Initiation aux effets algébriques

Xavier Van de Woestyne
xvw.github.io - margo.com

ScalaIO 2018

- Xavier Van de Woestyne (Data Engineer chez **Margo.com**)
- https://xvw.github.io
- **xvw** sur Gitlab et Github
- **@vdwxv** sur Twitter et **@xvw@sunbeam.city** sur Mastodon
- Phutur, LilleFP
- J'aime bien programmer (OCaml, Haskell, Erlang, Elm, Io, F#)
- **Je suis débutant !!!**

Objectifs de la présentation

- Présenter la notion d'**effets**
- Survoler des approches "historiques" de l'implémentation d'effets
- Présenter les effets **algébriques** et les **gestionnaires** d'effets (un pan assez actif de la recherche)
- Comprendre les intérêts liés à l'utilisation d'effets (algébriques)
- Soit, présenter quelque chose d'assez simple à utiliser (mais dur à implémenter).

  C'est une présentation pour débuter !

Non-objectifs

- Ce n'est pas un *Monad-tutorial*
- Les **co-effets**

Quelques pré-requis

Eléments syntaxiques de OCaml

```
val e : t
'a list, ('a, 'b) result, ('a * 'b)
type 'a option = Some of 'a | None
type _ t =  A : int t | B : float t
```

Qu'est-ce que (et à quoi sert) le λ-calcul ?

Effets et effets de bords

Programmer sans effets reviendrait à dérouler un programme dans une boite opaque sans possibilité **d'orchestration** ou de **résultat** "*visible*".

Un effet est potentiellement différent d'un effet de bord.

Un effet de bord

$\Gamma \vdash e : \tau$ représente une expression standard.

Dans l'environnement Γ, l'expression e est de type τ.

Observons ces deux signatures

```
val incr : int -> int
val print_string : string -> unit
```

*Un effet de bord est un effet **observable** d'un calcul, **autre** que sa valeur de retour. Par exemple, écrire une chaine de caractère sur `stdout`.*

Ce que l'on voudrait

$\Gamma \vdash e : \tau \& \phi$:

```
val print_string : string -> unit & io
```

Ou plus précisemment, avec des types paramétrés :

$\Gamma \vdash e : \phi(\tau)$:

```
val print_string : string -> unit io (** OCaml *)
printString :: String -> IO () -- Haskell
```

Comment **contrôler** des effets dans un langage “pur” (ou presque pur)... ou en voulant “tenter” d’être pur ?

Monades et effets

Deux approches différentes, mais liées, des monades

- **Moggi** : raisonner un programme à effet
  - Une monade est la **sémantique dénotationnelle** d'un effet :
  - Un λ-calcul avec des effets traduit en un λ-calcul pure (**Style direct**, Grammar)
  - Soit : raisonner un langage à effets
- **Wadler** : implémenter un programme à effet
  - Une monade est une technique de programmation
  - Encodage des effets dans un λ-calcul pur (**Style indirect**, CPS, "*The mother of all monads*")
  - Soit : écrire et raisonner un encodage pur

Les deux solutions offrent des opportunités

Un exemple un peu capilotracté :

Style direct	**Style indirect**
Async + Await	Promise

Quelques problèmes solutionnés par l'abandon de la pureté

List non-exhaustive issue des recherches de Moggi

- **Non déterminisme** : un calcul peut renvoyer plusieurs valeurs potentielles
- **Mutation/lectures** : lire un état/environnement, écrire un état/log, lire et écrire un état
- **Exception** : fonction partielles pouvant échouer
- **Continuation** : capacité à stocker un état du programme et le restituer à la demande
- **I/O interactif**

Implémentation via des monades

- **Non déterminisme** : Monade List
- **Mutation/lectures** : Monade Reader/Writer/State
- **Exception** : Monade d'erreur
- **Continuation** : Monade de continuation
- **I/O interactif** Monade I/O

A propos de la composition des effets

- **Les** monades ne **composent** pas
- **Transformation de monades** à la rescousse
  - *Boilerplate lourd*
  - impossibilité de regrouper intelligemment les effets

Au delà de ne pas se composer, encoder des effets dans des monades viole un des principes de l'encapsulation :

> *"Program to an interface, no to an implementation"*

Effets algébriques et gestionnaire d'effets

Gordon et Plotkin, 2002

Effet algébriques

- Une représentation “formelle” d’un effet de bord
- Un effet présenté comme un ensemble d’opération (comme set/get)
- alternative aux monades

Gestionnaire d’effets (Effect Handler)

- Un *design* de langage de programmation “inspiré” par les effets

Langages offrant des “effets algébriques”

Ils sont tous en développement :

- **Koka**
- **Eff**
- **Links**
- **Frank**
- **OCaml+effects**

Exemple

```
exception FakeEffect of int
let f n = (** Une fonction qui raise FakeEffect *)
let () =
  let result =
    try f 10 with
    | FakeEffect n -> n + 1
  in print_int result
```

Exemple

Une syntaxe alternative (inspirée des effets) qui remonte l'idée des valeur de retour contre les retour d'exceptions

```
exception FakeEffect of int
let f n = (** Une fonction qui raise FakeEffect *)
let () =
  let result = match f x with
  | exception (FakeEffect x) -> x + 1
  | x -> x
  in print_int result
```

On voudrait pouvoir “continuer” via des exceptions résumables

```
exception FakeEffect of int
let f n = (** Une fonction qui raise FakeEffect *)
let () =
  let result = match f x with
  | exception (FakeEffect continuation _) ->
    let x = read_int () in
    continue continuation x
  | x -> x
  in print_int result
```

Comment définir le type de la continuation ?

Utilisation d'effets algébriques

Concrètement, un effet algébrique, ce n'est rien de plus qu'une exception "résumable".

Définition d'effets :

Une signature d'effets : un **ensemble** d'opérations

```
effect choice =
  | Choose : bool
effect IO =
  | Print : string -> unit
  | Read : string
effect int_state =
  | Get : int
  | Set : int -> unit
effect scheduler =
  | Spawn : (unit -> unit) -> unit
  | Yield : unit
```

Code à effet (exécution d'effets)

```
let rec in_interval a b =
  if a = b then a
  else
    if perform Choose then a
    else in_interval (a + 1) b
```

Plutôt que de retourner une valeur, on **perform** une opération.

Gestionnaire d'effets avec la syntaxe “handle”

```
let a_number =
  handle (in_interval 1 10) with
  | Choose k -> continue k (Random.bool ())
```

Concrètement

L'intégration des effets algébriques ajoute plusieurs mots-clé

- effect, pour définir l'ensemble des opérations de l'effet
- perform, pour qu'une fonction exécute un effet
- handle et continue pour déclarer un gestionnaire d'effet (un try catch avec une notion de continuation).

L'exécution d'un effet sans handler implique une erreur de compilation.

Concrètement, ça laisse à l'utilisateur **définir ses propres effets**. Ça structure le code et le rend plus facile à raisonner.

Impacte sur le système de type

- `val print_string : string - [IO] -> unit`
- `-[!e]->` pour représenter un effet générique

Apports des effets algébriques

- Rendre le code lisible et mieux structuré malgré la présence d'effets
- Implémenter facilement des comportements complexes (qui n'introduisent pas de nouveaux mots clés)
- Fragmenter le programme de sa partie pure et impure (comme dans une monade libre)
- Les effets se composent (et s'ordonnent)

En OCaml, ça permet d'implémenter, entre autre, le support du multi-coeur.