Vérification formelle d'extractions de racines entières

Yves Bertot

Vérification formelle d'extractions de racines entières

Yves Bertot
INRIA Sophia Antipolis
2004, route des lucioles, B.P.93
06902 Sophia Antipolis
Yves.Bertot@inria.fr

Spring 2005

Abstract

Nous décrivons la vérification formelle d'algorithmes de calcul de racines carrées, cubiques et énièmes dans un cadre fonctionnel. Nous montrons que les premiers algorithmes se décrivent bien en utilisant la représentation binaire des entiers, qui permet en outre d'assurer la terminaison de ces algorithmes. La même structure est sous-jacente à l'algorithme de racines énièmes, mais la terminaison de l'algorithme est vérifiée en utilisant des outils plus complexes. Le travail de vérification formelle a été effectué en utilisant le système Coq.

Mots-clefs Vérification formelle de logiciels, Calcul des constructions, arithmétique des ordinateurs, Coq.

1 Introduction

Parce qu'elle est très proche de la structure des polynômes, la représentation des nombres par position dans une base donnée est bien adaptée pour un grand nombre d'opérations, comme la multiplication, l'addition ou la division. Parmi les opérations fréquemment utilisées dans le calcul scientifique, l'extraction de racines, en particulier l'extraction de racines carrées, n'échappe pas à la règle.

Nous avons déjà étudié les algorithmes d'extraction de racines carrées pour les grands nombres en base arbitraire dans un travail précédent [BER 02]. Cette étude poussait le travail jusqu'à la description d'un algorithme impératif et incluait en particulier la vérification formelle d'une optimisation fine dans l'utilisation de l'espace mémoire. Dans ce travail précédent, l'approche est de type *diviser pour régner* au sens où la moitié des chiffres du résultat est déterminée en une itération (sachant que l'autre moitié est traitée par un appel récursif). Le nombre d'appels récursifs de l'algorithme est proportionnel au logarithme du nombre de chiffres utilisés pour représenter la donnée initiale.

Le travail présenté ici reprend l'algorithme d'extraction de racine carrée dans une forme plus basique, où un seul bit est déterminé à chaque itération. Il est assez facile d'étendre cet algorithme pour obtenir un algorithme d'extraction de racines cubiques, puis un algorithme d'extraction de racines énièmes qui peut éventuellement fonctionner pour de très grand nombres, avec un nombre d'appels récursifs cette fois-ci proportionnel au nombre de chiffres utilisés pour représenter la donnée initiale.

Du point de vue de la vérification formelle en théorie des types, les algorithmes de racines carrées et cubiques et l'algorithme de racines énièmes ne relèvent pas de la même catégorie. Les deux premiers se décrivent comme des algorithmes récursifs *structurels* pour lesquels le raisonnement est aisé, car il suffit d'effectuer des raisonnements par récurrence sur la structure des données. De tels raisonnements sont particulièrement bien supportés dans les systèmes de démonstration assistée par ordinateur en général. Une légère difficulté apparaît dans la mesure où les appels récursifs ne se font pas en suivant directement la structure des données, mais nous montrons comment reprendre une technique déjà présentée dans [BER 04] pour obtenir les outils de preuve adaptés. En suivant cette technique, nous arrivons à démontrer la correction de nos algorithmes de racines carrées et de racines cubiques en suivant exactement le même plan de preuve.

Pour les racines énièmes, le problème se pose différemment, car l'argument de l'appel récursif est le résultat d'une division. La bonne définition de l'algorithme, et en particulier l'assurance qu'il termine toujours, ne peut plus être reconnue par simple observation de la structure, mais doit reposer sur des propriétés de décroissance dérivées des propriétés de la fonction de division. Le problème est plus complexe car la définition de l'algorithme comme une fonction récursive doit alors combiner le contenu calculatoire de l'algorithme et la preuve que l'algorithme termine. Cette nécessité de combiner des concepts habituellement considérés séparément rend souvent la description de ce type de fonction récursive difficile. Dans cet article, nous comparons un outil que nous avons mis au point dans des travaux précédents et une fonction fournie dans le système Coq dont l'objectif est de réduire le niveau de compétence requis du programmeur pour la description des fonctions récursives et la preuve de leur propriétés.

Nous ne décrivons qu'une partie de la formalisation complète, en précisant surtout la façon dont les différentes fonctions sont décrites et les théorèmes qui ont été démontrés. La structure des démonstrations est seulement esquissée, mais les démonstrations sont disponibles sur internet[1], de sorte que cet article peut aussi être utilisé comme exemple didactique pour la description de fonctions récursives générales dans le calcul des constructions inductives. En particulier, nous décrivons comment utiliser deux outils disponibles, l'un est une commande `Recursive Definition` et l'autre est une fonction prédéfinie de Coq, nommée `Fix`.

[1] `ftp://ftp-sop.inria.r/lemme/Yves.Bertot/roots.tar.gz`

2 Racine carrée entière

2.1 Description informelle de l'algorithme

Étant donnée une entrée N, on cherche les nombres s et r tels que $N = s^2 + r$ et $s^2 \leq N < (s+1)^2$. L'idée de l'algorithme est de diviser l'entrée N par 100 (le nombre qui s'écrit ainsi dans la base en cours, c'est-à-dire cent, si l'on est en base dix, et quatre si l'on est en base deux), puis de calculer une première approximation de la racine carrée en calculant la racine du quotient. La racine trouvée, multipliée par 10 (c'est-à-dire, dix ou deux, suivant la base en cours) est une approximation par défaut du résultat, il ne reste plus qu'à calculer le chiffre de poids faible.

L'algorithme que nous étudions fait partie du folklore enseigné dans le secondaire il y a encore quelques dizaines d'années. Bien sûr, notre culture nous amène à travailler en base dix, ce qui constitue une différence notable avec l'algorithme que nous décrirons ici, mais le principe reste le même. La première étape consiste donc à grouper les chiffres du nombre considéré deux par deux, en commençant par les chiffres de la droite (donc en commençant par les chiffres de poids faible). Il faut ensuite calculer la racine carrée du nombre composé du groupe le plus à gauche, qui est constitué de un ou deux chiffres, en conservant d'une part la partie entière de cette racine carrée et d'autre part le reste. On procède ensuite répétitivement de la façon suivante: On accole à droite du reste (du côté des chiffres des poids faibles) un groupe de deux chiffres venant de la donnée initiale, on multiplie par deux la valeur courante de la racine carrée, et l'on cherche le chiffre le plus grand tel que ce chiffre accolé au double de la racine carrée, le tout multiplié par le même chiffre soit plus petit que le reste. La nouvelle racine carrée est obtenue en accolant ce chiffre à l'ancienne racine carrée. Le nouveau reste est la différence obtenue au moment de la comparaison. On peut alors recommencer, jusqu'à ce que tous les chiffres de la donnée initiale soient épuisés. La figure que l'on dessine sur la feuille ressemble grossièrement à la figure que l'on dessine pour une division euclidienne.

Voici un exemple. Nous voulons calculer la racine carrée du nombre 45678. Le groupement des chiffres produit les groupes 78, 56, et 4. Nous commençons par calculer la racine carrée de 4 (c'est 2 et le reste est 0). Nous amenons ensuite le groupe de deux chiffres 56 à côté de 0 (nous obtenons ainsi le nombre 56), et par ailleurs nous calculons le double de 2, c'est 4, et nous cherchons le plus grand chiffre c tel que $4c \times c$ soit inférieur à 56. Nous trouvons le chiffre 1 (car $41 \leq 56 < 42 \times 2$). La nouvelle racine carrée est 21, le nouveau reste est 15. A la deuxième étape récursive, nous accolons 78 à 15 pour obtenir le nombre 1578, et nous cherchons le chiffre c le plus grand tel que $42c \times c$ soit inférieur à 1578. Nous trouvons le chiffre 3 et le reste est 309. Le lecteur suspicieux pourra vérifier les égalités $21^2 + 15 = 456$ et $213^2 + 309 = 45678$.

Si l'on cherche à décomposer cet algorithme de façon récursive, il est judicieux de le faire de façon abstraite vis-à-vis de la base utilisée pour écrire le nombre. Notons β cette base. La première étape de groupage des chiffres deux par deux correspond en fait à des divisions successives par le carré de la

base. À chaque étape récursive, on considère donc que le nombre étudié N est supérieur à β^2, que l'on a décomposé ce nombre en deux fragments N' et N'' tels que $N = N'\beta^2 + N''$, que N'' est inférieur à β^2 et que l'on a déjà calculé la racine carrée de N' pour obtenir deux nombres s' (la racine de N') et r' (le reste) tels que $N' = s'^2 + r'$ et $(s'+1)^2 > N'$. Lorsque nous prenons deux chiffres de la donnée initiale pour les accoler à droite du reste r', nous fabriquons le nombre $r' \times \beta^2 + N''$. Le chiffre cherché c est le plus grand tel que $(2 \times s' \times \beta + c) \times c$ est inférieur à $r' \times \beta^2 + N''$ et on obtient un nouveau reste $r'' = (r' \times \beta^2 + N'') - (2 \times s' \times \beta + c) \times c$. La combinaison de ces équations donne bien $N = (s' \times \beta + c)^2 + r''$ et la condition sur c d'être le plus grand assure également que $(s' \times \beta + c + 1)^2 > N$.

Evidemment, cet algorithme fonctionne de façon similaire dans n'importe quelle base, mais en base deux les choix possibles pour c sont plus réduits et il n'est pas nécessaire de faire de multiplication supplémentaire. il suffit de vérifier si $4 \times s' + 1$ est plus grand ou plus petit que le nombre obtenu en accolant les deux derniers bits au reste de l'opération précédente.

2.2 Description en théorie des types

Nous n'avons pas formalisé l'algorithme général, mais seulement l'algorithme pour la base deux. Notre formalisation a été effectuée dans le calcul des constructions inductives à l'aide du système Coq [Coq04]. Cette théorie permet de décrire des algorithmes fonctionnels et d'en démontrer les propriétés. Les calculs récursifs sont autorisés dans cette théorie, mais des contraintes sont imposées pour assurer que les calculs terminent. Ces contraintes reposent sur la notion de type inductif. Intuitivement, chaque élément d'un type inductif est représentable par un arbre et les fonctions récursives ne sont autorisées à effectuer des appels récursifs que sur des sous-arbres de l'argument initial. De même que pour les langages de programmation fonctionnels typés de la famille de ML et Haskell [WEI 99, CHA 00, THO 96], les analyses de données peuvent être effectuées par filtrage, ce qui permet d'écrire une fonction en donnant les différents cas de son comportement.

Le type inductif `positive` est utilisé pour représenter les nombres entiers positifs non nuls codés de façon binaire. Il ne s'agit pas de la représentation binaire usuelle, car on représente un nombre entier par la liste des bits qui le composent en commençant par le bit de poids faible et sans zéro superflu. L'avantage de cette représentation est qu'elle est unique. Ce type inductif repose sur la constatation que tout nombre entier positif non nul est soit le nombre 1, soit le double d'un nombre entier positif non nul, soit le double d'un nombre entier positif non nul plus 1. Les deux derniers cas montrent bien que le type est récursif, puisqu'il faut déjà disposer d'un nombre positif pour en obtenir le double ou le double plus 1.

Dans le système Coq chacun de ces cas est décrit comme un constructeur du type `positive`, le cas 1 est appelé `xH`, le cas "double" est appelé `xO` et le cas "double plus 1" est appelé `xI`. Le système Coq fournit également un type inductif `Z` pour représenter tous les entiers relatifs et la fonction `Zpos`

(qui est un constructeur de `Z`) établit l'injection naturelle des entiers de type `positive` dans le type `Z`. Le système Coq fournit donc au moins deux types de données pour représenter les nombres entiers. Nous utiliserons ces deux types de données, en considérant par défaut qu'une expression numérique est dans le type `Z` et nous marquerons les expressions appartenant au type `positive` à l'aide de la marque `%positive` lorsque les ambiguités sont trop fortes pour la communication homme-machine.

Lorsqu'un nombre est plus grand que 4, il y a quatre possibilités différentes pour les deux bits de poids faible et ceci se traduit en quatre schémas obtenus en combinant les constructeurs `xO` et `xI`. Par exemple, le nombre $4 \times p + 2$ sera écrit `xO(xI p)`. Du point de vue de la récursion, `p` apparaît bien comme un sous-terme de `xO(xI p)`. Une notation de la forme

```
if Z_le_gt_dec 4 (Zpos x) then
  let (s', r') := sqrt (div4 x) in sqrt_aux s' r' (mod4 x)
else ...
```

permettrait de traiter tous les cas supérieurs à 4 en une ligne, mais cette approche n'est pas acceptée car elle ne montre pas explicitement que `div4 x` est un sous-terme de `x`.

Pour rendre explicite le fait que les appels récursifs sont bien effectués sur des sous-termes, nous sommes obligés de répéter dans `sqrt` quatre instances du code effectué dans les appels récursifs, une pour chaque valeur possible des deux bits de poids faible. Il convient de noter que `1%positive` est une notation alternative pour `xH` et `2%positive` est une alternative pour `xO xH`.

```
Definition mod4 (x:positive) : Z :=
  match x with
    1%positive => 1 | 2%positive => 2 | 3%positive => 3
    | xO(xO p) => 0
    | xO(xI p) => 2
    | xI(xO p) => 1
    | xI(xI p) => 3
  end.

Definition sqrt_aux (s' r' n'':Z) : Z*Z :=
  if Z_le_gt_dec (4*s'+1) (4*r'+n'') then
     (2*s'+1, (4*r'+n'')-(4*s'+1))
  else
     (2*s', 4*r'+n'').

Fixpoint sqrt (x:positive) : Z*Z :=
  match x with
    1 => (1,0)
  | 2 => (1,1)
  | 3 => (1,2)
  | xO(xO p) => let (s',r') := sqrt p in sqrt_aux s' r' (mod4 x)
```

```
  | xO(xI p) => let (s',r') := sqrt p in sqrt_aux s' r' (mod4 x)
  | xI(xO p) => let (s',r') := sqrt p in sqrt_aux s' r' (mod4 x)
  | xI(xI p) => let (s',r') := sqrt p in sqrt_aux s' r' (mod4 x)
  end.
```

Nous utilisons la fonction `Z_le_gt_dec` fournie dans les bibliothèques de Coq. Elle retourne une valeur positive lorsque son premier argument est inférieur au second.

On peut vérifier par quelques tests que les valeurs retournées par `sqrt` sont satisfaisantes, mais nous voulons maintenant obtenir une preuve formelle qui assure ce résultat pour toutes les valeurs possibles de l'argument initial.

2.3 Techniques de démonstration

Pour chaque définition de type inductif le système Coq fournit un principe de récurrence adapté pour le raisonnement sur ce type [PAU 93a]. Intuitivement, ce principe de récurrence indique que toute propriété qui est conservée par les constructeurs du type est vraie pour tous les éléments du type. Pour chaque constructeur du type inductif qui présente de la récursion, ce principe de récurrence permet de raisonner avec l'hypothèse que la propriété est déjà vérifiée pour les sous-termes, cette hypothèse est communément appelée une hypothèse de récurrence.

Pour le type inductif `positive`, le principe de récurrence a la forme suivante:

```
positive_ind :
 ∀ P : positive → Prop,
  (∀ p, P p → P (xI p)) →
  (∀ p, P p → P (xO p)) →
  P 1%positive →
  ∀ x, P x
```

Ce principe fait apparaître trois cas correspondant aux trois formes possibles d'un nombre positif non nul. Il est bien adapté pour raisonner sur les fonctions qui effectuent des appels récursifs sur les sous-termes directs de l'argument, mais ce n'est pas le cas pour nous, puisque les appels récursifs se font sur des sous-termes de deuxième rang.

Un principe de récurrence mieux adapté devrait avoir sept cas, comme ceux qui apparaissent dans la fonction `sqrt`. Ce principe aurait la forme suivante:

```
sqrt_ind :
 ∀ P : positive → Prop,
 P 1%positive → P 2%positive → P 3%positive →
 (∀ p, P p → P (xO (xO p))) →
 (∀ p, P p → P (xO (xI p))) →
 (∀ p, P p → P (xI (xO p))) →
 (∀ p, P p → P (xI (xI p))) →
 ∀ x, P x
```

Ce principe se démontre assez aisément dans Coq en utilisant le principe de récurrence `positive_ind` pour la propriété suivante (suivant une technique déjà présentée dans [BER 04]):

```
P x ∧ P (xO x) ∧ P (xI x)
```

Il est d'ailleurs possible de systématiser la construction de principes de récurrences adaptés pour le raisonnement sur les fonctions récursives, comme cela est décrit dans [BAR 02].

Le point le plus technique de la vérification formelle est de démontrer que les calculs effectués à l'intérieur de la fonction `sqrt_aux` sont corrects, sous l'hypothèse que les calculs effectués dans les appels récursifs sont déjà corrects. Ces démonstrations se font généralement bien, mais il faut faire attention parce que l'existence de carrés de variables dans les formules rend délicate l'utilisation de la procédure principale de raisonnement sur les inégalités, qui ne traite que l'arithmétique de Presburger, où les variables ne peuvent être multipliées que par des constantes entières. L'énoncé de ce lemme a la forme suivante:

```
Lemma sqrt_aux_correct :
  ∀ x, ge4 x →
  ∀ s' r', s'*s'+r' = Zpos (div4 x) ∧
       Zpos (div4 x) < (s'+1)*(s'+1) →
  ∀ s r, (s,r) = sqrt_aux s' r' (mod4 x) →
  s*s+r= Zpos x ∧ Zpos x < (s+1)*(s+1).
```

Le prédicat `ge4` exprime simplement que le nombre `x` est plus grand que 4. Nous utilisons une fonction `div4` pour décrire la division par 4. Cette fonction est totale et retourne une valeur inhabituelle pour les arguments inférieurs à 4, parce qu'il n'y a pas de donnée dans le type `positive` pour représenter le nombre zéro. Ce n'est pas grave ici puisque le lemme ne décrit que le cas où `x` est plus grand que 4.

Avec le lemme `sqrt_aux_correct` et le principe de récurrence adapté, on arrive aisément au théorème suivant:

```
Theorem sqrt_correct :
  ∀ x s r, (s,r)=sqrt x →
    s*s+r=Zpos x ∧ Zpos x < (s+1)*(s+1).
```

Par souci de complétude, notons que l'auteur a également introduit dans la bibliothèque standard de Coq une autre description de la fonction de racine carrée qui évite la construction d'un principe de récurrence spécifique, mais repose de façon plus complexe sur la notion de type dépendant (voir [BER 04], chap. 9).

3 Racine cubique

Pour calculer la racine cubique, le même procédé s'applique, on détermine encore un chiffre à chaque itération, mais il s'agit maintenant de grouper les chiffres de

la donnée par paquets de trois. La formule utilisée pour déterminer le nouveau chiffre est plus complexe, puisqu'elle fait aussi intervenir le carré de la racine cubique déjà calculée.

On procède donc de la manière suivante à chaque itération: on décompose le nombre N de telle manière que $N = N' \times \beta^3 + N''$ et $N'' < \beta^3$. Si v' et r' sont la racine cubique et le reste pour N', on accole le nombre N'' au reste r' (pour obtenir le nombre $r'\beta^3 + N''$), puis l'on cherche le plus grand chiffre c tel que

$$3 \times v'^2 \times \beta^2 \times c + 3 \times v' \times \beta \times c^2 + c^3$$

est plus petit que cette valeur $r'\beta^3 + N''$. Lorsque la base est 2, le calcul est plus simple, puisqu'il s'agit uniquement de comparer la valeur $12 \times v'^2 + 6 \times v' + 1$ avec $8 \times r' + N''$.

Le nombre de cas à considérer dans la fonction récursive croît encore. Nous avons maintenant 7 cas de base (sans appels récursifs) et 8 cas récursifs. La fonction prend la forme suivante:

```
Definition cubic_aux (v' r':Z)(x:positive) : Z * Z :=
  let n'' := mod8 x in
  if Z_le_gt_dec (12*(v'*v')+6*v'+1)(8*r'+n'') then
    (2*v'+1, (8*r'+n'')-(12*(v'*v')+6*v'+1))
  else
    (2*v', 8*r'+n'').

Fixpoint cubic (x:positive) : Z*Z :=
  match x with
    1%positive => (1,0)
  | 2%positive => (1,1)
    3%positive => (1,2)
  | 4%positive => (1,3)
    5%positive => (1,4)
  | 6%positive => (1,5)
    7%positive => (1,6)
  | xO(xO(xO p)) => let (v,r) := cubic p in cubic_aux v r x
  | xO(xO(xI p)) => let (v,r) := cubic p in cubic_aux v r x
  | xO(xI(xO p)) => let (v,r) := cubic p in cubic_aux v r x
  | xO(xI(xI p)) => let (v,r) := cubic p in cubic_aux v r x
  | xI(xO(xO p)) => let (v,r) := cubic p in cubic_aux v r x
  | xI(xO(xI p)) => let (v,r) := cubic p in cubic_aux v r x
  | xI(xI(xO p)) => let (v,r) := cubic p in cubic_aux v r x
  | xI(xI(xI p)) => let (v,r) := cubic p in cubic_aux v r x
  end.
```

La fonction auxiliaire `cubic_aux` est similaire à la fonction `sqrt_aux` que nous avons décrite à la section précédente, tout en reposant maintenant sur deux fonctions `div8` et `mod8` pour le quotient et le reste dans la division par 8. La procédure de preuve pour prouver la correction de cette fonction est la même

que pour `sqrt_aux` et n'est paramétrée que par trois éléments qui varient d'une preuve à l'autre: une fonction `square` est remplacée par une fonction `cube`, la référence à la fonction `sqrt_aux` est remplacée par la référence à la fonction `cubic_aux`. Ceci est possible parce que nous utilisons les tactiques automatiques `ring` et `omega` pour traiter tous les énoncés arithmétiques automatiquement. Malgré ces aspects automatiques, les démonstrations communes sont effectuées par une procédure d'une vingtaine de lignes.

L'une des preuves doit couvrir 7 cas tandis que l'autre doit en couvrir 15. En fait, les cas sont à classer en deux grandes familles: soit il s'agit d'un cas de base et alors la preuve se fait rapidement par calcul et vérification (car les cas de base sont en nombre fini), soit il s'agit d'un cas récursif et les théorèmes auxiliaires établis pour les fonctions `sqrt_aux` et `cubic_aux` s'appliquent directement. Le développement formel pour le calcul de racine carrée et le calcul de racine cubique tient dans un fichier de 250 lignes.

4 Racine énième

Même si nous avons réussi à décrire la racine carrée et la racine cubique en suivant une même structure, il est évident que cette structure ne peut s'utiliser uniformément pour les racines d'ordre plus élevé. En effet, le nombre de cas qui apparaît dans chaque fonction est $2^n - 1$ si l'on veut extraire la racine d'ordre n. Cette structure est encore moins adaptée si l'on veut disposer d'une fonction unique capable de traiter les racines d'ordre n pour tout n. Nous allons devoir mettre en œuvre une technique plus élaborée de définition de fonctions récursives pour décrire cette fonction.

4.1 Description informelle de l'algorithme

Si nous voulons calculer récursivement la racine énième d'un nombre x positif pour le rang n nous pouvons procéder de la façon suivante:

1. si x est 0, alors la racine est 0 et le reste est 0,
2. si x est non nul et inférieur à 2^n, alors la racine est 1, et le reste est $x - 1$,
3. Si x est supérieur à 2^n, on divise x par 2^n pour obtenir un quotient x' et un reste x'',
4. on calcule la racine v' et le reste r' pour x',
5. on calcule les valeurs $r'' = r' \times 2^n + x''$ et $d = (2 \times v' + 1)^n - (2 \times v')^n$,
6. on compare la valeur de r'' avec d:
   - si r'' est plus grand, alors le résultat est $2 \times v' + 1$ et le reste est $r'' - d$,
   - sinon le résultat est $2 \times v'$ et le reste est r''.

Par souci d'efficacité, il est souhaitable de calculer une fois pour toute la valeur 2^n et le polynôme $P_n = y \mapsto (2 \times y + 1)^n - (2 \times y)^n$. En particulier, si nous calculons formellement ce polynôme comme une somme de monômes, il ne contient pas de monômes de degré n.

Nous ne pouvons plus utiliser directement la structure du type inductif `positive` pour décrire l'algorithme, parce que le nombre de cas qu'il faut faire apparaître pour respecter les contraintes de la récurrence structurelle varie avec le degré de la racine n. Nous avons préféré décrire notre algorithme directement comme un algorithme travaillant sur les nombres entiers de type `Z`, même s'il ne retourne pas de valeur significative lorsque le nombre à traiter est négatif et lorsque le degré de la racine à calculer est négatif ou nul. Notre algorithme est probablement mieux adapté pour travailler sur des nombres dont la représentation interne est la base 2, parce que l'opération de division par 2^n y est alors mise en œuvre de façon efficace, mais il s'adapte sans difficulté à d'autres bases.

4.2 L'outil `Recursive Definition`

La fonction de racine énième que nous considérons n'est pas une fonction récursive structurelle comme celles que nous avons vues dans les sections précédentes. Dans la description informelle de l'algorithme, nous voyons que l'appel récursif se fait sur la valeur x', le quotient de la division par 2^n. Cette valeur est obtenue par un appel à une fonction plutôt que par une construction de filtrage et x' n'apparaît donc pas clairement comme un sous-terme de la structure de x. Nous dirons que de telles fonctions sont des fonctions *récursives générales*.

Le système Coq fournit quelques outils pour la programmation de fonctions récursives générales, le plus commun est la programmation par récursion *bien fondée*. Le principe est toujours d'assurer qu'aucune fonction récursive n'entrera dans un calcul infini par une succession interminable d'appels récursifs. Au lieu d'imposer des contraintes sur la forme des fonctions définies, comme c'est le cas pour la récursion structurelle, on utilise un raisonnement logique basé sur l'utilisation de relations bien fondées.

Les relations bien fondées sont des relations qui ne présentent pas de suites infinies décroissantes. Lorsque l'on définit une fonction récursive, on peut imposer que les arguments des appels récursifs dans la fonction soient des prédécesseurs de l'argument initial pour une relation bien fondée fixée. Comme la relation n'admet pas de suites infinies décroissantes, il ne pourra pas y avoir de suite infinie d'appels récursifs et les calculs infinis seront évités.

Plusieurs méthodes de définitions sont fournies pour définir des fonctions récursives structurées. Nous avons utilisé l'outil `Recursive Definition` que nous avons décrit dans [BAL 02] et qui est fourni sur le site internet du système Coq parmi les contributions des utilisateurs.

Dans le prototype actuellement fourni, seules les fonctions à un seul argument sont traitées, pour les autres cas la méthode mise en œuvre dans l'outil peut être appliquée manuellement en suivant le schéma décrit dans [BAL 02, BER 04]. Cet outil permet de définir une fonction récursive en donnant son

type, la relation bien fondée qui contrôle l'appel récursif, la preuve que cette relation est bien fondée, les théorèmes qui montrent que les arguments des appels récursifs sont bien des prédécesseurs de l'argument initial (dans la suite ces théorèmes seront appelés *théorèmes de décroissance*) et l'équation récursive qui décrit le contenu algorithmique de la fonction.

Un avantage de l'outil est que l'équation récursive est très similaire au programme que l'on écrirait dans un langage de programmation fonctionnel usuel en s'imposant seulement la contrainte d'écrire un programme purement fonctionnel.

Nous l'avons dit, le prototype ne considère pour l'instant que les fonctions récursives à un argument. Ceci n'est pas une contrainte importante, puisque plusieurs arguments peuvent être regroupés dans un seul multiplet. Ici, nous voulons éviter que la fonction récursive re-calcule le polynôme P_n et la valeur 2^n à chaque étape et nous voulons donc passer ces valeurs en argument de la fonction. Nous prévoyons donc que l'argument de la fonction soit un triplet:

- la première composante est le nombre dont on veut extraire la racine énième,
- la seconde composante est la liste des coefficients du polynôme P_n,
- le troisième argument est le nombre 2^n.

Nous définissons la structure de données correspondante de la manière suivante:

```
Record data : Set :=
 mk_data {d_p1:Z; d_p2:list Z; d_p3}.
```

Cette définition indique que l'on pourra construire un triplet à l'aide de `mk_data` et que l'on peut récupérer les différentes composantes d'un triplet à l'aide des fonctions de projection `d_p1`, etc.

La clef de la programmation bien fondée est de fournir une relation bien fondée. Il existe quelques relations bien fondées de base et plusieurs outils pour en fabriquer de nouvelles. Par exemple, les bibliothèques de Coq fournissent une relation ternaire `Zwf` telle que "`Zwf` k v_1 v_2" est satisfait si et seulement si $k \leq v_2 \wedge v_1 < v_2$. Un théorème est également fourni pour exprimer que pour tout k fixé la relation binaire "`Zwf` k" est une relation bien fondée. Parmi les outils fournis pour construire de nouvelles relations bien fondées, un outil important est le théorème qui exprime que l'image inverse d'une relation bien fondée par une fonction arbitraire est toujours bien fondée. Nous utilisons ce théorème pour définir la relation bien fondée qui contrôle notre fonction récursive. En effet, nous ne comparons que les premières composantes de deux triplets et vérifions que ces composantes sont reliées par la relation "`Zwf 0`". Ainsi, la relation bien fondée que nous choisissons est l'image inverse de la relation "`Zwf 0`" par la première projection des triplets (fonction `d_p1`).

L'outil `Recursive Definition` fournit une aide à l'utilisateur en indiquant quel but devra être démontré pour assurer que l'appel récursif respecte la relation bien fondée. Il suffit d'appeler cet outil en fournissant tous les éléments de

la définition, mais un mauvais théorème de décroissance (ici nous fournissons le théorème `I` qui est une preuve de `True`):

```
Recursive Definition nroot_aux (data → Z*Z)
  (fun x y => Zwf 0 (d_p1 x) (d_p1 y))
  (wf_inverse_image data Z (Zwf 0) d_p1
     (Zwf_well_founded 0))
  I
  (∀ t,
    nroot_aux t =
    let (p, l, twopn) := t in
    (let (p', p'') := Zdiv_eucl p twopn in
     if Z_le_gt_dec p' 0 then
         if Z_eq_dec p 0 then (0, 0) else (1, p - 1)
       else let (v', r') := nroot_aux (mk_data p' l twopn) in
            let aux_v := eval_poly v' l in
              if Z_le_gt_dec aux_v (twopn * r' + p'')
                then (2 * v' + 1, (twopn * r' + p'') - aux_v)
                else (2 * v', twopn * r' + p'')))).
```

Le texte commençant par la ligne "`nroot_aux t =`" décrit la fonction récursive que nous voulons définir. Les lecteurs habitués de la programmation récursive dans les langages de programmation fonctionnels reconnaîtront que ce texte est similaire à une définition de fonction usuelle.

Notre définition utilise quelques fonctions auxiliaires. La fonction `Zdiv_eucl` effectue la division de son premier argument par le deuxième, lorsque ce dernier est un entier strictement positif et retourne une valeur arbitraire sinon. Les fonctions `Z_le_gt_dec` et `Z_eq_dec` effectuent des comparaisons, la première répond une valeur positive si son premier argument est inférieur ou égal au second, la seconde répond une valeur positive si son premier argument est égal au second. La fonction `eval_poly` prend en premier argument un nombre entier et en second une liste d'entiers représentant les coefficients d'un polynôme et retourne la valeur de ce polynôme pour ce nombre entier.

L'outil répond en montrant le but de décroissance que l'on doit résoudre :

```
t : data
hrec :  ...
p : Z
l : list Z
twopn : Z
teq : t = (mk_data p l twopn)
p' : Z
p'' : Z
teq0 : Zdiv_eucl p twopn = (p', p'')
anonymous : p' > 0
teq1 : Z_le_gt_dec p' 0 = right (p' ≤ 0) anonymous
============================
```

```
Zwf 0 (d_p1 (mk_data p' l twopn)) (d_p1 (mk_data p l twopn))
```

Ce but est composé de deux parties. La première partie énumère l'ensemble des hypothèses qui peuvent être faites sur les données du problème. Par exemple, on peut supposer `p' > 0` (cette hypothèse est nommée `anonymous`) et l'on peut également supposer que `(p',p'')` est le couple du quotient et du reste de la division de `p` par `twopn` (cette hypothèse est nommée `teq1`). Dans ce but, `twopn` est un nom de variable arbitraire et l'on s'aperçoit rapidement que nous ne disposons pas d'assez d'information: si `twopn` est le nombre 1, alors `p'=p` et nous n'arriverons pas à démontrer le résultat attendu.

Il est donc nécessaire de préciser des informations sur les arguments de la fonction pour arriver à la définir. Nous voulons utiliser cette fonction avec une valeur entière `twopn` qui est une puissance non nulle de 2. Nous sommes donc sûrs que cette valeur est strictement supérieure à 1. Nous allons donc construire une nouvelle forme de multiplet, un multiplet qui contient également une preuve que la troisième composante est supérieure à 1. Il s'agit d'un multiplet dans lequel le quatrième champ dépend du troisième, puisque c'est une preuve que ce champ est supérieur à 1. Ici, la fonction pour construire un nouveau multiplet est appelée `nadc` et les projections `nad_p1`, etc.

```
Record nroot_arg_data : Set :=
  nadc {nad_p1:Z; nad_p2:list Z; nad_p3:Z; nad_p4 : 1 < nad_p3}.
```

Nous devons maintenant adapter notre fonction récursive à la nouvelle forme de l'argument et la commande prend la forme suivante:

```
Recursive Definition nroot_aux (nroot_arg_data → Z*Z)
  (fun (x y:nroot_arg_data) => Zwf 0 (nad_p1 x) (nad_p1 y))
  (wf_inverse_image nroot_arg_data Z (Zwf 0) nad_p1
     (Zwf_well_founded 0))
  I
  (∀ t,
    nroot_aux t =
    let (p, l, twopn, twopn_pos) := t in
    (let (p', p'') := Zdiv_eucl p twopn in
     if Z_le_gt_dec p' 0 then
         if Z_eq_dec p 0 then (0, 0) else (1, p - 1)
       else let (v', r') :=
                 nroot_aux (nadc p' l twopn twopn_pos) in
            let aux_v := eval_poly v' l in
              if Z_le_gt_dec aux_v (twopn * r' + p'')
                then (2 * v' + 1, (twopn * r' + p'') - aux_v)
                else (2 * v', twopn * r' + p'')))).
```

L'outil `Recursive Definition` répond en indiquant qu'il faut démontrer le but suivant:

```
  t : nroot_arg_data
```

```
hrec : ...
p : Z
l : list Z
twopn : Z
twopn_pos : 1 < twopn
teq : t = nadc p l twopn twopn_pos
p' : Z
p'' : Z
teq0 : Zdiv_eucl p twopn = (p', p'')
anonymous : p' > 0
teq1 : Z_le_gt_dec p' 0 = right (p' ≤ 0) anonymous
============================
 Zwf 0 (nad_p1 (nadc p' l twopn twopn_pos))
   (nad_p1 (nadc p l twopn twopn_pos))
```

Nous sommes maintenant dans une meilleure situation, car nous disposons de l'hypothèse supplémentaire `twopn_pos` qui exprime la contrainte nécessaire sur `twopn`.

Dans notre développement nous avons démontré le théorème de décroissance suivant:

```
Theorem dec_thm :
  ∀ (p p' p'' twopn:Z)(l:list Z)(twopn_pos : 1 < twopn),
  Zdiv_eucl p twopn = (p', p'') → p' > 0 →
  Zwf 0 (nad_p1 (nadc p' l twopn twopn_pos))
        (nad_p1 (nadc p l twopn twopn_pos)).
```

Toutes les prémisses de ce théorème sont satisfaites dans le but produit par notre outil. Prouver ce théorème, c'est donc démontrer que l'équation récursive décrit bien une fonction récursive qui termine toujours.

Lorsque l'on donne le théorème `dec_thm` comme argument à la place de `I`, l'outil `Recursive Definition` fonctionne normalement et produit plusieurs objets, en particulier une fonction `nroot_aux` et un théorème `nroot_aux_equation` dont l'énoncé est exactement l'équation récursive. Ce théorème est un élément clef pour démontrer que la fonction `nroot_aux` calcule effectivement la racine énième d'un nombre.

4.3 Théorèmes compagnons

Le type de la fonction `nroot_aux` ne permet pas d'exprimer que la valeur retournée est bien la racine énième de l'argument. Nous devons fournir un théorème supplémentaire. Bien sûr, la fonction n'effectue les calculs attendus que si les arguments supplémentaires de la fonction représentent bien le polynôme P_n et la valeur 2^n. Nous démontrons donc les théorèmes suivants :

```
Theorem nroot_aux_correct_eq:
 ∀ p n v r,
```

```
 0 ≤ p →
 nroot_aux (nadc p (poly1 n) (2^(Zpos n)) (Zpower2_gt1 n)) =
   (v, r) →
  v^(Zpos n) + r = p.

Theorem nroot_aux_correct_interval:
 ∀ p n v r,
 0 ≤ p →
 nroot_aux (nadc p (poly1 n) (2^(Zpos n)) (Zpower2_gt1 n)) =
   (v, r) →
  v^(Zpos n) ≤ p < (v + 1)^(Zpos n).
```

Nous utilisons la fonction `Zpower` fournie dans les bibliothèques de Coq avec la notation "`a ^ b`" pour "`Zpower a b`" pour décrire l'exponentiation d'un nombre. Nous utilisons aussi une fonction `poly1` qui fabrique le bon polynôme. Cette fonction repose sur des fonctions d'arithmétique des polynômes qui simplifient leur résultat au fur-et-à-mesure et que nous avons programmées pour l'occasion: la fonction `poly_pow` calcule la puissance d'un polynôme à un certain degré, la fonction `poly_mult` multiplie un polynôme par un entier, et la fonction `poly_plus` additionne deux polynômes. Enfin, la notation `a::b::...::nil` désigne la liste contenant `a`, `b`, etc, et `nil` désigne la liste vide.

```
Definition poly1 (p : positive) : list Z :=
   poly_plus
    (poly_pow (1 :: (2 :: nil)) p)
    (poly_mult1 (- 1) (poly_pow (0 :: (2 :: nil)) p)).
```

La correction de cette fonction est exprimée par le théorème suivant:

```
Theorem poly1_correct:
 ∀ v n,  eval_poly v (poly1 n) =
    (2 * v + 1) ^ Zpos n - (2 * v) ^ Zpos n.
```

Ce théorème est une conséquence directe des propriétés des opérations d'addition et de multiplication pour les polynômes.

Comme c'est toujours le cas pour les fonctions définies en programmation bien fondée, les propriétés de ces fonctions se démontrent également en utilisant un principe de récurrence bien fondée. La démonstration s'effectue donc en s'appuyant sur le théorème suivant:

```
well_founded_ind (Zwf_well_founded 0):
∀ P : Z → Prop,
       (∀ x : Z, (∀ y : Z, Zwf 0 y x → P y) → P x) →
       ∀ a : Z, P a
```

Ce théorème exprime donc que nous ferons notre démonstration en supposant que la propriété cherchée est déjà satisfaite pour les nombres plus petits que le nombre considéré mais positifs ou nuls. Si nous appliquons ce principe de

récurrence à l'argument p des théorèmes nroot_aux_correct_eq et nroot_aux_correct_interval, cela signifie en particulier que nous supposerons que la propriété est déjà satisfaite pour les appels récursifs.

Il y a quatre cas dans chaque démonstration, parce que la fonction nroot_aux effectue un test pour déterminer si un appel récursif sera nécessaire puis elle effectue un test supplémentaire dans chacun des cas pour déterminer la valeur finale. Il est donc justifié d'établir huit théorèmes auxiliaires avec les énoncés suivants:

```
Theorem nroot_0_eq : ∀ n, 0 ^(Zpos n) + 0 = 0.

Theorem nroot_lt_twopn_eq : ∀ p n, 1^(Zpos n) + (p-1) = p.

Theorem nroot_gt_twopn_fst_case_eq :
 ∀ p n p' p'' v' r',
   p = 2^(Zpos n)*p' + p'' →
   v'^(Zpos n) + r' = p' →
   (2*v'+1)^(Zpos n) +
    (r'*2^(Zpos n)+p'' -((2*v'+1)^(Zpos n)-(2*v')^(Zpos n))) = p.

Theorem nroot_gt_twopn_snd_case_eq :
 ∀ p n p' p'' v' r',
   p = p'*2^(Zpos n)+ p'' →
   v'^(Zpos n)+ r' = p' →
   (2*v')^(Zpos n)+ r'*2^(Zpos n)+p'' = p.

Theorem nroot_0_interval :
  ∀ n, 0^(Zpos n)≤ 0 < (0+1)^(Zpos n).

Theorem nroot_lt_twopn_interval :
∀ p n, 0 ≤ p → 0 <> p → 2^(Zpos n) > p →
 1^(Zpos n) ≤ p < (1+1)^(Zpos n).

Theorem nroot_gt_twopn_snd_case_interval :
 ∀ p n p' p'' v' r',
   p = 2^(Zpos n)*p' + p'' →
   0 ≤ p'' < 2^(Zpos n) →
   v'^(Zpos n) + r' = p' →
   v'^(Zpos n) ≤ p' < (v'+1)^(Zpos n) →
   (2*v'+1)^(Zpos n) - (2*v')^(Zpos n) ≤
       r' * 2 ^(Zpos n) + p'' →
   (2*v)^(Zpos n) ≤ p < (2*v'+1)^(Zpos n).
```

Dans ces théorèmes, les variables v' et r' tiennent le rôle du résultat des appels récursifs. Il faut noter que les deux derniers énoncés d'intervalles raisonnent sur des données v' et r' qui doivent vérifier l'égalité $v'^n + r' = p'$. Nous utilisons donc le résultat nroot_aux_correct_eq pour le résultat nroot_aux_correct_interval.

4.4 L'outil `Fix`

Les deux principales qualités de l'outil `Recursive Definition` sont de permettre la définition de fonctions récursives générales en prouvant l'équation récursive associée à ces fonctions et de permettre la description de la fonction récursive en minimisant le recours aux types dépendants. Néanmoins, cet outil a le défaut d'être encore à l'état de prototype. La bibliothèque standard de Coq fournit un autre outil pour construire des fonctions récursives générales, sous la forme d'une fonction à type dépendant `Fix` et d'un théorème associé `Fix_eq`.

```
Fix : ∀ (A : Set) (R : A → A → Prop),
      well_founded R →
      ∀ P : A → Set,
      ∀ F : (∀ x : A,
                  ∀ rec: ∀ y : A, R y x → P y, P x),
      ∀ x : A, P x

Fix_eq
   : ∀ (A : Set)(R : A → A → Prop)(Rwf : well_founded R)
       (P : A → Set)
       (F : ∀ x : A, (∀ y : A, R y x → P y) → P x),
     (∀ (x : A) (f g : ∀ y : A, R y x → P y),
      (∀ (y : A) (p : R y x), f y p = g y p) →
                                          F x f = F x g) →
     ∀ x : A,
     Fix Rwf P F x =
         F x (fun (y : A) (_ : R y x) => Fix Rwf P F y)
```

Dans cette section, nous allons montrer comment utiliser la fonction `Fix` et le théorème associé pour re-définir la fonction `nroot_aux`. Nous verrons que cette approche demande une plus grande maîtrise des types dépendants.

Les arguments de la fonction `Fix` sont, dans l'ordre, le type de départ de la fonction que l'on veut définir (nommé `A`), une relation binaire sur ce type (nommée `R`), un théorème démontrant que cette relation binaire est bien fondée, une fonction décrivant le type d'arrivée de la fonction (elle est nommée `P`, c'est une fonction car la fonction `Fix` peut être utilisée pour définir des fonctions à types dépendants) et une fonction (nommée `F`) qui décrit l'algorithme utilisé dans la fonction à définir. Le premier argument (nommé `x`) de la fonction `F` est l'argument de la fonction récursive que l'on veut définir, le second argument (nommé `rec`) est la fonction qui est utilisée pour représenter les appels récursifs. Cet argument a lui même un type qui force les appels récursifs à n'avoir lieu que sur les prédécesseurs de l'argument initial pour la relation `R`.

Alors que l'outil `Recursive Definition` autorisait l'utilisateur à fournir une équation récursive dans laquelle les arguments de décroissance n'apparaissaient pas explicitement, la fonction `Fix` exige que l'on fournisse une description de l'algorithme dans laquelle l'argument de décroissance est fourni à la fonction `rec` lorsqu'elle est utilisée. Nous pouvons utiliser le même théorème `dec_thm` que

dans la section précédente, mais la description de l'algorithme doit être modifiée pour faire apparaître les éléments nécessaires à l'application de ce théorème.

Le premier élément manquant est une hypothèse de la forme

```
Zdiv_eucl p twopn = (p', p'')
```

En effet notre description initiale de l'algorithme contient le fragment

```
let (p',p'') := Zdiv_eucl p twopn in  ...
```

Ceci introduit bien les variables `p'` et `p''` dans le programme, mais pas la preuve de l'égalité nécessaire. Pour obtenir cette égalité, nous utilisons une technique déjà décrite dans [BER 04] sous le terme "renforcement minimal de spécification" (en anglais *minimal specification strengthening*) pour construire une fonction qui retourne les mêmes valeurs `p` et `p'` plus une preuve de l'égalité requise:

```
Inductive div_eucl_data (p d:Z) : Set :=
 dedc : ∀ q r, Zdiv_eucl p d = (q, r) → div_eucl_data p d.

Definition Zdiv_eucl' (p d:Z) : div_eucl_data p d :=
  (match Zdiv_eucl p d as v
       return Zdiv_eucl p d = v → div_eucl_data p d with
     (q, r) => fun h => dedc p d q r h
   end (refl_equal (Zdiv_eucl p d))).
```

Le théorème `dec_thm` requiert également une preuve de `p' > 0`, mais cette preuve est fournie naturellement par le test "`Z_le_gt_dec 0 p`" si l'on se trouve dans la branche adéquate, à condition d'utiliser une construction de filtrage au lieu d'une construction `if-then-else`. La fonction que nous allons fournir à `Fix` aura donc la forme suivante:

```
Definition nroot_aux'_F (t : nroot_arg_data) :=
    match t return
      (∀ y:nroot_arg_data,
             (Zwf 0 (nad_p1 y) (nad_p1 t))→Z*Z)→Z*Z with
     nadc p l twopn twopn_pos =>
      fun nroot_aux' =>
       let (p', p'', h) := Zdiv_eucl' p twopn in
       match Z_le_gt_dec p' 0  with
         left _ => if Z_eq_dec p 0 then (0, 0) else (1, p - 1)
       | right hgt =>
         let (v', r') :=
              nroot_aux' (nadc p' l twopn twopn_pos)
               (dec_thm p p' p'' twopn l twopn_pos h hgt) in
              let aux_v := eval_poly v' l in
                if Z_le_gt_dec aux_v (r' * twopn + p'')
                  then (2 * v' + 1, (r' * twopn + p'') - aux_v)
```

```
                else (2 * v', r' * twopn + p'')
      end
    end.

Definition nroot_aux' : nroot_arg_data → Z*Z :=
 Fix (wf_inverse_image nroot_arg_data Z (Zwf 0) nad_p1
          (Zwf_well_founded 0))
   (fun _ => (Z*Z)%type) nroot_aux'_F.
```

Dans cette approche nous avons utilisé à deux reprises des constructions de filtrage dépendant, la première fois dans la fonction `Zdiv_eucl'` et la seconde fois dans la fonction `nroot_aux'_F`. Ces constructions de filtrage dépendant permettent d'identifier les expressions de type `nroot_arg_data` ou `Z*Z` et leur contenu.

La fonction `nroot_aux'` satisfait la même équation récursive que la fonction `nroot_aux`. Nous prouvons cette équation facilement à l'aide du théorème `Fix_eq`. Après réécriture à l'aide de ce théorème, deux parties apparaissent dans la preuve. Dans la première partie, il faut montrer que la fonction `nroot_aux'_F` et l'algorithme décrit dans l'équation récursive coïncident. La seule différence entre ces deux parties est l'utilisation de la fonction `Zdiv_eucl'` dans un cas et de la fonction `Zdiv_eucl` dans l'autre. Du point de vue algorithmique, la différence entre ces deux fonctions est négligeable, et il est rapide de démontrer que les similitudes entre ces deux fonctions suffisent à assurer l'égalité.

La deuxième partie consiste à démontrer l'hypothèse du théorème `Fix_eq`. Cette hypothèse est facile à démontrer car il s'agit simplement de démontrer que la même expression effectue les mêmes calculs de part et d'autre. Cette démonstration ne se fait pas en une seule étape parce que la théorie supportée dans le calcul des constructions n'est pas extensionnelle. Néanmoins, nous avons démontré dans [BAL 00] que cette démonstration se faisait systématiquement, au point d'être pratiquement automatisable (nous disposons d'un prototype).

Puisque la fonction `nroot_aux'` satisfait la même équation que `nroot_aux` on pourra démontrer les mêmes propriétés pour cette fonction.

Nous n'avons pas épuisé toutes les possibilités pour décrire notre algorithme en nous reposant sur la récursion bien fondée. Une troisième technique plus directe passe par la description d'une fonction dont le type exprime directement que le résultat est composé de deux nombres et d'une preuve que ces nombres forment bien la racine énième de l'entrée. Cette technique repose encore plus massivement sur les types dépendants et est décrite de façon détaillée dans [BER 04] (Sect. 15.2).

4.5 La fonction principale de calcul de racines énièmes

Les fonctions `nroot_aux` et `nroot_aux'` ne sont que des fonctions auxiliaires qui travaillent avec un polynôme et un diviseur arbitraires. Il est nécessaire d'appeler l'une de ces fonctions avec les bons arguments pour calculer effectivement une racine énième. En particulier, le théorème `Zpower2_gt1` est une

preuve que 2^n est toujours strictement supérieur à 1 (quand n est un nombre strictement positif)

```
Definition nroot (p n : Z) :=
   match n with
     Zpos v => nroot_aux
      (nadc p (poly1 v) (2 ^ (Zpos v))  (Zpower2_gt1 v))
    | _ => (0, 0)
   end.
```

La correction de cette fonction est assurée par le théorème suivant qui se démontre aisément à l'aide des théorèmes précédents.

```
Theorem nroot_correct:
 ∀ p n v r,
 0 ≤ p →  0 < n →
 nroot p n = (v, r) →
    v ^ n + r = p ∧ ( v ^ n ≤ p < (v + 1) ^ n ).
```

L'ensemble des développements pour la racine énième se décompose en un peu plus d'une centaine de lignes pour décrire les opérations sur les polynômes, une quarantaine de lignes pour montrer que l'algorithme termine, une quinzaine de lignes pour définir la fonction `nroot_aux` en utilisant l'outil `Recursive Definition` et environ 350 lignes pour démontrer que la fonction `nroot_aux` satisfait sa spécification.

Si l'on utilise l'outil `Fix` il faut remplacer la quinzaine de lignes de définition pour `nroot_aux` par une cinquantaine de lignes.

4.6 Extraction de programmes conventionnels

Les algorithmes fonctionnels étudiés peuvent être directement traduits vers des programmes, écrits dans les langages Ocaml, Haskell, ou Scheme. L'outil fourni pour effectuer ce travail est l'outil d'extraction intégré au système Coq [PAU 93b, LET 03]. En plus de la traduction d'un langage à l'autre, cet outil se charge d'enlever les parties de la description formelle qui ne sont reliées qu'à la vérification de propositions logiques pour ne garder que les calculs qui sont effectivement nécessaire à la construction du résultat final.

Lorsque la fonction d'extraction de racines est décrite à l'aide de l'outil `Recursive Definition`, le code engendré contient une fonction `nroot_aux` et une fonction `nroot_aux_terminate`, la première appelle directement la deuxième et retourne directement le même résultat, sans calcul supplémentaire. C'est la fonction appelée `nroot_aux_terminate` qui est récursive, et sa forme est exactement celle décrite dans l'équation récursive utilisée pour la définition formelle.

Lorsque la fonction d'extraction de racines est décrite à l'aide de `Fix`, il est nécessaire de configurer l'outil d'extraction pour que les fonctions `Fix`, `nroot_aux_F` et une fonction auxiliaire de `Fix` appelée `Fix_F` n'apparaissent pas dans le code final. Cette configuration est effectuée par la commande suivante:

```
Extraction Inline Fix nroot_aux'_F Fix_F.
```

On peut ensuite extraire le programme fonctionnel en lançant la commande suivante:

```
Extraction "nroot.ml" nroot nroot_aux nroot_aux'.
```

Ce programme peut être utilisé pour effectuer des calculs de racines sur des grands nombres, mais il faut soit s'astreindre à utiliser la représentation obtenue en traduisant les types `Z` et `positive`, soit se munir de fonctions de conversions vers d'autres formats de représentation des grands nombres entiers. Pour effectuer quelques tests, nous avons mis au point un convertisseur vers une représentation de chaînes de caractères donnant la représentation décimale (fourni avec notre développement sur internet).

5 Travaux similaires

L'auteur a également fourni la fonction de racine carrée fournie dans la bibliothèque standard du système Coq. Ce développement n'a pas été fait en utilisant les approches décrites ici, mais en construisant directement une fonction récursive fortement spécifiée comme démontré dans [BER 04] (sect. 9.4.2).

De nombreux chercheurs travaillent sur la certification de l'arithmétique des ordinateurs dans le monde, en se concentrant sur les opérations flottantes [HAR 98, DAU 01]. Une partie de ces recherches est soutenue par les fabricants de microprocesseurs [HAR 99, RUS 99]. En particulier, nous avons travaillé sur la certification d'un algorithme de calcul de racines carrées sur les grands nombres qui généralise l'algorithme utilisé dans le présent article pour obtenir une approche de type *diviser pour régner* [BER 02]. La généralisation de l'algorithme utilisé ici pourrait également mener à une implémentation efficace pour les grands nombres.

Une deuxième contribution de cet article est la description qui est faite des techniques disponibles pour modéliser des algorithmes récursifs généraux. Les deux techniques que nous avons proposées ici ont en commun de reposer sur la notion de relation bien fondée. D'autres solutions ont été proposées qui reposent sur une utilisation plus forte de types dépendants. Dubois et Donzeau-Gouge défendent le point de vue qui consiste à décrire la terminaison de toute fonction récursive par un prédicat inductif défini automatiquement pour la circonstance [DUB 98]. La même approche est défendue par Bove dans [BOV 03]. Cette approche présente l'avantage de permettre un pouvoir expressif important et en particulier de permettre la description de fonctions récursives partielles. Les travaux de Bove ont surtout été effectués dans le cadre de la théorie des types de Martin-Löf mais leur transposition dans le calcul des constructions se fait assez aisément. Par exemple, nous avons utilisé cette technique dans la description d'algorithmes calculant sur les nombres rationnels [NIQ 04]. McBride et McKinna défendent un point de vue similaire et vont plus loin en proposant une nouvelle construction de filtrage adaptée à la programmation utilisant ces

prédicats inductifs [MCB 04]. Ces approches sont puissantes et élégantes, mais nous pensons que l'approche fournie avec l'outil `Recursive Definition` apporte une contribution significative en réduisant la compétence requise dans la manipulation de types dépendants, que nous considérons un obstacle à la diffusion des techniques de formalisation logicielle en théorie des types.

D'autres systèmes de démonstration formelle fournissent des moyens de description de fonctions récursives, souvent dans le contexte de types inductifs ou en se reposant sur une théorie de relations nœtheriennes ou bien fondées [SLI 96, OWR 01, KAU 00]. Dans ces systèmes, les types dépendants ne sont pas fournis (ou pratiquement pas). Ceci réduit le pouvoir expressif mais en contrepartie, la difficulté d'apprentissage est moindre.

Remerciements

L'auteur tient à remercier Paul Zimmermann et Nicolas Magaud pour leur collaboration dans le travail sur les racines carrées et Mélanie Vaillant qui a mis en œuvre les premières expériences sur la racine énième. Enfin, l'outil `Recursive Definition` développé en collaboration avec Antonia Balaa a particulièrement facilité notre expérience et la fonction `Fix` a été introduite dans la bibliothèque standard de Coq par Christine Paulin-Mohring.

6 Conclusion

L'algorithme que nous avons décrit pour la racine énième n'utilise pas du tout la structure binaire utilisée pour la représentation des nombres, il est donc adaptable à n'importe quelle autre base et on pourra réutiliser la même étude pour la vérification formelle d'un algorithme travaillant dans une base arbitraire, en particulier les algorithmes reposant directement sur la représentation usuelle de l'arithmétique des ordinateurs. Néanmoins, il sera préférable de décrire un algorithme qui détermine au moins un chiffre à chaque itération, alors que l'algorithme décrit ici ne détermine qu'un bit à chaque itération.

Au delà des qualités intrinsèques de l'algorithme, cet article est également une présentation didactique de techniques qui permettent la définition de fonctions récursives générales dans le calcul des constructions.

References

[BAL 00] Balaa A., Bertot Y., ¡¡Fix-point Equations for Well-Founded Recursion in Type Theory¿¿, In Harrison J., Aagaard M., eds., *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, Vol. 1869 of *Lecture Notes in Computer Science*, Springer-Verlag, 2000, pp. 1–16.

[BAL 02] Balaa A., Bertot Y., ¡¡Fonctions récursives générales par itération en théorie des types¿¿, In *Journées Francophones pour les Langages Applicatifs*, Jan 2002.

[BAR 02] Barthe G., Courtieu P., ¡¡Efficient reasoning about executable specifications in Coq¿¿, In Carreño V., Muñoz C., Tahar S., eds., *Proceedings of TPHOLs'02*, Vol. 2410 of *Lecture Notes in Computer Science*, Springer-Verlag, 2002, pp. 31–46.

[BER 02] Bertot Y., Magaud N., Zimmermann P., ¡¡A proof of GMP square root¿¿, *Journal of Automated Reasoning*, Vol. 22, No. 3–4, 2002, pp. 225–252.

[BER 04] Bertot Y., Castéran P., *Interactive theorem proving and program development, Coq'art: the calculus of inductive constructions*, Texts in Theoretical Computer Science: an EATCS series, Springer-Verlag, 2004.

[BOV 03] Bove A., ¡¡General Recursion in Type Theory¿¿, In Geuvers H., Wiedijk F., eds., *Types for Proofs and Programs, International Workshop TYPES 2002, The Netherlands*, No. 2646 in Lecture Notes in Computer Science, March 2003, pp. 39–58.

[CHA 00] Chailloux E., Manoury P., Pagano B., *Développement d'applications avec Objective Caml*, O'Reilly and associates, 2000.

[Coq04] Coq development team, ¡¡The Coq Proof Assistant Reference Manual, version 8.0¿¿, 2004.

[DAU 01] Daumas M., Rideau L., Théry L., ¡¡A generic library of floating-point numbers and its application to exact computing¿¿, In Boulton R. J., Jackson P. B., eds., *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, Vol. 2152 of *Lecture Notes in Computer Science*, Springer-Verlag, Sep. 2001, pp. 169-184.

[DUB 98] Dubois C., Viguié-Donzeau-Gouge V., ¡¡A step towards the mechanization of partial functions: domains as inductive predicates¿¿, Jul. 1998, `http:/www.cs.bham.ac.uk/~mmk/cade98-partiality`.

[HAR 98] Harrison J., *Theorem Proving with the Real Numbers*, Distinguished dissertations, Springer-Verlag, London, 1998.

[HAR 99] Harrison J., ¡¡A Machine-Checked Theory of Floating Point Arithmetic¿¿, In Bertot Y., Dowek G., Hirschowitz A., Paulin C., Théry L., eds., *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, Vol. 1690 of *Lecture Notes in Computer Science*, Nice, France, 1999, Springer-Verlag, pp. 113–130.

[KAU 00] Kaufmann M., Manolios P., Moore J. S., *Computer-aided reasoning: an approach*, Kluwer Academic Publishing, 2000.

[LET 03] Letouzey P., ¡¡A New Extraction for Coq¿¿, In Geuvers H., Wiedijk F., eds., *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, Vol. 2646 of *Lecture Notes in Computer Science*, Springer-Verlag, 2003, pp. 200–219.

[MCB 04] McBride C., McKinna J., ¡¡The view from the left¿¿, *Journal of Functional Programming*, Vol. 14, No. 1, 2004, pp. 69–111.

[NIQ 04] Niqui M., Bertot Y., ¡¡QArith: Coq Formalization of Lazy Rational Arithmetic¿¿, In *Types for proofs and programs*, No. 3085 in Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 309–323.

[OWR 01] Owre S., Shankar N., Rushby J. M., Stringer-Calvert D. W. J., ¡¡PVS Language Reference, version 2.4¿¿, Computer Science Laboratory, SRI International, Menlo Park, CA, Nov. 2001.

[PAU 93a] Paulin-Mohring C., ¡¡Inductive Definitions in the System Coq - Rules and Properties¿¿, In Bezem M., Groote J.-F., eds., *Proceedings of the conference Typed Lambda Calculi and Applications*, No. 664 in Lecture Notes in Computer Science, 1993, pp. 328–345, LIP research report 92-49.

[PAU 93b] Paulin-Mohring C., Werner B., ¡¡Synthesis of ML programs in the system Coq¿¿, *Journal of Symbolic Computation*, Vol. 15, 1993, pp. 607–640.

[RUS 99] Russinoff D. M., ¡¡A Mechanically Checked Proof of IEEE Compliance of the AMD K5 Floating-Point Square Root Microcode¿¿, *Formal Methods in System Design*, Vol. 14, No. 1, 1999, pp. 75–125.

[SLI 96] Slind K., ¡¡Function definition in higher order logic¿¿, In *Theorem Proving in Higher Order Logics*, Vol. 1125 of *Lecture Notes in Computer Science*, Sprinter Verlag, Aug. 1996, pp. 381–397.

[THO 96] Thompson S., *Haskell, the craft of functional programming*, Addison-Wesley, 1996.

[WEI 99] Weis P., Leroy X., *Le Langage Caml*, Dunod, 1999.

Contents