Towards Modular Composition of Inductive Types Using Lean Meta-programming

Ramy Shahin

Qualgebra
ramy@qualgebra.com

Introduction Inductive types are ubiquitous building blocks in many programming and theorem proving languages. An inductive type is a closed set of constructors from which values of the type can be created. That set of constructors cannot be extended though once a type is defined. This limits extensibility, reuse, and modular separation of concerns when defining types and functions operating over their values. The *expression problem* [13] is one manifestation of this limitation, where extending an expression language with new syntactic constructors while reusing existing ones without having to modify or re-compile them is a challenge in almost all programming languages.

This extended abstract briefly presents a set of syntactic extensions to the Lean programming language that allow the modular compositions of inductive types[1]. Lean 4 [7] includes meta-programming constructs that allow developers to extend the syntax of the language, and provide user-defined elaborators of the extended syntactic constructs. We utilize those meta-programming facilities to allow the definition of inductive types that are built from *clones* of the constructors of constituent types. In addition, we automatically generate coercion operators that allow passing values of constituent types to functions expecting values of the extended type, and vice versa when applicable.

```
namespace Boolean

inductive T where
| Bool

inductive Term where
| True
| False
| If (c t₁ t₂: Term)

inductive TRel: Term → T → Prop
| TT: TRel .True .Bool
| FF: TRel .False .Bool
| If: TRel c .Bool → TRel t₁ τ → TRel t₂ τ
    → TRel (.If c t₁ t₂) τ

end Boolean
```

(a) Boolean type definition.

```
namespace Nat

inductive T where
| N

inductive Term where
| Zero
| Succ (t: Term)
| Pred (t: Term)

inductive TRel: Term → T → Prop where
| Z: TRel .Zero .N
| S: TRel t .N → TRel (.Succ t) .N
| P: TRel t .N → TRel (.Pred t) .N

end Nat
```

(b) Nat type definition.

Figure 1: Separate definitions of `Boolean` and `Nat` types, syntactic terms, and type relation.

[1]Prototype implementation can be found at https://github.com/qualgebra/LeanToolkit/tree/TYPES2025

```
inductive T := Boolean.T + Nat.T

⇓

inductive T : Type
| Bool : T
| N : T
```

(a) Composing `Boolean.T` and `Nat.T`.

```
inductive Term := Boolean.Term + Nat.Term
| isZero (t: Term)

⇓

inductive Term : Type
| True : Term
| False : Term
| If : Term → Term → Term → Term
| Zero : Term
| Succ : Term → Term
| Pred : Term → Term
| isZero : Term → Term
```

(b) Composing `Boolean.Term` and `Nat.Term`.

```
inductive TRel: Term → T → Prop := Boolean.TRel + Nat.TRel
| iz: TRel t T.N → TRel (.isZero t) T.Bool

⇓

inductive TRel : Term → T → Prop
| TT: TRel Term.True T.Bool
| FF: TRel Term.False T.Bool
| If: ∀{c t₁: Term} {τ:T} {t₂: Term}, TRel c T.Bool → TRel t₁ τ → TRel t₂ τ → TRel (c.If t₁ t₂) τ
| Z: TRel Term.Zero T.N
| S: ∀ {t : Term}, TRel t T.N → TRel t.Succ T.N
| P: ∀ {t : Term}, TRel t T.N → TRel t.Pred T.N
| iz: ∀ {t : Term}, TRel t T.N → TRel t.isZero T.Bool
```

(c) Composing `Boolean.TRel` and `Nat.TRel`.

Figure 2: Composing `Boolean` and `Nat` using the '+' operator on inductive types.

Example This example is inspired by the Typed Lambda Calculus (TLC) presentation from [8]. We assume we are defining TLC with two native types: `Boolean` (Fig. 1a), and `Nat` (Fig. 1b). Each of the two separate definitions includes inductive types for the set of relevant types (`T`), the set of valid syntactic terms (`Term`), and a type relation (`TRel`) between terms and types.

Now we would like to compose the two sets of definitions into a language with both `Boolean` and `Nat` native types. We use Lean meta-programming to extend the Lean syntax with a new construct for *summing up* multiple inductive types. Fig. 2 shows three examples: composing `Boolean.T` and `Nat.T` (Fig. 2a), composing `Boolean.Term` and `Nat.Term`, while adding an extra constructor `isZero` (Fig. 2b), and finally composing `Boolean.TRel` and `Nat.TRel`, adding the extra constructor `iz` (Fig. 2c). The '+' operator (implemented and elaborated using Lean meta-programming) is used in all three examples to compose multiple inductive types, and optionally adding extra constructors like in the cases of `Term` and `TRel`. Each of the examples shows the Lean definition automatically generated as a result.

In addition, instances of the `Coe` typeclass are also generated to allow safe automatic coercion from values of the constituent types to the newly defined composite type. For example, the following code snippet typechecks because the automatically generated coercion operator converts `Boolean.T.Bool` into T.Bool:

```
def x := Boolean.T.Bool
def y: T := x
```

Coercion in the opposite direction (e.g., from `T.Bool` to `Boolean.T.Bool`) is possible only for values known at compile time. The Lean standard library includes the dependent coercion typeclass `CoeDep`. We generate instances of this typeclass for each of the constructors of the summed up type, coercing them back to their respective constituent types. As a result, the following Lean definition typechecks:

```
def z: Boolean.T := T.Bool
```

Related Work Previous work tried to reuse proofs on modular definitions in Coq [2, 9], by extending an inductive type by individual extra constructors, and functions with individual pattern matching cases. Modular composition of definitions and theorems into feature-based product lines was presented in [3]. Inspired by the data types a-la-carte work for Haskell [11], similar approaches to solving the expression problem in theorem provers include Meta-theory a la carte [4], Coq-a-la-carte [5], and extensible metatheory mechanization [6]. Other attempts at solving the expression problem include four different solutions relying on generic data types in Java-like languages are presented in [12], and a symmetric view of algebraic data types and codata types [1].

Our approach of composing constructors is similar to that of Boite [2] with three main differences that we know of.

1. Boite's approach incrementally adds constructors to an existing type, while we focus on composing multiple types, and also support adding extra constructors if needed. We also rely on coercion operators for interoperation between composed and constituent types.
2. We heavily leverage Lean metaprogramming to simplify the implementation, while Boite's work predates MetaCoq [10].
3. This is more of a limitation on our side at this point, we do not support composing proof objects. This is one of our future work directions.

Acknowledgments The author would like to thank anonymous reviewers for their feedback and insightful suggestions.

References

[1] David Binder, Julian Jabs, Ingo Skupin, and Klaus Ostermann. Decomposition diversity with symmetric data and codata. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[2] Olivier Boite. Proof Reuse with Extended Inductive Types. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics*, pages 50–65, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[3] Benjamin Delaware, William Cook, and Don Batory. Product Lines of Theorems. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 595–608, New York, NY, USA, 2011. ACM.

[4] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. *SIGPLAN Not.*, 48(1):207–218, January 2013.

[5] Yannick Forster and Kathrin Stark. Coq à la carte: a practical approach to modular syntax with binders. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 186–200, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Ende Jin, Nada Amin, and Yizhou Zhang. Extensible metatheory mechanization via family polymorphism. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.

[7] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, page 625–635, Cham, 2021. Springer International Publishing.

[8] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[9] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Ornaments for Proof Reuse in Coq. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[10] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, February 2020.

[11] Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, July 2008.

[12] Mads Torgersen. The expression problem revisited. In Martin Odersky, editor, *ECOOP 2004 – Object-Oriented Programming*, pages 123–146, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[13] Philip Wadler. The Expression Problem. `https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`. *Note to Java Genericity mailing list.*