JFP **31**, e17, 38 pages, 2021.
doi:10.1017/S0956796821000137

A trustful monad for axiomatic reasoning with probability and nondeterminism

REYNALD AFFELDT

National Institute of Advanced Industrial Science and Technology, Digital Architecture Research Center, Tokyo, Japan
(*e-mail:* reynald.affeldt@aist.go.jp)

JACQUES GARRIGUE

Nagoya University, Graduate School of Mathematics, Nagoya, Japan
(*e-mail:* garrigue@math.nagoya-u.ac.jp)

DAVID NOWAK

Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

TAKAFUMI SAIKAWA

Nagoya University, Graduate School of Mathematics, Nagoya, Japan
(*e-mail:* tscompor@gmail.com)

Abstract

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 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 nontrivial 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, and that we integrate in a framework for monadic equational reasoning.

1 Introduction

In their ICFP paper "Just do It: Simple Monadic Equational Reasoning" (Gibbons & Hinze, 2011), the authors present an axiomatic approach to equational reasoning about programs with effects, thus recovering one of the appeals of pure functional programming. This approach uses monads to encapsulate the effects, hence the name *monadic equational reasoning*. In particular, to handle the effects of probability and nondeterminism, Gibbons and Hinze propose a combination of two theories: 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 combined with distributivity of probabilistic choice over nondeterministic choice resulted in a degenerate theory. Fortunately, the previous work in question (Gibbons & Hinze, 2011) was not relying on this mistake.

The example above shows that there is a need for a formal account of the non-degeneracy of such a theory. One way to achieve it is to construct in a proof assistant a monad realizing the theory, which is, in our case, the combination of theories of probabilistic and nondeterministic choices. Monadic equational reasoning is not the only motivation to provide a formalized monad. Indeed, such a monad could be used to give semantics to programs mixing probabilities and nondeterminism (e.g., (Kaminski *et al.*, 2016)). The infrastructure needed to formalize such a monad could be used to formalize further foundational results in the blooming area of semantics combining probabilistic and nondeterministic choices (e.g., (Bonchi *et al.*, 2020a; Goy & Petrisan, 2020; Mio & Vignudelli, 2020)).

In previous work (Affeldt *et al.*, 2019), we used the Coq proof assistant (The Coq Development Team, 2021) to both define *interfaces* (i.e., sets of operators and axioms suitable for equational reasoning) and provide models for a wide array of simpler monads, including nondeterminism and probabilistic choice taken alone.

In this paper, we provide a Coq framework with which we formalize a monad with an interface representing the combined algebraic theory of probabilistic and nondeterministic choices; we moreover verify the axiomatization of this theory and illustrate it with examples. 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 (Mislove *et al.*, 2004; Keimel & Plotkin, 2017). We will stick here to Gibbons and Hinze's axiomatization, removing just the incriminated right distributivity. This gives us a trustful monad to reproduce Gibbons and Hinze's examples of monadic equational reasoning.

To formally model the combination of probabilistic and nondeterministic choices, we can rely on a large body of work (e.g., (Mislove, 2000; Varacca & Winskel, 2006; Beaulieu, 2008; Tix *et al.*, 2009; Gibbons, 2012; Keimel & Plotkin, 2017; Cheung, 2017)), even larger if we consider concurrency too. So what should be a monad modeling this axiomatization? Since we already have the finite powerset monad and the finitely supported distributions monad for these two choices, one could think of composing them. At first sight, monadic distributive laws (Beck, 1969) look like a candidate approach but unfortunately it has been proved that distributivity between these two monads is impossible (Varacca & Winskel, 2006, Proposition 3.2). A very recent result (Goy & Petrisan, 2020) indicates that weak distributive laws provide a solution to this composition problem. A more direct approach is to rethink the construction of a model of the intended monad 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 (Gibbons, 2012, Section 5.2). The answer in the literature appears to be non-empty convex sets of probability distributions. Convexity is in particular necessary to allow for 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). It highlights in particular the central role of *convex spaces* (Stone, 1949; Jacobs, 2010; Fritz, 2015) to formalize convexity without resorting to vector spaces.

Contributions. In this paper, we provide a construction of the geometrically convex monad that can be formalized by integrating reusable components. To the best of our knowledge, this is the first formalization in a proof assistant of the monad that combines probabilistic and nondeterminism choices while retaining idempotence of probabilistic choice. This construction is original; in particular, we adapt the pencil-and-paper construction of Cheung (2017) to an infinitary setting using Beaulieu's operator for infinite nondeterministic choice (Beaulieu, 2008, Def. 3.2.3). We partly build on previous formalization work: theories of convex spaces (Affeldt *et al.*, 2020a), interfaces for monadic equational reasoning and finitely supported probability distributions (Affeldt *et al.*, 2019). The new components that complete the construction of the geometrically convex monad are: a formalization of the (non-empty) convex powerset functor (Bonchi *et al.*, 2017, Section 5.1) and affine functions (based on convex spaces), a formalization of semicomplete semilattice structures (related to Beaulieu's work), and an original formalization of concrete categories. They are built in a reusable way following in particular the methodology of packed classes (Garillot *et al.*, 2009). We will discuss how our choices allow these distinct formalizations to fit together. All these formal libraries are now available to tackle similar formalizations that are already numerous as explained above. Our formalization of the geometrically convex monad already has a direct application: it is used to complete an existing formalization of monadic equational reasoning called MONAE (Affeldt *et al.*, 2019). The latter 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. Our work improves the trusted base of this practical tool by filling this hole.

Paper outline. In Section 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 discuss the subtleties arising from their interaction. We provide for that purpose several illustrative examples including a mechanization of the Monty Hall problem as presented by Gibbons, before giving an overview of the construction of the geometrically convex monad. In Section 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 Section 4, we explain the formalization of semicomplete semilattice structures, which provide an operator to represent a nondeterministic choice compatible with the probabilistic choice. In Section 5, we explain a formalization of concrete categories to build monads out of adjoint functors. In Section 6, we define several adjunctions, from which we derive the geometrically convex monad through composition. In Section 7, we verify that the geometrically convex monad can be equipped with the combined choice and that the latter enjoys the expected properties. Finally, we further comment on related work in Section 8 and conclude in Section 9.

About notations. For the sake of clarity, we try to display the Coq source code as it is. However, to limit the amount of code, we often indicate the surrounding namespace using a comment instead of displaying the precise Coq constructs (most of the time, this means that the name of the surrounding `Module` appears as a comment for the reader to figure out the fully qualified names). To further ease reading, we perform some beautification using LaTeX symbols instead of ASCII art. When there are too many details, we omit parts of the source code (and mark them as "...") and instead provide a paraphrase and indicate to the reader where to look in the formalization. In the prose, we use as much as possible standard mathematical notations, sometimes augmented to avoid too much overloading (for example, we write $f@(X)$ for the direct image of the set X by f but $F\#(g)$ for the application of the functor F to the morphism g).

About the formalization. This paper comes with a Coq formalization which is available online as open source software (Infotheo, 2021; Monae, 2021).

2 Formalization target and approach

Our goal is to construct a monad that combines probabilistic and nondeterministic choices, as intended by Gibbons & Hinze (2011). Here, we review an existing formalization in Coq of Gibbons and Hinze's monads and their interfaces (Affeldt *et al.*, 2019); our formalization target is the model of the monad of type `altProbMonad`.

2.1 An existing hierarchy of probability-related monads

Figure 1 provides an excerpt of an existing hierarchy of effects formalized (Affeldt *et al.*, 2019) in Coq that includes the ones by Gibbons & Hinze (2011) (amended as suggested by Abou-Saleh *et al.* (2016)). The complete hierarchy can be found in the online development (Monae, 2021, file `hierarchy.v`).

In Figure 1, `functor` and `monad` are Coq types, respectively, for endofunctors and monads on Coq's `Type` universe. In particular, `monad` is equipped with a join operator `Join` and a unit operator `Ret`. In this section we use rather the bind operator, defined as $m \gg\!\!= f \stackrel{\text{def}}{=} \mathit{Join}(M\#(f)\, m)$ for the monad `M`. The precise definitions of `functor` and `monad` are not relevant here but they are documented in previous work (Affeldt *et al.*, 2019, Section 2.1).

The `functor` and `monad` types, as well as the other monad types in Figure 1, are implemented using the methodology of *packed classes* (Garillot *et al.*, 2009). Packed classes provide the same functionality as type classes in Agda (The Agda Team, 2020) or Idris (Brady, 2013). Providing an implementation for a type defined as a packed class amounts to defining an instance of the corresponding type class. 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 main ingredient to extend a packed class is the notion of *mixin*, which is essentially an interface, defining new operators and/or axioms. In the following, we will focus on mixins when introducing new packed classes, because understanding mixins is sufficient to understand our contributions in this paper.

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

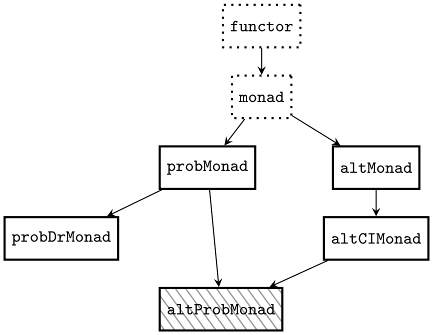

Fig. 1. Hierarchy of effects related to the monad type `altProbMonad` that combines nondeterministic and probabilistic choices. The interfaces above `altProdMonad` have already been given models in MONAE (Affeldt *et al.*, 2019), while the model of `altProdMonad` is the purpose of this paper.

For the sake of completeness, this section however ends with an illustration of how extension using packed classes is actually implemented in COQ and more details can be found in previous work (Affeldt *et al.*, 2019, Section 2.2).

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 $a \triangleleft p \triangleright b$, where a and b are computations and p is a *probability*, i.e., a real number p such that $0 \le p \le 1$. The intuition is that the computation $a \triangleleft p \triangleright b$ represents the computation that behaves like a with probability p and like b with probability $1 - p$. The properties, or *axioms*, of the interface are identity axioms (lines 5 and 6), skewed commutativity (line 7), idempotence (line 8), quasi-associativity (line 9), and the fact that bind left distributes over probabilistic choice (line 12).

```
1  (* Module MonadProb. *)
2  Record mixin_of (M : monad) := Mixin {
3    choice : ∀ (p : prob) T, M T → M T → M T
4             where "a ◁ p ▷ b" := (choice p a b) ;
5    _ : ∀ T (a b : M T), a ◁ 0%:pr ▷ b = b ;
6    _ : ∀ T (a b : M T), a ◁ 1%:pr ▷ b = a ;
7    _ : ∀ T p (a b : M T), a ◁ p ▷ b = b ◁ p.~%:pr ▷ a ;
8    _ : ∀ T p (a : M T), a ◁ p ▷ a = a ;
9    _ : ∀ T (p q r s : prob) (a b c : M T),
10       p = r * s :> R ∧ s.~ = p.~ * q.~ →
11       a ◁ p ▷ (b ◁ q ▷ c) = (a ◁ r ▷ b) ◁ s ▷ c ;
12   _ : ∀ p A B (m1 m2 : M A) (k : A → M B),
13       (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). Here, the piece of syntax `_ = _ :> R` coerces probabilities to the type of COQ real numbers before checking equality. Note the naming convention: we define the type `probMonad` in a module called `MonadProb`. Skewed commutativity allows to derive one of the identity axioms from the other; here we are just preserving the original interface from Gibbons & Hinze (2011). It is well known that finitely supported distributions provide a model of

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

`probMonad`, which we have already formalized in MONAE (Affeldt *et al.*, 2019, Section 6.2) (see also Section 6.2 in this paper).

The monad type `probDrMonad` extends `probMonad` with right distributivity of bind over probabilistic choice, i.e., the law that causes degeneracy when combined with nondeterministic choice, as said in introduction. We do not display its interface 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 17 below), it also states that bind left distributes over nondeterministic choice (line 18), as specified by the following mixin:

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

One can provide a formal model for `altMonad` using lists or sets (Affeldt *et al.*, 2019). 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 & Hinze, 2011, Section 4.2).

The monad type `altCIMonad` extends `altMonad` with commutativity and idempotence of nondeterministic choice, as expressed by the following mixin, where `op x y` stands for `x □ y`:

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

One can provide a formal model for `altCIMonad` using sets (Affeldt *et al.*, 2019).

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

```
(* Module MonadAltProb. *)
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, *classes*, and *structures*. For example, here are the class and structure definitions for `altProbMonad`.

```
(* Module MonadAltProb. *)
Record class_of (M : Type → Type) := Class {
  base : MonadAltCI.class_of M ;
  mixin_prob : MonadProb.mixin_of
```

[1] Gibbons and Hinze use the name “choice” and the identifier `alt` for what we call nondeterministic choice; they call nondeterministic choice a combination of choice and failure (Gibbons & Hinze, 2011, Section 4.3).

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

```
31       (Monad.Pack (MonadAlt.base (MonadAltCI.base base))) ;
32     mixin_altProb : @mixin_of
33       (MonadAltCI.Pack base) (@MonadProb.choice _ mixin_prob) }.
34   Structure altProbMonad := Pack {
35     acto :> Type → Type ; class : class_of acto }.
```

(In the code above, the modifier `@` disables implicit arguments and the type declaration `:>` turns the corresponding structure field into a coercion.) The class definition inherits from `altCIMonad` through its class (line 29), and extends it with two mixins: the one we have seen for `probMonad` (line 30) and the additional distributivity axiom we have just defined (line 32). The structure (line 34) then packages together the type constructor `acto` with the class defined above. Finally, the triple mixin class structure is completed with additional coercions and unification hints to achieve the inheritance relations depicted in Figure 1. Unification hints are provided by the `Canonical` command of Coq as explained by Mahboubi & Tassi (2013). More technical details about the construction of the hierarchy of effects can be found in previous work (Affeldt *et al.*, 2019, Section 2).

The Coq community is in the process of automating the construction of packed class hierarchies. This shall remove the need to explicitly craft the `class_of` record and declare coercions and unification hints. However, the heart of the construction, i.e., the mixin definitions, will not change significantly (Cohen *et al.*, 2020).

2.2 Simple examples combining nondeterministic and probabilistic choice

We reproduce sample programs by Gibbons (2012, Section 5.1) and simple proofs by monadic equational reasoning 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.
```

These two programs can be used to make a probabilistic choice followed by an arbitrary choice and to compare the results:

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

or to make an arbitrary choice followed by a probabilistic choice and to compare the results:

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

Here, the order matters. In the first case, we make an arbitrary choice when the coin has already been flipped. Since the choice is arbitrary, whether it matches the coin or not is arbitrary too, making the probabilistic choice irrelevant. In the latter case, we flip a coin

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

```
     coinarb p
=    do c ← (Ret true ◁p▷ Ret false); do a ← arb; Ret (a == c)
=    (do c ← Ret true; do a ← arb; Ret (a == c)) ◁p▷
     (do c ← Ret false; do a ← arb; Ret (a == c))        (prob_bindDl)
=    (do a ← arb; Ret (a == true)) ◁p▷
     (do a ← arb; Ret (a == false))                      (!bindretf)
=    (Ret (true == true) □ Ret (false == true)) ◁p▷
     (Ret (true == false) □ Ret (false == false))        (!alt_bindDl,!bindretf)
=    (Ret true □ Ret false) ◁p▷ (Ret false □ Ret true)
=    (Ret true □ Ret false) ◁p▷ (Ret true □ Ret false)   (altC)
=    Ret true □ Ret false                                (choicemm)
=    arb
```

Fig. 2. Rewriting steps for `coinarb_spec`.

after making an arbitrary choice. Since the coin is biased, the preliminary choice changes the probability whether they match or not, yet the result is not completely arbitrary.

Using MONAE, one can prove that `coinarb` and `arb` are actually the same program by means of mere rewritings:

```
Lemma coinarb_spec p : coinarb p = arb.
Proof.
rewrite /coinarb /bcoin prob_bindDl !bindretf.
by rewrite /arb !alt_bindDl !bindretf [X in _ ◁_▷ X]altC choicemm.
Qed.
```

The rewriting steps are in Figure 2. Each lemma corresponds to an axiom from an interface. In order, `prod_bindDl` corresponds to left distributivity of bind over probabilistic choice, `bindretf` corresponds to the fact that `Ret` is the left neutral of bind, `alt_bindDl` corresponds to left distributivity of bind over nondeterministic choice, `altC` corresponds to the commutativity of nondeterministic choice, which we apply to the right side of probabilistic choice, and `choicemm` to the idempotence of probabilistic choice. Other `rewrite` invocations are to unfold definitions.

In the same way, one can also prove that `arbcoin` is effectively an arbitrary choice between two probabilities, corresponding to two coins with opposite bias.

```
Lemma arbcoin_spec p : arbcoin p = bcoin p □ bcoin p.~%:pr.
Proof.
rewrite /arbcoin /arb alt_bindDl 2!bindretf bindmret; congr (_ □ _).
by rewrite /bcoin choiceC prob_bindDl 2!bindretf eqxx.
Qed.
```

The proof just relies on basic monadic laws, the left distributivity of bind over both choices, and the symmetry of probabilistic choice.

Furthermore, we can use the `arb` and `bcoin` programs to give a concrete intuition of how *convexity* arises in the semantics. Convexity manifests itself as the possibility to extend any arbitrary choice with a probabilistic choice combining its sides:

```
Lemma alt_absorbs_choice T (x y : M T) p : x □ y = (x □ y) □ x ◁ p ▷ y.
```

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

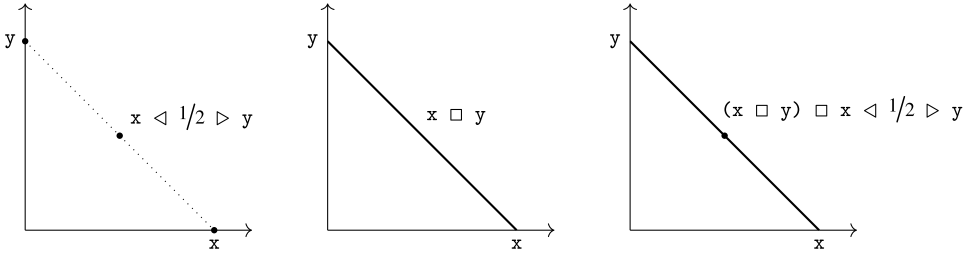

Fig. 3. Geometric intuition for the lemma `alt_absorbs_choice`.

This directly applies to `arb`, which is an arbitrary choice between `Ret true` and `Ret false`:

```
Corollary arb_spec_convexity p : arb = arb □ bcoin p.
```

Again, these proofs rely only on the interfaces we described in Section 2.1; they can be found online (Monae, 2021, file `proba_lib.v`).

The examples in this section show that there are subtle interactions between the two choices that shall be clarified thanks to an explicit model of `altProbMonad`. Moreover, a closer look at these examples already provides us with some intuition about the construction of such a model. In particular, the lemma `alt_absorbs_choice` signals that we will need some closure property on the sets representing nondeterminism in combined choice. This is to contrast with the use of arbitrary sets for simple nondeterminism. This leads us to the well-known solution of using non-empty convex sets of probability distributions, as was hinted at in the introduction. An intuitive way to understand this for probability distributions of up to three-point sets is to use a geometric representation proposed by Abou-Saleh *et al.* (2016). Figure 3 illustrates this representation for the lemma `alt_absorbs_choice`. The set `x ◁ 1/2 ▷ y` is the middle point between `x` and `y`. The set `x □ y` is the segment between `x` and `y`. The last figure is explained by the fact that `x ◁ 1/2 ▷ y` is a subset of `x □ y`.

2.3 A larger example: Mechanization of the Monty Hall problem

We provide a mechanization of the Monty Hall problem using probability *and* nondeterminism as described by Gibbons (2012, Section 6.1) (we have also mechanized a purely probabilistic variant (Gibbons, 2012, Section 6; Gibbons & Hinze, 2011, Section 8.1) as well as a forgetful variant (Gibbons, 2012, Section 7.2)). This example demonstrates monadic equational reasoning using `altProbMonad`, whose construction is the main topic of the rest of this paper.

Let us recall the Monty Hall problem. The player is given a choice of three doors: there is a car behind one door and there are goats behind the other doors. First, the player picks one door and the host opens one of the other doors behind which there is a goat. The player is then asked whether they want to stick to their first choice or switch to the other door. It turns out that the best strategy is to switch, although it appears to be counterintuitive for many, as shown by the controversy the problem sparked when exposed in 1990 in Parade, an American Sunday newspaper magazine (the problem was originally posed and solved

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

in the American Statistician in 1975). It illustrates subtleties of dealing with probabilistic choice.

2.3.1 Problem setting

Let us consider the datatype `door` consisting of three different doors `A`, `B`, and `C` (`doors` is a list consisting of these three doors). The host hides the car behind one of the three doors chosen nondeterministically (hence `altMonad`) (below, `def` has type `door` unless quantified and the suffix `_n` emphasizes functions that depend on nondeterministic choice):

```
Definition hide_n {M : altMonad} : M door := arbitrary def doors.
```

The function `arbitrary` takes a default element and a list and returns an element of the list chosen nondeterministically (or the default element if the list is empty). It is defined using standard functions as follows:

```
Definition arbitrary {M : altMonad} {A : Type} (def : A) : seq A → M A :=
  foldr1 (Ret def) (fun x y ⇒ x □ y) \o map Ret.
```

The player picks one of the doors uniformly at random (using `probMonad`):

```
Definition pick {M : probMonad} : M door := uniform def doors.
```

The function `uniform` is defined using the binary probabilistic choice as follows:

```
Fixpoint uniform {M : probMonad} {A : Type} (def : A) (s : seq A) : M A :=
  match s with
    | [::] ⇒ Ret def
    | [:: x] ⇒ Ret x
    | x :: xs ⇒
      Ret x ◁ (/ IZR (Z_of_nat (size (x :: xs))))%:pr ▷ uniform def xs
  end.
```

The host teases the player by opening a door, which is neither the one hiding the car nor the one picked by the player, chosen nondeterministically:

```
Definition tease_n {M : altMonad} (h p : door) : M door :=
  arbitrary def (doors \\ [:: h; p]).
```

We can now arrange above elements chronologically to represent a game, the latter being parameterized by the strategy of the player (using the `do` notation instead of the bind operator):

```
(* generic game *)
Definition monty {M : monad} hide pick tease
    (strategy : door → door → M door) :=
  do h ← hide ;
  do p ← pick ;
  do t ← tease h p ;
  do s ← strategy p t ;
  Ret (s == h).

(* nondeterministic variant *)
Variable M : altProbMonad.
Definition play_n (strategy : door → door → M door) : M bool :=
  monty hide_n (pick def) tease_n strategy.
```

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

We finally provide the two possible strategies. The "stick" strategy is defined by returning the already-chosen door:

```
Definition stick {M : monad} (p t : door) : M door := Ret p.
```

The "switch" strategy is defined by returning the other door (the one that was neither picked nor used for teasing):

```
Definition switch {M : monad} (p t : door) : M door :=
  Ret (head def (doors \\ [:: p ; t])).
```

2.3.2 Switch is better than stick

One can prove that the "switch" strategy is better than the "stick" strategy by comparison with a biased coin (as defined in the previous section).

More precisely, one can show that the "switch" strategy is as good as a 2/3-biased coin (recall from Section 2.1 that `(/ 3).~%:pr` is the probability $1 - 1/3 = 2/3$):

```
Lemma monty_switch : play_n (switch def) = bcoin (/ 3).~%:pr.
```

The proof goes as follows.

1. The left-hand side `play_n (switch def)` can be rewritten as:

   ```
   hide_n >>= (fun h => pick def >>= (fun p => tease_n h p >>=
       (fun t => Ret (h == head def (doors \\ [:: p; t])))))
   ```

   This step essentially amounts to using the property that the unit is the left neutral of bind.
2. The rightmost continuation can furthermore be rewritten to lead to:

   ```
   hide_n >>= (fun h => pick def >>= (fun p => tease_n h p >>
       (if h == p then Ret false else Ret true)))
   ```

   This step is essentially by case analysis on `h == p` and observation of the expression

   ```
   head def (doors \\ [:: p; t]).
   ```

   When `h == p`, this expression cannot be `h`. When `h != p`, it is `h`.
3. Since teasing does not influence the outcome anymore, the left-hand side can furthermore be simplified into:

   ```
   hide_n >>= (fun h => pick def >>= (fun p => Ret (h != p)))
   ```

   The main lemma needed for this step can be stated in a generic way as follows (where `m1 >> m2` is a notation for `m1 >>= (fun _ => m2)`):

   ```
   Lemma arbitrary_inde (M : altCIMonad) T (a : T) s U (m : M U) :
     0 < size s -> arbitrary a s >> m = m.
   ```

4. The last step produces the expected biased coin `bcoin (/ 3).~%:pr`. This is captured by the following lemma:

   ```
   Lemma bcoin23E :
     arbitrary def doors >>=
       (fun h => uniform def doors >>= (fun p => Ret (h != p))) =
     bcoin (/ 3).~%:pr.
   ```

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

Its proof essentially appeals to the properties of probabilistic choice as specified by the interface of probMonad seen in Section 2.1 and to the fact that bind left distributes over nondeterministic choice, a property of altMonad.

On the other hand, the "stick" strategy is as good as a 1/3-biased coin:

```
Lemma monty_stick : play_n stick = bcoin (/ 3)%:pr.
```

The proof is a bit simpler. It suffices to observe that the teasing does not influence the outcome and use the lemma arbitrary_inde. It is completed by computations similar to the last step of the proof for the "switch" strategy and uses the fact that bind left distributes over nondeterministic choice and probabilistic choice.

As the reader has observed in this section, the example of the Monty Hall problem uses only the interfaces of the involved monads, including the altProbMonad; the rest of this paper (up to Section 6.4.1) provides a formal model for this monad.

See also related work for more proofs by monadic equational reasoning (Gibbons & Hinze, 2011; Gibbons, 2012; Mu, 2019a,b) and their mechanization (Affeldt *et al.*, 2019).

2.4 Alternative axiomatizations of the combined choice

As we mentioned in the introduction, the axiomatization of combined choice we have followed is not the only possible one. We will consider briefly other possible axiomatizations.

The first alternative is obtained by replacing the distributivity axiom added in altProbMonad by the dual axiom, i.e., distributivity of nondeterministic choice over probabilistic choice. This makes sense when probabilistic choice must be resolved before nondeterministic choice (Mislove *et al.*, 2004, Section 1).

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

Using this law, Gibbons (Gibbons, 2012, Section 5) observes that probabilistic choice becomes tainted with nondeterministic choice, as shown by the following example:

```
p ◁ ½ ▷ q = (p ◁ ½ ▷ q) ◁ ½ ▷ (p □ q).
```

More generally, Keimel & Plotkin (2017) have shown that with this law probabilities different from 0 and 1 become indistinguishable (i.e., x ◁ p ▷ y = x ◁ q ▷ y for any $0 < p, q < 1$). The algebraic theory of combined choice then boils down to a bisemilattice (two semilattices with their operators mutually distributing over each other). This is equivalent to having both distributivity laws. While it can be modeled by a powerset monad, the structure is poor, as probability information is lost, so we did not try to formalize this axiomatization. Another way to reach the same axiomatization is to inherit from probDrMonad rather than probMonad (Abou-Saleh *et al.*, 2016, Section 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 alternative is obtained by keeping the same distributivity axiom as in altProdMonad, but removing the idempotence of probabilistic choice from probMonad, i.e.,

$$\begin{tikzcd}
\mathbf{Set} \arrow[r, bend left, "F_0"] \arrow[r, phantom, "\perp"] & \mathsf{Mod}(\textsc{prob}) \arrow[l, bend left, "U_0"] \arrow[r, bend left, "F_1"] \arrow[r, phantom, "\perp"] & \mathsf{Mod}(\textsc{prob} \triangleright \textsc{ndet}) \arrow[l, bend left, "U_1"]
\end{tikzcd}$$

Fig. 4. Cheung's original diagram of adjunctions (Cheung, 2017, Figure 6.1).

we lose the equality `x ◁ p ▷ x = x`. Varacca & Winskel (2006) have 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 can be problematic depending on the application, 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 (which includes idempotence) does not result in a degenerate system is to provide a model where we can check that different computations can be properly distinguished (which we will do for probabilistic choice with the geometrically convex monad in Section 7.2).

2.5 Formalization of the Geometrically Convex Monad: Overview

As already hinted at in the introduction (Section 1) and in Section 2.2, 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, as depicted in Figure 4. This figure depicts three categories related by two adjunctions. The category $\mathsf{Mod}(\textsc{prob})$ consists of models of convexity (for probabilistic choice) and the category $\mathsf{Mod}(\textsc{prob} \triangleright \textsc{ndet})$ consists of models of convexity *and* nondeterminism. The geometrically convex monad results from the composed adjunction $F_1 \circ F_0 \dashv U_0 \circ U_1$. We can derive the monad $U_0 \circ U_1 \circ F_1 \circ F_0$ directly from this adjunction (Mac Lane, 1998).

Now that we have given an overview of the construction of the geometrically convex monad, let us take a step back to think ahead what we need to achieve its formalization. First, we need a formalization of convex spaces. This work has actually started independently (Affeldt *et al.*, 2020a,b) and provides a formalization of convex spaces that can be easily reused (among others, it develops a theory of convex functions). Second, we need a formalization of probability distributions that can be used as an instance of convex spaces and that can be used to form the probability monad. Such a formalization happens to be available in the form of a theory of finitely supported probability distributions (Affeldt *et al.*, 2019), which comes as an enhancement of a theory of finite probability distributions (Affeldt *et al.*, 2014) which could not be used to build a genuine monad because their

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

$$\begin{tikzcd}
\begin{array}{c}\texttt{Type}\\ \mathcal{C}_T\end{array} \arrow[r, bend left, "F_C"] \arrow[r, phantom, "\perp"] & \begin{array}{c}\texttt{choiceType}\\ \mathcal{C}_C\end{array} \arrow[l, bend left, "U_C"] \arrow[r, bend left, "F_0"] \arrow[r, phantom, "\perp"] & \begin{array}{c}\texttt{convType}\\ \mathcal{C}_V\end{array} \arrow[l, bend left, "U_0"] \arrow[r, bend left, "F_1"] \arrow[r, phantom, "\perp"] & \begin{array}{c}\texttt{semiCompSemiLattConvType}\\ \mathcal{C}_S\end{array} \arrow[l, bend left, "U_1"]
\end{tikzcd}$$

Fig. 5. Adjunctions between the categories involved in the construction of the geometrically convex monad.

type cannot give rise to an endofunctor. Third, we can draw inspiration from our previous work on formalizing monadic equational reasoning (Monae, 2021). This work contains in particular a formalization of the basic elements of Cheung's construction (functors, adjunctions, monads, etc.) in the specialized setting of the category **Set**. Our experience with this work led us more precisely to the following technical insights: (1) packed classes are a satisfactory approach to formalizing the needed mathematical structures, (2) affine functions can be accommodated to act as morphisms *provided* one uses concrete categories to generalize from the specialized setting using the category **Set**. The very last bit of the story was to understand precisely Cheung's proofs to realize that an infinitary operator for the representation of the nondeterministic choice was called for (namely, Beaulieu's operator already mentioned in Section 1).

We are now ready to recast Cheung's definition into the Coq formalization we will explain in this paper. Figure 5 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 $\texttt{Type}_0$, $\texttt{Type}_1$, etc. such that $\texttt{Type}_i$ is a subtype of $\texttt{Type}_{i+1}$. By default, Coq hides the indices from the user. We can regard `Type` as a category by seeing each $\texttt{Type}_i$ as a Grothendieck universe (Timany & Jacobs, 2016). The category $\mathcal{C}_C$ corresponds to types satisfying the axiom of choice (i.e., equipped with a choice function). The type `choiceType` (Garillot *et al.*, 2009, Section 3.1) comes from the Mathematical Components library (hereafter, MathComp (Mathematical Components Team, 2007)). The category $\mathcal{C}_V$ is a formalization of $\mathsf{Mod}(\textsc{prob})$ and the category $\mathcal{C}_S$ is a formalization of a subcategory of $\mathsf{Mod}(\textsc{prob} \rhd \textsc{ndet})$ with an infinitary operator for nondeterministic choice instead of a binary one; 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. The desired monad $P_\Delta = P_\Delta^{\text{right}} \circ P_\Delta^{\text{left}}$ is obtained by composing adjunctions:

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

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

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

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 effectively adding the axiom of choice to the ambient logic, which is known to be sound in Coq (The Coq Development Team, 2019). 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.

3 Convexity toolbox

The formalization of the geometrically convex monad naturally calls for a formal theory of convexity. As alluded to in Section 2.5, it can be used to represent the probabilistic choice, convex spaces (needed for the categories $\mathcal{C}_V$ and $\mathcal{C}_S$), non-empty convex sets (to represent computations in a monad modeling `altProbMonad`), convex hulls (to represent nondeterminism), and also to represent the morphisms of the categories $\mathcal{C}_V$ and $\mathcal{C}_S$ (these morphisms are affine functions) and the (non-empty) convex powerset functor F_1. For that purpose, we extend an existing formalization of convex spaces (Affeldt *et al.*, 2020a).

We recall our formalization of convex spaces in Section 3.1; its axiom system leads to a formalization of convex sets and convex hulls, as explained in Section 3.2. We extend this formalization with affine functions and their properties in Section 3.3. See the online development for technical details about this section (Infotheo, 2021, file `convex.v`).

3.1 Formalization of convex spaces

A convex space (a.k.a. barycentric calculus (Stone, 1949)) is an algebraic structure allowing convex combinations of its elements by an operator satisfying several equational axioms. The interface is in fact similar to the interface of the `probMonad` we saw in Section 2.1. It provides an operator $a \triangleleft p \triangleright b$ where a and b are elements of the convex space and p is in the closed unit interval. The axioms about the operator are similar to the ones already explained in Section 2.1 (the reader can observe a difference of presentation for the axiom of quasi-associativity but it is equivalent). Of course, contrary to `probMonad`, convex spaces have no axiom about a bind operator.

```
(* Module ConvexSpace. *)
Record mixin_of (T : choiceType) := 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 }.
```

[2] See Section 6.1 for more details. Actually, `choiceType`s are required because of the FinMap library (Cohen & Sakaguchi, 2015) (which builds upon MathComp) but we do not use the axiom of choice directly in our development.

The notation `[s_of p, q]` stands for $\overline{pq}$; the notation `[r_of p, q]` stands for $p / \overline{pq}$. Note that we assume the carrier type of convex spaces to be a `choiceType` even though it is not strictly required by the axioms; this is rather to fit the rest of the development.

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

We can show, for example, that the real numbers form a convex space by taking the weighted 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 $a \triangleleft p \triangleright b$ to n points, namely $\triangleleft\!\triangleright_d f$, where f consists of n points and d is a distribution of n probabilities.

3.2 Convex sets and convex hulls

We use convex spaces to define *convex sets* and *convex hulls*. As already said in Section 2.5, we put ourselves in a classical setting that extends 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" means "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`` from the MathComp-Analysis library is for the direct image $g@(\text{setT})$ where `setT` is the full set, here, the full set of numbers of type `'I_n` as inferred by the type of g.

The concept of *generator* of a convex set is dual to that of hull: if $Y =$ `hull` X, then X is a generator for the convex set Y.

Remark. At this point, it is worth coming back to the lemma `alt_absorbs_choice` from Section 2.2 (duplicated here for the convenience of the reader):

```
Lemma alt_absorbs_choice T (x y : M T) p : x □ y = x □ y □ x ◁ p ▷ y.
```

We have already stated that we intend outcomes of programs to be represented by convex sets of finitely supported distributions. What this lemma says is that if the semantics of `x` (resp. `y`) is the convex set X (resp. Y), then the semantics of `x □ y` should be the convex hull of $X \cup Y$.

3.3 *Affine functions*

We are interested in *affine functions* because they are used for the morphisms of the categories $\mathcal{C}_V$ and $\mathcal{C}_S$ (Section 2.5). For example, in real analysis, affine functions correspond to the functions of the form $x \mapsto ax + b$. But the real line is just one example of convex space. In fact, the generic operator of convex spaces provides an easy, generic definition.

We define affine functions by first defining an axiom that characterizes functions that distribute over convex combination:

```
(* Module Affine. *)
Variables (U V : convType).
Definition axiom (f : U → V) := ∀ p x y, f (x ◁ p ▷ y) = f x ◁ p ▷ f y.
```

We then define a type for Coq functions packaged with this axiom. This is a methodology similar to packed classes that comes from the MathComp library:

```
(* Module Affine. *)
Structure map (phUV : phant (U → V)) := Pack {apply; _ : axiom apply}.
```

The argument `phant` is an inductive type with one parameter and one constructor `Phant`. It is used among others to define the notation `{affine U → V}` for functions from the convex space `U` to the convex space `V` as follows:

```
(* Module Affine. *)
Notation "{ 'affine' fUV }" := (map (Phant fUV)).
```

See the online development (Infotheo, 2021) for details.

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

```
Proposition image_preserves_convex_hull (f : {affine T → U}) (Z : set T) :
  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. Technical details about this section can be found in the online development (Infotheo, 2021, 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 inhabited 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 & Tassi, 2013).

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

4.1 Semicomplete semilattice

The first structure we introduce provides a unary operator `op` that turns a non-empty set of elements of a lattice into a single element (line 116). As a concrete instance, we will use in Section 4.3 non-empty convex sets as elements and a combination of hull and union for the operator. The first axiom of this structure says that this operator applied to a singleton set returns the sole element of the set (line 117). The second axiom starting at line 118 collapses a non-empty collection `f` (the indexing set `s` itself is not empty) of non-empty sets into one element.

```
114  (* Module SemiCompleteSemiLattice. *)
115  Record mixin_of (T : choiceType) := Mixin {
116    op : neset T → T ;
117    _ : ∀ x : T, op [set x]%:ne = x ;
118    _ : ∀ I (s : neset I) (f : I → neset T),
119          op (⋃_(i ∈ 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 partitions because in our experience they cause technical difficulties in formal proofs.

Hereafter we denote by $\bigsqcup$ the operator introduced by the above mixin and use the mixin to define the type `semiCompSemiLattType` of *semicomplete semilattices* (Bergman, 2015, p. 185). We chose a least upper bound symbol because it seems to be the most appropriate to convey intuitions, but actually we do not use the induced ordering in this paper.

4.2 Combining semicomplete semilattice with convex space

We now extend the structure of semicomplete semilattices from the previous section (Section 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:

$$x \triangleleft p \triangleright \bigsqcup I = \bigsqcup ((\lambda y. x \triangleleft p \triangleright y)@(I))$$

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

```
(* Module SemiCompSemiLattConvType. *)
Record mixin_of (L : semiCompSemiLattType) (conv : prob → L → L → L) :=
  Mixin { _ : ∀ (p : prob) (x : L) (I : neset L),
                conv p x (⨆ I) = ⨆ ((conv 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 (Section 4.1) and the properties of convex spaces (Section 3.1). The methodology to achieve this multiple inheritance is again the one of packed classes.

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

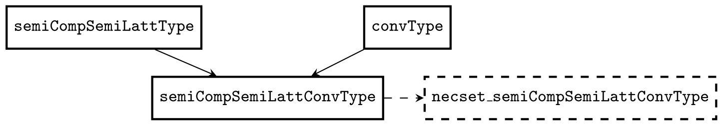

Fig. 6. Hierarchy of semicomplete semilattices structures (dashed lines are for instances).

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

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

The intuition is that for any convex set `hull X`, its least upper bound is the least upper bound of its generator `X`.

The proof is as follows. First, we lift the operator of convex spaces $(\triangleleft p \triangleright)$ from points to sets of points; we denote this lifted operator by $(:\!\triangleleft p \triangleright\!:)$. We use this lifted operator to define a new binary operator $X :\!\Box\!: Y := \bigcup_{p\in[0,1]} X :\!\triangleleft p \triangleright\!: Y$. Second, we show that $\texttt{hull}\, X = \bigcup_{i\in\mathbb{N}} \underbrace{X :\!\Box\!: X :\!\Box\!: \cdots :\!\Box\!: X}_{i+1 \text{ occurrences of } X}$. Then, we show that $\bigsqcup(X) = \bigsqcup(X :\!\Box\!: X :\!\Box\!: \cdots :\!\Box\!: X)$, 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 `biglub_hull`. It can also be used to establish technical results from Beaulieu's work (Beaulieu, 2008, p. 56, l. 3) or similar ones as in Varacca and Winskel's work (Varacca & 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. It is this instance that we will use in particular to produce the adjunction $F_1 \dashv U_1$ (Figure 5).

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 Section 3.2 (and again the methodology of packed classes).

We then instantiate the semicomplete semilattice operator on non-empty convex sets using union and hull operators (`A` below is a convex space):

$$\begin{array}{rccc} \bigsqcup: & \texttt{neset (necset A)} & \longrightarrow & \texttt{necset A} \\ & X & \longmapsto & \texttt{hull}\left(\bigcup_{x\in X} x\right) \end{array}$$

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 `convType A`. We will use this type as the object part of $F_1 : \mathcal{C}_V \to \mathcal{C}_S$.

The structures and instances explained in this section can be summarized as the hierarchy pictured in Figure 6.

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

5 Formalization of category theory based on concrete categories

The purpose of this section is to provide a formalization of enough category theory to construct the geometrically convex monad. This formalization is interesting in itself because it features an original use of concrete categories through their shallow embedding. It also fits our application because it comes as a conservative extension of MONAE (Affeldt *et al.*, 2019). The online development provides all the technical details (Monae, 2021, file `category.v`).

5.1 Formalization based on concrete categories

5.1.1 Shallow embedding of concrete categories

As we saw in Section 2.5, we need to formalize several categories to formalize the geometrically convex monad; this is in contrast with MONAE, which could get along on the sole category **Set** of sets. Among the various possibilities, we chose to favor a definition akin to a shallow embedding. We represent objects as COQ types and morphisms as ordinary COQ functions; as a consequence, we can use the typing relation of COQ to declare elements and a morphism can be applied to an element of an object (as illustrated by the example below). The starting idea 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 `Type`s (the function `el` below at line 128). In this setting, we can then look at the morphisms of a category through the realizer and identify the set of morphisms between two objects as a subset of the function space between two realized objects (via the predicate defining the hom set at line 129).

```
126 (* Module Category. *)
127 Record mixin_of (obj : Type) := Mixin {
128   el : obj → Type ; (* interpretation operation, "realizer" *)
129   inhom : ∀ A B, (el A → el B) → Prop ; (* subset of morphisms *)
130   _ : ∀ A, @inhom A A idfun ; (* idfun is in inhom *)
131   _ : ∀ A B C (f : el A → el B) (g : el B → el C),
132       inhom f → inhom g → inhom (g \o f)
133       (* inhom is closed under composition *) }.
134 Structure type := Pack
135   { carrier : Type ; class : mixin_of carrier }.
```

This definition has two salient features. First, the parameter `obj` lets us choose how we index our objects and use those indices to declare morphisms (e.g., `A` and `B` in `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.
```

Here, `{hom A, B}` is essentially the type of functions `el A → el B` equipped with a proof that they are morphisms (i.e., `f` such that `inhom A B f` holds); there is a coercion from `{hom A, B}` to `el A → el B`. Last, observe that, thanks to the shallow embedding, the

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

laws of units and composition are unnecessary because they are valid definitionally (e.g., line 130 states that `idfun`—the identify function provided by Coq—is a morphism, and it behaves as the identity w.r.t. the composition `\o` of Coq functions by virtue of their native properties).

The resulting encoding is by no way ad hoc: it actually corresponds to a shallow embedding 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 function `inhom` represents the hom sets of $\mathcal{C}$ by their images. For the categorically knowledgeable, the following diagram explains how the morphism part F_{Mor} of the faithful functor F is represented through its image in the hom sets of **Set**.

$$\begin{tikzcd}
 & \mathrm{Im}(F_{\mathrm{Mor}}\restriction_{\mathcal{C}(A,B)}) \arrow[r, "!"] \arrow[d, hook] \arrow[dr, phantom, "\lrcorner", very near start] & * \arrow[d, "\texttt{True}"] \\
\mathcal{C}(A,B) \arrow[ur] \arrow[r, tail, "F_{\mathrm{Mor}}\restriction_{\mathcal{C}(A,B)}"'] & \mathbf{Set}(F(A),F(B)) \arrow[r, "\texttt{inhom A B}"'] & \texttt{Prop}
\end{tikzcd}$$

Let $\mathcal{C}(A,B)$ be a hom set of $\mathcal{C}$, which is mapped by F_{Mor} (restricted to $\mathcal{C}(A,B)$) injectively into the corresponding hom set $\mathbf{Set}(F(A),F(B))$ of **Set**. Note that $\mathbf{Set}(F(A),F(B))$ appears in the Coq code as the type `el A → el B`. The triangle on the left is the image decomposition of $F_{\mathrm{Mor}}\restriction_{\mathcal{C}(A,B)}$. The square on the right is a pullback diagram, with `inhom a b` being the characteristic morphism of the image $\mathrm{Im}(F_{\mathrm{Mor}}\restriction_{\mathcal{C}(A,B)})$.

Except for some hard examples (including homotopy categories), many abstract categories can be concretized, i.e., we can find some faithful functor from the category to **Set** and rephrase it in our framework. The categories in this paper are concretized just by injections, this is also the case for slice categories. Other examples require some encoding of objects and morphisms (e.g., product categories).

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 Section 2.5.

The categories $\mathcal{C}_T$ and $\mathcal{C}_C$. We want to define the category $\mathcal{C}_T$, i.e., the situation in which $\mathcal{C}$ is **Set**, a.k.a. `Type`. We just need to instantiate $F_{\mathrm{Mor}}\restriction_{\mathcal{C}}$ to the identity function and keep all the morphisms. Technically, this amounts to instantiating the mixin of the previous section with the identity function `fun x : Type ⇒ x` 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).
Definition Type_category := Category.Pack Type_category_mixin.
```

(The identifier `I` is a proof of `True` in the standard library of Coq.)

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

Using this setting, we can now use the type Type of Coq as if it were actually the category $\mathcal{C}_T$. The very last ingredient is the declaration of Type_category as a canonical instance of categories:

```
Variable A : Type.
Fail Variable x : el A.
Canonical Type_category.
Variable x : el A.
```

The command Canonical (that we already mentioned for its use in the packed classes methodology in Section 2) provides a unification hint to Coq's type checker to automatically endow Type with the structure of a category when needed. The other instances of categories in this section are also made canonical but we only display the mixins, which hold the relevant information.

Similarly to the category $\mathcal{C}_T$, 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). Again 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).
```

Note that the morphisms of $\mathcal{C}_C$ need not respect the choice functions accompanying choiceTypes, i.e., they do not commute with one another.

The category of convex spaces $\mathcal{C}_V$. The objects are convex spaces (Section 3.1) and the morphisms are affine functions (between convex spaces), which can be enforced by using the axiom from Section 3.3. In our formalization, the objects are indexed by the type of convex spaces convType, and realized by its coercion into Type. 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 idfun_is_affine) and that the composition of affine functions is affine (proof below omitted because generated interactively):

```
Program Definition convType_category_mixin : Category.mixin_of convType :=
  @Category.Mixin
    convType(* Section 3.1 *) (fun A : convType ⇒ A)
    Affine.axiom(* Section 3.3 *) idfun_is_affine _.
```

The category of semicomplete semilattice convex spaces $\mathcal{C}_S$. The objects are semicomplete semilattice convex spaces (Section 4.2) and the morphisms are affine functions f such that $f(\bigsqcup X) = \bigsqcup(f@(X))$ for any non-empty convex set X (definition BiglubAffine.class_of below). We can show that identity functions are such functions (proof idfun_is_biglub_affine) and that composition preserves these properties (proof below omitted because generated interactively), leading to the following definition of $\mathcal{C}_S$:

```
Program Definition semiCompSemiLattConvType_category_mixin :
    Category.mixin_of semiCompSemiLattConvType :=
  @Category.Mixin
```

```
semiCompSemiLattConvType(* Section 4.2 *)
 (fun U : semiCompSemiLattConvType ⇒ U)
BiglubAffine.class_of idfun_is_biglub_affine _.
```

5.2 Formalization of functors, natural transformations, and monads

We now formalize functors, natural transformations, and monads using the concrete categories formalized in the previous section. 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 165 below) and an action on morphisms represented by a function `f : ∀ A B, {hom A, B} → {hom m A, m B}` (line 166) equipped with proofs that `f` preserves the identity (line 167) and composition (line 168):

```
164  (* Module Functor. *)
165  Record mixin_of (C D : category) (m : C → D) := Mixin {
166    f : ∀ (A B : Type), {hom A, B} → {hom m A, m B} ;
167    _ : FunctorLaws.id f ;
168    _ : 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 being interpreted as the category **Set**):

```
Record mixin_of (m : 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 that 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 packaged together with a proof of naturality, we have a genuine natural transformation that we denote by `F ~> G` (note the shorter arrow).

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 180 and 181. These two natural transformations furthermore satisfy three coherence conditions (lines 182, 183, and 184):

```
176  (* Module Monad. *)
177  Record mixin_of (C : category) (M : functor C C) := Mixin {
178    ret : ∀ A, {hom A, M A} ;
179    join : ∀ A, {hom M (M A), M A} ;
```

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

```
  _ : naturality FId M ret ;
  _ : naturality (M ∘ M) M join ;
  _ : ∀ A, join A \o ret (M A) = id ;
  _ : ∀ A, join A \o M # ret A = id ;
  _ : ∀ A, join A \o M # join A = join A \o join (M A) }.
```

We already said above that our formalization of functors generalizes the one of MONAE, the formal framework for monadic equational reasoning on which our work is based. Our formalization of monads also 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 `Type`, regarded as the category **Set**). This way, it will be possible to (1) prove that our formalization of the geometrically convex monad satisfies the expected axioms and (2) retrofit it 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 (see the online development (Monae, 2021, file `category.v`)).

5.3.1 Definition of adjunction

Two functors $F : C \to D$ and $G : D \to C$ are *adjoint* (denoted by $F \dashv G$) when there are two natural transformations $\eta : 1 \rightsquigarrow G \circ F$ (the *unit* of the adjunction) and $\varepsilon : F \circ G \rightsquigarrow 1$ (the *counit* of the adjunction) 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 provide two natural transformations `eta` and `eps` together with the proofs that they satisfy the triangular laws. The corresponding constructor has the following type (where all arguments except 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 adjunctions

It is well known that two adjunctions $F \dashv G$ (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:

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

```
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 of 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 and affine functions in Section 3, semicomplete semilattice structures in Section 4, and category theory in Section 5. In this section, we explain the formalization of the adjunctions explained in Section 2.5. See the online development for technical details (Monae, 2021, file `gcm_model.v`).

6.1 The adjunction $F_C \dashv U_C$

The raison d'être of the adjunction $F_C \dashv U_C$ in our formalization is essentially 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. Rather than introducing coercions from `Type` to `choiceType` in an ad hoc way, we choose to first put ourselves into a world where all types are equipped with a choice function, the category $\mathcal{C}_C$. We do this through an adjunction, which will be eventually combined into our monad.

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, Section 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} :=
  HomPack A B f I.
Local Notation 𝒞_T := Type_category.
Local Notation m := choice_of_Type.
Definition free_choiceType_mor (T U : 𝒞_T) (f : {hom T, U}) :
  {hom m T, m U} := hom_choiceType (f : m T → m U).
```

The purpose of the function `hom_choiceType` is to turn a COQ function between two `choiceType`s into a morphism of the category $\mathcal{C}_C$. Here, `I` (that we already saw in Section 5.1.2) acts as a trivial proof that `f` is indeed a morphism; it is sufficient because

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

in this category all functions are morphisms. The notation `HomPack V W h P` builds a morphism `h` from `V` to `W` where `P` is a proof that this morphism belongs to the hom set (Monae, 2021). 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 retrieves the carrier type of a `choiceType` (see `forget_choiceType` in (Monae, 2021, file `gcm_model.v`)).

The unit $\eta_C : 1 \rightsquigarrow U_C \circ F_C$ is an identity function for each type. The counit $\varepsilon_c : F_C \circ U_C \rightsquigarrow 1$ is also essentially identity for each `choiceType` A, but restoring the original choice function A had before it was endowed with another choice function by F_C. Since morphisms do not respect the choice functions, the proofs of the triangular laws are 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, Section 6.2) that we recall briefly. In the definition of `FSDist.t` below, the first field (line 203) 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 requirement that the input of a finitely supported function be a `choiceType` is a design choice of the FinMap library we are using. The second (anonymous) field (line 205) contains proofs that (1) the probability function outputs positive reals and that (2) its outputs sum to 1.

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

It is important to observe that `FSDist.t` has type `choiceType → choiceType` so that one can talk about distributions over distributions over some `choiceType`. Hereafter, `{dist A}` is a notation for finitely supported distributions over `A` (interpreted as a convex space when appropriate).

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 $\cdot \triangleleft \cdot \triangleright \cdot$ be the operation of the convex space of finitely supported distributions (see Section 3.1) and let $\gg\!=$ be the bind operator of the probability monad. We have $(d_1 \triangleleft p \triangleright d_2) \gg\!= f = (d_1 \gg\!= f) \triangleleft p \triangleright (d_2 \gg\!= f)$, 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 : 𝒞_C) (f : {hom A, B})
    : {hom {dist A}, {dist B}} :=
  HomPack {dist A} {dist B} (FSDistfmap f) (FSDistfmap_affine f).
```

Here, `FSDistfmap_affine` is the proof that `FSDistfmap f` is affine.

We can show that `free_convType_mor` satisfies the functor laws, leading to the definition of the functor F_0:

```
Definition free_convType : functor 𝒞_C 𝒞_V :=
  Functor free_convType_mor_id free_convType_mor_comp.
```

Here, `free_convType_mor_id` and `free_convType_mor_comp` are the proofs of the functor laws and `Functor` builds a functor (recall the definitions of Section 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 (Monae, 2021, 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:

$$\begin{array}{rccc} \varepsilon_0 : & \{\mathtt{dist}\ C\} & \longrightarrow & C \\ & d & \longmapsto & \triangleleft\!\triangleright_d\ \mathtt{finsupp}\,(d)\,. \end{array}$$

In this definition, C is a `convType`; the operation "$\triangleleft\!\triangleright_{\cdot}\ \cdot$" has been explained in Section 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`:

$$\begin{array}{rccc} \eta_0 : & C & \longrightarrow & \{\mathtt{dist}\ C\} \\ & x & \longmapsto & \mathtt{FSDist1.d}\ x. \end{array}$$

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

$$\triangleleft\!\triangleright_{\mathtt{FSDistfmap}\,\mathtt{FSDist1.d}\,d}\ \mathtt{finsupp}\,(\mathtt{FSDistfmap}\ \mathtt{FSDist1.d}\,d) = d.$$

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 $F_1 \dashv U_1$

The third adjunction $F_1 \dashv U_1$ corresponds to the nondeterminism part of the geometrically convex monad, giving a nondeterminism monad over the category $\mathcal{C}_V$ of convex spaces. It consists of the (non-empty) convex powerset functor F_1 and a corresponding forgetful functor U_1.

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

6.3.1 Functors

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

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

The notation `{necset ...}` is a generic notation for non-empty convex sets, here appropriately interpreted by Coq as `necset_semiCompSemiLattConvType` (see Section 4.3).

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—Section 3.3):

```
Definition free_semiCompSemiLattConvType_mor : {hom {necset A},
 {necset B}} :=
  HomPack {necset A} {necset B} free_semiCompSemiLattConvType_mor'
    (BiglubAffine.Class free_semiCompSemiLattConvType_mor'_affine
                        free_semiCompSemiLattConvType_mor'_biglub_morph).
```

To be more precise, this is the lemma `free_semiCompSemiLattConvType_mor'_biglub_morph` that uses the lemma `image_preserves_convex_hull` explained in Section 3.3.

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

```
Definition free_semiCompSemiLattConvType : functor 𝒞_V 𝒞_S :=
  Functor 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` $\mathcal{C}_S$ $\mathcal{C}_V$ is just formalized by substituting the category $\mathcal{C}_S$ by the category $\mathcal{C}_V$ in morphisms (see `forget_semiCompSemiLattConvType` in (Monae, 2021, file `gcm_model.v`)).

6.3.2 Counit/unit

The counit $\varepsilon_1 : F_1 \circ U_1 \rightsquigarrow 1_{\mathcal{C}_S}$ is exactly the $\bigsqcup$ operator seen in Section 4.3:

$$\begin{array}{rccc} \varepsilon_1 : & \texttt{neset}(\texttt{necset}\,T) & \longrightarrow & \texttt{necset}\,T \\ & X & \longmapsto & \bigsqcup X. \end{array}$$

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 `biglub_hull` from Section 4.2), and that it is affine, i.e., $\varepsilon_1(X \triangleleft p \triangleright Y) = \varepsilon_1 X \triangleleft p \triangleright \varepsilon_1 Y$.

Let us comment on the proof that ε_1 preserves the nondeterministic choice to highlight a key difference with Cheung's work (Cheung, 2017). From the proof that ε_1 preserves the infinitary nondeterministic choice (Monae, 2021, lemma `eps1''_biglubmorph`, file `gcm_model.v`), we can derive the proof that it preserves the binary nondeterministic choice (Infotheo, 2021, lemma `biglub_lub_morph`, file `necset.v`). In contrast, Cheung proves the binary version directly. Cheung's setting is finitary but his proofs rely on an implicit connection between finitary and infinitary uses of convex hulls which makes them incomplete (at best). This manifests concretely in the use of an undefined infinitary operator (Cheung, 2017, p. 160). We think that there is a way to make sense of his proof, seeing it as using finitary operators on finite sets whose convex hulls correspond to the infinite sets appearing in his proof, but the theory underlying that reading is completely omitted. We have also experienced in practice that an infinitary setting is more comfortable for formal proofs. Those are the reasons why we think that this formalization is best performed in an infinitary setting.

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.

$$\begin{array}{rccc} \eta_1 : & \texttt{necset}\, T & \longrightarrow & \texttt{neset}(\texttt{necset}\, T) \\ & X & \longmapsto & \{X\} \end{array}$$

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 Section 5.3.1.

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

```
Definition Aprob := adj_comp AC A0.
Definition Agcm := adj_comp Aprob A1.
```

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

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

The very last step is to use the function `Monad_of_category_monad` of Section 5.2 to recover a monad compatible with the MONAE formal framework of monadic equational reasoning[3]:

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

6.4.2 Description of computations inside the Geometrically Convex Monad

Let us look at the computational contents of the geometrically convex monad to gain concrete insights about the model it defines.

We can first observe that the first step of the composed adjunction `Aprob` defines exactly the probability monad. This can be ensured by comparing it to the probability monad

[3] We can also recover the probability monad of Affeldt *et al.* (2019) which is definitionally equal to `Monad_of_category_monad (Monad_of_adjoint (adj_comp AC A0))`.

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

directly defined in MONAE (`M` below). Not only are the types resulting from the two monads identical, their `Join` and `Ret` operations can be proved equal:

```
(* probability monad built directly *)
Definition M := proba_monad_model.MonadProbModel.prob.
(* probability monad built using adjunctions *)
Definition N := Monad_of_category_monad (Monad_of_adjoint Aprob).
Lemma actmE T : N T = M T.
Proof. reflexivity. Qed. (* M T and N T are definitionally equal *)
Lemma JoinE T : (Join : (N ∘ N) T → N T) = (Join : (M ∘ M) T → M T).
Lemma RetE T : (Ret : FId T → N T) = (Ret : FId T → M T).
```

We can also check that the join of the geometrically convex monad 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 Section 6.1, has no computational contents), 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 over non-empty sets of distributions over `T`. The function ε_0 (Section 6.2.2) computes barycenters. Applying it to the elements of `X`, the second component of the function composition returns an object of type `{necset {necset {dist T}}}`. The function ε_1 (Section 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 Geometrically Convex Monad

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—Section 6) satisfies the expected distributivity axioms that we discussed in Section 2.1 *and* to check that it is meaningful, i.e., that it really distinguishes the different choice operators. The missing technical details can be found in the online development (Monae, 2021, file `altprob_model.v`).

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 Section 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 implements `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.~) →
  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 & Hinze (2011).

Leveraging abstraction levels. The proof steps of the above lemmas correspond to the abstraction levels introduced to define the geometrically convex monad. The proof of `bindaltDl` provides an interesting example involving several abstractions.

It proceeds in the following steps:

1. The first step involves reasoning on monads on the category of sets, and deals with monadic expressions that appear in programs. At this level we can apply monad laws that do not dig into other categories than of sets.
   Proof step [`bindaltDl`] Let P_Δ be the geometrically convex monad. The original statement is as follows: for any sets A, B, elements $x, y \in P_\Delta\, A$ and function $k : A \to P_\Delta\, B$,
   $$(\mathsf{do}\ x \leftarrow x \,\square\, y;\ kx) = (\mathsf{do}\ x \leftarrow x;\ kx) \,\square\, (\mathsf{do}\ x \leftarrow y;\ kx).$$
   Rewriting bind in terms of join, this is equivalent to the following equation:
   $$\mu(P_\Delta\#(k)x \,\square\, P_\Delta\#(k)y) = \mu(P_\Delta\#(k)x) \,\square\, \mu(P_\Delta\#(k)y).$$
   We prove it in a more general form: for any set A and elements $u, v \in P_\Delta^2 A$,
   $$\mu(u \,\square\, v) = \mu u \,\square\, \mu v.$$

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

2. The second step deals with more generic category theory. At this level we can unfold the definitions of monads and apply lemmas for adjunctions, natural transformations, etc., involving various categories.
   Proof step [`bindaltDl`] Unfolding the monad, join is reduced to a chain of counits:
   $$\mu = \varepsilon_1 \cdot (F_1 * \varepsilon_0 * U_1) \cdot (F_1 * F_0 * \varepsilon_C * U_0 * U_1)$$
   where $\cdot$ and $*$ are vertical and horizontal compositions of natural transformations respectively. We can then use category-level lemmas to compute both sides of the equality:
   $$\varepsilon_1(F_1\#(\varepsilon_0)(u \,\Box\, v)) = \varepsilon_1(F_1\#(\varepsilon_0)u) \,\Box\, \varepsilon_1(F_1\#(\varepsilon_0)v).$$
3. The third step digs below the level of category theory: the concrete definitions of specific natural transformations and functors. We want to say that some category-theoretic operations satisfy specific algebraic laws.
   Proof step [`bindaltDl`] It only remains to show that both ε_1 and $F_1\#(\varepsilon_0)$ commute with $\Box$. This follows from their being morphisms of the category $\mathcal{C}_S$, which by definition commute with semilattice and convex operations. The commutativity proofs are readily in the morphism structure carried by ε_1 and $F_1\#(\varepsilon_0)$.

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 degenerate, meaning that it indeed distinguishes different probabilities contrary to the alternative axiomatizations described in Section 2.4. It is sufficient to check that there exists a model which is not degenerate 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 works throughout this paper. We add in this section further comments that are better explained now that we have completed the technical presentation of our contributions.

The formalization of convex spaces comes from previous work (Affeldt *et al.*, 2020a) that develops applications of convex spaces such as convex and concave functions and formalizes equivalences between various axiomatizations of binary and multiary convex

https://doi.org/10.1017/S0956796821000137 Published online by Cambridge University Press

operators. Here, we use the multiary convex combination operator in Section 6.2, we further develop the theory of affine functions, and we extend convex spaces to build the convex powerset functor.

In our formalization of semicomplete semilattices (in Section 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, Section 6.3.1). In Section 6.3.2, we explained that Cheung also implicitly uses an infinitary version of his operator and that we find an infinitary operator to be more comfortable and clearer from the viewpoint of formalization.

The monad for probability and nondeterminism can also be presented using finitely generated convex sets of distributions (Bonchi *et al.*, 2019, Section 3.1). Here, we did not insist on having finitely generated convex sets because our first attempt at doing so led to technically involved formal proofs. Now that we have completed our formalization, it should be easier to extend it with finitely generated convex sets. Indeed, looking at Bonchi *et al.* (2020*b*), we recognize several technical results that we have already formalized (e.g., parts of Lemma 4.4). Concretely, the approach would start by defining the data structure for the non-empty finitely generated convex sets by adding an axiom for the existence of a finite generator to the type `necset`, and then by replaying and fixing the proofs (the category part of our framework should stay unchanged). This could open the door to the construction of an executable model, using for instance rational numbers. Such a model would allow computations on concrete programs. More ambitiously, one could use decidability to prove properties of programs in the monad by reflection.

The geometrically convex monad is not the first example of a formalized monad that combines probabilistic and nondeterministic choices: Tassarotti & Harper (2019) formalized in Coq the indexed valuation monad by Varacca and Winskel that we already mentioned in Section 2.4. In this monad, probabilistic choice is not idempotent and therefore it is not suitable for our purpose. Our formalization looks arguably more modular than the one by Tassarotti and Harper who build their monad in a direct manner.

We have been formalizing one model that combines probabilistic and nondeterministic choices: the one advocated by Gibbons & Hinze (2011) because it fits well with functional programming. pGCL (McIver & Morgan, 2005) is another such model that has been formalized in the Isabelle/HOL proof assistant (Cock, 2014) (as such it qualifies as the first formalized model that provides both probabilistic and nondeterministic choices). However, its default semantics is given in different terms (using predicate transformers, no category theory involved, refinement instead of equations) so that the formalizations of the geometrically convex monad and of pGCL turn out to be different tasks. The book by McIver & Morgan (2005) contains also another semantics, the relational demonic semantics whose mathematical construction (Definition 5.4.4) is similar to Cheung's. Yet, it is not presented as a monad with algebraic laws, which is a crucial aspect of our framework, and, to the best of our knowledge, it has not been formalized in a proof assistant.

There is a number of formalizations of category theory in proof assistants (many of which being listed by 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.*,

Table 1. Overview of Relevant Formalization Files

Filename	Contents	Spec.	Proofs	Comments
Files related to combined choice from INFOTHEO (Infotheo, 2021)				
`fsdist.v`	Finitely supported distributions (see Section 6.2 and Affeldt *et al.* (2019))	322	632	41
`convex.v`	Convexity theory (see Section 3 and Affeldt *et al.* (2020*a*))	1002	827	161
`necset.v`	Non-empty convex sets (see Section 4)	568	433	98
Files related to monadic equational reasoning from MONAE (Monae, 2021)				
`hierarchy.v`	Hierarchy of monads (see Section 2.1 and Affeldt *et al.* (2019))	1294	198	142
`category.v`	Category theory (Section 5)	677	264	102
`gcm_model.v`	Geometrically convex monad (Section 6)	323	215	61
`altprob_model.v`	`altProbMonad` model (Section 7)	129	92	5
`proba_lib.v`	Examples from Section 2.2	175	280	94
`example_monty.v`	Monty Hall example (Section 2.3)	137	396	19

(lines of code as provided by `coqwc` (The Coq Development Team, 2021), Spec. stands for "specifications" and corresponds to formal definitions and statements)

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 Section 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 Conclusion and future work

In this paper, we proposed a formalization in the COQ proof assistant of an infinitary version of the geometrically convex monad, a monad that combines probabilistic and nondeterministic choice with idempotence of probabilistic choice. To the best of our knowledge, this is the first formalization of such a monad. Our development led us to develop several formal mathematical theories of broader interest such as a formalization of the convex powerset functor and a formalization of concrete categories. A direct application was to complete an existing formalization of monadic equational reasoning, which was lacking the model of the combined interface of probabilistic and nondeterministic choices and which we illustrated with an extended example. We could also use our model to check that the probabilistic operator does not collapse with the choice of axioms by Gibbons & Hinze (2011) (as fixed by Abou-Saleh *et al.* (2016)).

Our formalization is split between two developments: INFOTHEO (Infotheo, 2021), which provides theories about probabilities and lattices, and MONAE (Monae, 2021) which provides monadic equational reasoning. Table 1 displays the COQ files that are most relevant to this paper. In the end, the original contents amounts to about nine thousands lines of code, but it should be said that these files underwent several rewritings and also benefited from other technical improvements of INFOTHEO and MONAE that are more difficult to quantify.

We formalized an infinitary nondeterministic choice operator. As we discussed in Section 8, it would be interesting to formalize a finitary one with the insights from recent work on finitely generated convex sets (Bonchi *et al.*, 2020a). We could also go in the opposite direction and introduce an infinitary probabilistic choice operator. We just explain the countable case. The countable probability monad `M` should have an operator $\lhd\!\rhd_d m$ where d is a distribution, represented by a function of type `nat → prob`, and `m` is a function of type `nat → M T`. Building such a monad requires two things. First we need to deal with countable sums, which let us define countable distributions and countable convex combinations, to build a model. We also need an algebraic counterpart, based on superconvex spaces (Konig, 1986), which are the countable version of convex spaces. Beaulieu (2008) actually defined such a combined choice monad for infinitary operators. In order to formalize this monad, we could follow the same steps as in the construction we did here, redoing (almost) all the proofs in a countable setting. At this point the only ingredient we have already formalized is the countable sums, which can be found in MATHCOMP-ANALYSIS (Affeldt *et al.*, 2018).

Our experiment is an example of combination of two monads that requires a substantial amount of work. There also exist a number of generic results about the combination of monads such as distributive laws (Zwart & Marsden, 2019) or weak ones (Goy & Petrisan, 2020) that would 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 (Affeldt *et al.*, 2019) from its specialization to the category **Set**.

Acknowledgments

We acknowledge the support of the JSPS KAKENHI Grant Number 18H03204 and the JSPS-CNRS bilateral program "FoRmal tools for IoT sEcurity" (PRC2199), 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, Kazunari Tanaka who contributed to the formalization of categories, Jeremy Gibbons and Joseph Tassarotti for their comments.

Conflicts of interest

None.

References

Abou-Saleh, F., Cheung, K.-H. & Gibbons, J. (2016) Reasoning about probability and nondeterminism. In *POPL Workshop on Probabilistic Programming Semantics*.

Affeldt, R., Hagiwara, M. & Sénizergues, J. (2014) Formalization of Shannon's theorems. *J. Automat. Reason.* **53**(1), 63–103.

Affeldt, R., Cohen, C. & Rouhling, D. (2018) Formalization techniques for asymptotic reasoning in classical analysis. *J. Formaliz. Reason.* **11**(1), 43–76.

Affeldt, R., Nowak, D. & Saikawa, T. (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*, Hutton, G. (ed), Lecture Notes in Computer Science, vol. 11825. Springer, pp. 226–254.

Affeldt, R., Garrigue, J. & Saikawa, T. (2020a) Formal adventures in convex and conical spaces. In *13th International Conference on Intelligent Computer Mathematics (CICM 2020), Bertinoro, Italy, July 26–31, 2020*, Benzmüller, C. & Miller, B. R. (eds), Lecture Notes in Computer Science, vol. 12236. Springer, pp. 23–38.

Affeldt, R., Garrigue, J. & Saikawa, T. (2020b) Reasoning with conditional probabilities and joint distributions in Coq. *Comput. Softw.* **37**(3), 79–95.

Beaulieu, G. (2008) *Probabilistic Completion of Nondeterministic Models*. PhD thesis, Faculty of Graduate and Postdoctoral Studies, University of Ottawa.

Beck, J. (1969) Distributive laws. In *Seminar on Triples and Categorical Homology Theory*, Eckmann, B. (ed), Lecture Notes in Mathematics, vol. 80. Springer, pp. 119–140.

Bergman, G. (2015) *An Invitation to General Algebra and Universal Constructions*. Universitext. Springer International Publishing.

Bonchi, F., Silva, A. & Sokolova, A. (2017) The power of convex algebras. In *28th International Conference on Concurrency Theory (CONCUR 2017), September 5–8, 2017, Berlin, Germany*, Meyer, R. & Nestmann, U. (eds), LIPIcs, vol. 85. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 23:1–23:18.

Bonchi, F., Sokolova, A. & Vignudelli, V. (2019) The theory of traces for systems with nondeterminism and probability. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2019), Vancouver, BC, Canada, June 24–27, 2019*. IEEE, pp. 1–14.

Bonchi, F., Sokolova, A. & Vignudelli, V. (2020a) *Presenting convex sets of probability distributions by convex semilattices and unique bases*. https://arxiv.org/abs/2005.01670. arXiv cs.LO.

Bonchi, F., Sokolova, A. & Vignudelli, V. (2020b) *The Theory of Traces for Systems with Nondeterminism, Probability, and Termination*. https://arxiv.org/abs/1808.00923v4. arXiv cs.LO.

Brady, E. (2013) Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.* **23**(5), 552–593.

Cheung, K.-H. (2017) *Distributive Interaction of Algebraic Effects*. PhD thesis, Merton College, University of Oxford.

Cock, D. (2014) *Leakage in Trustworthy Systems*. PhD thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia.

Cohen, C. & Sakaguchi, K. (2015) *A Finset and Finmap Library: Finite Sets, Finite Maps, Multisets and Order*. Available at: https://github.com/math-comp/finmap. Last stable release: 1.5.0 (2020).

Cohen, C., Sakaguchi, K. & Tassi, E. (2020) Hierarchy builder: Algebraic hierarchies made easy in Coq with Elpi (system description). In *FSCD 2020*. LIPIcs, vol. 167, pp. 34:1–34:21.

Fritz, T. (2015) *Convex Spaces I: Definition and Examples*. https://arxiv.org/abs/0903.5522. arXiv math.MG. First version: 2009.

Garillot, F., Gonthier, G., Mahboubi, A. & Rideau, L. (2009) Packaging mathematical structures. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009), Munich, Germany, August 17–20, 2009*, Berghofer, S., Nipkow, T., Urban, C. & Wenzel, M. (eds), Lecture Notes in Computer Science, vol. 5674. Springer, pp. 327–342.

Gibbons, J. (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*, Wolff, B., Gaudel, M. and Feliachi, A. (eds), Lecture Notes in Computer Science, vol. 7681. Springer, pp. 23–67.

Gibbons, J. & Hinze, R. (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,* Chakravarty, M. M. T., Hu, Z. & Danvy, O. (eds), pp. 2–14. ACM.

Giry, M. (1982) A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, Banaschewski, B. (ed). Lecture Notes in Mathematics, vol. 915. Springer, pp. 68–85.

Goy, A. & Petrisan, D. (2020) Combining probabilistic and non-deterministic choice via weak distributive laws. In *35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2020), Saarbrücken, Germany, July 8–11, 2020*. ACM, pp. 454–464.

Gross, J., Chlipala, A. & Spivak, D. I. (2014) Experience implementing a performant category-theory library in Coq. In *Interactive Theorem Proving,* Klein, G. & Gamboa, R. (eds). Springer International Publishing, pp. 275–291.

Infotheo. (2021) *A Coq Formalization of Information Theory and Linear Error-Correcting Codes*. https://github.com/affeldt-aist/infotheo. Open source software. Since 2009. Version 0.3.2. Software Heritage Archive: swh:1:dir:f8c270f648dd5bc6f822ef16e9354195ddb8f6ad.

Jacobs, B. (2010) Convexity, duality and effects. In *IFIP TCS*. IFIP Advances in Information and Communication Technology, vol. 323. Springer, pp. 1–19.

Kaminski, B. L., Katoen, J., Matheja, C. & Olmedo, F. (2016) Weakest precondition reasoning for expected run-times of probabilistic programs. In *25th European Symposium on Programming Languages and Systems (ESOP 2016), Eindhoven, The Netherlands, April 2–8, 2016*, Thiemann, P. (ed), Lecture Notes in Computer Science, vol. 9632. Springer, pp. 364–389.

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

Konig, H. (1986) Theory and applications of superconvex spaces. In *Aspects of Positivity in Functional Analysis*, Nagel, R., Schloterbeck, U. & Wolff, M. (eds), Mathematics Studies, vol. 122. North-Holland, pp. 79–118.

Mac Lane, S. (1998) *Categories for the Working Mathematician*, 2nd edn. Graduate Texts in Mathematics, vol. 5. Berlin and New York: Springer-Verlag.

Mahboubi, A. & Tassi, E. (2013) Canonical structures for the working Coq user. In *4th International Conference on Interactive Theorem Proving (ITP 2013), Rennes, France, July 22–26, 2013*, Blazy, S., Paulin-Mohring, C. & Pichardie, D. (eds), Lecture Notes in Computer Science, vol. 7998. Springer, pp. 19–34.

Mathematical Components Team. (2007) *Mathematical Components Library*. https://github.com/math-comp/math-comp. Development version. Last stable version 1.12 (2020). Available on the same website.

McBride, C. (1999) *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh.

McIver, A. & Morgan, C. (2005) *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer.

Mio, M. & Vignudelli, V. (2020) *Monads and Quantitative Equational Theories for Nondeterminism and Probability*. https://arxiv.org/abs/2005.07509. arXiv cs.LO.

Mislove, M., Ouaknine, J. & Worrell, J. (2004) Axioms for probability and nondeterminism. *Electronic Notes in Theoretical Computer Science* **96**, 7–28. Proceedings of the 10th International Workshop on Expressiveness in Concurrency.

Mislove, M. W. (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*, Palamidessi, C. (ed), Lecture Notes in Computer Science, vol. 1877. Springer, pp. 350–364.

Monae. (2021) *Monadic Effects and Equational Reasoning in Coq*. https://github.com/affeldt-aist/monae. Open source software. Since 2018. Version 0.3.2. Software Heritage Archive: swh:1:dir:2d68878d365fe72744f8b085fa29df385567f6c9.

Mu, S.-C. (2019a) *Calculating a Backtracking Algorithm: An Exercise in Monadic Program Derivation*. Tech. rept. TR-IIS-19-003. Institute of Information Science, Academia Sinica.

Mu, S.-C. (2019b) *Equational Reasoning for Non-deterministic Monad: A Case study of Spark Aggregation*. Tech. rept. TR-IIS-19-002. Institute of Information Science, Academia Sinica.

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

Tassarotti, J. & Harper, R. (2019) A separation logic for concurrent randomized programs. *Proc. ACM Program. Lang.* **3**(POPL), 64:1–64:30.

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. (2019) *The Logic of Coq*. Available at: https://github.com/coq/coq/wiki/The-Logic-of-Coq. Part of the Coq FAQ.

The Coq Development Team. (2021) *The Coq Proof Assistant Reference Manual*. Inria. Available at: https://coq.inria.fr. Version 8.13.0.

Timany, A. & Jacobs, B. (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*, Kesner, D. and Pientka, B. (eds), LIPIcs, vol. 52, pp. 30:1–30:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Tix, R., Keimel, K. & Plotkin, G. D. (2009) Semantic domains for combining probability and non-determinism. *Electron. Notes Theor. Comput. Sci.* **222**, 3–99.

Varacca, D. & Winskel, G. (2006) Distributing probability over nondeterminism. *Mathematical Structures in Compuer Science* **16**(1), 87–113.

Voevodsky, V., Ahrens, B., Grayson, D., *et al.* (2014) *UniMath — A Computer-Checked Library of Univalent Mathematics*. Available at: https://github.com/UniMath/UniMath. Last stable release 0.1 (2016).

Zwart, M. & Marsden, D. (2019) No-go theorems for distributive laws. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2019), Vancouver, BC, Canada, June 24–27, 2019*. IEEE, pp. 1–13.