A computer-checked library of category theory

Formally verifying currying via the product-exponential adjunction

Gabriel-Ciprian Stanciu[1]

Supervisors: Benedikt Ahrens[1]**, Lucas Escot**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Gabriel-Ciprian Stanciu
Final project course: CSE3000 Research Project
Thesis committee: Benedikt Ahrens, Lucas Escot, Kaitai Liang

Abstract. Existing implementations of category theory for proof assistants aim to be as generic as possible in order to be reusable and extensible, often at the expense of readability and clarity. We present a (partial) formalisation of category theory in the proof assistant Lean limited in purpose to explaining currying, intended to be faithful to the language and definitions used in mathematics literature. We also present some design features of our library and contrast the extent and educational merit with other implementations.

1 Introduction

Category theory is a branch of mathematics that, essentially, provides a formal instrument for defining *categories*, (mathematical) structures formed of objects of the same type and relations between them, *functors*, mappings between two categories that preserve their structure, and other notions that emphasize studying the relations between mathematical objects and not their specific details. As category theory focuses on studying structure and abstracting away objects' specific details that are tied to their area of mathematics, it helps reveal connections between different mathematical fields and potentially find or lift results from one field to apply in multiple others.

Although it is pure mathematics, this concern with structure has led to category theory being used to reason about notions in other sciences such as physics, where similarity has been found between categories that model the theory of general relativity and quantum theory, or logic, where the properties of categories modelling different systems of logic provide insight into the rules it follows [1]. More relevant to computer science, programming languages have lifted some of its concepts to better structure programs. Notably, Haskell types and functions form a structure akin to a category and the language has borrowed categorical concepts such as functors [2] or monads [3].

Dependent type theory is a language that can be used for encoding mathematics, suitable for computer proof assistants as it allows the program to spot mistakes — statements that are syntactically valid but are incorrect. Computer proof assistants can be used to verify the correctness of formal mathematical proofs by checking every step in the proof and using other already computer-checked theorems, down to the fundamental base axioms. In this context and taking into account the abstraction level of category theory, computer proof assistants are useful — they provide complete confidence in the correctness of a result which could otherwise contain mistakes overlooked by reviewers.

A bachelor's degree education in computer science tends not to go into category theory, while it almost always contains courses about functional programming or programming languages in general. For this reason, understanding how concepts such as currying or monads work, or the motivation behind them, can be difficult.

This paper aims to provide a readable and understandable description of currying, a notion of isomorphism between arrows from a binary product and arrows to an exponential that also appears in computer science, alongside its prerequisite concepts, supported by a computer-checked implementation of parts of category theory. It also provides details behind the decisions taken during its implementation and differences to other category theory libraries.

The inclusion of a formally verified library has two purposes: to serve as a learning experience for the author and be targeted toward beginner users who may find existing implementations for working categoricians daunting and too abstract. It

also prefers more explicit proofs using less automation for educational purposes, while it has unambiguity as an advantage over a classic mathematical explanation.

Section 2 goes over the presentation of the content included in the paper and provides a short overview of the proof assistant chosen and the type theory powering it, along with a description of the library structure. Section 3 contains short explanations of the core part of the library — definitions and theorems developed by the whole project group for the basic notions in category theory. An explanation of adjunction and an example useful to computer science, currying, can be found in section 4. Section 5 reviews design decisions taken during the development of the project and includes a comparison with other related work. Finally, section 6 finishes the paper and proposes a future direction for the work.

2 Method

Explaining currying requires some familiarity with other concepts from category theory. In computer science, currying transforms a function taking two variables into one that takes the first variable as an argument and returns a new function. The result should take in the second variable as its only argument and return the same result as the original function. In category theory however, currying is the natural isomorphism between morphisms from *binary products* and morphisms to *exponential objects*.

In order to introduce it, we will start from the basic notions of *category* and *functor* and incrementally build upon this knowledge. We will be stating the next concepts and presenting an abridged version of the formal definition from the library, alongside examples if necessary.

2.1 Formal verification

We chose to implement our library in Lean 3.x[1], a proof assistant with a type theory based on the *Calculus of Inductive Constructions* (CIC) [4], capable of representing proofs in higher-order predicate logic and of representing data types efficiently using inductive definitions.

The reasons for this choice of proof assistant were Lean's accessible ecosystem, existing documentation and category theory library [5]–[7], and its use of CIC which is at the base of many existing formalisations of category theory [8, Table 1], showing its suitability.

As Lean adheres to the Curry-Howard correspondence, a proposition is represented by a type with objects of the same type constituting proofs of the proposition. Propositions such as $A \wedge B$ are typed as pairs `A×B` of proofs of A and B, implications $A \Rightarrow B$ are of type `A→B` that convert proofs of A to proofs of B, while $\forall x : A, B$ also can be written as dependent types `Π x: A, B` that map each x in A to a proof.

Owing to the type theory used, a program written in Lean that type-checks can be considered to be valid; that is, if all objects are successfully checked to be of the proper type then the proofs they represent are also valid.

Lean has a useful tool for creating proofs in the form of *tactics*, programs that change the goal that needs to be proved based on hypotheses derived from the initial proposition and previously proved theorems. Although the code snippets in the next sections are meant to be understandable without much prior knowledge, table 1 contains short descriptions of the more common tactics.

[1] `https://leanprover.github.io/`, `https://leanprover-community.github.io/`

`intros`	adds hypotheses (terms) $x : P_1$ $y : P_2$ ... if the goal is of the form $P_1 \to P_2 \to g$ or $\forall (x : P_1)(y : P_2), g$ and changes the goal to g
`refl`	completes the proof if the goal is an equality with both sides equal
`split`	for a goal of the form $p_1 \wedge p_2$, splits it into two separate goals to be proved in sequence p_1, p_2
`simp`	tries to rewrite the goal using built-in theorems and the current hypotheses
`rw h...`	rewrites the goal using the theorems or hypotheses specified
`apply h`	for a theorem or hypothesis of the form $H : p \to g$, changes the goal g to p
`have h : <type>`	introduces a new hypothesis h of the specified type and switches the current goal to proving the new hypothesis
`let id := <expr>`	introduces a new term *id* defined by `<expr>`
`unfold id`	replaces *id* with its definition
`exact h`	completes the proof if the goal is of the same type as h

Table 1: Overview of common Lean tactics.

2.2 Project structure, contributors

The code of the project can be studied at `https://github.com/sgciprian/ct`.

Folder `doc` contains files explaining how some concepts were formalised in our library, showing how to type the Unicode notation used in the project, and a bibliography with sources for category theory and Lean.

The actual source code can be found in folder `src` containing concepts from category theory implemented in the files or folders with the same name. The folder `instances/` contains examples of categories. Some concepts that have multiple definitions or examples implemented contain both a file (eg. `functors.lean`) and a folder (eg. `functors/`). The file serves as a meta-file, including the entire content of the folder.

The implementation part of the project was done in two stages, within the research project group. The first stage focused on building up a base of core definitions for the library, while the second focused solely on defining adjunctions and providing examples.

The second part was individual work (albeit with help and support provided by the group colleagues at times) and roughly includes the files contained in the folders `adjunctions/` and `universal_properties/`. The rest of the source files were either contributed by the other members for either of the two stages, or by the author during the first, fully collaborative stage.

The paper itself is the own work of the author.

3 Core definitions

This section introduces the prerequisite notions necessary for introducing *adjunctions*: categories, functors, and natural transformations, also showcasing the use of the library. The presentation of the definitions is largely based on Leinster's introduction to category theory [9], with parts from Mac Lane [10] and Pierce [11].

3.1 Categories

Categories are structures consisting of *objects*, *morphisms* or equivalently *arrows* that link two objects together, a *composition* operation that creates a new arrow whenever we have two arrows that end and start, respectively, on the same object, and an *identity* morphism unique for each object that points to itself. Objects are not important beyond their existence; they are specified only by the arrows that connect them to other objects, so an object could stand for a number, a set, or any other notion.

There are some rules that any structure with objects and morphism must have in order to be a category: the composition operator should be associative, and the identity combined with any morphism should construct the original morphism.

```
structure category :=
  --attributes
  (C₀      : Sort u)
  (hom     : Π (X Y : C₀), Sort v)
  (id      : Π (X : C₀),   hom X X)
  (compose : Π {X Y Z : C₀} (g : hom Y Z) (f : hom X Y),
           hom X Z)
  --axioms
  (left_id  : ∀ {X Y : C₀} (f : hom X Y),
            compose f (id X) = f)
  (right_id : ∀ {X Y : C₀} (f : hom X Y),
            compose (id Y) f = f)
  (assoc    : ∀ {X Y Z W : C₀}
              (f : hom X Y) (g : hom Y Z) (h : hom Z W),
            compose h (compose g f) = compose (compose h g) f)
--src/category.lean
```

With this definition we can define various instances of categories, for example the snippet below defines the object type, morphism type, the identity and composition and proves the three category laws for the product category — a category where each object is a pair of objects from the two "parent" categories and each morphism is also a pair of morphisms from the two categories.

```
def Product (C D : category) : category :=
{
  C₀ := C × D,
  hom := λ p p', (C.hom p.fst p'.fst) × (D.hom p.snd p'.snd),
  -- 1(c, d) = (1c, 1d), where 1 is the identity morphism
  id  := λ p, (1C p.fst, 1D p.snd),
  -- Composition composes each morphism component in its category.
  compose := λ {p q r} (g : (C.hom q.fst r.fst) × (D.hom q.snd r.snd))
                       (f : (C.hom p.fst q.fst) × (D.hom p.snd q.snd)),
             ((C.compose g.fst f.fst), (D.compose g.snd f.snd)),
  -- We will use the laws of the parent categories to prove these rules.
  left_id := by { intros, simp, rw C.left_id, rw D.left_id, simp }
  right_id := by { intros, simp, rw C.right_id, rw D.right_id, simp }
  assoc := by { intros, simp, rw C.assoc, rw D.assoc }
}
--src/instances/Product_category.lean
```

3.2 Functors

A functor is a mapping that takes objects and arrows from one category to another category while preserving the relations between objects. To achieve this, it includes functions mapping each object and morphism from the first category to objects and morphisms in the second one, while satisfying some laws. The identity morphism should map to identity, and the composition of morphisms should give the same result regardless of whether it is applied before or after the morphism mapping.

```
structure functor (C D : category) :=
(map_obj : C → D)
(map_hom : Π {X Y : C} (f : C.hom X Y),
         D.hom (map_obj X) (map_obj Y))
(id   : ∀ (X : C),
      map_hom (C.id X) = D.id (map_obj X))
(comp : ∀ {X Y Z : C} (f : C.hom X Y) (g : C.hom Y Z),
      map_hom (C.compose g f) = D.compose (map_hom g) (map_hom f))
--src/functors/functor.lean
```

Using the product category we defined in section 3.1, we can define an example functor from category C to category $C \times C$: the diagonal functor, which maps each object c to the pair (c, c) and similarly the morphisms.

```
def diagonal_functor (C : category) : functor C (Product C C) :=
{
  map_obj := λ (c : C), (c, c),
  map_hom := λ {c d : C} (h : C.hom c d), (h, h),
  id      := by { intros, refl }  -- trivial proofs
  comp    := by { intros, refl }
}
--src/functors/diagonal.lean
```

3.3 Universal constructions

Universal constructions correspond to the most general object in a category that satisfies some property. As an important example, we will look at *product objects*.

A product of two objects c and d is the *unique* (up to isomorphism[2]) object $c \times d$ with two projection morphisms p_1 to c and p_2 to d. In addition, for all objects x with arrows to c and d, there is a *unique* morphism from x to $c \times d$ that composed with the projections returns the initial arrows of x.

```
structure binary_product[3] {C : category} (c d : C) :=
(p : C)
(p₁ : C.hom p c)
(p₂ : C.hom p d)
(ue  : Π (x : C) (x₁ : C.hom x c) (x₂ : C.hom x d),
     C.hom x p)
(ump : ∀ (x : C) (x₁ : C.hom x c) (x₂ : C.hom x d),
     x₁ = C.compose p₁ (ue x) ∧ x₂ = C.compose p₂ (ue x))
```

[2]There may be more objects with the property, but all have the same relations with other objects.

[3]This structure is changed from the library code; there `x`, `x₁` and `x₂` are all bundled together.

```
(uu  : ∀ (x : C) (x₁ : C.hom x c) (x₂ : C.hom x d) (h : C.hom x.x p),
     x₁ = C.compose p₁ h ∧ x₂ = C.compose p₂ h → h = ue x)
--src/universal_properties/binary_product.lean
```

By binary product's universal property, we can always construct a morphism from $c \times i$ to $d \times j$ that essentially maps c to d by morphism f, and i to j by morphism g; we will name this the product of morphisms $f \times g$.

```
def product_morphism[4] {C : category} {c d i j : C}
{cxi : binary_product c i} {dxj : binary_product d j}
(f : C.hom c d) (g : C.hom i j) : C.hom cxi.p dxj.p
:= dxj.ue cxi.p (C.compose f cxi.p₁) (C.compose g cxi.p₂)
--src/universal_properties/product_morphism.lean
```

We can now construct a product functor from category $C \times C$ to C as some sort of analogue to the diagonal functor, as long as category C has all products defined. It is trivial to see that a pair (c, d) in $C \times C$ can be mapped to $c \times d$, while a pair of morphisms (f, g) in $C \times C$ can be mapped to the product of morphisms $f \times g$.

```
def product_functor (C : category) [has_all_products C]
: functor (Product C C) C :=
{
  map_obj := λ (c : Product C C), (po c.fst c.snd).p,
  map_hom := λ {p q : Product C C} (m : (Product C C).hom p q),
    begin
      -- for ease we define consistent with previous notation
      let f := m.fst, -- f as the left element of the tuple m
      let g := m.snd, -- g as the right element of m
      -- now we just construct product_morphism
      exact product_morphism f g, -- fxg
    end,
  id   := by { intros, simp, rw identity_morphism_of_product, refl }
  comp := by { intros, simp, symmetry,
               apply product_of_composible_morphisms }
}
--src/functors/product.lean
```

We have to skip over the complete proofs of identity and composition for lack of space, however they can be found in the file defining the product of morphism `universal_properties/product_morphism.lean` under the names used in the code snippet above.

3.4 Natural transformations

Natural transformations are perhaps the core concept of category theory, with pioneer Mac Lane remarking that the notions of categories and functors were introduced only after natural transformations [12].

Natural transformations represent *uniform* ways with which to transform objects mapped by one functor into objects mapped by another functor, assuming both functors are between the same categories and in the same direction. Uniform means

[4] This definition is changed in a similar manner to `binary_product`.

that, if we have two objects X and Y in one category and functors F and G to some other category, then for each morphism f from X to Y, mapping $F(X)$ to $G(Y)$ and then applying $G(f)$ to get $G(Y)$, or applying $F(f)$ to get $F(Y)$ and then using the natural mapping to $G(Y)$ leads to the same results, for every combination of X and Y. The two paths in figure 1 should be the same (the diagram should commute), where α_X is the natural mapping from $F(X)$ to $G(X)$.

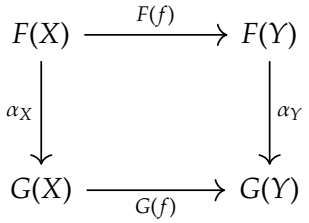

Figure 1: Natural transformation commuting diagram.

```
structure natural_transformation {C D : category} (F G : functor C D) :=
(α : Π (X : C.C₀) , D.hom (F.map_obj X) (G.map_obj X))
(naturality_condition : ∀ {X Y : C.C₀} (f : C.hom X Y),
  D.compose (G.map_hom f) (α X)  =
  D.compose (α Y) (F.map_hom f)
)
--src/natural_transformation.lean
```

4 Adjunctions

We can now begin to talk about *adjunctions*. Adjunctions, defined by two *adjoint functors* between the same categories but in reverse directions, express the idea of the functors being opposite or *dual* operations. Remarkably, very often it is the case that mathematical constructions are adjoint functors [10, p. vii].

For example, if we consider $(\mathbb{Z}, \leq)$ and $(\mathbb{R}, \leq)$ as categories with functors U taking each integer number to its real representation and $\lceil r \rceil$ taking each real r to its ceiling, then the two functors form an adjunction, with the ceiling functor being left adjoint to U [11, p. 39].

4.1 Definitions

Adjunctions have multiple definitions, all of them being equivalent. Here we will only mention two. Let C and D be two categories, with L a functor from C to D and R a functor from D to C. Then, L and R are adjoint and define an adjunction if we have a bijection φ mapping each morphism in D of the form $L\ c \to d$ (where c in an object in C and d is an object in D) to a morphism in C of the form $c \to R\ d$, so that it is natural in c and d (that is, figures 2 and 3 below commute for every morphism h to c and every morphism k to d), and likewise for the reverse direction of the bijection.

There is an alternative definition for adjunction in terms of natural transformations which is sometimes easier to use. Two functors F and R defined as before are adjoint if there are two natural transformations η — the *unit* — from Id_D (identity functor of D) to $R \circ L$ and ϵ — the *counit* — from $L \circ R$ to Id_C, so that for all c, $\epsilon(L\ c) \circ L(\eta\ c) = id(L\ c)$ and likewise for all d.

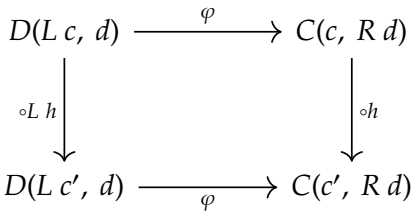

Figure 2: Naturality in c.

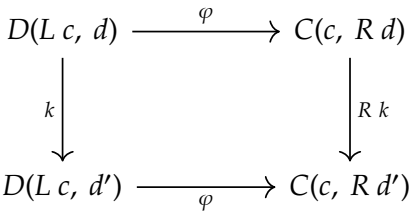

Figure 3: Naturality in d.

```
structure adjunction_hom {C D : category}
    (L : functor C D) (R : functor D C) :=
(ϕ : Π {c : C} {d : D},  (D.hom (L c) d) → (C.hom c (R d)))
(ϕr : Π {c : C} {d : D}, (C.hom c (R d)) → (D.hom (L c) d))
(sect : ∀ {c : C} {d : D} (h : C.hom c (R d)), (ϕ ∘ ϕr) h = h)
(retr : ∀ {c : C} {d : D} (k : D.hom (L c) d), (ϕr ∘ ϕ) k = k)
(naturality_c : ∀ (c : C) (d : D) (dh : D.hom (L c) d),
                ∀ {c' : C} (h : C.hom c' c),
              C.compose (ϕ dh) h = ϕ (D.compose dh (L.map_hom h)))
(naturality_d : ∀ (c : C) (d : D) (dh : D.hom (L c) d),
                ∀ {d' : D} (k : D.hom d d'),
              C.compose (R.map_hom k) (ϕ dh) = ϕ (D.compose k dh))
(naturality_cr : ∀ (c : C) (d : D) (ch : C.hom c (R d)),
                 ∀ {c' : C} (h : C.hom c' c),
               D.compose (ϕr ch) (L.map_hom h) = ϕr (C.compose ch h))
(naturality_dr : ∀ (c : C) (d : D) (ch : C.hom c (R d)),
                 ∀ {d' : D} (k : D.hom d d'),
               D.compose k (ϕr ch) = ϕr (C.compose (R.map_hom k) ch))
--src/adjunctions/homset.lean
```

```
structure adjunction_unit {C D : category}
    (L : functor C D) (R : functor D C) :=
(η : natural_transformation (Id C) (R∘L))
(ε : natural_transformation (L∘R) (Id D))
(id_L : ∀ (c : C),
  D.compose (ε.α (L c)) (L.map_hom (η.α c)) = D.id (L.map_obj c))
(id_R : ∀ (d : D),
  C.compose (R.map_hom (ε.α d)) (η.α (R d)) = C.id (R.map_obj d))
--src/adjunctions/unit_counit.lean
```

These two definitions are equivalent as proved in files `adjunctions/to_hom.lean` and `adjunctions/to_unit.lean`.

As another example, we have proved that the two functors introduced in section 3 form an adjunction. We shall have to skip once more over the code, but it can also be found in the repository.

4.2 Currying

In programming languages, *currying* is a way to change functions of n variables into a sequence of n functions that take a single variable each. For a more theoretic definition, currying represents an equivalence between morphisms from product objects: $b \times c \to d$, and morphisms from the first object in the product to some

sort of *function object* standing in for morphisms from the second object to a result: $b \to$ *object of* $C(c,d)$[5].

This function object is formalised in category theory by the *exponential object*, defined as a universal construction which specifies the object d^c and its *evaluation arrow* $d^c \times c \to d$, such that for all objects b and morphisms $g : b \times c \to d$, there is an unique morphism $g* : b \to d^c$ preserving the expected behaviour of the function object (making diagram 4 commute).

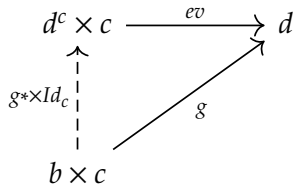

Figure 4: Exponential commuting diagram.

```
structure exponent {C : category} [has_all_products C] (b a : C) :=
(ob : C)
(ev : C.hom (po ob a).p b)
(ue : Π (c : C) (g : C.hom (po c a).p b),
    C.hom c ob)
(ump : ∀ (c : C) (g : C.hom (po c a).p b),
     g = C.compose ev (product_morphism (ue c g) (C.id a)))
(uu : ∀ (c : C) (g : C.hom (po c a).p b) (h : C.hom c ob),
    g = C.compose ev (product_morphism h (C.id a)) → h = ue c g)
--src/universal_properties/exponent.lean
```

We can already identify this concept of exponentiation with functions that can be fully memoized into *lookup tables* in computing [13, p. 144]. Both concepts create a new object within the type system that, when coupled with an index or exponent respectively, evaluates to the result of the original function. Separate from memoization, exponentials also give rise to higher-order functions that either take another function as argument or return it.

A category that has all products, exponentials, and a terminal object (object to which all objects in a category have a unique morphism to) is called cartesian closed. This notion is especially interesting to theoretical computer scientists as there is a correspondence between these types of categories and typed lambda calculus [14], a model of computation for functions and applications on which many functional programming languages such as Haskell or ML are based.

We have now covered almost all background knowledge necessary to introduce our final result: the proof that currying works, that there exists an equivalence between morphisms of types $b \times c \to d$ and $b \to d^c$. It is clear that currying can only work in a cartesian closed category (where we have all products and exponentials), and that the equivalence of morphisms can be expressed in terms of an adjunction by our first definition.

By introducing two new functors: the right product functor $-\times c$ mapping objects b to $b \times c$ and morphisms $f : b \to d$ to $f \times Id_c$; and the exponential functor $(-)^c$ mapping objects b to b^c and morphisms $f : b \to d$ to the left component of the

[5] In the same notation as the figures in 4.1.

unique morphism between $b^c \times c$ and $d^c \times c$ (of type $b^c \to d^c$, see diagram 5), we can express the currying adjunction as such:

$$- \times c \dashv (-)^c$$

The bijection between $b \times c \to d$ and $b \to d^c$ is given precisely by diagram 4 (we obtain the direct mapping by constructing $g*$ from g with exponentials' universal property, while the reverse mapping is a consequence of the diagram commuting). The proof that these two mappings form a bijection, that they are reversible mappings, comes from the uniqueness and mapping properties of exponentials, respectively.

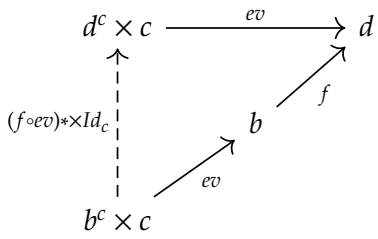

Figure 5: Exponential functor morphism mapping.

```
def rproduct_exponentiation_adjoint {C : category} [has_all_products C]
[has_exponentiation C] (c : C)
: adjunction_hom (r_product_functor c) (exponentiation_functor c) :=
{
  ϕ := by { intros a b h, exact (exp b c).ue a h, },
  ϕr := by { intros a b h',
     exact C.compose (exp b c).ev (product_morphism h' (C.id c)), },
  sect := -- to prove: (ϕ ∘ ϕr) ∘ h = h
  begin
      intros a b h', simp, symmetry,
      apply (exp b c).uu a
          (C.compose (exp b c).ev (product_morphism h' (C.id c))) h',
      refl,
    end,
  retr := -- to prove: (ϕr ∘ ϕ) ∘ h = h
    begin
      intros a b h, simp, symmetry,
      exact (exp b c).ump a h,
    end,
--src/adjunctions/rproduct_exponentiation.lean
```

We shall have to once more skip over parts of code, in this case the naturality proofs of the adjunctions, but these can be proved using product morphism lemmas and the properties of the exponential construction, as can be seen in the repository.

We have showed that the right product and exponentiation functors form an adjunction. For functional programming, where the language's functionality is similar to cartesian closed categories that have this adjunction, the meaning of this is: we can freely convert between curried and uncurried versions of functions. In Haskell

for example, where functions are curried by default, we can identify uncurried functions with tuple arguments.

5 Discussion

5.1 Design decisions

In this section we will present a few design decisions taken while implementing the library along with the motivation behind them and their importance.

Category definition. There are two main ways to define a category: with a collection of morphisms for each pair of objects (the definition we used), and with a single collection of morphisms along with two functions that define the source and target of each morphisms. The definitions are equivalent, but each may be more appropiate for different purposes (for example, an alternative category definition that avoids representing objects[6] is based on the single-collection definition [10, p. 279]).

Given the limited scope of the project, the use of the first definition by most introductory texts, and its similarity to dependent type theory (the type of a morphism depends on the types of the objects), we chose to build our library starting with the family of collections of morphisms definition.

Structure design. Out of the box, Lean has two tools that can help with defining the mathematical concepts we deal with: the *structure* datatype which groups together multiple values, and *type classes* which define a family of types with common properties.

We found that using type classes for mathematical structures is not helpful for our use case, although recommended by some literature [15]; most importantly, since the name of the actual instance of the type class becomes hidden, `X.hom c d` would be expanded to `category.hom c d` in the information view, hiding the name of the category or other structures implemented using type classes.

One other aspect involved in the design of the mathematical structures in our code was the packing, or lack of, of structure members into bundles. Packing represents a trade-off between the readability and writability of proofs. This can be seen in the implementation of some universal constructions: the binary product has some auxillary structures named *bundles*[7] that pack together an object with its two projections, while the exponential was defined without any packing.

```
structure binary_product_bundle {C : category} (c d : C) :=
(x : C)
(x₁ : C.hom x c)
(x₂ : C.hom x d)
--src/universal_properties/binary_product.lean
```

In these two cases, the packed universal construction is easier to reason about and the proofs are easier to understand, as we do not need to fully specify the object and its arrows when applying the properties of lemmas. However, we have to create one such bundle object every time we want to use it, some identities are not inferred by Lean and have to be manually specified, and we have to convert between packed and unpacked form whenever the other is more convenient to use. The unpacked representation tends to be easier to prove theorems with, at the cost of larger expressions.

[6] Objects can be identified by their identity morphism, after all.

[7] Similar to the mixins mentioned in [16].

The last structure design choice we will discuss is the bundling of parameters. A set of properties that apply to an object can generate type classes such as `is_universal` parametrised on the object, with its properties as components and with instances proving the object respects these properties. While very generic, this representation cannot easily serve as a standalone object, thus in our implementation the object is bundled with its properties and proofs in a standalone structure. An analysis of this design choice together with an alternative solution found in Lean's standard library has been made by Baanen [17, 4:9].

Preserving computability. Lean is based on constructive logic, a more restrictive foundation than classical logic. Although it can also support classical reasoning, that introduces noncomputable theorems which we will try to avoid. Existential qualifiers have different interpretations under classical and constructive logic. We have avoided the problem of extracting a witness from $\exists$ by converting expressions using this quantifier to remove it.

Taking binary products as an example, the universal property as stated in subsection 3.3 contains an existential qualifier. We can Skolemize [18] this proposition, turning the existential qualifier into a function taking an object with its two arrows and "naming" the unique morphism. Without an existential qualifier, in the code we refer to the unique morphism by `ue`, and all other morphisms with the same properties will be equal to it (as illustrated by the `uu` component in the code).

5.2 Comparison with other libraries

Our library implements a very limited section of category theory. It does not go much further than adjunctions, while examples are kept simple: defining the category of categories, in most other libraries the unique example of a category, has not even been attempted. This was a conscious decision, as supporting more than small categories already introduces universe inconsistencies in Lean. The usual solutions are admittedly slightly awkward constructions that would take away from the clarity of the library.

In comparison with the mathlib library of mathematics for Lean [7], our library has more basic instances of concepts, with little to no proof automation. That would be a problem for a library meant for use in proof-writing, however this makes our proofs step-by-step and explicit, an advantage for educational use. It also generates a sizeable body of lemmas that would otherwise be automatically inferred by Lean.

We will now present some of the concepts that do exist in this library but not in others. Exponential objects are a part of the library, unlike [7], [19]. Currying, in the form of the hom-functor adjunction, is not present in [8]. As far as we know, some of the simple instances (preorder forming a category) do not exist in many of the other categories.

6 Conclusion

In this paper we presented the categorical background behind currying, backed by a formalisation of category theory in Lean. The difference between this library of category theory and existing ones lies in the focus: readability at the expense of generalisation, more examples rather than more concepts. We also discussed some of the decisions taken during the development of the library, and very briefly contrasted it to other similar work. Compared to that, our library is very incomplete but likely more easily understood by beginners, featuring less automation in proofs and a structure closer to reference textbooks.

A possible future improvement to this work could be to continue the development of the library and add more definitions, theorems and lemmas to it. It would also be interesting to test it in a learning environment such as within a university course teaching category theory. However, a rework of the library would probably be needed to make the parts contributed by different people more uniform.

Responsible research

The product of this research, the collection of formal definitions, proofs and examples for category theory provides a rigurous-by-default example of mathematical reasoning. We will address two aspects in this section: reproducibility and ethics.

The reproducibility of the research is guaranteed as the code is publicily available online along with instructions for running the project and a list of necessary software. As code written in a theorem prover, the behaviour is deterministic — as long as the version of code downloaded has passed the CI pipeline it will continue to typecheck under all installations of the same version of Lean. We have also included a discussion of some design choices made in section 5, which can aid readers intending to create their own implementation.

Regarding ethical considerations, we are aware that there exist ethical risks in all fields of mathematics [20]. This library serves as a tool for exploration into abstract mathematics, with no likely way of influencing decisions in fields such as policy-making or legal systems. Although it may not have any immediate impact on the world, in contrast to a statistical model for example, there is always a possibility of this work having effects, perhaps by introducing more students to the fields of category theory or formal verification which we view as a good consequence, however we believe this work has little likelihood of having negative ethical consequences.

References

[1] J. Baez and M. Stay, "Physics, Topology, Logic and Computation: A Rosetta Stone," in *New Structures for Physics*, B. Coecke, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 95–172, ISBN: 978-3-642-12821-9. DOI: 10.1007/978-3-642-12821-9_2.

[2] M. P. Jones, "A system of constructor classes: Overloading and implicit higher-order polymorphism," *Journal of Functional Programming*, vol. 5, no. 1, pp. 1–35, 1995. DOI: 10.1017/S0956796800001210.

[3] P. Wadler, "Comprehending monads," in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, ser. LFP '90, Nice, France: Association for Computing Machinery, 1990, pp. 61–78, ISBN: 089791368X. DOI: 10.1145/91556.91592.

[4] C. Paulin-Mohring, "Introduction to the Calculus of Inductive Constructions," in *All about Proofs, Proofs for All*, ser. Studies in Logic (Mathematical logic and foundations), B. W. Paleo and D. Delahaye, Eds., vol. 55, College Publications, Jan. 2015.

[5] J. Avigad, L. de Moura and S. Kong, *Theorem Proving in Lean*. 2023. [Online]. Available: https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf.

[6] A. Baanen, A. Bentkamp, J. Blanchette, J. Hölzl and J. Limperg, *The Hitchhiker's Guide to Logical Verification*. 2022. [Online]. Available: https://raw.githubusercontent.com/blanchette/logical_verification_2022/main/hitchhikers_guide.pdf.

[7] The mathlib Community, "The Lean Mathematical Library," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2020, New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 367–381, ISBN: 9781450370974. DOI: 10.1145/3372885.3373824.

[8] J. Z. S. Hu and J. Carette, "Formalizing Category Theory in Agda," in *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2021, Virtual, Denmark: Association for Computing Machinery, 2021, pp. 327–342, ISBN: 9781450382991. DOI: 10.1145/3437992.3439922.

[9] T. Leinster, *Basic Category Theory* (Cambridge Studies in Advanced Mathematics). Cambridge University Press, 2014. DOI: 10.1017/CBO9781107360068.

[10] S. Mac Lane, *Categories for the Working Mathematician* (Graduate texts in mathematics), Second Edition. Springer-Verlag New York, 1998, vol. 5.

[11] B. C. Pierce, "A taste of category theory for computer scientists," Feb. 2011. DOI: 10.1184/R1/6602756.v1.

[12] S. Mac Lane, "The development and prospects for category theory," *Applied Categorical Structures*, vol. 4, no. 2, pp. 129–136, 1996. DOI: 10.1007/BF00122247.

[13] B. Milewski, *Category Theory for Programmers*. 2023. [Online]. Available: https://github.com/hmemcpy/milewski-ctfp-pdf/releases/download/v36-98b71ac/ctfp--98b71ac.pdf.

[14] J. Lambek, "Cartesian closed categories and typed λ-calculi," in *Combinators and Functional Programming Languages*, G. Cousineau, P.-L. Curien and B. Robinet, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 136–175, ISBN: 978-3-540-47253-7.

[15] B. Spitters and E. van der Weegen, "Type classes for mathematics in type theory," *Mathematical Structures in Computer Science*, vol. 21, no. 4, pp. 795–825, 2011. DOI: 10.1017/S0960129511000119.

[16] F. Garillot, G. Gonthier, A. Mahboubi and L. Rideau, "Packaging mathematical structures," in *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, ser. TPHOLs '09, Munich, Germany: Springer-Verlag, 2009, pp. 327–342, ISBN: 9783642033582. DOI: 10.1007/978-3-642-03359-9_23.

[17] A. Baanen, "Use and Abuse of Instance Parameters in the Lean Mathematical Library," in *13th International Conference on Interactive Theorem Proving (ITP 2022)*, J. Andronick and L. de Moura, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 237, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 4:1–4:20, ISBN: 978-3-95977-252-5. DOI: 10.4230/LIPIcs.ITP.2022.4.

[18] A. Bundy and L. Wallen, "Skolemization," in *Catalogue of Artificial Intelligence Tools*, A. Bundy and L. Wallen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 123–123, ISBN: 978-3-642-96868-6. DOI: 10.1007/978-3-642-96868-6_235.

[19] J. Gross, A. Chlipala and D. I. Spivak, "Experience implementing a performant category-theory library in coq," in *Interactive Theorem Proving*, G. Klein and R. Gamboa, Eds., Cham: Springer International Publishing, 2014, pp. 275–291, ISBN: 978-3-319-08970-6.

[20] M. Chiodo and P. Bursill-Hall, "Four levels of ethical engagement," *Ethics in Mathematics Project*, Ethics in Mathematics Discussion Papers, no. 1/2018, [Online]. Available: https://ethics.maths.cam.ac.uk/assets/dp/18_1.pdf.