# Projet Typage et Polymorphisme : Etude de l'article *Lightweight Monadic Programming in ML* *(Swamy, Guts, Leijen, Hicks)*

Aurélien Deharbe, Jérémie Salvucci

3 novembre 2011

## Table des matières

**1 Introduction** **2**

**2 Les monades** **2**
- 2.1 Théorie . . . 2
- 2.2 Pratique . . . 3

**3 Transformations de programmes** **5**
- 3.1 Transformations par la syntaxe . . . 5
- 3.2 Transformations dirigées par les types . . . 6
  - 3.2.1 Morphismes et contraintes . . . 6
  - 3.2.2 Règles de typage . . . 6
  - 3.2.3 Compatibilité avec ML . . . 7
  - 3.2.4 L'algorithme . . . 7

**4 L'implémentation proposée : Coco** **8**

**5 Conclusion et perspectives** **9**

# 1 Introduction

Il s'agit dans ce rapport de présenter l'étude d'un article concernant l'intégration des monades dans un langage ML [SGLH11]. Nous rappellerons donc dans une première partie les principes et le fonctionnement des monades, puis nous introduirons la problématique avant d'exposer les solutions proposées par les auteurs, basées sur la résolution de contraintes de types. Nous terminerons cette étude par la présentation d'une implémentation, puis par les perspectives que cette solution met en avant.

# 2 Les monades

Les monades trouvent leur origine dans la théorie des catégories, et ont été introduites en informatique par les travaux d'Eugenio Moggi. Néanmoins, elles ne seront réellement utilisées qu'après que Philip Wadler les popularisent. A l'origine, les monades avaient pour but de modulariser la sémantique dénotationnelle. Aujourd'hui certains langages fonctionnels adoptent le style monadique pour structurer les comportements d'un programme. Nous en verrons une approche d'abord plus théorique avant d'observer leur utilisation en pratique.

## 2.1 Théorie

Formellement, on peut voir une monade comme un triplet $(M, unit, bind)$ :

- un constructeur de type monadique $\alpha\ M$,
- une fonction qui construit un objet de type monadique à partir d'un élément du type sous-jacent, appelée $unit$ ou $return$ (de signature $unit : \forall\alpha.\alpha \rightarrow \alpha\ M$),
- une fonction associant à un type monadique et une fonction d'association un autre type monadique (de signature $bind : \forall\alpha,\beta.\alpha\ M \rightarrow (\alpha \rightarrow \beta\ M) \rightarrow \beta\ M$).

On peut voir la monade comme une structure complexe encapsulant un objet de type existant dans un objet portant plus d'information. Nous pouvons alors encapsuler les effets ne dépendant pas de l'algorithme. Citons par exemple l'encapsulation des effets de bord en Haskell dans la monade IO.

Pour garantir la composition de fonctions, une monade doit obéir aux lois suivantes [Wad92] :

- composition neutre par $unit$ à gauche : $bind\ (unit\ a)\ f \equiv f\ d$,
- composition neutre par $unit$ à droite : $bind\ m\ unit \equiv m$,

– associativité :
$bind\ (bind\ m\ f)\ g \equiv bind\ m\ (\lambda x\ \rightarrow\ (bind\ (f\ x)\ g))$.

### 2.2 Pratique

On se propose d'étudier les différents mécanismes syntaxiques, pour les monades, mis en place dans différents langages de programmation fonctionnelle, comme Haskell et OCaml.

Nous développerons les points suivants à partir d'un exemple concret. Il nous permettra de distinguer les avantages et les inconvénients de chacune des méthodes, puis d'illustrer la transformation de programme apportée par l'article. On considère le tirage de trois cartes dans le cadre d'une partie de blackjack. On introduit les monades Simulation et Maybe, permettant d'encapsuler respectivement les tirages aléatoires et la possibilité de traitement des erreurs (nous noterons par la suite $unitS$ et $bindS$ les fonctions de la monade Simulation, et $unitM$ et $bindM$ les fonctions de la monade Maybe afin d'éviter toute confusion).

```
module Simulation = struct
    type state = int
    type α mon = state → α * state

    let unit x = fun s → (x, s)
    let bind mon f =
        fun s → let (x', s') = mon s in f x' s'
    let run seed mon = fst (mon seed)

    let next_state s = s * 25173 + 1725

    let rand n =
        fun s → ((abs s) mod n, next_state s)

    let choose p a b =
        fun s →
        if float (abs s) ≤ p *. float max_int
        then a (next_state s)
        else b (next_state s)
end
```

```
module Maybe = struct
    type α mon = Nothing | Just of α

    let unit x = Just x
    let fail () = Nothing
    let bind mon f = match mon with
        | Nothing -> Nothing
        | Just x -> f x
end
```

**Les monades en Haskell** Haskell supporte nativement la manipulation des monades, grâce à son système de type et à la notation *do*. Notre exemple se code alors :

```
do
    c1 ← randomRIO (1, 10)
    c2 ← randomRIO (1, 10)
    c3 ← randomRIO (1, 10)
    return (c1 + c2 + c3)
```

**Les monades en OCaml avec pa_monad** Le support des monades en OCaml est assuré par l'extension syntaxique pa_monad, dont le fonction-

nement s'appuie sur le préprocesseur OCamlP4. Il en résulte une notation proche de celle d'Haskell :

```
perform
    c1 ← rand 10
    c2 ← rand 10
    c3 ← rand 10
    return (c1+1 + c2+1 + c3+1)
```

**Les monades en OCaml avec bind et unit explicites** Lorsque l'on ne souhaite pas utiliser d'extension du langage OCaml, il reste la possiblité d'exploiter les modules de monades définis précédemment pour lier explicitement les opérations monadiques :

```
bindS (rand 10) (fun c1 →
    bindS (rand 10) (fun c2 →
        bindS (rand 10) (fun c3 →
            unitS (1+c1 + 1+c2 + 1+c3)))) ; ;
```

**Les monades en OCaml avant transformation** Dans un souci de lisibilité du code du programme, les auteurs ont cherché à intégrer un style monadique dans le langage OCaml, par le biais d'une transformation de programme source à source. Nous souhaiterions alors pouvoir écrire :

```
let c1 = rand 10 in
let c2 = rand 10 in
let c3 = rand 10 in
1+c1 + 1+c2 + 1+c3 ; ;
```

Cependant, ce code comporte des erreurs de typage. Les variables $c1$, $c2$ et $c3$ sont de type $Simulation\ int$, et ne peuvent donc pas être sommées directement avec l'opérateur d'addition

```
val (+) : int → int → int
```

Il faut alors transformer le programme pour que les opérations s'effectuent sur les types monadiques.

# 3 Transformations de programmes

En vue d'une éventuelle intégration dans la distribution d'OCaml, une transformation envisageable est source à source (à cause du système de types).

## 3.1 Transformations par la syntaxe

Les langages fonctionnels étant basés sur le $\lambda$-calcul, on ne considérera que les règles de transformations liées à celui-ci.

| | |
|---|---|
| **Constante** : | $[\![N]\!] = \text{unit } N$ |
| **Variable** : | $[\![x]\!] = \text{unit } x$ |
| **Abstraction** : | $[\![\lambda x.a]\!] = \text{unit } (\lambda x \rightarrow [\![a]\!])$ |
| **Application** : | $[\![a\ b]\!] = \text{bind } [\![a]\!]\ (\lambda v_a \rightarrow \text{bind } [\![b]\!]\ (\lambda v_b \rightarrow v_a\ v_b))$ |

On présente ici un exemple simple, basé sur le tirage d'une valeur d'une unique carte de notre exemple précédent :

$[\![$ **let** card = rand 10 **in** card + 1 $]\!] =$ bind (bind (unit rand) ($\lambda v_1 \rightarrow$
bind (unit 10) ($\lambda v_2 \rightarrow$
$v_1 v_2$)))
($\lambda v_3 \rightarrow$ bind (unit $v_3$) ($\lambda v_4 \rightarrow$
bind (unit 1) ($\lambda v_5 \rightarrow$
unit ($v_4 + v_5$))))

On peut ensuite appliquer un certain nombre de réductions, grâce à la première loi monadique (composition neutre par *unit* à gauche) :

$[\![$ **let** card = rand 10 **in** card + 1 $]\!] =$ bind (rand 10) ($\lambda v \rightarrow (v + 1)$)

Cependant, cette méthode de transformation du programme comporte une limitation importante : on ne considère qu'un type de monade. En effet, la transformation présentée ci-dessus ne tient compte que d'une fonctionnalité : la génération aléatoire. Qu'en est-il lorsque nous souhaiterons ajouter la possibilité de dépasser le score de 21, à l'aide de la monade *Maybe* ?

Dans un programme réaliste, on voudra certainement effectuer plusieurs traitements différents (affecter des variables, déclencher des exceptions, manipuler des listes, générer des valeurs aléatoires, ...), au sein de différentes monades. Malheureusement, sans moyen pour détecter à quelle monade les *unit* et *bind* à ajouter appartiennent, la transformation par la syntaxe seule ne permet pas l'utilisation de plusieurs monades dans un programme.

Les auteurs envisagent dans l'article étudié une transformation source à source dirigée par les types.

### 3.2 Transformations dirigées par les types

Le système de types ciblé est System F. La transformation consiste en la réécriture du programme en inférant les types et en insérant les différents $unit$, $bind$ et morphismes correspondants.

Le but est d'inférer les types les plus généraux. Pour cela on repose sur les types qualifiés. Les prédicats sont composés d'ordres sur les monades (morphismes).

#### 3.2.1 Morphismes et contraintes

L'objectif est d'inférer les types les plus généraux possibles tout en utilisant les monades les plus simples (à morphisme près). Pour cela, les auteurs infèrent des types qualifiés de la forme $\forall \bar{\nu}.P \Rightarrow \tau$, où $\bar{\nu}$ est un vecteur de variables de types et $P$ un ensemble de contraintes. Les contraintes sont écrites sous la forme $m_1 \triangleright m_2$ ($m_1$ peut être *lifté* en $m_2$) et représentent les possibilités de morphismes entre monades.

Dans notre exemple, il s'agira d'introduire une monade *SimMaybe* (Monad ($SimMaybe$, $bindSM$, $unitSM$)) qui encapsulera à la fois la génération aléatoire et le caractère optionnel. Les contraintes seraient alors :

$$P = \begin{cases} Simulation & \triangleright\ SimMaybe \\ Maybe & \triangleright\ SimMaybe \end{cases}$$

La résolution de contraintes peut lever plusieurs réécritures différentes d'un même programme. Par exemple, le code suivant repose sur les classes de types d'Haskell et lève une ambigüité :

```
f :: (Read a, Show a) ⇒ String → String
f = show . read
```

bien que $a$ n'intervienne pas dans la sémantique du corps de la fonction. Ce comportement n'est pas souhaitable pour cette transformation, les auteurs ont donc choisi de créer leur propre algorithme garantissant la cohérence (équivalence sémantique) de l'ensemble des réécritures possibles. Ainsi, ils choisissent d'insérer les morphismes le plus tard possible et d'utiliser les monades les plus simples (plus petite borne supérieure dans le semi-treillis formé par les morphismes entre monades).

#### 3.2.2 Règles de typage

On se propose dans cette section de choisir une règle de typage ; on observera la résolution des contraintes dans le cas d'une application.

$$\frac{P \mid \Gamma \vdash e_1 : m_1\ (\tau_2 \rightarrow m_3\ \tau) \qquad P \mid \Gamma \vdash e_2 : m_2\ \tau_2 \qquad \forall i. P \models m_i \triangleright m}{P \mid \Gamma \vdash e_1\ e_2 : m\ \tau} \quad (\text{TI-APP})$$

$e_1$ est l'abstraction (*bind*) appartenant à la monade $m_1$. $e_2$ est une valeur de type $\tau_2$ (compatible avec le paramètre de l'abstraction $e_1$) encapsulée dans une monade $m_2$. On cherche la plus petite borne supérieure $m$ de $\{m_1, m_2, m_3\}$ car on souhaite introduire les monades les plus générales le plus tard possible. Après application des morphismes aux monades, on peut inférer le type de l'application qui est de type $m$.

### 3.2.3 Compatibilité avec ML

Le projet assure une compatibilité avec le langage ML. Pour cela, il accepte l'ensemble des programmes acceptés par les règles du système de types Hindley-Milner. Ils en donnent également une preuve par induction sur les règles de dérivation de types.

### 3.2.4 L'algorithme

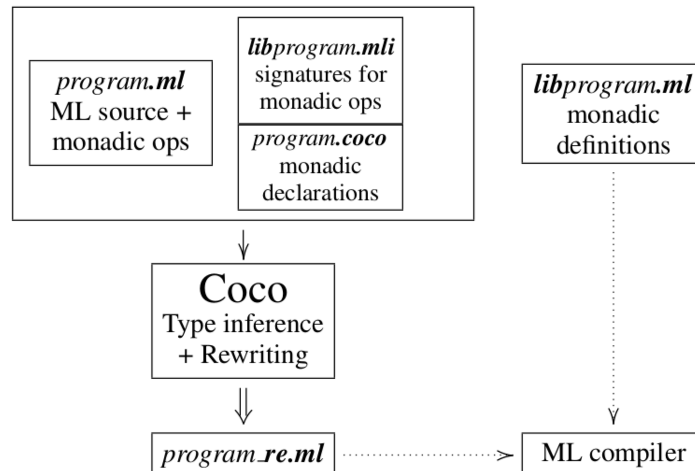
Les auteurs prouvent qu'il est possible de réencoder leur système vers OML [FFA99] en profitant des propriétés connues du système de types d'Haskell. Pour les raisons précédemment décrites, cette solution n'est pas retenue. Le système de types d'Haskell ne tenant pas compte des lois de morphismes, de nombreux programmes seraient rejetés (ambigüité). Ils ont donc écrit un algorithme reposant sur l'algorithme W en prenant en compte ces lois. La preuve de cohérence de cet algorithme permet de justifier leur choix de réécriture.

L'algorithme consiste à éliminer l'ensemble des contraintes en insérant les morphismes en garantissant la propriété de plus petite borne supérieure. Si l'ensemble des contraintes est résolu alors la réécriture est possible et la sémantique du programme réécrit est garantie. Sinon, une erreur est levée et la réécriture est impossible.

# 4 L'implémentation proposée : Coco

Dans le cadre de l'article, une implémentation de l'algorithme d'inférence de types a été développée [SGLH] par les auteurs. Le programme, codé en OCaml et suivant la stratégie de base d'insertion de morphismes, a permis de vérifier la bonne marche de la transformation source à source pour, au minimum, tous les exemples du papier. Les programmes réécrits ont ensuite été typés à l'aide de Flow-Caml [1], afin de tester leur validité.

Le fonctionnement global de l'outil est le suivant : nous apportons en entrée du programme le fichier source *ml* à réécrire, ainsi que les fichiers contenants les signatures des monades et les déclarations de ces monades (ainsi que les relations de morphismes entre elles). En sortie le programme fournira le code transformé, qui doit par la suite être compilé en utilisant les définitions de monades à apporter.




Bien qu'encore au stade de développement (Coco ne supporte qu'un sous-ensemble du langage OCaml en entrée), l'outil se révèle apporter plusieurs options utiles ainsi que plusieurs modes de résolution.

1. Flow-Caml est l'une des premières implémentations utilisables d'un langage équipé d'un système de types prenant en compte à la fois le sous-typage, le polymorphisme à la ML et une inférence de type complète.

## 5 Conclusion et perspectives

L'article étudié présente une utilisation intéressante des systèmes d'inférence de types : on retiendra que le système de types peut permettre d'encoder des caractéristiques syntaxiques à un langage fonctionnel.

Toutefois, en l'état actuel, le compilateur Coco ne prend pas en charge la totalité du langage OCaml. Lors de nos tests, nous nous sommes également aperçus que le typage implicite d'OCaml rendait ambigüe la distinction entre du code monadique et du code "traditionnel". Ces caractéristiques font qu'il reste difficile de l'employer dans un projet à grande échelle.

Les auteurs précisent qu'ils sont ouverts à toute remarque de la communauté, et envisage d'étendre leur technique à des langages à types dépendants.

## Références

[FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 192–203, New York, NY, USA, 1999. ACM.

[SGLH] Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. Coco. http ://research.microsoft.com/en-us/projects/coco/.

[SGLH11] Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. Lightweight monadic programming in ml. In *ICFP*, pages 15–27, 2011.

[Wad92] Philip Wadler. The essence of functional programming. In *POPL*, pages 1–14, 1992.