TD6: Les flux

Thèmes et objectifs

45 mins:

- les flux
- la paresse

45 mins:

- énumérations
- simulation de systèmes dynamiques

▷ **Support étudiant**

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

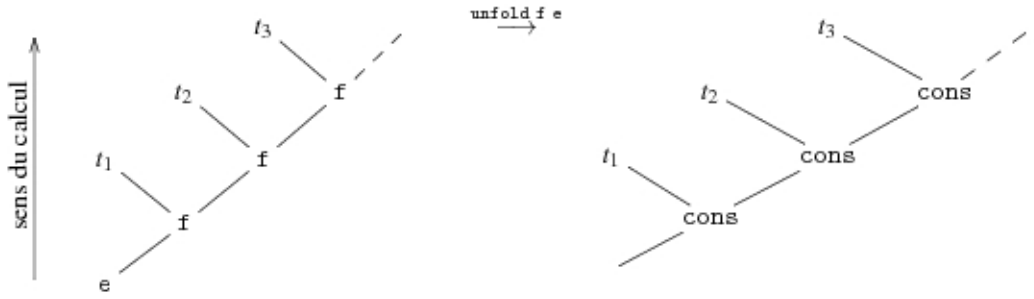
Un flux, étant potentiellement infini, ne peut être (en OCaml) une pure structure de données. De plus, un itérateur similaire à un "fold" est interdit sur les flux car il est supposé replier une structure de données depuis la fin jusqu'au début. Par contre, on peut toujours déplier, i.e. parcourir un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

Remarque : on choisit de ne pas parler de l'implantation des flux, afin de se familiariser avec l'usage de l'interface seule du T.A. L'implantation est abordée plus tard. On note la correspondance entre uncons et le filtrage de listes, où [] correspond ici à None et t :: q à Some(t, q). Il s'agit donc d'un principe d'élimination de flux, qu'on aurait pu aussi définir comme: uncons : 'a t -> 'b -> ('a -> 'a t-> 'b) -> 'b.

▷ **Support étudiant**

Exemples de flux:

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

▷ ***Solution***

La fonction map2 *est plus difficile à écrire a priori. Le principe, à faire trouver, est que chaque élément* e1 *du premier flux soit transformé en une fonction* (**fun** e2 -> f e1 e2) *qui attend l'élément correspondant* e2 *du second flux, l'application de cette fonction se faisant grâce à* apply.

```
let constant c =
 Flux.unfold (fun c -> Some (c, c)) c;;

let map f x =
 Flux.apply (constant f) x;;

let map2 f x y =
 Flux.apply (map f x) y;;
```

On a une structure de foncteur applicatif. On a aussi une structure de monade, mais pas très naturelle/utile. Ces notions seront abordées plus tard de toutes façons.

▷ **Support étudiant**

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 fonctionnelle, 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)));;
```

Remarque : l'implantation choisie répond à la spécification générale des flux, avec toutes ses fonctions à définir. Dans des cas où seulement une faible partie de ces fonctions est nécessaire, on peut envisager d'autres implantations plus légères, i.e. qui ne construisent pas de structures explicites contenant toutes ces fermetures.

▷ **Support étudiant**

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****.*

▷ ***Solution***

L'utilisation d'une marelle est simplifiée ici car il n'y a pas de cas particuliers de terminaison. On se contente de transporter les deux dernières valeurs de la suite dans une paire `(f_n,f_Sn)`*. Voici le schéma co-récursif utilisé, où toutes les valeurs utilisées (pour l'addition) sont définies* ***avant*** *dans le flux:*

$$\begin{array}{cccccccc} & & & f_0 & f_1 & f_2 & \cdots \\ + & & & f_1 & f_2 & f_3 & \cdots \\ \hline = & f_0 & f_1 & f_2 & f_3 & f_4 & \cdots \end{array}$$

On notera que la définition récursive directe du flux fibonacci*, de la forme* **let rec** xx = f xx *n'est pas admise en* OCaml*, pour des raisons de constructivité. Pour être certain que cette définition ait bien un sens, on doit la "protéger" en faisant explicitement apparaître un constructeur, ici* Tick*, et non à travers la fonction* cons *par exemple. Cette limitation peut être contournée (voir plus loin), du moment que la définition récursive est "sémantiquement" constructive, à défaut de l'être syntaxiquement. Les langages paresseux natifs, comme Haskell, n'ont pas ce problème.*

D'une part, cette solution avec flux récursif dévoile l'implantation (bof); d'autre part, elle est aussi inefficace que la fonction récursive naïve, pour les mêmes raisons. On aurait besoin de constructions/langages réellement paresseux.

```
(* version avec marelle *)
let fibonacci =
 Flux.unfold (fun (f_n, f_Sn) -> Some (f_n, (f_Sn, f_n+f_Sn))) (0, 1);;

(* version directe et co-recursive *)
let tail f =
 match Flux.uncons f with
 | Some (_, q) -> q
 | None        -> failwith "flux vide !";;

let rec fibonacci =
 (* la valeur recursive doit etre constructive *)
  Tick (lazy (Some (0, Tick (lazy (Some (1, Flux.map2 (+) fibonacci (tail fibonacci ))))))));;
```

▷ **Exercice 3** *Crible d'Erasthostène. Principe:*

1. *on dresse la suite des entiers à partir de 2.*
2. *le premier élément de cette suite est un nombre premier.*
3. *on retire ses multiples aux nombres restants.*
4. *on recommence en 2.*

- *Définir un flux* nats_from_2 *contenant tous les entiers à partir de 2.*
- *Définir une fonction qui retire les multiples d'un entier donné à un flux.*
- *Définir une fonction* next_prime *qui à partir d'un flux non vide, renvoie son premier élément ainsi que le flux débarrassé de ses multiples.*
- *En déduire une définition non récursive de* sieve*, le flux des nombres premiers.*

▷ ***Solution***

```
(* nats_from_2 : int Flux.t                                    *)
let nats_from_2 = Flux.unfold (fun p -> Some (p, p+1)) 2;;

(* next_prime : int Flux.t -> (int * int Flux.t) option        *)
(*   parametre : un flux d'entiers non nuls                    *)
(*   resultat  : si le flux est vide, un flux vide             *)
(*               sinon, (p, f) avec p le premier element du flux *)
(*                      et f le flux sans les multiples de p   *)
let next_prime flux =
 match Flux.uncons flux with
  | None        -> None
  | Some (t, q) -> Some (t, Flux.filter (fun i -> i mod t <> 0) q);;

(* sieve : int flux.t                                          *)
let sieve =
 Flux.unfold next_prime nats_from_2;;
```

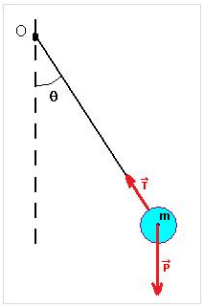
▷ **Support étudiant**

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 4 (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.*

▷ *Solution*

La fonction integre *doit définir un flux récursif* acc*, définit par:* $acc_0 = 0; acc_{i+1} = acc_i + dt * flux_i$*, ce qu'on traduit directement. On obtient bien un flux constructif, avec un constructeur* Tick *apparent. Puis, on reproduit les équations aux intégrales directement. Malheureusement, on définit récursivement* theta *à travers* integre *et non directement. Comme la définition est quand même contructive, on peut contourner sans risque le problème du constructeur non apparent en encapsulant la définition de* theta *dans un terme de la forme:*

```
let rec theta = Flux.Tick (lazy (Flux.uncons expr))
```

Ce terme est évidemment neutre, i.e. représente la fonction identité. On a choisi, par souci de simplicité, de protéger ainsi toutes le définitions récursives, mais en jouant avec les définitions locales, seule une d'entre elles nécessite réellement ce traitement, typiquement theta*.*

```
let g = 9.81;;
let l = 30.;;
let dt = 0.1;;
let theta0 = 1.;;
let dtheta0 = 0.1;;

let integre dt flux =
  let rec acc =
    Tick (lazy (Some (0., Flux.map2 (fun a f -> a +. dt *. f) acc flux)))
  in acc;;

let rec theta =
   Tick (lazy (Flux.uncons (
     Flux.map ((+.) theta0) (integre dt dtheta)
   )))
and dtheta =
   Tick (lazy (Flux.uncons (
     Flux.map ((+.) dtheta0) (integre dt ddtheta)
   )))
and ddtheta =
   Tick (lazy (Flux.uncons (
     Flux.map (fun theta_i -> -. g /. l *. sin theta_i) theta
   )));;
```

Il existe également pour integre *la solution utilisant* unfold*. Même si le résultat est peut-être moins élégant, on pourra toujours y arriver, les arguments du* unfold *constituant la sémantique "naturelle" des flux, c'est-à-dire qu'il existe une implantation des flux par des paires (valeur initiale, générateur).*

```
let integre dt flux =
  let iter (acc, flux) =
    match Flux.uncons flux with
    | None        -> None
    | Some (t, q) -> Some (acc, (acc +. dt *. t, q)) in
    Flux.unfold iter (0., flux);;
```

À noter que cette implantation n'est pas équivalente à la version récursive. En effet, cette version de la fonction integre *gère différemment la fin de flux en terminant un instant avant (mais dans les systèmes dynamiques, les flux sont infinis) et n'est plus* **strictement** *productive, car sa première valeur nécessite maintenant l'évaluation de son argument. On peut corriger cela en "décalant" le* unfold*, on obtient le même flux, mais strictement productif, plus apte à être utilisé pour les systèmes dynamiques:*

```
let integre dt flux =
  let iter (acc, flux) =
    match Flux.uncons flux with
    | None          -> None
    | Some (t, q) -> let acc' = acc +. dt *. t
                     in Some (acc', (acc', q)) in
    Flux.cons 0. (Flux.unfold iter  (0., flux ));;
```

Remarque: Pour obtenir une simulation "temps-réel", il faudrait n'afficher les valeurs successives des flux que tous les dt unité de temps. Cela nécessite une synchronisation système que l'on ne fera pas, même en TP. La précision numérique de ce schéma naïf n'est pas bonne, car le pas de calcul dt est fixé, indépendamment de la fonction à intégrer. Par exemple, si dt est trop grand, on constate que le pendule va gagner de l'énergie et son balancement va devenir rotation, de plus en plus rapide. Au lieu de raisonner sur dt, on doit plutôt raisonner sur l'écart maximum entre deux valeurs successives de la fonction et ajuster dt à cet effet, à la façon de l'intégrale de Lebesgue. On obtient ainsi des schémas mieux maîtrisés. En contrepartie, dt est alors une sortie (un flux) de la simulation.