Pell's Equation

Manuel Eberl

March 17, 2025

Abstract

This article gives the basic theory of Pell's equation $x^2 = 1 + Dy^2$, where $D \in \mathbb{N}$ is a parameter and x, y are integer variables.

The main result that is proven is the following: If D is not a perfect square, then there exists a *fundamental solution* (x_0, y_0) that is not the trivial solution $(1, 0)$ and which generates all other solutions (x, y) in the sense that there exists some $n \in \mathbb{N}$ such that $|x| + |y|\sqrt{D} = (x_0 + y_0\sqrt{D})^n$. This also implies that the set of solutions is infinite, and it gives us an explicit and executable characterisation of all the solutions.

Based on this, simple executable algorithms for computing the fundamental solution and the infinite sequence of all non-negative solutions are also provided.

Contents

theory *Efficient-Discrete-Sqrt*
imports
Complex-Main
HOL−Computational-Algebra.Computational-Algebra
HOL−Library.Discrete-Functions
HOL−Library.Tree
HOL−Library.IArray
begin

1 Efficient Algorithms for the Square Root on $\mathbb{N}$

1.1 A Discrete Variant of Heron's Algorithm

An algorithm for calculating the discrete square root, taken from Cohen [2]. This algorithm is essentially a discretised variant of Heron's method or Newton's method specialised to the square root function.

lemma *sqrt-eq-floor-sqrt*: *floor-sqrt n* $=$ *nat* $\lfloor$*sqrt n*$\rfloor$
proof −
have *real* $((nat\ \lfloor sqrt\ n\rfloor)^2) = (real\ (nat\ \lfloor sqrt\ n\rfloor))^2$
by *simp*
also have $\ldots \leq$ *sqrt* (*real n*) ^ *2*
by (*intro power-mono*) *auto*
also have $\ldots =$ *real n* **by** *simp*
finally have $(nat\ \lfloor sqrt\ n\rfloor)^2 \leq n$
by (*simp only*: *of-nat-le-iff*)
moreover have $n < (Suc\ (nat\ \lfloor sqrt\ n\rfloor))^2$ **proof** −
have $(1 + \lfloor sqrt\ n\rfloor)^2 > n$
using *floor-correct*[*of sqrt n*] *real-le-rsqrt*[*of 1* + $\lfloor$*sqrt n*$\rfloor$ *n*]
of-int-less-iff[*of n* $(1 + \lfloor sqrt\ n\rfloor)^2$] *not-le*
by *fastforce*
then show *?thesis*
using *le-nat-floor*[*of Suc* (*nat* $\lfloor$*sqrt n*$\rfloor$) *sqrt n*]
of-nat-le-iff[*of* $(Suc\ (nat\ \lfloor sqrt\ n\rfloor))^2$ *n*] *real-le-rsqrt*[*of* - *n*] *not-le*
by *fastforce*
qed
ultimately show *?thesis* **using** *floor-sqrt-unique* **by** *fast*
qed

fun *newton-sqrt-aux* :: *nat* ⇒ *nat* ⇒ *nat* **where**
newton-sqrt-aux x n =
(*let y* = (*x* + *n div x*) *div 2*
in if y < *x then newton-sqrt-aux y n else x*)

declare *newton-sqrt-aux.simps* [*simp del*]

lemma *newton-sqrt-aux-simps*:
(*x* + *n div x*) *div 2* < *x* ⟹ *newton-sqrt-aux x n* = *newton-sqrt-aux* ((*x* + *n div*

```
x) div 2) n
  (x + n div x) div 2 ≥ x ⟹ newton-sqrt-aux x n = x
  by (subst newton-sqrt-aux.simps; simp add: Let-def)+

lemma heron-step-real: ⟦t > 0; n ≥ 0⟧ ⟹ (t + n/t) / 2 ≥ sqrt n
  using arith-geo-mean-sqrt[of t n/t] by simp

lemma heron-step-div-eq-floored:
  (t::nat) > 0 ⟹ (t + (n::nat) div t) div 2 = nat ⌊(t + n/t) / 2⌋
proof −
  assume t > 0
  then have ⌊(t + n/t) / 2⌋ = ⌊(t*t + n) / (2*t)⌋
    by (simp add: mult-divide-mult-cancel-right[of t t + n/t 2, symmetric]
        algebra-simps)
  also have ... = (t*t + n) div (2*t)
    using floor-divide-of-nat-eq by blast
  also have ... = (t*t + n) div t div 2
    by (simp add: div-mult2-eq ac-simps)
  also have ... = (t + n div t) div 2
    by (simp add: ‹0 < t› power2-eq-square)
  finally show ?thesis by simp
qed

lemma heron-step: t > 0 ⟹ (t + n div t) div 2 ≥ floor-sqrt n
proof −
  assume t > 0
  have floor-sqrt n = nat ⌊sqrt n⌋ by (rule sqrt-eq-floor-sqrt)
  also have ... ≤ nat ⌊(t + n/t) / 2⌋
    using heron-step-real[of t n] ‹t > 0› by linarith
  also have ... = (t + n div t) div 2
    using heron-step-div-eq-floored[OF ‹t > 0›] by simp
  finally show ?thesis .
qed

lemma newton-sqrt-aux-correct:
  assumes x ≥ floor-sqrt n
  shows   newton-sqrt-aux x n = floor-sqrt n
  using assms
proof (induction x n rule: newton-sqrt-aux.induct)
  case (1 x n)
  show ?case
  proof (cases x = floor-sqrt n)
    case True
    then have (x ^ 2) div x ≤ n div x by (intro div-le-mono) simp-all
    also have (x ^ 2) div x = x by (simp add: power2-eq-square)
    finally have (x + n div x) div 2 ≥ x by linarith
    with True show ?thesis by (auto simp: newton-sqrt-aux-simps)
  next
    case False
```

with *1.prems* **have** *x-gt-sqrt*: $x > \mathit{floor\text{-}sqrt}\ n$ **by** *auto*
with *le-floor-sqrt-iff*[*of x n*] **have** $n < x\ \hat{}\ 2$ **by** *simp*
have $x * (n\ div\ x) \leq n$ **using** *mult-div-mod-eq*[*of x n*] **by** *linarith*
also have $\ldots < x\ \hat{}\ 2$ **using** *le-floor-sqrt-iff*[*of x n*] **and** *x-gt-sqrt* **by** *simp*
also have $\ldots = x * x$ **by** (*simp add*: *power2-eq-square*)
finally have $n\ div\ x < x$ **by** (*subst* (*asm*) *mult-less-cancel1*) *auto*
then have *step-decreasing*: $(x + n\ div\ x)\ div\ 2 < x$ **by** *linarith*
with *x-gt-sqrt* **have** *step-ge-sqrt*: $(x + n\ div\ x)\ div\ 2 \geq \mathit{floor\text{-}sqrt}\ n$
by (*simp add*: *heron-step*)
from *step-decreasing* **have** *newton-sqrt-aux x n* = *newton-sqrt-aux* $((x + n\ div\ x)\ div\ 2)\ n$
by (*simp add*: *newton-sqrt-aux-simps*)
also have $\ldots =$ *floor-sqrt n*
by (*intro 1.IH step-decreasing step-ge-sqrt*) *simp-all*
finally show *?thesis* **.**
qed
qed

definition *newton-sqrt* :: *nat* $\Rightarrow$ *nat* **where**
newton-sqrt n = *newton-sqrt-aux n n*

declare *floor-sqrt-code* [*code del*]

theorem *Discrete-sqrt-eq-newton-sqrt* [*code*]: *floor-sqrt n* = *newton-sqrt n*
unfolding *newton-sqrt-def* **by** (*simp add*: *newton-sqrt-aux-correct floor-sqrt-le*)

1.2 Square Testing

Next, we implement an algorithm to determine whether a given natural number is a perfect square, as described by Cohen [2]. Essentially, the number first determines whether the number is a square. Essentially

definition *q11* :: *nat set*
where $q11 = \{0, 1, 3, 4, 5, 9\}$
definition *q63* :: *nat set*
where $q63 = \{0, 1, 4, 7, 9, 16, 28, 18, 22, 25, 36, 58, 46, 49, 37, 43\}$
definition *q64* :: *nat set*
where $q64 = \{0, 1, 4, 9, 16, 17, 25, 36, 33, 49, 41, 57\}$
definition *q65* :: *nat set*
where $q65 = \{0, 1, 4, 10, 14, 9, 16, 26, 30, 25, 29, 40, 56, 36, 49, 61, 35, 51, 39, 55, 64\}$

definition *q11-array* **where**
q11-array = *IArray* [*True,True,False,True,True,True,False,False,False,True,False*]

definition *q63-array* **where**
q63-array = *IArray* [*True,True,False,False,True,False,False,True,False,True,False,False,
False,False,False,False,True,False,True,False,False,False,True,False,False,True,False,
False,True,False,False,False,False,False,False,False,True,True,False,False,False,False,*

False, *True*, *False*, *False*, *True*, *False*, *False*, *True*, *False*, *False*, *False*, *False*, *False*, *False*, *False*,
False, *True*, *False*, *False*, *False*, *False*, *False*]

definition *q64-array* **where**
q64-array = *IArray* [*True*, *True*, *False*, *False*, *True*, *False*, *False*, *False*, *False*, *True*, *False*, *False*,
False, *False*, *False*, *False*, *True*, *True*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *True*, *False*,
False, *False*, *False*, *False*, *False*, *False*, *True*, *False*, *False*, *True*, *False*, *False*, *False*, *False*, *True*,
False, *False*, *False*, *False*, *False*, *False*, *False*, *True*, *False*, *False*, *False*, *False*, *False*, *False*,
False, *True*, *False*, *False*, *False*, *False*, *False*, *False*, *False*]

definition *q65-array* **where**
q65-array = *IArray* [*True*, *True*, *False*, *False*, *True*, *False*, *False*, *False*, *False*, *True*, *True*, *False*,
False, *False*, *True*, *False*, *True*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *False*, *True*, *True*,
False, *False*, *True*, *True*, *False*, *False*, *False*, *False*, *True*, *True*, *False*, *False*, *True*, *True*, *False*,
False, *False*, *False*, *False*, *False*, *False*, *False*, *True*, *False*, *True*, *False*, *False*, *False*, *True*, *True*
, *False*, *False*, *False*, *False*, *True*, *False*, *False*, *True*, *False*]

lemma *sub-q11-array*: $i \in \{..<11\} \Longrightarrow$ *IArray.sub q11-array i* $\longleftrightarrow i \in$ *q11*
by (*simp add*: *lessThan-nat-numeral lessThan-Suc q11-def q11-array-def*, *elim disjE*; *simp*)

lemma *sub-q63-array*: $i \in \{..<63\} \Longrightarrow$ *IArray.sub q63-array i* $\longleftrightarrow i \in$ *q63*
by (*simp add*: *lessThan-nat-numeral lessThan-Suc q63-def q63-array-def*, *elim disjE*; *simp*)

lemma *sub-q64-array*: $i \in \{..<64\} \Longrightarrow$ *IArray.sub q64-array i* $\longleftrightarrow i \in$ *q64*
by (*simp add*: *lessThan-nat-numeral lessThan-Suc q64-def q64-array-def*, *elim disjE*; *simp*)

lemma *sub-q65-array*: $i \in \{..<65\} \Longrightarrow$ *IArray.sub q65-array i* $\longleftrightarrow i \in$ *q65*
by (*simp add*: *lessThan-nat-numeral lessThan-Suc q65-def q65-array-def*, *elim disjE*; *simp*)

lemma *in-q11-code*: *x mod 11* $\in$ *q11* $\longleftrightarrow$ *IArray.sub q11-array* (*x mod 11*)
by (*subst sub-q11-array*) *auto*

lemma *in-q63-code*: *x mod 63* $\in$ *q63* $\longleftrightarrow$ *IArray.sub q63-array* (*x mod 63*)
by (*subst sub-q63-array*) *auto*

lemma *in-q64-code*: *x mod 64* $\in$ *q64* $\longleftrightarrow$ *IArray.sub q64-array* (*x mod 64*)
by (*subst sub-q64-array*) *auto*

lemma *in-q65-code*: *x mod 65* $\in$ *q65* $\longleftrightarrow$ *IArray.sub q65-array* (*x mod 65*)
by (*subst sub-q65-array*) *auto*

definition *square-test* :: *nat* $\Rightarrow$ *bool* **where**
square-test n =

$(n\ mod\ 64 \in q64 \wedge (let\ r = n\ mod\ 45045\ in$
$r\ mod\ 63 \in q63 \wedge r\ mod\ 65 \in q65 \wedge r\ mod\ 11 \in q11 \wedge n = (floor\text{-}sqrt\ n)^2))$

lemma *square-test-code* [*code*]:
square-test $n =$
$(IArray.sub\ q64\text{-}array\ (n\ mod\ 64) \wedge (let\ r = n\ mod\ 45045\ in$
$IArray.sub\ q63\text{-}array\ (r\ mod\ 63) \wedge$
$IArray.sub\ q65\text{-}array\ (r\ mod\ 65) \wedge$
$IArray.sub\ q11\text{-}array\ (r\ mod\ 11) \wedge n = (floor\text{-}sqrt\ n)^2))$
using *in-q11-code* [*symmetric*] *in-q63-code* [*symmetric*]
in-q64-code [*symmetric*] *in-q65-code* [*symmetric*]
by (*simp add*: *Let-def square-test-def*)

lemma *square-mod-lower*: $m > 0 \Longrightarrow (q^2 :: nat)\ mod\ m = a \Longrightarrow \exists\, q' < m.\ q'^2\ mod\ m = a$
using *mod-less-divisor mod-mod-trivial power-mod* **by** *blast*

lemma *q11-upto-def*: $q11 = (\lambda k.\ k^2\ mod\ 11)$ ‘ $\{..<11\}$
by (*simp add*: *q11-def lessThan-nat-numeral lessThan-Suc insert-commute*)

lemma *q11-infinite-def*: $q11 = (\lambda k.\ k^2\ mod\ 11)$ ‘ $\{0..\}$
unfolding *q11-upto-def image-def* **proof** (*auto*, *goal-cases*)
case (*1 xa*)
show *?case*
using *square-mod-lower*[*of 11 xa* xa^2 *mod 11*]
ex-nat-less-eq[*of 11* $\lambda x.\ xa^2\ mod\ 11 = x^2\ mod\ 11$]
by *auto*
qed

lemma *q63-upto-def*: $q63 = (\lambda k.\ k^2\ mod\ 63)$ ‘ $\{..<63\}$
by (*simp add*: *q63-def lessThan-nat-numeral lessThan-Suc insert-commute*)

lemma *q63-infinite-def*: $q63 = (\lambda k.\ k^2\ mod\ 63)$ ‘ $\{0..\}$
unfolding *q63-upto-def image-def* **proof** (*auto*, *goal-cases*)
case (*1 xa*)
show *?case*
using *square-mod-lower*[*of 63 xa* xa^2 *mod 63*]
ex-nat-less-eq[*of 63* $\lambda x.\ xa^2\ mod\ 63 = x^2\ mod\ 63$]
by *auto*
qed

lemma *q64-upto-def*: $q64 = (\lambda k.\ k^2\ mod\ 64)$ ‘ $\{..<64\}$
by (*simp add*: *q64-def lessThan-nat-numeral lessThan-Suc insert-commute*)

lemma *q64-infinite-def*: $q64 = (\lambda k.\ k^2\ mod\ 64)$ ‘ $\{0..\}$
unfolding *q64-upto-def image-def* **proof** (*auto*, *goal-cases*)
case (*1 xa*)
show *?case*
using *square-mod-lower*[*of 64 xa* xa^2 *mod 64*]

ex-nat-less-eq[*of 64* $\lambda x.\ xa^2$ *mod 64* = x^2 *mod 64*]
by *auto*
qed

lemma *q65-upto-def*: *q65* = ($\lambda k.\ k^2$ *mod 65*) ' {..<*65*}
by (*simp add*: *q65-def lessThan-nat-numeral lessThan-Suc insert-commute*)

lemma *q65-infinite-def*: *q65* = ($\lambda k.\ k^2$ *mod 65*) ' {*0*..}
unfolding *q65-upto-def image-def* **proof** (*auto*, *goal-cases*)
case (*1 xa*)
show *?case*
using *square-mod-lower*[*of 65 xa* xa^2 *mod 65*]
ex-nat-less-eq[*of 65* $\lambda x.\ xa^2$ *mod 65* = x^2 *mod 65*]
by *auto*
qed

lemma *square-mod-existence*:
fixes *n k* :: *nat*
assumes $\exists q.\ q^2 = n$
shows $\exists q.$ *n mod k* = q^2 *mod k*
using *assms* **by** *auto*

theorem *square-test-correct*: *square-test n* $\longleftrightarrow$ *is-square n*
proof *cases*
assume *is-square n*
hence *rhs*: $\exists q.\ q^2 = n$ **by** (*auto elim*: *is-nth-powerE*)
note *sq-mod* = *square-mod-existence*[*OF this*]
have *q64-member*: *n mod 64* $\in$ *q64* **using** *sq-mod*[*of 64*]
unfolding *q64-infinite-def image-def* **by** *simp*
let *?r* = *n mod 45045*
have *11 dvd* (*45045*::*nat*) *63 dvd* (*45045*::*nat*) *65 dvd* (*45045*::*nat*) **by** *force+*
then have *mod-45045*: *?r mod 11* = *n mod 11* *?r mod 63* = *n mod 63* *?r mod 65* = *n mod 65*
using *mod-mod-cancel*[*of - 45045 n*] **by** *presburger+*
then have *?r mod 11* $\in$ *q11* *?r mod 63* $\in$ *q63* *?r mod 65* $\in$ *q65*
using *sq-mod*[*of 11*] *sq-mod*[*of 63*] *sq-mod*[*of 65*]
unfolding *q11-infinite-def q63-infinite-def q65-infinite-def image-def mod-45045*
by *fast+*
then show *?thesis* **unfolding** *square-test-def Let-def* **using** *q64-member rhs* **by** *auto*
next
assume *not-rhs*: $\neg$*is-square n*
hence $\nexists q.\ q^2 = n$ **by** *auto*
then have (*floor-sqrt n*)$^2 \neq n$ **by** *simp*
then show *?thesis* **unfolding** *square-test-def* **by** (*auto simp*: *is-nth-power-def*)
qed

definition *get-nat-sqrt* :: *nat* $\Rightarrow$ *nat option*

where *get-nat-sqrt n* = (*if is-square n then Some* (*floor-sqrt n*) *else None*)

lemma *get-nat-sqrt-code* [*code*]:
get-nat-sqrt n =
(*if IArray.sub q64-array* (*n mod 64*) ∧ (*let r* = *n mod 45045 in*
IArray.sub q63-array (*r mod 63*) ∧
IArray.sub q65-array (*r mod 65*) ∧
IArray.sub q11-array (*r mod 11*)) *then*
(*let x* = *floor-sqrt n in if* x^2 = *n then Some x else None*) *else None*)
unfolding *get-nat-sqrt-def square-test-correct* [*symmetric*] *square-test-def*
using *in-q11-code* [*symmetric*] *in-q63-code* [*symmetric*]
in-q64-code [*symmetric*] *in-q65-code* [*symmetric*]
by (*auto split*: *if-splits simp*: *Let-def*)

end

2 Pell's equation

theory *Pell*
imports
Complex-Main
HOL−Computational-Algebra.Computational-Algebra
begin

Pell's equation has the general form $x^2 = 1 + Dy^2$ where $D \in \mathbb{N}$ is a parameter and x, y are $\mathbb{Z}$-valued variables. As we will see, that case where D is a perfect square is trivial and therefore uninteresting; we will therefore assume that D is not a perfect square for the most part.

Furthermore, it is obvious that the solutions to Pell's equation are symmetric around the origin in the sense that (x, y) is a solution iff $(\pm x, \pm y)$ is a solution. We will therefore mostly look at solutions (x, y) where both x and y are non-negative, since the remaining solutions are a trivial consequence of these.

Information on the material treated in this formalisation can be found in many textbooks and lecture notes, e. g. [3, 1].

2.1 Preliminary facts

lemma *gcd-int-nonpos-iff* [*simp*]: *gcd x* (*y* :: *int*) ≤ *0* ⟷ *x* = *0* ∧ *y* = *0*
proof
assume *gcd x y* ≤ *0*
with *gcd-ge-0-int*[*of x y*] **have** *gcd x y* = *0* **by** *linarith*
thus *x* = *0* ∧ *y* = *0* **by** *auto*
qed *auto*

lemma *minus-in-Ints-iff* [*simp*]:
$-x \in \mathbb{Z} \longleftrightarrow x \in \mathbb{Z}$

using *Ints-minus*[*of* x] *Ints-minus*[*of* $-x$] **by** *auto*

A (positive) square root of a natural number is either a natural number or irrational.

lemma *nonneg-sqrt-nat-or-irrat*:
  assumes $x \mathbin{\char`\^} 2 = \mathit{real}\ a$ **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 ^ *2*) ∗ *real* a = *real* (p ^ *2*)
    by (*simp add*: *field-simps*)
  also have *real* (q ^ *2*) ∗ *real* a = *real* (q ^ *2* ∗ a)
    by *simp*
  finally have p ^ *2* = q ^ *2* ∗ a
    by (*subst* (*asm*) *of-nat-eq-iff*) *auto*
  hence q ^ *2* *dvd* p ^ *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 ^ *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*)
qed

corollary *sqrt-nat-or-irrat'*:
  sqrt (*real a*) ∈ ℕ ∨ *sqrt* (*real a*) ∉ 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*) ∈ ℕ ⟷ *is-square n*
proof
  assume *sqrt* (*real n*) ∈ 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* ⟹ *sqrt* (*real n*) ∉ ℚ
  using *sqrt-nat-or-irrat'*[*of n*] **by** (*auto simp*: *sqrt-nat-iff-is-square*)

2.2 The case of a perfect square

As we have noted, the case where D is a perfect square is trivial: In fact, we will show that the only solutions in this case are the trivial solutions $(x, y) = (\pm 1, 0)$ if D is a non-zero perfect square, or $(\pm 1, y)$ for arbitrary $y \in \mathbb{Z}$ if $D = 0$.

context
  fixes *D* :: *nat*
  assumes *square-D*: *is-square D*
begin

lemma *pell-square-solution-nat-aux*:
  fixes *x y* :: *nat*
  assumes *D* > *0* **and** *x* ^ *2* = *1* + *D* ∗ *y* ^ *2*
  shows (*x*, *y*) = (*1*, *0*)
proof −
  from *assms* **have** *x-nz*: *x* > *0* **by** (*auto intro*!: *Nat.gr0I*)
  from *square-D* **obtain** *d* **where** [*simp*]: $D = d^2$
    by (*auto elim*: *is-nth-powerE*)
  have *int x* ^ *2* = *int* (*x* ^ *2*) **by** *simp*
  also note *assms*(*2*)
  also have *int* (*1* + *D* ∗ *y* ^ *2*) = *1* + *int D* ∗ *int y* ^ *2* **by** *simp*
  finally have (*int x* + *int d* ∗ *int y*) ∗ (*int x* − *int d* ∗ *int y*) = *1*
    by (*simp add*: *algebra-simps power2-eq-square*)
  hence ∗: *int x* + *int d* ∗ *int y* = *1* ∧ *int x* − *int d* ∗ *int y* = *1*
    using *x-nz* **by** (*subst* (*asm*) *pos-zmult-eq-1-iff*) (*auto intro*: *add-pos-nonneg*)
  from ∗ **have** [*simp*]: *x* = *1* **by** *simp*

moreover from $*$ **and** *assms*(*1*) **have** $y = 0$ **by** *auto*
ultimately show *?thesis* **by** *simp*
qed

lemma *pell-square-solution-int-aux*:
fixes $x\ y$ :: *int*
assumes $D > 0$ **and** $x \mathbin{\hat{}} 2 = 1 + D * y \mathbin{\hat{}} 2$
shows $x \in \{-1, 1\} \wedge y = 0$
proof $-$
define $x'\ y'$ **where** $x' = nat\ |x|$ **and** $y' = nat\ |y|$
have x: $x = sgn\ x * x'$ **and** y: $y = sgn\ y * y'$
by (*auto simp*: *sgn-if* x'*-def* y'*-def*)
have *zero-iff*: $x = 0 \longleftrightarrow x' = 0\ y = 0 \longleftrightarrow y' = 0$
by (*auto simp*: x'*-def* y'*-def*)
note *assms*(*2*)
also have $x \mathbin{\hat{}} 2 = int\ (x' \mathbin{\hat{}} 2)$
by (*subst x*) (*auto simp*: *sgn-if zero-iff*)
also have $1 + D * y \mathbin{\hat{}} 2 = int\ (1 + D * y' \mathbin{\hat{}} 2)$
by (*subst y*) (*auto simp*: *sgn-if zero-iff*)
also note *of-nat-eq-iff*
finally have $x'^2 = 1 + D * y'^2$ **.**
from ‹$D > 0$› **and** *this* **have** $(x', y') = (1, 0)$
by (*rule pell-square-solution-nat-aux*)
thus *?thesis* **by** (*auto simp*: x'*-def* y'*-def*)
qed

lemma *pell-square-solution-nat-iff*:
fixes $x\ y$ :: *nat*
shows $x \mathbin{\hat{}} 2 = 1 + D * y \mathbin{\hat{}} 2 \longleftrightarrow x = 1 \wedge (D = 0 \vee y = 0)$
using *pell-square-solution-nat-aux*[*of x y*] **by** (*cases* $D = 0$) *auto*

lemma *pell-square-solution-int-iff*:
fixes $x\ y$ :: *int*
shows $x \mathbin{\hat{}} 2 = 1 + D * y \mathbin{\hat{}} 2 \longleftrightarrow x \in \{-1, 1\} \wedge (D = 0 \vee y = 0)$
using *pell-square-solution-int-aux*[*of x y*] **by** (*cases* $D = 0$) (*auto simp*: *power2-eq-1-iff*)

end

2.3 Existence of a non-trivial solution

Let us now turn to the case where D is not a perfect square.

We first show that Pell's equation always has at least one non-trivial solution (apart from the trivial solution $(1, 0)$). For this, we first need a lemma about the existence of rational approximations of real numbers.

The following lemma states that for any positive integer s and real number x, we can find a rational approximation t / u to x with an error of most $1 / (u * s)$ where the denominator u is at most s.

lemma *pell-approximation-lemma*:

fixes s :: *nat* **and** x :: *real*
assumes *s*: $s > 0$
shows $\exists\, u{::}nat.\ \exists\, t{::}int.\ u > 0 \wedge \mathit{coprime}\ u\ t \wedge 1\ /\ s \in \{|t - u * x|{<}..1\ /\ u\}$
proof −
define f **where** $f = (\lambda u.\ \lceil u * x \rceil)$
define g :: *nat* ⇒ *int* **where** $g = (\lambda u.\ \lfloor \mathit{frac}\ (u * x) * s \rfloor)$
{
fix u :: *nat* **assume** *u*: $u \le s$
hence $\mathit{frac}\ (u * x) * \mathit{real}\ s < 1 * \mathit{real}\ s$
using *s* **by** (*intro mult-strict-right-mono*) (*auto simp*: *frac-lt-1*)
hence $g\ u < \mathit{int}\ s$ **by** (*auto simp*: *floor-less-iff g-def*)
}
hence g ‘ $\{..s\} \subseteq \{0..{<}s\}$
by (*auto simp*: *g-def floor-less-iff*)
hence $\mathit{card}\ (g$ ‘ $\{..s\}) \le \mathit{card}\ \{0..{<}\mathit{int}\ s\}$
by (*intro card-mono*) *auto*
also have $\ldots < \mathit{card}\ \{..s\}$ **by** *simp*
finally have ¬*inj-on* g $\{..s\}$ **by** (*rule pigeonhole*)
then obtain a b **where** *ab*: $a \le s$ $b \le s$ $a \ne b$ $g\ a = g\ b$
by (*auto simp*: *inj-on-def*)
define *u1* **and** *u2* **where** $u1 = \mathit{max}\ a\ b$ **and** $u2 = \mathit{min}\ a\ b$
have *u12*: $u1 \le s$ $u2 \le s$ $u2 < u1$ $g\ u1 = g\ u2$
using *ab* **by** (*auto simp*: *u1-def u2-def*)

define u t **where** $u = u1 - u2$ **and** $t = \lfloor u1 * x \rfloor - \lfloor u2 * x \rfloor$
have *u*: $u > 0$ $|u| \le s$
using *u12* **by** (*simp-all add*: *u-def*)

from ‹$g\ u1 = g\ u2$› **have** $|\mathit{frac}\ (u2 * x) * s - \mathit{frac}\ (u1 * x) * s| < 1$
unfolding *g-def* **by** *linarith*
also have $|\mathit{frac}\ (u2 * x) * s - \mathit{frac}\ (u1 * x) * s| =$
$|\mathit{real}\ s| * |\mathit{frac}\ (u2 * x) - \mathit{frac}\ (u1 * x)|$
by (*subst abs-mult* [*symmetric*]) (*simp add*: *algebra-simps*)
finally have $|t - u * x| * s < 1$ **using** ‹$u1 > u2$›
by (*simp add*: *g-def u-def t-def frac-def algebra-simps of-nat-diff*)
with ‹$s > 0$› **have** *less*: $|t - u * x| < 1\ /\ s$ **by** (*simp add*: *divide-simps*)

define d **where** $d = \mathit{gcd}\ (\mathit{nat}\ |t|)\ u$
define t' :: *int* **and** u' :: *nat* **where** $t' = t\ \mathit{div}\ d$ **and** $u' = u\ \mathit{div}\ d$
from *u* **have** $d \ne 0$
by (*intro notI*) (*auto simp*: *d-def*)
have $\mathit{int}\ (\mathit{gcd}\ (\mathit{nat}\ |t|)\ u) = \mathit{gcd}\ |t|\ (\mathit{int}\ u)$
by *simp*
hence *t′-u′*: $t = t' * d$ $u = u' * d$
by (*auto simp*: *t′-def u′-def d-def nat-dvd-iff*)

from ‹$d \ne 0$› **have** $|t' - u' * x| * 1 \le |t' - u' * x| * |\mathit{real}\ d|$
by (*intro mult-left-mono*) *auto*
also have $\ldots = |t - u * x|$ **by** (*subst abs-mult* [*symmetric*]) (*simp add*: *alge-*

```
bra-simps t′-u′)
  also note less
  finally have |t′ − u′ ∗ x| < 1 / s by simp
  moreover {
    from ‹s > 0› and u have 1 / s ≤ 1 / u
      by (simp add: divide-simps u-def)
    also have . . . = 1 / u′ / d by (simp add: t′-u′ divide-simps)
    also have . . . ≤ 1 / u′ / 1 using ‹d ≠ 0› by (intro divide-left-mono) auto
    finally have 1 / s ≤ 1 / u′ by simp
  }
  ultimately have 1 / s ∈ {|t′ − u′ ∗ x|<..1 / u′} by auto
  moreover from ‹u > 0› have u′ > 0 by (auto simp: t′-u′)
  moreover {
    have gcd u t = gcd t′ u′ ∗ int d
      by (simp add: t′-u′ gcd-mult-right gcd.commute)
    also have int d = gcd u t
      by (simp add: d-def gcd.commute)
    finally have gcd u′ t′ = 1 using u by (simp add: gcd.commute)
  }
  ultimately show ?thesis by blast
qed
```

As a simple corollary of this, we can show that for irrational x, there is an infinite number of rational approximations t / u to x whose error is less that $1 / u^2$.

```
corollary pell-approximation-corollary:
  fixes x :: real
  assumes x ∉ ℚ
  shows infinite {(t :: int, u :: nat). u > 0 ∧ coprime u t ∧ |t − u ∗ x| < 1 / u}
    (is infinite ?A)
proof
  assume fin: finite ?A
  let ?f = λ(t :: int, u :: nat). |t − u ∗ x|
  from fin have fin′: finite (insert 1 (?f ‘ ?A)) by blast
  have Min (insert 1 (?f ‘ ?A)) > 0
  proof (subst Min-gr-iff)
    have a ≠ b ∗ x if b > 0 for a :: int and b :: nat
    proof
      assume a = b ∗ x
      with ‹b > 0› have x = a / b by (simp add: field-simps)
      with ‹x ∉ ℚ› and ‹b > 0› show False by (auto simp: Rats-eq-int-div-nat)
    qed
    thus ∀ x∈insert 1 (?f ‘ ?A). x > 0 by auto
  qed (insert fin′, simp-all)
  also note real-arch-inverse
  finally obtain M :: nat where M: M ≠ 0 inverse M < Min (insert 1 (?f ‘ ?A))
    by blast
  hence M > 0 by simp
```

from *pell-approximation-lemma*[*OF this*, *of x*] **obtain** $u :: nat$ **and** $t :: int$
  where *ut*: $u > 0$ *coprime u t* $1\ /\ real\ M \in \{?f\ (t, u){<}..1\ /\ u\}$ **by** *auto*
from *ut* **have** $?f\ (t, u) < 1\ /\ real\ M$ **by** *simp*
also from *M* **have** $\ldots\ < Min\ (insert\ 1\ (?f\ `\ ?A))$
  by (*simp add*: *divide-simps*)
also from *ut* **have** $Min\ (insert\ 1\ (?f\ `\ ?A)) \le ?f\ (t, u)$
  by (*intro Min.coboundedI fin*$'$) *auto*
finally show *False* **by** *simp*
qed

locale *pell* =
  fixes $D :: nat$
  assumes *nonsquare-D*: $\neg$*is-square D*
begin

lemma *D-gt-1*: $D > 1$
proof −
  from *nonsquare-D* **have** $D \neq 0$ $D \neq 1$ **by** (*auto intro*!: *Nat.gr0I*)
  thus *?thesis* **by** *simp*
qed

lemma *D-pos*: $D > 0$
  using *nonsquare-D* **by** (*intro Nat.gr0I*) *auto*

With the above corollary, we can show the existence of a non-trivial solution. We restrict our attention to solutions (x, y) where both x and y are non-negative.

theorem *pell-solution-exists*: $\exists (x::nat)\ (y::nat).\ y \neq 0 \wedge x^2 = 1 + D * y^2$
proof −
  define *S* **where** $S = \{(t :: int, u :: nat).\ u > 0 \wedge coprime\ u\ t \wedge |t - u * sqrt\ D| < 1\ /\ u\}$
  let $?f = \lambda(t :: int, u :: nat).\ t^2 - u^2 * D$
  define *M* **where** $M = \lfloor 1 + 2 * sqrt\ D \rfloor$
  have *infinite*: $\neg$*finite S* **unfolding** *S-def*
    by (*intro pell-approximation-corollary irrat-sqrt-nonsquare nonsquare-D*)

  have *subset*: $?f\ `\ S \subseteq \{-M..M\}$
  proof *safe*
    fix $u :: nat$ **and** $t :: int$
    assume *tu*: $(t, u) \in S$
    from *tu* **have** [*simp*]: $u > 0$ **by** (*auto simp*: *S-def*)
    have $|t + u * sqrt\ D| = |t - u * sqrt\ D + 2 * u * sqrt\ D|$ **by** *simp*
    also have $\ldots \le |t - u * sqrt\ D| + |2 * u * sqrt\ D|$
      by (*rule abs-triangle-ineq*)
    also have $|2 * u * sqrt\ D| = 2 * u * sqrt\ D$ **by** *simp*
    also have $|t - u * sqrt\ D| \le 1\ /\ u$
      using *tu* **by** (*simp add*: *S-def*)
    finally have *le*: $|t + u * sqrt\ D| \le 1\ /\ u + 2 * u * sqrt\ D$ **by** *simp*

have $|t^2 - u^2 * D| = |t - u * \mathit{sqrt}\ D| * |t + u * \mathit{sqrt}\ D|$
  by (*subst abs-mult* [*symmetric*]) (*simp add*: *algebra-simps power2-eq-square*)
also have $\ldots \le 1 / u * (1 / u + 2 * u * \mathit{sqrt}\ D)$
  using *tu* **by** (*intro mult-mono le*) (*auto simp*: *S-def*)
also have $\ldots = 1 / \mathit{real}\ u \,\hat{}\, 2 + 2 * \mathit{sqrt}\ D$
  by (*simp add*: *algebra-simps power2-eq-square*)
also from ‹$u > 0$› **have** $\mathit{real}\ u \ge 1$ **by** *linarith*
hence $1 / \mathit{real}\ u \,\hat{}\, 2 \le 1 / 1 \,\hat{}\, 2$
  by (*intro divide-left-mono power-mono*) *auto*
finally have $|t^2 - u^2 * D| \le 1 + 2 * \mathit{sqrt}\ D$ **by** *simp*
hence $t^2 - u^2 * D \ge -M$ $t^2 - u^2 * D \le M$ **unfolding** *M-def* **by** *linarith*+
thus $t^2 - u^2 * D \in \{-M..M\}$ **by** *simp*
qed
hence *fin*: *finite* (*?f* ‘ *S*) **by** (*rule finite-subset*) *auto*

from *pigeonhole-infinite*[*OF infinite fin*]
  obtain *z* **where** *z*: $z \in S$ *infinite* $\{z' \in S.\ ?f\ z' = ?f\ z\}$ **by** *blast*
define *k* **where** $k = ?f\ z$
with *subset* **and** *z* **have** *k*: $k \in \{-M..M\}$ *infinite* $\{z{\in}S.\ ?f\ z = k\}$
  by (*auto simp*: *k-def*)

have *k-nz*: $k \neq 0$
proof
  assume [*simp*]: $k = 0$
  note *k*(*2*)
  also have $?f\ z \neq 0$ **if** $z \in S$ **for** *z*
  proof
    assume $*$: $?f\ z = 0$
    obtain *t u* **where** [*simp*]: $z = (t, u)$ **by** (*cases z*)
    from $*$ **have** $t \,\hat{}\, 2 = \mathit{int}\ u \,\hat{}\, 2 * \mathit{int}\ D$ **by** *simp*
    hence $\mathit{int}\ u \,\hat{}\, 2\ \mathit{dvd}\ t \,\hat{}\, 2$ **by** *simp*
    hence $\mathit{int}\ u\ \mathit{dvd}\ t$ **by** *simp*
    then obtain *k* **where** [*simp*]: $t = \mathit{int}\ u * k$ **by** (*auto elim*!: *dvdE*)
    from $*$ **and** ‹$z \in S$› **have** $k \,\hat{}\, 2 = \mathit{int}\ D$
      by (*auto simp*: *power-mult-distrib S-def*)
    also have $k \,\hat{}\, 2 = \mathit{int}\ (\mathit{nat}\ |k| \,\hat{}\, 2)$ **by** *simp*
    finally have $D = \mathit{nat}\ |k| \,\hat{}\, 2$ **by** (*simp only*: *of-nat-eq-iff*)
    hence *is-square D* **by** *auto*
    with *nonsquare-D* **show** *False* **by** *contradiction*
  qed
  hence $\{z{\in}S.\ ?f\ z = k\} = \{\}$ **by** *auto*
  finally show *False* **by** *simp*
qed

let $?h = \lambda(t :: \mathit{int}, u :: \mathit{nat}).\ (t\ \mathit{mod}\ (\mathit{abs}\ k), u\ \mathit{mod}\ (\mathit{abs}\ k))$
have $?h$ ‘ $\{z{\in}S.\ ?f\ z = k\} \subseteq \{0..{<}\mathit{abs}\ k\} \times \{0..{<}\ \mathit{abs}\ k\}$
  using *k-nz* **by** (*auto simp*: *case-prod-unfold*)
hence *finite* (*?h* ‘ $\{z{\in}S.\ ?f\ z = k\}$) **by** (*rule finite-subset*) *auto*

from *pigeonhole-infinite*[*OF k*(*2*) *this*] **obtain** z'
  where z': $z' \in S$ *?f* $z' = k$ *infinite* $\{z''\in\{z\in S.\ ?f\ z = k\}.\ ?h\ z'' = ?h\ z'\}$
  by *blast*
define *l1* **and** *l2* **where** $l1 = fst\ (?h\ z')$ **and** $l2 = snd\ (?h\ z')$
define S' **where** $S' = \{(t,u) \in S.\ ?f\ (t,u) = k \wedge t\ mod\ abs\ k = l1 \wedge u\ mod\ abs\ k = l2\}$
note $z'(3)$
also have $\{z''\in\{z\in S.\ ?f\ z = k\}.\ ?h\ z'' = ?h\ z'\} = S'$
  by (*auto simp*: *l1-def l2-def case-prod-unfold* S'*-def*)
finally have *infinite*: *infinite* S' **.**

from $z'(1)$ **and** *k-nz* **have** *l12*: $l1 \in \{0..<abs\ k\}$ $l2 \in \{0..<abs\ k\}$
  by (*auto simp*: *l1-def l2-def case-prod-unfold*)

from *infinite-arbitrarily-large*[*OF infinite*]
obtain X **where** X: *finite* X *card* $X = 2$ $X \subseteq S'$ **by** *blast*
from *finite-distinct-list*[*OF this*(*1*)] **obtain** *xs* **where** *xs*: *set xs* $= X$ *distinct xs*
  by *blast*
with X **have** *length xs* $= 2$ **using** *distinct-card*[*of xs*] **by** *simp*
then obtain *z1 z2* **where** [*simp*]: $xs = [z1,\ z2]$
  by (*auto simp*: *length-Suc-conv eval-nat-numeral*)
from X *xs* **have** S': $z1 \in S'$ $z2 \in S'$ **and** *neq*: $z1 \neq z2$ **by** *auto*
define *t1 u1 t2 u2* **where** $t1 = fst\ z1$ **and** $u1 = snd\ z1$ **and** $t2 = fst\ z2$ **and** $u2 = snd\ z2$
have [*simp*]: $z1 = (t1,\ u1)$ $z2 = (t2,\ u2)$
  by (*simp-all add*: *t1-def u1-def t2-def u2-def*)

from S' **have** $*$ [*simp*]: $t1\ mod\ abs\ k = l1$ $t2\ mod\ abs\ k = l1$ $u1\ mod\ abs\ k = l2$ $u2\ mod\ abs\ k = l2$
  by (*simp-all add*: S'*-def*)
define x **where** $x = (t1 * t2 - D * u1 * u2)\ div\ k$
define y **where** $y = (t1 * u2 - t2 * u1)\ div\ k$

from S' **have** $(t1^2 - u1^2 * D) = k$ $(t2^2 - u2^2 * D) = k$
  by (*auto simp*: S'*-def*)
hence $(t1^2 - u1^2 * D) * (t2^2 - u2^2 * D) = k\ \hat{}\ 2$
  unfolding *power2-eq-square* **by** *simp*
also have $(t1^2 - u1^2 * D) * (t2^2 - u2^2 * D) =$
    $(t1 * t2 - D * u1 * u2)\ \hat{}\ 2 - D * (t1 * u2 - t2 * u1)\ \hat{}\ 2$
  by (*simp add*: *power2-eq-square algebra-simps*)
finally have *eq*: $(t1 * t2 - D * u1 * u2)^2 - D * (t1 * u2 - t2 * u1)^2 = k^2$ **.**

have $(t1 * u2 - t2 * u1)\ mod\ abs\ k = (l1 * l2 - l1 * l2)\ mod\ abs\ k$
  using *l12* **by** (*intro mod-diff-cong mod-mult-cong*) (*auto simp*: *mod-pos-pos-trivial*)
hence *dvd1*: $k\ dvd\ t1 * u2 - t2 * u1$ **by** (*simp add*: *mod-eq-0-iff-dvd*)

have $k^2\ dvd\ k^2 + D * (t1 * u2 - t2 * u1)^2$
  using *dvd1* **by** (*intro dvd-add*) *auto*
also from *eq* **have** $\ldots = (t1 * t2 - D * u1 * u2)^2$

by (*simp add*: *algebra-simps*)
finally have *dvd2*: $k\ dvd\ t1 * t2 - D * u1 * u2$
by *simp*

note *eq*
also from *dvd2* have $t1 * t2 - D * u1 * u2 = k * x$
by (*simp add*: *x-def*)
also from *dvd1* have $t1 * u2 - t2 * u1 = k * y$
by (*simp add*: *y-def*)
also have $(k * x)^2 - D * (k * y)^2 = k^2 * (x^2 - D * y^2)$
by (*simp add*: *power-mult-distrib algebra-simps*)
finally have *eq'*: $x^2 - D * y^2 = 1$
using *k-nz* by *simp*
hence $x^2 = 1 + D * y^2$ by *simp*
also have $x^2 = int\ (nat\ |x|\ \hat{}\ 2)$ by *simp*
also have $1 + D * y^2 = int\ (1 + D * nat\ |y|\ \hat{}\ 2)$ by *simp*
also note *of-nat-eq-iff*
finally have *eq''*: $(nat\ |x|)^2 = 1 + D * (nat\ |y|)^2$.

have $t1 * u2 \neq t2 * u1$
proof
assume $*$: $t1 * u2 = t2 * u1$
hence $|t1| * |u2| = |t2| * |u1|$ by (*simp only*: *abs-mult* [*symmetric*])
moreover from S' have *coprime u1 t1 coprime u2 t2*
by (*auto simp*: *S'-def S-def*)
ultimately have *eq*: $|t1| = |t2| \wedge u1 = u2$
by (*subst* (*asm*) *coprime-crossproduct-int*) (*auto simp*: *S'-def S-def gcd.commute coprime-commute*)
moreover from S' have $u1 \neq 0\ u2 \neq 0$ by (*auto simp*: *S'-def S-def*)
ultimately have $t1 = t2$ using $*$ by *auto*
with *eq* and *neq* show *False* by *auto*
qed
with *dvd1* have $y \neq 0$
by (*auto simp add*: *y-def dvd-div-eq-0-iff*)
hence $nat\ |y| \neq 0$ by *auto*
with *eq''* show $\exists x\ y.\ y \neq 0 \wedge x^2 = 1 + D * y^2$ by *blast*
qed

2.4 Definition of solutions

We define some abbreviations for the concepts of a solution and a non-trivial solution.

definition *solution* :: $('a \times 'a :: comm\text{-}semiring\text{-}1) \Rightarrow bool$ where
solution $= (\lambda(a, b).\ a^2 = 1 + of\text{-}nat\ D * b^2)$

definition *nontriv-solution* :: $('a \times 'a :: comm\text{-}semiring\text{-}1) \Rightarrow bool$ where
nontriv-solution $= (\lambda(a, b).\ (a, b) \neq (1, 0) \wedge a^2 = 1 + of\text{-}nat\ D * b^2)$

lemma *nontriv-solution-altdef*: *nontriv-solution* $z \longleftrightarrow$ *solution* $z \wedge z \neq (1, 0)$

by (*auto simp*: *solution-def nontriv-solution-def*)

lemma *solution-trivial-nat* [*simp*, *intro*]: *solution* (*Suc 0*, *0*)
by (*simp add*: *solution-def*)

lemma *solution-trivial* [*simp*, *intro*]: *solution* (*1*, *0*)
by (*simp add*: *solution-def*)

lemma *solution-uminus-left* [*simp*]: *solution* ($-x$, y :: $'a$ :: *comm-ring-1*) $\longleftrightarrow$ *solution* (x, y)
by (*simp add*: *solution-def*)

lemma *solution-uminus-right* [*simp*]: *solution* (x, $-y$ :: $'a$ :: *comm-ring-1*) $\longleftrightarrow$ *solution* (x, y)
by (*simp add*: *solution-def*)

lemma *solution-0-snd-nat-iff* [*simp*]: *solution* (a :: *nat*, *0*) $\longleftrightarrow$ $a = 1$
by (*auto simp*: *solution-def*)

lemma *solution-0-snd-iff* [*simp*]: *solution* (a :: $'a$ :: *idom*, *0*) $\longleftrightarrow$ $a \in \{1, -1\}$
by (*auto simp*: *solution-def power2-eq-1-iff*)

lemma *no-solution-0-fst-nat* [*simp*]: $\neg$*solution* (*0*, b :: *nat*)
by (*auto simp*: *solution-def*)

lemma *no-solution-0-fst-int* [*simp*]: $\neg$*solution* (*0*, b :: *int*)
proof −
have $1 + int\ D * b^2 > 0$ **by** (*intro add-pos-nonneg*) *auto*
thus *?thesis* **by** (*auto simp add*: *solution-def*)
qed

lemma *solution-of-nat-of-nat* [*simp*]:
solution (*of-nat a*, *of-nat b* :: $'a$ :: {*comm-ring-1*, *ring-char-0*}) $\longleftrightarrow$ *solution* (a, b)
by (*simp only*: *solution-def prod.case of-nat-power* [*symmetric*]
of-nat-1 [*symmetric*, **where** *?'a* = *'a*] *of-nat-add* [*symmetric*]
of-nat-mult [*symmetric*] *of-nat-eq-iff of-nat-id*)

lemma *solution-of-nat-of-nat'* [*simp*]:
solution (*case z of* (a, b) $\Rightarrow$ (*of-nat a*, *of-nat b* :: $'a$ :: {*comm-ring-1*, *ring-char-0*})) $\longleftrightarrow$
solution z
by (*auto simp*: *case-prod-unfold*)

lemma *solution-nat-abs-nat-abs* [*simp*]:
solution (*nat* $|x|$, *nat* $|y|$) $\longleftrightarrow$ *solution* (x, y)
proof −
define x' **and** y' **where** $x' = nat\ |x|$ **and** $y' = nat\ |y|$
have x: $x = sgn\ x * x'$ **and** y: $y = sgn\ y * y'$

by (*auto simp*: *x′-def y′-def sgn-if*)
have [*simp*]: $x = 0 \longleftrightarrow x' = 0$ $y = 0 \longleftrightarrow y' = 0$
by (*auto simp*: *x′-def y′-def*)
show *solution* $(x', y') \longleftrightarrow$ *solution* (x, y)
by (*subst x*, *subst y*) (*auto simp*: *sgn-if*)
qed

lemma *nontriv-solution-of-nat-of-nat* [*simp*]:
nontriv-solution (*of-nat a*, *of-nat b* :: *′a* :: {*comm-ring-1*, *ring-char-0*}) $\longleftrightarrow$ *nontriv-solution* (*a*, *b*)
by (*auto simp*: *nontriv-solution-altdef*)

lemma *nontriv-solution-of-nat-of-nat′* [*simp*]:
nontriv-solution (*case z of* (*a*, *b*) $\Rightarrow$ (*of-nat a*, *of-nat b* :: *′a* :: {*comm-ring-1*, *ring-char-0*})) $\longleftrightarrow$
nontriv-solution z
by (*auto simp*: *case-prod-unfold*)

lemma *nontriv-solution-imp-solution* [*dest*]: *nontriv-solution z* $\Longrightarrow$ *solution z*
by (*auto simp*: *nontriv-solution-altdef*)

2.5 The Pell valuation function

Solutions (x,y) have an interesting correspondence to the ring $\mathbb{Z}[\sqrt{D}]$ via the map $(x, y) \mapsto x + y\sqrt{D}$. We call this map the *Pell valuation function*. It is obvious that this map is injective, since $\sqrt{D}$ is irrational.

definition *pell-valuation* :: *int* × *int* ⇒ *real* **where**
pell-valuation = (λ(*a*,*b*). *a* + *b* ∗ *sqrt D*)

lemma *pell-valuation-nonneg* [*simp*]: *fst z* $\geq 0 \Longrightarrow$ *snd z* $\geq 0 \Longrightarrow$ *pell-valuation z* ≥ 0
by (*auto simp*: *pell-valuation-def case-prod-unfold*)

lemma *pell-valuation-uminus-uminus* [*simp*]: *pell-valuation* $(-x, -y) = -$*pell-valuation* (x, y)
by (*simp add*: *pell-valuation-def*)

lemma *pell-valuation-eq-iff* [*simp*]:
pell-valuation z1 = *pell-valuation z2* $\longleftrightarrow$ *z1* = *z2*
proof
assume ∗: *pell-valuation z1* = *pell-valuation z2*
obtain *a b* **where** [*simp*]: *z1* = (*a*, *b*) **by** (*cases z1*)
obtain *u v* **where** [*simp*]: *z2* = (*u*, *v*) **by** (*cases z2*)
have $b = v$
proof (*rule ccontr*)
assume $b \neq v$
with ∗ **have** *sqrt D* = (*u* − *a*) / (*b* − *v*)
by (*simp add*: *field-simps pell-valuation-def*)
also have ... $\in \mathbb{Q}$ **by** *auto*

finally show *False* **using** *irrat-sqrt-nonsquare nonsquare-D* **by** *blast*
qed
moreover from *this* **and** $*$ **have** $a = u$
by (*simp add*: *pell-valuation-def*)
ultimately show $z1 = z2$ **by** *simp*
qed *auto*

2.6 Linear ordering of solutions

Next, we show that solutions are linearly ordered w. r. t. the pointwise order on products. This means thatfor two different solutions (a, b) and (x, y), we always either have $a < x$ and $b < y$ or $a > x$ and $b > y$.

lemma *solutions-linorder*:
fixes *a b x y* :: *nat*
assumes *solution* (a, b) *solution* (x, y)
shows $a \leq x \wedge b \leq y \vee a \geq x \wedge b \geq y$
proof −
have $b \leq y$ **if** $a \leq x$ *solution* (a, b) *solution* (x, y) **for** *a b x y* :: *nat*
proof −
from *that* **have** $a \,\hat{}\, 2 \leq x \,\hat{}\, 2$ **by** (*intro power-mono*) *auto*
with *that* **and** *D-gt-1* **have** $b^2 \leq y^2$
by (*simp add*: *solution-def*)
thus $b \leq y$
by (*simp add*: *power2-nat-le-eq-le*)
qed
from *this*[*of a x b y*] **and** *this*[*of x a y b*] **and** *assms* **show** *?thesis*
by (*cases* $a \leq x$) *auto*
qed

lemma *solutions-linorder-strict*:
fixes *a b x y* :: *nat*
assumes *solution* (a, b) *solution* (x, y)
shows $(a, b) = (x, y) \vee a < x \wedge b < y \vee a > x \wedge b > y$
proof −
have $b = y$ **if** $a = x$
using *that assms* **and** *D-gt-1* **by** (*simp add*: *solution-def*)
moreover have $a = x$ **if** $b = y$
proof −
from *that* **and** *assms* **have** $a^2 = Suc\ (D * y^2)$
by (*simp add*: *solution-def*)
also from *that* **and** *assms* **have** $\ldots = x^2$
by (*simp add*: *solution-def*)
finally show $a = x$ **by** *simp*
qed
ultimately have [*simp*]: $a = x \longleftrightarrow b = y$ **..**
show *?thesis* **using** *solutions-linorder*[*OF assms*]
by (*cases a x rule*: *linorder-cases*; *cases b y rule*: *linorder-cases*) *simp-all*
qed

lemma *solutions-le-iff-pell-valuation-le*:
  fixes $a\ b\ x\ y :: nat$
  assumes *solution* $(a,\ b)$ *solution* $(x,\ y)$
  shows $a \le x \wedge b \le y \longleftrightarrow$ *pell-valuation* $(a,\ b) \le$ *pell-valuation* $(x,\ y)$
proof
  assume $a \le x \wedge b \le y$
  thus *pell-valuation* $(a,\ b) \le$ *pell-valuation* $(x,\ y)$
    unfolding *pell-valuation-def prod.case* **using** *D-gt-1*
    by (*intro add-mono mult-right-mono*) *auto*
next
  assume $*$: *pell-valuation* $(a,\ b) \le$ *pell-valuation* $(x,\ y)$
  from *assms* **have** $a \le x \wedge b \le y \vee x \le a \wedge y \le b$
    by (*rule solutions-linorder*)
  thus $a \le x \wedge b \le y$
  proof
    assume $x \le a \wedge y \le b$
    hence *pell-valuation* $(a,\ b) \ge$ *pell-valuation* $(x,\ y)$
      unfolding *pell-valuation-def prod.case* **using** *D-gt-1*
      by (*intro add-mono mult-right-mono*) *auto*
    with $*$ **have** *pell-valuation* $(a,\ b) =$ *pell-valuation* $(x,\ y)$ **by** *linarith*
    hence $(a,\ b) = (x,\ y)$ **by** *simp*
    thus $a \le x \wedge b \le y$ **by** *simp*
  qed *auto*
qed

lemma *solutions-less-iff-pell-valuation-less*:
  fixes $a\ b\ x\ y :: nat$
  assumes *solution* $(a,\ b)$ *solution* $(x,\ y)$
  shows $a < x \wedge b < y \longleftrightarrow$ *pell-valuation* $(a,\ b) <$ *pell-valuation* $(x,\ y)$
proof
  assume $a < x \wedge b < y$
  thus *pell-valuation* $(a,\ b) <$ *pell-valuation* $(x,\ y)$
    unfolding *pell-valuation-def prod.case* **using** *D-gt-1*
    by (*intro add-strict-mono mult-strict-right-mono*) *auto*
next
  assume $*$: *pell-valuation* $(a,\ b) <$ *pell-valuation* $(x,\ y)$
  from *assms* **have** $(a,\ b) = (x,\ y) \vee a < x \wedge b < y \vee x < a \wedge y < b$
    by (*rule solutions-linorder-strict*)
  thus $a < x \wedge b < y$
  proof (*elim disjE*)
    assume $x < a \wedge y < b$
    hence *pell-valuation* $(a,\ b) >$ *pell-valuation* $(x,\ y)$
      unfolding *pell-valuation-def prod.case* **using** *D-gt-1*
      by (*intro add-strict-mono mult-strict-right-mono*) *auto*
    with $*$ **have** *False* **by** *linarith*
    thus *?thesis* **..**
  qed (*insert* $*$, *auto*)
qed

2.7 The fundamental solution

The *fundamental solution* is the non-trivial solution (x, y) with non-negative x and y for which the Pell valuation $x + y\sqrt{D}$ is minimal, or, equivalently, for which x and y are minimal.

definition *fund-sol* :: *nat* × *nat* **where**
fund-sol = (*THE* *z*::*nat*×*nat*. *is-arg-min* (*pell-valuation* :: *nat* × *nat* ⇒ *real*) *nontriv-solution* *z*)

The well-definedness of this follows from the injectivity of the Pell valuation and the fact that smaller Pell valuation of a solution is smaller than that of another iff the components are both smaller.

theorem *fund-sol-is-arg-min*:
is-arg-min (*pell-valuation* :: *nat* × *nat* ⇒ *real*) *nontriv-solution* *fund-sol*
unfolding *fund-sol-def*
proof (*rule theI′*)
show ∃!*z*::*nat*×*nat*. *is-arg-min* (*pell-valuation* :: *nat* × *nat* ⇒ *real*) *nontriv-solution* *z*
proof (*rule ex-ex1I*)
fix *z1 z2* :: *nat* × *nat*
assume *is-arg-min* (*pell-valuation* :: *nat* × *nat* ⇒ *real*) *nontriv-solution z1*
is-arg-min (*pell-valuation* :: *nat* × *nat* ⇒ *real*) *nontriv-solution z2*
hence *pell-valuation z1* = *pell-valuation z2*
by (*cases z1*, *cases z2*, *intro antisym*) (*auto simp*: *is-arg-min-def not-less*)
thus *z1* = *z2* **by** (*auto split*: *prod.splits*)
next
define *y* **where** *y* = (*LEAST y*. *y* > *0* ∧ *is-square* (*1* + *D* ∗ y^2))
have ∃ *y*>*0*. *is-square* (*1* + *D* ∗ y^2)
using *pell-solution-exists* **by** (*auto simp*: *eq-commute*[*of* - *Suc* -])
hence *y*: *y* > *0* ∧ *is-square* (*1* + *D* ∗ y^2)
unfolding *y-def* **by** (*rule LeastI-ex*)
have *y-le*: *y* ≤ *y′* **if** *y′* > *0* *is-square* (*1* + *D* ∗ y'^2) **for** *y′*
unfolding *y-def* **using** *that* **by** (*intro Least-le*) *auto*
from *y* **obtain** *x* **where** *x*: x^2 = *1* + *D* ∗ y^2
by (*auto elim*: *is-nth-powerE*)
with *y* **have** *nontriv-solution* (*x*, *y*)
by (*auto simp*: *nontriv-solution-def*)

have *is-arg-min* (*pell-valuation* :: *nat* × *nat* ⇒ *real*) *nontriv-solution* (*x*, *y*)
unfolding *is-arg-min-linorder*
proof *safe*
fix *a b* :: *nat*
assume ∗: *nontriv-solution* (*a*, *b*)
hence *b* > *0* **and** *Suc* (*D* ∗ b^2) = a^2
by (*auto simp*: *nontriv-solution-def intro*!: *Nat.gr0I*)
hence *is-square* (*1* + *D* ∗ b^2)
by (*auto simp*: *nontriv-solution-def*)
from ‹*b* > *0*› **and** *this* **have** *y* ≤ *b* **by** (*rule y-le*)

with ‹*nontriv-solution* (x, y)› **and** ∗ **have** $x \leq a$
using *solutions-linorder-strict*[*of* x y a b] **by** (*auto simp*: *nontriv-solution-altdef*)
with ‹$y \leq b$› **show** *pell-valuation* (*int* x, *int* y) $\leq$ *pell-valuation* (*int* a, *int* b)
unfolding *pell-valuation-def prod.case* **by** (*intro add-mono mult-right-mono*) *auto*
qed *fact*+
thus $\exists z$. *is-arg-min* (*pell-valuation* :: *nat* × *nat* ⇒ *real*) *nontriv-solution* z **..**
qed
qed

corollary
fund-sol-is-nontriv-solution: *nontriv-solution fund-sol*
and *fund-sol-minimal*:
nontriv-solution (a, b) ⟹ *pell-valuation fund-sol* $\leq$ *pell-valuation* (*int* a, *int* b)
and *fund-sol-minimal*′:
nontriv-solution (z :: *nat* × *nat*) ⟹ *pell-valuation fund-sol* $\leq$ *pell-valuation* z
using *fund-sol-is-arg-min* **by** (*auto simp*: *is-arg-min-linorder case-prod-unfold*)

lemma *fund-sol-minimal*′′:
assumes *nontriv-solution* z
shows *fst fund-sol* $\leq$ *fst* z *snd fund-sol* $\leq$ *snd* z
proof −
have *pell-valuation* (*fst fund-sol*, *snd fund-sol*) $\leq$ *pell-valuation* (*fst* z, *snd* z)
using *fund-sol-minimal*′[*OF assms*] **by** (*simp add*: *case-prod-unfold*)
hence *fst fund-sol* $\leq$ *fst* z ∧ *snd fund-sol* $\leq$ *snd* z
using *assms fund-sol-is-nontriv-solution*
by (*subst solutions-le-iff-pell-valuation-le*) (*auto simp*: *case-prod-unfold*)
thus *fst fund-sol* $\leq$ *fst* z *snd fund-sol* $\leq$ *snd* z **by** *blast*+
qed

2.8 Group structure on solutions

As was mentioned already, the Pell valuation function provides an injective map from solutions of Pell's equation into the ring $\mathbb{Z}[\sqrt{D}]$. We shall see now that the solutions are actually a subgroup of the multiplicative group of $\mathbb{Z}[\sqrt{D}]$ via the valuation function as a homomorphism:

- The trivial solution $(1, 0)$ has valuation 1, which is the neutral element of $\mathbb{Z}[\sqrt{D}]^*$
- Multiplication of two solutions $a+b\sqrt{D}$ and $x+y\sqrt{D}$ leads to $\bar{x}+\bar{y}\sqrt{D}$ with $\bar{x} = xa + ybD$ and $\bar{y} = xb + ya$, which is again a solution.
- The conjugate $(x, -y)$ of a solution (x, y) is an inverse element to this multiplication operation, since $(x + y\sqrt{D})(x - y\sqrt{D}) = 1$.

definition *pell-mul* :: ($'a$ :: *comm-semiring-1* × $'a$) ⇒ ($'a$ × $'a$) ⇒ ($'a$ × $'a$) **where**
pell-mul = (λ(*a*,*b*) (*x*,*y*). (*x* ∗ *a* + *y* ∗ *b* ∗ *of-nat D*, *x* ∗ *b* + *y* ∗ *a*))

definition *pell-cnj* :: ($'a$ :: *comm-ring-1* × $'a$) ⇒ $'a$ × $'a$ **where**
pell-cnj = (λ(*a*,*b*). (*a*, −*b*))

lemma *pell-cnj-snd-0* [*simp*]: *snd z* = *0* ⟹ *pell-cnj z* = *z*
by (*cases z*) (*simp-all add*: *pell-cnj-def*)

lemma *pell-mul-commutes*: *pell-mul z1 z2* = *pell-mul z2 z1*
by (*auto simp*: *pell-mul-def algebra-simps case-prod-unfold*)

lemma *pell-mul-assoc*: *pell-mul z1* (*pell-mul z2 z3*) = *pell-mul* (*pell-mul z1 z2*) *z3*
by (*auto simp*: *pell-mul-def algebra-simps case-prod-unfold*)

lemma *pell-mul-trivial-left* [*simp*]: *pell-mul* (*1*, *0*) *z* = *z*
by (*auto simp*: *pell-mul-def algebra-simps case-prod-unfold*)

lemma *pell-mul-trivial-right* [*simp*]: *pell-mul z* (*1*, *0*) = *z*
by (*auto simp*: *pell-mul-def algebra-simps case-prod-unfold*)

lemma *pell-mul-trivial-left-nat* [*simp*]: *pell-mul* (*Suc 0*, *0*) *z* = *z*
by (*auto simp*: *pell-mul-def algebra-simps case-prod-unfold*)

lemma *pell-mul-trivial-right-nat* [*simp*]: *pell-mul z* (*Suc 0*, *0*) = *z*
by (*auto simp*: *pell-mul-def algebra-simps case-prod-unfold*)

definition *pell-power* :: ($'a$ :: *comm-semiring-1* × $'a$) ⇒ *nat* ⇒ ($'a$ × $'a$) **where**
pell-power z n = ((λ*z*′. *pell-mul z*′ *z*) ^^ *n*) (*1*, *0*)

lemma *pell-power-0* [*simp*]: *pell-power z 0* = (*1*, *0*)
by (*simp add*: *pell-power-def*)

lemma *pell-power-one* [*simp*]: *pell-power* (*1*, *0*) *n* = (*1*, *0*)
by (*induction n*) (*auto simp*: *pell-power-def*)

lemma *pell-power-one-right* [*simp*]: *pell-power z 1* = *z*
by (*simp add*: *pell-power-def*)

lemma *pell-power-Suc*: *pell-power z* (*Suc n*) = *pell-mul z* (*pell-power z n*)
by (*simp add*: *pell-power-def pell-mul-commutes*)

lemma *pell-power-add*: *pell-power z* (*m* + *n*) = *pell-mul* (*pell-power z m*) (*pell-power z n*)
by (*induction m arbitrary*: *z*)
(*simp-all add*: *funpow-add o-def pell-power-Suc pell-mul-assoc*)

lemma *pell-valuation-mult* [*simp*]:

pell-valuation (*pell-mul z1 z2*) = *pell-valuation z1* ∗ *pell-valuation z2*
by (*simp add*: *pell-valuation-def pell-mul-def case-prod-unfold algebra-simps*)

lemma *pell-valuation-mult-nat* [*simp*]:
pell-valuation (*case pell-mul z1 z2 of* (*a*, *b*) ⇒ (*int a*, *int b*)) =
pell-valuation z1 ∗ *pell-valuation z2*
by (*simp add*: *pell-valuation-def pell-mul-def case-prod-unfold algebra-simps*)

lemma *pell-valuation-trivial* [*simp*]: *pell-valuation* (*1*, *0*) = *1*
by (*simp add*: *pell-valuation-def*)

lemma *pell-valuation-trivial-nat* [*simp*]: *pell-valuation* (*Suc 0*, *0*) = *1*
by (*simp add*: *pell-valuation-def*)

lemma *pell-valuation-cnj*: *pell-valuation* (*pell-cnj z*) = *fst z* − *snd z* ∗ *sqrt D*
by (*simp add*: *pell-valuation-def pell-cnj-def case-prod-unfold*)

lemma *pell-valuation-snd-0* [*simp*]: *pell-valuation* (*a*, *0*) = *of-int a*
by (*simp add*: *pell-valuation-def*)

lemma *pell-valuation-0-iff* [*simp*]: *pell-valuation z* = *0* ⟷ *z* = (*0*, *0*)
proof
assume ∗: *pell-valuation z* = *0*
have *snd z* = *0*
proof (*rule ccontr*)
assume *snd z* ≠ *0*
with ∗ **have** *sqrt D* = −*fst z* / *snd z*
by (*simp add*: *pell-valuation-def case-prod-unfold field-simps*)
also have . . . ∈ ℚ **by** *auto*
finally show *False* **using** *nonsquare-D irrat-sqrt-nonsquare* **by** *blast*
qed
with ∗ **have** *fst z* = *0* **by** (*simp add*: *pell-valuation-def case-prod-unfold*)
with ‹*snd z* = *0*› **show** *z* = (*0*, *0*) **by** (*cases z*) *auto*
qed (*auto simp*: *pell-valuation-def*)

lemma *pell-valuation-solution-pos-nat*:
fixes *z* :: *nat* × *nat*
assumes *solution z*
shows *pell-valuation z* > *0*
proof −
from *assms* **have** *z* ≠ (*0*, *0*) **by** (*intro notI*) *auto*
hence *pell-valuation z* ≠ *0* **by** (*auto split*: *prod.splits*)
moreover have *pell-valuation z* ≥ *0* **by** (*intro pell-valuation-nonneg*) (*auto split*: *prod.splits*)
ultimately show *?thesis* **by** *linarith*
qed

lemma
assumes *solution z*

shows *pell-mul-cnj-right*: *pell-mul z* (*pell-cnj z*) = (*1*, *0*)
and *pell-mul-cnj-left*: *pell-mul* (*pell-cnj z*) *z* = (*1*, *0*)
using *assms* **by** (*auto simp*: *pell-mul-def pell-cnj-def solution-def power2-eq-square*)

lemma *pell-valuation-cnj-solution*:
fixes *z* :: *nat* × *nat*
assumes *solution z*
shows *pell-valuation* (*pell-cnj z*) = *1* / *pell-valuation z*
proof −
have *pell-valuation* (*pell-cnj z*) ∗ *pell-valuation z* = *pell-valuation* (*pell-mul* (*pell-cnj z*) *z*)
by *simp*
also from *assms* **have** *pell-mul* (*pell-cnj z*) *z* = (*1*, *0*)
by (*subst pell-mul-cnj-left*) (*auto simp*: *case-prod-unfold*)
finally show *?thesis* **using** *pell-valuation-solution-pos-nat*[*OF assms*]
by (*auto simp*: *divide-simps*)
qed

lemma *pell-valuation-power* [*simp*]: *pell-valuation* (*pell-power z n*) = *pell-valuation z* ^ *n*
by (*induction n*) (*simp-all add*: *pell-power-Suc*)

lemma *pell-valuation-power-nat* [*simp*]:
pell-valuation (*case pell-power z n of* (*a*, *b*) ⇒ (*int a*, *int b*)) = *pell-valuation z* ^ *n*
by (*induction n*) (*simp-all add*: *pell-power-Suc*)

lemma *pell-valuation-fund-sol-ge-2*: *pell-valuation fund-sol* ≥ *2*
proof −
obtain *x y* **where** [*simp*]: *fund-sol* = (*x*, *y*) **by** (*cases fund-sol*)
from *fund-sol-is-nontriv-solution* **have** *eq*: x^2 = *1* + *D* ∗ y^2
by (*auto simp*: *nontriv-solution-def*)

consider *y* > *0* | *y* = *0 x* ≠ *1*
using *fund-sol-is-nontriv-solution* **by** (*force simp*: *nontriv-solution-def*)
thus *?thesis*
proof *cases*
assume *y* > *0*
hence *1* + *1* ∗ *1* ≤ *1* + *D* ∗ y^2
using *D-pos* **by** (*intro add-mono mult-mono*) *auto*
also from *eq* **have** . . . = x^2 **..**
finally have x^2 > 1^2 **by** *simp*
hence *x* > *1* **by** (*rule power2-less-imp-less*) *auto*
with ‹*y* > *0*› **have** *x* + *y* ∗ *sqrt D* ≥ *2* + *1* ∗ *1*
using *D-pos* **by** (*intro add-mono mult-mono*) *auto*
thus *?thesis* **by** (*simp add*: *pell-valuation-def*)
next
assume [*simp*]: *y* = *0* **and** *x* ≠ *1*
with *eq* **have** *x* ≠ *0* **by** (*intro notI*) *auto*

with ‹$x \neq 1$› **have** $x \geq 2$ **by** *simp*
thus *?thesis* **by** (*auto simp*: *pell-valuation-def*)
qed
qed

lemma *solution-pell-mul* [*intro*]:
assumes *solution z1 solution z2*
shows *solution* (*pell-mul z1 z2*)
proof −
obtain *a b* **where** [*simp*]: *z1* = (*a*, *b*) **by** (*cases z1*)
obtain *c d* **where** [*simp*]: *z2* = (*c*, *d*) **by** (*cases z2*)
from *assms* **show** *?thesis*
by (*simp add*: *solution-def pell-mul-def case-prod-unfold power2-eq-square algebra-simps*)
qed

lemma *solution-pell-cnj* [*intro*]:
assumes *solution z*
shows *solution* (*pell-cnj z*)
using *assms* **by** (*auto simp*: *solution-def pell-cnj-def*)

lemma *solution-pell-power* [*simp*, *intro*]: *solution z* $\Longrightarrow$ *solution* (*pell-power z n*)
by (*induction n*) (*auto simp*: *pell-power-Suc*)

lemma *pell-mul-eq-trivial-nat-iff*:
pell-mul z1 z2 = (*Suc 0*, *0*) $\longleftrightarrow$ *z1* = (*Suc 0*, *0*) $\wedge$ *z2* = (*Suc 0*, *0*)
using *D-gt-1* **by** (*cases z1*; *cases z2*) (*auto simp*: *pell-mul-def*)

lemma *nontriv-solution-pell-nat-mul1*:
solution (*z1* :: *nat* × *nat*) $\Longrightarrow$ *nontriv-solution z2* $\Longrightarrow$ *nontriv-solution* (*pell-mul z1 z2*)
by (*auto simp*: *nontriv-solution-altdef pell-mul-eq-trivial-nat-iff*)

lemma *nontriv-solution-pell-nat-mul2*:
nontriv-solution (*z1* :: *nat* × *nat*) $\Longrightarrow$ *solution z2* $\Longrightarrow$ *nontriv-solution* (*pell-mul z1 z2*)
by (*auto simp*: *nontriv-solution-altdef pell-mul-eq-trivial-nat-iff*)

lemma *nontriv-solution-power-nat* [*intro*]:
assumes *nontriv-solution* (*z* :: *nat* × *nat*) *n* > *0*
shows *nontriv-solution* (*pell-power z n*)
proof −
have *nontriv-solution* (*pell-power z n*) $\vee$ *n* = *0*
by (*induction n*)
(*insert assms*(*1*), *auto intro*: *nontriv-solution-pell-nat-mul1 simp*: *pell-power-Suc*)
with *assms*(*2*) **show** *?thesis* **by** *auto*
qed

2.9 The different regions of the valuation function

Next, we shall explore what happens to the valuation function for solutions (x, y) for different signs of x and y:

- If $x > 0$ and $y > 0$, we have $x + y\sqrt{D} > 1$.
- If $x > 0$ and $y < 0$, we have $0 < x + y\sqrt{D} < 1$.
- If $x < 0$ and $y > 0$, we have $-1 < x + y\sqrt{D} < 0$.
- If $x < 0$ and $y < 0$, we have $x + y\sqrt{D} < -1$.

In particular, this means that we can deduce the sign of x and y if we know in which of these four regions the valuation lies.

lemma
  assumes *x > 0 y > 0 solution (x, y)*
  shows *pell-valuation-pos-pos*: *pell-valuation (x, y) > 1*
    and *pell-valuation-pos-neg-aux*: *pell-valuation (x, −y) ∈ {0<..<1}*
proof −
  from *D-gt-1 assms* **have** *x + y ∗ sqrt D ≥ 1 + 1 ∗ 1*
    by (*intro add-mono mult-mono*) *auto*
  hence *gt-1*: *x + y ∗ sqrt D > 1* **by** *simp*
  thus *pell-valuation (x, y) > 1* **by** (*simp add*: *pell-valuation-def*)

  from *assms* **have** *1 = x^2 − D ∗ y^2* **by** (*simp add*: *solution-def*)
  also have *of-int ... = (x − y ∗ sqrt D) ∗ (x + y ∗ sqrt D)*
    by (*simp add*: *field-simps power2-eq-square*)
  finally have *eq*: *(x − y ∗ sqrt D) = 1 / (x + y ∗ sqrt D)*
    using *gt-1* **by** (*simp add*: *field-simps*)

  note *eq*
  also from *gt-1* **have** *1 / (x + y ∗ sqrt D) < 1 / 1*
    by (*intro divide-strict-left-mono*) *auto*
  finally have *x − y ∗ sqrt D < 1* **by** *simp*

  note *eq*
  also from *gt-1* **have** *1 / (x + y ∗ sqrt D) > 0*
    by (*intro divide-pos-pos*) *auto*
  finally have *x − y ∗ sqrt D > 0* **.**
  with ‹*x − y ∗ sqrt D < 1*› **show** *pell-valuation (x, −y) ∈ {0<..<1}*
    by (*simp add*: *pell-valuation-def*)
qed

lemma *pell-valuation-pos-neg*:
  assumes *x > 0 y < 0 solution (x, y)*
  shows *pell-valuation (x, y) ∈ {0<..<1}*
  using *pell-valuation-pos-neg-aux[of x −y] assms* **by** *simp*

lemma *pell-valuation-neg-neg*:
  assumes $x < 0$ $y < 0$ *solution* (x, y)
  shows *pell-valuation* $(x, y) < -1$
  using *pell-valuation-pos-pos*[*of* $-x$ $-y$] *assms* **by** *simp*

lemma *pell-valuation-neg-pos*:
  assumes $x < 0$ $y > 0$ *solution* (x, y)
  shows *pell-valuation* $(x, y) \in \{-1<..<0\}$
  using *pell-valuation-pos-neg*[*of* $-x$ $-y$] *assms* **by** *simp*

lemma *pell-valuation-solution-gt1D*:
  assumes *solution z pell-valuation* $z > 1$
  shows *fst* $z > 0 \wedge$ *snd* $z > 0$
  using *pell-valuation-pos-pos*[*of fst z snd z*] *pell-valuation-pos-neg*[*of fst z snd z*]
    pell-valuation-neg-pos[*of fst z snd z*] *pell-valuation-neg-neg*[*of fst z snd z*]
    assms
  by (*cases fst z 0* :: *int rule*: *linorder-cases*;
    cases snd z 0 :: *int rule*: *linorder-cases*;
    cases z) *auto*

2.10 Generating property of the fundamental solution

We now show that the fundamental solution generates the set of the (non-negative) solutions in the sense that each solution is a power of the fundamental solution. Combined with the symmetry property that (x,y) is a solution iff $(\pm x, \pm y)$ is a solution, this gives us a complete characterisation of all solutions of Pell's equation.

definition *nth-solution* :: *nat* $\Rightarrow$ *nat* $\times$ *nat* **where**
  nth-solution n = *pell-power fund-sol n*

lemma *pell-valuation-nth-solution* [*simp*]:
  pell-valuation (*nth-solution n*) = *pell-valuation fund-sol* ^ *n*
  by (*simp add*: *nth-solution-def*)

theorem *nth-solution-inj*: *inj nth-solution*
proof
  fix *m n* :: *nat*
  assume *nth-solution m* = *nth-solution n*
  hence *pell-valuation* (*nth-solution m*) = *pell-valuation* (*nth-solution n*)
    by (*simp only*:)
  also have *pell-valuation* (*nth-solution m*) = *pell-valuation fund-sol* ^ *m*
    by *simp*
  also have *pell-valuation* (*nth-solution n*) = *pell-valuation fund-sol* ^ *n*
    by *simp*
  finally show $m = n$
    using *pell-valuation-fund-sol-ge-2* **by** (*subst* (*asm*) *power-inject-exp*) *auto*
qed

theorem *nth-solution-sound* [*intro*]: *solution* (*nth-solution* *n*)
  using *fund-sol-is-nontriv-solution* **by** (*auto simp*: *nth-solution-def*)

theorem *nth-solution-sound′* [*intro*]: $n > 0 \Longrightarrow$ *nontriv-solution* (*nth-solution* *n*)
  using *fund-sol-is-nontriv-solution* **by** (*auto simp*: *nth-solution-def*)

theorem *nth-solution-complete*:
  fixes z :: *nat* × *nat*
  assumes *solution* z
  shows $z \in$ *range nth-solution*
proof (*cases* $z = (1, 0)$)
  case *True*
  hence $z =$ *nth-solution 0* **by** (*simp add*: *nth-solution-def*)
  thus *?thesis* **by** *auto*
next
  case *False*
  with *assms* **have** *nontriv-solution* z **by** (*auto simp*: *nontriv-solution-altdef*)
  show *?thesis*
  proof (*rule ccontr*)
    assume ¬*?thesis*
    hence ∗: *pell-power fund-sol* $n \neq z$ **for** n **unfolding** *nth-solution-def* **by** *blast*

    define u **where** $u =$ *pell-valuation fund-sol*
    define v **where** $v =$ *pell-valuation* z
    define n **where** $n =$ *nat* $\lfloor$*log* u $v\rfloor$
    have *u-ge-2*: $u \geq 2$ **using** *pell-valuation-fund-sol-ge-2* **by** (*auto simp*: *u-def*)
    have *v-pos*: $v > 0$ **unfolding** *v-def* **using** *assms*
      by (*intro pell-valuation-solution-pos-nat*) *auto*
    have *u-le-v*: $u \leq v$ **unfolding** *u-def v-def* **by** (*rule fund-sol-minimal′*) *fact*

    have *u-power-neq-v*: u ^ $k \neq v$ **for** k
    proof
      assume u ^ $k = v$
      also have u ^ $k =$ *pell-valuation* (*pell-power fund-sol* k)
        by (*simp add*: *u-def*)
      also have $\ldots = v \longleftrightarrow$ *pell-power fund-sol* $k = z$
        unfolding *v-def* **by** (*subst pell-valuation-eq-iff*) (*auto split*: *prod.splits*)
      finally show *False* **using** ∗ **by** *blast*
    qed

    from *u-le-v v-pos u-ge-2* **have** *log-ge-1*: *log* u $v \geq 1$
      by (*subst one-le-log-cancel-iff*) *auto*

    define z' **where** $z' =$ *pell-mul* z (*pell-power* (*pell-cnj fund-sol*) n)
    define x **and** y **where** $x =$ *nat* $|$*fst* $z'|$ **and** $y =$ *nat* $|$*snd* $z'|$
    have *solution* z' **using** *assms fund-sol-is-nontriv-solution* **unfolding** z'*-def*
      by (*intro solution-pell-mul solution-pell-power solution-pell-cnj*) (*auto simp*:
case-prod-unfold)

```
    have u ^ n < v
    proof -
      from u-ge-2 have u ^ n = u powr real n by (subst powr-realpow) auto
      also have ... ≤ u powr log u v using u-ge-2 log-ge-1
        by (intro powr-mono) (auto simp: n-def)
      also have ... = v
        using u-ge-2 v-pos by (subst powr-log-cancel) auto
      finally have u ^ n ≤ v .
      with u-power-neq-v[of n] show ?thesis by linarith
    qed
    moreover have v < u ^ Suc n
    proof -
      have v = u powr log u v
        using u-ge-2 v-pos by (subst powr-log-cancel) auto
      also have log u v ≤ 1 + real-of-int ⌊log u v⌋ by linarith
      hence u powr log u v ≤ u powr real (Suc n) using u-ge-2 log-ge-1
        by (intro powr-mono) (auto simp: n-def)
      also have ... = u ^ Suc n using u-ge-2 by (subst powr-realpow) auto
      finally have u ^ Suc n ≥ v .
      with u-power-neq-v[of Suc n] show ?thesis by linarith
    qed
    ultimately have v / u ^ n ∈ {1<..<u}
      using u-ge-2 by (simp add: field-simps)
    also have v / u ^ n = pell-valuation z'
      using fund-sol-is-nontriv-solution
      by (auto simp add: z'-def u-def v-def pell-valuation-cnj-solution field-simps)
    finally have val: pell-valuation z' ∈ {1<..<u} .

    from val and ‹solution z'› have nontriv-solution z'
      by (auto simp: nontriv-solution-altdef)
    from ‹solution z'› and val have fst z' > 0 ∧ snd z' > 0
      by (intro pell-valuation-solution-gt1D) auto

    hence [simp]: z' = (int x, int y)
      by (auto simp: x-def y-def)

    from ‹nontriv-solution z'› have pell-valuation (int x, int y) ≥ u
      unfolding u-def by (intro fund-sol-minimal) auto
    with val show False by simp
  qed
qed

corollary solution-iff-nth-solution:
  fixes z :: nat × nat
  shows solution z ⟷ z ∈ range nth-solution
  using nth-solution-sound nth-solution-complete by blast

corollary solution-iff-nth-solution':
  fixes z :: int × int
```

shows *solution* $(a,\ b) \longleftrightarrow (\mathit{nat}\ |a|,\ \mathit{nat}\ |b|) \in \mathit{range}\ \mathit{nth\text{-}solution}$
proof −
have *solution* $(a,\ b) \longleftrightarrow \mathit{solution}\ (\mathit{nat}\ |a|,\ \mathit{nat}\ |b|)$
by *simp*
also have $\ldots \longleftrightarrow (\mathit{nat}\ |a|,\ \mathit{nat}\ |b|) \in \mathit{range}\ \mathit{nth\text{-}solution}$
by (*rule solution-iff-nth-solution*)
finally show *?thesis* **.**
qed

corollary *infinite-solutions*: *infinite* $\{z :: \mathit{nat} \times \mathit{nat}.\ \mathit{solution}\ z\}$
proof −
have *infinite* (*range nth-solution*)
by (*intro range-inj-infinite nth-solution-inj*)
also have *range nth-solution* $= \{z :: \mathit{nat} \times \mathit{nat}.\ \mathit{solution}\ z\}$
by (*auto simp*: *solution-iff-nth-solution*)
finally show *?thesis* **.**
qed

corollary *infinite-solutions'*: *infinite* $\{z :: \mathit{int} \times \mathit{int}.\ \mathit{solution}\ z\}$
proof
assume *finite* $\{z :: \mathit{int} \times \mathit{int}.\ \mathit{solution}\ z\}$
hence *finite* (*map-prod* $(\mathit{nat} \circ \mathit{abs})$ $(\mathit{nat} \circ \mathit{abs})$ ‘ $\{z :: \mathit{int} \times \mathit{int}.\ \mathit{solution}\ z\}$)
by (*rule finite-imageI*)
also have (*map-prod* $(\mathit{nat} \circ \mathit{abs})$ $(\mathit{nat} \circ \mathit{abs})$ ‘ $\{z :: \mathit{int} \times \mathit{int}.\ \mathit{solution}\ z\}) =$
$\{z :: \mathit{nat} \times \mathit{nat}.\ \mathit{solution}\ z\}$
by (*auto simp*: *map-prod-def image-iff intro*!: *exI*[*of* - *int x* **for** *x*])
finally show *False* **using** *infinite-solutions* **by** *contradiction*
qed

lemma *strict-mono-pell-valuation-nth-solution*: *strict-mono* (*pell-valuation* ∘ *nth-solution*)
using *pell-valuation-fund-sol-ge-2*
by (*auto simp*: *strict-mono-def intro*!: *power-strict-increasing*)

lemma *strict-mono-nth-solution*:
strict-mono (*fst* ∘ *nth-solution*) *strict-mono* (*snd* ∘ *nth-solution*)
proof −
let *?g* = *nth-solution*
have *fst* (*?g m*) < *fst* (*?g n*) ∧ *snd* (*?g m*) < *snd* (*?g n*) **if** $m < n$ **for** *m n*
using *pell-valuation-fund-sol-ge-2 that*
by (*subst solutions-less-iff-pell-valuation-less*) *auto*
thus *strict-mono* (*fst* ∘ *nth-solution*) *strict-mono* (*snd* ∘ *nth-solution*)
by (*auto simp*: *strict-mono-def*)
qed

end

2.11 The case of an "almost square" parameter

If D is equal to $a^2 - 1$ for some $a > 1$, we have a particularly simple case where the fundamental solution is simply $(1, a)$.

context
  fixes a :: *nat*
  assumes a: $a > 1$
begin

lemma *pell-square-minus1*: *pell* $(a^2 - Suc\ 0)$
proof
  show $\neg$*is-square* $(a^2 - Suc\ 0)$
  proof
    assume *is-square* $(a^2 - Suc\ 0)$
    then obtain k **where** $k^2 = a^2 - 1$ **by** (*auto elim*: *is-nth-powerE*)
    with a **have** $a^2 = Suc\ (k^2)$ **by** *simp*
    hence $a = 1$ **using** *pell-square-solution-nat-iff*[*of 1 a k*] **by** *simp*
    with a **show** *False* **by** *simp*
  qed
qed

interpretation *pell* $a^2 - Suc\ 0$
  by (*rule pell-square-minus1*)

lemma *fund-sol-square-minus1*: *fund-sol* $= (a, 1)$
proof $-$
  from a **have** *sol*: *nontriv-solution* $(a, 1)$
    by (*simp add*: *nontriv-solution-def*)
  from *sol* **have** *snd fund-sol* ≤ 1
    using *fund-sol-minimal''*[*of* $(a, 1)$] **by** *auto*
  with *solutions-linorder-strict*[*of a 1 fst fund-sol snd fund-sol*]
      fund-sol-is-nontriv-solution sol
  show *fund-sol* $= (a, 1)$
    by (*cases fund-sol*) (*auto simp*: *nontriv-solution-altdef*)
qed

end

2.12 Alternative presentation of the main results

theorem *pell-solutions*:
 fixes D :: *nat*
 assumes $\nexists k.\ D = k^2$
 obtains $x_0\ y_0$ :: *nat*
 where $\forall (x{::}int)\ (y{::}int).$
    $x^2 - D * y^2 = 1 \longleftrightarrow$
    $(\exists n{::}nat.\ nat\ |x| + sqrt\ D * nat\ |y| = (x_0 + sqrt\ D * y_0)\ \widehat{}\ n)$
proof $-$
  from *assms* **interpret** *pell*

by *unfold-locales* (*auto simp*: *is-nth-power-def*)
show *?thesis*
proof (*rule that*[*of fst fund-sol snd fund-sol*], *intro allI*, *goal-cases*)
case (*1 x y*)
have (x^2 − *int D* ∗ y^2 = *1*) ⟷ *solution* (*x*, *y*)
by (*auto simp*: *solution-def*)
also have . . . ⟷ (∃ *n*. (*nat* |*x*|, *nat* |*y*|) = *nth-solution n*)
by (*subst solution-iff-nth-solution′*) *blast*
also have (λ*n*. (*nat* |*x*|, *nat* |*y*|) = *nth-solution n*) =
(λ*n*. *pell-valuation* (*nat* |*x*|, *nat* |*y*|) = *pell-valuation* (*nth-solution n*))
by (*subst pell-valuation-eq-iff*) (*auto simp add*: *case-prod-unfold prod-eq-iff fun-eq-iff*)
also have . . . = (λ*n*. *nat* |*x*| + *sqrt D* ∗ *nat* |*y*| = (*fst fund-sol* + *sqrt D* ∗ *snd fund-sol*) ^ *n*)
by (*subst pell-valuation-nth-solution*)
(*simp add*: *pell-valuation-def case-prod-unfold mult-ac*)
finally show *?case* **.**
qed
qed

corollary *pell-solutions-infinite*:
fixes *D* :: *nat*
assumes ∄ *k*. $D = k^2$
shows *infinite* {(*x* :: *int*, *y* :: *int*). x^2 − *D* ∗ y^2 = *1*}
proof −
from *assms* **interpret** *pell*
by *unfold-locales* (*auto simp*: *is-nth-power-def*)
have {(*x* :: *int*, *y* :: *int*). x^2 − *D* ∗ y^2 = *1*} = {*z*. *solution z*}
by (*auto simp*: *solution-def*)
also have *infinite* . . . **by** (*rule infinite-solutions′*)
finally show *?thesis* **.**
qed

end

2.13 Executable code

theory *Pell-Algorithm*
imports
Pell
Efficient-Discrete-Sqrt
HOL−Library.Discrete-Functions
HOL−Library.While-Combinator
HOL−Library.Stream
begin

2.13.1 Efficient computation of powers by squaring

The following is a tail-recursive implementation of exponentiation by squaring. It works for any binary operation f that fulfils $f\ x\ (f\ x\ z) = f\ (f\ x\ x)\ z$, i. e. some weak form of associativity.

context
  fixes *f* :: *'a* ⇒ *'a* ⇒ *'a*
begin

function *efficient-power* :: *'a* ⇒ *'a* ⇒ *nat* ⇒ *'a* **where**
  efficient-power y x 0 = *y*
| *efficient-power y x* (*Suc 0*) = *f x y*
| *n* ≠ *0* ⟹ *even n* ⟹ *efficient-power y x n* = *efficient-power y* (*f x x*) (*n div 2*)
| *n* ≠ *1* ⟹ *odd n* ⟹ *efficient-power y x n* = *efficient-power* (*f x y*) (*f x x*) (*n div 2*)
  by *force+*
termination by (*relation measure* (*snd* ∘ *snd*)) (*auto elim*: *oddE*)

lemma *efficient-power-code* [*code*]:
  efficient-power y x n =
    (*if n* = *0 then y*
     else if n = *1 then f x y*
     else if even n then efficient-power y (*f x x*) (*n div 2*)
     else efficient-power (*f x y*) (*f x x*) (*n div 2*))
  by (*induction y x n rule*: *efficient-power.induct*) *auto*

lemma *efficient-power-correct*:
  assumes ⋀*x z*. *f x* (*f x z*) = *f* (*f x x*) *z*
  shows *efficient-power y x n* = (*f x* ^^ *n*) *y*
proof −
  have [*simp*]: *f* ^^ *2* = (λ*x*. *f* (*f x*)) **for** *f* :: *'a* ⇒ *'a*
    by (*simp add*: *eval-nat-numeral o-def*)
  show *?thesis*
    by (*induction y x n rule*: *efficient-power.induct*)
       (*auto elim*!: *evenE oddE simp*: *funpow-mult* [*symmetric*] *funpow-Suc-right assms*
          simp del: *funpow.simps*(*2*))
qed

end

2.13.2 Multiplication and powers of solutions

We define versions of Pell solution multiplication and exponentiation specialised to natural numbers, both for efficiency reasons and to circumvent the problem of generating code for definitions made inside locales.

fun *pell-mul-nat* :: *nat* ⇒ *nat* × *nat* ⇒ - **where**
  pell-mul-nat D (*a*, *b*) (*x*, *y*) = (*a* ∗ *x* + *D* ∗ *b* ∗ *y*, *a* ∗ *y* + *b* ∗ *x*)

lemma (**in** *pell*) *pell-mul-nat-correct* [*simp*]: *pell-mul-nat D* = *pell.pell-mul D*
by (*auto simp add*: *pell-mul-def fun-eq-iff*)

definition *efficient-pell-power* :: *nat* ⇒ *nat* × *nat* ⇒ *nat* ⇒ *nat* × *nat* **where**
efficient-pell-power D z n = *efficient-power* (*pell-mul-nat D*) (*1*, *0*) *z n*

lemma *efficient-pell-power-correct* [*simp*]:
efficient-pell-power D z n = (*pell-mul-nat D z* ^^ *n*) (*1*, *0*)
unfolding *efficient-pell-power-def*
by (*intro efficient-power-correct*) (*auto simp*: *algebra-simps*)

2.13.3 Finding the fundamental solution

In the following, we set up a very simple algorithm for computing the fundamental solution (x, y). We try inreasing values for y until $1 + Dy^2$ is a perfect square, which we check using an efficient square-detection algorithm. This is efficient enough to work on some interesting small examples.

Much better algorithms (typically based on the continued fraction expansion of $\sqrt{D}$) are available, but they are also considerably more complicated.

lemma *Discrete-sqrt-square-is-square*:
assumes *is-square n*
shows *floor-sqrt n* ^ *2* = *n*
using *assms* **unfolding** *is-nth-power-def* **by** *force*

definition *find-fund-sol-step* :: *nat* ⇒ *nat* × *nat* + *nat* × *nat* ⇒ - **where**
find-fund-sol-step D = (λ*Inl* (*y*, *y'*) ⇒
(*case get-nat-sqrt y' of*
Some x ⇒ *Inr* (*x*, *y*)
| *None* ⇒ *Inl* (*y* + *1*, *y'* + *D* ∗ (*2* ∗ *y* + *1*))))

definition *find-fund-sol* **where**
find-fund-sol D =
(*if square-test D then*
(*0*, *0*)
else
sum.projr (*while sum.isl* (*find-fund-sol-step D*) (*Inl* (*1*, *1* + *D*))))

lemma *fund-sol-code*:
assumes ¬*is-square* (*D* :: *nat*)
shows *pell.fund-sol D* = *sum.projr* (*while isl* (*find-fund-sol-step D*) (*Inl* (*Suc 0*, *Suc D*)))
proof −
from *assms* **interpret** *pell D* **by** *unfold-locales*
note [*simp*] = *find-fund-sol-step-def*
define *f* **where** *f* = *find-fund-sol-step D*
define *P* :: *nat* ⇒ *bool* **where** *P* = (λ*y*. *y* > *0* ∧ *is-square* (*y*^*2* ∗ *D* + *1*))
define *Q* :: *nat* × *nat* ⇒ *bool* **where**

$Q = (\lambda(x,y).\ P\ y \wedge (\forall\, y' \in \{0{<}..{<}y\}.\ \neg P\ y') \wedge x = \textit{floor-sqrt}\ (y\hat{}2 * D + 1))$
define $R :: nat \times nat + nat \times nat \Rightarrow bool$
where $R = (\lambda s.\ case\ s\ of$
$Inl\ (m, m') \Rightarrow m > 0 \wedge (m' = m\hat{}2 * D + 1) \wedge (\forall\, y \in \{0{<}..{<}m\}.$
$\neg \textit{is-square}\ (y\hat{}2 * D + 1))$
$\mid Inr\ x \Rightarrow Q\ x)$
define $rel :: ((nat \times nat + nat \times nat) \times (nat \times nat + nat \times nat))\ set$
where $rel = \{(A,B).\ (case\ (A, B)\ of$
$(Inl\ (m, \text{-}), Inl\ (m', \text{-})) \Rightarrow m' > 0 \wedge m > m' \wedge m \leq snd$
$\textit{fund-sol}$
$\mid (Inr\ \text{-}, Inl\ (m', \text{-})) \Rightarrow m' \leq snd\ \textit{fund-sol}$
$\mid \text{-} \Rightarrow False) \wedge A = f\ B\}$
obtain *x y* **where** *xy*: $sum.projr\ (while\ isl\ f\ (Inl\ (Suc\ 0, Suc\ D))) = (x, y)$
by $(cases\ sum.projr\ (while\ isl\ f\ (Inl\ (Suc\ 0, Suc\ D))))$

have *neq-fund-solI*: $y \neq snd\ \textit{fund-sol}$ **if** $\neg\ \textit{is-square}\ (Suc\ (y^2 * D))$ **for** *y*
proof
assume $y = snd\ \textit{fund-sol}$
with *fund-sol-is-nontriv-solution* **have** $Suc\ (y^2 * D) = fst\ \textit{fund-sol}\ \hat{}\ 2$
by (*simp add*: *nontriv-solution-def case-prod-unfold*)
hence $\textit{is-square}\ (Suc\ (y^2 * D))$ **by** *simp*
with *that* **show** *False* **by** *contradiction*
qed

have $\textit{case-sum}\ (\lambda\text{-}.\ False)\ Q\ (while\ sum.isl\ f\ (Inl\ (m, m\hat{}2 * D + 1)))$
if $\forall\, y \in \{0{<}..{<}m\}.\ \neg \textit{is-square}\ (y\hat{}2 * D + 1)\ m > 0$ **for** *m*
proof (*rule while-rule*[**where** $b = sum.isl$])
show $R\ (Inl\ (m, m^2 * D + 1))$
using *that* **by** (*auto simp*: *R-def*)
next
fix *s* **assume** *R s isl s*
thus $R\ (f\ s)$
by (*auto simp*: *not-less-less-Suc-eq Q-def P-def R-def f-def get-nat-sqrt-def*
power2-eq-square algebra-simps split: *sum.splits prod.splits*)
next
fix *s* **assume** $R\ s\ \neg isl\ s$
thus $case\ s\ of\ Inl\ \text{-} \Rightarrow False \mid Inr\ x \Rightarrow Q\ x$
by (*auto simp*: *R-def split*: *sum.splits*)
next
fix *s* **assume** *s*: *R s isl s*
show $(f\ s, s) \in rel$
proof (*cases s*)
case [*simp*]: $(Inl\ s')$
obtain *a b* **where** [*simp*]: $s' = (a, b)$ **by** $(cases\ s')$
from *s* **have** $*$: $a > 0\ b = Suc\ (a^2 * D)\ \bigwedge y.\ y \in \{0{<}..{<}a\} \Longrightarrow \neg\ \textit{is-square}$
$(Suc\ (y^2 * D))$
by (*auto simp*: *R-def*)

have $a < snd\ \textit{fund-sol}$ **if** $**$: $\neg\ \textit{is-square}\ (Suc\ (a^2 * D))$

proof –
  from *neq-fund-solI* **have** $y' \neq$ *snd fund-sol* **if** $y' \in \{0<..<Suc\ a\}$ **for** y'
    using $*$ $**$ *that* **by** (*cases* $y' = a$) *auto*
  moreover have *snd fund-sol* $\neq 0$ **using** *fund-sol-is-nontriv-solution*
    by (*intro notI*, *cases fund-sol*) (*auto simp*: *nontriv-solution-altdef*)
  ultimately have $\forall y' \leq a.\ y' \neq$ *snd fund-sol* **by** (*auto simp*: *less-Suc-eq-le*)
  thus *snd fund-sol* $> a$ **by** (*cases* $a <$ *snd fund-sol*) (*auto simp*: *not-less*)
qed
moreover have $a \leq$ *snd fund-sol*
proof –
  have $\forall y' \in \{0<..<a\}.\ y' \neq$ *snd fund-sol* **using** *neq-fund-solI* $*$
    by (*auto simp*: *less-Suc-eq-le*)
  moreover have *snd fund-sol* $\neq 0$ **using** *fund-sol-is-nontriv-solution*
    by (*intro notI*, *cases fund-sol*) (*auto simp*: *nontriv-solution-altdef*)
  ultimately have $\forall y' < a.\ y' \neq$ *snd fund-sol* **by** (*auto simp*: *less-Suc-eq-le*)
  thus *snd fund-sol* $\geq a$ **by** (*cases* $a \leq$ *snd fund-sol*) (*auto simp*: *not-less*)
qed
ultimately show *?thesis* **using** $*$
  by (*auto simp*: *f-def get-nat-sqrt-def rel-def*)
qed (*insert s*, *auto*)
next
define *rel'*
  where $rel' = \{(y, x).\ (case\ x\ of\ Inl\ (m, \text{-}) \Rightarrow m \leq snd\ fund\text{-}sol \mid Inr\ \text{-} \Rightarrow False) \wedge y = f\ x\}$
have *wf rel'* **unfolding** *rel'-def*
  by (*rule wf-if-measure*[**where** $f = \lambda z.\ case\ z\ of\ Inl\ (m, \text{-}) \Rightarrow Suc\ (snd\ fund\text{-}sol) - m \mid \text{-} \Rightarrow 0$])
    (*auto split*: *prod.splits sum.splits simp*: *f-def get-nat-sqrt-def*)
moreover have $rel \subseteq rel'$
proof *safe*
  fix *w z* **assume** $(w, z) \in rel$
  thus $(w, z) \in rel'$ **by** (*cases w*; *cases z*) (*auto simp*: *rel-def rel'-def*)
qed
ultimately show *wf rel* **by** (*rule wf-subset*)
qed
from *this*[*of 1*] **and** *xy* **have** $*$: $Q\ (x, y)$
  by (*auto split*: *sum.splits*)

from $*$ **have** *is-square* $(Suc\ (y^2 * D))$ **by** (*simp add*: *Q-def P-def*)
with $*$ **have** $x^2 = Suc\ (y^2 * D)$ $y > 0$
  by (*auto simp*: *Q-def P-def Discrete-sqrt-square-is-square*)
hence *nontriv-solution* (x, y)
  by (*auto simp*: *nontriv-solution-def*)
from *this* **have** *snd fund-sol* $\leq$ *snd* (x, y)
  by (*rule fund-sol-minimal''*)
moreover have *snd fund-sol* $\geq y$
proof –
  from $*$ **have** $(\forall y' \in \{0<..<y\}.\ \neg$ *is-square* $(Suc\ (y'^2 * D)))$
    by (*simp add*: *Q-def P-def*)

with *neq-fund-solI* **have** $(\forall y' \in \{0 <..< y\}.\ y' \neq snd\ fund\text{-}sol)$
by *auto*
moreover have *snd fund-sol* $\neq 0$
using *fund-sol-is-nontriv-solution*
by (*cases fund-sol*) (*auto intro*!: *Nat.gr0I simp*: *nontriv-solution-altdef*)
ultimately have $(\forall y' < y.\ y' \neq snd\ fund\text{-}sol)$ **by** *auto*
thus *snd fund-sol* $\geq y$ **by** (*cases snd fund-sol* $\geq y$) (*auto simp*: *not-less*)
qed
ultimately have *snd fund-sol* $= y$ **by** *simp*
with *solutions-linorder-strict*[*of x y fst fund-sol snd fund-sol*]
fund-sol-is-nontriv-solution ‹*nontriv-solution* (x, y)›
have *fst fund-sol* $= x$ **by** (*cases fund-sol*) (*auto simp*: *nontriv-solution-altdef*)
with ‹*snd fund-sol* $= y$› **have** *fund-sol* $= (x, y)$
by (*cases fund-sol*) *simp*
with *xy* **show** *?thesis* **by** (*simp add*: *f-def*)
qed

lemma *find-fund-sol-correct*: *find-fund-sol D* = (*if is-square D then* (*0*, *0*) *else pell.fund-sol D*)
by (*simp add*: *find-fund-sol-def fund-sol-code square-test-correct*)

2.13.4 The infinite list of all solutions

definition *pell-solutions* :: *nat* ⇒ (*nat* × *nat*) *stream* **where**
pell-solutions D = (*let z* = *find-fund-sol D in siterate* (*pell-mul-nat D z*) (*1*, *0*))

lemma (**in** *pell*) *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*])

2.13.5 Computing the n-th solution

definition *find-nth-solution* :: *nat* ⇒ *nat* ⇒ *nat* × *nat* **where**
find-nth-solution D n =
(*if is-square D then* (*0*, *0*) *else*
let z = *sum.projr* (*while isl* (*find-fund-sol-step D*) (*Inl* (*Suc 0*, *Suc D*)))
in efficient-pell-power D z n)

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

end

2.13.6 Tests

theory *Pell-Algorithm-Test*
imports
Pell-Algorithm
HOL−Library.Code-Target-Numeral

HOL−Library.Code-Lazy
begin

code-lazy-type *stream*

value *find-fund-sol 73*
value *find-fund-sol 106*

value *stake 100* (*pell-solutions 73*)
value *snth* (*pell-solutions 73*) *600*

value *find-nth-solution 73 600*
value *find-nth-solution 106 10*

end

References

[1] Pell's equation, handout for MATHS 714. Lecture notes, University of Auckland, 2008.

[2] H. Cohen. *A Course in Computational Algebraic Number Theory.* Springer, 2010.

[3] M. Jacobson and H. Williams. *Solving the Pell Equation.* CMS Books in Mathematics. Springer New York, 2008.