Des promesses, des actions, par flots, en OCaml

Simon Archipoff[1], David Janin[1], and Bernard Serpette[2]

[1] UMR CNRS LaBRI, Bordeaux INP,
Université de Bordeaux
Simon.Archipoff | David.Janin@labri.fr
[2] Inria Bordeaux Sud-Ouest
Bernard.Serpette@inria.fr

Résumé

Nous nous intéressons aux traitements de flux audio synchrones et aux contrôles de ces traitements par des flux de commandes asynchrones : un exemple archétypal de problème de programmation globalement asynchrone et localement synchrone (GALS). Pour faire cela, un modèle qui s'impose tout à la fois par son efficacité et son élégance se situe à la croisée de plusieurs concepts clés de la programmation fonctionnelle : les actions monadiques, les promesses, et les flots récursifs d'actions ou de promesses. Suite à une expérimentation de cette modélisation en Haskell, l'objet de cet article est d'en faire une expérimentation en OCaml. De façon quelque peu inattendue, les deux approches se complètent et nous conduisent finalement à une axiomatisation originale de la notion de promesse. De plus, étendu aux flots récursifs d'actions, il apparaît que les promesses de flots d'actions peuvent être vue comme des flots de promesses. Une mise en œuvre de ces concepts est proposée en s'appuyant sur les extensions d'OCaml par threads Posix ou par threads légers (lwt).

1 Introduction

Poursuivant notre étude de la modélisation de systèmes multimédia interactifs et temporisés [1, 2] (musique, son, animation, etc...) pour leur programmation à l'aide de languages fonctionnels typés génériques tels qu'OCaml ou Haskell, nous nous intéressons aujourd'hui aux traitements de flux audio synchrones et aux contrôles de ces traitements par des flux de commandes asynchrones : un exemple archetype de problème de programmation globalement asynchrone et localement synchrone (GALS).

Actions monadiques. Depuis les travaux de Moggi [13] et de Wadler [14], on sait qu'il est possible de modéliser les effets de bords en programmation fonctionnelle pure grâce à la notion d'actions monadiques.

En première approche, une action monadique peut être vue comme une fonction qui prend en entrée un « état du monde » et produit une valeur tout en modifiant cet « état du monde ». Cette modification de l' « état du monde » modélise l'effet de bord provoqué par l'application de cette action.

Sans forcément être identifiée comme telle, cette approche est omniprésente dans l'étude de la sémantique des langages de programmation. Par exemple, la sémantique d'une affectation `x := e` dans un langage impératif peut être vue comme l'action monadique qui prend en entrée l'état mémoire d'un programme en cours d'exécution, un état dans lequel la valeur de la variable `x` est définie, évalue l'expression `e` et modifie la valeur de la variable `x` en conséquence.

Avantage de cette approche, les effets de bord sont limités aux états de la monade considérée. On peut alors analyser les propriétés des programmes en raisonnant sur l'action qu'ils ont sur les états de la monade. La logique de Hoare avec pré et post-conditions sur les programmes impératifs en est un exemple.

Flots monadiques. Nous nous proposons ici d'étudier une application de cette programmation monadique à la programmation par flots de données.

Typiquement, en programmation par flots de données, l'accès à une nouvelle entrée, tout comme la production d'une nouvelle sortie, est un effet de bord. En suivant l'approche de Moggi et Wadler, il est donc naturel de modéliser ce traitement d'un nouvel élément d'un flot de données par une action monadique. De plus, le traitement de cet élement ouvre aussi, pourrait-on dire, l'accès à la suite de ce flot. Il apparait alors qu'un flot de données peut être modélisé par une action monadique qui, lorsqu'elle est évaluée, produit, selon le cas :

(1) un marqueur de fin dans le cas où le flot est terminé,

(2) une paire constituée de la valeur de l'élément courant et de la continuation de ce flot de données dans le cas contraire.

Autrement dit, ces flots, que nous appellerons flots monadiques, apparaissent comme des listes explicitement paresseuses, dont l'accès est systématiquement gardé par une action monadique.

Bien entendu, on le devine ici, nous ne modélisons que des flots discrets ou, pour être plus précis, des flots localement finis. Plus précisément, un flot de données est dit *localement fini* lorsque le nombre de valeurs produites par ce flot sur tout intervalle de temps de durée finie est nécessairement fini. Cette définition exclut donc les flots de type « Zenon [1] » qui ne sont pas localement fini puisqu'ils présentent des points temporels d'accumulation d'une infinité de valeurs.

Fonctions de flots. L'un des points clés de notre approche pour la programmation par flots de données est qu'une même structure de données, la notion de flot monadique, permet de représenter tout à la fois les flots d'entrées et les flots de sorties. En effet, un flot monadique apparait comme :

(1) un flot d'entrée lorsqu'on *exécute* (récursivement) les actions (imbriquées) qui le définissent pour *lire* les valeurs qu'il porte,

(2) un flot de sortie lorsqu'on *produit* (récursivement) les actions (imbriquées) qui le définissent pour *écrire* les valeurs qu'il porte.

Dans le codage d'une fonction de flots, la correspondance entre l'exécution des actions d'entrée et la production des actions de sortie définit alors la synchronicité entre les entrées et les sorties. La programmation monadique offre un contrôle total de cette synchronicité. En particulier, lorsque les entrées sont cadencées au même rythme que les sorties, on parlera de fonctions synchrones, et notre approche permet, comme nous le verrons, de programmer toutes les fonctions synchrones définissables à l'aide d'une machine de Mealy.

Et des promesses. Pour une application réelle de notre approche aux traitements des flots de données, il ne suffit pas de pouvoir définir toutes ces fonctions synchrones. Le contrôle de ces fonctions, souvent cadencés à des fréquences moins élevées que les flots traités, doit aussi pouvoir être décrits et mis en œuvre. Les mécanismes de concurrence asynchrone offerts par la notion de promesses, initialement appelées *future values* [4], permettent cela [3, 12, 11].

Plus encore, la notion de promesse va aussi nous permettre de partager des flots de données sans en partager les effets de bord. Par exemple, un flots d'entrée typique que nous définirons ici, est le flot des entrées standards. Sa lecture fait appel, de façon répétitive, à une action monadique `getChar` qui renvoie (de façon bloquante) le caractère suivant entré sur le clavier. Le partage de ce flot monadique parmi plusieurs processus indépendants ne conduit pas à

1. voir https://fr.wikipedia.org/wiki/Paradoxes_de_Z%C3%A9non

dupliquer ces entrées mais, au contraire, à les distribuer, chaque processus utilisateur de ce flux exécutant ses propres instances de l'action `getChar`.

Le partage de la même suite de caractères d'entrée parmi plusieurs processus peut être réalisé par une promesse de flots, une promesse qui, comme nous le verrons, peut se définir simplement et efficacement comme un flot de promesses.

2 Actions monadiques

En Haskell [5], langage de programmation paresseux et donc sans effets de bord, la programmation par monades est nécessaire et, d'une certaine façon, omniprésente. Le codage des flots monadiques est donc relativement aisé. Le lecteur interessé pourra consulter un exemple de ce codage [7] utilisé pour du traitement et du controle audio temps réel.

En OCaml [10], langage de programmation strict avec effets de bord, la programmation par monades est d'un usage beaucoup plus confidentiel parce que largement inutile. La plupart des instructions s'exécutent dans une monade IO implicite mais omniprésente. Nous proposons ici un codage générique de ces monades et une instanciation particulière qui explicite et mime en quelque sorte la monade IO d'Haskell.

Un aspect important de la programmation monadique est que les actions dites monadiques qu'on y manipule sont des valeurs comme les autres. Sauf à être passées en argument d'une fonction d'`exec`, elles ne sont pas exécutées. Au contraire, la programmation monadique consiste à créer puis à combiner des actions les unes avec les autres, parfois en prenant d'autres actions en paramètres, produisant ainsi des programmes toujours plus complexes qui seront, in fine, exécutés dans l'ordre prescrit, lors de l'exécution du programme principal.

Autrement dit, les actions monadiques forment un type de données qui est, par essence, paresseux. En OCaml, le codage le plus simple de la paresse consiste à définir des *glaçons*[2], c'est à dire une fonction de type `unit -> 'a`.

Les actions monadiques seront alors définies comme des glaçons encapsulés par un constructeur de type afin de les distinguer, par typage, des fonctions de type `unit -> 'a` que nous pourrions utiliser par ailleurs. Dans ce codage, l'état du monde évoqué en introduction reste implicitement représenté par l'état mémoire du programme OCaml. L'unique valeur de type `unit`, passée en paramètre d'un glaçon, ne sert qu'à déclencher son évaluation.

2.1 Définition générique des monades : module `Monad`

En OCaml, la signature d'une monade peut être définie par :

```
module type MONAD_CORE =
  sig
    type 'a m
    val return : 'a -> 'a m
    val bind   : 'a m -> ('a -> 'b m) -> 'b m
  end
```

avec le constructeur de type `m` pour les actions monadiques, la fonction `return` qui permet de

2. En anglais, ces glaçons s'appellent des *thunks*, un mot dont nous avons échoué à trouver une traduction littérale adéquate. Bien entendu, à la différence des *thunks* utilisés pour coder une évaluation paresseuse, nous ne sauvegardons pas la valeur retournée, deux executions distinctes d'une même action, telle `getChar`, pouvant retourner des valeurs différences.

créer une action monadique, et la fonction `bind` qui permet de combiner deux actions monadiques, la seconde action prenant en paramètre le résultat de la première. Ces fonctions doivent satisfaire les équations suivantes :

$$\texttt{bind (return a) f} \equiv \texttt{f a} \quad (1)$$

$$\texttt{bind m return} \equiv \texttt{m} \quad (2)$$

$$\texttt{bind (bind m f) g} \equiv \texttt{bind m (fun a -> bind (f a) g)} \quad (3)$$

pour toute valeur `a : a`, toute action monadique `m : 'a m` et toutes fonctions `f : 'a -> 'b m` et `g : 'b -> 'c m`. Les équations (1) et (2) assurent que, en un certain sens, la fonction `return` agit comme un neutre à gauche et à droite pour la fonction `bind`, l'équation (3) assurant que ce `bind` est associatif.

Les primitives d'une monade, définies par un module de type `MONAD_CORE`, sont systématiquement étendues par les fonctions suivantes :

```
module type MONAD =
  sig
    include MONAD_CORE
    val (>>=) : 'a m -> ('a -> 'b m) -> 'b m
    val (>>) : 'a m -> 'b m -> 'b m
    val fmap : ('a -> 'b) -> 'a m -> 'b m
  end
```

définies par :

```
module Monad (M: MONAD_CORE) : MONAD
       with type 'a m = 'a M.m
  =
  struct
    type 'a m = 'a M.m
    let return = M.return
    let bind = M.bind
    let (>>=) = bind
    let (>>) m1 m2 = bind m1 (fun _ -> m2)
    let fmap f m = bind m (fun x -> return ( f x))
  end
```

L'opérateur `(>>=)` n'est qu'une redéfinition infixe de la fonction `bind`. La fonction `fmap` explicite le fait que le constructeur de type `m` dans une monade peut être vu comme un foncteur envoyant l'identité sur l'identité et la composition de deux fonctions sur la composition de leurs images. Grâce aux équations (1)–(3), on peut en effet prouver que :

$$\texttt{fmap id} \equiv \texttt{id} \quad (4)$$

$$\texttt{fmap (g} \circ \texttt{f)} \equiv \texttt{(fmap g)} \circ \texttt{(fmap f)} \quad (5)$$

pour `id = fun x -> x` et toutes fonctions `f : 'a -> 'b` et `g : ' b -> 'c`.

D'un point de vue sémantique, nous dirons que deux actions sont équivalentes lorsqu'elles ont le même comportement dans tout contexte d'utilisation, effets de bord inclus. Autrement dit, deux actions sont équivalentes lorsqu'elles agissent de la même façon sur l'état de la monade et produisent des valeurs équivalentes dans tout contexte d'utilisation.

2.2 Une instance de monade : la monade `IO`

Un exemple simple de monade, mimant en quelque sorte la monade IO d'Haskell, l'état du monde étant implicitement définit par le runtime d'OCaml, est obtenu en définissant `'a io`, le type des actions monadiques retournant une valeur de type `'a`, comme une encapsulation du type des glaçons `unit -> 'a`. Plus précisément, on pose :

```
type 'a io = IO of (unit -> 'a)
module IO_CORE : MONAD_CORE
     with type 'a m = 'a io
  =
  struct
    type 'a m = 'a io
    let return a = IO (fun () -> a)
    let bind (IO g) f
      = IO (fun () -> let a = g () in let IO m = f a in m ())
  end
```

On étend alors la monade résultante d'une fonction `run` qui nous permet d'exécuter, en temps utile, une action IO, en posant :

```
module type MONAD_IO =
  sig
    include MONAD
    val run : 'a m -> 'a
  end

module IO : MONAD_IO with type 'a m = 'a io =
  struct
    include Monad (IO_CORE)
    let run (IO s) = s ()
  end
```

Il est important de se souvenir que les actions monadiques ne sont pas exécutées, sauf à être passées en argument de la function `run` ci-dessus.

Deux exemples classiques d'actions dans la monade IO sont les suivantes :

```
let getChar : 'char io
  = IO (fun () -> input_char stdin)

let printChar c : unit io
  = IO (fun () -> print_char c)
```

Ces fonctions nous permettrons d'illustrer la notion de flot monadique dans la monade IO sur les entrées et sorties standards.

3 Flots d'actions monadiques

Nous avons maintenant tous les ingrédients disponibles pour définir et manipuler, en OCaml, la notion de flots d'actions monadiques ou, plus simplement, flots monadiques.

3.1 Flots monadiques

Dans un contexte de programmation réactive, une façon simple de représenter les flots de données, finis ou infinis, consiste à définir un flot de type `'a s`, c'est-à-dire un flot contenant des valeurs de type `'a`, comme l'encapsulation d'une action monadique dont l'exécution renvoie :

(1) soit la valeur `None` lorsque le flot est *terminé*,

(2) soit `Some (a , sc)` lorsque le flot *produit* la valeur `a` et *continue* comme `sc`.

Cette définition, augmentée de quelques fonctions auxiliaires, est formalisée en OCaml par l'interface suivante :

```
module type STREAM  = functor (M : MONAD)  ->
  sig
    type 'a s = Stream of ('a * 'a s) option M.m
    val exec : 'a s -> unit M.m
    val fromStream : 'a s -> ('a * 'a s) option M.m
    val toStream : ('a s) M.m -> 'a s
    val fmap : ('a -> 'b) -> 'a s -> 'b s
  end
```

instanciée par :

```
module Stream : STREAM = functor (M : MONAD) ->
  struct
    type 'a s = Stream of (('a * ('a s)) option) M.m
    let rec exec (Stream m) = M.bind m (function
          | None -> M.return ()
          | Some (a,sc) -> exec sc)
    let fromStream (Stream m) = m
    let toStream ms = Stream (M.bind ms fromStream)
    let rec fmap f (Stream m) = Stream (M.bind m  (function
         | None -> M.return None
         | Some (a,sc) -> M.return (Some ((f a),(fmap f sc)))))
  end
```

Le constructeur de type `s` du module `Stream(M)` défini ci-dessus est connu pour être le *transformateur de monade* associé aux listes inductives. Appliqué au foncteur identité, il ne fait que reproduire le type liste.

La fonction `fromStream` permet simplement d'accéder, en évitant un « pattern matching », à l'action définissant le flot monadique. La fonction `toStream : 'a s m -> 'a s`, qui n'est pas réciproque de la précédente, permet de transformer une action produisant un flot en le flot qu'elle produit. Ces deux fonctions seront surtout utilisées pour simplifier notre code.

La fonction `fmap` est un exemple archétype de constructeur de fonctions synchrones (voir section suivante). Elle illustre en particulier la façon dont les flots monadiques codent, tout à la fois, des flots d'entrées comme des flots de sorties. En effet, un flot monadique peut être vu :

(1) comme un flot d'entrée lorsqu'on « exécute » récursivement les actions qui le compose,

(2) comme un flot de sortie lorsqu'on « produit » récursivement les actions qui le compose.

D'une certaine façon, dans le code de `fmap` c'est la fonction `bind` qui réalise cette correspondance entre entrées et sorties, le premier argument du bind étant une action d'entrée, à exécuter, dont le résultat est transmis à son deuxième argument qui produit l'action de sortie. La fonction

`fmap` montre aussi que le constructeur de type `s` peut être vu comme un foncteur de type. On peut en effet vérifier que les équations (4) et (5) sont satisfaites.

3.2 Fonctions synchrones dérivées

Nous dirons qu'une fonction sur les flots de données est synchrone lorsque tous ses flots d'entrée et son flot de sortie sont lus/produits à la même cadence. Comme déjà illustré ci-dessus par la fonction `fmap`, définir une fonction synchrone, revient à associer chaque action(s) de(s) flot(s) d'entrée à une action du flot de sortie. Des exemples de fonctions synchrones sont définis dans le module suivant :

```
module SStream (M : MONAD) =
  struct
    include Stream(M)
```

La fonction `zip : 'a s -> 'b s -> ('a * 'b) s` permet de fusionner, de façon synchrone, deux flots d'entrées, en utilisant l'opérateur `>>=`, version infixe de la fonction `bind` :

```
    let rec zip (Stream m1) (Stream m2) =
      Stream (m1 >>= (function
         | None -> return None
         | Some (a1,sc1) ->
            m2 >>= (function
               | None -> return None
               | Some (a2,sc2) ->
                  return (Some ((a1,a2),zip sc1 sc2)))))
```

Remarquons que, si les actions de ces flots d'entrées, supposées bloquantes, sont cadencées sur des horloges non synchronisées, le flot de sortie du zip est cadencé à chaque instant sur l'horloge la plus lente ; cette horloge de sortie peut être vue comme la borne supérieures des horloges d'entrées.

La fonction `iterate : 'a M.m -> 'a s` permet de créer un flux infini obtenu par itération d'une action d'entrée. La fonction `iterateSome : 'a option M.m -> 'a s` est une variante de la fonction précédente produisant un flot qui s'arrête dès que l'action itérée retourne `None`.

```
    let rec iterate m
      = Stream (m >>= (fun x -> return (Some (x, iterate m))))
    let rec iterateSome m
      = Stream (m >>= (function
              | None -> return None
              | Some x -> return (Some (x, iterateSome m))))
```

La fonction `loop : 's -> ('a * 's -> 'b * 's)-> 'a s -> 'b s` permet de coder n'importe quelle machine de Mealy.

```
    let rec loop s f (Stream m)
      = Stream (m >>= (function
           | None -> return None
           | Some (a,sc) ->  (f (s,a)) >>= (function
               | (s',b) -> return (Some (b, loop s' f sc)))))
  end
```

En effet, dans un appel `loop s f`, le premier argument `s : 's` définit l'état initial de la ma-

chine, le second argument `f : 'a * 's -> 'b * 's` définit tout à la fois la fonction d'entrée/-sortie et la fonction de mise à jour de son état courant.

3.3 Application aux flots d'IO : module `streamIO`

Appliqué à la monade IO, on obtient le module `StreamIO` qui permet d'illustrer le concept de flot monadique sur les entrées et sorties standards.

```
module StreamIO = SStream (IO)

let stdinStream = StreamIO.iterate getChar
```

La fonction d'écho de l'entrée standard sur la sortie standard peut alors être codée par :

```
let printStream s = StreamIO.exec (StreamIO.fmap print_char s)

let echo = IO.run (printStream (stdinStream))
```

On peut aussi lui préférer une version qui termine sur l'entrée d'un caractère particulier `c` agissant comme marque de fin :

```
let stdinStreamUntil c
  = StreamIO.iterateSome
      (IO.bind getChar
         (fun x -> if (x == c) then IO.return None
                   else IO.return (Some x)))

let echoUntil c = IO.run (printStream (stdinStreamUntil c))
```

Dans les deux cas, l'expérimentation de ces fonctions d'échos montre qu'elles s'exécutent sans fuite de mémoire.

4 Des flots aux promesses de flots

Lire le flot d'entrées à travers le flot monadique `stdinStream` signifie, sans surprise, exécuter de façon itérative l'action `getChar`. Cela implique que deux processus partageant ce même flot `stdinStream` feront, chacun, des appel à `getChar`. Les valeurs de ce flot d'entrée ne seront donc pas partagées par ces deux processus, mais, on contraire, distribuées entre ces processus, chacun exécutant de façon indépendante l'action `getChar`. C'est avec une notion de promesse de flot, qui se codera par un flot de promesses, que nous remédierons à cela : les promesses de flots, comme les promesses usuelles, permettant de partager le résultat d'une action sans reproduire ses effets de bord.

Plus précisément, nous définissons dans un premier temps une notion de référence monadique dont nous dériverons la notion de promesse telle qu'elle apparait dans les bibliothèques `async` et `lwt`. Défini au sein d'une monade, nous pouvons aussi axiomatiser quelques unes des propriétés de ces références monadiques, nous permettant alors de prouver que les promesses qui en découlent forment bien une monade. Appliquées aux flots monadiques, ces références monadiques nous permettent de définir les promesses de flots annoncées et, in fine, des fonctions asynchrones sur ces flots.

4.1 Références et promesses monadiques

Une référence monadique est associée à une action monadique en cours d'exécution. Elle permet d'attendre puis d'accéder, quand elle est disponible, à la valeur retournée par cette action. En OCaml, nous pouvons la définir de façon générique avec l'interface :

```
module type MONADREF =
  sig
    include MONAD
    type 'a mref
    val fork : 'a m -> 'a mref m
    val read : 'a mref -> 'a m
  end
```

L'action `fork m` permet de lancer l'action `m` passée en paramètre et retourne une référence à cette action. L'action `read r` permet au contraire d'attendre et de lire la valeur produite par l'action référencée par `r`.

La sémantique élementaire de ces références est capturée par les trois équations suivantes qui devront être satisfaites par toutes instances.

$$\texttt{(fork m)>>= read} \equiv \texttt{m} \tag{6}$$

$$\texttt{fork} \circ \texttt{read} \equiv \texttt{return} \tag{7}$$

$$\texttt{fork(m >>= f)} \equiv \texttt{(fork m)>>= fun r -> fork(read r >>= f)} \tag{8}$$

pour tout action `m : 'a m`, et toute fonction `f : 'a -> 'b m`. C'est le résultat de la fonction fork, de type `'a mref m`, qui est défini comme une *promesse*, l'équation (6) ci-dessus explicitant la composition monadique (bind) d'une promesse a un comportement équivalent à celui de l'action qui a été lancée. Le lecteur intéressé pourra consulter l'étude détaillée [6] de la notion de référence monadique.

4.2 Exemple de références d'IO : le module `IO_Ref`

Les références d'IO sont construites à l'aide de `MVar` [8], définit grâce aux `Mutex` et `Control` de l'extension `threads` d'OCaml. La création de références d'IO (et donc de promesses d'IO) passe par le lancement de threads indépendants qui vont mettre à jour ces promesses.

```
module IO_Ref : MONADREF
  with type 'a m = 'a io and type 'a mref =  'a MVar.mvar
  =
  struct
    include IO
    type 'a mref = 'a MVar.mvar
    let rec fork m
      = let v = MVar.createEmpty () in
        ignore (Thread.create (fun _ -> MVar.put v (run m)) ());
        return v
    let read v = return (MVar.read v)
  end
```

Les `mvar` permettent la communication entre les threads. Une `mvar` ne peut être que dans deux états : vide ou pleine. La fonction `put` met une valeur dans une `mvar`, en bloquant tant qu'elle

est pleine. La fonction `read` permet de lire la valeur d'une `mvar` sans la modifier. La fonction `take` permet de lire la valeur d'une `mvar` en la laissant vide. Ces deux fonctions sont bloquantes tant que la `mvar` est vide.

Cette instance non triviale de notre définition de références monadiques permet de vérifier, sur toutes sortes d'exemples concrets, la validité des équations (6)–(8).

4.3 Les promesses forment-elles une monade ?

Dans les deux bibliothèques `async` et `lwt` d'OCaml qui implémentent les promesses, ces dernières sont présentées à travers le prisme de la programmation monadique puisque c'est une fonction `bind` qui permet d'associer une *call-back* à une promesse. Est-ce une convention d'écriture ou l'existence réelle d'un foncteur monadique sur les types induit par les promesses ?

Nous avons défini ci-dessus les promesses comme les actions renvoyées par la fonction `fork`. On retrouve une interface analogue à ces deux bibliothèques en posant :

```
module type ASYNC = functor (M : MONADREF) ->
  sig
    type 'a p = 'a M.mref M.m
    val return : 'a -> 'a p
    val bind : 'a p -> ('a -> 'b p) -> 'b p
    val fmap : ('a -> 'b) -> 'a p  -> 'b p
  end
```

instancié par :

```
module Async : ASYNC = functor (M : MONADREF) ->
  struct
    type 'a p = 'a M.mref M.m
    let return a = M.fork (M.return a)
    let bind m f = M.bind m
          (fun r -> M.fork (M.bind (M.bind (M.read r) f) M.read))
    let fmap f m = bind m (fun r -> return (f r))
  end
```

Grâce aux équations (6)–(8) on peut vérifier que la fonction `fmap` satisfait bien les équations (4)–(5) prouvant ainsi que la fonction de type `'a -> 'a p` induite par une monade avec références `M` est bien un foncteur de type, dans le *cas général* d'actions monadiques de type `'a mref m`. On peut aussi formellement prouver que la fonction `fmap` peut être directement définie dans la monade `M` par :

```
Async.fmap m f  ≡  fork (m >>= fun r -> bind (read r)>>= (return ∘ f))   (9)
```

dès lors que les propriétés (6)–(8) sont satisfaites. Par ailleurs, tout comme dans la bibliothèque `async` la fonction `Async.bind` définie ici permet d'associer à une promesse `m` une fonction de *call-back* qui sera déclenchée à la réalisation de la promesse. Les fonctions `return` et `bind` définies ci-dessus définissent-elle pour autant une monade de promesses ? La réponse à cette question est positive *à condition* de se limiter à la notion de promesses définies ci-dessus.

Plus précisément, dans le cas général d'actions de type `'a p`, on peut vérifier que les équations (2)–(3) sont satisfaites. Par contre, l'équation (1) ne l'est pas. Il faut se restreindre aux fonctions `f : 'a -> 'b p` qui renvoient bien des promesses, c'est-à-dire des fonctions `f` de la forme `fork ∘ f1` avec `f1 : 'a -> 'b m`.

En effet, un raisonnement simple montre que la validité de l'équation (1) dans le cas général implique, dans la monade `m` étendue par des références, l'équivalence

```
fork (bind m read) ≡ m
```

pour toute action `m : 'b mref m`. Mais cette équivalence admet un contre exemple dans n'importe quelle extension de type `MONADREF` de la monade IO en posant :

```
m = bind readChar (fun c -> fork (return c))
```

on constante en effet que l'action `m` bloque tant qu'un caractère n'est pas entré sur l'entrée standard, alors que l'action `fork(bind m read)` renvoie sans attendre une référence monadique. N'ayant pas les mêmes effets de bord, ces deux actions ne peuvent être équivalentes, bien qu'*in fine* elles renvoient la promesse du même caractère d'entrée.

Plus encore, on peut vérifier que la notion de référence d'action monadique induit une sorte de catégorie de Kleisli dont les objets sont les types et les flèches de la forme `F = fork ∘ f1` pour toute fonction `f1 : ' a -> 'b m` de la catégorie de Kleisli associée à la monade `m`, avec la composition définie par :

```
g ∘ f = fun a -> fork((f a) >>= read >>= g >>= read)
```

Modulo les équivalences d'actions monadiques induites par (1)–(3) pour la monad `m` et les équivalences de promesses induites par (6)–(8) pour son extension par des références, il apparait que ces deux catégories de Kleisli sont *isomorphes*.

Autrement dit, on peut prouver formellement que la programmation par promesses définies dans une monade `m` est *structurellement* équivalente à la programmation par actions monadiques dans cette même monade.

4.4 Flots de promesses

Nous souhaitons maintenant définir une extension de cette notion de promesses applicables aux flots monadiques. Plus précisément, nous souhaitons disposer d'un mécanisme permettant de lancer l'évaluation d'un flot et obtenir en retour un accesseur aux valeurs portées par ce flot, c'est-à-dire une promesse de flot.

Pour ce faire, nous transformons la définition de nos flots monadiques en une définition de flots de références monadiques.

```
module StreamRef (M : MONADREF)
  =
  struct
    include SStream(M)
    open M
    type 'a sref
       = StreamRef of (('a * ('a sref)) option) mref
```

On peut observer l'analogie entre les inductions définissant les flots dans `Stream` et `StreamRef`. Elles ne diffèrent en effet que par le nom de leur constructeur et le constructeur de type utilisé en paramètre : `m` pour `Stream` et `mref` pour `StreamRef`. La définition du module `StreamRef(M)` se poursuit avec :

```
    let forkStream s =
      let rec evalAndFork (Stream m)
        = m >>= function
              | None -> return None
              | Some (a,sc) -> (fork (evalAndFork sc)) >>=
                   (fun r -> return (Some (a, StreamRef r)))
      in (fork (evalAndFork s)) >>=
            (fun r -> return (StreamRef r))
```

de type `'a s -> 'a sref m` qui permet de créer une promesse de flot en lançant en quelque sorte l'évaluation du flot argument, et

```
    let rec readStream (StreamRef r)
      = Stream ((read r) >>= function
               | None -> M.return None
               | Some (a,rc) ->
                  M.return (Some (a, readStream rc)))
end
```

de type `'a sref -> 'a s` qui permet de (re)lire, autant de fois que nécessaire, le flot référencé. On peut vérifier qu'on a bien :

$$\texttt{(forkStream s) >>= (return} \circ \texttt{readStream)} \quad \equiv \quad \texttt{return s} \tag{10}$$

pour tout flot monadique `s : 'a s`. On peut donc formellement définir une promesse de flot de type `'a s` comme le résultat de la fonction `forkStream` de type `'a sref m`, le bind de la monade sous-jacente nous permettant d'associer une call-back non pas à un seul évènement promis, mais aux flots d'évènements promis par cette promesse de flot.

Remarquons que, tout comme avec les références monadiques, alors que l'évaluation d'un flot peut avoir des effets de bord, sa relecture à travers une promesse n'en a essentiellement aucun, sauf à attendre les valeurs promises par le flot lancé.

Ces promesses de flots se révèlent aussi très utiles pour dupliquer le contenu d'un flot sans dupliquer les effets de bord associés à l'exécution des actions monadiques qui le définissent. On peut en effet tester la correction de toutes les fonctions ci-dessus en posant :

```
let echoRZUntil c = IO.run (printStream(toStream
     (forkStream (stdinStreamUntil c) >>=
        fun r -> return (readStream r))))

let print_double_char = fun (c1,c2) ->
  print_char '(' ;print_char c1 ;print_char ',';
  print_char c2 ;print_char ')'

let printDoubleStream s
  = exec (streamIO.fmap  print_double_char s)

let echo2RZUntil c = IO.run (printDoubleStream(toStream
     (forkStream (stdinStreamUntil c) >>=
        fun r -> return (zip (readStream r) (readStream r)))))
```

qui effectue, sans perte de mémoire observable, un double echo de l'entrée standard.

5 Un peu plus de concurrence

L'ajout de promesses et de concurrence ne saurait être complet sans offrir, de plus, la possibilité de faire des lectures de promesses non bloquantes et de trier les promesses par ordre de terminaison.

Comme nous allons le voir, cela va nous permettre d'enrichir notre bibliothèque de flots monadiques par des fonctions asynchrones nous permettant de faire, comme annoncé en introduction, de la programmation GALS.

5.1 Promesses concurrentes

L'interface de ces promesses un peu plus concurrentes est définie par :

```
module type  MONADREF_PLUS =
  sig
    include MONADREF
    val tryRead : 'a mref -> 'a option m
    val parRead : 'a mref -> 'b mref -> ('a,'b) sum m
  end
```

avec le constructeur de type somme défini par :

```
type ('a ,'b) sum =
  | Left of 'a
  | Right of 'b
let toLeft a = Left a
let toRight b = Right b
```

où la fonction `parRead` doit renvoyer, soit la valeur retournée par l'action terminant la première, soit n'importe laquelle des deux valeurs retournées si les deux actions sont déjà terminées, ou terminent en même temps. Grâce à la notion de `MVar`, nous pouvons instancier cette interface comme une extension de notre monade IO en posant :

```
module IO_Ref_Plus : MONADREF_PLUS =
  struct
    include IO_Ref
    let tryRead v =
      IO.return (MVar.tryRead v)
    let rec parRead  pl pr  =
      let v = MVar.createEmpty () in
      let tryWrite (p,f) =
        MVar.tryPut v (f (IO.run (read p)))
      in (ignore (Thread.create tryWrite (pl,toLeft));
          ignore (Thread.create tryWrite (pr,toRight));
          read v)
  end
```

5.2 Fusion asynchrone de flots monadique

La possibilité offerte par la fonction `parRead` de trier des actions référencées par ordre de terminaison nous conduit à notre première définition de fonctions asynchrones : la fusion temporelle

de deux flots monadiques. Plus précisément, on peut définir de façon générique :

```
module StreamRef_Plus  (M : MONADREF_PLUS)
  =
  struct
    include StreamRef(M)

    let fmapStream = fmap
    open M
    let rec mergeStreamRef (StreamRef r1) (StreamRef r2)
      = Stream ((parRead r1 r2) >>= function
          | Left None -> fromStream (readStream (StreamRef r2))
          | Left (Some(a1,sc)) -> return (Some (a1,
             mergeStreamRef sc (StreamRef r2)))
          | Right None -> fromStream (readStream (StreamRef r1))
          | Right (Some(a2,sc)) -> return (Some (a2,
             mergeStreamRef (StreamRef r1) sc)))
    let rec merge s1 s2
      = toStream (forkStream s1 >>= fun r1 ->
                  forkStream s2 >>= fun r2 ->
                  return (mergeStreamRef r1 r2))
    let mergeStream s1 s2
      = merge (fmapStream toLeft s1) (fmapStream toRight s2)
  end
```

La fonction `merge : 'a s -> 'a s -> 'a s` prend deux flots monadiques en argument et les fusionne, par ordre d'arrivée des valeurs qu'ils portent, pour produire le flux monadique résultant de cette fusion.

Cette fonction `merge` est l'archétype de ce qu'on pourrait appeler une fonction asynchrone, puisque l'horloge du flot de sortie est égale à l'union des horloges des flots d'entrées, certains ticks pouvant être dupliqués en cas de réceptions synchrones. Elle est à mettre en regard de la fonction `zip` qui, tout au contraire, produit un flot dont l'horloge est égale aux horloges des flux d'entrées, quitte à « forcer » l'égalité des horloges des flux d'entrées en prenant leur borne supérieure.

On peut vérifier que la fusion induit une structure de monoïde commutatif sur les flots monadiques, le flot vide défini par `emptyStream = Stream (return None)` servant de neutre [3]. A partir de ce merge, on peut aussi définir une monade avec le constructeur de type flot monadique, la fonction `bind s f` remplaçant chaque élément `a`, porté et placé dans le temps par le flot `s`, par son image `f a`, ces flots images étant alors fusionnés à la volée.

La fonction `merge` permet aussi de coder `readAll : 'a mref list -> 'a stream` qui, appliquée à une liste de références monadiques, produit le flot des valeurs retournées par les actions référencées, triées temporellement. Une telle fonction est très utile pour faire, par exemple, un « foldmap » concurrent. Néanmoins, ce codage, uniforme, est de complexité quadratique en nombre d'appel à `parRead`. Dans la monade IO on pourra donc lui préférer une version lineaire obtenu simplement en généralisant le code de `parRead`. Plus généralement, la fonction `parRead` dérivant facilement de toute implémentation de la fonction `readAll`, on pourra lui préférer cette

3. Techniquement, le flot vide défini en OCaml n'est que faiblement polymorphe. On peut remédier à cela en spécifiant le constructeur de type monadique `m` comme covariant, mais cela nous interdit alors de définir des monades via des structures mutables tels que les `MVar`.

dernière dans la signature `StreamRef_Plus`.

Le cœur de notre expérimentation de programmation GALS, combinant le contrôle asynchone de traitement audio synchrone [7] peut alors être définit de la façon suivante :

```
module AStream (M : MONADREF_PLUS) =
  struct
    include StreamRef_Plus(M)
    open M
    let rec asyncMapRef f (StreamRef mf) s =
      let applyAndCont f sf (Stream m) = m >>= (function
        | Some(a,sc) -> return (Some (f a , asyncMapRef f sf sc))
        | None -> return None) in
      Stream (tryRead mf >>= (function
        | None -> applyAndCont f  (StreamRef mf) s
        | Some (None) -> fromStream (fmapStream f s)
        | Some (Some(f1 , scf)) -> applyAndCont f1 scf s))
    let  asyncMap f sf s = toStream
       (forkStream sf >>= fun r -> return(asyncMapRef f r s))
  end
```

La fonction `asyncMap : ('a -> 'b)-> ('a -> 'b)s -> 'a s -> 'b s` reçoit en effet, de façon asynchrone, un flot de fonctions de type `'a -> 'b`, et applique de façon synchrone la dernière fonction arrivée (où la première donnée en paramètre) à son dernier paramètre, une flot de type `'a s`.

6 Performances comparées de promesses de flots

La fonction d'écho et ses variantes nous permettent de tester les performances de notre implémentation décrite ici et de la comparer à celle déjà réalisée en Haskell. En Haskell nous utilisons, aux choix, les threads léger de sa bibliothèque concurrente ou les thread légers du système unix. Dans la version OCaml proposée ici, nous n'utilisons que les threads légers Unix. Une autre version, dont nous rapportons aussi les tests, utilise les threads légers de la bibliothèque `lwt`.

	type thread	echo	echoR	echo2RZ
Haskell	Light	375 kHz	21 kHz	16.5 kHz
Haskell	System	375 kHz	4.9 kHz	4.8 kHz
OCaml	System	653 kHz	7.34 kHz	7.3 kHz
OCaml	Lwt	653 kHz	90 kHz	72 ,7 kHz

Nos tests de performance, effectué sur un ordinateur portable standard, consistent à faire un « pipe » unix d'un programme énumérant les entiers et d'observer, après 20 seconde, le nombre d'entiers traités par les programmes d'échos testés, rapporté à une seconde. Trois programmes d'échos sont testés, qui effectuent, selon le cas, un echo simple (`echo`) sans threads, un echo avec une promesse du flot d'entrée et une seul lecture (`echoR`) et, enfin, un echo avec une promesse du flot d'entrée et deux lectures zippées (`echo2RZ`). Les codes sont compilés avec `ghc -O3` pour Haskell et `ocamlopt -O3 I +threads` pour OCaml.

On constate, peut-être sans surprise à cause de l'évaluation paresseuse en Haskell, que les versions OCaml sont sensiblement plus rapides que les versions Haskell, les version avec threads légers étant aussi plus performantes que les versions avec threads systèmes. Remarquons cependant que la bibliothèque `lwt` offre bien moins de service que la librairie concurrente d'Haskell.

7 Conclusion

La programmation concurrente asynchrone a pour avantage, par rapport à la programmation concurrente généralisée, d'interdir par construction tout programme avec interblocages. Son applicabilité s'en trouve limitée ainsi limité. Néanmoins, combinée avec la programmation par flots monadiques et la notion de promesses de flots qui en découle, ces limites restent confortable pour de nombreuses applications. Notre étude permet ainsi d'identifier une première série de *primitives* asynchrones qui offrent, tout à la fois, la souplesse de la concurrence et l'assurance de programmes sans interblocages.

Cette étude n'est cependant pas terminée. En effet, il serait intéressant de pouvoir typer les actions monadiques comme étant bloquantes (actions d'entrée) ou non bloquantes (actions de traitement ou de sortie) afin de développer un système de type nous permettant de vérifier la cohérence temporelle des fonctions sur les flots monadiques. Des types modaux [9] pourraient être utilisés pour cela. La programmation OCaml est aussi, avec ses effets de bord, une programmation implicitement de type monadique. Il pourrait être interessant d'assumer ce fait en intégrant toutes les interfaces proposées ici au top-level de la programmation OCaml, en évitant ainsi le recours à des appels explicites de `return` et `bind`.

Références

[1] S. Archipoff and D. Janin. Structured reactive programming with polymorphic temporal tiles. In *ACM Work. on Functional Art, Music, Modeling and Design (FARM)*. ACM Press, 2016.

[2] S. Archipoff and D. Janin. Unified media programming : An algebraic approach. In *ACM Work. on Functional Art, Music, Modeling and Design (FARM)*. ACM, 2017.

[3] C. Deleuze. Concurrence légère en ocaml : muthreads. In *Journées Francophones des Langages Applicatifs (JFLA)*, 2013.

[4] Robert H. Halstead, Jr. Multilisp : A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4) :501–538, 1985.

[5] P. Hudak, J. Hugues, S. Peyton Jones, and P. Wadler. A history of Haskell : Being lazy with class. In *Third ACM SIGPLAN History of Programming Languages (HOPL)*. ACM Press, 2007.

[6] D. Janin. An equational modeling of asynchronous concurrent programming. Technical report, LaBRI, Université de Bordeaux, 2019.

[7] D. Janin. Screaming in the IO monad. In *ACM Work. on Functional Art, Music, Modeling and Design (FARM)*. ACM, 2019.

[8] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *Principles of Programing Languages (POPL)*, New York, 1996. ACM.

[9] N. R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. In *Int. Conf. Func. Prog. (ICFP)*, 2013.

[10] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system, release 4.08, Documentation and user's manual*. Inria, 2019.

[11] S. Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly, 2013.

[12] Y. Minsky, J. Hickey, and A. Madhavapeddy. *Real World Ocaml : Functional programming for the masses*. O'Reilly, 2013.

[13] E. Moggi. A modular approach to denotational semantics. In *Category Theory and Computer Science (CTCS)*, volume 530 of *LNCS*. Springer-Verlag, 1991.

[14] P. Wadler. Comprehending monads. In *Conference on LISP and Functional Programming (LFP)*, New York, 1990. ACM.