Practical Aspects of Monadic Equational Reasoning in Coq

Ayumu Saito[1] and Reynald Affeldt[2]

[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 [1] Functional programs with side effects represented by monads are amenable to equational reasoning. This approach to program verification is called monadic equational reasoning and has been experimented several times using proof assistants based on dependent type theory. In order to improve such formalizations, we extend Monae, an existing Coq library that supports monadic equational reasoning. First, to improve the scalability of Monae, we reimplement its hierarchy of effects using a generic tool to build hierarchies of mathematical structures and extend it with the array and the plus monad. Second, we discuss a recurring technical difficulty due to the shallow embedding of monads in the proof assistant. 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 a principled approach using dependent types to deal with this problem. Third, we augment Monae with an improved theory about nondeterministic permutations and, thanks to these contributions, we are finally able to completely formalize derivations of quicksort by Mu and Chiang.

1 Introduction

Functional programming languages are suitable for equational reasoning because of their referential transparency. However, most practical programs have effects. Still, programs written with monads to represent effects are amenable to equational reasoning: this is called "monadic equational reasoning" [GH11]. Monadic equational reasoning has been used to verify several programs (e.g., [GH11, OSC12, Mu19b, Mu19a, PSM19, MC20]) and some of these experiments come with formalizations in the dependently typed proof assistants Coq and Agda (e.g., [OSC12, PSM19, ANS19, MC20]).

The Coq library MONAE [Mon21] is an effort to provide a library for formal verification of monadic equational reasoning. It already proved useful by uncovering errors in pencil-and-paper proofs (e.g., [ANS19, Sect. 4.4]), leading to new fixes for known errors (e.g., [AN21]), and providing clarifications for the construction of monads used in probabilistic programs (e.g., [AGNS21, Sect. 6.3.1]).

In this paper, we report on several improvements of MONAE that are of general interest for the formal verification of monadic equational reasoning. Before explaining our contributions, let us illustrate concretely the main ingredients of monadic equational reasoning in a proof assistant.

Example of proof by monadic equational reasoning Let us assume that we are given a type `monad` for monads with the `Ret` notation for the unit and the $\gg=$ notation for the bind operator ($\gg$ being a version of the bind operator whose continuation ignores its argument). We can use this type to define a generic function that repeats a computation `mx`:

[1]This paper was presented at PPL 2022 (`https://jssst-ppl.org/workshop/2022/`) which is a Japanese domestic workshop with no formal proceedings.

```
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 `\o` is function composition):

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

Let us use monadic equational reasoning to prove "tick fusion" [OSC12, Sect. 4.1] (in a state monad; `addn` is the addition of natural numbers):

```
Lemma tick_fusion n : rep n tick = get >>= (put \o 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:

`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 proof language [The21]) shows that tick fusion can be proved by a sequence of rewritings involving mostly monadic laws (see Fig. 1 for the intermediate goals displayed by Coq or [OSC12, Sect. 4.1] for a pencil-and-paper proof):

```
Lemma tick_fusion n : rep n tick = get >>= (put \o 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), monadic equational reasoning involves several monads with inheritance relations (here the state monad satisfies more laws than a generic monad).

In this paper, our contribution is to improve an existing formalization of monadic equational reasoning. 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 a reusable hierarchy requires advanced formalization techniques. The largest hierarchy [ANS19] we are aware of uses the technique of *packed classes* [GGMR09]. This approach is manual and verbose, and therefore is error-prone and does not scale very well. In this work, we reimplement and extend this hierarchy using a more scalable 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 small inconvenience when proving lemmas in general and when proving termination in particular. Indeed, contrary to a standard functional programming language, say, Haskell, a type-based proof assistant requires termination proofs for every function

$$
\begin{array}{rcll}
\texttt{rep n tick} &=& \texttt{get} \ggg\!= (\texttt{put} \ \backslash\texttt{o addn n}) & \text{(initial goal)} \\
\texttt{rep } 0 \texttt{ tick} &=& \texttt{get} \ggg\!= (\texttt{put} \ \backslash\texttt{o addn } 0) & \text{(base case)} \\
\texttt{get} \ggg\!= \texttt{put} &=& \texttt{get} \ggg\!= (\texttt{put} \ \backslash\texttt{o addn } 0) & \text{(by } \texttt{getputskip}\text{)} \\
\texttt{rep n}.{+}1 \texttt{ tick} &=& \texttt{get} \ggg\!= (\texttt{put} \ \backslash\texttt{o addn n}.{+}1) & \text{(inductive case)} \\
(\texttt{get} \ggg\!= (\texttt{put} \ \backslash\texttt{o succn})) \gg (\texttt{get} \ggg\!= (\texttt{put} \ \backslash\texttt{o addn n})) &=& \texttt{get} \ggg\!= (\texttt{put} \ \backslash\texttt{o addn n}.{+}1) & \text{(by inductive hyp.)} \\
\texttt{get} \ggg\!= (\texttt{fun x} \Rightarrow (\texttt{put} \ \backslash\texttt{o succn}) \texttt{ x} \gg (\texttt{get} \ggg\!= (\texttt{put} \ \backslash\texttt{o addn n}))) &=& \texttt{get} \ggg\!= (\texttt{put} \ \backslash\texttt{o addn n}.{+}1) & \text{(by } \texttt{bindA}\text{)} \\
(\texttt{put} \ \backslash\texttt{o succn}) \texttt{ m} \gg (\texttt{get} \ggg\!= (\texttt{put} \ \backslash\texttt{o addn n})) &=& (\texttt{put} \ \backslash\texttt{o addn n}.{+}1) & \text{(by extensionality)} \\
((\texttt{put} \ \backslash\texttt{o succn}) \texttt{ m} \gg \texttt{get}) \ggg\!= (\texttt{put} \ \backslash\texttt{o addn n}) &=& (\texttt{put} \ \backslash\texttt{o addn n}.{+}1) \texttt{ m} & \text{(by } \texttt{bindA}\text{)} \\
(\texttt{put m}.{+}1 \gg \texttt{Ret m}.{+}1) \ggg\!= (\texttt{put} \ \backslash\texttt{o addn n}) &=& (\texttt{put} \ \backslash\texttt{o addn n}.{+}1) \texttt{ m} & \text{(by } \texttt{putget}\text{)} \\
\texttt{put m}.{+}1 \gg (\texttt{Ret m}.{+}1 \ggg\!= (\texttt{put} \ \backslash\texttt{o addn n})) &=& (\texttt{put} \ \backslash\texttt{o addn n}.{+}1) \texttt{ m} & \text{(by } \texttt{bindA}\text{)} \\
\texttt{put m}.{+}1 \gg (\texttt{put} \ \backslash\texttt{o addn n}) \texttt{ m}.{+}1 &=& (\texttt{put} \ \backslash\texttt{o addn n}.{+}1) \texttt{ m} & \text{(by } \texttt{bindretf}\text{)} \\
\texttt{put } (\texttt{n} + \texttt{m}.{+}1) &=& (\texttt{put} \ \backslash\texttt{o addn n}.{+}1) \texttt{ m} & \text{(by } \texttt{putput}\text{)} \\
\texttt{put } (\texttt{n} + \texttt{m}.{+}1) &=& \texttt{put } (\texttt{n} + \texttt{m}.{+}1) & \text{(by } \texttt{addSnnS}\text{)}
\end{array}
$$

Figure 1. Intermediate goals displayed by Coq when executing the proof script for tick fusion (see Sect. 1)

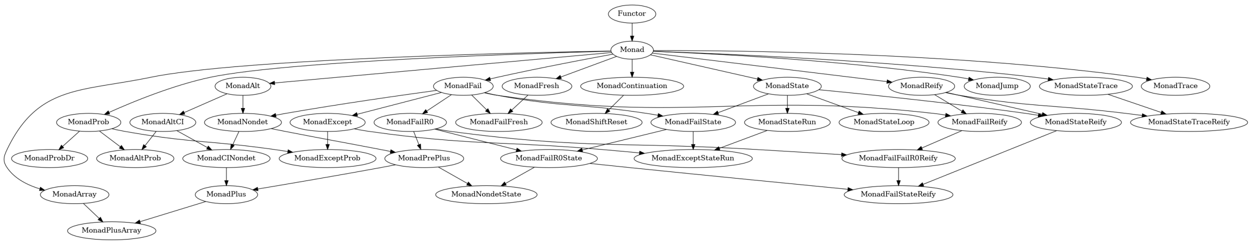

Figure 2. The hierarchy of monads provided by MONAE [Mon21] at the time of this writing

involved. However, it happens that the tooling provided by proof assistants to deal with non-structurally recursive functions is at best bothersome for monadic equational reasoning. We explain how to deal with such proofs in a principled way (Sections 3 and 4).

- Last, we demonstrate the usefulness of the two previous contributions by completing an existing formalization of quicksort (Sect. 5) and as a by-product enriching our library of monadic equational reasoning, in particular, with theories of nondeterministic permutations.

2 An extensible implementation of monad interfaces

In this section, we explain how we formalize a hierarchy of interfaces for monads used in monadic equational reasoning. This hierarchy is a conservative extension of previous work [ANS19, AN21, AGNS21] that we have reimplemented using a generic tool called HIERARCHY-BUILDER [CST20] for the formalization of hierarchies of mathematical structures. This provides us with a hierarchy that is easier to extend with new monads as we will demonstrate with the plus monad and the array monad and that does not suffer type inference problems (see Sect. 6).

2.1 Formalizing a hierarchy of monads using HIERARCHY-BUILDER

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. In this setting, a functor is defined as a function `M` of type `Type → Type` that represents the action on objects and a function `actm` that represents the action on morphisms. Using HIERARCHY-BUILDER, this definition takes the form of a record `isFunctor` called a *mixin* (line 1):

```
1 HB.mixin Record isFunctor (M : Type → Type) := {
2   actm : ∀ A B, (A → B) → M A → M 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   HB.structure Definition Functor := {M of isFunctor M}.
6   Notation functor := Functor.type.
```

The actions on objects and morphisms appear at lines 1 and 2 respectively. The function `actm` satisfies the functor laws (lines 3 and 4). The type of functors is obtained by declaring the mixin as a *structure* (line 5). Finally, line 6 defines a notation for convenience. Given a functor `M` and a morphism `f`, we note `M # f` the action of `M` on `f`.

Then, given two functors `M` and `N`, we formalize natural transformations as a family of functions `f` of type `∀ A, M A → N A` (notation: `M ~~> N`) that satisfies the following predicate:

```
Definition naturality (M N : functor) (f : M ~~> N) :=
  ∀ A B (h : A → B), (N # h) \o f A = f B \o (M # h).
```

We formalize the type of natural transformations as a *packed class* [GGMR09]. Packed classes are actually what Hierarchy-Builder implements. However, the current implementation of Hierarchy-Builder does not handle completely hierarchies of morphisms yet. That is why we resort to a manual encoding. The packed class for natural transformations consists of a mixin (line 4) and a structure (line 5). The verbose re-definition of the structure at line 6 is required in the absence of Hierarchy-Builder for type inference to work as intended.

```
1   Module Natural.
2   Section natural.
3   Variables M N : functor.
4   Record mixin_of (f : M ~~> N) := Mixin { _ : naturality M N f }.
5   Structure type := Pack { cpnt : M ~~> N ; mixin : mixin_of cpnt }.
6   Definition type_of (phM : @phantom (Type → Type) M) (phN : @phantom (Type → Type) N) := type.
7   Module Exports.
8   Notation nattrans := type.
9   Coercion cpnt : type >-> Funclass.
10  Notation "M ⟹ N" := (@type_of _ _ (@Phantom (Type → Type) M) (@Phantom (Type → Type) N)).
11  Identity Coercion type_of_type : type_of >-> type.
12  End Exports.
13  End Natural.
```

(The modifier `@` in Coq disables implicit arguments.) As a result of this declaration, we obtain in particular the notation `M ⟹ N` of natural transformations from the functor `M` to the functor `N` (see line 10).

Finally, a monad is defined by two natural transformations: the unit `ret` (line 2 below) and the multiplication `join` (line 3) that satisfy three monad laws (lines 7–9). The interface provided by the mixin further provides an identifier for the bind operator (line 4). It also features two equations that respectively link (i) the action on morphisms with bind and unit (line 5) and (ii) bind with its definition in term of unit and multiplication (line 6).

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

The `fmapE` and `bindE` equations are not surprising because they correspond to standard monadic laws. They are necessary to make the action on morphisms of the functor agree with the multiplication/bind of the monad. For their addition, we have been guided by Hierarchy-Builder, which has recently been extended to detect such needs in general (this is an instance of "forgetful inheritance" [ACK+20]). Hereafter, we use `Ret` as a notation for `(@ret _ _)`.

Note that the above definition of monads is not the interface one uses to define a new monad. Using Hierarchy-Builder, one rather uses *factories* to instantiate structures. Factories present themselves as a smaller mixin from which the actual definition is recovered:

```
HB.factory Record Monad_of_ret_bind (M : Type → Type) of isFunctor M := {
  ret : idfun ⟹ M ;
  bind : ∀ A B, M A → (A → M B) → M B ;
  fmapE : ∀ A B (f : A → B) (m : M A), ([the functor of M] # f) m = bind _ _ m (@ret _ \o f) ;
  bindretf : BindLaws.left_neutral bind ret ;
  bindmret : BindLaws.right_neutral bind ret ;
  bindA : BindLaws.associative bind }.
```

This is closer to the textbook definition of a monad and does not require the simultaneous definition of the unit, the multiplication *and* bind.

2.2 Extension with the array monad and the plus monad

The array monad The array monad extends a basic monad with a notion of indexed array (see, e.g., [MC20, 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.

Thanks to Hierarchy-Builder, 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 (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 (k : S → M A), aput i s ≫ aget i ≫= k = aput i s ≫ k s ;
  (* other laws omitted to save space, see [Mon21, file hierarchy.v] for details *) }.
HB.structure Definition MonadArray S (I : eqType) := { M of isMonadArray S I M & }.
Notation arrayMonad := MonadArray.type.
```

The plus monad We define the plus monad following [PSM19] and [MC20, 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 [AGNS21] 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 [AN21]) 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 [ANS19]. In other words, only the right-distributivity law is missing.

We therefore implement the `plusMonad` by extending above monads with the right-distributivity law as follows. First, we defined the intermediate `prePlusMonad` by adding right-distributivity to the combination of `nondetMonad` and `failR0Monad` (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 _) }.
HB.structure Definition MonadPrePlus := {M of isMonadPrePlus M & }.
Notation prePlusMonad := MonadPrePlus.type.
```

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

```
HB.structure Definition MonadPlus := {M of MonadCINondet M & MonadPrePlus M}.
Notation plusMonad := MonadPlus.type.
```

The plus-array monad Finally, we can combine the array and the plus monad to obtain the `plusArrayMonad` [MC20, Sect. 5]:

```
HB.structure Definition MonadPlusArray S (I : eqType) := {M of MonadPlus M & isMonadArray S I M}.
Notation plusArrayMonad := MonadPlusArray.type.
```

The resulting hierarchy of monad interfaces is depicted in Fig. 2.

3 Difficulties with the termination of monadic functions

Functions defined in a proof assistant based on dependent types need to terminate to preserve logical consistency. In practice, providing termination proofs is bothersome, in particular when it is not at the heart of the target formalization. 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 [Agd21].

The goal of this section is to illustrate concretely difficulties when proving the termination of functions in the context of monadic equational reasoning in Coq. We first recall standard tooling to prove termination (Sect. 3.1) and provide concrete examples of difficulties (Sect. 3.2).

3.1 Background: standard Coq tooling to prove termination

3.1.1 The Function command

In Coq, the Function command [The21, Chapter Functional induction] 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 below can be any ordered type [Mat21]):

```
Function qsort (s : seq T) {measure size s} : seq T :=
  match s with
  | [::] ⇒ [::]
  | h :: t ⇒ let: (ys, zs) := partition h t in qsort ys ++ h :: qsort zs
  end.
```

The function call `partition h t` returns a pair of lists that partitions `t` with elements smaller (resp. greater) than `h`. The annotation `{measure size s}` indicates that the size of the input list is expected to decrease. Once the Function 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 well-known accessibility predicate [BC04, Chapter 15]. To the best of our knowledge, Agda users enjoy almost no automation when dealing with general recursion.

At first sight, the approach using Function is appealing: the syntax is minimal and, as a by-product, it automatically generates additional useful lemmas, e.g., in the case of `qsort`, lemmas capturing the fixpoint equation of `qsort` and expressing structural induction/recursion principles over objects of type `qsort` [The21, Chapter Functional induction].

3.1.2 The Program/Fix approach

The Program/Fix approach is more primitive and verbose than the Function approach, but it is also more flexible and robust to changes w.r.t. hidden automation. It is a combination of the Program command for dependent type programming [The21, 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 Function 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:

```
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'.
```

3.2 Limitations of Coq standard tooling to prove termination

In the context of monadic equational reasoning, the standard tooling provided by Coq to prove termination is not always sufficient. We illustrate this with the example of a function that computes permutations nondeterministically: the `perm` function from [MC20], below written in Agda.

```
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) }
```

The function `split` splits a list nondeterministically. The notation || corresponds to nondeterministic choice (the notation `[~]` in Sect. 2.2). The function `perm` uses `split` and `liftM2`, a generic monadic function that lifts a function `h : A -> B -> C` to apply to a monadic function of type `M A -> M B -> M C`.

3.2.1 Difficulty with the Function approach

First, we observe that since it is structurally recursive, `split` can be encoded directly in Coq as a `Fixpoint` (using the `altMonad`, see Sect. 2.2) [2]:

```
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)).
```

However, the `Function` command fails to define directly the `perm` function[3]:

```
Fail Function qperm {M : altMonad} A (s : seq A) {measure size s} : M (seq A) :=
  if s isn't x :: xs then Ret [::] else splits xs ≫=
    (fun '(ys, zs) ⇒ liftM2 (fun a b ⇒ a ++ x :: b) (qperm ys) (qperm zs)).
```

It seems that the `Function` command cannot deal automatically with the recursive call that appears after bind (≫=), which is related to our use of a shallow embedding for monads.

3.2.2 Difficulty with the Program/Fix approach

Applying the `Program/Fix` approach to define `qperm` does not fail immediately but leads to a dead end. To explain this, let us define an intermediate function `qperm'` as we explained in Sect. 3.1.2:

```
Program Definition qperm' {M : altMonad} A (s : seq A)
    (f : ∀ s', size s' < size s → M (seq A)) : M (seq A) :=
  if s isn't x :: xs then Ret [::] else
    splits xs ≫=
    (fun a ⇒ liftM2 (fun a b ⇒ a ++ x :: b) (f a.1 _) (f a.2 _)).
```

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

```
xxs : ∀ s' : seq A, size s' < size (x :: xs) → M (seq A)
x : A
xs : seq A
a, b : seq A
==============================================
size a < (size xs).+1
```

There is no way to prove this goal since there is no information about the list `a`. We propose solutions to deal with this problem in the next section.

4 Add dependent types to return types for formal proofs

In this section, we explain that we can always enrich the return type of monadic functions with dependent types to complete formal proofs, in particular to prove termination in the context of monadic equational reasoning.

4.1 Add dependent types to called functions to prove termination

We explain how to prove the termination of the `qperm` function that we introduced in Sect. 3.2.2. The idea is to augment the return type of the called function `splits` with dependent types so that the `Program/Fix` approach succeeds.

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

[3] We call this function `qperm` to avoid conflicts with other definitions of permutations.

The splits function is defined such that its return type is `M (seq A * seq A)` (Sect. 3.2.1). 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` (the type of "bounded-size lists" comes from the MathComp library [Mat21]).

```
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 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. Last, the notation `[bseq of x :: ys]` triggers MathComp automation (using canonical structures [Mat21]) to build a `n.+1.-bseq A` list using the fact that `ys` is itself a `n.-bseq A` list.

Next, we re-define `qperm'` (following Sect. 3.2.2) using `splits_bseq` (instead of `splits`):

```
Program Definition qperm' {M : altMonad} A (s : seq A)
    (f : ∀ s', size s' < size s -> M (seq A)) : M (seq A) :=
  if s isn't x :: xs then Ret [::] else
    splits_bseq xs >>=
    (fun '(ys, zs) => liftM2 (fun a b => a ++ x :: b) (f ys _) (f zs _)).
```

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

Finally, the function `qperm` can be defined using `Fix` as explained in Sect. 3.1.2:

```
Definition qperm {M : altMonad} A : seq A -> M (seq A) :=
  Fix well_founded_size (fun _ => M _) qperm'.
```

The nonderministic computation of permutations using nondeterministic selection is another example of this approach [GH11, Sect. 4.4] (see [Mon21, file `fail_lib.v`]).

4.2 Add dependent types with a dependently-typed assertion

The approach explained in the previous section is satisfactory when the needed type already available in some standard library. It is less practical otherwise. Yet, we can reach a similar result 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 : failMoand} 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:

```
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 [MC20]). 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 induction of the size of the input list), we run into the following subgoal (we abbreviate the continuations following `splits` in the code of `qperm` as the functions `k1` and `k2` to keep the displayed code short):

```
  s : seq A
  ns : size s < n
====================
  do x ← splits s; (fun x : seq A * seq A ⇒ k1 x.1 x.2) =
  do x ← splits s; (fun x : seq A * seq A ⇒ k2 x.1 x.2)
```

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
  a, b : seq A
====================
  k1 a b = k2 a b
```

As in Sect. 3.2.2, 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 with the information that the concatenation of the returned lists is of the same size as the input (as defined by `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` that projects the witness of the dependent type:

```
Lemma dsplitsE {M : prePlusMonad} A (s : seq A) :
  splits s = fmap (fun x ⇒ (dsplitsT1 x, dsplitsT2 x)) (dsplits s) :> M _.
```

We can locally introduce `dsplits` using `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
  a, b : seq A
  ab : size a + size b == size s
====================
  k1 a b = k2 a b
```

See [Mon21, file example_iquicksort.v] for the complete script.

Although we use here a dependently-typed assertion to prove a lemma, we will see in the next section 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 techniques we explained so far to provide a complete formalization of quicksort derivations [MC20]. 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. The function qperm is indeed a good fit to specify quicksort, but it is not the most obvious definition [MC20, Sect. 3] and its shape 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 (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) : @perm 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 [MC20] (see Table 1).

5.2 Program refinement

The rest of this paper uses notion of program refinement introduced by Mu and Chiang [MC20, 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).
```

Definition/lemma in [MC20]		Coq equivalent in [Mon21]
file `Implementation.agda`		
`ext`	postulate	standard axiom of functional extensionality
file `Monad.agda`		
`write-write-swap`	postulate	not used
`writeList-++`	postulate	see `writeList_cat` (file `array_lib.v`)
`writeList-writeList-comm`	postulate	see `writeListC` (file `array_lib.v`)
file `Nondet.agda`		
`return⊑perm`	postulate	see `refin_qperm_ret` (file `fail_lib.v`)
`perm-idempotent`	postulate	see Sect. 5.1
`perm-snoc`	postulate	see `qperm_rcons` (file `fail_lib.v`)
`sorted-cat3`	postulate	see Sect. 5.3
`perm-preserves-length`	postulate	see Sect. 4.2
`perm-preserves-all`	postulate	see Sect. 5.3
`perm`	`TERMINATING`	see Sect. 4.1
`mpo-perm`	`TERMINATING`	commutation of computations (see Sect. 5.3)
`partl/partl-spec`	`TERMINATING`	see `partl` (file `example_iqsort.v`), solved by currying
`partl'/partl'-spec`	`TERMINATING`	see `qperm_partl` (file `example_iqsort.v`)
`mpo-partl'`	`TERMINATING`	commutation of computations (see Sect. 5.3)
`qsort/qsort-spec`	`TERMINATING`	see Sect. 3.1.1
file `IPartl.agda`		
`ipartl/ipartl-spec`	`TERMINATING`	see Sect. 5.4, solved by currying
`introduce-swap` [MC20, eqn 11]	postulate	see `refin_writeList_rcons_aswap` (file `array_lib.v`)
`introduce-read`	postulate	not used
file `IQSort.agda`		
`iqsort/iqsort-spec`	`TERMINATING`	see Sect. 4.2
`introduce-read`	postulate	see `writeListRet` (file `array_lib.v`)
`introduce-swap` [MC20, eqn 13]	postulate	see `refin_writeList_cons_aswap` (file `array_lib.v`)

Table 1. Admitted facts in [MC20] and their formalization in [Mon21] (Lemmas `xyz-spec` require the `TERMINATING` pragma as a consequence of the function `xyz` being postulated as terminating.)

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

Using the techniques described above, we formalized quicksort as a pure function as Mu and Chiang did [MC20]. We explain how we proved in Coq the few axioms in their Agda formalization.

What we actually prove is the correctness as a sort algorithm of the `qsort` function of Sect. 3.1.1. We proved it by showing that it refines an algorithm that is obviously correct. This algorithm is `slowsort`: a function that filters only the sorted permutations of all permutations derived by `qperm`, which is obviously correct as a sorting algorithm, but of course cannot be used in practice.

```
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 \o qsort ⊆̇ slowsort.
```

Among the axioms left by Mu and Chiang that we prove, we can distinguish axioms about termination and axioms 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[4] 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 permutations as a result of perm preserves 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 importantly 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.
```

Its proof uses induction on syntax as explained in [ANS19, Sect. 5.1].

We claim that our formalization is shorter than Mu and Chiang's. It is difficult 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, 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 [Mon21]), 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 [MC20].

The first difficulty is to prove the termination of in-place quicksort function. Let us first explain the Agda implementation (which has termination postulated). The partition step is performed by the function ipartl:

```
{-# 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 function ipartl takes a pivot p, an index i to the array (from which the contents correspond to some list, say, ys ++ zs ++ xs), and the three sizes of the lists yz, zs, and xs. It returns the sizes of the two partitions. The code makes use of the array monad (Sect. 2.2). The swap function uses the read/write operators of the array monad to swap two cells of the array.

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:

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

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

We encode this definition in Coq and prove its termination using the technique we 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. However, the direct definition of `iqsort` in Coq using the `Program`/`Fix` approach (Sect. 3.1.2) fails for the same reasons as explained in Sect. 3.2.2: 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 the return type of `ipartl` to `M {n : nat * nat | (n.1 ≤ x + y + z) ∧ (n.2 ≤ x + y + z)}` (where `x`, `y`, `z` are the sizes of the lists input to `ipartl`) so as to ensure that the sizes returned by the partition function are smaller than the size of the array being processed[5]:

```
Definition dipartlT y z x := {n : nat * nat | (n.1 ≤ x + y + z) ∧ (n.2 ≤ x + y + z)}.
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` using the `Program`/`Fix` approach (the notation `%:Z` is for injecting natural numbers into integers):

```
Program Fixpoint 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 [Mon21, 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 at the time of this writing. Anyway, the following refinement proof requires the plus-array monad.

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.
```

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. 2.2:

```
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 carefully explained by Mu and Chiang in their paper [MC20]. In fact, we did not need to look at the Agda code except for the very last part which is lacking details [MC20, Sect. 5.3]. Our understanding is that the key aspect of the derivation (and

[5]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.

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. 3.1.1). 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 [ANS19, AN21]. 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 Sect. 2.2. It is also more "robust"; indeed, we discovered that previous work [AN21, Fig. 1] lacked an intermediate interface, which was making troubling type inference (see [Hie21] for details). Note that type classes provide an alternative approach to the implementation of a hierarchy of monad interfaces (see, e.g., [MC20]). Hierarchy-Builder has advantages: it helps designing a hierarchy for example by detecting forgetful inheritance (Sect. 2.1).

The examples used in this paper stem from the derivations of quicksort by Mu and Chiang [MC20]. Together with their paper, the authors provide as accompanying material a 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 formalization of monadic equational reasoning. In this paper, we explained in particular how to complete their formalization, which we actually rework from scratch, favoring equational reasoning; in other words, our formalization is not a port.

For the purpose of this work, 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 [GH11, Sect. 4.4], using nondeterministic selection and the function `unfoldM` [Mu19a, Sect. 3.2], using nondeterministic insertion [Mu19b, Sect. 3], or using `liftM2` [MC20, Sect. 3]. The current version of Monae has a formalization of each.

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 [JNGH18], Maillard et al. have been developing a framework to verify programs with monadic effects using Dijkstra monads [MAA+19], Christiansen et al. have been verifying effectful Haskell programs in Coq [CDB19], and Letan et al. have been exploring verification in Coq of impure computations using a variant of the free monad [LR20].

7 Conclusion

We reported on various techniques that can be applied to formal monadic equational reasoning, and also on the complete formalization of functional quicksort and in-place quicksort as their application. For that purpose, we used an existing Coq library called Monae. To ease the addition of new monad interfaces, we reimplemented the hierarchy of interfaces of Monae using Hierarchy-Builder and illustrated this extension with the plus-array monad. We also observed a recurring technical issue due to the shallow embedding of monadic functions, whose termination is not easy to guarantee using Coq's standard commands. We proposed solutions using dependent types. We applied these techniques and the extension of Monae to the formalization 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., [SPWJ19, PSM19]). 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 [Gar21].

Acknowledgements The authors would like to thank Cyril Cohen and Enrico Tassi for their assistance with Hierarchy-Builder, the members of the Programming Research Group of the Department of Mathematical and Computing Science at the Tokyo Institute of Technology for fruitful discussions, and the anonymous reviewers of PPL2022 for their comments. The second author acknowledges the support of the JSPS KAKENHI Grant Number 18H03204.

References

[ACK+20] Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. 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*, volume 12167 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2020.

[Agd21] Agda. *Agda's documentation v2.6.2.1*, 2021. Available at `https://agda.readthedocs.io/en/v2.6.2.1/`.

[AGNS21] Reynald Affeldt, Jacques Garrigue, David Nowak, and Takafumi Saikawa. A trustful monad for axiomatic reasoning with probability and nondeterminism. *Journal of Functional Programming*, 31:e17, 2021.

[AN21] Reynald Affeldt and David Nowak. Extending equational monadic reasoning with monad transformers. In *26th International Conference on Types for Proofs and Programs (TYPES 2020)*, volume 188 of *Leibniz International Proceedings in Informatics*, pages 2:1–2:21. Schloss Dagstuhl, Jun 2021.

[ANS19] Reynald Affeldt, David Nowak, and Takafumi Saikawa. 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*, volume 11825 of *Lecture Notes in Computer Science*, pages 226–254. Springer, 2019.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[CDB19] Jan Christiansen, Sandra Dylus, and Niels Bunkenburg. Verifying effectful Haskell programs in Coq. In *12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019), Berlin, Germany, August 18-23, 2019*, pages 125–138. ACM, 2019.

[CST20] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. 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)*, volume 167 of *LIPIcs*, pages 34:1–34:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[Gar21] Jacques Garrigue. Proving the correctness of OCaml typing by translation into Coq. The 17th Theorem Proving and Provers meeting (TPP 2021), Nov 2021. Presentation.

[GGMR09] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009), Munich, Germany, August 17–20, 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2009.

[GH11] Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. In *16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19–21, 2011*, pages 2–14. ACM, 2011.

[Hie21] Hierarchy Builder. Hierarchy builder wiki—missingjoin. Available at `https://github.com/math-comp/hierarchy-builder/wiki/MissingJoin`, 2021.

[JNGH18] Narjes Jomaa, David Nowak, Gilles Grimaud, and Samuel Hym. Formal proof of dynamic memory isolation based on MMU. *Sci. Comput. Program.*, 162:76–92, 2018.

[LR20] Thomas Letan and Yann Régis-Gianas. 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*, pages 32–46. ACM, 2020.

[MAA+19] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *Proc. ACM Program. Lang.*, 3(ICFP):104:1–104:29, 2019.

[Mat21] MathComp. The mathematical components repository. Available at `https://github.com/math-comp/math-comp`, 2021. Version 1.13.0. See `https://github.com/math-comp/math-comp/blob/master/mathcomp/ssreflect/tuple.v#L460-L499` for the bseq type.

[MC20] Shin-Cheng Mu and Tsung-Ju Chiang. Declarative pearl: Deriving monadic quicksort. In *15th International Symposium on Functional and Logic Programming (FLOPS 2020), Akita, Japan, September 14–16, 2020*, volume 12073 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2020.

[Mon21] Monae. Monadic effects and equational reasoning in Coq. Available at `https://github.com/affeldt-aist/monae`, 2021. Version 0.4.0.

[Mu19a] Shin-Cheng Mu. Calculating a backtracking algorithm: An exercise in monadic program derivation. Technical report, Academia Sinica, 2019. TR-IIS-19-003.

[Mu19b] Shin-Cheng Mu. Equational reasoning for non-determinism monad: A case study of spark aggregation. Technical report, Academia Sinica, 2019. TR-IIS-19-002.

[OSC12] Bruno C. D. S. Oliveira, Tom Schrijvers, and William R. Cook. MRI: Modular reasoning about interference in incremental programming. *Journal of Functional Programming*, 22:797–852, 2012.

[PSM19] Koen Pauwels, Tom Schrijvers, and Shin-Cheng Mu. Handling local state with global state. In *13th International Conference on Mathematics of Program Construction (MPC 2019), Porto, Portugal, October 7–9, 2019*, volume 11825 of *Lecture Notes in Computer Science*, pages 18–44. Springer, 2019.

[SPWJ19] Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. 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*, pages 98–113. ACM, 2019.

[The21] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Inria, 2021. Available at `https://coq.inria.fr`. Version 8.14.1.