Translating Lean 4 for Higher-Order SMT Solvers

Max Nowak

IPD Snelting

Abstract. Hammers boost interactive theorem prover (ITP) productivity tremendously, as seen with Isabelle's Sledgehammer. Only first order hammers exist for ITPs based on Martin-Löf type theory, such as SMT-Coq for Coq and lean-smt for Lean, however, recent advances in SMT solvers allow for efficient solving of higher order problems. As a first step towards a higher order hammer for Lean, we present a translation from Lean to simply typed higher order logic. We erase type indices by splitting inductive data types with type indices into a pure data component and a pure propositional component. We monomorphize polymorphism. Translation steps are implemented as Lean tactics, making use of Lean's extensive metaprogramming capabilities, and enabling a compositional approach.

Keywords: Hammers · Martin-Löf Type Theory · Lean 4 · Higher-Order Logic · SMT Solvers

1 Introduction

Interactive theorem provers (ITPs) live at the intersection between programming languages and mathematics. ITPs can be used much like a programming language, allowing users to define functions and introduce new data types, but additionally also allow users to state theorems and write proofs of those theorems, all in the same file. Crucially, these proofs are checked by the ITP's trusted but small *kernel* for correctness, giving a stronger correctness guarantee than proofs on paper.

ITPs usually feature a pure functional language with polymorphism, but are based on different underlying theories. *Isabelle* [1, 2] is an ITP based on *higher order logic* (HOL). *Coq* [3] and *Lean 4* [4] are ITPs based on *Martin-Löf type theory* (MLTT) [5, 6] and the *calculus of inductive constructions* [7], which together constitute an alternative foundation of mathematics to Zermelo-Fraenkel set theory. While MLTT may not be more powerful than ZFC with HOL, it is more expressive. For example, MLTT's dependent function type[1] $(\texttt{x} : \alpha) \to \beta$, where β may refer to `x`, can not be directly expressed in HOL.

Under MLTT's *propositions-as-types* interpretation [6], propositions are encoded as types and values of those types are proofs thereof. Hence, the same language can be used to express code, definitions, data structures, theorems, and proofs of those theorems. For example it is possible to have a data structure

[1] In literature commonly referred to as the Π-type, written $\Pi_{x:\alpha}\ \beta$.

where one of the fields is an invariant restricting the values of the other fields; it is then enforced by the compiler (i.e. the ITP) that values of that data structure may only be constructed if a proof of the invariant is provided. This brings with it the flexibility of mixing code and proof at will, but comes with the drawback of complexity.

ITP proofs require explicit mentions of which rule was applied with which parameters, and are not allowed to skip over steps which seem "obvious" to humans. As a result ITP proofs can be hard for a human to comprehend and even more difficult for a human to write.

1.1 SMT Solvers and Hammers

Satisfiability modulo theories (SMT) solvers are SAT solvers with the ability to also reason about specific theories and quantifiers, such as the theory of arrays, bitvectors, or real numbers. SMT solvers use heuristics to find a proof[2] for a given problem consisting of premises and a proof goal. Most SMT solvers specialize in solving untyped FOL problems. Recent advances in SMT solvers have brought the success rate for simply typed higher order problems on par with that of first order problems. SMT solvers are powerful, but are not able to reason about the high level languages of ITPs. Few SMT solvers are capable of solving polymorphic problems, and success rates are low. No SMT solvers exist which can directly solve MLTT problems. A more detailed overview of SMT solvers and their proof search success rates can be found in [8].

Hammers [9] automate proof search by delegating the problem to an SMT solver. The first step is to find around 1000 auxiliary lemmas which hopefully enable the SMT solver to derive a proof for our proof goal. Then the goal, the auxiliary premises, and any other necessary items such as data types need to be translated to a logic that the SMT solver understands. Finally, if the SMT solver did find a proof for the translated goal, a proof for the original goal needs to be reconstructed. An overview over the four steps can be seen in figure 1.1.

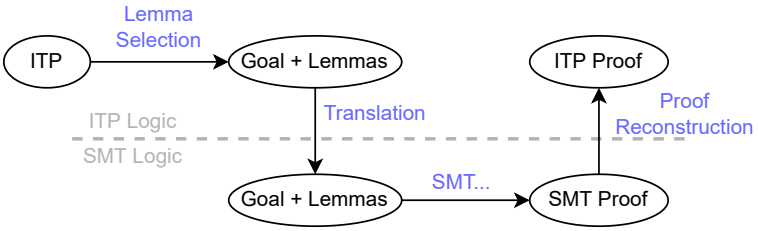

Fig. 1. The four steps of a hammer.

[2] SMT solvers try to find models. If an SMT solver is given the negated conjecture and is able to prove the absence of a model, that constitutes a proof of the conjecture.

1.2 Related Work

Isabelle/HOL's Sledgehammer [10, 11] uses monomorphization [12] in order to translate polymorphism. Sledgehammer often throws away the proof found by the SMT solver and just looks at which lemmas were necessary in order to prove the goal, essentially (mis)using the SMT solver as a relevance filter. The lemmas are then given to a resolution or superposition proof method such as metis.

Two hammers exist for Coq, namely *CoqHammer* [13] and *SMTCoq* [14]. CoqHammer translates to first order logic without arithmetic, and usually struggles with higher-order constructs such as `map` or `foldr`. SMTCoq attempts to use arithmetic as well, but makes no effort to use the higher order capabilities of SMT solvers.

Lean's tooling ecosystem is rapidly evolving, with work underway for hammers and other proof search tools. In contrast to hammers, *Aesop* [15] takes a white-box approach, and does not use SMT solvers. Aesop works by applying existing tactics repeatedly. Aesop is extensible by users, enabling them to customize which rules are applied. Aesop is implemented entirely in Lean, and does not invoke any SMT solvers, therefore does not need to translate MLTT to a different logic.

Another proof automation tool for Lean 4 is *duper*, which is a superposition prover implemented as a Lean 4 tactic. Superposition provers are based on the superposition calculus which is a further development of resolution, and has been extended to higher-order inputs recently as well [16]. Such provers are more powerful capable of more than proofs by simplification, but currently less powerful than SMT solvers. As such, resolution provers are often used for proof reconstruction by hammers.

In parallel to lean-hosmt, another hammer for Lean 4 is being developed called lean-smt. The most prominent difference between lean-smt and lean-hosmt is that lean-smt targets FOL, while lean-hosmt targets HOL SMT solvers. As such, lean-smt can not directly reason about higher-order constructs such as `map` and `fold` without unrolling.

2 The Translation Architecture at a Glance

A *proof goal* is a metavariable, i.e. a placeholder `?g` with an expected type φ in a given context Γ, usually with φ `: Prop`. In MLTT finding a proof for a proposition is the same problem as finding a value for a given type. Goals can be discharged by assigning its metavariable a suitably typed value, for example the goal `?g` in $\Gamma \vdash$ `?g : And` $\alpha\ \beta$ can be discharged by assigning `?g := And.intro a ?b` given that $\Gamma \vdash$ `a :` α. The value assigned to a metavariable can itself contain more metavariables, resulting in new proof goals, such as `?b` in the preceding example. This is the basic principle on which *tactics* operate, they are metaprograms which construct a proof of the desired type, thus either solving the goal outright or generating sub-goals.

Lean uses the `Expr` data type in order to encode expressions, which includes values, types, propositions, and proofs. An `Expr` can for example be a constant, a

function application, a (dependent) function type, or a lambda expression. Our translation works by transforming expressions. Metaprograms in Lean are able to infer types of expressions and add new definitions and inductive types on the fly. Additionally, we can use Lean's internal helper functions, such as the existing functions to generate injectivity theorems and equations for recursive functions using pattern matching. Tactics can invoke other tactics.

Our translation is implemented as a tactic, which is a composition of multiple translation steps, which themselves are tactics in principle. This way we decompose our translation into separate stages. By applying sufficient translation steps, we arrive at a Lean proof goal which is directly encodable to TPTP THF0. We explain THF0 and the final encoding step in section 6.

Definition 1. *A type τ is a* simple type *when τ : `Type _` and $\tau \not\equiv$ `Type _`. A type τ is* THF0-compatible *when it is a simple type or when $\tau \equiv \alpha \to \beta$ given two THF0-compatible types α and β. A term `x` is* THF0-compatible *when `x` : τ for a THF0-compatible type, and all its subexpressions have a THF0-compatible type. An example of a THF0-compatible type is `(Nat → Prop) → Nat`.*

Definition 2. *Let α and β be THF0-compatible propositions. A proposition φ with φ : `Prop` is* THF0-compatible *when $\varphi \equiv$ `And` α β or $\varphi \equiv$ `Or` α β or $\varphi \equiv$ `Not` α or $\varphi \equiv \alpha \to \beta$ or $\varphi \equiv$ `(x : τ) → α` for a THF0-compatible type τ, or $\varphi \equiv$ `λx : τ, α`, or φ refers to a THF0-compatible proposition definition, or φ refers to a THF0-compatible inductive predicate.*

Many Lean expressions have a direct equivalent in THF0, hence do not need to be translated. Propositions φ with φ : `Prop` are THF0-compatible already with exception of non-simple inductive types, which in Lean can be polymorphic or contain type indices. As a result, most of the translation is concerned with to translating inductive data types, as that is where most of the additional complexity of MLTT resides. We explain how to translate inductive types in section 3.

We use $\lfloor \cdot \rfloor$ to denote *traversal functions* which map expressions to expressions, and may have side effects such as adding a new declaration to the environment. A traversal function checks whether the expression matches a pattern, replacing it with a translated expression if it does, and then recursively descends into subexpressions. For example, descending through dependent function types works as $\lfloor$`(x : α) → β`$\rfloor$:≡ `(x :` $\lfloor\alpha\rfloor$`) →` $\lfloor\beta\rfloor$. Traversal descends into environment items such as definitions and inductive types, and since traversal functions are metaprograms, they have access to information provided by Lean such as number of type parameters and number of type indices (see section 3). A *translation step* is a traversal function applied to a proof goal.

During the translation of binders such as quantifiers and lambdas the body can become intermittently ill-typed. For example $\Gamma \vdash \lfloor$`λx : List Nat, sum Nat x`$\rfloor$ will be translated as $\Gamma \vdash$ `λx : List', ⌊sum Nat x⌋` due to monomorphization (see section 5), which contains the sub-expression `sum Nat x`, which is ill-typed due to $\Gamma \vdash$ `sum Nat : List Nat → Nat`, which is incompatible with a monomorphized argument `x` with $\Gamma \vdash$ `x : List'`. In practice this is not an issue, since the

translation continues and eventually replaces `sum Nat x` with the monomorphized version `sum' x`, which is well-typed. This flaw could be remedied by wrapping `x` in a helper function $\lceil\cdot\rceil$ `: List' → List Nat` inside the body, which then annihilates once our translation reaches it. For example λ`x : List', ⌊sum Nat ⌈x⌉⌋` $\equiv$ λ`x : List', sum' ⌊⌈x⌉⌋` $\equiv$ λ`x : List', sum' x`, thus preserving well-typedness at every step.

3 Translating Inductive Types

Definition 3. *A* telescope a `::` A *is short hand notation for zero or more binders* a_1 `:` A_1`, ...,` a_n `:` A_n*, where* a_1 `:` A_1`, ...,` a_{i-1} `:` $A_{i-1} \vdash A_i$*. The meaning in different expressions is as follows:*

$$(a :: A) \to \beta :\equiv (a_1 : A_1) \to \ldots \to (a_n : A_n) \to \beta,$$
$$\lambda a :: A,\ body :\equiv \lambda a_1 : A_1, \ldots, \lambda a_n : A_n,\ body,$$
$$a :: A \vdash \beta \text{ is equivalent to } a_1 : A_1, \ldots, a_n : A_n \vdash \beta,$$
$$f\ a :\equiv f\ a_1\ \ldots\ a_n.$$

Inductive types in Lean have a general form as can be seen in listing 1.1.

```
inductive T (u :: U) : I → Sort v where
| ctor₁ (u :: U) (a₁ :: A₁) : T u i₁
⋮
| ctorₖ (u :: U) (aₖ :: Aₖ) : T u iₖ
```

Listing 1.1. General form of inductive types in Lean. Here `I` stands for multiple binders.

The parameters `u` are *uniform* parameters (∗U), since they occur uniformly in all constructors. The parameters typed `I` on the other hand may differ between constructors, and are referred to as *type indices* (∗X). Type indices make it possible to discern which constructor must have been used to construct a value based on that value's type. Both uniform parameters and type indices can be further subdivided by whether they take a value (V∗) or a type (T∗).

Example 1. The type of lists of known length is given by `Vect` in listing 1.2. Here, α is a TU parameter, and the second parameter is a VX type index. We can infer that a value typed `Vect Nat 4` must have been constructed using the `Vect.cons` constructor.

```
inductive Vect (α : Type) : Nat → Type where
| nil  (α:Type) : Vect α 0
| cons (α:Type) (k:Nat) : (x : α) → (tail : Vect α k) → Vect α (k+1)
```

Listing 1.2. The `Vect` type with known length.

We use monomorphization in order to translate TU parameters, which is described in section 5. We use a guard construction in order to erase VX parameters, which is described in section 4. We do not translate TX parameters, making our translation incomplete, and argue that TX parameters are sufficiently rare. Monomorphization is applicable to any uniformly occurring parameter, and thus also applicable to VU parameters. When determining the parameters to monomorphize, it is sufficient to simply choose VU parameters as well. Alternatively, we can use the guard construction to translate VU parameters, as uniform parameters are merely a special case of type indices.

Fig. 2. VU parameters can be translated either along with monomorphization, or along with the guard construction which is used to translate type indices.

Unlike Coq, Lean's kernel has no native concept of a `match` or `fixpoint`, and instead every inductive type `T` in comes with a recursor axiom `T.rec` which enables pattern matching and provides an induction schema. Expressions using `match` are compiled to invocations of recursors. In fact, all proofs by induction in Lean boil down to inductive types and their recursors.

3.1 Structures and Type Classes

Structures in Lean are inductive types with exactly one constructor, usually named `mk`, are not recursive, i.e. types of fields can not refer to the structure type, and have no type indices. Structures can have fields with types which in turn have type indices. Structure field types can depend on values of their preceding structure fields, as such, structures are essentially named Σ-types.

Type classes in Lean are structures, for which an instance can be synthesized. After elaboration, type class instances are already synthesized, hence type class synthesis is not a concern during translation. Our translation does not handle type classes in any different way than structures.

Accessing fields in structures formally requires using the recursor `T.rec`, however Lean has an optimization for structures. Projection functions for structure fields have a special representation in Lean's `Expr` data structure. Given `x : T` α, the expression $\texttt{T.field}_i$ α `x` is represented internally as `Expr.proj "T" (i - 1) x`, with `"T"` being the name of the inductive type, and `i - 1` the index of the field. Notably, type arguments such as α are not present, but can be obtained by inferring the type of `x` thanks to Lean's metaprogramming capabilities. Our translation maintains this projection optimization.

4 Index Erasure

A type index constrains the amount of values a type can admit. This constraint can be modeled as a predicate, and a scheme exists to derive an inductive predicate from a type's constructors [17]. Type indices have no direct correspondence in HOL, but can be modeled by a guard construction. Given a type τ and a predicate `P` with `P : τ → Prop`, the subtype `{ x : τ // P x }` of τ can only assume values which satisfy `P`.

Given an inductive type `T` as in example 1.1, we first obtain `E` with erased type indices, as can be seen in listing 1.3. Let $[\![\texttt{A}]\!]_\epsilon$ be the type `A` with every occurence of `T u i` replaced with its type index erased `E u` version.

```
inductive E (u :: U) : Type _ where
| ctor_k (u :: U) (a_k :: [[A_k]]_ε) : E u
```

Listing 1.3. Erasing type indices of `T` yields `E`.

Since `E` is a supertype of `T` due to being less restrictive, we restrict its domain back with a guard construction. Let $[\![\texttt{A}]\!]_\xi$ be the type `A` with every occurence of `T u i` replaced with `{ target : E u // G u i target }`. Let $[\![\texttt{a}]\!]_\delta$ be the expression `a` with every subterm `x` typed `T u i` replaced with `x.val`, i.e. extracting the data component from the subtype expression, in order to maintain `x.val : E u`. For example, while erasing the type index of `V : Nat → Type`, we have $[\![\texttt{Nat}]\!]_\xi \equiv \texttt{Nat}$, and we have $[\![\texttt{V 3}]\!]_\xi \equiv$ `{v : VE // VG 3 v }`. Given $\texttt{a}_1$ `: Nat`, we have $[\![\texttt{a}_1]\!]_\delta \equiv \texttt{a}_1$ and given $\texttt{a}_2$ `: {t : VE // VG 3 t}` we have $[\![\texttt{a}_2]\!]_\delta \equiv \texttt{a}_2\texttt{.val}$ and thus $\texttt{a}_2\texttt{.val}$ `: VE`. We obtain the inductive predicate `G` following the schema seen in listing 1.4.

```
inductive G (u :: U) : I → (target : E u) → Prop where
| ctor_k (u :: U) (a_k :: [[A_k]]_ξ) : G u i_k (E.ctor_k u [[a_k]]_δ)
```

Listing 1.4. Inductive predicate enforcing type indices.

In summary, we split a type `T` into its index-erased data component `E` and its pure propositional guard component `G`. Even though `G` has more type indices than `T`, we have progressed closer towards a THF0-compatible type `{ t : E u // G u i t }`, since inductive types of form `inductive G : ... → Prop` are encodeable as inductive predicates into TPTP, with type indices acting as normal function parameters. Usages of `T` are replaced by the guard construction, with the TPTP encoding simply ignoring propositions when encoding functions, and adding pre and postconditions.

Example 2. From example 1.2 we obtain `VectE : Type → Type` as seen in listing 1.5. The `k` parameter on `VectE.cons` still remains despite no longer serving any purpose. We can not erase `k`, since it is indistinguishable from data parameters such as `elem`. We then construct the inductive predicate `VectG` as can be seen in listing 1.6. The proposition `VectG α k v` corresponds to $\mathsf{VectG}_\alpha(k, v) \iff \mathsf{length}(v) = k$.

```
inductive VectE (α : Type) : Type where
| nil  (α:Type) : VectE α
```

```
| cons (α:Type) (k:Nat) : (elem : α) → (tail : VectE α) → VectE α
```

Listing 1.5. Vector with erased type indices.

```
inductive VectG (α : Type) : Nat → VectE α → Prop where
| nil  (α:Type) :
       VectG α 0 (VectE.nil α)
| cons (α:Type) (k:Nat) : (elem : α)
    → (tail : { v : VectE α // VectG α k v })
    → VectG α (k + 1) (VectE.cons α (k + 1) elem tail)
```

Listing 1.6. Inductive predicate enforcing invariants of erased type indices.

Lean has a limitation which does not allow for subtype expressions to be used while translating type indices. As such, any binder such as `(x : { t : τ // P t }) → Rest` needs to be written as two binders `(x : τ) → P t → Rest`.

We can use index erasure to erase VU parameters on a structure `S : (u :: U) → (i :: I) → Type _`, where `i` are the VU parameters, with constructor `S.mk : (u :: U) → (i :: I) → (a :: A) → S u i`, where `a` correspond to the fields. The erased `E : (u :: U) → Type _` type then has the constructor `E.mk : (u :: U) → (i :: I) → (a :: A) → E u`. Since the projection field index 0 refers to the first field after the last parameter, `E` effectively has more fields. We thus need to update `Expr.proj "S" i x` and shift `i` by the amount of erased VU parameters.

5 Monomorphization

Few SMT solvers support polymorphism, and when polymorphism is supported, the percentage of solved problems by the SMT solver decreases [8]. Polymorphic types are types which depend on other types. For example, given two variables α and β with α `: Type` and β `: Type`, the types $\alpha \to \beta$ and `List` α are polymorphic. Lean only supports parametric polymorphism, as opposed to ad-hoc polymorphism. We assume that all binder types are of form `T args` or `Sort _`, where `T` is some inductive type. It is possible that constructors of an inductive type are themselves polymorphic. We consider polymorphic constructors to be sufficiently rare and choose to ignore them.

Let $\tau :\equiv$ `(a`$_1$ `: A`$_1$`)` $\to \ldots$ `(a`$_n$ `: A`$_n$`)` $\to \beta$. We want to determine the set $M \subseteq \{1, \ldots, n\}$ of indices of binders `(a`$_i$ `: A`$_i$`)` which need to be monomorphized. Any binder `(a`$_i$ `: A`$_i$`)` with `A`$_i \equiv$ `Type _` is not THF0-compatible and needs to be monomorphized, therefore $i \in M$. By simply choosing a different pattern, we can also handle VU parameters to inductive types. Since monomorphization happens after type index erasure, choosing $M = \{1, \ldots, n\}$ will also monomorphize VU parameters.

We can assume that binders which need to be monomorphized are sorted to the left. Binders which do not depend on each other can be reordered, i.e. `(a`$_1$ `: A`$_1$`)` $\to$ `(a`$_2$ `: A`$_2$`)` $\to \gamma$ and `(a`$_2$ `: A`$_2$`)` $\to$ `(a`$_1$ `: A`$_1$`)` $\to \gamma$ are both well-typed expressions given that `B` does not depend on `a`, and that `A` does not depend on `b`.

In our implementation we use a *monomorphization pattern* instead of reordering binders, allowing us to monomorphize `f a b c` into `f' a c`.

Definition 4. *The* monoTy *and* monoVal *expression transformers are defined as follows. Herein, without loss of generality, we assume that binders which need to be monomorphized are sorted to the left and are exactly* `(a :: A)`.

$$\lfloor \texttt{(a :: A)} \to \texttt{(b :: B)} \to \beta \rfloor_{monoTy(x)} :\equiv \texttt{((b :: B)} \to \beta\texttt{)}[\texttt{a} \mapsto \texttt{x}], \tag{1}$$

$$\lfloor \lambda\texttt{a :: A, } \lambda\texttt{b :: B, body} \rfloor_{monoVal(x)} :\equiv \texttt{(}\lambda\texttt{b :: B, body)}[\texttt{a} \mapsto \texttt{x}]. \tag{2}$$

Additionally, we define a transformer monoInd *for inductive types as* $\lfloor \texttt{T} \rfloor_{monoInd(x)} :\equiv \texttt{T'}$*, with* `T` *and* `T'` *as follows:*

```
inductive T : (a :: A) → (b :: B) → Sort _ where
| ctor_j : (a :: A) → (b :: B) → (c :: C) → T

inductive T' : (b :: B) → Sort _ where
| ctor_j : ((b :: B) → (c :: C) → T')[a ↦ x]
```

Definition 5. *We define an expression traversal function* $\lfloor \cdot \rfloor_m$ *which replaces patterns listed in equations 3, 4, and 5, given that* `T` *and* `f` *are polymorphic, i.e.* `f : (α : Type)` $\to \ldots$*. If no pattern matches, it simply descends into each subexpression with* $\lfloor \texttt{f a} \rfloor_m :\equiv \lfloor \texttt{f} \rfloor_m \ \lfloor \texttt{a} \rfloor_m$ *for function application,* $\lfloor \texttt{(x : } \tau\texttt{)} \to \beta \rfloor_m :\equiv \texttt{(x : } \lfloor \tau \rfloor_m\texttt{)} \to \lfloor \beta \rfloor_m$ *for (dependent) function types, analogously for lambda expressions, etc.*

$$\lfloor \texttt{f x y} \rfloor_m :\equiv \lfloor \lfloor \texttt{f} \rfloor_{monoVal(x)} \ \texttt{y} \rfloor_m \tag{3}$$

$$\lfloor \texttt{T x y} \rfloor_m :\equiv \lfloor \lfloor \texttt{T} \rfloor_{monoInd(x)} \ \texttt{y} \rfloor_m \tag{4}$$

$$\lfloor \texttt{T.ctor}_j \ \texttt{x y} \rfloor_m :\equiv \lfloor (\lfloor \texttt{T} \rfloor_{monoInd(x)}\texttt{).ctor}_j \ \texttt{y} \rfloor_m \tag{5}$$

Since definitions in Lean are nothing more than named expressions, we also obtain a schema for monomorphizing definitions. Theorems in Lean are nothing more than definitions for which no executable code is generated, which for the purposes of creating a hammer is irrelevant. Axioms are definitions without a body, and thus only their type needs to be translated. As a result we can transform Lean environment items as seen in listing 1.7.

```
⌊def fn : τ := body⌋ₘ :≡ def fn' : ⌊⌊τ⌋monoTy(x)⌋ₘ := ⌊⌊body⌋monoVal(x)⌋ₘ
⌊theorem t : τ := prf⌋ₘ :≡ theorem t' : ⌊⌊τ⌋monoTy(x)⌋ₘ := ⌊⌊prf⌋monoVal(x)⌋ₘ
⌊axiom ax : τ⌋ₘ :≡ axiom ax' : ⌊⌊τ⌋monoTy(x)⌋ₘ
```

Listing 1.7. Monomorphization of Lean definitions, theorems, and axioms, given zero or more arguments `x` to monomorphize for, with `τ : A → B → R` and `x :: A`.

Often we encounter polymorphic goals which can not be monomorphized. Let `τ :≡ (α : Type) → [Ring α] → P α`. For example a statement `τ` about all rings `α`. In order to still be able to prove these goals, we use the *α-trick*. Given a proof goal `τ`, we "introduce" the first binder, but instead of adding the declaration

α `: Type u` to the context, we add a new environment item `axiom` α `: Type u`, making sure that α is a new unique name. This axiom corresponds to an inductive type with unknown constructors and no recursor, and, crucially, can be encoded to our target langauge TPTP. We then change every occurence of the variable α in our goal τ to the environment item α. Now α is a type we can monomorphize to. The α-trick can only be applied to the goal.

Example 3. The following example showcases how monomorphization works. We obtain expression (7) via the α-trick. We obtain expression (8) by applying $\lfloor\cdot\rfloor_{\mathsf{m}}$ multiple times. We then add monomorphized versions of `List` and `sum` to the environment, and obtain the final result (9), which is now THF0-compatible.

$$\lfloor(\alpha : \texttt{Type}) \to (\texttt{xs} : \texttt{List}\ \alpha) \to \texttt{sum}\ \alpha\ \texttt{xs} = 1\rfloor_{\mathsf{m}} \tag{6}$$

$$\equiv \lfloor(\texttt{xs} : \texttt{List}\ \alpha) \to \texttt{sum}\ \alpha\ \texttt{xs} = 1\rfloor_{\mathsf{m}} \tag{7}$$

$$\equiv (\texttt{xs} : \lfloor\texttt{List}\rfloor_{\mathsf{monoInd}(\alpha)}) \to \lfloor\texttt{sum}\rfloor_{\mathsf{monoVal}(\alpha)}\ \lfloor\texttt{xs}\rfloor_{\mathsf{m}} = 1 \tag{8}$$

$$\equiv (\texttt{xs} : \texttt{List'}) \to \texttt{sum'}\ \texttt{xs} = 1 \tag{9}$$

From the point of view of the SMT solvers, the monomorphized types `ListNat` $\equiv \lfloor\texttt{List}\rfloor_{\mathsf{monoInd}(\texttt{Nat})}$ and `ListInt` $\equiv \lfloor\texttt{List}\rfloor_{\mathsf{monoInd}(\texttt{Int})}$ are unrelated. As such, any theorems about `ListNat` will not be applicable to goals about `ListInt`, and vice-versa.

6 Encoding to TPTP

We encode to TPTP's THF0 form [18], i.e. typed higher order form without polymorphism. Many Lean constructs have direct equivalents in TPTP. Lean function application `f a` corresponds to `f @ a` in TPTP. Logical connectives and equality have direct equivalents as well, such as `And` α β to α `&` β, and `Eq` τ α β to α `=` β. Non-dependent function types $\alpha \to \beta$ correspond to α `>` β, provided that α and β are THF0-compatible types. We encode Lean's type of propositions `Prop` as the boolean type `$o`. The type `Nat` $\to$ `Prop` is translated as `nat > $o`. Lambdas λ`x : `α`, body` correspond to `^[X : `α`]: body`. Propositional dependent function types, i.e. forall quantifiers $\forall$`x : `α`,` β, correspond to `![X :` α`]:` β, provided that α is a THF0-compatible type and β is a THF0-compatible proposition. The type universe token `$tType` can only be used to introduce new types. The boolean type is `$o`. A *problem* in TPTP is a list of axioms, typing declarations, and one or more conjectures. An example can be seen in listing 1.8. For the sake of clarity, we often use Lean syntax instead of TPTP syntax.

```
thf(a1, type, nat : $tType).
thf(a2, type, zero : nat).
thf(a2, type, add : nat > (nat > nat)).
thf(a3, axiom, ![X : nat]: (![Y : nat]: (
  ((add @ X) @ Y) = ((add @ Y) @ X)
))).
thf(a4, conjecture, ![X : nat]: (((add @ X) @ zero) = X)).
```

Listing 1.8. Example TPTP THF0 problem.

We use Lean's type inference when encoding expressions in order to decide whether an expression should be encoded as a proposition or a term.

6.1 Inductive Data Types

We only consider inductive data types `T` as seen in listing 1.1 with zero parameters. The judgement `T : Type _` is encoded as a typing declaration `thf(ty, type, t : $tType)`. Each constructor $\texttt{T.ctor}_j$ with $\texttt{T.ctor}_j$ `: (a :: A)` $\rightarrow$ `T` is encoded as a typing delcaration `thf(ty, type,` $\texttt{ctor}_j$ `:` $\texttt{a}_1$ `>` $\texttt{a}_2$ `> ... >` $\texttt{a}_n$ `> T)`. Lean generates injectivity theorems for each non-trivial constructor, which we add as axioms to the resulting problem.

Every inductive type in Lean has an associated eliminator, the recursor `T.rec`, which provides both an induction schema and a recursion schema. While the equations obtained for functions do provide some indirect way to recurse through inductive data types, we currently do not add a general induction schema axiom for inductive types. Even without a recursor, problems can be solved by the SMT solver given enough auxiliary lemmas.

6.2 Inductive Predicates

While inductive data types are encoded as new types, inductive predicates are encoded as new boolean-valued functions. Since MLTT pulls formulas into the type system, we push formulas back down. As such, the proposition `Even n` as seen in listing 1.9, i.e. an inductive type, is encoded as an inductively-defined predicate as seen in listing 1.10.

```
inductive Even : Nat → Prop where
| base : Even 0
| step : (n : Nat) → Even n → Even (n + 2)
```

Listing 1.9. Original inductive predicate `Even` in Lean.

```
thf(a1, type, even : nat > $o).
thf(a2, axiom, even 0).
thf(a3, axiom, (∀N : nat, even N → even (N + 2))).
```

Listing 1.10. Encoded inductive predicate `Even` in THF0.

6.3 Definitions

Definitions in Lean are merely named expressions. Recursion and pattern matching are handled by recursors, which may occur deeply nested in expressions. Functions can be encoded in multiple ways to TPTP.

Functions using no pattern matching (and thus no recursion), can be encoded directly, as seen in listing 1.11. This is enabled thanks to HOL natively being able to handle lambda expressions.

```
thf(type, f : a → b).
thf(axiom, f = (λx : a, body)).
```

Listing 1.11. Encoding of functions without pattern matching.

Functions which do use pattern matching can be encoded by one equation axiom per pattern. For example, addition of natural numbers is encoded as seen in listing 1.12. If a function matches on values nested deeper inside the function, Lean lifts the subexpression containing the match into a new function.

```
thf(type, add : nat → nat → nat).
thf(axiom, (∀a : nat, add 0 n = 0)).
thf(axiom, (∀a : nat, ∀b : nat, add a (b + 1) = (add a b) + 1)).
```

Listing 1.12. Encoding of functions with pattern matching.

6.4 Structures and Projections

A solid encoding of structures and projection functions is important, as both occur frequently in Lean's Mathlib due to type classes. We replace Lean's optimization of projections `Expr.proj "T" i x` with its associated projection function $\texttt{T.field}_i$ before encoding. When encoding the projection function $\texttt{T.field}_i$ itself, we add a custom axiom as seen in listing 1.13 instead of using the recursor. Since encoding to TPTP happens after index erasure and monomorphization, we can assume that `T` is a simple type.

```
thf(type, T.field_i : T → F_i).
thf(axiom, (∀x : T, (x = T.mk f_1 f_2 ... f_n) → (T.field_i x = f_i))).
```

Listing 1.13. Encoding projection functions. $\texttt{f}_i$ are the fields, typed $\texttt{F}_i$ respectively.

7 Conclusion

We have implemented a translation from the Martin-Löf type theory based Lean to TPTP THF0, i.e. simply typed higher order logic. The main challenge, translating inductive types, is solved with two techniques, namely monomorphization and type index erasure. Targeting higher order logic enables a direct mapping of lambda expressions and partial function application, obviating the need for SKBCI combinators, unrolling, or modeling either explicitly with first order logic.

Our approach focuses on each translation step mapping Lean proof goals to new Lean proof goals, thus utilizing Lean's powerful metaprogramming capabilities. We translate from one valid proof goal and environment containing inductive types, to another valid proof goal and environment. Through this, we enable a compositional approach, with each translation step communicating solely[3] through Lean's tactic framework. Additionally, since adding new inductive types

[3] Currently, translation steps have some shared state for caching, and can be moved to Lean environment extensions.

and definitions requires interacting with Lean's kernel, common mistakes by a translation step are caught early. Another advantage is proof reconstruction.

A disadvantage of translating Lean to Lean is the increased complexity. Lean implements many optimizations, such as direct representation of projections for structure-like inductive types in `Expr`, which necessitate handling these optimizations when traversing Lean expressions. We must construct valid Lean expressions, including features which are entirely removed in subsequent steps, such as type universes.

Nonetheless, our translation is incomplete. Although during translation steps we maintain type universes, they are erased during the encoding to TPTP, which makes it possible for an SMT solver to derive `False` for an otherwise true theorem. We do not translate recursors, which leads to theorems about `map` or `fold` to translate, but be unprovable without additional theorems from lemma selection. We do not translate TX parameters to inductive types.

7.1 Observations

We are able to translate all 122 theorems found in Lean's Mathlib4 modules `Mathlib.Algebra.Abs`, `Mathlib.Algebra.NeZero`, and `Mathlib.Algebra.Group.Defs`. Of these 122 theorems, 40 are proven by CVC5 with only all necessary definitions provided, and no auxiliary lemmas provided. Of the 96 theorems in `Mathlib.Algebra.Group.Commute` translation fails on 52, as we do not have translation of `let`-expressions implemented yet.

Mathlib heavily relies on hierarchic type classes, which lead to long chains of projections in order to obtain a parent type class, which bloats the translation and confuses the SMT solver. For example, both theorems in listing 1.14 are translated properly, but CVC5 is only able to prove `th1`, and times out on `th2`. The proof generated for `th1` is 250kB in size.

```
theorem th1 : 1     + 2 = 3 := by smt
theorem th2 : 1 + 0 + 2 = 3 := by smt
```

Listing 1.14. Two simple examples.

We are able to translate higher order constructs such as `map` and `fold`. CVC5 is able to find a proof with no auxiliary lemmas for theorem `mapConcrete`, but fails for `mapAny`.

```
theorem mapConcrete : List.map (λx, x + 1) [1, 2] = [2, 3] := by smt
theorem mapAny : (l : List Nat) → List.map (λx, x) l = l := by smt
```

Listing 1.15. Higher order examples.

The index erasure is able to handle type indices which depend on other type indices, which has previously been an open question in the Nunchaku paper [17].

7.2 Future Work

Translation is only one of three key ingredients for a hammer, making lemma selection and proof reconstruction the next steps.

So far, we only monomorphize environment items for every usage site. However, we also obtain a large amount of auxiliary lemmas from the lemma selection stage of the hammer. These lemmas do not occur in the goal and have no usage sites, and as such we do not know what to monomorphize them to. An overview of potential heuristics can be found in [9].

When encoding to TPTP we strip all propositions, such as the subtype guard construction used in index erasure. Given a function `f : { t : E // G t } → { t : E // G t }`, stripping the guard will enlargen `f`'s domain. Hence it is necessary to add pre and postconditions as axioms to the TPTP problem to enforce the guards.

Encoding recursors will presumably help prove theorems such as `mapAny` in listing 1.15.

Type classes are used extensively in Lean's Mathlib, with long inheritance chains, causing long projection chains. Flattening structures might help reduce these projection chains.

A Source Code

The source code for lean-hosmt can be found at `https://git.scc.kit.edu/unevo/pdf-hammer`. The latest git commit hash is `d72f5c0b67d34642da383653ca`.

References

[1] "5. The Rules of the Game". In: *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Ed. by Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 67–104. ISBN: 978-3-540-45949-1. DOI: 10.1007/3-540-45949-9_5. URL: https://doi.org/10.1007/3-540-45949-9_5.

[2] Lawrence C. Paulson. *The Foundation of a Generic Theorem Prover*. 2000. arXiv: cs/9301105 [cs.LO].

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

[4] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language". In: *Automated Deduction – CADE 28*. Ed. by André Platzer and Geoff Sutcliffe. Cham: Springer International Publishing, 2021, pp. 625–635. ISBN: 978-3-030-79876-5.

[5] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*. Vol. 9. Bibliopolis Naples, 1984.

[6] Rob Nederpelt and Herman Geuvers. *Type theory and formal proof: an introduction*. Cambridge University Press, 2014.

[7] Christine Paulin-Mohring. *Introduction to the calculus of inductive constructions*. 2015.

[8] Martin Desharnais et al. "Seventeen Provers under the Hammer". 2021. URL: https://matryoshka-project.github.io/pubs/seventeen.pdf.

[9] Jasmin Christian Blanchette et al. "Hammering towards QED". In: *Journal of Formalized Reasoning* 9.1 (2016), pp. 101–148.

[10] Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. "Automatic Proof and Disproof in Isabelle/HOL". In: *Frontiers of Combining Systems*. Ed. by Cesare Tinelli and Viorica Sofronie-Stokkermans. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 12–27. ISBN: 978-3-642-24364-6.

[11] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. "Extending Sledgehammer with SMT Solvers". In: *Journal of Automated Reasoning* 51.1 (June 1, 2013), pp. 109–128. ISSN: 1573-0670. DOI: 10.1007/s10817-013-9278-5. URL: https://doi.org/10.1007/s10817-013-9278-5.

[12] Jasmin Christian Blanchette et al. "Encoding monomorphic and polymorphic types". In: *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 19*. Springer. 2013, pp. 493–507.

[13] Łukasz Czajka and Cezary Kaliszyk. "Hammer for Coq: Automation for dependent type theory". In: *Journal of automated reasoning* 61.1 (2018), pp. 423–453.

[14] Burak Ekici et al. "SMTCoq: A Plug-In for Integrating SMT Solvers into Coq". In: *Computer Aided Verification*. Ed. by Rupak Majumdar and Viktor Kunčak. Cham: Springer International Publishing, 2017, pp. 126–133. ISBN: 978-3-319-63390-9.

[15] 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.

[16] Petar Vukmirović et al. "Making Higher-Order Superposition Work". In: *Automated Deduction – CADE 28*. Ed. by André Platzer and Geoff Sutcliffe. Cham: Springer International Publishing, 2021, pp. 415–432. ISBN: 978-3-030-79876-5.

[17] Simon Cruanes and Jasmin Christian Blanchette. "Extending Nunchaku to dependent type theory". In: *arXiv preprint arXiv:1606.05945* (2016).

[18] Christoph Benzmüller, Florian Rabe, and Geoff Sutcliffe. "THF0–the core of the TPTP language for higher-order logic". In: *Automated Reasoning: 4th International Joint Conference, IJCAR 2008 Sydney, Australia, August 12-15, 2008 Proceedings 4*. Springer. 2008, pp. 491–506.