Vers la formalisation en Coq des transformateurs de monades modulaires

Célestine Sauvage[1], Reynald Affeldt[2], and David Nowak[1]

[1] CRIStAL *, Lille, France
[2] National Institute of Advanced Industrial Science and Technology (AIST), Tsukuba, Japan

Résumé

Nous étudions la vérification formelle de programmes avec effets de bord en utilisant un langage purement fonctionnel. Dans le cadre de cette étude, nous avons développé Monae, une librairie Coq qui propose une formalisation des monades et de leurs lois algébriques. Les preuves se font par raisonnement équationnel en utilisant les capacités de réécriture de Coq. Les programmes n'utilisent généralement pas un seul type d'effet de bord, mais une combinaison de plusieurs d'entre eux. On utilise les transformateurs de monades dans ce but. Cependant, l'approche traditionnelle pour le *lifting* des primitives n'est pas modulaire. Il est intéressant de définir de manière canonique les opérations algébriques des monades et leurs primitives `lift`. Dans cet article, nous présentons l'implémentation des transformateurs de monades modulaires et les preuves des théorèmes qui en découlent en Coq. Nous montrons également leurs utilisations comparées aux transformateurs de monades classiques.

1 Introduction

La programmation fonctionnelle utilise des monades pour représenter les différents types d'effets de bord existant dans les programmes impératifs, tels que les états, les exceptions ou les continuations. Chaque monade s'accompagne d'opérations qui lui sont propres et ces opérations doivent respecter certaines conditions de cohérence. Les programmes utilisant généralement plusieurs sortes d'effets de bord, on voudrait pouvoir en combiner les monades correspondantes. Cependant, obtenir une monade qui modélise cette combinaison de plusieurs effets peut être difficile. Certaines méthodes ont vu le jour pour pallier ce problème.

Dans un premier temps, une manière de composer des monades a été étudiée dans Monae suivant l'idée de Moggi [6]. Mais cela implique de devoir réécrire un modèle de monade pour chaque combinaison [1]. Une autre approche plus modulaire a ensuite vu le jour, les *transformateurs de monades* [5]. Un transformateur de monades ajoute, pour toute monade, un nouvel effet tout en assurant que l'effet initial soit toujours effectif à l'aide de la primitive `lift`.

Malgré tout, le *lifting* des opérations de la monade sous-jacente a besoin d'être spécifié de manière ad hoc, pour chaque transformateur de monades, ce qui n'est pas modulaire. De plus, le nombre de *lifting* des primitives augmente avec le nombre de monades combinées. Pour améliorer cette modularité, nous formalisons en Coq les transformateurs de monades modulaires [4].

Dans cet article, nous décrivons l'état actuel de cette formalisation [1]. Nous présenterons dans un premier temps Monae et son fontionnement (section 2), puis nous montrerons comment nous avons ajouté les transformateurs de monades dans Monae (section 3). Nous terminerons enfin en nous focalisant sur le *lifting* modulaire des opérations liées aux monades (section 4).

*Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France

1. Le code et les preuves décrits dans ce papier sont publics et disponibles à l'adresse suivante : https://github.com/affeldt-aist/monae.

2 Préliminaires techniques

Monae est une librairie qui formalise une hiérarchie de monades et leurs effets [1]. Dans cette hiérarchie, les monades sont des endo-foncteurs dans une catégorie particulière : les objets sont les types qui ont le type `Type` de Coq et les morphismes sont les fonctions de type `Type -> Type`. La base de cette hiérarchie est le type `functor`. Étant donné un foncteur `F`, on note `F A` l'application de `F` au type `A` et `F # g` l'application de `F` à la fonction `g`. Les monades sont représentées par un type `monad` qui étend le type `functor` en l'équipant avec deux transformations naturelles (`Ret` et `Join`) et les lois de monade usuelles. On définit aussi l'opération `Bind m f` (notée $m \gg\!= f$ dans les exemples de code) à partir de `Ret` et `Join`. On peut ainsi définir les monades à partir de `Ret` et `Bind` également. La hiérarchie de Monae est construite en étendant le type `monad` avec des interfaces qui définissent les effets sous la forme d'opérations et de lois équationnelles comme préconisé par Gibbons et Hinze [3]. Les interfaces peuvent être combinées, et les modèles sont construits de manière ad hoc dans un deuxième temps.

Exemple : La monade d'état

Pour illustrer cette construction, nous présenterons ici la monade d'état définie dans Monae [2]. Les monades sont formalisées en utilisant la technique des *packed classes* [2]. Dans un premier temps, on définit une interface pour la monade d'état (`mixin_of`) qui étend la classe `monad` (`class_of`). Le résultat est empaqueté avec une fonction de type `Type -> Type` qui représente l'action sur les objets de la monade :

```
Record mixin_of S (M : monad) : Type := Mixin {
  (* Operations *)
  get : M S ;
  put : S -> M unit ;
  (* Équations *)
  _ : forall s s', put s >> put s' = put s' ;
  _ : forall s, put s >> get = put s >> Ret s ;
  _ : get >>= put = skip ;
  _ : forall A (k : S -> S -> M A),
      get >>= (fun s => get >>= k s) = get >>= fun s => k s s }.
Record class_of S (m : Type -> Type) := Class {
  base : Monad.class_of m ; mixin : mixin_of S (Monad.Pack base) }.
Structure t S : Type := Pack { m : Type -> Type ; class : class_of S m }.
```

La cohérence des propriétés est vérifiée dans un deuxième temps en implémentant le modèle classique pour la monade d'état (voir section 3.2 pour plus de détails sur la validation d'un modèle dans Monae).

3 Étendre Monae avec les transformateurs de monades

Comme expliqué précédemment, un transformateur de monades peut être utilisé pour composer des effets de manière modulaire, produisant une couche d'effets monadiques élémentaires. Un transformateur de monades est un constructeur de type `T` prenant en argument une monade `M`, de telle sorte que `T M` soit elle aussi une monade. Pour pouvoir faire le *lifting* de l'effet

2. fichier `state_monad.v`

monadique de M, le transformateur de monades est équipé d'une primitive, lift, qui satisfait les lois suivantes :

$$\begin{aligned} \texttt{lift} \circ \texttt{Ret}_{\texttt{M}} &= \texttt{Ret}_{\texttt{T M}} \\ \texttt{lift}(m \gg\!=_{\texttt{M}} k) &= \texttt{lift}(m) \gg\!=_{\texttt{T M}} (\texttt{lift} \circ k) \end{aligned}$$

Comme les opérations peuvent se combiner de différentes manières suivant le transformateur de monades au-dessus, il est nécessaire d'exprimer manuellement le *lifting* de ces opérations à travers le transformateur. On définit les transformateurs de monades dans la section 3.1 qu'on illustre avec l'exemple du transformateur de monades d'erreur dans la section 3.2.

3.1 Morphisme de monades et transformateur de monades

On formalise[3] les transformateurs de monades en suivant la présentation de Jaskelioff [4] qui définit d'abord les morphismes de monades.

Soient deux monades M and N. Un *morphisme de monades* est une fonction e de type forall A, M A -> N A (on écrit M ~~> N par la suite) telle que, pour tout type A, Ret = $\texttt{e}_{\texttt{A}} \circ$ Ret et, pour tout types A et B, et m et f de types appropriés, $\texttt{e}_{\texttt{B}}(\texttt{m} \gg\!= \texttt{f}) = \texttt{e}_{\texttt{A}}\texttt{m} \gg\!= (\texttt{e}_{\texttt{B}} \circ \texttt{f})$:

```
(* dans le Module monadM *)
Record class_of (e : M ~~> N) := Class {
  _ : forall A, Ret = e A \o Ret;
  _ : forall A B (m : M A) (f : A -> M B),
    e B (m >>= f) = e A m >>= (e B \o f) }.
Structure t := Pack { e : M ~~> N ; class : class_of e }.
```

Un *transformateur de monades* est une fonction T de type monad -> monad (pour toute monade M, T M est donc une monade) munie d'une opération liftT telle que liftT M est un morphisme de monades :

```
(* dans le Module MonadT *)
Record class_of (T : monad -> monad) := Class {
  liftT : forall M : monad, monadM M (T M) }.
Record t := Pack {m : monad -> monad ; class : class_of m}.
```

3.2 Exemple : le transformateur de monades d'erreur

Le transformateur de monades d'erreur[4] ajoute à une monade M existante un effet qui permet à un programme d'échouer et de retourner une erreur de type Z. Pour définir ce transformateur de monades, on va devoir fournir :

1. une fonction de type monad -> monad ;
2. un morphisme de monades pour définir liftX.

Considérons le type de l'erreur Z et une monade M comme donnés. L'action sur les objets de la nouvelle monade est :

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

3. fichier monad_transformer.v
4. fichier monad_transformer.v

On montre que l'action sur les morphismes `MX_fmap` satisfait les lois des foncteurs ce qui nous permet de construire un foncteur `MX_functor` (`fmap f` est une notation pour `_ # f` où le foncteur est inféré automatiquement) :

```
Definition MX_map A B (f : A -> B) (m : MX A) : MX B :=
  fmap (fun x => match x with inl y => inl y | inr y => inr (f y) end) m.
Definition MX_functor : functor. (* lois des foncteurs omises *)
```

Les définitions de `retX` et `bindX` sont celles usuelles de la définition du transformateur de monades d'erreur, et où le `Ret` et le `Bind` (`>>=`) utilisés sont ceux de la monade `M` :

```
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.
```

On prouve que `retX` est une transformation naturelle du foncteur identité `FId` vers le foncteur `MX_functor` (construction `retX_natural` omise), ce qui nous permet de construire une nouvelle monade en utilisant le constructeur `Monad_of_ret_bind` de Monae :

```
Definition eErrorMonadM : monad :=
  @Monad_of_ret_bind MX_functor retX_natural bindX _ _ _ (* lois omises *).
```

Il ne nous reste plus qu'à fournir la fonction `liftX` qui transforme toute computation `M X` en une computation `eErrorMonadM X` :

```
Definition liftX X (m : M X) : eErrorMonadM X :=
       @Bind M _ _ m (fun x => @Ret eexceptionMonadM _ x).
```

Comme cette dernière satisfait les lois de morphismes de monades (construction `ErrorMonadM` omise), on obtient finalement un transformateur de monades

```
Definition errorMonadM : monadM M eErrorMonadM :=
  monadM.Pack (@monadM.Class _ _ liftX _ _).
```

4 Opérations et *lifting*

On montre dans cette section que la formalisation de transformateur de monades de la section précédente permet de retrouver la preuve du premier théorème de [4] au moyen d'une preuve succincte à base de réécriture.

4.1 Opérations et opérations algébriques

Comme évoqué au-dessus, en plus des opérations de base `Ret` et `Bind`, on associe aux monades des opérations pour représenter des effets. Pour parler de *lifting* des opérations par morphisme de monades, Jaskelioff [4] distingue les *(E,M)-opérations* et les *opérations algébriques* (les opérations usuelles peuvent être retrouvées à partir des (E,M)-opérations).

Étant donnés un foncteur `E` et une monade `M`, on appelle **(E,M)-opération** une transformation naturelle de `E \O M` vers `M` (où `\O` est la composition de foncteurs).

Une (E,M)-opération `op` est algébrique quand, pour tout `A`, `B`, `t` et `f`, on a

```
(op A t >>= f) = op B ((E # (fun m => m >>= f)) t).
```

Soient un morphisme de monades `e` de `M` vers `N`, un foncteur `E` et une (E,M)-operation `op`. Un *lifting* de `op` vers `N` le long de `e` est une (E,N)-operation `op'` telle que :

```
forall X, e X \o op X = op' X \o (E # (e X)).
```

4.2 Exemple : Opération algébrique put

Soit S un type pour un état. D'après [4, section 3], on définit [5] le foncteur put_fun tel que :

```
(* Foncteur pour l'opération put (action sur les objets) *)
Definition put_acto X := (S * X).
(* Map associée au foncteur (action sur les morphismes) *)
Definition put_actm X Y (f : X -> Y) (sx : put_acto X) : put_acto Y :=
 (sx.1, f sx.2).
(* Pack dans la classe Functor *)
Program Definition put_fun :=
  Functor.Pack (@Functor.Class put_acto put_actm _ _).
(* Preuves des lois de foncteurs omises *)
```

On vérifie ensuite que l'on a bien une transformation naturelle de put_fun S \O MS S vers MS S (où MS est le transformateur de monades d'état) :

```
Definition n_put S A (s : S) (m : MS S A) : MS S A :=
  fun _ => m s.
Lemma naturality_put S : naturality (put_fun S \O MS S) (MS S)
  (fun A => uncurry (n_put (A:=A))).
```

On montre que l'on obtient bien une opération algébrique suivant la définition donnée dans la section 4.1 :

```
Definition put_op S : operation (put_fun S) (ModelMonad.State.t S) :=
  Natural.Pack (@naturality_put S).
Lemma algebraic_put S : algebraicity (put_op S). Proof. ... Qed.
Program Definition put_aop S : aoperation (put_fun S) (MS S) :=
  AOperation.Pack (AOperation.Mixin (@algebraic_put S)).
```

L'opérateur put peut être défini à partir de l'(E,M)-opération put_op et a la même sémantique que l'opération usuelle de la littérature :

```
Definition put : S -> MS S unit := fun s => put_op _ (s, Ret tt).
Lemma putE : put = fun s' _ => (tt, s'). Proof. by []. Qed.
```

4.3 *Lifting* des opérations algébriques

Le premier théorème que Jaskelioff prouve à propos du *lifting* des opérations montre que les opérations algébriques sont transformées en opérations algébriques par morphisme de monades.

Theorem 1 (*Lifting* algébrique)**.** *Soit la (E,M)-opération* `op`*. La (E,N)-opération définie par*

```
fun X => Join \o e (N X) \o phi op (N X)
```

où `phi op` *est*

```
fun X => op X \o (E # Ret)
```

est algébrique [4, Théorème 19].

5. fichier monad_model.v

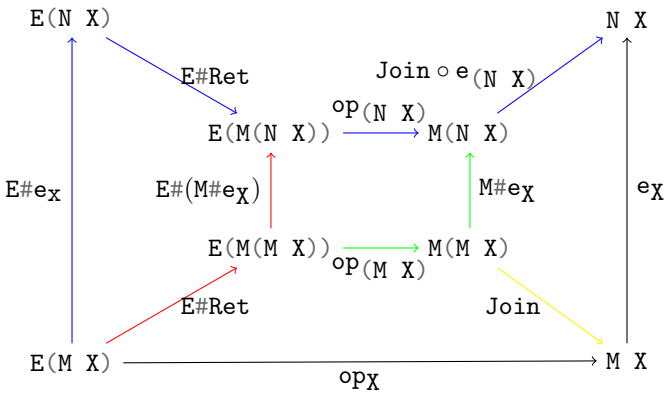

FIGURE 1 – Preuve du théorème [4, Théorème 19]

Démonstration. [6] Par définition, le but original est :

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

Le membre gauche correspond au chemin noir de la figure 1 et le membre de droite correspond au chemin bleu. La première étape est de montrer que la première partie du chemin bleu commute avec le chemin rouge, c'est-à-dire :

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

La preuve consiste en une série de réécritures. On utilise d'abord la loi de compositions du foncteur `E` dans les deux membres de l'équation, puis la naturalité de `Ret` de la monade `M`, et enfin une propriété du foncteur identité :

```
rewrite -[in LHS]compE -functor_o.
(* (E # (Ret \o e X)) Y = (E # (M # e X)) ((E # Ret) Y) *)
rewrite -[in RHS]compE -functor_o.
(* (E # (Ret \o e X)) Y = (E # (M # e X \o Ret)) Y *)
rewrite (natural RET).
(* (E # (Ret \o e X)) Y = (E # (Ret \o FId # e X)) Y *)
by rewrite FIdf.
```

Ensuite, on montre qu'une partie du chemin rouge-bleu commute avec le chemin vert :

```
op (N X) (E # (M # e X)) = (M # e X) (op (M X)).
```

Puis on montre que le chemin vert-bleu commute avec le chemin vert-jaune-noir :

```
e X (Join (op (M X))) = Join (e (N X) ((M # e X) (op (M X)))).
```

Toutes les étapes de preuve jusqu'à présent sont affaire de réécriture des lois de foncteurs et de monades. La dernière étape est l'égalité entre le chemin rouge-vert-jaune et `op X Y` et repose sur un lemme intermédiaire [4, Proposition 17]. □

6. fichier `monad_transformer.v`

4.4 Exemple : Lifting des opérateurs `get` et `put`

On voudrait écrire un programme qui utilise à la fois la monade option (un cas particulier de la monade d'erreur avec `unit` le type de retour de l'erreur) et la monade d'état. On comparera le *lifting* classique des (`E`,`M`)-opérations algébriques de la monade d'état et le *lifting* algébrique de ces mêmes opérations.

4.4.1 *Lifting* classique

Il faut préciser la manière de faire le *lifting* des deux opérations dans le transformateur de monades d'option :

```
Let M := stateMonad.
Let erZ : monadT := errorT unit.

Definition getOpt : erZ (M S) := liftX get.
Definition putOpt : erZ (M S) := fun s' => liftX (put s').
```

On peut maintenant écrire un programme mélant les effets de la monade option et de la monade d'état :

```
Let incr := getOpt >>= (fun i => putOpt (i.+1)).
Let prog := incr >> Fail >> incr.
```

`Fail` étant une opération de la monade d'erreur.

4.4.2 *Lifting* algébrique

Suivant la définition donnée pour le *lifting* algébrique (`alifting`) de la section 4.3, Théorème 1, on obtient les (`E`,`N`)-opérations suivantes (où `N` correspond à une monade résultant de l'application du transformateur de monades d'erreur) :

```
Let lift_getX S : (get_fun S) \O (erZ (M S)) ~~> (erZ (M S)) :=
  alifting (get_aop S) (LiftT erZ (M S)).
Let lift_putX S : (put_fun S) \O (erZ (M S)) ~~> (erZ (M S)) :=
  alifting (put_aop S) (LiftT erZ (M S)).
(* LiftT erZ (M S) correspondant ici à liftX *)
```

On peut à présent écrire des programmes avec les (`E`,`N`)-opérations (en respectant l'utilisation de ces opérations décrites dans [4, section 3] pour que leurs sémantiques soient identiques au `get` et `put` usuels).

```
Let incr := (lift_getX Ret) >>= (fun i => lift_putX (i.+1, Ret tt)).
Let prog := incr >> Fail >> incr.
```

On peut comparer que le *lifting* algébrique et le *lifting* classique ont le même comportement :

```
Goal lift_getX Ret = @liftX  _ _ _ get.
Proof. by rewrite /lift_getX aliftingE. Qed.

Goal (fun s' => (lift_putX (s', Ret tt))) =
     fun s => (@liftX _ _ _ (put s)).
Proof. by rewrite/lift_putX aliftingE. Qed.
```

5 Conclusion et travaux futurs

Dans cet article, nous avons donné un aperçu de Monae, une formalisation en Coq d'une hiérarchie de monades pour le raisonnement équationnel [1] et expliqué comment l'étendre avec des transformateurs de monades. Pour ce faire, nous avons formalisé une proposition existante qui insiste sur les aspects modulaires [4]. Monae semble se prêter naturellement à cette extension : nous avons pu reproduire les définitions (à propos des effets pour les états, les exceptions, mais aussi les continuations) et les premières preuves de Jaskelioff en utilisant la réécriture (voir en particulier section 4.3). Nous avons pu développer quelques exemples sur différents transformateurs de monades et leurs opérations algébriques.

Nous comptons maintenant terminer l'expérience de formalisation de la proposition de Jaskelioff, qui définit aussi le *lifting* d'opérations non-nécessairement algébriques [4, section 5], et utiliser les transformateurs de monades pour le raisonnement équationnel à la Gibbons et Hinze [3]. Dans ce dernier cas, il nous faudra généraliser les interfaces des monades pour y accommoder les transformateurs de monades.

Références

[1] 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*. Springer, 2019.

[2] 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.

[3] 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.

[4] 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.

[5] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1995)*, pages 333–343. ACM, 1995.

[6] Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1) :55–92, 1991.