Turkish Journal of Mathematics

Volume 46 | Number 4 Article 31

1-1-2022

# Formal categorical reasoning

BURAK EKİCİ

TÜBİTAK

Turkish Journal of Mathematics

http://journals.tubitak.gov.tr/math/

Research Article

Turk J Math
(2022) 46: 1538 – 1552

doi:10.55730/1300-0098.3178

# Formal categorical reasoning

**Burak EKİCİ***
Department of Computer Engineering, TED University, Ankara, Turkey

**Received:** 27.02.2022 • **Accepted/Published Online:** 08.04.2022 • **Final Version:** 05.05.2022

**Abstract:** In this paper, we present a category theory library developed in the proof assistant Coq. We discuss the design principles of the library in comparison with those existing out there. To explicitly demonstrate the utility of the library, we conclude with a case study in which a Coq formalized soundness proof of the intuitionistic propositional logic within a category theoretical settings is examined.

**Key words:** Categorical logic, denotational semantics of programming constructs, formal proofs, the Coq proof assistant

## 1. Introduction

Category theory proposes structural foundations to do mathematics. A category is simply a structure or collection of objects and arrows such that for every single object there is the identity arrow, the composition operation has been defined over arrows which must be associative and identities cancel within the composition. Therefore, it may be viewed as an alternative especially to axiomatic set theories.

Among other capabilities, category theory serves as a domain language of interpretation for various sorts of computational calculi [8]. For instance, the category induced by any cartesian closed preorder speaks the same formal language with intuitionistic propositional logic (IPL). Similarly, simply typed lambda calculi can be viewed as the internal language of cartesian closed categories [11, 13, 14]. Monads [15] thus adjunctions have been benefited by and large when it comes to capturing computational side effects, and implemented in Haskell language [21]. Besides, the structural nature of category theory makes it very suitable to be formalized in proof assistants such as Coq [10, 19, 20, 22], Agda [2, 12, 16, 17] and in Mizar [1].

**Outline.** In Section 2.1, we briefly present the Coq proof assistant as it is the formalization environment for the category theory library introduced in Sections 2.2 and 2.3. The library [6] is centered around objects that help modeling programming language constructs, and can be viewed as the first step towards providing some formal recipe to study programming language semantics in a mechanized fashion. We conclude, in Section 3, with a case study where we discuss a Coq formalized proof of soundness for the intuitionistic propositional logic within the scope of a category induced by any cartesian closed preorder.

## 2. Categorical structures in Coq

In this section, we give a brief overview of the proof assistant Coq before diving into the design principles of the category theory library we develop therein.

*Correspondence: burak.ekici@tedu.edu.tr

### 2.1. Coq: an interactive theorem prover

Coq [3] is a formal proof management system (so called proof assistant) which embodies a functional programming language (as known as Gallina) alongside a specification language. Gallina allows for developing recursive functions while the specification language permits one to state properties of such functions and give them proofs. The language Gallina is not Turing complete as it rejects any sort of nonterminating function or type construction. This is simply because the nontermination effect in such kind of proof systems would result in proofs of "false", making the entire system inconsistent.

The philosophical and mathematical foundations on which the proof management system Coq has been based is motivated by the Curry–Howard isomorphism (CHI).

| Logic | $\sim$ | Type theory |
|---|---|---|
| Proposition | | Type |
| Proof | | Program |
| $A \wedge B$ | | $A \times B$ |
| $A \vee B$ | | $A + B$ |
| $A \implies B$ | | $A \to B$ |
| $\forall x \in A, P\,x$ | | $\Pi_{x:\,A}, P\,x$ |
| $\exists x \in A, P\,x$ | | $\Sigma_{x:\,A}, P\,x$ |
| $=_A$ | | $\mathrm{Id}_A$ |

As demonstrated in the above table, the CHI highlights the direct relation, equivalence or correspondence in between constructive logic and type theory. One could simply think of type theory as an alternative to set theory proposing foundations to do constructive mathematics. According to CHI, a proposition or assertion by the side of constructive logic could be viewed as a type (naively, a collection of things or inhabitants) by type theory side. Similarly, any valid proof that makes the proposition hold can be taken as a program of the corresponding type. Having these said, we could now state the CHI slogan: "under certain type settings, proofs can be programmed." Otherwise put, instead of writing proofs down on paper, one could mechanize proofs by formalizing them within a proof assistant like Coq. This process is beneficial in the sense that mechanized proofs would be "more trustworthy" provided that the Coq kernel (the part responsible for performing proof-checking steps) is trusted. Namely, a proof might be much less error prone when it gets checked by a mechanical kernel as opposed to being checked by a pair of human eyes.

The type theory (or the formal logic) backing the Coq proof assistant is called the calculus of inductive constructions (CIC). The logic in fact extends simply typed lambda calculus with a few structures such as dependent types, type operators, type polymorphism, a restricted form of inductive types and primitive recursion. The restriction is to ensure that every single inductive construction and recursive function terminates. As this is a form of the Halting Problem which is provably undecidable, the Coq termination checker is naturally restricted, and suffices to check whether recursive calls are performed on "structurally decreasing" arguments. This means that a function which terminates in practice but not implemented embarking on the structurally decreasing arguments method could very well be rejected by Coq's termination checker. In such a case, one needs to provide additional information on the function definition usually employing well founded relations [9]. To ensure that inductive constructions halt, the strict positivity [4] is the criteria to be met.

The CIC is a constructive logic, namely it does not assume any classical axiom such as law of excluded middle (LEM) or any kind of equivalent like double negation elimination (DNE). However, it is provably safe to introduce classical axioms into CIC when it comes to doing classical mathematics. This means that CIC +

classical axioms does not yield in proofs of false.

In CIC, everything, even types themselves, has a type. The design of types of types (so called universes) is also a crucial point to be mentioned at a certain degree. There are there sorts of universes in Coq: `Prop`, `Set` and `Type`. `Prop` is indeed the impredicative universe of propositions in which computational abilities are restricted. This is due to the fact that one could prove false in an impredicative universe enriched with classical axioms and computational abilities. `Set` is the predicative universe of computations in which any form of classical axioms could safely be assumed. In general, to avoid inconsistencies, the computational abilities in `Prop` and the impredicativity property in `Set` are sacrificed. `Type` is in fact the universe of both `Set` and `Prop` presented in an hierarchy (or a stratification) to elude Girard-like paradoxes. In the below box, we further detail universe organization in CIC:

$$\texttt{CIC}\left\|\begin{array}{l}\mathcal{S} := \texttt{Set}, \texttt{Prop}, \texttt{Type}_{\texttt{i}} \text{ s.t. } \texttt{i} \in \mathbb{N} \\ \mathcal{A} := (\texttt{Set} : \texttt{Type}_0), (\texttt{Prop} : \texttt{Type}_0), (\texttt{Type}_{\texttt{i}} : \texttt{Type}_{\texttt{i}+1}) \text{ s.t. } \texttt{i} \in \mathbb{N} \\ \mathcal{R} := (\texttt{Prop}, \texttt{Prop}), (\texttt{Set}, \texttt{Prop}), (\texttt{Type}_{\texttt{i}}, \texttt{Prop}), (\texttt{Prop}, \texttt{Set}), (\texttt{Set}, \texttt{Set}), \\ \qquad (\texttt{Type}_{\texttt{i}}, \texttt{Set}), (\texttt{Type}_{\texttt{i}}, \texttt{Type}_{\texttt{j}}, \texttt{Type}_{\texttt{max(i,j)}}), \text{ s.t. } \texttt{i}, \texttt{j} \in \mathbb{N}\end{array}\right.$$

There, $\mathcal{S}$ is the set of universes while $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ being the set of axioms clarifying the typing relations between universes: the universes `Set` and `Prop` live in $\texttt{Type}_0$ while the universe $\texttt{Type}_{\texttt{i}}$ lives in $\texttt{Type}_{\texttt{i}+1}$ for each natural number `i`. And, $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is the set of object level rules which determine what kind of (dependent) function types can be formed, and in which universe they live. For instance, the second rule in the first column could be read as it is possible in `CiC` to form function types from any instance of universe `Set` to any instance of `Prop`, and the newly formed function types live in the universe `Prop`. In addition, `CIC` has:

- universe cumulativity: $\texttt{Prop} \subseteq \texttt{Type}_0$, $\texttt{Set} \subseteq \texttt{Type}_0$, $\texttt{Type}_{\texttt{i}} \subseteq \texttt{Type}_{\texttt{i}+1}$ s.t. $\texttt{i} \in \mathbb{N}$.

- typical ambiguity: no explicit mention of universe levels. From the users' viewpoint, it is $\texttt{Type} : \texttt{Type}$ but not in practice. This would allow for a proof of Girard's paradox leading to an inconsistency.

### 2.2. Design principles

In order to computerize a mathematical framework or theory, the first thing that needs to be done is formalizing mathematical objects in the provided programming language. In these lines, this section is devoted to briefly discuss the design principles of the category theory library we have been developing in the language of Coq. For the entire formalization, please refer to the link below:

https://github.com/ekiciburak/CatTheo

In developing category theoretical objects, we employ Coq type classes along with the use of dependent types. For instance, below given is indeed the formalization of categorical functors.

```
Class Functor (C D: Category): Type ≜
  {
    fobj            : @obj C  →  @obj D;
    fmap            : ∀ {a b: @obj C} (f: arrow b a), arrow (fobj b) (fobj a);
    preserve_id     : ∀ {a: @obj C}, fmap (@identity C a) = (@identity D (fobj a));
    preserve_comp   : ∀ {a b c: @obj C} (g : @arrow C c b) (f: @arrow C b a),
                        fmap (g o f) = (fmap g) o (fmap f)
  }.
```

As a side note, a functor is simply a map of categories mapping objects and arrows preserving the identities and composition. Therefore, the field named `fobj` maps objects while the one called `fmap` is responsible for transportation of arrows. The last two fields, namely `preserve_id` and `preserve_comp`, are in fact the proof obligations making sure that the identities and composition are preserved by `fmap`.

The use of dependent types becomes much more clear in the context of the `fmap` field. Suppose we have an arbitrarily given pair of categories $\mathcal{C}$ and $\mathcal{D}$ together with a functor $F\colon \mathcal{C} \to \mathcal{D}$. Then, the functor $F$ maps every single arrow from $a \to b$ in the category $\mathcal{C}$ into an arrow `fobj` $F\ a \to$ `fobj` $F\ b$ in the category $\mathcal{D}$. As clearly demonstrated here, the type of `fobj` totally depends on the functor instance $F$. This is not surprising at all. Indeed, this is a prize that we want to obtain in the end. On the one hand, employing dependent types when it comes to defining or declaring mathematical objects brings in a high level of flexibility and increases the expressiveness score. However, on the other hand, it may be pretty cumbersome to prove properties of dependently typed objects. To clarify this point further, we go through an example in what follows.

Suppose we are given a pair of categories $\mathcal{C}$ to $\mathcal{D}$ along with functors $F$ and $G$ defined from the category $\mathcal{C}$ to $\mathcal{D}$, and the task is to prove that $F = G$. The sameness measure "=" here is the Leibniz equality of type theory which is solely constructed by the reflexivity property. To conclude that such an equality holds, under the Coq implementation presented above, one simply needs to demonstrate as subgoals that functors $F$ and $G$ maps objects and arrows in the same way. Namely,

$$\begin{array}{lllll} (1) & \texttt{fobj} & \mathrm{F} & = & \texttt{fobj}\ \ \mathrm{G} \\ (2) & \texttt{fmap} & \mathrm{F} & = & \texttt{fmap}\ \ \mathrm{G}. \end{array}$$

In order to compare instances with respect to the Leibniz equality, one first of all needs to ensure that they are of the same type. Namely, they inhabit the same type. Equation (1) above could be stated in Coq with no problem as the type of both sides are the same. This means that the expression is well-typed. Then, the question of "whether the equality holds" of course depends on the definitions of $F$ and $G$. Unlikely, Equation (2) is ill-typed thus cannot be directly implemented in Coq. The type of the expression `fmap` $F$ is $\forall a\, b$ in $\mathcal{C}, (a \to b) \Rightarrow (\texttt{fobj}\, F\, a \to \texttt{fobj}\, F\, b)$ while that of `fmap` $G$ is $\forall a\, b$ in $\mathcal{C}, (a \to b) \Rightarrow (\texttt{fobj}\, G\, a \to \texttt{fobj}\, G\, b)$. The mentioned types are obviously different. In such a case, one needs to first convince the Coq type-checker that these types unify, and only then is permitted to attempt at proving that $F$ and $G$ maps arrows in the same manner. Indeed, this problem originates from the fact that the contextual equality of category theory cannot be entirely represented by the constructive (Leibniz) equality of type theory. This mismatch could be reduced to the problem of higher order unification which is undecidable in general [18]. However for particular cases, we could come up with certain solutions working out (contextually) explicit substitutions as presented in [5].

```
{p: (Π a b : obj, arrow b a → arrow (fobj F b) (fobj F a)) =
    (Π a b : obj, arrow b a → arrow (fobj G b) (fobj G a)) &
       match p in (_ = y) return y with
          | eq_refl ⇒ fmap F
       end = fmap G}
```

Even when we manage to prove that `fmap` $F$ and `fmap` $G$ are instances of the same type, we are still supposed to show that this proof is indeed the reflexivity proof. This is also due to the aforementioned mismatch. In order to close this gap, at this stage we employ the uniqueness of identity proofs (UIP) axiom to be able to state that any proof of equality is indeed the reflexivity itself.

Suppose we managed to show that the functors $F$ and $G$ maps objects and arrows in the same way. Would that suffice to state that $F = G$? Not really; as we are still supposed to demonstrate that the proof obligations, `preserve_id` and `preserve_comp`, for $F$ and $G$ are the same. To conclude this, we make use of the proof irrelevance axiom to obtain that different proofs of a proposition are equal.

Almost all existing libraries out there embark on similar design principles with small differences lying underneath the formalization of contextual equality of category theory. To do so, we employ the constructive equality of Coq similar to that of [19]. In [22], this task has been accomplished by Setoid equivalences. This approach makes things simpler when it comes to performing sameness proofs but introduces an overhead: requires to get the statement of "every function send equivalent elements to equivalent elements" proven every time.

Alternatively, [10] and [20] benefit from homotopy type theory proposals: there, the sameness measure is the constructive equality enriched by the univalence axiom (UA) of Voevodsky. The UIP is avoided to prevent contradictions simply because UA + UIP is inconsistent. This way, one could reshape the constructive equality into a certain form of equivalences and get the related proofs done accordingly.

### 2.3. Coverage and novelties

We managed to formalize, in Coq, many categorical structures ranging from categories themselves to monad algebras and Eilenberg–Moore structures together with lots of related property proofs. Below itemization clearly lists the coverage of the library.

- Categories, functors, Cat, natural transformations, functor categories, monads, adjunctions
- Preorder, monoid and poset formed/induced categories, intuitionistic propositional logic (IPL) $\cong$ Category induced by a cartesian closed preorder
- Simply typed lambda calculus: syntax, rules
- Initials, terminals, (co)products, exponentials
- Yoneda lemma
- Kleisli structures, comparison theorem for the Kleisli construction
- Monad algebras, Eilenberg–Moore sructures

The contribution that this paper makes lies on the formalization of some categorical structures and related property proofs that are not included in the existing libraries out there, to the best knowledge of the author. In these lines, novel implementations are in fact a few folded as motivated below:

1. We have formalized adjunctions with and without the use of Hom-functors, and proven the fact that they are equivalent. In a certain context, this may be crucial as for some proofs, the definition with Hom-functors might better work while some other proofs are better benefit from the definition with no use of Hom-functors. This way, it is possible to customize and switch between definitions in formalizing proofs. We also have universal property (UP) of adjunctions formalized. The UP provides sufficient conditions for an arbitrarily given functor in having a left or a right adjoint, and plays a role in interpreting dependent products in a categorical setting.

2. We have implemented a soundness proof of the IPL in the category induced by any cartesian closed preorder. We detail a few corner cases of the formalization in Section 3 as the case study of the paper. Provided that IPL is one of the simplest models of computation, its categorical interpretation along with a soundness proof could be taken as a very first step towards having Coq mechanized soundness guarantees for computational theories.

3. The formal proof of Mac Lane's comparison theorem for the Kleisli construction is another novel component of the library, and detailed in [7]. The theorem states that there is a unique functor that bridges an arbitrary adjunction and the adjunction induced by the Kleisli category of the monad handled out of the starting arbitrary adjunction. It is used by Duval and Jacobs in their categorical settings to interpret the state effect in impure programming languages. The proof is based on showing functor equalities for which we detailed a few challenges in Section 2.2. Currently, we are struggling with a version of the theorem involving Eilenberg–Moore structures and monad algebras replacing Kleisli constructions. This version also necessitates proving functor equalities. We did get the relevant challenges handled. The rest is all about having the category theoretical mathematics properly implemented, and reserved for future work.

## 3. Case study

This section is devoted to an interpretation of a fragment of intuitionistic preorder logic (IPL) within the scope of the category induced by any cartesian closed preorder. To get there, we first recall the domain and codomain of the interpretation; namely basics of (cartesian closed) preorders followed by the syntax and rules of the IPL fragment in Sections 3.1 and 3.2, form the interpretation function, in Section 3.3, and only then construct the soundness proof accordingly in Section 3.4. We put the Coq formalization up alongside the structures presented in the pen-and-paper style.

### 3.1. Cartesian closed preorders in Coq

A preorder $(P, \leqslant)$ is a collection $P$ of members enriched by a binary relation $\leqslant$ which is reflexive and transitive. We formalize preorders in Coq again by exploiting Coq type classes as follows:

```
Class PreOrder: Type ≜
{
   pos    : Set;
   pohrel : pos → pos → bool;
   porefl : ∀ x: pos, pohrel x x = true;
   potrans: ∀ x y z: pos, pohrel x y = true ∧ pohrel y z = true → pohrel x z = true
}.
```

Obviously, the field `pos` denotes the collection $P$, and is declared to be an instance of the `Set` universe of Coq. Simlarly, the `pohrel` field represents the underlying binary (Boolean valued) relation $\leqslant$. For instance, the term `pohrel x y = true` formalizes the fact that $x \leqslant y = true$ in $P$. The remaining fields are in fact proof obligations ensuring that the underlying relation is reflexive and transitive.

Preorders and preorder maps form a category as well as every single preorder induces a category. We are interested in the latter fact in terms of the case study of the paper. The category induced by a preorder $(P, \leqslant)$ contains members of $P$ as objects while arrows are specified with the following definition: for every pair of members $x, y$ of $P$, the induced category has a unique map from $x$ to $y$ if and only if $x \leqslant y = true$ holds in $P$. Implied by this definition, the identity arrows and arrow compositions are uniquely determined just because it is only possible to define at most one single arrow between every pair of objects. This could be reflected in a Coq implementation as follows:

```
Definition PreOrderICMap (P: PreOrder) (x y: @pos P): Type ≜
  if (@pohrel P x y) then unit else Empty_set.

Definition PreOrderDetCat (P: PreOrder): Category.
Proof. unshelve econstructor.
       - exact (@pos P).
       - intros x y. exact (PreOrderICMap P y x).
       - intro x. exact (PreOrderICid P x).
       - intros z y x g f. exact (PreOrderICcomp P x y z f g).
       - ...
Defined.
```

In the definition of `PreOrderDetCat`, we benefit from the use of Coq tactics which are indeed simple code pieces that could manipulate proof states. For instance, the tactic `unshelve econstructor` allows for a construction of the intended category field by field as opposed to have it done in one go. The objects of the induced category are set to be the underlying collection `@pos P` of the provided preorder `P`, thanks to the `exact` tactic. Similarly, the arrows are exactly stated by the above given `PreOrderICMap` function parameterized by any pair of `@pos P` members `x` and `y`. The tactic `intro` (with `intros` being the pluralized version) is indeed the introduction rule for (dependent) function types. Simply put, it separates the hypotheses from the goal statement. The sign '`@`' makes all parameters explicitly provided. The entire formalization including the definitions `PreOrderICid` and `PreOrderICcomp` could be found in the source file `Preorder.v`.

A preorder $(P, \leqslant)$ is called cartesian closed if it has

1. the greatest element $\top$ such that
   - $\forall p \in P, p \leqslant \top =$ true.
2. Binary meets "$p \diamond q$" for every pair of members $p$ and $q$ if and only if

- $(p \diamond q) \leqslant p =$ true and $(p \diamond q) \leqslant q =$ true;
- $\forall r \in P, r \leqslant (p \diamond q) =$ true $\iff r \leqslant p =$ true $\wedge\, r \leqslant q =$ true.

3. Heyting implications "$p \multimap q$" for every pair of members $p$ and $q$ if and only if

- $(p \multimap q) \diamond p \leqslant q =$ true;
- $\forall r \in P, r \leqslant (p \multimap q) =$ true $\iff (r \diamond p) \leqslant q =$ true.

As a matter of this fact, we get additional structures popping up in the induced category by a cartesian closed preorder such that (1) the greatest element constructs the terminal object, (2) binary meets form categorical binary products and (3) Heyting implications build categorical exponentials. This could simply be reflected in a Coq implementation, in the given order, as follows:

```
Class top (P: PreOrder): Type ≜
{
   ptop : @pos P;
   potob: ∀ (x: @pos P), (@pohrel P x ptop) = true
}.
```

```
Lemma tPreOrderDetCat: ∀ (P: PreOrder) (t: top P),
  Terminal (PreOrderDetCat P) (@ptop P t).
```

```
Class POBM (P: PreOrder) (p q: @pos P): Type ≜
{
   pobm   : @pos P;
   pobmpi1: @pohrel P pobm p = true;
   pobmpi2: @pohrel P pobm q = true;
   pobmuni: ∀ r, @pohrel P r pobm = true  ⟺
     @pohrel P r p = true ∧ @pohrel P r q = true
}.

Class hasPOBM (P: PreOrder): Type ≜
{
  hpobm: ∀ p q, POBM P p q
}.
```

```
Lemma binarymeet: ∀ (P: PreOrder)
  (p q: @obj (PreOrderDetCat P)) (h: POBM P p q),
  Product (PreOrderDetCat P) p q (@pobm P p q h).
```

```
Class POHI (P: PreOrder) (h: hasPOBM P)
           (p q: @pos P): Type ≜
{
   hi   : @pos P;
   hiapp: @pohrel P (@pobm P hi p (@hpobm P h hi p)) q = true;
   hiob : ∀ r, @pohrel P r hi = true  ⟺
     @pohrel P (@pobm P r p (@hpobm P h r p)) q = true
}.
```

```
Lemma heyting: ∀ (P: PreOrder)
  (p q: @obj (PreOrderDetCat P))
  (h1: hasPOBM P) (h2: hasPOHI P h1),
  @Exponential (PreOrderDetCat P)
     (hasProductsPreOrderDetCat P h1) p q
     (@hi P h1 p q (@hhi P h1 h2 p q)).
```

Here, we only state definitions on the left, alongside the propositions that state the corresponding categorical structures as briefly explained above. For the whole definitions and proof formalizations please refer to the source codes contained in the files `Terminal.v` (for the lemma `tPreOrderDetCat`), `Product.v` (for that of `binarymeet`) and `Exponential.v` (for details of `heyting`). Note also that in order to make the paper

self-contained, we provide a summary of related categorical structures accompanied by Coq implementations in Appendix 4. On a last note, provided that a category with the terminal object, binary products and exponentials is said to be cartesian closed, the category induced by any cartesian closed preorder is of course cartesian closed.

### 3.2. A fragment of IPL in Coq

The intuitionistic propositional logic (IPL) is one of the very primitive computational models consisting of a set of terms (a.k.a. propositions or formulae) alongside a set of rules governing the terms. The IPL is constructive. Namely, it lacks classical axioms such as LEM and any kind of equivalent like DNE. We present below, a fragment of IPL in which terms constructors are ground propositions, the truth value, the logical conjunction and implication.

$$
\begin{array}{lll}
\text{Terms of IPL} & \varphi, \psi, \theta, \ldots := & \\
\quad | & p, q, r, \ldots & \text{(ground) propositional identifiers} \\
\quad | & \text{true} & \text{truth} \\
\quad | & \varphi \,\&\, \psi & \text{conjunction} \\
\quad | & \varphi \Rightarrow \psi & \text{implication}
\end{array}
$$

$$
\begin{array}{lll}
\text{Sequents of IPL} & \Gamma := & \\
\quad | & [\,] & \text{empty} \\
\quad | & \varphi, \Gamma & \text{nonempty}
\end{array}
$$

Notice that there is no falsity listed, as a term constructor, simply because the logic is constructive and does not speak of "false" propositions. Note also that every single valid IPL term is stated under a context which could be viewed as a (potentially empty or `nil`) finite list or sequent of propositions.

We formalize the above syntax in the proof assistant Coq employing an inductive type `IPL` such that for every term constructor there is an `IPL` constructor listed in the exact same order.

```
Inductive IPL: Type ≜
 | pi   : IPL
 | truth: IPL
 | conju: IPL → IPL → IPL
 | impli: IPL → IPL → IPL.

Definition ctx ≜ list (IPL)%type.
Definition extend (c: ctx) (x: IPL) ≜ x :: c.
```

Also, we benefit from Coq polymorphic lists to implement contexts (`ctx`) of formulae. Then, the context extension is simply the `cons` (denoted "`::`") operator.

Abovestated IPL syntax is governed by the recursively defined "entailment $\vdash$" relation, and presented below.

$$\frac{}{\varphi, \Gamma \vdash \varphi}\ (\text{ax}) \qquad \frac{\Gamma \vdash \varphi}{\psi, \Gamma \vdash \varphi}\ (\text{wk}) \qquad \frac{\Gamma \vdash \varphi \quad \varphi, \Gamma \vdash \psi}{\Gamma \vdash \psi}\ (\text{cut})$$

$$\frac{}{\Gamma \vdash \text{true}}\ (\text{true}) \qquad \frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \,\&\, \psi}\ (\&\text{I}) \qquad \frac{\varphi, \Gamma \vdash \psi}{\Gamma \vdash \varphi \Rightarrow \psi}\ (\Rightarrow\text{I})$$

$$\frac{\Gamma \vdash \varphi \,\&\, \psi}{\Gamma \vdash \varphi}\ (\&\text{E}_1) \qquad \frac{\Gamma \vdash \varphi \,\&\, \psi}{\Gamma \vdash \psi}\ (\&\text{E}_2) \qquad \frac{\Gamma \vdash \varphi \Rightarrow \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ (\Rightarrow\text{E})$$

There, the expression $\Gamma \vdash \varphi$ denotes that the proposition $\varphi$ is entailed by the list of propositions contained in the context $\Gamma$. Remark also that the context extensions are demonstrated with the use of comma sign. E.g., "$\varphi, \Gamma$" denotes the new context obtained by extending $\Gamma$ with the proposition $\varphi$.

Semantically, the rules indeed ensure "how to introduce and eliminate" IPL terms. In these lines, the rule named "&I" clearly describes how to form or introduce the conjunction of an arbitrarily given pair of propositions $\varphi$ and $\psi$; while the ones called "$\&\mathrm{E}_1$" and "$\&\mathrm{E}_2$" clarify how to eliminate or destruct a given term conjunction into left and right components under some context $\Gamma$. The rule with the name "$\Rightarrow$E" is the eliminator for implication which is known as the modus-ponens, and allows for deduction of some proposition $\psi$, provided $\varphi \Rightarrow \psi$ and $\varphi$ at the same time, under some context $\Gamma$.

Similar to that of terms, we make use of a Coq inductive type `IPLE` to get the rules formalized. The type `IPLE` lands in Coq's `Prop` and is parameterized by a context and an IPL term.

```
Inductive IPLE: ctx → IPL → Prop ≜
  | ax1: ∀ (c: ctx) (phi: IPL), IPLE (phi :: c) phi
  | ax2: ∀ (c: ctx) (phi psi: IPL), IPLE c phi → IPLE (psi :: c) phi
  | ax3: ∀ (c: ctx) (phi psi: IPL), IPLE c phi → IPLE (phi :: c) psi → IPLE c psi
  | ax4: ∀ (c: ctx), IPLE c truth
  | ax5: ∀ (c: ctx) (phi psi: IPL), IPLE c phi → IPLE c psi → IPLE c (conju phi psi)
  | ax6: ∀ (c: ctx) (phi psi: IPL), IPLE (phi :: c) psi → IPLE c (impli phi psi)
  | ax7: ∀ (c: ctx) (phi psi: IPL), IPLE c (conju phi psi) → IPLE c phi
  | ax8: ∀ (c: ctx) (phi psi: IPL), IPLE c (conju phi psi) → IPLE c psi
  | ax9: ∀ (c: ctx) (phi psi: IPL), IPLE c (impli phi psi) → IPLE c phi → IPLE c psi.
```

Each rule is developed as an `IPLE` constructor with a single difference in rule and proposition renamings: we number the rules from the left-top to the bottom-right. For instance, the rule "ax" is named `ax1` while the one "&I" is called `ax4` in the implementation. We use the letter `c` to denote contexts $\Gamma$.

In order to sort things out, we present in the example below, the step by step derivation of the proposition $\varphi \Rightarrow \theta$ under the context $\varphi \Rightarrow \psi, \psi \Rightarrow \theta, [\,] =: \Gamma$ alongside a Coq formalized proof which holds rule applications in the exact same order.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\psi \Rightarrow \theta, [\,] \vdash \psi \Rightarrow \theta}\,(\mathrm{ax})}{\Gamma \vdash \psi \Rightarrow \theta}\,(\mathrm{wk})}{\varphi, \Gamma \vdash \psi \Rightarrow \theta}\,(\mathrm{wk}) \qquad \dfrac{\dfrac{}{\Gamma \vdash \varphi \Rightarrow \psi}\,(\mathrm{ax})}{\varphi, \Gamma \vdash \varphi \Rightarrow \psi}\,(\mathrm{wk}) \qquad \dfrac{}{\varphi, \Gamma \vdash \varphi}\,(\mathrm{ax})}{\dfrac{\varphi, \Gamma \vdash \psi}{\varphi, \Gamma \vdash \theta}\,(\Rightarrow\mathrm{E})}\,(\Rightarrow\mathrm{E})}{\Gamma \vdash \varphi \Rightarrow \theta}\,(\Rightarrow\mathrm{I})$$

```
Example IPLExample:
  ∀ (phi psi theta: IPL),
  IPLE ((impli phi psi) :: (impli psi
     theta) :: nil)
       (impli phi theta).
Proof. intros phi psi theta.
       apply ax6.
       apply ax9 with (p ≜ psi).
       - apply ax2.
         apply ax2.
         apply ax1.
       - apply ax9 with (p ≜ phi).
         + apply ax2.
           apply ax1.
         + apply ax1.
Qed.
```

To conclude that some proposition $\psi$ holds, provided some hypothesis $H$ of the form $\varphi \Rightarrow \psi$, the Coq tactic `apply` $H$ transforms the goal into the shape of $\varphi$, performing backwards reasoning. The tactic can

definitely do forwards reasoning. For instance, given a pair of hypotheses $H_1 : \varphi$ and $H_2 : \varphi \Rightarrow \psi$, `apply` $H_2$ `in` $H_1$ results in $H_1$ taking the shape of $\psi$. Also, the dash '`-`' and plus '`+`' signs are used to focus on the subgoals generated by the application of the rule `ax9`, namely the "$\Rightarrow$E".

### 3.3. Interpretation: IPL in a category induced by cartesian closed preorder

We interpret the logic IPL into the category $\mathcal{C}_P$ induced by any cartesian closed preorder $(P, \leqslant)$. The interpretation function $[\![ \cdot ]\!] : \text{IPL} \to \mathcal{C}_P$ is recursively defined as follows:

| IPL | | $\mathcal{C}_P$ | |
|---|---|---|---|
| $[\![\varphi]\!]$ | := | $x$ | some $x$ in obj $\mathcal{C}_\mathcal{P}$ |
| $[\![\text{true}]\!]$ | := | $\mathbb{1}$ | terminal object |
| $[\![\varphi \,\&\, \psi]\!]$ | := | $[\![\varphi]\!] \times [\![\psi]\!]$ | product object |
| $[\![\varphi \Rightarrow \psi]\!]$ | := | $[\![\psi]\!]^{[\![\varphi]\!]}$ | exponential object |
| $[\![[\,]]\!]$ | := | $\mathbb{1}$ | terminal object |
| $[\![\varphi, \Gamma]\!]$ | := | $[\![\varphi]\!] \times [\![\Gamma]\!]$ | product object |

In brief, any ground proposition has been modeled by a distinguished object of the category $\mathcal{C}_P$. The truth value has been interpreted by the terminal object. The conjunction is handled by categorical products while the implication is mapped to exponential objects. The empty context has been modeled by the terminal object while the nonempty context is represented by the binary products of individual interpretations of contained propositions.

```
Class interp (P: PreOrder) (h1: top P) (h2: hasPOBM P) (h3: hasPOHI P h2): Type ≜
{
  M : IPL → (@obj (@uc (PreOrderDetCatisCCC P h1 h2 h3)));
  i1: { p: (@obj (@uc (PreOrderDetCatisCCC P h1 h2 h3))) | M pi = p };
  i2: M truth = @tobj (@uc (PreOrderDetCatisCCC P h1 h2 h3))
                      (@hasT (@uc (PreOrderDetCatisCCC P h1 h2 h3))
                      (@uobs (PreOrderDetCatisCCC P h1 h2 h3)));
  i3: ∀ p q, M (conju p q) = @pobj (@uc (PreOrderDetCatisCCC P h1 h2 h3))
                                   (@hasP (@uc (PreOrderDetCatisCCC P h1 h2 h3))
                                   (@uobs (PreOrderDetCatisCCC P h1 h2 h3))) (M p) (M q);
  i4: ∀ p q, M (impli p q) = @eobj (@uc (PreOrderDetCatisCCC P h1 h2 h3))
                                   (@hasP (@uc (PreOrderDetCatisCCC P h1 h2 h3))
                                   (@uobs (PreOrderDetCatisCCC P h1 h2 h3)))
                                   (@hasE (@uc (PreOrderDetCatisCCC P h1 h2 h3))
                                   (@uobs (PreOrderDetCatisCCC P h1 h2 h3)))
                                   (M p) (M q);
  ML: ctx → (@obj (@uc (PreOrderDetCatisCCC P h1 h2 h3)));
  i5: ∀ (l: ctx), ML l = recI P h1 h2 h3 l M
}.
```

As shown above, the entire model has been developed as a Coq class named `interp` which embodies the interpretation function $[\![ \cdot ]\!] : \text{IPL} \to \mathcal{C}_P$ under the name `M`. It is defined from the `IPL` into the category determined by the preorder `P` which has the greatest element (`h1`), binary meets (`h2`) and Heyting implications (`h3`) given as parameters to the `interp` class. The function `M` is constructed by cases not based on the definitional equality of Coq but on the propositional Leibniz equality ranging from the fields `i1` to `i4`.

The field `i1` implements the ground propositions as objects of the category while `i2` maps the truth value

into the terminal object. Conjunctions and implications are respectively modeled by products and exponentials as part of the fields `i3` and `i4`.

The interpretation of contexts has been implemented in a similar fashion to that of terms. For which, we employ a function called `ML` defined from the `IPL` contexts to the objects of the underlying category. It performs along the same lines with the recursive function `recI` (modulo Leibniz equality of Coq given by the field `i5`) which is shown below.

```
Fixpoint recI (P: PreOrder) (h1: top P) (h2: hasPOBM P) (h3: hasPOHI P h2) (l: list IPL)
  (M: IPL → (@obj (@uc (PreOrderDetCatisCCC P h1 h2 h3)))):
  (@obj (@uc (PreOrderDetCatisCCC P h1 h2 h3))) ≜
  match l with
    | nil     ⇒ @tobj (@uc (PreOrderDetCatisCCC P h1 h2 h3))
                      (@hasT (@uc (PreOrderDetCatisCCC P h1 h2 h3))
                      (@uobs (PreOrderDetCatisCCC P h1 h2 h3)))
    | x :: xs ⇒ @pobj (@uc (PreOrderDetCatisCCC P h1 h2 h3))
                      (@hasP (@uc (PreOrderDetCatisCCC P h1 h2 h3))
                      (@uobs (PreOrderDetCatisCCC P h1 h2 h3)))
                      (M x) (recI P h1 h2 h3 xs M)
   end.
```

The function `recI` maps the empty context into the terminal object, and the nonempty context into the binary products of individual interpretations, by the function `M`, of the contained propositions. In order to zoom into the details of the entire interpretation, please refer to the `IPL.v` file in the library.

### 3.4. Soundness proof: a sketch

Now, we state a helper lemma first and then the soundness theorem for the IPL within the scope of the category induced by any cartesian closed preorder.

**Lemma 3.1 (helper)** *For all $(P,\leqslant)$, we have $x \leqslant y = true$ in $(P,\leqslant) \implies x \to y$ in $\mathcal{C}_P$.*

**Proof** Trivially follows from unfolding the definition of arrows in $\mathcal{C}_P$. □

**Theorem 3.2 (IPL soundness)** *Given*

- *$\Gamma, \varphi$, for all category $\mathcal{C}_P$ induced by cartesian closed pre-orders $(P,\leqslant)$*
- *all interpretations $[\![\cdot]\!]$ of the propositional identifiers as members of $\mathcal{C}_P$ (or $P$).*

*If $\Gamma \vdash \varphi$ is provable in IPL then the map $[\![\Gamma]\!] \to [\![\varphi]\!]$ can be defined in $\mathcal{C}_P$.*

**Proof** We first apply Lemma 3.1, and turn the goal into the following shape: $[\![\Gamma]\!] \leqslant [\![\varphi]\!] = \texttt{true}$ in $(P,\leqslant)$. We then argue by induction on the structure of the entailment relation $\Gamma \vdash \varphi$. Therefore, there are nine subgoals to close. As the rest is folklore, we suffice to focus on the case where the proposition $\varphi$ is a conjunction of a pair of propositions $p$ and $q$ just for demonstration and reinforcement purposes for the corresponding Coq proof.

| | |
|---|---|
| subgoal: $[\![\Gamma]\!] \leqslant [\![p \,\&\, q]\!] = \texttt{true}$ | ($\varphi := p \,\&\, q$, IH: $[\![\Gamma]\!] \leqslant [\![p]\!] = \texttt{true} \wedge [\![\Gamma]\!] \leqslant [\![q]\!] = \texttt{true}$) |
| $[\![\Gamma]\!] \leqslant [\![p]\!] = \texttt{true} \wedge [\![\Gamma]\!] \leqslant [\![q]\!] = \texttt{true}$ | (by the IH) |
| $[\![\Gamma]\!] \leqslant [\![p]\!] \diamond [\![q]\!] = \texttt{true}$ | (by the uniqueness of binary meets) |
| $[\![\Gamma]\!] \leqslant [\![p \,\&\, q]\!] = \texttt{true}$ | (by definition of the $[\![\cdot]\!]$) |

□

We carry on with the reflection of above given lemma and theorem statements and the corresponding portion of the soundness proof in Coq.

```
Lemma PO_POIC_1: ∀ (P: PreOrder) (x y: @pos P) (C ≜ PreOrderDetCat P),
  @pohrel P x y = true → @arrow C y x.
Proof. intros (P, R, r, t) x y C H.
       unfold PreOrderDetCat in C.
       ...
       unfold PreOrderICMap.
       ...
Qed.
```

In the lemma statement, `@arrow C y x` denotes the type (or simply collection) of maps defined from the object `x` into `y` in the category `C` induced by the preorder `P`. The proof proceeds with unfolding the definition of `C` and the arrows therein.

```
Lemma soundnessIPL_CCP: ∀ (x: IPL) (c: ctx) (P: PreOrder)
                          (h1: top P) (h2: hasPOBM P) (h3: hasPOHI P h2)
                          (i: interp P h1 h2 h3)
                          (C ≜ PreOrderDetCat P),
                          IPLE c x →
                          @arrow C ((@M P h1 h2 h3 i) x) ((@ML P h1 h2 h3 i) c).
Proof. intros x c P h1 h2 h3 i C H.
       apply PO_POIC_1.
       ...
       induction H.
       ...
       - (* case 5 *)
         ...
         destruct i as (M, i1, i2, i3, i4, ML, i5).
         ...
         rewrite i3.
         ...
         rewrite pobmuni.
         apply (conj IHIPLE1 IHIPLE2).
       ...
Qed.
```

The Coq proof of the main theorem statement starts with the application of the helper lemma `PO_POIC_1`. The goal at this stage looks like `pohrel (ML c) (M x) = true`. We then apply induction over the structure of `IPLE c x` introduced as an hypothesis under the name `H` into the proof context. As noted earlier, we have nine cases to show. For the sake of brevity, we focus on the case number five in which the propositional formula is an IPL conjunction. The corresponding subgoal in this state is `pohrel (ML c) (M (conju p q)) = true` for some propositional terms `p` and `q`. The tactic `destruct i` makes no change in the goal statement but projects the fields of the `interp` class instance `i` into individual hypotheses just as they are named in the class declaration. The tactic `rewrite i3` replaces the subterm `(M (conju p q))` with the binary meet of propositions `p` and `q`, that is `pobm`, and turns the goal into `pohrel (ML c) pobm = true`. Employing `rewrite pobmuni` reshapes the goal into `pohrel (ML c) (M p) = true ∧ pohrel (ML c) (M q) = true` by declaration (refer to the class `POBM` in Section 3.1). The current state of the goal is simply the conjunction of the induction hypotheses `IHIPLE1: pohrel (ML c) (M p) = true` and `IHIPLE2: pohrel (ML c) (M q) = true`, and closed by the tactic `apply`

`(conj IHIPLE1 IHIPLE2)`. The constructor `conj` here refers to propositional conjunction operator of Coq, and should not be confused with that of IPL. Note lastly that the intermediate tactic applications skipped by triple dots in the above demonstration are auxiliary, and employed to get the hypotheses or the goal into the correct shape so that the "main" tactic applications could be performed with no issue. For the entire formalization of the proof, please look into the `IPL.v` file.

## 4. Conclusion and future directions

We briefly exhibited a category theory library developed in the Coq proof assistant. The design principles of the library have been stated in a comparative fashion with the existing libraries out there. We concluded with a case study, in which the IPL soundness has been proven with respect to a certain category theoretical setting, to demonstrate the utility of the library.

Also, we stress out the mismatch in between contextual equality of category theory and the structural Leibniz equality of type theory, and brought forward that the former cannot be represented by the latter entirely. The problem implied by this mismatch is undecidable in general as it could be reduced to the problem of higher order unification. However, it could be solved for particular cases. So far, we managed to solve a case in formalizing a proof of the comparison theorem for the Kleisli construction. Even though the proof has not yet been entirely developed, we did overcome the mismatch in proving the same theorem for the Eilenberg–Moore construction. Completing this proof is in fact the next practical goal to be achieved which could be chased by a formalization of Beck's monadicity theorem. These proofs and related constructions could be viewed as some certain recipe when it comes to performing formal reasoning over imperative and impure programs. As for pure programs, we are planning to make an attempt at formalizing semantics of dependent products in a categorical setting for which many essentials have already been implemented such as the UP of adjunctions.

## References

[1] Byliński C. Introduction to categories and functors. Formalized Mathematics,1990: 1(2):409-420.

[2] Capriotti P. pcapriotti/agda-categories at GitHub.

[3] Coq Development Team. The Coq proof assistant reference manual, 2018. Version 8.8.1.

[4] Coquand T, Paulin C. Inductively defined types. In: Martin-Löf P, Mints G (editors). International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings, volume 417 of Lecture Notes in Computer Science, pages 50–66. Springer, 1988.

[5] Dowek G, Hardin T, Kirchner C. Higher order unification via explicit substitutions. Information and Computation, 2000; 157(1-2):183-235.

[6] Ekici B. ekiciburak/CatTheo at GitHub.

[7] Ekici B, Kaliszyk C. Mac Lane's comparison theorem for the Kleisli construction formalized in Coq. Mathematics in Computer Science, Feb 2020. ISSN 1661-8289.

[8] Elliott C. Compiling to categories. Proceedings of the ACM on Programming Languages, 1(ICFP):2017; 27: 1-27:27.

[9] Freire JL, Brañas EF, Blanco A. On recursive functions and well-founded relations in the calculus of constructions. In: Moreno-Díaz R, Pichler F, Quesada-Arencibia A (editors). Computer Aided Systems Theory - EUROCAST 2005, 10th International Conference on Computer Aided Systems Theory, Las Palmas de Gran Canaria, Spain, February 7-11, 2005, Revised Selected Papers, volume 3643 of Lecture Notes in Computer Science, pages 69-80. Springer, 2005.

[10] Gross J, Chlipala A, Spivak DI. Experience implementing a performant category-theory library in Coq. In: Klein G, Gamboa R (editors). Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings, volume 8558 of LNCS, pages 275–291. Springer, 2014.

[11] Huet GP. Cartesian closed categories and lambda-calculus. In: Cousineau G, Curien P, Robinet B (editors). Combinators and Functional Programming Languages, Thirteenth Spring School of the LITP, Val d'Ajol, France, May 6-10, 1985, Proceedings, volume 242 of Lecture Notes in Computer Science, pages 123–135. Springer, 1985.

[12] Ishii H. konn/category-agda at GitHub.

[13] Lambek J. From $\lambda$-calculus to cartesian closed categories. To HB Curry: essays on combinatory logic, lambda calculus and formalism, pages 1980: 375-402.

[14] Lambek J. Cartesian closed categories and typed lambda-calculi. In: Cousineau G, Curien P, Robinet B (editors). Combinators and Functional Programming Languages, Thirteenth Spring School of the LITP, Val d'Ajol, France, May 6-10, 1985, Proceedings, volume 242 of Lecture Notes in Computer Science, pages 136-175. Springer, 1985.

[15] Moggi E. Notions of computation and monads. Information and Computation, 93(1):55–92, July 1991. ISSN 0890-5401.

[16] Peebles D. copumpkin/categories at GitHub.

[17] Pouillard N. crypto-agda/crypto-agda at GitHub.

[18] Spies S, Forster Y. Undecidability of higher-order unification formalised in Coq. In: Blanchette J, Hritcu C (editors). Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020, pages 143-157. ACM, 2020.

[19] Timany A, Jacobs B. Category theory in Coq 8.5. In Proceedings of the 1st International Conference on Formal Structures for Computation and Deduction, Porto, Portugal, pages 2016; 30: 1-30.

[20] Voevodsky V, Ahrens B, Grayson D, et al. UniMath — a computer-checked library of univalent mathematics.

[21] Wadler P. Monads for functional programming. In: Jeuring J, Meijer E (editors). Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text, volume 925 of Lecture Notes in Computer Science, pages 24–52. Springer, 1995.

[22] Wiegley J. jwiegley/category-theory at GitHub.

## Appendices

### A. Terminals, products and exponentials

If exists, an object is said to be terminal in a category $\mathcal{C}$ if and only if there is a unique incoming arrow from every single $\mathcal{C}$-object. We develop terminal objects in Coq as follows:

```
Class Terminal (C: Category) (tobj: @obj C): Type ≜
{
  tmorph : ∀ X, @arrow C tobj X;
  tmorphu: ∀ X (f g: @arrow C tobj X), f = g
}.
```

This is simply read as: we have a terminal object `tobj` in an arbitrarily given category `C`, if there is an incoming arrow `tmorph` from every object `X` of `C` such that any alternative to such arrow is indeed itself ensuring the unicity. The proof obligation is stated by the field `tmorphu`.

In any category $\mathcal{C}$, if exists, binary product of a pair of objects $A, B$ is a quadruple $(P, \pi_1, \pi_2, p)$ with

$$\begin{array}{lcll} P & := & A \times B & \text{(product object)} \\ \pi_1 & : & P \to A & \text{(first projection)} \\ \pi_2 & : & P \to A & \text{(second projection)} \\ p & : & \forall (Z \text{ in } \mathcal{C})\,(f\colon Z \to A)\,(g\colon Z \to B),\, Z \to P & \text{(product map)} \end{array}$$

such that

$$\begin{array}{lcl} pob_1 & : & \forall (Z \text{ in } \mathcal{C})\,(f\colon Z \to A)\,(g\colon Z \to B),\, \pi_1 \circ (p\,Z\,f\,g) = f \\ pob_2 & : & \forall (Z \text{ in } \mathcal{C})\,(f\colon Z \to A)\,(g\colon Z \to B),\, \pi_2 \circ (p\,Z\,f\,g) = g \\ pobu & : & \forall (Z \text{ in } \mathcal{C})\,(f\colon Z \to A)\,(g\colon Z \to B)\,(q\colon Z \to P), \\ & & \pi_1 \circ q = f \wedge \pi_2 \circ q = g \implies (p\,Z\,f\,g) = q. \end{array}$$

The product $P$ is indeed the object of pairs handled by pairing up the members of $A$ and $B$. It comes with a couple of projections $\pi_1$ and $\pi_2$ along with product map $p$ defined from any object $Z$ into $P$ provided the arrows $f\colon Z \to A$ and $g\colon Z \to B$. The unique map $p$ gives the arrow $f$ back out of the first projection and $g$ out of the second one as respectively implied by the obligations $pobu$, $pob_1$ and $pob_2$. This construction is reflected in a Coq implementation as follows:

```
Class Product (C: Category) (A B prod: @obj C): Type ≜
{
  pi1       : @arrow C A prod;
  pi2       : @arrow C B prod;
  prod_f    : ∀ (Z: @obj C) (f: @arrow C A Z) (g: @arrow C B Z), @arrow C prod Z;
  prod_f_c1 : ∀ (Z: @obj C) (f: @arrow C A Z) (g: @arrow C B Z), f = pi1 o prod_f Z f g;
  prod_f_c2 : ∀ (Z: @obj C) (f: @arrow C A Z) (g: @arrow C B Z), g = pi2 o prod_f Z f g;
  prod_f_uni: ∀ (Z: @obj C) (f: @arrow C A Z) (g: @arrow C B Z) (q: @arrow C prod Z),
              f = pi1 o q → g = pi2 o q → (prod_f Z f g) = q
}
```

The product object $P$, of arbitrarily given objects $A$ and $B$, is named `prod`. The fields `pi1` and `pi2` implement the projections $\pi_1$ and $\pi_2$ respectively. The product map $p$ is denoted by the field `prod_f` while obligations $pob_1$ and $pob_2$ are developed by the fields called `prod_f_c1` and `prod_f_c2`. Obviously, the field `prod_f_uni` represents the uniqueness of the product map formalizing the obligation $pobu$.

In any category $\mathcal{C}$ with binary products, if exists, exponential of a pair of objects $X$ and $Y$ is a triple $(E, app, cur)$ with

$$\begin{array}{lcll} E & := & Y^X & \text{(exponential object)} \\ app & : & E \times X \to Y & \text{(application)} \\ cur & : & \forall (Z \text{ in } \mathcal{C})\,(f\colon Z \times X \to Y),\, Z \to E & \text{(currification)} \end{array}$$

such that

$$\begin{array}{lcl} curcomm & : & \forall (Z \text{ in } \mathcal{C})\,(f\colon Z \times X \to Y)\,, app \circ (cur\, Z\, f \times \mathrm{id}_X) = f \\ curuni & : & \forall (Z \text{ in } \mathcal{C})\,(f\colon Z \times X \to Y)\,(e\colon Z \to E), \\ & & app \circ (cur\, Z\, f \times \mathrm{id}_X) = e \implies (cur\, Z\, f) = e \end{array}$$

The exponential $E$ is in fact the object of maps defined from $X$ to $Y$ along with a pair of interface functions $app$ and $cur$. The former obviously applies a map $f\colon X \to Y$ in a member of $X$ returning a member of $Y$. The latter transforms a function of pairs into a function of functions. The proof obligation $currcomm$ indicates that first currifying a function $f\colon Z \times X \to Y$ and then applying it in a member of $X$, after arranging things into the correct shape, gives the arrow $f$ back as it is. While the obligation $curuni$ definitely clears up the fact that there must be a unique way of currifying arrows. One way to formalize this construction in the proof assistant Coq is listed as follows:

```
Class Exponential (C: Category) {p: hasProducts C} (X Y exp: @obj C): Type ≜
{
  app    : @arrow C Y (@pobj C p exp X);
  cur    : ∀ (Z: @obj C) (f: @arrow C Y (@pobj C p Z X)), @arrow C exp Z;
  curcomm: ∀ (Z: @obj C) (f: @arrow C Y (@pobj C p Z X)), app o (fprod (cur Z f) (@identity C X)) = f;
  curuni : ∀ (Z: @obj C) (f: @arrow C Y (@pobj C p Z X)) (g: @arrow C exp Z),
    app o (fprod g (@identity C X)) = f  →  (cur Z f) = g
}.
```

The exponential object $E$, of arbitrarily given objects $X$ and $Y$, is called `exp`. The fields `app` and `curr` implement interface functions $app$ and $curr$. while those `curcomm` and `curuni` are respectively formalizing proof obligations $currcomm$ and $curuni$.