HAL
archives-ouvertes.fr

QArith: Coq Formalisation of Lazy Rational Arithmetic

Milad Niqui, Yves Bertot

▶ **To cite this version:**

Milad Niqui, Yves Bertot. QArith: Coq Formalisation of Lazy Rational Arithmetic. [Research Report] RR-5004, INRIA. 2003, pp.19. <inria-00077041>

HAL Id: inria-00077041
https://hal.inria.fr/inria-00077041

Submitted on 29 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

QArith: Coq Formalisation of Lazy Rational Arithmetic

Milad Niqui — Yves Bertot

N° 5004

Novembre 2003

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--5004--FR+ENG

QArith: Coq Formalisation of Lazy Rational Arithmetic

Milad Niqui *, Yves Bertot

Thème 2 — Génie logiciel
et calcul symbolique
Projets Lemme

Rapport de recherche n° 5004 — Novembre 2003 — 19 pages

Abstract: In this paper we present the *Coq* formalisation of the *QArith* library which is an implementation of rational numbers as binary sequences for both lazy and strict computation. We use the representation also known as the Stern-Brocot representation for rational numbers. This formalisation uses advanced machinery of the *Coq* theorem prover and applies recent developments in formalising general recursive functions. This formalisation highlights the rôle of type theory both as a tool to verify hand-written programs and as a tool to generate verified programs.

Key-words: formal methods, type theory, calculus of constructions, exact computation

* From the University of Nijmegen, and supported by the grant nr. R 61-526 from the Netherlands Organization for Scientific Research (NWO).

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

QArith: Développement formel de l'arithmétique paresseuse sur les rationnels

Résumé : Nous présentons une formalisation en *Coq* de la bibliothèque *QArith* qui donne une construction des nombres rationnels comme des séquences binaires sur lesquelles on peut calculer de manière stricte ou paresseuse. La représentation utilisée est celle de Stern et Brocot. Le développement formel utilise des techniques avancées pour la description de fonctions récursives générales dans le système *Coq*. Ce développement souligne la puissance de *Coq* dans le développement de programmes certifiés.

Mots-clés : méthodes formelles, théorie des types, calcul des constructions, calcul exact

Contents

1 Introduction

The present work is the continuation of two earlier parallel works of the authors [3, 12] with two main goals:

1. To present a library of rational numbers for *Coq* [17] based on a canonical representation for rational numbers also known as Stern-Brocot representation[1].

2. To verify in *Coq* the correctness of a family of lazy algorithms for exact rational arithmetic.

In the present paper we do not detail the lazy algorithms that are described in [12]. For the complete formal development, we refer the reader to [13].

In Sect. 2, we present the set of rational numbers as an inductively defined set of signed binary sequences. In Sect. 3, we describe strict algorithms for the field operations. In Sect. 4, we describe lazy algorithms for these operations, based on homographic and quadratic transformations. In Sect. 5 we discuss the proof of correctness for these algorithms. In sect. 6, we

[1] A presentation of Stern-Brocot trees and related publications is given in [12]

discuss the question of program extraction as it is provided in the *Coq* type theory and the impact this question had on our formal work. Possible further work is mentioned in Sect. 7.

2 Rational Numbers as Binary Sequences

We define the set of positive rational numbers, inductively as:

```
Inductive Q+ :Set := nR: Q+→Q+| dL: Q+→Q+| One: Q+.
```

Using this we define the set of rational numbers, merely by adding a sign bit:

```
Inductive Q : Set := Zero: Q| Qpos: Q+→Q| Qneg: Q+→Q.
```

We provide an encoding function to map pairs of natural numbers p and q to the rational number representing $\frac{p}{q}$. This relies on an auxiliary function where recursion is bounded by an extra measure argument.

```
Fixpoint Q+c [p, q, n : nat] : Q+ :=
  Cases n of
    O ⇒ One
  | (S n') ⇒
    Cases (minus p q) of
      O ⇒ Cases (minus q p) of O ⇒ One | v ⇒ (dL (Q+c p v n')) end
    | v ⇒ (nR (Q+c v q n'))
    end
  end.
```

If either of p or q is zero, the outcome of this function is irrelevant. The main function always calls this auxiliary function with positive input.

```
Definition makeQ[m,n:Z]:=
  Cases m n of
    (POS _) (POS _) ⇒ (Qpos (Q+c (abs m) (abs n) (plus (abs m) (abs n))))
  | (NEG _) (NEG _) ⇒ (Qpos (Q+c (abs m) (abs n) (plus (abs m) (abs n))))
  | ZERO  _  ⇒ Zero
  | _  ZERO  ⇒ Zero
  | _ _      ⇒ (Qneg (Q+c (abs m) (abs n) (plus (abs m) (abs n))))
  end.
```

Here the function `abs` is the forgetful projection from `Z` onto `nat`. Thus for p, q two integers, ***makeQ*** `p q` produces the signed binary sequence corresponding to $\frac{p}{q}$. For example:

```
Eval Compute in (makeQ '9' '14').
= (Qpos (dL (nR (dL (nR (nR (nR One)))))) : Q
```

Decoding functions have a similar structure, with a main function and a recursive function. We proved formally that if $\texttt{Q}_i^+$ `w = (p,q)` then $p, q \neq 0$. Here the function `inject_nat` is the trivial injection of `nat` into `Z`.

```
Fixpoint Q_i^+ [w : Q^+] : nat * nat :=
  Cases w of
    One ⇒ ((S O), (S O))
  | (nR w') ⇒ let (p,q) = (Q_i^+ w') in ((plus p q), q)
  | (dL w') ⇒ let (p,q) = (Q_i^+ w') in (p, (plus p q))
  end.

Definition decodeQ [q:Q]:=
  Cases q of
    (Qpos p) ⇒ ((inject_nat (Fst (Q_i^+ p))),(inject_nat (Snd (Q_i^+ p))))
  | (Qneg p) ⇒ ((Zopp (inject_nat (Fst (Q_i^+ p)))),(inject_nat (Snd (Q_i^+ p))))
  | Zero ⇒ ( ZERO, (POS xH))
  end.
```

We also proved that encoding and decoding are inverse operations.

```
Lemma makeQdecodeQ:∀q:Q (makeQ (Fst (decodeQ q)) (Snd (decodeQ q)))=q.
```

Note that this equality between the resulting signed binary sequence and the original sequence is syntactical (Leibniz equality). For the converse direction we can prove the following lemma:

```
Lemma decodeQmakeQ:∀m,n:Z let (p,q)=(decodeQ (makeQ m n)) in n≠0→m*q=n*p.
```

Here the equality is not syntactical, rather it is the definitional equality on positive fractions. These lemmata show the advantage of our binary representation for rational numbers. In a system like *Coq*, reasoning with data types is considerably easier when we are dealing with the corresponding syntactical equality, because we can use the rewriting machinery of the theorem prover to ease the equational reasoning. But the benefits of this canonical representation are not restricted to machinery of the theorem provers. For a more detailed discussion and examples of simplified mathematical proofs see [3].

The lemmata *makeQdecodeQ* and *decodeQmakeQ* demonstrate that the inductively defined set `Q` is a representation for rational numbers. In the rest of this paper we will show how we can equip the set `Q` with the usual algebraic operations and prove the correctness of these operations.

3 Field Structure: A Strict Implementation

In this section, we present the formalisation of algebraic operations on `Q` in the natural mathematical way. When computing an operation with rational numbers, mathematicians usually

perform regular natural number computations with the numerators and denominators and then simplify the result to a reduced fraction, using a greatest common divisor computation. For this reason, we shall use the term *fraction* to denote the pair of a numerator and denominator.

In our case, we start with values in the type `Q` and we use the function ***decodeQ*** to obtain fractions for each operand and then perform the usual natural number computations. At the end of the computation, the resulting fraction is directly encoded using the function $\mathtt{Q}_c^+$, because this function already integrates the greatest common divisor computation, as was shown in [3].

We only provide operations to manipulate positive fractions. The encapsulating functions that take care of conversions between the type `Q` take care of sign problems. This means that we need to provide three basic operations for fractions: addition, multiplication, and subtraction. Both addition and subtraction on fractions with positive components are needed for the addition on rational numbers, because adding two numbers with opposite signs tantamounts to a subtraction. Computing the result sign for an addition when the two numbers have opposite sign also requires a function to compare two rational numbers. We do not implement this comparison function at the level of fractions but rather at the level of the type $\mathtt{Q}^+$. For multiplication, the situation is simpler, multiplying rational numbers reduces to multiplying the absolute values and then computing the sign of the result.

Comparing two numbers in the type $\mathtt{Q}^+$ is simple. The constructors `nR` and `dL` can actually be interpreted as monotonically increasing functions and the former always return a result greater than 1 while the second one always returns a result less than 1. Thus, it suffices to compare the two bit bitwise from left to right.

```
Fixpoint Q+_le_bool [w, w' : Q+] : bool :=
  Cases w of
    One ⇒ Cases w' of (dL y) ⇒ false | _ ⇒ true end
  | (dL y) ⇒ Cases w' of (dL y') ⇒ (Q+_le_bool y y') | _ ⇒ true end
  | (nR y) ⇒ Cases w' of (nR y') ⇒ (Q+_le_bool y y') | _ ⇒ false end
  end.
```

This function is used to defined a two argument predicate $\mathtt{Q}^+\mathtt{\_le}$ and a strongly specified test function $\mathtt{Q}^+\mathtt{\_le\_dec}$ which plays key role in the operations implementations, because the subtraction operation is only meaningful when the first argument is greater than the second argument.

```
Definition Qplus :=
[x, y:Q]Cases x y of (Qpos x')(Qpos y') ⇒ (Qpos (Q+_plus x' y'))
                   | (Qpos x') (Qneg y') ⇒
                       Case (Q+_le_dec x' y') of
                        [h:(Q+_le x' y')]
                         Case (Q+_eq_dec x' y') of
                          [h:x'=y']Zero  [h:x'≠y'](Qneg (Q+_sub y' x'))
                         end
                        [h :¬(Q+_le x' y')](Qpos (Q+_sub x' y'))
```

```
                    end
                  | (Qneg x') (Qneg y') ⇒ (Qneg (Q+_plus x' y'))
                  ...

Definition Qmult :=
[x, y:Q] Cases x y of (Qpos x') (Qpos y') ⇒ (Qpos (Q+_mult x' y'))
                    | (Qpos x') (Qneg y') ⇒ (Qneg (Q+_mult x' y'))
                    ...
```

The unary operations of computing the opposite of a rational number is a trivial matter. To compute the inverse, we need to compute the inverse of a positive integer in $\mathtt{Q}^+$. It actually is a very simple function, where we do not need to convert to a fraction of natural numbers and back.

```
Fixpoint Q+_inv [w:Q+]:Q+:=
  Cases w of One ⇒ One
           | (nR w') ⇒ (dL (Q+_inv w'))
           | (dL w') ⇒ (nR (Q+_inv w'))
           end.
```

With all these operations, it is then quite easy to show that the type `Q` with comparison, addition, subtraction, multiplication, and inversion is an ordered field. This 'natural' implementation provides a reference implementation (but not a very efficient one) of the field operations.

Most of these algorithms for the basic operations are strict, in the sense that we need to process the entire bit strings of both arguments to obtain the numerators and denominators and start computing the result. Only the inverse and comparison function can start to return results without having processed their entire input.

4 Field Structure: A Lazy Implementation

4.1 Laziness

Lazy computation is a constructive interpretation of continuity[2]. The idea is that if we are computing continuous functions on sequences, it is possible to do this computation in a lazy manner, outputting partial information about the final result after having processed only initial segments of the input. If we consider streams (infinite sequences) instead of finite sequences, we can make this notion more precise by calling a function on streams *lazy* if it is continuous with respect to the Cantor space topology on the set of streams. In our case the operations addition, multiplication, division and subtraction are all continuous both on $\mathbb{R}$ and on $\mathbb{Q}$ with the subspace topology, and we can devise lazy algorithms for these additions.

[2]There are other aspects of laziness (e.g. laziness in the sense of sharing the reduction) which we do not consider in this paper.

However, the inputs we consider are finite and this explains why we could provide a strict implementation.

A lazy algorithm on sequences usually consists of two steps: (1) Looking at initial segments of the input, the *absorption* step. (2) Outputting an initial segment of the output, the *emission* step. An algorithm terminates when it emits the empty sequence. When there are several inputs, the algorithm also contains an *absorption strategy* to decide which input initial segment to absorb next. Classical examples of lazy algorithms are those given by Gosper [7] for adding and multiplying regular continued fractions. The work by Gosper was later generalised in for exact arithmetic [18, 11, 16, 8].

We devised lazy algorithms to compute directly on the $\mathbb{Q}^+$ and $\mathbb{Q}$ structures without going through computations on fractions and we then showed their equivalence with the strict algorithms from section 3. One can show that the lazy approach should have a lower computational complexity, especially when partial answers are useful (for instance in case of dealing with fractions with large denominators). But we discover in this work that the proof complexity has an impact on the usability of the algorithm and the strict approach is still the most efficient for some purposes. The contrast between these two paradigms will be discussed again in Sect. 6.3.

4.2 Homographic and Quadratic Algorithms

In this section we briefly discuss the homographic and quadratic algorithms to compute on signed binary sequences. The algorithms that we use are explained in detail in [12] and are available on-line as part of the *Coq* contribution package *QArith* [13]. Following [7], to devise the basic field operations, we consider a larger class of unary and binary operations. The basic operations are then simultaneously obtained from the general algorithms.

A *homographic transformation* of matrix M is a function of the form:

$$h_M(x) = \frac{ax+b}{cx+d} \qquad a,b,c,d \in \mathbb{Z}; \qquad M = \begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

A *quadratic transformation* of matrix T is a binary function of the form:

$$q_T(x,y) = \frac{axy+bx+cy+d}{exy+fx+gy+h} \qquad a,b,c,d,e,f,g,h \in \mathbb{Z} \qquad \text{and } T = \begin{bmatrix} a & b & c & d \\ e & f & g & h \end{bmatrix}.$$

By taking special values for T we obtain the algorithms for basic arithmetic operations.

$$T_{\oplus} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T_{\otimes} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
$$T_{\ominus} = \begin{bmatrix} 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T_{\oslash} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

In [12] the homographic algorithm is presented using two auxiliary algorithms: *the sign algorithm* and *the output-bit algorithm*. The sign algorithm is a function $\mathcal{S}\_ : M_{2\times 2}(\mathbb{Z}) \times \mathbb{Q}^+ \longmapsto \{0,+1,-1\} \times M_{2\times 2}(\mathbb{Z}) \times \mathbb{Q}^+$ (here $M_{2\times 2}(\mathbb{Z})$ denotes the type of $\overline{2} \times 2$ matrices over $\mathbb{Z}$). The output-bit algorithm is a function $\mathcal{B}\_ : M_{2\times 2}(\mathbb{Z}) \times \mathbb{Q}^+ \longmapsto \mathbb{Q}^+$. Finally the

homographic algorithm is a function $\mathcal{H}\_ : M_{2\times 2}(\mathbb{Z}) \times \mathtt{Q}^+ \longmapsto \mathtt{Q}$ which combines the two functions $\mathcal{S}$ and $\mathcal{B}$. Both functions $\mathcal{S}$ and $\mathcal{B}$ are recursive. In the case of $\mathcal{S}$ we are dealing with a simple structural recursion on the binary sequence. The recursion in $\mathcal{B}$ is more complex. If the recursion was structural, then all recursive calls would be absorption steps. In our case, some recursive calls are only emission steps, but the total sum of the matrix coefficients decrease while remaining positive. This is one of the main difficulties of the verification process: in *Coq* formalising non-structural yet terminating recursion is possible in various ways but all of the methods either require *a priori* knowledge of the algorithm complexity (for example the Balaa and Bertot's method [1]) or lead to very large proof terms by changing the representation of the function domain (for example the Bove and Capretta's method [5]). In Sect. 4.4 we explain how we used in our formalisation a variant of Bove and Capretta's method to formalise the non-structural recursion.

Similarly in the case of the quadratic algorithm, the sign algorithm is a function $\mathcal{S}_2\_ : T_{2\times(2\times 2)}(\mathbb{Z}) \times \mathtt{Q}^+ \times \mathtt{Q}^+ \longmapsto \{0,+1,-1\} \times T_{2\times(2\times 2)}(\mathbb{Z}) \times \mathtt{Q}^+ \times \mathtt{Q}^+$ where $T_{2\times(2\times 2)}(\mathbb{Z})$ is the type of 2×4 integer matrices. The output-bit algorithm is a function $\mathcal{B}_2\_ : T_{2\times(2\times 2)}(\mathbb{Z}) \times \mathtt{Q}^+ \times \mathtt{Q}^+ \longmapsto \mathtt{Q}^+$ and the quadratic algorithm is a function $\mathcal{Q}\_ : T_{2\times(2\times 2)}(\mathbb{Z}) \times \mathtt{Q}^+ \times \mathtt{Q}^+ \longmapsto \mathtt{Q}$ which combines the two functions $\mathcal{S}_2$ and $\mathcal{B}_2$. The functions $\mathcal{S}_2$ is a structurally recursive function with respect to the binary structure of both inputs and the function $\mathcal{B}_2$ is not.

4.3 Lazy Proof Obligation

In [12] we implicitly assumed that the denominators of all transformations involved are nonzero. This imposes a restriction on the formalisation because it makes the algorithms partial. A standard way to formalise partial functions in type theory is to add a proof obligation to the function's arguments, using a predicate to specify function's domain. For $\mathcal{H}$, the domain predicate has the form

$$\lambda a, b, c, d : \mathbb{Z}\ ; q : \mathtt{Q}.\ \Phi_{\mathcal{H}}(c, d, q)$$

and the predicate $\Phi_{\mathcal{H}} : Z^2 \times \mathtt{Q} \to \mathtt{Prop}$ *means* $c{*}q{+}d{\neq}0$, but we want to avoid using the strict operations to define the predicate. We first define a domain predicate for the sign algorithm as an inductive property of triples $(c, d, p) : \mathbb{Z} \times \mathbb{Z} \times \mathtt{Q}^+$, using only operations on integers and pattern matching on the first bit of bit strings:

```
Inductive Φ_S:Z→Z→Q⁺→Prop:=
  Φ_S0:(c,d:Z;p:Q⁺)p=One→c+d≠0→(Φ_S c d p)                  (* p=One *)
| Φ_S1:(c,d:Z;xs:Q⁺)(Φ_S c c+d xs)→(Φ_S c d (nR xs))      (* p=(nR xs) *)
| Φ_S2:(c,d:Z;xs:Q⁺)(Φ_S c+d d xs)→(Φ_S c d (dL xs)).     (* p=(dL xs) *)
```

The domain predicate for *homo* is obtained by adapting $\Phi_{\mathcal{S}}$ according to the sign bit of each rational number.

```
Inductive Φ_H:Z→Z→Q→Prop:=
  Φ_H0:(c,d:Z;q:Q)q=Zero→d≠0→(Φ_H c d q)
| Φ_H1:(c,d:Z;q:Q;p:Q⁺)q=(Qpos p)→(Φ_S c d p)→(Φ_H c d q)
| Φ_H2:(c,d:Z;q:Q;p:Q⁺)q=(Qneg p)→(Φ_S -c d p)→(Φ_H c d q).
```

There is also a domain predicate for the output-bit algorithm, but this is a consequence of the accessibility predicate (Sect. 4.4).

The precise type of the *Coq* formalisation of the homographic algorithm becomes

$\mathcal{H}$:(a,b,c,d:Z;q:Q;H:($\Phi_{\mathcal{H}}$ c d q))→Q.

Note that the type of ($\Phi_{\mathcal{H}}$ c d q) is Prop, the proof obligation is removed during extraction (Sect. 6.1), and the extracted programs is close to those given in [12]. The same technique is used for the quadratic algorithm and the *Coq* function has the type
$\mathcal{Q}$:(a,b,c,d,e,f,g,h:Z;q1,q2:Q;H:($\Phi_{\mathcal{Q}}$ e f g h q1 q2))→Q.

We need only one lemma to prove that the usual field operations satisfy the proof obligations:

```
Lemma addmultPrf:(x,y:Q)(Φ_Q 0 0 0 1 x y).

Definition Qplus Lazy[x,y:Q]:= (Q 0 1 1 0 0 0 0 1 x y (addmultPrf x y)).
Definition Qmult Lazy[x,y:Q]:= (Q 1 0 0 0 0 0 0 1 x y (addmultPrf x y)).
```

4.4 Accessibility

The mathematical argument to ensure that the output-bit algorithms terminate is that the input sequence decreases in absorption steps and that the the total sum of matrix coefficients decrease in emission steps. Checking that the total sum is decreasing is not syntactically possible impromptu. We follow Bove and Capretta's method of formalising general recursive functions and define the function domain as an inductive predicate so that the algorithm can be described as a structural recursive function with respect to this predicate. This method is also known as recursion on an *ad hoc predicate*. For the reason that we discuss in Sect. 6.2, we use a variant of the method suggested by Paulin [14] and further explored in detail in [4, Ch. 14.4]. The domain of the function $\mathcal{B}$ will be

$$\lambda a, b, c, d : \mathbb{Z}\ p : \mathtt{Q}^+.\ \Psi_{\mathcal{H}}(a, b, c, d, p),$$

where $\Psi_{\mathcal{H}}(a, b, c, d, p)$: Prop is an inductively defined predicate that determines which of the 5-tuples (a, b, c, d, p) are accessible for the recursive branches of the output-bit algorithm (cf. [12, Def. 4.3]):

```
Definition isAbove[a,b,c,d:Z]:=(c≤a∧d<b)∨(c<a∧d≤b).

Inductive Ψ_H:Z→Z→Z→Z→Q⁺→Prop:=
  Ψ_H0:(a,b,c,d:Z;p:Q⁺)p=One→0<a+b→0<c+d→(Ψ_H a b c d p)
| Ψ_H1:(a,b,c,d:Z;p:Q⁺)p≠One→(isAbove a b c d)→
     (Ψ_H (a-c) (b-d) c d p)→(Ψ_H a b c d p)
| Ψ_H2:(a,b,c,d:Z;p:Q⁺)p≠One→¬(isAbove a b c d)→
```

```
      (isAbove c d a b)→(Ψ_ℋ a b c-a d-b p)→(Ψ_ℋ a b c d p)
  | Ψ_ℋ3:(a,b,c,d:Z;xs:Q⁺)¬(isAbove a b c d)→¬(isAbove c d a b)→
      (Ψ_ℋ a a+b c c+d xs)→(Ψ_ℋ a b c d (nR xs))
  | Ψ_ℋ3':(a,b,c,d:Z;xs:Q⁺)¬(isAbove a b c d)→¬(isAbove c d a b)→
      (Ψ_ℋ a+b b c+d d xs)→(Ψ_ℋ a b c d (dL xs)).
```

We use this accessibility predicate to formalise the homographic output-bit algorithm. This function's type becomes

```
ℋ_Q⁺_to_Q⁺:(a,b,c,d:Z; p:Q⁺; H_acc:(Ψ_ℋ a b c d p))→ Q⁺.
```

After defining this function in *Coq*, each time we want to use it we should supply a term `H_acc:(Ψ_ℋ a b c d p)`. But we know that all positive values of a, b, c, d are in the function's domain.

```
Lemma Ψ_ℋ_Wf:(a,b,c,d:Z;p:Q⁺)0<a+b→0<c+d→0≤a→
                          0≤b→0≤c→0≤d→(Ψ_ℋ a b c d p).
```

We prove this lemma by well-founded induction on the intrinsic order of the accessibility predicate. We denote this order by $<_5$ and we define it as follows:

$$(a_1, a_2, a_3, a_4, p) <_5 (a'_1, a'_2, a'_3, a'_4, q) \text{ iff } \mathsf{len}(p) < \mathsf{len}(q) \vee (p{=}q \wedge \sum_{i=1}^{4} a_i < \sum_{i=1}^{4} a'_i)$$

We prove that this order is well-founded on the set $\mathbb{Z}^{+4} \times \mathbb{Q}^+$ (where $\mathbb{Z}^+$ is the set of nonnegative integers) and the lemma $\Psi_{\mathcal{H}}\_Wf$ is a direct consequence.

We take the similar approach for the formalisation of the output-bit algorithm for the quadratic function. There we have a accessibility predicate $\Psi_{\mathcal{Q}}$ on $\mathbb{Z}^8 \times \mathbb{Q}^{+2}$. Using that we can define a function with the following type

```
Q_Q⁺_to_Q⁺:(a,b,c,d,e,f,g,h:Z;p1,p2:Q⁺;
                        H_acc:(Ψ_Q a b c d e f g h p1 p2))→Q⁺.
```

The well-founded order corresponding to this accessibility predicate is an order on 10-tuples defined as:

$$(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, p_1, p_2) <_{10} (a'_1, a'_2, a'_3, a'_4, a'_5, a'_6, a'_7, a'_8, q_1, q_2) \text{ iff}$$
$$\mathsf{len}(p_1) < \mathsf{len}(q_1) \vee (p_1{=}q_1 \wedge \sum_{i=1}^{8} a_i < \sum_{i=1}^{8} a'_i).$$

We can prove that $<_{10}$ is well founded on the set $\mathbb{Z}^{+8} \times \mathbb{Q}^{+2}$. Consequently we use well-founded induction to prove the following lemma:

```
Lemma Ψ_Q_Wf:(a,b,c,d,e,f,g,h:Z;p1,p2:Q⁺)0 <a+b+c+d→
        0 < e+f+g+h→0≤a→0≤b→0≤c→0≤d→
        0≤e→0≤f→0≤g→0≤h→(Ψ_Q a b c d e f g h p1 p2).
```

This lemma shows that we can use the output-bit algorithm (the function $\mathcal{B}_2$) to compute the quadratic transformations with nonnegative coefficients and with at least one positive coefficient in the numerator and one positive coefficient in the denominator. In order to compute the quadratic transformations with negative coefficients and on negative inputs we use the quadratic algorithm (the function $\mathcal{Q}$) to modify the coefficients with respect to the sign bit and call the output-bit algorithm with nonnegative coefficients.

5 Correctness Proofs

5.1 Using Strict Implementations

In Sects. 3 and 4 we showed how we formalised strict and lazy arithmetic operations on the data type `Q`. In this section we discuss how the lazy algorithms were formally proven to be correct. One possible approach is similar to what we did for strict operations: to prove that the lazy operations satisfy all the axioms of a field. The second possibility is to use the strict algorithms as a *specification* for the lazy ones, with lemmata of the following form:

```
Lemma QplusLazy_Qplus:(x,y:Q)(QplusLazy x y)=(Qplus x y).
```

In our development we took this approach and proved two general theorems. The one for the quadratic algorithm has the following form, momentarily using $\oplus$ and $\otimes$ do denote the strict operations.

```
Theorem Q_Correctness:(a,b,c,d,e,f,g,h:Z;q1,q2:Q;
                       H:(((((e⊗q1)⊗q2)⊕(f⊗ q1))⊕(g⊗q2))⊕ h) ≠ Zero)
 (Q a b c d e f g h q1 q2 (Q_nonzeroCorrect e f g h q1 q2 H))=
((((((a⊗q1)⊗q2)⊕(b⊗q1))⊕(c⊗q2))⊕d)⊗
    (Qinv (((((e⊗q1)⊗q2)⊕(f⊗q1))⊕(g⊗q2))⊕h))).
```

The correctness of field operations is just a special instance of the general theorems. In the statement of the above theorems, the term *Q_nonzeroCorrect* correspond to the correctness of the lazy proof obligations that we defined in Sect. 4.3. It shows the equivalence of the inductively defined predicate with the strict predicate stating that the denominators are not zero.

```
Lemma Q_nonzeroCorrect:(e,f,g,h:Z;q1,q2:Q)
(Q_nonzero e f g h q1 q2)↔(((((e⊗q1)⊗q2)⊕(f⊗q1))⊕(g⊗q2))⊕ h)≠Zero.
```

5.2 Using the `Field` Tactic

The `Field` tactic was devised by Delahaye and Mayero [6] to ease the equational reasoning on field structures in *Coq*. It is a decision for simple equations that generates proof obligations for each occurrence of division.

We could use this tactic directly after proving that the strict operations of Sect. 3 satisfy the field axioms. It was of great help in the correctness proof. However there were instances where we had to fine-tune the reduction behaviour of the `Field` tactic to prevent unnecessary reduction that slowed down the tactic behaviour to an unacceptable level. The default reduction behaviour is based on eager reduction, probably because the original design was based on an axiomatic field structure. Our experience shows that the `Field` tactic will be more useful for equational reasoning on concrete fields (as opposed to abstract axiomatic fields), if this reduction behaviour is less eager. Note that the `Ring` tactic, which is the *Coq* tactic for equational reasoning on rings, does not have this problem with eager reduction and hence it is very useful in reasoning about concrete rings such as the ring of integers.

After proving the correctness of the lazy algorithms, we could define a *second* field structure based on lazy operations. Therefore we have a single data type with two different field structures on it. This is an interesting situation and deserves a deeper investigation of whether it is useful — from the theorem proving point of view — to have two underlying fields on the same carrier type, or whether it adds to the complexity of the proofs.

5.3 Functional Induction

As it is obvious from the quadratic and homographic algorithms given in [12] and the formalisation we discussed in Sect. 4, we are dealing with functions of up to 11 arguments (in quadratic algorithms) which are defined by case distinctions of up to 43 cases (in homographic and quadratic sign algorithms). The case distinctions in the definition of functions gets in the way when we want to prove these functions' properties. This means that if we want to prove the correctness of the homographic sign algorithm, we should consider 43 different cases. During the proof of correctness many of these cases should be handled in a similar way, they can be solved by automatic tactics or are degenerate cases. The tactic `Functional Induction` and its variant `Functional Scheme` are designed by Barthe and Courtieu [2] to assist the user in dealing with this kind of situations by providing some automation. When given a *Coq* function, the tactic `Functional Induction` tries to automatically generate an elimination principle which is tailored to the shape of that function; and then applied this elimination principle on the current goal which generates all the possible different cases based on the case distinctions in the definition of the function. This will generate one subgoal for each case. The tactic then applies some heuristics to solve as many subgoals as possible. In the course of developing the *QArith* in proving the correctness of the lazy operations, we benefited immensely from the beta version of this tactic. Our usage also contributed in making this tactic more efficient by exposing some of the bugs of that version. Our experience shows that this tactic can make *Coq* a better framework

for reasoning about realistic algorithms which are often based on heavy case analysis on a multitude of arguments.

6 Programs vs. Proofs

The algorithms for lazy arithmetic on $\mathtt{Q}^+$ were first implemented in *Haskell*. The original *Haskell* implementation was about 16 kilobytes of code. The *Coq* formalisation of the lazy algorithms led to fixing some exception handling bugs in the original *Haskell* code. Moreover the *Coq* formalisation highlighted the symmetries between fractions, homographic transformations and quadratic transformations as members of the larger family of *multilinear functions*. This resulted in generalising the algorithms for multilinear forms of n variables [12]. Such improvements show some advantages of formalising functions in type theory. There is however, the disadvantage that formalising the programs in type theory is a time consuming process and the amount of automation and heuristics available in present day theorem provers is not satisfactory.

Table 1 shows the relative size (in kilobytes) of the various phases of formalisation. During this project we used the most novel facilities of *Coq*. The statistics in table 1 might thus discourage people from using type theoretical tools for verification purposes. Our answer is that, without the existing automation tools in recent versions of *Coq*, and without the recent theoretical advances, such a project would seem impossible only a couple of years ago. This makes us confident that in coming years, similar projects will help improving theorem provers to make them more programmer-oriented and will alleviate the task of formalising and verification of the algorithms in theorem provers based on type theory. Our second argument is that such formalisations have a generic nature which can be applied to similar algorithms. The lazy algorithms of *QArith*, are inspired by and very similar to the existing algorithms for arithmetic on continued fractions [7, 18, 11]. We believe that a verification of the continued fraction arithmetic is possible based on our *QArith* project (cf Sect. 7).

6.1 Extraction

One important aspect of type theory of *Coq* is the distinction between informative and non-informative objects. The informative objects are terms of type `Set`, and consist of those whose computational content is important for the programmer. The non-informative objects are terms of type `Prop` which bear solely a logical and not computational importance. Inside the type theory of *Coq*, however, these non-informative objects are first class citizens and they should be type checked and evaluated when necessary.

In order to recapture the computational content of the formalised programs *Coq* has *the program extraction mechanism*, which is a tool to extract the underlying program of an object in type theory into a program in a conventional programming language [15, 9].

In our case, after finishing the formalisation of the lazy algorithms, we used the program extraction into *Haskell*, to obtain the verified version of the algorithms. It is interesting

to compare the *Coq*-generated *Haskell* code (*postverification code*) with the original hand-written *Haskell* code (*preverification code*). Not surprisingly the basic algorithms have the same time and space complexity. The main difference is that in the postverification code all the usual data types such as natural numbers, booleans and integers are reimplemented as new algebraic data types in *Haskell*, while in the preverification code we use the data types already defined in the standard prelude of *Haskell* (and sometimes even built-in as primitive data types). This makes the preverification code much faster and also is the main reason of the difference of size of pre- and postverification code (see table 1).

Table 1: **Comparison of the ASCII size of programs and proofs.**

development	*Coq* code
strict operations	112 KB
lazy operations	748 KB
correctness	304 KB
total project	1164 KB

function	preverification	*Coq* code	postverification
homographic	8 KB	200 KB	20 KB
quadratic	8 KB	476 KB	60 KB
total lazy	16 KB	748 KB	88 KB

There is another difference between the pre- and postverification codes. Recall that in *Coq* we had to add lazy proof obligations and accessibility predicates to the definition of the functions. Those terms were non-informative objects from the programmer's point of view, hence they had the type `Prop`. During the extraction all the terms of type `Prop` will be replaced by the sole constructor of the unit data type, which is merely a dummy term in *Haskell*. Thus for example the homographic function in the postverification code has 6 arguments, while in the preverification code it has 5 arguments. This difference is practically negligible and doesn't affect the performance of the postverification code.

6.2 `Prop`-sorted Accessibility

Originally, in Bove and Capretta's method, the inductive domain of a non-structurally recursive function is a term of type `Set`. This is because Bove and Capretta work in Martin-Löf type theory where there is no distinction between `Set` and `Prop`. In the case of function $\mathcal{B}$ of Sect. 4.2, in every recursive branch we will once evaluate the the value of the function (based on input sequence and four coefficients which we carry around) and once evaluate the new subdomain of the new recursive call. Recall that if we extract this term from *Coq* to a *Haskell* program, all the terms of type `Set` will be extracted. This means that if we follow Bove and Capretta's method and take the domain of the function $\mathcal{B}$ to be an inductively

defined set rather than a predicate, in the *Haskell* extraction of the function the inductively defined accessible domain is also extracted, and this will considerably decrease the efficiency.

Incidentally that is the approach we took in the beginning. Later we modified the whole formalisation and we used the `Prop`-sorted accessibility. Our tests showed a 25% to 30% decrease in both time and memory usage of the extracted algorithms. However for evaluation inside *Coq*, the time and memory complexity of the proof objects does not change. This emphasises the importance of program extraction as one of the basic philosophies behind the design of the type system of *Coq* versus Martin-Löf type theory.

We mentioned that our first approach was to follow Bove and Capretta's original method and use `Set`-sorted accessibility. This is because unfortunately the second approach is more technical and requires an advanced knowledge of the internals of *Coq* [4, Ch. 14.4]. The first author initially applied the original Bove and Capretta's method and the second author showed how it is possible to modify the proofs to suit the `Prop`-sorted variant. During this modification a detailed study of the proof terms of *Coq* was necessary.

6.3 Computations Inside *Coq*

One of the main goals of the project was to supply *Coq* with a library of arithmetic on rational numbers. This library had to be similar to the existing libraries for natural numbers and integers. This means that we should at least be able to perform easy computations in the language of *Coq*. The *QArith* library fulfills this requirement. After defining the strict operations, one can add pretty printer and parser for expressions involving rational numbers. This is especially facilitated in recent versions of *Coq*, where user can easily extend the grammar of *Coq*. The syntactic sugars for work with current version (7.4) of *Coq* can be found in the file `Qsyntax.v` [13].

In Sect. 4.1 we argued that lazy functions on sequences are more efficient. This is true in a programming language like *Haskell* where we do not bother with termination checking. But could we use the lazy operations in order to do arithmetic *inside Coq*? The answer is negative, and the reason is that inside *Coq* all the proof obligations and accessibility predicates, albeit computationally irrelevant, should be type-checked and evaluated. Consequently a full evaluation of the quadratic function inside *Coq* results in a explosion of proof terms and with current computational power is impractical. However as we discussed in Sect. 6.1, the program extraction will obviate all the non-informative terms and we end up with an efficient program. Therefore for computations inside *Coq* we use the strict version of field operations.

7 Conclusion and Further Work

The experience with formalisation of *QArith* library shows that the *Coq* theorem prover, in its current state, not only is a good framework for formalisation of mathematical structures and their purely algebraic properties, but also is capable of being used to verify nontrivial algorithms. The algorithms we formalised, have the same underlying complexity as the

state of the art algorithms in the field of exact arithmetic [16]. We also contrasted the preverification code versus postverification code. This consists of starting from a hand-written code, formalising it in a theorem prover which offers the possibility of program extraction, and extracting into the verified executable code in the programming language of the origin. We believe that a more careful investigation of the difference between pre- and postverification codes gives the designers of programming languages (resp. theorem provers) valuable insight into logical (resp. computational) power of their creations.

We discuss two possible extensions of our work. One is to use the intrinsic similarity between our algorithms and the algorithms of continued fraction arithmetic in order to verify the correctness of those ubiquitous algorithms. This requires a clever reuse of the proof objects we supplied during the present work, in order to minimise the amount of additional effort. The recent work by Magaud [10] seems to provide a useful theoretical background for this approach.

The second possible improvement on our work is to extend the inductive data types and the lazy algorithms on them to coinductive data types and corecursive functions on them in order to obtain a verified exact arithmetic on real numbers. In *Haskell* there is no distinction between inductive types (data) and coinductive types (codata), so all our algorithms written in *Haskell* are valid for potentially infinite sequences. But in *Coq* there is a clear distinction between infinite and finite objects and one has to use coinductive types in order to formalise algorithms which work on streams, even though the extracted algorithms into *Haskell* will be identical to those for the finite sequences. In an upcoming work the first author will describe admissible representations — those which come with a intuitive notion of computability induced by Cantor space topology — based on Stern-Brocot tree and formalisable by means of coinductive types. A problem to tackle is the syntactic constraints that *Coq* puts on corecursive functions. These constraints are the dual constraint of structural recursion, and require similar approaches to the ones we discussed in Sects. 4.4, 6.2.

Acknowledgements The authors wish to thank Pierre Courtieu for his help on the use of `Functional Induction` tactic. This work was completed during the first author's visit to INRIA, made possible by a grant from the Dutch Organization for Scientific Research (NWO).

References

[1] A. Balaa and Y. Bertot. Fonctions récursives générales par itération en théorie des types. In *Proceedings of JFLA'2002*. INRIA, 2002.

[2] G. Barthe and P. Courtieu. Efficient Reasoning about Executable Specifications in Coq. In V. Carreño, C. Muñoz, and S. Tashar, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2002*, volume 2410 of *LNCS*, pages 31–46. Springer-Verlag, 2002.

[3] Y. Bertot. Simple canonical representation of rational numbers. In H. Geuvers and F. Kamareddine, editors, *Electronic Notes in Theoretical Computer Science*, volume 85(7). Elsevier, 2003.

[4] Y. Bertot and P. Castéran. Coq'Art. Draft of the english version, jul 2003.

[5] A. Bove and V. Capretta. Nested general recursion and partiality in type theory. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2001*, volume 2152 of *LNCS*, pages 121–135. Springer-Verlag, 2001.

[6] D. Delahaye and M. Mayero. Field: une procédure de décision pour les nombres réels en coq. In *Proceedings of JFLA'2001*. INRIA, 2001.

[7] R. W. Gosper. HAKMEM, Item 101 B. `http://www.inwap.com/pdp10/hbaker/hakmem/cf.html#item101b`, feb. 1972. MIT AI Laboratory Memo No.239.

[8] M. Konečný. *Many-Valued Real Functions Computable by Finite Transducers using IFS-Representations*. PhD thesis, The University of Birmingham, oct 2000.

[9] P. Letouzey. A New Extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, TYPES 2002*, volume 2646 of *LNCS*, pages 388–405. Springer-Verlag, 2003.

[10] N. Magaud. Changing Data Representation within the Coq System. In D. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2003*, volume 2758 of *LNCS*, pages 87–102. Springer-Verlag, 2003.

[11] V. Ménissier-Morain. *Arithmétique exacte, conception, algorithmique et performances d'une implémentation informatique en précision arbitraire*. Thèse, Université Paris 7, dec 1994.

[12] M. Niqui. Exact Arithmetic on Stern-Brocot Tree. *submitted for publication*, `http://www.cs.kun.nl/~milad/publications/exact.pdf`, jan 2003.

[13] M. Niqui and Y. Bertot. `http://coqcvs.inria.fr/cgi-bin/cvswebcoq.cgi/contrib/Nijmegen/QArith/`, may 2003. Coq contribution.

[14] Ch. Paulin. Coq club mailing list correspondence, `http://coq.inria.fr/mailing-lists/coqclub/200208/msg00003.html`, aug 2002.

[15] Ch. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Proceedings of POPL 1989*, pages 89–104. ACM, January 1989.

[16] P. J. Potts. *Exact Real Arithmetic using Möbius Transformations*. PhD thesis, Uiversity of London, Imperial College, jul 1998.

[17] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 7.4*. INRIA, `http://coq.inria.fr/doc/main.html`, feb 2003.

[18] J. E. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8):1087–1105, aug 1990.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399