Des descriptions fonctionnelles aux implémentations impératives des programmes

Yves Bertot

INRIA Sophia-Antipolis
2004 Route des lucioles, B.P. 93
06902 Sophia Antipolis Cedex, FRANCE
`Yves.Bertot@sophia.inria.fr`

1. Introduction

La théorie des types se répand comme outil de description d'algorithmes et même comme outil de programmation. L'un des principaux avantages de ce cadre formel est qu'il contient naturellement un langage fonctionnel. En particulier, le calcul des constructions sert de base au système Coq [2] qui fournit ainsi des moyens d'expérimenter rapidement avec ces concepts. Il est donc possible de décrire rapidement des programmes, de les tester, et de vérifier leurs propriétés logiques. Des outils d'extraction [8] peuvent être adjoints pour traduire ces descriptions formelles d'algorithmes en programmes purement fonctionnels, rédigés dans des langages comme OCaml [7, 3], qui peuvent ensuite être compilés et être utilisés directement. Une autre approche possible est de décrire directement dans le cadre fonctionnel des programmes présentant des traits impératifs et de raisonner sur ces programmes. La question que nous abordons ici est celle de savoir si les deux approches sont incompatibles, ou si l'approche fonctionnelle, plus aisée, peut aussi contribuer à obtenir rapidement des programmes impératifs certifiés. Notre propos n'est pas de promouvoir de nouveaux outils, puisqu'ils existent, mais de comprendre les similitudes entre les différents points de vue que l'on peut avoir sur un même programme.

Dans cet article nous étudions la description formelle de deux algorithmes dans le système Coq, en profitant de l'outil `Correctness` fourni par J.-C. Filliâtre [4, 5] pour conduire l'étude jusqu'à une implémentation impérative. Le premier algorithme est un algorithme de tri et la structure de contrôle repose sur la notion de boucle, tandis que le deuxième algorithme est un algorithme d'arithmétique sur les grands nombres et la structure de contrôle est donnée par des appels de procédures récursives avec effets de bord.

Dans les deux cas, la structure de données principale est décrite à l'aide de tableaux et le programme obtenu est le plus proche possible d'algorithmes finement optimisés programmables dans les langages efficaces les plus populaires, même si ces langages ne sont pas toujours applicatifs. Néanmoins, nous tentons de souligner l'intérêt que peut aborder une modélisation en deux temps, où le premier temps conserve un recul par rapport à l'implémentation et décrit des données mathématiques (comme des grands entiers) avant de parler de leur représentation dans la mémoire d'un ordinateur.

Nous tirons deux leçons de ces expériences. La première leçon est que nous pensons qu'il existe une méthode permettant de relier les invariants de boucles avec les fonctions récursives représentées par ces boucles. Faute de recul, nous ne parvenons pas à dégager les principes de cette méthode mais nous en donnons deux instances. La seconde leçon est que le travail dans une mémoire représentée par des tableaux dans lesquels on accède segment par segment demande de faire de nombreuses démonstrations d'appartenance d'indices dans des intervalles. Fort heureusement, les procédures de démonstration automatique sur l'arithmétique linéaire permettent déjà de s'affranchir de ces démonstrations.

2. Tri par insertion

Le tri par insertion est l'un des algorithmes de tri les plus simples qu'il soit possible de concevoir. Il fonctionne en ajoutant les éléments à trier progressivement dans une séquence déjà triée. Sa complexité dans le pire cas n'est pas très bonne (mais pas pire que pour le fameux *Quicksort*), mais il a néanmoins des caractéristiques qui le rendent souvent préférable. Premièrement, cet algorithme est stable, c'est à dire que si la liste donnée en argument est déjà triée, alors la liste donnée en résultat le sera également. Deuxièmement, si la liste est « pratiquement » triée, sa complexité devient très proche d'une complexité linéaire. Deux propriétés que n'a pas *Quicksort* [9].

2.1. Spécifications

Une séquence de nombres est triée si son premier élément est plus petit que le second et ainsi de suite. Si la séquence est vide ou réduite à un élément, il est naturel de considérer qu'elle est effectivement triée.

Pour les spécifications dans un système comme Coq le plus naturel pour représenter une séquence d'objets de même type est d'utiliser la notion de liste. Nous dirons qu'une liste à aux moins deux éléments est triée si son premier élément est plus petit que son second élément et si la liste commençant au deuxième élément est elle-même triée. Ici encore, la liste vide et les listes à un seul élément sont automatiquement triées. Ceci s'exprime avec la définition inductive suivante:

$$\forall n_1, n_2, \forall l.(n_1 \leq n_2 \wedge \mathit{sorted}(n_2 \cdot l)) \Rightarrow \mathit{sorted}(n_1 \cdot n_2 \cdot l)$$

$$\forall n.\mathit{sorted}([n]) \qquad \mathit{sorted}(\emptyset)$$

Lorsque l'on veut trier une séquence de nombres, on ne se contente pas d'exiger un résultat qui est une séquence triée (la séquence vide suffirait) mais on veut également que le résultat soit une permutation de l'entrée. Par permutation, on entend que le résultat soit obtenu à partir de l'entrée en échangeant des éléments deux-à-deux, et en répétant cette opération un nombre arbitraire de fois. On peut exprimer ceci avec la relation de permutation, qui se définit comme une propriété inductive en utilisant les deux affirmations suivantes:

$$\forall l.\mathit{permutation}(l, l)$$

$$\forall l_1, l_2, l_3, l_4, \forall n_1, n_2.$$
$$\mathit{permutation}(l_1 \cdot n_2 \cdot l_2 \cdot n_1 \cdot l_3, l_4) \Rightarrow \mathit{permutation}(l_1 \cdot n_1 \cdot l_2 \cdot n_2 \cdot l_3, l_4)$$

2.2. Implémentation fonctionnelle

Le tri par insertion utilise deux fonctions récursives. La première effectue simplement l'insertion dans une liste, tandis que la seconde répète le procédé d'insertion pour tous les éléments de la liste en entrée. Si l'on insère un élément dans une liste triée, alors le résultat est encore trié. Le processus commence par une insertion dans une liste vide, qui est déjà triée et, de proche en proche, le résultat doit donc être une liste triée. Les deux fonctions peuvent se décrire de la façon suivante (en suivant la syntaxe du système Coq):

```
Fixpoint insert_plain [n:nat; l:(list nat)]:(list nat):=
  Cases l of
    nil => (cons n (nil nat))
  | (cons n1 l1) =>
       Cases (le_lt_dec n n1) of
         (left _) => (cons n (cons n1 l1))
```

```
      | (right _) => (cons n1 (insert_plain n l1))
      end
  end.
```

L'expression `Cases (le_lt_dec n n1)...` correspond à une expression conditionnelle: l'expression `(cons n (cons n1 l1))` est la valeur retournée lorsque `n` est plus petit que `n1` et l'expression `(cons n1 (insert_plain n l1))` est la valeur retournée sinon.

Lorsque la fonction d'insertion est définie, il est facile de fournir la fonction de tri, décrite par les quelques lignes suivantes:

```
Fixpoint sort_plain [l:(list nat)] : (list nat) :=
  Cases l of
    nil => (nil nat)
  | (cons n1 l1) => (insert_plain n1 (sort_plain l1))
  end.
```

La certification d'un programme fonctionnel suit généralement la structure de ce programme: on traite d'abord la fonction d'insertion et l'on se repose sur la démonstration de théorèmes qui décrivent la relation entre les comportements de la fonction et les notions de listes triées et de permutations. Mais comme c'est souvent le cas dans les preuves par récurrence, on est amené une propriété plus forte. Ici, on a besoin de montrer que le premier élément du résultat après avoir inséré le nombre n dans la liste l est soit le nombre n lui-même, soit le premier élément de la liste l. Nous résumons cette propriété par une propriété inductive *first_prop*, qui peut être définie par les deux constructeurs suivants:

$$\forall n, n2, \forall l_1, l_2 \mathit{first\_prop}(n, n_2 \cdot l_1, n_2 \cdot l_2)$$

$$\forall n, \forall l_1, l_2 \mathit{first\_prop}(n, l_1, n \cdot l_2)$$

Ainsi deux premiers théorèmes indiquent ce qui se passe lorsque l'insertion se fait en tête.

$$\forall n, n_1, \forall l. \mathit{sorted}(n_1 \cdot l) \rightarrow \mathit{sorted}(n \cdot n_1 \cdot l) \wedge \mathit{first\_prop}(n, n_1 \cdot l, n \cdot n_1 \cdot l)$$

$$\forall n_1, \forall l_1 l_2. \mathit{permutation}(l_1, l_2) \Rightarrow \mathit{permutation}(n_1 \cdot l_1, n_1 \cdot l_2)$$

Deux autres théorèmes décrivent ce qui se passe lorsque l'insertion se fait plus loin qu'à la tête de la liste.

$$\begin{aligned}\forall n, n_1, \forall l_1, l_2. (\mathit{sorted}(n_1 \cdot l_1) \wedge \neg n \leq n_1 \wedge \mathit{sorted}(l_2) \wedge \mathit{first\_prop}(n, l_1, l_2))& \\ \Rightarrow \mathit{sorted}(n_1 \cdot l_2) \wedge \mathit{first\_prop}(n, n_1 \cdot l_1, n_1 \cdot l_2)&\end{aligned}$$

$$\forall l_1, l_2, \forall n, n_1. \mathit{permutation}(n \cdot l_1, l_2) \Rightarrow \mathit{permutation}(n \cdot n_1 \cdot l_1, n_1 \cdot l_2)$$

D'autres théorèmes sont également fournis pour décrire les cas limites, mais nous ne les fournirons pas ici.

En pratique, la certification du programme fonctionnel décrit les propriétés assurées *dans le futur* ou bien *en fin de calcul*. En effet, lorsque l'on démontre que la fonction d'insertion retourne bien une liste triée, on raisonne sur la valeur retournée par les appels récursifs de la même fonction, et on utilise des hypothèses de récurrence pour exprimer que les résultats de ces appels récursifs satisfont déjà la propriété recherchée vis-à-vis de leurs données. Bien sûr, cette distinction entre *futur* et *présent* a peu de sens en programmation fonctionnelle, puisque ce type de programmation, sans effet de bord, permet de raisonner sur des valeurs immuables. Mais nous verrons que c'est justement une distinction importante avec la programmation impérative.

2.3. Implémentation impérative

Le premier changement qui apparait dans la programmation impérative est l'introduction d'un caractère temporel dans la logique. Il n'est plus question de considérer les variables comme des valeurs immuables, mais comme des expressions qui ont un sens dans un état donné. La logique de Hoare et le calcul de pré-condition de Dijkstra permettent de ramener toutes les considérations logiques dans le même état et affranchissent donc le programmeur de raisonner sur les transitions entre plusieurs états. Mais ceci a un prix: il est plus difficile d'écrire des formules logiques qui mettent en relation les entrées et les sorties d'un programme, car, pour de nombreux programmes (parmi les plus efficaces), il n'est pas prévu que ces données cohabitent en mémoire, la donnée en entrée étant détruite avant que la donnée en sortie ne soit produite.

Dans les programmes impératifs, une nouvelle structure de données apparait également: la structure de tableau. Un tableau peut bien sûr être représenté par une liste, mais certaines opérations naturelles sur les tableaux n'ont qu'une représentation complexe dans les listes. Par exemple, c'est par un jeu simple de translation d'indice ou de borne que l'on peut récupérer un sous-segment d'un tableau. Du point de vue de la complexité algorithmique, cette opération se fait donc en temps constant. En revanche, récupérer un sous-segment d'une liste se fait au moins en temps linéaire, soit vis-à-vis de la longueur du segment délaissé, soit vis-à-vis de la longueur du tableau initial. On peut également considérer les tableaux comme des cas particuliers de fonctions des entiers vers le type des éléments, c'est d'ailleurs ce qui est fait dans les contributions du système Coq.

La principale structure de contrôle permettant la répétition de calculs est la boucle, par opposition à l'appel récursif de fonctions en programmation fonctionnelle. Pour mettre en correspondance des propriétés et des programmes, on annote les programmes avec des formules logiques. En particulier, les boucles sont annotées avec des propriétés qui doivent être vérifiées au début et à la fin de chaque itération: les *invariants*.

Indépendamment des assertions que l'on sera amené à introduire pour certifier le programme, le tri par insertion se décrit par le programme extrêmement simple suivant (`b` est la longueur du tableau à trier, le tableau lui-même est décrit par la variable `a`, `x` et `y` sont des variables entières utilisées comme indices dans le tableau, `f` est une variable booléenne, utilisée pour indiquer si le calcul d'insertion est terminé).

```
 if b >= 2 then
 begin
  x := b - 2;
  while 0 <= x do
   v := a[x]; y :=  x+1; f := true;
   while f do
    if v =< a[y] then begin f := false; a[y - 1] := v end
    else
     begin
      a[y - 1] := a[y];
      if y = b-1 then begin f := false; a[y] := v end
      else
       y:=y+1
     end
   done;
   x:=x-1
  done
end
```

Ce programme comporte deux boucles. La boucle externe correspond à la fonction `sort_plain` tandis que la boucle interne correspond à la fonction `insert_plain`.

Le point que nous cherchons à défendre ici est que le travail préliminaire sur une description fonctionnelle de l'algorithme nous donne un outil avantageux pour aborder la certification de ce programme. Observons d'abord l'annotation que nous allons ajouter à la boucle externe (à partir de maintenant nous utiliserons la syntaxe imposée par l'outil `Correctness`). Cette annotation est ajoutée juste après la ligne `while 0 <= x do`.

```
'-1 <= x <= b-2' /\
 ((i:Z)'0<= i < x+1' -> #a[i]=#a@init[i])/\
 (atl' a 'x+1')=(sort_plain (atl' a@init 'x+1'))
```

Cet invariant comporte trois parties. La première partie est la suivante:

```
'-1 <= x <= b-2'
```

Cette partie est nécessaire pour justifier que tous les accès effectués dans le tableau à l'index `x` le sont entre les bornes du tableau. La seconde partie de l'invariant est la suivante:

```
((i:Z)'0<= i < x+1' -> #a[i]=#a@init[i])
```

Cette partie indique que la partie du tableau comprise entre les indices `0` et `x` n'a pas encore été modifiée. Cette partie répond à un besoin spécifique de la programmation impérative: puisque l'algorithme évolue en modifiant certaines parties de la mémoire, il est nécessaire d'exprimer explicitement quelles parties ne sont pas modifiées pendant certaines phases de calcul. La troisième partie de l'invariant est de la forme suivante:

```
(atl' a 'x+1')=(sort_plain (atl' a@init 'x+1')
```

Cette partie exprime simplement que le segment du tableau compris entre les indices `x+1` et `b` est le résultat du tri du segment de tableau compris initialement entre les mêmes indices. Cette partie repose sur la fonction `atl'` qui associe à un tableau de longueur `b` et un indice `x` la liste contenant la séquence de valeurs `a[x]`, `a[x+1]`, ..., `a[b-1]`. Nous voyons ici comment la fonction `sort_plain` nous permet d'exprimer simplement l'invariant dont nous avons besoin pour la boucle externe du tri. Mais cette simplicité peut être trompeuse.

L'invariant que nous ajoutons pour la boucle interne a la forme suivante:

```
invariant 'x+1 <= y < b' /\
 ((m:(array b nat))
  ((i:Z)'x<=i<y-1'->#m[i]=#a[i])->
  (atl' m 'y-1')=(insert_plain v (atl' a@s y))->
  (atl' m x)=(insert_plain v (atl' a@s 'x+1'))) /\
 ((i:Z)'0<=i<x' -> #a[i]=#a@s[i]) /\
 (f = false->(atl' a 'y-1')=(insert_plain v (atl' a@s y))) /\
 (f = true -> (atl' a y)=(atl' a@s y))
```

Cet invariant est ajouté après la ligne `while f do`, il contient encore plusieurs parties, qui ont été mises au point au fur-et-à-mesure des progrès du travail de certification, en fonction des obligations de preuve fournies par les outils automatiques. La première partie,

```
'x+1 <= y < b'
```

sert à indiquer que les accès dans le tableau à l'index `y` ont bien lieu entre les bornes du tableau, comme pour la boucle externe. La seconde partie a la forme suivante:

```
((m:(array b nat))
 ((i:Z)`x<=i<y-1`->#m[i]=#a[i])->
 (atl' m `y-1`)=(insert_plain v (atl' a@s y))->
 (atl' m x)=(insert_plain v (atl' a@s `x+1`)))
```

Cette partie est apparemment beaucoup plus complexe que ce que nous avons vu pour le premier invariant. Pour commencer, plutôt que de raisonner sur l'état courant de la mémoire, cette formule logique décrit les propriétés d'un tableau hypothétique `m` vérifiant les propriétés suivantes:

1. le fragment de tableau compris entre les indices `x` et `y-1` serait identique au même fragment dans la mémoire à l'instant considéré,

2. le fragment de tableau compris entre les indices `y-1` et `b` serait le résultat de l'insertion de la valeur `v` dans le tableau compris entre les indices `y` et `b` avant d'entrer dans la boucle interne (instant dénoté par `s`).

Alors, ce tableau aurait la propriété de contenir le résultat de l'insertion de la valeur `v` dans la séquence entre les indices `x+1` et `b` dans le tableau qui existait en mémoire avant d'entrer dans la boucle interne (instant `s`).

L'utilisation de cette forme d'invariant répond à un besoin précis. A un instant donné au cours de l'exécution de la boucle interne, le résultat de l'insertion de la valeur `v` dans une liste n'a pas encore été calculé. Ce sont les itérations restantes qui vont se charger de calculer l'insertion de `v` dans le fragment de liste qui n'a pas encore été vu. Nous avons donc besoin de parler de l'état mémoire *dans le futur* pour établir une relation avec la présentation fonctionnelle de l'algorithme. Ceci semble aller à l'encontre de la notion d'invariant fournie par la logique de Hoare.

En d'autres termes, s'il reste k itérations de la boucle interne à effectuer et que chaque itération prend une unité de temps, nous démontrons que le calcul qui reste à effectuer entre les instants t et $t+k$ est correct sous l'hypothèse que le calcul effectué entre les instants $t+1$ $t+k$ est correct. En quelque sorte, nous retournons l'échelle des temps et en déduisons la correction à l'instant $t+k$ de la correction à l'instant $t+k+1$, alors que le calcul d'invariant en logique de Hoare demande de démontrer la conservation de l'invariant à l'instant $t+1$ s'il est déjà assuré à l'instant t. Ce retournement est obtenu simplement en introduisant une implication dans l'invariant, dont la conclusion est la spécification qui doit être assurée en fin de boucle F et dont la prémisse est la propriété assurée par les itérations restantes après la date t, $P(t)$. Ainsi, prouver la conservation de l'invariant exige de prouver la formule suivante:

$$(P(t) \Rightarrow F) \Rightarrow (P(t+1) \Rightarrow F)$$

ce qui est équivalent à la formule suivante:

$$P(t+1) \Rightarrow P(t)$$

L'invariant de la boucle interne contient d'autres parties, la troisième partie est de la forme suivante:

```
((i:Z)`0<=i<x` -> #a[i]=#a@s[i])
```

Cet invariant exprime encore qu'un certain segment de la mémoire est inchangé depuis l'entrée dans la boucle interne. La quatrième partie de l'invariant est de la forme suivante:

```
(f = false->(atl' a `y-1`)=(insert_plain v (atl' a@s y)))
```

Cet invariant exprime que si l'on sort de la boucle, c'est que l'insertion est bien terminée. La cinquième partie de l'invariant est de la forme suivante:

```
(f = true -> (atl' a y)=(atl' a@s y))
```

Cette partie exprime également qu'une certaine partie du tableau est inchangée depuis l'entrée dans la boucle interne.

Ainsi l'invariant peut être décomposé en trois parties:

1. une partie exprime des relations de bornes sur les indices utilisés pour contrôler la boucle, `x` dans la première boucle et `y` dans la seconde,

2. une seconde partie est utilisée pour établir une relation entre les calculs effectués dans le programme impératif et les calculs effectués dans le programme fonctionnel. Dans certains cas, on peut avoir besoin d'une la technique de *retournement* du temps pour établir réellement cette correspondance,

3. une troisième partie est utilisée pour établir les parties de la mémoire qui sont inchangées entre des instants précis du calcul.

Dans un travail antérieur, Nicolas Magaud et J.-C. Filliâtre ont déjà étudié le tri par insertion [6]. Leur travail est plus direct, au sens où il ne passe pas par une modélisation fonctionnelle. Les théorèmes qu'ils utilisent font apparaitre un concept supplémentaire, celui de concaténation de listes (représenté par la notation $l_1 \cdot l_2$ dans le présent article). La justification principale utilisée dans leur travail est un théorème qui établi sous quelles conditions la concaténation de deux listes triées peut être triée.

$$\forall l_1, l_2, \forall n. sorted(l_1 \cdot n) \wedge sorted(n \cdot l_2) \Rightarrow sorted(l_1 \cdot n \cdot l_2).$$

Notre utilisation du *retournement de temps* nous évite d'avoir à démontrer ce théorème supplémentaire.

3. Racine carrée des grands nombres

Un autre exemple d'algorithmes qui font un usage extensif de tableaux est celui des algorithmes de calcul sur les grands nombres. Un grand nombre est alors représenté par un tableau de petits nombres, chacun de ces petits nombres étant borné par le plus grand nombre représentable dans une cellule de mémoire. Dans la collaboration AOC avec les équipes *Spaces* et *Arénaire* organisée grâce au soutien des *actions de recherche concertées* AOC proposées par la direction scientifique de l'INRIA, Nicolas Magaud, Paul Zimmermann et moi-même avons eu l'occasion d'étudier la certification d'un algorithme de calcul de racine carrée pour de tels grands nombres [10]. Cet algorithme est intéressant car il contient quelques optimisations spectaculaires, dont certaines ont bien sûr trait à l'arithmétique, mais d'autres reposent sur une connaissance précise de l'occupation de la mémoire dans l'algorithme.

3.1. Spécifications

Pris comme un problème sur les entiers relatifs, le calcul de la racine carrée doit être résolu par un programme qui accepte en entrée un nombre entier positif N et retourne deux nombres entiers positifs S et R, vérifiant les équations et inéquations suivantes:

1. $N = S^2 + R$,

2. $0 \leq S$,

3. $0 \leq R \leq 2S$.

Il suffit d'un calcul rapide pour se convaincre que $R \leq 2S$ suffit à assurer que l'on a bien $N < (S+1)^2$.

En terme de consommation mémoire, si le nombre N est représentable dans un segment de tableau de longueur $2n$, alors la racine carrée S sera représentable dans un segment de longueur n. Pour R le problème est légèrement plus compliqué. Comme R est seulement inférieur à $2S$, nous n'avons pas le résultat que R est représentable dans un segment de longueur n, néanmoins, si nous notons β la base, c'est-à-dire le nombre maximum représentable dans une cellule mémoire plus un, nous avons $R = q\beta^n + \overline{R}$, où q vaut 1 ou 0 et $0 \leq \overline{R} < \beta^n$, donc $\overline{R}$ est bien représentable dans un segment de longueur n.

3.2. Description abstraite

L'algorithme fonctionne en coupant la taille du nombre à traiter en deux, séparant ainsi ce nombre en une partie de poids fort et une partie de poids faible. Un appel récursif de l'algorithme permet d'obtenir la racine carrée de la partie de poids fort et un reste pour cette racine carrée. On recombine ensuite ce reste avec la partie de poids faible pour la diviser par le double de la racine carrée de la partie de poids fort. Le quotient de cette division, adjoint à la racine carrée de la partie de poids fort, nous permet d'obtenir une approximation très précise de la racine carrée cherchée. Le calcul se termine en déterminant le reste, ce qui permet également de remplacer l'approximation trouvée par une valeur exacte.

Décrivons ce procédé de manière plus formelle. Le nombre N est supposé plus petit que β^{2n}. Pour couper N en une partie de poids fort et une partie de poids faible, nous déterminons simplement deux nombres H et L tels que $\beta^{2n} = (HL)^2$ (en pratique, on prendra $l = n/2$ arrondi par défaut, $h = n/2$ arrondi par excès, $L = \beta^l$ et $H = \beta^h$). On déterminera les parties de poids fort et de poids faible en divisant N par L^2:

$$N = N' \times L^2 + N'' \qquad 0 \leq N'' < L^2$$

L'appel récursif de l'algorithme de calcul de racine carrée sur N' donne deux valeurs S' et R' telles que:

$$N' = S'^2 + R' \qquad 0 \leq R' \leq 2S'$$

Ensuite on décompose encore N'' en deux parties N_1 et N_0, en utilisant L comme séparation:

$$N'' = N_1 \times L + N_0 \qquad 0 \leq N_0 < L$$

et on divise le nombre $R' \times L + N_1$ par $2S'$ pour obtenir un quotient Q et un reste R'':

$$R' \times L + N_1 = Q \times (2S') + R'' \qquad 0 \leq R'' < (2S')$$

Tous ces calculs nous mènent au résultat suivant:

$$\begin{aligned}
N &= N' \times L^2 + N_1 \times L + N_0 \\
&= (S'^2 + R') \times L^2 + N_1 \times L + N_0 \\
&= (S' \times L)^2 + (R' \times L + N_1) \times L + N_0 \\
&= (S' \times L)^2 + (Q \times 2S' + R'') \times L + N_0 \\
&= (S' \times L)^2 + 2Q \times (S' \times L) + R'' \times L + N_0 \\
&= (S' \times L)^2 + 2Q \times (S' \times L) + Q^2 - Q^2 + R'' \times L + N_0 \\
&= (S' \times L + Q)^2 + (R'' \times L + N_0 - Q^2)
\end{aligned}$$

Ce raisonnement suggère que $S'L + Q$ est une bonne approximation de la racine carrée de N. Pour savoir à quel point cette approximation est bonne, il faut vérifier si les inéquation suivantes sont bien vérifiées.

$$0 \leq R'' \times L + N_0 - Q^2 \leq 2(S'L + Q).$$

C'est ici qu'intervient la première des optimisations spectaculaires de cet algorithme. Si $L \leq H$ et si $(HL)^2 \leq 4N$, alors on peut démontrer que $S_1 = S'L + Q$ est soit la valeur recherchée, soit une approximation d'une seule unité. C'est le signe de l'expression

$$R_1 = L + N_0 - Q^2$$

qui permet de savoir si on est dans le cas exact ou dans le cas d'une approximation par excès. Si l'on est dans le cas d'une approximation par excès, il suffit de calculer les valeurs S_2 et R_2 suivantes pour finir le calcul:

$$S_2 = S_1 - 1 \qquad R_2 = R_1 + 2S_1 - 1$$

La démonstration de cette propriété requiert quelques calculs arithmétiques assez précis que nous ne détaillerons pas ici.

Pour que l'optimisation fonctionne à chaque étape, il faut vérifier que la propriété $(HL)^2 \leq 4N$ se transmet bien lors de l'appel récursif, ce qui se traduit ici par la contrainte:

$$H^2 \leq 4N'.$$

Comme les nombres H et L sont calculés à chaque appel récursif en fonction de β et n, nous obtenons en fait la contrainte que β doit être pair (cette contrainte est apparue dans la formalisation, Paul Zimmermann qui travaille sur une implémentation beaucoup plus précise où β est une puissance de 2 et vérifie forcément la contrainte n'était pas conscient de cette possibilité: en fait l'algorithme pourrait fonctionner si la représentation en mémoire reposait sur la base 10 plutôt que sur une base qui est une puissance de 2).

Notons que nous n'avons décrit ici que les calculs effectués dans une itération d'une fonction récursive, en supposant que l'appel récursif retourne lui-même une valeur correcte pour la racine carrée et le reste. Le cas de base de cette récursion correspond au cas où n vaut 1 et ne peut plus être divisé. Dans ce cas, on suppose l'existence d'une fonction de racine carrée capable de faire le calcul de la racine carrée des nombres codés dans deux cellules mémoires. Un tel algorithme pourrait par exemple être obtenu par une recherche dans une table.

3.3. Implémentation fonctionnelle

L'implémentation fonctionnelle de l'algorithme est possible, mais elle n'est ni aussi facile ni aussi directe que pour le tri par insertion. La raison pour cette complexité provient de la forme de l'algorithme qui n'est pas récursif structurel, puisque la récursion n'est structurelle par rapport à N ni structurelle par rapport à n. Nous faisons donc intervenir une récursion bien fondée sur n pour la relation $<$. Dans nos expériences, cette implémentation fonctionnelle a été effectuée avant que nous ayons mis au point avec Antonia Balaa une technique simplifiant la description de telles fonctions récursives [1]. Cette fonction peut être extraite, mais ne présente qu'un intérêt limité dans la mesure où elle ne tire pas de réel bénéfice de la possibilité de représenter les grands nombres dans des segments de tableaux. Toutefois, la question du transfert de l'invariant $(HL)^2 \leq 4N$ vers $H^2 \leq 4N'$ prend tout son sens ici, et construire l'implémentation fonctionnelle permet d'assurer que l'on n'oublie pas cette partie de la modélisation.

3.4. Une spécification plus concrète

Pour une spécification concrète, nous allons encore nous munir d'une fonction qui établit la relation entre le niveau concret et le niveau abstrait. Ici, il s'agit d'une fonction qui prend en paramètres un tableau de nombres bornés et deux indices dans ce tableau et retourne le nombre représenté en mémoire entre ces deux indices. La spécification est maintenant enrichie: la procédure que nous voulons décrire reçoit en paramètres les indices décrivant où se trouve la donnée en entrée et les indices décrivant où fournir le résultat, mais la procédure doit aussi assurer qu'elle ne modifie pas la mémoire en dehors de régions précises: on ne veut pas qu'un calcul de racine carrée sur un nombre dans une portion de la mémoire ne corrompe les nombres conservés ailleurs en mémoire. De toutes façons, cette propriété que la mémoire reste inchangée en dehors de zones d'actions connues jouera également un rôle pour la certification de la fonction de racine carrée elle-même.

Dans la mémoire, nous supposerons que la donnée (N) est fournie entre les indices pos_a et pos_a+2n et que l'on demande de fournir la racine carrée (S) entre les indices pos_s et pos_s+n et le reste (R) entre les indices pos_a et pos_a+n. Comme nous l'avons vu, R peut être plus grand que β^n, mais dans ce cas le dépassement peut être décrit par un seul nombre compris entre 0 et 1. En programmation C ou en programmation Ocaml, ce nombre représentant le dépassement peut simplement être la valeur retournée par la fonction, les autres effets de la fonction étant le résultat d'effets de bord.

La consommation mémoire de cet algorithme est impressionnante: la donnée peut être représentée dans $2n$ cellules mémoires et les résultats peuvent être représentés dans $2n+1$ cellules mémoires. Le calcul total s'effectuera en ne modifiant au plus que $3n+k$ cellules mémoires, où k est un nombre constant, plus éventuellement une consommation en pile proportionnelle à $log(n)$.

3.5. Implémentation impérative

Le premier aspect de l'implémentation impérative est que plusieurs des opérations de division ou de multiplication par L ou L^2 apparaissant dans la description abstraite ne sont tout simplement pas des opérations dans l'implémentation, mais simplement des changements de point de vue. Si $n=h+l$, $H=\beta^h$, $L=\beta^l$, et N est le nombre représenté en mémoire entre les indices pos_a et pos_a+2n, alors N' est simplement le nombre représenté en mémoire entre les indices pos_a+2l et pos_a+2n, N_1 est le nombre entre les indices pos_a+l et pos_a+2l et N_0 est le nombre entre les indices pos_a et pos_a+l. De même pour construire le nombre $R'L+N1$, il suffit que l'appel récursif de la procédure de calcul de racine carrée retourne son reste entre les positions pos_a+2l et pos_a+2l+h sans modifier la mémoire entre les positions pos_a+l et pos_a+2l pour que le nombre $R'L+N1$ soit représenté en mémoire entre les positions pos_a+l et pos_a+2l+h. Enfin, le nombre $S'L+Q$ est pratiquement déjà en mémoire si l'on demande à recevoir la racine carrée de l'appel récursif à l'adresse pos_s+l et le quotient de la division par $2S'$ à l'adresse pos_s. En plusieurs endroits il faut faire attention aux débordements possibles, mais la juxtaposition de données en mémoire suffit souvent à représenter les divisions et multiplications par L.

La table d'utilisation de la mémoire dans le temps est fournie dans la figure 1. Cette table ne donne pas tous les détails de l'algorithme dans la mesure où les cas de débordement ne sont pas représentés. Néanmoins, deux aspects méritent d'être notés.

Premièrement, la division par $2S'$ se fait en deux étapes pour éviter de construire en mémoire le nombre $2S'$. La méthode employée est plutôt de diviser par S', puis de diviser à nouveau par 2. La deuxième division est d'ailleurs effectuée par un simple décalage bit par bit, après avoir vérifié que le nombre que l'on divise par 2 est pair. Si ce nombre n'est pas pair, il faut ajouter S' au reste.

Deuxièmement, il est nécessaire de calculer le nombre Q^2 pour effectuer la soustraction finale. Il y a justement une zone mémoire disponible pour représenter ce nombre au moment où le besoin de cette zone mémoire se fait sentir.

La démonstration du programme impératif s'effectue de façon assez naturelle en associant les valeurs entières utilisées dans la démonstration abstraite avec les nombres représentés en mémoire aux instants indiqués dans la table de la figure 1. Néanmoins, chacune des opérations indique qu'elle laisse certaines parties de la mémoire inchangées avec des formules de la forme suivante ($m@$ représente la mémoire avant l'opération).

$$\forall i.\neg(pos_1 \leq i < pos_1 + l_1) \Rightarrow \neg(pos_2 \leq i < pos_2 + l_2) \Rightarrow m[i] = m@[i]$$

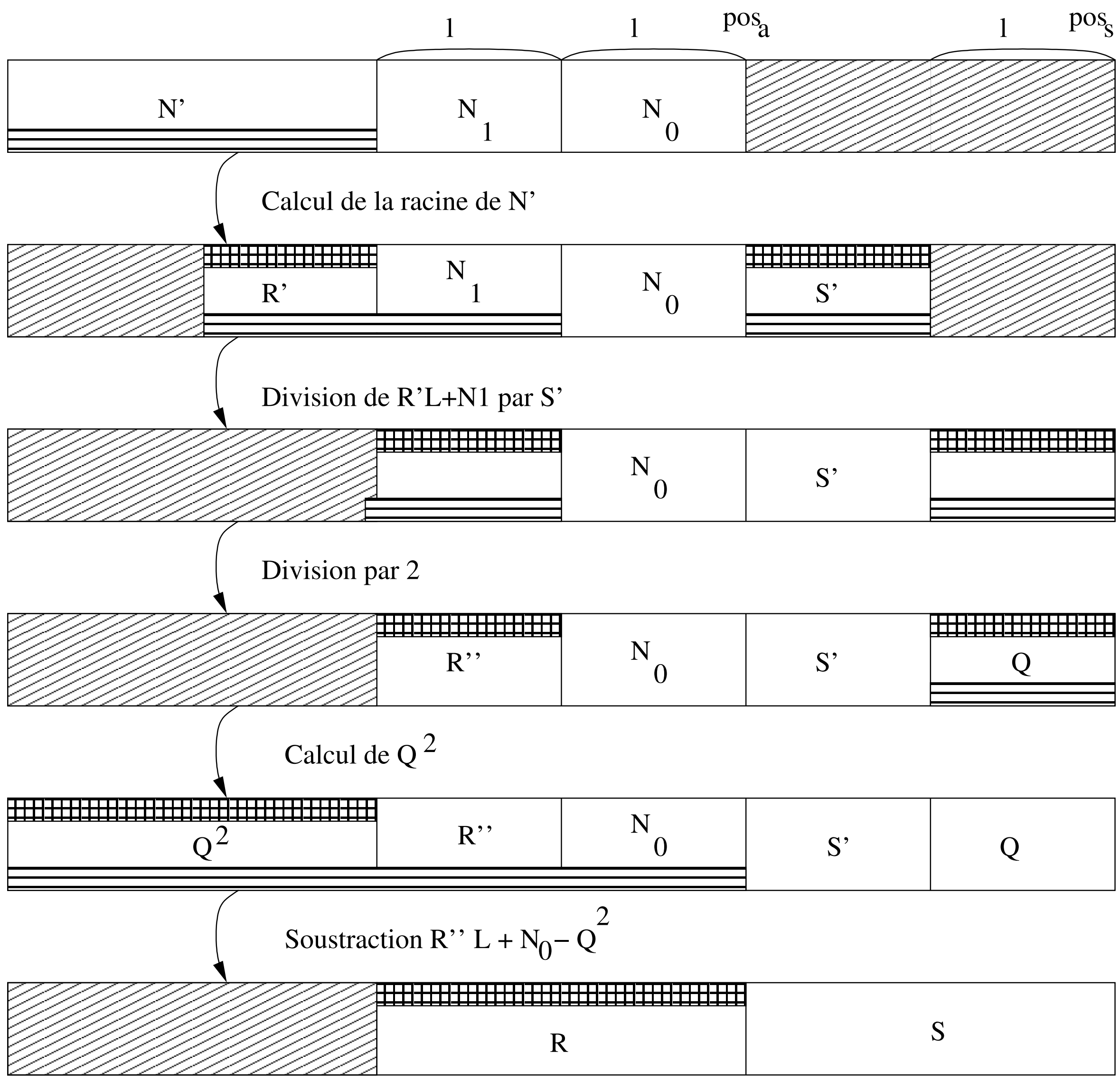

Figure 1: Utilisation de la mémoire dans le temps. Les hachures obliques représentent des données inutiles, les hachures horizontales représentent les données utilisées par la prochaine opération et les hachures à double sens représentent les données produites par chaque opération. Nous avons juxtaposé la zone de mémoire pour la donnée en entrée et pour le résultat mais cette juxtaposition n'est pas nécessaire.

Ainsi chaque opération comporte également un intervalle ou une réunion d'intervalles où la mémoire reste inchangée, ici cette réunion d'intervalle est décrite par la condition

$$\neg(pos_1 \leq i < pos_1 + l_1) \wedge \neg(pos_2 \leq i < pos_2 + l_2).$$

Pour assurer qu'une certaine donnée produite à l'étape k est encore présente en mémoire à l'étape k' il sera nécessaire de montrer que tous les indices utilisés dans la représentation de cette donnée vérifient les conditions de non-changement pour toutes les opérations qui ont eu lieu entre ces deux étapes.

En pratique, toutes les vérifications reliées au propriétés de mémoire inchangées ont été aisément automatisées grâce à la procédure de démonstration automatique sur l'arithmétique linéaire fournie dans les librairies de Coq: `Omega`.

Cet algorithme est similaire à l'algorithme de tri décrit plus haut dans la mesure où l'on travaille par effet de bord sur un tableau et l'on contrôle précisément les zones de la mémoire qui sont modifiées ou inchangées à chaque étape. En revanche, il y a une différence notable dans la façon dont la répétition des calculs est organisée. Dans l'algorithme de tri la répétition repose sur une boucle et la certification sur un invariant de boucle. Ici, la répétition est décrite par un appel récursif, comme en programmation fonctionnelle pure.

4. Conclusion

Le propos de cet article n'était pas de démontrer l'avantage fourni par la description fonctionnelle des algorithmes lorsque l'on cherche à les formaliser pour montrer leur conformité à des spécifications. Cet avantage est en effet déjà bien perçu par la majorité de la communauté, surtout si l'on s'adresse à des praticiens de la programmation fonctionnelle sous toutes ses formes. Surtout nous tenons à souligner que la description fonctionnelle n'est pas incompatible avec un objectif final qui serait de fournir une description formelle précise pour une implémentation impérative et que le travail effectué dans cette description fonctionnelle est souvent adapté pour la certification du programme impératif, même cette dernière pourra exiger des efforts supplémentaires.

Que la vérification formelle du programme impératif demande un travail plus important de démonstration n'est pas surprenant, puisque c'est aussi l'argument utilisé pour défendre la programmation fonctionnelle comme une méthode de programmation plus sûre que la programmation impérative. Puisque la programmation fonctionnelle est plus sûre, il est naturel que le travail de certification d'un programme fonctionnel soit strictement moindre par comparaison au travail de certification d'un programme impératif.

Néanmoins, la simplicité de la présentation faite dans cet article ne doit pas leurrer le lecteur: vérifier entièrement un programme comme le programme impératif de calcul de racine carrée décrit ici a quand même pris plusieurs mois de travail intensif. Une partie du temps a été passé à maitriser les barrières de complexité inhérentes aux outils de description de programme. Dans nos expériences, nous avons en particulier été ralentis par les barrières de complexité imposées par les outils de vérification. En effet, l'énoncé exact des invariants est obtenu dans un travail interactif avec le système de démonstration qui génère de nouvelles obligations de preuve à chaque proposition de l'utilisateur. La boucle « rédaction du programme et des invariants, compilation des obligations de preuve correspondantes, puis tentative de démonstration de ces obligations » remplace la boucle « rédaction des programmes, compilation puis test » d'un travail de programmation traditionnel. Ce travail interactif n'est praticable que si le temps de réponse du système de preuve est inférieur à la minute, or nous avons parfois pu noter un temps de calcul exponentiel en fonction du nombre d'annotations portées par le programme.

Une autre opération parfois faite sur les algorithmes est l'opération qui consiste à enlever leur caractère récursif pour leur donner un caractère itératif. Le changement d'un algorithme récursif à un algorithme itératif peut parfois s'accompagner d'un changement de structure de calcul, même si la justification initiale de l'algorithme reste la même. Ceci apparait dans les implémentations itératives de l'algorithme de tri par fusion décrites dans [9]. En effet, l'algorithme récursif fonctionne mieux si l'on choisit les longueurs de sous-séquences à trier en divisant successivement par deux la longueur de la séquence initiale, tandis qu'une implémentation itérative fonctionnera en multipliant par deux la longueur de séquence minimale, qui sera 1 si l'on implémente un tri par fusion pur, ou même un nombre arbitraire mais petit, si l'on décide d'utiliser un autre algorithme (comme l'alogirhtme de tri par insertion) pour les séquences ayant cette taille minimale. Néanmoins, lorsque de tels changements dans la structure des calculs sont possibles, c'est que la spécification le permet bien, et il est probable que le passage d'abord par une description fonctionnelle continuera à donner un avantage significatif au travail de formalisation.

Remerciements

Je tiens à remercier Jean-Christophe Filliâtre, dont les travaux ont permis la réalisation des expériences décrites dans cet article, Paul Zimmermann qui nous a fourni la description détaillée du programme de calcul de racine carrée efficace et Nicolas Magaud, qui a effectué une grande partie du travail de description du programme impératif.

Bibliographie

[1] Antonia Balaa and Yves Bertot. Fonctions récursives générales par itération en théorie des types. In *Journées Francophones pour les Langages Applicatifs*, January 2002.

[2] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report RT-0203, INRIA, August 1997. revised version distributed with Coq.

[3] Guy Cousineau and Michel Mauny. *The Functional Approach to Programming.* Cambridge University Press, 1998.

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

[5] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 2001. English translation of [4]. To appear.

[6] J.-C. Filliâtre and N. Magaud. Certification of sorting algorithms in the system Coq. In *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999.

[7] Xavier Leroy. Le système caml special light: modules et compilation efficace en caml. Technical Report RR-2721, INRIA, 1995.

[8] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[9] Robert Sedgewick. *Algorithms.* Addison-Wesley, 1988.

[10] Paul Zimmermann. Karatsuba square root. Research Report RR-3805, INRIA, 1999.