Streams et Threads

INF 441 : Programmation Avancée

Xavier Rival

15 mai 2017

Quelques éléments administratifs

Tutorat :

mardi 30 mai, 17h45-19h15, salle 30, à confirmer

sous réserve de disponibilité des enseigants

- une séance a eu lieu le 16 mai...
- syntaxe OCaml, utilisation des outils
- apporter votre machine si nécessaire
  (installation OCaml sur votre environnement personnel)
- merci de me dire si vous êtes intéressés

Café-DIX : réunion d'information sur la 3A et les PA d'info

prévue mercredi 24 mai, de 13h00 à 14h00

Projets : **deadline le 31 mai** (ça approche...)

Rappels des cours précédents

Types et **structures** :

- comment **organiser** les données
- comment obtenir une **garantie de sûreté**
  i.e., que certaines erreurs sont impossibles

Fonctions :

- technique de **paramétrisation** et d'**abstraction**
- élément **essentiel pour l'expressivité** du langage
  fonctions de première classe, ordre supérieur, clôtures. . .

Modules et **foncteurs** :

- vers du **code modulaire** et **paramétrable**

Objets et **classes** :

- une construction des programmes qui relie **structures** et **algorithmes**

Rappels des cours précédents : réducteurs

Technique à base de **réducteurs** :

- place le code itératif **côté fournisseur**
- le **client** doit passer l'opération à effectuer
  e.g., via une fonction d'ordre supérieur
  et l'itération échappe ensuite à son contrôle

Code "Fournisseur"	Code "Client"
structure de données fonctions spécifiques add, mem, remove...	fonctions génériques aucune hypothèse sur le code fournisseur

Signature :

```
type t
val add: t -> elt -> t
val mem: t -> elt -> bool
val iter: (elt -> unit) -> t -> unit
val fold: (elt -> 'a -> 'a) -> t -> 'a -> 'a
```

Rappels des cours précédents : itérateurs

Technique à base d'**itérateurs** :

- place le code itératif **côté client**
- le **fournisseur** définit un type et des primitives pour initialiser l'itération, et pour passer à l'élément suivant

Code "Fournisseur"	**Code "Client"**
structure de données fonctions spécifiques add, mem, remove...	fonctions génériques aucune hypothèse sur le code fournisseur

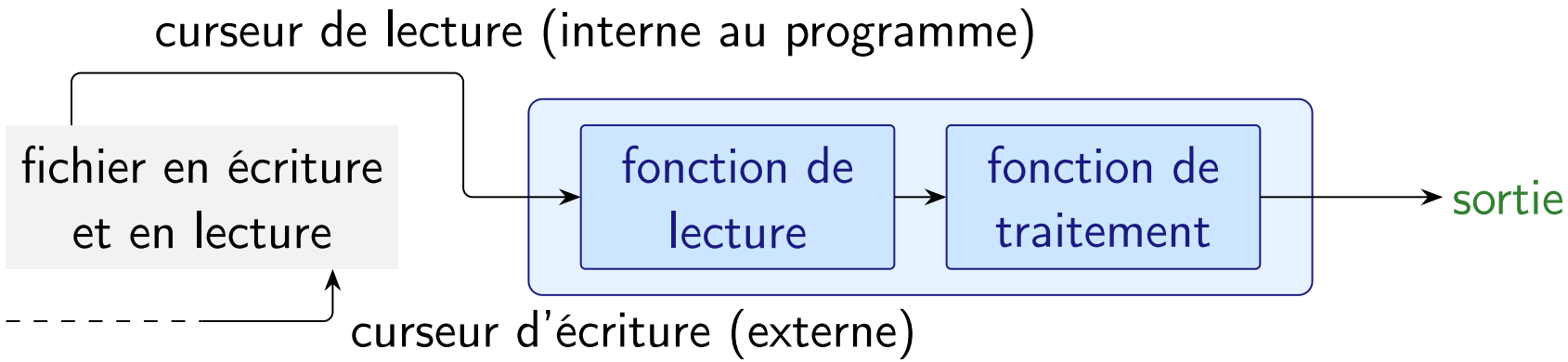
Signature :

```
type t
val add: t -> elt -> t
val mem: t -> elt -> bool
type it
val iter_new: t -> it
val iter_next: it -> elt option
```

Itération non bornée / infinie

Traitement du contenu d'un fichier qui est **en cours d'écriture** :

1. **lecture** d'une entrée dans le fichier (si disponible)
2. **traitement** de l'entrée lue
3. puis **attente** d'une nouvelle entrée...



On retrouve le **même schéma** dans de **nombreux domaines** :

- **systèmes d'exploitation** (Linux, Unix, MacOS, etc)
- **interfaces graphiques** (KDE, XFCE, mais aussi emacs ou eclipse)
- **streaming** audio ou vidéo, **téléphonie**
- **serveurs web**, etc

Technique basique d'itération infinie

Une **technique classique** : la **boucle d'événements** (**event loop**)

```
let e_loop =
  while true do
    let ev =
      wait_nxt_event () in
    handle_event ev
  done
```

- **encodage direct** de ce motif attente d'événements et traitement
- on peut ajouter une condition de sortie (EOF, System Off...)

Limitations :

- cette construction n'est **ni jolie ni compositionnelle** : comment pourrions nous chaîner plusieurs telles tâches ?
- la boucle contient des **attentes bloquantes** i.e., pendant lesquelles le programme ne fait rien... comment **réduire les attentes** ?

Programme de ce cours

Nous allons étudier des **techniques de programmation** permettant de traiter ces problèmes :

1. **Flots / streams** :
   séquence finie ou infinie, calculée **incrémentalement**

2. **Threads légers / fibres** :
   calculs **désynchronisés**, que l'on peut voir comme parallèles

Dans les deux cas, on considère des **librairies**
(pas des constructions natives du langage)

Outline

Retour sur le type d'un itérateur immutable

Définition satisfaisant l'interface :

```
type t  (* type abstrait des elements *)
type it (* type abstrait des iterateurs *)
(* construction d'un iterateur *)
val iter_new: t -> it
(* extraction de l'element suivant *)
val iter_next: it -> (it * elt) option
```

Utilisation ("consommation"), côté code client :

```
let it = iter_new t in
let rec aux i0 =
  match iter_next i0 with
  | None -> ( ) (* fini ! *)
  | Some (v, i1) -> doit v; aux i1 in aux it
```

Que nous manque-t'il ?

- l'élément `t` **doit être connu lors de la création** de `it`
- on voudrait ne pas avoir `t`, et lire / calculer `it` **à la demande**

Évaluation retardée

Revenons sur la notion d'**évaluation retardée** :

- `64` définit une **valeur immédiate**, connue sans plus de calcul
- `fun () -> 64` définit une **fonction constante**, mais pas seulement :
  on peut aussi voir cette expression comme une **valeur**,
  qui ne sera délivrée qu'après application à `()`

C'est une astuce bien connue en programmation fonctionnelle...

Application :

- `Printf.printf "fubar"` affiche immédiatement quelque chose...
  c'est un **effet de bord** qu'on ne peut pas capturer ou propager ailleurs
- par contre, `fun () -> Printf.printf "fubar"` décrit le même effet de bord, mais **ne l'effectue pas immédiatement**
  on peut le dupliquer, l'effectuer plus tard, ou jamais...

Nous allons mettre ce principe en pratique.

Flot : une première définition de type

On souhaite définir un **flot** d'éléments, qui décrit :

- soit le **flot vide** :
  aucun élément, et on sait qu'il n'y en aura plus
- soit **un élément, mais qui n'arrivera peut être pas tout de suite**
  et **le flot des éléments suivants**

On obtient le **type suivant** :

```
type 'a flot_now =
  | Vide
  | NonVide of 'a * 'a flot
and 'a flot = unit -> 'a flot_now
```

Ici, `flot_now` décrit un flot dont le premier élément est prêt, et `flot` un flot dont le premier élément n'est pas nécessairement prêt (la définition du poly ne fait pas cette distinction).

Production d'un flot

Premier exemple :

un flot correspondant aux entiers compris dans intervalle donné

```
let rec flot_intv a b =
  fun ( ) ->
    if a > b then Vide
    else NonVide (a, flot_intv (a + 1) b)
```

Lorsque l'on appelle `flot_itv n0 n1` on obtient une fonction avec un argument de type `unit` :

- `flot_itv 10 0 ()` renvoie `Vide`
  dans ce cas, le flot produit exactement 0 valeur
- `flot_itv 5 7 ()` renvoie `(5, f)` où `f` est une fonction décrivant l'intervalle à partir de 6...
  dans ce cas, le flot engendre trois valeurs puis retourne `Vide`

Production d'un flot

On peut aussi produire un **flot infini**.

Flot infini constant :

```
let rec flot_cst a =
  fun ( ) -> NonVide (a, flot_cst a)
```

- noter le type polymorphe : `'a -> 'a flot`
- par exemple, `flot_cst true ()` renvoie `true` et le même flot...

Flot infini décrivant les entiers à partir de 0 :

```
let rec flot_int n =
  fun ( ) -> NonVide (n, flot_int (n + 1))
```

- à partir de 0, on obtient 0, puis 1, puis 2...
- du fait de l'arithmétique modulaire, on finit par obtenir une valeur négative...

Et sans évaluation retardée

À chaque pas de lecture du flot, nous avons **inséré une évaluation retardée**. Qu'observerions nous **sans cela** ?

```
# let flot_int_0 =
  let rec flot n = n :: flot (n + 1) in
  flot 0 ;;
Stack overflow during evaluation (looping recursion?).
```

Autrement dit, l'évaluation retardée a ici deux effets :

1. elle soumet la production de l'élément suivant **à une action explicite**, à savoir un appel de fonction
2. et du même coup, elle **évite de construire une récursion immédiate infinie**, même si l'objet calculé est bien infini...

Consommation d'un flot

La **consommation d'un flot** est **très similaire** à **l'utilisation d'un itérateur** comme nous l'avons vue au cours précédent :

- pour accéder à l'élément suivant, on se contente d'appeler la fonction "next", qui revient ici à **forcer l'évaluation** en appliquant à `()`
- si on lit la valeur `Vide`, le flot est vide (cas d'un flot fini)
- sinon, on peut **continuer** à accéder à des éléments

Exemple :

recherche du premier élément d'un flot `f`
qui **satisfait un prédicat** défini par `p: 'a -> bool`

```
let rec first p f0 =
  match f0 ( ) with
  | Vide -> None (* pas de solution *)
  | NonVide (a, f1) ->
      if p a then Some a (* solution trouvee *)
      else first p f1 (* continuer a chercher *)
```

Transformation d'un flot

On a observé qu'un flot définit un **calcul infini sans exiger immédiatement d'effectuer ce calcul**.

On peut exploiter cela pour **calculer incrémentalement un flot**.

Application : fonction `map`, sur les flots

```
let rec map (op: 'a -> 'b) (f0: 'a flot): 'b flot =
  fun ( ) ->
    match f0 ( ) with
    | Vide -> Vide
    | NonVide (a, f1) -> NonVide (op a, map op f1)
```

Considérons **l'évaluation du flot** `map op f0` :

- si on **force l'accès au premier élément de ce flot**, on déclenche le **calcul du premier élément** de `f0`
- autrement dit, les deux flots sont calculés **au même rythme**

Composition

On peut illustrer cette **fonction de transformation** en **composant** plusieurs opérations sur un flot...

Exemple : résolution de l'équation `f x = 0, x >= 0`
(où `f: int -> int`)

1. création du **flot des entiers** à partir de 0
2. **transformation** vers un flot contenant les images par `f`
3. **extraction de la première solution**

Solution directe :

```
let Some (a, _) =
  first (fun (_, b) -> b = 0) (* etape 3 *)
    (map (fun x -> (x, f x))  (* etape 2 *)
      (flot_int 0))           (* etape 1 *)
```

Composition

On peut illustrer cette **fonction de transformation** en **composant** plusieurs opérations sur un flot...

Exemple : résolution de l'équation `f x = 0, x >= 0`
(où `f: int -> int`)

1. création du **flot des entiers** à partir de 0
2. **transformation** vers un flot contenant les images par `f`
3. **extraction de la première solution**

Pour une telle composition, on peut utiliser **l'opérateur pipe** `|>` :

- `a |> f` représente `f a`
- `a |> f |> g` représente `g (f a)` (associativité à gauche)
- ici, on veut `a` = flot initial, `f` = `map (...)` et `g` = `first (...)`

Composition

On peut illustrer cette **fonction de transformation** en **composant** plusieurs opérations sur un flot...

Exemple : résolution de l'équation `f x = 0, x >= 0`
(où `f: int -> int`)

1. création du **flot des entiers** à partir de 0
2. **transformation** vers un flot contenant les images par `f`
3. **extraction de la première solution**

Solution avec pipe :

```
(flot_int 0)                                (* etape 1 *)
  |> (map (fun x -> (x, f x)))              (* etape 2 *)
  |> (first (fun (_, b) -> b = 0))          (* etape 3 *)
```

(l'ordre peut paraître plus intuitif que pour la solution directe)

Flot et itérateur persistant

Nous avons déjà mentionné les **liens** entre **flots** et **itérateurs**.

Plus concrètement, nous pouvons construire un **itérateur mutable** à partir d'un flot :

```
let flot_to_iter (f: 'a flot): unit -> 'a option =
  let cur = ref f in
  fun () ->
    match !cur () with
    | Vide -> None
    | NonVide (a, f) -> cur := f; Some a
```

- la référence a pour but de **stocker l'état interne**
- **élément suivant** :
  chaque application à `()` correspond à un appel à `iter_next` dans le cadre du cours précédent
- autrement dit, cette fonction correspond à `iter_new`...

Flot à partir d'un type arborescent

On peut aussi très souvent **construire un flot en s'inspirant de la définition d'un itérateur**...

Exemple : **flot produisant les éléments d'un arbre**

Type :

```
type 'a tree =
  | Leaf
  | Node of 'a tree * 'a * 'a tree
```

Flot à partir d'un type arborescent

On peut aussi très souvent **construire un flot en s'inspirant de la définition d'un itérateur**...

Exemple : **flot produisant les éléments d'un arbre**

Itérateur (mutable) :

```
type 'a iter = { mutable cur: 'a tree list }
let iter_new t = { cur = [ t ] }
let rec iter_next it =
  match it.cur with
  | [ ] -> None
  | Leaf :: a ->
      it.cur <- a;
      iter_next it
  | Node (x, a, y) :: l ->
      it.cur <- x :: y :: l;
      Some a
```

Flot à partir d'un type arborescent

On peut aussi très souvent **construire un flot en s'inspirant de la définition d'un itérateur**...

Exemple : **flot produisant les éléments d'un arbre**

Flot correspondant :

```
let rec mk_flot t acc =
  fun () -> mk_flot_now t acc
and mk_flot_now t acc =
  match t with
  | Leaf -> acc
  | Node (u, x, v) -> mk_flot_now u (NonVide (x,
      mk_flot v acc))
let tree_to_flot t = mk_flot t Vide
```

- l'accumulateur **se remplit** lorsqu'il faut chercher de nouveaux noeuds, en explorant une **branche gauche maximale**

Une limitation

Nous avons vu que nous pouvions **composer des opérations en chaîne** à partir d'un flot donné :

```
(flot_int 0)                                (* etape 1 *)
  |> (map (fun x -> (x, f x)))              (* etape 2 *)
  |> (first (fun (_, b) -> b = 0))          (* etape 3 *)
```

Comment pourrions nous faire, si nous avions besoin de faire **plusieurs opérations en parallèle, sur un même flot** ?

e.g.,

- **générer la suite des entiers**
- **d'un côté**, chercher les entiers **qui sont solution d'une équation**
- **de l'autre**, sélectionner les **entiers premiers**

Nous allons voir plus loin comment corriger cela...

La notion de calcul retardé

Depuis le début du cours, nous avons utilisé la notion de **calcul retardé** à l'aide de la construction `fun () -> calcul`

Formalisons cette construction à l'aide d'une **interface abstraite** :

```
type 'a delayed
val delay: (unit -> 'a) -> 'a delayed
val force: 'a delayed -> 'a
```

On a **l'implémentation triviale** :

```
type 'a delayed = unit -> 'a
let delay f = f
let force f = f ()
```

Inconvénient : le code suivant **évalue deux fois** la fonction

```
let d = delay (fun () -> (* ... long calcul ... *))
let x0 = force d
let x1 = force d
```

La notion de mémoisation

Comment éviter le recalcul ?

Nous avons déjà vu une solution (en TD) : la **mémoisation**

Définition : mémoisation

Technique consistant à **stocker des résultats de calculs** (en général à l'aide de structures mutables) de manière à **éviter de les effectuer à nouveau**.

Exemple : **mémoisation d'une fonction** `f: 'a -> 'b` :

- nécessite une **structure mutable permettant de stocker** des ensembles de paires `'a * 'b` (table de hachage...)
- lors d'un calcul de `f v`, **rechercher** `v` dans la table
- si `v` est présent, **renvoyer l'image correspondante**
- sinon, **calculer** l'image, et **la stocker dans la table**

Ici, il nous faut mémoiser une fonction `unit -> 'a` : **une référence suffit**

Module Lazy

Déclarations des **types** (en fait abstraits dans `Lazy`) :

```
type 'a lazy_state =
  | Delayed of unit -> 'a
  | Value of 'a
  | Exn of exn
type 'a lazy = 'a lazy_state ref
```

On peut construire directement un élément `'a lazy` à partir d'une fonction `unit -> 'a` (c'est le constructeur `Delayed`).

Évaluation :

```
let force (l: 'a lazy): 'a =
  match !l with
  | Value x -> x
  | Exn e -> raise e
  | Delayed f ->
      try let v = f () in l := Value v; v
      with e -> l := Exn e; raise e
```

Retour sur la définition des flots

Nous venons de voir que `lazy` **implémente deux fonctionnalités utiles pour décrire des flots** :

- **calculs retardés** :
  nous les utilisons depuis le début, à l'aide de `fun () -> ...`
- **mémoisation** :
  utiles pour soumettre la sortie d'un flot **à deux clients**

D'où une **nouvelle définition** :

```
type 'a stream_now =
  | Vide
  | NonVide of 'a * 'a stream
and 'a stream = 'a stream_now Lazy.t
```

Le **retard des calculs** et leur **mémoisation** se font directement à l'intérieur du module `Lazy`.

Production d'un flot

Nous rappelons notre **définition précédente** :

```
let rec flot_intv a b =
  fun ( ) ->
    if a > b then Vide
    else NonVide (a, flot_intv (a + 1) b)
```

À l'aide de **types paresseux**, nous obtenons la définition :

```
let rec stream_intv a b =
  lazy
    (if a > b then Vide
     else NonVide (a, stream_intv (a + 1) b))
```

- **peu de changements** :
  le mot clé `'a lazy` remplace ici l'évaluation retardée explicite `fun () -> ...`
- bien sûr, cela **change aussi l'utilisation** du flot ; voyons comment...

Consommation d'un flot

Recherche du premier élément satisfaisant un prédicat, avec la **définition précédente** :

```
let rec first p f0 =
  match f0 ( ) with
  | Vide -> None (* pas de solution *)
  | NonVide (a, f1) ->
      if p a then Some a (* solution trouvee *)
      else first p f1 (* continuer a chercher *)
```

À l'aide de **types paresseux**, nous obtenons la définition :

```
let rec first p f0 =
  match Lazy.force f0 with
  | Vide -> None
  | NonVide (a, f1) ->
      if p a then Some a else first p f1
```

- à nouveau, **peu de changements** : `Lazy.force` remplace `f ()`

Double utilisation des calculs

Étude d'une **suite d'opérations** :

1. **déclaration d'un flot** : `let itv = stream_intv 0 200000000`
   temps $\mathcal{O}(1)$, espace $\mathcal{O}(1)$
2. **premier calcul** : `first (fun x -> x = 100000000) itv`
   lent, temps $\mathcal{O}(10^8)$, espace (tas) $\mathcal{O}(10^8)$
3. **second calcul** : `first (fun x -> x = 100000001) itv`
   beaucoup moins lent, temps $\mathcal{O}(10^8)$ mais constante faible, n'alloue qu'une cellule supplémentaire

Considérations pratiques importantes :

- la **mémoisation** a pour principe d'**utiliser de l'espace** pour **économiser du temps de calcul**
- on peut programmer **pour ne pas conserver un lien vers les calculs intermédiaires inutiles** : **c'est conseillé !**

Outline

Programmation séquentielle et limites

Jusqu'à présent, nous n'avons construit que des programmes en **style séquentiel** :

Définition : style séquentiel

Un **programme séquentiel** est tel que toutes les opérations **sont exécutées en séquence**, l'une après l'autre.

Exemple : `op_0; op_1; op_2; ...`

- on construit une structure, **puis** on la modifie, **puis** on la transforme, **et ensuite** on l'affiche...
- on construit un flot, **puis** on accède au premier élément, **puis** on effectue une opération dessus, **puis** on passe au suivant **et ainsi de suite**...

Inconvénient : **une opération lente est nécessairement bloquante**
(long calcul, requête sur le réseau, attente d'une entrée utilisateur)

Programmation concurrente

Solution : supprimer les points de synchronisation

Définition : style concurrent

Un **programme concurrent** est composé de **plusieurs tâches** qui **s'exécutent sans être synchronisées entre elles** (autrement dit, elles sont **concurrentes**).

Exemple : `op_0 | op_1` (lire `op_0` et `op_1` en parallèle)

- `op_0` et `op_1` sont supposés pouvoir s'exécuter **en parallèle** ou non,
  il est possible que `op_0` finisse avant le début de `op_1` ;
  il est possible que `op_1` finisse avant le début de `op_0` ;
  il est possible que les deux exécutions soient **entrelacées**...
- **avantage** :
  si `op_0` **bloque** (attend) ou **prend un temps considérable**, `op_1` peut quand même progresser
- on va voir plus loin des constructions permettant de **programmer "|"**

Communication

L'**absence de contraintes de synchronisation** induit un **possible non-déterminisme** des programmes concurrents, lorsque ceux-ci **partagent des ressources** : variables, effets de bords,...

Exemple : **data race**

```
x := -1 | x := 1
```

que vaut `!x` à la fin ? **pas clair, du fait des entrelacements possibles**

- si le premier programme s'exécute d'abord, alors `!x` vaut 1 ;
- si le second programme s'exécute d'abord, alors `!x` vaut -1.

Plusieurs modèles de communication :

- système **distribué** :
  plusieurs ordinateurs distants, communiquent via le réseau
- programme **parallèle** (ou **multi-threads**) :
  plusieurs composants d'un programme communiquent via une **mémoire partagée**

Synchronisation

Un **non-déterminisme excessif** peut être **très gênant**.

Exemple :

`op_0` | `op_1`

- `op_0` lit des données par blocs sur le réseau et les organise en mémoire
- `op_1` traite les informations en mémoire par bloc
- il faut **éviter le traitement d'un état incohérent**
  e.g., bloc partiellement écrit...

Solution : ajout de **mécanismes de synchronisation**
i.e., **qui rendent certains entrelacements impossibles**

- **mutex** : indique si une **ressource est disponible**
- **acquisition** nécessaire avant d'utiliser une ressource
- **libération** après utilisation
- principe d'**exclusion mutuelle** : seul l'un des `op_i` peut avoir le mutex à un instant donné, sinon il faut attendre

Synchronisation : difficulté

Considérons le programme (conceptuel) suivant :

`prog0`	`prog1`
acquire m_A	**acquire** m_B
acquire m_B	**acquire** m_A
...	...
release m_B	**release** m_A
release m_A	**release** m_B

Une exécution possible (l'un des nombreux entrelacements possibles) :

1. `prog0` acquiert le mutex m_A
2. `prog1` acquiert le mutex m_B
3. `prog0` attend m_B et `prog1` attend m_A **pour toujours**

On parle d'**interbloquage** (ou **dead-lock**)...
C'est une situation **très gênante** puisque **tout le monde est bloqué**.

Plusieurs paradigmes de programmation asynchrone

Il existe **de nombreuses formes de concurrence** :

- **création de plusieurs processus** : chacun sa mémoire
  possible en OCaml, par exemple via le module `Unix` (librairie standard)
- **un processus, plusieurs threads vraiment parallèles** :
  actuellement non supporté en OCaml
  raison : GC concurrent indisponible
- **un processus, plusieurs threads légers** (ou fibres) :
  à chaque instant, un thread léger s'exécute, et peut laisser la place à un autre à tout moment, à son initiative ;
  plusieurs librairies `Async`, `Lwt`.

Dans la suite, nous allons nous concentrer sur les **threads légers** (ou **fibres**), après une brève discussion des **threads classiques**.

Threads classiques

Un **thread système** est une entité qui dispose d'un contexte interne et **peut être interrompu à tout moment par le système** (scheduling).

Opérations importantes du module `Thread` de OCaml :

- **Création**, via

  ```
  Thread.create: ('a -> 'b)-> 'a -> Thread.t
  ```

  `(Thread.create f a)` crée un nouveau thread exécutant `f a`
- **Attente**, via

  ```
  Thread.join: t -> unit
  ```

  `(Thread.join t)` force le thread courant à attendre la fin de `t` (attention aux interbloquages!)
- **Synchronisation** : `Thread.Mutex`

En OCaml, pas d'exécution parallèle

Threads légers

Un **thread léger** est une entité qui dispose d'un contexte interne mais **ne peut être interrompu que de son plein gré**.

On parle de **threads coopératifs**.

Nous considérons la librairie Ocsigen `Lwt`, qui est très utilisée actuellement, mais les principes sur lesquels repose `Async` sont les mêmes.

Installation :

```
opam install lwt
```

Threads Lwt (aussi appelés promesses)

Le type `'a Lwt.t` décrit une **promesse** d'une valeur de type `'a`, qui correspond au **résultat d'un thread** décrivant :

- soit une **valeur disponible** `x` ;
- soit une **exception** `exn` ;
- soit un **calcul en sommeil** (état initial).

Type et **fonctions de construction** :

```
type 'a Lwt.t
  (* = Return of 'a | Sleep | Fail of exn *)
val Lwt.return: 'a -> 'a Lwt.t
val Lwt.fail: exn -> 'a Lwt.t
val Lwt.wait: unit -> 'a Lwt.t * 'a Lwt.u
```

L'expression `Lwt.return true` renvoie non pas `true`, mais un thread, prêt à renvoyer la valeur `true`, lorsqu'on le lui demandera.

Ce type peut paraître obscur, mais nous allons bientôt voir comment l'utiliser...

Application : entrées et sorties

Fonction classique :

```
val input_char: in_channel -> char
```

- **attend un caractère** sur un canal d'entrée
- i.e., tant que le caractère n'arrive pas, **reste bloquée**

Fonction Lwt :

```
val Lwt.read_char: Lwt_io.in_channel -> char Lwt.t
```

- procède également **à la lecture d'un caractère**
- mais ne délivre qu'une **promesse d'un caractère**

Différence fondamentale :
dans le cas de la fonction de Lwt, **un autre thread peut s'exécuter en attendant** l'arrivée d'un caractère

Contrainte : seules les fonctions de Lwt garantissent cela

Chaînage

Considérons la **fonction de chaînage** `bind` :

```
val Lwt.bind: 'a Lwt.t -> ('a -> 'b Lwt.t) -> 'b Lwt.t
```

Dans l'appel `Lwt.bind t f`

1. on attend d'abord que le thread `t` renvoie une valeur `v`
2. puis on passe la valeur obtenue à `f`
3. lorsque `f` retourne, on renvoie la valeur résultante

Application :

```
Lwt.bind
  (Lwt_io.read_line Lwt_io.stdin)
  (String.split_on_char ' ')
```

- lorsqu'une ligne peut être lue sur l'entrée...
- lire celle-ci, puis la découper en mots

Exemple

Voyons maintenant **quelques exemples brefs** (inspirés par des exemples présentés sur les sites `lwt` et `Ocsigen`), un programme plus conséquent étant prévu en TD.

Horloge paramétrée par une **durée** :

```
let rec clock d =
  print_endline "tick";
  let t = Lwt_unix.sleep d in
  Lwt.bind t (fun () -> clock d)
```

- produit d'abord un affichage (immédiat)
  ensuite, effectue une attente **puis** se rappelle récursivement
- structure assez **comparable à celle d'un flot**...

Exemple

Traitement des éléments d'une liste en parallèle (`map`) :

```
let rec map_conc f l =
  match l with
  | [ ] -> Lwt.return [ ]
  | x :: m ->
      let x0 = f x
      and m0 = map_conc f m in
      Lwt.bind x0
        (fun x1 ->
          Lwt.bind m0
           (fun m1 ->
             Lwt.return (x1 :: m1)))
```

- **type** `('a -> 'b Lwt.t) -> 'a list -> 'b list Lwt.t`
- **pas de point de séquencement** entre `f x` et `map_conc f m`
- l'évaluation des deux **doit avoir lieu avant production du résultat** `x1 :: m1`

Exemple

Traitement des éléments d'une liste en séquentiel (map) :

```
let rec map_seq f l =
  match l with
  | [ ] -> Lwt.return [ ]
  | x :: m ->
      Lwt.bind (f x)
        (fun x0 ->
          Lwt.bind (map_seq f m)
           (fun m0 ->
             Lwt.return (x0 :: m0)))
```

- **type** `('a -> 'b Lwt.t) -> 'a list -> 'b list Lwt.t`
- **point de séquencement** entre `f x` et `map_seq f m`
- **point de séquencement** entre `map_seq f m` et production de `x0 :: m0`

Exécution d'un programme complet

Vie d'un thread :

- **initialement en sommeil** : pas encore complètement calculé
- peut passer vers un état décrivant un **calcul achevé** ou une **exception**
- peu gênant lorsqu'on **compose des threads**
- plus problématique pour le **programme complet** :
  quand s'exécute-t'il ?

On peut **forcer l'évaluation** :

```
val return: 'a Lwt.t -> 'a
```

Principe :

la fonction `return` abstrait une **boucle d'événements**

Exemple de programme complet :

```
let t1 = Lwt.bind t0 f0 in
let t2 = Lwt.bind t1 f1 in
Lwt.run t2
```

Join

Jusqu'à présent nous avons vu des programmes qui **composent des threads l'un après l'autre** ; qu'en est il lorsqu'on souhaite **lancer plusieurs threads en parallèle** ?

Attente $\forall$: [plusieurs threads] [attente que tous terminent]

```
val join: unit Lwt.t list -> unit Lwt.t
```

- `Lwt.join [ t0 ; t1 ; ... ; tn ]` renvoie quand **tous** les `t0`, ..., `tn` **ont renvoyé**
- on peut bien sûr faire `Lwt.bind (Lwt.join l) f`

Attente $\exists$: [plusieurs threads] [attente qu'un termine]

```
val Lwt.choose: 'a Lwt.t list -> 'a Lwt.t
```

- `Lwt.choose [ t0 ; t1 ; ... ; tn ]` renvoie quand **l'un** des `t0`, ..., `tn` **a renvoyé**, et produit la **même valeur ou exception**
- on peut également faire `Lwt.choose (Lwt.join l) f`

Mutexes

Les **synchronisations entre threads** sont **rarement nécessaires** en `Lwt`.

Un exemple : les **mutexes**, définis par `Lwt_mutex.t`
et créés par la fonction `Lwt_mutex.create: unit -> Lwt_mutex.t`
Un mutex agit comme un **verrou qu'un seul thread peut acquérir à un instant donné** (mais il peut passer **d'un thread à l'autre**).

Verrouillage : `Lwt_mutex.lock: Lwt_mutex.t -> unit Lwt.t`
Considérons un thread faisant `Lwt_mutex.lock m`

- si `m` est **libre**, le thread **continue** et `m` **est verrouillé**
- si `m` est **verrouillé**, le thread **attend sa libération**

Autres opérations :

- **libération** `Lwt_mutex.unlock: Lwt_mutex.t -> unit`
- **test** au verrouillage `Lwt_mutex.is_locked: Lwt_mutex.t -> bool`

Outline

Principaux éléments à retenir

Nous avons vu de nouvelles manières d'**organiser les liens entre production et consommation** de données :

- **flots** : données organisées comme des **suites de valeurs** qui sont **produites incrémentalement**
- **threads** : calculs **non synchronisés** entre eux
  pouvant être effectués dans n'importe quel ordre ou en même temps

Dans les deux cas, on **manipule des calculs en cours**
(suspensions `fun () ->`, valeur `lazy`, promesses...).

Deux éléments importants :

- les **points de séquencement** :
  pour quelles opérations a-t'on une contrainte d'ordre ?
- la **mémoire utilisée** :
  quelles données sont conservées en mémoire ?

Travaux

1. **Flot sur les éléments d'un arbre** (transparent 20) :
   écrire et tester (par exemple en affichant les éléments)

2. **Exercice 6.10** (poly)

3. **Exercice 6.11** (poly)

4. En vous inspirant des fonctions `map_seq` et `map_conc` sur les listes, faire de même sur un arbre binaire...
   (par exemple avec une fonction d'affichage)