Title TBD

Jiahua Chen

April, 2024

Abstract

Acknowledgements

Notation

Throughout the paper, you might encounter some visual or symbolic notation that is specific to this project or specific to the semantics of Forge and/or Lean. These are noted here.

“$\bowtie$” denotes 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.

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.

```
-- 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?
```

is an example of a Lean implementation code block. 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.

```
-- 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
}
```

is an example of a Forge code block. This denotes examples of a model (or a snippet of a model) in Forge.

```
-- 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
```

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

Contents

§1 Introduction

Formal methods are increasingly being applied in industry and 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, 12]. These applications prove invaluable when the functionality of an existing yet 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 [19].

Yet, there is a multitude of tools and flavors that exist in the realm of formal methods: type-checked programming languages [8, 4], property-based testing frameworks [7, 13], modeling and specification languages [9, 10, 17], SMT solvers [6], proof assistants [14], etc. Each of these tools offers a tailored set of features and is based upon specific yet different logical frameworks. 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 [19]. This problem of selecting tools is so prevalent that there have been surveys and methodologies devised for this task [11, 5].

This paper introduces LFORGE[1], a tool that implements the Forge specification language [17] (a pedagogical offshoot of Alloy [9], 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, a graphical proof assistant [14].

In doing such, LFORGE aims to serve as 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 resolving capabilities of multiple formal methods models, 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) [10, 17]. 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] [2]. This extension of the Curry-Howard correspondence provides a trans-

[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 [3].

lation of mathematical proofs into computer programs (terms in simply-typed lambda calculus). Systems are implemented within the functional programming language, and 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]. By allowing users to input Forge directly into a Lean source program, users can on one hand harness the automated reasoning tools that Forge provides, yet circumvent any bound limitations of Forge by proving theorems 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 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.

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

§2 Background

§2.1 Related Work

§2.2 *Lean* and other proof assistants

§2.3 *Forge*, *Alloy*, and other relational specification languages

§3 Motivation

§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 of the fact that Lean 4 is mostly implemented in itself. As users, we are able to utilize and emit the same data structures used in Lean's own implementation to extend the functionality of Lean [14]. These metaprogramming capabilities of Lean make our work implementing a Forge module in Lean much easier.

The Lean 4 compilation process is structured as follows:

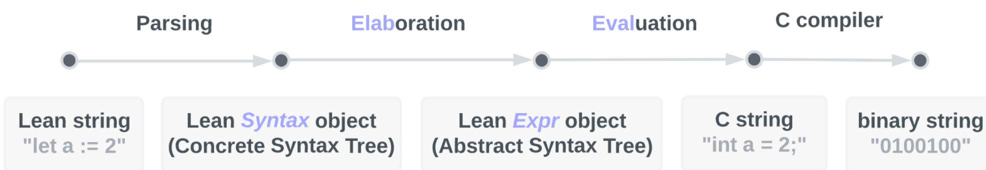

Figure 1: A diagram from [18] summarizing the Lean 4 compilation process.

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 new Lean syntax objects that correspond to a concrete syntax tree 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).

As a result, there 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[4]. complete?

§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 Lean `Syntax` 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 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.

[4]We are very fortunate that there are no conflicts between Forge and Lean syntax that would hinder this.

2. By defining Forge 'blocks' as a Lean `command`—which is the top-level syntax category[5]—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 term 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). variable?

For example, the grammar[6] of Forge arguments and predicates is:

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

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

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

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 are able to translate the entirety of the grammar of Forge[7] into Lean syntax definitions.

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
```

§4.3 Elaboration

Complete

§4.4 The Forge Model in Lean

Complete

[5] For example, a definition "`def x: Int := 0`" is a Lean command.

[6] Where ,+ and ,∗ denote one/zero or more comma-separated occurrences respectively. + and ∗ denote one/zero or more repetitions.

[7] At least, a useful subset of the Forge language we care about. This is based off the grammar of Alloy [9, 10, 17].

§5 Challenges and Implementation Details

§5.1 Finiteness of Forge Sigs

Recall as summarized in section 4.4 that Forge Sigs get (most naturally) translated to Lean Types. However, we need to be cautious 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 semantic of Forge faithfully.

One semantic difference in translating Forge Sigs into Lean Types directly is that we lose all sig 'finiteness guarantees' that came with Forge. Since Forge compiles to a bounded SAT problem, 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 path through a graph with one source node and claim that this path will never be injective:

```
sig Node {
  next: one Node
}

pred notInjective {
  not ( all a, b: Node |
    a.next = b.next => a = b )
}
```

For any number of `Node` sigs we initialize Forge with (that is, for any bandwidth), Forge will be able to tell us that `notInjective` is theorem.

Yet, in the Lean formulation of this specification, we realize that `notInjective` need not be true at all. 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 (think `succ`) that is not injective.

The disparity between the Forge and Lean models is that while Forge models are finitely inhabited (albeit, arbitrarily so), Lean makes no assumption about the size of models or types and requires explicit statements of finiteness of types. Pertaining to the example above, we would want to be able to prove the nonexistence of such an injective function using the translated facts produced by our module.

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
```

which gives us guarantees that `Node` is both inhabited and contains a finite number of elements within the type. To illustrate, our proof of `notInjective` would utilize the pigeon hole principle and appeal to the fact that `Node` is finitely inhabited.

We discuss further in section 5.5 how these instances enable us to perform integer and cardinality operations on sigs and translated Forge expressions.

§5.2 Inheritance

Many signatures in Forge have complex inheritance structures [9] 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 4.4):

```
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 `Undergrads` are also `Students`, and since fields are typed to the sig that they are a part of in Lean[8], 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:

```
1  sig Class {}
2  sig Student {
3    registration : set Class
4  }
5  sig Undergrad extends Student {}
6
7  fun ugradsIn[c : Class] : Undergrad {
8    all u : Undergrad |
9      c in u.registration
10 }
```

```
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!
```

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 interchangably 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 `Undergrads` are `Students`. 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.

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

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 `Undergrads` to `Grads` 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`[9], this becomes burdensome and involved:

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

The existential which quantifies over all `Undergrads` 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.

In addition, Forge introduces the concept of *abstract* sigs [9]. If `Student` were an abstract sig, we might encounter `Undergrad` and `Grad` student as concrete subtypes of `abstract Student`. We might encounter:

```
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
```

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

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

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 that makes such a solution of model specification 'mixins' 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.

§5.3 "Everything is a Set"

Overloaded operations

The predominantly relational nature of Forge introduces another 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). 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 are able to 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 possibly 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` ⋈ `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.

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[10] 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 the 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[11] of our implementation of relational join as a type class `HJoin` ('has join'), following our implementations of join from the examples demonstrated above:

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

[11] 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.

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

-- Join singleton with arity-2 relation
@[reducible] 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] 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 nondiscriminately 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.

For many operators on expressions (see section 5.4 below), their implementations in Lean are overloaded to accomodate 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 is able to determine types of operands at the time of translation.

Types as sets

Forge's typing ambiguity further exemplifies itself in allowing types to be used where a set is expected to denote the set of all elements in said type. For example:

```
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 5.1), 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. check this after writing that section

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

§5.4 Relational Joins, Cross, Inclusion, Equality

We need to take special care when implementing expression operators in Lean whenever one of these conditions are 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 simpliest.

We discuss (2) extensively in section 5.3, and introduce using Lean's type classing 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 1.

Custom syntax for the binary operations to pretty print

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 5.3.*
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`

Table 1: Forge binary operators and corresponding implementations based on operand types.

§5.5 Integers

Forge and Alloy come with a unique integer model—due to the fact that models are compiled down to boolean constraints to be solved by a SAT solver, integers are severely limited in their bitwidth [9, 16]. The default bitwidth on integers in Forge is 4, which gives us a grand total of 16 integers.

Some programs will inadvertently (or cleverly) utilize integer overflow which affects their model in a meaningful way [17]. 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 piggybacks on an honest-to-goodness inductive model for the natural numbers [2]. 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 are able to use Lean's integer and finite set/types libraries out of the box with little modification.

Our translated Lean models treats 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 [9] 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[12], 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 }
```

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`[13]. While this function has a junk value

[12] 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 [15, 16].

[13] More specifically, we do need to utilize the approach in section 5.3 of using type classes to implement this, since

when a set is infinite, we had remedied this earlier in section 5.1 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 ⋈ 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 14
```

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 are able to endow *additional* functionality to the Forge specification language by interpreting it in a proof assistant instead of the standard relational Forge implementation.

the cardinality of a singleton ought to be 1. In all other cases, `Set.ncard` is the implementation of cardinality.

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

§6 Results and Examples

§6.1 Forge as a Lean DSL

§6.2 A Motivating Example

Better section name

§7 Discussion

Bibliography

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

[2] Jeremy Avigad, Leonardo De Moura, Soonho Kong, and Sebastian Ullrich. Theorem proving in lean. 2024.

[3] 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*, pages 12–16. Springer, 2008.

[4] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and precise type checking for javascript. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30, 2017.

[5] Albert Mo Kim Cheng. A survey of formal verification methods and tools for embedded and real-time systems. *International Journal of Embedded Systems*, 2(3-4):184–195, 2006.

[6] 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*, pages 337–340. Springer, 2008.

[7] George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, 1997.

[8] 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)*, pages 758–769. IEEE, 2017.

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

[10] Daniel Jackson. Alloy: a language and tool for exploring software designs. *Communications of the ACM*, 62(9):66–76, 2019.

[11] 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*, pages 213–228. Springer, 2016.

[12] Thierry Lecomte, David Déharbe, Étienne Prun, and Erwan Mottin. 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*, pages 70–87. Springer, 2017.

[13] David R MacIver, Zac Hatfield-Dodds, et al. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019.

[14] 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*, pages 625–635. Springer, 2021.

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

[16] Tim Nelson, Ben Greenman, Siddhartha Prasad, Tristan Dyer, Ethan Bove, Qianfan Chen, Charles Cutting, Thomas Del Vecchio, Sidney LeVine, Julianne Rudner, Ben Ryjikov, Alexander Varga, Andrew Wagner, Luke West, and Shriram Krishnamurthi. Artifact for Forge: A Tool and Language for Teaching Formal Methods, January 2024.

[17] Tim Nelson, Ben Greenman, Siddhartha Prasad, Tristan Dyer, Ethan Bove, Qianfan Chen, Charles Cutting, Thomas Del Vecchio, Sidney LeVine, Julianne Rudner, Ben Ryjikov, Alexander Varga, Andrew Wagner, Luke West, and Shriram Krishnamurthi. Forge: A tool and language for teaching formal methods. *PACMPL*, 8(OOPSLA1):1–31, 2024.

[18] Arthur Paulino, Damiano Testa, Edwards Ayers, Evgenia Karunus, Henrik Böving, Jannis Limperg, Siddhartha Gadgil, and Siddharth Bhat. *Metaprogramming in Lean 4*. 2024.

[19] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4):1–36, 2009.