Heterogeneous-Arrow : modéliser le rythme du *Data-Flow*

Modélisation du rythme de traitement et du changement de structure des flux dans un contexte *Data-Flow*

Gautier DI FOLCO

INRIA, équipe DICE, gdifolco@ens-lyon.fr

Résumé Le *Data-Flow* est un outil de programmation permettant d'exprimer des flux de données circulant entre des entités d'exécutions.

Après avoir présenté ses usages et ses enjeux, nous montrons, à l'aide d'un cas pratique, une faiblesse dans le modèle actuel qui réside dans la difficulté de composer des contextes de calculs différents. Nous fournissons une généralisation des `Arrow`s en Haskell.

1 Introduction

Le *Data-Flow* modélise le "continu" dans les programmes informatiques. Ce qui permet par exemple de décrire des animations graphiques sous l'aspect de formules mathématiques sans gérer la manière dont elles seront interprêtées. Les formules étant exprimées selon des variables continues, elles seront ensuite automatiquement discrétisées pour le rendu visuel. Du fait que le continu est représenté par une infinité de valeurs entre deux bornes, il en découle une notion d'infini, les bornes étant inacessibles. Ce qui force le calcul sur des données incomplètes entrainant des calculs partiels.

Cette notion d'incomplétude des données est utilisée pour répondre à un problème de charge de traitements. Par exemple l'afflux de données provenant de l'Internet a tellement augmenté qu'il devient trop coûteux en ressources de lancer un calcul donné sur l'ensemble des données. De plus, lorsqu'on modélise un flux, il est déjà existant, donc nous n'avons pas accès au début et le flux n'a pas de fin. Ainsi des calculs partiels sont effectués et complétés au fur et à mesure de la réception de nouvelles données.

Notre objectif est de proposer des outils pour manipuler des flux ayant des rythmes et des structures différents explicitement modélisés.

2 Historique

Nous exposons l'historique des langages *Data-Flow* afin de mettre en avant l'importance du rythme dans les systèmes *Data-Flow*.

Strachey [16] décrit l'état du monde de la programmation informatique, en opposant deux paradigmes : la programmation fonctionnelle et impérative.

L'une des premières implantation du *Data-Flow* [6] réalisée est présentée comme une généralisation de LISP, un langage fonctionnel.

Le *Data-Flow* est défini par Dennis et al. [6] comme un modèle dont l'exécution est déclenchée par la disponibilité des données. Il y a une dépendance explicitement modélisée du code vis-à-vis des données.

2.1 Réseaux de Kahn

Kahn [12] fournit un langage pour effectuer des calculs en parallèle, où les programmes sont découpés en entités communiquant entre elles via des *channel*s, qui sont des listes *FIFO* infinies ayant une taille de tampon fixe.

2.2 Prémisses de la modélisation du continu avec le *Flow-based programming*

Lucid [2] modélise l'infini en se basant sur les suites mathématiques, c'est-à-dire par une valeur de départ et un code pour calculer les valeurs suivantes.

Une notion de temps a été ajoutée au flux par Esterel [4]. Les programmes sont proches des filtres UNIX : ils n'ont qu'une entrée auxquels ils réagissent sur leur sortie.

Les flux sont considérés comme un ensemble et plus comme un flux unique dans Signal [3]. Ce qui permet, comme dans StreamIT [17], de mettre en relation des flux pour en créer de nouveaux.

Les programmes sont des graphes acycliques orientés, où les noeuds représentent des algorithmes élémentaires et les arcs les dépendances sur les données, comme dans Lustre [5].

2.3 L'avènement des environnements interactifs avec le *Reactive-based programming*

TBAG [10] introduit la notion de variation temporelle. De manière graphique l'utilisateur va définir un ensemble de relations entre les objects qui vont varier au cours du temps. TBAG se charge d'évaluer ces relations pour des valeurs temporelles significatives afin que l'animation semble avoir été calculée depuis un flux continu.

Par la suite, MediaFlow [9] va permettre la combinaison visuelle de flux continus.

Functional Reactive Animation (Fran) [8] [7] s'appuie sur une spécification programmatique des relations entre des objets en fonction du temps. Il effectue ensuite son rendu graphique dynamiquement.

Fran est le point de départ de nombreux papiers comme RT-FRP [19] qui ajoute un aspect temps-réel ou Nilsson et al.[15] qui se base sur la déclaration de l'enchaînement des traitements.

Le *Functional reactive programming* considère, comme dans André [1], les programmes comme une réaction à un ensemble de signaux. Un signal perçu déclenche l'exécution d'un comportement, le programme réagit à un principe de stimuli.

2.4 Conclusion

La hausse du volume de données générées, échangées, traitées et stockées augmente de plus en plus.

Notre hypothèse est que les moyens de traitement et de stockage ne sont plus suffisants.

Nous devons mettre en place des mécanismes pour permettre de diminuer la charge de traitements à effectuer.

Le choix que nous avons fait est de considérer que l'information n'est pas complète. Nous n'en possédons pas la fin, et nous ne possédons pas souvent le début.

Ainsi, nos calculs sont effectués sur une information incomplète, ce qui nous donne un calcul partiel. Lorsque nous avons une nouvelle information, nous reprenons le résultat précédemment calculé pour le compléter avec la nouvelle donnée.

Il y a une diminution du nombre de calculs, puisque ceux-ci ne sont effectués que partiellement.

Rythme et nature du flux Nous appelons nature d'un flux la méthode de construction des données sortantes. Par exemple, le fait d'aggréger les informations d'un flux à un autre change sa nature car nous obtenons une donnée qui dépend de son historique à partir d'une donnée unitaire.

Le rythme est modifié si, par exemple, le calcul est lancé uniquement toutes les N nouvelles données, le flux de sortie aura N fois moins de messages que le flux d'entrée.

3 Modélisation du rythme et de la structure des flux

3.1 Cas pratique : box office

Nous disposons d'un flux de tickets de cinéma. Un ticket est identifié par un nom de film.

Nous souhaitons avoir un premier flux contenant une liste de films ordonnés par nombre d'entrées décroissant, obtenu via un noeud `genRanking`. Ce qui implique un changement de nature du flux. Le rythme du flux ne change pas puisque la liste est mise à jour à chaque nouveau ticket.

À partir de ce flux, nous voulons limiter le débit en le divisant par 3 pour que la donnée puisse être affichée de manière stable, obtenu via un noeud `stabilize`. Nous opérons un changement de rythme.

3.2 Mise en oeuvre avec Haskell

Haskell [14] est un langage purement fonctionnel non-strict. Nous l'avons choisi car il est le support de nombreuses approches à la fois *Reactive-based programming* et en *Functional reactive programming* et qu'il dispose d'un système de types assez avancé pour modéliser notre domaine d'étude.

Modélisation sans notion de rythme La méthode classique pour gérer le *Data-Flow* en Haskell sont les `Arrows` [11]. C'est une spécialisation des `Category` qui définissent la notion de composition sous forme de *typeclass*. Un *typeclass* représente la capacité à appeler un ensemble de fonctions sur un ensemble de type de données. Il s'agit d'un mécanisme proche des *template*s spécialisées du C++.

En Haskell, tout est expression, le type d'expressions le plus courant est le type fonction noté `->`. Il est fourni par le langage. Sa pseudo déclaration est la suivante :

```
1 data i -> o = ...
```

Il a un paramètre de type `i` en entrée et retourne une valeur de type `o`. Le type d'une fonction `f` attendant un entier, noté `Int`, et produisant un caractère noté `Char` sera noté :

```
1 f :: Int -> Char
```

Les types de données sont nommés par une lettre commençant en majuscule. Les types de données commençant en minuscule sont des types qui seront déterminés en fonction de leur(s) usage(s), comme les *template*s C++. Lorsque deux types commençant en miniscule sont les mêmes, le type assigné à l'usage sera le même.

```
1 f :: a    -> a    -- Definition
2 f :: Int  -> Int  -- Legal
3 f :: Char -> Char -- Legal
4 f :: Char -> Int  -- Illegal
```

Pour qu'une fonction prennent plusieurs arguments, il faut que, conceptuellement, à chaque nouvel argument appliqué, elle retourne une nouvelle fonction. Ainsi, les fonctions suivante ont le même type :

```
1 f :: a -> b -> c -> d -> e
2 g :: a -> (b -> (c -> (d -> e)))
```

On parle de currification ou *currying*.

Ici nous déclarons le *typeclass* `Category`, qui représente la notion de composabilité.

```
1 class Category cat where
2   id :: cat a a
3   (.) :: cat b c -> cat a b -> cat a c
```

Pour un type arbitraire `cat`, une fonction et un opérateur lui est associés.

La fonction `id`, qui représente l'identité, c'est à dire que `cat` a ses deux paramètres de même type, le type `a`.

L'opérateur de composition `(.)` qui crée une `Category` ayant pour types paramétriques `a c` en redirigeant la sortie d'une `Category b c` vers l'entrée d'une `Category a b`.

Les fonctions classiques sont nommées avec des caractères alphanumériques et commencent toujours par une minuscule. Les opérateurs ne sont composés que de caractères non-alphanumériques.

La différence entre les deux est visible au niveau de la position par défaut lors de l'appel. Les fonctions classiques ont une position préfixe :

```
1 add 4 5
```

ici, `add` est une fonction classique, elle est donc placée devant ses arguments.

Les opérateurs ont une position infixe :

```
1 4 + 5
```

ici, `+` est un opérateur, il est donc placé entre ses deux premiers arguments.

Nous pouvons changer la position des fonctions classiques et des opérateurs en les entourant, respectivement, de *back-quotes* (‘) et de parenthèses. Ainsi, voici l'équivalent des deux exemples précédents en inversant les positions :

```
1 4 ‘add‘ 5
2 (+) 4 5
```

Cette distinction est importante car, lorsque nous mentionnons une fonction en position préfixe et que nous omettons un ou plusieurs de ses dernier arguments, une fonction est automatiquement créée, grâce au *currying*. Cette fonction aura pour paramètres, les paramètres omis. On parle d'application partielle. Comme nous l'avons vu plus haut, de manière plus général, en Haskell, toutes les fonctions n'attendent qu'un argument et ne renvoient qu'une valeur. Ainsi, conceptuellement, lorsqu'une fonction attend plusieurs arguments, lors de chacun de ses appels, une nouvelle fonction est retournée :

```
1 add :: Int -> Int -> Int
2 add 4 :: Int -> Int
3 add 4 5 :: Int
```

Alors que les fonctions en position préfixes créent automatiquement des fonctions lorsqu'elles ne sont que partiellement appliquées, si un des deux premiers arguments manquent à un opérateur, une erreur de syntaxe est levée. Les exemples suivants ne sont pas syntaxiquement valides :

```
1 ‘add‘
2 ‘add‘ 5
3 4 ‘add‘
4 +
5 + 5
6 4 +
```

Pour les rendres valides, il faut les entourer de parenthèses :

```
1 (‘add‘)
2 (‘add‘ 5)
3 (4 ‘add‘)
```

```
4 (+)
5 (+ 5)
6 (4 +)
```

C'est la raison pour laquelle, lorsque l'on déclare le type d'un opérateur, comme l'opérateur de composition `(.)`, nous sommes obligés de le parenthéser, puisque nous ne lui passons aucune valeur.

Un exemple d'`instance` de `Category` concerne le type des fonctions. Le type fonction est un type à part entière noté `(->)`. Ainsi, à partir de la déclaration du *typeclass* `Category`, nous remplaçons toutes les occurences de `cat` par `(->)`.

```
1 instance Category (->) where
2 -- Version prefixe : id :: (->) a a
3 -- Version infixe  : id :: a -> a
4   id x = x
5
6 -- Version prefixe : (.) :: (->) b c -> (->) a b -> (->) a c
7 -- Version infixe  : (.) :: (b -> c) -> (a -> b) -> (a -> c)
8   f . g = \x -> f (g x)
```

`id`, ligne 4, attend un argument `x` et le retourne. Le type de `x` est quelconque, noté `a`, et le type de retour de `id` est le même que le type de son paramètre, soit `a`.

`(.)`, ligne 8, attend deux fonctions pour en produire une troisième. Son premier argument, `f`, est une fonction attendant un type quelconque noté `b` et renvoyant un type quelconque `c`, le tout est noté `b -> c`. Son second argument, `g`, est une fonction attendant un type quelconque noté `a` et renvoyant un type quelconque `b`, le tout est noté `a -> b`. `(.)` crée une *lambda expression*, une sorte de fonction anonyme, via `\x ->` attendant un paramètre `x`. Il applique ce paramètre à la fonction `g`, via `g x`. Puis le résultat de cette application sera appliqué à la fonction `f`. Par conséquent, la fonction résultante aura un paramètre du type de celui de la fonction `g`, ie. `a`, et un type de retour de sa fonction `f`, ie. `c`. Donc, c'est une fonction qui prend deux fonctions, l'une prenant un `a` et renvoyant un `b`, l'autre prenant un `b` et renvoyant un `c`. Elle renvoie une fonction prenant un `a` et renvoyant un `c`, qui est bien la notion de composition.

Par exemple, si nous disposons de deux fonctions `add3` et `prod2`, qui, respectivement, ajoute `3` à un nombre et le multiplie par `2`, nous pouvons les combiner de la manière suivante :

```
1 add3 :: Int -> Int
2 add3 x = x + 3
3
4 prod2 :: Int -> Int
5 prod2 x = x * 2
6
7 prod2Add3 :: Int -> Int
8 prod2Add3 = add3 . prod2
```

Ainsi `prod2Add3 1` renvoie `5`. Nous aurions pu l'ecrire `prod2Add3 x = prod2 (add3 x)`. En fait il se passe le mécanisme de substitution, appelé évaluation, suivant :

```
1  prod2Add3 = add3 . prod2
2  -- f . g = \x -> f (g x)
3  -- f est remplace par add3 et g par prod2
4  prod2Add3 = \x -> add3 (prod2 x)
5  -- lorsque 1 est applique a prod2Add3 :
6  prod2Add3 1
7  (\x -> add3 (prod2 x)) 1
8  -- x est remplace par 1
9  add3 (prod2 1)
10 add3 (1 * 2)
11 add3 2
12 2 + 3
13 5
```

Pour représenter notre cas pratique, nous modélisons la notion de noeud de calcul par un type de données `Node`.

```
1  data Node a b = Node { process :: a -> b }
```

Il contient une fonction transformant un `a` en un `b`, ces types étant les types paramétriques de `Node`.

Le mot clef `data` permet de définir de nouveaux types de données. Il est suivi de l'identifieur de type, qui commence toujours par une lettre majuscule lorsqu'il est alphanumérique, ici `Node`. Il peut être ensuite suivi d'un ou plusieurs paramètres de types, ici `a` et `b`. L'identifieur de type est utilisé uniquement dans les déclarations de types.

Ensuite, après le `=` vient le ou les constructeurs, ici `Node`. Ils suivent les mêmes conventions de nommage que les identifieurs de types, mais l'identifieur de type et le constructeur n'ont pas obligatoirement le même nom. Un construteur peut attendre des valeurs ou non, ici il attend une fonction de type `a -> b`. `process` est une fonction créé automatiquement, elle a le type `Node -> a -> b`.

`Node` peut se manipuler de la manière suivante :

```
1  -- Pour : f :: Int -> Char
2  Node f :: Node Int Char
3
4  process' :: Node a b -> a -> b -- equivalent de process
5  process' (Node f) = f
```

Nous définissons ensuite les fonctions nécessaires pour que `Node` soit une `instance` du *typeclass* `Category`, ligne 2 et 3.

```
1  instance Category Node where
2      id = Node id
3      (Node g) . (Node f) = Node (g . f)
```

Nous n'avons pas besoin d'indiquer le type des fonctions, Haskell le déduit automatiquement via *typeclass* qu'il implante. `id`, ligne 2, est un appel au constructeur `Node` avec la fonction générique `id`. Ainsi le type de cette fonction est `Node a a`.

L'opérateur de composition, ligne 3, extrait par déconstruction, filtrage par motifs ou *pattern matching*, les valeurs du constructeur de `Node` de chacun de ses argument. `g` et `f` sont deux fonctions qui sont recomposées au sein d'un nouveau `Node`.

Ensuite viennent les types plus spécifiques à notre problème :

```
1 data Rank = Rank { title :: String, visitors :: Int }
      deriving (Show, Eq)
2 type Ticket = String
3 type Counter = Int
4 type Ranking = [Rank]
```

Ici, nous définissons un type `Rank`, ligne 1, composé d'une chaine de caractère, `String`, et d'un entier, `Int`. Nous demandons au compilateur d'instancier automatiquement deux *typeclass* : `Show`, pour pouvoir afficher les valeurs et `Eq`, pour faire des comparaisons entre deux valeurs. Le compilateur est en mesure de déduire les implantations naives pour un certain nombre de *typeclass*.

Puis, trois alias de types : `Ticket`, `Counter` et `Ranking` qui sont, respectivement, les alias de `String`, `Int` et de `[Rank]`. Les alias ne font qu'améliorer la lisibilité des types, ils n'influent pas sur la compilation ou l'exécution.

`[Char]` représente une liste de `Char`. Il est défini de la manière suivante :

```
1 data [] a = []
2           | a : [a]
```

Le type liste est composé de deux constructeurs, séparés par un `|`. Le premier, ligne 1, `[]`, ne prend aucun valeur et représente l'élément liste vide. Le second, ligne 2, est l'opérateur `(:)`, qui attend deux arguments : le premier est une valeur de type `a` et le second est une autre liste de type `a`, noté `[a]`. Il s'agit d'un type récursif puisqu'un de ses constructeurs a un de ses paramètres de son propre type.

Nous implémentons un noeud `stabilize` qui limite le débit d'un flux. Pour cela nous avons besoin d'une valeur quelconque et d'un compteur en entrée, nous utilisons une paire `a, Int`. Le noeud produit une nouvelle valeur pour le compteur et peut-être une valeur, en fonction de l'état de son compteur.

La possibilité de présence ou de l'absence d'une valeur est représenté par le type suivant :

```
1 data Maybe a = Nothing
2              | Just a
```

Le constructeur `Nothing`, ligne 1, représente l'absence de valeur. Tandis que `Just`, ligne 2, représente la présence d'une valeur de type `a` qu'il contient.

Nous implémentons `stabilize` de cette manière :

```
1 stabilize :: Node (a, Counter) (Maybe a, Counter)
2 stabilize = Node (\ (x, i) -> if i < 2
3                               then (Nothing, i + 1)
4                               else (Just x, 0))
```

Nous faisons appel au constructeur `Node`, ligne 2, en lui fournissant une *lambda expression*. Celle-ci possède un paramètre qui doit être une paire d'une valeur de type arbitraire nommée `x` et un `Counter` nommé `i`, soit le type `(Maybe a, Counter)`.

Si le compteur, `i`, est inférieur à 2, nous renvoyons, ligne 3, une paire dont le premier élément est `Nothing`, car nous souhaitons ne renvoyer qu'une valeur sur 3, et nous incrémentons le compteur `i`. Si non, ligne 4, nous renvoyons une paire contenant la valeur et nous remettons le compteur à 0.

L'implantation des noeuds `genRanking` qui met à jour un classement nécessite un classement et un ticket et un classement pour générer un nouveau classement. Comme pour `stabilize`, `genRanking` attend une paire en entrée, comprenant un `Ticket` et un `Ranking` et produira un `Ranking`.

```
1 genRanking :: Node (Ticket, Ranking) Ranking
2 genRanking = Node (uncurry updateRanking)
```

`uncurry` est une fonction qui transforme une fonction attendant deux éléments en fonction attendant une paire.

`updateRanking` attend un `Ticket t` et un `Ranking r`.

```
1 updateRanking :: Ticket -> Ranking -> Ranking
2 updateRanking t r = case b of
3                       []          -> a ++ [Rank t 1]
4                       (x : xs)    -> sortBy (flip compare
                            `on` visitors) (Rank t (1 +
                            visitors x) : a ++ xs)
5     where (a, b) = break (\ (Rank i _) -> i == t) r
```

La fonction `break` est appelée à la ligne 5. Son rôle est de, à partir d'une fonction renvoyant un booléen en fonction d'un élément de son second paramètre, une liste, ici les `Ranks r`, parcourir sa liste, de gauche à droite et lorsque la fonction passée renvoie `True`, générer une paire de tableau dans lequel son premier membre sera l'ensemble des éléments déjà parcourus et son membre de gauche sera le tableau contenant l'élement courant et le reste de la liste. La fonction donnée à `break` renvoie `True` lorsque le titre du film `i` de l'élément courant est égale au titre du film sur le `Ticket t`. Le résultat de `break` est séparé en `a` et `b`.

Nous distinguons deux cas pour `b`.

S'il est vide, ligne 3, `[]`, c'est que c'est le premier `Ticket` pour un film. Dans ce cas, nous ajoutons le film avec une entrée à la fin du `Ranking`.

Si non, ligne 4, nous extrayons le premier élément de `b` dans `x` et le reste de la liste dans `xs`. `x` est le `Rank` correspondant au film de notre `Ticket`. Nous ajoutons la première partie de la liste `a`, au reste de la liste `xs` au `Rank` incrémenté, puis nous trions le tout pour obtenir un `Ranking` ordonné.

Si nous souhaitons composer ces deux noeuds nous sommes dans l'incapacité de le faire car le type de sortie de `genRanking` n'est pas le même que le type d'entrée de `stabilize` :

```
1 genRanking :: Node (Ticket, Ranking) Ranking
2 stabilize :: Node (a, Counter) (Maybe a, Counter)
```

Les `Arrows` [11] ont été introduits pour pouvoir combiner de manière standard des fonctions au sein d'une `Category`. Il s'agit d'une spécialisation des `Category`. Les `Arrows` sont définis en Haskell via le *typeclass* suivant :

```
1 class Category a => Arrow a where
2     arr :: (b -> c) -> a b c
3     first :: a b c -> a (b, d) (c, d)
4     second :: a b c -> a (d, b) (d, c)
5     (***) :: a b c -> a b' c' -> a (b, b') (c, c')
6     (&&&) :: a b c -> a b c' -> a b (c, c')
```

Ici la fonction `arr` construit un type de données `a b c` à partir d'une fonction `b -> c`. Typiquement il s'agit de stocker la fonction dans un type de données.

`first` et `second` changent le type d'entrée et de sortie en paire ayant un membre supplémentaire. Cette valeur a un type quelconque et sera simplement recopiée lors de l'évaluation du calcul. Cette valeur se situe en seconde position de la paire pour `first` et en première position de la paire pour `second`.

`***` aggrége deux calculs qui seront effectués sans interaction l'un avec l'autre.

`&&&` crée un calcul qui applique un même argument à deux calculs ayant le même type d'entrée et retournera ensuite leur résultat respectif dans une paire.

De plus deux opérateurs de composition sont introduits :

```
1 (>>>) :: Category cat => cat a b -> cat b c -> cat a c
2 (<<<) :: Category cat => cat b c -> cat a b -> cat a c
```

`<<<` est un alias de l'opérateur de composition `(.)`. `>>>` est aussi un alias mais dont les paramètres ont été inversés. Ce sont des fonctions qui dépendent d'un *typeclass*, mais qui n'en font pas partie, elles n'ont qu'une définition et ne sont pas spécialisables par type.

Nous définissons ensuite les fonctions nécessaires pour que `Node` soit une `instance` du *typeclass* `Arrow`, ligne 2 et 3. Toutes les fonctions n'ont pas besoin d'être définies car certaines d'entre-elles disposent d'une implantation par défaut, qui repose sur d'autres fonctions du *typeclass*, dans la définition du *typeclass*.

```
1 instance Arrow Node where
2     arr = Node
3     first (Node f) = Node (\ ~(b, c) -> (f b, c))
```

`arr`, ligne 2, appel le constructeur de `Node`. Par currification, cela revient à écrire `arr x = Node x`.

`first`, ligne 3, extrait la fonction `f` de `Node`. Il crée un nouveau `Node`, dont la fonction est définie par une *lambda expression*. Celle-ci attend une paire et en

renvoie un dont le second membre, `c`, sera inchangé, mais dont le premier, `b`, se voit appliqué à la fonction `f`.

Ainsi, si nous appliquons `first` à `genRanking`, nous obtenons un type de sortie compatible au type d'entrée de `stabilize` :

```
1 genRanking :: Node (Ticket, Ranking) Ranking
2 first genRanking :: Node ((Ticket, Ranking), a) (Ranking, a)
3 stabilize :: Node (a, Counter) (Maybe a, Counter)
```

Nous combinons ensuite ces noeuds pour n'en avoir qu'un seul, via la fonction de composition `>>>` :

```
1 stabilizedRanking :: Node ((Ticket, Ranking), Counter)
      (Maybe Ranking, Counter)
2 stabilizedRanking = first genRanking >>> stabilize
```

Cette implantation pose problème car elle ne permet pas de construire un classement complet. `genRanking` voit sa sortie non-conservée si `stabilize` produit `(Nothing, _)`. `genRanking` est donc limité par `stabilize`.

Il faudrait être en mesure de stocker un état qui conserverait le dernier classement généré. Malheureusement Haskell étant un langage fonctionnel pur, cela signifie qu'il dispose de la transparence référentielle, ainsi, toute fonction appelée avec les mêmes arguments donnera toujours le même résultat.

Ainsi, un mécanisme nommé `Monad` [18] a été mis en place. Les `Monad`s sont définies en Haskell via le *typeclass* suivant :

```
1 class Monad (m :: * -> *) where
2   (>>=) :: m a -> (a -> m b) -> m b
3   (>>) :: m a -> m b -> m b
4   return :: a -> m a
5   fail :: String -> m a
```

Ici on se place dans le cas d'un type de données `m` avec un paramètre `m :: * -> *`.

Par exemple la fonction `return` attend un paramètre de type quelconque `a` et l'encapsule, transmet cette valeur à un constructeur, dans un type de donnée `m` paramétré par le type de la valeur de son argument `a`, il aura donc pour type de retour `m a`.

L'opérateur important ici est `>>=`, il attend une valeur de type `a` dans un contexte `m`, soit le type `m a`, puis une fonction ayant un paramètre de type `a` et renvoyant une valeur de type `b` encapsulée dans le même contexte `m` que le premier arguement de l'opérateur, la fonction attendue retourne une valeur de type `m b`. L'opérateur retourne ensuite une valeur de type `m b`. L'implantation naive consiste à renvoyer le retour de la fonction attendue à laquelle on a transmis la valeur de type `a` désencapsulée de son contexte `m a`.

Par exemple prenons le type de données `Maybe` vu plus haut dont la définition est la suivante :

```
1 data Maybe a = Nothing | Just a
```

Il nous faut ensuite définir l'`instance Maybe` du *typeclass* `Monad` :

```
1 instance Monad Maybe where
2     Just x  >>= f = f x
3     Nothing >>= _ = Nothing
4
5     return x = Just x
6
7     a >> b = a >>= \_ -> b
8
9     fail _ = Nothing
```

Les lignes 2 et 3 définissent le corps de `>>=`. Par *pattern matching*, pour le constructeur `Just`, ligne 2, la valeur de retour sera la valeur de retour de la fonction `f` à laquelle la valeur contenue par le constructeur, `x`, lui aura été appliquée. Pour le constructeur `Nothing`, ligne 3, on ignore de le second argument `\_` et on renvoie `Nothing`.

`return`, ligne 5, prend une valeur et la passe au constructeur `Just` et le revoie.

`>>`, ligne 7, appel `>>=` en lui passant son premier paramètre puis en créant une *lambda expression* qui ignore son paramètre d'entrée et renvoie le second paramètre de `>>`.

`fail`, ligne 9, ignore son paramètre et renvoye `Nothing`.

Prenons l'exemple de cette fonction qui pour un entier donné, l'incrémente jusqu'à 3 puis renvoie `Nothing`.

```
1 addUntil3 :: Int -> Maybe Int
2 addUntil3 x = if x < 3
3               then Just (x + 1)
4               else Nothing
```

Notre objectif est de pouvoir chaîner les appels à cette fonction. Si nous faisons `addUntil3 1` nous obtenons `Just 2` du type `Maybe Int`. Impossible alors d'appeler directement la fonction sur ce résultat. Nous sommes obligés de sortir `2` de son contexte. C'est le rôle de `>>=`. Ainsi, `addUntil3 1 >>= addUntil3` renverra `Just 3`. De plus, si nous l'appliquons encore, elle renverra toujours `Nothing`, sans que `addUntil3` soit évaluée.

Nous pouvons réécrire `addUntil3' 1 >>= addUntil3` en `Just 1 >>= addUntil3 >>= addUntil3` puisque `>>=` extrait `1` de `Just`.

Si nous regardons l'implantation de l'interface, `Just 1 >>= addUntil3 >>= addUntil3` est équivalent à `return 1 >>= addUntil3 >>= addUntil3` et `addUntil3` peut être réécrit de cette manière :

```
1 addUntil3' :: (Monad m) => Int -> m Int
2 addUntil3' x = if x < 3
3                then return (x + 1)
4                else fail "too␣big"
```

Comme nous le voyons dans la signature `addUntil3'` n'est plus liée au type de donnée `Maybe`, à n'importe quelle type de données implantant le *typeclass* `Monad`.

Notre but à présent est de créer une `Monad` capable de stocker un état. Elle se nomme la *State Monad* [13] et est définie de la manière suivante :

```
1 newtype State s a = State { runState :: s -> (a, s) }
```

`State` est un type qui n'aura pas d'existence réelle lors de l'exécution, `newtype` est un mot-clef permettant d'indiquer au compilateur qu'un type existe uniquement pour distinguer certaines expressions, sans qu'elles n'aient de structure particulière. Ainsi, `State` réprésente certaines expressions qui sont des fonctions prenant un état de type `s` et renvoyant une paire avec un état de type `s` et une valeur générée de type `a`.

L'implantation de l'`instance` de la `Monad State` est la suivante :

```
1 instance Monad (State s) where
2   return v = State (\s -> (v, s))
3   State v >>= f = State (\s -> let (a, b) = v s
4                                in runState (f a) b)
```

Nous avons vu que `Monad` attendait des types ayant un et un seul paramètre. Or, `State` en possède deux. Pour être en mesure de définir l'`instance Monad`, nous utilisons la currification au niveau des types et nous fixons `s`, ligne 1, ce qui signifie que ce qui compte pour que `State` soit une `Monad`, c'est le type de son état.

`return`, ligne 2, prend une valeur, qui sera produite lors de l'exécution de la fonction représentée par `State`. Cette fonction se bornera à renvoyer l'état qui lui sera transmis.

`>>=`, ligne 3, récupère la fonction `v` de `State`. Il crée une nouvelle fonction dans `State` qui prend un argument d'état `s`, il sera appliqué à `v`, ce qui produira un couple `(a, b)` où `a` est la valeur produite et `b` le nouvel état. `let` fonctionne comme le `where` vu plus haut. Ensuite nous allons appliquer `f` à la valeur produite `a`, nous donnant un nouveau `State`. Nous appelons `runState` qui sort la fonction de `State` puis nous lui passons le nouvel etat `b`.

`State` possède également des fonctions pour rendre sa manipulation plus simple :

```
1 evalState :: State s a -> s -> a
2 evalState act = fst . runState act
3
4 execState :: State s a -> s -> s
5 execState act = snd . runState act
```

`evalState` récupère le premier membre de la paire générée par l'exécution de la `Monad`, via la fonction `fst`. `execState` récupère le second membre de la paire générée par l'exécution de la `Monad`, via la fonction `snd`.

Il faut également des fonctions pour récupérer et assigner l'état :

```
1 get :: State s s
2 get = State (\x -> (x, x))
3
```

```
4 put :: s -> State s ()
5 put x = State (\_ -> ((), x))
```

Les cas d'utilisations sont les suivants :

```
1 runState (put 1) 2 -- ((),1)
2 runState get 2 -- (2,2)
```

Nous souhaitons créer une fonction qui récupère l'état courant, l'incrémente et produise la valeur initiale de l'état ajouté à lui-même. L'implantation est la suivante :

```
1 incrIncr :: State Int Int
2 incrIncr = do
3     i <- get
4     put (i + 1)
5     return (i + i)
```

À la ligne 2, nous récupérons la valeur de l'état avec `get`, que nous représentons par `i`. Nous mettons ensuite sa valeur incrémentée dans l'état avec `put` à la ligne 3. Puis nous renvoyons l'état initial, `i`, ajouté à lui-même.

Cette notation est appelée `do`-notation et est en fait du sucre syntaxique pour :

```
1 incrIncr' :: State Int Int
2 incrIncr' = get >>= \i -> put (i + 1) >> return (i + i)
```

Les `->` sont remplacés par des `>>= \v ->` et les retour à la ligne par des `>>`.

Ainsi, `runState incrIncr 2` génère `(4, 3)`.

Nous pouvons les appliquer plusieurs fois pour que `runState incr3 2` génère `(8, 5)`.

```
1 incr3 :: State Int Int
2 incr3 = do
3     incrIncr -- (4, 3)
4     incrIncr -- (6, 4)
5     incrIncr -- (8, 5)
```

Nous pouvons réimplanter `stabilize` et `genRanking` en tant que `State Monad` :

```
1  stabilize :: a -> State Counter (Maybe a)
2  stabilize x = do
3      i <- get
4      if i < 2
5      then put (i + 1) >> return Nothing
6      else put 0 >> return (Just x)
7
8  -- runState (stabilize 'a') 0
9  -- => (Nothing,1)
10 -- runState (stabilize 'a') 3
11 -- => (Just 'a',0)
```

```
12
13 genRanking :: Ticket -> State Ranking Ranking
14 genRanking t = do
15     r <- get
16     put (updateRanking t r)
17     r' <- get
18     return r'
19
20 -- runState (genRanking "a") []
21 -- => ([Rank {title = "a", visitors = 1}],[Rank {title =
       "a", visitors = 1}])
22 -- runState (genRanking "a") [Rank "b" 1, Rank "a" 1]
23 -- => ([Rank {title = "a", visitors = 2},Rank {title = "b",
       visitors = 1}],[Rank {title = "a", visitors = 2},Rank
       {title = "b", visitors = 1}])
```

Si nous tentons de les combiner en tant que `Category`, c'est impossible car le second type de `genRanking`, `Ranking`, n'est pas le même que le premier type de `stabilize`, `Counter`.

Il en va de même pour les `Monad`. En effet `>>=` a le type suivant :

```
1 (>>=) :: m a -> (a -> m b) -> m b
```

Et nous avons vu que `State` était considéré comme une `Monad` avec son premier paramètre de type, `State s`. Ainsi `State Ranking` et `State Counter` sont deux types distincts et `>>=` nécessite deux `Monad` du même type.

La différence entre les `Arrow`s et les `Monad`s est que nous pouvons uniquement manipuler le type de sortie d'une `Monad` alors que nous pouvons manipuler le type d'entrées et le type de sortie d'un `Arrow`. C'est ce qui les rend plus composable que les `Monad`s.

Il nous faut donc combiner une `State Monad` pour conserver l'état et un `Arrow` pour la composition.

Nous obtenons un `Kielsi Arrow` [11]. `Node` devient alors inutile puisque `Kleisli` implante déjà les *typeclass* `Arrow` et `Monad`.

Ils sont déclarés de la manière suivante :

```
1 newtype Kleisli m a b = Kleisli { runKleisli :: a -> m b }
```

`Kleisli` représente donc une fonction prenant une valeur et renvoyant une valeur dans un type ayant un paramètre.

Si nous mettons côte-à-côte `runKleisli`, `genRanking` et `stabilize`, nous nous rendons compte que `Kleisli` est une généralisation.

```
1 stabilize  :: a          -> State Counter    (Maybe a)
2 genRanking :: Ticket     -> State Ranking    Ranking
3 runKleisli :: a          -> m                b
```

L'`instance Category` de `Kleisli` est définie de la manière suivante :

```
1 instance Monad m => Category (Kleisli m) where
```

```
2       id = Kleisli return
3       (Kleisli f) . (Kleisli g) = Kleisli (\b -> g b >>= f)
```

Le premier type paramétrique de `Kleisli` doit être une `Monad`. Comme nous l'avons vu pour la déclaration de l'`instance Monad` de `State`, nous fixons le premier type paramétrique pour que ça corresponde au nombre de type paramétriques attendu, ici, 2.

`id`, ligne 2, appel le constructeur de `Kleisli` avec l'équivalent monadique de la fonction identité, à savoir `return`.

La fonction de composition, `(.)`, ligne 3, extrait les fonctions `f` et `g` des constructeurs `Kleisli`. Il crée ensuite une nouvelle valeur `Kleisli` qui est composée d'une *lambda expression* attendant un argument qui sera appliqué à `g`. La valeur résultante sera une `Monad`, il faut donc la passer à `g` en utilisant `>>=`.

Comme pour l'`instance Category` de `Kleisli`, l'`instance Arrow` fixe son premier type paramétrique qui est une `Monad`.

```
1  instance Monad m => Arrow (Kleisli m) where
2      arr f = Kleisli (return . f)
3      first  (Kleisli f) = Kleisli (\ ~(b,d) -> f b >>= \c ->
           return (c,d))
4      second (Kleisli f) = Kleisli (\ ~(d,b) -> f b >>= \c ->
           return (d,c))
```

`arr` attend un argument, une fonction `f` et construit `Kleisli` en mettant dans une `Monad`, via `return`, le résultat de `f`.

`first` récupère la fonction `f` de `Kleisli` et construit un nouveau `Kleisli` à l'aide d'une *lambda expression* qui attend une pair `(b, d)`. `b` est appliqué à `f` et sa valeur est extraite de la `Monad` résultante pour être placé dans `c`. `return` encapsule une paire contenant le résultat produit, `c`, et la valeur `d` qui est inchangé.

`second` est construite comme `first` mais dans l'ordre inverse au niveau des paires.

Nous reprennons nos types précédents, sans `Node` qui est devenu inutile avec `Kleisli` :

```
1  data Rank = Rank { title :: String, visitors :: Int }
       deriving (Show, Eq)
2  type Ticket = String
3  type Counter = Int
4  type Ranking = [Rank]
```

Pour que nos noeuds soient conformes à `Kleisli`, nous encapsulons les versions monadiques dans le constructeur de `Kleisli`.

```
1  genRanking :: Kleisli (State Ranking) Ticket Ranking
2  genRanking = Kleisli (\t -> get >>= put . updateRanking t >>
       get)
3
4  updateRanking :: Ticket -> Ranking -> Ranking
5  updateRanking t r = case b of
```

```
6                           []          -> a ++ [Rank t 1]
7                           (x : xs)    -> sortBy (flip compare
                               `on` visitors) (Rank t (1 +
                               visitors x) : a ++ xs)
8       where (a, b) = break (\ (Rank i _) -> i == t) r
9
10  stabilize :: Kleisli (State Counter) a (Maybe a)
11  stabilize = Kleisli (\e -> get >>= \i ->  if i < 2
12                                            then put (i + 1)
                                                  >> return
                                                  Nothing
13                                            else put 0 >>
                                                  return (Just
                                                  e))
```

Comme nous l'avons vu, `Kleisli` prend pour premier type paramétrique une `Monad`, `State Ranking` pour `genRanking` et `State Counter` pour `stabilize`, selon qu'ils doivent maintenir l'état d'un `Ranking` ou d'un `Counter`. `Kleisli` attend ensuite, comme tout `Arrow`, un type d'entrée, `Ticket` et `a`, ainsi qu'un type de sortie, `Ranking` et `Maybe a`. Ce qui implique que `runKleisli` aura les types suivants :

```
1  Kleisli m                a          b          -- generique
2   runKleisli ::           a ->       m b
3  Kleisli (State Ranking) Ticket     Ranking    -- genRanking
4   runKleisli ::           Ticket ->  State Ranking Ranking
5  Kleisli (State Counter)  a          (Maybe a)  -- stabilize
6   runKleisli ::           a ->       Maybe a
```

La composition ne peut plus se faire selon les fonctions classiques comme `>>>` puisque les contextes, ici `Kleisli (State Ranking)` et `Kleisli (State Counter)`, sont différents.

```
1  stabilizedRanking :: Kleisli (State (Ranking, Counter))
       Ticket (Maybe Ranking)
2  stabilizedRanking = Kleisli stabilizedRanking'
3
4  stabilizedRanking' :: Ticket -> State (Ranking, Counter)
       (Maybe Ranking)
5  stabilizedRanking' e = do
6      (ir, is) <- get
7      let (rv, rs) = runState (runKleisli genRanking e) ir
8      let (sv, ss) = runState (runKleisli stabilize rv) is
9      put (rs, ss)
10     return sv
```

Notre `Monad` stocke un état qui est une paire comprennant les états des deux noeuds le composant, `Ranking` et `Counter`.

Nous récupérons et séparons les deux états ligne 6.

Nous ajoutons, ligne 7, le nouveau `Ticket e` en appliquant `e` au noeud `genRanking`, via `runKleisli genRanking e`. Nous obtenons un `State Monad` que nous évaluons

avec l'état `Ranking` de la `Monad` du niveau supérieur, via `runState (runKleisli genRanking e)ir`.

Nous faisons la même opération avec `stabilize` et ses fonctions respectives, ligne 9.

Nous changeons l'état de la `Monad` du niveau supérieur avec les nouveaux états, ligne 9.

Enfin nous renvoyons la valeur produite `sv`, ligne 10, par l'évaluation du noeud `stabilize`, ligne 8.

Cette composition est coûteuse car nous devons définir nous-même la sémantique d'évaluation qui n'est pas propre à ces deux noeuds en particulier.

Ce qui entraîne une forte duplication qui ne va faire que s'accroître de composition en composition. Le mécanisme ne passe pas à l'échelle puisque la paire sera de plus en plus grande en fonction des composition.

Ce mécanisme est adhoc est non pérein.

Modélisation avec du rythme Pour modéliser la notion de rythme, nous pouvons encapsuler les noeuds précédents dans de nouveaux types, mais comme nous l'avons, la composition n'en serait que plus difficile.

3.3 Introduction des *Heterogeneous-Arrow*s

L'idée de base est de prendre 2 types et d'en générer un 3ème, nous pouvons décrire le *typeclass* de la manière suivante :

```
1 class HeterogeneousArrow x y z where
2     (>*>) :: x a b -> y b c -> z a c
```

`HeterogeneousArrow` est un *typeclass* qui se base sur trois types : `x`, `y` et `z`. Il possède un opérateur, `>*>`, qui combine deux types dont les paramètres sont composables dans un troisième.

Cette définition est trop générale, en effet, cela nous laisse la possibilité de définir n'importe quel `z` en fonction de `x` et de `y`. Nous devons mettre en évidence que `z` dépend du couple `x` et `y`. Pour cela nous utilisons les dépendances fonctionnelles, ou *functional dependencies*.

```
1 class HeterogeneousArrow x y z | x y -> z where
2     (>*>) :: x a b -> y b c -> z a c
```

L'`instance` la plus triviale est lorsque `x` et `y` sont de même type et qu'il est composable, au sens qu'il implante l'`instance` de `Category` :

```
1 instance (Category x) => HeterogeneousArrow x x x where
2     f >*> g = g . f
```

Nous définissons ainsi que les `Category` ne sont qu'un cas particulier de `HeterogeneousArrow`. Dans ce cas, `>*>` équivault à la composition, `(.)`, de `Category`.

Nous définissons l'`instance Kleisli (State s)` de `HeterogeneousArrow` de la manière suivante :

```
1 instance HeterogeneousArrow (Kleisli (State s)) (Kleisli
      (State s')) (Kleisli (State (s, s'))) where
2 f >*> g = Kleisli (\e -> do
3                           (ir, is) <- get
4                           let (rv, rs) = runState (runKleisli
                                f e) ir
5                           let (sv, ss) = runState (runKleisli
                                g rv) is
6                           put (rs, ss)
7                           return sv)
```

Nous avons utilisé une méthode similaire pour `stabilizedRanking`. Il y a deux différences : d'une part les `Kleisli` sont paramétrées, d'autre part, nous nous assurons de manière générique que le type d'entrée de la seconde `Kleisli` est bien du type de sortie de la première.

`stabilizedRanking` devient donc :

```
1 stabilizedRanking :: Kleisli (State (Ranking, Counter))
      Ticket (Maybe Ranking)
2 stabilizedRanking = genRanking >*> stabilize
```

`stabilizedRanking` en est réduit à sa forme la plus simple car c'est `HeterogeneousArrow` qui a la charge de la difficulté de la composition.

Nous rajoutons à présent une notion de rythme. Elle prendra la forme de deux types encapsulés, via `newtype`, il s'agit simplement pour le compilateur de distinquer deux types.

Le type `Fast`, avec son constructeur `F`, est déstiné à encapsuler `genRanking`. `Slow` contient `stabilize`.

```
1 newtype Fast s i o = F (Kleisli (State s) i o)
2 newtype Slow s i o = S (Kleisli (State s) i o)
3
4 encStabilize :: Slow Counter a (Maybe a)
5 encStabilize = S stabilize
6
7 encGenRanking :: Fast Ranking Ticket Ranking
8 encGenRanking = F genRanking
```

`Fast` et `Slow` n'ayant pas d'`instance` de `HeterogeneousArrow`, il faut en définir une pour pouvoir les composer :

```
1 instance HeterogeneousArrow (Fast s) (Slow s') (Slow (s,
      s')) where
2     F f >*> S g = S (f >*> g)
```

Comme nous le voyons ci-dessus, `Fast` et `Slow` n'apportent aucune donnée, elles ne sont là que pour distinguer deux valeurs de même type qui n'ont pas la même signification au sein d'un programme.

Nous voyons ici que composer deux traitements, dont un lent donnera un traitement lent.

`stabilizedRanking` est donc définie de la manière suivante :

```
1 stabilizedRanking :: Slow (Ranking, Counter) Ticket (Maybe
      Ranking)
2 stabilizedRanking = encGenRanking >*> encStabilize
```

4 Conclusion

Nous arrivons donc à modéliser des traitements générants des flux de natures et de rythmes différents grâce aux *Heterogeneous-Arrow*.

Il fallait généraliser la composition pour obtenir une table de correspondance entre les différents éléments.

Même si notre méthode résout des problèmes de compositions, elle se base sur un ensemble de paires, ce qui peut nuire à la lisibilité et à la manipulation. Une des pistes d'amélioration est de considérer les états comme des collections hétérogènes, ou *heterogeneous collections*.

Références

1. André, C. : Representation and analysis of reactive behaviors : A synchronous approach (1996)
2. Ashcroft, E.A., Wadge, W.W. : Lucid - a formal system for writing and proving programs. SIAM J. Comput. 5(3), 336–354 (1976)
3. Benveniste, A., Bournai, P., Gautier, T., Le Guernic, P. : SIGNAL : a data flow oriented language for signal processing. Rapport de recherche RR-0378, INRIA (1985)
4. Berry, G., Cosserat, L. : The esterel synchronous programming language and its mathematical semantics. In : Brookes, S., Roscoe, A., Winskel, G. (eds.) Seminar on Concurrency, Lecture Notes in Computer Science, vol. 197, pp. 389–448. Springer Berlin Heidelberg (1984)
5. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A. : Lustre : A declarative language for real-time programming. In : Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 178–188. POPL '87, ACM, New York, NY, USA (1987)
6. Dennis, J., Fosseen, J., Linderman, J. : Data flow schemas. In : Ershov, A., Nepomniaschy, V.A. (eds.) International Symposium on Theoretical Programming, Lecture Notes in Computer Science, vol. 5, pp. 187–216. Springer Berlin Heidelberg (1972)
7. Elliott, C. : Functional implementations of continuous modeled animation. In : Proceedings of the 10th International Symposium on Principles of Declarative Programming. pp. 284–299. PLILP '98/ALP '98, Springer-Verlag, London, UK, UK (1998)
8. Elliott, C., Hudak, P. : Functional reactive animation. In : Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming. pp. 263–273. ICFP '97, ACM, New York, NY, USA (1997)

9. Elliott, C., Schechter, G., Abi-Ezzi, S. : MediaFlow, a framework for distributed integrated media. Tech. Rep. SMLI TR-95-40, Sun Microsystems Laboratories (1995)
10. Elliott, C., Schechter, G., Yeung, R., Abi-Ezzi, S. : Tbag : A high level framework for interactive, animated 3d graphics applications. In : Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques. pp. 421–434. SIGGRAPH '94, ACM, New York, NY, USA (1994)
11. Hughes, J. : Generalising monads to arrows. Sci. Comput. Program. 37(1-3), 67–111 (May 2000)
12. Kahn, G. : The semantics of simple language for parallel programming. In : IFIP Congress. pp. 471–475 (1974)
13. Launchbury, J., Peyton Jones, S. : State in haskell. LISP and Symbolic Computation 8(4), 293–341 (1995)
14. Marlow, S. : Haskell 2010 language report (2010)
15. Nilsson, H., Courtney, A., Peterson, J. : Functional reactive programming, continued. In : Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02). pp. 51–64. ACM Press, Pittsburgh, Pennsylvania, USA (Oct 2002)
16. Strachey, C. : The varieties of programming language. Tech. Rep. PRG-10, Oxford University (March 1973)
17. Thies, W., Karczmarek, M., Amarasinghe, S. : Streamit : A language for streaming applications. In : Horspool, R. (ed.) Compiler Construction, Lecture Notes in Computer Science, vol. 2304, pp. 179–196. Springer Berlin Heidelberg (2002)
18. Wadler, P. : Comprehending monads. In : Proceedings of the 1990 ACM Conference on LISP and Functional Programming. pp. 61–78. LFP '90, ACM, New York, NY, USA (1990)
19. Wan, Z. : Functional Reactive Programming for Real-Time Reactive Systems. Ph.D. thesis, Department of Computer Science, Yale University (December 2002)