A Trustful Monad for Axiomatic Reasoning with Probability and Nondeterminism

REYNALD AFFELDT, National Institute of Advanced Industrial Science and Technology, Japan
JACQUES GARRIGUE, Nagoya University, Japan
DAVID NOWAK, CNRS & Lille University, France
TAKAFUMI SAIKAWA, Nagoya University, Japan

The algebraic properties of the combination of probabilistic choice and nondeterministic choice have long been a research topic in program semantics. This paper explains a formalization (the first one to the best of our knowledge) in the Coq proof assistant of a monad equipped with both choices: the geometrically convex monad. This formalization has an immediate application: it provides a model for a monad that implements a non-trivial interface which allows for proofs by equational reasoning using probabilistic and nondeterministic effects. We explain the technical choices we made to go from the literature to a complete Coq formalization, from which we identify reusable theories about mathematical structures such as convex spaces and concrete categories.

Additional Key Words and Phrases: monad, formalization, probabilities, nondeterminism, convexity, categories

1 INTRODUCTION

In their ICFP paper "Just do It: Simple Monadic Equational Reasoning" [Gibbons and Hinze 2011], the authors present an axiomatic approach to reason about programs with effects using equational reasoning, thus recovering one of the appeals of pure functional programming. This approach uses monads to encapsulate the effects and is called *monadic equational reasoning*. In particular, to handle the effects of probability and nondeterminism, Gibbons and Hinze propose a combination of two interfaces: one for monads equipped with an operator for probabilistic choice and one for monads equipped with an operator for nondeterministic choice. It was later observed that in the proposed combination the authors "got [the algebraic properties that characterise their interaction] wrong" [Abou-Saleh et al. 2016]. The problem was that right-distributivity of bind over probabilistic choice is not compatible with distributivity of probabilistic choice over nondeterministic choice. Fortunately, the previous work in question [Gibbons and Hinze 2011] was not relying on this mistake. Yet, this stresses the need for a formal account of combinations of interfaces (representing algebraic theories) and the monads realizing them.

While many sets of axioms have been suggested as axiomatizations of the combination of probabilistic and nondeterministic choice, only few give rise to interesting models [Keimel and Plotkin 2017; Mislove et al. 2004]. We will stick here to Gibbons and Hinze's axiomatization, removing just the incriminated right-distributivity.

We can rely on a large body of work to model formally the combination of probabilistic and nondeterministic choice (e.g., [Beaulieu 2008; Cheung 2017; Gibbons 2012; Keimel and Plotkin 2017; Mislove 2000; Tix et al. 2009; Varacca and Winskel 2006], and much more if we consider concurrency). So what should be a monad modeling this axiomatization? Since we already have the finite

Authors' addresses: Reynald Affeldt, Cyber Physical Security Center, National Institute of Advanced Industrial Science and Technology, Japan, reynald.affeldt@aist.go.jp; Jacques Garrigue, Graduate School of Mathematics, Nagoya University, Japan, garrigue@math.nagoya-u.ac.jp; David Nowak, CRIStAL, CNRS & Lille University, France; Takafumi Saikawa, Graduate School of Mathematics, Nagoya University, Japan, tscompor@gmail.com.

2020. XXXX-XXXX/2020/3-ART $15.00
https://doi.org/

powerset monad and the finitely-supported distributions monad for these two choices, one could think of composing them using monadic distributive laws [Beck 1969]. Unfortunately, it has been proved that distributivity between these two monads is impossible [Varacca and Winskel 2006, Proposition 3.2]. We thus have to rethink the construction of a model by looking into what it should be more precisely. The presence of probabilistic choice suggests that sets of distributions might be a model, like it is the case with the probability monad. Yet, the semantics must also be convex-closed because if two distributions d_1 and d_2 are possible outcomes, so is any convex combination $pd_1 + (1-p)d_2$ $(0 \leq p \leq 1)$ of them [Gibbons 2012, Sect. 5.2]. Convexity is in particular necessary to have idempotence of probabilistic choice. Unfortunately these observations do not readily lead to a formalization, as they leave many technical details unsettled. In his PhD thesis, Cheung derives a monad (called the *geometrically convex monad*) for the theory resulting from the combination of the effects of probability and nondeterminism [Cheung 2017, Chapter 6]. His pencil-and-paper construction highlights the central role of several technical devices of independent interest: the notion of *convex spaces* [Fritz 2015; Jacobs 2010; Stone 1949] to formalize convexity without resorting to vector spaces and Beaulieu's operator for infinite nondeterministic choice [Beaulieu 2008, Def. 3.2.3].

But we should not forget that the original incentive for a formalization of a monad that combines probabilistic and nondeterministic choices is monadic equational reasoning. There actually exists a formalization in the Coq proof assistant [The Coq Development Team 2019a] of the many examples of Gibbons et al. in the form of a library called Monae [Affeldt et al. 2019]. It comes with concrete monads modeling several interfaces *except* the one that combines probabilistic and nondeterministic choices, because it is arguably more difficult than the others.

Contributions. In this paper, we provide a formal construction of the geometrically convex monad. To the best of our knowledge, this is the first formalization of a monad that supports probabilistic and nondeterministic choices. It has been carried out in the Coq proof assistant.

Our formalization of the geometrically convex monad has a direct application: it is used to complete an existing formalization of monadic equational reasoning. The latter was assuming the existence of a such monad without a model; our work improves the trusted base of this practical tool by filling this hole.

Our main contribution (the formalization of the geometrically convex monad) is made possible by a number of technical contributions of independent interest: a formalization of convex spaces (the first formalization to the best of our knowledge), a formalization of semicomplete semilattice structures, an original formalization of concrete categories, and extensions to an existing theory of finitely-supported probability distributions. We will discuss the technical decisions that made it possible to develop these formalizations in such a way that they all fit together to achieve a formalization of the geometrically convex monad.

Paper Outline. In Sect. 2, we clarify our formalization target by reviewing the formalization of monadic equational reasoning we aim at extending. We explain the operators of interest and their properties, and we give an overview of the construction of the geometrically convex monad. In Sect. 3, we give an overview of a formalization of convex spaces, an important ingredient of our construction to represent probabilistic choice, convex sets, hulls, and affine functions. In Sect. 4, we explain the formalization of semicomplete semilattice structures, which provide an operator to represent a nondeterministic choice compatible with the probabilistic choice. In Sect. 5, we explain a formalization of concrete categories to build monads out of adjoint functors. In Sect. 6, we define several adjunctions, from which we derive the geometrically convex monad through composition. Finally, in Sect. 7, we verify that the geometrically convex monad can be equipped

with the combined choice and that the latter enjoys the expected properties. We further comment on related work in Sect. 8 and conclude in Sect. 9.

About Notations. This paper displays Coq source code as it is, with some beautification to ease reading using LaTeX symbols instead of ASCII art. When there are too much details we omit parts of the source code and instead provide a paraphrase and indicate to the reader where to look in the formalization. Otherwise, we use standard mathematical notations, sometimes augmented to avoid too much overloading (for example, we note $f@(X)$ the direct image of the set X by f but $F\#(g)$ the application of the functor F to the morphism g).

2 FORMALIZATION TARGET AND APPROACH

Our goal is to construct a monad that combines probabilistic and nondeterministic choices. This corresponds to the monad intended by Gibbons et al. [Gibbons and Hinze 2011]. We here review a formalization in Coq of Gibbons et al.'s monads and their interfaces. Our formalization target is a monad of type `altProbMonad`.

2.1 An Existing Hierarchy of Probability-related Monads

Figure 1 provides an excerpt of an existing hierarchy of effects [Affeldt et al. 2019] that includes the ones by Gibbons et al. [Gibbons 2012; Gibbons and Hinze 2011] (amended as suggested by Abou-Saleh et al. [Abou-Saleh et al. 2016]).

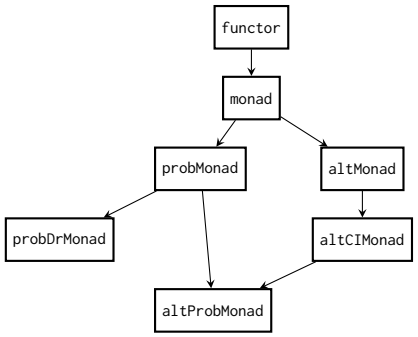

Fig. 1. Hierarchy of effects related to the monad type `altProbMonad` that combines nondeterministic and probabilistic choices.

We assume given two *types* `functor` and `monad` for endofunctors and monads on Coq's `Type` universe. The type `monad` is equipped with a join operator `Join` and a unit operator `Ret`. In this section we rather use the bind operator, defined as $m \gg\!= f$ = `Join`$(M\#(f)\, m)$ for the monad `M`. The precise definitions of `functor` and `monad` are not relevant at this stage but can be found in related work [Affeldt et al. 2019, Sect. 2.1].

Note that these so-called "types" are actually data-structures that provide the same functionality as type classes in Agda [The Agda Team 2020] or Idris [Brady 2013], i.e., providing an implementation for such a type amounts to defining an instance of the corresponding type class. Moreover, thanks to implicit coercions, this implementation itself can be used as a type, so that assuming `(M : monad)` allows one to write the type `M T` of computations resulting in a value of type `T` inside the monad `M`. The other nodes represent various monad types, that extend `monad` through the incremental additions of *mixins*, using the methodology of *packed classes* [Garillot et al. 2009].

We first extend the type `monad` into the type of the probability monad `probMonad`. The interface of `probMonad` takes the form of a mixin that introduces an operator for probabilistic choice $a \lhd p \rhd b$,

where a and b are computations and p is a *probability*, i.e., a real number p such that $0 \leq p \leq 1$. The intuition is that the computation $a \lhd p \rhd b$ represents the computation a with probability p or the computation b with probability $1-p$. The properties, or *axioms*, of the interface are identity axioms (lines 4 and 5), skewed commutativity (line 6), idempotence (line 7), quasi-associativity (line 8), and the fact that bind left-distributes over probabilistic choice (line 11) (see also [Affeldt et al. 2020, file `proba_monad.v`]).

```
1  Record mixin_of (M : monad) : Type := Mixin {
2    choice : ∀ (p : prob) T, M T → M T → M T
3             where "a ◁ p ▷ b" := (choice p a b) ;
4    _ : ∀ T (a b : M T), a ◁ 0%:pr ▷ b = b ;
5    _ : ∀ T (a b : M T), a ◁ 1%:pr ▷ b = a ;
6    _ : ∀ T p (a b : M T), a ◁ p ▷ b = b ◁ p.~%:pr ▷ a ;
7    _ : ∀ T p (a : M T), a ◁ p ▷ a = a ;
8    _ : ∀ T (p q r s : prob) (a b c : M T),
9        (p = r * s :> R ∧ s.~ = p.~ * q.~)%R →
10       a ◁ p ▷ (b ◁ q ▷ c) = (a ◁ r ▷ b) ◁ s ▷ c ;
11   _ : ∀ p A B (m1 m2 : M A) (k : A → M B),
12       (m1 ◁ p ▷ m2) >>= k = m1 >>= k ◁ p ▷ m2 >>= k}.
```

In Coq, the type `prob` is for probabilities. The notation `%:pr` turns a real number into a probability when possible. The notation p`.~` is for `1 - `p (often written $\overline{p}$ on paper). Skewed commutativity allows to derive one of the identity axioms from the other; here we are just preserving the original interface from Gibbons and Hinze [Gibbons and Hinze 2011].

The monad type `probDrMonad` extends `probMonad` with right-distributivity of bind over probabilistic choice. We do not display its implementation because we do not model this monad in this paper; we mention it for the sake of completeness.

The monad type `altMonad` introduces an operator `□` for nondeterministic choice[1]. Besides associativity of nondeterministic choice (line 15 below), it also states that bind left-distributes over nondeterministic choice (line 16) (see also [Affeldt et al. 2020, file `fail_monad.v`]), as specified by the following mixin:

```
13 Record mixin_of (M : monad) : Type := Mixin {
14   alt : ∀ T, M T → M T → M T where "a □ b" := (alt a b) ;
15   _ : ∀ T (x y z : M T), x □ (y □ z) = (x □ y) □ z ;
16   _ : ∀ A B (m1 m2 : M A) (k : A → M B),
17       (m1 □ m2) >>= k = m1 >>= k □ m2 >>= k }.
```

Gibbons and Hinze do not require right-distributivity (i.e., $m \gg\!= (\lambda x.\, k_1 x \,\square\, k_2 x) = (m \gg\!= k_1) \,\square\, (m \gg\!= k_2)$) by default, due in particular to undesirable interactions with non-idempotent effects [Gibbons and Hinze 2011, Sect. 4.2].

The monad type `altCIMonad` extends `altMonad` with commutativity and idempotence of nondeterministic choice, which we express by the following mixin, where `op x y` stands for `x □ y`.

```
Record mixin_of (M : Type → Type) (op : ∀ {T}, M T → M T → M T) : UU1 := Mixin {
  _ : ∀ T (x : M T),   op x x = x ;
  _ : ∀ T (x y : M T), op x y = op y x }.
```

[1]Gibbons and Hinze actually call "choice" and use the identifier `alt` for what we call nondeterministic choice; they call nondeterministic choice a combination of choice and failure [Gibbons and Hinze 2011, Sect. 4.3].

Finally, in the monad type `altProbMonad`, probabilistic choice distributes over nondeterministic choice, which we express by another mixin, where `op p x y` is intended to denote `x ◁ p ▷ y`.

```
Record mixin_of (M : altCIMonad) (op : prob → ∀ {T}, M T → M T → M T) := Mixin {
  _ : ∀ T p (x y z : M T), op p x (y □ z) = op p x y □ op p x z }.
```

Implementation of Inheritance Relations with Packed Classes. Up to now, we have only shown the mixin part of the inheritance hierarchy. The packed class methodology [Garillot et al. 2009] actually contains three ingredients: mixins, types (implemented as *structures*), and *classes*. Here are the class and structure definitions for `altProbMonad`.

```
23  Record class_of (m : Type → Type) := Class {
24    base : MonadAltCI.class_of m ;
25    mixin_prob : MonadProb.mixin_of (Monad.Pack (MonadAlt.base (MonadAltCI.base base))) ;
26    mixin_altProb : @mixin_of (MonadAltCI.Pack base) (@MonadProb.choice _ mixin_prob) }.
27  Structure altProbMonad : Type := Pack { m :> Type → Type ; class : class_of m }.
```

The class definition inherits from `altCIMonad` through its class (line 24), and extends it with two mixins, the one we have seen for `probMonad` (line 25), and the extra distributivity axiom we have just defined (line 26). The structure then packs together the type operator `m` which can be implicitly accessed through a coercion, with the above defined class. Finally, the triple mixin-class-structure is completed with extra coercions and unification hints (provided by the `Canonical` command [Mahboubi and Tassi 2013] of Coq) to achieve the inheritance relations depicted in Fig. 1.

Sample Programs. Last, let us reproduce for illustration sample programs by Gibbons [Gibbons 2012, Sect. 5.1] using the operators we have introduced so far in the syntax of Monae. Here is a biased coin, with probability p of returning `true` and probability $\overline{p}$ of returning `false`:

```
Definition bcoin {M : probMonad} (p : prob) : M bool :=
  Ret true ◁ p ▷ Ret false.
```

Here is an arbitrary nondeterministic choice between Booleans:

```
Definition arb {M : altMonad} : M bool := Ret true □ Ret false.
```

Using the do notation instead of the bind operator, these two programs can be used to make an arbitrary choice followed by a probabilistic choice:

```
Definition arbcoin p : M bool :=
  (do a ← arb ; (do c ← bcoin p; Ret (a == c) : M _))%Do.
```

or the other way round:

```
Definition coinarb p : M bool :=
  (do c ← bcoin p ; (do a ← arb; Ret (a == c) : M _))%Do.
```

Monadic equational reasoning is about proving the properties of such programs mostly by rewriting using the axioms of the various interfaces proposed by Gibbons et al. See the various related work for sample proofs [Gibbons 2012; Gibbons and Hinze 2011; Mu 2019a,b] and their mechanization [Affeldt et al. 2019].

2.2 Alternative Axiomatizations

As we mentioned in the introduction, the axiomatization of combined choice we have followed is not the only possible one. We will consider shortly two other possible axiomatizations for which non-trival models are known.

The first one is obtained by replacing the distributivity axiom added in `altProbMonad` by the dual one, i.e., distributivity of nondeterministic choice over probabilistic choice.

```
x □ (y ◁ p ▷ z) = (x ◁ p ▷ y) □ (x ◁ p ▷ z)
```

Keimel et al. [Keimel and Plotkin 2017] have shown that in this case probabilities different from 0 and 1 become undistinguishable (i.e., `x ◁ p ▷ y = x ◁ q ▷ y` for any $0 < p, q < 1$). This means that the algebraic theory of combined choice in this case boils down to a bi-semilattice (two semilattices with their operators mutually distributing over each other), which can be modeled by finite sets of finite sets, easily turned into a monad. While the monad construction is easy, the structure is poor, as all probability information is lost, so we did not try to formalize this axiomatization. This also means that this axiomatization is equivalent to the one with both distributivity axioms, as the original axiom can be derived from the dual one. Another way to reach the same axiomatization is to inherit from `probDrMonad` rather than `probMonad` [Abou-Saleh et al. 2016, Sect. 3]. It appears that, while left-distributivity of bind over probabilistic choice is fine alone, right-distributivity can be used to deduce the distributivity of nondeterministic choice over probabilistic choice from its dual, which leads to the same collapse of probability information as above.

The second one is obtained by keeping the same distributivity axiom as in `altProdMonad`, but removing the idempotence of probabilistic choice from `probMonad`, i.e., we lose the equality `x ◁ p ▷ x = x`. Varacca [Varacca and Winskel 2006] has shown that this relaxed `probMonad` can be modeled by a monad of real quasi-cones, which distributes over the finite powerset monad modeling `altCIMonad`. As a result, one can use Beck's construction [Beck 1969] to create a monad combining both. While this is a clever approach, the loss of the idempotence axiom is dire, and Varacca presents in the same paper another construction using a convex powerset functor to obtain a model including the idempotence axiom, in a way similar to the geometrically convex powerdomain [Mislove 2000; Tix et al. 2009].

Ultimately, the only way to be sure that our choice of axioms follows our expectations, is to provide a model where we can check that different computations can be properly distinguished (which we will do with the geometrically convex monad in Sect. 7.2).

2.3 Formalization of the Geometrically Convex Monad: Overview

As already hinted at in the introduction (Sect. 1), a computation using the monadic operations defined in the type `altProbMonad` can be modeled by a non-empty convex set of finitely-supported probability distributions. Cheung provides a construction for such a monad and calls the resulting monad the geometrically convex monad [Cheung 2017, Chapter 6]. It is built by composition of adjunctions.

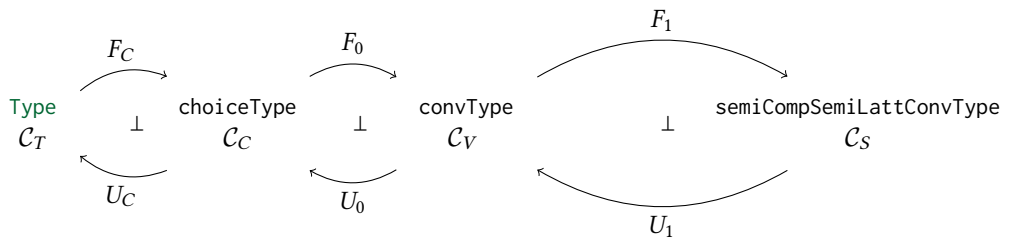

Fig. 2. Adjunctions between the categories involved in the construction of the geometrically convex monad

Figure 2 depicts four concrete categories related by three adjunctions. Each category is named after a Coq type to which it corresponds. The category $\mathcal{C}_T$ corresponds to Coq's type `Type`. The

latter actually represents a countably infinite hierarchy of types $\texttt{Type}_0$, $\texttt{Type}_1$, ... such that $\texttt{Type}_i$ is a subtype of $\texttt{Type}_{i+1}$. By default, Coq hides the indices to the user. We can regard Type as a category by seeing each $\texttt{Type}_i$ as a Grothendieck universe [Timany and Jacobs 2016]. The category $\mathcal{C}_C$ corresponds to types with a choice function. The type choiceType [Garillot et al. 2009, Sect. 3.1] comes from the Mathematical Components library (hereafter, MathComp) [Mathematical Components Team 2007]. The category $\mathcal{C}_V$ corresponds to spaces with a convexity operator (for probabilistic choice) and the category $\mathcal{C}_S$ corresponds to spaces with a convexity operator and an infinitary operator (for nondeterministic choice); the details of these two categories are one of the purposes of this paper. The three adjunctions are composed of six functors. The unit and counit of $F_C \dashv U_C$ are η_C and ε_C respectively (resp. η_0, ε_0 for $F_0 \dashv U_0$ and η_1, ε_1 for $F_1 \dashv U_1$). In particular, U_C, U_0, and U_1 are forgetful functors, which makes F_C, F_0, and F_1 free functors.

Our setting features three adjunctions while Cheung's has only two. The additional adjunction is the one between Type and choiceType. It comes from the fact that the formalization of monadic equational reasoning we build upon [Affeldt et al. 2019] represents monads as endofunctors over Type, whereas our construction requires types to be equipped with a choice function[2]. In practice, the functor F_C only amounts to adding a choice function to the type, without changing the values. Note that, since we assume the existence of such a choice function for all types, we are actually adding the axiom of choice to the ambient logic, which is known to be sound in Coq [The Coq Development Team 2019b]. It is simpler to assume a well known axiom than to try to define all our monads on choiceType, and prove that all the types we use can actually be equipped with a concrete choice function.

The basis of the construction of the geometrically convex monad is easy to explain: by composing adjunctions, we build the adjunction

$$P_\Delta^{\text{left}} = F_1 \circ F_0 \circ F_C \dashv U_C \circ U_0 \circ U_1 = P_\Delta^{\text{right}}.$$

Then we can derive the monad $P_\Delta = P_\Delta^{\text{right}} \circ P_\Delta^{\text{left}}$ directly from this adjunction [Mac Lane 1998]. Yet, the formalization is difficult mostly for two reasons. First, it requires several non-trivial mathematical theories, which, to the best of our knowledge, could not be found in a single formal setting, in particular because some of them (e.g., convex spaces) have never been formalized before. Second, Cheung's construction could not be reproduced directly and we had to come up with some tuning to model nondeterminism (switching from a binary operator to an infinitary one) or to formalize categories (using concrete categories as a solution).

3 CONVEXITY TOOLBOX

A convex space is an algebraic structure allowing convex combinations of its elements by an operator satisfying several equational axioms. This axiom system can be used to define various related concepts: probabilistic choice, convex sets, convex hulls, affine functions, etc. More precisely, for our application, convex sets are used to represent computations in a monad modeling altProbMonad and convex hulls are used to represent nondeterminism.

In this section, we provide a formal definition of convex space (a.k.a. barycentric calculus [Stone 1949]) as a basis to represent the category $\mathcal{C}_V$ (and later the category $\mathcal{C}_S$) (both categories have been seen in Sect. 2.3) and we also provide definitions and lemmas to define in particular the functor F_1. The most relevant file of the accompanying development for this section is [Affeldt et al. 2020, file convex_choice.v].

[2]Actually, these choice functions are not used in our development itself, but choiceTypes are required due to our use of the Finmap library [Cohen and Sakaguchi 2015] (which builds upon MathComp). See Sect. 6.1 for more details.

3.1 Formalization of Convex Spaces

The interface of convex spaces is similar to the interface of the `probMonad` we saw in Sect. 2.1. It provides a similar operator $a \lhd p \rhd b$ where a and b are elements of the convex space and p is a probability. The axioms about the operator are similar to the ones already explained in Sect. 2.1 (the reader can observe a difference of presentation for the axiom of quasi-associativity but it is not relevant). Of course, contrary to `probMonad`, convex spaces have no axiom about a bind operator.

```
Record mixin_of (T : choiceType) : Type := Class {
  conv : prob → T → T → T where "a ◁ p ▷ b" := (conv p a b);
  _ : ∀ a b, a ◁ 1%:pr ▷ b = a ;
  _ : ∀ p a, a ◁ p ▷ a = a ;
  _ : ∀ p a b, a ◁ p ▷ b = b ◁ p.~%:pr ▷ a;
  _ : ∀ (p q : prob) (a b c : T),
      a ◁ p ▷ (b ◁ q ▷ c) = (a ◁ [r_of p, q] ▷ b) ◁ [s_of p, q] ▷ c }.
```

The notation `[s_of p, q]` stands for $\overline{\bar{p}\bar{q}}$; the notation `[r_of p, q]` stands for $p \,/\, \overline{\bar{p}\bar{q}}$. Here we assume the carrier type of convex spaces to be a `choiceType`.

The above mixin is used to define the type `convType` using the packed classes methodology (that we briefly overviewed in Sect. 2.1).

We can show for example that the real numbers form a convex space by taking the averaging function $\lambda p\, x\, y.\, px + \bar{p}y$ to be the operator. Similarly, finitely-supported probability distributions form a convex space with the operator $\lambda p\, d_1\, d_2.\, pd_1 + \bar{p}d_2$ where d_1 and d_2 are distributions.

We will later need a generalization of the binary operator $a \lhd p \rhd b$ to n points, namely $\langle\rhd_d f$, where f consists of n points and d is a distribution of n probabilities whose sum is 1.

3.2 Convex Sets and Convex Hulls

We use convex spaces to define *convex sets* and *convex hulls*. From now on, we put ourselves in a classical setting, by extending the logic of Coq with a number of axioms known to be compatible with it. Concretely, we use the axioms provided by MathComp-Analysis, an extension of MathComp for classical analysis [Affeldt et al. 2018]. In this setting, `Prop` and `bool` are equivalent (strong excluded middle), and we can freely embed `Prop`-valued formulas such as `∀ x, P x` into `bool` using a notation: `` `[<∀ x, P x>] : bool ``. From MathComp-Analysis, we also reuse a library of sets. Here sets mean sets of elements of a specific type. They are represented by `Prop`-valued characteristic functions, and thus not necessarily finite. The type `set A` stands for sets over the type `A`.

A set D is convex when any convex combination of any two points is still inside D:

```
Variable A : convType.
Definition is_convex_set (D : set A) : bool :=
  `[<∀ x y t, D x → D y → D (x ◁ t ▷ y)>].
```

The hull of a set X is the set of points p such that p is the convex combination of points belonging to X. The notation `[set p : T | P p]` is for sets defined by comprehension.

```
Definition hull (T : convType) (X : set T) : set T :=
  [set p : T | ∃ n (g : 'I_n → T) d, g @` setT ⊆ X ∧ p = <|>_d g].
```

We represent the n points to be combined as $g_0, g_1, \ldots$, hence the function `g : 'I_n → T` from `'I_n`, the MathComp type of natural numbers smaller than `n`. The notation `g @` setT` is for the direct image `g@(setT)` where `setT` is the full set (these are part of the library of sets that comes with the MathComp-Analysis library).

3.3 Affine Functions

We are interested in *affine functions* because they are used for the morphisms of the categories $\mathcal{C}_V$ and $\mathcal{C}_S$ (Sect. 2.3). In real analysis, affine functions correspond to the functions of the form $x \mapsto ax + b$. But the real line is just an example of convex space, whose generic operator actually provides an easy, generic definition:

```
Variables (A B : convType).
Definition affine_function_at (f : A → B) a b t :=
  f (a ◁ t ▷ b) = f a ◁ t ▷ f b.
```

If we endow convex spaces with an order, we can also define both convex and concave functions, and alternatively define affine functions as functions that are both convex and concave.

As a sample proposition, we can observe that convex hulls are preserved by affine functions:

```
Proposition image_preserves_convex_hull (f : {affine A → B}) (Z : set A) :
  f @` (hull Z) = hull (f @` Z).
```

This property will be used to define the functor F_1, whose action on morphisms defined by the direct image needs to preserve convex hulls.

4 SEMICOMPLETE SEMILATTICE STRUCTURES

In this section, we define generic structures that provide an operator to represent nondeterministic choice in a way that is compatible with probabilistic choice. The most relevant file in the accompanying development is [Affeldt et al. 2020, file `necset.v`].

As a prerequisite, we introduce the type of non-empty sets. The type `neset T` is the type of sets over T that have at least one element. As a convenience, this type comes with a postfix notation `%:ne` such that `s%:ne` is the non-empty set corresponding to the set `s`. This notation infers the proof of non-emptiness in several situations such as when `s` is a singleton set, the image of a non-empty set, the union of non-empty sets, etc. using Coq's canonical structures [Mahboubi and Tassi 2013].

4.1 Semicomplete Semilattice

The first structure we introduce provides a unary operator `op` that turns a non-empty set of elements into a single element (line 53). The first axiom of this structure says that this operator applied to a singleton set returns the sole element of the set (line 54). The second axiom starting at line 55 collapses a non-empty collection (the indexing set itself is not empty) of non-empty sets (`f` is the collection) into one element:

```
52  Record mixin_of (T : choiceType) : Type := Mixin {
53    op : neset T → T ;
54    _ : ∀ x : T, op [set x]%:ne = x ;
55    _ : ∀ I (s : neset I) (f : I → neset T),
56          op (\bigcup_(i in s) f i)%:ne = op (op @` (f @` s))%:ne }.
```

The theory defined by this mixin is similar to Beaulieu's theory for infinite nondeterministic choice [Beaulieu 2008, Def. 3.2.3]. The difference is that the right-hand side of the second axiom in Beaulieu's work is expressed by means of a partition of the indexing set. We prefer to avoid dealing with partitions because in our experience they are heavy to deal with in formal proofs.

Hereafter we denote by $\bigsqcup$ the operator introduced by the above mixin and use the mixin to define the type `semiCompSemiLatt` of *semicomplete semilattices* [Bergman 2015, p. 185].

4.2 Combining Semicomplete Semilattice with Convex Space

We now extend the structure of semicomplete semilattices from the previous section (Sect. 4.1) with an axiom that captures the interaction between the operator $\bigsqcup$ and probabilistic choice. This interaction is akin to a distribution law that can be stated informally as follows:

$$x \lhd p \rhd \bigsqcup I = \bigsqcup((\lambda y.\, x \lhd p \rhd y)@(I))$$

Formally, this axiom is provided as a mixin parameterized by a semicomplete semilattice and a ternary operator `op` indexed by a probability:

```
Record mixin_of (L : semiCompSemiLattType) (op : prob → L → L → L) := Mixin {
  _ : ∀ (p : prob) (x : L) (I : neset L),
    op p x (⨆ I) = ⨆ ((op p x) @` I)%:ne }.
```

We use this mixin to extend the type of semicomplete semilattices to the type of *semicomplete semilattice convex spaces* (`semiCompSemiLattConvType` in Coq scripts) that inherits both the properties of semicomplete semilattices (Sect. 4.1) and the properties of convex spaces (Sect. 3.1). The methodology to achieve this multiple inheritance is again the one of packed classes.

We conclude this section with a sample property of the operator $\bigsqcup$ that is both important and non-trivial:

```
Variable L : semiCompSemiLattConvType.
Lemma lub_op_hull (X : neset L) : ⨆ (hull X)%:ne = ⨆ X.
```

The proof is as follows. First, we lift the operator of convex spaces $(\lhd p \rhd)$ from points to sets of points; we denote this lifted operator by $(:\lhd p \rhd:)$. We use this lifted operator to define a new binary operator $X :\square: Y := \bigcup_{p\in[0,1]} X :\lhd p \rhd: Y$. Second, we show that $\mathtt{hull}\, X = \bigcup_{i\in\mathbb{N}} \underbrace{X :\square: X :\square: \cdots :\square: X}_{i+1 \text{ occurrences of } X}$. Then, we show that $\bigsqcup(X) = \bigsqcup(X :\square: X :\square: \cdots :\square: X)$, using the property introduced by semicomplete semilattice convex spaces. Finally, we conclude the proof by appealing to the properties of semicomplete semilattices.

We will later provide a concrete example of use of the lemma `lub_op_hull`. It can also be used to establish technical results from Beaulieu's work (e.g., [Beaulieu 2008, p. 56, l. 3]) or similar ones as in Varacca and Winskel's work (e.g., [Varacca and Winskel 2006, Lemma 5.6]).

4.3 Instances with Non-empty Convex Sets

The definitions of semicomplete semilattices and of semicomplete semilattice convex spaces that we have provided in the previous sections are just interfaces. To instantiate them, it turns out that it suffices to use non-empty convex sets instead of mere non-empty sets. This is this instance that we will use in particular to produce the adjunction $F_1 \dashv U_1$.

Thus we start by extending the type `neset` of non-empty sets into the type `necset` of non-empty convex sets, using the definition from the Sect. 3.2 (and again the methodology of packed classes).

We then instantiate the semicomplete semilattice operator on nonempty convex sets using union and hull operators (`necset A` is the type of nonempty convex sets over a convex space `A`):

$$\begin{array}{rcl} \bigsqcup & : & \mathtt{neset\ (necset\ A)} \to \mathtt{necset\ A} \\ & & X \mapsto \mathtt{hull}\,(\bigcup_{x\in X} x) \end{array}$$

This gives us in particular the type `necset_semiCompSemiLattConvType A`: a generic instance of `semiCompSemiLattConvType` where the carrier consists of non-empty convex sets over `A`, a `convType`.

5 FORMALIZATION OF CONCRETE CATEGORIES IN COQ

The purpose of this section is to provide a formalization of enough category theory to construct the geometrically convex monad. The interest of this formalization is that it comes as a conservative extension of related work (Monae [Affeldt et al. 2019]) and that it features an original use of concrete categories to achieve a shallow embedding of categories in the Coq proof assistant. The most relevant file in the accompanying development is [Affeldt et al. 2020, file `category.v`].

5.1 Formalization of Categories

5.1.1 Definition of Concrete Categories. As we saw in Sect. 2.3, we need to formalize several categories; this is in contrast with Monae, which could get along with the sole category of sets. Among the various possibilities, we chose to favor a definition (akin to a shallow embedding) that lets us use the typing relation of Coq to declare elements of an object and apply morphisms to them as if morphisms were ordinary Coq functions. Our solution is to represent categories with a *universe à la Tarski*, i.e., a type with an interpretation operation, or *realizer*, allowing us to regard terms of this type as types (the function `el` below at line 63). In this setting, we can look at the morphisms of a category through the realizer and identify morphisms as a subset of the function space (via the predicate defining the hom-set at line 64).

```
62  Record mixin_of (obj : Type) : Type := Mixin {
63    el : obj → Type ; (* "interpretation" operation, "realizer" *)
64    hom : ∀ A B, (el A → el B) → Prop ; (* subset of morphisms *)
65    _ : ∀ A, @hom A A id ; (* id is in hom *)
66    _ : ∀ A B C (f : el A → el B) (g : el B → el C),
67        hom f → hom g → hom (g \o f) (* hom is closed under composition *) }.
```

In the code above, the modifier `@` is a Coq feature to disable implicit arguments.

The advantage of this definition is twofold. First, the parameter `obj` lets us choose how we index our objects and use this index to declare morphisms (e.g., `f : el A → el B`). Second, we can use morphisms as functions and apply them to elements, as illustrated by the following script:

```
Variable C : category.
Variable a b : C.
Variable x : el a.
Variable f : {hom a, b}.
Check f x : el b.
```

The result is by no way ad hoc: it actually corresponds to the definition of *concrete categories*. A category $\mathcal{C}$ is said to be concrete if it comes with a faithful functor from $\mathcal{C}$ to **Set**, that is, a functor whose action on each hom-set is injective. The indexing type `obj` and the realizer `el` together form the object part. The morphism part is directly represented by its images in **Set**, without specifying its source for which we did not find a use.

5.1.2 Categories to Build the Geometrically Convex Monad. In this section, we instantiate our definition of concrete categories with the categories that were described in Sect. 2.3.

The Categories $\mathcal{C}_T$ and $\mathcal{C}_C$. To define the category $\mathcal{C}_T$, we take the identity function as the realizer and the third argument of `@Category.Mixin` to be the true predicate `fun _ _ _ ⇒ True`, so that the faithful functor for the concrete category is full (i.e., surjective on hom-sets):

```
Definition Type_category_mixin : Category.mixin_of Type :=
  @Category.Mixin Type (fun x : Type ⇒ x) (fun _ _ _ ⇒ True)
    (fun⇒ I) (fun _ _ _ _ _ _ _ ⇒ I).
```

```
Canonical Type_category := Category.Pack Type_category_mixin.
```

The command `Canonical` (that we already mentioned for its use in the packed classes methodology) provides a unification hint to Coq's type-checker to automatically endow `Type` with a structure of category when needed.

To define the category $\mathcal{C}_C$, we take the function `(fun x : choiceType ⇒ Choice.sort x)`, that returns the carrier type (in `Type`) of its argument (we make `Choice.sort` appear explicitly here but it is actually an implicit coercion in Coq). As for $\mathcal{C}_T$, the faithful functor is full:

```
Definition choiceType_category_mixin : Category.mixin_of choiceType :=
  @Category.Mixin choiceType (fun x : choiceType ⇒ Choice.sort x)
    (fun _ _ _ ⇒ True) (fun⇒ I) (fun _ _ _ _ _ _ _ ⇒ I).
```

The Category of Convex Spaces $\mathcal{C}_V$. The objects are convex spaces (Sect. 3.1) and the morphisms are affine functions (between convex spaces). In our formalization, the objects are indexed by the type of convex spaces `convType`, and realized by its coercion into `Type`. The morphisms are affine functions between convex spaces. Contrary to the previous two examples, being affine is not just a true predicate and requires us to prove that the identity function over a convex space is affine (proof `affine_function_id_proof`) and that the composition of affine functions is affine (proof `affine_function_comp_proof'`):

```
Definition convType_category_mixin : Category.mixin_of convType :=
  @Category.Mixin convType (fun A : convType ⇒ A) AffineFunction.axiom
    affine_function_id_proof affine_function_comp_proof'.
Canonical convType_category := Category.Pack convType_category_mixin.
```

The Category of Semicomplete Semilattice Convex Spaces $\mathcal{C}_S$. The objects are semicomplete semilattice convex spaces (Sect. 4.2) and the morphisms are affine functions f such that $f@(\bigsqcup X) = \bigsqcup(f@(X))$ for any non-empty convex set X. We can show that identity functions are such functions (proof `lub_op_affine_id_proof`) and that composition preserves these properties (proof `lub_op_affine_co`) leading to the following definition of $\mathcal{C}_S$:

```
Definition semiCompSemiLattConvType_category_mixin :
    Category.mixin_of semiCompSemiLattConvType :=
  @Category.Mixin semiCompSemiLattConvType (fun U : semiCompSemiLattConvType ⇒ U)
  LubOpAffine.class_of lub_op_affine_id_proof lub_op_affine_comp_proof.
Canonical semiCompSemiLattConvType_category :=
  Category.Pack semiCompSemiLattConvType_category_mixin.
```

5.2 Formalization of Functors, Natural Transformations, and Monads

Equipped with the formalization of categories from the previous section, we now formalize functors, natural transformations, and monads. In the following, `C` and `D` are two categories.

We encode a functor from `C` to `D` as an action on objects represented by a function `m : C → D` (line 91 below) and an action on morphisms represented by a function `f : ∀ A B, {hom A, B} → {hom m A, m B}` (line 92) equipped with proofs that `f` preserves the identity (line 93) and composition (line 94):

```
90  (* Module Functor. *)
91  Record mixin_of (C D : category) (m : C → D) : Type := Mixin {
92    f : ∀ (A B : Type), {hom A, B} → {hom m A, m B} ;
93    _ : FunctorLaws.id f ;
```

```
94     _ : FunctorLaws.comp f }.
```

By way of comparison, functors in Monae [Affeldt et al. 2019] were specialized to the category **Set** of sets and functions (the type `Type` of Coq is interpreted as the category **Set**):

```
Record mixin_of (m : Type → Type) : Type := Class {
  f : ∀ (A B : Type), (A → B) → m A → m B ;
  _ : FunctorLaws.id f ;
  _ : FunctorLaws.comp f }.
```

It is clear that the new, more general setting introduced above improves on this specialized setting because it makes it possible to talk about morphisms that are, e.g., affine functions. Hereafter, we denote by `F # g` the application of a functor `F` to a morphism `g`.

Let `F` and `G` be two functors from `C` to `D`. We encode a natural transformation from `F` to `G` as a family of maps `f : ∀ a, {hom F a ,G a}` (hereafter, denoted by `F ~> G`) such the `naturality` predicate holds:

```
Variables (F G : functor C D).
Definition naturality (f : F ~> G) := ∀ a b (h : {hom a, b}),
    (G # h) \o (f a) = (f b) \o (F # h).
```

When `F ~> G` is packed together with a proof of naturality, we have a genuine natural transformation that we denote by `F ~~> G`.

Finally, we define a monad as an endofunctor `M` equipped with two natural transformations: `ret` from the identify functor (denoted by `FId`) to `M`, and `join` from the composition of `M` with itself (denoted by `M ∘ M`) to `M`. The proofs of naturality appear at lines 106 and 107. These two natural transformations furthermore satisfy three coherence conditions (lines 108, 109, and 110):

```
102  (* Module Monad. *)
103  Record mixin_of (C : category) (M : functor C C) : Type := Mixin {
104    ret : ∀ A, {hom A, M A} ;
105    join : ∀ A, {hom M (M A), M A} ;
106    _ : naturality FId M ret ;
107    _ : naturality (M ∘ M) M join ;
108    _ : ∀ A, join A \o ret (M A) = id ;
109    _ : ∀ A, join A \o M # ret A = id ;
110    _ : ∀ A, join A \o M # join A = join A \o join (M A) }.
```

We already said that our formalization of functors generalizes the one of Monae, the formal framework for monadic equational reasoning on which our work is based. Similarly, our formalization of monads generalizes the one of Monae in a conservative way. Concretely, we provide a function `Monad_of_category_monad` that given a monad (as defined just above) over the category $\mathcal{C}_T$, returns a monad as defined in Monae (over the category **Set**). This way, it will be possible to retrofit our formalization of the geometrically convex monad (the topic of the next section) back into Monae.

5.3 Formalization of Adjoint Functors

We use adjoint functors to build the geometrically convex monad. In this section, we recall the lemmas used for this construction and give a brief overview of their formalization. We do not provide all the technical details because these lemmas are well-known lemmas and their formalization follows naturally from the definitions we saw so far (and whose design decisions are more important).

5.3.1 Definition of Adjunction. Two functors F and G are *adjoint* (denoted by $F \dashv G$) when there are two natural transformations $\eta : 1 \rightsquigarrow G \circ F$ and $\varepsilon : F \circ G \rightsquigarrow 1$ such that η and ε satisfy the triangular laws $\forall c.\, \varepsilon(F\,c) \circ F\#(\eta\,c) = id$ (triangular left) and $\forall d.\, G\#(\varepsilon\,d) \circ \eta(G\,d) = id$ (triangular right).

In Coq, we provide the notation `F ⊣ G` for the following type (where the categories `C` and `D` are implicit arguments):

```
AdjointFunctors.t : ∀ C D : category, functor C D → functor D C → Type
```

To build an adjunction, one needs to provides two natural transformations `eta` and `eps` together with the proofs that they satisfy the triangular laws. The corresponding constructors has the following type (where all arguments expect the proofs of the triangular laws are implicit):

```
AdjointFunctors.mk : ∀ (C D : category) (F : functor C D) (G : functor D C)
  (eta : FId ⇝ G ∘ F) (eps : F ∘ G ⇝ FId),
  TriangularLaws.left eta eps → TriangularLaws.right eta eps → F ⊣ G
```

5.3.2 Composition of Adjunction. It is well-known that two adjunction $F \dashv G$ (with unit/counit η/ε) and $F' \dashv G'$ (with unit/counit η'/ε') can be composed to form another adjunction $F' \circ F \dashv G \circ G'$ by taking the unit to be $\lambda A.\, G\#(\eta'(F_A)) \circ \eta_A$ and the counit to be $\lambda A.\, \varepsilon'_A \circ F'\#(\varepsilon(G'_A))$. Using the constructs we have defined so far, we provide a Coq function that performs this composition:

```
adj_comp : ∀ (C0 C1 C2 : category)
  (F : functor C0 C1)  (G : functor C1 C0),  F ⊣ G →
∀ (F' : functor C1 C2) (G' : functor C2 C1), F' ⊣ G' →
  F' ∘ F ⊣ G ∘ G'
```

5.3.3 Monad Defined by Adjointness. It is well-known that an adjunction $F \dashv G$ gives rise to a monad $G \circ F$ by taking η to be the unit and $\lambda A.\, G\#(\varepsilon(F_A))$ to be the join operator. In our formalization, this construction takes the form the following function:

```
Monad_of_adjoint : ∀ (C D : category) (F : functor C D) (G : functor D C),
  F ⊣ G → monad C
```

Observe that contrary to Monae where all monads are over the category **Set**, here our monad is over some category `C` which appears explicitly in the type.

6 ADJOINT FUNCTORS FOR THE GEOMETRICALLY CONVEX MONAD

At this point, we have explained the formalization of all the elements necessary to construct the geometrically convex monad: convex spaces in Sect. 3, semicomplete semilattice structures in Sect. 4, and category theory in Sect. 5. In this section, we explain the formalization of the adjunctions explained in Sect. 2.3. The most relevant file from the accompanying development is [Affeldt et al. 2020, file `gcm_model.v`].

6.1 The Adjunction $F_C \dashv U_C$

The existence of the adjunction $F_C \dashv U_C$ in our formalization is technical: it comes from the use of the Coq type `Type` in Monae and the need to use a `choiceType` in the definition of finitely-supported distributions.

Let us first define the functor F_C from $\mathcal{C}_T$ to $\mathcal{C}_C$. The action on objects consists in turning a type in `Type` into a `choiceType`. This is performed by the function `choice_of_Type` which relies on an axiom inherited from a MathComp library and whose validity is explained elsewhere [Affeldt et al.

2018, Sect. 5.2]. The action on morphisms turns a morphism $f : T \to U$ into the same morphism but with type choice_of_Type $T \to$ choice_of_Type U:

```
Definition hom_choiceType (A B : choiceType) (f : A → B) : {hom A, B} :=
  Hom (I : hom (f : el A → el B)).
Definition free_choiceType_mor (T U : C_T) (f : {hom T, U}) :
  {hom choice_of_Type T, choice_of_Type U} :=
    hom_choiceType (f : choice_of_Type T → choice_of_Type U).
```

The purpose of the function hom_choiceType is to turn a Coq function between two choiceTypes into a morphism of the category $\mathcal{C}_C$. I from the Coq standard library is a trivial proof that f is indeed a morphism; it is sufficient here because in this category all functions are morphisms (see the accompanying development [Affeldt et al. 2020] for the notation Hom). The functor laws are trivially proved and together with the definitions above, this leads to the definition of the functor free_choiceType of type functor $\mathcal{C}_T$ $\mathcal{C}_C$.

The definition of the corresponding forgetful functor U_C is similar. The main difference is that instead of using the function choice_of_Type to augment a type in Type, we use the coercion Choice.sort that retrieved the carrier type of a choiceType (see forget_choiceType in [Affeldt et al. 2020, file gcm_model.v]).

The unit $\eta_C : 1 \rightsquigarrow U_C \circ F_C$ and the counit $\varepsilon_c : F_C \circ U_C \rightsquigarrow 1$ are also essentially identity functions and the proofs of the triangular laws are therefore trivial.

6.2 The Adjunction $F_0 \dashv U_0$

The second adjunction $F_0 \dashv U_0$ corresponds to the probability monad [Giry 1982]. It relies on an existing formalization of finitely-supported distributions [Affeldt et al. 2019, Sect. 6.2] that we recall briefly. In the definition of FSDist.t below, the first field (line 128) is a finitely-supported function f from the choiceType A to the type of real numbers from the standard Coq library; this function evaluates to 0 outside its support finsupp f. The second (anonymous) field (line 129) contains proofs that (1) the probability function outputs positive reals and that (2) its outputs sum to 1.

```
126  (* Module FSDist. *)
127  Record t := mk {
128    f :> {fsfun A → R with 0} ;
129    _ : all (fun x ⇒ 0 < f x) (finsupp f) ∧ \sum_(a ← finsupp f) f a == 1 } .
```

It is important to observe that FSDist.t has type choiceType → choiceType and can therefore be used to build an endofunctor and a monad on top of it. Hereafter, {dist A} is a notation for FSDist.t A.

6.2.1 Functors. The action on morphisms of F_0 is the map of the probability monad associated with finitely-supported distributions. Indeed, let $\cdot \lhd \cdot \rhd \cdot$ be the operation of the convex space of finitely-supported distributions (see Sect. 3.1) and let $\gg\!=$ be the bind operator of the probability monad. We have $(d_1 \lhd p \rhd d_2) \gg\!= f = (d_1 \gg\!= f) \lhd p \rhd (d_2 \gg\!= f)$, which is equivalent to the map of the probability monad being affine.

In Coq, we define the action on morphisms of F_0 as follows, where FSDistfmap is the map operation of the probability monad:

```
Definition free_convType_mor (A B : choiceType) (f : {hom A, B}) :
  {hom FSDist_convType A, FSDist_convType B}.
refine (@Hom.Pack C_V _ _ _ (FSDistfmap f) _).
(* property that FSDistfmap is affine *)
```

```
Defined.
```

The type `FSDist_convType A` is the type of convex spaces of finitely-supported distributions over `A`.

We can show that `free_convType_mor` satisfies the functor laws (proofs `free_convType_mor_id` and `free_convType_mor_comp`), leading to the definition of the functor F_0 (recall the definitions of Sect. 5.2):

```
Definition free_convType : functor C_C C_V :=
  Functor.Pack (Functor.Mixin free_convType_mor_id free_convType_mor_comp).
```

The constructors `Functor.Mixin` and `Functor.Pack` are respectively for the mixin and the type of functors explained in Sect. 5.2.

The forgetful functor U_0 of type `functor` $\mathcal{C}_V$ $\mathcal{C}_C$ is just formalized by substituting the category $\mathcal{C}_V$ by the category $\mathcal{C}_C$ in morphisms (see `forget_convType` in [Affeldt et al. 2020, file `gcm_model.v`]).

6.2.2 Counit / unit. The counit is the natural transformation $\varepsilon_0 : F_0 \circ U_0 \rightsquigarrow 1_{\mathcal{C}_V}$ essentially defined by the following function:

$$\begin{aligned} \varepsilon_0 : \quad & \{\mathtt{dist}\, C\} \to C \\ & d \mapsto \langle\!\rangle_d\, \mathtt{finsupp}\,(d)\,. \end{aligned}$$

In this definition, C is a `convType`; the operation $\langle\!\rangle_{\_} \cdot$ has been explained in Sect. 3.1. Intuitively, ε_0 corresponds to the computation of a barycenter.

The unit is the natural transformation $\eta_0 : 1_{\mathcal{C}_C} \rightsquigarrow U_0 \circ F_0$ defined by the point-supported distribution `FSDist1.d`:

$$\begin{aligned} \eta_0 : \quad & C \to \{\mathtt{dist}\, C\} \\ & x \mapsto \mathtt{FSDist1.d}\ x. \end{aligned}$$

The proofs of the triangular laws required us to substantially enrich the theory of finitely-supported distributions of Monae. The reason can be understood by looking at the proof of the left triangular law `triL0`. The latter essentially amounts to prove that we have for any probability distribution d:

$$\langle\!\rangle_{\mathtt{FSDistfmap\ FSDist1.d}\ d}\, \mathtt{finsupp}\,(\mathtt{FSDistfmap\ FSDist1.d}\ d) = d.$$

One can observe that this statement involves distributions of distributions

```
Check FSDistfmap (@FSDist1.d C) d : {dist {dist C}}.
```

whose properties called for new lemmas. Comparatively, the proof of the right triangular law `triR0` is simpler.

6.3 The Adjunction $F_1 \dashv U_1$

6.3.1 Functors. The action on objects of F_1 is `necset_semiCompSemiLattConvType`, explained in Sect. 4.3. The action on morphisms of F_1 is defined by the direct image $f \mapsto \lambda X.\, f@(X)$ (where X is non-empty convex set):

```
Variables (A B : convType) (f : {hom A , B}).
Definition free_semiCompSemiLattConvType_mor'
    (X : necset_convType A) : necset_convType B :=
  ... (* definition using the direct image omitted *) ...
```

We can show that the image of a morphism is still a morphism: it is affine and preserves $\bigsqcup$ (because convex hulls are preserved by taking the direct image along affine functions—Sect. 3.3):

```
Definition free_semiCompSemiLattConvType_mor :
  {hom necset_semiCompSemiLattConvType A, necset_semiCompSemiLattConvType B} :=
  locked (@Hom.Pack C_S _ _ _ free_semiCompSemiLattConvType_mor'
    (LubOpAffine.Class free_semiCompSemiLattConvType_mor'_affine
                       free_semiCompSemiLattConvType_mor'_lub_op_morph)).
```

Finally, we show that the action on morphisms satisfy the functor laws, leading to the following definition of F_1:

```
Definition free_semiCompSemiLattConvType : functor C_V C_S :=
  Functor.Pack (Functor.Mixin free_semiCompSemiLattConvType_mor_id
                              free_semiCompSemiLattConvType_mor_comp).
```

Like for the adjunction $F_0 \dashv U_0$, the forgetful functor U_1 of type `functor C_S C_V` is just formalized by substituting the category $\mathcal{C}_S$ by the category $\mathcal{C}_V$ in morphisms (see `forget_semiCompSemiLattConvType` in [Affeldt et al. 2020, file `gcm_model.v`]).

6.3.2 Counit / unit. Let us explain how we implement the counit $\varepsilon_1 : F_1 \circ U_1 \rightsquigarrow 1_{\mathcal{C}_S}$. It is exactly the $\bigsqcup$ operator seen in Sect. 4.3:

$$\begin{aligned} \varepsilon_1 : \quad & \mathtt{neset}(\mathtt{necset}\, T) \to \mathtt{necset}\, T \\ & X \mapsto \bigsqcup X. \end{aligned}$$

We need to show that it is natural, that it preserves the operator $\bigsqcup$, i.e., $\varepsilon_1(\bigsqcup(X)) = \bigsqcup(\varepsilon_1 @(X))$ (for that purpose we use the lemma `lub_op_hull` from Sect. 4.2), and that it is affine, i.e., $\varepsilon_1(X \lhd p \rhd Y) = \varepsilon_1 X \lhd p \rhd \varepsilon_1 Y$.

The unit $\eta_1 : 1_{\mathcal{C}_V} \rightsquigarrow U_1 \circ F_1$ is the singleton map, which is easily shown to be natural and affine.

$$\begin{aligned} \eta_1 : \quad & \mathtt{necset}\, T \to \mathtt{neset}(\mathtt{necset}\, T) \\ & X \mapsto \{T\} \end{aligned}$$

We call the corresponding triangular laws `triL1` and `triR1`.

6.4 Putting it All Together

6.4.1 Formalization of the Geometrically Convex Monad. We use the proofs of the triangular laws of Sections 6.1, 6.2.2, and 6.3.2 to create the three adjunctions $F_C \dashv U_C$, $F_0 \dashv U_0$, and $F_1 \dashv U_1$:

```
Definition AC := AdjointFunctors.mk triLC triRC.
Definition A0 := AdjointFunctors.mk triL0 triR0.
Definition A1 := AdjointFunctors.mk triL1 triR1.
```

The definition of these adjunctions has been given in Sect. 5.3.1.

We then build the adjunction resulting from the composition of the three adjunctions we have just defined, using the function of Sect. 5.3.2:

```
Definition Agcm := adj_comp AC (adj_comp A0 A1).
```

Finally, we obtain the geometrically convex monad from the resulting adjunction using the generic lemma explained at the end of Sect. 5.3.3:

```
Definition Mgcm := Monad_of_adjoint Agcm.
```

The very last step is to use the function `Monad_of_category_monad` of Sect. 5.2 to recover a monad compatible with the Monae formal framework of monadic equational reasoning:

```
Definition gcm := Monad_of_category_monad Mgcm.
```

6.4.2 Informal Description of the Join of the Monad. At this stage, it is worth taking a step back to check that the join of the monad we have built indeed corresponds to the intuition one can have of the execution of a program mixing probabilistic choice and nondeterministic choice. Provided we ignore the function ε_C (the counit of the adjunction $F_C \dashv U_C$, which, as we already explained in Sect. 6.1, is here essentially for technical reasons), the join operator can informally be explained as the following function:

$$\varepsilon_1 \circ (\lambda X.\, \varepsilon_0 @ (X)).$$

The input of this function is indeed `necset {dist (necset {dist T})}`, i.e., it takes non-empty sets of distributions. The function ε_0 (Sect. 6.2.2) computes barycenters, so that when applied the left-hand side of the function composition returns an object of type `necset (necset {dist T})`. The function ε_1 (Sect. 6.3.2) computes the hull of the union of its input, which results in an object of type `necset (dist T)`, as expected.

7 THE PROPERTIES OF COMBINED CHOICE OF THE GCM

The very last step of our construction is to show that the geometrically convex monad (that we obtained as a result of the previous section—Sect. 6) satisfies the expected distributivity axioms that we discussed in Sect. 2.1 *and* to check that it is meaningful, i.e., that it really distinguishes the different choice operators. This corresponds to [Affeldt et al. 2020, file `altprob_model.v`] in the accompanying development.

7.1 The Geometrically Convex Monad has the Properties of Combined Choice

First, we start by defining nondeterministic choice for the geometrically convex monad using a binary version of the operator $\bigsqcup$ of Sect. 4.1:

```
Definition alt A (x y : gcm A) : gcm A := x ⊔ y.
```

We construct a monad `gcmA` implementing `altMonad` by proving the following properties, which are essentially consequences of the properties of the operator $\bigsqcup$:

```
Lemma altA A : associative (@alt A).
Lemma bindaltDl : BindLaws.left_distributive (@monad.Bind gcm) alt.
Definition gcmA : altMonad := MonadAlt.Pack ...
```

We extend the monad `gcmA` to the monad `gcmACI` that implements `altCIMonad`:

```
Lemma altxx A : idempotent (@Alt gcmA A).
Lemma altC A : commutative (@Alt gcmA A).
Definition gcmACI : altCIMonad := MonadAltCI.Pack ...
```

Second, we go on defining probabilistic choice for the geometrically convex monad using the operator of convex spaces:

```
Definition choice p A (x y : gcm A) : gcm A := x ◁ p ▷ y.
```

Most properties are direct consequences of the properties of convex spaces, and they lead to the definition of the monad `gcmp` that implemented `probMonad`:

```
Lemma choice0 A (x y : gcm A) : x ◁ 0%:pr ▷ y = y.
Lemma choice1 A (x y : gcm A) : x ◁ 1%:pr ▷ y = x.
Lemma choiceC A p (x y : gcm A) : x ◁p▷ y = y ◁p.~%:pr▷ x.
Lemma choicemm A p : idempotent (@choice p A).
Lemma choiceA A (p q r s : prob) (x y z : gcm A) :
  p = (r * s) :> R ∧ s.~ = (p.~ * q.~)%R →
  x ◁ p ▷ (y ◁ q ▷ z) = (x ◁ r ▷ y) ◁ s ▷ z.
```

```
Definition gcmp : probMonad := MonadProb.Pack ...
```

Finally, we prove left-distributivity of bind over the probabilistic choice and right-distributivity of the probabilistic choice over the nondeterministic choice

```
Lemma bindchoiceDl p : BindLaws.left_distributive (@monad.Bind gcm) (@choice p)
Lemma choicealtDr A (p : prob) :
  right_distributive (fun x y : gcmACI A ⇒ x ◁ p ▷ y) Alt.
```

and use these lemmas to instantiate `atlProbMonad` into the monad `gcmAP`:

```
Definition gcmAP : altProbMonad := MonadAltProb.Pack ...
```

This completes the construction of the monad proposed by Gibbons et al. [Abou-Saleh et al. 2016; Gibbons and Hinze 2011].

7.2 The Combined Choice is not a Trivial Theory

We conclude this section with a formal check that probabilistic choice in our axiom system of combined choice is not trivial, meaning that it indeed distinguishes different probabilities. It is sufficient to check that there exists a model which is not trivial in this sense, and our construction of geometrically convex monad serves this purpose nicely:

```
Example gcmAP_choice_nontrivial (p q : prob) :
  p ≠ q →
  Ret true ◁p▷ Ret false ≠ Ret true ◁q▷ Ret false :> gcmAP bool.
Proof.
...
Qed.
```

Here `:> gcmAP bool` indicates the type of this inequality, which forces the resolution of monadic operations inside our instance of `altProbMonad`. The proof just requires to unfold definitions and provides further evidence that the geometrically convex monad is not a trivial model.

8 RELATED WORK

We have already commented on several related work throughout this paper. We add in this section further comments that are better explained after the presentation of our contributions.

In our formalization of semicomplete semilattices (in Sect. 4), the nondeterministic choice is modeled as an infinitary operator. This is similar to Beaulieu's "infinite nondeterministic choice" [Beaulieu 2008, Def. 3.2.3] and, at first sight, looks different from Cheung's approach, who models nondeterministic choice as a binary operator [Cheung 2017, Sec. 6.3.1]. However, we observe that Cheung also implicitly uses an infinitary version of his operator (e.g., [Cheung 2017, p. 160]).

There is a number of formalizations of category theory in proof assistants (many of which being listed by Gross et al. [Gross et al. 2014]). However, we could not find a readily usable formalization of concrete categories in Coq. For example, UniMath is a large Coq library that aims at formalizing mathematics using the univalent point of view [Voevodsky et al. 2014]. It contains a substantial formalization of abstract categories but does not seem to feature a formalization of concrete categories. Since we needed only a handful of theorems about category theory, we formalized concrete categories from scratch and developed their theories as a generalization of Monae (in Sect. 5).

The idea of using categories as a package to handle functions with proofs was already presented by McBride [McBride 1999, Chapter 7, Section 3.1]. He also proposed the use of concrete categories for such a lightweight use of category theory, noting that the convertibility of terms is an easier way than propositional equality to handle the equational laws for morphisms, such as unit and

associativity laws. His formal definition of categories differs from ours in that it is also indexing on hom-sets, while in our definition, hom-sets are embedded as predicates. This difference further affects later definitions such as functors. Our definition makes it clearer that concrete categories are shallow embeddings of categories.

9 CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a formalization in the Coq proof assistant of the geometrically convex monad, a monad that combines probabilistic and nondeterministic choice. To the best of our knowledge, this is the first formalization of such a monad. A direct application is to complete an existing formalization of monadic equational reasoning which was lacking the model of the combined interface of probabilistic and nondeterministic choices. Our development of this formal model also led us to develop several mathematical theories of broader interest such as a formalization of convex spaces and a formalization of concrete categories. We can now use our model to check the non-triviality of Gibbons et al.'s choice of axioms.

Our experiment is an example of combination of two monads that requires a substantial amount of work. Although they could not be used here, there also exist a number of generic results about the combination of monads such as monad transformers [Jaskelioff 2009] or distributive laws [Zwart and Marsden 2018] that could also deserve formalization. By introducing a formalization of concrete categories to support the construction of the geometrically convex monad, our work also raises the question of the generalization of Monae from its specialization to the **Set** category.

ACKNOWLEDGMENTS

We acknowledge the support of the JSPS-CNRS bilateral program "FoRmal tools for IoT sEcurity" (PRC2199) and the JSPS KAKENHI Grant Number 18H03204, and thank all the participants of these projects for fruitful discussions. We also thank Cyril Cohen and Shinya Katsumata for guidance about the formalization of monads, and Kazunari Tanaka who contributed to the formalization of categories.

REFERENCES

Faris Abou-Saleh, Kwok-Ho Cheung, and Jeremy Gibbons. 2016. Reasoning about Probability and Nondeterminism. In *POPL workshop on Probabilistic Programming Semantics*. http://pps2016.soic.indiana.edu/2015/12/11/reasoning-about-probability-and-nondeterminism/

Reynald Affeldt, Cyril Cohen, and Damien Rouhling. 2018. Formalization Techniques for Asymptotic Reasoning in Classical Analysis. *J. Formalized Reasoning* 11, 1 (2018), 43–76. https://doi.org/10.6092/issn.1972-5787/8124

Reynald Affeldt, Jacques Garrigue, David Nowak, and Takafumi Saikawa. 2020. A Trustful Monad for Axiomatic Reasoning with Probability and Nondeterminism. See https://github.com/affeldt-aist/infotheo and https://github.com/affeldt-aist/monae. Coq scripts.

Reynald Affeldt, David Nowak, and Takafumi Saikawa. 2019. A Hierarchy of Monadic Effects for Program Verification Using Equational Reasoning. In *13th International Conference on Mathematics of Program Construction (MPC 2019), Porto, Portugal, October 7–9, 2019 (Lecture Notes in Computer Science)*, Graham Hutton (Ed.), Vol. 11825. Springer, 226–254. https://doi.org/10.1007/978-3-030-33636-3_9

Guy Beaulieu. 2008. *Probabilistic Completion of Nondeterministic Models*. Ph.D. Dissertation. Faculty of Graduate and Postdoctoral Studies, University of Ottawa.

John Beck. 1969. Distributive laws. In *Seminar on Triples and Categorical Homology Theory (Lecture Notes in Mathematics)*, Eckmann B. (Ed.). Sringer, 119–140.

George Bergman. 2015. *An Invitation to General Algebra and Universal Constructions*. https://doi.org/10.1007/978-3-319-11478-1

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

Kwok-He Cheung. 2017. *Distributive Interaction of Algebraic Effects*. Ph.D. Dissertation. Merton College, University of Oxford.

Cyril Cohen and Kazuhiko Sakaguchi. 2015. math-comp/finmap: Finite sets, finite maps, multisets and order. Available at https://github.com/math-comp/finmap. Last stable release: 1.4.0 (2019).

Tobias Fritz. 2015. Convex Spaces I: Definition and Examples. (2015). Available at https://arxiv.org/abs/0903.5522. First version: 2009.

François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. 2009. Packaging Mathematical Structures. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009), Munich, Germany, August 17–20, 2009 (Lecture Notes in Computer Science)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer, 327–342. https://doi.org/10.1007/978-3-642-03359-9_23

Jeremy Gibbons. 2012. Unifying Theories of Programming with Monads. In *Revised Selected Papers of the 4th International Symposium on Unifying Theories of Programming (UTP 2012), Paris, France, August 27–28, 2012 (Lecture Notes in Computer Science)*, Burkhart Wolff, Marie-Claude Gaudel, and Abderrahmane Feliachi (Eds.), Vol. 7681. Springer, 23–67. https://doi.org/10.1007/978-3-642-35705-3_2

Jeremy Gibbons and Ralf Hinze. 2011. Just do it: simple monadic equational reasoning. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19–21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 2–14. https://doi.org/10.1145/2034773.2034777

Michèle Giry. 1982. A categorical approach to probability theory. In *Categorical aspects of topology and analysis (Lecture Notes in Mathematics)*, B. Banaschewski (Ed.), Vol. 915. Springer, 68–85.

Jason Gross, Adam Chlipala, and David I. Spivak. 2014. Experience Implementing a Performant Category-Theory Library in Coq. In *Interactive Theorem Proving*, Gerwin Klein and Ruben Gamboa (Eds.). Springer International Publishing, Cham, 275–291.

Bart Jacobs. 2010. Convexity, Duality and Effects. In *IFIP TCS (IFIP Advances in Information and Communication Technology)*, Vol. 323. Springer, 1–19.

Mauro Jaskelioff. 2009. Modular Monad Transformers. In *18th European Symposium on Programming (ESOP 2009), York, UK, March 22–29, 2009. Proceedings (Lecture Notes in Computer Science)*, Vol. 5502. Springer, 64–79.

Klaus Keimel and Gordon D. Plotkin. 2017. Mixed powerdomains for probability and nondeterminism. *Logical Methods in Computer Science* 13, 1:2 (2017), 1–84.

Saunders Mac Lane. 1998. *Categories for the Working Mathematician* (second ed.). Graduate Texts in Mathematics, Vol. 5. Springer-Verlag, Berlin and New York.

Assia Mahboubi and Enrico Tassi. 2013. Canonical Structures for the Working Coq User. In *4th International Conference on Interactive Theorem Proving (ITP 2013), Rennes, France, July 22–26, 2013 (Lecture Notes in Computer Science)*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.), Vol. 7998. Springer, 19–34. https://doi.org/10.1007/978-3-642-39634-2_5

Mathematical Components Team. 2007. Mathematical Components Library. https://github.com/math-comp/math-comp. Development version. Last stable version 1.10 (2019) available on the same website.

Conor McBride. 1999. *Dependently Typed Functional Programs and their Proofs*. Ph.D. Dissertation. University of Edinburgh.

Michael Mislove, JoÃńl Ouaknine, and James Worrell. 2004. Axioms for Probability and Nondeterminism. *Electronic Notes in Theoretical Computer Science* 96 (2004), 7 – 28. https://doi.org/10.1016/j.entcs.2004.04.019 Proceedings of the 10th International Workshop on Expressiveness in Concurrency.

Michael W. Mislove. 2000. Nondeterminism and probabilistic choice: Obeying the laws. In *11th International Conference on Concurrency Theory (CONCUR 2000), University Park, PA, USA, August 22–25, 2000 (Lecture Notes in Computer Science)*, Catuscia Palamidessi (Ed.), Vol. 1877. Springer, 350–364.

Shin-Cheng Mu. 2019a. *Calculating a Backtracking Algorithm: An Exercise in Monadic Program Derivation.* Technical Report TR-IIS-19-003. Institute of Information Science, Academia Sinica.

Shin-Cheng Mu. 2019b. *Equational Reasoning for Non-deterministic Monad: A Case study of Spark Aggregation.* Technical Report TR-IIS-19-002. Institute of Information Science, Academia Sinica.

Marshall Harvey Stone. 1949. Postulates for the barycentric calculus. *Annali di Matematica Pura ed Applicata* 29 (1949), 25–30.

The Agda Team. 2020. *The Agda User Manual.* Available at https://agda.readthedocs.io/en/v2.6.0.1. Version 2.6.0.1.

The Coq Development Team. 2019a. *The Coq Proof Assistant Reference Manual.* Inria. Available at https://coq.inria.fr. Version 8.10.2.

The Coq Development Team. 2019b. The Logic of Coq. Available at https://github.com/coq/coq/wiki/The-Logic-of-Coq. Part of the Coq FAQ.

Amin Timany and Bart Jacobs. 2016. Category Theory in Coq 8.5. In *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016), June 22–26, 2016, Porto, Portugal (LIPIcs)*, Delia Kesner and Brigitte Pientka (Eds.), Vol. 52. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:18.

https://doi.org/10.4230/LIPIcs.FSCD.2016.30

Regina Tix, Klaus Keimel, and Gordon D. Plotkin. 2009. Semantic Domains for Combining Probability and Non-Determinism. *Electron. Notes Theor. Comput. Sci.* 222 (2009), 3–99. https://doi.org/10.1016/j.entcs.2009.01.002

Daniele Varacca and Glynn Winskel. 2006. Distributing probability over nondeterminism. *Mathematical Structures in Compuer Science* 16, 1 (2006), 87–113.

Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. 2014. UniMath — a computer-checked library of univalent mathematics. available at https://github.com/UniMath/UniMath. https://github.com/UniMath/UniMath Last stable release 0.1 (2016).

Maaike Zwart and Dan Marsden. 2018. Don't Try This at Home: No-Go Theorems for Distributive Laws. arXiv:1811.06460v2. Version 2.