Réseaux de Kahn

Projet de systèmes

Naïm Favier

17 mai 2020

Compilation

J'ai fait ce projet en Haskell. Il dépend de GHC (au moins 6.8.1) et des librairies `unix` et `network`.

Pour compiler, lancer `make`, après avoir choisi l'implémentation et le processus souhaités dans `Main.hs`. Les processus fournis sont :

- `printIntegers` : affiche les entiers à partir de 2 en utilisant un seul canal ;
- `primes` : implémentation du crible d'Ératosthène donnée dans le papier *Coroutines and networks of parallel processes* de Gilles Kahn ;
- `pingpong` : deux processus s'éxécutent en parallèle en utilisant un canal pour se synchroniser.

L'exécutable généré est `Kahn`.

Implémentation

Le module `Kahn` définit une *typeclass* qui fournit la signature des réseaux de Kahn donnée dans l'énoncé, à quelques différences près :

- `newChannel` renvoie un processus ;
- `put` (resp. `get`) a la contrainte `Show` (resp. `Read`) sur le type `a` : cela permet de (dé)sérialiser facilement ;
- les arguments de `put` sont inversés ;
- `run` renvoie une action `IO` (c'est implicite en OCaml).

`Kahn` est une sous-classe de `MonadIO`, qui fournit `return`, `(>>=)` et `liftIO` (cette dernière permet de transformer des actions IO en processus, ce qui est fait implicitement en OCaml).

Les implémentations fournies sont les suivantes :

`KahnThreads`

Cette implémentation utilise des threads (il s'agit de *green threads* propres à la runtime de Haskell, qui ne correspondent pas forcément aux threads de l'OS).

Les processus ont le type `IO a`.

Les canaux sont ceux fournis par le module `Control.Concurrent.Chan`.

`KahnProcesses`

Cette implémentation utilise des processus (Unix) communiquant entre eux par des tubes.

Les processus ont le type `IO a`.

La sérialisation des données se fait de la manière la plus naïve possible : `put c a` écrit la représentation en chaîne de caractères de `a` sur le tube `c`, terminée par un caractère nul ; `get c` lit des caractères depuis `c` jusqu'à rencontrer un caractère nul, et transforme la chaîne obtenue en le type souhaité.

`KahnSockets`

Cette implémentation utilise à nouveau des threads, mais les canaux sont implémentés par des *sockets* TCP connectées à `localhost` sur un port aléatoire.

Les processus ont le type `IO a`.

La sérialisation se fait de la même manière que pour les tubes.

`KahnSequential`

Dans cette implémentation, le parallélisme est simulé par le programme.

Les processus ont le type `Process a`, où `Process` est une implémentation de la **monade libre** par rapport au foncteur `IO` :

```haskell
data Process a = Pure a | Atom (IO (Process a))
```

La monade libre est la monade la plus simple qu'on puisse définir sur un foncteur, c'est à dire qu'elle se contente de vérifier les lois monadiques et ne fait rien d'autre. Par analogie, le monoïde libre sur un ensemble A est le monoïde le plus simple possible : c'est l'ensemble des listes d'éléments de A muni de la concaténation de listes et de la liste vide comme élément neutre. Ainsi, on peut traduire l'expression $1 + 2 + 3 = 6$ dans sa version "libre" $[1] + [2] + [3] = [1, 2, 3]$, ce qui permet de garder une trace du calcul effectué, et pas seulement son résultat.

Cette analogie est particulièrement bien fondée puisqu'une monade est un monoïde particulier (dans la catégorie monoïdale des endofoncteurs munie de la composition) : la monade libre `Process` permet de garder une trace des actions `IO` à effectuer, ce qui permet d'intercaler les actions de plusieurs processus. C'est ce que fait `doco` : étant donnée une liste de processus, on exécute chaque action atomique en "tête de liste", ce qui fournit une nouvelle liste de processus à exécuter récursivement. L'équivalent sur les listes est une sorte d'opération d'intercalage : `doco [[1, 2, 3], [4, 5]] = [1, 4, 2, 5, 3]`.

La fonction `liftIO` correspond à l'opération $x \mapsto [x]$ et permet de transformer une action `IO` en un processus qui exécute cette action de manière atomique.

La fonction `run` n'a plus qu'à exécuter la liste d'actions fournie par le processus, cette fois dans la monade `IO`.

Les canaux sont implémentés par le type `IORef (Queue a)`, où `Queue` est l'implémentation classique des files utilisant deux listes, et `IORef` fournit une référence mutable vers un objet manipulable dans `IO`. Ainsi, `put c a` est une action atomique qui modifie la file `c` pour y ajouter `a` ; `get c` essaye de lire depuis `c` (de manière atomique), et réessaye tant que `c` est vide.

Commentaires

Les implémentations fournies sont très simples et généralement peu efficaces.

La sérialisation pour les implémentations par tubes et par *sockets* pourrait être améliorée en utilisant une "vraie" bibliothèque de sérialisation, mais j'ai voulu garder les choses simples et les dépendances minimales.

Pour l'implémentation "en réseau", on aurait pu envisager d'utiliser la bibliothèque Cloud Haskell pour exécuter du code à distance.

Enfin, l'implémentation séquentielle est extrèmement inefficace : l'exécution de `doco [a, doco [b, c]]` accorde autant de "temps" à `a` qu'aux deux processus `b` et `c`, donc plus on imbrique d'appels à `doco`, moins les processus les plus imbriqués ont de temps pour s'exécuter. La solution proposée par Koen Claessen dans *A poor man's concurrency monad* semble éviter ce problème, mais, encore une fois, j'ai voulu garder les choses simples.