Computing N-th Roots using the Babylonian Method*

René Thiemann

September 19, 2015

Abstract

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

Contents

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

2 Executable algorithms for p-th roots 3
  2.1 Logarithm ................................. 4
  2.2 Computing the p-th root of an integer number .......................... 4
  2.3 Floor and ceiling of roots .............................. 7
  2.4 Downgrading algorithms to the naturals ............................... 8
  2.5 Upgrading algorithms to the rationals ............................... 9

3 Executable algorithms for square roots 10
  3.1 The Babylonian method ................................. 10
  3.2 The Babylonian method using integer division .......................... 10
  3.3 Square roots for the naturals ............................... 12
  3.4 Square roots for the rationals ............................... 13
  3.5 Approximating square roots ............................... 14
  3.6 Some tests ........................................ 15

*This research is supported by FWF (Austrian Science Fund) project P22767-N13.
1 Auxiliary lemmas which might be moved into the Isabelle distribution.

theory Sqrt-Babylonian-Auxiliary
imports ∼∼/src/HOL/Transcendental
begin

lemma mod-div-equality-int: (n :: int) div x * x = n - n mod x
⟨proof⟩

lemma log-pow-cancel[simp]: a > 0 ⇒ a ≠ 1 ⇒ log a (a ^ b) = b
⟨proof⟩

lemma real-of-rat-floor[simp]: floor (real-of-rat x) = floor x
⟨proof⟩

lemma abs-of-rat[simp]: |real-of-rat x| = real-of-rat |x|
⟨proof⟩

lemma real-of-rat-ceiling[simp]: ceiling (real-of-rat x) = ceiling x
⟨proof⟩

lemma div-is-floor-divide-rat: n div y = ⌊rat-of-int n / rat-of-int y⌋
⟨proof⟩

lemma divide-less-floor1: n / y < of-int (floor (n / y)) + 1
⟨proof⟩

context linordered-idom
begin

lemma sgn-int-pow[simp]: sgn ((x :: 'a) ^ p) = sgn x ^ p
⟨proof⟩

end
lemma sgn-int-pow[simp]: assumes x: (x :: 'a) ≠ 0 shows sgn x ^ p = (if even p then 1 else sgn x)
⟨proof⟩

lemma compare-pow-le-iff: p > 0 ⟹ (x :: 'a) ≥ 0 ⟹ y ≥ 0 ⟹ (x ^ p ≤ y ^ p) = (x ≤ y)
⟨proof⟩

lemma compare-pow-less-iff: p > 0 ⟹ (x :: 'a) ≥ 0 ⟹ y ≥ 0 ⟹ (x ^ p < y ^ p) = (x < y)
⟨proof⟩
end

lemma quotient-of-int[simp]: quotient-of (of-int i) = (i,1)
⟨proof⟩

lemma quotient-of-nat[simp]: quotient-of (of-nat i) = (int i,1)
⟨proof⟩

lemma square-lesseq-square: ⋀ x y. 0 ≤ (x :: 'a :: linordered-field) ⟹ 0 ≤ y ⟹ (x * x ≤ y * y) = (x ≤ y)
⟨proof⟩

lemma square-less-square: ⋀ x y. 0 ≤ (x :: 'a :: linordered-field) ⟹ 0 ≤ y ⟹ (x * x < y * y) = (x < y)
⟨proof⟩

lemma sqrt-sqrt[simp]: x ≥ 0 ⟹ sqrt x * sqrt x = x
⟨proof⟩

lemma abs-lesseq-square: abs (x :: real) ≤ abs y ⟷ x * x ≤ y * y
⟨proof⟩
end

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

theory NthRoot-Impl
imports
  Sqrt-Babylonian-Auxiliary
  ../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.
2.1 Logarithm

For computing the \( p \)-th root of a number \( n \), we must choose a starting value in the iteration. Here, we use \((2'::'nat)[of-int\ [\log 2\ n] / p]\). Of course, this requires an algorithm to compute logarithms. Here, we just multiply with the base, until we exceed the argument.

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

\[
\begin{align*}
\text{partial-function} & \quad (\text{tailrec}) \log-ceil-impl :: \text{nat} \Rightarrow \text{int} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \text{ where} \\
& \quad [\text{code}]: \log-ceil-impl \ b \ x \ \text{prod} \ \text{sum} = \begin{cases} 
\text{if} \ \text{prod} \geq x \ \text{then} \ \text{sum} \ \text{else} \ \log-ceil-impl \ b \ x \\
(\text{prod} \ast b) \ (\text{sum} + 1)) 
\end{cases} \\
\text{definition} & \quad \log-ceil :: \text{nat} \Rightarrow \text{int} \Rightarrow \text{nat} \text{ where} \\
& \quad \log-ceil \ b \ x \equiv \begin{cases} 
\text{if} \ b > 1 \ \land \ x \geq 0 \ \text{then} \ \log-ceil-impl \ b \ x \ 1 \ 0 \ \text{else} \ 0 
\end{cases} \\
\text{context} & \quad \text{fixes} \ b :: \text{nat} \\
& \quad \text{assumes} \ b: b > 1 \\
\text{begin} \\
\text{function} & \quad \log-ceil MAIN :: \text{int} \Rightarrow \text{int} \Rightarrow \text{nat} \Rightarrow \text{nat} \text{ where} \\
& \quad \log-ceil MAIN \ x \ \text{prod} \ \text{sum} = \begin{cases} 
\text{if} \ \text{prod} > 0 \ \text{then} \ (\text{if} \ \text{prod} \geq x \ \text{then} \ \text{sum} \ \text{else} \ \log-ceil MAIN \ x \ (\text{prod} \ast b) \ (\text{sum} + 1)) \ \text{else} \ 0 
\end{cases} \\
\text{termination} & \quad \langle \text{proof} \rangle \\
\text{lemma} & \quad \log-ceil-impl: \ \text{prod} > 0 \ \Longrightarrow \ \log-ceil-impl \ b \ x \ \text{prod} \ \text{sum} = \log-ceil MAIN \ x \ \text{prod} \ \text{sum} \\
\text{\langle \text{proof} \rangle} \\
\text{end} \\
\text{lemma} & \quad \log-ceil [\text{simp}]: \ \text{assumes} \ b: b > 0 \ \text{and} \ x: x > 0 \\
& \quad \text{shows} \ \log-ceil \ b \ x = \lfloor \log b \ x \rfloor \\
\text{\langle \text{proof} \rangle} \\

2.2 Computing the \( p \)-th root of an integer number

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

\[
\begin{align*}
\text{definition} & \quad \text{start-value} :: \text{int} \Rightarrow \text{nat} \Rightarrow \text{int} \text{ where} \\
& \quad \text{start-value} \ n \ p = 2 ^ \ (\text{nat} [\text{of-int} \ (\log-ceil \ 2 \ n) / \text{rat-of-nat} \ p]) \\
\text{lemma} & \quad \text{start-value MAIN}: \ \text{assumes} \ x: x \geq 0 \ \text{and} \ p: p > 0 \\
\end{align*}
\]
shows \( x \leq (\text{start-value } x \ p) \ p \land \text{start-value } x \ p \geq 0 \)

(proof)

lemma start-value: assumes \( x \geq 0 \) and \( p : p > 0 \) shows \( x \leq (\text{start-value } x \ p) \ p \ p \ \text{start-value } x \ p \geq 0 \)

(proof)

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

locale fixed-root =

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

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

begin

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

\[ \text{root-newton-int-main } x \ n = \begin{cases} (0, \text{False}) & \text{if } (x < 0 \lor n < 0) \\
\text{root-newton-int-main } ((n \text{ div } (x \ p m) + x \ p y m) \text{ div } (\text{int } p)) \ n & \text{else}
\end{cases} \]

(proof)

end

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

partial-function (tailrec) root-int-main’ :: \( \text{nat} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{int} \times \text{bool} \) where

\[ \text{root-int-main’ } pm \ pm ipm ip x n = \begin{cases} (\text{let } x pm = x p m; \ x p = x pm * x \text{ in if } \ x p \leq n \text{ then } (x, \ x p n) \\
\text{else root-int-main’ } pm \ p m (\text{int } p m) (\text{int } p) (\text{start-value } n \ p) \ n & \text{else}
\end{cases} \]

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

definition root-int-main :: \( \text{nat} \rightarrow \text{int} \rightarrow \text{int} \times \text{bool} \) where

\[ \text{root-int-main } p \ n \equiv \begin{cases} (1, n = 1) & \text{if } p = 0 \\
\text{let } pm = p - 1 \\
in \text{root-int-main’ } pm (\text{int } pm) (\text{int } p) (\text{start-value } n \ p) \ n & \text{else}
\end{cases} \]

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

definition root-int :: \( \text{nat} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{list} \) where

\[ \text{root-int } p x \equiv \begin{cases} [\text{let } e = \text{even } p; \ s = \text{sgn } x; \ x’ = \text{abs } x \\
in \text{if } x < 0 \land e \text{ then } [] \text{ else case root-int-main } p \ x’ \text{ of } (y, \text{True}) \Rightarrow \text{if } e \text{ then } [y, -y] \text{ else case } [s * y] \text{ of } - \Rightarrow []
\end{cases} \]

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

context fixed-root
lemma iteration-mono-eq: assumes \( x^n \in \mathbb{P} \)
shows \( \left( \frac{n \div x^m + x \times \text{int } m}{\text{int } p} \right) = x \)

proof

lemma p0: \( p \neq 0 \)
proof

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

lemma iteration-mono-less: assumes \( x \geq 0 \) and \( n \geq 0 \) and \( x^n > (n :: \text{int}) \)
shows \( \left( \frac{n \div x^m + x \times \text{int } m}{\text{int } p} \right) < x \)
proof

lemma iteration-mono-lesseq: assumes \( x \geq 0 \) and \( n \geq 0 \) and \( x^n > (n :: \text{int}) \)
shows \( \left( \frac{n \div x^m + x \times \text{int } m}{\text{int } p} \right) \leq x \)
proof

termination
proof

We next prove that \textit{root-int-main'} is a correct implementation of \textit{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' \( \equiv \) \textit{root-int-main'} \textit{pm} \( \text{int } m \) \( \text{int } p \)

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

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

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

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

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

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

lemma root-newton-int-main-upper:
\( y \ ^{p} \leq n \implies y \geq 0 \implies n \geq 0 \implies \text{root-newton-int-main} y \ n = (x, b) \implies n < (x + 1) \ ^{p} \)
\( (\text{proof}) \)

lemma root-main\prime-upper:
\( x \ ^{p} \geq n \implies x \geq 0 \implies n \geq 0 \implies \text{root-main}' \ x \ n = (y, b) \implies n < (y + 1) \ ^{p} \)
\( (\text{proof}) \)

end

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

lemma root-int-main-all: assumes \( n: n \geq 0 \)
\( \text{and rm: root-int-main} \ p \ n = (y, b) \)
\( \text{shows y} \geq 0 \land b = (y \ ^{p} = n) \land (p > 0 \implies y \ ^{p} \leq n \land n < (y + 1) \ ^{p}) \)
\( \land (p > 0 \implies x \geq 0 \implies x \ ^{p} = n \implies y = x \land b) \)
\( (\text{proof}) \)

lemma root-int-main: assumes \( n: n \geq 0 \)
\( \text{and rm: root-int-main} \ p \ n = (y, b) \)
\( \text{shows y} \geq 0 \land b = (y \ ^{p} = n) \land (p > 0 \implies y \ ^{p} \leq n \land p > 0 \implies n < (y + 1) \ ^{p}) \)
\( \land (p > 0 \implies x \geq 0 \implies x \ ^{p} = n \implies y = x \land b) \)
\( (\text{proof}) \)

lemma root-int[simp]: assumes \( p: p \neq 0 \lor x \neq 1 \)
\( \text{shows set (root-int p x) = \{y . y \ ^{p} = x\}} \)
\( (\text{proof}) \)

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

2.3 Floor and ceiling of roots

Using the bounds for 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 root-int-floor-pos p x = (if p = 0 then 0 else fst (root-int-main p x))
definition root-int-ceiling-pos p x = (if p = 0 then 0 else case root-int-main p x of (y, b) => if b then y else y + 1))

lemma root-int-floor-pos-lower: assumes \( p0: p \neq 0 \) and \( x: x \geq 0 \)
shows root-int-floor-pos p x ∨ p ≤ x
⟨proof⟩

lemma root-int-floor-pos-pos: assumes x: x ≥ 0
  shows root-int-floor-pos p x ≥ 0
⟨proof⟩

lemma root-int-floor-pos-upper: assumes p0: p ≠ 0 and x: x ≥ 0
  shows (root-int-floor-pos p x + 1) ∨ p > x
⟨proof⟩

lemma root-int-floor-pos: assumes x: x ≥ 0
  shows root-int-floor-pos p x = floor (root p (real x))
⟨proof⟩

lemma root-int-ceiling-pos: assumes x: x ≥ 0
  shows root-int-ceiling-pos p x = ceiling (root p (real x))
⟨proof⟩

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 (real x))
⟨proof⟩
lemma root-int-ceiling[simp]: root-int-ceiling p x = ceiling (root p (real x))
⟨proof⟩

2.4 Downgrading algorithms to the naturals
definition root-nat-floor :: nat ⇒ nat ⇒ int where
  root-nat-floor p x = root-int-floor-pos p (int x)
definition root-nat-ceiling :: nat ⇒ nat ⇒ int where
  root-nat-ceiling p x = root-int-ceiling-pos p (int x)
definition root-nat :: nat ⇒ nat ⇒ nat list where
  root-nat p x = map nat (take 1 (root-int p x))
lemma root-nat-floor[simp]: root-nat-floor p x = floor (root p (real x))
  ⟨proof⟩
lemma root-nat-floor-lower: assumes p0: p ≠ 0
  shows root-nat-floor p x ∨ p ≤ x
  ⟨proof⟩
lemma root-nat-floor-upper: assumes p0: p ≠ 0
  shows (root-nat-floor p x + 1) `p > x
⟨proof⟩

lemma root-nat-ceiling [simp]: root-nat-ceiling p x = ceiling (root p x)
⟨proof⟩

lemma root-nat: assumes p0: p ≠ 0 ∨ x ≠ 1
  shows set (root-nat p x) = { y. y `p = x}
⟨proof⟩

2.5 Upgrading algorithms to the rationals

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

definition 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⟩

lemma root-rat-floor [simp]: root-rat-floor p x = floor (root p (of-rat x))
⟨proof⟩

lemma root-rat-ceiling [simp]: root-rat-ceiling p x = ceiling (root p (of-rat x))
⟨proof⟩

lemma root-rat[simp]: assumes p: p ≠ 0 ∨ x ≠ 1
  shows set (root-rat p x) = { y. y `p = x}
⟨proof⟩

end

theory Sqrt-Babylonian
imports
3 Executable algorithms for square roots

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

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

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 $a = \sqrt{4} = 2, b = -\sqrt{9} = -3.$

3.1 The Babylonian method

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

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

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

3.2 The Babylonian method using integer division

First, the algorithm is developed for the non-negative integers. Here, the division operation $\frac{x}{y}$ is replaced by $x \text{ div } y = \lfloor \text{of-int } x \text{ / of-int y} \rfloor$. 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 $NthRoot-Impl$, 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 $sqrt-int-main' : : \mathit{int} \Rightarrow \mathit{int} \Rightarrow \mathit{int} \times \mathit{bool}$

[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 $sqrt-int-main' ((n \div x + x) \div 2) n)\n
\langle proof \rangle$

definition $sqrt-int-main : : \mathit{int} \Rightarrow \mathit{int} \times \mathit{bool}$

[simp]: $sqrt-int-main x = root-int-main 2 x$

lemma $sqrt-int-main-code[code]: sqrt-int-main x = sqrt-int-main' (start-value x 2) x$

\langle proof \rangle$

definition $sqrt-int : : \mathit{int} \Rightarrow \mathit{int} \mathit{list}$

$sqrt-int x = root-int 2 x$

lemma $sqrt-int-code[code]: sqrt-int x = (if x < 0 then [] \ else case sqrt-int-main x of (y, True) \Rightarrow if y = 0 then [0] \ else [y, -y] \ \Rightarrow [])\n
\langle proof \rangle$

lemma $sqrt-int[simp]: set (sqrt-int x) = \{ y . y * y = x \}$

\langle proof \rangle$

lemma $sqrt-int-pos: \textbf{assumes res: sqrt-int x = Cons s ms}$

shows $s \geq 0$

\langle proof \rangle$

definition $[\text{simp}]: sqrt-int-floor-pos x = root-int-floor-pos 2 x$

lemma $sqrt-int-floor-pos-code[code]: sqrt-int-floor-pos x = \mathit{fst} (sqrt-int-main x)$

\langle proof \rangle$

lemma $sqrt-int-floor-pos: \textbf{assumes x: x \geq 0}$

shows $sqrt-int-floor-pos x = [ sqrt (real x) ]$
\begin{proof}
\end{proof}

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

\textbf{lemma} \(\text{sqrt-int-ceiling-pos-code [code]}: \text{sqrt-int-ceiling-pos } x = (\text{case } \text{sqrt-int-main } x \text{ of } (\text{y, b}) \Rightarrow \text{if } b \text{ then } \text{y else } \text{y + 1})\)
\begin{proof}
\end{proof}

\textbf{lemma} \(\text{sqrt-int-ceiling-pos: assumes} x: x \geq 0\)
\begin{proof} shows \(\text{sqrt-int-ceiling-pos } x = \lceil \text{sqrt (real } x \rceil \rceil \)
\end{proof}

\textbf{definition} \(\text{sqrt-int-floor } x = \text{root-int-floor } 2 \times x\)

\textbf{lemma} \(\text{sqrt-int-floor-code [code]}: \text{sqrt-int-floor } x = (\text{if } x \geq 0 \text{ then } \text{sqrt-int-floor-pos } x \text{ else } -\text{sqrt-int-ceiling-pos } (-\text{} x))\)
\begin{proof}
\end{proof}

\textbf{lemma} \(\text{sqrt-int-floor[simp]}: \text{sqrt-int-floor } x = \lfloor \text{sqrt (real } x \rfloor \lfloor \)
\begin{proof}
\end{proof}

\textbf{definition} \(\text{sqrt-int-ceiling } x = \text{root-int-ceiling } 2 \times x\)

\textbf{lemma} \(\text{sqrt-int-ceiling-code [code]}: \text{sqrt-int-ceiling } x = (\text{if } x \geq 0 \text{ then } \text{sqrt-int-ceiling-pos } x \text{ else } -\text{sqrt-int-floor-pos } (-\text{} x))\)
\begin{proof}
\end{proof}

\textbf{lemma} \(\text{sqrt-int-ceiling[simp]}: \text{sqrt-int-ceiling } x = \lceil \text{sqrt (real } x \rceil \rceil \)
\begin{proof}
\end{proof}

\section{3.3 Square roots for the naturals}

\textbf{definition} \(\text{sqrt-nat :: nat } \Rightarrow \text{nat list}\)
\begin{proof}
\end{proof}

\textbf{where} \(\text{sqrt-nat } x = \text{root-nat } 2 \times x\)

\textbf{lemma} \(\text{sqrt-nat-code [code]}: \text{sqrt-nat } x \equiv \text{map nat } (\text{take 1 } (\text{sqrt-int (int } x)))\)
\begin{proof}
\end{proof}

\textbf{lemma} \(\text{sqrt-nat[simp]}: \text{set (sqrt-nat } x) = \{ y. y * y = x \}\)
\begin{proof}
\end{proof}

\textbf{definition} \(\text{sqrt-nat-floor :: nat } \Rightarrow \text{int where}\)
\begin{proof}
\end{proof}

\textbf{where} \(\text{sqrt-nat-floor } x = \text{root-nat-floor } 2 \times x\)

\textbf{lemma} \(\text{sqrt-nat-floor-code [code]}: \text{sqrt-nat-floor } x = \text{sqrt-int-floor-pos (int } x)\)
\begin{proof}
\end{proof}

\textbf{lemma} \(\text{sqrt-nat-floor[simp]}: \text{sqrt-nat-floor } x = \lfloor \text{sqrt (real } x \rfloor \lfloor \)
\begin{proof}
\end{proof}
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)
⟨proof⟩

lemma sqrt-nat-ceiling[simp]: sqrt-nat-ceiling x = ⌈ sqrt (real x) ⌉
⟨proof⟩

3.4 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⟩

lemma sqrt-rat[simp]: set (sqrt-rat x) = { y. y * y = x}
⟨proof⟩

lemma sqrt-rat-pos: assumes sqrt: sqrt-rat x = Cons s ms
  shows s ≥ 0
⟨proof⟩

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)
⟨proof⟩

lemma sqrt-rat-floor[simp]: sqrt-rat-floor x = ⌊ sqrt (of-rat x) ⌋
⟨proof⟩

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))
⟨proof⟩

lemma sqrt-rat-ceiling: sqrt-rat-ceiling x = ⌈ sqrt (of-rat x) ⌉
⟨proof⟩

lemma sqrt-rat-of-int: assumes x: x * x = rat-of-int i
  shows 3 j :: int. j * j = i
3.5 Approximating square roots

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

We first provide the executable version without guard \((\theta::'a) < x\) as partial function, and afterwards prove termination and soundness for a similar algorithm that is defined within the upcoming locale.

\textbf{partial-function} \((\text{tailrec})\) \texttt{sqrt-approx-main-impl} :: 'a :: linordered-field \(\Rightarrow\) 'a \(\Rightarrow\) 'a \(\Rightarrow\) where

\[ \begin{align*}
\texttt{sqrt-approx-main-impl} &\::\:\varepsilon \::\:n \::\:x = (\text{if } x \ast x - n < \varepsilon \text{ then } x \text{ else } \texttt{sqrt-approx-main-impl} \\
&\quad \varepsilon \::\:n \::\:((n / x + x) / 2))
\end{align*} \]

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

\textbf{locale} \texttt{sqrt-approximation} =

\texttt{fixes} \varepsilon :: 'a :: \{linordered-field,floor-ceiling\}

\texttt{and} \quad n :: 'a

\texttt{assumes} \quad \varepsilon : \varepsilon > 0

\texttt{and} \quad n : n > 0

\texttt{begin}

\textbf{function} \texttt{sqrt-approx-main} :: 'a \(\Rightarrow\) 'a \(\Rightarrow\) where

\[ \texttt{sqrt-approx-main} \cdot x = (\text{if } x > 0 \text{ then } (\text{if } x \ast x - n < \varepsilon \text{ then } x \text{ else } \texttt{sqrt-approx-main} \\
\quad ((n / x + x) / 2)) \text{ else } 0) \]

\(\langle\text{proof}\rangle\)

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 \textit{rat} then there is no square root of 2. Therefore, the error-rate \( \frac{x}{\sqrt{n}} - 1 \) is not expressible. Instead we use the expression \( \frac{x^2}{n} - 1 \) as error-rate which does not require any square-root operation.

\textbf{termination} \(\langle\text{proof}\rangle\)

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

\textbf{lemma} \texttt{sqrt-approx-main-impl} \(x > 0 \Rightarrow \texttt{sqrt-approx-main-impl} \varepsilon \cdot n \cdot x = \texttt{sqrt-approx-main} \cdot x \)

\(\langle\text{proof}\rangle\)

Also soundness is not complicated.
lemma sqrt-approx-main-sound: assumes \( x: x > 0 \) and \( xx: x \cdot x > n \)
shows \( \sqrt{\text{approx-main}} \cdot \sqrt{\text{approx-main}} > n \land \sqrt{\text{approx-main}} \cdot \sqrt{\text{approx-main}} - n < \varepsilon \)
(proof)
end

It remains to assemble everything into one algorithm.
definition sqrt-approx :: ′a :: {linordered-field,floor-ceiling} ⇒ ′a where
sqrt-approx \varepsilon x ≡ if \varepsilon > 0 then (if \( x = 0 \) then 0 else let xpos = abs \( x \) in sqrt-approx-main-impl \varepsilon xpos (xpos + 1)) else 0

lemma sqrt-approx: assumes \( \varepsilon: \varepsilon > 0 \)
shows \( |\sqrt{\text{approx}} \cdot \sqrt{\text{approx}} - x| < \varepsilon \)
(proof)

3.6 Some tests
Testing executability and show that \( \sqrt{2} \) is irrational
lemma ¬ (\( \exists \ i :: \text{rat.} \ i \cdot i = 2 \))
(proof)

Testing speed
lemma ¬ (\( \exists \ i :: \text{int.} \ i \cdot i = 1234567890123456789012345678901234567890 \))
(proof)

The following test
value let \( \varepsilon = 1 / 100000000 :: \text{rat;} \ s = \sqrt{\text{approx}} \cdot \varepsilon \cdot 2 \) in \( (s, s \cdot s - 2, |s \cdot s - 2| < \varepsilon) \)
results in \((1.4142135623731116, 4.738200762148612e-14, True)\).
end

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

References