Computing N-th Roots using the Babylonian Method*

René Thiemann

March 19, 2025

Abstract

We implement the Babylonian method [1] to compute n-th roots of numbers. We provide precise algorithms for naturals, integers and rationals, and offer an approximation algorithm for square roots within linear ordered fields. Moreover, there are precise algorithms to compute the floor and the ceiling of n-th roots.

Contents

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

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

theory *Sqrt-Babylonian-Auxiliary*
imports
  Complex-Main
begin

lemma *mod-div-equality-int*: $(n :: int)$ div $x * x = n - n$ mod x
  using *div-mult-mod-eq*[*of n x*] **by** *arith*

lemma *div-is-floor-divide-rat*: n div $y = \lfloor rat\text{-}of\text{-}int\ n\ /\ rat\text{-}of\text{-}int\ y \rfloor$
  unfolding *Fract-of-int-quotient*[*symmetric*] *floor-Fract* **by** *simp*

lemma *div-is-floor-divide-real*: n div $y = \lfloor real\text{-}of\text{-}int\ n\ /\ of\text{-}int\ y \rfloor$
  unfolding *div-is-floor-divide-rat*[*of n y*]
  by (*metis Ratreal-def of-rat-divide of-rat-of-int-eq real-floor-code*)

lemma *floor-div-pos-int*:
  fixes $r :: 'a :: floor\text{-}ceiling$
  assumes n: $n > 0$
  shows $\lfloor r\ /\ of\text{-}int\ n \rfloor = \lfloor r \rfloor$ div n (**is** $?l = ?r$)
proof −
  let $?of\text{-}int = of\text{-}int :: int \Rightarrow 'a$
  define *rhs* **where** $rhs = \lfloor r \rfloor$ div n
  let $?n = ?of\text{-}int\ n$
  define *m* **where** $m = \lfloor r \rfloor$ mod n
  let $?m = ?of\text{-}int\ m$
  from *div-mult-mod-eq*[*of floor r n*] **have** *dm*: $rhs * n + m = \lfloor r \rfloor$ **unfolding** *rhs-def m-def* **by** *simp*
  have *mn*: $m < n$ **and** *m0*: $m \geq 0$ **using** *n m-def* **by** *auto*
  define *e* **where** $e = r - ?of\text{-}int\ \lfloor r \rfloor$
  have *e0*: $e \geq 0$ **unfolding** *e-def*
    by (*metis diff-self eq-iff floor-diff-of-int zero-le-floor*)
  have *e1*: $e < 1$ **unfolding** *e-def*
    by (*metis diff-self dual-order.refl floor-diff-of-int floor-le-zero*)
  have $r = ?of\text{-}int\ \lfloor r \rfloor + e$ **unfolding** *e-def* **by** *simp*
  also have $\lfloor r \rfloor = rhs * n + m$ **using** *dm* **by** *simp*
  finally have $r = ?of\text{-}int\ (rhs * n + m) + e$ **.**
  hence $r\ /\ ?n = ?of\text{-}int\ (rhs * n)\ /\ ?n + (e + ?m)\ /\ ?n$ **using** *n* **by** (*simp add*: *field-simps*)
  also have $?of\text{-}int\ (rhs * n)\ /\ ?n = ?of\text{-}int\ rhs$ **using** *n* **by** *auto*
  finally have $*$: $r\ /\ ?of\text{-}int\ n = (e + ?of\text{-}int\ m)\ /\ ?of\text{-}int\ n + ?of\text{-}int\ rhs$ **by** *simp*
  have $?l = rhs + floor\ ((e + ?m)\ /\ ?n)$ **unfolding** $*$ **by** *simp*
  also have $floor\ ((e + ?m)\ /\ ?n) = 0$
  proof (*rule floor-unique*)
    show $?of\text{-}int\ 0 \leq (e + ?m)\ /\ ?n$ **using** *e0 m0 n*
    by (*metis add-increasing2 divide-nonneg-pos of-int-0 of-int-0-le-iff of-int-0-less-iff*)

show $(e + ?m) / ?n < ?of\text{-}int\ 0 + 1$
proof (*rule ccontr*)
from *n* **have** n': $?n > 0$ $?n \geq 0$ **by** *simp-all*
assume $\neg$ *?thesis*
hence $(e + ?m) / ?n \geq 1$ **by** *auto*
from *mult-right-mono*[*OF this* $n'(2)$]
have $?n \leq e + ?m$ **using** $n'(1)$ **by** *simp*
also have $?m \leq ?n - 1$ **using** *mn*
by (*metis of-int-1 of-int-diff of-int-le-iff zle-diff1-eq*)
finally have $?n \leq e + ?n - 1$ **by** *auto*
with *e1* **show** *False* **by** *arith*
qed
qed
finally show *?thesis* **unfolding** *rhs-def* **by** *simp*
qed

lemma *floor-div-neg-int*:
fixes $r :: 'a :: floor\text{-}ceiling$
assumes *n*: $n < 0$
shows $\lfloor r / of\text{-}int\ n \rfloor = \lceil r \rceil\ div\ n$
proof −
from *n* **have** n': $-n > 0$ **by** *auto*
have $\lfloor r / of\text{-}int\ n \rfloor = \lfloor -r / of\text{-}int\ (-n) \rfloor$ **using** *n*
by (*metis floor-of-int floor-zero less-int-code*(*1*) *minus-divide-left minus-minus nonzero-minus-divide-right of-int-minus*)
also have $\ldots = \lfloor -r \rfloor\ div\ (-n)$ **by** (*rule floor-div-pos-int*[*OF* n'])
also have $\ldots = \lceil r \rceil\ div\ n$ **using** *n*
by (*metis ceiling-def div-minus-right*)
finally show *?thesis* **.**
qed

lemma *divide-less-floor1*: $n / y < of\text{-}int\ (floor\ (n / y)) + 1$
by (*metis floor-less-iff less-add-one of-int-1 of-int-add*)

context *linordered-idom*
begin

lemma *sgn-int-pow-if* [*simp*]:
$sgn\ x \mathbin{\hat{}} p = (if\ even\ p\ then\ 1\ else\ sgn\ x)$ **if** $x \neq 0$
using *that* **by** (*induct p*) *simp-all*

lemma *compare-pow-le-iff*: $p > 0 \Longrightarrow (x :: 'a) \geq 0 \Longrightarrow y \geq 0 \Longrightarrow (x \mathbin{\hat{}} p \leq y \mathbin{\hat{}} p) = (x \leq y)$
by (*rule power-mono-iff*)

lemma *compare-pow-less-iff*: $p > 0 \Longrightarrow (x :: 'a) \geq 0 \Longrightarrow y \geq 0 \Longrightarrow (x \mathbin{\hat{}} p < y \mathbin{\hat{}} p) = (x < y)$
using *compare-pow-le-iff* [*of p x y*]

using *local.dual-order.order-iff-strict local.power-strict-mono* **by** *blast*

end

lemma *quotient-of-int*[*simp*]: *quotient-of* (*of-int i*) = (*i*,*1*)
by (*metis Rat.of-int-def quotient-of-int*)

lemma *quotient-of-nat*[*simp*]: *quotient-of* (*of-nat i*) = (*int i*,*1*)
by (*metis Rat.of-int-def Rat.quotient-of-int of-int-of-nat-eq*)

lemma *square-lesseq-square*: $\bigwedge$ *x y*. *0* ≤ (*x* :: *'a* :: *linordered-field*) ⟹ *0* ≤ *y* ⟹ (*x* ∗ *x* ≤ *y* ∗ *y*) = (*x* ≤ *y*)
by (*metis mult-mono mult-strict-mono' not-less*)

lemma *square-less-square*: $\bigwedge$ *x y*. *0* ≤ (*x* :: *'a* :: *linordered-field*) ⟹ *0* ≤ *y* ⟹ (*x* ∗ *x* < *y* ∗ *y*) = (*x* < *y*)
by (*metis mult-mono mult-strict-mono' not-less*)

lemma *sqrt-sqrt*[*simp*]: *x* ≥ *0* ⟹ *sqrt x* ∗ *sqrt x* = *x*
by (*metis real-sqrt-pow2 power2-eq-square*)

lemma *abs-lesseq-square*: *abs* (*x* :: *real*) ≤ *abs y* ⟷ *x* ∗ *x* ≤ *y* ∗ *y*
using *square-lesseq-square*[*of abs x abs y*] **by** *auto*

end

2 A Fast Logarithm Algorithm

theory *Log-Impl*
imports
Sqrt-Babylonian-Auxiliary
begin

We implement the discrete logarithm function in a manner similar to a repeated squaring exponentiation algorithm.

In order to prove termination of the algorithm without intermediate checks we need to ensure that we only use proper bases, i.e., values of at least 2. This will be encoded into a separate type.

typedef *proper-base* = {*x* :: *int*. *x* ≥ *2*} **by** *auto*

setup-lifting *type-definition-proper-base*

lift-definition *get-base* :: *proper-base* ⇒ *int* **is** λ *x*. *x* **.**

lift-definition *square-base* :: *proper-base* ⇒ *proper-base* **is** λ *x*. *x* ∗ *x*
proof −
fix *i* :: *int*
assume *i*: *2* ≤ *i*

```
  have 2 ∗ 2 ≤ i ∗ i
    by (rule mult-mono[OF i i], insert i, auto)
  thus 2 ≤ i ∗ i by auto
qed

lift-definition into-base :: int ⇒ proper-base is λ x. if x ≥ 2 then x else 2 by auto

lemma square-base: get-base (square-base b) = get-base b ∗ get-base b
  by (transfer, auto)

lemma get-base-2: get-base b ≥ 2
  by (transfer, auto)

lemma b-less-square-base-b: get-base b < get-base (square-base b)
  unfolding square-base using get-base-2[of b] by simp

lemma b-less-div-base-b: assumes xb: ¬ x < get-base b
  shows x div get-base b < x
proof −
  from get-base-2[of b] have b: get-base b ≥ 2 .
  with xb have x2: x ≥ 2 by auto
  with b int-div-less-self[of x (get-base b)]
  show ?thesis by auto
qed
```

We now state the main algorithm.

```
function log-main :: proper-base ⇒ int ⇒ nat × int where
  log-main b x = (if x < get-base b then (0,1) else
    case log-main (square-base b) x of
      (z, bz) ⇒
    let l = 2 ∗ z; bz1 = bz ∗ get-base b
      in if x < bz1 then (l,bz) else (Suc l,bz1))
  by pat-completeness auto

termination by (relation measure (λ (b,x). nat (1 + x − get-base b)),
  insert b-less-square-base-b, auto)

lemma log-main: x > 0 =⇒ log-main b x = (y,by) =⇒ by = (get-base b)^y ∧
(get-base b)^y ≤ x ∧ x < (get-base b)^(Suc y)
proof (induct b x arbitrary: y by rule: log-main.induct)
  case (1 b x y by)
  note x = 1(2)
  note y = 1(3)
  note IH = 1(1)
  let ?b = get-base b
  show ?case
  proof (cases x < ?b)
    case True
    with x y show ?thesis by auto
```

```
  next
    case False
    obtain z bz where zz: log-main (square-base b) x = (z,bz)
      by (cases log-main (square-base b) x, auto)
   have id: get-base (square-base b) ^ k = ?b ^ (2 * k) for k unfolding square-base
      by (simp add: power-mult semiring-normalization-rules(29))
    from IH[OF False x zz, unfolded id]
    have z: ?b ^ (2 * z) ≤ x x < ?b ^ (2 * Suc z) and bz: bz = get-base b ^ (2 *
z) by auto
    from y[unfolded log-main.simps[of b x] Let-def zz split] bz False
    have yy: (if x < bz * ?b then (2 * z, bz) else (Suc (2 * z), bz * ?b)) =
      (y, by) by auto
    show ?thesis
    proof (cases x < bz * ?b)
      case True
      with yy have yz: y = 2 * z by = bz by auto
      from True z(1) bz show ?thesis unfolding yz by (auto simp: ac-simps)
    next
      case False
      with yy have yz: y = Suc (2 * z) by = ?b * bz by auto
      from False have ?b ^ Suc (2 * z) ≤ x by (auto simp: bz ac-simps)
      with z(2) bz show ?thesis unfolding yz by auto
    qed
  qed
qed
```

We then derive the floor- and ceiling-log functions.

```
definition log-floor :: int ⇒ int ⇒ nat where
  log-floor b x = fst (log-main (into-base b) x)

definition log-ceiling :: int ⇒ int ⇒ nat where
  log-ceiling b x = (case log-main (into-base b) x of
     (y,by) ⇒ if x = by then y else Suc y)

lemma log-floor-sound: assumes b > 1 x > 0 log-floor b x = y
  shows b^y ≤ x x < b^(Suc y)
proof −
  from assms(1,3) have id: get-base (into-base b) = b by transfer auto
  obtain yy bb where log: log-main (into-base b) x = (yy,bb)
    by (cases log-main (into-base b) x, auto)
  from log-main[OF assms(2) log] assms(3)[unfolded log-floor-def log] id
  show b^y ≤ x x < b^(Suc y) by auto
qed

lemma log-ceiling-sound: assumes b > 1 x > 0 log-ceiling b x = y
  shows x ≤ b^y y ≠ 0 ⟹ b^(y − 1) < x
proof −
  from assms(1,3) have id: get-base (into-base b) = b by transfer auto
  obtain yy bb where log: log-main (into-base b) x = (yy,bb)
```

by (*cases log-main* (*into-base b*) *x*, *auto*)
from *log-main*[*OF assms*(*2*) *log*, *unfolded id*] *assms*(*3*)[*unfolded log-ceiling-def log split*]
have *bnd*: $b \,\hat{}\, yy \leq x$ $x < b \,\hat{}\, Suc\ yy$ **and**
y: $y = (if\ x = b \,\hat{}\, yy\ then\ yy\ else\ Suc\ yy)$ **by** *auto*
have $x \leq b\hat{}y \wedge (y \neq 0 \longrightarrow b\hat{}(y - 1) < x)$
proof (*cases* $x = b \,\hat{}\, yy$)
case *True*
with *y bnd assms*(*1*) **show** *?thesis* **by** (*cases yy*, *auto*)
next
case *False*
with *y bnd* **show** *?thesis* **by** *auto*
qed
thus $x \leq b\hat{}y$ $y \neq 0 \Longrightarrow b\hat{}(y - 1) < x$ **by** *auto*
qed

Finally, we connect it to the *log* function working on real numbers.

lemma *log-floor*[*simp*]: **assumes** *b*: $b > 1$ **and** *x*: $x > 0$
shows *log-floor* $b\ x = \lfloor log\ b\ x \rfloor$
proof −
obtain *y* **where** *y*: *log-floor* $b\ x = y$ **by** *auto*
note *main* = *log-floor-sound*[*OF assms y*]
from *b x* **have** ∗: $1 <$ *real-of-int* b $0 <$ *real-of-int* $(b \,\hat{}\, y)$ $0 <$ *real-of-int* x
and ∗∗: $1 <$ *real-of-int* b $0 <$ *real-of-int* x $0 <$ *real-of-int* $(b \,\hat{}\, Suc\ y)$
by *auto*
show *?thesis* **unfolding** *y*
proof (*rule sym*, *rule floor-unique*)
show *real-of-int* (*int y*) $\leq$ *log* (*real-of-int b*) (*real-of-int x*)
using *main*(*1*)[*folded log-le-cancel-iff*[*OF* ∗, *unfolded of-int-le-iff*]]
using *log-pow-cancel*[*of b y*] *b* **by** *auto*
show *log* (*real-of-int b*) (*real-of-int x*) $<$ *real-of-int* (*int y*) + *1*
using *main*(*2*)[*folded log-less-cancel-iff*[*OF* ∗∗, *unfolded of-int-less-iff*]]
using *log-pow-cancel*[*of b Suc y*] *b* **by** *auto*
qed
qed

lemma *log-ceiling*[*simp*]: **assumes** *b*: $b > 1$ **and** *x*: $x > 0$
shows *log-ceiling* $b\ x = \lceil log\ b\ x \rceil$
proof −
obtain *y* **where** *y*: *log-ceiling* $b\ x = y$ **by** *auto*
note *main* = *log-ceiling-sound*[*OF assms y*]
from *b x* **have** ∗: $1 <$ *real-of-int* b $0 <$ *real-of-int* $(b \,\hat{}\, (y - 1))$ $0 <$ *real-of-int* x
and ∗∗: $1 <$ *real-of-int* b $0 <$ *real-of-int* x $0 <$ *real-of-int* $(b \,\hat{}\, y)$
by *auto*
show *?thesis* **unfolding** *y*
proof (*rule sym*, *rule ceiling-unique*)
show *log* (*real-of-int b*) (*real-of-int x*) $\leq$ *real-of-int* (*int y*)
using *main*(*1*)[*folded log-le-cancel-iff*[*OF* ∗∗, *unfolded of-int-le-iff*]]

```
    using log-pow-cancel[of b y] b by auto
  from x have x: x ≥ 1 by auto
  show real-of-int (int y) − 1 < log (real-of-int b) (real-of-int x)
  proof (cases y = 0)
    case False
    thus ?thesis
      using main(2)[folded log-less-cancel-iff[OF ∗, unfolded of-int-less-iff]]
      using log-pow-cancel[of b y − 1] b x by auto
  next
    case True
    have real-of-int (int y) − 1 = log b (1/b) using True b
      by (subst log-divide, auto)
    also have . . . < log b 1
      by (subst log-less-cancel-iff, insert b, auto)
    also have . . . ≤ log b x
      by (subst log-le-cancel-iff, insert b x, auto)
    finally show real-of-int (int y) − 1 < log (real-of-int b) (real-of-int x) .
  qed
 qed
qed

end
```

3 Executable algorithms for p-th roots

```
theory NthRoot-Impl
imports
  Log-Impl
  Cauchy.CauchysMeanTheorem
begin
```

We implemented algorithms to decide $\sqrt[p]{n} \in \mathbb{Q}$ and to compute $\lfloor \sqrt[p]{n} \rfloor$. To this end, we use a variant of Newton iteration which works with integer division instead of floating point or rational division. To get suitable starting values for the Newton iteration, we also implemented a function to approximate logarithms.

3.1 Logarithm

For computing the p-th root of a number n, we must choose a starting value in the iteration. Here, we use $(2::'a)^{nat\ \lceil of\text{-}int\ \lceil log\ 2\ n\rceil\ /\ p\rceil}$.

We use a partial efficient algorithm, which does not terminate on corner-cases, like $b = 0$ or $p = 1$, and invoke it properly afterwards. Then there is a second algorithm which terminates on these corner-cases by additional guards and on which we can perform induction.

3.2 Computing the p-th root of an integer number

Using the logarithm, we can define an executable version of the intended starting value. Its main property is the inequality $x \leq (\textit{start-value } x\ p)^p$, i.e., the start value is larger than the p-th root. This property is essential, since our algorithm will abort as soon as we fall below the p-th root.

definition *start-value* :: *int* ⇒ *nat* ⇒ *int* **where**
  start-value n p = *2* ^(*nat* ⌈*of-nat* (*log-ceiling 2 n*) / *rat-of-nat p*⌉)

lemma *start-value-main*: **assumes** *x*: *x* ≥ *0* **and** *p*: *p* > *0*
  shows *x* ≤ (*start-value x p*)^*p* ∧ *start-value x p* ≥ *0*
proof (*cases x* = *0*)
  case *True*
  with *p* **show** *?thesis* **unfolding** *start-value-def True* **by** *simp*
next
  case *False*
  with *x* **have** *x*: *x* > *0* **by** *auto*
  define *l2x* **where** *l2x* = ⌈*log 2 x*⌉
  define *pow* **where** *pow* = *nat* ⌈*rat-of-int l2x* / *of-nat p*⌉
  have *root p x* = *x powr* (*1* / *p*) **by** (*rule root-powr-inverse*, *insert x p*, *auto*)
  also have ... = (*2 powr* (*log 2 x*)) *powr* (*1* / *p*) **using** *powr-log-cancel*[*of 2 x*] *x*
by *auto*
  also have ... = *2 powr* (*log 2 x* ∗ (*1* / *p*)) **by** (*rule powr-powr*)
  also have *log 2 x* ∗ (*1* / *p*) = *log 2 x* / *p* **using** *p* **by** *auto*
  finally have *r*: *root p x* = *2 powr* (*log 2 x* / *p*) **.**
  have *lp*: *log 2 x* ≥ *0* **using** *x* **by** *auto*
  hence *l2pos*: *l2x* ≥ *0* **by** (*auto simp*: *l2x-def*)
  have *log 2 x* / *p* ≤ *l2x* / *p* **using** *x p* **unfolding** *l2x-def*
    by (*metis divide-right-mono le-of-int-ceiling of-nat-0-le-iff*)
  also have ... ≤ ⌈*l2x* / (*p* :: *real*)⌉ **by** (*simp add*: *ceiling-correct*)
  also have *l2x* / *real p* = *l2x* / *real-of-rat* (*of-nat p*)
    by (*metis of-rat-of-nat-eq*)
  also have *of-int l2x* = *real-of-rat* (*of-int l2x*)
    by (*metis of-rat-of-int-eq*)
  also have *real-of-rat* (*of-int l2x*) / *real-of-rat* (*of-nat p*) = *real-of-rat* (*rat-of-int l2x* / *of-nat p*)
    by (*metis of-rat-divide*)
  also have ⌈*real-of-rat* (*rat-of-int l2x* / *rat-of-nat p*)⌉ = ⌈*rat-of-int l2x* / *of-nat p*⌉
    by *simp*
  also have ⌈*rat-of-int l2x* / *of-nat p*⌉ ≤ *real pow* **unfolding** *pow-def* **by** *auto*
  finally have *le*: *log 2 x* / *p* ≤ *pow* **.**
  from *powr-mono*[*OF le*, *of 2*, *folded r*]
  have *root p x* ≤ *2 powr pow* **by** *auto*
  also have ... = *2* ^ *pow* **by** (*rule powr-realpow*, *auto*)
  also have ... = *of-int* ((*2* :: *int*) ^ *pow*) **by** *simp*
  also have *pow* = (*nat* ⌈*of-int* (*log-ceiling 2 x*) / *rat-of-nat p*⌉)
    unfolding *pow-def l2x-def* **using** *x* **by** *simp*
  also have *real-of-int* ((*2* :: *int*) ^ ...) = *start-value x p* **unfolding** *start-value-def*
by *simp*

finally have *less*: *root p x* $\leq$ *start-value x p* **.**
have *0* $\leq$ *root p x* **using** *p x* **by** *auto*
also have ... $\leq$ *start-value x p* **by** (*rule less*)
finally have *start*: *0* $\leq$ *start-value x p* **by** *simp*
from *power-mono*[*OF less*, *of p*] **have** *root p* (*of-int x*) ^ *p* $\leq$ *of-int* (*start-value x p*) ^ *p* **using** *p x* **by** *auto*
also have ... = *start-value x p* ^ *p* **by** *simp*
also have *root p* (*of-int x*) ^ *p* = *x* **using** *p x* **by** *force*
finally have *x* $\leq$ (*start-value x p*) ^ *p* **by** *presburger*
with *start* **show** *?thesis* **by** *auto*
qed

lemma *start-value*: **assumes** *x*: *x* $\geq$ *0* **and** *p*: *p* > *0* **shows** *x* $\leq$ (*start-value x p*) ^ *p start-value x p* $\geq$ *0*
using *start-value-main*[*OF x p*] **by** *auto*

We now define the Newton iteration to compute the p-th root. We are working on the integers, where every (/) is replaced by (*div*). We are proving several things within a locale which ensures that $p > 0$, and where $pm = p - 1$.

locale *fixed-root* =
fixes *p pm* :: *nat*
assumes *p*: *p* = *Suc pm*
begin

function *root-newton-int-main* :: *int* ⇒ *int* ⇒ *int* × *bool* **where**
root-newton-int-main x n = (*if* (*x* < *0* ∨ *n* < *0*) *then* (*0*,*False*) *else* (*if x* ^ *p* ≤ *n then* (*x*, *x* ^ *p* = *n*)
else root-newton-int-main ((*n div* (*x* ^ *pm*) + *x* ∗ *int pm*) *div* (*int p*)) *n*))
by *pat-completeness auto*
end

For the executable algorithm we omit the guard and use a let-construction

partial-function (*tailrec*) *root-int-main′* :: *nat* ⇒ *int* ⇒ *int* ⇒ *int* ⇒ *int* ⇒ *int* × *bool* **where**
[*code*]: *root-int-main′ pm ipm ip x n* = (*let xpm* = *x*^*pm*; *xp* = *xpm* ∗ *x in if xp* ≤ *n then* (*x*, *xp* = *n*)
else root-int-main′ pm ipm ip ((*n div xpm* + *x* ∗ *ipm*) *div ip*) *n*)

In the following algorithm, we start the iteration. It will compute $\lfloor$*root p n*$\rfloor$ and a boolean to indicate whether the root is exact.

definition *root-int-main* :: *nat* ⇒ *int* ⇒ *int* × *bool* **where**
root-int-main p n ≡ *if p* = *0 then* (*1*,*n* = *1*) *else*
let pm = *p* − *1*
in root-int-main′ pm (*int pm*) (*int p*) (*start-value n p*) *n*

Once we have proven soundness of *fixed-root.root-newton-int-main* and equivalence to *root-int-main*, it is easy to assemble the following algorithm which computes all roots for arbitrary integers.

definition *root-int* :: *nat* ⇒ *int* ⇒ *int list* **where**
root-int p x ≡ *if p* = *0 then* [] *else*
if x = *0 then* [*0*] *else*
let e = *even p*; *s* = *sgn x*; *x'* = *abs x*
in if x < *0* ∧ *e then* [] *else case root-int-main p x' of* (*y*,*True*) ⇒ *if e then* [*y*,−*y*] *else* [*s* ∗ *y*] | - ⇒ []

We start with proving termination of *fixed-root.root-newton-int-main*.

context *fixed-root*
begin
lemma *iteration-mono-eq*: **assumes** *xn*: *x* ^ *p* = (*n* :: *int*)
shows (*n div x* ^ *pm* + *x* ∗ *int pm*) *div int p* = *x*
proof −
have [*simp*]: ⋀ *n*. (*x* + *x* ∗ *n*) = *x* ∗ (*1* + *n*) **by** (*auto simp*: *field-simps*)
show *?thesis* **unfolding** *xn*[*symmetric*] *p* **by** *simp*
qed

lemma *p0*: *p* ≠ *0* **unfolding** *p* **by** *auto*

The following property is the essential property for proving termination of *root-newton-int-main*.

lemma *iteration-mono-less*: **assumes** *x*: *x* ≥ *0*
and *n*: *n* ≥ *0*
and *xn*: *x* ^ *p* > (*n* :: *int*)
shows (*n div x* ^ *pm* + *x* ∗ *int pm*) *div int p* < *x*
proof −
let *?sx* = (*n div x* ^ *pm* + *x* ∗ *int pm*) *div int p*
from *xn* **have** *xn-le*: *x* ^ *p* ≥ *n* **by** *auto*
from *xn x n* **have** *x0*: *x* > *0*
using *not-le p* **by** *fastforce*
from *p* **have** *xp*: *x* ^ *p* = *x* ∗ *x* ^ *pm* **by** *auto*
from *x n* **have** *n div x* ^ *pm* ∗ *x* ^ *pm* ≤ *n*
by (*auto simp add*: *minus-mod-eq-div-mult* [*symmetric*] *mod-int-pos-iff not-less power-le-zero-eq*)
also have . . . ≤ *x* ^ *p* **using** *xn* **by** *auto*
finally have *le*: *n div x* ^ *pm* ≤ *x* **using** *x x0* **unfolding** *xp* **by** *simp*
have *?sx* ≤ (*x*^*p div x* ^ *pm* + *x* ∗ *int pm*) *div int p*
by (*rule zdiv-mono1*, *insert le p0*, *unfold xp*, *auto*)
also have *x*^*p div x* ^ *pm* = *x* **unfolding** *xp* **by** *auto*
also have *x* + *x* ∗ *int pm* = *x* ∗ *int p* **unfolding** *p* **by** (*auto simp*: *field-simps*)
also have *x* ∗ *int p div int p* = *x* **using** *p* **by** *force*
finally have *le*: *?sx* ≤ *x* **.**
{
assume *?sx* = *x*
from *arg-cong*[*OF this*, *of* λ *x*. *x* ∗ *int p*]
have *x* ∗ *int p* ≤ (*n div x* ^ *pm* + *x* ∗ *int pm*) *div* (*int p*) ∗ *int p* **using** *p0* **by** *simp*
also have . . . ≤ *n div x* ^ *pm* + *x* ∗ *int pm*
unfolding *mod-div-equality-int* **using** *p* **by** *auto*

```
    finally have n div x^pm ≥ x by (auto simp: p field-simps)
    from mult-right-mono[OF this, of x ^ pm]
    have ge: n div x^pm * x^pm ≥ x^p unfolding xp using x by auto
   from div-mult-mod-eq[of n x^pm] have n div x^pm * x^pm = n − n mod x^pm
by arith
    from ge[unfolded this]
    have le: x^p ≤ n − n mod x^pm .
    from x n have ge: n mod x ^ pm ≥ 0
      by (auto simp add: mod-int-pos-iff not-less power-le-zero-eq)
    from le ge
    have n ≥ x^p by auto
    with xn have False by auto
  }
  with le show ?thesis unfolding p by fastforce
qed

lemma iteration-mono-lesseq: assumes x: x ≥ 0 and n: n ≥ 0 and xn: x ^ p ≥
(n :: int)
  shows (n div x ^ pm + x * int pm) div int p ≤ x
proof (cases x ^ p = n)
  case True
  from iteration-mono-eq[OF this] show ?thesis by simp
next
  case False
  with assms have x ^ p > n by auto
  from iteration-mono-less[OF x n this]
  show ?thesis by simp
qed

termination
proof −
  let ?mm = λ x  n :: int. nat x
  let ?m1 = λ (x,n). ?mm x n
  let ?m = measures [?m1]
  show ?thesis
  proof (relation ?m)
    fix x n :: int
    assume ¬ x ^ p ≤ n
    hence x: x ^ p > n by auto
    assume ¬ (x < 0 ∨ n < 0)
    hence x-n: x ≥ 0 n ≥ 0 by auto
    from x x-n have x0: x > 0 using p by (cases x = 0, auto)
    from iteration-mono-less[OF x-n x] x0
    show (((n div x ^ pm + x * int pm) div int p, n), x, n) ∈ ?m by auto
  qed auto
qed
```

We next prove that *root-int-main′* is a correct implementation of *root-newton-int-main*. We additionally prove that the result is always positive, a lower bound, and that the returned boolean indicates whether the result has a root or not. We prove all these results in one go, so that we can share the inductive proof.

abbreviation *root-main′* **where** *root-main′* ≡ *root-int-main′ pm* (*int pm*) (*int p*)

lemmas *root-main′-simps* = *root-int-main′.simps*[*of pm int pm int p*]

lemma *root-main′-newton-pos*: $x \geq 0 \Longrightarrow n \geq 0 \Longrightarrow$
  root-main′ x n = *root-newton-int-main* x n ∧ (*root-main′* x n = $(y,b) \longrightarrow y \geq 0$ ∧ $y\hat{\ }p \leq n$ ∧ $b = (y\hat{\ }p = n)$)
proof (*induct x n rule*: *root-newton-int-main.induct*)
  case (*1 x n*)
  have *pm-x*[*simp*]: $x \hat{\ } pm * x = x \hat{\ } p$ **unfolding** *p* **by** *simp*
  from *1* **have** *id*: $(x < 0 \vee n < 0)$ = *False* **by** *auto*
  note *d* = *root-main′-simps*[*of x n*] *root-newton-int-main.simps*[*of x n*] *id if-False Let-def*
  show *?case*
  proof (*cases* $x \hat{\ } p \leq n$)
    case *True*
    thus *?thesis* **unfolding** *d* **using** *1*(*2*) **by** *auto*
  next
    case *False*
    hence *id*: $(x \hat{\ } p \leq n)$ = *False* **by** *simp*
    from *1*(*3*) *1*(*2*) **have** *not*: $\neg (x < 0 \vee n < 0)$ **by** *auto*
    then have *x*: $x > 0 \vee x = 0$
      by *auto*
    with ‹$0 \leq n$› **have** $0 \leq$ (*n div x* ^ *pm* + *x* ∗ *int pm*) *div int p*
      by (*auto simp add*: *p algebra-simps pos-imp-zdiv-nonneg-iff power-0-left*)
    then show *?thesis* **unfolding** *d id pm-x*
      by (*rule 1*(*1*)[*OF not False - 1*(*3*)])
  qed
qed

lemma *root-main′*: $x \geq 0 \Longrightarrow n \geq 0 \Longrightarrow$ *root-main′* x n = *root-newton-int-main x n*
  using *root-main′-newton-pos* **by** *blast*

lemma *root-main′-pos*: $x \geq 0 \Longrightarrow n \geq 0 \Longrightarrow$ *root-main′* x n = $(y,b) \Longrightarrow y \geq 0$
  using *root-main′-newton-pos* **by** *blast*

lemma *root-main′-sound*: $x \geq 0 \Longrightarrow n \geq 0 \Longrightarrow$ *root-main′* x n = $(y,b) \Longrightarrow b = (y \hat{\ } p = n)$
  using *root-main′-newton-pos* **by** *blast*

In order to prove completeness of the algorithms, we provide sharp upper and lower bounds for *root-main′*. For the upper bounds, we use Cauchy's mean theorem where we added the non-strict variant to Porter's formalization of this theorem.

lemma *root-main′-lower*: $x \geq 0 \Longrightarrow n \geq 0 \Longrightarrow$ *root-main′* x n = $(y,b) \Longrightarrow y \hat{\ } p \leq n$
  using *root-main′-newton-pos* **by** *blast*

```
lemma root-newton-int-main-upper:
  shows y ^ p ≥ n ⟹ y ≥ 0 ⟹ n ≥ 0 ⟹ root-newton-int-main y n = (x,b)
⟹ n < (x + 1) ^ p
proof (induct y n rule: root-newton-int-main.induct)
  case (1 y n)
  from 1(3) have y0: y ≥ 0 .
  then have y > 0 ∨ y = 0
    by auto
  from 1(4) have n0: n ≥ 0 .
  define y' where y' = (n div (y ^ pm) + y * int pm) div (int p)
  from ‹y > 0 ∨ y = 0› ‹n ≥ 0› have y'0: y' ≥ 0
    by (auto simp add: y'-def p algebra-simps pos-imp-zdiv-nonneg-iff power-0-left)
  let ?rt = root-newton-int-main
  from 1(5) have rt: ?rt y n = (x,b) by auto
  from y0 n0 have not: ¬ (y < 0 ∨ n < 0) (y < 0 ∨ n < 0) = False by auto
  note rt = rt[unfolded root-newton-int-main.simps[of y n] not(2) if-False, folded
y'-def]
  note IH = 1(1)[folded y'-def, OF not(1) - - y'0 n0]
  show ?case
  proof (cases y ^ p ≤ n)
    case False note yyn = this
    with rt have rt: ?rt y' n = (x,b) by simp
    show ?thesis
    proof (cases n ≤ y' ^ p)
      case True
      show ?thesis
        by (rule IH[OF False True rt])
    next
      case False
      with rt have x: x = y' unfolding root-newton-int-main.simps[of y' n]
        using n0 y'0 by simp
      from yyn have yyn: y^p > n by simp
      from False have yyn': n > y' ^ p by auto
      {
        assume pm: pm = 0
        have y': y' = n unfolding y'-def p pm by simp
        with yyn' have False unfolding p pm by auto
      }
      hence pm0: pm > 0 by auto
      show ?thesis
      proof (cases n = 0)
        case True
        thus ?thesis unfolding p
          by (metis False y'0 zero-le-power)
      next
        case False note n00 = this
        let ?y = of-int y :: real
        let ?n = of-int n :: real
```

```
from yyn n0 have y00: y ≠ 0 unfolding p by auto
from y00 y0 have y0: ?y > 0 by auto
from n0 False have n0: ?n > 0 by auto
define Y where Y = ?y ∗ of-int pm
define NY where NY = ?n / ?y ^ pm
note pos-intro = divide-nonneg-pos add-nonneg-nonneg mult-nonneg-nonneg
have NY0: NY > 0 unfolding NY-def using y0 n0
  by (metis NY-def zero-less-divide-iff zero-less-power)
let ?ls = NY # replicate pm ?y
have prod: ∏:replicate pm ?y = ?y ^ pm
  by (induct pm, auto)
have sum: ∑:replicate pm ?y = Y unfolding Y-def
  by (induct pm, auto simp: field-simps)
have pos: pos ?ls unfolding pos-def using NY0 y0 by auto
have root p ?n = gmean ?ls unfolding gmean-def using y0
  by (auto simp: p NY-def prod)
also have ... < mean ?ls
proof (rule CauchysMeanTheorem-Less[OF pos het-gt-0I])
  show NY ∈ set ?ls by simp
  from pm0 show ?y ∈ set ?ls by simp
  have NY < ?y
  proof −
    from yyn have less: ?n < ?y ^ Suc pm unfolding p
      by (metis of-int-less-iff of-int-power)
    have NY < ?y ^ Suc pm / ?y ^ pm unfolding NY-def
      by (rule divide-strict-right-mono[OF less], insert y0, auto)
    thus ?thesis using y0 by auto
  qed
  thus NY ≠ ?y by blast
qed
also have ... = (NY + Y) / real p
  by (simp add: mean-def sum p)
finally have ∗: root p ?n < (NY + Y) / real p .
have ?n = (root p ?n)^p using n0
  by (metis neq0-conv p0 real-root-pow-pos)
also have ... < ((NY + Y) / real p)^p
  by (rule power-strict-mono[OF ∗], insert n0 p, auto)
finally have ineq1: ?n < ((NY + Y) / real p)^p by auto
{
  define s where s = n div y ^ pm + y ∗ int pm
  define S where S = NY + Y
  have Y0: Y ≥ 0 using y0 unfolding Y-def
    by (metis 1.prems(2) mult-nonneg-nonneg of-int-0-le-iff of-nat-0-le-iff)
  have S0: S > 0 using NY0 Y0 unfolding S-def by auto
  from p have p0: p > 0 by auto
  have ?n / ?y ^ pm  < of-int (floor (?n / ?y^pm)) + 1
    by (rule divide-less-floor1)
  also have floor (?n / ?y ^ pm) = n div y^pm
    unfolding div-is-floor-divide-real by (metis of-int-power)
```

```
        finally have NY < of-int (n div y ^ pm) + 1 unfolding NY-def by simp
         hence less: S < of-int s + 1 unfolding Y-def s-def S-def by simp
         {
           have f1: ∀x0. rat-of-int ⌊rat-of-nat x0⌋ = rat-of-nat x0
             using of-int-of-nat-eq by simp
           have f2: ∀x0. real-of-int ⌊rat-of-nat x0⌋ = real x0
             using of-int-of-nat-eq by auto
              have f3: ∀x0 x1. ⌊rat-of-int x0 / rat-of-int x1⌋ = ⌊real-of-int x0 /
real-of-int x1⌋
             using div-is-floor-divide-rat div-is-floor-divide-real by simp
           have f4: 0 < ⌊rat-of-nat p⌋
             using p by simp
           have ⌊S⌋ ≤ s using less floor-le-iff by auto
           hence ⌊rat-of-int ⌊S⌋ / rat-of-nat p⌋ ≤ ⌊rat-of-int s / rat-of-nat p⌋
             using f1 f3 f4 by (metis div-is-floor-divide-real zdiv-mono1)
           hence ⌊S / real p⌋ ≤ ⌊rat-of-int s / rat-of-nat p⌋
             using f1 f2 f3 f4 by (metis div-is-floor-divide-real floor-div-pos-int)
           hence S / real p ≤ real-of-int (s div int p) + 1
               using f1 f3 by (metis div-is-floor-divide-real floor-le-iff floor-of-nat
less-eq-real-def)
         }
         hence S / real p ≤ of-int(s div p) + 1 .
         note this[unfolded S-def s-def]
       }
       hence ge: of-int y' + 1 ≥ (NY + Y) / p unfolding y'-def
         by simp
       have pos1: (NY + Y) / p ≥ 0 unfolding Y-def NY-def
         by (intro divide-nonneg-pos add-nonneg-nonneg mult-nonneg-nonneg,
         insert y0 n0 p0, auto)
       have pos2: of-int y' + (1 :: rat) ≥ 0 using y'0 by auto
       have ineq2: (of-int y' + 1) ^ p ≥ ((NY + Y) / p) ^ p
         by (rule power-mono[OF ge pos1])
       from order.strict-trans2[OF ineq1 ineq2]
       have ?n < of-int ((x + 1) ^ p) unfolding x
         by (metis of-int-1 of-int-add of-int-power)
       thus n < (x + 1) ^ p using of-int-less-iff by blast
     qed
   qed
  next
   case True
   with rt have x: x = y by simp
   with 1(2) True have n: n = y ^ p by auto
   show ?thesis unfolding n x using y0 unfolding p
    by (metis add-le-less-mono add-less-cancel-left lessI less-add-one add.right-neutral
le-iff-add power-strict-mono)
  qed
qed

lemma root-main'-upper:
```

$x \hat{} p \geq n \Longrightarrow x \geq 0 \Longrightarrow n \geq 0 \Longrightarrow$ *root-main'* $x\ n = (y,b) \Longrightarrow n < (y + 1) \hat{}$ p
using *root-newton-int-main-upper*[*of n x y b*] *root-main'*[*of x n*] **by** *auto*
end

Now we can prove all the nice properties of *root-int-main*.

lemma *root-int-main-all*: **assumes** *n*: $n \geq 0$
and *rm*: *root-int-main* $p\ n = (y,b)$
shows $y \geq 0 \wedge b = (y \hat{} p = n) \wedge (p > 0 \longrightarrow y \hat{} p \leq n \wedge n < (y + 1)\hat{}p)$
$\wedge (p > 0 \longrightarrow x \geq 0 \longrightarrow x \hat{} p = n \longrightarrow y = x \wedge b)$
proof (*cases* $p = 0$)
case *True*
with *rm*[*unfolded root-int-main-def*]
have *y*: $y = 1$ **and** *b*: $b = (n = 1)$ **by** *auto*
show *?thesis* **unfolding** *True y b* **using** *n* **by** *auto*
next
case *False*
from *False* **have** *p-0*: $p > 0$ **by** *auto*
from *False* **have** $(p = 0) = False$ **by** *simp*
from *rm*[*unfolded root-int-main-def this Let-def*]
have *rm*: *root-int-main'* $(p - 1)$ (*int* $(p - 1)$) (*int* p) (*start-value* $n\ p$) $n = (y,b)$ **by** *simp*
from *start-value*[*OF n p-0*] **have** *start*: $n \leq$ (*start-value* $n\ p$)$\hat{}p$ $0 \leq$ *start-value* $n\ p$ **by** *auto*
interpret *fixed-root* $p\ p - 1$
by (*unfold-locales*, *insert False*, *auto*)
from *root-main'-pos*[*OF start(2) n rm*] **have** *y*: $y \geq 0$ **.**
from *root-main'-sound*[*OF start(2) n rm*] **have** *b*: $b = (y \hat{} p = n)$ **.**
from *root-main'-lower*[*OF start(2) n rm*] **have** *low*: $y \hat{} p \leq n$ **.**
from *root-main'-upper*[*OF start n rm*] **have** *up*: $n < (y + 1) \hat{} p$ **.**
{
assume *n*: $x \hat{} p = n$ **and** *x*: $x \geq 0$
with *low up* **have** *low*: $y \hat{} p \leq x \hat{} p$ **and** *up*: $x \hat{} p < (y+1) \hat{} p$ **by** *auto*
from *power-strict-mono*[*of x y, OF - x p-0*] *low* **have** *x*: $x \geq y$ **by** *arith*
from *power-mono*[*of* $(y + 1)$ *x p*] *y up* **have** *y*: $y \geq x$ **by** *arith*
from *x y* **have** $x = y$ **by** *auto*
with *b n*
have $y = x \wedge b$ **by** *auto*
}
thus *?thesis* **using** *b low up y* **by** *auto*
qed

lemma *root-int-main*: **assumes** *n*: $n \geq 0$
and *rm*: *root-int-main* $p\ n = (y,b)$
shows $y \geq 0$ $b = (y \hat{} p = n)$ $p > 0 \Longrightarrow y \hat{} p \leq n$ $p > 0 \Longrightarrow n < (y + 1)\hat{}p$
$p > 0 \Longrightarrow x \geq 0 \Longrightarrow x \hat{} p = n \Longrightarrow y = x \wedge b$
using *root-int-main-all*[*OF n rm, of x*] **by** *blast+*

lemma *root-int*[*simp*]: **assumes** *p*: $p \neq 0 \vee x \neq 1$

```
  shows set (root-int p x) = {y . y ^ p = x}
proof (cases p = 0)
  case True
  with p have x ≠ 1 by auto
  thus ?thesis unfolding root-int-def True by auto
next
  case False
  hence p: (p = 0) = False and p0: p > 0 by auto
  note d = root-int-def p if-False Let-def
  show ?thesis
  proof (cases x = 0)
    case True
    thus ?thesis unfolding d using p0 by auto
  next
    case False
    hence x: (x = 0) = False by auto
    show ?thesis
    proof (cases x < 0 ∧ even p)
      case True
      hence left: set (root-int p x) = {} unfolding d by auto
      {
        fix y
        assume x: y ^ p = x
        with True have y ^ p < 0 ∧ even p by auto
        hence False by presburger
      }
      with left show ?thesis by auto
    next
      case False
      with x p have cond: (x = 0) = False (x < 0 ∧ even p) = False by auto
      obtain y b where rt: root-int-main p |x| = (y,b) by force
      have abs x ≥ 0 by auto
      note rm = root-int-main[OF this rt]
      have ?thesis =
          (set (case root-int-main p |x| of (y, True) ⇒ if even p then [y, − y] else
[sgn x ∗ y] | (y, False) ⇒ []) =
        {y. y ^ p = x}) unfolding d cond by blast
       also have (case root-int-main p |x| of (y, True) ⇒ if even p then [y, − y]
else [sgn x ∗ y] | (y, False) ⇒ [])
        = (if b then if even p then [y, − y] else [sgn x ∗ y] else []) (is - = ?lhs)
        unfolding rt by auto
      also have set ?lhs = {y. y ^ p = x} (is - = ?rhs)
      proof −
        {
          fix z
          assume idx: z ^ p = x
          hence eq: (abs z) ^ p = abs x by (metis power-abs)
          from idx x p0 have z: z ≠ 0 unfolding p by auto
          have (y, b) = (|z|, True)
```

```
          using rm(5)[OF p0 - eq] by auto
        hence id: y = abs z b = True by auto
         have z ∈ set ?lhs unfolding id using z by (auto simp: idx[symmetric],
cases z < 0, auto)
      }
      moreover
      {
        fix z
        assume z: z ∈ set ?lhs
        hence b: b = True by (cases b, auto)
        note z = z[unfolded b if-True]
        from rm(2) b have yx: y ^ p = |x| by auto
        from rm(1) have y: y ≥ 0 .
        from False have odd p ∨ even p ∧ x ≥ 0 by auto
        hence z ∈ ?rhs
        proof
          assume odd: odd p
          with z have z = sgn x ∗ y by auto
          hence z ^ p = (sgn x ∗ y) ^ p by auto
         also have . . . = sgn x ^ p ∗ y ^ p unfolding power-mult-distrib by auto
          also have . . . = sgn x ^ p ∗ abs x unfolding yx by simp
          also have sgn x ^ p = sgn x using x odd by auto
          also have sgn x ∗ abs x = x by (rule mult-sgn-abs)
          finally show z ∈ ?rhs by auto
        next
          assume even: even p ∧ x ≥ 0
          from z even have z = y ∨ z = −y by auto
          hence id: abs z = y using y by auto
          with yx x even have z: z ≠ 0 using p0 by (cases y = 0, auto)
          have z ^ p = (sgn z ∗ abs z) ^ p by (simp add: mult-sgn-abs)
          also have . . . = (sgn z ∗ y) ^ p using id by auto
            also have . . . = (sgn z)^p ∗ y ^ p unfolding power-mult-distrib  by
simp
          also have . . . = sgn z ^ p ∗ x unfolding yx using even by auto
          also have sgn z ^ p = 1 using even z by (auto)
          finally show z ∈ ?rhs by auto
        qed
      }
      ultimately show ?thesis by blast
    qed
    finally show ?thesis by auto
  qed
 qed
qed

lemma root-int-pos: assumes x: x ≥ 0 and ri: root-int p x = y # ys
 shows y ≥ 0
proof −
 from x have abs: abs x = x by auto
```

```
  note ri = ri[unfolded root-int-def Let-def abs]
  from ri have p: (p = 0) = False by (cases p, auto)
  note ri = ri[unfolded p if-False]
  show ?thesis
  proof (cases x = 0)
    case True
    with ri show ?thesis by auto
  next
    case False
    hence (x = 0) = False (x < 0 ∧ even p) = False using x by auto
    note ri = ri[unfolded this if-False]
    obtain y′ b′ where r: root-int-main p x = (y′,b′) by force
    note ri = ri[unfolded this]
    hence y: y = (if even p then y′ else sgn x ∗ y′) by (cases b′, auto)
    from root-int-main(1)[OF x r] have y′: 0 ≤ y′ .
    thus ?thesis unfolding y using x False by auto
  qed
qed
```

3.3 Floor and ceiling of roots

Using the bounds for *root-int-main* we can easily design algorithms which compute $\lfloor root\ p\ x \rfloor$ and $\lceil root\ p\ x \rceil$. To this end, we first develop algorithms for non-negative x, and later on these are used for the general case.

```
definition root-int-floor-pos p x = (if p = 0 then 0 else fst (root-int-main p x))
definition root-int-ceiling-pos p x = (if p = 0 then 0 else (case root-int-main p x
of (y,b) ⇒ if b then y else y + 1))

lemma root-int-floor-pos-lower: assumes p0: p ≠ 0 and x: x ≥ 0
  shows root-int-floor-pos p x ^ p ≤ x
  using root-int-main(3)[OF x, of p] p0 unfolding root-int-floor-pos-def
  by (cases root-int-main p x, auto)

lemma root-int-floor-pos-pos: assumes x: x ≥ 0
  shows root-int-floor-pos p x ≥ 0
  using root-int-main(1)[OF x, of p]
  unfolding root-int-floor-pos-def
  by (cases root-int-main p x, auto)

lemma root-int-floor-pos-upper: assumes p0: p ≠ 0 and x: x ≥ 0
  shows (root-int-floor-pos p x + 1) ^ p > x
  using root-int-main(4)[OF x, of p] p0 unfolding root-int-floor-pos-def
  by (cases root-int-main p x, auto)

lemma root-int-floor-pos: assumes x: x ≥ 0
  shows root-int-floor-pos p x = floor (root p (of-int x))
proof (cases p = 0)
  case True
  thus ?thesis by (simp add: root-int-floor-pos-def)
```

next
  case *False*
  hence *p*: *p* > *0* **by** *auto*
  let *?s1* = *real-of-int* (*root-int-floor-pos p x*)
  let *?s2* = *root p* (*of-int x*)
  from *x* **have** *s1*: *?s1* ≥ *0*
    by (*metis of-int-0-le-iff root-int-floor-pos-pos*)
  from *x* **have** *s2*: *?s2* ≥ *0*
    by (*metis of-int-0-le-iff real-root-pos-pos-le*)
  from *s1* **have** *s11*: *?s1* + *1* ≥ *0* **by** *auto*
  have *id*: *?s2* ^ *p* = *of-int x* **using** *x*
    by (*metis p of-int-0-le-iff real-root-pow-pos2*)
  show *?thesis*
  proof (*rule floor-unique*[*symmetric*])
    show *?s1* ≤ *?s2*
      unfolding *compare-pow-le-iff*[*OF p s1 s2*, *symmetric*]
      unfolding *id*
      using *root-int-floor-pos-lower*[*OF False x*]
      by (*metis of-int-le-iff of-int-power*)
    show *?s2* < *?s1* + *1*
      unfolding *compare-pow-less-iff*[*OF p s2 s11*, *symmetric*]
      unfolding *id*
      using *root-int-floor-pos-upper*[*OF False x*]
      by (*metis of-int-add of-int-less-iff of-int-power of-int-1*)
  qed
qed

lemma *root-int-ceiling-pos*: **assumes** *x*: *x* ≥ *0*
  shows *root-int-ceiling-pos p x* = *ceiling* (*root p* (*of-int x*))
proof (*cases p* = *0*)
  case *True*
  thus *?thesis* **by** (*simp add*: *root-int-ceiling-pos-def*)
next
  case *False*
  hence *p*: *p* > *0* **by** *auto*
  obtain *y b* **where** *s*: *root-int-main p x* = (*y*,*b*) **by** *force*
  note *rm* = *root-int-main*[*OF x s*]
  note *rm* = *rm*(*1*−*2*) *rm*(*3*−*5*)[*OF p*]
  from *rm*(*1*) **have** *y*: *y* ≥ *0* **by** *simp*
  let *?s* = *root-int-ceiling-pos p x*
  let *?sx* = *root p* (*of-int x*)
  note *d* = *root-int-ceiling-pos-def*
  show *?thesis*
  proof (*cases b*)
    case *True*
    hence *id*: *?s* = *y* **unfolding** *s d* **using** *p* **by** *auto*
    from *rm*(*2*) *True* **have** *xy*: *x* = *y* ^ *p* **by** *auto*
    show *?thesis* **unfolding** *id* **unfolding** *xy* **using** *y*
      by (*simp add*: *p real-root-power-cancel*)

next
case *False*
hence *id*: *?s = root-int-floor-pos p x + 1* **unfolding** *d root-int-floor-pos-def*
using *s p* **by** *simp*
from *False* **have** *x0*: $x \neq 0$ **using** *rm*(*5*)[*of 0*] **using** *s* **unfolding** *root-int-main-def Let-def* **using** *p*
by (*cases x = 0*, *auto*)
show *?thesis* **unfolding** *id root-int-floor-pos*[*OF x*]
proof (*rule ceiling-unique*[*symmetric*])
show *?sx* ≤ *real-of-int* (⌊*root p* (*of-int x*)⌋ + *1*)
by (*metis of-int-add real-of-int-floor-add-one-ge of-int-1*)
let *?l = real-of-int* (⌊*root p* (*of-int x*)⌋ + *1*) − *1*
let *?m = real-of-int* ⌊*root p* (*of-int x*)⌋
have *?l = ?m* **by** *simp*
also have ... < *?sx*
proof −
have *le*: *?m* ≤ *?sx* **by** (*rule of-int-floor-le*)
have *neq*: *?m* ≠ *?sx*
proof
assume *?m = ?sx*
hence *?m* ^ *p = ?sx* ^ *p* **by** *auto*
also have ... = *of-int x* **using** *x False*
by (*metis p real-root-ge-0-iff real-root-pow-pos2 root-int-floor-pos root-int-floor-pos-pos zero-le-floor zero-less-Suc*)
finally have *xs*: *x* = ⌊*root p* (*of-int x*)⌋ ^ *p*
by (*metis floor-power floor-of-int*)
hence ⌊*root p* (*of-int x*)⌋ ∈ *set* (*root-int p x*) **using** *p* **by** *simp*
hence *root-int p x* ≠ [] **by** *force*
with *s False* ‹*p* ≠ *0*› *x x0* **show** *False* **unfolding** *root-int-def*
by (*cases p*, *auto*)
qed
from *le neq* **show** *?thesis* **by** *arith*
qed
finally show *?l* < *?sx* **.**
qed
qed
qed

definition *root-int-floor p x* = (*if x* ≥ *0 then root-int-floor-pos p x else* − *root-int-ceiling-pos p* (− *x*))
definition *root-int-ceiling p x* = (*if x* ≥ *0 then root-int-ceiling-pos p x else* − *root-int-floor-pos p* (− *x*))

lemma *root-int-floor*[*simp*]: *root-int-floor p x = floor* (*root p* (*of-int x*))
proof −
note *d = root-int-floor-def*
show *?thesis*
proof (*cases x* ≥ *0*)

case *True*
with *root-int-floor-pos*[*OF True*, *of p*] **show** *?thesis* **unfolding** *d* **by** *simp*
next
case *False*
hence $- x \geq 0$ **by** *auto*
from *False root-int-ceiling-pos*[*OF this*] **show** *?thesis* **unfolding** *d*
by (*simp add*: *real-root-minus ceiling-minus*)
qed
qed

lemma *root-int-ceiling*[*simp*]: *root-int-ceiling p x* = *ceiling* (*root p* (*of-int x*))
proof −
note *d* = *root-int-ceiling-def*
show *?thesis*
proof (*cases* $x \geq 0$)
case *True*
with *root-int-ceiling-pos*[*OF True*] **show** *?thesis* **unfolding** *d* **by** *simp*
next
case *False*
hence $- x \geq 0$ **by** *auto*
from *False root-int-floor-pos*[*OF this*, *of p*] **show** *?thesis* **unfolding** *d*
by (*simp add*: *real-root-minus floor-minus*)
qed
qed

3.4 Downgrading algorithms to the naturals

definition *root-nat-floor* :: *nat* ⇒ *nat* ⇒ *int* **where**
root-nat-floor p x = *root-int-floor-pos p* (*int x*)

definition *root-nat-ceiling* :: *nat* ⇒ *nat* ⇒ *int* **where**
root-nat-ceiling p x = *root-int-ceiling-pos p* (*int x*)

definition *root-nat* :: *nat* ⇒ *nat* ⇒ *nat list* **where**
root-nat p x = *map nat* (*take 1* (*root-int p x*))

lemma *root-nat-floor* [*simp*]: *root-nat-floor p x* = *floor* (*root p* (*real x*))
unfolding *root-nat-floor-def* **using** *root-int-floor-pos*[*of int x p*]
by *auto*

lemma *root-nat-floor-lower*: **assumes** *p0*: $p \neq 0$
shows *root-nat-floor p x* ^ $p \leq x$
using *root-int-floor-pos-lower*[*OF p0*, *of x*] **unfolding** *root-nat-floor-def* **by** *auto*

lemma *root-nat-floor-upper*: **assumes** *p0*: $p \neq 0$
shows (*root-nat-floor p x* + *1*) ^ $p > x$
using *root-int-floor-pos-upper*[*OF p0*, *of x*] **unfolding** *root-nat-floor-def* **by** *auto*

lemma *root-nat-ceiling* [*simp*]: *root-nat-ceiling p x* = *ceiling* (*root p x*)

unfolding *root-nat-ceiling-def* **using** *root-int-ceiling-pos*[*of x p*]
by *auto*

lemma *root-nat*: **assumes** *p0*: $p \neq 0 \vee x \neq 1$
shows *set* (*root-nat p x*) = { *y*. *y* ^ *p* = *x*}
proof −
{
fix *y*
assume $y \in$ *set* (*root-nat p x*)
note *y* = *this*[*unfolded root-nat-def*]
then obtain *yi ys* **where** *ri*: *root-int p x* = *yi* # *ys* **by** (*cases root-int p x*, *auto*)
with *y* **have** *y*: *y* = *nat yi* **by** *auto*
from *root-int-pos*[*OF - ri*] **have** *yi*: $0 \leq yi$ **by** *auto*
from *root-int*[*of p int x*] *p0 ri* **have** *yi* ^ *p* = *x* **by** *auto*
from *arg-cong*[*OF this*, *of nat*] *yi* **have** *nat yi* ^ *p* = *x*
by (*metis nat-int nat-power-eq*)
hence $y \in$ {*y*. *y* ^ *p* = *x*} **using** *y* **by** *auto*
}
moreover
{
fix *y*
assume *yx*: *y* ^ *p* = *x*
hence *y*: *int y* ^ *p* = *int x*
by (*metis of-nat-power*)
hence *set* (*root-int p* (*int x*)) $\neq$ {} **using** *root-int*[*of p int x*] *p0*
by (*metis* (*mono-tags*) *One-nat-def* ‹*y* ^ *p* = *x*› *empty-Collect-eq nat-power-eq-Suc-0-iff*)
then obtain *yi ys* **where** *ri*: *root-int p* (*int x*) = *yi* # *ys*
by (*cases root-int p* (*int x*), *auto*)
from *root-int-pos*[*OF - this*] **have** *yip*: $yi \geq 0$ **by** *auto*
from *root-int*[*of p int x*, *unfolded ri*] *p0* **have** *yi*: *yi* ^ *p* = *int x* **by** *auto*
with *y* **have** *int y* ^ *p* = *yi* ^ *p* **by** *auto*
from *arg-cong*[*OF this*, *of nat*] **have** *id*: *y* ^ *p* = *nat yi* ^ *p*
by (*metis* ‹*y* ^ *p* = *x*› *nat-int nat-power-eq yi yip*)
{
assume *p*: $p \neq 0$
hence *p0*: $p > 0$ **by** *auto*
obtain *yy b* **where** *rm*: *root-int-main p* (*int x*) = (*yy*,*b*) **by** *force*
from *root-int-main*(*5*)[*OF - rm p0 - y*] **have** *yy* = *int y* **and** *b* = *True* **by** *auto*
note *rm* = *rm*[*unfolded this*]
hence $y \in$ *set* (*root-nat p x*)
unfolding *root-nat-def p root-int-def* **using** *p0 p yx*
by *auto*
}
moreover
{
assume *p*: *p* = *0*
with *p0* **have** $x \neq 1$ **by** *auto*

with y p have $False$ by $auto$
}
ultimately have $y \in set$ ($root\text{-}nat$ p x) by $auto$
}
ultimately show $?thesis$ by $blast$
qed

3.5 Upgrading algorithms to the rationals

The main observation to lift everything from the integers to the rationals is the fact, that one can reformulate $\frac{a}{b}^{1/p}$ as $\frac{(ab^{p-1})^{1/p}}{b}$.

definition *root-rat-floor* :: *nat* ⇒ *rat* ⇒ *int* **where**
root-rat-floor p x ≡ *case quotient-of x of* (*a*,*b*) ⇒ *root-int-floor p* (*a* ∗ *b*^(*p* − *1*)) *div b*

definition *root-rat-ceiling* :: *nat* ⇒ *rat* ⇒ *int* **where**
root-rat-ceiling p x ≡ − (*root-rat-floor p* (−*x*))

definition *root-rat* :: *nat* ⇒ *rat* ⇒ *rat list* **where**
root-rat p x ≡ *case quotient-of x of* (*a*,*b*) ⇒ *concat*
(*map* (λ *rb*. *map* (λ *ra*. *of-int ra* / *rat-of-int rb*) (*root-int p a*)) (*take 1* (*root-int p b*)))

lemma *root-rat-reform*: **assumes** *q*: *quotient-of x* = (*a*,*b*)
shows *root p* (*real-of-rat x*) = *root p* (*of-int* (*a* ∗ *b* ^ (*p* − *1*))) / *of-int b*
proof (*cases p* = *0*)
case *False*
from *quotient-of-denom-pos*[*OF q*] **have** *b*: *0* < *b* **by** *auto*
hence *b*: *0* < *real-of-int b* **by** *auto*
from *quotient-of-div*[*OF q*] **have** *x*: *root p* (*real-of-rat x*) = *root p* (*a* / *b*)
by (*metis of-rat-divide of-rat-of-int-eq*)
also have *a* / *b* = *a* ∗ *real-of-int b* ^ (*p* − *1*) / *of-int b* ^ *p* **using** *b False*
by (*cases p*, *auto simp*: *field-simps*)
also have *root p* . . . = *root p* (*a* ∗ *real-of-int b* ^ (*p* − *1*)) / *root p* (*of-int b* ^ *p*) **by** (*rule real-root-divide*)
also have *root p* (*of-int b* ^ *p*) = *of-int b* **using** *False b*
by (*metis neq0-conv real-root-pow-pos real-root-power*)
also have *a* ∗ *real-of-int b* ^ (*p* − *1*) = *of-int* (*a* ∗ *b* ^ (*p* − *1*))
by (*metis of-int-mult of-int-power*)
finally show *?thesis* **.**
qed *auto*

lemma *root-rat-floor* [*simp*]: *root-rat-floor p x* = *floor* (*root p* (*of-rat x*))
proof −
obtain *a b* **where** *q*: *quotient-of x* = (*a*,*b*) **by** *force*
from *quotient-of-denom-pos*[*OF q*] **have** *b*: *b* > *0* **.**
show *?thesis*
unfolding *root-rat-floor-def q split root-int-floor*

unfolding *root-rat-reform*[*OF q*] *floor-div-pos-int*[*OF b*] **..**
qed

lemma *root-rat-ceiling* [*simp*]: *root-rat-ceiling p x* = *ceiling* (*root p* (*of-rat x*))
unfolding
root-rat-ceiling-def
ceiling-def
real-root-minus
root-rat-floor
of-rat-minus
..

lemma *root-rat*[*simp*]: **assumes** *p*: $p \neq 0 \vee x \neq 1$
shows *set* (*root-rat p x*) = { *y*. *y* ^ *p* = *x*}
proof (*cases p* = *0*)
case *False*
note *p* = *this*
obtain *a b* **where** *q*: *quotient-of x* = (*a*,*b*) **by** *force*
note *x* = *quotient-of-div*[*OF q*]
have *b*: *b* > *0* **by** (*rule quotient-of-denom-pos*[*OF q*])
note *d* = *root-rat-def q split set-concat set-map*
{
fix *q*
assume $q \in$ *set* (*root-rat p x*)
note *mem* = *this*[*unfolded d*]
from *mem* **obtain** *rb xs* **where** *rb*: *root-int p b* = *Cons rb xs* **by** (*cases root-int p b*, *auto*)
note *mem* = *mem*[*unfolded this*]
from *mem* **obtain** *ra* **where** *ra*: $ra \in$ *set* (*root-int p a*) **and** *q*: *q* = *of-int ra* / *of-int rb*
by (*cases root-int p a*, *auto*)
from *rb* **have** $rb \in$ *set* (*root-int p b*) **by** *auto*
with *ra p* **have** *rb*: *b* = *rb* ^ *p* **and** *ra*: *a* = *ra* ^ *p* **by** *auto*
have $q \in$ {*y*. *y* ^ *p* = *x*} **unfolding** *q x ra rb*
by (*auto simp*: *power-divide*)
}
moreover
{
fix *q*
assume $q \in$ {*y*. *y* ^ *p* = *x*}
hence *q* ^ *p* = *of-int a* / *of-int b* **unfolding** *x* **by** *auto*
hence *eq*: *of-int b* ∗ *q* ^ *p* = *of-int a* **using** *b* **by** *auto*
obtain *z n* **where** *quo*: *quotient-of q* = (*z*,*n*) **by** *force*
note *qzn* = *quotient-of-div*[*OF quo*]
have *n*: *n* > *0* **using** *quotient-of-denom-pos*[*OF quo*] **.**
from *eq*[*unfolded qzn*] **have** *rat-of-int b* ∗ *of-int z*^*p* / *of-int n*^*p* = *of-int a*
unfolding *power-divide* **by** *simp*
from *arg-cong*[*OF this*, *of* λ *x*. *x* ∗ *of-int n*^*p*] *n* **have** *rat-of-int b* ∗ *of-int z*^*p* = *of-int a* ∗ *of-int n* ^ *p*

by *auto*
also have *rat-of-int b* ∗ *of-int z*^*p* = *rat-of-int* (*b* ∗ *z*^*p*) **unfolding** *of-int-mult of-int-power* **..**
also have *of-int a* ∗ *rat-of-int n* ^ *p* = *of-int* (*a* ∗ *n* ^ *p*) **unfolding** *of-int-mult of-int-power* **..**
finally have *id*: *a* ∗ *n* ^ *p* = *b* ∗ *z* ^ *p* **by** *linarith*
from *quotient-of-coprime*[*OF quo*] **have** *cop*: *coprime* (*z* ^ *p*) (*n* ^ *p*)
by *simp*
from *coprime-crossproduct-int*[*OF quotient-of-coprime*[*OF q*] *this*] *arg-cong*[*OF id*, *of abs*]
have |*n* ^ *p*| = |*b*|
by (*simp add*: *field-simps abs-mult*)
with *n b* **have** *bnp*: *b* = *n* ^ *p* **by** *auto*
hence *rn*: *n* ∈ *set* (*root-int p b*) **using** *p* **by** *auto*
then obtain *rb rs* **where** *rb*: *root-int p b* = *Cons rb rs* **by** (*cases root-int p b*, *auto*)
from *id*[*folded bnp*] *b* **have** *a* = *z* ^ *p* **by** *auto*
hence *a*: *z* ∈ *set* (*root-int p a*) **using** *p* **by** *auto*
from *root-int-pos*[*OF - rb*] *b* **have** *rb0*: *rb* ≥ *0* **by** *auto*
from *root-int*[*OF disjI1*[*OF p*], *of b*] *rb* **have** *rb* ^ *p* = *b* **by** *auto*
with *bnp* **have** *id*: *rb* ^ *p* = *n* ^ *p* **by** *auto*
have *rb* = *n* **by** (*rule power-eq-imp-eq-base*[*OF id*], *insert n rb0 p*, *auto*)
with *rb* **have** *b*: *n* ∈ *set* (*take 1* (*root-int p b*)) **by** *auto*
have *q* ∈ *set* (*root-rat p x*) **unfolding** *d qzn* **using** *b a* **by** *auto*
}
ultimately show *?thesis* **by** *blast*
next
case *True*
with *p* **have** *x*: *x* ≠ *1* **by** *auto*
obtain *a b* **where** *q*: *quotient-of x* = (*a*,*b*) **by** *force*
show *?thesis* **unfolding** *True root-rat-def q split root-int-def* **using** *x*
by *auto*
qed

end

theory *Sqrt-Babylonian*
imports
Sqrt-Babylonian-Auxiliary
NthRoot-Impl
begin

4 Executable algorithms for square roots

This theory provides executable algorithms for computing square-roots of numbers which are all based on the Babylonian method (which is also known as Heron's method or Newton's method).

For integers / naturals / rationals precise algorithms are given, i.e., here *sqrt x* delivers a list of all integers / naturals / rationals y where $y^2 = x$. To this end, the Babylonian method has been adapted by using integer-divisions.

In addition to the precise algorithms, we also provide approximation algorithms. One works for arbitrary linear ordered fields, where some number y is computed such that $|y^2 - x| < \varepsilon$. Moreover, for the naturals, integers, and rationals we provide algorithms to compute $\lfloor sqrt\ x \rfloor$ and $\lceil sqrt\ x \rceil$ which are all based on the underlying algorithm that is used to compute the precise square-roots on integers, if these exist.

The major motivation for developing the precise algorithms was given by CeTA [2], a tool for certifiying termination proofs. Here, non-linear equations of the form $(a_1x_1 + \ldots a_nx_n)^2 = p$ had to be solved over the integers, where p is a concrete polynomial. For example, for the equation $(ax + by)^2 = 4x^2 - 12xy + 9y^2$ one easily figures out that $a^2 = 4, b^2 = 9$, and $ab = -6$, which results in a possible solution $a = \sqrt{4} = 2, b = -\sqrt{9} = -3$.

4.1 The Babylonian method

The Babylonian method for computing $\sqrt{n}$ iteratively computes

$$x_{i+1} = \frac{\frac{n}{x_i} + x_i}{2}$$

until $x_i^2 \approx n$. Note that if $x_0^2 \geq n$, then for all i we have both $x_i^2 \geq n$ and $x_i \geq x_{i+1}$.

4.2 The Babylonian method using integer division

First, the algorithm is developed for the non-negative integers. Here, the division operation $\frac{x}{y}$ is replaced by $x\ div\ y = \lfloor of\text{-}int\ x\ /\ of\text{-}int\ y \rfloor$. Note that replacing $\lfloor of\text{-}int\ x\ /\ of\text{-}int\ y \rfloor$ by $\lceil of\text{-}int\ x\ /\ of\text{-}int\ y \rceil$ would lead to non-termination in the following algorithm.

We explicititly develop the algorithm on the integers and not on the naturals, as the calculations on the integers have been much easier. For example, $y-x+x = y$ on the integers, which would require the side-condition $y \geq x$ for the naturals. These conditions will make the reasoning much more tedious—as we have experienced in an earlier state of this development where everything was based on naturals.

Since the elements $x_0, x_1, x_2, \ldots$ are monotone decreasing, in the main algorithm we abort as soon as $x_i^2 \leq n$.

Since in the meantime, all of these algorithms have been generalized to arbitrary p-th roots in *Sqrt-Babylonian.NthRoot-Impl*, we just instantiate the general algorithms by $p = 2$ and then provide specialized code equations which are more efficient than the general purpose algorithms.

definition *sqrt-int-main′* :: *int* ⇒ *int* ⇒ *int* × *bool* **where**
[*simp*]: *sqrt-int-main′ x n* = *root-int-main′ 1 1 2 x n*

lemma *sqrt-int-main′-code*[*code*]: *sqrt-int-main′ x n* = (*let x2* = *x* ∗ *x in if x2* ≤ *n then* (*x*, *x2* = *n*)
else sqrt-int-main′ ((*n div x* + *x*) *div 2*) *n*)
using *root-int-main′.simps*[*of 1 1 2 x n*]
unfolding *Let-def* **by** *auto*

definition *sqrt-int-main* :: *int* ⇒ *int* × *bool* **where**
[*simp*]: *sqrt-int-main x* = *root-int-main 2 x*

lemma *sqrt-int-main-code*[*code*]: *sqrt-int-main x* = *sqrt-int-main′* (*start-value x 2*) *x*
by (*simp add*: *root-int-main-def Let-def*)

definition *sqrt-int* :: *int* ⇒ *int list* **where**
sqrt-int x = *root-int 2 x*

lemma *sqrt-int-code*[*code*]: *sqrt-int x* = (*if x* < *0 then* [] *else case sqrt-int-main x of* (*y*,*True*) ⇒ *if y* = *0 then* [*0*] *else* [*y*,−*y*] | - ⇒ [])
proof −
interpret *fixed-root 2 1* **by** (*unfold-locales*, *auto*)
obtain *b y* **where** *res*: *root-int-main 2 x* = (*b*,*y*) **by** *force*
show *?thesis*
unfolding *sqrt-int-def root-int-def Let-def*
using *root-int-main*[*OF - res*]
using *res*
by *simp*
qed

lemma *sqrt-int*[*simp*]: *set* (*sqrt-int x*) = {*y*. *y* ∗ *y* = *x*}
unfolding *sqrt-int-def* **by** (*simp add*: *power2-eq-square*)

lemma *sqrt-int-pos*: **assumes** *res*: *sqrt-int x* = *Cons s ms*
shows *s* ≥ *0*
proof −
note *res* = *res*[*unfolded sqrt-int-code Let-def*, *simplified*]
from *res* **have** *x0*: *x* ≥ *0* **by** (*cases ?thesis*, *auto*)
obtain *ss b* **where** *call*: *sqrt-int-main x* = (*ss*,*b*) **by** *force*

from *res*[*unfolded call*] *x0* **have** *ss* = *s*
  by (*cases b*, *cases ss* = *0*, *auto*)
from *root-int-main*(*1*)[*OF x0 call*[*unfolded this sqrt-int-main-def*]]
show *?thesis* **.**
qed

definition [*simp*]: *sqrt-int-floor-pos x* = *root-int-floor-pos 2 x*

lemma *sqrt-int-floor-pos-code*[*code*]: *sqrt-int-floor-pos x* = *fst* (*sqrt-int-main x*)
  by (*simp add*: *root-int-floor-pos-def*)

lemma *sqrt-int-floor-pos*: **assumes** *x*: $x \geq 0$
  shows *sqrt-int-floor-pos x* = ⌊ *sqrt* (*of-int x*) ⌋
  using *root-int-floor-pos*[*OF x*, *of 2*] **by** (*simp add*: *sqrt-def*)

definition [*simp*]: *sqrt-int-ceiling-pos x* = *root-int-ceiling-pos 2 x*

lemma *sqrt-int-ceiling-pos-code*[*code*]: *sqrt-int-ceiling-pos x* = (*case sqrt-int-main x of* (*y*,*b*) ⇒ *if b then y else y* + *1*)
  by (*simp add*: *root-int-ceiling-pos-def*)

lemma *sqrt-int-ceiling-pos*: **assumes** *x*: $x \geq 0$
  shows *sqrt-int-ceiling-pos x* = ⌈ *sqrt* (*of-int x*) ⌉
  using *root-int-ceiling-pos*[*OF x*, *of 2*] **by** (*simp add*: *sqrt-def*)

definition *sqrt-int-floor x* = *root-int-floor 2 x*

lemma *sqrt-int-floor-code*[*code*]: *sqrt-int-floor x* = (*if* $x \geq 0$ *then sqrt-int-floor-pos x else* − *sqrt-int-ceiling-pos* (− *x*))
  unfolding *sqrt-int-floor-def root-int-floor-def* **by** *simp*

lemma *sqrt-int-floor*[*simp*]: *sqrt-int-floor x* = ⌊ *sqrt* (*of-int x*) ⌋
  by (*simp add*: *sqrt-int-floor-def sqrt-def*)

definition *sqrt-int-ceiling x* = *root-int-ceiling 2 x*

lemma *sqrt-int-ceiling-code*[*code*]: *sqrt-int-ceiling x* = (*if* $x \geq 0$ *then sqrt-int-ceiling-pos x else* − *sqrt-int-floor-pos* (− *x*))
  unfolding *sqrt-int-ceiling-def root-int-ceiling-def* **by** *simp*

lemma *sqrt-int-ceiling*[*simp*]: *sqrt-int-ceiling x* = ⌈ *sqrt* (*of-int x*) ⌉
  by (*simp add*: *sqrt-int-ceiling-def sqrt-def*)

lemma *sqrt-int-ceiling-bound*: $0 \leq x \Longrightarrow x \leq$ (*sqrt-int-ceiling x*)^*2*
  unfolding *sqrt-int-ceiling* **using** *le-of-int-ceiling sqrt-le-D*
  by (*metis of-int-power-le-of-int-cancel-iff*)

4.3 Square roots for the naturals

definition *sqrt-nat* :: *nat* ⇒ *nat list*
where *sqrt-nat x* = *root-nat 2 x*

lemma *sqrt-nat-code*[*code*]: *sqrt-nat x* ≡ *map nat* (*take 1* (*sqrt-int* (*int x*)))
unfolding *sqrt-nat-def root-nat-def sqrt-int-def* **by** *simp*

lemma *sqrt-nat*[*simp*]: *set* (*sqrt-nat x*) = { *y*. *y* ∗ *y* = *x*}
unfolding *sqrt-nat-def* **using** *root-nat*[*of 2 x*] **by** (*simp add*: *power2-eq-square*)

definition *sqrt-nat-floor* :: *nat* ⇒ *int* **where**
sqrt-nat-floor x = *root-nat-floor 2 x*

lemma *sqrt-nat-floor-code*[*code*]: *sqrt-nat-floor x* = *sqrt-int-floor-pos* (*int x*)
unfolding *sqrt-nat-floor-def root-nat-floor-def* **by** *simp*

lemma *sqrt-nat-floor*[*simp*]: *sqrt-nat-floor x* = ⌊ *sqrt* (*real x*) ⌋
unfolding *sqrt-nat-floor-def* **by** (*simp add*: *sqrt-def*)

definition *sqrt-nat-ceiling* :: *nat* ⇒ *int* **where**
sqrt-nat-ceiling x = *root-nat-ceiling 2 x*

lemma *sqrt-nat-ceiling-code*[*code*]: *sqrt-nat-ceiling x* = *sqrt-int-ceiling-pos* (*int x*)
unfolding *sqrt-nat-ceiling-def root-nat-ceiling-def* **by** *simp*

lemma *sqrt-nat-ceiling*[*simp*]: *sqrt-nat-ceiling x* = ⌈ *sqrt* (*real x*) ⌉
unfolding *sqrt-nat-ceiling-def* **by** (*simp add*: *sqrt-def*)

4.4 Square roots for the rationals

definition *sqrt-rat* :: *rat* ⇒ *rat list* **where**
sqrt-rat x = *root-rat 2 x*

lemma *sqrt-rat-code*[*code*]: *sqrt-rat x* = (*case quotient-of x of* (*z*,*n*) ⇒ (*case sqrt-int n of*
[] ⇒ []
| *sn* # *xs* ⇒ *map* (λ *sz*. *of-int sz* / *of-int sn*) (*sqrt-int z*)))
proof −
obtain *z n* **where** *q*: *quotient-of x* = (*z*,*n*) **by** *force*
show *?thesis*
unfolding *sqrt-rat-def root-rat-def q split sqrt-int-def*
by (*cases root-int 2 n*, *auto*)
qed

lemma *sqrt-rat*[*simp*]: *set* (*sqrt-rat x*) = { *y*. *y* ∗ *y* = *x*}
unfolding *sqrt-rat-def* **using** *root-rat*[*of 2 x*]
by (*simp add*: *power2-eq-square*)

lemma *sqrt-rat-pos*: **assumes** *sqrt*: *sqrt-rat x* = *Cons s ms*

shows $s \geq 0$
proof −
obtain $z\ n$ **where** q: *quotient-of* $x = (z,n)$ **by** *force*
note *sqrt* = *sqrt*[*unfolded sqrt-rat-code q*, *simplified*]
let *?sz* = *sqrt-int z*
let *?sn* = *sqrt-int n*
from q **have** n: $n > 0$ **by** (*rule quotient-of-denom-pos*)
from *sqrt* **obtain** *sz mz* **where** *sz*: *?sz* = *sz* # *mz* **by** (*cases ?sn*, *auto*)
from *sqrt* **obtain** *sn mn* **where** *sn*: *?sn* = *sn* # *mn* **by** (*cases ?sn*, *auto*)
from *sqrt-int-pos*[*OF sz*] *sqrt-int-pos*[*OF sn*] **have** *pos*: $0 \leq sz$ $0 \leq sn$ **by** *auto*
from *sqrt sz sn* **have** s: s = *of-int sz* / *of-int sn* **by** *auto*
show *?thesis* **unfolding** s **using** *pos*
by (*metis of-int-0-le-iff zero-le-divide-iff*)
qed

definition *sqrt-rat-floor* :: *rat* ⇒ *int* **where**
sqrt-rat-floor x = *root-rat-floor 2 x*

lemma *sqrt-rat-floor-code*[*code*]: *sqrt-rat-floor x* = (*case quotient-of x of* (*a*,*b*) ⇒ *sqrt-int-floor* (*a* ∗ *b*) *div b*)
unfolding *sqrt-rat-floor-def root-rat-floor-def* **by** (*simp add*: *sqrt-def*)

lemma *sqrt-rat-floor*[*simp*]: *sqrt-rat-floor x* = ⌊ *sqrt* (*of-rat x*) ⌋
unfolding *sqrt-rat-floor-def* **by** (*simp add*: *sqrt-def*)

definition *sqrt-rat-ceiling* :: *rat* ⇒ *int* **where**
sqrt-rat-ceiling x = *root-rat-ceiling 2 x*

lemma *sqrt-rat-ceiling-code*[*code*]: *sqrt-rat-ceiling x* = − (*sqrt-rat-floor* (−*x*))
unfolding *sqrt-rat-ceiling-def sqrt-rat-floor-def root-rat-ceiling-def* **by** *simp*

lemma *sqrt-rat-ceiling*: *sqrt-rat-ceiling x* = ⌈ *sqrt* (*of-rat x*) ⌉
unfolding *sqrt-rat-ceiling-def* **by** (*simp add*: *sqrt-def*)

lemma *sqr-rat-of-int*: **assumes** x: $x * x$ = *rat-of-int i*
shows ∃ *j* :: *int*. $j * j = i$
proof −
from x **have** *mem*: $x \in$ *set* (*sqrt-rat* (*rat-of-int i*)) **by** *simp*
from x **have** *rat-of-int i* ≥ 0 **by** (*metis zero-le-square*)
hence ∗: *quotient-of* (*rat-of-int i*) = (*i*,*1*) **by** (*metis quotient-of-int*)
have *1*: *sqrt-int 1* = [*1*,−*1*] **by** *code-simp*
from *mem sqrt-rat-code* ∗ *split 1*
have x: $x \in$ *rat-of-int* ‘ {*y*. *y* ∗ *y* = *i*} **by** *auto*
thus *?thesis* **by** *auto*
qed

4.5 Approximating square roots

The difference to the previous algorithms is that now we abort, once the distance is below ϵ. Moreover, here we use standard division and not integer division. This part is not yet generalized by *Sqrt-Babylonian.NthRoot-Impl*.

We first provide the executable version without guard $0 < x$ as partial function, and afterwards prove termination and soundness for a similar algorithm that is defined within the upcoming locale.

partial-function (*tailrec*) *sqrt-approx-main-impl* :: *'a* :: *linordered-field* ⇒ *'a* ⇒ *'a* ⇒ *'a* **where**
[*code*]: *sqrt-approx-main-impl ε n x* = (*if x ∗ x − n < ε then x else sqrt-approx-main-impl ε n*
((*n* / *x* + *x*) / *2*))

We setup a locale where we ensure that we have standard assumptions: positive ϵ and positive n. We require sort *floor-ceiling*, since $\lfloor x \rfloor$ is used for the termination argument.

locale *sqrt-approximation* =
fixes *ε* :: *'a* :: {*linordered-field*,*floor-ceiling*}
and *n* :: *'a*
assumes *ε* : *ε* > *0*
and *n*: *n* > *0*
begin

function *sqrt-approx-main* :: *'a* ⇒ *'a* **where**
sqrt-approx-main x = (*if x* > *0 then* (*if x ∗ x − n < ε then x else sqrt-approx-main*

((*n* / *x* + *x*) / *2*)) *else 0*)
by *pat-completeness auto*

Termination essentially is a proof of convergence. Here, one complication is the fact that the limit is not always defined. E.g., if *'a* is *rat* then there is no square root of 2. Therefore, the error-rate $\frac{x}{\sqrt{n}} - 1$ is not expressible. Instead we use the expression $\frac{x^2}{n} - 1$ as error-rate which does not require any square-root operation.

termination
proof −
define *er* **where** *er x* = (*x ∗ x* / *n* − *1*) **for** *x*
define *c* **where** *c* = *2 ∗ n* / *ε*
define *m* **where** *m x* = *nat* ⌊ *c ∗ er x* ⌋ **for** *x*
have *c*: *c* > *0* **unfolding** *c-def* **using** *n ε* **by** *auto*
show *?thesis*
proof
show *wf* (*measures* [*m*]) **by** *simp*
next
fix *x*
assume *x*: *0* < *x* **and** *xe*: ¬ *x ∗ x − n < ε*

```
    define y where y = (n / x + x) / 2
    show ((n / x + x) / 2,x) ∈ measures [m] unfolding y-def[symmetric]
    proof (rule measures-less)
      from n have inv-n: 1 / n > 0 by auto
      from xe have x * x − n ≥ ε by simp
      from this[unfolded mult-le-cancel-left-pos[OF inv-n, of ε, symmetric]]
     have erxen: er x ≥ ε / n unfolding er-def using n by (simp add: field-simps)
      have en: ε / n > 0 and ne: n / ε > 0 using ε n by auto
      from en erxen have erx: er x > 0 by linarith
      have pos: er x * 4 + er x * (er x * 4) > 0 using erx
        by (auto intro: add-pos-nonneg)
      have er y = 1 / 4 * (n / (x * x) − 2 + x * x / n) unfolding er-def y-def
using x n
        by (simp add: field-simps)
     also have . . . = 1 / 4 * er x * er x / (1 + er x) unfolding er-def using x n
        by (simp add: field-simps)
      finally have er y = 1 / 4 * er x * er x / (1 + er x) .
      also have . . . < 1 / 4 * (1 + er x) * er x / (1 + er x) using erx erx pos
        by (auto simp: field-simps)
      also have . . . = er x / 4 using erx by (simp add: field-simps)
      finally have er-y-x: er y ≤ er x / 4 by linarith
     from erxen have c * er x ≥ 2 unfolding c-def mult-le-cancel-left-pos[OF ne,
of - er x, symmetric]
        using n ε by (auto simp: field-simps)
      hence pos: ⌊c * er x⌋ > 0 ⌊c * er x⌋ ≥ 2 by auto
      show m y < m x unfolding m-def nat-mono-iff[OF pos(1)]
      proof −
        have ⌊c * er y⌋ ≤ ⌊c * (er x / 4)⌋
          by (rule floor-mono, unfold mult-le-cancel-left-pos[OF c], rule er-y-x)
        also have . . . < ⌊c * er x / 4 + 1⌋ by auto
        also have . . . ≤ ⌊c * er x⌋
          by (rule floor-mono, insert pos(2), simp add: field-simps)
        finally show ⌊c * er y⌋ < ⌊c * er x⌋ .
      qed
    qed
  qed
qed
```

Once termination is proven, it is easy to show equivalence of *sqrt-approx-main-impl* and *sqrt-approx-main*.

```
lemma sqrt-approx-main-impl: x > 0 =⇒ sqrt-approx-main-impl ε n x = sqrt-approx-main
x
proof (induct x rule: sqrt-approx-main.induct)
  case (1 x)
  hence x: x > 0 by auto
  hence nx: 0 < (n / x + x) / 2 using n by (auto intro: pos-add-strict)
  note simps = sqrt-approx-main-impl.simps[of - - x] sqrt-approx-main.simps[of x]
  show ?case
  proof (cases x * x − n < ε)
```

case *True*
thus *?thesis* **unfolding** *simps* **using** *x* **by** *auto*
next
case *False*
show *?thesis* **using** *1(1)[OF x False nx]* **unfolding** *simps* **using** *x False* **by** *auto*
qed
qed

Also soundness is not complicated.

lemma *sqrt-approx-main-sound*: **assumes** *x*: $x > 0$ **and** *xx*: $x * x > n$
shows $\textit{sqrt-approx-main}\ x * \textit{sqrt-approx-main}\ x > n \wedge \textit{sqrt-approx-main}\ x * \textit{sqrt-approx-main}\ x - n < \varepsilon$
using *assms*
proof (*induct x rule*: *sqrt-approx-main.induct*)
case (*1 x*)
from *1* **have** *x*: $x > 0$ $(x > 0) = \textit{True}$ **by** *auto*
note *simp* = *sqrt-approx-main.simps*[*of x*, *unfolded x if-True*]
show *?case*
proof (*cases* $x * x - n < \varepsilon$)
case *True*
with *1* **show** *?thesis* **unfolding** *simp* **by** *simp*
next
case *False*
let $?y = (n\ /\ x + x)\ /\ 2$
from *False simp* **have** *simp*: *sqrt-approx-main x* = *sqrt-approx-main ?y* **by** *simp*
from *n x* **have** *y*: $?y > 0$ **by** (*auto intro*: *pos-add-strict*)
note *IH* = *1(1)[OF x(1) False y]*
from *x* **have** *x4*: $4 * x * x > 0$ **by** (*auto intro*: *mult-sign-intros*)
show *?thesis* **unfolding** *simp*
proof (*rule IH*)
show $n < ?y * ?y$
unfolding *mult-less-cancel-left-pos*[*OF x4*, *of n*, *symmetric*]
proof −
have *id*: $4 * x * x * (?y * ?y) = 4 * x * x * n + (n - x * x) * (n - x * x)$ **using** *x(1)*
by (*simp add*: *field-simps*)
from *1(3)* **have** $x * x - n > 0$ **by** *auto*
from *mult-pos-pos*[*OF this this*]
show $4 * x * x * n < 4 * x * x * (?y * ?y)$ **unfolding** *id*
by (*simp add*: *field-simps*)
qed
qed
qed
qed

end

It remains to assemble everything into one algorithm.

definition *sqrt-approx* :: $'a$ :: {*linordered-field*,*floor-ceiling*} $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ **where**
sqrt-approx ε *x* $\equiv$ *if* ε > *0* *then* (*if* *x* = *0* *then* *0* *else* *let* *xpos* = *abs* *x* *in* *sqrt-approx-main-impl* ε *xpos* (*xpos* + *1*)) *else* *0*

lemma *sqrt-approx*: **assumes** ε: ε > *0*
shows |*sqrt-approx* ε *x* * *sqrt-approx* ε *x* − |*x*|| < ε
proof (*cases* *x* = *0*)
case *True*
with ε **show** *?thesis* **unfolding** *sqrt-approx-def* **by** *auto*
next
case *False*
let *?x* = |*x*|
let *?sqrti* = *sqrt-approx-main-impl* ε *?x* (*?x* + *1*)
let *?sqrt* = *sqrt-approximation.sqrt-approx-main* ε *?x* (*?x* + *1*)
define *sqrt* **where** *sqrt* = *?sqrt*
from *False* **have** *x*: *?x* > *0* *?x* + *1* > *0* **by** *auto*
interpret *sqrt-approximation* ε *?x*
by (*unfold-locales*, *insert* *x* ε, *auto*)
from *False* ε **have** *sqrt-approx* ε *x* = *?sqrti* **unfolding** *sqrt-approx-def* **by** (*simp* *add*: *Let-def*)
also have *?sqrti* = *?sqrt*
by (*rule* *sqrt-approx-main-impl*, *auto*)
finally have *id*: *sqrt-approx* ε *x* = *sqrt* **unfolding** *sqrt-def* **.**
have *sqrt*: *sqrt* * *sqrt* > *?x* ∧ *sqrt* * *sqrt* − *?x* < ε **unfolding** *sqrt-def*
by (*rule* *sqrt-approx-main-sound*[*OF* *x*(*2*)], *insert* *x* *mult-pos-pos*[*OF* *x*(*1*) *x*(*1*)], *auto* *simp*: *field-simps*)
show *?thesis* **unfolding** *id* **using** *sqrt* **by** *auto*
qed

4.6 Some tests

Testing executabity and show that sqrt 2 is irrational

lemma ¬ (∃ *i* :: *rat*. *i* * *i* = *2*)
proof −
have *set* (*sqrt-rat* *2*) = {} **by** *eval*
thus *?thesis* **by** *simp*
qed

Testing speed

lemma ¬ (∃ *i* :: *int*. *i* * *i* = *1234567890123456789012345678901234567890*)
proof −
have *set* (*sqrt-int* *1234567890123456789012345678901234567890*) = {} **by** *eval*
thus *?thesis* **by** *simp*
qed

The following test

value *let* ε = *1* / *100000000* :: *rat*; *s* = *sqrt-approx* ε *2* *in* (*s*, *s* * *s* − *2*, |*s* * *s* − *2*| < ε)

results in (1.4142135623731116, 4.738200762148612e-14, True).

end

Acknowledgements

We thank Bertram Felgenhauer for for mentioning Cauchy's mean theorem during the formalization of the algorithms for computing n-th roots.

References

[1] T. Heath. *A History of Greek Mathematics*, volume 2, pages 323–326. Clarendon Press, 1921.

[2] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proc. TPHOLs'09*, volume 5674 of *LNCS*, pages 452–468, 2009.