Rapport de projet 1A

Efficient code generation

Bourdillas Romain
Bourgery Yoann
Oillarburu Jean-marie
Seoane Margaux

*E*ncadrant : M. Antoine Rollet

22 Mars 2016 - 20 Mai 2016

Table des matières

Introduction

Ce projet a pour objectif de générer de manière automatique un code en scheme et C, le tout en utilisant le langage scheme. L'exemple choisi ici est celui du calcul de x^n. Cela nous a amené au problème du calcul de chaînes d'addition pour un entier n qui est une liste d'entiers strictement croissante commençant par 1, telle que tout entier dans la liste est la somme de deux entiers présents dans les indices antérieurs, et telle que le dernier élément soit égal à n. L'obtention de cette liste pour un entier n permettra de calculer x^n : il s'agit de la suite d'exposants associée à un calcul de x^n par multiplications successives de termes précédents, en n'utilisant qu'une seule multiplication à chaque étape. Pour cela, nous avons dû nous intéresser à différentes méthodes de calcul de chaînes d'addition. Les chaînes les plus courtes donneront les temps de calcul de l'exponentiation minimaux.

1 Vérification

1.1 Vérification de la chaîne d'addition

Avant de nous intéresser à différentes méthodes pour générer les chaînes d'additions, il est primordial de savoir tester si une liste est bel et bien une chaîne d'addition.

Il nous faut donc une fonction qui puisse vérifier qu'une liste commence par 1, qu'elle est strictement croissante et que chaque élément de la liste est la somme de deux éléments de la liste.

Notre fonction `verif-chaine` se charge de toutes ces vérifications, notamment grâce à la fonction `est-sommable` qui vérifie si l'élément donné en argument est sommable à partir de 2 éléments d'une liste donnée.

La complexité de cet algorithme est quadratique($\Theta(n^2)$) par rapport à la longueur de la chaîne puisque `verif-chaine` effectue un parcours de liste pour chaque élément de la chaîne un parcours (`est-sommable`).

1.2 Algorithme d'application d'une chaîne

A présent, il s'agit d'être capable d'utiliser une chaîne d'addition afin de calculer x^n. Pour cela, il nous faut mémoriser - pour tout élément k de la liste d'exposants - la valeur de x^k. C'est donc assez naturellement que nous avons choisi d'utiliser une liste de couple pour stocker le tout. Cette liste de couple est donc de la forme : `'((n,` x^n`) ⋯ (i,` x^i`) ⋯ (2,` x^2`) (1,` x`))`
avec `1, 2, ⋯, i, ⋯, n` appartenant à la chaîne donnée.

Elle est construite récursivement en parcourant la liste d'exposants et est stockée dans un accumulateur. Les éléments s'y ajoute comme-ci : lorsque nous arrivons à un élément i de la chaîne, nous cherchons les deux éléments de la chaîne qui, en se sommant, donnent i. Une fois que nous avons ces deux éléments, il suffit de multiplier leurs seconds membres du couple associés dans l'accumulateur. Enfin, on ajoute le couple créé en tête de l'accumulateur. Pour obtenir x^n, il nous suffit alors de prendre le deuxième élément du couple de tête.

Sur un exemple : prenons la chaîne d'addition `'(1 2 3 6 7)`. Supposons que dans l'algorithme expliqué ci-dessus, nous en sommes à l'étape où nous considérons l'élément `3` de la liste. A ce moment là, l'accumulateur vaut : `'((2,` x^2`) (1,` x`))`. `3` est construit en sommant `2` et `1`. Les éléments associés à 2 et 1 dans l'accumulateur sont x^2 et x. On les multiplie pour donner x^3 et on ajoute le couple `'(3,` x^3`)` en tête de l'accumulateur qui vaut donc `'((3,` x^3`) (2,` x^2`) (1,` x`))`.
La complexité de l'algorithme est de $\Theta(n^2)$

1.3 Registres et complexité

Une fois qu'il est possible de calculer x^n à partir d'une chaîne d'addition, il faut savoir la complexité du calcul en terme de registre à partir d'une chaîne d'addition donnée. Nous acceptons les calculs in-place : c'est à dire que nous nous permettons de calculer la valeur d'un registre en utilisant les valeurs définies précédemment.

Pour cela, nous avons implémenté la fonction `count-nb-register-chaine`.

Cette fonction appelle des fonctions qui retournent l'ensemble des registres nécessaires pour une chaîne donnée : à une chaîne d'addition '(1 2 4 8 9) on associe la liste '((1 1) (9 9) (9 9) (9 9) (9 1)) En effet nous pouvons remarquer que les deux calculs suivant sont équivalents :

$x2 \leftarrow x1 \times x1$	$x8 \leftarrow x1 \times x1$
$x4 \leftarrow x2 \times x2$	$x9 \leftarrow x9 \times x9$
$x8 \leftarrow x4 \times x4$	$x9 \leftarrow x9 \times x9$
$x9 \leftarrow x8 \times x1$	$x9 \leftarrow x9 \times x1$

Ensuite, il ne reste plus qu'à aplatir cette liste, à retirer les doublons de cette liste et calculer sa longueur. Nous verrons le principe algorithmique pour avoir la liste des noms des registres dans la partie 3.2 *Production d'algorithme en C*.

2 Génération de chaînes

2.1 Génération de l'ensemble des chaînes

Nous avons décidé de ne représenter que les chaînes utiles, c'est à dire sans valeurs calculées mais inutilisées par la suite. Nous avons décidé, pour calculer l'ensemble des chaînes d'additions, de générer un arbre dont chaque branche est une chaîne d'addition. La construction de l'arbre se fait de manière récursive : à chaque étape s'ajoute à la liste la somme du premier élément avec tous les éléments de cette liste.
Cela permet de construire un arbre tel que celui-ci :

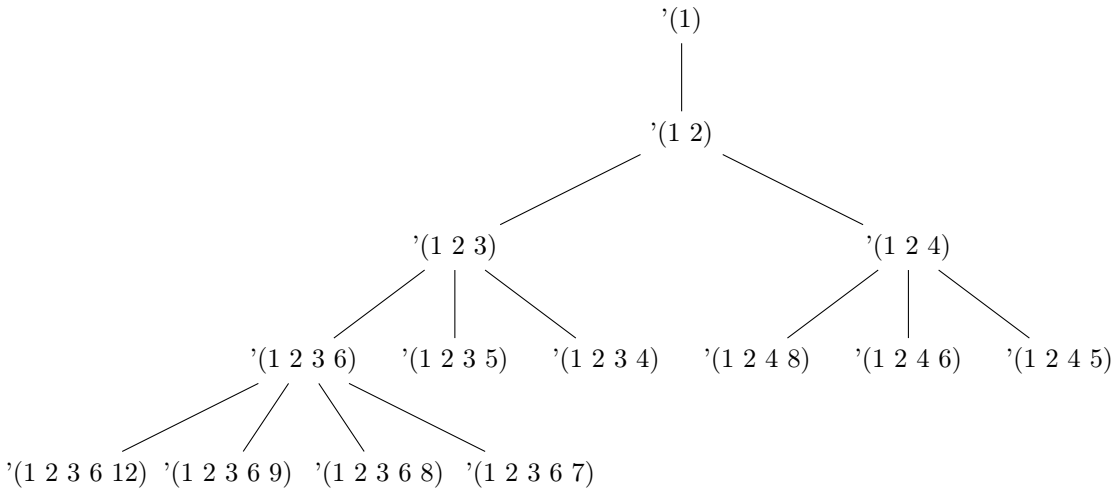


FIGURE 1 – Exemple de génération de chaîne

Ce principe a une complexité clairement exponentielle voire pire qu'exponentielle : à la ligne n, il y a $(n-1)!$ listes différentes possibles. Elle devient donc très vite inutilisable lorsque la valeur en entrée est grande.

2.2 Méthode binaire

La méthode binaire est très efficace pour calculer un bon nombre de chaînes d'addition. Elle donne les meilleures chaînes pour les puissances de 2 et est également très facile à calculer.

Il suffit de multiplier la dernière valeur de la chaîne d'addition par deux et de l'ajouter à la chaîne d'addition jusqu'à obtenir ou dépasser la valeur que l'on souhaite obtenir, que l'on notera n. Si l'on dépasse n, on remonte dans la chaîne d'addition afin de sommer la valeur courante à la dernière valeur de la chaîne.

Prenons la chaîne d'addition de la valeur 8, en gras à gauche la valeur utilisée pour calculer le terme suivant, la chaîne d'addition donnée sera donc :

Chaîne de départ	Calcul de l'élément suivant	Nouvelle chaîne
'(**1**)	$2 \times 1 = 2$	'(1 2)
'(1 **2**)	$2 \times 2 = 4$	'(1 2 4)
'(1 2 **4**)	$2 \times 4 = 8$	'(1 2 4 8)

Par contre, si l'on souhaite calculer 9, la chaîne d'addition donnée sera :

Chaîne de départ	Calcul de l'élément suivant	Nouvelle chaîne
'(**1**)	$1 + 1 = 2$	'(1 **2**)
'(1 **2**)	$2 + 2 = 4$	'(1 2 **4**)
'(1 2 **4**)	$4 + 4 = 8$	'(1 2 4 **8**)
'(1 2 4 **8**)	$8 + 8 = 16 > 9$	'(1 2 4 8)
'(1 2 **4** **8**)	$8 + 4 = 12 > 9$	'(1 2 4 8)
'(1 **2** 4 **8**)	$8 + 2 = 10 > 9$	'(1 2 4 8)
'(**1** 2 4 **8**)	$8 + 1 = 9$	'(1 2 4 8 **9**)

La complexité de cette méthode est de $\Theta(log(n))$

2.3 Méthode du facteur premier

Nous allons ici présenter une méthode efficace pour calculer des chaînes d'addition : la méthode des facteurs premiers. Le principe est le suivant : pour construire la chaîne de n, on regarde d'abord si n est premier. Si c'est le cas, on utilise la chaîne de $n - 1$ pour construire celle de n. Si ce n'est pas le cas, on cherche p et q tel que $n = pq$ et p le plus petit facteur premier de n. Ensuite, on construit la chaîne d'addition de p suivie de celle de q (en concaténant) multipliée par p.

Voici l'algorithme (l'informalité est intentionnelle), nous supposons que l'utilisateur ne demande pas la chaîne triviale de 1 :

```
   input  : Un entier n
   output: La chaîne d'addition
 1 if n = 2 then
 2 |  return liste(1, 2);
 3 else
 4 |  if premier(n) then
 5 |  |  return concatène(liste(n), chaine(n − 1));
 6 |  else
 7 |  |  n ← p ∗ q;
 8 |  |  return concatène(chaine(p), chaine(q));
 9 |  end
10 end
```

Algorithm 1: Méthode facteur premier

La fonction `couple-pq` de notre code nous permet d'obtenir le couple (p, q) tel que $n = pq$ et p est le plus petit facteur premier de n.

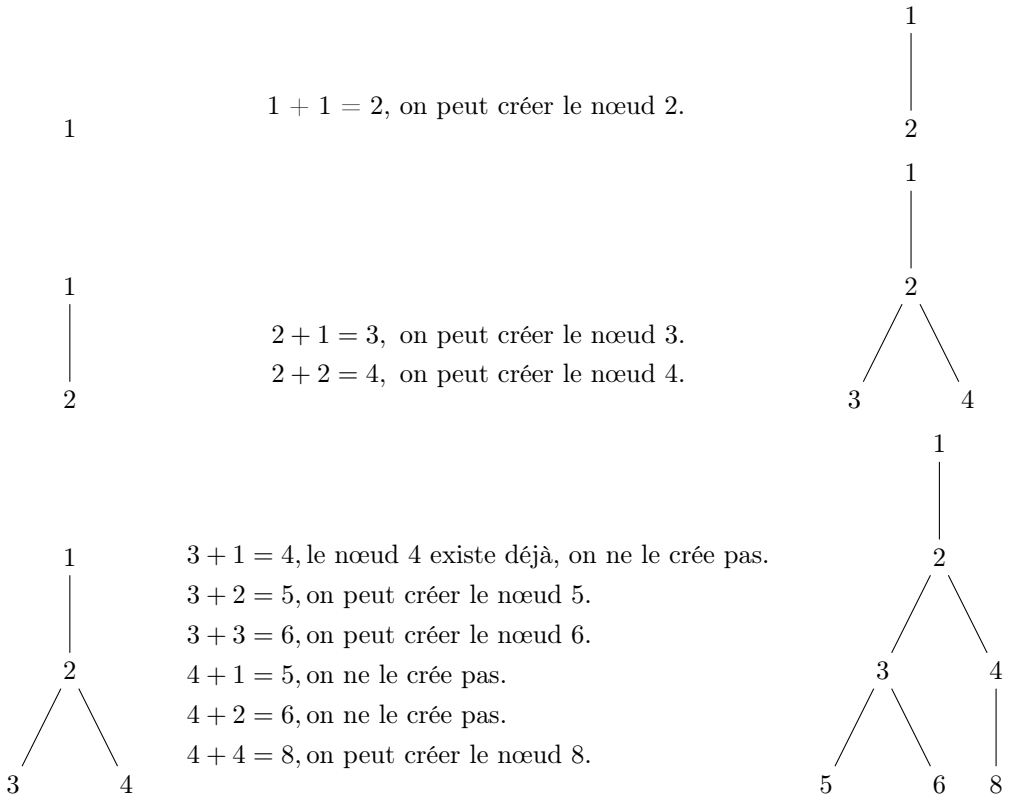
La chaîne renvoyée par l'algorithme ci-dessus présente certains doublons et n'est pas strictement croissante. Pour cela, nous utilisons notre fonction `unique` et la fonction de Racket `sort`.

La complexité de cet algorithme dépend du coût de `couple-pq`, qui est au pire linéaire, et du nombre de facteurs premiers de n. En notant l la longueur de la chaîne et m le nombre de facteurs premier, la complexité de l'algorithme est en $\Theta(lm)$. Elle est donc moins efficace en temps que la méthode binaire qui est de coût logarithmique. Au niveau de la longueur des chaînes, elle est d'efficacité similaire à la méthode binaire.

2.4 Méthode de l'arbre de puissance

Cette dernière méthode est basée sur la construction d'un arbre appelé arbre de Knuth. Chaque nœud est construit de la manière suivante : il est la somme de son père avec un de ses plus hauts parents, y compris le père lui même. Si le nœud existe déjà alors on ne l'ajoute pas.

Voici un exemple de construction de l'arbre jusqu'à la profondeur 3 :



Pour lire la chaîne de n, il suffit de trouver le noeud n et de regarder le chemin allant de n jusqu'à la racine de l'arbre. Ce chemin est la chaîne.

L'arbre est construit en largeur, la difficulté au niveau de l'implémentation a été de stocker les bonnes choses aux bons endroits. En effet, dans la construction de l'arbre, nous avons besoin d'avoir accès :

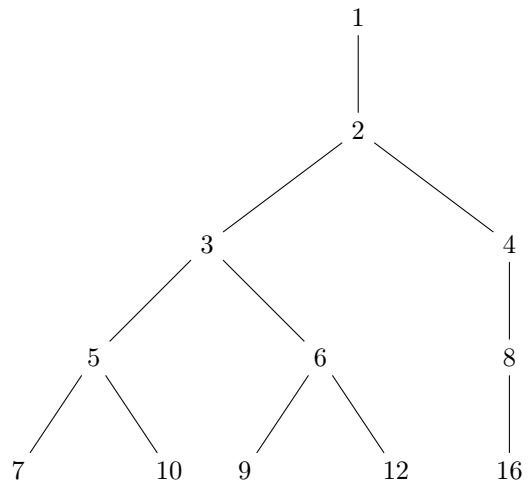
— à la liste des nœuds de la profondeur dont on veut créer les fils

— au reste de l'arbre (pour vérifier si un noeud existe déjà)

Notre programme s'arrête dès lors que l'on a construit le nœud n.

Nous avons modélisé l'arbre sous forme de listes de listes, les sous-listes représentant les diverses chaînes.

Exemple :



Arbre : '((1) (1 2) (1 2 3) (1 2 4) (1 2 3 5) (1 2 3 6) (1 2 4 8) (1 2 3 5 7) (1 2 3 5 10) (1 2 3 6 9) (1 2 3 6 12) (1 2 4 8 16))

`gen-arbre` génère donc l'arbre de Knuth pour l'entier n donné en argument. La génération de l'arbre est coûteuse en temps. En effet, pour chaque nœud créé, il faut parcourir l'arbre (qui est de taille exponentielle par rapport à n) pour s'assurer qu'il n'est pas déjà existant.

Une solution - que nous n'avons pas implémentée - pour réduire ce coût serait de calculer un arbre de Knuth pour un n très grand qui servirait de base de données, puis de rechercher la chaîne voulue dedans. En ne comptant pas la génération de l'arbre, le calcul de la chaîne serait logarithmique.

D'après l'ENS Lyon, cette méthode est la plus efficace au niveau de la longueur de la chaîne. En effet, pour n plus petit que 100 000, elle génère une chaîne plus longue que les deux autres méthodes dans 6 cas seulement.

3 Production de code

3.1 Production d'algorithme en Scheme

Il est à présent temps de générer le code en Scheme permettant de calculer x^n grâce à une chaîne d'addition donnée. Le but est de créer une liste de symboles comprenant toutes les instructions en Scheme. Ainsi, il sera possible d'évaluer la liste grâce à la fonction de Scheme `eval`.

Le code que nous cherchons à produire est de la forme :

```
(lambda (mult x)
(let* ((x2 (mult x x))
       (x3 (mult x x2))
       (x6 (mult x3 x3))
       (x7 (mult x x6)))
       x7))
```

Ce code est produit par notre fonction `gen-code-scheme` qui concatène les bons symboles grâce aux fonctions auxiliaires :

- `cherche-somme-liste` : permet d'obtenir la bonne combinaison d'exposants à sommer afin d'obtenir celui voulu.
- `gen-list` : permet d'obtenir (à partir des argument $n1$ et $n2$) $x(n1+n2)$ (mult $x(n1)$ $x(n2)$).
- `gen-let-aux` : permet d'obtenir le corps du $let*$.
- `concat` : permet (à partir des arguments a et b) d'obtenir le symbole ab.

Pour rendre l'évaluation possible, nous devons utiliser les définitions suivantes fournies par Mr.Renault :

```
(define-namespace-anchor anchor)
(define ns (namespace-anchor->namespace anchor))
(define (eval-expr expr) (eval expr ns))
```

Enfin, notre fonction `evaluation` permet - grâce aux définitions précédentes - d'évaluer automatiquement le code Scheme pour un x et une chaîne donnée.

3.2 Production d'algorithme en C

Nous avons créé deux versions de générateur de code C : une version classique et une version qui optimise la mémoire. En effet, il est possible de diminuer le nombre de variables à utiliser (comme vu en partie 1.3 *Registres et complexité*). Pour cela, nous utilisons un algorithme récursif qui construit les noms des registres de manière récursive. Cet algorithme nécessite de connaître la chaine d'addition (que nous noterons CHAINE par la suite) ainsi que l'ensemble des couples nécessaires pour obtenir un élément de la chaine d'addition (que nous noterons CH-PREDECESSEUR).

Voici quelques exemples pour illustrer cette structure :

- CHAINE = (1 2 4 8 12 14)
  CH-PREDECESSEUR = ((1 1) (2 2) (4 4) (8 4) (12 2))

— CHAINE = (1 2 3 6 12 15)
CH-PREDECESSEUR = ((1 1) (2 1) (3 3) (6 6) (12 3))
— CHAINE = (1 2 3 6 12 15 30 31)
CH-PREDECESSEUR = ((1 1) (2 1) (3 3) (6 6) (12 3) (15 15) (30 1))

Nous pouvons remarquer qu'il y a un élément de moins dans CH-PREDECESSEUR que dans CHAINE. C'est pourquoi nous considérerons par la suite que CHAINE ne possède pas l'élément 1.

L'idée est de parcourir CHAINE et CH-PREDECESSEUR de manière décroissante et d'enregistrer les variables utiles pour savoir que lors de leurs calcul elles seront utile par la suite, et donc de ne pas les écraser. Nous utilisons l'algorithme ci-dessous :

```
1  Fonction récursive nom-registre(chaine, ch-predecesseur, utile,
     ch-finale, predecesseur-final)
2  if est-vide(chaine) then
3  |  ret ch-finale, predecesseur-final
4  else
5  |  tete ← tete(chaine)
6  |  (a,b) ← tete(ch-predecesseur)
7  |  if a ∉ utile ∧ a = b then
8  |  |  nom-registre(queue(remplace(a, tete, chaine)),
   |  |   queue(remplace(a, tete, ch-predecesseur)), utile, cons(tete,
   |  |   ch-finale), cons((tete, tete), predecesseur-final))
9  |  else
10 |  |  if a ∉ utile then
11 |  |  |  nom-registre(queue(remplace(a, tete, chaine)),
   |  |  |   queue(remplace(a, tete, ch-predecesseur)), cons(b, utile),
   |  |  |   cons(tete, ch-finale), cons((tete, b), predecesseur-final))
12 |  |  else
13 |  |  |  if b ∉ utile then
14 |  |  |  |  nom-registre(queue(remplace(b, tete, chaine)),
   |  |  |  |   queue(remplace(b, tete, ch-predecesseur)), utile,
   |  |  |  |   cons(tete, ch-finale), cons((a, tete), predecesseur-final))
15 |  |  |  else
16 |  |  |  |  nom-registre(queue(chaine), queue(ch-predecesseur),
   |  |  |  |   utile, cons(tete, ch-finale), cons((a, b),
   |  |  |  |   predecesseur-final))
17 |  |  |  end
18 |  |  end
19 |  end
20 end
```

Algorithm 2: calcul des noms des registres

"utile" est la liste des registres utiles pour la suite du calcul,
"ch-final" est la chaine d'addition représentée par le noms des registres des va-

riable,
"predecesseur-final" est la liste de couples des prédécesseurs représentés par le nom des variables à multiplier.

Appliquons cet algorithme sur l'exemple suivant avec : CHAINE = (1 2 3 6 12 15 30 31) et CH-PREDECESSEUR = ((1 1) (2 1) (3 3) (6 6) (12 3) (15 15) (30 1))

Ci-dessous, l'arbre des appels :

nom-registre((30 15 12 6 3 2), ((15 15) (12 3) (6 6) (3 3) (2 1) (1 1)), (1 30), (), ())

nom-registre((15 12 6 3 2), ((12 3) (6 6) (3 3) (2 1) (1 1)), (1 30), (30), ((30, 30)))

nom-registre((12 6 3 2), ((6 6) (3 3) (2 1) (1 1)), (1 30 3), (30 30), ((30, 30) (30 3)))

nom-registre((6 3 2), ((3 3) (2 1) (1 1)), (1 30 3), (30 30 30), ((30, 30) (30 3) (30 30)))

nom-registre((3 2), ((2 1) (1 1)), (1 30 3), (30 30 30 30), ((30, 30) (30 3) (30 30) (3 3)))

nom-registre((2), ((1 1)), (1 30 3), (30 30 30 30 3), ((30, 30) (30 3) (30 30) (3 3) (3 1)))

nom-registre((), (), (1 30 3), (30 30 30 30 3 3), ((30, 30) (30 3) (30 30) (3 3) (3 1) (1 1)))

(30 30 30 30 3 3), ((30, 30) (30 3) (30 30) (3 3) (3 1) (1 1))

En comparant les multiplications à effectuer en parallèle des noms des registre, nous nous rendons bien compte que cet algorithme fonctionne :

$x2 \leftarrow x1 \times x1$	$x3 \leftarrow x1 \times x1$
$x3 \leftarrow x2 \times x1$	$x3 \leftarrow x3 \times x1$
$x6 \leftarrow x3 \times x3$	$x30 \leftarrow x3 \times x3$
$x12 \leftarrow x6 \times x6$	$x30 \leftarrow x30 \times x30$
$x15 \leftarrow x12 \times x3$	$x30 \leftarrow x30 \times x3$
$x30 \leftarrow x15 \times x15$	$x30 \leftarrow x30 \times x30$

4 Conclusion

Nous avons réussi à mettre au point une implémentation pour chaque objectif du projet. Nous sommes capables de vérifier si une chaîne est bien une chaîne d'addition pour un entier n donné, de compter le nombre de registres nécessaires minimum et de générer des chaînes d'addition pour un entier n avec trois méthodes différentes : binaire, facteur premier et arbre des puissances. Nous pouvons aussi évaluer une chaîne d'addition avec un entier donné afin d'effectuer x^n

Nous sommes également parvenus à produire des algorithmes en scheme et en c fonctionnels et utilisables pour calculer une valeur à la puissance n.

Ce projet a été très intéressant dans la mesure où il est possible de faire beaucoup de chose en scheme. Implémenter un algorithme qui permet de générer n'importe quel algorithme d'exponentiation rapide, que cela soit en scheme ou en c, demande un peu de travail mais peut permettre un gain de temps non négligeable par la suite.