Continuations, coroutines et *Green Threads*

Objectifs

- Utilisation des continuations natives.
- Application aux *Green Threads*.
- Application aux canaux de communication asynchrones (*channels*).

On utilisera le fichier archive `sourceEtu.tar` fourni, contenant des éléments permettant la réalisation des exercices de la séance.

1 Continuations natives

Les continuations natives sont disponibles dans la librairie `Delimcc`, comme vu en TD.

On rappelle les types et primitives nécessaires de `Delimcc` :

```
(* le type des marqueurs de pile *)
type 'reset prompt;;

val new_prompt   : unit -> 'reset prompt
val push_prompt  : 'reset prompt -> (unit -> 'reset) -> 'reset
val shift : 'reset prompt -> (('shift -> 'reset) -> 'reset) -> 'shift
```

2 Programmation concurrente et *Green Threads*

Une possibilité intéressante en programmation est de permettre la création et manipulation de *threads*, i.e. de processus légers, au niveau utilisateur, comme une librairie standard, sans faire (trop) appel aux couches systèmes. Cette solution évite les appels systèmes de gestion de processus, réputés lourds, et rend de fait la notion de processus relativement indépendante du système. Ceci est exact car la copie de parties de la pile est réalisable simplement, en assembleur ou à l'aide de primitives systèmes simples.

Ces processus utilisateurs, plus économes en ressources système, sont appelés *green threads*. Ils s'apparentent aux coroutines car ils sont la plupart du temps non-préemptifs (coopératifs) et peuvent être implantés grâce aux continuations.

Concrètement, nous allons implanter un *scheduler* de processus. Ces processus possèdent trois commandes spécifiques :

- `yield ()` qui rend la main au *scheduler* pour qu'il choisisse un autre processus à exécuter dans la queue des processus inactifs ;
- `fork p` qui rend la main également pour démarrer l'exécution du processus `p`.
- `exit ()` qui termine l'exécution du processus. Le *scheduler* choisit un autre processus à exécuter.

On utilisera pour cela le module `Queue` de la librairie standard qui implante des files impératives et dont voici un extrait utile de l'interface :

```
type 'a t
val is_empty : 'a t -> bool
```

```
val create : unit -> 'a t
val pop : 'a t -> 'a
val push : 'a -> 'a t -> unit
```

▷ **Exercice 1**
On définit les processus comme étant de type `unit -> unit`*. Définir le type de retour* `res` *correspondant aux différentes façons pour un processus de rendre la main au* scheduler*. Définir les opérations suivantes :*

```
yield : unit -> unit
fork : (unit -> unit) -> unit
exit : unit -> unit
```

▷ **Exercice 2**
Définir la fonction `scheduler : (unit -> unit) -> unit` *qui démarre la gestion des* Green Threads *avec un processus initial unique. Il devra gérer un unique processus actif et une file de processus en attente. Chaque processus devra être lancé dans un contexte où l'on traitera les différentes façons de rendre la main, afin de changer en conséquence le processus actif, la file des processus en attente, etc.*

▷ **Exercice 3 (Application : ping-pong)**

1. *Définir un processus* `ping : unit -> unit` *qui affiche* 10 *fois "ping !" en rendant la main après chaque affichage. Définir de même* `pong`*.*
2. *Définir un processus principal* `ping_pong : unit -> unit` *qui lance* `ping` *et* `pong` *avant de terminer. Exécuter* `ping_pong` *dans le* scheduler*. Que constatez-vous ?*

3 Extension : les canaux asynchrones (*Channels*)

Les canaux de communication asynchrone, ou *channels*, peuvent être définis dans le modèle des *Green Threads*. Ils respectent l'interface suivante. La création d'un canal correspond à la donnée de deux fonctions :

— l'écriture, de type `'a -> unit`, non bloquante.
— la lecture, de type `unit -> 'a`, bloquante. Elle doit rendre la main si le canal est vide.

```
module type Channel =
  sig
    val create : unit -> ('a -> unit) * (unit -> 'a)
  end
```

▷ **Exercice 4 (Canaux de commmunication asynchrones)**
Définir un module réalisant l'interface `Channel`*, en utilisant les* green threads*.*

L'application fournie `sieve : unit -> unit` implante un crible au moyen d'une chaîne de processus reliés par des canaux de communication, chaîne de plus en plus longue au fur et à mesure que l'on découvre de nouveaux nombres premiers. Les chaînons sont des processus filtres qui conservent leur première valeur reçue (un nombre premier), et envoie les suivantes au prochain maillon, qu'il a lui-même créé. Au début de la chaîne, un processus produit tous les nombres entiers successifs jusqu'à un certain rang.

▷ **Exercice 5 (Application : crible)**
Tester votre implantation en exécutant le programme `sieve`*. On doit observer l'affichage de tous les nombres premiers de* 2 *à* 1000*, puis une terminaison normale.*