ÉVALUATION DES EXPRESSIONS ARITHMÉTIQUES SUR UNE PRAM CREW

A. LEGRAND

Ce sujet est un peu dur mais relativement classique. Je vous encourage donc à le chercher à plusieurs si vous le désirez mais par contre, je veux que chacun rédige à sa sauce et me rende un devoir personnel. Je suis à votre disposition si l'énoncé vous paraît un peu obscur. La dernière question est un peu longue et fastidieuse mais ce n'est pas une raison pour ne pas la chercher. Enfin, ce devoir est à rendre pour le mercredi 21 novembre à midi *au plus tard.*

Résumé. Le sujet de ce devoir concerne l'évaluation d'une expression arithmétique de longueur n en temps $O(\log n)$ avec $O\left(\frac{n}{\log n}\right)$ processeurs sur une PRAM CREW. L'algorithme proposé est donc à la fois optimal en temps (regarder l'évaluation de $\sum_{i=1}^{2^m} x_i$) et en travail (puisque le travail global est de l'ordre de n). La première partie consiste à construire un arbre d'évaluation de l'expression arithmétique en temps $O(\log n)$ avec $O\left(\frac{n}{\log n}\right)$ processeurs et la seconde partie porte sur l'évaluation de cet arbre en temps $O(\log n)$ avec $O(n)$ processeurs (l'algorithme optimal étant trop long pour ce devoir) en utilisant des techniques de type saut de pointeurs. Contrairement à ce qu'on pourrait imaginer, le principe n'est pas de construire un arbre très «dense» pour utiliser le schéma de réduction appliqué à une somme par exemple : il faut construire un arbre qui se prête bien à une évaluation «pseudo-formelle».

1. Idées générales

Le sujet de ce devoir concerne l'évaluation d'une expression arithmétique $(+,-,\times,/)$ de longueur n en temps $O(\log n)$ avec $O\left(\frac{n}{\log n}\right)$ processeurs sur une PRAM CREW. L'algorithme proposé est donc à la fois optimal en temps (regarder l'évaluation de $\sum_{i=1}^{2^m} x_i$) et en travail (puisque le travail global est de l'ordre de n). La première partie consiste à construire un arbre d'évaluation de l'expression arithmétique en temps $O(\log n)$ avec $O\left(\frac{n}{\log n}\right)$ processeurs et la seconde partie porte sur l'évaluation de cet arbre en temps $O(\log n)$ avec $O(n)$ processeurs (l'algorithme optimal étant trop long pour ce devoir) en utilisant des techniques de type saut de pointeurs. Contrairement à ce qu'on pourrait imaginer, le principe n'est pas de construire un arbre très «dense» pour utiliser le schéma de réduction appliqué à une somme par exemple. L'idée est de construire un arbre qui se prête bien à une évaluation «pseudo-formelle».

En effet, dans le cas général, il n'est pas facile de construire un arbre d'évaluation équilibré pour une expression arithmétique. Regardons par exemple l'évaluation d'un polynôme par le schéma de Horner :

$$a_1 + x \times (a_2 + x \times (a_3 + x \times (a_4 + x)))$$

Les arbres naturels correspondant au schéma de Horner et à la forme développée sont présentés dans la figure 1. L'évaluation avec la forme développée conduit à un arbre mieux équilibré mais elle introduit des calculs redondants et ne permet donc pas une évaluation beaucoup plus rapide. Dans la figure 2, nous présentons brièvement le schéma d'évaluation que nous allons utiliser. L'idée est d'associer à chaque nœud (i.e. chaque opérateur) de l'arbre une fonction et d'utiliser une forme de saut de pointeur (afin d'obtenir le temps en $O(\log n)$) pour évaluer l'expression.

2. Construction de l'arbre d'évaluation

Définition 1. Une expression correctement parenthésée est une expression arithmétique contenant autant de parenthèses ouvrantes que de parenthèses fermantes et telle qu'il y ait plus de parenthèses ouvrantes que de parenthèses fermantes dant tout préfixe.

Date: 26 novembre 2001.

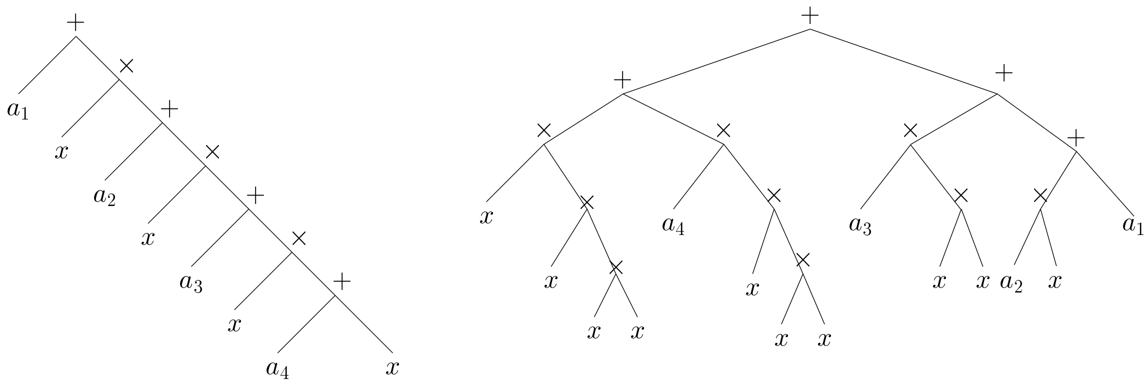

Fig. 1 – Arbres d'évaluation du polynôme.

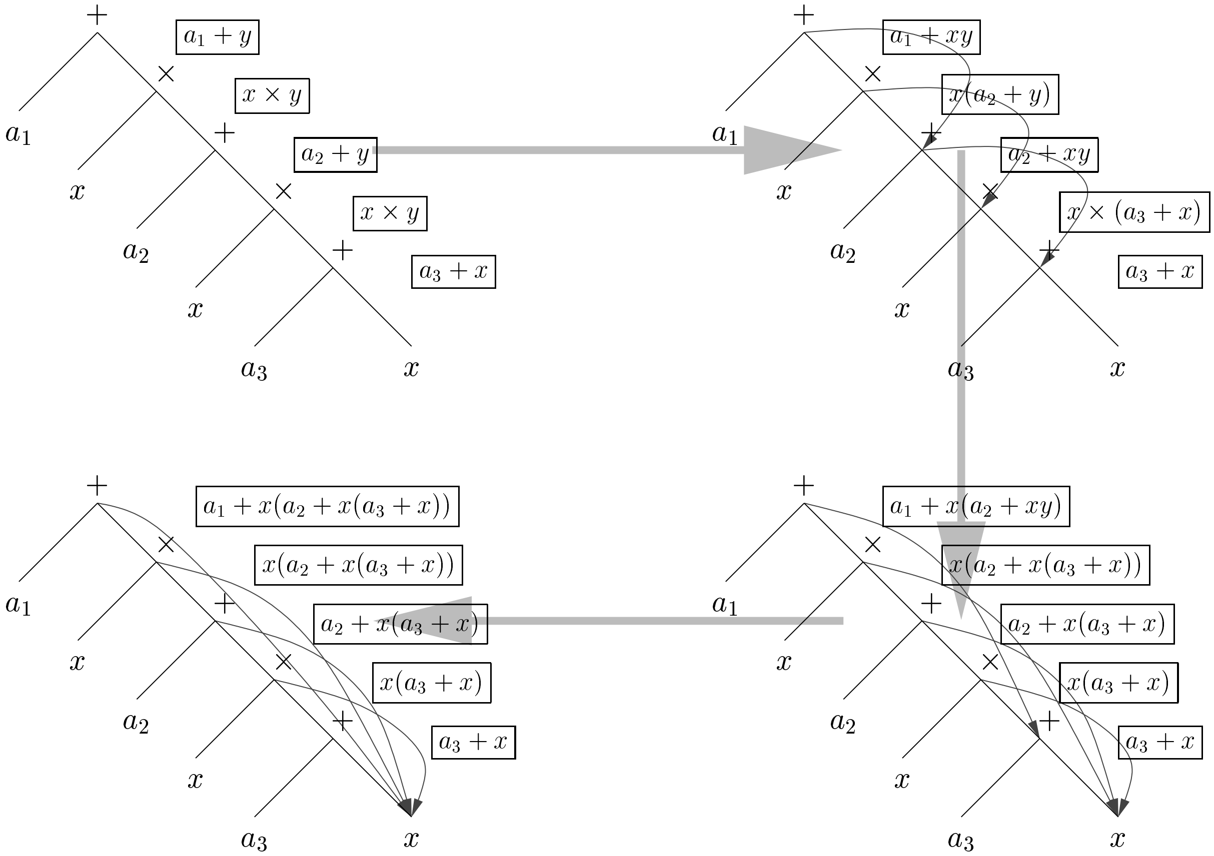

Fig. 2 – Arbre d'évaluation symbolique du polynôme.

Définition 2. Une expression simple est soit une variable entourée de parenthèses, soit une expression arithmétique de la forme

$$(E_1 \, op_1 \, E_2 \ldots op_{k-1} \, E_k),$$

où tous les opérateurs op_i ont la même priorité et où les E_i sont elles mêmes des expressions simples.

Définition 3 (La fonction match). Pour une expression correctement parenthésée, on définit la fonction match associant à une parenthèse ouvrante la penthèse fermante qui lui correspond.

▷ **Question 1.** Écrire un algorithme qui transforme une expression correctement parenthésée entourée de parenthèses en une expression simple en temps $O(\log n)$ avec $O\left(\frac{n}{\log n}\right)$ processeurs.

Réponse. En fait, on va écrire un algorithme qui réalise cette opération en temps $O(1)$ avec $O(n)$ et qui pourra donc être directement simulé par un algorithme qui s'exécute en temps $O(\log n)$ avec $O\left(\frac{n}{\log n}\right)$ processeurs. On suppose que l'expression est stockée sous la forme d'une liste chaînée et que chaque caractère de l'expression est associé à un processeur. On applique alors l'algorithme 1.

Conversion
Pour tout i **en parallèle**
  Si $val(i) = \times$ **Alors** $val(i) =) \times ($
  Si $val(i) = /$ **Alors** $val(i) =)/($
  Si $val(i) = +$ **Alors** $val(i) =)) + (($
  Si $val(i) = -$ **Alors** $val(i) =)) - (($
  Si $val(i) = ($ **Alors** $val(i) = ((($
  Si $val(i) =)$ **Alors** $val(i) =)))$

Algo. 1: Conversion d'une expression en expression simple.

La preuve de la correction de l'algorithme est facile mais longue. L'idée est simplement d'ajouter des parenthèses autour de $\times$ et $/$ pour obtenir une expression simple. Comme $\times$ et $/$ ont une priorité supérieure à $+$ et $-$, il est nécessaire de protéger $+$ et $-$ en insérant des parenthèses complémentaires.
Pour expliquer le fonctionnement de l'algorithme, nous montrons la transformation de l'expression

$$(a_1 + x \times (a_2 + x \times (a_3 + x \times (a_4 + x)))).$$

L'expression simple obtenue est

$$(\mathbf{((}a_1\mathbf{))} + \mathbf{((}x\mathbf{)} \times \mathbf{(}(\mathbf{((}a_2\mathbf{))} + \mathbf{((}x\mathbf{)} \times \mathbf{(}(\mathbf{((}a_3\mathbf{))} + \mathbf{((}x\mathbf{)} \times \mathbf{(}(\mathbf{((}a_4\mathbf{))} + \mathbf{((}x\mathbf{))})\mathbf{))})\mathbf{))})\mathbf{))})\mathbf{))}.$$

où les parenthèses ajoutées sont indiquées en gras.
Bien entendu, il est possible (dans ce cas) de supprimer certaines parenthèses tout en conservant une expression simple. Nous réaliserons cette opération un peu plus tard car il n'est pas facile de l'effectuer efficacement tant que la fonction match n'a pas été calculée.
Il est important de noter que l'expression simple obtenue est au plus cinq fois plus longue que la chaîne initiale (puisque chaque caractère est remplace par au plus cinq caractères). □

On considère maintenant une chaîne constituée uniquement de parenthèses (de longueur 2^m avec $m|2^m$) obtenue par la suppression dans une expression simple de tous les autres caractères. On va montrer comment calculer la fonction match sur cette séquence de parenthèses.

Définition 4. On définit l'opérateur de réduction $\mathcal{R}$ sur une paire de parenthèse par

$$\mathcal{R}((() = ((\qquad \mathcal{R}())) =)) \qquad \mathcal{R}(()) = \qquad \mathcal{R}()() =)($$

On dira qu'une séquence de parenthèses a été réduite (est sous forme irréductible) si l'application de l'opérateur de réduction sur une paire quelconque de deux parenthèses successives est sans effet.

▷ **Question 2.** Quelle est la forme d'une séquence réduite de parenthèses dans le cas général. Quelle est la forme réduite obtenue quand on fusionne deux séquences irréductibles de parenthèses.

Réponse. Une séquence irréductible de parenthèses est sous la forme

$$)^i(^j.$$

En effet, il est facile de vérifier que si une parenthèse ouvrante se trouve à gauche d'une parenthèse fermante (ou si une parenthèse fermante se trouve à droite d'une parenthèse ouvrante) alors la séquence peut être

encore réduite.
On vérifie également que la réduction de la fusion de deux séquences irréductibles $)^i(^j$ et $)^k(^l$ donne

$$)^{i+k-j}(^l \text{ si } k \geqslant j \qquad \text{et} \qquad)^i(^{l+j-k} \text{ si } j \geqslant k.$$

□

▹ **Question 3.** Donner un algorithme parallèle pour construire un construire en temps $O(\log n)$ avec $O\left(\frac{n}{\log n}\right)$ processeurs un arbre dont chaque nœud représente la séquence irréductible des feuilles de son sous arbre (on pourra supposer que la longueur de la séquence de parenthèses initiale est 2^k et que k divise 2^k).

Réponse. On sait (d'après la question précédente) qu'on peut réaliser la fusion de deux séquences irréductibles en temps constant. Il est donc très facile de concevoir un algorithme qui permet de construire l'arbre en temps $O(\log n)$ avec $O(n)$ processeurs en utilisant les techiques vues en cours sur le calcul d'opérateurs associatifs. Le théorème de Brent nous permet d'ailleurs de réduire le nombre de processeurs $O\left(\frac{n}{\log n}\right)$ processeurs. On peut aussi mettre en œuvre cette réduction en partageant la séquence initiale de longueur 2^k en $\frac{2^k}{k}$ séquences de longueurs k. Chaque séquence est associée à un processeur qui construit séquentiellement le sous arbre correspondant en temps $O(k)$. Ensuite, on applique l'algorithme parallèle de construction de l'arbre qui permet de déterminer la valeur de tous les nœuds en temps $O(\log(\frac{n}{\log n})) = O(\log n)$ avec $O(\frac{n}{\log n})$ processeurs. □

▹ **Question 4.** En déduire un algorithme séquentiel pour calculer en temps $O(\log n)$ la fonction «match» sur une parenthèse ouvrante donnée.

Réponse. La construction de la fonction match est un peu délicate. L'idée est de parcourir l'arbre que nous venons de construire de la parenthèse ouvrante jusqu'à la parenthèse fermante correspondante. En parcourant l'arbre il est en effet possible de connaître (en phase de remontée) la séquence irréductible correspondant aux feuilles du sous arbre situées à droite de la parenthèse ouvrante correspondante. S'il y a suffisamment de parenthèses fermantes dans le frère droit du nœud considéré pour fermer la séquence irréductible obtenue, alors la parenthèse fermante correspondante est dans le sous arbre enraciné en le frère droit et on peut entamer une phase de descente. Plus formellement, les algorithmes de remontée et de descente sont donnés aux figures 2 et 3.

```
REMONTÉE
Si le nœud courant est un fils droit Alors on passe au père
Sinon
    Si zR(val(frère droit)) commence par ) Alors
        Aller sur le frère droit
        Attaquer la phase de descente.
    Sinon
        z = zR(val(frère droit))
        Aller sur le père
```

ALGO. 2: Phase de remontée dans l'arbre.

```
DESCENTE
Si zR(val(fils gauche)) commence par ) Alors
    Aller sur le fils gauche
Sinon
    z = zR(val(fils gauche))
    Aller sur le fils droit
```

ALGO. 3: Phase de descente dans l'arbre.

La phase de remontée et la phase de descente durent au plus $O(\log n)$ étapes donc la complexité séquentielle de cet algorithme est $O(\log n)$ pour une parenthèse. Notons que si on dispose de $O(n)$ processeurs, il est facile d'exécuter cet algorithme en parallèle pour chaque parenthèse (ce qui ne nécessite que des lectures concurrentes). On peut ainsi réaliser le calcul de la fonction en temps $O(\log n)$ avec $O(n)$ processeurs. □

▹ **Question 5.** Montrer comment construire la fonction match pour toutes les parenthèses ouvrantes en temps $O(\log n)$ avec $O(\frac{n}{\log n})$ processeurs. L'approche benête ne fonctionnant pas, on pourra supposer que la longueur de la séquence de parenthèses initiale est 2^k et que k divise 2^k (quitte à rajouter quelques parenthèses, c'est toujours possible). Cela permettra de réduire des séquences disjointes de longueur k, de calculer la fonction match sur certaines parenthèses de ces séquences puis, en utilisant les résultats, de calculer la fonction match sur l'intégralité des parenthèses.

Réponse. Nous avons vu dans la question précédente comment calculer la fonction match en temps $O(\log n)$ avec $O(n)$ processeurs. Le passage de $O(n)$ processeurs à $O\left(\frac{n}{\log n}\right)$ est assez délicat. Nous omettrons la preuve détaillée de quelques points. L'algorithme propose comporte 3 grandes étapes.

(1) Chacun des $\frac{2^k}{k}$ processeurs réduit une séquence de longueur de longueur k en temps $O(k)$. On obtient ainsi $\frac{2^k}{k}$ séquences de la forme $)^i(^j$ et de longueur au plus k.
(2) Dans chaque séquence irréductible, on marque la première parenthèse ouvrante et la dernière parenthèse fermante (qui sont dans le cas général en position i et $i+1$) et on calcule la fonction match sur chacune de ces parenthèses. On a donc au plus $2 \cdot \frac{2^k}{k}$ fonctions match a calculer, ce qui peut se faire en temps $O\left(\log\left(\frac{2^k}{k}\right)\right) = O(k)$ avec $O(\frac{2^k}{k})$ processeurs. On marque également les parenthèses correspondantes aux parenthèses marquées. En tout, on obtient donc au plus $4 \cdot \frac{2^k}{k}$ parenthèses marquées par séquence irréductible initiale.
(3) Dans chaque séquence irréductible initiale, on définit des intervalles droit et gauche comme suit :
   - Un intervalle gauche est un ensemble de parenthèses ouvrantes consécutives entre deux parenthèses marquées.
   - Un intervalle droit est un ensemble de parenthèses fermantes consécutives entre deux parenthèses marquées.

   On obtient ainsi au plus 5 intervalles par séquence irréductible initiale, soit au plus $5 \cdot \frac{2^k}{k}$ intervalles en tout. Il est relativement facile (mais assez technique) de montrer que si on remplace les intervalles gauches par des [et les intervalles droits par des] on obtient une séquence correctement crochetée. Enfin, si $match(l) = r$, où l est la première parenthèse ouvrante d'un intervalle gauche et r est la dernière parenthèse d'un intervalle droit, alors les intervalles gauche et droit correspondant ont la même longueur k et si

$$\left.\begin{array}{l} IG(l) = < l, l+1, \dots, l+k > \text{ et} \\ ID(r) = < r-k, \dots, r-1, r > \end{array}\right\} \text{ alors } \forall i \in [1,k], \quad match(l+i) = r - i.$$

   Chaque processeur peut donc alors calculer en séquentiel la fonction match pour les au plus $O(\log n)$ parenthèses de la séquence irréductible initiale à laquelle il est attaché en temps $O(\log n)$.

Il est donc possible de calculer la fonction match en temps $O(\log n)$ avec seulement $O\left(\frac{n}{\log n}\right)$ processeurs. □

On est maintenant en mesure de proposer un algorithme pour construire l'arbre d'évaluation d'une formule simple. On rappelle que l'arbre d'évaluation correspondant à la formule simple

$$((E_1)op_1(E_2)\dots op_{k-1}(E_k))$$

est indiqué à la figure 3.

▹ **Question 6.** Proposer un algorithme qui construit l'arbre d'évaluation d'une expression simple en temps $O(\log n)$ avec $O\left(\frac{n}{\log n}\right)$ processeurs.

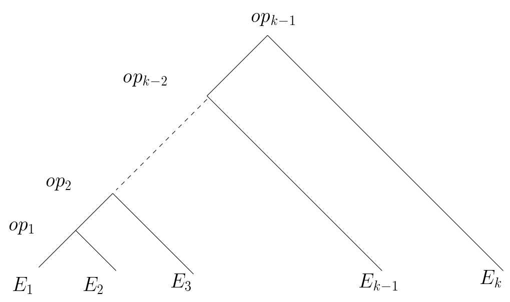

FIG. 3 – Arbre d'évaluation correspondant à une expression simple.

Réponse. En fait, on va construire l'arbre d'évaluation en temps $O(1)$ avec $O(n)$ processeurs, qu'on simulera ensuite en temps $O(\log n)$ avec $O\left(\frac{n}{\log n}\right)$ processeurs.
Les nœuds de l'arbre d'évaluation sont des opérateurs et les séquences de texte autour d'un opérateur sont de la forme) *op* (. Il suffit donc de trouver le fils gauche (qui est soit à gauche de la parenthèse correspondante à la parenthèse fermante située juste avant l'opérateur, soit le dernier opérateur de l'expression simple qui se finit en cette parenthèse fermante) et le fils droit (qui est soit l'expression simple élémentaire correspondant à la parenthèse ouvrante située juste après l'opérateur, soit le dernier opérateur de cette expression simple.).
Plus formellement, cet algorithme est spécifie à la figure 4. Il est facile de vérifier que cet algorithme n'utilise

CONSTRUCTION
Pour tout i **en parallèle**
  Si i est un opérateur **Alors**
    Si $match(i-1)-1 = op$ **Alors**
      fg(i)=match(i-1)-1
    Sinon
      Si $i-2 =)$ **Alors** $fg(i) = match(i-2)-1$
      Sinon fg(i)=i-2
    Si $match(i+1)-1 =)$ **Alors**
      fd(i)=match(match(i+1)-1)-1
    Sinon fd(i)=match(i+1)-1

ALGO. 4: Construction de l'arbre d'évaluation.

que des lectures concurrentes (puisque chaque processeur ne met à jour que les pointeurs associés à l'opérateur qu'il traite). Il réalise la construction de l'arbre en temps $O(1)$ avec $O(n)$ processeurs, ce qui peut facilement être simule en temps $O(\log n)$ avec $O\left(\frac{n}{\log n}\right)$ processeurs. □

Nous avons donc montré qu'il est possible de construire l'arbre d'évaluation d'une expression quelconque correctement parenthésée de longueur n en temps $O(\log n)$ avec $O\left(\frac{n}{\log n}\right)$ processeurs.

Dans les questions suivantes, nous allons montrer comment évaluer l'expression en temps $O(\log n)$ avec $O(n)$ processeurs. Il est possible de réaliser cette évaluation en temps $O(\log n)$ avec $O\left(\frac{n}{\log n}\right)$ processeurs mais cela sort du cadre de ce devoir.

3. ÉVALUATION DE L'EXPRESSION

Le principe de l'évaluation est le suivant. À chaque nœud v de l'arbre d'évaluation, on va associer un nœud $cond(v)$. Au départ, pour tout nœud v on pose $cond(v) = v$. Un nœud est dit marqué si on sait évaluer la valeur de l'expression du sous arbre enraciné en ce nœud. Dès qu'un des fils d'un nœud est marqué, on dit que le nœud est actif. À tout moment, si un nœud est actif, la connaissance de la valeur de $cond(v)$ doit être suffisante au calcul de la valeur de v . Pour cela, on associe à chaque nœud actif une fonction de la forme $f(x) = \frac{ax+b}{cx+d}$, où x représente la valeur de $cond(v)$. Pour obtenir une évaluation rapide de l'expression, l'idée est de faire du saut de pointeur sur $cond(v)$, ce qui nécessite de composer les fonctions associées au nœud.

▷ **Question 7.** Montrer qu'il est possible de calculer la composée de deux fonctions associées à deux nœuds en temps constant et qu'il est possible d'évaluer la fonction en temps constant dès que $cond(v)$ est marqué. Proposer des fonctions Activer, Sauter et Marquer.

Réponse. Les fonctions Activer, Sauter et Marquer sont définies à la figure 5.

Activer
Pour tout nœud v **en parallèle**
    Si v n'est pas actif **Et** exactement un fils de v est marqué **Alors**
        $cond(v)$ devient l'autre fils
        $f_v(x) = x$ op la valeur du fils marqué
    Si v n'est pas actif **Et** ses 2 fils sont marqués **Alors**
        $cond(v)$ devient l'un des fils aléatoirement
        $f_v(x) = x$ op la valeur de l'autre fils marqué

Sauter
Pour tout nœud v **en parallèle**
    $cond(v) = cond(cond(v))$
    $f_v(x) = f_v(f_{cond(v)}(x))$

Marquer
Pour tout nœud v **en parallèle**
    Si $cond(v)$ est marqué **Alors** Marquer v
    Évaluer $f_v(x)$.

Algo. 5: Activer, Sauter et Marquer.

Il faut maintenant vérifier qu'on est capable de composer deux fonctions f_v et d'évaluer f_v quand v devient marqué en temps constant. Pour cela, nous allons montrer par récurrence que f_v est toujours de forme $f_v(x) = \frac{ax+b}{cx+d}$. En effet, au moment où f_v est initialisée, on a $f_v(x) = x$ op a, où a est la valeur (déjà évaluée) d'un fils de v. Ensuite si

$$f_v(x) = \frac{a_1x + b_1}{c_1x + d_1} \text{ et } f_{cond(v)}(x) = \frac{a_2x + b_2}{c_2x + d_2},$$

alors

$$f_v(f_{cond(v)}(x)) = \frac{a_3x + b_3}{c_3x + d_3},$$

où

$$a_3 = (a_1a_2 + b_1c_2), \quad b_3 = (a_1b_2 + b_1d_2), \quad c_3 = (c_1a_2 + d_1c_2) \text{ et } d_3 = (c_1b_2 + d_1d_2).$$

Le coût de la composition est donc de 8 multiplications et de 4 additions et le coût de l'évaluation, quand $cond(v)$ est marqué c'est a dire quand x est connu, est de 2 multiplications, 2 additions et une division.
En conclusion, il n'y a pas de complexification des fonctions f_v pendant le déroulement de l'algorithme, et les trois fonctions Activer, Sauter et Marquer peuvent être exécutées en temps $O(1)$ avec $O(n)$ processeurs. □

On définit une étape de l'algorithme d'évaluation de la fonction par la séquence (Marquer,Activer,Sauter,Sauter). Dans la suite, on va montrer qu'après $\log n$ étapes, tous les nœuds de l'arbre d'évaluation sont marqués, et donc en particulier que la racine contient la valeur de l'expression. On aura donc ainsi produit un algorithme qui évalue une expression quelconque en temps $O(\log n)$ avec $O(n)$ processeurs, puisque chaque étape est atomique avec $O(n)$ processeurs.

Définition 5. Soit x un nœud de l'arbre d'évaluation. T_x désigne le sous arbre enracine en x et $size(x)$ représente le nombre de feuilles de T_x. Si $y \in T_x$, alors $size(x/y) = size(x) - size(y)$.

Pour démontrer la correction de l'algorithme, on a utiliser les deux invariants suivants

Définition 6 ($(I1)_k$). Après k étapes de l'algorithme, si $size(x) \leqslant 2^k$ alors le nœud x est marqué.

Définition 7 ($(I2)_k$). Soit x un nœud de l'arbre. Après k étapes de l'algorithme
- soit $size(x/cond(x)) \geqslant 2^k$,
- soit aucun de fils de $cond(x)$ n'est marqué,
- soit $cond(x)$ est une feuille.

▷ **Question 8.** Montrer $(I1)_k$ en utilisant $(I1)_{k-1}$ et $(I2)_{k-1}$.

Réponse. Soit x un nœud tel que $size(x) \leqslant 2^k$. Nous allons d'abord montrer que chaque nœud de T_x a un fils marqué après la $k^{\text{ième}}$ étape.
Soit y un nœud de T_x. Puisque $size(x) \leqslant 2^k$, $size(y) \leqslant 2^k$ et donc au moins un des fils z de y vérifie $size(z) \leqslant 2^{k-1}$ (par un simple argument de dénombrement). Avec $(I1)_{k-1}$, on peut donc conclure que z est marqué et que tout nœud de T_x a un fils marqué. En particulier, puisque $cond(x)$ appartient a T_x, $cond(x)$ a un fils marqué.
On utilise maintenant $(I2)_{k-1}$. Puisque $cond(x)$ a un fils marqué, on sait que soit $cond(x)$ est une feuille a la fin de l'étape $k-1$, soit $size(x/cond(x)) \geqslant 2^{k-1}$.
Si $cond(x)$ est une feuille a l'étape $k-1$, alors x est marqué dans la phase de marquage de l'étape k.
Si $size(x/cond(x)) \geqslant 2^{k-1}$ a la fin de l'étape $k-1$, alors $size(cond(x)) = size(x) - size(x/cond(x)) \leqslant 2^{k-1}$ a la fin de l'étape $k-1$ et donc $cond(x)$ était marqué a la fin de l'étape $k-1$ et x a été marqué dans la phase de marquage de l'étape k.
Dans tous les cas, on vérifie donc que si $size(x) \leqslant 2^k$, alors x est marqué a la fin de l'étape k. □

▷ **Question 9.** Montrer $(I2)_k$ en utilisant $(I1)_{k-1}$ et $(I2)_{k-1}$.

Réponse. Soit $y = cond(x)$ a la fin de l'étape $k-1$. On sait d'après l'hypothèse $(I2)_{k-1}$ appliquée a x que soit y est une feuille, soit $size(x/y) \geqslant 2^{k-1}$, soit aucun fils de y n'est marqué.

- Si y est une feuille a la fin de l'étape $k-1$, alors $cond(x)$ est toujours égal a y a la fin de l'étape k et donc $cond(x)$ est une feuille a la fin de l'étape k.

- Supposons maintenant que $size(x/y) \geqslant 2^{k-1}$ a la fin de l'étape $k-1$ et posons $z = cond(y)$ a la fin de l'étape $k-1$. Appliquons l'hypothèse $(I2)_{k-1}$ a y.
  - Supposons $size(y/z) \geqslant 2^{k-1}$ a la fin de l'étape $k-1$. On sait que deux phases de saut se déroule a la $k^{\text{ième}}$ étape, donc $cond(x)$ est un descendant (au sens large) de z après l'étape k. On en déduit en particulier que $size(cond(x)) \leqslant size(z)$ a la fin de l'étape k. Or,
  $$size(x/cond(x)) = size(x/y) + size(y/cond(x)) \geqslant size(x/y) + size(y/z) \geqslant 2^k.$$
  On vérifie donc que dans ce cas $(I2)_k$ est vérifiée.
  - Supposons que z est une feuille a la fin de l'étape $k-1$. Alors $cond(x)$ devient égal a z après un saut dans l'étape k et n'est pas affecte par le deuxième saut, donc $cond(x)$ est une feuille a la fin de l'étape k et $(I2)_k$ est vérifiée.
  - Supposons qu'aucun fils de z n'est marqué après l'étape $k-1$. Soient z_1 et z_2 les fils de z.
  Si aucun fils de z n'est marqué après l'étape k, alors $cond(z) = z$ et $cond(x) = cond(z) = z$ a la fin de l'étape k. Dans ce cas $(I2)_k$ est bien vérifiée.
  Si z_1 est marqué a la fin de l'étape k, alors $cond(z)$ devient z_2 après l'activation de l'étape k et $cond(x)$ est un descendant note z_3 de z_2 a la fin de l'étape k. Par construction, on vérifie que
  $$size(x/z_3) \geqslant size(x/y) + size(z_1),$$
  puisque dans les feuilles de T_x qui ne sont pas dans T_{z3}, on trouve toutes les feuilles qui ne sont pas dans T_y et toutes les feuilles qui sont dans T_y mais pas dans T_{z2}. Puisque z_1 n'était pas marqué a la fin de l'étape $k-1$, on sait en plus que $size(z_1) > 2^{k-1}$. On vérifie donc que
  $$size(x/cond(x)) > 2^{k-1} + 2^{k-1} = 2^k,$$
  et donc $(I2)_k$ est bien vérifiée. Si $size(y/z) \geqslant 2^{k-1}$ a la fin de l'étape $k-1$, on vérifie donc que $(I2)_k$ est vérifiée.

– Supposons maintenant que y n'est pas une feuille et qu'aucun des fils de y n'est marqué à la fin de l'étape $k-1$.
Soient z_1 et z_2 les fils de y.
Si aucun des fils n'est marqué à la fin de l'étape k, alors $(I2)_k$ est vérifiée.
Supposons que z_1 soit marqué à la fin de l'étape k. A la fin de l'activation de l'étape k, $cond(y)$ devient égal a z_2. Puisque z_1 n'était pas marqué à la fin de l'étape $k-1$, on sait que $size(z_1) > 2^{k-1}$ et donc $size(x/z_2) > size(y_1) > 2^{k-1}$. Supposons qu'après l'étape $k-1$, $cond(z_2) = v$ et $cond(v) = w$. La discussion se fait ensuite sur la condition de $(I2)_{k-1}$ vérifiée par v (la situation générale dans cette situation est donnée dans la figure 4). Dans tous les cas, on vérifie qu'après l'étape k, on a $cond(x) = w$.
  – Si $cond(v/w) \geqslant 2^{k-1}$, alors
$$size(x/cond(x)) \geqslant size(z_1) + size(v/w) \geqslant 2^{k-1} + 23^{k-1} \geqslant 2^k.$$
  et $(I2)_k$ est donc bien vérifiée.
  – Si $cond^{(k-1)}(v) = w$ est une feuille, alors $cond^{(k)}(x) = w$ est une feuille et $(I2)_k$ est donc bien vérifiée.
  – Supposons maintenant (et pour finir...) qu'aucun fils de $cond(v)$ n'est marqué après l'étape $k-1$. Comme précédemment, si aucun fils de $cond(v)$ n'est marqué après l'étape k, $(I2)_k$ est bien vérifiée, et dans le cas contraire, si v_1 et v_2 sont les fils de v et que v_1 est marqué durant l'étape k, alors on sait que $size(v_1) > 2^{k-1}$ et que $cond(x) = w \in T_{v2}$.
  On en déduit que
$$size(x/cond(x)) \geqslant size(z_1) + size(v_1) \geqslant 2^k,$$
  ce qui conclue la preuve.

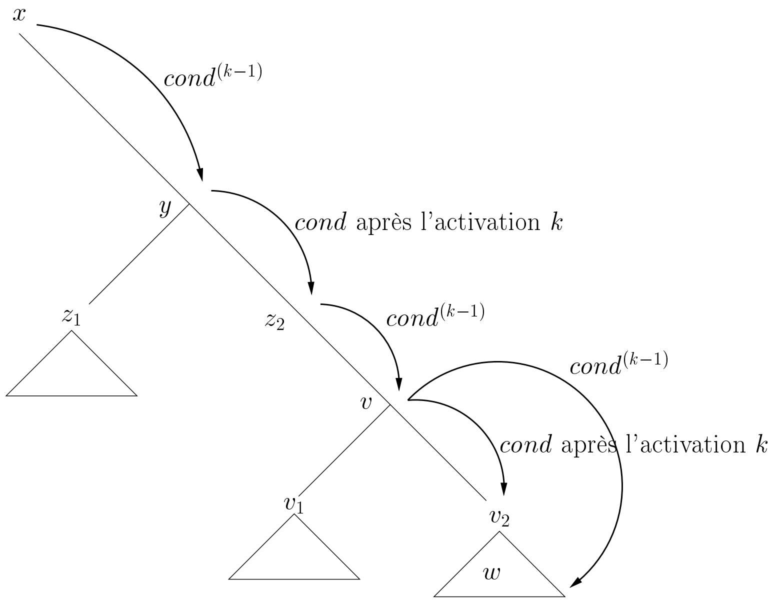

FIG. 4 – Situation correspondant a la dernière discussion.

□