A Hierarchy of Monadic Effects for Program Verification Using Equational Reasoning

Reynald Affeldt, David Nowak, Takafumi Saikawa

▶ **To cite this version:**

Reynald Affeldt, David Nowak, Takafumi Saikawa. A Hierarchy of Monadic Effects for Program Verification Using Equational Reasoning. Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Oct 2019, Porto, Portugal. pp.226-254, 10.1007/978-3-030-33636-3_9 . hal-02359796

HAL Id: hal-02359796
https://hal.science/hal-02359796

Submitted on 25 Nov 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Hierarchy of Monadic Effects for Program Verification using Equational Reasoning*

Reynald Affeldt[0000−0002−2327−953X]1, David Nowak[2], and Takafumi Saikawa[0000−0003−4492−745X]3

[1] National Institute of Advanced Industrial Science and Technology, Japan
[2] CRIStAL**, France
[3] Nagoya University & Peano System Inc., Japan

Abstract. One can perform equational reasoning about computational effects with a purely functional programming language thanks to monads. Even though equational reasoning for effectful programs is desirable, it is not yet mainstream. This is partly because it is difficult to maintain pencil-and-paper proofs of large examples. We propose a formalization of a hierarchy of effects using monads in the Coq proof assistant that makes equational reasoning practical. Our main idea is to formalize the hierarchy of effects and algebraic laws like it is done when formalizing hierarchy of traditional algebras. We can then take advantage of the sophisticated rewriting capabilities of Coq to achieve concise proofs of programs. We also show how to ensure the consistency of our hierarchy by providing rigorous models. We explain the various techniques we use to formalize a rich hierarchy of effects (with nondeterminism, state, probability, and more), to mechanize numerous examples from the literature, and we furthermore discuss extensions and new applications.

1 Introduction

Our goal is to provide a framework to produce formal proofs of semantical correctness for programs with effects. To formalize effects, we use *monads*. The notion of monad is one of the category-theoretic frameworks that are used to formalize effects in programming languages and reason about them. It is not the only available option for this purpose (for example, algebraic effects provide an alternative [36, § 5]), but monads comparatively have a longer history in proving themselves useful for the study of semantics [28] as well as for actual programming languages like Haskell as a construct to represent effects [40]. Though there exist a few formalizations of monads in proof assistants, they do not support well our interest in proving programs. Existing formalizations often focus on category theory [17,39] or on meta-theory of programming languages [9]. In

* This is a preprint of a paper to be presented at MPC 2019 (`http://www.cs.nott.ac.uk/~pszgmh/mpc19.html`)

** Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France

contrast, proving programs raises specific practical challenges, among which the generic problem of combining monads is a central issue.

In the practical use-cases of monads in programming, a programmer often has to combine two or more monads in order to deal with several effects in the same context. The combination of monads can be carried out in an ad-hoc way [22]. There exist more generic ways to combine monads under specific conditions [21] including the distributive law between monads, which are unfortunately not always satisfied [38] and therefore do not provide a practical solution.

In this paper, we propose a formalization of monads in the Coq proof assistant that addresses monad combination in a practical way. The main idea is to favor a good representation of the hierarchy of effects and their equational theory in terms of interfaces. In other words, monads are composed as in Haskell. We insist on interfaces but this does not preclude the formal construction of models: they just come afterwards. It happens that this corresponds to the presentation of monads as used in monadic equational reasoning [13], so that a direct consequence of our approach is that we can reproduce formally *and* faithfully pencil-and-paper proofs from the literature.

When it comes to proving properties of effectful programs, there is more than the hierarchy of effects: one also needs to provide practical tools to perform equational reasoning. With this respect, the second aspect of our approach is to leverage the rewriting capabilities of Coq by favoring a *shallow embedding*. Shallow embedding is a well-known encoding technique through which one can reuse the native language of the proof assistant at hand. This bears the promise of a reduced formalization effort and it indeed experimentally met some success [16,20] (formal verification using a shallow embedding often relies on a combination of monads and Hoare logic, e.g., [19]). However, most formal verification frameworks proceeds via a deep embedding of the target language, which requires substantial instrumentations of syntax and semantics, resulting in technical lemmas that are difficult to use, which in turn call for meta-programming. Though this paper favors shallow embedding, it does not prevent syntactical reasoning, as we will demonstrate.

Our main contribution in this paper is to demonstrate a combination of formalization techniques that make formal reasoning about effectful programs in Coq practical:

- We formalize a rich hierarchy of effects (failure, exception, nondeterminism, state, probability, and more) whose heart is the theory by Gibbons and Hinze [13] that we extend with more monads and formal models. The key technique is packed classes [11], a methodology used in the MathComp library [26] to formalize the hierarchy of mathematical structures. We do not know of another mechanization with that many monads.
- We provide many definitions and lemmas that allow for the mechanization of several examples. Because we use a shallow embedding, we can leverage Coq native rewriting capabilities, in particular SSReflect's `rewrite` tactic [15].
- The proof scripts we obtain are faithful to the original proofs. We benchmark our library against numerous examples of the literature (most examples

from [12,13,30,31]) and observe that formal proofs closely match their pencil-and-paper counterparts and that they can actually be shorter thanks to the terseness of SSReflect's tactic language. We also apply our framework to new examples such as the formalization of the semantics of an imperative language.

Outline. In Sections 2 and 3, we show how we build a hierarchy of algebraic laws on top of the theory of monads. In Sect. 4, we illustrate its usability for mechanizing pencil-and-paper proofs. We then deal with syntactic properties in Sect. 5. In Sect. 6, we show how we can give models to our algebraic laws, thus ensuring their consistency. In Sect. 7, we discuss some technical aspects of our formalization of monads that are specific to Coq. We finally discuss related work in Sect. 8 before concluding in Sect. 9.

2 Build a Hierarchy of Algebraic Laws on Top of the Theory of Monads

The heart of our formalization is a hierarchy of effects. Each effect is represented by a monad with some additional algebraic structure that defines the effect, providing effect operators and equations that capture the properties of operators. These effects form a hierarchy in the sense that each effect is the result of a series of extensions starting from the theory of functors, each step extending an existing one in such a way that it shares operators and properties with its parents. We use the methodology of packed classes, which was originally used to formalize mathematical structures [11]. We explain how we use packed classes to formalize monads in Sect. 2.1 and to combine monads in Sect. 2.2. The next section makes a thorough presentation of the complete hierarchy (depicted in Fig. 1).

2.1 Basic Layers: Theories of Functors and Monads

Our formalization of monads starts with a formal definition of functors. This is in contrast to the hierarchy from Gibbons and Hinze [13], where the monad's functor action on morphisms (*fmap*) is defined using *bind* (hereafter, we use the infix notation $\gg=$ for *bind*); starting with functors simplifies the organization of lemmas used in monadic equational reasoning and results in a more robust hierarchy.

Functors as the Base Packed Class. The class of functors is defined in the module `Functor` below. The definition follows the usual one in category theory [25] except that the domain and codomain of functors are fixed to `Type`. In set-theoretical semantics, `Type` is interpreted as the universe of sets, thus rendering our functors to be the endofunctors on the category **Set** of sets and functions.

We use Coq modules only to get a namespace. Inside this namespace, functors are defined by the dependent record `class_of` with one field `f` satisfying the

functor laws (the naming should be self-explanatory, see Table 2, Appendix B in case of doubt). The type of functors `t` is a dependent record[4] with a function `m` of type `Type -> Type`, which is the object part of the functor, that satisfies the `class_of` interface. The morphism part appears as `f` in the record. We define `Fun` to refer to it, but the purpose of the definition is essentially technical. It does not reduce (thanks to the `simpl never` declaration) and can therefore be used to provide a stable notation: `F # g` denotes the action of a functor `F` on a function `g`. Last, we provide a notation `functor` that denotes the type `Functor.t` outside of the module and a coercion so that functors can be used as if they were functions (by taking the first projection `m` of the dependent record that represents their type).

```
Module Functor.
Record class_of (m : Type -> Type) : Type := Class {
  f : forall A B, (A -> B) -> m A -> m B ;
  _ : FunctorLaws.id f ;
  _ : FunctorLaws.comp f }.
Structure t : Type := Pack { m : Type -> Type ; class : class_of m }.
Module Exports.
Definition Fun (F : t) : forall A B, (A -> B) -> m F A -> m F B :=
  let: Pack _ (Class f _ _) := F
  return forall A B, (A -> B) -> m F A -> m F B in f.
Arguments Fun _ [A] [B] : simpl never.
Notation functor := t.
Coercion m : functor >-> Funclass.
End Exports.
End Functor.
Export Functor.Exports.
Notation "F # g" := (Fun F g).
```

Monads as a Packed Class Extension. A monad in category theory is defined as an endofunctor M with two natural transformations $\eta : \mathrm{Id} \to M$ (where Id is the identity endofunctor) and $\mu : M^2 \to M$ satisfying some laws [25]. Following the above definition, our class of monads is defined as an extension of the class of functors.

Inside the module `Monad` below, the interface of monads is captured by the dependent record `mixin_of` with two fields `ret` and `join`, that correspond to η and μ respectively, satisfying the monad laws (Table 2, Appendix B). The type of monads `Monad.t` is a dependent record with a function `Monad.m` of type `Type -> Type` that satisfies a `class_of` interface; the latter extends the class of functors (its `base`) with the `mixin` of monads. Thanks to the definition `baseType`, a monad can also be seen as a functor. This fact is handled transparently by the type system of Coq thanks to the `Canonical` command.

```
Module Monad.
Record mixin_of (M : functor) : Type := Mixin {
```

[4] `Record` and `Structure` are synonymous but the latter is used to emphasize that it is to be made `Canonical`.

```
  ret : forall A, A -> M A ;
  join : forall A, M (M A) -> M A ;
  _ : JoinLaws.ret_naturality ret ;
  _ : JoinLaws.join_naturality join ;
  _ : JoinLaws.left_unit ret join ;
  _ : JoinLaws.right_unit ret join ;
  _ : JoinLaws.associativity join }.
Record class_of (M : Type -> Type) := Class {
  base : Functor.class_of M ; mixin : mixin_of (Functor.Pack base) }.
Structure t : Type := Pack { m : Type -> Type ; class : class_of m }.
Definition baseType (M : t) := Functor.Pack (base (class M)).
Module Exports.
(* intermediate definitions of Ret and Join omitted *)
Notation monad := t.
Coercion baseType : monad >-> functor.
Canonical baseType.
End Exports.
End Monad.
Export Monad.Exports.
```

The monad above is defined in terms of `ret` and `join`. In programming, the operator *bind* is more common. Using Coq notation, its type can be written `forall A B, M A -> (A -> M B) -> M B`. The second argument of type `A -> M B` is a Coq function that represents a piece of effectful program. This concretely shows that we are heading for a framework using a shallow embedding. We provide an alternative way to define monads using *ret* and *bind*. Let us assume that we are given `ret` and `bind` functions that satisfy the monad laws:

```
Variable M : Type -> Type.
Variable bind : forall A B, M A -> (A -> M B) -> M B.
Variable ret : forall A, A -> M A.
Hypothesis bindretf : BindLaws.left_neutral bind ret.
Hypothesis bindmret : BindLaws.right_neutral bind ret.
Hypothesis bindA : BindLaws.associative bind.
```

We can then define `fmap` that satisfies the functor laws:

```
Definition fmap A B (f : A -> B) (m : M A) := bind m (ret (A:=B) \o f).
Lemma fmap_id : FunctorLaws.id fmap.
Lemma fmap_o : FunctorLaws.comp fmap.
```

We can use these lemmas to build `M'` of type `functor` and use `M'` to define `join`:

```
Definition join A (pp : M' (M' A)) := bind pp id.
```

It is now an exercise to prove that `ret` and `join` satisfy the monad laws, using which we eventually build `M` of type `monad`. We call `Monad_of_ret_bind` this construction that we use in the rest of this paper.

2.2 Extensions: Specific Monads as Combined Theories

In the previous section, we explained the case of a simple extension: one structure that extends another. In this section we explain how a structure extends *two* structures. Here, we just explain how we combine theories, how we provide concrete models for combined theories is the topic of Sect. 6.

For the sake of illustration, we use the nondeterminism monad that extends both the failure monad and the choice monad. The failure monad `failMonad` extends the class of monads (Sect. 2.1) with a failure operator `fail` that is a left-zero of *bind*. Since the extension methodology is the same as in Sect. 2.1, we provide the code with little explanations[5]:

```
Module MonadFail.
Record mixin_of (M : monad) : Type := Mixin {
  fail : forall A, M A ;
  _ : BindLaws.left_zero (@Bind M) fail }.
Record class_of (m : Type -> Type) := Class {
  base : Monad.class_of m ; mixin : mixin_of (Monad.Pack base) }.
Structure t := Pack { m : Type -> Type ; class : class_of m }.
Definition baseType (M : t) := Monad.Pack (base (class M)).
Module Exports.
(* intermediate definition of Fail omitted *)
Notation failMonad := t.
Coercion baseType : failMonad >-> monad.
Canonical baseType.
End Exports.
End MonadFail.
Export MonadFail.Exports.
```

The choice monad `altMonad` extends the class of monads with a choice operator `alt` (infix notation: `[~]`; prefix: `[~p]`) that is associative and such that *bind* distributes leftwards over choice (the complete code is displayed in Appendix A).

The nondeterminism monad `nondetMonad` defined below extends both the failure monad and the choice monad. This extension is performed by first selecting the failure monad as the `base` whose base itself is further required to satisfy the mixin of the choice monad (see `base2` below). As a result, a nondeterminism monad can be regarded both as a failure monad (definition `baseType`) or as a choice monad (definition `alt_of_nondet`): both views are declared as `Canonical`.

```
Module MonadNondet.
Record mixin_of (M : failMonad) (a : forall A, M A -> M A -> M A) : Type :=
  Mixin { _ : BindLaws.left_id (@Fail M) a ;
          _ : BindLaws.right_id (@Fail M) a }.
Record class_of (m : Type -> Type) : Type := Class {
  base : MonadFail.class_of m ;
  base2 : MonadAlt.mixin_of (Monad.Pack (MonadFail.base base)) ;
  mixin : @mixin_of (MonadFail.Pack base) (MonadAlt.alt base2) }.
```

[5] Just note that the prefix `@` turns off implicit arguments in Coq.

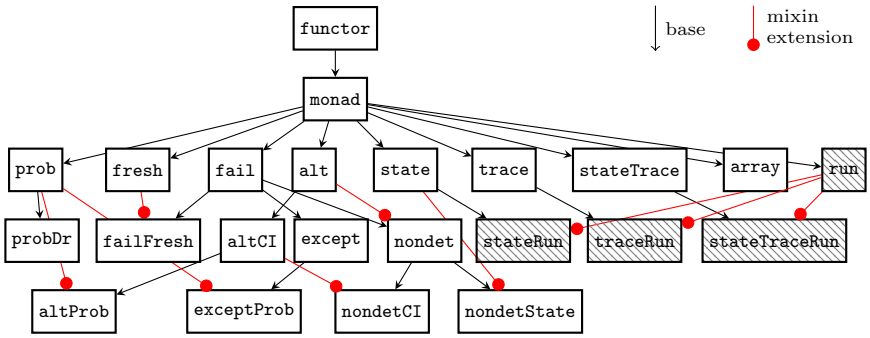

Fig. 1. Hierarchy of effects formalized. See Table 3 for the algebraic laws. In the Coq scripts [3], the monad `xyz` appears as `xyzMonad`.

```
Structure t : Type := Pack { m : Type -> Type ; class : class_of m }.
Definition baseType (M : t) := MonadFail.Pack (base (class M)).
Module Exports.
Notation nondetMonad := t.
Coercion baseType : nondetMonad >-> failMonad.
Canonical baseType.
Definition alt_of_nondet (M : nondetMonad) : altMonad :=
  MonadAlt.Pack (MonadAlt.Class (base2 (class M))).
Canonical alt_of_nondet.
End Exports.
End MonadNondet.
Export MonadNondet.Exports.
```

3 More Monads from our Hierarchy of Effects

This section complements the previous one by explaining more monads from our hierarchy of effects (Fig. 1). We explain these monads in particular because they are used later in the paper[6] They are all obtained using the combination technique previously explained in Sect. 2.2.

3.1 The Exception Monad

The exception monad `exceptMonad` extends the failure monad (Sect. 2.2) with a `Catch` operator with monoidal properties (the `Fail` operator being the neutral) and the property that unexceptional bodies need no handler [13, §5]:

[6] The exception monad is used in the motivating example of Sect. 4.1, state-related monads are used in particular to discuss the relation with deep embedding in Sect. 5.1, the state-trace monad is used in the application of Sect. 5.2, and a model of the probability monad is provided in Sect. 6.2.

```
Record mixin_of (M : failMonad) : Type := Mixin {
  catch : forall A, M A -> M A -> M A ;
  _ : forall A, right_id Fail (@catch A) ;
  _ : forall A, left_id Fail (@catch A) ;
  _ : forall A, associative (@catch A) ;
  _ : forall A x, left_zero (Ret x) (@catch A) }.
```

The algebraic laws are given self-explanatory names; see Table 1, Appendix B in case of doubt.

3.2 The State Monad and Derived Structures

The state monad is certainly the first monad that comes to mind when speaking of effects. It denotes computations that transform a state (type `S` below). It comes with a `Get` operator to yield a copy of the state and a `Put` operator to overwrite it. These functions are constrained by four laws [13]:

```
Record mixin_of (M : monad) (S : Type) : Type := Mixin {
  get : M S ;
  put : S -> M unit ;
  _ : forall s s', put s >> put s' = put s' ;
  _ : forall s, put s >> get = put s >> Ret s ;
  _ : get >>= put = skip ;
  _ : forall k : S -> S -> M S,
      get >>= (fun s => get >>= k s) = get >>= fun s => k s s }.
```

Reification of State Monads. We introduce a `Run` operator to reify state-related monads (this topic is briefly exposed in [13, §6.2], we use reification in Sect. 3.3). First, the operator `run` defines the semantics of `Ret` and `Bind` according to the following equations:

```
Record mixin_of S (M : monad) : Type := Mixin {
 run : forall A, M A -> S -> A * S ;
 _ : forall A (a : A) s, run (Ret a) s = (a, s) ;
 _ : forall A B (m : M A) (f : A -> M B) s,
    run (do a <- m ; f a) s = let: (a', s') := run m s in run (f a') s' }.
```

The type of `run` shows that it turns a state into a pair of a value and a state. We call the monad that extends `monad` with such an operator a `runMonad`. Second, we combine `stateMonad` with `runMonad` and extend it with `Run` equations for `Get` and `Put`; this forms the `stateRunMonad`:

```
Record mixin_of S (M : runMonad S)
  (get : M S) (put : S -> M unit) : Type := Mixin {
  _ : forall s, Run get s = (s, s) ;
  _ : forall s s', Run (put s') s = (tt, s') }.
```

Monads with the `Run` operator appear shaded in Fig. 1, they can be given concrete models so as to run sample programs inside Coq (there are toy examples in [3, file `smallstep_examples.v`]).

The Backtrackable-state Monad. The monad `nondetStateMonad` combines state with nondeterminism (recall that the nondeterminism monad is itself already the result of such a combination) and extends their properties with the properties of *backtrackable-state* ([13, §6], [30, §4]):

```
Record mixin_of (M : nondetMonad) : Type := Mixin {
  _ : BindLaws.right_zero (@Bind M) (@Fail _) ;
  _ : BindLaws.right_distributive (@Bind M) [~p] }.
```

Failure is a right zero of composition to discard any accumulated stateful effects and composition distributes over choice.

3.3 The State-trace Monad

The state-trace monad is the result of combining a state monad with a trace monad. Our trace monad extends monads with a `Mark` operator to record events:

```
Record mixin_of T (m : Type -> Type) : Type :=
  Mixin { mark : T -> m unit }.
```

We call the operators of the state-trace monad `st_get`, `st_put`, and `st_mark` (notations: `stGet`, `stPut`, `stMark`). `stGet` and `stPut` fulfill laws similar to the ones of `Get` and `Put`, but their interactions with `stMark` call for two more laws:

```
Record mixin_of S T (M : monad) : Type := Mixin {
  st_get : M S ;
  st_put : S -> M unit ;
  st_mark : T -> M unit ;
  _ : forall s s', st_put s >> st_put s' = st_put s' ;
  _ : forall s, st_put s >> st_get = st_put s >> Ret s ;
  _ : st_get >>= st_put = skip ;
  _ : forall k : S -> S -> M S,
      st_get >>= (fun s => st_get >>= k s) = st_get >>= fun s => k s s ;
  _ : forall s e, st_put s >> st_mark e = st_mark e >> st_put s ;
  _ : forall e (k : _ -> _ S),
      st_get >>= (fun v => st_mark e >> k v) = st_mark e >> st_get >>= k }
```

3.4 The Probability Monad

First, we define a type `prob` of probabilities [4] as reals of type `R` between 0 and 1:

```
(* Module Prob *)
Record t := mk { p :> R ; pO1 : 0 <= p <= 1 }.
Definition O1 (p : t) := pO1 p.
Arguments O1 : simpl never.
Notation prob := t.
Notation "!`Pr' q" := (@mk q (@O1 _)).
```

This definition is interesting because the notation makes it possible to write concrete probabilities succinctly: the proof that the real is between 0 and 1 is hidden and can be inferred automatically. For example, the probability $\frac{1}{2}$ is written `` `Pr /2 ``, the probability $\bar{p} = 1 - p$ (where p is a probability) is written `` `Pr p.~ ``, etc. This is under the condition that we equip Coq with appropriate canonical structures. For example, here follows the registration of the proof $0 \leq \frac{1}{p} \leq 1$ that makes it possible to write `` `Pr /2 `` (`IZR` injects integers into reals):

```
Lemma prob_IZR (p : positive) : 0 <= / IZR (Zpos p) <= 1.
Canonical probIZR (p : positive) := @Prob.mk _ (prob_IZR p).
```

The above type and notation for probabilities lead us to the following mixin for the probability monad [13, § 8]:

```
1  Record mixin_of (M : monad) : Type := Mixin {
2    choice : forall (p : prob) A, M A -> M A -> M A
3             where "mx <| p |> my" := (choice p mx my) ;
4    _ : forall A (mx my : M A), mx <| `Pr 0 |> my = my ;
5    _ : forall A (mx my : M A), mx <| `Pr 1 |> my = mx ;
6    _ : forall A p (mx my : M A), mx <| p |> my = my <| `Pr p.~ |> mx ;
7    _ : forall A p, idempotent (@choice p A) ;
8    _ : forall A (p q r s : prob) (mx my mz : M A),
9      p = r * s /\ s.~ = p.~ * q.~ ->
10     mx <| p |> (my <| q |> mz) = (mx <| r |> my) <| s |> mz ;
11   _ : forall p, BindLaws.left_distributive (@Bind M) (choice p)  }.
```

`mx <p> my` behaves as `mx` with probability `p` and as `my` with probability $\bar{p}$. Lines 6 and 7 are a skewed commutativity law and idempotence. Lines 8–10 is a quasi associativity law. Above laws are the same as convex spaces [18, Def 3]. Line 11 says that *bind* left-distributes over probabilistic choice.

3.5 Other Monads in the Hierarchy of Effects

Figure 1 pictures the hierarchy of effects that we have formalized; Table 3 (Appendix C) lists the corresponding algebraic laws. The starting point is the hierarchy of [13]. It needed to be adjusted to fit other papers [1,12,30,31]:

- As explained in Sect. 2.1, we put functors at the top to simplify formal proofs.
- The examples of [13] relying on nondeterministic choice use `altMonad`. However, the combination of nondeterminism and probability in `altProbMonad` requires idempotence and commutativity of nondeterministic choice [12]. Idempotence and commutativity are also required in the first part of [31]. We therefore insert the monad `altCIMonad` with those properties in the hierarchy, and also the monad `nondetCIMonad` to deal more specifically with the second part of [31].
- The probability monad `probMonad` is explained in Sect. 3.4. The probability monad `probDrMonad` is explained in [13, §8]. The main difference with [13] is that we extract `probMonad` from `probDrMonad` as an intermediate step.

`probDrMonad` extends `probMonad` with right distributivity of *bind* $(\cdot \gg\!= \cdot)$ over probabilistic choice $(\cdot \triangleleft \cdot \triangleright \cdot)$. The reason is that this property is not compatible with distributivity of probabilistic choice over nondeterministic choice $(\cdot \square \cdot)$ and therefore needs to be put aside to be able to form `altProbMonad` by combining `probMonad` and `altMonad` (the issue is explained in [1]).

There are two more monads that we have not explained. `exceptProbMonad` combines probability and exception [12, §7.1]. `freshMonad` and `failFreshMonad` are explained in [13, §9.1]; `freshMonad` provides an operator to generate fresh labels.

We have furthermore extended the hierarchy of [13] with reification (Sect. 3.2), the trace and state-trace monads (Sect. 3.3), and the array monad [35].

4 Monadic Equational Reasoning

The faithful mechanization of pencil-and-paper proofs by monadic equational reasoning is the main benefit of a hierarchy of effects built with packed classes. After a motivating example in Sect. 4.1, we explain how the Coq `rewrite` tactics copes with notation and lemma overloading in Sect. 4.2. Section 4.3 explains the technical issue of rewriting under function abstractions. Section 4.4 provides an overview of the existing proofs that we have mechanized.

4.1 Motivating Example: the Fast Product

This example shows the equivalence between a functional implementation of the `product` of integers with a monadic version (`fastprod`) [13]. On the left of Fig. 2 we (faithfully) reproduce the series of rewritings that constitute the original proof. On the right, we display the equivalent series of Coq goals and tactics.

The `product` of natural numbers is simply defined as `foldr muln 1`. A "faster" product can be implemented using the failure monad (Sect. 2.2) and the exception monad (Sect. 3.1):

```
Definition work (M : failMonad) s : M nat :=
  if 0 \in s then Fail else Ret (product s).
Definition fastprod (M : exceptMonad) s : M nat := Catch (work s) (Ret 0).
```

We observe that the user can write a monadic program with one monad and use a notation from a monad below in the hierarchy. Concretely, here, `work` is written with `failMonad` but still uses the unit operator `Ret` of the base monad. The same can be said of `fastprod`. This is one consequence of packed classes. What happens is that Coq inserts appropriate calls to canonical structures so that the program type-checks. In fact, the program `work` and `fastprod` are actually *equal* to the following (more verbose) ones:

```
Let Work (M : failMonad) s := if 0 \in s
then @Fail M nat else @Ret (MonadFail.baseType M) nat (product s).
Let Fastprod (M : exceptMonad) s := @Catch M nat
(@work (MonadExcept.baseType M) s) (@Ret (MonadExcept.monadType M) nat 0).
```

Pencil-and-paper proof [13, §5.1]	Coq intermediate goals and tactics
`fastprod xs`	`fastprod s`
=⟦ definition of `fastprod` ⟧	=⟦ `rewrite /fastprod` ⟧
`catch (work xs) (ret 0)`	`Catch (work s) (Ret 0)`
=⟦ specification of `work` ⟧	=⟦ `rewrite /work` ⟧
`catch (if 0 in xs then fail`	`Catch (if 0 \in s then Fail`
`else ret (product xs)) (ret 0)`	`else Ret (product s)) (Ret 0)`
=⟦ lift out the conditional ⟧	=⟦ `rewrite lift_if if_ext` ⟧
`if 0 in xs then catch fail (ret 0)`	`if 0 \in s then Catch Fail (Ret 0)`
`else catch (ret (product xs)) (ret 0)`	`else Catch (Ret (product s)) (Ret 0)`
=⟦ laws of catch, fail, and ret ⟧	=⟦ `rewrite catchfailm catchret` ⟧
`if 0 in xs then ret 0`	`if 0 \in s then Ret 0`
`else ret (product xs)`	`else Ret (product s)`
=⟦ arithmetic: 0 in xs ⇒ `product xs` = 0 ⟧	=⟦ `case: ifPn => // /product0` ⟧
`if 0 in xs then ret (product xs)`	(`product0` $\stackrel{def}{=} \forall s.\ 0 \in s \to$ `product` $s = 0$)
`else ret (product xs)`	`Ret 0`
=⟦ redundant conditional ⟧	=⟦ `move <-` ⟧
`ret (product xs)`	`Ret (product s)`

Fig. 2. Comparison between an existing proof and our Coq formalization

The Coq proof that `fastprod` is pure, i.e., that it never throws an unhandled exception, can be compared to its pencil-and-paper counterpart in Fig. 2. Both proofs are essentially the same, though in practice the Coq proof will be streamlined in two lines (of less than 80 characters) of script:

```
Lemma fastprodE s : fastprod s = Ret (product s).
Proof.
rewrite /fastprod /work lift_if if_ext catchfailm.
by rewrite catchret; case: ifPn => // /product0 <-.
Qed.
```

The fact that we achieve the same conciseness as the pencil-and-paper proof is not because the example is simple: the same can be said of all the examples we mechanized (see Sect. 4.4).

4.2 Basics of Equational Reasoning with Packed Classes

Packed classes not only allow sharing of notations but also sharing of lemmas: one can rewrite a monadic program with any algebraic law from structures below in the hierarchy of effects. SSReflect's advanced `rewrite` tactic[7] becomes available to faithfully reproduce monadic equational reasoning.

For illustration, let us consider a function that nondeterministically builds a subsequence of a list using the choice monad [12, §3.1]:

[7] SSReflect extends Coq's `rewrite` with contextual patterns, unfolding, etc. [15]. The main benefit is that semantically-close actions can be performed on the same line of script, instead of having to interleave with other Coq tactics such as `pattern` or `unfold`.

```
Variables (M : altMonad) (A : Type).
Fixpoint subs (s : seq A) : M (seq A) :=
  if s isn't h :: t then Ret [::]
  else let t' := subs t in fmap (cons h) t' [~] t'.
```

The mixed use of algebraic laws from various monads can be observed when proving that subsequences of concatenation are concatenations of subsequences:

```
1  Lemma subs_cat (xs ys : seq A) :
2    subs (xs ++ ys) = do us <- subs xs; do vs <- subs ys; Ret (us ++ vs).
3  Proof.
4  elim: xs ys => [ys |x xs IH ys].
5    rewrite /= bindretf. (* Ret is left neutral *)
6    by rewrite bindmret. (* Ret is right neutral *)
7  rewrite [in RHS]/=. (* beta-reduction of the rhs *)
8  rewrite alt_bindDl. (* left-distribution of Bind over Alt *)
9  rewrite bindA. (* associativity of Bind *)
10 rewrite [in RHS]/=. (* to be continued in Sect. 4.3 *)
```

The proof is by induction on the sequence `xs` (line 4). While the lemma `alt_bindDl` (line 8) belongs to the interface of the `altMonad` interface, the lemma `bindA` (line 9) comes from the `monad` interface.

4.3 Rewriting under Function Abstractions

In pencil-and-paper proofs of monadic equational reasoning, whether rewriting occurs under a function abstraction or not does not make any difference. We need custom automation to support this feature in Coq which does not natively perform rewriting in this situation.

The proof from the previous section led us to the following subgoal:

```
subs ((x :: xs) ++ ys) =
do x0 <- subs xs; do us <- Ret (x :: x0); do vs <- subs ys; Ret (us ++ vs)
[~] (do us <- subs xs; do vs <- subs ys; Ret (us ++ vs))
```

We want to turn the first branch of the nondeterministic choice

```
do x0 <- subs xs; do us <- Ret (x :: x0); do vs <- subs ys; Ret (us ++ vs)
```

into

```
do x0 <- subs xs; do vs <- subs ys; Ret (x :: x0 ++ vs)
```

but since the occurrence of `Ret` of interest is under the binder "`do x0 <-`", `rewrite bindretf` fails. Instead, we "open" the continuation with a custom tactic `Open (X in subs xs >>= X)` to get a new subgoal

```
do us <- Ret (x :: x0); do vs <- subs ys; Ret (us ++ vs) = ?g x0
```

where `?g` is an existential variable. Now, `rewrite bindretf` succeeds:

```
do vs <- subs ys; Ret ((x :: x0) ++ vs) = ?g x0
```

Yet, the last `Ret` is still under a binder. We could again "open" the continuation but instead we use a custom "rewrite under" tactic `rewrite_ cat_cons` to get:

```
do x1 <- subs ys; Ret (x :: x0 ++ x1) = ?g x0
```

Now we can trigger unification to instantiate the existential variable and thus complete the intended rewriting.

In practice, there is little need for `Open` and most situations can be handled directly without revealing the existential variable using `rewrite_`. We chose to explain `Open` here because it shows how `rewrite_` is implemented.

4.4 Mechanization of Existing Pencil-and-paper Proofs

We used our framework to mechanize the definitions, lemmas, and examples from [13] (except Sect. 10.2), from [12] (up to Sect. 7.2, which overlaps and complements [13]), examples from [30,31], and examples from [21] (up to Sect. 3). This includes in particular:

- Spark aggregation: Spark is a platform for distributed computing, in which the aggregation of data is therefore nondeterministic. Monadic equational reasoning can be used to sort out the conditions under which aggregation is actually deterministic [31, §4.2] as well as other properties. We have mechanized these results [3, file `example_spark.v`], which are part of a larger specification [6].
- The n-queens puzzle: This puzzle is used to illustrate the combination of state and nondeterminism. We have mechanized the relations between functional and stateful implementations [13, §6–7] [3, file `example_nqueens.v`], as well as the derivation of a version of the algorithm using monadic hylo-fusion [30, §5]. This example demonstrates the importance of commutativity lemmas, calling for syntax reflection (see Sect. 5).
- The Monty Hall problem: We have mechanized the probability calculations for several variants of the Monty Hall problem [12,13] using `probMonad`, `altProbMonad`, and `exceptProbMonad` [3, file `example_monty.v`].
- The tree relabeling example: This example originally motivated monadic equational reasoning [13]. It amounts to show that the labels of a binary tree are distinct when the latter has been relabeled with fresh (see `freshMonad`) labels. We have mechanized this result [3, file `example_relabeling.v`].
- The *swap* construction: This is an example of monad composition [21]. Strictly speaking, this is not monadic equational reasoning: formalization does not require a mechanism such as canonical structures. Yet, our framework proved adequate because it allows to mix in a single equation different *ret*'s and *join*'s without explicit mention of which monad they belong to; inference is automatic thanks to coercions.

The level of details provided by the authors using monadic equational reasoning is helpful and provides a way to check that our mechanization is faithful. Among the differences between pencil-and-paper and mechanized proofs,

the main one is maybe function termination. Pencil-and-paper proofs assume Haskell and do not require particular care about function termination, whereas Coq functions must terminate, so that formalization requires an extra effort. See for example the formalization of `unfoldM` and `hyloM` [3] which are not structurally terminating. These difficulties are known [32] and can be addressed using standard techniques. Another difference is that Coq functions must be total, so that some Haskell functions cannot be formalized as such (e.g., `foldr1`).

We discovered a few problems in the work we have formalized. The main one was an error in a proof of monadic hylo-fusion for the n-queens puzzle from a draft paper [29] which has been reported to the author and fixed [30]. In short[8], the functional specification of the n-queens puzzle can be rewritten using `nondetStateMonad` as

```
Get >>= (fun ini => Put (0, [::], [::]) >>
  queensBody (map Z_of_nat (iota 0 n)) >>= overwrite ini)
```

in which `queensBody` can be rewritten as

```
hyloM (@opdot_queens M) [::] (@nilp _)
  select seed_select (@well_founded_size _)
```

The heart of this last step was a theorem [29, Thm 4.2] (now [30, Thm 5.1]) whose hypotheses did not properly match the ones available in the course of the proof. However, we were able to complete the proof with a variant of the theorem in question. Other problems were at the level of typos (they could be easily caught by type-checking): almost none in [13], a few in the appendices of [6] (whose mechanization has not been completed yet).

5 Properties Proved using Syntax

Our formalization is a shallow embedding: a monadic program is a Coq function of return-type `M A` for some monad `M` and some type `A`. This is practical because we can use the Coq language to write, execute, and prove programs. However, it happens that some properties require an explicit syntax to be proved. In this section, we show how to handle such situations. The basic idea is to *locally* restrict programs to a subset characterized by a deep embedding. Section 5.1 is an example of property of backtrackable-states. Section 5.2 is an example of equivalence between an operational and a denotational semantics, the latter being given by a monad.

5.1 The Commutativity of State and Nondeterminism

The commutativity of state and nondeterminism is an important aspect of backtrackable-states [30]. Such a property can be proved directly on specific

[8] We just show the main steps of the derivation, we cannot reproduce all the definitions for lack of space, see the source code [3] for all the details.

programs using their semantics but it can also be proved more generally using syntax.

The following predicate [30, Def 4.2] defines the commutativity of two computations `m` and `n` (in the same monad `M`):

```
Definition commute {M : monad} A B
  (m : M A) (n : M B) C (f : A -> B -> M C) : Prop :=
  m >>= (fun x => n >>= (fun y => f x y)) =
  n >>= (fun y => m >>= (fun x => f x y)).
```

In order to state a generic property of commutativity between nondeterminism and state monads, we first define a predicate that captures syntactically nondeterminism monads. They are written with the following (higher-order abstract [33]) syntax:

```
(* Module SyntaxNondet *)
Inductive t : Type -> Type :=
| ret : forall A, A -> t A
| bind : forall B A, t B -> (B -> t A) -> t A
| fail : forall A, t A
| alt : forall A, t A -> t A -> t A.
```

Let `denote` be a function that turns the above syntax into the corresponding monadic computation:

```
Fixpoint denote (M : nondetMonad) A (m : t A) : M A :=
match m with
| ret A a => Ret a
| bind A B m f => denote m >>= (fun x => denote (f x))
| fail A => Fail
| alt A m1 m2 => denote m1 [~] denote m2
end.
```

Using above definitions, we can write a predicate that captures computations in a `nondetStateMonad` that are actually just computations in a `nondetMonad`:

```
Definition nondetState_sub S (M : nondetStateMonad S) A (n : M A) :=
  {m | denote m = n}.
```

Eventually, it becomes possible to prove *by induction on the syntax* that two computations `m` and `n` using both state and choice commute when `m` actually does not use the state effects:

```
Lemma commute_nondetState S (M : nondetStateMonad S)
  A (m : M A) B (n : M B) C (f : A -> B -> M C) :
  nondetState_sub m -> commute m n f.
```

5.2 Equivalence between Operational and Denotation Semantics

We consider a small imperative language with a state and an operator to generate events. We equip this language with a small-step semantics and a denotational

semantics using `stateTraceMonad` (Sect. 3.3), and prove that both semantics are equivalent. We will see that we need an induction on the syntax to prove this equivalence.

Here follows the (higher-order abstract) syntax of our imperative language:

```
Inductive program : Type -> Type :=
| p_ret  : forall {A}, A -> program A
| p_bind : forall {A B}, program A -> (A -> program B) -> program B
| p_cond : forall {A}, bool -> program A -> program A -> program A
| p_get  : program S
| p_put  : S -> program unit
| p_mark : T -> program unit | ... (* see Appendix~D *)
```

We give our language a small-step semantics specified with continuations in the style of CompCert [5]. We distinguish two kinds of continuations: `stop` for halting and `cont` (notation: $\cdot;\cdot$) for sequencing:

```
Inductive continuation : Type :=
| stop : forall A, A -> continuation
| cont : forall A, program A -> (A -> continuation) -> continuation.
```

We can then define the ternary relation `step` that relates a state to the next one and optionally an event:

```
Definition state : Type := S * @continuation T S.
Inductive step : state -> option T -> state -> Prop :=
| s_ret  : forall s A a (k : A -> _), step (s, p_ret a `; k) None (s, k a)
| s_bind : forall s A B p (f : A -> program B) k,
    step (s, p_bind p f `; k) None (s, p `; fun a => f a `; k)
| s_cond_true : forall s A p1 p2 (k : A -> _),
    step (s, p_cond true p1 p2 `; k) None (s, p1 `; k)
| s_cond_false : forall s A p1 p2 (k : A -> _),
    step (s, p_cond false p1 p2 `; k) None (s, p2 `; k)
| s_get  : forall s k, step (s, p_get `; k) None (s, k s)
| s_put  : forall s s' k, step (s, p_put s' `; k) None (s', k tt)
| s_mark : forall s t k, step (s, p_mark t `; k) (Some t) (s, k tt)
| ... (* see Appendix~D *)
```

Its reflexive and transitive closure `step_star` of type `state -> seq T -> state -> Prop` is defined as one expects. We prove that `step` is deterministic and that `step_star` is confluent and deterministic.

We also give our language a denotational semantics using the `stateTraceMonad`:

```
Variable M : stateTraceMonad S T.
Fixpoint denote A (p : program A) : M A :=
  match p with
  | p_ret _ v => Ret v
  | p_bind _ _ m f => do a <- denote m; denote (f a)
  | p_cond _ b p1 p2 => if b then denote p1 else denote p2
  | p_get => stGet
  | p_put s' => stPut s'
  | p_mark t => stMark t | ... (* see Appendix~D *) end.
```

It is important to note here that the operators `stGet` and `stPut` can only read and update the state (of type `S`) but not the log of emitted events (of type `seq T`). Only the operator `stMark` has access to the list of emitted events but it can neither read nor overwrite it: it can only log a new event to the list.

We proved the correctness and completeness of the small-step semantics `step_star` w.r.t. the denotational semantics `denote` [3, file `smallstep_monad.v`]. For that we use only the equations of the run interface of the state-trace monad (Sect. 3.3). We now come to those parts of the proofs of correctness and completeness that require induction on the syntax. They take the form of two lemmas. Like in the previous section, we introduce a predicate to distinguish the monadic computations that can be written with the syntax of the programming language:

```
Definition stateTrace_sub A (m : M A) := { p | denote p = m }.
```

The first lemma states that once an event is emitted it cannot be deleted:

```
Lemma denote_prefix_preserved A (m : M A) : stateTrace_sub m ->
  forall s s' l1 l a, Run m (s, l1) = (a, (s', l)) ->
    exists l2, l = l1 ++ l2.
```

The second lemma states that the remaining execution of a program does not depend on the previously emitted events:

```
Lemma denote_prefix_independent A (m : M A) : stateTrace_sub m ->
  forall s l1 l2, Run m (s, l1 ++ l2) =
    let res := Run m (s, l2) in (res.1, (res.2.1, l1 ++ res.2.2)).
```

Those are natural properties that ought to be true for any monadic code, and not only the monadic code that results from the denotation of a `program`. But this is not the case with our monad. Indeed, the interface specifies those operators that should be implemented but does not prevent one to add other operators that might break the above properties of emitted events. This is why we restrict those properties to monadic code using the `stateTrace_sub` predicate, thus allowing us to prove the two above lemmas by induction on the syntax.

6 Models of Monads

Sections 2 and 3 explained how to build a hierarchy of effects. In this section, we complete this formalization by explaining how to provide models, i.e., concrete objects that validate the equational theories. Providing a model amounts to define a function of type `Type -> Type` for the base monad and instantiate all the interfaces up to the monad of interest. For illustration, we explain models of state monads and of the probability monad; see [3, file `monad_model.v`] for simpler models.

6.1 Models of State Monads

State-trace Monad. A model for `stateTraceMonad` (Sect. 3.3) is a function `fun A => S * seq T -> A * (S * seq T)`. We start by providing the *ret* and *bind* operators of the base `monad` using the constructor `Monad_of_ret_bind` (Sect. 2.1):

```
(* Module ModelMonad *)
Variables S : Type.
Let m := fun A => S -> A * S.
Definition state : monad.
refine (@Monad_of_ret_bind m
  (fun A a => fun s => (a, s)) (* ret *)
  (fun A B m f => fun s => uncurry f (m s)) (* bind *) _ _ _).
```

One needs to prove the monad laws to complete this definition. This gives a monad `ModelMonad.state` upon which we define the get, put, and mark operators:

```
(* Module ModelStateTrace *)
Variables (S T : Type).
Program Definition mk : stateTraceMonad S T :=
  let m := Monad.class (@ModelMonad.state (S * seq T)) in
  let stm := @MonadStateTrace.Class S T _ m
  (@MonadStateTrace.Mixin _ _ (Monad.Pack m)
    (fun s => (s.1, s)) (* st_get *)
    (fun s' s => (tt, (s', s.2))) (* st_put *)
    (fun t s => (tt, (s.1, rcons s.2 t))) (* st_mark *) _ _ _ _ _ _) in
  @MonadStateTrace.Pack S T _ stm.
```

The laws of the state-trace monad are proved automatically by Coq.

Backtrackable-state. A possible model for `nondetStateMonad` (Sect. 3.2) is `fun A => S -> {fset (A * S)}`, where `{fset X}` is the type of finite sets over `X` provided by the Finmap library. This formalization of finite sets is based on list representations of finite predicates. The canonical representation is chosen uniquely among its permutations. This choice requires the base type `X` of `{fset X}` to be a `choiceType`, i.e., a type equipped with a choice function, thus satisfying a form of the axiom of choice. To be able to use the Finmap library, we use a construct (`gen_choiceMixin`) from the MathComp-Analysis library that can turn any type into a `choiceType`. We use it to define a model for `nondetStateMonad` as follows:

```
Let choice_of_Type (T : Type) : choiceType :=
  Choice.Pack (Choice.Class (equality_mixin_of_Type T) gen_choiceMixin).
Definition _m : Type -> Type :=
  fun A => S -> {fset (choice_of_Type A * choice_of_Type S)}.
```

It remains to prove all the algebraic laws of the interfaces up to `nondetStateMonad`; see [3, file `monad_model.v`] for details.

6.2 A Model of the Probability Monad

A theory of probability distributions provides a model for the probability monad (Sect. 3.4). For this paper, we propose the following definition of probability distribution [4]:

```
(* Module Dist *)
Record t := mk {
  f :> {fsfun A -> R with 0} ;
  f01 : all (fun x => 0 < f x) (finsupp f) &&
        \sum_(a <- finsupp f) f a == 1}.
```

The first field is a finitely-supported function `f`: it evaluates to `0` outside its support `finsupp f`. The second field contains proofs that (1) the probability function outputs positive reals and that (2) its outputs sum to 1. Let `Dist` be a notation for `Dist.t`. It has type `choiceType -> choiceType` and can therefore be used to build a `monad` (thanks to `choice_of_Type` from the previous section).

The *bind* operator is well-known: given `p : Dist A` and `g : A -> Dist B`, it returns a distribution with probability mass function $b \mapsto \sum_{a \in \mathrm{supp(p)}} \mathrm{p}(a) \cdot \mathrm{g}(a, b)$. This is implemented by the following combinator:

```
(* Module DistBind *)
Variables (A B : choiceType) (p : Dist A) (g : A -> Dist B).
Let D := ... (* definition of the support omitted *)
Definition f : {fsfun B -> R with 0} :=
  [fsfun b in D => \sum_(a <- finsupp p) p a * (g a) b | 0].
Definition d : Dist B := ... (* packaging of f omitted *)
```

The resulting combinator `DistBind.d` can be proved to satisfy the monad laws, for example, associativity:

```
Lemma DistBindA A B C (m : Dist A) (f : A -> Dist B) (g : B -> Dist C) :
  DistBind.d (DistBind.d m f) g =
    DistBind.d m (fun x => DistBind.d (f x) g).
```

Completing the model with a distribution for the *ret* operator and the other properties of monads is an exercise.

The last step is to provide an implementation for the interface of the probability monad. The probabilistic choice operator corresponds to the construction of a distribution `d` from two distributions `d1` and `d2` biased by a probability `p`:

```
(* Module Conv2Dist *)
Variables (A : choiceType) (d1 d2 : Dist A) (p : prob).
Definition d : Dist A := locked
  (ConvDist.d (I2Dist.d p) (fun i => if i == ord0 then d1 else d2)).
```

The combinator `ConvDist.d` is a generalization that handles the combination of any distribution of distributions: it is instantiated here with the binary distribution `I2Dist.d p` [4]. We finally prove that the probabilistic choice `d` have the expected properties, for example, skewed commutativity:

```
Notation "x <| p |> y" := (d x y p). (* probabilistic choice *)
Lemma convC (p : prob) (a b : Dist A) : a <| p |> b = b <| `Pr p.~ |> a.
```

7 Technical Aspects of Formalization in Coq

About Coq *Commands and Tactics.* There are several Coq commands and tactics that are instrumental in our formalization. Most importantly, we use Coq canonical structures (as implemented by the command `Canonical`) to implement packed classes (Sect. 2), but also to implement other theories such as probabilities (Sect. 3.4). We already mentioned that the `rewrite` tactic from SSReflect is important to obtain short proof scripts (Sect. 4). We take advantage of the reals of the Coq standard library which come with automation: the `field` and `lra` (linear real/rational arithmetic) tactics are important in practice to compute probabilities (for example in the Monty Hall problem).

About Useful Coq *Libraries.* We use the SSReflect library for lists because it is closer to the Haskell library than the Coq standard library. It provides Haskell-like notations (e.g., notation for comprehension) and more functions (e.g., `allpairs`, a.k.a. `cp` in Haskell). We use the Finmap library of MathComp for its finite sets (see Sect. 6.1). We also benefit from other libraries compatible with MathComp to formalize the model of the probability monad [4].

About the Use of Extra Axioms. We use axioms inherited from the MathComp-Analysis library (they are explained in [2, §5]). More precisely, we use functional extensionality in particular to identify the Coq functions that appear in the *bind* operator. We use `gen_choiceMixin` to turn `Type`s into `choiceType`s when constructing models (see Sect. 6). To provide a model for the probability monad (Sect. 6.2), we proposed a type of probability distributions that requires reals to also enjoy an axiom of choice. We also have a localized use of the axiom of proof irrelevance to prove properties of functors [3, file `monad.v`]. All these axioms make our Coq environment resemble classical set theory. We choose to go with these axioms because it does not restrict the applicability of our work: equational reasoning does not forbid a classical meta-theory with the axiom of choice.

8 Related Work

Formalization of Monads in Coq. Monads are widely used for modeling programming languages with effects. For instance, Delaware et al. formalize several monads and monad transformers, each one associated with a *feature theorem* [9]. When monads are combined, those feature theorems can then be combined to prove type soundness. In comparison, the work we formalize here contains more monads and focuses on equational reasoning about concrete programs instead of meta-theory about programming languages.

Monads have been used in Coq to verify low-level systems [19,20] or for their modular verification [23] based on free monads. Our motivation is similar: enable formal reasoning for effectful programs using monads.

There are more formalizations of monads in other proof assistants. To pick one example that can be easily compared to our mechanization, one can find

a formalization of the Monty Hall problem in Isabelle [8] (but using the pGCL programming language).

About Monadic Equational Reasoning. Although enabling equational reasoning for reasoning about monadic programs seems to be a natural idea, there does not seem to be much related work. Gibbons and Hinze seem to be the first to synthesize monadic equational reasoning as an approach [1,12,13]. This viewpoint is also adopted by other authors [6,30,31,37].

Applicative functor is an alternative approach to represent effectful computations. It has been formalized in Isabelle/HOL together with the tree relabeling example [24]. This work focuses on the lifting of equations to allow for automation, while our approach is rather the one of *small-scale reflection* [14]: the construction of a hierarchy backed up by a rich library of definitions and lemmas to make the most out of the rewriting facilities of Coq.

We extended the hierarchy of Gibbons and Hinze with a state-trace monad with the intent of performing formal verification about programs written with the syntax and semantics of Sect. 5.2. There are actually more topics to explore about the formalization of tracing and monads [34].

About Formalization Techniques. We use packed classes [11] to formalize the hierarchy of effects. It should be possible to use other techniques. In fact, a preliminary version of our formalization was using a combination of telescopes and canonical structures. It did not suffer major problems but packed classes are more disciplined and are known to scale up to deep hierarchies. Coq's type classes have been reported to replace canonical structures in many situations, but we have not tested them here.

The problem of rewriting under function abstraction (Sect. 4.3) is not specific to monadic equational reasoning. For example, it also occurs when dealing with the big operators of the MathComp library, a situation for which a forthcoming version of Coq provides automation [27].

9 Conclusions and Future Work

We reported on the formalization in the Coq proof assistant of an extensive hierarchy of effects with their algebraic laws, and its application to monadic equational reasoning. The key technique is the one of packed classes, which allows for the sharing of notations and properties of various monads, enforces modularity by insisting on interfaces, while preserving the ability to provide rigorous models. We also discussed other techniques of practical interest for monadic equational reasoning such as reasoning on the syntax despite dealing with a shallow embedding. As a benchmark, we applied our formalization to several pencil-and-paper proofs and furthermore formalized and proved properties of the semantics of an imperative programming language. Our approach is successful in the sense that our proof scripts closely match their paper-and-pencil counterparts. Our work also led us to revisit existing proofs and extend the hierarchy of effects originally

proposed by Gibbons and Hinze. We believe that our experiments demonstrate that the formalization of monadic equational reasoning with packed classes and a shallow embedding provides a practical tool for formal verification of effectful programs.

Future Work. We have started the formalization of more examples of monadic equational reasoning [3, branch `experiments`]: [6] is underway, [10] proposes a sharing monad whose equations seems to call for more syntax reflection and brings to the table the issue of infinite data structures.

In its current state the `rewrite_` tactic (Sect. 4.3) is not completely satisfactory. Its main defect is practical: it cannot be chained with the standard `rewrite` tactic. We defer the design of a better solution to future work because the topic is actually more general (as discussed in Sect. 8).

The main task that we are now addressing is the formalization of the model of the monad that combines probability and nondeterminism. Though well-understood [7], its formalization requires a careful formalization of convexity, which is work in progress.

It remains to check whether we can improve the modularity of model construction (or even the extension of the hierarchy) through formalizing other generic methods for combining effects, such as algebraic effects and distributive laws between monads.

Acknowledgements. We acknowledge the support of the JSPS-CNRS bilateral program "FoRmal tools for IoT sEcurity" (PRC2199) and the JSPS KAKENHI Grant Number 18H03204, and thank all the participants of these projects for fruitful discussions. In particular, we thank Jacques Garrigue and Samuel Hym for taking the time to have extended discussions and giving us feedback on drafts of this paper. We also thank Cyril Cohen and Shinya Katsumata for comments about the formalization of monads.

A The Choice Monad

The following excerpt from the source code [3] corresponds to the choice monad first mentioned in Sect. 2.2:

```
Module MonadAlt.
Record mixin_of (M : monad) : Type := Mixin {
  alt : forall A, M A -> M A -> M A ;
  _ : forall A, associative (@alt A) ;
  _ : BindLaws.left_distributive (@Bind M) alt }.
Record class_of (m : Type -> Type) : Type := Class {
  base : Monad.class_of m ; mixin : mixin_of (Monad.Pack base) }.
Structure t := Pack { m : Type -> Type ; class : class_of m }.
Definition baseType (M : t) := Monad.Pack (base (class M)).
Module Exports.
Definition Alt M : forall A, m M A -> m M A -> m M A :=
  let: Pack _ (Class _ (Mixin x _ _)) := M
```

```
  return forall A, m M A -> m M A -> m M A in x.
Arguments Alt {M A} : simpl never.
Notation "'[~p]'" := (@Alt _). (* prefix notation *)
Notation "x '[~]' y" := (Alt x y). (* infix notation *)
Notation altMonad := t.
Coercion baseType : altMonad >-> monad.
Canonical baseType.
End Exports.
End MonadAlt.
Export MonadAlt.Exports.
```

B Generic Algebraic Laws

The algebraic laws used in this paper are instances of generic definitions with self-explanatory names. Table 1 summarizes the laws defined in SSReflect (file `ssrfun.v` from the standard distribution of Coq). Table 2 summarizes the laws introduced in this paper. The Coq definitions are available online [3].

Table 1. Algebraic laws defined in SSReflect

`associative op`	$\forall x, y, z.\ x \,\mathtt{op}\, (y \,\mathtt{op}\, z) = (x \,\mathtt{op}\, y) \,\mathtt{op}\, z$
`left_id e op`	$\forall x.\ e \,\mathtt{op}\, x = x$
`right_id e op`	$\forall x.\ x \,\mathtt{op}\, e = x$
`left_zero z op`	$\forall x.\ z \,\mathtt{op}\, x = z$
`idempotent op`	$\forall x.\ x \,\mathtt{op}\, x = x$

C Summary of Monads and their Algebraic Laws

Table 3 summarizes the structures and the algebraic laws that we formalize and explain in this paper. Precise Coq definitions are available online [3].

D Details about the Imperative Language from Sect. 5.2

For the sake of completeness, we provide the definition of the syntax (`program`) and semantics (operational `step` and denotational `denote`) of the imperative language of Sect. 5.2 where we omitted looping constructs to help reading:

```
Inductive program : Type -> Type :=
| p_ret  : forall {A}, A -> program A
| p_bind : forall {A B}, program A -> (A -> program B) -> program B
| p_cond : forall {A}, bool -> program A -> program A -> program A
| p_get  : program S
| p_put  : S -> program unit
| p_mark : T -> program unit.
```

Table 2. Algebraic laws defined in this paper

`Module FunctorLaws.`	
`id f`	$\mathtt{f}\ id = id$
`comp f`	$\forall g, h.\ \mathtt{f}\ (g \circ h) = \mathtt{f}\ g \circ \mathtt{f}\ h$
`Module JoinLaws.`	
`ret_naturality ret`	$\forall h.\ fmap\ h \circ \mathtt{ret} = \mathtt{ret} \circ h$
`join_naturality join`	$\forall h.\ fmap\ h \circ \mathtt{join} = \mathtt{join} \circ fmap\ (fmap\ h)$
`left_unit ret join`	$\mathtt{join} \circ \mathtt{ret} = id$
`right_unit ret join`	$\mathtt{join} \circ fmap\ \mathtt{ret} = id$
`associativity join`	$\mathtt{join} \circ fmap\ \mathtt{join} = \mathtt{join} \circ \mathtt{join}$
`Module BindLaws.`	
`associative bind`	$\forall m, f, g.\ (m \gg\!= f) \gg\!= g = m \gg\!= \lambda x.(f(x) \gg\!= g)$
`left_id op ret`	$\forall m.\ \mathtt{ret}\ \mathtt{op}\ m = m$
`right_id op ret`	$\forall m.\ m\ \mathtt{op}\ \mathtt{ret} = m$
`left_neutral bind ret`	$\forall f.\ \mathtt{ret} \gg\!= f = f$
`right_neutral bind ret`	$\forall m.\ m \gg\!= \mathtt{ret} = m$
`left_zero bind z`	$\forall f.\ \mathtt{z} \gg\!= f = \mathtt{z}$
`right_zero bind z`	$\forall m.\ m \gg\!= \mathtt{z} = \mathtt{z}$
`left_distributive bind op`	$\forall m, n, f.\ m\ \mathtt{op}\ n \gg\!= f = (m \gg\!= f)\ \mathtt{op}\ (n \gg\!= f)$
`right_distributive bind op`	$\forall m, f, g.\ m \gg\!= \lambda x.(f\ x)\ \mathtt{op}\ (g\ x) = (m \gg\!= f)\ \mathtt{op}\ (m \gg\!= g)$

```
| p_repeat : nat -> program unit -> program unit
| p_while : nat -> (S -> bool) -> program unit -> program unit

Variables T S : Type.
Definition state : Type := S * @continuation T S.
Inductive step : state -> option T -> state -> Prop :=
| s_ret  : forall s A a (k : A -> _), step (s, p_ret a `; k) None (s, k a)
| s_bind : forall s A B p (f : A -> program B) k,
    step (s, p_bind p f `; k) None (s, p `; fun a => f a `; k)
| s_cond_true : forall s A p1 p2 (k : A -> _),
    step (s, p_cond true p1 p2 `; k) None (s, p1 `; k)
| s_cond_false : forall s A p1 p2 (k : A -> _),
    step (s, p_cond false p1 p2 `; k) None (s, p2 `; k)
| s_get  : forall s k, step (s, p_get `; k) None (s, k s)
| s_put  : forall s s' k, step (s, p_put s' `; k) None (s', k tt)
| s_mark : forall s t k, step (s, p_mark t `; k) (Some t) (s, k tt).
| s_repeat_O : forall s p k, step (s, p_repeat O p `; k) None (s, k tt)
| s_repeat_S : forall s n p k,
    step (s, p_repeat n.+1 p `; k) None
         (s, p `; fun _ => p_repeat n p `; k)
| s_while_true : forall fuel s c p k, c s = true ->
    step (s, p_while fuel.+1 c p `; k) None
         (s, p `; fun _ => p_while fuel c p `; k)
| s_while_false : forall fuel s c p k, c s = false ->
    step (s, p_while fuel.+1 c p `; k) None (s, k tt)
| s_while_broke : forall s c p k,
    step (s, p_while O c p `; k) None (s, k tt)
```

```
Variables S T : Type.
Variable M : stateTraceMonad S T.
Fixpoint denote A (p : program A) : M A :=
  match p with
  | p_ret _ v => Ret v
  | p_bind _ _ m f => do a <- denote m; denote (f a)
  | p_cond _ b p1 p2 => if b then denote p1 else denote p2
  | p_get => stGet
  | p_put s' => stPut s'
  | p_mark t => stMark t
  | p_repeat n p => (fix loop m : M unit :=
    if m is m'.+1 then denote p >> loop m' else Ret tt) n
  | p_while fuel c p => (fix loop m : M unit :=
    if m is m'.+1
    then (do s <- stGet ; if c s then denote p >> loop m' else Ret tt)
    else Ret tt) fuel
  end.
```

References

1. Abou-Saleh, F., Cheung, K.H., Gibbons, J.: Reasoning about probability and nondeterminism. In: Workshop on probabilistic programming semantics, St. Petersburg, FL, USA, January 23, 2016 (Jan 2016)
2. Affeldt, R., Cohen, C., Rouhling, D.: Formalization techniques for asymptotic reasoning in classical analysis. J. Formaliz. Reason. **11**(1), 43–76 (2018)
3. Affeldt, R., Garrigue, J., Nowak, D., Saikawa, T.: A Coq formalization of monadic equational reasoning. `https://github.com/affeldt-aist/monae` (2018)
4. Affeldt, R., Hagiwara, M., Sénizergues, J., Garrigue, J., Sakaguchi, K., Asai, T., Saikawa, T., Obata, N.: A Coq formalization of information theory and linear error-correcting codes. `https://github.com/affeldt-aist/infotheo` (2018)
5. Appel, A.W., Blazy, S.: Separation logic for small-step Cminor. In: 20th Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs 2007). Lecture Notes in Computer Science, vol. 4732, pp. 5–21. Springer (2007)
6. Chen, Y., Hong, C., Lengál, O., Mu, S., Sinha, N., Wang, B.: An executable sequential specification for spark aggregation. In: 5th Int. Conf. on Networked Systems (NETYS 2017) Marrakech, Morocco, May 17–19, 2017. Lecture Notes in Computer Science, vol. 10299, pp. 421–438 (2017)
7. Cheung, K.H.: Distributive Interaction of Algebraic Effects. Ph.D. thesis, Merton College, University of Oxford (2017)
8. Cock, D.: Verifying probabilistic correctness in Isabelle with pGCL. In: 7th Systems Software Verification, Sydney, Australia. pp. 1–10 (Nov 2012)
9. Delaware, B., Keuchel, S., Schrijvers, T., d. S. Oliveira, B.C.: Modular monadic meta-theory. In: ACM SIGPLAN Int. Conf. on Functional Programming (ICFP 2013), Boston, MA, USA, September 25–27, 2013. pp. 319–330 (2013)
10. Fischer, S., Kiselyov, O., Shan, C.: Purely functional lazy nondeterministic programming. J. Funct. Program. **21**(4–5), 413–465 (2011)
11. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: 22nd Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs 2009), Munich, Germany, August 17–20, 2009. Lecture Notes in Computer Science, vol. 5674, pp. 327–342. Springer (2009)

12. Gibbons, J.: Unifying theories of programming with monads. In: 4th Int. Symp. on Unifying Theories of Programming (UTP 2012), Paris, France, August 27–28, 2012, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7681, pp. 23–67. Springer (2013)
13. Gibbons, J., Hinze, R.: Just do it: simple monadic equational reasoning. In: 16th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP 2011), Tokyo, Japan, September 19–21, 2011. pp. 2–14. ACM (2011)
14. Gonthier, G., Mahboubi, A.: An introduction to small scale reflection in Coq. J. Formaliz. Reasoning **3**(2), 95–152 (2010)
15. Gonthier, G., Tassi, E.: A language of patterns for subterm selection. In: 3rd Int. Conf. on Interactive Theorem Proving (ITP 2012), Princeton, NJ, USA, August 13–15, 2012. Lecture Notes in Computer Science, vol. 7406, pp. 361–376. Springer (2012)
16. Greenaway, D.: Automated Proof-Producing Abstraction of C Code. Ph.D. thesis, University of New South Wales, Sydney, Australia (Jan 2015)
17. Hirschowitz, A., Maggesi, M.: Modules over monads and initial semantics. Inf. Comput. **208**(5), 545–564 (2010)
18. Jacobs, B.: Convexity, duality and effects. In: IFIP TCS. IFIP Advances in Information and Communication Technology, vol. 323, pp. 1–19. Springer (2010)
19. Jomaa, N., Nowak, D., Grimaud, G., Hym, S.: Formal proof of dynamic memory isolation based on MMU. Sci. Comput. Program. **162**, 76–92 (2018)
20. Jomaa, N., Torrini, P., Nowak, D., Grimaud, G., Hym, S.: Proof-oriented design of a separation kernel with minimal trusted computing base. In: 18th Int. Work. on Automated Verification of Critical Systems (AVOCS 2018), July 2018, Oxford, UK. Electronic Communications of the EASST Open Access Journal (2018)
21. Jones, M.P., Duponcheel, L.: Composing monads. Tech. Rep. YALEU/DCS/RR-1004, Yale University (Dec 1993)
22. King, D.J., Wadler, P.: Combining monads. In: Functional Programming. pp. 134–143. Workshops in Computing, Springer (1992)
23. Letan, T., Régis-Gianas, Y., Chifflier, P., Hiet, G.: Modular verification of programs with effects and effect handlers in Coq. In: 22nd Int. Symp. on formal methods (FM 2018), Oxford, UK (Jul 2018)
24. Lochbihler, A., Schneider, J.: Equational reasoning with applicative functors. In: 7th Int. Conf. on Interactive Theorem Proving (ITP 2016), Nancy, France, August 22–25, 2016. Lecture Notes in Computer Science, vol. 9807, pp. 252–273. Springer (2016)
25. Mac Lane, S.: Categories for the Working Mathematician, Graduate Texts in Mathematics, vol. 5. Springer-Verlag, New York, second edn. (1998)
26. Mahboubi, A., Tassi, E.: Mathematical Components. Available at `https://math-comp.github.io/mcb/` (2016), with contributions by Yves Bertot and Georges Gonthier. Version of 2018/08/11.
27. Martin-Dorel, E., Tassi, E.: SSReflect in Coq 8.10. In: The Coq Workshop 2019, Portland, OR, USA, September 8, 2019. pp. 1–2 (Sep 2019)
28. Moggi, E.: Computational lambda-calculus and monads. In: LICS. pp. 14–23. IEEE Computer Society (1989)
29. Mu, S.C.: Functional pearls, reasoning and derivation of monadic programs, a case study of non-determinism and state (Jul 2017), draft. Available at `http://flolac.iis.sinica.edu.tw/flolac18/files/test.pdf` (last access: 2019/07/10)
30. Mu, S.C.: Calculating a backtracking algorithm: An exercise in monadic program derivation. Tech. Rep. TR-IIS-19-003, Institute of Information Science, Academia Sinica (Jun 2019)

31. Mu, S.C.: Equational reasoning for non-determinism monad: A case study of Spark aggregation. Tech. Rep. TR-IIS-19-002, Institute of Information Science, Academia Sinica (Jun 2019)
32. Mu, S., Ko, H., Jansson, P.: Algebra of programming in Agda: Dependent types for relational program derivation. J. Funct. Program. **19**(5), 545–579 (2009)
33. Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1988), Atlanta, GA, USA, June 22–24, 1988. pp. 199–208. ACM (1988)
34. Piróg, M., Gibbons, J.: Tracing monadic computations and representing effects. In: 4th Workshop on Mathematically Structured Functional Programming (MSFP 2012), Tallinn, Estonia, March 25, 2012. EPTCS, vol. 76, pp. 90–111 (2012)
35. Plotkin, G.D., Power, J.: Notions of computation determine monads. In: FoSSaCS. Lecture Notes in Computer Science, vol. 2303, pp. 342–356. Springer (2002)
36. Pretnar, M.: An introduction to algebraic effects and handlers (invited tutorial paper). Electr. Notes Theor. Comput. Sci. **319**, 19–35 (2015)
37. Shan, C.C.: Equational reasoning for probabilistic programming. In: POPL 2018 TutorialFest (Jan 2018)
38. Varacca, D., Winskel, G.: Distributing probability over non-determinism. Mathematical Structures in Computer Science **16**(1), 87–113 (2006)
39. Voevodsky, V., Ahrens, B., Grayson, D., et al.: UniMath—a computer-checked library of univalent mathematics. Available at `https://github.com/UniMath/UniMath`
40. Wadler, P.: Comprehending monads. In: LISP and Functional Programming. pp. 61–78 (1990)

Table 3. Monads Defined in this Paper and the Algebraic Laws They Introduce

Structure	Operators	Equations
`functor` (§2.1)	`Fun/#`	`functor_id`, `functor_o`
`monad` (§2.1)	`Ret`	`ret_naturality`
	`Join`	`join_naturality`, `joinretM` (left unit), `joinMret` (right unit), `joinA` (associativity)
	`Bind/>>=/>>`	`bindretf` (left neutral), `bindmret` (right neutral), `bindA` (associativity)
`failMonad` (§2.2)	`Fail`	`bindfailf` (fail left-zero of bind)
`altMonad` (§A)	`Alt/[~]/[~p]`	`alt_bindDl` (bind left-distributes over choice), `altA` (associativity)
`nondetMonad` (§2.2)		`altmfail` (right-id), `altfailm` (left-id)
`exceptMonad` (§3.1)	`Catch`	`catchfailm` (left-id),`catchmfail` (right-id), `catchA` (associativity), `catchret` (left-zero)
`stateMonad` (§3.2)	`Get`, `Put`	`putget`, `getputskip`, `putput`, `getget`
`runMonad` (§3.2)	`Run`	`runret`, `runbind`
`stateRunMonad` (§3.2)		`runget`, `runput`
`nondetStateMonad` (§3.2)		`bindmfail` (right-zero), `alt_bindDr` (bind right-distributes over choice)
`traceMonad` (§3.3)	`Mark`	
`stateTraceMonad` (§3.3)	`stGet`	`st_getget`
	`stPut`	`st_putput`, `st_putget`, `st_getputskip`
	`stMark`	`st_putmark`, `st_getmark`
`traceRunMonad` (§3.3)		`runmark`
`stateTraceRunMonad` (§3.3)		`runstget`, `runstput`, `runstmark`
`probMonad` (§3.4)	`Choice`	`choicemm` (idempotence), `choice0`, `choice1` (identity laws), `choiceA` (quasi associativity), `choiceC` (skewed commutativity), `prob_bindDl` (bind left-distributes over choice)
`altCIMonad` (§3.5)		`altmm` (idempotence), `altC` (commutativity)
`nondetCIMonad` (§3.5)		
`freshMonad` (§3.5)	`Fresh`	
`failFreshMonad` (§3.5)	`Distinct`	`failfresh_bindmfail` (fail right-zero of bind) `bassert (Distinct M) \o Symbols = Symbols`
`arrayMonad` (§3.5)	`aGet i`, `aPut i s`	`aputput`, `aputget`, `agetputskip`, `agetget`, `agetC`, `aputC`, `aputgetC`
`probDrMonad` (§3.5)		`prob_bindDr` (bind right-distributes over choice)
`altProbMonad` (§3.5)		`choiceDr` (probabilistic choice right-distributes over nondeterministic choice)
`exceptProbMonad` (§3.5)		`catchDl` (catch left-distributes over choice)