Programmation Fonctionnelle Avancée

Université Paris Diderot

Vincent Balat

janvier 2010

Table des matières

Chapitre 1

Structures de données et algorithmes

1.1 Partage et copie

Concaténation de listes

```
# let rec append l1 l2 = match l1 with
    | [] -> l2
    | a::l -> a::(append l l2);;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [1; 2] [3; 4];;
- : int list = [1; 2; 3; 4]
# 1::(2::[]);;
- : int list = [1; 2]
```

Arbre binaires de recherche

```
# type arbre = Feuille | Noeud of (int * arbre * arbre);;
type arbre = Feuille | Noeud of (int * arbre * arbre)
# let rec ajout v a = match a with
    | Feuille -> Noeud (v, Feuille, Feuille)
    | Noeud (w, a1, a2) -> if v <= w
       then Noeud (w, (ajout v a1), a2)
       else Noeud (w, a1, ajout v a2);;
val ajout : int -> arbre -> arbre = <fun>
```

1.2 Files (FIFO)

1re version (fonctionnelle, coût élevé)

Interface

```
module type FIFO = sig
  type 'a t
  exception Empty
  val empty : 'a t
  val is_empty : 'a t -> bool
  val add : 'a -> 'a t -> 'a t
  val remove : 'a t -> ('a * 'a t)
  (** raises [Empty] if the queue is empty *)
```

```
end
```

Implémentation

```
module Fifo : FIFO = struct
  type 'a t = 'a list
  let empty = []
  let is_empty f = f = []
  let add a f = f@[a]
  let remove = function
  | [] -> raise Empty
  | a::l -> (a, l)
end
```

2^e version (impérative)

Interface

```
module type FIFOIMP = sig
  type 'a t
  exception Empty
  val create : unit -> 'a t
  val add : 'a -> 'a t -> unit
  val remove : 'a t -> 'a
  (** raises [Empty] if the queue is empty *)
end
```

Implémentation

```
module Fifo2 : FIFOIMP = struct
  type 'a content = Vide | Cons of 'a * 'a content ref;;

  type 'a t = {mutable first: 'a content; mutable last : 'a content};;

  exception Empty

  let create () = {first = Vide; last = Vide};;

  let add a f =
  match f.last with
    | Vide -> (* Dans ce cas F.first doit être Vide aussi *)
      f.first <- Cons (a, ref Vide);
      f.last <- f.first;
    | Cons (_, r) ->
      r := Cons (a, ref Vide);
      f.last <- !r

  let remove f = match f.first with
  | Vide -> raise Empty
  | Cons (a, r) ->
     if f.last = f.first
     then f.last <- !r;
     f.first <- !r ; a

  let is_empty f = f.first = Vide
end
```

3^e version (fonctionnelle, plus efficace)

Interface : la même que la première version

Implémentation

```
module Fifo3 : FIFO = struct
  type 'a t = 'a list * 'a list

  exception Empty

  let empty = ([], [])

  let add x (l1, l2) = (x::l1, l2)

  let remove (l1, l2) = match l2 with
  | a::l -> (a, (l1, l))
  | [] -> match List.rev l1 with
      | [] -> raise Empty
      | a::l -> (a, ([], l));;
end
```

Notion de coût *amorti*.

On peut étendre cette idée par exemple aux « doubles files » (*dqueues* ou *deques*), c'est-à-dire à des files dans lesquelles on peut écrire et lire à n'importe quel bout. Quand l'une des listes est vide, on coupe l'autre en deux et retourne la moitié.

1.3 Flux *(streams)*

```
# type flux = Fini | Cont of (string * (unit -> flux));;
type flux = Fini | Cont of (string * (unit -> flux))
# let monflux = Cont (1, fun () -> Cont (2, fun () -> Fini));;
val monflux : int flux = Cont (1, <fun>)
# match monflux with Cont (v, f) -> f ();;
Characters 0-38:
  match monflux with Cont (v, f) -> f ();;
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Fini
- : int flux = Cont (2, <fun>)
# let rec f i = Cont (2*i, fun () -> f (i+1));; (* flux des entiers pairs *)
val f : int -> int flux = <fun>
# let rec f () =
    let s = read_line () in
    if s = "quitter"
    then Fini
    else Cont (s, f);;
val f : unit -> flux = <fun>
# let rec lit_flux flux = match flux with
  | Fini -> ()
  | Cont (s, f) -> print_string "J'ai lu : ";
     print_endline s;
```

```
     lit_flux (f ());;
val lit_flux : string flux -> unit = <fun>
# lit_flux (f ());;
qdsqsd
J'ai lu : qdsqsd
qsdqs
J'ai lu : qsdqs
qsqdqdf
J'ai lu : qsqdqdf
qsqqs
J'ai lu : qsqqs
qsdqsd
J'ai lu : qsdqsd
quitter
- : unit = ()
```

Concaténation de flux :

```
# let rec concat fl1 fl2 =
    match fl1 with
    | Fini -> fl2
    | Cont (s, f) -> Cont (s, fun () -> concat (f ()) fl2);;
val concat : flux -> flux -> flux = <fun>
```

1.4 Tableaux et chaînes de caractères

Un tableau est une table d'association à clés entières. Plusieurs implémentation fonctionnelles nous donnent des structures avec un accès rapide aux données ($O(log(n))$) qui ont en plus la propriété d'être extensibles (contrairement aux tableaux).

1.4.1 Tableaux extensibles

Utilisation d'arbre binaires équilibrés (module `Map`). Voir aussi le module `Vec` de Luca de Alfaro, qui optimise les arbres binaires équilibrés à ce cas particulier.

1.4.2 Tableaux fonctionnels de Paulson

1991

Arbre binaires avec valeurs aux noeuds. L'élément d'indice 1 est à la racine. Pour trouver un autre élément, on divise son indice par deux et l'on va à droite si le reste de la division est pair, à gauche s'il est impair (autrement dit : décomposition en binaire sans se préoccuper du premier 1).

Temps de recherche d'un élément : logarithmique. Mais ces « tableaux » peuvent être étendus facilement, sans avoir à recopier toute la structure.

1.4.3 Buffer

Module `Buffer` d'Ocaml. Chaînes de caractères qui grossisent automatiquement. Ajout de caractères par concaténation en temps quasi-linéaire.

Principe : les chaînes sont stockées dans un buffer plus grand, ce qui permet de rallonger la chaîne facilement. La copie du buffer n'a lieu que lorsqu'il est trop petit. On double sa taille à chaque fois, pour garder un buffer dont du même ordre de grandeur que la chaîne stockée.

Donc évitez le code du style `s1^s2^s3^s4` qui est quadratique (les concaténations sont faites par deux, en recopiant les deux chaînes à chaques fois). À la place utilisez `String.concat` ou des buffers.

Cela reste une structure de donnée impérative, avec modification en place (on ne garde pas l'ancienne valeur du buffer).

1.4.4 Cordes *(Ropes)*

Boehm, Atkinson et Plass.

Les cordes sont une implémentation des chaînes qui permet la concaténation rapide, sans copie de la chaîne, et en permettant la persistence (pas de mutation de la chaîne).

Principe : On représente les cordes par des arbres binaires dont les noeuds sont des chaînes. Concaténer une chaîne revient juste à rajouter un noeud.

1.5 Zipper

Se promener (en avant en arrière) sur une structure de données, ajouter des valeurs au milieu, sans pointeurs arrières.

1.5.1 Exemple sans zipper : les listes doublement chaînées

Avec des références

1.5.2 Zipper des listes

Gérard Huet 1997

```
type 'a chemin = 'a list

type 'a listzipper = 'a chemin * 'a list

let a_gauche (chemin, liste) = match chemin with
| [] -> failwith "Déjà à gauche"
| a::l -> (l, a::liste)

let a_droite (chemin, liste) = match liste with
| [] -> failwith "Déjà à droite"
| a::l -> (a::l, liste)
```

1.5.3 Zipper des arbres

Arbres binaires

```
type 'a arbre =
  | Feuille
  | Noeud of 'a * 'a arbre * 'a arbre

type 'a chemin =
  | Haut
  | Gauche of 'a * 'a arbre * 'a chemin
  | Droite of  'a * 'a arbre * 'a chemin

type 'a zipper = Zip of 'a arbre * 'a chemin

let a_gauche (Zip (arbre, chemin)) = match arbre with
| Feuille -> failwith "Feuille"
| Noeud (x, t1, t2) -> Zip (t1, Gauche (x, t2, chemin))

let a_droite (Zip (arbre, chemin)) = match arbre with
| Feuille -> failwith "Feuille"
```

```
| Noeud (x, t1, t2) -> Zip (t2, Droite (x, t1, chemin))

let en_haut (Zip (arbre, chemin)) = match chemin with
| Haut  -> failwith "Haut"
| Gauche (x, t, chemin')  -> Zip (Noeud (x, arbre, t), chemin')
| Droite (x, t, chemin') -> Zip (Noeud (x, t, arbre), chemin')
```

Arbres n-aires

```
type 'a arbre =
| Feuille of 'a
| Noeud of 'a arbre list

type 'a chemin =
| Haut
| Dueon of 'a arbre list * 'a chemin * 'a arbre list

type 'a zipper = Zip of 'a arbre * 'a chemin

let a_gauche (Zip (t, p)) = match p with
| Haut -> failwith "rien à gauche (en haut)"
| Dueon(l::gauche, up, droite) -> Zip(l, Dueon(gauche, up, t::droite))
| Dueon([], up, droite) -> failwith "rien à gauche"

let a_droite (Zip (t, p)) = match p with
| Haut -> failwith "rien à droite (en haut)"
| Dueon (gauche, up, r::droite) -> Zip(r, Dueon(t::gauche, up, droite))
| _ -> failwith "rien à droite"

let en_haut (Zip (t, p)) = match p with
| Haut -> failwith "rien en haut"
| Dueon(gauche, up, droite) ->
     Zip (Noeud((List.rev gauche) @ (t::droite)), up)

let en_bas (Zip (t, p)) = match t with
| Feuille _ -> failwith "rien en bas"
| Noeud (t1::arbres) -> Zip (t1, Dueon([], p, arbres))
| _ -> failwith "rien en bas"
```

Autres utilisations

Éditeur de texte (structuré)

Utilisation dans le gestionnaire de fenêtres XMonad (Haskell).

In this picture we have a window manager managing 6 virtual workspaces. Workspace 4 is currently on screen. Workspaces 1, 2, 4 and 6 are non-empty, and have some windows. The window currently receiving keyboard focus is window 2 on the workspace 4. The focused windows on the other non-empty workspaces are being tracked though, so when we view another workspace, we will know which window to set focus on. Workspaces 3 and 5 are empty.

1.6 Paresse *(laziness)*

Pour être paresseux, il faut :

- Ne travailler que si c'est vraiment nécessaire ;
- Ne jamais refaire quelque chose que l'on a déjà fait.

La première proposition implique de retarder au maximum le travail (calcul) jusqu'à ce que l'on soit sûrs d'avoir besoin du résultat.
Avantage de la paresse : on travaille moins
Inconvénients : on risque d'être pris de court (gros calcul à effectuer alors que l'on n'a plus de temps)

Pour retarder un calcul on a déjà vu (flux) que l'on peut remplacer une valeur de type `'a` par une fonction de type `unit -> 'a`. Mais si on l'utilise plusieurs fois, on fera le calcul plusieurs fois... Solution : mémoïsation

```
# let rec boucle = function 0 -> () | n -> boucle (n-1);;
val boucle : int -> unit = <fun>
# let gros_calcul () = boucle 100000000; 4;;
val gros_calcul : unit -> int = <fun>
# let v = gros_calcul ();;  (* c'est long *)
val v : int = 4
# v + 1;; (* c'est rapide *)
- : int = 5
# let v () = gros_calcul ();; (* c'est rapide *)
val v : unit -> int = <fun>
# v () + 1;; (* c'est long *)
- : int = 5
# v () + 1;; (* c'est long *)
- : int = 5
# let v =
    let r = ref None in
    fun () -> match !r with
              | Some v -> v
              | None -> let v = (gros_calcul ()) in
                        r := Some v;
                        v;;
val v : unit -> int = <fun>
# v () + 1;; (* c'est long *)
- : int = 5
# v () + 1;; (* c'est rapide *)
- : int = 5
```

En fait cette technique est tellement courante qu'OCaml implémente ce comportement de manière native avec la syntaxe spéciale **`lazy`** et le module `Lazy` :

```
# let v = lazy (gros_calcul ());;
val v : int lazy_t = <lazy>
# Lazy.force v;; (* c'est long *)
- : int = 4
# Lazy.force v;; (* c'est rapide *)
- : int = 4
```

La valeur `v` est appelée une *suspension*.
Allez voir l'interface du module `Lazy`.

Les flux réimplémentés avec la paresse :

```
# type flux = Fini | Cont of string * flux Lazy.t;;
type flux = Fini | Cont of string * flux Lazy.t
```

Par rapport à la précédente implémentation, ces flux permettent la *persistence* de la structure de données, alors que dans le cas précédent, le flux était détruit lors du calcul (et réutiliser le flux impliquait de refaire les calculs).

```
# let rec concat fl1 fl2 =
    match fl1 with
    | Fini -> fl2
    | Cont (s, f) -> Cont (s, lazy (concat (Lazy.force f) fl2));;
val concat : flux -> flux -> flux = <fun>
```

Comparer avec :

```
# let conc l1 l2 = lazy ((Lazy.force l1) @ (Lazy.force l2));;
val conc : 'a list Lazy.t -> 'a list Lazy.t -> 'a list lazy_t = <fun>
```

La fonction `conc` est dite *monolithique* alors que `concat` est *incrémentale*.
Exercice : écrire `reverse`. Est-elle monolithique ou incrémentale ?

1.7 Utilisation de la paresse pour améliorer le coût

Dans cette section nous allons améliorer notre structure de files fonctionnelles en utilisant la paresse. Cette structure de données est due à Chris Okasaki.

Principe

Nous avons déjà implémenté une version fonctionnelle des files dont le coût de l'ajout est $O(1)$ et le coût du retrait est en moyenne $O(1)$ mais au pire $O(n)$. Cette implémentation n'est cependant pas satisfaisante pour une utilisation « temps réel » où les coûts doivent être réguliers. Dans cette section nous allons voir une autre version des files, fonctionnelle, mais pour laquelle le coût de chaque opération est toujours $O(1)$.

Rappelons l'implémentation de `List.rev` :

```
# let rev l =
    let rec aux acc = function
     | [] -> acc
     | x::xs -> aux (x::acc) xs
    in aux [] l
val rev : 'a list -> 'a list = <fun>
```

Idée : plutôt que retourner la liste en une fois, nous allons faire une étape du retournement à chaque ajout. Autrement dit, nous allons utiliser une version incrémentale de `List.rev` plutôt que la version monolithique. Pour cela nous allons remplacer la liste de tête de la file (d'où l'ont retire les éléments) par un flux paresseux. Le calcul d'une des suspensions de ce flux correspond à une étape du retournement de la liste. Nous voulons forcer une suspension de ce flux à chaque ajout et retrait dans la file. Pour cela, nous allons garder un pointeur vers la première suspension non évaluée du flux.

Notre structure de donnée `'a t` des files sera donc définie par :

```
type 'a fluxcell = Vide | Cont of 'a * 'a flux
and 'a flux = 'a fluxcell Lazy.t

type 'a t = 'a list * 'a flux * 'a flux
```

La liste représente la queue de la file (ajout). Le premier flux est la tête (retrait). Le deuxième flux est en fait juste un pointeur vers un sous-flux du premier (on donc du partage) qui correspond à sa première suspension non encore évaluée. Forcer la tête du deuxième flux aura donc pour effet de forcer un élément du premier.

Remarques : Notons (l, f, s) le triplet.

– s peut être vu comme un pointeur vers un noeud de f.

- La liste l est « vidée » dans f lorsque le flux s est vide (et non plus f, comme dans la précédente implémentation).
- Lors de ce (début de retournement) « retournement », on modifie f complètement. Donc s devient égal à f.
- Lorsque l'on ajoute un élément dans l, si s n'est pas vide, alors on force son premier élément et on l'enlève de s (ce qui revient à décaler le pointeur s d'une case dans f).
- On peut montrer facilement que $|l| + |s| = |f|$ (où la notation $|\cdot|$ représente la longueur d'une liste ou d'un flux). Idée de la preuve : Lorsque l'on retourne l, on a $s = f$, et l'on diminue s d'un à chaque fois que l'on ajoute un élément dans l ou que l'on en enlève un de f.

Implémentation

L'interface du module est la même qu'avant.

L'opération qui force la première suspension de s est la suivante :

```
let force_s (l, f, s) = match Lazy.force s with
  | Cont (_, ss) -> (l, f, ss)
  | Vide -> let f' = retourne l f (lazy Vide) in ([], f', f')
```

Dans le cas où s n'est pas vide, elle se contente de forcer sa tête et de l'enlever. Sinon elle fait une rotation (incrémentale) de l en utilisant la fonction `retourne` qui crée le nouveau flux f en retournant l à la fin de l'ancien. En fait, plutôt qu'implémenter une fonction `retourne` générique, on va remarquer que dans l'algorithme, la rotation a toujours lieu quand $|l| = |f| + 1$, ce qui permet de simplifier la fonction. Cette fonction utilise un accumulateur, comme `List.rev`.

```
let rec retourne l f acc = match (l, Lazy.force f) with
  | (y::ys, Cont (v, fs)) ->
    lazy (Cont (v, retourne ys fs (lazy (Cont (y, acc)))))
  | ([y], Vide) -> lazy (Cont (y, acc))
  | _ -> assert false (* cas impossibles grâce à notre invariant *)
```

Implémenter le retrait est facile (ne pas oublier de forcer une suspension de s en appelant `force_s`) :

```
let remove (l, f, s) = match Lazy.force f with
  | Vide -> raise Empty
  | Cont (a, ff) -> (a, force_s (l, ff, s))
```

L'ajout est simple aussi :

```
let add x (l, f, s) = force_s (x::l, f, s)
```

La figure 1.1 montre l'implémentation de tout le nouveau module de files.

Conclusion

D'après l'auteur, le coût de `force_s` est constant (les calculs sont retardés grâce au mot-clé **`lazy`**). L'utilisation d'une structure de données paresseuse nous a permis de transformer un coût amorti (complexité en moyenne $O(1)$) en un coût fixe (complexité dans le pire des cas $O(1)$).

```
module Fifo4 : FIFO = struct
  type 'a fluxcell = Vide | Cont of 'a * 'a flux
  and 'a flux = 'a fluxcell Lazy.t

  type 'a t = 'a list * 'a flux * 'a flux

  exception Empty

  let rec retourne l f acc = match (l, Lazy.force f) with
  | (y::ys, Cont (v, fs)) ->
    lazy (Cont (v, retourne ys fs (lazy (Cont (y, acc)))))
  | ([y], Vide) -> lazy (Cont (y, acc))
  | _ -> assert false (* cas impossibles grâce à notre invariant *)

  let force_s (l, f, s) = match Lazy.force s with
  | Cont (_, ss) -> (l, f, ss)
  | Vide -> let f' = retourne l f (lazy Vide) in ([], f', f')

  let remove (l, f, s) = match Lazy.force f with
  | Vide -> raise Empty
  | Cont (a, ff) -> (a, force_s (l, ff, s))

  let add x (l, f, s) = force_s (x::l, f, s)

  let empty = let f = lazy Vide in ([], f, f)

  let is_empty (_, f, _) = Lazy.force f = Vide
end;;
```

FIGURE 1.1 – Files fonctionnelles efficaces

Chapitre 2

Concurrence

Dans ce chapitre nous allons étudier plusieurs styles de programmation concurrente.

La méthode la plus rudimentaire pour simuler la concurrence est la *boucle d'événement*. Le processus a une boucle principale qui attend des événements et les traite dans l'ordre d'arrivée. Les traitements doivent être rapides. Sinon ils doivent être découpés manuellement en tâches plus petites (qui placent leur *continuation* dans la file d'événement).

Nous allons voir deux techniques plus évoluées : les threads et la coopération. Nous étudierons également plusieurs techniques pour comuniquer entre threads.

2.1 Threads préemptifs

Implémentés par le système d'exploitation (threads natifs) ou la machine virtuelle (« VM threads » ou « green threads »).

2.1.1 Création et lancement d'un thread en OCaml

```
val create : ('a -> 'b) -> 'a -> t
val self : unit -> t
val id : t -> int
val exit : unit -> unit
val kill : t -> unit
```

2.1.2 Synchronisation

Java	OCaml	
`Thread.sleep(1000)`	`Thread.delay(1.0)`	Attente de durée « fixe »
`t.join()`	`Thread.join t`	Attente de la terminaison d'un thread
	`Thread.select ...`	Attente sur descripteurs de fichiers (Unix seulement)
	`Thread.yield ()`	Donne la main à un autre thread
`c.wait()`	`Condition.wait c m`	Attendre un événement
`c.notify()`	`Condition.signal c`	Réveiller un thread en attente d'un événement
`c.notifyAll()`	`Condition.broadcast c`	Réveiller les threads en attente d'un événement

2.2 Communication entre threads

2.2.1 Mémoire partagée

($\neq$ mémoire répartie)

À première vue la façon la plus simple d'échanger des informations entre threads.

On parle de *sections critiques*, qui doivent être exécutées en *exclusion mutuelle*. Mécanismes de synchronisation nécessaires pour accéder aux données (éviter les « race conditions »/situations de compétition). Mais risques d'oublis, d'interblocages, de famine.

Mutexes

En OCaml

```
let m = Mutex.create () in
...
Mutex.lock m;
(* section critique *) ...
Mutex.unlock m;
```

2.2.2 Passage de messages

Pour éviter les problèmes de la mémoire partagée, on peut décider de ne communiquer entre threads que par passage de messages.

Avantage : pas de problème d'accès concurrent

Inconvénients : Oblige à penser le programme différemment. Par exemple deux threads ne peuvent pas utiliser la même grosse structure de données.

Messages synchrones : bloquent l'émetteur tant que le récepteur n'est pas prêt à recevoir, et vice-versa.

Messages asynchrones : La réception est découplée de l'envoi. Il y a souvent une file d'attente des messages.

Canaux synchrones typés

Exemple du module `Event` de OCaml :

Extrait de l'interface :

```
type 'a channel
The type of communication channels carrying values of type 'a.

val new_channel : unit -> 'a channel
Return a new channel.

type +'a event
The type of communication events returning a result of type 'a.

val send : 'a channel -> 'a -> unit event
send ch v returns the event consisting in sending the value v over the channel ch.
The result value of this event is ().

val receive : 'a channel -> 'a event
receive ch returns the event consisting in receiving a value from the channel ch.
The result value of this event is the value received.

val sync : 'a event -> 'a
``Synchronize'' on an event: offer all the communication possibilities specified in
the event to the outside world, and block until one of the communications succeed.
The result value of that communication is returned.
```

```
val poll : 'a event -> 'a option
Non-blocking version of Event.sync: offer all the communication possibilities
specified in the event to the outside world, and if one can take place immediately,
perform it and return Some r where r is the result value of that communication.
Otherwise, return None without blocking

val choose : 'a event list -> 'a event
choose evl returns the event that is the alternative of all the events in the list evl.

val select : 'a event list -> 'a
``Synchronize'' on an alternative of events. select evl is shorthand for
sync(choose evl).

val always : 'a -> 'a event
always v returns an event that is always ready for synchronization.
The result value of this event is v.
```

Notes : `send` et `receive` ne sont pas bloquantes.

Exemple : référence partagée

```
let ch = Event.new_channel ()
let v = ref 0
let reader () = Event.sync (Event.receive ch)
let writer () = Event.sync (Event.send ch ("S" ^ (string_of_int !v)))

let loop_reader s d () =
  for i=1 to 10 do
    let r = reader() in print_string (s ^ " " ^ r); print_newline();
    Thread.delay d
  done

let loop_writer d () =
  for i=1 to 10 do incr v; writer(); Thread.delay d done

let _ =
  Thread.create (loop_reader "A" 1.1) ();
  Thread.create (loop_reader "B" 1.5) ();
  Thread.create (loop_reader "C" 1.9) ();
  Thread.delay 2.0;
  loop_writer 1. ()
```

(pour exécuter au toplevel `ocaml -I +threads unix.cma threads.cma`)

Exemple : gensym

```
type uid = UID of string Event.channel;;
let makeUidSrc () =
  let ch = Event.new_channel () in
  let rec loop i =
    Event.sync (Event.send ch ("S"^(string_of_int i)));
    loop (i+1)
  in
  Thread.create (fun () -> loop 0) () ;
  UID ch
```

```
let getUid (UID ch) = Event.sync(Event.receive ch)

let ch1 = makeUidSrc ()

let main ti msg () =
  while true do
    Thread.delay(ti);
    let r = getUid ch1 in
    print_string (msg); print_string " -- ";
    print_string r; print_newline();
  done

let _ = ignore (Thread.create (main 1.1 "A") ());
        ignore (Thread.create (main 2.1 "B") ())
```

2.3 Threads coopératifs avec Lwt

2.3.1 Les threads coopératifs

- Plus d'ordonnanceur préemptif
- Les threads s'exécutent sans interruption jusqu'à ce qu'ils fassent un appel à une fonction implémentée de façon à laisser la main à un autre thread (on aura une fonction `yield`, qui se contente de laisser la main, mais aussi une implémentation de `read`, `sleep`, etc.)
- Lors d'un tel *point de coopération*, le thread courant est mis en attente dans une file et le plus vieux thread prêt est relancé.

Le principe est très proche de celui des *co-routine* (routines possédants plusieurs points d'entrée permettant de continuer après chaque `yield`).

Inconvénients :

- Danger : Il ne faut pas appeler de fonction bloquante, ou qui prend trop de temps, sinon tous les threads vont être bloqués. En particulier, il ne faut pas appeler `Unix.sleep` ni `Unix.read`, mais les fonctions coopératives correspondantes.
- On ne tire pas directement partie des multi-processeurs, puisque tout le programme est en fait séquentiel. (Pour en tirer partie il faut organiser le programme de manière à mêler coopération et « vrais » threads).

Avantages :

- C'est très léger ;
- Quasiment plus de problème d'accès concurrent alors que l'on a la mémoire partagée ! Une portion de code située entre deux points de coopération est toujours exécutée en exclusion mutuelle ;
- Plus facile de rendre l'application déterministe (et de comprendre ce qu'il se passe).

2.3.2 Principe général de Lwt

Une implémentation des threads coopératifs originale et simple à utiliser. Un grosse différence avec les threads préemptifs est le niveau de granularité des threads : au lieu de travailler avec quelques threads (fils d'exécution) bien définis, on a beaucoup de petits calculs dépendants les uns des autres, qui peuvent être exécutés de manière asynchrone, et une façon d'exprimer cette dépendance.

Lwt utilise un ordonnanceur (boucle principale) non préemptif qui est en fait une boucle d'événements. À chaque point de coopération, un thread redonne la main à l'ordonnanceur qui choisit le nou-

veau thread à exécuter.

Un « thread » qui calcule une valeur de type `'a` est représenté par une valeur du type abstrait `'a Lwt.t`. On parle plutôt de « *promesse* » plutôt que de thread. On utilisera le mot thread pour parler d'un enchaînement de promesses qui dépendent les unes des autres. Une promesse peut avoir trois états :

Terminée la valeur est déjà calculée ;

Endormie le calcul n'est pas terminé et pourra reprendre plus tard ;

Échec le calcul a échoué (avec une exception).

Par exemple la fonction `Lwt_unix.sleep`, version coopérative de `Unix.sleep` est de type `float -> unit Lwt.t`. Elle renvoie *immédiatement* une promesse de renvoyer `()` au bout d'un certain temps (si l'ordonnanceur lui rend la main).

La fonction `Lwt.return : 'a -> 'a Lwt.t` renvoie immédiatement une promesse terminée dont la valeur est passée en paramètre.

La fonction `Lwt.fail : exn -> 'a Lwt.t` renvoie immédiatement une promesse échouée avec l'exception passée en paramètre.

Il est possible de « chaîner » les promesses entre elles grâce à la fonction

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

L'expression `Lwt.bind p f` renvoie immédiatement une promesse de type `'b Lwt.t` calculée de la façon suivante :

- Si `p` est terminée, elle passe son résultat à la fonction `f`.
- Si `p` est endormie, elle sauvegarde `f` dans la liste des fonctions en attente du résultat de `p`. Lorsque `p` sera terminée, l'ordonnanceur réveillera ces fonctions les unes après les autres.
- Si `p` a échoué, alors elle échoue avec la même exception.

La fonction `bind` permet donc de faire de la synchronisation. Ne pas exécuter `f` tant que `p` n'est pas terminé. Notez que vous pouvez utiliser `bind` plusieurs fois sur la même promesse.

Exemples (au toplevel, en utilisant ocamlfind) :

```
 rlwrap ocaml
        Objective Caml version 3.11.1

# #use "topfind";;
# #require "lwt";;
# #load "/usr/lib/ocaml/unix.cma";;
# #load "/usr/local/lib/ocaml/3.11.1/lwt/lwt_unix.cma";;
# Lwt.bind (Lwt.return 3) (fun x -> print_int x; Lwt.return ());;
3- : unit Lwt.t = <abstr>
# Lwt.bind (Lwt_unix.sleep 3.0) (fun () -> print_endline "salut"; Lwt.return ());;
- : unit Lwt.t = <abstr>
```

- `Lwt.bind (Lwt.return 3) (fun x -> print_int x; Lwt.return ())` retourne immédiatement une promesse de type `unit Lwt.t` après avoir affiché `3`. Notez que le `Lwt.return ()` est nécessaire pour que l'appel à `bind` soit bien typé (on accroche une promesse sur une autre).
- `Lwt.bind (Lwt_unix.sleep 3.0) (fun () -> print_endline "salut"; Lwt.return ())` retourne immédiatement une promesse de type `unit Lwt.t` et ne fait rien d'autre.

Pour que la dernière commande se comporte comme on attend, il faut lancer la boucle principale (l'ordonnanceur), implémentée par la fonction :

```
Lwt_unix.run : 'a Lwt.t -> 'a
```

Cette fonction maintient une file des threads en attente et les relance dès qu'ils sont prêts (mais sans préemption). Elle s'arrête lorsque le thread passé en paramètre est terminé.
Quand on lance l'ordonnanceur, le calcul suspendu ci-dessus se termine :

```
# Lwt_unix.run (Lwt_unix.sleep 4.);;
salut
- : unit = ()
```

En fait on peut lancer l'ordonnanceur automatiquement et le faire interagir correctement avec le toplevel en faisant :

```
# #use "topfind";;
# #require "lwt";;
/usr/lib/ocaml/react: added to search path
/usr/lib/ocaml/react/react.cmo: loaded
/usr/local/lib/ocaml/3.11.1/lwt: added to search path
/usr/local/lib/ocaml/3.11.1/lwt/lwt.cma: loaded
# #require "lwt.unix";;
/usr/lib/ocaml/unix.cma: loaded
/usr/local/lib/ocaml/3.11.1/lwt/lwt_unix.cma: loaded
/usr/local/lib/ocaml/3.11.1/lwt/simple_top.cma: loaded
```

Cela charge le module `Lwt_unix`, mais aussi le module `Simple_top` qui s'occupe de lancer l'odonnanceur de menière transparente.

2.3.3 Le module `Lwt_unix`

Le module `Lwt_unix` implémente des versions coopératives de la plupart des fonctions du modules `Unix`, ainsi que l'ordonnanceur basé sur `select`. On peut parfois avoir à utiliser Lwt avec un autre ordonnanceur, par exemple sur une plateforme où le module `Unix` n'est pas disponible.

La fonction

```
Lwt_unix.yield : unit -> unit Lwt.t
```

permet d'ajouter manuellement un point de synchronisation. Elle place le thread en queue de la file des threads prêtes à exécuter. Elle est équivalente à `Lwt_sleep 0.`. En fait on l'utilise très rarement (uniquement lorsque vous voulez découper des très gros calculs qui pourraient bloquer tout le programme).

La fonction :

```
Lwt_unix.read : Lwt_unix.file_descr -> string -> int -> int -> int Lwt.t
```

s'utilise comme

```
Unix.read : Unix.file_descr -> string -> int -> int -> int
```

Sauf que la deuxième bloque (en tout cas par défaut) jusqu'à ce que les données arrivent, alors que la première renvoie toujours immédiatement une promesse du résultat. En fait `Lwt_unix.read` fait les actions suivantes :

- Elle essaie de lire sur le descripteur de fichier, de manière non bloquante ;
- S'il y a des données, elle retourne un résultat ;
- Sinon elle place la promesse dans une table des promesses en attente dont la clé est le descripteur de fichier.

Que fait `Lwt_unix.run` ?

- Elle maintient une file des promesses prêtes à exécuter. Tant que cette file n'est pas vide, elle les exécute dans l'ordre.

- Elle maintient une table des descripteurs de fichiers actuellement utilisés, avec les promesses qui attendent sur ces descripteurs. Elle fait l'appel système `select` sur ces descripteurs de fichiers [1] et met dans la file des promesses à exécuter celles qui ont reçues leur données, ou celles qui ont atteint leur délai.
- Elle recommence ces opérations et s'arrête lorsque la promesse qu'elle a reçu en paramètre est terminée.

Attention :

- Il est fortement recommandé ne n'utiliser `Lwt_unix.run` qu'une seule fois dans un programme. En tout cas ne pas avoir deux exécutions simultanées (sinon celle lancée en premier ne pourra pas se terminer tant que les suivantes ne sont pas terminées).
- `Lwt.bind` n'est pas un point de coopération.

2.3.4 Syntaxe

En général, pour simplifier l'écriture, on définit :

```
let (>>=) = Lwt.bind
```

ce qui permet d'utiliser l'opérateur infixe `>>=` au lieu de la fonction `Lwt.bind`.

Exemple de programme avec cette écriture :

```
Lwt_chan.input_line ch >>= fun s ->
Lwt_unix.sleep 3. >>= fun () ->
print_endline s;
Lwt.return ()
```

Le `Lwt.return ()` est nécessaire parce que la fonction doit renvoyer une promesse. Notez que l'on n'a pas besoin de parenthèses autour des fonctions. Notez aussi l'indentation : pour favoriser la lisibilité, il est conseillé de ne pas indenter la deuxième ligne [2]. En fait la version synchrone de ce programme est :

```
let s = input_line ch in
Unix.sleep 3.;
print_endline s
```

On voit bien qu'un `bind` est une version asynchrone du **`let`** ou du point-virgule. En fait il est possible d'utiliser une extension de syntaxe pour caml qui définit un nouveau mot clé `lwt`

```
lwt s = input_line ch in
lwt () = Lwt_unix.sleep 3. in
print_endline s;
Lwt.return ()
```

2.3.5 Exemples

Un thread qui écrit `hello` toutes les dix secondes

```
let rec f () =
  print_endline "hello";
  Lwt_unix.sleep 10. >>= f
in f ();
```

Lancer plusieurs threads en parallèle

Supposons que l'on ait deux fonctions `f : unit -> unit Lwt.t` et `g : unit -> unit Lwt.t`. On peut les lancer en parallèle avec :

1. `select` attend qu'il se passe quelque chose sur un des descripteurs de fichiers passés en paramètre (avec un timeout).
2. malheureusement les éditeurs de texte ne suivent pas souvent cette convention

```
ignore (f ());
ignore (g ());
```

Chacun des deux appels retourne immédiatement. On ignore les promesses renvoyées, mais si l'ordonnanceur est en marche, il pourra (s'il reprend la main) exécuter les promesses en attente crées par `f` et `g`.

Si l'on veut lancer les deux calculs en parallèles et ensuite attendre que les deux valeurs soient calculées :

```
(* Je lance les deux calculs en parallèle : *)
let first_thread = f () in
let second_thread = g () in
(* Maintenant j'attend la terminaison du premier : *)
first_thread >>= fun first_result ->
(* et du deuxième : *)
second_thread >>= fun second_result ->
...
```

Cooperative `List.map`

Voici un exemple tiré du module `Lwt_util`. Il définit deux fonctions `map` sur les listes. Un thread est lancé pour chaque valeur de la liste. Dans la première version, tous les threads sont lancés en parallèle. Dans la seconde ils sont séquentialisés (on attend la terminaison du premier pour lancer le deuxième).

```
let rec map f l =
  match l with
  | [] -> return []
  | v :: r ->
      let t = f v in
      let rt = map f r in
      t >>= fun v' ->
      rt >>= fun l' ->
      return (v' :: l')

let rec map_serial f l =
  match l with
  | [] -> return []
  | v :: r ->
      f v >>= fun v' ->
      map_serial f r >>= fun l' ->
      return (v' :: l')
```

Create a thread waiting for an event

Pour créer un thread qui attend un événement, nous allons utiliser la fonction `Lwt.wait ()`, qui crée :
- une promesse qui attend ;
- et un « wakener » que l'on peut utiliser avec la fonction `Lwt.wakeup` pour réveiller la promesse.

```
wait : unit -> 'a Lwt.t * 'a Lwt.u
wakeup : 'a Lwt.u -> 'a -> unit
```

Exemple :

```
(* Création de l'événement : *)
let waiter, wakener = wait () in
```

```
(* On attache une action à l'événement *)
ignore (waiter >>= Lwt_io.printl);
...

(* On déclenche l'événement : *)
wakeup wakener "HELLO";
(* Tous les threads en attente sur waiter sont réveillé,
   avec la valeur "HELLO". *)
```

2.3.6 Exceptions

Il faut être vigilant à la gestion des exceptions avec Lwt. En effet, la construction **`try ... with ...`** ne rattrapera pas les exceptions (puisqu'elles ont lieu de manière asynchrone). La seule façon de propager l'exception aux calculs en attente est de remplacer

```
try e
with ...
```

par

```
Lwt.catch
  (fun () -> ...)
  (function ... | exn -> fail exn)
```

Il faut aussi utiliser `Lwt.fail` au lieu de `raise`.

2.3.7 Utiliser des fonctions non coopératives

Comme l'ordonnanceur de Lwt n'est pas préemptif, l'utilisation d'une fonction non-coopérative peut bloquer tout le programme. Il faut donc faire très attention à ne jamais le faire.

Si jamais vous voulez utiliser une fonction non-coopérative réentrante (c'est-à-dire « thread-safe » pour les threads préemptifs), alors vous pouvez la faire exécuter dans un thread préemptif en faisant

```
Lwt_preemptive.detach f a
```

(où `f` est la fonction et `a` son paramètre). Exemple :

```
Lwt_preemptive.detach Unix.sleep 5 >>= fun () -> Lwt.return e
```

Si votre fonction n'est pas prévue non plus pour être utilisée avec des threads préemptifs, alors il faut la réécrire, ou bien lancer un autre processus.

Attention : Lwt n'est pas thread-safe. Ne pas l'utiliser à l'intérieur de plusieurs threads préemptifs en même temps.

2.3.8 Exemple

Here is an example of of program that use Lwt. It waits for a connection on a port given as first parameter. It then connect to another port (second parameter) and relay everything it receives in either side to the other side. It exits when either side closes the connection.

```
(* Usage: relay <listening_port> <dest_port> *)

(* This program waits for a connection on <listening_port>. It then
   connect to <dest_port> and relay everything it receives in either
   side to the other side.  It exits when either side closes the
   connection. *)

let listening_port = int_of_string Sys.argv.(1)
```

```
let dest_port = int_of_string Sys.argv.(2)

open Lwt

let rec really_write out_ch buffer pos len =
  Lwt_unix.write out_ch buffer pos len >>= fun len' ->
  if len = len' then return () else
  really_write out_ch buffer (pos + len') (len - len')

let relay in_ch out_ch =
  let rec relay_rec previous_write =
    let buffer = String.create 8192 in
    (* Read some data from the input socket *)
    Lwt_unix.read in_ch buffer 0 8192 >>= fun len ->
    (* If we read nothing, this means that the connection has been
       closed.  In this case, we stop relaying. *)
    if len = 0 then return () else begin
      (* Otherwise, we write the data to the ouput socket *)
      let write =
        (* First wait for the previous write to terminate *)
        previous_write >>= (fun () ->
        (* Then write the contents of the buffer *)
        really_write out_ch buffer 0 len)
      in
      relay_rec write
    end
  in
  relay_rec (return ())

let new_socket () = Lwt_unix.socket Unix.PF_INET Unix.SOCK_STREAM 0
let local_addr num = Unix.ADDR_INET (Unix.inet_addr_any, num)

let _ =
  Lwt_unix.run begin
    (* Initialize the listening address *)
    let listening_socket = new_socket () in
    Lwt_unix.setsockopt listening_socket Unix.SO_REUSEADDR true;
    Lwt_unix.bind listening_socket (local_addr listening_port);
    Lwt_unix.listen listening_socket 1024;
    (* Wait for a connection *)
    Lwt_unix.accept listening_socket >>= fun (inp, _) ->
    (* Connect to the destination port *)
    let out = new_socket () in
    Lwt_unix.connect out (local_addr dest_port) >>= fun () ->
    (* Start relaying *)
    Lwt.choose [relay inp out; relay out inp]
  end
```

Chapitre 3

Usages évolués du système de types

Ce chapitre est en version préliminaire. Tout commentaire apprécié.

Le but de ce chapitre est de :
- Montrer (ou revoir) quelques aspects avancés du système de type de OCaml ;
- Comprendre comment utiliser au maximum ce système de types pour faire en sorte de vérifier le maximum de propriétés sur le programme *statiquement* et éviter ainsi les possibilités de bugs ;
- Comprendre que le système de type diminue l'expressivité du langage et donc la liberté du programmeur. La plupart du temps c'est pour son bien, mais parfois les restrictions sont trop grandes et le compilateur rejette des programmes corrects. Il faut donc travailler à améliorer le système de types pour le rendre plus riche et donc moins contraignant, mais en évitant de le rendre trop complexe.

3.1 Variants polymorphes

3.1.1 Syntaxe

Vous connaissez déjà les types sommes (appelés aussi variants), par exemple :

```
# type t = A | B of int;;
type t = A | B of int
# B 4;;
- : t = B 4
```

Ici le constructeur `B` ne peut pas être utilisé si le type `t` n'est pas précédemment défini.

Les variants polymorphes sont d'autres constructeurs qui peuvent être utilisés sans avoir été déclarés dans un type. Pour les reconnaître, on les fait précéder d'une apostrophe à l'envers (`` ` ``)[1].

Cela permet à un même constructeur d'appartenir à plusieurs types et même d'être utilisé avec plusieurs types d'arguments (d'où le nom).

```
# `A;;
- : [> `A ] = `A
# `B 4;;
- : [> `B of int ] = `B 4
# [`A  ; (`B 4)];;
- : [> `A | `B of int ] list = [`A; `B 4]
```

1. Malheureusement dans certaines polices de caractères, la différence n'est pas évidente... Ceci est une apostrophe : ' et ceci une anti-apostrophe : `

Remarquez comment sont indiqués les types de ces valeurs.
La barre verticale signifie « ou ».
Le « > » signifie : « un type qui contient *au moins* les constructeurs suivants ».

Par exemple ici, on a une liste de valeurs qui appartiennent à un type qui contient au moins le constructeur `` `A `` sans argument et le constructeur `` `B `` avec un argument entier.

Les types « variants polymorphes » peuvent aussi parfois contenir un « < » qui signifie : « un type qui contient *au plus* les constructeurs suivants ». Exemple :

```
# let f = function
  | `Bip a -> 1
  | `Toto -> 2;;
val f : [< `Bip of 'a | `Toto ] -> int = <fun>
```

Plus compliqué :

```
# let g x = match x with `Bip _ -> 4 | _ -> f x;;
val g : [< `Bip of 'a | `Toto > `Bip ] -> int = <fun>
```

Attention : il peut être tentant d'utiliser toujours des variants polymorphes à la place des types sommes habituels. Je ne vous le conseille pas pour les raisons suivantes :
- ils induisent des problèmes de typage parfois complexes
- ils induisent une discipline de type moins forte (moins de vérifications sont faites statiquement, notamment il sera facile d'utiliser un constructeur qui n'existe pas, ou avec un mauvais type).
- ils induisent une petite perte d'efficacité des programmes.

N'utilisez les variants polymorphes que si vous avez vraiment besoin de leurs spécificités.

3.1.2 Contraintes sur les types sur les arguments des constructeurs

```
# let f1 = fun (`Nombre x) -> x = 0;;
val f1 : [< `Nombre of int ] -> bool = <fun>
# let f2 = fun (`Nombre x) -> x = 0.0;;
val f2 : [< `Nombre of float ] -> bool = <fun>
# let f x = f1 x || f2 x;;
val f : [< `Nombre of float & int ] -> bool = <fun>
```

Ici le type de l'argument du constructeur `` `Nombre `` doit être à la fois un `int` et un `float`, ce qui est indiqué avec la syntaxe `&` dans le type. Dans cet exemple, la contrainte est impossible à satisfaire. On ne pourra pas appliquer `f`.

3.1.3 Sous-typage

```
# type 'a vlist = [`Nil | `Cons of 'a * 'a vlist];;
type 'a vlist = [ `Cons of 'a * 'a vlist | `Nil ]
# type 'a wlist = [`Nil | `Cons of 'a * 'a wlist | `Snoc of 'a wlist * 'a];;
type 'a wlist = [ `Cons of 'a * 'a wlist | `Nil | `Snoc of 'a wlist * 'a ]
```

Le type `'a vlist` est inclus dans le type `'a wlist`. On dit que c'est un *sous-type* de `'a wlist`.

Pour convertir une `'a vlist` en `'a wlist`, il faut faire une *coercion* explicite. En OCaml, la coercion d'une valeur `v` d'un type `t1` vers un type `t2` se note `(v : t1 :> t2)`. (On peut parfois omettre le `: t1`). Exemple :

```
# let wlist_of_vlist  l = (l : 'a vlist :> 'a wlist);;
val wlist_of_vlist : 'a vlist -> 'a wlist = <fun>
# let a : int vlist = `Cons (1, `Nil);;
val a : int vlist = `Cons (1, `Nil)
# wlist_of_vlist a;;
- : int wlist = `Cons (1, `Nil)
```

On peut aussi *ouvrir* un type :

```
# let open_vlist l = (l : 'a vlist :> [> 'a vlist]);;
val open_vlist : 'a vlist -> [> 'a vlist ] = <fun>
# open_vlist a;;
- : [> int vlist ] = `Cons (1, `Nil)
```

3.1.4 Définition de types variants polymorphes

Attention il n'est pas possible de donner un nom à un type ouvert (puisqu'un type ouvert représente un ensemble de type).

```
# type t = [ `A | `B of int];;
type t = [ `A | `B of int ]
# type t = [> `A | `B of int];;
Characters 9-27:
  type t = [> `A | `B of int];;
           ^^^^^^^^^^^^^^^^^^
Unbound type parameter ..
```

On voit que le type ouvert est représenté en interne avec une variable de type spéciale (notée `..`). Cette variable de type est appelée *variable de rangée* [2].

3.1.5 Variants polymorphes et types abstraits

Lorsque le type variant polymorphe est utilisé en paramètre d'un type abstrait, il n'y a pas moyen a priori de savoir s'il va être utilisé à gauche ou à droite d'une flèche, et donc s'il faut mettre un « < » ou un « > »... Dans ce cas, *OCaml* permet de préciser dans l'interface la *variance* des paramètres de types, c'est-à-dire si le paramètre de type est utilisé à droite (type covariant, ou positif) d'une flèche ou à gauche (type contravariant, ou négatif).

Exemple d'interface précisant la variance des paramètres de types abstraits :

```
type +'a t1

type ('a, -'b) t2
```

3.2 Types fantômes

Un type fantôme est une annotation de type ajoutée à l'aide d'une contrainte de type (en général comme paramètre de type) pour indiquer certaines propriétés de nos données. On les appelle *fantômes* parce qu'aucune valeur de ces types n'est jamais crée. Ils sont utilisés juste au moment de la compilation pour rajouter des contraintes de typage.

2. Il est en revanche possible de définir un type paramétré avec la syntaxe `type 'a t = 'a constraint 'a = [> `A];;` (on rajoute une contrainte sur le type manuellement). On peut aussi utiliser des *types de rangées privés* mais c'est une autre histoire.

3.2.1 Listes vides ou non vides

Exemple :

```
# module Liste = (struct
    type vide
    type nonvide
    type ('a, 'b) liste = 'b list
    let listevide : (vide, 'b) liste = []
    let cons (v : 'b) (l : ('a, 'b) liste) : (nonvide, 'b) liste = v::l
    let head (l : (nonvide, 'b) liste) = match l with
      | [] -> assert false
      | a::_ -> a
  end : sig
    type vide
    type nonvide
    type ('a, 'b) liste
    val listevide : (vide, 'b) liste
    val cons : 'b -> ('a, 'b) liste -> (nonvide, 'b) liste
    val head : (nonvide, 'b) liste -> 'b
  end);;
# open Liste;;
# listevide;;
- : (Liste.vide, 'a) Liste.liste = <abstr>
# cons 3 listevide;;
- : (Liste.nonvide, int) Liste.liste = <abstr>
# head (cons 3 listevide);;
- : int = 3
# head listevide;;
Characters 5-14:
  head listevide;;
       ^^^^^^^^^
Error: This expression has type (Liste.vide, 'a) Liste.liste
       but an expression was expected of type (Liste.nonvide, 'b) Liste.liste
```

3.2.2 Fantômes variants polymorphes

Dans l'exemple précédent, les paramètres fantômes étaient des types vides. Utiliser des variants polymorphes comme paramètres fantômes permet de profiter du sous-typage des variants polymorphes dans les contraintes de typage induites par les types fantômes. On aurait pu écrire :

```
# module Liste = (struct
    type ('a, 'b) liste = 'b list
    let listevide : ([ `Vide ], 'b) liste = []
    let cons (v : 'b) (l : ([< `Vide | `Nonvide ], 'b) liste)
      : ([ `Nonvide ], 'b) liste = v::l
    let head (l : ([ `Nonvide ], 'b) liste) = match l with
      | [] -> assert false
      | a::_ -> a
  end : sig
    type ('a, 'b) liste
    val listevide : ([ `Vide ], 'b) liste
    val cons : 'b -> ([< `Vide | `Nonvide ], 'b) liste -> ([ `Nonvide ], 'b) liste
    val head : ([ `Nonvide ], 'b) liste -> 'b
  end);;
```

```
# open Liste;;
# listevide;;
- : ([ `Vide ], 'a) Liste.liste = <abstr>
# cons 3 listevide;;
- : ([ `Nonvide ], int) Liste.liste = <abstr>
# head (cons 3 listevide);;
- : int = 3
# head listevide;;
Characters 5-14:
  head listevide;;
       ^^^^^^^^^
Error: This expression has type ([ `Vide ], 'a) Liste.liste
       but an expression was expected of type ([ `Nonvide ], 'b) Liste.liste
       These two variant types have no intersection
```

On peut même coder (en unaire) la longueur de la liste dans le type :

```
module Liste = (struct
  type ('a, 'b) liste = 'b list
  let listevide : ([ `Zero ], 'b) liste = []
  let cons (v : 'b) (l : ('a, 'b) liste) : ([ `Succ of 'a ], 'b) liste = v::l
  let head (l : ([ `Succ of 'a ], 'b) liste) : 'b = match l with
    | [] -> assert false
    | a::_ -> a
  let tail (l : ([ `Succ of 'a ], 'b) liste) : ('a, 'b) liste = match l with
    | [] -> assert false
    | _::l -> l
end : sig
  type ('a, 'b) liste
  val listevide : ([ `Zero ], 'b) liste
  val cons : 'b -> ('a, 'b) liste -> ([`Succ of 'a], 'b) liste
  val head : ([`Succ of 'a], 'b) liste -> 'b
  val tail : ([`Succ of 'a], 'b) liste -> ('a, 'b) liste
end)

# let l = cons 1 (cons 2 (cons 3 (cons 4 listevide)));;
val l :
  ([ `Succ of [ `Succ of [ `Succ of [ `Succ of [ `Zero ] ] ] ] ], int)
  Liste.liste = <abstr>
# tail l ;;
- : ([ `Succ of [ `Succ of [ `Succ of [ `Zero ] ] ] ], int) Liste.liste =
<abstr>
# tail listevide;;
Characters 5-14:
  tail listevide;;
       ^^^^^^^^^
Error: This expression has type ([ `Zero ], 'a) Liste.liste
       but an expression was expected of type
         ([ `Succ of 'b ], 'c) Liste.liste
       These two variant types have no intersection
```

On utilise aussi souvent les types objets comme fantômes.

3.2.3 Types algébriques généralisés

En fait, dans le cas où l'on se contente de l'information vide ou non-vide, on peut remarquer que le fantôme dépend juste du constructeur. On aimerait écrire :

```
type ('a, 'b) list =
 | Nil with 'a = vide
 | Cons of ('b * ('c, 'b) list) with 'a = nonvide
```

Ou bien, pour les types qui codent la longueur :

```
type ('a, 'b) list =
 | Nil with 'a = [ `Zero ]
 | Cons of ('b * ('c, 'b) list) with 'a = [ `Succ of 'c ]
```

Avec cette extension on pourrait même imaginer que le compilateur
- Autorise à ne pas mettre les cas inutiles (`assert false`)
- Autorise un pattern matching dont le type de retour dépend du fantôme (et donc peut être différent suivant les cas !)

On appelle ces types des *types algébriques généralisés* (GADT). Ils ne sont pas implémentés dans OCaml pour l'instant, mais ils existent en Haskell.

3.3 Types universels et existentiels

3.3.1 Quantification universelle en OCaml

Les quantificateurs universels sur les variables de type sont toujours prénexes sauf pour les champs d'enregistrement et les méthodes. On ne les note que quand elles ne sont pas prénexes.

```
# let f x = x;;
val f : 'a -> 'a = <fun>
```

signifie en fait `'a. 'a -> 'a` (qui se lit $\forall\alpha.\alpha \to \alpha$).

```
# type r = {f : 'a -> 'a};;
Characters 14-16:
  type r = {f : 'a -> 'a};;
               ^^
Error: Unbound type parameter 'a
# type 'a r = {f : 'a -> 'a};;
type 'a r = { f : 'a -> 'a; }
# type r = {f : 'a. 'a -> 'a};;
type r = { f : 'a. 'a -> 'a; }
```

Lorsque nous avons besoin d'un quantificateur universel ailleurs que dans un champ de record ou une méthode, il suffit de créer un record à un seul champ.

3.3.2 Existentiels

Parfois on aimerait aussi avoir une quantification existentielle sur les variables de type. Imaginez que vous voulez stocker dans une tables des données hétérogènes et la fonction pour les manipuler. Par exemple les couples `(3, print_int)` et `(3.14, print_float)`. On ne peut pas mettre ces couples dans une même liste, puisqu'ils n'ont pas le même type. On aimerait faire une liste de type $(\exists\alpha.(\alpha \times (\alpha \to$ `unit`$)))$ `list`. Lors de la déstructuration d'une telle liste, le premier élément d'un couple ne pourrait être utilisé qu'avec la fonction donnée en deuxième élément.

En général, on peut souvent s'en sortir en stockant dans la liste l'application de la fonction sur l'élément :

```
[print_int 3; print_float 3.14]
```

éventuellement retardée en la mettant à l'intérieur d'une autre fonction si vous ne voulez pas que le calcul et les effets de bord aient lieu tout de suite :

```
[(fun () -> print_int 3); (fun () -> print_float 3.14)]
```

Des extensions du système de type d'OCaml sont à l'étude pour permettre de manipuler des types existentiels. En attendant, il existe une astuce (un peu lourde). En effet, $\exists\beta.\tau$ est équivalent à $\forall\alpha.(\forall\beta.(\tau \rightarrow \alpha)) \rightarrow \alpha$. Autrement dit : « Si tu me donnes une fonction capable de transformer τ en α quel que soit β, alors je te donne un α ».

Dans notre exemple, on définit le type $\forall\beta.(\tau \rightarrow \alpha)$, qui servira a stocker les fonctions capables d'utiliser (transformer) notre type existentiel :

```
type 'a user = { use : 'b. ('b * ('b -> unit)) -> 'a }
```

Par exemple, la fonction qui nous permettra d'appliquer le 2e élément du couple au premier sera définie par :

```
let applique = { use = fun (x, f) -> f x }
```

Notre type existentiel sera défini comme une fonction qui applique un `user` :

```
type t = { f : 'a. ('a user -> 'a)}
```

Le constructeur pour ce type sera :

```
let pack v = { f = fun user -> user.use v}
```

La fonction suivante permet d'appliquer un `user` :

```
let use f v = v.f f
```

Exemples d'utilisation :

```
# pack (1, print_int);;
- : t = {f = <fun>}
# let l = [pack (1, print_int); pack (3.14, print_float)];;
val l : t list = [{f = <fun>}; {f = <fun>}]
# let a = List.hd l;;
val a : t = {f = <fun>}
# use applique a;;
1- : unit = ()
```

En fait on peut faire un foncteur qui crée les fonctions `pack` et `use` pour un type donné :

```
module Make(X : sig type 'a t end) : sig
  type t
  val pack : 'a X.t -> t
  type 'a user = { use : 'b. 'b X.t -> 'a }
  val use : 'a user -> t -> 'a
end = struct
  type 'a user = { use : 'b. 'b X.t -> 'a }
  type t = { f : 'a. 'a user -> 'a }
  let pack impl = { f = fun user -> user.use impl }
  let use f p = p.f f
end
```

3.4 Fonctions à type variable

On souhaite parfois faire des fonctions qui peuvent être appelées sur des valeurs de types différents, ou bien dont le type (et la valeur) de retour dépendent d'une valeur en entrée. On peut par exemple avoir envie d'écrire une fonction d'affichage qui fonctionne pour plusieurs types.

Ocaml ne dispose pas de surcharge. Le type de chaque fonction est unique. On peut cependant faire une fonction dont le type de retour dépend du type du paramètre. Dans ce cas il faut réussir à faire entrer dans le type de la valeur d'entrée suffisamment d'information pour que le typeur soit capable de connaître le type de retour.

3.4.1 Type `format` en OCaml

Prenons l'exemple de la fonction `Printf.printf` de OCaml :

```
# Printf.printf "%s %d\n" "aa" 45;;
aa 45
- : unit = ()
# Printf.printf "%s %d\n";;
- : string -> int -> unit = <fun>
# Printf.printf "%s";;
- : string -> unit = <fun>
```

Pour réussir à faire cela de manière bien typée, il ne faut pas que le premier paramètre de `printf` soit une chaîne de caractères. L'implémentation d'OCaml utilise juste un *hack* pour permettre d'utiliser la syntaxe des chaînes pour les formats :

```
# Printf.printf;;
- : ('a, out_channel, unit) format -> 'a = <fun>
# let s = "%s";;
val s : string = "%s"
# Printf.printf s;;
Characters 14-15:
  Printf.printf s;;
                ^
Error: This expression has type string but an expression was expected of type
         ('a, out_channel, unit) format
# format_of_string "%s";;
- : (string -> '_a, '_b, '_c, '_d, '_d, '_a) format6 = <abstr>
```

Nous allons voir quelques solutions pour implémenter des fonctions à la `printf` (c'est-à-dire des fonctions pouvant être exécutées sur des types différents), en passant ou non ce type en paramètre.

3.4.2 Utilisation de surcharge ou de types dynamiques

En Java il n'est pas rare de tester dynamiquement la classe d'un objet. Cela suppose d'avoir encore l'information de type au moment de l'exécution, ce que l'on appelle des *types dynamiques*[3]. En fait, si le système de type est suffisamment expressif, on peut se passer de types dynamiques. OCaml a fait le choix de ne pas avoir du tout d'information de type à l'exécution, ce qui permet de gagner de la place en mémoire et du temps.

Dans un langage disposant de types dynamiques, on pourrait implémenter une fonction d'affichage en testant le type de la valeur au moment de l'exécution et appeler les fonctions d'affichage en fonction de ce type.

Dans un langage disposant de surcharge, on pourrait donner le même nom aux fonction d'affichage des entiers et des paires, etc. Dans un langage où la résolution de la surcharge est statique, cela revient exactement à appeler des fonctions `print_int`, `print_pair`, etc. Dans un langage où la résolution de la surcharge doit être faite dynamiquement (par exemple à cause du polymorphisme), on utilisera des types dynamiques.

3. Ne pas confondre avec le *typage dynamique*, utilisé par les langages de script, qui sous-entend une « vérification » dynamique des types, mais n'aide pas à la fiabilité du programme.

Dans un langage tout objet, on peut avoir une méthode `print` dans chaque objet, et utiliser le mécanisme de liaison tardive pour déterminer le code à exécuter.

En Haskell, on pourra utiliser les *classes de types*.

Quand on n'a ni surcharge, ni types dynamique, ou quand le « format » n'est pas le type, on doit passer ce format en paramètre à la fonction. Les sections suivantes détaillent deux solutions pour faire cela.

3.4.3 *Functional unparsing*

Technique due à Olivier Danvy.

Le *functional unparsing* consiste à voir le format comme une *combinaison* de fonctions d'affichage élémentaires.

```
# let int x = print_int x;;
val int : int -> unit = <fun>
# let printf format v = format v;;
val printf : ('a -> 'b) -> 'a -> 'b = <fun>
# printf int 3;;
3- : unit = ()
# let pair (f1, f2) (x, y) = f1 x; f2 y;;
val pair : ('a -> 'b) * ('c -> 'd) -> 'a * 'c -> 'd = <fun>
# pair (int, int);;
- : int * int -> unit = <fun>
# printf (pair (int, int)) (4,5);;
45- : unit = ()
# printf (pair ((pair (int, int)), int)) ((4, 3),5);;
435- : unit = ()
# let ( ** ) = pair
 val ( ** ) : ('a -> 'b) * ('c -> 'd) -> 'a * 'c -> 'd = <fun>
# printf ((int ** int) ** int) ((4, 3),5);;
435- : unit = ()
```

Autre version, qui permet de définir des fonctions à nombre variable d'arguments :

```
type ('a, 'b) t = 'a -> 'b
let int : ('a, int -> 'a) t =
  fun k x -> (print_int x; k)
let string : ('a, string -> 'a) t =
  fun k x -> (print_string x; k)
let ( ** ) (f1 : ('b,'c) t) (f2 : ('a, 'b) t) : ('a, 'c) t =
    fun k -> f1 (f2 k)
let printf (format : (unit, 'a) t) : 'a =
  format ()

# printf int 5;;
5- : unit = ()
# printf (int ** string ** int) 5 "hello" 3;;
5hello3- : unit = ()
# printf (int ** string ** int ** int) 5 "hello";;
5hello- : int -> int -> unit = <fun>
```

3.4.4 Types algébriques généralisés

Les types algébriques généralisés permettraient de proposer une solution plus simple (mais moins extensible) à ce problème. Il suffirait de définir :

```
type 'a ty =
  | Int with 'a = int
  | Pair of ('b ty * 'c ty) with 'a = 'b * 'c

# let printf format v = match format with
  | Int -> print_int v (* ici v est un int *)
  | Pair (b, c) -> printf b (fst v); printf c (snd v) (* ici v est une paire *)
val printf : 'a ty -> 'a -> unit = <fun>
```

3.5 Conclusion

Vocabulaire à connaître :

Inférence de type

Sous-typage On distingue le sous-typage *structurel* basé sur la structure des types (variants polymorphes et objets en OCaml) du sous-typage *nominal*, basé sur les noms de types.

Types existentiels

Types algébriques généralisés (GADT)

Types dynamiques

Typage dynamique

D'autres types intéressants dont nous n'avons pas parlé sont les types *expressions régulières*, qui permettent de définir un type par une expression régulière et de faire du pattern matching avec des motifs qui ont la forme d'expressions régulières. Ces types sont particulièrement intéressants pour manipuler des documents XML et les transfomer en s'assurant de leur conformité à un standard (DTD). Les langages CDuce, XDuce ont été créés sur ce principe. Le langage OcamlDuce est une extension d'OCaml qui lui rajoute des types XML.

On peut être tenté de rajouter beaucoup d'expressivité au système de types pour exprimer toujours plus de propriétés des programmes dans les types. Mais cela se fait souvent au détriment de la simplicité. Cela peut par exemple entraîner la perte de l'inférence de types. Coq utilise un système de types très riche, qui permet d'exprimer n'importe quelle propriété mathématique des programmes (par exemple « entiers pairs », « fonction qui à un entier pair associe l'entier tel que ... »). L'avantage est que vous êtes sûrs que le programme fait exactement ce qu'il faut. L'inconvénient est que vous devez prouver vous-même la propriété (avec l'aide de Coq).

Chapitre 4

λ-calcul et ordre d'évaluation

4.1 λ-calcul

Quelques rappels du cours de sémantique

Les termes

$t ::= x \mid t\ t \mid$ `fun` x `->` t

où x est une *variable*,
$t\ t$ est une *application* d'un terme à un autre,
et `fun` x `->` t est appelé une *abstraction* (fonction).

En λ-calcul, `fun` x `->` t est en général noté $\lambda x.\ t$ (d'où le nom). On appelle cela aussi un *lieur* (car il réalise une *liaison* entre une valeur et un nom de variable). Dans le terme `fun` x `->` t, toutes les occurrences de x apparaissant dans t sont dites *liées*. Par opposition, les variables qui ne sont pas *sous un lambda* sont dites *libres*.

La β-réduction

Évaluer un programme du λ-calcul consiste à appliquer des règles de β-réduction :

$$\beta \qquad (\texttt{fun}\ x\ \texttt{->}\ t)\ u \to t[u/x]$$

ce qui se lit : « l'application de l'application du terme `fun` x `->` t au terme u se réduit en t dans lequel toutes les occurrences libres de x ont été substituées par u ». Un terme de la forme (`fun` x `->` t) u s'appelle un *redex*.

Exemple. β-réduire les termes ci-dessous :

```
(fun x -> x) (fun y -> y)
```

```
(fun y -> (fun t -> t y) y) (t x)
```

En fait le nom des variables n'est pas significatif. On peut remplacer un nom de variable par un autre, à condition qu'il n'y ait pas de *capture*. L'exemple ci-dessus pourrait s'écrire aussi

```
(fun y -> (fun u -> u y) y) (t x)
```

On dit que ces deux termes sont α-équivalents. On considérera toujours les termes modulo α-équivalence (ce qui correspond à ce que font les langages de programmation).

Le terme `fun x -> x x` est souvent appelé Δ . Essayez de réduire le terme $\Delta\ \Delta$:

```
(fun x -> x x) (fun x -> x x)
```

On notera $\to^*$ la fermeture réflexive transitive de la β-réduction, c'est-à-dire l'application successive de zéro, une ou plusieurs règles de β-réduction.

λ-calcul et OCaml

Comme en Ocaml, on s'autorisera d'écrire `fun x y -> t` comme sucre syntaxique pour `fun x -> fun y -> t`.
Comme en OCaml on s'autorisera d'écrire `let x = t1 in t2` comme sucre syntaxique pour `(fun x -> t2) t1`.

Extension du λ-calcul

On peut montrer que le λ-calcul a la même puissance calculatoire que les machines de Turing ou les langages informatiques usuels (C, Pascal, OCaml, Java...). C'est-à-dire que l'on peut calculer exactement les mêmes fonctions (d'entiers dans les entiers). Pour montrer cela, il faut coder les entiers par des termes du λ-calcul (par exemple les *entiers de Church*) ainsi que les opérations usuelles (multiplication, addition, etc.).

Pour simplifier, nous allons étendre le λ-calcul avec des entiers (i), des paires, des projections, des sommes binaires et des injections :

$$\begin{array}{rcl} t & ::= & i \mid x \mid t\ t \mid \texttt{fun}\ x\ \texttt{->}\ t \\ & & \mid (t,\ t) \mid \texttt{fst}\ t \mid \texttt{snd}\ t \\ & & \mid \texttt{Left}\ t \mid \texttt{Right}\ t \mid \texttt{match}\ t\ \texttt{with Left}\ x\ \texttt{->}\ t\ \texttt{|\ Right}\ x\ \texttt{->}\ t \end{array}$$

Il faut ajouter les règles de β-réduction correspondantes :

$$\begin{array}{lrcl} \beta_{\times 1} & \texttt{fst}\ (t,\ u) & \to & t \\ \beta_{\times 2} & \texttt{snd}\ (t,\ u) & \to & u \\ \beta_{+1} & \texttt{match Left}\ t\ \texttt{with Left}\ x\ \texttt{->}\ u\ \texttt{|\ Right}\ y\ \texttt{->}\ v & \to & u[t/x] \\ \beta_{+2} & \texttt{match Right}\ t\ \texttt{with Left}\ x\ \texttt{->}\ u\ \texttt{|\ Right}\ y\ \texttt{->}\ v & \to & v[t/y] \end{array}$$

On pourrait rajouter aussi les opérations sur les entiers et les règles associées.

On appellera *λ-calcul pur* le λ-calcul non étendu.

λ-calcul typé

Le λ-calcul tel qu'on l'a vu pour l'instant n'est pas typé. C'est-à-dire que l'on peut écrire n'importe quel terme avec le langage ci-dessus. Si une règle de β-réduction s'applique, alors le terme peut se réduire, sinon on est bloqué.

Si l'on rajoute un système de types au λ-calcul, on va limiter le nombre de termes possibles. Exemples de termes mal typés (pour Ocaml) :

```
fun x -> x x
match (fun x -> x) with Left x -> x | Right x -> x
fst (fun x -> x)
```

4.2 Ordre d'évaluation

Forme normale

Theorem 1. *La β-réduction du λ-calcul pur est* confluente *: Soient t, u_1 et u_2 des λ-termes tels que $t \to^* u_1$ et $t \to^* u_2$. Alors il existe un λ-terme v tel que $u_1 \to^* v$ et $u_2 \to^* v$.*

Theorem 2. *Le λ-calcul pur*[1] typé *est* fortement normalisant*, c'est-à-dire que toutes les réductions d'un terme t terminent sur un terme unique (pour chaque t), que l'on appelle la* forme normale *de t (et qui ne contient plus de redex).*

Opérateur de point fixe

La propriété de forte normalisation empêche d'écrire des programmes qui bouclent en λ-calcul typé. Pour réintroduire cette possibilité, nous ajouterons au λ-calcul une construction **`let rec`** ou bien un terme spécial `fix` dont la règle de réduction est :

$$\beta_{\texttt{fix}} \qquad \texttt{fix}\, t \quad \to \quad t\, (\texttt{fix t})$$

En λ-calcul non type, on peut définir des termes qui ont cette propriété, par exemple le combinateur de point fixe de Curry $Y =$ `fun f -> (fun x -> f (x x)) (fun x -> f (x x))`.

Stratégies de réduction

Un terme peut contenir plusieurs redex, donc il y a plusieurs façon de réduire un même terme. Le choix de l'ordre d'évaluation est important. Par exemple en λ-calcul pur, la réduction du terme `(fun x -> fun y -> x) 1` $(\Delta\ \Delta)$ se réduit en `1` si l'on commence par réduire les deux premiers redex, mais elle peut boucler si l'on décide de commencer à réduire $\Delta\ \Delta$. En λ-calcul typé avec opérateur de point fixe on a le même problème.

Mais même sans boucle infinie, l'ordre de réduction peut avoir un impact. Prenez par exemple le terme

```
(fun y -> (y, y)) ((fun x -> x) 1)
```

Si l'on commence par réduire le redex **`fun`** `y -> ...`, alors l'argument sera dupliqué et le calcul de `((fun x -> x) 1)` aura lieu deux fois (perte d'efficacité).

Enfin l'ordre d'évaluation a une importance cruciale dans un langage avec effets de bord (par exemple des modifications de références ou des affichages). Exemple : que vaut l'expression OCaml suivante ?

```
let a = ref 0 in
let f x y = () in
f (a:=1) (a:=2);
!a
```

Suivant la stratégie d'évaluation le terme précédent peut se réduire sur 2, sur 1, ou même sur 0.

Nous allons maintenant essayer de classifier les stratégies de réduction.

Stratégies de réduction faible

Observez les exemples suivants (en OCaml) :

```
# (fun x -> print_endline "salut"; x);;
- : 'a -> 'a = <fun>
# (fun x -> print_endline "salut"; x) 4;;
salut
- : int = 4
```

On voit que le redex `print_endline "salut"` (qui produit un effet de bord) n'est réduit que dans le 2e cas.

Sauf dans des cas très particuliers, les langages de programmation *ne réduisent pas sous les λ* (autrement ne réduisent le corps des fonctions que lorsqu'elles sont appliquées à leurs arguments). C'est ce que l'on appelle la *réduction faible* (par opposition à réduction forte, qui cherche à obtenir la forme normale,

1. La propriété s'étend bien aux produits mais c'est plus compliqué avec les sommes.

si elle existe).

La réduction faible, si elle termine, transforme un terme quelconque en un terme où tous les redex sont dans le corps de fonctions (« sous des λ »). On appelle ces termes des *valeurs*.

En λ-calcul typé avec les extensions définies plus haut, on peut définir l'ensemble des valeurs par :

$$\begin{array}{rcl} v & ::= & i \mid x \mid \texttt{fun}\ x\ \texttt{->}\ t \\ & & \mid (v,\ v) \\ & & \mid \texttt{Left}\ v \mid \texttt{Right}\ v \end{array}$$

Notez le t dans l'abstraction (terme quelconque, pas seulement une valeur).

Stratégies internes et externes

En présence d'un redex $(\texttt{fun}\ x\ \texttt{->}\ t)\ u$, il y a deux stratégies de réduction faible possibles :

- Soit l'on commence par réduire le terme u, avant d'appliquer la règle de β-réduction au redex externe. C'est ce que l'on appelle une stratégie de *réduction interne*.
- Soit l'on commence par réduire le redex externe. C'est ce que l'on appelle une stratégie de *réduction externe*.

Appel par valeur *(call by value)*

La plupart des langages de programmation font de la *réduction interne faible*, que l'on appelle plus souvent *appel par valeur* (en anglais *call by value*, CBV) [2]. C'est le cas d'OCaml, C ou Java.

En appel par valeur, certains programmes peuvent boucler alors qu'une autre stratégie aurait permis de trouver une forme normale (par exemple `(fun x -> ()) loop`, où `loop` boucle indéfiniment). De même on peut faire des calculs inutiles.

Notez qu'il faut préciser tout de même l'ordre d'évaluation des termes lors d'une application. Est-ce que l'on évalue la fonction avant son paramètre ou le contraire ? OCaml ne spécifie pas ce comportement. Cela veut dire que deux compilateurs OCaml peuvent se comporter différemment, et les programmes ne doivent pas dépendre de cela. Il vaut mieux calculer d'abord les valeurs des paramètres (dans les `let ... in`) et ensuite faire l'appel. Le compilateur de l'INRIA évalue de droite à gauche (le paramètre avant la fonction).

Les langages qui implémentent l'appel par valeur sont dits *stricts*.

Appel par nom *(call by name)*

Une autre stratégie possible pour un langage de programmation est la *réduction externe faible*, appelée aussi *appel par nom*. En appel par nom, on ne fera jamais de calcul inutile. En revanche, certains calculs peuvent être dupliqués.

On peut aussi montrer qu'en appel par nom, la réduction trouvera toujours la forme normale si elle existe.

Appel par nécessité *(call by need)*

Une amélioration de l'appel par nom consiste à retarder l'évaluation des paramètres, mais sans faire de duplication. Les λ-termes sont alors représentés comme des graphes (au lieu d'arbres), puisque certains sous-arbres sont partagés. On appelle aussi cela l'*évaluation paresseuse* : l'évaluation n'est faite qu'en cas de besoin, et elle n'est faite qu'une seule fois. C'est le style d'évaluation utilisé par Haskell (et son ancêtre Miranda). On peut la simuler en OCaml avec des valeurs paresseuses.

On peut imaginer d'autre styles d'appel, par exemple évaluer les arguments en parallèle.

2. Exception : les conditionnelles (`if ... then ... else ...`) utilisent une stratégie de réduction externe.

4.3 Machines virtuelles

4.3.1 Évaluateurs symboliques, interpéteurs, compilateurs

On peut écrire un évaluateur symbolique qui implémente la β-réduction sur des termes, en faisant de la *substitution*. Il faut être particulièrement vigilant au problème de la capture.

Les interpréteurs ou compilateurs ne font pas de substitution. Au lieu de cela, ils maintiennent un *environnement* qui associe aux variables les termes auxquelles elles sont liées. En appel par valeur, cet environnement contient des valeurs (termes en forme normale faible).

Dans un interpréteur, l'environnement est représenté par une table d'association entre un nom de variable et une valeur. Dans un compilateur, les noms de variables sont remplacés par des adresses mémoire. L'environnement est donc directement la mémoire du système, indexée par les adresses.

4.3.2 Valeurs, fermetures

Plaçons-nous dans le cadre du λ-calcul étendu aux entiers (sans opérations sur les entiers), en nous interdisant les termes non clos. Cela correspond exactement au cas d'OCaml en se limitant aux fonctions, applications, variables et entiers.

Si on utilise un évaluateur symbolique, l'ensembles des valeurs est dans ce cas :

$$v \quad ::= \quad i \mid \texttt{fun}\ x\ \texttt{->}\ t$$

où t est un terme clos.

Mais si l'on n'utilise plus la réduction symbolique mais un interprète ou un compilateur (sans substitution), il faut se souvenir de l'environnement de chaque valeur (en fait seulement des abstractions). Cet environnement contient « les substitutions qui auraient dû être faites » si l'on avait utilisé un évaluateur symbolique.

Dans un langage non fonctionnel, ce problème ne se pose pas, parce que les valeurs ne peuvent jamais contenir des variables libres. Mais dans un langage fonctionnel, une valeur peut être une fonction avec des variables libres. Il faut donc préciser la valeur de ces variables au moment où la fonction a été définie, pour *clore* la fonction. Une valeur fonctionnelle sera donc représentée par son code (ou un pointeur vers son code), avec l'environnement au moment de sa création. On appelle ce type de valeur des *fermetures* (ou *clôtures*). On les notera $<\texttt{fun}\ x\ \texttt{->}\ t,\ \rho>$. L'ensemble des valeurs de notre interpréteur ou compilateur est donc :

$$v \quad ::= \quad i \mid\ <\texttt{fun}\ x\ \texttt{->}\ t,\ \rho>$$

Dans un interpréteur, on peut représenter un clôture par un triplet variable, terme, environnement. On peut restreindre l'environnement aux variables libres de la fonction.

Dans un compilateur, une fermeture sera représentée par un bloc mémoire contenant le pointeur vers le code de la fonction et les valeurs de toutes les variables libres.

Voici les types que nous allons utiliser pour écrire un interprète en OCaml :

```
type expr =
  | Int of int
  | Var of string
  | Fun of (string * expr)
  | App of (expr * expr)

type env = (string * value) list

and value =
  | VInt of int
  | VClosure of (string * expr * env)
```

4.3.3 Sémantique opérationnelle à grand pas

En appel par dvaleur, nous représenterons les environnements par des fonctions $\rho : X \to V$, où X est l'ensemble des variables et V l'ensemble des valeurs. Un interpréteur ou un compilateur évaluent des termes dans un environnement donné. Une variable x est évaluée en allant chercher sa valeur dans l'environnement. L'application d'une fonction à un terme est évalué en évaluant le corps de la fonction, après avoir ajouté la nouvelle liaison dans l'environnement. On résume cela par les règles suivantes :

$$\frac{}{\rho \vdash x \quad \Rightarrow \quad \rho(x)} \qquad \frac{}{\rho \vdash \texttt{fun}\ x\ \texttt{->}\ t \quad \Rightarrow \quad <\texttt{fun}\ x\ \texttt{->}\ t,\ \rho>}$$

$$\frac{\rho \vdash t_1 \quad \Rightarrow \quad <\texttt{fun}\ x\ \texttt{->}\ t'_1,\ \rho'> \qquad \rho \vdash t_2 \quad \Rightarrow \quad v_2 \qquad \rho' \oplus \{x \mapsto v_2\} \vdash t'_1 \quad \Rightarrow \quad v}{\rho \vdash t_1\ t_2 \quad \Rightarrow \quad v}$$

où l'opération $\oplus$ est définie par $(\rho \oplus \rho')(x) = \begin{array}{ll} \rho'(x) & \text{si } x \in dom(\rho') \\ \rho(x) & \text{sinon.} \end{array}$

Remarquez que cette opération n'est pas commutative. La nouvelle liaison masque une éventuelle précédente liaison de la même variable.

Notez que les règles données ci-dessus évaluent les paramètres de gauche à droite.

En appel par nom, nous représenterons les environnements par des fonctions $\rho : X \to T$, où T est l'ensemble des termes quelconques. Les règles deviennent :

$$\frac{}{\rho \vdash x \quad \Rightarrow \quad \rho(x)} \qquad \frac{}{\rho \vdash \texttt{fun}\ x\ \texttt{->}\ t \quad \Rightarrow \quad <\texttt{fun}\ x\ \texttt{->}\ t,\ \rho>}$$

$$\frac{\rho \vdash t_1 \quad \Rightarrow \quad <\texttt{fun}\ x\ \texttt{->}\ t'_1,\ \rho'> \qquad \rho' \oplus \{x \mapsto t_2\} \vdash t'_1 \quad \Rightarrow \quad t}{\rho \vdash t_1\ t_2 \quad \Rightarrow \quad t}$$

Exercice : compléter les règles pour les extensions du λ-calcul avec produits et sommes.

4.3.4 Interpréteur

Voici un interpréteur du λ-calcul en appel par valeur. Il suit bêtement les règles de la sémantique opérationnelle à grand pas.

```
exception Error

let interp =
  let rec aux env = function
    | Int i -> VInt i
    | Var x -> List.assoc x env
    | Fun (x, e) -> VClosure (x, e, env)
        (* on pourrait filtrer env pour ne laisser que les variables
           libres de la fonction *)
    | App (e1, e2) ->
        (* évaluation de gauche à droite *)
        let v1 = aux env e1 in
        let v2 = aux env e2 in
        (match v1 with
           | VClosure (x, e, env') -> aux ((x, v2)::env') e
           | _ -> raise Error)
  in aux []
```

Exercice : écrire un interprète en appel par nom.

4.3.5 Fonctions récursives

Pour interpréter les fonctions récursives, vous pouvez :
- soit utiliser des environnement récursifs : une fonction récursive `f` est associée à une clôture dont l'environnement est celui que vous êtes en train de définir :

```
| Letrec (f, x, e1, e2) ->
  let rec new_env = (f, VClosure (x, e1, new_env))::env in
  aux new_env e2
```

- soit définir des fermetures spéciales qui contiennent en plus le nom de la fonction. Lorsque vous appliquez ce type de fermeture, vous ajoutez la liaison de la variable dans l'environnement (comme d'habitude), mais aussi la liaison de la fonction :

```
(match v1 with
   | VClosure (x, e, env') -> aux ((x, v2)::env') e
   | VRecClosure (f, x, e, env') as recclos ->
       aux ((x, v2)::(f, recclos)::env') e
   | _ -> raise Error)
```

4.4 Flot de contrôle

4.4.1 Pile

On peut remarquer que dans l'interpréteur précédent, l'environnement est utilisé comme une pile : c'est liste et à chaque fois que l'on fait un appel de fonction, on empile (ajoute en tête de list) l'argument. En sortant de l'appel de fonction, on reprend l'environnement précédent (on dépile).

C'est exactement ce qu'il se passe également avec un compilateur.

Les langages de programmation n'ont pas toujours utilisé des piles. L'utilisation d'une pile n'est possible en fait que parce que les appels de fonction sont correctement imbriqués : si f appelle g, alors on sort de g avant de sortir de f.

En particulier, les langages avec `GOTO` cassent cette discipline. Ils permettent de se rendre à n'importe quel point du programme. Si l'utilisation de `GOTO` à l'intérieur d'une même fonction est envisageable, sauter d'une fonction à l'autre est très hasardeux, parce que cela permet de se retrouver dans un mauvais environnement.

Nous allons voir dans cette section des façons de modifier le déroulement normal du programme (casser le *flot de contrôle*) de manière sûre, c'est-à-dire de manière compatible avec la discipline de pile. Pour cela, il faut non seulement changer le point d'exécution du programme (comme `GOTO`), mais il faut aussi se placer dans un environnement cohérent (c'est-à-dire une pile qui correspond bien au nouveau point d'exécution).

Remarquez qu'il est très difficile de construire depuis zéro une nouvelle pile cohérente avec un point du programme donné. Les solutions suivantes permetteront donc seulement de se replacer dans des contextes déjà vus au cours du programme.

4.4.2 Exceptions

Les exceptions sont une première façon de sortir du flot normal de contrôle. En fait c'est une version sophistiquée de la commande `exit` qui permet de sortir du programme avant sa terminaison (c'est-à-dire sans dépiler un à un les appels de fonctions).

Les exceptions permettent de dépiler d'un seul coup plusieurs appels de fonction, jusqu'au prochain « harnais » *(handler)*. Lorsqu'on lève une exception, il faut se placer dans le *contexte* du dernier **`try ... with`** non terminé : on dépile tout ce qui a été empilé depuis et on exécute le **`with`**, qui peut reprendre le cours normal de l'exécution dans son contexte ou bien transférer l'exception au harnais précédent.

4.4.3 Opérateur de contrôle `callcc`

Même si elles ont suffisantes dans presque tous les cas, les exceptions ne permettent de sortir que de façon limitée du flot normal du programme. Les exceptions ne permettent que de se placer dans un contexte où la pile est une sous-pile de la pile actuelle. Autrement dit : si vous dépilez des données, vous ne pourrez plus revenir dans un contexte avec ces données sur la pile (sauf si vous les empilez de nouveau).

Certains langages, comme Scheme, permettent cependant de faire cela. Le principe consiste à sauvegarder à un moment donné tout le contexte (toute la pile et le point d'exécution), à lui donner un nom, et donner la possibilité plus tard d'y revenir. Cet *opérateur de contrôle* s'appelle `callcc` *(call with current continuation)*.

Utilisation

Nous allons utiliser une implémentation expérimentale de `callcc` en OCaml pour montrer son fonctionnement. Cette implémentation est très naïve puisqu'elle copie entièrement la pile, ce qui très inefficace. Des implémentations plus rusées permettent de ne copier que des portions de la pile. L'interface de ce module est la suivante :

```
type 'a cont = 'a Callcc.cont
val callcc : ('a cont -> 'a) -> 'a
val throw : 'a cont -> 'a -> 'b
```

Et voici des exemples d'utilisation :

```
# 1 + Callcc.callcc (fun k -> 2 + Callcc.throw k 3);;
- : int = 4
# let exists pred lst =
  Callcc.callcc (fun k ->
    List.iter
      (fun x -> if pred x then Callcc.throw k (Some x) else ())
    lst;
    None);;
val exists : ('a -> bool) -> 'a list -> 'a option = <fun>
# exists (fun x -> x<0) [1;2;3;4];;
- : int option = None
# exists (fun x -> x<0) [1;2;-3;4];;
- : int option = Some (-3)
```

Nous avons transformé `List.iter` en `List.exists` !

Il existe d'autres opérateurs de contrôle, par exemple `shift/reset` qui ont un comportement un peu différent.

Le terme *continuation*

On a vu que `callcc` signifie « appelle avec la continuation courante ». Autrement dit, c'est une instruction qui permet d'appeller la fonction en lui passant la *continuation* courante. On utilise donc le terme *continuation* pour parler du contexte courant. En informatique, le terme *continuation* est utilisé pour désigner « la suite du calcul », « ce qu'il faut faire après ». Effectivement, connaître l'état de la pile et le pointeur de programme permet de savoir exactement ce qu'il faut faire après (alors que ce qui a été fait avant, c'est-à-dire comment on est arrivé à ce point du programme, est perdu).

L'opérateur `callcc` permet de manipuler les continuations comme des *valeurs de première classe*.

Chapitre 5

Monades et transformations de programmes

Ce chapitre est inspiré de cours de Xavier Leroy, Giuseppe Castagna et Andrew Appel.

On a vu qu'ajouter les fermetures à un langage (donc d'en faire un langage fonctionnel) permettait d'augmenter son expressivité. Dans la première section de ce chapitre, nous allons voir comment simuler un langage fonctionnel lorsque l'on utilise un langage impératif.

Dans la deuxième section, nous allons voir comment se débarrasser des traits impératifs d'un programme. Pour cela nous utiliserons souvent des *monades*.

La troisième partie détaille un peu plus formellement cette notion et montre d'autres exemples.

5.1 Transformer un programme fonctionnel en un programme impératif

Vous avez vu en cours de compilation qu'il est facile d'implémenter des langages qui permettent de passer des fonctions en paramètres à d'autres fonctions. Il suffit de passer un pointeur vers le code de la fonction. Des langages comme C ou Pascal permettent de faire cela.

En revanche il est beaucoup plus compliqué de faire des fonctions qui renvoient des fonctions, parce qu'il faut renvoyer une fermeture : le pointeur vers le code, mais aussi l'environnement (valeurs des variables libres de la fonction).

5.1.1 La conversion de fermeture *(closure conversion)*

Une première idée pour simuler le comportement d'un langage fonctionnel dans un langage impératif, consiste à construire les fermetures à la main :

- Chaque fonction reçoit un paramètre supplémentaire contenant son environnement (la valeur de ses variables libres) ;
- Une fermeture est représentée par un couple dont le premier argument est cette fonction close, et le deuxième le n-uplet contenant les valeurs des variables libres.

Exemple :

Voici un programme OCaml :

```
fun x lst ->
  let rec map f lst =
     match lst with [] -> [] | hd :: tl -> f hd :: map f tl
  in
```

```
  map (fun y -> x + y) lst
```

Et le résultat d'une conversion de fermeture (partielle) pour l'argument de `map` :

```
fun x lst ->
  let rec map (fc, e) lst =
     match lst with
     | [] -> []
     | hd :: tl -> (fc (e, hd)) :: map (fc, e) tl
  in
  map ((fun (x, y) -> x + y), x) lst
```

Règle de transformation

Cela donne les règles suivantes :

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![\texttt{fun x -> } a]\!] &= ((\texttt{fun ((f, x1, ..., xn), x) -> } [\![a]\!]), \texttt{x1, ..., xn}) \\
&\quad \text{où x1, ..., xn sont les variables libres de } \texttt{fun x -> } a \\
[\![a\ b]\!] &= \texttt{let ((f, x1, ..., xn) as c) =} [\![a]\!] \texttt{ in f (c,} [\![b]\!]\texttt{)}
\end{aligned}
$$

Remarquez qu'on a mis la fonction `f` elle-même dans la clôture (`c`), pour pouvoir faire des fonctions récursives.

Conversion de fermeture en Java

En Java, on peut faire de la conversion de clôture en utilisant un objet à la place du couple. Cet objet a une méthode `apply`, et les valeurs des variables libres sont des champs de l'objet. La classe suivante correspond à la fermeture `fun x -> a` :

```
public class C {
  protected Object x1, ..., xn;
  public C(Object x1, ..., Object xn) {
    this.x1 = x1; ...; this.xn = xn;
  }
  public Object apply(Object x) {
    return [[a]];
  }
}
```

5.1.2 La défonctionalisation

Une autre technique consiste à utiliser un type somme avec un constructeur pour chaque fermeture apparaissant dans le programme. Ces constructeurs ont en arguments la valeur des variables libres de la fonction au moment où la fermeture est construite. On ajoute une fonction `apply` qui regarde ce constructeur et applique ses arguments à la bonne fonction.

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![\texttt{fun x -> } a]\!] &= \texttt{C}_k(x_1, ..., x_n) \\
&\quad \text{where } x_1, ..., x_n \text{ are the free variables of } \texttt{fun x -> a} \\
[\![a\ b]\!] &= \texttt{apply (}[\![a]\!]\texttt{,,}[\![b]\!]\texttt{)}
\end{aligned}
$$

La fonction `apply` est globale à tout le programme et rassemble le corps de toutes les fonctions et exécute la bonne en fonction du constructeur reçu. La défonctionalisation est donc une transformation de tout le programme (contrairement à la conversion de fermetures, qui peut être locale).

```
let rec apply (f, arg) =
   match f with
    | Ck (x1, ..., xn) -> let x = arg in ⟦a⟧
    | ...
in ⟦program⟧
```

Il est facile d' au traitement des fonctions récursives.
Attention : le code précédent n'est pas typable en OCaml (et avec des variants polymorphes ?).
Exemple : Défonctionalisation de `(fun x y -> x) 1 2`

```
let rec apply (f, arg) =
   match f with
   | C1 ()  -> let x = arg in C2 (x)
   | C2 (x) -> let y = arg in x
in
apply(apply(C1 (), 1), 2)
```

5.1.3 Optimiser les appels terminaux : le trampoline

Les appels terminaux

Dans un langage à pile, chaque appel de fonction empile des données qui sont dépilées lorsque la fonction termine. Dans un langage fonctionnel, on utilise beaucoup la récursion, et l'on peut avoir de longues suites d'appels de fonctions qui vont vite faire déborder la pile. C'est le cas en particulier si l'on-cherche à simuler une boucle (**`while`**, **`for`**...) avec une fonction récursive. Par exemple on aurait envie d'implémenter un serveur avec cette boucle infinie :

```
let rec loop () =
    (* do something *)
    loop ()
  in loop ();;
```

Heureusement, des langages (surtout les langages fonctionnels) optimisent les *appels terminaux (tail call)*, ce qui permet d'écrire la boucle ci-dessus sans débordement de pile. Le principe est le suivant : lorsque la dernière instruction d'une fonction est un appel à une autre fonction, rien ne sert de conserver sur la pile le bloc de la fonction appelante. Le compilateur peut donc dépiler *avant* de faire l'appel terminal. Ainsi le nombre d'appels imbriqués peut être aussi grand que l'on veut.

Dans le cas où l'appel terminal est un appel récursif à la même fonction, on peut même éviter de dépiler et empiler : on conserve le bloque existant. Dans ce cas (par exemple la fonction `loop` ci-dessus), on obtient un programme exactement équivalent à une boucle.

Simuler l'optimisation des appels terminaux

Dans un langage qui ne dispose pas de cette optimisation, il faut éviter de faire beaucoup d'appels récursifs. Cependant dans certains cas, on ne peut pas s'en passer. Par exemple si vous compilez un langage fonctionnel vers un langage sans cette optimisation (Java, Javascript,...), ou bien lorsque vous programmez en CPS (voir plus loin) ou que vous transformez vos programmes en CPS (par exemple pour simulers les opérateurs de contrôle).

Dans ces cas, vous pouvez utiliser l'astuce suivante, appelée *trampoline* : Un trampoline est un bout de code par lequel passent tous les appels des fonctions pouvant faire des appels terminaux. Au lieu de faire leur appel terminal, ces fonctions renvoient à leur trampoline :

– soit le résultat de leurs calculs (en cas de terminaison) ;

– soit une fonction et l'argument à lui passer (en cas d'appel terminal), et c'est le trampoline qui fera l'appel.

Exemple d'implémentation (ici on ne retourne que des fonctions prenant `()` en paramètre, ce qui revient au même et évite de retourner le paramètre) :

```
# type 'a trampo = Value of 'a | TailCall of (unit -> 'a trampo);;
type 'a trampo = Value of 'a | TailCall of (unit -> 'a trampo)
# let trampoline f = (* ecrite dans un affreux style imperatif
          comme si ocaml n'optimisait pas les appels recursifs *)
    let continue = ref true in
    let call = ref f in
    let res = ref None in
    while !continue do
      match !call () with
      | Value v -> res := Some v; continue := false
      | TailCall f -> call := f
    done;
    match !res with None -> assert false | Some v -> v;;
val trampoline : (unit -> 'a trampo) -> 'a = <fun>
# let rec loop' n () =
    if n = 0
    then Value ()
    else (print_int n; TailCall (loop' (n-1)));;
val loop' : int -> unit -> unit trampo = <fun>
# trampoline (loop' 10);;
10987654321- : unit = ()
```

Remarquez que `loop'` ne fait pas d'appel récursif : `loop'` ne sera réellement appelée que lorsqu'on lui passera tous ses arguments. Donc tous les appels à `loop'` se font depuis la fonction `trampoline`.

Cette technique est coûteuse. Il existe d'autres techniques pour optimiser les appels terminaux :

– Vous pouvez optimiser seulement les appels récursifs terminaux et les transformer en boucles (efficace mais ne fonctionne pas pour tous les appels terminaux) ;
– Une technique due à Baker, qui consiste essentiellement à faire une transformation CPS de tout le programme (voir plus loin). En CPS, tous les appels sont teminaux, donc les anciens blocs de pile ne sont plus utilisés. Il suffit d'effacer complètement la pile de temps en temps pour éviter qu'elle ne devienne trop grosse.

5.2 Programmer en fonctionnel pur

5.2.1 Pourquoi programmer en fonctionnel pur ?

Les ordinateurs d'aujourd'hui fonctionnent de manière très impérative :

– ils font des entrées sorties ;
– les processeurs manipulent des registres, pas des fonctions ;
– la mémoire est un tableau mutable.

Par ailleurs, des traits impératifs, comme les exceptions ou les références permettent parfois de gagner en efficacité et en clareté.

On peut donc se demander pourquoi un langage comme Haskell a fait le choix du fonctionnel pur, c'est-à-dire de ne proposer aucun trait impératif. La raison vient du fait que le fonctionnel pur a (lui aussi) beaucoup d'avantages :

– L'analyse statique de programmes est beaucoup plus facile, ainsi que les preuves de correction de programmes ;
– on peut faire varier l'ordre d'évaluation, par exemple utiliser l'appel par nécessité (pas possible en présence d'effets de bord) ;

– Beaucoup d'optimisations sont possibles, par exemple :

la déforestation : faire une seule passe au lieu de deux. Par exemple, si `f` et `g` ne font pas d'effet de bord, on peut remplacer `List.map f (List.map g l)` par `List.map (fun x -> f (g x)) l`.

la mémoïsation de valeurs : une fonction appliquée à un même argument renverra toujours la même valeur).

la parallélisation : en l'absence d'effets de bords, deux calculs peuvent être faits en parallèle, et en l'absence de données mutables plus de problème d'accès concurrent aux valeurs ;

la réduction forte : en l'absence d'effets de bord, il devient possible souvent de réduire au moment de la compilation le corps des fonctions (réduire sous les λ) pour faire tous les calculs connus au moment de la compilation et gagner du temps à l'exécution. Cette technique est particulièrement utile lorsque vous écrivez une fonction générique et que vous voulez en déduire une version spécialisée pour une donnée particulière. On parle alors de *spécialisation* ou *évaluation partielle*.

5.2.2 Simuler les exceptions

Principe

Comment simuler le fonctionnement des exceptions dans un langage fonctionnel pur ?
Idée : on ne pourra plus court-circuiter le dépilement, mais on peut propager les exceptions.
Une fonction qui peut « échouer » renverra soit une valeur, soit une exception. En OCaml, on peut par exemple utiliser le type :

```
type 'a value = Exn of exn | Value of 'a;;
```

Exemple d'utilisation :

```
# exception Empty;;
exception Empty

# let head = function
  | [] -> Exn Empty
  | a::l -> Value a;;
val head : 'a list -> 'a value = <fun>

# head [];;
- : 'a value = Exn "Liste vide"

# head [5; 6; 7];;
- : int value = Value 5
```

Mais l'utilisation telle quelle est très lourde :

```
# let div a b = if b = 0 then Exn Division_par_zero else Value (a/b);;
val div : int -> int -> int value = <fun>

# (div 3 0) + 4;;
Characters 0-9:
  (div 3 0) + 4;;
  ^^^^^^^^^
This expression has type int value but is here used with type int

# let plus a b = match a,b with
  | Value a, Value b -> Value (a+b)
```

```
  | Exn s, Value _ -> Exn s
  | Value _, Exn s -> Exn s
  | Exn s1, Exn s2 -> Exn s1 (* ou s2 ?? *);;
val plus : int value -> int value -> int value = <fun>

# let div a b = match a,b with
  | Value a, Value b -> if b = 0 then Exn Division_par_zero else Value (a/b)
  | Exn s, Value _ -> Exn s
  | Value _, Exn s -> Exn s
  | Exn s1, Exn s2 -> Exn s1 (* ou s2 ?? *);;
val div : int value -> int value -> int value = <fun>

# plus (div (Value 3) (Value 0)) (Value 2);;
- : int value = Exn Division_par_zero
```

Manipulation simple du type `'a value`

Il faudrait trouver un moyen plus simple pour :
- utiliser le résultat ou propager les exceptions ;
- rattraper les exceptions.

En fait il nous faut un moyen d'enchaîner les calculs, en oubliant le fait que notre valeur est encapsulée dans le type `'a value`. On va même rendre ce type abstrait. On va définir une fonction qui prendra une `'a value` et la fonction à laquelle on voudrait la passer. Elle passe la valeur à cette fonction ou bien propage l'exception :

```
# let bind v f = match v with
  | Value w -> f w
  | Exn e -> Exn e;;
val bind : 'a value -> ('a -> 'b value) -> 'b value = <fun>
```

Cela vous rappelle quelque chose ?

Il nous faut une fonction pour créer une valeur sans échec et une pour créer une valeur échouée :

```
# let return a = Value a;;
val return : 'a -> 'a value = <fun>
# let fail e = Exn e;;
val fail : exn -> 'a value = <fun>
```

Exemple :

```
# let (>>=) = bind;;
# let div va vb =
    va >>= fun a ->
    vb >>= fun b ->
    if b = 0
    then fail Division_par_zero
    else return (a/b);;
val div : int value -> int value -> int value = <fun>
# div (return 4) (return 2);;
- : int value = Value 2
# div (return 4) (return 0);;
- : int value = Exn Division_par_zero
# div (return 4) (return 0) >>= fun x -> return (x+1);;
- : int value = Exn Division_par_zero
```

Monade

On appelle le type `'a value` avec ses opérations associées la *monade d'exception*.

Remarquez que comme pour Lwt, une expression de la forme `e1 >>= fun x -> e2` peut se lire comme `let x = e1 in e2`, mais cette fois-ci avec un `let` qui propage les exceptions.

Remarquons que à partir du moment où notre calcul est de type `'a value`, le seul moyen de l'utiliser est de le faire avec la fonction `bind`, qui renvoie une `'b value`. Autrement dit, une fois que l'on travaille dans la monade, on y reste (puisque l'on doit toujours propager une éventuelle exception).

On peut cependant définir une fonction `run : 'a value -> 'a` qui va nous permettre de sortir de la monade et revenir au style avec exceptions.

```
let run = function
  | Value v -> v
  | Exn e -> raise e
```

Rattraper les exceptions

Il nous faut maintenant un façon de rattraper les exceptions. On peut le faire en définissant la fonction suivante :

```
# let catch v f =
    match v with
    | Exn e -> f e
    | a -> a;;
val catch : 'a value -> (exn -> 'a value) -> 'a value = <fun>
# catch (div (return 4) (return 0))
        (function Division_par_zero -> return 999 | e -> fail e);;
 - : int value = Value 999
```

Ne pas oublier de repropager l'exception si elle n'est pas interceptée.

5.2.3 Simuler les références

Dans un langage impératif, les valeurs sont mutables. C'est-à-dire : il est possible de modifier des valeurs après leur construction. Un langage impératif dispose donc d'un *état* global la mémoire), qui est en fait une table d'association entre adresses et valeurs.

Exemple avec une seule référence

Essayons de traduire le programme suivant en fonctionnel pur.

```
let r = ref 0 in
let f x = r := !r + 1; 4*x in
let v = f 3 in
v + !r
```

Pour éviter la référence, nous allons passer la valeur courante du compteur en paramètre supplémentaire de la fonction `f` et lui faire renvoyer la nouvelle valeur, en plus de son résultat.

```
let f x c = (4*x, c+1) in
let c = 0 in
let (v, c) = f 3 c in
v + c
```

Simuler un état extensible

Pour simuler une mémoire pouvant contenir plusieurs références avec en programmation fonctionnelle pure, nous allons propager un état non mutable (appelons son type `store`) dans tous nos calculs. Cet état sera une table d'association entre références et valeurs, avec les opérations suivantes [1]

```
val empty_store : store
val read_store : 'a reference -> store -> 'a
val write_store : 'a reference -> 'a -> store -> store
val new_ref : 'a -> store -> 'a reference * store
```

Il faut ensuite rajouter un paramètre à chacune des fonctions de notre programme (en tout cas celle modifiant le store) pour passer le store courant, et renvoyer le store modifié en plus du résultat. Par exemple le programme suivant

```
let next =
  let r = ref 0 in
  fun () -> let v = !r + 1 in r := v; v
in next () + next ()
```

pourrait être traduit par :

```
let (next, store0) =
  let (r, store0) = new_ref 0 empty_store in
  ((fun store ->
      let v = read_store r store + 1 in
      (v, write_store r v store)),
   store0)
in
let (v1, store1) = next store0 in
let (v2, store2) = next store1 in
v1 + v2
```

Monade

On aimerait que la manipulation des stores soit plus transparente. En fait nous voulons transformer une fonction de type t_1 `->` t_2 en une fonction de type (t_1 `* store) -> (`t_2 `* store`) ou (version curryfiée) : t_1 `-> store -> (`t_2 `* store`).

On pourrait être tenté de définir un type :

```
type 'a t = 'a * store
```

mais cette définition ne simplifierait pas beaucoup l'écriture. Nous voudrions définir encore une fois une opération `>>=` mais qui cette fois va propager les états. L'exemple précédent devrait pouvoir s'écrire :

1. On peut implémenter ces fonctions avec n'importe quels tableaux extensibles. Une implémentation très basique pourrait être :

```
type 'a ref = int (* abstrait dans l'interface *)
let empty_store = []
let read_store r s = List.nth s r
let rec write_store r v s = match r, s with
  | 0, a::l -> v::l
  | n, a::l -> a::write_store (n-1) v l
  | _ -> assert false (* avec la monade et le type abstrait, ce cas est impossible *)
let new_ref v s =
  let rec aux n = function
    | [] -> n, [v]
    | a::l -> let m, ll = aux (n+1) l in m, a::ll
  in aux 0 s
```

Mais en fait il faudrait pouvoir stocker des valeurs de type quelconque dans la table... Comment faire ça ? Vous pouvez jeter un œil sur le module `Polytable` d'Ocsigen.

```
(make_ref 0 >>= fun r ->
 return (fun () ->
           get r >>= fun w ->
           let v = w+1 in            (* ou : return (w+1) >>= fun v -> *)
           set r v >>= fun () ->
           return v)) >>= fun next ->
next () >>= fun r1 ->
next () >>= fun r2 ->
return (r1 + r2)
```

Si ça ne vous paraît pas clair, essayez de réécrire ce programme en transformant les `>>=` en **`let`**.

Ici on utilise une monade `'a t` avec les primitives suivantes :

```
return : 'a -> 'a t
bind : 'a t -> ('a -> 'b t) -> 'b t
run : 'a t -> 'a
make_ref : 'a -> 'a reference t
get : 'a reference -> 'a t
set : 'a reference -> 'a -> unit t
```

En comparant `make_ref`, `get` et `set` respectivement avec les fonctions non monadiques `new_ref`, `read_store` et `set`, on voit que l'on peut définir

```
type 'a t = store -> 'a * store
```

On appelle cela la monade d'état.

Voici l'implémentation des primitives :

```
let return a s = (a, s)
let bind v f s0 =
  let (w, s1) = v s0 in
  f w s1
let run v = fst (v empty_store)
let make_ref = new_ref
let get r s = (read_store r s, s)
let set r v s = ((), write_store r v s)
```

5.2.4 Manipuler les continuations explicitement

Continuation-passing style (CPS)

Nous avons vu qu'il existe des opérateurs de contrôles permettant de manipuler les *continuations* comme des valeurs de première classe. Une continuation représente la suite du calcul. Dans un langage à pile, elle est entièrement définie par le point d'exécution du programme et l'état de la pile.

Dans un langage qui n'a pas d'opérateur de contrôle, on peut utiliser un style de programmation appelé « passage de continuations ». Cela consiste à passer en paramètre supplémentaire à chaque fonction sa continuation, (sous la forme d'une fermeture). Les fonctions ne renvoient jamais plus de résultat mais au lieu de cela appellent la continuation sur leur résultat. La dernière instruction de chaque fonction est donc l'appel de la continuation. Si le langage n'optimise pas les appels terminaux, la pile va exploser très vite puisque l'on ne dépile jamais (les fonctions ne retournent jamais). Mais dans un langage qui optimise les appels terminaux, notre programme n'utilisera quasiment pas de pile (en fait un seul bloc de pile, celui de la fonction courante).

Exemple : Considérons le programme `(1+2)*(3+4)`. La continuation de `(1+2)`, exprimée sous la forme d'une fonction, est **`fun`** `x -> x * (3+4)`. La continuation de `(3+4)` est la fonction **`fun`** `x -> 3 * x` (car `(1+2)` a été évalué en `3`).

Un programme qui n'est pas en CPS est dit en *style direct*.

Formellement, on peut définir la continuation d'une sous-expression a d'un programme p par le calcul qu'il reste à faire pour obtenir p si l'on a déjà évalué a. On la représente par une fonction qui associe à la valeur de la sous-expression a la valeur du programme p.
Autre exemple : voici une fonction en style direct

```
let rec fact n = if n = 0 then 1 else n * fact (n-1)
```

et sa version en CPS :

```
# let rec fact n k =
    if n = 0
    then k 1
    else fact (n-1) (fun m -> k (n * m));;
val fact : int -> (int -> 'a) -> 'a = <fun>
# fact 5 print_int;;
120- : unit = ()
```

Autre exemple : voici une version CPS des flux. Plutôt que faire un flux qui renvoie une valeur nouvelle à chaque fois, on peut faire un flux qui prend une fonction est l'applique à une nouvelle valeur à chaque fois.

```
# type 'a streamcell = Fini | Cont of 'a * 'a stream
and 'a stream = unit -> 'a streamcell;; (* version en style direct *)
type 'a streamcell = Fini | Cont of 'a * 'a stream
and 'a stream = unit -> 'a streamcell
# let rec f i = fun () -> Cont (2*i, f (i+1));; (* flux des entiers pairs *)
val f : int -> int stream = <fun>
# f 0;;
- : int stream = <fun>
# f 0 ();;
- : int streamcell = Cont (0, <fun>)

# type 'a streamcell = Fini | Cont of 'a stream
and 'a stream = ('a -> unit) -> 'a streamcell;; (* version en cps *)
type 'a streamcell = Fini | Cont of 'a stream
and 'a stream = ('a -> unit) -> 'a streamcell
# let rec f i = fun k -> k (2*i) ; Cont (f (i+1));;
val f : int -> int stream = <fun>
# f 0 print_int;;
0- : int streamcell = Cont <fun>
```

Monade des continuations

On aimerait simplifier l'écriture d'un programme en CPS en utilisant une monade. Pour cela nous définissons :

```
# type answer = unit;;
type answer = unit
# type 'a t = ('a -> answer) -> answer;;
type 'a t = ('a -> answer) -> answer
# let return a : 'a t = fun k -> k a;;
val return : 'a -> 'a t = <fun>
# let (>>=) (m : 'a t) (f : 'a -> 'b t) : 'b t = fun k -> m (fun v -> f v k);;
val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t = <fun>
# let run (m : unit t) = m (fun x -> x);;
val run : unit t -> unit = <fun>
```

Remarquons que nous avons choisi une fois pour toute le type du résultat du programme (ici `unit`, mais cela pourrait être par exemple `int`).

Et voici la factorielle écrite avec les monades :

```
# let rec fact n =
    if n = 0
    then return 1
    else fact (n-1) >>= fun m -> return (n * m);;
val fact : int -> int t = <fun>
# fact 5 >>= fun v -> print_int v; return ();; (* c'est une fermeture donc pas evalue *)
- : unit t = <fun>
# run (fact 5 >>= fun v -> print_int v; return ());;
120- : unit = ()
```

Simuler `callcc`

On peut définir `callcc` et `throw` :

```
# let callcc f = fun k -> f k k;;
val callcc : ('a -> 'a -> 'b) -> 'a -> 'b = <fun>
# let throw x y = fun k -> x y;;
val throw : ('a -> 'b) -> 'a -> 'c -> 'b = <fun>
```

Notre exemple `1 + Callcc.callcc (fun k -> 2 + Callcc.throw k 3)` s'écrit maintenant :

```
# run (callcc (fun k ->
         throw k 3 >>= fun trois ->
         return (2 + trois)) >>= fun dpt ->
       return (1 + dpt));;
- : int = 4
```

Transformation CPS

On peut définir une transformation de programmes appelée *transformation CPS* qui permet de transformer automatiquement un programme écrit en style direct en un programme complètement en CPS. Tous les appels de fonctions prennent alors la continuation en paramètre, et les fonctions terminent en faisant l'appel à cette continuation. Les fonctions ne retournent donc jamais, et tous les appels sont terminaux. Donc dans un langage qui optimise les appels terminaux, la pile ne croît jamais.

Voici la transformation CPS en appel par valeur de gauche à droite :

$$
\begin{aligned}
[\![i]\!] &= \texttt{fun k -> k } i \\
[\![x]\!] &= \texttt{fun k -> k } x \\
[\![\texttt{fun x -> } a]\!] &= \texttt{fun k -> k (fun x ->}[\![a]\!]\texttt{)} \\
[\![a\ b]\!] &= \texttt{fun k ->}\ [\![a]\!]\ \texttt{(fun x ->}[\![b]\!]\ \texttt{(fun y -> x y k))} \\
[\![callcc\ a]\!] &= \texttt{fun k ->}\ [\![a]\!]\ \texttt{k k} \\
[\![throw\ a\ b]\!] &= \texttt{fun k ->}\ [\![a]\!]\ \texttt{(fun x ->}[\![b]\!]\texttt{(fun y -> x y))}
\end{aligned}
$$

5.3 Monades

5.3.1 Définition

Une monade est définie par un type paramétré `'a t` et des opérations :

```
return : 'a -> 'a t
bind : 'a t -> ('a -> 'b t) -> 'b t
run : 'a t -> 'a
```

Le type `'a t` est le types des calcul qui vont produire une valeur de type `'a`.
`return a` encapsule une expression `a` comme un calcul trivial qui produit immédiatement la valeur de `a`.
`bind a (fun x ->`t`)` fait le calcul de `a` et lie sa valuer à `x` pour fair le calcul de tt
`run a` exécute un programme monadique complet et extrait sa valeur de retour.

Les opérations `return` et `>>=` doivent vérifier les lois suivantes :

`return a >>= f`	$\approx$	`f a`	neutralité à gauche
`a >>= fun x -> return x`	$\approx$	`a`	neutralité à droite
`(a >>= fun x -> b) >>= fun y -> c`	$\approx$		
`a >>= fun x -> b >>= fun y -> c`			associativité

Il faudrait définir $\approx$ plus précisément. Mais intuitivement cela signifie « se comporte de la me façon ».

5.3.2 Autres exemples de monades

Monade des listes

Dans la monade des listes, le type `'a t` est le type des listes, et l'opération `bind` d'un liste `l` sur une fonction `f : 'a -> 'a list` permet de construire la concaténation des listes obtenues en appliquant `f` à tous les éléments de la liste initiale.

Exemple : à partir de deux ensembles d'entier A et B, vous voulez construire l'ensemble $\{x + y \mid x \in A,\ y \in B\}$. En représentant les ensembles par des listes, on peut écrire :

```
# [1;2;3] >>= fun a ->
  [100;200;300] >>= fun b ->
  return (a+b);;
- : int list = [101; 201; 301; 102; 202; 302; 103; 203; 303]
```

Voici l'implémentation de la monade :

```
module ListMonad = struct
  type 'a mon = 'a list
  exception Failed
  let return a = a :: []
  let rec bind m f =
    match m with [] -> [] | hd :: tl -> f hd @ bind tl f
  let run m = match m with hd :: tl -> hd | _ -> raise Failed
  let runall m = m
  let fail = []
  let either a b = a @ b
end
```

Voici la fonction qui calcule toutes les façons d'insérer un élément dans une liste :

```
# let rec insert x l =
  either (return (x :: l))
    (match l with
     | [] -> fail
     | hd :: tl ->
        bind (insert x tl) (fun l' -> return (hd :: l')));;
val insert : 'a -> 'a list -> 'a list list = <fun>
# insert 100 [1; 2; 3; 4];;
- : int list list =
[[100; 1; 2; 3; 4]; [1; 100; 2; 3; 4]; [1; 2; 100; 3; 4];
 [1; 2; 3; 100; 4]; [1; 2; 3; 4; 100]]
```

Voici la fonction qui calcule toutes les permutations d'une liste :

```
# let rec permut = function
  | [] -> return []
  | hd :: tl -> bind (permut tl) (fun l' -> insert hd l');;
val permut : 'a list -> 'a list list = <fun>
# permut [1; 2; 3];;
- : int list list =
[[1; 2; 3]; [2; 1; 3]; [2; 3; 1]; [1; 3; 2]; [3; 1; 2]; [3; 2; 1]]
```

Monade des flux

Dans l'implémentation précédente, on calcule toute la liste d'un coup, ce qui peut prendre beaucoup de place en mémoire. En remplaçant les listes par des flux, on obtient un flux qui énumère les résultats.

```
type 'a streamcell = Fini | Cont of 'a * 'a stream
and 'a stream = unit -> 'a streamcell
let rec concat fl1 fl2 = match fl1 () with
  | Fini -> fl2
  | Cont (s, f) -> fun () -> Cont (s, concat f fl2)

module StreamMonad = struct
  type 'a mon = 'a stream
  exception Failed
  let return a () = Cont (a, fun () -> Fini)
  let rec bind m f =
    match m () with
    | Fini -> fun () -> Fini
    | Cont (hd, tl) -> concat (f hd) (bind tl f)
  let run m = match m () with Cont (hd, tl) -> hd | _ -> raise Failed
  let runall m : 'a stream = m
  let fail () = Fini
  let either a b = concat a b
end
```

On peut définir `insert` et `permut` exactement comme avant (copier/coller du code) :

```
# insert 100 [1; 2; 3; 4];;
- : int list stream = <fun>
# insert 100 [1; 2; 3; 4] ();;
- : int list streamcell = Cont ([100; 1; 2; 3; 4], <fun>)
# permut [1; 2; 3] ();;
- : int list streamcell = Cont ([1; 2; 3], <fun>)
```

Les fonctions `insert` et `permut` dans la monade des continuations

On peut définir `either` et `fail` dans la monade des continuations :

```
let either a b k = a k; b k
let fail k = ()
```

Que font `insert` et `permut` dans ce cas ?

```
# insert 100 [1; 2; 3; 4] (fun l -> print_int (List.length l));;
55555- : unit = ()
# permut [1; 2; 3] (fun l -> print_int (List.hd l));;
122133- : unit = ()
```

5.3.3 Transformations de monades

Lorsque l'on veut travailler avec plusieurs monades en même temps, il faut construire la monade qui réalise la composition des deux monades.

En OCaml, on peut faire des transformateurs de monades avec des foncteurs. Voici la signature des monades :

```
module type MONAD = sig
  type 'a mon
  val ret: 'a -> 'a mon
  val bind: 'a mon -> ('a -> 'b mon) -> 'b mon
  val run: 'a mon -> 'a
end
```

La monade identité :

```
module Identity = struct
  type 'a mon = 'a
  let ret x = x
  let bind m f = f m
  let run m = m
end
```

Le transformeur qui rajoute les exceptions :

```
module ExceptionTransf(M: MONAD) = struct
  type 'a outcome = V of 'a | E of exn
  type 'a mon = ('a outcome) M.mon
  let ret x = M.ret (V x)
  let bind m f =
    M.bind m (function E e -> M.ret (E e) | V v -> f v)
  let lift x = M.bind x (fun v -> M.ret (V v))
  let run m = M.run (M.bind m (function V x -> M.ret x))
  let raise e = M.return (E e)
  let trywith m f =
    M.bind m (function E e -> f e | V v -> M.ret (V v))
end
```

Le transformeur qui rajoute la monade des continuations :

```
module ContTransf(M: MONAD) = struct
  type 'a mon = ('a -> answer M.mon) -> answer M.mon
  let ret x = fun k -> k x
  let bind m f = fun k -> m (fun v -> f v k)
  let lift m = fun k -> M.bind m k
  let run m = M.run (m (fun x -> M.ret x))
  let callcc f = fun k -> f k k
  let throw c x = fun k -> c x
end
```

etc.

Chapitre 6

Cours d'Haskell pour programmeurs OCaml qui connaissent les monades

Haskell est un langage généraliste purement fonctionnel paresseux. Il tire son nom du logicien Haskell Brooks Curry (1900-1982), qui est un des pères du λ-calcul (et qui a donné aussi son nom à la *curryfication* et à la *correspondance de Curry-Howard*). Il dispose d'un système de types très expressif avec types sommes, et *classes de types*, et utilise l'inférence de types.

- purement fonctionnel $\Rightarrow$ pas de traits impératifs (utilisation de monades)
- paresseux : toutes les valeurs sont paresseuses (appel par nécessité)

Le langage a été créé en 1985. Il est défini précisément par une norme, appelée Haskell 98. Il existe plusieurs implémentations d'Haskell, notamment GHC et Hugs (interpréteur). Nous utiliserons GHC.
Compiler : `ghc toto.hs`
Boucle interactive : `ghci`

Références : `http://book.realworldhaskell.org/`
`http://gorgonite.developpez.com/livres/traductions/haskell/gentle-haskell/`
et *Une introduction à Haskell* d'Emmanuel Beffara (dont j'ai repris beaucoup d'exemples).

6.1 Syntaxe

6.1.1 Syntaxe de base, définitions de valeurs, fonctions

Syntaxe de base :

- les noms des modules, **des types** et des constructeurs commencent par une majuscule, les variables par une minuscule ;
- le nommage des fonctions appartenant à d'autres modules se fait comme en OCaml : `Array.IArray`, `List.length` ;
- Comme en OCaml, les opérateurs infixes sont des suites de symboles non-alphanumériques : `*`, `>>=`, `++`,...
- Commentaires : *`{- bla bla -}`* ou bien *`-- commentaire en fin de ligne`*

Comme en OCaml on a une valeur `()` d'un type singleton (le type s'appelle lui aussi `()`).

La définition de valeurs se fait de façon très déclarative, sans **`let`** :

```
answer = 42
twice n = 2 * n
something x y = x + y * (x - 1)
```

L'ordre de déclaration n'a pas d'importance. Une définition peut utiliser une valeur définie après.

6.1.2 Listes et chaînes de caractères

```
[]
[1, 2, 3]
1:2:3:[]
[1, 2] ++ [3, 4]   -- concatenation
[1, 2, 3]!!2       -- 2e element (en comptant a partir de 0)
[x + 3 | x <- lst ]  -- equivalent a    map (\x -> x + 3) lst
[x / 2 | x <- lst, x > 5]   -- map, suivi de filter
[1..4]           -- enumeration
                 -- (une fonction qui cree la liste des entiers entre deux valeurs)
```

La syntaxe des dernières lignes s'appelle une écriture des listes *en compréhension*.

Allez faire un petit tour dans la documentation du module **`List`**.

Les chaînes de caractères sont des listes de caractères :

```
"lkjljlj\nljlj"            -- chaine avec un retour chariot

Prelude> ['h', 'e', 'l', 'l', 'o']
"hello"
```

6.1.3 Filtrage

Le filtrage (pattern-matching) s'introduit par la construction **`case ... of`** :

```
 case wrapped of
      Nothing     -> defval
      Just value  -> value
```

(Les contructeurs **`Nothing`** et **`Just`** appartiennent au type **`Maybe`** `a`, équivalent du type `'a option` en OCaml).
Lorsque la définition d'une fonction se fait par cas, on peut faire le filtrage en écrivant plusieurs définitions successives :

```
fac 0 = 1
fac n = n * fac (n - 1)

length [] = 0
length (_:tail) = 1 + length tail

somme [] _ = []
somme _ [] = []
somme (h1:t1) (h2:t2) = (h1 + h2):somme t1 t2
```

Notez que les valeurs sont toujours récursives : fonctions (cf ci-dessus), mais aussi autres valeurs :

```
l = 1:l
```

Deux exemples de pattern-matching avec gardes (**`where`** en OCaml) :

```
fibo n | n <= 1 = 1
       | True = fibo (n - 1) + fibo (n - 2)

machin x y | x > 1 = y + 1
           | y <= 100 = x * y
           | otherwise = x - y
```

6.1.4 Définitions locales

```
let x = 3
    y = 4
in x+y

x+y where x = 3
          y = 4

machin x y | x > z = y + 1
           | y <= z = x * y
           | otherwise = z / (x - y)
  where z = (x + y) / 2
```

6.1.5 Indentation

L'indentation est significative !

```
z = f a + f b where a = 5
                    b = 4
                    f x = x+1
```

est équivalent à

```
z = f a + f b where {a = 5; b = 4; f x = x + 1}
```

et à :

```
z = f a + f b where a = 5; b = 4
                    f x =x +1
```

6.1.6 Fonctions

Fonctions anonymes : `\x -> x` (pour $\lambda x.x$) et `\x y -> x` (pour $\lambda x.\lambda y.x$)
Composition de fonctions : `f . g . h` (pour $f \circ g \circ h$)
Application : `f $ g $ h x` (pour $f(g(h(x)))$)

Opérateurs binaires

Chaque opérateur binaire peut avoir une priorité donnée :
`infixr 6 +` déclare + associatif à droite de priorité 6.

Chaque opérateur peut être appliqué partiellement :
- `(/)` est la fonction de division
- `(3/)` est équivalent à `\x 3 / x`
- `(/3)` est équivalent à `\x x / 3`

Exemple :

```
map (/2) $ filter (>5) lst
```

(peut s'écrire aussi en compréhension : `[x / 2 | x <- lst, x > 5]`).

Les fonctions peuvent être utilisées de façon infixe :
`lst `union` [1, 2, 3]` est équivalent à `union lst [1, 2, 3]`

6.1.7 Quelques petits exemples compacts

En Haskell, on aime programmer en combinant des opérateurs très génériques :

```
sum = foldl (+) 0
combinedLength = foldr ((+) . length) 0
```

Voici le tri rapide :

```
qsort [] = []
qsort (x : xs) = qsort lower ++ [x] ++ qsort upper
  where (lower, upper) = partition (<x) xs

changestr str =
  filter
    (\c -> if (c == '-') || (c == '\'') || (c == ' ')
           then False
           else True)
  str
```

6.2 La boucle d'interaction et le compilateur

6.2.1 Compiler

Le compilateur s'appelle `ghc`. La commande `ghc toto.hs` produit un `toto.hi` (interface compilée), un `toto.o` et un `a.out` (exécutable natif).

Le programme principale est une valeur `main`, dont l'exécution est forcée (les autres ne sont évaluées qu'au besoin). Exemple (avec la fonction `changestr` définie ci-dessus) :

```
main = putStrLn (changestr "a-b'c de")
```

La fonction **`putStrLn`** affiche une chaîne de caractères.

Avec l'option `-c` vous ne produisez que le `.o` et le `.hi`. Elle est obligatoire si votre module ne contient pas de valeur `main`. Ensuite vous liez les différents `.o` avec la commande `ghc` (et éventuellement l'option `-o` pour spécifier le nom de l'exécutable).

6.2.2 Utilisation interactive

Le programme `ghci` permet de taper des commandes Haskell et supporte un certain nombre de directives dont :

`:load` *nom*	charge un fichier `.hs`
`:reload`	recharge le script courant
`:module` *nom*	charge un module `.hs`
`:type` *expr*	montre le type de l'expression
`:?`	aides des commandes de `ghci`
`:quit`	quitter
`:info` *nom*	permet d'avoir des information sur un nom (de type, de classe, de valeur...)

Le module initialement chargé, qui contient la bibliothèque standard de base s'appelle le prélude. Le prompt indique les modules actuellement chargés.

```
$ ghci
GHCi, version 6.8.2: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
Prelude>
```

Vous pouvez changer le prompt avec la commande `:set prompt`.

Attention ! la boucle interactive ne s'utilise pas comme celle d'ocaml (ou python etc). Vous ne pouvez pas copier/coller n'importe quel morceau de code Haskell. Un seul script (chargé avec `:load`) est en cours d'utilisation en même temps (programme principal).

Vous pouvez taper des expressions, mais elles ne doivent pas prendre plus d'une ligne. Ou alors il faut les encadrer pas `:{` et `:}` (directives spéciales de la boucle d'interaction).

Vous ne pouvez pas définir de valeurs directement. Cependant `ghci` introduit un mot-clé spécial **`let`** (sans **`in`**) (qui ne doit pas être utilisé dans les programmes).

```
Prelude> let {f 0 = 0 ; f n = 8}
Prelude> f 0
0
Prelude Main> f 3
8
```

Pour exécuter un script Haskell, on peut faire :
`runhaskell toto.hs` (équivalent à `runghc toto.hs`)

6.2.3 Modules

Si votre programme contient plusieurs fichiers, chacun doit déclarer un nom de module (on ne peut pas avoir plusieurs modules dans un même fichier). Pour cela, vous mettez au début du fichier :

```
module MonModule where
```

Par défaut toutes les valeurs sont exportées. Pour en cacher certaines, vous pouvez préciser entre parenthèse les valeurs et types que vous voulez exporter :

```
module MonModule
    ( MonType(..)
    , mavaleur1
    , mavaleur2
    ) where
```

La syntaxe `(..)` à côté du type indique que vous voulez exporter le type avec ses constructeurs. Si vous l'omettez, le type sera abstrait.

Pour utiliser des valeurs définies dans un autre module, vous préciser le nom du module suivi d'un point et du nom de la valeur : `MonModule.mavaleur1`. Pour éviter d'avoir à écrire le nom du module à chaque fois, vous pouvez écrire **`import`** `AutreModule` au début du fichier juste après le **`where`**.

Pour créer un exécutable, le compilateur cherche un module `Main` qui contient une valeur `main`. Le nom de module par défaut est `Main`, si un autre n'est pas précisé.

6.3 Types

Avec GHCi, on peut afficher le type d'une expression avec la commande `:type` *e* (qui peut s'abréger `:t` *e*). On peut aussi demander d'afficher les types à chaque fois avec `:set +t`.

6.3.1 Types prédéfinis

Exemples de types :

```
Prelude> :type Nothing
Nothing :: Maybe a
Prelude> :t Just Nothing
Just Nothing :: Maybe (Maybe a)
Prelude> :t Just
Just :: a -> Maybe a
```

On peut diviser les types en :

Types de base
- Bornés : `()`, **`Bool`** , **`Int`**, **`Char`**
- Non bornés : **`Integer`**

Types construits
- listes : `[a]`
- tuples : `(a, b, c)`
- fonctions : `a -> b`
- et aussi **`Maybe`** `a` (avec les constructeurs **`Nothing`** ou **`Just`** `a`), **`Either`** `a b` (contructeurs **`Left`** `a` ou **`Right`** `b`),...

On remarque qu'en Haskell, les constructeurs de types listes ou tuples sont les mêmes que les constructeurs des données. On remarque aussi que le paramètre de type est postfixé.

6.3.2 Déclarations de types

Syntaxe

Il y a deux mots clés pour définir de types : on utilise **`type`** pour définir des synonymes, et **`data`** pour définir des types sommes.

```
type String = [Char]
data Bool = False | True
data Tree a b = Leaf b
              | Node a (Tree a b) (Tree a b)
type SimpleTree = Tree ()     -- application partielle !
```

Les constructeurs s'écrivent de façon curryfiée par défaut et peuvent être appliqués partiellement. On peut aussi donner des étiquettes de champs (sorte de records). Par exemple :
`data Point = Pt {pointx, pointy :: Float}`
(qui définit les deux fonctions `pointx` et `pointy`)

```
*Main> let p = Pt {pointx=3, pointy=4}
*Main> pointx p
3.0
```

Type algébriques généralisés *GADT*

Certaines implémentations d'Haskell ont les GADT (dont on a parlé au chapitre 3).
Par exemple définissons un type `Liste`.

```
data Liste a = Vide | Cons a (Liste a)
```

En fait [1] ce type peut aussi se définir :

```
data Liste a where Vide :: Liste a
                   Cons :: a -> (Liste a) -> (Liste a)
```

et l'on peut ainsi préciser les valeurs des paramètres de type [2] :

1. en lançant `ghc` avec l'option `-XGADTs`
2. Pour faire des types vides, j'ai dû ajouter l'option `-XEmptyDataDecls`

```
data LVide
data LNonVide
data Liste a b where Vide :: Liste a LVide
                     Cons :: a -> (Liste a c) -> (Liste a LNonVide)
```

6.3.3 Prototypage

Haskell fait de l'inférence de types, mais vous pouvez aussi préciser les types vous-même :

```
succ :: Integer -> Integer
succ n = n + 1

rev = do_rev []
  where do_rev :: [a] -> [a] -> [a]
        do_rev acc [] = acc
        do_rev acc (h:t) = do_rev (h:acc) t
```

Un prototype est une « déclaration » presque comme les autres. Préciser les types n'est jamais nécessaire mais cela permet d'éviter des erreurs ou de spécifier des types plus précis que ceux inférés.

6.3.4 Sortes

Formellement, l'algèbre des types Haskell se définit par

$$\begin{array}{rcl} \tau & ::= & C \\ & | & \alpha \\ & | & \texttt{forall}\ \alpha.\ \tau \\ & | & \tau\ \tau \end{array}$$

où

- C est une constante, par exemple **`Bool`**, **`Int`**, **`Maybe`**, **`Either`**, `(->)`...
- α est une variable de type, par exemple `a` ou `b`.

En particulier `a -> b` peut aussi s'écrire `(->) a b`.

Chaque constante ou variable de type a une *sorte* (« type » de type) :

```
Prelude> :kind Maybe
Maybe :: * -> *
Prelude> :kind Int
Int :: *
Prelude> :kind []
[] :: * -> *
Prelude> :kind (->)
(->) :: ?? -> ? -> *
Prelude> :kind (,,,,)
(,,,,) :: * -> * -> * -> * -> * -> *
...
Prelude Main> :kind Tree    -- defini ci-dessus
Tree :: * -> * -> *
Prelude Main> :kind SimpleTree
SimpleTree :: * -> *
```

On peut même paramétrer un type par une sorte. Ici on définit un type d'arbre générique :

```
data GenericTree b v = Leaf v
                     | Node (b (GenericTree b v ))
```

Puis on en déduit un type pour les arbre n-aires :

```
type RoseTree = GenericTree []
```

Et un type pour les arbre binaires :

```
data Pair a = Pair a a
bin :: GenericTree Pair Int
bin = Node (Pair (Leaf 1) (Node (Pair bin (Leaf 2))))
```

Sortes :

```
Prelude Main> :kind GenericTree
GenericTree :: (* -> *) -> * -> *
Prelude Main> :kind RoseTree
RoseTree :: * -> *
```

6.3.5 Classes de types

Définition de classes de types

Les classes de types permettent de regrouper les types qui disposent de certaines propriétés. Par exemple on peut définir la classe des types totalement ordonnés, comme tout les types disposant d'une *méthode* **compare** (en fait elle est prédéfinie) :

```
data Ordering = LT | EQ | GT
class Ord a where
  compare :: a -> a -> Ordering
```

Ensuite on peut définir une implémentation de la classe pour un type particulier. On appelle ça une déclaration d'*instances* :

```
instance Ord Bool where
  compare x y =
    if x == y then EQ else if y then GT else LT
```

En suite on peut définir par exemple :

```
x < y = compare x y == LT
```

Les classes sont visibles dans les types, comme des contraintes :

```
(<) :: (Ord a) => a -> a -> Bool
```

Attention : le vocabulaire est le même et les concepts proches mais ce n'est pas la même choses que les classes des langages orientés objets. Notamment contrairement à la programmation orientée objet, les données et les fonctions sont séparées.

On peut imposer plusieurs contraintes. Par exemple [3] :

```
sort_show :: (Ord a, Show a) => [a] -> String
sort_show = (foldl write "") . sort
  where write "" obj = show obj
        write str obj = str ++ "," ++ show obj
```

3. **foldl** correspond à la fonction **List.fold_left** d'Ocaml :

```
*Main> :t foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Méthodes par défaut

En plus des types des méthodes, une classe peut définir des implémentations par défaut. Exemple :

```
class BasicEq3 a where
    isEqual3 :: a -> a -> Bool
    isEqual3 x y = not (isNotEqual3 x y)

    isNotEqual3 :: a -> a -> Bool
    isNotEqual3 x y = not (isEqual3 x y)
```

Dans cet exemple, les instances doivent définir au moins une des deux méthodes.

Surcharge

Les classes de types permettent de faire de la surcharge de façon propre. En Haskell, les opérations arithmétiques sont définies sur les types de la classe **Num** :

```
*Main> :t (+)
(+) :: (Num a) => a -> a -> a
*Main> 3+4
7
*Main> 3.4 + 4.5
7.9
*Main> 3 + 4.3
7.3
```

Contraintes sur les instances et sur les classes

On peut mettre des contraintes sur les instances. Ici on définit un ordre sur les listes d'éléments ordonnés :

```
instance Ord a => Ord [a] where
  compare [] [] = EQ
  compare _ [] = GT
  compare [] _ = LT
  compare (x : xs) (y : ys)
    | comp == EQ = compare xs ys
    | otherwise = comp
    where comp = compare x y
```

Et l'on peut définir des contraintes sur les classes (ici la classe des types bornés) :

```
class Ord a => Bounded a where
  maxBound :: a
  minBound :: a
```

On dit que la classe **Bounded** hérite de **Ord**. Elle héritera notamment des méthodes par défaut. Une classe peut hériter de plusieurs classes.

Ceci permet de définir une *hiérarchie de classes*. Mais attention : un type instance peut redéfinir les méthodes par défaut mais on ne peut pas redéfinir une méthode par défaut en héritant.

Un exemple qui montre que l'on a de la liaison tardive :

```
class CA a where
  f :: a -> Int
  g :: a -> Int
  f x = 4
```

```
  g x = f x

instance CA () where
  f x = 5          -- Je definie la fonction f pour le type ()
                   -- g () renvoie 5

class CA a => CB a where
  f :: a -> Int
  f x = 88         -- NON ! On ne peut pas definir deux fois la fonction f
                   -- pour un meme type
```

On peut définir avec **newtype** un type qui va être égal à un autre mais avec une autre implémentation des méthodes (voir le manuel ou des livres pour plus d'information).

Exemple de classe avec multiparamètres :

```
class Eq e => Collection c e where
     insert :: c -> e -> c
     member :: c -> e -> Bool

 instance Eq a => Collection [a] a where
     insert = flip (:)
     member = flip elem
```

Quelques classes standard

Egalité et ordre

- **Eq** a définit (==), égalité structurelle (! ! !) Haskell n'a pas d'égalité physique (pas évident à cause de la paresse).
- **Ord** a ajoute **compare**, (<), (<=) . . .
- **Bounded** a ajoute **minBound** et **maxBound**

Types numériques

- **Num** a donne un structure d'anneau (+, -, * . . .)
- **Fractional** a ajoute quotient et inverse pour les types dans lesquels c'est toujours défini (comme **Float**).
- **Floating** a ajoute les fonctions usuelles (**exp**, **cos**, . . .)

Un effet amusant : 42 est de type **Num** a **=>**a.

D'autres classes intéressantes

- **Show** a fournit **show :: a -> String**
- **Read** a fournit de quoi décoder une chaîne
- **Enum** a identifie les types indexables par les entiers

Dérivation automatique d'instances

Pour certaines classes standard, une implémentation peut être dérivée automatiquement de la définition du type :

```
data Color = Red | Green | Blue deriving Enum
```

définit un encodage et un décodage naturel de Color dans **Int**.

```
data Term a = Const a
| Var String
| App (Term a) (Term a)
deriving (Eq, Show )
```

produit automatiquement des instances de la forme :

```
instance Eq a => Eq (Term a) where...
instance Show a => Show (Term a) where...
```

La dérivation est implémentée pour les classes suivantes (et seulement celles-ci, en tout cas dans la norme) : **Eq**, **Ord**, **Enum**, **Bounded**, **Show** et **Read**. En général, tous les types que vous définissez devront être instances d'au moins **Eq**, sinon il sera impossible de les comparer avec ==.

Classes pour des types d'autres sortes que *

Pour l'instant les classes que nous avons vues ne concernaient que des types de sorte *. Mais on peut créer des classes pour d'autres sortes. par exemple ici la classe des conteneurs :

```
class Set s where
  empty :: s a
  singleton :: a -> s a
  union :: s a -> s a -> s a
  elem :: a -> s a -> Bool
  map :: (a -> b) -> s a -> s b
  size :: s a -> Int
```

6.4 Programmer en fonctionnel pur paresseux

6.4.1 Paresse

En Haskell, les valeurs ne sont évaluées qu'au besoin (et une seule fois) :

```
*Main> (\x y -> x) 0 (System.Posix.sleep 66) -- c'est rapide
0
*Main> (\x y -> x) (System.Posix.sleep 6) 0 -- c'est long
0
```

En Haskell, toutes les valeurs sont paresseuses, ce qui permet de définir des listes infinies (flux). Voici deux définitions équivalentes de la liste de tous les entiers [4] :

```
ints = f 0 where f i = i:f (succ i)
ints = 0 : map succ ints
```

On peut aussi définir :

```
bottom = bottom             -- !!!
evens = map twice ints      -- entiers pairs
```

Autre façon de définir les entiers pairs :

```
evens = filter even ints
  where even 0 = True
        even 1 = False
        even n = even (n-2)
```

Il est cependant possible de forcer une évaluation en utilisant
- soit l'opération d'application strict `$!` ;
- soit des *constructeurs stricts* (syntaxe : `A !a`).

4. En Ocaml :

```
# let ints = let rec f i = i::f (succ i) in f 0;;
Stack overflow during evaluation (looping recursion?).
```

6.4.2 Monades

Puisque le langage est fonctionnel pur, nous allons utiliser souvent les monades. Haskell définit une classe de type qui s'appelle **Monad** de la façon suivante :

```
infixl 1  >>, >>=       -- priorite des operateurs infixes
class Monad m where
    (>>=)            :: m a -> (a -> m b) -> m b
    (>>)             :: m a -> m b -> m b
    return           :: a -> m a
    fail             :: String -> m a
    m >> k           =  m >>= \_ -> k
```

L'opérateur >> est une sorte de >>= qui ignore le résultat (en fait si le >>= est un **let**, le >> est un point-virgule).

```
Prelude> :t (>>)
(>>) :: (Monad m) => m a -> m b -> m b
```

Les opérateurs >>=, **return** et >> sont surchargés, ce qui permet de les utiliser avec n'importe quelle monade. En fait les monades sont tellement courantes en Haskell que le langage définit du sucre syntaxique qui permet preque de les oublier. Le programme

```
e1 >>= \x ->
e2 >>= \_ ->
e3 >>
e4 >>= \y ->
e5
```

peut s'écrire

```
do x <- e1
   e2
   e3
   y <- e4
   e5
```

ou encore

```
do { x <- e1; e2; e3
     y <- e4; e5 }
```

6.4.3 Entrées/sorties

Dans un langage paresseux, il est difficile de savoir quand ont lieu les calculs. Lorsque l'on fait des entrées sorties, cela pose un problème, puisque l'ordre n'est pas conservé. Les monades vont nous permettre de séquentialiser les entrées/sorties, c'est à dire d'expliciter le fait qu'une entrée ou sortie doit avoir lieu avant une autre. Haskell définit pour ça la *monade d'entrée sortie* (type `IO a`).

Toutes les fonctions qui font des entrées/sorties renvoient une valeur de cette monade :

```
Prelude> :t getChar
getChar :: IO Char
Prelude> :t putChar
putChar :: Char -> IO ()
Prelude> putChar 'a'
aPrelude>
```

La fonction `main` du module `Main` doit avoir le type `IO ()`. Par exemple :

```
main :: IO ()
main =  do c <- getChar
           putChar c
```

Comme d'habitude, lorsque l'on rentre dans la monade, on ne peut pas en sortir, il faut la propager. Donc le code suivant est mauvais :

```
ready :: IO Bool
ready =  do c <- getChar
            c == 'y'  -- Bad!!!
```

Mais celui là est bon :

```
ready :: IO Bool
ready =  do c <- getChar
            return (c == 'y')
```

Autre exemple :

```
getLine :: IO String
getLine =  do c <- getChar
              if c == '\n'
              then return ""
              else do l <- getLine
                      return (c :l)
```

Remarquez que cela ressemble beaucoup à de la programmation impérative.

Toutes les fonctions qui font des entrées/sorties doivent renvoyer une valeur dans la monade. Par exemple, si le type d'une fonction est `Int -> Int -> Int`, alors vous êtes sûrs qu'elle ne fait aucune entrée/sortie.
Encore un exemple :

```
sequence_ :: [IO ()] -> IO ()
sequence_ =  foldr (>>) (return ())
```

Fichiers

La manipulation des fichiers se fait de la même façon. Voici quelques valeurs utiles (du module `IO`) :

```
type FilePath =  String  -- path names in the file system
openFile      :: FilePath -> IOMode -> IO Handle
data IOMode   =  ReadMode | WriteMode | AppendMode | ReadWriteMode
hClose        :: Handle -> IO ()
hGetChar      :: Handle -> IO Char
stdin         :: Handle
```

6.4.4 Exceptions

Erreurs d'entrées/sorties

Si une erreur se produit pendant une entrée/sortie, il faut être capable de l'intercepter. En fait la monade d'entrées/sorties cache une monade d'exception. Les erreurs d'entrées/sorties sont des valeur du type `IOError`. On peut les intercepter par exemple avec l'une des fonctions :

```
catch :: IO a -> (IOError -> IO a) -> IO a
Control.Exception.try :: IO a -> IO (Either GHC.IOBase.Exception a)
Control.Exception.handle :: (GHC.IOBase.Exception -> IO a) -> IO a -> IO a
```

Exemple :

```
getChar' :: IO Char
getChar' =  getChar `catch` eofHandler where
    eofHandler e = if isEofError e then return '\n' else ioError e
```

La fonction `isEofError` teste si l'erreur est la fin de fichier. La fonction **`ioError :: IOError -> IO a`** lance une exception d'entrée/sortie (qui pourra être rattrapée par le prochain **`catch`**).

Autres exceptions

On peut évidemment définir sa propre monade d'exception si l'on définit un nouveau type d'exception. Mais en fait Haskell utilise la plupart du temps les exceptions de la monade **`IO`**. Observons comment est traitée la division (entière) par zéro :

```
Prelude> div 2 0    -- division entiere
*** Exception: divide by zero
Prelude> :t div
div :: (Integral a) => a -> a -> a
```

Le type de **`div`** n'a pas l'air de signaler qu'une exception est possible (pas de monade)... En fait **`div`** n'échoue jamais (il est paresseux, donc le calcul n'a pas lieu !). C'est l'affichage qui échoue !

```
Prelude> let a = 1 `div` 0
Prelude> print a
*** Exception: divide by zero
```

Les exceptions peuvent donc être rattrapées par les fonctions de rattrapage de la monade **`IO`**.

Note : on peut forcer l'exécution avec `Control.Exception.evaluate :: a -> IO a`.

Lever une exception

Utilisez une des deux fonctions :

```
Control.Exception.throw :: GHC.IOBase.Exception -> a
Control.Exception.throwIO :: GHC.IOBase.Exception -> IO a
```

ou bien **`ioError`**.

Définir ses exceptions

Dans les versions récentes de GHC vous pouvez définir vos propres exceptions. Il suffit de définir un type somme qui implémente la classe `IOBase.Exception`, qui est définie dans la bibliothèque standard par :

```
class (Typeable e, Show e) => Exception e where
  toException :: e -> SomeException Source
  fromException :: SomeException -> Maybe e Source
```

Par exemple :

```
data MyException = ThisException | ThatException
     deriving (Show, Typeable)  -- pas besoin d'implementer nous-meme
                                -- Show et Typeable
    -- Typeable est derivable dans les versions recentes de ghc

instance Exception MyException  -- je ne redefinis pas les methodes
```

6.4.5 Monade des listes

On a déjà vu la monade des listes dans le cours sur les monades. Voici des exmples d'utilisation avec Haskell :

```
Prelude> :info []
data [] a = [] | a : [a]        -- Defined in GHC.Base
instance (Eq a) => Eq [a] -- Defined in GHC.Base
instance Monad [] -- Defined in GHC.Base
instance Functor [] -- Defined in GHC.Base
instance (Ord a) => Ord [a] -- Defined in GHC.Base
instance (Read a) => Read [a] -- Defined in GHC.Read
instance (Show a) => Show [a] -- Defined in GHC.Show

Prelude> [1, 2, 3] >>= \x -> [x, x]
[1,1,2,2,3,3]
Prelude> do { x <- [1, 2, 3] ; y <- [100, 200] ; return (x+y) }
[101,201,102,202,103,203]
```

Notez que l'on a :

```
map f lst = lst >>= \x -> [f x]
filter p lst = lst >>= \x -> if p x then [x] else []
```

Comprendre les compréhensions

L'écriture `[x^2| x <- [1..5]]` est en fait du sucre syntaxique pour `do x <- [1..5]; return (x^2)`.

De même `[(x, y)| x <- [1..5], y <- [1,2,3]]` est en fait une autre façon d'écrire `do x <- [1..5]; y <- [1,2,3] ; return (x,y)`.

On peut par exemple définir :

```
concat    :: [[a]] -> [a]
concat xll = [xl <- xll, x <- xl]
```

Dans les compréhensions, on peut aussi exprimer des conditions. Par exemple ici les diviseurs d'un nombre :

```
diviseurs  :: Int -> [Int]
diviseurs n = [x | x <- [1..n], n `mod` x = 0]
```

Encore une fois, c'est juste du sucre syntaxique. Si l'on définit :

```
assert      :: Bool -> [()]
assert True  = [()]
assert False = []
```

On peut écrire

```
diviseurs n = do x <- [1..n]
                 assert (n `mod` x == 0)
                 return x
```

ou encore

```
diviseurs n = do [1..n] >>= \x ->
                 assert (n `mod` x = 0) >>
                 return x
```

6.4.6 Monade d'état

La monade d'état est définie dans les modules `Control.Monad.ST.Strict` et `Control.Monad.ST.Lazy`. Dans la deuxième version, les opérations sur l'état sont retardées jusqu'à ce que l'on en ait vraiment besoin.

La monade est de la forme `ST s a`. Elle retourne une valeur de type `a`, dans un store de type `s`.

Pour sortir de la monade on peut utiliser :

```
runST :: forall a. (forall s. ST s a) -> a
```

L'exemple suivant calcule la fonction de Fibonacci de manière « impérative » et en espace constant (récursive terminale).

```
fibST :: Integer -> Integer
fibST n =
    if n < 2
    then n
    else runST $ do
        x <- newSTRef 0
        y <- newSTRef 1
        fibST' n x y

    where fibST' 0 x _ = readSTRef x
          fibST' n x y = do
              x' <- readSTRef x
              y' <- readSTRef y
              writeSTRef x y'
              writeSTRef y (x'+y')
              fibST' (n-1) x y
```

6.4.7 Composition de monades

Imaginons que l'on veuille avoir à la fois une monade d'état ou d'entrées/sorties et une monade de listes. On peut « ajouter » la monade des listes à une autre monade de la façon suivante :

```
data ListeMon m a = LM (m [a])

instance Monad m => Monad (ListeMon m) where
  return = LM . return . return
  (LM x) >>= g = LM $ do lst <- x
                         lst <- mapM (unLM . g) lst
                         return (concat lst )
    where unLM (LM x ) = x
```

où **`mapM`** est le **`map`** monadique :

```
Prelude> :t mapM
mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
```

Le type `ListeMon` est un *transformeur de monade*.
Exemple de monade transformée : `ListeMon IO`.

6.5 Conclusion

Comparaison avec OCaml

OCaml et Haskell sont des langages très proches, issus de la même communauté. Ce sont deux langages fonctionnels avec typage strict, types produits, sommes avec constructeurs et inférence de type. Ils

ont tous les deux le pattern matching.

On a vu les deux principales différences sont :
- Haskell est purement fonctionnel alors qu'OCaml a des traits impératifs ;
- Haskell est paresseux alors qu'Ocaml est strict.

On a vu que le fonctionnel pur avait un intérêt mais il complique aussi la programmation et a un impact en terme d'efficacité. L'évaluation paresseuse peut parfois rendre la compréhension des programmes difficiles et n'a en pratique pas d'impact sur l'efficacité. Par ailleurs cela demande d'être assez vigilant lorsque l'on veut programmer des programmes « temps réel ». Ces considérations nous montrent qu'un langage purement fonctionnel strict aurait un réel intérêt, mais il n'en existe pas vraiment.

Les systèmes de types se ressemblent également, avec quelques différences :
- Haskell a les classes de types (ce qui introduit du sous-typage structurel). Cela permet de faire de la surcharge très proprement ;
- Haskell a des GADTs (peut-être bientôt dans OCaml ?) ;
- OCaml a des objets et des classes (héritage et liaison tardive) ;
- OCaml a des foncteurs (modules paramétrés) mais les classes de types ont un comportement similaire (voir par exemple la transformation de monade) ;
- OCaml a des variants polymorphes (sous-typage sur les types variants) ;
- OCaml a une syntaxe extensible (et modifiable) grâce à Camlp4/5 ; (par exemple il existe des syntaxes pour les monades et les compréhensions en OCaml).

En terme d'efficacité, il semble que les programmes Haskell soient un peu moins rapide en général mais il existe des pages montrant des techniques (parfois très tordues) pour optimiser les programmes et dans ce cas, un programme Haskell peut être très rapide. En tout cas les deux ont des performances équivalentes (un peu plus lent que C et beaucoup plus rapide que Python ou Ruby).

En terme de communauté, richesse des bibliothèques, et documentation, je pense qu'ils sont à peu près équivalents (c'est-à-dire loin derrière C et Java).

En conclusion : deux langages très expressifs sûrs et rapides, issus tous les deux d'une recherche poussée sur les langages de programmation...

Pour aller plus loin

Les amateurs de sensations fortes pourront se pencher sur les sujets suivants :
- Classes de types multi-paramètres..
  - Pour définir des relations entre types
  - Dépendances fonctionnelles...
- Plus fort que les monades : les arrows
  - Functional Reactive Programmming
  - Circuits...
- Foncteurs et théorie des catégories
- Questions d'implémentation
  - Classes de types et programmation objet
  - Évaluation optimiste...
- Template Haskell