La supériorité de l'ordre supérieur

J.-C. Filliâtre[1]

1: Laboratoire de Recherche en Informatique,
Université Paris Sud,
91405 Orsay CEDEX, France
`filliatr@lri.fr`

Résumé

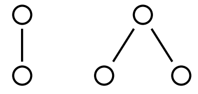
Nous présentons ici une écriture fonctionnelle de l'algorithme de Koda-Ruskey, un algorithme pour engendrer une large famille de codes de Gray. En s'inspirant de techniques de programmation par continuation, nous aboutissons à un code de neuf lignes seulement, plus élégant que les implantations purement impératives proposées jusqu'ici, notamment par Knuth. Dans un second temps, nous montrons comment notre code peut être légèrement modifié pour aboutir à une version de complexité optimale. Notre implantation en Objective Caml rivalise d'efficacité avec les meilleurs codes C. Nous détaillons les calculs de complexité, un exercice intéressant en présence d'ordre supérieur et d'effets de bord combinés.

1. Introduction

Le lieu commun qui veut que les langages de programmation fonctionnels soient moins efficaces que leurs homologues impératifs a la vie dure. Même s'il n'a plus lieu d'être, il faut encore souvent convaincre « par la pratique ». Dans cet article, nous montrons comment un algorithme complexe, jusqu'à présent uniquement présenté de manière impérative, peut être judicieusement écrit dans un langage où les fonctions sont des valeurs de première classe. Au delà de sa concision, le code obtenu s'avère également très efficace et beaucoup plus facile à certifier.

L'algorithme en question est dû à Y. Koda et F. Ruskey [7]. Le but de cet algorithme est d'énumérer les idéaux de certains ensembles finis partiellement ordonnés — ceux dont les diagrammes de Hasse sont des forêts — sous la forme de codes de Gray. Un code de Gray désigne de manière générale une énumération de mots où deux mots consécutifs ne diffèrent que d'un seul caractère. Un cas particulier classique est celui de l'énumération de tous les entiers binaires de $00\cdots0$ à $11\cdots1$ sous la forme d'un code de Gray, un problème largement abordé dans la littérature informatique. Les codes de Gray trouvent des applications dans des domaines aussi variés que les mathématiques, l'électronique, l'optique, l'ordonnancement, la fiabilité des réseaux, etc. Une section entière leur est d'ailleurs consacrée dans le quatrième volume à paraître de l'œuvre de Knuth *The Art of Computer Programming*. Une version préliminaire de cette section est actuellement disponible sur le site Internet de Knuth [5], et sa rédaction a amené son auteur à s'intéresser à l'algorithme de Koda-Ruskey, dont il propose deux implantations sur ce même site [4]. Ce sont ces pré-publications qui nous ont motivés.

L'algorithme de Koda-Ruskey peut être exposé simplement. On suppose donnée une forêt quelconque et l'on en cherche tous les coloriages possibles sous la forme d'un code de Gray. Un coloriage consiste à marquer les nœuds de noir ou de blanc, avec l'unique contrainte que les descendants d'un nœud blanc doivent nécessairement être blancs également. Ainsi, si l'on considère la forêt suivante



(1)

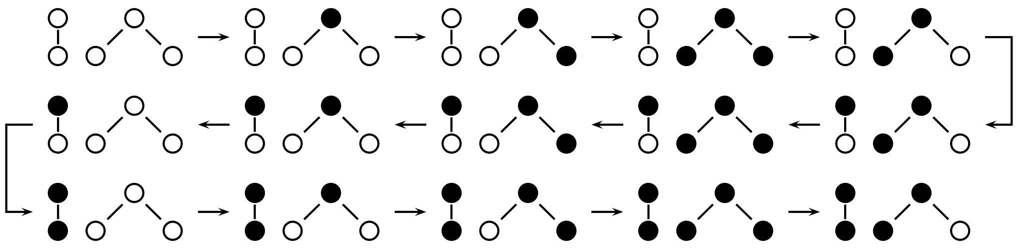
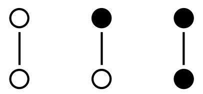


FIG. 1: *L'algorithme de Koda-Ruskey appliqué à la forêt* (1) .

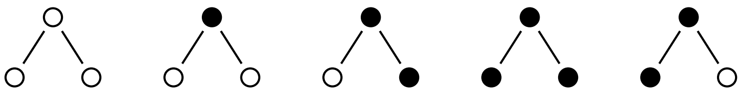
elle possède exactement 15 coloriages différents, qui sont donnés figure 1. Les coloriages forment un code de Gray si, par définition, ils apparaissent tous une fois et une seule, chacun différant du précédent par la couleur d'exactement un nœud.

Illustrons le principe de l'algorithme de Koda-Ruskey sur la forêt (1). L'idée principale est d'entremêler les coloriages des différents arbres se trouvant côte à côte dans une forêt. Il s'agit donc ici d'entremêler les trois coloriages du premier arbre (à deux nœuds), à savoir



(2)

avec les cinq coloriages du second arbre (à trois nœuds), à savoir



(3)

Ainsi, sur la première ligne de la figure 1 on a le premier coloriage du premier arbre et tous les coloriages du second ; sur la deuxième ligne, on a le deuxième coloriage du premier arbre et de nouveau tous les coloriages du second, mais cette fois-ci dans l'ordre inverse — il est clair que l'image miroir d'un code de Gray reste un code de Gray ; enfin, on a sur la dernière ligne le dernier coloriage du premier arbre et une dernière fois tous les coloriages du second, de nouveau dans l'ordre initial.

Reste à expliquer comment sont obtenus les coloriages d'un arbre : le premier est formé d'un arbre entièrement blanc ; les suivants sont obtenus en coloriant la racine en noir et en énumérant dessous tous les coloriages de la forêt formée par les arbres-fils. Le résultat est bien un code de Gray, car seule la couleur de la racine change entre les deux premiers arbres, puis la couleur de la racine n'est plus modifiée et les coloriages des fils forment un code de Gray par construction. Cette construction est illustrée ci-dessus en (2) et (3). On notera en particulier que le dernier coloriage d'un arbre n'est pas nécessairement celui où tous les nœuds sont noirs.

L'article original de Koda et Ruskey [7] décrit deux implantations de cet algorithme, sous forme de pseudo-code impératif. La première a pour complexité $O(nN)$, où n est le nombre de nœuds de la forêt et N le nombre de ses coloriages, c'est-à-dire la longueur du code de Gray à produire. La seconde implantation est un raffinement de la première pour obtenir une complexité $O(N)$, donc optimale. Un code Pascal est donné en annexe de l'article. Récemment, deux implantations de l'algorithme de Koda-Ruskey ont été données par Knuth [4], en CWEB[1]. Nous ne décrivons pas ici ces diverses implantations, car seul le principe même de l'algorithme est important, mais il est utile de savoir qu'il s'agit de codes relativement longs (entre 50 et 80 lignes) faisant un usage intensif de branchements

1. CWEB est un outil de programmation littéraire produisant d'une part du code C et d'autre part un document TEX.

inconditionnels (`goto`) ou de manipulation de pointeurs. À titre d'illustration, nous donnons en annexe le code C de la seconde implantation de Knuth.

Cet article décrit deux implantations de l'algorithme de Koda-Ruskey. Le langage de programmation utilisé, dans la syntaxe duquel sont écrits les fragments de code de cet article, est Objective Caml [2], mais ce pourrait être tout autre langage supportant les fonctions comme valeurs de première classe (tout autre dialecte de ML, Lisp, Scheme, etc.) La section 2 décrit une première version de notre programme. La section 3 montre comment celui-ci peut être légèrement modifié pour aboutir à une version de complexité optimale $O(N)$. Enfin, la section 4 compare les performances de nos programmes avec ceux proposés par Knuth.

2. Un code par continuations

Notre programme est disponible à l'adresse `http://www.lri.fr/~filliatr/software.fr.html`. D'un point de vue opérationnel, il est identique à celui de Knuth [4] : la forêt est donnée en entrée sur la ligne de commande et le code de Gray est affiché sous la forme de mots binaires, une fois dans un sens puis une autre fois dans le sens inverse. La forêt est donnée sous la forme de n paires de parenthèses correctement associées. Il y a en effet une bijection simple entre de telles paires de parenthèses et les forêts : une paire de parenthèses a pour descendants les paires de parenthèses qu'elle contient. Ainsi la forêt (1) est-elle représentée par les 5 paires de parenthèses suivantes :

```
(())(()())
```

Au démarrage, notre programme analyse cette chaîne de parenthèses pour construire la forêt correspondante, sous la forme d'une valeur du type Caml `forest` suivant :

```
type tree  = Node of int × forest
and forest = tree list
```

où α `list` est le type Caml prédéfini des listes d'éléments de type α. La liste contenant x_1, x_2, …,x_n dans cet ordre se note `[`x_1`; `x_2`; ...; `x_n`]` ; en particulier, la liste vide se note `[]`. L'adjonction d'un élément x au début d'une liste l se note $x :: l$.

Les n nœuds de la forêt sont annotés par les entiers $0, 1, \ldots, n-1$, selon un parcours postfixe. Les nœuds de la forêt (1) sont ainsi numérotés de la manière suivante :

```
1       4
|      / \
0     2   3
```

Afin d'obtenir en sortie le même code de Gray que les implantations de Knuth, les listes codant les forêts se présentent dans l'ordre inverse de celui donné par le parcours postfixe. La forêt ci-dessus est donc représentée de manière interne par la valeur Caml suivante :

```
[ Node (4, [Node (3, []); Node (2, [])]);
  Node (1, [Node (0, [])]) ]
```

Le programme ne stocke pas l'ensemble du code de Gray mais uniquement le coloriage courant de la forêt, dans un tableau `bits` de n entiers valant 0 (pour le blanc) ou 1 (pour le noir).

Nous en arrivons au codage de l'algorithme proprement dit, dont le principe a été exposé dans la section précédente. Les arbres et les forêts étant deux notions mutuellement récursives, nous avons naturellement deux fonctions mutuellement récursives `loop_t` et `loop_f` énumérant respectivement les

```
let rec loop_f k = function
  | [] → k ()
  | t :: f → loop_f (fun () → loop_t k t) f
and loop_t k (Node (i,f)) =
  if bits.(i) = 0 then begin
    k (); bits.(i) ← 1; loop_f k f
  end else begin
    loop_f k f; bits.(i) ← 0; k ()
  end
```

FIG. 2: *Un premier codage de Koda-Ruskey (C1).*

coloriages d'un arbre et d'une forêt. L'idée principale de notre programme est de donner un argument supplémentaire à ces deux fonctions, sous la forme d'une fonction `k` de type `unit → unit`, qui sera appelée pour chaque coloriage de l'arbre (resp. de la forêt). Cette fonction `k` peut être vue comme une continuation, et c'est ainsi que nous l'appellerons par la suite. L'intuition derrière la continuation `k` est qu'elle énumère les coloriages d'une certaine forêt `f0` ; dès lors, `(loop_f k f)` énumère les coloriages de la forêt `f0 @ f` et `(loop_t k t)` ceux de la forêt `f0 @ [t]`, où `@` dénote la concaténation des forêts.

Le code est donné figure 2, et nous l'appelons C1 par la suite. Nous allons maintenant le détailler en commençant par la fonction `loop_f`. Si la forêt est vide, alors on se contente d'appeler la continuation. Si en revanche la forêt contient au moins un arbre `t` au bord d'une forêt `f` alors on énumère les coloriages de `f`, en appelant récursivement `loop_f` sur `f`, mais avec cette fois-ci une continuation énumérant les coloriages de l'arbre `t` avec la continuation `k`. Ainsi, sur la forêt (1) contenant deux arbres t_1 et t_2, `loop_f` sera appelée initialement sur la liste $[t_2; t_1]$ avec une continuation initiale k_0, ce qui aura pour effet de rappeler `loop_f` sur la liste $[t_1]$ avec une continuation effectuant maintenant l'énumération de l'arbre t_2 avec la continuation k_0. Ensuite, `loop_f` sera appelée une troisième fois sur la liste vide, avec une continuation effectuant maintenant l'énumération de l'arbre t_1, cette énumération ayant elle-même une continuation énumérant t_2. Arrivée sur la liste vide, `loop_f` exécute sa continuation, ce qui va effectivement lancer l'énumération de t_1, avec entre chaque étape une énumération de t_2 ; c'est le comportement escompté.

Remarquons ici que nous aurions pu écrire la seconde ligne de `loop_f` sous la forme

```
  | t :: f → loop_t (fun () → loop_f k f) t
```

signifiant à l'inverse un parcours de l'arbre `t` avec une continuation énumérant les coloriages de la forêt `f`. Le résultat eut été différent mais correct. Encore une fois, c'est pour obtenir la même sortie que les implantations de Knuth que nous préférons la première écriture — qui possède d'ailleurs l'intérêt supplémentaire d'être récursive terminale.

La tâche de `loop_t` est légèrement plus compliquée, car elle doit énumérer les coloriages une fois dans un sens et une fois dans l'autre. Pour déterminer le sens de cette énumération, `loop_t` teste la couleur de la racine de l'arbre passé en argument, c'est-à-dire `bits.(i)`. Si cette couleur est blanche, alors on sait que tout l'arbre est blanc. Dès lors, on appelle la continuation, puis on colorie en noir la racine, et enfin on énumère les coloriages des fils à l'aide de `loop_f`. Inversement, si la racine est noire, alors on commence par appeler `loop_f` sur les fils (ce qui a pour effet d'en énumérer les coloriages jusqu'à celui ne contenant que des nœuds blancs), puis on colorie en blanc la racine, et enfin on appelle la continuation.

Ainsi sur l'exemple (1) nous avons vu que `loop_t` est appelée sur l'arbre t_1 (à 2 nœuds) avec une continuation `k` énumérant les coloriages du second arbre t_2 (à 3 nœuds). La racine de t_1 étant blanche initialement, on se trouve dans le premier cas de figure et on commence donc par appeler la continuation et ceci a pour effet d'énumérer une fois tous les coloriages de t_2 ; c'est la première ligne de la figure 1. On colorie ensuite la racine de t_1 en noir ; c'est le premier coloriage de la seconde ligne (tout à droite). Enfin on appelle `loop_f` sur les fils de t_1, à savoir un unique nœud (le nœud 0, blanc pour l'instant), la continuation étant toujours `k` énumérant les coloriages de t_2. `loop_f` va alors s'appeler elle-même sur la liste vide, avec une nouvelle continuation énumérant les coloriages de l'arbre se réduisant au nœud 0 avec une continuation `k`. Le coloriage de cet arbre commence par un appel à la continuation `k`, ce qui donne la deuxième ligne de la figure 1, puis le nœud 0 est colorié en noir, ce qui donne le premier coloriage de la troisième ligne, et enfin `loop_f` est appelée sur les fils du nœud 0, à savoir la liste vide. Ceci a pour effet d'appeler `k`, ce qui donne la dernière ligne de la figure 1.

Pour appliquer le programme à une forêt donnée `f`, il suffit d'appeler `loop_f` avec une continuation effectuant l'affichage du contenu du tableau `bits` :

```
loop_f (fun () → (* affichage du coloriage *)) f
```

Ce code de 9 lignes seulement a le mérite de la concision. Nous verrons section 4 que ses performances sont de plus très satisfaisantes. Enfin, il est important de noter que ce code allie de façon efficace des traits de programmation fonctionnelle — des fonctions sont passées en argument et construites dynamiquement — et de programmation impérative — on modifie en place un tableau alloué statiquement.

Complexité. Pour déterminer la complexité du code C1, commençons par introduire les deux grandeurs dont peut dépendre cette complexité, à savoir la taille d'une forêt et le nombre de ses coloriages. Par la suite, nous utilisons la syntaxe des listes de Caml pour les forêts et la notation `Node` f pour un arbre dont les fils constituent une forêt f (l'indice du nœud n'est pas utile pour la détermination de la complexité).

La taille d'une forêt f (resp. d'un arbre t) est notée $n(f)$ (resp. $n(t)$) ; c'est le nombre de nœuds, défini par récurrence sur la structure des arbres et des forêts de la manière suivante :

$$\begin{array}{rcl} n(\texttt{[]}) & = & 0 \\ n(t::f) & = & n(t)+n(f) \\ n(\texttt{Node}\ f) & = & 1+n(f) \end{array}$$

Le nombre de coloriages possibles d'une forêt f (resp. d'un arbre t) est noté $N(f)$ (resp. $N(t)$), et se calcule également par récurrence sur la structure des arbres et des forêts :

$$\begin{array}{rcl} N(\texttt{[]}) & = & 1 \\ N(t::f) & = & N(t)\times N(f) \\ N(\texttt{Node}\ f) & = & 1+N(f) \end{array}$$

Lorsqu'il n'y a pas ambiguïté, nous notons directement n et N ces deux grandeurs. Pour la forêt (1), on a $n = 5$ et $N = 15$.

Avant d'en venir au calcul-même de la complexité, il faut faire quelques hypothèses quant aux coûts des diverses opérations effectuées. Au delà des appels de fonctions, dont nous négligeons le coût [2], il y a deux opérations à prendre en compte : la modification du tableau `bits`, d'une part, et la construction de la clôture, d'autre part. Cette dernière a comme coût principal celui d'une allocation, donc d'un appel au ramasse-miettes de Caml. Bien qu'il dépend de l'implantation de ce dernier, il est raisonnable de considérer que son coût *amorti* est constant [3]. La modification du tableau `bits` a également un

2. Cette hypothèse simplifie légèrement les calculs par la suite, mais sans en affecter les résultats : considérer le coût des appels de fonctions ne change en effet les complexités obtenues que par un facteur constant.

3. Par coût amorti, nous entendons la moyenne du coût sur l'ensemble de l'exécution.

coût constant. Ces deux coûts n'étant pas disproportionnés, nous considérons sans perte de généralité qu'ils sont tous deux unitaires.

Notons alors $F(k,f)$ le coût d'un appel à la fonction `loop_f` sur une forêt f avec une continuation dont le coût d'exécution est k. De même, notons $T(k,t)$ le coût d'un appel à `loop_t` sur un arbre t. D'après le code C1, les équations de récurrence régissant ces valeurs sont les suivantes :

$$\begin{aligned} F(k,\texttt{[]}) &= k && (4)\\ F(k,t::f) &= 1+F(T(k,t),f) && (5)\\ T(k,\texttt{Node}\ f) &= 1+k+F(k,f) && (6) \end{aligned}$$

Dans l'équation (5), le coût unitaire est celui de la construction de la clôture `(fun () → loop_t k f)`, qui est par hypothèse une continuation de coût $T(k,t)$, d'où le second terme $F(T(k,t),f)$. Dans l'équation (6), le coût unitaire est celui de la modification du tableau `bits`. Montrons alors que l'on a les majorations suivantes :

$$\begin{aligned} F(k,f) &\leq N(f)\times(k+n(f))\\ T(k,t) &\leq N(t)\times(k+n(t))-1 \end{aligned}$$

La preuve se fait par récurrence sur la taille de la forêt ou de l'arbre, en utilisant les équations (4), (5) et (6). Pour la forêt vide `[]`, on a

$$F(k,\texttt{[]}) = k = N(\texttt{[]})\times(k+n(\texttt{[]})) \qquad \text{car } N(\texttt{[]})=1 \text{ et } n(\texttt{[]})=0$$

Pour la forêt $t::f$, on a

$$\begin{aligned} F(k,t::f) &= 1+F(T(k,t),f) && \text{par (5)}\\ &\leq 1+N(f)\times(T(k,t)+n(f)) && \text{par h. r. sur } f\\ &\leq 1+N(f)\times(N(t)\times(k+n(t))-1+n(f)) && \text{par h. r. sur } t\\ &\leq N(f)\times N(t)\times(k+n(t)+n(f))+1-N(f) && \text{car } N(t)\geq 1\\ &= N(t::f)\times(k+n(t::f))+1-N(f) &&\\ &\leq N(t::f)\times(k+n(t::f)) && \text{car } N(f)\geq 1 \end{aligned}$$

Enfin, pour l'arbre `Node` f, on a

$$\begin{aligned} T(k,\texttt{Node}\ f) &= 1+k+F(k,f) && \text{par (6)}\\ &\leq 1+k+N(f)\times(k+n(f)) && \text{par h. r. sur } f\\ &= (1+N(f))\times(k+1+n(f))-n(f)-N(f) &&\\ &= N(\texttt{Node}\ f)\times(k+n(\texttt{Node}\ f))-n(f)-N(f) &&\\ &\leq N(\texttt{Node}\ f)\times(k+n(\texttt{Node}\ f))-1 && \text{car } n(f)\geq 0, N(f)\geq 1 \end{aligned}$$

En particulier, lorsque l'on applique `loop_f` à une forêt f avec une continuation initiale dont on ignore le coût, la majoration du coût total se résume à $N(f)\times n(f)$, et l'on peut donc affirmer que

le code C1 a une complexité $O(nN)$

On peut montrer de manière analogue que le nombre de clôtures construites est majoré par $N(f)-1$ et donc que la complexité en espace est $O(N)$.

3. Encore plus d'ordre supérieur

On peut faire encore mieux, car notre programme effectue pour l'instant de nombreuses fois les mêmes calculs. En effet, chaque arbre (resp. chaque forêt) est traversé de nombreuses fois, et à chaque

```
let rec loop_f k = function
  | [] → k
  | t :: f → loop_f (loop_t k t) f
and loop_t k (Node (i,f)) =
  let lf = loop_f k f in
  fun () →
    if bits.(i) = 0 then begin
      k (); bits.(i) ← 1; lf ()
    end else begin
      lf (); bits.(i) ← 0; k ()
    end
```

FIG. 3: *Un second codage de Koda-Ruskey (C2).*

fois les mêmes continuations sont reconstruites. Ainsi, sur l'exemple de la forêt (1), `loop_t` va être appelée trois fois avec la continuation initiale sur l'arbre t_2, ce qui correspond aux trois lignes de la figure 1, et à chaque fois la même continuation va être fabriquée par l'appel de `loop_f` sur la forêt constituée des nœuds 2 et 3. De manière générale, dès qu'il y a plusieurs arbres dans une forêt, les mêmes continuations sont construites plusieurs fois pour chacun des arbres de la forêt à l'exception du premier.

On peut en fait factoriser ces constructions identiques en modifiant très légèrement notre premier code. L'idée est d'avoir maintenant des fonctions `loop_t` et `loop_f` qui ne font plus l'énumération directement mais retournent des fonctions, de type `unit → unit`, qui le feront lorsqu'elles seront appliquées.

Le code est donné figure 3, et nous l'appelons C2. Il diffère en trois points du précédent. Premièrement, lorsque la forêt est vide on retourne `k` directement au lieu de l'appliquer, puisqu'une fonction est maintenant attendue. Deuxièmement, dans le cas d'une forêt non vide il suffit d'évaluer `(loop_t k t)` pour obtenir une fonction, par spécification de `loop_t` ; il n'est plus besoin de placer ce code sous une abstraction comme auparavant. Troisièmement, et c'est là le plus important, l'appel à `loop_f` dans `loop_t` est effectué une unique fois (dans la variable `lf`, qui est une fonction) puis la fonction à retourner est construite. Ainsi l'évaluation de `loop_f` ne sera faite qu'une seule fois dès que deux arguments seront passés à `loop_t`. Or, c'est justement une telle application partielle de `loop_t` qui est faite dans `loop_f`.

Pour appliquer le programme à une forêt donnée `f`, il faut maintenant appliquer `loop_f` à une continuation effectuant l'affichage et à `f`, ce qui nous donne une fonction, puis appliquer cette dernière à `()`, ce qui s'écrit directement

```
loop_f (fun () → (* affichage du coloriage *)) f ()
```

Ainsi, sur l'exemple de la forêt (1), `loop_f` est appliquée à la continuation initiale k_0 et à la forêt $[t_2; t_1]$. Ceci a pour effet d'évaluer `loop_f` (`loop_t` k_0 t_2) $[t_1]$. L'application partielle (`loop_t` k_0 t_2) va alors effectuer un appel à `loop_f` avec la continuation k_0 et la forêt constituée des nœuds 2 et 3, puis retourner une fonction k_1. Cet appel à `loop_f` est maintenant unique, là où il était fait trois fois dans le code initial, même si la fonction retournée k_1 sera effectivement utilisée trois fois.

Ce nouveau code, à peine plus long que le précédent (11 lignes contre 9), exploite encore plus l'ordre supérieur : on a maintenant des fonctions retournées comme résultats et des applications partielles de fonctions. Le principe reste le même, à savoir que la continuation énumère les coloriages d'une certaine forêt f_0 accolée à la forêt argument de `loop_f` (resp. l'arbre argument de `loop_t`). On a seulement séparé le calcul en deux phases successives : une première qui calcule la fonction d'énumération par récurrence sur la structure des forêts et des arbres, et la seconde qui l'applique effectivement.

Remarquons enfin que la fonction `loop_f` est devenue un simple "`fold`" de la fonction `loop_t` sur la liste que constitue la forêt passée en argument. On pouvait en effet écrire de manière équivalente

```
let rec loop_f k f = List.fold_left loop_t k f
and loop_t ...
```

Ceci n'était pas possible dans le cas de C1 à cause de l'application d'une fonction avec effets de bords (la continuation `k`) une fois arrivé sur la liste vide.

Complexité. Les fonctions `loop_f` et `loop_t` ont maintenant trois arguments. Si l'application au premier argument n'entraîne pas de calcul, les applications à un deuxième puis à un troisième ont des coûts propres, qu'il s'agit d'évaluer séparément.

Le coût de l'application aux deux premiers arguments est aisé à déterminer. En effet, chaque nœud entraîne un coût unitaire dû à la construction de la clôture dans `loop_t`, et le reste des calculs n'est constitué que d'appels récursifs traversant tous les nœuds une fois et une seule. En conséquence, le coût total est exactement n, le nombre de nœuds. D'autre part, vu que seule cette première phase alloue de la mémoire (pour les clôtures), on en déduit d'ores et déjà que la complexité totale en espace de C2 est $O(n)$.

La détermination du coût de l'application au troisième argument utilise une technique similaire à celle introduite dans la section précédente. Notons maintenant $F(k,f)$ le coût de l'exécution de la fonction retournée par `loop_f` et $T(k,t)$ le coût de l'exécution de celle retournée par `loop_t`, k désignant toujours le coût de la continuation. D'après le code C2, les équations de récurrence régissant ces nouvelles valeurs sont les suivantes :

$$\begin{aligned} F(k,\texttt{[]}) &= k && (7) \\ F(k,t::f) &= F(T(k,t),f) && (8) \\ T(k,\texttt{Node}\ f) &= 1+k+F(k,f) && (9) \end{aligned}$$

Ces équations ne diffèrent des précédentes que par la seconde ; cette différence exprime que l'on ne retraverse plus les forêts à chaque fois que l'on en cherche les coloriages, car on dispose maintenant d'une fonction pour les traverser, calculée une seule fois. Montrons alors que l'on a les majorations suivantes :

$$\begin{aligned} F(k,f) &\leq N(f)\times(k+1)-1 \\ T(k,t) &\leq N(t)\times(k+1)-1 \end{aligned}$$

La preuve est menée comme dans le cas de C1. Pour la forêt vide `[]`, on a

$$F(k,\texttt{[]}) = k = N(\texttt{[]})\times(k+1)-1 \qquad \text{car } N(\texttt{[]})=1$$

Pour la forêt $t::f$, on a

$$\begin{aligned} F(k,t::f) &= F(T(k,t),f) && \text{par (8)} \\ &\leq N(f)\times(T(k,t)+1)-1 && \text{par h. r. sur } f \\ &\leq N(f)\times N(t)\times(k+1)-1 && \text{par h. r. sur } t \\ &= N(t::f)\times(k+1)-1 \end{aligned}$$

Enfin, pour l'arbre `Node` f, on a

$$\begin{aligned} T(k,\texttt{Node}\ f) &= 1+k+F(k,f) && \text{par (9)} \\ &\leq 1+k+N(f)\times(k+1)-1 && \text{par h. r. sur } f \\ &= (1+N(f))\times(k+1)-1 \\ &= N(\texttt{Node}\ f)\times(k+1)-1 \end{aligned}$$

En particulier, lorsque l'on applique `loop_f` à une continuation initiale dont on ignore le coût et à une forêt f on a un premier coût $n(f)$ puis lorsque l'on applique enfin à `()`, on a un deuxième coût $N(f) - 1$. Vu que l'on a $n(f) \leq N(f)$ on peut donc affirmer que

le code C2 a une complexité $O(N)$

Cette complexité est bien entendu optimale.

4. Performances

Nous avons comparé les performances de nos deux implantations C1 et C2 en Objective Caml à celles en C de Knuth [4]. La première, appelons-la K1, est constituée de n co-routines et est codée uniquement à l'aide de `goto`. La seconde, appelons-la K2, utilise une structure doublement chaînée introduite dans l'article de Koda et Ruskey [7]. Ce code a une complexité optimale $O(N)$ et est donné en annexe de cet article à titre de comparaison.

Les quatre programmes ont été exécutés sur des forêts engendrées aléatoirement, de diverses tailles et diverses hauteurs. La figure 4 donne les résultats pour quatre de ces forêts de taille variant entre 34 et 44 nœuds[4]. Les affichages des programmes ont été supprimés, de sorte qu'après les phases d'initialisation, très courtes, seul le temps d'exécution de l'algorithme proprement dit est pris en compte. Les tests ont été menés sur un Pentium III 450 MHz sous Linux. Les temps d'exécution sont donnés en secondes, arrondis à trois chiffres significatifs.

	Forêt 1	Forêt 2	Forêt 3	Forêt 4
n (nb nœuds)	34	38	42	44
N (nb coloriages)	18 661 061	55 377 985	89 598 110	1 625 431 136
K1	17.70 s	51.80 s	95.5 s	1920 s
K2	2.39 s	7.13 s	10.6 s	161 s
C1	5.22 s	16.10 s	27.8 s	461 s
C2	3.92 s	11.80 s	19.6 s	378 s

FIG. 4: *Temps d'exécution sur quatre forêts aléatoires.*

Comme on peut le voir, le code le plus rapide est K2, mais notre seconde implantation C2 reste dans un rapport très satisfaisant (jamais plus de 2,5). On peut également constater que notre première implantation C1, bien que plus lente que C2, offre des performances tout à fait raisonnables, notamment par rapport au code K1, uniquement écrit à l'aide de branchements `goto` à des fins d'efficacité.

Il faut cependant relativiser ces comparaisons, pour au moins deux raisons : d'une part, la quantité de calcul entre deux coloriages est tellement faible qu'une comparaison numérique est difficile, surtout pour des codes écrits dans des langages différents ; d'autre part, l'implantation K2 parvient à n'effectuer qu'une quantité bornée de calcul entre deux coloriages, ce qui n'est pas le cas de nos deux codes.

5. Conclusion

Nous avons présenté une implantation de l'algorithme de Koda-Ruskey dans un langage de programmation fonctionnel, à savoir Objective Caml, en y exploitant l'ordre supérieur, c'est-à-dire le caractère de première classe des fonctions.

4. Nos tests ont porté sur un plus large échantillon de forêts, avec des résultats identiques.

Le premier avantage de cette implantation est son élégance : notre code se compose de 9 lignes seulement, là où les codes proposés jusqu'ici font au minimum 50 lignes et impliquent des constructions impératives complexes (branchements `goto` ou manipulations de pointeurs). Nous avons montré également comment une première version de notre code pouvait être raffinée simplement en un code de complexité optimale.

Au delà de sa concision, notre implantation de l'algorithme de Koda-Ruskey possède un autre avantage : l'espoir d'en prouver formellement la correction. Une preuve formelle dans l'assistant de preuve Coq [1] est en cours de développement. Elle exploite la tactique de preuve de programmes impératifs développée par l'auteur dans sa thèse [3], qui autorise les fonctions d'ordre supérieur. La preuve est difficile mais semble réalisable. Il serait intéressant de la comparer ensuite à des preuves formelles des codes K1 et K2. (Tels qu'ils sont écrits, K1 et K2 semblent hors de portée des outils de certification actuels, mais ils pourraient probablement être reformulés d'une manière adaptée à la preuve formelle.)

Notons enfin qu'il existe une généralisation de l'algorithme de Koda-Ruskey à des graphes orientés totalement acycliques dûe à G. Li et F. Ruskey. Ce travail n'a pas encore été publié mais Knuth en propose déjà une implantation de complexité optimale sur son site Internet. Nous envisageons bien entendu de voir comment notre code pourra être étendu à l'algorithme de Li-Ruskey.

Remerciements. L'auteur remercie Benjamin Monate et les rapporteurs anonymes pour leurs remarques et leurs suggestions.

Références

[1] The Coq Proof Assistant. `http://coq.inria.fr/`.

[2] The Objective Caml language. `http://caml.inria.fr/`.

[3] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. Thèse de doctorat, Université Paris-Sud, Juillet 1999.

[4] Donald E. Knuth. Implantation de l'algorithme de Koda-Ruskey, 2001. Disponible à l'adresse `http://www-cs-faculty.stanford.edu/~knuth/programs.html`.

[5] Donald E. Knuth. *The Art of Computer Programming*, volume Volume 4, Pre-Fascicle 2a: A Draft of Section 7.2.1.1: Generating all n-tuples. Addison-Wesley, 2001. Livre à paraître. Fascicule disponible à l'adresse `http://www-cs-staff.Stanford.EDU/~knuth/news.html`.

[6] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation*. Addison-Wesley, 1993. Le logiciel CWEB est disponible à l'adresse `http://www-cs-staff.Stanford.EDU/~knuth/cweb.html`.

[7] Yasunori Koda and Frank Ruskey. A Gray Code for the Ideals of a Forest Poset. *Journal of Algorithms*, (15):324–340, 1993.

Annexe : Une implantation en C

À titre de comparaison, nous donnons ici le code C de la seconde implantation de Knuth [4], obtenu à partir du code CWEB. La forêt y est représentée par un tableau `scope` où `scope`[i] représente l'indice du plus petit descendant du nœud i, incluant i lui-même.

```
#define forestsize 100
```

```
int scope[forestsize];
```

Nous de donnons ci-dessous que le code de l'algorithme lui-même ; n'apparaissent donc pas le remplissage du tableau `scope` à partir de la forêt donnée sur la ligne de commande, ni le code affichant le coloriage courant. En ce qui concerne le détail du fonctionnement de cet algorithme, nous renvoyons le lecteur au code CWEB de Knuth [4], très bien commenté, ou encore à l'article original de Koda et Ruskey [7]. Le code ci-dessous permet néanmoins de saisir la taille (54 lignes) et surtout la complexité de cette implantation.

```
typedef struct lnode_struct {
  char bit;
  struct lnode_struct *left, *right;
  struct lnode_struct *lchild;
  struct lnode_struct *focus;
} lnode;
lnode lnode_table[forestsize+1];
#define head (lnode_table+nn)
boolean been_there_and_done_that;

{
  register lnode *p,*q,*r;

  for(k= 0; k<=nn; k++) {
    lnode_table[k].focus = lnode_table+k;
    if (scope[k] < k) {
      for(j = k-1; scope[j] > scope[k]; j = scope[j]-1) {
        lnode_table[j].left = lnode_table+scope[j]-1;
        lnode_table[scope[j]-1].right = lnode_table+j;
      }
      lnode_table[k].lchild = lnode_table+j;
    }
  }
  head→left = head-1, (head-1)→right = head;
  head→right = head→lchild, head→lchild→left = head;

  while (1) {
    /* l'affichage du coloriage courant se fait ici */
    q = head→left;
    p = q→focus;
    q→focus = q;
    if (p != head) {
      if (p→bit == 0) {
        p→bit = 1;
        if (p→lchild) {
          q = p→right;
          q→left = p-1, (p-1)→right = q;
          p→right = p→lchild, p→lchild→left = p;
        }
      } else {
        p→bit = 0;
        if( p→lchild) {
          q = (p-1)→right;
```

```
          p→right = q, q→left = p;
        }
      }
    } else if (been_there_and_done_that) break;
    else {
      been_there_and_done_that = true; continue;
    }
    p→focus = p→left→focus;
    p→left→focus = p→left;
  }
}
```