Extending Equational Monadic Reasoning with Monad Transformers

Reynald Affeldt
National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan

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

Abstract

There is a recent interest for the verification of monadic programs using proof assistants. This line of research raises the question of the integration of monad transformers, a standard technique to combine monads. In this paper, we extend Monae, a Coq library for monadic equational reasoning, with monad transformers and we explain the benefits of this extension. Our starting point is the existing theory of modular monad transformers, which provides a uniform treatment of operations. Using this theory, we simplify the formalization of models in Monae and we propose an approach to support monadic equational reasoning in the presence of monad transformers. We also use Monae to revisit the lifting theorems of modular monad transformers by providing equational proofs and explaining how to patch a known bug using a non-standard use of Coq that combines impredicative polymorphism and parametricity.

2012 ACM Subject Classification Theory of computation → Logic and verification; Software and its engineering → Formal software verification

Keywords and phrases monads, monad transformers, Coq, impredicativity, parametricity

Digital Object Identifier 10.4230/LIPIcs.TYPES.2020.2

Related Version *Previous Version*: `https://arxiv.org/abs/2011.03463`

Supplementary Material *Software (Proof Scripts)*: `https://github.com/affeldt-aist/monae/` archived at `swh:1:dir:2d68878d365fe72744f8b085fa29df385567f6c9`

Funding We acknowledge the support of the JSPS KAKENHI Grant Number 18H03204.

Acknowledgements We thank all the participants of the JSPS-CNRS bilateral program "FoRmal tools for IoT sEcurity" (PRC2199) for fruitful discussions. We also thank Takafumi Saikawa for his comments. This work is based on joint work with Célestine Sauvage [29].

1 Introduction

There is a recent interest for the formal verification of monadic programs stemming from *monadic equational reasoning*: an approach to the verification of monadic programs that emphasizes equational reasoning [8, 9, 25–27]. In this approach, an effect is represented by an operator belonging to an interface together with equational laws. The interfaces all inherit from the type class of monads and the interfaces are organized in a hierarchy where they are extended and composed. There are several efforts to bring monadic equational reasoning to proof assistants [1, 2, 28].

In monadic equational reasoning, the user cannot rely on the *model* of the interfaces because the implementation of the corresponding monads is kept hidden. The construction of models is nevertheless important to avoid mistakes when adding equational laws [1]. This means that a formalization of monadic equational reasoning needs to provide tools to formalize models.

In this paper, we extend an existing formalization of monadic equational reasoning (called Monae [2]) with *monad transformers*. Monad transformers is a well-known approach to combine monads that is both modular and practical [20]. It is also commonly used to write Haskell programs. The interest in extending monadic equational reasoning with monad

transformers is therefore twofold: (1) it enriches the toolbox to build formal models of monad interfaces, and (2) it makes programs written with monad transformers amenable to equational reasoning.

In fact, the interest for a formal theory of monad transformers goes beyond its application to monadic equational reasoning. Past research advances about monad transformers could have benefited from formalization. For example, a decade ago, Jaskelioff identified a lack of uniformity in the definitions of the liftings of operations through monad transformers [15]. He proposed *modular monad transformers* which come with a uniform definition of lifting for operations that qualify as *sigma-operations* or their sub-class of *algebraic operations*. Unfortunately, the original proposal in terms of System Fω was soon ruled out as faulty [17, Sect. 6] [14, p. 7] and its fix gave rise to a more involved presentation in terms of (non-trivial) category theory [17]. More recently, this is the comparison between monad transformers and algebraic effects that attracts attention, and it connects back to reasoning using equational laws (e.g., [30, Sect. 7]). This is why in this paper, not only do we provide examples of monad transformers and applications of monadic equational reasoning, but also formalize a theory of monad transformers.

Contributions. In this paper, we propose a formalization in the Coq proof assistant [32] of monad transformers. This formalization comes as an extension of Monae, an existing library that provides a hierarchy of monad interfaces [2]. The benefits of this extension are as follows.

- The addition of sigma-operations and of monad transformers to Monae improves the implementation of models of monads. These models are often well-known and it is tempting to define them in an ad hoc way. Sigma-operations help us discipline proof scripts and naming, which are important aspects of proof-engineering.
- We illustrate with an example how to extend Monae to verify a program written with a monad transformer. Verification is performed by equational reasoning using equational laws from a monad interface whose model is built using a monad transformer.
- We use our formalization of monad transformers to formalize the theory of lifting of modular monad transformers. Thanks to Monae, the main theorems of modular monad transformers can be given short formal proofs in terms of equational reasoning.

Regarding the theory of lifting of modular monad transformers, our theory fixes the original presentation [15]. This fix consists in a non-standard use of Coq combining impredicativity and parametricity (as implemented by ParamCoq [18]) that allows for an encoding using the language of the proof assistant and thus avoids the hassle of going through a technical formalization of category theory (which is how Jaskelioff fixed his original proposal). It must be said that this was not possible at the time of the original paper on modular monad transformers because parametric models of dependent type theory were not known [4] (but were "expected" [16]). We are therefore in the situation where formalization using a proof assistant allowed for a fruitful revisit of pencil-and-paper proofs.

Regarding the benefit of extending Monae with sigma-operations and monad transformers, we would like to stress that this is also a step towards more modularity in our formalization of monadic equational reasoning. Indeed, one important issue that we have been facing is the quality of our proof scripts. Proof scripts that reproduce monadic equational reasoning must be as concise as they are on paper. Proof scripts that build models (and prove lemmas) should be maintainable (to be improved or fixed easily in case of changes in the hypotheses) and understandable (this means having a good balance between the length of the proof script and its readability). This manifests as mundane but important tasks such as factorization

of proof scripts, generalization of lemmas, abstraction of data structures, etc. From the viewpoint of proof-engineering, striving for modularity is always a good investment because it helps in breaking the formalization task into well-identified, loosely-coupled pieces.

Outline. In Sect. 2, we recall the main constructs of Monae. In Sect. 3, we formalize the basics of modular monad transformers: sigma-operations, monad transformers, and their variants (algebraic operations and functorial monad transformers). Section 4 is our first application: we show with an example how to extend Monae to verify a program written using monad transformers. In Sect. 5, we use our formalization of monad transformers to prove a first theorem about modular monad transformers (namely, the lifting of algebraic operations) using equational reasoning. In Sect. 6, we formalize (and fix) the main theorem of modular monad transformers (namely, the lifting of sigma-operations that are not necessarily algebraic along functorial monad transformers). We review related work in Sect. 7 and conclude in Sect. 8.

2 Overview of the Monae Library

Monae [2] is a formal library implemented in the Coq proof assistant [32] to support monadic equational reasoning [9]. It takes advantage of the rewriting capabilities of the tactic language called SSReflect [10] to achieve formal proofs by rewriting that are very close to their pencil-and-paper counterparts. Monae provides a hierarchy of monad interfaces formalized using the methodology of *packed classes* [6]. Effects are declared as operations in interfaces together with equational laws, and some effects extend others by (simple or multiple) inheritance. This modularity is important to achieve natural support for monadic equational reasoning.

Let us briefly explain some types and notations provided by Monae that we will use in the rest of this paper. Monae provides basic category-theoretic definitions such as functors, natural transformations, and monads. By default, they are specialized to `UU0`, the lowest universe in the hierarchy of Coq types, understood as a category[1]. The type of functors is `functor`. The application of a functor `F` to a function `f` is denoted by `F # f`. The composition of functors is denoted by the infix notation `\O`. The identity functor is denoted by `FId`. Natural transformations from the functor `F` to the functor `G` are denoted by `F ~> G`. Natural transformations are formalized by their components (represented by the type `forall A, F A -> G A`, denoted by `F ~~> G`) together with the proof that they are natural, i.e., the proof that they satisfy the following predicate:

```
Definition naturality (M N : functor) (m : M ~~> N) :=
  forall (A B : UU0) (h : A -> B), (N # h) \o m A = m B \o (M # h).
```

(The infix notation `\o` is for function composition.) Vertical composition of natural transformations is denoted by the infix notation `\v`. The application of a functor `F` to a natural transformation `n` is denoted by `F ## n`.

The type of monads is `monad`, which inherits from the type `functor`. Let `M` be of type `monad`. Then `Ret` is a natural transformation `FId ~> M` and `Join` is a natural transformation `M \O M ~> M`. Using `Ret` and `Join`, we define the standard bind operator with the notation `>>=`.

In this paper, we show Coq proof scripts verbatim when it is reasonable to do so. When we write mathematical formulas, we keep the same typewriter font, but, for clarity and to ease reading, we make explicit some information that would otherwise by implicitly inferred

[1] Monae also provides a more generic setting [24, file `category.v`] but we do not use it in this paper.

by Coq. For example, one simply writes `Ret` or `Join` in proof scripts written using Monae because it has been implemented in such a way that Coq infers from the context which monad they refer to and which type they apply to. In mathematical formulas, we sometimes make the monad explicit by writing it as a superscript of `Ret` or `Join`, and we sometimes write the argument of a function application as a subscript. This leads to terms such as $\texttt{Ret}_{\texttt{A}}^{\texttt{M}}$: the unit of the monad `M` applied to some type `A`.

See the online development for technical details (in particular, [24, file `hierarchy.v`]).

3 Sigma-operations and Monad Transformers in Monae

The first step is to formalize sigma-operations (Sect. 3.1) and monad transformers (Sect. 3.3). We illustrate sigma-operations with the example of the model of the state monad (Sect. 3.1.1) and its get operation (Sect. 3.1.2).

3.1 Extending Monae with Sigma-operations

Given a functor `E`, an *E-operation* for a monad `M` (sigma-operation for short) is a natural transformation from `E \O M` to `M`. The fact that sigma-operations are defined in terms of natural transformations is helpful to build models because it involves structured objects (functors and natural transformations) already instrumented with lemmas. In other words, we consider sigma-operations as a disciplined way to formalize effects. For illustration, we explain how the get operation of the state monad is formalized.

3.1.1 Example: Model of the State Monad

First we define a model `State.t` for the state monad (without get and put for the time being). We assume a type `S` (line 2) and define the action on objects `acto` (line 3), abbreviated as `M` (line 4). We define the action on morphisms `map` (line 5) and prove the functor laws (omitted here, see [24, file `monad_model.v`] for details). This provides us with a functor `functor` (line 8, `Functor.Pack` and `Functor.Mixin` are constructors from Monae and are named after the packed classes methodology [6]). We define the unit of the monad by first providing its components `ret_component` (line 9), and prove naturality (line 10, proof script omitted). We then package this proof to form a genuine natural transformation at line 12 (`Natural.Pack` and `Natural.Mixin` are constructors from Monae). We furthermore define `bind` (line 14), prove the properties of the unit and bind (omitted). Finally, we call the function `Monad_of_ret_bind` from Monae to build the monad (line 16):

```
1  (* in Module State *)
2  Variable S : UU0.
3  Definition acto := fun A => S -> A * S.
4  Local Notation M := acto.
5  Definition map A B (f : A -> B) (m : M A) : M B :=
6   fun (s : S) => let (x1, x2) := m s in (f x1, x2).
7  (* functor laws map_id and map_comp omitted *)
8  Definition functor := Functor.Pack (Functor.Mixin map_id map_comp).
9  Definition ret_component : FId ~~> M := fun A a => fun s => (a, s).
10 Lemma naturality_ret : naturality FId functor ret_component.
11 (* proof script of naturality omitted *)
12 Definition ret : FId ~> functor :=
13   Natural.Pack (Natural.Mixin naturality_ret).
14 Definition bind := fun A B (m : M A) (f : A -> M B) => uncurry f \o m.
15 (* proofs of neutrality of ret and of associativity of bind omitted *)
16 Definition t := Monad_of_ret_bind left_neutral right_neutral associative.
```

3.1.2 Example: The Get Operation as a Sigma-operation

By definition, for each sigma-operation we need a functor. The functor corresponding to the get operation is defined below as `Get.func` (line 5): `acto` is the action on the objects, `actm` is the action on the morphisms (the prefix `@` disables implicit arguments in Coq):

```
1  (* in Module Get *)
2  Variable S : UU0.
3  Definition acto X := S -> X.
4  Definition actm (X Y : UU0) (f : X -> Y) (t : acto X) : acto Y := f \o t.
5  Program Definition func := Functor.Pack (@Functor.Mixin _ actm _ _).
6  (*  proofs of the functors law omitted *)
```

We then define the sigma-operation itself (`StateOps.get_op` at line 7), which is a natural transformation from `Get.func S \O M` to `M`, where `M` is the state monad `State.t` built in Sect. 3.1.1. Note that this get operation ($\lambda s.\, k\, s\, s$, line 4) is *not* the usual operation [15, Example 13].

```
1  (* in Module StateOps *)
2  Variable S : UU0.
3  Local Notation M := (State.t S).
4  Definition get A (k : S -> M A) : M A := fun s => k s s.
5  Lemma naturality_get : naturality (Get.func S \O M) M get.
6  (* proof script of naturality omitted *)
7  Definition get_op : (Get.func S).-operation M :=
8    Natural.Pack (Natural.Mixin naturality_get).
```

3.1.3 Example: Model of the Interface of the State Monad

Monae originally comes with an interface `stateMonad` for the state monad (*with* the get and put operations). It implements the interface as presented by Gibbons and Hinze [9, Sect. 6]; it therefore expects the operations to be the usual ones. We show how to instantiate it using the definition of sigma-operations. First, we need to define the usual get from `StateOps.get_op` (line 4 below):

```
1  (* in Module ModelState *)
2  Variable S : UU0.
3  Local Notation M := (ModelMonad.State.t S).
4  Definition get : M S := StateOps.get_op _ Ret.
```

We do the same for the put operation (omitted). We then build the model of interface of the state monad (with its operations) using the appropriate constructors from Monae:

```
Program Definition state : stateMonad S := MonadState.Pack (MonadState.Class
  (@MonadState.Mixin _ _ get put _ _ _ _)).
(* proofs of the laws of get and put automatically discharged *)
```

Similarly, using sigma-operations, we have formalized the operations of the list, the output, the state, the environment, and the continuation monads, which are the monads discussed along with modular monad transformers [15, Fig. 1] (see [24, file `monad_model.v`] for their formalization).

3.2 The Sub-class of Algebraic Operations

An `E`-operation `op` for `M` is *algebraic* [15, Def. 15] when it satisfies the predicate `algebraicity` defined as follows in Coq (observe the position of the continuation `>>= f`):

```
forall A B (f : A -> M B) (t : E (M A)),
  op A t >>= f = op B ((E # (fun m => m >>= f)) t).
```

Algebraic operations are worth distinguishing because they lend themselves more easily to lifting, and this result can be used to define lifting for the whole class of sigma-operations (this is the purpose of Sections 5 and 6). We can check using Coq that, as expected, all the operations discussed along with modular monad transformers [15, Fig. 1] are algebraic except for flush, local, and handle[2].

Example: the Get operation is Algebraic

For example, the get operation of the state monad is algebraic:

```
Lemma algebraic_get S : algebraicity (@StateOps.get_op S). Proof. by []. Qed.
```

In the Coq formalization, we furthermore provide the type `E.-aoperation M` (note the prefix "`a`") of an `E.-operation M` that is actually algebraic. For example, here is how we define the algebraic version of the get operation:

```
Definition get_aop S : (StateOps.Get.func S).-aoperation (ModelMonad.State.t S) :=
  AOperation.Pack (AOperation.Class (AOperation.Mixin (@algebraic_get S))).
```

3.3 Extending Monae with Monad Transformers

Given two monads `M` and `N`, a *monad morphism* `e` is a function of type `M ~~> N` such that for all types `A`, `B` the following laws hold:

```
e A \o Ret = Ret.                          (* MonadMLaws.ret *)

forall (m : M A) (f : A -> M B),           (* MonadMLaws.bind *)
  e B (m >>= f) = e A m >>= (e B \o f).
```

In Coq, we define the type of monad morphisms `monadM` that implement the two laws above. Monad morphisms are also natural transformations (this can be proved easily using the laws of monad morphisms). We therefore equip monad morphisms `e` with a canonical structure of natural transformation. Since it is made canonical, Coq is able to infer it in proof scripts but we need to make it explicit in statements; we provide the notation `monadM_nt e` for that purpose.

A *monad transformer* `t` is a function of type `monad -> monad` with an operator `Lift` such that for any monad `M`, `Lift t M` is a monad morphism from `M` to `t M`. Let `monadT` be the type of monad transformers in Monae. We reproduced all the examples of modular monad transformers (state, exception, environment, output, continuation monad transformers, resp. `stateT`, `exceptT`, `envT`, `outputT`, and `contT` in [24, file `monad_transformer.v`]).

Example: The Exception Monad Transformer

Let us assume given some type `Z : UU0` for exceptions and some monad `M`. First, we define the action on objects of the monad transformed by the exception monad transformer (the type `Z + X` represents the sum type of the types `Z` and `X`):

---

[2] In fact, we had to fix the output operation of the output monad. Indeed, it is defined as follows in [15, Example 32]: $\mathsf{output}((w,m) : W \times \mathsf{O}X) : \mathsf{O}X \triangleq \mathsf{let}\,(x,w') = m\,\mathsf{in}\,(x, \mathsf{append}(w',w))$. We changed $\mathsf{append}(w',w)$ to $\mathsf{append}(w,w')$ to be able to prove algebraicity.

```
Definition MX := fun X : UU0 => M (Z + X).
```

We also define the unit and the bind operator of the transformed monad (the constructors `inl`/`inr` inject a type into the left/right of a sum type):

```
Definition retX X x : MX X := Ret (inr x).
Definition bindX X Y (t : MX X) (f : X -> MX Y) : MX Y :=
  t >>= fun c => match c with inl z => Ret (inl z) | inr x => f x end.
```

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`. This is similar to the construction of the state monad we saw in Sect. 3.1. We start by defining the underlying functor `MX_map`, prove the two functor laws (let us call `MX_map_i` and `MX_map_o` these proofs), and package them as a functor:

```
Definition MX_functor := Functor.Pack (Functor.Mixin MX_map_i MX_map_o).
```

We then provide the natural transformation `retX_natural` corresponding to `retX` and call the Monae constructor `Monad_of_ret_bind` (like we did in Sect. 3.1):

```
Program Definition exceptTmonad : monad :=
  @Monad_of_ret_bind MX_functor retX_natural bindX _ _ _.
(* proofs of monad laws omitted *)
```

Then we define the lift operation as a function that given a computation `m` in the monad `M X` returns a computation in the monad `exceptTmonad X`:

```
Definition liftX X (m : M X) : exceptTmonad X := m >>= (@RET exceptTmonad _).
```

(The function `RET` is a variant of `Ret` better suited for type inference here.) We can finally package the definition of `liftX` to form a monad morphism:

```
Program Definition exceptTmonadM : monadM M exceptTmonad :=
  monadM.Pack (@monadM.Mixin _ _ liftX _ _).
(* proof of monad morphism laws omitted *)
```

The exception monad transformer merely packages the monad morphism we have just defined to give it the type `monadT`:

```
Definition exceptT Z := MonadT.Pack (MonadT.Mixin (exceptTmonadM Z)).
```

One might wonder what is the relation between the monads that can be built with these monad transformers and the monads already present in Monae. For example, in Sect. 3.1, we already mentioned the `stateMonad` interface and we built a model for it (namely, `ModelState.state`). On the other hand, we can now, say, build a model for the identity monad (let us call it `identity`) and build a model for that state monad as `stateT S identity` (we have not provided the details of `stateT`, see [2]). We can actually prove in Coq that `stateT S identity` and `State.t` are *equal*[3], so that no confusion has been introduced by extending Monae with monad transformers.

[3] [24, Section `instantiations_with_the_identity_monad`, file `monad_model.v`]

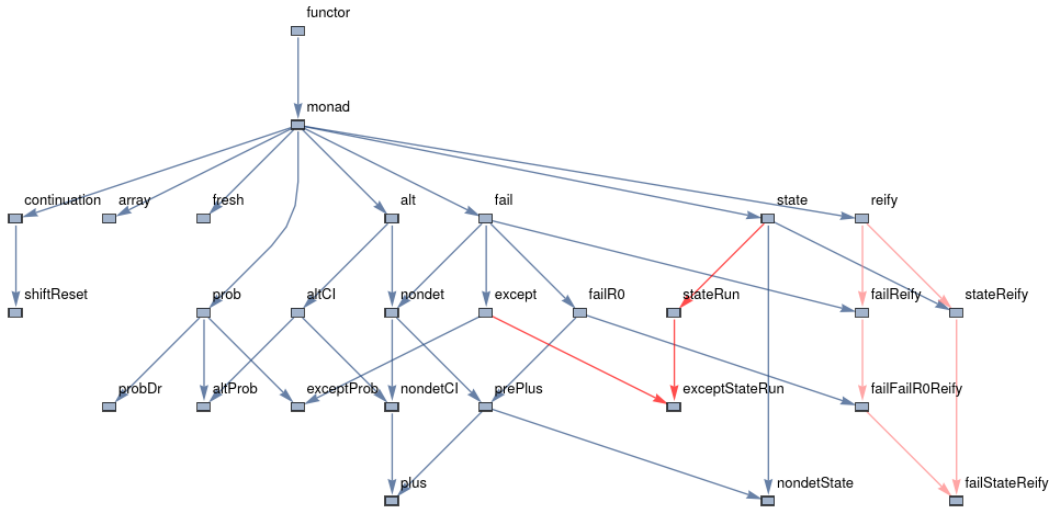

Figure 1 Hierarchy of Monad Interfaces Provided by MONAE.

3.4 Functorial Monad Transformers

A *functorial monad transformer* [15, Def. 20] is a monad transformer `t` with a function `h` (hereafter denoted by `Hmap t`) of type

```
forall (M N : monad), (M ~> N) -> (t M ~> t N)
```

such that (1) `h` preserves monad morphisms (the laws `MonadMLaws.ret` and `MonadMLaws.bind` seen in Sect. 3.3), (2) `h` preserves identities and composition of natural transformations, and (3) `Lift t` is natural, i.e.,

```
forall (M N : monad) (n : M ~> N) X, h M N n X \o Lift t M X = Lift t N X \o n X.
```

Note that we cannot define the naturality of `Lift t` using the predicate `naturality` we saw in Sect. 2 because it is restricted to endofunctors on `UU0`. Also note that Jaskelioff distinguishes monad transformers from functorial monad transformers while Maillard defines monad transformers as functorial by default [21, Def. 4.1.1].

4 Application 1: Monadic Equational Reasoning in the Presence of Monad Transformers

We apply our formalization of monad transformers to the verification of a recursive program combining the effects of state and exception. We argue that this program is similar in style to what an Haskell programmer would typically write with monad transformers. Despite this programming style and the effects, the correctness proof is by equational reasoning.

4.1 Extending the Hierarchy

The first thing to do is to extend the hierarchy of interfaces with `stateRunMonad` and `exceptStateRunMonad` (Fig. 1).

The interface `stateRunMonad` is a parameterized interface that extends `stateMonad` with the primitive `RunStateT` and its equations. Concretely, let `N` be a monad and `S` be the type of states. When `m` is a computation in the monad `stateRunMonad S N`, `RunStateT m s` runs `m` in a state `s` and returns a computation in the monad `N`. There is one equation for each combination of `RunStateT` with operations below in the hierarchy:

```
RunStateT (Ret a)   s = Ret (a, s)
RunStateT (m >>= f) s = RunStateT m s >>= fun x => RunStateT (f x.1) x.2
RunStateT Get       s = Ret (s, s)
RunStateT (Put s')  s = Ret (tt, s')
```

This is the methodology of packed classes that allows for the overloading of the notations `Ret` and `>>=` here. The notation `.1` (resp. `.2`) is for the first (resp. second) projection of a pair. The unique value of type `unit` is `tt`. The operations `Get` and `Put` are the standard operations of the state monad. Intuitively, given a monad `M` that inherits from the state monad, `Get` is a computation of type `M S` that returns the state and `Put` has type `S -> M unit` and updates the state (see Sect. 3 for a model of these operations).

The interface `exceptStateRunMonad` is the combination of the operations and equations of `stateRunMonad` and `exceptMonad` [9, Sect. 5] [24, file `hierarchy.v`] plus two additional equations on the combination of `RunStateT` with the operations of `exceptMonad`. Recall that the operations of the exception monad are the computations `Fail` of type `M A` and `Catch` of type `M A -> M A -> M A` for some type `A` (which happens to be the type of the state in this example); intuitively, `Fail` raises an exception while `Catch` handles it.

```
RunStateT Fail          s = Fail
RunStateT (Catch m1 m2) s = Catch (RunStateT m1 s) (RunStateT m2 s)
```

Using our formalization of monad transformers presented in this paper, it is then easy to build a model that validates those equations, whereas in previous work we had to build a model from scratch each time we were introducing a new combination of effects.

4.2 Example: The Fast Product

Now let us write a program and reason on it equationally. First, we write a recursive function that traverses a list of natural numbers to compute their product, but fails in case a 0 is met. Intermediate results are stored in the state:

```
Variables (N : exceptMonad) (M : exceptStateRunMonad nat N).
Fixpoint fastProductRec l : M unit :=
  match l with
  | [::] => Ret tt
  | 0 :: _ => Fail
  | n.+1 :: l' => Get >>= fun m => Put (m * n.+1) >> fastProductRec l'
  end.
```

Then, the main function will catch an eventual failure. If there is a failure, then the result is 0, else the result is the value stored in the state:

```
Variables (N : exceptMonad) (M : exceptStateRunMonad nat N).
Definition fastProduct l : M _ :=
  Catch (Put 1 >> fastProductRec l >> Get) (Ret 0 : M _).
```

To implement this algorithm in Haskell, we would use the state monad transformer applied to the exception monad. It would then be necessary to prefix each primitive of the exception monad with a lifting operation (`lift` or `mapStateT2` in Haskell). Here, we avoid this by using the hierarchy of interfaces, and the use of monad transformers is restricted to the construction of models for the interfaces.

The correctness states that the result of the fast product is always the same as a purely functional version:

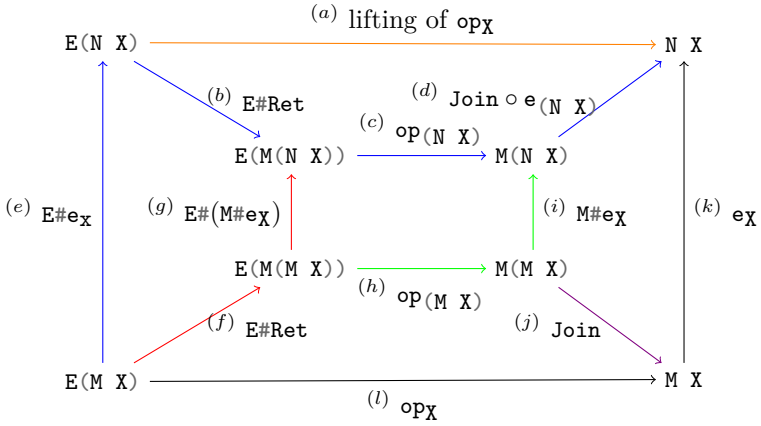

Figure 2 Proof of Uniform Algebraic Lifting (Theorem 1).

```
Lemma fastProductCorrect l n : evalStateT (fastProduct l) n = Ret (product l).
```

where `evalStateT m s` is defined as `RunStateT m s >>= fun x => Ret x.1`. This proposition is proved easily with a 10 lines proof script that consists of an induction on `l`, rewriting with the equations in `exceptStateRunMonad` and application of standard arithmetic (see Appendix A.1).

Note that in this section we are dealing with the state monad transformer applied to the exception monad, and that the last equation in Sect. 4.1 specifies that the state is "backtracked", i.e., if the state is modified in `m1` before an exception occurs, then this change is forgotten before `m2` is executed. This is usual in Haskell. The alternative semantics without backtracking would be closer to, say, OCaml, where the state is not be backtracked in case of an exception. Our program would behave the same way because it happens that the exception handler ignores the state. However, we would need to devise new equations to deal with `Fail` and `Catch`.

5 Application 2: Formalization of the Lifting of an Algebraic Operation

This section is an application of Monae extended with the formalization of sigma-operations and of monad transformers of Sect. 3. We prove using equational reasoning a theorem about the lifting of algebraic operations along monad morphisms. This corresponds to the theorem that concludes the first of part of the original paper on modular monad transformers [15, Sect.2–4].

In the following `M` and `N` are two monads. Given an `E`-operation `op` for `M` and a monad morphism `e` from `M` to `N`, a *lifting* of `op` (to `N`) along `e` is an `E`-operation `op'` for `N` such that for all `X`:

$$\mathtt{e}_X \circ \mathtt{op}_X = \mathtt{op'}_X \circ (\mathtt{E\#e}_X).$$

▶ **Theorem 1** (Uniform Algebraic Lifting [15, Thm. 19]). *Given an algebraic `E`-operation `op` for `M` and a monad morphism `e` from `M` to `N`, let `op'` be*

$$X \mapsto \mathit{Join}^N_X \circ e_{(N\ X)} \circ \mathit{op}_{(N\ X)} \circ (\mathit{E\#Ret}^M_{(N\ X)}).$$

Then `op'` is an algebraic `E`-operation for `N` and a lifting of `op` along `e`.

Proof. The proof that `op'` is a lifting is depicted by the diagram of Fig. 2.

The first step is to show that the path (a) (→) and the path (b)-(c)-(d) (→) are equal, which is by definition of a lifting. The resulting goal is rendered in Coq as follows (for any `Y`):

```
e X (op X Y) = Join (e (N X) (op (N X) ((E # Ret) ((E # e X) Y))))
```

The second step of the proof is to show that the path (e)-(b) (→) and the path (f)-(g) (→) are equal, which is achieved by appealing to the functor laws and the naturality of `Ret`. More precisely, to prove

```
(E # Ret) ((E # e X) Y) = (E # (M # e X)) ((E # Ret) Y),
```

it suffices to execute the following sequence of rewritings:

```
rewrite -[in LHS]compE -functor_o. (* functor composition law in the lhs *)
rewrite -[in RHS]compE -functor_o. (* functor composition law in the rhs *)
(* the goal is now: (E # (Ret \o e X)) Y = (E # (M # e X \o Ret)) Y *)
rewrite (natural RET).             (* naturality of ret *)
(* the goal is now: (E # (Ret \o e X)) Y = (E # (Ret \o FId # e X)) Y *)
by rewrite FIdf.                   (* property of the identity functor *)
```

The next step is to show that the path (g)-(c) and the path (h)-(i) (→) are equal; this is by naturality of `op`.

The next step is to show that the paths (i)-(d) and (j)-(k) are equal, which is by the bind law of monad morphisms and naturality of monad morphisms.

The last step (equality of the paths (f)-(h)-(j) and (l)) amounts to proving:

```
op X Y = Join (op (M X) ((E # Ret) Y)).
```

This step depends of an intermediate lemma [15, Prop. 17]. Let us explain it because it introduces functions and we will use one of them again later in this paper. Given a natural transformation `n : E ~> M`, `psi` is an `E`-operation for `M` defined by the function $\texttt{X} \mapsto \texttt{Join}_{\texttt{X}} \circ \texttt{n}$. Given an `E`-operation for `M`, `phi` is a natural transformation `E ~> M` defined by the function $\texttt{X} \mapsto \texttt{op}_{\texttt{X}} \circ (\texttt{E\#Ret})$. It turns out that `psi` is algebraic and that `psi` cancels `phi` for algebraic operations (proofs omitted here, see [24]), which proves the last goal.

The second part of the proof is to prove that `op'` is algebraic. This is a direct consequence of the fact that `psi` is algebraic.

It should be noted that, even though the statement of the theorem defines the lifting as the composition of the functions `Join`, `e`, etc., it is actually much more practical from the view point of formal proof to define it as `psi (monadM_nt e \v phi op)`, i.e., the application of the function `psi` to the vertical composition of `e` and `phi op`, because this object (let us call it `alifting`) is endowed with the properties of algebraic operations, whose immediate availability facilitates the formal proof. ◀

The reader can observe in Appendix A.2 that the complete proof script for Theorem 1 essentially amounts to a small number of rewritings, as has been partially illustrated in the proof just above.

Example: Lifting the get Operation along the Exception Monad Transformer

Let us assume the availability of a type `S` for states and of a type `Z` for exceptions. We consider `M` to be the state monad. To define the lifting of the get operation of `M` (more precisely its algebraic version seen in Sect. 3.2) along `exceptT` (Sect. 3.3) it suffices to call the `alifting` function with the right arguments:

```
Let M S : monad := ModelState.state S.

Definition aLGet {Z S} : (StateOps.Get.func S).-aoperation (exceptT Z (M S)) :=
  alifting (get_aop S) (Lift (exceptT Z) (M S)).
```

By the typing, we see that the result `aLGet` is also an algebraic operation.

For example, we can check that the resulting sigma-operation is indeed the get operation of the transformed monad:

```
Goal forall Z (S : UU0) X (k : S -> exceptT Z (M S) X),
  aLGet _ k = StateOps.get_op _ k. by [].
```

6 Application 3: Formalization of the Lifting of Sigma-Operations

This section is an application of our formalization of sigma-operations and (functorial) monad transformers of Sect. 3 and also of Theorem 2. Using Monae, we give an equational proof for a theorem that generalizes the lifting of Sect. 5 which was restricted to algebraic operations. This corresponds to the second part of the original paper on modular monad transformers [15, Sect. 5].

This application requires us to use a non-standard setting of Coq. Section 6.1 introduces a monad transformer whose formalization requires impredicativity. Section 6.2 focuses on the main technical difficulty that we identified when going from the pencil-and-paper proofs to a formalization using Coq: an innocuous-looking proof that actually calls for an argument based on parametricity. We conclude this section with the formal statement of [15, Thm. 27] and its formal proof (Sect. 6.3).

6.1 Impredicativity Setting for the Codensity Monad Transformer

To implement the lifting of an operation along a functorial monad transformer, Jaskelioff introduces a monad transformer `codensityT` related to the construction of the codensity monad for an endofunctor [15, Def. 23]. Its formalization requires impredicativity and if nothing is done, the standard setting of Coq would lead to *universe inconsistencies*.

Let us give a bit of background on impredicativity with Coq. The type theory of Coq is constrained by a hierarchy of universes `Set`, $\texttt{Type}_1$, $\texttt{Type}_2$, etc. The Coq language only provides the keywords `Set` and `Type`, the Coq system figures out the right indices for `Type`s. Universes are not impredicative by default; yet, Coq has an option (`-impredicative-set`) that changes the logical theory by declaring the universe `Set` as impredicative. This option is useful in Coq to formalize System $F/F\omega$, their impredicative encodings of data types, and for extraction of programs in CPS style. It is known to be inconsistent with some standard axioms of classical mathematics [7, 31] but we do not rely on them here[4]. To keep a firm grip on the universes involved, we fix a few universes at the beginning of the formal development [24, file `ihierarchy.v`]:

```
Definition UU2 : Type := Type.
Definition UU1 : UU2 := Type.
Definition UU0 : UU1 := Set.
```

and only use them instead of `Set` or `Type` (so far we have been using `UU0` but it is really another name for the native `Set` universe).

[4] More precisely, the development we discuss in this paper [24, directory `impredicative_set`] uses together with impredicative `Set` only the standard axioms of functional extensionality and proof irrelevance, which are compatible.

Now that we have set Coq appropriately, we define the codensity monad transformer. Given a monad `M`, a computation of a value of type `A` in the monad `codensityT M` has type `forall (B : UU0), (A -> M B) -> M B` of type `UU0`: here, impredicativity comes into play. We abbreviate this type expression as `MK M A` in the following. We do not detail the formalization of `codensityT` because it follows the model of the exception monad transformer that we explained in Sect. 3.3. Let us just display its main ingredients, i.e., the unit, bind, and lift operations [15, Def. 23]:

```
Definition retK (A : UU0) (a : A) : MK M A :=
  fun (B : UU0) (k : A -> M B) => k a.
Definition bindK (A B : UU0) (m : MK M A) f : MK M B :=
  fun (C : UU0) (k : B -> M C) => m C (fun a : A => (f a) C k).
(* definition of codensityTmonadM omitted *)
Definition liftK (A : UU0) (m : M A) : codensityTmonadM A :=
  fun (B : UU0) (k : A -> M B) => m >>= k.
```

We can check in Coq that they indeed give rise to a monad transformer in the sense of Sect. 3.3, so that `codensityT` does have the type `monadT` (Sect. 3.3) of monad transformers.

6.2 Parametricity to Prove Naturality

The monad transformer `codensityT` is needed to state the theorem about the lifting of sigma-operations and in particular to define a natural transformation called `from` [15, Prop. 26]. Formally, we can define `from`'s components as follows (`M` is a monad):

```
Definition from_component : codensityT M ~~> M :=
  fun (A : UU0) (c : codensityT M A) => c A Ret.
```

At first sight, the naturality of `from_component` seems obvious and indeed no proof is given in the original paper on modular monad transformers (see the first of the two statements of [15, Prop. 26]). It is however a bit more subtle than it appears and, as a matter of fact, it is shown in a later paper that this claim is wrong: $\texttt{from}_{\texttt{M}}$ cannot be a natural transformation in the setting of $F\omega$ [17, p. 4452]. We explain how we save the day in Coq by relying on parametricity.

We state the naturality of $\texttt{from}_{\texttt{M}}$ as `naturality (codensityT M) M from_component`. This goal reduces[5] to:

```
forall (m : codensityT M A) (h : A -> B), (M # h \o m A) Ret = m B (M # h \o Ret).
```

This last goal is an instance of a more general statement (recall from Sect. 6.1 that `MK M` is the action on the objects of the monad `codensityT M`):

```
forall (M : monad) (A : UU0) (m : MK M A) (A1 B : UU0) (h : A1 -> B),
  M # h \o m A1 = m B \o (fun f : A -> M A1 => (M # h) \o f).
```

This is actually a special case of naturality as one can observe by rewriting the type of `m` with the appropriate functors: `exponential_F A \O M` and `M`, where `exponential_F` is the functor whose action on objects is `forall X : UU0, A -> X`:

```
forall (M : monad) (A : UU0) (m : MK M A), naturality (exponential_F A \O M) M m
```

Unfortunately, we are not able to prove it in plain Coq (with or without impredicative `Set`), even if we consider particular functors `M` such as the identity functor.

[5] By functional extensionality, by naturality of `Ret`, and by definition of `from_component`.

The solution consists in assuming an axiom of parametricity for each functor `M` and derive naturality from it. That is, we follow the approach advocated by Wadler [34]. It has been shown to be sound in Coq [4, 5, 18, 19] and it is implemented by the ParamCoq plugin [18]. For instance, let us describe what happens when `M` is the list monad. First, we rewrite the naturality statement above in the case of the list functor (`map` is the map function of lists):

```
forall (X Y : UU0) (f : X -> Y) (g : A -> seq X),
  (map f \o m X) g = (m Y \o (exponential_F A \O M) # f) g.
```

The proof proceeds by induction on a proof-term of type

```
list_R X Y (fun x y => f x = y) (m X g) ((m Y \o (exponential_F A \O M) # f) g)
```

where `list_R X Y X_R l1 l2` means that the elements of lists `l1` and `l2` are pairwise related by the relation `X_R`. The role of ParamCoq is to generate definitions (including `list_R`) for us to be able to produce this proof. Concretely, starting from `MK`, ParamCoq generates the logical relation `T_R` of type (it is obtained by induction on types [11]):

```
(forall X : UU0, (A -> list X) -> list X) ->
  (forall X : UU0, (A -> list X) -> list X) -> UU0
```

Here, `T_R m1 m2` expands to:

```
forall (X1 X2 : UU0) (RX : X1 -> X2 -> UU0)
  (f1 : A -> list X1) (f2 : A -> list X2),
(forall a1 a2 : A, a1 = a2 -> list_R X1 X2 RX (f1 a1) (f2 a2)) ->
list_R X1 X2 RX (m1 X1 f1) (m2 X2 f2)
```

It is then safe to assume the following parametricity axiom:

```
Axiom param : forall m : MK M A, T_R m m.
```

The application of `param` is the first step to produce the proof required for the induction:

```
have : list_R X Y (fun x y = f x = y) (m X g) ((m Y \o (exponential_F A \O M) # f) g).
  apply: param.
  (* ∀ a a', a = a' ->
    list_R X Y (fun x y = f x = y) (g a) (((exponential_F A \O M) # f) g a') *)
```

The goal generated is proved by induction on `g a` which is a list.

The same approach is applied to other monads (identity, exception, option, state) [24, file `iparametricity_codensity.v`].

6.3 Lifting of Sigma-operations: Formal Statement

Before stating and proving the main theorem about lifting of sigma-operations, we formally define a special algebraic operation [15, Def. 25]. Let `E` be a functor, `M` be a monad, and `op` be an `E`-operation for `M`. The natural transformation `kappa` from `E` to `codensityT M` is defined by the components

$$\texttt{A}, (\texttt{s} : \texttt{E A}), \texttt{B}, (\texttt{k} : \texttt{A} \to \texttt{M B}) \mapsto \texttt{op B}\,((\texttt{E\#k})\,\texttt{s})$$

and `psik` is the algebraic `E`-operation for the monad `codensityT M` defined by:

```
Definition psik : E.-aoperation (codensityT M) := psi (kappa op).
```

Recall that the function `psi` has been defined in the proof of Theorem 1.

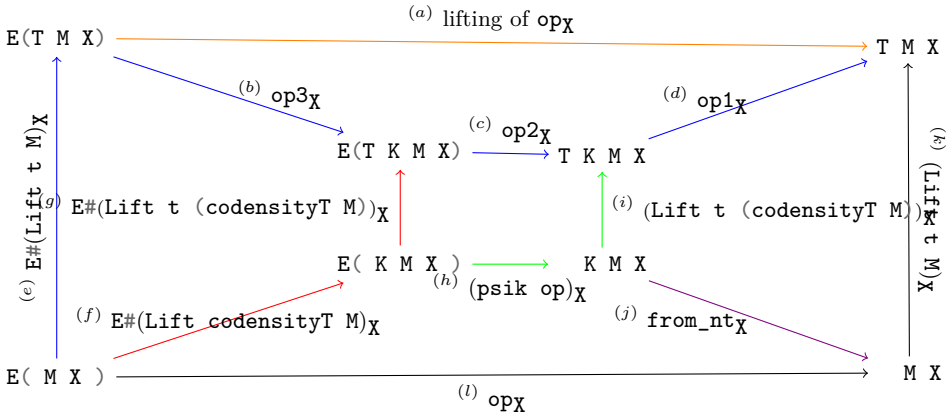

Figure 3 Proof of Uniform Lifting (Theorem 2).

▶ **Theorem 2** (Uniform Lifting [15, Thm. 27]). *Let `M` be a monad such that any computation `m : MK M A` is natural in the sense of Sect. 6.2 (hypothesis `naturality_MK`). Let `op` be an `E`-operation for `M` and `t` be a functorial monad transformer. We denote:*

- *by `op1` the term `Hmap t (from naturality_MK)` (see Sect. 6.2 for `from`, `Hmap` was defined in Sect. 3.4),*
- *by `op2` the algebraic lifting along `Lift t` of `(psik op)` (see just above for `psik`), and*
- *by `op3` the term `E ## Hmap t (monadM_nt (Lift codensityT M))` (see Sect. 6.1 for `codensityT`).*

Then the operation `op1 \v op2 \v op3` (where `\v` is the vertical composition seen in Sect. 2) is a lifting of `op` along `t`.

Proof. The proof is depicted by the diagram in Fig. 3.

The first step of the proof is to unfold the definition of lifting (which amounts to showing that the paths (a) (→) and (b)-(c)-(d) are equal). Consequently, the proof goal is rendered in Coq as follows (for all `X : UU0`):

```
Lift t M X \o op X = (op1 \v op2 \v op3) X \o E # Lift t M X
```

The second step of the proof is to show that the path (e)-(b) and the path (f)-(g) (→) are equal, which is achieved by appealing to the law of functor composition and the naturality of `Hmap`.

The next step is to show that the path (g)-(c) and the path (h)-(i) (→) are equal; this is by applying Theorem 1.

At this point, the goal becomes:

```
Lift t M X \o op X = (op1 X \o (Lift t (codensityT M) X \o psik op X))
                     \o E # Lift codensityT M X
```

It happens that we can use the naturality of `Hmap` to make the `from` function appear in the right-hand side of the goal:

```
Lift t M X \o op X = ((Lift t M X \o from naturality_MK X)
                     \o psik op X) \o E # Lift codensityT M X
```

The last step is to identify `op` with the composition of the `from` function, `psik op`, and `E # List codensityT M`, which is the purpose of a lemma [15, Prop. 26] (see [24, file `ifmt_lifting.v`, lemma `psikE`]). ◀

The proof script corresponding to the proof above is reproduced in Appendix A.3.

Finally, we show that, for all the monad transformers considered in this paper, the lifting of an algebraic operation provided by Theorem 2 coincides with the one provided by Theorem 1. This corresponds to the last results about modular monad transformers [15, Prop. 28].

7 Related Work

The example we detail in Sect. 4 adds to several examples of monadic equational reasoning [8, 9, 25–28]. Its originality is to use a parameterized interface and the `RunStateT` command, which are typical of programs written using monad transformers.

Huffman formalizes three monad transformers in the Isabelle/HOL proof assistant [12]. 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. Compared to Isabelle/HOL, the type system of Coq is more expressive so that we could formalize a much larger theory, even relying on extra features of Coq such as impredicativity and parametricity to do so.

Maillard proposes a meta language to define monad transformers in the Coq proof assistant [21, Chapter 4]. It is an instance implementation of one element of a larger framework to verify programs with monadic effects using Dijskstra monads [22]. The lifting of operations is one topic of this framework but it does not go as far as the deep analysis of Jaskelioff [14, 15, 17].

There are also formalizations of monads and their morphisms that focus on the mathematical aspects, e.g., UniMath [33]. However, the link to the monad transformers of functional programming is not done.

Monad transformers is one approach to combine effects. Algebraic effects is a recent alternative. It turns out that the two are related [30] and we have started to extend Monae to clarify formally this relation.

8 Conclusions and Future Work

In this paper, we extended Monae, a formalization of monadic equational reasoning, with monad transformers. We explained how it helps us to better organize the models of monads, thanks to sigma-operations in particular. We also explained how to extend the hierarchy of monad interfaces to handle programs written with monad transformers in mind. We also used our formalization of monad transformers to formalize the theory of liftings of modular monad transformers [15] using equational reasoning. For that purpose, we needed to fix the original presentation by using Coq's impredicativity and parametricity.

The main result of this paper is a robust, formal theory of monad transformers. We plan to extend the hierarchy of monad interfaces of Monae similarly to how we proceeded for `exceptStateRunMonad`. Such an extension will call for more models to be formalized and we expect our formalized theory of liftings to be useful on this occasion.

Results up to Sect. 5 hold whether or not `Set` is impredicative. In contrast, the setting of Sect. 6 conflicts with Monae programs relying on some data structures from the MathComp library [23] (such as fixed-size lists) or from the InfoTheo library [13] (such as probability distributions) because these data structures are in `Type` and cannot be computed with monads in `Set`. One could think about reimplementing them but this is a substantial amount of work. A cheap way to preserve these data structures together with the theorem on lifting of sigma-operations is to disable universe checking as soon as this theorem is used; this way,

monads can stay in `Type`. Disabling universe checking is not ideal because it is unsound in general[6]; note however that this is sometimes used for the formalization of category-theoretic notions (e.g., [3, Sect. 6]). How to improve this situation is another direction for future work.

References

1 Reynald Affeldt, Jacques Garrigue, David Nowak, and Takafumi Saikawa. A trustful monad for axiomatic reasoning with probability and nondeterminism. arXiv cs.LO 2003.09993, 2020. `arXiv:2003.09993`.

2 Reynald Affeldt, David Nowak, and Takafumi Saikawa. A hierarchy of monadic effects for program verification using equational reasoning. In *13th International Conference on Mathematics of Program Construction (MPC 2019), Porto, Portugal, October 7–9, 2019*, volume 11825 of *Lecture Notes in Computer Science*, pages 226–254. Springer, 2019.

3 Benedikt Ahrens, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. Categorical structures for type theory in univalent foundations. In *26th EACSL Annual Conference on Computer Science Logic (CSL 2017), August 20–24, 2017, Stockholm, Sweden*, volume 82 of *LIPIcs*, pages 8:1–8:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

4 Robert Atkey, Neil Ghani, and Patricia Johann. A relationally parametric model of dependent type theory. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014), San Diego, CA, USA, January 20–21, 2014*, pages 503–516. ACM, 2014.

5 Jean-Philippe Bernardy and Guilhem Moulin. A computational interpretation of parametricity. In *27th Annual IEEE Symposium on Logic in Computer Science (LICS 2012), Dubrovnik, Croatia, June 25–28, 2012*, pages 135–144. IEEE Computer Society, 2012.

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

7 Herman Geuvers. Inconsistency of classical logic in type theory. Short note, December 2001. URL: `http://www.cs.ru.nl/~herman/PUBS/newnote.ps.gz`.

8 Jeremy Gibbons. Unifying theories of programming with monads. In *4th International Symposium on Unifying Theories of Programming (UTP 2012), Paris, France, August 27–28, 2012*, volume 7681 of *Lecture Notes in Computer Science*, pages 23–67. Springer, 2012.

9 Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. In *16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011), Tokyo, Japan, September 19–21, 2011*, pages 2–14. ACM, 2011.

10 Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A small scale reflection extension for the Coq system. Technical report, INRIA, 2008. Version 17 (Nov 2016). Now part of [32].

11 Jean Goubault-Larrecq, Slawomir Lasota, and David Nowak. Logical relations for monadic types. *Math. Struct. Comput. Sci.*, 18(6):1169–1217, 2008.

12 Brian Huffman. Formal verification of monad transformers. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 15–16. ACM, 2012.

13 Infotheo. A Coq formalization of information theory and linear error-correcting codes. Coq scripts. Last stable release: 0.3, 2021. URL: `https://github.com/affeldt-aist/infotheo/`.

14 Mauro Jaskelioff. *Lifting of Operations in Modular Monadic Semantics*. PhD thesis, University of Nottingham, 2009.

[6] One can derive `False` by applying a variant of Hurkens paradox (see `https://coq.inria.fr/library/Coq.Logic.Hurkens.html`).

15 Mauro Jaskelioff. Modular monad transformers. In *18th European Symposium on Programming (ESOP 2009), York, UK, March 22–29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2009.

16 Mauro Jaskelioff. Private communication, May 2020.

17 Mauro Jaskelioff and Eugenio Moggi. Monad transformers as monoid transformers. *Theor. Comput. Sci.*, 411(51-52):4441–4466, 2010.

18 Chantal Keller and Marc Lasson. Parametricity in an impredicative sort. In *21st Annual Conference of the EACSL on Computer Science Logic (CSL 2012), September 3–6, 2012, Fontainebleau, France*, volume 16 of *LIPIcs*, pages 381–395. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.

19 Neelakantan R. Krishnaswami and Derek Dreyer. Internalizing relational parametricity in the extensional calculus of constructions. In *Computer Science Logic 2013 (CSL 2013), September 2–5, 2013, Torino, Italy*, volume 23 of *LIPIcs*, pages 432–451. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.

20 Sheng Liang, Paul Hudak, and Mark P. Jones. 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*, pages 333–343. ACM Press, 1995.

21 Kenji Maillard. *Principes de la Vérification de Programmes à Effets Monadiques Arbitraires*. PhD thesis, Université PSL, November 2019.

22 Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *PACMPL*, 3(ICFP):104:1–104:29, 2019.

23 Mathematical Components Team. Mathematical Components library, 2007. Last stable version 1.11 (2020). URL: `https://github.com/math-comp/math-comp`.

24 Monae. Monadic effects and equational reasoning in Coq. Coq scripts. Last stable release: 0.3, 2021. URL: `https://github.com/affeldt-aist/monae/`.

25 Shin-Cheng Mu. Calculating a backtracking algorithm: An exercise in monadic program derivation. Technical Report TR-IIS-19-003, Institute of Information Science, Academia Sinica, June 2019.

26 Shin-Cheng Mu. Equational reasoning for non-deterministic monad: A case study of Spark aggregation. Technical Report TR-IIS-19-002, Institute of Information Science, Academia Sinica, June 2019.

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

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

29 Célestine Sauvage, Reynald Affeldt, and David Nowak. Vers la formalisation en Coq des transformateurs de monades modulaires. In *Trente-et-unièmes Journées Francophones des Langages Applicatifs (JFLA 2020), Janvier 2020, Gruissan, France*, pages 23–30, 2020. In French. URL: `https://hal.archives-ouvertes.fr/hal-02434736v2/document`.

30 Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. Monad transformers and modular algebraic effects: what binds them together. In *12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019), Berlin, Germany, August 18–23, 2019*, pages 98–113. ACM, 2019.

31 The Coq Development Team. Impredicative set, 2019. Last revision: 2019-07-12. URL: `https://github.com/coq/coq/wiki/Impredicative-Set`.

32 The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Inria, 2021. Version 8.13.0. URL: `https://coq.inria.fr/distrib/current/refman/`.

33 Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. UniMath – a computer-checked library of univalent mathematics. Available at `https://github.com/UniMath/UniMath`.

34 Philip Wadler. Theorems for free! In *4th international conference on Functional programming languages and computer architecture (FPCA 1989), London, UK, September 11–13, 1989*, pages 347–359. ACM, 1989.

A Proof Scripts for the Three Applications of This Paper

The following proof scripts have been copied verbatim from MONAE [24] for the reader's convenience. We claim that these proof scripts are readable and short in the sense that each line corresponds to a genuine proof step and that there are few administrative tactics hampering reading. The terseness of the SSREFLECT tactic language could actually make these proof scripts much shorter but that is not our point here. In particular, we make explicit the proof steps of Theorems 1 and 2 (in Appendices A.2 and A.3) with `transitivity` steps or explicit `rewrite` steps followed by indented (sub-)proof scripts.

A.1 Proof Script for the Correctness of `fastProduct`

The following proof script can be found in [24, file `example_transformer.v`].

```
Lemma fastProductCorrect l n :
  evalStateT (fastProduct l) n = Ret (product l).
Proof.
rewrite /fastProduct -(mul1n (product _)); move: 1.
elim: l => [ | [ | x] l ih] m.
- rewrite muln1 bindA bindretf putget.
  rewrite /evalStateT RunStateTCatch RunStateTBind RunStateTPut bindretf.
  by rewrite RunStateTRet RunStateTRet catchret bindretf.
- rewrite muln0.
  rewrite /evalStateT RunStateTCatch RunStateTBind RunStateTBind RunStateTPut.
  by rewrite bindretf RunStateTFail bindfailf catchfailm RunStateTRet bindretf.
- rewrite [fastProductRec _]/=.
  by rewrite -bindA putget bindA bindA bindretf -bindA -bindA putput ih mulnA.
Qed.
```

A.2 Proof Script for Theorem 1 [15, Thm. 19]

The following proof script can be found in [24, file `monad_transformer.v`].

```
Section uniform_algebraic_lifting.
Variables (E : functor) (M : monad) (op : E.-aoperation M).
Variables (N : monad) (e : monadM M N).

Definition alifting : E.-aoperation N := psi (monadM_nt e \v phi op).

Lemma aliftingE :
  alifting = (fun X => Join \o e (N X) \o phi op (N X)) :> (_ ~~> _).
Proof. by []. Qed.

Theorem uniform_algebraic_lifting : lifting op e alifting.
Proof.
move=> X.
apply fun_ext => Y.
```

```
rewrite /alifting !compE psiE vcompE phiE !compE.
rewrite (_ : (E # Ret) ((E # e X) Y) =
             (E # (M # e X)) ((E # Ret) Y)); last first.
  rewrite -[in LHS]compE -functor_o.
  rewrite -[in RHS]compE -functor_o.
  rewrite (natural RET).
  by rewrite FIdf.
rewrite (_ : op (N X) ((E # (M # e X)) ((E # Ret) Y)) =
             (M # e X) (op (M X) ((E # Ret) Y))); last first.
  rewrite -(compE (M # e X)).
  by rewrite (natural op).
transitivity (e X (Join (op (M X) ((E # Ret) Y)))); last first.
  rewrite joinE monadMbind.
  rewrite bindE -(compE _ (M # e X)).
  by rewrite -natural.
by rewrite -[in LHS](phiK op).
Qed.
End uniform_algebraic_lifting.
```

A.3 Proof Script for Theorem 2 [15, Thm. 27]

The following proof script can be found in [24, file `ifmt_lifting.v`].

```
Section uniform_sigma_lifting.
Variables (E : functor) (M : monad) (op : E.-operation M) (t : FMT).
Hypothesis naturality_MK : forall (A : UU0) (m : MK M A),
  naturality_MK m.

Let op1 : t (codensityT M) ~> t M := Hmap t (from naturality_MK).
Let op2 := alifting (psik op) (Lift t _).
Let op3 : E \O t M ~> E \O t (codensityT M) :=
  E ## Hmap t (monadM_nt (Lift codensityT M)).

Definition slifting : E.-operation (t M) := op1 \v op2 \v op3.

Theorem uniform_sigma_lifting : lifting_monadT op slifting.
Proof.
rewrite /lifting_monadT /slifting => X.
apply/esym.
transitivity ((op1 \v op2) X \o op3 X \o E # Lift t M X).
  by rewrite (vassoc op1).
rewrite -compA.
transitivity ((op1 \v op2) X \o
  ((E # Lift t (codensityT M) X) \o (E # Lift codensityT M X))).
  congr (_ \o _); rewrite /op3.
  by rewrite -functor_o -natural_hmap functor_o functor_app_naturalE.
transitivity (op1 X \o
  (op2 X \o E # Lift t (codensityT M) X) \o E # Lift codensityT M X).
  by rewrite vcompE -compA.
rewrite -uniform_algebraic_lifting.
transitivity (Lift t M X \o from naturality_MK X \o (psik op) X \o
  E # Lift codensityT M X).
  congr (_ \o _).
  by rewrite compA natural_hmap.
rewrite -2!compA.
```

```
congr (_ \o _).
by rewrite compA -psikE.
Qed.
End uniform_sigma_lifting.
```