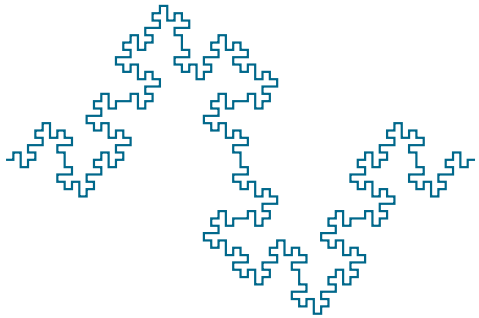
HLIN403 – Programmation Applicative

Programmation par flux de données (dataflow), suite

Christophe Dony – Annie Chateau
Université Montpellier – Faculté des Sciences



Dataflow, implémentation

De nombreux langages existent pour implémenter le modèle dataflow.

Exemple de framework récent, open-source et facile à intégrer dans Eclipse : Orcc

http ://orcc.sourceforge.net/

Dans ce cours, nous allons approcher le paradigme dataflow en scheme à l'aide de la structure de données `stream`...

Opérations génériques sur les listes

Les suites servent d'interfaces standard pour combiner les modules d'un programme.

On a vu des abstractions puissantes pour manipuler les suites, comme `map`, `filter` et `accumulate`. C'est une façon élégante de programmer.

Mais si on représente les suites comme des listes, cette élégance coûte cher en efficacité (en temps et en espace), car les programmes doivent construire et copier des structures de données potentiellement grandes, à chaque étape.

Exemple d'inefficacité des listes pour implémenter les suites

On considère les deux programmes suivants pour calculer la somme des entiers premiers dans un intervalle d'entiers $[a, b]$:

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count) (iter (+ count 1) (+ count accum)))
          (else (iter (+ count 1) accum))))
  (iter a 0))
```

```
(define (sum-primes a b)
  (accumulate +
              0
              (filter prime? (enumerate-interval a b))))
```

Exemple d'inefficacité des listes pour implémenter les suites

Le premier programme n'a besoin de stocker que la somme en train d'être calculée.

Dans le deuxième programme, le filtre ne peut pas faire de test tant que `enumerate-interval` n'a pas terminé de construire une liste complète des nombres dans l'intervalle. La fonction `filter` génère une autre liste, qui est à son tour traitée pour réaliser la somme.

Même problème si on veut le deuxième entier premier dans l'intervalle $[10000, 1000000]$.

```
(car (cdr (filter prime?
                  (enumerate-interval 10000 1000000))))
```

Exemple d'inefficacité des listes pour implémenter les suites

Cette expression trouve le deuxième nombre premier, mais le temps est déraisonnable. On construit une liste de presque un million d'entiers, on la filtre en testant la primalité de chaque élément, puis on ignore presque l'intégralité du résultat.

Dans un style de programmation plus traditionnel, on intercalerait l'énumération et le filtrage, en s'arrêtant dès le deuxième nombre premier.

La structure de données `stream` (= flot)

Les flots permettent de manipuler des suites d'éléments sans le coût inhérent à leur implémentation sous forme de listes.

Le meilleur des deux mondes : programmation élégante et efficacité.

Principe : on construit le flot seulement de façon partielle, en fonction du besoin du programme qui le consomme.

Interface des `stream`

En surface, les flots sont justes des listes, avec des fonctions de manipulation ayant des noms différents.

`cons-stream` (constructeur)
`stream-car` et `stream-cdr` (accesseurs)
`stream-null?`

Interface des stream

```
(define (stream-ref s n)
  (if (= n 0)
      (stream-car s)
      (stream-ref (stream-cdr s) (- n 1))))

(define (stream-map proc s)
  (if (stream-null? s)
      the-empty-stream
      (cons-stream (proc (stream-car s))
                   (stream-map proc (stream-cdr s)))))

(define (stream-for-each proc s)
  (if (stream-null? s)
      'done
      (begin (proc (stream-car s))
             (stream-for-each proc (stream-cdr s)))))
```

Interface des `stream`

Stream-for-each est utile pour visualiser les flots :

```
(define (display-stream s)
  (stream-for-each display-line s))

(define (display-line x)
  (newline)
  (display x))
```

Implémentation de l'interface

L'implémentation est basée sur une forme spéciale appelée `delay`. L'évaluation de `(delay <exp>)` n'évalue pas l'expression `<exp>`, mais renvoie un objet « différé » qui promet d'évaluer l'expression plus tard.

La procédure `force` prend un objet différé et réalise l'évaluation.

```
(cons-stream <a> <b>)
```

est équivalent à

```
(cons <a> (delay <b>))
```

```
(define (stream-car stream) (car stream))
```

```
(define (stream-cdr stream) (force (cdr stream)))
```

Retour à l'exemple

```
(stream-car
 (stream-cdr
  (stream-filter prime?
                 (stream-enumerate-interval 10000 1000000))))
```

On appelle `stream-enumerate-interval` sur les arguments 10,000 et 1,000,000.

```
(define (stream-enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream
       low
       (stream-enumerate-interval (+ low 1) high))))
```

Retour à l'exemple

Le résultat renvoyé sur cet appel par `stream-enumerate-interval` est

```
(cons 10000
      (delay (stream-enumerate-interval 10001 1000000)))
```

```
(define (stream-filter pred stream)
  (cond ((stream-null? stream) the-empty-stream)
        ((pred (stream-car stream))
         (cons-stream (stream-car stream)
                      (stream-filter pred
                                     (stream-cdr stream))))
        (else (stream-filter pred (stream-cdr stream)))))
```

La procédure `stream-filter` teste le `stream-car` du flot qui est 10,000.

Retour à l'exemple

Comme 10000 n'est pas premier, `stream-filter` examine le `stream-cdr` de son flot d'entrée. Cela force l'évaluation du `stream-enumerate-interval` différé, qui renvoie :

```
(cons 10001
      (delay (stream-enumerate-interval 10002 1000000)))
```

Et ainsi de suite jusque 10007 qui est premier.

Retour à l'exemple

`stream-filter` renvoie alors

```
(cons-stream (stream-car stream)
             (stream-filter pred (stream-cdr stream)))
```

C'est-à-dire

```
(cons 10007
      (delay
        (stream-filter
         prime?
         (cons 10008
               (delay
                 (stream-enumerate-interval 10009
                                            1000000))))))
```

Retour à l'exemple

Ce résultat est passé à `stream-cdr` dans l'expression originelle. Cela force le `stream-filter` différé, qui à son tour force le `stream-enumerate-interval` différé jusqu'à ce qu'il trouve le nombre premier suivant, 10009. Finalement, le résultat passé au `stream-car` dans l'expression orifginelle est :

```
(cons 10009
      (delay
        (stream-filter
         prime?
         (cons 10010
               (delay
                 (stream-enumerate-interval 10011
                                            1000000))))))
```

Retour à l'exemple

`stream-car` renvoie 10009 et le calcul est terminé. On n'a testé qu'autant d'entiers qu'il fallait, et l'intervalle n'a pas été énuméré plus loin.

Implémentation de `delay` et `force`

`(delay <exp>)`

est un sucre syntactique pour

`(lambda () <exp>)`

`force` appelle simplement la procédure sans argument produite par `delay`

```
(define (force delayed-object)
  (delayed-object))
```