Computing N-th Roots using the Babylonian Method*

René Thiemann

February 23, 2021

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 A Fast Logarithm Algorithm 4

3 Executable algorithms for p-th roots 8
  3.1 Logarithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
  3.2 Computing the p-th root of an integer number . . . . . . . . . 8
  3.3 Floor and ceiling of roots . . . . . . . . . . . . . . . . . . . . 20
  3.4 Downgrading algorithms to the naturals . . . . . . . . . . . . . 23
  3.5 Upgrading algorithms to the rationals . . . . . . . . . . . . . . 25

4 Executable algorithms for square roots 27
  4.1 The Babylonian method . . . . . . . . . . . . . . . . . . . . . 28
  4.2 The Babylonian method using integer division . . . . . . . . . 28
  4.3 Square roots for the naturals . . . . . . . . . . . . . . . . . . 30
  4.4 Square roots for the rationals . . . . . . . . . . . . . . . . . . 31
  4.5 Approximating square roots . . . . . . . . . . . . . . . . . . . 32
  4.6 Some tests . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

*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: \( (n :: int) \div x \ast x = n - n \mod x \)
using div-mult-mod-eq[of n x] by arith

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

lemma div-is-floor-divide-real: \( n \div y = \lfloor \frac{\text{real-of-int } n}{\text{of-int } y} \rfloor \)
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 \( r :: 'a :: floor-ceiling \)
assumes \( n > 0 \)
shows \( \lfloor \frac{r}{\text{of-int } n} \rfloor = \lfloor \frac{r}{n} \rfloor \div n \)
proof
  let \( ?\text{of-int } = \text{of-int} :: int \Rightarrow 'a \)
define rhs where rhs = \( \lfloor \frac{r}{\text{of-int } n} \rfloor \)
define m where m = \( \lfloor \frac{r}{n} \rfloor \mod n \)
define e where e = \( r - \text{of-int } \lfloor \frac{r}{n} \rfloor \)
show e0: \( e \geq 0 \)
  using e-def by (metis diff-self_eq_iff floor-diff-of-int zero_le_floor)
show e1: \( e < 1 \)
  using e-def by (metis diff-self dual_order refl floor-diff-of-int floor_le_zero)
have r = \( \text{of-int } \lfloor r / n \rfloor + e \)
  unfolding e-def by simp
also have \( \lfloor r \rfloor = \text{rhs } * n + m \)
  using dm by simp
finally have \( r = \text{of-int } (\text{rhs } * n + m) + e \).

hence \( r / n = \text{of-int } (\text{rhs } * n) / \lfloor n + (e + \text{of-int } m) / n \rfloor \)
using n by (simp add: field-simps)
also have \( \text{of-int } (\text{rhs } * n) / n = \text{of-int } \text{rhs } \)
by auto
finally have \( \lfloor r / n \rfloor = \text{of-int } (\text{rhs } * n + \text{of-int } m) / \lfloor n + (e + \text{of-int } m) / n \rfloor \)
unfolding * by simp
also have \( \text{floor } (\lfloor e + \text{of-int } m \rfloor / n) = 0 \)
proof (rule floor-unique)
  show \( \text{of-int } 0 \leq (e + \text{of-int } m) / n \)
    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 \( (e + \text{of-int } m) / n < \text{of-int } 0 + 1 \)
proof (rule contr)
  from n have n\': ?n > 0 ?n \geq 0 by simp-all
  assume \neg \thesis
  hence (\epsilon + ?m) / ?n \geq 1 by auto
  from mult-right-mono[OF this (\epsilon)\']
  have ?n \leq \epsilon + ?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 \epsilon + ?n - 1 by auto

finally have \thesis show False by arith
qed

lemma floor-div-neg-int:
  fixes r :: 'a :: floor-ceiling
  assumes n: n < 0
  shows \lfloor r / of-int n \rfloor = \lceil r \div n \rceil
proof -
  from n have n\': -n > 0 by auto
  have \lfloor r / of-int n \rfloor = \lfloor -r / of-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-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 ^ 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 \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)


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: \( \forall x y. 0 \leq (x :: 'a :: linordered-field) \Rightarrow 0 \leq y \Rightarrow (x \cdot x \leq y \cdot y) = (x \leq y) \)
  by (metis mult-mono mult-strict-mono' not-less)

lemma square-less-square: \( \forall x y. 0 \leq (x :: 'a :: linordered-field) \Rightarrow 0 \leq y \Rightarrow (x \cdot x < y \cdot y) = (x < y) \)
  by (metis mult-mono mult-strict-mono' not-less)

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

lemma abs-lesseq-square: abs (x :: real) \leq abs y \longleftrightarrow x \cdot x \leq y \cdot 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 \geq 2} by auto

setup-lifting type-definition-proper-base

lift-definition get-base :: proper-base \Rightarrow int is \( \lambda x. x \).

lift-definition square-base :: proper-base \Rightarrow proper-base is \( \lambda x. x \cdot x \)

proof –
  fix i :: int
  assume i: \( 2 \leq i \)
  have \( 2 \cdot 2 \leq i \cdot i \)
    by (rule mult-mono[OF i i], insert i, auto)

end
thus $2 \leq i * i$ by auto

**qed**

**lift-definition** into-base :: int => proper-base is \( \lambda x. \) if \( x \geq 2 \) 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 \( b \) \( \geq \) 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[\( \text{of } b \)] by simp

**lemma** b-less-div-base-b: assumes \( xb \): \( \neg x < \) get-base \( b \)
sows \( x \) div get-base \( b \) < \( x \)
proof
from get-base-2[\( \text{of } b \)] have \( b \): get-base \( b \) \( \geq \) 2 .
with \( xb \) have \( x2 \): \( x \geq 2 \) by auto
with \( b \) int-div-less-self[\( \text{of } x \) (get-base \( b \))] show \( \neg \text{thesis by auto} \)
**qed**

We now state the main algorithm.

**function** log-main :: proper-base => int => nat \times 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 (\( \lambda \) \( (b,x) \). nat (1 + \( x - \) get-base \( b \))), insert b-less-square-base-b, auto)

**lemma** log-main: \( x > 0 \) \( \implies \) log-main \( b \) \( x \) = \( (y,by) \) \( \implies \) by = \( (get-base \) \( b \) \( \rightleftharpoons y \) \( \wedge \) \( (get-base \) \( b \) \( \rightleftharpoons y \) \( \leq \) \( x \) \( \wedge \) \( x < \) \( (get-base \) \( b \) \( \rightleftharpoons (Suc \) \( y \))))
proof (induct \( b \) \( x \) arbitrary: \( y \) by rule: log-main.induct)
case \( (1 \) \( x \) \( y \) \( by) \)
note \( x = 1(2) \)
note \( y = 1(3) \)
note \( IH = 1(1) \)
let \( ?b = \) get-base \( b \)
show \( ?\text{case} \)
proof (cases \( x < ?b \))
case \( True \)
  with \( x \) \( y \) show \( ?\text{thesis by auto} \)
next
case \( False \)
obtain $z$ bz where $zz$: \textit{log-main} (square-base $b$) $x = (z,bz)$

by (cases $\textit{log-main} \ (\text{square-base} \ b) \ x$, auto)

have id: \textit{get-base} (square-base $b$) $^\ast k = ?b ^\ast (2 \ast k)$ for $k$ unfolding square-base

by (simp add: power-mul semiring-normalization-rules(29))

from \textit{IH}[OF False $zz$, unfolded id]

have $z$: $?b ^\ast (2 \ast z) \leq x x < ?b ^\ast (2 \ast \text{Suc } z)$ and $bz$: $bz = \textit{get-base} b ^\ast (2 \ast z)$ by auto

from $y[unfolded \ \textit{log-main}.\text{simp}[of } b \ x]$ Let-def zz split| bz False

have $yy$: (if $x < bz \ast ?b$ then $(2 \ast z, bz)$ else $(\text{Suc } (2 \ast z), bz \ast ?b)) = (y, by)$ by auto

show ?thesis

proof (cases $x < bz \ast ?b$)

  case True

  with $yy$ have $yz$: $y = 2 \ast z$ by $bz$ by auto

  from True $z(1)$ $bz$ show ?thesis unfolding $yz$ by (auto simp: ac-simps)

  next

  case False

  with $yy$ have $yz$: $y = \text{Suc } (2 \ast z)$ by $bz$ by auto

  from False have $?b ^\ast \text{Suc } (2 \ast z) \leq x$ by (auto simp: bz ac-simps)

  with $z(2)$ $bz$ show ?thesis unfolding $yz$ by auto

qed

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

\textbf{definition} \textit{log-floor} :: $\texttt{int} \Rightarrow \texttt{int} \Rightarrow \texttt{nat}$ where

$\textit{log-floor} \ b \ x = \texttt{fst} \ (\textit{log-main} \ (\textit{into-base} \ b) \ x)$

\textbf{definition} \textit{log-ceiling} :: $\texttt{int} \Rightarrow \texttt{int} \Rightarrow \texttt{nat}$ where

$\textit{log-ceiling} \ b \ x = (\text{case } \textit{log-main} \ (\textit{into-base} \ b) \ x \ of

$($(y,by)$) \Rightarrow \text{if } x = \text{by} \text{ then } y \text{ else } \text{Suc } y)$

\textbf{lemma} \textit{log-floor-sound}: assumes $b > 1 \ x > 0 \ \text{log-floor} \ b \ x = y$

shows $b^\sim y \leq x x < b^\sim (\text{Suc } y)$

proof –

from $\texttt{assms}(1,3)$ have id: \textit{get-base} (into-base $b$) $= b$ by transfer auto

obtain $yy \ bb$ where log: $\textit{log-main} \ (\textit{into-base} \ b) \ x = (yy,bb)$

by (cases $\textit{log-main} \ (\textit{into-base} \ b) \ x$, auto)

from $\textit{log-main}[OF \ \texttt{assms}(2) \ log] \ \texttt{assms}(3)\{unfolded \ \textit{log-floor-def log}\}$ id

show $b^\sim y \leq x x < b^\sim (\text{Suc } y)$ by auto

qed

\textbf{lemma} \textit{log-ceiling-sound}: assumes $b > 1 \ x > 0 \ \text{log-ceiling} \ b \ x = y$

shows $x \leq b^\sim y \neq 0 \Rightarrow b^\sim (y - 1) < x$

proof –

from $\texttt{assms}(1,3)$ have id: \textit{get-base} (into-base $b$) $= b$ by transfer auto

obtain $yy \ bb$ where log: $\textit{log-main} \ (\textit{into-base} \ b) \ x = (yy,bb)$

by (cases $\textit{log-main} \ (\textit{into-base} \ b) \ x$, auto)

from $\textit{log-main}[OF \ \texttt{assms}(2) \ log, \ unfolded \ id] \ \texttt{assms}(3)\{unfolded \ \textit{log-ceiling-def log}\}$
split

have bnd: $b \sim yy \leq x < b \sim Suc yy$ \and\n\n  \ y: y = (if x = b \sim yy then yy else Suc yy) \by auto

have $x \leq b \sim y$ \and\n(y \neq 0 \implies b \sim (y - 1) < x)

proof (cases $x = b \sim yy$)
  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 $x \leq b \sim y$ y \neq 0 = \implies b \sim (y - 1) < x \by auto

qed

Finally, we connect it to the \texttt{log} function working on real numbers.

\texttt{lemma log-floor[simp]: assumes b: b > 1 and x: x > 0}
\texttt{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 *: $1 < \realofint b \ 0 < \realofint (b \sim y) \ 0 < \realofint x$
  \and **: $1 < \realofint b \ 0 < \realofint x \ 0 < \realofint (b \sim Suc y)$
  \by auto

show \thesis unfolding y

proof (rule sym, rule floor-unique)
  show real-of-int (int y) \leq \log (realofint b) (realofint x)
    using main(1)[folded log-le-cancel-iff[OF *, unfolded af-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

\texttt{lemma log-ceiling[simp]: assumes b: b > 1 and x: x > 0}
\texttt{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 *: $1 < \realofint b \ 0 < \realofint (b \sim (y - 1)) \ 0 < \realofint x$
  \and **: $1 < \realofint b \ 0 < \realofint x \ 0 < \realofint (b \sim Suc y)$
  \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 af-int-le-iff]]
  using log-pow-cancel[of b y] b \by auto

  from x have x: $x \geq 1$ \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 \( (2::'a)^{nat} \lfloor \log 2 \cdot n \rfloor / p \).

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 \( x \leq (\text{start-value } x \cdot p)^p \),
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 $\Rightarrow$ nat $\Rightarrow$ int where

start-value $n$ $p$ = $2$ $\sim$ (nat [af-nat (log-ceiling 2 $n$) / rat-of-nat $p$])

**lemma** start-value-main: assumes $x$ | $x \geq 0$ and $p$ | $p > 0$

shows $x \leq$ (start-value $x$ $p$) $\sim$ $p$ $\land$ start-value $x$ $p$ $\geq$ 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 $l2x$ = $\lceil \log 2 \, x \rceil$
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 $x$ $\geq$ 0 using $x$ by auto

hence $l2pos$: $l2x$ $\geq$ 0 by (auto simp: $l2x$-def)

have log 2 $x$ / $p$ $\leq$ $l2x$ / $p$ using $x$ $p$ unfolding $l2x$-def

by (metis divide-right-mono le-of-int-ceiling of-nat-0-le-iff)

also have ... $\leq$ [$l2x$ / ($p$ :: real)] by (simp add: ceiling-correct)

also have $l2x$ / real $p$ = $l2x$ / real-of-rat (of-nat $p$)

by (metis of-rat-of-int-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 [rat-of-int $l2x$ / of-nat $p$] $\leq$ real pow unfolding pow-def by auto

finally have le: log 2 $x$ / $p$ $\leq$ pow.

from powr-mono[OF le, of 2, folded $r$]

have root $p$ $x$ $\leq$ 2 powr pow by auto

also have ... = 2 $\sim$ pow by (rule powr-realpow, auto)

also have ... = of-int ((2 :: int) $\sim$ pow) by simp

also have pow = (nat [af-int (log-ceil 2 $x$) / rat-of-nat $p$])

unfolding pow-def $l2x$-def using $x$ by simp

also have real-of-int ((2 :: int) $\sim$ ...) = 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 ≤ 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 ≤ (start-value x p) ^ p by presburger
with start show thesis by auto
qed

lemma start-value: assumes x: x ≥ 0 and p: p > 0 shows x ≤ (start-value x p) ^ p
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 [root p n] 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.

class fixed-root
begin
lemma iteration-mono-eq: assumes \( x^n \cdot p = (n :: int) \)
shows \( (n \div x \cdot p + x \cdot \text{int } pm) \div \text{int } p = x \)
proof –
  have [simp]: \( \land n. (x + x \cdot n) = x \cdot (1 + n) \) by (auto simp: field-simps)
  show ?thesis unfolding \( x^n \cdot \text{symmetric} \) \( p \) 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 \( x : x \geq 0 \)
and \( n : n \geq 0 \)
and \( xn: x \cdot p > (n :: int) \)
shows \( (n \div x \cdot pm + x \cdot \text{int } pm) \div \text{int } p < x \)
proof –
  let \(?sx\) = \( (n \div x \cdot pm + x \cdot \text{int } pm) \div \text{int } p \)
  from xn have xn-le: \( x \cdot p \geq n \) by auto
  from xn x n have x0: \( x > 0 \)
    using not-le p by fastforce
  from p have xp: \( x \cdot p = x \cdot x \cdot \text{int } pm \) by auto
  from x n have n div x \cdot pm \cdot x \cdot pm \leq n
    by (auto simp add: minus-mod-eq-div-mult \( \text{symmetric} \) \( \text{mod-int-pos-iff not-less power-le-zero-eq} \))
    also have . . . \( \leq x \cdot p \) using xn by auto
finally have le: \( n \div x \cdot pm \leq x \) using x0 unfolding xp by simp
have \(?sx \leq (x \cdot p \div x \cdot pm + x \cdot \text{int } pm) \div \text{int } p \)
  by (rule zdiv-mono1, insert le p0, unfold xp, auto)
also have \( x \cdot p \div x \cdot pm = x \) unfolding xp by auto
also have \( x + x \cdot \text{int } pm = x \cdot \text{int } p \) unfolding \( p \) by (auto simp: field-simps)
also have \( x \cdot \text{int } p \) div \( \text{int } p = x \) using \( p \) by force
finally have le: \( ?sx \leq x \).
{ assume \(?sx = x\)
  from arg-cong[OF this, of \( \lambda x\). \( x \cdot \text{int } p \)]
  have \( x \cdot \text{int } p \leq (n \div x \cdot pm + x \cdot \text{int } pm) \div (\text{int } p) \cdot \text{int } p \) using p0 by simp
  also have . . . \( \leq n \div x \cdot pm + x \cdot \text{int } pm \)
    unfolding \text{mod-div-equality-int} using \( p \) by auto
finally have \( n \div x \cdot pm \geq x \) by (auto simp: p field-simps)
from \text{mult-right-mono}(OF this, of \( x \cdot pm \))
have ge: \( n \div x \cdot pm \cdot x \cdot pm \geq x \cdot p \) 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 \leq n - n mod x^pm .
  from x n have ge: n mod x^pm \geq 0
    by (auto simp add: mod-int-pos-iff not-less power-le-zero-eq)
  from le ge
  have n \geq x^p by auto
  with xn have False by auto
}
with le show ?thesis unfolding p by fastforce
qed

lemma iteration-mono-lessleq: assumes x: x \geq 0 and n: n \geq 0 and xn: x^p \geq (n :: int)
shows (n div x^pm + x * int pm) div int p \leq 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 = \lambda x n :: int. nat x
  let ?mI = \lambda (x,n). ?mm x n
  let ?m = measures [?mI]
  show ?thesis
proof (relation ?m)
  fix x n :: int
  assume \neg x^p \leq n
  hence x: x^p > n by auto
  assume \neg (x < 0 \lor n < 0)
  hence x-n: x \geq 0 n \geq 0 by auto
  from x x-n have x\theta: x > 0 using p by (cases x = 0, auto)
  from iteration-mono-less[OF x-n x] x\theta
  show (((n div x^pm + x * int pm) div int p, n), x, n) \in ?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: \( x \geq 0 \implies n \geq 0 \implies \)
root-main' \( x = \text{root-newton-int-main} \ x \land (\text{root-main'} \ x = (y,b) \implies y \geq 0 \)
\( \land y^p \leq n \land b = (y^p = n) \)
proof (induct \( x \land n \) rule: root-newton-int-main.induct)
case \( 1 \ x \ n \)
  have pm-x[simp]: \( x ^ \ pm \ast x = x ^ p \) unfolding \( p \) by simp
  from \( 1 \) have id: \( x < 0 \lor n < 0 \) = False by auto
note \( d = \text{root-main'}-\text{simps[of x n]} \text{ root-newton-int-main.simps[of x n]} \) id if-False
  Let-def show ?case
  proof (cases \( x ^ p \leq n \))
    case True
    thus ?thesis unfolding \( d \) using \( 1 \) by auto
  next
    case False
    hence id: \( x ^ p \leq n \) = False by simp
    from \( 1(2) \) \( 1(2) \) have not: \( - (x < 0 \lor n < 0) \) by auto
    then have \( x: x > 0 \lor x = 0 \)
      by auto
      with \( 0 \leq n \) have \( 0 \leq (n \div x ^ \ pm + x \ast \ \text{int pm}) \div \text{int p} \)
        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 \( 1(1)[OF \text{not False - 1(3)}] \))
  qed
qed

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

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

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

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

lemma root-newton-int-main-upper:
\[ y^p \geq n \Rightarrow y \geq 0 \Rightarrow n \geq 0 \Rightarrow \text{root-newton-int-main } y \ n = (x, b) \]

\[ \n < (x + 1)^p \]

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

- **case** (1 y n)
  - from \( \ell \) have \( y0: y \geq 0 \).
  - then have \( y > 0 \lor y = 0 \)
    - by auto
  - from \( \ell \) have \( n0: n \geq 0 \).
  - define \( y' \) where \( y' = (n \div (y^p) + y \ast \text{int } p) \div \text{int } p \)
    - by (auto simp add: \( y' \)-def \text{algebra-simps pos-imp-zdiv-nonneg-iff power-0-left})
  - let \( ?rt = \text{root-newton-int-main} \)
  - from \( \ell \) have \( \text{rt } y \ n = (x, b) \)
    - by simp
  - show \( ?\text{thesis} \)
    - proof (cases \( n \leq y' \))
      - case False
        - thus \( ?\text{thesis} \)
          - unfolding \( p \) by (metis \text{False } y'0 zero-le-power)
    - next
      - case True
        - thus \( ?\text{thesis} \)
          - unfolding \( p \) by (metis \text{True } n00 \ y' \)

**next**

- **case** (False note n00 = this)
  - let \( \ell y = \text{of-int } y :: \text{real} \)
  - let \( ?n = \text{of-int } n :: \text{real} \)
  - from \( y0n0 \) have \( y00: y \neq 0 \)
    - unfolding \( p \) by auto
  - from \( y00 \) 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 lls = NY # replicate pm ?y

have prod: \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 root p ?n = gmean lls unfolding gmean-def using y0
  by (auto simp: p NY-def prod)

also have . . . < mean lls
proof (rule CauchysMeanTheorem-Less[OF pos het-gt-0I])
  show NY \in set lls by simp
  from pm0 show ?y \in set lls 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
  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 \geq 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 / rat-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 ⌊ S ⌋/ real p ≤ real-of-int (s div int p) + 1
    using f1 f3 f4 by (metis div-is-floor-divide-real zdiv-mono1)

have f′: 0 < [rat-of-nat p]
    using p by simp
have f1: (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 pos1: (NY + Y) / p ≥ 0 unfolding Y-def NY-def
    by (rule power-mono[OF ge pos1])
from order.strict-trans2[OF ineq1 ineq2]
have ?n < of-int ((x + 1) ^ p)
    using of-int-less-iff by blast

have f5: (y ^ 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
Now we can prove all the nice properties of $\text{root-int-main}$.

**lemma** $\text{root-int-main-all}$: assumes $n \geq 0$

and $\text{rm}$: $\text{root-int-main} p n = (y,b)$

shows $y \geq 0 \land b = (y \sim p = n) \land (p > 0 \to y \sim p \leq n \land n < (y + 1) \sim p)$

$\land (p > 0 \to x \geq 0 \to x \sim p = n \to y = x \land b)$

**proof** (cases $p = 0$)

  - case $\text{True}$
    - from $\text{rm[unfolded root-int-main-def]}$
    - have $y$: $y = 1$ and $b$: $b = (n = 1)$ by auto
    - show $?thesis$ unfolding $\text{True y b using n by auto}$
  - case $\text{False}$
    - from $\text{root-main}$
      - have $\text{start-value n p} p 0$ by auto
    - interpret $\text{fixed-root p p = 1}$
      - by (unfold-locales, insert False, auto)
    - from $\text{root-main'-pos[OF start(2) n rm]}$
      - have $y$: $y \geq 0$
    - from $\text{root-main'-sound[OF start(2) n rm]}$
      - have $b$: $b = (y \sim p = n)$
    - from $\text{root-main'-lower[OF start(2) n rm]}$
      - have low: $y \sim p \leq n$
    - from $\text{root-main'-upper[OF start n rm]}$
      - have up: $n < (y + 1) \sim p$

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

thus $?thesis$ using $b$ low up $y$ by auto

**qed**

**lemma** $\text{root-int-main}$: assumes $n \geq 0$

and $\text{rm}$: $\text{root-int-main} p n = (y,b)$

shows $y \geq 0 \land b = (y \sim p = n) \land (p > 0 \to y \sim p \leq n \land n < (y + 1) \sim p)$

$p > 0 \to x \geq 0 \to x \sim p = n \to y = x \land b$

**proof** (cases $p = 0$)

  - case $\text{True}$

**lemma** $\text{root-int[simp]}$: assumes $p$: $p \neq 0 \lor x \neq 1$

shows set (root-int $p x$) = $\{ y \cdot y \sim p = x \}$

**proof** (cases $p = 0$)

  - case $\text{True}$
with \( p \) have \( x \neq 1 \) 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

next

case False

hence \( x: (x = 0) = False \) by auto

hence \( x: (x = 0) = False \) by auto

hence \( x: (x = 0) = False \) by auto

hence \( x: (x = 0) = False \) by auto

hence \( x: (x = 0) = False \) by auto

hence \( x: (x = 0) = False \) by auto
cases \( z < 0, \text{ auto} \) 
\}
moreover
\{
\begin{align*}
\text{fix } z \\
\text{assume } z: z \in \text{ set } \mathcal{L}_\text{hs} \\
\text{hence } b: b = \text{ True by } \{\text{cases } b, \text{ auto}\} \\
\text{note } z = z[\text{unfolded } b \text{ if-True}] \\
\text{from } \text{ rm}(2) b \text{ have } yz: y^p = |x| \text{ by } \text{ auto} \\
\text{from } \text{ rm}(1) \text{ have } y: y \geq 0 . \\
\text{from } \text{ False have } \text{ odd } p \lor \text{ even } p \land x \geq 0 \text{ by } \text{ auto} \\
\text{hence } z \in ?\text{rhs} \\
\end{align*}
\]
\begin{proof}
\begin{align*}
\text{assume odd: odd } p \\
\text{with } z \text{ have } z = \text{ sgn } x \ast y \text{ by } \text{ auto} \\
\text{hence } z^p = (\text{ sgn } x \ast y)^p \text{ by } \text{ auto} \\
\text{also have } \ldots = \text{ sgn } x^p \ast y^p \text{ unfolding } \text{ power-mult-distrib by } \text{ auto} \\
\text{also have } \ldots = \text{ sgn } x^p \ast \text{ abs } x \text{ unfolding } yz \text{ by } \text{ simp} \\
\text{also have } \ldots = \text{ sgn } x^p \ast \text{ abs } x = x \text{ by } (\text{ rule mult-sgn-abs}) \\
\text{finally show } z \in ?\text{rhs by } \text{ auto} \\
\end{align*}
\end{proof}
\begin{proof}
\begin{align*}
\text{next} \\
\text{assume even: even } p \land x \geq 0 \\
\text{from } z \text{ even have } z = y \lor z = -y \text{ by } \text{ auto} \\
\text{hence id: abs } z = y \text{ using } y \text{ by } \text{ auto} \\
\text{with } yz x \text{ even have } z: z \neq 0 \text{ using } p0 \text{ by } (\text{ cases } y = 0, \text{ auto}) \\
\text{have } z^p = (\text{ sgn } z \ast \text{ abs } z)^p \text{ by } (\text{ simp add: mult-sgn-abs}) \\
\text{also have } \ldots = (\text{ sgn } z \ast y)^p \text{ using } \text{ id by } \text{ auto} \\
\text{also have } \ldots = (\text{ sgn } z)^p \ast y^p \text{ unfolding } \text{ power-mult-distrib by } \text{ simp} \\
\text{also have } \ldots = \text{ sgn } z^p \ast x \text{ unfolding } yz \text{ using } \text{ even by } \text{ auto} \\
\text{also have } \text{ sgn } z^p = 1 \text{ using even } z \text{ by } (\text{ auto}) \\
\text{finally show } z \in ?\text{rhs by } \text{ auto} \\
\end{align*}
\end{proof}
\}
ultimately show ?thesis by blast
\}
\end{proof}
\end{proof}
\end{proof}

lemma root-int-pos: assumes \( x: x \geq 0 \) and \( ri: \text{ root-int } p \ x = y \# \ ys \)
shows \( y \geq 0 \)
\begin{proof}
\begin{align*}
\text{from } x \text{ have } \text{ abs } x = x \text{ by } \text{ auto} \\
\text{note } ri = ri[\text{unfolded } \text{ root-int-def Let-def abs}] \\
\text{from } ri \text{ have } p: (p = \text{ 0}) = \text{ False by } \{\text{cases } p, \text{ auto}\} \\
\text{note } ri = ri[\text{unfolded } p \text{ if-False}] \\
\text{show } ?\text{thesis}
\end{align*}
\end{proof}
proof \((\text{cases } x = 0)\)
  
  case \text{True}
  
  with \(ri\) show \(?\text{thesis}\) by auto
  
  next
  
  case \text{False}
  
  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 \(y = (\text{if even } p \text{ then } y' \text{ else } \text{sgn } x \ast y' )\) by (cases \(b', \text{auto}\))
  
  from \(\text{root-int-main}(1)[OF } x \] have \(y': 0 \leq y'\).
  
  thus \(?\text{thesis}\) unfolding \(y\) using \(x\) False by auto
  
  qed
  
  qed

3.3 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 \(\text{root-int-floor-pos-lower: assumes } p 0: p \neq 0 \text{ and } x: x \geq 0\)
  shows \(\text{root-int-floor-pos } p x \geq x\)
  using \(\text{root-int-main}(3)[OF } x , \text{ of } p\] \(p0\) unfolding \(\text{root-int-floor-pos-def}\)
  by (cases \(\text{root-int-main } p x , \text{auto}\))

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 , \text{ of } p\]
  unfolding \(\text{root-int-floor-pos-def}\)
  by (cases \(\text{root-int-main } p x , \text{auto}\))

lemma \(\text{root-int-floor-pos-upper: assumes } p 0: p \neq 0 \text{ and } x: x \geq 0\)
  shows \(\text{root-int-floor-pos } p x + 1 \geq p > x\)
  using \(\text{root-int-main}(4)[OF } x , \text{ of } p\] \(p0\) unfolding \(\text{root-int-floor-pos-def}\)
  by (cases \(\text{root-int-main } p x , \text{auto}\))

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

proof \((\text{cases } p = 0)\)
  
  case \text{True}
  
  thus \(?\text{thesis}\) by (simp add: \(\text{root-int-floor-pos-def}\))
  
  next
  
  case \text{False}
  
  hence \(p: p > 0\) by auto
  
  let \(?s1 = \text{real-of-int } (\text{root-int-floor-pos } p x)\)
let \( s_2 = \text{root } p \ (\text{of-int } x) \)

from \( x \) have \( s_1: \ s_1 \geq 0 \)
  by (metis of-int-0-le-iff root-int-floor-pos-pos)

from \( x \) have \( s_2: \ s_2 \geq 0 \)
  by (metis of-int-0-le-iff real-root-pos-pos-le)

from \( s_1 \) have \( s_{11}: \ s_1 + 1 \geq 0 \) by auto

have id: \( s_2 ^ p = \text{of-int } x \) using \( x \)
  by (metis p of-int-0-le-iff real-root-pow-pos2)

show \( \text{thesis} \)
proof (rule floor-unique[symmetric])
  show \( s_1 \leq s_2 \)
    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-le-iff of-int-power)
  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 of-int-add of-int-less-iff of-int-power of-int-1)
qed

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

21
from False have x0: \( x \neq 0 \) using \( \text{rm}(5)[\text{of } 0] \) using \( s \) unfolding root-int-main-def
Let-def using \( p \)
by (cases \( x = 0 \), auto)
show \(?\text{thesis}\) unfolding id root-int-floor-pos[OF \( x \)]
proof (rule ceiling-unique[symmetric])
show \(?sx \leq \text{real-of-int} (\lfloor \text{root } p \ (\text{of-int } x) \rfloor + 1)\)
by (metis of-int-add real-of-int-floor-add-one-ge of-int-1)
let \(?l = \text{real-of-int} (\lfloor \text{root } p \ (\text{of-int } x) \rfloor + 1) - 1\)
let \(?m = \text{real-of-int} \lfloor \text{root } p \ (\text{of-int } x) \rfloor\)
have \(?l = ?m\) 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 \(?m ^ p = ?sx ^ p\) by auto
also have \(\ldots = \text{of-int } x\) 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: \( x = \lfloor \text{root } p \ (\text{of-int } x) \rfloor ^ p\)
by (metis floor-power floor-of-int)
hence \(\lfloor \text{root } p \ (\text{of-int } x) \rfloor \in \text{set} \ (\text{root-int } p \ x)\) using \( p \) by simp
hence \(\text{root-int } p \ x \neq []\) 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 \(?\text{thesis}\) by arith
qed
finally show \(?l < ?sx\).
qed
qed
qed

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

lemma root-int-floor[simp]: root-int-floor \( p \ x = \text{floor} \ (\text{root } p \ (\text{of-int } x))\)
proof –
note \( d = \text{root-int-floor-def}\)
show \(?\text{thesis}\)
proof (cases \( x \geq 0 \))
case True
with root-int-floor-pos[OF \( True \), \( p \)] show \(?\text{thesis}\) 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

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 :: $nat \Rightarrow nat \Rightarrow int$ where
  root-nat-floor $p$ $x$ = root-int-floor-pos ($p$ ($int$ $x$))
definition root-nat-ceiling :: $nat \Rightarrow nat \Rightarrow int$ where
  root-nat-ceiling $p$ $x$ = root-int-ceiling-pos ($p$ ($int$ $x$))
definition root-nat :: $nat \Rightarrow nat \Rightarrow nat$ 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 $p$ $x$ $\leq p \leq x$
  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 $p0$: $p \neq 0 \lor x \neq 1$
shows set (root-nat p x) = \{ y. y ^ p = x \}

proof –

\{
  \begin{align*}
    \text{fix } y \\
    \text{assume } y \in \text{set (root-nat p x)} \\
    \text{note } y = \text{this[unfolded root-nat-def]} \\
    \text{then obtain } y_i y_s \text{ where } ri: \text{root-int p x } y_i \# y_s \text{ by (cases root-int p x, auto)} \\
    \text{with } y \text{ have } y = \text{nat y_i by auto} \\
    \text{from root-int-pos[OF - ri] have } y_i: 0 \leq y_i \text{ by auto} \\
    \text{from root-int[of p int x] p0 ri have } y_i ^ p = x \text{ by auto} \\
    \text{from arg-cong[OF this, of nat]} \text{ yi have } \text{nat y_i } ^ p = x \\
    \text{by (metis nat-int nat-power-eq)} \\
    \text{hence } y \in \{ y. y ^ p = x \} \text{ using } y \text{ by auto}
  \end{align*}
\}

moreover

\{
  \begin{align*}
    \text{fix } y \\
    \text{assume } yx: y ^ p = x \\
    \text{hence } y: \text{int } y ^ p = \text{int } x \\
    \text{by (metis of-nat-power)} \\
    \text{hence set (root-int p (int x)) } \neq \{ \} \text{ using root-int[of p int x] p0} \\
    \text{by (metis (mono-tags) One-nat-def \{ y ^ p = x \} empty-Collect-eq nat-power-eq-Suc-0-iff)} \\
    \text{then obtain } y_i y_s \text{ where } ri: \text{root-int p (int x) } y_i \# y_s \\
    \text{by (cases root-int p (int x), auto)} \\
    \text{from root-int-pos[OF - this] have } y_i p: y_i \geq 0 \text{ by auto} \\
    \text{from root-int[of p int x, unfolded ri] p0 have } y_i: y_i ^ p = \text{int } x \text{ by auto} \\
    \text{with } y \text{ have int } y ^ p = y_i ^ p \text{ by auto} \\
    \text{from arg-cong[OF this, of nat]} \text{ have id: y ^ p = nat y_i ^ p} \\
    \text{by (metis (y ^ p = x) nat-int nat-power-eq yi yip)}
  \end{align*}
\}

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

\text{note } rm = \text{rm[unfolded this]} \\
\text{hence } y \in \text{set (root-nat p x)} \\
\text{unfolding root-nat-def p root-int-def using p0 p yx} \\
\text{by auto}

moreover

\{
  \begin{align*}
    \text{assume } p: p = 0 \\
    \text{with } p0 \text{ have } x \neq 1 \text{ by auto} \\
    \text{with } y p \text{ have False by auto}
  \end{align*}
\}

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

\section*{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 \(\frac{a^{1/p}}{b^{1/p}}\) as \(\frac{(ab^{p-1})^{1/p}}{b}\).

**definition** \texttt{root-rat-floor} :: \texttt{rat \Rightarrow int where}

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

**definition** \texttt{root-rat-ceiling} :: \texttt{nat \Rightarrow rat \Rightarrow int where}

\[\text{root-rat-ceiling } p \ x \equiv - (\text{root-rat-floor } p \ (-x))\]

**definition** \texttt{root-rat} :: \texttt{nat \Rightarrow rat \Rightarrow rat list where}

\[\text{root-rat } p \ x \equiv \text{case quotient-of } x \ of \ (a,b) \Rightarrow \text{concat} (\text{map } (\lambda \ rb. \text{map } (\lambda ra. \text{of-int } ra / \text{rat-of-int } rb) \ (\text{root-int } p a)) \ (\text{take } 1 \ (\text{root-int } p b)))\]

**lemma** \texttt{root-rat-reform}: \texttt{assumes q: quotient-of } x \ (a,b) \ shows \(\text{root } p \ (\text{real-of-rat } x) = \text{root } p \ (\text{of-int } (a * b ^{(p - 1)})) / \text{of-int } b\)

**proof** (cases \(p = 0\))

\texttt{case False}

\texttt{from quotient-of-denom-pos[OF q] have } b: \texttt{0 < b by auto}

\texttt{hence } b: \texttt{0 < real-of-int } b \texttt{ by auto}

\texttt{from quotient-of-div[OF q] have } x: \texttt{root } p \ (\text{real-of-rat } x) = \texttt{root } p \ (a / b)

\texttt{by (metis of-rat-divide of-rat-of-int-eq)}

\texttt{also have } a / b = a * \text{real-of-int } b ^{(p - 1)} / \text{of-int } b ^{(p)} \texttt{ using } b \texttt{ False}

\texttt{by (cases } p, \texttt{auto simp: field-simps)}

\texttt{also have } \texttt{root } p \ ... \equiv \texttt{root } p \ (a * \text{real-of-int } b ^{(p - 1)}) / \texttt{root } p \ (\text{of-int } b ^{(p)})

\texttt{by (rule real-root-divide)}

\texttt{also have } \texttt{root } p \ (\text{of-int } b ^{(p)}) = \texttt{of-int } b \texttt{ using } b \texttt{ False}

\texttt{by (metis of-rat-conv real-root-pow-pos real-root-power)}

\texttt{also have } a * \text{real-of-int } b ^{(p - 1)} = \texttt{of-int } (a * b ^{(p - 1)})

\texttt{by (metis of-int-mult of-int-power)}

finally show \(?thesis .

**qed auto**

**lemma** \texttt{root-rat-floor [simp]: root-rat-floor } p \ x \equiv \texttt{floor } (\text{root } p \ (\text{of-rat } x))

**proof** –

\texttt{obtain } a b \texttt{ where } q: \texttt{quotient-of } x = (a,b) \texttt{ by force}

\texttt{from quotient-of-denom-pos[OF q] have } b: \texttt{b > 0 .}

\texttt{show } ?thesis

\texttt{unfolding root-rat-floor-def q split root-int-floor}

\texttt{unfolding root-rat-reform[OF q] floor-div-pos-int[OF b] ..}

**qed**

**lemma** \texttt{root-rat-ceiling [simp]: root-rat-ceiling } p \ x = \texttt{ceiling } (\text{root } p \ (\text{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 ≠ 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)
      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 ∈ 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 ∈ set (root-int p b) by auto
      with ra p have rb: b = rb ^ p and ra: a = ra ^ p by auto
      have q ∈ {y. y ^ p = x} unfolding q x ra rb
        by (auto simp: power-divide)
  }
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 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: \( a \ast n^p = b \ast z^p \) by linarith
from quotient-of-caprine[OF quo] have cop: caprine \((z^p) \ (n^p)\)
  by simp
from coprine-crossproduct-int[OF quotient-of-caprine[OF quot]] 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: \( n \in \text{set \ (root-int p b)} \) using p by auto
then obtain rb rs where rb: \( \text{root-int p b} = \text{Cons rb rs} \) by (cases root-int p b, auto)
from id[folded bnp] b have a = \( z^p \) by auto
hence a: \( z \in \text{set \ (root-int p a)} \) 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 ^ 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: \( n \in \text{set \ (take 1 \ (root-int p b))} \) by auto
have q \( \in \text{set \ (root-rat p x)} \) unfolding d qzn using b a by auto
}
ultimately show \( ?\text{thesis} \) by blast
next
case True
with p have x: \( x \neq 1 \) by auto
obtain a b where q: quotient-of x = \( (a, b) \) by force
show \( ?\text{thesis} \) 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 CeTÀ [2], a tool for certifying 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

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

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 $\frac{y}{x}$ 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 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
\[ \text{simplify:} \quad \text{sqrt-int-main'} \ x \ n = \text{root-int-main'} \ 1 \ 1 \ 2 \ x \ n \]

lemma sqrt-int-main'-code[code]: sqrt-int-main' \ x \ n = (let \ x^2 = x * x \ in \ if \ x^2 \leq n \ then \ (x, x^2 = n) \ else \ sqrt-int-main' \ ((n \div \ x + x) \div 2) \ n) \using \text{root-int-main'.simpl[of 1 1 2 x n]}
unfolding \text{Let-def by auto}

definition sqrt-int-main :: int ⇒ int × bool where
\[ \text{simplify:} \quad \text{sqrt-int-main} \ x = \text{root-int-main} \ 2 \ x \]

lemma sqrt-int-main-code[code]: sqrt-int-main \ x = sqrt-int-main' \ (\text{start-value} \ 2 \ x) \ x 
by \text{(simpl add: root-int-main-def Let-def)}

definition sqrt-int :: int ⇒ int list where
\[ \text{simplify:} \quad \text{sqrt-int} \ x = \text{root-int} \ 2 \ x \]

lemma sqrt-int-code[code]: sqrt-int \ x = (if \ x < 0 \ then \ [] \ else \ case \ sqrt-int-main \ x \of \ (y, \text{True}) \Rightarrow \text{if} \ y = 0 \ then \ [0] \ else \ [y, -y] \mid - \Rightarrow []) \proof
interpret fixed-root 2 1 by (unfold-locales, auto)
obtain b \ y \ where \ res: \text{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 \geq 0 
proof
note res = res[unfolded sqrt-int-code Let-def, simplified]
from res have \ x\$>: \ x \geq 0 \by (cases ?thesis, auto)
obtain \ ss b \ where \ call: \sqrt-int-main \ x = (ss, b) \by force
from res[unfolded call] \ x\$> have \ ss = s
by (cases b, cases \ ss = 0, auto)
from root-int-main(1)[\OF \ x\$> call[unfolded this sqrt-int-main-def]]
show \ ?thesis .
qed

definition \[\text{simplify:} \quad \text{sqrt-int-floor-pos} \ x = \text{root-int-floor-pos} \ 2 \ x \]
lemma sqrt-int-floor-pos-code[code]: \( \text{sqrt-int-floor-pos } x = \text{fst (sqrt-int-main } x) \)
by \((\text{simp add: root-int-floor-pos-def})\)

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

declaration [simp]: \( \text{sqrt-int-ceiling-pos } x = \text{root-int-ceiling-pos 2 } x \)

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

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

declaration sqrt-int-floor x = root-int-floor 2 x

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

lemma sqrt-int-floor[simp]: \( \text{sqrt-int-floor } x = [ \text{sqrt (of-int } x)] \)
by \((\text{simp add: sqrt-int-floor-def sqrt-def})\)

declaration sqrt-int-ceiling x = root-int-ceiling 2 x

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

lemma sqrt-int-ceiling[simp]: \( \text{sqrt-int-ceiling } x = [ \text{sqrt (of-int } x)] \)
by \((\text{simp add: sqrt-int-ceiling-def sqrt-def})\)

lemma sqrt-int-ceiling-bound: \( 0 \leq x \Rightarrow x \leq (\text{sqrt-int-ceiling } x)^2 \)
unfolding sqrt-int-ceiling using le-of-int-ceiling sqrt-le-D
by \((\text{metis of-int-power-le-of-int-cancel-iff})\)

4.3 Square roots for the naturals
declaration sqrt-nat :: nat \( \Rightarrow \) nat list
where sqrt-nat x = root-nat 2 x

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

lemma sqrt-nat[simp]: \( \text{set (sqrt-nat } x) = \{ y. y \ast y = x\} \)
unfolding sqrt-nat-def using root-nat[of 2 x] by \((\text{simp add: power2-eq-square})\)
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 ≥ 0
  proof –
    obtain z n where q: quotient-of x = (z,n) by force
    note sqrt = sqrt[code][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 ?sz, 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 ≤ sz 0 ≤ sn` by auto
from `sz sz sn` have `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` ∈ 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, i) 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 simp
  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::`'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` where
  [code]: `sqrt-approx-main-impl` `ε` `n` `x` = (if `x` * `x` − `n` < `ε` then `x` else `sqrt-approx-main-impl`)
\( \varepsilon n \)

\( \left( \frac{n \cdot \sqrt{x} + x}{2} \right) \)

We setup a locale where we ensure that we have standard assumptions: positive \( \varepsilon \) and positive \( n \). We require sort floor-ceiling, since \([x]\) is used for the termination argument.

locale sqrt-approximation =

fixes \( \varepsilon :: 'a \) :: {linordered-field, floor-ceiling}
and \( n :: 'a \)
assumes \( \varepsilon > 0 \)
and \( n > 0 \)
begin

function sqrt-approx-main :: 'a \Rightarrow 'a
where

\( \sqrt-approx-main x = \begin{cases} x & \text{if } x > 0 \text{ and } x \cdot x - n < \varepsilon \\ \sqrt-approx-main \left( \frac{n \cdot \sqrt{x} + x}{2} \right) & \text{else} \end{cases} \)
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 \( x \cdot \sqrt{n} - 1 \) is not expressible. Instead we use the expression \( x^2/n - 1 \) as error-rate which does not require any square-root operation.

termination
proof –
define er where \( er x = (x \cdot x - n - 1) / n \) for \( x \)
define c where \( c = 2 \cdot n / \varepsilon \)
define m where \( m x = \text{nat} \lfloor c \cdot er x \rfloor \) for \( x \)

have \( c > 0 \) unfolding c-def using \( n \varepsilon \) by auto

show \(?thesis

proof

show \( \text{wf (measures \[m\])} \) by simp

next

fix \( x \)
assume \( x: 0 < x \) and \( \varepsilon x: x \cdot x - n < \varepsilon \)
define y where \( y = \left( \frac{n \cdot \sqrt{x} + x}{2} \right) \)

show \( \left( \frac{n \cdot \sqrt{x} + x}{2}, x \right) \in \text{measures \[m\]} \) unfolding y-def[symmetric]

proof (rule measures-less)

from \( n \) have \( \text{inv-n: } 1 / n > 0 \) by auto

from \( x \) have \( x \cdot x - n \geq \varepsilon \) by simp

from \( \text{this[unfolded mult-le-cancel-left-pos]} \) \([\text{OF inv-n, of } \varepsilon, \text{symmetric}]\]

have \( \text{erx: } er x \geq \varepsilon / n \) unfolding er-def using \( n \) by (simp add: field-simps)

have \( \text{enx: } \varepsilon / n > 0 \) and \( \text{ne: } n / \varepsilon > 0 \) using \( \varepsilon n \) by auto

from \( \text{en erx } \)
have \( \text{ex: } er x > 0 \) by linarith

have \( \text{pos: } er x \cdot 4 + er x \cdot (er x \cdot 4) > 0 \) using \( \text{erx}\)

by (auto intro: add-pos-nonneg)

have \( \text{er y = } 1 / 4 \cdot (n / (x \cdot x) - 2 + x \cdot x / n) \) unfolding er-def y-def
using \( x \ n \)
by (simp add: field-simps)
also have \( \ldots = 1 / 4 * \ er x * \ er x / (1 + \ er x) \) 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 \( \ldots < 1 / 4 * (1 + \ er x) * \ er x / (1 + \ er x) \) using \( \er x \ er x \)
by (auto simp: field-simps)
also have \( \ldots = \er x * \ er x / (1 + \ er x) \).
also have \( \ldots < 1 / 4 * (1 + \ er x) * \ er x / (1 + \ er x) \) using \( \er x \ er x \)
by (auto simp: field-simps)
finally have \( \er y * \er x = 1 / 4 * \ er x * \ er x / (1 + \ er x) \).

Also soundness is not complicated.

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

proof (induct x rule: sqrt-approx-main.induct)
  case (1 x)
  from 1 have x: \( x > 0 \) \( (x > 0) = True \) by auto
  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 1 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
        qed
    qed
  qed
end

It remains to assemble everything into one algorithm.

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

lemma sqrt-approx: assumes \( \varepsilon :: \varepsilon > 0 \)
  shows \( |\text{sqrt-approx} \varepsilon \ x - |x| \| < \varepsilon \)
proof (cases \( x = 0 \))
  case True
  with \( \varepsilon \) show ?thesis unfolding sqrt-approx-def by auto
next
  case False

35
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 executability 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 / 1000000000 :: rat; s = sqrt-approx ε 2 in (s, s * s - 2, |s * s - 2| < ε)

results in (1.4142135623731116, 4.7382007621468612e-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
