Gaussian Integers

Manuel Eberl

May 14, 2024

Abstract

The Gaussian integers are the subring $\mathbb{Z}[i]$ of the complex numbers, i. e. the ring of all complex numbers with integral real and imaginary part. This article provides a definition of this ring as well as proofs of various basic properties, such as that they form a Euclidean ring and a full classification of their primes. An executable (albeit not very efficient) factorisation algorithm is also provided.

Lastly, this Gaussian integer formalisation is used in two short applications:

1. The characterisation of all positive integers that can be written as sums of two squares
2. Euclid's formula for primitive Pythagorean triples

While elementary proofs for both of these are already available in the AFP, the theory of Gaussian integers provides more concise proofs and a more high-level view.

Contents

1 Gaussian Integers

theory *Gaussian-Integers*
imports
HOL−Computational-Algebra.Computational-Algebra
HOL−Number-Theory.Number-Theory
begin

1.1 Auxiliary material

lemma *coprime-iff-prime-factors-disjoint*:
fixes $x\ y :: 'a :: factorial\text{-}semiring$
assumes $x \neq 0\ y \neq 0$
shows $coprime\ x\ y \longleftrightarrow prime\text{-}factors\ x \cap prime\text{-}factors\ y = \{\}$
proof
assume *coprime x y*
have *False* **if** $p \in prime\text{-}factors\ x\ p \in prime\text{-}factors\ y$ **for** p
proof −
from *that assms* **have** *p dvd x p dvd y*
by (*auto simp*: *prime-factors-dvd*)
with ‹*coprime x y*› **have** *p dvd 1*
using *coprime-common-divisor* **by** *auto*
with *that assms* **show** *False* **by** (*auto simp*: *prime-factors-dvd*)
qed
thus $prime\text{-}factors\ x \cap prime\text{-}factors\ y = \{\}$ **by** *auto*
next
assume *disjoint*: $prime\text{-}factors\ x \cap prime\text{-}factors\ y = \{\}$
show *coprime x y*
proof (*rule coprimeI*)
fix *d* **assume** *d*: *d dvd x d dvd y*
show *is-unit d*
proof (*rule ccontr*)
assume ¬*is-unit d*
moreover from *this* **and** *d assms* **have** $d \neq 0$ **by** *auto*
ultimately obtain *p* **where** *p*: *prime p p dvd d*
using *prime-divisor-exists* **by** *auto*
with *d* **and** *assms* **have** $p \in prime\text{-}factors\ x \cap prime\text{-}factors\ y$
by (*auto simp*: *prime-factors-dvd*)
with *disjoint* **show** *False* **by** *auto*
qed
qed
qed

lemma *product-dvd-irreducibleD*:
fixes $a\ b\ x :: 'a :: algebraic\text{-}semidom$
assumes *irreducible x*
assumes *a ∗ b dvd x*
shows *a dvd 1 ∨ b dvd 1*
proof −
from *assms* **obtain** *c* **where** $x = a * b * c$

```
    by auto
  hence x = a * (b * c)
    by (simp add: mult-ac)
  from irreducibleD[OF assms(1) this] show a dvd 1 ∨ b dvd 1
    by (auto simp: is-unit-mult-iff)
qed

lemma prime-elem-mult-dvdI:
  assumes prime-elem p p dvd c b dvd c ¬p dvd b
  shows   p * b dvd c
proof −
  from assms(3) obtain a where c: c = a * b
    using mult.commute by blast
  with assms(2) have p dvd a * b
    by simp
  with assms have p dvd a
    by (subst (asm) prime-elem-dvd-mult-iff) auto
  with c show ?thesis by (auto intro: mult-dvd-mono)
qed

lemma prime-elem-power-mult-dvdI:
  fixes p :: 'a :: algebraic-semidom
  assumes prime-elem p p ^ n dvd c b dvd c ¬p dvd b
  shows   p ^ n * b dvd c
proof (cases n = 0)
  case False
  from assms(3) obtain a where c: c = a * b
    using mult.commute by blast
  with assms(2) have p ^ n dvd b * a
    by (simp add: mult-ac)
  hence p ^ n dvd a
    by (rule prime-power-dvd-multD[OF assms(1)]) (use assms False in auto)
  with c show ?thesis by (auto intro: mult-dvd-mono)
qed (use assms in auto)

lemma prime-mod-4-cases:
  fixes p :: nat
  assumes prime p
  shows   p = 2 ∨ [p = 1] (mod 4) ∨ [p = 3] (mod 4)
proof (cases p = 2)
  case False
  with prime-gt-1-nat[of p] assms have p > 2 by auto
  have ¬4 dvd p
    using assms product-dvd-irreducibleD[of p 2 2]
    by (auto simp: prime-elem-iff-irreducible simp flip: prime-elem-nat-iff)
  hence p mod 4 ≠ 0
    by (auto simp: mod-eq-0-iff-dvd)
  moreover have p mod 4 ≠ 2
  proof
```

```
    assume p mod 4 = 2
    hence p mod 4 mod 2 = 0
      by (simp add: cong-def)
    thus False using ‹prime p› ‹p > 2› prime-odd-nat[of p]
      by (auto simp: mod-mod-cancel)
  qed
  moreover have p mod 4 ∈ {0,1,2,3}
    by auto
  ultimately show ?thesis by (auto simp: cong-def)
qed auto

lemma of-nat-prod-mset: of-nat (prod-mset A) = prod-mset (image-mset of-nat A)
  by (induction A) auto

lemma multiplicity-0-left [simp]: multiplicity 0 x = 0
  by (cases x = 0) (auto simp: not-dvd-imp-multiplicity-0)

lemma is-unit-power [intro]: is-unit x ⟹ is-unit (x ^ n)
  by (subst is-unit-power-iff) auto

lemma (in factorial-semiring) pow-divides-pow-iff:
  assumes n > 0
  shows   a ^ n dvd b ^ n ⟷ a dvd b
proof (cases b = 0)
  case False
  show ?thesis
  proof
    assume dvd: a ^ n dvd b ^ n
    with ‹b ≠ 0› have a ≠ 0
      using ‹n > 0› by (auto simp: power-0-left)
    show a dvd b
    proof (rule multiplicity-le-imp-dvd)
      fix p :: 'a assume p: prime p
      from dvd ‹b ≠ 0› have multiplicity p (a ^ n) ≤ multiplicity p (b ^ n)
        by (intro dvd-imp-multiplicity-le) auto
      thus multiplicity p a ≤ multiplicity p b
      using p ‹a ≠ 0› ‹b ≠ 0› ‹n > 0› by (simp add: prime-elem-multiplicity-power-distrib)
    qed fact+
  qed (auto intro: dvd-power-same)
qed (use assms in ‹auto simp: power-0-left›)

lemma multiplicity-power-power:
  fixes p :: 'a :: {factorial-semiring, algebraic-semidom}
  assumes n > 0
  shows   multiplicity (p ^ n) (x ^ n) = multiplicity p x
proof (cases x = 0 ∨ p = 0 ∨ is-unit p)
  case True
  thus ?thesis using ‹n > 0›
    by (auto simp: power-0-left is-unit-power-iff multiplicity-unit-left)
```

```
next
  case False
  show ?thesis
  proof (intro antisym multiplicity-geI)
    have (p ^ multiplicity p x) ^ n dvd x ^ n
      by (intro dvd-power-same) (simp add: multiplicity-dvd)
    thus (p ^ n) ^ multiplicity p x dvd x ^ n
      by (simp add: mult-ac flip: power-mult)
  next
    have (p ^ n) ^ multiplicity (p ^ n) (x ^ n) dvd x ^ n
      by (simp add: multiplicity-dvd)
    hence (p ^ multiplicity (p ^ n) (x ^ n)) ^ n dvd x ^ n
      by (simp add: mult-ac flip: power-mult)
    thus p ^ multiplicity (p ^ n) (x ^ n) dvd x
      by (subst (asm) pow-divides-pow-iff) (use assms in auto)
  qed (use False ‹n > 0› in ‹auto simp: is-unit-power-iff›)
qed

lemma even-square-cong-4-int:
  ‹[x^2 = 0] (mod 4)› if ‹even x› for x :: int
proof −
  from that obtain y where ‹x = 2 * y› ..
  then show ?thesis by (simp add: cong-def)
qed

lemma even-square-cong-4-nat:
  ‹[x^2 = 0] (mod 4)› if ‹even x› for x :: nat
  using that even-square-cong-4-int [of ‹int x›] by (simp flip: cong-int-iff)

lemma odd-square-cong-4-int:
  ‹[x^2 = 1] (mod 4)› if ‹odd x› for x :: int
proof −
  from that obtain y where ‹x = 2 * y + 1› ..
  then have ‹x^2 = 4 * (y^2 + y) + 1›
    by (simp add: power2-eq-square algebra-simps)
  also have ‹... mod 4 = ((4 * (y^2 + y)) mod 4 + 1 mod 4) mod 4›
    by (simp only: mod-simps)
  also have ‹... = 1 mod 4›
    by simp
  finally show ?thesis
    by (simp only: cong-def)
qed

lemma odd-square-cong-4-nat:
  ‹[x^2 = 1] (mod 4)› if ‹odd x› for x :: nat
  using that odd-square-cong-4-int [of ‹int x›] by (simp flip: cong-int-iff)
```

Gaussian integers will require a notion of an element being a power up to a unit, so we introduce this here. This should go in the library eventually.

definition *is-nth-power-upto-unit* **where**
is-nth-power-upto-unit n x $\longleftrightarrow$ ($\exists$ *u*. *is-unit u* $\wedge$ *is-nth-power n* (*u* $*$ *x*))

lemma *is-nth-power-upto-unit-base*: *is-nth-power n x* $\Longrightarrow$ *is-nth-power-upto-unit n x*
by (*auto simp*: *is-nth-power-upto-unit-def intro*: *exI*[*of* - *1*])

lemma *is-nth-power-upto-unitI*:
assumes *normalize* (*x* $\hat{}$ *n*) = *normalize y*
shows *is-nth-power-upto-unit n y*
proof −
from *associatedE1*[*OF assms*] **obtain** *u* **where** *is-unit u u* $*$ *y* = *x* $\hat{}$ *n*
by *metis*
thus *?thesis*
by (*auto simp*: *is-nth-power-upto-unit-def intro*!: *exI*[*of* - *u*])
qed

lemma *is-nth-power-upto-unit-conv-multiplicity*:
fixes *x* :: *'a* :: *factorial-semiring*
assumes *n* > *0*
shows *is-nth-power-upto-unit n x* $\longleftrightarrow$ ($\forall$ *p*. *prime p* $\longrightarrow$ *n dvd multiplicity p x*)
proof (*cases x* = *0*)
case *False*
show *?thesis*
proof *safe*
fix *p* :: *'a* **assume** *p*: *prime p*
assume *is-nth-power-upto-unit n x*
then obtain *u y* **where** *uy*: *is-unit u u* $*$ *x* = *y* $\hat{}$ *n*
by (*auto simp*: *is-nth-power-upto-unit-def elim*!: *is-nth-powerE*)
from *p uy assms False* **have** [*simp*]: *y* $\neq$ *0* **by** (*auto simp*: *power-0-left*)
have *multiplicity p* (*u* $*$ *x*) = *multiplicity p* (*y* $\hat{}$ *n*)
by (*subst uy*(*2*) [*symmetric*]) *simp*
also have *multiplicity p* (*u* $*$ *x*) = *multiplicity p x*
by (*simp add*: *multiplicity-times-unit-right uy*(*1*))
finally show *n dvd multiplicity p x*
using *False* **and** *p* **and** *uy* **and** *assms*
by (*auto simp*: *prime-elem-multiplicity-power-distrib*)
next
assume $*$: $\forall$ *p*. *prime p* $\longrightarrow$ *n dvd multiplicity p x*
have *multiplicity p* (($\prod$ *p*$\in$*prime-factors x*. *p* $\hat{}$ (*multiplicity p x div n*)) $\hat{}$ *n*) =
multiplicity p x **if** *prime p* **for** *p*
proof −
from *that* **and** $*$ **have** *n dvd multiplicity p x* **by** *blast*
have *multiplicity p x* = *0* **if** *p* $\notin$ *prime-factors x*
using *that* **and** ‹*prime p*› **by** (*simp add*: *prime-factors-multiplicity*)
with *that* **and** $*$ **and** *assms* **show** *?thesis* **unfolding** *prod-power-distrib power-mult* [*symmetric*]
by (*subst multiplicity-prod-prime-powers*) (*auto simp*: *in-prime-factors-imp-prime elim*: *dvdE*)

```
    qed
    with assms False
      have normalize ((∏ p∈prime-factors x. p ^ (multiplicity p x div n)) ^ n) =
normalize x
      by (intro multiplicity-eq-imp-eq) (auto simp: multiplicity-prod-prime-powers)
    thus is-nth-power-upto-unit n x
      by (auto intro: is-nth-power-upto-unitI)
  qed
qed (use assms in ‹auto simp: is-nth-power-upto-unit-def›)

lemma is-nth-power-upto-unit-0-left [simp, intro]: is-nth-power-upto-unit 0 x ⟷
is-unit x
proof
  assume is-unit x
  thus is-nth-power-upto-unit 0 x
    unfolding is-nth-power-upto-unit-def by (intro exI[of - 1 div x]) auto
next
  assume is-nth-power-upto-unit 0 x
  then obtain u where is-unit u u * x = 1
    by (auto simp: is-nth-power-upto-unit-def)
  thus is-unit x
    by (metis dvd-triv-right)
qed

lemma is-nth-power-upto-unit-unit [simp, intro]:
  assumes is-unit x
  shows   is-nth-power-upto-unit n x
  using assms by (auto simp: is-nth-power-upto-unit-def intro!: exI[of - 1 div x])

lemma is-nth-power-upto-unit-1-left [simp, intro]: is-nth-power-upto-unit 1 x
  by (auto simp: is-nth-power-upto-unit-def intro: exI[of - 1])

lemma is-nth-power-upto-unit-mult-coprimeD1:
  fixes x y :: 'a :: factorial-semiring
  assumes coprime x y is-nth-power-upto-unit n (x * y)
  shows   is-nth-power-upto-unit n x
proof −
  consider n = 0 | x = 0 n > 0 | x ≠ 0 y = 0 n > 0 | n > 0 x ≠ 0 y ≠ 0
    by force
  thus ?thesis
  proof cases
    assume [simp]: n = 0
    from assms have is-unit (x * y)
      by auto
    hence is-unit x
      using is-unit-mult-iff by blast
    thus ?thesis using assms by auto
  next
    assume x = 0 n > 0
```

```
    thus ?thesis by (auto simp: is-nth-power-upto-unit-def)
  next
    assume *: x ≠ 0 y = 0 n > 0
    with assms show ?thesis by auto
  next
    assume *: n > 0 and [simp]: x ≠ 0 y ≠ 0
    show ?thesis
    proof (subst is-nth-power-upto-unit-conv-multiplicity[OF ‹n > 0›]; safe)
      fix p :: 'a assume p: prime p
      show n dvd multiplicity p x
      proof (cases p dvd x)
        case False
        thus ?thesis
          by (simp add: not-dvd-imp-multiplicity-0)
      next
        case True
        have n dvd multiplicity p (x * y)
       using assms(2) ‹n > 0› p by (auto simp: is-nth-power-upto-unit-conv-multiplicity)
        also have ... = multiplicity p x + multiplicity p y
          using p by (subst prime-elem-multiplicity-mult-distrib) auto
        also have ¬p dvd y
          using ‹coprime x y› ‹p dvd x› p not-prime-unit coprime-common-divisor
by blast
        hence multiplicity p y = 0
          by (rule not-dvd-imp-multiplicity-0)
        finally show ?thesis by simp
      qed
    qed
  qed
qed

lemma is-nth-power-upto-unit-mult-coprimeD2:
  fixes x y :: 'a :: factorial-semiring
  assumes coprime x y is-nth-power-upto-unit n (x * y)
  shows   is-nth-power-upto-unit n y
  using assms is-nth-power-upto-unit-mult-coprimeD1[of y x]
  by (simp-all add: mult-ac coprime-commute)
```

1.2 Definition

Gaussian integers are the ring $\mathbb{Z}[i]$ which is formed either by formally adjoining an element i with $i^2 = -1$ to $\mathbb{Z}$ or by taking all the complex numbers with integer real and imaginary part.

We define them simply by giving an appropriate ring structure to $\mathbb{Z}^2$, with the first component representing the real part and the second component the imaginary part:

```
codatatype gauss-int = Gauss-Int (ReZ: int) (ImZ: int)
```

The following is the imaginary unit i in the Gaussian integers, which we will denote as $\mathrm{i}_{\mathbb{Z}}$:

primcorec *gauss-i* **where**
  ReZ gauss-i = *0*
| *ImZ gauss-i* = *1*

lemma *gauss-int-eq-iff*: $x = y \longleftrightarrow ReZ\ x = ReZ\ y \wedge ImZ\ x = ImZ\ y$
  by (*cases x*; *cases y*) *auto*

Next, we define the canonical injective homomorphism from the Gaussian integers into the complex numbers:

primcorec *gauss2complex* **where**
  Re (*gauss2complex z*) = *of-int* (*ReZ z*)
| *Im* (*gauss2complex z*) = *of-int* (*ImZ z*)

declare [[*coercion gauss2complex*]]

lemma *gauss2complex-eq-iff* [*simp*]: *gauss2complex z* = *gauss2complex u* $\longleftrightarrow$ *z* = *u*
  by (*simp add*: *complex-eq-iff gauss-int-eq-iff*)

Gaussian integers also have conjugates, just like complex numbers:

primcorec *gauss-cnj* **where**
  ReZ (*gauss-cnj z*) = *ReZ z*
| *ImZ* (*gauss-cnj z*) = $-$*ImZ z*

In the remainder of this section, we prove that Gaussian integers are a commutative ring of characteristic 0 and several other trivial algebraic properties.

instantiation *gauss-int* :: *comm-ring-1*
begin

primcorec *zero-gauss-int* **where**
  ReZ zero-gauss-int = *0*
| *ImZ zero-gauss-int* = *0*

primcorec *one-gauss-int* **where**
  ReZ one-gauss-int = *1*
| *ImZ one-gauss-int* = *0*

primcorec *uminus-gauss-int* **where**
  ReZ (*uminus-gauss-int x*) = $-$*ReZ x*
| *ImZ* (*uminus-gauss-int x*) = $-$*ImZ x*

primcorec *plus-gauss-int* **where**
  ReZ (*plus-gauss-int x y*) = *ReZ x* + *ReZ y*

| *ImZ* (*plus-gauss-int x y*) = *ImZ x* + *ImZ y*

primcorec *minus-gauss-int* **where**
  ReZ (*minus-gauss-int x y*) = *ReZ x* − *ReZ y*
| *ImZ* (*minus-gauss-int x y*) = *ImZ x* − *ImZ y*

primcorec *times-gauss-int* **where**
  ReZ (*times-gauss-int x y*) = *ReZ x* ∗ *ReZ y* − *ImZ x* ∗ *ImZ y*
| *ImZ* (*times-gauss-int x y*) = *ReZ x* ∗ *ImZ y* + *ImZ x* ∗ *ReZ y*

instance
  by *intro-classes* (*auto simp*: *gauss-int-eq-iff algebra-simps*)

end

lemma *gauss-i-times-i* [*simp*]: $\mathrm{i}_{\mathbb{Z}} * \mathrm{i}_{\mathbb{Z}} = (-1 :: \textit{gauss-int})$
  and *gauss-cnj-i* [*simp*]: *gauss-cnj* $\mathrm{i}_{\mathbb{Z}} = -\mathrm{i}_{\mathbb{Z}}$
  by (*simp-all add*: *gauss-int-eq-iff*)

lemma *gauss-cnj-eq-0-iff* [*simp*]: *gauss-cnj z* = *0* ⟷ *z* = *0*
  by (*auto simp*: *gauss-int-eq-iff*)

lemma *gauss-cnj-eq-self*: *Im z* = *0* ⟹ *gauss-cnj z* = *z*
  and *gauss-cnj-eq-minus-self*: *Re z* = *0* ⟹ *gauss-cnj z* = −*z*
  by (*auto simp*: *gauss-int-eq-iff*)

lemma *ReZ-of-nat* [*simp*]: *ReZ* (*of-nat n*) = *of-nat n*
  and *ImZ-of-nat* [*simp*]: *ImZ* (*of-nat n*) = *0*
  by (*induction n*; *simp*)+

lemma *ReZ-of-int* [*simp*]: *ReZ* (*of-int n*) = *n*
  and *ImZ-of-int* [*simp*]: *ImZ* (*of-int n*) = *0*
  by (*induction n*; *simp*)+

lemma *ReZ-numeral* [*simp*]: *ReZ* (*numeral n*) = *numeral n*
  and *ImZ-numeral* [*simp*]: *ImZ* (*numeral n*) = *0*
  by (*subst of-nat-numeral* [*symmetric*], *subst ReZ-of-nat ImZ-of-nat*, *simp*)+

lemma *gauss2complex-0* [*simp*]: *gauss2complex 0* = *0*
  and *gauss2complex-1* [*simp*]: *gauss2complex 1* = *1*
  and *gauss2complex-i* [*simp*]: *gauss2complex* $\mathrm{i}_{\mathbb{Z}}$ = i
  and *gauss2complex-add* [*simp*]: *gauss2complex* (*x* + *y*) = *gauss2complex x* + *gauss2complex y*
  and *gauss2complex-diff* [*simp*]: *gauss2complex* (*x* − *y*) = *gauss2complex x* − *gauss2complex y*
  and *gauss2complex-mult* [*simp*]: *gauss2complex* (*x* ∗ *y*) = *gauss2complex x* ∗ *gauss2complex y*
  and *gauss2complex-uminus* [*simp*]: *gauss2complex* (−*x*) = −*gauss2complex x*
  and *gauss2complex-cnj* [*simp*]: *gauss2complex* (*gauss-cnj x*) = *cnj* (*gauss2complex*

x)
by (*simp-all add*: *complex-eq-iff*)

lemma *gauss2complex-of-nat* [*simp*]: *gauss2complex* (*of-nat n*) = *of-nat n*
by (*simp add*: *complex-eq-iff*)

lemma *gauss2complex-eq-0-iff* [*simp*]: *gauss2complex x* = *0* ⟷ *x* = *0*
and *gauss2complex-eq-1-iff* [*simp*]: *gauss2complex x* = *1* ⟷ *x* = *1*
and *zero-eq-gauss2complex-iff* [*simp*]: *0* = *gauss2complex x* ⟷ *x* = *0*
and *one-eq-gauss2complex-iff* [*simp*]: *1* = *gauss2complex x* ⟷ *x* = *1*
by (*simp-all add*: *complex-eq-iff gauss-int-eq-iff*)

lemma *gauss-i-times-gauss-i-times* [*simp*]: $\mathrm{i}_{\mathbb{Z}} * (\mathrm{i}_{\mathbb{Z}} * x) = (-x :: \textit{gauss-int})$
by (*subst mult.assoc* [*symmetric*], *subst gauss-i-times-i*) *auto*

lemma *gauss-i-neq-0* [*simp*]: $\mathrm{i}_{\mathbb{Z}} \neq 0\ 0 \neq \mathrm{i}_{\mathbb{Z}}$
and *gauss-i-neq-1* [*simp*]: $\mathrm{i}_{\mathbb{Z}} \neq 1\ 1 \neq \mathrm{i}_{\mathbb{Z}}$
and *gauss-i-neq-of-nat* [*simp*]: $\mathrm{i}_{\mathbb{Z}} \neq$ *of-nat n of-nat n* $\neq \mathrm{i}_{\mathbb{Z}}$
and *gauss-i-neq-of-int* [*simp*]: $\mathrm{i}_{\mathbb{Z}} \neq$ *of-int n of-int n* $\neq \mathrm{i}_{\mathbb{Z}}$
and *gauss-i-neq-numeral* [*simp*]: $\mathrm{i}_{\mathbb{Z}} \neq$ *numeral m numeral m* $\neq \mathrm{i}_{\mathbb{Z}}$
by (*auto simp*: *gauss-int-eq-iff*)

lemma *gauss-cnj-0* [*simp*]: *gauss-cnj 0* = *0*
and *gauss-cnj-1* [*simp*]: *gauss-cnj 1* = *1*
and *gauss-cnj-cnj* [*simp*]: *gauss-cnj* (*gauss-cnj z*) = *z*
and *gauss-cnj-uminus* [*simp*]: *gauss-cnj* (−*a*) = −*gauss-cnj a*
and *gauss-cnj-add* [*simp*]: *gauss-cnj* (*a* + *b*) = *gauss-cnj a* + *gauss-cnj b*
and *gauss-cnj-diff* [*simp*]: *gauss-cnj* (*a* − *b*) = *gauss-cnj a* − *gauss-cnj b*
and *gauss-cnj-mult* [*simp*]: *gauss-cnj* (*a* ∗ *b*) = *gauss-cnj a* ∗ *gauss-cnj b*
and *gauss-cnj-of-nat* [*simp*]: *gauss-cnj* (*of-nat n1*) = *of-nat n1*
and *gauss-cnj-of-int* [*simp*]: *gauss-cnj* (*of-int n2*) = *of-int n2*
and *gauss-cnj-numeral* [*simp*]: *gauss-cnj* (*numeral n3*) = *numeral n3*
by (*simp-all add*: *gauss-int-eq-iff*)

lemma *gauss-cnj-power* [*simp*]: *gauss-cnj* (*a* ^ *n*) = *gauss-cnj a* ^ *n*
by (*induction n*) *auto*

lemma *gauss-cnj-sum* [*simp*]: *gauss-cnj* (*sum f A*) = $(\sum x{\in}A.$ *gauss-cnj* (*f x*))
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *gauss-cnj-prod* [*simp*]: *gauss-cnj* (*prod f A*) = $(\prod x{\in}A.$ *gauss-cnj* (*f x*))
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *of-nat-dvd-of-nat*:
assumes *a dvd b*
shows *of-nat a dvd* (*of-nat b* :: *'a* :: *comm-semiring-1*)
using *assms* **by** *auto*

lemma *of-int-dvd-imp-dvd-gauss-cnj*:

fixes z :: *gauss-int*
assumes *of-int n dvd z*
shows *of-int n dvd gauss-cnj z*
proof −
from *assms* **obtain** u **where** $z = \textit{of-int}\ n * u$ **by** *blast*
hence *gauss-cnj z* = *of-int n* ∗ *gauss-cnj u*
by *simp*
thus *?thesis* **by** *auto*
qed

lemma *of-nat-dvd-imp-dvd-gauss-cnj*:
fixes z :: *gauss-int*
assumes *of-nat n dvd z*
shows *of-nat n dvd gauss-cnj z*
using *of-int-dvd-imp-dvd-gauss-cnj*[*of int n*] *assms* **by** *simp*

lemma *of-int-dvd-of-int-gauss-int-iff*:
(*of-int m* :: *gauss-int*) *dvd of-int n* ⟷ *m dvd n*
proof
assume *of-int m dvd* (*of-int n* :: *gauss-int*)
then obtain a :: *gauss-int* **where** *of-int n* = *of-int m* ∗ *a*
by *blast*
thus *m dvd n*
by (*auto simp*: *gauss-int-eq-iff*)
qed *auto*

lemma *of-nat-dvd-of-nat-gauss-int-iff*:
(*of-nat m* :: *gauss-int*) *dvd of-nat n* ⟷ *m dvd n*
using *of-int-dvd-of-int-gauss-int-iff*[*of int m int n*] **by** *simp*

lemma *gauss-cnj-dvd*:
assumes *a dvd b*
shows *gauss-cnj a dvd gauss-cnj b*
proof −
from *assms* **obtain** c **where** $b = a * c$
by *blast*
hence *gauss-cnj b* = *gauss-cnj a* ∗ *gauss-cnj c*
by *simp*
thus *?thesis* **by** *auto*
qed

lemma *gauss-cnj-dvd-iff*: *gauss-cnj a dvd gauss-cnj b* ⟷ *a dvd b*
using *gauss-cnj-dvd*[*of a b*] *gauss-cnj-dvd*[*of gauss-cnj a gauss-cnj b*] **by** *auto*

lemma *gauss-cnj-dvd-left-iff*: *gauss-cnj a dvd b* ⟷ *a dvd gauss-cnj b*
by (*subst gauss-cnj-dvd-iff* [*symmetric*]) *auto*

lemma *gauss-cnj-dvd-right-iff*: *a dvd gauss-cnj b* ⟷ *gauss-cnj a dvd b*
by (*rule gauss-cnj-dvd-left-iff* [*symmetric*])

instance *gauss-int* :: *idom*
proof
  fix *z u* :: *gauss-int*
  assume $z \neq 0$ $u \neq 0$
  hence *gauss2complex z* $*$ *gauss2complex u* $\neq 0$
    by *simp*
  also have *gauss2complex z* $*$ *gauss2complex u* $=$ *gauss2complex* $(z * u)$
    by *simp*
  finally show $z * u \neq 0$
    unfolding *gauss2complex-eq-0-iff* **.**
qed

instance *gauss-int* :: *ring-char-0*
  by *intro-classes* (*auto intro*!: *injI simp*: *gauss-int-eq-iff*)

1.3 Pretty-printing

The following lemma collection provides better pretty-printing of Gaussian integers so that e.g. evaluation with the 'value' command produces nicer results.

lemma *gauss-int-code-post* [*code-post*]:
  Gauss-Int 0 0 $= 0$
  Gauss-Int 0 1 $= \mathrm{i}_{\mathbb{Z}}$
  Gauss-Int 0 $(-1) = -\mathrm{i}_{\mathbb{Z}}$
  Gauss-Int 1 0 $= 1$
  Gauss-Int 1 1 $= 1 + \mathrm{i}_{\mathbb{Z}}$
  Gauss-Int 1 $(-1) = 1 - \mathrm{i}_{\mathbb{Z}}$
  Gauss-Int (-1) *0* $= -1$
  Gauss-Int (-1) *1* $= -1 + \mathrm{i}_{\mathbb{Z}}$
  Gauss-Int (-1) $(-1) = -1 - \mathrm{i}_{\mathbb{Z}}$
  Gauss-Int (*numeral b*) *0* $=$ *numeral b*
  Gauss-Int $(-$*numeral b*$)$ *0* $= -$*numeral b*
  Gauss-Int (*numeral b*) *1* $=$ *numeral b* $+ \mathrm{i}_{\mathbb{Z}}$
  Gauss-Int $(-$*numeral b*$)$ *1* $= -$*numeral b* $+ \mathrm{i}_{\mathbb{Z}}$
  Gauss-Int (*numeral b*) $(-1) =$ *numeral b* $- \mathrm{i}_{\mathbb{Z}}$
  Gauss-Int $(-$*numeral b*$)$ $(-1) = -$*numeral b* $- \mathrm{i}_{\mathbb{Z}}$
  Gauss-Int 0 (*numeral b*) $=$ *numeral b* $* \mathrm{i}_{\mathbb{Z}}$
  Gauss-Int 0 $(-$*numeral b*$) = -$*numeral b* $* \mathrm{i}_{\mathbb{Z}}$
  Gauss-Int 1 (*numeral b*) $= 1 +$ *numeral b* $* \mathrm{i}_{\mathbb{Z}}$
  Gauss-Int 1 $(-$*numeral b*$) = 1 -$ *numeral b* $* \mathrm{i}_{\mathbb{Z}}$
  Gauss-Int (-1) (*numeral b*) $= -1 +$ *numeral b* $* \mathrm{i}_{\mathbb{Z}}$
  Gauss-Int (-1) $(-$*numeral b*$) = -1 -$ *numeral b* $* \mathrm{i}_{\mathbb{Z}}$
  Gauss-Int (*numeral a*) (*numeral b*) $=$ *numeral a* $+$ *numeral b* $* \mathrm{i}_{\mathbb{Z}}$
  Gauss-Int (*numeral a*) $(-$*numeral b*$) =$ *numeral a* $-$ *numeral b* $* \mathrm{i}_{\mathbb{Z}}$
  Gauss-Int $(-$*numeral a*$)$ (*numeral b*) $= -$*numeral a* $+$ *numeral b* $* \mathrm{i}_{\mathbb{Z}}$
  Gauss-Int $(-$*numeral a*$)$ $(-$*numeral b*$) = -$*numeral a* $-$ *numeral b* $* \mathrm{i}_{\mathbb{Z}}$
  by (*simp-all add*: *gauss-int-eq-iff*)

value $\mathrm{i}_{\mathbb{Z}} \mathbin{\hat{}} 3$
value $2 * (3 + \mathrm{i}_{\mathbb{Z}})$
value $(2 + \mathrm{i}_{\mathbb{Z}}) * (2 - \mathrm{i}_{\mathbb{Z}})$

1.4 Norm

The square of the complex norm (or complex modulus) on the Gaussian integers gives us a norm that always returns a natural number. We will later show that this is also a Euclidean norm (in the sense of a Euclidean ring).

definition *gauss-int-norm* :: *gauss-int* ⇒ *nat* **where**
  gauss-int-norm z = *nat* (*ReZ z* ^ 2 + *ImZ z* ^ 2)

lemma *gauss-int-norm-0* [*simp*]: *gauss-int-norm 0* = *0*
  and *gauss-int-norm-1* [*simp*]: *gauss-int-norm 1* = *1*
  and *gauss-int-norm-i* [*simp*]: *gauss-int-norm* $\mathrm{i}_{\mathbb{Z}}$ = *1*
  and *gauss-int-norm-cnj* [*simp*]: *gauss-int-norm* (*gauss-cnj z*) = *gauss-int-norm z*
  and *gauss-int-norm-of-nat* [*simp*]: *gauss-int-norm* (*of-nat n*) = *n* ^ 2
  and *gauss-int-norm-of-int* [*simp*]: *gauss-int-norm* (*of-int m*) = *nat* (*m* ^ 2)
  and *gauss-int-norm-of-numeral* [*simp*]: *gauss-int-norm* (*numeral n'*) = *numeral* (*Num.sqr n'*)
  by (*simp-all add*: *gauss-int-norm-def nat-power-eq*)

lemma *gauss-int-norm-uminus* [*simp*]: *gauss-int-norm* (−*z*) = *gauss-int-norm z*
  by (*simp add*: *gauss-int-norm-def*)

lemma *gauss-int-norm-eq-0-iff* [*simp*]: *gauss-int-norm z* = *0* ⟷ *z* = *0*
proof
  assume *gauss-int-norm z* = *0*
  hence *ReZ z* ^ 2 + *ImZ z* ^ 2 ≤ *0*
    by (*simp add*: *gauss-int-norm-def*)
  moreover have *ReZ z* ^ 2 + *ImZ z* ^ 2 ≥ *0*
    by *simp*
  ultimately have *ReZ z* ^ 2 + *ImZ z* ^ 2 = *0*
    by *linarith*
  thus *z* = *0*
    by (*auto simp*: *gauss-int-eq-iff*)
qed *auto*

lemma *gauss-int-norm-pos-iff* [*simp*]: *gauss-int-norm z* > *0* ⟷ *z* ≠ *0*
  using *gauss-int-norm-eq-0-iff*[*of z*] **by** (*auto intro*: *Nat.gr0I*)

lemma *real-gauss-int-norm*: *real* (*gauss-int-norm z*) = *norm* (*gauss2complex z*) ^ 2
  by (*auto simp*: *cmod-def gauss-int-norm-def*)

lemma *gauss-int-norm-mult*: *gauss-int-norm* (*z* ∗ *u*) = *gauss-int-norm z* ∗ *gauss-int-norm*

u
proof −
have *real* (*gauss-int-norm* (*z* ∗ *u*)) = *real* (*gauss-int-norm z* ∗ *gauss-int-norm u*)
unfolding *of-nat-mult* **by** (*simp add*: *real-gauss-int-norm norm-power norm-mult power-mult-distrib*)
thus *?thesis* **by** (*subst* (*asm*) *of-nat-eq-iff*)
qed

lemma *self-mult-gauss-cnj*: *z* ∗ *gauss-cnj z* = *of-nat* (*gauss-int-norm z*)
by (*simp add*: *gauss-int-norm-def gauss-int-eq-iff algebra-simps power2-eq-square*)

lemma *gauss-cnj-mult-self*: *gauss-cnj z* ∗ *z* = *of-nat* (*gauss-int-norm z*)
by (*subst mult.commute*, *rule self-mult-gauss-cnj*)

lemma *self-plus-gauss-cnj*: *z* + *gauss-cnj z* = *of-int* (2 ∗ *ReZ z*)
and *self-minus-gauss-cnj*: *z* − *gauss-cnj z* = *of-int* (2 ∗ *ImZ z*) ∗ $\mathrm{i}_{\mathbb{Z}}$
by (*auto simp*: *gauss-int-eq-iff*)

lemma *gauss-int-norm-dvd-mono*:
assumes *a dvd b*
shows *gauss-int-norm a dvd gauss-int-norm b*
proof −
from *assms* **obtain** *c* **where** *b* = *a* ∗ *c* **by** *blast*
hence *gauss-int-norm b* = *gauss-int-norm* (*a* ∗ *c*)
by *metis*
thus *?thesis* **by** (*simp add*: *gauss-int-norm-mult*)
qed

A Gaussian integer is a unit iff its norm is 1, and this is the case precisely for the four elements ±*1* and ±i:

lemma *is-unit-gauss-int-iff*: *x dvd 1* ⟷ *x* ∈ {*1*, −*1*, $\mathrm{i}_{\mathbb{Z}}$, $-\mathrm{i}_{\mathbb{Z}}$ :: *gauss-int*}
and *is-unit-gauss-int-iff′*: *x dvd 1* ⟷ *gauss-int-norm x* = *1*
proof −
have *x dvd 1* **if** *x* ∈ {*1*, −*1*, $\mathrm{i}_{\mathbb{Z}}$, $-\mathrm{i}_{\mathbb{Z}}$}
proof −
from *that* **have** ∗: *x* ∗ *gauss-cnj x* = *1*
by (*auto simp*: *gauss-int-norm-def*)
show *x dvd 1* **by** (*subst* ∗ [*symmetric*]) *simp*
qed
moreover have *gauss-int-norm x* = *1* **if** *x dvd 1*
using *gauss-int-norm-dvd-mono*[*OF that*] **by** *simp*
moreover have *x* ∈ {*1*, −*1*, $\mathrm{i}_{\mathbb{Z}}$, $-\mathrm{i}_{\mathbb{Z}}$} **if** *gauss-int-norm x* = *1*
proof −
from *that* **have** ∗: (*ReZ x*)^2 + (*ImZ x*)^2 = *1*
by (*auto simp*: *gauss-int-norm-def nat-eq-iff*)
hence *ReZ x* ^ *2* ≤ *1* **and** *ImZ x* ^ *2* ≤ *1*
using *zero-le-power2*[*of ImZ x*] *zero-le-power2*[*of ReZ x*] **by** *linarith*+
hence |*ReZ x*| ≤ *1* **and** |*ImZ x*| ≤ *1*
by (*auto simp*: *abs-square-le-1*)

```
    hence ReZ x ∈ {−1, 0, 1} and ImZ x ∈ {−1, 0, 1}
      by auto
    thus x ∈ {1, −1, iℤ, −iℤ :: gauss-int}
      using ∗ by (auto simp: gauss-int-eq-iff)
  qed
  ultimately show x dvd 1 ⟷ x ∈ {1, −1, iℤ, −iℤ :: gauss-int}
            and x dvd 1 ⟷ gauss-int-norm x = 1
    by blast+
qed

lemma is-unit-gauss-i [simp, intro]: (gauss-i :: gauss-int) dvd 1
  by (simp add: is-unit-gauss-int-iff)

lemma gauss-int-norm-eq-Suc-0-iff: gauss-int-norm x = Suc 0 ⟷ x dvd 1
  by (simp add: is-unit-gauss-int-iff')

lemma is-unit-gauss-cnj [intro]: z dvd 1 ⟹ gauss-cnj z dvd 1
  by (simp add: is-unit-gauss-int-iff')

lemma is-unit-gauss-cnj-iff [simp]: gauss-cnj z dvd 1 ⟷ z dvd 1
  by (simp add: is-unit-gauss-int-iff')
```

1.5 Division and normalisation

We define a rounding operation that takes a complex number and returns a Gaussian integer by rounding the real and imaginary parts separately:

```
primcorec round-complex :: complex ⇒ gauss-int where
  ReZ (round-complex z) = round (Re z)
| ImZ (round-complex z) = round (Im z)
```

The distance between a rounded complex number and the original one is no more than $\frac{1}{2}\sqrt{2}$:

```
lemma norm-round-complex-le: norm (z − gauss2complex (round-complex z)) ^ 2
≤ 1 / 2
proof −
  have (Re z − ReZ (round-complex z)) ^ 2 ≤ (1 / 2) ^ 2
    using of-int-round-abs-le[of Re z]
    by (subst abs-le-square-iff [symmetric]) (auto simp: abs-minus-commute)
  moreover have (Im z − ImZ (round-complex z)) ^ 2 ≤ (1 / 2) ^ 2
    using of-int-round-abs-le[of Im z]
    by (subst abs-le-square-iff [symmetric]) (auto simp: abs-minus-commute)
  ultimately have (Re z − ReZ (round-complex z)) ^ 2 + (Im z − ImZ (round-complex
z)) ^ 2 ≤
                  (1 / 2) ^ 2 + (1 / 2) ^ 2
    by (rule add-mono)
  thus norm (z − gauss2complex (round-complex z)) ^ 2 ≤ 1 / 2
    by (simp add: cmod-def power2-eq-square)
qed
```

lemma *dist-round-complex-le*: *dist z* (*gauss2complex* (*round-complex z*)) ≤ *sqrt 2 / 2*
proof −
  have *dist z* (*gauss2complex* (*round-complex z*)) ^ *2* =
      norm (*z* − *gauss2complex* (*round-complex z*)) ^ *2*
    by (*simp add*: *dist-norm*)
  also have ... ≤ *1 / 2*
    by (*rule norm-round-complex-le*)
  also have ... = (*sqrt 2 / 2*) ^ *2*
    by (*simp add*: *power2-eq-square*)
  finally show *?thesis*
    by (*rule power2-le-imp-le*) *auto*
qed

We can now define division on Gaussian integers simply by performing the division in the complex numbers and rounding the result. This also gives us a remainder operation defined accordingly for which the norm of the remainder is always smaller than the norm of the divisor.

We can also define a normalisation operation that returns a canonical representative for each association class. Since the four units of the Gaussian integers are ±*1* and ±i, each association class (other than *0*) has four representatives, one in each quadrant. We simply define the on in the upper-right quadrant (i.e. the one with non-negative imaginary part and positive real part) as the canonical one.

Thus, the Gaussian integers form a Euclidean ring. This gives us many things, most importantly the existence of GCDs and LCMs and unique factorisation.

instantiation *gauss-int* :: *algebraic-semidom*
begin

definition *divide-gauss-int* :: *gauss-int* ⇒ *gauss-int* ⇒ *gauss-int* **where**
  divide-gauss-int a b = *round-complex* (*gauss2complex a* / *gauss2complex b*)

instance proof
  fix *a* :: *gauss-int*
  show *a div 0* = *0*
    by (*auto simp*: *gauss-int-eq-iff divide-gauss-int-def*)
next
  fix *a b* :: *gauss-int* **assume** *b* ≠ *0*
  thus *a* ∗ *b div b* = *a*
    by (*auto simp*: *gauss-int-eq-iff divide-gauss-int-def*)
qed

end

instantiation *gauss-int* :: *semidom-divide-unit-factor*

begin

definition *unit-factor-gauss-int* :: *gauss-int* ⇒ *gauss-int* **where**
unit-factor-gauss-int z =
(*if z* = *0 then 0 else*
if ImZ z ≥ *0* ∧ *ReZ z* > *0 then 1*
else if ReZ z ≤ *0* ∧ *ImZ z* > *0 then* $\mathrm{i}_{\mathbb{Z}}$
else if ImZ z ≤ *0* ∧ *ReZ z* < *0 then* −*1*
else $-\mathrm{i}_{\mathbb{Z}}$)

instance proof
show *unit-factor* (*0* :: *gauss-int*) = *0*
by (*simp add*: *unit-factor-gauss-int-def*)
next
fix *z* :: *gauss-int*
assume *is-unit z*
thus *unit-factor z* = *z*
by (*subst* (*asm*) *is-unit-gauss-int-iff*) (*auto simp*: *unit-factor-gauss-int-def*)
next
fix *z* :: *gauss-int*
assume *z*: *z* ≠ *0*
thus *is-unit* (*unit-factor z*)
by (*subst is-unit-gauss-int-iff*) (*auto simp*: *unit-factor-gauss-int-def*)
next
fix *z u* :: *gauss-int*
assume *is-unit z*
hence *z* ∈ {*1*, −*1*, $\mathrm{i}_{\mathbb{Z}}$, $-\mathrm{i}_{\mathbb{Z}}$}
by (*subst* (*asm*) *is-unit-gauss-int-iff*)
thus *unit-factor* (*z* ∗ *u*) = *z* ∗ *unit-factor u*
by (*safe*; *auto simp*: *unit-factor-gauss-int-def gauss-int-eq-iff*[*of u 0*])
qed

end

instantiation *gauss-int* :: *normalization-semidom*
begin

definition *normalize-gauss-int* :: *gauss-int* ⇒ *gauss-int* **where**
normalize-gauss-int z =
(*if z* = *0 then 0 else*
if ImZ z ≥ *0* ∧ *ReZ z* > *0 then z*
else if ReZ z ≤ *0* ∧ *ImZ z* > *0 then* $-\mathrm{i}_{\mathbb{Z}} * z$
else if ImZ z ≤ *0* ∧ *ReZ z* < *0 then* −*z*
else $\mathrm{i}_{\mathbb{Z}} * z$)

instance proof
show *normalize* (*0* :: *gauss-int*) = *0*
by (*simp add*: *normalize-gauss-int-def*)
next

fix z :: *gauss-int*
show *unit-factor* $z * normalize\ z = z$
by (*auto simp*: *normalize-gauss-int-def unit-factor-gauss-int-def algebra-simps*)
qed

end

lemma *normalize-gauss-int-of-nat* [*simp*]: *normalize* (*of-nat* n :: *gauss-int*) = *of-nat* n
and *normalize-gauss-int-of-int* [*simp*]: *normalize* (*of-int* m :: *gauss-int*) = *of-int* $|m|$
and *normalize-gauss-int-of-numeral* [*simp*]: *normalize* (*numeral* n' :: *gauss-int*) = *numeral* n'
by (*auto simp*: *normalize-gauss-int-def*)

lemma *normalize-gauss-i* [*simp*]: *normalize* $\mathrm{i}_{\mathbb{Z}} = 1$
by (*simp add*: *normalize-gauss-int-def*)

lemma *gauss-int-norm-normalize* [*simp*]: *gauss-int-norm* (*normalize* x) = *gauss-int-norm* x
by (*simp add*: *normalize-gauss-int-def gauss-int-norm-mult*)

lemma *normalized-gauss-int*:
assumes *normalize* $z = z$
shows $ReZ\ z \geq 0\ ImZ\ z \geq 0$
using *assms*
by (*cases ReZ z 0* :: *int rule*: *linorder-cases*;
cases ImZ z 0 :: *int rule*: *linorder-cases*;
simp add: *normalize-gauss-int-def gauss-int-eq-iff*)+

lemma *normalized-gauss-int'*:
assumes *normalize* $z = z\ z \neq 0$
shows $ReZ\ z > 0\ ImZ\ z \geq 0$
using *assms*
by (*cases ReZ z 0* :: *int rule*: *linorder-cases*;
cases ImZ z 0 :: *int rule*: *linorder-cases*;
simp add: *normalize-gauss-int-def gauss-int-eq-iff*)+

lemma *normalized-gauss-int-iff*:
$normalize\ z = z \longleftrightarrow z = 0 \vee ReZ\ z > 0 \wedge ImZ\ z \geq 0$
by (*cases ReZ z 0* :: *int rule*: *linorder-cases*;
cases ImZ z 0 :: *int rule*: *linorder-cases*;
simp add: *normalize-gauss-int-def gauss-int-eq-iff*)+

instantiation *gauss-int* :: *idom-modulo*
begin

definition *modulo-gauss-int* :: *gauss-int* ⇒ *gauss-int* ⇒ *gauss-int* **where**
modulo-gauss-int $a\ b = a - a\ div\ b * b$

instance proof
fix *a b* :: *gauss-int*
show *a div b * b + a mod b = a*
by (*simp add*: *modulo-gauss-int-def*)
qed

end

lemma *gauss-int-norm-mod-less-aux*:
assumes [*simp*]: *b ≠ 0*
shows *2 * gauss-int-norm (a mod b) ≤ gauss-int-norm b*
proof −
define *a' b'* **where** *a' = gauss2complex a* **and** *b' = gauss2complex b*
have [*simp*]: *b' ≠ 0* **by** (*simp add*: *b'-def*)
have *gauss-int-norm (a mod b) =*
*norm (gauss2complex (a − round-complex (a' / b') * b)) ^ 2*
unfolding *modulo-gauss-int-def*
by (*subst real-gauss-int-norm* [*symmetric*]) (*auto simp add*: *divide-gauss-int-def a'-def b'-def*)
also have *gauss2complex (a − round-complex (a' / b') * b) =*
*a' − gauss2complex (round-complex (a' / b')) * b'*
by (*simp add*: *a'-def b'-def*)
also have *... = (a' / b' − gauss2complex (round-complex (a' / b'))) * b'*
by (*simp add*: *field-simps*)
also have *norm ... ^ 2 = norm (a' / b' − gauss2complex (round-complex (a' / b'))) ^ 2 * norm b' ^ 2*
by (*simp add*: *norm-mult power-mult-distrib*)
also have *... ≤ 1 / 2 * norm b' ^ 2*
by (*intro mult-right-mono norm-round-complex-le*) *auto*
also have *norm b' ^ 2 = gauss-int-norm b*
by (*simp add*: *b'-def real-gauss-int-norm*)
finally show *?thesis* **by** *linarith*
qed

lemma *gauss-int-norm-mod-less*:
assumes [*simp*]: *b ≠ 0*
shows *gauss-int-norm (a mod b) < gauss-int-norm b*
proof −
have *gauss-int-norm b > 0* **by** *simp*
thus *gauss-int-norm (a mod b) < gauss-int-norm b*
using *gauss-int-norm-mod-less-aux*[*OF assms*, *of a*] **by** *presburger*
qed

lemma *gauss-int-norm-dvd-imp-le*:
assumes *b ≠ 0*
shows *gauss-int-norm a ≤ gauss-int-norm (a * b)*
proof (*cases a = 0*)
case *False*

thus *?thesis* **using** *assms* **by** (*intro dvd-imp-le gauss-int-norm-dvd-mono*) *auto*
qed *auto*

instantiation *gauss-int* :: *euclidean-ring*
begin

definition *euclidean-size-gauss-int* :: *gauss-int* ⇒ *nat* **where**
[*simp*]: *euclidean-size-gauss-int* = *gauss-int-norm*

instance proof
show *euclidean-size* (*0* :: *gauss-int*) = *0*
by *simp*
next
fix *a b* :: *gauss-int* **assume** [*simp*]: *b* ≠ *0*
show *euclidean-size* (*a mod b*) < *euclidean-size b*
using *gauss-int-norm-mod-less*[*of b a*] **by** *simp*
show *euclidean-size a* ≤ *euclidean-size* (*a* ∗ *b*)
by (*simp add*: *gauss-int-norm-dvd-imp-le*)
qed

end

instance *gauss-int* :: *normalization-euclidean-semiring* **..**

instantiation *gauss-int* :: *euclidean-ring-gcd*
begin

definition *gcd-gauss-int* :: *gauss-int* ⇒ *gauss-int* ⇒ *gauss-int* **where**
gcd-gauss-int ≡ *normalization-euclidean-semiring-class.gcd*
definition *lcm-gauss-int* :: *gauss-int* ⇒ *gauss-int* ⇒ *gauss-int* **where**
lcm-gauss-int ≡ *normalization-euclidean-semiring-class.lcm*
definition *Gcd-gauss-int* :: *gauss-int set* ⇒ *gauss-int* **where**
Gcd-gauss-int ≡ *normalization-euclidean-semiring-class.Gcd*
definition *Lcm-gauss-int* :: *gauss-int set* ⇒ *gauss-int* **where**
Lcm-gauss-int ≡ *normalization-euclidean-semiring-class.Lcm*

instance
by *intro-classes*
(*simp-all add*: *gcd-gauss-int-def lcm-gauss-int-def Gcd-gauss-int-def Lcm-gauss-int-def*)

end

lemma *multiplicity-gauss-cnj*: *multiplicity* (*gauss-cnj a*) (*gauss-cnj b*) = *multiplicity a b*
unfolding *multiplicity-def gauss-cnj-power* [*symmetric*] *gauss-cnj-dvd-iff* **..**

lemma *multiplicity-gauss-int-of-nat*:
multiplicity (*of-nat a*) (*of-nat b* :: *gauss-int*) = *multiplicity a b*
unfolding *multiplicity-def of-nat-power* [*symmetric*] *of-nat-dvd-of-nat-gauss-int-iff*

..

lemma *gauss-int-dvd-same-norm-imp-associated*:
assumes *z1 dvd z2 gauss-int-norm z1 = gauss-int-norm z2*
shows *normalize z1 = normalize z2*
proof (*cases z1 = 0*)
case [*simp*]: *False*
from *assms*(*1*) **obtain** *u* **where** *u*: *z2 = z1 * u* **by** *blast*
from *assms* **have** *gauss-int-norm u = 1*
by (*auto simp*: *gauss-int-norm-mult u*)
hence *is-unit u*
by (*simp add*: *is-unit-gauss-int-iff'*)
with *u* **show** *?thesis* **by** *simp*
qed (*use assms* **in** *auto*)

lemma *gcd-of-int-gauss-int*: *gcd* (*of-int a* :: *gauss-int*) (*of-int b*) = *of-int* (*gcd a b*)
proof (*induction nat* |*b*| *arbitrary*: *a b rule*: *less-induct*)
case (*less b a*)
show *?case*
proof (*cases b = 0*)
case *False*
have *of-int* (*gcd a b*) = (*of-int* (*gcd b* (*a mod b*)) :: *gauss-int*)
by (*subst gcd-red-int*) *auto*
also have ... = *gcd* (*of-int b*) (*of-int* (*a mod b*))
using *False* **by** (*intro less* [*symmetric*]) (*auto intro*!: *abs-mod-less*)
also have *a mod b* = (*a* − *a div b* * *b*)
by (*simp add*: *minus-div-mult-eq-mod*)
also have *of-int* ... = *of-int* (−(*a div b*)) * *of-int b* + (*of-int a* :: *gauss-int*)
by (*simp add*: *algebra-simps*)
also have *gcd* (*of-int b*) ... = *gcd* (*of-int b*) (*of-int a*)
by (*rule gcd-add-mult*)
finally show *?thesis* **by** (*simp add*: *gcd.commute*)
qed *auto*
qed

lemma *coprime-of-int-gauss-int*: *coprime* (*of-int a* :: *gauss-int*) (*of-int b*) = *coprime a b*
unfolding *coprime-iff-gcd-eq-1 gcd-of-int-gauss-int* **by** *auto*

lemma *gcd-of-nat-gauss-int*: *gcd* (*of-nat a* :: *gauss-int*) (*of-nat b*) = *of-nat* (*gcd a b*)
using *gcd-of-int-gauss-int*[*of int a int b*] **by** *simp*

lemma *coprime-of-nat-gauss-int*: *coprime* (*of-nat a* :: *gauss-int*) (*of-nat b*) = *coprime a b*
unfolding *coprime-iff-gcd-eq-1 gcd-of-nat-gauss-int* **by** *auto*

lemma *gauss-cnj-dvd-self-iff*: *gauss-cnj z dvd z* ⟷ *ReZ z = 0* ∨ *ImZ z = 0* ∨ |*ReZ z*| = |*ImZ z*|

proof
assume *gauss-cnj z dvd z*
hence *normalize* (*gauss-cnj z*) = *normalize z*
by (*rule gauss-int-dvd-same-norm-imp-associated*) *auto*
then obtain *u* :: *gauss-int* **where** *is-unit u* **and** *u*: *gauss-cnj z* = *u* ∗ *z*
using *associatedE1* **by** *blast*
hence *u* ∈ {*1*, −*1*, $\mathrm{i}_{\mathbb{Z}}$, −$\mathrm{i}_{\mathbb{Z}}$}
by (*simp add*: *is-unit-gauss-int-iff*)
thus *ReZ z* = *0* ∨ *ImZ z* = *0* ∨ |*ReZ z*| = |*ImZ z*|
proof (*elim insertE emptyE*)
assume [*simp*]: *u* = $\mathrm{i}_{\mathbb{Z}}$
have *ReZ z* = *ReZ* (*gauss-cnj z*)
by *simp*
also have *gauss-cnj z* = $\mathrm{i}_{\mathbb{Z}}$ ∗ *z*
using *u* **by** *simp*
also have *ReZ* ... = −*ImZ z*
by *simp*
finally show *ReZ z* = *0* ∨ *ImZ z* = *0* ∨ |*ReZ z*| = |*ImZ z*|
by *auto*
next
assume [*simp*]: *u* = −$\mathrm{i}_{\mathbb{Z}}$
have *ReZ z* = *ReZ* (*gauss-cnj z*)
by *simp*
also have *gauss-cnj z* = −$\mathrm{i}_{\mathbb{Z}}$ ∗ *z*
using *u* **by** *simp*
also have *ReZ* ... = *ImZ z*
by *simp*
finally show *ReZ z* = *0* ∨ *ImZ z* = *0* ∨ |*ReZ z*| = |*ImZ z*|
by *auto*
next
assume [*simp*]: *u* = *1*
have *ImZ z* = −*ImZ* (*gauss-cnj z*)
by *simp*
also have *gauss-cnj z* = *z*
using *u* **by** *simp*
finally show *ReZ z* = *0* ∨ *ImZ z* = *0* ∨ |*ReZ z*| = |*ImZ z*|
by *auto*
next
assume [*simp*]: *u* = −*1*
have *ReZ z* = *ReZ* (*gauss-cnj z*)
by *simp*
also have *gauss-cnj z* = −*z*
using *u* **by** *simp*
also have *ReZ* ... = −*ReZ z*
by *simp*
finally show *ReZ z* = *0* ∨ *ImZ z* = *0* ∨ |*ReZ z*| = |*ImZ z*|
by *auto*
qed
next

assume $ReZ\ z = 0 \vee ImZ\ z = 0 \vee |ReZ\ z| = |ImZ\ z|$
thus *gauss-cnj z dvd z*
proof *safe*
assume $|ReZ\ z| = |ImZ\ z|$
then obtain $u :: int$ **where** *is-unit u* **and** u: $ImZ\ z = u * ReZ\ z$
using *associatedE2*[*of ReZ z ImZ z*] **by** *auto*
from ‹*is-unit u*› **have** $u \in \{1, -1\}$
by *auto*
hence $z = gauss\text{-}cnj\ z * (of\text{-}int\ u * \mathrm{i}_{\mathbb{Z}})$
using *u* **by** (*auto simp*: *gauss-int-eq-iff*)
thus *?thesis*
by (*metis dvd-triv-left*)
qed (*auto simp*: *gauss-cnj-eq-self gauss-cnj-eq-minus-self*)
qed

lemma *self-dvd-gauss-cnj-iff*: $z\ dvd\ gauss\text{-}cnj\ z \longleftrightarrow ReZ\ z = 0 \vee ImZ\ z = 0 \vee |ReZ\ z| = |ImZ\ z|$
using *gauss-cnj-dvd-self-iff*[*of z*] **by** (*subst* (*asm*) *gauss-cnj-dvd-left-iff*) *auto*

1.6 Prime elements

Next, we analyse what the prime elements of the Gaussian integers are. First, note that according to the conventions of Isabelle's computational algebra library, a prime element is called a prime iff it is also normalised, i.e. in our case it lies in the upper right quadrant.

As a first fact, we can show that a Gaussian integer whose norm is $\mathbb{Z}$-prime must be $\mathbb{Z}[i]$-prime:

lemma *prime-gauss-int-norm-imp-prime-elem*:
assumes *prime* (*gauss-int-norm q*)
shows *prime-elem q*
proof −
have *irreducible q*
proof (*rule irreducibleI*)
fix *a b* **assume** $q = a * b$
hence *gauss-int-norm q* = *gauss-int-norm a* * *gauss-int-norm b*
by (*simp-all add*: *gauss-int-norm-mult*)
thus *is-unit a* ∨ *is-unit b*
using *assms* **by** (*auto dest*!: *prime-product simp*: *gauss-int-norm-eq-Suc-0-iff*)
qed (*use assms* **in** ‹*auto simp*: *is-unit-gauss-int-iff* ′›)
thus *prime-elem q*
using *irreducible-imp-prime-elem-gcd* **by** *blast*
qed

Also, a conjugate is a prime element iff the original element is a prime element:

lemma *prime-elem-gauss-cnj* [*intro*]: *prime-elem z* ⟹ *prime-elem* (*gauss-cnj z*)
by (*auto simp*: *prime-elem-def gauss-cnj-dvd-left-iff*)

lemma *prime-elem-gauss-cnj-iff* [*simp*]: *prime-elem* (*gauss-cnj z*) ⟷ *prime-elem z*
using *prime-elem-gauss-cnj*[*of z*] *prime-elem-gauss-cnj*[*of gauss-cnj z*] **by** *auto*

1.6.1 The factorisation of 2

2 factors as $-i(1+i)^2$ in the Gaussian integers, where $-i$ is a unit and $1+i$ is prime.

lemma *gauss-int-2-eq*: *2* = −$\mathrm{i}_{\mathbb{Z}}$ * (*1* + $\mathrm{i}_{\mathbb{Z}}$) ^ *2*
by (*simp add*: *gauss-int-eq-iff power2-eq-square*)

lemma *prime-elem-one-plus-i-gauss-int*: *prime-elem* (*1* + $\mathrm{i}_{\mathbb{Z}}$)
by (*rule prime-gauss-int-norm-imp-prime-elem*) (*auto simp*: *gauss-int-norm-def*)

lemma *prime-one-plus-i-gauss-int*: *prime* (*1* + $\mathrm{i}_{\mathbb{Z}}$)
by (*simp add*: *prime-def prime-elem-one-plus-i-gauss-int*
gauss-int-eq-iff normalize-gauss-int-def)

lemma *prime-factorization-2-gauss-int*:
prime-factorization (*2* :: *gauss-int*) = {#*1* + $\mathrm{i}_{\mathbb{Z}}$, *1* + $\mathrm{i}_{\mathbb{Z}}$#}
proof −
have *prime-factorization* (*2* :: *gauss-int*) =
(*prime-factorization* (*prod-mset* {#*1* + *gauss-i*, *1* + *gauss-i*#}))
by (*subst prime-factorization-unique*) (*auto simp*: *gauss-int-eq-iff normalize-gauss-int-def*)
also have *prime-factorization* (*prod-mset* {#*1* + *gauss-i*, *1* + *gauss-i*#}) =
{#*1* + *gauss-i*, *1* + *gauss-i*#}
using *prime-one-plus-i-gauss-int* **by** (*subst prime-factorization-prod-mset-primes*) *auto*
finally show *?thesis* **.**
qed

1.6.2 Inert primes

Any $\mathbb{Z}$-prime congruent 3 modulo 4 is also a Gaussian prime. These primes are called *inert*, because they do not decompose when moving from $\mathbb{Z}$ to $\mathbb{Z}[i]$.

lemma *gauss-int-norm-not-3-mod-4*: [*gauss-int-norm z* ≠ *3*] (*mod 4*)
proof −
have *A*: *ReZ z mod 4* ∈ {*0..3*} *ImZ z mod 4* ∈ {*0..3*} **by** *auto*
have *B*: {*0..3*} = {*0*, *1*, *2*, *3* :: *int*} **by** *auto*

have [*ReZ z* ^ *2* + *ImZ z* ^ *2* = (*ReZ z mod 4*) ^ *2* + (*ImZ z mod 4*) ^ *2*] (*mod 4*)
by (*intro cong-add cong-pow*) (*auto simp*: *cong-def*)
moreover have ((*ReZ z mod 4*) ^ *2* + (*ImZ z mod 4*) ^ *2*) *mod 4* ≠ *3 mod 4*
using *A* **unfolding** *B* **by** *auto*
ultimately have [*ReZ z* ^ *2* + *ImZ z* ^ *2* ≠ *3*] (*mod 4*)
unfolding *cong-def* **by** *metis*

```
  hence [int (nat (ReZ z ^ 2 + ImZ z ^ 2)) ≠ int 3] (mod (int 4))
    by simp
  thus ?thesis unfolding gauss-int-norm-def
    by (subst (asm) cong-int-iff)
qed

lemma prime-elem-gauss-int-of-nat:
  fixes n :: nat
  assumes prime: prime n and [n = 3] (mod 4)
  shows   prime-elem (of-nat n :: gauss-int)
proof (intro irreducible-imp-prime-elem irreducibleI)
  from assms show of-nat n ≠ (0 :: gauss-int)
    by (auto simp: gauss-int-eq-iff)
next
  show ¬is-unit (of-nat n :: gauss-int)
    using assms by (subst is-unit-gauss-int-iff) (auto simp: gauss-int-eq-iff)
next
  fix a b :: gauss-int
  assume *: of-nat n = a * b
  hence gauss-int-norm (a * b) = gauss-int-norm (of-nat n)
    by metis
  hence *: gauss-int-norm a * gauss-int-norm b = n ^ 2
    by (simp add: gauss-int-norm-mult power2-eq-square flip: nat-mult-distrib)
  from prime-power-mult-nat[OF prime this] obtain i j :: nat
    where ij: gauss-int-norm a = n ^ i gauss-int-norm b = n ^ j by blast

  have i + j = 2
  proof -
    have n ^ (i + j) = n ^ 2
      using ij * by (simp add: power-add)
    from prime-power-inj[OF prime this] show ?thesis by simp
  qed
  hence i = 0 ∧ j = 2 ∨ i = 1 ∧ j = 1 ∨ i = 2 ∧ j = 0
    by auto
  thus is-unit a ∨ is-unit b
  proof (elim disjE)
    assume i = 1 ∧ j = 1
    with ij have gauss-int-norm a = n
      by auto
    hence [gauss-int-norm a = n] (mod 4)
      by simp
    also have [n = 3] (mod 4) by fact
    finally have [gauss-int-norm a = 3] (mod 4) .
    moreover have [gauss-int-norm a ≠ 3] (mod 4)
      by (rule gauss-int-norm-not-3-mod-4)
    ultimately show ?thesis by contradiction
  qed (use ij in ‹auto simp: is-unit-gauss-int-iff'›)
qed
```

theorem *prime-gauss-int-of-nat*:
fixes $n :: nat$
assumes *prime*: *prime* n **and** $[n = 3]$ $(mod\ 4)$
shows *prime* $(of\text{-}nat\ n :: gauss\text{-}int)$
using *prime-elem-gauss-int-of-nat*[*OF assms*]
unfolding *prime-def* **by** *simp*

1.6.3 Non-inert primes

Any $\mathbb{Z}$-prime congruent 1 modulo 4 factors into two conjugate Gaussian primes.

lemma *minimal-QuadRes-neg1*:
assumes *QuadRes* $n\ (-1)$ $n > 1$ *odd* n
obtains $x :: nat$ **where** $x \leq (n - 1)\ div\ 2$ **and** $[x \hat{}\ 2 + 1 = 0]$ $(mod\ n)$
proof −
from ‹*QuadRes* $n\ (-1)$› **obtain** x **where** $[x \hat{}\ 2 = (-1)]$ $(mod\ (int\ n))$
by (*auto simp*: *QuadRes-def*)
hence $[x \hat{}\ 2 + 1 = -1 + 1]$ $(mod\ (int\ n))$
by (*intro cong-add*) *auto*
also have $x \hat{}\ 2 + 1 = int\ (nat\ |x|\ \hat{}\ 2 + 1)$
by *simp*
finally have $[int\ (nat\ |x|\ \hat{}\ 2 + 1) = int\ 0]$ $(mod\ (int\ n))$
by *simp*
hence $[nat\ |x|\ \hat{}\ 2 + 1 = 0]$ $(mod\ n)$
by (*subst* (*asm*) *cong-int-iff*)

define x' **where**
$x' = (if\ nat\ |x|\ mod\ n \leq (n - 1)\ div\ 2\ then\ nat\ |x|\ mod\ n\ else\ n - (nat\ |x|\ mod\ n))$
have x'-*quadres*: $[x' \hat{}\ 2 + 1 = 0]$ $(mod\ n)$
proof (*cases* $nat\ |x|\ mod\ n \leq (n - 1)\ div\ 2$)
case *True*
hence $[x' \hat{}\ 2 + 1 = (nat\ |x|\ mod\ n)\ \hat{}\ 2 + 1]$ $(mod\ n)$
by (*simp add*: x'-*def*)
also have $[(nat\ |x|\ mod\ n)\ \hat{}\ 2 + 1 = nat\ |x|\ \hat{}\ 2 + 1]$ $(mod\ n)$
by (*intro cong-add cong-pow*) (*auto simp*: *cong-def*)
also have $[nat\ |x|\ \hat{}\ 2 + 1 = 0]$ $(mod\ n)$ **by** *fact*
finally show *?thesis* **.**
next
case *False*
hence $[int\ (x' \hat{}\ 2 + 1) = (int\ n - int\ (nat\ |x|\ mod\ n))\ \hat{}\ 2 + 1]$ $(mod\ int\ n)$
using ‹$n > 1$› **by** (*simp add*: x'-*def of-nat-diff add-ac*)
also have $[(int\ n - int\ (nat\ |x|\ mod\ n))\ \hat{}\ 2 + 1 =$
$(0 - int\ (nat\ |x|\ mod\ n))\ \hat{}\ 2 + 1]$ $(mod\ int\ n)$
by (*intro cong-add cong-pow*) (*auto simp*: *cong-def*)
also have $[(0 - int\ (nat\ |x|\ mod\ n))\ \hat{}\ 2 + 1 = int\ ((nat\ |x|\ mod\ n)\ \hat{}\ 2 + 1)]$ $(mod\ (int\ n))$
by (*simp add*: *add-ac*)
finally have $[x' \hat{}\ 2 + 1 = (nat\ |x|\ mod\ n)^2 + 1]$ $(mod\ n)$

```
    by (subst (asm) cong-int-iff)
  also have [(nat |x| mod n)^2 + 1 = nat |x| ^ 2 + 1] (mod n)
    by (intro cong-add cong-pow) (auto simp: cong-def)
  also have [nat |x| ^ 2 + 1 = 0] (mod n) by fact
  finally show ?thesis .
 qed
 moreover have x'-le: x' ≤ (n - 1) div 2
  using ‹odd n› by (auto elim!: oddE simp: x'-def)
 ultimately show ?thesis by (intro that[of x'])
qed
```

Let p be some prime number that is congruent 1 modulo 4.

```
locale noninert-gauss-int-prime =
 fixes p :: nat
 assumes prime-p: prime p and cong-1-p: [p = 1] (mod 4)
begin

lemma p-gt-2: p > 2 and odd-p: odd p
proof -
 from prime-p and cong-1-p have p > 1 p ≠ 2
  by (auto simp: prime-gt-Suc-0-nat cong-def)
 thus p > 2 by auto
 with prime-p show odd p
  using primes-dvd-imp-eq two-is-prime-nat by blast
qed
```

-1 is a quadratic residue modulo p, so there exists some x such that $x^2 + 1$ is divisible by p. Moreover, we can choose x such that it is positive and no greater than $\frac{1}{2}(p-1)$:

```
lemma minimal-QuadRes-neg1:
 obtains x where x > 0 x ≤ (p - 1) div 2 [x ^ 2 + 1 = 0] (mod p)
proof -
 have [Legendre (-1) (int p) = (-1) ^ ((p - 1) div 2)] (mod (int p))
  using prime-p p-gt-2 by (intro euler-criterion) auto
 also have [p - 1 = 1 - 1] (mod 4)
  using p-gt-2 by (intro cong-diff-nat cong-refl) (use cong-1-p in auto)
 hence 2 * 2 dvd p - 1
  by (simp add: cong-0-iff)
 hence even ((p - 1) div 2)
  using dvd-mult-imp-div by blast
 hence (-1) ^ ((p - 1) div 2) = (1 :: int)
  by simp
 finally have Legendre (-1) (int p) mod p = 1
  using p-gt-2 by (auto simp: cong-def)
 hence Legendre (-1) (int p) = 1
  using p-gt-2 by (auto simp: Legendre-def cong-def zmod-minus1 split: if-splits)
 hence QuadRes p (-1)
  by (simp add: Legendre-def split: if-splits)
 from minimal-QuadRes-neg1[OF this] p-gt-2 odd-p
```

```
  obtain x where x: x ≤ (p − 1) div 2 [x ^ 2 + 1 = 0] (mod p) by auto
 have x > 0
  using x p-gt-2 by (auto intro!: Nat.gr0I simp: cong-def)
 from x and this show ?thesis by (intro that[of x]) auto
qed
```

We can show from this that p is not prime as a Gaussian integer.

```
lemma not-prime: ¬prime-elem (of-nat p :: gauss-int)
proof
 assume prime: prime-elem (of-nat p :: gauss-int)
 obtain x where x: x > 0 x ≤ (p − 1) div 2 [x ^ 2 + 1 = 0] (mod p)
  using  minimal-QuadRes-neg1 .

 have of-nat p dvd (of-nat (x ^ 2 + 1) :: gauss-int)
  using x by (intro of-nat-dvd-of-nat) (auto simp: cong-0-iff)
 also have eq: of-nat (x ^ 2 + 1) = ((of-nat x + iℤ) * (of-nat x − iℤ) :: gauss-int)
   using ‹x > 0› by (simp add: algebra-simps gauss-int-eq-iff power2-eq-square
of-nat-diff)
 finally have of-nat p dvd ((of-nat x + iℤ) * (of-nat x − iℤ) :: gauss-int) .

 from prime and this
  have of-nat p dvd (of-nat x + iℤ :: gauss-int) ∨ of-nat p dvd (of-nat x − iℤ ::
gauss-int)
  by (rule prime-elem-dvd-multD)
 hence dvd: of-nat p dvd (of-nat x + iℤ :: gauss-int) of-nat p dvd (of-nat x − iℤ
:: gauss-int)
  by (auto dest: of-nat-dvd-imp-dvd-gauss-cnj)

 have of-nat (p ^ 2) = (of-nat p * of-nat p :: gauss-int)
  by (simp add: power2-eq-square)
 also from dvd have ... dvd ((of-nat x + iℤ) * (of-nat x − iℤ))
  by (intro mult-dvd-mono)
 also have ... = of-nat (x ^ 2 + 1)
  by (rule eq [symmetric])
 finally have p ^ 2 dvd (x ^ 2 + 1)
  by (subst (asm) of-nat-dvd-of-nat-gauss-int-iff)
 hence p ^ 2 ≤ x ^ 2 + 1
  by (intro dvd-imp-le) auto
 moreover have p ^ 2 > x ^ 2 + 1
 proof −
  have x ^ 2 + 1 ≤ ((p − 1) div 2) ^ 2 + 1
   using x by (intro add-mono power-mono) auto
  also have ... ≤ (p − 1) ^ 2 + 1
   by auto
  also have (p − 1) * (p − 1) < (p − 1) * (p + 1)
   using p-gt-2 by (intro mult-strict-left-mono) auto
  hence (p − 1) ^ 2 + 1 < p ^ 2
   by (simp add: algebra-simps power2-eq-square)
  finally show ?thesis .
```

```
  qed
  ultimately show False by linarith
qed
```

Any prime factor of p in the Gaussian integers must have norm p.

```
lemma norm-prime-divisor:
  fixes q :: gauss-int
  assumes q: prime-elem q q dvd of-nat p
  shows gauss-int-norm q = p
proof −
  from assms obtain r where r: of-nat p = q * r
    by auto
  have p ^ 2 = gauss-int-norm (of-nat p)
    by simp
  also have ... = gauss-int-norm q * gauss-int-norm r
    by (auto simp: r gauss-int-norm-mult)
  finally have *: gauss-int-norm q * gauss-int-norm r = p ^ 2
    by simp
  hence ∃ i j. gauss-int-norm q = p ^ i ∧ gauss-int-norm r = p ^ j
    using prime-p by (intro prime-power-mult-nat)
  then obtain i j where ij: gauss-int-norm q = p ^ i gauss-int-norm r = p ^ j
    by blast
  have ij-eq-2: i + j = 2
  proof −
    from * have p ^ (i + j) = p ^ 2
      by (simp add: power-add ij)
    thus ?thesis
      using p-gt-2 by (subst (asm) power-inject-exp) auto
  qed
  hence i = 0 ∧ j = 2 ∨ i = 1 ∧ j = 1 ∨ i = 2 ∧ j = 0 by auto
  hence i = 1
  proof (elim disjE)
    assume i = 2 ∧ j = 0
    hence is-unit r
      using ij by (simp add: gauss-int-norm-eq-Suc-0-iff)
    hence prime-elem (of-nat p :: gauss-int) using ‹prime-elem q›
      by (simp add: prime-elem-mult-unit-left r mult.commute[of - r])
    with not-prime show i = 1 by contradiction
  qed (use q ij in ‹auto simp: gauss-int-norm-eq-Suc-0-iff›)
  thus ?thesis using ij by simp
qed
```

We now show two lemmas that characterise the two prime factors of p in the Gaussian integers: they are two conjugates $x \pm iy$ for positive integers x and y such that $x^2 + y^2 = p$.

```
lemma prime-divisor-exists:
  obtains q where prime q prime-elem (gauss-cnj q) ReZ q > 0 ImZ q > 0
                  of-nat p = q * gauss-cnj q gauss-int-norm q = p
proof −
```

```
  have ∃ q::gauss-int. q dvd of-nat p ∧ prime q
   by (rule prime-divisor-exists) (use prime-p in ‹auto simp: is-unit-gauss-int-iff'›)
  then obtain q :: gauss-int where q: prime q q dvd of-nat p
    by blast
  from ‹prime q› have [simp]: q ≠ 0 by auto
  have normalize q = q
    using q by simp
  hence q-signs: ReZ q > 0 ImZ q ≥ 0
    by (subst (asm) normalized-gauss-int-iff; simp)+

  from q have gauss-int-norm q = p
    using norm-prime-divisor[of q] by simp
  moreover from this have gauss-int-norm (gauss-cnj q) = p
    by simp
  hence prime-elem (gauss-cnj q)
    using prime-p by (intro prime-gauss-int-norm-imp-prime-elem) auto
  moreover have of-nat p = q * gauss-cnj q
    using ‹gauss-int-norm q = p› by (simp add: self-mult-gauss-cnj)
  moreover have ImZ q ≠ 0
  proof
    assume [simp]: ImZ q = 0
    define m where m = nat (ReZ q)
    have [simp]: q = of-nat m
      using q-signs by (auto simp: gauss-int-eq-iff m-def)
    with q have m dvd p
      by (simp add: of-nat-dvd-of-nat-gauss-int-iff)
    with prime-p have m = 1 ∨ m = p
      using prime-nat-iff by blast
    with q show False using not-prime by auto
  qed
  with q-signs have ImZ q > 0 by simp
  ultimately show ?thesis using q q-signs by (intro that[of q])
qed

theorem prime-factorization:
  obtains q1 q2
  where prime q1 prime q2 prime-factorization (of-nat p) = {#q1, q2#}
        gauss-int-norm q1 = p gauss-int-norm q2 = p q2 = iℤ * gauss-cnj q1
        ReZ q1 > 0 ImZ q1 > 0 ReZ q1 > 0 ImZ q2 > 0
proof −
  obtain q where q: prime q prime-elem (gauss-cnj q) ReZ q > 0 ImZ q > 0
                    of-nat p = q * gauss-cnj q gauss-int-norm q = p
    using prime-divisor-exists by metis
  from ‹prime q› have [simp]: q ≠ 0 by auto
  define q' where q' = normalize (gauss-cnj q)
  have prime-factorization (of-nat p) = prime-factorization (prod-mset {#q, q'#})
    by (subst prime-factorization-unique) (auto simp: q q'-def)
  also have ... = {#q, q'#}
    using q by (subst prime-factorization-prod-mset-primes) (auto simp: q'-def)
```

finally have *prime-factorization* (*of-nat p*) = {#*q*, *q′*#} .
moreover have *q′* = i$_{\mathbb{Z}}$ ∗ *gauss-cnj q*
using *q* **by** (*auto simp*: *normalize-gauss-int-def q′-def*)
moreover have *prime q′*
using *q* **by** (*auto simp*: *q′-def*)
ultimately show *?thesis* **using** *q*
by (*intro that*[*of q q′*]) (*auto simp*: *q′-def gauss-int-norm-mult*)
qed

end

In particular, a consequence of this is that any prime congruent 1 modulo 4 can be written as a sum of squares of positive integers.

lemma *prime-cong-1-mod-4-gauss-int-norm-exists*:
fixes *p* :: *nat*
assumes *prime p* [*p* = *1*] (*mod 4*)
shows ∃ *z*. *gauss-int-norm z* = *p* ∧ *ReZ z* > *0* ∧ *ImZ z* > *0*
proof −
from *assms* **interpret** *noninert-gauss-int-prime p*
by *unfold-locales*
from *prime-divisor-exists* **obtain** *q*
where *q*: *prime q of-nat p* = *q* ∗ *gauss-cnj q*
ReZ q > *0 ImZ q* > *0 gauss-int-norm q* = *p* **by** *metis*
have *p* = *gauss-int-norm q*
using *q* **by** *simp*
thus *?thesis* **using** *q* **by** *blast*
qed

1.6.4 Full classification of Gaussian primes

Any prime in the ring of Gaussian integers is of the form

- *1* + i$_{\mathbb{Z}}$
- p where $p \in \mathbb{N}$ is prime in $\mathbb{N}$ and congruent 1 modulo 4
- $x + iy$ where x, y are positive integers and $x^2 + y^2$ is a prime congruent 3 modulo 4

or an associated element of one of these.

theorem *gauss-int-prime-classification*:
fixes *x* :: *gauss-int*
assumes *prime x*
obtains
(*one-plus-i*) *x* = *1* + i$_{\mathbb{Z}}$
| (*cong-3-mod-4*) *p* **where** *x* = *of-nat p prime p* [*p* = *3*] (*mod 4*)
| (*cong-1-mod-4*) *prime* (*gauss-int-norm x*) [*gauss-int-norm x* = *1*] (*mod 4*)
ReZ x > *0 ImZ x* > *0 ReZ x* ≠ *ImZ x*

```
proof −
  define N where N = gauss-int-norm x
  have x dvd x ∗ gauss-cnj x
    by simp
  also have ... = of-nat (gauss-int-norm x)
    by (simp add: self-mult-gauss-cnj)
  finally have x ∈ prime-factors (of-nat N)
    using assms by (auto simp: in-prime-factors-iff N-def)
  also have N = prod-mset (prime-factorization N)
     using assms unfolding N-def by (subst prod-mset-prime-factorization-nat)
auto
  also have (of-nat ... :: gauss-int) =
              prod-mset (image-mset of-nat (prime-factorization N))
    by (subst of-nat-prod-mset) auto
  also have prime-factors ... = (⋃ p∈prime-factors N. prime-factors (of-nat p))
    by (subst prime-factorization-prod-mset) auto
  finally obtain p where p: p ∈ prime-factors N x ∈ prime-factors (of-nat p)
    by auto

  have prime p
    using p by auto
  hence ¬(2 ∗ 2) dvd p
    using product-dvd-irreducibleD[of p 2 2]
    by (auto simp flip: prime-elem-iff-irreducible)
  hence [p ≠ 0] (mod 4)
    using p by (auto simp: cong-0-iff in-prime-factors-iff)
  hence p mod 4 ∈ {1,2,3} by (auto simp: cong-def)
  thus ?thesis
  proof (elim singletonE insertE)
    assume p mod 4 = 2
    hence p mod 4 mod 2 = 0
      by simp
    hence p mod 2 = 0
      by (simp add: mod-mod-cancel)
    with ‹prime p› have [simp]: p = 2
      using prime-prime-factor two-is-prime-nat by blast
    have prime-factors (of-nat p) = {1 + iℤ :: gauss-int}
      by (simp add: prime-factorization-2-gauss-int)
    with p show ?thesis using that(1) by auto
  next
    assume ∗: p mod 4 = 3
    hence prime-factors (of-nat p) = {of-nat p :: gauss-int}
      using prime-gauss-int-of-nat[of p] ‹prime p›
      by (subst prime-factorization-prime) (auto simp: cong-def)
    with p show ?thesis using that(2)[of p] ∗
      by (auto simp: cong-def)
  next
    assume ∗: p mod 4 = 1
    then interpret noninert-gauss-int-prime p
```

```
    by unfold-locales (use ‹prime p› in ‹auto simp: cong-def›)
  obtain q1 q2 :: gauss-int where q12:
    prime q1 prime q2 prime-factorization (of-nat p) = {#q1, q2#}
    gauss-int-norm q1 = p gauss-int-norm q2 = p q2 = iℤ * gauss-cnj q1
    ReZ q1 > 0 ImZ q1 > 0 ReZ q1 > 0 ImZ q2 > 0
    using prime-factorization by metis
  from p q12 have x = q1 ∨ x = q2 by auto
  with q12 have **: gauss-int-norm x = p ReZ x > 0 ImZ x > 0
    by auto
  have ReZ x ≠ ImZ x
  proof
    assume ReZ x = ImZ x
    hence even (gauss-int-norm x)
      by (auto simp: gauss-int-norm-def nat-mult-distrib)
    hence even p using ‹gauss-int-norm x = p›
      by simp
    with ‹p mod 4 = 1› show False
      by presburger
  qed
  thus ?thesis using that(3) ‹prime p› * **
    by (simp add: cong-def)
 qed
qed

lemma prime-gauss-int-norm-squareD:
 fixes z :: gauss-int
 assumes prime z gauss-int-norm z = p ^ 2
 shows   prime p ∧ z = of-nat p
 using assms(1)
proof (cases rule: gauss-int-prime-classification)
 case one-plus-i
 have prime (2 :: nat) by simp
 also from one-plus-i have 2 = p ^ 2
   using assms(2) by (auto simp: gauss-int-norm-def)
 finally show ?thesis by (simp add: prime-power-iff)
next
 case (cong-3-mod-4 p)
 thus ?thesis using assms by auto
next
 case cong-1-mod-4
 with assms show ?thesis
   by (auto simp: prime-power-iff)
qed

lemma gauss-int-norm-eq-prime-squareD:
 assumes prime p and [p = 3] (mod 4) and gauss-int-norm z = p ^ 2
 shows   normalize z = of-nat p and prime-elem z
proof −
 have ∃ q::gauss-int. q dvd z ∧ prime q
```

```
  by (rule prime-divisor-exists) (use assms in ‹auto simp: is-unit-gauss-int-iff'›)
 then obtain q :: gauss-int where q: q dvd z prime q by blast
 have gauss-int-norm q dvd gauss-int-norm z
  by (rule gauss-int-norm-dvd-mono) fact
 also have ... = p ^ 2 by fact
 finally obtain i where i: i ≤ 2 gauss-int-norm q = p ^ i
  by (subst (asm) divides-primepow-nat) (use assms q in auto)
 from i assms q have i ≠ 0
  by (auto intro!: Nat.gr0I simp: gauss-int-norm-eq-Suc-0-iff)
 moreover from i assms q have i ≠ 1
  using gauss-int-norm-not-3-mod-4[of q] by auto
 ultimately have i = 2 using i by auto
 with i have gauss-int-norm q = p ^ 2 by auto
 hence [simp]: q = of-nat p
  using prime-gauss-int-norm-squareD[of q p] q by auto
 have normalize (of-nat p) = normalize z
  using q assms
  by (intro gauss-int-dvd-same-norm-imp-associated) auto
 thus ∗: normalize z = of-nat p by simp

 have prime (normalize z)
  using prime-gauss-int-of-nat[of p] assms by (subst ∗) auto
 thus prime-elem z by simp
qed
```

The following can be used as a primality test for Gaussian integers. It effectively reduces checking the primality of a Gaussian integer to checking the primality of an integer.

A Gaussian integer is prime if either its norm is either $\mathbb{Z}$-prime or the square of a $\mathbb{Z}$-prime that is congruent 3 modulo 4.

```
lemma prime-elem-gauss-int-iff:
 fixes z :: gauss-int
 defines n ≡ gauss-int-norm z
 shows   prime-elem z ⟷ prime n ∨ (∃p. n = p ^ 2 ∧ prime p ∧ [p = 3] (mod
4))
proof
 assume prime n ∨ (∃p. n = p ^ 2 ∧ prime p ∧ [p = 3] (mod 4))
 thus prime-elem z
  by (auto intro: gauss-int-norm-eq-prime-squareD(2)
                  prime-gauss-int-norm-imp-prime-elem simp: n-def)
next
 assume prime-elem z
 hence prime (normalize z) by simp
 thus prime n ∨ (∃p. n = p ^ 2 ∧ prime p ∧ [p = 3] (mod 4))
 proof (cases rule: gauss-int-prime-classification)
  case one-plus-i
  have n = gauss-int-norm (normalize z)
   by (simp add: n-def)
  also have normalize z = 1 + iℤ
```

by *fact*
also have *gauss-int-norm ... = 2*
by (*simp add*: *gauss-int-norm-def*)
finally show *?thesis* **by** *simp*
next
case (*cong-3-mod-4 p*)
have *n = gauss-int-norm* (*normalize z*)
by (*simp add*: *n-def*)
also have *normalize z = of-nat p*
by *fact*
also have *gauss-int-norm ... = p ^ 2*
by *simp*
finally show *?thesis* **using** *cong-3-mod-4* **by** *simp*
next
case *cong-1-mod-4*
thus *?thesis* **by** (*simp add*: *n-def*)
qed
qed

1.6.5 Multiplicities of primes

In this section, we will show some results connecting the multiplicity of a Gaussian prime p in a Gaussian integer z to the $\mathbb{Z}$-multiplicity of the norm of p in the norm of z.

The multiplicity of the Gaussian prime $1 + \mathrm{i}_{\mathbb{Z}}$ in an integer c is simply twice the $\mathbb{Z}$-multiplicity of 2 in c:

lemma *multiplicity-prime-1-plus-i-aux*: *multiplicity* (*1* + $\mathrm{i}_{\mathbb{Z}}$) (*of-nat c*) = *2* ∗ *multiplicity 2 c*
proof (*cases c = 0*)
case [*simp*]: *False*
have *2* ∗ *multiplicity 2 c = multiplicity 2* (*c ^ 2*)
by (*simp add*: *prime-elem-multiplicity-power-distrib*)
also have *multiplicity 2* (*c ^ 2*) = *multiplicity* (*of-nat 2*) (*of-nat c ^ 2* :: *gauss-int*)
by (*simp flip*: *multiplicity-gauss-int-of-nat*)
also have *of-nat 2* = (−$\mathrm{i}_{\mathbb{Z}}$) ∗ (*1* + $\mathrm{i}_{\mathbb{Z}}$) ^ *2*
by (*simp add*: *algebra-simps power2-eq-square*)
also have *multiplicity ...* (*of-nat c ^ 2*) = *multiplicity* ((*1* + $\mathrm{i}_{\mathbb{Z}}$) ^ *2*) (*of-nat c ^ 2*)
by (*subst multiplicity-times-unit-left*) *auto*
also have ... = *multiplicity* (*1* + $\mathrm{i}_{\mathbb{Z}}$) (*of-nat c*)
by (*subst multiplicity-power-power*) *auto*
finally show *?thesis* **..**
qed *auto*

Tha multiplicity of an inert Gaussian prime $q \in \mathbb{Z}$ in a Gaussian integer z is precisely half the $\mathbb{Z}$-multiplicity of q in the norm of z.

lemma *multiplicity-prime-cong-3-mod-4*:

assumes *prime* (*of-nat q* :: *gauss-int*)
shows *multiplicity q* (*gauss-int-norm z*) = *2* ∗ *multiplicity* (*of-nat q*) *z*
proof (*cases z* = *0*)
case [*simp*]: *False*
have *multiplicity q* (*gauss-int-norm z*) =
multiplicity (*of-nat q*) (*of-nat* (*gauss-int-norm z*) :: *gauss-int*)
by (*simp add*: *multiplicity-gauss-int-of-nat*)
also have ... = *multiplicity* (*of-nat q*) (*z* ∗ *gauss-cnj z*)
by (*simp add*: *self-mult-gauss-cnj*)
also have ... = *multiplicity* (*of-nat q*) *z* + *multiplicity* (*gauss-cnj* (*of-nat q*)) (*gauss-cnj z*)
using *assms* **by** (*subst prime-elem-multiplicity-mult-distrib*) *auto*
also have *multiplicity* (*gauss-cnj* (*of-nat q*)) (*gauss-cnj z*) = *multiplicity* (*of-nat q*) *z*
by (*subst multiplicity-gauss-cnj*) *auto*
also have ... + ... = *2* ∗ ...
by *simp*
finally show *?thesis* **.**
qed *auto*

For Gaussian primes p whose norm is congruent 1 modulo 4, the $\mathbb{Z}[i]$-multiplicity of p in an integer c is just the $\mathbb{Z}$-multiplicity of their norm in c.

lemma *multiplicity-prime-cong-1-mod-4-aux*:
fixes *p* :: *gauss-int*
assumes *prime-elem p ReZ p* > *0 ImZ p* > *0 ImZ p* ≠ *ReZ p*
shows *multiplicity p* (*of-nat c*) = *multiplicity* (*gauss-int-norm p*) *c*
proof (*cases c* = *0*)
case [*simp*]: *False*
show *?thesis*
proof (*intro antisym multiplicity-geI*)
define *k* **where** *k* = *multiplicity p* (*of-nat c*)
have *p* ^ *k dvd of-nat c*
by (*simp add*: *multiplicity-dvd k-def*)
moreover have *gauss-cnj p* ^ *k dvd of-nat c*
using *multiplicity-dvd*[*of gauss-cnj p of-nat c*]
multiplicity-gauss-cnj[*of p of-nat c*] **by** (*simp add*: *k-def*)
moreover have ¬*p dvd gauss-cnj p*
using *assms* **by** (*subst self-dvd-gauss-cnj-iff*) *auto*
hence ¬*p dvd gauss-cnj p* ^ *k*
using *assms prime-elem-dvd-power* **by** *blast*
ultimately have *p* ^ *k* ∗ *gauss-cnj p* ^ *k dvd of-nat c*
using *assms* **by** (*intro prime-elem-power-mult-dvdI*) *auto*
also have *p* ^ *k* ∗ *gauss-cnj p* ^ *k* = *of-nat* (*gauss-int-norm p* ^ *k*)
by (*simp flip*: *self-mult-gauss-cnj add*: *power-mult-distrib*)
finally show *gauss-int-norm p* ^ *k dvd c*
by (*subst* (*asm*) *of-nat-dvd-of-nat-gauss-int-iff*)
next
define *k* **where** *k* = *multiplicity* (*gauss-int-norm p*) *c*

```
    have p ^ k dvd (p * gauss-cnj p) ^ k
      by (intro dvd-power-same) auto
    also have ... = of-nat (gauss-int-norm p ^ k)
      by (simp add: self-mult-gauss-cnj)
    also have ... dvd of-nat c
     unfolding of-nat-dvd-of-nat-gauss-int-iff by (auto simp: k-def multiplicity-dvd)
    finally show p ^ k dvd of-nat c .
  qed (use assms in ‹auto simp: gauss-int-norm-eq-Suc-0-iff›)
qed auto
```

The multiplicity of a Gaussian prime with norm congruent 1 modulo 4 in some Gaussian integer z and the multiplicity of its conjugate in z sum to the the $\mathbb{Z}$-multiplicity of their norm in the norm of z:

```
lemma multiplicity-prime-cong-1-mod-4:
  fixes p :: gauss-int
  assumes prime-elem p ReZ p > 0 ImZ p > 0 ImZ p ≠ ReZ p
  shows multiplicity (gauss-int-norm p) (gauss-int-norm z) =
          multiplicity p z + multiplicity (gauss-cnj p) z
proof (cases z = 0)
  case [simp]: False
  have multiplicity (gauss-int-norm p) (gauss-int-norm z) =
         multiplicity p (of-nat (gauss-int-norm z))
    using assms by (subst multiplicity-prime-cong-1-mod-4-aux) auto
  also have ... = multiplicity p (z * gauss-cnj z)
    by (simp add: self-mult-gauss-cnj)
  also have ... = multiplicity p z + multiplicity p (gauss-cnj z)
    using assms by (subst prime-elem-multiplicity-mult-distrib) auto
  also have multiplicity p (gauss-cnj z) = multiplicity (gauss-cnj p) z
    by (subst multiplicity-gauss-cnj [symmetric]) auto
  finally show ?thesis .
qed auto
```

The multiplicity of the Gaussian prime $1 + \mathrm{i}_{\mathbb{Z}}$ in a Gaussian integer z is precisely the $\mathbb{Z}$-multiplicity of 2 in the norm of z:

```
lemma multiplicity-prime-1-plus-i: multiplicity (1 + iℤ) z = multiplicity 2 (gauss-int-norm
z)
proof (cases z = 0)
  case [simp]: False
  note [simp] = prime-elem-one-plus-i-gauss-int
   have 2 * multiplicity 2 (gauss-int-norm z) = multiplicity (1 + iℤ) (of-nat
(gauss-int-norm z))
    by (rule multiplicity-prime-1-plus-i-aux [symmetric])
  also have ... = multiplicity (1 + iℤ) (z * gauss-cnj z)
    by (simp add: self-mult-gauss-cnj)
   also have ... = multiplicity (1 + iℤ) z + multiplicity (gauss-cnj (1 − iℤ))
(gauss-cnj z)
    by (subst prime-elem-multiplicity-mult-distrib) auto
  also have multiplicity (gauss-cnj (1 − iℤ)) (gauss-cnj z) = multiplicity (1 − iℤ)
z
```

by (*subst multiplicity-gauss-cnj*) *auto*
also have $1 - \mathrm{i}_{\mathbb{Z}} = (-\mathrm{i}_{\mathbb{Z}}) * (1 + \mathrm{i}_{\mathbb{Z}})$
by (*simp add*: *algebra-simps*)
also have *multiplicity* ... z = *multiplicity* $(1 + \mathrm{i}_{\mathbb{Z}})$ z
by (*subst multiplicity-times-unit-left*) *auto*
also have ... + ... = 2 * ...
by *simp*
finally show *?thesis* **by** *simp*
qed *auto*

1.7 Coprimality of an element and its conjugate

Using the classification of the primes, we now show that if the real and imaginary parts of a Gaussian integer are coprime and its norm is odd, then it is coprime to its own conjugate.

lemma *coprime-self-gauss-cnj*:
assumes *coprime* (*ReZ z*) (*ImZ z*) **and** *odd* (*gauss-int-norm z*)
shows *coprime z* (*gauss-cnj z*)
proof (*rule coprimeI*)
fix *d* **assume** *d dvd z d dvd gauss-cnj z*
have *: *False* **if** *p* ∈ *prime-factors z p* ∈ *prime-factors* (*gauss-cnj z*) **for** *p*
proof −
from *that* **have** *p*: *prime p p dvd z p dvd gauss-cnj z*
by *auto*

define *p′* **where** *p′* = *gauss-cnj p*
define *d* **where** *d* = *gauss-int-norm p*
have *of-nat-d-eq*: *of-nat d* = *p* * *p′*
by (*simp add*: *p′-def self-mult-gauss-cnj d-def*)
have *prime-elem p prime-elem p′ p dvd z p′ dvd z p dvd gauss-cnj z p′ dvd gauss-cnj z*
using *that* **by** (*auto simp*: *in-prime-factors-iff p′-def gauss-cnj-dvd-left-iff*)

have *prime p*
using *that* **by** *auto*
then obtain *q* **where** *q*: *prime q of-nat q dvd z*
proof (*cases rule*: *gauss-int-prime-classification*)
case *one-plus-i*
hence 2 = *gauss-int-norm p*
by (*auto simp*: *gauss-int-norm-def*)
also have *gauss-int-norm p dvd gauss-int-norm z*
using *p* **by** (*intro gauss-int-norm-dvd-mono*) *auto*
finally have *even* (*gauss-int-norm z*) **.**
with ‹*odd* (*gauss-int-norm z*)› **show** *?thesis*
by *contradiction*
next
case (*cong-3-mod-4 q*)
thus *?thesis* **using** *that*[*of q*] *p* **by** *simp*
next

```
    case cong-1-mod-4
    hence ¬p dvd p'
      unfolding p'-def by (subst self-dvd-gauss-cnj-iff) auto
    hence p * p' dvd z using p
      by (intro prime-elem-mult-dvdI) (auto simp: p'-def gauss-cnj-dvd-left-iff)
    also have p * p' = of-nat (gauss-int-norm p)
      by (simp add: p'-def self-mult-gauss-cnj)
    finally show ?thesis using that[of gauss-int-norm p] cong-1-mod-4
      by simp
  qed

  have of-nat q dvd gcd (2 * of-int (ReZ z)) (2 * iℤ * of-int (ImZ z))
  proof (rule gcd-greatest)
    have of-nat q dvd (z + gauss-cnj z)
      using q by (auto simp: gauss-cnj-dvd-right-iff)
    also have ... = 2 * of-int (ReZ z)
      by (simp add: self-plus-gauss-cnj)
    finally show of-nat q dvd (2 * of-int (ReZ z) :: gauss-int) .
  next
    have of-nat q dvd (z - gauss-cnj z)
      using q by (auto simp: gauss-cnj-dvd-right-iff)
    also have ... = 2 * iℤ * of-int (ImZ z)
      by (simp add: self-minus-gauss-cnj)
    finally show of-nat q dvd (2 * iℤ * of-int (ImZ z)) .
  qed
  also have ... = 2
  proof -
    have odd (ReZ z) ∨ odd (ImZ z)
      using assms by (auto simp: gauss-int-norm-def even-nat-iff)
    thus ?thesis
    proof
      assume odd (ReZ z)
      hence coprime (of-int (ReZ z)) (of-int 2 :: gauss-int)
        unfolding coprime-of-int-gauss-int coprime-right-2-iff-odd .
      thus ?thesis
        using assms
        by (subst gcd-mult-left-right-cancel)
       (auto simp: coprime-of-int-gauss-int coprime-commute is-unit-left-imp-coprime
                   is-unit-right-imp-coprime gcd-proj1-if-dvd gcd-proj2-if-dvd)
    next
      assume odd (ImZ z)
      hence coprime (of-int (ImZ z)) (of-int 2 :: gauss-int)
        unfolding coprime-of-int-gauss-int coprime-right-2-iff-odd .
      hence gcd (2 * of-int (ReZ z)) (2 * iℤ * of-int (ImZ z)) = gcd (2 * of-int
(ReZ z)) (2 * iℤ)
        using assms
        by (subst gcd-mult-right-right-cancel)
       (auto simp: coprime-of-int-gauss-int coprime-commute is-unit-left-imp-coprime
                   is-unit-right-imp-coprime)
```

```
      also have ... = normalize (2 * gcd (of-int (ReZ z)) iℤ)
        by (subst gcd-mult-left) auto
      also have gcd (of-int (ReZ z)) iℤ = 1
        by (subst coprime-iff-gcd-eq-1 [symmetric], rule is-unit-right-imp-coprime)
auto
      finally show ?thesis by simp
    qed
  qed
  finally have of-nat q dvd (of-nat 2 :: gauss-int)
    by simp
  hence q dvd 2
    by (simp only: of-nat-dvd-of-nat-gauss-int-iff)
  with ‹prime q› have q = 2
    using primes-dvd-imp-eq two-is-prime-nat by blast
  with q have 2 dvd z
    by auto

  have 2 dvd gauss-int-norm 2
    by simp
  also have ... dvd gauss-int-norm z
    using ‹2 dvd z› by (intro gauss-int-norm-dvd-mono)
  finally show False using ‹odd (gauss-int-norm z)› by contradiction
 qed

 fix d :: gauss-int
 assume d: d dvd z d dvd gauss-cnj z
 show is-unit d
 proof (rule ccontr)
  assume ¬is-unit d
  moreover from d assms have d ≠ 0
    by auto
  ultimately obtain p where p: prime p p dvd d
    using prime-divisorE by blast
  with d have p ∈ prime-factors z p ∈ prime-factors (gauss-cnj z)
    using assms by (auto simp: in-prime-factors-iff)
  with *[of p] show False by blast
 qed
qed
```

1.8 Square decompositions of prime numbers congruent 1 mod 4

```
lemma prime-1-mod-4-sum-of-squares-unique-aux:
 fixes p x y :: nat
 assumes prime p [p = 1] (mod 4) x ^ 2 + y ^ 2 = p
 shows   x > 0 ∧ y > 0 ∧ x ≠ y
proof safe
 from assms show x > 0 y > 0
   by (auto intro!: Nat.gr0I simp: prime-power-iff)
```

next
assume $x = y$
with *assms* **have** $p = 2 * x \hat{} 2$
by *simp*
with ‹*prime p*› **have** $p = 2$
by (*auto dest*: *prime-product*)
with ‹[$p = 1$] (*mod 4*)› **show** *False*
by (*simp add*: *cong-def*)
qed

Any prime number congruent 1 modulo 4 can be written *uniquely* as a sum of two squares $x^2 + y^2$ (up to commutativity of the addition). Additionally, we have shown above that x and y are both positive and $x \neq y$.

lemma *prime-1-mod-4-sum-of-squares-unique*:
fixes p :: *nat*
assumes *prime p* [$p = 1$] (*mod 4*)
shows $\exists!(x,y).\ x \leq y \wedge x \hat{} 2 + y \hat{} 2 = p$
proof (*rule ex-ex1I*)
obtain z **where** z: *gauss-int-norm z* = p
using *prime-cong-1-mod-4-gauss-int-norm-exists*[*OF assms*] **by** *blast*
show $\exists z.$ *case z of* $(x,y) \Rightarrow x \leq y \wedge x \hat{} 2 + y \hat{} 2 = p$
proof (*cases* $|ReZ\ z| \leq |ImZ\ z|$)
case *True*
with z **show** *?thesis* **by**
(*intro exI*[*of* - (*nat* $|ReZ\ z|$, *nat* $|ImZ\ z|$)])
(*auto simp*: *gauss-int-norm-def nat-add-distrib simp flip*: *nat-power-eq*)
next
case *False*
with z **show** *?thesis* **by**
(*intro exI*[*of* - (*nat* $|ImZ\ z|$, *nat* $|ReZ\ z|$)])
(*auto simp*: *gauss-int-norm-def nat-add-distrib simp flip*: *nat-power-eq*)
qed
next
fix *z1 z2*
assume *z1*: *case z1 of* $(x, y) \Rightarrow x \leq y \wedge x^2 + y^2 = p$
assume *z2*: *case z2 of* $(x, y) \Rightarrow x \leq y \wedge x^2 + y^2 = p$
define *z1'* :: *gauss-int* **where** *z1'* = *of-nat* (*fst z1*) + $\mathrm{i}_{\mathbb{Z}}$ * *of-nat* (*snd z1*)
define *z2'* :: *gauss-int* **where** *z2'* = *of-nat* (*fst z2*) + $\mathrm{i}_{\mathbb{Z}}$ * *of-nat* (*snd z2*)
from *assms* **interpret** *noninert-gauss-int-prime p*
by *unfold-locales auto*
have *norm-z1'*: *gauss-int-norm z1'* = p
using *z1* **by** (*simp add*: *z1'-def gauss-int-norm-def case-prod-unfold nat-add-distrib nat-power-eq*)
have *norm-z2'*: *gauss-int-norm z2'* = p
using *z2* **by** (*simp add*: *z2'-def gauss-int-norm-def case-prod-unfold nat-add-distrib nat-power-eq*)

have *sgns*: *fst z1* > *0 snd z1* > *0 fst z2* > *0 snd z2* > *0 fst z1* ≠ *snd z1 fst z2* ≠ *snd z2*

```
  using prime-1-mod-4-sum-of-squares-unique-aux[OF assms, of fst z1 snd z1] z1
        prime-1-mod-4-sum-of-squares-unique-aux[OF assms, of fst z2 snd z2] z2
by auto
  have [simp]: normalize z1' = z1' normalize z2' = z2'
    using sgns by (subst normalized-gauss-int-iff; simp add: z1'-def z2'-def)+
  have prime z1' prime z2'
    using norm-z1' norm-z2' assms unfolding prime-def
    by (auto simp: prime-gauss-int-norm-imp-prime-elem)

  have of-nat p = z1' * gauss-cnj z1'
    by (simp add: self-mult-gauss-cnj norm-z1')
  hence z1' dvd of-nat p
    by simp
  also have of-nat p = z2' * gauss-cnj z2'
    by (simp add: self-mult-gauss-cnj norm-z2')
  finally have z1' dvd z2' ∨ z1' dvd gauss-cnj z2' using assms
    by (subst (asm) prime-elem-dvd-mult-iff)
       (simp add: norm-z1' prime-gauss-int-norm-imp-prime-elem)
  thus z1 = z2
  proof
    assume z1' dvd z2'
    with ‹prime z1'› ‹prime z2'› have z1' = z2'
      by (simp add: primes-dvd-imp-eq)
    thus ?thesis
      by (simp add: z1'-def z2'-def gauss-int-eq-iff prod-eq-iff)
  next
    assume dvd: z1' dvd gauss-cnj z2'
    have normalize (iℤ * gauss-cnj z2') = iℤ * gauss-cnj z2'
      using sgns by (subst normalized-gauss-int-iff) (auto simp: z2'-def)
    moreover have prime-elem (iℤ * gauss-cnj z2')
      by (rule prime-gauss-int-norm-imp-prime-elem)
         (simp add: gauss-int-norm-mult norm-z2' ‹prime p›)
    ultimately have prime (iℤ * gauss-cnj z2')
      by (simp add: prime-def)
    moreover from dvd have z1' dvd iℤ * gauss-cnj z2'
      by simp
    ultimately have z1' = iℤ * gauss-cnj z2'
      using ‹prime z1'› by (simp add: primes-dvd-imp-eq)
    hence False using z1 z2 sgns
      by (auto simp: gauss-int-eq-iff z1'-def z2'-def)
    thus ?thesis ..
  qed
qed

lemma two-sum-of-squares-nat-iff: (x :: nat) ^ 2 + y ^ 2 = 2 ⟷ x = 1 ∧ y =
1
proof
  assume eq: x ^ 2 + y ^ 2 = 2
  have square-neq-2: n ^ 2 ≠ 2 for n :: nat
```

```
  proof
    assume ∗: n ^ 2 = 2
    have prime (2 :: nat)
      by simp
    thus False by (subst (asm) ∗ [symmetric]) (auto simp: prime-power-iff)
  qed

  from eq have x ^ 2 < 2 ^ 2 y ^ 2 < 2 ^ 2
    by simp-all
  hence x < 2 y < 2
    using power2-less-imp-less[of x 2] power2-less-imp-less[of y 2] by auto
  moreover have x > 0 y > 0
    using eq square-neq-2[of x] square-neq-2[of y] by (auto intro!: Nat.gr0I)
  ultimately show x = 1 ∧ y = 1
    by auto
qed auto

lemma prime-sum-of-squares-unique:
  fixes p :: nat
  assumes prime p p = 2 ∨ [p = 1] (mod 4)
  shows   ∃!(x,y). x ≤ y ∧ x ^ 2 + y ^ 2 = p
  using assms(2)
proof
  assume [simp]: p = 2
  have ∗∗: (λ(x,y). x ≤ y ∧ x ^ 2 + y ^ 2 = p) = (λz. z = (1,1 :: nat))
    using two-sum-of-squares-nat-iff by (auto simp: fun-eq-iff)
  thus ?thesis
    by (subst ∗∗) auto
qed (use prime-1-mod-4-sum-of-squares-unique[of p] assms in auto)
```

We now give a simple and inefficient algorithm to compute the canonical decomposition $x^2 + y^2$ with $x \leq y$.

```
definition prime-square-sum-nat-decomp :: nat ⇒ nat × nat where
  prime-square-sum-nat-decomp p =
     (if prime p ∧ (p = 2 ∨ [p = 1] (mod 4))
      then THE (x,y). x ≤ y ∧ x ^ 2 + y ^ 2 = p else (0, 0))

lemma prime-square-sum-nat-decomp-eqI:
  assumes prime p x ^ 2 + y ^ 2 = p x ≤ y
  shows   prime-square-sum-nat-decomp p = (x, y)
proof −
  have [gauss-int-norm (of-nat x + iℤ ∗ of-nat y) ≠ 3] (mod 4)
    by (rule gauss-int-norm-not-3-mod-4)
  also have gauss-int-norm (of-nat x + iℤ ∗ of-nat y) = p
    using assms by (auto simp: gauss-int-norm-def nat-add-distrib nat-power-eq)
  finally have [p ≠ 3] (mod 4) .
  with prime-mod-4-cases[of p] assms have ∗: p = 2 ∨ [p = 1] (mod 4)
    by auto
```

have *prime-square-sum-nat-decomp p* = (*THE* (*x*,*y*). *x* ≤ *y* ∧ *x* ^ 2 + *y* ^ 2 = *p*)
using ∗ ‹*prime p*› **by** (*simp add*: *prime-square-sum-nat-decomp-def*)
also have ... = (*x*, *y*)
proof (*rule the1-equality*)
show ∃!(*x*,*y*). *x* ≤ *y* ∧ *x* ^ 2 + *y* ^ 2 = *p*
using ‹*prime p*› ∗ **by** (*rule prime-sum-of-squares-unique*)
qed (*use assms* **in** *auto*)
finally show *?thesis* **.**
qed

lemma *prime-square-sum-nat-decomp-correct*:
assumes *prime p p* = 2 ∨ [*p* = 1] (*mod* 4)
defines *z* ≡ *prime-square-sum-nat-decomp p*
shows *fst z* ^ 2 + *snd z* ^ 2 = *p fst z* ≤ *snd z*
proof −
define *z*′ **where** *z*′ = (*THE* (*x*,*y*). *x* ≤ *y* ∧ *x* ^ 2 + *y* ^ 2 = *p*)
have *z* = *z*′
unfolding *z-def z*′*-def* **using** *assms* **by** (*simp add*: *prime-square-sum-nat-decomp-def*)
also have∃!(*x*,*y*). *x* ≤ *y* ∧ *x* ^ 2 + *y* ^ 2 = *p*
using *assms* **by** (*intro prime-sum-of-squares-unique*)
hence *case z*′ *of* (*x*, *y*) ⇒ *x* ≤ *y* ∧ *x* ^ 2 + *y* ^ 2 = *p*
unfolding *z*′*-def* **by** (*rule theI*′)
finally show *fst z* ^ 2 + *snd z* ^ 2 = *p fst z* ≤ *snd z*
by *auto*
qed

lemma *sum-of-squares-nat-bound*:
fixes *x y n* :: *nat*
assumes *x* ^ 2 + *y* ^ 2 = *n*
shows *x* ≤ *n*
proof (*cases x* = 0)
case *False*
hence *x* ∗ 1 ≤ *x* ^ 2
unfolding *power2-eq-square* **by** (*intro mult-mono*) *auto*
also have ... ≤ *x* ^ 2 + *y* ^ 2
by *simp*
also have ... = *n*
by *fact*
finally show *?thesis* **by** *simp*
qed *auto*

lemma *sum-of-squares-nat-bound*′:
fixes *x y n* :: *nat*
assumes *x* ^ 2 + *y* ^ 2 = *n*
shows *y* ≤ *n*
using *sum-of-squares-nat-bound*[*of y x*] *assms* **by** (*simp add*: *add.commute*)

lemma *is-singleton-conv-Ex1*:

is-singleton A $\longleftrightarrow$ $(\exists!x.\ x \in A)$
proof
assume *is-singleton A*
thus $\exists!x.\ x \in A$
by (*auto elim*!: *is-singletonE*)
next
assume $\exists!x.\ x \in A$
thus *is-singleton A*
by (*metis equals0D is-singletonI'*)
qed

lemma *the-elemI*:
assumes *is-singleton A*
shows *the-elem A* $\in$ *A*
using *assms* **by** (*elim is-singletonE*) *auto*

lemma *prime-square-sum-nat-decomp-code-aux*:
assumes *prime p p* = *2* $\vee$ [*p* = *1*] (*mod 4*)
defines *z* $\equiv$ *the-elem* (*Set.filter* (λ(*x*,*y*). *x* ^ *2* + *y* ^ *2* = *p*) (*SIGMA x*:{*0*..*p*}. {*x*..*p*}))
shows *prime-square-sum-nat-decomp p* = *z*
proof −
let *?A* = *Set.filter* (λ(*x*,*y*). *x* ^ *2* + *y* ^ *2* = *p*) (*SIGMA x*:{*0*..*p*}. {*x*..*p*})
have *eq*: *?A* = {(*x*,*y*). *x* $\leq$ *y* $\wedge$ *x* ^ *2* + *y* ^ *2* = *p*}
using *sum-of-squares-nat-bound* [*of* - - *p*] *sum-of-squares-nat-bound'* [*of* - - *p*]
by *auto*
have *z*: *z* $\in$ *Set.filter* (λ(*x*,*y*). *x* ^ *2* + *y* ^ *2* = *p*) (*SIGMA x*:{*0*..*p*}. {*x*..*p*})
unfolding *z-def eq* **using** *prime-sum-of-squares-unique*[*OF assms*(*1*,*2*)]
by (*intro the-elemI*) (*simp add*: *is-singleton-conv-Ex1*)
have *prime-square-sum-nat-decomp p* = (*fst z*, *snd z*)
using *z* **by** (*intro prime-square-sum-nat-decomp-eqI*[*OF assms*(*1*)]) *auto*
also have ... = *z*
by *simp*
finally show *?thesis* **.**
qed

lemma *prime-square-sum-nat-decomp-code* [*code*]:
prime-square-sum-nat-decomp p =
(*if prime p* $\wedge$ (*p* = *2* $\vee$ [*p* = *1*] (*mod 4*))
then the-elem (*Set.filter* (λ(*x*,*y*). *x* ^ *2* + *y* ^ *2* = *p*) (*SIGMA x*:{*0*..*p*}. {*x*..*p*}))
else (*0*, *0*))
using *prime-square-sum-nat-decomp-code-aux*[*of p*]
by (*auto simp*: *prime-square-sum-nat-decomp-def*)

1.9 Executable factorisation of Gaussian integers

Lastly, we use all of the above to give an executable (albeit not very efficient) factorisation algorithm for Gaussian integers based on factorisation of regular integers. Note that we will only compute the set of prime factors

without multiplicity, but given that, it would be fairly easy to determine the multiplicity as well.

First, we need the following function that computes the Gaussian integer factors of a $\mathbb{Z}$-prime p:

```
definition factor-gauss-int-prime-nat :: nat ⇒ gauss-int list where
  factor-gauss-int-prime-nat p =
    (if p = 2 then [1 + iℤ]
     else if [p = 3] (mod 4) then [of-nat p]
     else case prime-square-sum-nat-decomp p of
           (x, y) ⇒ [of-nat x + iℤ * of-nat y, of-nat y + iℤ * of-nat x])

lemma factor-gauss-int-prime-nat-correct:
  assumes prime p
  shows   set (factor-gauss-int-prime-nat p) = prime-factors (of-nat p)
  using prime-mod-4-cases[OF assms]
proof (elim disjE)
  assume p = 2
  thus ?thesis
    by (auto simp: prime-factorization-2-gauss-int factor-gauss-int-prime-nat-def)
next
  assume *: [p = 3] (mod 4)
  with assms have prime (of-nat p :: gauss-int)
    by (intro prime-gauss-int-of-nat)
  thus ?thesis using assms *
   by (auto simp: prime-factorization-prime factor-gauss-int-prime-nat-def cong-def)
next
  assume *: [p = 1] (mod 4)
  then interpret noninert-gauss-int-prime p
    using ‹prime p› by unfold-locales
  define z where z = prime-square-sum-nat-decomp p
  define x y where x = fst z and y = snd z
  have xy: x ^ 2 + y ^ 2 = p x ≤ y
    using prime-square-sum-nat-decomp-correct[of p] * assms
    by (auto simp: x-def y-def z-def)
  from xy have xy-signs: x > 0 y > 0
    using prime-1-mod-4-sum-of-squares-unique-aux[of p x y] assms * by auto
  have norms: gauss-int-norm (of-nat x + iℤ * of-nat y) = p
              gauss-int-norm (of-nat y + iℤ * of-nat x) = p
    using xy by (auto simp: gauss-int-norm-def nat-add-distrib nat-power-eq)
  have prime: prime (of-nat x + iℤ * of-nat y) prime (of-nat y + iℤ * of-nat x)
    using norms xy-signs ‹prime p› unfolding prime-def normalized-gauss-int-iff
    by (auto intro!: prime-gauss-int-norm-imp-prime-elem)

  have normalize ((of-nat x + iℤ * of-nat y) * (of-nat y + iℤ * of-nat x)) = of-nat
p
  proof -
    have (of-nat x + iℤ * of-nat y) * (of-nat y + iℤ * of-nat x) = (iℤ * of-nat p ::
gauss-int)
```

```
    by (subst xy(1) [symmetric]) (auto simp: gauss-int-eq-iff power2-eq-square)
  also have normalize ... = of-nat p
    by simp
  finally show ?thesis .
 qed
 hence prime-factorization (of-nat p) =
       prime-factorization (prod-mset {#of-nat x + iℤ * of-nat y, of-nat y + iℤ *
of-nat x#})
  using assms xy by (subst prime-factorization-unique) (auto simp: gauss-int-eq-iff)
 also have ... = {#of-nat x + iℤ * of-nat y, of-nat y + iℤ * of-nat x#}
  using prime by (subst prime-factorization-prod-mset-primes) auto
 finally have prime-factors (of-nat p) = {of-nat x + iℤ * of-nat y, of-nat y + iℤ
* of-nat x}
  by simp
 also have ... = set (factor-gauss-int-prime-nat p)
  using * unfolding factor-gauss-int-prime-nat-def case-prod-unfold
  by (auto simp: cong-def x-def y-def z-def)
 finally show ?thesis ..
qed
```

Next, we lift this to compute the prime factorisation of any integer in the Gaussian integers:

```
definition prime-factors-gauss-int-of-nat :: nat ⇒ gauss-int set where
 prime-factors-gauss-int-of-nat n = (if n = 0 then {} else
   (⋃p∈prime-factors n. set (factor-gauss-int-prime-nat p)))

lemma prime-factors-gauss-int-of-nat-correct:
 prime-factors-gauss-int-of-nat n = prime-factors (of-nat n)
proof (cases n = 0)
 case False
 from False have [simp]: n > 0 by auto
 have prime-factors (of-nat n :: gauss-int) =
       prime-factors (of-nat (prod-mset (prime-factorization n)))
  by (subst prod-mset-prime-factorization-nat [symmetric]) auto
 also have ... = prime-factors (prod-mset (image-mset of-nat (prime-factorization
n)))
  by (subst of-nat-prod-mset) auto
 also have ... = (⋃p∈prime-factors n. prime-factors (of-nat p))
  by (subst prime-factorization-prod-mset) auto
 also have ... = (⋃p∈prime-factors n. set (factor-gauss-int-prime-nat p))
  by (intro SUP-cong refl factor-gauss-int-prime-nat-correct [symmetric]) auto
 finally show ?thesis by (simp add: prime-factors-gauss-int-of-nat-def)
qed (auto simp: prime-factors-gauss-int-of-nat-def)
```

We can now use this to factor any Gaussian integer by computing a factorisation of its norm and removing all the prime divisors that do not actually divide it.

```
definition prime-factors-gauss-int :: gauss-int ⇒ gauss-int set where
 prime-factors-gauss-int z = (if z = 0 then {}
```

else Set.filter (λ*p*. *p dvd z*) (*prime-factors-gauss-int-of-nat* (*gauss-int-norm z*)))

lemma *prime-factors-gauss-int-correct* [*code-unfold*]: *prime-factors z* = *prime-factors-gauss-int z*
proof (*cases z* = *0*)
case [*simp*]: *False*
define *n* **where** *n* = *gauss-int-norm z*
from *False* **have** [*simp*]: *n* > *0* **by** (*auto simp*: *n-def*)

have *prime-factors-gauss-int z* = *Set.filter* (λ*p*. *p dvd z*) (*prime-factors* (*of-nat n*))
by (*simp add*: *prime-factors-gauss-int-of-nat-correct prime-factors-gauss-int-def n-def*)
also have *of-nat n* = *z* ∗ *gauss-cnj z*
by (*simp add*: *n-def self-mult-gauss-cnj*)
also have *prime-factors* ... = *prime-factors z* ∪ *prime-factors* (*gauss-cnj z*)
by (*subst prime-factors-product*) *auto*
also have *Set.filter* (λ*p*. *p dvd z*) ... = *prime-factors z*
by (*auto simp*: *in-prime-factors-iff*)
finally show *?thesis* **by** *simp*
qed (*auto simp*: *prime-factors-gauss-int-def*)

end

theory *Gaussian-Integers-Test*
imports
Gaussian-Integers
Polynomial-Factorization.Prime-Factorization
HOL−Library.Code-Target-Numeral
begin

Lastly, we apply our factorisation algorithm to some simple examples:

value (*1234* + *5678* ∗ $\mathrm{i}_{\mathbb{Z}}$) *mod* (*321* + *654* ∗ $\mathrm{i}_{\mathbb{Z}}$)
value *prime-factors* (*1* + *3* ∗ $\mathrm{i}_{\mathbb{Z}}$)
value *prime-factors* (*4830* + *1610* ∗ $\mathrm{i}_{\mathbb{Z}}$)

end

1.10 Sums of two squares

theory *Gaussian-Integers-Sums-Of-Two-Squares*
imports *Gaussian-Integers*
begin

As an application, we can now easily prove that a positive natural number is the sum of two squares if and only if all prime factors congruent 3 modulo 4 have even multiplicity.

inductive *sum-of-2-squares-nat* :: *nat* ⇒ *bool* **where**

sum-of-2-squares-nat (*a* ^ *2* + *b* ^ *2*)

lemma *sum-of-2-squares-nat-altdef*: *sum-of-2-squares-nat n* ⟷ *n* ∈ *range gauss-int-norm*
proof (*safe elim*!: *sum-of-2-squares-nat.cases*)
fix *a b* :: *nat*
have *a* ^ *2* + *b* ^ *2* = *gauss-int-norm* (*of-nat a* + $\mathrm{i}_{\mathbb{Z}}$ ∗ *of-nat b*)
by (*auto simp*: *gauss-int-norm-def nat-add-distrib nat-power-eq*)
thus *a* ^ *2* + *b* ^ *2* ∈ *range gauss-int-norm* **by** *blast*
next
fix *z* :: *gauss-int*
have *gauss-int-norm z* = *nat* |*ReZ z*| ^ *2* + *nat* |*ImZ z*| ^ *2*
by (*auto simp*: *gauss-int-norm-def nat-add-distrib simp flip*: *nat-power-eq*)
thus *sum-of-2-squares-nat* (*gauss-int-norm z*)
by (*auto intro*: *sum-of-2-squares-nat.intros*)
qed

lemma *sum-of-2-squares-nat-gauss-int-norm* [*intro*]: *sum-of-2-squares-nat* (*gauss-int-norm z*)
by (*auto simp*: *sum-of-2-squares-nat-altdef*)

lemma *sum-of-2-squares-nat-0* [*simp*, *intro*]: *sum-of-2-squares-nat 0*
and *sum-of-2-squares-nat-1* [*simp*, *intro*]: *sum-of-2-squares-nat 1*
and *sum-of-2-squares-nat-Suc-0* [*simp*, *intro*]: *sum-of-2-squares-nat* (*Suc 0*)
and *sum-of-2-squares-nat-2* [*simp*, *intro*]: *sum-of-2-squares-nat 2*
using *sum-of-2-squares-nat.intros*[*of 0 0*] *sum-of-2-squares-nat.intros*[*of 0 1*]
sum-of-2-squares-nat.intros[*of 1 1*] **by** (*simp-all add*: *numeral-2-eq-2*)

lemma *sum-of-2-squares-nat-mult* [*intro*]:
assumes *sum-of-2-squares-nat x sum-of-2-squares-nat y*
shows *sum-of-2-squares-nat* (*x* ∗ *y*)
proof −
from *assms* **obtain** *z1 z2* **where** *x* = *gauss-int-norm z1 y* = *gauss-int-norm z2*
by (*auto simp*: *sum-of-2-squares-nat-altdef*)
hence *x* ∗ *y* = *gauss-int-norm* (*z1* ∗ *z2*)
by (*simp add*: *gauss-int-norm-mult*)
thus *?thesis* **by** *auto*
qed

lemma *sum-of-2-squares-nat-power* [*intro*]:
assumes *sum-of-2-squares-nat m*
shows *sum-of-2-squares-nat* (*m* ^ *n*)
using *assms* **by** (*induction n*) *auto*

lemma *sum-of-2-squares-nat-prod* [*intro*]:
assumes ⋀*x*. *x* ∈ *A* ⟹ *sum-of-2-squares-nat* (*f x*)
shows *sum-of-2-squares-nat* (∏ *x*∈*A*. *f x*)
using *assms* **by** (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *sum-of-2-squares-nat-prod-mset* [*intro*]:

assumes $\bigwedge x.\ x \in\# A \Longrightarrow$ *sum-of-2-squares-nat x*
shows *sum-of-2-squares-nat* (*prod-mset A*)
using *assms* **by** (*induction A*) *auto*

lemma *sum-of-2-squares-nat-necessary*:
assumes *sum-of-2-squares-nat n n* > *0*
assumes *prime p* [*p* = *3*] (*mod 4*)
shows *even* (*multiplicity p n*)
proof −
define *k* **where** *k* = *multiplicity p n*
from *assms* **obtain** *z* **where** *z*: *gauss-int-norm z* = *n*
by (*auto simp*: *sum-of-2-squares-nat-altdef*)
from *assms* **and** *z* **have** [*simp*]: *z* ≠ *0*
by *auto*
have *prime′*: *prime* (*of-nat p* :: *gauss-int*)
using *assms prime-gauss-int-of-nat* **by** *blast*
have [*simp*]: *multiplicity* (*of-nat p*) (*gauss-cnj z*) = *multiplicity* (*of-nat p*) *z*
using *multiplicity-gauss-cnj*[*of of-nat p z*] **by** *simp*
have *multiplicity* (*of-nat p*) (*of-nat n* :: *gauss-int*) =
multiplicity (*of-nat p*) (*z* ∗ *gauss-cnj z*)
using *z* **by** (*simp add*: *self-mult-gauss-cnj*)
also have ... = *2* ∗ *multiplicity* (*of-nat p*) *z*
using *prime′* **by** (*subst prime-elem-multiplicity-mult-distrib*) *auto*
finally have *multiplicity p n* = *2* ∗ *multiplicity* (*of-nat p*) *z*
by (*subst* (*asm*) *multiplicity-gauss-int-of-nat*)
thus *?thesis* **by** *auto*
qed

lemma *sum-of-2-squares-nat-sufficient*:
fixes *n* :: *nat*
assumes *n* > *0*
assumes $\bigwedge p.\ p \in$ *prime-factors n* $\Longrightarrow$ [*p* = *3*] (*mod 4*) $\Longrightarrow$ *even* (*multiplicity p n*)
shows *sum-of-2-squares-nat n*
proof −
define *P2* **where** *P2* = {*p*∈*prime-factors n*. [*p* = *1*] (*mod 4*)}
define *P3* **where** *P3* = {*p*∈*prime-factors n*. [*p* = *3*] (*mod 4*)}
from ‹*n* > *0*› **have** *n* = ($\prod$ *p*∈*prime-factors n*. *p* ^ *multiplicity p n*)
by (*subst prime-factorization-nat*) *auto*
also have ... = ($\prod$ *p*∈{*2*}∪*P2*∪*P3*. *p* ^ *multiplicity p n*)
using *prime-mod-4-cases*
by (*intro prod.mono-neutral-left*)
(*auto simp*: *P2-def P3-def in-prime-factors-iff not-dvd-imp-multiplicity-0*)
also have ... = ($\prod$ *p*∈{*2*}∪*P2*. *p* ^ *multiplicity p n*) ∗ ($\prod$ *p*∈*P3*. *p* ^ *multiplicity p n*)
by (*intro prod.union-disjoint*) (*auto simp*: *P2-def P3-def cong-def*)
also have ($\prod$ *p*∈{*2*}∪*P2*. *p* ^ *multiplicity p n*) =
2 ^ *multiplicity 2 n* ∗ ($\prod$ *p*∈*P2*. *p* ^ *multiplicity p n*)
by (*subst prod.union-disjoint*) (*auto simp*: *P2-def cong-def*)

```
  also have (∏p∈P3. p ^ multiplicity p n) = (∏p∈P3. (p ^ 2) ^ (multiplicity p
n div 2))
  proof (intro prod.cong refl)
    fix p :: nat assume p: p ∈ P3
    have (p ^ 2) ^ (multiplicity p n div 2) = p ^ (2 * (multiplicity p n div 2))
      by (simp add: power-mult)
    also have even (multiplicity p n)
      using assms p by (auto simp: P3-def)
    hence 2 * (multiplicity p n div 2) = multiplicity p n
      by simp
    finally show p ^ multiplicity p n = (p ^ 2) ^ (multiplicity p n div 2)
      by simp
  qed
  finally have n = 2 ^ multiplicity 2 n * (∏p∈P2. p ^ multiplicity p n) *
                   (∏p∈P3. p^2 ^ (multiplicity p n div 2)) .

  also have sum-of-2-squares-nat ...
 proof (intro sum-of-2-squares-nat-mult sum-of-2-squares-nat-prod; rule sum-of-2-squares-nat-power)
    fix p :: nat assume p: p ∈ P2
   with prime-cong-1-mod-4-gauss-int-norm-exists[of p] show sum-of-2-squares-nat
p
      by (auto simp: P2-def)
  next
    fix p :: nat assume p: p ∈ P3
    have sum-of-2-squares-nat (gauss-int-norm (of-nat p)) ..
    also have gauss-int-norm (of-nat p) = p ^ 2
      by simp
    finally show sum-of-2-squares-nat (p ^ 2) .
  qed auto
  finally show ?thesis .
qed

theorem sum-of-2-squares-nat-iff:
  sum-of-2-squares-nat n ⟷
     n = 0 ∨ (∀p∈prime-factors n. [p = 3] (mod 4) ⟶ even (multiplicity p n))
  using sum-of-2-squares-nat-necessary[of n] sum-of-2-squares-nat-sufficient[of n]
by auto

end
```

1.11 Primitive Pythagorean triples

```
theory Gaussian-Integers-Pythagorean-Triples
  imports Gaussian-Integers
begin
```

In this section, we derive Euclid's formula for primitive Pythagorean triples using Gaussian integers, following Stillwell [1].

```
definition prim-pyth-triple :: nat ⇒ nat ⇒ nat ⇒ bool where
```

prim-pyth-triple x y z $\longleftrightarrow$ $x > 0 \wedge y > 0 \wedge$ *coprime x y* $\wedge\ x^2 + y^2 = z^2$

lemma *prim-pyth-triple-commute*: *prim-pyth-triple x y z* $\longleftrightarrow$ *prim-pyth-triple y x z*
by (*simp add*: *prim-pyth-triple-def coprime-commute add-ac conj-ac*)

lemma *prim-pyth-triple-aux*:
fixes *u v* :: *nat*
assumes $v \leq u$
shows $(2 * u * v) \hat{} 2 + (u \hat{} 2 - v \hat{} 2) \hat{} 2 = (u \hat{} 2 + v \hat{} 2) \hat{} 2$
proof −
have *int* $((2 * u * v) \hat{} 2 + (u \hat{} 2 - v \hat{} 2) \hat{} 2) =$
$(2 * int\ u * int\ v) \hat{} 2 + (int\ u \hat{} 2 - int\ v \hat{} 2) \hat{} 2$
using *assms* **by** (*simp add*: *of-nat-diff*)
also have $\ldots = (int\ u \hat{} 2 + int\ v \hat{} 2) \hat{} 2$
by (*simp add*: *power2-eq-square algebra-simps*)
also have $\ldots =$ *int* $((u \hat{} 2 + v \hat{} 2) \hat{} 2)$
by *simp*
finally show *?thesis*
by (*simp only*: *of-nat-eq-iff*)
qed

lemma *prim-pyth-tripleI1*:
assumes $0 < v\ v < u$ *coprime u v* $\neg$(*odd u* $\wedge$ *odd v*)
shows *prim-pyth-triple* $(2 * u * v)\ (u^2 - v^2)\ (u^2 + v^2)$
proof −
have $v \hat{} 2 < u \hat{} 2$
using *assms* **by** (*intro power-strict-mono*) *auto*
hence $\neg u \hat{} 2 < v \hat{} 2$ **by** *linarith*

from *assms* **have** *coprime* (*int u*) (*int* $v \hat{} 2$)
by *auto*
hence *coprime* (*int u*) (*int u* $*$ *int u* + $(-(int\ v \hat{} 2))$)
unfolding *coprime-iff-gcd-eq-1* **by** (*subst gcd-add-mult*) *auto*
also have *int u* $*$ *int u* + $(-(int\ v \hat{} 2)) = int\ (u \hat{} 2 - v \hat{} 2)$
using ‹$v < u$› **by** (*simp add*: *of-nat-diff flip*: *power2-eq-square*)
finally have *coprime1*: *coprime u* $(u \hat{} 2 - v \hat{} 2)$
by *auto*

from *assms* **have** *coprime* (*int v*) (*int* $u \hat{} 2$)
by (*auto simp*: *coprime-commute*)
hence *coprime* (*int v*) $((-int\ v) * int\ v + int\ u \hat{} 2)$
unfolding *coprime-iff-gcd-eq-1* **by** (*subst gcd-add-mult*) *auto*
also have $(-int\ v) * int\ v + int\ u \hat{} 2 = int\ (u \hat{} 2 - v \hat{} 2)$
using ‹$v < u$› **by** (*simp add*: *of-nat-diff flip*: *power2-eq-square*)
finally have *coprime2*: *coprime v* $(u \hat{} 2 - v \hat{} 2)$
by *auto*

have $(2 * u * v) \hat{} 2 + (u \hat{} 2 - v \hat{} 2) \hat{} 2 = (u \hat{} 2 + v \hat{} 2) \hat{} 2$

```
    using ‹v < u› by (intro prim-pyth-triple-aux) auto
  moreover have coprime (2 * u * v) (u ^ 2 - v ^ 2)
    using assms ‹¬u ^ 2 < v ^ 2› coprime1 coprime2 by auto
  ultimately show ?thesis using assms ‹v ^ 2 < u ^ 2›
    by (simp add: prim-pyth-triple-def)
qed

lemma prim-pyth-tripleI2:
  assumes 0 < v v < u coprime u v ¬(odd u ∧ odd v)
  shows   prim-pyth-triple (u^2 - v^2) (2 * u * v) (u^2 + v^2)
  using prim-pyth-tripleI1[OF assms] by (simp add: prim-pyth-triple-commute)

lemma primitive-pythagorean-tripleE-int:
  assumes z ^ 2 = x ^ 2 + y ^ 2
  assumes coprime x y
  obtains u v :: int
    where coprime u v and ¬(odd u ∧ odd v)
      and x = 2 * u * v ∧ y = u^2 - v^2  ∨  x = u^2 - v^2 ∧ y = 2 * u * v
proof -
  have ¬(even x ∧ even y)
    using not-coprimeI[of 2 x y] ‹coprime x y› by auto
  moreover have ¬(odd x ∧ odd y)
  proof safe
    assume odd x odd y
    hence [x ^ 2 + y ^ 2 = 1 + 1] (mod 4)
      by (intro cong-add odd-square-cong-4-int)
    hence [z ^ 2 = 2] (mod 4)
      by (simp add: assms)
    moreover have [z ^ 2 = 0] (mod 4) ∨ [z ^ 2 = 1] (mod 4)
      using even-square-cong-4-int[of z] odd-square-cong-4-int[of z]
      by (cases even z) auto
    ultimately show False
      by (auto simp: cong-def)
  qed
  ultimately have even y ⟷ odd x
    by blast

  have even z ⟷ even (z ^ 2)
    by auto
  also have even (z ^ 2) ⟷ even (x ^ 2 + y ^ 2)
    by (subst assms(1)) auto
  finally have odd z
    by (cases even x) (auto simp: ‹even y ⟷ ¬even x›)

  define t where t = of-int x + iℤ * of-int y
  from assms have t-mult-cnj: t * gauss-cnj t = of-int z ^ 2
    by (simp add: t-def power2-eq-square algebra-simps flip: of-int-mult of-int-add)

  have gauss-int-norm t = z ^ 2
```

by (*simp add*: *gauss-int-norm-def t-def assms*)
with ‹*coprime x y*› **and** ‹*odd z*› **have** *coprime t* (*gauss-cnj t*)
by (*intro coprime-self-gauss-cnj*)
(*auto simp*: *t-def gauss-int-norm-def assms*(*1*) [*symmetric*] *even-nat-iff*)
moreover have *is-square* (*t* ∗ *gauss-cnj t*)
by (*subst t-mult-cnj*) *auto*
hence *is-nth-power-upto-unit 2* (*t* ∗ *gauss-cnj t*)
by (*auto intro*: *is-nth-power-upto-unit-base*)
ultimately have *is-nth-power-upto-unit 2 t*
by (*rule is-nth-power-upto-unit-mult-coprimeD1*)
then obtain *a b* **where** *ab*: *is-unit a a* ∗ *t* = *b* ^ *2*
by (*auto simp*: *is-nth-power-upto-unit-def is-nth-power-def*)
from *ab*(*1*) **have** *a* ∈ {*1*, −*1*, $\mathrm{i}_{\mathbb{Z}}$, −$\mathrm{i}_{\mathbb{Z}}$}
by (*auto simp*: *is-unit-gauss-int-iff*)
then obtain *u v* :: *int* **where** *ReZ t* = *2* ∗ *u* ∗ *v* ∧ *ImZ t* = *u* ^ *2* − *v* ^ *2* ∨
ImZ t = *2* ∗ *u* ∗ *v* ∧ *ReZ t* = *u* ^ *2* − *v* ^ *2*
proof *safe*
assume [*simp*]: *a* = *1*
have *ReZ t* = *ReZ b* ^ *2* − *ImZ b* ^ *2 ImZ t* = *2* ∗ *ReZ b* ∗ *ImZ b* **using** *ab*(*2*)
by (*auto simp*: *gauss-int-eq-iff power2-eq-square*)
thus *?thesis* **using** *that* **by** *blast*
next
assume [*simp*]: *a* = −*1*
have *ReZ t* = *ImZ b* ^ *2* − (−*ReZ b*) ^ *2 ImZ t* = *2* ∗ *ImZ b* ∗ (−*ReZ b*)
using *ab*(*2*)
by (*auto simp*: *gauss-int-eq-iff power2-eq-square algebra-simps*)
thus *?thesis* **using** *that* **by** *blast*
next
assume [*simp*]: *a* = $\mathrm{i}_{\mathbb{Z}}$
hence *ImZ t* = *ImZ b* ^ *2* − *ReZ b* ^ *2 ReZ t* = *2* ∗ *ImZ b* ∗ *ReZ b* **using**
ab(*2*)
by (*auto simp*: *gauss-int-eq-iff power2-eq-square algebra-simps*)
thus *?thesis* **using** *that* **by** *blast*
next
assume [*simp*]: *a* = −$\mathrm{i}_{\mathbb{Z}}$
hence *ImZ t* = (−*ReZ b*) ^ *2* − *ImZ b* ^ *2 ReZ t* = *2* ∗ (−*ReZ b*) ∗ *ImZ b*
using *ab*(*2*)
by (*auto simp*: *gauss-int-eq-iff power2-eq-square algebra-simps*)
thus *?thesis* **using** *that* **by** *blast*
qed
also have *ReZ t* = *x*
by (*simp add*: *t-def*)
also have *ImZ t* = *y*
by (*simp add*: *t-def*)
finally have *xy*: *x* = *2* ∗ *u* ∗ *v* ∧ *y* = u^2 − v^2 ∨ *x* = u^2 − v^2 ∧ *y* = *2* ∗ *u* ∗ *v*
by *blast*

have *not-both-odd*: ¬(*odd u* ∧ *odd v*)
proof *safe*

```
    assume odd u odd v
    hence even x even y
      using xy by auto
    with ‹coprime x y› show False
      by auto
  qed

  have coprime u v
  proof (rule coprimeI)
    fix d assume d dvd u d dvd v
    hence d dvd (u^2 - v^2) d dvd 2 * u * v
      by (auto simp: power2-eq-square)
    with xy have d dvd x d dvd y
      by auto
    with ‹coprime x y› show is-unit d
      using not-coprimeI by blast
  qed
  with xy not-both-odd show ?thesis
    using that[of u v] by blast
qed

lemma prim-pyth-tripleE:
  assumes prim-pyth-triple x y z
  obtains u v :: nat
  where 0 < v and v < u and coprime u v and ¬(odd u ∧ odd v) and z = u^2 +
v^2
    and x = 2 * u * v ∧ y = u^2 - v^2  ∨  x = u^2 - v^2 ∧ y = 2 * u * v
proof -
  have *: (int z) ^ 2 = (int x) ^ 2 + (int y) ^ 2 coprime (int x) (int y)
   using assms by (auto simp flip: of-nat-power of-nat-add simp: prim-pyth-triple-def)
  obtain u v
    where uv: coprime u v ¬(odd u ∧ odd v)
            int x = 2 * u * v ∧ int y = u^2 - v^2  ∨  int x = u^2 - v^2 ∧ int y = 2
* u * v
    using primitive-pythagorean-tripleE-int[OF *] by metis
  define u' v' where u' = nat |u| and v' = nat |v|

  have **: a = 2 * u' * v' if int a = 2 * u * v for a
  proof -
    from that have nat |int a| = nat |2 * u * v|
      by (simp only: )
    thus a = 2 * u' * v'
      by (simp add: u'-def v'-def abs-mult nat-mult-distrib)
  qed
  have ***: a = u' ^ 2 - v' ^ 2 v' ≤ u' if int a = u ^ 2 - v ^ 2 for a
  proof -
    have v ^ 2 ≤ v ^ 2 + int a
      by simp
    also have ... = u ^ 2
```

```
    using that by simp
  finally have |v| ≤ |u|
    using abs-le-square-iff by blast
  thus v' ≤ u'
    by (simp add: v'-def u'-def)

  from that have u ^ 2 = v ^ 2 + int a
    by simp
  hence nat |u ^ 2| = nat |v ^ 2 + int a|
    by (simp only: )
  also have nat |u ^ 2| = u' ^ 2
    by (simp add: u'-def flip: nat-power-eq)
  also have nat |v ^ 2 + int a| = v' ^ 2 + a
    by (simp add: nat-add-distrib v'-def flip: nat-power-eq)
  finally show a = u' ^ 2 - v' ^ 2
    by simp
 qed

 have eq: x = 2 * u' * v' ∧ y = u'² - v'²  ∨  x = u'² - v'² ∧ y = 2 * u' * v'
and v' ≤ u'
  using uv(3) **[of x] **[of y] ***[of x] ***[of y] by blast+
 moreover have coprime u' v'
  using ‹coprime u v›
  by (auto simp: u'-def v'-def)
 moreover have ¬(odd u' ∧ odd v')
  using uv(2) by (auto simp: u'-def v'-def)
 moreover have v' ≠ u' v' > 0
  using ‹coprime u' v'› eq assms by (auto simp: prim-pyth-triple-def)
 moreover from this have v' < u'
  using ‹v' ≤ u'› by auto
 moreover have z = u'² + v'²
 proof -
  from assms have z ^ 2 = x ^ 2 + y ^ 2
    by (simp add: prim-pyth-triple-def)
  also have ... = (2 * u' * v') ^ 2 + (u' ^ 2 - v' ^ 2) ^ 2
    using eq by (auto simp: add-ac)
  also have ... = (u' ^ 2 + v' ^ 2) ^ 2
    by (intro prim-pyth-triple-aux) fact
  finally show ?thesis by simp
 qed
 ultimately show ?thesis using that[of v' u'] by metis
qed

theorem prim-pyth-triple-iff:
 prim-pyth-triple x y z ⟷
   (∃ u v. 0 < v ∧ v < u ∧ coprime u v ∧ ¬(odd u ∧ odd v) ∧
        (x = 2 * u * v ∧ y = u² - v²  ∨  x = u² - v² ∧ y = 2 * u * v) ∧ z =
u² + v²)
 (is - ⟷ ?rhs)
```

proof
assume *prim-pyth-triple x y z*
from *prim-pyth-tripleE*[*OF this*] **show** *?rhs* **by** *metis*
next
assume *?rhs*
then obtain *u v* **where** *uv*: $0 < v\ v < u$ *coprime u v* $\neg(odd\ u \wedge odd\ v)\ z = u^2 + v^2$ **and**
eq: $x = 2 * u * v \wedge y = u^2 - v^2 \ \vee \ x = u^2 - v^2 \wedge y = 2 * u * v$
by *metis*
thus *prim-pyth-triple x y z*
using *uv prim-pyth-tripleI1*[*OF uv*(1−4)] *prim-pyth-tripleI2*[*OF uv*(1−4)] *uv*(5) *eq* **by** *auto*
qed

end

theory *Gaussian-Integers-Everything*
imports
Gaussian-Integers
Gaussian-Integers-Test
Gaussian-Integers-Sums-Of-Two-Squares
Gaussian-Integers-Pythagorean-Triples
begin

end

References

[1] J. Stillwell. *The Gaussian integers*, pages 101–116. Springer New York, New York, NY, 2003.