

A practical formalization of monadic equational reasoning in dependent-type theory

REYNALD AFFELDT 

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

JACQUES GARRIGUE 

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

TAKAFUMI SAIKAWA 

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

Abstract

One can perform equational reasoning about computational effects with a purely functional programming language thanks to monads. Even though equational reasoning for effectful programs is desirable, it is not yet mainstream. This is partly because it is difficult to maintain pencil-and-paper proofs of large examples. We propose a formalization of a hierarchy of effects using monads in the Coq proof assistant that makes monadic equational reasoning practical. Our main idea is to formalize the hierarchy of effects and algebraic laws as interfaces like it is done when formalizing hierarchy of algebras in dependent-type theory. Thanks to this approach, we clearly separate equational laws from models. We can then take advantage of the sophisticated rewriting capabilities of Coq and build libraries of lemmas to achieve concise proofs of programs. We can also use the resulting framework to leverage on Coq's mathematical theories and formalize models of monads. In this article, we explain how we formalize a rich hierarchy of effects (nondeterminism, state, probability, etc.), how we mechanize examples of monadic equational reasoning from the literature, and how we apply our framework to the design of equational laws for a subset of ML with references.

1 Introduction

Pure functional programs are suitable for equational reasoning because they are referentially transparent. Monadic equational reasoning (Gibbons & Hinze, 2011) is an approach to reason equationally about programs with side effects using the algebraic properties of monads. In this approach, effects (such as state, nondeterminism and probability) are defined by interfaces with a set of equations, and these interfaces can be combined and extended to represent the combination of several effects. A number of programs using combined effects have been verified in this way (Gibbons & Hinze, 2011; d. S. Oliveira *et al.*, 2012; Chen *et al.*, 2017; Shan, 2018; Mu, 2019; Pauwels *et al.*, 2019), and this



approach is also applicable to program derivation (Mu, 2019a; Mu & Chiang, 2020). Among these experiments, some have been formally verified with proof assistants based on dependent-type theory such as COQ¹ and Agda.

The implementation of monadic equational reasoning in proof assistants based on dependent-type theory raises a number of issues. The main issue is the construction of a usable hierarchy of monad interfaces. For that purpose, one can use type classes and canonical structures (The Coq Development Team, 2024, Chapter Canonical Structures). Still, deep hierarchies are known to suffer scalability problems due to the complexity of type inference (Garillot *et al.*, 2009, Sect. 2.3). For a hierarchy of monad interfaces to be usable, it has to come with a rich basis of commonly known interfaces on top of which one can easily build new extensions. We also need to be able to construct a model for each extension in order to validate the new interface. Another issue is the use of shallow embedding. It is the privileged way to represent monadic functions in the context of monadic equational reasoning, but it raises problems on its own. For example, when needed, induction w.r.t. syntax requires the use of reflection. The use of shallow embedding also makes it important to know the techniques to deal with nonstructural recursion, as COQ functions must be terminating.² These are arguably general concerns in proof assistants but in the context of monadic equational reasoning they are all the more so important that they can prevent stating a proof goal or key reasoning steps.

In this article, we explain the implementation and the practice of a COQ library called MONAE that provides support for formal verification based on monadic equational reasoning. We leverage on existing tools for the COQ proof assistant: HIERARCHY-BUILDER (Cohen *et al.*, 2020), a tool intended at the formalization of mathematical structures that we use to formalize an extensible hierarchy of interfaces, and SSREFLECT (Gonthier & Mahboubi, 2010) (The Coq Development Team, 2024, Chapter The SSReflect proof language), an extension of COQ that provides more versatile rewriting tactics. MONAE has already proved useful by uncovering errors in pencil-and-paper proofs (Affeldt *et al.*, 2019, Sect. 4.4), leading to new fixes for known errors (Affeldt & Nowak, 2020) and providing clarifications for the construction of models of monads used in probabilistic programs (Affeldt *et al.*, 2021, Sect. 6.3.1). Our goal here is to summarize the technical aspects of MONAE: how we formalize interfaces of effects and models of monads, how we set up the formalization of an existing program derivation, and how we investigate the design of a new interface for an ML-like language with references.

Illustrating example: fast product We explain monadic equational reasoning in MONAE using an example by Gibbons & Hinze (2011) that shows the equivalence between a functional implementation of the product of integers (namely, `product`) and a monadic version (`fastprod`). On the left of Figure 1, we (faithfully) reproduce the series of rewritings that constitute the original proof (Gibbons & Hinze, 2011, Sect. 5.1). On the right, we display the equivalent series of COQ goals and tactics in MONAE.

¹ In the end of 2024, COQ will have been renamed ROCQ.

² This difficulty can be avoided in a proof assistant based on different foundations (like Isabelle/HOL) with a notion of partial function that does not admit evaluation.

Pencil-and-paper proof (Gibbons & Hinze, 2011, §5.1)	COQ intermediate goals and tactics using MONAE
<pre> fastprod xs = [[definition of fastprod]] catch (work xs) (ret 0) = [[specification of work]] catch (if 0 in xs then fail else ret (product xs)) (ret 0) = [[lift out the conditional]] if 0 in xs then catch fail (ret 0) else catch (ret (product xs)) (ret 0) = [[laws of catch, fail, and ret]] if 0 in xs then ret 0 else ret (product xs) = [[arithmetic: 0 in xs => product xs = 0]] if 0 in xs then ret (product xs) else ret (product xs) = [[redundant conditional]] ret (product xs) </pre>	<pre> fastprod s = [[rewrite /fastprod]] catch (work s) (Ret 0) = [[rewrite /work]] catch (if 0 \in s then fail else Ret (product s)) (Ret 0) = [[rewrite lift_if if_ext]] if 0 \in s then catch fail (Ret 0) else catch (Ret (product s)) (Ret 0) = [[rewrite catchfailm catchret]] if 0 \in s then Ret 0 else Ret (product s) = [[case: ifPn => // /product0]] (product0^{def} = ∀s. 0 ∈ s → product s = 0) Ret 0 = [[move <-]] Ret (product s) </pre>

Fig. 1. Comparison between a paper proof and a proof using MONAE.

In COQ, the product of natural numbers is simply defined as `foldr muln 1`. The “faster” product `fastprod` can be implemented using the failure monad and the exception monad as follows³:

```

1 Definition work (M : failMonad) (s : seq nat) : M nat :=
2   if 0 \in s then fail else Ret (product s).
3 Definition fastprod (M : exceptMonad) s : M nat :=
4   catch (work s) (Ret 0).
                    
```

The notation `Ret` (line 2) corresponds to the unit of the monad. The function `work` uses the `fail` operator of the failure monad `failMonad` and `fastprod` uses the `catch` operator of the exception monad `exceptMonad` (which extends the failure monad). We observe that the user can write a monadic program using one monad (`failMonad` or `exceptMonad`) and still use a notation to refer to the unit operator (`Ret`) of the base monad, illustrating subtyping.

Figure 1 displays the pencil-and-paper proof and the COQ proof that `fastprod` is pure, that is, that it never throws an unhandled exception. Both proofs are essentially the same, though in practice the COQ proof will be streamlined in a script of two lines (of less than 80 characters):

```

Lemma fastprodE s : fastprod s = Ret (product s).
Proof.
                    
```

³ The formalization of the relevant monads and their algebraic laws is explained in Sections 2.3, 2.4, and 2.5, but the details are not important to understand the illustrating example.

```
rewrite /fastprod /work lift_if if_ext catchfailm.
by rewrite catchret; case: ifPn => // /product0 <-.
Qed.
```

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

Outline In Section 2, we explain how we formalize an extensible hierarchy of interfaces of effects in COQ. This hierarchy allows for effects to be combined and is the main ingredient of monadic equational reasoning. In Section 3, we explain how we provide concrete models for monads, thus showing that the equations defining an effect are consistent. This includes in particular a formalization of monad transformers in Section 3.1. To illustrate the use of this framework, we explain in Section 4 the formalization of an existing program derivation by Mu & Chiang (2020); we focus on the setting of this experiment instead of the step-by-step program derivation. Section 5 demonstrates that we can use MONAE to investigate the design of new equational theories; concretely, we propose an equational theory for the *typed store monad* allowing for reasoning on ML-like languages such as OCaml. We review related work in Section 6 and conclude in Section 7.

2 An extensible implementation of interfaces of effects

2.1 HIERARCHY-BUILDER in a nutshell

HIERARCHY-BUILDER extends COQ with commands to define hierarchies of mathematical structures and is used in the Mathematical Components (hereafter, MATHCOMP) library (Mahboubi & Tassi, 2022). The commands are designed so that hierarchies can evolve (e.g., by splitting a structure into smaller structures) without breaking existing code. In addition to commands to build hierarchies, HIERARCHY-BUILDER also checks their validity by detecting missing interfaces (Saito & Affeldt, 2022, Sect. 6) or competing inheritance paths (Affeldt et al., 2020).

Each mathematical structure is presented as three layers: a carrier, operations, and properties. This layering is standard in mathematics,⁴ thus enabling a straightforward translation of textbook definitions of structures.

The main concept of HIERARCHY-BUILDER is the one of *factory*. This is a COQ record defined by the command `HB.factory` that packs operations and properties for a carrier, providing an interface to the theory of the corresponding mathematical structure.

Mixins (defined by the command `HB.mixin`) are the factories that are used to provide the primitive definition of mathematical structures; factories other than mixins are rather used to provide alternative definitions to the primitive ones.

Structures (defined by the command `HB.structure`) pack a carrier with one or more factories (intuitively, forming a sigma-type) to form an actual mathematical structure. More precisely, the command

⁴ This layering is also referred to as the “stuff, structure, and property” principle in a modern context (Baez & Shulman, 2010, Sect. 2.4).

```
HB.structure Definition M := {A of f1 & f2 & ... & fn}.
```

equips A with the interfaces (factories) or structures f_1, f_2, \dots, f_n . Concretely, it creates a COQ `Record` with a parameter corresponding to A in which each field is one of the interfaces applied to the parameter, a dependent pair whose first field corresponds to A and whose second field is an instance of the record mentioned just above, a module that contains the `Record` and the dependent pair, and unification hints (Cohen *et al.*, 2020, Sect. 3.2). In fine, HIERARCHY-BUILDER commands compile mathematical structures to *packed classes* (Garillot *et al.*, 2009), but the technical details (COQ modules, records, coercions, implicit arguments, canonical structures instances, notations, etc.) are hidden to the user.

One can associate *builders* to a factory by using the `HB.builders` command that opens a COQ section importing the factory, where one defines instances of mixins (the “builders”), before ending with the `HB.end` command. See Sect. 2.3 for an example.

Factories (including mixins) are to be eventually instantiated (command `HB.instance`) with concrete objects. Instances are registered with `.Build` functions that are provided automatically for each factory by HIERARCHY-BUILDER.

2.2 Functors and natural transformations

Our hierarchy of interfaces of effects starts with capturing the notion of functors on the category of sets. In our COQ definition, the domain and codomain of functors are fixed to COQ’s native type `Type`: this a predicative type that can be interpreted as the universe of sets in set-theoretic semantics.⁵

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. The functor laws appear in lines 3 and 4 (they correspond to the standard definitions, see Table 1 for details):

```
1 HB.mixin Record isFunctor (F : Type -> Type) := {
2   actm : forall A B : Type, (A -> B) -> F A -> F B ;
3   functor_id : FunctorLaws.id actm ;
4   functor_o : FunctorLaws.comp actm }.
```

Given a functor F and a function f (seen as a morphism), we denote by $F \# f$ the action of F on f , as a notation for `@actm F _ _ f` (where the prefix `@` is a COQ modifier to make all the arguments of a function explicit).

As explained in Section 2.1, the mixin `isFunctor` only provides an interface, the type of functors is defined by a HIERARCHY-BUILDER structure `Functor` that can be seen as a sigma-type of a carrier F satisfying the interface `isFunctor`:

⁵ In the actual COQ code, we introduce the alias `UU0` for `Type` for practical reasons: although a predicative `Type` is appropriate for the results presented here, there is at least one application of `MONAE` that requires `UU0` to be impredicative (Affeldt & Nowak, 2020). The `UU0` alias provides an easy way to substitute COQ’s `Type` for COQ’s `Set` which can be made impredicative without modifying other parts of our COQ development.

⁶ By convention, free variables in laws are universally quantified.

Table 1. Algebraic laws used in this article.⁶ See Monae (2018, file hierarchy.v) for the code.

Module FunctorLaws.	
id F	$F \text{ id} = \text{id}$
comp F	$F (g \circ h) = F g \circ F h$
Module JoinLaws. (given a functor F)	
right_unit ret join	$\text{join} \circ (F \text{ ret}) = \text{id}$
left_unit ret join	$\text{join} \circ \text{ret} = \text{id}$
	(where ret has F as an implicit parameter)
associativity join	$\text{join} \circ (F \text{ join}) = \text{join} \circ \text{join}$
	(where F is an implicit parameter of the right-most join)
Module BindLaws. (where $\cdot \gg= \cdot$ is a notation for the identifier bind)	
associative bind	$(m \gg= f) \gg= g = m \gg= \lambda x. (f(x) \gg= g)$
left_id op ret	$\text{ret} \text{ op } m = m$
right_id op ret	$m \text{ op } \text{ret} = m$
left_neutral bind ret	$\text{ret} \gg= f = f$
right_neutral bind ret	$m \gg= \text{ret} = m$
left_zero bind z	$z \gg= f = z$
right_zero bind z	$m \gg= z = z$
left_distributive bind op	$m \text{ op } n \gg= f = (m \gg= f) \text{ op } (n \gg= f)$
right_distributive bind op	$m \gg= \lambda x. (f x) \text{ op } (g x) = (m \gg= f) \text{ op } (m \gg= g)$

```
#[short(type=functor)]
HB.structure Definition Functor := { F of isFunctor F }.
```

HIERARCHY-BUILDER uses COQ modules to implement structures. Here, the type of functors is accessible as `Functor.type`; the pragma `#[short(type=functor)]` defines a shorthand for this type, and this is actually the preferred way to refer to it.

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, which is trivial (lines 3, 4):

```
1 Section functorid.
2 Let id_actm (A B : Type) (f : A -> B) : idfun A -> idfun B := f.
3 Let id_id : FunctorLaws.id id_actm. Proof. by []. Qed.
4 Let id_comp : FunctorLaws.comp id_actm. Proof. by []. Qed.
5 HB.instance Definition _ := isFunctor.Build idfun id_id id_comp.
6 End functorid.
```

Similarly, we define an instance for the COQ composition of function (notation `\o`):

```
1 Section functor_composition.
2 Variables F G : functor.
3 Let comp_actm (A B : Type) (h : A -> B) : (F \o G) A -> (F \o G) B :=
4   F # (G # h).
5 Let comp_id : FunctorLaws.id comp_actm.
```

```

6 Proof. (* use functor_id twice *) Qed.
7 Let comp_comp : FunctorLaws.comp comp_actm.
8 Proof. (* use functor_o twice and functional extensionality *) Qed.
9 HB.instance Definition _ := isFunctor.Build (F \o G) comp_id comp_comp.
10 End functor_composition.

```

We now define natural transformations. Given two functors F and G , we formalize the components of natural transformations as a family of functions f , of type `forall A, F A -> G A` (we note this type $F \rightsquigarrow G$ for short) that satisfies the following predicate (recall that \circ denotes function composition):

```

Definition naturality (F G : functor) (f : F ~> G) :=
  forall (A B : Type) (h : A -> B), (G # h) \o f A = f B \o (F # h).

```

Natural transformations are defined by means of the mixin and the structure below:

```

HB.mixin Record isNatural (F G : functor) (f : F ~> G) :=
  { natural : naturality F G f }.

#[short(type=nattrans)]
HB.structure Definition Nattrans (F G : functor) :=
  { f of isNatural F G f }.

```

In MONAE, we note $F \rightsquigarrow G$ instead of `nattrans F G` for the type of natural transformations from F to 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, see Table 1). Furthermore, we add to the mixin an identifier for the bind operator (line 4, hereafter noted $\gg=$ or \gg when the right-hand side ignores its input) 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 is explained below):⁷

```

1 HB.mixin Record isMonad (F : Type -> Type) of Functor F := {
2   ret : idfun ~> F ;
3   join : F \o F ~> F ;
4   bind : forall (A B : Type), F A -> (A -> F B) -> F B ;
5   bindE : forall (A B : Type) (f : A -> F B) (m : F A),
6     bind A B m f = join B ((F # f) m) ;
7   joinretM : JoinLaws.left_unit ret join ;
8   joinMret : JoinLaws.right_unit ret join ;
9   joinA : JoinLaws.associativity join }.
10

```

⁷ MONAE also features a formalization of (concrete) categories that has been used to formalize the geometrically convex monad (Affeldt *et al.*, 2021, 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 article is concerned, this generality is not useful.

```

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 mixin `isMonad` depends on. Hereafter, we also use the notation `Ret` instead of `ret`; this is for technical reasons that have to do with the setting of implicit arguments.

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 : idfun ~> F ;
  join : F \o F ~> F ;
  joinretM : JoinLaws.left_unit ret join ;
  joinMret : JoinLaws.right_unit ret join ;
  joinA : JoinLaws.associativity join }.

HB.builders Context M of isMonad_ret_join M.
Let F := [the functor of M].
Let bind (A B : Type) (m : F A) (f : A -> F B) : F B :=
  bind_of_join join m f.
Let bindE (A B : Type) (f : A -> M B) (m : M A) :
  bind m f = join B ((F # f) m).
Proof. by []. Qed.
HB.instance Definition _ := isMonad.Build M bindE joinretM joinMret joinA.
HB.end.

```

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 (Section 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 (and that in particular does not require naturality of operators):

```

HB.factory Record isMonad_ret_bind (F : Type -> Type) := {
  ret : forall (A : Type), A -> F A ;
  bind : forall (A B : Type), F A -> (A -> F B) -> F B ;
  bindretf : BindLaws.left_neutral bind ret ;
  bindmret : BindLaws.right_neutral bind ret ;
  bindA : BindLaws.associative bind }.

```

The definition of monad (interface `isMonad`) that we presented here is an improvement compared to the original formalization (Affeldt et al., 2019, 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 Monae (2018, file `monad_model.v`) for many instances of the monad structure handled by the `isMonad_ret_bind` factory.

Table 2. Algebraic laws defined in MATHCOMP.

associative op	$x \text{ op } (y \text{ op } z) = (x \text{ op } y) \text{ op } z$
left_id e op	$e \text{ op } x = x$
right_id e op	$x \text{ op } e = x$
left_zero z op	$z \text{ op } x = z$
idempotent op	$x \text{ op } x = x$

2.4 Nondeterminism monad

In the previous section, we explained the case of a simple extension: one structure (the one of monads) that extends another (the one of functors). In this section, we explain how a structure combines several interfaces.

The nondeterminism monad extends both the failure monad and the choice monad. The failure monad `failMonad` extends the class of monads (Section 2.3) with a failure operator `fail` (line 2 below) that is a left-zero of `bind` (line 3). This is the same extension methodology as in Section 2.3:

```

1 HB.mixin Record isMonadFail (M : Type -> Type) of Monad M := {
2   fail : forall A : Type, M A ;
3   bindfailf : BindLaws.left_zero (@bind M) fail }.
4
5 #[short(type=failMonad)]
6 HB.structure Definition MonadFail := { M of isMonadFail M & }.
```

The choice monad `altMonad` extends the class of monads with a choice operator `alt` (line 2 below, infix notation: `[~]`) that is associative (line 3) and such that `bind` distributes leftward over choice (line 4):

```

1 HB.mixin Record isMonadAlt (M : Type -> Type) of Monad M := {
2   alt : forall T : Type, M T -> M T -> M T ;
3   altA : forall T : Type, associative (@alt T) ;
4   alt_bindDl : BindLaws.left_distributive (@bind M) alt }.
5
6 #[short(type=altMonad)]
7 HB.structure Definition MonadAlt := { M of isMonadAlt M & }.
```

See Tables 1 and 2 for the definition of the predicates `associative` and `left_distributive`, respectively.

The nondeterminism monad `nondetMonad` defined below extends both the failure monad and the choice monad (as can be seen at line 2 below, see also Figure 2) and adds laws expressing that failure is an identity of choice (lines 3, 4):

```

1 HB.mixin Record isMonadNondet (M : Type -> Type)
2   of MonadFail M & MonadAlt M := {
3   altfailm : @BindLaws.left_id M (@fail M) (@alt M) ;
4   altmfail : @BindLaws.right_id M (@fail M) (@alt M) }.
5
6 #[short(type=nondetMonad)]
7 HB.structure Definition MonadNondet := { M of isMonadNondet M & }.
```

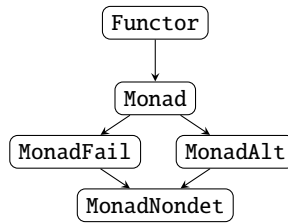


Fig. 2. Hierarchy of effects for the nondeterminism monad.

As a result, a nondeterminism monad can be regarded both as a failure monad or as a choice monad.

2.4.1 Assertions in monadic programs

The introduction of the failure monad already allows for the definition of reusable library functions.

Given a failure monad M , it is customary to define assertions as follows. A computation guard b of type $M \text{ unit}$ fails or skips according to a boolean value b :

```
Definition guard (b : bool) : 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 {A : Type} (p : pred A) (a : A) : M A :=
  guard (p a) >> Ret a.
```

It turns out that in practice it is even useful to define a *dependently typed assertion* that fails or returns a value *together with a proof* that the predicate is satisfied:

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

The notation $\{a \mid p a\}$ is for a dependent pair whose constructor is `exist`. To demonstrate the usefulness of dependently typed assertions, we need a bit of context that will be provided by a concrete example in Section 4.3.1.

2.5 Exception monad

Similarly to the nondeterminism monad that extends the failure monad in the previous section, the exception monad can be defined by extending the failure monad with a catch operator and four laws:

```
HB.mixin Record isMonadExcept (M : Type -> Type) of MonadFail M := {
  catch : forall A, M A -> M A -> M A ;
  catchmfail : forall A, right_id fail (@catch A) ;
  catchfailm : forall A, left_id fail (@catch A) ;
```

```

catchA : forall A, associative (@catch A) ;
catchret : forall A x, @left_zero (M A) (M A) (Ret x) (@catch A) }.

#[short(type=exceptMonad)]
HB.structure Definition MonadExcept := {M of isMonadExcept M & }.

```

See Table 2 for the definitions of the generic predicates used in this interface. As for an application, we send back the reader to the fastprod example explained in Figure 1 (Section 1).

2.6 State monad

The state monad is certainly the first monad that comes to mind when speaking of effects. It denotes computations that transform a state (type S below). It comes with a get operator to yield a copy of the state and a put operator to overwrite it. These functions are constrained by four laws (Gibbons & Hinze, 2011) that appear below at lines 4–8:

```

1  HB.mixin Record isMonadState (S : Type) (M : Type -> Type) of Monad M :=
2  { get : M S ;
3    put : S -> M unit ;
4    putput : forall s s', put s >> put s' = put s' ;
5    putget : forall s, put s >> get = put s >> Ret s ;
6    getput : get >>= put = skip ;
7    getget : forall (A : Type) (k : S -> S -> M A),
8      get >>= (fun s => get >>= k s) = get >>= fun s => k s s }.
9
10 #[short(type=stateMonad)]
11 HB.structure Definition MonadState (S : Type) :=
12   { M of isMonadState S M & }.

```

2.7 Array monad

The array monad extends a basic monad with a notion of indexed array (Mu & Chiang, 2020, Sect. 5.1). Intuitively, it is a generalization of the state monad (Section 2.6). 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 laws of Table 3. 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.

In MONAE, the array monad is formalized by extending the base monad with the following mixin:

```

HB.mixin Record isMonadArray (S : Type) (I : eqType) (M : Type -> Type)
of Monad M := {
  aget : I -> M S ;
  aput : I -> S -> M unit ;

```

Table 3. The laws of the array monad.

aputput	$\text{aput } i \ v \gg \text{aput } i \ v' = \text{aput } i \ v'$
aputget	$\text{aput } i \ v \gg \text{aget } i \ \gg= k = \text{aput } i \ v \gg k \ v$
agetput	$\text{aget } i \ \gg= \text{aput } i = \text{skip}$
agetget	$\text{aget } i \ \gg= (\text{fun } v \Rightarrow \text{aget } i \ \gg= k \ v) =$ $\text{aget } i \ \gg= \text{fun } v \Rightarrow k \ v \ v$
agetC	$\text{aget } i \ \gg= (\text{fun } u \Rightarrow \text{aget } j \ \gg= (\text{fun } v \Rightarrow k \ u \ v)) =$ $\text{aget } j \ \gg= (\text{fun } v \Rightarrow \text{aget } i \ \gg= (\text{fun } u \Rightarrow k \ u \ v))$
aputC ⁸	$i \neq j \ \vee \ u = v \rightarrow$ $\text{aput } i \ u \gg \text{aput } j \ v = \text{aput } j \ v \gg \text{aput } i \ u$
aputgetC	$i \neq j \rightarrow$ $\text{aput } i \ u \gg \text{aget } j \ \gg= k =$ $\text{aget } j \ \gg= (\text{fun } v \Rightarrow \text{aput } i \ u \gg k \ v)$

```

aputput : forall i s s', aput i s >> aput i s' = aput i s' ;
(* other laws similarly mimics the laws of Table 3,
   see (Monae, 2018, file hierarchy.v) for implementation details *)
}.

#[short(type=arrayMonad)]
HB.structure Definition MonadArray (S : Type) (I : eqType) :=
{ M of isMonadArray S I M & }.

```

Note that the type of indices is an `eqType`, that is, a type with decidable equality, as required by the laws of the array monad.

Using the array monad, we can write various functions that manipulate arrays. For example, in any context where `S : Type` and `I : eqType`, we can swap the contents of two cells by combining `aget` and `aput`:

```

Definition aswap {M : arrayMonad S I} (i j : I) : M unit :=
  aget i >>= (fun x => aget j >>= (fun y => aput i y >> aput j x)).

```

Specializing the type of indices to `nat`, we can also recursively call the `aput` operator to write a list `xs` to the array starting from the index `i`:

```

Fixpoint writeList {M : arrayMonad S nat} (i : nat) (s : seq S) : M unit :=
  if s is x :: xs then aput i x >> writeList i.+1 xs else Ret tt.

```

The `SSREFLECT` construct `if x is pat then ... else ...` performs matching on `x` w.r.t. the pattern `pat`.

2.8 Probability monad

Before defining the probability monad, we define a type `{prob R}` with `R` a type for real numbers to represent “probabilities,” that is, real numbers between 0 and 1. This definition

⁸ The law `aputC` is equivalent to a weaker law without the alternative `u = v`; we prefer this form for its better usability.

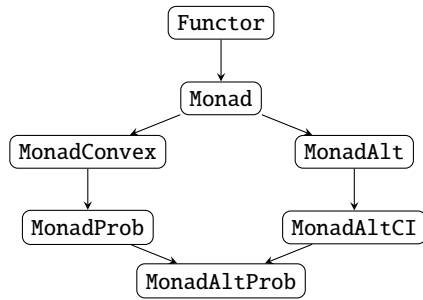


Fig. 3. Hierarchy of effects for the probability monad and the geometrically convex monad.

comes with a notation $p\%:pr$ such that the type $\{\text{prob } R\}$ is automatically inferred based on the shape of the expression p . For example, $1\%:pr$ represents the real number 1 seen as a probability.

In order to define the interface of the probability monad in a modular way, we introduce as an intermediate step the interfaces of “convex monads,” that is, an interface that provides a family of binary choice $\langle p \triangleright \cdot \rangle$ operators parameterized by probabilities p (see line 3 below). These choice operators satisfy the axioms of *convex spaces* (Jacobs, 2010, Def. 3):

- $a \langle 1 \triangleright b \rangle = a$ (line 4),
- $a \langle p \triangleright b \rangle = b \langle \bar{p} \triangleright a \rangle$ where $\bar{p} \stackrel{\text{def}}{=} 1 - p$ (skewed commutativity, line 6),
- $a \langle p \triangleright a \rangle = a$ (idempotence, line 7), and
- $a \langle p \triangleright (b \langle q \triangleright c \rangle) \rangle = (a \langle s(p, q) \triangleright b \rangle) \langle r(p, q) \triangleright c \rangle$ where $s(p, q) \stackrel{\text{def}}{=} \frac{p}{s(p, q)}$ and $r(p, q) \stackrel{\text{def}}{=} \frac{p}{s(p, q)}$ (quasi associativity, line 8).

```

1 HB.mixin Record isMonadConvex {R : realType} (M : Type -> Type)
2   of Monad M := {
3   choice : forall (p : {prob R}) (T : Type), M T -> M T -> M T ;
4   choice1 : forall (T : Type) (a b : M T), choice 1%:pr _ a b = a ;
5   choiceC : forall (T : Type) p (a b : M T),
6     choice p _ a b = choice (p.~%:pr) _ b a ;
7   choicemm : forall (T : Type) p, idempotent (@choice p T) ;
8   choiceA : forall (T : Type) (p q r s : {prob R}) (a b c : M T),
9     choice p _ a (choice q _ b c) =
10    choice [s_of p, q] _ (choice [r_of p, q] _ a b) c }.
  
```

Note that at line 6, the notation $r.\sim$ stands for \bar{r} .

The probability monad merely extends the convex monad by adding the axiom that `bind` left-distributes over probabilistic choice (line 3):

```

1 HB.mixin Record isMonadProb {R : realType} (M : Type -> Type)
2   of MonadConvex R M := {
3   choice_bindDl : forall p,
4     BindLaws.left_distributive (@bind M) (choice p) }.
5
6 #[short(type=probMonad)]
  
```

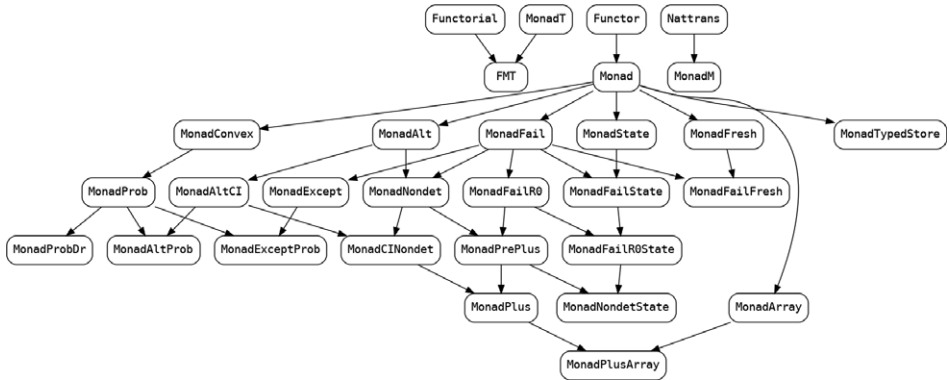


Fig. 4. The complete hierarchy of effects discussed in this article (as generated by the command `HB.graph` of HIERARCHY-BUILDER).

```

7  HB.structure Definition MonadProb {R : realType} :=
8  { M of isMonadProb R M & }.

```

2.8.1 Probability and nondeterminism

Last, we briefly mention the interface of a monad that mixes probability and nondeterminism: the *geometrically convex monad* (Cheung, 2017). Its interface just adds the right-distributivity of probabilistic choice over nondeterministic choice to the probability monad:

```

HB.mixin Record isMonadAltProb {R : realType} (M : Type -> Type)
  of MonadAltCI M & MonadProb R M :=
  { choiceDr : forall T p, right_distributive
    (@choice R M p T) (fun a b => a [~] b) }.

#[short(type=altProbMonad)]
HB.structure Definition MonadAltProb {R : realType} :=
  { M of isMonadAltProb R M & }.

```

The dependencies of the geometrically convex monad are displayed in Figure 3. The formalization of this monad is the topic of previous work (Affeldt et al., 2021) which explains that the construction of its model is important but not trivial.

The complete hierarchy of effects discussed in this article is displayed in Figure 4. MONAE actually contains a few more experimental interfaces (see Monae, 2018).

3 Model of monads

In the previous section (Section 2), we explained how we build a hierarchy of interfaces that allows for combining effects. In this section, we explain how we formalize models to show that the equations in interfaces are indeed consistent. We only consider the examples of monad transformers and of the probability monad; see Monae (2018,

Table 4. Algebraic laws for monad morphisms.

Module MonadMLaws . (where e has type M \rightsquigarrow N)	
ret e	forall A : Type, e A \o Ret = Ret
bind e	forall (A B : Type) (m : M A) (f : A -> M B), e B (m >>= f) = e A m >>= (e B \o f)

file `monad_model.v`) for more examples, and in particular (Affeldt *et al.*, 2021) for the geometrically convex monad.

3.1 Monad transformers

Monad transformers is a well-known approach to combine monads (Liang *et al.*, 1995) that is commonly used to write Haskell programs. The interest in extending MONAE with monad transformers is twofold: it provides an application of monadic equational reasoning (e.g., Jaskelioff's theory of modular monad transformers has been formalized in MONAE Affeldt & Nowak, 2020), and monad transformers can be used to formalize concrete models of monads (as we will see in Section 5.2.2).

Similarly to functors, monads, and interface for effects, monad transformers can be formalized in terms of HIERARCHY-BUILDER interfaces and structures. Given two monads M and N, a monad morphism is a function M \rightsquigarrow N (see Section 2.2 for the definition of \rightsquigarrow) that satisfies the laws of monad morphisms (Benton *et al.*, 2000, Def. 19) (Jaskelioff, 2009a, Def. 7):

```
HB.mixin Record isMonadM (M N : monad) (e : M  $\rightsquigarrow$  N) := {
  monadMret : MonadMLaws.ret e ;
  monadMbind : MonadMLaws.bind e }.
```

Table 4 provides the definitions of the laws. An important property of monad morphisms is that they are natural transformations so that we define the type of monad morphisms using the interface of monad morphisms *and* the interface of natural transformations (Section 2.2):

```
#[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  $\rightsquigarrow$  N) := {
  monadMret : MonadMLaws.ret e ;
  monadMbind : MonadMLaws.bind e }.
```

Upon declaration of this factory, we can use the `HB.builders` command of HIERARCHY-BUILDER to declare instances of the monad morphism and of the natural transformation

interfaces.⁹ Like for monads in Section 2.3, we are again in the situation where the textbook definition ought better be sought in factories rather than in the mixins used to formally define it in the first place.

A monad transformer τ is a function from `monad` to `monad` such that for any monad M it returns a monad morphism from M to τM (Benton *et al.*, 2000, Sect. 3.3) (Jaskelioff, 2009a, Def. 9):

```
HB.mixin Record isMonadT (T : monad -> monad) := {
  Lift : forall M, monadM M (T M) }.

#[short(type=monadT)]
HB.structure Definition MonadT := {T of isMonadT T}.
```

Going one step further, we can define *functorial monad transformers* (Jaskelioff, 2009a, Def. 20) (`Functorial` and `FMT` in Figure 4) and revise the formalization of Jaskelioff's modular monad transformers using `HIERARCHY-BUILDER` following Affeldt & Nowak (2020), see Monae (2018) for details.

3.1.1 Example: The state monad transformer

We explain how we instantiate the type of monad transformer with the example of the state monad transformer. Let us assume some type $S : \text{Type}$ for states and some monad M . First, we define the action on objects of the monad transformed by the state monad transformer:

```
Definition MS := fun A : Type => S -> M (A * S).
```

We also define the unit and the bind operator of the transformed monad:

```
Definition retS : idfun ~-> MS := fun A : Type => curry Ret.
Definition bindS (A B : Type) (m : MS A) f : MS B :=
  fun s => m s >>= uncurry f.
```

Second, we define the monad morphism that will be returned by the lift operator of the monad transformer. In `COQ`, we can formalize the corresponding function by constructing the desired monad assuming M (in a `COQ Section`) and using the factory `isMonad_ret_bind` from Section 2.3. It suffices to prove the laws of the monad:

```
Let bindSretf : BindLaws.left_neutral bindS retS. Proof. ... Qed.
Let bindSmret : BindLaws.right_neutral bindS retS. Proof. ... Qed.
Let bindSA : BindLaws.associative bindS. Proof. ... Qed.
HB.instance Definition _ :=
  isMonad_ret_bind.Build MS bindSretf bindSmret bindSA.
```

Then we define the lift operation as a function that given a computation $m : M A$ returns a computation in the monad MS :

⁹ It would have been less convoluted to define the `isMonadM` interface as inheriting from the `isNatural` interface, but `HIERARCHY-BUILDER` does not support yet natural transformation components as carrier, and this is why we redeclare the `isMonadM` interface as a factory and use `HB.builders` here.

```

Definition liftS (A : Type) (m : M A) : MS A :=
  fun s => m >>= (fun x => Ret (x, s)).
    
```

We can finally prove the lift operations satisfy the monad morphism laws of Table 4:

```

Let retliftS : MonadMLaws.ret liftS. Proof. ... Qed.
Let bindliftS : MonadMLaws.bind liftS. Proof. ... Qed.
HB.instance Definition _ :=
  isMonadM_ret_bind.Build M MS liftS retliftS bindliftS.
    
```

The state monad transformer is defined as an alias stateT to which the type monadT is associated:

```

Definition stateT : Type -> monad -> monad := MS.
HB.instance Definition _ (S : Type) :=
  isMonadT.Build (stateT S) (@liftS S).
    
```

Here, the alias stateT is necessary because isMonadT is expecting a transformer of type monad -> monad. However, when MS is first defined, it is not yet known that its return type is monad. After instantiation with isMonad_ret_bind.Build, this information can be inferred; the alias stateT records the result of this inference for HIERARCHY-BUILDER.

One should wonder what is the relation between the monads that can be built with the state monad transformer and the state monad of Section 2.6. In fact, we can define a Put and a Get operations for a monad MS S M so that it satisfies the laws of state monads:

```

Let bindputput s s' : Put s >> Put s' = Put s'. Proof. ... Qed.
Let bindputget s : Put s >> Get = Put s >> Ret s. Proof. ... Qed.
Let bindgetput : Get >>= Put = skip. Proof. ... Qed.
Let bindgetget (A : Type) (k : S -> S -> stateT S M A) :
  Get >>= (fun s => Get >>= k s) = Get >>= (fun s => k s s).
Proof. ... Qed.
HB.instance Definition _ := @isMonadState.Build
  S (MS S M) Get Put bindputput bindputget bindgetput bindgetget.
    
```

Section 5.2.2 provides an example of application of the state monad transformer (MS): we use it to extend the option monad and build the model of the typed store monad.

3.2 A model of the probability monad

As explained in Section 2.8, a probability monad has to turn each input type into a convex space and provide the bind operator in a way that is compatible with the convex space structure. We can achieve these two requirements by, for an input type A, constructing the type {dist A} of finitely supported distributions over A (Affeldt *et al.*, 2019, Sect. 6.2) (Infotheo, 2018):

$$\{\text{dist } A\} \stackrel{\text{def}}{=} \left\{ f : A \xrightarrow{\text{fin. supp.}} \mathbb{R} \mid (\forall x, 0 < f(x)) \text{ and } \sum_{x \in \text{supp}(f)} f(x) = 1 \right\},$$

$$\text{supp}(f) \stackrel{\text{def}}{=} \{x \in A \mid f(x) \neq 0\}.$$

Technically, we need A to satisfy the axiom of choice to let the finitely supported function have a canonical list representation (appearing as type annotations $A : \text{choiceType}$ in the code Infotheo, 2018).

To complete this definition into a functor, and furthermore, a monad, we can use the factory `isMonad_ret_bind` defined in Section 2.3, which requires us only to provide the two monadic operators `ret` and `bind` and to prove that they satisfy the needed laws.

The `bind` operator computes the weighted sum out of a given distribution $p : M(A)$ and a monadic function $g : A \rightarrow M(B)$, returning a new distribution with probability mass function:

$$b \mapsto \sum_{a \in \text{supp}(g)} p(a) \cdot g(a, b).$$

This function is implemented by the following code (Infotheo, 2018, file `fsdist.v`):

```
(* Section fsdistbind *)
Variables (A B : choiceType) (p : {dist A}) (g : A -> {dist B}).
Let D := \bigcup_{d <- g @ ` finsupp p} finsupp d. (* new support for f *)
Let f : {fsfun B -> R with 0} := (* 0 being the default value *)
[fsfun b in D => \sum_{a <- finsupp p} p a * (g a) b | 0].
```

The resulting operator can be proved to satisfy the monad laws, for example, associativity

```
Lemma fsdistbindA (A B C : choiceType) (m : {dist A}) (f : A -> {dist B})
  (g : B -> {dist C}) :
  (m >>= f) >>= g = m >>= (fun x => f x >>= g).
```

follows from the distributivity of multiplication over big sum, and computation on big unions, both handled by MATHCOMP's big operator library.

The `ret` operator embeds a given point $a : A$ into $M(A)$ as the distribution concentrated on the singleton $\{a\}$ (a.k.a. Dirac's delta), whose probability mass function is

$$x \mapsto \begin{cases} 1 & \text{if } x = a, \\ 0 & \text{otherwise.} \end{cases}$$

The corresponding code in Infotheo (2018) is straightforward:

```
(* Section fsdist1 *)
Variables (A : choiceType) (a : A).
Let D := [fset a]. (* the singleton set containing only a *)
Let f : {fsfun A -> R with 0} := [fsfun b in D => 1 | 0].
```

The other monad laws involving both `ret` and `bind` are proved similarly as above:

```
Lemma fsdist1bind (A B : choiceType) (a : A) (f : A -> {dist B}) :
  fsdist1 a >>= f = f a.
Lemma fsdistbind1 (A : choiceType) (p : {dist A}) :
  p >>= @fsdist1 A = p.
```

We can then pack these definitions and proofs into a monad by applying the builder function from the factory `isMonad_ret_bind` as follows (Monae, 2018, file `proba_monad_model.v`):

```

1 Definition acto : Type -> Type := fun T => {dist (choice_of_Type T)}.
2 Let left_neutral : BindLaws.left_neutral bind ret.
3 Proof. (* just use fsdistbind *) Qed.
4 Let right_neutral : BindLaws.right_neutral bind ret.
5 Proof. (* fsdistbind1 *) Qed.
6 Let associative : BindLaws.associative bind.
7 Proof. (* fsdistbindA *) Qed.
8 HB.instance Definition _ := isMonad_ret_bind.Build
9   acto left_neutral right_neutral associative.

```

Observe that, to define the monad, we need to convert $T : \text{Type}$ into a choiceType before feeding it to $\{\text{dist } _ \}$ (line 1). For that purpose, we assume the generic axiom of choice additionally to COQ's type theory in the form of a function:

$$\text{choice_of_Type} : \text{Type} \rightarrow \text{choiceType}.$$

Composing $\{\text{dist } _ \}$ and choice_of_Type , we obtain the object part of the model M of the probability monad ($M(T) \stackrel{\text{def}}{=} \{\text{dist } (\text{choice_of_Type}(T))\}$).

At this point, we have completed the monad structure on our construction. Since finitely supported distributions carry the convex space structure (thus, easily endowed with a *convex monad* (Section 2.8) structure), the remaining task is to prove the left-distributivity law of bind over the probabilistic choice:

```

(* in infotheo *)
Lemma fsdist_conv_bind_left_distr
  (A B : choiceType) (p : {prob R}) (a b : {dist A}) (f : A -> {dist B}) :
  (a <| p |> b) >>= f = (a >>= f) <| p |> (b >>= f).
Proof. (* proved in 8 lines *) Qed.

(* in monae *)
Let prob_bindDl p :
  BindLaws.left_distributive (@hierarchy.bind acto) (choice p).
Proof. (* just use the infotheo lemma *) Qed.

```

Packing this additional law using the mixin `isMonadProb`, we finally complete the model of the probability monad:

```

HB.instance Definition _ := isMonadProb.Build real_realType
  acto prob_bindDl.

```

4 Application 1: Quicksort

In this section, we formalize the setting of the derivation of quicksort by Mu & Chiang (2020). This derivation relies crucially on nondeterminism: it uses the plus monad (see Section 4.1), several reasoning steps amount to make nondeterministic computations commute (see Section 4.2), and the derivation goal is expressed in terms of refinement which is itself defined using nondeterminism. In Section 4.3, we state the proof goal of the derivation of quicksort. We observe in particular that only stating the goal requires care with dependent types to establish termination in COQ.

Table 5. The third set of laws satisfies by the plus monad (see Section 4.1) ($[\sim]$ is the notation for nondeterministic choice, Section 2.4).

left_zero	<code>forall A B (f : A -> M B), fail A >>= f = fail B</code>
right_zero	<code>forall A B (m : M A), m >> fail B = fail B</code>
left_distributivity	<code>forall A B (m1 m2 : M A) (f : A -> M B), m1 [~] m2 >>= f = (m1 >>= f) [~] (m2 >>= f)</code>
right_distributivity	<code>forall A B (m : M A) (f1 f2 : A -> M B), m >>= (fun x => f1 x [~] f2 x) = (m >>= f1) [~] (m >>= f2)</code>

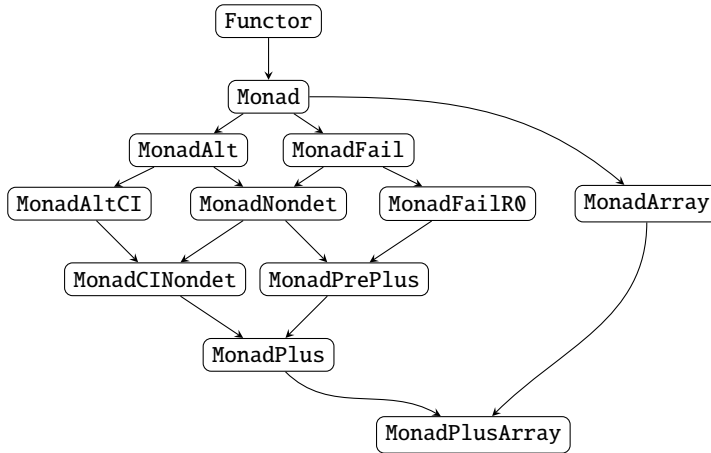


Fig. 5. Hierarchy of effects for the plus monad and the plus array monad.

4.1 Plus monad and plus array monad

First, we extend the hierarchy of Section 2 with the plus monad and the plus array monad. It extends the nondeterminism monad of Section 2.4.

We define the plus monad following Pauwels *et al.* (2019) and Mu & Chiang (2020) (see Mu & Chiang, 2020, Sect. 2). It extends a basic monad with two operators: failure and nondeterministic choice. These are the same operators as the nondeterminism monad (Section 2.4). It, however, satisfies more laws than the nondeterminism monad: (1) failure and choice form a monoid, (2) choice is idempotent and commutative, and (3) failure and choice interact with bind according to the laws of Table 5. We take advantage of existing monads to implement the plus monad. Indeed, we observe that most laws are already available. The monad `failMonad` (Section 2.4) provides the `left_zero` law. The monad `failR0Monad` is used to define the interface of backtrackable state (Affeldt *et al.*, 2019, Sect. 3.2) (`nondetStateMonad` in Figure 4) and in (Affeldt & Nowak, 2020); it introduces the `right_zero` law. The monad `altMonad` (Section 2.4) introduces nondeterministic choice and the `left_distributivity` law. The monad `altCIMonad` extends `altMonad` with commutativity and idempotence of nondeterministic choice. In other words, only the right-distributivity law is missing from previous work.

We therefore implement the plus monad by extending the monads mentioned above with the right-distributivity law, as shown in Figure 5. First, we define the intermediate

prePlusMonad by adding right-distributivity to the combination of nondetMonad and failR0Monad (recall that 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 M) alt }.

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

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

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

Plus array monad The plus array monad is a straightforward extension (Mu & Chiang, 2020, Sect. 5): it combines the plus monad with the array monad of Section 2.7:

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

4.2 Commutativity of nondeterministic computations

In monadic equational reasoning, it often happens that one needs to show that two computations commute, in particular in presence of nondeterminism. The following predicate (Mu, 2019a, Def. 4.2) defines the commutativity of two computations m and n in MONAE:

```
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)) :> M _.
```

The need to show commutativity occurs in particular when dealing with state (be it as in a state monad or in an array monad) and nondeterminism. For example, commutation is possible when a computation in the plus array monad is actually using only nondeterministic operations and therefore does not access the array. To capture computations that are actually using only nondeterministic operations, we define a predicate that characterizes syntactically nondeterminism monads. They are written with the following (higher-order abstract Pfenning & Elliott, 1988) syntax¹⁰ where each constructor reflects the eponymous monadic construct:

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

¹⁰ This encoding is reminiscent of *free monads* (Swierstra & Baanen, 2019).

```
| bind : forall B A, t B -> (B -> t A) -> t A
| fail : forall A, t A
| alt : forall A, t A -> t A -> t A.
```

Let `sem` be a function that computes the semantics of the above syntax in the nondeterminism monad:

```
Fixpoint sem {M : nondetMonad} {A} (m : t A) : M A :=
  match m with
  | ret A a => Ret a
  | bind A B m f => sem m >>= (sem \o f)
  | fail A => fail (* operator of the failure monad *)
  | alt A m1 m2 => sem m1 [~] sem m2
end.
```

Using the above syntax, we can for example write a predicate that captures computations in a `plusMonad` that are semantically just computations in the nondeterminism monad:

```
Definition plus_isNondet {M : plusMonad} A (n : M A) := {m | sem m = n}.
```

Recall that the notation $\{m \mid P m\}$ is for dependent pairs, as already explained in Section 2.4.1.

Eventually, it becomes possible to prove by induction on the syntax (and using the laws of the plus monad) that two computations `m` and `n` in the plus monad commute when `m` only uses nondeterministic operators:

```
Context {M : plusMonad}.
Lemma plus_commute A (m : M A) B (n : M B) C (f : A -> B -> M C) :
  plus_isNondet m -> commute m n f.
```

Using this lemma, to show that two computations in the plus monad commute, it suffices to show that one of the two satisfies the predicate `plus_isNondet`. When a computation does satisfy this predicate, it is good practice to prove it right away and register this fact as a hint in COQ so that when one uses the lemma `plus_commute` generated subgoals are discharged automatically. Section 4.3.3 provides a concrete example.

4.3 Quicksort derivation

This section formalizes the setting of Mu & Chiang (2020). First, we specify sorting functions. Second, we define quicksort using the array monad. Finally, we state the goal of the derivation of quicksort using refinement.

4.3.1 Nondeterministic computation of permutations

We formalize a function `qperm` that computes nondeterministically permutations. In the following, `M` is assumed to be of type `altMonad` (Section 2.4).

The first step is to define a function that splits a list nondeterministically. This is a structurally recursive function that can be encoded using the `Fixpoint` command of COQ. Recall that `[~]` is the notation for nondeterministic choice:

```

Fixpoint splits s : M (seq A * seq A) :=
  if s is h :: t then
    splits t >>= (fun xy => Ret (h :: xy.1, xy.2) [~] Ret (xy.1, h :: xy.2))
  else
    Ret ([:], [ :]).

```

The return type of the `splits` function is $M (\text{seq } A * \text{seq } A)$. We now write another function `splits_bseq` with the same semantics but whose return type contains size information that is useful to establish the termination of functions calling it. The return type of this new function is $M ((\text{size } s).\text{-bseq } A * (\text{size } s).\text{-bseq } A)$, where s is the input list and $n.\text{-bseq } A$ is the type of lists of size less than or equal to¹¹ n :

```

Fixpoint splits_bseq s : M ((size s).-bseq A * (size s).-bseq A) :=
  if s is h :: t then
    splits_bseq t >>= (fun '(x, y) =>
      Ret ([bseq of h :: x], widen_bseq (leqnSn _) y) [~]
      Ret (widen_bseq (leqnSn _) x, [bseq of h :: y]))
  else
    Ret ([bseq of [ : ]], [bseq of [ : ]]).

```

Provided that one ignores the notations and lemmas about bounded-size lists, the body of this definition is the same as the original `splits`. The notation `[bseq of [:]]` is for an empty list seen as a bounded-size list. The lemma `widen_bseq` captures the fact that a $m.\text{-bseq } T$ list can be seen as an $n.\text{-bseq } T$ list as long as $m \leq n$:

Lemma `widen_bseq T m n : m <= n -> m.-bseq T -> n.-bseq T`.

Since `leqnSn n` is a proof of $n \leq n+1$, we understand that `widen_bseq (leqnSn _)` turns a $n.\text{-bseq}$ list into an $n+1.\text{-bseq}$ list. The notation `[bseq of x :: ys]` is a MATHCOMP idiom that triggers type inference using canonical structures to build a $n+1.\text{-bseq}$ list using the fact that `ys` is itself a $n.\text{-bseq}$ list.

We can now write a function `qperm` that computes permutations nondeterministically. For this purpose, we use the COQ Equations command (Sozeau, 2009; Sozeau & Mangin, 2019) that provides support to prove the termination of functions whose recursion is not structural:

```

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

```

The annotation `by wf (size s) lt` indicates that the relation between the sizes of lists is well founded. The function `liftM2` is a generic monadic function that lifts a function h of type $A \rightarrow B \rightarrow C$ to a monadic function of type $M A \rightarrow M B \rightarrow M C$. Once the Equations command is processed, COQ asks for a proof that the size of the arguments are indeed decreasing. This fact is provable thanks to the additional type information returned by `splits_bseq`. Under the hood, COQ uses the accessibility predicate (Bertot & Castéran,

¹¹ This type of *bounded-size lists* comes from the MATHCOMP library (MathComp, 2024, file `tuple.v`).

2004, Chapter 15). Note that `qperm` is not the most obvious definition for the task of generating nondeterministically permutations, but it is a good fit to specify quicksort (Mu & Chiang, 2020, Sect. 3).

Specification of sorting functions The function `qperm` provides a way to specify sorting functions. Indeed, it suffices to generate all the permutations and filter the sorted ones. Let us assume that `M` has type `plusMonad` (Section 4.1) and that `T` is an ordered type (as defined in `MATHCOMP`). We can use the generic `MATHCOMP` predicate `sorted` and define an obviously correct sorting “algorithm” using `qperm`, the Kleisli symbol `>=>`, and the assertion `assert` of Section 2.4.1:

```
Local Notation sorted := (sorted <=%0).
```

```
Definition slowsort : seq T -> M (seq T) := qperm >=> assert sorted.
```

4.3.2 In-place quicksort

We can now formally define quicksort following Mu & Chiang (2020).

The partition step is performed by the function `ipartl`, which uses the array monad (Section 2.7). The parameter `T` below is a generic ordered type for the array contents:

```
Variable (M : arrayMonad T nat).
Fixpoint ipartl p i ny nz nx : M (nat * nat) :=
  if nx is k.+1 then
    aget (i + (ny + nz)) >>= (fun x =>
      if x <= p then
        aswap (i + ny) (i + ny + nz) >> ipartl p i ny.+1 nz k
      else
        ipartl p i ny nz.+1 k)
  else Ret (ny, nz).
```

The call `ipartl p i ny nz nx` partitions the subarray ranging from index `i` (included) to index `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 (see Section 2.7 for the `aswap` function). Otherwise, partition proceeds with a `zs` enlarged by one element.

Note that since `ipartl` is written as a computation in the array monad, it represents a total function giving a single result; in particular, it cannot result in failure in the sense of `failMonad`.

Quicksort is not structurally recursive, and its definition in `COQ` using a shallow embedding therefore requires care to convince the termination checker that it is really terminating. Here, we formalize it using the `Fix` definition from the `Coq.Init.Wf` module for

well-founded fixpoint of the standard library of COQ.¹² First, we write using the `Program` command for dependent-type programming (The Coq Development Team, 2024, Chapter Program) an intermediate function `iqsort'` that implements the same logic as the intended quicksort function except that it does not call itself recursively but instead calls a parameter function (`f` at line 3 below). The function `iqsort'` takes as input a pair of an index and of a size (`ni`); it is a computation of the `unit` type. The parameter function `f` takes as an additional argument a proof that the size of the input list is decreasing. These corresponding proofs appear as holes (the `_` syntax) to be filled by the user once the declaration is processed. The code selects a pivot (line 6), calls the partition function (line 7), swaps two cells (line 10), and then calls the parameter function on the partitioned arrays (we explain below the use of the plus array monad):

```

1  Variables (d : unit) (T : orderType d) (M : plusArrayMonad T nat).
2  Program Fixpoint iqsort' ni
3    (f : forall mj, mj.2 < ni.2 -> M unit) : M unit :=
4    match ni.2 with
5    | 0 => Ret tt
6    | n.+1 => aget ni.1 >>= (fun p =>
7      dipartl p ni.1.+1 0 0 n >>= (fun nyz =>
8        let ny := nyz.1 in
9        let nz := nyz.2 in
10       aswap ni.1 (ni.1 + ny) >>
11       f (ni.1, ny) _ >> f (ni.1 + ny.+1, nz) _)
12  end.

```

The partition function is not exactly the `ipartl` function that we explained at the beginning of this section but a dependently-typed version `dipartl` 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 `y`, `z`, and `x` are the sizes of the lists input to `ipartl` and this dependent type captures at the level of types 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 the predicate for dependently typed assertions of Section 2.4.1:

```

Definition dipartl 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)].

```

Finally, the wanted `iqsort` function can be written using `Fix`. This requires a (trivial) proof that the order chosen for the measure is well founded:

¹² Contrary to Section 4.3.1, we have not been able to use the `Equations` command here for technical reasons. The proof of termination of `iqsort` relies on type information carried by the return type of `dipartl` that the default setting of `Equations` fails to preserve. The `Program/Fix` is essentially the manual version of the `Equations` approach; it is less user-friendly, but in practice it seems more widely applicable.

```

1 Lemma well_founded_lt2 :
2   well_founded (fun x y : nat * nat => x.2 < y.2).
3 Proof. (* see (Monae, 2018, file example_iquicksort.v) *) Qed.
4 Definition iqsort : nat * nat -> M unit :=
5   Fix well_founded_lt2 (fun _ => M unit) iqsort'.

```

The example of in-place quicksort is an example of the unfortunate but unavoidable fact that programs must be terminating to be shallow-embedded and verified in a proof assistant. Understanding the support such as the Equations command or the Fix construct in COQ as well as the need to sometimes reflect size information at the level of types is important in practice and the in-place quicksort provides a good example. In theory, *sized types* have been shown to be useful to guarantee the termination of programs such as quicksort (Barthe *et al.*, 2008). However, as of today, it appears that users of the COQ proof assistant still need a support library to prove termination manually, since there are indications that sized types for COQ might not be practical (Chan *et al.*, 2023).

The use of `dipart1` in lieu of `ipart1` explains why `iqsort` uses the plus array monad, which is the only array monad that provides the failure operator needed by `dassert`. Yet, its semantics does not rely on the fail operator. Indeed, it can be shown that, even though `dipart1` is written using `dassert`, it is the same computation as `ipart1` ignoring the proof part of `dipart1`:

```

Lemma ipart1E p i y z x :
  ipart1 p i y z x = (M # sval) (dipart1 p i y z x).

```

The standard way to take the first projection of a dependent pair in MATHCOMP is the `sval` function. We can use this fact to show that the recursive call of `iqsort` can be captured by the following equation that uses `ipart1` (in the array monad) instead of `dipart1`:

```

Lemma iqsort_cons i (n : nat) : iqsort (i, n.+1) =
  aget i >>= (fun p => ipart1 p i.+1 0 0 n >>= (fun nyz =>
    let '(ny, nz) := nyz in
    aswap i (i + ny) >>
    iqsort (i, ny) >> iqsort (i + ny.+1, nz))).

```

This provides an evidence that `iqsort` cannot fail in any reasonable model where `fail` is distinct from other operations.

4.3.3 Statement of the derivation of quicksort

To state the correctness of `iqsort`, Mu & Chiang (2020) define a notion of program refinement that relates two monadic computations using nondeterministic choice:

```

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

```

As the notation indicates, `m1 `<=` m2` represents a relationship akin to set inclusion, which means that the result of `m1` is included in the one of `m2`: we say that `m1` *refines* `m2`.

The specification of the derivation of quicksort can now be written as a refinement relation between, on the one hand, a program that writes a list to memory (see Section 2.7) and

then calls `iqsort`, and, on the other hand, a call to `slowsort` (see Section 4.3.1) followed by a program that writes the result to memory (Mu & Chiang, 2020, Eqn 12):

```
Lemma iqsort_slowsort i xs :
  writeList i xs >> iqsort (M := M) (i, size xs) `<=`
  slowsort xs >>= writeList i.
```

The complete formal proof can be found in Monae (2018, file `example_iquicksort.v`). Let us illustrate this formal proof with an excerpt. In the course of deriving in-place quicksort, Mu & Chiang (2020) mutate an array $ys ++ zs ++ [x]$ to $ys ++ [x] ++ zs'$ where zs' is a permutation of zs by swapping x with the leftmost element of zs . This is formalized by the following refinement relation (Mu & Chiang, 2020, Eqn 11, p. 12):

$$\begin{aligned} \text{writeList } (i + |ys|) (zs ++ [x]) &\ggg \text{swap } (i + |ys|) (i + |ys| + |zs|) \\ &\subseteq \\ \text{perm } zs &\ggg \lambda zs'. \text{writeList } (i + |ys|) ([x] ++ zs'). \end{aligned}$$

In MONAE, we formalize the above relation as the following lemma (where M is the plus array monad):

```
Lemma refin_writeList_rcons_cat_aswap (i : nat) x (ys zs : seq E) :
  writeList i (rcons (ys ++ zs) x) >>
  aswap (M := M) (i + size ys) (i + size (ys ++ zs))
  `<=`
  qperm zs >>= (fun zs' => writeList i (ys ++ x :: zs')).
```

There is almost a one-to-one match with the pencil-and-paper version just above (the only difference is that we found it slightly more practical to have the index i as a parameter instead of an arithmetic expression). Incidentally, the proof of the above lemma provides an application of commutativity (Section 4.2):

```
qperm zs >>= (fun s => writeList i ys >> writeList (i + size ys) (x :: s))
= [[rewrite (plus_commute (qperm zs))]]. ]
writeList i ys >> (qperm zs >>= (fun s => writeList (i + size ys) (x :: s)))
```

This proof step can be performed as a single tactic provided that `qperm` has been proved to satisfy the `plus_isNondet` predicate of Section 4.2 and that this fact has been registered as a hint in COQ.

5 Application 2: ML programs with references

In order to verify OCaml programs, we extend MONAE with a new monad and its equational theory so that programs generated by COQGEN (Garrigue & Saikawa, 2022), an OCaml backend that outputs COQ code, can be verified using monadic equational reasoning. We call this monad the *typed store monad* because it consists essentially of a mutable typed store. The original goal of COQGEN was ensuring the soundness of OCaml type inference by exploiting the richness of COQ's type system. Using this approach, we can effectively reuse the output of COQGEN, instead of discarding it as a mere witness of type checking, and harness it as a target for formal verification. In this section, we do not give a

full account of the typed store monad, as its theory is still in flux, but explain our methodology to develop a new monad, and demonstrate how it can handle a nontrivial example involving cyclic lists.

5.1 Representation of OCaml types

The typed store having heterogeneous contents, we need a concrete representation of OCaml types in order to handle the dynamic typing required to access it. In COQGEN, this is achieved through an inductive type `ml_type` of syntactic type representations (as plain terms), and a computable function `coq_type`, interpreting them into COQ types in the style of a Tarski universe (Aczel, 1977; Martin-Löf, 1984), which can be abstracted as the following module interface from the source code of COQGEN (CoqGen, 2021):

```
Parameter ml_type : Set.
Variant loc : ml_type -> Type := mkloc T : nat -> loc T.
Parameter coq_type : forall N : Type -> Type, ml_type -> Type.
```

The `loc` identifier above is for memory locations. The interpretation function `coq_type` (Garrigue & Saikawa, 2022) is parameterized by a monad `N`, used to translate function types. Concrete definitions for `ml_type` and `coq_type` are generated by the COQGEN compiler. The presence of the Tarski universe is the main difference with other transpilers from functional programming languages to COQ, such as `coq-of-ocaml` (Claret, 2018) and `hs-to-coq` (Spector-Zabusky et al., 2018).

In COQGEN, the concrete monad used to run programs is obtained as a fixpoint involving `coq_type`, which enables higher-order stores but requires bypassing the so-called strict-positivity requirement of inductive types, inducing a logical inconsistency. As far as the primary goal of COQGEN is concerned, this is a reasonable choice because type soundness is not directly affected.

In the context of MONAE, we restrict ourselves to programs that do not store functions that themselves access the store; there is therefore no need for higher-order stores and we can work in a consistent setting.

5.2 Designing an equational theory from an implementation

As COQGEN itself is built around a monad which models OCaml computations, it seemed natural to use MONAE for reasoning with it. While COQGEN supports a large subset of OCaml, including references and exceptions, here we limit ourselves to references, with reference creation, access, and update.

5.2.1 Interface for effects

In MONAE, these effects are introduced as follows, starting with our Tarski universe. The `ml_nonempty` and `val_nonempty` fields are only here to make the construction of the monad model easier:

```

HB.mixin Structure isML_universe (ml_type : Type) := {
  coq_type : forall M : Type -> Type, ml_type -> Type ;
  ml_nonempty : ml_type ;
  val_nonempty : forall M, coq_type M ml_nonempty }.

#[short(type=ML_universe)]
HB.structure Definition ML_UNIVERSE :=
  { ml_type & isML_universe ml_type & Equality ml_type }.

HB.mixin Record isMonadTypedStore (MLU : ML_universe) (N : monad)
  (locT : eqType) (M : Type -> Type) of Monad M := {
  cnew : forall {T : MLU}, coq_type N T -> M (loc locT T) ;
  cget : forall {T}, loc locT T -> M (coq_type N T) ;
  cput : forall {T}, loc locT T -> coq_type N T -> M unit ;
  ... }.

```

See Monae (2018, file hierarchy.v). Note that, in order to preserve the soundness of our logic, we do not take a fixpoint so that this interface uses two different monads: M for computations using the typed store and N for computations by values in the store.¹³

5.2.2 Definition of the model

The next step is to provide a model for this monad. We started with the model from COQGEN but soon realized that it would be difficult to use it to prove laws, as it required too many internal invariants. Fortunately, we could quickly come up with an alternative model, whose semantics is essentially identical, but which does not require explicit invariants:

```

Variables (MLU : ML_universe) (N : monad).
Local Notation coq_type := (@coq_type MLU N). (* coq_type field of MLU *)
Local Notation ml_type := (MLU : Type).      (* ml_type field of MLU *)

Record binding :=
  mkbind { bind_type : ml_type; bind_val : coq_type bind_type }.

Definition M : Type -> Type := MS (seq binding) option_monad.

Let cnew T (v : coq_type T) : M (loc T) :=
  fun st => let n := size st in
    Ret (mkloc T n, rcons st (mkbind v)).

Let cget T (r : loc T) : M (coq_type T) :=
  fun st =>
    if nth_error st (loc_id r) is Some (mkbind T' v) then
      if coerce T v is Some u then Ret (u, st) else fail
    else fail.

```

See Monae (2018, file monad_model.v). Here, we just reuse the state monad transformer of MONAE (Section 3.1.1), applied to the option monad, to allow computations to fail.

¹³ Currently, the relation between these two monads is not yet clear. It seems that the existence of a monad morphism $N \rightarrow M$ is desired, but at the time of this writing, we have not investigated it.

Table 6. Laws adapted from the array monad.

cgetputskip	$\text{cget } r \gg= \text{cput } r = \text{cget } r \gg \text{skip}$
cputC	$\text{cput } r_1 s_1 \gg \text{cput } r_2 s_2 = \text{cput } r_2 s_2 \gg \text{cput } r_1 s_2$ if $\text{loc_id } r_1 \neq \text{loc_id } r_2 \vee \text{JMeq } s_1 s_2$
cputgetC	$\text{cput } r_1 s \gg (\text{cget } r_2 \gg= k) = \text{cget } r_2 \gg= \lambda v. \text{cput } r_1 s \gg k v$ if $\text{loc_id } r_1 \neq \text{loc_id } r_2$
cgetputC	$\text{cget } r_1 \gg \text{cput } r_2 s = \text{cput } r_2 s \gg (\text{cget } r_1 \gg \text{skip})$

Interestingly, failure is only an internal feature of the model and does not appear in the interface: correctly translated OCaml programs never fail on memory access, and none of the laws we present here mentions failure.¹⁴ The state itself is a list of dynamically typed values (binding) combining a type representation and a value of this type. The type `loc` will contain a position in this list. The definitions of `cnew` and `cget` are introduced by `Let` as we only need them to define an instance of our interface. In the implementation of `cnew`, we generate a new reference cell by adding a new value at the end of the list, using the `rcons` operation. Since we use the length of the list as location, we can be sure that this location was invalid before calling `cnew`. We also show the code for `cget`, which demonstrates how we access the store using a standard library function, `nth_error`, which returns `None` if the index is out of the list. We then use `coerce T v`, which uses the decidable equality of `mL_type` to coerce a value to an expected type but again returns `None` if the dynamic type representation for `v` is not `T`. We can see that `cget T r` will only succeed on a store `st` if `r` is a valid location which contains a value of type `T`. The implementation of `cput` is similar to `cget`.

5.2.3 Extending the interface with laws

We move to the next step of our methodology: from this model, we infer useful laws which we will prove and add to the interface.

Adapting an existing theory As a starting point, we mimicked the array monad, which already contains laws for `aput` and `aget` (see Table 6). The only difference is that now types may differ, so we only show rules `cgetputskip`, `cputC` and `cputgetC`, as `cputput`, `cputget`, `cgetget`, and `cgetC` are identical to their array monad counterparts.

The most interesting change may be `cgetputskip`. Since our judgments do not include a context part, we cannot guarantee that access to `r` will be valid for any store. So the left-hand side might fail and is not equivalent to `skip`. As a first approximation, we replace it with a call to `cget`, which would exhibit the same failure and ignore its result by composing with `skip`. We will see later that this can be seen as a separate operation, which checks the validity of a reference. For `cputC`, there are two changes. The first one is that we have a specific notion of identity for locations, which allows us to compare locations of different types. The second one is that, if we want to check the equality of the updated values,

¹⁴ If we add an effect `crun` that discards state, and returns only the result of a computation, failure becomes visible in the interface. However, OCaml does not provide such a function.

Table 7. Laws for `cnew` and `cchk` (excerpt).

<code>cnewget</code>	$\text{cnew } s \gg= \lambda r. \text{cget } r \gg= k \ r = \text{cnew } s \gg= \lambda r. k \ r \ s$
<code>cnewput</code>	$\text{cnew } s \gg= \lambda r. \text{cput } r \ t \gg k \ r = \text{cnew } t \gg= k$
<code>cnewchk</code>	$\text{cnew } s \gg= \lambda r. \text{cchk } r \gg k \ r = \text{cnew } s \gg= k$
<code>cchknewC</code>	$\text{cchk } r_1 \gg (\text{cnew } s \gg= \lambda r_2. \text{cchk } r_1 \gg k \ r_2) = \text{cchk } r_1 \gg (\text{cnew } s \gg= k)$
<code>cchknewE</code>	$\text{cchk } r_1 \gg (\text{cnew } s \gg= k_1) = \text{cchk } r_1 \gg (\text{cnew } s \gg= k_2)$ if $k_1 \ r_2 = k_2 \ r_2$ for all r_2 s.t. $\text{loc_id } r_1 \neq \text{loc_id } r_2$
<code>cchkputC</code>	$\text{cchk } r_1 \gg \text{cput } r_2 \ s = \text{cput } r_2 \ s \gg \text{cchk } r_1$
<code>cgetputchk</code>	$\text{cget } r \gg= \text{cput } r = \text{cchk } r$

we need to use the John Major equality (McBride & McKinna, 2004), which can still be written (but not proved) whenever the types differ. The law `cputgetC` is almost unchanged, but we added a variant of it, `cgetputC`, where the result of `cget` is ignored, and which requires no side condition.

Inferring new laws through experimentation We then went on and added rules for `cnew`. The intuition is that they should be similar to rules for `cput`, as they define the contents of a reference, but there appears a new difficulty, as there is no way to refer to the location of this reference before it is created. This is only after experimenting with sample programs that we realized that this could be exploited the other way around: if a reference is valid before the creation of another one, then they cannot be equal. And since the only way to check validity is through an operation in the monad, we introduced `cchk` *r* as an abbreviation for `cget` *r* \gg `skip`. We show in Table 7 a non-exhaustive list of laws involving `cnew` and `cchk`. Note that some rules are intended to be used in the opposite direction, for instance, `cnewchk` allows to introduce a new `cchk`, which `cchknewC` duplicates under another `cnew`. The law `cchknewE` uses the same mechanism to infer an inequation $\text{loc_id } r_1 \neq \text{loc_id } r_2$ between locations, which can in turn be assumed to prove that continuations k_1 and k_2 are equal (possibly using rules `cputC` or `cputgetC`).

5.3 Examples

During this process, we verified a number of example programs (Monae, 2018, file `example_typed_store.v`): cyclic lists, fibonacci, and factorial, all implemented in OCaml using references, and transpiled using COQGEN. Let us consider the verification in COQ of the following implementation of cyclic lists in OCaml:

```
type 'a rlist = Nil | Cons of 'a * 'a rlist ref

let cycle a b =
  let r = ref Nil in
  r := Cons (a, ref (Cons (b, r)));
  r

let rtl r = match !r with Nil -> r | Cons (_, l) -> l
```

Generating the corresponding COQ formalization is as easy as running `ocamlc -c -coq cycle.ml`.¹⁵ We only have to annotate uses of `coq_type` with the appropriate monad (`M` or `N`, while `COQGEN` only uses `M`):

```

Inductive ml_type : Set :=
| ml_int : ml_type
| ml_bool : ml_type
| ml_unit : ml_type
| ml_ref : ml_type -> ml_type
| ml_arrow : ml_type -> ml_type -> ml_type
| ml_rlist : ml_type -> ml_type.

... (* Proof of decidable equality for ml_type *)

Inductive rlist (a : Type) (a_1 : ml_type) :=
| Nil : rlist a a_1
| Cons : a -> loc (ml_rlist a_1) -> rlist a a_1.

Fixpoint coq_type_nat (M : Type -> Type) (T : ml_type) : Type :=
  match T with
  | ml_int => nat (* coqgen rather uses PrimInt63.int *)
  | ml_bool => bool
  | ml_unit => unit
  | ml_arrow T1 T2 => coq_type_nat M T1 -> M (coq_type_nat M T2)
  | ml_ref T1 => loc T1
  | ml_rlist T1 => rlist (coq_type_nat M T1) T1
  end.

(* Create an ML universe instance *)
HB.instance Definition _ :=
  @isML_universe.Build ml_type coq_type_nat ml_unit val_nonempty.

(* Use the corresponding interface *)
Variables (N : monad) (M : typedStoreMonad ml_type N monad_model.locT_nat).

Definition cycle T (a b : coq_type N T) : M (loc (ml_rlist T)) :=
  do r <- cnew (ml_rlist T) (Nil (coq_type N T) T);
  do l <-
    (do v <- cnew (ml_rlist T) (Cons (coq_type N T) T b r);
     Ret (Cons (coq_type N T) T a v));
  do _ <- cput r l; Ret r.

Definition rtl T (r : loc (ml_rlist T)) : M (loc (ml_rlist T)) :=
  do v <- cget r; match v with | Nil => Ret r | Cons _ l => Ret l end.

```

Note that `rlist`, being defined before `coq_type_nat`, has to take two parameters corresponding to the same type, which are later related by `coq_type`.

As for the correctness statement, we will only check that the list created by `cycle` is indeed a cycle of length 2:

¹⁵ For the implementation, see [CoqGen \(2021\)](#).

Table 8. Derivation of `rtl_tl_self`.

Freshly inserted subexpressions are underlined. The notation $x ::: y$ stands for `Cons _ _ x y`.

```

(cnew Nil >>= λr.(cnew (f ::: r) >>= λv.ret (t ::: v)) >>= λl.cput r l >> ret l)
  >>= λl.rtl l >>= rtl
= [ rewrite bindA -cnewchk. ] (* insert cchk *)
cnew Nil >>= λr.cchk r >> (((cnew (f ::: r) >>= λv.ret (t ::: v)) >>= λl.cput r l >> ret l)
  >>= λl.rtl l >>= rtl)
= [ under eq_bind => r1. ] (* go under binder *)
'Under [ cchk r >> (((cnew (f ::: r) >>= λv.ret (t ::: v)) >>= λl.cput r l >> ret l)
  >>= λl.rtl l >>= rtl) ]
= [ rewrite bindA; under eq_bind do rewrite !bindA. ]
'Under [ cchk r >> (cnew (f ::: r) >>= λv.(ret (t ::: v) >>= λl.cput r l >> ret l)
  >>= λl.rtl l >>= rtl) ]
= [ under cchknewE => r2 r1r2. ] (* deduce r1r2 from cchk >> cnew *)
hypothesis r1r2 : loc_id r1 != loc_id r2 introduced
'Under [ (ret (t ::: r2) >>= λl.cput r1 l >> ret r1) >>= λl.rtl l >>= rtl l1 ]
= [ rewrite bindretf bindA bindretf. ] (* substitutions *)
'Under [ cput r1 (t ::: r2) >> (rtl r1 >>= rtl) ]
= [ rewrite bindA cputget. ] (* evaluate (rtl r1) *)
'Under [ cput r1 (t ::: r2) >> (ret r2 >>= rtl) ]
= [ rewrite bindretf. ] (* substitution *)
'Under [ cput r1 (t ::: r2) >> rtl r2 ]
= [ rewrite -bindA. ] (* decompose rtl *)
'Under [ (cput r1 (t ::: r2) >> cget r2) >>= λv.ret (match v with ... end) ]
= [ rewrite cputgetC //. ] (* commute using r1r2 *)
'Under [ cget r2 >>= λv.cput r1 (t ::: r2) >> ret (match v with ... end) ]
= [ over. ] (* close the "under cchknewE" tactic *)
hypothesis r1r2 discharged
'Under [ cchk r1 >> (cnew (f ::: r1) >>= λr2.cget r2
  >>= λv.cput r1 (t ::: r2) >> ret (match v with Nil => r2 | _ ::: t => t end)) ]
= [ rewrite cnewget. ] (* substitute v in continuation *)
'Under [ cchk r1 >> (cnew (f ::: r1) >>= λr2.cput r1 (t ::: r2) >> ret r1) ]
= [ over. ] (* close the "under eq_bind" tactic *)
cnew Nil >>= λr1.cchk r1 >> (cnew (f ::: r1) >>= λr2.cput r1 (t ::: r2) >> ret r1)
= [ rewrite cnewchk. ]
cnew Nil >>= λr1.cnew (f ::: r1) >>= λr2.cput r1 (t ::: r2) >> ret r1

```

```

Lemma rtl_tl_self T (a b : coq_type N T) :
  do l <- cycle T a b; do l1 <- rtl l; rtl l1 = cycle T a b.

```

We show the derivation of this equality in Table 8, where we use the `under` tactic (Martin-Dorel & Tassi, 2019) to perform rewriting under λ -abstractions provided an appropriate extensionality lemma. Written as a proof script it is just 8 lines long.

We have also proved the same result for cycles of length n . In that case, it requires several lemmas, which together take about 60 lines to prove. We hope to be able to eventually improve scalability.

6 Related work

6.1 Formalization of monads in COQ

Monads have been widely used to model programming languages with effects in the COQ proof assistant.

The main motivation is the verification of programs. For example, monads have been used in COQ to verify low-level systems (Jomaa *et al.*, 2018), to verify effectful Haskell programs (Christiansen *et al.*, 2019), or for the modular verification of low-level systems based on free monads (Letan *et al.*, 2018). More directly related to the application of Section 4, Sakaguchi provides a formalization of the quicksort algorithm in COQ using the array state monad (Sakaguchi, 2020, Sect. 6.2). His formalization is primarily motivated by the generation of efficient executable code. This makes for an intricate definition of quicksort (e.g., all the arguments corresponding to indices are bounded). Though his framework does not prevent program verification (Sakaguchi, 2020, 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).

Monads have also been used in COQ to reason about the meta-theory of programming languages. For example, Delaware *et al.* (2013) formalize several monads and monad transformers in COQ, each one associated with a so-called feature theorem. When monads are combined, these feature theorems can then be combined to prove type soundness

There are of course formalizations of monads in other proof assistants. To pick one example that can be easily compared with a formalization in MONAE, one can find a formalization of the Monty Hall problem in Isabelle/HOL (Cock, 2012) using the pGCL programming language (to compare with Affeldt *et al.*, 2021, Sect. 2.3).

The idea to use packed classes (Garillot *et al.*, 2009) to represent a hierarchy of monad interfaces has been experimented in Affeldt *et al.* (2019). Packed classes are known to scale up to deep hierarchies of interfaces, but their direct usage involves a mix of techniques that lead to verbose code. The tool HIERARCHY-BUILDER (Cohen *et al.*, 2020) provides automation to hide technical details from the user. It is also more robust. Indeed, we discovered that our previous work (Affeldt & Nowak, 2020, Fig. 1) lacked an intermediate interface, which required us to insert some type constraints for type inference to succeed (see Hierarchy Builder, 2021 for details). HIERARCHY-BUILDER detects such omissions automatically. The better-known type classes have been reported to replace packed classes in many situations, so they might provide an alternative to formalize hierarchies of monad interfaces; they have been used for this purpose to a lesser extent in related work (see, e.g., the accompanying material of Mu & Chiang, 2020).

The application of Section 4 is a formalization of Mu & Chiang (2020) which comes with an Agda formalization as accompanying material. The latter contains axiomatized facts (Saito & Affeldt, 2022, Table 1), including the termination of quicksort, that are arguably orthogonal to the issue of quicksort derivation but that reveals practical issues directly related to monadic equational reasoning. We completed (and actually reworked from scratch) their formalization using MONAE. To complete Mu and Chiang's formalization, we needed in particular to formalize a thorough theory of nondeterministic permutations (Saito & Affeldt, 2022, 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 (Gibbons & Hinze, 2011, Sect. 4.4), using

nondeterministic selection and the function `unfoldM` (Mu, 2019a, Sect. 3.2), using nondeterministic insertion (Mu, 2019, Sect. 3), or using `liftM2` (Mu & Chiang, 2020, Sect. 3). The current version of MONAE has now a formalization of each.

6.2 About monadic equational reasoning

Gibbons and Hinze seem to be the first to synthesize monadic equational reasoning as an approach (Gibbons & Hinze, 2011; Gibbons, 2012; Abou-Saleh *et al.*, 2016). This viewpoint is also adopted by other authors (d. S. Oliveira *et al.*, 2012; Chen *et al.*, 2017; Mu, 2019a; Pauwels *et al.*, 2019; Mu & Chiang, 2020) that we have already mentioned in Section 1.

Applicative functor is an alternative approach to represent effectful computations. It has been formalized in Isabelle/HOL together with the tree relabeling example (Lochbihler & Schneider, 2016). This work focuses on the lifting of equations to allow for automation, while our approach is rather the one of MATHCOMP: the construction of a hierarchy of mathematical structures backed up by a rich library of definitions and lemmas to make the most out of the rewriting facilities of SSREFLECT.

6.3 About monad transformers

The idea to formalize monad transformers using packed classes was experimented by Affeldt & Nowak (2020) who formalize Jaskelioff's theory of modular monad transformers.

Huffman formalizes three monad transformers in the Isabelle/HOL proof assistant (Huffman, 2012). This experiment is part of a larger effort to overcome the limitations of Isabelle/HOL type classes to reason about Haskell programs that use (Haskell) type classes. In the dependent-type theory of COQ, we could formalize a much larger theory, even relying on extra features of COQ such as impredicativity and parametricity (see Affeldt & Nowak, 2020 for details).

Maillard proposes a meta-language to define monad transformers in the COQ proof assistant (Maillard, 2019, Chapter 4). It is an instance implementation of one element of a larger framework to verify programs with monadic effects using Dijkstra monads (Maillard *et al.*, 2019). 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. Like Jaskelioff's theory of modular monad transformers, the lifting of operations is one topic of this framework, but it does not go as far as the deep analysis of Jaskelioff (Jaskelioff, 2009a,b; Jaskelioff & Moggi, 2010).

There are also formalizations of monads and their morphisms that focus on the mathematical aspects, for example, UniMath (Voevodsky *et al.*, 2014). However, the link to the monad transformers of functional programming is not done.

6.4 About equational theories for ML with references

The typed store monad, particularly when it is extended with a `crun` operation to observe a result discarding the contents of the store, is very similar to the ST monad of

Haskell (Launchbury & Jones, 1994). While we could not find a description of the equational theory of the ST monad per se, there is a line of work on models for state using the same operations. This includes theories for local state, which allow partially or completely discarding the state, and for global state, which do not allow this. The state of the art for local state is work by Kammar *et al.* (2017), which builds on work by Staton (2010) to provide a theory for *full ground state*, where one can put reference cells in the store, but not functions, that is, basically the same restriction as us, but we are currently limited to global state. Note that they do not explicitly provide an equational theory, but their model supports nominal equational reasoning in the style of Staton (2010), that is, using judgments with a context that makes valid references explicit. This is different from our plain equational reasoning, which does not rely on such a context. For global state, Sterling *et al.* (2022) recently proposed a model for *higher-order state*, that is, allowing functions in the store, using synthetic guarded domain theory. They went on to develop a program logic based on it (Aagaard *et al.*, 2023), based on separation logic. It allows equational reasoning to some extent, but again it relies on a separation logic context when expressing equations. Moreover, equations contain extra tick operations, to account for guardedness.

7 Conclusions

In this article, we reported on an approach to formalize in the COQ proof assistant an extensible hierarchy of effects with their algebraic laws (Section 2) as well as their models (Section 3). The idea is to leverage on existing tools available for COQ: we use HIERARCHY-BUILDER to formalize interfaces of effects and models of monads, and we use SSREFLECT's rewriting tactics to reproduce proofs by monadic equational reasoning. We have illustrated our framework by two applications. We formalized the setting of a derivation of in-place quicksort by Mu & Chiang (2020) in Section 4. We have also designed an original set of equations so that OCaml programs become amenable to monadic equational reasoning in Section 5.

Besides the two applications we have explained in this article, we have also reproduced a number of standard examples of monadic equational reasoning:

- The tree relabeling example: This example originally motivated monadic equational reasoning (Gibbons & Hinze, 2011). It amounts to show that the labels of a binary tree are distinct when the latter has been relabeled with fresh labels using the `freshMonad`. See Monae (2018, file `example_relabeling.v`).
- The n -queens puzzle: This puzzle is used to illustrate the combination of state and nondeterminism. We have mechanized the relations between functional and stateful implementations (Gibbons & Hinze, 2011, Sections 6,7), as well as the derivation of a version of the algorithm using monadic hylo-fusion (Mu, 2019a, Sect. 5). See Monae (2018, file `example_nqueens.v`). Like the quicksort example, this example demonstrates the importance of commutativity lemmas (Section 4.2).
- The Monty Hall problem: We have mechanized the probability calculations for several variants of the Monty Hall problem (Gibbons & Hinze, 2011; Gibbons, 2012) using `probMonad` (Section 2.8), `altProbMonad` (Section 2.8.1), and `exceptProbMonad` (Monae, 2018, file `example_monty.v`) (Affeldt *et al.*, 2021, Sect. 2.3).

- Spark aggregation: Spark is a platform for distributed computing, in which the aggregation of data is therefore nondeterministic. Monadic equational reasoning can be used to sort out the conditions under which aggregation is actually deterministic (Mu, 2019, Sect. 4.2). We have mechanized this result (Monae, 2018, file `example_spark.v`), which is part of a larger specification (Chen *et al.*, 2017).
- The swap construction: This is an example of monad composition (Jones & Duponcheel, 1993). Strictly speaking, this is not monadic equational reasoning: formalization does not require a mechanism such as packed classes. Yet, our framework proved adequate because it allows to mix in a single equation different units and joins without explicit mention of which monad they belong to (Monae (2018, file `monad_composition.v`)).

We believe that our approach is successful in the sense that it helps find and fix errors in related work (as we have already explained in Section 1), that proof scripts closely match their pencil-and-paper counterparts (see, e.g., Figure 1 and Section 4.3.3), and that it provides support to investigate the design of new sets of equations (see Section 5). All these results have been aggregated as one coherent COQ library (Monae, 2018). It provides a rich hierarchy of interfaces and several examples of models from which users can draw inspiration to add a new monad, either starting from its interface or its model. Many reusable lemmas about specific monads (such as the one for commutativity of nondeterministic computations in Section 4.2) are available as library files so that new examples of monadic equational reasoning can be carried out easily.

Acknowledgments

The authors would like to thank Cyril Cohen and Enrico Tassi for their assistance with HIERARCHY-BUILDER, Jeremy Gibbons, Ohad Kammar, and Shin-Cheng Mu for their helpful input, and Yoshihiro Imai for his contribution to Infotheo (2018). This article is based on the revision of two papers presented at the MPC conference (Affeldt *et al.*, 2019; Saito & Affeldt, 2022) to which David Nowak and Ayumu Saito have participated, and on work that was presented at the Coq Workshop 2023 (Affeldt *et al.*, 2023). The authors acknowledge support of the JSPS KAKENHI Grants Number 22H00520 and Number 22K11902.

Conflicts of interest

The authors report no conflict of interest.

References

- Aagaard, F. L., Sterling, J. & Birkedal, L. (2023) A denotationally-based program logic for higher-order store. *Electron. Notes Theoret. Inf. Comput. Sci.* 3. Proceedings of the 39th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIX), Bloomington, IL, USA, June 19–23, 2023.
- Abou-Saleh, F., Cheung, K.-H. & Gibbons, J. (2016) Reasoning about probability and nondeterminism. In *POPL Workshop on Probabilistic Programming Semantics*.
- Aczel, P. (1977) The strength of Martin-Löf's intuitionistic type theory with one universe. In *Symposiums on mathematical logic in Oulu 1974 and in Helsinki 1975*, 1–32.

- Affeldt, R., Cohen, C., Kerjean, M., Mahboubi, A., Rouhling, D. & Sakaguchi, K. (2020) 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, June 29–July 6. Springer, pp. 3–20.
- Affeldt, R., Garrigue, J., Nowak, D. & Saikawa, T. (2021) A trustful monad for axiomatic reasoning with probability and nondeterminism. *J. Funct. Program.* **31**(e17), 1–38.
- Affeldt, R., Garrigue, J. & Saikawa, T. (2023) Environment-friendly monadic equational reasoning for OCaml. In The Coq Workshop 2023, Bialystok, Poland, July 31, 2023. Abstract available at: https://coq-workshop.gitlab.io/2023/abstracts/coq2023_monadic-reasoning.pdf.
- Affeldt, R. & Nowak, D. (2020) Extending equational monadic reasoning with monad transformers. In 26th International Conference on Types for Proofs and Programs (TYPES 2020), March 2–5, 2020, University of Turin, Italy. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 2:1–2:21.
- Affeldt, R., Nowak, D. & Saikawa, T. (2019) A hierarchy of monadic effects for program verification using equational reasoning. In 13th International Conference on Mathematics of Program Construction (MPC 2019), Porto, Portugal, October 7–9, 2019. Springer, pp. 226–254.
- Baez, J. C. & Shulman, M. (2010) Lectures on n-categories and cohomology. In *Towards Higher Categories*. New York, NY: Springer New York, pp. 1–68.
- Barthe, G., Grégoire, B. & Riba, C. (2008) Type-based termination with sized products. In 22nd International Workshop on Computer Science Logic (CSL 2008), Bertinoro, Italy, September 16–19, 2008. Springer, pp. 493–507.
- Benton, N., Hughes, J. & Moggi, E. (2000) Monads and effects. In Applied Semantics, International Summer School (APPSEM 2000), Caminha, Portugal, September 9–15, 2000, Advanced Lectures. Springer, pp. 42–122.
- Bertot, Y. & Castéran, P. (2004) *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer.
- Chan, J., Li, Y. & Bowman, W. J. (2023) Is sized typing for coq practical? *J. Funct. Program.* **33**(e1), 1–55.
- Chen, Y., Hong, C., Lengál, O., Mu, S., Sinha, N. & Wang, B. (2017) An executable sequential specification for spark aggregation. In 5th International Conference on Networked Systems (NETYS 2017), Marrakech, Morocco, May 17–19, 2017, pp. 421–438.
- Cheung, K.-H. (2017) *Distributive Interaction of Algebraic Effects*. PhD thesis. Merton College, University of Oxford.
- Christiansen, J., Dylus, S. & Bunkenburg, N. (2019) Verifying effectful Haskell programs in Coq. In 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019), Berlin, Germany, August 18–23, 2019. ACM, pp. 125–138.
- Claret, G. (2018) *Program in Coq. (Programmer en Coq)*. PhD thesis. Paris Diderot University, France.
- Cock, D. A. (2012) Verifying probabilistic correctness in Isabelle with pGCL. In 7th Conference on Systems Software Verification (SSV 2012), Sydney, Australia, 28–30 November 2012, pp. 167–178.
- Cohen, C., Sakaguchi, K. & Tassi, E. (2020) 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). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 34:1–34:21.
- CoqGen. (2021) A Coq generation backend for OCaml. Available at: <https://github.com/COCTI/ocaml/pull/3>. Authors: Jacques Garrigue, Takafumi Saikawa et al. Last modification: 2024.
- d. S. Oliveira, B. C., Schrijvers, T. & Cook, W. R. (2012) MRI: Modular reasoning about interference in incremental programming. *J. Funct. Program.* **22**(6), 797–852.
- Delaware, B., Keuchel, S., Schrijvers, T. & d. S. Oliveira, B. C. (2013) Modular monadic meta-theory. In ACM SIGPLAN International Conference on Functional Programming (ICFP 2013), Boston, MA, USA, September 25–27, 2013, pp. 319–330.

- Garillot, F., Gonthier, G., Mahboubi, A. & Rideau, L. (2009) Packaging mathematical structures. In 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009), Munich, Germany, August 17–20, 2009. Springer, pp. 327–342.
- Garrigue, J. & Saikawa, T. (2022) Validating OCaml soundness by translation into Coq. In 28th International Conference on Types for Proofs and Programs (TYPES 2022), Nantes, France, 20–25 June, 2022. Abstract available at: <https://www.math.nagoya-u.ac.jp/~garrigue/papers/types2022.pdf>. Implementation: A Coq generation backend for OCaml. Available at: <https://github.com/COCTI/ocaml/pull/3>. Authors: Jacques Garrigue, Takafumi Saikawa *et al.* Last modification: 2024.
- Gibbons, J. (2012) Unifying theories of programming with monads. In Revised Selected Papers of the 4th International Symposium on Unifying Theories of Programming (UTP 2012), Paris, France, August 27–28, 2012. Springer, pp. 23–67.
- Gibbons, J. & Hinze, R. (2011) Just do it: Simple monadic equational reasoning. In 16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011), Tokyo, Japan, September 19–21, 2011. ACM, pp. 2–14.
- Gonthier, G. & Mahboubi, A. (2010) An introduction to small scale reflection in Coq. *J. Formaliz. Reasoning* 3(2), 95–152.
- Hierarchy Builder. (2021) Hierarchy builder wiki—missingjoin. Available at: <https://github.com/math-comp/hierarchy-builder/wiki/MissingJoin>.
- Huffman, B. (2012) Formal verification of monad transformers. In ACM SIGPLAN International Conference on Functional Programming (ICFP 2012), Copenhagen, Denmark, September 9–15, 2012. ACM, pp. 15–16.
- Infotheo. (2018) Infotheo: A Coq formalization of information theory and linear error-correcting codes. Available at: <https://github.com/affeldt-aist/infotheo>. Authors: Reynald Affeldt, Manabu Hagiwara, Jonas Sénizergues, Jacques Garrigue, Kazuhiko Sakaguchi, Taku Asai, Takafumi Saikawa, and Naruomi Obata. Last stable release: 0.7.4 (2024).
- Jacobs, B. (2010) Convexity, duality and effects. In IFIP TCS. Springer, pp. 1–19.
- Jaskelioff, M. (2009a) Modular monad transformers. In Programming Languages and Systems, 18th European Symposium on Programming (ESOP 2009), York, UK, March 22–29, 2009. Springer, pp. 64–79.
- Jaskelioff, M. (2009b) *Lifting of Operations in Modular Monadic Semantics*. PhD thesis. University of Nottingham. Available at: <https://www.fceia.unr.edu.ar/~mauro/pubs/Thesis.pdf>.
- Jaskelioff, M. & Moggi, E. (2010) Monad transformers as monoid transformers. *Theor. Comput. Sci.* 411(51-52), 4441–4466.
- Jomaa, N., Nowak, D., Grimaud, G. & Hym, S. (2018) Formal proof of dynamic memory isolation based on MMU. *Sci. Comput. Program.* 162, 76–92.
- Jones, M. P. & Duponcheel, L. (1993) Composing monads. Technical Report YALEU/DCS/RR-1004. Yale University.
- Kammar, O., Levy, P. B., Moss, S. K. & Staton, S. (2017) A monad for full ground reference cells. In 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2017), Reykjavik, Iceland. IEEE Press.
- Launchbury, J. & Jones, S. L. P. (1994) Lazy functional state threads. In ACM SIGPLAN’94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20–24, 1994. ACM, pp. 24–35.
- Letan, T., Régis-Gianas, Y., Chifflier, P. & Hiet, G. (2018) Modular verification of programs with effects and effect handlers in coq. In 22nd International Symposium on Formal Methods (FM 2018), Oxford, UK, July 15–17, 2018. Springer, pp. 338–354.
- Liang, S., Hudak, P. & Jones, M. P. (1995) Monad transformers and modular interpreters. In 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1995), San Francisco, California, USA, January 23–25, 1995. ACM Press, pp. 333–343.
- Lochbihler, A. & Schneider, J. (2016) Equational reasoning with applicative functors. In 7th International Conference on Interactive Theorem Proving (ITP 2016), Nancy, France, August 22–25, 2016. Springer, pp. 252–273.

- Mahboubi, A. & Tassi, E. (2022) *Mathematical Components*. Zenodo. Available at: <https://doi.org/10.5281/zenodo.3999478>.
- Maillard, K. (2019) *Principes de la Vérification de Programmes à Effets Monadiques Arbitraires*. PhD thesis. Université PSL.
- Maillard, K., Ahman, D., Atkey, R., Martínez, G., Hritcu, C., Rivas, E. & Tanter, É. (2019) Dijkstra monads for all. *PACMPL* **3**(ICFP), 104:1–104:29.
- Martin-Dorel, E. & Tassi, E. (2019) SSReflect in Coq 8.10. In *The Coq Workshop 2019*, Portland State University, Portland, OR, USA, September 8, 2019. Presentation slides available at: <https://staff.aist.go.jp/reynald.affeldt/coq2019/coqws2019-martindorel-tassi-slides.pdf>.
- Martin-Löf, P. (1984) *Intuitionistic Type Theory*, Studies in Proof Theory, vol. 1. Bibliopolis.
- MathComp. (2024) The mathematical components repository. Available at: <https://github.com/math-comp/math-comp>. Version 2.2.0.
- McBride, C. & McKinna, J. (2004) The view from the left. *J. Funct. Program.* **14**(1), 69–111.
- Monae. (2018) Monae: Monadic effects and equational reasoning in Coq. Available at: <https://github.com/affeldt-aist/monae>. Authors: Reynald Affeldt, David Nowak, Takafumi Saikawa, Jacques Garrigue, Ayumu Saito, Celestine Sauvage, and Kazunari Tanaka. Last stable release: 0.7.1 (2024).
- Mu, S. (2019) Equational Reasoning for Non-determinism Monad: A Case Study of Spark Aggregation. Technical Report TR-IIS-19-002. Academia Sinica.
- Mu, S. (2019a) Calculating a Backtracking Algorithm: An Exercise in Monadic Program Derivation. Technical Report TR-IIS-19-003. Academia Sinica.
- Mu, S. & Chiang, T. (2020) Declarative pearl: Deriving monadic quicksort. In *15th International Symposium on Functional and Logic Programming (FLOPS 2020)*, Akita, Japan, September 14–16, 2020. Springer, pp. 124–138.
- Pauwels, K., Schrijvers, T. & Mu, S. (2019) Handling local state with global state. In *13th International Conference on Mathematics of Program Construction (MPC 2019)*, Porto, Portugal, October 7–9, 2019. Springer, pp. 18–44.
- Pfenning, F. & Elliott, C. (1988) Higher-order abstract syntax. In *the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI 1988)*, Atlanta, Georgia, USA, June 22–24, 1988. ACM, pp. 199–208.
- Saito, A. & Affeldt, R. (2022) Towards a practical library for monadic equational reasoning in Coq. In *14th International Conference on Mathematics of Program Construction (MPC 2022)*, Tbilisi, Georgia, September 26–28, 2022. Springer, pp. 151–177.
- Sakaguchi, K. (2020) Program extraction for mutable arrays. *Sci. Comput. Program.* **191**, 102372.
- Shan, C.-C. (2018) Equational reasoning for probabilistic programming. In *POPL 2018 TutorialFest*.
- Sozeau, M. (2009) Equations—a function definitions plugin. Available at: <https://mattam82.github.io/Coq-Equations/>. Last stable release: 1.3 (2022).
- Sozeau, M. & Mangin, C. (2019) Equations reloaded: High-level dependently-typed functional programming and proving in coq. *Proc. ACM Program. Lang.* **3**(ICFP), 86:1–86:29.
- Spector-Zabusky, A., Breitner, J., Rizkallah, C. & Weirich, S. (2018) Total Haskell is reasonable Coq. In *7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*, Los Angeles, CA, USA, January 8–9, 2018. ACM, pp. 14–27.
- Staton, S. (2010) Completeness for algebraic theories of local state. In *Foundations of Software Science and Computational Structures*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 48–63.
- Sterling, J., Gratzer, D. & Birkedal, L. (2022) Denotational semantics of general store and polymorphism. CoRR. abs/2210.02169.
- Swierstra, W. & Baanen, T. (2019) A predicate transformer semantics for effects (functional pearl). *Proc. ACM Program. Lang.* **3**(ICFP), 103:1–103:26.
- The Coq Development Team. (2024) *The Coq Proof Assistant Reference Manual*. Inria. Available at: <https://coq.inria.fr>. Version 8.19.1.
- Voevodsky, V., Ahrens, B., Grayson, D. et al.. (2014) UniMath—a computer-checked library of univalent mathematics. Available at: <https://github.com/UniMath/UniMath>.