

Computing N-th Roots using the Babylonian Method*

René Thiemann

February 23, 2021

Abstract

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

Contents

1	Auxiliary lemmas which might be moved into the Isabelle distribution.	2
2	A Fast Logarithm Algorithm	3
3	Executable algorithms for p-th roots	5
3.1	Logarithm	5
3.2	Computing the p -th root of an integer number	5
3.3	Floor and ceiling of roots	8
3.4	Downgrading algorithms to the naturals	9
3.5	Upgrading algorithms to the rationals	10
4	Executable algorithms for square roots	10
4.1	The Babylonian method	11
4.2	The Babylonian method using integer division	11
4.3	Square roots for the naturals	13
4.4	Square roots for the rationals	14
4.5	Approximating square roots	14
4.6	Some tests	16

*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*: $(n :: \text{int}) \text{ div } x * x = n - n \text{ mod } x$
<proof>

lemma *div-is-floor-divide-rat*: $n \text{ div } y = \lfloor \text{rat-of-int } n / \text{rat-of-int } y \rfloor$
<proof>

lemma *div-is-floor-divide-real*: $n \text{ div } y = \lfloor \text{real-of-int } n / \text{of-int } y \rfloor$
<proof>

lemma *floor-div-pos-int*:

fixes $r :: 'a :: \text{floor-ceiling}$

assumes $n: n > 0$

shows $\lfloor r / \text{of-int } n \rfloor = \lfloor r \rfloor \text{ div } n$ (**is** $?l = ?r$)

<proof>

lemma *floor-div-neg-int*:

fixes $r :: 'a :: \text{floor-ceiling}$

assumes $n: n < 0$

shows $\lfloor r / \text{of-int } n \rfloor = \lceil r \rceil \text{ div } n$

<proof>

lemma *divide-less-floor1*: $n / y < \text{of-int } (\text{floor } (n / y)) + 1$
<proof>

context *linordered-idom*

begin

lemma *sgn-int-pow-if* [*simp*]:

$\text{sgn } x \wedge^p = (\text{if even } p \text{ then } 1 \text{ else } \text{sgn } x)$ **if** $x \neq 0$

<proof>

lemma *compare-pow-le-iff*: $p > 0 \implies (x :: 'a) \geq 0 \implies y \geq 0 \implies (x \wedge^p \leq y \wedge^p) = (x \leq y)$

<proof>

lemma *compare-pow-less-iff*: $p > 0 \implies (x :: 'a) \geq 0 \implies y \geq 0 \implies (x \wedge^p < y \wedge^p) = (x < y)$

<proof>

end

lemma *quotient-of-int[simp]*: $\text{quotient-of } (\text{of-int } i) = (i, 1)$
<proof>

lemma *quotient-of-nat[simp]*: $\text{quotient-of } (\text{of-nat } i) = (\text{int } i, 1)$
<proof>

lemma *square-lesseq-square*: $\bigwedge x y. 0 \leq (x :: 'a :: \text{linordered-field}) \implies 0 \leq y \implies (x * x \leq y * y) = (x \leq y)$
<proof>

lemma *square-less-square*: $\bigwedge x y. 0 \leq (x :: 'a :: \text{linordered-field}) \implies 0 \leq y \implies (x * x < y * y) = (x < y)$
<proof>

lemma *sqr-sqr[simp]*: $x \geq 0 \implies \text{sqr } x * \text{sqr } x = x$
<proof>

lemma *abs-lesseq-square*: $\text{abs } (x :: \text{real}) \leq \text{abs } y \longleftrightarrow x * x \leq y * y$
<proof>

end

2 A Fast Logarithm Algorithm

theory *Log-Impl*

imports

Sqr-Babylonian-Auxiliary

begin

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

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

typedef *proper-base* = $\{x :: \text{int}. x \geq 2\}$ *<proof>*

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

lift-definition *get-base* :: $\text{proper-base} \Rightarrow \text{int}$ **is** $\lambda x. x$ *<proof>*

lift-definition *square-base* :: $\text{proper-base} \Rightarrow \text{proper-base}$ **is** $\lambda x. x * x$
<proof>

lift-definition *into-base* :: $\text{int} \Rightarrow \text{proper-base}$ **is** $\lambda x. \text{if } x \geq 2 \text{ then } x \text{ else } 2$ *<proof>*

lemma *square-base*: $\text{get-base } (\text{square-base } b) = \text{get-base } b * \text{get-base } b$
<proof>

lemma *get-base-2*: *get-base* $b \geq 2$

<proof>

lemma *b-less-square-base-b*: *get-base* $b < \text{get-base } (\text{square-base } b)$

<proof>

lemma *b-less-div-base-b*: **assumes** $xb: \neg x < \text{get-base } b$

shows $x \text{ div } \text{get-base } b < x$

<proof>

We now state the main algorithm.

function *log-main* :: *proper-base* \Rightarrow *int* \Rightarrow *nat* \times *int* **where**

log-main b $x = (\text{if } x < \text{get-base } b \text{ then } (0,1) \text{ else}$

case log-main (*square-base* b) x of

$(z, bz) \Rightarrow$

let $l = 2 * z; bz1 = bz * \text{get-base } b$

in $\text{if } x < bz1 \text{ then } (l, bz) \text{ else } (\text{Suc } l, bz1)$

<proof>

termination *<proof>*

lemma *log-main*: $x > 0 \Longrightarrow \text{log-main } b$ $x = (y, by) \Longrightarrow by = (\text{get-base } b)^{\wedge} y \wedge$

$(\text{get-base } b)^{\wedge} y \leq x \wedge x < (\text{get-base } b)^{\wedge} (\text{Suc } y)$

<proof>

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

definition *log-floor* :: *int* \Rightarrow *int* \Rightarrow *nat* **where**

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

definition *log-ceiling* :: *int* \Rightarrow *int* \Rightarrow *nat* **where**

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

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

lemma *log-floor-sound*: **assumes** $b > 1$ $x > 0$ *log-floor* b $x = y$

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

<proof>

lemma *log-ceiling-sound*: **assumes** $b > 1$ $x > 0$ *log-ceiling* b $x = y$

shows $x \leq b^{\wedge} y$ $y \neq 0 \Longrightarrow b^{\wedge} (y - 1) < x$

<proof>

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

lemma *log-floor[simp]*: **assumes** $b: b > 1$ **and** $x: x > 0$

shows *log-floor* b $x = \lfloor \log b x \rfloor$

<proof>

lemma *log-ceiling[simp]*: **assumes** $b: b > 1$ **and** $x: x > 0$

shows *log-ceiling* b $x = \lceil \log b x \rceil$

<proof>

end

3 Executable algorithms for p -th roots

theory *NthRoot-Impl*

imports

Log-Impl

Cauchy.CauchysMeanTheorem

begin

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

3.1 Logarithm

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

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

3.2 Computing the p -th root of an integer number

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

definition *start-value* :: *int* \Rightarrow *nat* \Rightarrow *int* **where**

start-value $n \ p = 2^{\wedge}(\text{nat } \lceil \text{of-nat } (\text{log-ceiling } 2 \ n) / \text{rat-of-nat } p \rceil)$

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

shows $x \leq (\text{start-value } x \ p)^{\wedge} p \wedge \text{start-value } x \ p \geq 0$

<proof>

lemma *start-value*: **assumes** $x: x \geq 0$ **and** $p: p > 0$ **shows** $x \leq (\text{start-value } x \ p)^{\wedge} p$ *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 $(/)$ is replaced by (div) . We are

proving several things within a locale which ensures that $p > 0$, and where $pm = p - 1$.

```

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

```

```

function root-newton-int-main ::  $int \Rightarrow int \Rightarrow int \times bool$  where
  root-newton-int-main  $x\ n = (if\ (x < 0 \vee n < 0)\ then\ (0, False)\ else\ (if\ x \wedge^p \leq n\ then\ (x, x \wedge^p = n)\ else\ root-newton-int-main\ ((n\ div\ (x \wedge^{pm}) + x * int\ pm)\ div\ (int\ p))\ n))$ 
   $\langle proof \rangle$ 
end

```

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

```

partial-function (tailrec) root-int-main' ::  $nat \Rightarrow int \Rightarrow int \Rightarrow int \Rightarrow int \Rightarrow int \Rightarrow int \times bool$  where
  [code]: root-int-main'  $pm\ ipm\ ip\ x\ n = (let\ xpm = x \wedge^{pm};\ xp = xpm * x\ in\ if\ xp \leq n\ then\ (x, xp = n)\ else\ root-int-main'\ pm\ ipm\ ip\ ((n\ div\ xpm + x * ipm)\ div\ ip)\ n)$ 

```

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

```

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

```

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

```

definition root-int ::  $nat \Rightarrow int \Rightarrow int\ list$  where
  root-int  $p\ x \equiv if\ p = 0\ then\ []\ else$ 
   $if\ x = 0\ then\ [0]\ else$ 
   $let\ e = even\ p; s = sgn\ x; x' = abs\ x$ 
   $in\ if\ x < 0 \wedge e\ then\ []\ else\ case\ root-int-main\ p\ x'\ of\ (y, True) \Rightarrow if\ e\ then$ 
   $[y, -y]\ else\ [s * y] \mid - \Rightarrow []$ 

```

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

```

context fixed-root

```

```

begin

```

```

lemma iteration-mono-eq: assumes  $xn: x \wedge^p = (n :: int)$ 
  shows  $(n\ div\ x \wedge^{pm} + x * int\ pm)\ div\ int\ p = x$ 
   $\langle proof \rangle$ 

```

```

lemma p0:  $p \neq 0$   $\langle proof \rangle$ 

```

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

lemma *iteration-mono-less*: **assumes** $x: x \geq 0$
and $n: n \geq 0$
and $xn: x \hat{\ } p > (n :: int)$
shows $(n \text{ div } x \hat{\ } pm + x * int \text{ pm}) \text{ div } int \text{ p} < x$
 $\langle proof \rangle$

lemma *iteration-mono-lesseq*: **assumes** $x: x \geq 0$ **and** $n: n \geq 0$ **and** $xn: x \hat{\ } p \geq (n :: int)$
shows $(n \text{ div } x \hat{\ } pm + x * int \text{ pm}) \text{ div } int \text{ p} \leq x$
 $\langle proof \rangle$

termination
 $\langle proof \rangle$

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

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

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

lemma *root-main'-newton-pos*: $x \geq 0 \implies n \geq 0 \implies$
 $root\text{-}main' \ x \ n = root\text{-}newton\text{-}int\text{-}main \ x \ n \wedge (root\text{-}main' \ x \ n = (y,b) \implies y \geq 0$
 $\wedge y \hat{\ } p \leq n \wedge b = (y \hat{\ } p = n))$
 $\langle proof \rangle$

lemma *root-main'*: $x \geq 0 \implies n \geq 0 \implies root\text{-}main' \ x \ n = root\text{-}newton\text{-}int\text{-}main \ x \ n$
 $\langle proof \rangle$

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

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

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

lemma *root-main'-lower*: $x \geq 0 \implies n \geq 0 \implies root\text{-}main' \ x \ n = (y,b) \implies y \hat{\ } p \leq n$
 $\langle proof \rangle$

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

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

<proof>

lemma *root-main'-upper:*

$x \hat{\ }^p \geq n \implies x \geq 0 \implies n \geq 0 \implies \text{root-main}' x n = (y, b) \implies n < (y + 1) \hat{\ }^p$

<proof>

end

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

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

and *rm:* *root-int-main* $p n = (y, b)$

shows $y \geq 0 \wedge b = (y \hat{\ }^p = n) \wedge (p > 0 \implies y \hat{\ }^p \leq n \wedge n < (y + 1) \hat{\ }^p)$

$\wedge (p > 0 \implies x \geq 0 \implies x \hat{\ }^p = n \implies y = x \wedge b)$

<proof>

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

and *rm:* *root-int-main* $p n = (y, b)$

shows $y \geq 0 \wedge b = (y \hat{\ }^p = n) \wedge p > 0 \implies y \hat{\ }^p \leq n \wedge p > 0 \implies n < (y + 1) \hat{\ }^p$

$p > 0 \implies x \geq 0 \implies x \hat{\ }^p = n \implies y = x \wedge b$

<proof>

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

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

<proof>

lemma *root-int-pos:* **assumes** $x: x \geq 0$ **and** *ri:* *root-int* $p x = y \# ys$

shows $y \geq 0$

<proof>

3.3 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* $p x = (\text{if } p = 0 \text{ then } 0 \text{ else } \text{fst } (\text{root-int-main } p x))$

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

lemma *root-int-floor-pos-lower:* **assumes** $p0: p \neq 0$ **and** $x: x \geq 0$

shows *root-int-floor-pos* $p x \hat{\ }^p \leq x$

<proof>

lemma *root-int-floor-pos-pos:* **assumes** $x: x \geq 0$

shows *root-int-floor-pos* $p x \geq 0$

<proof>

lemma *root-int-floor-pos-upper:* **assumes** $p0: p \neq 0$ **and** $x: x \geq 0$

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

<proof>

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

lemma *root-int-ceiling-pos*: **assumes** $x: x \geq 0$
shows $\text{root-int-ceiling-pos } p \ x = \text{ceiling } (\text{root } p \ (\text{of-int } x))$
 $\langle \text{proof} \rangle$

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

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

lemma *root-int-floor[simp]*: $\text{root-int-floor } p \ x = \text{floor } (\text{root } p \ (\text{of-int } x))$
 $\langle \text{proof} \rangle$

lemma *root-int-ceiling[simp]*: $\text{root-int-ceiling } p \ x = \text{ceiling } (\text{root } p \ (\text{of-int } x))$
 $\langle \text{proof} \rangle$

3.4 Downgrading algorithms to the naturals

definition *root-nat-floor* $:: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{int}$ **where**
 $\text{root-nat-floor } p \ x = \text{root-int-floor-pos } p \ (\text{int } x)$

definition *root-nat-ceiling* $:: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{int}$ **where**
 $\text{root-nat-ceiling } p \ x = \text{root-int-ceiling-pos } p \ (\text{int } x)$

definition *root-nat* $:: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat list}$ **where**
 $\text{root-nat } p \ x = \text{map nat } (\text{take } 1 \ (\text{root-int } p \ x))$

lemma *root-nat-floor [simp]*: $\text{root-nat-floor } p \ x = \text{floor } (\text{root } p \ (\text{real } x))$
 $\langle \text{proof} \rangle$

lemma *root-nat-floor-lower*: **assumes** $p0: p \neq 0$
shows $\text{root-nat-floor } p \ x \wedge p \leq x$
 $\langle \text{proof} \rangle$

lemma *root-nat-floor-upper*: **assumes** $p0: p \neq 0$
shows $(\text{root-nat-floor } p \ x + 1) \wedge p > x$
 $\langle \text{proof} \rangle$

lemma *root-nat-ceiling [simp]*: $\text{root-nat-ceiling } p \ x = \text{ceiling } (\text{root } p \ x)$
 $\langle \text{proof} \rangle$

lemma *root-nat*: **assumes** $p0: p \neq 0 \vee x \neq 1$
shows $\text{set } (\text{root-nat } p \ x) = \{ y. y \wedge p = x \}$
 $\langle \text{proof} \rangle$

3.5 Upgrading algorithms to the rationals

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

Sqrt-Babylonian-Auxiliary

NthRoot-Impl

begin

4 Executable algorithms for square roots

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

For integers / naturals / rationals precise algorithms are given, i.e., here

`sqrt x` delivers a list of all integers / naturals / rationals y where $y^2 = x$. To this end, the Babylonian method has been adapted by using integer-divisions.

In addition to the precise algorithms, we also provide approximation algorithms. One works for arbitrary linear ordered fields, where some number y is computed such that $|y^2 - x| < \varepsilon$. Moreover, for the naturals, integers, and rationals we provide algorithms to compute $\lfloor \text{sqrt } x \rfloor$ and $\lceil \text{sqrt } x \rceil$ 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 $(a_1x_1 + \dots a_nx_n)^2 = p$ had to be solved over the integers, where p is a concrete polynomial. For example, for the equation $(ax + by)^2 = 4x^2 - 12xy + 9y^2$ one easily figures out that $a^2 = 4, b^2 = 9$, and $ab = -6$, which results in a possible solution $a = \sqrt{4} = 2, b = -\sqrt{9} = -3$.

4.1 The Babylonian method

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

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

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

4.2 The Babylonian method using integer division

First, the algorithm is developed for the non-negative integers. Here, the division operation $\frac{x}{y}$ is replaced by $x \text{ div } y = \lfloor \text{of-int } x / \text{of-int } y \rfloor$. 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, \dots are monotone decreasing, in the main algorithm we abort as soon as $x_i^2 \leq n$.

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

definition $\text{sqrt-int-main}' :: \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \times \text{bool}$ **where**
 $[\text{simp}] : \text{sqrt-int-main}' x n = \text{root-int-main}' 1 1 2 x n$

lemma $\text{sqrt-int-main}'\text{-code}[\text{code}] : \text{sqrt-int-main}' x n = (\text{let } x2 = x * x \text{ in if } x2 \leq n \text{ then } (x, x2 = n) \text{ else } \text{sqrt-int-main}' ((n \text{ div } x + x) \text{ div } 2) n)$
 $\langle \text{proof} \rangle$

definition $\text{sqrt-int-main} :: \text{int} \Rightarrow \text{int} \times \text{bool}$ **where**
 $[\text{simp}] : \text{sqrt-int-main} x = \text{root-int-main} 2 x$

lemma $\text{sqrt-int-main-code}[\text{code}] : \text{sqrt-int-main} x = \text{sqrt-int-main}' (\text{start-value } x 2) x$
 $\langle \text{proof} \rangle$

definition $\text{sqrt-int} :: \text{int} \Rightarrow \text{int list}$ **where**
 $\text{sqrt-int} x = \text{root-int} 2 x$

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

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

lemma sqrt-int-pos : **assumes** $\text{res} : \text{sqrt-int} x = \text{Cons } s \text{ ms}$
shows $s \geq 0$
 $\langle \text{proof} \rangle$

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

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

lemma $\text{sqrt-int-floor-pos}$: **assumes** $x : x \geq 0$
shows $\text{sqrt-int-floor-pos} x = \lfloor \text{sqrt } (\text{of-int } x) \rfloor$
 $\langle \text{proof} \rangle$

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

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

lemma $\text{sqrt-int-ceiling-pos}$: **assumes** $x : x \geq 0$
shows $\text{sqrt-int-ceiling-pos} x = \lceil \text{sqrt } (\text{of-int } x) \rceil$
 $\langle \text{proof} \rangle$

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

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

lemma $\text{sqrt-int-floor}[simp]$: $\text{sqrt-int-floor } x = \lfloor \text{sqrt } (\text{of-int } x) \rfloor$
 $\langle \text{proof} \rangle$

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

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

lemma $\text{sqrt-int-ceiling}[simp]$: $\text{sqrt-int-ceiling } x = \lceil \text{sqrt } (\text{of-int } x) \rceil$
 $\langle \text{proof} \rangle$

lemma $\text{sqrt-int-ceiling-bound}$: $0 \leq x \implies x \leq (\text{sqrt-int-ceiling } x)^2$
 $\langle \text{proof} \rangle$

4.3 Square roots for the naturals

definition $\text{sqrt-nat} :: \text{nat} \Rightarrow \text{nat list}$
where $\text{sqrt-nat } x = \text{root-nat } 2 \ x$

lemma $\text{sqrt-nat-code}[code]$: $\text{sqrt-nat } x \equiv \text{map nat } (\text{take } 1 \ (\text{sqrt-int } (\text{int } x)))$
 $\langle \text{proof} \rangle$

lemma $\text{sqrt-nat}[simp]$: $\text{set } (\text{sqrt-nat } x) = \{ y. y * y = x \}$
 $\langle \text{proof} \rangle$

definition $\text{sqrt-nat-floor} :: \text{nat} \Rightarrow \text{int}$ **where**
 $\text{sqrt-nat-floor } x = \text{root-nat-floor } 2 \ x$

lemma $\text{sqrt-nat-floor-code}[code]$: $\text{sqrt-nat-floor } x = \text{sqrt-int-floor-pos } (\text{int } x)$
 $\langle \text{proof} \rangle$

lemma $\text{sqrt-nat-floor}[simp]$: $\text{sqrt-nat-floor } x = \lfloor \text{sqrt } (\text{real } x) \rfloor$
 $\langle \text{proof} \rangle$

definition $\text{sqrt-nat-ceiling} :: \text{nat} \Rightarrow \text{int}$ **where**
 $\text{sqrt-nat-ceiling } x = \text{root-nat-ceiling } 2 \ x$

lemma $\text{sqrt-nat-ceiling-code}[code]$: $\text{sqrt-nat-ceiling } x = \text{sqrt-int-ceiling-pos } (\text{int } x)$
 $\langle \text{proof} \rangle$

lemma $\text{sqrt-nat-ceiling}[simp]$: $\text{sqrt-nat-ceiling } x = \lceil \text{sqrt } (\text{real } x) \rceil$
 $\langle \text{proof} \rangle$

4.4 Square roots for the rationals

definition *sqrt-rat* :: *rat* \Rightarrow *rat list* **where**

$$\text{sqrt-rat } x = \text{root-rat } 2 \ x$$

lemma *sqrt-rat-code*[code]: *sqrt-rat* $x = (\text{case quotient-of } x \text{ of } (z, n) \Rightarrow (\text{case sqrt-int } n \text{ of$

$$\square \Rightarrow \square$$

$$| \text{sn} \# \text{xs} \Rightarrow \text{map } (\lambda \text{sz. of-int } \text{sz} / \text{of-int } \text{sn}) (\text{sqrt-int } z)))$$

<proof>

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

<proof>

lemma *sqrt-rat-pos*: **assumes** *sqrt*: *sqrt-rat* $x = \text{Cons } s \ ms$

shows $s \geq 0$

<proof>

definition *sqrt-rat-floor* :: *rat* \Rightarrow *int* **where**

$$\text{sqrt-rat-floor } x = \text{root-rat-floor } 2 \ x$$

lemma *sqrt-rat-floor-code*[code]: *sqrt-rat-floor* $x = (\text{case quotient-of } x \text{ of } (a, b) \Rightarrow \text{sqrt-int-floor } (a * b) \text{ div } b)$

<proof>

lemma *sqrt-rat-floor[simp]*: *sqrt-rat-floor* $x = \lfloor \text{sqrt } (\text{of-rat } x) \rfloor$

<proof>

definition *sqrt-rat-ceiling* :: *rat* \Rightarrow *int* **where**

$$\text{sqrt-rat-ceiling } x = \text{root-rat-ceiling } 2 \ x$$

lemma *sqrt-rat-ceiling-code*[code]: *sqrt-rat-ceiling* $x = - (\text{sqrt-rat-floor } (-x))$

<proof>

lemma *sqrt-rat-ceiling*: *sqrt-rat-ceiling* $x = \lceil \text{sqrt } (\text{of-rat } x) \rceil$

<proof>

lemma *sqrt-rat-of-int*: **assumes** $x: x * x = \text{rat-of-int } i$

shows $\exists j :: \text{int. } j * j = i$

<proof>

4.5 Approximating square roots

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

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

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

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

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

function *sqrt-approx-main* :: 'a \Rightarrow 'a **where**
sqrt-approx-main x = (if x > 0 then (if x * x - n < ε then x else *sqrt-approx-main* ((n / x + x) / 2)) else 0)
 <proof>

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

termination
 <proof>

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

lemma *sqrt-approx-main-impl*: $x > 0 \implies \text{sqrt-approx-main-impl } \varepsilon$ n x = *sqrt-approx-main* x
 <proof>

Also soundness is not complicated.

lemma *sqrt-approx-main-sound*: **assumes** x: $x > 0$ **and** xx: $x * x > n$
shows *sqrt-approx-main* x * *sqrt-approx-main* x > n \wedge *sqrt-approx-main* x * *sqrt-approx-main* x - n < ε
 <proof>

end

It remains to assemble everything into one algorithm.

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

lemma *sqrt-approx*: **assumes** $\varepsilon: \varepsilon > 0$
shows $|\text{sqrt-approx } \varepsilon x * \text{sqrt-approx } \varepsilon x - |x|| < \varepsilon$
<proof>

4.6 Some tests

Testing executability and show that sqrt 2 is irrational

lemma $\neg (\exists i :: \text{rat. } i * i = 2)$
<proof>

Testing speed

lemma $\neg (\exists i :: \text{int. } i * i = 1234567890123456789012345678901234567890)$
<proof>

The following test

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

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

end

Acknowledgements

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

References

- [1] T. Heath. *A History of Greek Mathematics*, volume 2, pages 323–326. Clarendon Press, 1921.
- [2] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proc. TPHOLs'09*, volume 5674 of *LNCS*, pages 452–468, 2009.