Projet Réseaux de Kahn

Le but du projet est de réaliser différentes implantations de réseaux de Kahn [2, 3]. Chacune des implantations devra respecter l'interface suivante :

```
module type S = sig
  type 'a process
  type 'a in_port
  type 'a out_port

  val new_channel: unit -> 'a in_port * 'a out_port
  val put: 'a -> 'a out_port -> unit process
  val get: 'a in_port -> 'a process

  val doco: unit process list -> unit process

  val return: 'a -> 'a process
  val bind: 'a process -> ('a -> 'b process) -> 'b process

  val run: 'a process -> 'a
end
```

Cette interface est donnée sous forme monadique et va permettre ainsi de faire des implantations séquentielles et parallèles [1].

Le type `'a process` représente un calcul qui peut prendre du temps (peut être interrompu) et qui retourne un valeur de type `'a`.

Le type `'a in_port` représente la partie d'un canal de communication dans laquelle on peut lire des valeurs et le type `'a out_port` représente la partie dans laquelle on peut écrire.

La fonction `new_channel` crée un nouveau canal. Les fonctions `put` et `get` permettent respectivement d'envoyer une valeur dans un canal et de récupérer une valeur dans un canal. Ces fonctions sont des processus car cela permettra de modéliser des entrées/sorties bloquantes.

La fonction `doco` prend en paramètre une liste de processus et crée un processus qui les exécute en parallèle. L'exécution du processus termine lorsque tous les processus qu'il a lancé ont terminé leur exécution.

La fonction `return` crée un processus qui termine instantanément en retournant la valeur donnée en paramètre.

La fonction `bind` crée un processus qui évalue le processus donnée en premier argument puis exécute le processus donnée en second argument en lui transmettant la valeur calculée par le premier.

Enfin, `run` exécute un processus et retourne la valeur calculée par celui-ci.

Illustrons l'utilisation de cette interface avec la programmation des processus suivants :

- `integers` qui génère sur un canal la séquence des entiers naturels à partir de deux ;
- `output` qui affiche les valeurs qu'il reçoit sur un canal ;
- `main` qui exécute en parallèle une instance de `integers` et `output` qui communiquent par un canal.

```
module Lib (K : S) = struct
  let ( >>= ) x f = K.bind x f

  let delay f x =
    (K.return ()) >>= (fun () -> K.return (f x))
end

module Example (K : Kahn.S) = struct
  module Lib = Kahn.Lib(K)
  open Lib

  let integers (qo : int K.out_port) : unit K.process =
    let rec loop n =
      (K.put n qo) >>= (fun () -> loop (n + 1))
    in
    loop 2

  let output (qi : int K.in_port) : unit K.process =
    let rec loop () =
      (K.get qi) >>= (fun v -> Format.printf "%d@." v; loop ())
    in
    loop ()

  let main : unit K.process =
    (delay K.new_channel ()) >>=
    (fun (q_in, q_out) -> K.doco [ integers q_out ; output q_in ; ])
end
```

Ces deux modules sont paramétrés par un module `K` qui doit respecter l'interface précédente. Cela permet de les exécuter simplement avec différentes implantations.

Le module `Lib` contient des définition utilitaire : une notation infixe pour l'opérateur `bind` et une fonction `delay` qui permet de transformer une fonction en un processus.

Exemple d'implantation

Pour préciser le comportement souhaité de l'interface de programmation de réseaux de Kahn, nous vous en donnons une implantation naïve qui utilise la bibliothèque de threads d'OCaml.

```
module Th: S = struct
  type 'a process = (unit -> 'a)

  type 'a channel = { q: 'a Queue.t ; m: Mutex.t; }
  type 'a in_port = 'a channel
  type 'a out_port = 'a channel

  let new_channel () =
    let q = { q = Queue.create (); m = Mutex.create (); } in
    q, q

  let put v c () =
    Mutex.lock c.m;
    Queue.push v c.q;
    Mutex.unlock c.m;
    Thread.yield ()

  let rec get c () =
    try
      Mutex.lock c.m;
      let v = Queue.pop c.q in
      Mutex.unlock c.m;
      v
    with Queue.Empty ->
      Mutex.unlock c.m;
      Thread.yield ();
      get c ()

  let doco l () =
    let ths = List.map (fun f -> Thread.create f ()) l in
    List.iter (fun th -> Thread.join th) ths

  let return v = (fun () -> v)

  let bind e e' () =
    let v = e () in
    Thread.yield ();
    e' v ()

  let run e = e ()
end
```

Dans cette implantation, nous représentons le processus par des fonctions de type `unit -> 'a`. L'utilisation de fonctions permet de contrôler le moment où les calculs doivent être effectuer. On peut ainsi constater dans l'implantation de `bind` que l'évaluation du premier argument `e` est effectué uniquement lorsque la fonction est complètement appliquée.

Travail demandé

Vous devez réaliser plusieurs implantations respectant l'interface `S`. Par celles-ci vous devrez en particulier en réaliser une reposant sur l'utilisation de processus Unix communiquant par des tubes, une version s'exécutant à travers le réseau et une version séquentielle ou le parallélisme est simulé par votre programme.

Dans l'implantation séquentielle, les processus pourrons par exemple avoir le type suivant :

```
type 'a process = ('a -> unit) -> unit
```

C'est-à-dire ce sont des fonctions qui prennent en argument leur continuation (le calcule à effectué une fois l'exécution du processus terminé).

Vous pouvez réaliser des implantations qui mélange l'utilisation de processus et de threads, qui fonctionnent avec un nombre borné de processus et/ou de threads, etc.

Vous pouvez partir des fichiers disponibles à l'adresse :

`https://www.lri.fr/~mandel/systeme-ens/projet.tgz`

Modalité de rendu

Le projet est à faire en binôme. Il doit être remis par email à `louis.mandel@lri.fr` et `marc.pouzet@ens.fr`. Votre projet doit se présenter sous forme d'une archive `tar` compressée (option `z` de `tar`), appelée `vos_noms.tgz` qui doit contenir un répertoire appelé `vos_noms` (exemple : `dupont-durand.tgz`). Dans ce répertoire doivent se trouver les sources de votre programme (inutile d'inclure les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit compiler votre programme. La commande `make clean` doit effacer tous les fichiers que `make` a engendrés et ne laisser dans le répertoire que les fichiers sources. L'archive doit également contenir un court rapport expliquant les différents choix techniques qui ont été faits et, le cas échéant, les difficultés rencontrées ou les éléments non réalisés. Ce rapport pourra être fourni dans un format ASCII, PostScript ou PDF.

Références

[1] Koen Claessen. A poor man's concurrency monad. *J. Funct. Program.*, 9(3):313–323, May 1999.

[2] Gilles Kahn. The semantics of simple language for parallel programming. In *Proceedings of IFIP 74 Conference*, pages 471–475, 1974.

[3] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pages 993–998, 1977.