Extending programs and proofs

ANONYMOUS AUTHOR(S)

Interactive theorem provers (ITPs) do not easily allow users to adapt and extend existing codebases without either changing the original code or duplicating it. This problem, referred to as the *the expression problem*[32], is present in virtually all programming languages, but is exacerbated in ITPs by the need to also extend proofs. The pattern of extending definitions occurs particularly often when mechanizing programming language theory, type theory, and program verification, leading to a lot of code duplication and high maintenance burden. Existing approaches to the problem either require manipulating non-trivial encodings—obfuscating the development— or rely on custom intermediate representations of syntax via metaprogramming facilities, which were underdeveloped in most ITPs until recently. In particular, none of the existing tools provide ways to extend formalisations *after the fact*, *i.e.*, being able to extend an independent mechanisation without the need to adapt it to the tool used to extend it.

We approach the problem with the insight that, conditional on working in a proof-assistant with appropriate metaprogramming tooling, no intermediate representations are needed to provide extensibility. This in turn allows for extensible reasoning at no added cost to the user-experience. In particular, this allows for extending non-extensible codebases without needing to adapt any of the original code.

To demonstrate this, we develop **Gemel**[1] : a production-ready tool implemented in Lean 4 to produce extensible code, leveraging the strengths of dependent types and modern metaprogramming techniques. We showcase the strengths of the tool through various case-studies, showing a measurable decrease in the number of lines written when using the tool, at a reasonable performance cost.

CCS Concepts: • **Type Systems**; • **Software Verification**; • **Syntax**;

Additional Key Words and Phrases: Modular Programming, Type Theory, Proof Assistants, Lean

1 INTRODUCTION

A central limitation in the adoption and use of interactive theorem provers (ITPs) is their verbosity: the size of mechanized formalisations, relative to their pen-and-paper counterparts, tends to be bigger by a measurable factor, referred to as the “de Bruijn factor” [33]. In that regard, one luxury informal constructions possess over mechanised ones is the ability to be extended with additional data. Whatever proofs were done on the original construction are carried over and adapted to the extended version, making abstraction of the work shared between the two. For programming languages and type systems in particular, it is common to build new concepts by extending previous systems[12, 17, 19, 22–24].

In comparison, the design of ITPs does not generally allow for such extensibility: formalisations use inductive types, as well as functions and proofs about them. These encode the relevant objects and properties getting mechanised. The dependent type theories popular ITPs rely on do not provide internal notions of extensions for inductive type, let alone for functions and proofs that would make use of extended types. Two basic solutions arise to counteract this issue: copy-pasting, and indexing. The former, copy-pasting, consists of "extending" a formalisation by simply duplicating it, and extending the duplicate constructions by hand until the extended results are proved. The latter, indexing, consists of "bundling" all the potentially interesting extensions of a given construction into one big inductive type, indexing that type with ways to toggle each extension’s features. Copy-pasting, while being a "working" solution, ignores the reality of mechanizations: these evolve, get extended, and get refactored to have their performance, readability and usability improved over time. These maintenance efforts tend to be proportional to the size of the codebase being maintained. Having to repeat the same maintenance work over multiple extensions of a mechanisation is

[1] Pronounced gem·el, describes pairs of trees that have undergone inosculation, *i.e.*, the act of connecting and merging with one another.

$$x, y : \text{var}$$
$$s, t : \text{term} ::= \lambda x.\ s \mid s\ t \mid x$$

$$size(x) := 0$$
$$size(\lambda x.\ t) := size(t) + 1$$
$$size(s\ t) := size(s) + size(t) + 1$$

Lemma 1.1. $\forall (t : term), size(t) >= 1$

Proof. Proof by cases:
(1) case x: ...
(2) case $\lambda x.\ s$: ...
(3) case $s\ t$: ...

□

A grammar for lambda-calculus

$$s, t : \text{term} ::= \ldots \mid (s, t) \mid let(x, y) := s \text{ in } t$$

$$size((s, t)) := size(s) + size(t) + 1$$
$$size(let(x, y) := s \text{ in } t) := size(s) + size(t) + 1$$

Lemma 1.2. *1.1 applies to the extended term*

Proof. Extending on the proof by cases:
(1) case (s, t): ...
(2) case $let(x, y) := s$ in t: ...

□

An extension with products

thus not viable. To showcase this issue, Adjedj et al. [2] present a mechanisation of Martin-Löf Type Theory[21] (MLTT), and was extended using copy-pasting to produce mechanizations with new features such as subtyping[17] and quotations[22]. In parallel to those extensions getting constructed, the original mechanisation was refactored to showcase clearer dependencies between the meta-theoretic properties of the type-system[18], allowing for easier mechanisation efforts. None of the extensions got to make use of this refactor because the burden of porting it was too high. Indexing on the other hand, while seemingly avoiding the issue of burden of maintenance, does not in fact solve the issue of extensibility: not only may some extensions not be compatible with one another, the expression problem re-appears as soon as a wanted extension is not already part of the initially produced bundle (going back to the previous example of extensions for MLTT, whether the two aforementioned extensions would be compatible is unknown). Furthermore, shaping a mechanization this way requires a non-trivial amount of work, which slows down the initial mechanisation greatly.

To solve this instance of the *the expression problem*[32], many tentative solutions have been produced[9, 11, 13, 15, 16, 25]. These solutions either make use of encodings[9, 15, 16, 25], or use metaprogramming to achieve their results[11, 13]. Encodings are great for extensibility: working with the semantic interpretation of a datatype rather than its syntax allows for internalizing its extensibility properties, making them explicit and available for the user. On the metaprogramming side, existing solutions, all written in Rocq, expose some additional syntax for the user to write in order to produce extensible formalisms.

We however believe all existing solutions fall short of the benefits such a tool is expected to offer. We pinpoint 3 concerns extensibility solutions should address in order to get adopted, and justify the lack of widespread adoption from existing tools by the fact none of them fill all these requirements.

Full Expressivity. Extensions need to be as expressive as their non-extensible counterparts: an extensible mechanization should not need to give up anything relative to one done using copy-pasting. Solutions using encodings tend to either have explicit results showing their expressivity is limited, or have no results at all to study the question. It is safe to assume, in general, that encodings to do not provide, in general, the same features that a native inductive type would. This lack of features may in particular include not being able to encode some complex features of inductive

types such as nested, mutual or indexed types, having too weak an induction principle, or having an induction principle which does not possess the expected reduction rules. Solutions that rely on metaprogramming tend to not share this issue.

User-friendly syntax. The user-experience of working with extensions should be as similar as possible to that of writing non-extensible code. The learning curve for ITPs being already fairly steep, softening that learning curve when having to use a new tool is important to ensure its adoption. Encodings-based solutions fail to manage this: the experience of manipulating encodings tends to be measurably worse than that of using inductive datatypes. Indeed, features like pattern-matching, recursion and tactics tend to not interact well with encodings. On the other hand, metaprogramming should in theory offer the tools necessary to achieve this. In practice however, all existing metaprogramming solutions rely on some sort of intermediate representation which they manipulate to construct and extend types and proofs. Having a good user-experience would require these representations and their APIs to provide feature parity when compared with the regular declarations/syntax trees used by the ITP. This would require a lot of efforts, as ITP frontends (a.k.a elaborators) can be hundreds of thousands of lines of code long and provide a lot of features (e.g docstrings, arbitrary attributes, LSP data, extraction instructions, lazily-generated auxiliary lemmas), some of which may not even be exposed by the metaprogramming APIs offered by the language. We can see a glimpse of the work that would be needed for such adaptations in the fact all existing tools relying on metaprogramming have implemented variants of existing tactics to adapt them to their specific use-case (e.g `fconstructor` for Rocqet, `mconstructor` for Rocq à la carte, both variants of the existing `constructor` tactic).

***After the fact* extensions.** One shared limitation, present in all past solutions, is the inability to extend mechanisations which were not made with extensibility in mind: they need to have been initially built using the proposed framework in order for said framework to work. This is particularly limiting for the adoption of these solutions, as any mechanisation that would want to benefit from using one of these tools would first need to be significantly refactored. As such, we consider the ability to do *after the fact* extensions to be a crucial feature for such tools' adoption in real-world usage. This limitation seems intrinsic to the way encoding-based solution function: translating an inductive type into an equivalent encoding can be a non-trivial task, and no existing encoding provides the assurance that the encoded type possesses the same definitional equalities as the original one. This means that translating functions defined over the inductive type may lead to differences in their reduction behaviour, which in turn can lead to type-checking failures down the line. Metaprogramming may again offer the flexibility needed to resolve this issue. However, for existing solutions, the obstacle of having to encode regular declarations into their respective intermediate representations means would be a lot of work, and might require having to extend these IRs substantially to adapt for this need, making the systems much more complex.

In light of this, we turn our attention to Lean 4. This ITP has a fully extensible parser and elaborator, allowing users to extend its syntax arbitrarily. It is furthermore bootstrapped (*i.e.,* Lean is written in Lean itself), which means that all of the robust tooling and APIs used to make the language is made available for arbitrary metaprograms. In particular, we make the observation that extensibility tools based on metaprogramming need not rely on intermediate representations, and that avoiding these allows to make full use of the robust tooling provided by such an ITP:

An extensible proof assistant allows for extensible proofs

We present **Gemel**, a new framework for writing extensible code, written in the Lean 4 proof assistant. This library aims to provide a user-experience as similar to writing regular non-extensible

	Rocq à la Carte	Meta-theory à la Carte	Generic Datatypes à la Carte	Modular Type-Safety Proofs in Agda	Pyrosome	Rocqet	**Gemel**
Extensibility technique	MP + IR	Encoding	Encoding	Encoding	Encoding	MP + IR	MP
Full expressivity	✓	✗	✓	✗	✗	✗	✓
User-friendly syntax	✗	✗	✗	✗	✗	✗	✓
After the fact extensions	✗	✗	✗	✗	✗	✗	✓

(MP) : Metaprogramming, (IR): Intermediate Representation

code as possible. This objective is achieved by avoiding the use of any internal encodings, manipulating Lean's concrete and abstract syntax trees directly, and making use of the powerful metaprogramming capabilities of the language. **Gemel** declarations integrate all the features of regular Lean declarations such as dependent pattern-matching, tactic-based proofs, structural and well-founded recursion, except for mutual and nested inductive types as well as mutually recursive functions. In particular, **Gemel** also allows for *after the fact* extensions.

```
inductive Term where
  | var (x : Nat)
  | lam (t : Term)
  | app (s t : Term)

def Term.size : Term → Nat
  | var _ => 0
  | lam t => size t + 1
  | app s t => size s + size t + 1

theorem Term.size_ge_1 (t : Term)
  : t.size >= 1 := by
  cases t
  case var _   => ...
  case lam _   => ...
  case app _ _ => ...
```

A grammar for lambda-calculus in Lean

```
mod inductive ProdTerm extends Term where
  | mkProd (s t : Term)
  | letProd (x y : Var) (s t : Term)

mod def ProdTerm.size extends Term.size where
  extend with
  | mkProd s t      => size s + size t + 1
  | letProd _ _ s t => size s + size t + 1

mod def ProdTerm.size_ge_1
  extends Term.size_ge_1 where
  finally
  case mkProd _ _       => ...
  case letProd _ _ _ _  => ...
```

An extension with products using **Gemel**

To demonstrate the capacities of our system, we present case-studies implemented using **Gemel**. Two of these case-studies extend **independently-produced** formalisations of type-systems-STLC and System F-with new constructions, extending key results of the original formalisations to the extended calculi. While such formalizations form a recurring case-study in previous works on extensibility in ITPs, ours is the only one that shows the capacity to build such extended formalizations *after the fact*, *i.e.,* on a formalization not made with extensibility in mind.

Another case study consists of a construction of STLC and a Continuation-Passing Style (CPS) calculus, as well as a translation pass from the former to the latter. The formalisation then extends both STLC and CPS with various constructions, and extends the translation pass between the respective extensions. This reproduces the central case study produced by Jamner et al. [15] in our own framework, without the need for any encodings.

Together, these case-studies serve to show that the fundamentally basic ideas behind **Gemel** are powerful enough to scale up to all features provided by previous tools, as well as allowing one to extend an independent formalisation *after the fact*, which is unique to our framework.

Contributions. We construct **Gemel**, a new tool for extending programs and proofs in Lean 4:

(1) The tool is production-ready
(2) It builds on a novel paradigm, relying fully on the extensive metaprogramming facilities provided by Lean 4, avoiding the need for any intermediate representation for our extensions. (§2)

(3) This paradigm allows for *after the fact* extensions, a central feature for the adoption of extensibility tools that is missing from all other existing solutions. (§3.1,§3.2)
(4) *After the fact* extensions have feature-parity with regards to regular lean declarations, including dependent pattern-matching, tactic proofs and arbitrary recursion strategies. (§3.1, §3.2)
(5) **Gemel** can merge extensions, allowing for true extensibility. (§4)
(6) We show the usefulness of the tool through various case-studies, showcasing a reduction in lines of code by a factor of 3, and elaboration times on par with regular Lean declarations. (§5)

2 REPRESENTATION OF EXTENSIBLE SYNTAX

We examine existing solutions to producing extensible code in ITPs, showing their respective strength and weaknesses, before explaining our framework for extending syntax in **Gemel**. We first provide a quick description of "Data types à la Carte", a central contribution to answer *the expression problem* for functional programming languages, despite it not working for ITPs, as it is influential to many proposed solutions in this context.

Data types à la Carte. Swierstra [29] proposes the idea of encoding "extensible inductives" as least-fixedpoints of arbitrary functors, i.e functions `f : Type → Type` equipped with a mapping operator `fmap : (α → β) → f α → f β` for any `α, β : Type`.

```
inductive Expr (F : Type → Type) where
  | in : F (Expr F) → Expr F
```

Consider the following type families:

```
inductive ExpVar (exp : Type): Type where
  | var : Nat → ExpVar exp
```

```
inductive ExpLam (exp : Type): Type where
  | app : exp → exp → ExpLam exp
  | lam : exp → ExpLam exp
```

The observation made by the paper is that these functors can be summed into a new functor `ExpVarLam (exp : Type) := ExpVar exp ⊕ ExpLam exp`. The least fixedpoint of this type family is then definable as `Expr ExpVarLam`. The paper also presents how type-class resolution can be used to easily coerce from one instance of `Expr` (e.g `Expr ExpVar`) to an extension of it (e.g `Expr ExpVarLam`), and similarly for functions defined by induction on such types. Unfortunately, the same principle fails in ITPs. In order to ensure the soundness of these systems, inductive types may only occur recursively in "strictly positive" positions in their constructors. In particular, in the definition of `Expr`, `F (Expr F)` is not a strictly positive position. However, the idea of using functor fixpoints to encode extensibility, coined as the "à la carte" technique, is still employed in solutions proposed for ITPs.

Rocq à la Carte. Forster and Stark [13]'s design centers on manipulating user-defined datatypes constructed as "inlined" à la carte datatypes. Using the previous example of "Data types à la Carte", rather than rely on an explicit fixpoint `Expr` to construct `Expr ExpVarLam`, Rocq à la carte instead asks of a user to define the type in an inlined fashion:

```
inductive Exp₁ where
  | expvar : ExpVar Exp₁ → Exp₁
  | explam : ExpLam Exp₁ → Exp₁
```

Such a type, unlike the `Expr` type, is considered to be a strictly positive (nested) inductive type. Functions over types defined using the functors can then be defined by first constructing said functions over the various feature functors, and composing them in an appropriate recursive function for the composed type:

```
def ExpVar.count (f : exp → Nat)
  : ExpVar exp → Nat
  | var _ => 1

def ExpLam.count (f : exp → Nat)
  : ExpLam exp → Nat
  | app e₁ e₂ => (f e₁) + (f e₂)
  | lam e => f e

def Exp₁.count : Exp₁ → Nat -- Error: Failed to show termination
  | expvar v => ExpVar.count Exp₁.count v
  | explam v => ExpLam.count Exp₁.count v
```

However, replicating (and extending on) this technique to bring extensibility tooling in the Lean proof assistant is not really feasible. The previous code snippet would work in Rocq, but throws a non-termination error in Lean. Indeed, functions in ITPs are required to be total for the sake of ensuring the soundness of the type-system (*i.e.,* the inability to prove `False`). Every popular ITP ensures this restriction differently (this is discussed more in depth in Section 3.2.2), but an important divergence between Rocq and Lean is that Rocq's system for detecting structural recursion works up to reduction, whereas Lean doesn't. This feature is paramount to Rocq à la Carte's method for producing extensible code, and cannot be easily replicated in Lean due to divergences in the Lean's design decisions.

Meta-theory à la Carte. Delaware et al. [9] attempt to work around the issue of non-positivity by working with an encoding of datatypes which allows for à la Carte techniques in ITPs, namely Mendler-style[30] Church encodings[6]. This provides a way to produce "extensible datatypes" at the cost of having to use additional abstractions relative to using datatypes that are native to the ITP. These encodings can unfortunately only be constructed in Rocq with a specific option turned on (namely, impredicative `Set`). This option is rarely used, incompatible with common axioms for mathematics, and not available in any other popular ITP. Furthermore, these encodings are weaker than Rocq's native inductive type in the sense that their induction principles are generally weaker. Because of this, users can prove less things about these encodings than they can about their inductive counterparts.

Modular Type-Safety Proofs in Agda. Schwaab and Siek [25] adapt the "Data types à la Carte" approach to Agda by defining a type of strictly positive functors, and a function mapping said functors to type-level functions:

```
inductive Functor where
  | id
  | const  (T : Type)
  | prod   (F G : Functor)
  | coProd (F G : Functor)

def eval : Functor → Type → Type
  | id       , X => X
  | const T  , _ => T
  | prod F G , T => F T × G T
  | coProd F G, T => F T ⊕ G T

inductive Expr (F : Functor) where
 | in : eval F (Expr F) → Expr F
```

This definition of `Expr` passes Agda's positivity checker, but does not do so for Rocq and Lean. Furthermore, Schwaab and Siek [25] mention that their final proof which composes all the features of their case-study does not pass Agda's termination-checker, showing limitations to the method used.

Generic Datatypes à la Carte. Keuchel and Schrijvers [16] combine ideas of "Modular Type-Safety Proofs in Agda" and "Meta-theory à la Carte" to avoid shortcomings of both solutions. Namely, their approach subsumes the concept of strictly positive functors of Schwaab and Siek [25], doesn't necessitate working in an impredicative universe like Delaware et al. [9] do, and provides the right

induction principles for the encoded types. Their encoding of choice is containers[1]. A container consists of a type of codes and an interpretation function mapping codes to types. This solution still carries the burden of being based on encodings: constructing the right APIs around these container construction requires a lot of boilerplate, exposes internal details of these encodings to the user, and does not provide the same user-experience as using inductive datatypes.

Pyrosome. Jamner et al. [15] also relies on encodings of datatypes to achieve extensibility. The encoding provided by this tool is Generalised Algebraic Theories[5], also known as GATs. A GAT consists of a set of rules that either introduce sorts (i.e types being defined), constructors for those sorts (i.e constructors of inductive types), and equations describing ways in which constructors may be equal to one another. Rather than a general "à la Carte" approach, the tool focuses on formalising compiler IRs and passes. In particular, extending GATs, as well as functions from one GAT to another (e.g between two compiler IRs) is made easy, at the cost of having to work with a non-trivial abstraction. The extensibility results that the tool exhibits are however fairly limited to the concept of proving semantic preservation for type-systems. In particular, the system does not provide any theoretical results about its definition of GATs: no induction principles are provided, and the equations for a GAT do not relate to Rocq's internal notion of equality. Instead, for a given GAT, Pyrosome defines an appropriate "equivalence class" for the types being defined. In particular, this equivalence class does not benefit from Rocq's elimination principle for equality. These issues greatly limit the amount of constructions and proofs that are doable using this tool.

Rocqet. Similarly to Pyrosome, Ebresafe et al. [11]'s design focuses on compiler verification specifically, rather than generic extensible programming. The tool structures the code around building and extending "families", *i.e.,* bundles of inductive definitions and fixpoint functions, to construct translation passes between the extensions. Rocqet differs however from Pyrosome in the fact that it does not rely on encodings to construct these types, definitions and proofs, but on metaprogramming instead. However, the notion of inductive types and functions made available by the tool are not exactly those provided by the Rocq language. Instead, Rocqet possesses its own representation for these objects, which it can feasibly extend. These families and extension then get translated back into real Rocq declarations which don't rely on encodings. Working on such an intermediate representation induces limitations in the tool's design, as its fixpoint function can only case-match at the top-level, and can only do structural induction (i.e doesn't handle well-founded induction). It furthermore means the tool cannot do *after the fact* extensions.

Looking at these existing tools, we believe that avoiding encodings leads to a better user-experience, and is necessary to allow for doing *after the fact* extensions. In that regard, the solutions proposed by Rocqet are the most promising. We however make the observation that this tool is limited by its reliance on a custom intermediate representation. This reliance means that the tool needs, in effect, to reinvent existing APIs, syntax, tactics and interactions for existing Rocq commands to adapt them to their own need. The issue is further exacerbated by the lack of metaprogramming tooling provided by Rocq to help improve the situation. As such, we turn our attention to Lean 4, and decide to construct **Gemel** using metaprogramming, with an emphasis on reusing Lean's abstract and concrete internal syntax trees, which allows us to make full use of the language's existing tooling.

3 MODULAR EXTENSIONS

We want to be able to translate any definition or theorem written about some inductive types into one that instead mentions extended versions of said types. We refer to this translation function as an **extension map**.

Let's consider the previous example presented in the introduction: the user had defined an inductive type `Term`, a function mapping a `Term` to its size (i.e the number of nodes that compose it), and a proof by case showing the size is always greater or equal to 1. The user then wants to define a new type `ProdTerm` corresponding to `Term` extended with two new constructors for constructing and eliminating products, before extending the definition of `size` and the associated proof.

An extension mapping function needs to traverse the definition of `Term.size`, and transform every occurrence of `Term` to `ProdTerm`. Similarly, each constructor of `Term` must get mapped accordingly, *i.e.,* `Term.var x` maps to `ProdTerm.var x` and similarly for `Term.lam` and `Term.app`. Such a translation need only be structural, keeping the shape of the original function while transforming the constants present inside it. To know what constants to transform, and into what, an extension map depends on a **mapping context**, *i.e.,* a partial function mapping some constants of the global environment to new terms. This mapping context gets expanded as more things are translated, *e.g.,* after extending the `Term.size` function to `ProdTerm.size`, the mapping context will get appropriately expanded, mapping the former to the latter.

The following sections showcase how **Gemel** makes use of this system of extension maps to achieve its goal of providing a user-friendly system of extensions.

3.1 Inductive Types and recursors

We use our system of extension maps to provide users with ways to extend previously defined programs and formalizations. From a user perspective, a Lean program mainly consists of defining new datatypes, definitions and theorems. The main way to construct new datatypes in Lean is by defining new inductive types [8]. This section focuses on how **Gemel** allows users to extend inductive definitions through the addition of new constructors.

3.1.1 Inductive types in type theory. First, let us look at what an inductive type is. The general intuition of an inductive type is that it is a regular datatype, of which terms can be constructed by using recursors, and which can be eliminated through the use of a recursor, *i.e.,* an induction principle. Inductive types are shaped as follows:

$$\begin{aligned}&\textbf{inductive}\ \mathrm{Ind}\ \overrightarrow{(p : P)} : \overrightarrow{(i : I)} \to \mathtt{Type}\ \textbf{where}\\ &\mid \textbf{ctor}_1 : \overrightarrow{(a_1 : A_1)} \to I\ \overrightarrow{p}\ \overrightarrow{d_1}\\ &\quad \dots\\ &\mid \textbf{ctor}_n : \overrightarrow{(a_n : A_n)} \to I\ \overrightarrow{p}\ \overrightarrow{d_n}\end{aligned}$$

An inductive type is composed of a number of different components: A list of parameters $\overrightarrow{(p : P)}$ which are uniform in that they stay the same in every occurrence of the type in its constructors, a telescope of indices $\overrightarrow{(i : I)}$ that may vary in each occurrence, and a list of constructors for this type. Each constructor contains a list of fields living over the context formed by the parameters, and instantiates the indices of the inductive type they inhabit in a context containing those fields. Basic examples of inductive types include the natural numbers, lists, and vectors (*i.e.,* lists indexed by their lengths):

```
inductive Nat : Type where
  | Z : Nat
  | S : Nat → Nat
```

```
inductive Vec (A : Type) : Nat → Type where
  | nil  : Vec A Z
  | cons {n}: A → Vec A n → Vec A (S n)
```

`Nat` is an inductive type with no parameters and no indices, as well as two constructors `Z` and `S`, the first one containing no field and the second one recursive occurrence of `Nat` as its sole field. In comparison, `Vec` is an inductive type which has one parameter `A`, is indexed by a natural number,

and has two constructors. The former has no fields and instantiates its index at 0, the second has 3 fields with one recursive one. `Vec A n` can be interpreted as the type of lists of `A` of size `n`.

To each inductive type is associated a recursor, *i.e.,* a way to eliminate a term of that type. For example, the recursor for `Nat` has the following type:

$$\texttt{Nat.rec} : (\textbf{motive} : \texttt{Nat} \to \texttt{Type}) \to \textbf{motive}\ \texttt{Z} \to ((n : \texttt{Nat}) \to \textbf{motive}\ n \to \textbf{motive}\ (\texttt{S}\ n)) \to (t : \texttt{Nat}) \to \textbf{motive}\ t$$

The existence of that recursor can be interpreted as the statement that, given a predicate **motive** on natural numbers, an instance of that predicate on 0, and for every n, an instance of **motive** (`S` n) given **motive** n, the predicate holds for any natural number t. This corresponds exactly to the regular induction principle for natural numbers. Furthermore, recursors hold computational content in that they may reduce when applied to a constructor. For example, the two reduction rules for `Nat.rec` are as follows:

$$\texttt{Nat.rec P PZ PS Z} \rightsquigarrow \texttt{PZ}$$

$$\texttt{Nat.rec P PZ PS (S } n\texttt{)} \rightsquigarrow \texttt{PS } n \texttt{ (Nat.rec P PZ PS } n\texttt{)}$$

3.1.2 Inductive types in ***Gemel****.* With the general structure of an inductive type in mind, we can now consider ways in which one may extend an inductive type into another. **Gemel** provides a new Lean command which, given an inductive and new constructors, constructs another inductive type which extends the original one with these new constructors, adding the relevant mappings between from the first to the second. The syntax is as follows:

```
mod inductive <new inductive> extends <old inductive> where
  <new constructors>
```

Consider the following example: A user defines a notion of a variable type `Var`, which only has one constructor consisting of a natural number.

```
inductive Var where
  | var : Nat → Var
```

This representation for variables is common in both imperative and functional intermediate representations of programs. After producing an extensive API for this type, one may want to use this notion of variables as part of another type. Consider a new type `Term` which corresponds to that of a lambda-calculus. Such a calculus is constructed using variables, lambdas and applications. Such a type can be defined by extending the variable type as follows:

```
mod inductive Term extends Var where
  | lam : Term → Term
  | app : Term → Term → Term

#print Term
/- inductive Term : Type
   constructors :
   Term.lam : Term → Term
   Term.app : Term → Term → Term
   Term.var : Nat → Term -/
```

Similarly, one may want to extend the lambda-calculus with other constructs, such as booleans. Since `Term` is also just an inductive type, it can itself be extended with new constructions:

```
mod inductive BoolTerm extends Term where
  | true  : BoolTerm
  | false : BoolTerm
```

```
  --- if  c  then  a  else  b
  | ite : BoolTerm → BoolTerm → BoolTerm → BoolTerm
-- λ b => if b then false else true
#check lam (ite (var 0) false true) -- BoolTerm
```

Internally, after constructing this new type, **Gemel** adds the relevant mappings from the old type to the new one in the mapping context: Consider the previous example of `Var` and `Term`, their respective recursors are as follows:

```
Var.rec  : (motive : Var  → Type) → ((a : Nat)  → motive (var a)) →
           (t : Var) → motive t
Term.rec : (motive : Term → Type) → ((a : Nat)  → motive (var a)) →
           ((a : Term) → motive a → motive (lam a)) →
           ((f t : Term) → motive f → motive t → motive (app f t)) →
           (t : Term) → motive t
```

Consider a pretty-printing function for `Var`, defined explicitly using the type's recursor:

```
def Var.repr (t : Var) : String := Var.rec (fun _ => String) (fun x => s!"{x}") t
```

We see that the motive, existing minor for `var` and the major `t` can be translated into a `Term.rec` call as follows:

```
def Term.repr (t : Term) : String := Term.rec (fun _ => String) (fun x => s!"{x}") ?lam ?app t
```

Where `?lam` and `?app` are metavariables, *i.e.,* holes that still need to be resolved for the definition to be valid. These holes correspond to the branches for the `lam` and `app` cases respectively. The function can then be completed by giving a value for each hole:

```
def Term.repr (t : Term) : String := Term.rec (fun _ => String) (fun x => s!"{x}")
                       (fun _ sf => s!"λ {sf}") (fun _ _ sf st => s!"{sf} {st}") t
```

More generally, `Var.rec motive pvar t` can be extended to `Term.rec motive' pvar' ?lam ?app t'` where `motive'` is the extended mapping of `motive`, similarly for `pvar'` and `t'`. Note that metavariable holes such as `?lam` and `?app` cannot, in general, be resolved automatically by Lean, and are instead meant to be resolved by the user; the following section on definition extensions discusses the interface provided by **Gemel** for that purpose in more detail.

To generalize the previous example, consider an inductive `A` of parameters $\overrightarrow{(p : P)}$, indices $\overrightarrow{(i : I)}$ and constructors $\overrightarrow{\textbf{ctor}}$, as well as another type `B` with the same parameters and indices, as well as the same constructors plus a new constructor **newctor**. Extending this to manage adding multiple constructors is trivial, we omit this detail here. We can map the type A to B and respectively every constructor of A to B's. The last missing piece of this translation is the recursor. translation is the recursor.

Recursors have the following shape:

$$\begin{aligned}\text{A.rec} : \overrightarrow{(p : P)} \to (\textbf{motive} : \overrightarrow{(i : I)} \to A\ \overrightarrow{p}\ \overrightarrow{i} \to \text{Type}) \to \overrightarrow{(\textbf{minor} : \ldots \to \textbf{motive}\ (\textbf{ctor}\ \ldots))} \\ \to \overrightarrow{(i : I)} \to (a : A\ \overrightarrow{p}\ \overrightarrow{i}) \to \textbf{motive}\ \overrightarrow{i}\ a\end{aligned}$$

where for every constructor of the type, there is a `minor` covering the (potentially recursive) case associated to it. `B`'s recursor is very similar, except it contains an additional minor for the `newctor`'s case. **Gemel** maps any application with head `A.rec` into `B.rec` by mapping each argument (*i.e.,* the parameters, motive, minors and major), and adding a metavariable hole for the minor corresponding to `newctor`, *i.e.,* A.rec $\overrightarrow{p}$ **motive** $\overrightarrow{\textbf{minor}}$ $\overrightarrow{i}$ a gets mapped to B.rec $\overrightarrow{p'}$ **motive′** $\overrightarrow{\textbf{minor}'}$ $?m$ $\overrightarrow{i'}$ a', where **motive′** corresponds to the mapping of **motive**, and similarly for all other arguments.

This mapping is automatically produced and added to the mapping context upon constructing a `mod inductive`.

In addition to the primitives surrounding an inductive type, Lean also automatically produces more than a dozen auxiliary declarations, meant to make the automation and user-experience friendlier. For example, upon defining an inductive type `Foo`, Lean will, for every constructor `ctor`, define a helper lemma `Foo.ctor.inj` which proves the injectivity property of that constructor. In order to avoid making users define mappings between auxiliary declarations produced by the original and the extended type by hand, **Gemel** automatically generates said mappings automatically (e.g. `Var.var.inj` gets automatically mapped to `Term.var.inj` in the previous example).

A current limitation of **Gemel** is that the tool does not support extending mutual and nested inductive types. This is not a fundamental limitation of the techniques used by the tool, but rather a matter of further engineering, left for future work.

3.2 Extending definitions

Our framework now allows for inductive types to be extended through the addition of new constructors. This section focuses on how declarations (*i.e.,* definitions and theorems) can be extended and extension mapped correctly.

Going back to the previous example of `Term` extending `Var`, consider a function `Var.repr`, this time written idiomatically using pattern-matching rather than a recursor:

```
def Var.repr : Var → String
  | var x => s!"#{x}"
```

Currently, a user would require duplicating the `var` case in order to define a `Term.repr` function, e.g. as follows:

```
def Term.repr : Term → String
  | var x   => s!"#{x}"
  | app f x => s!"{Term.repr f} {Term.repr x}"
  | lam f   => s!"λ {Term.repr f}"
```

This, however, leads to an added burden of maintenance: if the user was to change how variables are pretty-printed in the `Var.var` case, this change would have to be copy-pasted for the `Term.var` case, which would be hard to track in general. Instead, **Gemel** introduces a new `mod def` syntax to allow users to reuse existing branches in a previously defined function, and only require them to fill-in the holes needed to go from a definition over one type to a definition of the same shape over an extension of said type. This syntax allows to rewrite `Term.repr` as follows:

```
mod def Term.repr extends Var.repr where
  extend with
  | app f x => s!"{Term.repr f} {Term.repr x}"
  | lam f => s!"λ {Term.repr f}"
```

Now, any change done to `Var.repr` would also change downstream of it for `Term.repr`.

The general elaboration process of that syntax is as follows. Given a declaration (e.g. `Var.repr`), one may extension map its body into a function over the extended type (here `Term`). Once that is done, users may be asked to fill in any remaining holes in the translated body to construct a well-formed declaration. Once that is done, the new declaration (here `Term.repr`) can be safely added to the global environment of declarations, and a mapping from `Var.repr` to `Term.repr` can be added to the mapping context.

In practice, implementing all of this in Lean has exposed some complications which necessitates careful design decisions, in particular with regards to how pattern-matching and recursion are to be managed. These two features are handled very differently by popular dependently-typed

ITPs. In Rocq, matches and fixpoints are distinct primitive operations made explicit in the abstract syntax. Agda bundles these two notions in a single top-level primitive called a case-tree, where each function (and pattern-match present in a function) gets translated into a top-level case-tree. In both systems, termination-checking is a syntactic criteria that goes through a given fixpoint definition and checks that every recursive call in a function is ran on a strict subterm. Lean diverges from the tradition of having primitive operations, both for pattern-matching and for recursion, and elaborates both down to inductive recursor calls, following and extending on techniques described in Goguen et al. [14] and Cockx and Abel [7]. For pattern-matching, it may appear as though Lean does have match expressions: they are part of the concrete syntax and get appropriately pretty-printed when looking back at the abstract syntax. Take for example a function of the form

```
def is_var_zero : Var → Bool
  | var 0   => true
  | var (n+1) => false
```

The shape of the match get elaborated into an auxiliary function[2]:

```
is_var_zero.match_1 : (motive : Var → Type) → (x : Var) →
                      (Unit → motive (var 0)) → ((n : Nat) → motive (var (n + 1))) → motive x
```

The function is then applied with the right arguments to explicit the return type of the match-here called the `motive`-, the discriminant -here `x`-, and the right-hand-side of each branch:

```
def is_var_zero : Var → Bool := fun x =>
  is_var_zero.match_1 (fun _ => Bool) x (fun _ => true) (fun _ => false)
```

For recursions, when a recursive function is defined, Lean tries to elaborate the function either into a structurally recursive term, using the recursor of whatever term decreases structurally in the recursive calls to write the function or tries to prove the function be well-founded, morally recursing over the accessibility predicate, similarly to Rocq's Equations [28].

3.2.1 Extending pattern-matches. When it comes to extending matches, we are presented with two options. The first one consists of recognizing that matches are simply encoded using recursors. As such, we may simply extension map a given matcher and ask users to fill in the holes in it. This is unreasonable, as it exposes far too many internal details. The definition of a given match can quickly become complex if it matches on multiple elements, needs to generalize some variables, or contains some proofs of equality between the thing matched on and a given constructor in a branch. Extending any of this would make an unreadable mess for the users and is too detached from what a user would do had they just written this as a normal definition. Consider the case of `is_var_zero` being extended to `Term`. For both the `app` and `lam` case, the result is the same (*i.e.,* `false`). Producing this extension would require both extending the implementation of `match_1` (by adding a branch for `(t : Term) → motive (lam t)` and `(f t : Term) → motive (app f t)`), and adding the right arguments for the right-hand-side of said new branches (*i.e.,* `fun t => false` and `fun f t => false`). This means, in this specific example, that a user would need to fill in 4 holes, half of them having a non-trivial shape which mentions an arbitrary `motive` that did not appear in the concrete syntax of the original definition.

A second option for extending matches would be to note that extending a match amounts to adding new branches to it such that it covers the relevant new constructors in the extended inductive type. As such, we may ask users to write down the relevant branches, in a syntax similar to how normal matches are written. This option turns out to be the most ergonomic, although it makes the implementation work measurably harder than the former. In order to accommodate for this

[2]Note that a `Unit` argument appears in the `var 0` branch. This happens because Lean's compiled code is call-by-value, which justifies adding thunks to avoid computing such branches eagerly

feature, when extension mapping the term of a given declaration, any application whose head is a matcher gets translated into a metavariable hole, saving the original shape of the expression along the way. When trying to solve that metavariable, the original shape of the matcher is reconstructed, similarly to how the pretty-printer handles this expression. We then elaborate the new match branches provided by the user, and re-elaborate this into a new match-expression. Note that the implementation does **not** do any anti-quotations (*i.e.,* it does not construct concrete syntax from the original abstract syntax), and instead manipulates both the abstract syntax of the old matcher and the concrete syntax of the new branches in parallel, using Lean's existing internals for elaborating matchers. The end-result is a legible syntax for extending past declarations.

Take the example of `is_var_zero` getting extended to `Term`:

```
def Term.is_var_zero extends is_var_zero where
  extend with
  | app _ _ => false
  | lam _   => false
```

First, the extension system of **Gemel** detects that `is_var_zero.match_1 (fun _ => Bool) x (fun _ => true) (fun_ => false)` is a matcher application and stores the information about its shape (*i.e.,* what the motive, discriminants and existing branches are). All of this information gets extension mapped to `Term` rather than the original `Var` (e.g. `Var.var` becomes `Term.var` in the left-hand side of each branch). Then, the branches provided by the user (*i.e.,* `| app _ => false` and `| lam _ _ => false`) get elaborated as new branches to be added to the previous ones. This bundle of data is then passed out to the existing API Lean uses to elaborate normal pattern-matches, and produces a new match application that the old one then gets translated to.

3.2.2 Extending recursive functions. Though recursion is present at the level of the concrete syntax, it gets elaborated away into structural and well-founded recursion at the abstract syntax level, which means that functions internally get translated into things that are different in shape to a user's original definition. Because of this, directly extension mapping the body of a recursive definition is not great, since it exposes internal encodings to the user, and asks one to manipulate those directly to extend the definition. Thankfully, Lean usually provides auxiliary lemmas that exhibit the original shape of the function that was defined. One of them, `foo.eq_def`, is a theorem of the form:

$$(a_1 : A_1) \to \ldots \to (a_n : A_n) \to \texttt{foo}\ a_1 \ldots a_n = \texttt{<foo's definition>}$$

As such, rather than using the original body of a definition, we can instead extension map the right-hand side of a function's `eq_def` whenever available, and then reuse the existing elaboration APIs Lean provides to translate such syntactically recursive functions to structurally recursive or well-founded functions. This in effect means a user can adapt the termination measure of an extended definition relative to the original, a feature no other extension system provides to our knowledge. Said feature is of particular importance when extending a non-recursive type to a recursive one, as is the case with the extension from `Var.repr` to `Term.repr`.

In particular, this allows us to turn an originally non-recursive function into a recursive one if needed, as is the case with `Term.repr` extending `Var.repr`, or even change the termination measure that was originally used to adapt it to the new extended function.

3.2.3 Putting it all together. In practice, being able to change the proof of termination of an extended function compared to the original provides us with much more extensibility, and is a feature we have not seen in any other project of the sort.

Furthermore, having matches be handled specially provides a clear distinction between the holes generated for matches and those generated by explicit uses of recursors, which usually only appear

in proof terms generated by tactic calls such as `induction` and `cases`. branches. The complete syntax for `mod def` ends up as the following:

```
mod def <new function name> extends <old function name> where
  <match extension>
by
  <tactics>
<termination-information>
```

This in essence means our syntax for extended definitions offers a clear separation of concerns between proof holes, solved by tactics, and non-proof holes, solved by completing matches. The syntax asks for matchers to be completed first, before asking proof holes to be solved in a `by` block. The `by` block mainly only appears when extending theorems rather than definitions. Consider the following theorem:

```
theorem is_var_zero_eq (t : Var) (h : is_var_zero t) : t = var 0 := by
  cases t
  case var n =>
    cases n
    case zero => rfl
    case succ _ => nomatch h
```

Extending it for `Term` will add two new proof holes originating from the `cases t`. These are given by the user in the `by` block as follows:

```
mod def Term.is_var_zero_eq extends is_var_zero_eq where by
  case lam _   => nomatch h
  case app _ _ => nomatch h
```

Termination information such as `termination_by` or `decreasing_by` can be additionally provided if needed to help with producing a well-founded recursive function. For example, the previous `Term.repr` function could have had an explicitly given annotation such as `termination_by structural t => t`, indicating that the function is structurally recursive over it's argument `t : Term`. In practice, and as is the case for the regular Lean definition, most functions' termination information is trivial enough to be inferred by the Lean, and thus does not require additional user-input.

A current limitation of **Gemel** is that the tool does not currently support extending mutually recursive functions. Similarly to the fact the tool cannot extend mutual and nested inductive types, this feature is only a matter of additional engineering and left for future work.

4 MERGING EXTENSIONS

The current setup allows one to extend previous definitions iteratively, though one may argue these extensions are not strictly "extensible". In particular, one can currently only construct a declaration as an extension of a single other declaration rather than multiple. This, in particular, means one isn't able to compose different extensions. Take the example of the Barendregt lambda-cube [3]:

The various vertices of the cube are extensions of the initial vertex corresponding to STLC, each adding new features to the original type theory. However, an important feature of this cube is that all extensions of λ can be written as mergings of its 3 adjacent vertices, namely λP, $\lambda 2$ and $\lambda \underline{\omega}$ (e.g. $\lambda P2$ corresponds to the merging of $\lambda 2$ and λP). If one wanted to formalise each vertex of the cube before **Gemel**, they would need to write down 8 different formalisations. With our current framework, they need only write down 1 formalisation and 7 extensions of that base formalisation. If our system was truly extensible, however, one would only need to write down the base formalization and the 3 adjacent extensions, the rest being simply generated by merging the adjacent extensions. To make this possible, we extend **Gemel**'s handling of inductive types and definitions extensions and

describe a way to make them composable, thus achieving true extensibility. In practice, from a user perspective, all that gets added is the ability to define a `mod inductive` or `mod def` that `extends` more than one single declaration.

4.1 Merging inductive types

Extending `mod inductive` to be able to extend more than one inductive type is a fairly trivial task. Rather than only taking the constructors of one inductive, users may take constructors from multiple types. The mapping context will then map every old type to the new one, every old constructor from each old type to the relevant new ones. Consider the example of `BoolTerm` extending `Term` in Section 3.1.2, one may also extend `Term` to handle natural numbers, and merge the two extensions as follows:

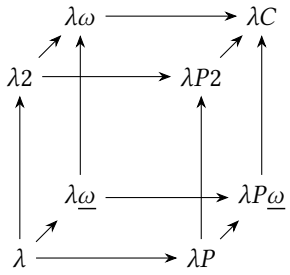


Fig. 5. Barendregt lambda-cube

```
mod inductive NatTerm extends Term where
  | zero : NatTerm
  | succ : NatTerm → NatTerm
  --   match n with | 0 => P0 | n+1 => Pn n
  | natmatch : NatTerm → NatTerm → NatTerm → NatTerm
mod inductive BoolNatTerm extends BoolTerm, NatTerm
```

The merging of constructors is nominal and type-driven: when merging `BoolTerm` and `NatTerm` together, **Gemel** remarks that the `var`, `app` and `lam` constructors are in common between the two, confirming their types are compatible, and does not duplicate them in the merging. As such, the generated inductive `BoolNatTerm` will contain all of the basic `Term` constructors, as well as the additional ones provided by `BoolTerm` and `NatTerm`, and maps the old inductives' constructions to the appropriate constructions of the new one.

4.2 Merging definitions

Consider the previous examples of defining `is_var_zero` and `is_var_zero_eq` to `BoolNatTerm`, consider the two declarations have already been extended to `BoolTerm` and `NatTerm`:

```
mod def BoolTerm.is_var_zero extends Term.is_var_zero where
  extend with
  | true | false | ite _ _ _ => false
mod def NatTerm.is_var_zero extends Term.is_var_zero where
  extend with
  | zero | succ _ | natmatch _ _ _ => false
```

We can extension map both declarations into functions talking about `BoolNatTerm`. Both of them will be incomplete in that the pattern-match will be incomplete in both cases (e.g. the cases for `zero` will be missing from the extension map of `BoolTerm.is_var_zero`). However, both declarations will have the same shape modulo having different holes in different places. We makes use of that and construct an algorithm that syntactically merges two expressions together, and throws an error if their shape diverges. In practice, in the case of `is_var_zero`, this means the respective matches get merged correctly, meaning there is no need for the user to add any more information in order to merge the two declarations:

```
-- No `extend with` block needed to complete the match
mod def BoolNatTerm.is_var_zero
  extends BoolTerm.is_var_zero, NatTerm.is_var_zero
```

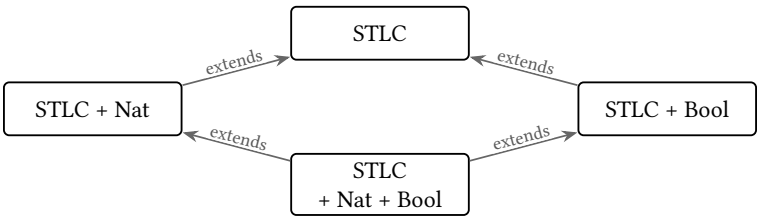
Similarly, the proof for `is_var_zero_eq` need not any more information after merging both `BoolTerm.is_var_zero_eq` and `NatTerm.is_var_zero_eq`:

```
mod def BoolTerm.is_var_zero_eq extends Term.is_var_zero_eq where by
  ... -- proof omitted
mod def NatTerm.is_var_zero_eq extends Term.is_var_zero_eq where by
  ... -- proof omitted
-- no `by` block needed
mod def BoolNatTerm.is_var_zero_eq
  extends BoolTerm.is_var_zero_eq, NatTerm.is_var_zero _eq
```

5 CASE STUDIES

As a demonstration of the advantages provided by **Gemel**, we present two separate case-studies. The former shows **Gemel**'s capacity to do *after the fact* extensions by taking an independent mechanised proof of strong normalization for the Simply Typed Lambda-Calculus (STLC) and extending that result to extensions of the calculus. The latter case-study shows the scale at which **Gemel** can function by replicating parts of the central case-study presented by Jamner et al. [15]. We then compare the code written in these case-study to equivalent code written without the use of extensibility tools, and measure the performance cost, as well as number of lines of code needed, for the two solutions.

5.1 Strong normalisation of STLC



In order to iterate on this implementation and ensure its usability, we studied the case of taking an existing implementation of STLC [20] which proves the strong normalization property of the system, and extended it with additional constructors, allowing us to extensibly prove the strong normalization property of the extended system. In particular, the original normalization was **not** built with extensibility in mind, and was still extendable *after the fact*. The original formalisation follows the usual proof of normalisation using a logical relation.

```
inductive Base.Ty : Type where
  | arr : Ty → Ty → Ty
```

```
inductive Base.Term where
  | var : Nat → Term
  | app : Term → Term → Term
  | lam : Term → Term
```

```
inductive Base.Typing : List Ty → Term → Ty → Type where
  | tyvar : Γ[n]? = some A → Typing Γ (var n) A
  | tyapp : Typing Γ f (arr A B) → Typing Γ t A → Typing Γ (app f t) B
  | tylam : Typing (A :: Γ) t B → Typing Γ (lam t) B
... --other definitions/theorems
```

We extend this base definition to add natural numbers.

```
modular NatTerm
mod inductive Ty
  extends Base.Ty where
  | nat

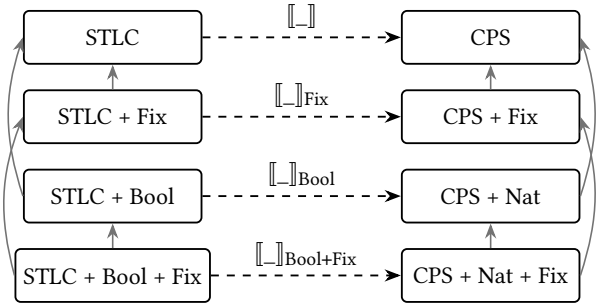
namespace NatTerm
mod inductive Term
  extends Base.Term where
  | zero   : Term
  | succ   : Term → Term
  | natRec : Term → Term → Term → Term

mod inductive Typing extends Base.Typing where
  | zero   : Typing Γ zero nat
  | succ   : Typing Γ n nat → Typing Γ (succ n) .nat
  | natRec : Typing Γ n nat → Typing Γ P0 A
  → Typing Γ PS (arr nat (arr A A)) → Typing Γ (natRec n P0 PS) A
... --other definitions/theorems
end NatTerm
```

The total formalisation contains 12 inductive types as well as 92 definitions/theorems. Almost all of these were extended with no issue. The original formalisation contains a limitation to extensions in that the logical relation `LR` it defines to prove strong normalisation is too weak to prove the property on a system extended with natural numbers. We considered manually adding an extension map from `LR` to a variant strong enough for our purpose. However, the theorems in the original formalisation that makes use of that LR rely heavily on its definitional equalities, which do not hold in a stronger formulation of the relation, meaning such a mapping leads to ill-typed terms when translating said theorems. As such, `LR` and the fundamental lemma had to be written as normal lean `def`s rather than as extensions. A solution to circumventing such issues, based on the idea of removing the reliance on some definitional equalities on a given term, is discussed in the future work (§7).

To showcase the capacity to merge separate extensions, we produce another extension of STLC containing primitives for booleans, namely a boolean constructor in `Ty`, and term constructors for `true`, `false` and `if-then-else`. We then successfully construct a formalisation of STLC with both natural numbers and booleans by merging the two previous formalisations, thus proving the strong normalization of "STLC + Bool + Nat" fully extensibly.

5.2 STLC to CPS translation



To showcase the capabilities of our framework, and provide a point of comparison with another extensibility system, we reproduce the central case study shown in Pyrosome's [15] paper. This case study introduces two type system: STLC, and CPS-style type-system, as well as a translation from the former to the latter which proves $\beta\eta$-equivalence is conserved through that translation pass. The case-study then extends both STLC and CPS with new constructions (resp. booleans and a fixpoint operator for STLC, natural numbers and a fixpoint operator for CPS), and extends

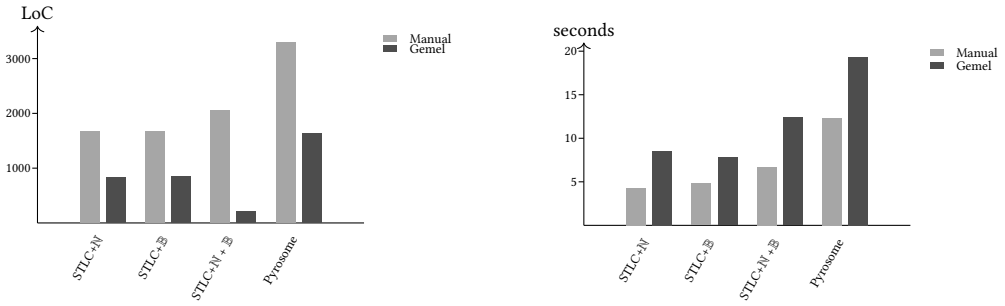


Fig. 6. Lines of code and compilation times for the available case studies. AA ▶*TODO update with new numbers for STLC*◀

the translation pass accordingly. One feature of Pyrosome's case study is that their framework allows for limited forms of horizontal extensions, which our framework doesn't. In particular, STLC terms are separated between values (*i.e.,* terms whose head can't be anymore reduced) and expressions (who can reduce). This STLC formalisation differs from the first case study in that typing is enforced intrinsically here, meaning the type of STLC terms are indexed by a context and a type (*i.e.,* `Term` has type `List Ty → Ty → Type`), whereas well-typedness is enforced extrinsically through a predicate in the former (*i.e.,* `Term` has type `Type` and a predicate `Typing: List Ty → Ty → Term → Prop` is defined). In order to extensibly reuse as much API as possible between STLC, CPS and their respective extensions (such as substitution and renaming operations), both type systems are defined as extensions of a `Base` bundle, for which terms are just variables, and which defines a common API for both extensions to build off of. The shape of both systems differ in that both expressions and values are indexed by a type in STLC, but only values are in CPS. The type of terms is instead indexed by a context, a tag, and a dependent family `Tag.Data` over that tag which explicits by what data a term of a given tag is indexed by. As such, both STLC and CPS can extend the `Tag` type, and `Tag.Data` for their respective needs:

5.3 Benchmarks

To measure the effectiveness of the **Gemel**, we compare code written using the tool with equivalent code written by copy-pasting the original code and extending it. We compare both the elaboration time as well as number of lines both use. The results (**??**) across both case-studies shows lines of code are reduced by a factor of ~ 2 when extending existing formalisations, and by a factor of ~ 9 when merging separate extensions into one. Regarding performance, we currently observe that declarations generated using **Gemel** are slower to elaborate by a factor of ~ 2. However, the performance is currently limited by a bug in Lean we uncovered. This bug constrains the tool to type-check certain parts of the translated expressions many times to make unification work correctly, which leads to a quadratic blowup in performance [citation absent for anonymity]. Once this issue is fixed, and **Gemel** is adapted to that fix, we expect the tool to have similar performance to regular definitions. In fact, **Gemel** is sometimes already faster to elaborate than copy-pasted code. Indeed, since **Gemel** directly translates fully-annotated terms, it does not need to re-compute expensive elaborations features like implicit arguments, typeclass resolution or tactic executions, making it faster in certain cases.

6 RELATED WORK

We compare our approach with the existing solutions to the extensibility problem in ITPs.

"Meta-Theory à la Carte" [9], "Pyrosome" [15] "Generic Datatypes à la Carte"[16] and "Modular Type-Safety Proofs in Agda"[25] base their extensibility on internal encodings of types. In all cases,

the constructions are inefficient, incapable of extending previously user-defined inductive types (*i.e.,* expecting users to rely on the aforementioned encodings from the ground up), and exposes the underlying internals of the encodings to the user. Having to deal with encodings of types rather than types adds a heavy burden in particular for new users interested in program verification.

On the other hand, Rocq à la Carte [13] relies on the metaprogramming capabilities offered by the MetaRocq Project [27] to allow users to construct new inductive types and functions by merging other inductive types, and/or adding new constructors. New functions on a "merged" datatype can then be constructed by merging past functions. The metaprogram then uses the given piece of information to reconstruct a new inductive type, and new functions, based on the information given by the user. While great for extending constructions "vertically" (*i.e.,* by adding constructors to a type), this system does not allow for horizontal extensions (*i.e.,* extending the type signature of inductive types and their constructors). Furthermore, this approach has been hindered in the past by the lack of good metaprogramming frameworks in ITPs.

Rocqet comes the closest in philosophy to our implementation, relying heavily on metaprogramming and seemingly manipulating Rocq's AST directly. It however also doesn't allow for *after the fact* extensions, and their interface for extending past definitions is more restricted. Indeed, the only allows for constructing recursive functions with a single pattern-match with a single discriminant at the top of the function, where **Gemel** handles arbitrary pattern-matching. Furthermore, Rocqet is suited for defining and extending functions and proofs defined using structural induction on the datatype getting extended, **Gemel** on the other hand can handle structural and well-founded induction over arbitrary terms. Lastly, Rocqet requires users to explicit the motive function of each induction in their system. All of this comes at a cost for the user-experience.

None of these systems, independently of whether they use encodings or the metaprogramming, handle all of the type-system of ITPs they are implemented for (*e.g.,* none handle co-inductive types), or allow users to extend past formalizations *after the fact*. Instead, formalizations have to be built from the ground up with the expectation that they will be extended with a specific framework in mind, making them less useful for real-world uses.

The only system we have found which attempts to provide *after the fact* extensions is Shahin [26]. Their tool has a similar scope to **Gemel**'s, in that it seemingly allows for combining inductive types into one, as well as merging functions together. However, similarly to Rocqet, it is limited by the fact it can only manipulate types, functions and proofs of a specific shape. In particular, it cannot handle indexed inductive types, or functions which doesn't consist of a sequence of nested lambdas, followed optionally by a single pattern-match. What this means is that, while the tool can in principle extend regular types and functions, it can only do so if those use the limited subset this tool is capable of handling, a limitation not shared by **Gemel** which supports all regular Lean constructions.

7 FUTURE WORK

Gemel shows how directly manipulating the core AST of an ITP without relying on encodings can work for producing extensible formalisations. Some further design decisions remain to be experimented with, and may open up the door for further improvements to the tool.

As mentioned in §3.1.2 and §3.2.3, **Gemel** currently does not allow for extending mutual and nested inductive types, as well as mutually recursive functions. The APIs Lean provides to handle mutual and nested types is the same as the ones for regular types, and similarly for mutual and regular declarations. As such, adding these features is only a matter of further engineering work.

There are morally two ways to extend an inductive type, referred to as vertical and horizontal extensions [10]. The former consists of adding new constructors to an inductive type, the latter of adding new fields to a type's constructor, or to the type itself. Most extensibility systems only allow

for doing vertical extensions, as it is both the easiest one to justify and implement, and arguably the most useful one. Currently, **Gemel** only manages vertical extensions, though our second case-study (Section 5.2) shows how some horizontal extensionality can be recovered in the system through careful use of dependent types. The space of handling horizontal extensions has not been explored much in existing literature, the only system which, to our understanding, manages some form of horizontal extensions is Pyrosome. However, the scope of their extensions is much more limited and cares only about translation passes for compilers.

The central way in which extensions are added to the environment is by mapping constants to new (potentially incomplete) terms. For a translation using a constant found in the mapping context to work well, it is necessary that the reduction rules involving past constants also hold when mapping the constant to its associated term. This restriction is implicitly present when translating recursors and matches. However, as seen in the first case-study, it may sometimes be necessary for a formalisation to extend something (e.g. the logical relation in the case study) with something possibly completely new, that may not hold the same definitional equalities. In order to do this, **Gemel** could, in the future, have some mechanism for transforming an expression which relies on some particular reduction rules/definitional equalities into one that does not. This idea of removing such rules is not new, and appears in works producing translations from Extensional Type Theory to Intensional Type Theory [34], as well as works on removing definitional equalities from Lean using a similar translation [31]. The latter work could in particular possibly be adapted to our needs since it is already implemented in Lean.

One challenge with extending independent formalisations is that the shape of an original proof might not be fitting to prove an extended version of it. A regular example of this issue is proofs done by induction: it appeared a few times when testing **Gemel** that a proof that was done by induction needed, for the extended proof to work, to generalize some variables when doing said induction, so as to strengthen the induction hypotheses. Similarly, it could happen that part of a proof originally just needed a case split where the new proof would require the full power of induction. However, the existing interface of metavariable holes morally only allows us to add new content at the tips of our proof trees, not modify the proof-tree in its generality. To extend this framework, we could imagine allowing for some *after the fact* transformations, such as generalisations or turning a case split into an induction, to the rest of the proof tree, allowing for a more comfortable experience of doing extensions of independent formalisations.

8 CONCLUSION

We contribute a ready-to-use tool to easily extend formalizations. That tool is fairly complete in its original objective, as shown by our case-studies, though it exposes some limitations and a potential for further improvements. Improvements include the capacity to handle horizontal extensions, *i.e.,* the ability to add new fields to an existing constructor of a datatype, rather than just the ability to add new constructors entirely. A good next step would be to apply tool to real-world cases such as Lean's computer science library CSLib [4], where e.g. various type-systems formalizations could benefit from the tool.

AI use disclosure. The design and implementation of **Gemel**, as well as the two central case-studies (STLC and Pyrosome) were human written. LLMs tools such as Claude Code were used to help understand bugs and performance issues in **Gemel** during the end of its development, to construct additional case-studies not mentioned in the paper, as well as to produce the shell script used to produce the benchmarks provided in 5.3.

REFERENCES

[1] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. [n. d.]. Containers: Constructing strictly positive types. 342, 1 ([n. d.]), 3–27. https://doi.org/10.1016/J.TCS.2005.06.002

[2] Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. [n. d.]. Martin-Löf à la Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs* (London, UK, 2024) *(CPP 2024)*. Association for Computing Machinery, New York, NY, USA, 230–245. https://doi.org/10.1145/3636501.3636951

[3] Henk Barendregt. [n. d.]. An Introduction to Generalized Type Systems. 1 ([n. d.]), 125–154. https://doi.org/10.1017/S0956796800020025

[4] Clark Barrett, Swarat Chaudhuri, Fabrizio Montesi, Jim Grundy, Pushmeet Kohli, Leonardo de Moura, Alexandre Rademaker, and Sorrachai Yingchareonthawornchai. 2026. CSLib: The Lean Computer Science Library. *arXiv e-prints* (Feb. 2026). https://doi.org/10.48550/arXiv.2602.04846 arXiv:2602.04846 [cs.LO]

[5] John Cartmell. [n. d.]. Generalised algebraic theories and contextual categories. 32 ([n. d.]), 209–243. https://doi.org/10.1016/0168-0072(86)90053-9

[6] Alonzo Church. 1933. A Set of Postulates For the Foundation of Logic. *Annals of Mathematics* 34, 4 (1933), 839–864. http://www.jstor.org/stable/1968702

[7] Jesper Cockx and Andreas Abel. [n. d.]. Elaborating Dependent (Co)Pattern Matching. 2, ICFP, Article 75 ([n. d.]), 30 pages. https://doi.org/10.1145/3236770

[8] Thierry Coquand and Christine Paulin. [n. d.]. *Inductively defined types*. Springer Berlin Heidelberg, 50–66. https://doi.org/10.1007/3-540-52335-9_47

[9] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013. Meta-theory à la carte. *SIGPLAN Not.* 48, 1 (Jan. 2013), 207–218. https://doi.org/10.1145/2480359.2429094

[10] Dominic Duggan and Constantinos Sourelis. 1996. Mixin modules. *SIGPLAN Not.* 31, 6 (June 1996), 262–273. https://doi.org/10.1145/232629.232654

[11] Oghenevwogaga Ebresafe, Ian Zhao, Ende Jin, Arthur Bright, Charles Jian, and Yizhou Zhang. 2025. Certified Compilers à la Carte. *Proc. ACM Program. Lang.* 9, PLDI, Article 162 (June 2025), 24 pages. https://doi.org/10.1145/3729261

[12] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2017. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. *Proc. ACM Program. Lang.* 1, ICFP, Article 13 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110257

[13] Yannick Forster and Kathrin Stark. [n. d.]. 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, New Orleans, LA, USA, January 20-21, 2020* (2020), Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 186–200. https://doi.org/10.1145/3372885.3373817

[14] Healfdene Goguen, Conor McBride, and James McKinna. 2006. Eliminating Dependent Pattern Matching. In *Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday (Lecture Notes in Computer Science, Vol. 4060)*, Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer (Eds.). Springer, 521–540. https://doi.org/10.1007/11780274_27

[15] Dustin Jamner, Gabriel Kammer, Ritam Nag, and Adam Chlipala. 2025. Pyrosome: Verified Compilation for Modular Metatheory. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 274 (Oct. 2025), 28 pages. https://doi.org/10.1145/3763052

[16] Steven Keuchel and Tom Schrijvers. 2013. Generic datatypes à la carte. In *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming* (Boston, Massachusetts, USA) *(WGP '13)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/2502488.2502491

[17] Théo Laurent, Meven Lennon-Bertrand, and Kenji Maillard. [n. d.]. Definitional Functoriality for Dependent (Sub)Types. In *33rd European Symposium on Programming, ESOP 2024* (2024) *(Lecture Notes in Computer Science, Vol. 14576)*, Stephanie Weirich (Ed.). Springer, 302–331. https://doi.org/10.1007/978-3-031-57262-3_13

[18] Meven Lennon-Bertrand. [n. d.]. What Does It Take to Certify a Conversion Checker?. In *10th International Conference on Formal Structures for Computation and Deduction (FSCD 2025)* (2025) *(Leibniz International Proceedings in Informatics (LIPIcs), Vol. 337)*, Maribel Fernández (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 27:1–27:23. https://doi.org/10.4230/LIPIcs.FSCD.2025.27

[19] Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. [n. d.]. Gradualizing the Calculus of Inductive Constructions. 44, 2, Article 7 ([n. d.]), 82 pages. https://doi.org/10.1145/3495528

[20] Andrew Marmaduke. 2026. lean-stlc. Github repository. https://github.com/amarmaduke/lean-stlc

[21] Per Martin-Löf. [n. d.]. An intuitionistic theory of types.

[22] Pierre-Marie Pédrot. [n. d.]. "Upon This Quote I Will Build My Church Thesis". In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science* (Tallinn, Estonia, 2024) *(LICS '24)*. Association for Computing Machinery, New York, NY, USA, Article 65, 12 pages. https://doi.org/10.1145/3661814.3662070

[23] Pierre-Marie Pédrot and Nicolas Tabareau. [n. d.]. The Fire Triangle: How to Mix Substitution, Dependent Elimination, and Effects. 4, POPL ([n. d.]). https://doi.org/10.1145/3371126

[24] Loïc Pujet and Nicolas Tabareau. [n. d.]. Observational Equality: Now for Good. 6, POPL, Article 32 ([n. d.]), 27 pages. https://doi.org/10.1145/3498693

[25] Christopher Schwaab and Jeremy G. Siek. 2013. Modular type-safety proofs in Agda. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification* (Rome, Italy) *(PLPV '13)*. Association for Computing Machinery, New York, NY, USA, 3–12. https://doi.org/10.1145/2428116.2428120

[26] Ramy Shahin. 2026. Modular Composition of Inductive Types Using Lean Meta-programming. *arXiv e-prints*, Article arXiv:2609.25427 (Sept. 2026), arXiv:2609.25427 pages. https://doi.org/10.48550/arXiv.2609.25427 arXiv:2609.25427 [cs.PL]

[27] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *Journal of Automated Reasoning* (2020). https://doi.org/10.1007/s10817-019-09540-0

[28] Matthieu Sozeau and Cyprien Mangin. 2019. Equations Reloaded: High-Level Dependently-Typed Functional Programming and Proving in Coq. *Proc. ACM Program. Lang.* 3, ICFP, Article 86 (July 2019), 29 pages. https://doi.org/10.1145/3341690

[29] Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (July 2008), 423–436. https://doi.org/10.1017/S0956796808006758

[30] Tarmo Uustalu and Varmo Vene. 1999. Mendler-style inductive types, categorically. *Nordic J. of Computing* 6, 3 (Sept. 1999), 343–361.

[31] Rishikesh Vaishnav. 2025. Lean4Less: Eliminating Definitional Equalities from Lean via an Extensional-to-Intensional Translation. In *Theoretical Aspects of Computing – ICTAC 2025 22nd International Colloquium*, Vol. Lecture Notes in Computer Science. Marrakech, Morocco. https://inria.hal.science/hal-05310102

[32] Philip Wadler. 1998. The expression problem. Discussion on the Java Genericity mailing list. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

[33] Freek Wiedijk. 2000. The De Bruijn Factor. (08 2000).

[34] Théo Winterhalter. 2020. *Formalisation and Meta-Theory of Type Theory*. Theses. Université de Nantes. https://doi.org/10.70675/f3c96514z4373z4f99z8b3fzb1b41307d558