Sur le problème de Hamming, l'infini, la paresse, et l'expressivité d'un langage de programmation

Jean-Paul Roy
Département Informatique
Faculté des Sciences de Nice Sophia-Antipolis
`http://deptinfo.unice.fr/~roy`
roy@unice.fr

Le but de cet article n'est pas de trouver un algorithme pour résoudre un problème mathématique mais d'étudier dans le cas particulier du problème de Hamming la *distance* entre la caractérisation mathématique de la solution et sa programmation effective, distance qui mesure le *degré d'expressivité du langage* pour ce problème.

L'ensemble de Hamming **H** est constitué des entiers naturels n'ayant pas d'autres facteurs premiers que 2, 3 et 5. Il s'agit du groupe multiplicatif des entiers dont la décomposition en facteurs premiers est de la forme $2^a3^b5^c$ avec $a,b,c \geq 0$. Les premiers éléments de **H** sont donc :

$$H = \{1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36,...\}$$

L'ensemble **H** est *infini*, mais fixons-nous comme objectif d'en calculer le 20000-ème élément. Il s'agit donc d'être capable d'énumérer un à un les éléments de **H**. Ne nous éloignons pas des mathématiques sans une très bonne raison, et résistons à la tentation de nous munir d'une boucle *while* pour faire je ne sais quoi… La caractérisation mathématique de **H** est en effet très simple, en notant kH l'ensemble des produits kh lorsque h décrit H :

$$H = \{1\} \cup 2H \cup 3H \cup 5H \qquad (*)$$

Ayant très envie de coller à cette élégante caractérisation de l'ensemble **H**, je vais utiliser le langage algorithmique Scheme (`www.plt-scheme.org`) pour procéder à mon calcul. J'ai choisi Scheme (prononcez *skime*) car il s'agit de mon langage préféré, mais vous serez ravi de faire la même chose avec votre langage, chacun de nous disposant comme on sait du meilleur langage qui soit, pouvant exprimer les calculs le plus simplement du monde…

Les listes en Scheme

Le langage Scheme utilise des listes *(a b c d)* lorsque les mathématiciens utilisent des ensembles *{a,b,c,d}*. La différence est que la liste *(d c b a)* n'est pas la même que la liste *(a b c d)* alors que les ensembles *{d,c,b,a}* et *{a,b,c,d}* sont les mêmes, mais à la limite tant mieux car pour notre problème l'ordre est important. L'axiome du choix de la théorie des ensembles s'exprime dans l'univers des listes par la possibilité d'extraire le premier élément d'une liste *L* avec la fonction *(first L)*, le reste de la liste étant noté *(rest L)*. Nous aurez remarqué la notation bizarre *(f x (g y z))* utilisée par Scheme à la suite de Lisp, au lieu du traditionnel *f(x,g(y,z))* des maths. Il y a d'excellentes raisons pour cela (les expressions sont des arbres), mais peu importe : ce n'est que de la notation, n'est-ce pas ? La fonction *(cons x L)* permet de construire la liste obtenue en ajoutant x en tête de la liste *L*. Par exemple, si *L* est la liste *(5 2 4)*, alors *(cons 1 L)* est la liste *(1 5 2 4)*. Les axiomes reliant ces fonctions sont donc :

```
(first (cons x L)) ≡ x
(rest (cons x L)) ≡ L
(cons (first L) (rest L)) ≡ L
```

Voici par exemple la fonction permettant d'obtenir la liste des carrés de *[a,b]* avec $a \leq b$, par récurrence montante sur l'entier *a* :

```
(define (carrés a b)      ; liste des carrés de [a,b] par récurrence sur a
  (if (> a b)
      empty               ; la constante empty dénote la liste vide ()
      (cons (* a a) (carrés (+ a 1) b))))
```

```
> (carrés 5 10)           ; le langage est interactif, je teste ma fonction
(25 36 49 64 81 100)
```

Si j'ai besoin de calculer la liste *kL* des multiples d'une liste *L*, il me suffit de programmer une fonction *(zoom k L)* :

```
(define (zoom k L)        ; la liste des kx lorsque x parcourt L
  (if (empty? L)
      L
      (cons (* k (first L)) (zoom k (rest L)))))       ; par récurrence sur L

> (zoom 2 (list 1 5 2 4))
(2 10 4 8)
```

Mais ici H est infini !?

Oui, je sais, Scheme n'a pas prévu les listes infinies. Et pour cause, la mémoire de l'ordinateur est finie ! Mais empli d'espérance je fais comme si de rien n'était… Je ne pourrai pas utiliser *first*, *rest* et *cons* puisqu'elles traitent de listes finies. Je suppose hardiment que je dispose d'analogues *sfirst*, *srest* et *scons* fonctionnant dans cet univers éthéré des listes infinies. La définition de **H** qu'il me plairait d'écrire est bien entendu la suivante, traduisant fidèlement la caractérisation mathématique (*) :

```
(define H (scons 1 (mélanger (szoom 2 H) (mélanger (szoom 3 H) (szoom 5 H)))))
```

La fonction *(szoom k L)* fournit le liste infinie des produits par *k* des éléments de la liste infinie *L*, et sa rédaction est immédiate :

```
(define (szoom k L)        ; la liste des kx lorsque x parcourt la liste infinie L
  (scons (* k (sfirst L)) (szoom k (srest L)))))
```

Le lecteur ayant des lettres s'étonnera d'une récurrence sans cas de base. Il s'apercevra heureusement un peu plus loin qu'il avait tort de s'inquiéter, wait and see… Quant à la fonction *(mélanger L1 L2)*, elle est censée fusionner deux listes infinies croissantes *L1* et *L2* en une liste infinie croissante, et ne pose d'autre problème que la peur innée de l'infini :

```
(define (mélanger L1 L2)              ; L1 et L2 infinies croissantes
  (if (< (sfirst L1) (sfirst L2))
      (scons (sfirst L1) (mélanger (srest L1) L2))
      (scons (sfirst L2) (mélanger L1 (srest L2)))))
```

Et bien voilà, nous tenons notre programme, modulo… l'existence des listes infinies, sur lesquelles nous allons devoir nous pencher ! Un mathématicien avait passé une partie de sa vie à démontrer de profonds théorèmes sur les corps finis non commutatifs jusqu'à ce beau jour de 1905 où Wedderburn a réussi à prouver que tout corps fini était commutatif. Cela fait froid dans le dos, non ? Courage, fuyons en avant !

Des listes finies, infinies ou alors quoi[1] ?

Il est clair que l'infini sera potentiel et non actuel. Ne pouvant stocker une infinité de données dans la mémoire de l'ordinateur, je me contenterai de l'infini au sens suivant : je dois être capable d'extraire le k-ème élément d'une liste infinie *L*. Cela n'exclut pas bien entendu les problèmes liés à la complexité, il se peut que l'accès au k-ème élément soit difficile, en temps ou en espace mémoire. Mais le principe est posé.

Reprenons la fonction *(zoom k L)* opérant sur les listes usuelles de Scheme :

```
(zoom k L) ≡ (cons (* k (first L)) (zoom k (rest L))))          (**)
```

Comme toute fonction, *cons* évalue ses deux arguments avant de procéder à l'ajout d'un nouvel élément. C'est ce que l'on nomme l'**appel par valeur**, classique dans la plupart des langages de programmation (Scheme, C, Java, Caml, etc). C'est lui qui est en faute dans le cas des listes infinies, car l'évaluation du second argument de *cons* dans (**) déclenche le calcul du reste de la liste, qui ne terminerait pas ! Il s'agit donc de retarder ce calcul jusqu'au moment où l'on en aura vraiment besoin. En quelque sorte de transformer l'appel par valeur en un **appel par nécessité**.

[1] « Introduction à la métaphysique du Mou », par Botul, Editions des Mille et une nuits, 2007. Nous suivons ici la piste de l'alorquoitisme botulien, remplaçant le tiers exclu par le tiers systématiquement envisagé ☺…

Comment retarder le calcul d'une expression ? En l'enveloppant dans une fonction sans argument. Par exemple, en transformant *(+ 1 2)* en la fonction *(λ () (+ 1 2))*. Le langage Scheme (comme Caml, Python, Ruby, Javascript, Maple, etc mais pas C par exemple) permet au programmeur de construire des fonctions anonymes. Ce que le mathématicien note x → x+2 s'écrirait en Scheme *(lambda (x) (+ x 2))* qui se lit « la fonction qui à x associe x+2 ». On peut certes donner un nom à cette fonction anonyme[2] avec *define* :

```
(define f (lambda (x) (+ x 2)))        ; ⇔ (define (f x) (+ x 2))
```

Mais l'existence des fonctions anonymes est passionnante dans la mesure où elle permet au programme de construire lui-même dynamiquement des fonctions en cours d'exécution ! Bref, nous pouvons transformer une expression *e* en une « promesse de calcul», promesse qui pourra être tenue au moment souhaité. Le langage Scheme offre à cet effet la forme *(delay e)* opérant cette transformation. Elle n'est pas difficile à définir en Scheme pur (4 ou 5 lignes) mais un peu technique, acceptons-la ainsi que sa forme duale *(force r)* prenant une expression retardée *r* construite avec *delay*, et forçant le calcul de l'expression encapsulée.

```
> (define r (delay (+ 1 2)))
> r
#<promise:r>        ; r est une promesse de calcul
> (force r)         ; je force l'exécution de la promesse
3
```

Cerise sur le gâteau : dès que le calcul de la promesse est effectué, il est mémorisé et donc immédiatement disponible si la promesse était forcée à nouveau ultérieurement. Encore une fois, *force* et *delay* sont faciles à définir si elles n'existaient pas d'emblée, cf [PCPS].

La fonction *(scons x L)* dans le cas d'une liste infinie *L* devra donc retarder la récurrence. Paresseux que nous sommes en train de devenir, nous retarderons aussi la construction de la nouvelle liste. En somme :

```
(scons x L) ⇔ (delay (cons x (delay L)))
```

Cette équivalence est une **équivalence syntaxique**. Là où j'écris le membre gauche, l'interprète Scheme ne verra que le membre droit ! Ce bluff syntaxique se nomme une **macro**, les programmeurs en langage C les pratiquent à petite échelle avec leur *#define* par exemple. De tous les langages de programmation, Scheme possède le système de macros le plus élaboré (il est dit *hygiénique*, mais cela nous entraînerait trop loin, cf [PCPS]). La plupart des langages, par exemple Java, ne permettent pas ce bluff réputé sportif et ne disposent pas de macros. En Scheme, la primitive *define-syntax* permet d'étendre la syntaxe du langage, c'est elle par exemple qui permet de définir les mots *while* et *for* pour programmer des boucles[3].

```
(define-syntax scons
  (syntax-rules ()
    ((scons x L) (delay (cons x (delay L))))))     ; une « règle syntaxique »
```

Je suis d'ores et déjà en situation de définir l'ensemble infini des entiers naturels **N** :

```
(define (entiers-depuis n)                         ; la liste infinie [n,+∞[
  (scons n (entiers-depuis (+ n 1))))

(define N (entiers-depuis 0))
```

Ah, vous sentez cet air frais au voisinage de l'infini ? Il est cependant trop tôt pour demander à contempler **N** :

```
> N
#<promise>
```

N est encore à l'état de promesse. Celle-ci sera tenue le jour où nous aurons besoin d'avoir effectivement des entiers naturels… Vous voulez les multiples de 3 ? Qu'à cela ne tienne :

```
(define 3N (szoom 3 N))
```

En ce qui concerne l'axiome du choix, il nous faut être capable d'exhiber un élément d'une liste infinie *L*, par exemple le premier disponible, ainsi que le reste. Implémentons *sfirst* et *srest* :

---

[2] Les lambda-fonctions sont à l'origine du *lambda-calcul* de Church vers 1940. En ce sens, Scheme n'est qu'un lambda-calcul appliqué.

[3] La récurrence étant optimisée en Scheme contrairement à Java, C, Python ou Ruby, elle permet de programmer des boucles sans avoir besoin de mots spéciaux comme *while* ou de construction autre que la récurrence.

```
(define (sfirst L) (first (force L)))
(define (srest L) (force (rest (force L))))
```

L'utilisation de la primitive *force* permet de tenir les promesses et de forcer les calculs qui avaient été retardés par *delay*. Nous sommes dès lors en possession de calculer le *k*-ème élément d'une liste infinie *L* :

```
(define (sref L k)       ; l'élément numéro k ≥ 0 d'une liste infinie L
  (if (= k 0)
      (sfirst L)
      (sref (srest L) (- k 1))))       ; par récurrence sur le couple (L,k)
```

Serions-nous prêts à calculer le 20000-ème nombre de Hamming ? Accrochez vos ceintures :

```
> (time (sref H 20000))
Time = 0.2 sec
154418349070986750000000                    ; il s'agit de 2^6 3^31 5^8
```

Comme disait Einstein, Dieu est sophistiqué, mais il n'est pas méchant. Vous me direz que c'est quand même beaucoup de travail et de concepts pas encore populaires pour résoudre un seul problème ! Je vous rétorquerai que tout ce travail va produire une boîte à outils largement réutilisable, et pour vous en convaincre, je n'irai pas du côté des séries formelles, où ces outils s'avèrent d'une puissance étonnante, mais simplement d'une définition toujours aussi élégante de la suite **F** de Fibonacci :

$$\mathbf{F} = 0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ldots$$

Traditionnellement cette suite est définie par une récurrence double :

$$f_0 = 0,\ f_1 = 1,\ f_n = f_{n-1} + f_{n-2}$$

qui conduit à un calcul de coût exponentiel. On torture d'habitude ce joli algorithme avec une boucle pour obtenir un processus plus rapide, mais en s'éloignant alors de la pensée mathématique pure. Or la liste infinie **F** s'auto-définit par récurrence très simplement :

```
  0 1 1 2 3  5  8 13 ...              F
+ 1 1 2 3 5  8 13 21 ...            + (srest F)
-------------------------           = ---------------
= 1 2 3 5 8 13 21 34 ...            (srest (srest F))
```

Il manque juste les deux premiers termes 0 et 1, que nous rajoutons à la main. La définition de **F** peut difficilement être plus courte, une marge aurait suffit à Fermat :

```
(define F (scons 0 (scons 1 (add F (srest F)))))
```

```
> (time (sref F 300))          ; le terme d'indice 300 de la suite de Fibonacci
Time = 0.002 sec
222232244629420445529739893461909967206666939096499764990979600
```

Je vous laisse définir en une ligne l'addition *(add L1 L2)* de deux listes infinies et vous remercie de votre patience. Techniquement, les « listes infinies » se nomment des **flots** (*streams*), conduisant à une **programmation paresseuse** (*lazy*). Il est aussi intéressant de savoir définir les primitives sur les flots que d'utiliser ces derniers. Cela suppose bien entendu que le langage de programmation utilisé est suffisamment malléable pour phagocyter un paradigme de programmation pour lequel il n'avait pas été prévu. C'est le cas du langage algorithmique Scheme, dont le degré d'expressivité est étonnemment élevé.

Bibliographie :

[R^{6}RS] : *Revised^6 Report on the Algorithmic Language Scheme* (`http://www.r6rs.org`).

[SICP] : *Structure and Interpretation of Computer Programs*, H. Abelson & G. Sussman, MIT Press, lisible sur le Web (`http://mitpress.mit.edu/sicp`). Nous avons développé de manière descendante la solution de l'exercice 3.56.

[PCPS] : *Premiers Cours de Programmation avec Scheme*, J-P. Roy, Editions Ellipses, à paraître à la rentrée 2010.