Computing N-th Roots using the Babylonian Method*

René Thiemann

May 27, 2015

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

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

2 Executable algorithms for p-th roots 5
  2.1 Logarithm .............................. 5
  2.2 Computing the p-th root of an integer number .......................... 7
  2.3 Floor and ceiling of roots .............................. 19
  2.4 Downgrading algorithms to the naturals ......................... 22
  2.5 Upgrading algorithms to the rationals .......................... 24

3 Executable algorithms for square roots 26
  3.1 The Babylonian method .............................. 27
  3.2 The Babylonian method using integer division ......................... 27
  3.3 Square roots for the naturals .............................. 29
  3.4 Square roots for the rationals .............................. 30
  3.5 Approximating square roots .............................. 31
  3.6 Some tests .............................. 35

*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
  ~~/src/HOL/Transcendental
begin

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

lemma log-pow-cancel[simp]: a > 0 ⇒ a ≠ 1 ⇒ log a (a ^ b) = b
  by (metis monoid-mult-class.mult.right-neutral log-eq-one log-nat-power)

lemma real-of-rat-floor[simp]: floor (real-of-rat x) = floor x
  by (metis Ratreal-def real-floor-code)

lemma abs-of-rat[simp]: |real-of-rat x| = real-of-rat |x|
proof (cases x ≥ 0)
  case False
  def y ≡ - x
  from False have y: y ≥ 0 x = - y by (auto simp: y-def)
  thus thesis by (auto simp: of-rat-minus)
qed auto

lemma real-of-rat-ceiling[simp]: ceiling (real-of-rat x) = ceiling x
unfolding ceiling-def
by (metis of-rat-minus real-of-rat-floor)

lemma div-is-floor-divide-rat: n div y = ⌊rat-of-int n / rat-of-int y⌋
unfolding Fract-of-int-quotient[symmetric] floor-Fract
by simp

lemma div-is-floor-divide-real: n div y = [real-of-int n / of-int y]
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-ceiling
  assumes n: n > 0
  shows [r / af-int n] = [r] div n (is ?l = ?r)
proof –
  let ?of-int = of-int :: int ⇒ 'a
  def rhs ≡ [r] div n
  let ?n = ?of-int n
  def m ≡ [r] mod n
  let ?m = ?of-int m
  from mod-div-equality[of floor r n] have dm: rhs * n + m = [r] unfolding rhs-def m-def by simp
  have mn: m < n and m0: m ≥ 0 using n m-def by auto
  def e ≡ r - ?of-int [r]


have \( e0 : e \geq 0 \) unfolding e-def
  by (metis diff-self eq_iff floor-diff_of_int zero_floor)
have \( e1 : e < 1 \) unfolding e-def
  by (metis diff-self dual_order refl floor-diff_of_int floor_le_zero)
also have \([r] = \text{rhs} \cdot n + m\) using dm by simp
finally have \( r = ?\text{of-int} \lfloor r \rfloor + e \).

hence \( r / ?n = ?\text{of-int} \lfloor (\text{rhs} \cdot n) / ?n + (e + ?m) / ?n\) using \( n \) by (simp add: field_simps)
also have \( ?\text{of-int} (\text{rhs} \cdot n) / ?n = ?\text{of-int} \text{rhs} \) using \( n \) by auto
finally have \( \lfloor r \rfloor / ?n = ?\text{of-int} ((\text{rhs} \cdot n) / ?n + e) \).

have e0: \( e \geq 0 \) unfolding e-def
  by (metis diff-self eq_iff floor-diff_of_int zero_floor)
have e1: \( e < 1 \) unfolding e-def
  by (metis diff-self dual_order refl floor-diff_of_int floor_le_zero)
also have \([r] = \text{rhs} \cdot n + m\) using dm by simp
finally have \( r = ?\text{of-int} \lfloor (\text{rhs} \cdot n + m) + e \).

hence \( r / ?n = ?\text{of-int} \lfloor (\text{rhs} \cdot n) / ?n + (e + ?m) / ?n\) using \( n \) by (simp add: field_simps)
also have \( ?\text{of-int} (\text{rhs} \cdot n) / ?n = ?\text{of-int} \text{rhs} \) using \( n \) by auto
finally have \( \lfloor r \rfloor / ?n = ?\text{of-int} ((\text{rhs} \cdot n) / ?n + e) \).

lemma floor_div_neg_int:
  fixes \( r :: \text{a} :: \text{floor}\_ceiling\)
  assumes \( n : n < 0 \)
  shows \([r / \text{of-int} n] = \lfloor r \rfloor \div n\)
proof
  from \( n \) have \( n' : ?n > 0 \) ?n \( \geq 0 \) by simp-all
  assume \( \neg \text{thesis} \)
  hence \( (e + ?m) / ?n \geq 1 \) by auto
  from mult_right_mono[OF this \( n' \)]
  have \( ?n \leq e + ?m \) using \( n \) by simp
  also have \( ?m \leq ?n - 1 \) using \( m \)
    by (metis of_int_minus of_int_diff of_int_less of_int_diff1_eq)
  finally have \( ?n \leq e + 1 \) by auto
  with e1 show False by arith
qed

finally show \text{thesis} unfolding rhs-def by simp
qed

lemma floor_div_neg_int:
  fixes \( r :: \text{a} :: \text{floor}\_ceiling\)
  assumes \( n : n < 0 \)
  shows \([r / \text{of-int} n] = \lfloor r \rfloor \div n\)
proof
  from \( n \) have \( n' : ?n > 0 \) ?n \( \geq 0 \) by auto
  have \( [r / \text{of-int} n] = [-r] \div (-n) \) 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 \( \ldots = [-r] \div (-n) \) by (rule floor_div_pos_int[OF \( n' \)])
  also have \( \ldots = [r] \div n \) using \( n \)
    by (metis ceiling_def div_minus_right)
  finally show \text{thesis} .
qed
lemma divide-less-floor1: \( n / y < \text{of-int} \, (\text{floor} \, (n / y)) + 1 \)
by (metis floor-less-iff less-add-one of-int-1 of-int-add)

context linordered-idom
begin

lemma sgn-int-pow[simp]: \( \text{sgn} \, ((x :: 'a) ^ p) = \text{sgn} \, x ^ p \)
by (induct p, auto simp: sgn-times)

lemma sgn-int-pow-if[simp]: assumes \( x :: 'a \neq 0 \)
shows \( \text{sgn} \, x ^ p = (\text{if even} \, p \, \text{then} \, 1 \, \text{else} \, \text{sgn} \, x) \)
proof (induct p, auto)
show \( \text{sgn} \, x \times \text{sgn} \, x = 1 \)
using \( x \)
by (metis linorder-neqE-linordered-idom mult-eq-0-iff not-square-less-zero sgn-pos sgn-times)
qed

lemma compare-pow-le-iff: \( p > 0 \implies (x :: 'a) \geq 0 \implies y \geq 0 \implies (x ^ p \leq y ^ p) = (x \leq y) \)
by (metis eq-iff linear power-eq-imp-eq-base power-mono)

lemma compare-pow-less-iff: \( p > 0 \implies (x :: 'a) \geq 0 \implies y \geq 0 \implies (x ^ p < y ^ p) = (x < y) \)
by (metis power-less-imp-less-base power-strict-mono)
end

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

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

lemma square-lesseq-square: \( \forall x \, y. \, 0 \leq (x :: 'a :: \text{linordered-field}) \implies 0 \leq y \implies (x \times x \leq y \times y) = (x \leq y) \)
by (metis mult-mono mult-strict-mono' not-less)

lemma square-less-square: \( \forall x \, y. \, 0 \leq (x :: 'a :: \text{linordered-field}) \implies 0 \leq y \implies (x \times x < y \times y) = (x < y) \)
by (metis mult-mono mult-strict-mono' not-less)

lemma sqrt-sqrt[simp]: \( x \geq 0 \implies \sqrt{x} \times \sqrt{x} = x \)
by (metis real-sqrt-pow2 power2-eq-square)

lemma abs-lesseq-square: abs \( (x :: \text{real}) \) \leq abs \( y \) \iff \( x \times x \leq y \times y \)
using square-lesseq-square[of abs \( x \) \( abs \, y \)] by auto

end
2 Executable algorithms for \( p \)-th roots

theory NthRoot-Impl

imports
  Sqrt-Babylonian-Auxiliary
  ../Cauchy/CauchyMeanTheorem

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.

2.1 Logarithm

For computing the \( p \)-th root of a number \( n \), we must choose a starting value in the iteration. Here, we use \((2::':\mathbb{N})^\lceil \lceil \log_2 n \rceil / p \rceil\). Of course, this requires an algorithm to compute logarithms. Here, we just multiply with the base, until we exceed the argument.

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.

partial-function (tailrec) log-ceil-impl :: \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N} where
  [code]: log-ceil-impl b x prod sum = (if prod ≥ x then sum else log-ceil-impl b x (prod * b) (sum + 1))

definition log-ceil :: \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N} where
  log-ceil b x ≡ if b > 1 ∧ x ≥ 0 then log-ceil-impl b 1 0 else 0

context
  fixes b :: \mathbb{N}
  assumes b: b > 1

begin

function log-ceil-main :: \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N} where
  log-ceil-main x prod sum = (if prod > 0 then (if prod ≥ x then sum else log-ceil-main x (prod * b) (sum + 1)) else 0)
  by pat-completeness auto

termination by (relation measure (\( \lambda (x,prod,sum). \mathbb{N} (x + 1 - prod) \))) (insert b, auto)

lemma log-ceil-impl: prod > 0 ⟹ log-ceil-impl b x prod sum = log-ceil-main x prod sum
proof (induct x prod sum rule: log-ceil-main.induct)
  case (1 x p s)
hence id: (0 < p) = True using b by auto
have pos: Suc 0 < b 0 < p 0 < p * int b
   by (metis Suc-lessD mult-pos-pos of-nat-0-less-iff)
show ?case unfolding log-ceil-impl.simps[of b p s] log-ceil-main.simps[of b p s]
   id if-True
   by (rule if-cong[OF refl refl 1(1)], insert 1(2) b pos, auto)
qed
end

lemma log-ceil[simp]: assumes b: b > 0 and x: x > 0
shows log-ceil b x = ⌈log b x⌉ proof (cases b = 1)
   case True
   hence log-ceil b x = (0 :: int)
      unfolding log-ceil-def by auto
also have .. = ⌈log b x⌉ unfolding True by (simp add: log-def)
finally show ?thesis by auto
next
   case False
   with b have b: b > 1 unfolding b by auto
   def p ≡ 1 :: int
   def s ≡ 0 :: nat
   have int b ^ s = p unfolding p-def s-def using x by force
   have inv: s = 0 ∨ int b ^ (s - 1) < x unfolding s-def by auto
   hence int (log-ceil b x) = log-ceil-impl b x 1 0 using x b unfolding log-ceil-def
      by auto
   also have .. = log-ceil-main b x p s unfolding log-ceil-impl[OF b, of x p s]
      by (simp add: p-def s-def)
   also have .. = log-ceil-main b x p s using log-ceil-main[OF b, of p s]
      by (simp add: log-ceil-def)
proof (cases x ≤ p)
   case True
   have b0: b > (0 :: real) and b1: b ≠ (1 :: real) b > (1 :: real)
      using b by auto
   from True[folded 1(2)] have low: x ≤ int b ^ s by auto
   from 1(3) have up: s = 0 ∨ s ≠ 0 ∧ int b ^ (s - 1) < x by auto
   from True have id: int (log-ceil-main b x p s) = s unfolding id by simp
   from low have real x ≤ real b ^ s
      by (metis real-of-int-le-iff real-of-int-of-nat-eq real-of-int-power)
   hence log b x ≤ log b (real b ^ s)
      using log-le-cancel-iff[of b x real b ^ s] b x 1(4)
      by (metis less-eq-real-def not-le real-of-int-gl-zero-cancel-iff real-of-int-1
      real-of-nat-ge-zero real-of-nat-less-iff zero-le-power)
also have \ldots = s using \( s \) by \( \text{simp} \)
also have \ldots = real-of-int \((\text{int} \ s)\)
by \(\text{metis real-eq-of-int real-of-int-def real-of-int-of-nat-eq}\)
finally have low: \(\log b x \leq \text{of-int} \((\text{int} \ s)\)\).
show \(\text{thesis}\) unfolding \(\text{id}\).

proof (rule sym, rule ceiling-unique[OF - low])
from up show real-of-int \((\text{int} \ s)\) - 1 < \(\log b x\)
proof
assume \(s = 0\)
have \(\log b x \geq 0\) using \(b \ (4)\) by \(\text{simp}\)
with \((s = 0)\) show \(\text{thesis}\) by \(\text{auto}\)
next
def \(ss \equiv s - 1\)
assume \(\ast\): \(s \neq 0 \wedge \text{int b} \ ^{\sim} \ (s - 1) < x\)
hence real-of-int \((\text{int} \ s)\) - 1 = of-int \((\text{int} \ (ss + 1))\) - 1 unfolding \(ss\)-def
by \(\text{auto}\)
also have \ldots = \(ss\)
by \(\text{metis add-diff-cancel of-int-of-nat-eq of-nat-1 real-of-nat-add real-of-nat-def}\)
finally have \(\text{id}\): real-of-int \((\text{int} \ s)\) - 1 = \(ss\) .
have \(0 < \text{real} \ ((\text{int} b \ ^{\sim} \ ss))\) 0 < real \(x\) using \(b \ (4)\) by \(\text{auto}\)
note \(\log-\text{mono} = \log-\text{less-cancel-iff}[OF b1(2) \ \text{this}]\)
from \(\ast\) have up: \(\text{int b} \ ^{\sim} \ ss < x\) unfolding \(ss\)-def by \(\text{auto}\)
hence real-of-int \((\text{int} \ (ss))\) < \(x\) by \(\text{fast}\)
from \text{this[folded log-\text{mono}]} have \(\log b \ (\text{real} \ ((\text{int} b \ ^{\sim} \ ss))) < \log b x\) .
also have \(\text{real} \ ((\text{int} b \ ^{\sim} \ ss)) = \text{real b} \ ^{\sim} \ ss\) by \(\text{simp}\)
also have \(\log b \ (\text{real b} \ ^{\sim} \ ss) = ss\) using \(b\) by \(\text{simp}\)
finally show \(\text{thesis}\) unfolding \(\text{id}\).
qed
qed
next
case \text{False}
hence \(x > p\) by \(\text{auto}\)
with \(\text{id}\) have \(\text{id}\): log-ceil-main \(b \ x \ p \ s = \log-\text{ceil-main} \ b \ x \ (p \ast \text{int b}) \ (s + 1)\)
by \(\text{auto}\)
from 1(2) have prod: \((\text{int b}) ^{\ (s + 1)} = p \ast \text{int b}\) by \(\text{auto}\)
show \(\text{thesis}\) unfolding \(\text{id}\)
by \(\text{rule 1(1)[OF p False prod - 1(4)], insert False 1(2), auto}\)
qed
qed
finally show \(\text{thesis}\) .
qed

2.2 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 \(x \leq (\text{start-value} \ x \ p)^p\), 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 :: \text{int} \Rightarrow \text{nat} \Rightarrow \text{int} where
\[ \text{start-value } n \cdot p = 2^\ast \left( \text{nat } \left( \text{of-int } \left( \log \text{-ceil} \ 2 \ n \right) / \text{rat-of-nat } p \right) \right) \]

**lemma** start-value-main: assumes \( x : x \geq 0 \) and \( p : p > 0 \)
shows \( x \leq (\text{start-value } x \cdot p) \wedge \text{start-value } x \cdot p \geq 0 \)
proof (cases \( x = 0 \))
case True
with \( p \) show \( \text{thesis unfolding start-value-def True by simp} \)
next
case False
with \( x \) have \( x : x > 0 \) by auto
def \( l2x \equiv \lceil \log 2 \cdot x \rceil \)
def \( \text{pow} \equiv \text{nat } \left[ \text{rat-of-int } l2x / \text{of-nat } p \right] \)
\[ \begin{aligned}
\text{have} \ & \text{root } p \ x = x \ \text{pow} \ (1 / p) \ \text{by} \ (\text{rule root-powr-inverse, insert } x \ p, \ \text{auto}) \\
n\text{also have } \ldots = (2 \ \text{pow} \ (\log 2 \ x)) \ \text{pow} \ (1 / p) \ \text{using} \ \text{powr-log-cancel[of } 2 \ x] \ x \ \text{by auto} \\
\text{also have} \ \ldots = 2 \ \text{pow} \ (\log 2 \ x * (1 / p)) \ \text{by} \ (\text{rule powr-powr}) \\
\text{also have} \ \log 2 \ x * (1 / p) = \log 2 \ x / p \ \text{using} \ p \ \text{by auto} \\
\text{finally have} \ r : \ \text{root } p \ x = 2 \ \text{pow} \ (\log 2 \ x / p). \\
\text{have } l p : \ \log 2 \ x \geq 0 \ \text{using} \ x \ \text{by auto} \\
\text{hence } l2pos : \ l2x \geq 0 \ \text{by} \ (\text{auto simp: } l2x-def) \\
\text{have } \log 2 \ x / p \leq \ l2x / p \ \text{using} \ x \ \text{unfolding} \ l2x-def \ \\
\text{by} \ (\text{metis divide-right-mono real-of-int-ceiling-ge real-of-nat-ge-zero}) \\
\text{also have} \ \ldots \leq \ \lceil \text{l2x} / (p :: \text{real}) \rceil \ \text{by simp} \\
\text{also have} \ l2x / \text{real } p = l2x / \text{real-of-rat (of-nat } p) \\
\text{by} \ (\text{metis of-rat-of-nat-eq real-eq-of-nat real-of-nat-def}) \\
\text{also have} \ \text{real } l2x = \text{real-of-rat (of-int l2x) \ \\
\text{by} \ (\text{metis of-rat-of-int-eq real-eq-of-int real-of-int-def}) \\
\text{also have} \ \text{real-of-rat (of-int l2x)} / \text{real-of-rat (of-nat } p) = \text{real-of-rat (rat-of-int l2x / of-nat } p) \\
\text{by} \ (\text{metis of-rat-divide}) \\
\text{also have} \ \text{real-of-rat (rat-of-int l2x / rat-of-nat } p) = \text{rat-of-int l2x / of-nat } p \\
\text{by simp} \\
\text{also have} \ \text{rat-of-int l2x / of-nat } p \leq \text{real pow unfolding pow-def by auto} \\
\text{finally have } \text{le : log 2 \ x / p } \leq \text{pow}. \\
\text{from} \ \text{powr-mono[OF le, of 2, folded r]} \\
\text{have} \ \text{root } p \ x \leq 2 \ \text{powr pow by auto} \\
\text{also have} \ \ldots = 2 \ \ast \ \text{pow by (rule powr-realpow, auto)} \\
\text{also have} \ \ldots = \text{real } ((2 :: \text{int}) \ \ast \ \text{pow}) \ \text{by simp} \\
\text{also have} \ \text{pow} = \ \text{(nat } [\text{of-int } (\log-ceil 2 \ x) / \text{rat-of-nat } p] \text{)} \\
\text{unfolding pow-def l2x-def using } x \ \text{by simp} \\
\text{also have} \ \text{real } ((2 :: \text{int}) \ \ast \ldots) = \text{start-value } x \ p \ \text{unfolding start-value-def by simp} \\
\text{finally have } \text{less : root } p \ x \leq \text{start-value } x \ p. \\
\text{have } 0 \leq \text{root } p \ x \ \text{using } p \ x \ \text{by auto} \\
\text{also have} \ \ldots \leq \text{start-value } x \ p \ \text{by (rule less)} \\
\text{finally have} \ \text{start : 0 } \leq \text{start-value } x \ p \ \text{by simp} \\
\text{from} \ \text{power-mono[OF less, of } p] \text{ have root } p \ \text{(real } x) \ \ast \ p \leq \text{real } (\text{start-value } x \ p) \ \ast \ p \ \text{using } p \ x \ \text{by auto} \]
also have \ldots = \text{start-value } x \ p \ p \ \text{by simp}
also have \root p (\text{real } x) \ p = x \ \text{using } p \ x \ \text{by force}
finally have \( x \leq (\text{start-value } x \ p) \ p \ \text{by presburger}
\) with \text{start show } \text{thesis by auto}
qed

lemma \text{start-value: assumes } x: x \geq 0 \ \text{and } p: p > 0 \ \text{shows } x \leq (\text{start-value } x \ p) \ p \ \text{start-value } x \ p \geq 0
\) using \text{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 \( \text{op } / \) is replaced by \( \text{op div} \). We are proving several things within a locale which ensures that \( p > 0 \), and where \( pm = p - 1 \).

locale \text{fixed-root} =
  fixes \ p pm :: nat
  assumes \ p: p = Suc pm
begin

function \text{root-newton-int-main} :: \text{int } \Rightarrow \text{int } \Rightarrow \text{int } \Rightarrow \text{int } \Rightarrow \text{int} \Rightarrow \text{int } \times \text{bool} \ where
\text{root-newton-int-main } x n = (\text{if } x < 0 \lor n < 0 \ then \ (0,\text{False}) \ else \ (\text{if } x \ p \leq n \ then \ (x, x \ p = n)) \ else \ \text{root-newton-int-main} ((n \text{ div } (x \ pm) + x \text{ int } pm) \text{ div}(\text{int } p) \ n))
\) by pat-completeness auto
end

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

partial-function \ (\text{tailrec}) \text{root-int-main’} :: \text{nat } \Rightarrow \text{int } \Rightarrow \text{int } \Rightarrow \text{int } \Rightarrow \text{int } \Rightarrow \text{int } \times \text{bool} \ where
\text{code}:: \text{root-int-main’ } pm ipm ip x n = (\text{let } xpm = x \ pm; xp = xpm \times x \ \text{in } xp \leq n \ then \ (x, xp = n) \ else \ \text{root-int-main’ } pm ipm ip ((n \text{ div } xpm + x \text{ int } ipm) \text{ div}(\text{int } p) \ n)
\)

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

definition \text{root-int-main} :: \text{nat } \Rightarrow \text{int } \Rightarrow \text{int } \times \text{bool} \ where
\text{root-int-main } p n \equiv \text{if } p = 0 \ then \ (1, n = 1) \ else
\text{let } pm = p - 1
\text{ in } \text{root-int-main’ } pm (\text{int } pm) (\text{int } p) (\text{start-value n p}) n
\)

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

definition \text{root-int} :: \text{nat } \Rightarrow \text{int } \Rightarrow \text{int list} \ where
\text{root-int } p x \equiv \text{if } p = 0 \ then \ [] \ else
\text{if } x = 0 \ then \ [0] \ else
\text{let } e = \text{even } p; s = \text{sgn } x; x’ = \text{abs } x
\text{ in } \text{if } x < 0 \ \land \ e \ then \ [] \ else \ \text{case} \ \text{root-int-main } p x’ \ of \ (y,\text{True}) \Rightarrow \text{if } e \ then \ [y, -y] \ else \ [s \times y] \ | - \Rightarrow []
We start with proving termination of fixed-root.root-newton-int-main.

context fixed-root
begin

lemma iteration-mono-eq: assumes $x^n = (n :: int)$
  shows $(n \div x^p + x \ast int p) \div int p = x$
proof –
  have [simp]: $\land \; n \cdot (x + x \ast n) = x \ast (1 + n)$ by (auto simp: field-simps)
show ?thesis unfolding $x^n$[symmetric] $p$ by simp
qed

lemma p0: $p \neq 0$ unfolding $p$ by auto

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

lemma iteration-mono-less: assumes $x \geq 0$
  and $n \geq 0$
  and $x^n > (n :: int)$
  shows $(n \div x^p + x \ast int p) \div int p < x$
proof –
  let $?sx = (n \div x^p + x \ast int p) \div int p$
from $x^n$ have $x^n \leq x$ by auto
from $x^n \ast n$ have $x0: x > 0$
  using not-le $p$ by fastforce
from $p$ have $xp: x^p = x \ast x^p$ by auto
have $n \div x^p \ast x^p \leq n$ unfolding mod-div-equality-int
  using transfer-nat-int-function-closures $x \ast n$
by simp
also have $\ldots \leq x^p$ using $x^n$ by auto
finally have le: $n \div x^p \leq x$ using $x^n \ast 0$ unfolding $xp$ by simp
have $?sx \leq (x^p \div x^p \ast x \ast int pm) \div int p$
  by (rule zdv-monot, insert le p0, unfold $xp$, auto)
also have $x^p \div x^p = x$ unfolding $xp$ by auto
also have $x + x \ast int pm = x \ast int p$ unfolding $p$ by (auto simp: field-simps)
also have $x \ast int p \div int p = x$ using $p$ by force
finally have le: $?sx \leq x$.
{
  assume $?sx = x$
  from arg-cong[OF this, of $\lambda \; x. \; x \ast int p$]
  have $x \ast int p \leq (n \div x^p + x \ast int pm) \div int p \ast int p$ using p0 by simp
  also have $\ldots \leq n \div x^p + x \ast int pm$
    unfolding mod-div-equality-int using $p$ by auto
finally have $n \div x^p \geq x$ by (auto simp: field-simps)
  from mult-right-monot[OF this, of $x^p$]
  have $ge: n \div x^p \ast x^p \geq x^p$ unfolding $xp$ by auto
  from mod-div-equality[of $n \ast x^p$] have $n \div x^p \ast x^p = n - n mod x^p$ by arith
  from $ge$[unfolded this]
  have le: $x^p \leq n - n mod x^p$.
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 ≥ 0 → n ≥ 0 →
  root-main' x n = root-newton-int-main x n ∧ (root-main' x n = (y,b) → y ≥ 0
\[ y^{p} \leq n \land b = (y^{p} = n) \]

**proof** (induct \(x, n\) rule: root-newton-int-main.induct)

- **case** (\(1 \ x \ n\))
  
  - **have** pm-x[simp]: \(x \ ^{p} x = x \ ^{p}\) unfolding \(p\) by simp
  
  - **from** \(1\) **have** id: \((x < 0 \lor n < 0) = False\) by auto

**note** \(d = \text{root-main}'-\text{simps}[of } x, n]\) root-newton-int-main.simps[of \(x, n\)] id if-False

Let-def

**show** ?case

- **proof** (cases \(x \ ^{p} \leq n\))

  - **case** True
    
    - **thus** ?thesis unfolding \(d\) using \(1(2)\) by auto

  - **next**
    
    - **case** False
      
      - **hence** id: \((x \ ^{p} \leq n) = False\) by simp
      
      - **from** \(1(3)\) **have** not: \(\neg (x < 0 \lor n < 0)\) by auto

    **show** ?thesis unfolding \(d\) id pm-x
      
      - **by** (rule \(1(1)[OF not False - 1(3)]\), unfold \(p\), insert \(1(2), 3\),
        
        - **metis** Divides.transfer-nat-int-function-closures(1) add-nonneg-nonneg of-nat-0-le-iff
        
        split-mult-pos-le zero-le-power)

    **qed**

**qed**

**lemma** root-main': \(x \geq 0 \implies n \geq 0 \implies \text{root-main'} x n = \text{root-newton-int-main} x n\)

- **using** root-main'-newton-pos by blast

**lemma** root-main'-pos: \(x \geq 0 \implies n \geq 0 \implies \text{root-main'} x n = (y, b) \implies y \geq 0\)

- **using** root-main'-newton-pos by blast

**lemma** root-main'-sound: \(x \geq 0 \implies n \geq 0 \implies \text{root-main'} x n = (y, b) \implies b = (y \ ^{p} = n)\)

- **using** root-main'-newton-pos by blast

In order to prove completeness of the algorithms, we provide sharp upper and lower bounds for \(\text{root-main'}\). 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: \(x \geq 0 \implies n \geq 0 \implies \text{root-main'} x n = (y, b) \implies y \ ^{p} \leq n\)

- **using** root-main'-newton-pos by blast

**lemma** root-newton-int-main-upper:

- **shows** \(y \ ^{p} \geq n \implies y \geq 0 \implies n \geq 0 \implies \text{root-newton-int-main} y n = (x, b) \implies n < (x + 1) \ ^{p}\)

**proof** (induct \(y, n\) rule: root-newton-int-main.induct)

- **case** (\(1 y \ n\))
  
  - **from** \(1(3)\) **have** \(y0\): \(y \geq 0\)
  
  - **from** \(1(4)\) **have** \(n0\): \(n \geq 0\)

  **def** \(y' = (n \ div (y \ ^{pm}) + y * \ int pm) \ div (int p)\)
from y0 n0 have y'0: y' ≥ 0 unfolding y'-def
  by (metis Divides.transfer-nat-int-function-closures(1) add-increasing nonneg-int-cases
  zero-zle-int zmult-int zpower-int)
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)
case False
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 yyn0: y ≠ 0 unfolding p by auto
from y00 y0 have y00: ?y > 0 by auto
from n0 False have n0: ?n > 0 by auto
def Y ≡ ?y * of-int pm
def 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: \( \sum \) 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 \( \ldots < \) mean ?ls
proof (rule CauchysMeanTheorem-Less[OF pos het-gt-0I])
show NY \in set ?ls by simp
from pm0 show ?y \in 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 \neq ?y by blast
qed
also have \( \ldots = \) \((NY + Y) / real p\)
by (simp add: mean-def sum p)
finally have \( \ast: \) 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 \( \ldots < \) \((NY + Y) / real p\)^p
by (rule power-strict-mono[OF \( \ast \)], insert n0 p, auto)
finally have ineq1: ?n < \((NY + Y) / real p\)`^p by auto
{ 
def s \equiv n div y ^ pm + y * int pm
def S \equiv NY + Y
have Y0: Y \geq 0 using y0 unfolding Y-def
by (metis 1.prems(2) mult-nonneg-nonneg of-int-0-le-iff zero-zle-int)

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: \( \forall x_0. \) rat-of-int [rat-of-nat x_0] = rat-of-nat x_0
using of-int-of-nat-eq by simp
have f2: \( \forall x_0. \) real-of-int [rat-of-nat x_0] = real x_0
using of-int-of-nat-eq real-eq-of-nat by auto
have f3: \( \forall x_0 x_1. \) [rat-of-int x_0 / rat-of-int x_1] = [real-of-int x_0 / real-of-int x_1]
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
  using f1 f3 f4 by (metis div-is-floor-divide-real zdiv-mono1)
hence [S / real p] ≤ real-of-int (s div int p) + 1
  using f1 f2 f3 f4 by (metis div-is-floor-divide-real floor-div-pos-int)
hence S / real p ≤ of-int (s div 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 ineq1: (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 add-le-less-mono add-less-cancel-left lessI less-add-one monoid-add-class.
ordered-cancel-comm-monoid-diff-class.le-iff-add power-strict-mono)
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 monoid-add-class.add.
right-neutral ordered-cancel-comm-monoid-diff-class.le-iff-add power-strict-mono)
qed
qed

lemma root-main'-upper:
  x ^ p ≥ n ⇒ x ≥ 0 ⇒ n ≥ 0 ⇒ root-main' x n = (y,b) ⇒ n < (y + 1) ^ 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 ≥ 0
  and rm: root-int-main p n = (y,b)
  shows y ≥ 0 ∧ b = (y ^ p = n) ∧ (p > 0 ⇒ y ^ p ≤ n ∧ n < (y + 1) ^ p)
\( \land (p > 0 \rightarrow x \geq 0 \rightarrow x \leq p = n \rightarrow y = x \land b) \)

proof (cases \( p = 0 \))
  
case True
    
with \( \text{rm[unfolded root-int-main-def]} \)
  
have \( y = 1 \) and \( b \colon b = (n = 1) \) by auto
next

  
proof
    
shows \( \text{start-value from root-main from root-main} \)

  
case True
    
from \( \text{start-value[OF n p-0]} \) have start: \( n \leq (\text{start-value n p}) \cdot p \leq \text{start-value n p} \) by auto
interpret fixed-root \( p p = 1 \)

  
by (unfold-locales, insert False, auto)
from root-main-p-pos[\( \text{OF start(2) n rm} \)] have \( y = y \geq 0 . \)
from root-main-p-sound[\( \text{OF start(2) n rm} \)] have \( b \colon b = (y \cdot p = n) . \)
from root-main-p-upper[\( \text{OF start n rm} \)] have \( \text{up} \colon n < (y + 1) \cdot p . \)

  
\{  
  assume \( n \cdot x \cdot p = n \) and \( x \geq 0 \)
  with \( \text{low up} \) have low: \( y \cdot p \leq x \cdot p \) and \( \text{up} : x \cdot p < (y + 1) \cdot p \) by auto
  from power-strict-mono[\( \text{of x y, OF - x p-0} \)] low have \( x : x \geq y \) by arith
  from power-mono[\( \text{of (y + 1) x p} \)] y up have \( y : y \geq x \) by arith
  from \( x y \) have \( x = y \) by auto
  with \( b n \)
  have \( y = x \land b \) by auto
  \}
thus \( \text{thesis using b low up y by auto} \)

  
qed

lemma root-int-main: assumes \( n : n \geq 0 \)
  and \( \text{rm: root-int-main p n = (y,b)} \)
  shows \( y \geq 0 b = (y \cdot p = n) p > 0 \Longrightarrow y \cdot p \leq n \quad p > 0 \Longrightarrow x \geq 0 \Longrightarrow x \cdot p = n \Longrightarrow y = x \land b \) using root-int-main-all[\( \text{OF n rm, of x} \)] by blast+

lemma root-int[simp]: assumes \( p \colon p \neq 0 \lor x \neq 1 \)
  shows set (root-int \( p x) = \{ y : y \cdot p = x \} \)
proof (cases \( p = 0 \))
  
case True
  
  with \( p \) have \( x \neq 1 \) by auto
  thus \( \text{thesis unfolding root-int-def True by auto} \)
next

  
  case False
  
hence \( (p = 0) = \text{False and p0: p > 0} \) by auto
  note \( d = \text{root-int-def p if-False Let-def} \)
show \( \text{thesis} \)
proof (cases \( x = 0 \))
  case True
  thus \( \text{thesis} \) unfolding \( d \) using \( p \) by auto
next
  case False
  hence \( x : (x = 0) = \text{False} \) by auto
show \( \text{thesis} \)
proof (cases \( x < 0 \land \text{even} \ p \))
  case True
  hence \( \text{left} : \{ \text{set (root-int \( p \) \( x \))} = {} \} \) unfolding \( d \) by auto
  { fix \( y \)
    assume \( x : y \hat{\cdot} p = x \)
    with True have \( y \hat{\cdot} p < 0 \land \text{even} \ p \) by auto
    hence \( \text{False} \) by presburger
  }
  with \( \text{left} \) show \( \text{thesis} \) by auto
next
  case False
  with \( x \ p \) have \( \text{cond} : (x = 0) = \text{False} \) by auto
  obtain \( y \) \( b \) where \( \text{rt} : \text{root-int-main} \ p \mid x \mid = (y, b) \) by force
  have \( \text{abs} \ x \geq 0 \) by auto
  note \( \text{rm} = \text{root-int-main}[\text{OF this} \ \text{rt}] \)
  have \( \text{thesis} = (\{ \text{set (case root-int-main} \ p \mid x \mid \text{of} (y, \text{True}) \Rightarrow \text{if even} \ p \text{ then } [y, -y] \text{ else } [sgn} \ x \hat{\cdot} y] \mid (y, \text{False}) \Rightarrow []\}) = (\{y. y \hat{\cdot} p = x\}) \text{ unfolding } d \text{ cond by blast}
  also have \( \text{thesis} = (\{ \text{set (case root-int-main} \ p \mid x \mid \text{of} (y, \text{True}) \Rightarrow \text{if even} \ p \text{ then } [y, -y] \text{ else } [sgn} \ x \hat{\cdot} y] \text{ else } []\}) \text{ unfolding } \text{rt} \text{ by auto}
  also have \( \text{thesis} = (\{y. y \hat{\cdot} p = x\}) \text{ unfolding } \text{thesis} \)
proof –
  { fix \( z \)
    assume idz: \( z \hat{\cdot} p = x \)
    hence eq: \( (\text{abs} \ z) \hat{\cdot} p = \text{abs} \ x \) by (metis \text{power-abs})
    from \( idz \) \( p \) \( 0 \) have \( z : z \neq 0 \) unfolding \( p \) by auto
    have \( (y, b) = (|z|, \text{True}) \)
    using \( \text{rm}(5) \) \( \text{OF} \ p \) \( 0 \) \( -eq \) by auto
    hence id: \( y = \text{abs} \ z \) \( b = \text{True} \) by auto
    have \( z \in \text{set} \ ?lhs \) unfolding \( \text{id} \) using \( z \) by (auto \text{ simp: idz[ symmetric]}, \text{ cases} \( z < 0, \) \text{ auto} )
  }
  moreover
  { fix \( z \)
    assume \( z : z \in \text{set} \ ?lhs \)
hence $b$: $b = True$ by (cases $b$, auto)

note $z = z[unfolded b if-True]

from $\text{rm}(2)$ have $yx$: $y = p = |x|$ by auto

from $\text{rm}(1)$ have $y$: $y \geq 0$.

from False have odd $p \lor even$ $p \land x \geq 0$ by auto

hence $z \in \text{?rhs}$

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 \ldots = $sgn \ x \ ^\ ^\ p \ y \ ^\ ^\ p$ unfolding power-mult-distrib by auto

also have \ldots = $sgn \ x \ ^\ ^\ p \ abs \ x$ unfolding $yx$ by simp

also have $sgn \ x \ ^\ ^\ p = sgn \ x$ using $x$ odd by auto

also have $sgn \ x \ y \ ^\ ^\ p$ unfolding power-mult-distrib by simp

also have \ldots = $sgn \ x \ y \ ^\ ^\ p$ unfolding $yx$ by simp

also have $sgn \ x = 1$ using even $z$ by auto

finally show $z \in \text{?rhs}$ by auto

next

assume even: even $p \land x \geq 0$

from $z$ even have $z = y \lor z = -y$ by auto

hence id: $abs \ z = y$ using $y$ by auto

with $yx \ x$ even have $z$: $z \neq 0$ using $p0$ by (cases $y = 0$, auto)

have $z \ ^\ ^\ p = (sgn \ z \ abs \ z) \ ^\ ^\ p$ by (simp add: mult-sgn-abs)

also have \ldots = $(sgn \ z \ y) \ ^\ ^\ p$ using id by auto

also have \ldots = $(sgn \ z) \ ^\ ^\ p \ y \ ^\ ^\ p$ unfolding power-mult-distrib by simp

also have \ldots = $(sgn \ z) \ ^\ ^\ p \ y \ ^\ ^\ p$ unfolding $yx$ using even by auto

also have $sgn \ z \ ^\ ^\ p = 1$ using even $z$ by (auto)

finally show $z \in \text{?rhs}$ by auto

qed
hence \((x = 0) = False \ (x < 0 \land \text{even } p) = False\) using \(x\) by auto

note \(ri = ri[\text{unfolded this if-False}]\)

obtain \(y' \ b'\) where \(r: \text{root-int-main } p \ x = (y', b')\) by force

note \(ri = ri[\text{unfolded this}]\)

hence \(y: y = (\text{if even } p \text{ then } y' \text{ else } \text{sgn } x \times y')\) by (cases \(b'\), auto)

thus \(\text{thesis}\) unfolding \(y\) using \(x\ False\) by auto

qed

2.3 Floor and ceiling of roots

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

definition \(\text{root-int-floor-pos } p \ x = (\text{if } p = 0 \text{ then } 0 \text{ else } \text{fst } (\text{root-int-main } p \ x))\)

definition \(\text{root-int-ceiling-pos } p \ x = (\text{if } p = 0 \text{ then } 0 \text{ else } (\text{case } \text{root-int-main } p \ x \text{ of } (y, b) \Rightarrow \text{if } b \text{ then } y \text{ else } y + 1))\)

lemma \(\text{root-int-floor-pos-lower}: \assumes p0: p \neq 0 \text{ and } x: x \geq 0\)

shows \(\text{root-int-floor-pos } p \ x \land p \leq x\)

using \(\text{root-int-main}(3)[OF x, of p]\) unfolding \(\text{root-int-floor-pos-def}\)

by (cases \(\text{root-int-main } p \ x\), auto)

lemma \(\text{root-int-floor-pos-pos}: \assumes x: x \geq 0\)

shows \(\text{root-int-floor-pos } p \ x \geq 0\)

using \(\text{root-int-main}(1)[OF x, of p]\)

unfolding \(\text{root-int-floor-pos-def}\)

by (cases \(\text{root-int-main } p \ x\), auto)

lemma \(\text{root-int-floor-pos-upper}: \assumes p0: p \neq 0 \text{ and } x: x \geq 0\)

shows \(\text{root-int-floor-pos } p \ x + 1 \land p > x\)

using \(\text{root-int-main}(4)[OF x, of p]\) unfolding \(\text{root-int-floor-pos-def}\)

by (cases \(\text{root-int-main } p \ x\), auto)

lemma \(\text{root-int-floor-pos}: \assumes x: x \geq 0\)

shows \(\text{root-int-floor-pos } p \ x = \text{floor } (\text{root } p \ (\text{real } x))\)

proof (cases \(p = 0\))

\(\text{case True}\)

thus \(\text{thesis}\) by (simp add: \(\text{root-int-floor-pos-def}\))

next

\(\text{case False}\)

hence \(p: p > 0\) by auto

let \(\text{?s1} = \text{real-of-int } (\text{root-int-floor-pos } p \ x)\)

let \(\text{?s2} = \text{root } p \ (\text{real } x)\)

from \(x\) have \(\text{?s1} \geq 0\)

by (metis \(\text{of-int-0-le-iff}\) \(\text{root-int-floor-pos-pos}\))

from \(x\) have \(\text{?s2} \geq 0\)

by (metis \(\text{real-of-int-ge-zero-cancel-iff}\) \(\text{real-root-pos-pos-le}\))
from $s_1$ have $s_{11} + 1 ≥ 0$ by auto
have id: $s_2 ∗ p = \text{real } x$ using $x$
  by (metis $p$ of-int-power real-eq-of-int)
show ?thesis
proof (rule floor-unique[symmetric])
  show $s_1 ≤ s_2$
    unfolding compare-pow-le-iff[OF $p$ $s_1$ $s_2$, symmetric]
    unfolding id
    using root-int-floor-pos-lower[OF False $x$]
    by (metis of-int-power real-eq-of-int)
  show $s_2 < s_1 + 1$
    unfolding compare-pow-less-iff[OF $p$ $s_2$ $s_{11}$, symmetric]
    unfolding id
    using root-int-floor-pos-upper[OF False $x$]
    by (metis real-of-int-le-iff real-of-int-zero-cancel)
qed

lemma root-int-ceiling-pos: assumes $x$: $x ≥ 0$
  shows root-int-ceiling-pos $p$ $x = \text{ceiling } (\text{root } p (\text{real } x))$
proof (cases $p = 0$)
  case True
  hence $s$: $s = \text{root-int-main } p$ $x = (y, b)$ 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 $s = \text{root-int-ceiling-pos } p$ $x$
  let $ss = \text{root } p (\text{real } x)$
  note d = root-int-ceiling-pos-def
  show ?thesis
proof (cases $b$)
  case True
  hence $s = \text{y unfolding } s$ $d$ using $p$ by auto
  from rm(2) True have xy: $x = y ∗ p$ by auto
  show ?thesis unfolding id unfolding xy using $y$
    by (metis ceiling-real-of-int $p$ real-of-int-power real-of-int-zero-cancel)
next
  case False
  hence $id$: $s = \text{root-int-floor-pos } p$ $x + 1$ unfolding $d$ root-int-floor-pos-def
    using $s$ $p$ by simp
  from False have $x0$: $x ≠ 0$ using rm(5)[of $0$] 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 \leq real-of-int \floor{\root{p}{\real{x}}} + 1
  by (metis real-of-int-add real-of-int-def real-of-int-floor-add-one-ge real-of-one)
let \lm = real-of-int \floor{\root{p}{\real{x}}}
have \l = \lm by simp
also have \ldots < \sx
proof
  have le: \m \leq \sx by (rule of-int-floor-le)
  have neq: \m \neq \sx
  proof
    assume \m = \sx
    hence \m \cdot p = \sx \cdot p by simp
    also have \ldots = real 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 = \floor{\root{p}{\real{x}}} \cdot p
    by (metis floor-power floor-real-of-int real-of-int-floor)
  hence \floor{\root{p}{\real{x}}} \in \set{\root-int p x} using p by simp
  hence root-int p x \neq \[-] by force
  with s False \p \neq 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

definition root-int-floor p x = (if x \geq 0 then root-int-floor-pos p x else \ - root-int-ceiling-pos p \ (- x))
definition root-int-ceiling p x = (if x \geq 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 \ (real x))
proof
  note d = root-int-floor-def
show \thesis
  proof (cases x \geq 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)
lemma root-int-ceiling [simp]: root-int-ceiling p x = ceiling (root p (real x))
proof
  note d = root-int-ceiling-def
  show ?thesis
  proof (cases x ≥ 0)
    case True
    with root-int-ceiling-pos [OF True] show ?thesis unfolding d by simp
  next
    case False
    hence − x ≥ 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

2.4 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 ≠ 0
  shows root-nat-floor p x ° p ≤ 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 ≠ 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 ≠ 0 ∨ x ≠ 1
  shows set (root-nat p x) = { y. y ° p = x}
  proof
    {
fix \( y \)
assume \( y \in \text{set} \ (\text{root-nat} \ p \ x) \)
note \( y = \text{this[unfolded root-nat-def]} \)
then obtain \( y_i \ y_s \) where \( r_i: \text{root-int} \ p \ x = y_i \# y_s \) by (cases root-int \( p \ x \), auto)

with \( y \) have \( y = \text{nat} \ y_i \) by auto
from root-int-pos[\( OF - r_i \)] have \( y_i: 0 \leq y_i \) by auto
from root-int[of \( p \ \text{int} \ x \)] \( p0 \ r_i \) have \( y_i \# p = x \) by auto
from arg-cong[\( OF \ this, \ of \ nat \) \( y_i \) have \( \text{n}at \ y_i \# p = x \)
by (metis \( nat \-\text{int} \ nat\-\text{power-eq} \)

hence \( y \in \{ y. \ y \# p = x \} \) using \( y \) by auto

} moreover

{ fix \( y \)
assume \( yx: y \# p = x \)
hence \( y: \text{int} \ y \# p = \text{int} \ x \)
by (metis \( of\-\text{nat}\-\text{power} \)

hence \( \text{set} \ (\text{root-int} \ p \ (\text{int} \ x)) \neq \{ \} \) using \( \text{root-int[of} \ p \ \text{int} \ x \] \( p0 \)
by (metis (mono-ta) \( \text{One-nat-def} \ (y \# p = x) \emptyset\-\text{Collect-eq} \text{nat\-power-eq} \text{Suc}\-\text{0-iff} \)
then obtain \( y_i \ y_s \) where \( r_i: \text{root-int} \ p \ (\text{int} \ x) = y_i \# y_s \) by (cases root-int \( p \ (\text{int} \ x) \), auto)

from root-int-pos[\( OF - \text{this} \)] have \( y_i: y \geq 0 \) by auto
from root-int[of \( p \ \text{int} \ x \), unfolded \( r_i \)] \( p0 \) have \( y_i \# p = \text{int} \ x \) by auto
with \( y \) have \( \text{int} \ y \# p = y_i \# p \) by auto
from arg-cong[\( OF \ this, \ of \ nat \) \( y_i \) have \( \text{id} \ (y \# p = \text{nat} \ y_i \# p \)
by (metis \( y \# p = x \) \( \text{nat\-int} \ \text{nat\-power-eq} \ y_i \)

\}
assume \( p: p \neq 0 \)
hence \( p0: p > 0 \) by auto
obtain \( yy \ b \) where \( rm: \text{root-int-main} \ p \ (\text{int} \ x) = (yy,b) \) by force
from root-int-main[5][\( OF - r m \ p0 - y \)] have \( yy = \text{int} \ y \) and \( b = \text{True} \) by auto

note \( rm = \text{rm[unfolded this]} \)
hence \( y \in \text{set} \ (\text{root-nat} \ p \ x) \)

unfolding \( \text{root-nat-def} \ p \ \text{root-int-def} \) using \( p0 \ p \ yx \)
by auto

} moreover

{ assume \( p: p = 0 \)
with \( p0 \) have \( x \neq 1 \) by auto
with \( y \ p \) have \( \text{False} \) by auto

} ultimately have \( y \in \text{set} \ (\text{root-nat} \ p \ x) \) by auto

} ultimately show \( \text{thesis} \) by blast
qed
2.5 Upgrading algorithms to the rationals

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

**Definition** \( \text{root-rat-floor} :: \text{nat} \Rightarrow \text{rat} \Rightarrow \text{int} \) where
\[
\text{root-rat-floor} \ p \ x \equiv \text{case quotient-of} \ x \ of \ (a,b) \Rightarrow \text{root-int-floor} \ p \ (a \ast b^{-(p-1)}) \ div \ b
\]

**Definition** \( \text{root-rat-ceiling} :: \text{nat} \Rightarrow \text{rat} \Rightarrow \text{int} \) where
\[
\text{root-rat-ceiling} \ p \ x \equiv -\left(\text{root-rat-floor} \ p \ (-x)\right)
\]

**Definition** \( \text{root-rat} :: \text{nat} \Rightarrow \text{rat} \Rightarrow \text{rat list} \) where
\[
\text{root-rat} \ p \ x \equiv \text{case quotient-of} \ x \ of \ (a,b) \Rightarrow \text{concat} \left(\text{map} \ (\lambda \text{ra}. \text{of-int ra} / \text{rat-of-int rb}) \left(\text{root-int} \ p \ a\right)\right) \left(\text{take} \ 1 \left(\text{root-int} \ p \ b\right)\right)
\]

**Lemma** \( \text{root-rat-reform} \): assumes \( q \): quotient-of \( x \) = \((a,b)\) shows \( \text{root} \ p \ (\text{real-of-rat} \ x) = \text{root} \ p \ (\text{of-int} \ (a \ast b^{-(p-1)}) \div \text{of-int} \ b) \)
proof (cases \( p = 0 \))
\[
\begin{align*}
\text{case False} \\
& \text{from quotient-of-denom-pos}[OF} \ q | \text{have b: 0 < b by auto} \\
& \text{hence b: 0 < real b by auto} \\
& \text{from quotient-of-div}[OF} \ q | \text{have x: root} \ p \ (\text{real-of-rat} \ x) = \text{root} \ p \ (a / b) \\
& \text{by (metis of-rat-divide of-rat-of-int-eq real-of-int-def)} \\
& \text{also have a / b = a * real b ^ (p - 1) / real b ^ p using b False} \\
& \text{by (cases p, auto simp: field-simps)} \\
& \text{also have root p ... = root} \ p \ (a * real b ^ (p - 1)) / root p (real b ^ p) \text{ by (rule real-root-divide)} \\
& \text{also have root p (real b ^ p) = of-int b using False b} \\
& \text{by (metis real-rat-pow-pos real-root-power real-eq-of-int)} \\
& \text{also have a * real b ^ (p - 1) = of-int (a * b ^ (p - 1))} \\
& \text{by (metis real-of-int-def real-of-int-mult real-of-int-power)} \\
\end{align*}
\]
finally show \( \text{thesis} \).
qed auto

**Lemma** \( \text{root-rat-floor} \ [simp]: \text{root-rat-floor} \ p \ x = \text{floor} \ (\text{root} \ p \ (\text{of-rat} \ x)) \)
proof –
\[
\begin{align*}
& \text{obtain a b where} \ q: \text{quotient-of} \ x = (a,b) \text{ by force} \\
& \text{from quotient-of-denom-pos}[OF} \ q | \text{have b: b > 0 .} \\
& \text{show} \ \text{thesis} \\
& \text{unfolding} \ \text{root-rat-floor-def} \ q \ \text{split root-int-floor} \\
& \text{unfolding} \ \text{root-rat-reform}[OF} \ q | \\
& \text{unfolding} \ \text{floor-div-pos-int}[OF} \ b | \ \text{real-eq-of-int} \ .. \\
\end{align*}
\]
qed

**Lemma** \( \text{root-rat-ceiling} \ [simp]: \text{root-rat-ceiling} \ p \ x = \text{ceiling} \ (\text{root} \ p \ (\text{of-rat} \ x)) \)

unfolding
\[
\begin{align*}
& \text{root-rat-ceiling-def} \\
\end{align*}
\]
lemma root-rat[simp]: assumes p: p ≠ 0 ∧ x ≠ 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 ∈ set (root-rat p x)
    have b: b > 0 unfolding x by (cases root-int p a, auto)
    note mem = mem[unfolded this]
    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 ∈ set (root-int p a) and q: q = of-int ra / of-int rb
      by (cases root-int p a, auto)
    from ra have q ∈ { y. y ^ p = x} unfolding q x ra rb by (auto simp: power-divide of-int-power)
  }
moreover
  { fix q
    assume q ∈ { 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 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] 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 \((z \cdot p)\) \((n \cdot p)\)
  by (metis gcd-int.commute coprime-exp-int)
from coprime-crossproduct-int[OF quotient-of-coprime[OF q] this] arg-cong[OF id, of abs]
  have \(|n \cdot p| = |b|\)
  by (simp add: field-simps abs-mult)
with \(n\ b\) have bnp: \(b = n \cdot p\) by auto
hence rn: \(n \in \text{set} \ (\text{root-int} \ p\ b)\) using \(p\) by auto
then obtain \(rb\ rs\) where \(rb\): \(\text{root-int} \ p\ b = \text{Cons} \ rb\ rs\) by (cases \(\text{root-int} \ p\ b\), auto)
  from \(id[\text{folded bnp}]\) \(b\) have \(a = z \cdot p\) by auto
hence \(a\): \(z \in \text{set} \ (\text{root-int} \ p\ a)\) using \(p\) by auto
from root-int-pos[OF \(- rb\) \(b\)] have rb0: \(rb \geq 0\) by auto
from root-int[OF disjI1[OF \(p\), of \(b\)]] \(rb\) have \(rb \cdot p = b\) by auto
with bnp have id: \(rb \cdot p = n \cdot p\) by auto
have \(rb = n\) by (rule power-eq-imp-eq-base[OF id], insert \(n\ rb0\ p\), auto)
with \(rb\) have \(b\): \(n \in \text{set} \ (\text{take} \ 1 \ (\text{root-int} \ p\ b))\) by auto
have \(q\): \(\in \text{set} \ (\text{root-rat} \ p\ x)\) unfolding d qzn using \(b\ a\) by auto
}
ultimately show \(?thesis\) by blast
next
  case True
with \(p\) have \(x\): \(x \neq 1\) by auto
obtain \(a\ b\) where \(q\): \(\text{quotient-of} \ x = (a,b)\) 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

3 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.

26
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 certifying termination proofs. Here, non-linear equations of the form \((a_1x_1 + \ldots a_n x_n)^2 = p\) 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 \( a = \sqrt{4} = 2, b = -\sqrt{9} = -3\).

3.1 The Babylonian method

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

\[
x_{i+1} = \frac{n}{x_i} + x_i
\]

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 \( x_i \geq x_{i+1} \).

3.2 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 \( x \div y = \lfloor \text{of-int } x / \text{of-int } y \rfloor \). Note that replacing \( \lfloor \text{of-int } x / \text{of-int } y \rfloor \) by \( \lceil \text{of-int } x / \text{of-int } y \rceil \) would lead to non-termination in the following algorithm.

We explicitly 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 \texttt{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.

\[
\text{definition } \text{sqrt-int-main}' :: \text{int } \Rightarrow \text{int } \Rightarrow \text{int } \times \text{bool } \text{where}
\]

\[
[\text{simp}]: \text{sqrt-int-main}' x n = \text{root-int-main}' 1 1 2 x n
\]
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 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 \textit{sqrt-int-floor-pos}; \textbf{assumes} \ x: \ x \geq 0
\textbf{shows} \ \textit{sqrt-int-floor-pos} \ x = \lfloor \sqrt{(\text{real}\ x)} \rfloor
\textbf{using} \ \textit{root-int-floor-pos}(\text{OF} \ x, \ \text{of} \ 2) \ \textbf{by} \ (\text{simp add: sqrt-def})

definition \textbf{[simp]}: \ \textit{sqrt-int-ceiling-pos} \ x = \textit{root-int-ceiling-pos} \ 2 \ x

lemma \textit{sqrt-int-ceiling-pos-code}[\text{code}]: \ \textit{sqrt-int-ceiling-pos} \ x = (\text{case} \ \textit{sqrt-int-main} \ x \ of \ (y,b) \ \Rightarrow \ \text{if} \ b \ \text{then} \ y \ \text{else} \ y + 1)
\textbf{by} \ (\text{simp add: root-int-ceiling-pos-def})

lemma \textit{sqrt-int-ceiling-pos}; \textbf{assumes} \ x: \ x \geq 0
\textbf{shows} \ \textit{sqrt-int-ceiling-pos} \ x = \lceil \sqrt{(\text{real}\ x)} \rceil
\textbf{using} \ \textit{root-int-ceiling-pos}(\text{OF} \ x, \ \text{of} \ 2) \ \textbf{by} \ (\text{simp add: sqrt-def})

definition \ \textit{sqrt-int-floor} \ x = \textit{root-int-floor} \ 2 \ x

lemma \textit{sqrt-int-floor-code}[\text{code}]: \ \textit{sqrt-int-floor} \ x = (\text{if} \ x \geq 0 \ \text{then} \ \textit{sqrt-int-floor-pos} \ x \ \text{else} \ - \ \textit{sqrt-int-ceiling-pos} (-x))
\textbf{unfolding} \ \textit{sqrt-int-floor-def} \ \textit{root-int-floor-def} \ \textbf{by} \ \text{simp}

lemma \textit{sqrt-int-floor}[\text{simp}]: \ \textit{sqrt-int-floor} \ x = \lfloor \sqrt{(\text{real}\ x)} \rfloor
\textbf{by} \ (\text{simp add: sqrt-int-floor-def sqrt-def})

definition \ \textit{sqrt-int-ceiling} \ x = \textit{root-int-ceiling} \ 2 \ x

lemma \textit{sqrt-int-ceiling-code}[\text{code}]: \ \textit{sqrt-int-ceiling} \ x = (\text{if} \ x \geq 0 \ \text{then} \ \textit{sqrt-int-ceiling-pos} \ x \ \text{else} \ - \ \textit{sqrt-int-floor-pos} (-x))
\textbf{unfolding} \ \textit{sqrt-int-ceiling-def} \ \textit{root-int-ceiling-def} \ \textbf{by} \ \text{simp}

lemma \textit{sqrt-int-ceiling}[\text{simp}]: \ \textit{sqrt-int-ceiling} \ x = \lceil \sqrt{(\text{real}\ x)} \rceil
\textbf{by} \ (\text{simp add: sqrt-int-ceiling-def sqrt-def})

3.3 Square roots for the naturals

definition \ \textit{sqrt-nat} :: \ \textbf{nat} \ \Rightarrow \ \textbf{nat} \ \textbf{list}
\textbf{where} \ \textit{sqrt-nat} \ x = \textit{root-nat} \ 2 \ x

lemma \textit{sqrt-nat-code}[\text{code}]: \ \textit{sqrt-nat} \ x \equiv \textbf{map} \nat \ (\text{take} \ 1 \ (\textit{sqrt-int} \ \textit{int} \ x))
\textbf{unfolding} \ \textit{sqrt-nat-def} \ \textit{root-nat-def} \ \textit{sqrt-int-def} \ \textbf{by} \ \text{simp}

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

definition \ \textit{sqrt-nat-floor} :: \ \textbf{nat} \ \Rightarrow \ \textbf{int}
\textbf{where} \ \textit{sqrt-nat-floor} \ x = \textit{root-nat-floor} \ 2 \ x

lemma \textit{sqrt-nat-floor-code}[\text{code}]: \ \textit{sqrt-nat-floor} \ x = \textit{sqrt-int-floor-pos} \ (\textit{int} \ x)
\textbf{unfolding} \ \textit{sqrt-nat-floor-def} \ \textit{root-nat-floor-def} \ \textbf{by} \ \text{simp}
lemma sqrt-nat-floor[simp]: \( \text{sqrt-nat-floor } x = \lfloor \text{sqrt}(\text{real } x) \rfloor \) 
unfolding sqrt-nat-floor-def by (simp add: sqrt-def)

definition sqrt-nat-ceiling :: nat \Rightarrow int where 
\( \text{sqrt-nat-ceiling } x = \text{root-nat-ceiling } 2 \times x \)

lemma sqrt-nat-ceiling-code[code]: \( \text{sqrt-nat-ceiling } x = \lfloor \text{sqrt}(\text{real } x) \rfloor \) 
unfolding sqrt-nat-ceiling-def by simp

lemma sqrt-nat-ceiling[simp]: \( \text{sqrt-nat-ceiling } x = \lceil \text{sqrt}(\text{real } x) \rceil \) 
unfolding sqrt-nat-ceiling-def by (simp add: sqrt-def)

3.4 Square roots for the rationals
definition sqrt-rat :: rat \Rightarrow rat list where 
\( \text{sqrt-rat } x = \text{root-rat } 2 \times x \)

lemma sqrt-rat-code[code]: \( \text{sqrt-rat } x = \left( \text{case quotient-of } x \text{ of } (z,n) \Rightarrow (\text{case sqrt-int } n \text{ of } []) \Rightarrow [] \mid sn \# xs \Rightarrow \text{map } (\lambda sz. \text{of-int } sz / \text{of-int } sn) (\text{sqrt-int } z) \right) \) 
proof 
  obtain z n where q: \text{quotient-of } x = (z,n) by force 
  show ?thesis 
  unfolding sqrt-rat-def \text{root-rat-def } q \text{split } sqrt-int-def 
  by (cases root-int 2 n, auto) 
qed

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

lemma sqrt-rat-pos: assumes sqrt: \( \text{sqrt-rat } x = \text{Cons } s \text{ ms} \) 
shows s \geq 0 
proof 
  obtain z n where q: \text{quotient-of } x = (z,n) by force 
  note sqrt = \text{sqrt}[unfolded sqrt-rat-code q, simplified] 
  let ?sz = sqrt-int z 
  let ?sn = sqrt-int n 
  from q have n: n \geq 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 sqrt-rat-of-int: assumes x: x * x = rat-of-int i
  shows ∃ j :: int. j * j = i
  proof −
    from x have mem: x ∈ 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 ∈ rat-of-int ‘ {y. y * y = i} by auto
    thus ?thesis by auto
  qed

3.5 Approximating square roots

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

We first provide the executable version without guard \((\theta)\! ":\!'a\! <\! x\) 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 \( \epsilon \) and positive \( n \). We require sort floor-ceiling, since \([x]\) 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 \( \frac{x^2}{n} - 1 \) 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 –
def er ≡ λ x. (x * x / n - 1)
def c ≡ 2 * n / ε
def m ≡ λ x. nat ⌊ c * er 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
def 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 enxen 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

32
by (auto simp: field-simps)
also have \ldots = \frac{er x}{4} using erx by (simp add: field-simps)
finally have \frac{er y - x}{4} \leq \frac{er x}{4} by linarith
from erxen have \(c \times er x \geq 2\) unfolding \(c\text{-def mult-le-cancel-left-pos}\)[OF ne, of - er x, symmetric]
using n \in \mathbb{N} by (auto simp: field-simps)
hence pos: \(c \times er x > 0\) \(c \times er x \geq 2\) by auto
show \(m y < m x\) unfolding m-def nat-mono-iff[OF pos(1)]
proof -
  have \(\lfloor c \times er y \rfloor \leq \lfloor c \times \frac{er x}{4} \rfloor\)
    by (rule floor-mono, unfold mult-le-cancel-left-pos[OF c], rule er-y-x)
  also have \ldots < \(\lfloor c \times \frac{er x + 1}{4} \rfloor\) by auto
  also have \ldots \leq \(\lfloor c \times er x \rfloor\)
    by (rule floor-mono, insert pos(2), simp add: field-simps)
finally show \([c \times er y] < [c \times er x]\).
qed
qed

Once termination is proven, it is easy to show equivalence of \(\sqrt{\text{approx-main-impl}}\)
and \(\sqrt{\text{approx-main}}\).

**Lemma** \(\sqrt{\text{approx-main-impl}}\): \(x > 0 \Rightarrow \sqrt{\text{approx-main-impl}} \varepsilon n x = \sqrt{\text{approx-main}}\)

**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 \(x \times x - n < \varepsilon\))
    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

Also soundness is not complicated.

**Lemma** \(\sqrt{\text{approx-main-sound}}\): assumes \(x: x > 0\) and \(xx: x \times x > n\)
shows \(\sqrt{\text{approx-main}} x \times \sqrt{\text{approx-main}} x > n \land \sqrt{\text{approx-main}} x \times \sqrt{\text{approx-main}} x - n < \varepsilon\)
using assms

**Proof** (induct \(x\) rule: sqrt-approx-main.induct)
  case (1 x)
  from 1 have \(x: x > 0\) (\(x > 0\)) = True by auto
\textbf{note} \quad \text{simp} = \text{sqrt-approx-main.simps}\{of \, x, \text{unfolded} \, x \text{ if-True}\]

\textbf{show} \quad \text{?case}

\textbf{proof} \quad (\text{cases} \, x \cdot x - n < \varepsilon)

\text{case True}

\quad with \ \text{?thesis unfolding simp by simp}

\textbf{next}

\text{case False}

\quad \text{let} \ ?y = (n / x + x) / 2

\quad \text{from False simp have simp: sqrt-approx-main x = sqrt-approx-main ?y by simp}

\quad \text{from n x have y: ?y > 0 by (auto intro: pos-add-strict)}

\quad \text{from x have x4: 4 \cdot x \cdot x > 0 by (auto intro: mult-sign-intros)}

\quad \text{show ?thesis unfolding simp}

\quad \text{proof (rule IH)}

\quad \text{show n < ?y * ?y}

\quad \text{unfolding mult-less-cancel-left-pos[OF x4, of n, symmetric]}

\quad \text{proof –}

\quad \quad \text{have id: 4 \cdot x \cdot x \cdot (?y * ?y) = 4 \cdot x \cdot x \cdot n + (n - x \cdot x) \cdot (n - x \cdot x) using x(1)}

\quad \quad \text{by (simp add: field-simps)}

\quad \quad \text{from 1(3) have x \cdot x - n > 0 by auto}

\quad \quad \text{from mult-pos-pos[OF this this]}

\quad \quad \text{show 4 \cdot x \cdot x \cdot n < 4 \cdot x \cdot x \cdot (?y * ?y) unfolding id}

\quad \quad \text{by (simp add: field-simps)}

\quad \quad \text{qed}

\quad \quad \text{qed}

\quad \quad \text{qed}

\quad \quad \text{qed}

\text{end}

It remains to assemble everything into one algorithm.

\textbf{definition} \quad \text{sqrt-approx} :: 'a :: \{linordered-field,floor-ceiling\} \Rightarrow 'a \Rightarrow 'a

\quad where

\quad sqrt-approx \ \varepsilon \ \text{if} \ \varepsilon > 0 \ \text{then} \ \text{if} \ \text{x} = 0 \ \text{then} 0 \ \text{else} \ \text{let} \ \text{xpos} = \text{abs} \ \text{x} \ \text{in}

\quad sqrt-approx-main-impl \ \varepsilon \ \text{xpos} (\text{xpos} + 1) \ \text{else} \ 0

\textbf{lemma} \quad \text{sqrt-approx: assumes} \ \varepsilon: \varepsilon > 0

\quad \text{shows} \ |\text{sqrt-approx} \ \varepsilon \ \text{x} \ \equiv \ |\text{sqrt-approx} \ \varepsilon \ \text{x} - |\text{x}|| < \varepsilon

\textbf{proof} \quad (\text{cases} \ x = 0)

\quad \text{case True}

\quad \text{with} \ \varepsilon \ \text{show} \ \text{?thesis unfolding sqrt-approx-def by auto}

\textbf{next}

\text{case False}

\quad \text{let} \ ?x = |\text{x}|

\quad \text{let} \ ?sqrti = \text{sqrt-approx-main-impl} \ ?x (\text{?x} + 1)

\quad \text{let} \ ?sqrt = \text{sqrt-approximation.sqrt-approx-main} \ ?x (\text{?x} + 1)

\text{def} \ \text{sqrt} \equiv \ ?sqrt
from False have \( ?x > 0 \) \( ?x + 1 > 0 \) by auto

interpret sqrt-approximation \( \varepsilon \) ?x

by (unfold-locales, insert \( x \varepsilon \), auto)

from False \( \varepsilon \) have sqrt-approx \( \varepsilon \) \( x \) unfolding sqrt-approx-def by (simp add: Let-def)

also have \( \varepsilon x = \varepsilon \) by (rule sqrt-approx-main-impl, auto)

finally have id: sqrt-approx \( \varepsilon \) \( x \) = sqrt unfolding sqrt-def .

have sqrt: sqrt \( \ast \) sqrt \( \geq \) \( ?x \) \& sqrt \( \ast \) sqrt \( \geq \) \( ?x \varepsilon \) 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 \( \varepsilon \)thesis unfolding id using sqrt by auto

qed

3.6 Some tests

Testing executability and show that sqrt 2 is irrational

lemma \( \neg (\exists i :: \text{rat}. i \ast i = 2) \)

proof –

have set (sqrt-rat 2) = {} by eval

thus \( \varepsilon \)thesis by simp

qed

Testing speed

lemma \( \neg (\exists i :: \text{int}. i \ast i = 1234567890123456789012345678901234567890) \)

proof –

have set (sqrt-int 1234567890123456789012345678901234567890) = {} by eval

thus \( \varepsilon \)thesis by simp

qed

The following test

value let \( \varepsilon = 1 / 100000000 :: \text{rat} \) ; \( s = \text{sqrt-approx} \varepsilon \) 2 in \( (s, s \ast s - 2, |s \ast s - 2| \varepsilon) \)

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