Mesurer la hauteur d'un arbre

Jean-Christophe Filliâtre

Mesurer la hauteur d'un arbre

Jean-Christophe Filliâtre

CNRS

Résumé

Dans cet article, nous nous intéressons au problème du calcul de la hauteur d'un arbre. Le problème a l'air plutôt simple, à priori, puisqu'il suffit de suivre la définition mathématique avec une simple fonction récursive de quelques lignes. Néanmoins, une telle fonction peut facilement faire déborder la pile d'appels. Après avoir laissé le lecteur réfléchir à une solution, nous en discutons plusieurs, notamment au regard de ce qu'offre le langage de programmation. Ce problème illustre la difficulté qu'il peut y avoir à se passer de récursivité.

1 Le problème

Chacun sait que le soleil et le théorème de Thalès peuvent avantageusement être utilisés pour mesurer la hauteur d'un arbre, à l'instar de ce que Thalès lui-même fit pour mesurer la hauteur de la pyramide de Khéops. Le lecteur, cependant, aura compris qu'il s'agit plutôt ici d'écrire un programme informatique calculant la hauteur d'une structure de données arborescente.

Pour fixer le problème précisément, supposons qu'il s'agisse d'arbres binaires immuables. L'information contenue dans les nœuds ne nous intéresse pas ; on suppose cependant qu'elle ne stocke pas déjà la hauteur de chaque sous-arbre, sans quoi le problème serait trivial. Dans notre langage de programmation préféré, OCaml, un type pour de tels arbres peut être le suivant.

```
type tree = E | N of tree * tree
```

La hauteur est définie comme le nombre maximal de nœuds le long d'un chemin de la racine à une feuille (voir par exemple Knuth [5, Sec. 2.3]). De façon équivalente, on peut suivre la définition récursive du type `tree` pour définir ainsi la hauteur $h(t)$ d'un arbre t :

$$\begin{aligned} h(\texttt{E}) &= 0, \\ h(\texttt{N}(l,r)) &= 1 + \max(h(l), h(r)). \end{aligned}$$

La question qui nous intéresse ici est d'écrire une fonction `height`, de type `tree -> int`, recevant un arbre en argument et renvoyant sa hauteur. Il paraît évident de suivre la définition mathématique, c'est-à-dire d'écrire ce programme :

```
let rec height = function
  | E        -> 0
  | N (l, r) -> 1 + max (height l) (height r)
```

Cependant, cette solution a le défaut de faire déborder la pile d'appels pour un arbre dont la hauteur serait trop grande [1], ce qui est très facilement atteint avec des arbres qui seraient très linéaires, même avec peu de mémoire. Une telle situation n'est pas acceptable, notamment lorsqu'une fonction comme `height` fait partie d'une bibliothèque et que son auteur ne peut donc

1. Sur notre machine, cette valeur est de l'ordre de 275 000. Elle s'explique par une taille de pile de 8 Mo et des tableaux d'activation de 32 octets chacun (une adresse de retour, deux valeurs sauvegardées sur la pile et un alignement de chaque tableau sur 16 octets).

pas préjuger de son utilisation ni des conditions de son exécution au regard de la taille de la pile. Le problème que nous posons ici est donc de proposer une alternative à la fonction `height`, ayant le même type et la même spécification, mais s'exécutant en espace de pile constant.

Le lecteur est vivement invité à interrompre ici sa lecture
et à chercher une solution à ce problème.

Au delà du problème spécifique de la hauteur d'un arbre, c'est le problème plus général d'un éventuel débordement de pile par une fonction récursive qui nous intéresse. On entend souvent l'argument « il suffit d'utiliser une pile », justifié par le fait que c'est exactement ce que fait le compilateur. Mais lorsqu'il s'agit de le faire en pratique, c'est rapidement difficile. Si le lecteur a pris le temps de chercher une solution à notre problème, il doit maintenant en être convaincu.

Si nous avons choisi la hauteur d'un arbre plutôt que sa taille, c'est justement pour qu'il ne soit pas immédiat de simplement déposer des sous-arbres sur une pile, en accumulant la taille dans une variable globale. Le fait de devoir calculer un maximum *après* le calcul de la hauteur des deux sous-arbres est justement ce qui rend le problème intéressant. On peut analyser ainsi les raisons de cette difficulté : lorsque le compilateur utilise la pile d'appels pour compiler notre fonction récursive `height`, il y dépose des données (un sous-arbre dont la hauteur reste à calculer, une hauteur déjà calculée) mais également du contrôle, sous la forme d'*adresses de retour*. Lorsqu'on utilise une structure de pile explicite, il est facile d'y déposer des données mais moins évident d'y déposer du contrôle.

Dans cet article, nous présentons différentes solutions à notre problème. Certaines sont spécifiques au calcul de la hauteur d'un arbre (Section 2) et d'autres s'appliquent en revanche à toute fonction récursive ou toute structure arborescente (Section 3). Nous en comparons les performances (Section 4) avant de considérer quelques variantes du problème (Section 5) et de conclure. Tous les programmes décrits dans cet article sont accessibles sur la page https://www.lri.fr/~filliatr/hauteur/.

2 Des solutions ad hoc

Dans cette section, nous présentons deux premières solutions, spécifiques au calcul de la hauteur d'un arbre. Nous verrons dans la section 4 qu'elles sont particulièrement efficaces.

Parcours en largeur. Certains lecteurs auront sûrement imaginé utiliser un parcours en largeur et c'est là une excellente solution, concise et efficace. Un tel parcours en largeur peut être réalisé par une fonction qui manipule un entier `m` et deux listes, une liste `curr` contenant des nœuds de l'arbre situés à la profondeur `m` et une liste `next` des nœuds situés à la profondeur `m`+1. Lorsque la première liste vient à se vider, on incrémente `m` et on échange les deux listes. Lorsque les deux listes sont vides, la valeur de `m` est renvoyée.

```
let rec hbfsaux m next = function
  | []               -> if next=[] then m else hbfsaux (m+1) [] next
  | E       :: curr -> hbfsaux m next         curr
  | N (l, r) :: curr -> hbfsaux m (l::r::next) curr
```

Il n'y a pas lieu d'utiliser de file dans ce parcours en largeur, car l'ordre du parcours des nœuds d'une même profondeur ne nous intéresse pas. Il s'avère que l'on réalise ici un *boustrophédon* (une alternance de parcours de la gauche vers la droite puis de la droite vers la gauche). Il ne reste plus qu'à démarrer le parcours en largeur avec une liste `curr` contenant l'arbre tout entier, une liste `next` qui est vide et un accumulateur valant 0.

```
let hbfs t = hbfsaux 0 [] [t]
```

Il est important de noter ici que les trois appels récursifs à `hbfsaux` sont *terminaux*. Le compilateur OCaml optimisant de tels appels, en les remplaçant par des sauts [2], cette solution s'exécute bien en espace constant. Bien entendu, il serait très facile de réécrire la fonction `hbfsaux` avec une boucle `while`, même si elle serait sans doute rendue un peu moins élégante par la présence de références.

Comme on le voit, les arbres vides `E` sont ajoutés dans la liste `next` comme les autres, puis ignorés par le deuxième cas du filtrage dans `hbfsaux`. On peut modifier le code pour ne jamais ajouter d'arbre `E` dans les listes manipulées. On perd en lisibilité mais le programme obtenu est 40% plus rapide.

Avec une pile. Une autre solution ad hoc consiste à utiliser une pile contenant des sous-arbres, où chaque sous-arbre est accompagné de sa profondeur dans l'arbre original. On a donc une pile contenant des paires. Outre cette pile, le code maintient dans un accumulateur `m` le maximum des profondeurs rencontrées.

```
let rec hstackaux m = function
  | []                 -> m
  | (n, E)        :: s -> hstackaux (max m n) s
  | (n, N (l, r)) :: s -> hstackaux m         ((n+1,l) :: (n+1,r) :: s)
```

Comme le lecteur le constate, il s'agit maintenant d'un parcours en profondeur. Là encore, les deux appels récursifs sont terminaux et on a donc bien une solution en espace constant. Pour calculer la hauteur d'un arbre t, il suffit de démarrer avec la paire $(0, t)$.

```
let hstack t = hstackaux 0 [0, t]
```

Bien que plus concise encore que la précédente, cette solution est bien moins efficace (plus de deux fois plus lente). L'une des raisons est une sollicitation plus importante du GC, car on alloue deux fois plus de blocs (deux blocs par nœud maintenant). On peut y remédier avec un type ad hoc de listes de paires, comme celui-ci :

```
type stack = Nil | Cons of int * tree * stack
```

On obtient alors une version 30% plus rapide. Elle reste cependant moins performante que `hbfs`, notamment parce qu'on y fait beaucoup plus d'arithmétique : deux additions par nœud et des appels à `max`, là où `hbfs` ne fait qu'une addition par changement de niveau. En expansant la fonction `max`, et surtout en évitant d'empiler des arbres vides, comme nous l'avions fait pour `hbfs`, on obtient au final un programme très efficace.

3 Des solutions génériques

Dans cette section, nous présentons des solutions plus générales, qui fonctionnent pour toute fonction récursive ou pour le parcours de toute structure récursive.

2. On peut le vérifier en examinant le code assembleur produit par `ocamlopt`, avec l'option `-S` de ce dernier.

Passage de continuations. Une solution élégante consiste à adopter un style *par continuation* (en anglais CPS, pour *Continuation-Passing Style* [8]). L'idée est de généraliser le problème, en ne calculant pas la hauteur $h(t)$ de l'arbre t, mais $k(h(t))$ pour une fonction k passée en argument, appelée « continuation ». La solution s'en déduira au final en prenant pour k la fonction identité. Mais le fait d'avoir généralisé le problème va le rendre plus simple, comme souvent en mathématiques. D'une part, il est à peine plus complexe d'écrire une telle fonction. La voici :

```
let rec hcpsaux t k = match t with
  | E        -> k 0
  | N (l, r) -> hcpsaux l (fun hl ->
                hcpsaux r (fun hr -> k (1 + max hl hr)))
```

D'autre part, et c'est là tout l'intérêt, on note que *tous* les appels faits dans cette fonction sont maintenant des appels terminaux, aussi bien les deux appels récursifs à `hcspaux` que les deux appels à `k`. Dès lors, cette fonction s'exécutera bien en espace de pile constant, tout comme la fonction `hcps` qu'on en déduit.

```
let hcps t = hcpsaux t (fun h -> h)
```

Dans cette solution, toute l'information nécessaire au calcul est contenue dans les *clôtures* allouées sur le tas. Ainsi, la clôture correspondant à `fun hl ->` capture les valeurs de `r` et `k` et la clôture correspondant à `fun hr ->` capture les valeurs de `hl` et `k`. En particulier, comme chaque clôture capture la valeur d'une autre continuation `k`, les clôtures forment une liste chaînée. Cette liste se termine avec la continuation identité (`fun h -> h`) qui a été passée en argument initialement.

Bien entendu, une telle solution suppose deux choses : que notre langage propose des fonctions de première classe et que les appels terminaux soient correctement optimisés. Dans certains langages, on ne dispose tout simplement pas de fonction de première classe. Un exemple est le langage C. Dans d'autres langages, on dispose de fonctions de première classe mais les appels terminaux ne sont pas optimisés ou ne le sont pas tous. C'est le cas des langages Java ou Kotlin [3], pour n'en citer que deux.

Lorsque le langage ne propose pas de fonction de première classe, ou que son compilateur n'optimise pas les appels terminaux, le programmeur peut néanmoins utiliser la solution du passage de continuations. Il lui suffit de *défonctionnaliser* le programme [7, 2], c'est-à-dire de remplacer les clôtures par un type ad hoc. Ici, ce type fait la somme des trois continuations différentes (les trois constructions `fun` du code ci-dessus).

```
type cont = Kid | Kleft of tree * cont | Kright of int * cont
```

On reconnaît bien là une structure de liste chaînée, comme identifiée plus haut. Il faut maintenant écrire *deux* fonctions, l'une qui est la version défonctionnalisée de `hcpsaux` et une autre pour appliquer une continuation à un argument.

3. En Kotlin, seuls les appels terminaux *récursifs* sont optimisés, à la demande explicite du programmeur, mais un appel récursif fait à l'intérieur d'une clôture ne le sera pas. Dans le cas de notre fonction `hcpsaux`, cela veut dire qu'un seul des quatre appels sera optimisé, à savoir le premier appel récursif à `hcpsaux`.

```
let rec hdefunaux t k = match t with
  | E        -> hdefuncont k 0
  | N (l, r) -> hdefunaux l (Kleft (r, k))
and hdefuncont k v = match k with
  | Kid            -> v
  | Kleft (r, k)   -> hdefunaux r (Kright (v, k))
  | Kright (hl, k) -> hdefuncont k (1 + max hl v)
```

Comme on le voit, il s'agit de deux fonctions mutuellement récursives, où tous les appels sont toujours des appels terminaux. Il ne reste plus qu'à appeler `hdefunaux` avec la continuation `Kid`.

```
let hdefun t = hdefunaux t Kid
```

Si le compilateur n'optimise pas correctement l'appel terminal, on peut modifier encore une fois le programme pour remplacer ces deux fonctions mutuellement récursives par une boucle `while`. Le code en ligne qui accompagne cet article contient également cette version.

Le zipper. Une autre solution générique, s'appliquant au parcours d'une structure récursive quelconque, consiste à utiliser un *zipper* [4]. Il s'agit là d'une structure de données permettant de désigner un sous-terme d'une structure récursive et d'effectuer des déplacements locaux. Dans le cas d'un arbre binaire, qui nous intéresse ici, cela veut dire désigner un sous-arbre et se déplacer en descendant dans son sous-arbre gauche ou son sous-arbre droit ou en remontant au nœud parent.

Le *zipper* implémente un tel curseur comme une paire (p, t), où p est le chemin entre la racine et la position considérée et t le sous-arbre se trouvant à cette position. L'idée clé du *zipper* consiste à représenter le chemin depuis la position considérée jusqu'à la racine, plutôt que l'inverse. Ainsi, les déplacements s'opèrent en tête de liste, en temps constant. Voici un type `path` pour de tels chemins :

```
type path = Top | Left of path * tree | Right of tree * path
```

La constant `Top` représente la racine de l'arbre ; une valeur `Left`(p, t) (resp `Right`(t, p)) représente un sous-arbre gauche (resp. droit) dont le parent est désigné par le chemin p et dont le sous-arbre droit (resp. gauche) est t. On peut alors facilement définir de petites fonctions `left`, `right` et `up` pour se déplacer dans l'arbre (voir l'article de Huet [4]) et en déduire facilement un parcours de l'arbre sous la forme d'une boucle. Le code est donné en ligne. Comme pour les solutions précédentes `hcps` et `hdefun`, on a remplacé de l'espace utilisé sur la pile par de l'espace utilisé sur le tas, ici avec des valeurs du type `path`[4].

4 Performances

On donne ici une mesure des performances de ces diverses solutions. Pour chaque fonction, on mesure le temps total passé dans le calcul de la hauteur

— de tous les peignes à gauche de hauteur n avec $0 \leq n \leq 10\,000$;
— de tous les peignes à droite de hauteur n avec $0 \leq n \leq 10\,000$;
— de tous les arbres binaires parfaits de hauteur n avec $0 \leq n \leq 20$;
— de cent arbres construits aléatoirement avec $2000 \times n$ nœuds avec $0 \leq n < 100$.

4. Un lien peut être fait entre la version défonctionnalisée de la solution CPS et cette solution utilisant un *zipper* ; ceci est notamment discuté par Danvy [1].

La construction des arbres n'est pas inclue dans cette mesure. Les arbres aléatoires sont les mêmes pour chaque mesure. La mesure se fait en répétant cinq fois le calcul, puis en éliminant la valeur la plus petite et la valeur la plus grande, pour enfin faire la moyenne des trois valeurs restantes. Les mesures ont été faites avec OCaml version 4.07.1 sur un unique processeur Intel Core i7 1.90 GHz. Les temps sont donnés en secondes.

version	temps
`height`	1,28

version	temps
`hbfs`	0,94
`hbfs_opt`	0,58
`hstack`	2,32
`hstack_opt`	0,52

version	temps
`hcps`	2,55
`hdefun`	1,80
`hzipper`	3,75

À titre de comparaison, on a inclus la fonction `height` dans le tableau de gauche, même si elle n'est pas acceptable en pratique. Les versions appelées `_opt` sont celles qui évitent d'ajouter des arbres vides dans les listes. Les amoureux de la programmation fonctionnelle seront sûrement un peu déçus de voir que la version CPS est relativement peu efficace, même une fois défonctionnalisée.

5 Variantes

Arbre n-aire. Une première variante consiste à généraliser au calcul de la hauteur d'un arbre n-aire. Un type pour de tels arbres peut être le suivant :

```
type tree = N of tree list
```

On note qu'il n'y a plus d'arbre vide ; tout arbre contient maintenant au moins un nœud. Les programmes s'adaptent néanmoins très facilement. Ils sont donnés en ligne.

Arbre mutable. Une autre variante consiste à considérer le cas d'un arbre binaire *mutable*. Dans ce cas, on peut effectuer un parcours infixe de l'arbre *en espace constant*, en modifiant puis restaurant la structure de l'arbre pendant le parcours. Un tel algorithme est dû à Morris [6]. La figure 1 contient un programme C réalisant un tel parcours, modifié pour calculer et renvoyer la hauteur. Les explications sont dans les commentaires. Quoi de mieux pour célébrer le quarantième anniversaire de cet article merveilleux ! À titre de comparaison, les performances d'un tel programme — écrit en OCaml pour comparer des choses comparables — sont de $1,55$ s sur les tests décrits section 4.

On pourrait également utiliser l'algorithme de Schorr-Waite [3], un algorithme général pour parcourir un graphe en inversant les pointeurs puis en les restaurant. Mais l'algorithme de Schorr-Waite nécessite de pouvoir stocker un peu d'information dans chaque nœud, en l'occurrence un bit d'information par nœud dans le cas d'un arbre binaire, afin de retenir si on était descendu dans le sous-arbre gauche ou droit.

6 Conclusion

Dans cet article, nous avons exploré la difficulté qu'il peut y avoir à transformer un programme susceptible de faire déborder la pile d'appel en un programme s'exécutant en espace de pile constant. Nous avons pris l'exemple du calcul de la hauteur d'un arbre, car il contient toute la difficulté du problème tout en étant concis. Nous avons proposé de nombreuses solutions,

```c
typedef struct T* tree;
struct T { tree left, right; };

int height(tree t) {
  int d = 0, h = 0;
  while (t != NULL) {
    if (t->left == NULL) { // pas de sous-arbre gauche, on descend à droite
      t = t->right;
      h = max(h, ++d);
    } else {
      tree p = t->left;    // sinon, on cherche le prédécesseur p de t
      int delta = 1;
      while (p->right != NULL && p->right != t) {
        p = p->right;
        delta++;
      }
      if (p->right == NULL) { // on le trouve pour la première fois
        p->right = t;         // on le fait pointer sur t
        t = t->left;          // et on descend à gauche
        h = max(h, ++d);
      } else {                // on le trouve pour la seconde fois
        p->right = NULL;      // on restaure l'arbre
        t = t->right;         // et on descend à droite
        d -= delta;
      }
    }
  }
  return h;
}
```

FIGURE 1 – Calcul de la hauteur avec l'algorithme de Morris (en C).

certaines ad hoc, utilisant des parcours en largeur ou en profondeur, et d'autres génériques, utilisant une transformation CPS ou un *zipper*.

Bien évidemment, une solution plus simple encore consisterait à stocker la hauteur dans l'arbre, en la calculant (en temps constant) chaque fois qu'un nœud est construit. Et c'est évidemment ce qu'il faut faire si la hauteur doit être calculée souvent, par exemple lorsque l'on implémente des AVL.

Nous avons utilisé le langage OCaml pour la présentation des diverses solutions mais toute cette discussion n'est en rien spécifique à OCaml. Le problème du débordement de pile se présente en effet dans (presque) tous les langages de programmation. Les solutions que nous avons proposées s'adaptent plus ou moins facilement dans un autre langage, notamment selon que ce langage propose des fonctions comme valeurs de première classe et que son compilateur optimise les appels terminaux.

Enfin, il est important de faire remarquer que toutes les solutions que nous avons présentées dans cet article utilisent un espace $O(N)$ dans le pire des cas, où N est la taille de l'arbre. Si l'arbre occupe une grande partie de la mémoire, on n'a pas forcément le loisir d'utiliser un

espace aussi grand pour en calculer la hauteur. Le code en ligne accompagnant cet article inclut un programme, dû à Martin Clochard, qui calcule la hauteur en espace $\log(N)$. Son temps de calcul, en revanche, est prohibitif.

Remerciements. Je remercie toutes les personnes, collègues ou étudiants, qui se sont prêtées au jeu lorsque j'ai testé sur elles ce problème.

Références

[1] Olivier Danvy. Defunctionalized interpreters for programming languages. *SIGPLAN Not.*, 43(9) :131–142, September 2008.

[2] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, pages 162–174. ACM Press, 2001.

[3] David Gries. The schorr-waite graph marking algorithm. *Acta Inf.*, 11 :223–232, 1979.

[4] Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5) :549–554, September 1997.

[5] Donald E. Knuth. *The Art of Computer Programming, volumes 1–4A*. Addison Wesley Professional, 1997.

[6] Joseph M. Morris. Traversing binary trees simply and cheaply. *Inf. Process. Lett.*, 9(5) :197–200, 1979.

[7] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM.

[8] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3) :233–247, Nov 1993.