Pell's Equation

Manuel Eberl

December 14, 2021

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
HOL−Library.Tree
HOL−Library.IArray
begin

1 Efficient Algorithms for the Square Root on 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*: $Discrete.sqrt\ n = nat\ \lfloor sqrt\ n \rfloor$
⟨*proof*⟩

fun *newton-sqrt-aux* :: $nat \Rightarrow nat \Rightarrow nat$ **where**
newton-sqrt-aux x n =
$(let\ y = (x + n\ div\ x)\ div\ 2$
$in\ if\ y < x\ then\ newton\text{-}sqrt\text{-}aux\ y\ n\ else\ x)$

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

lemma *newton-sqrt-aux-simps*:
$(x + n\ div\ x)\ div\ 2 < x \Longrightarrow newton\text{-}sqrt\text{-}aux\ x\ n = newton\text{-}sqrt\text{-}aux\ ((x + n\ div\ x)\ div\ 2)\ n$
$(x + n\ div\ x)\ div\ 2 \geq x \Longrightarrow newton\text{-}sqrt\text{-}aux\ x\ n = x$
⟨*proof*⟩

lemma *heron-step-real*: $[\![t > 0;\ n \geq 0]\!] \Longrightarrow (t + n/t)\ /\ 2 \geq sqrt\ n$
⟨*proof*⟩

lemma *heron-step-div-eq-floored*:
$(t{::}nat) > 0 \Longrightarrow (t + (n{::}nat)\ div\ t)\ div\ 2 = nat\ \lfloor (t + n/t)\ /\ 2 \rfloor$
⟨*proof*⟩

lemma *heron-step*: $t > 0 \Longrightarrow (t + n\ div\ t)\ div\ 2 \geq Discrete.sqrt\ n$
⟨*proof*⟩

lemma *newton-sqrt-aux-correct*:
assumes $x \geq Discrete.sqrt\ n$
shows $newton\text{-}sqrt\text{-}aux\ x\ n = Discrete.sqrt\ n$
⟨*proof*⟩

definition *newton-sqrt* :: *nat* ⇒ *nat* **where**
newton-sqrt n = *newton-sqrt-aux n n*

declare *Discrete.sqrt-code* [*code del*]

theorem *Discrete-sqrt-eq-newton-sqrt* [*code*]: *Discrete.sqrt n* = *newton-sqrt n*
⟨*proof*⟩

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* ∈ {..<*11*} ⟹ *IArray.sub q11-array i* ⟷ *i* ∈ *q11*

$\langle proof \rangle$

lemma *sub-q63-array*: $i \in \{..<63\} \Longrightarrow$ *IArray.sub q63-array* $i \longleftrightarrow i \in q63$
$\langle proof \rangle$

lemma *sub-q64-array*: $i \in \{..<64\} \Longrightarrow$ *IArray.sub q64-array* $i \longleftrightarrow i \in q64$
$\langle proof \rangle$

lemma *sub-q65-array*: $i \in \{..<65\} \Longrightarrow$ *IArray.sub q65-array* $i \longleftrightarrow i \in q65$
$\langle proof \rangle$

lemma *in-q11-code*: x *mod* $11 \in q11 \longleftrightarrow$ *IArray.sub q11-array* (x *mod* 11)
$\langle proof \rangle$

lemma *in-q63-code*: x *mod* $63 \in q63 \longleftrightarrow$ *IArray.sub q63-array* (x *mod* 63)
$\langle proof \rangle$

lemma *in-q64-code*: x *mod* $64 \in q64 \longleftrightarrow$ *IArray.sub q64-array* (x *mod* 64)
$\langle proof \rangle$

lemma *in-q65-code*: x *mod* $65 \in q65 \longleftrightarrow$ *IArray.sub q65-array* (x *mod* 65)
$\langle proof \rangle$

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* $11 \in q11 \wedge n =$ (*Discrete.sqrt* $n)^2$))

lemma *square-test-code* [*code*]:
square-test n =
(*IArray.sub q64-array* (n *mod* 64) $\wedge$ (*let* $r = n$ *mod* 45045 *in*
IArray.sub q63-array (r *mod* 63) $\wedge$
IArray.sub q65-array (r *mod* 65) $\wedge$
IArray.sub q11-array (r *mod* 11) $\wedge n =$ (*Discrete.sqrt* $n)^2$))
$\langle proof \rangle$

lemma *square-mod-lower*: $m > 0 \Longrightarrow (q^2 :: nat)$ *mod* $m = a \Longrightarrow \exists\, q' < m.\ q'^2$ *mod* $m = a$
$\langle proof \rangle$

lemma *q11-upto-def*: $q11 = (\lambda k.\ k^2$ *mod* $11)$ ‘ $\{..<11\}$
$\langle proof \rangle$

lemma *q11-infinite-def*: $q11 = (\lambda k.\ k^2$ *mod* $11)$ ‘ $\{0..\}$
$\langle proof \rangle$

lemma *q63-upto-def*: $q63 = (\lambda k.\ k^2\ mod\ 63)$ ‘ $\{..<63\}$
⟨*proof*⟩

lemma *q63-infinite-def*: $q63 = (\lambda k.\ k^2\ mod\ 63)$ ‘ $\{0..\}$
⟨*proof*⟩

lemma *q64-upto-def*: $q64 = (\lambda k.\ k^2\ mod\ 64)$ ‘ $\{..<64\}$
⟨*proof*⟩

lemma *q64-infinite-def*: $q64 = (\lambda k.\ k^2\ mod\ 64)$ ‘ $\{0..\}$
⟨*proof*⟩

lemma *q65-upto-def*: $q65 = (\lambda k.\ k^2\ mod\ 65)$ ‘ $\{..<65\}$
⟨*proof*⟩

lemma *q65-infinite-def*: $q65 = (\lambda k.\ k^2\ mod\ 65)$ ‘ $\{0..\}$
⟨*proof*⟩

lemma *square-mod-existence*:
fixes *n k* :: *nat*
assumes $\exists q.\ q^2 = n$
shows $\exists q.\ n\ mod\ k = q^2\ mod\ k$
⟨*proof*⟩

theorem *square-test-correct*: *square-test n* ⟷ *is-square n*
⟨*proof*⟩

definition *get-nat-sqrt* :: *nat* ⇒ *nat option*
where *get-nat-sqrt n* = (*if is-square n then Some* (*Discrete.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* = *Discrete.sqrt n in if* $x^2 = n$ *then Some x else None*) *else None*)
⟨*proof*⟩

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*]: $gcd\ x\ (y :: int) \leq 0 \longleftrightarrow x = 0 \wedge y = 0$
⟨*proof*⟩

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

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

lemma *nonneg-sqrt-nat-or-irrat*:
assumes $x\ \hat{}\ 2 = real\ a$ **and** $x \geq 0$
shows $x \in \mathbb{N} \vee x \notin \mathbb{Q}$
⟨*proof*⟩

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 $x \in \mathbb{Z} \vee x \notin \mathbb{Q}$
⟨*proof*⟩

corollary *sqrt-nat-or-irrat'*:
$sqrt\ (real\ a) \in \mathbb{N} \vee sqrt\ (real\ a) \notin \mathbb{Q}$
⟨*proof*⟩

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*:
$sqrt\ (real\ n) \in \mathbb{N} \longleftrightarrow is\text{-}square\ n$
⟨*proof*⟩

corollary *irrat-sqrt-nonsquare*: $\neg is\text{-}square\ n \Longrightarrow sqrt\ (real\ n) \notin \mathbb{Q}$
⟨*proof*⟩

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** $x \,\hat{}\, 2 = 1 + D * y \,\hat{}\, 2$
  shows $(x, y) = (1, 0)$
⟨*proof*⟩

lemma *pell-square-solution-int-aux*:
  fixes $x\ y$ :: *int*
  assumes $D > 0$ **and** $x \,\hat{}\, 2 = 1 + D * y \,\hat{}\, 2$
  shows $x \in \{-1, 1\} \wedge y = 0$
⟨*proof*⟩

lemma *pell-square-solution-nat-iff*:
  fixes $x\ y$ :: *nat*
  shows $x \,\hat{}\, 2 = 1 + D * y \,\hat{}\, 2 \longleftrightarrow x = 1 \wedge (D = 0 \vee y = 0)$
  ⟨*proof*⟩

lemma *pell-square-solution-int-iff*:
  fixes $x\ y$ :: *int*
  shows $x \,\hat{}\, 2 = 1 + D * y \,\hat{}\, 2 \longleftrightarrow x \in \{-1, 1\} \wedge (D = 0 \vee y = 0)$
  ⟨*proof*⟩

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 $\exists\, u{::}nat.\ \exists\, t{::}int.\ u > 0 \wedge coprime\ u\ t \wedge 1\ /\ s \in \{|t - u * x|{<}..1\ /\ u\}$
⟨*proof*⟩

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 $x \notin \mathbb{Q}$
shows $infinite\ \{(t :: int,\ u :: nat).\ u > 0 \wedge coprime\ u\ t \wedge |t - u * x| < 1\ /\ u\}$
(**is** *infinite ?A*)
⟨*proof*⟩

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

lemma *D-gt-1*: $D > 1$
⟨*proof*⟩

lemma *D-pos*: $D > 0$
⟨*proof*⟩

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*⟩

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

lemma *nontriv-solution-altdef*: $nontriv\text{-}solution\ z \longleftrightarrow solution\ z \wedge z \neq (1, 0)$
⟨*proof*⟩

lemma *solution-trivial-nat* [*simp*, *intro*]: $solution\ (Suc\ 0,\ 0)$
⟨*proof*⟩

lemma *solution-trivial* [*simp*, *intro*]: $solution\ (1,\ 0)$

⟨proof⟩

lemma *solution-uminus-left* [*simp*]: *solution* $(-x,\ y :: {'a} :: \mathit{comm\text{-}ring\text{-}1}) \longleftrightarrow$ *solution* $(x,\ y)$
⟨proof⟩

lemma *solution-uminus-right* [*simp*]: *solution* $(x,\ -y :: {'a} :: \mathit{comm\text{-}ring\text{-}1}) \longleftrightarrow$ *solution* $(x,\ y)$
⟨proof⟩

lemma *solution-0-snd-nat-iff* [*simp*]: *solution* $(a :: \mathit{nat},\ 0) \longleftrightarrow a = 1$
⟨proof⟩

lemma *solution-0-snd-iff* [*simp*]: *solution* $(a :: {'a} :: \mathit{idom},\ 0) \longleftrightarrow a \in \{1,\ -1\}$
⟨proof⟩

lemma *no-solution-0-fst-nat* [*simp*]: $\neg$*solution* $(0,\ b :: \mathit{nat})$
⟨proof⟩

lemma *no-solution-0-fst-int* [*simp*]: $\neg$*solution* $(0,\ b :: \mathit{int})$
⟨proof⟩

lemma *solution-of-nat-of-nat* [*simp*]:
solution $(\mathit{of\text{-}nat}\ a,\ \mathit{of\text{-}nat}\ b :: {'a} :: \{\mathit{comm\text{-}ring\text{-}1},\ \mathit{ring\text{-}char\text{-}0}\}) \longleftrightarrow$ *solution* $(a,\ b)$
⟨proof⟩

lemma *solution-of-nat-of-nat′* [*simp*]:
solution $(\mathit{case}\ z\ \mathit{of}\ (a,\ b) \Rightarrow (\mathit{of\text{-}nat}\ a,\ \mathit{of\text{-}nat}\ b :: {'a} :: \{\mathit{comm\text{-}ring\text{-}1},\ \mathit{ring\text{-}char\text{-}0}\})) \longleftrightarrow$
solution z
⟨proof⟩

lemma *solution-nat-abs-nat-abs* [*simp*]:
solution $(\mathit{nat}\ |x|,\ \mathit{nat}\ |y|) \longleftrightarrow$ *solution* $(x,\ y)$
⟨proof⟩

lemma *nontriv-solution-of-nat-of-nat* [*simp*]:
nontriv-solution $(\mathit{of\text{-}nat}\ a,\ \mathit{of\text{-}nat}\ b :: {'a} :: \{\mathit{comm\text{-}ring\text{-}1},\ \mathit{ring\text{-}char\text{-}0}\}) \longleftrightarrow$ *nontriv-solution* $(a,\ b)$
⟨proof⟩

lemma *nontriv-solution-of-nat-of-nat′* [*simp*]:
nontriv-solution $(\mathit{case}\ z\ \mathit{of}\ (a,\ b) \Rightarrow (\mathit{of\text{-}nat}\ a,\ \mathit{of\text{-}nat}\ b :: {'a} :: \{\mathit{comm\text{-}ring\text{-}1},\ \mathit{ring\text{-}char\text{-}0}\})) \longleftrightarrow$
nontriv-solution z
⟨proof⟩

lemma *nontriv-solution-imp-solution* [*dest*]: *nontriv-solution z* $\Longrightarrow$ *solution z*

⟨*proof*⟩

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* ≥ *0* ⟹ *snd z* ≥ *0* ⟹ *pell-valuation z* ≥ *0*
⟨*proof*⟩

lemma *pell-valuation-uminus-uminus* [*simp*]: *pell-valuation* (−*x*, −*y*) = −*pell-valuation* (*x*, *y*)
⟨*proof*⟩

lemma *pell-valuation-eq-iff* [*simp*]:
pell-valuation z1 = *pell-valuation z2* ⟷ *z1* = *z2*
⟨*proof*⟩

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* ≤ *x* ∧ *b* ≤ *y* ∨ *a* ≥ *x* ∧ *b* ≥ *y*
⟨*proof*⟩

lemma *solutions-linorder-strict*:
fixes *a b x y* :: *nat*
assumes *solution* (*a*, *b*) *solution* (*x*, *y*)
shows (*a*, *b*) = (*x*, *y*) ∨ *a* < *x* ∧ *b* < *y* ∨ *a* > *x* ∧ *b* > *y*
⟨*proof*⟩

lemma *solutions-le-iff-pell-valuation-le*:
fixes *a b x y* :: *nat*
assumes *solution* (*a*, *b*) *solution* (*x*, *y*)
shows *a* ≤ *x* ∧ *b* ≤ *y* ⟷ *pell-valuation* (*a*, *b*) ≤ *pell-valuation* (*x*, *y*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

corollary
fund-sol-is-nontriv-solution: *nontriv-solution fund-sol*
and *fund-sol-minimal*:
nontriv-solution $(a, b) \Longrightarrow$ *pell-valuation fund-sol* $\leq$ *pell-valuation* (*int a*, *int b*)
and *fund-sol-minimal′*:
nontriv-solution (*z* :: *nat* × *nat*) $\Longrightarrow$ *pell-valuation fund-sol* $\leq$ *pell-valuation z*
⟨*proof*⟩

lemma *fund-sol-minimal′′*:
assumes *nontriv-solution z*
shows *fst fund-sol* $\leq$ *fst z snd fund-sol* $\leq$ *snd z*
⟨*proof*⟩

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* :: $('a :: \textit{comm-semiring-1} \times 'a) \Rightarrow ('a \times 'a) \Rightarrow ('a \times 'a)$ **where**
pell-mul = $(\lambda(a,b)\ (x,y).\ (x * a + y * b * \textit{of-nat}\ D,\ x * b + y * a))$

definition *pell-cnj* :: $('a :: \textit{comm-ring-1} \times 'a) \Rightarrow 'a \times 'a$ **where**
pell-cnj = $(\lambda(a,b).\ (a, -b))$

lemma *pell-cnj-snd-0* [*simp*]: *snd z* = *0* $\Longrightarrow$ *pell-cnj z* = *z*
⟨*proof*⟩

lemma *pell-mul-commutes*: *pell-mul z1 z2* = *pell-mul z2 z1*
⟨*proof*⟩

lemma *pell-mul-assoc*: *pell-mul z1* (*pell-mul z2 z3*) = *pell-mul* (*pell-mul z1 z2*) *z3*
⟨*proof*⟩

lemma *pell-mul-trivial-left* [*simp*]: *pell-mul* (*1*, *0*) *z* = *z*
⟨*proof*⟩

lemma *pell-mul-trivial-right* [*simp*]: *pell-mul z* (*1*, *0*) = *z*
⟨*proof*⟩

lemma *pell-mul-trivial-left-nat* [*simp*]: *pell-mul* (*Suc 0*, *0*) *z* = *z*
⟨*proof*⟩

lemma *pell-mul-trivial-right-nat* [*simp*]: *pell-mul z* (*Suc 0*, *0*) = *z*
⟨*proof*⟩

definition *pell-power* :: $('a :: \textit{comm-semiring-1} \times 'a) \Rightarrow \textit{nat} \Rightarrow ('a \times 'a)$ **where**
pell-power z n = $((\lambda z'.\ \textit{pell-mul}\ z'\ z)\ \widehat{\ }\widehat{\ }\ n)\ (1, 0)$

lemma *pell-power-0* [*simp*]: *pell-power z 0* = (*1*, *0*)
⟨*proof*⟩

lemma *pell-power-one* [*simp*]: *pell-power* (*1*, *0*) *n* = (*1*, *0*)
⟨*proof*⟩

lemma *pell-power-one-right* [*simp*]: *pell-power z 1* = *z*
⟨*proof*⟩

lemma *pell-power-Suc*: *pell-power z* (*Suc n*) = *pell-mul z* (*pell-power z n*)
⟨*proof*⟩

lemma *pell-power-add*: *pell-power z* (*m* + *n*) = *pell-mul* (*pell-power z m*) (*pell-power z n*)
⟨*proof*⟩

lemma *pell-valuation-mult* [*simp*]:
pell-valuation (*pell-mul z1 z2*) = *pell-valuation z1* ∗ *pell-valuation z2*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *pell-valuation-trivial* [*simp*]: *pell-valuation* (*1* , *0*) = *1*
⟨*proof*⟩

lemma *pell-valuation-trivial-nat* [*simp*]: *pell-valuation* (*Suc 0* , *0*) = *1*
⟨*proof*⟩

lemma *pell-valuation-cnj*: *pell-valuation* (*pell-cnj z*) = *fst z* − *snd z* ∗ *sqrt D*
⟨*proof*⟩

lemma *pell-valuation-snd-0* [*simp*]: *pell-valuation* (*a*, *0*) = *of-int a*
⟨*proof*⟩

lemma *pell-valuation-0-iff* [*simp*]: *pell-valuation z* = *0* ⟷ *z* = (*0* , *0*)
⟨*proof*⟩

lemma *pell-valuation-solution-pos-nat*:
fixes *z* :: *nat* × *nat*
assumes *solution z*
shows *pell-valuation z* > *0*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *pell-valuation-cnj-solution*:
fixes *z* :: *nat* × *nat*
assumes *solution z*
shows *pell-valuation* (*pell-cnj z*) = *1* / *pell-valuation z*
⟨*proof*⟩

lemma *pell-valuation-power* [*simp*]: *pell-valuation* (*pell-power z n*) = *pell-valuation z* ^ *n*
⟨*proof*⟩

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

n
⟨*proof*⟩

lemma *pell-valuation-fund-sol-ge-2*: *pell-valuation fund-sol* $\geq$ *2*
⟨*proof*⟩

lemma *solution-pell-mul* [*intro*]:
assumes *solution z1 solution z2*
shows *solution* (*pell-mul z1 z2*)
⟨*proof*⟩

lemma *solution-pell-cnj* [*intro*]:
assumes *solution z*
shows *solution* (*pell-cnj z*)
⟨*proof*⟩

lemma *solution-pell-power* [*simp*, *intro*]: *solution z* $\Longrightarrow$ *solution* (*pell-power z n*)
⟨*proof*⟩

lemma *pell-mul-eq-trivial-nat-iff*:
pell-mul z1 z2 = (*Suc 0*, *0*) $\longleftrightarrow$ *z1* = (*Suc 0*, *0*) $\wedge$ *z2* = (*Suc 0*, *0*)
⟨*proof*⟩

lemma *nontriv-solution-pell-nat-mul1*:
solution (*z1* :: *nat* $\times$ *nat*) $\Longrightarrow$ *nontriv-solution z2* $\Longrightarrow$ *nontriv-solution* (*pell-mul z1 z2*)
⟨*proof*⟩

lemma *nontriv-solution-pell-nat-mul2*:
nontriv-solution (*z1* :: *nat* $\times$ *nat*) $\Longrightarrow$ *solution z2* $\Longrightarrow$ *nontriv-solution* (*pell-mul z1 z2*)
⟨*proof*⟩

lemma *nontriv-solution-power-nat* [*intro*]:
assumes *nontriv-solution* (*z* :: *nat* $\times$ *nat*) *n* > *0*
shows *nontriv-solution* (*pell-power z n*)
⟨*proof*⟩

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* $(x, -y) \in \{0<..<1\}$
⟨*proof*⟩

lemma *pell-valuation-pos-neg*:
  assumes $x > 0$ $y < 0$ *solution* (x, y)
  shows *pell-valuation* $(x, y) \in \{0<..<1\}$
  ⟨*proof*⟩

lemma *pell-valuation-neg-neg*:
  assumes $x < 0$ $y < 0$ *solution* (x, y)
  shows *pell-valuation* $(x, y) < -1$
  ⟨*proof*⟩

lemma *pell-valuation-neg-pos*:
  assumes $x < 0$ $y > 0$ *solution* (x, y)
  shows *pell-valuation* $(x, y) \in \{-1<..<0\}$
  ⟨*proof*⟩

lemma *pell-valuation-solution-gt1D*:
  assumes *solution z pell-valuation z* > 1
  shows *fst z* $> 0 \wedge$ *snd z* > 0
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

theorem *nth-solution-inj*: *inj nth-solution*
⟨*proof*⟩

theorem *nth-solution-sound* [*intro*]: *solution* (*nth-solution n*)
⟨*proof*⟩

theorem *nth-solution-sound′* [*intro*]: $n > 0 \Longrightarrow$ *nontriv-solution* (*nth-solution n*)
⟨*proof*⟩

theorem *nth-solution-complete*:
fixes *z* :: *nat* × *nat*
assumes *solution z*
shows $z \in$ *range nth-solution*
⟨*proof*⟩

corollary *solution-iff-nth-solution*:
fixes *z* :: *nat* × *nat*
shows *solution z* ⟷ $z \in$ *range nth-solution*
⟨*proof*⟩

corollary *solution-iff-nth-solution′*:
fixes *z* :: *int* × *int*
shows *solution* (*a*, *b*) ⟷ (*nat* |*a*|, *nat* |*b*|) ∈ *range nth-solution*
⟨*proof*⟩

corollary *infinite-solutions*: *infinite* {*z* :: *nat* × *nat*. *solution z*}
⟨*proof*⟩

corollary *infinite-solutions′*: *infinite* {*z* :: *int* × *int*. *solution z*}
⟨*proof*⟩

lemma *strict-mono-pell-valuation-nth-solution*: *strict-mono* (*pell-valuation* ∘ *nth-solution*)
⟨*proof*⟩

lemma *strict-mono-nth-solution*:
strict-mono (*fst* ∘ *nth-solution*) *strict-mono* (*snd* ∘ *nth-solution*)
⟨*proof*⟩

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*⟩

interpretation *pell* $a^2 - Suc\ 0$
  ⟨*proof*⟩

lemma *fund-sol-square-minus1*: $fund\text{-}sol = (a,\ 1)$
⟨*proof*⟩

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 $\forall (x{::}int)\ (y{::}int).$
   $x^2 - D * y^2 = 1 \longleftrightarrow$
   $(\exists n{::}nat.\ nat\ |x| + sqrt\ D * nat\ |y| = (x_0 + sqrt\ D * y_0)\ \hat{}\ n)$
⟨*proof*⟩

corollary *pell-solutions-infinite*:
 fixes $D :: nat$
 assumes $\nexists k.\ D = k^2$
 shows $infinite\ \{(x :: int,\ y :: int).\ x^2 - D * y^2 = 1\}$
⟨*proof*⟩

end

2.13 Executable code

theory *Pell-Algorithm*
imports
  Pell
  Efficient-Discrete-Sqrt
  HOL−Library.Discrete
  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 $f :: 'a \Rightarrow 'a \Rightarrow 'a$
begin

function *efficient-power* $:: 'a \Rightarrow 'a \Rightarrow nat \Rightarrow 'a$ **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)*
⟨*proof*⟩
termination ⟨*proof*⟩

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))
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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)
⟨*proof*⟩

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 *Discrete.sqrt n ^ 2 = n*
  ⟨*proof*⟩

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*⟩

lemma *find-fund-sol-correct*: *find-fund-sol D* = (*if is-square D then* (*0*, *0*) *else pell.fund-sol D*)
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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.