Cours de Programmation
MIM1, 2000-2001, Devoir 1

Laure DANTHONY, Sylvain FOUR
pour le 13 octobre 2000

1 Commentaires généraux

Les deux algorithmes développés dans ce D.M. ont de nombreux points communs : ils sont tous les deux exponentiels, et tous les deux récursifs. De plus, ils nécessitent l'emploi des outils graphiques de Caml.

Les codes ne sont pas inclus dans le compte-rendu. Ils sont sur une feuille annexe. Par contre, nous avons choisi d'inclure les résultats graphiques.

2 Le flocon de Von Koch

2.1 Le problème

Le but de la section est de dessiner une courbe fractale sur l'écran graphique de Caml. Soit le motif de base suivant :

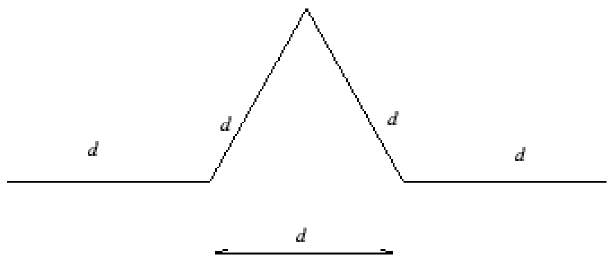

FIG. 1 – *Le motif de base*

Il s'agit de redessiner ce motif sur chaque sous-segment du motif précédent (évidemment, en divisant les longueurs par trois !), et ainsi de suite...

2.2 Une première version

2.2.1 La procédure `avance distance angle`

Ecrivons tout d'abord une procédure `avance distance angle` qui trace à partir du point courant un segment de longueur `distance` dans la direction définie par `angle`. Le typage de cette procédure est le suivant :

```
avance : float -> float -> unit = <fun>.
```

On considérera en effet les variables `distance` et `angle` comme des flottants pour une question de précision des calculs, précision qui s'avérera non négligeable comme on le verra dans la suite.

Dans un premier temps, la manière la plus naturelle de programmer cette procédure est de partir du couple d'entiers donné par la fonction `current_point`. On transforme ce couple d'entiers en couple de flottants afin de créer un couple de coordonnées flottantes tels que le segment formé par ces deux points soit de longueur `distance` et de direction définie par `angle`. Enfin, on trace un segment du point courant vers le point dont on vient de calculer les coordonnées. Malheureusement, on est contraint (pour une question de typage de la procédure `lineto`) d'utiliser la procédure `int_of_float` qui va tronquer chaque flottant en un entier. On remarquera que la fonction `int_of_float` aura été appelée sur `x_cible_float` et non pas uniquement sur `cos angle` afin de conserver un seuil minimum de précision.

2.2.2 La procédure `motif generation distance angle`

Cette procédure trace les quatre motifs de la génération $n+1$ en s'appelant récursivement sur des motifs de génération n. Elle a ainsi pour typage :

```
motif : int -> float -> float -> unit = <fun>.
```

La génération 0 correspond à un segment de longueur `distance` et de direction définie par `angle` d'où l'appel à la procédure `avance distance angle`. Lorsque `motif` est appelé à la génération n, il lui suffit de créer un motif de génération $n-1$ sur chaque segment de `motif 1 distance angle`. C'est pourquoi on appelle la procédure `motif (n-1) (distance /. 3.)` respectivement sur `angle` (pour un sous-motif dans la direction précédente), sur `angle +. (pi /. 3.)` et `angle -. (pi /. 3.)` (pour deux sous-motifs formant le "pic") puis encore une fois sur `angle` (pour la fin du motif dans l'ancienne direction). Pour éviter que la procédure calcule plusieurs fois `distance /. 3.` et `pi /. 3.`, on a choisi de créer, dans le premier cas, une variable locale `nouvelle_distance` et, dans le deuxième cas, une variable globale `pi_sur_trois`.

2.2.3 Résultats de la première version

Après des appels à `motif n 300. 0.` avec $n \in \{0,1,2,3,4,5\}$, on obtient la figure page suivante.

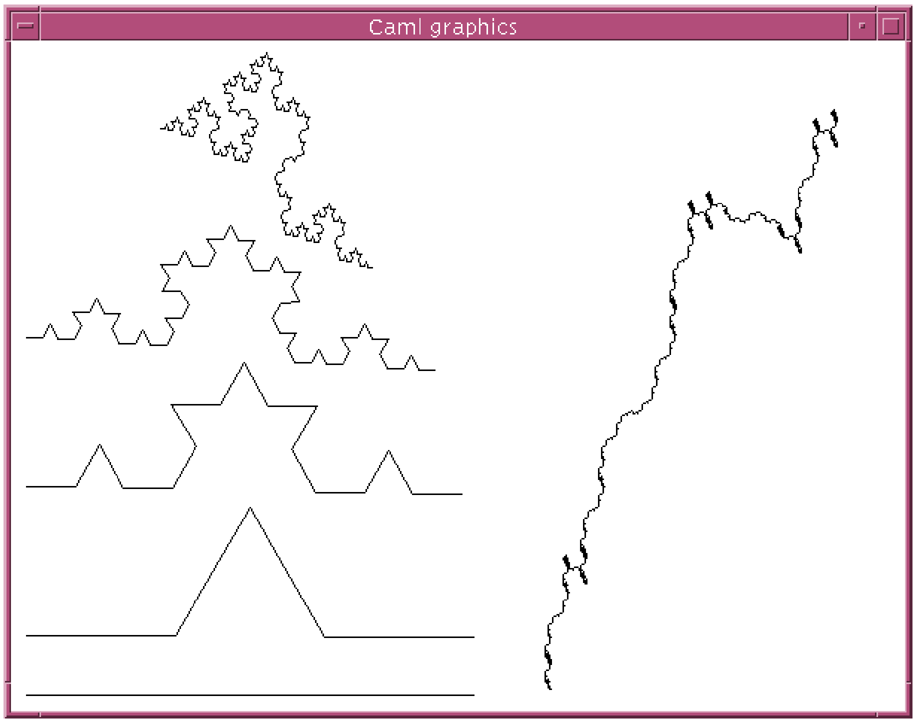

FIG. 2 – *La toute première version*

Les appels pour $n \in \{0, 1, 2\}$ sont satisfaisants : on n'observe aucune distorsion. Par contre, l'appel de `motif 3 300. 0.` montre que le motif n'est pas horizontal. Cette constation devient très claire sur les appels suivants où on a des difficultés à reconnaître le motif. On peut expliquer les erreurs de la première version par un manque de précision dans le calcul des coordonnées du point cible de la procédure `avance distance angle`.

L'écriture d'une deuxième version semble donc nécessaire : elle aura pour but de pallier à ces problèmes de précision.

2.3 Une deuxième version

2.3.1 Les quelques changements...

La première solution à laquelle on pense consiste à incriminer la procédure de troncature `int_of_float` qui s'avère créer des imprécisions énormes au fil des appels à `avance distance angle`. En effet, les erreurs se répercutent dans la procédure globale `current_point` et s'y ajoutent. Il suffit alors de créer une procédure `arrondi f_float` ayant pour typage :

```
arrondi : float -> int = <fun>.
```

`arrondi f_float` prend en argument un flottant qu'elle arrondit au mieux c'est-à-dire :

$$\texttt{arrondi f\_float} = \begin{cases} tronque(\texttt{f\_float}) - 1 & \text{si } \texttt{f\_float} - tronque(\texttt{f\_float}) < -0.5 \\ tronque(\texttt{f\_float}) + 1 & \text{si } \texttt{f\_float} - tronque(\texttt{f\_float}) > 0.5 \\ tronque(\texttt{f\_float}) & \text{sinon} \end{cases}$$

Ceci constitue l'unique changement apporté à la deuxième version.

2.3.2 Résultats de la deuxième version

Après des appels à `motif n 300. 0.` avec $n \in \{0, 1, 2, 3, 4, 5, 6\}$, on obtient :

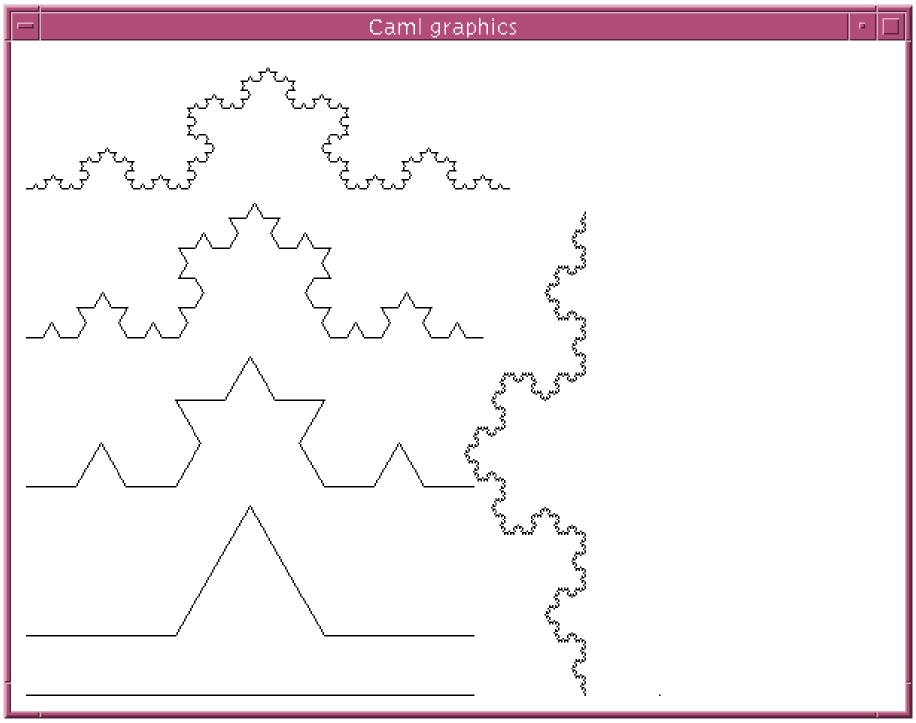

FIG. 3 – *Von Koch, deuxième version*

Cette fois-ci, l'ensemble des résultats paraît satisfaisant puisqu'il n'y a pas de points manquants, ni de distorsions. Cependant, une remarque s'impose... Mais, où est passé le résultat de l'appel `motif 6 300. 0.` ? En fait, l'appel de la procédure avec de tels arguments ne conduit à aucune réponse visible... Pourquoi ?

Ceci s'explique lorsqu'on "déplie" la récurrence. Si on la déplie jusqu'à la génération 0, on tombe sur l'appel de `motif` avec pour paramètre `distance`, le flottant $\frac{300.}{3^6}$. Or, ce dernier est strictement inférieur à 0.5. Ainsi s'explique le fait qu'à chaque appel de la procédure `avance distance angle`, aucun segment n'est tracé puisque le point courant a les mêmes coordonnées que le point cible. Le typage de la procédure `lineto` nous avait en effet contraints à garder des coordonnées entières pour les points, d'où un seuil de précision trop faible.

2.4 Une troisième et ultime version

2.4.1 Les derniers changements...

Pour franchir ce seuil de précision, la seule solution consiste à garder en mémoire un point courant de type `float`. Lors d'un appel initial à `avance distance angle` avec `distance` < 0.5, aucun segment ne sera certes tracé par `lineto` mais le point courant flottant aura, lui, changé de coordonnées et pourra, grâce à des appels ultérieurs, répercuter ce déplacement lorsque la distance entre le point initial et le point cible sera suffisante.

Voilà la raison pour laquelle on a décidé de créer une variable globale `point` de type `point_float`. Il s'agit en effet d'une variable laissant deux champs disponibles, son abscisse et son ordonnée, toutes deux flottantes. Nous aurions pu modéliser ce point courant par deux simples variables globales, mais nous avons préféré conserver la modélisation implémentée en créant des procédures similaires : `point_courant_float ()` d'une part, `moveto_float x_float y_float` et `lineto_float x_float y_float` d'autre part. Elles ont le même rôle que leurs homologues OCaml excepté le typage :

```
point_courant_float : unit -> float * float = <fun>,
   moveto_float : float -> float -> unit = <fun>,
   lineto_float : float -> float -> unit = <fun>.
```

Il ne restait plus qu'à implémenter la procédure `flocon n distance` qui place un motif de génération n sur chaque côté d'un triangle équilatéral. Son typage est clair :

```
motif : int -> float -> unit = <fun>.
```

Remarque : Hormis la construction élémentaire des trois motifs, il a aussi fallu faire un choix sur la position initiale du point courant. On a décidé de le choisir en bas à gauche afin de ne pas à avoir à tenir compte de la taille de la fenêtre graphique, et afin de prendre le moins de place possible pour la construction de flocons de grande taille. L'ordonnée du point courant est initialisé à $\frac{\texttt{distance}}{3}$ afin que le flocon ne "dépasse" pas de l'écran graphique.

2.4.2 Les derniers résultats...

A la suite d'appels à `motif n 300. 0.` avec $n \in \{0, 1, 2, 3, 4, 5, 6, 7\}$, on obtient la figure page suivante.

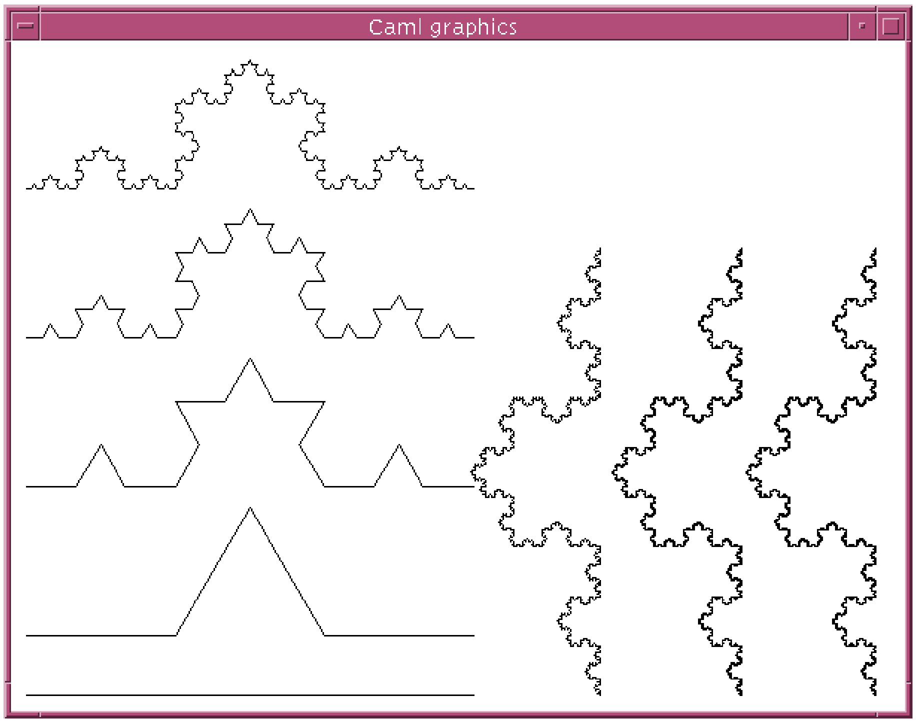

Fig. 4 – *Von Koch, troisième version*

Cette fois-ci, tout semble satisfaisant : il ne manque aucun motif et on n'observe ni point manquant ni distorsion. Il ne reste plus qu'à faire l'appel à `flocon 7 300.` et on obtient la figure page suivante.

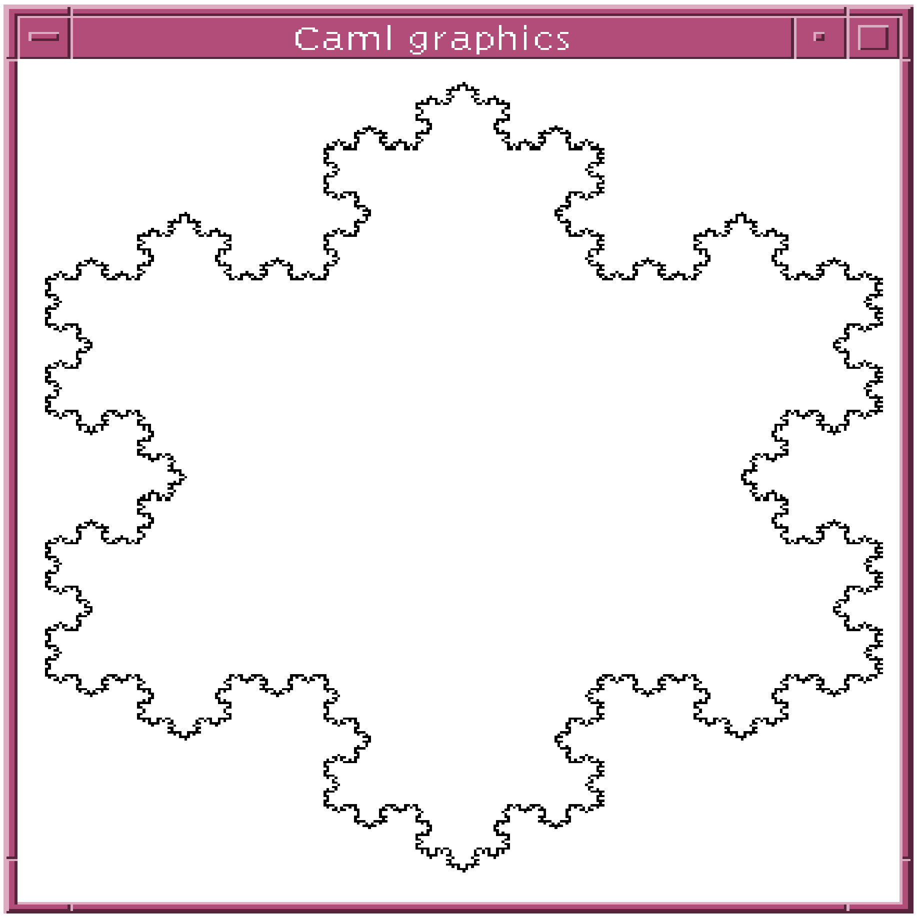

Fig. 5 – *C'est très joli !*

Ici encore, rien à signaler.

2.5 Perspectives

On pourrait maintenant modifier le motif afin d'observer de nouvelles fractales. Pour éviter de faire un code pour chaque motif, on pourrait créer une procédure qui prendrait en argument la génération, et une série d'instructions créant le motif initial.

3 Les tours de Hanoï

3.1 Le problème : vocabulaire, notations, règle du jeu

Le joueur dispose de trois *piquets* et de n *rondelles* de tailles différentes (disons de 1 à n).

Au début du jeu, toutes les rondelles sont posées par ordre de taille sur le piquet de gauche (piquet A). Le but du jeu est de les déplacer **sans jamais poser une rondelle sur une rondelle plus petite** vers le piquet de droite (piquet C) en se servant du piquet du milieu (piquet B).

Dès lors *"résoudre hanoï n"*, c'est donner les *mouvements* successifs des rondelles d'un piquet *origine* à un piquet *destination* qui ont pour résultat un déplacement "légal" des rondelles empilées sur le piquet A vers le piquet C.

3.2 Analyse du problème et premier (petit) plantage

Le programmeur qui ne sait pas (de tête) résoudre le problème `hanoï 5` se dit qu'il va laisser faire le travail à Caml, sachant que, vu qu'il n'est pas totalement bête, il sait résoudre `hanoï 1`.

La résolution du problème (par récursivité) est basée sur la remarque suivante : si on sait résoudre `hanoi n-1`, il suffit pour résoudre `hanoi n` de déplacer les $n-1$ rondelles les plus petites de A vers B (en se servant du piquet C vide au départ) puis de déplacer la dernière rondelle de A vers C et enfin de déplacer les $n-1$ disques de B vers C (en se servant du piquet A, bien sûr !).

Le (tout) premier programme ne fonctionnait pas à cause d'une mauvaise analyse : le programmeur un peu rapide pensait qu'il suffisait de déplacer la première rondelle de A vers B puis de faire `hanoi (n-1)` de A vers le piquet C, puis enfin de ramener la dernière rondelle de B vers C. Le lecteur attentif aura compris qu'il y a un (petit) problème, vu que l'on se sert de B comme piquet intermédiaire alors qu'il contient la plus petite rondelle.

3.3 Premiers codes, résultats, complexité

3.3.1 Ecriture d'une fonction auxilliaire

L'objet de cette section est l'écriture d'une fonction `mouvement` prenant en argument `origine` et `destination` et qui affiche le mouvement d'une rondelle du piquet *origine* vers le piquet *destination*. Elle sera donc du type :

```
string -> string -> unit = <fun>
```

On fait ici un choix : le nom des piquets est de type `string`, par la suite on pourra éventuellement utiliser la fonction `string_of_int` par exemple. De plus, pour vérifier le calcul de complexité qui suivra, on choisit d'inclure un compteur du nombre de mouvements. Il faut donc gérer une variable globale `compt` et donc la passer en paramètre à la fonction `mouvement`. Le typage devient :

```
mouvement : string -> string -> int ref -> unit = <fun>
```

3.3.2 Code de HANOI

Une fois la fonction `mouvement` écrite, l'écriture de la fonction `hanoi` découle immédiatement de l'analyse du problème.

Cette fonction est du type :

```
hanoi : int -> int = <fun>
```

3.3.3 Résultats de la première version

```
# hanoi 2;;
mouvement A ---> B
mouvement A ---> C
mouvement B ---> C
- : int = 3
# hanoi 3;;
mouvement A ---> C
mouvement A ---> B
mouvement C ---> B
mouvement A ---> C
mouvement B ---> A
mouvement B ---> C
mouvement A ---> C
- : int = 7
# hanoi 4;;
mouvement A ---> B
mouvement A ---> C
mouvement B ---> C
mouvement A ---> B
mouvement C ---> A
mouvement C ---> B
mouvement A ---> B
mouvement A ---> C
mouvement B ---> C
mouvement B ---> A
mouvement C ---> A
mouvement B ---> C
mouvement A ---> B
mouvement A ---> C
mouvement B ---> C
- : int = 15
#
```

FIG. 6 – *version élémentaire, somme toute !*

3.3.4 Analyse de la complexité de l'algorithme

Etudions la complexité de cet algorithme récursif : il s'agit de résoudre la récurrence suivante ($C(n)$ désigne le nombre de mouvements qu'effectue cet algorithme dans la résolution de `hanoï n`) :

$$C(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2C(n-1)+1 & \text{sinon} \end{cases}$$

On remarque tout d'abord que la relation de récurrence se réécrit en

$$C(n) + 1 = 2\big(C(n-1)+1\big)$$

d'où la nouvelle récurrence, en posant $U(n) = C(n) + 1$:

$$U(n) = \begin{cases} 2 & \text{si } n = 1 \\ 2U(n-1) & \text{sinon} \end{cases}$$

ce qui se résout facilement et donc finalement $C(n) = 2^n - 1$.

3.4 Pour aller plus loin ...

L'objet de cette section est l'élaboration d'une interface graphique permettant de visualiser les mouvements successifs des rondelles.

3.4.1 Remarques, conventions

On s'aperçoit très rapidement qu'il est nécessaire de stocker à tout moment l'état des piquets, c'est à dire :

- le nombre de rondelles sur chaque piquet,
- le numéro (c'est à dire la taille) de chaque rondelle dans l'ordre d'empilement sur le piquet.

Le choix fait ici est le suivant : on stocke un tableau `etat_piquets` de 3 lignes et $n+1$ colonnes (attention, en Caml les indices vont de 0 à 2 pour les lignes et de 0 à n pour les colonnes !). Chaque appel à la procédure `mouvement` modifiera ce tableau **en place**.

Les conventions utilisées sont les suivantes :

- à la case `etat_piquet.(i).(0)` on stocke le nombre de rondelles sur le piquet i
- à la case `etat_piquet.(i).(j)` on stocke la taille de la $j^{\text{ième}}$ rondelle qui se situe sur le plateau i.

Remarque : Attention, les piquets sont maintenat numérotés de 0 à 2 !

3.4.2 La procédure `mouvement`

La procédure `mouvement` :

```
mouvement : int -> int -> int array array -> int -> unit = <fun>
```

répercute le mouvement de *origine* vers *destination* de la façon suivante :

- le nombre de rondelles sur le plateau *origine* (resp.*destination*) est décrémenté (resp. incrémenté),
- le numéro de la rondelle déplacée de *origine* est imprimé dans la case adéquate du piquet *destination*, c'est à dire au-dessus de la dernière rondelle (accessible facilement vu que l'on connaît justement ce nombre de rondelles avant déplacement)

Cependant, la procédure `mouvement` ne se contente pas de modifier le tableau `etat_piquets`. Elle fait aussi appel à une fonction `dessine_etat` de type `int array array -> int -> unit = <fun> >` qui est chargée de dessiner dans la fenêtre graphique le nouvel état après le mouvement et qui fait elle-même appel à des fonctions graphiques auxilliaires , telles que :

- `dessine_piquets : int-> int -> unit = <fun>` qui dessine un piquet A, B ou C de hauteur paramétrée par le nombre n.

- `dessine_rondelle : int -> int -> int -> int -> unit = <fun>` qui dessine sur un piquet donné une rondelle de taille donnée à la position passée en paramètre.

Remarque : Ce travail sur la procédure mouvement permet (presque) de laisser tel quel le code de `hanoi n`, ce qui est plus pratique pour de simples raisons de lisibilité. Ceci dit, il faut tout de même remplir le tableau `etat_piquets` à la configuration initiale, c'est à dire "mettre" dans les cases correspondantes les n rondelles de taille 1 à n sur le premier piquet. C'est l'objet de la fonction `init_etats` de type :

```
int -> int array array = <fun>
```

3.4.3 Dernières remarques

- Pour le choix de la structure de données à stocker, le fait que les opérations soient *dépiler* et *empiler* et que l'on stocke $\frac{2}{3}n$ zéros en trop à chaque fois peut nous faire penser à utiliser, au lieu de la matrice, trois listes dont la taille varie (une pour chaque support). Ceci dit, dans l'élaboration graphique, les listes sont moins pratiques pour le dessin des rondelles car il faut avoir accès à tous les éléments sans pour autant modifier la liste. La solution consistant en des copies de listes a été abandonnée.

- Dans la procédure mouvement il s'agit *d'agir* en place sur le tableau donné en paramètre. Après vérification, les tableaux passés en paramètre d'une fonction Caml sont modifiés, même si la fonction est de type `unit`.

- Dans la procédure mouvement, on remarque qu'il n'est pas nécessaire d'effacer la rondelle déplacée (c'est à dire de la remplacer par 0 dans le tableau) car l'on sait que l'on n'ira pas lire cette case du tableau (il est toutefois nécessaire de mettre à jour le nombre de disques de chaque piquet).

3.4.4 Résultats de Hanoï version graphique

Voir page suivante.

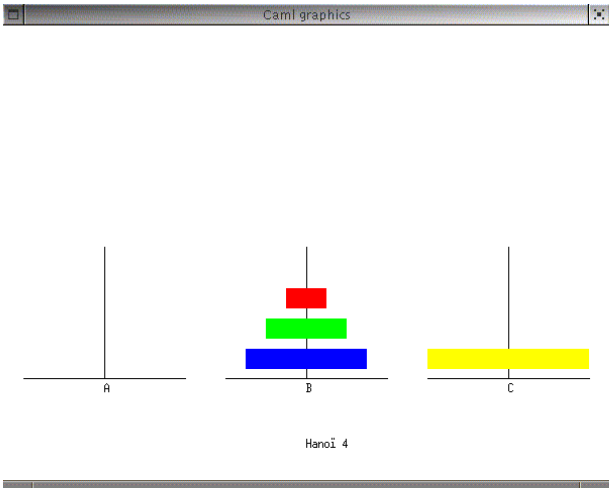

FIG. 7 – *une photo au milieu de l'exécution*

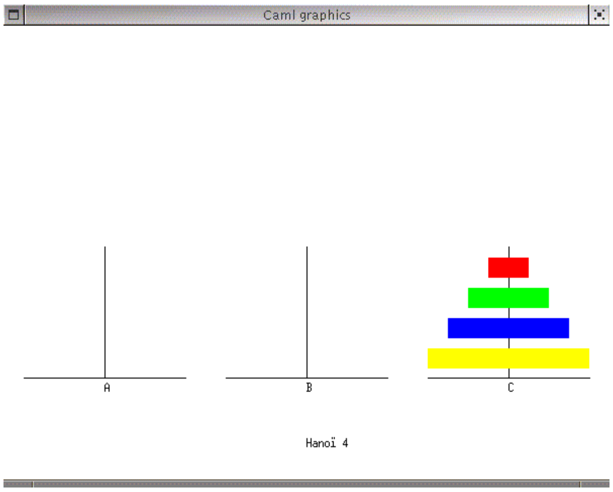

FIG. 8 – *et voilà le problème résolu!*

4 Dernières remarques

Afin de partager les tâches, et de rendre le travail de rédaction plus facile, nous avons choisi de traiter séparément les deux algorithmes, en nommant un responsable par partie. Toutefois, les mises en commun se sont révélées fructueuses, car chacun a pu apporter sa contribution au travail de l'autre. En plus des discussions concernant notamment la gestion de la précision pour `Von Koch` et des mouvements (tableau de listes ou matrice ?) pour `Hanoi`, il a aussi fallu résoudre de petits problèmes, et à deux, c'est plus rapide ! Par exemple, l'un a pu chercher pour l'autre les versions OCaml de certaines commandes Caml Light (comme `make_matrix`), l'autre s'occupant de la gestion des copies d'écran en LaTeX. Ce D.M. a aussi été l'occasion pour l'un de faire ses premiers pas en LaTeX, et pour l'autre de parfaire ses connaissances, notamment en ce qui concerne la gestion des images eps.