Implementing field extensions of the form $\mathbb{Q}[\sqrt{b}]$*

René Thiemann

March 17, 2025

Abstract

We apply data refinement to implement the real numbers, where we support all numbers in the field extension $\mathbb{Q}[\sqrt{b}]$, i.e., all numbers of the form $p + q\sqrt{b}$ for rational numbers p and q and some fixed natural number b. To this end, we also developed algorithms to precisely compute roots of a rational number, and to perform a factorization of natural numbers which eliminates duplicate prime factors.

Our results have been used to certify termination proofs which involve polynomial interpretations over the reals.

Contents

1 Introduction

It has been shown that polynomial interpretations over the reals are strictly more powerful for termination proving than polynomial interpretations over the rationals. To this end, also automated termination prover started to generate such interpretations. [3, 4, 5, 7, 8]. However, for all current implementations, only reals of the form $p + q \cdot \sqrt{b}$ are generated where b is some fixed natural number and p and q may be arbitrary rationals, i.e., we get numbers within $\mathbb{Q}[\sqrt{b}]$.

*This research is supported by FWF (Austrian Science Fund) project P22767-N13.

To support these termination proofs in our certifier CeTA [6], we therefore required executable functions on $\mathbb{Q}[\sqrt{b}]$, which can then be used as an implementation type for the reals. Here, we used ideas from [1, 2] to provide a sufficiently powerful partial implementations via data refinement.

2 Auxiliary lemmas which might be moved into the Isabelle distribution.

theory *Real-Impl-Auxiliary*
imports
  HOL−Computational-Algebra.Primes
begin

lemma *multiplicity-prime*:
  assumes *p*: *prime* $(i :: nat)$ **and** *ji*: $j \neq i$
  shows *multiplicity* $j\ i = 0$
  ⟨*proof*⟩

end

3 Prime products

theory *Prime-Product*
imports
  Real-Impl-Auxiliary
  Sqrt-Babylonian.Sqrt-Babylonian
begin

Prime products are natural numbers where no prime factor occurs more than once.

definition *prime-product*
  where *prime-product* $(n :: nat) = (\forall\ p.\ prime\ p \longrightarrow multiplicity\ p\ n \leq 1)$

The main property is that whenever b_1 and b_2 are different prime products, then $p_1 + q_1\sqrt{b_1} = p_2 + q_2\sqrt{b_2}$ implies $(p_1, q_1, b_1) = (p_2, q_2, b_2)$ for all rational numbers p_1, q_1, p_2, q_2. This is the key property to uniquely represent numbers in $\mathbb{Q}[\sqrt{b}]$ by triples. In the following we develop an algorithm to decompose any natural number n into $n = s^2 \cdot p$ for some s and prime product p.

function *prime-product-factor-main* :: $nat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow nat \times nat$ **where**
  prime-product-factor-main factor-sq factor-pr limit n i =
    (*if* $i \leq limit \wedge i \geq 2$ *then*
      (*if i dvd n*
        then (*let* $n' = n$ *div i in*
          (*if i dvd* n' *then*

let n'' = n' div i in
prime-product-factor-main (factor-sq ∗ i) factor-pr (nat (root-nat-floor 3 n'')) n'' i
else
(case sqrt-nat n' of
Cons sn - ⇒ (factor-sq ∗ sn, factor-pr ∗ i)
| [] ⇒ prime-product-factor-main factor-sq (factor-pr ∗ i) (nat (root-nat-floor 3 n')) n' (Suc i)
)
)
)
else
prime-product-factor-main factor-sq factor-pr limit n (Suc i))
else
(factor-sq, factor-pr ∗ n)) ⟨*proof*⟩

termination
⟨*proof*⟩

lemma *prime-product-factor-main*: **assumes** ¬ (∃ *s*. *s* ∗ *s* = *n*)
and *limit* = *nat* (*root-nat-floor 3 n*)
and *m* = *factor-sq* ∗ *factor-sq* ∗ *factor-pr* ∗ *n*
and *prime-product-factor-main factor-sq factor-pr limit n i* = (*sq*, *p*)
and *i* ≥ *2*
and ⋀ *j*. *j* ≥ *2* ⟹ *j* < *i* ⟹ ¬ *j dvd n*
and ⋀ *j*. *prime j* ⟹ *j* < *i* ⟹ *multiplicity j factor-pr* ≤ *1*
and ⋀ *j*. *prime j* ⟹ *j* ≥ *i* ⟹ *multiplicity j factor-pr* = *0*
and *factor-pr* > *0*
shows *m* = *sq* ∗ *sq* ∗ *p* ∧ *prime-product p*
⟨*proof*⟩

definition *prime-product-factor* :: *nat* ⇒ *nat* × *nat* **where**
prime-product-factor n = (*case sqrt-nat n of*
(*Cons s -*) ⇒ (*s*,*1*)
| [] ⇒ *prime-product-factor-main 1 1* (*nat* (*root-nat-floor 3 n*)) *n 2*)

lemma *prime-product-one*[*simp*, *intro*]: *prime-product 1*
⟨*proof*⟩

lemma *prime-product-factor*: **assumes** *pf*: *prime-product-factor n* = (*sq*,*p*)
shows *n* = *sq* ∗ *sq* ∗ *p* ∧ *prime-product p*
⟨*proof*⟩

end

4 A representation of real numbers via triples

theory *Real-Impl*

imports
Sqrt-Babylonian.Sqrt-Babylonian
begin

We represent real numbers of the form $p + q \cdot \sqrt{b}$ for $p, q \in \mathbb{Q}$, $n \in \mathbb{N}$ by triples (p, q, b). However, we require the invariant that $\sqrt{b}$ is irrational. Most binary operations are implemented via partial functions where the common the restriction is that the numbers b in both triples have to be identical. So, we support addition of $\sqrt{2} + \sqrt{2}$, but not $\sqrt{2} + \sqrt{3}$.

The set of natural numbers whose sqrt is irrational

definition *sqrt-irrat* = { *q* :: *nat*. ¬ (∃ *p*. *p* ∗ *p* = *rat-of-nat q*)}

lemma *sqrt-irrat*: **assumes** *choice*: *q* = *0* ∨ *b* ∈ *sqrt-irrat*
  and *eq*: *real-of-rat p* + *real-of-rat q* ∗ *sqrt* (*of-nat b*) = *0*
  shows *q* = *0*
⟨*proof*⟩

To represent numbers of the form $p + q \cdot \sqrt{b}$, use mini algebraic numbers, i.e., triples (p, q, b) with irrational $\sqrt{b}$.

typedef *mini-alg* =
{(*p*,*q*,*b*) | (*p* :: *rat*) (*q* :: *rat*) (*b* :: *nat*).
q = *0* ∨ *b* ∈ *sqrt-irrat*}
⟨*proof*⟩

setup-lifting *type-definition-mini-alg*

lift-definition *real-of* :: *mini-alg* ⇒ *real* **is**
λ (*p*,*q*,*b*). *of-rat p* + *of-rat q* ∗ *sqrt* (*of-nat b*) ⟨*proof*⟩

lift-definition *ma-of-rat* :: *rat* ⇒ *mini-alg* **is** λ *x*. (*x*,*0*,*0*) ⟨*proof*⟩

lift-definition *ma-rat* :: *mini-alg* ⇒ *rat* **is** *fst* ⟨*proof*⟩
lift-definition *ma-base* :: *mini-alg* ⇒ *nat* **is** *snd o snd* ⟨*proof*⟩
lift-definition *ma-coeff* :: *mini-alg* ⇒ *rat* **is** *fst o snd* ⟨*proof*⟩

lift-definition *ma-uminus* :: *mini-alg* ⇒ *mini-alg* **is**
λ (*p1*,*q1*,*b1*). (− *p1*, − *q1*, *b1*) ⟨*proof*⟩

lift-definition *ma-compatible* :: *mini-alg* ⇒ *mini-alg* ⇒ *bool* **is**
λ (*p1*,*q1*,*b1*) (*p2*,*q2*,*b2*). *q1* = *0* ∨ *q2* = *0* ∨ *b1* = *b2* ⟨*proof*⟩

definition *ma-normalize* :: *rat* × *rat* × *nat* ⇒ *rat* × *rat* × *nat* **where**
ma-normalize x ≡ *case x of* (*a*,*b*,*c*) ⇒ *if b* = *0 then* (*a*,*0*,*0*) *else* (*a*,*b*,*c*)

lemma *ma-normalize-case*[*simp*]: (*case ma-normalize r of* (*a*,*b*,*c*) ⇒ *real-of-rat a* + *real-of-rat b* ∗ *sqrt* (*of-nat c*))
= (*case r of* (*a*,*b*,*c*) ⇒ *real-of-rat a* + *real-of-rat b* ∗ *sqrt* (*of-nat c*))
⟨*proof*⟩

lift-definition *ma-plus* :: *mini-alg* ⇒ *mini-alg* ⇒ *mini-alg* **is**
λ (*p1*,*q1*,*b1*) (*p2*,*q2*,*b2*). *if q1* = *0 then*
(*p1* + *p2*, *q2*, *b2*) *else ma-normalize* (*p1* + *p2*, *q1* + *q2*, *b1*) ⟨*proof*⟩

lift-definition *ma-times* :: *mini-alg* ⇒ *mini-alg* ⇒ *mini-alg* **is**
λ (*p1*,*q1*,*b1*) (*p2*,*q2*,*b2*). *if q1* = *0 then*
ma-normalize (*p1*∗*p2*, *p1*∗*q2*, *b2*) *else*
ma-normalize (*p1*∗*p2* + *of-nat b2*∗*q1*∗*q2*, *p1*∗*q2* + *q1*∗*p2*, *b1*) ⟨*proof*⟩

lift-definition *ma-inverse* :: *mini-alg* ⇒ *mini-alg* **is**
λ (*p*,*q*,*b*). *let d* = *inverse* (*p* ∗ *p* − *of-nat b* ∗ *q* ∗ *q*) *in*
ma-normalize (*p* ∗ *d*, − *q* ∗ *d*, *b*) ⟨*proof*⟩

lift-definition *ma-floor* :: *mini-alg* ⇒ *int* **is**
λ (*p*,*q*,*b*). *case* (*quotient-of p*,*quotient-of q*) *of* ((*z1*,*n1*),(*z2*,*n2*)) ⇒
let z2n1 = *z2* ∗ *n1*; *z1n2* = *z1* ∗ *n2*; *n12* = *n1* ∗ *n2*; *prod* = *z2n1* ∗ *z2n1* ∗ *int b in*
(*z1n2* + (*if z2n1* ≥ *0 then sqrt-int-floor-pos prod else* − *sqrt-int-ceiling-pos prod*)) *div n12* ⟨*proof*⟩

lift-definition *ma-sqrt* :: *mini-alg* ⇒ *mini-alg* **is**
λ (*p*,*q*,*b*). *let* (*a*,*b*) = *quotient-of p*; *aa* = *abs* (*a* ∗ *b*) *in*
case sqrt-int aa of [] ⇒ (*0*,*inverse* (*of-int b*),*nat aa*) | (*Cons s* -) ⇒ (*of-int s* / *of-int b*,*0*,*0*)
⟨*proof*⟩

lift-definition *ma-equal* :: *mini-alg* ⇒ *mini-alg* ⇒ *bool* **is**
λ (*p1*,*q1*,*b1*) (*p2*,*q2*,*b2*).
p1 = *p2* ∧ *q1* = *q2* ∧ (*q1* = *0* ∨ *b1* = *b2*) ⟨*proof*⟩

lift-definition *ma-ge-0* :: *mini-alg* ⇒ *bool* **is**
λ (*p*,*q*,*b*). *let bqq* = *of-nat b* ∗ *q* ∗ *q*; *pp* = *p* ∗ *p in*
0 ≤ *p* ∧ *bqq* ≤ *pp* ∨ *0* ≤ *q* ∧ *pp* ≤ *bqq* ⟨*proof*⟩

lift-definition *ma-is-rat* :: *mini-alg* ⇒ *bool* **is**
λ (*p*,*q*,*b*). *q* = *0* ⟨*proof*⟩

definition *ge-0* :: *real* ⇒ *bool* **where** [*code del*]: *ge-0 x* = (*x* ≥ *0*)

lemma *ma-ge-0*: *ge-0* (*real-of x*) = *ma-ge-0 x*
⟨*proof*⟩

lemma *ma-0*: *0* = *real-of* (*ma-of-rat 0*) ⟨*proof*⟩

lemma *ma-1*: *1* = *real-of* (*ma-of-rat 1*) ⟨*proof*⟩

lemma *ma-uminus*:
− (*real-of x*) = *real-of* (*ma-uminus x*)

⟨*proof*⟩

lemma *ma-inverse*: *inverse* (*real-of r*) = *real-of* (*ma-inverse r*)
⟨*proof*⟩

lemma *ma-sqrt-main*: *ma-rat r* ≥ *0* ⟹ *ma-coeff r* = *0* ⟹ *sqrt* (*real-of r*) = *real-of* (*ma-sqrt r*)
⟨*proof*⟩

lemma *ma-sqrt*: *sqrt* (*real-of r*) = (*if ma-coeff r* = *0 then*
(*if ma-rat r* ≥ *0 then real-of* (*ma-sqrt r*) *else* − *real-of* (*ma-sqrt* (*ma-uminus r*)))
else Code.abort (*STR ''cannot represent sqrt of irrational number''*) (λ -. *sqrt* (*real-of r*)))
⟨*proof*⟩

lemma *ma-plus*:
(*real-of r1* + *real-of r2*) = (*if ma-compatible r1 r2*
then real-of (*ma-plus r1 r2*) *else*
Code.abort (*STR ''different base''*) (λ -. *real-of r1* + *real-of r2*))
⟨*proof*⟩

lemma *ma-times*:
(*real-of r1* ∗ *real-of r2*) = (*if ma-compatible r1 r2*
then real-of (*ma-times r1 r2*) *else*
Code.abort (*STR ''different base''*) (λ -. *real-of r1* ∗ *real-of r2*))
⟨*proof*⟩

lemma *ma-equal*:
HOL.equal (*real-of r1*) (*real-of r2*) = (*if ma-compatible r1 r2*
then ma-equal r1 r2 else
Code.abort (*STR ''different base''*) (λ -. *HOL.equal* (*real-of r1*) (*real-of r2*)))
⟨*proof*⟩

lemma *ma-floor*: *floor* (*real-of r*) = *ma-floor r*
⟨*proof*⟩

lemma *comparison-impl*:
(*x* :: *real*) ≤ (*y* :: *real*) = *ge-0* (*y* − *x*)
(*x* :: *real*) < (*y* :: *real*) = (*x* ≠ *y* ∧ *ge-0* (*y* − *x*))
⟨*proof*⟩

lemma *ma-of-rat*: *real-of-rat r* = *real-of* (*ma-of-rat r*)
⟨*proof*⟩

definition *is-rat* :: *real* ⇒ *bool* **where**
[*code-abbrev*]: *is-rat x* ⟷ *x* ∈ ℚ

lemma *ma-is-rat*: *is-rat* (*real-of x*) = *ma-is-rat x*
⟨*proof*⟩

definition *sqrt-real x* = (*if x* ∈ ℚ ∧ *x* ≥ *0 then* (*if x* = *0 then* [*0*] *else* (*let sx* = *sqrt x in* [*sx*,−*sx*])) *else* [])

lemma *sqrt-real*[*simp*]: **assumes** *x*: *x* ∈ ℚ
  shows *set* (*sqrt-real x*) = {*y* . *y* ∗ *y* = *x*}
⟨*proof*⟩

code-datatype *real-of*

declare [[*code drop*:
  plus :: *real* ⇒ *real* ⇒ *real*
  uminus :: *real* ⇒ *real*
  times :: *real* ⇒ *real* ⇒ *real*
  inverse :: *real* ⇒ *real*
  floor :: *real* ⇒ *int*
  sqrt
  HOL.equal :: *real* ⇒ *real* ⇒ *bool*
]]

lemma [*code*]:
  Ratreal = *real-of* ∘ *ma-of-rat*
  ⟨*proof*⟩

lemmas *ma-code-eqns* [*code equation*] = *ma-ge-0 ma-floor ma-0 ma-1 ma-uminus ma-inverse ma-sqrt ma-plus ma-times ma-equal ma-is-rat*
  comparison-impl

lemma [*code equation*]:
  (*x* :: *real*) / (*y* :: *real*) = *x* ∗ *inverse y*
  (*x* :: *real*) − (*y* :: *real*) = *x* + (− *y*)
  ⟨*proof*⟩

Some tests with small numbers. To work on larger number, one should additionally import the theories for efficient calculation on numbers

value ⌊*101.1* ∗ (*3* ∗ *sqrt 2* + *6* ∗ *sqrt 0.5*)⌋
value ⌊*606.2* ∗ *sqrt 2* + *0.001*⌋
value *101.1* ∗ (*3* ∗ *sqrt 2* + *6* ∗ *sqrt 0.5*) = *606.2* ∗ *sqrt 2* + *0.001*
value *101.1* ∗ (*3* ∗ *sqrt 2* + *6* ∗ *sqrt 0.5*) > *606.2* ∗ *sqrt 2* + *0.001*
value (*sqrt 0.1* ∈ ℚ, *sqrt* (− *0.09*) ∈ ℚ)

end

5 A unique representation of real numbers via triples

theory *Real-Unique-Impl*
imports
  Prime-Product

Real-Impl
Show.Show-Instances
Show.Show-Real
begin

We implement the real numbers again using triples, but now we require an additional invariant on the triples, namely that the base has to be a prime product. This has the consequence that the mapping of triples into ℝ is injective. Hence, equality on reals is now equality on triples, which can even be executed in case of different bases. Similarly, we now also allow different basis in comparisons. Ultimately, injectivity allows us to define a show-function for real numbers, which pretty prints real numbers into strings.

typedef *mini-alg-unique* =
{ *r* :: *mini-alg* . *ma-coeff r* = *0* ∧ *ma-base r* = *0* ∨ *ma-coeff r* ≠ *0* ∧ *prime-product* (*ma-base r*)}
⟨*proof*⟩

setup-lifting *type-definition-mini-alg-unique*

lift-definition *real-of-u* :: *mini-alg-unique* ⇒ *real* **is** *real-of* ⟨*proof*⟩
lift-definition *mau-floor* :: *mini-alg-unique* ⇒ *int* **is** *ma-floor* ⟨*proof*⟩
lift-definition *mau-of-rat* :: *rat* ⇒ *mini-alg-unique* **is** *ma-of-rat* ⟨*proof*⟩
lift-definition *mau-rat* :: *mini-alg-unique* ⇒ *rat* **is** *ma-rat* ⟨*proof*⟩
lift-definition *mau-base* :: *mini-alg-unique* ⇒ *nat* **is** *ma-base* ⟨*proof*⟩
lift-definition *mau-coeff* :: *mini-alg-unique* ⇒ *rat* **is** *ma-coeff* ⟨*proof*⟩
lift-definition *mau-uminus* :: *mini-alg-unique* ⇒ *mini-alg-unique* **is** *ma-uminus* ⟨*proof*⟩
lift-definition *mau-compatible* :: *mini-alg-unique* ⇒ *mini-alg-unique* ⇒ *bool* **is** *ma-compatible* ⟨*proof*⟩
lift-definition *mau-ge-0* :: *mini-alg-unique* ⇒ *bool* **is** *ma-ge-0* ⟨*proof*⟩
lift-definition *mau-inverse* :: *mini-alg-unique* ⇒ *mini-alg-unique* **is** *ma-inverse*
⟨*proof*⟩
lift-definition *mau-plus* :: *mini-alg-unique* ⇒ *mini-alg-unique* ⇒ *mini-alg-unique* **is** *ma-plus*
⟨*proof*⟩
lift-definition *mau-times* :: *mini-alg-unique* ⇒ *mini-alg-unique* ⇒ *mini-alg-unique* **is** *ma-times*
⟨*proof*⟩
lift-definition *ma-identity* :: *mini-alg* ⇒ *mini-alg* ⇒ *bool* **is** (=) ⟨*proof*⟩
lift-definition *mau-equal* :: *mini-alg-unique* ⇒ *mini-alg-unique* ⇒ *bool* **is** *ma-identity* ⟨*proof*⟩
lift-definition *mau-is-rat* :: *mini-alg-unique* ⇒ *bool* **is** *ma-is-rat* ⟨*proof*⟩

lemma *Ratreal-code*[*code*]:
Ratreal = *real-of-u* ∘ *mau-of-rat*
⟨*proof*⟩

lemma *mau-floor*: *floor* (*real-of-u r*) = *mau-floor r*

⟨*proof*⟩
lemma *mau-inverse*: *inverse* (*real-of-u r*) = *real-of-u* (*mau-inverse r*)
⟨*proof*⟩
lemma *mau-uminus*: − (*real-of-u r*) = *real-of-u* (*mau-uminus r*)
⟨*proof*⟩
lemma *mau-times*:
(*real-of-u r1* ∗ *real-of-u r2*) = (*if mau-compatible r1 r2*
then real-of-u (*mau-times r1 r2*) *else*
Code.abort (*STR ''different base''*) (λ -. *real-of-u r1* ∗ *real-of-u r2*))
⟨*proof*⟩
lemma *mau-plus*:
(*real-of-u r1* + *real-of-u r2*) = (*if mau-compatible r1 r2*
then real-of-u (*mau-plus r1 r2*) *else*
Code.abort (*STR ''different base''*) (λ -. *real-of-u r1* + *real-of-u r2*))
⟨*proof*⟩

lemma *real-of-u-inj*[*simp*]: *real-of-u x* = *real-of-u y* ⟷ *x* = *y*
⟨*proof*⟩

lift-definition *mau-sqrt* :: *mini-alg-unique* ⇒ *mini-alg-unique* **is**
λ *ma. let* (*a*,*b*) = *quotient-of* (*ma-rat ma*); (*sq*,*fact*) = *prime-product-factor* (*nat* (*abs a* ∗ *b*));
ma′ = *ma-of-rat* (*of-int* (*sgn*(*a*)) ∗ *of-nat sq* / *of-int b*)
in ma-times ma′ (*ma-sqrt* (*ma-of-rat* (*of-nat fact*)))
⟨*proof*⟩

lemma *sqrt-sgn*[*simp*]: *sqrt* (*of-int* (*sgn a*)) = *of-int* (*sgn a*)
⟨*proof*⟩

lemma *mau-sqrt-main*: *mau-coeff r* = *0* ⟹ *sqrt* (*real-of-u r*) = *real-of-u* (*mau-sqrt r*)
⟨*proof*⟩

lemma *mau-sqrt*: *sqrt* (*real-of-u r*) = (*if mau-coeff r* = *0 then*
real-of-u (*mau-sqrt r*)
else Code.abort (*STR ''cannot represent sqrt of irrational number''*) (λ -. *sqrt* (*real-of-u r*)))
⟨*proof*⟩

lemma *mau-0*: *0* = *real-of-u* (*mau-of-rat 0*) ⟨*proof*⟩

lemma *mau-1*: *1* = *real-of-u* (*mau-of-rat 1*) ⟨*proof*⟩

lemma *mau-equal*:
HOL.equal (*real-of-u r1*) (*real-of-u r2*) = *mau-equal r1 r2* ⟨*proof*⟩

lemma *mau-ge-0*: *ge-0* (*real-of-u x*) = *mau-ge-0 x* ⟨*proof*⟩

definition *real-lt* :: *real* ⇒ *real* ⇒ *bool* **where** *real-lt* = (<)

The following code equation terminates if it is started on two different inputs.

lemma *real-lt* [*code equation*]: *real-lt x y* = (*let fx* = *floor x*; *fy* = *floor y in*
(*if fx* < *fy then True else if fx* > *fy then False else real-lt* (*x* ∗ *1024*) (*y* ∗ *1024*)))
⟨*proof*⟩

For comparisons we first check for equality. Then, if the bases are compatible we can just compare the differences with 0. Otherwise, we start the recursive algorithm *real-lt* which works on arbitrary bases. In this way, we have an implementation of comparisons which can compare all representable numbers.

Note that in *Real-Impl.Real-Impl* we did not use *real-lt* as there the code-equations for equality already require identical bases.

lemma *comparison-impl*:
real-of-u x ≤ *real-of-u y* ⟷ *real-of-u x* = *real-of-u y* ∨
(*if mau-compatible x y then ge-0* (*real-of-u y* − *real-of-u x*) *else real-lt* (*real-of-u x*) (*real-of-u y*))
real-of-u x < *real-of-u y* ⟷ *real-of-u x* ≠ *real-of-u y* ∧
(*if mau-compatible x y then ge-0* (*real-of-u y* − *real-of-u x*) *else real-lt* (*real-of-u x*) (*real-of-u y*))
⟨*proof*⟩

lemma *mau-is-rat*: *is-rat* (*real-of-u x*) = *mau-is-rat x* ⟨*proof*⟩

lift-definition *ma-show-real* :: *mini-alg* ⇒ *string* **is**
λ (*p*,*q*,*b*). *let sb* = *shows* ''*sqrt*('' ∘ *shows b* ∘ *shows* '')'';
qb = (*if q* = *1 then sb else if q* = −*1 then shows* ''−'' ∘ *sb else shows q* ∘ *shows* ''∗'' ∘ *sb*) *in*
if q = *0 then shows p* [] *else*
if p = *0 then qb* [] *else*
if q < *0 then* ((*shows p* ∘ *qb*) [])
else ((*shows p* ∘ *shows* ''+'' ∘ *qb*) []) ⟨*proof*⟩

lift-definition *mau-show-real* :: *mini-alg-unique* ⇒ *string* **is** *ma-show-real* ⟨*proof*⟩

overloading *show-real* ≡ *show-real*
begin
definition *show-real*
where *show-real x* ≡
(*if* (∃ *y*. *x* = *real-of-u y*) *then mau-show-real* (*THE y*. *x* = *real-of-u y*) *else* [])
end

lemma *mau-show-real*: *show-real* (*real-of-u x*) = *mau-show-real x*
⟨*proof*⟩

code-datatype *real-of-u*

declare [[*code drop*:

plus :: *real* ⇒ *real* ⇒ *real*
uminus :: *real* ⇒ *real*
times :: *real* ⇒ *real* ⇒ *real*
inverse :: *real* ⇒ *real*
floor :: *real* ⇒ *int*
sqrt
HOL.equal :: *real* ⇒ *real* ⇒ *bool*
ge-0
is-rat
less :: *real* ⇒ *real* ⇒ *bool*
less-eq :: *real* ⇒ *real* ⇒ *bool*
]]

lemmas *mau-code-eqns* [*code*] = *mau-floor mau-0 mau-1 mau-uminus mau-inverse mau-sqrt mau-plus mau-times mau-equal mau-ge-0 mau-is-rat mau-show-real comparison-impl*

Some tests with small numbers. To work on larger number, one should additionally import the theories for efficient calculation on numbers

value $\lfloor 101.1 * (sqrt\ 18 + 6 * sqrt\ 0.5) \rfloor$
value $\lfloor 324 * sqrt\ 7 + 0.001 \rfloor$
value $101.1 * (sqrt\ 18 + 6 * sqrt\ 0.5) = 324 * sqrt\ 7 + 0.001$
value $101.1 * (sqrt\ 18 + 6 * sqrt\ 0.5) > 324 * sqrt\ 7 + 0.001$
value $show\ (101.1 * (sqrt\ 18 + 6 * sqrt\ 0.5))$
value $(sqrt\ 0.1 \in \mathbb{Q},\ sqrt\ (-\ 0.09) \in \mathbb{Q})$

end

Acknowledgements

We thank Bertram Felgenhauer for interesting discussions and especially for mentioning Cauchy's mean theorem during the formalization of the algorithms for computing roots.

References

[1] F. Haftmann, A. Krauss, O. Kunčar, and T. Nipkow. Data refinement in Isabelle/HOL. In *Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 100–115, 2013.

[2] A. Lochbihler. Light-weight containers for Isabelle: Efficient, extensible, nestable. In *Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 116–132, 2013.

[3] S. Lucas. Polynomials over the reals in proofs of termination: From theory to practice. *RAIRO Theoretical Informatics and Applications*, 39(3):547–586, 2005.

[4] S. Lucas. On the relative power of polynomials with real, rational, and integer coefficients in proofs of termination of rewriting. *Appl. Algebra in Engineering, Communication and Computing*, 17(1):49–73, 2006.

[5] S. Lucas. Practical use of polynomials over the reals in proofs of termination. In *Principles and Practice of Declarative Programming*, pages 39–50, 2007.

[6] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Theorem Proving in Higher Order Logics*, LNCS 5674, pages 452–468, 2009.

[7] H. Zankl and A. Middeldorp. Satisfiability of non-linear (ir)rational arithmetic. In *Logic for Programming and Automated Reasoning*, volume 6355, pages 481–500, 2010.

[8] H. Zankl, R. Thiemann, and A. Middeldorp. Satisfiability of non-linear arithmetic over algebraic numbers. In *Symbolic Computation in Software Science*, volume 10-10 of *RISC-Linz Technical Report*, pages 19–24, 2010.