Sophie Germain's Theorem

Benoît Ballenghien
Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF

October 13, 2025

Contents

1 Introduction

Fermat's Last Theorem (often abbreviated to FLT) states that for any integer $2 < n$, the equation $x^n + y^n = z^n$ has no nontrivial solution in the integers. Pierre de Fermat first conjectured this result in the 17th century, claiming to have a proof that did not fit in the margin of his notebook. However, it remained an open problem for centuries until Andrew Wiles and Richard Taylor provided a complete proof in 1995 using advanced techniques from algebraic geometry and modular forms.

But in the meantime, many mathematicians have made partial progress on the problem. In particular, Sophie Germain's theorem states that p is a prime such that $2 * p + 1$ is also a prime, then there are no integer solutions to the equation $x^p + y^p = z^p$ such that p divides neither x, y nor z.

This result is not only included in the extended list of Freek's "Top 100 theorems" [1], but is also very familiar to students taking the French "agrégation" mathematics competitive examination. Hoping that this submission might also be useful to them, we developed separately the classical version of the theorem as presented in [1] and a generalization that one can find for example in [2].

[1] http://www.cs.ru.nl/~freek/100/

The session displayed in 1 is organized as follows:

- `FLT_Sufficient_Conditions` provides sufficient conditions for proving FLT,
- `SG_Premilinaries` establish some useful lemmas and introduces the concept of Sophie Germain prime,
- `SG_Theorem` proves Sophie Germain's theorem and
- `SG_Generalization` gives a generalization of it.

2 Preliminaries

2.1 Coprimality

We start with this useful elimination rule: when *a* and *b* are not *coprime* and are not both equal to *0*, there exists some common *prime* factor.

lemma (**in** *factorial-semiring-gcd*) *not-coprime-nonzeroE* :
‹⟦¬ *coprime a b*; $a \neq 0 \vee b \neq 0$; ⋀*p*. *prime p* ⟹ *p dvd a* ⟹ *p dvd b* ⟹ *thesis*⟧ ⟹ *thesis*›
by (*metis gcd-eq-0-iff gcd-greatest-iff is-unit-gcd prime-divisor-exists*)

Still referring to the notion of *coprime* (but generalized to a set), we prove that when *Gcd A* ≠ *0*, the elements of {*a div Gcd A* |*a*. *a* ∈ *A*} are setwise *coprime*.

lemma (**in** *semiring-Gcd*) *GCD-div-Gcd-is-one* :
‹(*GCD a* ∈ *A*. *a div Gcd A*) = *1*› **if** ‹*Gcd A* ≠ *0*›
proof (*rule ccontr*)
assume ‹(*GCD a*∈*A*. *a div Gcd A*) ≠ *1*›
then obtain *d* **where** ‹¬ *is-unit d*› ‹∀ *a*∈*A*. *d dvd* (*a div Gcd A*)›
by (*metis* (*no-types*, *lifting*) *Gcd-dvd associated-eqI image-eqI normalize-1 normalize-Gcd one-dvd*)
from ‹∀ *a*∈*A*. *d dvd* (*a div Gcd A*)› **have** ‹∀ *a*∈*A*. *d* ∗ *Gcd A dvd a*›
by (*meson Gcd-dvd dvd-div-iff-mult* ‹*Gcd A* ≠ *0*›)
with *Gcd-greatest* **have** ‹*d* ∗ *Gcd A dvd Gcd A*› **by** *blast*
with ‹¬ *is-unit d*› **show** *False* **by** (*metis div-self dvd-mult-imp-div that*)
qed

2.2 Power

Now we want to characterize the fact of admitting an *n*-th root with a condition on the *multiplicity* of each prime factor.

lemma *exists-nth-root-iff* :
‹($\exists x$. *normalize* $y = x$ ^ n) $\longleftrightarrow$ ($\forall p \in$*prime-factors* y. n *dvd multiplicity* p y)›
if ‹$y \neq 0$› **for** y :: ‹$'a$ :: *factorial-semiring-multiplicative*›
proof (*rule iffI*)
show ‹$\exists x$. *normalize* $y = x$ ^ $n \Longrightarrow \forall p \in$*prime-factors* y. n *dvd multiplicity* p y›
proof (*elim exE*, *rule ballI*)
fix x p **assume** ‹*normalize* $y = x$ ^ n› **and** ‹$p \in$ *prime-factors* y›
hence ‹$p \in$ *prime-factors* x›
by (*metis prime-factorization-normalize empty-iff power-0 prime-factorization-1 prime-factors-power set-mset-empty zero-less-iff-neq-zero*)
with ‹*normalize* $y = x$ ^ n› **show** ‹n *dvd multiplicity* p y›
by (*metis dvd-def in-prime-factors-iff multiplicity-normalize-right normalization-semidom-class.prime-def prime-elem-multiplicity-power-distrib*)
qed
next
assume $*$: ‹$\forall p \in$*prime-factors* y. n *dvd multiplicity* p y›
define f **where** ‹f $p \equiv$ *multiplicity* p y *div* n› **for** p
have ‹*normalize* y = ($\prod p \in$*prime-factors* y. p ^ *multiplicity* p y)›
by (*fact prod-prime-factors*[*OF* ‹$y \neq 0$›, *symmetric*])
also have ‹... = ($\prod p \in$*prime-factors* y. p ^ (f p $*$ n))›
by (*rule prod.cong*[*OF refl*]) (*simp add*: $*$ *f-def*)
also have ‹... = ($\prod p \in$*prime-factors* y. p ^ f p) ^ n›
by (*simp add*: *power-mult prod-power-distrib*)
finally show ‹$\exists x$. *normalize* $y = x$ ^ n› **..**
qed

We use this result to obtain the following elimination rule.

corollary *prod-is-some-powerE* :
fixes a b :: ‹$'a$ :: *factorial-semiring-multiplicative*›
assumes ‹*coprime* a b› **and** ‹$a * b = x$ ^ n›
obtains α **where** ‹*normalize* $a = \alpha$ ^ n›
proof (*cases* ‹$a = 0$›)
from ‹$a * b = x$ ^ n› **show** ‹($\bigwedge \alpha$. *normalize* $a = \alpha$ ^ $n \Longrightarrow$ *thesis*) $\Longrightarrow a = 0$ $\Longrightarrow$ *thesis*› **by** *simp*
next
assume ‹$a \neq 0$› **and** *hyp* : ‹*normalize* $a = \alpha$ ^ $n \Longrightarrow$ *thesis*› **for** α
from ‹$a \neq 0$› **have** ‹$\exists \alpha$. *normalize* $a = \alpha$ ^ n›
proof (*rule exists-nth-root-iff*[*THEN iffD2*, *rule-format*])
fix p **assume** ‹$p \in$ *prime-factors* a›
with ‹$a * b = x$ ^ n› **have** ‹p *dvd* x›
by (*metis dvd-mult2 in-prime-factors-iff prime-dvd-power*)
hence ‹p ^ n *dvd* x ^ n› **by** (*simp add*: *dvd-power-same*)
with ‹$p \in$ *prime-factors* a› ‹$a * b = x$ ^ n› **have** ‹n *dvd multiplicity* p (x ^ n)›
by (*metis dvd-triv-left gcd-nat.extremum in-prime-factors-iff multiplicity-unit-left multiplicity-zero not-dvd-imp-multiplicity-0 power-0-left prime-elem-multiplicity-power-distrib prime-imp-prime-elem*)
also from ‹$p \in$ *prime-factors* a› ‹*coprime* a b› ‹$a * b = x$ ^ n›
have ‹*multiplicity* p (x ^ n) = *multiplicity* p a›
by (*metis* (*no-types*, *opaque-lifting*) *add.right-neutral coprime-0-right-iff co-*

prime-def
in-prime-factors-iff normalization-semidom-class.prime-def prime-factorization-empty-iff prime-elem-multiplicity-eq-zero-iff prime-elem-multiplicity-mult-distrib)
finally show ‹*n dvd multiplicity p a*› **.**
qed
with *hyp* **show** *thesis* **by** *blast*
qed

2.3 Sophie Germain Prime

Finally, we introduce Sophie Germain primes.

definition *SophGer-prime* :: ‹*nat* ⇒ *bool*› (‹*SG*›)
where ‹*SG p* ≡ *odd p* ∧ *prime p* ∧ *prime* (*2* ∗ *p* + *1*)›

lemma *SophGer-primeI* : ‹*odd p* ⟹ *prime p* ⟹ *prime* (*2* ∗ *p* + *1*) ⟹ *SG p*›
unfolding *SophGer-prime-def* **by** *simp*

lemma *SophGer-primeD* : ‹*odd p*› ‹*prime p*› ‹*prime* (*2* ∗ *p* + *1*)› **if** ‹*SG p*›
using *that* **unfolding** *SophGer-prime-def* **by** *simp-all*

We can easily compute Sophie Germain primes less than *2000*.

value ‹[*p. p* ← [*0..2000*], *SG* (*nat p*)]›

context fixes *p* **assumes** ‹*SG p*› **begin**

local-setup ‹*Local-Theory.map-background-naming* (*Name-Space.mandatory-path SG-simps*)›

lemma *nonzero* : ‹*p* ≠ *0*› **using** ‹*SG p*› **by** (*simp add*: *odd-pos SophGer-primeD*(*1*))

lemma *pos* : ‹*0* < *p*› **using** *nonzero* **by** *blast*

lemma *ge-3* : ‹*3* ≤ *p*›
by (*metis* ‹*SG p*› *SophGer-prime-def gcd-nat.order-iff-strict not-less-eq-eq numeral-2-eq-2 numeral-3-eq-3 order-antisym-conv prime-ge-2-nat*)

lemma *ge-7* : ‹*7* ≤ *2* ∗ *p* + *1*› **using** *ge-3* **by** *auto*

lemma *notcong-zero* :
‹[− *3* ≠ *0* :: *int*] (*mod 2* ∗ *p* + *1*)› ‹[− *1* ≠ *0* :: *int*] (*mod 2* ∗ *p* + *1*)›
‹[*1* ≠ *0* :: *int*] (*mod 2* ∗ *p* + *1*)› ‹[*3* ≠ *0* :: *int*] (*mod 2* ∗ *p* + *1*)›
using *SophGer-primeD*(*2*)[*OF* ‹*SG p*›]
by (*simp-all add*: *cong-def zmod-zminus1-not-zero prime-nat-iff''*)

lemma *notcong-p* :
‹[− *1* ≠ *p* :: *int*] (*mod 2* ∗ *p* + *1*)›

```
  ‹[  0 ≠ p :: int] (mod 2 * p + 1)›
  ‹[  1 ≠ p :: int] (mod 2 * p + 1)›
  using SophGer-primeD(2)[OF ‹SG p›]
  by (auto simp add: pos cong-def zmod-zminus1-eq-if)


lemma p-th-power-mod-q :
  ‹[m ^ p = 1] (mod 2 * p + 1) ∨ [m ^ p = − 1] (mod 2 * p + 1)›
  if ‹¬ 2 * p + 1 dvd m› for m :: int
proof −
  wlog ‹0 < m› generalizing m keeping that
    by (cases ‹0 < − m›)
      (metis (no-types, opaque-lifting) ‹¬ 2 * p + 1 dvd m› add.inverse-inverse
        cong-minus-minus-iff dvd-minus-iff hypothesis uminus-power-if,
        use ‹¬ 2 * p + 1 dvd m› ‹¬ 0 < m› in auto)

  with ‹¬ 2 * p + 1 dvd m› obtain n where ‹m = int n› ‹¬ 2 * p + 1 dvd n›
    by (metis int-dvd-int-iff pos-int-cases)
  from ‹0 < m› have ‹0 < m ^ p› by simp

  have ‹[m ^(2 * p) = n ^(2 * p)] (mod 2 * p + 1)› by (simp add: ‹m = int n›)
  moreover have ‹[n ^(2 * p) = 1] (mod 2 * p + 1)›
     by (metis SophGer-prime-def ‹SG p› ‹¬ 2 * p + 1 dvd n› add-implies-diff
fermat-theorem)
   ultimately have ‹[m ^ (2 * p) = 1] (mod 2 * p + 1)› by (metis cong-def
int-ops(2) zmod-int)
  also have ‹m ^ (2 * p) = m ^ p * m ^ p› by (simp add: mult-2 power-add)
  finally have ‹[m ^ p * m ^ p = 1] (mod 2 * p + 1)› .
  thus ‹[m ^ p = 1] (mod 2 * p + 1) ∨ [m ^ p = − 1] (mod 2 * p + 1)›
   by (meson SophGer-primeD(3) ‹0 < m ^ p› ‹SG p› cong-square prime-nat-int-transfer)
qed



end
```

2.4 Fermat's little Theorem for Integers

```
lemma fermat-theorem-int :
  ‹[a ^(p − 1) = 1] (mod p)› if ‹prime p› and ‹¬ p dvd a›
  for p :: nat and a :: int
proof (cases a)
  show ‹a = int n ⟹ [a ^(p − 1) = 1] (mod p)› for n
    by (metis cong-int-iff fermat-theorem int-dvd-int-iff of-nat-1 of-nat-power that)
next
  fix n assume ‹a = − int (Suc n)›
  from ‹prime p› have ‹p = 2 ∨ odd p›
    by (metis prime-prime-factor two-is-prime-nat)
  thus ‹[a ^(p − 1) = 1] (mod p)›
```

```
  proof (elim disjE)
    assume ‹p = 2›
    with ‹¬ p dvd a› have ‹[a = 1] (mod p)› by (simp add: cong-iff-dvd-diff)
    with ‹p = 2› show ‹[a ^ (p − 1) = 1] (mod p)› by simp
  next
    assume ‹odd p›
    hence ‹even (p − 1)› by simp
    hence ‹a ^ (p − 1) = (int (Suc n)) ^ (p − 1)›
      by (metis ‹a = − int (Suc n)› uminus-power-if)
    also have ‹[. . . = 1] (mod p)›
      by (metis ‹a = − int (Suc n)› cong-int-iff dvd-minus-iff
          fermat-theorem int-dvd-int-iff of-nat-1 of-nat-power that)
    finally show ‹[a ^ (p − 1) = 1] (mod p)› .
  qed
qed
```

3 Sufficient Conditions for FLT

Recall that FLT stands for "Fermat's Last Theorem". FLT states that there is no nontrivial integer solutions to the equation $x^n + y^n = z^n$ for any natural number $2 < n$. as soon as the natural number n is greater than 2. More formally: $2 < n \Longrightarrow \nexists x\ y\ z.\ x^n + y^n = z^n$. We give here some sufficient conditions.

3.1 Coprimality

We first notice that it is sufficient to prove FLT for integers x, y and z that are (setwise) *coprime*.

```
lemma (in semiring-Gcd) FLT-setwise-coprime-reduction :
  assumes ‹x ^ n + y ^ n = z ^ n› ‹x ≠ 0› ‹y ≠ 0› ‹z ≠ 0›
  defines ‹d ≡ Gcd {x, y, z}›
  shows ‹(x div d) ^ n + (y div d) ^ n = (z div d) ^ n› ‹x div d ≠ 0›
    ‹y div d ≠ 0› ‹z div d ≠ 0› ‹Gcd {x div d, y div d, z div d} = 1›
proof −
  have ‹d dvd x› ‹d dvd y› ‹d dvd z› by (unfold d-def, rule Gcd-dvd; simp)+
  thus ‹x div d ≠ 0› ‹y div d ≠ 0› ‹z div d ≠ 0›
    by (simp-all add: ‹x ≠ 0› ‹y ≠ 0› ‹z ≠ 0› dvd-div-eq-0-iff)

  have ‹{x div d, y div d, z div d} = (λn. n div d) ‘ {x, y, z}› by blast
  thus ‹Gcd {x div d, y div d, z div d} = 1›
    by (metis GCD-div-Gcd-is-one ‹x div d ≠ 0› d-def div-by-0)
```

from ‹x ^ n + y ^ n = z ^ n› **show** ‹(x *div* d) ^ n + (y *div* d) ^ n = (z *div* d) ^ n›
  by (*metis* ‹d *dvd* x› ‹d *dvd* y› ‹d *dvd* z› *div-add div-power dvd-power-same*)
qed

corollary (**in** *semiring-Gcd*) *FLT-for-coprime-is-sufficient* :
‹$\nexists x\ y\ z.\ x \neq 0 \land y \neq 0 \land z \neq 0 \land Gcd\ \{x, y, z\} = 1 \land x$ ^ $n + y$ ^ $n = z$ ^ n $\Longrightarrow$
 $\nexists x\ y\ z.\ x \neq 0 \land y \neq 0 \land z \neq 0 \land x$ ^ $n + y$ ^ $n = z$ ^ n›
by (*metis* (*no-types*) *FLT-setwise-coprime-reduction*)

— These very generic lemmas are of course working for integers.
lemma ‹*OFCLASS*(*int*, *semiring-Gcd-class*)› **by** *intro-classes*

This version involving congruences will be useful later.

lemma *FLT-setwise-coprime-reduction-mod-version* :
fixes $x\ y\ z$ :: *int*
assumes ‹x ^ $n + y$ ^ $n = z$ ^ n› ‹$[x \neq 0]$ (*mod* m)› ‹$[y \neq 0]$ (*mod* m)› ‹$[z \neq 0]$ (*mod* m)›
defines ‹$d \equiv Gcd\ \{x, y, z\}$›
shows ‹(x *div* d) ^ n + (y *div* d) ^ n = (z *div* d) ^ n› ‹[x *div* $d \neq 0$] (*mod* m)›
 ‹[y *div* $d \neq 0$] (*mod* m)› ‹[z *div* $d \neq 0$] (*mod* m)› ‹*Gcd* {x *div* d, y *div* d, z *div* d} = *1*›
proof −
have ‹d *dvd* x› ‹d *dvd* y› ‹d *dvd* z› **by** (*unfold d-def*, *rule Gcd-dvd*; *simp*)+
show ‹[x *div* $d \neq 0$] (*mod* m)›
  by (*metis* ‹d *dvd* x› *assms*(*2*) *cong-0-iff dvd-mult dvd-mult-div-cancel*)
show ‹[y *div* $d \neq 0$] (*mod* m)›
  by (*metis* ‹d *dvd* y› *assms*(*3*) *cong-0-iff dvd-mult dvd-mult-div-cancel*)
show ‹[z *div* $d \neq 0$] (*mod* m)›
  by (*metis* ‹d *dvd* z› *assms*(*4*) *cong-0-iff dvd-mult dvd-mult-div-cancel*)

have ‹{x *div* d, y *div* d, z *div* d} = (λn. n *div* d) ‘ {x, y, z}› **by** *blast*
thus ‹*Gcd* {x *div* d, y *div* d, z *div* d} = *1*›
 by (*metis GCD-div-Gcd-is-one* ‹[x *div* $d \neq 0$] (*mod* m)› *cong-refl d-def div-by-0*)

from ‹x ^ n + y ^ n = z ^ n› **show** ‹(x *div* d) ^ n + (y *div* d) ^ n = (z *div* d) ^ n›
  by (*metis* ‹d *dvd* x› ‹d *dvd* y› ‹d *dvd* z› *div-plus-div-distrib-dvd-right div-power dvd-power-same*)
qed

Actually, it is sufficient to prove FLT for integers x, y and z that are pairwise *coprime*

lemma (**in** *semiring-Gcd*) *FLT-setwise-coprime-imp-pairwise-coprime* :
‹*coprime* x y› **if** ‹$n \neq 0$› ‹x ^ $n + y$ ^ $n = z$ ^ n› ‹$Gcd\ \{x, y, z\} = 1$›
proof (*rule ccontr*)
assume ‹¬ *coprime* x y›
with *is-unit-gcd* **obtain** d **where** ‹¬ *is-unit* d› ‹d *dvd* x› ‹d *dvd* y› **by** *blast*

from ‹*d dvd x*› ‹*d dvd y*› **have** ‹*d ^ n dvd x ^ n*› ‹*d ^ n dvd y ^ n*›
  by (*simp-all add*: *dvd-power-same*)
moreover from *calculation* ‹*x ^ n + y ^ n = z ^ n*› **have** ‹*d ^ n dvd z ^ n*›
  by (*metis dvd-add-right-iff*)
moreover from ‹*Gcd* {*x*, *y*, *z*} = *1*› **have** ‹*Gcd* {*x ^ n*, *y ^ n*, *z ^ n*} = *1*›
  by (*simp add*: *gcd-exp-weak*)
ultimately have ‹*is-unit* (*d ^ n*)› **by** (*metis Gcd-2 Gcd-insert gcd-greatest*)
with ‹¬ *is-unit d*› **show** *False* **by** (*metis is-unit-power-iff* ‹*n* ≠ *0*›)
qed

3.2 Odd prime Exponents

From `Fermat3_4`, FLT is already proven for $n = 4$. Using this, we can prove that it is sufficient to prove FLT for *odd prime* exponents.

lemma (**in** *semiring-1-no-zero-divisors*) *FLT-exponent-reduction* :
  assumes ‹*x ^ n + y ^ n = z ^ n*› ‹*x* ≠ *0*› ‹*y* ≠ *0*› ‹*z* ≠ *0*› ‹*p dvd n*›
  shows ‹(*x ^* (*n div p*)) *^ p* + (*y ^* (*n div p*)) *^ p* = (*z ^* (*n div p*)) *^ p*›
    ‹*x ^* (*n div p*) ≠ *0*› ‹*y ^* (*n div p*) ≠ *0*› ‹*z ^* (*n div p*) ≠ *0*›
proof −
  from *power-not-zero*[*OF* ‹*x* ≠ *0*›] **show** ‹*x ^* (*n div p*) ≠ *0*› **.**
  from *power-not-zero*[*OF* ‹*y* ≠ *0*›] **show** ‹*y ^* (*n div p*) ≠ *0*› **.**
  from *power-not-zero*[*OF* ‹*z* ≠ *0*›] **show** ‹*z ^* (*n div p*) ≠ *0*› **.**

  from ‹*p dvd n*› **have** ∗ : ‹*n* = (*n div p*) ∗ *p*› **by** *simp*
  from ‹*x ^ n + y ^ n = z ^ n*›
  show ‹(*x ^* (*n div p*)) *^ p* + (*y ^* (*n div p*)) *^ p* = (*z ^* (*n div p*)) *^ p*›
    by (*subst* (*asm*) (*1 2 3*) ∗) (*simp add*: *power-mult*)
qed

lemma ‹*OFCLASS*(*int*, *semiring-1-no-zero-divisors-class*)› **by** *intro-classes*

lemma *odd-prime-or-four-factorE* :
  fixes *n* :: *nat* **assumes** ‹*2* < *n*›
  obtains *p* **where** ‹*p dvd n*› ‹*odd p*› ‹*prime p*› | ‹*4 dvd n*›
proof −
  assume *hyp1* : ‹*p dvd n* ⟹ *odd p* ⟹ *prime p* ⟹ *thesis*› **for** *p*
  assume *hyp2* : ‹*4 dvd n* ⟹ *thesis*›

  show *thesis*
  proof (*cases* ‹∃ *p*. *p dvd n* ∧ *odd p* ∧ *prime p*›)
    from *hyp1* **show** ‹∃ *p*. *p dvd n* ∧ *odd p* ∧ *prime p* ⟹ *thesis*› **by** *blast*
  next
    assume ‹∄ *p*. *p dvd n* ∧ *odd p* ∧ *prime p*›
    hence ‹*p* ∈ *prime-factors n* ⟹ *p* = *2*› **for** *p*
      by (*metis in-prime-factors-iff primes-dvd-imp-eq two-is-prime-nat*)
    then obtain *k* **where** ‹*prime-factorization n* = *replicate-mset k 2*›
      by (*metis set-mset-subset-singletonD singletonI subsetI*)

hence ‹$n = 2 \,\hat{}\, k$› **by** (*subst prod-mset-prime-factorization-nat*[*symmetric*])
(*use assms* **in** *simp-all*)
with ‹$2 < n$› **have** ‹$1 < k$› **by** (*metis nat-power-less-imp-less pos2 power-one-right*)
with ‹$n = 2 \,\hat{}\, k$› **have** ‹*4 dvd n*›
by (*metis Suc-leI dvd-power-iff-le numeral-Bit0-eq-double*
power.simps(*2*) *power-one-right verit-comp-simplify1*(*2*))
with *hyp2* **show** *thesis* **by** *blast*
qed
qed

Finally, proving FLT for odd prime exponents is sufficient.

corollary *FLT-for-odd-prime-exponents-is-sufficient* :
‹$\nexists x\ y\ z :: int.\ x \neq 0 \wedge y \neq 0 \wedge z \neq 0 \wedge x \,\hat{}\, n + y \,\hat{}\, n = z \,\hat{}\, n$› **if** ‹$2 < n$›
and *odd-prime-FLT* :
‹$\bigwedge p.\ odd\ p \Longrightarrow prime\ p \Longrightarrow$
$\nexists x\ y\ z :: int.\ \ x \neq 0 \wedge y \neq 0 \wedge z \neq 0 \wedge x \,\hat{}\, p + y \,\hat{}\, p = z \,\hat{}\, p$›
proof (*rule ccontr*)
assume ‹$\neg\ (\nexists x\ y\ z :: int.\ x \neq 0 \wedge y \neq 0 \wedge z \neq 0 \wedge x \,\hat{}\, n + y \,\hat{}\, n = z \,\hat{}\, n)$›
then obtain *x y z* :: *int*
where ‹$x \neq 0$› ‹$y \neq 0$› ‹$z \neq 0$› ‹$x \,\hat{}\, n + y \,\hat{}\, n = z \,\hat{}\, n$› **by** *blast*
from *odd-prime-or-four-factorE* ‹$2 < n$›
consider *p* **where** ‹*p dvd n*› ‹*odd p*› ‹*prime p*› | ‹*4 dvd n*› **by** *blast*
thus *False*
proof *cases*
fix *p* **assume** ‹*p dvd n*› ‹*odd p*› ‹*prime p*›
from *FLT-exponent-reduction* [*OF* ‹$x \,\hat{}\, n + y \,\hat{}\, n = z \,\hat{}\, n$› ‹$x \neq 0$› ‹$y \neq 0$›
‹$z \neq 0$› ‹*p dvd n*›]
odd-prime-FLT[*OF* ‹*odd p*› ‹*prime p*›]
show *False* **by** *blast*
next
assume ‹*4 dvd n*›
from *fermat-mult4*[*OF* ‹$x \,\hat{}\, n + y \,\hat{}\, n = z \,\hat{}\, n$› ‹*4 dvd n*›] ‹$x \neq 0$› ‹$y \neq 0$› ‹$z$
$\neq 0$›
show *False* **by** (*metis mult-eq-0-iff*)
qed
qed

4 Sophie Germain's Theorem: classical Version

The proof we give here is from [1].

4.1 A Crucial Lemma

lemma *Sophie-Germain-lemma-computation* :
  fixes x y :: *int* **assumes** ‹*odd p*›
  defines ‹$S \equiv \sum k = 0..p - 1.\ (-\ y)\ \hat{}\ (p - 1 - k) * x\ \hat{}\ k$›
  shows ‹$(x + y) * S = x\ \hat{}\ p + y\ \hat{}\ p$›
proof −
  from ‹*odd p*› **have** ‹$0 < p$› **by** (*simp add*: *odd-pos*)

  from *int-distrib*(*1*) **have** ‹$(x + y) * S = x * S - (-\ y) * S$› **by** *auto*
  have ‹$x * S = (\sum k = 0..p - 1.\ (-\ y)\ \hat{}\ (p - 1 - k) * x\ \hat{}\ (k + 1))$›
    by (*unfold S-def*, *subst sum-distrib-left*) (*rule sum.cong*[*OF refl*], *simp*)
  also have ‹$\ldots = (\sum k = 0..p - 1.\ (-\ y)\ \hat{}\ (p - (k + 1)) * x\ \hat{}\ (k + 1))$› **by**
simp
  also have ‹$\ldots = x\ \hat{}\ p + (\sum k = 1..p - 1.\ (-\ y)\ \hat{}\ (p - k) * x\ \hat{}\ k)$›
    by (*subst sum.shift-bounds-cl-nat-ivl*[*symmetric*])
      (*simp*, *metis One-nat-def* ‹$0 < p$› *not-gr0 power-eq-if*)
  finally have *S1* : ‹$x * S = x\ \hat{}\ p + (\sum k = 1..p - 1.\ (-\ y)\ \hat{}\ (p - k) * x\ \hat{}\ k)$›
.

  have ‹$k \in \{0..p - 1\} \Longrightarrow (-\ y)\ \hat{}\ Suc\ (p - 1 - k) * x\ \hat{}\ k = (-\ y)\ \hat{}\ (p - k)$
$* x\ \hat{}\ k$› **for** k
    by (*rule arg-cong*[**where** f = ‹$\lambda n.\ (-\ y)\ \hat{}\ n * x\ \hat{}$ -›])
      (*metis Suc-diff-le Suc-pred′* ‹$0 < p$› *atLeastAtMost-iff*)
  hence ‹$(-\ y) * S = (\sum k = 0..p - 1.\ (-\ y)\ \hat{}\ (p - k) * x\ \hat{}\ k)$›
    by (*unfold S-def*, *subst sum-distrib-left*, *intro sum.cong*[*OF refl*])
      (*subst mult.assoc*[*symmetric*], *subst power-Suc*[*symmetric*], *simp*)
  also have ‹$\ldots = (-\ y)\ \hat{}\ (p - 0) * x\ \hat{}\ 0 + (\sum k = 1..p - 1.\ (-\ y)\ \hat{}\ (p - k)$
$* x\ \hat{}\ k)$›
    by (*unfold One-nat-def*, *subst sum.atLeast-Suc-atMost*[*symmetric*]) *simp-all*
  also have ‹$(-\ y)\ \hat{}\ (p - 0) * x\ \hat{}\ 0 = -\ (y\ \hat{}\ p)$›
    by (*simp add*: ‹*odd p*›)
  finally have *S2* : ‹$-\ y * S = -\ (y\ \hat{}\ p) + (\sum k = 1..p - 1.\ (-\ y)\ \hat{}\ (p - k) *$
$x\ \hat{}\ k)$› **.**

  show ‹$(x + y) * S = x\ \hat{}\ p + y\ \hat{}\ p$›
    unfolding ‹$(x + y) * S = x * S - (-\ y) * S$› *S1 S2* **by** *simp*
qed

lemma *Sophie-Germain-lemma-computation-cong-simp* :
  fixes p :: *nat* **and** n x y :: *int* **assumes** ‹$p \neq 0$› ‹$[y = -\ x]\ (mod\ n)$›
  defines ‹$S \equiv \lambda x\ y.\ \sum k = 0..p - 1.\ (-\ y)\ \hat{}\ (p - 1 - k) * x\ \hat{}\ k$›
  shows ‹$[S\ x\ y = p * x\ \hat{}\ (p - 1)]\ (mod\ n)$›
proof −
  from ‹$[y = -\ x]\ (mod\ n)$› **have** ‹$[S\ x\ y = S\ x\ (-\ x)]\ (mod\ n)$›
    unfolding *S-def*
    by (*meson cong-minus-minus-iff cong-pow cong-scalar-right cong-sum*)
  also have ‹$S\ x\ (-\ x) = (\sum k = 0..p - 1.\ x\ \hat{}\ (p - 1))$›
    unfolding *S-def*
    by (*rule sum.cong*[*OF refl*], *simp*)

```
    (metis One-nat-def diff-Suc-eq-diff-pred le-add-diff-inverse2 power-add)
  also from ‹p ≠ 0› have ‹... = p ∗ x ^ (p − 1)› by simp
  finally show ‹[S x y = p ∗ x ^ (p − 1)] (mod n)› .
qed

lemma Sophie-Germain-lemma-only-possible-prime-common-divisor :
  fixes x y z :: int and p :: nat
  defines S-def: ‹S ≡ λx y. ∑ k = 0..p − 1. (− y) ^ (p − 1 − k) ∗ x ^ k›
  assumes ‹prime p› ‹prime q› ‹coprime x y› ‹q dvd x + y› ‹q dvd S x y›
  shows ‹q = p›
proof (rule ccontr)
  from ‹prime p› have ‹p ≠ 0› by auto
  assume ‹q ≠ p›
  from ‹q dvd x + y› have ‹[y = − x] (mod q)›
    by (metis add-minus-cancel cong-iff-dvd-diff uminus-add-conv-diff)
  from Sophie-Germain-lemma-computation-cong-simp[OF ‹p ≠ 0› this]
  have ‹[S x y = p ∗ x ^ (p − 1)] (mod q)› unfolding S-def .
  with ‹q dvd S x y› ‹q ≠ p› ‹prime q› ‹prime p› have ‹q dvd x ^ (p − 1)›
   by (metis cong-dvd-iff prime-dvd-mult-iff prime-nat-int-transfer primes-dvd-imp-eq)
   with ‹prime q› prime-dvd-power-int prime-nat-int-transfer have ‹q dvd x› by
blast
 with ‹q dvd x + y› ‹[y = − x] (mod q)› have ‹q dvd y› by (simp add: cong-dvd-iff)
  with ‹coprime x y› ‹q dvd x› ‹prime q› show False
    by (metis coprime-def not-prime-unit)
qed

lemma Sophie-Germain-lemma :
  fixes x y z :: int
  assumes ‹odd p› and ‹prime p› and fermat : ‹x ^ p + y ^ p + z ^ p = 0›
    and ‹[x ≠ 0] (mod p)› and ‹coprime y z›
  defines ‹S ≡ ∑ k = 0..p − 1. (− z) ^ (p − 1 − k) ∗ y ^ k›
  shows ‹∃ a α. y + z = a ^ p ∧ S = α ^ p›
proof −
  from Sophie-Germain-lemma-computation[OF ‹odd p›]
  have ‹(y + z) ∗ S = y ^ p + z ^ p› unfolding S-def .
  also from fermat have ‹... = (− x) ^ p› by (simp add: ‹odd p›)
  finally have ‹(y + z) ∗ S = ...› .

  have ‹coprime (y + z) S›
  proof (rule ccontr)
    assume ‹¬ coprime (y + z) S›
    then consider ‹y + z = 0› | ‹S = 0› | q :: nat where ‹prime q› ‹q dvd y +
z› ‹q dvd S›
      by (elim not-coprime-nonzeroE)
        (use ‹(y + z) ∗ S = (− x) ^ p› ‹[x ≠ 0] (mod p)› in force,
          metis nat-0-le prime-int-nat-transfer)
    hence ‹p dvd (y + z) ∗ S›
    proof cases
      fix q :: nat assume ‹prime q› ‹q dvd y + z› ‹q dvd S›
```

from *Sophie-Germain-lemma-only-possible-prime-common-divisor*
[OF ‹prime p› - ‹coprime y z› ‹q dvd y + z› ‹q dvd S›[unfolded S-def]]
show *‹p dvd (y + z) ∗ S›* **using** *‹int q dvd S› ‹prime q›* **by** *auto*
qed *simp-all*
with *‹(y + z) ∗ S = (− x) ^ p› ‹[x ≠ 0] (mod p)›* **show** *False*
by *(metis ‹prime p› cong-0-iff dvd-minus-iff prime-dvd-power-int prime-nat-int-transfer)*
qed

from *prod-is-some-powerE[OF coprime-commute[THEN iffD1, OF ‹coprime (y + z) S›]]*
obtain *α* **where** *‹normalize S = α ^ p›*
by *(metis (no-types, lifting) ‹(y + z) ∗ S = (− x) ^ p› mult.commute)*
moreover from *prod-is-some-powerE[OF ‹coprime (y + z) S› ‹(y + z) ∗ S = (− x) ^ p›]*
obtain *a* **where** *‹normalize (y + z) = a ^ p›* **by** *blast*
ultimately have *‹S = (if 0 ≤ S then α ^ p else (− α) ^ p)›*
‹y + z = (if 0 ≤ y + z then a ^ p else (− a) ^ p)›
by *(metis ‹odd p› abs-of-nonneg abs-of-nonpos*
add.inverse-inverse linorder-linear normalize-int-def power-minus-odd)+
thus *‹∃ a α. y + z = a ^ p ∧ S = α ^ p›* **by** *meson*
qed

4.2 The Theorem

theorem *Sophie-Germain-theorem* :
‹∄ x y z :: int. x ^ p + y ^ p = z ^ p ∧ [x ≠ 0] (mod p) ∧
[y ≠ 0] (mod p) ∧ [z ≠ 0] (mod p)› **if** *SG* : *‹SG p›*
proof *(rule ccontr)* — The proof is done by contradiction.
from *SophGer-primeD(1)[OF ‹SG p›]* **have** *odd-p* : *‹odd p›* **.**
from *SG-simps.pos[OF ‹SG p›]* **have** *pos-p* : *‹0 < p›* **.**

assume *‹¬ (∄ x y z. x ^ p + y ^ p = z ^ p ∧ [x ≠ 0] (mod int p) ∧*
[y ≠ 0] (mod int p) ∧ [z ≠ 0] (mod int p))›
then obtain *x y z :: int*
where *fermat* : *‹x ^ p + y ^ p = z ^ p›*
and *not-cong-0* : *‹[x ≠ 0] (mod p)› ‹[y ≠ 0] (mod p)› ‹[z ≠ 0] (mod p)›* **by** *blast*

— We first assume w.l.o.g. that *x*, *y* and *z* are setwise *coprime*.
let *?gcd = ‹Gcd {x, y, z}›*
wlog *coprime* : *‹?gcd = 1›* **goal** *False* **generalizing** *x y z* **keeping** *fermat not-cong-0*
using *FLT-setwise-coprime-reduction-mod-version[OF fermat not-cong-0]*
hypothesis **by** *blast*

— Then we can deduce that *x*, *y* and *z* are pairwise *coprime*.
have *coprime-x-y* : *‹coprime x y›*
by *(fact FLT-setwise-coprime-imp-pairwise-coprime*
[OF SG-simps.nonzero[OF ‹SG p›] fermat coprime])

have *coprime-y-z* : ‹*coprime y z*›
proof (*subst coprime-minus-right-iff*[*symmetric*],
rule FLT-setwise-coprime-imp-pairwise-coprime[*OF SG-simps.nonzero*[*OF* ‹*SG p*›]]])
from *fermat* ‹*odd p*› **show** ‹$y \ \hat{}\ p + (- z) \ \hat{}\ p = (- x) \ \hat{}\ p$› **by** *simp*
next
show ‹*Gcd* $\{y, - z, - x\} = 1$›
by (*metis Gcd-insert coprime gcd-neg1-int insert-commute*)
qed
have *coprime-x-z* : ‹*coprime x z*›
proof (*subst coprime-minus-right-iff*[*symmetric*],
rule FLT-setwise-coprime-imp-pairwise-coprime[*OF SG-simps.nonzero*[*OF* ‹*SG p*›]]])
from *fermat* ‹*odd p*› **show** ‹$x \ \hat{}\ p + (- z) \ \hat{}\ p = (- y) \ \hat{}\ p$› **by** *simp*
next
show ‹*Gcd* $\{x, - z, - y\} = 1$›
by (*metis Gcd-insert coprime gcd-neg1-int insert-commute*)
qed

let *?q* = ‹$2 * p + 1$›
— From ⟦*SG p*; ¬ *int* $(2 * p + 1)$ *dvd m*⟧ ⟹ $[m^p = 1]$ (*mod int* $(2 * p + 1)$) ∨ $[m^p = - 1]$ (*mod int* $(2 * p + 1)$) we have that among x, y and z, one (and only one, see below) is a multiple of $2 * p + 1$.
have *q-dvd-xyz* : ‹*?q dvd x* ∨ *?q dvd y* ∨ *?q dvd z*›
proof (*rule ccontr*)
have *cong-add-here* : ‹$[x \ \hat{}\ p = n1]$ (*mod ?q*) ⟹ $[y \ \hat{}\ p = n2]$ (*mod ?q*) ⟹ $[z \ \hat{}\ p = n3]$ (*mod ?q*) ⟹
$[x \ \hat{}\ p + y \ \hat{}\ p + (- z) \ \hat{}\ p = n1 + n2 - n3]$ (*mod ?q*)› **for** *n1 n2 n3*
by (*simp add*: *cong-add cong-diff* ‹*odd p*›)
assume ‹¬ (*?q dvd x* ∨ *?q dvd y* ∨ *?q dvd z*)›
hence ‹¬ *?q dvd x*› ‹¬ *?q dvd y*› ‹¬ *?q dvd z*› **by** *simp-all*
from *this*[*THEN SG-simps.p-th-power-mod-q*[*OF* ‹*SG p*›]] *cong-add-here* ‹*odd p*›
have ‹ $[x \ \hat{}\ p + y \ \hat{}\ p + (- z) \ \hat{}\ p = - 3]$ (*mod ?q*) ∨ $[x \ \hat{}\ p + y \ \hat{}\ p + (- z) \ \hat{}\ p = - 1]$ (*mod ?q*)
∨ $[x \ \hat{}\ p + y \ \hat{}\ p + (- z) \ \hat{}\ p = 1]$ (*mod ?q*) ∨ $[x \ \hat{}\ p + y \ \hat{}\ p + (- z) \ \hat{}\ p = 3]$ (*mod ?q*)› (**is** *?disj-congs*)
by (*elim disjE*) *fastforce+*
moreover from *fermat* ‹*odd p*› **have** ‹$[x \ \hat{}\ p + y \ \hat{}\ p + (- z) \ \hat{}\ p = 0]$ (*mod ?q*)› **by** *simp*
ultimately show *False* **by** (*metis cong-def SG-simps.notcong-zero*[*OF* ‹*SG p*›])
qed

— Without loss of generality, we can assume that x is a multiple of $2 * p + 1$.
wlog ‹*?q dvd x*› **goal** *False* **generalizing** *x y z*
keeping *fermat not-cong-0 coprime-x-y coprime-y-z coprime-x-z q-dvd-xyz*
proof −
from *negation q-dvd-xyz* **have** ‹*?q dvd y* ∨ *?q dvd z*› **by** *simp*

```
    thus False
    proof (elim disjE)
      assume ‹?q dvd y›
      thus False
      proof (rule hypothesis[OF - - not-cong-0(2, 1, 3)])
        from fermat show ‹y ^ p + x ^ p = z ^ p› by linarith
      next
        show ‹coprime y x› ‹coprime x z› ‹coprime y z›
          by (simp-all add: coprime-commute coprime-x-y coprime-x-z coprime-y-z)
      next
        from q-dvd-xyz show ‹?q dvd y ∨ ?q dvd x ∨ ?q dvd z› by linarith
      qed
    next
      assume ‹?q dvd z›
      hence ‹?q dvd − z› by simp
      thus False
      proof (rule hypothesis)
        from fermat ‹odd p› show ‹(− z) ^ p + x ^ p = (− y) ^ p› by simp
      next
        from ‹[x ≠ 0] (mod p)› ‹[y ≠ 0] (mod p)› ‹[z ≠ 0] (mod p)›
        show ‹[x ≠ 0] (mod p)› ‹[− y ≠ 0] (mod p)› ‹[− z ≠ 0] (mod p)›
          by (simp-all add: cong-0-iff)
      next
        show ‹coprime (− z) x› ‹coprime x (− y)› ‹coprime (− z) (− y)›
          by (simp-all add: coprime-commute coprime-x-y coprime-x-z coprime-y-z)
      next
        from q-dvd-xyz show ‹?q dvd − z ∨ ?q dvd x ∨ ?q dvd − y› by auto
      qed
    qed
  qed
```

— Now we can use the lemma above.

```
  let ?S = ‹λy z. ∑k = 0..p − 1. (− z) ^ (p − 1 − k) ∗ y ^ k›
  from fermat ‹odd p› have ‹y ^ p + x ^ p + (− z) ^ p = 0›
    ‹x ^ p + y ^ p + (− z) ^ p = 0› ‹(−z ) ^ p + x ^ p + y ^ p = 0› by simp-all
  from Sophie-Germain-lemma[OF SophGer-primeD(1−2)[OF ‹SG p›]
      ‹x ^ p + y ^ p + (− z) ^ p = 0› ‹[x ≠ 0] (mod p)›]
  obtain a α where a-prop : ‹y + (− z) = a ^ p›
    and α-prop : ‹?S y (−z) = α ^ p›
    using coprime-minus-right-iff coprime-y-z by blast

  from Sophie-Germain-lemma[OF SophGer-primeD(1−2)[OF ‹SG p›]
      ‹y ^ p + x ^ p + (− z) ^ p = 0› ‹[y ≠ 0] (mod p)›]
  obtain b where b-prop : ‹x + − z = b ^ p›
    by (metis coprime-minus-right-iff coprime-x-z)

  from Sophie-Germain-lemma[OF SophGer-primeD(1−2)[OF ‹SG p›]
      ‹(−z ) ^ p + x ^ p + y ^ p = 0›] coprime-x-z ‹[z ≠ 0] (mod p)›
  obtain c where c-prop : ‹x + y = c ^ p›
```

by (*meson cong-0-iff coprime-x-y dvd-minus-iff*)

from ‹*?q dvd x*› **have** ‹¬ *?q dvd y*› **and** ‹¬ *?q dvd z*›
using *coprime-x-y coprime-x-z not-coprimeI not-prime-unit prime-nat-int-transfer*
by (*metis SophGer-primeD*(*3*)[*OF* ‹*SG p*›] *prime-nat-int-transfer*)+

from *b-prop* ‹*?q dvd x*› **have** ‹[b ^ p = − z] (*mod ?q*)›
by (*metis add-diff-cancel-right′ cong-iff-dvd-diff*)
with ‹¬ *?q dvd z*› *cong-dvd-iff dvd-minus-iff* **have** ‹¬ *?q dvd b* ^ *p*› **by** *blast*
with ‹*0* < *p*› **have** ‹¬ *?q dvd b*› **by** (*meson dvd-power dvd-trans*)
with *SG-simps.p-th-power-mod-q*[*OF* ‹*SG p*›]
have *cong1* : ‹[b ^ p = *1*] (*mod ?q*) ∨ [b ^ p = − *1*] (*mod ?q*)› **by** *blast*

from *c-prop* ‹*?q dvd x*› **have** ‹[c ^ p = y] (*mod ?q*)›
by (*metis add-diff-cancel-right′ cong-iff-dvd-diff*)
with ‹¬ *?q dvd y*› *cong-dvd-iff* **have** ‹¬ *?q dvd c* ^ *p*› **by** *blast*
with ‹*0* < *p*› **have** ‹¬ *?q dvd c*› **by** (*meson dvd-power dvd-trans*)
with *SG-simps.p-th-power-mod-q*[*OF* ‹*SG p*›]
have *cong2* : ‹[c ^ p = *1*] (*mod ?q*) ∨ [c ^ p = − *1*] (*mod ?q*)› **by** *blast*

have ‹*?q dvd a*›
proof (*rule ccontr*)
have *cong-add-here* : ‹[b ^ p = *n1*] (*mod ?q*) ⟹ [c ^ p = *n2*] (*mod ?q*) ⟹ [a ^ p = *n3*] (*mod ?q*) ⟹
[b ^ p + c ^ p − a ^ p = *n1* + *n2* − *n3*] (*mod ?q*)› **for** *n1 n2 n3*
by (*intro cong-add cong-diff*)
assume ‹¬ *?q dvd a*›
with *SG-simps.p-th-power-mod-q*[*OF* ‹*SG p*›]
have *cong3* : ‹[a ^ p = *1*] (*mod ?q*) ∨ [a ^ p = − *1*] (*mod ?q*)› **by** *blast*
from *cong1 cong2 cong3 cong-add-here*
have ‹ [b ^ p + c ^ p − a ^ p = − *3*] (*mod ?q*) ∨ [b ^ p + c ^ p − a ^ p = − *1*] (*mod ?q*)
∨ [b ^ p + c ^ p − a ^ p = *1*] (*mod ?q*) ∨ [b ^ p + c ^ p − a ^ p = *3*] (*mod ?q*)› (**is** *?disj-congs*)
by (*elim disjE*) *fastforce*+
have ‹b ^ p + c ^ p − a ^ p = *2* ∗ x› **by** (*fold a-prop b-prop c-prop*) *simp*
also from ‹*?q dvd x*› *cong-0-iff* **have** ‹[*2* ∗ x = *0*] (*mod ?q*)›
by (*metis dvd-add-right-iff mult-2*)
finally have ‹[b ^ p + c ^ p − a ^ p = *0*] (*mod ?q*)› **.**
with ‹*?disj-congs*› **show** *False* **by** (*metis cong-def SG-simps.notcong-zero*[*OF* ‹*SG p*›])
qed
with *oddE* ‹*odd p*› **have** ‹*?q dvd a* ^ *p*› **by** *fastforce*
with *a-prop* **have** ‹[y = z] (*mod ?q*)› **by** (*simp add*: *cong-iff-dvd-diff*)
with *cong-sym* **have** ‹[z = y] (*mod ?q*)› **by** *blast*

— It is now time to conclude the proof!
have ‹α ^ *p* = *?S y* (−*z*)› **by** (*fact* α*-prop*[*symmetric*])
also from *SG-simps.nonzero*[*OF* ‹*SG p*›] ‹[*z* = *y*] (*mod ?q*)› *cong-minus-minus-iff*
have ‹[*?S y* (−*z*) = *p* ∗ *y* ^ (*p* − *1*)] (*mod ?q*)›
by (*blast intro*: *Sophie-Germain-lemma-computation-cong-simp*)
finally have ‹[α ^ *p* = *p* ∗ *y* ^ (*p* − *1*)] (*mod ?q*)› **.**

from *SG-simps.p-th-power-mod-q*[*OF* ‹*SG p*› ‹¬ *?q dvd c* ^ *p*›] ‹[*c* ^ *p* = *y*] (*mod ?q*)›
have ‹[*y* = *1*] (*mod ?q*) ∨ [*y* = − *1*] (*mod ?q*)› **by** (*metis cong2 cong-def*)
hence ‹[*y* ^ (*p* − *1*) = *1*] (*mod ?q*)›
by (*elim disjE*) (*drule cong-pow*[**where** *n* = ‹*p* − *1*›], *simp add*: ‹*odd p*›)+
from *cong-trans*[*OF* ‹[α ^ *p* = *p* ∗ *y* ^ (*p* − *1*)] (*mod ?q*)› *cong-mult*[*OF cong-refl this*]]
have ‹[α ^ *p* = *p*] (*mod ?q*)› **by** *simp*

— But α^p is congruent to − *1*, *0* or *1* modulo *2* ∗ *p* + *1*, whereas *p* can not be; hence the final contradiction.
moreover from *SG-simps.p-th-power-mod-q*[*OF* ‹*SG p*›]
have ‹[α ^ *p* = − *1*] (*mod ?q*) ∨ [α ^ *p* = *0*] (*mod ?q*) ∨ [α ^ *p* = *1*] (*mod ?q*)›
by (*metis cong-0-iff cong-pow power-0-left*)
ultimately show *False* **by** (*metis SG-simps.notcong-p*[*OF* ‹*SG p*›] *cong-def*)
qed

5 Sophie Germain's Theorem: generalized Version

The proof we give here is from [2].

5.1 Auxiliary Primes

abbreviation *non-consecutivity-condition* :: ‹*nat* ⇒ *nat* ⇒ *bool*› (‹*NC*›)
where ‹*NC p q* ≡ ∄ *x y* :: *int*. [*x* ≠ *0*] (*mod q*) ∧ [*y* ≠ *0*] (*mod q*) ∧ [*x* ^ *p* = *1* + *y* ^ *p*] (*mod q*)›

lemma *non-consecutivity-condition-bis* :
‹*NC p q* ⟷ (∄ *x y a b*. [*a* :: *int* ≠ *0*] (*mod q*) ∧ [*a* ^ *p* = *x*] (*mod q*) ∧
[*b* :: *int* ≠ *0*] (*mod q*) ∧ [*b* ^ *p* = *y*] (*mod q*) ∧ [*x* = *1* + *y*] (*mod q*))›
by (*simp add*: *cong-def*) (*metis mod-add-right-eq*)

abbreviation *not-pth-power* :: ‹*nat* ⇒ *nat* ⇒ *bool*› (‹*PPP*›)
  where ‹*PPP p q* ≡ ∄*x* :: *int*. [*p* = *x* ^ *p*] (*mod q*)›

definition *auxiliary-prime* :: ‹*nat* ⇒ *nat* ⇒ *bool*› (‹*aux′-prime*›)
  where ‹*aux-prime p q* ≡ *prime p* ∧ *prime q* ∧ *NC p q* ∧ *PPP p q*›

lemma *auxiliary-primeI* :
  ‹⟦*prime p*; *prime q*; *NC p q*; *PPP p q*⟧ ⟹ *aux-prime p q*›
  unfolding *auxiliary-prime-def* **by** *auto*

lemma *auxiliary-primeD* :
  ‹*prime p*› ‹*prime q*› ‹*NC p q*› ‹*PPP p q*› **if** ‹*aux-prime p q*›
  using *that* **by** (*auto simp add*: *auxiliary-prime-def*)

We do not give code equation yet, let us first work around these notions.

lemma *gen-mult-group-mod-prime-as-ord* : ‹*ord p g* = *p* − *1*›
  if ‹*prime p*› ‹{*1* .. *p* − *1*} = {*g* ^ *k mod p* |*k*. *k* ∈ *UNIV*}›
proof −
  from *that*(*2*) **have** ‹*g mod p* ∈ {*1* .. *p* − *1*}›
    by (*simp add*: *set-eq-iff*) (*metis power-one-right*)
  hence ‹[*g* ≠ *0*] (*mod p*)› **by** (*simp add*: *cong-def*)

  with *cong-0-iff prime-imp-coprime* ‹*prime p*›
  have ‹*ord p g* = (*LEAST d*. *0* < *d* ∧ [*g* ^ *d* = *1*] (*mod p*))›
    unfolding *ord-def* **by** *auto*
  also have ‹. . . = *p* − *1*›
  proof (*rule ccontr*)
    assume ‹(*LEAST d*. *0* < *d* ∧ [*g* ^ *d* = *1*] (*mod p*)) ≠ *p* − *1*›
    with *fermat-theorem* ‹*prime p*› ‹[*g* ≠ *0*] (*mod p*)›
    obtain *k* **where** ‹*0* < *k*› ‹*k* < *p* − *1*› ‹[*g* ^ *k* = *1*] (*mod p*)›
      by (*metis calculation cong-0-iff coprime-ord linorder-neqE-nat*
          lucas-coprime-lemma ord-minimal prime-gt-1-nat zero-less-diff)
    { fix *l m*
      have ‹*g* ^ (*m* + (*l* ∗ *k*)) *mod p* = (*g* ^ *m mod p* ∗ ((*g* ^ *k*) ^ *l mod p*)) *mod p*›
        by (*simp add*: *mod-mult-eq mult.commute power-add power-mult*)
      also from ‹[*g* ^ *k* = *1*] (*mod p*)› **have** ‹((*g* ^ *k*) ^ *l mod p*) = *1*›
        by (*metis cong-def cong-pow mod-if power-one prime-nat-iff* ‹*prime p*›)
      finally have ‹*g* ^ (*m* + (*l* ∗ *k*)) *mod p* = *g* ^ *m mod p*› **by** *simp*
    } note $ = *this*

    have ‹*UNIV* = (⋃*l*. {*m* + (*l* ∗ *k*) |*m*. *m* ∈ {*0*..*k* − *1*}})›
     by *auto* (*metis Suc-pred* ‹*0* < *k*› *add.commute div-mod-decomp mod-Suc-le-divisor*)
    hence ‹{*g* ^ *k mod p* |*k*. *k* ∈ *UNIV*} =
            (⋃*l*. {*g* ^ (*m* + (*l* ∗ *k*)) *mod p* |*m*. *m* ∈ {*0*..*k* − *1*}})›
      by (*simp add*: *set-eq-iff*) *metis*
    also have ‹. . . = {*g* ^ *m mod p* |*m*. *m* ∈ {*0*..*k* − *1*}}› **by** (*simp add*: $)

finally have ‹$card\ \{g \,\widehat{}\, k\ mod\ p\ |k.\ k \in UNIV\} = card\ \ldots$› **by** *simp*
also have ‹$\{g \,\widehat{}\, m\ mod\ p\ |m.\ m \in \{0..k - 1\}\} =$
$(\lambda m.\ g \,\widehat{}\, m\ mod\ p)$ ‘ $\{0..k - 1\}$› **by** *auto*
also from *card-image-le* **have** ‹$card \ldots \le card\ \{0..k - 1\}$› **by** *blast*
also have ‹$\ldots = k$› **by** (*simp add*: ‹$0 < k$›)
finally show *False*
by (*metis that*(*2*) ‹$k < p - 1$› *card-atLeastAtMost diff-Suc-1 linorder-not-less*)
qed
finally show ‹$ord\ p\ g = p - 1$› **.**
qed

lemma *exists-nth-power-mod-prime-iff* :
fixes *p n* **assumes** ‹*prime p*›
defines *d-def* : ‹$d \equiv gcd\ n\ (p - 1)$›
shows ‹$(\exists x :: int.\ [a = x \,\widehat{}\, n]\ (mod\ p)) \longleftrightarrow$
$(n \ne 0 \wedge [a = 0]\ (mod\ p)) \vee [a \,\widehat{}\, ((p - 1)\ div\ d) = 1]\ (mod\ p)$›
proof (*cases* ‹$n = 0$›)
show ‹$n = 0 \Longrightarrow$ *?thesis*›
by (*simp add*: *d-def*)
(*metis* ‹*prime p*› *Suc-0-not-prime-nat Suc-pred div-self power-one-right prime-gt-0-nat*)
next
show *?thesis* **if** ‹$n \ne 0$›
proof (*cases* ‹$[a = 0]\ (mod\ p)$›)
show ‹$[a = 0]\ (mod\ p) \Longrightarrow$ *?thesis*›
by (*auto simp add*: *cong-def power-0-left* ‹$n \ne 0$› *intro*!: *exI*[*of - 0*])
next
have ‹$0 < int\ p$› **by** (*simp add*: *prime-gt-0-nat* ‹*prime p*›)
from ‹*prime p*› *residue-prime-mult-group-has-gen gen-mult-group-mod-prime-as-ord*
obtain *g* **where** $*$: ‹$\{1 .. p - 1\} = \{g \,\widehat{}\, k\ mod\ p\ |k.\ k \in UNIV\}$› **and** ‹$ord$
$p\ g = p - 1$› **by** *blast*
have ‹$[g \ne 0]\ (mod\ p)$›
by (*metis* ‹$ord\ p\ g = p - 1$› ‹*prime p*› *nat-less-le ord-0-right-nat*
ord-cong prime-nat-iff zero-less-diff)

show ‹*?thesis*› **if** ‹$[a \ne 0]\ (mod\ p)$›
proof (*rule iffI*)
assume ‹$\exists x.\ [a = x \,\widehat{}\, n]\ (mod\ p)$›
then obtain *x* **where** ‹$[a = x \,\widehat{}\, n]\ (mod\ p)$› **..**
from *cong-less-unique-int*[*OF* ‹$0 < int\ p$›, *of x*]
obtain *y* :: *nat* **where** ‹$0 \le y$› ‹$y < p$› ‹$[x = y]\ (mod\ p)$›
by (*metis int-nat-eq less-eq-nat.simps*(*1*) *nat-less-iff*)
from ‹$[a \ne 0]\ (mod\ p)$› ‹$[a = x \,\widehat{}\, n]\ (mod\ p)$› **have** ‹$[x \ne 0]\ (mod\ p)$›
by (*metis cong-pow cong-sym cong-trans power-0-left* ‹$n \ne 0$›)
with ‹$[x = y]\ (mod\ p)$› **have** ‹$y \ne 0$› **by** (*metis of-nat-0*)
with ‹$0 \le y$› ‹$y < p$› **have** ‹$y \in \{1 .. p - 1\}$› **by** *simp*
with $*$ ‹$[x = y]\ (mod\ p)$› *zmod-int* **obtain** *k* **where** ‹$[x = g \,\widehat{}\, k]\ (mod\ p)$› **by**
auto

with ‹[a = x ^ n] (mod p)› **have** ‹[a = g ^ (k * n)] (mod p)›
by (*metis* (*no-types*, *lifting*) *cong-pow cong-trans of-nat-power power-mult*)
hence ‹[a ^ ((p − 1) div d) = (g ^ (k * n)) ^ ((p − 1) div d)] (mod p)›
by (*simp add*: *cong-pow*)
moreover have ‹[(g ^ (k * n)) ^ ((p − 1) div d) = g ^ (k * n * (p − 1) div
d)] (mod p)›
by (*metis* (*no-types*) *d-def cong-refl div-mult-swap gcd-dvd2 power-mult*)
moreover have ‹[g ^ (k * n * (p − 1) div d) = (g ^ (k * n div d)) ^ (p −
1)] (mod p)›
by (*metis* (*no-types*) *d-def cong-def dvd-div-mult dvd-mult gcd-dvd1 power-mult*)
moreover have ‹[(g ^ (k * n div d)) ^ (p − 1) = 1] (mod p)›
by (*rule fermat-theorem*[*OF* ‹*prime p*›])
(*metis* ‹[g ≠ 0] (mod p)› *cong-0-iff prime-dvd-power-nat* ‹*prime p*›)
ultimately have ‹[a ^ ((p − 1) div d) = 1] (mod p)›
by (*metis* (*no-types*, *opaque-lifting*) *cong-def cong-int-iff of-nat-1*)
thus ‹ n ≠ 0 ∧ [a = 0] (mod p) ∨ [a ^ ((p − 1) div d) = 1] (mod p)› **..**
next
assume ‹ n ≠ 0 ∧ [a = 0] (mod p) ∨ [a ^ ((p − 1) div d) = 1] (mod p)›
with ‹[a ≠ 0] (mod p)› **have** ‹[a ^ ((p − 1) div d) = 1] (mod p)› **by** *blast*

from *cong-less-unique-int*[*OF* ‹0 < int p›, *of a*]
obtain *b* :: *nat* **where** ‹0 ≤ b› ‹b < p› ‹[a = b] (mod p)›
by (*metis int-nat-eq less-eq-nat.simps*(1) *nat-less-iff*)
from ‹[a ≠ 0] (mod p)› ‹[a = b] (mod p)› **have** ‹b ≠ 0› **by** (*metis of-nat-0*)
with ‹0 ≤ b› ‹b < p› **have** ‹b ∈ {1 .. p − 1}› **by** *simp*
with * **have** ‹b ∈ {g ^ k mod p |k. k ∈ UNIV}› **by** *blast*
with ‹[a = b] (mod p)› *zmod-int* **obtain** *k* **where** ‹[a = g ^ k] (mod p)› **by**
auto
from *this*[*THEN cong-pow*, *of* ‹(p − 1) div d›] ‹[a ^ ((p − 1) div d) = 1]
(mod p)›
have ‹[(g ^ k) ^ ((p − 1) div d) = 1] (mod p)›
by (*simp flip*: *cong-int-iff*) (*metis* (*no-types*) *cong-def*)
hence ‹[g ^ (k * (p − 1) div d) = 1] (mod p)›
by (*metis* (*no-types*) *d-def div-mult-swap gcd-dvd2 power-mult*)
hence ‹p − 1 dvd k * (p − 1) div d›
by (*simp add*: *ord-divides′* ‹ord p g = p − 1›)
hence ‹d dvd k›
by (*metis* ‹*prime p*› *d-def div-mult-swap dvd-div-eq-0-iff dvd-mult-div-cancel*
dvd-times-right-cancel-iff gcd-dvd2 less-numeral-extra(3) *prime-gt-1-nat*
zero-less-diff)
then obtain *l* **where** ‹k = l * d› **by** (*metis dvd-div-mult-self*)
moreover from *bezout-nat*[*OF* ‹n ≠ 0›]
obtain *u v* **where** ‹u * n = v * (p − 1) + d› **by** (*metis d-def mult.commute*)
ultimately have ‹l * u * n = l * v * (p − 1) + k›
by (*metis distrib-left mult.assoc*)
hence ‹(g ^ (l * u)) ^ n = (g ^ (l * v)) ^ (p − 1) * g ^ k›
by (*metis power-add power-mult*)
hence ‹[(g ^ (l * u)) ^ n = (g ^ (l * v)) ^ (p − 1) * g ^ k] (mod p)› **by** *simp*
moreover have ‹[(g ^ (l * v)) ^ (p − 1) = 1] (mod p)›

```
        by (metis ‹ord p g = p − 1› dvd-triv-right ord-divides power-mult)
      ultimately have ‹[(g ^ (l ∗ u)) ^ n = g ^ k] (mod p)›
        by (metis cong-scalar-right cong-trans mult-1)
      with ‹[a = g ^ k] (mod p)› have ‹[a = (g ^ (l ∗ u)) ^ n] (mod p)›
        by (meson cong-int-iff cong-sym cong-trans)
      thus ‹∃ x. [a = x ^ n] (mod p)› by auto
    qed
  qed
qed


corollary not-pth-power-iff :
  ‹PPP p q ⟷ [p ≠ 0] (mod q) ∧ [p ^ ((q − 1) div gcd p (q − 1)) ≠ 1] (mod q)›
  if ‹prime p› ‹prime q›
  by (subst exists-nth-power-mod-prime-iff[OF ‹prime q›, of p p])
    (metis cong-int-iff not-prime-0 of-nat-0 of-nat-1 of-nat-power ‹prime p›)

corollary not-pth-power-iff-mod :
  ‹PPP p q ⟷ ¬ q dvd p ∧ p ^ ((q − 1) div gcd p (q − 1)) mod q ≠ 1›
  if ‹prime p› and ‹prime q›
  by (subst not-pth-power-iff[OF ‹prime p› ‹prime q›])
    (simp add: cong-def mod-eq-0-iff-dvd prime-gt-Suc-0-nat)


lemma non-consecutivity-condition-iff-enum-mod :
  — This version is oriented towards code generation.
  ‹NC p q ⟷
   (∀ x ∈ {1..q − 1}. let x-p-mod = x ^ p mod q
                      in ∀ y ∈ {1..q − 1}. x-p-mod ≠ (1 + y ^ p mod q) mod q)›
  if ‹q ≠ 0›
proof (unfold Let-def, intro iffI conjI ballI)
  fix x y assume ‹NC p q› ‹x ∈ {1..q − 1}› ‹ y ∈ {1..q − 1}›
  show ‹x ^ p mod q ≠ (1 + y ^ p mod q) mod q›
  proof (rule ccontr)
    assume ‹¬ x ^ p mod q ≠ (1 + y ^ p mod q) mod q›
    hence ‹[x ^ p = 1 + y ^ p] (mod q)›
      by (simp add: cong-def) presburger
    with ‹NC p q› have ‹[x = 0] (mod q) ∨ [y = 0] (mod q)›
      by (metis (mono-tags, opaque-lifting) cong-int-iff int-ops(1)
          of-nat-Suc of-nat-power-eq-of-nat-cancel-iff plus-1-eq-Suc)
    with cong-less-modulus-unique-nat
    have ‹x ∉ {1..q − 1} ∨ y ∉ {1..q − 1}› by force
    with ‹x ∈ {1..q − 1}› ‹ y ∈ {1..q − 1}› show False by blast
  qed
next
  show ‹NC p q› if ∗ : ‹∀ x∈{1..q − 1}. ∀ y∈{1..q − 1}. x ^ p mod q ≠ (1 + y
^ p mod q) mod q›
  proof (rule ccontr)
    assume ‹¬ NC p q›
```

```
    then obtain x y :: int where ‹[x ≠ 0] (mod q)› ‹[y ≠ 0] (mod q)›
      ‹[x ^ p = 1 + y ^ p] (mod q)› by blast

    from ‹[x ≠ 0] (mod q)› ‹q ≠ 0› have ‹x mod q ∈ {1..q − 1}›
      by (simp add: cong-0-iff)
        (metis linorder-not-le mod-by-1 mod-eq-0-iff-dvd
          mod-pos-pos-trivial of-nat-0-less-iff pos-mod-sign)
    then obtain x' :: nat where ‹x' ∈ {1..q − 1}› ‹x' = x mod q›
      by (cases ‹x mod q›) simp-all

    from ‹[y ≠ 0] (mod q)› ‹q ≠ 0› have ‹y mod q ∈ {1..q − 1}›
      by (simp add: cong-0-iff)
        (metis linorder-not-le mod-by-1 mod-eq-0-iff-dvd
          mod-pos-pos-trivial of-nat-0-less-iff pos-mod-sign)
    then obtain y' :: nat where ‹y' ∈ {1..q − 1}› ‹y' = y mod q›
      by (cases ‹y mod q›) simp-all

    from ‹[x ^ p = 1 + y ^ p] (mod q)›
    have ‹(x mod q) ^ p mod q = (1 + (y mod q) ^ p mod q) mod q›
      by (simp add: cong-def mod-add-right-eq power-mod)
    hence ‹x' ^ p mod q = (1 + y' ^ p mod q) mod q›
      by (metis ‹x' = x mod int q› ‹y' = y mod int q› nat-mod-as-int
          of-nat-Suc of-nat-power plus-1-eq-Suc zmod-int)
    with * ‹x' ∈ {1..q − 1}› ‹y' ∈ {1..q − 1}› show False by blast
  qed
qed


lemma auxiliary-prime-iff-enum-mod [code] :
  — We will have a more optimized version later.
  ‹aux-prime p q ⟷
   prime p ∧ prime q ∧
   ¬ q dvd p ∧ p ^ ((q − 1) div gcd p (q − 1)) mod q ≠ 1 ∧
   (∀ x ∈ {1..q − 1}. let x-p-mod = x ^ p mod q
                      in ∀ y ∈ {1..q − 1}. x-p-mod ≠ (1 + y ^ p mod q) mod q)›
proof (cases ‹prime p›; cases ‹prime q›)
  assume ‹prime p› and ‹prime q›
  from ‹prime q› have ‹q ≠ 0› by auto
  show ?thesis
    unfolding auxiliary-prime-def not-pth-power-iff-mod[OF ‹prime p› ‹prime q›]
      non-consecutivity-condition-iff-enum-mod[OF ‹q ≠ 0›] by blast
next
  show ‹¬ prime q ⟹ ?thesis›
    and ‹¬ prime q ⟹ ?thesis›
    and ‹¬ prime p ⟹ ?thesis›
    by (simp-all add: auxiliary-prime-def)
qed
```

We can for example compute pairs of auxiliary primes less than *110*.

```
value ‹[(p, q). p ← [1..110], q ← [1..110], aux-prime (nat p) (nat q)]›


lemma auxiliary-primeI' :
  ‹⟦prime p; prime q; ¬ q dvd p; p ^ ((q − 1) div gcd p (q − 1)) mod q ≠ 1;
    ⋀x y. x ∈ {1..q − 1} ⟹ y ∈ {1..q − 1} ⟹ [x ^ p ≠ 1 + y ^ p] (mod q)⟧
   ⟹ aux-prime p q›
  by (simp add: auxiliary-prime-iff-enum-mod cong-def mod-Suc-eq)


lemma two-is-not-auxiliary-prime : ‹¬ aux-prime p 2›
  by (simp add: auxiliary-prime-iff-enum-mod) presburger


lemma auxiliary-prime-of-2 : ‹aux-prime 2 q ⟷ q = 3 ∨ q = 5›
proof (rule iffI)
  show ‹q = 3 ∨ q = 5 ⟹ aux-prime 2 q›
  proof (elim disjE)
    show ‹q = 3 ⟹ aux-prime 2 q›
    proof (intro auxiliary-primeI')
      show ‹prime (2 :: nat)› and ‹q = 3 ⟹ prime q› by simp-all
    next
      fix x y assume ‹q = 3› ‹x ∈ {1..q − 1}› ‹y ∈ {1..q − 1}›
     hence ‹x = 1 ∧ y = 1 ∨ x = 1 ∧ y = 2 ∨ x = 2 ∧ y = 1 ∨ x = 2 ∧ y = 2›
        by simp (metis le-Suc-eq le-antisym numeral-2-eq-2)
      with ‹q = 3› show ‹[x² ≠ 1 + y²] (mod q)›
        by (fastforce simp add: cong-def)
    next
      show ‹q = 3 ⟹ ¬ q dvd 2› by simp
    next
      show ‹q = 3 ⟹ 2 ^ ((q − 1) div gcd 2 (q − 1)) mod q ≠ 1› by simp
    qed
  next
    show ‹q = 5 ⟹ aux-prime 2 q›
    proof (intro auxiliary-primeI')
      show ‹prime (2 :: nat)› and ‹q = 5 ⟹ prime q› by simp-all
    next
      fix x y assume ‹q = 5› ‹x ∈ {1..q − 1}› ‹y ∈ {1..q − 1}›
      hence ‹(x = 1 ∨ x = 2 ∨ x = 3 ∨ x = 4) ∧ (y = 1 ∨ y = 2 ∨ y = 3 ∨ y
= 4)›
        by (simp add: numeral-eq-Suc) linarith
      with ‹q = 5› show ‹[x² ≠ 1 + y²] (mod q)›
        by (fastforce simp add: cong-def)
    next
      show ‹q = 5 ⟹ ¬ q dvd 2› by simp
    next
      show ‹q = 5 ⟹ 2 ^ ((q − 1) div gcd 2 (q − 1)) mod q ≠ 1› by simp
    qed
```

qed
next
assume *aux-q* : ‹*aux-prime 2 q*›
with *two-is-not-auxiliary-prime* **have** ‹$q \neq 2$› **by** *blast*
show ‹$q = 3 \vee q = 5$›
proof (*rule ccontr*)
assume ‹$\neg (q = 3 \vee q = 5)$›
with ‹$q \neq 2$› *auxiliary-primeD*(*1−2*)[*OF aux-q*] *prime-prime-factor*
le-neq-implies-less prime-ge-2-nat
have ‹*prime q*› ‹*odd q*› ‹$2 < q$› **by** *metis+*

hence ‹$5 \leq q$›
by (*metis Suc-1* ‹$\neg (q = 3 \vee q = 5)$› *add.commute add-Suc-right eval-nat-numeral*(*3*)
even-numeral not-less-eq-eq numeral-Bit0 order-antisym-conv plus-1-eq-Suc
prime-ge-2-nat)
with ‹*prime q*› **have** ‹*gcd 4 q* = (*1* :: *int*)›
by (*metis coprime-imp-gcd-eq-1 eval-nat-numeral*(*3*) *gcd.commute less-Suc-eq*
of-nat-1 order-less-le-trans prime-nat-iff″ zero-less-numeral)
with *cong-solve-dvd-int* **obtain** *inv-4* :: *int*
where *inv-4*: ‹[$4 * inv\text{-}4 = 1$] (*mod q*)›
by (*metis dvd-refl gcd-int-int-eq of-nat-numeral*)
define *x* **where** ‹$x \equiv 1 + inv\text{-}4$›
define *y* **where** ‹$y \equiv 1 - inv\text{-}4$›

from *inv-4* **have** ‹[$x^2 = 1 + y^2$] (*mod q*)›
by (*simp add*: *x-def y-def power2-eq-square cong-iff-dvd-diff ring-class.ring-distribs*)
moreover obtain x' y' :: *nat* **where** ‹[$x' = x$] (*mod q*)› ‹[$y' = y$] (*mod q*)›
by (*metis* ‹*prime q*› *cong-less-unique-int cong-sym int-eq-iff of-nat-0-less-iff*
prime-gt-0-nat)
ultimately have ‹[$x'^2 = 1 + y'^2$] (*mod q*)›
by (*simp flip*: *cong-int-iff*)
(*meson cong-add cong-pow cong-refl cong-sym-eq cong-trans*)
moreover have ‹[$x' \neq 0$] (*mod q*)›
proof (*rule ccontr*)
assume ‹$\neg$ [$x' \neq 0$] (*mod q*)›
with ‹[$x' = x$] (*mod q*)› **have** ‹[$x = 0$] (*mod q*)›
by (*metis cong-0-iff cong-dvd-iff int-dvd-int-iff*)
hence ‹[$4 * x = 0$] (*mod q*)›
by (*metis cong-scalar-left mult-zero-right*)
with *cong-add*[*OF cong-refl*[*of 4*] *inv-4*] **have** ‹*q dvd 5*›
by (*simp add*: *x-def*) (*metis cong-0-iff cong-dvd-iff int-ops*(*3*) *of-nat-dvd-iff*)
with ‹*prime q*› **have** ‹$q = 5$› **by** (*auto intro*: *primes-dvd-imp-eq*)
with ‹$\neg (q = 3 \vee q = 5)$› **show** *False* **by** *blast*
qed
moreover have ‹[$y' \neq 0$] (*mod q*)›
proof (*rule ccontr*)
assume ‹$\neg$ [$y' \neq 0$] (*mod q*)›
with ‹[$y' = y$] (*mod q*)› **have** ‹[$y = 0$] (*mod q*)›
by (*metis cong-0-iff cong-dvd-iff int-dvd-int-iff*)

```
      hence ‹[4 * y = 0] (mod q)›
        by (metis cong-scalar-left mult-zero-right)
      with cong-diff[OF cong-refl[of 4] inv-4] have ‹q dvd 3›
       by (simp add: y-def) (metis cong-0-iff cong-dvd-iff int-ops(3) of-nat-dvd-iff)
      with ‹prime q› have ‹q = 3› by (auto intro: primes-dvd-imp-eq)
      with ‹¬ (q = 3 ∨ q = 5)› show False by blast
    qed
    ultimately have ‹[(int x')² = 1 + (int y')²] (mod q) ∧
      [int x' ≠ 0] (mod q) ∧ [int y' ≠ 0] (mod q)›
      by (metis cong-int-iff of-nat-0 of-nat-Suc of-nat-power plus-1-eq-Suc)
    with auxiliary-primeD(3) aux-q show False by blast
  qed
qed
```

An auxiliary prime q of p is generally of the form $q = (2::'a) * n * p + 1$.

```
lemma auxiliary-prime-pattern-aux :
  ‹∃x y. [x ≠ 0] (mod q) ∧ [y ≠ 0] (mod q) ∧ [x ^ p = 1 + y ^ p] (mod q)›
  if ‹p ≠ 0› ‹prime q› ‹coprime p (q − 1)› ‹odd q›
proof −
  from bezout-nat ‹coprime p (q − 1)› ‹p ≠ 0›
  obtain u v where bez : ‹u * p = 1 + v * (q − 1)›
    by (metis add.commute coprime-imp-gcd-eq-1 mult.commute)
  have ‹[x ≠ 0] (mod q) ⟹ [(x ^ v) ^ (q − 1) = 1] (mod q)› for x
    by (meson cong-0-iff fermat-theorem prime-dvd-power ‹prime q›)
  hence * : ‹[(x ^ u) ^ p = x] (mod q)› for x
    by (fold power-mult, unfold bez, unfold power-add power-mult)
      (metis cong-0-iff cong-def cong-scalar-left ‹prime q›
        mult.right-neutral power-one-right prime-dvd-mult-iff)
  obtain x0 y0 where ‹[x0 ≠ 0] (mod q)› ‹[y0 ≠ 0] (mod q)› ‹[x0 = 1 + y0]
(mod q)›
    by (metis ‹odd q› ‹prime q› cong-0-iff cong-refl not-prime-unit
        one-add-one prime-prime-factor two-is-prime-nat)
  from * this(3) have ‹[(x0 ^ u) ^ p = 1 + (y0 ^ u) ^ p] (mod q)›
    by (metis cong-add-lcancel-nat cong-def)
  moreover from ‹[x0 ≠ 0] (mod q)› ‹[y0 ≠ 0] (mod q)›
  have ‹[x0 ^ u ≠ 0] (mod q)› ‹[y0 ^ u ≠ 0] (mod q)›
    by (meson cong-0-iff prime-dvd-power ‹prime q›)+
  ultimately show ?thesis by blast
qed
```

```
theorem auxiliary-prime-pattern :
  ‹p = 2 ∧ (q = 3 ∨ q = 5) ∨ odd p ∧ (∃n≥1. q = 2 * n * p + 1)› if aux-p :
‹aux-prime p q›
proof −
  from auxiliary-prime-of-2 consider ‹p = 2› ‹q = 3 ∨ q = 5› | ‹odd p› ‹q ≠ 2›
   by (metis aux-p auxiliary-prime-def prime-prime-factor two-is-not-auxiliary-prime)
  thus ?thesis
  proof cases
```

```
    show ‹p = 2 ⟹ q = 3 ∨ q = 5 ⟹ ?thesis› by blast
  next
    assume ‹odd p› ‹q ≠ 2›
    have ‹2 < q› ‹odd q›
      by (use ‹q ≠ 2› auxiliary-prime-def le-neq-implies-less prime-ge-2-nat that in
presburger)
        (metis ‹q ≠ 2› auxiliary-prime-def prime-prime-factor that two-is-prime-nat)
   from aux-p have ‹prime p› and ‹prime q› by (simp-all add: auxiliary-primeD(1−2))
    from euler-criterion[OF ‹prime q› ‹2 < q›]
    have ∗ : ‹[Legendre p q = p ^ ((q − 1) div 2)] (mod q)› by simp
    have ‹¬ coprime p (q − 1)›
      by (metis auxiliary-prime-def cong-0-iff coprime-iff-gcd-eq-1
          div-by-1 fermat-theorem not-pth-power-iff aux-p)
    with ‹prime p› prime-imp-coprime have ‹p dvd q − 1› by blast
    then obtain k where ‹q = k ∗ p + 1›
      by (metis add.commute ‹prime q› dvd-div-mult-self
          le-add-diff-inverse less-or-eq-imp-le prime-gt-1-nat)
    with ‹odd q› ‹odd p› have ‹even k› by simp
    then obtain n where ‹k = 2 ∗ n› by fast
    with ‹q = k ∗ p + 1› have ‹q = 2 ∗ n ∗ p + 1› by blast
    with ‹2 < q› have ‹1 ≤ n›
      by (metis One-nat-def add.commute less-one linorder-not-less
          mult-is-0 one-le-numeral plus-1-eq-Suc)
    with ‹odd p› ‹q = 2 ∗ n ∗ p + 1› show ?thesis by blast
  qed
qed
```

```
lemma auxiliary-prime-imp-less : ‹aux-prime p q ⟹ p < q›
  by (auto dest: auxiliary-prime-pattern simp add: less-Suc-eq)

lemma auxiliary-primeE :
  assumes ‹aux-prime p q›
  obtains ‹p = 2› ‹q = 3› | ‹p = 2› ‹q = 5›
  | n where ‹odd p› ‹1 ≤ n› ‹q = 2 ∗ n ∗ p + 1›
    ‹NC p (2 ∗ n ∗ p + 1)› ‹PPP p (2 ∗ n ∗ p + 1)›
  by (metis assms auxiliary-prime-def auxiliary-prime-pattern)
```

With this, we can quickly eliminate numbers that cannot be auxiliary.

```
declare auxiliary-prime-iff-enum-mod [code del]

lemma auxiliary-prime-iff-enum-mod-optimized [code] :
  ‹aux-prime p q ⟷
   p = 2 ∧ (q = 3 ∨ q = 5) ∨
   p < q ∧
   2 ∗ p dvd q − 1 ∧
   prime p ∧ prime q ∧
   ¬ q dvd p ∧ p ^ ((q − 1) div gcd p (q − 1)) mod q ≠ 1 ∧
```

(∀ x ∈ {1..q − 1}. let x-p-mod = x ^ p mod q
                 in ∀ y ∈ {1..q − 1}. x-p-mod ≠ (1 + y ^ p mod q) mod q)›
  by (*fold auxiliary-prime-iff-enum-mod*)
    (*metis add-diff-cancel-right′ auxiliary-prime-imp-less auxiliary-prime-of-2*
      auxiliary-prime-pattern dvd-refl even-mult-iff mult-dvd-mono)

value ‹[(p, q). p ← [1..1000], q ← [1..110], aux-prime (nat p) (nat q)]›

5.2 Sophie Germain Primes are auxiliary

When *p* is an *odd prime* and *2 ∗ p + 1* is also a *prime* (what we call a Sophie Germain *prime*), *2 ∗ p + 1* is automatically an *aux-prime*.

lemma *SophGer-prime-imp-auxiliary-prime* :
  fixes *p* **assumes** ‹*SG p*› **defines** *q-def*: ‹*q ≡ 2 ∗ p + 1*›
  shows ‹*aux-prime p q*›
proof (*rule auxiliary-primeI*)
  from *SophGer-primeD(2−3)[OF* ‹*SG p*›]
  show ‹*prime p*› **and** ‹*prime q*› **by** (*unfold q-def*)
next
  from *SophGer-primeD[OF* ‹*SG p*›, *folded q-def*]
  have ‹*odd p*› ‹*prime p*› ‹*prime q*› ‹*prime (int q)*› ‹*p ≠ 0*› **by** *simp-all*
  show ‹*NC p q*›
  proof (*rule ccontr*)
    assume ‹¬ *NC p q*›
    then obtain *x y* :: *int* **where** ‹[x ≠ 0] (mod q)› ‹[y ≠ 0] (mod q)›
      ‹[x ^ p = 1 + y ^ p] (mod q)› **by** *blast*
    from *SG-simps.p-th-power-mod-q* ‹[x ≠ 0] (mod q)› ‹*SG p*› *cong-0-iff q-def*
    consider ‹[x ^ p = 1] (mod q)› | ‹[x ^ p = − 1] (mod q)› **by** *blast*
    thus *False*
    proof *cases*
      assume ‹[x ^ p = 1] (mod q)›
      with ‹[x ^ p = 1 + y ^ p] (mod q)› **have** ‹[y ^ p = 0] (mod q)›
        by (*metis add.right-neutral cong-add-lcancel cong-sym cong-trans*)
      with ‹[y ≠ 0] (mod q)› **show** *False*
        by (*meson* ‹*prime (int q)*› *cong-0-iff prime-dvd-power-int*)
    next
      assume ‹[x ^ p = − 1] (mod q)›
      with ‹[x ^ p = 1 + y ^ p] (mod q)›
      have ‹[− 1 = 1 + y ^ p] (mod q)› **by** (*metis cong-def*)
      moreover have ‹− (1::int) = 1 + − 2› **by** *force*
      ultimately have ‹[y ^ p = − 2] (mod q)›
        by (*metis cong-add-lcancel cong-sym*)
      with ‹*odd p*› *cong-minus-minus-iff* **have** ‹[(− y) ^ p = 2] (mod q)› **by** *force*
      with *cong-sym* **have** ‹∃ x :: int. [2 = x ^ p] (mod q)› **by** *blast*
      with ‹p ≠ 0› *exists-nth-power-mod-prime-iff[OF* ‹*prime q*›]
      have ‹([2 = 0] (mod q) ∨ [4 = 1] (mod q))›
        by (*simp add*: *q-def flip*: *cong-int-iff*)
      hence ‹p ≤ 1›
      proof (*elim disjE*)

```
        from ‹p ≠ 0› show ‹[2 = 0] (mod q) ⟹ p ≤ 1›
          by (auto simp add: q-def cong-def)
      next
        from linorder-not-less show ‹[4 = 1] (mod q) ⟹ p ≤ 1›
          by (fastforce simp add: q-def cong-def)
      qed
      with ‹SG p› show False
        by (metis ‹prime p› linorder-not-less prime-nat-iff)
    qed
  qed
next
  from ‹SG p›[THEN SophGer-primeD(3), folded q-def]
  have ‹prime q› ‹prime (int q)› by simp-all
  from SG-simps.p-th-power-mod-q[OF ‹SG p›]
  have ‹¬ q dvd x ⟹ [x ^ p = 1] (mod q) ∨ [x ^ p = − 1] (mod q)› for x :: int
    unfolding q-def .
  moreover have ‹[p ≠ (1 :: int)] (mod q)› ‹[p ≠ − 1] (mod q)›
     using SG-simps.notcong-p(1, 3)[OF ‹SG p›] cong-sym unfolding q-def by
blast+
  ultimately have ‹¬ q dvd x ⟹ [p ≠ x ^ p] (mod q)› for x :: int
    using cong-trans by blast
  moreover have ‹q dvd x ⟹ [p ≠ x ^ p] (mod q)› for x :: int
    by (metis SG-simps.pos Suc-eq-plus1 ‹SG p› cong-dvd-iff dvd-power dvd-trans
          int-dvd-int-iff less-add-Suc1 mult.commute mult-2-right nat-dvd-not-less
q-def)
  ultimately show ‹PPP p q› by blast
qed
```

5.3 Main Theorems

```
theorem Sophie-Germain-auxiliary-prime :
  ‹q dvd x ∨ q dvd y ∨ q dvd z›
  if ‹x ^ p + y ^ p = z ^ p› and ‹aux-prime p q› for x y z :: int
proof (rule ccontr)
  assume not-dvd : ‹¬ (q dvd x ∨ q dvd y ∨ q dvd z)›
  from auxiliary-primeD[OF ‹aux-prime p q›]
  have ‹prime p› ‹prime q› ‹NC p q› by simp-all

  have ‹coprime x q›
   by (meson coprime-commute not-dvd prime-imp-coprime prime-nat-int-transfer
‹prime q›)
  with bezout-int[of x q] obtain inv-x v :: int where ‹inv-x ∗ x + v ∗ q = 1› by
auto
  hence inv-x : ‹[x ∗ inv-x = 1] (mod q)› by (metis cong-iff-lin mult.commute)

  from ‹x ^ p + y ^ p = z ^ p› have ‹z ^ p = x ^ p + y ^ p› ..
  hence ‹(inv-x ∗ z) ^ p = (inv-x ∗ x) ^ p + (inv-x ∗ y) ^ p›
    by (metis distrib-left power-mult-distrib)
  with inv-x have ‹[(inv-x ∗ z) ^ p = 1 + (inv-x ∗ y) ^ p] (mod q)›
```

by (*metis cong-add-rcancel cong-pow mult.commute power-one*)
moreover from *inv-x* ‹*prime q*› **have** ‹$[(inv\text{-}x * z) \mathbin{\widehat{}} p \neq 0]$ $(mod\ q)$›
by (*metis cong-dvd-iff dvd-0-right not-dvd not-prime-unit prime-dvd-mult-eq-int prime-dvd-power prime-nat-int-transfer*)
moreover from *inv-x* ‹*prime q*› **have** ‹$[(inv\text{-}x * y) \mathbin{\widehat{}} p \neq 0]$ $(mod\ q)$›
by (*metis cong-dvd-iff dvd-0-right not-dvd not-prime-unit prime-dvd-mult-eq-int prime-dvd-power prime-nat-int-transfer*)
moreover obtain $y'\ z'$ :: *int* **where** ‹$[y' = inv\text{-}x * y]$ $(mod\ q)$› ‹$[z' = inv\text{-}x * z]$ $(mod\ q)$›
by (*metis* ‹*prime q*› *cong-less-unique-int cong-sym int-eq-iff of-nat-0-less-iff prime-gt-0-nat*)
ultimately show *False*
by (*metis* ‹*NC p q*› ‹*prime p*› ‹*prime q*› *cong-0-iff prime-dvd-power-iff prime-gt-0-nat prime-nat-int-transfer*)
qed

theorem *Sophie-Germain-generalization* :
‹$\nexists x\ y\ z :: int.\ x \mathbin{\widehat{}} p + y \mathbin{\widehat{}} p = z \mathbin{\widehat{}} p\ \wedge$
$[x \neq 0]\ (mod\ p^2) \wedge [y \neq 0]\ (mod\ p^2) \wedge [z \neq 0]\ (mod\ p^2)$›
if *odd-p* : ‹*odd p*› **and** *aux-prime* : ‹*aux-prime p q*›
proof (*rule ccontr*) — The proof is done by contradiction.
from ‹*aux-prime p q*› **have** *prime-p* : ‹*prime p*›
by (*metis auxiliary-primeD*(*1*))
hence *not-p-0* : ‹$p \neq 0$› **and** *prime-int-p* : ‹*prime* (*int p*)› **by** *simp-all*
from ‹*aux-prime p q*› **have** *prime-q* : ‹*prime q*›
by (*metis auxiliary-primeD*(*2*))
hence *prime-int-q* : ‹*prime* (*int q*)› **by** *simp*
from ‹*odd p*› ‹*prime p*› **have** *p-ge-3* : ‹$3 \leq p$›
by (*simp add*: *numeral-eq-Suc*)
(*metis Suc-le-eq dvd-refl le-antisym not-less-eq-eq prime-gt-Suc-0-nat*)

assume ‹$\neg\ (\nexists x\ y\ z.\ x \mathbin{\widehat{}} p + y \mathbin{\widehat{}} p = z \mathbin{\widehat{}} p \wedge [x \neq 0]\ (mod\ int\ (p^2)) \wedge$
$[y \neq 0]\ (mod\ int\ (p^2)) \wedge [z \neq 0]\ (mod\ (int\ p)^2))$›
then obtain $x\ y\ z$ :: *int*
where *fermat* : ‹$x \mathbin{\widehat{}} p + y \mathbin{\widehat{}} p = z \mathbin{\widehat{}} p$›
and *not-cong-0* : ‹$[x \neq 0]\ (mod\ p^2)$› ‹$[y \neq 0]\ (mod\ p^2)$› ‹$[z \neq 0]\ (mod\ p^2)$›
by *auto*

— We first assume without loss of generality that x, y and z are setwise *coprime*.
let *?gcd* = ‹*Gcd* $\{x,\ y,\ z\}$›
wlog *coprime* : ‹*?gcd* = *1*› **goal** *False* **generalizing** $x\ y\ z$ **keeping** *fermat not-cong-0*
using *FLT-setwise-coprime-reduction-mod-version*[*OF fermat not-cong-0*]
hypothesis **by** *blast*

— Then we can deduce that x, y and z are pairwise *coprime*.
have *coprime-x-y* : ‹*coprime x y*›

by (*fact FLT-setwise-coprime-imp-pairwise-coprime*[*OF* ‹$p \neq 0$› *fermat coprime*])
have *coprime-y-z* : ‹*coprime y z*›
proof (*subst coprime-minus-right-iff* [*symmetric*],
rule FLT-setwise-coprime-imp-pairwise-coprime[*OF* ‹$p \neq 0$›])
from *fermat* ‹*odd p*› **show** ‹y ^ p + (− z) ^ p = (− x) ^ p› **by** *simp*
next
show ‹*Gcd* {y, − z, − x} = *1*›
by (*metis Gcd-insert coprime gcd-neg1-int insert-commute*)
qed
have *coprime-x-z* : ‹*coprime x z*›
proof (*subst coprime-minus-right-iff* [*symmetric*],
rule FLT-setwise-coprime-imp-pairwise-coprime[*OF* ‹$p \neq 0$›])
from *fermat* ‹*odd p*› **show** ‹x ^ p + (− z) ^ p = (− y) ^ p› **by** *simp*
next
show ‹*Gcd* {x, − z, − y} = *1*›
by (*metis Gcd-insert coprime gcd-neg1-int insert-commute*)
qed

— From ⟦x^p + y^p = z^p; *aux-prime p q*⟧ ⟹ *int q dvd x* ∨ *int q dvd y* ∨ *int q dvd z* we have that among x, y and z, one (and only one, see below) is a multiple of q.
from *Sophie-Germain-auxiliary-prime*[*OF fermat aux-prime*]
have *q-dvd-xyz* : ‹*q dvd x* ∨ *q dvd y* ∨ *q dvd z*› **.**

— Without loss of generality, we can assume that x is a multiple of q.
wlog *q-dvd-z* : ‹*q dvd z*› **goal** *False* **generalizing** *x y z*
keeping *fermat not-cong-0 coprime-x-y coprime-y-z coprime-x-z*
proof −
from *negation q-dvd-xyz* **have** ‹*q dvd x* ∨ *q dvd y*› **by** *simp*
thus *False*
proof (*elim disjE*)
show ‹*q dvd x* ⟹ *False*›
by (*erule hypothesis*[*of x* ‹− *y*› *z*])
(*use fermat not-cong-0* ‹*odd p*› **in**
‹*simp-all add*: *cong-0-iff coprime-commute coprime-x-y coprime-x-z coprime-y-z*›)
next
show ‹*q dvd y* ⟹ *False*›
by (*erule hypothesis*[*of y* ‹− *x*› *z*])
(*use fermat not-cong-0* ‹*odd p*› **in**
‹*simp-all add*: *cong-0-iff coprime-commute coprime-x-y coprime-x-z coprime-y-z*›)
qed
qed

define *S* **where** ‹$S \equiv \lambda y\ z :: int.\ \sum k = 0..p - 1.\ (- z)$ ^ $(p - 1 - k) * y$ ^ k›

— Now we prove that x, y or x is dividable by p.
have *p-dvd-xyz* : ‹*p dvd x* ∨ *p dvd y* ∨ *p dvd z*›

```
proof (rule ccontr)
  assume ‹¬ (p dvd x ∨ p dvd y ∨ p dvd z)›
  hence ‹[x ≠ 0] (mod p)› ‹[y ≠ 0] (mod p)› ‹[z ≠ 0] (mod p)›
    by (simp-all add: cong-0-iff)
  from Sophie-Germain-lemma[OF ‹odd p› ‹prime p›, of ‹− z› x y]
    coprime-x-y fermat ‹[z ≠ 0] (mod p)›
  obtain a α where ‹x + y = a ^ p› ‹S x y = α ^ p›
   by (simp add: S-def ‹odd p› coprime-commute) (meson cong-0-iff dvd-minus-iff)
  from Sophie-Germain-lemma[OF ‹odd p› ‹prime p›, of ‹− x› z ‹− y›]
    coprime-y-z fermat ‹[x ≠ 0] (mod int p)›
  obtain b β where ‹z − y = b ^ p› ‹S z (− y) = β ^ p›
   by (simp add: S-def ‹odd p› coprime-commute) (meson cong-0-iff dvd-minus-iff)
  from Sophie-Germain-lemma[OF ‹odd p› ‹prime p›, of ‹− y› z ‹− x›]
    coprime-x-z fermat ‹[y ≠ 0] (mod p)›
  obtain c γ where ‹z − x = c ^ p› ‹S z (− x) = γ ^ p›
   by (simp add: S-def ‹odd p› coprime-commute) (meson cong-0-iff dvd-minus-iff)

  have ‹a ^ p + b ^ p + c ^ p = x + y + (z − y) + (z − x)›
    by (simp add: ‹x + y = a ^ p› ‹z − y = b ^ p› ‹z − x = c ^ p›)
  also have ‹. . . = 2 ∗ z› by simp
  also from ‹q dvd z› have ‹[. . . = 0] (mod q)› by (simp add: cong-0-iff)
  finally have ‹[a ^ p + b ^ p + c ^ p = 0] (mod q)› .

  have ‹[a = 0] (mod q) ∨ [b = 0] (mod q) ∨ [c = 0] (mod q)›
  proof (rule ccontr)
    assume ‹¬ ([a = 0] (mod q) ∨ [b = 0] (mod q) ∨ [c = 0] (mod q))›
    hence ‹[a ≠ 0] (mod q)› ‹[b ≠ 0] (mod q)› ‹[c ≠ 0] (mod q)› by simp-all
    from ‹[c ≠ 0] (mod q)› have ‹gcd c q = 1›
      by (meson aux-prime auxiliary-prime-def cong-0-iff coprime-iff-gcd-eq-1
          residues-prime.p-coprime-right-int residues-prime-def)
    from bezout-int[of c q, unfolded this]
    obtain u v where ‹u ∗ c + v ∗ int q = 1› by blast
    with ‹[a ≠ 0] (mod q)› have ‹[u ≠ 0] (mod q)›
    by (metis cong-0-iff cong-dvd-iff cong-iff-lin dvd-mult mult.commute unit-imp-dvd)

    from ‹[a ^ p + b ^ p + c ^ p = 0] (mod q)›
    have ‹[(u ∗ a) ^ p + (u ∗ b) ^ p + (u ∗ c) ^ p = 0] (mod q)›
      by (simp add: power-mult-distrib)
        (metis cong-scalar-left distrib-left mult.commute mult-zero-left)
    also from ‹u ∗ c + v ∗ int q = 1› have ‹u ∗ c = 1 − v ∗ int q› by simp
    finally have ‹[(u ∗ a) ^ p + (u ∗ b) ^ p + (1 − v ∗ int q) ^ p = 0] (mod
q)› .
    moreover have ‹[(1 − v ∗ int q) ^ p = 1] (mod q)›
      by (metis add-uminus-conv-diff cong-0-iff cong-add-lcancel-0
          cong-pow dvd-minus-iff dvd-triv-right power-one)
    ultimately have ‹[(u ∗ a) ^ p + (u ∗ b) ^ p + 1 = 0] (mod q)›
      by (meson cong-add-lcancel cong-sym cong-trans)
    hence ‹[1 + (u ∗ b) ^ p = (− (u ∗ a)) ^ p] (mod q)›
      by (simp add: ‹odd p› cong-iff-dvd-diff) presburger
```

hence ‹$[(-\ (u * a))\ \hat{}\ p = 1 + (u * b)\ \hat{}\ p]\ (mod\ q)$› **by** (*fact cong-sym*)
moreover from ‹$[a \neq 0]\ (mod\ q)$› ‹$[u \neq 0]\ (mod\ q)$›
cong-prime-prod-zero-int[*OF* - ‹*prime* (*int q*)›, *of u a*] *cong-minus-minus-iff*
have ‹$[-\ u * a \neq 0]\ (mod\ q)$› **by** *force*
moreover from ‹$[b \neq 0]\ (mod\ q)$› ‹$[u \neq 0]\ (mod\ q)$›
cong-prime-prod-zero-int[*OF* - ‹*prime* (*int q*)›, *of u b*]
have ‹$[u * b \neq 0]\ (mod\ q)$› **by** *blast*
ultimately show *False*
using *aux-prime auxiliary-primeD*(*3*) **by** *auto*
qed
hence ‹*q dvd a*›
proof (*elim disjE*)
show ‹$[a = 0]\ (mod\ q) \Longrightarrow q\ dvd\ a$› **by** (*simp add*: *cong-0-iff*)
next
assume ‹$[b = 0]\ (mod\ q)$›
with ‹$z - y = b\ \hat{}\ p$› ‹*q dvd z*› ‹*prime p*› **have** ‹*q dvd y*›
by (*metis cong-0-iff cong-dvd-iff cong-iff-dvd-diff*
dvd-power dvd-trans prime-gt-0-nat)
with ‹*prime* (*int q*)› ‹*q dvd z*› ‹*coprime y z*› **have** *False*
by (*metis coprime-def not-prime-unit*)
thus ‹*q dvd a*› **..**
next
assume ‹$[c = 0]\ (mod\ q)$›
with ‹$z - x = c\ \hat{}\ p$› ‹*q dvd z*› ‹*prime p*› **have** ‹*q dvd x*›
by (*metis cong-0-iff cong-dvd-iff cong-iff-dvd-diff*
dvd-power dvd-trans prime-gt-0-nat)
with ‹*prime* (*int q*)› ‹*q dvd z*› ‹*coprime x z*› **have** *False*
by (*metis coprime-def not-prime-unit*)
thus ‹*q dvd a*› **..**
qed
with ‹$x + y = a\ \hat{}\ p$› ‹$p \neq 0$› ‹*prime* (*int q*)› **have** ‹$[y = -\ x]\ (mod\ q)$›
by (*metis add.commute add.inverse-inverse add-uminus-conv-diff*
bot-nat-0.not-eq-extremum cong-iff-dvd-diff prime-dvd-power-int-iff)
hence ‹$[S\ x\ y = p * x\ \hat{}\ (p - 1)]\ (mod\ q)$›
unfolding *S-def* **by** (*fact Sophie-Germain-lemma-computation-cong-simp*[*OF*
‹$p \neq 0$›])
moreover from ‹$z - x = c\ \hat{}\ p$› ‹*q dvd z*› **have** ‹$[x = (-\ c)\ \hat{}\ p]\ (mod\ q)$›
by (*simp add*: ‹*odd p*› *cong-iff-dvd-diff*)
(*metis add-diff-cancel-left′ diff-diff-eq2*)
ultimately have ‹$[S\ x\ y = p * ((-\ c)\ \hat{}\ p)\ \hat{}\ (p - 1)]\ (mod\ q)$›
by (*meson cong-pow cong-scalar-left cong-trans*)
also have ‹$S\ x\ y = \alpha\ \hat{}\ p$› **by** (*fact* ‹$S\ x\ y = \alpha\ \hat{}\ p$›)
also have ‹$((-\ c)\ \hat{}\ p)\ \hat{}\ (p - 1) = ((-\ c)\ \hat{}\ (p - 1))\ \hat{}\ p$›
by (*metis mult.commute power-mult*)
finally have ‹$[\alpha\ \hat{}\ p = p * ((-\ c)\ \hat{}\ (p - 1))\ \hat{}\ p]\ (mod\ q)$› **.**

have ‹$gcd\ q\ ((-\ c)\ \hat{}\ (p - 1)) = 1$›
by (*metis* ‹$[x = (-\ c)\ \hat{}\ p]\ (mod\ int\ q)$› ‹*int q dvd z*› *cong-dvd-iff*
coprime-def coprime-imp-gcd-eq-1 coprime-x-z dvd-mult not-prime-unit

power-eq-if prime-imp-coprime-int ‹$p \neq 0$› ‹*prime* (*int q*)›)
with *bezout-int* **obtain** *u v*
where ‹$u * int\ q + v * (- c) \hat{} (p - 1) = 1$› **by** *metis*
hence ‹$v * (- c) \hat{} (p - 1) = 1 - u * int\ q$› **by** *simp*
have ‹$[1 - u * int\ q = 1]\ (mod\ q)$› **by** (*simp add*: *cong-iff-lin*)
from ‹$[\alpha \hat{} p = p * ((- c) \hat{} (p - 1)) \hat{} p]\ (mod\ q)$›
have ‹$[(v * \alpha) \hat{} p = p * (v * (- c) \hat{} (p - 1)) \hat{} p]\ (mod\ q)$›
by (*simp add*: *power-mult-distrib*) (*metis cong-scalar-left mult.left-commute*)
with ‹$[1 - u * int\ q = 1]\ (mod\ int\ q)$› **have** ‹$[(v * \alpha) \hat{} p = p]\ (mod\ q)$›
by (*unfold* ‹$v * (- c) \hat{} (p - 1) = 1 - u * int\ q$›)
(*metis cong-pow cong-scalar-left cong-trans mult.comm-neutral power-one*)
with ‹*aux-prime p q*›[*THEN auxiliary-primeD*(*4*)] *cong-sym* **show** *False* **by** *blast*
qed

— Without loss of generality, we can assume that it is *z*.
wlog *p-dvd-z* : ‹*p dvd z*› **goal** *False* **generalizing** *x y z S*
keeping *fermat not-cong-0 coprime-x-y coprime-y-z coprime-x-z S-def*
proof −
from *negation p-dvd-xyz* **have** ‹$p\ dvd\ x \lor p\ dvd\ y$› **by** *simp*
thus *False*
proof (*elim disjE*)
show ‹$p\ dvd\ x \Longrightarrow False$›
by (*erule hypothesis*[*of x* ‹− *y*› *z*])
(*use fermat not-cong-0* ‹*odd p*› **in**
‹*simp-all add*: *cong-0-iff coprime-commute coprime-x-y coprime-x-z coprime-y-z*›)
next
show ‹$p\ dvd\ y \Longrightarrow False$›
by (*erule hypothesis*[*of y* ‹− *x*› *z*])
(*use fermat not-cong-0* ‹*odd p*› **in**
‹*simp-all add*: *cong-0-iff coprime-commute coprime-x-y coprime-x-z coprime-y-z*›)
qed
qed

— The rest of the proof consists in deducing that actually p^2 divides *z*, which contradicts $[z \neq 0]\ (mod\ int\ (p^2))$.
from ‹*p dvd z*› *coprime-x-z coprime-y-z*
have ‹$[x \neq 0]\ (mod\ p)$› ‹$[y \neq 0]\ (mod\ p)$›
by (*simp-all add*: *cong-0-iff*)
(*meson not-coprimeI not-prime-unit* ‹*prime* (*int p*)›)+

from *Sophie-Germain-lemma*[*OF* ‹*odd p*› ‹*prime p*›, *of* ‹− *x*› *z* ‹− *y*›]
coprime-y-z fermat ‹$[x \neq 0]\ (mod\ int\ p)$›
obtain *b β* **where** ‹$z - y = b \hat{} p$› ‹$S\ z\ (- y) = \beta \hat{} p$›
by (*simp add*: *S-def* ‹*odd p*› *coprime-commute*) (*meson cong-0-iff dvd-minus-iff*)
from *Sophie-Germain-lemma*[*OF* ‹*odd p*› ‹*prime p*›, *of* ‹− *y*› *z* ‹− *x*›]
coprime-x-z fermat ‹$[y \neq 0]\ (mod\ p)$›

obtain *c* γ **where** ‹$z - x = c$ ^ p› ‹$S\ z\ (- x) = \gamma$ ^ p›
by (*simp add*: *S-def* ‹*odd p*› *coprime-commute*) (*meson cong-0-iff dvd-minus-iff*)

from *fermat* **have** ‹$(- z)$ ^ $p + x$ ^ $p + y$ ^ $p = 0$› **by** (*simp add*: ‹*odd p*›)
from *Sophie-Germain-lemma-computation*[*OF* ‹*odd p*›] *fermat*
have ‹$(x + y) * S\ x\ y = z$ ^ p› **by** (*simp add*: *S-def*)
with ‹$[z \neq 0]$ (*mod* p^2)› **have** ‹$x + y \neq 0$› ‹$S\ x\ y \neq 0$› **by** *auto*

define z' **where** ‹$z' \equiv z$ *div* p›
from ‹*p dvd z*› ‹$[z \neq 0]$ (*mod* p^2)› ‹$p \neq 0$›
have ‹$z = z' * p$› ‹$[z' \neq 0]$ (*mod p*)›
by (*simp-all add*: *cong-0-iff* z'*-def dvd-div-iff-mult power2-eq-square*)

from *Sophie-Germain-lemma-only-possible-prime-common-divisor*[*OF* ‹*prime p*›
- ‹*coprime x y*›]
have ‹$\exists\ q \in \#$*prime-factorization r*. $q \neq p \Longrightarrow \neg$ *r dvd* $x + y \lor \neg$ *r dvd S x y*›
for *r* :: *nat*
unfolding *S-def*
by (*meson dvd-trans in-prime-factors-iff int-dvd-int-iff*
of-nat-eq-iff prime-nat-int-transfer)
from *this*[*OF Ex-other-prime-factor*[*OF - -* ‹*prime p*›]]
have ‹*r dvd* $x + y \Longrightarrow$ *r dvd S x y* $\Longrightarrow r = 0 \lor (\exists k.\ r = p$ ^ $k)$› **for** *r* :: *nat*
by *auto*
moreover have ‹$\neg$ (*p* ^ *k dvd* $x + y \land p$ ^ *k dvd S x y*)› **if** ‹$1 < k$› **for** *k*
proof (*rule ccontr*)
assume ‹$\neg$ ($\neg$ (*p* ^ *k dvd* $x + y \land p$ ^ *k dvd S x y*))›
moreover from ‹$1 < k$› **have** ‹p^2 *dvd p* ^ *k*›
by (*simp add*: *le-imp-power-dvd*)
ultimately have ‹p^2 *dvd* $x + y$› ‹p^2 *dvd S x y*›
by (*meson dvd-trans of-nat-dvd-iff*)+
from ‹p^2 *dvd* $x + y$› **have** ‹$[y = - x]$ (*mod* p^2)›
by (*simp add*: *add.commute cong-iff-dvd-diff*)
hence ‹$[S\ x\ y = p * x$ ^ $(p - 1)]$ (*mod* p^2)›
unfolding *S-def* **by** (*fact Sophie-Germain-lemma-computation-cong-simp*[*OF*
‹$p \neq 0$›])
moreover from ‹$[x \neq 0]$ (*mod p*)› ‹$z = z' * p$› ‹$[z \neq 0]$ (*mod* p^2)› ‹*prime* (*int*
p)›
have ‹$\neg\ p^2$ *dvd* $p * x$ ^ $(p - 1)$›
by (*metis cong-0-iff dvd-mult-cancel-left mult-zero-right*
of-nat-power power2-eq-square prime-dvd-power-int)
ultimately show *False* **using** ‹p^2 *dvd S x y*› *cong-dvd-iff* **by** *blast*
qed
ultimately have *p-only-nontrivial-div* :
‹*r dvd* $x + y \Longrightarrow$ *r dvd S x y* $\Longrightarrow r = 1 \lor r = p$› **for** *r* :: *nat*
by (*metis* ‹$S\ x\ y \neq 0$› *dvd-0-left-iff less-one*
linorder-neqE-nat of-nat-eq-0-iff power-0 power-one-right)
— *p* is therefore the only possible nontrivial common divisor.

define *mul-x-plus-y* **where** ‹*mul-x-plus-y* = *multiplicity p* $(x + y)$›

define *mul-S-x-y* **where** ‹*mul-S-x-y* = *multiplicity p* (*S x y*)›
from ‹$(x + y) * S\ x\ y = z \mathbin{\char`\^} p$›
have ‹$(x + y) * S\ x\ y = z' \mathbin{\char`\^} p * p \mathbin{\char`\^} p$›
  by (*simp add*: ‹$z = z' * p$› *power-mult-distrib*)

have ‹*mul-x-plus-y* + *mul-S-x-y* = *multiplicity p* (z ^ p)›
  unfolding *mul-x-plus-y-def mul-S-x-y-def*
  by (*metis* ‹$(x + y) * S\ x\ y = z$ ^ p› ‹$S\ x\ y \neq 0$› ‹$x + y \neq 0$› *prime-def*
    prime-elem-multiplicity-mult-distrib ‹*prime* (*int p*)›)
also have ‹z ^ $p = z'$ ^ $p * p$ ^ p›
  by (*simp add*: ‹$z = z' * p$› *power-mult-distrib*)
also have ‹*multiplicity p* ... = *p*›
  by (*metis* ‹[$z' \neq 0$] (*mod int p*)› *aux-prime auxiliary-prime-def cong-0-iff*
    mult-of-nat-commute multiplicity-decomposeI of-nat-eq-0-iff of-nat-power
    prime-dvd-power-iff prime-gt-0-nat ‹$p \neq 0$› ‹*prime* (*int p*)›)
finally have ‹*mul-x-plus-y* + *mul-S-x-y* = *p*› **.**

moreover have ‹($2 \leq$ *mul-x-plus-y* $\longrightarrow$ *mul-S-x-y* ≤ 1) $\wedge$
      ($2 \leq$ *mul-S-x-y* $\longrightarrow$ *mul-x-plus-y* ≤ 1)›
proof (*rule ccontr*)
  assume ‹$\neg$ *?thesis*›
  hence ‹$2 \leq$ *mul-x-plus-y* $\wedge$ $2 \leq$ *mul-S-x-y*› **by** *linarith*
  hence ‹p^2 *dvd* $(x + y)$ $\wedge$ p^2 *dvd* $(S\ x\ y)$›
    by (*simp add*: *mul-x-plus-y-def mul-S-x-y-def multiplicity-dvd'*)
  thus *False*
    by (*metis cong-0-iff less-numeral-extra*(*3*) *one-eq-prime-power-iff*
      p-dvd-z p-only-nontrivial-div pos2 ‹[$z \neq 0$] (*mod* p^2)› ‹*prime p*›)
qed
ultimately consider ‹*mul-x-plus-y* = *p*› ‹*mul-S-x-y* = *0*›
  | ‹*mul-x-plus-y* = *0*› ‹*mul-S-x-y* = *p*›
  | ‹*mul-x-plus-y* = *1*› ‹*mul-S-x-y* = $p - 1$›
  | ‹*mul-x-plus-y* = $p - 1$› ‹*mul-S-x-y* = *1*›
  by (*metis Nat.add-diff-assoc add-cancel-right-right add-diff-cancel-left'*
    diff-is-0-eq not-less-eq-eq one-add-one plus-1-eq-Suc)
thus *False*
proof *cases*

  assume ‹*mul-x-plus-y* = *p*› ‹*mul-S-x-y* = *0*›
  from ‹*mul-x-plus-y* = *p*› **have** ‹*p dvd* $(x + y)$›
    by (*metis mul-x-plus-y-def not-dvd-imp-multiplicity-0* ‹$p \neq 0$›)
  hence ‹[$y = - x$] (*mod p*)›
    by (*simp add*: *add.commute cong-iff-dvd-diff*)
  hence ‹[$S\ x\ y = S\ x\ (- x)$] (*mod p*)›
    unfolding *S-def*
    by (*meson cong-minus-minus-iff cong-pow cong-scalar-right cong-sum*)
  also have ‹$S\ x\ (- x) = (\sum k = 0..p - 1.\ x$ ^ $(p - 1))$›
    unfolding *S-def*
    by (*rule sum.cong*[*OF refl*], *simp*)

```
    (metis One-nat-def diff-Suc-eq-diff-pred le-add-diff-inverse2 power-add)
  also from ‹p ≠ 0› have ‹... = p * x ^ (p − 1)› by simp
  finally have ‹[S x y = p * x ^ (p − 1)] (mod p)› .
  with ‹[x ≠ 0] (mod p)› have ‹p dvd S x y› by (simp add: cong-dvd-iff)
  with ‹mul-S-x-y = 0› show False
    by (metis ‹S x y ≠ 0› mul-S-x-y-def not-one-le-zero not-prime-unit
        power-dvd-iff-le-multiplicity power-one-right ‹prime (int p)›)
next

  assume ‹mul-x-plus-y = 0› ‹mul-S-x-y = p›
  from ‹mul-S-x-y = p› have ‹p dvd S x y›
    by (metis mul-S-x-y-def not-dvd-imp-multiplicity-0 ‹p ≠ 0›)
  from Sophie-Germain-lemma-computation[OF ‹odd p›]
  have ‹(x + y) * S x y = x ^ p + y ^ p› unfolding S-def .
  moreover from ‹p dvd S x y› have ‹[(x + y) * S x y = 0] (mod p)›
    by (simp add: cong-0-iff)
  moreover have ‹[x ^ p + y ^ p = x + y] (mod p)›
  proof (rule cong-add)
    have ‹x ^ p = x * x ^ (p − 1)›
      by (simp add: power-eq-if ‹p ≠ 0›)
    moreover from ‹[x ≠ 0] (mod p)› have ‹[x ^ (p − 1) = 1] (mod p)›
      using cong-0-iff fermat-theorem-int ‹prime p› by blast
    ultimately show ‹[x ^ p = x] (mod p)›
      by (metis cong-scalar-left mult.right-neutral)
  next
    have ‹y ^ p = y * y ^ (p − 1)›
      by (simp add: power-eq-if ‹p ≠ 0›)
    moreover from ‹[y ≠ 0] (mod p)› have ‹[y ^ (p − 1) = 1] (mod p)›
      using cong-0-iff fermat-theorem-int ‹prime p› by blast
    ultimately show ‹[y ^ p = y] (mod p)›
      by (metis cong-scalar-left mult.right-neutral)
  qed
  ultimately have ‹p dvd x + y›
    by (simp add: cong-0-iff cong-dvd-iff)
  with ‹mul-x-plus-y = 0› show False
    by (metis ‹x + y ≠ 0› mul-x-plus-y-def multiplicity-eq-zero-iff
        not-prime-unit ‹prime (int p)›)
next

  define x-plus-y' where ‹x-plus-y' ≡ (x + y) div p›
  define S-x-y' where ‹S-x-y' ≡ (S x y) div p ^ (p − 1)›
  define s where ‹s ≡ x + y›
  let ?f = ‹λk. (p choose k) * (− x) ^ k * s ^ (p − k)›
  let ?f' = ‹λk. (p choose k) * (− x) ^ k * s ^ (p − 1 − k)›

  assume ‹mul-x-plus-y = 1› ‹mul-S-x-y = p − 1›
  hence ‹s = p * x-plus-y'› ‹S x y = p ^ (p − 1) * S-x-y'›
    by (simp-all add: s-def x-plus-y'-def S-x-y'-def)
      (metis dvd-mult-div-cancel mul-x-plus-y-def multiplicity-dvd power-Suc0-right,
```

metis dvd-mult-div-cancel mul-S-x-y-def multiplicity-dvd)

from *fermat* **have** ‹$s * S\ x\ y = (s - x) \widehat{}\ p + x \widehat{}\ p$›
by (*simp add*: *s-def* ‹$(x + y) * S\ x\ y = z \widehat{}\ p$›)
also have ‹$s - x = (- x + s)$› **by** *simp*
also have ‹$(- x + s) \widehat{}\ p = (\sum k \le p.\ (p\ choose\ k) * (- x) \widehat{}\ k * s \widehat{}\ (p - k))$›
by (*fact binomial-ring*)
also have ‹$\ldots = (\sum k \in \{0..p - 1\}.\ ?f\ k) + (\sum k \in \{p\}.\ ?f\ k)$›
by (*rule sum-Un-eq*[*symmetric*])
(*auto simp add*: *linorder-not-le prime-gt-0-nat* ‹*prime p*›)
also have ‹$(\sum k \in \{0..p - 1\}.\ ?f\ k) = (\sum k \in \{0\}.\ ?f\ k) + (\sum k \in \{1..p - 1\}.\ ?f\ k)$›
by (*rule sum-Un-eq*[*symmetric*]) *auto*
also have ‹$(\sum k \in \{1..p - 1\}.\ ?f\ k) = (\sum k \in \{1..p - 2\}.\ ?f\ k) + (\sum k \in \{p - 1\}.\ ?f\ k)$›
by (*rule sum-Un-eq*[*symmetric*]) (*use* ‹$3 \le p$› **in** *auto*)
also have ‹$(\sum k \in \{0\}.\ ?f\ k) = s * s \widehat{}\ (p - 1)$› **by** (*simp add*: *power-eq-if* ‹$p \neq 0$›)
also have ‹$(\sum k \in \{1..p - 2\}.\ ?f\ k) = (\sum k \in \{1..p - 2\}.\ s * ?f'\ k)$›
by (*rule sum.cong*, *simp-all*)
(*metis Suc-diff-Suc diff-less less-2-cases-iff less-zeroE*
linorder-neqE order.strict-iff-not power-Suc ‹$p \neq 0$›)
also have ‹$\ldots = s * (\sum k \in \{1..p - 2\}.\ ?f'\ k)$›
by (*simp add*: *mult.assoc sum-distrib-left*)
also have ‹$(\sum k \in \{p - 1\}.\ ?f\ k) = s * p * x \widehat{}\ (p - 1)$›
by (*simp del*: *One-nat-def*, *subst binomial-symmetric*[*symmetric*])
(*use* ‹$3 \le p$› **in** ‹*auto simp add*: ‹*odd p*››)
finally have ‹$s * S\ x\ y =$
$s * (s \widehat{}\ (p - 1) + (\sum k \in \{1..p - 2\}.\ ?f'\ k) + p * x \widehat{}\ (p - 1))$›
by (*simp add*: ‹*odd p*› *distrib-left int-distrib*(*4*))
hence *S-x-y-developed* : ‹$S\ x\ y = s \widehat{}\ (p - 1) + (\sum k \in \{1..p - 2\}.\ ?f'\ k) + p * x \widehat{}\ (p - 1)$›
using ‹$x + y \neq 0$› *mult-cancel-left* **unfolding** *s-def* **by** *blast*
have ‹$[p * x \widehat{}\ (p - 1) = 0]\ (mod\ p^2)$›
proof (*rule cong-trans*[*OF - cong-sym*])
show ‹$[p * x \widehat{}\ (p - 1) = 0 + 0 + p * x \widehat{}\ (p - 1)]\ (mod\ p^2)$› **by** *simp*
next
show ‹$[0 = 0 + 0 + p * x \widehat{}\ (p - 1)]\ (mod\ p^2)$›
proof (*rule cong-trans*)
have ‹$p \widehat{}\ (p - 1)\ dvd\ S\ x\ y$›
by (*simp add*: ‹$S\ x\ y = p \widehat{}\ (p - 1) * S\text{-}x\text{-}y'$›)
moreover from ‹$3 \le p$› **have** ‹$p \widehat{}\ 2\ dvd\ p \widehat{}\ (p - 1)$›
by (*auto intro*: *le-imp-power-dvd*)
ultimately show ‹$[0 = S\ x\ y]\ (mod\ p^2)$›
by (*metis cong-0-iff cong-sym dvd-trans of-nat-dvd-iff*)
next
show ‹$[S\ x\ y = 0 + 0 + p * x \widehat{}\ (p - 1)]\ (mod\ p^2)$›
proof (*unfold S-x-y-developed*, *rule cong-add*[*OF - cong-refl*])
show ‹$[s \widehat{}\ (p - 1) + (\sum k = 1..p - 2.\ ?f'\ k) = 0 + 0]\ (mod\ p^2)$›

```
          proof (rule cong-add)
            have ‹p dvd s› by (simp add: ‹s = p ∗ x-plus-y'›)
            hence ‹p ^ (p − 1) dvd s ^ (p − 1)› by (simp add: dvd-power-same)
            moreover from ‹3 ≤ p› have ‹p ^ 2 dvd p ^ (p − 1)›
              by (auto intro: le-imp-power-dvd)
            ultimately show ‹[s ^ (p − 1) = 0] (mod p²)›
              by (metis cong-0-iff dvd-trans of-nat-dvd-iff)
          next
            show ‹[∑ k = 1..p − 2. ?f' k = 0] (mod p²)›
            proof (rule cong-trans)
              show ‹[∑ k = 1..p − 2. ?f' k = ∑ k∈{1..p − 2}. 0] (mod p²)›
              proof (rule cong-sum)
                fix k assume ‹k ∈ {1..p − 2}›
                from ‹k ∈ {1..p − 2}› have ‹p dvd (p choose k)›
                  by (auto intro: dvd-choose-prime simp add: ‹prime p›)
                moreover from ‹k ∈ {1..p − 2}› have ‹p dvd s ^ (p − 1 − k)›
             by (auto simp add: ‹prime p› ‹s = p ∗ x-plus-y'› prime-dvd-power-int-iff)
                ultimately show ‹[?f' k = 0] (mod p²)›
                  by (simp add: cong-0-iff mult-dvd-mono power2-eq-square)
              qed
            next
              show ‹[∑ k = 1..p − 2. 0 = 0] (mod int (p²))› by simp
            qed
          qed
        qed
      qed
    qed
    hence ‹p dvd x ^ (p − 1)› by (simp add: cong-iff-dvd-diff power2-eq-square ‹p
≠ 0›)
    with prime-dvd-power ‹prime (int p)› have ‹p dvd x› by blast
    with ‹[x ≠ 0] (mod p)› show False by (simp add: cong-0-iff)
  next

    define x-plus-y' where ‹x-plus-y' ≡ (x + y) div p ^ (p − 1)›
    define S-x-y' where ‹S-x-y' ≡ (S x y) div p›
    assume ‹mul-x-plus-y = p − 1› ‹mul-S-x-y = 1›
    with ‹(x + y) ∗ S x y = z ^ p› ‹mul-x-plus-y + mul-S-x-y = p›
    have ‹x-plus-y' ∗ S-x-y' = z' ^ p›
      unfolding x-plus-y'-def S-x-y'-def z'-def
      by (metis (no-types, opaque-lifting)
          div-mult-div-if-dvd div-power mul-S-x-y-def mul-x-plus-y-def
          multiplicity-dvd of-nat-power p-dvd-z power-add power-one-right)
    have ‹coprime x-plus-y' S-x-y'›
    proof (rule ccontr)
      assume ‹¬ coprime x-plus-y' S-x-y'›
      then obtain r where ‹prime r› ‹r dvd x-plus-y'› ‹r dvd S-x-y'›
        by (metis coprime-absorb-left not-coprime-nonzeroE
            prime-factor-int unit-imp-dvd zdvd1-eq)
      from ‹r dvd x-plus-y'› have ‹r dvd x + y›
```

by (*metis* ‹*mul-x-plus-y* = *p* − *1*› *dvd-div-iff-mult dvd-mult-left*
mul-x-plus-y-def multiplicity-dvd of-nat-eq-0-iff of-nat-power
power-not-zero ‹*p* ≠ *0*› *x-plus-y′-def*)
moreover from ‹*r dvd S-x-y′*› **have** ‹*r dvd S x y*›
by (*metis S-x-y′-def* ‹*mul-S-x-y* = *1*› *dvd-mult-div-cancel dvd-trans*
dvd-triv-right mul-S-x-y-def multiplicity-dvd power-one-right)
ultimately have ‹*r* = *p*›
by (*metis* ‹*prime r*› *not-prime-1 p-only-nontrivial-div*
pos-int-cases prime-gt-0-int prime-nat-int-transfer)
with ‹[*z′* ≠ *0*] (*mod int p*)› ‹*r dvd S-x-y′*› ‹*prime p*› ‹*prime* (*int p*)›
‹*x-plus-y′* ∗ *S-x-y′* = *z′* ^ *p*› **show** *False*
by (*metis cong-0-iff dvd-trans dvd-triv-right prime-dvd-power-int-iff prime-gt-0-nat*)
qed
from *prod-is-some-powerE*[*OF* ‹*coprime x-plus-y′ S-x-y′*› ‹*x-plus-y′* ∗ *S-x-y′* =
z′ ^ *p*›]
obtain *a* **where** ‹*normalize x-plus-y′* = *a* ^ *p*› **by** *blast*
moreover from *prod-is-some-powerE*
[*OF coprime-commute*[*THEN iffD1*, *OF* ‹*coprime x-plus-y′ S-x-y′*›]]
obtain *α* **where** ‹*normalize S-x-y′* = *α* ^ *p*›
by (*metis* ‹*x-plus-y′* ∗ *S-x-y′* = *z′* ^ *p*› *mult.commute*)
ultimately have ‹*x-plus-y′* = (*if 0* ≤ *x-plus-y′ then a* ^ *p else* (− *a*) ^ *p*)›
‹*S-x-y′* = (*if 0* ≤ *S-x-y′ then α* ^ *p else* (− *α*) ^ *p*)›
by (*metis* ‹*odd p*› *abs-of-nonneg abs-of-nonpos add.inverse-inverse*
linorder-linear normalize-int-def power-minus-odd)+
then obtain *α a* **where** ‹*x-plus-y′* = *a* ^ *p*› ‹*S-x-y′* = *α* ^ *p*› **by** *meson*

from *Sophie-Germain-lemma*[*OF* ‹*odd p*› ‹*prime p*›, *of* ‹− *x*› *z* ‹− *y*›]
coprime-y-z fermat ‹[*x* ≠ *0*] (*mod int p*)›
obtain *b β* **where** ‹*z* − *y* = *b* ^ *p*› ‹*S z* (− *y*) = *β* ^ *p*›
by (*simp add*: *S-def* ‹*odd p*› *coprime-commute*) (*meson cong-0-iff dvd-minus-iff*)
from *Sophie-Germain-lemma*[*OF* ‹*odd p*› ‹*prime p*›, *of* ‹− *y*› *z* ‹− *x*›]
coprime-x-z fermat ‹[*y* ≠ *0*] (*mod p*)›
obtain *c γ* **where** ‹*z* − *x* = *c* ^ *p*› ‹*S z* (− *x*) = *γ* ^ *p*›
by (*simp add*: *S-def* ‹*odd p*› *coprime-commute*) (*meson cong-0-iff dvd-minus-iff*)

have ‹¬ *p dvd c*›
by (*metis* ‹[*x* ≠ *0*] (*mod int p*)› ‹*z* − *x* = *c* ^ *p*› *cong-0-iff cong-dvd-iff*
cong-iff-dvd-diff dvd-def dvd-trans p-dvd-z power-eq-if ‹*p* ≠ *0*›)
have ‹¬ *p dvd b*›
by (*metis* ‹[*y* ≠ *0*] (*mod int p*)› ‹*z* − *y* = *b* ^ *p*› *cong-0-iff cong-dvd-iff*
cong-iff-dvd-diff dvd-def dvd-trans p-dvd-z power-eq-if ‹*p* ≠ *0*›)

have ‹*p dvd 2* ∗ *z* − *x* − *y*›
by (*metis* ‹*mul-S-x-y* = *1*› ‹*mul-x-plus-y* + *mul-S-x-y* = *p*› *comm-monoid-add-class.add-0*
diff-diff-eq dvd-diff int-ops(*2*)
mul-x-plus-y-def not-dvd-imp-multiplicity-0 one-dvd p-dvd-z prime-dvd-mult-iff
wlog-keep.prime-int-p)
hence ‹[*2* ∗ *z* − *x* − *y* = *0*] (*mod p*)› **by** (*simp add*: *cong-0-iff*)
also from ‹*z* − *x* = *c* ^ *p*› ‹*z* − *y* = *b* ^ *p*›

```
    have ‹2 * z − x − y = c ^ p + b ^ p› by presburger
    also have ‹... = c ^(p − 1) * c + b ^(p − 1) * b›
      by (simp add: power-eq-if ‹p ≠ 0›)
    finally have ‹[c ^(p − 1) * c + b ^(p − 1) * b = 0] (mod p)› .
    moreover have ‹[c ^(p − 1) = 1] (mod p)›
      by (fact fermat-theorem-int[OF ‹prime p› ‹¬ p dvd c›])
    moreover have ‹[b ^(p − 1) = 1] (mod p)›
      by (fact fermat-theorem-int[OF ‹prime p› ‹¬ p dvd b›])
    ultimately have ‹[1 * c + 1 * b = 0] (mod p)›
      by (meson cong-add cong-scalar-right cong-sym-eq cong-trans)
    hence ‹[c + b = 0] (mod p)› by simp
    hence ‹[b = − c] (mod p)› by (simp add: add.commute cong-iff-dvd-diff)
    hence ‹[S c b = p * c ^(p − 1)] (mod p)›
      unfolding S-def
      by (fact Sophie-Germain-lemma-computation-cong-simp[OF ‹p ≠ 0›])
    hence ‹p dvd S c b› by (simp add: cong-dvd-iff)
     moreover from ‹[c + b = 0] (mod p)› have ‹p dvd c + b› by (simp add:
cong-dvd-iff)
    moreover from Sophie-Germain-lemma-computation[OF ‹odd p›]
    have ‹c ^ p + b ^ p = (c + b) * S c b› unfolding S-def ..
    ultimately have ‹p² dvd c ^ p + b ^ p›
      by (simp add: mult-dvd-mono power2-eq-square)
    moreover have ‹p² dvd x + y›
      by (metis ‹mul-S-x-y = 1› ‹mul-x-plus-y + mul-S-x-y = p› add-0
          bot-nat-0.not-eq-extremum dvd-trans linorder-not-less
          mul-x-plus-y-def multiplicity-dvd' nat-dvd-not-less odd-even-add
          odd-p of-nat-power one-dvd prime-prime-factor ‹prime p›)
    ultimately have ‹p² dvd 2 * z›
      by (metis ‹2 * z − x − y = c ^ p + b ^ p› diff-diff-eq zdvd-zdiffD)
    moreover have ‹coprime (p²) 2› by (simp add: ‹odd p›)
    ultimately have ‹p² dvd z›
      by (simp add: coprime-dvd-mult-left-iff mult.commute)
    with ‹[z ≠ 0] (mod p²)› show False by (simp add: cong-0-iff)
  qed
qed
```

Since $SG\ p \Longrightarrow aux\text{-}prime\ p\ (2 * p + 1)$, this result is a generalization of $SG\ p \Longrightarrow \nexists x\ y\ z.\ x^p + y^p = z^p \wedge [x \neq 0]\ (mod\ int\ p) \wedge [y \neq 0]\ (mod\ int\ p) \wedge [z \neq 0]\ (mod\ int\ p)$.

References

[1] S. Francinou, H. Gianella, and S. Nicolas. *Oraux X-ENS Algèbre 1*. Cassini, 2014.

[2] A. Kiefer. Le théorème de Fermat vu par M. Le Blanc. *Brussels Summer School of Mathematics, Notes de la cinquième BSSM*, 2012.

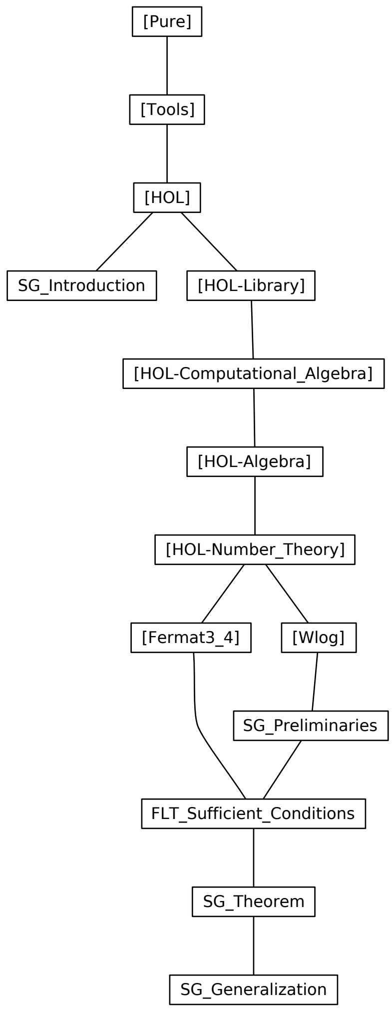

Figure 1: Dependency graph of the session `Sophie_Germain`