INFO0054-1
Programmation fonctionnelle

LIÈGE université

Chapitre 15 : Tabulation et Mémoïsation

Christophe Debruyne

(c.debruyne@uliege.be)

Cette présentation est basée sur les transparents de Prof. Dr. Em. Pascal Gribomont.

Tables et tabulation

LIÈGE université

Idée : ne pas recalculer un élément déjà calculé.

Exemple : rendre efficace le programme naïf `fib`
et même la version avec accumulateur

Stratégie : maintenir une table de valeurs calculées.

Une table est une liste de paires pointées `(variable . valeur)`.

```
(define (fib n)
  (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))

(define (fib2 n)
  (fib2-a n 0 1))
(define (fib2-a n a b)
  (if (= n 0) a (fib2-a (- n 1) b (+ a b))))
```

Tables et tabulation

```
(define *TABLE-fib* '((1 . 1) (0 . 0)))

(define m-fib
  (lambda (n)
    (let ((v (assv n *TABLE-fib*)))
    (if v
        (cdr v)
        (let ((a (m-fib (- n 1)))
              (b (m-fib (- n 2))))
          (let ((val (+ a b)))
            (set! *TABLE-fib* (cons (cons n val) *TABLE-fib*))
            val))))))
```

`assv` localise le premier élément d'une liste dont le `car` est égal à `n` selon `eqv?`. Si un tel élément existe, la paire est renvoyée. Sinon, le résultat est `#f`. (documentation)

Nous utilison un `let` pour `val` car nous utilisons `val` deux fois.

Tables et tabulation

```
> (time (m-fib 50000))
cpu time: 640 real time: 650 gc time: 578
1077773489307297...

> (time (fib2 50000))
cpu time: 62 real time: 69 gc time: 31
1077773489307297...

> (time (m-fib 50000))
cpu time: 0 real time: 0 gc time: 0
1077773489307297...

> (time (m-fib 40000))
cpu time: 0 real time: 0 gc time: 0
1432600165457807...
```

`m-fib` a besoin plus de temps pour calculer le résultats. Ceci est du au instructions pour manipuler la table, mais (!) un fois le résultat et les résultats intermédiaires sont stockés dans le table, `m-fib` est plus performantes pour les appels postérieure.

Problèmes :

- Version tabulée à écrire pour chaque fonction.
- Introduction d'une variable globale.

Tables et tabulation


```
(define lookup
  (lambda (obj table succ-p fail-p)
    (if (null? table)
        (fail-p)
        (let ((pr (car table)))
          (if (equal? (car pr) obj)
              (succ-p pr)
              (lookup obj (cdr table) succ-p fail-p))))))

;; succ-p est une fonction que nous appliquons
;; à l'objet que nous avons trouvé
;; fail-p est une fonction que nous appliquons sans
;; arguments quand nous avons pas trouvé l'objet dans
;; le table
```

Tables et tabulation

```
(define memoize
  (lambda (proc)
    (let ((table '()))
      (lambda (arg)
        (lookup
          arg
          table
          cdr
          (lambda ()
            (let ((val (proc arg)))
              (set! table (cons (cons arg val) table))
              val)))))))
```

`memoize` renvoie un lambda prenant un argument. De lambda est lié as son propre table dans le "télescopage".

Si on trouve l'élément dans le table, on applique le `cdr` à cet élément. Sinon on calcule application de proc à l'argument arg et on le stocke dans le table.

Tables et tabulation

```
(define m-fib1 (memoize fib))

> (time (m-fib1 40))        cpu time: 8671 real time: 8737 gc time: 968
102334155
> (time (m-fib1 41))        cpu time: 13437 real time: 13495 gc time: 1015
165580141
> (time (m-fib1 40))        cpu time: 0 real time: 0 gc time: 0
102334155
> (time (m-fib1 39))        cpu time: 5046 real time: 5148 gc time: 171
63245986
> (time (m-fib1 41))        cpu time: 0 real time: 0 gc time: 0
165580141
```

Attention à l'approche naïve ! Seul le premier appel est stocké dans le table !

Il faut donc redéfinir les fonctions en fonction de la version mémorisée.

Tables et tabulation

```
(define m-fib2
  (memoize
    (lambda (n)
      (if (< n 2) n (+ (m-fib2 (- n 1)) (m-fib2 (- n 2))) ))))
```

```
> (time (m-fib1 50000))     cpu time: 140 real time: 149 gc time: 62
10777734893...
> (time (m-fib1 50001))     cpu time: 0 real time: 4 gc time: 0
17438741378...
> (time (m-fib1 50000))     cpu time: 0 real time: 0 gc time: 0
10777734893...
> (time (m-fib1 49999))     cpu time: 0 real time: 0 gc time: 0
66610064856...
> (time (m-fib1 50200))     cpu time: 703 real time: 676 gc time: 140
67616945271...
```

Pratiquer la mémoïsation

Avec la mémoïsation, nous n'avons plus de la programmation fonctionnelle pur car nous changeons l'état (i.e., nous avons des effets de bord).

La mémoïsation et néanmoins souvent utilisée pour rendre certains processus plus efficace. Une bonne pratique est de créer une solution "pur" et puis introduire, où possible, cette approche.

Dans le syllabus de Prof. Gribomont :

- L'utilisation de vecteurs au lieux de listes, mais cela nécessite prévoir un nombre de places dans un vecteur (et la gestion des vecteurs)
- La représentation de fonctions prenant plusieurs arguments
- …

Pratiquer la mémoïsation

Racket fournit un bibliothèque pour la mémoïsation :

```
(require memoize)

(define/memo (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Ref: https://docs.racket-lang.org/memoize/index.html

Pratiquer la mémoïsation

En Python, par exemple, nous avons des "décorateurs" pour "emballer" nos fonctions avec la mémoïsation.

```
@cache
def factorial(n):
    return n * factorial(n-1) if n else 1
```

"Simple lightweight unbounded function cache. Sometimes called "memorize". Returns the same as lru_cache(maxsize=None), creating a thin wrapper around a dictionary lookup for the function arguments. "

(https://docs.python.org/3/library/functools.html)