Computing N-th Roots using the Babylonian Method*

René Thiemann

March 19, 2025

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

*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*: $(n :: int)\ div\ x * x = n - n\ mod\ x$
  ⟨*proof*⟩

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

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

lemma *floor-div-pos-int*:
  fixes $r :: 'a :: floor\text{-}ceiling$
  assumes n: $n > 0$
  shows $\lfloor r\ /\ of\text{-}int\ n \rfloor = \lfloor r \rfloor\ div\ n$ (**is** $?l = ?r$)
⟨*proof*⟩

lemma *floor-div-neg-int*:
  fixes $r :: 'a :: floor\text{-}ceiling$
  assumes n: $n < 0$
  shows $\lfloor r\ /\ of\text{-}int\ n \rfloor = \lceil r \rceil\ div\ n$
⟨*proof*⟩

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

context *linordered-idom*
begin

lemma *sgn-int-pow-if* [*simp*]:
  $sgn\ x\ \hat{}\ p = (if\ even\ p\ then\ 1\ else\ sgn\ x)$ **if** $x \neq 0$
  ⟨*proof*⟩

lemma *compare-pow-le-iff*: $p > 0 \Longrightarrow (x :: 'a) \geq 0 \Longrightarrow y \geq 0 \Longrightarrow (x\ \hat{}\ p \leq y\ \hat{}\ p) = (x \leq y)$
  ⟨*proof*⟩

lemma *compare-pow-less-iff*: $p > 0 \Longrightarrow (x :: 'a) \geq 0 \Longrightarrow y \geq 0 \Longrightarrow (x\ \hat{}\ p < y\ \hat{}\ p) = (x < y)$
  ⟨*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*: $\bigwedge$ *x y*. $0 \le (x :: 'a :: linordered\text{-}field) \Longrightarrow 0 \le y \Longrightarrow (x * x \le y * y) = (x \le y)$
⟨*proof*⟩

lemma *square-less-square*: $\bigwedge$ *x y*. $0 \le (x :: 'a :: linordered\text{-}field) \Longrightarrow 0 \le y \Longrightarrow (x * x < y * y) = (x < y)$
⟨*proof*⟩

lemma *sqrt-sqrt*[*simp*]: $x \ge 0 \Longrightarrow sqrt\ x * sqrt\ x = x$
⟨*proof*⟩

lemma *abs-lesseq-square*: $abs\ (x :: real) \le abs\ y \longleftrightarrow x * x \le y * y$
⟨*proof*⟩

end

2 A Fast Logarithm Algorithm

theory *Log-Impl*
imports
Sqrt-Babylonian-Auxiliary
begin

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

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

typedef *proper-base* = $\{x :: int.\ x \ge 2\}$ ⟨*proof*⟩

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

lift-definition *get-base* :: *proper-base* ⇒ *int* **is** λ *x*. *x* ⟨*proof*⟩

lift-definition *square-base* :: *proper-base* ⇒ *proper-base* **is** λ *x*. $x * x$
⟨*proof*⟩

lift-definition *into-base* :: *int* ⇒ *proper-base* **is** λ *x*. *if* $x \ge 2$ *then x else 2* ⟨*proof*⟩

lemma *square-base*: *get-base* (*square-base b*) = *get-base b* ∗ *get-base b*
⟨*proof*⟩

lemma *get-base-2*: *get-base b* $\geq$ *2*
⟨*proof*⟩

lemma *b-less-square-base-b*: *get-base b* $<$ *get-base* (*square-base b*)
⟨*proof*⟩

lemma *b-less-div-base-b*: **assumes** *xb*: $\neg$ *x* $<$ *get-base b*
shows *x div 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 = (*if x* $<$ *get-base b then* (*0*,*1*) *else*
case log-main (*square-base b*) *x of*
(*z*, *bz*) $\Rightarrow$
let l = *2* $*$ *z*; *bz1* = *bz* $*$ *get-base b*
in if x $<$ *bz1 then* (*l*,*bz*) *else* (*Suc l*,*bz1*))
⟨*proof*⟩

termination ⟨*proof*⟩

lemma *log-main*: *x* $>$ *0* $\Longrightarrow$ *log-main b x* = (*y*,*by*) $\Longrightarrow$ *by* = (*get-base b*)^*y* $\wedge$ (*get-base b*)^*y* $\leq$ *x* $\wedge$ *x* $<$ (*get-base b*)^(*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 = *fst* (*log-main* (*into-base b*) *x*)

definition *log-ceiling* :: *int* $\Rightarrow$ *int* $\Rightarrow$ *nat* **where**
log-ceiling b x = (*case log-main* (*into-base b*) *x of*
(*y*,*by*) $\Rightarrow$ *if x* = *by then y else Suc y*)

lemma *log-floor-sound*: **assumes** *b* $>$ *1 x* $>$ *0 log-floor b x* = *y*
shows *b*^*y* $\leq$ *x x* $<$ *b*^(*Suc y*)
⟨*proof*⟩

lemma *log-ceiling-sound*: **assumes** *b* $>$ *1 x* $>$ *0 log-ceiling b x* = *y*
shows *x* $\leq$ *b*^*y y* $\neq$ *0* $\Longrightarrow$ *b*^(*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 $(2::'a)^{nat\ \lceil of\text{-}int\ \lceil log\ 2\ n\rceil\ /\ p\rceil}$.

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 $x \leq (start\text{-}value\ x\ p)^p$, 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* ⇒ *nat* ⇒ *int* **where**
  start-value n p = *2* ^ (*nat* ⌈*of-nat* (*log-ceiling 2 n*) / *rat-of-nat p*⌉)

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*⟩

lemma *start-value*: **assumes** *x*: *x* ≥ *0* **and** *p*: *p* > *0* **shows** *x* ≤ (*start-value x p*) ^ *p start-value x p* ≥ *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* ⇒ *int* ⇒ *int* × *bool* **where**
root-newton-int-main x n = (*if* (*x* < *0* ∨ *n* < *0*) *then* (*0*,*False*) *else* (*if x* ^ *p* ≤ *n then* (*x*, *x* ^ *p* = *n*)
else root-newton-int-main ((*n div* (*x* ^ *pm*) + *x* ∗ *int pm*) *div* (*int p*)) *n*))
⟨*proof*⟩
end

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

partial-function (*tailrec*) *root-int-main′* :: *nat* ⇒ *int* ⇒ *int* ⇒ *int* ⇒ *int* ⇒ *int* × *bool* **where**
[*code*]: *root-int-main′ pm ipm ip x n* = (*let xpm* = *x*^*pm*; *xp* = *xpm* ∗ *x in if xp* ≤ *n then* (*x*, *xp* = *n*)
else root-int-main′ pm ipm ip ((*n div xpm* + *x* ∗ *ipm*) *div ip*) *n*)

In the following algorithm, we start the iteration. It will compute ⌊*root p n*⌋ and a boolean to indicate whether the root is exact.

definition *root-int-main* :: *nat* ⇒ *int* ⇒ *int* × *bool* **where**
root-int-main p n ≡ *if p* = *0 then* (*1*,*n* = *1*) *else*
let pm = *p* − *1*
in root-int-main′ pm (*int pm*) (*int p*) (*start-value n p*) *n*

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

definition *root-int* :: *nat* ⇒ *int* ⇒ *int list* **where**
root-int p x ≡ *if p* = *0 then* [] *else*
if x = *0 then* [*0*] *else*
let e = *even p*; *s* = *sgn x*; *x′* = *abs x*
in if x < *0* ∧ *e then* [] *else case root-int-main p x′ of* (*y*,*True*) ⇒ *if e then* [*y*,−*y*] *else* [*s* ∗ *y*] | - ⇒ []

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

context *fixed-root*
begin
lemma *iteration-mono-eq*: **assumes** *xn*: *x* ^ *p* = (*n* :: *int*)
shows (*n div x* ^ *pm* + *x* ∗ *int pm*) *div int p* = *x*
⟨*proof*⟩

lemma *p0*: *p* ≠ *0* ⟨*proof*⟩

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 $(n\ div\ x\ \hat{}\ pm + x * int\ pm)\ div\ int\ p < x$
⟨*proof*⟩

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

termination
⟨*proof*⟩

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

abbreviation *root-main′* **where** *root-main′* ≡ *root-int-main′ pm* (*int pm*) (*int p*)

lemmas *root-main′-simps* = *root-int-main′.simps*[*of pm int pm int p*]

lemma *root-main′-newton-pos*: $x \geq 0 \Longrightarrow n \geq 0 \Longrightarrow$
root-main′ $x\ n$ = *root-newton-int-main* $x\ n \wedge$ (*root-main′* $x\ n = (y,b) \longrightarrow y \geq 0 \wedge y\hat{}p \leq n \wedge b = (y\hat{}p = n))$
⟨*proof*⟩

lemma *root-main′*: $x \geq 0 \Longrightarrow n \geq 0 \Longrightarrow$ *root-main′* $x\ n$ = *root-newton-int-main* $x\ n$
⟨*proof*⟩

lemma *root-main′-pos*: $x \geq 0 \Longrightarrow n \geq 0 \Longrightarrow$ *root-main′* $x\ n = (y,b) \Longrightarrow y \geq 0$
⟨*proof*⟩

lemma *root-main′-sound*: $x \geq 0 \Longrightarrow n \geq 0 \Longrightarrow$ *root-main′* $x\ n = (y,b) \Longrightarrow b = (y \,\hat{}\, p = n)$
⟨*proof*⟩

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*: $x \geq 0 \Longrightarrow n \geq 0 \Longrightarrow$ *root-main′* $x\ n = (y,b) \Longrightarrow y \,\hat{}\, p \leq n$
⟨*proof*⟩

lemma *root-newton-int-main-upper*:
shows $y \,\hat{}\, p \geq n \Longrightarrow y \geq 0 \Longrightarrow n \geq 0 \Longrightarrow$ *root-newton-int-main* $y\ n = (x,b) \Longrightarrow n < (x + 1) \,\hat{}\, p$

⟨*proof*⟩

lemma *root-main′-upper*:
$x \,\hat{}\, p \geq n \Longrightarrow x \geq 0 \Longrightarrow n \geq 0 \Longrightarrow root\text{-}main'\ x\ n = (y,b) \Longrightarrow n < (y + 1) \,\hat{}\, p$
⟨*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*: $root\text{-}int\text{-}main\ p\ n = (y,b)$
shows $y \geq 0 \wedge b = (y \,\hat{}\, p = n) \wedge (p > 0 \longrightarrow y \,\hat{}\, p \leq n \wedge n < (y + 1)\hat{}p)$
$\wedge (p > 0 \longrightarrow x \geq 0 \longrightarrow x \,\hat{}\, p = n \longrightarrow y = x \wedge b)$
⟨*proof*⟩

lemma *root-int-main*: **assumes** *n*: $n \geq 0$
and *rm*: $root\text{-}int\text{-}main\ p\ n = (y,b)$
shows $y \geq 0\ b = (y \,\hat{}\, p = n)\ p > 0 \Longrightarrow y \,\hat{}\, p \leq n\ p > 0 \Longrightarrow n < (y + 1)\hat{}p$
$p > 0 \Longrightarrow x \geq 0 \Longrightarrow x \,\hat{}\, p = n \Longrightarrow y = x \wedge b$
⟨*proof*⟩

lemma *root-int*[*simp*]: **assumes** *p*: $p \neq 0 \vee x \neq 1$
shows $set\ (root\text{-}int\ p\ x) = \{y\ .\ y \,\hat{}\, p = x\}$
⟨*proof*⟩

lemma *root-int-pos*: **assumes** *x*: $x \geq 0$ **and** *ri*: $root\text{-}int\ p\ x = y\ \#\ ys$
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 $\lfloor root\ p\ x \rfloor$ and $\lceil root\ p\ x \rceil$. To this end, we first develop algorithms for non-negative x, and later on these are used for the general case.

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

lemma *root-int-floor-pos-lower*: **assumes** *p0*: $p \neq 0$ **and** *x*: $x \geq 0$
shows $root\text{-}int\text{-}floor\text{-}pos\ p\ x \,\hat{}\, p \leq x$
⟨*proof*⟩

lemma *root-int-floor-pos-pos*: **assumes** *x*: $x \geq 0$
shows $root\text{-}int\text{-}floor\text{-}pos\ p\ x \geq 0$
⟨*proof*⟩

lemma *root-int-floor-pos-upper*: **assumes** *p0*: $p \neq 0$ **and** *x*: $x \geq 0$
shows $(root\text{-}int\text{-}floor\text{-}pos\ p\ x + 1) \,\hat{}\, p > x$
⟨*proof*⟩

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

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

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

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*))
⟨*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*⟩

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 $\frac{a}{b}^{1/p}$ 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 $\lfloor \mathit{sqrt}\ x \rfloor$ and $\lceil \mathit{sqrt}\ x \rceil$ 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 certifiying termination proofs. Here, non-linear equations of the form $(a_1x_1 + \ldots a_nx_n)^2 = p$ had to be solved over the integers, where p is a concrete polynomial. For example, for the equation $(ax + by)^2 = 4x^2 - 12xy + 9y^2$ one easily figures out that $a^2 = 4, b^2 = 9$, and $ab = -6$, which results in a possible solution $a = \sqrt{4} = 2, b = -\sqrt{9} = -3$.

4.1 The Babylonian method

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

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

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

We explicititly develop the algorithm on the integers and not on the naturals, as the calculations on the integers have been much easier. For example, $y - x + x = y$ on the integers, which would require the side-condition $y \geq x$ for the naturals. These conditions will make the reasoning much more tedious—as we have experienced in an earlier state of this development where everything was based on naturals.

Since the elements $x_0, x_1, x_2, \ldots$ are monotone decreasing, in the main algorithm we abort as soon as $x_i^2 \leq n$.

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

definition *sqrt-int-main'* :: *int* ⇒ *int* ⇒ *int* × *bool* **where**
[*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* ≤ *n then* (*x*, *x2* = *n*)
else sqrt-int-main' ((*n div x* + *x*) *div 2*) *n*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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

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

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*)
⟨*proof*⟩

lemma *sqrt-int-floor-pos*: **assumes** *x*: *x* ≥ *0*
shows *sqrt-int-floor-pos x* = ⌊ *sqrt* (*of-int x*) ⌋
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *sqrt-int-ceiling-pos*: **assumes** *x*: *x* ≥ *0*
shows *sqrt-int-ceiling-pos x* = ⌈ *sqrt* (*of-int x*) ⌉
⟨*proof*⟩

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*))
⟨*proof*⟩

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

definition *sqrt-int-ceiling x* = *root-int-ceiling 2 x*

lemma *sqrt-int-ceiling-code*[*code*]: *sqrt-int-ceiling x* = (*if x* ≥ *0 then sqrt-int-ceiling-pos x else* − *sqrt-int-floor-pos* (− *x*))
⟨*proof*⟩

lemma *sqrt-int-ceiling*[*simp*]: *sqrt-int-ceiling x* = ⌈ *sqrt* (*of-int x*) ⌉
⟨*proof*⟩

lemma *sqrt-int-ceiling-bound*: *0* ≤ *x* ⟹ *x* ≤ (*sqrt-int-ceiling x*)^*2*
⟨*proof*⟩

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 nat* (*take 1* (*sqrt-int* (*int x*)))
⟨*proof*⟩

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

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

lemma *sqrt-nat-floor-code*[*code*]: *sqrt-nat-floor x* = *sqrt-int-floor-pos* (*int x*)
⟨*proof*⟩

lemma *sqrt-nat-floor*[*simp*]: *sqrt-nat-floor x* = ⌊ *sqrt* (*real x*) ⌋
⟨*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*⟩

4.4 Square roots for the rationals

definition *sqrt-rat* :: *rat* ⇒ *rat list* **where**
sqrt-rat x = *root-rat 2 x*

lemma *sqrt-rat-code*[*code*]: *sqrt-rat x* = (*case quotient-of x of* (*z*,*n*) ⇒ (*case sqrt-int n of*
[] ⇒ []
| *sn* # *xs* ⇒ *map* (λ *sz. of-int sz* / *of-int sn*) (*sqrt-int z*)))
⟨*proof*⟩

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 *sqr-rat-of-int*: **assumes** *x*: *x* ∗ *x* = *rat-of-int i*
shows ∃ *j* :: *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 $0 < x$ 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 : ε > *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* $\Longrightarrow$ *sqrt-approx-main-impl* ε *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* ε > *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** ε: $\varepsilon > 0$
shows $|sqrt\text{-}approx\ \varepsilon\ x * sqrt\text{-}approx\ \varepsilon\ x - |x|| < \varepsilon$
⟨*proof*⟩

4.6 Some tests

Testing executabity and show that sqrt 2 is irrational

lemma $\neg\ (\exists\ i :: rat.\ i * i = 2)$
⟨*proof*⟩

Testing speed

lemma $\neg\ (\exists\ i :: int.\ i * i = 1234567890123456789012345678901234567890)$
⟨*proof*⟩

The following test

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

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

end

Acknowledgements

We thank Bertram Felgenhauer for 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.