Experimenting with Monadic Equational Reasoning in Coq

Reynald Affeldt　David Nowak

In order to prove properties of low-level programs in a proof-assistant, it is common to proceed by refining an abstract model into a deep encoding of the program. Here, we lay the groundwork for exploring the alternative approach consisting in reasoning directly on a shallow encoding of the program to be verified. For that purpose, we mechanize in the Coq proof-assistant an approach by Gibbons and Hinze for monadic equational reasoning. The main idea of our mechanization is to formalize monads and their algebraic laws like it is done for mathematical structures so as to take advantage of Coq's rewriting capabilities. This leads to a mechanization with little overhead that adds the possibility for rigorous model instantiations. In this paper, we explain the key ideas of our mechanization, discuss in particular the probability monad, and give a denotational semantics to a small imperative programming language with a combination of the state and trace monads. The latter application leads us to extend monadic equational reasoning with equations linking a state monad with a trace monad.

1 Introduction

1.1 Motivation

Our ultimate goal is to use the Coq proof-assistant to reason about low-level languages (e.g., produce formal proofs of functional correctness of programs). For that purpose, the approach that is usually taken is to proceed via a deep embedding of the target language. This requires substantial instrumentations of syntax and semantics, resulting in technical lemmas that are difficult to use, which in turn call for meta-programming (using Ltac in Coq). The alternative approach consisting of using a shallow embedding bears the promise of a more direct access to the proof-assistant native tactics and it indeed experimentally met some success (e.g., [16][18]). Most of the time, this approach relies on a combination of monads and Hoare logic (e.g., [17]). There is however another approach based on monads to reason about programs: *monadic equational rewriting*, an approach best explained by Gibbons and Hinze [12]. We believe that it is worth exploring because the rewriting tactics of Coq have been polished over the years and are now powerful enough to gear huge formal developments; for example, proofs in the Mathematical Components library (MathComp) essentially rely on rewriting [15].

As a preliminary step towards the verification of low-level programs, in this paper, we mechanize monadic equational rewriting. Our goal is to make sure that we can produce formal proofs based on monadic equational reasoning that are at least as pleasant as pencil-and-paper proofs. For that purpose, we mechanize in Coq the paper by Gibbons and Hinze [12], as well as related work [11][22].

Coq における Monadic 等式推論の実験

アフェルト・レナルド, 産業技術総合研究所, National Institute of Advanced Industrial Science and Technology.
ノヴァック・ダヴィッド, フランス国立科学研究センター・リール大学, CNRS/University of Lille.

This provides us with a solid mechanization of algebraic effects such as failures, exceptions, nondeterminism, states, probabilities (and more), that are key to formalize the semantics of low-level languages.

It should be noted that the formalization of monadic equational reasoning is already interesting in itself. Its correctness can be tricky (for example, the early [12] needed to be patched in [1]) so that mechanized tools are certainly welcome. There is also an interest in formal frameworks to reason about Haskell-like programs in Coq (e.g., [7]) so that a proper mechanization of monadic equational reasoning could actually be readily useful.

1.2 Contributions

We formalize the theory of algebraic effects from [12] (see Sections 3 and 4) and even extend it by providing formal models (see Sect. 5).

We mechanize numerous examples: all the examples from [12], most examples from [11] (which overlaps and complements [12]), and also some examples from related work [22]. The result is very satisfactory. The formal proofs closely match their pencil-and-paper counterparts (discrepancies are minor, see Sect. 7.2). Proofs can actually be made even shorter thanks to the terseness of Coq's tactic language and its automation capabilities (Sect. 2 provides a simple example, see the code [5] for more).

Last, we extend our mechanization with other monads and present an application that provides a shallow encoding of a semantics using monads (see Sect. 6).

1.3 Formalization Approach

Our successful mechanization owes much to the level of details provided by the authors using monadic equational reasoning in their papers. It is also the result of appropriate technical choices:

- We use Coq canonical structures [20]. We use them in particular to formalize the hierarchy of algebraic effects (see Sect. 3) using *packed classes* [10], a methodology used in the MathComp library to formalize the hierarchy of mathematical structures. We also use canonical structures to formalize probabilities (see Sect. 5).
- We use the SSReflect tactics and libraries [14]. SSReflect tactics emphasize proof by rewriting, making it easier to mimic monadic equational reasoning. The SSReflect library for lists is closer to the Haskell library than Coq's standard library: it already provides Haskell-like notations (e.g., notation for comprehension) and more functions (e.g., `allpairs`, a.k.a. `cp` in Haskell). Also, we benefit from other SSReflect/MathComp-compatible libraries to formalize the model of the probability monad (the theory of finitely supported distributions comes from [4]).
- We benefit from the real numbers of the Coq standard library: they provide tactics to deal automatically with reals despite their axiomatic encoding. In particular the tactics `field`, `fourier`, and `nstaz` are important in practice to compute probabilities.

2 A Simple Example of Monadic Equational Reasoning in Coq

We start with a simple example of monadic equational reasoning taken from [12]. It establishes the equivalence between a functional implementation of the product of integers (`product` below) with a monadic version (`fastproduct`). In [12], the proof is carried out by a series of rewritings that we have reproduced (faithfully) in Fig. 1 (on the left). Let us comment on the equivalent Coq mechanization that is displayed on the right of Fig. 1.

We first define the product of natural numbers

Original proof (*Sect. 5.1 of [12]*)	Coq proof (lhs of the goal, tactics inbetween)
`fastprod xs`	`fastproduct s`
=⟦ definition of `fastprod` ⟧	=⟦ `rewrite /fastproduct` ⟧
`catch (work xs) (return 0)`	`Catch (work s) (Ret 0)`
=⟦ specification of `work` ⟧	=⟦ `rewrite /work` ⟧
`catch (if 0 ∈ xs then fail`	`Catch (if 0 \in s then Fail`
`else return (product xs)) (return 0)`	`else Ret (product s)) (Ret 0)`
=⟦ lift out the conditional ⟧	=⟦ `rewrite lift_if if_ext` ⟧
`if 0 ∈ xs then catch fail (return 0)`	`((if 0 \in s then Catch Fail (Ret 0)`
`else catch (return (product xs)) (return 0)`	`else Catch (Ret (product s)) (Ret 0))`
=⟦ laws of catch, fail, and return ⟧	=⟦ `rewrite catchfailm catchret` ⟧
`if 0 ∈ xs then return 0 else return (product xs)`	`(if 0 \in s then Ret 0 else Ret (product s))`
=⟦ arithmetic: $0 \in xs \Rightarrow$ `product xs` $= 0$ ⟧	=⟦ `case: ifPn => // /product0` ⟧
`if 0 ∈ xs then return (product xs)`	`(Lemma product0 s : 0 \in s -> product s = 0.)`
`else return (product xs)`	`Ret 0`
=⟦ redundant conditional ⟧	=⟦ `move <-` ⟧
`return (product xs)`	`Ret (product s)`

Figure 1 Comparison between a proof from [12] and its Coq formalization

as follows:

```
Definition product (s : seq nat) :=
  foldr muln 1 s.
```

A "faster" product can be implemented using the failure monad (with operator `Fail`) and the exception monad (with operator `Catch`):

```
Definition work
  {M : failMonad} (s : seq nat) : M nat :=
  if 0 \in s then Fail else Ret (product s).
Definition fastproduct
  {M : exceptMonad} s : M nat :=
  Catch (work s) (Ret 0 : M _).
```

The formalization of the failure monad is explained in the next section (Sect. 3). The exception monad inherits[†1] from the failure monad to which it adds in particular the following properties:

```
Variables (M : exceptMonad).
Lemma catchfailm : forall A,
  left_id Fail (@Catch M A).
Lemma catchret : forall A x,
  left_zero (Ret x : M _) (@Catch M A).
```

The formal proof that `fastproduct` is pure, i.e., that it never throws an unhandled exception, can be compared to its pencil-and-paper counterpart in Fig. 1. One can observe that both proofs are essentially the same, though in practice the Coq proof script will be streamlined in two lines of code (of less than 80 characters):

```
Lemma fastproductE s : fastproduct s = Ret (product s).
Proof.
rewrite /fastproduct /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 on this example is not because it is short. In fact, the same can be said of all the examples in [12], even the longer and more complicated Monty Hall problem and tree-relabeling example (see [5]).

The formal proof above relies on a formalization of monads organized as a hierarchy in such a way that properties of a monad at a lower level (say, `monad`) can be enjoyed by monads at higher levels (such as `exceptMonad`). How to achieve this is the

†1 Inheritance is also explained in the next section (Sect. 3).

purpose of the next section.

3 Formalization of Algebraic Effects in Coq

The heart of our mechanization is a formalization of a hierarchy of monads and their algebraic laws. It is implemented using *packed classes* [10]. In this section, we explain the formalization using an example: the combination of the failure monad and the choice monad into the non-determinism monad. The next section (Sect. 4) provides an overview of the complete hierarchy.

3.1 The Type of Monad and its Extension as a Packed Class

The class of monads is formalized as a dependent record (`class_of`, in the module[†2] `Monad`) with two constructors (`ret` and `bind`) satisfying the monad laws. The type of monads `monad` (a notation for `Monad.t`) is a pair of a `Type -> Type` function that satisfies the `class_of` interface:

```
(* Module Monad *)
Record class_of (m : Type -> Type) : Type :=
Class {
  ret : forall A, A -> m A ;
  bind : forall A B, m A -> (A -> m B) -> m B ;
  _ : Laws.left_neutral bind ret ;
  _ : Laws.right_neutral bind ret ;
  _ : Laws.associative bind }.
Record t : Type := Pack {
  m : Type -> Type ; class : class_of m }.
Notation monad := t.
Coercion m : monad >-> Funclass.
```

The purpose of the coercion from the type `monad` is to let one write `M A` when `M` has the type `monad`, `M` being then understood as the projection `m M`.

The failure monad is formalized by providing a mixin with a `fail` operator and the property that the latter is a left-zero of sequential composition. The class of failure monads are monads that moreover satisfy the corresponding mixin (see `class_of` below). The type of failure monads is a dependent record with a function of type `Type -> Type` and a proof that this function belongs to the class of failure monads:

```
(* Module MonadFail *)
Record mixin_of (M : monad) : Type := Mixin {
  fail : forall A, M A ;
  _ : Laws.left_zero (@Bind M) fail }.
Record class_of (m : Type -> Type) := Class {
  base : Monad.class_of m ;
  mixin : mixin_of (Monad.Pack base) }.
Structure t := Pack {
  m : Type -> Type ; class : class_of m }.
Notation failMonad := t.
Coercion baseType : failMonad >-> monad.
Canonical baseType.
```

Failure monads are furthermore coerced to monads and made canonical.

It is cumbersome to use the `fail` operator of the mixin directly. For this reason, it is redefined (as `op_fail`) so that its type features record projections w.r.t. the monad. This is actually under this form that we will use the `fail` operator (through the notation `Fail`):

```
(* in Module MonadFail *)
Definition op_fail (M : t) : forall A, m M A :=
  let: Pack _ (Class _ (Mixin x _)) := M
  return forall A, m M A in x.
Arguments op_fail {M A} : simpl never.
Notation Fail := op_fail.
```

Likewise, the property that `Fail` is a left-zero of `Bind` is better wrapped as an additional lemma:

```
(* outside of Module MonadFail *)
Variable (M : failMonad).
Lemma bindfailm :
  Laws.left_zero (@Bind _) (@Fail M).
```

3.2 Combination of Effects: Failure and Choice into Non-determinism

Similarly to the failure monad, let us define the choice monad. It extends the type `monad` with the operator `alt` (pencil-and-paper notation: $\cdot \Box \cdot$) which is associative and such that `Bind` left-distributes over it:

```
(* Module MonadAlt *)
Record mixin_of (M : monad) : Type := Mixin {
  alt : forall A, M A -> M A -> M A ;
  _ : forall A, associative (@alt A) ;
  _ : Laws.bind_left_distributive (@Bind M) alt }.
```

†2 We use modules just for the name-space.

```
Record class_of (m : Type -> Type) : Type :=
Class {
  base : Monad.class_of m ;
  mixin : mixin_of (Monad.Pack base) }.
Structure t := Pack {
  m : Type -> Type ; class : class_of m }.
Definition op_alt M := (* omitted *).
Notation Alt := op_alt.
Notation altMonad := t.
```

Let `[~i]` be an infix Coq notation for the non-deterministic choice:

```
Notation "x '[~i]' y" := (Alt x y).
```

The non-determinism monad is a combination of a failure monad with a choice monad. This is done by first specifying the additional properties of the non-deterministic choice (the fact that `Fail` is a unit of choice) as a new mixin (see `mixin_of` just below) and then declaring a class that combines this mixin with the class of failure monads and choice monads (`class_of` below):

```
(* Module MonadNondet *)
Record mixin_of (M : failMonad)
  (a : forall A, M A -> M A -> M A) : Type :=
Mixin {
  _ : Laws.left_id (@Fail M) a ;
  _ : Laws.right_id (@Fail M) a
}.
Record class_of (m : Type -> Type) : Type :=
Class {
  base : MonadFail.class_of m ;
  mixin : MonadAlt.mixin_of
    (Monad.Pack (MonadFail.base base)) ;
  ext : @mixin_of (MonadFail.Pack base)
                  (@MonadAlt.alt _ mixin)
}.
Structure t : Type := Pack {
  m : Type -> Type ; class : class_of m }.
Notation nondetMonad := t.
```

We provide canonical structures so that monads of type `nondetMonad` can be seen as monads of type `failMonad` or `altMonad`:

```
(* in Module Nondet *)
Definition baseType (M : t) :=
  MonadFail.Pack (base (class M)).
Coercion baseType : nondetMonad >-> failMonad.
Canonical baseType.
Definition alt_of_nondet (M : nondetMonad) :=
  MonadAlt.Pack (MonadAlt.Class
    (mixin (class M))).
Canonical alt_of_nondet.
```

As a consequence of the use of packed classes, it becomes possible for monads to enjoy the notations and properties of the monads from which they derive. For instance, in the example just below, it becomes possible with a monad of type `nondetMonad` to use in combination (1) the `Fail` definition and the `[~i]` notation, and (2) the properties of `failMonad` with a monad of type `nondetMonad`:

```
Lemma test_canonical (M : nondetMonad)
  A (a : M A) (b : A -> M A) :
  a [~i] (Fail >>= b) = a [~i] Fail.
Proof.
by rewrite bindfailm.
Qed.
```

Looking (with, say, `Set Printing All`) at the proof term reveals that Coq has introduced calls to `MonadNondet.baseType` and `alt_of_nondet` appropriately. This shows that packed classes achieve at the same time both notation overloading and inheritance, thus providing the main ingredients to the formalization of monadic equational reasoning.

4 The Hierarchy of Monads

The hierarchy of monads and their algebraic laws that we formalize is essentially the one from [12], with minor adjustments dictated by [11] and [1]. The result is displayed in Fig. 2.

We have already explained `monad`, `failMonad`, `altMonad`, and `nondetMonad` in Sect. 3.2, and `exceptMonad` in Sect. 2. The combination of effects to form other monads is achieved similarly to the non-determinism monad: we use one existing class as the base, extend it with an existing mixin, and possibly add properties as a new mixin.

Prose explanations about each monad can be found in the literature. Here follows a short reading guide. See the code [5] for details.

- `altMonad` is the choice monad already seen in Sect. 3.2. The examples of [12] relying on non-deterministic choice use this monad. However, the combination of non-determinism and

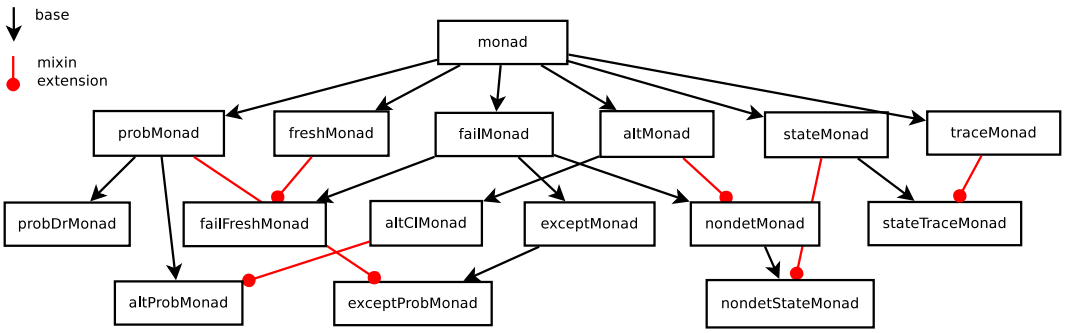

Figure 2 Hierarchy of monads formalized. This includes the monads from Gibbons et al. [12] [11] [1]. See Table 1 for an overview of the algebraic laws.

probability in `altProbMonad` requires idempotence and commutativity [11]. Idempotence and commutativity are also required in related work [22]. We have therefore inserted the monad `altCIMonad` in the hierarchy.

- We saw `exceptMonad` in the example of Sect. 2. It is explained in *Sect. 5 of [12]*.
- The state monad `stateMonad` and the state monad with non-determinism `nondetStateMonad` are explained in *Sect. 6 of [12]*. We come back to `stateMonad` in Sect. 6. 1.
- The probability monad `probDrMonad` was originally explained in *Sect. 8 of [12]*. The main difference with [12] is that we extract from the probability monad `probDrMonad` the monad `probMonad` as an intermediate step. `probDrMonad` extends `probMonad` with right distributivity of bind $(\cdot \gg\!= \cdot)$ over probabilistic choice $(\cdot \triangleleft \cdot \triangleright \cdot)$. The reason is that this latter property is not compatible with distributivity of probabilistic choice over non-deterministic choice $(\cdot \,\square\, \cdot)$ and therefore needs to be put aside so as to be able to form `altProbMonad` by combining `probMonad` and `altMonad` (the issue is explained in [1]). We come back to `probMonad` in Sect. 5.
- `exceptProbMonad` combines probability and exception and comes from *Sect. 7.1 of [11]*.
- `freshMonad` and `failFreshMonad` are explained in *Sect. 9.1 of [12]*. `freshMonad` provides an operator to generate fresh labels.
- `traceMonad` and `stateTraceMonad` are the topic of Sect. 6. 1.

Our formalization [5] includes all the examples from [12], as well as examples from [11] and [22]. This includes in particular the eight queens puzzle from [12], several variants of the Monty Hall problem from [12] and [11] (to illustrate the various probability monads), and the tree-relabeling example from [12] that originally motivated monadic equational reasoning.

5 Formalization of the Probability Monad and its Model

In the previous sections, we explained how we formalized a hierarchy of monads that mostly comes from [12]. In this section, we provide more details about the formalization of the probability monad.

The formalization of the probability monad is interesting because it shows that using Coq makes it possible to complete the work by Gibbons et al. by showing how to provide rigorous models. In *Sect. 4 of [11]*, the model of the probability monad is informally explained in terms of the datatype *Dist* of probability-weighted lists that does not take into account permutations, zero-weighted/repeated elements, etc. However (as hinted at by Gibbons et

Interface	Operators	Axioms
monad	Ret	bindretf (left neutral)
	Bind/>>=/>>	bindmret (right neutral)
		bindA (associativity)
failMonad	Fail	bindfailm (fail left-zero of bind)
altMonad	Alt/[~i]	alt_bindDl (bind left-distributes over alt)
		altA (associativity)
altCIMonad		altmm (idempotence)
		altC (commutativity)
nondetMonad		altmfail (fail right-id of alt)
		altfailm (fail left-id of alt)
exceptMonad	Catch	catchfailm (fail left-id of catch)
		catchmfail (fail right-id of catch)
		catchA (associativity)
		catchret (ret left-zero of catch)
stateMonad	Get	putget, getputskip, putput, getget (see Sect. 6.1.1)
	Put	
nondetStateMonad		bindmfail (fail right-zero of bind)
		alt_bindDr (bind right-distributes over choice)
freshMonad	Fresh	
failFreshMonad	distinct	failfresh_bindmail (fail right-zero of bind)
		assert (distinct M) \o Symbols = Symbols
probMonad	Choice	choicemm (idempotence)
		choice0, choice1 (identity laws)
		choiceA (quasi associativity)
		choiceC (skewed commutativity)
		prob_bindDl (bind left-distributivites over choice)
probDrMonad		prob_bindDr (bind right-distributes over choice)
altProbMonad		choiceDr (prob. choice right-distributes over nondet. choice)
exceptProbMonad		catchDl (catch left-distributes over choice)
traceMonad	Mark	
stateTraceMonad	Run	runret, runbind, runget, runput, runmark (see Sect. 6.1.3)

Table 1 Summary of monad operators and their algebraic laws (mostly coming from [12])

al.), it takes a bit more effort to provide an initial model.

5.1 The Probability Monad

The first step towards a formal model of the probability monad is to provide a type `Prob.t` of probabilities (reals between 0 and 1):

```
Module Prob.
Record t := mk {
  p :> R ;
  H : (0 <= p <= 1)%R }.
Definition H' (p : t) := H p.
Arguments H' : simpl never.
Notation "[Pr 'of' q ]" := (@mk q (@H' _)).
```

The significance of this definition and this notation is that it makes it possible to succinctly write: `[Pr of 0]` for the 0 probability, `[Pr of / 2]` for the $\frac{1}{2}$ probability, `[Pr of INR m / INR (m + n)]` for the probability $\frac{m}{m+n}$, etc. This is under the condition that we equip Coq with appropriate canonical structures. For example, here follows the registration of the proof $0 \leq 0 \leq 1$ that makes it possible to write `[Pr of 0]`:

```
Lemma OO1 : (R0 <= R0 <= R1)%R.
Canonical prob0 := Prob.mk OO1.
```

The above datatype and notation lead us to the following mixin (the one of `probMonad` from Fig. 2):

```
Record mixin_of (M : monad) : Type := Mixin {
choice : forall (p : Prob.t) A, M A -> M A -> M A
  where "mx <| p |> my" := (choice p mx my) ;
_ : forall A (mx my : M A),
  mx <| [Pr of 0] |> my = my ;
_ : forall A (mx my : M A),
  mx <| [Pr of 1] |> my = mx ;
_ : forall A p (mx my : M A),
```

```
  mx <| p |> my = my <| [Pr of p.~] |> mx ;
_ : forall A p, idempotent (@choice p A) ;
_ : forall A (p q r s : Prob.t) (mx my mz : M A),
  (p = r * s :> R /\ s.~ = p.~ * q.~)%R ->
  mx <| p |> (my <| q |> mz) =
    (mx <| r |> my) <| s |> mz ;
_ : forall p, Laws.bind_left_distributive
              (@Bind M) (choice p) }.
```

`p.~` is a notation for the real $\bar{p} = 1 - p$, which happens to be a probability when p is (and can thus be handled by the probability notation `[Pr of ...]` thanks to an appropriate canonical structure declaration).

5.2 A Model Using Probability Distributions

Providing a formal model for the monad seen in the previous section (Sect. 5.1) amounts to instantiate its interface with concrete objects.

5.2.1 Probability Distributions

In [2][3][4], we introduced a formalization of probability distributions over a finite type (`A` below has type `finType`) based on the following definition:

```
Record dist := mkDist {
  pmf :> A -> R+ ;
  pmf1 : \rsum_(a in A) pmf a = 1%R}.
```

The first field is a probability mass function with non-negative outputs coupled with a proof that the outputs sum to 1.

5.2.2 Formal Model of Monads

Providing a formal model of monads amounts to provide concrete implementation for `Ret` and the bind operator†3. For instance, the bind operator can be defined using distributions as follows:

```
(* Module DistBind *)
Variables (A B : finType)
  (p : dist A) (g : A -> dist B).
Definition f b := \rsum_(a in A) p a * (g a) b.
Definition d : dist B := ...
```

This operator can further be proved to satisfy the monad laws, for example, associativity:

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

Completing the model with `Ret` and its properties is not difficult.

5.2.3 Formal Model of the Probability Monad

Last, we need to provide an implementation for the interface of the probability monad seen in Sect. 5.1. For instance, the probabilistic choice operator corresponds to the construction of a new distribution `d` from two distributions `d1` and `d2` biased by a probability `p`:

```
(* Module ConvexDist *)
Variables (A : finType) (d1 d2 : dist A) (p : R).
Definition f a := (p * d1 a + p.~ * d2 a)%R.
Definition d : dist A := ...
```

This implementation of probabilistic choice can further be proved to have the expected properties, for example, skewed commutativity:

```
Lemma quasi_commute (d1 d2 : dist A) p
  (Hp : 0 <= p <= 1) (Hp' : 0 <= p.~ <= 1) :
  d d1 d2 Hp = d d2 d1 Hp'.
```

We see here that using `Prob.t` (instead of, say, the mere type for reals `R`) is crucial to provide a model.

6 An Application to Program Semantics

In this section, we extend the approach of monadic equational reasoning to the formalization of program semantics. For that purpose, we add a *trace monad* to the hierarchy of [12] (see Fig. 2). Below, we explain how we compose it with the state monad. The purpose of the trace monad is to be able to express properties over the trace of a program. For example, if the trace includes nonces, we might want to prove that they are all distinct.

6.1 The State-Trace Monad

The state-trace monad is the result of combining a state monad with a trace monad. We start by recalling the interface of the state monad.

†3 The type of monad in Sect. 3.1 used `Type -> Type`. In this section, we need to specialize it to `finType -> Type`.

6.1.1 The State Monad

The state monad denotes computations that transform a state (of type `S` below). It comes with a `Get` function to yield a copy of the state and a `Put` function to overwrite it. These functions are constrained by four axioms whose usage is demonstrated by Gibbons and Hinze with the eight queens puzzle ([11], Coq formalization in [5]):

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

6.1.2 The Trace Monad

Our trace monad just extends monads with a `Mark` operator to record events:

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

Its semantics becomes apparent in the run interface of the next section (Sect. 6.1.3).

6.1.3 The Run Interface

The run interface extends the state monad simultaneously with a `Run` operator and a `Mark` operator. The meaning of the `Run` operator is captured by several axioms that give a meaning to each operator involved in the monad. Here follows the run interface; the `Mark` operator is abstracted as `op`:

```
Record mixin_of T S (M : stateMonad S)
  (op : T -> M unit) : Type := Mixin {
run : forall A,
  M A -> S * seq T -> A * (S * seq T) ;
_ : forall A (a : A) s, run (Ret a) s = (a, s) ;
_ : forall A B (m : M A) (f : A -> M B) s,
  run (Do{ a <- m ; f a}) s =
  let: (a', s') := run m s in run (f a') s' ;
_ : forall s l, run Get (s, l) = (s, (s, l)) ;
_ : forall s l s',
  run (Put s') (s, l) = (tt, (s', l)) ;
_ : forall t s l,
  run (op t) (s, l) = (tt, (s, l ++ [:: t]))
}.
```

6.2 Semantics of an Imperative Language

Here we are considering a small imperative language with a global state and a primitive to emit an event. In practice, emitting an event might consist in printing a message.

6.2.1 Operational and denotational semantics

We define the higher-order abstract syntax [23] of the language as follows:

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

We give it a small-step semantics specified with continuation in the style of Compcert [6]. We distinguish two kinds of continuations: `stop` for stopping, and `seq` (Notation: $\cdot;\cdot$) for sequencing:

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

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

```
Definition state : Type := S * continuation.

Inductive step :
  state -> option T -> state -> Prop :=
| s_ret  : forall s A a (k : A -> _),
    step (s, p_ret a ; k) None (s, k a)
| s_bind : forall s A B p f (k : B -> _),
    step (s, p_bind p f ; k) None
         (s, p ; fun a : A => f a ; k)
| s_cond_true : forall s A p1 p2 (k : A -> _),
    step (s, p_cond true p1 p2 ; k) None
         (s, p1 ; k)
| s_cond_false : forall s A p1 p2 (k : A -> _),
    step (s, p_cond false p1 p2 ; k) None
         (s, p2 ; k)
| s_get  : forall s k,
    step (s, p_get ; k) None (s, k s)
| s_put  : forall s s' k,
    step (s, p_put s' ; k) None (s', k tt)
| s_mark : forall s t k,
```

```
    step (s, p_mark t ; k) (Some t) (s, k tt).
```

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

In order to give our language a denotational semantics, we instantiate the class `stateTraceMonad` in the obvious manner. We first provide the definition of the base monad:

```
Let m : Type -> Type :=
  fun A => S * list T -> A * (S * list T).
Program Definition MONAD : monad := Monad.Pack
  (@Monad.Class m
  (fun A a => fun s => (a, s)) (* ret *)
  (fun A B m f => fun s =>
    let (a, s') := m s in f a s') (* bind *)
  _ _ _).
```

Second, we provide an implementation of the mark function for the trace monad:

```
Program Definition TRACE :=
  @MonadTrace.Mixin T MONAD
  (fun log s => (tt, (s.1, s.2 ++ [log]))).
```

Next, we provide an implementation of the state monad:

```
Program Definition STATE := @MonadState.Class
  S m (Monad.class MONAD)
  (@MonadState.Mixin S MONAD
  (* get *) (fun s => (s.1, s))
  (* put *) (fun s' s => (tt, (s', s.2)))
  _ _ _ _).
```

Last, we bundle the above monads together with an implementation of the run interface:

```
Program Definition STATETRACE :=
  @MonadStateTrace.Pack T S m
  (@MonadStateTrace.Class T S m STATE TRACE
  (@MonadStateTrace.Mixin T S
    (MonadState.Pack STATE)
     _ (* mark *)
    (fun A (m : MONAD A) (s : S * list T) => m s)
    (* run *) _ _ _ _ _)).
```

It is important to note here that the primitives `get` and `put` can only read and update the global state (of type `S`) but not the list of emitted events (of type `list T`). Only the primitive `mark` has access to the list of emitted events but it can neither read nor overwrite it: it can only add a new event `log` to the list. This is necessary to prove the correctness and completeness of the small-step semantics with respect to the denotational semantics `denotation {A : Type} (p : program A) : M A` (see *File* `smallstep_monad.v` *of [5]*).

6.2.2 The need for a small piece of deep embedding

More precisely, correctness and completeness (of the denotational semantics `denotation` w.r.t. the operational semantics `step_star`) require that we prove the two following lemmas on emitted events:

```
Lemma denotation_prefix_preserved
  A (p : program A) : forall s s' l1 l a,
  Run (denotation p) (s, l1) = (a, (s', l)) ->
  exists l2, l = l1 ++ l2.

Lemma denotation_prefix_independent
  A (p : program A) s l1 l2 :
  Run (denotation p) (s, l1 ++ l2) =
  let res := Run (denotation p) (s, l2) in
  (res.1, (res.2.1, l1 ++ res.2.2)).
```

They respectively state that: once an event is emitted it cannot be deleted; and the remaining execution of a program does not depend on the previouly emitted events. Those are natural properties that ought to be true for any monadic code, and not only the monadic code that results from the denotation of a program `p`. But this is not the case with our above instantiation of the class `stateTraceMonad`. Indeed, the class specifies those primitives that should be implemented but do not prevent one to add other primitives that might break the above properties of emitted events. This is why we restrict those properties to monadic code `(denotation p)` resulting from the denotation of a program `p`, thus allowing us to prove them by induction on the syntax. One way to overcome this limitation would be to include the syntactic restriction into the instatiation. That is to say that the previous definition of the monad functor `m` as `fun A => S * list T -> A * (S * list T)` would be extended with a program `p` and a proof that its denotation is equal to the function. That is to say it would become:

```
fun A => {
  f : S * list T -> A * (S * list T) &
  { p : program A | denotation p = f }}
```

7 Some Technical Aspects of the Formalization

7.1 Rewriting Under Function Abstractions

In pencil-and-paper proofs of monadic equational reasoning, whether a rewrite occurs under a lambda or not does not make any difference, but Coq does not natively allow for rewrites in the body of functions. Following [21], we provide a bit of automation (the tactics `Open` and `rewrite_` below) for that purpose. Let us consider for illustration a function that non-deterministically builds a subsequence of a list (*Sect. 3.1 of [11]*):

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

We want to prove the following lemma:

```
Lemma subs_cat (xs ys : seq A) :
  subs (xs ++ ys) = Do{us <- subs xs;
    Do{vs <- subs ys; Ret (us ++ vs)}}.
```

The proof eventually leads to the following subgoal:

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

We want to turn the branch

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

into

```
Do{ x0 <- subs xs;
  Do{ vs <- subs ys; Ret (x :: x0 ++ vs)}}
```

Since the target `Ret` is below `Do{ x0 <- ...; ...}`, `rewrite bindretf` fails. Instead, we "open" the continuation with `Open (X in subs xs >>= X)` to get the goal

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

on which `rewrite bindretf` now succeeds:

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

Yet, `Ret` is still out of reach of `rewrite`. We could open again the continuation but we use a "rewrite under" tactic `rewrite_ cat_cons` to get:

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

Now we can close the goal by `reflexivity` and we are done. In practice, there is little need for `Open` and most situations can be handled directly without revealing the evar using `rewrite_`. We chose to explain `Open` here because it shows how `rewrite_` is implemented.

7.2 Minor Discrepancies between Coq and Haskell

The differences between Coq and Haskell are folklore. Coq functions must terminate, so that we sometimes need an extra effort (but only standard techniques) to convince Coq that a function is really terminating. See for example the function `perms` that non-deterministically build a permutation of a list (it is not structurally terminating) [12] [5]. Also, Coq functions need to be total and some Haskell functions cannot be formalized as such (e.g., `foldr1`).

8 Related Work

8.1 About Monadic Equational Reasoning

Although enabling equational reasoning for reasoning about monadic programs seems to be a natural idea, there does not seem to be much related work. Gibbons et al. seem to be the first to synthesize monadic equational reasoning as an approach [12] [11] [1]. This viewpoint is also adopted by other authors (e.g., [22] and [25]—the latter is about equational reasoning for probabilistic programming).

8.2 Formalization of Monads in Coq

Monads are widely used for modelling programming languages with effects. For instance, [9] contains a formalization in Coq of several monads and monad transformers, each one associated with a so-called feature theorem. When monads are combined, those feature theorems can then be easily combined to prove type soundness. In comparison, the work we formalize here contains more monads but focus on equational reasoning on concrete programs instead of meta-theory on programming languages.

Monads have been used in Coq to verify low-level systems [17][18] or for their modular verification [19] based on free monads. We share the same original motivation: enable formal reasoning about low-level programs using monads.

There are more formalizations of monads in other proof-assistants. To pick one example that can be easily compared to our mechanization, one can find a formalization of the Monty Hall problem in Isabelle [8] but using a different theory called pGCL and due to McIver and Morgan.

8.3 About shallow and deep embedding

Shallow and deep embedding have been widely compared in functional programming, e.g., in [13], and combinations of shallow and deep embedding have been proposed, e.g., in [26].

8.4 About Formalization Techniques

We use packed classes [10] to formalize the hierarchy of algebraic effects. It would be possible to use other techniques. In fact, the first version of our formalization was using a combination of telescopes and canonical structures; it did not suffer problems but packed classes are more disciplined. Coq's type classes have been reported to replace canonical structures in many situations, but we have not tested them here.

This problem of rewriting under function abstraction (discussed in Sect. 7.1) is not specific to monadic equational reasoning. For example, it also occurs when dealing with the big operators of the MathComp library, a situation for which [21] provides an automated solution.

9 Conclusions and future work

We have fully formalized in the Coq proof assistant the monadic equational reasoning as advocated by Gibbons et al. Our approach is successful in the sense that our Coq proofs match surprisingly well their paper-and-pencil proofs. We have shown the applicability of the approach by instantiating the probability monad and formalizing the denotational semantics of an imperative language with the state-trace monad. The latter application has led us to extend the work by Gibbons et al. on the state monad. This has also allowed us to expose a limitation of the approach. Indeed, equations can specify what should do some primitives, but cannot prevent the existence of other primitives following different rules. We have thus proposed a solution in Sect. 6.2.2 that consists in combining the shallow embedding with a limited amount of deep embedding.

A natural direction for future work would be the formalization of [24] as it would also be the continuation of our formalization of the trace monad in Sect. 6.1.2. The formalization of more monadic equational reasoning examples [22] to improve the user experience is also underway.

Acknowledgements

We acknowledge the support of the JSPS-CNRS bilateral program "FoRmal tools for IoT sEcurity" (PRC2199) and thank all the participants of this project for fruitful discussions, as well as Cyril Cohen and Shinya Katsumata for comments about the formalization of monads.

References

[1] Abou-Saleh, F., Cheung, K.-H., and Gibbons, J.: Reasoning about Probability and Nondeterminism, *POPL workshop on Probabilistic Programming Semantics*, January 2016.

[2] Affeldt, R. and Hagiwara, M.: Formalization of Shannon's Theorems in SSReflect-Coq, *3rd Conference on Interactive Theorem Proving (ITP 2012), Princeton, New Jersey, USA, August 13–15, 2012*, Lecture Notes in Computer Science, Vol. 7406, Springer, Aug 2012, pp. 233–249.

[3] Affeldt, R., Hagiwara, M., and Sénizergues, J.: Formalization of Shannon's Theorems, *Journal of Automated Reasoning*, Vol. 53, No. 1(2014), pp. 63–103.

[4] Affeldt, R., Hagiwara, M., Senizergues, J., Garrigue, J., Sakaguchi, K., Asai, T., Saikawa, T., and Obata, N.: A Coq formalization of information theory and linear error-correcting codes, `https://github.com/affeldt-aist/infotheo`, 2018.

[5] Affeldt, R. and Nowak, D.: A Coq formalization of monadic equational reasoning, `https://github.com/affeldt-aist/monae`, 2018.

[6] Appel, A. W. and Blazy, S.: Separation logic for small-step Cminor, *Theorem Proving in Higher Order Logics, 20th Int. Conf. TPHOLs 2007*, Lecture Notes in Computer Science, Vol. 4732, Springer, 2007, pp. 5–21.

[7] Breitner, J., Spector-Zabusky, A., Li, Y., Rizkallah, C., Wiegley, J., and Weirich, S.: Ready, Set, Verify! Applying hs-to-coq to real-world Haskell code (Experience Report), *ICFP 2018*, 2018. To appear.

[8] Cock, D.: Verifying probabilistic correctness in Isabelle with pGCL, *7th Systems Software Verification, Sydney, Australia*, Nov 2012, pp. 1–10.

[9] Delaware, B., Keuchel, S., Schrijvers, T., and d. S. Oliveira, B. C.: Modular monadic metatheory, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, 2013, pp. 319–330.

[10] Garillot, F., Gonthier, G., Mahboubi, A., and Rideau, L.: Packaging Mathematical Structures, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, Berghofer, S., Nipkow, T., Urban, C., and Wenzel, M.(eds.), Lecture Notes in Computer Science, Vol. 5674, Springer, 2009, pp. 327–342.

[11] Gibbons, J.: Unifying Theories of Programming with Monads, *Unifying Theories of Programming, 4th International Symposium, UTP 2012, Paris, France, August 27-28, 2012, Revised Selected Papers*, Wolff, B., Gaudel, M., and Feliachi, A.(eds.), Lecture Notes in Computer Science, Vol. 7681, Springer, 2012, pp. 23–67.

[12] Gibbons, J. and Hinze, R.: Just do it: simple monadic equational reasoning, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Chakravarty, M. M. T., Hu, Z., and Danvy, O.(eds.), ACM, 2011, pp. 2–14.

[13] Gibbons, J. and Wu, N.: Folding domain-specific languages: deep and shallow embeddings (functional Pearl), *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Jeuring, J. and Chakravarty, M. M. T.(eds.), ACM, 2014, pp. 339–347.

[14] Gonthier, G., Mahboubi, A., and Tassi, E.: A Small Scale Reflection Extension for the Coq system, Technical report, INRIA, 2008. Version 17 (Nov 2016).

[15] Gonthier, G. and Tassi, E.: A Language of Patterns for Subterm Selection, *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, Beringer, L. and Felty, A. P.(eds.), Lecture Notes in Computer Science, Vol. 7406, Springer, 2012, pp. 361–376.

[16] Greenaway, D.: *Automated Proof-Producing Abstraction of C Code*, PhD Thesis, University of New South Wales, Sydney, Australia, Jan 2015.

[17] Jomaa, N., Nowak, D., Grimaud, G., and Hym, S.: Formal proof of dynamic memory isolation based on MMU, *Science of Computer Programming*, Vol. 162(2018), pp. 76–92.

[18] Jomaa, N., Torrini, P., Nowak, D., Grimaud, G., and Hym, S.: Proof-Oriented Design of a Separation Kernel with Minimal Trusted Computing Base, *18th International Workshop on Automated Verification of Critical Systems (AVOCS 2018), Jul 2018, Oxford, United Kingdom. Electronic Communications of the EASST Open Access Journal*, 2018.

[19] Letan, T., Régis-Gianas, Y., Chifflier, P., and Hiet, G.: Modular Verification of Programs with Effects and Effet Handlers in Coq, *22nd international symposium on formal methods (FM 2018), Oxford, United Kingdom*, Jul 2018.

[20] Mahboubi, A. and Tassi, E.: Canonical Structures for the Working Coq User, *4th International Conference on Interactive Theorem Proving (ITP 2013), Rennes, France, July 22–26, 2013*, Lecture Notes in Computer Science, Vol. 7998, Springer, 2013, pp. 19–34.

[21] Martin-Dorel, E.: ssr-under-tac, `https://github.com/erikmd/ssr-under-tac`, 2016.

[22] Mu, S.-C.: Functional Pearls, Reasoning and Derivation of Monadic Programs, A Case Study of Non-determinism and State, Jul 2017. Submitted for publication. Available at `http://flolac.iis.sinica.edu.tw/flolac18/files/test.pdf` (last access: 2018/07/29).

[23] Pfenning, F. and Elliott, C.: Higher-Order

Abstract Syntax, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, Wexelblat, R. L.(ed.), ACM, 1988, pp. 199–208.

[24] Piróg, M. and Gibbons, J.: Tracing monadic computations and representing effects, *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia, 25 March 2012.*, Chapman, J. and Levy, P. B.(eds.), EPTCS, Vol. 76, 2012, pp. 90–111.

[25] Shan, C.-C.: Equational reasoning for probabilistic programming, *POPL 2018 TutorialFest*, Jan 2018.

[26] Svenningsson, J. and Axelsson, E.: Combining deep and shallow embedding of domain-specific languages, *Computer Languages, Systems & Structures*, Vol. 44(2015), pp. 143–165.