Continued Fractions

Manuel Eberl

April 18, 2024

Abstract

This article provides a formalisation of continued fractions of real numbers and their basic properties. It also contains a proof of the classic result that the irrational numbers with periodic continued fraction expansions are precisely the quadratic irrationals, i. e. real numbers that fulfil a non-trivial quadratic equation $ax^2 + bx + c = 0$ with integer coefficients.

Particular attention is given to the continued fraction expansion of $\sqrt{D}$ for a non-square natural number D. Basic results about the length and structure of its period are provided, along with an executable algorithm to compute the period (and from it, the entire expansion).

This is then also used to provide a fairly efficient, executable, and fully formalised algorithm to compute solutions to Pell's equation $x^2 - Dy^2 = 1$. The performance is sufficiently good to find the solution to Archimedes's cattle problem in less than a second on a typical computer. This involves the value $D = 410286423278424$, for which the solution has over 200000 decimals.

Lastly, a derivation of the continued fraction expansions of Euler's number e and an executable function to compute continued fraction expansions using interval arithmetic is also provided.

Contents

1 Continued Fractions

theory *Continued-Fractions*
imports
Complex-Main
Coinductive.Lazy-LList
Coinductive.Coinductive-Nat
HOL−Number-Theory.Fib
HOL−Library.BNF-Corec
Coinductive.Coinductive-Stream
begin

1.1 Auxiliary results

coinductive *linfinite* :: *'a llist* ⇒ *bool* **where**
linfinite xs ⟹ *linfinite* (*LCons x xs*)

lemma *llength-llist-of-stream* [*simp*]: *llength* (*llist-of-stream xs*) = ∞
by (*simp add*: *not-lfinite-llength*)

lemma *linfinite-conv-llength*: *linfinite xs* ⟷ *llength xs* = ∞
proof
assume *linfinite xs*
thus *llength xs* = ∞
proof (*coinduction arbitrary*: *xs rule*: *enat-coinduct2*)
fix *xs* :: *'a llist*
assume *llength xs* ≠ *0 linfinite xs*
thus (∃ *xs'*::*'a llist*. *epred* (*llength xs*) = *llength xs'* ∧ *epred* ∞ = ∞ ∧ *linfinite xs'*) ∨
epred (*llength xs*) = *epred* ∞
by (*intro disjI1 exI*[*of* - *ltl xs*]) (*auto simp*: *linfinite.simps*[*of xs*])
next
fix *xs* :: *'a llist* **assume** *linfinite xs***thus** (*llength xs* = *0*) ⟷ (∞ = (*0*::*enat*))
by (*subst* (*asm*) *linfinite.simps*) *auto*
qed
next
assume *llength xs* = ∞
thus *linfinite xs*
proof (*coinduction arbitrary*: *xs*)
case *linfinite*
thus ∃ *xsa x*.
xs = *LCons x xsa* ∧
((∃ *xs*. *xsa* = *xs* ∧ *llength xs* = ∞) ∨
linfinite xsa)
by (*cases xs*) (*auto simp*: *eSuc-eq-infinity-iff*)
qed
qed

definition *lnth-default* :: *'a* ⇒ *'a llist* ⇒ *nat* ⇒ *'a* **where**
lnth-default dflt xs n = (*if n* < *llength xs then lnth xs n else dflt*)

lemma *lnth-default-code* [*code*]:
lnth-default dflt xs n =
(*if lnull xs then dflt else if n* = *0 then lhd xs else lnth-default dflt* (*ltl xs*) (*n* − *1*))
proof (*induction n arbitrary*: *xs*)
case *0*
thus *?case*
by (*cases xs*) (*auto simp*: *lnth-default-def simp flip*: *zero-enat-def*)
next
case (*Suc n*)
show *?case*
proof (*cases xs*)
case *LNil*
thus *?thesis*
by (*auto simp*: *lnth-default-def*)
next
case (*LCons x xs'*)
thus *?thesis*
by (*auto simp*: *lnth-default-def Suc-ile-eq*)
qed
qed

lemma *enat-le-iff*:
$enat\ n \le m \longleftrightarrow m = \infty \lor (\exists m'.\ m = enat\ m' \land n \le m')$
by (*cases m*) *auto*

lemma *enat-less-iff*:
$enat\ n < m \longleftrightarrow m = \infty \lor (\exists m'.\ m = enat\ m' \land n < m')$
by (*cases m*) *auto*

lemma *real-of-int-divide-in-Ints-iff*:
$real\text{-}of\text{-}int\ a\ /\ real\text{-}of\text{-}int\ b \in \mathbb{Z} \longleftrightarrow b\ dvd\ a \lor b = 0$
proof *safe*
assume $real\text{-}of\text{-}int\ a\ /\ real\text{-}of\text{-}int\ b \in \mathbb{Z}$ $b \neq 0$
then obtain *n* **where** $real\text{-}of\text{-}int\ a\ /\ real\text{-}of\text{-}int\ b = real\text{-}of\text{-}int\ n$
by (*auto simp*: *Ints-def*)
hence $real\text{-}of\text{-}int\ b * real\text{-}of\text{-}int\ n = real\text{-}of\text{-}int\ a$
using ‹$b \neq 0$› **by** (*auto simp*: *field-simps*)
also have $real\text{-}of\text{-}int\ b * real\text{-}of\text{-}int\ n = real\text{-}of\text{-}int\ (b * n)$
by *simp*
finally have $b * n = a$
by *linarith*
thus *b dvd a*
by *auto*
qed *auto*

lemma *frac-add-of-nat*: *frac* (*of-nat y* + *x*) = *frac x*
unfolding *frac-def* **by** *simp*

```
lemma frac-add-of-int: frac (of-int y + x) = frac x
  unfolding frac-def by simp

lemma frac-fraction: frac (real-of-int a / real-of-int b) = (a mod b) / b
proof -
  have frac (a / b) = frac ((a mod b + b * (a div b)) / b)
    by (subst mod-mult-div-eq) auto
  also have (a mod b + b * (a div b)) / b = of-int (a div b) + a mod b / b
    unfolding of-int-add by (subst add-divide-distrib) auto
  also have frac ... = frac (a mod b / b)
    by (rule frac-add-of-int)
  also have ... = a mod b / b
    by (simp add: floor-divide-of-int-eq frac-def)
  finally show ?thesis .
qed

lemma Suc-fib-ge: Suc (fib n) ≥ n
proof (induction n rule: fib.induct)
  case (3 n)
  show ?case
  proof (cases n < 2)
    case True
    thus ?thesis by (cases n) auto
  next
    case False
    hence Suc (Suc (Suc n)) ≤ Suc n + n by simp
    also have ... ≤ Suc (fib (Suc n)) + Suc (fib n)
      by (intro add-mono 3)
    also have ... = Suc (Suc (fib (Suc (Suc n))))
      by simp
    finally show ?thesis by (simp only: Suc-le-eq)
  qed
qed auto

lemma fib-ge: fib n ≥ n − 1
  using Suc-fib-ge[of n] by simp

lemma frac-diff-of-nat-right [simp]: frac (x − of-nat y) = frac x
  using floor-diff-of-int[of x int y] by (simp add: frac-def)

lemma of-nat-ge-1-iff: of-nat n ≥ (1 :: 'a :: linordered-semidom) ⟷ n > 0
  using of-nat-le-iff[of 1 n] unfolding of-nat-1 by auto

lemma not-frac-less-0: ¬frac x < 0
  by (simp add: frac-def not-less)

lemma frac-le-1: frac x ≤ 1
  unfolding frac-def by linarith
```

lemma *divide-in-Rats-iff1*:
  $(x::real) \in \mathbb{Q} \Longrightarrow x \neq 0 \Longrightarrow x\ /\ y \in \mathbb{Q} \longleftrightarrow y \in \mathbb{Q}$
proof *safe*
  assume $*$: $x \in \mathbb{Q}$ $x \neq 0$ $x\ /\ y \in \mathbb{Q}$
  from $*(1,3)$ **have** $x\ /\ (x\ /\ y) \in \mathbb{Q}$
    by (*rule Rats-divide*)
  also from $*$ **have** $x\ /\ (x\ /\ y) = y$ **by** *simp*
  finally show $y \in \mathbb{Q}$ **.**
qed (*auto intro*: *Rats-divide*)

lemma *divide-in-Rats-iff2*:
  $(y::real) \in \mathbb{Q} \Longrightarrow y \neq 0 \Longrightarrow x\ /\ y \in \mathbb{Q} \longleftrightarrow x \in \mathbb{Q}$
proof *safe*
  assume $*$: $y \in \mathbb{Q}$ $y \neq 0$ $x\ /\ y \in \mathbb{Q}$
  from $*(3,1)$ **have** $x\ /\ y * y \in \mathbb{Q}$
    by (*rule Rats-mult*)
  also from $*$ **have** $x\ /\ y * y = x$ **by** *simp*
  finally show $x \in \mathbb{Q}$ **.**
qed (*auto intro*: *Rats-divide*)

lemma *add-in-Rats-iff1*: $x \in \mathbb{Q} \Longrightarrow x + y \in \mathbb{Q} \longleftrightarrow y \in \mathbb{Q}$
  using *Rats-diff*[*of* $x + y$ x] **by** *auto*

lemma *add-in-Rats-iff2*: $y \in \mathbb{Q} \Longrightarrow x + y \in \mathbb{Q} \longleftrightarrow x \in \mathbb{Q}$
  using *Rats-diff*[*of* $x + y$ y] **by** *auto*

lemma *diff-in-Rats-iff1*: $x \in \mathbb{Q} \Longrightarrow x - y \in \mathbb{Q} \longleftrightarrow y \in \mathbb{Q}$
  using *Rats-diff*[*of* x $x - y$] **by** *auto*

lemma *diff-in-Rats-iff2*: $y \in \mathbb{Q} \Longrightarrow x - y \in \mathbb{Q} \longleftrightarrow x \in \mathbb{Q}$
  using *Rats-add*[*of* $x - y$ y] **by** *auto*

lemma *frac-in-Rats-iff* [*simp*]: *frac* $x \in \mathbb{Q} \longleftrightarrow x \in \mathbb{Q}$
  by (*simp add*: *frac-def diff-in-Rats-iff2*)

lemma *filterlim-sequentially-shift*:
  filterlim ($\lambda n.$ f $(n + m)$) F *sequentially* $\longleftrightarrow$ *filterlim* f F *sequentially*
proof (*induction* m)
  case (*Suc* m)
  have *filterlim* ($\lambda n.$ f $(n + Suc\ m)$) F *at-top* $\longleftrightarrow$
        filterlim ($\lambda n.$ f $(Suc\ n + m)$) F *at-top* **by** *simp*
  also have $\ldots \longleftrightarrow$ *filterlim* ($\lambda n.$ f $(n + m)$) F *at-top*
    by (*rule filterlim-sequentially-Suc*)
  also have $\ldots \longleftrightarrow$ *filterlim* f F *at-top*
    by (*rule Suc.IH*)
  finally show *?case* **.**
qed *simp-all*

1.2 Bounds on alternating decreasing sums

lemma *alternating-decreasing-sum-bounds*:
  fixes f :: *nat* ⇒ *'a* :: {*linordered-ring*, *ring-1*}
  assumes $m \leq n$ $\bigwedge k.\ k \in \{m..n\} \Longrightarrow f\ k \geq 0$
    $\bigwedge k.\ k \in \{m..<n\} \Longrightarrow f\ (Suc\ k) \leq f\ k$
  defines $S \equiv (\lambda m.\ (\sum k=m..n.\ (-1)\ \hat{}\ k * f\ k))$
  shows *if even m then* $S\ m \in \{0..f\ m\}$ *else* $S\ m \in \{-f\ m..0\}$
  using *assms*(*1*)
proof (*induction rule*: *inc-induct*)
  case (*step m'*)
  have [*simp*]: $-a \leq b \longleftrightarrow a + b \geq (0 :: 'a)$ **for** *a b*
    by (*metis le-add-same-cancel1 minus-add-cancel*)
  have [*simp*]: $S\ m' = (-1)\ \hat{}\ m' * f\ m' + S\ (Suc\ m')$
    using *step.hyps* **unfolding** *S-def*
    by (*subst sum.atLeast-Suc-atMost*) *simp-all*
  from *step.hyps* **have** *nonneg*: $f\ m' \geq 0$
    by (*intro assms*) *auto*
  from *step.hyps* **have** *mono*: $f\ (Suc\ m') \leq f\ m'$
    by (*intro assms*) *auto*
  show *?case*
  proof (*cases even m'*)
    case *True*
    hence $0 \leq f\ (Suc\ m') + S\ (Suc\ m')$
      using *step.IH* **by** *simp*
    also note *mono*
    finally show *?thesis* **using** *True step.IH* **by** *auto*
  next
    case *False*
    with *step.IH* **have** $S\ (Suc\ m') \leq f\ (Suc\ m')$
      by *simp*
    also note *mono*
    finally show *?thesis* **using** *step.IH False* **by** *auto*
  qed
qed (*insert assms*, *auto*)

lemma *alternating-decreasing-sum-bounds'*:
  fixes f :: *nat* ⇒ *'a* :: {*linordered-ring*, *ring-1*}
  assumes $m < n$ $\bigwedge k.\ k \in \{m..n-1\} \Longrightarrow f\ k \geq 0$
    $\bigwedge k.\ k \in \{m..<n-1\} \Longrightarrow f\ (Suc\ k) \leq f\ k$
  defines $S \equiv (\lambda m.\ (\sum k=m..<n.\ (-1)\ \hat{}\ k * f\ k))$
  shows *if even m then* $S\ m \in \{0..f\ m\}$ *else* $S\ m \in \{-f\ m..0\}$
proof (*cases n*)
  case *0*
  thus *?thesis* **using** *assms* **by** *auto*
next
  case (*Suc n'*)
  hence *if even m then* $(\sum k=m..n-1.\ (-1)\ \hat{}\ k * f\ k) \in \{0..f\ m\}$
    else $(\sum k=m..n-1.\ (-1)\ \hat{}\ k * f\ k) \in \{-f\ m..0\}$
    using *assms* **by** (*intro alternating-decreasing-sum-bounds*) *auto*

also have $(\sum k{=}m..n{-}1.\ (-1)\ \hat{}\ k * f\ k) = S\ m$
unfolding *S-def* **by** (*intro sum.cong*) (*auto simp*: *Suc*)
finally show *?thesis* **.**
qed

lemma *alternating-decreasing-sum-upper-bound*:
fixes $f :: nat \Rightarrow 'a :: \{linordered\text{-}ring, ring\text{-}1\}$
assumes $m \le n$ $\bigwedge k.\ k \in \{m..n\} \Longrightarrow f\ k \ge 0$
$\bigwedge k.\ k \in \{m..{<}n\} \Longrightarrow f\ (Suc\ k) \le f\ k$
shows $(\sum k{=}m..n.\ (-1)\ \hat{}\ k * f\ k) \le f\ m$
using *alternating-decreasing-sum-bounds*[*of m n f*, *OF assms*] *assms*(*1*)
by (*auto split*: *if-splits intro*: *order.trans*[*OF - assms*(*2*)])

lemma *alternating-decreasing-sum-upper-bound′*:
fixes $f :: nat \Rightarrow 'a :: \{linordered\text{-}ring, ring\text{-}1\}$
assumes $m < n$ $\bigwedge k.\ k \in \{m..n{-}1\} \Longrightarrow f\ k \ge 0$
$\bigwedge k.\ k \in \{m..{<}n{-}1\} \Longrightarrow f\ (Suc\ k) \le f\ k$
shows $(\sum k{=}m..{<}n.\ (-1)\ \hat{}\ k * f\ k) \le f\ m$
using *alternating-decreasing-sum-bounds′*[*of m n f*, *OF assms*] *assms*(*1*)
by (*auto split*: *if-splits intro*: *order.trans*[*OF - assms*(*2*)])

lemma *abs-alternating-decreasing-sum-upper-bound*:
fixes $f :: nat \Rightarrow 'a :: \{linordered\text{-}ring, ring\text{-}1\}$
assumes $m \le n$ $\bigwedge k.\ k \in \{m..n\} \Longrightarrow f\ k \ge 0$
$\bigwedge k.\ k \in \{m..{<}n\} \Longrightarrow f\ (Suc\ k) \le f\ k$
shows $|(\sum k{=}m..n.\ (-1)\ \hat{}\ k * f\ k)| \le f\ m$ (**is** *abs ?S* ≤ -)
using *alternating-decreasing-sum-bounds*[*of m n f*, *OF assms*]
by (*auto split*: *if-splits simp*: *minus-le-iff*)

lemma *abs-alternating-decreasing-sum-upper-bound′*:
fixes $f :: nat \Rightarrow 'a :: \{linordered\text{-}ring, ring\text{-}1\}$
assumes $m < n$ $\bigwedge k.\ k \in \{m..n{-}1\} \Longrightarrow f\ k \ge 0$
$\bigwedge k.\ k \in \{m..{<}n{-}1\} \Longrightarrow f\ (Suc\ k) \le f\ k$
shows $|(\sum k{=}m..{<}n.\ (-1)\ \hat{}\ k * f\ k)| \le f\ m$
using *alternating-decreasing-sum-bounds′*[*of m n f*, *OF assms*]
by (*auto split*: *if-splits simp*: *minus-le-iff*)

lemma *abs-alternating-decreasing-sum-lower-bound*:
fixes $f :: nat \Rightarrow 'a :: \{linordered\text{-}ring, ring\text{-}1\}$
assumes $m < n$ $\bigwedge k.\ k \in \{m..n\} \Longrightarrow f\ k \ge 0$
$\bigwedge k.\ k \in \{m..{<}n\} \Longrightarrow f\ (Suc\ k) \le f\ k$
shows $|(\sum k{=}m..n.\ (-1)\ \hat{}\ k * f\ k)| \ge f\ m - f\ (Suc\ m)$
proof –
have $(\sum k{=}m..n.\ (-1)\ \hat{}\ k * f\ k) = (\sum k{\in}insert\ m\ \{m{<}..n\}.\ (-1)\ \hat{}\ k * f\ k)$
using *assms* **by** (*intro sum.cong*) *auto*
also have $\ldots = (-1)\ \hat{}\ m * f\ m + (\sum k{\in}\{m{<}..n\}.\ (-1)\ \hat{}\ k * f\ k)$
by *auto*
also have $(\sum k{\in}\{m{<}..n\}.\ (-1)\ \hat{}\ k * f\ k) = (\sum k{\in}\{m..{<}n\}.\ (-1)\ \hat{}\ Suc\ k * f\ (Suc\ k))$

by (*intro sum.reindex-bij-witness*[*of* - *Suc* $\lambda i.\ i - 1$]) *auto*
also have (-1)^*m* ∗ *f m* + ... = (-1)^*m* ∗ *f m* − ($\sum k \in \{m..<n\}$. (-1) ^ *k* ∗ *f* (*Suc k*))
by (*simp add*: *sum-negf*)
also have |...| ≥ |(-1)^*m* ∗ *f m*| − |($\sum k \in \{m..<n\}$. (-1) ^ *k* ∗ *f* (*Suc k*))|
by (*rule abs-triangle-ineq2*)
also have |(-1)^*m* ∗ *f m*| = *f m*
using *assms* **by** (*cases even m*) *auto*
finally have *f m* − |$\sum k = m..<n$. (− 1) ^ *k* ∗ *f* (*Suc k*)|
≤ |$\sum k = m..n$. (− 1) ^ *k* ∗ *f k*| **.**
moreover have *f m* − |($\sum k \in \{m..<n\}$. (-1) ^ *k* ∗ *f* (*Suc k*))| ≥ *f m* − *f* (*Suc m*)
using *assms* **by** (*intro diff-mono abs-alternating-decreasing-sum-upper-bound′*) *auto*
ultimately show *?thesis* **by** (*rule order.trans*[*rotated*])
qed

lemma *abs-alternating-decreasing-sum-lower-bound′*:
fixes *f* :: *nat* ⇒ ′*a* :: {*linordered-ring*, *ring-1*}
assumes *m*+*1* < *n* ⋀*k*. *k* ∈ {*m..n*} ⟹ *f k* ≥ *0*
⋀*k*. *k* ∈ {*m..<n*} ⟹ *f* (*Suc k*) ≤ *f k*
shows |($\sum k=m..<n$. (-1) ^ *k* ∗ *f k*)| ≥ *f m* − *f* (*Suc m*)
proof (*cases n*)
case *0*
thus *?thesis* **using** *assms* **by** *auto*
next
case (*Suc n′*)
hence |($\sum k=m..n-1$. (-1) ^ *k* ∗ *f k*)| ≥ *f m* − *f* (*Suc m*)
using *assms* **by** (*intro abs-alternating-decreasing-sum-lower-bound*) *auto*
also have ($\sum k=m..n-1$. (-1) ^ *k* ∗ *f k*) = ($\sum k=m..<n$. (-1) ^ *k* ∗ *f k*)
by (*intro sum.cong*) (*auto simp*: *Suc*)
finally show *?thesis* **.**
qed

lemma *alternating-decreasing-suminf-bounds*:
assumes ⋀*k*. *f k* ≥ (*0* :: *real*) ⋀*k*. *f* (*Suc k*) ≤ *f k*
f ⟶ *0*
shows ($\sum k$. (-1) ^ *k* ∗ *f k*) ∈ {*f 0* − *f 1*..*f 0*}
proof −
have *summable* (λ*k*. (-1) ^ *k* ∗ *f k*)
by (*intro summable-Leibniz′ assms*)
hence *lim*: (λ*n*. $\sum k \le n$. (-1) ^ *k* ∗ *f k*) ⟶ ($\sum k$. (-1) ^ *k* ∗ *f k*)
by (*auto dest*: *summable-LIMSEQ′*)
have *bounds*: ($\sum k=0..n$. (− 1) ^ *k* ∗ *f k*) ∈ {*f 0* − *f 1*..*f 0*}
if *n* > *0* **for** *n*
using *alternating-decreasing-sum-bounds*[*of 1 n f*] *assms that*
by (*subst sum.atLeast-Suc-atMost*) *auto*
note [*simp*] = *atLeast0AtMost*
note [*intro*!] = *eventually-mono*[*OF eventually-gt-at-top*[*of 0*]]

from *lim* **have** $(\sum k.\ (-1)\ \hat{}\ k * f\ k) \geq f\ 0 - f\ 1$
by (*rule tendsto-lowerbound*) (*insert bounds*, *auto*)
moreover from *lim* **have** $(\sum k.\ (-1)\ \hat{}\ k * f\ k) \leq f\ 0$
by (*rule tendsto-upperbound*) (*use bounds* **in** *auto*)
ultimately show *?thesis* **by** *simp*
qed

lemma
assumes $\bigwedge k.\ k \geq m \Longrightarrow f\ k \geq (0 :: real)$
$\bigwedge k.\ k \geq m \Longrightarrow f\ (Suc\ k) \leq f\ k\ f \longrightarrow 0$
defines $S \equiv (\sum k.\ (-1)\ \hat{}(k + m) * f\ (k + m))$
shows *summable-alternating-decreasing*: $summable\ (\lambda k.\ (-1)\ \hat{}(k + m) * f\ (k + m))$
and *alternating-decreasing-suminf-bounds'*:
$if\ even\ m\ then\ S \in \{f\ m - f\ (Suc\ m)\ ..\ f\ m\}$
$else\ S \in \{-f\ m..f\ (Suc\ m) - f\ m\}$ (**is** *?th1*)
and *abs-alternating-decreasing-suminf*:
$abs\ S \in \{f\ m - f\ (Suc\ m)..f\ m\}$ (**is** *?th2*)
proof −
have *summable*: $summable\ (\lambda k.\ (-1)\ \hat{}\ k * f\ (k + m))$
using *assms* **by** (*intro summable-Leibniz'*) (*auto simp*: *filterlim-sequentially-shift*)
thus $summable\ (\lambda k.\ (-1)\ \hat{}(k + m) * f\ (k + m))$
by (*subst add.commute*) (*auto simp*: *power-add mult.assoc intro*: *summable-mult*)
have $S = (\sum k.\ (-1)\ \hat{}\ m * ((-1)\ \hat{}\ k * f\ (k + m)))$
by (*simp add*: *S-def power-add mult-ac*)
also have $\ldots = (-1)\ \hat{}\ m * (\sum k.\ (-1)\ \hat{}\ k * f\ (k + m))$
using *summable* **by** (*rule suminf-mult*)
finally have $S = (-\ 1)\ \hat{}\ m * (\sum k.\ (-\ 1)\ \hat{}\ k * f\ (k + m))$ **.**
moreover have $(\sum k.\ (-1)\ \hat{}\ k * f\ (k + m)) \in$
$\{f\ (0 + m) - f\ (1 + m)\ ..\ f\ (0 + m)\}$
using *assms*
by (*intro alternating-decreasing-suminf-bounds*)
(*auto simp*: *filterlim-sequentially-shift*)
ultimately show *?th1* **by** (*auto split*: *if-splits*)
thus *?th2* **using** *assms*(*2*)[*of m*] **by** (*auto split*: *if-splits*)
qed

lemma
assumes $\bigwedge k.\ k \geq m \Longrightarrow f\ k \geq (0 :: real)$
$\bigwedge k.\ k \geq m \Longrightarrow f\ (Suc\ k) < f\ k\ f \longrightarrow 0$
defines $S \equiv (\sum k.\ (-1)\ \hat{}\ (k + m) * f\ (k + m))$
shows *alternating-decreasing-suminf-bounds-strict'*:
$if\ even\ m\ then\ S \in \{f\ m - f\ (Suc\ m){<}..{<}f\ m\}$
$else\ S \in \{-f\ m{<}..{<}f\ (Suc\ m) - f\ m\}$ (**is** *?th1*)
and *abs-alternating-decreasing-suminf-strict*:
$abs\ S \in \{f\ m - f\ (Suc\ m){<}..{<}f\ m\}$ (**is** *?th2*)
proof −
define S' **where** $S' = (\sum k.\ (-1)\ \hat{}(k + Suc\ (Suc\ m)) * f\ (k + Suc\ (Suc\ m)))$

have ($\lambda k.$ (-1) ^ $(k + m) * f$ $(k + m)$) *sums S* **using** *assms* **unfolding** *S-def*
by (*intro summable-sums summable-Leibniz′ summable-alternating-decreasing*)
(*auto simp*: *less-eq-real-def*)
from *sums-split-initial-segment*[*OF this*, *of 2*]
have S': $S' = S - (-1)$ ^ $m * (f\ m - f\ (Suc\ m))$
by (*simp-all add*: *sums-iff S′-def algebra-simps lessThan-nat-numeral*)
have *if even* (*Suc* (*Suc m*)) *then* $S' \in \{f\ (Suc\ (Suc\ m)) - f\ (Suc\ (Suc\ (Suc\ m)))..f\ (Suc\ (Suc\ m))\}$
else $S' \in \{- f\ (Suc\ (Suc\ m))..f\ (Suc\ (Suc\ (Suc\ m))) - f\ (Suc\ (Suc\ m))\}$
unfolding *S′-def*
using *assms* **by** (*intro alternating-decreasing-suminf-bounds′*) (*auto simp*: *less-eq-real-def*)
thus *?th1* **using** *assms*(*2*)[*of Suc m*] *assms*(*2*)[*of Suc* (*Suc m*)]
unfolding S' **by** (*auto simp*: *algebra-simps*)
thus *?th2* **using** *assms*(*2*)[*of m*] **by** (*auto split*: *if-splits*)
qed

datatype *cfrac* = *CFrac int nat llist*

quickcheck-generator *cfrac constructors*: *CFrac*

lemma *type-definition-cfrac′*:
type-definition ($\lambda x.$ *case x of CFrac a b* ⇒ (*a*, *b*)) (λ(*x*,*y*). *CFrac x y*) *UNIV*
by (*auto simp*: *type-definition-def split*: *cfrac.splits*)

setup-lifting *type-definition-cfrac′*

lift-definition *cfrac-of-int* :: *int* ⇒ *cfrac* **is**
$\lambda n.$ (*n*, *LNil*) **.**

lemma *cfrac-of-int-code* [*code*]: *cfrac-of-int n* = *CFrac n LNil*
by (*auto simp*: *cfrac-of-int-def*)

lift-definition *cfrac-of-stream* :: *int stream* ⇒ *cfrac* **is**
$\lambda xs.$ (*shd xs*, *llist-of-stream* (*smap* ($\lambda x.$ *nat* $(x - 1)$) (*stl xs*))) **.**

instantiation *cfrac* :: *zero*
begin
definition *zero-cfrac* **where** *0* = *cfrac-of-int 0*
instance ..
end

instantiation *cfrac* :: *one*
begin
definition *one-cfrac* **where** *1* = *cfrac-of-int 1*
instance ..
end

lift-definition *cfrac-tl* :: *cfrac* ⇒ *cfrac* **is**
$\lambda(\text{-}, bs) \Rightarrow$ *case bs of LNil* ⇒ (*1*, *LNil*) | *LCons b bs'* ⇒ (*int b* + *1*, *bs'*) **.**

lemma *cfrac-tl-code* [*code*]:
cfrac-tl (*CFrac a bs*) =
(*case bs of LNil* ⇒ *CFrac 1 LNil* | *LCons b bs'* ⇒ *CFrac* (*int b* + *1*) *bs'*)
by (*auto simp*: *cfrac-tl-def split*: *llist.splits*)

definition *cfrac-drop* :: *nat* ⇒ *cfrac* ⇒ *cfrac* **where**
cfrac-drop n c = (*cfrac-tl* ^^ *n*) *c*

lemma *cfrac-drop-Suc-right*: *cfrac-drop* (*Suc n*) *c* = *cfrac-drop n* (*cfrac-tl c*)
by (*simp add*: *cfrac-drop-def funpow-Suc-right del*: *funpow.simps*)

lemma *cfrac-drop-Suc-left*: *cfrac-drop* (*Suc n*) *c* = *cfrac-tl* (*cfrac-drop n c*)
by (*simp add*: *cfrac-drop-def*)

lemma *cfrac-drop-add*: *cfrac-drop* (*m* + *n*) *c* = *cfrac-drop m* (*cfrac-drop n c*)
by (*simp add*: *cfrac-drop-def funpow-add*)

lemma *cfrac-drop-0* [*simp*]: *cfrac-drop 0* = ($\lambda x.\ x$)
by (*simp add*: *fun-eq-iff cfrac-drop-def*)

lemma *cfrac-drop-1* [*simp*]: *cfrac-drop 1* = *cfrac-tl*
by (*simp add*: *fun-eq-iff cfrac-drop-def*)

lift-definition *cfrac-length* :: *cfrac* ⇒ *enat* **is**
$\lambda(\text{-}, bs) \Rightarrow$ *llength bs* **.**

lemma *cfrac-length-code* [*code*]: *cfrac-length* (*CFrac a bs*) = *llength bs*
by (*simp add*: *cfrac-length-def*)

lemma *cfrac-length-tl* [*simp*]: *cfrac-length* (*cfrac-tl c*) = *cfrac-length c* − *1*
by *transfer* (*auto split*: *llist.splits*)

lemma *enat-diff-Suc-right* [*simp*]: *m* − *enat* (*Suc n*) = *m* − *n* − *1*
by (*auto simp*: *diff-enat-def enat-1-iff split*: *enat.splits*)

lemma *cfrac-length-drop* [*simp*]: *cfrac-length* (*cfrac-drop n c*) = *cfrac-length c* − *n*
by (*induction n*) (*auto simp*: *cfrac-drop-def*)

lemma *cfrac-length-of-stream* [*simp*]: *cfrac-length* (*cfrac-of-stream xs*) = ∞
by *transfer auto*

lift-definition *cfrac-nth* :: *cfrac* ⇒ *nat* ⇒ *int* **is**
λ(*a* :: *int*, *bs* :: *nat llist*). λ(*n* :: *nat*).
if n = *0 then a*
else if n ≤ *llength bs then int* (*lnth bs* (*n* − *1*)) + *1 else 1* **.**

lemma *cfrac-nth-code* [*code*]:
  cfrac-nth (*CFrac a bs*) *n* = (*if n* = *0 then a else lnth-default 0 bs* (*n* − *1*) + *1*)
proof −
  have *n* > *0* ⟶ *enat* (*n* − *Suc 0*) < *llength bs* ⟷ *enat n* ≤ *llength bs*
    by (*metis Suc-ile-eq Suc-pred*)
  thus *?thesis* **by** (*auto simp*: *cfrac-nth-def lnth-default-def*)
qed

lemma *cfrac-nth-nonneg* [*simp*, *intro*]: *n* > *0* ⟹ *cfrac-nth c n* ≥ *0*
  by *transfer auto*

lemma *cfrac-nth-nonzero* [*simp*]: *n* > *0* ⟹ *cfrac-nth c n* ≠ *0*
  by *transfer* (*auto split*: *if-splits*)

lemma *cfrac-nth-pos*[*simp*, *intro*]: *n* > *0* ⟹ *cfrac-nth c n* > *0*
  by *transfer auto*

lemma *cfrac-nth-ge-1*[*simp*, *intro*]: *n* > *0* ⟹ *cfrac-nth c n* ≥ *1*
  by *transfer auto*

lemma *cfrac-nth-not-less-1*[*simp*, *intro*]: *n* > *0* ⟹ ¬*cfrac-nth c n* < *1*
  by *transfer* (*auto split*: *if-splits*)

lemma *cfrac-nth-tl* [*simp*]: *cfrac-nth* (*cfrac-tl c*) *n* = *cfrac-nth c* (*Suc n*)
  apply *transfer*
  apply (*auto split*: *llist.splits nat.splits simp*: *Suc-ile-eq lnth-LCons enat-0-iff*
      simp flip: *zero-enat-def*)
  done

lemma *cfrac-nth-drop* [*simp*]: *cfrac-nth* (*cfrac-drop n c*) *m* = *cfrac-nth c* (*m* + *n*)
  by (*induction n arbitrary*: *m*) (*auto simp*: *cfrac-drop-def*)

lemma *cfrac-nth-0-of-int* [*simp*]: *cfrac-nth* (*cfrac-of-int n*) *0* = *n*
  by *transfer auto*

lemma *cfrac-nth-gt0-of-int* [*simp*]: *m* > *0* ⟹ *cfrac-nth* (*cfrac-of-int n*) *m* = *1*
  by *transfer* (*auto simp*: *enat-0-iff*)

lemma *cfrac-nth-of-stream*:
  assumes *sset* (*stl xs*) ⊆ {*0*<..}
  shows *cfrac-nth* (*cfrac-of-stream xs*) *n* = *snth xs n*
  using *assms*
proof (*transfer*′, *goal-cases*)
  case (*1 xs n*)
  thus *?case*
    by (*cases xs*; *cases n*) (*auto simp*: *subset-iff*)
qed

lift-definition *cfrac* :: (*nat* ⇒ *int*) ⇒ *cfrac* **is**
λ*f*. (*f 0*, *inf-llist* (λ*n*. *nat* (*f* (*Suc n*) − *1*))) **.**

definition *is-cfrac* :: (*nat* ⇒ *int*) ⇒ *bool* **where** *is-cfrac f* ⟷ (∀ *n*>*0*. *f n* > *0*)

lemma *cfrac-nth-cfrac* [*simp*]:
assumes *is-cfrac f*
shows *cfrac-nth* (*cfrac f*) *n* = *f n*
using *assms* **unfolding** *is-cfrac-def* **by** *transfer auto*

lemma *llength-eq-infty-lnth*: *llength b* = ∞ ⟹ *inf-llist* (*lnth b*) = *b*
by (*simp add*: *llength-eq-infty-conv-lfinite*)

lemma *cfrac-cfrac-nth* [*simp*]: *cfrac-length c* = ∞ ⟹ *cfrac* (*cfrac-nth c*) = *c*
by *transfer* (*auto simp*: *llength-eq-infty-lnth*)

lemma *cfrac-length-cfrac* [*simp*]: *cfrac-length* (*cfrac f*) = ∞
by *transfer auto*

lift-definition *cfrac-of-list* :: *int list* ⇒ *cfrac* **is**
λ*xs*. *if xs* = [] *then* (*0*, *LNil*) *else* (*hd xs*, *llist-of* (*map* (λ*n*. *nat n* − *1*) (*tl xs*))) **.**

lemma *cfrac-length-of-list* [*simp*]: *cfrac-length* (*cfrac-of-list xs*) = *length xs* − *1*
by *transfer* (*auto simp*: *zero-enat-def*)

lemma *cfrac-of-list-Nil* [*simp*]: *cfrac-of-list* [] = *0*
unfolding *zero-cfrac-def* **by** *transfer auto*

lemma *cfrac-nth-of-list* [*simp*]:
assumes *n* < *length xs* **and** ∀ *i*∈{*0*<..<*length xs*}. *xs* ! *i* > *0*
shows *cfrac-nth* (*cfrac-of-list xs*) *n* = *xs* ! *n*
using *assms*
proof (*transfer*, *goal-cases*)
case (*1 n xs*)
show *?case*
proof (*cases n*)
case (*Suc n′*)
with *1* **have** *xs* ! *n* > *0*
using *1* **by** *auto*
hence *int* (*nat* (*tl xs* ! *n′*) − *Suc 0*) + *1* = *xs* ! *Suc n′*
using *1*(*1*) *Suc* **by** (*auto simp*: *nth-tl of-nat-diff*)
thus *?thesis*
using *Suc 1*(*1*) **by** (*auto simp*: *hd-conv-nth zero-enat-def*)
qed (*use 1* **in** ‹*auto simp*: *hd-conv-nth*›)
qed

primcorec *cfrac-of-real-aux* :: *real* ⇒ *nat llist* **where**

cfrac-of-real-aux x =
(*if x* ∈ {*0*<..<*1*} *then LCons* (*nat* ⌊*1*/*x*⌋ − *1*) (*cfrac-of-real-aux* (*frac* (*1*/*x*))) *else LNil*)

lemma *cfrac-of-real-aux-code* [*code*]:
cfrac-of-real-aux x =
(*if x* > *0* ∧ *x* < *1 then LCons* (*nat* ⌊*1*/*x*⌋ − *1*) (*cfrac-of-real-aux* (*frac* (*1*/*x*))) *else LNil*)
by (*subst cfrac-of-real-aux.code*) *auto*

lemma *cfrac-of-real-aux-LNil* [*simp*]: *x* ∉ {*0*<..<*1*} ⟹ *cfrac-of-real-aux x* = *LNil*
by (*subst cfrac-of-real-aux.code*) *auto*

lemma *cfrac-of-real-aux-0* [*simp*]: *cfrac-of-real-aux 0* = *LNil*
by (*subst cfrac-of-real-aux.code*) *auto*

lemma *cfrac-of-real-aux-eq-LNil-iff* [*simp*]: *cfrac-of-real-aux x* = *LNil* ⟷ *x* ∉ {*0*<..<*1*}
by (*subst cfrac-of-real-aux.code*) *auto*

lemma *lnth-cfrac-of-real-aux*:
assumes *n* < *llength* (*cfrac-of-real-aux x*)
shows *lnth* (*cfrac-of-real-aux x*) (*Suc n*) = *lnth* (*cfrac-of-real-aux* (*frac* (*1*/*x*))) *n*
using *assms*
apply (*induction n arbitrary*: *x*)
apply (*subst cfrac-of-real-aux.code*)
apply *auto* []
apply (*subst cfrac-of-real-aux.code*)
apply (*auto*)
done

lift-definition *cfrac-of-real* :: *real* ⇒ *cfrac* **is**
λ*x*. (⌊*x*⌋, *cfrac-of-real-aux* (*frac x*)) **.**

lemma *cfrac-of-real-code* [*code*]: *cfrac-of-real x* = *CFrac* ⌊*x*⌋ (*cfrac-of-real-aux* (*frac x*))
by (*simp add*: *cfrac-of-real-def*)

lemma *eq-epred-iff*: *m* = *epred n* ⟷ *m* = *0* ∧ *n* = *0* ∨ *n* = *eSuc m*
by (*cases m*; *cases n*) (*auto simp*: *enat-0-iff enat-eSuc-iff infinity-eq-eSuc-iff*)

lemma *epred-eq-iff*: *epred n* = *m* ⟷ *m* = *0* ∧ *n* = *0* ∨ *n* = *eSuc m*
by (*cases m*; *cases n*) (*auto simp*: *enat-0-iff enat-eSuc-iff infinity-eq-eSuc-iff*)

lemma *epred-less*: *n* > *0* ⟹ *n* ≠ ∞ ⟹ *epred n* < *n*
by (*cases n*) (*auto simp*: *enat-0-iff*)

lemma *cfrac-nth-of-real-0* [*simp*]:
  cfrac-nth (*cfrac-of-real x*) *0* = ⌊*x*⌋
  by *transfer auto*

lemma *frac-eq-0* [*simp*]: *x* ∈ ℤ ⟹ *frac x* = *0*
  by *simp*

lemma *cfrac-tl-of-real*:
  assumes *x* ∉ ℤ
  shows *cfrac-tl* (*cfrac-of-real x*) = *cfrac-of-real* (*1* / *frac x*)
  using *assms*
proof (*transfer*, *goal-cases*)
  case (*1 x*)
  hence *int* (*nat* ⌊*1* / *frac x*⌋ − *Suc 0*) + *1* = ⌊*1* / *frac x*⌋
    by (*subst of-nat-diff*) (*auto simp*: *le-nat-iff frac-le-1*)
  with ‹*x* ∉ ℤ› **show** *?case*
  by (*subst cfrac-of-real-aux.code*) (*auto split*: *llist.splits simp*: *frac-lt-1*)
qed

lemma *cfrac-nth-of-real-Suc*:
  assumes *x* ∉ ℤ
  shows *cfrac-nth* (*cfrac-of-real x*) (*Suc n*) = *cfrac-nth* (*cfrac-of-real* (*1* / *frac x*)) *n*
proof −
  have *cfrac-nth* (*cfrac-of-real x*) (*Suc n*) =
      cfrac-nth (*cfrac-tl* (*cfrac-of-real x*)) *n*
    by *simp*
  also have *cfrac-tl* (*cfrac-of-real x*) = *cfrac-of-real* (*1* / *frac x*)
    by (*simp add*: *cfrac-tl-of-real assms*)
  finally show *?thesis* **.**
qed

fun *conv* :: *cfrac* ⇒ *nat* ⇒ *real* **where**
  conv c 0 = *real-of-int* (*cfrac-nth c 0*)
| *conv c* (*Suc n*) = *real-of-int* (*cfrac-nth c 0*) + *1* / *conv* (*cfrac-tl c*) *n*

The numerator and denominator of a convergent:

fun *conv-num* :: *cfrac* ⇒ *nat* ⇒ *int* **where**
  conv-num c 0 = *cfrac-nth c 0*
| *conv-num c* (*Suc 0*) = *cfrac-nth c 1* ∗ *cfrac-nth c 0* + *1*
| *conv-num c* (*Suc* (*Suc n*)) = *cfrac-nth c* (*Suc* (*Suc n*)) ∗ *conv-num c* (*Suc n*) + *conv-num c n*

fun *conv-denom* :: *cfrac* ⇒ *nat* ⇒ *int* **where**
  conv-denom c 0 = *1*
| *conv-denom c* (*Suc 0*) = *cfrac-nth c 1*
| *conv-denom c* (*Suc* (*Suc n*)) = *cfrac-nth c* (*Suc* (*Suc n*)) ∗ *conv-denom c* (*Suc n*) + *conv-denom c n*

lemma *conv-num-rec*:
  $n \geq 2 \Longrightarrow$ *conv-num c n* = *cfrac-nth c n* ∗ *conv-num c* (*n* − *1*) + *conv-num c* (*n* − *2*)
  by (*cases n*; *cases n* − *1*) *auto*

lemma *conv-denom-rec*:
  $n \geq 2 \Longrightarrow$ *conv-denom c n* = *cfrac-nth c n* ∗ *conv-denom c* (*n* − *1*) + *conv-denom c* (*n* − *2*)
  by (*cases n*; *cases n* − *1*) *auto*

fun *conv′* :: *cfrac* ⇒ *nat* ⇒ *real* ⇒ *real* **where**
  conv′ c 0 z = *z*
| *conv′ c* (*Suc n*) *z* = *conv′ c n* (*real-of-int* (*cfrac-nth c n*) + *1* / *z*)

Occasionally, it can be useful to extend the domain of *conv-num* and *conv-denom* to −*1* and −*2*.

definition *conv-num-int* :: *cfrac* ⇒ *int* ⇒ *int* **where**
  conv-num-int c n = (*if n* = −*1 then 1 else if n* < *0 then 0 else conv-num c* (*nat n*))

definition *conv-denom-int* :: *cfrac* ⇒ *int* ⇒ *int* **where**
  conv-denom-int c n = (*if n* = −*2 then 1 else if n* < *0 then 0 else conv-denom c* (*nat n*))

lemma *conv-num-int-rec*:
  assumes *n* ≥ *0*
  shows *conv-num-int c n* = *cfrac-nth c* (*nat n*) ∗ *conv-num-int c* (*n* − *1*) + *conv-num-int c* (*n* − *2*)
proof (*cases n* ≥ *2*)
  case *True*
  define *n′* **where** *n′* = *nat* (*n* − *2*)
  have *n*: *n* = *int* (*Suc* (*Suc n′*))
    using *True* **by** (*simp add*: *n′-def*)
  show *?thesis*
    by (*simp add*: *n conv-num-int-def nat-add-distrib*)
qed (*use assms* **in** ‹*auto simp*: *conv-num-int-def*›)

lemma *conv-denom-int-rec*:
  assumes *n* ≥ *0*
  shows *conv-denom-int c n* = *cfrac-nth c* (*nat n*) ∗ *conv-denom-int c* (*n* − *1*) + *conv-denom-int c* (*n* − *2*)
proof −
  consider *n* = *0* | *n* = *1* | *n* ≥ *2*
    using *assms* **by** *force*
  thus *?thesis*
  proof *cases*
    assume *n* ≥ *2*

define *n′* **where** *n′ = nat (n − 2)*
have *n*: *n = int (Suc (Suc n′))*
using ‹*n ≥ 2*› **by** (*simp add*: *n′-def*)
show *?thesis*
by (*simp add*: *n conv-denom-int-def nat-add-distrib*)
qed (*use assms* **in** ‹*auto simp*: *conv-denom-int-def*›)
qed

The number $[a_0;\ a_1,\ a_2,\ \ldots]$ that the infinite continued fraction converges to:

definition *cfrac-lim* :: *cfrac* ⇒ *real* **where**
cfrac-lim c =
(*case cfrac-length c of* ∞ ⇒ *lim* (*conv c*) | *enat l* ⇒ *conv c l*)

lemma *cfrac-lim-code* [*code*]:
cfrac-lim c =
(*case cfrac-length c of enat l* ⇒ *conv c l*
| - ⇒ *Code.abort* (*STR ″Cannot compute infinite continued fraction″*) (λ-. *cfrac-lim c*))
by (*simp add*: *cfrac-lim-def split*: *enat.splits*)

definition *cfrac-remainder* **where** *cfrac-remainder c n* = *cfrac-lim* (*cfrac-drop n c*)

lemmas *conv′-Suc-right* = *conv′.simps*(*2*)

lemma *conv′-Suc-left*:
assumes *z > 0*
shows *conv′ c* (*Suc n*) *z* =
real-of-int (*cfrac-nth c 0*) + *1* / *conv′* (*cfrac-tl c*) *n z*
using *assms*
proof (*induction n arbitrary*: *z*)
case (*Suc n z*)
have *conv′ c* (*Suc* (*Suc n*)) *z* =
conv′ c (*Suc n*) (*real-of-int* (*cfrac-nth c* (*Suc n*)) + *1* / *z*)
by *simp*
also have ... = *cfrac-nth c 0* + *1* / *conv′* (*cfrac-tl c*) (*Suc n*) *z*
using *Suc.prems* **by** (*subst Suc.IH*) (*auto intro*!: *add-nonneg-pos cfrac-nth-nonneg*)
finally show *?case* **.**
qed *simp-all*

lemmas [*simp del*] = *conv′.simps*(*2*)

lemma *conv′-left-induct*:
assumes ⋀*c. P c 0 z* ⋀*c n. P* (*cfrac-tl c*) *n z* ⟹ *P c* (*Suc n*) *z*
shows *P c n z*
using *assms* **by** (*rule conv.induct*)

lemma *enat-less-diff-conv* [*simp*]:

assumes $a = \infty \vee b < \infty \vee c < \infty$
shows $a < c - (b :: enat) \longleftrightarrow a + b < c$
using *assms* **by** (*cases a*; *cases b*; *cases c*) *auto*

lemma *conv-eq-conv'*: *conv c n* = *conv' c n* (*cfrac-nth c n*)
proof (*cases n* = *0*)
case *False*
hence *cfrac-nth c n* > *0* **by** (*auto intro*!: *cfrac-nth-pos*)
thus *?thesis*
by (*induction c n rule*: *conv.induct*) (*simp-all add*: *conv'-Suc-left*)
qed *simp-all*

lemma *conv-num-pos'*:
assumes *cfrac-nth c 0* > *0*
shows *conv-num c n* > *0*
using *assms* **by** (*induction n rule*: *fib.induct*) (*auto simp*: *intro*!: *add-pos-nonneg*)

lemma *conv-num-nonneg*: *cfrac-nth c 0* ≥ *0* ⟹ *conv-num c n* ≥ *0*
by (*induction c n rule*: *conv-num.induct*)
(*auto simp*: *intro*!: *mult-nonneg-nonneg add-nonneg-nonneg*
intro: *cfrac-nth-nonneg*)

lemma *conv-num-pos*:
cfrac-nth c 0 ≥ *0* ⟹ *n* > *0* ⟹ *conv-num c n* > *0*
by (*induction c n rule*: *conv-num.induct*)
(*auto intro*!: *mult-pos-pos mult-nonneg-nonneg add-pos-nonneg conv-num-nonneg cfrac-nth-pos*
intro: *cfrac-nth-nonneg simp*: *enat-le-iff*)

lemma *conv-denom-pos* [*simp*, *intro*]: *conv-denom c n* > *0*
by (*induction c n rule*: *conv-num.induct*)
(*auto intro*!: *add-nonneg-pos mult-nonneg-nonneg cfrac-nth-nonneg*
simp: *enat-le-iff*)

lemma *conv-denom-not-nonpos* [*simp*]: ¬*conv-denom c n* ≤ *0*
using *conv-denom-pos*[*of c n*] **by** *linarith*

lemma *conv-denom-not-neg* [*simp*]: ¬*conv-denom c n* < *0*
using *conv-denom-pos*[*of c n*] **by** *linarith*

lemma *conv-denom-nonzero* [*simp*]: *conv-denom c n* ≠ *0*
using *conv-denom-pos*[*of c n*] **by** *linarith*

lemma *conv-denom-nonneg* [*simp*, *intro*]: *conv-denom c n* ≥ *0*
using *conv-denom-pos*[*of c n*] **by** *linarith*

lemma *conv-num-int-neg1* [*simp*]: *conv-num-int c* (−*1*) = *1*
by (*simp add*: *conv-num-int-def*)

lemma *conv-num-int-neg* [*simp*]: $n < 0 \Longrightarrow n \neq -1 \Longrightarrow$ *conv-num-int c n* = *0*
  by (*simp add*: *conv-num-int-def*)

lemma *conv-num-int-of-nat* [*simp*]: *conv-num-int c* (*int n*) = *conv-num c n*
  by (*simp add*: *conv-num-int-def*)

lemma *conv-num-int-nonneg* [*simp*]: $n \geq 0 \Longrightarrow$ *conv-num-int c n* = *conv-num c* (*nat n*)
  by (*simp add*: *conv-num-int-def*)

lemma *conv-denom-int-neg2* [*simp*]: *conv-denom-int c* (−*2*) = *1*
  by (*simp add*: *conv-denom-int-def*)

lemma *conv-denom-int-neg* [*simp*]: $n < 0 \Longrightarrow n \neq -2 \Longrightarrow$ *conv-denom-int c n* = *0*
  by (*simp add*: *conv-denom-int-def*)

lemma *conv-denom-int-of-nat* [*simp*]: *conv-denom-int c* (*int n*) = *conv-denom c n*
  by (*simp add*: *conv-denom-int-def*)

lemma *conv-denom-int-nonneg* [*simp*]: $n \geq 0 \Longrightarrow$ *conv-denom-int c n* = *conv-denom c* (*nat n*)
  by (*simp add*: *conv-denom-int-def*)

lemmas *conv-Suc* [*simp del*] = *conv.simps*(*2*)

lemma *conv′-gt-1*:
  assumes *cfrac-nth c 0* > *0 x* > *1*
  shows *conv′ c n x* > *1*
  using *assms*
proof (*induction n arbitrary*: *c x*)
  case (*Suc n c x*)
  from *Suc.prems* **have** *pos*: *cfrac-nth c n* > *0* **using** *cfrac-nth-pos*[*of n c*]
    by (*cases n* = *0*) (*auto simp*: *enat-le-iff*)
  have *1* < *1* + *1* / *x*
    using *Suc.prems* **by** *simp*
  also have ... ≤ *cfrac-nth c n* + *1* / *x* **using** *pos*
    by (*intro add-right-mono*) (*auto simp*: *of-nat-ge-1-iff*)
  finally show *?case*
    by (*subst conv′-Suc-right*, *intro Suc.IH*)
      (*use Suc.prems* **in** ‹*auto simp*: *enat-le-iff*›)
qed *auto*

lemma *enat-eq-iff*: *a* = *enat b* ⟷ (∃ *a′*. *a* = *enat a′* ∧ *a′* = *b*)
  by (*cases a*) *auto*

lemma *eq-enat-iff*: *enat a* = *b* ⟷ (∃ *b′*. *b* = *enat b′* ∧ *a* = *b′*)
  by (*cases b*) *auto*

```
lemma enat-diff-one [simp]: enat a − 1 = enat (a − 1)
  by (cases enat (a − 1)) (auto simp flip: idiff-enat-enat)

lemma conv′-eqD:
  assumes conv′ c n x = conv′ c′ n x x > 1 m < n
  shows   cfrac-nth c m = cfrac-nth c′ m
  using assms
proof (induction n arbitrary: m c c′)
  case (Suc n m c c′)
  have gt: conv′ (cfrac-tl c) n x > 1 conv′ (cfrac-tl c′) n x > 1
    by (rule conv′-gt-1;
        use Suc.prems in ‹force intro: cfrac-nth-pos simp: enat-le-iff›)+
  have eq: cfrac-nth c 0 + 1 / conv′ (cfrac-tl c) n x =
            cfrac-nth c′ 0 + 1 / conv′ (cfrac-tl c′) n x
    using Suc.prems by (subst (asm) (1 2) conv′-Suc-left) auto
  hence ⌊cfrac-nth c 0 + 1 / conv′ (cfrac-tl c) n x⌋ =
         ⌊cfrac-nth c′ 0 + 1 / conv′ (cfrac-tl c′) n x⌋
    by (simp only: )
  also from gt have floor (cfrac-nth c 0 + 1 / conv′ (cfrac-tl c) n x) = cfrac-nth
c 0
    by (intro floor-unique) auto
  also from gt have floor (cfrac-nth c′ 0 + 1 / conv′ (cfrac-tl c′) n x) = cfrac-nth
c′ 0
    by (intro floor-unique) auto
  finally have [simp]: cfrac-nth c 0 = cfrac-nth c′ 0 by simp

  show ?case
  proof (cases m)
    case (Suc m′)
    from eq and gt have conv′ (cfrac-tl c) n x = conv′ (cfrac-tl c′) n x
      by simp
    hence cfrac-nth (cfrac-tl c) m′ = cfrac-nth (cfrac-tl c′) m′
      using Suc.prems
      by (intro Suc.IH[of cfrac-tl c cfrac-tl c′]) (auto simp: o-def Suc enat-le-iff)
    with Suc show ?thesis by simp
  qed simp-all
qed simp-all

context
  fixes c :: cfrac and h k
  defines h ≡ conv-num c and k ≡ conv-denom c
begin

lemma conv′-num-denom-aux:
  assumes z: z > 0
  shows   conv′ c (Suc (Suc n)) z ∗ (z ∗ k (Suc n) + k n) =
            (z ∗ h (Suc n) + h n)
  using z
```

proof (*induction n arbitrary*: *z*)
  case *0*
  hence *1* + *z* ∗ *cfrac-nth c 1* > *0*
    by (*intro add-pos-nonneg*) (*auto simp*: *cfrac-nth-nonneg*)
  with *0* **show** *?case*
    by (*auto simp add*: *h-def k-def field-simps conv′-Suc-right max-def not-le*)
next
  case (*Suc n*)
  have [*simp*]: *h* (*Suc* (*Suc n*)) = *cfrac-nth c* (*n+2*) ∗ *h* (*n+1*) + *h n*
    by (*simp add*: *h-def*)
  have [*simp*]: *k* (*Suc* (*Suc n*)) = *cfrac-nth c* (*n+2*) ∗ *k* (*n+1*) + *k n*
    by (*simp add*: *k-def*)
  define *z′* **where** *z′* = *cfrac-nth c* (*n+2*) + *1* / *z*
  from ‹*z* > *0*› **have** *z′* > *0*
    by (*auto simp*: *z′-def intro*!: *add-nonneg-pos cfrac-nth-nonneg*)

  have *z* ∗ *real-of-int* (*h* (*Suc* (*Suc n*))) + *real-of-int* (*h* (*Suc n*)) =
        z ∗ (*z′* ∗ *h* (*Suc n*) + *h n*)
    using ‹*z* > *0*› **by** (*simp add*: *algebra-simps z′-def*)
  also have . . . = *z* ∗ (*conv′ c* (*Suc* (*Suc n*)) *z′* ∗ (*z′* ∗ *k* (*Suc n*) + *k n*))
    using ‹*z′* > *0*› **by** (*subst Suc.IH* [*symmetric*]) *auto*
  also have . . . = *conv′ c* (*Suc* (*Suc* (*Suc n*))) *z* ∗
          (*z* ∗ *k* (*Suc* (*Suc n*)) + *k* (*Suc n*))
    unfolding *z′-def* **using** ‹*z* > *0*›
    by (*subst* (*2*) *conv′-Suc-right*) (*simp add*: *algebra-simps*)
  finally show *?case* **..**
qed

lemma *conv′-num-denom*:
  assumes *z* > *0*
  shows   *conv′ c* (*Suc* (*Suc n*)) *z* =
          (*z* ∗ *h* (*Suc n*) + *h n*) / (*z* ∗ *k* (*Suc n*) + *k n*)
proof −
  have *z* ∗ *real-of-int* (*k* (*Suc n*)) + *real-of-int* (*k n*) > *0*
    using *assms* **by** (*intro add-pos-nonneg mult-pos-pos*) (*auto simp*: *k-def*)
  with *conv′-num-denom-aux*[*of z n*] *assms* **show** *?thesis*
    by (*simp add*: *divide-simps*)
qed

lemma *conv-num-denom*: *conv c n* = *h n* / *k n*
proof −
  consider *n* = *0* | *n* = *Suc 0* | *m* **where** *n* = *Suc* (*Suc m*)
    using *not0-implies-Suc* **by** *blast*
  thus *?thesis*
  proof *cases*
    assume *n* = *Suc 0*
    thus *?thesis*
      by (*auto simp*: *h-def k-def field-simps max-def conv-Suc*)
  next

fix *m* **assume** [*simp*]: *n* = *Suc* (*Suc m*)
have *conv c n* = *conv′ c* (*Suc* (*Suc m*)) (*cfrac-nth c* (*Suc* (*Suc m*)))
by (*subst conv-eq-conv′*) *simp-all*
also have ... = *h n* / *k n*
by (*subst conv′-num-denom*) (*simp-all add*: *h-def k-def*)
finally show *?thesis* **.**
qed (*auto simp*: *h-def k-def*)
qed

lemma *conv′-num-denom′*:
assumes *z* > *0* **and** *n* ≥ *2*
shows *conv′ c n z* = (*z* ∗ *h* (*n* − *1*) + *h* (*n* − *2*)) / (*z* ∗ *k* (*n* − *1*) + *k* (*n* − *2*))
using *assms conv′-num-denom*[*of z n* − *2*]
by (*auto simp*: *eval-nat-numeral Suc-diff-Suc*)

lemma *conv′-num-denom-int*:
assumes *z* > *0*
shows *conv′ c n z* =
(*z* ∗ *conv-num-int c* (*int n* − *1*) + *conv-num-int c* (*int n* − *2*)) /
(*z* ∗ *conv-denom-int c* (*int n* − *1*) + *conv-denom-int c* (*int n* − *2*))
proof −
consider *n* = *0* | *n* = *1* | *n* ≥ *2* **by** *force*
thus *?thesis*
proof *cases*
case *1*
thus *?thesis* **using** *conv-num-int-neg1* **by** *auto*
next
case *2*
thus *?thesis* **using** *assms* **by** (*auto simp*: *conv′-Suc-right field-simps*)
next
case *3*
thus *?thesis* **using** *conv′-num-denom′*[*OF assms*(*1*), *of nat n*]
by (*auto simp*: *nat-diff-distrib h-def k-def*)
qed
qed

lemma *conv-nonneg*: *cfrac-nth c 0* ≥ *0* ⟹ *conv c n* ≥ *0*
by (*subst conv-num-denom*)
(*auto intro*!: *divide-nonneg-nonneg conv-num-nonneg simp*: *h-def k-def*)

lemma *conv-pos*:
assumes *cfrac-nth c 0* > *0*
shows *conv c n* > *0*
proof −
have *conv c n* = *h n* / *k n*
using *assms* **by** (*intro conv-num-denom*)
also from *assms* **have** ... > *0* **unfolding** *h-def k-def*
by (*intro divide-pos-pos*) (*auto intro*!: *conv-num-pos′*)

```
  finally show ?thesis .
qed

lemma conv-num-denom-prod-diff:
  k n ∗ h (Suc n) − k (Suc n) ∗ h n = (−1) ^ n
  by (induction c n rule: conv-num.induct)
     (auto simp: k-def h-def algebra-simps)

lemma conv-num-denom-prod-diff':
  k (Suc n) ∗ h n − k n ∗ h (Suc n) = (−1) ^ Suc n
  by (induction c n rule: conv-num.induct)
     (auto simp: k-def h-def algebra-simps)

lemma
  fixes n :: int
  assumes n ≥ −2
  shows   conv-num-denom-int-prod-diff:
            conv-denom-int c n ∗ conv-num-int c (n + 1) −
              conv-denom-int c (n + 1) ∗ conv-num-int c n = (−1) ^ (nat (n + 2))
(is ?th1)
  and     conv-num-denom-int-prod-diff':
            conv-denom-int c (n + 1) ∗ conv-num-int c n −
              conv-denom-int c n ∗ conv-num-int c (n + 1) = (−1) ^ (nat (n + 3))
(is ?th2)
proof −
  from assms consider n = −2 | n = −1 | n ≥ 0 by force
  thus ?th1 using conv-num-denom-prod-diff[of nat n]
    by cases (auto simp: h-def k-def nat-add-distrib)
  moreover from assms have nat (n + 3) = Suc (nat (n + 2)) by (simp add:
nat-add-distrib)
  ultimately show ?th2 by simp
qed

lemma coprime-conv-num-denom: coprime (h n) (k n)
proof (cases n)
  case [simp]: (Suc m)
  {
    fix d :: int
    assume d dvd h n and d dvd k n
    hence abs d dvd abs (k n ∗ h (Suc n) − k (Suc n) ∗ h n)
      by simp
    also have ... = 1
      by (subst conv-num-denom-prod-diff) auto
    finally have is-unit d by simp
  }
  thus ?thesis by (rule coprimeI)
qed (auto simp: h-def k-def)

lemma coprime-conv-num-denom-int:
```

```
  assumes n ≥ −2
  shows   coprime (conv-num-int c n) (conv-denom-int c n)
proof −
  from assms consider n = −2 | n = −1 | n ≥ 0 by force
  thus ?thesis by cases (insert coprime-conv-num-denom[of nat n], auto simp: h-def
k-def)
qed

lemma mono-conv-num:
  assumes cfrac-nth c 0 ≥ 0
  shows   mono h
proof (rule incseq-SucI)
  show h n ≤ h (Suc n) for n
  proof (cases n)
    case 0
    have 1 ∗ cfrac-nth c 0 + 1 ≤ cfrac-nth c (Suc 0) ∗ cfrac-nth c 0 + 1
      using assms by (intro add-mono mult-right-mono) auto
    thus ?thesis using assms by (simp add: le-Suc-eq Suc-le-eq h-def 0)
  next
    case (Suc m)
    have 1 ∗ h (Suc m) + 0 ≤ cfrac-nth c (Suc (Suc m)) ∗ h (Suc m) + h m
      using assms
      by (intro add-mono mult-right-mono)
         (auto simp: Suc-le-eq h-def intro!: conv-num-nonneg)
    with Suc show ?thesis by (simp add: h-def)
  qed
qed

lemma mono-conv-denom: mono k
proof (rule incseq-SucI)
  show k n ≤ k (Suc n) for n
  proof (cases n)
    case 0
    thus ?thesis by (simp add: le-Suc-eq Suc-le-eq k-def)
  next
    case (Suc m)
    have 1 ∗ k (Suc m) + 0 ≤ cfrac-nth c (Suc (Suc m)) ∗ k (Suc m) + k m
      by (intro add-mono mult-right-mono) (auto simp: Suc-le-eq k-def)
    with Suc show ?thesis by (simp add: k-def)
  qed
qed

lemma conv-num-leI: cfrac-nth c 0 ≥ 0 ⟹ m ≤ n ⟹ h m ≤ h n
  using mono-conv-num by (auto simp: mono-def)

lemma conv-denom-leI: m ≤ n ⟹ k m ≤ k n
  using mono-conv-denom by (auto simp: mono-def)

lemma conv-denom-lessI:
```

```
  assumes m < n 1 < n
  shows   k m < k n
proof (cases n)
  case [simp]: (Suc n')
  show ?thesis
  proof (cases n')
    case [simp]: (Suc n'')
    from assms have k m ≤ 1 ∗ k n' + 0
      by (auto intro: conv-denom-leI simp: less-Suc-eq)
    also have . . . ≤ cfrac-nth c n ∗ k n' + 0
      using assms by (intro add-mono mult-mono) (auto simp: Suc-le-eq k-def)
    also have . . . < cfrac-nth c n ∗ k n' + k n'' unfolding k-def
      by (intro add-strict-left-mono conv-denom-pos assms)
    also have . . . = k n by (simp add: k-def)
    finally show ?thesis .
  qed (insert assms, auto simp: k-def)
qed (insert assms, auto)

lemma conv-num-lower-bound:
  assumes cfrac-nth c 0 ≥ 0
  shows   h n ≥ fib n unfolding h-def
  using assms
proof (induction c n rule: conv-denom.induct)
  case (3 c n)
  hence conv-num c (Suc (Suc n)) ≥ 1 ∗ int (fib (Suc n)) + int (fib n)
    using 3.prems unfolding conv-num.simps
    by (intro add-mono mult-mono 3.IH) auto
  thus ?case by simp
qed auto

lemma conv-denom-lower-bound: k n ≥ fib (Suc n)
  unfolding k-def
proof (induction c n rule: conv-denom.induct)
  case (3 c n)
  hence conv-denom c (Suc (Suc n)) ≥ 1 ∗ int (fib (Suc (Suc n))) + int (fib (Suc
n))
    using 3.prems unfolding conv-denom.simps
    by (intro add-mono mult-mono 3.IH) auto
  thus ?case by simp
qed (auto simp: Suc-le-eq)

lemma conv-diff-eq: conv c (Suc n) − conv c n = (−1) ^ n / (k n ∗ k (Suc n))
proof −
  have pos: k n > 0 k (Suc n) > 0 unfolding k-def
    by (intro conv-denom-pos)+
  have conv c (Suc n) − conv c n =
          (k n ∗ h (Suc n) − k (Suc n) ∗ h n) / (k n ∗ k (Suc n))
     using pos by (subst (1 2) conv-num-denom) (simp add: conv-num-denom
field-simps)
```

also have *k n* ∗ *h* (*Suc n*) − *k* (*Suc n*) ∗ *h n* = (−*1*) ^ *n*
  by (*rule conv-num-denom-prod-diff*)
finally show *?thesis* **by** *simp*
qed

lemma *conv-telescope*:
  assumes *m* ≤ *n*
  shows *conv c m* + (∑ *i*=*m*..<*n*. (−*1*) ^ *i* / (*k i* ∗ *k* (*Suc i*))) = *conv c n*
proof −
  have (∑ *i*=*m*..<*n*. (−*1*) ^ *i* / (*k i* ∗ *k* (*Suc i*))) =
      (∑ *i*=*m*..<*n*. *conv c* (*Suc i*) − *conv c i*)
    by (*simp add*: *conv-diff-eq assms del*: *conv.simps*)
  also have *conv c m* + ... = *conv c n*
    using *assms* **by** (*induction rule*: *dec-induct*) *simp-all*
  finally show *?thesis* **.**
qed

lemma *fib-at-top*: *filterlim fib at-top at-top*
proof (*rule filterlim-at-top-mono*)
  show *eventually* (λ*n*. *fib n* ≥ *n* − *1*) *at-top*
    by (*intro always-eventually fib-ge allI*)
  show *filterlim* (λ*n*::*nat*. *n* − *1*) *at-top at-top*
    by (*subst filterlim-sequentially-Suc* [*symmetric*])
      (*simp-all add*: *filterlim-ident*)
qed

lemma *conv-denom-at-top*: *filterlim k at-top at-top*
proof (*rule filterlim-at-top-mono*)
  show *filterlim* (λ*n*. *int* (*fib* (*Suc n*))) *at-top at-top*
    by (*rule filterlim-compose*[*OF filterlim-int-sequentially*])
      (*simp add*: *fib-at-top filterlim-sequentially-Suc*)
  show *eventually* (λ*n*. *fib* (*Suc n*) ≤ *k n*) *at-top*
    by (*intro always-eventually conv-denom-lower-bound allI*)
qed

lemma
  shows *summable-conv-telescope*:
      summable (λ*i*. (−*1*) ^ *i* / (*k i* ∗ *k* (*Suc i*))) (**is** *?th1*)
    and *cfrac-remainder-bounds*:
      |(∑ *i*. (−*1*) ^ (*i* + *m*) / (*k* (*i* + *m*) ∗ *k* (*Suc i* + *m*)))| ∈
        {*1*/(*k m* ∗ (*k m* + *k* (*Suc m*))) <..< *1* / (*k m* ∗ *k* (*Suc m*))} (**is** *?th2*)
proof −
  have [*simp*]: *k n* > *0 k n* ≥ *0* ¬*k n* = *0* **for** *n*
    by (*auto simp*: *k-def*)
  have *k-rec*: *k* (*Suc* (*Suc n*)) = *cfrac-nth c* (*Suc* (*Suc n*)) ∗ *k* (*Suc n*) + *k n* **for** *n*
    by (*simp add*: *k-def*)
  have [*simp*]: *a* + *b* = *0* ⟷ *a* = *0* ∧ *b* = *0* **if** *a* ≥ *0 b* ≥ *0* **for** *a b* :: *real*
    using *that* **by** *linarith*

define *g* **where** *g* = (λ*i*. *inverse* (*real-of-int* (*k i* ∗ *k* (*Suc i*))))

{
fix *m* :: *nat*
have *filterlim* (λ*n*. *k n*) *at-top at-top* **and** *filterlim* (λ*n*. *k* (*Suc n*)) *at-top at-top*
by (*force simp*: *filterlim-sequentially-Suc intro*: *conv-denom-at-top*)+
hence *lim*: *g* ⟶ *0*
unfolding *g-def of-int-mult*
by (*intro tendsto-inverse-0-at-top filterlim-at-top-mult-at-top*
filterlim-compose[*OF filterlim-real-of-int-at-top*])
from *lim* **have** *A*: *summable* (λ*n*. (−*1*) ^ (*n* + *m*) ∗ *g* (*n* + *m*)) **unfolding** *g-def*
by (*intro summable-alternating-decreasing*)
(*auto intro*!: *conv-denom-leI mult-nonneg-nonneg*)

have *1* / (*k m* ∗ (*real-of-int* (*k* (*Suc m*)) + *k m* / *1*)) ≤
1 / (*k m* ∗ (*k* (*Suc m*) + *k m* / *cfrac-nth c* (*m*+*2*)))
by (*intro divide-left-mono mult-left-mono add-left-mono mult-pos-pos add-pos-pos divide-pos-pos*)
(*auto simp*: *of-nat-ge-1-iff*)
also have ... = *g m* − *g* (*Suc m*)
by (*simp add*: *g-def k-rec field-simps add-pos-pos*)
finally have *le*: *1* / (*k m* ∗ (*real-of-int* (*k* (*Suc m*)) + *k m* / *1*)) ≤ *g m* − *g* (*Suc m*) **by** *simp*
have ∗: |(∑*i*. (−*1*) ^ (*i* + *m*) ∗ *g* (*i* + *m*))| ∈ {*g m* − *g* (*Suc m*) <..< *g m*}
using *lim* **unfolding** *g-def*
by (*intro abs-alternating-decreasing-suminf-strict*) (*auto intro*!: *conv-denom-lessI*)
also from *le* **have** ... ⊆ {*1* / (*k m* ∗ (*k* (*Suc m*) + *k m*)) <..< *g m*}
by (*subst greaterThanLessThan-subseteq-greaterThanLessThan*) *auto*
finally have *B*: |∑*i*. (− *1*) ^ (*i* + *m*) ∗ *g* (*i* + *m*)| ∈ ... **.**
note *A B*
} **note** *AB* = *this*

from *AB*(*1*)[*of 0*] **show** *?th1* **by** (*simp add*: *field-simps g-def*)
from *AB*(*2*)[*of m*] **show** *?th2* **by** (*simp add*: *g-def divide-inverse add-ac*)
qed

lemma *convergent-conv*: *convergent* (*conv c*)
proof −
have *convergent* (λ*n*. *conv c 0* + (∑*i*<*n*. (−*1*) ^ *i* / (*k i* ∗ *k* (*Suc i*))))
using *summable-conv-telescope*
by (*intro convergent-add convergent-const*)
(*simp-all add*: *summable-iff-convergent*)
also have ... = *conv c*
by (*rule ext*, *subst* (*2*) *conv-telescope* [*of 0*, *symmetric*]) (*simp-all add*: *atLeast0LessThan*)
finally show *?thesis* **.**
qed

lemma *LIMSEQ-cfrac-lim*: *cfrac-length c* = ∞ ⟹ *conv c* ⟶ *cfrac-lim c*

using *convergent-conv* **by** (*auto simp*: *convergent-LIMSEQ-iff cfrac-lim-def*)

lemma *cfrac-lim-nonneg*:
assumes *cfrac-nth c 0* $\geq$ *0*
shows *cfrac-lim c* $\geq$ *0*
proof (*cases cfrac-length c*)
case *infinity*
have *conv c* $\longrightarrow$ *cfrac-lim c*
by (*rule LIMSEQ-cfrac-lim*) *fact*
thus *?thesis*
by (*rule tendsto-lowerbound*)
(*auto intro*!: *conv-nonneg always-eventually assms*)
next
case (*enat l*)
thus *?thesis* **using** *assms*
by (*auto simp*: *cfrac-lim-def conv-nonneg*)
qed

lemma *sums-cfrac-lim-minus-conv*:
assumes *cfrac-length c* = ∞
shows (λ*i*. ($-$*1*) ^ (*i* + *m*) / (*k* (*i* + *m*) $*$ *k* (*Suc i* + *m*))) *sums* (*cfrac-lim c* $-$ *conv c m*)
proof $-$
have (λ*n*. *conv c* (*n* + *m*) $-$ *conv c m*) $\longrightarrow$ *cfrac-lim c* $-$ *conv c m*
by (*auto intro*!: *tendsto-diff LIMSEQ-cfrac-lim simp*: *filterlim-sequentially-shift assms*)
also have (λ*n*. *conv c* (*n* + *m*) $-$ *conv c m*) =
(λ*n*. ($\sum$ *i*=*0* + *m*..<*n* + *m*. ($-$*1*) ^ *i* / (*k i* $*$ *k* (*Suc i*))))
by (*subst conv-telescope* [*of m*, *symmetric*]) *simp-all*
also have ... = (λ*n*. ($\sum$ *i*<*n*. ($-$*1*) ^ (*i* + *m*) / (*k* (*i* + *m*) $*$ *k* (*Suc i* + *m*))))
by (*subst sum.shift-bounds-nat-ivl*) (*simp-all add*: *atLeast0LessThan*)
finally show *?thesis* **unfolding** *sums-def* **.**
qed

lemma *cfrac-lim-minus-conv-upper-bound*:
assumes *m* $\leq$ *cfrac-length c*
shows $|$*cfrac-lim c* $-$ *conv c m*$|$ $\leq$ *1* / (*k m* $*$ *k* (*Suc m*))
proof (*cases cfrac-length c*)
case *infinity*
have *cfrac-lim c* $-$ *conv c m* = ($\sum$ *i*. ($-$*1*) ^ (*i* + *m*) / (*k* (*i* + *m*) $*$ *k* (*Suc i* + *m*)))
using *sums-cfrac-lim-minus-conv infinity* **by** (*simp add*: *sums-iff*)
also note *cfrac-remainder-bounds*[*of m*]
finally show *?thesis* **by** *simp*
next
case [*simp*]: (*enat l*)
show *?thesis*
proof (*cases l* = *m*)
case *True*

thus *?thesis* **by** (*auto simp*: *cfrac-lim-def k-def*)
next
case *False*
let *?S* = ($\sum$ *i*=*m*..<*l*. (−*1*) ^ *i* ∗ (*1* / *real-of-int* (*k i* ∗ *k* (*Suc i*))))
have [*simp*]: *k n* ≥ *0 k n* > *0* **for** *n*
by (*simp-all add*: *k-def*)
hence *cfrac-lim c* − *conv c m* = *conv c l* − *conv c m*
by (*simp add*: *cfrac-lim-def*)
also have . . . = *?S*
using *assms* **by** (*subst conv-telescope* [*symmetric*, *of m*]) *auto*
finally have *cfrac-lim c* − *conv c m* = *?S* **.**
moreover have |*?S*| ≤ *1* / *real-of-int* (*k m* ∗ *k* (*Suc m*))
unfolding *of-int-mult* **using** *assms False*
by (*intro abs-alternating-decreasing-sum-upper-bound′ divide-nonneg-nonneg frac-le mult-mono*)
(*simp-all add*: *conv-denom-leI del*: *conv-denom.simps*)
ultimately show *?thesis* **by** *simp*
qed
qed

lemma *cfrac-lim-minus-conv-lower-bound*:
assumes *m* < *cfrac-length c*
shows |*cfrac-lim c* − *conv c m*| ≥ *1* / (*k m* ∗ (*k m* + *k* (*Suc m*)))
proof (*cases cfrac-length c*)
case *infinity*
have *cfrac-lim c* − *conv c m* = ($\sum$ *i*. (−*1*) ^(*i* + *m*) / (*k* (*i* + *m*) ∗ *k* (*Suc i* + *m*)))
using *sums-cfrac-lim-minus-conv infinity* **by** (*simp add*: *sums-iff*)
also note *cfrac-remainder-bounds*[*of m*]
finally show *?thesis* **by** *simp*
next
case [*simp*]: (*enat l*)
let *?S* = ($\sum$ *i*=*m*..<*l*. (−*1*) ^ *i* ∗ (*1* / *real-of-int* (*k i* ∗ *k* (*Suc i*))))
have [*simp*]: *k n* ≥ *0 k n* > *0* **for** *n*
by (*simp-all add*: *k-def*)
hence *cfrac-lim c* − *conv c m* = *conv c l* − *conv c m*
by (*simp add*: *cfrac-lim-def*)
also have . . . = *?S*
using *assms* **by** (*subst conv-telescope* [*symmetric*, *of m*]) (*auto simp*: *split*: *enat.splits*)
finally have *cfrac-lim c* − *conv c m* = *?S* **.**

moreover have |*?S*| ≥ *1* / (*k m* ∗ (*k m* + *k* (*Suc m*)))
proof (*cases m* < *cfrac-length c* − *1*)
case *False*
hence [*simp*]: *m* = *l* − *1* **and** *l* > *0* **using** *assms*
by (*auto simp*: *not-less*)
have *1* / (*k m* ∗ (*k m* + *k* (*Suc m*))) ≤ *1* / (*k m* ∗ *k* (*Suc m*))
unfolding *of-int-mult*

by (*intro divide-left-mono mult-mono mult-pos-pos*) (*auto intro*!: *add-pos-pos*)
also from ‹*l* > *0*› **have** {*m*..<*l*} = {*m*} **by** *auto*
hence *1* / (*k m* ∗ *k* (*Suc m*)) = |*?S*|
by *simp*
finally show *?thesis* **.**
next
case *True*
with *assms* **have** *less*: *m* < *l* − *1*
by *auto*
have *k m* + *k* (*Suc m*) > *0*
by (*intro add-pos-pos*) (*auto simp*: *k-def*)
hence *1* / (*k m* ∗ (*k m* + *k* (*Suc m*))) ≤ *1* / (*k m* ∗ *k* (*Suc m*)) − *1* / (*k* (*Suc m*) ∗ *k* (*Suc* (*Suc m*)))
by (*simp add*: *divide-simps*) (*auto simp*: *k-def algebra-simps*)
also have ... ≤ |*?S*|
unfolding *of-int-mult* **using** *less*
by (*intro abs-alternating-decreasing-sum-lower-bound′ divide-nonneg-nonneg frac-le mult-mono*)
(*simp-all add*: *conv-denom-leI del*: *conv-denom.simps*)
finally show *?thesis* **.**
qed
ultimately show *?thesis* **by** *simp*
qed

lemma *cfrac-lim-minus-conv-bounds*:
assumes *m* < *cfrac-length c*
shows |*cfrac-lim c* − *conv c m*| ∈ {*1* / (*k m* ∗ (*k m* + *k* (*Suc m*)))..*1* / (*k m* ∗ *k* (*Suc m*))}
using *cfrac-lim-minus-conv-lower-bound*[*of m*] *cfrac-lim-minus-conv-upper-bound*[*of m*] *assms*
by *auto*

end

lemma *conv-pos′*:
assumes *n* > *0 cfrac-nth c 0* ≥ *0*
shows *conv c n* > *0*
using *assms* **by** (*cases n*) (*auto simp*: *conv-Suc intro*!: *add-nonneg-pos conv-pos*)

lemma *conv-in-Rats* [*intro*]: *conv c n* ∈ ℚ
by (*induction c n rule*: *conv.induct*) (*auto simp*: *conv-Suc o-def*)

lemma
assumes *0* < *z1 z1* ≤ *z2*
shows *conv′-even-mono*: *even n* ⟹ *conv′ c n z1* ≤ *conv′ c n z2*
and *conv′-odd-mono*: *odd n* ⟹ *conv′ c n z1* ≥ *conv′ c n z2*
proof −
let *?P* = (λ*n* (*f*::*nat*⇒*real*⇒*real*).

```
              if even n then f n z1 ≤ f n z2 else f n z1 ≥ f n z2)
  have ?P n (conv' c) using assms
  proof (induction n arbitrary: z1 z2)
    case (Suc n)
    note z12 = Suc.prems
    consider n = 0 | even n n > 0 | odd n by force
    thus ?case
    proof cases
      assume n = 0
      thus ?thesis using Suc by (simp add: conv'-Suc-right field-simps)
    next
      assume n: even n n > 0
      with Suc.IH have IH: conv' c n z1 ≤ conv' c n z2
        if 0 < z1 z1 ≤ z2 for z1 z2 using that by auto
      show ?thesis using Suc.prems n z12
            by (auto simp: conv'-Suc-right field-simps intro!: IH add-pos-nonneg
mult-nonneg-nonneg)
    next
      assume n: odd n
      hence [simp]: n > 0 by (auto intro!: Nat.gr0I)
      from n and Suc.IH have IH: conv' c n z1 ≥ conv' c n z2
        if 0 < z1 z1 ≤ z2 for z1 z2 using that by auto
      show ?thesis using Suc.prems n
        by (auto simp: conv'-Suc-right field-simps
                intro!: IH add-pos-nonneg mult-nonneg-nonneg)
    qed
  qed auto
  thus even n ⟹ conv' c n z1 ≤ conv' c n z2
       odd n ⟹ conv' c n z1 ≥ conv' c n z2 by auto
qed

lemma
  shows   conv-even-mono: even n ⟹ n ≤ m ⟹ conv c n ≤ conv c m
    and   conv-odd-mono:  odd n  ⟹ n ≤ m ⟹ conv c n ≥ conv c m
proof –
  assume even n
  have A: conv c n ≤ conv c (Suc (Suc n)) if even n for n
  proof (cases n = 0)
    case False
    with ‹even n› show ?thesis
      by (auto simp add: conv-eq-conv' conv'-Suc-right intro: conv'-even-mono)
  qed (auto simp:  conv-Suc)

  have B: conv c n ≤ conv c (Suc n) if even n for n
  proof (cases n = 0)
    case False
    with ‹even n› show ?thesis
      by (auto simp add: conv-eq-conv' conv'-Suc-right intro: conv'-even-mono)
  qed (auto simp:  conv-Suc)
```

```
  show conv c n ≤ conv c m if n ≤ m for m
    using that
  proof (induction m rule: less-induct)
    case (less m)
    from ‹n ≤ m› consider m = n | even m m > n | odd m m > n
      by force
    thus ?case
    proof cases
      assume m: even m m > n
      with ‹even n› have m': m − 2 ≥ n by presburger
      with m have conv c n ≤ conv c (m − 2)
        by (intro less.IH) auto
      also have ... ≤ conv c (Suc (Suc (m − 2)))
        using m m' by (intro A) auto
      also have Suc (Suc (m − 2)) = m
        using m by presburger
      finally show ?thesis .
    next
      assume m: odd m m > n
      hence conv c n ≤ conv c (m − 1)
        by (intro less.IH) auto
      also have ... ≤ conv c (Suc (m − 1))
        using m by (intro B) auto
      also have Suc (m − 1) = m
        using m by simp
      finally show ?thesis .
    qed simp-all
  qed
next
  assume odd n
  have A: conv c n ≥ conv c (Suc (Suc n)) if odd n for n
    using that
   by (auto simp add: conv-eq-conv' conv'-Suc-right odd-pos intro!: conv'-odd-mono)
  have B: conv c n ≥ conv c (Suc n) if odd n for n using that
   by (auto simp add: conv-eq-conv' conv'-Suc-right odd-pos intro!: conv'-odd-mono)

  show conv c n ≥ conv c m if n ≤ m for m
    using that
  proof (induction m rule: less-induct)
    case (less m)
    from ‹n ≤ m› consider m = n | even m m > n | odd m m > n
      by force
    thus ?case
    proof cases
      assume m: odd m m > n
      with ‹odd n› have m': m − 2 ≥ n m ≥ 2 by presburger+
      from m and ‹odd n› have m = Suc (Suc (m − 2)) by presburger
      also have conv c ... ≤ conv c (m − 2)
```

using *m m'* **by** (*intro A*) *auto*
also have ... ≤ *conv c n*
using *m m'* **by** (*intro less.IH*) *auto*
finally show *?thesis* **.**
next
assume *m*: *even m m* > *n*
from *m* **have** *m* = *Suc* (*m* − *1*) **by** *presburger*
also have *conv c* ... ≤ *conv c* (*m* − *1*)
using *m* **by** (*intro B*) *auto*
also have ... ≤ *conv c n*
using *m* **by** (*intro less.IH*) *auto*
finally show *?thesis* **.**
qed *simp-all*
qed
qed

lemma
assumes *m* ≤ *cfrac-length c*
shows *conv-le-cfrac-lim*: *even m* ⟹ *conv c m* ≤ *cfrac-lim c*
and *conv-ge-cfrac-lim*: *odd m* ⟹ *conv c m* ≥ *cfrac-lim c*
proof −
have *if even m then conv c m* ≤ *cfrac-lim c else conv c m* ≥ *cfrac-lim c*
proof (*cases cfrac-length c*)
case [*simp*]: *infinity*
show *?thesis*
proof (*cases even m*)
case *True*
have *eventually* (λ*i. conv c m* ≤ *conv c i*) *at-top*
using *eventually-ge-at-top*[*of m*] **by** *eventually-elim* (*rule conv-even-mono*[*OF True*])
hence *conv c m* ≤ *cfrac-lim c*
by (*intro tendsto-lowerbound*[*OF LIMSEQ-cfrac-lim*]) *auto*
thus *?thesis* **using** *True* **by** *simp*
next
case *False*
have *eventually* (λ*i. conv c m* ≥ *conv c i*) *at-top*
using *eventually-ge-at-top*[*of m*] **by** *eventually-elim* (*rule conv-odd-mono*[*OF False*])
hence *conv c m* ≥ *cfrac-lim c*
by (*intro tendsto-upperbound*[*OF LIMSEQ-cfrac-lim*]) *auto*
thus *?thesis* **using** *False* **by** *simp*
qed
next
case [*simp*]: (*enat l*)
show *?thesis*
using *conv-even-mono*[*of m l c*] *conv-odd-mono*[*of m l c*] *assms*
by (*auto simp*: *cfrac-lim-def*)
qed
thus *even m* ⟹ *conv c m* ≤ *cfrac-lim c* **and** *odd m* ⟹ *conv c m* ≥ *cfrac-lim c*

```
    by auto
qed

lemma cfrac-lim-ge-first: cfrac-lim c ≥ cfrac-nth c 0
  using conv-le-cfrac-lim[of 0 c] by (auto simp: less-eq-enat-def split: enat.splits)

lemma cfrac-lim-pos: cfrac-nth c 0 > 0 ⟹ cfrac-lim c > 0
  by (rule less-le-trans[OF - cfrac-lim-ge-first]) auto

lemma conv'-eq-iff:
  assumes 0 ≤ z1 ∨ 0 ≤ z2
  shows   conv' c n z1 = conv' c n z2 ⟷ z1 = z2
proof
  assume conv' c n z1 = conv' c n z2
  thus z1 = z2 using assms
  proof (induction n arbitrary: z1 z2)
    case (Suc n)
    show ?case
    proof (cases n = 0)
      case True
      thus ?thesis using Suc by (auto simp: conv'-Suc-right)
    next
      case False
      have conv' c n (real-of-int (cfrac-nth c n) + 1 / z1) =
              conv' c n (real-of-int (cfrac-nth c n) + 1 / z2) using Suc.prems
        by (simp add: conv'-Suc-right)
      hence real-of-int (cfrac-nth c n) + 1 / z1 = real-of-int (cfrac-nth c n) + 1 /
z2
        by (rule Suc.IH)
          (insert Suc.prems False, auto intro!: add-nonneg-pos add-nonneg-nonneg)
      with Suc.prems show z1 = z2 by simp
    qed
  qed auto
qed auto

lemma conv-even-mono-strict:
  assumes even n n < m
  shows   conv c n < conv c m
proof (cases m = n + 1)
  case [simp]: True
  show ?thesis
  proof (cases n = 0)
    case True
    thus ?thesis using assms by (auto simp: conv-Suc)
  next
    case False
    hence conv' c n (real-of-int (cfrac-nth c n)) ≠
            conv' c n (real-of-int (cfrac-nth c n) + 1 / real-of-int (cfrac-nth c (Suc
n)))
```

```
      by (subst conv'-eq-iff) auto
    with assms have conv c n ≠ conv c m
      by (auto simp: conv-eq-conv' conv'-eq-iff conv'-Suc-right field-simps)
    moreover from assms have conv c n ≤ conv c m
      by (intro conv-even-mono) auto

    ultimately show ?thesis by simp
  qed
next
  case False
  show ?thesis
  proof (cases n = 0)
    case True
    thus ?thesis using assms
      by (cases m) (auto simp: conv-Suc conv-pos)
  next
    case False
    have 1 + real-of-int (cfrac-nth c (n+1)) ∗ cfrac-nth c (n+2) > 0
      by (intro add-pos-nonneg) auto
    with assms have conv c n ≠ conv c (Suc (Suc n))
      unfolding conv-eq-conv' conv'-Suc-right using False
      by (subst conv'-eq-iff) (auto simp: field-simps)
    moreover from assms have conv c n ≤ conv c (Suc (Suc n))
      by (intro conv-even-mono) auto
    ultimately have conv c n < conv c (Suc (Suc n)) by simp
    also have ... ≤ conv c m using assms ‹m ≠ n + 1›
      by (intro conv-even-mono) auto
    finally show ?thesis .
  qed
qed

lemma conv-odd-mono-strict:
  assumes odd n n < m
  shows   conv c n > conv c m
proof (cases m = n + 1)
  case [simp]: True
  from assms have n > 0 by (intro Nat.gr0I) auto
  hence conv' c n (real-of-int (cfrac-nth c n)) ≠
        conv' c n (real-of-int (cfrac-nth c n) + 1 / real-of-int (cfrac-nth c (Suc n)))
    by (subst conv'-eq-iff) auto
  hence conv c n ≠ conv c m
    by (simp add: conv-eq-conv' conv'-Suc-right)
  moreover from assms have conv c n ≥ conv c m
    by (intro conv-odd-mono) auto
  ultimately show ?thesis by simp
next
  case False
  from assms have n > 0 by (intro Nat.gr0I) auto
  have 1 + real-of-int (cfrac-nth c (n+1)) ∗ cfrac-nth c (n+2) > 0
```

```
    by (intro add-pos-nonneg) auto
  with assms ‹n > 0› have conv c n ≠ conv c (Suc (Suc n))
    unfolding conv-eq-conv' conv'-Suc-right
    by (subst conv'-eq-iff) (auto simp: field-simps)
  moreover from assms have conv c n ≥ conv c (Suc (Suc n))
    by (intro conv-odd-mono) auto
  ultimately have conv c n > conv c (Suc (Suc n)) by simp
  moreover have conv c (Suc (Suc n)) ≥ conv c m using assms False
    by (intro conv-odd-mono) auto
  ultimately show ?thesis by linarith
qed

lemma conv-less-cfrac-lim:
  assumes even n n < cfrac-length c
  shows   conv c n < cfrac-lim c
proof (cases cfrac-length c)
  case (enat l)
  with assms show ?thesis by (auto simp: cfrac-lim-def conv-even-mono-strict)
next
  case [simp]: infinity
  from assms have conv c n < conv c (n + 2)
    by (intro conv-even-mono-strict) auto
  also from assms have ... ≤ cfrac-lim c
    by (intro conv-le-cfrac-lim) auto
  finally show ?thesis .
qed

lemma conv-gt-cfrac-lim:
  assumes odd n n < cfrac-length c
  shows   conv c n > cfrac-lim c
proof (cases cfrac-length c)
  case (enat l)
  with assms show ?thesis by (auto simp: cfrac-lim-def conv-odd-mono-strict)
next
  case [simp]: infinity
  from assms have cfrac-lim c ≤ conv c (n + 2)
    by (intro conv-ge-cfrac-lim) auto
  also from assms have ... < conv c n
    by (intro conv-odd-mono-strict) auto
  finally show ?thesis .
qed

lemma conv-neq-cfrac-lim:
  assumes n < cfrac-length c
  shows   conv c n ≠ cfrac-lim c
  using conv-gt-cfrac-lim[OF - assms] conv-less-cfrac-lim[OF - assms]
  by (cases even n) auto

lemma conv-ge-first: conv c n ≥ cfrac-nth c 0
```

using *conv-even-mono*[*of 0 n c*] **by** *simp*

definition *cfrac-is-zero* :: *cfrac* ⇒ *bool* **where** *cfrac-is-zero c* ⟷ *c* = *0*

lemma *cfrac-is-zero-code* [*code*]: *cfrac-is-zero* (*CFrac n xs*) ⟷ *lnull xs* ∧ *n* = *0*
unfolding *cfrac-is-zero-def lnull-def zero-cfrac-def cfrac-of-int-def*
by (*auto simp*: *cfrac-length-def*)

definition *cfrac-is-int* **where** *cfrac-is-int c* ⟷ *cfrac-length c* = *0*

lemma *cfrac-is-int-code* [*code*]: *cfrac-is-int* (*CFrac n xs*) ⟷ *lnull xs*
unfolding *cfrac-is-int-def lnull-def* **by** (*auto simp*: *cfrac-length-def*)

lemma *cfrac-length-of-int* [*simp*]: *cfrac-length* (*cfrac-of-int n*) = *0*
by *transfer auto*

lemma *cfrac-is-int-of-int* [*simp*, *intro*]: *cfrac-is-int* (*cfrac-of-int n*)
unfolding *cfrac-is-int-def* **by** *simp*

lemma *cfrac-is-int-iff*: *cfrac-is-int c* ⟷ (∃ *n*. *c* = *cfrac-of-int n*)
proof −
have *c* = *cfrac-of-int* (*cfrac-nth c 0*) **if** *cfrac-is-int c*
using *that* **unfolding** *cfrac-is-int-def* **by** *transfer auto*
thus *?thesis*
by *auto*
qed

lemma *cfrac-lim-reduce*:
assumes ¬*cfrac-is-int c*
shows *cfrac-lim c* = *cfrac-nth c 0* + *1* / *cfrac-lim* (*cfrac-tl c*)
proof (*cases cfrac-length c*)
case [*simp*]: *infinity*
have *0* < *cfrac-nth* (*cfrac-tl c*) *0*
by *simp*
also have ... ≤ *cfrac-lim* (*cfrac-tl c*)
by (*rule cfrac-lim-ge-first*)
finally have (λ*n*. *real-of-int* (*cfrac-nth c 0*) + *1* / *conv* (*cfrac-tl c*) *n*) ⟶
real-of-int (*cfrac-nth c 0*) + *1* / *cfrac-lim* (*cfrac-tl c*)
by (*intro tendsto-intros LIMSEQ-cfrac-lim*) *auto*
also have (λ*n*. *real-of-int* (*cfrac-nth c 0*) + *1* / *conv* (*cfrac-tl c*) *n*) = *conv c* ∘ *Suc*
by (*simp add*: *o-def conv-Suc*)
finally have ∗: *conv c* ⟶ *real-of-int* (*cfrac-nth c 0*) + *1* / *cfrac-lim* (*cfrac-tl c*)
by (*simp add*: *o-def filterlim-sequentially-Suc*)
show *?thesis*
by (*rule tendsto-unique*[*OF - LIMSEQ-cfrac-lim* ∗]) *auto*

next
  case [*simp*]: (*enat l*)
  from *assms* **obtain** *l′* **where** [*simp*]: *l* = *Suc l′*
    by (*cases l*) (*auto simp*: *cfrac-is-int-def zero-enat-def*)
  thus *?thesis*
    by (*auto simp*: *cfrac-lim-def conv-Suc*)
qed

lemma *cfrac-lim-tl*:
  assumes ¬*cfrac-is-int c*
  shows *cfrac-lim* (*cfrac-tl c*) = *1* / (*cfrac-lim c* − *cfrac-nth c 0*)
  using *cfrac-lim-reduce*[*OF assms*] **by** *simp*

lemma *cfrac-remainder-Suc′*:
  assumes *n* < *cfrac-length c*
  shows *cfrac-remainder c* (*Suc n*) ∗ (*cfrac-remainder c n* − *cfrac-nth c n*) = *1*
proof −
  have *0* < *real-of-int* (*cfrac-nth c* (*Suc n*)) **by** *simp*
  also have *cfrac-nth c* (*Suc n*) ≤ *cfrac-remainder c* (*Suc n*)
    using *cfrac-lim-ge-first*[*of cfrac-drop* (*Suc n*) *c*]
    by (*simp add*: *cfrac-remainder-def*)
  finally have . . . > *0* **.**

  have *cfrac-remainder c* (*Suc n*) = *cfrac-lim* (*cfrac-tl* (*cfrac-drop n c*))
    by (*simp add*: *o-def cfrac-remainder-def cfrac-drop-Suc-left*)
  also have . . . = *1* / (*cfrac-remainder c n* − *cfrac-nth c n*) **using** *assms*
   by (*subst cfrac-lim-tl*) (*auto simp*: *cfrac-remainder-def cfrac-is-int-def enat-less-iff enat-0-iff*)
  finally show *?thesis*
    using ‹*cfrac-remainder c* (*Suc n*) > *0*›
    by (*auto simp add*: *cfrac-remainder-def field-simps*)
qed

lemma *cfrac-remainder-Suc*:
  assumes *n* < *cfrac-length c*
  shows *cfrac-remainder c* (*Suc n*) = *1* / (*cfrac-remainder c n* − *cfrac-nth c n*)
proof −
  have *cfrac-remainder c* (*Suc n*) = *cfrac-lim* (*cfrac-tl* (*cfrac-drop n c*))
    by (*simp add*: *o-def cfrac-remainder-def cfrac-drop-Suc-left*)
  also have . . . = *1* / (*cfrac-remainder c n* − *cfrac-nth c n*) **using** *assms*
   by (*subst cfrac-lim-tl*) (*auto simp*: *cfrac-remainder-def cfrac-is-int-def enat-less-iff enat-0-iff*)
  finally show *?thesis* **.**
qed

lemma *cfrac-remainder-0* [*simp*]: *cfrac-remainder c 0* = *cfrac-lim c*
  by (*simp add*: *cfrac-remainder-def*)

```
context
  fixes c h k x
  defines h ≡ conv-num c and k ≡ conv-denom c and x ≡ cfrac-remainder c
begin

lemma cfrac-lim-eq-num-denom-remainder-aux:
  assumes Suc (Suc n) ≤ cfrac-length c
  shows   cfrac-lim c ∗ (k (Suc n) ∗ x (Suc (Suc n)) + k n) = h (Suc n) ∗ x (Suc
(Suc n)) + h n
  using assms
proof (induction n)
  case 0
  have cfrac-lim c ≠ cfrac-nth c 0
    using conv-neq-cfrac-lim[of 0 c] 0 by (auto simp: enat-le-iff)
  moreover have cfrac-nth c 1 ∗ (cfrac-lim c − cfrac-nth c 0) ≠ 1
    using conv-neq-cfrac-lim[of 1 c] 0
    by (auto simp: enat-le-iff conv-Suc field-simps)
  ultimately show ?case using assms
    by (auto simp: cfrac-remainder-Suc divide-simps x-def h-def k-def enat-le-iff)
      (auto simp: field-simps)
next
  case (Suc n)
  have less: enat (Suc (Suc n)) < cfrac-length c
    using Suc.prems by (cases cfrac-length c) auto
  have ∗: x (Suc (Suc n)) ≠ real-of-int (cfrac-nth c (Suc (Suc n)))
    using conv-neq-cfrac-lim[of 0 cfrac-drop (n+2) c] Suc.prems
    by (cases cfrac-length c) (auto simp: x-def cfrac-remainder-def)
  hence cfrac-lim c ∗ (k (Suc (Suc n)) ∗ x (Suc (Suc (Suc n))) + k (Suc n)) =
          (cfrac-lim c ∗ (k (Suc n) ∗ x (Suc (Suc n)) + k n)) / (x (Suc (Suc n)) −
cfrac-nth c (Suc (Suc n)))
    unfolding x-def k-def h-def using less
    by (subst cfrac-remainder-Suc) (auto simp: field-simps)
  also have cfrac-lim c ∗ (k (Suc n) ∗ x (Suc (Suc n)) + k n) =
              h (Suc n) ∗ x (Suc (Suc n)) + h n using less
    by (intro Suc.IH) auto
  also have (h (Suc n) ∗ x (Suc (Suc n)) + h n) / (x (Suc (Suc n)) − cfrac-nth c
(Suc (Suc n))) =
              h (Suc (Suc n)) ∗ x (Suc (Suc (Suc n))) + h (Suc n) using ∗
    unfolding x-def k-def h-def using less
    by (subst (3) cfrac-remainder-Suc) (auto simp: field-simps)
  finally show ?case .
qed

lemma cfrac-remainder-nonneg: cfrac-nth c n ≥ 0 =⇒ cfrac-remainder c n ≥ 0
  unfolding cfrac-remainder-def by (rule cfrac-lim-nonneg) auto

lemma cfrac-remainder-pos: cfrac-nth c n > 0 =⇒ cfrac-remainder c n > 0
  unfolding cfrac-remainder-def by (rule cfrac-lim-pos) auto
```

lemma *cfrac-lim-eq-num-denom-remainder*:
  assumes *Suc* (*Suc n*) < *cfrac-length c*
  shows *cfrac-lim c* = (*h* (*Suc n*) ∗ *x* (*Suc* (*Suc n*)) + *h n*) / (*k* (*Suc n*) ∗ *x* (*Suc* (*Suc n*)) + *k n*)
proof −
  have *k* (*Suc n*) ∗ *x* (*Suc* (*Suc n*)) + *k n* > *0*
    by (*intro add-nonneg-pos mult-nonneg-nonneg*)
      (*auto simp*: *k-def x-def intro*!: *conv-denom-pos cfrac-remainder-nonneg*)
  with *cfrac-lim-eq-num-denom-remainder-aux*[*of n*] *assms* **show** *?thesis*
    by (*auto simp add*: *field-simps h-def k-def x-def*)
qed

lemma *abs-diff-successive-convs*:
  shows |*conv c* (*Suc n*) − *conv c n*| = *1* / (*k n* ∗ *k* (*Suc n*))
proof −
  have [*simp*]: *k n* ≠ *0* **for** *n* :: *nat*
    unfolding *k-def* **using** *conv-denom-pos*[*of c n*] **by** *auto*
  have *conv c* (*Suc n*) − *conv c n* = *h* (*Suc n*) / *k* (*Suc n*) − *h n* / *k n*
    by (*simp add*: *conv-num-denom k-def h-def*)
  also have . . . = (*k n* ∗ *h* (*Suc n*) − *k* (*Suc n*) ∗ *h n*) / (*k n* ∗ *k* (*Suc n*))
    by (*simp add*: *field-simps*)
  also have *k n* ∗ *h* (*Suc n*) − *k* (*Suc n*) ∗ *h n* = (−*1*) ^ *n*
    unfolding *h-def k-def* **by** (*intro conv-num-denom-prod-diff*)
  finally show *?thesis* **by** (*simp add*: *k-def*)
qed

lemma *conv-denom-plus2-ratio-ge*: *k* (*Suc* (*Suc n*)) ≥ *2* ∗ *k n*
proof −
  have *1* ∗ *k n* + *k n* ≤ *cfrac-nth c* (*Suc* (*Suc n*)) ∗ *k* (*Suc n*) + *k n*
    by (*intro add-mono mult-mono*)
      (*auto simp*: *k-def Suc-le-eq intro*!: *conv-denom-leI*)
  thus *?thesis* **by** (*simp add*: *k-def*)
qed

end

lemma *conv′-cfrac-remainder*:
  assumes *n* < *cfrac-length c*
  shows *conv′ c n* (*cfrac-remainder c n*) = *cfrac-lim c*
  using *assms*
proof (*induction n arbitrary*: *c*)
  case (*Suc n c*)
  have *conv′ c* (*Suc n*) (*cfrac-remainder c* (*Suc n*)) =
      cfrac-nth c 0 + *1* / *conv′* (*cfrac-tl c*) *n* (*cfrac-remainder c* (*Suc n*))
    using *Suc.prems*
    by (*subst conv′-Suc-left*) (*auto intro*!: *cfrac-remainder-pos*)
  also have *cfrac-remainder c* (*Suc n*) = *cfrac-remainder* (*cfrac-tl c*) *n*
    by (*simp add*: *cfrac-remainder-def cfrac-drop-Suc-right*)

also have *conv' (cfrac-tl c) n ... = cfrac-lim (cfrac-tl c)*
using *Suc.prems* **by** (*subst Suc.IH*) (*auto simp*: *cfrac-remainder-def enat-less-iff*)
also have *cfrac-nth c 0 + 1 / ... = cfrac-lim c*
using *Suc.prems* **by** (*intro cfrac-lim-reduce* [*symmetric*]) (*auto simp*: *cfrac-is-int-def*)
finally show *?case* **by** (*simp add*: *cfrac-remainder-def cfrac-drop-Suc-right*)
qed *auto*

lemma *cfrac-lim-rational* [*intro*]:
assumes *cfrac-length c* $< \infty$
shows *cfrac-lim c* $\in \mathbb{Q}$
using *assms* **by** (*cases cfrac-length c*) (*auto simp*: *cfrac-lim-def*)

lemma *linfinite-cfrac-of-real-aux*:
$x \notin \mathbb{Q} \Longrightarrow x \in \{0<..<1\} \Longrightarrow$ *linfinite* (*cfrac-of-real-aux x*)
proof (*coinduction arbitrary*: *x*)
case (*linfinite x*)
hence $1 / x \notin \mathbb{Q}$ **using** *Rats-divide*[*of 1 1 / x*] **by** *auto*
thus *?case* **using** *linfinite Ints-subset-Rats*
by (*intro disjI1 exI*[*of - nat* $\lfloor 1/x \rfloor - 1$] *exI*[*of - cfrac-of-real-aux* (*frac* (*1/x*))]
exI[*of - frac* (*1/x*)] *conjI*)
(*auto simp*: *cfrac-of-real-aux.code*[*of x*] *frac-lt-1*)
qed

lemma *cfrac-length-of-real-irrational*:
assumes $x \notin \mathbb{Q}$
shows *cfrac-length* (*cfrac-of-real x*) $= \infty$
proof (*insert assms*, *transfer*, *clarify*)
fix *x* :: *real* **assume** $x \notin \mathbb{Q}$
thus *llength* (*cfrac-of-real-aux* (*frac x*)) $= \infty$
using *linfinite-cfrac-of-real-aux*[*of frac x*] *Ints-subset-Rats*
by (*auto simp*: *linfinite-conv-llength frac-lt-1*)
qed

lemma *cfrac-length-of-real-reduce*:
assumes $x \notin \mathbb{Z}$
shows *cfrac-length* (*cfrac-of-real x*) = *eSuc* (*cfrac-length* (*cfrac-of-real* (*1 / frac x*)))
using *assms*
by (*transfer*, *subst cfrac-of-real-aux.code*) (*auto simp*: *frac-lt-1*)

lemma *cfrac-length-of-real-int* [*simp*]: $x \in \mathbb{Z} \Longrightarrow$ *cfrac-length* (*cfrac-of-real x*) = *0*
by *transfer auto*

lemma *conv-cfrac-of-real-le-ge*:
assumes $n \leq$ *cfrac-length* (*cfrac-of-real x*)
shows *if even n then conv* (*cfrac-of-real x*) $n \leq x$ *else conv* (*cfrac-of-real x*) $n \geq x$
using *assms*
proof (*induction n arbitrary*: *x*)

case (*Suc n x*)
hence [*simp*]: $x \notin \mathbb{Z}$
using *Suc* **by** (*auto simp*: *enat-0-iff*)
let *?x′ = 1 / frac x*
have *enat n ≤ cfrac-length* (*cfrac-of-real* (*1 / frac x*))
using *Suc.prems* **by** (*auto simp*: *cfrac-length-of-real-reduce simp flip*: *eSuc-enat*)
hence *IH*: *if even n then conv* (*cfrac-of-real ?x′*) *n ≤ ?x′ else ?x′ ≤ conv* (*cfrac-of-real ?x′*) *n*
using *Suc.prems* **by** (*intro Suc.IH*) *auto*
have *remainder-pos*: *conv* (*cfrac-of-real ?x′*) *n > 0*
by (*rule conv-pos*) (*auto simp*: *frac-le-1*)
show *?case*
proof (*cases even n*)
case *True*
have *x ≤ real-of-int* ⌊*x*⌋ + *frac x*
by (*simp add*: *frac-def*)
also have *frac x ≤ 1 / conv* (*cfrac-of-real ?x′*) *n*
using *IH True remainder-pos frac-gt-0-iff*[*of x*] **by** (*simp add*: *field-simps*)
finally show *?thesis* **using** *True*
by (*auto simp*: *conv-Suc cfrac-tl-of-real*)
next
case *False*
have *real-of-int* ⌊*x*⌋ + *1 / conv* (*cfrac-of-real ?x′*) *n ≤ real-of-int* ⌊*x*⌋ + *frac x*
using *IH False remainder-pos frac-gt-0-iff*[*of x*] **by** (*simp add*: *field-simps*)
also have ... = *x*
by (*simp add*: *frac-def*)
finally show *?thesis* **using** *False*
by (*auto simp*: *conv-Suc cfrac-tl-of-real*)
qed
qed *auto*

lemma *cfrac-lim-of-real* [*simp*]: *cfrac-lim* (*cfrac-of-real x*) = *x*
proof (*cases cfrac-length* (*cfrac-of-real x*))
case (*enat l*)
hence *conv* (*cfrac-of-real x*) *l = x*
proof (*induction l arbitrary*: *x*)
case *0*
hence $x \in \mathbb{Z}$
using *cfrac-length-of-real-reduce zero-enat-def* **by** *fastforce*
thus *?case* **by** (*auto elim*: *Ints-cases*)
next
case (*Suc l x*)
hence [*simp*]: $x \notin \mathbb{Z}$
by (*auto simp*: *enat-0-iff*)
have *eSuc* (*cfrac-length* (*cfrac-of-real* (*1 / frac x*))) = *enat* (*Suc l*)
using *Suc.prems* **by** (*auto simp*: *cfrac-length-of-real-reduce*)
hence *conv* (*cfrac-of-real* (*1 / frac x*)) *l = 1 / frac x*
by (*intro Suc.IH*) (*auto simp flip*: *eSuc-enat*)
thus *?case*

by (*simp add*: *conv-Suc cfrac-tl-of-real frac-def*)
qed
thus *?thesis* **by** (*simp add*: *enat cfrac-lim-def*)
next
case [*simp*]: *infinity*
have *lim*: *conv* (*cfrac-of-real x*) ⟶ *cfrac-lim* (*cfrac-of-real x*)
by (*simp add*: *LIMSEQ-cfrac-lim*)
have *cfrac-lim* (*cfrac-of-real x*) ≤ *x*
proof (*rule tendsto-upperbound*)
show (λ*n*. *conv* (*cfrac-of-real x*) (*n* ∗ *2*)) ⟶ *cfrac-lim* (*cfrac-of-real x*)
by (*intro filterlim-compose*[*OF lim*] *mult-nat-right-at-top*) *auto*
show *eventually* (λ*n*. *conv* (*cfrac-of-real x*) (*n* ∗ *2*) ≤ *x*) *at-top*
using *conv-cfrac-of-real-le-ge*[*of n* ∗ *2 x* **for** *n*] **by** (*intro always-eventually*) *auto*
qed *auto*
moreover have *cfrac-lim* (*cfrac-of-real x*) ≥ *x*
proof (*rule tendsto-lowerbound*)
show (λ*n*. *conv* (*cfrac-of-real x*) (*Suc* (*n* ∗ *2*))) ⟶ *cfrac-lim* (*cfrac-of-real x*)
by (*intro filterlim-compose*[*OF lim*] *filterlim-compose*[*OF filterlim-Suc*] *mult-nat-right-at-top*) *auto*
show *eventually* (λ*n*. *conv* (*cfrac-of-real x*) (*Suc* (*n* ∗ *2*)) ≥ *x*) *at-top*
using *conv-cfrac-of-real-le-ge*[*of Suc* (*n* ∗ *2*) *x* **for** *n*] **by** (*intro always-eventually*) *auto*
qed *auto*
ultimately show *?thesis* **by** (*rule antisym*)
qed

lemma *Ints-add-left-cancel*: *x* ∈ ℤ ⟹ *x* + *y* ∈ ℤ ⟷ *y* ∈ ℤ
using *Ints-diff*[*of x* + *y x*] **by** *auto*

lemma *Ints-add-right-cancel*: *y* ∈ ℤ ⟹ *x* + *y* ∈ ℤ ⟷ *x* ∈ ℤ
using *Ints-diff*[*of x* + *y y*] **by** *auto*

lemma *cfrac-of-real-conv′*:
fixes *m n* :: *nat*
assumes *x* > *1 m* < *n*
shows *cfrac-nth* (*cfrac-of-real* (*conv′ c n x*)) *m* = *cfrac-nth c m*
using *assms*
proof (*induction n arbitrary*: *c m*)
case (*Suc n c m*)
from *Suc.prems* **have** *gt-1*: *1* < *conv′* (*cfrac-tl c*) *n x*
by (*intro conv′-gt-1*) (*auto simp*: *enat-le-iff intro*: *cfrac-nth-pos*)
show *?case*
proof (*cases m*)
case *0*
thus *?thesis* **using** *gt-1 Suc.prems*
by (*simp add*: *conv′-Suc-left nat-add-distrib floor-eq-iff*)
next

```
    case (Suc m′)
    from gt-1 have 1 / conv′ (cfrac-tl c) n x ∈ {0<..<1}
      by auto
    have 1 / conv′ (cfrac-tl c) n x ∉ ℤ
    proof
      assume 1 / conv′ (cfrac-tl c) n x ∈ ℤ
      then obtain k :: int where k: 1 / conv′ (cfrac-tl c) n x = of-int k
        by (elim Ints-cases)
      have real-of-int k ∈ {0<..<1}
        using gt-1 by (subst k [symmetric]) auto
      thus False by auto
    qed
    hence not-int: real-of-int (cfrac-nth c 0) + 1 / conv′ (cfrac-tl c) n x ∉ ℤ
      by (subst Ints-add-left-cancel) (auto simp: field-simps elim!: Ints-cases)
    have cfrac-nth (cfrac-of-real (conv′ c (Suc n) x)) m =
           cfrac-nth (cfrac-of-real (of-int (cfrac-nth c 0) + 1 / conv′ (cfrac-tl c) n x))
(Suc m′)
      using ‹x > 1› by (subst conv′-Suc-left) (auto simp: Suc)
    also have ... = cfrac-nth (cfrac-of-real (1 / frac (1 / conv′ (cfrac-tl c) n x)))
m′
         using ‹x > 1› Suc not-int by (subst cfrac-nth-of-real-Suc) (auto simp:
frac-add-of-int)
    also have 1 / conv′ (cfrac-tl c) n x ∈ {0<..<1} using gt-1
      by (auto simp: field-simps)
    hence frac (1 / conv′ (cfrac-tl c) n x) = 1 / conv′ (cfrac-tl c) n x
      by (subst frac-eq) auto
    hence 1 / frac (1 / conv′ (cfrac-tl c) n x) = conv′ (cfrac-tl c) n x
      by simp
    also have cfrac-nth (cfrac-of-real ... ) m′ = cfrac-nth c m
      using Suc.prems by (subst Suc.IH) (auto simp: Suc enat-le-iff)
    finally show ?thesis .
  qed
qed simp-all

lemma cfrac-lim-irrational:
  assumes [simp]: cfrac-length c = ∞
  shows   cfrac-lim c ∉ ℚ
proof
  assume cfrac-lim c ∈ ℚ
  then obtain a :: int and b :: nat where ab: b > 0 cfrac-lim c = a / b
    by (auto simp: Rats-eq-int-div-nat)
  define h and k where h = conv-num c and k = conv-denom c

  have filterlim (λm. conv-denom c (Suc m)) at-top at-top
    using conv-denom-at-top filterlim-Suc by (rule filterlim-compose)
  then obtain m where m: conv-denom c (Suc m) ≥ b + 1
    by (auto simp: filterlim-at-top eventually-at-top-linorder)

  have *: (a * k m − b * h m) / (k m * b) = a / b − h m / k m
```

using ‹$b > 0$› **by** (*simp add*: *field-simps k-def*)
have $|$*cfrac-lim c* $-$ *conv c m*$|$ = $|$($a * k\ m - b * h\ m$) / ($k\ m * b$)$|$
by (*subst* $*$) (*auto simp*: *ab h-def k-def conv-num-denom*)
also have ... = $|a * k\ m - b * h\ m|$ / ($k\ m * b$)
by (*simp add*: *k-def*)
finally have *eq*: $|$*cfrac-lim c* $-$ *conv c m*$|$ = *of-int* $|a * k\ m - b * h\ m|$ / *of-int* ($k\ m * b$) .

have $|$*cfrac-lim c* $-$ *conv c m*$|$ $*$ ($k\ m * b$) $\neq 0$
using *conv-neq-cfrac-lim*[*of m c*] ‹$b > 0$› **by** (*auto simp*: *k-def*)
also have $|$*cfrac-lim c* $-$ *conv c m*$|$ $*$ ($k\ m * b$) = *of-int* $|a * k\ m - b * h\ m|$
using ‹$b > 0$› **by** (*subst eq*) (*auto simp*: *k-def*)
finally have $|a * k\ m - b * h\ m| \geq 1$ **by** *linarith*
hence *real-of-int* $|a * k\ m - b * h\ m| \geq 1$ **by** *linarith*
hence 1 / *of-int* ($k\ m * b$) $\leq$ *of-int* $|a * k\ m - b * h\ m|$ / *real-of-int* ($k\ m * b$)
using ‹$b > 0$› **by** (*intro divide-right-mono*) (*auto simp*: *k-def*)
also have ... = $|$*cfrac-lim c* $-$ *conv c m*$|$
by (*rule eq* [*symmetric*])
also have ... $\leq$ 1 / *real-of-int* (*conv-denom c m* $*$ *conv-denom c* (*Suc m*))
by (*intro cfrac-lim-minus-conv-upper-bound*) *auto*
also have ... = 1 / (*real-of-int* ($k\ m$) $*$ *real-of-int* (k (*Suc m*)))
by (*simp add*: *k-def*)
also have ... < 1 / (*real-of-int* ($k\ m$) $*$ *real b*)
using *m* ‹$b > 0$›
by (*intro divide-strict-left-mono mult-strict-left-mono*) (*auto simp*: *k-def*)
finally show *False* **by** *simp*
qed

lemma *cfrac-infinite-iff*: *cfrac-length c* = $\infty \longleftrightarrow$ *cfrac-lim c* $\notin \mathbb{Q}$
using *cfrac-lim-irrational*[*of c*] *cfrac-lim-rational*[*of c*] **by** *auto*

lemma *cfrac-lim-rational-iff*: *cfrac-lim c* $\in \mathbb{Q} \longleftrightarrow$ *cfrac-length c* $\neq \infty$
using *cfrac-lim-irrational*[*of c*] *cfrac-lim-rational*[*of c*] **by** *auto*

lemma *cfrac-of-real-infinite-iff* [*simp*]: *cfrac-length* (*cfrac-of-real x*) = $\infty \longleftrightarrow x \notin \mathbb{Q}$
by (*simp add*: *cfrac-infinite-iff*)

lemma *cfrac-remainder-rational-iff* [*simp*]:
cfrac-remainder c n $\in \mathbb{Q} \longleftrightarrow$ *cfrac-length c* $< \infty$
proof −
have *cfrac-remainder c n* $\in \mathbb{Q} \longleftrightarrow$ *cfrac-lim* (*cfrac-drop n c*) $\in \mathbb{Q}$
by (*simp add*: *cfrac-remainder-def*)
also have ... $\longleftrightarrow$ *cfrac-length c* $\neq \infty$
by (*cases cfrac-length c*) (*auto simp add*: *cfrac-lim-rational-iff*)
finally show *?thesis* **by** *simp*
qed

lift-definition *cfrac-cons* :: *int* $\Rightarrow$ *cfrac* $\Rightarrow$ *cfrac* **is**

λa bs. case bs of (*b*, *bs*) ⇒ *if b* ≤ *0 then* (*1*, *LNil*) *else* (*a*, *LCons* (*nat* (*b* − *1*)) *bs*) **.**

lemma *cfrac-nth-cons*:
assumes *cfrac-nth x 0* ≥ *1*
shows *cfrac-nth* (*cfrac-cons a x*) *n* = (*if n* = *0 then a else cfrac-nth x* (*n* − *1*))
using *assms*
proof (*transfer*, *goal-cases*)
case (*1 x a n*)
obtain *b bs* **where** [*simp*]: *x* = (*b*, *bs*)
by (*cases x*)
show *?case* **using** *1*
by (*cases llength bs*) (*auto simp*: *lnth-LCons eSuc-enat le-imp-diff-is-add split*: *nat.splits*)
qed

lemma *cfrac-length-cons* [*simp*]:
assumes *cfrac-nth x 0* ≥ *1*
shows *cfrac-length* (*cfrac-cons a x*) = *eSuc* (*cfrac-length x*)
using *assms* **by** *transfer auto*

lemma *cfrac-tl-cons* [*simp*]:
assumes *cfrac-nth x 0* ≥ *1*
shows *cfrac-tl* (*cfrac-cons a x*) = *x*
using *assms* **by** *transfer auto*

lemma *cfrac-cons-tl*:
assumes ¬*cfrac-is-int x*
shows *cfrac-cons* (*cfrac-nth x 0*) (*cfrac-tl x*) = *x*
using *assms* **unfolding** *cfrac-is-int-def*
by *transfer* (*auto split*: *llist.splits*)

1.3 Non-canonical continued fractions

As we will show later, every irrational number has a unique continued fraction expansion. Every rational number x, however, has two different expansions: The canonical one ends with some number n (which is not equal to 1 unless $x = 1$) and a non-canonical one which ends with $n - 1, 1$.

We now define this non-canonical expansion analogously to the canonical one before and show its characteristic properties:

- The length of the non-canonical expansion is one greater than that of the canonical one.
- If the expansion is infinite, the non-canonical and the canonical one coincide.
- The coefficients of the expansions are all equal except for the last two.

The last coefficient of the non-canonical expansion is always 1, and the second to last one is the last of the canonical one minus 1.

```
lift-definition cfrac-canonical :: cfrac ⇒ bool is
  λ(x, xs). ¬lfinite xs ∨ lnull xs ∨ llast xs ≠ 0 .

lemma cfrac-canonical [code]:
  cfrac-canonical (CFrac x xs) ⟷ lnull xs ∨ llast xs ≠ 0 ∨ ¬lfinite xs
  by (auto simp add: cfrac-canonical-def)

lemma cfrac-canonical-iff:
  cfrac-canonical c ⟷
     cfrac-length c ∈ {0, ∞} ∨ cfrac-nth c (the-enat (cfrac-length c)) ≠ 1
proof (transfer, clarify, goal-cases)
  case (1 x xs)
  show ?case
    by (cases llength xs)
       (auto simp: llast-def enat-0 lfinite-conv-llength-enat split: nat.splits)
qed

lemma llast-cfrac-of-real-aux-nonzero:
  assumes lfinite (cfrac-of-real-aux x) ¬lnull (cfrac-of-real-aux x)
  shows   llast (cfrac-of-real-aux x) ≠ 0
  using assms
proof (induction cfrac-of-real-aux x arbitrary: x rule: lfinite-induct)
  case (LCons x)
  from LCons.prems have x ∈ {0<..<1}
    by (subst (asm) cfrac-of-real-aux.code) (auto split: if-splits)
  show ?case
  proof (cases 1 / x ∈ ℤ)
    case False
    thus ?thesis using LCons
      by (auto simp: llast-LCons frac-lt-1 cfrac-of-real-aux.code[of x])
  next
    case True
    then obtain n where n: 1 / x = of-int n
      by (elim Ints-cases)
    have 1 / x > 1 using ‹x ∈ -› by auto
    with n have n > 1 by simp
    from n have x = 1 / of-int n
      using ‹n > 1› ‹x ∈ -› by (simp add: field-simps)
    with ‹n > 1› show ?thesis
     using LCons cfrac-of-real-aux.code[of x] by (auto simp: llast-LCons frac-lt-1)
  qed
qed auto

lemma cfrac-canonical-of-real [intro]: cfrac-canonical (cfrac-of-real x)
  by (transfer fixing: x) (use llast-cfrac-of-real-aux-nonzero[of frac x] in force)
```

primcorec *cfrac-of-real-alt-aux* :: *real* ⇒ *nat llist* **where**
cfrac-of-real-alt-aux x =
(*if x* ∈ {*0*<..<*1*} *then*
if 1 / *x* ∈ ℤ *then*
LCons (*nat* ⌊*1*/*x*⌋ − *2*) (*LCons 0 LNil*)
else LCons (*nat* ⌊*1*/*x*⌋ − *1*) (*cfrac-of-real-alt-aux* (*frac* (*1*/*x*)))
else LNil)

lemma *cfrac-of-real-aux-alt-LNil* [*simp*]: *x* ∉ {*0*<..<*1*} ⟹ *cfrac-of-real-alt-aux x* = *LNil*
by (*subst cfrac-of-real-alt-aux.code*) *auto*

lemma *cfrac-of-real-aux-alt-0* [*simp*]: *cfrac-of-real-alt-aux 0* = *LNil*
by (*subst cfrac-of-real-alt-aux.code*) *auto*

lemma *cfrac-of-real-aux-alt-eq-LNil-iff* [*simp*]: *cfrac-of-real-alt-aux x* = *LNil* ⟷ *x* ∉ {*0*<..<*1*}
by (*subst cfrac-of-real-alt-aux.code*) *auto*

lift-definition *cfrac-of-real-alt* :: *real* ⇒ *cfrac* **is**
λ*x*. *if x* ∈ ℤ *then* (⌊*x*⌋ − *1*, *LCons 0 LNil*) *else* (⌊*x*⌋, *cfrac-of-real-alt-aux* (*frac x*)) **.**

lemma *cfrac-tl-of-real-alt*:
assumes *x* ∉ ℤ
shows *cfrac-tl* (*cfrac-of-real-alt x*) = *cfrac-of-real-alt* (*1* / *frac x*)
using *assms*
proof (*transfer*, *goal-cases*)
case (*1 x*)
show *?case*
proof (*cases 1* / *frac x* ∈ ℤ)
case *False*
from *1* **have** *int* (*nat* ⌊*1* / *frac x*⌋ − *Suc 0*) + *1* = ⌊*1* / *frac x*⌋
by (*subst of-nat-diff*) (*auto simp*: *le-nat-iff frac-le-1*)
with *False* **show** *?thesis*
using ‹*x* ∉ ℤ›
by (*subst cfrac-of-real-alt-aux.code*) (*auto split*: *llist.splits simp*: *frac-lt-1*)
next
case *True*
then obtain *n* **where** *1* / *frac x* = *of-int n*
by (*auto simp*: *Ints-def*)
moreover have *1* / *frac x* > *1*
using *1* **by** (*auto simp*: *divide-simps frac-lt-1*)
ultimately have *1* / *frac x* ≥ *2*
by *simp*
hence *int* (*nat* ⌊*1* / *frac x*⌋ − *2*) + *2* = ⌊*1* / *frac x*⌋
by (*subst of-nat-diff*) (*auto simp*: *le-nat-iff frac-le-1*)
thus *?thesis*
using ‹*x* ∉ ℤ›

by (*subst cfrac-of-real-alt-aux.code*) (*auto split*: *llist.splits simp*: *frac-lt-1*)
qed
qed

lemma *cfrac-nth-of-real-alt-Suc*:
assumes $x \notin \mathbb{Z}$
shows *cfrac-nth* (*cfrac-of-real-alt x*) (*Suc n*) = *cfrac-nth* (*cfrac-of-real-alt* (*1* / *frac x*)) *n*
proof −
have *cfrac-nth* (*cfrac-of-real-alt x*) (*Suc n*) =
cfrac-nth (*cfrac-tl* (*cfrac-of-real-alt x*)) *n*
by *simp*
also have *cfrac-tl* (*cfrac-of-real-alt x*) = *cfrac-of-real-alt* (*1* / *frac x*)
by (*simp add*: *cfrac-tl-of-real-alt assms*)
finally show *?thesis* **.**
qed

lemma *cfrac-nth-gt0-of-real-int* [*simp*]:
$m > 0 \Longrightarrow$ *cfrac-nth* (*cfrac-of-real* (*of-int n*)) *m* = *1*
by *transfer* (*auto simp*: *lnth-LCons eSuc-def enat-0-iff split*: *nat.splits*)

lemma *cfrac-nth-0-of-real-alt-int* [*simp*]:
cfrac-nth (*cfrac-of-real-alt* (*of-int n*)) *0* = *n* − *1*
by *transfer auto*

lemma *cfrac-nth-gt0-of-real-alt-int* [*simp*]:
$m > 0 \Longrightarrow$ *cfrac-nth* (*cfrac-of-real-alt* (*of-int n*)) *m* = *1*
by *transfer* (*auto simp*: *lnth-LCons eSuc-def split*: *nat.splits*)

lemma *llength-cfrac-of-real-alt-aux*:
assumes $x \in \{0<..<1\}$
shows *llength* (*cfrac-of-real-alt-aux x*) = *eSuc* (*llength* (*cfrac-of-real-aux x*))
using *assms*
proof (*coinduction arbitrary*: *x rule*: *enat-coinduct*)
case (*Eq-enat x*)
show *?case*
proof (*cases* $1 / x \in \mathbb{Z}$)
case *False*
with *Eq-enat* **have** *frac* (*1* / *x*) $\in \{0<..<1\}$
by (*auto intro*: *frac-lt-1*)
hence $\exists x'$. *llength* (*cfrac-of-real-alt-aux* (*frac* (*1* / *x*))) =
llength (*cfrac-of-real-alt-aux* x') ∧
llength (*cfrac-of-real-aux* (*frac* (*1* / *x*))) = *llength* (*cfrac-of-real-aux* x')
∧
$0 < x' \wedge x' < 1$
by (*intro exI*[*of* - *frac* (*1* / *x*)]) *auto*
thus *?thesis* **using** *False Eq-enat*
by (*auto simp*: *cfrac-of-real-alt-aux.code*[*of x*] *cfrac-of-real-aux.code*[*of x*])
qed (*use Eq-enat* **in** ‹*auto simp*: *cfrac-of-real-alt-aux.code*[*of x*] *cfrac-of-real-aux.code*[*of*

```
x]›)
qed

lemma cfrac-length-of-real-alt:
  cfrac-length (cfrac-of-real-alt x) = eSuc (cfrac-length (cfrac-of-real x))
  by transfer (auto simp: llength-cfrac-of-real-alt-aux frac-lt-1)

lemma cfrac-of-real-alt-aux-eq-regular:
  assumes x ∈ {0<..<1} llength (cfrac-of-real-aux x) = ∞
  shows   cfrac-of-real-alt-aux x = cfrac-of-real-aux x
  using assms
proof (coinduction arbitrary: x)
  case (Eq-llist x)
  hence ∃ x'. cfrac-of-real-aux (frac (1 / x)) =
        cfrac-of-real-aux x' ∧
        cfrac-of-real-alt-aux (frac (1 / x)) =
        cfrac-of-real-alt-aux x' ∧ 0 < x' ∧ x' < 1 ∧ llength (cfrac-of-real-aux x') =
∞
    by (intro exI[of - frac (1 / x)])
       (auto simp: cfrac-of-real-aux.code[of x] cfrac-of-real-alt-aux.code[of x]
                   eSuc-eq-infinity-iff frac-lt-1)
  with Eq-llist show ?case
    by (auto simp: eSuc-eq-infinity-iff)
qed

lemma cfrac-of-real-alt-irrational [simp]:
  assumes x ∉ ℚ
  shows   cfrac-of-real-alt x = cfrac-of-real x
proof –
  from assms have cfrac-length (cfrac-of-real x) = ∞
    using cfrac-length-of-real-irrational by blast
  with assms show ?thesis
    by transfer
       (use Ints-subset-Rats in
      ‹auto intro!: cfrac-of-real-alt-aux-eq-regular simp: frac-lt-1 llength-cfrac-of-real-alt-aux›)
qed

lemma cfrac-nth-of-real-alt-0:
  cfrac-nth (cfrac-of-real-alt x) 0 = (if x ∈ ℤ then ⌊x⌋ − 1 else ⌊x⌋)
  by transfer auto

lemma cfrac-nth-of-real-alt:
  fixes n :: nat and x :: real
  defines c ≡ cfrac-of-real x
  defines c' ≡ cfrac-of-real-alt x
  defines l ≡ cfrac-length c
  shows   cfrac-nth c' n =
          (if enat n = l then
             cfrac-nth c n − 1
```

else if enat n = l + 1 then
1
else
cfrac-nth c n)
unfolding *c-def c′-def l-def*
proof (*induction n arbitrary*: *x rule*: *less-induct*)
case (*less n*)
consider $x \notin \mathbb{Q}$ | $x \in \mathbb{Z}$ | $n = 0$ $x \in \mathbb{Q} - \mathbb{Z}$ | n' **where** $n = Suc\ n'$ $x \in \mathbb{Q} - \mathbb{Z}$
by (*cases n*) *auto*
thus *?case*
proof *cases*
assume $x \notin \mathbb{Q}$
thus *?thesis*
by (*auto simp*: *cfrac-length-of-real-irrational*)
next
assume $x \in \mathbb{Z}$
thus *?thesis*
by (*auto simp*: *Ints-def one-enat-def zero-enat-def*)
next
assume ∗: $n = 0$ $x \in \mathbb{Q} - \mathbb{Z}$
have *enat 0* ≠ *cfrac-length* (*cfrac-of-real x*) + *1*
using *zero-enat-def* **by** *auto*
moreover have *enat 0* ≠ *cfrac-length* (*cfrac-of-real x*)
using ∗ *cfrac-length-of-real-reduce zero-enat-def* **by** *auto*
ultimately show *?thesis* **using** ∗
by (*auto simp*: *cfrac-nth-of-real-alt-0*)
next
fix n' **assume** ∗: $n = Suc\ n'$ $x \in \mathbb{Q} - \mathbb{Z}$
from *less.IH* [*of n′ 1 / frac x*] **and** ∗ **show** *?thesis*
by (*auto simp*: *cfrac-nth-of-real-Suc cfrac-nth-of-real-alt-Suc cfrac-length-of-real-reduce*

eSuc-def one-enat-def enat-0-iff split: *enat.splits*)
qed
qed

lemma *cfrac-of-real-length-eq-0-iff*: *cfrac-length* (*cfrac-of-real x*) = *0* ⟷ $x \in \mathbb{Z}$
by *transfer* (*auto simp*: *frac-lt-1*)

lemma *conv′-cong*:
assumes (⋀*k*. *k* < *n* ⟹ *cfrac-nth c k* = *cfrac-nth c′ k*) *n* = *n′* *x* = *y*
shows *conv′ c n x* = *conv′ c′ n′ y*
using *assms*(*1*) **unfolding** *assms*(*2*,*3*) [*symmetric*]
by (*induction n arbitrary*: *x*) (*auto simp*: *conv′-Suc-right*)

lemma *conv-cong*:
assumes (⋀*k*. *k* ≤ *n* ⟹ *cfrac-nth c k* = *cfrac-nth c′ k*) *n* = *n′*
shows *conv c n* = *conv c′ n′*
using *assms*(*1*) **unfolding** *assms*(*2*) [*symmetric*]
by (*induction n arbitrary*: *c c′*) (*auto simp*: *conv-Suc*)

lemma *conv′-cfrac-of-real-alt*:
  assumes *enat n* ≤ *cfrac-length* (*cfrac-of-real x*)
  shows *conv′* (*cfrac-of-real-alt x*) *n y* = *conv′* (*cfrac-of-real x*) *n y*
proof (*cases cfrac-length* (*cfrac-of-real x*))
  case *infinity*
  thus *?thesis* **by** *auto*
next
  case [*simp*]: (*enat l′*)
  with *assms* **show** *?thesis*
    by (*intro conv′-cong refl*; *subst cfrac-nth-of-real-alt*) (*auto simp*: *one-enat-def*)
qed

lemma *cfrac-lim-of-real-alt* [*simp*]: *cfrac-lim* (*cfrac-of-real-alt x*) = *x*
proof (*cases cfrac-length* (*cfrac-of-real x*))
  case *infinity*
  thus *?thesis* **by** *auto*
next
  case (*enat l*)
  thus *?thesis*
  proof (*induction l arbitrary*: *x*)
    case *0*
    hence $x \in \mathbb{Z}$
      using *cfrac-of-real-length-eq-0-iff zero-enat-def* **by** *auto*
    thus *?case*
     by (*auto simp*: *Ints-def cfrac-lim-def cfrac-length-of-real-alt eSuc-def conv-Suc*)
  next
    case (*Suc l x*)
    hence ∗: ¬*cfrac-is-int* (*cfrac-of-real-alt x*) $x \notin \mathbb{Z}$
      by (*auto simp*: *cfrac-is-int-def cfrac-length-of-real-alt Ints-def zero-enat-def eSuc-def*)
    hence *cfrac-lim* (*cfrac-of-real-alt x*) =
          of-int ⌊*x*⌋ + *1* / *cfrac-lim* (*cfrac-tl* (*cfrac-of-real-alt x*))
      by (*subst cfrac-lim-reduce*) (*auto simp*: *cfrac-nth-of-real-alt-0*)
    also have *cfrac-length* (*cfrac-of-real* (*1* / *frac x*)) = *l*
     using *Suc.prems* ∗ **by** (*metis cfrac-length-of-real-reduce eSuc-enat eSuc-inject*)
    hence *1* / *cfrac-lim* (*cfrac-tl* (*cfrac-of-real-alt x*)) = *frac x*
      by (*subst cfrac-tl-of-real-alt*[*OF* ∗(*2*)], *subst Suc*) (*use Suc.prems* ∗ **in** *auto*)
    also have *real-of-int* ⌊*x*⌋ + *frac x* = *x*
      by (*simp add*: *frac-def*)
    finally show *?case* **.**
  qed
qed

lemma *cfrac-eqI*:
  assumes *cfrac-length c* = *cfrac-length c′* **and** ⋀*n*. *cfrac-nth c n* = *cfrac-nth c′ n*
  shows *c* = *c′*
proof (*use assms* **in** *transfer*, *safe*, *goal-cases*)
  case (*1 a xs b ys*)

```
  from 1(2)[of 0] show ?case
    by auto
next
  case (2 a xs b ys)
  define f where f = (λxs n. if enat (Suc n) ≤ llength xs then int (lnth xs n) +
1 else 1)
  have ∀ n. f xs n = f ys n
    using 2(2)[of Suc n for n] by (auto simp: f-def cong: if-cong)
  with 2(1) show xs = ys
  proof (coinduction arbitrary: xs ys)
    case (Eq-llist xs ys)
    show ?case
    proof (cases lnull xs ∨ lnull ys)
      case False
      from False have ∗: enat (Suc 0) ≤ llength ys
        using Suc-ile-eq zero-enat-def by auto
      have llength (ltl xs) = llength (ltl ys)
        using Eq-llist by (cases xs; cases ys) auto
      moreover have lhd xs = lhd ys
        using False ∗ Eq-llist(1) spec[OF Eq-llist(2), of 0]
        by (auto simp: f-def lnth-0-conv-lhd)
      moreover have f (ltl xs) n = f (ltl ys) n for n
        using Eq-llist(1) ∗ spec[OF Eq-llist(2), of Suc n]
        by (cases xs; cases ys) (auto simp: f-def Suc-ile-eq split: if-splits)
      ultimately show ?thesis
        using False by auto
    next
      case True
      thus ?thesis
        using Eq-llist(1) by auto
    qed
  qed
qed

lemma cfrac-eq-0I:
  assumes cfrac-lim c = 0 cfrac-nth c 0 ≥ 0
  shows   c = 0
proof −
  have ∗: cfrac-is-int c
  proof (rule ccontr)
    assume ∗: ¬cfrac-is-int c
    from ∗ have conv c 0 < cfrac-lim c
    by (intro conv-less-cfrac-lim) (auto simp: cfrac-is-int-def simp flip: zero-enat-def)
    hence cfrac-nth c 0 < 0
      using assms by simp
    thus False
      using assms by simp
  qed
  from ∗ assms have cfrac-nth c 0 = 0
```

by (*auto simp*: *cfrac-lim-def cfrac-is-int-def*)
from ∗ **and** *this* **show** *c = 0*
unfolding *zero-cfrac-def cfrac-is-int-def* **by** *transfer auto*
qed

lemma *cfrac-eq-1I*:
assumes *cfrac-lim c = 1 cfrac-nth c 0 ≠ 0*
shows *c = 1*
proof −
have ∗: *cfrac-is-int c*
proof (*rule ccontr*)
assume ∗: ¬*cfrac-is-int c*
from ∗ **have** *conv c 0 < cfrac-lim c*
by (*intro conv-less-cfrac-lim*) (*auto simp*: *cfrac-is-int-def simp flip*: *zero-enat-def*)
hence *cfrac-nth c 0 < 0*
using *assms* **by** *simp*

have *cfrac-lim c = real-of-int* (*cfrac-nth c 0*) + *1* / *cfrac-lim* (*cfrac-tl c*)
using ∗ **by** (*subst cfrac-lim-reduce*) *auto*
also have *real-of-int* (*cfrac-nth c 0*) < *0*
using ‹*cfrac-nth c 0 < 0*› **by** *simp*
also have *1* / *cfrac-lim* (*cfrac-tl c*) ≤ *1*
proof −
have *1* ≤ *cfrac-nth* (*cfrac-tl c*) *0*
by *auto*
also have ... ≤ *cfrac-lim* (*cfrac-tl c*)
by (*rule cfrac-lim-ge-first*)
finally show *?thesis* **by** *simp*
qed
finally show *False*
using *assms* **by** *simp*
qed

from ∗ *assms* **have** *cfrac-nth c 0 = 1*
by (*auto simp*: *cfrac-lim-def cfrac-is-int-def*)
from ∗ **and** *this* **show** *c = 1*
unfolding *one-cfrac-def cfrac-is-int-def* **by** *transfer auto*
qed

lemma *cfrac-coinduct* [*coinduct type*: *cfrac*]:
assumes *R c1 c2*
assumes *IH*: ⋀*c1 c2*. *R c1 c2* ⟹
cfrac-is-int c1 = cfrac-is-int c2 ∧
cfrac-nth c1 0 = cfrac-nth c2 0 ∧
(¬*cfrac-is-int c1* ⟶ ¬*cfrac-is-int c2* ⟶ *R* (*cfrac-tl c1*) (*cfrac-tl c2*))
shows *c1 = c2*
proof (*rule cfrac-eqI*)
show *cfrac-nth c1 n = cfrac-nth c2 n* **for** *n*
using *assms*(*1*)

```
  proof (induction n arbitrary: c1 c2)
    case 0
    from IH[OF this] show ?case
      by auto
  next
    case (Suc n)
    thus ?case
      using IH by (metis cfrac-is-int-iff cfrac-nth-0-of-int cfrac-nth-tl)
  qed
next
  show cfrac-length c1 = cfrac-length c2
    using assms(1)
  proof (coinduction arbitrary: c1 c2 rule: enat-coinduct)
    case (Eq-enat c1 c2)
    show ?case
    proof (cases cfrac-is-int c1)
      case True
      thus ?thesis
        using IH[OF Eq-enat(1)] by (auto simp: cfrac-is-int-def)
    next
      case False
     with IH[OF Eq-enat(1)] have **: ¬cfrac-is-int c1 R (cfrac-tl c1) (cfrac-tl c2)
        by auto
      have *: (cfrac-length c1 = 0) = (cfrac-length c2 = 0)
        using IH[OF Eq-enat(1)] by (auto simp: cfrac-is-int-def)
      show ?thesis
        by (intro conjI impI disjI1 *, rule exI[of - cfrac-tl c1], rule exI[of - cfrac-tl
c2])
           (use ** in ‹auto simp: epred-conv-minus›)
    qed
  qed
qed

lemma cfrac-nth-0-cases:
  cfrac-nth c 0 = ⌊cfrac-lim c⌋ ∨ cfrac-nth c 0 = ⌊cfrac-lim c⌋ − 1 ∧ cfrac-tl c
= 1
proof (cases cfrac-is-int c)
  case True
  hence cfrac-nth c 0 = ⌊cfrac-lim c⌋
    by (auto simp: cfrac-lim-def cfrac-is-int-def)
  thus ?thesis by blast
next
  case False
  note not-int = this
  have bounds: 1 / cfrac-lim (cfrac-tl c) ≥ 0 ∧ 1 / cfrac-lim (cfrac-tl c) ≤ 1
  proof −
    have 1 ≤ cfrac-nth (cfrac-tl c) 0
      by simp
    also have . . . ≤ cfrac-lim (cfrac-tl c)
```

by (*rule cfrac-lim-ge-first*)
finally show *?thesis*
using *False* **by** (*auto simp*: *cfrac-lim-nonneg*)
qed

thus *?thesis*
proof (*cases cfrac-lim* (*cfrac-tl c*) = *1*)
case *False*
have $\lfloor$*cfrac-lim c*$\rfloor$ = *cfrac-nth c 0* + $\lfloor$*1* / *cfrac-lim* (*cfrac-tl c*)$\rfloor$
using *not-int* **by** (*subst cfrac-lim-reduce*) *auto*
also have *1* / *cfrac-lim* (*cfrac-tl c*) $\geq$ *0* $\wedge$ *1* / *cfrac-lim* (*cfrac-tl c*) < *1*
using *bounds False* **by** (*auto simp*: *divide-simps*)
hence $\lfloor$*1* / *cfrac-lim* (*cfrac-tl c*)$\rfloor$ = *0*
by *linarith*
finally show *?thesis* **by** *simp*
next
case *True*
have *cfrac-nth c 0* = $\lfloor$*cfrac-lim c*$\rfloor$ − *1*
using *not-int True* **by** (*subst cfrac-lim-reduce*) *auto*
moreover have *cfrac-tl c* = *1*
using *True* **by** (*intro cfrac-eq-1I*) *auto*
ultimately show *?thesis* **by** *blast*
qed
qed

lemma *cfrac-length-1* [*simp*]: *cfrac-length 1* = *0*
unfolding *one-cfrac-def* **by** *simp*

lemma *cfrac-nth-1* [*simp*]: *cfrac-nth 1 m* = *1*
unfolding *one-cfrac-def* **by** *transfer* (*auto simp*: *enat-0-iff*)

lemma *cfrac-lim-1* [*simp*]: *cfrac-lim 1* = *1*
by (*auto simp*: *cfrac-lim-def*)

lemma *cfrac-nth-0-not-int*:
assumes *cfrac-lim c* $\notin \mathbb{Z}$
shows *cfrac-nth c 0* = $\lfloor$*cfrac-lim c*$\rfloor$
proof −
have *cfrac-tl c* $\neq$ *1*
proof
assume *eq*: *cfrac-tl c* = *1*
have ¬*cfrac-is-int c*
using *assms* **by** (*auto simp*: *cfrac-lim-def cfrac-is-int-def*)
hence *cfrac-lim c* = *of-int* $\lfloor$*cfrac-nth c 0*$\rfloor$ + *1*
using *eq* **by** (*subst cfrac-lim-reduce*) *auto*
hence *cfrac-lim c* $\in \mathbb{Z}$
by *auto*
with *assms* **show** *False* **by** *auto*

qed
with *cfrac-nth-0-cases*[*of c*] **show** *?thesis* **by** *auto*
qed

lemma *cfrac-of-real-cfrac-lim-irrational*:
assumes *cfrac-lim c* $\notin \mathbb{Q}$
shows *cfrac-of-real* (*cfrac-lim c*) = *c*
proof (*rule cfrac-eqI*)
from *assms* **show** *cfrac-length* (*cfrac-of-real* (*cfrac-lim c*)) = *cfrac-length c*
using *cfrac-lim-rational-iff* **by** *auto*
next
fix *n*
show *cfrac-nth* (*cfrac-of-real* (*cfrac-lim c*)) *n* = *cfrac-nth c n*
using *assms*
proof (*induction n arbitrary*: *c*)
case (*0 c*)
thus *?case*
using *Ints-subset-Rats* **by** (*subst cfrac-nth-0-not-int*) *auto*
next
case (*Suc n c*)
from *Suc.prems* **have** [*simp*]: *cfrac-lim c* $\notin \mathbb{Z}$
using *Ints-subset-Rats* **by** *blast*
have *cfrac-nth* (*cfrac-of-real* (*cfrac-lim c*)) (*Suc n*) =
cfrac-nth (*cfrac-tl* (*cfrac-of-real* (*cfrac-lim c*))) *n*
by (*simp flip*: *cfrac-nth-tl*)
also have *cfrac-tl* (*cfrac-of-real* (*cfrac-lim c*)) = *cfrac-of-real* (*1* / *frac* (*cfrac-lim c*))
using *Suc.prems Ints-subset-Rats* **by** (*subst cfrac-tl-of-real*) *auto*
also have *1* / *frac* (*cfrac-lim c*) = *cfrac-lim* (*cfrac-tl c*)
using *Suc.prems* **by** (*subst cfrac-lim-tl*) (*auto simp*: *frac-def cfrac-is-int-def cfrac-nth-0-not-int*)
also have *cfrac-nth* (*cfrac-of-real* (*cfrac-lim* (*cfrac-tl c*))) *n* = *cfrac-nth c* (*Suc n*)
using *Suc.prems* **by** (*subst Suc.IH*) (*auto simp*: *cfrac-lim-rational-iff*)
finally show *?case* **.**
qed
qed

lemma *cfrac-eqI-first*:
assumes ¬*cfrac-is-int c* ¬*cfrac-is-int c'*
assumes *cfrac-nth c 0* = *cfrac-nth c' 0* **and** *cfrac-tl c* = *cfrac-tl c'*
shows *c* = *c'*
using *assms* **unfolding** *cfrac-is-int-def*
by *transfer* (*auto split*: *llist.splits*)

lemma *cfrac-is-int-of-real-iff*: *cfrac-is-int* (*cfrac-of-real x*) $\longleftrightarrow x \in \mathbb{Z}$
unfolding *cfrac-is-int-def* **by** *transfer* (*use frac-lt-1* **in** *auto*)

lemma *cfrac-not-is-int-of-real-alt*: ¬*cfrac-is-int* (*cfrac-of-real-alt x*)

unfolding *cfrac-is-int-def* **by** *transfer* (*auto simp*: *frac-lt-1*)

lemma *cfrac-tl-of-real-alt-of-int* [*simp*]: *cfrac-tl* (*cfrac-of-real-alt* (*of-int n*)) = *1*
unfolding *one-cfrac-def* **by** *transfer auto*

lemma *cfrac-is-intI*:
assumes *cfrac-nth c 0* ≥ ⌊*cfrac-lim c*⌋ **and** *cfrac-lim c* ∈ ℤ
shows *cfrac-is-int c*
proof (*rule ccontr*)
assume ∗: ¬*cfrac-is-int c*
from ∗ **have** *conv c 0* < *cfrac-lim c*
by (*intro conv-less-cfrac-lim*) (*auto simp*: *cfrac-is-int-def simp flip*: *zero-enat-def*)
with *assms* **show** *False*
by (*auto simp*: *Ints-def*)
qed

lemma *cfrac-eq-of-intI*:
assumes *cfrac-nth c 0* ≥ ⌊*cfrac-lim c*⌋ **and** *cfrac-lim c* ∈ ℤ
shows *c* = *cfrac-of-int* ⌊*cfrac-lim c*⌋
proof −
from *assms* **have** *int*: *cfrac-is-int c*
by (*intro cfrac-is-intI*) *auto*
have [*simp*]: *cfrac-lim c* = *cfrac-nth c 0*
using *int* **by** (*simp add*: *cfrac-lim-def cfrac-is-int-def*)
from *int* **have** *c* = *cfrac-of-int* (*cfrac-nth c 0*)
unfolding *cfrac-is-int-def* **by** *transfer auto*
also from *assms* **have** *cfrac-nth c 0* = ⌊*cfrac-lim c*⌋
using *int* **by** *auto*
finally show *?thesis* **.**
qed

lemma *cfrac-lim-of-int* [*simp*]: *cfrac-lim* (*cfrac-of-int n*) = *of-int n*
by (*simp add*: *cfrac-lim-def*)

lemma *cfrac-of-real-of-int* [*simp*]: *cfrac-of-real* (*of-int n*) = *cfrac-of-int n*
by *transfer auto*

lemma *cfrac-of-real-of-nat* [*simp*]: *cfrac-of-real* (*of-nat n*) = *cfrac-of-int* (*int n*)
by *transfer auto*

lemma *cfrac-int-cases*:
assumes *cfrac-lim c* = *of-int n*
shows *c* = *cfrac-of-int n* ∨ *c* = *cfrac-of-real-alt* (*of-int n*)
proof −
from *cfrac-nth-0-cases*[*of c*] **show** *?thesis*
proof (*rule disj-forward*)
assume *eq*: *cfrac-nth c 0* = ⌊*cfrac-lim c*⌋
have *c* = *cfrac-of-int* ⌊*cfrac-lim c*⌋
using *assms eq* **by** (*intro cfrac-eq-of-intI*) *auto*

with *assms eq* **show** *c = cfrac-of-int n*
by *simp*
next
assume ∗: *cfrac-nth c 0* = ⌊*cfrac-lim c*⌋ − *1* ∧ *cfrac-tl c* = *1*
have ¬*cfrac-is-int c*
using ∗ **by** (*auto simp*: *cfrac-is-int-def cfrac-lim-def*)
hence *cfrac-length c* = *eSuc* (*cfrac-length* (*cfrac-tl c*))
by (*subst cfrac-length-tl*; *cases cfrac-length c*)
(*auto simp*: *cfrac-is-int-def eSuc-def enat-0-iff split*: *enat.splits*)
also have *cfrac-tl c* = *1*
using ∗ **by** *auto*
finally have *cfrac-length c* = *1*
by (*simp add*: *eSuc-def one-enat-def*)
show *c* = *cfrac-of-real-alt* (*of-int n*)
by (*rule cfrac-eqI-first*)
(*use* ‹¬*cfrac-is-int c*› ∗ *assms* **in** ‹*auto simp*: *cfrac-not-is-int-of-real-alt*›)
qed
qed

lemma *cfrac-cases*:
c ∈ {*cfrac-of-real* (*cfrac-lim c*), *cfrac-of-real-alt* (*cfrac-lim c*)}
proof (*cases cfrac-length c*)
case *infinity*
hence *cfrac-lim c* ∉ ℚ
by (*simp add*: *cfrac-lim-irrational*)
thus *?thesis*
using *cfrac-of-real-cfrac-lim-irrational* **by** *simp*
next
case (*enat l*)
thus *?thesis*
proof (*induction l arbitrary*: *c*)
case (*0 c*)
hence *c* = *cfrac-of-real* (*cfrac-nth c 0*)
by *transfer* (*auto simp flip*: *zero-enat-def*)
with *0* **show** *?case* **by** (*auto simp*: *cfrac-lim-def*)
next
case (*Suc l c*)
show *?case*
proof (*cases cfrac-lim c* ∈ ℤ)
case *True*
thus *?thesis*
using *cfrac-int-cases*[*of c*] **by** (*force simp*: *Ints-def*)
next
case [*simp*]: *False*
have ¬*cfrac-is-int c*
using *Suc.prems* **by** (*auto simp*: *cfrac-is-int-def enat-0-iff*)
show *?thesis*
using *cfrac-nth-0-cases*[*of c*]
proof (*elim disjE conjE*)

```
        assume *: cfrac-nth c 0 = ⌊cfrac-lim c⌋ − 1 cfrac-tl c = 1
        hence cfrac-lim c ∈ ℤ
          using ‹¬cfrac-is-int c› by (subst cfrac-lim-reduce) auto
        thus ?thesis
          by (auto simp: cfrac-int-cases)
      next
        assume eq: cfrac-nth c 0 = ⌊cfrac-lim c⌋
        have cfrac-tl c = cfrac-of-real (cfrac-lim (cfrac-tl c)) ∨
              cfrac-tl c = cfrac-of-real-alt (cfrac-lim (cfrac-tl c))
          using Suc.IH[of cfrac-tl c] Suc.prems by auto
        hence c = cfrac-of-real (cfrac-lim c) ∨
               c = cfrac-of-real-alt (cfrac-lim c)
        proof (rule disj-forward)
          assume eq′: cfrac-tl c = cfrac-of-real (cfrac-lim (cfrac-tl c))
          show c = cfrac-of-real (cfrac-lim c)
            by (rule cfrac-eqI-first)
               (use ‹¬cfrac-is-int c› eq eq′ in
               ‹auto simp: cfrac-is-int-of-real-iff cfrac-tl-of-real cfrac-lim-tl frac-def›)
        next
          assume eq′: cfrac-tl c = cfrac-of-real-alt (cfrac-lim (cfrac-tl c))
          have eq′′: cfrac-nth (cfrac-of-real-alt (cfrac-lim c)) 0 = ⌊cfrac-lim c⌋
            using Suc.prems by (subst cfrac-nth-of-real-alt-0) auto
          show c = cfrac-of-real-alt (cfrac-lim c)
            by (rule cfrac-eqI-first)
               (use ‹¬cfrac-is-int c› eq eq′ eq′′ in
                  ‹auto simp: cfrac-not-is-int-of-real-alt cfrac-tl-of-real-alt cfrac-lim-tl
frac-def›)
        qed
        thus ?thesis by simp
      qed
    qed
  qed
qed

lemma cfrac-lim-eq-iff:
  assumes cfrac-length c = ∞ ∨ cfrac-length c′ = ∞
  shows   cfrac-lim c = cfrac-lim c′ ⟷ c = c′
proof
  assume *: cfrac-lim c = cfrac-lim c′
  hence cfrac-of-real (cfrac-lim c) = cfrac-of-real (cfrac-lim c′)
    by (simp only:)
  thus c = c′
    using assms *
    by (subst (asm) (1 2) cfrac-of-real-cfrac-lim-irrational)
       (auto simp: cfrac-infinite-iff)
qed auto

lemma floor-cfrac-remainder:
  assumes cfrac-length c = ∞
```

shows $\lfloor$*cfrac-remainder c n*$\rfloor$ = *cfrac-nth c n*
by (*metis add.left-neutral assms cfrac-length-drop cfrac-lim-eq-iff idiff-infinity cfrac-lim-of-real cfrac-nth-drop cfrac-nth-of-real-0 cfrac-remainder-def*)

1.4 Approximation properties

In this section, we will show that convergents of the continued fraction expansion of a number x are good approximations of x, and in a certain sense, the reverse holds as well.

lemma *sgn-of-int*:
sgn (*of-int x* :: *'a* :: {*linordered-idom*}) = *of-int* (*sgn x*)
by (*auto simp*: *sgn-if*)

lemma *conv-ge-one*: *cfrac-nth c 0* > *0* $\Longrightarrow$ *conv c n* $\geq$ *1*
by (*rule order.trans*[*OF - conv-ge-first*]) *auto*

context
fixes *c h k*
defines *h* $\equiv$ *conv-num c* **and** *k* $\equiv$ *conv-denom c*
begin

lemma *abs-diff-le-abs-add*:
fixes *x y* :: *real*
assumes $x \geq 0 \wedge y \geq 0 \vee x \leq 0 \wedge y \leq 0$
shows $|x - y| \leq |x + y|$
using *assms* **by** *linarith*

lemma *abs-diff-less-abs-add*:
fixes *x y* :: *real*
assumes $x > 0 \wedge y > 0 \vee x < 0 \wedge y < 0$
shows $|x - y| < |x + y|$
using *assms* **by** *linarith*

lemma *abs-diff-le-imp-same-sign*:
assumes $|x - y| \leq d$ $d < |y|$
shows *sgn x* = *sgn* (*y*::*real*)
using *assms* **by** (*auto simp*: *sgn-if*)

lemma *conv-nonpos*:
assumes *cfrac-nth c 0* < *0*
shows *conv c n* $\leq$ *0*
proof (*cases n*)
case *0*
thus *?thesis* **using** *assms* **by** *auto*
next
case [*simp*]: (*Suc n'*)
have *conv c n* = *real-of-int* (*cfrac-nth c 0*) + *1* / *conv* (*cfrac-tl c*) *n'*
by (*simp add*: *conv-Suc*)
also have ... $\leq -1 + 1 / 1$

using *assms* **by** (*intro add-mono divide-left-mono*) (*auto intro*!: *conv-pos conv-ge-one*)
finally show *?thesis* **by** *simp*
qed

lemma *cfrac-lim-nonpos*:
assumes *cfrac-nth c 0 < 0*
shows *cfrac-lim c ≤ 0*
proof (*cases cfrac-length c*)
case *infinity*
show *?thesis* **using** *LIMSEQ-cfrac-lim*[*OF infinity*]
by (*rule tendsto-upperbound*) (*use assms* **in** ‹*auto simp*: *conv-nonpos*›)
next
case (*enat l*)
thus *?thesis* **by** (*auto simp*: *cfrac-lim-def conv-nonpos assms*)
qed

lemma *conv-num-nonpos*:
assumes *cfrac-nth c 0 < 0*
shows *h n ≤ 0*
proof (*induction n rule*: *fib.induct*)
case *2*
have *cfrac-nth c* (*Suc 0*) ∗ *cfrac-nth c 0 ≤ 1* ∗ *cfrac-nth c 0*
using *assms* **by** (*intro mult-right-mono-neg*) *auto*
also have ... + *1 ≤ 0* **using** *assms* **by** *auto*
finally show *?case* **by** (*auto simp*: *h-def*)
next
case (*3 n*)
have *cfrac-nth c* (*Suc* (*Suc n*)) ∗ *h* (*Suc n*) ≤ *0*
using *3* **by** (*simp add*: *mult-nonneg-nonpos*)
also have ... + *h n ≤ 0*
using *3* **by** *simp*
finally show *?case*
by (*auto simp*: *h-def*)
qed (*use assms* **in** ‹*auto simp*: *h-def*›)

lemma *conv-best-approximation-aux*:
cfrac-lim c ≥ 0 ∧ h n ≥ 0 ∨ cfrac-lim c ≤ 0 ∧ h n ≤ 0
proof (*cases cfrac-nth c 0 ≥ 0*)
case *True*
from *True* **have** *0 ≤ conv c 0*
by *simp*
also have ... ≤ *cfrac-lim c*
by (*rule conv-le-cfrac-lim*) (*auto simp*: *enat-0*)
finally have *cfrac-lim c ≥ 0* **.**
moreover from *True* **have** *h n ≥ 0*
unfolding *h-def* **by** (*intro conv-num-nonneg*)
ultimately show *?thesis* **by** (*simp add*: *sgn-if*)
next

case *False*
thus *?thesis*
using *cfrac-lim-nonpos conv-num-nonpos*[*of n*] **by** (*auto simp*: *h-def*)
qed

lemma *conv-best-approximation-ex*:
fixes *a b* :: *int* **and** *x* :: *real*
assumes $n \le$ *cfrac-length c*
assumes $0 < b$ **and** $b \le k\ n$ **and** *coprime a b* **and** $n > 0$
assumes $(a, b) \ne (h\ n, k\ n)$
assumes ¬(*cfrac-length c* = *1* ∧ *n* = *0*)
assumes *Suc n* ≠ *cfrac-length c* ∨ *cfrac-canonical c*
defines *x* ≡ *cfrac-lim c*
shows $|k\ n * x - h\ n| < |b * x - a|$
proof (*cases* $|a| = |h\ n| \land b = k\ n$)
case *True*
with *assms* **have** [*simp*]: $a = -h\ n$
by (*auto simp*: *abs-if split*: *if-splits*)
have $k\ n > 0$
by (*auto simp*: *k-def*)
show *?thesis*
proof (*cases* $x = 0$)
case *True*
hence *c* = *cfrac-of-real 0* ∨ *c* = *cfrac-of-real-alt 0*
unfolding *x-def* **by** (*metis cfrac-cases empty-iff insert-iff*)
hence *False*
proof
assume *c* = *cfrac-of-real 0*
thus *False*
using *assms* **by** (*auto simp*: *enat-0-iff h-def k-def*)
next
assume [*simp*]: *c* = *cfrac-of-real-alt 0*
hence $n = 0 \lor n = 1$
using *assms* **by** (*auto simp*: *cfrac-length-of-real-alt enat-0-iff k-def h-def eSuc-def*)
thus *False*
using *assms True*
by (*elim disjE*) (*auto simp*: *cfrac-length-of-real-alt enat-0-iff k-def h-def eSuc-def*
cfrac-nth-of-real-alt one-enat-def split: *if-splits*)
qed
thus *?thesis* **..**
next
case *False*
have $h\ n \ne 0$
using *True assms*(*6*) *h-def* **by** *auto*
hence $x > 0 \land h\ n > 0 \lor x < 0 \land h\ n < 0$
using ‹$x \ne 0$› *conv-best-approximation-aux*[*of n*] **unfolding** *x-def* **by** *auto*
hence |*real-of-int* (*k n*) ∗ *x* − *real-of-int* (*h n*)| < |*real-of-int* (*k n*) ∗ *x* +

```
real-of-int (h n)|
      using ‹k n > 0›
    by (intro abs-diff-less-abs-add) (auto simp: not-le zero-less-mult-iff mult-less-0-iff)
    thus ?thesis using True by auto
  qed
next
  case False
  note * = this
  show ?thesis
  proof (cases n = cfrac-length c)
    case True
    hence x = conv c n
      by (auto simp: cfrac-lim-def x-def split: enat.splits)
    also have ... = h n / k n
      by (auto simp: h-def k-def conv-num-denom)
    finally have x: x = h n / k n .
    hence |k n * x − h n| = 0
      by (simp add: k-def)
    also have b * x ≠ a
    proof
      assume b * x = a
      hence of-int (h n) * of-int b = of-int (k n) * (of-int a :: real)
        using assms True by (auto simp: field-simps k-def x)
      hence of-int (h n * b) = (of-int (k n * a) :: real)
        by (simp only: of-int-mult)
      hence h n * b = k n * a
        by linarith
      hence h n = a ∧ k n = b
        using assms by (subst (asm) coprime-crossproduct')
                    (auto simp: h-def k-def coprime-conv-num-denom)
      thus False using True False by simp
    qed
    hence 0 < |b * x − a|
      by simp
    finally show ?thesis .
  next
    case False

    define s where s = (−1) ^ n * (a * k n − b * h n)
    define r where r = (−1) ^ n * (b * h (Suc n) − a * k (Suc n))
    have k n ≤ k (Suc n)
      unfolding k-def by (intro conv-denom-leI) auto

    have r * h n + s * h (Suc n) =
            (−1) ^ Suc n * a * (k (Suc n) * h n − k n * h (Suc n))
      by (simp add: s-def r-def algebra-simps h-def k-def)
    also have ... = a using assms unfolding h-def k-def
      by (subst conv-num-denom-prod-diff') (auto simp: algebra-simps)
    finally have eq1: r * h n + s * h (Suc n) = a .
```

have $r * k\ n + s * k\ (Suc\ n) =$
(-1) ^ $Suc\ n * b * (k\ (Suc\ n) * h\ n - k\ n * h\ (Suc\ n))$
by (*simp add*: *s-def r-def algebra-simps h-def k-def*)
also have ... $= b$ **using** *assms* **unfolding** *h-def k-def*
by (*subst conv-num-denom-prod-diff'*) (*auto simp*: *algebra-simps*)
finally have *eq2*: $r * k\ n + s * k\ (Suc\ n) = b$ **.**

have $k\ n < k\ (Suc\ n)$
using ‹$n > 0$› **by** (*auto simp*: *k-def intro*: *conv-denom-lessI*)

have $r \neq 0$
proof
assume $r = 0$
hence $a * k\ (Suc\ n) = b * h\ (Suc\ n)$ **by** (*simp add*: *r-def*)
hence $abs\ (a * k\ (Suc\ n)) = abs\ (h\ (Suc\ n) * b)$ **by** (*simp only*: *mult-ac*)
hence $*$: $abs\ (h\ (Suc\ n)) = abs\ a \wedge k\ (Suc\ n) = b$
unfolding *abs-mult h-def k-def* **using** *coprime-conv-num-denom assms*
by (*subst* (*asm*) *coprime-crossproduct-int*) *auto*
with ‹$k\ n < k\ (Suc\ n)$› **and** ‹$b \leq k\ n$› **show** *False* **by** *auto*
qed

have $s \neq 0$
proof
assume $s = 0$
hence $a * k\ n = b * h\ n$ **by** (*simp add*: *s-def*)
hence $abs\ (a * k\ n) = abs\ (h\ n * b)$ **by** (*simp only*: *mult-ac*)
hence $b = k\ n \wedge |a| = |h\ n|$ **unfolding** *abs-mult h-def k-def* **using** *coprime-conv-num-denom assms*
by (*subst* (*asm*) *coprime-crossproduct-int*) *auto*
with $*$ **show** *False* **by** *simp*
qed

have $r * k\ n + s * k\ (Suc\ n) = b$ **by** *fact*
also have ... $\in \{0<..<k\ (Suc\ n)\}$ **using** *assms* ‹$k\ n < k\ (Suc\ n)$› **by** *auto*
finally have $*$: $r * k\ n + s * k\ (Suc\ n) \in$ ... **.**

have *opposite-signs1*: $r > 0 \wedge s < 0 \vee r < 0 \wedge s > 0$
proof (*cases* $r \geq 0$; *cases* $s \geq 0$)
assume $r \geq 0\ s \geq 0$
hence $0 * (k\ n) + 1 * (k\ (Suc\ n)) \leq r * k\ n + s * k\ (Suc\ n)$
using ‹$s \neq 0$› **by** (*intro add-mono mult-mono*) (*auto simp*: *k-def*)
with $*$ **show** *?thesis* **by** *auto*
next
assume $\neg(r \geq 0)\ \neg(s \geq 0)$
hence $r * k\ n + s * k\ (Suc\ n) \leq 0$
by (*intro add-nonpos-nonpos mult-nonpos-nonneg*) (*auto simp*: *k-def*)
with $*$ **show** *?thesis* **by** *auto*
qed (*insert* ‹$r \neq 0$› ‹$s \neq 0$›, *auto*)

```
  have r ≠ 1
  proof
    assume [simp]: r = 1
    have b = r * k n + s * k (Suc n)
      using ‹r * k n + s * k (Suc n) = b› ..
    also have s * k (Suc n) ≤ (−1) * k (Suc n)
      using opposite-signs1 by (intro mult-right-mono) (auto simp: k-def)
    also have r * k n + (−1) * k (Suc n) = k n − k (Suc n)
      by simp
    also have . . . ≤ 0
      unfolding k-def by (auto intro!: conv-denom-leI)
    finally show False using ‹b > 0› by simp
  qed

  have enat n ≤ cfrac-length c enat (Suc n) ≤ cfrac-length c
    using assms False by (cases cfrac-length c; simp)+
  hence conv c n ≥ x ∧ conv c (Suc n) ≤ x ∨ conv c n ≤ x ∧ conv c (Suc n) ≥ x
    using conv-ge-cfrac-lim[of n c] conv-ge-cfrac-lim[of Suc n c]
          conv-le-cfrac-lim[of n c] conv-le-cfrac-lim[of Suc n c] assms
    by (cases even n) auto
  hence opposite-signs2: k n * x − h n ≥ 0 ∧ k (Suc n) * x − h (Suc n) ≤ 0 ∨
                         k n * x − h n ≤ 0 ∧ k (Suc n) * x − h (Suc n) ≥ 0
    using assms conv-denom-pos[of c n] conv-denom-pos[of c Suc n]
    by (auto simp: k-def h-def conv-num-denom field-simps)

  from opposite-signs1 opposite-signs2 have same-signs:
    r * (k n * x − h n) ≥ 0 ∧ s * (k (Suc n) * x − h (Suc n)) ≥ 0 ∨
     r * (k n * x − h n) ≤ 0 ∧ s * (k (Suc n) * x − h (Suc n)) ≤ 0
    by (auto intro: mult-nonpos-nonneg mult-nonneg-nonpos mult-nonneg-nonneg
mult-nonpos-nonpos)

  show ?thesis
  proof (cases Suc n = cfrac-length c)
    case True
    have x: x = h (Suc n) / k (Suc n)
    using True[symmetric] by (auto simp: cfrac-lim-def h-def k-def conv-num-denom
x-def)
    have r ≠ −1
    proof
      assume [simp]: r = −1
      have r * k n + s * k (Suc n) = b
        by fact
      also have b < k (Suc n)
        using ‹b ≤ k n› and ‹k n < k (Suc n)› by simp
      finally have (s − 1) * k (Suc n) < k n
        by (simp add: algebra-simps)
      also have k n ≤ 1 * k (Suc n)
        by (simp add: k-def conv-denom-leI)
```

finally have *s* < *2*
  by (*subst* (*asm*) *mult-less-cancel-right*) (*auto simp*: *k-def*)
moreover from *opposite-signs1* **have** *s* > *0* **by** *auto*
ultimately have [*simp*]: *s* = *1* **by** *simp*

have *b* = (*cfrac-nth c* (*Suc n*) − *1*) ∗ *k n* + *k* (*n* − *1*)
  using *eq2* ‹*n* > *0*› **by** (*cases n*) (*auto simp*: *k-def algebra-simps*)
also have *cfrac-nth c* (*Suc n*) > *1*
proof −
  have *cfrac-canonical c*
    using *assms True* **by** *auto*
  hence *cfrac-nth c* (*Suc n*) ≠ *1*
    using *True*[*symmetric*] **by** (*auto simp*: *cfrac-canonical-iff enat-0-iff*)
  moreover have *cfrac-nth c* (*Suc n*) > *0*
    by *auto*
  ultimately show *cfrac-nth c* (*Suc n*) > *1*
    by *linarith*
qed
hence (*cfrac-nth c* (*Suc n*) − *1*) ∗ *k n* + *k* (*n* − *1*) ≥ *1* ∗ *k n* + *k* (*n* − *1*)
  by (*intro add-mono mult-right-mono*) (*auto simp*: *k-def*)
finally have *b* > *k n*
  using *conv-denom-pos*[*of c n* − *1*] **unfolding** *k-def* **by** *linarith*
with *assms* **show** *False* **by** *simp*
qed
with ‹*r* ≠ *1*› ‹*r* ≠ *0*› **have** |*r*| > *1*
  by *auto*

from ‹*s* ≠ *0*› **have** *k n* ∗ *x* ≠ *h n*
  using *conv-num-denom-prod-diff*[*of c n*]
  by (*auto simp*: *x field-simps k-def h-def simp flip*: *of-int-mult*)
hence *1* ∗ |*k n* ∗ *x* − *h n*| < |*r*| ∗ |*k n* ∗ *x* − *h n*|
  using ‹|*r*| > *1*› **by** (*intro mult-strict-right-mono*) *auto*
also have ... = |*r*| ∗ |*k n* ∗ *x* − *h n*| + *0* **by** *simp*
also have ... ≤ |*r* ∗ (*k n* ∗ *x* − *h n*)| + |*s* ∗ (*k* (*Suc n*) ∗ *x* − *h* (*Suc n*))|
  unfolding *abs-mult of-int-abs* **using** *conv-denom-pos*[*of c Suc n*] ‹*s* ≠ *0*›
  by (*intro add-left-mono mult-nonneg-nonneg*) (*auto simp*: *field-simps k-def*)
also have ... = |*r* ∗ (*k n* ∗ *x* − *h n*) + *s* ∗ (*k* (*Suc n*) ∗ *x* − *h* (*Suc n*))|
  using *same-signs* **by** *auto*
also have ... = |(*r* ∗ *k n* + *s* ∗ *k* (*Suc n*)) ∗ *x* − (*r* ∗ *h n* + *s* ∗ *h* (*Suc n*))|
  by (*simp add*: *algebra-simps*)
also have ... = |*b* ∗ *x* − *a*|
  unfolding *eq1 eq2* **by** *simp*
finally show *?thesis* **by** *simp*
next
  case *False*
  from *assms* **have** *Suc n* < *cfrac-length c*
    using *False* ‹*Suc n* ≤ *cfrac-length c*› **by** *force*
  have *1* ∗ |*k n* ∗ *x* − *h n*| ≤ |*r*| ∗ |*k n* ∗ *x* − *h n*|
    using ‹*r* ≠ *0*› **by** (*intro mult-right-mono*) *auto*

also have ... = $|r| * |k\ n * x - h\ n| + 0$ **by** *simp*
also have $x \neq h\ (Suc\ n)\ /\ k\ (Suc\ n)$
using *conv-neq-cfrac-lim*[*of Suc n c*] ‹*Suc n* < *cfrac-length c*›
by (*auto simp*: *conv-num-denom h-def k-def x-def*)
hence $|s * (k\ (Suc\ n) * x - h\ (Suc\ n))| > 0$
using ‹$s \neq 0$› **by** (*auto simp*: *field-simps k-def*)
also have $|r| * |k\ n * x - h\ n| + \ldots \leq$
$|r * (k\ n * x - h\ n)| + |s * (k\ (Suc\ n) * x - h\ (Suc\ n))|$
unfolding *abs-mult of-int-abs* **by** (*intro add-left-mono mult-nonneg-nonneg*) *auto*
also have ... = $|r * (k\ n * x - h\ n) + s * (k\ (Suc\ n) * x - h\ (Suc\ n))|$
using *same-signs* **by** *auto*
also have ... = $|(r * k\ n + s * k\ (Suc\ n)) * x - (r * h\ n + s * h\ (Suc\ n))|$
by (*simp add*: *algebra-simps*)
also have ... = $|b * x - a|$
unfolding *eq1 eq2* **by** *simp*
finally show *?thesis* **by** *simp*
qed
qed
qed

lemma *conv-best-approximation-ex-weak*:
fixes *a b* :: *int* **and** *x* :: *real*
assumes $n \leq$ *cfrac-length c*
assumes $0 < b$ **and** $b < k\ (Suc\ n)$ **and** *coprime a b*
defines $x \equiv$ *cfrac-lim c*
shows $|k\ n * x - h\ n| \leq |b * x - a|$
proof (*cases* $|a| = |h\ n| \wedge b = k\ n$)
case *True*
note * = *this*
show *?thesis*
proof (*cases sgn a* = *sgn* (*h n*))
case *True*
with * **have** [*simp*]: $a = h\ n$
by (*auto simp*: *abs-if split*: *if-splits*)
thus *?thesis* **using** * **by** *auto*
next
case *False*
with *True* **have** [*simp*]: $a = -h\ n$
by (*auto simp*: *abs-if split*: *if-splits*)
have |*real-of-int* (*k n*) * *x* − *real-of-int* (*h n*)| ≤ |*real-of-int* (*k n*) * *x* + *real-of-int* (*h n*)|
unfolding *x-def* **using** *conv-best-approximation-aux*[*of n*]
by (*intro abs-diff-le-abs-add*) (*auto simp*: *k-def not-le zero-less-mult-iff*)
thus *?thesis* **using** *True* **by** *auto*
qed
next
case *False*
note * = *this*

```
  show ?thesis
  proof (cases n = cfrac-length c)
    case True
    hence x = conv c n
      by (auto simp: cfrac-lim-def x-def split: enat.splits)
    also have ... = h n / k n
      by (auto simp: h-def k-def conv-num-denom)
    finally show ?thesis by (auto simp: k-def)
  next
    case False

    define s where s = (−1) ^ n ∗ (a ∗ k n − b ∗ h n)
    define r where r = (−1) ^ n ∗ (b ∗ h (Suc n) − a ∗ k (Suc n))

    have r ∗ h n + s ∗ h (Suc n) =
            (−1) ^ Suc n ∗ a ∗ (k (Suc n) ∗ h n − k n ∗ h (Suc n))
      by (simp add: s-def r-def algebra-simps h-def k-def)
    also have ... = a using assms unfolding h-def k-def
      by (subst conv-num-denom-prod-diff') (auto simp: algebra-simps)
    finally have eq1: r ∗ h n + s ∗ h (Suc n) = a .

    have r ∗ k n + s ∗ k (Suc n) =
            (−1) ^ Suc n ∗ b ∗ (k (Suc n) ∗ h n − k n ∗ h (Suc n))
      by (simp add: s-def r-def algebra-simps h-def k-def)
    also have ... = b using assms unfolding h-def k-def
      by (subst conv-num-denom-prod-diff') (auto simp: algebra-simps)
    finally have eq2: r ∗ k n + s ∗ k (Suc n) = b .

    have r ≠ 0
    proof
      assume r = 0
      hence a ∗ k (Suc n) = b ∗ h (Suc n) by (simp add: r-def)
      hence abs (a ∗ k (Suc n)) = abs (h (Suc n) ∗ b) by (simp only: mult-ac)
    hence b = k (Suc n) unfolding abs-mult h-def k-def using coprime-conv-num-denom
assms
        by (subst (asm) coprime-crossproduct-int) auto
      with assms show False by simp
    qed

    have s ≠ 0
    proof
      assume s = 0
      hence a ∗ k n = b ∗ h n by (simp add: s-def)
      hence abs (a ∗ k n) = abs (h n ∗ b) by (simp only: mult-ac)
        hence b = k n ∧ |a| = |h n| unfolding abs-mult h-def k-def using co-
prime-conv-num-denom assms
        by (subst (asm) coprime-crossproduct-int) auto
      with ∗ show False by simp
    qed
```

```
    have r * k n + s * k (Suc n) = b by fact
    also have ... ∈ {0<..<k (Suc n)} using assms by auto
    finally have *: r * k n + s * k (Suc n) ∈ ... .

    have opposite-signs1: r > 0 ∧ s < 0 ∨ r < 0 ∧ s > 0
    proof (cases r ≥ 0; cases s ≥ 0)
      assume r ≥ 0 s ≥ 0
      hence 0 * (k n) + 1 * (k (Suc n)) ≤ r * k n + s * k (Suc n)
        using ‹s ≠ 0› by (intro add-mono mult-mono) (auto simp: k-def)
      with * show ?thesis by auto
    next
      assume ¬(r ≥ 0) ¬(s ≥ 0)
      hence r * k n + s * k (Suc n) ≤ 0
        by (intro add-nonpos-nonpos mult-nonpos-nonneg) (auto simp: k-def)
      with * show ?thesis by auto
    qed (insert ‹r ≠ 0› ‹s ≠ 0›, auto)

    have enat n ≤ cfrac-length c enat (Suc n) ≤ cfrac-length c
      using assms False by (cases cfrac-length c; simp)+
    hence conv c n ≥ x ∧ conv c (Suc n) ≤ x ∨ conv c n ≤ x ∧ conv c (Suc n) ≥ x
      using conv-ge-cfrac-lim[of n c] conv-ge-cfrac-lim[of Suc n c]
            conv-le-cfrac-lim[of n c] conv-le-cfrac-lim[of Suc n c] assms
      by (cases even n) auto
    hence opposite-signs2: k n * x − h n ≥ 0 ∧ k (Suc n) * x − h (Suc n) ≤ 0 ∨
                           k n * x − h n ≤ 0 ∧ k (Suc n) * x − h (Suc n) ≥ 0
      using assms conv-denom-pos[of c n] conv-denom-pos[of c Suc n]
      by (auto simp: k-def h-def conv-num-denom field-simps)

    from opposite-signs1 opposite-signs2 have same-signs:
      r * (k n * x − h n) ≥ 0 ∧ s * (k (Suc n) * x − h (Suc n)) ≥ 0 ∨
       r * (k n * x − h n) ≤ 0 ∧ s * (k (Suc n) * x − h (Suc n)) ≤ 0
      by (auto intro: mult-nonpos-nonneg mult-nonneg-nonpos mult-nonneg-nonneg
mult-nonpos-nonpos)

    have 1 * |k n * x − h n| ≤ |r| * |k n * x − h n|
      using ‹r ≠ 0› by (intro mult-right-mono) auto
    also have ... = |r| * |k n * x − h n| + 0 by simp
    also have ... ≤ |r * (k n * x − h n)| + |s * (k (Suc n) * x − h (Suc n))|
      unfolding abs-mult of-int-abs using conv-denom-pos[of c Suc n] ‹s ≠ 0›
      by (intro add-left-mono mult-nonneg-nonneg) (auto simp: field-simps k-def)
    also have ... = |r * (k n * x − h n) + s * (k (Suc n) * x − h (Suc n))|
      using same-signs by auto
    also have ... = |(r * k n + s * k (Suc n)) * x − (r * h n + s * h (Suc n))|
      by (simp add: algebra-simps)
    also have ... = |b * x − a|
      unfolding eq1 eq2 by simp
    finally show ?thesis by simp
  qed
```

qed

lemma *cfrac-canonical-reduce*:
  cfrac-canonical c ⟷
    cfrac-is-int c ∨ ¬*cfrac-is-int c* ∧ *cfrac-tl c* ≠ *1* ∧ *cfrac-canonical* (*cfrac-tl c*)
  unfolding *cfrac-is-int-def one-cfrac-def*
  by *transfer* (*auto simp*: *cfrac-canonical-def llast-LCons split*: *if-splits split*: *llist.splits*)

lemma *cfrac-nth-0-conv-floor*:
  assumes *cfrac-canonical c* ∨ *cfrac-length c* ≠ *1*
  shows *cfrac-nth c 0* = ⌊*cfrac-lim c*⌋
proof (*cases cfrac-is-int c*)
  case *True*
  thus *?thesis*
    by (*auto simp*: *cfrac-lim-def cfrac-is-int-def*)
next
  case *False*
  show *?thesis*
  proof (*cases cfrac-length c* = *1*)
    case *True*
    hence *cfrac-canonical c* **using** *assms* **by** *auto*
    hence *cfrac-tl c* ≠ *1* **using** *False*
      by (*subst* (*asm*) *cfrac-canonical-reduce*) *auto*
    thus *?thesis*
      using *cfrac-nth-0-cases*[*of c*] **by** *auto*
  next
    case *False*
    hence *cfrac-length c* > *1*
      using ‹¬*cfrac-is-int c*›
      by (*cases cfrac-length c*) (*auto simp*: *cfrac-is-int-def one-enat-def zero-enat-def*)
    have *cfrac-tl c* ≠ *1*
    proof
      assume *cfrac-tl c* = *1*
      have *0* < *cfrac-length c* − *1*
      proof (*cases cfrac-length c*)
        case [*simp*]: (*enat l*)
        have *cfrac-length c* − *1* = *enat* (*l* − *1*)
          by *auto*
        also have ... > *enat 0*
          using ‹*cfrac-length c* > *1*› **by** (*simp add*: *one-enat-def*)
        finally show *?thesis* **by** (*simp add*: *zero-enat-def*)
      qed *auto*
      also have ... = *cfrac-length* (*cfrac-tl c*)
        by *simp*
      also have *cfrac-tl c* = *1*
        by *fact*
      finally show *False* **by** *simp*
    qed
    thus *?thesis* **using** *cfrac-nth-0-cases*[*of c*] **by** *auto*

qed
qed

lemma *conv-best-approximation-ex-nat*:
fixes *a b* :: *nat* **and** *x* :: *real*
assumes $n \leq$ *cfrac-length c* $0 < b$ $b < k$ (*Suc n*) *coprime a b*
shows $|k\ n * \textit{cfrac-lim}\ c - h\ n| \leq |b * \textit{cfrac-lim}\ c - a|$
using *conv-best-approximation-ex-weak*[*OF assms*(*1*), *of b a*] *assms* **by** *auto*

lemma *abs-mult-nonneg-left*:
assumes $x \geq (0 :: 'a :: \{\textit{ordered-ab-group-add-abs}, \textit{idom-abs-sgn}\})$
shows $x * |y| = |x * y|$
proof −
from *assms* **have** $x = |x|$ **by** *simp*
also have $\ldots * |y| = |x * y|$ **by** (*simp add*: *abs-mult*)
finally show *?thesis* **.**
qed

Any convergent of the continued fraction expansion of x is a best approximation of x, i.e. there is no other number with a smaller denominator that approximates it better.

lemma *conv-best-approximation*:
fixes *a b* :: *int* **and** *x* :: *real*
assumes $n \leq$ *cfrac-length c*
assumes $0 < b$ **and** $b < k\ n$ **and** *coprime a b*
defines $x \equiv$ *cfrac-lim c*
shows $|x - \textit{conv}\ c\ n| \leq |x - a\ /\ b|$
proof −
have $b < k\ n$ **by** *fact*
also have $k\ n \leq k$ (*Suc n*)
unfolding *k-def* **by** (*intro conv-denom-leI*) *auto*
finally have ∗: $b < k$ (*Suc n*) **by** *simp*
have $|x - \textit{conv}\ c\ n| = |k\ n * x - h\ n|\ /\ k\ n$
using *conv-denom-pos*[*of c n*] *assms*(*1*)
by (*auto simp*: *conv-num-denom field-simps k-def h-def*)
also have $\ldots \leq |b * x - a|\ /\ k\ n$ **unfolding** *x-def* **using** *assms* ∗
by (*intro divide-right-mono conv-best-approximation-ex-weak*) *auto*
also from *assms* **have** $\ldots \leq |b * x - a|\ /\ b$
by (*intro divide-left-mono*) *auto*
also have $\ldots = |x - a\ /\ b|$ **using** *assms* **by** (*simp add*: *field-simps*)
finally show *?thesis* **.**
qed

lemma *conv-denom-partition*:
assumes $y > 0$
shows $\exists!n.\ y \in \{k\ n..<k\ (\textit{Suc}\ n)\}$
proof (*rule ex-ex1I*)
from *conv-denom-at-top*[*of c*] *assms* **have** ∗: $\exists\, n.\ k\ n \geq y + 1$
by (*auto simp*: *k-def filterlim-at-top eventually-at-top-linorder*)

define *n* **where** $n = (LEAST\ n.\ k\ n \geq y + 1)$
from *LeastI-ex*[*OF* ∗] **have** *n*: $k\ n > y$ **by** (*simp add*: *Suc-le-eq n-def*)
from *n* **and** *assms* **have** $n > 0$ **by** (*intro Nat.gr0I*) (*auto simp*: *k-def*)

have $k\ (n - 1) \leq y$
proof (*rule ccontr*)
assume $\neg k\ (n - 1) \leq y$
hence $k\ (n - 1) \geq y + 1$ **by** *auto*
hence $n - 1 \geq n$ **unfolding** *n-def* **by** (*rule Least-le*)
with ‹$n > 0$› **show** *False* **by** *simp*
qed
with *n* **and** ‹$n > 0$› **have** $y \in \{k\ (n - 1)..<k\ (Suc\ (n - 1))\}$ **by** *auto*
thus $\exists n.\ y \in \{k\ n..<k\ (Suc\ n)\}$ **by** *blast*
next
fix *m n*
assume $y \in \{k\ m..<k\ (Suc\ m)\}$ $y \in \{k\ n..<k\ (Suc\ n)\}$
thus $m = n$
proof (*induction m n rule*: *linorder-wlog*)
case (*le m n*)
show $m = n$
proof (*rule ccontr*)
assume $m \neq n$
with *le* **have** $k\ (Suc\ m) \leq k\ n$
unfolding *k-def* **by** (*intro conv-denom-leI assms*) *auto*
with *le* **show** *False* **by** *auto*
qed
qed *auto*
qed

A fraction that approximates a real number *x* sufficiently well (in a certain sense) is a convergent of its continued fraction expansion.

lemma *frac-is-convergentI*:
fixes *a b* :: *int* **and** *x* :: *real*
defines $x \equiv cfrac\text{-}lim\ c$
assumes $b > 0$ **and** *coprime a b* **and** $|x - a\ /\ b| < 1\ /\ (2 * b^2)$
shows $\exists n.\ enat\ n \leq cfrac\text{-}length\ c \wedge (a,\ b) = (h\ n,\ k\ n)$
proof (*cases* $a = 0$)
case *True*
with *assms* **have** [*simp*]: $a = 0$ $b = 1$
by *auto*

show *?thesis*
proof (*cases x 0* :: *real rule*: *linorder-cases*)
case *greater*
hence $0 < x$ $x < 1/2$
using *assms* **by** *auto*
hence $x \notin \mathbb{Z}$
by (*auto simp*: *Ints-def*)
hence $cfrac\text{-}nth\ c\ 0 = \lfloor x \rfloor$

```
      using assms by (subst cfrac-nth-0-not-int) (auto simp: x-def)
    also from ‹x > 0› ‹x < 1/2› have ... = 0
      by linarith
    finally have (a, b) = (h 0, k 0)
      by (auto simp: h-def k-def)
    thus ?thesis by (intro exI[of - 0]) (auto simp flip: zero-enat-def)
  next
    case less
    hence x < 0 x > −1/2
      using assms by auto
    hence x ∉ ℤ
      by (auto simp: Ints-def)
    hence not-int: ¬cfrac-is-int c
      by (auto simp: cfrac-is-int-def x-def cfrac-lim-def)
    have cfrac-nth c 0 = ⌊x⌋
      using ‹x ∉ ℤ› assms by (subst cfrac-nth-0-not-int) (auto simp: x-def)
    also from ‹x < 0› ‹x > −1/2› have ... = −1
      by linarith
    finally have [simp]: cfrac-nth c 0 = −1 .
    have cfrac-nth c (Suc 0) = cfrac-nth (cfrac-tl c) 0
      by simp
    have cfrac-lim (cfrac-tl c) = 1 / (x + 1)
      using not-int by (subst cfrac-lim-tl) (auto simp: x-def)
    also from ‹x < 0› ‹x > −1/2› have ... ∈ {1<..<2}
      by (auto simp: divide-simps)
    finally have ∗: cfrac-lim (cfrac-tl c) ∈ {1<..<2} .
    have cfrac-nth (cfrac-tl c) 0 = ⌊cfrac-lim (cfrac-tl c)⌋
      using ∗ by (subst cfrac-nth-0-not-int) (auto simp: Ints-def)
    also have ... = 1
      using ∗ by (simp, linarith?)
    finally have (a, b) = (h 1, k 1)
      by (auto simp: h-def k-def)
    moreover have cfrac-length c ≥ 1
      using not-int
     by (cases cfrac-length c) (auto simp: cfrac-is-int-def one-enat-def zero-enat-def)
    ultimately show ?thesis by (intro exI[of - 1]) (auto simp: one-enat-def)
  next
    case equal
    show ?thesis
      using cfrac-nth-0-cases[of c]
    proof
      assume cfrac-nth c 0 = ⌊cfrac-lim c⌋
      with equal have (a, b) = (h 0, k 0)
        by (simp add: x-def h-def k-def)
      thus ?thesis by (intro exI[of - 0]) (auto simp flip: zero-enat-def)
    next
      assume ∗: cfrac-nth c 0 = ⌊cfrac-lim c⌋ − 1 ∧ cfrac-tl c = 1
      have [simp]: cfrac-nth c 0 = −1
        using ∗ equal by (auto simp: x-def)
```

from ∗ **have** ¬*cfrac-is-int c*
  by (*auto simp*: *cfrac-is-int-def cfrac-lim-def floor-minus*)
have *cfrac-nth c 1* = *cfrac-nth* (*cfrac-tl c*) *0*
  by *auto*
also have *cfrac-tl c* = *1*
  using ∗ **by** *auto*
finally have *cfrac-nth c 1* = *1*
  by *simp*
hence (*a*, *b*) = (*h 1*, *k 1*)
  by (*auto simp*: *h-def k-def*)
moreover from ‹¬*cfrac-is-int c*› **have** *cfrac-length c* ≥ *1*
by (*cases cfrac-length c*) (*auto simp*: *one-enat-def zero-enat-def cfrac-is-int-def*)
ultimately show *?thesis*
  by (*intro exI*[*of - 1*]) (*auto simp*: *one-enat-def*)
qed
qed
next
case *False*
hence *a-nz*: *a* ≠ *0* **by** *auto*

have *x* ≠ *0*
proof
  assume [*simp*]: *x* = *0*
  hence |*a*| / *b* < *1* / (*2* ∗ *b* ^ *2*)
    using *assms* **by** *simp*
  hence |*a*| < *1* / (*2* ∗ *b*)
    using *assms* **by** (*simp add*: *field-simps power2-eq-square*)
  also have ... ≤ *1* / *2*
    using *assms* **by** (*intro divide-left-mono*) *auto*
  finally have *a* = *0* **by** *auto*
  with ‹*a* ≠ *0*› **show** *False* **by** *simp*
qed

show *?thesis*
proof (*rule ccontr*)
  assume *no-convergent*: ∄*n*. *enat n* ≤ *cfrac-length c* ∧ (*a*, *b*) = (*h n*, *k n*)
  from *assms* **have** ∃!*r*. *b* ∈ {*k r*..<*k* (*Suc r*)}
    by (*intro conv-denom-partition*) *auto*
  then obtain *r* **where** *r*: *b* ∈ {*k r*..<*k* (*Suc r*)} **by** *auto*
  have *k r* > *0*
    using *conv-denom-pos*[*of c r*] *assms* **by** (*auto simp*: *k-def*)

  show *False*
  proof (*cases enat r* ≤ *cfrac-length c*)
    case *False*
    then obtain *l* **where** *l*: *cfrac-length c* = *enat l*
      by (*cases cfrac-length c*) *auto*
    have *k l* ≤ *k r*
      using *False l* **unfolding** *k-def* **by** (*intro conv-denom-leI*) *auto*

also have ... ≤ *b*
  using *r* **by** *simp*
finally have *b* ≥ *k l* **.**

have *x* = *conv c l*
  by (*auto simp*: *x-def cfrac-lim-def l*)
hence *x-eq*: *x* = *h l* / *k l*
  by (*auto simp*: *conv-num-denom h-def k-def*)
have *k l* > *0*
  by (*simp add*: *k-def*)

have *b* ∗ *k l* ∗ |*h l* / *k l* − *a* / *b*| < *k l* / (*2*∗*b*)
  using *assms x-eq* ‹*k l* > *0*› **by** (*auto simp*: *field-simps power2-eq-square*)
also have *b* ∗ *k l* ∗ |*h l* / *k l* − *a* / *b*| = |*b* ∗ *k l* ∗ (*h l* / *k l* − *a* / *b*)|
  using ‹*b* > *0*› ‹*k l* > *0*› **by** (*subst abs-mult*) *auto*
also have ... = *of-int* |*b* ∗ *h l* − *a* ∗ *k l*|
  using ‹*b* > *0*› ‹*k l* > *0*› **by** (*simp add*: *algebra-simps*)
also have *k l* / (*2* ∗ *b*) < *1*
  using ‹*b* ≥ *k l*› ‹*b* > *0*› **by** *auto*
finally have *a* ∗ *k l* = *b* ∗ *h l*
  by *linarith*
moreover have *coprime* (*h l*) (*k l*)
  unfolding *h-def k-def* **by** (*simp add*: *coprime-conv-num-denom*)
ultimately have (*a*, *b*) = (*h l*, *k l*)
  using ‹*coprime a b*› **using** *a-nz* ‹*b* > *0*› ‹*k l* > *0*›
  by (*subst* (*asm*) *coprime-crossproduct′*) (*auto simp*: *coprime-commute*)
with *no-convergent* **and** *l* **show** *False*
  by *auto*

next

case *True*
have *k r* ∗ |*x* − *h r* / *k r*| = |*k r* ∗ *x* − *h r*|
  using ‹*k r* > *0*› **by** (*simp add*: *field-simps*)
also have |*k r* ∗ *x* − *h r*| ≤ |*b* ∗ *x* − *a*|
using *assms r True* **unfolding** *x-def* **by** (*intro conv-best-approximation-ex-weak*)
auto
also have ... = *b* ∗ |*x* − *a* / *b*|
  using ‹*b* > *0*› **by** (*simp add*: *field-simps*)
also have ... < *b* ∗ (*1* / (*2* ∗ b^2))
  using ‹*b* > *0*› **by** (*intro mult-strict-left-mono assms*) *auto*
finally have *less*: |*x* − *conv c r*| < *1* / (*2* ∗ *b* ∗ *k r*)
  using ‹*k r* > *0*› **and** ‹*b* > *0*› **and** *assms*
  by (*simp add*: *field-simps power2-eq-square conv-num-denom h-def k-def*)

have |*x* − *a* / *b*| < *1* / (*2* ∗ b^2) **by** *fact*
also have ... = *1* / (*2* ∗ *b*) ∗ (*1* / *b*)
  by (*simp add*: *power2-eq-square*)
also have ... ≤ *1* / (*2* ∗ *b*) ∗ (|*a*| / *b*)

```
      using a-nz assms by (intro mult-left-mono divide-right-mono) auto
    also have ... < 1 / 1 * (|a| / b)
      using a-nz assms
        by (intro mult-strict-right-mono divide-left-mono divide-strict-left-mono)
auto
    also have ... = |a / b| using assms by simp
    finally have sgn x = sgn (a / b)
      by (auto simp: sgn-if split: if-splits)
    hence sgn x = sgn a using assms by (auto simp: sgn-of-int)
    hence a ≥ 0 ∧ x ≥ 0 ∨ a ≤ 0 ∧ x ≤ 0
      by (auto simp: sgn-if split: if-splits)
    moreover have h r ≥ 0 ∧ x ≥ 0 ∨ h r ≤ 0 ∧ x ≤ 0
      using conv-best-approximation-aux[of r] by (auto simp: h-def x-def)
    ultimately have signs: h r ≥ 0 ∧ a ≥ 0 ∨ h r ≤ 0 ∧ a ≤ 0
      using ‹x ≠ 0› by auto

    with no-convergent assms assms True have |h r| ≠ |a| ∨ b ≠ k r
      by (auto simp: h-def k-def)

    hence |h r| * |b| ≠ |a| * |k r| unfolding h-def k-def
      using assms coprime-conv-num-denom[of c r]
      by (subst coprime-crossproduct-int) auto
    hence |h r| * b ≠ |a| * k r using assms by (simp add: k-def)
    hence k r * a − h r * b ≠ 0
      using signs by (auto simp: algebra-simps)
    hence |k r * a − h r * b| ≥ 1 by presburger
    hence real-of-int 1 / (k r * b) ≤ |k r * a − h r * b| / (k r * b)
      using assms
      by (intro divide-right-mono, subst of-int-le-iff) (auto simp: k-def)
    also have ... = |(real-of-int (k r) * a − h r * b) / (k r * b)|
      using assms by (simp add: k-def)
    also have (real-of-int (k r) * a − h r * b) / (k r * b) = a / b − conv c r
    using assms ‹k r > 0› by (simp add: h-def k-def conv-num-denom field-simps)
    also have |a / b − conv c r| = |(x − conv c r) − (x − a / b)|
      by (simp add: algebra-simps)
    also have ... ≤ |x − conv c r| + |x − a / b|
      by (rule abs-triangle-ineq4)
    also have ... < 1 / (2 * b * k r) + 1 / (2 * b^2)
      by (intro add-strict-mono assms less)
    finally have k r > b
      using ‹b > 0› and ‹k r > 0› by (simp add: power2-eq-square field-simps)
    with r show False by auto
  qed
 qed
qed

end
```

1.5 Efficient code for convergents

```
function conv-gen :: (nat ⇒ int) ⇒ int × int × nat ⇒ nat ⇒ int where
  conv-gen c (a, b, n) N =
    (if n > N then b else conv-gen c (b, b * c n + a, Suc n) N)
  by auto
termination by (relation measure (λ(-, (-, -, n), N). Suc N − n)) auto

lemmas [simp del] = conv-gen.simps

lemma conv-gen-aux-simps [simp]:
  n > N ⟹ conv-gen c (a, b, n) N = b
  n ≤ N ⟹ conv-gen c (a, b, n) N = conv-gen c (b, b * c n + a, Suc n) N
  by (subst conv-gen.simps, simp)+

lemma conv-num-eq-conv-gen-aux:
  Suc n ≤ N ⟹ conv-num c n = b * cfrac-nth c n + a ⟹
    conv-num c (Suc n) = conv-num c n * cfrac-nth c (Suc n) + b ⟹
    conv-num c N = conv-gen (cfrac-nth c) (a, b, n) N
proof (induction cfrac-nth c (a, b, n) N arbitrary: c a b n rule: conv-gen.induct)
  case (1 a b n N c)
  show ?case
  proof (cases Suc (Suc n) ≤ N)
    case True
    thus ?thesis
      by (subst 1) (insert 1.prems, auto)
  next
    case False
    thus ?thesis using 1
      by (auto simp: not-le less-Suc-eq)
  qed
qed

lemma conv-denom-eq-conv-gen-aux:
  Suc n ≤ N ⟹ conv-denom c n = b * cfrac-nth c n + a ⟹
    conv-denom c (Suc n) = conv-denom c n * cfrac-nth c (Suc n) + b ⟹
    conv-denom c N = conv-gen (cfrac-nth c) (a, b, n) N
proof (induction cfrac-nth c (a, b, n) N arbitrary: c a b n rule: conv-gen.induct)
  case (1 a b n N c)
  show ?case
  proof (cases Suc (Suc n) ≤ N)
    case True
    thus ?thesis
      by (subst 1) (insert 1.prems, auto)
  next
    case False
    thus ?thesis using 1
      by (auto simp: not-le less-Suc-eq)
  qed
qed
```

lemma *conv-num-code* [*code*]: *conv-num c n* = *conv-gen* (*cfrac-nth c*) (*0*, *1*, *0*) *n*
using *conv-num-eq-conv-gen-aux*[*of 0 n c 1 0*] **by** (*cases n*) *simp-all*

lemma *conv-denom-code* [*code*]: *conv-denom c n* = *conv-gen* (*cfrac-nth c*) (*1*, *0*, *0*) *n*
using *conv-denom-eq-conv-gen-aux*[*of 0 n c 0 1*] **by** (*cases n*) *simp-all*

definition *conv-num-fun* **where** *conv-num-fun c* = *conv-gen c* (*0*, *1*, *0*)
definition *conv-denom-fun* **where** *conv-denom-fun c* = *conv-gen c* (*1*, *0*, *0*)

lemma
assumes *is-cfrac c*
shows *conv-num-fun-eq*: *conv-num-fun c n* = *conv-num* (*cfrac c*) *n*
and *conv-denom-fun-eq*: *conv-denom-fun c n* = *conv-denom* (*cfrac c*) *n*
proof −
from *assms* **have** *cfrac-nth* (*cfrac c*) = *c*
by (*intro ext*) *simp-all*
thus *conv-num-fun c n* = *conv-num* (*cfrac c*) *n* **and** *conv-denom-fun c n* = *conv-denom* (*cfrac c*) *n*
by (*simp-all add*: *conv-num-fun-def conv-num-code conv-denom-fun-def conv-denom-code*)
qed

1.6 Computing the continued fraction expansion of a rational number

function *cfrac-list-of-rat* :: *int* × *int* ⇒ *int list* **where**
cfrac-list-of-rat (*a*, *b*) =
(*if b* = *0 then* [*0*]
else a div b # (*if a mod b* = *0 then* [] *else cfrac-list-of-rat* (*b*, *a mod b*)))
by *auto*
termination
by (*relation measure* (λ(*a*,*b*). *nat* (*abs b*))) (*auto simp*: *abs-mod-less*)

lemmas [*simp del*] = *cfrac-list-of-rat.simps*

lemma *cfrac-list-of-rat-correct*:
(*let xs* = *cfrac-list-of-rat* (*a*, *b*); *c* = *cfrac-of-real* (*a* / *b*)
in length xs = *cfrac-length c* + *1* ∧ (∀ *i*<*length xs*. *xs* ! *i* = *cfrac-nth c i*))
proof (*induction* (*a*, *b*) *arbitrary*: *a b rule*: *cfrac-list-of-rat.induct*)
case (*1 a b*)
show *?case*
proof (*cases b* = *0*)
case *True*
thus *?thesis*
by (*subst cfrac-list-of-rat.simps*) (*auto simp*: *one-enat-def*)
next
case *False*
define *c* **where** *c* = *cfrac-of-real* (*a* / *b*)

define *c'* **where** *c'* = *cfrac-of-real* (*b* / (*a mod b*))
define *xs'* **where** *xs'* = (*if a mod b* = *0 then* [] *else cfrac-list-of-rat* (*b*, *a mod b*))
define *xs* **where** *xs* = *a div b* # *xs'*

have [*simp*]: *cfrac-nth c 0* = *a div b*
by (*auto simp*: *c-def floor-divide-of-int-eq*)

obtain *l* **where** *l*: *cfrac-length c* = *enat l*
by (*cases cfrac-length c*) (*auto simp*: *c-def*)

have *length xs* = *l* + *1* ∧ (∀ *i*<*length xs*. *xs* ! *i* = *cfrac-nth c i*)
proof (*cases b dvd a*)
case *True*
thus *?thesis* **using** *l*
by (*auto simp*: *Let-def xs-def xs'-def c-def of-int-divide-in-Ints one-enat-def enat-0-iff*)
next
case *False*
have *l* ≠ *0*
using *l False cfrac-of-real-length-eq-0-iff*[*of a* / *b*] ‹*b* ≠ *0*›
by (*auto simp*: *c-def zero-enat-def real-of-int-divide-in-Ints-iff intro*!: *Nat.gr0I*)
have *c'*: *c'* = *cfrac-tl c*
using *False* ‹*b* ≠ *0*› **unfolding** *c'-def c-def*
by (*subst cfrac-tl-of-real*) (*auto simp*: *real-of-int-divide-in-Ints-iff frac-fraction*)
from *1* **have** *enat* (*length xs'*) = *cfrac-length c'* + *1*
and *xs'*: ∀ *i*<*length xs'*. *xs'* ! *i* = *cfrac-nth c' i*
using ‹*b* ≠ *0*› ‹¬*b dvd a*› **by** (*auto simp*: *Let-def xs'-def c'-def*)

have *enat* (*length xs'*) = *cfrac-length c'* + *1*
by *fact*
also have ... = *enat l* − *1* + *1*
using *c' l* **by** *simp*
also have ... = *enat* (*l* − *1* + *1*)
by (*metis enat-diff-one one-enat-def plus-enat-simps*(*1*))
also have *l* − *1* + *1* = *l*
using ‹*l* ≠ *0*› **by** *simp*
finally have [*simp*]: *length xs'* = *l*
by *simp*

from *xs'* **show** *?thesis*
by (*auto simp*: *xs-def nth-Cons c' split*: *nat.splits*)
qed
thus *?thesis* **using** *l False*
by (*subst cfrac-list-of-rat.simps*) (*simp-all add*: *xs-def xs'-def c-def one-enat-def*)
qed
qed

lemma *conv-num-cong*:

assumes $(\bigwedge k.\ k \leq n \Longrightarrow$ *cfrac-nth c k* $=$ *cfrac-nth c′ k*$)$ $n = n'$
shows *conv-num c n* $=$ *conv-num c′ n*
proof –
have *conv-num c n* $=$ *conv-num c′ n*
using *assms*(*1*)
by (*induction n arbitrary*: *rule*: *conv-num.induct*) *simp-all*
thus *?thesis* **using** *assms*(*2*)
by *simp*
qed

lemma *conv-denom-cong*:
assumes $(\bigwedge k.\ k \leq n \Longrightarrow$ *cfrac-nth c k* $=$ *cfrac-nth c′ k*$)$ $n = n'$
shows *conv-denom c n* $=$ *conv-denom c′ n′*
proof –
have *conv-denom c n* $=$ *conv-denom c′ n*
using *assms*(*1*)
by (*induction n arbitrary*: *rule*: *conv-denom.induct*) *simp-all*
thus *?thesis* **using** *assms*(*2*)
by *simp*
qed

lemma *cfrac-lim-diff-le*:
assumes ∀ *k*≤*Suc n*. *cfrac-nth c1 k* = *cfrac-nth c2 k*
assumes *n* ≤ *cfrac-length c1 n* ≤ *cfrac-length c2*
shows |*cfrac-lim c1* − *cfrac-lim c2*| ≤ *2* / (*conv-denom c1 n* ∗ *conv-denom c1* (*Suc n*))
proof –
define *d* **where** *d* = (λ*k*. *conv-denom c1 k*)
have |*cfrac-lim c1* − *cfrac-lim c2*| ≤ |*cfrac-lim c1* − *conv c1 n*| + |*cfrac-lim c2* − *conv c1 n*|
by *linarith*
also have |*cfrac-lim c1* − *conv c1 n*| ≤ *1* / (*d n* ∗ *d* (*Suc n*))
unfolding *d-def* **using** *assms*
by (*intro cfrac-lim-minus-conv-upper-bound*) *auto*
also have *conv c1 n* = *conv c2 n*
using *assms* **by** (*intro conv-cong*) *auto*
also have |*cfrac-lim c2* − *conv c2 n*| ≤ *1* / (*conv-denom c2 n* ∗ *conv-denom c2* (*Suc n*))
using *assms* **unfolding** *d-def* **by** (*intro cfrac-lim-minus-conv-upper-bound*) *auto*
also have *conv-denom c2 n* = *d n*
unfolding *d-def* **using** *assms* **by** (*intro conv-denom-cong*) *auto*
also have *conv-denom c2* (*Suc n*) = *d* (*Suc n*)
unfolding *d-def* **using** *assms* **by** (*intro conv-denom-cong*) *auto*
also have *1* / (*d n* ∗ *d* (*Suc n*)) + *1* / (*d n* ∗ *d* (*Suc n*)) = *2* / (*d n* ∗ *d* (*Suc n*))
by *simp*
finally show *?thesis*
by (*simp add*: *d-def*)
qed

lemma *of-int-leI*: $n \leq m \Longrightarrow$ (*of-int n* :: *'a* :: *linordered-idom*) $\leq$ *of-int m*
  by *simp*

lemma *cfrac-lim-diff-le'*:
  assumes $\forall$ *k*$\leq$*Suc n*. *cfrac-nth c1 k* = *cfrac-nth c2 k*
  assumes *n* $\leq$ *cfrac-length c1 n* $\leq$ *cfrac-length c2*
  shows |*cfrac-lim c1* − *cfrac-lim c2*| $\leq$ *2* / (*fib* (*n*+*1*) ∗ *fib* (*n*+*2*))
proof −
  have |*cfrac-lim c1* − *cfrac-lim c2*| $\leq$ *2* / (*conv-denom c1 n* ∗ *conv-denom c1* (*Suc n*))
    by (*rule cfrac-lim-diff-le*) (*use assms* **in** *auto*)
  also have ... $\leq$ *2* / (*int* (*fib* (*Suc n*)) ∗ *int* (*fib* (*Suc* (*Suc n*))))
    unfolding *of-nat-mult of-int-mult*
    by (*intro divide-left-mono mult-mono mult-pos-pos of-int-leI conv-denom-lower-bound*)
      (*auto intro*!: *fib-neq-0-nat simp del*: *fib.simps*)
  also have ... = *2* / (*fib* (*n*+*1*) ∗ *fib* (*n*+*2*))
    by *simp*
  finally show *?thesis* **.**
qed

end

2 Quadratic Irrationals

theory *Quadratic-Irrationals*
imports
  Continued-Fractions
  HOL−Computational-Algebra.Computational-Algebra
  HOL−Library.Discrete
  Coinductive.Coinductive-Stream
begin

lemma *snth-cycle*:
  assumes *xs* $\neq$ []
  shows *snth* (*cycle xs*) *n* = *xs* ! (*n mod length xs*)
proof (*induction n rule*: *less-induct*)
  case (*less n*)
  have *snth* (*shift xs* (*cycle xs*)) *n* = *xs* ! (*n mod length xs*)
  proof (*cases n* < *length xs*)
    case *True*
    thus *?thesis*
      by (*subst shift-snth-less*) *auto*
  next
    case *False*
    have *0* < *length xs*
      using *assms* **by** *simp*
    also have ... $\leq$ *n*
      using *False* **by** *simp*

finally have $n > 0$ **.**

from *False* **have** *snth* (*shift xs* (*cycle xs*)) $n =$ *snth* (*cycle xs*) ($n -$ *length xs*)
by (*subst shift-snth-ge*) *auto*
also have ... $=$ *xs* ! (($n -$ *length xs*) *mod length xs*)
using *assms* ‹$n > 0$› **by** (*intro less*) *auto*
also have ($n -$ *length xs*) *mod length xs* $= n$ *mod length xs*
using *False* **by** (*simp add*: *mod-if*)
finally show *?thesis* **.**
qed
also have *shift xs* (*cycle xs*) $=$ *cycle xs*
by (*rule cycle-decomp* [*symmetric*]) *fact*
finally show *?case* **.**
qed

2.1 Basic results on rationality of square roots

lemma *inverse-in-Rats-iff* [*simp*]: *inverse* (x :: *real*) $\in \mathbb{Q} \longleftrightarrow x \in \mathbb{Q}$
by (*auto simp*: *inverse-eq-divide divide-in-Rats-iff1*)

lemma *nonneg-sqrt-nat-or-irrat*:
assumes $x \,\hat{}\, 2 =$ *real a* **and** $x \geq 0$
shows $x \in \mathbb{N} \lor x \notin \mathbb{Q}$
proof *safe*
assume $x \notin \mathbb{N}$ **and** $x \in \mathbb{Q}$
from *Rats-abs-nat-div-natE*[*OF this*(*2*)]
obtain *p q* :: *nat* **where** *q-nz* [*simp*]: $q \neq 0$ **and** *abs* $x = p \,/\, q$ **and** *coprime*: *coprime p q* **.**
with ‹$x \geq 0$› **have** x: $x = p \,/\, q$
by *simp*
with *assms* **have** *real* ($q \,\hat{}\, 2$) $*$ *real a* $=$ *real* ($p \,\hat{}\, 2$)
by (*simp add*: *field-simps*)
also have *real* ($q \,\hat{}\, 2$) $*$ *real a* $=$ *real* ($q \,\hat{}\, 2 * a$)
by *simp*
finally have $p \,\hat{}\, 2 = q \,\hat{}\, 2 * a$
by (*subst* (*asm*) *of-nat-eq-iff*) *auto*
hence $q \,\hat{}\, 2$ *dvd* $p \,\hat{}\, 2$
by *simp*
hence *q dvd p*
by *simp*
with *coprime* **have** $q = 1$
by *auto*
with *x* **and** ‹$x \notin \mathbb{N}$› **show** *False*
by *simp*
qed

A square root of a natural number is either an integer or irrational.

corollary *sqrt-nat-or-irrat*:
assumes $x \,\hat{}\, 2 =$ *real a*

shows $x \in \mathbb{Z} \lor x \notin \mathbb{Q}$
proof (*cases* $x \geq 0$)
case *True*
with *nonneg-sqrt-nat-or-irrat*[*OF assms this*]
show *?thesis* **by** (*auto simp*: *Nats-altdef2*)
next
case *False*
from *assms* **have** $(-x)$ ^ *2* = *real a*
by *simp*
moreover from *False* **have** $-x \geq 0$
by *simp*
ultimately have $-x \in \mathbb{N} \lor -x \notin \mathbb{Q}$
by (*rule nonneg-sqrt-nat-or-irrat*)
thus *?thesis*
by (*auto simp*: *Nats-altdef2 minus-in-Ints-iff*)
qed

corollary *sqrt-nat-or-irrat'*:
sqrt (*real a*) $\in \mathbb{N} \lor$ *sqrt* (*real a*) $\notin \mathbb{Q}$
using *nonneg-sqrt-nat-or-irrat*[*of sqrt a a*] **by** *auto*

The square root of a natural number n is again a natural number iff *n is a perfect square.*

corollary *sqrt-nat-iff-is-square*:
sqrt (*real n*) $\in \mathbb{N} \longleftrightarrow$ *is-square n*
proof
assume *sqrt* (*real n*) $\in \mathbb{N}$
then obtain *k* **where** *sqrt* (*real n*) = *real k* **by** (*auto elim*!: *Nats-cases*)
hence *sqrt* (*real n*) ^ *2* = *real* (*k* ^ *2*) **by** (*simp only*: *of-nat-power*)
also have *sqrt* (*real n*) ^ *2* = *real n* **by** *simp*
finally have *n* = *k* ^ *2* **by** (*simp only*: *of-nat-eq-iff*)
thus *is-square n* **by** *blast*
qed (*auto elim*!: *is-nth-powerE*)

corollary *irrat-sqrt-nonsquare*: ¬*is-square n* $\Longrightarrow$ *sqrt* (*real n*) $\notin \mathbb{Q}$
using *sqrt-nat-or-irrat'*[*of n*] **by** (*auto simp*: *sqrt-nat-iff-is-square*)

lemma *sqrt-of-nat-in-Rats-iff*: *sqrt* (*real n*) $\in \mathbb{Q} \longleftrightarrow$ *is-square n*
using *irrat-sqrt-nonsquare*[*of n*] *sqrt-nat-iff-is-square*[*of n*] *Nats-subset-Rats* **by** *blast*

lemma *Discrete-sqrt-altdef*: *Discrete.sqrt n* = *nat* $\lfloor$*sqrt n*$\rfloor$
proof –
have *real* (*Discrete.sqrt n* ^ *2*) $\leq$ *sqrt n* ^ *2*
by *simp*
hence *Discrete.sqrt n* $\leq$ *sqrt n*
unfolding *of-nat-power* **by** (*rule power2-le-imp-le*) *auto*
moreover have *real* (*Suc* (*Discrete.sqrt n*) ^ *2*) > *real n*
unfolding *of-nat-less-iff* **by** (*rule Suc-sqrt-power2-gt*)

hence *real* (*Discrete.sqrt n* + *1*) ^ *2* > *sqrt n* ^ *2*
unfolding *of-nat-power* **by** *simp*
hence *real* (*Discrete.sqrt n* + *1*) > *sqrt n*
by (*rule power2-less-imp-less*) *auto*
hence *Discrete.sqrt n* + *1* > *sqrt n* **by** *simp*
ultimately show *?thesis* **by** *linarith*
qed

2.2 Definition of quadratic irrationals

Irrational real numbers x that satisfy a quadratic equation $ax^2 + bx + c = 0$ with a, b, c not all equal to 0 are called *quadratic irrationals*. These are of the form $p + q\sqrt{d}$ for rational numbers p, q and a positive integer d.

inductive *quadratic-irrational* :: *real* ⇒ *bool* **where**
$x \notin \mathbb{Q} \Longrightarrow$ *real-of-int a* ∗ *x* ^ *2* + *real-of-int b* ∗ *x* + *real-of-int c* = *0* ⟹
a ≠ *0* ∨ *b* ≠ *0* ∨ *c* ≠ *0* ⟹ *quadratic-irrational x*

lemma *quadratic-irrational-sqrt* [*intro*]:
assumes ¬*is-square n*
shows *quadratic-irrational* (*sqrt* (*real n*))
using *irrat-sqrt-nonsquare*[*OF assms*]
by (*intro quadratic-irrational.intros*[*of sqrt n 1 0* −*int n*]) *auto*

lemma *quadratic-irrational-uminus* [*intro*]:
assumes *quadratic-irrational x*
shows *quadratic-irrational* (−*x*)
using *assms*
proof *induction*
case (*1 x a b c*)
thus *?case* **by** (*intro quadratic-irrational.intros*[*of* −*x a* −*b c*]) *auto*
qed

lemma *quadratic-irrational-uminus-iff* [*simp*]:
quadratic-irrational (−*x*) ⟷ *quadratic-irrational x*
using *quadratic-irrational-uminus*[*of x*] *quadratic-irrational-uminus*[*of* −*x*] **by** *auto*

lemma *quadratic-irrational-plus-int* [*intro*]:
assumes *quadratic-irrational x*
shows *quadratic-irrational* (*x* + *of-int n*)
using *assms*
proof *induction*
case (*1 x a b c*)
define *x′* **where** *x′* = *x* + *of-int n*
define *a′ b′ c′* **where**
a′ = *a* **and** *b′* = *b* − *2* ∗ *of-int n* ∗ *a* **and**
c′ = *a* ∗ *of-int n* ^ *2* − *b* ∗ *of-int n* + *c*
from *1* **have** *0* = *a* ∗ (*x′* − *of-int n*) ^ *2* + *b* ∗ (*x′* − *of-int n*) + *c*
by (*simp add*: *x′-def*)

also have ... = $a' * x' \,\hat{}\, 2 + b' * x' + c'$
  by (*simp add*: *algebra-simps a'-def b'-def c'-def power2-eq-square*)
finally have ... = *0* **..**
moreover have $x' \notin \mathbb{Q}$
  using *1* **by** (*auto simp*: *x'-def add-in-Rats-iff2*)
moreover have $a' \neq 0 \lor b' \neq 0 \lor c' \neq 0$
  using *1* **by** (*auto simp*: *a'-def b'-def c'-def*)
ultimately show *?case*
  by (*intro quadratic-irrational.intros*[*of x + of-int n a' b' c'*]) (*auto simp*: *x'-def*)
qed

lemma *quadratic-irrational-plus-int-iff* [*simp*]:
quadratic-irrational (*x* + *of-int n*) ⟷ *quadratic-irrational x*
using *quadratic-irrational-plus-int*[*of x n*]
quadratic-irrational-plus-int[*of x* + *of-int n* −*n*] **by** *auto*

lemma *quadratic-irrational-minus-int-iff* [*simp*]:
quadratic-irrational (*x* − *of-int n*) ⟷ *quadratic-irrational x*
using *quadratic-irrational-plus-int-iff*[*of x* −*n*]
by (*simp del*: *quadratic-irrational-plus-int-iff*)

lemma *quadratic-irrational-plus-nat-iff* [*simp*]:
quadratic-irrational (*x* + *of-nat n*) ⟷ *quadratic-irrational x*
using *quadratic-irrational-plus-int-iff*[*of x int n*]
by (*simp del*: *quadratic-irrational-plus-int-iff*)

lemma *quadratic-irrational-minus-nat-iff* [*simp*]:
quadratic-irrational (*x* − *of-nat n*) ⟷ *quadratic-irrational x*
using *quadratic-irrational-plus-int-iff*[*of x* −*int n*]
by (*simp del*: *quadratic-irrational-plus-int-iff*)

lemma *quadratic-irrational-plus-1-iff* [*simp*]:
quadratic-irrational (*x* + *1*) ⟷ *quadratic-irrational x*
using *quadratic-irrational-plus-int-iff*[*of x 1*]
by (*simp del*: *quadratic-irrational-plus-int-iff*)

lemma *quadratic-irrational-minus-1-iff* [*simp*]:
quadratic-irrational (*x* − *1*) ⟷ *quadratic-irrational x*
using *quadratic-irrational-plus-int-iff*[*of x* −*1*]
by (*simp del*: *quadratic-irrational-plus-int-iff*)

lemma *quadratic-irrational-plus-numeral-iff* [*simp*]:
quadratic-irrational (*x* + *numeral n*) ⟷ *quadratic-irrational x*
using *quadratic-irrational-plus-int-iff*[*of x numeral n*]
by (*simp del*: *quadratic-irrational-plus-int-iff*)

lemma *quadratic-irrational-minus-numeral-iff* [*simp*]:
quadratic-irrational (*x* − *numeral n*) ⟷ *quadratic-irrational x*
using *quadratic-irrational-plus-int-iff*[*of x* −*numeral n*]

```
  by (simp del: quadratic-irrational-plus-int-iff)

lemma quadratic-irrational-inverse:
  assumes quadratic-irrational x
  shows   quadratic-irrational (inverse x)
  using assms
proof induction
  case (1 x a b c)
  from 1 have x ≠ 0 by auto
  have 0 = (real-of-int a ∗ x² + real-of-int b ∗ x + real-of-int c) / x ^ 2
    by (subst 1) simp
   also have ... = real-of-int c ∗ (inverse x) ^ 2 + real-of-int b ∗ inverse x +
real-of-int a
    using ‹x ≠ 0› by (simp add: field-simps power2-eq-square)
  finally have ... = 0 ..
  thus ?case using 1
    by (intro quadratic-irrational.intros[of inverse x c b a]) auto
qed

lemma quadratic-irrational-inverse-iff [simp]:
  quadratic-irrational (inverse x) ⟷ quadratic-irrational x
  using quadratic-irrational-inverse[of x] quadratic-irrational-inverse[of inverse x]
  by (cases x = 0) auto

lemma quadratic-irrational-cfrac-remainder-iff:
  quadratic-irrational (cfrac-remainder c n) ⟷ quadratic-irrational (cfrac-lim c)
proof (cases cfrac-length c = ∞)
  case False
  thus ?thesis
    by (auto simp: quadratic-irrational.simps)
next
  case [simp]: True
  show ?thesis
  proof (induction n)
    case (Suc n)
    from Suc.prems have cfrac-remainder c (Suc n) =
                        inverse (cfrac-remainder c n − of-int (cfrac-nth c n))
      by (subst cfrac-remainder-Suc) (auto simp: field-simps)
    also have quadratic-irrational ... ⟷ quadratic-irrational (cfrac-remainder c
n)
      by simp
    also have ... ⟷ quadratic-irrational (cfrac-lim c)
      by (rule Suc.IH)
    finally show ?case .
  qed auto
qed
```

2.3 Real solutions of quadratic equations

For the next result, we need some basic properties of real solutions to quadratic equations.

```
lemma quadratic-equation-reals:
  fixes a b c :: real
  defines f ≡ (λx. a ∗ x ^ 2 + b ∗ x + c)
  defines discr ≡ (b^2 − 4 ∗ a ∗ c)
  shows   {x. f x = 0} =
           (if a = 0 then
              (if b = 0 then if c = 0 then UNIV else {} else {−c/b})
            else if discr ≥ 0 then {(−b + sqrt discr) / (2 ∗ a), (−b − sqrt discr) /
(2 ∗ a)}
                                  else {}) (is ?th1)
proof (cases a = 0)
  case [simp]: True
  show ?th1
  proof (cases b = 0)
    case [simp]: True
    hence {x. f x = 0} = (if c = 0 then UNIV else {})
      by (auto simp: f-def)
    thus ?th1 by simp
  next
    case False
    hence {x. f x = 0} = {−c / b} by (auto simp: f-def field-simps)
    thus ?th1 using False by simp
  qed
next
  case [simp]: False
  show ?th1
  proof (cases discr ≥ 0)
    case True
    {
      fix x :: real
      have f x = a ∗ (x − (−b + sqrt discr) / (2 ∗ a)) ∗ (x − (−b − sqrt discr) /
(2 ∗ a))
        using True by (simp add: f-def field-simps discr-def power2-eq-square)
      also have ... = 0 ⟷ x ∈ {(−b + sqrt discr) / (2 ∗ a), (−b − sqrt discr)
/ (2 ∗ a)}
        by simp
      finally have f x = 0 ⟷ ... .
    }
    hence {x. f x = 0} = {(−b + sqrt discr) / (2 ∗ a), (−b − sqrt discr) / (2 ∗
a)}
      by blast
    thus ?th1 using True by simp
  next
    case False
    {
```

fix *x* :: *real*
assume *x*: *f x* = *0*
have $0 \le (x + b / (2 * a))$ ^ *2* **by** *simp*
also have *f x* = *a* ∗ ((*x* + *b* / (*2* ∗ *a*)) ^ *2* − *b* ^ *2* / (*4* ∗ *a* ^ *2*) + *c* / *a*)
by (*simp add*: *field-simps power2-eq-square f-def*)
with *x* **have** (*x* + *b* / (*2* ∗ *a*)) ^ *2* − *b* ^ *2* / (*4* ∗ *a* ^ *2*) + *c* / *a* = *0*
by *simp*
hence (*x* + *b* / (*2* ∗ *a*)) ^ *2* = *b* ^ *2* / (*4* ∗ *a* ^ *2*) − *c* / *a*
by (*simp add*: *algebra-simps*)
finally have $0 \le (b^2 / (4 * a^2) - c / a) * (4 * a^2)$
by (*intro mult-nonneg-nonneg*) *auto*
also have ... = $b^2 - 4 * a * c$ **by** (*simp add*: *field-simps power2-eq-square*)
also have ... < *0* **using** *False* **by** (*simp add*: *discr-def*)
finally have *False* **by** *simp*
}
hence {*x*. *f x* = *0*} = {} **by** *auto*
thus *?th1* **using** *False* **by** *simp*
qed
qed

lemma *finite-quadratic-equation-solutions-reals*:
fixes *a b c* :: *real*
defines *discr* ≡ (*b*^*2* − *4* ∗ *a* ∗ *c*)
shows *finite* {*x*. *a* ∗ *x* ^ *2* + *b* ∗ *x* + *c* = *0*} ⟷ *a* ≠ *0* ∨ *b* ≠ *0* ∨ *c* ≠ *0*
by (*subst quadratic-equation-reals*)
(*auto simp*: *discr-def card-eq-0-iff infinite-UNIV-char-0 split*: *if-split*)

lemma *card-quadratic-equation-solutions-reals*:
fixes *a b c* :: *real*
defines *discr* ≡ (*b*^*2* − *4* ∗ *a* ∗ *c*)
shows *card* {*x*. *a* ∗ *x* ^ *2* + *b* ∗ *x* + *c* = *0*} =
(*if a* = *0 then*
(*if b* = *0 then 0 else 1*)
else if discr ≥ *0 then if discr* = *0 then 1 else 2 else 0*) (**is** *?th1*)
by (*subst quadratic-equation-reals*)
(*auto simp*: *discr-def card-eq-0-iff infinite-UNIV-char-0 split*: *if-split*)

lemma *card-quadratic-equation-solutions-reals-le-2*:
card {*x* :: *real*. *a* ∗ *x* ^ *2* + *b* ∗ *x* + *c* = *0*} ≤ *2*
by (*subst card-quadratic-equation-solutions-reals*) *auto*

lemma *quadratic-equation-solution-rat-iff*:
fixes *a b c* :: *int* **and** *x y* :: *real*
defines *f* ≡ (λ*x*::*real*. *a* ∗ *x* ^ *2* + *b* ∗ *x* + *c*)
defines *discr* ≡ *nat* (*b* ^ *2* − *4* ∗ *a* ∗ *c*)
assumes *a* ≠ *0 f x* = *0*
shows *x* ∈ ℚ ⟷ *is-square discr*
proof −
define *discr'* **where** *discr'* ≡ *real-of-int* (*b* ^ *2* − *4* ∗ *a* ∗ *c*)

from *assms* **have** $x \in \{x.\ f\ x = 0\}$ **by** *simp*
with ‹$a \neq 0$› **have** $discr' \geq 0$ **unfolding** *discr'-def f-def of-nat-diff*
by (*subst* (*asm*) *quadratic-equation-reals*) (*auto simp*: *discr-def split*: *if-splits*)
hence $*$: *sqrt* (*discr'*) = *sqrt* (*real discr*) **unfolding** *of-int-0-le-iff discr-def discr'-def*
by (*simp add*: *algebra-simps nat-diff-distrib*)
from ‹$x \in \{x.\ f\ x = 0\}$› **have** $x = (-b + \mathit{sqrt\ discr}) / (2 * a) \lor x = (-b - \mathit{sqrt\ discr}) / (2 * a)$
using ‹$a \neq 0$› $*$ **unfolding** *discr'-def f-def*
by (*subst* (*asm*) *quadratic-equation-reals*) (*auto split*: *if-splits*)
thus *?thesis* **using** ‹$a \neq 0$›
by (*auto simp*: *sqrt-of-nat-in-Rats-iff divide-in-Rats-iff2 diff-in-Rats-iff2 diff-in-Rats-iff1*)
qed

2.4 Periodic continued fractions and quadratic irrationals

We now show the main result: A positive irrational number has a periodic continued fraction expansion iff it is a quadratic irrational.

In principle, this statement naturally also holds for negative numbers, but the current formalisation of continued fractions only supports non-negative numbers. It also holds for rational numbers in some sense, since their continued fraction expansion is finite to begin with.

theorem *periodic-cfrac-imp-quadratic-irrational*:
assumes [*simp*]: *cfrac-length* $c = \infty$
and *period*: $l > 0$ $\bigwedge k.\ k \geq N \Longrightarrow$ *cfrac-nth* c $(k + l)$ = *cfrac-nth* c k
shows *quadratic-irrational* (*cfrac-lim* c)
proof −
define h' **and** k' **where** h' = *conv-num-int* (*cfrac-drop* N c)
and k' = *conv-denom-int* (*cfrac-drop* N c)
define x' **where** x' = *cfrac-remainder* c N

have *c-pos*: *cfrac-nth* c $n > 0$ **if** $n \geq N$ **for** n
proof −
from *assms*(*1*,*2*) **have** *cfrac-nth* c $(n + l) > 0$ **by** *auto*
with *assms*(*3*)[*OF that*] **show** *?thesis* **by** *simp*
qed
have *k'-pos*: k' $n > 0$ **if** $n \neq -1$ $n \geq -2$ **for** n
using *that* **by** (*auto simp*: *k'-def conv-denom-int-def intro*!: *conv-denom-pos*)
have *k'-nonneg*: k' $n \geq 0$ **if** $n \geq -2$ **for** n
using *that* **by** (*auto simp*: *k'-def conv-denom-int-def intro*!: *conv-denom-pos*)
have *cfrac-nth* c $(n + (N + l))$ = *cfrac-nth* c $(n + N)$ **for** n
using *period*(*2*)[*of* $n + N$] **by** (*simp add*: *add-ac*)
have *cfrac-drop* $(N + l)$ c = *cfrac-drop* N c
by (*rule cfrac-eqI*) (*use period*(*2*)[*of* $n + N$ **for** n] **in** ‹*auto simp*: *algebra-simps*›)
hence *x'-altdef*: x' = *cfrac-remainder* c $(N + l)$
by (*simp add*: *x'-def cfrac-remainder-def*)
have *x'-pos*: $x' > 0$ **unfolding** *x'-def*
using *c-pos* **by** (*intro cfrac-remainder-pos*) *auto*

define *A* **where** $A = (k' \ (int\ l - 1))$
define *B* **where** $B = k' \ (int\ l - 2) - h' \ (int\ l - 1)$
define *C* **where** $C = -(h' \ (int\ l - 2))$

have *pos*: $(k' \ (int\ l - 1) * x' + k' \ (int\ l - 2)) > 0$
using *x′-pos* ‹$l > 0$›
by (*intro add-pos-nonneg mult-pos-pos*) (*auto intro*!: *k′-pos k′-nonneg*)
have *cfrac-remainder c N* = *conv′* (*cfrac-drop N c*) *l* (*cfrac-remainder c* (*l* + *N*))
unfolding *cfrac-remainder-def cfrac-drop-add*
by (*subst* (*2*) *cfrac-remainder-def* [*symmetric*]) (*auto simp*: *conv′-cfrac-remainder*)
hence *x′* = *conv′* (*cfrac-drop N c*) *l x′*
by (*subst* (*asm*) *add.commute*) (*simp only*: *x′-def* [*symmetric*] *x′-altdef* [*symmetric*])
also have ... $= (h' \ (int\ l - 1) * x' + h' \ (int\ l - 2)) \ / \ (k' \ (int\ l - 1) * x' + k' \ (int\ l - 2))$
using *conv′-num-denom-int*[*OF x′-pos*, *of - l*] **unfolding** *h′-def k′-def*
by (*simp add*: *mult-ac*)
finally have $x' * (k' \ (int\ l - 1) * x' + k' \ (int\ l - 2)) = (h' \ (int\ l - 1) * x' + h' \ (int\ l - 2))$
using *pos* **by** (*simp add*: *divide-simps*)
hence *quadratic*: $A * x' \hat{}\ 2 + B * x' + C = 0$
by (*simp add*: *algebra-simps power2-eq-square A-def B-def C-def*)
moreover have $x' \notin \mathbb{Q}$ **unfolding** *x′-def*
by *auto*
moreover have $A > 0$ **using** ‹$l > 0$› **by** (*auto simp*: *A-def intro*!: *k′-pos*)
ultimately have *quadratic-irrational x′* **using** ‹$x' \notin \mathbb{Q}$›
by (*intro quadratic-irrational.intros*[*of x′ A B C*]) *simp-all*
thus *?thesis*
using *assms* **by** (*simp add*: *x′-def quadratic-irrational-cfrac-remainder-iff*)
qed

lift-definition *pperiodic-cfrac* :: *nat list* ⇒ *cfrac* **is**
λxs. if xs = [] *then* (*0*, *LNil*) *else*
(*int* (*hd xs*), *llist-of-stream* (*cycle* (*map* (*λn. n*− *1*) (*tl xs* @ [*hd xs*])))) **.**

definition *periodic-cfrac* :: *int list* ⇒ *int list* ⇒ *cfrac* **where**
periodic-cfrac xs ys = *cfrac-of-stream* (*Stream.shift xs* (*Stream.cycle ys*))

lemma *periodic-cfrac-Nil* [*simp*]: *pperiodic-cfrac* [] = *0*
unfolding *zero-cfrac-def* **by** *transfer auto*

lemma *cfrac-length-pperiodic-cfrac* [*simp*]:
xs ≠ [] ⟹ *cfrac-length* (*pperiodic-cfrac xs*) = ∞
by *transfer auto*

lemma *cfrac-nth-pperiodic-cfrac*:
assumes *xs* ≠ [] **and** *0* ∉ *set xs*

```
  shows   cfrac-nth (pperiodic-cfrac xs) n = xs ! (n mod length xs)
  using assms
proof (transfer, goal-cases)
  case (1 xs n)
  show ?case
  proof (cases n)
    case (Suc n′)
    have int (cycle (tl (map (λn. n − 1) xs) @ [hd (map (λn. n − 1) xs)]) !! n′) +
1 =
          int (stl (cycle (map (λn. n − 1) xs)) !! n′) + 1
      by (subst cycle.sel(2) [symmetric]) (rule refl)
    also have ... = int (cycle (map (λn. n − 1) xs) !! n) + 1
      by (simp add: Suc del: cycle.sel)
    also have ... = int (xs ! (n mod length xs) − 1) + 1
      by (simp add: snth-cycle ‹xs ≠ []›)
    also have xs ! (n mod length xs) ∈ set xs
      using ‹xs ≠ []› by (auto simp: set-conv-nth)
    with 1 have xs ! (n mod length xs) > 0
      by (intro Nat.gr0I) auto
    hence int (xs ! (n mod length xs) − 1) + 1 = int (xs ! (n mod length xs))
      by simp
    finally show ?thesis
      using Suc 1 by (simp add: hd-conv-nth map-tl)
  qed (use 1 in ‹auto simp: hd-conv-nth›)
qed

definition pperiodic-cfrac-info :: nat list ⇒ int × int × intwhere
  pperiodic-cfrac-info xs =
    (let l = length xs;
         h = conv-num-fun (λn. xs ! n);
         k = conv-denom-fun (λn. xs ! n);
         A = k (l − 1);
         B = h (l − 1) − (if l = 1 then 0 else k (l − 2));
         C = (if l = 1 then −1 else −h (l − 2))
     in (B^2−4∗A∗C, B, 2 ∗ A))

lemma conv-gen-cong:
  assumes ∀ k∈{n..N}. f k = f′ k
  shows   conv-gen f (a,b,n) N = conv-gen f′ (a,b,n) N
  using assms
proof (induction N − n arbitrary: a b n N)
  case (Suc d n N a b)
  have conv-gen f (b, b ∗ f n + a, Suc n) N = conv-gen f′ (b, b ∗ f n + a, Suc n)
N
    using Suc(2,3) by (intro Suc) auto
  moreover have f n = f′ n
    using bspec[OF Suc.prems, of n] Suc(2) by auto
  ultimately show ?case
    by (subst (1 2) conv-gen.simps) auto
```

```
qed (auto simp: conv-gen.simps)

lemma
  assumes ∀k≤n. c k = cfrac-nth c' k
  shows   conv-num-fun-eq': conv-num-fun c n = conv-num c' n
    and   conv-denom-fun-eq': conv-denom-fun c n = conv-denom c' n
proof −
  have conv-num c' n = conv-gen (cfrac-nth c') (0, 1, 0) n
    unfolding conv-num-code ..
  also have ... = conv-gen c (0, 1, 0) n
    unfolding conv-num-fun-def using assms by (intro conv-gen-cong) auto
  finally show conv-num-fun c n = conv-num c' n
    by (simp add: conv-num-fun-def)
next
  have conv-denom c' n = conv-gen (cfrac-nth c') (1, 0, 0) n
    unfolding conv-denom-code ..
  also have ... = conv-gen c (1, 0, 0) n
    unfolding conv-denom-fun-def using assms by (intro conv-gen-cong) auto
  finally show conv-denom-fun c n = conv-denom c' n
    by (simp add: conv-denom-fun-def)
qed

lemma gcd-minus-commute-left: gcd (a − b :: 'a :: ring-gcd) c = gcd (b − a) c
  by (metis gcd.commute gcd-neg2 minus-diff-eq)

lemma gcd-minus-commute-right: gcd c (a − b :: 'a :: ring-gcd) = gcd c (b − a)
  by (metis gcd-neg2 minus-diff-eq)

lemma periodic-cfrac-info-aux:
  fixes D E F :: int
  assumes pperiodic-cfrac-info xs = (D, E, F)
  assumes xs ≠ [] 0 ∉ set xs
  shows   cfrac-lim (pperiodic-cfrac xs) = (sqrt D + E) / F
    and   D > 0 and F > 0
proof −
  define c where c = pperiodic-cfrac xs
  have [simp]: cfrac-length c = ∞
    using assms by (simp add: c-def)
  define h and k where h = conv-num-int c and k = conv-denom-int c
  define x where x = cfrac-lim c
  define l where l = length xs

  define A where A = (k (int l − 1))
  define B where B = k (int l − 2) − h (int l − 1)
  define C where C = −(h (int l − 2))
  define discr where discr = B ^ 2 − 4 * A * C

  have l > 0
    using assms by (simp add: l-def)
```

have *c-pos*: *cfrac-nth c n* > *0* **for** *n*
using *assms* **by** (*auto simp*: *c-def cfrac-nth-pperiodic-cfrac set-conv-nth*)
have *x-pos*: *x* > *0*
unfolding *x-def* **by** (*intro cfrac-lim-pos c-pos*)
have *h-pos*: *h n* > *0* **if** *n* > −*2* **for** *n*
using *that* **unfolding** *h-def* **by** (*auto simp*: *conv-num-int-def intro*: *conv-num-pos′ c-pos*)
have *k-pos*: *k n* > *0* **if** *n* > −*1* **for** *n*
using *that* **unfolding** *k-def* **by** (*auto simp*: *conv-denom-int-def*)
have *k-nonneg*: *k n* ≥ *0* **for** *n*
unfolding *k-def* **by** (*auto simp*: *conv-denom-int-def*)

have *pos*: (*k* (*int l* − *1*) ∗ *x* + *k* (*int l* − *2*)) > *0*
using *x-pos* ‹*l* > *0*›
by (*intro add-pos-nonneg mult-pos-pos*) (*auto intro*!: *k-pos k-nonneg*)
have *cfrac-drop l c* = *c*
using *assms* **by** (*intro cfrac-eqI*) (*auto simp*: *c-def cfrac-nth-pperiodic-cfrac l-def*)

have *x* = *conv′ c l* (*cfrac-remainder c l*)
unfolding *x-def* **by** (*rule conv′-cfrac-remainder*[*symmetric*]) *auto*
also have ... = *conv′ c l x*
unfolding *cfrac-remainder-def* ‹*cfrac-drop l c* = *c*› *x-def* **..**
finally have *x* = *conv′ c l x* **.**
also have ... = (*h* (*int l* − *1*) ∗ *x* + *h* (*int l* − *2*)) / (*k* (*int l* − *1*) ∗ *x* + *k* (*int l* − *2*))
using *conv′-num-denom-int*[*OF x-pos*, *of - l*] **unfolding** *h-def k-def*
by (*simp add*: *mult-ac*)
finally have *x* ∗ (*k* (*int l* − *1*) ∗ *x* + *k* (*int l* − *2*)) = (*h* (*int l* − *1*) ∗ *x* + *h* (*int l* − *2*))
using *pos* **by** (*simp add*: *divide-simps*)
hence *quadratic*: *A* ∗ *x* ^ *2* + *B* ∗ *x* + *C* = *0*
by (*simp add*: *algebra-simps power2-eq-square A-def B-def C-def*)

have *A* > *0* **using** ‹*l* > *0*› **by** (*auto simp*: *A-def intro*!: *k-pos*)
have *discr-altdef*: *discr* = (*k* (*int l*−*2*) − *h* (*int l*−*1*)) ^ *2* + *4* ∗ *k* (*int l*−*1*) ∗ *h* (*int l*−*2*)
by (*simp add*: *discr-def A-def B-def C-def*)

have *0* < *0* + *4* ∗ *A* ∗ *1*
using ‹*A* > *0*› **by** *simp*
also have *0* + *4* ∗ *A* ∗ *1* ≤ *discr*
unfolding *discr-altdef A-def* **using** *h-pos*[*of int l* − *2*] ‹*l* > *0*›
by (*intro add-mono mult-mono order.refl k-nonneg mult-nonneg-nonneg*) *auto*
finally have *discr* > *0* **.**

have *x* ∈ {*x*. *A* ∗ *x* ^ *2* + *B* ∗ *x* + *C* = *0*}
using *quadratic* **by** *simp*
hence *x-cases*: *x* = (−*B* − *sqrt discr*) / (*2* ∗ *A*) ∨ *x* = (−*B* + *sqrt discr*) / (*2*

∗ A)
  unfolding *quadratic-equation-reals of-int-diff* **using** ‹A > 0›
  by (*auto split*: *if-splits simp*: *discr-def*)

have $B \hat{} 2 < discr$
  unfolding *discr-def* **by** (*auto intro*!: *mult-pos-pos k-pos h-pos* ‹l > 0› *simp*: *A-def C-def*)
hence $|B| < sqrt\ discr$
  using ‹discr > 0› **by** (*simp add*: *real-less-rsqrt*)

have $x = (if\ x \geq 0\ then\ (sqrt\ discr - B) / (2 * A)\ else\ -(sqrt\ discr + B) / (2 * A))$
  using *x-cases*
proof
  assume *x*: $x = (-B - sqrt\ discr) / (2 * A)$
  have $(-B - sqrt\ discr) / (2 * A) < 0$
    using ‹|B| < sqrt discr› ‹A > 0› **by** (*intro divide-neg-pos*) *auto*
  also note *x*[*symmetric*]
  finally show *?thesis* **using** *x* **by** *simp*
next
  assume *x*: $x = (-B + sqrt\ discr) / (2 * A)$
  have $(-B + sqrt\ discr) / (2 * A) > 0$
    using ‹|B| < sqrt discr› ‹A > 0› **by** (*intro divide-pos-pos*) *auto*
  also note *x*[*symmetric*]
  finally show *?thesis* **using** *x* **by** *simp*
qed
also have $x \geq 0 \longleftrightarrow floor\ x \geq 0$
  by *auto*
also have $floor\ x = floor\ (cfrac\text{-}lim\ c)$
  by (*simp add*: *x-def*)
also have $\ldots = cfrac\text{-}nth\ c\ 0$
  by (*subst cfrac-nth-0-conv-floor*) *auto*
also have $\ldots = int\ (hd\ xs)$
  using *assms* **unfolding** *c-def* **by** (*subst cfrac-nth-pperiodic-cfrac*) (*auto simp*: *hd-conv-nth*)
finally have *x-eq*: $x = (sqrt\ discr - B) / (2 * A)$
  by *simp*

define h' **where** $h' = conv\text{-}num\text{-}fun\ (\lambda n.\ int\ (xs\ !\ n))$
define k' **where** $k' = conv\text{-}denom\text{-}fun\ (\lambda n.\ int\ (xs\ !\ n))$
have *num-eq*: $h'\ i = h\ i$
  if $i < l$ **for** *i* **using** *that assms* **unfolding** *h'-def h-def*
  by (*subst conv-num-fun-eq'*[**where** $c' = c$]) (*auto simp*: *c-def l-def cfrac-nth-pperiodic-cfrac*)
have *denom-eq*: $k'\ i = k\ i$
  if $i < l$ **for** *i* **using** *that assms* **unfolding** *k'-def k-def*
  by (*subst conv-denom-fun-eq'*[**where** $c' = c$]) (*auto simp*: *c-def l-def cfrac-nth-pperiodic-cfrac*)

have *1*: $h\ (int\ l - 1) = h'\ (l - 1)$

by (*subst num-eq*) (*use* ‹*l* > *0*› **in** ‹*auto simp*: *of-nat-diff*›)
have *2*: *k* (*int l* − *1*) = *k'* (*l* − *1*)
by (*subst denom-eq*) (*use* ‹*l* > *0*› **in** ‹*auto simp*: *of-nat-diff*›)
have *3*: *h* (*int l* − *2*) = (*if l* = *1 then 1 else h'* (*l* − *2*))
using ‹*l* > *0*› *num-eq*[*of l* − *2*] **by** (*auto simp*: *h-def nat-diff-distrib*)
have *4*: *k* (*int l* − *2*) = (*if l* = *1 then 0 else k'* (*l* − *2*))
using ‹*l* > *0*› *denom-eq*[*of l* − *2*] **by** (*auto simp*: *k-def nat-diff-distrib*)

have *pperiodic-cfrac-info xs* =
(*let A* = *k* (*int l* − *1*);
B = *h* (*int l* − *1*) − (*if l* = *1 then 0 else k* (*int l* − *2*));
C = (*if l* = *1 then* −*1 else* − *h* (*int l* − *2*))
in (B^2 − *4* ∗ *A* ∗ *C*, *B*, *2* ∗ *A*))
unfolding *pperiodic-cfrac-info-def Let-def* **using** *1 2 3 4* ‹*l* > *0*›
by (*auto simp*: *num-eq denom-eq h'-def k'-def l-def of-nat-diff*)
also have . . . = (B^2 − *4* ∗ *A* ∗ *C*, −*B*, *2* ∗ *A*)
by (*simp add*: *Let-def A-def B-def C-def h-def k-def algebra-simps power2-commute*)
finally have *per-eq*: *pperiodic-cfrac-info xs* = (*discr*, −*B*, *2* ∗ *A*)
by (*simp add*: *discr-def*)

show *x* = (*sqrt* (*real-of-int D*) + *real-of-int E*) / *real-of-int F*
using *per-eq assms* **by** (*simp add*: *x-eq*)
show *D* > *0 F* > *0*
using *assms per-eq* ‹*discr* > *0*› ‹*A* > *0*› **by** *auto*
qed

We can now compute surd representations for (purely) periodic continued fractions, e.g. $[1, 1, 1, \ldots] = \frac{\sqrt{5}+1}{2}$:

value *pperiodic-cfrac-info* [*1*]

We can now compute surd representations for periodic continued fractions, e.g. $[\overline{1, 1, 1, 1, 6}] = \frac{\sqrt{13}+3}{4}$:

value *pperiodic-cfrac-info* [*1*,*1*,*1*,*1*,*6*]

With a little bit of work, one could also easily derive from this a version for non-purely periodic continued fraction.

Next, we show that any quadratic irrational has a periodic continued fraction expansion.

theorem *quadratic-irrational-imp-periodic-cfrac*:
assumes *quadratic-irrational* (*cfrac-lim e*)
obtains *N l* **where** *l* > *0* **and** ⋀*n m*. *n* ≥ *N* ⟹ *cfrac-nth e* (*n* + *m* ∗ *l*) = *cfrac-nth e n*
and *cfrac-remainder e* (*N* + *l*) = *cfrac-remainder e N*
and *cfrac-length e* = ∞
proof −
have [*simp*]: *cfrac-length e* = ∞
using *assms* **by** (*auto simp*: *quadratic-irrational.simps*)

```
note [intro] = assms(1)
define x where x = cfrac-lim e
from assms obtain a b c :: int where
  nontrivial: a ≠ 0 ∨ b ≠ 0 ∨ c ≠ 0 and
      root: a * x^2 + b * x + c = 0 (is ?f x = 0)
  by (auto simp: quadratic-irrational.simps x-def)

define f where f = ?f
define h and k where h = conv-num e and k = conv-denom e
define X where X = cfrac-remainder e
have [simp]: k i > 0 k i ≠ 0 for i
  using conv-denom-pos[of e i] by (auto simp: k-def)
have k-leI: k i ≤ k j if i ≤ j for i j
  by (auto simp: k-def intro!: conv-denom-leI that)
have k-nonneg: k n ≥ 0 for n
  by (auto simp: k-def)
have k-ge-1: k n ≥ 1 for n
  using k-leI[of 0 n] by (simp add: k-def)

define R where R = conv e
define A where A = (λn. a * h (n − 1) ^ 2 + b * h (n − 1) * k (n − 1) + c
* k (n − 1) ^ 2)
define B where B = (λn. 2 * a * h (n − 1) * h (n − 2) + b * (h (n − 1) * k
(n − 2) + h (n − 2) * k (n − 1)) + 2 * c * k (n − 1) * k (n − 2))
define C where C = (λn. a * h (n − 2) ^ 2 + b * h (n − 2) * k (n − 2) + c
* k (n − 2) ^ 2)

define A' where A' = nat ⌊2 * |a| * |x| + |a| + |b|⌋
define B' where B' = nat ⌊(3 / 2) * (2 * |a| * |x| + |b|) + 9 / 4 * |a|⌋

have [simp]: X n ∉ ℚ for n unfolding X-def
  by simp
from this[of 0] have [simp]: x ∉ ℚ
  unfolding X-def by (simp add: x-def)

have a ≠ 0
proof
  assume a = 0
  with root and nontrivial have x = 0 ∨ x = −c / b
    by (auto simp: divide-simps add-eq-0-iff)
  hence x ∈ ℚ by (auto simp del: ‹x ∉ ℚ›)
  thus False by simp
qed

have bounds: (A n, B n, C n) ∈ {−A'..A'} × {−B'..B'} × {−A'..A'}
 and X-root: A n * X n ^ 2 + B n * X n + C n = 0 if n: n ≥ 2 for n
proof −
  define n' where n' = n − 2
  have n': n = Suc (Suc n') using ‹n ≥ 2› unfolding n'-def by simp
```

have ∗: *of-int* (*k* (*n* − *Suc 0*)) ∗ *X n* + *of-int* (*k* (*n* − *2*)) ≠ *0*
proof
assume *of-int* (*k* (*n* − *Suc 0*)) ∗ *X n* + *of-int* (*k* (*n* − *2*)) = *0*
hence *X n* = −*k* (*n* − *2*) / *k* (*n* − *1*) **by** (*auto simp*: *divide-simps mult-ac*)
also have ... ∈ ℚ **by** *auto*
finally show *False* **by** *simp*
qed

let *?denom* = (*k* (*n* − *1*) ∗ *X n* + *k* (*n* − *2*))
have *0* = *0* ∗ *?denom* ^ *2* **by** *simp*
also have *0* ∗ *?denom* ^ *2* = (*a* ∗ *x* ^ *2* + *b* ∗ *x* + *c*) ∗ *?denom* ^ *2* **using** *root* **by** *simp*
also have ... = *a* ∗ (*x* ∗ *?denom*) ^ *2* + *b* ∗ *?denom* ∗ (*x* ∗ *?denom*) + *c* ∗ *?denom* ∗ *?denom*
by (*simp add*: *algebra-simps power2-eq-square*)
also have *x* ∗ *?denom* = *h* (*n* − *1*) ∗ *X n* + *h* (*n* − *2*)
using *cfrac-lim-eq-num-denom-remainder-aux*[*of n* − *2 e*] ‹*n* ≥ *2*›
by (*simp add*: *numeral-2-eq-2 Suc-diff-Suc x-def k-def h-def X-def*)
also have *a* ∗ ... ^ *2* + *b* ∗ *?denom* ∗ ... + *c* ∗ *?denom* ∗ *?denom* = *A n* ∗ *X n* ^ *2* + *B n* ∗ *X n* + *C n*
by (*simp add*: *A-def B-def C-def power2-eq-square algebra-simps*)
finally show *A n* ∗ *X n* ^ *2* + *B n* ∗ *X n* + *C n* = *0* **..**

have *f-abs-bound*: |*f* (*R n*)| ≤ (*2* ∗ |*a*| ∗ |*x*| + |*b*|) ∗ (*1* / (*k n* ∗ *k* (*Suc n*))) + |*a*| ∗ (*1* / (*k n* ∗ *k* (*Suc n*))) ^ *2* **for** *n*
proof −
have |*f* (*R n*)| = |*?f* (*R n*) − *?f x*| **by** (*simp add*: *root f-def*)
also have *?f* (*R n*) − *?f x* = (*R n* − *x*) ∗ (*2* ∗ *a* ∗ *x* + *b*) + (*R n* − *x*) ^ *2* ∗ *a*
by (*simp add*: *power2-eq-square algebra-simps*)
also have |...| ≤ |(*R n* − *x*) ∗ (*2* ∗ *a* ∗ *x* + *b*)| + |(*R n* − *x*) ^ *2* ∗ *a*|
by (*rule abs-triangle-ineq*)
also have ... = |*2* ∗ *a* ∗ *x* + *b*| ∗ |*R n* − *x*| + |*a*| ∗ |*R n* − *x*| ^ *2*
by (*simp add*: *abs-mult*)
also have ... ≤ |*2* ∗ *a* ∗ *x* + *b*| ∗ (*1* / (*k n* ∗ *k* (*Suc n*))) + |*a*| ∗ (*1* / (*k n* ∗ *k* (*Suc n*))) ^ *2*
unfolding *x-def R-def* **using** *cfrac-lim-minus-conv-bounds*[*of n e*]
by (*intro add-mono mult-left-mono power-mono*) (*auto simp*: *k-def*)
also have |*2* ∗ *a* ∗ *x* + *b*| ≤ *2* ∗ |*a*| ∗ |*x*| + |*b*|
by (*rule order.trans*[*OF abs-triangle-ineq*]) (*auto simp*: *abs-mult*)
hence |*2* ∗ *a* ∗ *x* + *b*| ∗ (*1* / (*k n* ∗ *k* (*Suc n*))) + |*a*| ∗ (*1* / (*k n* ∗ *k* (*Suc n*))) ^ *2* ≤
... ∗ (*1* / (*k n* ∗ *k* (*Suc n*))) + |*a*| ∗ (*1* / (*k n* ∗ *k* (*Suc n*))) ^ *2*
by (*intro add-mono mult-right-mono*) (*auto intro*!: *mult-nonneg-nonneg k-nonneg*)
finally show |*f* (*R n*)| ≤ ...
by (*simp add*: *mult-right-mono add-mono divide-left-mono*)
qed

have *h-eq-conv-k*: *h i* = *R i* ∗ *k i* **for** *i*
using *conv-denom-pos*[*of e i*] **unfolding** *R-def*
by (*subst conv-num-denom*) (*auto simp*: *h-def k-def*)

have *A n* = *k* (*n* − *1*) ^ *2* ∗ *f* (*R* (*n* − *1*)) **for** *n*
by (*simp add*: *algebra-simps A-def n′ k-def power2-eq-square h-eq-conv-k f-def*)
have *A-bound*: |*A i*| ≤ *A′* **if** *i* > *0* **for** *i*
proof −
have *k i* > *0*
by *simp*
hence *k i* ≥ *1*
by *linarith*
have *A i* = *k* (*i* − *1*) ^ *2* ∗ *f* (*R* (*i* − *1*))
by (*simp add*: *algebra-simps A-def k-def power2-eq-square h-eq-conv-k f-def*)
also have |. . .| = *k* (*i* − *1*) ^ *2* ∗ |*f* (*R* (*i* − *1*))|
by (*simp add*: *abs-mult f-def*)
also have . . . ≤ *k* (*i* − *1*) ^ *2* ∗ ((*2* ∗ |*a*| ∗ |*x*| + |*b*|) ∗ (*1* / (*k* (*i* − *1*) ∗ *k* (*Suc* (*i* − *1*)))) +
|*a*| ∗ (*1* / (*k* (*i* − *1*) ∗ *k* (*Suc* (*i* − *1*)))) ^ *2*)
by (*intro mult-left-mono f-abs-bound*) *auto*
also have . . . = *k* (*i* − *1*) / *k i* ∗ (*2* ∗ |*a*| ∗ |*x*| + |*b*|) + |*a*| / *k i* ^ *2* **using** ‹*i* > *0*›
by (*simp add*: *power2-eq-square field-simps*)
also have . . . ≤ *1* ∗ (*2* ∗ |*a*| ∗ |*x*| + |*b*|) + |*a*| / *1* **using** ‹*i* > *0*› ‹*k i* ≥ *1*›
by (*intro add-mono divide-left-mono mult-right-mono*)
(*auto intro*!: *k-leI one-le-power simp*: *of-nat-ge-1-iff*)
also have . . . = *2* ∗ |*a*| ∗ |*x*| + |*a*| + |*b*| **by** *simp*
finally show *?thesis* **unfolding** *A′-def* **by** *linarith*
qed

have *C n* = *A* (*n* − *1*) **by** (*simp add*: *A-def C-def n′*)
hence *C-bound*: |*C n*| ≤ *A′* **using** *A-bound*[*of n* − *1*] *n* **by** *simp*

have *B n* = *k* (*n* − *1*) ∗ *k* (*n* − *2*) ∗
(*f* (*R* (*n* − *1*)) + *f* (*R* (*n* − *2*)) − *a* ∗ (*R* (*n* − *1*) − *R* (*n* − *2*)) ^ *2*)
by (*simp add*: *B-def h-eq-conv-k algebra-simps power2-eq-square f-def*)
also have |. . .| = *k* (*n* − *1*) ∗ *k* (*n* − *2*) ∗
|*f* (*R* (*n* − *1*)) + *f* (*R* (*n* − *2*)) − *a* ∗ (*R* (*n* − *1*) − *R* (*n* − *2*)) ^ *2*|
by (*simp add*: *abs-mult k-nonneg*)
also have . . . ≤ *k* (*n* − *1*) ∗ *k* (*n* − *2*) ∗
(((*2* ∗ |*a*| ∗ |*x*| + |*b*|) ∗ (*1* / (*k* (*n* − *1*) ∗ *k* (*Suc* (*n* − *1*)))) +
|*a*| ∗ (*1* / (*k* (*n* − *1*) ∗ *k* (*Suc* (*n* − *1*)))) ^ *2*) +
((*2* ∗ |*a*| ∗ |*x*| + |*b*|) ∗ (*1* / (*k* (*n* − *2*) ∗ *k* (*Suc* (*n* − *2*)))) +
|*a*| ∗ (*1* / (*k* (*n* − *2*) ∗ *k* (*Suc* (*n* − *2*)))) ^ *2*) +
|*a*| ∗ |*R* (*Suc* (*n* − *2*)) − *R* (*n* − *2*)| ^ *2*) (**is** - ≤ - ∗ (*?S1* + *?S2* + *?S3*))
by (*intro mult-left-mono order.trans*[*OF abs-triangle-ineq4*] *order.trans*[*OF abs-triangle-ineq*]

```
          add-mono f-abs-bound order.refl)
        (insert n, auto simp: abs-mult Suc-diff-Suc numeral-2-eq-2 k-nonneg)
    also have |R (Suc (n − 2)) − R (n − 2)| = 1 / (k (n − 2) ∗ k (Suc (n − 2)))
      unfolding R-def k-def by (rule abs-diff-successive-convs)
    also have of-int (k (n − 1) ∗ k (n − 2)) ∗ (?S1 + ?S2 + |a| ∗ . . . ^ 2) =
                (k (n − 2) / k n + 1) ∗ (2 ∗ |a| ∗ |x| + |b|) +
                |a| ∗ (k (n − 2) / (k (n − 1) ∗ k n ^ 2) + 2 / (k (n − 1) ∗ k (n −
2)))
     (is - = ?S) using n by (simp add: field-simps power2-eq-square numeral-2-eq-2
Suc-diff-Suc)
    also {
      have A: 2 ∗ real-of-int (k (n − 2)) ≤ of-int (k n)
        using conv-denom-plus2-ratio-ge[of e n − 2] n
        by (simp add: numeral-2-eq-2 Suc-diff-Suc k-def)
      have fib (Suc 2) ≤ k 2 unfolding k-def by (intro conv-denom-lower-bound)
      also have . . . ≤ k n by (intro k-leI n)
      finally have k n ≥ 2 by (simp add: numeral-3-eq-3)
      hence B: of-int (k (n − 2)) ∗ 2 ^ 2 ≤ (of-int (k (n − 1)) ∗ (of-int (k n))² ::
real)
        by (intro mult-mono power-mono) (auto intro: k-leI k-nonneg)
      have C: 1 ∗ 1 ≤ real-of-int (k (n − 1)) ∗ of-int (k (n − 2)) using k-ge-1
        by (intro mult-mono) (auto simp: Suc-le-eq of-nat-ge-1-iff k-nonneg)
      note A B C
    }
    hence ?S ≤ (1 / 2 + 1) ∗ (2 ∗ |a| ∗ |x| + |b|) + |a| ∗ (1 / 4 + 2)
      by (intro add-mono mult-right-mono mult-left-mono) (auto simp: field-simps)
    also have . . . = (3 / 2) ∗ (2 ∗ |a| ∗ |x| + |b|) + 9 / 4 ∗ |a| by simp
    finally have B-bound: |B n| ≤ B' unfolding B'-def by linarith
    from A-bound[of n] B-bound C-bound n
    show (A n, B n, C n) ∈ {−A'..A'} × {−B'..B'} × {−A'..A'} by auto
  qed

  have A-nz: A n ≠ 0 if n ≥ 1 for n
    using that
  proof (induction n rule: dec-induct)
    case base
    show ?case
    proof
      assume A 1 = 0
      hence real-of-int (A 1) = 0 by simp
      also have real-of-int (A 1) =
                  real-of-int a ∗ of-int (cfrac-nth e 0) ^ 2 +
                  real-of-int b ∗ cfrac-nth e 0 + real-of-int c
        by (simp add: A-def h-def k-def)
      finally have root': . . . = 0 .

      have cfrac-nth e 0 ∈ ℚ by auto
     also from root' and ‹a ≠ 0› have ?this ⟷ is-square (nat (b² − 4 ∗ a ∗ c))
        by (intro quadratic-equation-solution-rat-iff) auto
```

```
      also from root and ‹a ≠ 0› have ... ⟷ x ∈ ℚ
        by (intro quadratic-equation-solution-rat-iff [symmetric]) auto
      finally show False using ‹x ∉ ℚ› by contradiction
    qed
  next
    case (step m)
    hence nz: C (Suc m) ≠ 0 by (simp add: C-def A-def)
    show A (Suc m) ≠ 0
    proof
      assume [simp]: A (Suc m) = 0
      have X (Suc m) > 0 unfolding X-def
        by (intro cfrac-remainder-pos) auto
     with X-root[of Suc m] step.hyps nz have X (Suc m) = −C (Suc m) / B (Suc
m)
        by (auto simp: divide-simps mult-ac)
      also have ... ∈ ℚ by auto
      finally show False by simp
    qed
  qed

  have finite ({−A′..A′} × {−B′..B′} × {−A′..A′}) by auto
  from this and bounds have finite ((λn. (A n, B n, C n)) ‘ {2..})
    by (blast intro: finite-subset)
  moreover have infinite ({2..} :: nat set) by (simp add: infinite-Ici)
  ultimately have ∃k1∈{2..}. infinite {n ∈ {2..}. (A n, B n, C n) = (A k1, B
k1, C k1)}
    by (intro pigeonhole-infinite)
  then obtain k0 where k0: k0 ≥ 2 infinite {n ∈ {2..}. (A n, B n, C n) = (A
k0, B k0, C k0)}
    by auto
  from infinite-countable-subset[OF this(2)] obtain g :: nat ⇒ -
    where g: inj g range g ⊆ {n∈{2..}. (A n, B n, C n) = (A k0, B k0, C k0)} by
blast
  hence g-ge-2: g k ≥ 2 for k by auto
  from g have [simp]: A (g k) = A k0 B (g k) = B k0 C (g k) = C k0 for k
    by auto

  from g(1) have [simp]: g k1 = g k2 ⟷ k1 = k2 for k1 k2 by (auto simp:
inj-def)
  define z where z = (A k0, B k0, C k0)
  let ?h = λk. (A (g k), B (g k), C (g k))
  from g have g′: distinct [g 1, g 2, g 3] ?h 0 = z ?h 1 = z ?h 2 = z
    by (auto simp: z-def)
  have fin: finite {x :: real. A k0 ∗ x ^ 2 + B k0 ∗ x + C k0 = 0} using A-nz[of
k0] k0(1)
    by (subst finite-quadratic-equation-solutions-reals) auto
  from X-root[of g 0] X-root[of g 1] X-root[of g 2] g-ge-2 g
    have (X ∘ g) ‘ {0, 1, 2} ⊆ {x. A k0 ∗ x ^ 2 + B k0 ∗ x + C k0 = 0}
    by auto
```

```
  hence card ((X ∘ g) ` {0, 1, 2}) ≤ card ...
    by (intro card-mono fin) auto
  also have ... ≤ 2
    by (rule card-quadratic-equation-solutions-reals-le-2)
  also have ... < card {0, 1, 2 :: nat} by simp
  finally have ¬inj-on (X ∘ g) {0, 1, 2}
    by (rule pigeonhole)
  then obtain m1 m2 where
    m12: m1 ∈ {0, 1, 2} m2 ∈ {0, 1, 2} X (g m1) = X (g m2) m1 ≠ m2
    unfolding inj-on-def o-def by blast
  define n and l where n = min (g m1) (g m2) and l = nat |int (g m1) − g m2|
  with m12 g' have l: l > 0 X (n + l) = X n
    by (auto simp: min-def nat-diff-distrib split: if-splits)

  from l have cfrac-lim (cfrac-drop (n + l) e) = cfrac-lim (cfrac-drop n e)
    by (simp add: X-def cfrac-remainder-def)
  hence cfrac-drop (n + l) e = cfrac-drop n e
    by (simp add: cfrac-lim-eq-iff)
  hence cfrac-nth (cfrac-drop (n + l) e) = cfrac-nth (cfrac-drop n e)
    by (simp only:)
  hence period: cfrac-nth e (n + l + k) = cfrac-nth e (n + k) for k
    by (simp add: fun-eq-iff add-ac)
  have period: cfrac-nth e (k + l) = cfrac-nth e k if k ≥ n for k
    using period[of k − n] that by (simp add: add-ac)
  have period: cfrac-nth e (k + m ∗ l) = cfrac-nth e k if k ≥ n for k m
    using that
  proof (induction m)
    case (Suc m)
    have cfrac-nth e (k + Suc m ∗ l) = cfrac-nth e (k + m ∗ l + l)
      by (simp add: algebra-simps)
    also have ... = cfrac-nth e (k + m ∗ l)
      using Suc.prems by (intro period) auto
    also have ... = cfrac-nth e k
      using Suc.prems by (intro Suc.IH) auto
    finally show ?case .
  qed simp-all

  from this and l and that[of l n] show ?thesis by (simp add: X-def)
qed

theorem periodic-cfrac-iff-quadratic-irrational:
  assumes x ∉ ℚ x ≥ 0
  shows   quadratic-irrational x ⟷
          (∃ N l. l > 0 ∧ (∀ n≥N. cfrac-nth (cfrac-of-real x) (n + l) =
                                  cfrac-nth (cfrac-of-real x) n))
proof safe
  assume ∗: quadratic-irrational x
  with assms have ∗∗: quadratic-irrational (cfrac-lim (cfrac-of-real x)) by auto
  obtain N l where Nl: l > 0
```

$\bigwedge n\ m.\ N \leq n \Longrightarrow$ *cfrac-nth* (*cfrac-of-real x*) $(n + m * l) =$ *cfrac-nth* (*cfrac-of-real x*) *n*
cfrac-remainder (*cfrac-of-real x*) $(N + l) =$ *cfrac-remainder* (*cfrac-of-real x*) *N*
cfrac-length (*cfrac-of-real x*) $= \infty$
using *quadratic-irrational-imp-periodic-cfrac* [*OF* ∗∗] **by** *metis*
show ∃ *N l*. $l > 0$ ∧ (∀ *n*≥*N*. *cfrac-nth* (*cfrac-of-real x*) $(n + l) =$ *cfrac-nth* (*cfrac-of-real x*) *n*)
by (*rule exI*[*of - N*], *rule exI*[*of - l*]) (*insert Nl*(*1*) *Nl*(*2*)[*of - 1*], *auto*)
next
fix *N l* **assume** $l > 0$ ∀ *n*≥*N*. *cfrac-nth* (*cfrac-of-real x*) $(n + l) =$ *cfrac-nth* (*cfrac-of-real x*) *n*
hence *quadratic-irrational* (*cfrac-lim* (*cfrac-of-real x*)) **using** *assms*
by (*intro periodic-cfrac-imp-quadratic-irrational*[*of - l N*]) *auto*
with *assms* **show** *quadratic-irrational x*
by *simp*
qed

The following result can e.g. be used to show that a number is *not* a quadratic irrational.

lemma *quadratic-irrational-cfrac-nth-range-finite*:
assumes *quadratic-irrational* (*cfrac-lim e*)
shows *finite* (*range* (*cfrac-nth e*))
proof −
from *quadratic-irrational-imp-periodic-cfrac*[*OF assms*] **obtain** *l N*
where *period*: $l > 0$ $\bigwedge m\ n.\ n \geq N \Longrightarrow$ *cfrac-nth e* $(n + m * l) =$ *cfrac-nth e n*
by *metis*
have *cfrac-nth e k* ∈ *cfrac-nth e* ‘ {..<*N*+*l*} **for** *k*
proof (*cases* $k < N + l$)
case *False*
define *n m* **where** $n = N + (k - N)$ *mod l* **and** $m = (k - N)$ *div l*
have *cfrac-nth e n* ∈ *cfrac-nth e* ‘ {..<*N*+*l*}
using ‹$l > 0$› **by** (*intro imageI*) (*auto simp*: *n-def*)
also have *cfrac-nth e n* = *cfrac-nth e* $(n + m * l)$
by (*subst period*) (*auto simp*: *n-def*)
also have $n + m * l = k$
using *False* **by** (*simp add*: *n-def m-def*)
finally show *?thesis* **.**
qed *auto*
hence *range* (*cfrac-nth e*) ⊆ *cfrac-nth e* ‘ {..<*N*+*l*}
by *blast*
thus *?thesis* **by** (*rule finite-subset*) *auto*
qed

end

3 The continued fraction expansion of e

theory *E-CFrac*
imports

HOL−Analysis.Analysis
Continued-Fractions
Quadratic-Irrationals
begin

lemma *fact-real-at-top*: *filterlim* (*fact* :: *nat* ⇒ *real*) *at-top at-top*
proof (*rule filterlim-at-top-mono*)
  have *real n* ≤ *real* (*fact n*) **for** *n*
    unfolding *of-nat-le-iff* **by** (*rule fact-ge-self*)
  thus *eventually* (λ*n*. *real n* ≤ *fact n*) *at-top* **by** *simp*
qed (*fact filterlim-real-sequentially*)

lemma *filterlim-div-nat-at-top*:
  assumes *filterlim f at-top F m* > *0*
  shows *filterlim* (λ*x*. *f x div m* :: *nat*) *at-top F*
  unfolding *filterlim-at-top*
proof
  fix *C* :: *nat*
  from *assms*(*1*) **have** *eventually* (λ*x*. *f x* ≥ *C* ∗ *m*) *F*
    by (*auto simp*: *filterlim-at-top*)
  thus *eventually* (λ*x*. *f x div m* ≥ *C*) *F*
  proof *eventually-elim*
    case (*elim x*)
    hence (*C* ∗ *m*) *div m* ≤ *f x div m*
      by (*intro div-le-mono*)
    thus *?case* **using** ‹*m* > *0*› **by** *simp*
  qed
qed

The continued fraction expansion of e has the form $[2; 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, \ldots]$:

definition *e-cfrac* **where**
  e-cfrac = *cfrac* (λ*n*. *if n* = *0 then 2 else if n mod 3* = *2 then 2* ∗ (*Suc n div 3*) *else 1*)

lemma *cfrac-nth-e*:
  cfrac-nth e-cfrac n = (*if n* = *0 then 2 else if n mod 3* = *2 then 2* ∗ (*Suc n div 3*) *else 1*)
  unfolding *e-cfrac-def* **by** (*subst cfrac-nth-cfrac*) (*auto simp*: *is-cfrac-def*)

lemma *cfrac-length-e* [*simp*]: *cfrac-length e-cfrac* = ∞
  by (*simp add*: *e-cfrac-def*)

The formalised proof follows the one from Proof Wiki [2].

context
  fixes *A B C* :: *nat* ⇒ *real* **and** *p q* :: *nat* ⇒ *int* **and** *a* :: *nat* ⇒ *int*
  defines *A* ≡ (λ*n*. *integral* {*0..1*} (λ*x*. *exp x* ∗ *x* ^ *n* ∗ (*x* − *1*) ^ *n* / *fact n*))
    and *B* ≡ (λ*n*. *integral* {*0..1*} (λ*x*. *exp x* ∗ *x* ^ *Suc n* ∗ (*x* − *1*) ^ *n* / *fact n*))
    and *C* ≡ (λ*n*. *integral* {*0..1*} (λ*x*. *exp x* ∗ *x* ^ *n* ∗ (*x* − *1*) ^ *Suc n* / *fact n*))
    and *p* ≡ (λ*n*. *if n* ≤ *1 then 1 else conv-num e-cfrac* (*n* − *2*))

and $q \equiv (\lambda n.\ \mathit{if}\ n = \mathit{0}\ \mathit{then}\ \mathit{1}\ \mathit{else}\ \mathit{if}\ n = \mathit{1}\ \mathit{then}\ \mathit{0}\ \mathit{else}\ \mathit{conv\text{-}denom}\ \mathit{e\text{-}cfrac}\ (n - \mathit{2}))$
and $a \equiv (\lambda n.\ \mathit{if}\ n\ \mathit{mod}\ \mathit{3} = \mathit{2}\ \mathit{then}\ \mathit{2} * (\mathit{Suc}\ n\ \mathit{div}\ \mathit{3})\ \mathit{else}\ \mathit{1})$
begin

lemma
assumes $n \geq \mathit{2}$
shows *p-rec*: $p\ n = a\ (n - \mathit{2}) * p\ (n - \mathit{1}) + p\ (n - \mathit{2})$ (**is** *?th1*)
and *q-rec*: $q\ n = a\ (n - \mathit{2}) * q\ (n - \mathit{1}) + q\ (n - \mathit{2})$ (**is** *?th2*)
proof −
have *n-minus-3*: $n - \mathit{3} = n - \mathit{Suc}\ (\mathit{Suc}\ (\mathit{Suc}\ \mathit{0}))$
by (*simp add*: *numeral-3-eq-3*)
consider $n = \mathit{2} \mid n = \mathit{3} \mid n \geq \mathit{4}$
using *assms* **by** *force*
hence $?th1 \wedge ?th2$
by *cases* (*auto simp*: *p-def q-def cfrac-nth-e a-def conv-num-rec conv-denom-rec n-minus-3*)
thus *?th1 ?th2* **by** *blast*+
qed

lemma
assumes $n \geq \mathit{1}$
shows *p-rec0*: $p\ (\mathit{3} * n) = p\ (\mathit{3} * n - \mathit{1}) + p\ (\mathit{3} * n - \mathit{2})$
and *q-rec0*: $q\ (\mathit{3} * n) = q\ (\mathit{3} * n - \mathit{1}) + q\ (\mathit{3} * n - \mathit{2})$
proof −
define n' **where** $n' = n - \mathit{1}$
from *assms* **have** $(\mathit{3} * n' + \mathit{1})\ \mathit{mod}\ \mathit{3} \neq \mathit{2}$
by *presburger*
also have $(\mathit{3} * n' + \mathit{1}) = \mathit{3} * n - \mathit{2}$
using *assms* **by** (*simp add*: n'*-def*)
finally show $p\ (\mathit{3} * n) = p\ (\mathit{3} * n - \mathit{1}) + p\ (\mathit{3} * n - \mathit{2})$
$q\ (\mathit{3} * n) = q\ (\mathit{3} * n - \mathit{1}) + q\ (\mathit{3} * n - \mathit{2})$
using *assms* **by** (*subst p-rec q-rec*; *simp add*: *a-def*)+
qed

lemma
assumes $n \geq \mathit{1}$
shows *p-rec1*: $p\ (\mathit{3} * n + \mathit{1}) = \mathit{2} * \mathit{int}\ n * p\ (\mathit{3} * n) + p\ (\mathit{3} * n - \mathit{1})$
and *q-rec1*: $q\ (\mathit{3} * n + \mathit{1}) = \mathit{2} * \mathit{int}\ n * q\ (\mathit{3} * n) + q\ (\mathit{3} * n - \mathit{1})$
proof −
define n' **where** $n' = n - \mathit{1}$
from *assms* **have** $(\mathit{3} * n' + \mathit{2})\ \mathit{mod}\ \mathit{3} = \mathit{2}$
by *presburger*
also have $(\mathit{3} * n' + \mathit{2}) = \mathit{3} * n - \mathit{1}$
using *assms* **by** (*simp add*: n'*-def*)
finally show $p\ (\mathit{3} * n + \mathit{1}) = \mathit{2} * \mathit{int}\ n * p\ (\mathit{3} * n) + p\ (\mathit{3} * n - \mathit{1})$
$q\ (\mathit{3} * n + \mathit{1}) = \mathit{2} * \mathit{int}\ n * q\ (\mathit{3} * n) + q\ (\mathit{3} * n - \mathit{1})$
using *assms* **by** (*subst p-rec q-rec*; *simp add*: *a-def*)+
qed

lemma *p-rec2*: *p* (*3* ∗ *n* + *2*) = *p* (*3* ∗ *n* + *1*) + *p* (*3* ∗ *n*)
  and *q-rec2*: *q* (*3* ∗ *n* + *2*) = *q* (*3* ∗ *n* + *1*) + *q* (*3* ∗ *n*)
  by (*subst p-rec q-rec*; *simp add*: *a-def nat-mult-distrib nat-add-distrib*)+

lemma *A-0*: *A 0* = *exp 1* − *1* **and** *B-0*: *B 0* = *1* **and** *C-0*: *C 0* = *2* − *exp 1*
proof −
  have (*exp has-integral* (*exp 1* − *exp 0*)) {*0..1*::*real*}
    by (*intro fundamental-theorem-of-calculus*)
      (*auto intro*!: *derivative-eq-intros*
        simp flip: *has-real-derivative-iff-has-vector-derivative*)
  thus *A 0* = *exp 1* − *1* **by** (*simp add*: *A-def has-integral-iff*)

  have ((λ*x*. *exp x* ∗ *x*) *has-integral* (*exp 1* ∗ (*1* − *1*) − *exp 0* ∗ (*0* − *1*))) {*0..1*::*real*}
    by (*intro fundamental-theorem-of-calculus*)
      (*auto intro*!: *derivative-eq-intros*
        simp flip: *has-real-derivative-iff-has-vector-derivative simp*: *algebra-simps*)
  thus *B 0* = *1* **by** (*simp add*: *B-def has-integral-iff*)

  have ((λ*x*. *exp x* ∗ (*x* − *1*)) *has-integral* (*exp 1* ∗ (*1* − *2*) − *exp 0* ∗ (*0* − *2*))) {*0..1*::*real*}
    by (*intro fundamental-theorem-of-calculus*)
      (*auto intro*!: *derivative-eq-intros*
        simp flip: *has-real-derivative-iff-has-vector-derivative simp*: *algebra-simps*)
  thus *C 0* = *2* − *exp 1* **by** (*simp add*: *C-def has-integral-iff*)
qed

lemma *A-bound*: *norm* (*A n*) ≤ *exp 1* / *fact n*
proof −
  have *norm* (*exp t* ∗ *t* ^ *n* ∗ (*t* − *1*) ^ *n* / *fact n*) ≤ *exp 1* ∗ *1* ^ *n* ∗ *1* ^ *n* / *fact n*
    if *t* ∈ {*0..1*} **for** *t* :: *real* **using** *that* **unfolding** *norm-mult norm-divide norm-power norm-fact*
    by (*intro mult-mono divide-right-mono power-mono*) *auto*
  hence *norm* (*A n*) ≤ *exp 1* / *fact n* ∗ (*1* − *0*)
    unfolding *A-def* **by** (*intro integral-bound*) (*auto intro*!: *continuous-intros*)
  thus *?thesis* **by** *simp*
qed

lemma *B-bound*: *norm* (*B n*) ≤ *exp 1* / *fact n*
proof −
  have *norm* (*exp t* ∗ *t* ^ *Suc n* ∗ (*t* − *1*) ^ *n* / *fact n*) ≤ *exp 1* ∗ *1* ^ *Suc n* ∗ *1* ^ *n* / *fact n*
    if *t* ∈ {*0..1*} **for** *t* :: *real* **using** *that* **unfolding** *norm-mult norm-divide norm-power norm-fact*
    by (*intro mult-mono divide-right-mono power-mono*) *auto*
  hence *norm* (*B n*) ≤ *exp 1* / *fact n* ∗ (*1* − *0*)
    unfolding *B-def* **by** (*intro integral-bound*) (*auto intro*!: *continuous-intros*)
  thus *?thesis* **by** *simp*

qed

lemma *C-bound*: *norm* (*C n*) ≤ *exp 1* / *fact n*
proof −
  have *norm* (*exp t* ∗ *t* ^ *n* ∗ (*t* − *1*) ^ *Suc n* / *fact n*) ≤ *exp 1* ∗ *1* ^ *n* ∗ *1* ^ *Suc n* / *fact n*
      if *t* ∈ {*0..1*} **for** *t* :: *real* **using** *that* **unfolding** *norm-mult norm-divide norm-power norm-fact*
    by (*intro mult-mono divide-right-mono power-mono*) *auto*
  hence *norm* (*C n*) ≤ *exp 1* / *fact n* ∗ (*1* − *0*)
    unfolding *C-def* **by** (*intro integral-bound*) (*auto intro*!: *continuous-intros*)
  thus *?thesis* **by** *simp*
qed

lemma *A-Suc*: *A* (*Suc n*) = −*B n* − *C n*
proof −
  let *?g* = λ*x*. *x* ^ *Suc n* ∗ (*x* − *1*) ^ *Suc n* ∗ *exp x* / *fact* (*Suc n*)
  have *pos*: *fact n* + *real n* ∗ *fact n* > *0* **by** (*intro add-pos-nonneg*) *auto*
  have *A* (*Suc n*) + *B n* + *C n* =
          integral {*0..1*} (λ*x*. *exp x* ∗ *x* ^ *Suc n* ∗ (*x* − *1*) ^ *Suc n* / *fact* (*Suc n*) +
             exp x ∗ *x* ^ *Suc n* ∗ (*x* − *1*) ^ *n* / *fact n* + *exp x* ∗ *x* ^ *n* ∗ (*x* − *1*) ^ *Suc n* / *fact n*)
    unfolding *A-def B-def C-def*
    apply (*subst integral-add* [*symmetric*])
    subgoal
      by (*auto intro*!: *integrable-continuous-real continuous-intros*)
    subgoal
      by (*auto intro*!: *integrable-continuous-real continuous-intros*)
    apply (*subst integral-add* [*symmetric*])
      apply (*auto intro*!: *integrable-continuous-real continuous-intros*)
    done
  also have . . . = *integral* {*0..1*} (λ*x*. *exp x* / *fact* (*Suc n*) ∗
                  (*x* ^ *Suc n* ∗ (*x* − *1*) ^ *Suc n* + *Suc n* ∗ *x* ^ *Suc n* ∗ (*x* − *1*) ^ *n* +
                     Suc n ∗ *x* ^ *n* ∗ (*x* − *1*) ^ *Suc n*))
    (**is** - = *integral* - *?f*)
    apply (*simp add*: *divide-simps*)
    apply (*simp add*: *field-simps*)*?*
    done
  also have (*?f has-integral* (*?g 1* − *?g 0*)) {*0..1*}
    apply (*intro fundamental-theorem-of-calculus*)
    subgoal
      by *simp*
    unfolding *has-real-derivative-iff-has-vector-derivative* [*symmetric*]
    apply (*rule derivative-eq-intros refl* | *simp*)+
    apply (*simp add*: *algebra-simps*)*?*
    done
  hence *integral* {*0..1*} *?f* = *0*
    by (*simp add*: *has-integral-iff*)
  finally show *?thesis* **by** *simp*

```
qed

lemma B-Suc: B (Suc n) = −2 ∗ Suc n ∗ A (Suc n) + C n
proof −
  let ?g = λx. x ^ Suc n ∗ (x − 1) ^ (n+2) ∗ exp x / fact (Suc n)
  have pos: fact n + real n ∗ fact n > 0 by (intro add-pos-nonneg) auto
  have B (Suc n) + 2 ∗ Suc n ∗ A (Suc n) − C n =
          integral {0..1} (λx. exp x ∗ x^(n+2) ∗ (x − 1)^(n+1) / fact (Suc n) + 2
∗ Suc n ∗
            exp x ∗ x ^ Suc n ∗ (x − 1) ^ Suc n / fact (Suc n) − exp x ∗ x ^ n ∗ (x
− 1) ^ Suc n / fact n)
    unfolding A-def B-def C-def integral-mult-right [symmetric]
    apply (subst integral-add [symmetric])
    subgoal
      by (auto intro!: integrable-continuous-real continuous-intros)
    subgoal
      by (auto intro!: integrable-continuous-real continuous-intros)
    apply (subst integral-diff [symmetric])
     apply (auto intro!: integrable-continuous-real continuous-intros simp: mult-ac)
    done
  also have . . . = integral {0..1} (λx. exp x / fact (Suc n) ∗
                    (x^(n+2) ∗ (x − 1)^(n+1) + 2 ∗ Suc n ∗ x ^ Suc n ∗ (x − 1) ^
Suc n −
                     Suc n ∗ x ^ n ∗ (x − 1) ^ Suc n))
    (is - = integral - ?f)
    apply (simp add: divide-simps)
    apply (simp add: field-simps)?
    done
  also have (?f has-integral (?g 1 − ?g 0)) {0..1}
    apply (intro fundamental-theorem-of-calculus)
     apply (simp; fail)
    unfolding has-real-derivative-iff-has-vector-derivative [symmetric]
    apply (rule derivative-eq-intros refl | simp)+
    apply (simp add: algebra-simps)?
    done
  hence integral {0..1} ?f = 0
    by (simp add: has-integral-iff)
  finally show ?thesis by (simp add: algebra-simps)
qed

lemma C-Suc: C n = B n − A n
  unfolding A-def B-def C-def
  by (subst integral-diff [symmetric])
     (auto intro!: integrable-continuous-real continuous-intros simp: field-simps)

lemma unfold-add-numeral: c ∗ n + numeral b = Suc (c ∗ n + pred-numeral b)
  by simp

lemma ABC:
```

```
  A n = q (3 ∗ n) ∗ exp 1 − p (3 ∗ n) ∧
   B n = p (Suc (3 ∗ n)) − q (Suc (3 ∗ n)) ∗ exp 1 ∧
   C n = p (Suc (Suc (3 ∗ n))) − q (Suc (Suc (3 ∗ n))) ∗ exp 1
proof (induction n)
  case 0
  thus ?case by (simp add: A-0 B-0 C-0 a-def p-def q-def cfrac-nth-e)
next
  case (Suc n)
  note [simp] =
    conjunct1[OF Suc.IH] conjunct1[OF conjunct2[OF Suc.IH]] conjunct2[OF con-
junct2[OF Suc.IH]]
  have [simp]: 3 + m = Suc (Suc (Suc m)) for m by simp

  have A′: A (Suc n) = of-int (q (3 ∗ Suc n)) ∗ exp 1 − of-int (p (3 ∗ Suc n))
    unfolding A-Suc
    by (subst p-rec0 q-rec0, simp)+ (auto simp: algebra-simps)
  have B′: B (Suc n) = of-int (p (3 ∗ Suc n + 1)) − of-int (q (3 ∗ Suc n + 1)) ∗
exp 1
    unfolding B-Suc
    by (subst p-rec1 q-rec1 p-rec0 q-rec0, simp)+ (auto simp: algebra-simps A-Suc)
  have C′: C (Suc n) = of-int (p (3∗Suc n+2)) − of-int (q (3∗Suc n+2)) ∗ exp 1
    unfolding A-Suc B-Suc C-Suc using p-rec2[of n] q-rec2[of n]
    by ((subst p-rec2 q-rec2)+, (subst p-rec0 q-rec0 p-rec1 q-rec1, simp)+)
       (auto simp: algebra-simps A-Suc B-Suc)
  from A′ B′ C′ show ?case by simp
qed

lemma q-pos: q n > 0 if n ≠ 1
  using that by (auto simp: q-def)

lemma conv-diff-exp-bound: norm (exp 1 − p n / q n) ≤ exp 1 / fact (n div 3)
proof (cases n = 1)
  case False
  define n′ where n′ = n div 3
  consider n mod 3 = 0 | n mod 3 = 1 | n mod 3 = 2
    by force
  hence diff [unfolded n′-def]: q n ∗ exp 1 − p n =
    (if n mod 3 = 0 then A n′ else if n mod 3 = 1 then −B n′ else −C n′)
  proof cases
    assume n mod 3 = 0
    hence 3 ∗ n′ = n unfolding n′-def by presburger
    with ABC[of n′] show ?thesis by auto
  next
    assume ∗: n mod 3 = 1
    hence Suc (3 ∗ n′) = n unfolding n′-def by presburger
    with ∗ ABC[of n′] show ?thesis by auto
  next
    assume ∗: n mod 3 = 2
    hence Suc (Suc (3 ∗ n′)) = n unfolding n′-def by presburger
```

with ∗ *ABC*[*of n'*] **show** *?thesis* **by** *auto*
qed

note [[*linarith-split-limit* = *0*]]
have *norm* ((*q n* ∗ *exp 1* − *p n*) / *q n*) ≤ *exp 1* / *fact* (*n div 3*) / *1* **unfolding** *diff norm-divide*
using *A-bound*[*of n div 3*] *B-bound*[*of n div 3*] *C-bound*[*of n div 3*] *q-pos*[*OF* ‹*n* ≠ *1*›]
by (*subst frac-le*) (*auto simp*: *of-nat-ge-1-iff*)
also have (*q n* ∗ *exp 1* − *p n*) / *q n* = *exp 1* − *p n* / *q n*
using *q-pos*[*OF* ‹*n* ≠ *1*›] **by** (*simp add*: *divide-simps*)
finally show *?thesis* **by** *simp*
qed (*auto simp*: *p-def q-def*)

theorem *e-cfrac*: *cfrac-lim e-cfrac* = *exp 1*
proof −
have *num*: *conv-num e-cfrac n* = *p* (*n* + *2*)
and *denom*: *conv-denom e-cfrac n* = *q* (*n* + *2*) **for** *n*
by (*simp-all add*: *p-def q-def*)

have (λ*n*. *exp 1* − *p n* / *q n*) ⟶ *0*
proof (*rule Lim-null-comparison*)
show *eventually* (λ*n*. *norm* (*exp 1* − *p n* / *q n*) ≤ *exp 1* / *fact* (*n div 3*)) *at-top*
using *conv-diff-exp-bound* **by** (*intro always-eventually*) *auto*
show (λ*n*. *exp 1* / *fact* (*n div 3*) :: *real*) ⟶ *0*
by (*rule real-tendsto-divide-at-top tendsto-const filterlim-div-nat-at-top filterlim-ident filterlim-compose*[*OF fact-real-at-top*])+ *auto*
qed
moreover have *eventually* (λ*n*. *exp 1* − *p n* / *q n* = *exp 1* − *conv e-cfrac* (*n* − *2*)) *at-top*
using *eventually-ge-at-top*[*of 2*]
proof *eventually-elim*
case (*elim n*)
with *num*[*of n* − *2*] *denom*[*of n* − *2*] *wf* **show** *?case*
by (*simp add*: *eval-nat-numeral Suc-diff-Suc conv-num-denom*)
qed
ultimately have (λ*n*. *exp 1* − *conv e-cfrac* (*n* − *2*)) ⟶ *0*
using *Lim-transform-eventually* **by** *fast*
hence (λ*n*. *exp 1* − (*exp 1* − *conv e-cfrac* (*Suc* (*Suc n*) − *2*))) ⟶ *exp 1* − *0*
by (*subst filterlim-sequentially-Suc*)+ (*intro tendsto-diff tendsto-const*)
hence *conv e-cfrac* ⟶ *exp 1* **by** *simp*
moreover have *conv e-cfrac* ⟶ *cfrac-lim e-cfrac*
by (*intro LIMSEQ-cfrac-lim wf*) *auto*
ultimately have *exp 1* = *cfrac-lim e-cfrac*
by (*rule LIMSEQ-unique*)
thus *?thesis* **..**
qed

corollary *e-cfrac-altdef*: *e-cfrac* = *cfrac-of-real* (*exp 1*)

by (*metis e-cfrac cfrac-infinite-iff cfrac-length-e cfrac-of-real-cfrac-lim-irrational*)

This also provides us with a nice proof that e is not rational and not a quadratic irrational either.

```
corollary exp1-irrational: (exp 1 :: real) ∉ ℚ
  by (metis cfrac-length-e e-cfrac cfrac-infinite-iff)

corollary exp1-not-quadratic-irrational: ¬quadratic-irrational (exp 1 :: real)
proof −
  have range (λn. 2 ∗ (int n + 1)) ⊆ range (cfrac-nth e-cfrac)
  proof safe
    fix n :: nat
    have cfrac-nth e-cfrac (3∗n+2) ∈ range (cfrac-nth e-cfrac)
      by blast
    also have (3 ∗ n + 2) mod 3 = 2
      by presburger
    hence cfrac-nth e-cfrac (3∗n+2) = 2 ∗ (int n + 1)
      by (simp add: cfrac-nth-e)
    finally show 2 ∗ (int n + 1) ∈ range (cfrac-nth e-cfrac) .
  qed
  moreover have infinite (range (λn. 2 ∗ (int n + 1)))
    by (subst finite-image-iff) (auto intro!: injI)
  ultimately have infinite (range (cfrac-nth e-cfrac))
    using finite-subset by blast
  thus ?thesis using quadratic-irrational-cfrac-nth-range-finite[of e-cfrac]
    by (auto simp: e-cfrac)
qed

end
end
```

4 Continued fraction expansions for square roots of naturals

```
theory Sqrt-Nat-Cfrac
imports
  Quadratic-Irrationals
  HOL−Library.While-Combinator
  HOL−Library.IArray
begin
```

In this section, we shall explore the continued fraction expansion of $\sqrt{D}$, where D is a natural number.

```
lemma butlast-nth [simp]: n < length xs − 1 ⟹ butlast xs ! n = xs ! n
  by (induction xs arbitrary: n) (auto simp: nth-Cons split: nat.splits)
```

The following is the length of the period in the continued fraction expansion of $\sqrt{D}$ for a natural number D.

definition *sqrt-nat-period-length* :: *nat* ⇒ *nat* **where**
sqrt-nat-period-length D =
(*if is-square D then 0*
else (*LEAST l. l* > *0* ∧ (∀ *n. cfrac-nth* (*cfrac-of-real* (*sqrt D*)) (*Suc n* + *l*) =
cfrac-nth (*cfrac-of-real* (*sqrt D*)) (*Suc n*))))

Next, we define a more workable representation for the continued fraction expansion of $\sqrt{D}$ consisting of the period length, the natural number $\lfloor\sqrt{D}\rfloor$, and the content of the period.

definition *sqrt-cfrac-info* :: *nat* ⇒ *nat* × *nat* × *nat list* **where**
sqrt-cfrac-info D =
(*sqrt-nat-period-length D*, *Discrete.sqrt D*,
map (λ*n. nat* (*cfrac-nth* (*cfrac-of-real* (*sqrt D*)) (*Suc n*))) [*0..*<*sqrt-nat-period-length D*])

lemma *sqrt-nat-period-length-square* [*simp*]: *is-square D* ⟹ *sqrt-nat-period-length D* = *0*
by (*auto simp*: *sqrt-nat-period-length-def*)

definition *sqrt-cfrac* :: *nat* ⇒ *cfrac*
where *sqrt-cfrac D* = *cfrac-of-real* (*sqrt* (*real D*))

context
fixes *D D′* :: *nat*
defines *D′* ≡ *nat* ⌊*sqrt D*⌋
begin

A number $\alpha = \frac{\sqrt{D}+p}{q}$ for $p,\ q \in \mathbb{N}$ is called a *reduced quadratic surd* if $\alpha > 1$ and $bar\alpha \in (-1;0)$, where $\bar{\alpha}$ denotes the conjugate $\frac{-\sqrt{D}+p}{q}$.
It is furthermore called *associated* to D if q divides $D - p^2$.

definition *red-assoc* :: *nat* × *nat* ⇒ *bool* **where**
red-assoc = (λ(*p*, *q*).
q > *0* ∧ *q dvd* (*D* − *p*²) ∧ (*sqrt D* + *p*) / *q* > *1* ∧ (−*sqrt D* + *p*) / *q* ∈ {−*1*<..<*0*})

The following two functions convert between a surd represented as a pair of natural numbers and the actual real number and its conjugate:

definition *surd-to-real* :: *nat* × *nat* ⇒ *real*
where *surd-to-real* = (λ(*p*, *q*). (*sqrt D* + *p*) / *q*)

definition *surd-to-real-cnj* :: *nat* × *nat* ⇒ *real*
where *surd-to-real-cnj* = (λ(*p*, *q*). (−*sqrt D* + *p*) / *q*)

The next function performs a single step in the continued fraction expansion of $\sqrt{D}$.

definition *sqrt-remainder-step* :: *nat* × *nat* ⇒ *nat* × *nat* **where**

sqrt-remainder-step = (λ(*p*, *q*). *let X* = (*p* + *D*′) *div q*; *p*′ = *X* ∗ *q* − *p in* (*p*′, (*D* − *p*′²) *div q*))

If we iterate this step function starting from the surd $\frac{1}{\sqrt{D}-\lfloor\sqrt{D}\rfloor}$, we get the entire expansion.

definition *sqrt-remainder-surd* :: *nat* ⇒ *nat* × *nat*
  where *sqrt-remainder-surd* = (λ*n*. (*sqrt-remainder-step* ^^ *n*) (*D*′, *D* − *D*′²))

context
  fixes *sqrt-cfrac-nth* :: *nat* ⇒ *nat* **and** *l*
  assumes *nonsquare*: ¬*is-square D*
  defines *sqrt-cfrac-nth* ≡ (λ*n*. *case sqrt-remainder-surd n of* (*p*, *q*) ⇒ (*D*′ + *p*) *div q*)
  defines *l* ≡ *sqrt-nat-period-length D*
begin

lemma *D*′*-pos*: *D*′ > *0*
  using *nonsquare* **by** (*auto simp*: *D*′*-def of-nat-ge-1-iff intro*: *Nat.gr0I*)

lemma *D*′*-sqr-less-D*: *D*′² < *D*
proof −
  have *D*′ ≤ *sqrt D* **by** (*auto simp*: *D*′*-def*)
  hence *real D*′ ^ *2* ≤ *sqrt D* ^ *2* **by** (*intro power-mono*) *auto*
  also have … = *D* **by** *simp*
  finally have *D*′² ≤ *D* **by** *simp*
  moreover from *nonsquare* **have** *D* ≠ *D*′² **by** *auto*
  ultimately show *?thesis* **by** *simp*
qed

lemma *red-assoc-imp-irrat*:
  assumes *red-assoc pq*
  shows *surd-to-real pq* ∉ ℚ
proof
  assume *rat*: *surd-to-real pq* ∈ ℚ
  with *assms rat* **show** *False* **using** *irrat-sqrt-nonsquare*[*OF nonsquare*]
    by (*auto simp*: *field-simps red-assoc-def surd-to-real-def divide-in-Rats-iff2 add-in-Rats-iff1*)
qed

lemma *surd-to-real-cnj-irrat*:
  assumes *red-assoc pq*
  shows *surd-to-real-cnj pq* ∉ ℚ
proof
  assume *rat*: *surd-to-real-cnj pq* ∈ ℚ
  with *assms rat* **show** *False* **using** *irrat-sqrt-nonsquare*[*OF nonsquare*]
    by (*auto simp*: *field-simps red-assoc-def surd-to-real-cnj-def divide-in-Rats-iff2 diff-in-Rats-iff1*)
qed

lemma *surd-to-real-nonneg* [*intro*]: *surd-to-real pq* $\geq$ *0*
by (*auto simp*: *surd-to-real-def case-prod-unfold divide-simps intro*!: *divide-nonneg-nonneg*)

lemma *surd-to-real-pos* [*intro*]: *red-assoc pq* $\Longrightarrow$ *surd-to-real pq* > *0*
by (*auto simp*: *surd-to-real-def case-prod-unfold divide-simps red-assoc-def*
intro!: *divide-nonneg-nonneg*)

lemma *surd-to-real-nz* [*simp*]: *red-assoc pq* $\Longrightarrow$ *surd-to-real pq* $\neq$ *0*
by (*auto simp*: *surd-to-real-def case-prod-unfold divide-simps red-assoc-def*
intro!: *divide-nonneg-nonneg*)

lemma *surd-to-real-cnj-nz* [*simp*]: *red-assoc pq* $\Longrightarrow$ *surd-to-real-cnj pq* $\neq$ *0*
using *surd-to-real-cnj-irrat*[*of pq*] **by** *auto*

lemma *red-assoc-step*:
assumes *red-assoc pq*
defines $X \equiv (D' + fst\ pq)\ div\ snd\ pq$
defines $pq' \equiv$ *sqrt-remainder-step pq*
shows *red-assoc pq'*
surd-to-real pq' = *1* / *frac* (*surd-to-real pq*)
surd-to-real-cnj pq' = *1* / (*surd-to-real-cnj pq* − *X*)
$X > 0$ $X * snd\ pq \leq 2 * D'$ $X = nat\ \lfloor surd\text{-}to\text{-}real\ pq \rfloor$
$X = nat\ \lfloor -1\ /\ surd\text{-}to\text{-}real\text{-}cnj\ pq' \rfloor$
proof −
obtain *p q* **where** [*simp*]: *pq* = (*p*, *q*) **by** (*cases pq*)
obtain *p' q'* **where** [*simp*]: *pq'* = (*p'*, *q'*) **by** (*cases pq'*)
define α **where** $\alpha = (sqrt\ D + p)\ /\ q$
define α' **where** $\alpha' = 1\ /\ frac\ \alpha$
define *cnj*-α' **where** *cnj*-$\alpha' = (-sqrt\ D + (X * q - int\ p))\ /\ ((D - (X * q - int\ p)^2)\ div\ q)$
from *assms*(*1*) **have** $\alpha > 0$ $q > 0$
by (*auto simp*: α*-def red-assoc-def*)
from *assms*(*1*) *nonsquare* **have** $\alpha \notin \mathbb{Q}$
by (*auto simp*: α*-def red-assoc-def divide-in-Rats-iff2 add-in-Rats-iff2 irrat-sqrt-nonsquare*)
hence α'*-pos*: *frac* $\alpha > 0$ **using** *Ints-subset-Rats* **by** *auto*
from ‹*pq'* = (*p'*, *q'*)› **have** *p'-def*: $p' = X * q - p$ **and** *q'-def*: $q' = (D - p'^2)\ div\ q$
unfolding *pq'-def sqrt-remainder-step-def X-def* **by** (*auto simp*: *Let-def add-ac*)

have $D' + p = \lfloor sqrt\ D + p \rfloor$
by (*auto simp*: *D'-def*)
also have ... $div\ int\ q = \lfloor (sqrt\ D + p)\ /\ q \rfloor$
by (*subst floor-divide-real-eq-div* [*symmetric*]) *auto*
finally have *X-altdef*: $X = nat\ \lfloor (sqrt\ D + p)\ /\ q \rfloor$
unfolding *X-def zdiv-int* [*symmetric*] **by** *auto*

have *nz*: *sqrt* (*real D*) + ($X * q$ − *real p*) $\neq$ *0*
proof
assume *sqrt* (*real D*) + ($X * q$ − *real p*) = *0*

hence *sqrt* (*real D*) = *real p* − *X* ∗ *q* **by** (*simp add*: *algebra-simps*)
also have . . . ∈ ℚ **by** *auto*
finally show *False* **using** *irrat-sqrt-nonsquare nonsquare* **by** *blast*
qed

from *assms*(*1*) **have** *real* (*p* ^ *2*) ≤ *sqrt D* ^ *2*
unfolding *of-nat-power* **by** (*intro power-mono*) (*auto simp*: *red-assoc-def field-simps*)
also have *sqrt D* ^ *2* = *D* **by** *simp*
finally have p^2 ≤ *D* **by** (*subst* (*asm*) *of-nat-le-iff*)

have *frac* α = α − *X*
by (*simp add*: *X-altdef frac-def* α*-def*)
also have . . . = (*sqrt D* − (*X* ∗ *q* − *int p*)) / *q*
using ‹*q* > *0*› **by** (*simp add*: *field-simps* α*-def*)
finally have *1* / *frac* α = *q* / (*sqrt D* − (*X* ∗ *q* − *int p*))
by *simp*
also have . . . = *q* ∗ (*sqrt D* + (*X* ∗ *q* − *int p*)) /
((*sqrt D* − (*X* ∗ *q* − *int p*)) ∗ (*sqrt D* + (*X* ∗ *q* − *int p*))) (**is** - = *?A* / *?B*)
using *nz* **by** (*subst mult-divide-mult-cancel-right*) *auto*
also have *?B* = *real-of-int* (*D* − *int p* ^ *2* + *2* ∗ *X* ∗ *p* ∗ *q* − *int X* ^ *2* ∗ *q* ^ *2*)
by (*auto simp*: *algebra-simps power2-eq-square*)
also have *q dvd* (*D* − *p* ^ *2*) **using** *assms*(*1*) **by** (*auto simp*: *red-assoc-def*)
with ‹p^2 ≤ *D*› **have** *int q dvd* (*int D* − *int p* ^ *2*)
unfolding *of-nat-power* [*symmetric*] **by** (*subst of-nat-diff* [*symmetric*]) *auto*
hence *D* − *int p* ^ *2* + *2* ∗ *X* ∗ *p* ∗ *q* − *int X* ^ *2* ∗ *q* ^ *2* = *q* ∗ ((*D* − (*X* ∗ *q* − *int p*)^2) *div q*)
by (*auto simp*: *power2-eq-square algebra-simps*)
also have *?A* / . . . = (*sqrt D* + (*X* ∗ *q* − *int p*)) / ((*D* − (*X* ∗ *q* − *int p*)^2) *div q*)
unfolding *of-int-mult of-int-of-nat-eq*
by (*rule mult-divide-mult-cancel-left*) (*insert* ‹*q* > *0*›, *auto*)
finally have α′: α′ = . . . **by** (*simp add*: α′*-def*)

have *dvd*: *q dvd* (*D* − (*X* ∗ *q* − *int p*)^2)
using *assms*(*1*) ‹*int q dvd* (*int D* − *int p* ^ *2*)›
by (*auto simp*: *power2-eq-square algebra-simps*)

have *X* ≤ (*sqrt D* + *p*) / *q* **unfolding** *X-altdef* **by** *simp*
moreover have *X* ≠ (*sqrt D* + *p*) / *q*
proof
assume *X* = (*sqrt D* + *p*) / *q*
hence *sqrt D* = *q* ∗ *X* − *real p* **using** ‹*q* > *0*› **by** (*auto simp*: *field-simps*)
also have . . . ∈ ℚ **by** *auto*
finally show *False* **using** *irrat-sqrt-nonsquare*[*OF nonsquare*] **by** *simp*
qed
ultimately have *X* < (*sqrt D* + *p*) / *q* **by** *simp*
hence ∗: (*X* ∗ *q* − *int p*) < *sqrt D*

using ‹$q > 0$› **by** (*simp add*: *field-simps*)
moreover
have *pos*: *real-of-int* (*int D* − (*int X* ∗ *int q* − *int p*)^2) > *0*
proof (*cases X* ∗ *q* ≥ *p*)
case *True*
hence *real p* ≤ *real X* ∗ *real q* **unfolding** *of-nat-mult* [*symmetric*] *of-nat-le-iff*
.
hence *real-of-int* ((*X* ∗ *q* − *int p*) ^ *2*) < *sqrt D* ^ *2* **using** ∗
unfolding *of-int-power* **by** (*intro power-strict-mono*) *auto*
also have ... = *D* **by** *simp*
finally show *?thesis* **by** *simp*
next
case *False*
hence *less*: *real X* ∗ *real q* < *real p*
unfolding *of-nat-mult* [*symmetric*] *of-nat-less-iff* **by** *auto*
have (*real X* ∗ *real q* − *real p*)^2 = (*real p* − *real X* ∗ *real q*)^2
by (*simp add*: *power2-eq-square algebra-simps*)
also have ... ≤ *real p* ^ *2* **using** *less* **by** (*intro power-mono*) *auto*
also have ... < *sqrt D* ^ *2*
using ‹$q > 0$› *assms*(*1*) **unfolding** *of-int-power*
by (*intro power-strict-mono*) (*auto simp*: *red-assoc-def field-simps*)
also have ... = *D* **by** *simp*
finally show *?thesis* **by** *simp*
qed
hence *pos'*: *int D* − (*int X* ∗ *int q* − *int p*)^2 > *0*
by (*subst* (*asm*) *of-int-0-less-iff*)
from *pos* **have** *real-of-int* ((*int D* − (*int X* ∗ *int q* − *int p*)^2) *div q*) > *0*
using ‹$q > 0$› *dvd* **by** (*subst real-of-int-div*) (*auto intro*!: *divide-pos-pos*)
ultimately have *cnj-neg*: *cnj-α'* < *0* **unfolding** *cnj-α'-def* **using** *dvd*
unfolding *of-int-0-less-iff* **by** (*intro divide-neg-pos*) *auto*

have (*p* − *sqrt D*) / *q* < *0*
using *assms*(*1*) **by** (*auto simp*: *red-assoc-def X-altdef le-nat-iff*)
also have *X* ≥ *1*
using *assms*(*1*) **by** (*auto simp*: *red-assoc-def X-altdef le-nat-iff*)
hence *0* ≤ *real X* − *1* **by** *simp*
finally have *q* < *sqrt D* + *int q* ∗ *X* − *p*
using ‹$q > 0$› **by** (*simp add*: *field-simps*)
hence *q* ∗ (*sqrt D* − (*int q* ∗ *X* − *p*)) < (*sqrt D* + (*int q* ∗ *X* − *p*)) ∗ (*sqrt D* − (*int q* ∗ *X* − *p*))
using ∗ **by** (*intro mult-strict-right-mono*) (*auto simp*: *red-assoc-def X-altdef field-simps*)
also have ... = *D* − (*int q* ∗ *X* − *p*) ^ *2*
by (*simp add*: *power2-eq-square algebra-simps*)
finally have *cnj-α'* > −*1*
using *dvd pos* ‹$q > 0$› **by** (*simp add*: *real-of-int-div field-simps cnj-α'-def*)

from *cnj-neg* **and** *this* **have** *cnj-α'* ∈ {−*1*<..<*0*} **by** *auto*
have *α'* > *1* **using** ‹*frac α* > *0*›

by (*auto simp*: α'*-def field-simps frac-lt-1*)

have $0 = 1 + (-1 :: real)$
by *simp*
also have $1 + -1 < \alpha' + cnj\text{-}\alpha'$
using ‹$cnj\text{-}\alpha' > -1$› **and** ‹$\alpha' > 1$› **by** (*intro add-strict-mono*)
also have $\alpha' + cnj\text{-}\alpha' = 2 * (real\ X * q - real\ p) / ((int\ D - (int\ X * q - int\ p)^2)\ div\ int\ q)$
by (*simp add*: α' *cnj-*α'*-def add-divide-distrib* [*symmetric*])
finally have $real\ X * q - real\ p > 0$ **using** *pos dvd* ‹$q > 0$›
by (*subst* (*asm*) *zero-less-divide-iff*, *subst* (*asm*) (*1 2 3*) *real-of-int-div*)
(*auto simp*: *field-simps*)
hence $real\ (X * q) > real\ p$ **unfolding** *of-nat-mult* **by** *simp*
hence *p-less-Xq*: $p < X * q$ **by** (*simp only*: *of-nat-less-iff*)

from *pos'* **and** *p-less-Xq* **have** $int\ D > int\ ((X * q - p)^2)$
by (*subst of-nat-power*) (*auto simp*: *of-nat-diff*)
hence *pos''*: $D > (X * q - p)^2$ **unfolding** *of-nat-less-iff* **.**

from *dvd* **have** $int\ q\ dvd\ int\ (D - (X * q - p)^2)$
using *p-less-Xq pos''* **by** (*subst of-nat-diff*) (*auto simp*: *of-nat-diff*)
with *dvd* **have** *dvd'*: $q\ dvd\ (D - (X * q - p)^2)$
by *simp*

have α'*-altdef*: $\alpha' = (sqrt\ D + p') / q'$
using *dvd dvd' pos'' p-less-Xq* α'
by (*simp add*: *real-of-int-div p'-def q'-def real-of-nat-div mult-ac of-nat-diff*)
have *cnj-*α'*-altdef*: $cnj\text{-}\alpha' = (-sqrt\ D + p') / q'$
using *dvd dvd' pos'' p-less-Xq* **unfolding** *cnj-*α'*-def*
by (*simp add*: *real-of-int-div p'-def q'-def real-of-nat-div mult-ac of-nat-diff*)
from *dvd'* **have** *dvd''*: $q'\ dvd\ (D - p'^2)$
by (*auto simp*: *mult-ac p'-def q'-def*)
have $real\ ((D - p'^2)\ div\ q) > 0$ **unfolding** *p'-def*
by (*subst real-of-nat-div*[*OF dvd'*], *rule divide-pos-pos*) (*insert* ‹$q > 0$› *pos''*, *auto*)
hence $q' > 0$ **unfolding** *q'-def of-nat-0-less-iff* **.**

show *red-assoc pq'* **using** ‹$\alpha' > 1$› **and** ‹$cnj\text{-}\alpha' \in$ -› **and** *dvd''* **and** ‹$q' > 0$›
by (*auto simp*: *red-assoc-def* α'*-altdef cnj-*α'*-altdef*)

from *assms*(*1*) **have** $real\ p < sqrt\ D$
by (*auto simp add*: *field-simps red-assoc-def*)
hence $p \leq D'$ **unfolding** *D'-def* **by** *linarith*
with $*$ **have** $real\ (X * q) < sqrt\ (real\ D) + D'$
by *simp*
thus $X * snd\ pq \leq 2 * D'$ **unfolding** *D'-def* ‹$pq = (p, q)$› *snd-conv* **by** *linarith*

have $(sqrt\ D + p') / q' = \alpha'$
by (*rule* α'*-altdef* [*symmetric*])

also have $\alpha' = 1\ /\ frac\ ((sqrt\ D + p)\ /\ q)$
  by (*simp add*: *α′-def α-def*)
finally show *surd-to-real pq′* = *1* / *frac* (*surd-to-real pq*) **by** (*simp add*: *surd-to-real-def*)
from ‹$X \geq 1$› **show** $X > 0$ **by** *simp*
from *X-altdef* **show** $X = nat\ \lfloor surd\text{-}to\text{-}real\ pq \rfloor$ **by** (*simp add*: *surd-to-real-def*)

have $sqrt\ (real\ D) < real\ p + 1 * real\ q$
  using *assms*(*1*) **by** (*auto simp*: *red-assoc-def field-simps*)
also have $\ldots \leq real\ p + real\ X * real\ q$
  using ‹$X > 0$› **by** (*intro add-left-mono mult-right-mono*) (*auto simp*: *of-nat-ge-1-iff*)
finally have $sqrt\ (real\ D) < \ldots$ **.**

have $real\ p < sqrt\ D$
  using *assms*(*1*) **by** (*auto simp add*: *field-simps red-assoc-def*)
also have $\ldots \leq sqrt\ D + q * X$
  by *linarith*
finally have *less*: $real\ p < sqrt\ D + X * q$ **by** (*simp add*: *algebra-simps*)
moreover have $D + p * p' + X * q * sqrt\ D = q * q' + p * sqrt\ D + p' * sqrt\ D + X * p' * q$
  using *dvd′ pos″ p-less-Xq* ‹$q > 0$› **unfolding** *p′-def q′-def of-nat-mult of-nat-add*
  by (*simp add*: *power2-eq-square field-simps of-nat-diff real-of-nat-div*)
ultimately show ∗: $surd\text{-}to\text{-}real\text{-}cnj\ pq' = 1\ /\ (surd\text{-}to\text{-}real\text{-}cnj\ pq - X)$
  using ‹$q > 0$› ‹$q' > 0$› **by** (*auto simp*: *surd-to-real-cnj-def field-simps*)

have ∗∗: $a = nat\ \lfloor y \rfloor$ **if** $x \geq 0$ $x < 1$ $real\ a + x = y$ **for** $a :: nat$ **and** $x\ y :: real$
  using *that* **by** *linarith*
from *assms*(*1*) **have** *surd-to-real-cnj*: $surd\text{-}to\text{-}real\text{-}cnj\ (p,\ q) \in \{-1<..<0\}$
  by (*auto simp*: *surd-to-real-cnj-def red-assoc-def*)
have $surd\text{-}to\text{-}real\text{-}cnj\ (p,\ q) < X$
  using *assms*(*1*) *less* **by** (*auto simp*: *surd-to-real-cnj-def field-simps red-assoc-def*)
hence $real\ X = surd\text{-}to\text{-}real\text{-}cnj\ (p,\ q) - 1\ /\ surd\text{-}to\text{-}real\text{-}cnj\ (p',\ q')$ **using** ∗
  using *surd-to-real-cnj-irrat assms*(*1*) ‹*red-assoc pq′*› **by** (*auto simp*: *field-simps*)
thus $X = nat\ \lfloor -1\ /\ surd\text{-}to\text{-}real\text{-}cnj\ pq' \rfloor$ **using** *surd-to-real-cnj*
  by (*intro* ∗∗[*of* −*surd-to-real-cnj* (*p*, *q*)]) *auto*
qed

lemma *red-assoc-denom-2D*:
  assumes *red-assoc* (*p*, *q*)
  defines $X \equiv (D' + p)\ div\ q$
  assumes $X > D'$
  shows $q = 1$
proof −
  have $X * q \leq 2 * D'$ $X > 0$
    using *red-assoc-step*(*4*,*5*)[*OF assms*(*1*)] **by** (*simp-all add*: *X-def*)
  note *this*(*1*)
  also have $2 * D' < 2 * X$
    by (*intro mult-strict-left-mono assms*) *auto*
  finally have $q < 2$ **using** ‹$X > 0$› **by** *simp*
  moreover from *assms*(*1*) **have** $q > 0$ **by** (*auto simp*: *red-assoc-def*)

ultimately show *?thesis* **by** *simp*
qed

lemma *red-assoc-denom-1*:
assumes *red-assoc* $(p,\ 1)$
shows $p = D'$
proof −
from *assms* **have** $sqrt\ D > p$ $sqrt\ D < real\ p + 1$
by (*auto simp*: *red-assoc-def*)
thus $p = D'$ **unfolding** *D′-def*
by *linarith*
qed

lemma *red-assoc-begin*:
red-assoc $(D',\ D - D'^2)$
surd-to-real $(D',\ D - D'^2) = 1\ /\ frac\ (sqrt\ D)$
surd-to-real-cnj $(D',\ D - D'^2) = -1\ /\ (sqrt\ D + D')$
proof −
have *pos*: $D > 0$ $D' > 0$
using *nonsquare* **by** (*auto simp*: *D′-def of-nat-ge-1-iff intro*!: *Nat.gr0I*)

have $sqrt\ D \neq D'$
using *irrat-sqrt-nonsquare*[*OF nonsquare*] **by** *auto*
moreover have $sqrt\ D \geq 0$ **by** *simp*
hence $D' \leq sqrt\ D$ **unfolding** *D′-def* **by** *linarith*
ultimately have *less*: $D' < sqrt\ D$ **by** *simp*

have $sqrt\ D \neq D' + 1$
using *irrat-sqrt-nonsquare*[*OF nonsquare*] **by** *auto*
moreover have $sqrt\ D \geq 0$ **by** *simp*
hence $D' \geq sqrt\ D - 1$ **unfolding** *D′-def* **by** *linarith*
ultimately have *gt*: $D' > sqrt\ D - 1$ **by** *simp*

from *less* **have** $real\ D'\ \hat{}\ 2 < sqrt\ D\ \hat{}\ 2$ **by** (*intro power-strict-mono*) *auto*
also have $\ldots = D$ **by** *simp*
finally have *less′*: $D'^2 < D$ **unfolding** *of-nat-power* [*symmetric*] *of-nat-less-iff* **.**

moreover have $real\ D' * (real\ D' - 1) < sqrt\ D * (sqrt\ D - 1)$
using *less pos*
by (*intro mult-strict-mono diff-strict-right-mono*) (*auto simp*: *of-nat-ge-1-iff*)
hence $D'^2 + sqrt\ D < D' + D$
by (*simp add*: *field-simps power2-eq-square*)
moreover have $(sqrt\ D - 1) * sqrt\ D < real\ D' * (real\ D' + 1)$
using *pos gt* **by** (*intro mult-strict-mono*) *auto*
hence $D < sqrt\ D + D'^2 + D'$ **by** (*simp add*: *power2-eq-square field-simps*)
ultimately show *red-assoc* $(D',\ D - D'^2)$
by (*auto simp*: *red-assoc-def field-simps of-nat-diff less*)

have *frac*: $frac\ (sqrt\ D) = sqrt\ D - D'$ **unfolding** *frac-def D′-def*

by *auto*
show *surd-to-real* $(D', D - D'^2) = 1 \;/\; frac\ (sqrt\ D)$ **unfolding** *surd-to-real-def*
using *less less′ pos* **by** (*subst frac*) (*auto simp*: *of-nat-diff power2-eq-square field-simps*)

have *surd-to-real-cnj* $(D', D - D'^2) = -((sqrt\ D - D') \;/\; (D - D'^2))$
using *less less′ pos* **by** (*auto simp*: *surd-to-real-cnj-def field-simps*)
also have *real* $(D - D'^2) = (sqrt\ D - D') * (sqrt\ D + D')$
using *less′* **by** (*simp add*: *power2-eq-square algebra-simps of-nat-diff*)
also have $(sqrt\ D - D') \;/\; \ldots = 1 \;/\; (sqrt\ D + D')$
using *less* **by** (*subst nonzero-divide-mult-cancel-left*) *auto*
finally show *surd-to-real-cnj* $(D', D - D'^2) = -1 \;/\; (sqrt\ D + D')$ **by** *simp*
qed

lemma *cfrac-remainder-surd-to-real*:
assumes *red-assoc pq*
shows *cfrac-remainder* (*cfrac-of-real* (*surd-to-real pq*)) *n* =
surd-to-real ((*sqrt-remainder-step* ^^ *n*) *pq*)
using *assms*(*1*)
proof (*induction n arbitrary*: *pq*)
case *0*
hence *cfrac-lim* (*cfrac-of-real* (*surd-to-real pq*)) = *surd-to-real pq*
by (*intro cfrac-lim-of-real red-assoc-imp-irrat 0*)
thus *?case* **using** *0*
by *auto*
next
case (*Suc n*)
obtain *p q* **where** [*simp*]: *pq* = (*p*, *q*) **by** (*cases pq*)
have *surd-to-real* ((*sqrt-remainder-step* ^^ *Suc n*) *pq*) =
surd-to-real ((*sqrt-remainder-step* ^^ *n*) (*sqrt-remainder-step* (*p*, *q*)))
by (*subst funpow-Suc-right*) *auto*
also have ... = *cfrac-remainder* (*cfrac-of-real* (*surd-to-real* (*sqrt-remainder-step* (*p*, *q*)))) *n*
using *red-assoc-step*(*1*)[*of* (*p*, *q*)] *Suc.prems*
by (*intro Suc.IH* [*symmetric*]) (*auto simp*: *sqrt-remainder-step-def Let-def add-ac*)
also have *surd-to-real* (*sqrt-remainder-step* (*p*, *q*)) = *1* / *frac* (*surd-to-real* (*p*, *q*))
using *red-assoc-step*(*2*)[*of* (*p*, *q*)] *Suc.prems*
by (*auto simp*: *sqrt-remainder-step-def Let-def add-ac surd-to-real-def*)
also have *cfrac-of-real* ... = *cfrac-tl* (*cfrac-of-real* (*surd-to-real* (*p*, *q*)))
using *Suc.prems Ints-subset-Rats red-assoc-imp-irrat* **by** (*subst cfrac-tl-of-real*) *auto*
also have *cfrac-remainder* ... *n* = *cfrac-remainder* (*cfrac-of-real* (*surd-to-real* (*p*, *q*))) (*Suc n*)
by (*simp add*: *cfrac-drop-Suc-right cfrac-remainder-def*)
finally show *?case* **by** *simp*
qed

lemma *red-assoc-step′* [*intro*]: *red-assoc pq* ⟹ *red-assoc* (*sqrt-remainder-step pq*)
  using *red-assoc-step*(*1*)[*of pq*]
  by (*simp add*: *sqrt-remainder-step-def case-prod-unfold add-ac Let-def*)

lemma *red-assoc-steps* [*intro*]: *red-assoc pq* ⟹ *red-assoc* ((*sqrt-remainder-step* ^^ *n*) *pq*)
  by (*induction n*) *auto*

lemma *floor-sqrt-less-sqrt*: *D′* < *sqrt D*
proof −
  have *D′* ≤ *sqrt D* **unfolding** *D′-def* **by** *auto*
  moreover have *sqrt D* ≠ *D′*
    using *irrat-sqrt-nonsquare*[*OF nonsquare*] **by** *auto*
  ultimately show *?thesis* **by** *auto*
qed

lemma *red-assoc-bounds*:
  assumes *red-assoc pq*
  shows *pq* ∈ (*SIGMA p*:{*0*<..*D′*}. {*Suc D′* − *p*..*D′* + *p*})
proof −
  obtain *p q* **where** [*simp*]: *pq* = (*p*, *q*) **by** (*cases pq*)
  from *assms* **have** ∗: *p* < *sqrt D*
    by (*auto simp*: *red-assoc-def field-simps*)
  hence *p*: *p* ≤ *D′* **unfolding** *D′-def* **by** *linarith*
  from *assms* **have** *p* > *0* **by** (*auto intro*!: *Nat.gr0I simp*: *red-assoc-def*)

  have *q* > *sqrt D* − *p q* < *sqrt D* + *p*
    using *assms* **by** (*auto simp*: *red-assoc-def field-simps*)
  hence *q* ≥ *D′* + *1* − *p q* ≤ *D′* + *p*
    unfolding *D′-def* **by** *linarith*+
  with *p* ‹*p* > *0*› **show** *?thesis* **by** *simp*
qed

lemma *surd-to-real-cnj-eq-iff*:
  assumes *red-assoc pq red-assoc pq′*
  shows *surd-to-real-cnj pq* = *surd-to-real-cnj pq′* ⟷ *pq* = *pq′*
proof
  assume *eq*: *surd-to-real-cnj pq* = *surd-to-real-cnj pq′*
  from *assms* **have** *pos*: *snd pq* > *0 snd pq′* > *0* **by** (*auto simp*: *red-assoc-def*)
  have *snd pq* = *snd pq′*
  proof (*rule ccontr*)
    assume *snd pq* ≠ *snd pq′*
    with *eq* **have** *sqrt D* = (*real* (*fst pq′* ∗ *snd pq*) − *fst pq* ∗ *snd pq′*) / (*real* (*snd pq*) − *snd pq′*)
      using *pos* **by** (*auto simp*: *field-simps surd-to-real-cnj-def case-prod-unfold*)
    also have . . . ∈ ℚ **by** *auto*
    finally show *False* **using** *irrat-sqrt-nonsquare*[*OF nonsquare*] **by** *auto*
  qed
  moreover from *this eq pos* **have** *fst pq* = *fst pq′*

by (*auto simp*: *surd-to-real-cnj-def case-prod-unfold*)
ultimately show *pq* = *pq′* **by** (*simp add*: *prod-eq-iff*)
qed *auto*

lemma *red-assoc-sqrt-remainder-surd* [*intro*]: *red-assoc* (*sqrt-remainder-surd n*)
by (*auto simp*: *sqrt-remainder-surd-def intro*!: *red-assoc-begin*)

lemma *surd-to-real-sqrt-remainder-surd*:
surd-to-real (*sqrt-remainder-surd n*) = *cfrac-remainder* (*cfrac-of-real* (*sqrt D*)) (*Suc n*)
proof (*induction n*)
case *0*
from *nonsquare* **have** *D* > *0* **by** (*auto intro*!: *Nat.gr0I*)
with *red-assoc-begin* **show** *?case* **using** *nonsquare irrat-sqrt-nonsquare*[*OF nonsquare*]
using *Ints-subset-Rats cfrac-drop-Suc-right cfrac-remainder-def cfrac-tl-of-real sqrt-remainder-surd-def* **by** *fastforce*
next
case (*Suc n*)
have *surd-to-real* (*sqrt-remainder-surd* (*Suc n*)) =
surd-to-real (*sqrt-remainder-step* (*sqrt-remainder-surd n*))
by (*simp add*: *sqrt-remainder-surd-def*)
also have . . . = *1* / *frac* (*surd-to-real* (*sqrt-remainder-surd n*))
using *red-assoc-step*[*OF red-assoc-sqrt-remainder-surd*[*of n*]] **by** *simp*
also have *surd-to-real* (*sqrt-remainder-surd n*) =
cfrac-remainder (*cfrac-of-real* (*sqrt D*)) (*Suc n*) (**is** - = *?X*)
by (*rule Suc.IH*)
also have ⌊*cfrac-remainder* (*cfrac-of-real* (*sqrt* (*real D*))) (*Suc n*)⌋ =
cfrac-nth (*cfrac-of-real* (*sqrt* (*real D*))) (*Suc n*)
using *irrat-sqrt-nonsquare*[*OF nonsquare*] **by** (*intro floor-cfrac-remainder*) *auto*
hence *1* / *frac ?X* = *cfrac-remainder* (*cfrac-of-real* (*sqrt D*)) (*Suc* (*Suc n*))
using *irrat-sqrt-nonsquare*[*OF nonsquare*]
by (*subst cfrac-remainder-Suc*[*of Suc n*])
(*simp-all add*: *frac-def cfrac-length-of-real-irrational*)
finally show *?case* **.**
qed

lemma *sqrt-cfrac*: *sqrt-cfrac-nth n* = *cfrac-nth* (*cfrac-of-real* (*sqrt D*)) (*Suc n*)
proof −
have *cfrac-nth* (*cfrac-of-real* (*sqrt D*)) (*Suc n*) =
⌊*cfrac-remainder* (*cfrac-of-real* (*sqrt D*)) (*Suc n*)⌋
using *irrat-sqrt-nonsquare*[*OF nonsquare*] **by** (*subst floor-cfrac-remainder*) *auto*
also have *cfrac-remainder* (*cfrac-of-real* (*sqrt D*)) (*Suc n*) = *surd-to-real* (*sqrt-remainder-surd n*)
by (*rule surd-to-real-sqrt-remainder-surd* [*symmetric*])
also have *nat* ⌊*surd-to-real* (*sqrt-remainder-surd n*)⌋ = *sqrt-cfrac-nth n*
unfolding *sqrt-cfrac-nth-def* **using** *red-assoc-step*(*6*)[*OF red-assoc-sqrt-remainder-surd*[*of n*]]
by (*simp add*: *case-prod-unfold*)

finally show *?thesis*
by (*simp add*: *nat-eq-iff*)
qed

lemma *sqrt-cfrac-pos*: *sqrt-cfrac-nth k* > *0*
using *red-assoc-step*(*4*)[*OF red-assoc-sqrt-remainder-surd*[*of k*]]
by (*simp add*: *sqrt-cfrac-nth-def case-prod-unfold*)

lemma *snd-sqrt-remainder-surd-pos*: *snd* (*sqrt-remainder-surd n*) > *0*
using *red-assoc-sqrt-remainder-surd*[*of n*] **by** (*auto simp*: *red-assoc-def*)

lemma
shows *period-nonempty*: *l* > *0*
and *period-length-le-aux*: *l* ≤ *D′* ∗ (*D′* + *1*)
and *sqrt-remainder-surd-periodic*: ⋀*n*. *sqrt-remainder-surd n* = *sqrt-remainder-surd* (*n mod l*)
and *sqrt-cfrac-periodic*: ⋀*n*. *sqrt-cfrac-nth n* = *sqrt-cfrac-nth* (*n mod l*)
and *sqrt-remainder-surd-smallest-period*:
⋀*n*. *n* ∈ {*0*<..<*l*} ⟹ *sqrt-remainder-surd n* ≠ *sqrt-remainder-surd 0*
and *snd-sqrt-remainder-surd-gt-1*: ⋀*n*. *n* < *l* − *1* ⟹ *snd* (*sqrt-remainder-surd n*) > *1*
and *sqrt-cfrac-le*: ⋀*n*. *n* < *l* − *1* ⟹ *sqrt-cfrac-nth n* ≤ *D′*
and *sqrt-remainder-surd-last*: *sqrt-remainder-surd* (*l* − *1*) = (*D′*, *1*)
and *sqrt-cfrac-last*: *sqrt-cfrac-nth* (*l* − *1*) = *2* ∗ *D′*
and *sqrt-cfrac-palindrome*: ⋀*n*. *n* < *l* − *1* ⟹ *sqrt-cfrac-nth* (*l* − *n* − *2*) = *sqrt-cfrac-nth n*
and *sqrt-cfrac-smallest-period*:
⋀*l′*. *l′* > *0* ⟹ (⋀*k*. *sqrt-cfrac-nth* (*k* + *l′*) = *sqrt-cfrac-nth k*) ⟹ *l′* ≥ *l*
proof −
note [*simp*] = *sqrt-remainder-surd-def*
define *f* **where** *f* = *sqrt-remainder-surd*
have ∗[*intro*]: *red-assoc* (*f n*) **for** *n*
unfolding *f-def* **by** (*rule red-assoc-sqrt-remainder-surd*)

define *S* **where** *S* = (*SIGMA p*:{*0*<..*D′*}. {*Suc D′* − *p*..*D′* + *p*})
have [*intro*]: *finite S* **by** (*simp add*: *S-def*)
have *card S* = (∑ *p*=*1*..*D′*. *2* ∗ *p*) **unfolding** *S-def*
by (*subst card-SigmaI*) (*auto intro*!: *sum.cong*)
also have . . . = *D′* ∗ (*D′* + *1*)
by (*induction D′*) (*auto simp*: *power2-eq-square*)
finally have [*simp*]: *card S* = *D′* ∗ (*D′* + *1*) **.**

have *D′* ∗ (*D′* + *1*) + *1* = *card* {..*D′* ∗ (*D′* + *1*)} **by** *simp*
define *k1* **where**
k1 = (*LEAST k1*. *k1* ≤ *D′* ∗ (*D′* + *1*) ∧ (∃ *k2*. *k2* ≤ *D′* ∗ (*D′* + *1*) ∧ *k1* ≠ *k2* ∧ *f k1* = *f k2*))
define *k2* **where**
k2 = (*LEAST k2*. *k2* ≤ *D′* ∗ (*D′* + *1*) ∧ *k1* ≠ *k2* ∧ *f k1* = *f k2*)

have *f* ‘ {..D′ ∗ (D′ + 1)} ⊆ *S* **unfolding** *S-def*
  using *red-assoc-bounds*[*OF* ∗] **by** *blast*
hence *card* (*f* ‘ {..D′ ∗ (D′ + 1)}) ≤ *card S*
  by (*intro card-mono*) *auto*
also have *card S* = D′ ∗ (D′ + 1) **by** *simp*
also have ... < *card* {..D′ ∗ (D′ + 1)} **by** *simp*
finally have ¬*inj-on f* {..D′ ∗ (D′ + 1)}
  by (*rule pigeonhole*)
hence ∃ *k1*. *k1* ≤ D′ ∗ (D′ + 1) ∧ (∃ *k2*. *k2* ≤ D′ ∗ (D′ + 1) ∧ *k1* ≠ *k2* ∧ *f k1* = *f k2*)
  by (*auto simp*: *inj-on-def*)
from *LeastI-ex*[*OF this*, *folded k1-def*]
  have *k1* ≤ D′ ∗ (D′ + 1) ∃ *k2*≤D′ ∗ (D′ + 1). *k1* ≠ *k2* ∧ *f k1* = *f k2* **by** *auto*
moreover from *LeastI-ex*[*OF this*(*2*), *folded k2-def*]
  have *k2* ≤ D′ ∗ (D′ + 1) *k1* ≠ *k2* *f k1* = *f k2* **by** *auto*
moreover have *k1* ≤ *k2*
proof (*rule ccontr*)
  assume ¬(*k1* ≤ *k2*)
  hence *k2* ≤ D′ ∗ (D′ + 1) ∧ (∃ *k2′*. *k2′* ≤ D′ ∗ (D′ + 1) ∧ *k2* ≠ *k2′* ∧ *f k2* = *f k2′*)
    using ‹*k1* ≤ D′ ∗ (D′ + 1)› **and** ‹*k1* ≠ *k2*› **and** ‹*f k1* = *f k2*› **by** *auto*
  hence *k1* ≤ *k2* **unfolding** *k1-def* **by** (*rule Least-le*)
  with ‹¬(*k1* ≤ *k2*)› **show** *False* **by** *simp*
qed
ultimately have *k12*: *k1* < *k2* *k2* ≤ D′ ∗ (D′ + 1) *f k1* = *f k2* **by** *auto*

have [*simp*]: *k1* = *0*
proof (*cases k1*)
  case (*Suc k1′*)
  define *k2′* **where** *k2′* = *k2* − *1*
  have *Suc′*: *k2* = *Suc k2′* **using** *k12* **by** (*simp add*: *k2′-def*)
  have *nz*: *surd-to-real-cnj* (*sqrt-remainder-step* (*f k1′*)) ≠ *0*
      surd-to-real-cnj (*sqrt-remainder-step* (*f k2′*)) ≠ *0*
    using *surd-to-real-cnj-nz*[*OF* ∗[*of k2*]] *surd-to-real-cnj-nz*[*OF* ∗[*of k1*]]
    by (*simp-all add*: *f-def Suc Suc′*)

  define *a* **where** *a* = (D′ + *fst* (*f k1*)) *div snd* (*f k1*)
  define *a′* **where** *a′* = (D′ + *fst* (*f k1′*)) *div snd* (*f k1′*)
  define *a″* **where** *a″* = (D′ + *fst* (*f k2′*)) *div snd* (*f k2′*)
  have *a′* = *nat* ⌊− *1* / *surd-to-real-cnj* (*sqrt-remainder-step* (*f k1′*))⌋
    using *red-assoc-step*[*OF* ∗[*of k1′*]] **by** (*simp add*: *a′-def*)
  also have *sqrt-remainder-step* (*f k1′*) = *f k1*
    by (*simp add*: *Suc f-def*)
  also have *f k1* = *f k2* **by** *fact*
  also have *f k2* = *sqrt-remainder-step* (*f k2′*) **by** (*simp add*: *Suc′ f-def*)
  also have *nat* ⌊− *1* / *surd-to-real-cnj* (*sqrt-remainder-step* (*f k2′*))⌋ = *a″*
    using *red-assoc-step*[*OF* ∗[*of k2′*]] **by** (*simp add*: *a″-def*)
  finally have *a′-a″*: *a′* = *a″* **.**

have *surd-to-real-cnj* (*f k2'*) $\neq$ *a''*
  using *surd-to-real-cnj-irrat*[*OF* $*$[*of k2'*]] **by** *auto*
 hence *surd-to-real-cnj* (*f k2'*) = *1* / *surd-to-real-cnj* (*sqrt-remainder-step* (*f k2'*)) + *a''*
  using *red-assoc-step*(*3*)[*OF* $*$[*of k2'*], *folded a''-def*] *nz*
  by (*simp add*: *field-simps*)
also have ... = *1* / *surd-to-real-cnj* (*sqrt-remainder-step* (*f k1'*)) + *a'*
  using *k12* **by** (*simp add*: *a'-a'' k12 Suc Suc' f-def*)
also have *nz'*: *surd-to-real-cnj* (*f k1'*) $\neq$ *a'*
  using *surd-to-real-cnj-irrat*[*OF* $*$[*of k1'*]] **by** *auto*
hence *1* / *surd-to-real-cnj* (*sqrt-remainder-step* (*f k1'*)) + *a'* = *surd-to-real-cnj* (*f k1'*)
  using *red-assoc-step*(*3*)[*OF* $*$[*of k1'*], *folded a'-def*] *nz nz'*
  by (*simp add*: *field-simps*)
finally have *f k1'* = *f k2'*
  by (*subst* (*asm*) *surd-to-real-cnj-eq-iff*) *auto*
with *k12* **have** $k1' \leq D' * (D' + 1) \wedge (\exists k2 \leq D' * (D' + 1).\ k1' \neq k2 \wedge f\ k1' = f\ k2)$
  by (*auto simp*: *Suc Suc' intro*!: *exI*[*of - k2'*])
hence $k1 \leq k1'$ **unfolding** *k1-def* **by** (*rule Least-le*)
thus $k1 = 0$ **by** (*simp add*: *Suc*)
qed *auto*

have *smallest-period*: $f\ k \neq f\ 0$ **if** $k \in \{0<..<k2\}$ **for** *k*
proof
  assume $f\ k = f\ 0$
  hence $k \leq D' * (D' + 1) \wedge k1 \neq k \wedge f\ k1 = f\ k$
    using *k12 that* **by** *auto*
  hence $k2 \leq k$ **unfolding** *k2-def* **by** (*rule Least-le*)
  with *that* **show** *False* **by** *auto*
qed

have *snd-f-gt-1*: *snd* (*f k*) > *1* **if** $k < k2 - 1$ **for** *k*
proof −
  have *snd* (*f k*) $\neq$ *1*
  proof
    assume *snd* (*f k*) = *1*
    hence *f k* = (*D'*, *1*) **using** *red-assoc-denom-1*[*of fst* (*f k*)] $*$[*of k*]
      by (*cases f k*) *auto*
  hence *sqrt-remainder-step* (*f k*) = $(D', D - D'^2)$ **by** (*auto simp*: *sqrt-remainder-step-def*)
    hence *f* (*Suc k*) = *f 0* **by** (*simp add*: *f-def*)
    moreover have *f* (*Suc k*) $\neq$ *f 0*
      using *that* **by** (*intro smallest-period*) *auto*
    ultimately show *False* **by** *contradiction*
  qed
  moreover have *snd* (*f k*) > *0* **using** $*$[*of k*] **by** (*auto simp*: *red-assoc-def*)
  ultimately show *?thesis* **by** *simp*
qed

```
have sqrt-cfrac-le: sqrt-cfrac-nth k ≤ D′ if k < k2 − 1 for k
proof −
  define p and q where p = fst (f k) and q = snd (f k)
  have q ≥ 2 using snd-f-gt-1[of k] that by (auto simp: q-def)
  also have sqrt-cfrac-nth k ∗ q ≤ D′ ∗ 2
    using red-assoc-step(5)[OF ∗[of k]]
    by (simp add: sqrt-cfrac-nth-def p-def q-def case-prod-unfold f-def)
  finally show ?thesis by simp
qed

have last: f (k2 − 1) = (D′, 1)
proof −
  define p and q where p = fst (f (k2 − 1)) and q = snd (f (k2 − 1))
  have pq: f (k2 − 1) = (p, q) by (simp add: p-def q-def)
  have sqrt-remainder-step (f (k2 − 1)) = f (Suc (k2 − 1))
    by (simp add: f-def)
  also from k12 have Suc (k2 − 1) = k2 by simp
  also have f k2 = f 0
    using k12 by simp
  also have f 0 = (D′, D − D′^2) by (simp add: f-def)
  finally have eq: sqrt-remainder-step (f (k2 − 1)) = (D′, D − D′^2) .

  hence (D − D′^2) div q = D − D′^2 unfolding sqrt-remainder-step-def Let-def
pq
    by auto
  moreover have q > 0 using ∗[of k2 − 1]
    by (auto simp: red-assoc-def q-def)
  ultimately have q = 1 using D′-sqr-less-D
    by (subst (asm) div-eq-dividend-iff) auto
  hence p = D′
    using red-assoc-denom-1[of p] ∗[of k2 − 1] unfolding pq by auto
  with ‹q = 1› show f (k2 − 1) = (D′, 1) unfolding pq by simp
qed

have period: sqrt-remainder-surd n = sqrt-remainder-surd (n mod k2) for n
  unfolding sqrt-remainder-surd-def using k12
  by (metis ‹k1 = 0› f-def funpow-mod-eq funpow-0 sqrt-remainder-surd-def)
have period′: sqrt-cfrac-nth k = sqrt-cfrac-nth (k mod k2) for k
  using period[of k] by (simp add: sqrt-cfrac-nth-def)

have k2-le: l ≥ k2 if l > 0 ⋀k. sqrt-cfrac-nth (k + l) = sqrt-cfrac-nth k for l
proof (rule ccontr)
  assume ∗: ¬(l ≥ k2)
  hence sqrt-cfrac-nth (k2 − Suc l) = sqrt-cfrac-nth (k2 − 1)
    using that(2)[of k2 − Suc l] by simp
  also have . . . = 2 ∗ D′
    using last by (simp add: sqrt-cfrac-nth-def f-def)
  finally have 2 ∗ D′ = sqrt-cfrac-nth (k2 − Suc l) ..
```

also have ... $\leq D'$ **using** *k12 that* $*$
by (*intro sqrt-cfrac-le diff-less-mono2*) *auto*
finally show *False* **using** *D'-pos* **by** *simp*
qed

have $l = (LEAST\ l.\ 0 < l \wedge (\forall n.\ int\ (sqrt\text{-}cfrac\text{-}nth\ (n + l)) = int\ (sqrt\text{-}cfrac\text{-}nth\ n)))$
using *nonsquare* **unfolding** *sqrt-cfrac-def*
by (*simp add*: *l-def sqrt-nat-period-length-def sqrt-cfrac*)
hence *l-altdef*: $l = (LEAST\ l.\ 0 < l \wedge (\forall n.\ sqrt\text{-}cfrac\text{-}nth\ (n + l) = sqrt\text{-}cfrac\text{-}nth\ n))$
by *simp*

have [*simp*]: $D \neq 0$ **using** *nonsquare* **by** (*auto intro*!: *Nat.gr0I*)
have $\exists l.\ l > 0 \wedge (\forall k.\ sqrt\text{-}cfrac\text{-}nth\ (k + l) = sqrt\text{-}cfrac\text{-}nth\ k)$
proof (*rule exI*, *safe*)
fix *k* **show** $sqrt\text{-}cfrac\text{-}nth\ (k + k2) = sqrt\text{-}cfrac\text{-}nth\ k$
using *period'*[*of k*] *period'*[*of k* + *k2*] *k12* **by** *simp*
qed (*insert k12*, *auto*)
from *LeastI-ex*[*OF this*, *folded l-altdef*]
have *l*: $l > 0$ $\bigwedge k.\ sqrt\text{-}cfrac\text{-}nth\ (k + l) = sqrt\text{-}cfrac\text{-}nth\ k$
by (*simp-all add*: *sqrt-cfrac*)

have $l \leq k2$ **unfolding** *l-altdef*
by (*rule Least-le*) (*subst* (*1 2*) *period'*, *insert k12*, *auto*)
moreover have $k2 \leq l$ **using** *k2-le l* **by** *blast*
ultimately have [*simp*]: $l = k2$ **by** *auto*

define x' **where** $x' = (\lambda k.\ -1\ /\ surd\text{-}to\text{-}real\text{-}cnj\ (f\ k))$
{
fix *k* :: *nat*
have *nz*: $surd\text{-}to\text{-}real\text{-}cnj\ (f\ k) \neq 0$ $surd\text{-}to\text{-}real\text{-}cnj\ (f\ (Suc\ k)) \neq 0$
using *surd-to-real-cnj-nz*[*OF* $*$, *of k*] *surd-to-real-cnj-nz*[*OF* $*$, *of Suc k*]
by (*simp-all add*: *f-def*)

have $surd\text{-}to\text{-}real\text{-}cnj\ (f\ k) \neq sqrt\text{-}cfrac\text{-}nth\ k$
using *surd-to-real-cnj-irrat*[*OF* $*$[*of k*]] **by** *auto*
hence $x'\ (Suc\ k) = sqrt\text{-}cfrac\text{-}nth\ k + 1\ /\ x'\ k$
using *red-assoc-step*(*3*)[*OF* $*$[*of k*]] *nz*
by (*simp add*: *field-simps sqrt-cfrac-nth-def case-prod-unfold f-def x'-def*)
} **note** *x'-Suc* = *this*

have *x'-nz*: $x'\ k \neq 0$ **for** *k*
using *surd-to-real-cnj-nz*[*OF* $*$[*of k*]] **by** (*auto simp*: *x'-def*)
have *x'-0*: $x'\ 0 = real\ D' + sqrt\ D$
using *red-assoc-begin* **by** (*simp add*: *x'-def f-def*)

define c' **where** $c' = cfrac\ (\lambda n.\ sqrt\text{-}cfrac\text{-}nth\ (l - Suc\ n))$
define c'' **where** $c'' = cfrac\ (\lambda n.\ if\ n = 0\ then\ 2 * D'\ else\ sqrt\text{-}cfrac\text{-}nth\ (n -$

1))
have *nth-c′* [*simp*]: *cfrac-nth c′ n* = *sqrt-cfrac-nth* (*l* − *Suc n*) **for** *n*
unfolding *c′-def* **by** (*subst cfrac-nth-cfrac*) (*auto simp*: *is-cfrac-def intro*!: *sqrt-cfrac-pos*)
have *nth-c″* [*simp*]: *cfrac-nth c″ n* = (*if n* = *0 then 2* ∗ *D′ else sqrt-cfrac-nth* (*n* − *1*)) **for** *n*
unfolding *c″-def* **by** (*subst cfrac-nth-cfrac*) (*auto simp*: *is-cfrac-def intro*!: *sqrt-cfrac-pos*)

have *conv′ c′ n* (*x′* (*l* − *n*)) = *x′ l* **if** *n* ≤ *l* **for** *n*
using *that*
proof (*induction n*)
case (*Suc n*)
have *x′ l* = *conv′ c′ n* (*x′* (*l* − *n*))
using *Suc.prems* **by** (*intro Suc.IH* [*symmetric*]) *auto*
also have *l* − *n* = *Suc* (*l* − *Suc n*)
using *Suc.prems* **by** *simp*
also have *x′* . . . = *cfrac-nth c′ n* + *1* / *x′* (*l* − *Suc n*)
by (*subst x′-Suc*) *simp*
also have *conv′ c′ n* . . . = *conv′ c′* (*Suc n*) (*x′* (*l* − *Suc n*))
by (*simp add*: *conv′-Suc-right*)
finally show *?case* **..**
qed *simp-all*
from *this*[*of l*] **have** *conv′-x′-0*: *conv′ c′ l* (*x′ 0*) = *x′ 0*
using *k12* **by** (*simp add*: *x′-def*)

have *cfrac-nth* (*cfrac-of-real* (*x′ 0*)) *n* = *cfrac-nth c″ n* **for** *n*
proof (*cases n*)
case *0*
thus *?thesis* **by** (*simp add*: *x′-0 D′-def*)
next
case (*Suc n′*)
have *sqrt D* ∉ ℤ
using *red-assoc-begin*(*1*) *red-assoc-begin*(*2*) **by** *auto*
hence *cfrac-nth* (*cfrac-of-real* (*real D′* + *sqrt* (*real D*))) (*Suc n′*) =
cfrac-nth (*cfrac-of-real* (*sqrt* (*real D*))) (*Suc n′*)
by (*simp add*: *cfrac-tl-of-real frac-add-of-nat Ints-add-left-cancel flip*: *cfrac-nth-tl*)
thus *?thesis* **using** *x′-nz*[*of 0*]
by (*simp add*: *x′-0 sqrt-cfrac Suc*)
qed

show *sqrt-cfrac-nth* (*l* − *n* − *2*) = *sqrt-cfrac-nth n* **if** *n* < *l* − *1* **for** *n*
proof −
have *D* > *1* **using** *nonsquare* **by** (*cases D*) (*auto intro*!: *Nat.gr0I*)
hence *D′* + *sqrt D* > *0* + *1* **using** *D′-pos* **by** (*intro add-strict-mono*) *auto*
hence *x′ 0* > *1* **by** (*auto simp*: *x′-0*)
hence *cfrac-nth c′* (*Suc n*) = *cfrac-nth* (*cfrac-of-real* (*conv′ c′ l* (*x′ 0*))) (*Suc n*)
using ‹*n* < *l* − *1*› **using** *cfrac-of-real-conv′* **by** *auto*

```
    also have ... = cfrac-nth (cfrac-of-real (x′ 0)) (Suc n)
      by (subst conv′-x′-0) auto
    also have ... = cfrac-nth c′′ (Suc n) by fact
    finally show sqrt-cfrac-nth (l − n − 2) = sqrt-cfrac-nth n
      by simp
  qed

  show l > 0 l ≤ D′ ∗ (D′ + 1) using k12 by simp-all
  show sqrt-remainder-surd n = sqrt-remainder-surd (n mod l)
       sqrt-cfrac-nth n = sqrt-cfrac-nth (n mod l) for n
    using period[of n] period′[of n] by simp-all
  show sqrt-remainder-surd n ≠ sqrt-remainder-surd 0 if n ∈ {0<..<l} for n
    using smallest-period[of n] that by (auto simp: f-def)
  show snd (sqrt-remainder-surd n) > 1 if n < l − 1 for n
    using that snd-f-gt-1[of n] by (simp add: f-def)
  show f (l − 1) = (D′, 1) and sqrt-cfrac-nth (l − 1) = 2 ∗ D′
    using last by (simp-all add: sqrt-cfrac-nth-def f-def)
  show sqrt-cfrac-nth k ≤ D′ if k < l − 1 for k
    using sqrt-cfrac-le[of k] that by simp
  show l′ ≥ l if l′ > 0 ⋀k. sqrt-cfrac-nth (k + l′) = sqrt-cfrac-nth k for l′
    using k2-le[of l′] that by auto
qed

theorem cfrac-sqrt-periodic:
  cfrac-nth (cfrac-of-real (sqrt D)) (Suc n) =
   cfrac-nth (cfrac-of-real (sqrt D)) (Suc (n mod l))
  using sqrt-cfrac-periodic[of n] by (metis sqrt-cfrac)

theorem cfrac-sqrt-le: n ∈ {0<..<l} ⟹ cfrac-nth (cfrac-of-real (sqrt D)) n ≤ D′
  using sqrt-cfrac-le[of n − 1]
 by (metis Suc-less-eq Suc-pred add.right-neutral greaterThanLessThan-iff of-nat-mono
          period-nonempty plus-1-eq-Suc sqrt-cfrac)

theorem cfrac-sqrt-last: cfrac-nth (cfrac-of-real (sqrt D)) l = 2 ∗ D′
 using sqrt-cfrac-last by (metis One-nat-def Suc-pred period-nonempty sqrt-cfrac)

theorem cfrac-sqrt-palindrome:
  assumes n ∈ {0<..<l}
  shows   cfrac-nth (cfrac-of-real (sqrt D)) (l − n) = cfrac-nth (cfrac-of-real (sqrt
D)) n
proof −
  have cfrac-nth (cfrac-of-real (sqrt D)) (l − n) = sqrt-cfrac-nth (l − n − 1)
    using assms by (subst sqrt-cfrac) (auto simp: Suc-diff-Suc)
  also have ... = sqrt-cfrac-nth (n − 1)
    using assms by (subst sqrt-cfrac-palindrome [symmetric]) auto
  also have ... = cfrac-nth (cfrac-of-real (sqrt D)) n
    using assms by (subst sqrt-cfrac) auto
  finally show ?thesis .
qed
```

lemma *sqrt-cfrac-info-palindrome*:
assumes *sqrt-cfrac-info D* = (*a*, *b*, *cs*)
shows *rev* (*butlast cs*) = *butlast cs*
proof (*rule List.nth-equalityI*; *safe?*)
fix *i* **assume** *i* < *length* (*rev* (*butlast cs*))
with *period-nonempty* **have** *Suc i* < *length cs* **by** *simp*
thus *rev* (*butlast cs*) ! *i* = *butlast cs* ! *i*
using *assms cfrac-sqrt-palindrome*[*of Suc i*] *period-nonempty* **unfolding** *l-def*
by (*auto simp*: *sqrt-cfrac-info-def rev-nth algebra-simps Suc-diff-Suc simp del*: *cfrac.simps*)
qed *simp-all*

lemma *sqrt-cfrac-info-last*:
assumes *sqrt-cfrac-info D* = (*a*, *b*, *cs*)
shows *last cs* = *2* ∗ *Discrete.sqrt D*
proof −
from *assms* **show** *?thesis* **using** *period-nonempty cfrac-sqrt-last*
by (*auto simp*: *sqrt-cfrac-info-def last-map l-def D′-def Discrete-sqrt-altdef*)
qed

The following lemmas allow us to compute the period of the expansion of the square root:

lemma *while-option-sqrt-cfrac*:
defines *step′* ≡ (λ(*as*, *pq*). ((*D′* + *fst pq*) *div snd pq* # *as*, *sqrt-remainder-step pq*))
defines *b* ≡ (λ(-, *pq*). *snd pq* ≠ *1*)
defines *initial* ≡ ([] :: *nat list*, (*D′*, *D* − *D′*²))
shows *while-option b step′ initial* =
Some (*rev* (*map sqrt-cfrac-nth* [*0*..<*l* −*1*]), (*D′*, *1*))
proof −
define *P* **where**
P = (λ(*as*, *pq*). *let n* = *length as*
in n < *l* ∧ *pq* = *sqrt-remainder-surd n* ∧ *as* = *rev* (*map sqrt-cfrac-nth* [*0*..<*n*]))
define *μ* :: *nat list* × (*nat* × *nat*) ⇒ *nat* **where** *μ* = (λ(*as*, -). *l* − *length as*)
have [*simp*]: *P initial* **using** *period-nonempty*
by (*auto simp*: *initial-def P-def sqrt-remainder-surd-def*)
have *step′*: *P* (*step′ s*) ∧ *Suc* (*length* (*fst s*)) < *l* **if** *P s b s* **for** *s*
proof (*cases s*)
case (*fields as p q*)
define *n* **where** *n* = *length as*
from *that fields sqrt-remainder-surd-last* **have** *Suc n* ≤ *l*
by (*auto simp*: *b-def P-def Let-def n-def* [*symmetric*])
moreover from *that fields sqrt-remainder-surd-last* **have** *Suc n* ≠ *l*
by (*auto simp*: *b-def P-def Let-def n-def* [*symmetric*])
ultimately have *Suc n* < *l* **by** *auto*
with *that fields sqrt-remainder-surd-last* **show** *P* (*step′ s*) ∧ *Suc* (*length* (*fst s*)) < *l*

```
    by (simp add: b-def P-def Let-def n-def step'-def sqrt-cfrac-nth-def
              sqrt-remainder-surd-def case-prod-unfold)
  qed
  have [simp]: length (fst (step' s)) = Suc (length (fst s)) for s
    by (simp add: step'-def case-prod-unfold)

  have ∃x. while-option b step' initial = Some x
  proof (rule measure-while-option-Some)
    fix s assume *: P s b s
    from step'[OF *] show P (step' s) ∧ μ (step' s) < μ s
      by (auto simp: b-def μ-def case-prod-unfold intro!: diff-less-mono2)
  qed auto
  then obtain x where x: while-option b step' initial = Some x ..
  have P x by (rule while-option-rule[OF - x]) (insert step', auto)
  have ¬b x using while-option-stop[OF x] by auto

  obtain as p q where [simp]: x = (as, (p, q)) by (cases x)
  define n where n = length as
  have [simp]: q = 1 using ‹¬b x› by (auto simp: b-def)
  have [simp]: p = D' using ‹P x›
    using red-assoc-denom-1[of p] by (auto simp: P-def Let-def)
  have n < l sqrt-remainder-surd (length as) = (D', Suc 0)
      and as: as = rev (map sqrt-cfrac-nth [0..<n]) using ‹P x›
    by (auto simp: P-def Let-def n-def)
  hence ¬(n < l − 1)
    using snd-sqrt-remainder-surd-gt-1[of n] by (intro notI) auto
  with ‹n < l› have [simp]: n = l − 1 by auto
  show ?thesis by (simp add: as x)
qed

lemma while-option-sqrt-cfrac-info:
  defines step' ≡ (λ(as, pq). ((D' + fst pq) div snd pq # as, sqrt-remainder-step
pq))
  defines b ≡ (λ(-, pq). snd pq ≠ 1)
  defines initial ≡ ([], (D', D − D'²))
  shows sqrt-cfrac-info D =
          (case while-option b step' initial of
            Some (as, -) ⇒ (Suc (length as), D', rev ((2 ∗ D') # as)))
proof −
  have nat (cfrac-nth (cfrac-of-real (sqrt (real D))) (Suc k)) = sqrt-cfrac-nth k for
k
    by (metis nat-int sqrt-cfrac)
  thus ?thesis unfolding assms while-option-sqrt-cfrac
    using period-nonempty sqrt-cfrac-last
    by (cases l) (auto simp: sqrt-cfrac-info-def D'-def l-def Discrete-sqrt-altdef)
qed

end
end
```

lemma *sqrt-nat-period-length-le*: *sqrt-nat-period-length D* ≤ *nat* ⌊*sqrt D*⌋ ∗ (*nat* ⌊*sqrt D*⌋ + *1*)
  by (*cases is-square D*) (*use period-length-le-aux*[*of D*] **in** *auto*)

lemma *sqrt-nat-period-length-0-iff* [*simp*]:
  sqrt-nat-period-length D = *0* ⟷ *is-square D*
  using *period-nonempty*[*of D*] **by** (*cases is-square D*) *auto*

lemma *sqrt-nat-period-length-pos-iff* [*simp*]:
  sqrt-nat-period-length D > *0* ⟷ ¬*is-square D*
  using *period-nonempty*[*of D*] **by** (*cases is-square D*) *auto*

lemma *sqrt-cfrac-info-code* [*code*]:
  sqrt-cfrac-info D =
    (*let D′* = *Discrete.sqrt D*
     *in if D′*² = *D then* (*0*, *D′*, [])
        else
          case while-option
                (λ(-, *pq*). *snd pq* ≠ *1*)
                (λ(*as*, (*p*, *q*)). *let X* = (*p* + *D′*) *div q*; *p′* = *X* ∗ *q* − *p*
                                in (*X* # *as*, *p′*, (*D* − *p′*²) *div q*))
                ([], *D′*, *D* − *D′*²)
          of Some (*as*, -) ⇒ (*Suc* (*length as*), *D′*, *rev* ((*2* ∗ *D′*) # *as*)))
proof −
  define *D′* **where** *D′* = *Discrete.sqrt D*
  show *?thesis*
  proof (*cases is-square D*)
    case *True*
    hence *D′* ^ *2* = *D* **by** (*auto simp*: *D′-def elim*!: *is-nth-powerE*)
    thus *?thesis* **using** *True*
      by (*simp add*: *D′-def Let-def sqrt-cfrac-info-def sqrt-nat-period-length-def*)
  next
    case *False*
    hence *D′* ^ *2* ≠ *D* **by** (*subst eq-commute*) *auto*
    thus *?thesis* **using** *while-option-sqrt-cfrac-info*[*OF False*]
      by (*simp add*: *sqrt-cfrac-info-def D′-def Let-def*
                case-prod-unfold Discrete-sqrt-altdef add-ac sqrt-remainder-step-def)
  qed
qed

lemma *sqrt-nat-period-length-code* [*code*]:
  sqrt-nat-period-length D = *fst* (*sqrt-cfrac-info D*)
  by (*simp add*: *sqrt-cfrac-info-def*)

For efficiency reasons, it is often better to use an array instead of a list:

definition *sqrt-cfrac-info-array* **where**
  sqrt-cfrac-info-array D = (*case sqrt-cfrac-info D of* (*a*, *b*, *c*) ⇒ (*a*, *b*, *IArray c*))

```
lemma fst-sqrt-cfrac-info-array [simp]: fst (sqrt-cfrac-info-array D) = sqrt-nat-period-length
D
  by (simp add: sqrt-cfrac-info-array-def sqrt-cfrac-info-def)

lemma snd-sqrt-cfrac-info-array [simp]: fst (snd (sqrt-cfrac-info-array D)) = Dis-
crete.sqrt D
  by (simp add: sqrt-cfrac-info-array-def sqrt-cfrac-info-def)


definition cfrac-sqrt-nth :: nat × nat × nat iarray ⇒ nat ⇒ nat where
  cfrac-sqrt-nth info n =
     (case info of (l, a0, as) ⇒ if n = 0 then a0 else as !! ((n − 1) mod l))

lemma cfrac-sqrt-nth:
  assumes ¬is-square D
  shows   cfrac-nth (cfrac-of-real (sqrt D)) n =
            int (cfrac-sqrt-nth (sqrt-cfrac-info-array D) n) (is ?lhs = ?rhs)
proof (cases n)
  case (Suc n′)
  define l where l = sqrt-nat-period-length D
  from period-nonempty[OF assms] have l > 0 by (simp add: l-def)
  have cfrac-nth (cfrac-of-real (sqrt D)) (Suc n′) =
        cfrac-nth (cfrac-of-real (sqrt D)) (Suc (n′ mod l)) unfolding l-def
    using cfrac-sqrt-periodic[OF assms, of n′] by simp
  also have . . . = map (λn. nat (cfrac-nth (cfrac-of-real (sqrt D)) (Suc n))) [0..<l]
! (n′ mod l)
    using ‹l > 0› by (subst nth-map) auto
  finally show ?thesis using Suc
   by (simp add: sqrt-cfrac-info-array-def sqrt-cfrac-info-def l-def cfrac-sqrt-nth-def)
qed (simp-all add: sqrt-cfrac-info-def sqrt-cfrac-info-array-def
                   Discrete-sqrt-altdef cfrac-sqrt-nth-def)

lemma sqrt-cfrac-code [code]:
  sqrt-cfrac D =
     (let info = sqrt-cfrac-info-array D;
          (l, a0, -) = info
      in  if l = 0 then cfrac-of-int (int a0) else cfrac (cfrac-sqrt-nth info))
proof (cases is-square D)
  case True
  hence sqrt (real D) = of-int (Discrete.sqrt D)
    by (auto elim!: is-nth-powerE)
  thus ?thesis using True
   by (auto simp: Let-def sqrt-cfrac-info-array-def sqrt-cfrac-info-def sqrt-cfrac-def)
next
  case False
  have cfrac-sqrt-nth (sqrt-cfrac-info-array D) n > 0 if n > 0 for n
  proof −
    have int (cfrac-sqrt-nth (sqrt-cfrac-info-array D) n) > 0
      using False that by (subst cfrac-sqrt-nth [symmetric]) auto
```

thus *?thesis* **by** *simp*
qed
moreover have *sqrt D* $\notin \mathbb{Q}$
using *False irrat-sqrt-nonsquare* **by** *blast*
ultimately have *sqrt-cfrac D* = *cfrac* (*cfrac-sqrt-nth* (*sqrt-cfrac-info-array D*))
using *cfrac-sqrt-nth*[*OF False*]
by (*intro cfrac-eqI*) (*auto simp*: *sqrt-cfrac-def is-cfrac-def*)
thus *?thesis*
using *False* **by** (*simp add*: *Let-def sqrt-cfrac-info-array-def sqrt-cfrac-info-def*)
qed

As a test, we determine the continued fraction expansion of $\sqrt{129}$, which is $[11; \overline{2, 1, 3, 1, 6, 1, 3, 1, 2, 22}]$ (a period length of 10):

value *let info* = *sqrt-cfrac-info-array 129 in info*
value *sqrt-nat-period-length 129*

We can also compute convergents of $\sqrt{129}$ and observe that the difference between the square of the convergents and 129 vanishes quickly::

value *map* (*conv* (*sqrt-cfrac 129*)) [*0..<10*]
value *map* (λ*n*. |*conv* (*sqrt-cfrac 129*) *n* ^ *2* − *129*|) [*0..<20*]

end

5 Lifting solutions of Pell's Equation

theory *Pell-Lifting*
imports *Pell.Pell Pell.Pell-Algorithm*
begin

5.1 Auxiliary material

lemma (**in** *pell*) *snth-pell-solutions*: *snth* (*pell-solutions D*) *n* = *nth-solution n*
by (*simp add*: *pell-solutions-def Let-def find-fund-sol-correct nonsquare-D nth-solution-def pell-power-def pell-mul-commutes*[*of - fund-sol*])

definition *square-squarefree-part-nat* :: *nat* ⇒ *nat* × *nat* **where**
square-squarefree-part-nat n = (*square-part n*, *squarefree-part n*)

lemma *prime-factorization-squarefree-part*:
assumes *x* ≠ *0*
shows *prime-factorization* (*squarefree-part x*) =
mset-set {*p* ∈ *prime-factors x*. *odd* (*multiplicity p x*)} (**is** *?lhs* = *?rhs*)
proof (*rule multiset-eqI*)
fix *p* **show** *count ?lhs p* = *count ?rhs p*
proof (*cases prime p*)
case *False*
thus *?thesis* **by** (*auto simp*: *count-prime-factorization*)
next

case *True*
have *finite* (*prime-factors x*) **by** *simp*
hence *finite* {*p. p dvd x* ∧ *prime p*} **using** *assms*
by (*subst* (*asm*) *prime-factors-dvd*) (*auto simp*: *conj-commute*)
hence *finite* {*p. p dvd x* ∧ *prime p* ∧ *odd* (*multiplicity p x*)}
by (*rule finite-subset* [*rotated*]) *auto*
moreover have *odd* (*n* :: *nat*) ⟷ *n mod 2* = *Suc 0* **for** *n* **by** *presburger*
ultimately show *?thesis* **using** *assms*
by (*cases p dvd x*; *cases even* (*multiplicity p x*))
(*auto simp*: *count-prime-factorization prime-multiplicity-squarefree-part*
in-prime-factors-iff not-dvd-imp-multiplicity-0)
qed
qed

lemma *squarefree-part-nat*:
squarefree-part (*n* :: *nat*) = (∏{*p* ∈ *prime-factors n. odd* (*multiplicity p n*)})
proof (*cases n* = *0*)
case *False*
hence (∏{*p* ∈ *prime-factors n. odd* (*multiplicity p n*)}) =
prod-mset (*prime-factorization* (*squarefree-part n*))
by (*subst prime-factorization-squarefree-part*) (*auto simp*: *prod-unfold-prod-mset*)
also have . . . = *squarefree-part n*
by (*intro prod-mset-prime-factorization-nat Nat.gr0I*) *auto*
finally show *?thesis* **..**
qed *auto*

lemma *prime-factorization-square-part*:
assumes *x* ≠ *0*
shows *prime-factorization* (*square-part x*) =
(∑ *p* ∈ *prime-factors x. replicate-mset* (*multiplicity p x div 2*) *p*) (**is** *?lhs* = *?rhs*)
proof (*rule multiset-eqI*)
fix *p* **show** *count ?lhs p* = *count ?rhs p*
proof (*cases prime p* ∧ *p dvd x*)
case *False*
thus *?thesis* **by** (*auto simp*: *count-prime-factorization count-sum*
prime-multiplicity-square-part not-dvd-imp-multiplicity-0)
next
case *True*
thus *?thesis* **using** *assms*
by (*cases p dvd x*)
(*auto simp*: *count-prime-factorization prime-multiplicity-squarefree-part*
in-prime-factors-iff count-sum prime-multiplicity-square-part)
qed
qed

lemma *prod-mset-sum*: *prod-mset* (*sum f A*) = (∏*x*∈*A*. *prod-mset* (*f x*))
by (*induction A rule*: *infinite-finite-induct*) *auto*

```
lemma square-part-nat:
  assumes n > 0
  shows   square-part (n :: nat) = (∏ p ∈ prime-factors n. p ^ (multiplicity p n
div 2))
proof −
  have (∏ p ∈ prime-factors n. p ^ (multiplicity p n div 2)) =
          prod-mset (prime-factorization (square-part n)) using assms
     by (subst prime-factorization-square-part) (auto simp: prod-unfold-prod-mset
prod-mset-sum)
  also have ... = square-part n using assms
    by (intro prod-mset-prime-factorization-nat Nat.gr0I) auto
  finally show ?thesis ..
qed

lemma square-squarefree-part-nat-code [code]:
  square-squarefree-part-nat n = (if n = 0 then (0, 1)
     else let ps = prime-factorization n
          in  ((∏ p∈set-mset ps. p ^ (count ps p div 2)),
               ∏ (Set.filter (λp. odd (count ps p)) (set-mset ps))))
  by (cases n = 0)
   (auto simp: Let-def square-squarefree-part-nat-def squarefree-part-nat Set.filter-def

                 count-prime-factorization square-part-nat intro!: prod.cong)

lemma square-part-nat-code [code-unfold]:
  square-part (n :: nat) = (if n = 0 then 0
      else let ps = prime-factorization n in  (∏ p∈set-mset ps. p ^ (count ps p div
2)))
  using square-squarefree-part-nat-code[of n]
  by (simp add: square-squarefree-part-nat-def Let-def split: if-splits)

lemma squarefree-part-nat-code [code-unfold]:
  squarefree-part (n :: nat) = (if n = 0 then 1
       else let ps = prime-factorization n in (∏ (Set.filter (λp. odd (count ps p))
(set-mset ps))))
  using square-squarefree-part-nat-code[of n]
  by (simp add: square-squarefree-part-nat-def Let-def split: if-splits)

lemma is-nth-power-mult-nth-powerD:
  assumes is-nth-power n (a ∗ b ^ n) b > 0 n > 0
  shows   is-nth-power n (a::nat)
proof −
  from assms obtain k where k: k ^ n = a ∗ b ^ n
    by (auto elim: is-nth-powerE)
  with assms(2,3) have b dvd k
    by (metis dvd-triv-right pow-divides-pow-iff)
  then obtain l where k = b ∗ l
    by auto
  with k have a = l ^ n using assms(2)
```

by (*simp add*: *power-mult-distrib*)
thus *?thesis* **by** *auto*
qed

lemma (**in** *pell*) *fund-sol-eq-fstI*:
assumes *nontriv-solution* (x, y)
assumes $\bigwedge x'\ y'.$ *nontriv-solution* $(x', y') \Longrightarrow x \leq x'$
shows *fund-sol* $= (x, y)$
proof −
have $x =$ *fst fund-sol*
using *fund-sol-is-nontriv-solution assms*(*1*) *fund-sol-minimal''*[*of* (x, y)]
by (*auto intro*!: *antisym assms*(*2*)[*of fst fund-sol snd fund-sol*])
moreover from *this* **have** $y =$ *snd fund-sol*
using *assms*(*1*) *solutions-linorder-strict*[*of x y fst fund-sol snd fund-sol*]
fund-sol-is-nontriv-solution
by (*auto simp*: *nontriv-solution-imp-solution prod-eq-iff*)
ultimately show *?thesis* **by** *simp*
qed

lemma (**in** *pell*) *fund-sol-eqI-fst'*:
assumes *nontriv-solution xy*
assumes $\bigwedge x'\ y'.$ *nontriv-solution* $(x', y') \Longrightarrow$ *fst xy* $\leq x'$
shows *fund-sol* $=$ *xy*
using *fund-sol-eq-fstI*[*of fst xy snd xy*] *assms* **by** *simp*

lemma (**in** *pell*) *fund-sol-eq-sndI*:
assumes *nontriv-solution* (x, y)
assumes $\bigwedge x'\ y'.$ *nontriv-solution* $(x', y') \Longrightarrow y \leq y'$
shows *fund-sol* $= (x, y)$
proof −
have $y =$ *snd fund-sol*
using *fund-sol-is-nontriv-solution assms*(*1*) *fund-sol-minimal''*[*of* (x, y)]
by (*auto intro*!: *antisym assms*(*2*)[*of fst fund-sol snd fund-sol*])
moreover from *this* **have** $x =$ *fst fund-sol*
using *assms*(*1*) *solutions-linorder-strict*[*of x y fst fund-sol snd fund-sol*]
fund-sol-is-nontriv-solution
by (*auto simp*: *nontriv-solution-imp-solution prod-eq-iff*)
ultimately show *?thesis* **by** *simp*
qed

lemma (**in** *pell*) *fund-sol-eqI-snd'*:
assumes *nontriv-solution xy*
assumes $\bigwedge x'\ y'.$ *nontriv-solution* $(x', y') \Longrightarrow$ *snd xy* $\leq y'$
shows *fund-sol* $=$ *xy*
using *fund-sol-eq-sndI*[*of fst xy snd xy*] *assms* **by** *simp*

5.2 The lifting mechanism

The solutions of Pell's equations for parameters D and $a^2\ D$ stand in correspondence to one another: every solution $(x,\ y)$ for parameter D can be lowered to a solution $(x,\ ay)$ for $a^2\ D$, and every solution of the form $(x,\ ay)$ for parameter $a^2\ D$ can be lifted to a solution $(x,\ y)$ for parameter D.

locale *pell-lift* = *pell* +
  fixes *a* D' :: *nat*
  assumes *nz*: $a > 0$
  defines $D' \equiv D * a^2$
begin

lemma *nonsquare-D'*: ¬*is-square* D'
  using *nonsquare-D is-nth-power-mult-nth-powerD*[*of 2 D a*] *nz* **by** (*auto simp*: *D'-def*)

definition *lift-solution* :: *nat* × *nat* ⇒ *nat* × *nat* **where**
  lift-solution = (λ(*x*, *y*). (*x*, *y div a*))

definition *lower-solution* :: *nat* × *nat* ⇒ *nat* × *nat* **where**
  lower-solution = (λ(*x*, *y*). (*x*, *y* ∗ *a*))

definition *liftable-solution* :: *nat* × *nat* ⇒ *bool* **where**
  liftable-solution = (λ(*x*, *y*). *a dvd y*)

sublocale *lift*: *pell D'*
  by *unfold-locales* (*fact nonsquare-D'*)

lemma *lift-solution-iff*: *lift.solution xy* ⟷ *solution* (*lower-solution xy*)
  unfolding *solution-def lift.solution-def*
  by (*auto simp*: *lower-solution-def D'-def case-prod-unfold power-mult-distrib*)

lemma *lift-solution*:
  assumes *solution xy liftable-solution xy*
  shows *lift.solution* (*lift-solution xy*)
  using *assms* **unfolding** *solution-def lift.solution-def*
  by (*auto simp*: *liftable-solution-def lift-solution-def D'-def case-prod-unfold power-mult-distrib elim*!: *dvdE*)

In particular, the fundamental solution for $a^2\ D$ is the smallest liftable solution for D:

lemma *lift-fund-sol*:
  assumes ⋀*n*. $0 < n \Longrightarrow n < m \Longrightarrow$ ¬*liftable-solution* (*nth-solution n*)
  assumes *liftable-solution* (*nth-solution m*) $m > 0$
  shows *lift.fund-sol* = *lift-solution* (*nth-solution m*)
proof (*rule lift.fund-sol-eqI-fst'*)
  from *assms* **have** *nontriv-solution* (*nth-solution m*)
    by (*intro nth-solution-sound'*)

hence *lift-solution* (*nth-solution m*) ≠ (*1*, *0*) **using** *nz assms*(*2*)
by (*auto simp*: *lift-solution-def case-prod-unfold nontriv-solution-def liftable-solution-def*)
with *assms* **show** *lift.nontriv-solution* (*lift-solution* (*nth-solution m*))
by (*auto simp*: *lift.nontriv-solution-altdef intro*: *lift-solution*)
next
fix *x′ y′* :: *nat*
assume ∗: *lift.nontriv-solution* (*x′*, *y′*)
hence *nz′*: *x′* ≠ *1* **using** *nonsquare-D′*
by (*auto simp*: *lift.nontriv-solution-altdef lift.solution-def*)
from ∗ **have** *solution* (*lower-solution* (*x′*, *y′*))
by (*simp add*: *lift-solution-iff lift.nontriv-solution-altdef*)
hence *lower-solution* (*x′*, *y′*) ∈ *range nth-solution* **by** (*rule nth-solution-complete*)
then obtain *n* **where** *n*: *nth-solution n* = *lower-solution* (*x′*, *y′*) **by** *auto*
with *nz′* **have** *n* > *0* **by** (*auto intro*!: *Nat.gr0I simp*: *nth-solution-def lower-solution-def*)
with *n* **have** *liftable-solution* (*nth-solution n*)
by (*auto simp*: *liftable-solution-def lower-solution-def*)
with ‹*n* > *0*› **and** *assms*(*1*)[*of n*] **have** *n* ≥ *m* **by** (*cases n* ≥ *m*) *auto*
hence *fst* (*nth-solution m*) ≤ *fst* (*nth-solution n*)
using *strict-mono-less-eq*[*OF strict-mono-nth-solution*(*1*)] **by** *simp*
thus *fst* (*lift-solution* (*nth-solution m*)) ≤ *x′*
by (*simp add*: *lift-solution-def lower-solution-def n case-prod-unfold*)
qed

end

5.3 Accelerated computation of the fundamental solution for non-squarefree inputs

Solving Pell's equation for some D of the form $a^2\ D'$ can be done by solving it for D' and then lifting the solution. Thus, if D is not squarefree, we can compute its squarefree decomposition $a^2\ D'$ with D' squarefree and thus speed up the computation (since D' is smaller than D).

The squarefree decomposition can only be computed (according to current knowledge in mathematics) through the prime decomposition. However, given how big the solutions are for even moderate values of D, it is usually worth doing it if D is not squarefree.

lemma *squarefree-part-of-square* [*simp*]:
assumes *is-square* (*x* :: ′*a* :: {*factorial-semiring*, *normalization-semidom-multiplicative*})
assumes *x* ≠ *0*
shows *squarefree-part x* = *unit-factor x*
proof −
from *assms* **obtain** *y* **where** [*simp*]: *x* = *y* ^ *2*
by (*auto simp*: *is-nth-power-def*)
have *unit-factor x* ∗ *normalize x* = *squarefree-part x* ∗ *square-part x* ^ *2*
by (*subst squarefree-decompose* [*symmetric*]) *auto*
also have . . . = *squarefree-part x* ∗ *normalize x*
by (*simp add*: *square-part-even-power normalize-power*)

```
  finally show ?thesis using assms
    by (subst (asm) mult-cancel-right) auto
qed

lemma squarefree-part-1-imp-square:
  assumes squarefree-part x = 1
  shows   is-square x
proof –
  have is-square (square-part x ^ 2)
    by auto
  also have square-part x ^ 2 = squarefree-part x ∗ square-part x ^ 2
    using assms by simp
  also have . . . = x
    by (rule squarefree-decompose [symmetric])
  finally show ?thesis .
qed


definition find-fund-sol-fast where
  find-fund-sol-fast D =
     (let (a, D′) = square-squarefree-part-nat D
      in
        if D′ = 0 ∨ D′ = 1 then (0, 0)
        else if a = 1 then pell.fund-sol D
        else map-prod id (λy. y div a)
               (shd (sdrop-while (λ(-, y). y = 0 ∨ ¬a dvd y) (pell-solutions D′))))

lemma find-fund-sol-fast: find-fund-sol D = find-fund-sol-fast D
proof (cases is-square D ∨ square-part D = 1)
  case True
  thus ?thesis
    using squarefree-part-1-imp-square[of D]
    by (cases D = 0)
       (auto simp: find-fund-sol-correct find-fund-sol-fast-def
               square-squarefree-part-nat-def square-test-correct unit-factor-nat-def)
next
  case False
  define D′ a where D′ = squarefree-part D and a = square-part D
  have D > 0
    using False by (intro Nat.gr0I) auto
  have a > 0
    using ‹D > 0› by (intro Nat.gr0I) (auto simp: a-def)
  moreover have ¬is-square D′
    unfolding D′-def
   by (metis False is-nth-power-mult is-nth-power-nth-power squarefree-decompose)
  ultimately interpret lift: pell-lift D′ a D
    using False ‹D > 0›
    by unfold-locales (auto simp: D′-def a-def squarefree-decompose [symmetric])
```

define *i* **where** *i* = (*LEAST i. case lift.nth-solution i of* (-, *y*) ⇒ *y* > *0* ∧ *a dvd y*)
have *ex*: ∃ *i. case lift.nth-solution i of* (-, *y*) ⇒ *y* > *0* ∧ *a dvd y*
proof −
define *sol* **where** *sol* = *lift.lift.fund-sol*
have *is-sol*: *lift.solution* (*lift.lower-solution sol*)
unfolding *sol-def* **using** *lift.lift.fund-sol-is-nontriv-solution lift.lift-solution-iff* **by** *blast*
then obtain *j* **where** *j*: *lift.lower-solution sol* = *lift.nth-solution j*
using *lift.solution-iff-nth-solution* **by** *blast*
have *snd* (*lift.lower-solution sol*) > *0*
proof (*rule Nat.gr0I*)
assume ∗: *snd* (*lift.lower-solution sol*) = *0*
have *lift.solution* (*fst* (*lift.lower-solution sol*), *snd* (*lift.lower-solution sol*))
using *is-sol* **by** *simp*
hence *fst* (*lift.lower-solution sol*) = *1*
by (*subst* (*asm*) ∗) *simp*
with ∗ **have** *lift.lower-solution sol* = (*1*, *0*)
by (*cases lift.lower-solution sol*) *auto*
hence *fst sol* = *1*
unfolding *lift.lower-solution-def* **by** (*auto simp*: *lift.lower-solution-def case-prod-unfold*)
thus *False*
unfolding *sol-def*
using *lift.lift.fund-sol-is-nontriv-solution* ‹*D* > *0*›
by (*auto simp*: *lift.lift.nontriv-solution-def*)
qed
moreover have *a dvd snd* (*lift.lower-solution sol*)
by (*auto simp*: *lift.lower-solution-def case-prod-unfold*)
ultimately show *?thesis*
using *j* **by** (*auto simp*: *case-prod-unfold*)
qed

define *sol* **where** *sol* = *lift.nth-solution i*
have *sol*: *snd sol* > *0 a dvd snd sol*
using *LeastI-ex*[*OF ex*] **by** (*simp-all add*: *sol-def i-def case-prod-unfold*)
have *i* > *0*
using *sol* **by** (*intro Nat.gr0I*) (*auto simp*: *sol-def lift.nth-solution-def*)

have *find-fund-sol-fast D* = *map-prod id* (λ*y. y div a*)
(*shd* (*sdrop-while* (λ(-, *y*). *y* = *0* ∨ ¬*a dvd y*) (*pell-solutions D*′)))
unfolding *D*′*-def a-def find-fund-sol-fast-def* **using** *False squarefree-part-1-imp-square*[*of D*]
by (*auto simp*: *square-squarefree-part-nat-def*)
also have *sdrop-while* (λ(-, *y*). *y* = *0* ∨ ¬*a dvd y*) (*pell-solutions D*′) =
sdrop-while (*Not* ∘ (λ(-, *y*). *y* > *0* ∧ *a dvd y*)) (*pell-solutions D*′)
by (*simp add*: *o-def case-prod-unfold*)
also have ... = *sdrop i* (*pell-solutions D*′)
using *ex* **by** (*subst sdrop-while-sdrop-LEAST*) (*simp-all add*: *lift.snth-pell-solutions*

i-def)
  also have *shd* ... = *sol*
    by (*simp add*: *lift.snth-pell-solutions sol-def*)
  finally have *eq*: *find-fund-sol-fast D* = *map-prod id* (λ*y*. *y div a*) *sol* **.**

  have *lift.lift.fund-sol* = *lift.lift-solution sol*
    unfolding *sol-def*
  proof (*rule lift.lift-fund-sol*)
    show *i* > *0* **by** *fact*
    show *lift.liftable-solution* (*lift.nth-solution i*)
      using *sol* **by** (*simp add*: *sol-def lift.liftable-solution-def case-prod-unfold*)
  next
    fix *j* :: *nat* **assume** *j*: *j* > *0 j* < *i*
    show ¬*lift.liftable-solution* (*lift.nth-solution j*)
    proof
      assume *liftable*: *lift.liftable-solution* (*lift.nth-solution j*)
      have *snd* (*lift.nth-solution j*) > *0*
      using ‹*j* > *0*› **by** (*metis gr0I lift.nontriv-solution-altdef lift.nth-solution-sound'*

                                lift.solution-0-snd-nat-iff prod.collapse)
      hence *case lift.nth-solution j of* (-, *y*) ⇒ *y* > *0* ∧ *a dvd y*
        using ‹*j* > *0*› *liftable* **by** (*auto simp*: *lift.liftable-solution-def*)
      hence *i* ≤ *j*
        unfolding *i-def* **by** (*rule Least-le*)
      thus *False* **using** ‹*j* < *i*› **by** *simp*
    qed
  qed
  also have ... = *find-fund-sol-fast D*
    by (*simp add*: *eq lift.lift-solution-def case-prod-unfold map-prod-def*)
  finally show *?thesis*
    using ‹*D* > *0*› *False* **by** (*simp add*: *find-fund-sol-correct*)
qed

end

6 The Connection between the continued fraction expansion of square roots and Pell's equation

theory *Pell-Continued-Fraction*
imports
  Sqrt-Nat-Cfrac
  Pell.Pell-Algorithm
  Polynomial-Factorization.Prime-Factorization
  Pell-Lifting
begin

lemma *irrational-times-int-eq-intD*:
  assumes *p* ∗ *real-of-int a* = *real-of-int b*

assumes $p \notin \mathbb{Q}$
shows $a = 0 \wedge b = 0$
proof –
have $a = 0$
proof (*rule ccontr*)
assume $a \neq 0$
with *assms*(*1*) **have** $p = b / a$ **by** (*auto simp*: *field-simps*)
also have $\ldots \in \mathbb{Q}$ **by** *auto*
finally show *False* **using** *assms*(*2*) **by** *contradiction*
qed
with *assms* **show** *?thesis* **by** *simp*
qed

The solutions to Pell's equation for some non-square D are linked to the continued fraction expansion of $\sqrt{D}$, which we shall show here.

context
fixes D :: *nat* **and** *c h k P Q l*
assumes *nonsquare*: ¬*is-square D*
defines $c \equiv$ *cfrac-of-real* (*sqrt D*)
defines $h \equiv$ *conv-num c* **and** $k \equiv$ *conv-denom c*
defines $P \equiv$ *fst* ∘ *sqrt-remainder-surd D* **and** $Q \equiv$ *snd* ∘ *sqrt-remainder-surd D*
defines $l \equiv$ *sqrt-nat-period-length D*
begin

interpretation *pell D*
by *unfold-locales fact+*

lemma *cfrac-length-infinite* [*simp*]: *cfrac-length c* = ∞
proof –
have *sqrt D* ∉ ℚ
using *nonsquare* **by** (*simp add*: *irrat-sqrt-nonsquare*)
thus *?thesis*
by (*simp add*: *c-def*)
qed

lemma *conv-num-denom-pell*:
h 0 ^ *2* − *D* ∗ *k 0* ^ *2* < *0*
m > *0* ⟹ *h m* ^ *2* − *D* ∗ *k m* ^ *2* = (−*1*) ^ *Suc m* ∗ *Q m*
proof –
define *D'* **where** *D'* = *Discrete.sqrt D*
have *h 0* ^ *2* − *D* ∗ *k 0* ^ *2* = *int* (*D'* ^ *2*) − *int D*
by (*simp-all add*: *h-def k-def c-def Discrete-sqrt-altdef D'-def*)
also {
have *int* (*D'* ^ *2*) − *int D* ≤ *0*
using *Discrete.sqrt-power2-le*[*of D*] **by** (*simp add*: *D'-def*)
moreover have *D* ≠ *D'* ^ *2* **using** *nonsquare* **by** *auto*
ultimately have *int* (*D'* ^ *2*) − *int D* < *0* **by** *linarith*
}
finally show *h 0* ^ *2* − *D* ∗ *k 0* ^ *2* < *0* **.**

```
next
  assume m > 0
  define n where n = m − 1
  define α where α = cfrac-remainder c
  define α′ where α′ = sqrt-remainder-surd D
  have m: m = Suc n using ‹m > 0› by (simp add: n-def)
  from nonsquare have D > 1
    by (cases D) (auto intro!: Nat.gr0I)
  from nonsquare have irrat: sqrt D ∉ ℚ
    using irrat-sqrt-nonsquare by blast
  have [simp]: cfrac-lim c = sqrt D
    using irrat ‹D > 1› by (simp add: c-def)
  have α-pos: α n > 0 for n
    unfolding α-def using wf ‹D > 1› cfrac-remainder-pos[of c n]
    by (cases n = 0) auto
  have α′: α′ n = (P n, Q n) for n by (simp add: α′-def P-def Q-def)
  have Q-pos: Q n > 0 for n
    using snd-sqrt-remainder-surd-pos[OF nonsquare] by (simp add: Q-def)
  have k-pos: k n > 0 for n
    by (auto simp: k-def intro!: conv-denom-pos)
  have k-nonneg: k n ≥ 0 for n
    by (auto simp: k-def intro!: conv-denom-nonneg)

  let ?A = (sqrt D + P (n + 1)) ∗ h (n + 1) + Q (n + 1) ∗ h n
  let ?B = (sqrt D + P (n + 1)) ∗ k (n + 1) + Q (n + 1) ∗ k n
  have ?B > 0 using k-pos Q-pos k-nonneg
    by (intro add-nonneg-pos mult-nonneg-nonneg add-nonneg-nonneg) auto

  have sqrt D = conv′ c (Suc (Suc n)) (α (Suc (Suc n)))
    unfolding α-def by (subst conv′-cfrac-remainder) auto
  also have . . . = (α (n + 2) ∗ h (n + 1) + h n) / (α (n + 2) ∗ k (n + 1) + k n)
    using wf α-pos by (subst conv′-num-denom) (simp-all add: h-def k-def)
  also have α (n + 2) = surd-to-real D (α′ (Suc n))
    using surd-to-real-sqrt-remainder-surd[OF nonsquare, of Suc n]
    by (simp add: α′-def α-def c-def)
  also have . . . = (sqrt D + P (Suc n)) / Q (Suc n) (is - = ?α)
    by (simp add: α′ surd-to-real-def)
  also have ?α ∗ h (n + 1) + h n =
            1 / Q (n + 1) ∗ ((sqrt D + P (n + 1)) ∗ h (n + 1) + Q (n + 1) ∗ h n)
    using Q-pos by (simp add: field-simps)
  also have ?α ∗ k (n + 1) + k n =
            1 / Q (n + 1) ∗ ((sqrt D + P (n + 1)) ∗ k (n + 1) + Q (n + 1) ∗ k n)
    (is - = ?f k) using Q-pos by (simp add: field-simps)
  also have ?f h / ?f k = ((sqrt D + P (n + 1)) ∗ h (n + 1) + Q (n + 1) ∗ h n) /
                           ((sqrt D + P (n + 1)) ∗ k (n + 1) + Q (n + 1) ∗ k n)
    (is - = ?A / ?B) using Q-pos by (intro mult-divide-mult-cancel-left) auto
  finally have sqrt D ∗ ?B = ?A
    using ‹?B > 0› by (simp add: divide-simps)
  moreover have sqrt D ∗ sqrt D = D by simp
```

ultimately have *sqrt D* ∗ (*P* (*n* + *1*) ∗ *k* (*n* + *1*) + *Q* (*n* + *1*) ∗ *k n* − *h* (*n* + *1*)) =
P (*n* + *1*) ∗ *h* (*n* + *1*) + *Q* (*n* + *1*) ∗ *h n* − *k* (*n* + *1*) ∗ *D*
unfolding *of-int-add of-int-mult of-int-diff of-int-of-nat-eq of-nat-mult of-nat-add*
by *Groebner-Basis.algebra*
from *irrational-times-int-eq-intD*[*OF this*] *irrat*
have *1*: *h* (*Suc n*) = *P* (*Suc n*) ∗ *k* (*Suc n*) + *Q* (*Suc n*) ∗ *k n*
and *2*: *D* ∗ *k* (*Suc n*) = *P* (*Suc n*) ∗ *h* (*Suc n*) + *Q* (*Suc n*) ∗ *h n*
by (*simp-all del*: *of-nat-add of-nat-mult*)

have *h* (*Suc n*) ∗ *h* (*Suc n*) − *D* ∗ *k* (*Suc n*) ∗ *k* (*Suc n*) =
Q (*Suc n*) ∗ (*k n* ∗ *h* (*Suc n*) − *k* (*Suc n*) ∗ *h n*)
by (*subst 1*, *subst 2*) (*simp add*: *algebra-simps*)
also have *k n* ∗ *h* (*Suc n*) − *k* (*Suc n*) ∗ *h n* = (−*1*) ^ *n*
unfolding *h-def k-def* **by** (*rule conv-num-denom-prod-diff*)
finally have *h* (*Suc n*) ^ *2* − *D* ∗ *k* (*Suc n*) ^ *2* = (−*1*) ^ *n* ∗ *Q* (*Suc n*)
by (*simp add*: *power2-eq-square algebra-simps*)
thus *h m* ^ *2* − *D* ∗ *k m* ^ *2* = (−*1*) ^ *Suc m* ∗ *Q m*
by (*simp add*: *m*)
qed

Every non-trivial solution to Pell's equation is a convergent in the expansion of $\sqrt{D}$:

theorem *pell-solution-is-conv*:
assumes x^2 = *Suc* (*D* ∗ y^2) **and** *y* > *0*
shows (*int x*, *int y*) ∈ *range* (λ*n*. (*conv-num c n*, *conv-denom c n*))
proof −
have ∃ *n*. *enat n* ≤ *cfrac-length c* ∧ (*int x*, *int y*) = (*conv-num c n*, *conv-denom c n*)
proof (*rule frac-is-convergentI*)
have *gcd* (x^2) (y^2) = *1* **unfolding** *assms*(*1*)
using *gcd-add-mult*[*of* y^2 *D 1*] **by** (*simp add*: *gcd.commute*)
thus *coprime* (*int x*) (*int y*)
by (*simp add*: *coprime-iff-gcd-eq-1*)
next
from *assms* **have** *D* > *1*
using *nonsquare* **by** (*cases D*) (*auto intro*!: *Nat.gr0I*)
hence *pos*: *x* + *y* ∗ *sqrt D* > *0* **using** *assms*
by (*intro add-nonneg-pos*) *auto*

from *assms* **have** *real* (x^2) = *real* (*Suc* (*D* ∗ y^2))
by (*simp only*: *of-nat-eq-iff*)
hence *1* = *real x* ^ *2* − *D* ∗ *real y* ^ *2*
unfolding *of-nat-power* **by** *simp*
also have ... = (*x* − *y* ∗ *sqrt D*) ∗ (*x* + *y* ∗ *sqrt D*)
by (*simp add*: *field-simps power2-eq-square*)
finally have ∗: *x* − *y* ∗ *sqrt D* = *1* / (*x* + *y* ∗ *sqrt D*)
using *pos* **by** (*simp add*: *field-simps*)

```
    from pos have 0 < 1 / (x + y * sqrt D)
      by (intro divide-pos-pos) auto
    also have ... = x − y * sqrt D by (rule * [symmetric])
    finally have less: y * sqrt D < x by simp

    have sqrt D − x / y = −((x − y * sqrt D) / y)
      using ‹y > 0› by (simp add: field-simps)
    also have |...| = (x − y * sqrt D) / y
      using less by simp
    also have (x − y * sqrt D) / y = 1 / (y * (x + y * sqrt D))
      using ‹y > 0› by (subst *) auto
    also have ... ≤ 1 / (y * (y * sqrt D + y * sqrt D))
      using ‹y > 0› ‹D > 1› pos less
      by (intro divide-left-mono mult-left-mono add-right-mono mult-pos-pos) auto
    also have ... = 1 / (2 * y² * sqrt D)
      by (simp add: power2-eq-square)
    also have ... < 1 / (real (2 * y²) * 1) using ‹y > 0› ‹D > 1›
      by (intro divide-strict-left-mono mult-strict-left-mono mult-pos-pos) auto
    finally show |cfrac-lim c − int x / int y| < 1 / (2 * int y ^ 2)
      unfolding c-def using irrat-sqrt-nonsquare[of D] ‹¬is-square D› by simp
  qed (insert assms irrat-sqrt-nonsquare[of D], auto simp: c-def)
  thus ?thesis by auto
qed
```

Let l be the length of the period in the continued fraction expansion of $\sqrt{D}$ and let h_i and k_i be the numerator and denominator of the i-th convergent. Then the non-trivial solutions of Pell's equation are exactly the pairs of the form (h_{lm-1}, k_{lm-1}) for any m such that lm is even.

```
lemma nontriv-solution-iff-conv-num-denom:
  nontriv-solution (x, y) ⟷
    (∃ m>0. int x = h (l * m − 1) ∧ int y = k (l * m − 1) ∧ even (l * m))
proof safe
  fix m assume xy: x = h (l * m − 1) y = k (l * m − 1)
         and lm: even (l * m) and m: m > 0
  have l: l > 0 using period-nonempty[OF nonsquare] by (auto simp: l-def)
  from lm have l * m ≠ 1 by (intro notI) auto
  with l m have lm': l * m > 1 by (cases l * m) auto

  have (h (l * m − 1))² − D * (k (l * m − 1))² =
        (− 1) ^ Suc (l * m − 1) * int (Q (l * m − 1))
    using lm' by (intro conv-num-denom-pell) auto
  also have (− 1) ^ Suc (l * m − 1) = (1 :: int)
    using lm l m by (subst neg-one-even-power) auto
  also have Q (l * m − 1) = Q ((l * m − 1) mod l)
    unfolding Q-def l-def o-def by (subst sqrt-remainder-surd-periodic[OF non-
square]) simp
  also {
    have l * m − 1 = (m − 1) * l + (l − 1)
      using m l lm' by (cases m) (auto simp: mult-ac)
```

also have ... *mod l* = (*l* − *1*) *mod l*
by *simp*
also have ... = *l* − *1*
using *l* **by** (*intro mod-less*) *auto*
also have *Q* ... = *1*
using *sqrt-remainder-surd-last*[*OF nonsquare*] **by** (*simp add*: *Q-def l-def*)
finally have *Q* ((*l* ∗ *m* − *1*) *mod l*) = *1* **.**
}
finally have *h* (*l* ∗ *m* − *1*) ^ *2* = *D* ∗ *k* (*l* ∗ *m* − *1*) ^ *2* + *1*
unfolding *of-nat-Suc* **by** (*simp add*: *algebra-simps*)
hence *h* (*l* ∗ *m* − *1*) ^ *2* = *D* ∗ *k* (*l* ∗ *m* − *1*) ^ *2* + *1*
by (*simp only*: *of-nat-eq-iff*)
moreover have *k* (*l* ∗ *m* − *1*) > *0*
unfolding *k-def* **by** (*intro conv-denom-pos*)
ultimately have *nontriv-solution* (*int x*, *int y*)
using *xy* **by** (*simp add*: *nontriv-solution-def*)
thus *nontriv-solution* (*x*, *y*)
by *simp*
next
assume *nontriv-solution* (*x*, *y*)
hence *asm*: *x* ^ *2* = *Suc* (*D* ∗ *y* ^ *2*) *y* > *0*
by (*auto simp*: *nontriv-solution-def abs-square-eq-1 intro*!: *Nat.gr0I*)
from *asm* **have** *asm*′: *int x* ^ *2* = *int D* ∗ *int y* ^ *2* + *1*
by (*metis add.commute of-nat-Suc of-nat-mult of-nat-power-eq-of-nat-cancel-iff*)
have *l*: *l* > *0* **using** *period-nonempty*[*OF nonsquare*] **by** (*auto simp*: *l-def*)
from *pell-solution-is-conv*[*OF asm*] **obtain** *m* **where**
xy: *h m* = *x k m* = *y* **by** (*auto simp*: *c-def h-def k-def*)

have *m*: *m* > *0*
using *asm*′ *conv-num-denom-pell*(*1*) *xy* **by** (*intro Nat.gr0I*) *auto*
have *1* = *h m* ^ *2* − *D* ∗ *k m* ^ *2*
using *asm*′ *xy* **by** *simp*
also have ... = (− *1*) ^ *Suc m* ∗ *int* (*Q m*)
using *conv-num-denom-pell*(*2*)[*OF m*] **.**
finally have ∗: (− *1*) ^ *Suc m* ∗ *int* (*Q m*) = *1* **..**
from ∗ **have** *m*′: *odd m* ∧ *Q m* = *1*
by (*cases even m*) *auto*

define *n* **where** *n* = *Suc m div l*
have *l dvd Suc m*
proof (*rule ccontr*)
assume ∗: ¬(*l dvd Suc m*)
have *Q m* = *Q* (*m mod l*)
unfolding *Q-def l-def o-def* **by** (*subst sqrt-remainder-surd-periodic*[*OF non-square*]) *simp*
also {
have *m mod l* < *l* **using** ‹*l* > *0*› **by** *simp*
moreover have *Suc* (*m mod l*) ≠ *l* **using** ∗ *l* ‹*m* > *0*›
using *mod-Suc*[*of m l*] **by** *auto*

ultimately have *m mod l* < *l* − *1* **by** *simp*
hence *Q* (*m mod l*) > *1* **unfolding** *Q-def o-def l-def*
by (*rule snd-sqrt-remainder-surd-gt-1*[*OF nonsquare*])
}
finally show *False* **using** *m′* **by** *simp*
qed
hence *m-eq*: *Suc m* = *n* ∗ *l m* = *n* ∗ *l* − *1*
by (*simp-all add*: *n-def*)
hence *n* > *0* **by** (*auto intro*!: *Nat.gr0I*)
thus ∃ *n*>*0*. *int x* = *h* (*l* ∗ *n* − *1*) ∧ *int y* = *k* (*l* ∗ *n* − *1*) ∧ *even* (*l* ∗ *n*)
using *xy m-eq m′* **by** (*intro exI*[*of - n*]) (*auto simp*: *mult-ac*)
qed

Consequently, the fundamental solution is (h_n, k_n) where $n = l - 1$ if l is even and $n = 2l - 1$ otherwise:

lemma *fund-sol-conv-num-denom*:
defines *n* ≡ *if even l then l* − *1 else 2* ∗ *l* − *1*
shows *fund-sol* = (*nat* (*h n*), *nat* (*k n*))
proof (*rule fund-sol-eq-sndI*)
have [*simp*]: *h n* ≥ *0 k n* ≥ *0* **for** *n*
by (*auto simp*: *h-def k-def c-def intro*!: *conv-num-nonneg*)
show *nontriv-solution* (*nat* (*h n*), *nat* (*k n*))
by (*subst nontriv-solution-iff-conv-num-denom*, *rule exI*[*of - if even l then 1 else 2*])
(*simp-all add*: *n-def mult-ac*)
next
fix *x y* :: *nat* **assume** *nontriv-solution* (*x*, *y*)
then obtain *m* **where** *m*: *m* > *0 x* = *h* (*l* ∗ *m* − *1*) *y* = *k* (*l* ∗ *m* − *1*) *even* (*l* ∗ *m*)
by (*subst* (*asm*) *nontriv-solution-iff-conv-num-denom*) *auto*
have *l*: *l* > *0* **using** *period-nonempty*[*OF nonsquare*] **by** (*auto simp*: *l-def*)
from *m l* **have** *Suc n* ≤ *l* ∗ *m* **by** (*auto simp*: *n-def*)
hence *n* ≤ *l* ∗ *m* − *1* **by** *simp*
hence *k n* ≤ *k* (*l* ∗ *m* − *1*)
unfolding *k-def c-def* **using** *irrat-sqrt-nonsquare*[*OF nonsquare*]
by (*intro conv-denom-leI*) *auto*
with *m* **show** *nat* (*k n*) ≤ *y* **by** *simp*
qed

end

The following algorithm computes the fundamental solution (or the dummy result (*0*, *0*) if D is a square) fairly quickly by computing the continued fraction expansion of $\sqrt{D}$ and then computing the fundamental solution as the appropriate convergent.

lemma *find-fund-sol-code* [*code*]:
find-fund-sol D =
(*let info* = *sqrt-cfrac-info-array D*;
l = *fst info*

in if l = 0 then (0, 0) else
let
c = cfrac-sqrt-nth info;
n = if even l then l − 1 else 2 ∗ l − 1
in
(nat (conv-num-fun c n), nat (conv-denom-fun c n)))
proof −
have ∗: *is-cfrac (cfrac-sqrt-nth (sqrt-cfrac-info-array D))* **if** ¬*is-square D*
using *that cfrac-sqrt-nth[of D]* **unfolding** *is-cfrac-def*
by *(metis cfrac-nth-nonzero neq0-conv of-nat-0 of-nat-0-less-iff)*
have ∗∗: *cfrac (λx. int (cfrac-sqrt-nth (sqrt-cfrac-info-array D) x)) = cfrac-of-real (sqrt D)*
if ¬*is-square D*
using *that cfrac-sqrt-nth[of D]* ∗ **by** *(intro cfrac-eqI) auto*
show *?thesis* **using** ∗ ∗∗
by *(auto simp: square-test-correct find-fund-sol-correct conv-num-fun-eq conv-denom-fun-eq Let-def cfrac-sqrt-nth fund-sol-conv-num-denom conv-num-nonneg)*
qed

lemma *find-nth-solution-square [simp]: is-square D* ⟹ *find-nth-solution D n = (0, 0)*
by *(simp add: find-nth-solution-def)*

lemma *fst-find-fund-sol-eq-0-iff [simp]: fst (find-fund-sol D) = 0* ⟷ *is-square D*
proof *(cases is-square D)*
case *False*
then interpret *pell D* **by** *unfold-locales*
from *False* **have** *find-fund-sol D = fund-sol* **by** *(simp add: find-fund-sol-correct)*
moreover from *fund-sol-is-nontriv-solution* **have** *fst fund-sol > 0*
by *(auto simp: nontriv-solution-def intro!: Nat.gr0I)*
ultimately show *?thesis* **using** *False*
by *(simp add: find-fund-sol-def square-test-correct split: if-splits)*
qed *(auto simp: find-fund-sol-def square-test-correct)*

Arbitrary solutions can now be computed as powers of the fundamental solution.

lemma *find-nth-solution-code [code]:*
find-nth-solution D n =
(let xy = find-fund-sol D
in if fst xy = 0 then (0, 0) else efficient-pell-power D xy n)
proof *(cases is-square D)*
case *False*
then interpret *pell D* **by** *unfold-locales*
from *fund-sol-is-nontriv-solution* **have** *fst fund-sol > 0*
by *(auto simp: nontriv-solution-def intro!: Nat.gr0I)*
thus *?thesis* **using** *False*
by *(simp add: find-nth-solution-correct Let-def nth-solution-def pell-power-def pell-mul-commutes[of - fund-sol] find-fund-sol-correct)*
qed *auto*

```
lemma nth-solution-code [code]:
  pell.nth-solution D n =
    (let info = sqrt-cfrac-info-array D;
         l = fst info
     in  if l = 0 then
         Code.abort (STR ''nth-solution is undefined for perfect square parameter.'')
              (λ-. pell.nth-solution D n)
         else
           let
             c = cfrac-sqrt-nth info;
             m = if even l then l − 1 else 2 ∗ l − 1;
             fund-sol = (nat (conv-num-fun c m), nat (conv-denom-fun c m))
           in
             efficient-pell-power D fund-sol n)
proof (cases is-square D)
  case False
  then interpret pell by unfold-locales
  have ∗: is-cfrac (cfrac-sqrt-nth (sqrt-cfrac-info-array D))
    using False cfrac-sqrt-nth[of D] unfolding is-cfrac-def
    by (metis cfrac-nth-nonzero neq0-conv of-nat-0 of-nat-0-less-iff)
 have ∗∗: cfrac (λx. int (cfrac-sqrt-nth (sqrt-cfrac-info-array D) x)) = cfrac-of-real
(sqrt D)
    using False cfrac-sqrt-nth[of D] ∗ by (intro cfrac-eqI) auto

  from False ∗ ∗∗ show ?thesis
    by (auto simp: Let-def cfrac-sqrt-nth fund-sol-conv-num-denom nth-solution-def
                  pell-power-def pell-mul-commutes[of - (-, -)]
                  conv-num-fun-eq conv-denom-fun-eq conv-num-nonneg)
qed auto

lemma fund-sol-code [code]:
  pell.fund-sol D = (let info = sqrt-cfrac-info-array D;
         l = fst info
     in  if l = 0 then
            Code.abort (STR ''fund-sol is undefined for perfect square parameter.'')
              (λ-. pell.fund-sol D)
         else
           let
             c = cfrac-sqrt-nth info;
             n = if even l then l − 1 else 2 ∗ l − 1
           in
             (nat (conv-num-fun c n), nat (conv-denom-fun c n)))
proof (cases is-square D)
  case False
  then interpret pell by unfold-locales
  have ∗: is-cfrac (cfrac-sqrt-nth (sqrt-cfrac-info-array D))
    using False cfrac-sqrt-nth[of D] unfolding is-cfrac-def
    by (metis cfrac-nth-nonzero neq0-conv of-nat-0 of-nat-0-less-iff)
```

```
  have **: cfrac (λx. int (cfrac-sqrt-nth (sqrt-cfrac-info-array D) x)) = cfrac-of-real
(sqrt D)
    using False cfrac-sqrt-nth[of D] * by (intro cfrac-eqI) auto

  from False * ** show ?thesis
    by (auto simp: Let-def cfrac-sqrt-nth fund-sol-conv-num-denom nth-solution-def
                   pell-power-def pell-mul-commutes[of - (-, -)]
                   conv-num-fun-eq conv-denom-fun-eq conv-num-nonneg)
qed auto

end
```

7 Tests for Continued Fractions of Square Roots and Pell's Equation

```
theory Pell-Continued-Fraction-Tests
imports
  Pell.Efficient-Discrete-Sqrt
  HOL−Library.Code-Lazy
  HOL−Library.Code-Target-Numeral
  Pell-Continued-Fraction
  Pell-Lifting
begin

code-lazy-type stream


lemma lnth-code [code]:
 lnth xs 0 = (if lnull xs then undefined (0 :: nat) else lhd xs)
 lnth xs (Suc n) = (if lnull xs then undefined (Suc n) else lnth (ltl xs) n)
  by (auto simp: lnth.simps split: llist.splits)

value let c = sqrt-cfrac 1339 in map (cfrac-nth c) [0..<30]


fun arg-max-list where
  arg-max-list - [] = undefined
| arg-max-list f (x # xs) =
    foldl (λ(x, y) x'. let y' = f x' in if y' > y then (x', y') else (x, y)) (x, f x) xs


value [code] sqrt-cfrac-info 17
value [code] sqrt-cfrac-info 1339
value [code] sqrt-cfrac-info 121
value [code] sqrt-nat-period-length 4102864223278424
```

For which number $D < 100000$ does $\sqrt{D}$ have the longest period?

```
value [code] arg-max-list sqrt-nat-period-length [0..<100000]
```

7.1 Fundamental solutions of Pell's equation

value [*code*] *pell.fund-sol 12*
value [*code*] *pell.fund-sol 13*
value [*code*] *pell.fund-sol 61*
value [*code*] *pell.fund-sol 661*
value [*code*] *pell.fund-sol 6661*
value [*code*] *pell.fund-sol 4729494*

Project Euler problem #66: For which $D < 1000$ does Pell's equation have the largest fundamental solution?

value [*code*] *arg-max-list* (*fst* ∘ *find-fund-sol*) [*0..<1001*]

The same for $D < 100000$:

value [*code*] *arg-max-list* (*fst* ∘ *find-fund-sol*) [*0..<100000*]

The solution to the next example, which is at the core of Archimedes' cattle problem, is so big that termifying the result takes extremely long. Therefore, we simply compute the number of decimal digits in the result instead.

```
fun log10-aux :: nat ⇒ nat ⇒ nat where
  log10-aux acc n =
    (if n ≥ 10000000000 then log10-aux (acc + 10) (n div 10000000000)
     else if n = 0 then acc else log10-aux (Suc acc) (n div 10))
```

definition *log10* **where** *log10* = *log10-aux 0*

value [*code*] *map-prod log10 log10* (*pell.fund-sol 410286423278424*)

Factoring out the square factor 9314^2 does yield a significant speed-up in this case:

value [*code*] *map-prod log10 log10* (*find-fund-sol-fast 410286423278424*)

7.2 Tests for other operations

value [*code*] *pell.nth-solution 13 100*
value [*code*] *pell.nth-solution 4729494 3*

value [*code*] *stake 10* (*pell-solutions 13*)
value [*code*] *stake 10* (*pell-solutions 61*)

value [*code*] *pell.nth-solution 23 8*

end

8 Computing continued fraction expansions through interval arithmetic

theory *Continued-Fraction-Approximation*

imports
Complex-Main
HOL−Decision-Procs.Approximation
Coinductive.Coinductive-List
HOL−Library.Code-Lazy
HOL−Library.Code-Target-Numeral
Continued-Fractions
keywords *approximate-cfrac* :: *diag*
begin

The approximation package allows us to compute an enclosing interval for a given real constant. From this, we are able to compute an initial fragment of the continued fraction expansion of the number.

The algorithm essentially works by computing the continued fraction expansion of the lower and upper bound simultaneously and stopping when the results start to diverge.

This algorithm terminates because the lower and upper bounds, being rational numbers, have a finite continued fraction expansion.

```
definition float-to-rat :: float ⇒ int × int where
  float-to-rat f = (if exponent f ≥ 0 then
     (mantissa f ∗ 2 ^ nat (exponent f), 1) else (mantissa f, 2 ^ nat (−exponent
f)))

lemma float-to-rat: fst (float-to-rat f) / snd (float-to-rat f) = real-of-float f
  by (auto simp: float-to-rat-def mantissa-exponent powr-int)

lemma snd-float-to-rat-pos [simp]: snd (float-to-rat f) > 0
  by (simp add: float-to-rat-def)


function cfrac-from-approx :: int × int ⇒ int × int ⇒ int list where
  cfrac-from-approx (nl, dl) (nu, du) =
    (if nl = 0 ∨ nu = 0 ∨ dl = 0 ∨ du = 0 then []
     else let l = nl div dl; u = nu div du
          in  if l ≠ u then []
              else l # (let m = nl mod dl in if m = 0 then [] else
                        cfrac-from-approx (du, nu mod du) (dl, m)))
  by auto
termination proof (relation measure (λ((nl, dl), (nu, du)). nat (abs dl + abs
du)), goal-cases)
  case (2 nl dl nu du)
  hence |nl mod dl| + |nu mod du| < |dl| + |du|
    by (intro add-strict-mono) (auto simp: abs-mod-less)
  thus ?case using 2 by simp
qed auto

lemmas [simp del] = cfrac-from-approx.simps
```

lemma *cfrac-from-approx-correct*:
  assumes $x \in \{fst\ l\ /\ snd\ l..fst\ u\ /\ snd\ u\}$ **and** $snd\ l > 0$ **and** $snd\ u > 0$
  assumes $i < length\ (cfrac\text{-}from\text{-}approx\ l\ u)$
  shows $cfrac\text{-}nth\ (cfrac\text{-}of\text{-}real\ x)\ i = cfrac\text{-}from\text{-}approx\ l\ u\ !\ i$
  using *assms*
proof (*induction l u arbitrary*: *i x rule*: *cfrac-from-approx.induct*)
  case (*1 nl dl nu du i x*)
  from *1.prems* **have** ∗: $nl\ div\ dl = nu\ div\ du\ nl \neq 0\ nu \neq 0\ dl > 0\ du > 0$
    by (*auto simp*: *cfrac-from-approx.simps Let-def split*: *if-splits*)
  have $\lfloor nl\ /\ dl \rfloor \leq \lfloor x \rfloor\ \lfloor x \rfloor \leq \lfloor nu\ /\ du \rfloor$
    using *1.prems*(*1*) **by** (*intro floor-mono*; *simp*)+
  hence $nl\ div\ dl \leq \lfloor x \rfloor\ \lfloor x \rfloor \leq nu\ div\ du$
    by (*simp-all add*: *floor-divide-of-int-eq*)
  with ∗ **have** $\lfloor x \rfloor = nu\ div\ du$
    by *linarith*

  show *?case*
  proof (*cases i*)
    case *0*
    with *0* **and** ‹$\lfloor x \rfloor$ = -› **show** *?thesis* **using** *1.prems*
      by (*auto simp*: *Let-def cfrac-from-approx.simps*)
  next
    case [*simp*]: (*Suc i′*)
    from *1.prems* ∗ **have** $nl\ mod\ dl \neq 0$
      by (*subst* (*asm*) *cfrac-from-approx.simps*) (*auto split*: *if-splits*)
    have *frac-eq*: $frac\ x = x - nu\ div\ du$
      using ‹$\lfloor x \rfloor$ = -› **by** (*simp add*: *frac-def*)

    have $frac\ x \geq nl\ /\ dl - nl\ div\ dl$
      using ∗ *1.prems* **by** (*simp add*: *frac-eq*)
    also have $nl\ /\ dl - nl\ div\ dl = (nl - dl * (nl\ div\ dl))\ /\ dl$
      using ∗ **by** (*simp add*: *field-simps*)
    also have $nl - dl * (nl\ div\ dl) = nl\ mod\ dl$
      by (*subst minus-div-mult-eq-mod* [*symmetric*]) *auto*
    finally have $frac\ x \geq (nl\ mod\ dl)\ /\ dl$ **.**

    have $nl\ mod\ dl \geq 0$
      using ∗ **by** (*intro pos-mod-sign*) *auto*
    with ‹$nl\ mod\ dl \neq 0$› **have** $nl\ mod\ dl > 0$
      by *linarith*
    hence $0 < (nl\ mod\ dl)\ /\ dl$
      using ∗ **by** (*intro divide-pos-pos*) *auto*
    also have $\ldots \leq frac\ x$
      by *fact*
    finally have $frac\ x > 0$ **.**

    have $frac\ x \leq nu\ /\ du - nu\ div\ du$
      using ∗ *1.prems* **by** (*simp add*: *frac-eq*)
    also have $\ldots = (nu - du * (nu\ div\ du))\ /\ du$

using ∗ **by** (*simp add*: *field-simps*)
also have *nu* − *du* ∗ (*nu div du*) = *nu mod du*
by (*subst minus-div-mult-eq-mod* [*symmetric*]) *auto*
finally have *frac x* ≤ *real-of-int* (*nu mod du*) / *real-of-int du* **.**

have *0* < *frac x*
by *fact*
also have . . . ≤ (*nu mod du*) / *du*
by *fact*
finally have *nu mod du* > *0*
using ∗ **by** (*auto simp*: *field-simps*)

have *cfrac-nth* (*cfrac-of-real x*) *i* = *cfrac-nth* (*cfrac-tl* (*cfrac-of-real x*)) *i′*
by *simp*
also have *cfrac-tl* (*cfrac-of-real x*) = *cfrac-of-real* (*1* / *frac x*)
using ‹*frac x* > *0*› **by** (*intro cfrac-tl-of-real*) *auto*
also have *cfrac-nth* (*cfrac-of-real* (*1* / *frac x*)) *i′* =
cfrac-from-approx (*du*, *nu mod du*) (*dl*, *nl mod dl*) ! *i′*
proof (*rule 1.IH*[*OF - refl refl - refl*])
show ¬ (*nl* = *0* ∨ *nu* = *0* ∨ *dl* = *0* ∨ *du* = *0*) ¬ *nl div dl* ≠ *nu div du*
using *1.prems* **by** (*auto split*: *if-splits simp*: *Let-def cfrac-from-approx.simps*)
next
show *i′* < *length* (*cfrac-from-approx* (*du*, *nu mod du*) (*dl*, *nl mod dl*)) **using** *1.prems*
by (*subst* (*asm*) *cfrac-from-approx.simps*) (*auto split*: *if-splits simp*: *Let-def*)
next
have *1* / *frac x* ≤ *dl* / (*nl mod dl*)
using ‹*frac x* > *0*› **and** ‹*nl mod dl* > *0*› **and** ‹*frac x* ≥ (*nl mod dl*) / *dl*› **and** ∗
by (*auto simp*: *field-simps*)
moreover have *1* / *frac x* ≥ *du* / (*nu mod du*)
using ‹*frac x* > *0*› **and** ‹*nu mod du* > *0*› **and** ‹*frac x* ≤ (*nu mod du*) / *du*› **and** ∗
by (*auto simp*: *field-simps*)
ultimately show
1 / *frac x* ∈ {*real-of-int* (*fst* (*du*, *nu mod du*)) / *real-of-int* (*snd* (*du*, *nu mod du*))..
real-of-int (*fst* (*dl*, *nl mod dl*)) / *real-of-int* (*snd* (*dl*, *nl mod dl*))}
by *simp*
show *snd* (*du*, *nu mod du*) > *0 snd* (*dl*, *nl mod dl*) > *0* **and** *nl mod dl* ≠ *0*
using ‹*nu mod du* > *0*› **and** ‹*nl mod dl* > *0*› **by** *simp-all*
qed
also have *cfrac-from-approx* (*du*, *nu mod du*) (*dl*, *nl mod dl*) ! *i′* =
cfrac-from-approx (*nl*, *dl*) (*nu*, *du*) ! *i*
using *1.prems* ∗ ‹*nl mod dl* ≠ *0*› **by** (*subst* (*2*) *cfrac-from-approx.simps*) *auto*
finally show *?thesis* **.**
qed
qed

definition *cfrac-from-approx′* :: *float* ⇒ *float* ⇒ *int list* **where**
cfrac-from-approx′ l u = *cfrac-from-approx* (*float-to-rat l*) (*float-to-rat u*)

lemma *cfrac-from-approx′-correct*:
assumes $x \in \{$*real-of-float l..real-of-float u*$\}$
assumes $i <$ *length* (*cfrac-from-approx′ l u*)
shows *cfrac-nth* (*cfrac-of-real x*) *i* = *cfrac-from-approx′ l u* ! *i*
using *assms* **unfolding** *cfrac-from-approx′-def*
by (*intro cfrac-from-approx-correct*) (*auto simp*: *float-to-rat cfrac-from-approx′-def*)

definition *approx-cfrac* :: *nat* ⇒ *floatarith* ⇒ *int list* **where**
approx-cfrac prec e =
(*case approx′ prec e* [] *of*
None ⇒ []
| *Some ivl* ⇒ *cfrac-from-approx′* (*lower ivl*) (*upper ivl*))

ML-file ‹*approximation-cfrac.ML*›

Now let us do some experiments:

value *let prec* = *34*; *c* = *cfrac-from-approx′* (*lb-pi prec*) (*ub-pi prec*) *in c*
value *let prec* = *34*; *c* = *cfrac-from-approx′* (*lb-pi prec*) (*ub-pi prec*)
in map (λ*n*. (*conv-num-fun* ((!) *c*) *n*, *conv-denom-fun* ((!) *c*) *n*)) [*0..*<*length c*]

approximate-cfrac *prec*: *200 pi*
approximate-cfrac *ln 2*
approximate-cfrac *exp 1*
approximate-cfrac *sqrt 129*
approximate-cfrac (*sqrt 13* + *3*) / *4*
approximate-cfrac *arctan 1*

approximate-cfrac *123* / *97*
value *cfrac-list-of-rat* (*123*, *97*)

end

References

[1] A. Khinchin and H. Eagle. *Continued Fractions*. Dover books on mathematics. Dover Publications, 1997.

[2] Proof Wiki.