Towards a Practical Library for Monadic Equational Reasoning in Coq

Ayumu Saito[1] and Reynald Affeldt[2][0000−0002−2327−953X]

[1] Tokyo Institute of Technology, School of Computing, Department of Mathematical and Computing Science
[2] National Institute of Advanced Industrial Science and Technology (AIST)

Abstract. Functional programs with side effects represented by monads are amenable to equational reasoning. This approach to program verification has been experimented several times using proof assistants based on dependent type theory. These experiments have been performed independently but reveal similar technicalities such as how to build a hierarchy of interfaces and how to deal with non-structural recursion. As an effort towards the construction of a reusable framework for monadic equational reasoning, we report on several practical improvements of Monae, a Coq library for monadic equational reasoning. First, we reimplement the hierarchy of effects of Monae using a generic tool to build hierarchies of mathematical structures. This refactoring allows for easy extensions with new monad constructs. Second, we discuss a well-known but recurring technical difficulty due to the shallow embedding of monadic programs. Concretely, it often happens that the return type of monadic functions is not informative enough to complete formal proofs, in particular termination proofs. We explain library support to facilitate this kind of proof using standard Coq tools. Third, we augment Monae with an improved theory about nondeterministic permutations so that our technical contributions allow for a complete formalization of quicksort derivations by Mu and Chiang.

1 Introduction

Pure functional programs, being referentially transparent, are suitable for equational reasoning. To reason about programs with side effects, one can use monads and their rich algebraic properties. In monadic equational reasoning, effects are defined by interfaces with a set of equations; these interfaces can be combined and extended to represent the combination of several effects. A number of programs using combined effects (state, nondeterminism, probability, etc.) have been verified this way (e.g., [14, 22–26]), and some of them have been formally verified with proof assistants such as Coq and Agda (e.g., [4, 24–26]).

The application to monadic equational reasoning of different proof assistants raises common issues. The construction of the hierarchy of monad interfaces is such an issue. There exist several approaches such as canonical structures (e.g., [4]) or type classes (e.g., [24]). The construction of a hierarchy of interfaces nevertheless requires care because it is known that large hierarchies suffer scalability issues due to the complexity of type inference (as discussed, e.g., in [12]). In the context of monadic equational reasoning, shallow embedding is the privileged way to represent monadic functions. This is the source of other issues, which are well-known. For example, when needed, induction w.r.t. syntax requires the use of reflection (see, e.g., [4, Sect. 5.1]). Non-structural recursion is another issue related to the use of a shallow embedding. General approaches and tools have been developed to deal with non-structural recursion in proof assistants but they are still cumbersome to use.

The Coq library Monae is an effort to provide a tool for formal verification of monadic equational reasoning. It already proved useful by uncovering errors in pencil-and-paper proofs (e.g., [4, Sect. 4.4]), leading to new fixes for known errors (e.g., [3]), and providing clarifications for the construction of monads used in probabilistic programs (e.g., [2, Sect. 6.3.1]).

Before explaining our contribution in this paper, let us illustrate concretely the main ingredients of monadic equational reasoning in a proof assistant using Monae.

Example of proof by monadic equational reasoning Let us assume that we are given a type `monad` for monads, where `Ret` denotes unit and `>>=` denotes the bind operator (`>>` is defined by `m >> k = m >>= (fun _ => k)`). We can use this type to define a generic function that repeats a computation `mx` (the computation `skip` is `Ret tt`):

```
Fixpoint rep {M : monad} n (mx : M unit) :=
  if n is n.+1 then mx >> rep n mx else skip.
```

Let us also assume that we are given a type `stateMonad T` for monads with a state of type `T` equipped with the usual `get` and `put` operators. We can use this type to define a `tick` function (`succn` is the successor function of natural numbers and ∘ is function composition):

```
Definition tick {M : stateMonad nat} : M unit := get >>= (put ∘ succn).
```

Let us use monadic equational reasoning to prove "tick fusion" [25, Sect. 4.1] (in a state monad; `addn` is the addition of natural numbers):

```
Lemma tick_fusion n : rep n tick = get >>= (put ∘ addn n).
```

Despite the side effect, this proof can be carried out by equational reasoning using standard monadic laws. Computations in any monad satisfy the following laws:

```
bindA     ∀ A B C (m : M A) (f : A → M B) (g : B → M C),
          (m >>= f) >>= g = m >>= (fun a => f a >>= g)
bindretf  ∀ A B (a : A) (f : A → M B), Ret a >>= f = f a
bindmret  ∀ A (m : M A), m >>= Ret = m
```

Computations in a state monad moreover satisfy the following laws:

INITIAL GOAL	rep n tick = get ≫= (put ∘ addn n)	
BASE CASE	rep 0 tick = get ≫= (put ∘ addn 0)	
	get ≫= put = get ≫= (put ∘ addn 0)	(by getputskip)
INDUCTIVE CASE	rep n.+1 tick = get ≫= (put ∘ addn n.+1)	
	(get ≫= (put ∘ succn)) ≫ (get ≫= (put ∘ addn n)) = get ≫= (put ∘ addn n.+1)	(by inductive hyp.)
	get ≫= (fun x ⇒ (put ∘ succn) x ≫ (get ≫= (put ∘ addn n))) = get ≫= (put ∘ addn n.+1)	(by bindA)
	(put ∘ succn) m ≫ (get ≫= (put ∘ addn n)) = (put ∘ addn n.+1) m	(by extensionality)
	((put ∘ succn) m ≫ get) ≫= (put ∘ addn n) = (put ∘ addn n.+1) m	(by bindA)
	(put m.+1 ≫ Ret m.+1) ≫= (put ∘ addn n) = (put ∘ addn n.+1) m	(by putget)
	put m.+1 ≫ (Ret m.+1 ≫= (put ∘ addn n)) = (put ∘ addn n.+1) m	(by bindA)
	put m.+1 ≫ (put ∘ addn n) m.+1 = (put ∘ addn n.+1) m	(by bindretf)
	put (n + m.+1) = (put ∘ addn n.+1) m	(by putput)
	put (n + m.+1) = put (n + m.+1)	(by addSnnS)

Fig. 1. Intermediate goals displayed by Coq when executing the proof script for tick fusion

```
putput      ∀ s s', put s ≫ put s' = put s'
putget      ∀ s, put s ≫ get = put s ≫ Ret s
getputskip  get ≫= put = skip
getget      ∀ A (k : T → T → M A),
            get ≫= (fun s ⇒ get ≫= k s) = get ≫= fun s ⇒ k s s
```

The following proof script (written with the SSREFLECT dialect of the Coq proof language [31]) shows that tick fusion can be proved by induction (using the `elim` tactic) and a sequence of rewritings involving mostly monadic laws (see Fig. 1 for the intermediate goals displayed by Coq or [25, Sect. 4.1] for a pencil-and-paper proof):

```
Lemma tick_fusion n : rep n tick = get ≫= (put ∘ addn n).
Proof.
elim: n ⇒ [|n ih]; first by rewrite /= -getputskip.
rewrite /= /tick ih bindA; bind_ext ⇒ m.
by rewrite -bindA putget bindA bindretf putput /= addSnnS.
Qed.
```

This example illustrates the main ingredients of a typical formalization of monadic equational reasoning: monadic functions (such as `rep` and `tick`) are encoded as functions in the language of the proof assistant (this is a shallow embedding), and monadic equational reasoning involves several monads with inheritance relations (here the state monad satisfies more laws than a generic monad).

Our contribution in this paper is to improve several practical aspects of MONAE [21]. More specifically, we address the following issues:

– In monadic equational reasoning, monadic effects are the result of the combination of several interfaces. The formalization of these interfaces and their

combination in a coherent and reusable hierarchy requires advanced techniques. Monae provides the largest hierarchy [4] we are aware of by using the methodology of *packed classes* [12]. However, when this methodology is implemented manually as in [4], it is verbose and the extension of the hierarchy is error-prone. In this work, we reimplement and extend this hierarchy using a more scalable and robust approach (Sect. 2).
- As we observe in the above example, monadic functions are written with the language of the proof assistant. Though this shallow embedding is simple and natural, in practice it is also the source of inconveniences when proving lemmas in general and when proving termination in particular. Indeed, contrary to a standard functional programming language, a type-based proof assistant requires termination proofs for every function involved. However, it happens that in practice the tooling provided by proof assistants to deal with non-structurally recursive functions is insufficient in the context of monadic equational reasoning. We explain how one can enrich the return type of monadic functions using dependent types to deal with such proofs (Sect. 3); we also propose *dependently-typed assertions* for that purpose (Sect. 4).
- Last, we demonstrate the usefulness of the two previous technical contributions by completing an existing formalization of quicksort by Mu and Chiang (Sect. 5). The original motivation by Mu and Chiang [24] was to demonstrate program derivation using the non-determinism monad and refinement but they left a few postulates unproved in their Agda formalization (see Table 1). We explain how to provide the remaining formal proofs and, as a by-product, we furthermore enrich Monae with, in particular, a theory of nondeterministic permutations.

2 An extensible implementation of monad interfaces

We explain how we formalize a hierarchy of interfaces for monads used in monadic equational reasoning. This hierarchy is a conservative extension and complete reimplementation of previous work [2–4]. In previous work, the hierarchy in question was hand-written and its extension was error-prone (see Sect. 6). To allow for easy and flawless extensions, we use a generic tool called Hierarchy-Builder [11] for the formalization of hierarchies of mathematical structures. The first version of this tool handled hierarchies of structures whose carrier has a type `T : Type`; it has recently[3] been extended to allow carriers with a functional type, and this section is an application of this new feature. As illustrations, we will use Hierarchy-Builder to formalize the plus-array monad and revise a prior formalization of monad transformers.

2.1 Hierarchy-Builder in a nutshell

Hierarchy-Builder extends Coq with commands to define hierarchies of mathematical structures. It is designed so that hierarchies can evolve (for example

[3] Hierarchy-Builder version 1.1.0 (2021-03-30).

by splitting a structure into smaller structures) without breaking existing code. These commands are compiled to packed classes [12] but the technical details (Coq modules, records, coercions, implicit arguments, canonical structures instances, notations, etc.) are hidden to the user. The main concept is the one of *factory*. This is a record defined by the command `HB.factory` that packs a carrier, operations, and properties. This record usually corresponds to the standard definition of a mathematical structure. *Mixins* (defined by the command `HB.mixin`) are factories used as the default definition for a mathematical structure. *Structures* (defined by the command `HB.structure`) are essentially sigma-types with a carrier paired with one or more factories. A mixin usually extends a structure, so it typically takes as parameters a carrier and other structures.

A *builder* is a function that shows that a factory is sufficient to build a mixin. Factories (including mixins) can be instantiated (command `HB.instance`) with concrete objects. Instances are built with `.Build` functions that are automatically generated for each factory. To write a builder, one uses the command `HB.builders` that opens a Coq section starting from a factory and ending with instances of mixins.

In addition to commands to build hierarchies, Hierarchy-Builder also checks their validity by detecting missing interfaces (see Sect. 6) or *competing inheritance paths* [1].

2.2 Functors and natural transformations

Our hierarchy starts with the definition of functors on the category **Set** of sets. The domain and codomain of functors are fixed to the type `Type` of Coq, which can be interpreted as the universe of sets in set-theoretic semantics. Using Hierarchy-Builder, we define functors by the mixin `isFunctor` (line 1). The carrier is a function `F` of type `Type → Type` (line 1) that represents the action on objects and the operator `actm` (line 2) represents the action on morphisms.

```
1 HB.mixin Record isFunctor (F : Type → Type) := {
2   actm : ∀ A B, (A → B) → F A → F B ;
3   functor_id : FunctorLaws.id actm ;  (* actm id = id *)
4   functor_o : FunctorLaws.comp actm }. (* actm (g o h) = actm g o actm h *)
5 #[short(type=functor)]
6 HB.structure Definition Functor := {F of isFunctor F}.
```

The operator `actm` satisfies the functor laws (lines 3 and 4; `FunctorLaws.id` and `FunctorLaws.comp` are definitions whose meaning is indicated as comments). Line 5 prepares a notation for the type of functors. The structure of functors is defined at line 6 as the functions that satisfy the `isFunctor` mixin. Given a functor `F` and a morphism `f`, we note `F # f` the action of `F` on `f`.

We can now create instances of the type `functor`. For example, we can equip `idfun`, the standard identity function of Coq, with the structure of functor by using the `HB.instance` command (line 5 below). It is essentially a matter of proving that the functor laws are satisfied (lines 3, 4):

```
1 Section functorid.
```

```
Let id_actm (A B : Type) (f : A → B) : idfun A → idfun B := f.
Let id_id : FunctorLaws.id id_actm. Proof. by []. Qed.
Let id_comp : FunctorLaws.comp id_actm. Proof. by []. Qed.
HB.instance Definition _ := isFunctor.Build idfun id_id id_comp.
End functorid.
```

As a consequence of this declaration, we can use the Hierarchy-Builder notation `[the functor of idfun]` to invoke the functor corresponding to `idfun`. Since it is often used, we introduce the notation `FId` for this purpose. Similarly, we provide an instance so that `[the functor of F ∘ G]` denotes the composition of two functors `F` and `G`, where ∘ is the standard function composition of Coq.

We now define natural transformations. Given two functors `F` and `G`, we formalize the components of natural transformations as a family of functions `f`, each of type `∀ A, F A → G A` (short notation: `F ⇝ G`) that satisfies the following predicate:

```
Definition naturality (F G : functor) (f : F ⇝ G) :=
  ∀ A B (h : A → B), (G # h) ∘ f A = f B ∘ (F # h).
```

Natural transformations are defined by means of the mixin (line 1) and the structure (line 4) below.

```
1 HB.mixin Record isNatural (F G : functor) (f : F ⇝ G) := {
2  natural : naturality F G f }.
3 #[short(type=nattrans)]
4 HB.structure Definition Nattrans (F G : functor) := {f of isNatural F G f}.
5 Notation "f ⟹ g" := (nattrans f g) : monae_scope.
```

Hereafter, we use the notation `F ⟹ G` (declared at line 5) of natural transformations from the functor `F` to the functor `G`.

2.3 Formalization of monads

We now formalize monads. A monad extends a functor with two natural transformations: the unit `ret` (line 2 below) and the multiplication `join` (line 3). They satisfy three laws (lines 7–9). Furthermore, we add to the mixin an identifier for the bind operator (line 4) and an equation that defines bind in term of unit and multiplication (line 6). Note however that this does not mean that the creation of a new instance of monads requires the (redundant) definition of the unit, multiplication, *and* bind (this will be explained below).

```
1 HB.mixin Record isMonad (F : Type → Type) of Functor F := {
2  ret : FId ⟹ [the functor of F] ;
3  join : [the functor of F ∘ F] ⟹ [the functor of F] ;
4  bind : ∀ A B, F A → (A → F B) → F B ;
5  bindE : ∀ A B (f : A → F B) (m : F A),
6    bind A B m f = join B (([the functor of F] # f) m) ;
7  joinretM : JoinLaws.left_unit ret join ;  (* join ∘ ret (F A) = id *)
8  joinMret : JoinLaws.right_unit ret join ; (* join ∘ F # ret A = id *)
9  joinA : JoinLaws.associativity join }.  (* @join A ∘ F # @join A =
10                                           @join A ∘ @join (F A) *)
11 #[short(type=monad)]
12 HB.structure Definition Monad := {F of isMonad F &}.
```

The fact that a monad extends a functor can be observed at line 1 with the `of` keyword; also, when declaring the structure at line 12, the `&` mark indicates inheritance w.r.t. all the mixins on which the structure depends on. Hereafter, we use `Ret` as a notation for `(@ret _ _)` (the modifier `@` in Coq disables implicit arguments) and ≫= as a notation for `bind`.

The above definition of monads is not the privileged interface to define new instances of monads. We also provide factories with a smaller interface from which the above mixin is recovered. For example, here is the factory to build monads from the unit and the multiplication:

```
HB.factory Record isMonad_ret_join (F : Type → Type) of isFunctor F := {
  ret : FId ⟹ [the functor of F] ;
  join : [the functor of F ∘ F] ⟹ [the functor of F] ;
  joinretM : JoinLaws.left_unit ret join ;
  joinMret : JoinLaws.right_unit ret join ;
  joinA : JoinLaws.associativity join }.
```

This corresponds to the textbook definition of a monad, since it does not require the simultaneous definition of the unit, the multiplication, *and* bind. We use the `HB.builders` command (Sect. 2.1) to show that this lighter definition is sufficient to satisfy the `isMonad` interface.

Similarly, there is a factory to build monads from the unit and bind only:

```
HB.factory Record isMonad_ret_bind (F : Type → Type) of isFunctor F := {
 ret : FId ⟹ [the functor of F] ;
 bind : ∀ A B, F A → (A → F B) → F B ;
 fmapE : ∀ A B (f : A → B) (m : F A),
   ([the functor of F] # f) m = bind A B m (ret B ∘ f) ;
 bindretf : BindLaws.left_neutral bind ret ; (* ret ≫= f = f *)
 bindmret : BindLaws.right_neutral bind ret ; (* m ≫= ret = m *)
 bindA : BindLaws.associative bind }. (* (m ≫= f) ≫= g =
                                         m ≫= (fun x ⇒ f x ≫= g) *)
```

This new definition of monad is an improvement compared to the original formalization [4, Sect. 2.1] because there is now an explicit type of natural transformations (for `ret` and `join`) and because Hierarchy-Builder guarantees that monads instantiated by factories do correspond to the same type `monad`. See [21, file `monad_model.v`] for many instances of the `monad` structure handled by the `isMonad_ret_bind` factory.

2.4 Extending the hierarchy with new monad interfaces

Like we extended the type of functor to the type of monad in the previous section, we can extend the type of monad to the type of nondeterminism monad (by extending the interface of monad with a nondeterministic choice operator and more laws), the type of state monad, the type of exception monad, etc.

We actually ported all monads from our previous work [4] using Hierarchy-Builder; doing this port helped us identify and fix at least one type inference problem (see Sect. 6). In this section, we explain in particular the plus-array monad which is a new addition that we will use in Sect. 5.4 to formalize in-place quicksort.

The array monad The array monad extends a basic monad with a notion of indexed array (see, e.g., [24, Sect. 5.1]). It provides two operators to read and write indexed cells. Given an index `i`, `aget i` returns the value stored at `i` and `aput i v` stores the value `v` at `i`. These operators satisfy the following laws (where `S` is the type of the cells' contents):

```
aputput     ∀ i v v', aput i v ≫ aput i v' = aput i v'
aputget     ∀ i v A (k : S → M A),
              aput i v ≫ aget i ≫= k = aput i v ≫ k v
agetputskip ∀ i, aget i ≫= aput i = skip
agetget     ∀ i A (k : S → S → M A),
              aget i ≫= (fun v ⇒ aget i ≫= k v) =
              aget i ≫= fun v ⇒ k v v
agetC       ∀ i j A (k : S → S → M A),
              aget i ≫= (fun u ⇒ aget j ≫= (fun v ⇒ k u v)) =
              aget j ≫= (fun v ⇒ aget i ≫= (fun u ⇒ k u v))
aputC       ∀ i j u v, (i ≠ j) ∨ (u = v) →
              aput i u ≫ aput j v = aput j v ≫ aput i u
aputgetC    ∀ i j u A (k : S → M A), i ≠ j →
              aput i u ≫ aget j ≫= k =
              aget j ≫= (fun v ⇒ aput i u ≫ k v)
```

For example, `aputput` means that the result of storing the value `v` at index `i` and then storing the value `v'` at index `i` is the same as the result of storing the value `v'`. The law `aputget` means that it is not necessary to get a value after having stored it provided this value is directly passed to the continuation. Other laws can be interpreted similarly.

The extension of the array monad can be simply implemented by extending a basic monad with the following mixin (note that the type of indices is an `eqType`, i.e., a type with decidable equality, as required by the laws of the array monad):

```
HB.mixin Record isMonadArray (S : Type) (I : eqType) (M : Type → Type)
   of Monad M := {
 aget : I → M S ;
 aput : I → S → M unit ;
 aputput : ∀ i s s', aput i s ≫ aput i s' = aput i s' ;
 aputget : ∀ i s (A : Type) (k : S → M A), aput i s ≫ aget i ≫= k =
    aput i s ≫ k s ;
 (* other laws omitted to save space,
     see  [21, file hierarchy.v] for details *) }.
#[short(type=arrayMonad)]
HB.structure Definition MonadArray (S : Type) (I : eqType) :=
 { M of isMonadArray S I M & }.
```

The plus monad We define the plus monad following [26] and [24, Sect. 2]. It extends a basic monad with two operators: failure and nondeterministic choice. These operators satisfy three groups of laws: (1) failure and choice form a monoid, (2) choice is idempotent and commutative, and (3) failure and choice interact with bind according to the following laws (where `[~]` is a notation for nondeterministic choice):

```
left_zero           ∀ A B (f : A → M B), fail A >>= f = fail B
right_zero          ∀ A B (m : M A), m >> fail B = fail B
left_distributivity ∀ A B (m1 m2 : M A) (f : A → M B),
                      m1 [~] m2 >>= f = (m1 >>= f) [~] (m2 >>= f)
right_distributivity ∀ A B (m : M A) (f1 f2 : A → M B),
                      m >>= (fun x ⇒ f1 x [~] f2 x) =
                      (m >>= f1) [~] (m >>= f2)
```

We take advantage of monads already available in MONAE [2] to implement the plus monad with a minimal amount of code while staying conservative. Indeed, we observe that the needed operators and most laws are already available in MONAE. The monads `failMonad` and `failR0Monad` (which inherits from `failMonad` and comes from [3], see Fig. 2) introduce the failure operator, and the `left_zero` and `right_zero` laws. The monad `altMonad` introduces nondeterministic choice and the `left_distributivity` law. The monad `altCIMonad` (which extends `altMonad`) introduces commutativity and idempotence of nondeterministic choice. Finally, `nondetMonad` and `nondetCIMonad` (which is the combination of `altCIMonad` and `nondetMonad`) are combinations of `failMonad` and `altMonad`; these monads are coming from [4]. In other words, only the right-distributivity law is missing.

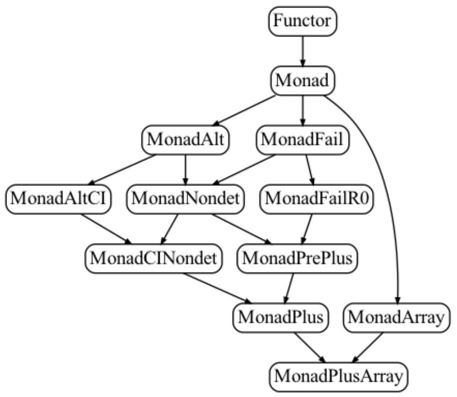

Fig. 2. The hierarchy of monad interfaces discussed in this paper. It is part of a larger hierarchy that can be found online [21].

We therefore implement the `plusMonad` by extending above monads with the right-distributivity law as follows. First, we define the intermediate `prePlusMonad` by adding right-distributivity to the combination of `nondetMonad` and `failR0Monad`[4]. Below, `alt` is the identifier behind the notation `[~]`.

```
HB.mixin Record isMonadPrePlus (M : Type → Type)
   of MonadNondet M & MonadFailR0 M :=
 { alt_bindDr : BindLaws.right_distributive (@bind [the monad of M]) alt }.
#[short(type=prePlusMonad)]
HB.structure Definition MonadPrePlus := {M of isMonadPrePlus M & }.
```

Second, `plusMonad` is defined as the combination of `nondetCIMonad` and `prePlusMonad`:

```
#[short(type=plusMonad)]
HB.structure Definition MonadPlus := {M of MonadCINondet M & MonadPrePlus M}.
```

The plus-array monad Finally, we can combine the array and the plus monads to obtain the `plusArrayMonad` [24, Sect. 5]:

```
#[short(type=plusArrayMonad)]
HB.structure Definition MonadPlusArray (S : Type) (I : eqType) :=
 { M of MonadPlus M & isMonadArray S I M}.
```

To instantiate the interface of the plus-array monad the basic idea is to define `aget` to be the function `fun i a ⇒ [set (a i, a)]` and `aput` to be the function `fun i s a ⇒ [set (tt, insert i s a)]`, where `[set _]` is a notation for singleton sets and `insert i s a` is `fun j ⇒ if i == j then s else a j`. One of course needs to instantiate the interfaces of Fig. 2 (except `MonadCINondet`, `MonadPlus`, and `MonadPlusArray` that are just joined interfaces). See [21, file `monad_model.v`] for details.

2.5 Monad transformers using Hierarchy-Builder

As another illustration of the ease to add monad constructs using Hierarchy-Builder, we explain how we formalize monad transformers. This improves the formalization of monad transformers from previous work [3, Sections 3.3–3.4] with more readable formal definitions and more robust type inference.

Given two monads `M` and `N`, a monad morphism is a function `M` $\rightsquigarrow$ `N` that satisfies the laws of monad morphisms [7, Def. 19] [16, Def. 7]:

```
HB.mixin Record isMonadM (M N : monad) (e : M ⇝ N) := {
  monadMret : MonadMLaws.ret e ; (*  e ∘ Ret = Ret *)
  monadMbind : MonadMLaws.bind e (* e (m ≫= f) = e m ≫= (e ∘ f) *) }.
```

An important property of monad morphisms is that they are natural transformations, so that we define the type of monad morphism using the interfaces of monad morphisms and natural transformations (Sect. 2.2):

[4] The existence of this intermediate interface is further justified by its use in the definition of the interface of the backtrackable state monad [14].

```
#[short(type=monadM)]
HB.structure Definition MonadM (M N : monad) :=
  {e of isMonadM M N e & isNatural M N e}.
```

However, since the laws of monad morphisms imply naturality, a monad morphism can be defined directly by a factory with *only* the laws of monad morphisms:

```
HB.factory Record isMonadM_ret_bind (M N : monad) (e : M ⇝ N) := {
  monadMret : MonadMLaws.ret e ;
  monadMbind : MonadMLaws.bind e }.
```

Like for monads in Sect. 2.3, we are again in the situation where the textbook definition ought better be sought in factories.

A monad transformer `t` is a function from `monad` to `monad` such that for any monad `M` there is a monad morphism from `M` to `t M` [7, Sect. 3.3] [16, Def. 9]:

```
HB.mixin Record isMonadT (t : monad → monad) := {
  Lift : ∀ M, monadM M (t M) }.
#[short(type=monadT)]
HB.structure Definition MonadT := {t of isMonadT t}.
```

Going one step further, we define *functorial monad transformers* [16, Def. 20]. A functorial monad transformer is a monad transformer `t` with an `hmap` operator of type `∀ M N : monad, (M ⟹ N) → t M ⟹ t N` (i.e., `hmap` preserves natural transformations) with additional properties. First, `hmap` respects identities of natural transformations (line 4) and (vertical) composition of natural transformations (line 6).

```
1 HB.mixin Record isFunctorial (t : monad → monad) := {
2   hmap : ∀ {M N : monad}, (M ⟹ N) → t M ⟹ t N ;
3   functorial_id : ∀ M : monad,
4     hmap [the _ ⟹ _ of NId M] = [the _ ⟹ _ of NId (t M)] ;
5   functorial_o : ∀ (M N P : monad) (t : M ⟹ N) (s : N ⟹ P),
6     hmap (s \v t) = hmap s \v hmap t }.
7 #[short(type=functorial)]
8 HB.structure Definition Functorial := {t of isFunctorial t}.
```

See [21, file `monad_lib.v`] for the definitions of the identity natural transformation `NId` and of vertical composition `\v`. Second, `hmap` preserves monad morphisms (lines 2–5) and the `Lift` operator of the monad transformer is natural (line 7):

```
1 HB.mixin Record isFMT (t : monad → monad) of MonadT t & Functorial t := {
2   fmt_ret : ∀ M N (e : monadM M N),
3     MonadMLaws.ret (hmap [the functorial of t] e) ;
4   fmt_bind : ∀ M N (e : monadM M N),
5     MonadMLaws.bind (hmap [the functorial of t] e) ;
6   natural_hmap : ∀ (M N : monad) (n : M ⟹ N),
7     hmap [the functorial of t] n \v Lift [the monadT of t] M =
8     Lift [the monadT of t] N \v n }.
9 HB.structure Definition FMT := {t of isFMT t & }.
```

Using the above formal definitions, we have been able to produce several instances of monad transformers (state, exception, environment, continuation, etc.) and functorial monad transformers, and revise a formalization of Jaskelioff's modular monad transformers [16]; see [21, directory `impredicative_set`].

3 Difficulties with the termination of monadic functions

In the context of monadic equational reasoning, we observe that we often run into difficulties when proving properties of monadic functions because their type is not informative enough. This happens in particular when proving termination. For example, in their derivations of quicksort, Mu and Chiang postulate the termination of several functions using the Agda pragma *`{-# TERMINATING #-}`*, which is not safe in general [5]. Before discussing practical solutions in the next section (Sect. 4), we provide in this section background information about the standard Coq tooling to prove termination in Sect. 3.1 and explain concrete examples of difficulties in Sect. 3.2.

3.1 Background: standard Coq tooling to prove termination

Functions defined in a proof assistant based on dependent types need to terminate to preserve logical consistency.

The `Equations` command In Coq, the `Equations` command [29, 30] provides support to prove the termination of functions whose recursion is not structural. For example, functional quicksort can be written as follows (the type `T` can be any ordered type [20]):

```
Equations? qsort (s : seq T) : seq T by wf (size s) lt :=
| [::] ⇒ [::]
| h :: t ⇒ qsort (partition h t).1 ++ h :: qsort (partition h t).2.
```

The function call `partition h t` returns a pair of lists (say, `ys` and `zs`) that partitions the list `t` w.r.t. the pivot `h` (the notations `.1` and `.2` are for taking the first and second projection of a pair). The annotation `by wf (size s) lt` indicates that the relation between the sizes of lists is well-founded (`lt` is the comparison for natural numbers). Once the `Equations?` declaration of `qsort` is processed, Coq asks for a proof that the arguments are indeed decreasing, that is, proofs of `size ys < size s` and `size zs < size s`. Under the hood, Coq uses the accessibility predicate [8, Chapter 15].

At first sight, the approach using `Equations` is appealing: the syntax is minimal and, as a by-product, it automatically generates additional useful lemmas, e.g., in the case of `qsort`, one equation for each branch (`qsort_equation_{1,2}`) and a lemma capturing the fixpoint equation (`qsort_elim`).

The Program/Fix approach The Program/Fix approach is more primitive and verbose than the Equations approach, but it is also more flexible and robust to changes because it relies on less automation. It is a combination of the Program command for dependent type programming [31, Chapter Program] and of the Fix definition from the Coq.Init.Wf module for well-founded fixpoint of the standard library. For the sake of explanation, let us show how to define functional quicksort using this approach.

First, one defines an intermediate function qsort' similar to the declaration that one would write with the Equations command except that its recursive calls are to a parameter function (f below). This parameter function takes as an additional argument a proof that the measure (here the size of the input list) is decreasing. These proofs appear as holes (_ syntax) to be filled next by the user[5]:

```
Program Definition qsort' (s : seq T)
  (f : ∀ s', (size s' < size s) → seq T) : seq T :=
  if s isn't h :: t then [::] else
  let: (ys, zs) := partition h t in f ys _ ++ h :: f zs _.
```

Second, one defines the actual qsort function using Fix. This requires a (trivial) proof that the order chosen for the measure is well-founded:

```
Definition qsort : seq T → seq T :=
  Fix (@well_founded_size _) (fun _ ⇒ _) qsort'.
```

The Program/Fix approach does not generate any helper lemmas, it is essentially the manual version of the Equations approach but it is more widely applicable as we will see in Sect. 5.4.

3.2 Limitations of Coq standard tooling to prove termination

We now illustrate some limitations of Coq standard tooling with a monadic function that computes permutations nondeterministically. The function perm below (written in Agda) is not the most obvious definition for this task but it is a good fit to specify quicksort and it is used as such by Mu and Chiang to perform program derivation [24, Sect. 3].

```
split : {{_ : MonadPlus M}} → List A → M (List A × List A)
split [] = return ([] , [])
split (x :: xs) = split xs >>=
  λ {(ys, zs) → return (x :: ys, zs) || return (ys, x :: zs)}

{-# TERMINATING #-}
perm : {{_ : MonadPlus M}} → List A → M (List A)
perm [] = return []
perm (x :: xs) = split xs >>=
  λ { (ys , zs) → liftM2 (_++[ x ]++_) (perm ys) (perm zs) }
```

[5] The Program command can be configured by the user so that Coq provides proofs automatically.

The function `split` splits a list nondeterministically. The notation || corresponds to nondeterministic choice (in Monae, this is the notation `[~]` that we already saw in Sect. 2.4). The function `perm` uses `split` and `liftM2`, a generic monadic function that lifts a function `h : A -> B -> C` to a monadic function of type `M A -> M B -> M C`.

First, we observe that since `split` is structurally recursive, it can be encoded directly in Coq as a `Fixpoint` (using the `altMonad` of Sect. 2.4) [6]:

```
Fixpoint splits {M : altMonad} A (s : seq A) : M (seq A * seq A) :=
  if s isn't x :: xs then Ret ([::], [::])
  else splits xs ≫= (fun '(ys, zs) ⇒
    Ret (x :: ys, zs) [~] Ret (ys, x :: zs)).
```

Applying the `Equations` approach to define `perm` (we call it `qperm` in Coq for the sake of clarity) does not fail immediately but the termination proof cannot be completed. Here follows the definition of `qperm`:

```
Equations? qperm (s : seq A) : M (seq A) by wf (size s) lt :=
| [::] ⇒ Ret [::]
| x :: xs ⇒
  splits xs ≫= (fun '(ys, zs) ⇒
      liftM2 (M := M) (fun a b ⇒ a ++ x :: b) (qperm ys) (qperm zs)).
```

As expected, Coq asks the user to prove that the size of the list is decreasing. The first generated subgoal is:

```
 qperm: ∀ s : seq A, size s < (size xs).+1 → M (seq A)
 x: A
 xs: seq A
 ys, zs: seq A
==============================================
 size ys < (size xs).+1
```

There is no way to prove this goal since there is no information about the list `ys` (the same is true for `zs`). The same problem happens with the `Program`/`Fix` approach.

4 Add dependent types to return types for formal proofs

The difficulty explained in the previous section is no surprise to the practitioner. It is known that such formal proofs can be completed by enriching the return type of functions with appropriate dependent types. However, in practice this is bothersome enough so that one is tempted to resort to axioms/postulates (e.g., [24]). In face of such a technical difficulty, we advocate the following pragmatic approach: use a standard dependent type if available and instrumented (Sect. 4.1), otherwise use a *dependently-typed assertion* (Sect. 4.2).

[6] Since Coq already has a `split` tactic, we call the function `splits`.

4.1 Add dependent types to called functions to prove termination

The first idea is to use standard dependent types to augment the return type of functions so that the `Equations` approach succeeds.

Let us explain how to prove the termination of the `qperm` function of Sect. 3.2. The `splits` function is defined such that its return type is `M (seq A * seq A)`. We add information about the size of the returned lists by providing another version of `splits` whose return type is `M ((size s).-bseq A * (size s).-bseq A)`, where `s` is the input list and `n.-bseq A` is the type of lists of size less than or equal to `n`. This type of "bounded-size lists" comes from the MathComp library [20].

```
Fixpoint splits_bseq {M : altMonad} A (s : seq A)
    : M ((size s).-bseq A * (size s).-bseq A) :=
  if s isn't x :: xs then Ret ([bseq of [::]], [bseq of [::]])
  else splits_bseq xs >>= (fun '(ys, zs) =>
    Ret ([bseq of x :: ys], widen_bseq (leqnSn _) zs) [~]
    Ret (widen_bseq (leqnSn _) ys, [bseq of x :: zs])).
```

The body of this definition is the same as the original one provided one ignores the notations and lemmas about bounded-size lists. The notation `[bseq of [::]]` is for an empty list seen as a bounded-size list. The lemma `widen_bseq` captures the fact that a `m.-bseq T` list can be seen as a `n.-bseq T` list provided that `m ≤ n`:

```
Lemma widen_bseq T m n : m ≤ n -> m.-bseq T -> n.-bseq T.
```

Since `leqnSn n` is a proof of `n ≤ n.+1`, we understand that `widen_bseq (leqnSn _)` turns a `n.-bseq A` list into a `n.+1.-bseq A` list. The notation `[bseq of x :: ys]` is a MathComp idiom that triggers automation canonical structures to build a `n.+1.-bseq A` list using the fact that `ys` is itself a `n.-bseq A` list.

Since there is a coercion from lists to bounded-size lists, we can define `qperm` like in Sect. 3.2 but using `splits_bseq` instead of `splits`:

```
Equations? qperm (s : seq A) : M (seq A) by wf (size s) lt :=
| [::] => Ret [::]
| x :: xs => splits_bseq xs >>= (fun '(ys, zs) =>
  liftM2 (M := M) (fun a b => a ++ x :: b) (qperm ys) (qperm zs)).
```

The proofs required by Coq now contain in their local context the additional information that the lists `ys` and `zs` are of type `(size xs).-bseq A`, which allows for completing the termination proof.

The nonderministic computation of permutations using nondeterministic selection is another example of the use of bounded-size lists [14, Sect. 4.4] (see [21, file `fail_lib.v`]).

This use of bounded-size lists to prove termination is reminiscent of *sized types* which have already been shown to be useful to guarantee the termination of programs such as quicksort [6]. However, as of today, it appears that users of proof assistants still need a support library to prove termination manually: though Agda as long been providing sized types, they are not considered safe anymore since Agda 2.6.2 and there are indications that sized types for Coq might not be practical [9].

4.2 Add dependent types with a dependently-typed assertion

The approach explained in the previous section is satisfactory when the needed type is already available and instrumented in some standard library. Otherwise, one needs to define a new, potentially ad hoc type, instrument it with lemmas, possibly with a new notation, etc. Yet, we can reach a similar result without this boilerplate using *dependently-typed assertions*.

For the fail monad `M`, it is customary to define assertions as follows. A computation `guard b` of type `M unit` fails or skips according to a boolean value `b`:

```
Definition guard {M : failMonad} b : M unit := if b then skip else fail.
```

An assertion `assert p a` is a computation of type `M A` that fails or returns `a` according to whether `p a` is true or not (`pred` is the type of boolean predicates in MathComp):

```
Definition assert {M : failMonad} A (p : pred A) a : M A :=
  guard (p a) ≫ Ret a.
```

Similarly, we define a dependently-typed assertion that fails or returns a value *together with a proof* that the predicate is satisfied:

```
Definition dassert {M : failMonad} A (p : pred A) a : M { a | p a } :=
  if Bool.bool_dec (p a) true is left pa then Ret (exist _ _ pa)
  else fail.
```

We illustrate the alternative approach of using `dassert` with a non-trivial property of the `qperm` function: the fact that it preserves the size of its input (this is a postulate in [24]). This statement uses the generic `preserves` predicate:

```
Definition preserves {M : monad} A B (f : A → M A) (g : A → B) :=
  ∀ x, (f x ≫= fun y ⇒ Ret (y, g y)) = (f x ≫= fun y ⇒ Ret (y, g x)).
Lemma qperm_preserves_size {M : prePlusMonad} A :
  preserves (@qperm M A) size.
```

In the course of proving `qperm_preserves_size` (by strong induction of the size of the input list), we run into the following subgoal:

```
s : seq A
ns : size s < n
m:=fun '(ys, zs) ⇒ liftM2 (fun a b ⇒ a ++ p :: b) (qperm ys) (qperm zs)
====================
splits s ≫= (fun x ⇒ m x ≫= (fun y ⇒ Ret (y, size y))) =
splits s ≫= (fun x ⇒ m x ≫= (fun y ⇒ Ret (y, (size s).+1)))
```

If we use the extensionality of bind to make progress (by applying the tactic `bind_ext ⇒ -[a b].`), we add to the local context two lists `a` and `b` that correspond to the output of `splits`:

```
s : seq A
ns : size s < n
m:=fun '(ys, zs) ⇒ liftM2 (fun a b ⇒ a ++ p :: b) (qperm ys) (qperm zs)
a, b : seq A
```

```
====================
m (a, b) ≫= (fun y ⇒ Ret (y, size y)) =
m (a, b) ≫= (fun y ⇒ Ret (y, (size s).+1))
```

As in Sect. 3.2, we cannot make progress because there is no size information about `a` and `b`. Instead of introducing a new variant of `splits`, we use `dassert` and bind to augment the return type of `splits` with the information that the concatenation of the returned lists is of the same size as the input (definition `dsplitsT` below):

```
Definition dsplitsT A n :=
  {x : seq A * seq A | size x.1 + size x.2 == n}.
Definition dsplits
    {M : nondetMonad} A (s : seq A) : M (dsplitsT A (size s)) :=
  splits s ≫= dassert [pred n | size n.1 + size n.2 == size s].
```

The equivalence between `splits` and `dsplits` can be captured by an application of `fmap` (`fmap f` is a notation for `_ # f`, the functor is inferred automatically) that projects the witness of the dependent type (`sval` returns the witness of a dependent pair):

```
Lemma dsplitsE {M : prePlusMonad} A (s : seq A) :
  splits s = fmap (fun x ⇒ ((sval x).1, (sval x).2)) (dsplits s) :> M _.
```

We can locally introduce `dsplits` using the lemma `dsplitsE` to complete our proof of `qperm_preserves_size`. Once `dassert` is inserted in the code, we can use the following lemma to lift the assertions to the local proof context:

```
Lemma bind_ext_dassert
    {M : failMonad} A (p : pred A) a B (m1 m2 : _ → M B) :
  (∀ x h, p x → m1 (exist _ x h) = m2 (exist _ x h)) →
  dassert p a ≫= m1 = dassert p a ≫= m2.
```

This leads us to a local proof context where the sizes of the output lists are related to the input list `s` with enough information to complete the proof:

```
s : seq A
ns : size s < n
m:=fun '(ys, zs) ⇒ liftM2 (fun a b ⇒ a ++ p :: b) (qperm ys) (qperm zs)
a, b : seq A
ab : size a + size b == size s
====================
m (a, b) ≫= (fun y ⇒ Ret (y, size y)) =
m (a, b) ≫= (fun y ⇒ Ret (y, (size s).+1))
```

See [21, file `example_iquicksort.v`] for the complete script.

Although we use here a dependently-typed assertion to prove a lemma, we will see in Sect. 5.4 an example of termination proof where `dassert` also comes in handy. Nevertheless, `dassert` requires to work with a monad that provides at least the failure operator.

5 A complete formalization of quicksort derivation

In this section, we apply the library support we explained so far to prove postulates left by Mu and Chiang in their formalization of quicksort derivations [24]. Beforehand we need to complete our theory of computations of nondeterministic permutations (Sect. 5.1). Then we will explain the key points of specifying and proving functional quicksort (Sect. 5.3) and in-place quicksort (Sect. 5.4). These proofs rely on the notion of refinement (Sect. 5.2).

5.1 Formal properties of nondeterministic permutations

The specifications of quicksort by Mu and Chiang rely on the properties of nondeterministic permutations as computed by `qperm` (Sect. 3.2). The shape of this function makes proving its properties painful, intuitively because of two non-structural recursive calls and the interplay with the properties of `splits`. As a matter of fact, Mu and Chiang postulates many properties of `qperm` in their Agda formalization, e.g., its idempotence (here stated using the Kleisli symbol):

```
Lemma qperm_idempotent {M : plusMonad} (E : eqType) :
  qperm >=> qperm = qperm :> (seq E → M (seq E)).
```

The main idea to prove these postulates is to work with a simpler definition of nondeterministic permutations, namely `iperm`, defined using nondeterministic insertion:

```
Fixpoint insert {M : altMonad} A (a : A) (s : seq A) : M (seq A) :=
  if s isn't h :: t then Ret [:: a] else
  Ret (a :: h :: t) [~] fmap (cons h) (insert a t).
Fixpoint iperm {M : altMonad} A (s : seq A) : M (seq A) :=
  if s isn't h :: t then Ret [::] else iperm t ≫= insert h.
```

Since `insert` and `iperm` each consist of one structural recursive call, their properties can be established by simple inductions, e.g., the idempotence of `iperm`:

```
Lemma iperm_idempotent {M : plusMonad} (E : eqType) :
  iperm >=> iperm = iperm :> (seq E → M _).
```

The equivalence between `iperm` and `qperm` can be proved easily by first showing that the recursive call to `iperm` can be given the same shape as `qperm`:

```
Lemma iperm_cons_splits (A : eqType) (s : seq A) u :
  iperm (u :: s) = do a ← splits s; let '(ys, zs) := a in
                   liftM2 (fun x y ⇒ x ++ u :: y) (iperm ys) (iperm zs).
```

We can use this last fact to show that `iperm` and `qperm` are equivalent

```
Lemma iperm_qperm {M : plusMonad} (A : eqType) : @iperm M A = @qperm M A.
```

Thanks to `iperm_qperm`, all the properties of `iperm` can be transported to `qperm`, providing formal proofs for several postulates from [24] (see Table 1).

Definition/lemma in [24]		Coq equivalent in [21] or in this paper
file `Implementation.agda`		
`ext`	postulate	standard axiom of functional extensionality
file `Monad.agda`		
`write-write-swap`	postulate	not used
`writeList-++`	postulate	`writeList_cat` (`array_lib.v`)
`writeList-writeList-comm`	postulate	`writeListC` (`array_lib.v`)
file `Nondet.agda`		
`return⊑perm`	postulate	`refin_qperm_ret` (`fail_lib.v`)
`perm-idempotent`	postulate	Sect. 5.1
`perm-snoc`	postulate	`qperm_rcons` (`fail_lib.v`)
`sorted-cat3`	postulate	Sect. 5.3
`perm-preserves-length`	postulate	Sect. 4.2
`perm-preserves-all`	postulate	Sect. 5.3
`perm`	TERMINATING	Sect. 4.1
`mpo-perm`	TERMINATING	commutation of computations (Sect. 5.3)
`partl/partl-spec`	TERMINATING	`partl` (`example_iqsort.v`), solved by currying
`partl'/partl'-spec`	TERMINATING	`qperm_partl` (`example_iqsort.v`)
`mpo-partl'`	TERMINATING	commutation of computations (Sect. 5.3)
`qsort/qsort-spec`	TERMINATING	Sect. 3.1
file `IPartl.agda`		
`ipartl/ipartl-spec`	TERMINATING	Sect. 5.4, solved by currying
`introduce-swap` [24, eqn 11]	postulate	`refin_writeList_rcons_aswap` (`array_lib.v`)
`introduce-read`	postulate	not used
file `IQSort.agda`		
`iqsort/iqsort-spec`	TERMINATING	Sect. 4.2
`introduce-read`	postulate	`writeListRet` (`array_lib.v`)
`introduce-swap` [24, eqn 13]	postulate	`refin_writeList_cons_aswap` (`array_lib.v`)

Table 1. Admitted facts in [24] and their formalization in [21] (Lemmas `xyz-spec` require the `TERMINATING` pragma as a consequence of the function `xyz` being postulated as terminating.)

5.2 Program refinement

The rest of this paper uses a notion of program refinement introduced by Mu and Chiang [24, Sect. 4]. This is about proving that two programs obey the following relation:

```
Definition refin {M : altMonad} A (m1 m2 : M A) : Prop := m1 [~] m2 = m2.
Notation "m1 ⊆ m2" := (refin m1 m2).
```

As the notation symbol indicates, it represents a relationship akin to set inclusion, which means that the result of `m1` is included in that of `m2`. We say that `m1` *refines* `m2`. The refinement relation is lifted as a pointwise relation as follows:

```
Definition lrefin {M : altMonad} A B (f g : A → M B) := ∀ x, f x ⊆ g x.
Notation "f ⊆̇ g" := (lrefin f g).
```

5.3 A complete formalization of functional quicksort

We explain how we formalize quicksort as a function as Mu and Chiang did [24], proving in Coq the few axioms they left in their Agda formalization.

What we actually prove is that the sort algorithm implemented by the `qsort` function of Sect. 3.1 refines an algorithm that is obviously correct. The algorithm in question is `slowsort`: a function that filters only the sorted permutations of all permutations derived by `qperm`, which is obviously correct as a sorting algorithm:

```
Definition slowsort {M : plusMonad} T : seq T → M (seq T) :=
  qperm >=> assert sorted.
```

Using the refinement relation, the specification that `qsort` should meet can be written as follows:

```
Lemma qsort_spec : Ret ∘ qsort ⊆̇ slowsort.
```

The axioms left by Mu and Chiang that we prove are either about termination or about equational reasoning. As for the former, we have explained the termination of `qperm` and `qsort` in Sect. 4. As for the axioms about equational reasoning, the main[7] one is `perm-preserves-all` which is stated as follows (using the Agda equivalent of the `preserves` predicate we saw in Sect. 4.2):

```
postulate
  perm-preserves-all : {{_ : MonadPlus M}} {{_ : Ord A}}
                     → (p : A → Bool) → perm preserves (all p)
```

This lemma says that all the permutations that result from `perm` preserve the fact that all the elements satisfy `p` or not. In Coq, we proved the equivalent (using `guard`) rewrite lemma `guard_all_qperm`:

```
Lemma guard_all_qperm
    {M : plusMonad} T B (p : pred T) s (f : seq T → M B) :
  qperm s ≫= (fun x ⇒ guard (all p s) ≫ f x) =
  qperm s ≫= (fun x ⇒ guard (all p x) ≫ f x).
```

The proof of `guard_all_qperm` is not trivial: it is carried out by strong induction, requires the intermediate use of the dependently-typed version of `splits` (Sect. 4.1), and more crucially because it relies on the fact that `guard` commutes with computations in the plus monad. This latter fact is captured by the following lemma:

```
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)).
Lemma commute_plus_guard
    {M : plusMonad} b B (n : M B) C (f : unit → B → M C) :
  commute (guard b) n f.
```

[7] There is another axiom `sorted-cat3`, but its proof is easy using lemmas from MathComp.

Its proof uses induction on syntax as explained in [4, Sect. 5.1].

Our formalization is shorter than Mu and Chiang's. It is not really fair to compare the total size of both formalizations in particular because the proof style in Agda is verbose (all the intermediate goals are spelled out). Yet, with SSReflect, we manage to keep each intermediate lemmas under the size of 15 lines. For example, the intermediate lemma `slowsort'-spec` in Agda is about 170 lines, while our proof in Coq is written in 15 lines (see `partition_slowsort_spec` [21]), which arguably is more maintainable.

5.4 A complete formalization of in-place quicksort

We now explain how we formalize the derivation of in-place quicksort by Mu and Chiang [24]. The first difficulty is to prove termination, which Mu and Chiang postulate (see Table 1).

Let us first explain the original Agda implementation. The partition step is performed by the function `ipartl`, which uses the array monad (Sect. 2.4):

```
{-# TERMINATING #-}
ipartl : {{_ : Ord A}} {{_ : MonadArr A M}} →
 A → ℕ → (ℕ × ℕ × ℕ) → M (ℕ × ℕ)
ipartl p i (ny, nz, 0) = return (ny, nz)
ipartl p i (ny, nz, suc k) = read (i + ny + nz) >>= λ x →
 if x ≤ᵇ p then swap (i + ny) (i + ny + nz) >> ipartl p i (ny + 1, nz, k)
           else ipartl p i (ny, nz + 1, k)
 where open Ord.Ord {{...}}
```

The call `ipartl p i (ny, nz, nx)` partitions the subarray ranging from index `i` (included) to `i + ny + nz + nx` (excluded) and returns the sizes of the two partitions. For the sake of explanation, we can think of the contents of this subarray as a list `ys ++ zs ++ xs`; `ys` and `zs` are the two partitions and `xs` is yet to be partitioned; `ny` and `nz` are the sizes of `ys` and `zs`. At each iteration, the first element of `xs` (i.e., the element at index `i + ny + nz`) is read and compared with the pivot `p`. If it is smaller or equal, it is swapped with the element following `ys` and partition proceeds with a `ys` enlarged by one element. (The `swap` function uses the read/write operators of the array monad to swap two cells of the array.) Otherwise, partition proceeds with a `zs` enlarged by one element.

The quicksort function `iqsort` takes an index and a size; it is a computation of the unit type. The code selects a pivot (line 5), calls `ipartl` (line 6), swaps two cells (line 7) and then recursively calls itself on the partitioned arrays:

```
1 {-# TERMINATING #-}
2 iqsort : {{_ : Ord A}} {{_ : MonadArr A M}} → ℕ → ℕ → M ⊤
3 iqsort i 0 = return tt
4 iqsort i (suc n)  =
5   read i >>= λ p →
6   ipartl p (i + 1) (0 , 0 , n) >>= λ { (ny , nz) →
7   swap i (i + ny) >>
8   iqsort i ny >> iqsort (i + ny + 1) nz }
```

We encode this definition in Coq and prove its termination as explained in Sect. 4.2. First, observe that the termination of the function `ipartl` need not be postulated: its curried form is accepted by Agda and Coq because the recursion is structural. Let us define `iqsort` in Coq using the `Program`/`Fix` approach (Sect. 3.1). The direct definition fails for the same reasons as explained in Sect. 3: it turns out that the termination proof requires more information about the relation between the input and the output of `ipartl` than the mere fact that it is a pair of natural numbers. We therefore introduce a dependently-typed version of `ipartl` that extends its return type to a dependent pair of type `dipartlT`:

```
Definition dipartlT y z x :=
  {n : nat * nat | (n.1 ≤ x + y + z) ∧ (n.2 ≤ x + y + z)}.
```

The parameters `x`, `y`, and `z` are the sizes of the lists input to `ipartl`; this dependent type ensures that the sizes returned by the partition function are smaller than the size of the array being processed. The dependently-typed version of `ipartl` is obtained by means of `dassert` (Sect. 4.2)[8]:

```
Definition dipartl
    {M : plusArrayMonad T Z_eqType} p i y z x : M (dipartlT y z x) :=
  ipartl p i y z x >>=
  dassert [pred n | (n.1 ≤ x + y + z) ∧ (n.2 ≤ x + y + z)].
```

Using `dipartl` instead of `ipartl` allows us to complete the definition of `iqsort` (the notation `%:Z` is for injecting natural numbers into integers):

```
Program Definition iqsort' {M : plusArrayMonad E Z_eqType} ni
    (f : ∀ mj, mj.2 < ni.2 → M unit) : M unit :=
  match ni.2 with
  | 0 ⇒ Ret tt
  | n.+1 ⇒ aget ni.1 >>= (fun p ⇒
            dipartl p (ni.1 + 1) 0 0 n >>= (fun nynz ⇒
              let ny := nynz.1 in let nz := nynz.2 in
              aswap ni.1 (ni.1 + ny%:Z) >>
              f (ni.1, ny) _ >> f (ni.1 + ny%:Z + 1, nz) _))
  end.
```

See [21, file `example_iquicksort.v`] for the complete termination proof. Note that our in-place quicksort is a computation in the plus-array monad which is the only array monad that provides the failure operator in our hierarchy. Anyway, the following refinement proof requires the plus-array monad. We have not been able to use the `Equations` approach here; it seems that the default setting does not give us access to the proof that we introduced through dependent types.

The specification of in-place quicksort uses the same `slowsort` function as for functional quicksort (Sect. 5.3):

```
Lemma iqsort_slowsort {M : plusArrayMonad E Z_eqType} i xs :
  writeList i xs >> iqsort (i, size xs) ⊆ slowsort xs >>= writeList i.
```

[8] The type `Z_eqType` is the type of integers equipped with decidable equality. This is a slight generalization of the original definition that is using natural numbers.

The function `writeList i xs` writes all the elements of the `xs` list to the array starting from the index `i`. This is just a recursive application of the `aput` operator we saw in Sect. 2.4:

```
Fixpoint writeList {M : arrayMonad T Z_eqType} i s : M unit :=
  if s isn't x :: xs then Ret tt else aput i x ≫ writeList (i + 1) xs.
```

Most of the derivation of in-place quicksort is explained by Mu and Chiang in their paper [24]. In fact, we did not need to look at the accompanying Agda code except for the very last part which is lacking details [24, Sect. 5.3]. Our understanding is that the key aspect of the derivation (and of the proof of `iqsort_slowsort`) is to show that the function `ipartl` refines a simpler function `partl` that is a slight generalization of `partition` used in the definition of functional quicksort (Sect. 3.1). In particular, this refinement goes through an intermediate function that fusions `qperm` (Sect. 4.1) with `partl`; this explains the importance of the properties of idempotence of `qperm` whose proof we explained in Sect. 5.1.

6 Related work

The hierarchy of interfaces we build in Sect. 2 is a reimplementation and an extension of previous work [3,4]. The latter was built using packed classes written manually. The use of Hierarchy-Builder is a significant improvement: it is less verbose and easier to extend as seen in Sections 2.4 and 2.5. It is also more robust. Indeed, we discovered that previous work [3, Fig. 1] lacked an intermediate interface, which required us to insert some type constraints for type inference to succeed (see [15] for details). Hierarchy-Builder detects such omissions automatically. Type classes provide an alternative approach for the implementation of a hierarchy of monad interfaces and it has been used to a lesser extent in some related work (e.g., [24]).

The examples used in this paper stem from the derivations of quicksort by Mu and Chiang [24]. Together with their paper, the authors provide an accompanying formalization in Agda. It contains axiomatized facts (see Table 1) that are arguably orthogonal to the issue of quicksort derivation but that reveals issues that need to be addressed to improve formal monadic equational reasoning in practice. In this paper, we explained in particular how to complete their formalization, which we actually rework from scratch, favoring equational reasoning and the creation of reusable lemmas; in other words, our formalization is not a port.

To complete Mu and Chiang's formalization, we needed in particular to formalize a thorough theory of nondeterministic permutations (see Sect. 5.1) It turns out that this is a recurring topic of monadic equational reasoning. They are written in different ways depending on the target specification: using nondeterministic selection [14, Sect. 4.4], using nondeterministic selection and the function `unfoldM` [22, Sect. 3.2], using nondeterministic insertion [23, Sect. 3], or using `liftM2` [24, Sect. 3]. The current version of Monae has now a formalization of each.

Sakaguchi provides a formalization of the quicksort algorithm in Coq using the array state monad [27, Sect. 6.2]. His formalization is primarily motivated by the generation of efficient executable code. This makes for an intricate definition of quicksort (for example, all the arguments corresponding to indices are bounded). Though his framework does not prevent program verification [27, Sect. 4], it seems difficult to reuse it for monadic equational reasoning (the type of monads is specialized to state/array and there is no hierarchy of monad interfaces).

This paper is focusing on monadic equational reasoning but this is not the only way to verify effectful programs using monads in Coq. For example, Jomaa et al. have been using a Hoare monad to verify properties of memory isolation [17]. They are therefore only dealing with the effects of state and exception. Maillard et al. have been developing a framework to verify programs with effects using Dijkstra monads [19]. Their Dijkstra monads are based on specification monads and are built using monad morphisms. Verification of a monadic computation amounts to type it in Coq with the appropriate Dijkstra monad. Christiansen et al. have been verifying effectful Haskell programs in Coq [10] and Letan et al. have been exploring verification in Coq of impure computations using a variant of the free monad [18].

The formalization of monads we explained in Sect. 2 is specialized to the category **Set** of sets. MONAE also features a formalization of (concrete) categories that has been used to formalize the geometrically convex monad [2, Sect. 5]. Both are connected in the sense that a monad over the category corresponding to the type `Type` of Coq (seen as a Grothendieck universe) can be used to instantiate the `isMonad` interface. Yet, as far as this paper is concerned, this generality is not useful.

7 Conclusion

In this paper, we reported on practical advances about formalization of monadic equational reasoning, illustrated by a complete formalization of functional quicksort and in-place quicksort. For that purpose, we improved an existing Coq library called MONAE. To ease the addition of new monad constructs, we reimplemented the hierarchy of interfaces of MONAE using HIERARCHY-BUILDER and illustrated this extension with the plus-array monad and monad transformers. We observed that the shallow embedding of monadic functions is a source of a recurring technical issue making some proofs (in particular, termination proofs) bothersome; we argued that appropriate library extensions using dependent types are useful in practice to complement Coq's standard tooling. We applied these techniques to the formal verification of quicksort derivations by Mu and Chiang that we were able to formalize without admitted facts.

As a result of the above experiment, we have substantially improved the MONAE library for formalization of monadic equational reasoning. As for future work, we plan to further enrich the hierarchy of interfaces and to apply MONAE to other formalization experiments (e.g., [26, 28]). We also plan to investigate the use of MONAE as a back-end for the formal verification of Coq programs, for example as generated automatically from OCaml [13].

Acknowledgements The authors would like to thank Cyril Cohen and Enrico Tassi for their assistance with HIERARCHY-BUILDER, Kazuhiko Sakaguchi and Takafumi Saikawa for their comments, the members of the Programming Research Group of the Department of Mathematical and Computing Science at the Tokyo Institute of Technology for their input, and the anonymous reviewers for many comments that improved this paper. The second author acknowledges the support of the JSPS KAKENHI grants 18H03204 and 22H00520.

References

1. Affeldt, R., Cohen, C., Kerjean, M., Mahboubi, A., Rouhling, D., Sakaguchi, K.: Competing inheritance paths in dependent type theory: A case study in functional analysis. In: 10th International Joint Conference on Automated Reasoning (IJCAR 2020), Paris, France, July 1–4, 2020, Part II. Lecture Notes in Computer Science, vol. 12167, pp. 3–20. Springer (2020). `https://doi.org/10.1007/978-3-030-51054-1_1`
2. Affeldt, R., Garrigue, J., Nowak, D., Saikawa, T.: A trustful monad for axiomatic reasoning with probability and nondeterminism. Journal of Functional Programming **31**, e17 (2021). `https://doi.org/10.1017/S0956796821000137`
3. Affeldt, R., Nowak, D.: Extending equational monadic reasoning with monad transformers. In: 26th International Conference on Types for Proofs and Programs (TYPES 2020). Leibniz International Proceedings in Informatics, vol. 188, pp. 2:1–2:21. Schloss Dagstuhl (Jun 2021). `https://doi.org/10.4230/LIPIcs.TYPES.2020.2`, `https://arxiv.org/abs/2011.03463`
4. Affeldt, R., Nowak, D., Saikawa, T.: A hierarchy of monadic effects for program verification using equational reasoning. In: 13th International Conference on Mathematics of Program Construction (MPC 2019), Porto, Portugal, October 7–9, 2019. Lecture Notes in Computer Science, vol. 11825, pp. 226–254. Springer (2019). `https://doi.org/10.1007/978-3-030-33636-3_9`
5. Agda: Agda's documentation v2.6.2.1 (2021), available at `https://agda.readthedocs.io/en/v2.6.2.1/`
6. Barthe, G., Grégoire, B., Riba, C.: Type-based termination with sized products. In: 22nd International Workshop on Computer Science Logic (CSL 2008), Bertinoro, Italy, September 16–19, 2008. Lecture Notes in Computer Science, vol. 5213, pp. 493–507. Springer (2008). `https://doi.org/10.1007/978-3-540-87531-4_35`
7. Benton, N., Hughes, J., Moggi, E.: Monads and effects. In: Applied Semantics, International Summer School (APPSEM 2000) Caminha, Portugal, September 9–15, 2000, Advanced Lectures. Lecture Notes in Computer Science, vol. 2395, pp. 42–122. Springer (2000). `https://doi.org/10.1007/3-540-45699-6_2`
8. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004). `https://doi.org/10.1007/978-3-662-07964-5`
9. Chan, J., Li, Y., Bowman, W.J.: Is sized typing for Coq practical? (2019), `https://arxiv.org/abs/1912.05601`

10. Christiansen, J., Dylus, S., Bunkenburg, N.: Verifying effectful Haskell programs in Coq. In: 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019), Berlin, Germany, August 18-23, 2019. pp. 125–138. ACM (2019). https://doi.org/10.1145/3331545.3342592
11. Cohen, C., Sakaguchi, K., Tassi, E.: Hierarchy builder: Algebraic hierarchies made easy in Coq with Elpi (system description). In: 5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020), June 29–July 6, 2020, Paris, France (Virtual Conference). LIPIcs, vol. 167, pp. 34:1–34:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.FSCD.2020.34
12. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009), Munich, Germany, August 17–20, 2009. Lecture Notes in Computer Science, vol. 5674, pp. 327–342. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_23
13. Garrigue, J.: Proving the correctness of OCaml typing by translation into Coq. The 17th Theorem Proving and Provers meeting (TPP 2021) (Nov 2021), presentation
14. Gibbons, J., Hinze, R.: Just do it: simple monadic equational reasoning. In: 16th ACM SIGPLAN international conference on Functional Programming (ICFP 2011), Tokyo, Japan, September 19–21, 2011. pp. 2–14. ACM (2011). https://doi.org/10.1145/2034773.2034777
15. Hierarchy Builder: Hierarchy builder wiki—missingjoin. Available at https://github.com/math-comp/hierarchy-builder/wiki/MissingJoin (2021)
16. Jaskelioff, M.: Modular monad transformers. In: Programming Languages and Systems, 18th European Symposium on Programming (ESOP 2009), York, UK, March 22–29, 2009. Lecture Notes in Computer Science, vol. 5502, pp. 64–79. Springer (2009). https://doi.org/10.1007/978-3-642-00590-9_6
17. Jomaa, N., Nowak, D., Grimaud, G., Hym, S.: Formal proof of dynamic memory isolation based on MMU. Sci. Comput. Program. **162**, 76–92 (2018). https://doi.org/10.1016/j.scico.2017.06.012
18. Letan, T., Régis-Gianas, Y.: FreeSpec: specifying, verifying, and executing impure computations in Coq. In: 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020), New Orleans, LA, USA, January 20–21, 2020. pp. 32–46. ACM (2020). https://doi.org/10.1145/3372885.3373812
19. Maillard, K., Ahman, D., Atkey, R., Martínez, G., Hritcu, C., Rivas, E., Tanter, É.: Dijkstra monads for all. Proc. ACM Program. Lang. **3**(ICFP), 104:1–104:29 (2019). https://doi.org/10.1145/3341708
20. MathComp: The mathematical components repository. Available at https://github.com/math-comp/math-comp (2022), version 1.14.0. See ssreflect/order.v for ordered types. See https://github.com/math-comp/math-comp/blob/251c8ec2490ff645a6afa45dd1ec238b9f71a554/mathcomp/ssreflect/tuple.v#L460-L499 for the bseq type.
21. Monae: Monadic effects and equational reasoning in Coq. Available at https://github.com/affeldt-aist/monae (2021), version 0.4.1
22. Mu, S.: Calculating a backtracking algorithm: An exercise in monadic program derivation. Tech. rep., Academia Sinica (2019), TR-IIS-19-003
23. Mu, S.: Equational reasoning for non-determinism monad: A case study of Spark aggregation. Tech. rep., Academia Sinica (2019), TR-IIS-19-002
24. Mu, S., Chiang, T.: Declarative pearl: Deriving monadic quicksort. In: 15th International Symposium on Functional and Logic Programming (FLOPS 2020), Akita,

Japan, September 14–16, 2020. Lecture Notes in Computer Science, vol. 12073, pp. 124–138. Springer (2020). `https://doi.org/10.1007/978-3-030-59025-3_8`

25. Oliveira, B.C.D.S., Schrijvers, T., Cook, W.R.: MRI: Modular reasoning about interference in incremental programming. Journal of Functional Programming **22**, 797–852 (2012). `https://doi.org/10.1017/S0956796812000354`
26. Pauwels, K., Schrijvers, T., Mu, S.: Handling local state with global state. In: 13th International Conference on Mathematics of Program Construction (MPC 2019), Porto, Portugal, October 7–9, 2019. Lecture Notes in Computer Science, vol. 11825, pp. 18–44. Springer (2019). `https://doi.org/10.1007/978-3-030-33636-3_2`
27. Sakaguchi, K.: Program extraction for mutable arrays. Sci. Comput. Program. **191**, 102372 (2020). `https://doi.org/10.1016/j.scico.2019.102372`
28. Schrijvers, T., Piróg, M., Wu, N., Jaskelioff, M.: Monad transformers and modular algebraic effects: what binds them together. In: 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019), Berlin, Germany, August 18–23, 2019. pp. 98–113. ACM (2019). `https://doi.org/10.1145/3331545.3342595`
29. Sozeau, M.: Equations—a function definitions plugin. Available at `https://mattam82.github.io/Coq-Equations/` (2009), last stable release: 1.3 (2021)
30. Sozeau, M., Mangin, C.: Equations reloaded: high-level dependently-typed functional programming and proving in coq. Proc. ACM Program. Lang. **3**(ICFP), 86:1–86:29 (2019). `https://doi.org/10.1145/3341690`
31. The Coq Development Team: The Coq Proof Assistant Reference Manual. Inria (2022), available at `https://coq.inria.fr`. Version 8.15.1