Computing N-th Roots using the Babylonian Method*

René Thiemann

August 16, 2018

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 ........................................ 9
  3.2 Computing the p-th root of an integer number ................. 9
  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 28
  4.1 The Babylonian method .................................. 28
  4.2 The Babylonian method using integer division .......... 29
  4.3 Square roots for the naturals ........................... 31
  4.4 Square roots for the rationals ......................... 31
  4.5 Approximating square roots ............................ 33
  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 * x = n - n mod x
using div-mult-mod-eq[of n x] by arith

lemma log-pow-cancel[simp]: a > 0 ⇒ a ≠ 1 ⇒ log a (a ^ b) = b
by (metis monoid-mul-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
  define y where y = - x
  from False have 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 / of-int n⌋ = ⌊r⌋ div n (is ?l = ?r)
proof
  let ?of-int = of-int :: int ⇒ 'a
  define rhs where rhs = ⌊r⌋ div n
  let ?n = ?of-int n
  define m where m = ⌊r⌋ mod n
  let ?m = ?of-int m
  from div-mult-mod-eq[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
  define e where e = r - ?of-int ⌊r⌋
have \( e_0 \colon e \geq 0 \) unfolding \( e \)-def
   by (metis diff-self eq-iff floor-diff-of-int zero-le-floor)
have \( e_1 \colon e < 1 \) 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 \( r = \text{?of-int} \lfloor r \rfloor + e \)
   unfolding \( e \)-def by simp
also have \( \lfloor r \rfloor = \text{rhs} \cdot n + m \) using \( n \) by (simp add: field-simps)
finally have \( r = \text{?of-int} (\lfloor r \rfloor + e) \)
   by (simp add: field-simps)

have \( l = \text{rhs} + \text{floor} ((e + ?m) / ?n)\)
   unfolding \( e \)-def by simp
also have \( \text{floor} ((e + ?m) / ?n) = 0 \)
proof (rule floor-unique)
  assume \( \neg \text{thesis} \)
  hence \( (e + \text{?m}) / ?n \geq 1 \) by auto
  from \( \text{mult-right-mono}[OF this \ n'(2)] \)
  show \( ?n \leq e + \text{?m} \) by simp
  also have \( \text{?m} \leq ?n - 1 \) by (simp)
    using \( mn \)
  finally show \( ?n \leq e + ?n - 1 \) by auto
    with \( e1 \) show False by arith
qed

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

lemma \( \text{floor-div-neg-int} \):
  fixes \( r \colon 'a \colon \text{floor-ceiling} \)
  assumes \( n \colon n < 0 \)
  shows \( \lfloor r / \text{af-int} n \rfloor = \lfloor r \rfloor \div n \)
proof
  from \( \text{n have n'} \colon -n > 0 \) by auto
  have \( \lfloor r / \text{af-int n} \rfloor = \lfloor - r - (n) \rfloor \) 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 \( \ldots = \lfloor r \rfloor \div n \) using \( n \)
    by (metis ceiling-def div-minus-right)
  finally show \( \neg \text{thesis} \)
qed

3
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-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)

end

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

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

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

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

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

lemma abs-lesseq-square: abs \( x :: real \) \leq abs \( y \) \iff \( x \cdot x \leq y \cdot y \)
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.

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

setup-lifting type-definition-proper-base

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

lift-definition square-base :: proper-base ⇒ proper-base is λ x. x * x
proof
   fix i :: int
   assume i: 2 ≤ i
   have 2 * 2 ≤ i * i
      by (rule mult-mono[OF i i], insert i, auto)
   thus 2 ≤ i * i by auto
qed

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

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

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

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

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

We now state the main algorithm.

function log-main :: proper-base ⇒ int ⇒ nat × int where
log-main b x = (if x < get-base b then (0,1) else
   case log-main (square-base b) x of
   (z, bz) ⇒
   let l = 2 * z; bz1 = bz * get-base b
   in if x < bz1 then (l,bz) else (Suc l,bz1))
termination by (relation measure \(\lambda (b,x). \text{nat} (1 + x - \text{get-base} b)\)), insert b-less-square-base-b, auto

lemma log-main: \(x > 0 \implies \text{log-main} b x = (y,by) \implies by = (\text{get-base} b) \wedge (x \leq y \wedge y < (\text{get-base} b) \cdot (\text{Suc} y)\)

proof (induct b x arbitrary; y by rule: log-main.induct)
  case (1 b x y by)
  note x = 1 (2)
  note y = 1 (3)
  note IH = 1 (1)
  let ?b = get-base b
  show \(?\text{thesis}\) by auto
next
  case False
  obtain z bz where zz: \(\text{log-main} (\text{square-base} b) x = (z,bz)\)
  by (cases log-main (square-base b) x, auto)
  have id: \(\text{get-base} (\text{square-base} b) \cdot k = ?b \cdot (2 * k)\) for k unfolding square-base
  by (simp add: power-mult semiring-normalization-rules (29))
  from IH [OF False x zz, unfolded id]
  have z: \(?b \cdot (2 * z) \leq x \wedge (2 * Suc z) \leq ?b \cdot (2 * Suc (2 * z))\)
  and bz: bz = get-base b \cdot (2 * z)
  by auto
  from y [unfolded log-main.simps[of b x] Let-def zz split] bz False
  have yz: \((y < bz \wedge ?b then (2 * z, bz) else (Suc (2 * z), bz * ?b)) = (y, by)\) by auto
  show \(?\text{thesis}\)
  proof (cases x < bz * ?b)
    case True
    with yz have yz: y = 2 * z by = bz by auto
    from True z(1) bz show \(?\text{thesis}\) unfolding yz by (auto simp: ac-simps)
next
    case False
    with yz have yz: y = Suc (2 * z) by = ?b * bz by auto
    from False have ?b \cdot Suc (2 * z) \leq x by (auto simp: bz ac-simps)
    with z(2) bz show \(?\text{thesis}\) unfolding yz by auto
  qed
  qed
  qed

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

definition log-floor :: int \Rightarrow int \Rightarrow \text{nat} where
log-floor b x = fst (log-main (into-base b) x)

definition log-ceiling :: int \Rightarrow int \Rightarrow \text{nat} where
log-ceiling b x = (case log-main (into-base b) x of
\[(y, by) \Rightarrow \text{if } x = by \text{ then } y \text{ else } \text{Suc } y\]

**Lemma log-floor-sound**: assumes \(b > 1 \land x > 0\) \(\log b \cdot x = y\)

shows \(b \cdot y \leq x \land x < b \cdot (\text{Suc } y)\)

**Proof** –

- from `assms(1,3)` have `id: get-base (into-base b) = b by transfer auto`
- obtain `yy bb` where `log: log-main (into-base b) x = (yy, bb)`
  - by `(cases log-main (into-base b) x, auto)`
- from `log-main[OF assms(2) log] assms(3)[unfolded log-floor-def log] id show` `b \cdot y \leq x \land x < b \cdot (\text{Suc } y) by auto`

**Qed**

**Lemma log-ceiling-sound**: assumes \(b > 1 \land x > 0\) \(\log b \cdot x = y\)

shows \(x \leq b \cdot y \neq 0 \implies b \cdot (y - 1) < x\)

**Proof** –

- from `assms(1,3)` have `id: get-base (into-base b) = b by transfer auto`
- obtain `yy bb` where `log: log-main (into-base b) x = (yy, bb)`
  - by `(cases log-main (into-base b) x, auto)`
- from `log-main[OF assms(2) log, unfolded id] assms(3)[unfolded log-ceiling-def log split]` have `bnd: b \cdot yy \leq x \land x < b \cdot \text{Suc } yy` and
  - `y: y = (if x = b \cdot yy then yy else Suc yy) by auto`
- have `x \leq b \cdot y \land (y \neq 0 \implies b \cdot (y - 1) < x)`
- proof `(cases x = b \cdot yy)`
  - case `True`
    - with `y bnd assms(1)` show `?thesis by (cases yy, auto)`
  - case `False`
    - with `y bnd show `?thesis by auto``
- qed

thus `x \leq b \cdot y \neq 0 \implies b \cdot (y - 1) < x by auto`

**Qed**

Finally, we connect it to the \(\log\) function working on real numbers.

**Lemma log-floor[simp]**: assumes \(b > 1 \land x > 0\)

shows \(\log b \cdot x = \lfloor \log b \cdot x \rfloor\)

**Proof** –

- obtain `y` where `y: log-floor b \cdot x = y by auto`
- note `main = log-floor-sound[OF assms y]

from `b x have `*: \(1 < \text{real-of-int } b \cdot 0 < \text{real-of-int } (b \cdot y) 0 < \text{real-of-int } x\)`
- and `**: \(1 < \text{real-of-int } b \cdot 0 < \text{real-of-int } x 0 < \text{real-of-int } (b \cdot \text{Suc } y)\)`
  - by `auto`
- show `?thesis unfolding y`

**Proof** `(rule sym, rule floor-unique)`

- show `\text{real-of-int } (\text{int } y) \leq \log (\text{real-of-int } b) (\text{real-of-int } x)`
  - using `main(1)[folded log-le-cancel-iff[OF *, unfolded of-int-le-iff]]`
  - using `log-pow-cancel[of b y] b by auto`
- show `\log (\text{real-of-int } b) (\text{real-of-int } x) < \text{real-of-int } (\text{int } y) + 1`
  - using `main(2)[folded log-less-cancel-iff[OF **, unfolded of-int-less-iff]]`
lemmas 

proof -

show ?thesis unfolding y

proof (cases y = 0)

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

end

3 Executable algorithms for $p$-th roots

theory NthRoot-Impl

imports Log-Impl Cauchy.CauchysMeanTheorem

begin

We implemented algorithms to decide $\sqrt[3]{n} \in \mathbb{Q}$ and to compute $\lfloor \sqrt[3]{n} \rfloor$. 

8
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 $$2^{\lfloor \log_2 n \rfloor / p}$$

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 $x \leq (\text{start-value } x \ p)^p$, 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**

```plaintext
definition start-value :: int ⇒ nat ⇒ int where
    start-value n p = 2 ^ (nat \( \lfloor \text{of-int } (\log\text{-ceiling } 2 \ n) / \text{rat-of-nat } p \rfloor \))
```

**Lemma**

```plaintext
lemma start-value-main: assumes x: x ≥ 0 and p: p > 0
    shows x ≤ (start-value x \ p)^\ p ∧ start-value x \ p ≥ 0
proof (cases x = 0)
    case True with p show \thesis unfolding start-value-def True by simp
next
    case False
    with x have x: x > 0 by auto
    define l2x where l2x = \lfloor \log_2 x \rfloor
    define pow where pow = nat \( \lfloor \text{rat-of-int } l2x / \text{of-nat } p \rfloor \)
    have root \ p x = x powr \( 1 \ / \ p \) by (rule root-powr-inverse, insert x \ p, auto)
    also have \ldots = (2 powr (\log_2 x)) powr \( 1 \ / \ p \) using powr-log-cancel[\lfloor \log_2 x \rfloor] x by auto
    also have \ldots = 2 powr (\log_2 x * \( 1 \ / \ p \)) by (rule powr-powr)
    also have \log_2 x * \( 1 \ / \ p \) = \log_2 x / \ p using p by auto
    finally have r: root \ p x = 2 powr (\log_2 x / \ p) .
    have lp: \log_2 x ≥ 0 using x by auto
    hence l2pos: l2x ≥ 0 by (auto simp: l2x-def)
    have \log_2 x / \ p ≤ l2x / \ p using x \ p unfolding l2x-def
        by (metis divide-right-monotone le-of-int-ceiling of-nat-0-le-iff)
    also have \ldots ≤ \lfloor l2x / \ (p :: real) \rfloor by (simp add: ceiling-correct)
    also have l2x / \ real p = l2x / \ real-of-rat (of-nat p)
        by (metis of-rat-of-nat-eq)
```
also have \( \text{of-int } l2x = \text{real-of-rat (of-int } l2x) \) by \( \text{metis of-rat-of-int-eq} \)
also have \( \text{real-of-rat (of-int } l2x) / \text{real-of-rat (of-nat } p) = \text{real-of-rat (rat-of-int l2x / of-nat } p) \) by \( \text{metis of-rat-divide} \)
also have \( [\text{real-of-rat (rat-of-int l2x / rat-of-nat } p)] = [\text{rat-of-int l2x / of-nat } p] \) by \( \text{simp} \)

finally have \( \lceil \text{real-of-rat (rat-of-int l2x / rat-of-nat } p) \rceil = \lceil \text{rat-of-int l2x / of-nat } p \rceil \) by \( \text{simp} \)
finally have \( \lceil \text{rat-of-int l2x / of-nat } p \rceil \leq \text{real pow} \) unfolding \( \text{pow-def} \) \( \text{l2x-def} \) using \( x \) by \( \text{simp} \)

finally have \( 0 \leq \text{start-value } x \ p \) using \( p \ x \) by \( \text{auto} \)
also have \( \text{start-value } x \ p \ 
\text{emp} \ x \ p \geq 0 \) using \( \text{start-value-main } [\text{OF } x \ p] \) by \( \text{auto} \)

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

locale fixed-root =

fixes \( p \ pm :: \text{nat} \)

assumes \( p :: p = \text{Suc } pm \)

begin

function \( \text{root-newton-int-main :: int} \Rightarrow \text{int} \Rightarrow \text{int} \times \text{bool} \) where

\( \text{root-newton-int-main } x \ n = (\text{if } x < 0 \lor n < 0 \text{ then } (0, \text{False}) \text{ else } (\text{if } x ^ p \leq n \text{ then } (x, x ^ p = n) \text{ else } \text{root-newton-int-main } ((n \text{ div } x ^ pm) + x * \text{int } pm) \text{ div } (\text{int } p)) \text{ n}))) \)

by \( \text{pat-completeness auto} \)

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

\[
\text{partial-function } \text{(tailrec)} \text{ root-int-main}': \text{nat} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \times \text{bool} \text{ where}
\]

\[
\begin{align*}
| \text{code} |: & \text{root-int-main}' \text{ pm ipm ip x n} = (\text{let} \ xpm = x^{\text{pm}}; \ xp = xpm * x \text{ in } \begin{cases} 
\text{if } xp \leq n \text{ then } (x, xp = n) \\
\text{else } \text{root-int-main}' \text{ pm ipm ip } ((n \text{ div } xpm + x * \text{ipm}) \text{ div } \text{ip}) \text{ n) }
\end{cases}) \\
\end{align*}
\]

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.

\[
\text{definition } \text{root-int-main} :: \text{nat} \Rightarrow \text{int} \Rightarrow \text{int} \times \text{bool} \text{ where}
\]

\[
\text{root-int-main p n } \equiv \begin{cases} 
1, n = 1 \\
\text{else}
\end{cases}
\]

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

\[
\text{definition } \text{root-int} :: \text{nat} \Rightarrow \text{int} \Rightarrow \text{int list} \text{ where}
\]

\[
\text{root-int p x } \equiv \begin{cases} 
\text{if p = 0 then } [] \\
\text{if x = 0 then } [0] \text{ else}
\end{cases}
\]

We start with proving termination of \(\text{fixed-root} \cdot \text{root-newton-int-main}\).

\[
\text{context } \text{fixed-root}
\]

\[
\text{begin}
\]

\[
\text{lemma iteration-mono-eq: assumes xn: } x^p = (n :: \text{int}) \text{ shows } (n \text{ div } x^\text{pm} + x * \text{int pm}) \text{ div } \text{int p} = x
\]

\[
\text{proof} -
\]

\[
\begin{align*}
\text{have [simp]: } & \land n, (x + x * n) = x * (1 + n) \text{ by (auto simp: field-simps)} \\
\text{show } & ?\text{thesis unfolding xn[symmetric]} \text{ p by simp}
\end{align*}
\]

\[
\text{qed}
\]

\[
\text{lemma p0: } p \neq 0 \text{ unfolding p by auto}
\]

The following property is the essential property for proving termination of \(\text{root-newton-int-main}\).

\[
\text{lemma iteration-mono-less: assumes x: } x \geq 0 \text{ and n: } n \geq 0 \\
\text{and xn: } x^p > (n :: \text{int}) \text{ shows } (n \text{ div } x^\text{pm} + x * \text{int pm}) \text{ div } \text{int p} < x
\]

\[
\text{proof} -
\]

\[
\begin{align*}
\text{let } \&sx = (n \text{ div } x^\text{pm} + x * \text{int pm}) \text{ div } \text{int p} \\
\text{from xn have xn-le: } x^p \geq n \text{ by auto} \\
\text{from xn x n have x0: } x > 0 \\
\text{using not-le p by fastforce}
\end{align*}
\]

\[
\text{from p have xp: } x^p = x * x^\text{pm} \text{ by auto}
\]
from \(x \div n \leq n\)

by \((\text{auto simp add: minus-mod-eq-div-mult \ [symmetric] \ mod-int-pos-iff not-less power-le-zero-eq})\)

also have \(\ldots \leq x \cdot p\) using \(\times n\) by \(\text{auto}\)

finally have \(\le\): \(n \div x \cdot p \leq n\) using \(\times x\) unfolding \(\times p\) by \(\text{simp}\)

by \((\text{rule zdiv-mono1, insert le \(p\), unfold \(xp\), auto})\)

also have \(x \cdot p = x\) unfolding \(xp\) by \(\text{auto}\)

also have \(x = x\cdot p\) unfolding \(p\) by \(\text{auto simp: field-simps}\)

also have \(x = x\cdot p\) div \(int \ p\) using \(\text{p}\) by \(\text{force}\)

finally have \(\le\): \(?sx \leq x\).

\{
assume \(?sx = x\)
from \(\text{arg-cong}\) \([\text{OF this}, \ of \ \lambda x. x \cdot \text{int} \ p]\)
  have \(x = \text{int} \ p\) \(\leq (n \div x \cdot \text{pm} + x \cdot \text{int} \ p) \div \text{int} \ p = \text{int} \ p\) using \(\text{p0}\) by \(\text{simp}\)

also have \(\ldots \leq n \div x \cdot \text{pm} + x \cdot \text{int} \ p\)
  unfolding \(\text{mod-div-equality-int}\) using \(\text{p}\) by \(\text{auto}\)

finally have \(\le\): \(n \div x \cdot p \leq x\) by \(\text{auto simp: field-simps}\)

from \(\text{mult-right-mono}\) \([\text{OF this}, \ of \ \times p]\)
  have \(\ge\): \(n \div x \cdot \text{pm} + x \cdot \text{pm} \leq x \cdot p\) unfolding \(\times p\) by \(\text{auto}\)

from \(\text{div-mult-mod-eq}\) \([\text{of } \times \text{pm}\) have \(n \div x \cdot \text{pm} \cdot x \cdot \text{pm} = n \div n \mod x \cdot \text{pm}\)

by \(\text{arith}\)

from \(\ge\) \([\text{unfolded this}]\)
  have \(\le\): \(x \cdot p \leq n \div n \mod x \cdot \text{pm}\).

from \(\times n\) have \(\ge\): \(n \mod x \cdot \text{pm} \geq 0\)
  by \(\text{auto simp add: mod-int-pos-iff not-less power-le-zero-eq}\)

from \(\le\) \([\text{ge}]\)
  have \(n \geq x \cdot p\) by \(\text{auto}\)

with \(\times n\) have \(\text{False}\) by \(\text{auto}\)
\}

with \(\le\) show \(?\text{thesis}\) unfolding \(p\) by \(\text{fastforce}\)

qed

lemma \(\text{iteration-mono-lesseq}\): assumes \(x: x \geq 0\) and \(n: n \geq 0\) and \(\times n\): \(x \cdot p \geq (n :: \text{int})\)

shows \(n \div x \cdot \text{pm} + x \cdot \text{int} \ p\) \(\div \text{int} \ p \leq x\)

proof \((\text{cases } x \cdot p = n)\)
  case True
  from \(\text{iteration-mono-eq}\) \([\text{OF this}]\) show \(?\text{thesis}\) by \(\text{simp}\)

next
case False
  with \(\text{assms}\) have \(x \cdot p > n\) by \(\text{auto}\)
  from \(\text{iteration-mono-less}\) \([\text{OF } \times n\) this]\)
  show \(?\text{thesis}\) by \(\text{simp}\)

qed

termination

proof –
let $m = \lambda x \cdot n :: \text{int}$
let $m1 = \lambda (x, n). m x n$
let $m = \text{measures [m1]}$

show $\text{thesis}$

proof (relation $m$)
  fix $x n :: \text{int}$
  assume $x ^ p > n$ by auto
  assume $x > 0 \lor n > 0$
  hence $x-n :: x ^ p \leq n$ by auto
  from $x x-n$ have $x0 :: x > 0$ using $p$ by (cases $x = 0$, auto)
  from $\text{iteration-mono-less[OF x-n x]}$ $x0$
  show $((n \text{ div } x ^ pm + x \ast \text{ int } pm) \text{ div int } p, n, x, n) \in m$ by auto
qed

We next prove that $\text{root-int-main'}$ is a correct implementation of $\text{root-newton-int-main}$.

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 $\text{root-main'}$ where $\text{root-main'} \equiv \text{root-int-main'} \ast \text{pm} (\text{int } pm) (\text{int } p)$

lemmas $\text{root-main'}$-simps = $\text{root-int-main'}$.simps[of $\text{pm}$ $\text{int } pm$ $\text{int } p$]

lemma $\text{root-main'}$-newton-pos: $x \geq 0 \Longrightarrow n \geq 0 \Longrightarrow$
  $\text{root-main'} x \cdot n = \text{root-newton-int-main } x n \land (\text{root-main'} x n = (y, b) \Longrightarrow y \geq 0$
  $\land y ^ p \leq n \land b = (y ^ p = n))$

proof (induct $x n$ rule: root-newton-int-main.induct)
  case (1 $x n$)
  have $pm-x[simp]: x ^ pm \ast x = x ^ p$ unfolding $p$ by simp
  from 1 have id: $(x < 0 \lor n < 0) = \text{False}$ by auto
  note $d = \text{root-main'}$-simps[of $x n$] root-newton-int-main.simps[of $x n$] id if-False
  Let-def
  show $\text{thesis}$ unfolding $d$ using 1(2) by auto
next
  case False
  hence id: $(x ^ p \leq n) = \text{False}$ by simp
  from 1(3) 1(2) have not: $\neg (x < 0 \lor n < 0)$ by auto
  then have $x :: x > 0 \lor n = 0$
  by auto
  with $(0 \leq n)$ have $0 \leq (n \text{ div } x ^ pm + x \ast \text{ int } pm) \text{ div int } p$
  by (auto simp add: $p$ algebra-simps pos-imp-zdiv-nonneg-iff power-0-left)
  then show $\text{thesis}$ unfolding $d$ id pm-x
  by (rule 1(1)[OF not False - 1(3)])
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: $x \geq 0 \implies n \geq 0 \implies \text{root-main}' \ x \ n = (y,b) \implies b = (y \ ^{\cdot} \ p = n)$
  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: $x \geq 0 \implies n \geq 0 \implies \text{root-main}' \ x \ n = (y,b) \implies y ^{\cdot} p \leq n$
  using root-main'-newton-pos by blast

lemma root-newton-int-main-upper:
  shows $y ^{\cdot} p \geq n \implies y \geq 0 \implies n \geq 0 \implies \text{root-newton-int-main} \ y \ n = (x,b)$
  proof (induct y n rule: root-newton-int-main.induct)
    case (1 y n)
    from 1(3) have y0: $y \geq 0$ ,
    then have $y > 0 \lor y = 0$
      by auto
    from 1(4) have n0: $n \geq 0$ .
    define $y'$ where $y' = (n \div (y ^{\cdot} p + y * \text{int} \ p)) \div \text{int} \ p$
    from $(y > 0 \lor y = 0) \implies n \geq 0$ have y'0: $y' \geq 0$
      by (auto simp add: y'-def \ p algebra-simps pos-imp-zdiv-nonneg-iff power-0-left)
    let ?rt = root-newton-int-main
    from 1(5) have rt: $\forall \ y \ n \ (x,b)$ by auto
    from y0 n0 have not: $(y < 0 \land n < 0) \lor (y < 0 \land n < 0) = \text{False}$ 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 ?thesis
    proof (cases $y ^{\cdot} p \leq n$)
      case False note yyn = this
      with rt have rt: $\forall \ y' \ n = (x,b)$ by simp
    show ?thesis
      proof (cases $n \leq y' ^{\cdot} p$)
        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 \( n0 \) \( y0' \) by simp
from \( yyn \) have \( yyn: y' > n \) by simp
from \( False \) have \( yyn': n > y' \) 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 \( False \) \( y0' \) 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 \( yyn0: y \neq 0 \) unfolding \( p \) by auto
  from \( yyn0 \) \( y0 \) have \( yyn: ?y > 0 \) by auto
  from \( n0 False \) \( n0 \) \( ?n > 0 \) by auto
  define \( Y \) where \( Y = ?y \cdot of-int pm \)
  define \( NY \) where \( NY = ?n / ?y ^ pm \)
  note \( pos-intro = divide-nonneg-pos add-nonneg-nonneg mult-nonneg-nonneg \)
  have \( NY0: NY > 0 \) unfolding \( NY-def \) using \( y0 \) \( n0 \)
    by (metis \( NY-def \) zero-less-divide-iff zero-less-power)
  let \( ?ls = NY \# replicate pm ?y \)
  have \( prod: \prod: replicate pm ?y = ?y ^ pm \)
    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
qed
also have \( \ldots = (NY + Y) / \) real p  
  by (simp add: mean-def sum p)  
finally have \( *: \) root p \(?n < (NY + Y) / \) real p  
  by (metis neq0_conv p0 real-root-pow-pos)  
also have \( \ldots < ((NY + Y) / \) 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 \( y \) \(^p \) + \( y \) \(* \) int \( p \)  
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 \(?n / \) \(?y \) \(^p \) < of-int (floor (\(?n / \) \(?y \) \(^p \)))) + 1  
  by (rule divide-less-floor1)  

also have floor (\(?n / \) \(?y \) \(^p \)) = \( n \) div \( y \) \(^p \)  
  unfolding div-is-floor-divide-real by (metis of-int-power)  

finally have NY < of-int (\( n \) div \( y \) \(^p \)) + 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 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 \([S] \leq s \) using less floor-le-iff by auto  
  hence [rat-of-int \( S \) / rat-of-nat \( p \)] \leq [rat-of-int \( s \) / rat-of-nat \( p \)]  
    using f1 f3 f4 by (metis div-is-floor-divide-real zdiv-mono1)  
  hence \([S / \) real \( p] \leq [\) rat-of-int \( s \) / rat-of-nat \( p \)] \)  
    using f1 f2 f3 f4 by (metis div-is-floor-divide-real floor-div-pos-int)  
  hence \( S / \) real \( p \leq \) 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 \( p \leq \) of-int (\( s \) div \( p \)) + 1  
  note this[unfolded S-def s-def]  
\}  

hence ge: \( of-int y' + 1 \geq (NY + Y) / p \) unfolding y'-def  
  by simp  

have pos1: \( (NY + Y) / p \geq 0 \) unfolding Y-def NY-def  
  by (intro divide-nonneg-pos add-nonneg-nonneg mult-nonneg-nonneg,  
    insert y0 n0 p0, auto)  

have pos2: \( of-int y' + (1 :: rat) \geq 0 \) using y'0 by auto  

have ineq2: \((\text{of-int } y' + 1)^p \geq ((NY + Y) / p)^p\)
by \((\text{rule power-mono(OF ge pos1)})\)
from order.strict-trans2[OF ineq1 ineq2]
have \(?n < \text{of-int } ((x + 1)^p)\)
unfolding x
by \((\text{metis of-int-1 of-int-add of-int-power)}\)
thus \(n < (x + 1)^p\) using of-int-less-iff by blast qed

next

have \(?n < \text{of-int } ((x + 1)^p)\)
unfolding x
by \((\text{metis of-int-1 of-int-add of-int-power)}\)
thus \(n < (x + 1)^p\) using of-int-less-iff by blast qed

end

Now we can prove all the nice properties of \texttt{root-int-main}.


\textbf{lemma} \texttt{root-main'-upper}: \n\begin{align*}
  x^p & \geq n \implies x \geq 0 \implies n \geq 0 \implies \text{root-main'} \ x \ n = (y,b) \implies n < (y + 1)^p
\end{align*}

using root-newton-int-main-upper[of n x y b]
\texttt{root-main'}[of x n]
by auto

\textbf{end}

}\begin{proof}\ \ (\text{cases } p = 0)\ \end{proof}

\textbf{case} True
\begin{align*}
  \text{with & \ \ \ \ rm[unfolded root-int-main-def]} \\
  \text{have \ } \ y \ = \ 1 \ \text{and \ } b \ = \ (n = 1) \ \text{by auto} \\
  \text{show \ } & \ ?thesis \ unfolding \ True \ y \ b \ using \ n \ by \ auto
\end{align*}

\textbf{next}

\textbf{case} False
\begin{align*}
  \text{from & \ \ \ \ False \ have \ p-0; \ p > 0 \ by \ auto} \\
  \text{from & \ \ \ \ False \ have \ (p = 0) = False \ by \ simp} \\
  \text{from \ rm[unfolded root-int-main-def this Let-def]} \\
  \text{have \ rm: \ root-int-main' \ (p - 1) \ (int (p - 1)) \ (int p) \ (start-value n p) \ n = (y,b) \ by \ simp} \\
  \text{from \ start-value[OF n p-0] \ have \ start: \ n \leq (start-value n p)^p \ 0 \leq \ start-value n p \ by \ auto} \\
  \text{interpret \ fixed-root \ p \ p - 1} \\
  \text{by \ (unfold-locales, insert False, auto)} \\
  \text{from \ root-main'-pos[OF start(2) n rm] \ have \ y: \ y \geq 0} \\
  \text{from \ root-main'-sound[OF start(2) n rm] \ have \ b: \ b = (y^p = n)} \\
  \text{from \ root-main'-lower[OF start(2) n rm] \ have \ low: \ y^p \leq n} \\
  \text{from \ root-main'-upper[OF start n rm] \ have \ up: \ n < (y + 1)^p}.
\end{proof}
{  
  assume n: \( x \cdot p = n \) and \( x \geq 0 \)  
  with low up have low: \( y \cdot p \leq x \cdot p \) and up: \( x \cdot p < (y+1) \cdot p \) by auto  
  from power-strict-mono[of x y, OF - x p-0] low have x: \( x \geq y \) by arith  
  from power-mono[of \( (y + 1) \cdot x \cdot p \)] y up have y: \( y \geq x \) by arith  
  from x y have x = y by auto  
  with b n  
  have y = x ∧ b by auto  
}  
thus \(?thesis\) using b low up y by auto  
qed

lemma root-int-main: assumes n: \( n \geq 0 \)  
and rm: root-int-main p n = \((y,b)\)  
shows y ≥ 0 b = \((y \cdot p = n)\) p > 0 \(\Rightarrow\) y ≤ n p > 0 \(\Rightarrow\) n ≤ \((y + 1) \cdot p\)  
p > 0 \(\Rightarrow\) x ≥ 0 \(\Rightarrow\) x = n \(\Rightarrow\) y = x ∧ b  
using root-int-main-all[OF n rm, of x] by blast+

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

  case False  
  hence p: \((p = 0) = False\) and p0: \( p > 0 \) by auto  
  note d = root-int-def p if-False Let-def  
  show \(?thesis\)  
  proof (cases x = 0)  
    case True  
    thus \(?thesis\) unfolding d using p0 by auto  
  next  
    case False  
    with x p have cond: \((x = 0) = False\) \((x < 0 \land even p) = False\) by auto
  qed

18
obtain $y \ b$ where $rt\colon \text{root-int-main} \ p \ | x \ = \ (y, b)$ by force

have $\abs x \geq 0$ by auto

note $\text{rm} = \text{root-int-main}[OF this rt]$

have $\Theta$ thesis =
  (set (case root-int-main $p \ | x \ = \ (y, b)$)
  unfolding $d$ cond by blast
  also have (case root-int-main $p \ | x \ = \ (y, True)$ ⇒
  if even $p$ then $[y, -y]$ else $[\text{sgn} x \ * y]$ | (y, False)⇒ [])
  unfolding $r$ by auto
  also have set $\Theta$ = \{ $y, y $ $\hat{=} p \ = \ x$\} (is - = $\Theta$)

proof −
{
  fix $z$
  assume $\text{id}x: z ^ \ ^ \ ^ \ p = x$
  hence eq: (abs $z$) ^ \ ^ $p = abs x$ by (metis power-abs)
  from $\text{id}x \ p0$ have $z: z \neq 0$ unfolding $p$ by auto
  have $(y, b) = (|z|, True)$
    using $\text{rm}(5)[OF p0 - eq]$ by auto
  hence id: $y = abs z$ $b = True$ by auto
  have $z \in set \Theta$ unfolding id using $z$ by (auto simp: $\text{id}[\text{symmetric}]$,
  cases $z < 0, auto$)
  }
moreover
{
  fix $z$
  assume $z: z \in set \Theta$
  hence $b: b = True$ by (cases $b$, auto)
  note $z = z[\text{unfolded} b \$ if-True$]
  from $\text{rm}(2)$ $b$ 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 $z \in \Theta$
  proof
    assume odd: odd $p$
    with $z$ have $z = \text{sgn} x \ * y$ by auto
    hence $z ^ \ ^ \ ^ \ p = (\text{sgn} x \ * y) ^ \ ^ \ p$ by auto
    also have $\ldots$ = $\text{sgn} x ^ \ ^ \ ^ \ p \ * y ^ \ ^ \ p$ unfolding $\text{power-mul-distrib}$ by auto
    also have $\ldots$ = $\text{sgn} x ^ \ ^ \ ^ \ p \ * abs x$ unfolding $yx$ by simp
    also have $\text{sgn} x ^ \ ^ \ ^ \ p = \text{sgn} x$ using $x$ odd by auto
    also have $\text{sgn} x \ * abs x = x$ by (rule $\text{mul-sgn-ns}$)
    finally show $z \in \Theta$ by auto
  next
    assume even: even $p \ \land x \geq 0$
    from $z$ even have $z = y \ ^ \ \ ^ \ \ ^ \ z = -y$ by auto
    hence id: $abs z = y$ using $y$ by auto
    with $yx x$ even have $z: z \neq 0$ using $p0$ by (cases $y = 0, auto$)
    have $z ^ \ ^ \ ^ \ p = (\text{sgn} z \ * abs z) ^ \ ^ \ p$ by (simp add: $\text{mul-sgn-ns}$)
also have \( \ldots = (\text{sgn } z \cdot y) \cdot p \) using \( \text{id} \) by \( \text{auto} \)
also have \( \ldots = (\text{sgn } z) \cdot p \cdot y \cdot p \) unfolding \( \text{power-mult-distrib} \) by \( \text{simp} \)
also have \( \ldots = \text{sgn } z \cdot p \cdot x \) unfolding \( yz \) using \( \text{even} \) by \( \text{auto} \)
also have \( \text{sgn } z \cdot p = 1 \) using \( \text{even } z \) by \( \text{(auto)} \)
finally show \( z \in \text{?rhs} \) by \( \text{auto} \)

\}
ultimately show \( \text{?thesis} \) by \( \text{blast} \)
qed
finally show \( \text{?thesis} \) by \( \text{auto} \)
qed
qed

\textbf{lemma root-int-pos}: assumes \( x: x \geq 0 \) and \( \text{ri: root-int } p \ x = y \# y s \)
shows \( y \geq 0 \)

\textbf{proof –}
from \( x \) have abs: \( \text{abs } x = x \) by \( \text{auto} \)
note \( \text{ri} = \text{ri[unfolded root-int-def Let-def abs]} \)
from \( \text{ri} \) have \( p: (p = 0) = \text{False} \) by \( \text{(cases } p, \text{ auto)} \)
note \( \text{ri} = \text{ri[unfolded } p \text{ if-False]} \)
show \( \text{?thesis} \)
\textbf{proof (cases } x = 0 \text{)}
  case True
  with \( \text{ri} \) show \( \text{?thesis} \) by \( \text{auto} \)
next
  case False
  hence \( (x = 0) = \text{False} (x < 0 \land \text{even } p) = \text{False} \) using \( x \) by \( \text{auto} \)
  note \( \text{ri} = \text{ri[unfolded this if-False]} \)
  obtain \( y' \) \( b' \) where \( r: \text{root-int-main } p \ x = (y',b') \) by \( \text{force} \)
  note \( \text{ri} = \text{ri[unfolded this]} \)
  hence \( y: y = (\text{if even } p \text{ then } y' \text{ else } \text{sgn } x \cdot y') \) by \( \text{(cases } b', \text{ auto)} \)
  from root-int-main(1)|\( \text{OF } x \ r \) have \( y': 0 \leq y' \).
  thus \( \text{?thesis} \) unfolding \( y \) using \( x \) \( \text{False} \) by \( \text{auto} \)
qed
qed

\textbf{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.

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

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

\textbf{lemma root-int-floor-pos-pos}: assumes $x: x \geq 0$
\text{ shows root-int-floor-pos p x $\geq 0$
\text{ using root-int-main(1)[OF \, x, \, of \, p]}$
\text{ unfolding root-int-floor-pos-def}$
\text{ by (cases root-int-main p \ x, \ auto)}

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

\textbf{lemma root-int-floor-pos}: assumes $x: x \geq 0$
\text{ shows root-int-floor-pos p x $=$ floor (root p \ (of-int x))}$
\text{ proof (cases p $=$ 0)}
\text{ case True$
\text{ thus \ ?thesis by (simp add: root-int-floor-pos-def)}$
\text{ next$
\text{ case False$
\text{ hence p: p $>$ 0 by auto}
\text{ let \ ?s1 $=$ real-of-int (root-int-floor-pos p x)
\text{ let \ ?s2 $=$ root p \ (of-int x)
\text{ from x have s1: ?s1 $\geq 0$
\text{ by (metis of-int-0-le-iff root-int-floor-pos-pos)
\text{ from x have s2: ?s2 $\geq 0$
\text{ by (metis of-int-0-le-iff real-root-pos-pos-le)
\text{ from s1 have s11: ?s1 $+$ 1 $\geq 0$ by auto
\text{ have id: ?s2 $\cdot$ p $=$ of-int x using x
\text{ by (metis p of-int-0-le-iff real-root-pow-pos2)
\text{ show \ ?thesis$
\text{ proof (rule floor-unique[ symmetric])
\text{ show \ ?s1 $\leq$ ?s2
\text{ unfolding compare-pow-le-iff[OF \, p \, s1 \, s2, \ symmetric]
\text{ unfolding id
\text{ using root-int-floor-pos-lower[OF \, False \, x]
\text{ by (metis of-int-le-iff of-int-power)
\text{ show \ ?s2 $<$ ?s1 $+$ 1
\text{ unfolding compare-pow-less-iff[OF \, p \, s2 \, s11, \ symmetric]
\text{ unfolding id
\text{ using root-int-floor-pos-upper[OF \, False \, x]
\text{ by (metis of-int-add of-int-less-iff of-int-power of-int-1)
\text{ qed
\text{ qed$
\text{ lemma root-int-ceiling-pos}: assumes x: x \geq 0$
\text{ shows root-int-ceiling-pos p x $=$ ceiling (root p \ (of-int x))}$
\text{ proof (cases p $=$ 0)}
case True
thus \(?thesis\) by (simp add: root-int-ceiling-pos-def)
next
case False
hence \(p > 0\) by auto
obtain \(y b\) where \(s\): root-int-main \(p\ \ x = (y,b)\) by force
note \(rm = \text{root-int-main}[OF\ x\ s]\)
note \(rm = \text{rm}(1-2)\ \ \text{rm}(3-5)[OF\ p]\)
from \(\text{rm}(1)\) 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 \(?thesis\)
proof (cases \(b\))
case True
hence id : \(?s = y\) unfolding \(s\ \ d\) using \(p\) by auto
from \(\text{rm}(2)\) True have \(xy\) : \(x = y^p\) by auto
show \(?thesis\) unfolding id unfolding \(xy\) using \(y\)
  by (simp add: \(p\ \text{real-root-power-cancel}\))
next
case False
hence id : \(?s = \text{root-int-floor-pos}\ p\ x + 1\) unfolding \(d\ \text{root-int-floor-pos-def}\)
  using \(s\ \ p\) by simp
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 \(x = 0,\ \text{auto}\))
show \(?thesis\) unfolding id root-int-floor-pos[OF \(x\)]
proof (rule \text{ceiling-unique}[symmetric])
  show \(?sx \leq \text{real-of-int}\ (\lceil \text{root}\ p\ (\text{of-int}\ x) \rceil + 1)\)
    by (metis \(\text{of-int-add}\ \text{real-of-int-floor-add-one-ge}\ \text{of-int-1}\))
  let \(?l = \text{real-of-int}\ (\lceil \text{root}\ p\ (\text{of-int}\ x) \rceil + 1) - 1\)
  let \(?m = \text{real-of-int}\ \lceil \text{root}\ p\ (\text{of-int}\ x) \rceil\)
  have \(?l = \ ?m\) by simp
also have \(\ldots < \ ?sx\)
proof -
  have le : \(?m \leq \ ?sx\) by (rule \text{of-int-floor-le})
  have neg : \(?m \neq \ ?sx\)
  proof
    assume \(?m = \ ?sx\)
    hence \(?m^p = \ ?sx^p\) by auto
    also have \(\ldots = \text{of-int}\ x\) using \(x\ \text{False}\)
    by (metis \(\text{real-root-ge-0-iff}\ \text{real-root-pow-pos}\\text{2}\ \text{root-int-floor-pos}\ \text{root-int-floor-pos-pos}\)
    finally have \(xs\) : \(x = \lceil \text{root}\ p\ (\text{of-int}\ x) \rceil ^ p\)
      by (metis \(\text{floor-power}\ \text{floor-of-int}\)
    hence \(\lceil \text{root}\ p\ (\text{of-int}\ x) \rceil \in \text{set}\ (\text{root-int}\ p\ x)\) using \(p\) by simp
    hence root-int \(p\ \ x \neq \emptyset\) by force
    with \(s\ \text{False}\ : \(p \neq 0\) \ x \text{0}\) show False unfolding root-int-def
      by (cases \(p,\ \text{auto}\))
definition root-int-floor p x = (if x ≥ 0 then root-int-floor-pos p x else − root-int-ceiling-pos p (− x))
definition root-int-ceiling p x = (if x ≥ 0 then root-int-ceiling-pos p x else − root-int-floor-pos p (− x))

lemma root-int-floor[simp]: root-int-floor p x = floor (root p (of-int x))
proof –
  note d = root-int-floor-def
  show ?thesis
  proof (cases x ≥ 0)
    case True
      with root-int-floor-pos[OF True, of p] show ?thesis unfolding d by simp
    next
    case False
      hence − x ≥ 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 ≥ 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

3.4 Downgrading algorithms to the naturals
definition root-nat-floor :: nat ⇒ nat ⇒ int where
  root-nat-floor p x = root-int-floor-pos p (int x)
definition root-nat-ceiling :: nat ⇒ nat ⇒ int where
root-nat-ceiling p x = root-int-ceiling-pos p (int x)

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

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

lemma root-nat-floor-lower: assumes p0: p ≠ 0
shows root-nat-floor p x ≤ 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) > 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 int x p]
by auto

lemma root-nat: assumes p0: p ≠ 0 ∨ x ≠ 1
shows set (root-nat p x) = {y. y ≈ p = x}
proof –
{ fixed y
assume y ∈ set (root-nat p x)
note y = this[unfolded root-nat-def]
then obtain yi ys where ri: root-int p x = yi ≠ ys by (cases root-int p x,
  auto)
with y have y: y = nat yi by auto
from root-int-pos[OF - ri] have yi: 0 ≤ yi by auto
from root-int[of p int x] p0 ri have yi ≈ p = x by auto
from arg-cong[OF this, of nat] yi have nat yi ≈ p = x
  by (metis nat-int nat-power-eq)
  hence y ∈ {y. y ≈ p = x} using y by auto
} moreover
{ fixed y
assume ys: y ≈ p = x
hence y: int y ≈ p = int x
  by (metis of-nat-power)
hence set (root-int p (int x)) ≠ {} using root-int[of p int x] p0
  by (metis (mono-tags) One-nat-def (y ≈ p = x) empty-Collect-eq nat-power-eq-Suc-0-iff)
then obtain yi ys where ri: root-int p (int x) = yi ≠ ys
  by (cases root-int p (int x), auto)
from root-int-pos[OF - this] have yi p: yi ≥ 0 by auto
from root-int[of p int x, unfolded ri] p0 have yi: yi ^ p = int x by auto
with y have int y ^ p = yi ^ p by auto
from arg-cong[OF this, of nat] have id: y ^ p = nat yi ^ p
  by (metis (y ^ p = x) nat-int nat-power-eq yi yip)

{ assume p: p ≠ 0
  hence p0: p > 0 by auto
  obtain yy b where rm: root-int-main p (int x) = (yy,b) by force
  from root-int-main(5)[OF - rm p0 - y] have yy = int y and b = True by auto
note rm = rm[unfolded this]
  unfolding root-nat-def p root-int-def using p0 p yx
  by auto
}
moreover
{ assume p: p = 0
  with p0 have x ≠ 1 by auto
  with y p have False by auto
}
ultimately have y ∈ set (root-nat p x) by auto
ultimately show ?thesis by blast
qed

3.5 Upgrading algorithms to the rationals

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

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

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

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

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

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

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

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

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

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

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

The major motivation for developing the precise algorithms was given by CeTA [2], a tool for 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 \( a = \sqrt{4} = 2, b = -\sqrt{9} = -3.\)

4.1 The Babylonian method

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

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

until \( x_i^2 \approx n \). Note that if \( x_0^2 \geq n \), then for all \( i \) we have both \( x_i^2 \geq n \) and \( x_i \geq x_{i+1} \).
4.2 The Babylonian method using integer division

First, the algorithm is developed for the non-negative integers. Here, the division operation \( \frac{x}{y} \) is replaced by \( \text{of-int} \ \frac{x}{\text{of-int} \ y} \). Note that replacing \( \lfloor \frac{x}{\text{of-int} \ y} \rfloor \) by \( \lceil \frac{x}{\text{of-int} \ y} \rceil \) 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{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.

```plaintext
definition sqrt-int-main' :: int ⇒ int ⇒ int × bool
where
[simp]: sqrt-int-main' x n = root-int-main' 1 1 2 x n

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

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

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

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

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

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

lemma sqrt-int-pos: assumes res: sqrt-int x = Cons s ms
  shows s ≥ 0
proof
  note res = res[unfolded sqrt-int-code Let-def, simplified]
  from res have x0: x ≥ 0 by (cases ?thesis, auto)
  obtain ss b where call: sqrt-int-main x = (ss, b) by force
  from res[unfolded call] x0 have ss = s
    by (cases b, cases ss = 0, auto)
  from root-int-main(1)[OF x0 call[unfolded this sqrt-int-main-def]]
  show ?thesis .
qed

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

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

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

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

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

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

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

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

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

definition sqrt-int-ceiling x = root-int-ceiling 2 x
4.3 Square roots for the naturals

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

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

lemma sqrt-nat[simp]: set (sqrt-nat x) = \{ y | y * y = x \}
  unfolding sqrt-nat-def by (simp add: power2_eq_square)

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

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

lemma sqrt-nat-floor[simp]: sqrt-nat-floor x = floor (sqrt (of-real x))
  unfolding sqrt-nat-floor-def by simp

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 (of-int x)
  unfolding sqrt-nat-ceiling-def root-nat-ceiling-def by simp

lemma sqrt-nat-ceiling[simp]: sqrt-nat-ceiling x = ceil (sqrt (of-real x))
  unfolding sqrt-nat-ceiling-def by simp

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) ⇒ map (sqrt-int n) of
  unfolding sqrt-rat-def root-rat-def by simp

lemma sqrt-rat[simp]: sqrt-rat x = map (sqrt (of-int z)) (sqrt-int z)
  unfolding sqrt-rat-def by simp

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

lemma \( \text{sqrt-rat-pos} \): assumes \( \text{sqrt} \): \( \text{sqrt-rat} \ x = \text{Cons} \ s \ ms \)
shows \( s \geq 0 \)
proof –
obtain \( z \) \( n \) where \( q \): quotient-of \( x = (z,n) \) by force
let \( ?sz = \text{sqrt-int} z \)
let \( ?sn = \text{sqrt-int} n \)
from \( q \) have \( n > 0 \) by (rule quotient-of-denom-pos)
from \( \text{sqrt} \) obtain \( sz \) \( mz \) where \( sz \): \( ?sz = sz \# mz \) by (cases \( ?sn \), auto)
from \( \text{sqrt} \) obtain \( sn \) \( mn \) where \( sn \): \( ?sn = sn \# mn \) by (cases \( ?sn \), auto)
from \( \text{sqrt-int-pos} \) \( OF sz \) \( \text{sqrt-int-pos} \) \( OF sn \) have \( \text{pos} \): \( 0 \leq sz \ 0 \leq sn \) by auto
from \( \text{sqrt} \) \( sz \) \( sn \) have \( s \): \( s = \text{of-int} sz / \text{of-int} sn \) by auto
show \( ?thesis \) unfolding \( s \) using \( \text{pos} \)
by (metis of-int-0-le-iff zero-le-divide-iff)

qed

definition \( \text{sqrt-rat-floor} :: \text{rat} \Rightarrow \text{int} \) where
\( \text{sqrt-rat-floor} \ x = \text{root-rat-floor} 2 \ x \)

lemma \( \text{sqrt-rat-floor-code}(code) \): \( \text{sqrt-rat-floor} \ x = (\text{case quotient-of} \ x \ of \ (a,b) \Rightarrow \text{sqrt-int-floor} (a * b) \div b) \)
unfolding \( \text{sqrt-rat-floor-def} \) root-rat-floor-def by (simp add: sqrt-def)

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

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

lemma \( \text{sqrt-rat-ceiling-code}(code) \): \( \text{sqrt-rat-ceiling} \ x = - (\text{sqrt-rat-floor} (-x)) \)
unfolding \( \text{sqrt-rat-ceiling-def} \) root-rat-floor-def by simp

lemma \( \text{sqrt-rat-ceiling} \) \( \text{sqrt-rat-floor} \) \( \text{root-rat-ceiling-def} \) by simp

lemma \( \text{sqrt-rat-of-int} \): assumes \( x \): \( x * x = \text{rat-of-int} i \)
shows \( \exists \ j :: \text{int} \ j * j = i \)
proof –
from \( x \) have \( \text{mem} \): \( x \in \text{set} (\text{sqrt-rat} \ (\text{rat-of-int} \ i)) \) by simp

32
from \( x \) have \( \text{rat-of-int } i \geq 0 \) by (metis zero-le-square)

hence \( \varepsilon : \text{quotient-of } (\text{rat-of-int } i) = (i, 1) \) by (metis quotient-of-int)

have \( 1 : \text{sqrt-int } 1 = [1, -1] \) by code-simp

from mem sqrt-rat-code \* split 1

have \( x : x \in \text{rat-of-int } \{ y. y \times y = i \} \) by auto

thus \?thesis by auto

qed

4.5 Approximating square roots

The difference to the previous algorithms is that now we abort, once the distance is below \( \epsilon \). 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 \Rightarrow 'a \Rightarrow 'a where

\[ \text{sqrt-approx-main-impl } \varepsilon \ n \ x \equiv \begin{cases} \text{if } x \times x - n < \varepsilon \text{ then } x \text{ else } \text{sqrt-approx-main-impl } \varepsilon \ n \left( \frac{n}{x + x} \right) \end{cases} \]

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

locale sqrt-approximation =

fixes \( \varepsilon :: 'a :: \{ \text{linordered-field, floor-ceiling} \} \)

and \( n :: 'a \)

assumes \( \varepsilon : \varepsilon > 0 \)

and \( n : n > 0 \)

begin

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

sqrt-approx-main \( x \equiv \begin{cases} \text{if } x > 0 \text{ then } \text{if } x \times x - n < \varepsilon \text{ then } \text{sqrt-approx-main } \varepsilon \ n \left( \frac{n}{x + x} \right) \text{ else } 0 \end{cases} \)

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 \( \text{rat} \) 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 –

define \( er \) where \( er \ x \equiv (x \times x \ / \ n - 1) \) for \( x \)

define \( c \) where \( c = 2 \times n / \varepsilon \)
\begin{verbatim}
\begin{verbatim}
define m where m x = \textbf{nat} \{ c * er x \} for x
have c: c > 0 unfolding c-def using n \in \textbf{auto}
show \textbf{thesis}
proof
  show \textbf{wf} \{ measures \{m\} \} by simp
next
  fix x
assume x: \textbf{0 < x} and xe: \textbf{- x * x - n < \varepsilon}
define y where y = \textbf{(n / x + x) / 2}
show \textbf{(n / x + x) / 2,} x \in \textbf{measures \{m\}} unfolding y-def[symmetric]
proof (rule measures-less)
  from n have inv-n: \textbf{1 / n > 0} by auto
  from xe have x * x - n \geq \varepsilon by simp
  from this[unfolded \textbf{malt-le-cancel-left-pos}[OF inv-n, \of \varepsilon, symmetric]]
have erxen: er x \geq \varepsilon / n unfolding \textbf{er-def} using n by (simp add: field-simps)
have ev: \textbf{e / n > 0} and ne: \textbf{n / \varepsilon > 0} using \textbf{e n by auto}
  from en erxen have erx: er x > 0 by linarith
  have pos: \textbf{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 \textbf{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 \textbf{er-def using x}
  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 erx erx pos
    by (auto simp: field-simps)
also have \ldots = er x / 4 using erx by (simp add: field-simps)
finally have er-y-x: er y \leq er x / 4 by linarith
  from erxen have c * er x \geq 2 unfolding \textbf{c-def mult-le-cancel-left-pos}[OF ne, of - er x, symmetric]
    using n \in \textbf{by (auto simp: field-simps)}
hence pos: \textbf{c * er x > 0} \{c * er x\} \geq 2 by auto
show m y < m x unfolding \textbf{m-def nat-mono-iff}[OF pos(1)]
proof
  have \textbf{c * er y} \leq \textbf{c * (er x / 4)}
    by (rule floor-mono, unfold mult-le-cancel-left-pos[OF c], rule er-y-x)
also have \ldots \leq \textbf{c * er x / 4 + 1} by auto
also have \ldots \leq \textbf{c * er x}
    by (rule floor-mono, insert pos(2), simp add: field-simps)
finally show \textbf{c * er y < c * er x} .
qed
qed
qed

Once termination is proven, it is easy to show equivalence of sqrt-approx-main-impl
and sqrt-approx-main.
\end{verbatim}
\end{verbatim}
\end{verbatim}
lemma sqrt-approx-main-impl: \( x > 0 \implies \text{sqrt-approx-main-impl}\ \varepsilon\ n\ x = \text{sqrt-approx-main}\ x \)

proof (induct rule: sqrt-approx-main.induct)
  case (1 x)
  hence x: \( x > 0 \) by auto
  hence nx: \( 0 < (n / x) + x / 2 \) using \( n \) by (auto intro: pos-add-strict)
  note simps = sqrt-approx-main-impl.simps[of - x] sqrt-approx-main.simps[of x]
  show ?case
  proof
    (cases x * x - n < \( \varepsilon \))
    case True
    thus ?thesis unfolding simps using x by auto
  next
    case False
  show ?thesis unfolding simps using x False by auto
  qed
qed

Also soundness is not complicated.

lemma sqrt-approx-main-sound: assumes \( x : x > 0 \) and \( xx : x * x > n \)
  shows \( \sqrt{x * x} > n \land \sqrt{x * x} - n < \varepsilon \)
  using assms
  proof (induct 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: \( \sqrt{x * x} = \text{sqrt-approx-main}\ ?y \) by simp
        from n x have ?y > 0 by (auto intro: pos-add-strict)
        note IH = 1(1)[OF x(1) False y]
        from x have \( x_4 : 4 * x * x > 0 \) by (auto intro: mult-sign-intros)
        show ?thesis unfolding simp
          proof (rule IH)
            show n < ?y * ?y
              unfolding mult-less-cancel-left-pos[OF x_4, 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 \(4 \times x \times x \times n < 4 \times x \times x \times (\gamma y \times \gamma y)\) unfolding id

by (simp add: field-simps)

qed

qed

qed

end

It remains to assemble everything into one algorithm.

definition sqrt-approx :: 'a :: {linordered-field, floor-ceiling} ⇒ 'a ⇒ 'a

where

sqrt-approx \(\varepsilon \times x\) ≡

if \(\varepsilon > 0\)

then if \(x = 0\) then 0 else let \(xpos = \text{abs} \ x\) in

sqrt-approx-main-impl \(\varepsilon \times xpos\) \(xpos + 1\)

else 0


lemma sqrt-approx: assumes \(\varepsilon > 0\)

shows \(|\sqrt{\varepsilon \times x} - \sqrt{\varepsilon \times x}| < \varepsilon\)

proof (cases \(x = 0\))

next
case False

let \(\text{?x} = |x|\)

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

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

define sqrt where \(\text{sqrt} = \text{?sqrt}\)

from False have \(x \cdot \text{?x} > 0 \wedge \text{?x} + 1 > 0\) by auto

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

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

from False \(\varepsilon\) have \(\text{sqrt-approx} \ (\varepsilon \times x) = \text{?sqrti}\)

unfolding sqrt-approx-def by (simp add: Let-def)

also havest sqrt-approx \(\varepsilon \times x = \text{?sqrti}\)

unfolding sqrt-approx-def by (rule sqrt-approx-main-impl, auto)

finally have id: sqrt-approx \(\varepsilon \times x = \text{sqrt}\) unfolding sqrt-def

have sqrt: \(\sqrt{\varepsilon} \times \sqrt{x} > \varepsilon \wedge \sqrt{\varepsilon} \times \sqrt{x} - \varepsilon \times x < \varepsilon\)

unfolding sqrt-def

by (rule sqrt-approx-main-sound[OF \(x(2)\)], insert \(x\) mult-pos-pos[OF \(x(1) \times (x(1))\)],

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 ¬ (\(\exists \ i :: \text{rat.} \ i \times i = 2\))

proof

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

thus ?thesis by simp

qed
Testing speed

\[ \neg (\exists \, i :: \text{int} \cdot i \ast i = 1234567890123456789012345678901234567890) \]

proof

\[
\begin{align*}
& \text{have set } (\sqrt{\text{int} 1234567890123456789012345678901234567890}) = \left\{ \right\} \text{ by eval} \\
& \text{thus } \text{thesis by simp} \\
& \text{qed}
\end{align*}
\]

The following test

\[
\begin{align*}
& \text{value } \text{let } \varepsilon = 1 / 100000000 :: \text{rat}; \ s = \text{sqrt-approx } \varepsilon \ 2 \text{ in } (s, s \ast s - 2, |s \ast s - 2| < \varepsilon) \\
& \text{results in } (1.4142135623731116, 4.7320076148612e-14, \text{True}).
\end{align*}
\]

end

Acknowledgements

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

References
