TD6 : Les flux

1 Type Abstrait : les flux

Les flux sont une structure très importante et très pratique. Un flux est une séquence d'éléments, potentiellement infinie, contrairement à une liste. Les flux permettent de traiter naturellement les sujets suivants :

- la reconnaissance de langages (dans un prochain TD)
- le traitement de signaux discrets (filtres numériques)
- les ensembles infinis et itérateurs
- les séquences infinies (séries formelles, de Taylor)
- les approximations numériques et les limites
- la simulation des systèmes dynamiques

La programmation par flux, lorsque le problème posé s'y prête, possède les avantages suivants par rapport à l'usage de listes/tableaux :

- éléments calculés à la volée
- faible consommation mémoire, localité des calculs
- charge de calcul mieux répartie au cours du temps
- Décorrélation/indépendance entre la production de valeurs et leur consommation

Remarque : Les flux sont à la base des langages dits **synchrones**, tels Lustre, utilisés dans les commandes d'avions, de trains, de centrales nucléaires, etc.

1.1 spécification

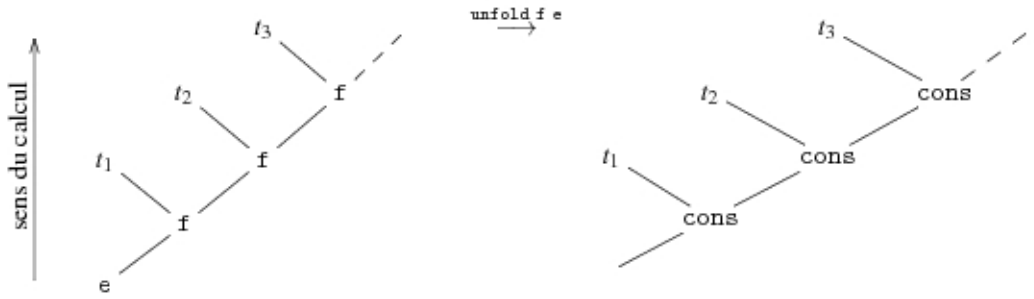
Un flux, étant potentiellement infini, ne peut être (en OCaml) une pure structure de données. De plus, un itérateur similaire à un "`fold`" est interdit sur les flux car il est supposé replier une structure de données depuis la fin jusqu'au début. Par contre, on peut toujours déplier, i.e. parcourir un flux depuis la première valeur jusqu'à... éventuellement l'infini !

Bien que le type A-liste et le type A-flux soient tous deux solution du schéma de type suivant :

$$X = 1 + A \times X$$

il faut considérer le type flux "à l'envers" du type liste. Pour les listes, on a un itérateur `fold_right` qui consomme une liste en appliquant une fonction (en orientant l'équation de droite à gauche), tandis que dans le T.A. flux on aura un co-itérateur "dual" `unfold` qui produit un flux en appliquant une fonction (en orientant l'équation de gauche à droite).

La fonction `unfold` transforme donc une structure de contrôle (le graphe itéré de la fonction) en une structure de données (le flux), son effet peut être spécifié graphiquement par cette transformation structurelle d'arbres :



Notons que les fonctions qui exigent d'examiner l'ensemble des valeurs d'un flux afin de déterminer leur résultat sont potentiellement non-terminantes et doivent être évitées. C'est le cas de l'égalité entre deux flux. De plus, définir des flux infinis directement par des fonctions récursives sera impossible (dans un langage strict comme OCaml), puisqu'il faudrait une infinité d'appels récursifs. Tous les flux seront définis par transformations de flux, ou bien encore grâce à l'itérateur `unfold`.

On aurait donc par exemple le type abstrait suivant, dans l'interface `Iter`, le module `Flux` étant une des réalisations possibles de cette interface.

```
module type Iter =
sig
  type 'a t

  val vide : 'a t
  val cons : 'a -> 'a t -> 'a t
  val uncons : 'a t -> ('a * 'a t) option
  val apply : ('a -> 'b) t -> ('a t -> 'b t)
  val unfold : ('b -> ('a * 'b) option) -> ('b -> 'a t)
  val filter : ('a -> bool) -> 'a t -> 'a t
  val append : 'a t -> 'a t -> 'a t
end
```

Le type de `unfold` est "dual" du type de `List.fold_right : ('a -> 'b -'> 'b) -> 'a list -> 'b -> 'b` $\approx$ `(('a * 'b) option -> 'b) -> ('a list -> 'b)`. Le code suivant illustre cette dualité :

```
(* fold : (('a * 'b) option -> 'b) -> ('a list -> 'b) *)
let rec fold f liste =
 match liste with
 | []   -> f  None
 | t::q -> f (Some (t, fold f q));;

(* unfold : ('b -> ('a * 'b) option) -> ('b -> 'a Flux.t) *)
let rec unfold f e =
 match f e with
 | None        -> Flux.vide
 | Some (t, e') -> Flux.cons t (unfold f e');;


(* le flux qui vaut constamment 0 *)
let flux_nul = Flux.unfold (fun c -> Some (c, c)) 0;;
(* ou: *)
let flux_nul = Flux.unfold (fun () -> Some (0, () )) ();;

(* le flux qui contient tous les entiers relatifs pairs, par ordre croissant en valeur absolue *)
let even_integers =
 Flux.unfold (fun i -> Some (2*i, if i <= 0 then 1-i else -i)) 0;;
```

▷ **Exercice 1** *Écrire, par analogie avec le type `'a list`, les fonctions usuelles suivantes :*

- *`constant : 'a -> 'a Flux.t`.*
- *`map : ('a -> 'b) -> 'a Flux.t-> 'b Flux.t`.*
- *`map2 : ('a -> 'b -> 'c) -> 'a Flux.t-> 'b Flux.t-> 'c Flux.t`.*

1.2 Une implantation possible

Un flux n'est pas une structure de données pure/stricte, il faut que la queue d'un flux ne soit calculée que si l'on en a besoin, de manière **paresseuse**, i.e. lorsqu'on réalise un `uncons`. Dans un langage fonctionnel, cela s'implante simplement à travers l'application (ou non) d'une fonction. Cette solution est particulièrement inefficace car on ne mémorise jamais le fait qu'une valeur d'un flux ait déjà été calculée ou non. Ceci entraîne qu'à chaque accès à une valeur d'un flux, on exécute à nouveau la fonction qui définit cette valeur, effectuant ainsi autant de fois tous les calculs...

Le module `Flux` peut ainsi être implanté comme suit :

```
type 'a t = Tick of (unit -> ('a * 'a t) option);;

let vide = Tick (fun () -> None);;

let cons t q = Tick (fun () -> Some (t, q));;

let uncons (Tick fflux) = fflux ();;

let rec unfold f e =
 Tick (fun () ->
 match f e with
 | None        -> None
 | Some (t, e') -> Some (t, unfold f e'));;

let rec apply f x =
 Tick (fun () ->
 match uncons f, uncons x with
 | None        , _            -> None
 | _           , None         -> None
 | Some (tf, qf), Some (tx, qx) -> Some (tf tx, apply qf qx));;

let rec filter p flux =
 Tick (fun () ->
 match uncons flux with
 | None       -> None
 | Some (t, q) -> if p t then Some (t, filter p q)
                         else uncons  (filter p q));;

let rec append flux1 flux2 =
 Tick (fun () ->
 match uncons flux1 with
 | None          -> uncons flux2
 | Some (t1, q1) -> Some (t1, append q1 flux2));;
```

1.3 Une meilleure implantation paresseuse

En OCaml, on peut bénéficier du mode d'évaluation paresseux, à l'aide de la libraire `Lazy`. Les flux nativement paresseux sont disponibles également dans d'autres langages stricts, comme JAVA. Il existe également des langages, comme Haskell, où le mode d'évaluation par défaut est paresseux et dans lesquels les flux s'encodent encore plus simplement sans la surcouche dûe à la librairie `Lazy`.

La librairie `Lazy` propose l'implantation d'un mécanisme d'évaluation qu'on pourrait identifier à des fermetures fonctionnelles (comme dans la précédente solution), couplées avec la mémorisation des résultats calculés. Les valeurs du type `Lazy.t` sont appelées des **glaçons**, qui contiennent une fermeture fonction-

nelle, i.e. un calcul “gelé” non évalué, ou bien, s’ils sont fondus, représentent une valeur, i.e. le résultat du dit calcul. On construit des glaçons avec la construction **lazy** (*expr*) (où *expr* n’est pas évaluée) et on les fait fondre avec `Lazy.force` ou encore à l’aide d’un filtrage de la forme **lazy** *pattern*.

```
type 'a t = Tick of ('a * 'a t) option Lazy.t;;

let vide = Tick (lazy None);;

let cons t q = Tick (lazy (Some (t, q)));;

let uncons (Tick flux) = Lazy.force flux;;
(* ou bien par filtrage *)
let uncons (Tick (lazy flux)) = flux;;

let rec apply f x =
  Tick (lazy (
  match uncons f, uncons x with
  | None         , _            -> None
  | _            , None         -> None
  | Some (tf, qf), Some (tx, qx) -> Some (tf tx, apply qf qx)));;

let rec unfold f e =
  Tick (lazy (
  match f e with
  | None        -> None
  | Some (t, e') -> Some (t, unfold f e')));;

let rec filter p flux =
  Tick (lazy (
   match uncons flux with
   | None        -> None
   | Some (t, q) -> if p t then Some (t, filter p q)
                    else uncons (filter p q)));;

let rec append flux1 flux2 =
 Tick (lazy (
 match uncons flux1 with
 | None          -> uncons flux2
 | Some (t1, q1) -> Some (t1, append q1 flux2)));;
```

2 Applications

2.1 Énumérations

On s’intéresse à représenter les ensembles énumérables par des flux (dont les valeurs successives énumèrent précisément les éléments).

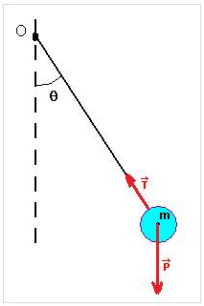
▷ **Exercice 2** *Suite de Fibonacci. On cherche à construire le flux des valeurs correspondant à la suite de Fibonacci* $f_0 = 0, f_1 = 1, f_{n+2} = f_{n+1} + f_n$. *On peut réutiliser le principe de la* ***marelle****.*

2.2 Systèmes dynamiques

La simulation de systèmes dynamiques modélisés par des flux fonctionnels (récursifs) constitue le domaine appelé *Functional Reactive Programming*, qui possède ses propres langages (fonctionnels réactifs) de programmation, comme Elm.

On va s'intéresser à modéliser le pendule pesant simple (masse ponctuelle). Rappelons que le pendule, modélisé ci-dessous, obéit à l'équation différentielle suivante :

$$\theta'' + \tfrac{g}{l}\sin\theta = 0, \text{ c'est-à-dire } \begin{cases} \frac{d\theta}{dt} = \theta' \\ \frac{d\theta'}{dt} = -\frac{g}{l}\sin\theta \end{cases}, \text{ ou encore } \begin{cases} \theta = \theta_0 + \int_0^t \theta' \, dt \\ \theta' = \theta'_0 + \int_0^t -\frac{g}{l}\sin\theta \, dt \end{cases}$$



Traditionnellement, simuler ce système revient à définir, de manière approchée, par un schéma d'Euler explicite par exemple, les valeurs successives de l'angle θ et de sa dérivée θ' au cours du temps, l'équation étant discrétisée par pas de h unités de temps. Avec les flux, on peut directement exprimer θ comme solution (approchée) de son équation différentielle.

▷ **Exercice 3 (Simulation numérique du pendule)**

- *Définir* `integre : float -> float t-> float t` *le flux récursif qui accumule/intègre, par pas de temps* `dt` *fixe, les valeurs d'un flux donné, avec une valeur initiale nulle.*
- *Définir récursivement les flux* `theta` *et* `dtheta`*, comme solutions approchées des équations aux intégrales précédentes.*