.

Table 2. A list of Forge expression syntax and their corresponding Lean implementations. `x` represents a formula; and `a` and `b` represent expressions.

Forge Syntax	Lean Implementation
`~a`	`Relation.Transpose a` or `a.swap`[16]
`^a`	`Relation.TransGen a`
`*a`	`Relation.ReflTransGen a`
`a + b`	`a` $\cup$ `b`
`a - b`	`a` $\setminus$ `b`
`a & b`	`a` $\cap$ `b`
`a.b` *or* `b[a]`	*Varies, see section 6.2.*
`a->b`	*Varies, see section 6.2.*
`if x then a else b`	`if x then a else b` *(or, `ite x a b`)*
`{ x : T |` $\langle fmla \rangle$`}`	λ `x` $\mapsto \langle fmla \rangle$
`let a =` $\langle term \rangle$ `| ...`	`let a` $:= \langle term \rangle$ `in ...`
$\langle fun \rangle$`[a,...]`	$\langle fun \rangle$ `a ...`
`# a`	`Set.ncard a`

Table 3. A list of Forge integer-related syntax and their corresponding Lean implementations. `n, m` represent integers; `a` represents expressions; and `T` represents the sig of expression `a`.

Forge Syntax	Lean Implementation
`sing[a]`	`(a :` $\mathbb{Z}$`)`
`sum[a]`	`Finset.sum a id`
`max[a]`	`Finset.max a`[17]
`min[a]`	`Finset.min a`
`abs[n]`	`Int.natAbs n`
`sign[n]`	`Int.sign n`
`add[n, m, ...]`	`n + m + ...`
`subtract[n, m, ...]`	`n - m - ...`
`multiply[n, m, ...]`	`n * m * ...`
`divide[n, m, ...]`	`(n / m) / ...`
`remainder[n, m]`	`Int.mod n m`
`sum a : T	{`$\langle int\text{-}expr \rangle$`}`	`Finset.sum (T : Set T)` $\langle int\text{-}expr \rangle$

[16] This depends on the type of `a`. In the specific case when `a` is a cross product, we can use `Prod.swap`.
[17] With slight modifications since behavior can be undefined (can produce $\bot$ or $\top$).

6. Implementation Details and Challenges

6.1. "Everything is a Set"

The predominantly relational nature of Forge introduces a point of friction between our translation from Forge to Lean. In Forge, every expression is implicitly a relation or a set (set when that expression has arity-1[18]). Even when we know that an expression is a relation or set with cardinality 1 (for example, it could be introduced as a binder from a quantification), they are used as if they were a singleton set in Forge expressions that expect a set as an operand. Under the hood, all expressions in Forge are treated as a set (or multi-arity relation).

This everything-is-a-set approach of Forge allows the following expression (within the existential quantifier), translated "there is some `Student` who is their own friend":

```
sig Student {
  friends : set Student
}
pred ownFriend {
  some s: Student |
    s in s.friends
}
```

```
opaque Student : Type
opaque friends : Student → Student → Prop

def ownFriend :=
  ∃ s : Student,
    friends s s
```

Note that `s` is a 'singleton', but it is used as if it were an honest-to-goodness set in the join operation (`s.friends`) and the inclusion operation (`s in ...`). We can concisely translate into a statement in Lean of the likes of "$\exists s$ such that on the `friends` relation, $(s, s) \in$ `friends`." Note that because `s` is a singleton—that is, no set with more than one element could be bound to `s` as a result of our existential quantifier—we were able to translate the join in `s` $\bowtie$ `friends` as the partial application to our relation `friends s`, and the `in` keyword became set membership `s` $\in$ `s` $\bowtie$ `friends` which was just `(friends s) s`.

Consider an alternative when we relax the requirement that `s` ought to be a singleton in the Forge source:

```
pred ownFriend[t: Student] {
  let s = t.friends |
    s in s.friends
}
```

```
def ownFriend (t : Student) :=
  let s := friends t,
    friends s s -- Type error!
```

Which is loosely "for a `Student t`, the set of `t`'s friends is a subset of the set of all *their* friends." Here, `s` in Forge is bound to a set of `t`'s friends. Had we translated this in the same way, `s` would be typed `Student → Prop` (or equivalently, a `Set Student`), and `friends s s` raise a type error.

We instead have to resolve the join `s` $\bowtie$ `friends` without our shortcut above of partially applying `s` to `friends`:

```
s.friends
```

```
λ x₂ ↦ ∃ x₁ : Student, s x₁ ∧ friends x₁ x₂
```

which is immediately more cumbersome than our earlier solution.

[18] The notation we use is that a `Set α`, which has equivalent type `α → Prop`, has arity-1, and so on. This is the arity convention in Forge and aligns with Lean's definitions of relations.

The same applies when we now try to implement the inclusion `in` operator:

```
s in s.friends
```

```
Set.Subset s (λ x₂ ↦ ∃ x₁ : Student, s x₁ ∧ friends x₁ x₂)
```

which becomes a subset operator[19] instead of set membership.

This example describes a fundamental incompatibility between Forge and Lean that we need to resolve. Forge is indifferent between whether an expression is a singleton or a relation/set and treats the two indiscriminately. This approach of treating everything as a set allows operations like relational join and 'in' to work across all scenarios alike.

However, Lean tends to prefer expressions that are not sets (that is, honest-to-goodness singletons). In the cases above, this allows for relational join to be a partial application, and 'in' to be set membership. For the majority of use cases, this singleton-friendly translation suffices. When we are dealing with sets, set operators such as join and 'in' (which is now the subset operator) become more convoluted as demonstrated above. Additionally, while joining a singleton and a relation via the partial application solution applies to relations of varying arities, a join expression between two arbitrary relations takes in different types, and hence implementations, depending on the respective arities and types of the arguments.

This means that we shouldn't take the same approach as Forge of treating everything as a set and performing the most generic set operation possible on them. Where possible, we ought to keep elements as elements and not cast them into singleton sets, since cutting this corner in translation necessarily comes at the cost that the output of the translation is more complicated.

The outline of our solution is to consistently emit only the simplest (and most type-tailored) translation possible, leveraging the fact that we know at the time of translation all types of inputs into an operator. We implement this through Lean's type class system, where a set of methods can be implemented across different types and dispatched according to the type of its arguments. For every pair of types for which a method might be different (in other words, overloaded), we can write an instance of that type class implementing its functionality.

For example, the following is an excerpt[20] of our implementation of relational join[21] as a type class `HJoin` ('has join'), following our implementations of join from the examples demonstrated above:

[19] Under the hood, `Set.Subset s₁ s₂` is defined as `∀ a, a ∈ s₁ → a ∈ s₂`.

[20] There is an instance for every pair of arities and types that could be passed into a join, hence there are many more instances than shown here. However, these implementation details are obscured to the end-user since the join function `HJoin.join` will only resolve to a single instance.

[21] We use "⋈" to denote the relational join operator. If $x : A \to B$ is a relation and $y : B \to C$ is a relation, then $x \bowtie y$ produces the relation $A \to C$ merged on common values in the rightmost (B) column of x and the leftmost (B) column of y. x and y can be of arbitrary arity, so long as their leftmost and rightmost columns respectively match. That is, on n-ary relation A and m-ary relation B,

$$A \bowtie B := \{(a_n, \ldots, a_{n-1}, b_2, \ldots, b_m) \mid \exists x, (a_1, \ldots, a_{n-1}, x) \in A \wedge (x, b_2, \ldots, b_m) \in B\}.$$

```
class HJoin (α : Type) (β : Type) (γ : outParam Type) :=
  (join : α → β → γ)

-- Join singleton with arity-2 relation
@[reducible, simp] instance {α β : Type} : HJoin (α) (α → β → Prop) (β → Prop) where
  join := fun a g ↦ g a

-- Join set with arity-2 relation
@[reducible, simp] instance {α β : Type} : HJoin (α → Prop) (α → β → Prop) (β → Prop) where
  join := fun l r b ↦ ∃ a : α, l a ∧ r a b
```

Then, when translating `a ⋈ b`, we can indiscriminately produce `HJoin.join a b` and have Lean synthesize which particular implementation to apply based on the types of `a` and `b`. This allows us to have the most specific translation of an expression depending on the types of operands. The `simp` attribute on the instances allows the Lean simplifier (calling the `simp` tactic) to consult this as an unfoldable definition at proof-time and we are left with the native meaning.

For many operators on expressions (see section 6.2 below), their implementations in Lean are overloaded to accommodate the fact that Lean prefers elements when they are elements and sets only when necessary, contrasted to Forge's 'everything-is-a-set' approach. This allows us to produce semantically equivalent translations that are more simplified when possible leveraging Lean's type class system that can determine types of operands at the time of translation.

6.2. Relational Joins, Cross, Inclusion, Equality

We need to take special care when implementing expression operators in Lean whenever one of these conditions is true:

(1) There is no direct out-of-the-box translation for a Forge operation within Lean, or

(2) there are several implementations of a Forge operation in Lean, depending on the types of the operands given, and where the most generic might not necessarily be the simplest.

We discuss (2) extensively in section 6.1, and introduce using Lean's type class system to implement varying translations of a method depending on the input types. There are several other Forge operations that require this treatment: membership, join (introduced above), cross, and equality.

The specific operators that we needed to take special care translating, the different types that they permit, and their translations in Lean are detailed below in table 4.

When we do specify special Forge operators, we additionally specify custom infix operators for our operations to pretty-print in the Lean infoview window (see fig. 8 for an example):

```
infix:50 " ⋈ " => HJoin.join
```

so translated statements look like `a ⋈ b` instead of `Forge.HJoin.join a b`.

Table 4. Forge binary operators and corresponding implementations based on operand types.

Forge Operator	**Possible Types** (singletons lowercase, sets uppercase)	**Lean Implementation**
Membership: `a in b`	`a in b`	`a = b`
	`a in B`	`a ∈ B`
	`A in b`	`A = Set.singleton b`
	`A in B`	`A ⊆ B`
Equality: `a = b`	`a = b`	`a = b`
	`a = B` or `A = b`	`Set.singleton a = B`, or vice versa.
	`A = B`	`A = B`
Join: `a.b` or `b[a]`	`a.B`	`B a`
	`A.B`	*Varies, see section 6.1.*
Cross: `a->b`	`a->b`	`(a, b)`
	`a->B` or `A->b`	*Varies*, like `λ a f a' b ↦ a = a' ∧ f b`
	`A->B`	`λ f g a b ↦ f a ∧ g b`

6.3. Boundedness of Forge Sigs

Recall as summarized in section 5 that Forge Sigs get (most naturally) translated to Lean Types. However, we need to be cautious about using this as a drop-in replacement for the concept of sigs. While we don't have soundness and completeness guarantees, we ought to feel confident that our translation preserves the semantics of Forge faithfully[22].

One semantic difference in translating Forge Sigs into Lean Types directly is that we lose all sig 'bounded guarantees' that came with Forge. Since Forge compiles to a bounded SAT problem, it operates under the assumption that all sigs are finitely bounded. This means that we can rely on the assumption that sets of sigs are all finite sets.

For example, we could write a specification of a graph with an injective `next` function and the existence of a root node that is not in the image of `next`.

```
sig Node {
  next: one Node
}

pred injective {
  all a, b : Node |
    a.next = b.next => a = b
}

pred someRoot {
  some r : Node |
    no next.r
}
```

For any number of `Node` sigs we initialize Forge with (that is, for any bandwidth), Forge will be able to tell us that `¬(injective ∧ someRoot)` is theorem (that is, no such relation can exist).

Yet, in the Lean formulation of this specification, we realize that `injective ∧ someRoot` could be true! If we tried to prove the same in Lean, we realize that it isn't actually possible to prove the injectivity of our next relation. In fact, if we conjured our type `Node` with the same structure as $\mathbb{N}$, we would have a perfectly valid `next` function (`succ`) that is injective and 0 has no predecessor.

The disparity between the Forge and Lean models is that while Forge models are bounded, Lean makes no assumption about the size of models or types and requires us to make explicit statements of the finiteness of types. For the semantics of Lean to match that of Forge being finite, we need guarantees that any types from Lean sigs are finite types.

Our solution is to include additional local instance axioms for every opaque sig that is translated from Forge and introduced to our Lean environment. In this case:

```
@[instance] axiom inhabited_node : Inhabited Node
@[instance] axiom fintype_node : Fintype Node
```

[22] See section 8.2 for a discussion of formal guarantees.

which gives us guarantees that `Node` is both inhabited[23] and contains a finite number of elements within the type. To illustrate, our proof of `injective ∧ someRoot` would utilize the pigeonhole principle and appeal to the fact that `Node` is finitely inhabited, and that for a root node to exist there must be a node that has 2 predecessors.

We discuss further in section 6.6 how these instances enable us to perform integer and cardinality operations on sigs and translated Forge expressions.

6.4. Sig Inheritance and Quantifiers

Inheritance

Many signatures in Forge have complex inheritance structures [24] that cannot be expressed in Lean using a naïve translation. Recall that conventionally, we would translate a signature in Forge into a corresponding opaque type in Lean as follows (see section 5):

```
sig Student {}
```

```
opaque Student : Type
```

However, how then would we represent another sig like `Undergrad` which inherits from a `Student` sig? In Forge, we can use the `extends` keyword to denote that `Undergrad` inherits fields from `Student`:

```
sig Undergrad extends Student {}
```

For `Undergrad` to *extend* `Student`, every field accessible to `Student` must also be available to `Undergrad`, and any expression of the `Undergrad` type should be interchangeable as expressions of type `Student`.

As we did before, we could try to define the corresponding type in the same way, without any regard to the fact that it inherits from such a parent sig:

```
opaque Undergrad : Type
```

However, Lean does not know that all `Undergrad`s are also `Student`s, and since fields are typed to the sig that they are a part of in Lean[24], any access into a field that belongs to the `Undergrad` sig inherited because it was part of the `Student` sig would fail. Consider the following Forge program and the wishfully translated Lean equivalent:

```
sig Class {}
sig Student {
  registration : set Class
}
sig Undergrad extends Student {}

fun ugradsIn[c : Class] : Undergrad {
  all u : Undergrad |
    c in u.registration
}
```

```
opaque Class : Type
opaque Student : Type
opaque registration : Student → Class → Prop

opaque Undergrad : Type

def ugradsIn (c : Class) : Set Undergrad :=
  ∀ u : Undergrad,
    registration u c -- Type error!
```

[23] We are required to do this to, say, create functions and subtypes on these types. See section 6.4 for how we handle the `abstract` quantifier.

[24] For sig `A` to have a field `f : set A` is for the field `f` to have type `A → A → Prop`.

Such a Lean translation would raise a type error at line 9 above as `registration` expects an object of the `Student` type for its first input but was given a `Undergrad` type instead. In the case of Forge, Forge is aware that all descendents of a particular sig can be used interchangeably when an expression of that sig is expected. We need to find a (clever) way to encode within Lean that in fact, `Undergrad` is a child sig of `Student` and all `Undergrad`s are `Student`s. As Lean is not 'object-oriented' in the way that Forge is, there is no directly equivalent concept of a type that inherits from another type in Lean natively.

This task has its subtleties and at the same time, we will need to keep usability in mind for an end-user who wishes to prove facts about their model. One direct solution motivated by our type error might be to introduce a coercion instance from `Undergrad`s to `Grad`s which immediately fixes our problem:

```
@[instance] axiom coe_undergrad_student : Coe Undergrad Student
```

The code snippet above would type check, and we would instantly be able to refer to the child sig in place of its parent sig. However, if we wish to query in a proof whether a `Student` object is an `Undergrad` object as well via a predicate like `IsUndergrad`[25], this becomes burdensome and involved:

```
def IsUndergrad (s : Student) : Prop := ∃ x : Undergrad, x = s
```

The existential which quantifies over all `Undergrad`s to check if they are equal to `s` is not very user-friendly and can become significantly involved, especially when we are utilizing inheritance liberally in a specific model. Furthermore, we have no straightforward solution given `(IsUndergrad u)` and `(u: Student)` to cast `u` back into an `Undergrad` type.

Instead, we can consider switching the order we define the child type and child type predicate to the dual of the translation above. If instead we define our membership predicate `IsUndergrad` first:

```
opaque IsUndergrad : Student → Prop
```

we can then use Lean's native subtyping to define our `Undergrad` type:

```
@[reducible] def Undergrad : Type :=
  { s : Student // IsUndergrad s }
```

In this implementation, we also happen to get the `Undergrad` to `Student` coercion automatically as a property of subtyping.

Abstract Sigs

In addition, Forge introduces the concept of *abstract* sigs [24]. If `Student` were an abstract sig, we might encounter `Undergrad` and `Grad` student as concrete subtypes of `abstract Student`, like:

[25] This is the `Undergrad` membership predicate on `Student`s, which we'll likely need to do to prove anything about objects within this inheritance relation.

```
abstract sig Student {}
sig Undergrad extends Student {}
sig Grad extends Student {}
```

which is to say that every object instance of `Student` had ought to be either a `Undergrad` or `Grad`. Within our framework, this can be implemented in Lean as

```
axiom abstract_student : ∀ s : Student, IsUndergrad s ∨ IsGrad s
```

provided both subtype instances for `Undergrad` and `Grad` have been generated correctly.

To inform Lean that our subtypes are distinct and unique, for each pair of subtypes, we generate an axiom with the `simp` modifier that states each subclass is disjoint:

```
@[reducible,simp] axiom disjoint_Student_IsUndergrad_IsGrad :
  ∀ s : Student, ¬ (IsUndergrad s ∧ IsGrad s)
```

One Sigs

If furthermore, `Undergrad` and `Grad` are `one` sigs without fields, they are not generated as subtypes but instead are generated as opaque elements of `Student`, with their membership predicates being defined as equality:

```
opaque Undergrad : Student
def IsUndergrad (s : Student) := (Undergrad = .)
```

For `one` sigs with fields, we still need the sig to be a type in Lean so we can construct the fields relating to that sig. In this case, we have defined a type class of `One` that contains the single element `One.one` and a proof that all elements of this type are equal to this element:

```
class SigQuantifier.One (α : Type) :=
  one : α
  allEq : ∀ x : α, x = one
```

and we also introduce a corresponding coercion from a `One α` to the single element of type `α`, which is discussed further in section 6.5.

Processing Sigs

Such an analysis and translation of inheritance requires a global processing of the Forge program in Lean, since every expression emitted by our elaborator ought to have well-defined types and terms (that is, cannot be waiting for a term to be defined). Since Forge fields can depend on any sig, we need to 'lift' sigs to define first before all fields and predicates can be defined. Furthermore, sig quantifiers and inheritance structures require preprocessing all sigs to generate a topological sort of the inheritance structure before expressing any of their translations in Lean. Since Forge is not a

'local' language and all sig declarations are lifted, this was not a problem for us before we entered Lean. As such, sigs can be specified in any order in Forge.

For the remaining of sig quantifiers like `one` and `lone`, because of their complex interactions with inheritance (a `one` sig means that there is only one inhabited member of that sig that *is not* any child sig, contrary to our intuition as to what a `one` or `lone` sig should be), our program prompts the user to write a customized axiom in Lean expressing their desired constraint. Note that this utilizes the seamless integration of Forge into Lean which makes such a solution of mixed execution possible. Anecdotally, `one` and `lone` sigs only apply to child sigs and the `abstract` quantification is only used on parent sigs (it doesn't make sense for a non-inherited sig to be `abstract`), so we expect that manual handling of sig quantifications to be an edge case.

6.5. Typing & Type Coercions

While Forge treats all expressions as sets (see section 6.1), there can often be several ways to represent a set-like object in Lean.

The most common representation of a set of type `α` is `α → Prop`, which can be thought of as the membership predicate representing that set. This is oftentimes useful since checking set membership of `x : α` on set `s : α → Prop` is just a functional application: `s x`. This is also the canonical representation of sets in Lean: a `Set α` is definitionally equal to `α → Prop`, and `a ∈ b` is definitionally equal to `b a`. These representations can, for the most part, be used interchangeably.

Singletons as Sets

However, there are additional expressions that Forge sees as sets but Lean does not. An element `x : α` is a singleton set, which we discuss extensively in section 6.1. When possible, we can use type classes to 'tailor' an operation to whether an expression is a singleton or a true set; and the different implementations are detailed in section 6.2.

However, we need an 'escape plan' if no such tailored implementation exists. Hence, we include a coercion from all singletons to sets of singletons. At worst, we can treat individual elements as the set containing just them[26]:

```
instance : Coe α (α → Prop) where
  coe := Eq
```

Sigs as Sets

Additionally, a translated sig `(α : Type)` is expected to denote the set of all elements in that sig. For example:

[26]The implementation of `coe` has type `α → α → Prop`, which we can implement point-free—and there is a desire for syntactically simpler translations which are easier to work with in the Lean environment.

```
1 pred isAFriend[s: Student] {
2   s in Student.friends
3 }
```

is the predicate that `s` is *someone*'s friend, where the set of all friends is the join of `Student` ⋈ `friends` (which is equivalent to the set comprehension expression `{s: Student | some t: Student | s in t.friends}`). Here, `Student` is being used to denote the *type* of `s` on line 1 and the set corresponding to type `Student` on line 2.

In our translation, we want to be able to use the type `Student` interchangeably in places that expect a set of type `Student` as the set of all `Students`. Since we included our finiteness `Fintype` property as an instance when translating sigs (see section 6.3), we can create a coercion that coerces a Lean `Type`, given that it is a finite type, into a set of that type using the definition of the `Fintype` typeclass.

```
instance [f: Fintype α] : CoeDep Type (α : Type) (Set α) where
  coe := (f.elems : Set α)
```

When a sig is quantified `one` (see section 6.4), we also have a corresponding coercion of that sig into the single value that inhabits it:

```
@[reducible, simp] instance [o: SigQuantifier.One α] : CoeDep Type (α : Type) α where
  coe := o.one
```

Multi-arity Sets

Furthermore, there are additionally more ways of representing multi-arity relations in Lean that aren't immediately interchangeable. Where `Set α` and `α → Prop` are definitionally the same type, `Set (α × β)` and `α → β → Prop` are not. For each arity, we need to make use of either `Function.curry` and `Function.uncurry`, or custom coercion functions, to interchange between the two, for example:

```
instance : Coe (Set (α × β)) (α → β → Prop) where
  coe := Function.curry

instance : Coe (α → β → Prop) (Set (α × β)) where
  coe := Function.uncurry

instance : Coe (Set (α × β × γ)) (α → β → γ → Prop) where
  coe := fun s ↦ fun a b c ↦ (a, b, c) ∈ s

instance : Coe (α → β → γ → Prop) (Set (α × β × γ)) where
  coe := fun r ↦ {p : α × β × γ | r p.1 p.2.1 p.2.2}
```

Future work will involve attempting to remove cross products and standardize all multi-arity sets, which are currently introduced by Forge cross products (see section 6.2).

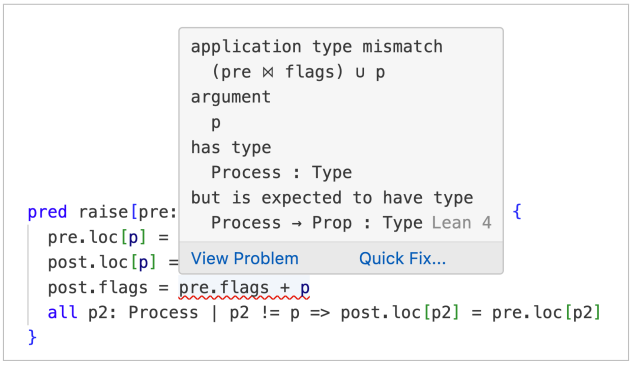
Explicit Coercions

The Lean elaborator, which is responsible for type unifications, is occasionally unable to find such coercions we have introduced due to having many possible paths of coercion. This is further exacerbated by the fact that our type might appear in an expression where we have encoded several implementations (like equality, see section 6.2).

In situations such as these, we might encounter type errors even when coercions should have been taken, such as

```
pred raise[pre: State, p: Process, post: State] {
  pre.loc[p] = Uninterested
  post.loc[p] = Waiting
  post.flags = pre.flags + p -- Type error!
  all p2: Process | p2 != p => post.loc[p2] = pre.loc[p2]
}
```

which produces the following error[27]:



To solve this, we retrofit syntax onto Forge that allows us to explicitly introduce casts where needed to provide Lean with additional type hints that it can utilize in type unification. We can cast `p` explicitly, which is a `Process` type, into a `Set Process` (the same as `Process → Prop`) type using the following syntax:

```
post.flags = pre.flags - p /* as Set Process */
```

which also coincides with Forge comments to preserve interoperability.

Future work on the translated type system will involve improving coercions and operations that reduce or completely eliminate the need for explicit type annotations.

6.6. Integers

Forge and Alloy come with a unique integer model—since models are compiled down to boolean constraints to be solved by a SAT solver, integers are severely limited in their bitwidth [24, 44]. The default bitwidth on integers in Forge is 4, which gives us a total of 16 integers.

[27] This example is taken from section 7.3.

Some programs will inadvertently utilize integer overflow which affects their model in meaningful ways, oftentimes counterintuitively [45, p. 22]. However, the prevailing documentation on integers in the Forge and Alloy solvers casts this effect as a necessary compromise in the design of the language architecture, and placing a bitwidth on integers is an unavoidable consequence due to the boolean formula-based backend of the language.

This is an area where Lean shines. Lean has an integer model which is defined using an honest-to-goodness inductive model for the natural numbers [5]. Lean's integer model is arbitrary precision and designed for proofwriting and numerical reasoning. This includes being able to reason about integers and write functions involving integers that are noncomputable, such as computing the cardinality of a set.

Here, we depart briefly from the convention that we've been following so far of reproducing Forge as accurately as possible in favor of both more extensive integer support, as well as reduced complexity in implementing integers within our translation. For the most part, we can use Lean's integer and finite set/types libraries out of the box with little modification.

Our translated Lean models treat all integers as expressions, which makes the translation from an integer expression in Forge to an integer in Lean relatively straightforward.

Our model is semantically equivalent to running Forge with an arbitrary bitwidth, more than the model would ever exceed and overflow. This gives the most accurate translation of what Forge tries to achieve with integers but is not technically capable of doing.

Here are two examples adapted from [24] that showcase some of the integer features in Forge.

In Forge, the `#` keyword, like on line 4 below, denotes the cardinality of a set.

```
1  sig Suit {}
2  sig Card { suit: one Suit }
3  pred threeOfAKind[hand: set Card] {
4    #hand.suit = 1 and #hand = 3
5  }
```

Fields of sigs can also be integers, and we can do arithmetic on them. By treating all integers as first-class expressions[28], we can also use integers in fields alongside integers that are the result of a set computation. For example, we could define a weighted graph with weighted edges *and* nodes:

```
1  sig Node {
2    node_weight: one Int
3  }
4  sig Edge {
5    start: one Node,
6    end: one Node,
7    edge_weight: one Int
8  }
9  pred nodeWeightIsEdgeWeightPlusOne[n: Node] {
10   n.node_weight = add[1, sum e: { l: Edge | l.start = n } | { e.edge_weight }]
11 }
```

[28] Forge denotes integers as atoms or values depending on whether an integer appears in a field or as a result of a computation, but casts seamlessly between [41, 44].

On line 10, we are defining a predicate that states a node n's weight is equal to 1 plus the sum of edge weights of those edges that start at n.

Of the integer operations, we can easily translate arithmetic operations (addition, subtraction, integer division, remainder, absolute value and sign) as well as inequalities directly into their integer equivalent in Lean. What requires more effort are the notions of counting (cardinality) and quantification in Forge as exemplified above.

We approach the cardinality problem by using `Set.ncard`[29]. While this function has a junk value when a set is infinite, we had remedied this earlier in section 6.3 by including `Fintype` axioms with every Forge type we introduce. Lean knows that every set of a `Fintype` is a `Finset` and has an honest-to-goodness cardinality. This allows us to implement sum, max, min, and a summation with a binder (see line 10 of the graph example above) using Lean `Finset` methods such as `Finset.sum`, `Finset.max`, etc.

To illustrate, the translations of the two predicates (playing card hand and graph) above in Lean, eliding sig and field translations, would be as follows:

```
def threeOfAKind (hand : Set Card) : Prop :=
  Set.ncard (hand !$\bowtie$! suit) = 1 ∧ Set.ncard hand = 3
```

and

```
def nodeWeightIsEdgeWeightPlusOne (n : Node ) : Prop :=
  node_weight n = 1 + Finset.sum { e : Edge | start e = n } edge_weight -- See footnote [30]
```

While we did need to retrofit additional instance axioms for each type generated to make an integer model work, it is impressive that we were able to extract so much integer functionality out of Forge in within our limited Lean model in the first place. Implementing Forge integers within our translation is also a hallmark of the motivation behind our project in the first place—that in some cases, we can endow *additional* functionality to the Forge specification language by interpreting it in a proof assistant instead of the standard relational Forge implementation.

The complete translation of Forge integers into Lean is overviewed in table 3 earlier in section 5.

[29] More specifically, we do need to utilize the approach in section 6.1 of using type classes to implement this, since the cardinality of a singleton ought to be 1. In all other cases, `Set.ncard` is the implementation of cardinality.

[30] There are select details surrounding type coercions, universe levels, and the noncomputability of our integer functions that remain to be resolved.

7. Results and Examples

7.1. 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)[31] are designed to have highly flexible and extensible user interfaces that expose useful APIs for implementers of DSLs and custom UI to utilize [38, 40]. Furthermore, Lean's extensible syntax and macro system are simple yet remarkably powerful [57, 49].

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 can interact with Lean's 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 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. 3, the syntax of a Forge specification is color-coded.

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 the type of the expression. In fig. 3, 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 [45], 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.

[31] 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.

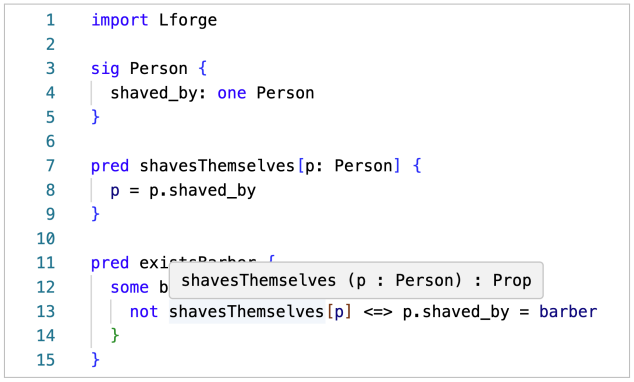


Figure 3. Tooltips containing type information are available on hover. Forge syntax is automatically highlighted without any extra work.

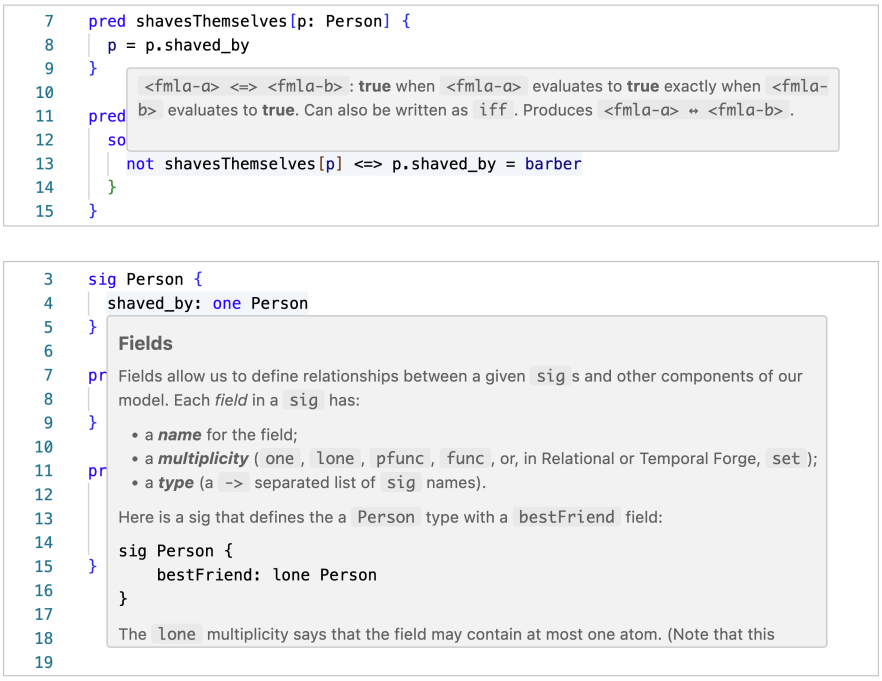


Figure 4. We can define our syntax definitions to print with custom documentation text for users new to using Forge syntax.

Forge documentation is included via docstrings that are placed inline with our syntax objects (see section 4.2), which is automatically included by Lean's LSP to display on the front end. Figure 4 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 prompt users attempting to use unsupported language features with clarification and a request to redefine their statements. Figure 5 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 an LSP, users immediately get immediate feedback on whether their source code parses and 'compiles'.[32]

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 6 illustrates error reporting at the level of specific identifiers.

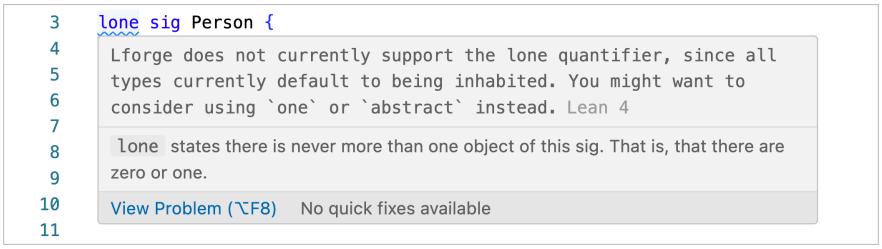


Figure 5. We can define custom error messages with our implementation to prompt users to change their specifications if a piece of syntax is ambiguous or not supported.

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 elaborator is a highly optimized algorithm that attempts to resolve type coercions, type classes, and reductions [16], it is 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 6.5, 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

[32]Forge translations in Lean are not 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 [23] and only reports type errors at runtime when the successful evaluation of expressions results in the empty expression [18]. This proves difficult to debug and unwieldy for users to understand, as [45] 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. 6, we pass the `Board` and `Player` arguments to `winRow` in the wrong order, which causes a type error. Lean can identify that the first input to `winRow`, `p`, has the wrong type and displays an appropriate error message.

```
58  pred winner[b: Board, p: Player] {
59    p != None
60    winRow[p, b]
61      application type mismatch
62        winRow p
63      argument
64        p
65      has type
66        Player : Type
67      but is expected to have type
68        Board : Type Lean 4
69
70      winRow (b : Board) (p : Player) : Prop
71      View Problem (⌥F8)    Quick Fix... (⌘.)
72  }
```

Figure 6. A Lean error message indicating a type mismatch in our Forge expression.

Mixed Execution

Our embedding of Forge within Lean, especially our choice to map Forge structures (sigs, fields, predicates) to corresponding Lean concepts (types, relations, functions) as faithfully as possible means that a Lean file with Forge specifications supports mixed execution, an embedding model explored and supported by similar tools that merge the Alloy specification into other imperative programming languages [35, 36, 34].

All Forge specifications, once inserted into a Lean file, will generate appropriate definitions directly into the Lean environment. Using Lean syntax, we can further interact with these definitions from Forge, perhaps writing predicates or definitions that build on these. This interaction goes the opposite way as well: where Forge expects an expression, predicate, or function, declarations from Lean can be used seamlessly. This provides a frictionless user experience and allows the user to add additional constraints and rules, written in Lean, to a preexisting Forge model. Furthermore, this alleviates the tension incurred of creating the perfect translation: we are aware of the fact that only a subset of Forge is implemented due to the technical restrictions of both platforms. However, with appropriate error reporting (see above Error Checking), users can be prompted to extend their Forge specifications with additional Lean rules. Mixed execution of Forge and Lean means that Forge specifications are now extensible using a much broader functional programming language.

The following is an example of mixed specification of Forge and Lean using Lforge. The full specification of this example, a model for mutual exclusion of processes, is discussed in section 7.3.

```
1  import Lforge
2
3  abstract sig Location {}
4  one sig Uninterested, Waiting, InCS extends Location {}
5
6  sig Process {}
7
8  sig State {
9    loc: func Process -> Location,
10   flags: set Process
11 }
12
13 def flags_good (s : State) :=
14   ∀ (p : Process), loc s p = InCS ∨ loc s p = Waiting → flags s p
15
16 pred good[s: State] {
17   flags_good[s]
18   lone {p: Process | s.loc[p] = InCS}
19 }
```

While the majority of this specification is in Forge, we are working in Lean using Lforge (line 1). We define relevant sigs and fields as a Forge specification. On lines 13-14, we use the types defined in Forge to write a predicate in Lean that states that all processes waiting or in a critical state have a flag raised. In line 17, we've switched back to specifying in Forge but can continue to utilize the `flags_good` predicates we wrote above in Lean. There are no walls or abstractions between the two languages, and Forge is indeed a first-class citizen in the Lean environment.

Since this interoperability works across imports and modules, we envision projects where sections of specifications can be written in Forge and other relevant sections in Lean, allowing users to interoperate between the two. This could also introduce possibilities for users accustomed to the two distinct languages or formalization techniques to collaborate on a shared specification.

7.2. 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 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:

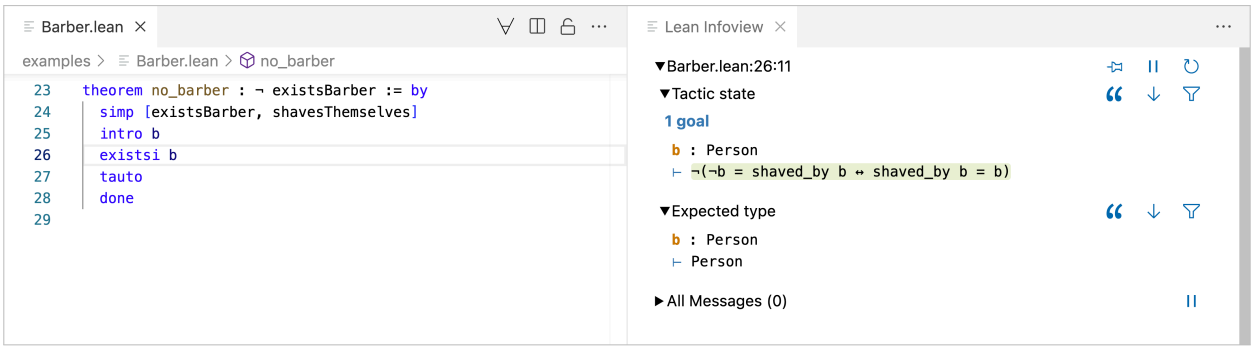


Figure 7. The proof of the nonexistence of a barber in the barber paradox, in Lean. The interactive proof state is on the right with the proof source on the left.

On line 25, we use the `simp` tactic (or 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. Full tactic states at each step of this proof are provided in appendix B.

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 7.3, presents a more elaborative example of LFORGE.

7.3. A Mutual-Exclusion Protocol

Here, we present a more comprehensive example that showcases more of the functionalities of LFORGE and hopefully motivates real-world use cases of our tool. We model a basic mutual exclusion (mutex) protocol based on one of the examples presented in CSCI 1710 Logic for Systems [42]. In the course, the example is posed with 2 competing processes over a mutex. Empowered with Forge within LFORGE, we expand the model to include any number of processes.

The example model contains `State` sigs that encapsulate the state of the entire system. Processes can have several states, `Uninterested`, `Waiting` (interested but not in critical state), `InCS` (in critical state). Each state contains a set of processes that have a flag raised demonstrating they are potentially interested in mutex. State transitions are modeled using predicates of the form

```
pred transition[pre: State, p: Process, post: State] { ... }
```

Processes have 4 transitions:

1. `raise`: they can transitions from `Uninterested` to `Waiting` by raising their flag;
2. `enter`: they can transition from `Waiting` to `InCS` provided they are the only flag raised;
3. `lower`: if there is more than one flag raised, they can transition from `Waiting` back to `Uninterested` by lowering the flag;
4. `leave`: they can transition from `InCS` to `Uninterested` when processes are done.

We define a `good` predicate that states an invariant of our model that we wish to be true. In our case, the `good` predicate stipulates no two processes can be in the critical state (have acquired the lock) on the mutex at the same time, and any process that is waiting or in a critical state has a 'flag' raised. We define an `init` state predicate that states all processes are in a state of `Uninterested` and no flags are raised. A predicate titled `properties` (users sometimes use the convention `traces`) encapsulates all properties of our system: that the initial state is good and for all pairs $\langle$`pre`, `post`$\rangle$ for which there is a transition between, `pre` is good implies `post` is good. This is to say, `good` is an invariant property given our transition rules.

To test our model and that it indeed has such desired properties, we can first run Forge on the test `properties is theorem` to check that Forge cannot find any counterexamples within its specified bounds. Then, as a next step, we can declare a theorem that states `properties` in Lean and prove our theorem.

To prove `properties`, we can split up our property into the base case (proving that the `init` state is `good`) and that each of the 4 transitions preserves `properties`. We prove each transition separately in lemmas.

An example of the tactic state during one of the proofs of one such lemma is showcased in fig. 8. We note that the tactic state in fig. 8 represents a typical Lean proof state and what a user might encounter while using LFORGE and Lean to prove specification properties, as opposed to our comically short proof in fig. 7.

However, the proof was not without friction. Throughout the proof, dealing with sets was by far the most difficult. While Forge defaults to sets and 'relations' as its primary type (see section 6.1), Lean prefers expressions that are objects. This meant that proving statements like

```
{ x | x = p } = { p' } → p = p'
```

were unfortunately more difficult than necessary. An area of exploration and further work is to develop a library of theorems, lemmas, and tactics that specifically aid in proving Forge 'set-style' statements within Forge.

For a rough reference of length, our specification is roughly 70 lines long and our proof is roughly 250 lines long, which is standard for each. Neither the specification nor proof were more involved

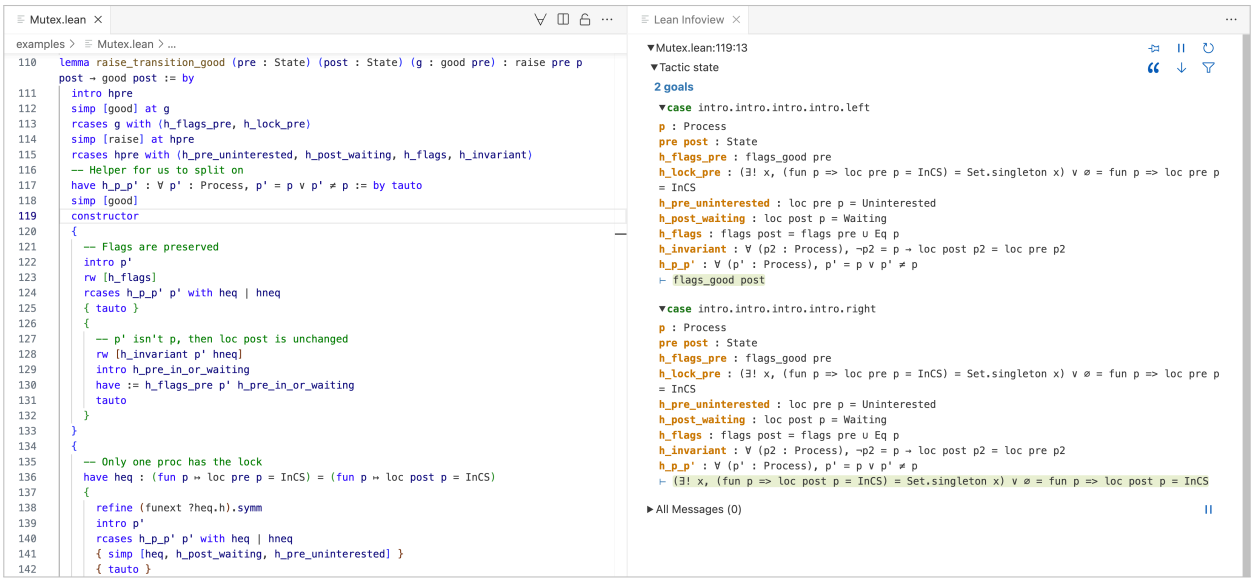


Figure 8. The Lean tactic state at one line of our proof that the `raise` transition is sound.

than had they been solely in Forge or Lean respectively. The full source of this example is referenced in appendix C.

7.4. Further Examples

We provide three further examples (without proofs), to illustrate LFORGE's translation capabilities. Said examples translate fully using LFORGE without any type errors.

From the Logic for Systems course [42], we adapt the first Forge assignment on family trees, as well as the in-class example with a Tic-Tac-Toe board. Our examples are minimally modified specifications from the course and demonstrate LFORGE's capabilities of working with existing Forge specifications with little-to-no modification.

Additionally, inspired by an ongoing research project that uses Forge to model distributed systems algorithms [59], we also model the two-phase atomic commitment protocol. Our example is an entirely new Forge specification that takes inspiration from this existing research. This serves as a more complex example of a system that might be valuable to be modeled in Forge and proven in Lean.

The sources of said examples are referenced and specified further in appendix D.

8. Discussion

8.1. Contributions

We summarize below some of the main contributions we make:

LFORGE as a tool First and foremost, we've created a tool that contributes utility both to Forge and to Lean. By allowing Forge specifications to be ported seamlessly into the Lean theorem prover, users are empowered to complement Forge's automated search capabilities with writing proofs for conjectured properties regarding their model. This creates a 'hypothesize-and-proof' workflow for users and students to specify properties about a model specification, quickly prototype and test the validity of their models on small bounded examples, and delve deeper to proving said properties more generally. By preserving Forge's semantics, we've allowed for Forge model specifications to be 'ported' into Lean, and users can feel comfortable that they are indeed modeling within the same relational framework that they are used to.

Usability-first translation LFORGE is guided by usability and simplicity. Translations follow a simplest-first approach (see section 6.2) that focuses on implementations tailored to specific type configurations over the most general translation. Compromises are made to create a *subset* of Forge that is most easily expressible in Lean, and users are guided to make changes and annotations that aid their translation. By leveraging Lean's LSP capabilities (see section 7.1), we can also expose the most relevant type and error information for our end-user.

A Lean DSL LFORGE also serves as one of few examples of programs that utilize Lean's metaprogramming capabilities to implement a domain-specific language within Lean itself. We test Lean's exposed metaprogramming capabilities to their limits (see section 7.1), from type unification/checking, error reporting, on-hover hinting, as well as mixed specification, and produce an embedding of Forge that interacts easily with its host language.

8.2. Future Work

We recognize that there is much work that is left to be done in this project. On top of the administrative work that remains—creating a coherent set of documentation and examples for LFORGE and preparing it for general use—we detail below some of the major areas that are yet to be explored.

Formal Guarantees

Throughout this implementation, we reference to the want for our translation to be *faithful*, that is, we optimally want some form of soundness guarantee. We surely do not want to be able to prove a property about a Forge specification in Lean that doesn't hold (that is, Forge can find a

counterexample). While we acknowledge completeness is not possible[33], we still hope for model properties to be generally provable—that is, it would be conceited if we were unable to prove any LFORGE-translated specification.

Both Forge and Lean, existing atop sound logical frameworks, can be formalized as logical objects. We ought to be able to prove properties about said translation in this higher logical framework, which should give us additional confidence in our translations.

A Proper Type System for Forge

While we discuss the merits of inheriting Lean's type system in section 7.1, this process was without friction. Forge's forgiving type system is the source of a lot of conscious design choices (see sections 6.1 and 6.2) as well as workarounds (see section 6.5). In its current state, Lean's elaborator (see section 4.3), which contains the type unification algorithm, is not able to fully resolve types emitted out of Forge models due to the number of alternatives and implementations that are type-dependent.

A proper, albeit tedious, solution to this issue is to intervene before elaboration to do a specific first-pass type check and type inference that is specific to Forge. This allows us to, at translation-time, include more type hints and type annotations for Lean's elaborator, eliminating the need to manually annotate types when Lean cannot infer coercions, as in section 6.5. This type system will be tailored to the complex behaviors of Forge types, and reduce the number of metavariables emitted as a result of our translation, which is currently a main source of confusion for the Lean elaborator[34].

Toward a Comprehensive Proof & Tactic System

We discuss in section 7.3 some of the friction that our translation introduces, particularly around sets (see section 6.1) and cardinality. While we can make use of techniques such as annotating instances and axioms with the `simp` modifier, proofs with translated Forge specifications still lack behind proofs in Lean.

Mathematical proofs in Lean have the backing of `mathlib4` [38], which contain a large library of theorems pertaining to mathematical objects they describe that complement the proof process. Translated LFORGE specifications do not have such a foundation to build on. Especially where it pertains to data structures that are already sparsely supported (such as `Sets`, transitive closures, etc.) or custom implementations that we implement ourselves (such as those in section 6.2, like relational join or cross-products), many proofs are excessively cumbersome.

[33] Forge specifications are decidable because they compile to SAT, which is decidable. LFORGE translates properties into formulas in first-order logic, for which satisfiability is undecidable. One could imagine specifying a Turing machine and attempting to prove a predicate that it halts in a finite number of steps.

[34] In other words, there is currently *not enough* type information that comes out of our translation for Lean to fully figure out types, especially given the complex coercion structures and type class structures we've built out to support Forge operations.

Up until now, we have largely combatted this issue by modifying our *translation* to be more granular, specific, and simpler when possible, such that emitted translations can be as simplified as possible. Future work should turn away from this solution in favor of a comprehensive library of theorems that are generally applicable to proving facts about the style of formulas generated by Forge specifications. Specifically, we need to work on developing a library of theorems and proof tactics that cater to Forge's 'set-styled' statements. Only then will we be able to use Lean to its fullest extent complementing the automated search capabilities of Forge.

LFORGE as a Pedagogical Tool

As mentioned in section 2.1, Forge is a language with pedagogy in mind. We've also designed a tool that focuses on usability and learnability (see section 7.1) with a context and background that centers around pedagogical formal methods (see section 3.1). We are interested in exploring this side of LFORGE—that it can be used as a tool for students of different formal methods courses to bridge their learning and work on a meaningful and significant modeling project: using Forge to prototype and 'check' properties automatically and formalizing their properties using proofs in Lean. This could contribute to a more complete and comprehensive formal methods workflow for students to explore more complex and interesting problems.

8.3. Lessons Learnt

We close with some lessons learned and knowledge gained from this project.

We echo the sentiment in [22] that documentation surrounding Lean's metaprogramming capabilities is still in its beta stages. Without expertise and knowledge on the inner workings of Lean which were largely undocumented, this project would have proved to be more challenging. Code search[35], browsing the Lean Community Zulip, and trial-and-error were essential in much of the progress made.

Fairy tales do not always have happy endings. We set out to port a significant subset of Forge into Lean, and the goal was for most existing Forge specifications to interoperate with Lean out-of-the-box. As we saw in section 6 (especially sections 6.1 and 6.5), and even in our example section 7.3, the difficulties were plenty. Retrofitting a type system that already exists—Lean's—onto a largely untyped specification model is difficult!

However, we can make compromises (elegantly), as we did in section 5 excluding certain quantifiers that proved difficult to model, or introducing additional syntax as in section 6.5 to make the task of translation easier for us. Forge already has language levels that suit different levels of learning and understanding [45]. With LFORGE, Forge has gained another sublanguage that is most suited for the two-sided task of automated verification as well as manual formal verification via proofs that keep usability in mind.

[35] Of publicly available GitHub repositories.

Bibliography

[1] Jean-Raymond Abrial. "Formal methods in industry: achievements, problems, future". In: *Proceedings of the 28th international conference on Software engineering*. 2006, pp. 761–768.

[2] Konstantine Arkoudas et al. "Integrating model checking and theorem proving for relational reasoning". In: *Relational and Kleene-Algebraic Methods in Computer Science: 7th International Seminar on Relational Methods in Computer Science and 2nd International Workshop on Applications of Kleene Algebra, Bad Malente, Germany, May 12-17, 2003, Revised Selected Papers 7*. Springer. 2004, pp. 21–33.

[3] Konstantinos Arkoudas. "Denotational proof languages". PhD thesis. Massachusetts Institute of Technology, 2000.

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

[5] Jeremy Avigad et al. *Theorem proving in Lean*. 2024.

[6] Edward W Ayers, Mateja Jamnik, and William T Gowers. "A graphical user interface framework for formal verification". In: (2021).

[7] Hamid Bagheri and Sam Malek. "Titanium: efficient analysis of evolving alloy specifications". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, pp. 27–38.

[8] Paul Bernays and Moses Schönfinkel. "Zum entscheidungsproblem der mathematischen logik". In: *Mathematische Annalen* 99.1 (1928), pp. 342–372.

[9] Yves Bertot. "A short presentation of Coq". In: *Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings 21*. Springer. 2008, pp. 12–16.

[10] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[11] Jasmin Christian Blanchette and Tobias Nipkow. "Nitpick: A counterexample generator for higher-order logic based on a relational model finder". In: *International conference on interactive theorem proving*. Springer. 2010, pp. 131–146.

[12] Avik Chaudhuri et al. "Fast and precise type checking for JavaScript". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), pp. 1–30.

[13] Albert Mo Kim Cheng. "A survey of formal verification methods and tools for embedded and real-time systems". In: *International Journal of Embedded Systems* 2.3-4 (2006), pp. 184–195.

[14] Łukasz Czajka and Cezary Kaliszyk. "Hammer for Coq: Automation for dependent type theory". In: *Journal of automated reasoning* 61 (2018), pp. 423–453.

[15] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[16] 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.

[17] Richard St-Denis. "A comparison of three solver-aided programming languages: αRby, ProB, and Rosette". In: *Journal of Computer Languages* 77 (2023), p. 101238.

[18] 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.

[19] Burak Ekici et al. "SMTCoq: A plug-in for integrating SMT solvers into Coq". In: *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II 30*. Springer. 2017, pp. 126–133.

[20] George Fink and Matt Bishop. "Property-based testing: a new approach to testing for assurance". In: *ACM SIGSOFT Software Engineering Notes* 22.4 (1997), pp. 74–80.

[21] Zheng Gao, Christian Bird, and Earl T Barr. "To type or not to type: quantifying detectable bugs in JavaScript". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE. 2017, pp. 758–769.

[22] Vladimir Gladshtein, George Pîrlea, and Ilya Sergey. "Small Scale Reflection for the Working Lean User". In: *arXiv preprint arXiv:2403.12733* (2024).

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

[24] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[25] Philipp Körner and Florian Mager. "An embedding of B in Clojure". In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 2022, pp. 598–606.

[26] Felix Kossak and Atif Mashkoor. "How to select the suitable formal method for an industrial application: a survey". In: *International conference on abstract state machines, alloy, b, tla, vdm, and z*. Springer. 2016, pp. 213–228.

[27] Sebastian Krings et al. "A Translation from Alloy to B". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 6th International Conference, ABZ 2018, Southampton, UK, June 5–8, 2018, Proceedings 6*. Springer. 2018, pp. 71–86.

[28] Thierry Lecomte et al. "Applying a formal method in industry: a 25-year trajectory". In: *Formal Methods: Foundations and Applications: 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29—December 1, 2017, Proceedings 20*. Springer. 2017, pp. 70–87.

[29] Robert Y Lewis and Minchao Wu. "A bi-directional extensible interface between lean and mathematica". In: *Journal of Automated Reasoning* 66.2 (2022), pp. 215–238.

[30] Jannis Limperg and Asta Halkjær From. "Aesop: White-box best-first proof search for Lean". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2023, pp. 253–266.

[31] David R MacIver, Zac Hatfield-Dodds, et al. "Hypothesis: A new approach to property-based testing". In: *Journal of Open Source Software* 4.43 (2019), p. 1891.

[32] Petra Malik, Lindsay Groves, and Clare Lenihan. "Translating z to alloy". In: *Abstract State Machines, Alloy, B and Z: Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings 2*. Springer. 2010, pp. 377–390.

[33] Leonid Mikhailov and Michael Butler. "An approach to combining B and Alloy". In: *ZB 2002: Formal Specification and Development in Z and B: 2nd International Conference of B and Z Users Grenoble, France, January 23–25, 2002 Proceedings 2*. Springer. 2002, pp. 140–161.

[34] Aleksandar Milicevic et al. "Advancing declarative programming". PhD thesis. Massachusetts Institute of Technology, 2015.

[35] Aleksandar Milicevic et al. "Executable specifications for Java programs". MA thesis. Massachusetts Institute of Technology, 2010.

[36] Aleksandar Milicevic, Ido Efrati, and Daniel Jackson. "α Rby—an embedding of Alloy in Ruby". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings 4*. Springer. 2014, pp. 56–71.

[37] Aleksandar Milicevic et al. "Alloy*: A general-purpose higher-order relational constraint solver". In: *Formal Methods in System Design* 55 (2019), pp. 1–32.

[38] 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.

[39] David Musser and Aytekin Vargun. *Proving theorems with Athena*. 2003.

[40] 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.

[41] Tim Nelson. *Forge Language Documentation*. https://csci1710.github.io/forge-documentation/home.html. 2024.

[42] Tim Nelson. *Logic for Systems*. 2024. URL: https://csci1710.github.io/book/.

[43] Tim Nelson, Andrew D Ferguson, and Shriram Krishnamurthi. "Static differential program analysis for software-defined networks". In: *FM 2015: Formal Methods: 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings 20*. Springer. 2015, pp. 395–413.

[44] Tim Nelson et al. *Artifact for Forge: A Tool and Language for Teaching Formal Methods*. Version 0.3. Jan. 2024. DOI: 10.5281/zenodo.10463960. URL: https://doi.org/10.5281/zenodo.10463960.

[45] 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.

[46] Tim Nelson et al. "Tierless programming and reasoning for {software-defined} networks". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 2014, pp. 519–531.

[47] Timothy Nelson et al. "The Margrave tool for firewall analysis". In: *24th Large Installation System Administration Conference (LISA 10)*. 2010.

[48] Timothy Nelson et al. "Toward a more complete Alloy". In: *Abstract State Machines, Alloy, B, VDM, and Z: Third International Conference, ABZ 2012, Pisa, Italy, June 18-21, 2012. Proceedings 3*. Springer. 2012, pp. 136–149.

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

[50] Lawrence C Paulson. *Isabelle: A generic theorem prover*. Springer, 1994.

[51] Frank P Ramsey. "On a problem of formal logic". In: *Classic Papers in Combinatorics*. Springer, 1987, pp. 1–24.

[52] Jan Oliver Ringert and Syed Waqee Wali. "Semantic comparisons of alloy models". In: *Proceedings of the 23rd acm/ieee international conference on model driven engineering languages and systems*. 2020, pp. 165–174.

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

[54] J Michael Spivey and Jean-Raymond Abrial. *The Z notation*. Vol. 29. Prentice Hall Hemel Hempstead, 1992.

[55] Emina Torlak and Daniel Jackson. "Kodkod: A relational model finder". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2007, pp. 632–647.

[56] Emina Torlak et al. "Applications and extensions of Alloy: past, present and future". In: *Mathematical Structures in Computer Science* 23.4 (2013), pp. 915–933.

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

[58] Sebastian Andreas Ullrich. "An Extensible Theorem Proving Frontend". PhD thesis. Dissertation, Karlsruhe, Karlsruher Institut für Technologie (KIT), 2023, 2023.

[59] Jinliang Wang and Tim Nelson. *Forge for Distributed Systems*. https://github.com/jinlang226/Forge-for-Distributed-System. 2024.

[60] Jim Woodcock et al. "Formal methods: Practice and experience". In: *ACM computing surveys (CSUR)* 41.4 (2009), pp. 1–36.

[61] Jiayi Yang et al. "AlloyMC: Alloy meets model counting". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 1541–1545.

Appendices

A. Data Availability

The source code for LFORGE, including all mentioned examples and proofs, is publicly available at this repository: https://github.com/jchen/lforge. The source code at the time of this thesis being submitted is tagged `thesis`. LFORGE is available as a package and can be included as a dependency using Lake.

Check tag.

B. Barber Paradox Proof

The proof of the barber paradox annotated with the Lean tactic state after each tactic/step is provided below. This is also provided at this path in the code repository: `examples/Barber.lean`.

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

C. Mutual Exclusion Protocol Specification & Proofs

The Forge specification and Lean proofs for the mutual exclusion protocol described in section 7.3 are provided at this path in the code repository: `examples/Mutex.lean`. The specification spans lines 14–85 and proofs span lines 89–336.

The Forge specification for this example is taken from [42] with slight modifications for more than two processes. We contribute the entirety of the Lean proof of correctness of this protocol.

D. Additional Examples

Additional examples are also provided in the `examples` directory.

Two examples, Tic-Tac-Toe and 'Grandpa', are based on course content from Logic for Systems [42] with minimal modifications. They serve solely to illustrate the capabilities of LFORGE on translating existing programs, and we do not claim to make additional contributions to these

models. Tic-Tac-Toe is at the following path: `examples/Board.lean`, and the 'Grandpa' is at the following path: `examples/Grandpa.lean`.

The specification regarding the two-phase atomic commitment protocol is inspired by [59] but does not use any code directly from the repository. This serves as an example of a more complex distributed system protocol that is translatable. The two-phase commitment protocol example is at the following path: `examples/TwoPC.lean`.

Colophon

This document is typeset using XeLaTeX with the `scrbook` document class. The bibliography is processed using Biblatex. Source code listings utilize the `minted` package and syntax is highlighted using *Pygments*.

Serif text is set in Computer Modern, sans serif text is set in MVB Solitaire Pro, and code is set in `DejaVu Sans Mono`.