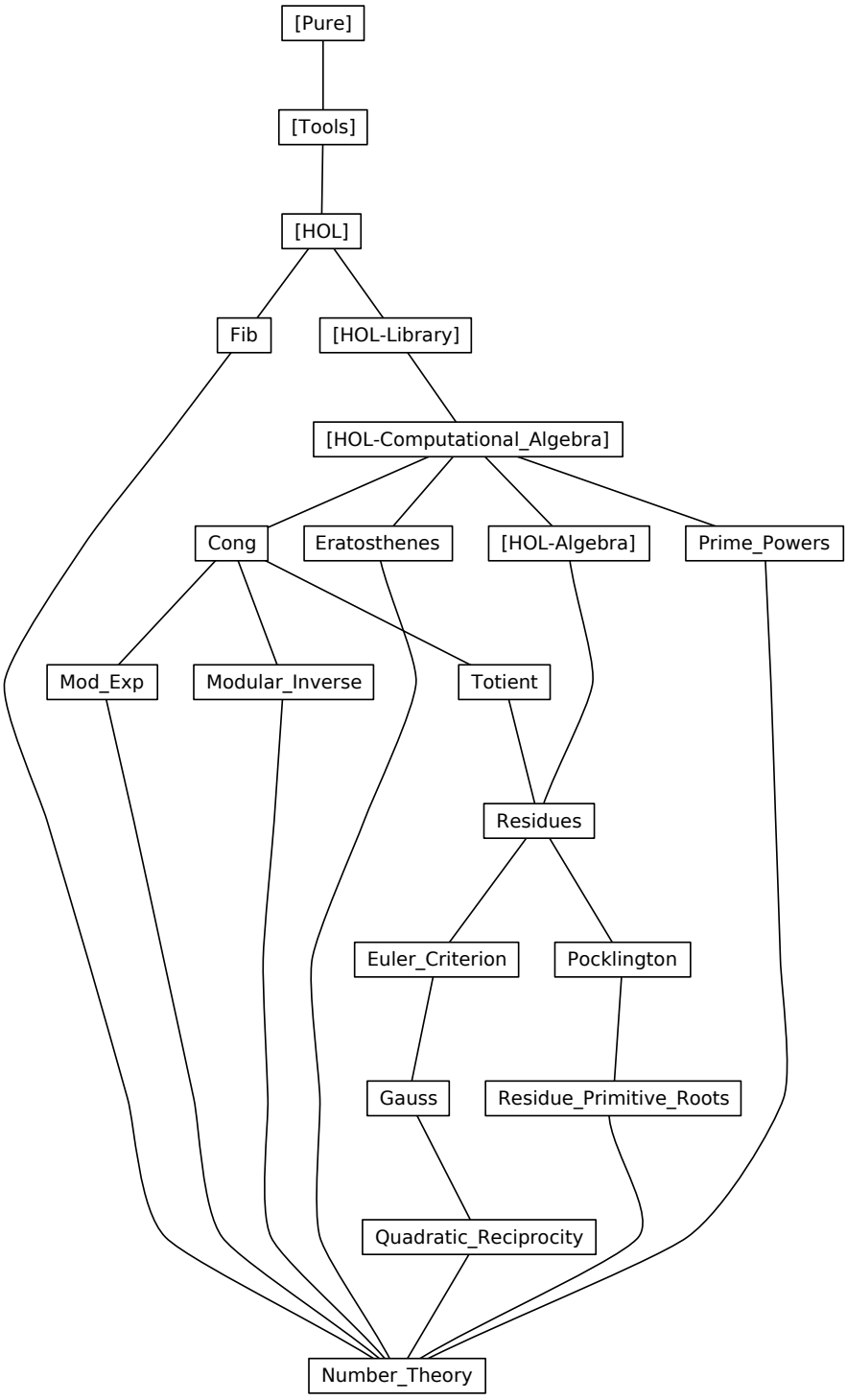
Various results of number theory

December 17, 2025

Contents


[Pure]
[Tools]
[HOL]
Fib
[HOL-Library]
[HOL-Computational_Algebra]
Cong
Eratosthenes
[HOL-Algebra]
Prime_Powers
Mod_Exp
Modular_Inverse
Totient
Residues
Euler_Criterion
Pocklington
Gauss
Residue_Primitive_Roots
Quadratic_Reciprocity
Number_Theory

1 The fibonacci function

theory *Fib*
imports *Complex-Main*
begin

1.1 Fibonacci numbers

fun *fib* :: *nat* ⇒ *nat*
where
fib0: *fib 0 = 0*
| *fib1*: *fib* (*Suc 0*) = *1*
| *fib2*: *fib* (*Suc* (*Suc n*)) = *fib* (*Suc n*) + *fib n*

1.2 Basic Properties

lemma *fib-1* [*simp*]: *fib 1 = 1*
by (*metis One-nat-def fib1*)

lemma *fib-2* [*simp*]: *fib 2 = 1*
using *fib.simps*(*3*) [*of 0*] **by** (*simp add*: *numeral-2-eq-2*)

lemma *fib-plus-2*: *fib* (*n* + *2*) = *fib* (*n* + *1*) + *fib n*
by (*metis Suc-eq-plus1 add-2-eq-Suc′ fib.simps*(*3*))

lemma *fib-add*: *fib* (*Suc* (*n* + *k*)) = *fib* (*Suc k*) ∗ *fib* (*Suc n*) + *fib k* ∗ *fib n*
by (*induct n rule*: *fib.induct*) (*auto simp add*: *field-simps*)

lemma *fib-neq-0-nat*: *n* > *0* ⟹ *fib n* > *0*
by (*induct n rule*: *fib.induct*) *auto*

lemma *fib-Suc-mono*: *fib m* ≤ *fib* (*Suc m*)
by(*induction m*) *auto*

lemma *fib-mono*: *m* ≤ *n* ⟹ *fib m* ≤ *fib n*
by (*simp add*: *fib-Suc-mono lift-Suc-mono-le*)

1.3 More efficient code

The naive approach is very inefficient since the branching recursion leads to many values of *fib* being computed multiple times. We can avoid this by "remembering" the last two values in the sequence, yielding a tail-recursive version. This is far from optimal (it takes roughly $O(n \cdot M(n))$ time where $M(n)$ is the time required to multiply two n-bit integers), but much better than the naive version, which is exponential.

fun *gen-fib* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where
gen-fib a b 0 = a
| *gen-fib a b* (*Suc 0*) = *b*

| *gen-fib a b (Suc (Suc n)) = gen-fib b (a + b) (Suc n)*

lemma *gen-fib-recurrence*: *gen-fib a b (Suc (Suc n)) = gen-fib a b n + gen-fib a b (Suc n)*
by (*induct a b n rule*: *gen-fib.induct*) *simp-all*

lemma *gen-fib-fib*: *gen-fib (fib n) (fib (Suc n)) m = fib (n + m)*
by (*induct m rule*: *fib.induct*) (*simp-all del*: *gen-fib.simps*(*3*) *add*: *gen-fib-recurrence*)

lemma *fib-conv-gen-fib*: *fib n = gen-fib 0 1 n*
using *gen-fib-fib*[*of 0 n*] **by** *simp*

declare *fib-conv-gen-fib* [*code*]

1.4 A Few Elementary Results

Concrete Mathematics, page 278: Cassini's identity. The proof is much easier using integers, not natural numbers!

lemma *fib-Cassini-int*: *int (fib (Suc (Suc n)) * fib n) - int((fib (Suc n))^2) = - ((-1)^n)*
by (*induct n rule*: *fib.induct*) (*auto simp add*: *field-simps power2-eq-square power-add*)

lemma *fib-Cassini-nat*:
*fib (Suc (Suc n)) * fib n =*
(if even n then (fib (Suc n))^2 - 1 else (fib (Suc n))^2 + 1)
using *fib-Cassini-int* [*of n*] **by** (*auto simp del*: *of-nat-mult of-nat-power*)

1.5 Law 6.111 of Concrete Mathematics

lemma *coprime-fib-Suc-nat*: *coprime (fib n) (fib (Suc n))*
apply (*induct n rule*: *fib.induct*)
apply (*simp-all add*: *coprime-iff-gcd-eq-1 algebra-simps*)
apply (*simp add*: *add.assoc* [*symmetric*])
done

lemma *gcd-fib-add*:
gcd (fib m) (fib (n + m)) = gcd (fib m) (fib n)
proof (*cases m*)
case *0*
then show *?thesis*
by *simp*
next
case (*Suc q*)
from *coprime-fib-Suc-nat* [*of q*]
have *coprime (fib (Suc q)) (fib q)*
by (*simp add*: *ac-simps*)
have *gcd (fib q) (fib (Suc q)) = Suc 0*
using *coprime-fib-Suc-nat* [*of q*] **by** *simp*

then have $*$: *gcd* (*fib n* $*$ *fib q*) (*fib n* $*$ *fib* (*Suc q*)) = *fib n*
by (*simp add*: *gcd-mult-distrib-nat* [*symmetric*])
moreover have *gcd* (*fib* (*Suc q*)) (*fib n* $*$ *fib q* + *fib* (*Suc n*) $*$ *fib* (*Suc q*)) =
gcd (*fib* (*Suc q*)) (*fib n* $*$ *fib q*)
using *gcd-add-mult* [*of fib* (*Suc q*)] **by** (*simp add*: *ac-simps*)
moreover have *gcd* (*fib* (*Suc q*)) (*fib n* $*$ *fib* (*Suc q*)) = *fib* (*Suc q*)
by *simp*
ultimately show *?thesis*
using *Suc* ‹*coprime* (*fib* (*Suc q*)) (*fib q*)›
by (*auto simp add*: *fib-add algebra-simps gcd-mult-right-right-cancel*)
qed

lemma *gcd-fib-diff*: $m \leq n \Longrightarrow$ *gcd* (*fib m*) (*fib* ($n - m$)) = *gcd* (*fib m*) (*fib n*)
by (*simp add*: *gcd-fib-add* [*symmetric*, *of* - $n-m$])

lemma *gcd-fib-mod*: $0 < m \Longrightarrow$ *gcd* (*fib m*) (*fib* (*n mod m*)) = *gcd* (*fib m*) (*fib n*)
proof (*induct n rule*: *less-induct*)
case (*less n*)
show *gcd* (*fib m*) (*fib* (*n mod m*)) = *gcd* (*fib m*) (*fib n*)
proof (*cases m* < *n*)
case *True*
then have $m \leq n$ **by** *auto*
with ‹$0 < m$› **have** $0 < n$ **by** *auto*
with ‹$0 < m$› ‹$m < n$› **have** $*$: $n - m < n$ **by** *auto*
have *gcd* (*fib m*) (*fib* (*n mod m*)) = *gcd* (*fib m*) (*fib* ((*n* − *m*) *mod m*))
by (*simp add*: *mod-if* [*of n*]) (*use* ‹*m* < *n*› **in** *auto*)
also have ... = *gcd* (*fib m*) (*fib* (*n* − *m*))
by (*simp add*: *less.hyps* $*$ ‹*0* < *m*›)
also have ... = *gcd* (*fib m*) (*fib n*)
by (*simp add*: *gcd-fib-diff* ‹*m* ≤ *n*›)
finally show *gcd* (*fib m*) (*fib* (*n mod m*)) = *gcd* (*fib m*) (*fib n*) **.**
next
case *False*
then show *gcd* (*fib m*) (*fib* (*n mod m*)) = *gcd* (*fib m*) (*fib n*)
by (*cases m* = *n*) *auto*
qed
qed

lemma *fib-gcd*: *fib* (*gcd m n*) = *gcd* (*fib m*) (*fib n*) — Law 6.111
by (*induct m n rule*: *gcd-nat-induct*) (*simp-all add*: *gcd-non-0-nat gcd.commute gcd-fib-mod*)

theorem *fib-mult-eq-sum-nat*: *fib* (*Suc n*) $*$ *fib n* = ($\sum k \in \{..n\}$. *fib k* $*$ *fib k*)
by (*induct n rule*: *nat.induct*) (*auto simp add*: *field-simps*)

1.6 Closed form

lemma *fib-closed-form*:
fixes φ ψ :: *real*

```
  defines φ ≡ (1 + sqrt 5) / 2
    and ψ ≡ (1 − sqrt 5) / 2
  shows of-nat (fib n) = (φ ^ n − ψ ^ n) / sqrt 5
proof (induct n rule: fib.induct)
  fix n :: nat
  assume IH1: of-nat (fib n) = (φ ^ n − ψ ^ n) / sqrt 5
  assume IH2: of-nat (fib (Suc n)) = (φ ^ Suc n − ψ ^ Suc n) / sqrt 5
  have of-nat (fib (Suc (Suc n))) = of-nat (fib (Suc n)) + of-nat (fib n) by simp
  also have ... = (φ^n * (φ + 1) − ψ^n * (ψ + 1)) / sqrt 5
    by (simp add: IH1 IH2 field-simps)
  also have φ + 1 = φ² by (simp add: φ-def field-simps power2-eq-square)
  also have ψ + 1 = ψ² by (simp add: ψ-def field-simps power2-eq-square)
  also have φ^n * φ² − ψ^n * ψ² = φ ^ Suc (Suc n) − ψ ^ Suc (Suc n)
    by (simp add: power2-eq-square)
  finally show of-nat (fib (Suc (Suc n))) = (φ ^ Suc (Suc n) − ψ ^ Suc (Suc n))
/ sqrt 5 .
qed (simp-all add: φ-def ψ-def field-simps)

lemma fib-closed-form′:
  fixes φ ψ :: real
  defines φ ≡ (1 + sqrt 5) / 2
    and ψ ≡ (1 − sqrt 5) / 2
  assumes n > 0
  shows fib n = round (φ ^ n / sqrt 5)
proof (rule sym, rule round-unique′)
  have |φ ^ n / sqrt 5 − of-int (int (fib n))| = |ψ| ^ n / sqrt 5
    by (simp add: fib-closed-form[folded φ-def ψ-def] field-simps power-abs)
  also {
    from assms have |ψ|^n ≤ |ψ|^1
      by (intro power-decreasing) (simp-all add: algebra-simps real-le-lsqrt)
    also have ... < sqrt 5 / 2 by (simp add: ψ-def field-simps)
    finally have |ψ|^n / sqrt 5 < 1/2 by (simp add: field-simps)
  }
  finally show |φ ^ n / sqrt 5 − of-int (int (fib n))| < 1/2 .
qed

lemma fib-asymptotics:
  fixes φ :: real
  defines φ ≡ (1 + sqrt 5) / 2
  shows (λn. real (fib n) / (φ ^ n / sqrt 5)) ⟶ 1
proof −
  define ψ :: real where ψ ≡ (1 − sqrt 5) / 2
  have φ > 1 by (simp add: φ-def)
  then have *: φ ≠ 0 by auto
  have (λn. (ψ / φ) ^ n) ⟶ 0
   by (rule LIMSEQ-power-zero) (simp-all add: φ-def ψ-def field-simps add-pos-pos)
  then have (λn. 1 − (ψ / φ) ^ n) ⟶ 1 − 0
    by (intro tendsto-diff tendsto-const)
  with * have (λn. (φ ^ n − ψ ^ n) / φ ^ n) ⟶ 1
```

by (*simp add*: *field-simps*)
then show *?thesis*
by (*simp add*: *fib-closed-form φ-def ψ-def*)
qed

1.7 Divide-and-Conquer recurrence

The following divide-and-conquer recurrence allows for a more efficient computation of Fibonacci numbers; however, it requires memoisation of values to be reasonably efficient, cutting the number of values to be computed to logarithmically many instead of linearly many. The vast majority of the computation time is then actually spent on the multiplication, since the output number is exponential in the input number.

lemma *fib-rec-odd*:
fixes φ ψ :: *real*
defines $\varphi \equiv (1 + sqrt\ 5)\ /\ 2$
and $\psi \equiv (1 - sqrt\ 5)\ /\ 2$
shows $fib\ (Suc\ (2 * n)) = fib\ n\hat{}2 + fib\ (Suc\ n)\hat{}2$
proof −
have $of\text{-}nat\ (fib\ n\hat{}2 + fib\ (Suc\ n)\hat{}2) = ((\varphi\ \hat{}\ n - \psi\ \hat{}\ n)^2 + (\varphi * \varphi\ \hat{}\ n - \psi * \psi\ \hat{}\ n)^2)/5$
by (*simp add*: *fib-closed-form*[*folded φ-def ψ-def*] *field-simps power2-eq-square*)
also
let $?A = \varphi\hat{}(2 * n) + \psi\hat{}(2 * n) - 2*(\varphi * \psi)\hat{}n + \varphi\hat{}(2 * n + 2) + \psi\hat{}(2 * n + 2) - 2*(\varphi * \psi)\hat{}(n + 1)$
have $(\varphi\ \hat{}\ n - \psi\ \hat{}\ n)^2 + (\varphi * \varphi\ \hat{}\ n - \psi * \psi\ \hat{}\ n)^2 = ?A$
by (*simp add*: *power2-eq-square algebra-simps power-mult power-mult-distrib*)
also have $\varphi * \psi = -1$
by (*simp add*: *φ-def ψ-def field-simps*)
then have $?A = \varphi\hat{}(2 * n + 1) * (\varphi + inverse\ \varphi) + \psi\hat{}(2 * n + 1) * (\psi + inverse\ \psi)$
by (*auto simp*: *field-simps power2-eq-square*)
also have $1 + sqrt\ 5 > 0$
by (*auto intro*: *add-pos-pos*)
then have $\varphi + inverse\ \varphi = sqrt\ 5$
by (*simp add*: *φ-def field-simps*)
also have $\psi + inverse\ \psi = -sqrt\ 5$
by (*simp add*: *ψ-def field-simps*)
also have $(\varphi\ \hat{}\ (2 * n + 1) * sqrt\ 5 + \psi\ \hat{}\ (2 * n + 1) * -\ sqrt\ 5)\ /\ 5 = (\varphi\ \hat{}\ (2 * n + 1) - \psi\ \hat{}\ (2 * n + 1)) * (sqrt\ 5\ /\ 5)$
by (*simp add*: *field-simps*)
also have $sqrt\ 5\ /\ 5 = inverse\ (sqrt\ 5)$
by (*simp add*: *field-simps*)
also have $(\varphi\ \hat{}\ (2 * n + 1) - \psi\ \hat{}\ (2 * n + 1)) * \ldots = of\text{-}nat\ (fib\ (Suc\ (2 * n)))$
by (*simp add*: *fib-closed-form*[*folded φ-def ψ-def*] *divide-inverse*)
finally show *?thesis*
by (*simp only*: *of-nat-eq-iff*)

qed

lemma *fib-rec-even*: *fib* (*2* ∗ *n*) = (*fib* (*n* − *1*) + *fib* (*n* + *1*)) ∗ *fib n*
proof (*induct n*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc n*)
  let *?rfib* = λ*x*. *real* (*fib x*)
  have *2* ∗ (*Suc n*) = *Suc* (*Suc* (*2* ∗ *n*)) **by** *simp*
  also have *real* (*fib* ...) = *?rfib n*^*2* + *?rfib* (*Suc n*)^*2* + (*?rfib* (*n* − *1*) + *?rfib* (*n* + *1*)) ∗ *?rfib n*
    by (*simp add*: *fib-rec-odd Suc*)
  also have (*?rfib* (*n* − *1*) + *?rfib* (*n* + *1*)) ∗ *?rfib n* = (*2* ∗ *?rfib* (*n* + *1*) − *?rfib n*) ∗ *?rfib n*
    by (*cases n*) *simp-all*
  also have *?rfib n*^*2* + *?rfib* (*Suc n*)^*2* + ... = (*?rfib* (*Suc n*) + *2* ∗ *?rfib n*) ∗ *?rfib* (*Suc n*)
    by (*simp add*: *algebra-simps power2-eq-square*)
  also have ... = *real* ((*fib* (*Suc n* − *1*) + *fib* (*Suc n* + *1*)) ∗ *fib* (*Suc n*)) **by** *simp*
  finally show *?case* **by** (*simp only*: *of-nat-eq-iff*)
qed

lemma *fib-rec-even′*: *fib* (*2* ∗ *n*) = (*2* ∗ *fib* (*n* − *1*) + *fib n*) ∗ *fib n*
  by (*subst fib-rec-even*, *cases n*) *simp-all*

lemma *fib-rec*:
  fib n =
    (*if n* = *0 then 0 else if n* = *1 then 1*
     else if even n then let n′ = *n div 2*; *fn* = *fib n′ in* (*2* ∗ *fib* (*n′* − *1*) + *fn*) ∗ *fn*
     else let n′ = *n div 2 in fib n′* ^ *2* + *fib* (*Suc n′*) ^ *2*)
  by (*auto elim*: *evenE oddE simp*: *fib-rec-odd fib-rec-even′ Let-def*)

1.8 Fibonacci and Binomial Coefficients

lemma *sum-drop-zero*: (∑ *k* = *0*..*Suc n*. *if 0*<*k then* (*f* (*k* − *1*)) *else 0*) = (∑ *j* = *0*..*n*. *f j*)
  by (*induct n*) *auto*

lemma *sum-choose-drop-zero*:
  (∑ *k* = *0*..*Suc n*. *if k* = *0 then 0 else* (*Suc n* − *k*) *choose* (*k* − *1*)) =
    (∑ *j* = *0*..*n*. (*n*−*j*) *choose j*)
  by (*rule trans* [*OF sum.cong sum-drop-zero*]) *auto*

lemma *ne-diagonal-fib*: (∑ *k* = *0*..*n*. (*n*−*k*) *choose k*) = *fib* (*Suc n*)
proof (*induct n rule*: *fib.induct*)
  case *1*
  show *?case* **by** *simp*

next
case *2*
show *?case* **by** *simp*
next
case (*3 n*)
have ($\sum k = 0..Suc\ n.\ Suc\ (Suc\ n) - k\ choose\ k$) =
($\sum k = 0..Suc\ n.\ (Suc\ n - k\ choose\ k) + (if\ k = 0\ then\ 0\ else\ (Suc\ n - k\ choose\ (k - 1))))$
by (*rule sum.cong*) (*simp-all add*: *choose-reduce-nat*)
also have ... =
($\sum k = 0..Suc\ n.\ Suc\ n - k\ choose\ k$) +
($\sum k = 0..Suc\ n.\ if\ k=0\ then\ 0\ else\ (Suc\ n - k\ choose\ (k - 1))$)
by (*simp add*: *sum.distrib*)
also have ... = ($\sum k = 0..Suc\ n.\ Suc\ n - k\ choose\ k$) + ($\sum j = 0..n.\ n - j\ choose\ j$)
by (*metis sum-choose-drop-zero*)
finally show *?case* **using** *3*
by *simp*
qed

end

2 Congruence

theory *Cong*
imports *HOL−Computational-Algebra.Primes*
begin

2.1 Generic congruences

context *unique-euclidean-semiring*
begin

definition *cong* :: *'a* ⇒ *'a* ⇒ *'a* ⇒ *bool*
(‹(‹*indent=1 notation=‹mixfix cong›*›[- = -] *'(' mod -')*)›)
where [*b* = *c*] (*mod a*) ⟷ *b mod a* = *c mod a*

abbreviation *notcong* :: *'a* ⇒ *'a* ⇒ *'a* ⇒ *bool*
(‹(‹*indent=1 notation=‹mixfix notcong›*›[- ≠ -] *'(' mod -')*)›)
where [*b* ≠ *c*] (*mod a*) ≡ ¬ *cong b c a*

lemma *cong-refl* [*simp*]:
[*b* = *b*] (*mod a*)
by (*simp add*: *cong-def*)

lemma *cong-sym*:
[*b* = *c*] (*mod a*) ⟹ [*c* = *b*] (*mod a*)
by (*simp add*: *cong-def*)

lemma *cong-sym-eq*:
[$b = c$] (*mod* a) $\longleftrightarrow$ [$c = b$] (*mod* a)
by (*auto simp add*: *cong-def*)

lemma *cong-trans* [*trans*]:
[$b = c$] (*mod* a) $\Longrightarrow$ [$c = d$] (*mod* a) $\Longrightarrow$ [$b = d$] (*mod* a)
by (*simp add*: *cong-def*)

lemma *cong-mult-self-right*:
[$b * a = 0$] (*mod* a)
by (*simp add*: *cong-def*)

lemma *cong-mult-self-left*:
[$a * b = 0$] (*mod* a)
by (*simp add*: *cong-def*)

lemma *cong-mod-left* [*simp*]:
[b *mod* $a = c$] (*mod* a) $\longleftrightarrow$ [$b = c$] (*mod* a)
by (*simp add*: *cong-def*)

lemma *cong-mod-right* [*simp*]:
[$b = c$ *mod* a] (*mod* a) $\longleftrightarrow$ [$b = c$] (*mod* a)
by (*simp add*: *cong-def*)

lemma *cong-0* [*simp*, *presburger*]:
[$b = c$] (*mod* 0) $\longleftrightarrow$ $b = c$
by (*simp add*: *cong-def*)

lemma *cong-1* [*simp*, *presburger*]:
[$b = c$] (*mod* 1)
by (*simp add*: *cong-def*)

lemma *cong-dvd-iff*:
a *dvd* $b \longleftrightarrow$ a *dvd* c **if** [$b = c$] (*mod* a)
using *that* **by** (*auto simp*: *cong-def dvd-eq-mod-eq-0*)

lemma *cong-0-iff*: [$b = 0$] (*mod* a) $\longleftrightarrow$ a *dvd* b
by (*simp add*: *cong-def dvd-eq-mod-eq-0*)

lemma *cong-add*:
[$b = c$] (*mod* a) $\Longrightarrow$ [$d = e$] (*mod* a) $\Longrightarrow$ [$b + d = c + e$] (*mod* a)
by (*auto simp add*: *cong-def intro*: *mod-add-cong*)

lemma *cong-mult*:
[$b = c$] (*mod* a) $\Longrightarrow$ [$d = e$] (*mod* a) $\Longrightarrow$ [$b * d = c * e$] (*mod* a)
by (*auto simp add*: *cong-def intro*: *mod-mult-cong*)

lemma *cong-scalar-right*:
[$b = c$] (*mod* a) $\Longrightarrow$ [$b * d = c * d$] (*mod* a)

by (*simp add*: *cong-mult*)

lemma *cong-scalar-left*:
[*b* = *c*] (*mod a*) ⟹ [*d* * *b* = *d* * *c*] (*mod a*)
by (*simp add*: *cong-mult*)

lemma *cong-pow*:
[*b* = *c*] (*mod a*) ⟹ [*b* ^ *n* = *c* ^ *n*] (*mod a*)
by (*simp add*: *cong-def power-mod* [*symmetric*, *of b n a*] *power-mod* [*symmetric*, *of c n a*])

lemma *cong-sum*:
[*sum f A* = *sum g A*] (*mod a*) **if** ⋀*x*. *x* ∈ *A* ⟹ [*f x* = *g x*] (*mod a*)
using *that* **by** (*induct A rule*: *infinite-finite-induct*) (*auto intro*: *cong-add*)

lemma *cong-prod*:
[*prod f A* = *prod g A*] (*mod a*) **if** (⋀*x*. *x* ∈ *A* ⟹ [*f x* = *g x*] (*mod a*))
using *that* **by** (*induct A rule*: *infinite-finite-induct*) (*auto intro*: *cong-mult*)

lemma *mod-mult-cong-right*:
[*c mod* (*a* * *b*) = *d*] (*mod a*) ⟷ [*c* = *d*] (*mod a*)
by (*simp add*: *cong-def mod-mod-cancel mod-add-left-eq*)

lemma *mod-mult-cong-left*:
[*c mod* (*b* * *a*) = *d*] (*mod a*) ⟷ [*c* = *d*] (*mod a*)
using *mod-mult-cong-right* [*of c a b d*] **by** (*simp add*: *ac-simps*)

lemma *cong-mod-leftI* [*simp*]:
[*b* = *c*] (*mod a*) ⟹ [*b mod a* = *c*] (*mod a*)
by (*simp add*: *cong-def*)

lemma *cong-mod-rightI* [*simp*]:
[*b* = *c*] (*mod a*) ⟹ [*b* = *c mod a*] (*mod a*)
by (*simp add*: *cong-def*)

lemma *cong-cmult-leftI*: [*a* = *b*] (*mod m*) ⟹ [*c* * *a* = *c* * *b*] (*mod* (*c* * *m*))
by (*metis cong-def local.mult-mod-right*)

lemma *cong-cmult-rightI*: [*a* = *b*] (*mod m*) ⟹ [*a* * *c* = *b* * *c*] (*mod* (*m* * *c*))
using *cong-cmult-leftI*[*of a b m c*] **by** (*simp add*: *mult.commute*)

lemma *cong-dvd-mono-modulus*:
assumes [*a* = *b*] (*mod m*) *m*′ *dvd m*
shows [*a* = *b*] (*mod m*′)
using *assms* **by** (*metis cong-def local.mod-mod-cancel*)

lemma *coprime-cong-transfer-left*:
assumes *coprime a b* [*a* = *a*′] (*mod b*)
shows *coprime a*′ *b*

using *assms* **by** (*metis cong-0 cong-def local.coprime-mod-left-iff*)

lemma *coprime-cong-transfer-right*:
assumes *coprime a b* [*b* = *b'*] (*mod a*)
shows *coprime a b'*
using *coprime-cong-transfer-left*[*of b a b'*] *assms*
by (*simp add*: *coprime-commute*)

lemma *coprime-cong-cong-left*:
assumes [*a* = *a'*] (*mod b*)
shows *coprime a b* ⟷ *coprime a' b*
using *assms cong-sym-eq coprime-cong-transfer-left* **by** *blast*

lemma *coprime-cong-cong-right*:
assumes [*b* = *b'*] (*mod a*)
shows *coprime a b* ⟷ *coprime a b'*
using *coprime-cong-cong-left*[*OF assms*] **by** (*simp add*: *coprime-commute*)

end

context *unique-euclidean-ring*
begin

lemma *cong-diff*:
[*b* = *c*] (*mod a*) ⟹ [*d* = *e*] (*mod a*) ⟹ [*b* − *d* = *c* − *e*] (*mod a*)
by (*auto simp add*: *cong-def intro*: *mod-diff-cong*)

lemma *cong-diff-iff-cong-0*:
[*b* − *c* = *0*] (*mod a*) ⟷ [*b* = *c*] (*mod a*) (**is** *?P* ⟷ *?Q*)
proof
assume *?P*
then have [*b* − *c* + *c* = *0* + *c*] (*mod a*)
by (*rule cong-add*) *simp*
then show *?Q*
by *simp*
next
assume *?Q*
with *cong-diff* [*of b c a c c*] **show** *?P*
by *simp*
qed

lemma *cong-minus-minus-iff*:
[− *b* = − *c*] (*mod a*) ⟷ [*b* = *c*] (*mod a*)
using *cong-diff-iff-cong-0* [*of b c a*] *cong-diff-iff-cong-0* [*of* − *b* − *c a*]
by (*simp add*: *cong-0-iff dvd-diff-commute*)

lemma *cong-modulus-minus-iff* [*iff*]:
[*b* = *c*] (*mod* − *a*) ⟷ [*b* = *c*] (*mod a*)
using *cong-diff-iff-cong-0* [*of b c a*] *cong-diff-iff-cong-0* [*of b c* −*a*]

by (*simp add*: *cong-0-iff*)

lemma *cong-iff-dvd-diff*:
[*a* = *b*] (*mod m*) ⟷ *m dvd* (*a* − *b*)
by (*simp add*: *cong-0-iff* [*symmetric*] *cong-diff-iff-cong-0*)

lemma *cong-iff-lin*:
[*a* = *b*] (*mod m*) ⟷ (∃ *k*. *b* = *a* + *m* ∗ *k*) (**is** *?P* ⟷ *?Q*)
proof −
have *?P* ⟷ *m dvd b* − *a*
by (*simp add*: *cong-iff-dvd-diff dvd-diff-commute*)
also have . . . ⟷ *?Q*
by (*auto simp add*: *algebra-simps elim*!: *dvdE*)
finally show *?thesis*
by *simp*
qed

lemma *cong-add-lcancel*:
[*a* + *x* = *a* + *y*] (*mod n*) ⟷ [*x* = *y*] (*mod n*)
by (*simp add*: *cong-iff-lin algebra-simps*)

lemma *cong-add-rcancel*:
[*x* + *a* = *y* + *a*] (*mod n*) ⟷ [*x* = *y*] (*mod n*)
by (*simp add*: *cong-iff-lin algebra-simps*)

lemma *cong-add-lcancel-0*:
[*a* + *x* = *a*] (*mod n*) ⟷ [*x* = *0*] (*mod n*)
using *cong-add-lcancel* [*of a x 0 n*] **by** *simp*

lemma *cong-add-rcancel-0*:
[*x* + *a* = *a*] (*mod n*) ⟷ [*x* = *0*] (*mod n*)
using *cong-add-rcancel* [*of x a 0 n*] **by** *simp*

lemma *cong-dvd-modulus*:
[*x* = *y*] (*mod n*) **if** [*x* = *y*] (*mod m*) **and** *n dvd m*
using *that* **by** (*auto intro*: *dvd-trans simp add*: *cong-iff-dvd-diff*)

lemma *cong-modulus-mult*:
[*x* = *y*] (*mod m*) **if** [*x* = *y*] (*mod m* ∗ *n*)
using *that* **by** (*simp add*: *cong-iff-dvd-diff*) (*rule dvd-mult-left*)

lemma *cong-uminus*: [*x* = *y*] (*mod m*) ⟹ [−*x* = −*y*] (*mod m*)
unfolding *cong-minus-minus-iff* **.**

end

lemma *cong-abs* [*simp*]:
[*x* = *y*] (*mod* |*m*|) ⟷ [*x* = *y*] (*mod m*)
for *x y* :: *'a* :: {*unique-euclidean-ring*, *linordered-idom*}

by (*simp add*: *cong-iff-dvd-diff*)

lemma *cong-square*:
prime p $\Longrightarrow$ *0* $<$ *a* $\Longrightarrow$ [*a* $*$ *a* $=$ *1*] (*mod p*) $\Longrightarrow$ [*a* $=$ *1*] (*mod p*) $\vee$ [*a* $=$ $-$ *1*] (*mod p*)
for *a p* :: *'a* :: {*normalization-semidom*, *linordered-idom*, *unique-euclidean-ring*}
by (*auto simp add*: *cong-iff-dvd-diff square-diff-one-factored dest*: *prime-dvd-multD*)

lemma *cong-mult-rcancel*:
[*a* $*$ *k* $=$ *b* $*$ *k*] (*mod m*) $\longleftrightarrow$ [*a* $=$ *b*] (*mod m*)
if *coprime k m* **for** *a k m* :: *'a*::{*unique-euclidean-ring*, *ring-gcd*}
using *that* **by** (*auto simp add*: *cong-iff-dvd-diff left-diff-distrib* [*symmetric*] *ac-simps coprime-dvd-mult-right-iff*)

lemma *cong-mult-lcancel*:
[*k* $*$ *a* $=$ *k* $*$ *b*] (*mod m*) $=$ [*a* $=$ *b*] (*mod m*)
if *coprime k m* **for** *a k m* :: *'a*::{*unique-euclidean-ring*, *ring-gcd*}
using *that cong-mult-rcancel* [*of k m a b*] **by** (*simp add*: *ac-simps*)

lemma *coprime-cong-mult*:
[*a* $=$ *b*] (*mod m*) $\Longrightarrow$ [*a* $=$ *b*] (*mod n*) $\Longrightarrow$ *coprime m n* $\Longrightarrow$ [*a* $=$ *b*] (*mod m* $*$ *n*)
for *a b* :: *'a* :: {*unique-euclidean-ring*, *semiring-gcd*}
by (*simp add*: *cong-iff-dvd-diff divides-mult*)

lemma *cong-gcd-eq*:
gcd a m $=$ *gcd b m* **if** [*a* $=$ *b*] (*mod m*)
for *a b* :: *'a* :: {*unique-euclidean-semiring*, *euclidean-semiring-gcd*}
proof (*cases m* $=$ *0*)
case *True*
with *that* **show** *?thesis*
by *simp*
next
case *False*
moreover have *gcd* (*a mod m*) *m* $=$ *gcd* (*b mod m*) *m*
using *that* **by** (*simp add*: *cong-def*)
ultimately show *?thesis*
by *simp*
qed

lemma *cong-imp-coprime*:
[*a* $=$ *b*] (*mod m*) $\Longrightarrow$ *coprime a m* $\Longrightarrow$ *coprime b m*
for *a b* :: *'a* :: {*unique-euclidean-semiring*, *euclidean-semiring-gcd*}
by (*auto simp add*: *coprime-iff-gcd-eq-1 dest*: *cong-gcd-eq*)

lemma *cong-cong-prod-coprime*:
[*x* $=$ *y*] (*mod* ($\prod$ *i*$\in$*A*. *m i*)) **if**
($\forall$ *i*$\in$*A*. [*x* $=$ *y*] (*mod m i*))
($\forall$ *i*$\in$*A*. ($\forall$ *j*$\in$*A*. *i* $\neq$ *j* $\longrightarrow$ *coprime* (*m i*) (*m j*)))
for *x y* :: *'a* :: {*unique-euclidean-ring*, *semiring-gcd*}

using *that* **by** (*induct A rule*: *infinite-finite-induct*)
(*auto intro*!: *coprime-cong-mult prod-coprime-right*)

2.2 Congruences on *nat* and *int*

lemma *cong-int-iff*:
[*int m* = *int q*] (*mod int n*) ⟷ [*m* = *q*] (*mod n*)
by (*simp add*: *cong-def of-nat-mod* [*symmetric*])

lemma *cong-Suc-0* [*simp*, *presburger*]:
[*m* = *n*] (*mod Suc 0*)
using *cong-1* [*of m n*] **by** *simp*

lemma *cong-diff-nat*:
[*a* − *c* = *b* − *d*] (*mod m*) **if** [*a* = *b*] (*mod m*) [*c* = *d*] (*mod m*)
and *a* ≥ *c b* ≥ *d* **for** *a b c d m* :: *nat*
proof −
have [*c* + (*a* − *c*) = *d* + (*b* − *d*)] (*mod m*)
using *that* **by** *simp*
with ‹[*c* = *d*] (*mod m*)› **have** [*c* + (*a* − *c*) = *c* + (*b* − *d*)] (*mod m*)
using *mod-add-cong* **by** (*auto simp add*: *cong-def*) *fastforce*
then show *?thesis*
by (*simp add*: *cong-def nat-mod-eq-iff*)
qed

lemma *cong-diff-iff-cong-0-nat*:
[*a* − *b* = *0*] (*mod m*) ⟷ [*a* = *b*] (*mod m*) **if** *a* ≥ *b* **for** *a b* :: *nat*
using *that* **by** (*simp add*: *cong-0-iff*) (*simp add*: *cong-def mod-eq-dvd-iff-nat*)

lemma *cong-diff-iff-cong-0-nat′*:
[*nat* |*int a* − *int b*| = *0*] (*mod m*) ⟷ [*a* = *b*] (*mod m*)
proof (*cases b* ≤ *a*)
case *True*
then show *?thesis*
by (*simp add*: *nat-diff-distrib′ cong-diff-iff-cong-0-nat* [*of b a m*])
next
case *False*
then have *a* ≤ *b*
by *simp*
then show *?thesis*
by (*simp add*: *nat-diff-distrib′ cong-diff-iff-cong-0-nat* [*of a b m*])
(*auto simp add*: *cong-def*)
qed

lemma *cong-altdef-nat*:
a ≥ *b* ⟹ [*a* = *b*] (*mod m*) ⟷ *m dvd* (*a* − *b*)
for *a b* :: *nat*
by (*simp add*: *cong-0-iff* [*symmetric*] *cong-diff-iff-cong-0-nat*)

lemma *cong-altdef-nat'*:
  $[a = b]\ (mod\ m) \longleftrightarrow m\ dvd\ nat\ |int\ a - int\ b|$
  using *cong-diff-iff-cong-0-nat'* [*of a b m*]
  by (*simp only*: *cong-0-iff* [*symmetric*])

lemma *cong-mult-rcancel-nat*:
  $[a * k = b * k]\ (mod\ m) \longleftrightarrow [a = b]\ (mod\ m)$
  if *coprime k m* **for** *a k m* :: *nat*
proof −
  have $[a * k = b * k]\ (mod\ m) \longleftrightarrow m\ dvd\ nat\ |int\ (a * k) - int\ (b * k)|$
    by (*simp add*: *cong-altdef-nat'*)
  also have ... $\longleftrightarrow m\ dvd\ nat\ |(int\ a - int\ b) * int\ k|$
    by (*simp add*: *algebra-simps*)
  also have ... $\longleftrightarrow m\ dvd\ nat\ |int\ a - int\ b| * k$
    by (*simp add*: *abs-mult nat-times-as-int*)
  also have ... $\longleftrightarrow m\ dvd\ nat\ |int\ a - int\ b|$
    by (*rule coprime-dvd-mult-left-iff*) (*use* ‹*coprime k m*› **in** ‹*simp add*: *ac-simps*›)
  also have ... $\longleftrightarrow [a = b]\ (mod\ m)$
    by (*simp add*: *cong-altdef-nat'*)
  finally show *?thesis* **.**
qed

lemma *cong-mult-lcancel-nat*:
  $[k * a = k * b]\ (mod\ m) = [a = b]\ (mod\ m)$
  if *coprime k m* **for** *a k m* :: *nat*
  using *that* **by** (*simp add*: *cong-mult-rcancel-nat ac-simps*)

lemma *coprime-cong-mult-nat*:
  $[a = b]\ (mod\ m) \Longrightarrow [a = b]\ (mod\ n) \Longrightarrow coprime\ m\ n \Longrightarrow [a = b]\ (mod\ m * n)$
  for *a b* :: *nat*
  by (*simp add*: *cong-altdef-nat' divides-mult*)

lemma *cong-less-imp-eq-nat*: $0 \leq a \Longrightarrow a < m \Longrightarrow 0 \leq b \Longrightarrow b < m \Longrightarrow [a = b]\ (mod\ m) \Longrightarrow a = b$
  for *a b* :: *nat*
  by (*auto simp add*: *cong-def*)

lemma *cong-less-imp-eq-int*: $0 \leq a \Longrightarrow a < m \Longrightarrow 0 \leq b \Longrightarrow b < m \Longrightarrow [a = b]\ (mod\ m) \Longrightarrow a = b$
  for *a b* :: *int*
  by (*auto simp add*: *cong-def*)

lemma *cong-less-unique-nat*: $0 < m \Longrightarrow (\exists!b.\ 0 \leq b \wedge b < m \wedge [a = b]\ (mod\ m))$
  for *a m* :: *nat*
  by (*auto simp*: *cong-def*) (*metis mod-mod-trivial mod-less-divisor*)

lemma *cong-less-unique-int*: $0 < m \Longrightarrow (\exists!b.\ 0 \leq b \wedge b < m \wedge [a = b]\ (mod\ m))$
  for *a m* :: *int*

by (*auto simp add*: *cong-def*) (*metis mod-mod-trivial pos-mod-bound pos-mod-sign*)

lemma *cong-iff-lin-nat*: $[a = b]\ (mod\ m) \longleftrightarrow (\exists\, k1\ k2.\ b + k1 * m = a + k2 * m)$
for *a b* :: *nat*
apply (*auto simp add*: *cong-def nat-mod-eq-iff*)
apply (*metis mult.commute*)
apply (*metis mult.commute*)
done

lemma *cong-cong-mod-nat*: $[a = b]\ (mod\ m) \longleftrightarrow [a\ mod\ m = b\ mod\ m]\ (mod\ m)$
for *a b* :: *nat*
by *simp*

lemma *cong-cong-mod-int*: $[a = b]\ (mod\ m) \longleftrightarrow [a\ mod\ m = b\ mod\ m]\ (mod\ m)$
for *a b* :: *int*
by *simp*

lemma *cong-add-lcancel-nat*: $[a + x = a + y]\ (mod\ n) \longleftrightarrow [x = y]\ (mod\ n)$
for *a x y* :: *nat*
by (*simp add*: *cong-iff-lin-nat*)

lemma *cong-add-rcancel-nat*: $[x + a = y + a]\ (mod\ n) \longleftrightarrow [x = y]\ (mod\ n)$
for *a x y* :: *nat*
by (*simp add*: *cong-iff-lin-nat*)

lemma *cong-add-lcancel-0-nat*: $[a + x = a]\ (mod\ n) \longleftrightarrow [x = 0]\ (mod\ n)$
for *a x* :: *nat*
using *cong-add-lcancel-nat* [*of a x 0 n*] **by** *simp*

lemma *cong-add-rcancel-0-nat*: $[x + a = a]\ (mod\ n) \longleftrightarrow [x = 0]\ (mod\ n)$
for *a x* :: *nat*
using *cong-add-rcancel-nat* [*of x a 0 n*] **by** *simp*

lemma *cong-dvd-modulus-nat*: $[x = y]\ (mod\ m) \Longrightarrow n\ dvd\ m \Longrightarrow [x = y]\ (mod\ n)$
for *x y* :: *nat*
by (*auto simp add*: *cong-altdef-nat'*)

lemma *cong-to-1-nat*:
fixes *a* :: *nat*
assumes $[a = 1]\ (mod\ n)$
shows *n dvd* (*a* − *1*)
proof (*cases a = 0*)
case *True*
then show *?thesis* **by** *force*
next
case *False*
with *assms* **show** *?thesis* **by** (*metis cong-altdef-nat leI less-one*)
qed

lemma *cong-0-1-nat'*: $[0 = Suc\ 0]\ (mod\ n) \longleftrightarrow n = Suc\ 0$
by (*auto simp*: *cong-def*)

lemma *cong-0-1-nat*: $[0 = 1]\ (mod\ n) \longleftrightarrow n = 1$
for n :: *nat*
by (*auto simp*: *cong-def*)

lemma *cong-0-1-int*: $[0 = 1]\ (mod\ n) \longleftrightarrow n = 1 \vee n = -\ 1$
for n :: *int*
by (*auto simp*: *cong-def zmult-eq-1-iff*)

lemma *cong-to-1'-nat*: $[a = 1]\ (mod\ n) \longleftrightarrow a = 0 \wedge n = 1 \vee (\exists m.\ a = 1 + m * n)$
for a :: *nat*
by (*metis add.right-neutral cong-0-1-nat cong-iff-lin-nat cong-to-1-nat dvd-div-mult-self leI le-add-diff-inverse less-one mult-eq-if*)

lemma *cong-le-nat*: $y \leq x \Longrightarrow [x = y]\ (mod\ n) \longleftrightarrow (\exists q.\ x = q * n + y)$
for $x\ y$ :: *nat*
by (*auto simp add*: *cong-altdef-nat le-imp-diff-is-add*)

lemma *cong-solve-nat*:
fixes a :: *nat*
shows $\exists x.\ [a * x = gcd\ a\ n]\ (mod\ n)$
proof (*cases* $a = 0 \vee n = 0$)
case *True*
then show *?thesis*
by (*force simp add*: *cong-0-iff cong-sym*)
next
case *False*
then show *?thesis*
using *bezout-nat* [*of a n*]
by *auto* (*metis cong-add-rcancel-0-nat cong-mult-self-left*)
qed

lemma *cong-solve-int*:
fixes a :: *int*
shows $\exists x.\ [a * x = gcd\ a\ n]\ (mod\ n)$
by (*metis bezout-int cong-iff-lin mult.commute*)

lemma *cong-solve-dvd-nat*:
fixes a :: *nat*
assumes *gcd a n dvd d*
shows $\exists x.\ [a * x = d]\ (mod\ n)$
proof −
from *cong-solve-nat* [*of a*] **obtain** x **where** $[a * x = gcd\ a\ n](mod\ n)$
by *auto*
then have $[(d\ div\ gcd\ a\ n) * (a * x) = (d\ div\ gcd\ a\ n) * gcd\ a\ n]\ (mod\ n)$

```
    using cong-scalar-left by blast
  also from assms have (d div gcd a n) * gcd a n = d
    by (rule dvd-div-mult-self)
  also have (d div gcd a n) * (a * x) = a * (d div gcd a n * x)
    by auto
  finally show ?thesis
    by auto
qed

lemma cong-solve-dvd-int:
  fixes a::int
  assumes b: gcd a n dvd d
  shows ∃x. [a * x = d] (mod n)
proof −
  from cong-solve-int [of a] obtain x where [a * x = gcd a n](mod n)
    by auto
  then have [(d div gcd a n) * (a * x) = (d div gcd a n) * gcd a n] (mod n)
    using cong-scalar-left by blast
  also from b have (d div gcd a n) * gcd a n = d
    by (rule dvd-div-mult-self)
  also have (d div gcd a n) * (a * x) = a * (d div gcd a n * x)
    by auto
  finally show ?thesis
    by auto
qed

lemma cong-solve-coprime-nat:
  ∃x. [a * x = Suc 0] (mod n) if coprime a n
  using that cong-solve-nat [of a n] by auto

lemma cong-solve-coprime-int:
  ∃x. [a * x = 1] (mod n) if coprime a n for a n x :: int
  using that cong-solve-int [of a n] by (auto simp add: zabs-def split: if-splits)

lemma coprime-iff-invertible-nat:
  coprime a m ⟷ (∃x. [a * x = Suc 0] (mod m)) (is ?P ⟷ ?Q)
proof
  assume ?P then show ?Q
    by (auto dest!: cong-solve-coprime-nat)
next
  assume ?Q
  then obtain b where [a * b = Suc 0] (mod m)
    by blast
  with coprime-mod-left-iff [of m a * b] show ?P
    by (cases m = 0 ∨ m = 1)
      (unfold cong-def, auto simp add: cong-def)
qed

lemma coprime-iff-invertible-int:
```

coprime a m ⟷ (∃ *x*. [*a* ∗ *x* = *1*] (*mod m*)) (**is** *?P* ⟷ *?Q*) **for** *m* :: *int*
proof
assume *?P* **then show** *?Q*
by (*auto dest*: *cong-solve-coprime-int*)
next
assume *?Q*
then obtain *b* **where** [*a* ∗ *b* = *1*] (*mod m*)
by *blast*
with *coprime-mod-left-iff* [*of m a* ∗ *b*] **show** *?P*
by (*cases m* = *0* ∨ *m* = *1*)
(*unfold cong-def*, *auto simp add*: *zmult-eq-1-iff*)
qed

lemma *coprime-iff-invertible′-nat*:
assumes *m* > *0*
shows *coprime a m* ⟷ (∃ *x*. *0* ≤ *x* ∧ *x* < *m* ∧ [*a* ∗ *x* = *Suc 0*] (*mod m*))
proof −
have ⋀*b*. ⟦*0* < *m*; [*a* ∗ *b* = *Suc 0*] (*mod m*)⟧ ⟹ ∃ *b′*<*m*. [*a* ∗ *b′* = *Suc 0*] (*mod m*)
by (*metis cong-def mod-less-divisor* [*OF assms*] *mod-mult-right-eq*)
then show *?thesis*
using *assms coprime-iff-invertible-nat* **by** *auto*
qed

lemma *coprime-iff-invertible′-int*:
fixes *m* :: *int*
assumes *m* > *0*
shows *coprime a m* ⟷ (∃ *x*. *0* ≤ *x* ∧ *x* < *m* ∧ [*a* ∗ *x* = *1*] (*mod m*))
using *assms* **by** (*simp add*: *coprime-iff-invertible-int*)
(*metis assms cong-mod-left mod-mult-right-eq pos-mod-bound pos-mod-sign*)

lemma *cong-cong-lcm-nat*: [*x* = *y*] (*mod a*) ⟹ [*x* = *y*] (*mod b*) ⟹ [*x* = *y*] (*mod lcm a b*)
for *x y* :: *nat*
by (*meson cong-altdef-nat′ lcm-least*)

lemma *cong-cong-lcm-int*: [*x* = *y*] (*mod a*) ⟹ [*x* = *y*] (*mod b*) ⟹ [*x* = *y*] (*mod lcm a b*)
for *x y* :: *int*
by (*auto simp add*: *cong-iff-dvd-diff lcm-least*)

lemma *cong-cong-prod-coprime-nat*:
[*x* = *y*] (*mod* (∏ *i*∈*A*. *m i*)) **if**
(∀ *i*∈*A*. [*x* = *y*] (*mod m i*))
(∀ *i*∈*A*. (∀ *j*∈*A*. *i* ≠ *j* ⟶ *coprime* (*m i*) (*m j*)))
for *x y* :: *nat*
using *that* **by** (*induct A rule*: *infinite-finite-induct*)
(*auto intro*!: *coprime-cong-mult-nat prod-coprime-right*)

lemma *binary-chinese-remainder-nat*:
fixes *m1 m2* :: *nat*
assumes *a*: *coprime m1 m2*
shows ∃ *x*. [*x* = *u1*] (*mod m1*) ∧ [*x* = *u2*] (*mod m2*)
proof −
have ∃ *b1 b2*. [*b1* = *1*] (*mod m1*) ∧ [*b1* = *0*] (*mod m2*) ∧ [*b2* = *0*] (*mod m1*) ∧ [*b2* = *1*] (*mod m2*)
proof −
from *cong-solve-coprime-nat* [*OF a*] **obtain** *x1* **where** *1*: [*m1* ∗ *x1* = *1*] (*mod m2*)
by *auto*
from *a* **have** *b*: *coprime m2 m1*
by (*simp add*: *ac-simps*)
from *cong-solve-coprime-nat* [*OF b*] **obtain** *x2* **where** *2*: [*m2* ∗ *x2* = *1*] (*mod m1*)
by *auto*
have [*m1* ∗ *x1* = *0*] (*mod m1*)
by (*simp add*: *cong-mult-self-left*)
moreover have [*m2* ∗ *x2* = *0*] (*mod m2*)
by (*simp add*: *cong-mult-self-left*)
ultimately show *?thesis*
using *1 2* **by** *blast*
qed
then obtain *b1 b2*
where [*b1* = *1*] (*mod m1*) **and** [*b1* = *0*] (*mod m2*)
and [*b2* = *0*] (*mod m1*) **and** [*b2* = *1*] (*mod m2*)
by *blast*
let *?x* = *u1* ∗ *b1* + *u2* ∗ *b2*
have [*?x* = *u1* ∗ *1* + *u2* ∗ *0*] (*mod m1*)
using ‹[*b1* = *1*] (*mod m1*)› ‹[*b2* = *0*] (*mod m1*)› *cong-add cong-scalar-left* **by** *blast*
then have [*?x* = *u1*] (*mod m1*) **by** *simp*
have [*?x* = *u1* ∗ *0* + *u2* ∗ *1*] (*mod m2*)
using ‹[*b1* = *0*] (*mod m2*)› ‹[*b2* = *1*] (*mod m2*)› *cong-add cong-scalar-left* **by** *blast*
then have [*?x* = *u2*] (*mod m2*)
by *simp*
with ‹[*?x* = *u1*] (*mod m1*)› **show** *?thesis*
by *blast*
qed

lemma *binary-chinese-remainder-int*:
fixes *m1 m2* :: *int*
assumes *a*: *coprime m1 m2*
shows ∃ *x*. [*x* = *u1*] (*mod m1*) ∧ [*x* = *u2*] (*mod m2*)
proof −
have ∃ *b1 b2*. [*b1* = *1*] (*mod m1*) ∧ [*b1* = *0*] (*mod m2*) ∧ [*b2* = *0*] (*mod m1*) ∧ [*b2* = *1*] (*mod m2*)
proof −

from *cong-solve-coprime-int* [*OF a*] **obtain** *x1* **where** *1*: [*m1* ∗ *x1* = *1*] (*mod m2*)
by *auto*
from *a* **have** *b*: *coprime m2 m1*
by (*simp add*: *ac-simps*)
from *cong-solve-coprime-int* [*OF b*] **obtain** *x2* **where** *2*: [*m2* ∗ *x2* = *1*] (*mod m1*)
by *auto*
have [*m1* ∗ *x1* = *0*] (*mod m1*)
by (*simp add*: *cong-mult-self-left*)
moreover have [*m2* ∗ *x2* = *0*] (*mod m2*)
by (*simp add*: *cong-mult-self-left*)
ultimately show *?thesis*
using *1 2* **by** *blast*
qed
then obtain *b1 b2*
where [*b1* = *1*] (*mod m1*) **and** [*b1* = *0*] (*mod m2*)
and [*b2* = *0*] (*mod m1*) **and** [*b2* = *1*] (*mod m2*)
by *blast*
let *?x* = *u1* ∗ *b1* + *u2* ∗ *b2*
have [*?x* = *u1* ∗ *1* + *u2* ∗ *0*] (*mod m1*)
using ‹[*b1* = *1*] (*mod m1*)› ‹[*b2* = *0*] (*mod m1*)› *cong-add cong-scalar-left* **by** *blast*
then have [*?x* = *u1*] (*mod m1*) **by** *simp*
have [*?x* = *u1* ∗ *0* + *u2* ∗ *1*] (*mod m2*)
using ‹[*b1* = *0*] (*mod m2*)› ‹[*b2* = *1*] (*mod m2*)› *cong-add cong-scalar-left* **by** *blast*
then have [*?x* = *u2*] (*mod m2*) **by** *simp*
with ‹[*?x* = *u1*] (*mod m1*)› **show** *?thesis*
by *blast*
qed

lemma *cong-modulus-mult-nat*: [*x* = *y*] (*mod m* ∗ *n*) ⟹ [*x* = *y*] (*mod m*)
for *x y* :: *nat*
by (*metis cong-def mod-mult-cong-right*)

lemma *cong-less-modulus-unique-nat*: [*x* = *y*] (*mod m*) ⟹ *x* < *m* ⟹ *y* < *m* ⟹ *x* = *y*
for *x y* :: *nat*
by (*simp add*: *cong-def*)

lemma *binary-chinese-remainder-unique-nat*:
fixes *m1 m2* :: *nat*
assumes *a*: *coprime m1 m2*
and *nz*: *m1* ≠ *0 m2* ≠ *0*
shows ∃!*x*. *x* < *m1* ∗ *m2* ∧ [*x* = *u1*] (*mod m1*) ∧ [*x* = *u2*] (*mod m2*)
proof −
obtain *y* **where** *y1*: [*y* = *u1*] (*mod m1*) **and** *y2*: [*y* = *u2*] (*mod m2*)
using *binary-chinese-remainder-nat* [*OF a*] **by** *blast*

```
  let ?x = y mod (m1 * m2)
  from nz have less: ?x < m1 * m2
    by auto
  have 1: [?x = u1] (mod m1)
    using y1 mod-mult-cong-right by blast
  have 2: [?x = u2] (mod m2)
    using y2 mod-mult-cong-left by blast
  have z = ?x if z < m1 * m2 [z = u1] (mod m1)  [z = u2] (mod m2) for z
  proof −
    have [?x = z] (mod m1)
      by (metis 1 cong-def that(2))
    moreover have [?x = z] (mod m2)
      by (metis 2 cong-def that(3))
    ultimately have [?x = z] (mod m1 * m2)
      using a by (auto intro: coprime-cong-mult-nat simp add: mod-mult-cong-left
mod-mult-cong-right)
    with ‹z < m1 * m2› ‹?x < m1 * m2› show z = ?x
      by (auto simp add: cong-def)
  qed
  with less 1 2 show ?thesis
    by blast
 qed

lemma chinese-remainder-nat:
  fixes A :: 'a set
    and m :: 'a ⇒ nat
    and u :: 'a ⇒ nat
  assumes fin: finite A
    and cop: ∀ i ∈ A. ∀ j ∈ A. i ≠ j ⟶ coprime (m i) (m j)
  shows ∃ x. ∀ i ∈ A. [x = u i] (mod m i)
proof −
  have ∃ b. (∀ i ∈ A. [b i = 1] (mod m i) ∧ [b i = 0] (mod (∏ j ∈ A − {i}. m j)))
  proof (rule finite-set-choice, rule fin, rule ballI)
    fix i
    assume i ∈ A
    with cop have coprime (∏ j ∈ A − {i}. m j) (m i)
      by (intro prod-coprime-left) auto
    then have ∃ x. [(∏ j ∈ A − {i}. m j) * x = Suc 0] (mod m i)
      by (elim cong-solve-coprime-nat)
    then obtain x where [(∏ j ∈ A − {i}. m j) * x = 1] (mod m i)
      by auto
    moreover have [(∏ j ∈ A − {i}. m j) * x = 0] (mod (∏ j ∈ A − {i}. m j))
      by (simp add: cong-0-iff)
    ultimately show ∃ a. [a = 1] (mod m i) ∧ [a = 0] (mod prod m (A − {i}))
      by blast
  qed
  then obtain b where b: ⋀i. i ∈ A ⟹ [b i = 1] (mod m i) ∧ [b i = 0] (mod
(∏ j ∈ A − {i}. m j))
    by blast
```

```
  let ?x = ∑ i∈A. (u i) * (b i)
  show ?thesis
  proof (rule exI, clarify)
    fix i
    assume a: i ∈ A
    show [?x = u i] (mod m i)
    proof −
      from fin a have ?x = (∑ j ∈ {i}. u j * b j) + (∑ j ∈ A − {i}. u j * b j)
        by (subst sum.union-disjoint [symmetric]) (auto intro: sum.cong)
      then have [?x = u i * b i + (∑ j ∈ A − {i}. u j * b j)] (mod m i)
        by auto
      also have [u i * b i + (∑ j ∈ A − {i}. u j * b j) =
                 u i * 1 + (∑ j ∈ A − {i}. u j * 0)] (mod m i)
      proof (intro cong-add cong-scalar-left cong-sum)
        show [b i = 1] (mod m i)
          using a b by blast
        show [b x = 0] (mod m i) if x ∈ A − {i} for x
        proof −
          have x ∈ A x ≠ i
            using that by auto
          then show ?thesis
            using a b [OF ‹x ∈ A›] cong-dvd-modulus-nat fin by blast
        qed
      qed
      finally show ?thesis
        by simp
    qed
  qed
qed

lemma coprime-cong-prod-nat: [x = y] (mod (∏ i∈A. m i))
  if ⋀i j. ⟦i ∈ A; j ∈ A; i ≠ j⟧ ⟹ coprime (m i) (m j)
    and ⋀i. i ∈ A ⟹ [x = y] (mod m i) for x y :: nat
  using that
proof (induct A rule: infinite-finite-induct)
  case (insert x A)
  then show ?case
    by simp (metis coprime-cong-mult-nat prod-coprime-right)
qed auto

lemma chinese-remainder-unique-nat:
  fixes A :: 'a set
    and m :: 'a ⇒ nat
    and u :: 'a ⇒ nat
  assumes fin: finite A
    and nz: ∀ i∈A. m i ≠ 0
    and cop: ∀ i∈A. ∀ j∈A. i ≠ j ⟶ coprime (m i) (m j)
  shows ∃!x. x < (∏ i∈A. m i) ∧ (∀ i∈A. [x = u i] (mod m i))
proof −
```

```
  from chinese-remainder-nat [OF fin cop]
  obtain y where one: (∀ i∈A. [y = u i] (mod m i))
    by blast
  let ?x = y mod (∏ i∈A. m i)
  from fin nz have prodnz: (∏ i∈A. m i) ≠ 0
    by auto
  then have less: ?x < (∏ i∈A. m i)
    by auto
  have cong: ∀ i∈A. [?x = u i] (mod m i)
    using fin one
    by (auto simp add: cong-def dvd-prod-eqI mod-mod-cancel)
  have unique: ∀ z. z < (∏ i∈A. m i) ∧ (∀ i∈A. [z = u i] (mod m i)) ⟶ z = ?x
  proof clarify
    fix z
    assume zless: z < (∏ i∈A. m i)
    assume zcong: (∀ i∈A. [z = u i] (mod m i))
    have ∀ i∈A. [?x = z] (mod m i)
      using cong zcong by (auto simp add: cong-def)
    with fin cop have [?x = z] (mod (∏ i∈A. m i))
      by (intro coprime-cong-prod-nat) auto
    with zless less show z = ?x
      by (auto simp add: cong-def)
  qed
  from less cong unique show ?thesis
    by blast
qed

lemma (in semiring-1-cancel) of-nat-eq-iff-cong-CHAR:
  of-nat x = (of-nat y :: 'a) ⟷ [x = y] (mod CHAR('a))
proof (induction x y rule: linorder-wlog)
  case (le x y)
  define z where z = y − x
  have [simp]: y = x + z
    using le by (auto simp: z-def)
  have (CHAR('a) dvd z) = [x = x + z] (mod CHAR('a))
    by (metis ‹y = x + z› cong-def le mod-eq-dvd-iff-nat z-def)
  thus ?case
    by (simp add: of-nat-eq-0-iff-char-dvd)
qed (simp add: eq-commute cong-sym-eq)

lemma (in ring-1) of-int-eq-iff-cong-CHAR:
  of-int x = (of-int y :: 'a) ⟷ [x = y] (mod int CHAR('a))
proof −
  have of-int x = (of-int y :: 'a) ⟷ of-int (x − y) = (0 :: 'a)
    by auto
  also have ... ⟷ (int CHAR('a) dvd x − y)
    by (rule of-int-eq-0-iff-char-dvd)
  also have ... ⟷ [x = y] (mod int CHAR('a))
    by (simp add: cong-iff-dvd-diff)
```

```
  finally show ?thesis .
qed
```

Thanks to Manuel Eberl

```
lemma prime-cong-4-nat-cases [consumes 1, case-names 2 cong-1 cong-3]:
  assumes prime (p :: nat)
  obtains p = 2 | [p = 1] (mod 4) | [p = 3] (mod 4)
proof −
  have [p = 2] (mod 4) ⟷ p = 2
  proof
    assume [p = 2] (mod 4)
    hence p mod 4 = 2
      by (auto simp: cong-def)
    hence even p
      by (simp add: even-even-mod-4-iff)
    with assms show p = 2
      unfolding prime-nat-iff by force
  qed auto
  moreover have [p ≠ 0] (mod 4)
  proof
    assume [p = 0] (mod 4)
    hence 4 dvd p
      by (auto simp: cong-0-iff)
    with assms have p = 4
      by (subst (asm) prime-nat-iff) auto
    thus False
      using assms by simp
  qed
  ultimately consider [p = 3] (mod 4) | [p = 1] (mod 4) | p = 2
    by (fastforce simp: cong-def)
  thus ?thesis
    using that by metis
qed

end
```

3 Fundamental facts about Euler's totient function

```
theory Totient
imports
  Complex-Main
  HOL−Computational-Algebra.Primes
  Cong
begin

definition totatives :: nat ⇒ nat set where
  totatives n = {k ∈ {0<..n}. coprime k n}
```

lemma *in-totatives-iff*: $k \in totatives\ n \longleftrightarrow k > 0 \wedge k \leq n \wedge coprime\ k\ n$
by (*simp add*: *totatives-def*)

lemma *totatives-code* [*code*]: $totatives\ n = Set.filter\ (\lambda k.\ coprime\ k\ n)\ \{0<..n\}$
by (*simp add*: *totatives-def*)

lemma *finite-totatives* [*simp*]: *finite* (*totatives n*)
by (*simp add*: *totatives-def*)

lemma *totatives-subset*: $totatives\ n \subseteq \{0<..n\}$
by (*auto simp*: *totatives-def*)

lemma *zero-not-in-totatives* [*simp*]: $0 \notin totatives\ n$
by (*auto simp*: *totatives-def*)

lemma *totatives-le*: $x \in totatives\ n \Longrightarrow x \leq n$
by (*auto simp*: *totatives-def*)

lemma *totatives-less*:
assumes $x \in totatives\ n\ n > 1$
shows $x < n$
proof −
from *assms* **have** $x \neq n$ **by** (*auto simp*: *totatives-def*)
with *totatives-le*[*OF assms*(*1*)] **show** *?thesis* **by** *simp*
qed

lemma *totatives-0* [*simp*]: $totatives\ 0 = \{\}$
by (*auto simp*: *totatives-def*)

lemma *totatives-1* [*simp*]: $totatives\ 1 = \{Suc\ 0\}$
by (*auto simp*: *totatives-def*)

lemma *totatives-Suc-0* [*simp*]: $totatives\ (Suc\ 0) = \{Suc\ 0\}$
by (*auto simp*: *totatives-def*)

lemma *one-in-totatives* [*simp*]: $n > 0 \Longrightarrow Suc\ 0 \in totatives\ n$
by (*auto simp*: *totatives-def*)

lemma *totatives-eq-empty-iff* [*simp*]: $totatives\ n = \{\} \longleftrightarrow n = 0$
using *one-in-totatives*[*of n*] **by** (*auto simp del*: *one-in-totatives*)

lemma *minus-one-in-totatives*:
assumes $n \geq 2$
shows $n - 1 \in totatives\ n$
using *assms coprime-diff-one-left-nat* [*of n*] **by** (*simp add*: *in-totatives-iff*)

lemma *power-in-totatives*:
assumes $m > 1\ coprime\ m\ g$

```
  shows   g ^ i mod m ∈ totatives m
proof −
  have ¬m dvd g ^ i
  proof
    assume m dvd g ^ i
    hence ¬coprime m (g ^ i)
      using ‹m > 1› by (subst coprime-absorb-left) auto
    with ‹coprime m g› show False by simp
  qed
  with assms show ?thesis
    by (auto simp: totatives-def coprime-commute intro!: Nat.gr0I)
qed

lemma totatives-prime-power-Suc:
  assumes prime p
  shows   totatives (p ^ Suc n) = {0<..p^Suc n} − (λm. p * m) ` {0<..p^n}
proof safe
  fix m assume m: p * m ∈ totatives (p ^ Suc n) and m: m ∈ {0<..p^n}
  thus False using assms by (auto simp: totatives-def gcd-mult-left)
next
  fix k assume k: k ∈ {0<..p^Suc n} k ∉ (λm. p * m) ` {0<..p^n}
  from k have ¬(p dvd k) by (auto elim!: dvdE)
  hence coprime k (p ^ Suc n)
    using prime-imp-coprime [OF assms, of k]
    by (cases n > 0) (auto simp add: ac-simps)
  with k show k ∈ totatives (p ^ Suc n) by (simp add: totatives-def)
qed (auto simp: totatives-def)

lemma totatives-prime: prime p ⟹ totatives p = {0<..<p}
  using totatives-prime-power-Suc [of p 0] by auto

lemma bij-betw-totatives:
  assumes m1 > 1 m2 > 1 coprime m1 m2
  shows   bij-betw (λx. (x mod m1, x mod m2)) (totatives (m1 * m2))
            (totatives m1 × totatives m2)
  unfolding bij-betw-def
proof
  show inj-on (λx. (x mod m1, x mod m2)) (totatives (m1 * m2))
  proof (intro inj-onI, clarify)
    fix x y assume xy: x ∈ totatives (m1 * m2) y ∈ totatives (m1 * m2)
                       x mod m1 = y mod m1 x mod m2 = y mod m2
    have ex: ∃!z. z < m1 * m2 ∧ [z = x] (mod m1) ∧ [z = x] (mod m2)
      by (rule binary-chinese-remainder-unique-nat) (insert assms, simp-all)
    have x < m1 * m2 ∧ [x = x] (mod m1) ∧ [x = x] (mod m2)
         y < m1 * m2 ∧ [y = x] (mod m1) ∧ [y = x] (mod m2)
      using xy assms by (simp-all add: totatives-less one-less-mult cong-def)
    from this[THEN the1-equality[OF ex]] show x = y by simp
  qed
next
```

show (λx. (x *mod* m1, x *mod* m2)) ‘ totatives (m1 ∗ m2) = totatives m1 × totatives m2
proof *safe*
fix x **assume** x ∈ totatives (m1 ∗ m2)
with *assms* **show** x *mod* m1 ∈ totatives m1 x *mod* m2 ∈ totatives m2
using *coprime-common-divisor* [*of* x m1 m1] *coprime-common-divisor* [*of* x m2 m2]
by (*auto simp add*: *in-totatives-iff mod-greater-zero-iff-not-dvd*)
next
fix a b **assume** *ab*: a ∈ totatives m1 b ∈ totatives m2
with *assms* **have** *ab'*: a < m1 b < m2 **by** (*auto simp*: *totatives-less*)
with *binary-chinese-remainder-unique-nat*[*OF assms*(3), *of* a b] **obtain** x
where x: x < m1 ∗ m2 x *mod* m1 = a x *mod* m2 = b **by** (*auto simp*: *cong-def*)
from x *ab assms*(3) **have** x ∈ totatives (m1 ∗ m2)
by (*auto intro*: *ccontr simp add*: *in-totatives-iff*)
with x **show** (a, b) ∈ (λx. (x *mod* m1, x *mod* m2)) ‘ totatives (m1∗m2) **by** *blast*
qed
qed

lemma *bij-betw-totatives-gcd-eq*:
fixes n d :: *nat*
assumes d *dvd* n n > 0
shows *bij-betw* (λk. k ∗ d) (totatives (n *div* d)) {k∈{0<..n}. gcd k n = d}
unfolding *bij-betw-def*
proof
show *inj-on* (λk. k ∗ d) (totatives (n *div* d))
by (*auto simp*: *inj-on-def*)
next
show (λk. k ∗ d) ‘ totatives (n *div* d) = {k∈{0<..n}. gcd k n = d}
proof (*intro equalityI subsetI*, *goal-cases*)
case (1 k)
then show *?case* **using** *assms*
by (*auto elim*: *dvdE simp add*: *in-totatives-iff ac-simps gcd-mult-right*)
next
case (2 k)
hence d *dvd* k **by** *auto*
then obtain l **where** k: k = l ∗ d **by** (*elim dvdE*) *auto*
from 2 *assms* **show** *?case*
using *gcd-mult-right* [*of* - d l]
by (*auto intro*: *gcd-eq-1-imp-coprime elim*!: *dvdE simp add*: k *image-iff in-totatives-iff ac-simps*)
qed
qed

definition *totient* :: *nat* ⇒ *nat* **where**
totient n = *card* (totatives n)

primrec *totient-naive* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**

```
  totient-naive 0 acc n = acc
| totient-naive (Suc k) acc n =
    (if coprime (Suc k) n then totient-naive k (acc + 1) n else totient-naive k acc
n)

lemma totient-naive:
  totient-naive k acc n = card {x ∈ {0<..k}. coprime x n} + acc
proof (induction k arbitrary: acc)
  case (Suc k acc)
  have totient-naive (Suc k) acc n =
        (if coprime (Suc k) n then 1 else 0) + card {x ∈ {0<..k}. coprime x n}
+ acc
    using Suc by simp
  also have (if coprime (Suc k) n then 1 else 0) =
             card (if coprime (Suc k) n then {Suc k} else {}) by auto
  also have ... + card {x ∈ {0<..k}. coprime x n} =
              card ((if coprime (Suc k) n then {Suc k} else {}) ∪ {x ∈ {0<..k}.
coprime x n})
    by (intro card-Un-disjoint [symmetric]) auto
  also have ((if coprime (Suc k) n then {Suc k} else {}) ∪ {x ∈ {0<..k}. coprime
x n}) =
             {x ∈ {0<..Suc k}. coprime x n} by (auto elim: le-SucE)
  finally show ?case .
qed simp-all

lemma totient-code-naive [code]: totient n = totient-naive n 0 n
  by (subst totient-naive) (simp add: totient-def totatives-def)

lemma totient-le: totient n ≤ n
proof −
  have card (totatives n) ≤ card {0<..n}
    by (intro card-mono) (auto simp: totatives-def)
  thus ?thesis by (simp add: totient-def)
qed

lemma totient-less:
  assumes n > 1
  shows totient n < n
proof −
  from assms have card (totatives n) ≤ card {0<..<n}
    using totatives-less[of - n] totatives-subset[of n] by (intro card-mono) auto
  with assms show ?thesis by (simp add: totient-def)
qed

lemma totient-0 [simp]: totient 0 = 0
  by (simp add: totient-def)

lemma totient-Suc-0 [simp]: totient (Suc 0) = Suc 0
  by (simp add: totient-def)
```

lemma *totient-1* [*simp*]: *totient 1 = Suc 0*
by *simp*

lemma *totient-0-iff* [*simp*]: *totient n = 0 ⟷ n = 0*
by (*auto simp*: *totient-def*)

lemma *totient-gt-0-iff* [*simp*]: *totient n > 0 ⟷ n > 0*
by (*auto intro*: *Nat.gr0I*)

lemma *totient-gt-1*:
assumes *n > 2*
shows *totient n > 1*
proof −
have {*1*, *n − 1*} ⊆ *totatives n*
using *assms coprime-diff-one-left-nat*[*of n*] **by** (*auto simp*: *totatives-def*)
hence *card* {*1*, *n − 1*} ≤ *card* (*totatives n*)
by (*intro card-mono*) *auto*
thus *?thesis* **using** *assms*
by (*simp add*: *totient-def*)
qed

lemma *card-gcd-eq-totient*:
n > 0 ⟹ *d dvd n* ⟹ *card* {*k*∈{*0*<..*n*}. *gcd k n = d*} = *totient* (*n div d*)
unfolding *totient-def* **by** (*rule sym*, *rule bij-betw-same-card*[*OF bij-betw-totatives-gcd-eq*])

lemma *totient-divisor-sum*: (∑ *d* | *d dvd n*. *totient d*) = *n*
proof (*cases n = 0*)
case *False*
hence *n > 0* **by** *simp*
define *A* **where** *A* = (λ*d*. {*k*∈{*0*<..*n*}. *gcd k n = d*})
have ∗: *card* (*A d*) = *totient* (*n div d*) **if** *d*: *d dvd n* **for** *d*
using ‹*n > 0*› **and** *d* **unfolding** *A-def* **by** (*rule card-gcd-eq-totient*)
have *n* = *card* {*1*..*n*} **by** *simp*
also have {*1*..*n*} = (⋃ *d*∈{*d*. *d dvd n*}. *A d*) **by** *safe* (*auto simp*: *A-def*)
also have *card* ... = (∑ *d* | *d dvd n*. *card* (*A d*))
using ‹*n > 0*› **by** (*intro card-UN-disjoint*) (*auto simp*: *A-def*)
also have ... = (∑ *d* | *d dvd n*. *totient* (*n div d*)) **by** (*intro sum.cong refl* ∗)
auto
also have ... = (∑ *d* | *d dvd n*. *totient d*) **using** ‹*n > 0*›
by (*intro sum.reindex-bij-witness*[*of* - (*div*) *n* (*div*) *n*]) (*auto elim*: *dvdE*)
finally show *?thesis* **..**
qed *auto*

lemma *totient-mult-coprime*:
assumes *coprime m n*
shows *totient* (*m* ∗ *n*) = *totient m* ∗ *totient n*
proof (*cases m > 1* ∧ *n > 1*)
case *True*

```
  hence mn: m > 1 n > 1 by simp-all
  have totient (m * n) = card (totatives (m * n)) by (simp add: totient-def)
  also have ... = card (totatives m × totatives n)
    using bij-betw-totatives [OF mn ‹coprime m n›] by (rule bij-betw-same-card)
  also have ... = totient m * totient n by (simp add: totient-def)
  finally show ?thesis .
next
  case False
  with assms show ?thesis by (cases m; cases n) auto
qed

lemma totient-prime-power-Suc:
  assumes prime p
  shows   totient (p ^ Suc n) = p ^ n * (p - 1)
proof -
  from assms have totient (p ^ Suc n) = card ({0<..p ^ Suc n} - (*) p ` {0<..p
^ n})
    unfolding totient-def by (subst totatives-prime-power-Suc) simp-all
  also from assms have ... = p ^ Suc n - card ((*) p ` {0<..p^n})
    by (subst card-Diff-subset) (auto intro: prime-gt-0-nat)
  also from assms have card ((*) p ` {0<..p^n}) = p ^ n
    by (subst card-image) (auto simp: inj-on-def)
  also have p ^ Suc n - p ^ n = p ^ n * (p - 1) by (simp add: algebra-simps)
  finally show ?thesis .
qed

lemma totient-prime-power:
  assumes prime p n > 0
  shows   totient (p ^ n) = p ^ (n - 1) * (p - 1)
  using totient-prime-power-Suc[of p n - 1] assms by simp

lemma totient-imp-prime:
  assumes totient p = p - 1 p > 0
  shows   prime p
proof (cases p = 1)
  case True
  with assms show ?thesis by auto
next
  case False
  with assms have p: p > 1 by simp
  have x ∈ {0<..<p} if x ∈ totatives p for x
    using that and p by (cases x = p) (auto simp: totatives-def)
  with assms have *: totatives p = {0<..<p}
    by (intro card-subset-eq) (auto simp: totient-def)
  have **: False if x ≠ 1 x ≠ p x dvd p for x
  proof -
    from that have nz: x ≠ 0 by (auto intro!: Nat.gr0I)
    from that and p have le: x ≤ p by (intro dvd-imp-le) auto
    from that and nz have ¬coprime x p
```

```
    by (auto elim: dvdE)
   hence x ∉ totatives p by (simp add: totatives-def)
   also note *
   finally show False using that and le by auto
  qed
  hence (∀ m. m dvd p ⟶ m = 1 ∨ m = p) by blast
  with p show ?thesis by (subst prime-nat-iff) (auto dest: **)
qed

lemma totient-prime:
  assumes prime p
  shows   totient p = p - 1
  using totient-prime-power-Suc[of p 0] assms by simp

lemma totient-2 [simp]: totient 2 = 1
  and totient-3 [simp]: totient 3 = 2
  and totient-5 [simp]: totient 5 = 4
  and totient-7 [simp]: totient 7 = 6
  by (subst totient-prime; simp)+

lemma totient-4 [simp]: totient 4 = 2
  and totient-8 [simp]: totient 8 = 4
  and totient-9 [simp]: totient 9 = 6
 using totient-prime-power[of 2 2] totient-prime-power[of 2 3] totient-prime-power[of
3 2]
  by simp-all

lemma totient-6 [simp]: totient 6 = 2
  using totient-mult-coprime [of 2 3] coprime-add-one-right [of 2]
  by simp

lemma totient-even:
  assumes n > 2
  shows   even (totient n)
proof (cases ∃ p. prime p ∧ p ≠ 2 ∧ p dvd n)
  case True
  then obtain p where p: prime p p ≠ 2 p dvd n by auto
  from ‹p ≠ 2› have p = 0 ∨ p = 1 ∨ p > 2 by auto
  with p(1) have odd p using prime-odd-nat[of p] by auto
  define k where k = multiplicity p n
 from p assms have k-pos: k > 0 unfolding k-def by (subst multiplicity-gt-zero-iff)
auto
  have p ^ k dvd n unfolding k-def by (simp add: multiplicity-dvd)
  then obtain m where m: n = p ^ k * m by (elim dvdE)
  with assms have m-pos: m > 0 by (auto intro!: Nat.gr0I)
  from k-def m-pos p have ¬ p dvd m
   by (subst (asm) m) (auto intro!: Nat.gr0I simp: prime-elem-multiplicity-mult-distrib

                                prime-elem-multiplicity-eq-zero-iff)
```

```
  with ‹prime p› have coprime p m
    by (rule prime-imp-coprime)
  with ‹k > 0› have coprime (p ^ k) m
    by simp
  then show ?thesis using p k-pos ‹odd p›
    by (auto simp add: m totient-mult-coprime totient-prime-power)
next
  case False
  from assms have n = (∏p∈prime-factors n. p ^ multiplicity p n)
    by (intro Primes.prime-factorization-nat) auto
  also from False have ... = (∏p∈prime-factors n. if p = 2 then 2 ^ multiplicity
2 n else 1)
    by (intro prod.cong refl) auto
  also have ... = 2 ^ multiplicity 2 n
    by (subst prod.delta[OF finite-set-mset]) (auto simp: prime-factors-multiplicity)
  finally have n: n = 2 ^ multiplicity 2 n .
  have multiplicity 2 n = 0 ∨ multiplicity 2 n = 1 ∨ multiplicity 2 n > 1 by force
  with n assms have multiplicity 2 n > 1 by auto
  thus ?thesis by (subst n) (simp add: totient-prime-power)
qed

lemma totient-prod-coprime:
  assumes pairwise coprime (f ‘ A) inj-on f A
  shows   totient (prod f A) = (∏a∈A. totient (f a))
  using assms
proof (induction A rule: infinite-finite-induct)
  case (insert x A)
  have ∗: coprime (prod f A) (f x)
  proof (rule prod-coprime-left)
    fix y
    assume y ∈ A
    with ‹x ∉ A› have y ≠ x
      by auto
    with ‹x ∉ A› ‹y ∈ A› ‹inj-on f (insert x A)› have f y ≠ f x
      using inj-onD [of f insert x A y x]
      by auto
    with ‹y ∈ A› show coprime (f y) (f x)
      using pairwiseD [OF ‹pairwise coprime (f ‘ insert x A)›]
      by auto
  qed
  from insert.hyps have prod f (insert x A) = prod f A ∗ f x by simp
  also have totient ... = totient (prod f A) ∗ totient (f x)
    using insert.hyps insert.prems by (intro totient-mult-coprime ∗)
  also have totient (prod f A) = (∏a∈A. totient (f a))
    using insert.prems by (intro insert.IH) (auto dest: pairwise-subset)
  also from insert.hyps have ... ∗ totient (f x) = (∏a∈insert x A. totient (f a))
by simp
  finally show ?case .
qed simp-all
```

lemma *prime-power-eq-imp-eq*:
fixes *p q* :: *'a* :: *factorial-semiring*
assumes *prime p prime q m* > *0*
assumes *p* ^ *m* = *q* ^ *n*
shows *p* = *q*
proof (*rule ccontr*)
assume *pq*: *p* ≠ *q*
from *assms* **have** *m* = *multiplicity p* (*p* ^ *m*)
by (*subst multiplicity-prime-power*) *auto*
also note ‹*p* ^ *m* = *q* ^ *n*›
also from *assms pq* **have** *multiplicity p* (*q* ^ *n*) = *0*
by (*subst multiplicity-distinct-prime-power*) *auto*
finally show *False* **using** ‹*m* > *0*› **by** *simp*
qed

lemma *totient-formula1*:
assumes *n* > *0*
shows *totient n* = (∏ *p*∈*prime-factors n*. *p* ^ (*multiplicity p n* − *1*) ∗ (*p* − *1*))
proof −
from *assms* **have** *n* = (∏ *p*∈*prime-factors n*. *p* ^ *multiplicity p n*)
by (*rule prime-factorization-nat*)
also have *totient* ... = (∏ *x*∈*prime-factors n*. *totient* (*x* ^ *multiplicity x n*))
proof (*rule totient-prod-coprime*)
show *pairwise coprime* ((λ*p*. *p* ^ *multiplicity p n*) ‘ *prime-factors n*)
proof (*rule pairwiseI*, *clarify*)
fix *p q* **assume** ∗: *p* ∈# *prime-factorization n q* ∈# *prime-factorization n*
p ^ *multiplicity p n* ≠ *q* ^ *multiplicity q n*
then have *multiplicity p n* > *0 multiplicity q n* > *0*
by (*simp-all add*: *prime-factors-multiplicity*)
with ∗ *primes-coprime* [*of p q*] **show** *coprime* (*p* ^ *multiplicity p n*) (*q* ^ *multiplicity q n*)
by *auto*
qed
next
show *inj-on* (λ*p*. *p* ^ *multiplicity p n*) (*prime-factors n*)
proof
fix *p q* **assume** *pq*: *p* ∈# *prime-factorization n q* ∈# *prime-factorization n*
p ^ *multiplicity p n* = *q* ^ *multiplicity q n*
from *assms* **and** *pq* **have** *prime p prime q multiplicity p n* > *0*
by (*simp-all add*: *prime-factors-multiplicity*)
from *prime-power-eq-imp-eq*[*OF this pq*(*3*)] **show** *p* = *q* **.**
qed
qed
also have ... = (∏ *p*∈*prime-factors n*. *p* ^ (*multiplicity p n* − *1*) ∗ (*p* − *1*))
by (*intro prod.cong refl totient-prime-power*) (*auto simp*: *prime-factors-multiplicity*)
finally show *?thesis* **.**
qed

lemma *totient-dvd*:
assumes *m dvd n*
shows *totient m dvd totient n*
proof (*cases m = 0 ∨ n = 0*)
case *False*
let *?M = λp m :: nat. multiplicity p m − 1*
have (∏ *p∈prime-factors m. p ^ ?M p m ∗ (p − 1)*) *dvd*
(∏ *p∈prime-factors n. p ^ ?M p n ∗ (p − 1)*) **using** *assms False*
by (*intro prod-dvd-prod-subset2 mult-dvd-mono dvd-refl le-imp-power-dvd diff-le-mono*
dvd-prime-factors dvd-imp-multiplicity-le) *auto*
with *False* **show** *?thesis* **by** (*simp add*: *totient-formula1*)
qed (*insert assms*, *auto*)

lemma *totient-dvd-mono*:
assumes *m dvd n n > 0*
shows *totient m ≤ totient n*
by (*cases m = 0*) (*insert assms*, *auto intro*: *dvd-imp-le totient-dvd*)

lemma *prime-factors-power*: *n > 0 ⟹ prime-factors (x ^ n) = prime-factors x*
by (*cases x = 0*; *cases n = 0*)
(*auto simp*: *prime-factors-multiplicity prime-elem-multiplicity-power-distrib*
zero-power)

lemma *totient-formula2*:
real (*totient n*) = *real n ∗* (∏ *p∈prime-factors n. 1 − 1 / real p*)
proof (*cases n = 0*)
case *False*
have *real* (*totient n*) = (∏ *p∈prime-factors n. real*
(*p ^* (*multiplicity p n − 1*) *∗* (*p − 1*)))
using *False* **by** (*subst totient-formula1*) *simp-all*
also have *. . .* = (∏ *p∈prime-factors n. real* (*p ^ multiplicity p n*) *∗* (*1 − 1 /*
real p))
by (*intro prod.cong refl*) (*auto simp add*: *field-simps prime-factors-multiplicity*
prime-ge-Suc-0-nat of-nat-diff power-Suc [*symmetric*] *simp del*: *power-Suc*)
also have *. . .* = *real* (∏ *p∈prime-factors n. p ^ multiplicity p n*) *∗*
(∏ *p∈prime-factors n. 1 − 1 / real p*) **by** (*subst prod.distrib*) *auto*
also have (∏ *p∈prime-factors n. p ^ multiplicity p n*) = *n*
using *False* **by** (*intro Primes.prime-factorization-nat* [*symmetric*]) *auto*
finally show *?thesis* **.**
qed *auto*

lemma *totient-gcd*: *totient* (*a ∗ b*) *∗ totient* (*gcd a b*) = *totient a ∗ totient b ∗ gcd*
a b
proof (*cases a = 0 ∨ b = 0*)
case *False*
let *?P = prime-factors :: nat ⇒ nat set*
have *real* (*totient a ∗ totient b ∗ gcd a b*) = *real* (*a ∗ b ∗ gcd a b*) *∗*

$((\prod p{\in}?P\ a.\ 1 - 1\ /\ real\ p) * (\prod p{\in}?P\ b.\ 1 - 1\ /\ real\ p))$
by (*simp add*: *totient-formula2*)
also have $?P\ a = (?P\ a - ?P\ b) \cup (?P\ a \cap ?P\ b)$ **by** *auto*
also have $(\prod p{\in}\ldots.\ 1 - 1\ /\ real\ p) =$
$(\prod p{\in}?P\ a - ?P\ b.\ 1 - 1\ /\ real\ p) * (\prod p{\in}?P\ a \cap ?P\ b.\ 1 - 1\ /$
real p)
by (*rule prod.union-disjoint*) *blast+*
also have $\ldots * (\prod p{\in}?P\ b.\ 1 - 1\ /\ real\ p) = (\prod p{\in}?P\ a - ?P\ b.\ 1 - 1\ /\ real$
$p) *$
$(\prod p{\in}?P\ b.\ 1 - 1\ /\ real\ p) * (\prod p{\in}?P\ a \cap ?P\ b.\ 1 - 1\ /\ real\ p)$ (**is**
$- = ?A * -$)
by (*simp only*: *mult-ac*)
also have $?A = (\prod p{\in}?P\ a - ?P\ b \cup ?P\ b.\ 1 - 1\ /\ real\ p)$
by (*rule prod.union-disjoint* [*symmetric*]) *blast+*
also have $?P\ a - ?P\ b \cup ?P\ b = ?P\ a \cup ?P\ b$ **by** *blast*
also have $real\ (a * b * gcd\ a\ b) * ((\prod p{\in}\ldots.\ 1 - 1\ /\ real\ p) *$
$(\prod p{\in}?P\ a \cap ?P\ b.\ 1 - 1\ /\ real\ p)) = real\ (totient\ (a * b) * totient$
$(gcd\ a\ b))$
using *False* **by** (*simp add*: *totient-formula2 prime-factors-product prime-factorization-gcd*)
finally show *?thesis* **by** (*simp only*: *of-nat-eq-iff*)
qed *auto*

lemma *totient-mult*: $totient\ (a * b) = totient\ a * totient\ b * gcd\ a\ b\ div\ totient$
$(gcd\ a\ b)$
by (*subst totient-gcd* [*symmetric*]) *simp*

lemma *of-nat-eq-1-iff*: $of\text{-}nat\ x = (1 :: 'a :: \{semiring\text{-}1, semiring\text{-}char\text{-}0\}) \longleftrightarrow$
$x = 1$
by (*fact of-nat-eq-1-iff*)

lemma *odd-imp-coprime-nat*:
assumes *odd* (*n*::*nat*)
shows *coprime n 2*
proof −
from *assms* **obtain** *k* **where** *n*: $n = Suc\ (2 * k)$ **by** (*auto elim*!: *oddE*)
have $coprime\ (Suc\ (2 * k))\ (2 * k)$
by (*fact coprime-Suc-left-nat*)
then show *?thesis* **using** *n*
by *simp*
qed

lemma *totient-double*: $totient\ (2 * n) = (if\ even\ n\ then\ 2 * totient\ n\ else\ totient$
$n)$
by (*simp add*: *totient-mult ac-simps odd-imp-coprime-nat*)

lemma *totient-power-Suc*: $totient\ (n \mathbin{\hat{}} Suc\ m) = n \mathbin{\hat{}} m * totient\ n$
proof (*induction m arbitrary*: *n*)
case (*Suc m n*)

have *totient* $(n \mathbin{\hat{}} Suc\ (Suc\ m)) = totient\ (n * n \mathbin{\hat{}} Suc\ m)$ **by** *simp*
also have $\ldots = n \mathbin{\hat{}} Suc\ m * totient\ n$
using *Suc.IH* **by** (*subst totient-mult*) *simp*
finally show *?case* **.**
qed *simp-all*

lemma *totient-power*: $m > 0 \Longrightarrow totient\ (n \mathbin{\hat{}} m) = n \mathbin{\hat{}} (m - 1) * totient\ n$
using *totient-power-Suc*[*of n m − 1*] **by** (*cases m*) *simp-all*

lemma *totient-gcd-lcm*: $totient\ (gcd\ a\ b) * totient\ (lcm\ a\ b) = totient\ a * totient\ b$
proof ($a = 0 \vee b = 0$)
case *False*
let *?P* = *prime-factors* :: *nat* ⇒ *nat set* **and** *?f* = $\lambda p{::}nat.\ 1 - 1\ /\ real\ p$
have $real\ (totient\ (gcd\ a\ b) * totient\ (lcm\ a\ b)) = real\ (gcd\ a\ b * lcm\ a\ b) *$
$(prod\ ?f\ (?P\ a \cap ?P\ b) * prod\ ?f\ (?P\ a \cup ?P\ b))$
using *False* **unfolding** *of-nat-mult*
by (*simp add*: *totient-formula2 prime-factorization-gcd prime-factorization-lcm*)
also have $gcd\ a\ b * lcm\ a\ b = a * b$ **by** *simp*
also have $?P\ a \cup ?P\ b = (?P\ a - ?P\ a \cap ?P\ b) \cup ?P\ b$ **by** *blast*
also have $prod\ ?f\ \ldots = prod\ ?f\ (?P\ a - ?P\ a \cap ?P\ b) * prod\ ?f\ (?P\ b)$
by (*rule prod.union-disjoint*) *blast+*
also have $prod\ ?f\ (?P\ a \cap ?P\ b) * \ldots =$
$prod\ ?f\ (?P\ a \cap ?P\ b \cup (?P\ a - ?P\ a \cap ?P\ b)) * prod\ ?f\ (?P\ b)$
by (*subst prod.union-disjoint*) *auto*
also have $?P\ a \cap ?P\ b \cup (?P\ a - ?P\ a \cap ?P\ b) = ?P\ a$ **by** *blast*
also have $real\ (a * b) * (prod\ ?f\ (?P\ a) * prod\ ?f\ (?P\ b)) = real\ (totient\ a *$
totient b)
using *False* **by** (*simp add*: *totient-formula2*)
finally show *?thesis* **by** (*simp only*: *of-nat-eq-iff*)
qed *auto*

end

4 Residue rings

theory *Residues*
imports
Cong
HOL−Algebra.Multiplicative-Group
Totient
begin

lemma (**in** *ring-1*) *CHAR-dvd-CARD*: $CHAR('a)\ dvd\ card\ (UNIV :: 'a\ set)$
proof ($card\ (UNIV :: 'a\ set) = 0$)
case *False*
hence [*intro*]: $CHAR('a) > 0$
by (*simp add*: *card-eq-0-iff finite-imp-CHAR-pos*)
define *G* **where** $G = (\!|\ carrier = (UNIV :: 'a\ set),\ monoid.mult = (+),\ one = (0 :: 'a)\ |\!)$

define H **where** $H = (\mathit{of\text{-}nat}\ `\ \{..<\mathit{CHAR}('a)\} :: 'a\ \mathit{set})$
interpret *group* G
proof (*rule groupI*)
  fix x **assume** x: $x \in \mathit{carrier}\ G$
  show $\exists\, y{\in}\mathit{carrier}\ G.\ y \otimes_G x = \mathbf{1}_G$
    by (*intro bexI*[*of* - $-x$]) (*auto simp*: *G-def*)
qed (*auto simp*: *G-def add-ac*)

interpret *subgroup* H G
proof
  show $\mathbf{1}_G \in H$
    using *False* **unfolding** *G-def H-def* **by** *force*
next
  fix $x\ y :: 'a$
  assume $x \in H\ y \in H$
  then obtain $x'\ y'$ **where** [*simp*]: $x = \mathit{of\text{-}nat}\ x'\ y = \mathit{of\text{-}nat}\ y'$
    by (*auto simp*: *H-def*)
  have $x + y = \mathit{of\text{-}nat}\ ((x' + y')\ \mathit{mod}\ \mathit{CHAR}('a))$
    by (*auto simp flip*: *of-nat-add simp*: *of-nat-eq-iff-cong-CHAR*)
  moreover have $(x' + y')\ \mathit{mod}\ \mathit{CHAR}('a) < \mathit{CHAR}('a)$
    using *H-def* ‹$y \in H$› **by** *fastforce*
  ultimately show $x \otimes_G y \in H$
    by (*auto simp*: *H-def G-def intro*!: *imageI*)
next
  fix $x :: 'a$
  assume x: $x \in H$
  then obtain x' **where** [*simp*]: $x = \mathit{of\text{-}nat}\ x'$ **and** x': $x' < \mathit{CHAR}('a)$
    by (*auto simp*: *H-def*)
  have $\mathit{CHAR}('a)\ \mathit{dvd}\ x' + (\mathit{CHAR}('a) - x')\ \mathit{mod}\ \mathit{CHAR}('a)$
    using *mod-eq-0-iff-dvd mod-if* x' **by** *fastforce*
  hence $x + \mathit{of\text{-}nat}\ ((\mathit{CHAR}('a) - x')\ \mathit{mod}\ \mathit{CHAR}('a)) = 0$
    by (*auto simp flip*: *of-nat-add simp*: *of-nat-eq-0-iff-char-dvd*)
 moreover from *this* **have** $\mathit{inv}_G\ x = \mathit{of\text{-}nat}\ ((\mathit{CHAR}('a) - x')\ \mathit{mod}\ \mathit{CHAR}('a))$
    by (*intro inv-equality*) (*auto simp*: *G-def add-ac*)
  moreover have $\mathit{of\text{-}nat}\ ((\mathit{CHAR}('a) - x')\ \mathit{mod}\ \mathit{CHAR}('a)) \in H$
    unfolding *H-def* **using** ‹$\mathit{CHAR}('a) > 0$› **by** (*intro imageI*) *auto*
  ultimately show $\mathit{inv}_G\ x \in H$ **by** *force*
qed (*auto simp*: *G-def H-def*)

have *card H dvd card* $(\mathit{rcosets}_G\ H) * \mathit{card}\ H$
  by *simp*
also have $\mathit{card}\ (\mathit{rcosets}_G\ H) * \mathit{card}\ H = \mathit{Coset.order}\ G$
proof (*rule lagrange-finite*)
  show *finite* (*carrier G*)
    using *False card-ge-0-finite* **by** (*auto simp*: *G-def*)
qed (*fact is-subgroup*)
finally have *card H dvd card* $(\mathit{UNIV} :: 'a\ \mathit{set})$
  by (*simp add*: *Coset.order-def G-def*)
also have $\mathit{card}\ H = \mathit{card}\ \{..<\mathit{CHAR}('a)\}$

unfolding *H-def* **by** (*intro card-image inj-onI*) (*auto simp*: *of-nat-eq-iff-cong-CHAR cong-def*)
finally show *CHAR*('*a*) *dvd card* (*UNIV* :: '*a set*)
by *simp*
qed *auto*

definition *QuadRes* :: *int* ⇒ *int* ⇒ *bool*
where *QuadRes p a* = (∃ *y*. ([*y*^2 = *a*] (*mod p*)))

definition *Legendre* :: *int* ⇒ *int* ⇒ *int*
where *Legendre a p* =
(*if* ([*a* = *0*] (*mod p*)) *then 0*
else if QuadRes p a then 1
else −*1*)

4.1 A locale for residue rings

definition *residue-ring* :: *int* ⇒ *int ring*
where
residue-ring m =
(|*carrier* = {*0*..*m* − *1*},
monoid.mult = λ*x y*. (*x* ∗ *y*) *mod m*,
one = *1*,
zero = *0*,
add = λ*x y*. (*x* + *y*) *mod m*|)

locale *residues* =
fixes *m* :: *int* **and** *R* (**structure**)
assumes *m-gt-one*: *m* > *1*
defines *R-m-def*: *R* ≡ *residue-ring m*
begin

lemma *abelian-group*: *abelian-group R*
proof −
have ∃ *y*∈{*0*..*m* − *1*}. (*x* + *y*) *mod m* = *0* **if** *0* ≤ *x x* < *m* **for** *x*
proof (*cases x* = *0*)
case *True*
with *m-gt-one* **show** *?thesis* **by** *simp*
next
case *False*
then have (*x* + (*m* − *x*)) *mod m* = *0*
by *simp*
with *m-gt-one that* **show** *?thesis*
by (*metis False atLeastAtMost-iff diff-ge-0-iff-ge diff-left-mono int-one-le-iff-zero-less less-le*)
qed
with *m-gt-one* **show** *?thesis*
by (*fastforce simp add*: *R-m-def residue-ring-def mod-add-right-eq ac-simps intro*!: *abelian-groupI*)

qed

lemma *comm-monoid*: *comm-monoid R*
proof −
have $\bigwedge x\ y\ z.\ [\![x \in carrier\ R;\ y \in carrier\ R;\ z \in carrier\ R]\!] \Longrightarrow x \otimes y \otimes z = x \otimes (y \otimes z)$
$\bigwedge x\ y.\ [\![x \in carrier\ R;\ y \in carrier\ R]\!] \Longrightarrow x \otimes y = y \otimes x$
unfolding *R-m-def residue-ring-def*
by (*simp-all add*: *algebra-simps mod-mult-right-eq*)
then show *?thesis*
unfolding *R-m-def residue-ring-def*
by *unfold-locales* (*use m-gt-one* **in** *simp-all*)
qed

interpretation *comm-monoid R*
using *comm-monoid* **by** *blast*

lemma *cring*: *cring R*
apply (*intro cringI abelian-group comm-monoid*)
unfolding *R-m-def residue-ring-def*
apply (*auto simp add*: *comm-semiring-class.distrib mod-add-eq mod-mult-left-eq*)
done

end

sublocale *residues* < *cring*
by (*rule cring*)

context *residues*
begin

These lemmas translate back and forth between internal and external concepts.

lemma *res-carrier-eq*: $carrier\ R = \{0..m - 1\}$
by (*auto simp*: *R-m-def residue-ring-def*)

lemma *res-add-eq*: $x \oplus y = (x + y)\ mod\ m$
by (*auto simp*: *R-m-def residue-ring-def*)

lemma *res-mult-eq*: $x \otimes y = (x * y)\ mod\ m$
by (*auto simp*: *R-m-def residue-ring-def*)

lemma *res-zero-eq*: $\mathbf{0} = 0$
by (*auto simp*: *R-m-def residue-ring-def*)

lemma *res-one-eq*: $\mathbf{1} = 1$
by (*auto simp*: *R-m-def residue-ring-def units-of-def*)

lemma *res-units-eq*: *Units R* = {*x*. *0* < *x* ∧ *x* < *m* ∧ *coprime x m*} (**is** - = *?rhs*)
proof
  show *Units R* ⊆ *?rhs*
    using *zero-less-mult-iff invertible-coprime*
    by (*fastforce simp*: *Units-def R-m-def residue-ring-def*)
next
  show *?rhs* ⊆ *Units R*
    unfolding *Units-def R-m-def residue-ring-def*
    by (*force simp add*: *cong-def coprime-iff-invertible'-int mult.commute*)
qed

lemma *res-neg-eq*: ⊖ *x* = (− *x*) *mod m*
proof −
  have ⊖ *x* = (*THE y*. *0* ≤ *y* ∧ *y* < *m* ∧ (*x* + *y*) *mod m* = *0* ∧ (*y* + *x*) *mod m* = *0*)
    by (*simp add*: *R-m-def a-inv-def m-inv-def residue-ring-def*)
  also have ... = (− *x*) *mod m*
  proof −
    have ⋀*y*. *0* ≤ *y* ∧ *y* < *m* ∧ (*x* + *y*) *mod m* = *0* ∧ (*y* + *x*) *mod m* = *0* ⟹
        y = − *x mod m*
      by (*metis minus-add-cancel mod-add-eq plus-int-code*(*1*) *zmod-trivial-iff*)
    then show *?thesis*
    by (*intro the-equality*) (*use m-gt-one* **in** ‹*simp add*: *add.commute mod-add-right-eq*›)
  qed
  finally show *?thesis* **.**
qed

lemma *finite* [*iff*]: *finite* (*carrier R*)
  by (*simp add*: *res-carrier-eq*)

lemma *finite-Units* [*iff*]: *finite* (*Units R*)
  by (*simp add*: *finite-ring-finite-units*)

The function $a \mapsto a \bmod m$ maps the integers to the residue classes. The following lemmas show that this mapping respects addition and multiplication on the integers.

lemma *mod-in-carrier* [*iff*]: *a mod m* ∈ *carrier R*
  unfolding *res-carrier-eq*
  using *insert m-gt-one* **by** *auto*

lemma *add-cong*: (*x mod m*) ⊕ (*y mod m*) = (*x* + *y*) *mod m*
  by (*auto simp*: *R-m-def residue-ring-def mod-simps*)

lemma *mult-cong*: (*x mod m*) ⊗ (*y mod m*) = (*x* ∗ *y*) *mod m*
  by (*auto simp*: *R-m-def residue-ring-def mod-simps*)

lemma *zero-cong*: **0** = *0*
  by (*auto simp*: *R-m-def residue-ring-def*)

lemma *one-cong*: **1** = *1 mod m*
using *m-gt-one* **by** (*auto simp*: *R-m-def residue-ring-def*)

lemma *pow-cong*: (*x mod m*) [^] *n* = *x*^*n mod m*
using *m-gt-one*
proof (*induct n*)
case *0*
then show *?case*
by (*simp add*: *one-cong*)
next
case (*Suc n*)
then show *?case*
by (*simp add*: *mult-cong power-commutes*)
qed

lemma *neg-cong*: ⊖ (*x mod m*) = (− *x*) *mod m*
by (*metis mod-minus-eq res-neg-eq*)

lemma (**in** *residues*) *prod-cong*: *finite A* ⟹ (⨂ *i*∈*A*. (*f i*) *mod m*) = (∏ *i*∈*A*. *f i*) *mod m*
by (*induct set*: *finite*) (*auto simp*: *one-cong mult-cong*)

lemma (**in** *residues*) *sum-cong*: *finite A* ⟹ (⨁ *i*∈*A*. (*f i*) *mod m*) = (∑ *i*∈*A*. *f i*) *mod m*
by (*induct set*: *finite*) (*auto simp*: *zero-cong add-cong*)

lemma *mod-in-res-units* [*simp*]:
assumes *1* < *m* **and** *coprime a m*
shows *a mod m* ∈ *Units R*
proof (*cases a mod m* = *0*)
case *True*
with *assms* **show** *?thesis*
by (*auto simp add*: *res-units-eq gcd-red-int* [*symmetric*])
next
case *False*
from *assms* **have** *0* < *m* **by** *simp*
then have *0* ≤ *a mod m* **by** (*rule pos-mod-sign* [*of m a*])
with *False* **have** *0* < *a mod m* **by** *simp*
with *assms* **show** *?thesis*
by (*auto simp add*: *res-units-eq gcd-red-int* [*symmetric*] *ac-simps*)
qed

lemma *res-eq-to-cong*: (*a mod m*) = (*b mod m*) ⟷ [*a* = *b*] (*mod m*)
by (*auto simp*: *cong-def*)

Simplifying with these will translate a ring equation in R to a congruence.

lemmas *res-to-cong-simps* =
add-cong mult-cong pow-cong one-cong

prod-cong sum-cong neg-cong res-eq-to-cong

Other useful facts about the residue ring.

lemma *one-eq-neg-one*: $\mathbf{1} = \ominus \mathbf{1} \Longrightarrow m = 2$
using *one-cong res-neg-eq res-one-eq zmod-zminus1-eq-if* **by** *fastforce*

end

4.2 Prime residues

locale *residues-prime* =
fixes p :: *nat* **and** R (**structure**)
assumes *p-prime* [*intro*]: *prime p*
defines $R \equiv$ *residue-ring* (*int p*)

sublocale *residues-prime* < *residues p*
proof
show *1* < *int p*
using *prime-gt-1-nat* **by** *auto*
qed

context *residues-prime*
begin

lemma *p-coprime-left*:
coprime p a $\longleftrightarrow \neg$ *p dvd a*
using *p-prime* **by** (*auto intro*: *prime-imp-coprime dest*: *coprime-common-divisor*)

lemma *p-coprime-right*:
coprime a p $\longleftrightarrow \neg$ *p dvd a*
using *p-coprime-left* [*of a*] **by** (*simp add*: *ac-simps*)

lemma *p-coprime-left-int*:
coprime (*int p*) *a* $\longleftrightarrow \neg$ *int p dvd a*
using *p-prime* **by** (*auto intro*: *prime-imp-coprime dest*: *coprime-common-divisor*)

lemma *p-coprime-right-int*:
coprime a (*int p*) $\longleftrightarrow \neg$ *int p dvd a*
using *coprime-commute p-coprime-left-int* **by** *blast*

lemma *is-field*: *field R*
proof −
have *0* < *x* $\Longrightarrow$ *x* < *int p* $\Longrightarrow$ *coprime* (*int p*) *x* **for** *x*
by (*rule prime-imp-coprime*) (*auto simp add*: *zdvd-not-zless*)
then show *?thesis*
by (*intro cring.field-intro2 cring*)
(*auto simp add*: *res-carrier-eq res-one-eq res-zero-eq res-units-eq ac-simps*)
qed

lemma *res-prime-units-eq*: *Units R* = {*1..p* − *1*}
by (*auto simp add*: *res-units-eq p-coprime-right-int zdvd-not-zless*)

end

sublocale *residues-prime* < *field*
by (*rule is-field*)

5 Test cases: Euler's theorem and Wilson's theorem

5.1 Euler's theorem

lemma (**in** *residues*) *totatives-eq*:
totatives (*nat m*) = *nat* ' *Units R*
proof −
from *m-gt-one* **have** |*m*| > *1*
by *simp*
then have *totatives* (*nat* |*m*|) = *nat* ' *abs* ' *Units R*
by (*auto simp add*: *totatives-def res-units-eq image-iff le-less*)
(*use m-gt-one zless-nat-eq-int-zless* **in** *force*)
moreover have |*m*| = *m abs* ' *Units R* = *Units R*
using *m-gt-one* **by** (*auto simp add*: *res-units-eq image-iff*)
ultimately show *?thesis*
by *simp*
qed

lemma (**in** *residues*) *totient-eq*:
totient (*nat m*) = *card* (*Units R*)
proof −
have ∗: *inj-on nat* (*Units R*)
by (*rule inj-onI*) (*auto simp add*: *res-units-eq*)
then show *?thesis*
by (*simp add*: *totient-def totatives-eq card-image*)
qed

lemma (**in** *residues-prime*) *prime-totient-eq*: *totient p* = *p* − *1*
using *p-prime totient-prime* **by** *blast*

lemma (**in** *residues*) *euler-theorem*:
assumes *coprime a m*
shows [*a* ^ *totient* (*nat m*) = *1*] (*mod m*)
proof −
have *a* ^ *totient* (*nat m*) *mod m* = *1 mod m*
by (*metis assms finite-Units m-gt-one mod-in-res-units one-cong totient-eq pow-cong units-power-order-eq-one*)
then show *?thesis*
using *res-eq-to-cong* **by** *blast*
qed

lemma *euler-theorem*:
fixes *a m* :: *nat*
assumes *coprime a m*
shows [*a* ^ *totient m* = *1*] (*mod m*)
proof (*cases m* = *0* ∨ *m* = *1*)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
with *assms* **show** *?thesis*
using *residues.euler-theorem* [*of int m int a*] *cong-int-iff*
by (*auto simp add*: *residues-def gcd-int-def*) *fastforce*
qed

lemma *fermat-theorem*:
fixes *p a* :: *nat*
assumes *prime p* **and** ¬ *p dvd a*
shows [*a* ^ (*p* − *1*) = *1*] (*mod p*)
proof −
from *assms prime-imp-coprime* [*of p a*] **have** *coprime a p*
by (*auto simp add*: *ac-simps*)
then have [*a* ^ *totient p* = *1*] (*mod p*)
by (*rule euler-theorem*)
also have *totient p* = *p* − *1*
by (*rule totient-prime*) (*rule assms*)
finally show *?thesis* **.**
qed

5.2 Wilson's theorem

lemma (**in** *field*) *inv-pair-lemma*: *x* ∈ *Units R* ⟹ *y* ∈ *Units R* ⟹
{*x*, *inv x*} ≠ {*y*, *inv y*} ⟹ {*x*, *inv x*} ∩ {*y*, *inv y*} = {}
by *auto*

lemma (**in** *residues-prime*) *wilson-theorem1*:
assumes *a*: *p* > *2*
shows [*fact* (*p* − *1*) = (−*1*::*int*)] (*mod p*)
proof −
let *?Inverse-Pairs* = {{*x*, *inv x*}| *x*. *x* ∈ *Units R* − {**1**, ⊖ **1**}}
have *UR*: *Units R* = {**1**, ⊖ **1**} ∪ ⋃ *?Inverse-Pairs*
by *auto*
have *11*: **1** ≠ ⊖ **1**
using *a one-eq-neg-one* **by** *force*
have (⨂ *i*∈*Units R*. *i*) = (⨂ *i*∈{**1**, ⊖ **1**}. *i*) ⊗ (⨂ *i*∈⋃ *?Inverse-Pairs*. *i*)
apply (*subst UR*)
apply (*subst finprod-Un-disjoint*)
using *inv-one inv-eq-neg-one-eq* **apply** (*auto intro*!: *funcsetI*)+

```
    done
  also have (⨂ i∈{1, ⊖ 1}. i) = ⊖ 1
    by (simp add: 11)
  also have (⨂ i∈(⋃ ?Inverse-Pairs). i) = (⨂ A∈?Inverse-Pairs. (⨂ y∈A. y))
   by (rule finprod-Union-disjoint) (auto simp: pairwise-def disjnt-def dest!: inv-eq-imp-eq)
  also have ... = 1
    apply (rule finprod-one-eqI)
    apply clarsimp
    apply (subst finprod-insert)
       apply auto
    apply (metis inv-eq-self)
    done
  finally have (⨂ i∈Units R. i) = ⊖ 1
    by simp
  also have (⨂ i∈Units R. i) = (⨂ i∈Units R. i mod p)
    by (rule finprod-cong') (auto simp: res-units-eq)
  also have ... = (∏ i∈Units R. i) mod p
    by (rule prod-cong) auto
  also have ... = fact (p − 1) mod p
    using assms
    by (simp add: res-prime-units-eq int-prod zmod-int prod-int-eq fact-prod)
  finally have fact (p − 1) mod p = ⊖ 1 .
  then show ?thesis
    by (simp add: cong-def res-neg-eq res-one-eq zmod-int)
qed

lemma wilson-theorem:
  assumes prime p
  shows [fact (p − 1) = − 1] (mod p)
proof (cases p = 2)
  case True
  then show ?thesis
    by (simp add: cong-def fact-prod)
next
  case False
  then show ?thesis
    using assms prime-ge-2-nat
    by (metis residues-prime.wilson-theorem1 residues-prime.intro le-eq-less-or-eq)
qed
```

This result can be transferred to the multiplicative group of $\mathbb{Z}/p\mathbb{Z}$ for p prime.

```
lemma mod-nat-int-pow-eq:
  fixes n :: nat and p a :: int
  shows a ≥ 0 ⟹ p ≥ 0 ⟹ (nat a ^ n) mod (nat p) = nat ((a ^ n) mod p)
  by (simp add: nat-mod-as-int)

theorem residue-prime-mult-group-has-gen:
 fixes p :: nat
```

assumes *prime-p* : *prime p*
shows $\exists\, a \in \{1\ ..\ p - 1\}.\ \{1\ ..\ p - 1\} = \{a\hat{}i \bmod p | i\ .\ i \in UNIV\}$
proof −
have $p \geq 2$
using *prime-gt-1-nat*[*OF prime-p*] **by** *simp*
interpret *R*: *residues-prime p residue-ring p*
by (*simp add*: *residues-prime-def prime-p*)
have *car*: *carrier* (*residue-ring* (*int p*)) − $\{\mathbf{0}_{\textit{residue-ring (int p)}}\} = \{1\ ..\ int\ p - 1\}$
by (*auto simp add*: *R.zero-cong R.res-carrier-eq*)

have $x\ [\hat{}]_{\textit{residue-ring (int p)}}\ i = x\ \hat{}\ i \bmod (int\ p)$
if $x \in \{1\ ..\ int\ p - 1\}$ **for** *x* **and** *i* :: *nat*
using *that R.pow-cong*[*of x i*] **by** *auto*
moreover
obtain *a* **where** *a*: $a \in \{1\ ..\ int\ p - 1\}$
and *a-gen*: $\{1\ ..\ int\ p - 1\} = \{a[\hat{}]_{\textit{residue-ring (int p)}} i | i{::}nat\ .\ i \in UNIV\}$
using *field.finite-field-mult-group-has-gen*[*OF R.is-field*]
by (*auto simp add*: *car*[*symmetric*] *carrier-mult-of*)
moreover
have *nat* ‘ $\{1\ ..\ int\ p - 1\} = \{1\ ..\ p - 1\}$ (**is** *?L* = *?R*)
proof
have $n \in ?R$ **if** $n \in ?L$ **for** *n*
using *that* ‹$p \geq 2$› **by** *force*
then show $?L \subseteq ?R$ **by** *blast*
have $n \in ?L$ **if** $n \in ?R$ **for** *n*
using *that* ‹$p \geq 2$› **by** (*auto intro*: *rev-image-eqI* [*of int n*])
then show $?R \subseteq ?L$ **by** *blast*
qed
moreover
have *nat* ‘ $\{a\hat{}i \bmod (int\ p) \mid i{::}nat.\ i \in UNIV\} = \{nat\ a\hat{}i \bmod p \mid i\ .\ i \in UNIV\}$ (**is** *?L* = *?R*)
proof
have $x \in ?R$ **if** $x \in ?L$ **for** *x*
proof −
from *that* **obtain** *i* **where** *i*: $x = nat\ (a\hat{}i \bmod (int\ p))$
by *blast*
then have $x = nat\ a\ \hat{}\ i \bmod p$
using *mod-nat-int-pow-eq*[*of a int p i*] *a* ‹$p \geq 2$› **by** *auto*
with *i* **show** *?thesis* **by** *blast*
qed
then show $?L \subseteq ?R$ **by** *blast*
have $x \in ?L$ **if** $x \in ?R$ **for** *x*
proof −
from *that* **obtain** *i* **where** *i*: $x = nat\ a\hat{}i \bmod p$
by *blast*
with *mod-nat-int-pow-eq*[*of a int p i*] *a* ‹$p \geq 2$› **show** *?thesis*
by *auto*
qed

then show *?R* ⊆ *?L* **by** *blast*
qed
ultimately have {*1* .. *p* − *1*} = {*nat a*^*i mod p* | *i*. *i* ∈ *UNIV*}
by *presburger*
moreover from *a* **have** *nat a* ∈ {*1* .. *p* − *1*} **by** *force*
ultimately show *?thesis* **..**
qed

5.3 Upper bound for the number of n-th roots

lemma *roots-mod-prime-bound*:
fixes *n c p* :: *nat*
assumes *prime p n* > *0*
defines *A* ≡ {*x*∈{..<*p*}. [*x* ^ *n* = *c*] (*mod p*)}
shows *card A* ≤ *n*
proof −
define *R* **where** *R* = *residue-ring* (*int p*)
from *assms*(*1*) **interpret** *residues-prime p R*
by *unfold-locales* (*simp-all add*: *R-def*)
interpret *R*: *UP-domain R UP R* **by** (*unfold-locales*)

let *?f* = *UnivPoly.monom* (*UP R*) $\mathbf{1}_R$ *n* ⊖$_{(UP\ R)}$ *UnivPoly.monom* (*UP R*) (*int* (*c mod p*)) *0*
have *in-carrier*: *int* (*c mod p*) ∈ *carrier R*
using *prime-gt-1-nat*[*OF assms*(*1*)] **by** (*simp add*: *R-def residue-ring-def*)

have *deg R ?f* = *n*
using *assms in-carrier* **by** (*simp add*: *R.deg-minus-eq*)
hence *f-not-zero*: *?f* ≠ $\mathbf{0}_{UP\ R}$ **using** *assms* **by** (*auto simp add* : *R.deg-nzero-nzero*)
have *roots-bound*: *finite* {*a* ∈ *carrier R*. *UnivPoly.eval R R id a ?f* = $\mathbf{0}_R$} ∧
card {*a* ∈ *carrier R*. *UnivPoly.eval R R id a ?f* = $\mathbf{0}_R$} ≤ *deg R ?f*
using *finite in-carrier* **by** (*intro R.roots-bound*[*OF - f-not-zero*])
simp
have *subs*: {*x* ∈ *carrier R*. *x* [^]$_R$ *n* = *int* (*c mod p*)} ⊆
{*a* ∈ *carrier R*. *UnivPoly.eval R R id a ?f* = $\mathbf{0}_R$}
using *in-carrier* **by** (*auto simp*: *R.evalRR-simps*)
then have *card* {*x* ∈ *carrier R*. *x* [^]$_R$ *n* = *int* (*c mod p*)} ≤
card {*a* ∈ *carrier R*. *UnivPoly.eval R R id a ?f* = $\mathbf{0}_R$}
using *finite* **by** (*intro card-mono*) *auto*
also have ... ≤ *n*
using ‹*deg R ?f* = *n*› *roots-bound* **by** *linarith*
also {
fix *x* **assume** *x* ∈ *carrier R*
hence *x* [^]$_R$ *n* = (*x* ^ *n*) *mod* (*int p*)
by (*subst pow-cong* [*symmetric*]) (*auto simp*: *R-def residue-ring-def*)
}
hence {*x* ∈ *carrier R*. *x* [^]$_R$ *n* = *int* (*c mod p*)} = {*x* ∈ *carrier R*. [*x* ^ *n* = *int c*] (*mod p*)}
by (*fastforce simp*: *cong-def zmod-int*)

also have *bij-betw int A* {$x \in$ *carrier R*. [$x \hat{} n =$ *int c*] (*mod p*)}
by (*rule bij-betwI*[*of int - - nat*])
(*use cong-int-iff* **in** ‹*force simp*: *R-def residue-ring-def A-def*›)+
from *bij-betw-same-card*[*OF this*] **have** *card* {$x \in$ *carrier R*. [$x \hat{} n =$ *int c*] (*mod p*)} = *card A* **..**
finally show *?thesis* **.**
qed

end

6 The sieve of Eratosthenes

theory *Eratosthenes*
imports *Main HOL−Computational-Algebra.Primes*
begin

6.1 Preliminary: strict divisibility

context *dvd*
begin

abbreviation *dvd-strict* :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infixl** ‹*dvd'-strict*› *50*)
where
b dvd-strict a ≡ *b dvd a* ∧ ¬ *a dvd b*

end

6.2 Main corpus

The sieve is modelled as a list of booleans, where *False* means *marked out*.

type-synonym *marks* = *bool list*

definition *numbers-of-marks* :: *nat* ⇒ *marks* ⇒ *nat set*
where
numbers-of-marks n bs = *fst* ‘ {$x \in$ *set* (*enumerate n bs*). *snd x*}

lemma *numbers-of-marks-simps* [*simp*, *code*]:
numbers-of-marks n [] = {}
numbers-of-marks n (*True* # *bs*) = *insert n* (*numbers-of-marks* (*Suc n*) *bs*)
numbers-of-marks n (*False* # *bs*) = *numbers-of-marks* (*Suc n*) *bs*
by (*auto simp add*: *numbers-of-marks-def intro*!: *image-eqI*)

lemma *numbers-of-marks-Suc*:
numbers-of-marks (*Suc n*) *bs* = *Suc* ‘ *numbers-of-marks n bs*
by (*auto simp add*: *numbers-of-marks-def enumerate-Suc-eq image-iff Bex-def*)

lemma *numbers-of-marks-replicate-False* [*simp*]:

numbers-of-marks n (replicate m False) = {}
by (*auto simp add*: *numbers-of-marks-def enumerate-replicate-eq*)

lemma *numbers-of-marks-replicate-True* [*simp*]:
numbers-of-marks n (replicate m True) = {n..<n+m}
by (*auto simp add*: *numbers-of-marks-def enumerate-replicate-eq image-def*)

lemma *in-numbers-of-marks-eq*:
m ∈ numbers-of-marks n bs ⟷ m ∈ {n..<n + length bs} ∧ bs ! (m − n)
by (*simp add*: *numbers-of-marks-def in-set-enumerate-eq image-iff add.commute*)

lemma *sorted-list-of-set-numbers-of-marks*:
sorted-list-of-set (numbers-of-marks n bs) = map fst (filter snd (enumerate n bs))
by (*auto simp add*: *numbers-of-marks-def distinct-map*
intro!: *sorted-filter distinct-filter inj-onI sorted-distinct-set-unique*)

Marking out multiples in a sieve

definition *mark-out* :: *nat ⇒ marks ⇒ marks*
where
mark-out n bs = map (λ(q, b). b ∧ ¬ Suc n dvd Suc (Suc q)) (enumerate n bs)

lemma *mark-out-Nil* [*simp*]: *mark-out n [] = []*
by (*simp add*: *mark-out-def*)

lemma *length-mark-out* [*simp*]: *length (mark-out n bs) = length bs*
by (*simp add*: *mark-out-def*)

lemma *numbers-of-marks-mark-out*:
numbers-of-marks n (mark-out m bs) = {q ∈ numbers-of-marks n bs. ¬ Suc m dvd Suc q − n}
by (*auto simp add*: *numbers-of-marks-def mark-out-def in-set-enumerate-eq image-iff*
nth-enumerate-eq less-eq-dvd-minus)

Auxiliary operation for efficient implementation

definition *mark-out-aux* :: *nat ⇒ nat ⇒ marks ⇒ marks*
where
mark-out-aux n m bs =
map (λ(q, b). b ∧ (q < m + n ∨ ¬ Suc n dvd Suc (Suc q) + (n − m mod Suc n))) (enumerate n bs)

lemma *mark-out-code* [*code*]: *mark-out n bs = mark-out-aux n n bs*
proof −
have *aux*: *False*
if *A*: *Suc n dvd Suc (Suc a)*
and *B*: *a < n + n*
and *C*: *n ≤ a*
for *a*
proof (*cases n = 0*)

```
    case True
    with A B C show ?thesis by simp
  next
    case False
    define m where m = Suc n
    then have m > 0 by simp
    from False have n > 0 by simp
    from A obtain q where q: Suc (Suc a) = Suc n * q by (rule dvdE)
    have q > 0
    proof (rule ccontr)
      assume ¬ q > 0
      with q show False by simp
    qed
    with ‹n > 0› have Suc n * q ≥ 2 by (auto simp add: gr0-conv-Suc)
    with q have a: a = Suc n * q − 2 by simp
    with B have q + n * q < n + n + 2 by auto
    then have m * q < m * 2 by (simp add: m-def)
    with ‹m > 0› ‹q > 0› have q = 1 by simp
    with a have a = n − 1 by simp
    with ‹n > 0› C show False by simp
  qed
  show ?thesis
    by (auto simp add: mark-out-def mark-out-aux-def in-set-enumerate-eq intro:
aux)
qed

lemma mark-out-aux-simps [simp, code]:
  mark-out-aux n m [] = []
  mark-out-aux n 0 (b # bs) = False # mark-out-aux n n bs
  mark-out-aux n (Suc m) (b # bs) = b # mark-out-aux n m bs
proof goal-cases
  case 1
  show ?case
    by (simp add: mark-out-aux-def)
next
  case 2
  show ?case
    by (auto simp add: mark-out-code [symmetric] mark-out-aux-def mark-out-def
      enumerate-Suc-eq in-set-enumerate-eq less-eq-dvd-minus)
next
  case 3
  { define v where v = Suc m
    define w where w = Suc n
    fix q
    assume m + n ≤ q
    then obtain r where q: q = m + n + r by (auto simp add: le-iff-add)
    { fix u
      from w-def have u mod w < w by simp
      then have u + (w − u mod w) = w + (u − u mod w)
```

```
      by simp
    then have u + (w − u mod w) = w + u div w * w
      by (simp add: minus-mod-eq-div-mult)
  }
  then have w dvd v + w + r + (w − v mod w) ⟷ w dvd m + w + r + (w
− m mod w)
    by (simp add: add.assoc add.left-commute [of m] add.left-commute [of v]
      dvd-add-left-iff dvd-add-right-iff)
  moreover from q have Suc q = m + w + r by (simp add: w-def)
  moreover from q have Suc (Suc q) = v + w + r by (simp add: v-def w-def)
  ultimately have w dvd Suc (Suc (q + (w − v mod w))) ⟷ w dvd Suc (q +
(w − m mod w))
    by (simp only: add-Suc [symmetric])
  then have Suc n dvd Suc (Suc (Suc (q + n) − Suc m mod Suc n)) ⟷
    Suc n dvd Suc (Suc (q + n − m mod Suc n))
    by (simp add: v-def w-def Suc-diff-le trans-le-add2)
 }
 then show ?case
  by (auto simp add: mark-out-aux-def
    enumerate-Suc-eq in-set-enumerate-eq not-less)
qed
```

Main entry point to sieve

```
fun sieve :: nat ⇒ marks ⇒ marks
where
  sieve n [] = []
| sieve n (False # bs) = False # sieve (Suc n) bs
| sieve n (True # bs) = True # sieve (Suc n) (mark-out n bs)
```

There are the following possible optimisations here:

- *sieve* can abort as soon as n is too big to let *mark-out* have any effect.
- Search for further primes can be given up as soon as the search position exceeds the square root of the maximum candidate.

This is left as an constructive exercise to the reader.

```
lemma numbers-of-marks-sieve:
  numbers-of-marks (Suc n) (sieve n bs) =
    {q ∈ numbers-of-marks (Suc n) bs. ∀ m ∈ numbers-of-marks (Suc n) bs. ¬ m
dvd-strict q}
proof (induct n bs rule: sieve.induct)
  case 1
  show ?case by simp
next
  case 2
  then show ?case by simp
next
```

```
  case (3 n bs)
  have aux: n ∈ Suc ‘ M ⟷ n > 0 ∧ n − 1 ∈ M (is ?lhs ⟷ ?rhs) for M n
  proof
    show ?rhs if ?lhs using that by auto
    show ?lhs if ?rhs
    proof −
      from that have n > 0 and n − 1 ∈ M by auto
      then have Suc (n − 1) ∈ Suc ‘ M by blast
      with ‹n > 0› show n ∈ Suc ‘ M by simp
    qed
  qed
  have aux1: False if Suc (Suc n) ≤ m and m dvd Suc n for m :: nat
  proof −
    from ‹m dvd Suc n› obtain q where Suc n = m * q ..
    with ‹Suc (Suc n) ≤ m› have Suc (m * q) ≤ m by simp
    then have m * q < m by arith
    with ‹Suc n = m * q› show ?thesis by simp
  qed
  have aux2: m dvd q
    if 1: ∀ q>0. 1 < q ⟶ Suc n < q ⟶ q ≤ Suc (n + length bs) ⟶
      bs ! (q − Suc (Suc n)) ⟶ ¬ Suc n dvd q ⟶ q dvd m ⟶ m dvd q
    and 2: ¬ Suc n dvd m q dvd m
    and 3: Suc n < q q ≤ Suc (n + length bs) bs ! (q − Suc (Suc n))
    for m q :: nat
  proof −
    from 1 have *: ⋀q. Suc n < q ⟹ q ≤ Suc (n + length bs) ⟹
      bs ! (q − Suc (Suc n)) ⟹ ¬ Suc n dvd q ⟹ q dvd m ⟹ m dvd q
      by auto
    from 2 have ¬ Suc n dvd q by auto
    moreover note 3
    moreover note ‹q dvd m›
    ultimately show ?thesis by (auto intro: *)
  qed
  from 3 show ?case
   apply (simp-all add: numbers-of-marks-mark-out numbers-of-marks-Suc Compr-image-eq
      inj-image-eq-iff in-numbers-of-marks-eq Ball-def imp-conjL aux)
    apply safe
    apply (simp-all add: less-diff-conv2 le-diff-conv2 dvd-minus-self not-less)
    apply (clarsimp dest!: aux1)
    apply (simp add: Suc-le-eq less-Suc-eq-le)
    apply (rule aux2)
    apply (clarsimp dest!: aux1)+
    done
qed
```

Relation of the sieve algorithm to actual primes

```
definition primes-upto :: nat ⇒ nat list
where
  primes-upto n = sorted-list-of-set {m. m ≤ n ∧ prime m}
```

```
lemma set-primes-upto: set (primes-upto n) = {m. m ≤ n ∧ prime m}
  by (simp add: primes-upto-def)

lemma sorted-primes-upto [iff]: sorted (primes-upto n)
  by (simp add: primes-upto-def)

lemma distinct-primes-upto [iff]: distinct (primes-upto n)
  by (simp add: primes-upto-def)

lemma set-primes-upto-sieve:
  set (primes-upto n) = numbers-of-marks 2 (sieve 1 (replicate (n − 1) True))
proof −
  consider n = 0 ∨ n = 1 | n > 1 by arith
  then show ?thesis
  proof cases
    case 1
    then show ?thesis
      by (auto simp add: numbers-of-marks-sieve numeral-2-eq-2 set-primes-upto
        dest: prime-gt-Suc-0-nat)
  next
    case 2
    {
      fix m q
      assume Suc (Suc 0) ≤ q
        and q < Suc n
        and m dvd q
      then have m < Suc n by (auto dest: dvd-imp-le)
      assume ∗: ∀ m∈{Suc (Suc 0)..<Suc n}. m dvd q ⟶ q dvd m
        and m dvd q and m ≠ 1
      have m = q
      proof (cases m = 0)
        case True with ‹m dvd q› show ?thesis by simp
      next
        case False with ‹m ≠ 1› have Suc (Suc 0) ≤ m by arith
        with ‹m < Suc n› ∗ ‹m dvd q› have q dvd m by simp
        with ‹m dvd q› show ?thesis by (simp add: dvd-antisym)
      qed
    }
    then have aux: ⋀m q. Suc (Suc 0) ≤ q ⟹
      q < Suc n ⟹
      m dvd q ⟹
      ∀ m∈{Suc (Suc 0)..<Suc n}. m dvd q ⟶ q dvd m ⟹
      m dvd q ⟹ m ≠ q ⟹ m = 1 by auto
    from 2 show ?thesis
     apply (auto simp add: numbers-of-marks-sieve numeral-2-eq-2 set-primes-upto
        dest: prime-gt-Suc-0-nat)
      apply (metis One-nat-def Suc-le-eq less-not-refl prime-nat-iff)
      apply (metis One-nat-def Suc-le-eq aux prime-nat-iff)
```

done
qed
qed

lemma *primes-upto-sieve* [*code*]:
primes-upto n = map fst (filter snd (enumerate 2 (sieve 1 (replicate (n − 1) True))))
using *primes-upto-def set-primes-upto set-primes-upto-sieve sorted-list-of-set-numbers-of-marks*
by *presburger*

lemma *prime-in-primes-upto*: *prime n ⟷ n ∈ set (primes-upto n)*
by (*simp add*: *set-primes-upto*)

6.3 Application: smallest prime beyond a certain number

definition *smallest-prime-beyond* :: *nat ⇒ nat*
where
smallest-prime-beyond n = (LEAST p. prime p ∧ p ≥ n)

lemma *prime-smallest-prime-beyond* [*iff*]: *prime (smallest-prime-beyond n)* (**is** *?P*)
and *smallest-prime-beyond-le* [*iff*]: *smallest-prime-beyond n ≥ n* (**is** *?Q*)
proof −
let *?least = LEAST p. prime p ∧ p ≥ n*
from *primes-infinite* **obtain** *q* **where** *prime q ∧ q ≥ n*
by (*metis finite-nat-set-iff-bounded-le mem-Collect-eq nat-le-linear*)
then have *prime ?least ∧ ?least ≥ n*
by (*rule LeastI*)
then show *?P* **and** *?Q*
by (*simp-all add*: *smallest-prime-beyond-def*)
qed

lemma *smallest-prime-beyond-smallest*: *prime p ⟹ p ≥ n ⟹ smallest-prime-beyond n ≤ p*
by (*simp only*: *smallest-prime-beyond-def*) (*auto intro*: *Least-le*)

lemma *smallest-prime-beyond-eq*:
prime p ⟹ p ≥ n ⟹ (⋀q. prime q ⟹ q ≥ n ⟹ q ≥ p) ⟹ smallest-prime-beyond n = p
by (*simp only*: *smallest-prime-beyond-def*) (*auto intro*: *Least-equality*)

definition *smallest-prime-between* :: *nat ⇒ nat ⇒ nat option*
where
smallest-prime-between m n =
(if (∃ p. prime p ∧ m ≤ p ∧ p ≤ n) then Some (smallest-prime-beyond m) else None)

lemma *smallest-prime-between-None*:
smallest-prime-between m n = None ⟷ (∀ q. m ≤ q ∧ q ≤ n ⟶ ¬ prime q)

by (*auto simp add*: *smallest-prime-between-def*)

lemma *smallest-prime-betwen-Some*:
smallest-prime-between m n = *Some p* ⟷ *smallest-prime-beyond m* = *p* ∧ *p* ≤ *n*
by (*auto simp add*: *smallest-prime-between-def dest*: *smallest-prime-beyond-smallest* [*of* - *m*])

lemma [*code*]: *smallest-prime-between m n* = *List.find* (λ*p*. *p* ≥ *m*) (*primes-upto n*)
proof −
have *List.find* (λ*p*. *p* ≥ *m*) (*primes-upto n*) = *Some* (*smallest-prime-beyond m*)
if *assms*: *m* ≤ *p prime p p* ≤ *n* **for** *p*
proof −
define *A* **where** *A* = {*p*. *p* ≤ *n* ∧ *prime p* ∧ *m* ≤ *p*}
from *assms* **have** *smallest-prime-beyond m* ≤ *p*
by (*auto intro*: *smallest-prime-beyond-smallest*)
from *this* ‹*p* ≤ *n*› **have** ∗: *smallest-prime-beyond m* ≤ *n*
by (*rule order-trans*)
from *assms* **have** *ex*: ∃ *p*≤*n*. *prime p* ∧ *m* ≤ *p*
by *auto*
then have *finite A*
by (*auto simp add*: *A-def*)
with ∗ **have** *Min A* = *smallest-prime-beyond m*
by (*auto simp add*: *A-def intro*: *Min-eqI smallest-prime-beyond-smallest*)
with *ex sorted-primes-upto* **show** *?thesis*
by (*auto simp add*: *set-primes-upto sorted-find-Min A-def*)
qed
then show *?thesis*
by (*auto simp add*: *smallest-prime-between-def find-None-iff set-primes-upto intro*!: *sym* [*of* - *None*])
qed

definition *smallest-prime-beyond-aux* :: *nat* ⇒ *nat* ⇒ *nat*
where
smallest-prime-beyond-aux k n = *smallest-prime-beyond n*

lemma [*code*]:
smallest-prime-beyond-aux k n =
(*case smallest-prime-between n* (*k* ∗ *n*) *of*
Some p ⇒ *p*
| *None* ⇒ *smallest-prime-beyond-aux* (*Suc k*) *n*)
by (*simp add*: *smallest-prime-beyond-aux-def smallest-prime-betwen-Some split*: *option.split*)

lemma [*code*]: *smallest-prime-beyond n* = *smallest-prime-beyond-aux 2 n*
by (*simp add*: *smallest-prime-beyond-aux-def*)

end

7 Fast modular exponentiation

theory *Mod-Exp*
imports *Cong HOL−Library.Power-By-Squaring*
begin

context *euclidean-semiring-cancel*
begin

definition *mod-exp-aux* :: *'a* ⇒ *'a* ⇒ *'a* ⇒ *nat* ⇒ *'a*
where *mod-exp-aux m* = *efficient-funpow* (λ*x y*. *x* ∗ *y mod m*)

lemma *mod-exp-aux-code* [*code*]:
mod-exp-aux m y x n =
(*if n* = *0 then y*
else if n = *1 then* (*x* ∗ *y*) *mod m*
else if even n then mod-exp-aux m y ((*x* ∗ *x*) *mod m*) (*n div 2*)
else mod-exp-aux m ((*x* ∗ *y*) *mod m*) ((*x* ∗ *x*) *mod m*) (*n div 2*))
unfolding *mod-exp-aux-def* **by** (*rule efficient-funpow-code*)

lemma *mod-exp-aux-correct*:
mod-exp-aux m y x n mod m = (*x* ^ *n* ∗ *y*) *mod m*
proof −
have *mod-exp-aux m y x n* = *efficient-funpow* (λ*x y*. *x* ∗ *y mod m*) *y x n*
by (*simp add*: *mod-exp-aux-def*)
also have ... = ((λ*y*. *x* ∗ *y mod m*) ^^ *n*) *y*
by (*rule efficient-funpow-correct*) (*simp add*: *mod-mult-left-eq mod-mult-right-eq mult-ac*)
also have ((λ*y*. *x* ∗ *y mod m*) ^^ *n*) *y mod m* = (*x* ^ *n* ∗ *y*) *mod m*
proof (*induction n*)
case (*Suc n*)
hence *x* ∗ ((λ*y*. *x* ∗ *y mod m*) ^^ *n*) *y mod m* = *x* ∗ *x* ^ *n* ∗ *y mod m*
by (*metis mod-mult-right-eq mult.assoc*)
thus *?case* **by** *auto*
qed *auto*
finally show *?thesis* **.**
qed

definition *mod-exp* :: *'a* ⇒ *nat* ⇒ *'a* ⇒ *'a*
where *mod-exp b e m* = (*b* ^ *e*) *mod m*

lemma *mod-exp-code* [*code*]: *mod-exp b e m* = *mod-exp-aux m 1 b e mod m*
by (*simp add*: *mod-exp-def mod-exp-aux-correct*)

end

lemmas [*code-abbrev*] = *mod-exp-def*[**where** *?'a* = *nat*] *mod-exp-def*[**where** *?'a* = *int*]

lemma *cong-power-nat-code* [*code-unfold*]:
  $[b \mathbin{\hat{}} e = (x :: nat)]\ (mod\ m) \longleftrightarrow mod\text{-}exp\ b\ e\ m = x\ mod\ m$
  by (*simp add*: *mod-exp-def cong-def*)

lemma *cong-power-int-code* [*code-unfold*]:
  $[b \mathbin{\hat{}} e = (x :: int)]\ (mod\ m) \longleftrightarrow mod\text{-}exp\ b\ e\ m = x\ mod\ m$
  by (*simp add*: *mod-exp-def cong-def*)

The following rules allow the simplifier to evaluate *mod-exp* efficiently.

lemma *eval-mod-exp-aux* [*simp*]:
  $mod\text{-}exp\text{-}aux\ m\ y\ x\ 0 = y$
  $mod\text{-}exp\text{-}aux\ m\ y\ x\ (Suc\ 0) = (x * y)\ mod\ m$
  $mod\text{-}exp\text{-}aux\ m\ y\ x\ (numeral\ (num.Bit0\ n)) =$
    $mod\text{-}exp\text{-}aux\ m\ y\ (x^2\ mod\ m)\ (numeral\ n)$
  $mod\text{-}exp\text{-}aux\ m\ y\ x\ (numeral\ (num.Bit1\ n)) =$
    $mod\text{-}exp\text{-}aux\ m\ ((x * y)\ mod\ m)\ (x^2\ mod\ m)\ (numeral\ n)$
proof −
  define n' **where** $n' = (numeral\ n :: nat)$
  have [*simp*]: $n' \neq 0$ **by** (*auto simp*: n'-*def*)

  show $mod\text{-}exp\text{-}aux\ m\ y\ x\ 0 = y$ **and** $mod\text{-}exp\text{-}aux\ m\ y\ x\ (Suc\ 0) = (x * y)\ mod\ m$
    by (*simp-all add*: *mod-exp-aux-def*)

  have $numeral\ (num.Bit0\ n) = (2 * n')$
    by (*subst numeral.numeral-Bit0*) (*simp del*: *arith-simps add*: n'-*def*)
  also have $mod\text{-}exp\text{-}aux\ m\ y\ x\ \ldots = mod\text{-}exp\text{-}aux\ m\ y\ (x\hat{}2\ mod\ m)\ n'$
    by (*subst mod-exp-aux-code*) (*simp-all add*: *power2-eq-square*)
  finally show $mod\text{-}exp\text{-}aux\ m\ y\ x\ (numeral\ (num.Bit0\ n)) =$
        $mod\text{-}exp\text{-}aux\ m\ y\ (x^2\ mod\ m)\ (numeral\ n)$
    by (*simp add*: n'-*def*)

  have $numeral\ (num.Bit1\ n) = Suc\ (2 * n')$
    by (*subst numeral.numeral-Bit1*) (*simp del*: *arith-simps add*: n'-*def*)
  also have $mod\text{-}exp\text{-}aux\ m\ y\ x\ \ldots = mod\text{-}exp\text{-}aux\ m\ ((x * y)\ mod\ m)\ (x\hat{}2\ mod\ m)\ n'$
    by (*subst mod-exp-aux-code*) (*simp-all add*: *power2-eq-square*)
  finally show $mod\text{-}exp\text{-}aux\ m\ y\ x\ (numeral\ (num.Bit1\ n)) =$
        $mod\text{-}exp\text{-}aux\ m\ ((x * y)\ mod\ m)\ (x^2\ mod\ m)\ (numeral\ n)$
    by (*simp add*: n'-*def*)
qed

lemma *eval-mod-exp* [*simp*]:
  $mod\text{-}exp\ b'\ 0\ m' = 1\ mod\ m'$
  $mod\text{-}exp\ b'\ 1\ m' = b'\ mod\ m'$
  $mod\text{-}exp\ b'\ (Suc\ 0)\ m' = b'\ mod\ m'$
  $mod\text{-}exp\ b'\ e'\ 0 = b' \mathbin{\hat{}} e'$
  $mod\text{-}exp\ b'\ e'\ 1 = 0$

mod-exp b′ e′ (Suc 0) = 0
mod-exp 0 1 m′ = 0
mod-exp 0 (Suc 0) m′ = 0
mod-exp 0 (numeral e) m′ = 0
mod-exp 1 e′ m′ = 1 mod m′
mod-exp (Suc 0) e′ m′ = 1 mod m′
mod-exp (numeral b) (numeral e) (numeral m) =
mod-exp-aux (numeral m) 1 (numeral b) (numeral e) mod numeral m
by (*simp-all add*: *mod-exp-def mod-exp-aux-correct*)

end

theory *Euler-Criterion*
imports *Residues*
begin

context
fixes *p* :: *nat*
fixes *a* :: *int*

assumes *p-prime*: *prime p*
assumes *p-ge-2*: *2 < p*
assumes *p-a-relprime*: [*a ≠ 0*](*mod p*)
begin

private lemma *odd-p*: *odd p*
using *p-ge-2 p-prime prime-odd-nat* **by** *blast*

private lemma *p-minus-1-int*:
int (p − 1) = int p − 1
by (*metis of-nat-1 of-nat-diff p-prime prime-ge-1-nat*)

private lemma *p-not-eq-Suc-0* [*simp*]:
p ≠ Suc 0
using *p-ge-2* **by** *simp*

private lemma *one-mod-int-p-eq* [*simp*]:
1 mod int p = 1
proof −
from *p-ge-2* **have** *int 1 mod int p = int 1*
by *simp*
then show *?thesis*
by *simp*
qed

private lemma *E-1*:
assumes *QuadRes* (*int p*) *a*
shows [*a* ^ ((*p − 1*) *div 2*) = *1*] (*mod int p*)

proof −
from *assms* **obtain** *b* **where** *b*: [*b* ^ 2 = *a*] (*mod int p*)
unfolding *QuadRes-def* **by** *blast*
then have [*a* ^ ((*p* − 1) *div* 2) = *b* ^ (2 ∗ ((*p* − 1) *div* 2))] (*mod int p*)
by (*simp add*: *cong-pow cong-sym power-mult*)
then have [*a* ^ ((*p* − 1) *div* 2) = *b* ^ (*p* − 1)] (*mod int p*)
using *odd-p* **by** *force*
moreover have [*b* ^ (*p* − 1) = 1] (*mod int p*)
proof −
have [*nat* |*b*| ^ (*p* − 1) = 1] (*mod p*)
using *p-prime* **proof** (*rule fermat-theorem*)
from *b p-a-relprime* **show** ¬ *p dvd nat* |*b*|
by (*auto simp add*: *cong-iff-dvd-diff power2-eq-square*)
(*metis cong-iff-dvd-diff cong-dvd-iff dvd-mult2*)
qed
then have *nat* |*b*| ^ (*p* − 1) *mod p* = 1 *mod p*
by (*simp add*: *cong-def*)
then have *int* (*nat* |*b*| ^ (*p* − 1) *mod p*) = *int* (1 *mod p*)
by *simp*
moreover from *odd-p* **have** |*b*| ^ (*p* − *Suc* 0) = *b* ^ (*p* − *Suc* 0)
by (*simp add*: *power-even-abs*)
ultimately show *?thesis*
by (*auto simp add*: *cong-def zmod-int*)
qed
ultimately show *?thesis*
by (*auto intro*: *cong-trans*)
qed

private definition *S1* :: *int set* **where** *S1* = {0 <.. *int p* − 1}

private definition *P* :: *int* ⇒ *int* ⇒ *bool* **where**
P x y ⟷ [*x* ∗ *y* = *a*] (*mod p*) ∧ *y* ∈ *S1*

private definition *f-1* :: *int* ⇒ *int* **where**
f-1 x = (*THE y*. *P x y*)

private definition *f* :: *int* ⇒ *int set* **where**
f x = {*x*, *f-1 x*}

private definition *S2* :: *int set set* **where** *S2* = *f* ‘ *S1*

private lemma *P-lemma*: **assumes** *x* ∈ *S1*
shows ∃! *y*. *P x y*
proof −
have ¬ *p dvd x* **using** *assms zdvd-not-zless S1-def* **by** *auto*
hence *co-xp*: *coprime x p* **using** *p-prime prime-imp-coprime-int*[*of p x*]
by (*simp add*: *ac-simps*)
then obtain *y*′ **where** *y*′: [*x* ∗ *y*′ = 1] (*mod p*) **using** *cong-solve-coprime-int* **by**
blast

moreover define y **where** $y = y' * a \ mod \ p$
ultimately have $[x * y = a] \ (mod \ p)$
using *mod-mult-right-eq* $[of \ x \ y' * a \ p]$
cong-scalar-right $[of \ x * y' \ 1 \ int \ p \ a]$
by (*auto simp add*: *cong-def ac-simps*)
moreover have $y \in \{0 \ .. \ int \ p - 1\}$ **unfolding** *y-def* **using** *p-ge-2* **by** *auto*
hence $y \in S1$ **using** *calculation cong-iff-dvd-diff p-a-relprime S1-def cong-dvd-iff*
by *fastforce*
ultimately have $P \ x \ y$ **unfolding** *P-def* **by** *blast*
moreover {
fix *y1 y2*
assume $P \ x \ y1 \ P \ x \ y2$
moreover hence $[y1 = y2] \ (mod \ p)$ **unfolding** *P-def*
using *co-xp cong-mult-lcancel*$[of \ x \ p \ y1 \ y2]$ *cong-sym cong-trans* **by** *blast*
ultimately have $y1 = y2$ **unfolding** *P-def S1-def* **using** *cong-less-imp-eq-int*
by *auto*
}
ultimately show *?thesis* **by** *blast*
qed

private lemma *f-1-lemma-1*: **assumes** $x \in S1$
shows $P \ x \ (f\text{-}1 \ x)$ **using** *assms P-lemma theI'*$[of \ P \ x]$ *f-1-def* **by** *presburger*

private lemma *f-1-lemma-2*: **assumes** $x \in S1$
shows $f\text{-}1 \ (f\text{-}1 \ x) = x$
using *assms f-1-lemma-1*$[of \ x]$ *f-1-def P-lemma*$[of \ f\text{-}1 \ x]$ *P-def* **by** (*auto simp*: *mult.commute*)

private lemma *f-lemma-1*: **assumes** $x \in S1$
shows $f \ x = f \ (f\text{-}1 \ x)$ **using** *f-def f-1-lemma-2*$[of \ x]$ *assms* **by** *auto*

private lemma *l1*: **assumes** $\neg$ *QuadRes p a* $x \in S1$
shows $x \neq f\text{-}1 \ x$
using *f-1-lemma-1*$[of \ x]$ *assms* **unfolding** *P-def QuadRes-def power2-eq-square*
by *fastforce*

private lemma *l2*: **assumes** $\neg$ *QuadRes p a* $x \in S1$
shows $[\prod \ (f \ x) = a] \ (mod \ p)$
using *assms l1 f-1-lemma-1 P-def f-def* **by** *auto*

private lemma *l3*: **assumes** $x \in S2$
shows *finite x* **using** *assms f-def S2-def* **by** *auto*

private lemma *l4*: $S1 = \bigcup \ S2$ **using** *f-1-lemma-1 P-def f-def S2-def* **by** *auto*

private lemma *l5*: **assumes** $x \in S2 \ y \in S2 \ x \neq y$
shows $x \cap y = \{\}$
proof −
obtain *a b* **where** *ab*: $x = f \ a \ a \in S1 \ y = f \ b \ b \in S1$ **using** *assms S2-def* **by**

auto
hence $a \neq b$ $a \neq f\text{-}1\ b$ $f\text{-}1\ a \neq b$ **using** *assms*(3) *f-lemma-1* **by** *blast+*
moreover hence $f\text{-}1\ a \neq f\text{-}1\ b$ **using** *f-1-lemma-2*[*of a*] *f-1-lemma-2*[*of b*] *ab* **by** *force*
ultimately show *?thesis* **using** *f-def ab* **by** *fastforce*
qed

private lemma *l6*: $prod\ Prod\ S2 = \prod\ S1$
using *prod.Union-disjoint*[*of S2* $\lambda x.\ x$] *l3 l4 l5* **unfolding** *comp-def* **by** *auto*

private lemma *l7*: $fact\ n = \prod\ \{0 <..\ int\ n\}$
proof (*induction n*)
case (*Suc n*)
have $int\ (Suc\ n) = int\ n + 1$ **by** *simp*
hence $insert\ (int\ (Suc\ n))\ \{0<..int\ n\} = \{0<..int\ (Suc\ n)\}$ **by** *auto*
thus *?case* **using** *prod.insert*[*of* $\{0<..int\ n\}$ $int\ (Suc\ n)$ $\lambda x.\ x$] *Suc fact-Suc* **by** *auto*
qed *simp*

private lemma *l8*: $fact\ (p - 1) = \prod\ S1$ **using** *l7*[*of* $p - 1$] *S1-def p-minus-1-int* **by** *presburger*

private lemma *l9*: $[prod\ Prod\ S2 = -1]\ (mod\ p)$
using *l6 l8 wilson-theorem*[*of p*] *p-prime* **by** *presburger*

private lemma *l10*: **assumes** $card\ S = n$ $\bigwedge x.\ x \in S \Longrightarrow [g\ x = a]\ (mod\ p)$
shows $[prod\ g\ S = a\ \hat{}\ n]\ (mod\ p)$ **using** *assms*
proof (*induction n arbitrary*: *S*)
case *0*
thus *?case* **using** *card-0-eq*[*of S*] *prod.empty prod.infinite* **by** *fastforce*
next
case (*Suc n*)
then obtain *x* **where** *x*: $x \in S$ **by** *force*
define S' **where** $S' = S - \{x\}$
hence $[prod\ g\ S' = a\ \hat{}\ n]\ (mod\ int\ p)$
using *x Suc*(1)[*of S'*] *Suc*(2) *Suc*(3) **by** (*simp add*: *card-ge-0-finite*)
moreover have $prod\ g\ S = g\ x * prod\ g\ S'$
using *x S'-def Suc*(2) *prod.remove*[*of S x g*] **by** *fastforce*
ultimately show *?case* **using** *x Suc*(3) *cong-mult*
by *simp blast*
qed

private lemma *l11*: **assumes** $\neg\ QuadRes\ p\ a$
shows $card\ S2 = (p - 1)\ div\ 2$
proof −
have $sum\ card\ S2 = 2 * card\ S2$
using *sum.cong*[*of S2 S2 card* $\lambda x.\ 2$] *l1 f-def S2-def assms* **by** *fastforce*
moreover have $p - 1 = sum\ card\ S2$
using *l4 card-UN-disjoint*[*of S2* $\lambda x.\ x$] *l3 l5 S1-def S2-def* **by** *auto*

ultimately show *?thesis* **by** *linarith*
qed

private lemma *l12*: **assumes** ¬ *QuadRes p a*
shows [*prod Prod S2* = *a* ^ ((*p* − *1*) *div 2*)] (*mod p*)
using *assms l2 l10 l11* **unfolding** *S2-def* **by** *blast*

private lemma *E-2*: **assumes** ¬ *QuadRes p a*
shows [*a* ^ ((*p* − *1*) *div 2*) = −*1*] (*mod p*) **using** *l9 l12 cong-trans cong-sym assms* **by** *blast*

lemma *euler-criterion-aux*: [(*Legendre a p*) = *a* ^ ((*p* − *1*) *div 2*)] (*mod p*)
using *p-a-relprime* **by** (*auto simp add*: *Legendre-def*
intro!: *cong-sym* [*of - 1*] *cong-sym* [*of - − 1*]
dest: *E-1 E-2*)

end

theorem *euler-criterion*: **assumes** *prime p 2* < *p*
shows [(*Legendre a p*) = *a* ^ ((*p* − *1*) *div 2*)] (*mod p*)
proof (*cases* [*a* = *0*] (*mod p*))
case *True*
then have [*a* ^ ((*p* − *1*) *div 2*) = *0* ^ ((*p* − *1*) *div 2*)] (*mod p*)
using *cong-pow* **by** *blast*
moreover have (*0*::*int*) ^ ((*p* − *1*) *div 2*) = *0*
using *zero-power* [*of* (*p* − *1*) *div 2*] *assms*(*2*) **by** *simp*
ultimately have [*a* ^ ((*p* − *1*) *div 2*) = *0*] (*mod p*)
using *True assms*(*1*) *prime-dvd-power-int-iff*
by (*simp add*: *cong-iff-dvd-diff*)
then show *?thesis* **unfolding** *Legendre-def* **using** *True cong-sym*
by *auto*
next
case *False*
then show *?thesis*
using *euler-criterion-aux assms* **by** *presburger*
qed

hide-fact *euler-criterion-aux*

end

8 Gauss' Lemma

theory *Gauss*
imports *Euler-Criterion*
begin

lemma *cong-prime-prod-zero-nat*:
[*a* ∗ *b* = *0*] (*mod p*) ⟹ *prime p* ⟹ [*a* = *0*] (*mod p*) ∨ [*b* = *0*] (*mod p*)

for $a :: nat$
by (*auto simp add*: *cong-altdef-nat prime-dvd-mult-iff*)

lemma *cong-prime-prod-zero-int*:
$[a * b = 0] \ (mod\ p) \Longrightarrow prime\ p \Longrightarrow [a = 0] \ (mod\ p) \vee [b = 0] \ (mod\ p)$
for $a :: int$
by (*simp add*: *cong-0-iff prime-dvd-mult-iff*)

locale *GAUSS* =
fixes $p :: nat$
fixes $a :: int$
assumes *p-prime*: *prime p*
assumes *p-ge-2*: $2 < p$
assumes *p-a-relprime*: $[a \neq 0](mod\ p)$
assumes *a-nonzero*: $0 < a$
begin

definition $A = \{0::int <.. ((int\ p - 1)\ div\ 2)\}$
definition $B = (\lambda x.\ x * a)\ `\ A$
definition $C = (\lambda x.\ x\ mod\ p)\ `\ B$
definition $D = C \cap \{.. (int\ p - 1)\ div\ 2\}$
definition $E = C \cap \{(int\ p - 1)\ div\ 2 <..\}$
definition $F = (\lambda x.\ (int\ p - x))\ `\ E$

8.1 Basic properties of p

lemma *odd-p*: *odd p*
by (*metis p-prime p-ge-2 prime-odd-nat*)

lemma *p-minus-one-l*: $(int\ p - 1)\ div\ 2 < p$
proof −
have $(p - 1)\ div\ 2 \leq (p - 1)\ div\ 1$
by (*metis div-by-1 div-le-dividend*)
also have $\ldots = p - 1$ **by** *simp*
finally show *?thesis*
using *p-ge-2* **by** *arith*
qed

lemma *p-eq2*: $int\ p = (2 * ((int\ p - 1)\ div\ 2)) + 1$
using *odd-p p-ge-2 nonzero-mult-div-cancel-left* [*of 2 p − 1*] **by** *simp*

lemma *p-odd-int*: **obtains** $z :: int$ **where** $int\ p = 2 * z + 1\ 0 < z$
proof
let *?z* = $(int\ p - 1)\ div\ 2$
show $int\ p = 2 * ?z + 1$ **by** (*rule p-eq2*)
show $0 < ?z$
using *p-ge-2* **by** *linarith*
qed

8.2 Basic Properties of the Gauss Sets

lemma *finite-A*: *finite A*
by (*auto simp add*: *A-def*)

lemma *finite-B*: *finite B*
by (*auto simp add*: *B-def finite-A*)

lemma *finite-C*: *finite C*
by (*auto simp add*: *C-def finite-B*)

lemma *finite-D*: *finite D*
by (*auto simp add*: *D-def finite-C*)

lemma *finite-E*: *finite E*
by (*auto simp add*: *E-def finite-C*)

lemma *finite-F*: *finite F*
by (*auto simp add*: *F-def finite-E*)

lemma *C-eq*: $C = D \cup E$
by (*auto simp add*: *C-def D-def E-def*)

lemma *A-card-eq*: *card A* = *nat* ((*int p* − *1*) *div 2*)
by (*auto simp add*: *A-def*)

lemma *inj-on-xa-A*: *inj-on* (λ*x*. *x* ∗ *a*) *A*
using *a-nonzero* **by** (*simp add*: *A-def inj-on-def*)

definition *ResSet* :: *int* ⇒ *int set* ⇒ *bool*
where *ResSet m X* ⟷ (∀ *y1 y2*. *y1* ∈ *X* ∧ *y2* ∈ *X* ∧ [*y1* = *y2*] (*mod m*) ⟶ *y1* = *y2*)

lemma *ResSet-image*:
0 < *m* ⟹ *ResSet m A* ⟹ ∀ *x* ∈ *A*. ∀ *y* ∈ *A*. ([*f x* = *f y*](*mod m*) ⟶ *x* = *y*) ⟹ *ResSet m* (*f* ‘ *A*)
by (*auto simp add*: *ResSet-def*)

lemma *A-res*: *ResSet p A*
using *p-ge-2* **by** (*auto simp add*: *A-def ResSet-def intro*!: *cong-less-imp-eq-int*)

lemma *B-res*: *ResSet p B*
proof −
have ∗: *x* = *y*
if *a*: [*x* ∗ *a* = *y* ∗ *a*] (*mod p*)
and *b*: *0* < *x*
and *c*: *x* ≤ (*int p* − *1*) *div 2*
and *d*: *0* < *y*
and *e*: *y* ≤ (*int p* − *1*) *div 2*
for *x y*

```
  proof −
    from p-a-relprime have ¬ p dvd a
      by (simp add: cong-0-iff)
    with p-prime prime-imp-coprime [of - nat |a|]
    have coprime a (int p)
      by (simp-all add: ac-simps)
    with a cong-mult-rcancel [of a int p x y] have [x = y] (mod p)
      by simp
    with cong-less-imp-eq-int [of x y p] p-minus-one-l
      order-le-less-trans [of x (int p − 1) div 2 p]
      order-le-less-trans [of y (int p − 1) div 2 p]
    show ?thesis
      by (metis b c cong-less-imp-eq-int d e zero-less-imp-eq-int of-nat-0-le-iff)
  qed
  show ?thesis
    using p-ge-2 p-a-relprime p-minus-one-l
   by (metis ∗ A-def A-res B-def GAUSS.ResSet-image GAUSS-axioms greaterThanAt-
Most-iff odd-p odd-pos of-nat-0-less-iff)
qed

lemma SR-B-inj: inj-on (λx. x mod p) B
proof −
  have False
    if a: x ∗ a mod p = y ∗ a mod p
    and b: 0 < x
    and c: x ≤ (int p − 1) div 2
    and d: 0 < y
    and e: y ≤ (int p − 1) div 2
    and f: x ≠ y
    for x y
  proof −
    from a have a′: [x ∗ a = y ∗ a](mod p)
      using cong-def by blast
    from p-a-relprime have ¬p dvd a
      by (simp add: cong-0-iff)
    with p-prime prime-imp-coprime [of - nat |a|]
    have coprime a (int p)
      by (simp-all add: ac-simps)
    with a′ cong-mult-rcancel [of a int p x y]
    have [x = y] (mod p) by simp
    with cong-less-imp-eq-int [of x y p] p-minus-one-l
      order-le-less-trans [of x (int p − 1) div 2 p]
      order-le-less-trans [of y (int p − 1) div 2 p]
    have x = y
      by (metis b c cong-less-imp-eq-int d e zero-less-imp-eq-int of-nat-0-le-iff)
    then show ?thesis
      by (simp add: f)
  qed
  then show ?thesis
```

by (*auto simp add*: *B-def inj-on-def A-def*) *metis*
qed

lemma *nonzero-mod-p*: $0 < x \Longrightarrow x < int\ p \Longrightarrow [x \neq 0](mod\ p)$
for x :: *int*
by (*simp add*: *cong-def*)

lemma *A-ncong-p*: $x \in A \Longrightarrow [x \neq 0](mod\ p)$
by (*rule nonzero-mod-p*) (*auto simp add*: *A-def*)

lemma *A-greater-zero*: $x \in A \Longrightarrow 0 < x$
by (*auto simp add*: *A-def*)

lemma *B-ncong-p*: $x \in B \Longrightarrow [x \neq 0](mod\ p)$
by (*auto simp*: *B-def p-prime p-a-relprime A-ncong-p dest*: *cong-prime-prod-zero-int*)

lemma *B-greater-zero*: $x \in B \Longrightarrow 0 < x$
using *a-nonzero* **by** (*auto simp add*: *B-def A-greater-zero*)

lemma *B-mod-greater-zero*:
$0 < x\ mod\ int\ p$ **if** $x \in B$
proof −
from *that* **have** $x\ mod\ int\ p \neq 0$
using *B-ncong-p cong-def cong-mult-self-left* **by** *blast*
moreover from *that* **have** $0 < x$
by (*rule B-greater-zero*)
then have $0 \leq x\ mod\ int\ p$
by (*auto simp add*: *mod-int-pos-iff intro*: *neq-le-trans*)
ultimately show *?thesis*
by *simp*
qed

lemma *C-greater-zero*: $y \in C \Longrightarrow 0 < y$
by (*auto simp add*: *C-def B-mod-greater-zero*)

lemma *F-subset*: $F \subseteq \{x.\ 0 < x \wedge x \leq ((int\ p - 1)\ div\ 2)\}$
using *p-ge-2* **by** (*auto simp add*: *F-def E-def C-def intro*: *p-odd-int*)

lemma *D-subset*: $D \subseteq \{x.\ 0 < x \wedge x \leq ((p - 1)\ div\ 2)\}$
by (*auto simp add*: *D-def C-greater-zero*)

lemma *F-eq*: $F = \{x.\ \exists\, y \in A.\ (x = p - ((y * a)\ mod\ p) \wedge (int\ p - 1)\ div\ 2 < (y * a)\ mod\ p)\}$
by (*auto simp add*: *F-def E-def D-def C-def B-def A-def*)

lemma *D-eq*: $D = \{x.\ \exists\, y \in A.\ (x = (y * a)\ mod\ p \wedge (y * a)\ mod\ p \leq (int\ p - 1)\ div\ 2)\}$
by (*auto simp add*: *D-def C-def B-def A-def*)

lemma *all-A-relprime*:
  coprime x p **if** $x \in A$
proof −
  from *A-ncong-p* [*OF that*] **have** ¬ *int p dvd x*
    by (*simp add*: *cong-0-iff*)
  with *p-prime* **show** *?thesis*
   by (*metis* (*no-types*) *coprime-commute prime-imp-coprime prime-nat-int-transfer*)
qed

lemma *A-prod-relprime*: *coprime* (*prod id A*) *p*
  by (*auto intro*: *prod-coprime-left all-A-relprime*)

8.3 Relationships Between Gauss Sets

lemma *StandardRes-inj-on-ResSet*: *ResSet m X* ⟹ *inj-on* (λ*b*. *b mod m*) *X*
  by (*auto simp add*: *ResSet-def inj-on-def cong-def*)

lemma *B-card-eq-A*: *card B* = *card A*
  using *finite-A* **by** (*simp add*: *finite-A B-def inj-on-xa-A card-image*)

lemma *B-card-eq*: *card B* = *nat* ((*int p* − *1*) *div 2*)
  by (*simp add*: *B-card-eq-A A-card-eq*)

lemma *F-card-eq-E*: *card F* = *card E*
  using *finite-E* **by** (*simp add*: *F-def card-image*)

lemma *C-card-eq-B*: *card C* = *card B*
proof −
  have *inj-on* (λ*x*. *x mod p*) *B*
    by (*metis SR-B-inj*)
  then show *?thesis*
    by (*metis C-def card-image*)
qed

lemma *D-E-disj*: *D* ∩ *E* = {}
  by (*auto simp add*: *D-def E-def*)

lemma *C-card-eq-D-plus-E*: *card C* = *card D* + *card E*
  by (*auto simp add*: *C-eq card-Un-disjoint D-E-disj finite-D finite-E*)

lemma *C-prod-eq-D-times-E*: *prod id E* ∗ *prod id D* = *prod id C*
 by (*metis C-eq D-E-disj finite-D finite-E inf-commute prod.union-disjoint sup-commute*)

lemma *C-B-zcong-prod*: [*prod id C* = *prod id B*] (*mod p*)
  apply (*auto simp add*: *C-def*)
  apply (*insert finite-B SR-B-inj*)
  apply (*drule prod.reindex* [*of* λ*x*. *x mod int p B id*])
  apply *auto*
  apply (*rule cong-prod*)

apply (*auto simp add*: *cong-def*)
done

lemma *F-Un-D-subset*: $(F \cup D) \subseteq A$
by (*intro Un-least subset-trans* [*OF F-subset*] *subset-trans* [*OF D-subset*]) (*auto simp*: *A-def*)

lemma *F-D-disj*: $(F \cap D) = \{\}$
proof (*auto simp add*: *F-eq D-eq*)
fix *y z* :: *int*
assume $p - (y * a) \bmod p = (z * a) \bmod p$
then have $[(y * a) \bmod p + (z * a) \bmod p = 0] \pmod p$
by (*metis add.commute diff-eq-eq dvd-refl cong-def dvd-eq-mod-eq-0 mod-0*)
moreover have $[y * a = (y * a) \bmod p] \pmod p$
by (*metis cong-def mod-mod-trivial*)
ultimately have $[a * (y + z) = 0] \pmod p$
by (*metis cong-def mod-add-left-eq mod-add-right-eq mult.commute ring-class.ring-distribs*(*1*))
with *p-prime a-nonzero p-a-relprime* **have** *a*: $[y + z = 0] \pmod p$
by (*auto dest*!: *cong-prime-prod-zero-int*)
assume *b*: $y \in A$ **and** *c*: $z \in A$
then have $0 < y + z$
by (*auto simp*: *A-def*)
moreover from *b c p-eq2* **have** $y + z < p$
by (*auto simp*: *A-def*)
ultimately show *False*
by (*metis a nonzero-mod-p*)
qed

lemma *F-Un-D-card*: *card* $(F \cup D) = nat\ ((p - 1)\ div\ 2)$
proof −
have *card* $(F \cup D) =$ *card E* + *card D*
by (*auto simp add*: *finite-F finite-D F-D-disj card-Un-disjoint F-card-eq-E*)
then have *card* $(F \cup D) =$ *card C*
by (*simp add*: *C-card-eq-D-plus-E*)
then show *card* $(F \cup D) = nat\ ((p - 1)\ div\ 2)$
by (*simp add*: *C-card-eq-B B-card-eq*)
qed

lemma *F-Un-D-eq-A*: $F \cup D = A$
using *finite-A F-Un-D-subset A-card-eq F-Un-D-card* **by** (*auto simp add*: *card-seteq*)

lemma *prod-D-F-eq-prod-A*: *prod id D* $*$ *prod id F* = *prod id A*
by (*metis F-D-disj F-Un-D-eq-A Int-commute Un-commute finite-D finite-F prod.union-disjoint*)

lemma *prod-F-zcong*: [*prod id F* = ((−1) ^ (*card E*)) $*$ *prod id E*] (*mod p*)
proof −
have *FE*: *prod id F* = *prod* ((−) *p*) *E*
using *finite-E prod.reindex*[*OF inj-on-diff-left*] **by** (*auto simp add*: *F-def*)
then have $\forall x \in E.\ [(p - x) \bmod p = -x] \pmod p$

by (*metis cong-def minus-mod-self1 mod-mod-trivial*)
then have [*prod* ((λ*x*. *x mod p*) ∘ ((−) *p*)) *E* = *prod* (*uminus*) *E*](*mod p*)
using *finite-E p-ge-2 cong-prod* [*of E* (λ*x*. *x mod p*) ∘ ((−) *p*) *uminus p*]
by *auto*
then have *two*: [*prod id F* = *prod* (*uminus*) *E*](*mod p*)
by (*metis FE cong-cong-mod-int cong-refl cong-prod minus-mod-self1*)
have *prod uminus E* = (−1) ^ *card E* ∗ *prod id E*
using *finite-E* **by** (*induct set*: *finite*) *auto*
with *two* **show** *?thesis*
by *simp*
qed

8.4 Gauss' Lemma

lemma *aux*: *prod id A* ∗ (− *1*) ^ *card E* ∗ *a* ^ *card A* ∗ (− *1*) ^ *card E* = *prod id A* ∗ *a* ^ *card A*
by *auto*

theorem *pre-gauss-lemma*: [*a* ^ *nat*((*int p* − *1*) *div 2*) = (−1) ^ (*card E*)] (*mod p*)
proof −
have [*prod id A* = *prod id F* ∗ *prod id D*](*mod p*)
by (*auto simp*: *prod-D-F-eq-prod-A mult.commute cong del*: *prod.cong-simp*)
then have [*prod id A* = ((−1) ^ (*card E*) ∗ *prod id E*) ∗ *prod id D*] (*mod p*)
by (*rule cong-trans*) (*metis cong-scalar-right prod-F-zcong*)
then have [*prod id A* = ((−1) ^ (*card E*) ∗ *prod id C*)] (*mod p*)
using *finite-D finite-E* **by** (*auto simp add*: *ac-simps C-prod-eq-D-times-E C-eq D-E-disj prod.union-disjoint*)
then have [*prod id A* = ((−1) ^ (*card E*) ∗ *prod id B*)] (*mod p*)
by (*rule cong-trans*) (*metis C-B-zcong-prod cong-scalar-left*)
then have [*prod id A* = ((−1) ^ (*card E*) ∗ *prod id* ((λ*x*. *x* ∗ *a*) ' *A*))] (*mod p*)
by (*simp add*: *B-def*)
then have [*prod id A* = ((−1) ^ (*card E*) ∗ *prod* (λ*x*. *x* ∗ *a*) *A*)] (*mod p*)
by (*simp add*: *inj-on-xa-A prod.reindex*)
moreover have *prod* (λ*x*. *x* ∗ *a*) *A* = *prod* (λ*x*. *a*) *A* ∗ *prod id A*
using *finite-A* **by** (*induct set*: *finite*) *auto*
ultimately have [*prod id A* = ((−1) ^ (*card E*) ∗ (*prod* (λ*x*. *a*) *A* ∗ *prod id A*))] (*mod p*)
by *simp*
then have [*prod id A* = ((−1) ^ (*card E*) ∗ *a* ^ (*card A*) ∗ *prod id A*)](*mod p*)
by (*rule cong-trans*)
(*simp add*: *cong-scalar-left cong-scalar-right finite-A ac-simps*)
then have *a*: [*prod id A* ∗ (−1) ^ (*card E*) =
((−1) ^ (*card E*) ∗ *a* ^ (*card A*) ∗ *prod id A* ∗ (−1) ^ (*card E*))](*mod p*)
by (*rule cong-scalar-right*)
then have [*prod id A* ∗ (−1) ^ (*card E*) = *prod id A* ∗
(−1) ^ (*card E*) ∗ *a* ^ (*card A*) ∗ (−1) ^ (*card E*)](*mod p*)
by (*rule cong-trans*) (*simp add*: *a ac-simps*)
then have [*prod id A* ∗ (−1) ^ (*card E*) = *prod id A* ∗ *a* ^ (*card A*)](*mod p*)

by (*rule cong-trans*) (*simp add*: *aux cong del*: *prod.cong-simp*)
with *A-prod-relprime* **have** [(− *1*) ^ *card E* = *a* ^ *card A*](*mod p*)
by (*metis cong-mult-lcancel*)
then show *?thesis*
by (*simp add*: *A-card-eq cong-sym*)
qed

theorem *gauss-lemma*: *Legendre a p* = (−*1*) ^ (*card E*)
proof −
from *euler-criterion p-prime p-ge-2* **have** [*Legendre a p* = *a*^(*nat* (((*p*) − *1*) *div* *2*))] (*mod p*)
by *auto*
moreover have *int* ((*p* − *1*) *div* *2*) = (*int p* − *1*) *div* *2*
using *p-eq2* **by** *linarith*
then have [*a* ^ *nat* (*int* ((*p* − *1*) *div* *2*)) = *a* ^ *nat* ((*int p* − *1*) *div* *2*)] (*mod* *int p*)
by *force*
ultimately have [*Legendre a p* = (−*1*) ^ (*card E*)] (*mod p*)
using *pre-gauss-lemma cong-trans* **by** *blast*
moreover from *p-a-relprime* **have** *Legendre a p* = *1* ∨ *Legendre a p* = −*1*
by (*auto simp add*: *Legendre-def*)
moreover have (−*1*::*int*) ^ (*card E*) = *1* ∨ (−*1*::*int*) ^ (*card E*) = −*1*
using *neg-one-even-power neg-one-odd-power* **by** *blast*
moreover have [*1* ≠ − *1*] (*mod int p*)
using *cong-iff-dvd-diff* [**where** *'a*=*int*] *nonzero-mod-p*[*of* *2*] *p-odd-int*
by *fastforce*
ultimately show *?thesis*
by (*auto simp add*: *cong-sym*)
qed

end

end

theory *Quadratic-Reciprocity*
imports *Gauss*
begin

The proof is based on Gauss's fifth proof, which can be found at https://www.lehigh.edu/~shw2/q-recip/gauss5.pdf.

locale *QR* =
fixes *p* :: *nat*
fixes *q* :: *nat*
assumes *p-prime*: *prime p*
assumes *p-ge-2*: *2* < *p*
assumes *q-prime*: *prime q*
assumes *q-ge-2*: *2* < *q*
assumes *pq-neq*: *p* ≠ *q*

begin

lemma *odd-p*: *odd p*
using *p-ge-2 p-prime prime-odd-nat* **by** *blast*

lemma *p-ge-0*: *0 < int p*
by (*simp add*: *p-prime prime-gt-0-nat*)

lemma *p-eq2*: *int p = (2 * ((int p − 1) div 2)) + 1*
using *odd-p* **by** *simp*

lemma *odd-q*: *odd q*
using *q-ge-2 q-prime prime-odd-nat* **by** *blast*

lemma *q-ge-0*: *0 < int q*
by (*simp add*: *q-prime prime-gt-0-nat*)

lemma *q-eq2*: *int q = (2 * ((int q − 1) div 2)) + 1*
using *odd-q* **by** *simp*

lemma *pq-eq2*: *int p * int q = (2 * ((int p * int q − 1) div 2)) + 1*
using *odd-p odd-q* **by** *simp*

lemma *pq-coprime*: *coprime p q*
using *pq-neq p-prime primes-coprime-nat q-prime* **by** *blast*

lemma *pq-coprime-int*: *coprime (int p) (int q)*
by (*simp add*: *gcd-int-def pq-coprime*)

lemma *qp-ineq*: *int p * k ≤ (int p * int q − 1) div 2 ⟷ k ≤ (int q − 1) div 2*
proof −
have *2 * int p * k ≤ int p * int q − 1 ⟷ 2 * k ≤ int q − 1*
using *p-ge-0* **by** *auto*
then show *?thesis* **by** *auto*
qed

lemma *QRqp*: *QR q p*
using *QR-def QR-axioms* **by** *simp*

lemma *pq-commute*: *int p * int q = int q * int p*
by *simp*

lemma *pq-ge-0*: *int p * int q > 0*
using *p-ge-0 q-ge-0 mult-pos-pos* **by** *blast*

definition *r = ((p − 1) div 2) * ((q − 1) div 2)*
definition *m = card (GAUSS.E p q)*
definition *n = card (GAUSS.E q p)*

abbreviation *Res k* ≡ {*0* .. *k* − *1*} **for** *k* :: *int*
abbreviation *Res-ge-0 k* ≡ {*0* <.. *k* − *1*} **for** *k* :: *int*
abbreviation *Res-0 k* ≡ {*0*::*int*} **for** *k* :: *int*
abbreviation *Res-l k* ≡ {*0* <.. (*k* − *1*) *div 2*} **for** *k* :: *int*
abbreviation *Res-h k* ≡ {(*k* − *1*) *div 2* <.. *k* − *1*} **for** *k* :: *int*

abbreviation *Sets-pq r0 r1 r2* ≡
{(*x*::*int*). *x* ∈ *r0* (*int p* ∗ *int q*) ∧ *x mod p* ∈ *r1* (*int p*) ∧ *x mod q* ∈ *r2* (*int q*)}

definition *A* = *Sets-pq Res-l Res-l Res-h*
definition *B* = *Sets-pq Res-l Res-h Res-l*
definition *C* = *Sets-pq Res-h Res-h Res-l*
definition *D* = *Sets-pq Res-l Res-h Res-h*
definition *E* = *Sets-pq Res-l Res-0 Res-h*
definition *F* = *Sets-pq Res-l Res-h Res-0*

definition *a* = *card A*
definition *b* = *card B*
definition *c* = *card C*
definition *d* = *card D*
definition *e* = *card E*
definition *f* = *card F*

lemma *Gpq*: *GAUSS p q*
using *p-prime pq-neq p-ge-2 q-prime*
by (*auto simp*: *GAUSS-def cong-iff-dvd-diff dest*: *primes-dvd-imp-eq*)

lemma *Gqp*: *GAUSS q p*
by (*simp add*: *QRqp QR.Gpq*)

lemma *QR-lemma-01*: (λ*x*. *x mod q*) ‘ *E* = *GAUSS.E q p*
proof
have *x* ∈ *E* ⟶ *x mod int q* ∈ *GAUSS.E q p* **if** *x* ∈ *E* **for** *x*
proof −
from *that* **obtain** *k* **where** *k*: *x* = *int p* ∗ *k*
unfolding *E-def* **by** *blast*
from *that E-def* **have** *x* ∈ *Res-l* (*int p* ∗ *int q*)
by *blast*
then have *k* ∈ *GAUSS.A q*
using *Gqp GAUSS.A-def k qp-ineq* **by** (*simp add*: *zero-less-mult-iff*)
then have *x mod q* ∈ *GAUSS.E q p*
using *GAUSS.C-def*[*of q p*] *Gqp k GAUSS.B-def*[*of q p*] *that GAUSS.E-def*[*of q p*]
by (*force simp*: *E-def*)
then show *?thesis* **by** *auto*
qed
then show (λ*x*. *x mod int q*) ‘ *E* ⊆ *GAUSS.E q p*
by *auto*
show *GAUSS.E q p* ⊆ (λ*x*. *x mod q*) ‘ *E*

```
  proof
    fix x
    assume x: x ∈ GAUSS.E q p
    then obtain ka where ka: ka ∈ GAUSS.A q x = (ka * p) mod q
      by (auto simp: Gqp GAUSS.B-def GAUSS.C-def GAUSS.E-def)
    then have ka * p ∈ Res-l (int p * int q)
      using Gqp p-ge-0 qp-ineq by (simp add: GAUSS.A-def Groups.mult-ac(2))
    then show x ∈ (λx. x mod q) ‘ E
      using ka x Gqp q-ge-0 by (force simp: E-def GAUSS.E-def)
  qed
qed

lemma QR-lemma-02: e = n
proof −
  have x = y if x: x ∈ E and y: y ∈ E and mod: x mod q = y mod q for x y
  proof −
    obtain p-inv where p-inv: [int p * p-inv = 1] (mod int q)
      using pq-coprime-int cong-solve-coprime-int by blast
    from x y E-def obtain kx ky where k: x = int p * kx y = int p * ky
      using dvd-def[of p x] by blast
    with x y E-def have 0 < x int p * kx ≤ (int p * int q − 1) div 2
        0 < y int p * ky ≤ (int p * int q − 1) div 2
      using greaterThanAtMost-iff mem-Collect-eq by blast+
    with k have 0 ≤ kx kx < q 0 ≤ ky ky < q
      using qp-ineq by (simp-all add: zero-less-mult-iff)
    moreover from mod k have (p-inv * (p * kx)) mod q = (p-inv * (p * ky))
mod q
      using mod-mult-cong by blast
    then have (p * p-inv * kx) mod q = (p * p-inv * ky) mod q
      by (simp add: algebra-simps)
    then have kx mod q = ky mod q
      using p-inv mod-mult-cong[of p * p-inv q 1]
      by (auto simp: cong-def)
    then have [kx = ky] (mod q)
      unfolding cong-def by blast
    ultimately show ?thesis
      using cong-less-imp-eq-int k by blast
  qed
  then have inj-on (λx. x mod q) E
    by (auto simp: inj-on-def)
  then show ?thesis
    using QR-lemma-01 card-image e-def n-def by fastforce
qed

lemma QR-lemma-03: f = m
proof −
  have F = QR.E q p
    unfolding F-def pq-commute using QRqp QR.E-def[of q p] by fastforce
  then have f = QR.e q p
```

unfolding *f-def* **using** *QRqp QR.e-def*[*of q p*] **by** *presburger*
then show *?thesis*
using *QRqp QR.QR-lemma-02 m-def QRqp QR.n-def* **by** *presburger*
qed

definition *f-1* :: *int* ⇒ *int* × *int*
where *f-1 x* = ((*x mod p*), (*x mod q*))

definition *P-1* :: *int* × *int* ⇒ *int* ⇒ *bool*
where *P-1 res x* ⟷ *x mod p* = *fst res* ∧ *x mod q* = *snd res* ∧ *x* ∈ *Res* (*int p* ∗ *int q*)

definition *g-1* :: *int* × *int* ⇒ *int*
where *g-1 res* = (*THE x*. *P-1 res x*)

lemma *P-1-lemma*:
fixes *res* :: *int* × *int*
assumes *0* ≤ *fst res fst res* < *p 0* ≤ *snd res snd res* < *q*
shows ∃!*x*. *P-1 res x*
proof −
obtain *y k1 k2* **where** *yk*: *y* = *nat* (*fst res*) + *k1* ∗ *p y* = *nat* (*snd res*) + *k2* ∗ *q*
using *chinese-remainder*[*of p q*] *pq-coprime p-ge-0 q-ge-0* **by** *fastforce*
have (*y mod* (*int p* ∗ *int q*)) *mod int p* = *fst res*
using *assms* **by** (*simp add*: *mod-mod-cancel yk*(*1*))
moreover have (*y mod* (*int p* ∗ *int q*)) *mod int q* = *snd res*
using *assms* **by** (*simp add*: *mod-mod-cancel yk*(*2*))
ultimately have *P-1 res* (*int y mod* (*int p* ∗ *int q*))
using *pq-ge-0* **by** (*simp add*: *P-1-def*)
moreover have *a* = *b* **if** *P-1 res a P-1 res b* **for** *a b*
proof −
from *that* **have** *int p* ∗ *int q dvd a* − *b*
using *divides-mult*[*of int p a* − *b int q*] *pq-coprime-int mod-eq-dvd-iff* [*of a - b*]
unfolding *P-1-def* **by** *force*
with *that* **show** *?thesis*
using *dvd-imp-le-int*[*of a* − *b*] **unfolding** *P-1-def* **by** *fastforce*
qed
ultimately show *?thesis* **by** *auto*
qed

lemma *g-1-lemma*:
fixes *res* :: *int* × *int*
assumes *0* ≤ *fst res fst res* < *p 0* ≤ *snd res snd res* < *q*
shows *P-1 res* (*g-1 res*)
using *assms P-1-lemma* [*of res*] *theI'* [*of P-1 res*] *g-1-def*
by *auto*

definition *BuC* = *Sets-pq Res-ge-0 Res-h Res-l*

```
lemma finite-BuC [simp]:
  finite BuC
proof −
  {
    fix p q :: nat
    have finite {x. 0 < x ∧ x < int p ∗ int q}
      by simp
    then have finite {x.
      0 < x ∧
      x < int p ∗ int q ∧
      (int p − 1) div 2
      < x mod int p ∧
      x mod int p < int p ∧
      0 < x mod int q ∧
      x mod int q ≤ (int q − 1) div 2}
      by (auto intro: rev-finite-subset)
  }
  then show ?thesis
    by (simp add: BuC-def)
qed

lemma QR-lemma-04: card BuC = card (Res-h p × Res-l q)
  using card-bij-eq[of f-1 BuC Res-h p × Res-l q g-1]
proof
  show inj-on f-1 BuC
  proof
    fix x y
    assume ∗: x ∈ BuC y ∈ BuC f-1 x = f-1 y
    then have int p ∗ int q dvd x − y
      using f-1-def pq-coprime-int divides-mult[of int p x − y int q]
        mod-eq-dvd-iff[of x - y]
      by auto
    with ∗ show x = y
      using dvd-imp-le-int[of x − y int p ∗ int q] unfolding BuC-def by force
  qed
  show inj-on g-1 (Res-h p × Res-l q)
  proof
    fix x y
    assume ∗: x ∈ Res-h p × Res-l q y ∈ Res-h p × Res-l q g-1 x = g-1 y
    then have 0 ≤ fst x fst x < p 0 ≤ snd x snd x < q
        0 ≤ fst y fst y < p 0 ≤ snd y snd y < q
      using mem-Sigma-iff prod.collapse by fastforce+
    with ∗ show x = y
      using g-1-lemma[of x] g-1-lemma[of y] P-1-def by fastforce
  qed
  show g-1 ‘ (Res-h p × Res-l q) ⊆ BuC
  proof
    fix y
```

```
    assume y ∈ g-1 ‘ (Res-h p × Res-l q)
    then obtain x where x: y = g-1 x x ∈ Res-h p × Res-l q
      by blast
    then have P-1 x y
      using g-1-lemma by fastforce
    with x show y ∈ BuC
      unfolding P-1-def BuC-def mem-Collect-eq using SigmaE prod.sel by fast-
force
  qed
qed (auto simp: finite-subset f-1-def, simp-all add: BuC-def)

lemma QR-lemma-05: card (Res-h p × Res-l q) = r
proof −
  have card (Res-l q) = (q − 1) div 2 card (Res-h p) = (p − 1) div 2
    using p-eq2 by force+
  then show ?thesis
   unfolding r-def using card-cartesian-product[of Res-h p Res-l q] by presburger
qed

lemma QR-lemma-06: b + c = r
proof −
  have B ∩ C = {} finite B finite C B ∪ C = BuC
    unfolding B-def C-def BuC-def by fastforce+
  then show ?thesis
   unfolding b-def c-def using card.empty card-Un-Int QR-lemma-04 QR-lemma-05
by fastforce
qed

definition f-2:: int ⇒ int
  where f-2 x = (int p ∗ int q) − x

lemma f-2-lemma-1: f-2 (f-2 x) = x
  by (simp add: f-2-def)

lemma f-2-lemma-2: [f-2 x = int p − x] (mod p)
  by (simp add: f-2-def cong-iff-dvd-diff)

lemma f-2-lemma-3: f-2 x ∈ S ⟹ x ∈ f-2 ‘ S
  using f-2-lemma-1[of x] image-eqI[of x f-2 f-2 x S] by presburger

lemma QR-lemma-07:
  f-2 ‘ Res-l (int p ∗ int q) = Res-h (int p ∗ int q)
  f-2 ‘ Res-h (int p ∗ int q) = Res-l (int p ∗ int q)
proof −
  have 1: f-2 ‘ Res-l (int p ∗ int q) ⊆ Res-h (int p ∗ int q)
    by (force simp: f-2-def)
  have 2: f-2 ‘ Res-h (int p ∗ int q) ⊆ Res-l (int p ∗ int q)
    using pq-eq2 by (fastforce simp: f-2-def)
  from 2 have 3: Res-h (int p ∗ int q) ⊆ f-2 ‘ Res-l (int p ∗ int q)
```

```
    using f-2-lemma-3 by blast
  from 1 have 4: Res-l (int p * int q) ⊆ f-2 ‘ Res-h (int p * int q)
    using f-2-lemma-3 by blast
  from 1 3 show f-2 ‘ Res-l (int p * int q) = Res-h (int p * int q)
    by blast
  from 2 4 show f-2 ‘ Res-h (int p * int q) = Res-l (int p * int q)
    by blast
qed

lemma QR-lemma-08:
    f-2 x mod p ∈ Res-l p ⟷ x mod p ∈ Res-h p
    f-2 x mod p ∈ Res-h p ⟷ x mod p ∈ Res-l p
  using f-2-lemma-2[of x] cong-def[of f-2 x p - x p] minus-mod-self2[of x p]
    zmod-zminus1-eq-if[of x p] p-eq2
  by auto

lemma QR-lemma-09:
    f-2 x mod q ∈ Res-l q ⟷ x mod q ∈ Res-h q
    f-2 x mod q ∈ Res-h q ⟷ x mod q ∈ Res-l q
  using QRqp QR.QR-lemma-08 f-2-def QR.f-2-def pq-commute by auto

lemma QR-lemma-10: a = c
  unfolding a-def c-def
  apply (rule card-bij-eq[of f-2 A C f-2])
  unfolding A-def C-def
  using QR-lemma-07 QR-lemma-08 QR-lemma-09 apply ((simp add: inj-on-def
f-2-def), blast)+
  apply fastforce+
  done

definition BuD = Sets-pq Res-l Res-h Res-ge-0
definition BuDuF = Sets-pq Res-l Res-h Res

definition f-3 :: int ⇒ int × int
  where f-3 x = (x mod p, x div p + 1)

definition g-3 :: int × int ⇒ int
  where g-3 x = fst x + (snd x - 1) * p

lemma QR-lemma-11: card BuDuF = card (Res-h p × Res-l q)
  using card-bij-eq[of f-3 BuDuF Res-h p × Res-l q g-3]
proof
  show f-3 ‘ BuDuF ⊆ Res-h p × Res-l q
  proof
    fix y
    assume y ∈ f-3 ‘ BuDuF
    then obtain x where x: y = f-3 x x ∈ BuDuF
      by blast
    then have x ≤ int p * (int q - 1) div 2 + (int p - 1) div 2
```

```
    unfolding BuDuF-def using p-eq2 int-distrib(4) by auto
  moreover from x have (int p − 1) div 2 ≤ − 1 + x mod p
    by (auto simp: BuDuF-def)
  moreover have int p ∗ (int q − 1) div 2 = int p ∗ ((int q − 1) div 2)
    by (subst div-mult1-eq) (simp add: odd-q)
  then have p ∗ (int q − 1) div 2 = p ∗ ((int q + 1) div 2 − 1)
    by fastforce
  ultimately have x ≤ p ∗ ((int q + 1) div 2 − 1) − 1 + x mod p
    by linarith
  then have x div p < (int q + 1) div 2 − 1
    using mult.commute[of int p x div p] p-ge-0 div-mult-mod-eq[of x p]
      and mult-less-cancel-left-pos[of p x div p (int q + 1) div 2 − 1]
    by linarith
  moreover from x have 0 < x div p + 1
    using pos-imp-zdiv-neg-iff[of p x] p-ge-0 by (auto simp: BuDuF-def)
  ultimately show y ∈ Res-h p × Res-l q
    using x by (auto simp: BuDuF-def f-3-def)
qed
show inj-on g-3 (Res-h p × Res-l q)
proof
  have ∗: f-3 (g-3 x) = x if x ∈ Res-h p × Res-l q for x
  proof −
    from that have ∗: (fst x + (snd x − 1) ∗ int p) mod int p = fst x
      by force
    from that have (fst x + (snd x − 1) ∗ int p) div int p + 1 = snd x
      by auto
    with ∗ show f-3 (g-3 x) = x
      by (simp add: f-3-def g-3-def)
  qed
  fix x y
  assume x ∈ Res-h p × Res-l q y ∈ Res-h p × Res-l q g-3 x = g-3 y
  from this ∗[of x] ∗[of y] show x = y
    by presburger
qed
show g-3 ‘ (Res-h p × Res-l q) ⊆ BuDuF
proof
  fix y
  assume y ∈ g-3 ‘ (Res-h p × Res-l q)
  then obtain x where x: x ∈ Res-h p × Res-l q and y: y = g-3 x
    by blast
  then have snd x ≤ (int q − 1) div 2
    by force
  moreover have int p ∗ ((int q − 1) div 2) = (int p ∗ int q − int p) div 2
    using int-distrib(4) div-mult1-eq[of int p int q − 1 2] odd-q by fastforce
  ultimately have (snd x) ∗ int p ≤ (int q ∗ int p − int p) div 2
    using mult-right-mono[of snd x (int q − 1) div 2 p] mult.commute[of (int q
− 1) div 2 p]
      pq-commute
    by presburger
```

```
    then have (snd x − 1) ∗ int p ≤ (int q ∗ int p − 1) div 2 − int p
      using p-ge-0 int-distrib(3) by auto
    moreover from x have fst x ≤ int p − 1 by force
    ultimately have fst x + (snd x − 1) ∗ int p ≤ (int p ∗ int q − 1) div 2
      using pq-commute by linarith
    moreover from x have 0 < fst x 0 ≤ (snd x − 1) ∗ p
      by fastforce+
    ultimately show y ∈ BuDuF
      unfolding BuDuF-def using q-ge-0 x g-3-def y by auto
  qed
  show finite BuDuF unfolding BuDuF-def by fastforce
qed (simp add: inj-on-inverseI[of BuDuF g-3] f-3-def g-3-def QR-lemma-05)+

lemma QR-lemma-12: b + d + m = r
proof −
  have B ∩ D = {} finite B finite D B ∪ D = BuD
    unfolding B-def D-def BuD-def by fastforce+
  then have b + d = card BuD
    unfolding b-def d-def using card-Un-Int by fastforce
  moreover have BuD ∩ F = {} finite BuD finite F
    unfolding BuD-def F-def by fastforce+
  moreover have BuD ∪ F = BuDuF
    unfolding BuD-def F-def BuDuF-def
    using q-ge-0 ivl-disj-un-singleton(5)[of 0 int q − 1] by auto
  ultimately show ?thesis
    using QR-lemma-03 QR-lemma-05 QR-lemma-11 card-Un-disjoint[of BuD F]
    unfolding b-def d-def f-def
    by presburger
qed

lemma QR-lemma-13: a + d + n = r
proof −
  have A = QR.B q p
    unfolding A-def pq-commute using QRqp QR.B-def[of q p] by blast
  then have a = QR.b q p
    using a-def QRqp QR.b-def[of q p] by presburger
  moreover have D = QR.D q p
    unfolding D-def pq-commute using QRqp QR.D-def[of q p] by blast
  then have d = QR.d q p
    using d-def  QRqp QR.d-def[of q p] by presburger
  moreover have n = QR.m q p
    using n-def QRqp QR.m-def[of q p] by presburger
  moreover have r = QR.r q p
    unfolding r-def using QRqp QR.r-def[of q p] by auto
  ultimately show ?thesis
    using QRqp QR.QR-lemma-12 by presburger
qed

lemma QR-lemma-14: (−1::int) ^ (m + n) = (−1) ^ r
```

proof −
have $m + n + 2 * d = r$
using *QR-lemma-06 QR-lemma-10 QR-lemma-12 QR-lemma-13* **by** *auto*
then show *?thesis*
using *power-add*[*of* -1::*int* $m + n$ $2 * d$] **by** *fastforce*
qed

lemma *Quadratic-Reciprocity*:
*Legendre p q * Legendre q p* = $(-1::int)$ ^ $((p - 1)$ *div* $2 * ((q - 1)$ *div* $2))$
using *Gpq Gqp GAUSS.gauss-lemma power-add*[*of* -1::*int m n*] *QR-lemma-14*
unfolding *r-def m-def n-def* **by** *auto*

end

theorem *Quadratic-Reciprocity*:
assumes *prime p* $2 < p$ *prime q* $2 < q$ $p \neq q$
shows *Legendre p q * Legendre q p* = $(-1::int)$ ^ $((p - 1)$ *div* $2 * ((q - 1)$ *div* $2))$
using *QR.Quadratic-Reciprocity QR-def assms* **by** *blast*

theorem *Quadratic-Reciprocity-int*:
assumes *prime* $(nat\ p)$ $2 < p$ *prime* $(nat\ q)$ $2 < q$ $p \neq q$
shows *Legendre p q * Legendre q p* = $(-1::int)$ ^ $(nat\ ((p - 1)$ *div* $2 * ((q - 1)$ *div* $2)))$
proof −
from *assms* **have** $0 \leq (p - 1)$ *div* 2 **by** *simp*
moreover have $(nat\ p - 1)$ *div* $2 = nat\ ((p - 1)$ *div* $2)$ $(nat\ q - 1)$ *div* $2 = nat\ ((q - 1)$ *div* $2)$
by *fastforce+*
ultimately have $(nat\ p - 1)$ *div* $2 * ((nat\ q - 1)$ *div* $2) = nat\ ((p - 1)$ *div* $2 * ((q - 1)$ *div* $2))$
using *nat-mult-distrib* **by** *presburger*
moreover have $2 < nat\ p$ $2 < nat\ q$ $nat\ p \neq nat\ q$ *int* $(nat\ p) = p$ *int* $(nat\ q) = q$
using *assms* **by** *linarith+*
ultimately show *?thesis*
using *Quadratic-Reciprocity*[*of nat p nat q*] *assms* **by** *presburger*
qed

end

9 Pocklington's Theorem for Primes

theory *Pocklington*
imports *Residues*
begin

9.1 Lemmas about previously defined terms

lemma *prime-nat-iff''*: *prime* (*p*::*nat*) $\longleftrightarrow$ $p \neq 0 \wedge p \neq 1 \wedge (\forall m.\ 0 < m \wedge m < p \longrightarrow$ *coprime p m*)
proof −
have §: $\bigwedge m.\ [\![0 < p; \forall m.\ 0 < m \wedge m < p \longrightarrow$ *coprime p m*; *m dvd p*; $m \neq p]\!] \Longrightarrow m =$ *Suc 0*
by (*metis One-nat-def coprime-absorb-right dvd-1-iff-1 dvd-nat-bounds nless-le*)
show *?thesis*
by (*auto simp*: *nat-dvd-not-less prime-imp-coprime-nat prime-nat-iff elim*!: §)
qed

lemma *finite-number-segment*: *card* $\{ m.\ 0 < m \wedge m < n \} = n - 1$
proof −
have $\{ m.\ 0 < m \wedge m < n \} = \{1..<n\}$ **by** *auto*
then show *?thesis* **by** *simp*
qed

9.2 Some basic theorems about solving congruences

lemma *cong-solve*:
fixes *n* :: *nat*
assumes *an*: *coprime a n*
shows $\exists x.\ [a * x = b]\ (mod\ n)$
proof (*cases a* = *0*)
case *True*
with *an* **show** *?thesis*
by (*simp add*: *cong-def*)
next
case *False*
from *bezout-add-strong-nat* [*OF this*]
obtain *d x y* **where** *dxy*: *d dvd a d dvd n a* $*$ *x* = *n* $*$ *y* + *d* **by** *blast*
then have *d1*: *d* = *1*
using *assms coprime-common-divisor* [*of a n d*] **by** *simp*
with *dxy*(*3*) **have** *a* $*$ *x* $*$ *b* = (*n* $*$ *y* + *1*) $*$ *b*
by *simp*
then have *a* $*$ (*x* $*$ *b*) = *n* $*$ (*y* $*$ *b*) + *b*
by (*auto simp*: *algebra-simps*)
then have *a* $*$ (*x* $*$ *b*) *mod n* = (*n* $*$ (*y* $*$ *b*) + *b*) *mod n*
by *simp*
then have *a* $*$ (*x* $*$ *b*) *mod n* = *b mod n*
by (*simp add*: *mod-add-left-eq*)
then have [*a* $*$ (*x* $*$ *b*) = *b*] (*mod n*)
by (*simp only*: *cong-def*)
then show *?thesis* **by** *blast*
qed

lemma *cong-solve-unique*:
fixes *n* :: *nat*

```
  assumes an: coprime a n and nz: n ≠ 0
  shows ∃!x. x < n ∧ [a * x = b] (mod n)
proof −
  from cong-solve[OF an] obtain x where x: [a * x = b] (mod n)
    by blast
  let ?P = λx. x < n ∧ [a * x = b] (mod n)
  let ?x = x mod n
  from x have ∗: [a * ?x = b] (mod n)
    by (simp add: cong-def mod-mult-right-eq[of a x n])
  from mod-less-divisor[ of n x] nz ∗ have Px: ?P ?x by simp
  have y = ?x if Py: y < n [a * y = b] (mod n) for y
  proof −
    from Py(2) ∗ have [a * y = a * ?x] (mod n)
      by (simp add: cong-def)
    then have [y = ?x] (mod n)
      by (metis an cong-mult-lcancel-nat)
    with mod-less[OF Py(1)] mod-less-divisor[ of n x] nz
    show ?thesis
      by (simp add: cong-def)
  qed
  with Px show ?thesis by blast
qed

lemma cong-solve-unique-nontrivial:
  fixes p :: nat
  assumes p: prime p
    and pa: coprime p a
    and x0: 0 < x
    and xp: x < p
  shows ∃!y. 0 < y ∧ y < p ∧ [x * y = a] (mod p)
proof −
  from pa have ap: coprime a p
    by (simp add: ac-simps)
  from x0 xp p have px: coprime x p
    by (auto simp add: prime-nat-iff'' ac-simps)
  obtain y where y: y < p [x * y = a] (mod p) ∀ z. z < p ∧ [x * z = a] (mod p)
⟶ z = y
    by (metis cong-solve-unique neq0-conv p prime-gt-0-nat px)
  have y ≠ 0
  proof
    assume y = 0
    with y(2) have p dvd a
      using cong-dvd-iff by auto
    with not-prime-1 p pa show False
      by (auto simp add: gcd-nat.order-iff)
  qed
  with y show ?thesis
    by blast
qed
```

lemma *cong-unique-inverse-prime*:
fixes p :: *nat*
assumes *prime p* **and** $0 < x$ **and** $x < p$
shows $\exists!y.\ 0 < y \wedge y < p \wedge [x * y = 1]\ (mod\ p)$
by (*rule cong-solve-unique-nontrivial*) (*use assms* **in** *simp-all*)

lemma *chinese-remainder-coprime-unique*:
fixes a :: *nat*
assumes *ab*: *coprime a b* **and** *az*: $a \neq 0$ **and** *bz*: $b \neq 0$
and *ma*: *coprime m a* **and** *nb*: *coprime n b*
shows $\exists!x.$ *coprime x* $(a * b) \wedge x < a * b \wedge [x = m]\ (mod\ a) \wedge [x = n]\ (mod\ b)$
proof −
let *?P* $= \lambda x.\ x < a * b \wedge [x = m]\ (mod\ a) \wedge [x = n]\ (mod\ b)$
from *binary-chinese-remainder-unique-nat*[*OF ab az bz*]
obtain x **where** x: $x < a * b$ $[x = m]\ (mod\ a)$ $[x = n]\ (mod\ b)$ $\forall y.$ *?P y* $\longrightarrow y = x$
by *blast*
from *ma nb x* **have** *coprime x a coprime x b*
using *cong-imp-coprime cong-sym* **by** *blast+*
then have *coprime x* $(a*b)$
by *simp*
with x **show** *?thesis*
by *blast*
qed

9.3 Lucas's theorem

lemma *lucas-coprime-lemma*:
fixes n :: *nat*
assumes m: $m \neq 0$ **and** *am*: $[a\hat{}m = 1]\ (mod\ n)$
shows *coprime a n*
proof −
consider $n = 1 \mid n = 0 \mid n > 1$ **by** *arith*
then show *?thesis*
proof *cases*
case *1*
then show *?thesis* **by** *simp*
next
case *2*
with *am m* **show** *?thesis*
by *simp*
next
case *3*
from m **obtain** m' **where** m': $m = Suc\ m'$ **by** (*cases m*) *blast+*
have $d = 1$ **if** d: *d dvd a d dvd n* **for** d
proof −
from *am mod-less*[*OF* ‹$n > 1$›] **have** *am1*: $a\hat{}m\ mod\ n = 1$
by (*simp add*: *cong-def*)

```
    from dvd-mult2[OF d(1), of a^m'] have dam: d dvd a^m
      by (simp add: m')
    from dvd-mod-iff[OF d(2), of a^m] dam am1 show ?thesis
      by simp
  qed
  then show ?thesis
    by (auto intro: coprimeI)
 qed
qed

lemma lucas-weak:
 fixes n :: nat
 assumes n: n ≥ 2
  and an: [a ^ (n − 1) = 1] (mod n)
  and nm: ∀ m. 0 < m ∧ m < n − 1 ⟶ ¬ [a ^ m = 1] (mod n)
 shows prime n
proof (rule totient-imp-prime)
 show totient n = n − 1
 proof (rule ccontr)
  have [a ^ totient n = 1] (mod n)
    by (rule euler-theorem, rule lucas-coprime-lemma [of n − 1]) (use n an in
auto)
  moreover assume totient n ≠ n − 1
  then have totient n > 0 totient n < n − 1
    using ‹n ≥ 2› and totient-less[of n] by simp-all
  ultimately show False
    using nm by auto
 qed
qed (use n in auto)

theorem lucas:
 assumes n2: n ≥ 2 and an1: [a^(n − 1) = 1] (mod n)
  and pn: ∀ p. prime p ∧ p dvd n − 1 ⟶ [a^((n − 1) div p) ≠ 1] (mod n)
 shows prime n
proof−
 from n2 have n01: n ≠ 0 n ≠ 1 n − 1 ≠ 0
  by arith+
 from mod-less-divisor[of n 1] n01 have onen: 1 mod n = 1
  by simp
 from lucas-coprime-lemma[OF n01(3) an1] cong-imp-coprime an1
 have an: coprime a n coprime (a ^ (n − 1)) n
  using ‹n ≥ 2› by simp-all
 have False if H0: ∃ m. 0 < m ∧ m < n − 1 ∧ [a ^ m = 1] (mod n) (is ∃ m. ?P
m)
 proof −
  from H0[unfolded exists-least-iff[of ?P]] obtain m where
    m: 0 < m m < n − 1 [a ^ m = 1] (mod n) ∀ k <m. ¬?P k
    by blast
  have False if nm1: (n − 1) mod m > 0
```

proof −
from *mod-less-divisor*[*OF m*(*1*)] **have** *th0*:(*n* − *1*) *mod m* < *m* **by** *blast*
let *?y* = *a*^((*n* − *1*) *div m* ∗ *m*)
note *mdeq* = *div-mult-mod-eq*[*of* (*n* − *1*) *m*]
have *yn*: *coprime ?y n*
using *an*(*1*) **by** (*cases* (*n* − *Suc 0*) *div m* ∗ *m* = *0*) *auto*
have *?y mod n* = (*a*^*m*)^((*n* − *1*) *div m*) *mod n*
by (*simp add*: *algebra-simps power-mult*)
also have ... = (*a*^*m mod n*)^((*n* − *1*) *div m*) *mod n*
using *power-mod*[*of a*^*m n* (*n* − *1*) *div m*] **by** *simp*
also have ... = *1* **using** *m*(*3*)[*unfolded cong-def onen*] *onen*
by (*metis power-one*)
finally have ∗: *?y mod n* = *1* **.**
have ∗∗: [*?y* ∗ *a* ^((*n* − *1*) *mod m*) = *?y*∗ *1*] (*mod n*)
using *an1*[*unfolded cong-def onen*] *onen*
div-mult-mod-eq[*of* (*n* − *1*) *m*, *symmetric*]
by (*simp add*:*power-add*[*symmetric*] *cong-def* ∗ *del*: *One-nat-def*)
have [*a* ^((*n* − *1*) *mod m*) = *1*] (*mod n*)
by (*metis cong-mult-rcancel-nat mult.commute* ∗∗ *yn*)
with *m*(*4*)[*rule-format*, *OF th0*] *nm1*
less-trans[*OF mod-less-divisor*[*OF m*(*1*), *of n* − *1*] *m*(*2*)] **show** *?thesis*
by *blast*
qed
then have (*n* − *1*) *mod m* = *0* **by** *auto*
then have *mn*: *m dvd n* − *1* **by** *presburger*
then obtain *r* **where** *r*: *n* − *1* = *m* ∗ *r*
unfolding *dvd-def* **by** *blast*
from *n01 r m*(*2*) **have** *r01*: *r* ≠ *0 r* ≠ *1* **by** *auto*
obtain *p* **where** *p*: *prime p p dvd r*
by (*metis prime-factor-nat r01*(*2*))
then have *th*: *prime p* ∧ *p dvd n* − *1*
unfolding *r* **by** (*auto intro*: *dvd-mult*)
from *r* **have** (*a* ^((*n* − *1*) *div p*)) *mod n* = (*a*^(*m*∗*r div p*)) *mod n*
by (*simp add*: *power-mult*)
also have ... = (*a*^(*m*∗(*r div p*))) *mod n*
using *div-mult1-eq*[*of m r p*] *p*(*2*)[*unfolded dvd-eq-mod-eq-0*] **by** *simp*
also have ... = ((*a*^*m*)^(*r div p*)) *mod n*
by (*simp add*: *power-mult*)
also have ... = ((*a*^*m mod n*)^(*r div p*)) *mod n*
using *power-mod* **..**
also from *m*(*3*) *onen* **have** ... = *1*
by (*simp add*: *cong-def*)
finally have [(*a* ^((*n* − *1*) *div p*))= *1*] (*mod n*)
using *onen* **by** (*simp add*: *cong-def*)
with *pn th* **show** *?thesis* **by** *blast*
qed
then have ∀ *m*. *0* < *m* ∧ *m* < *n* − *1* ⟶ ¬ [*a* ^*m* = *1*] (*mod n*)
by *blast*
then show *?thesis* **by** (*rule lucas-weak*[*OF n2 an1*])

qed

9.4 Definition of the order of a number mod n

definition *ord n a* = (*if coprime n a then Least* (λ*d*. *d* > *0* ∧ [*a* ^*d* = *1*] (*mod n*)) *else 0*)

This has the expected properties.

lemma *coprime-ord*:
fixes *n*::*nat*
assumes *coprime n a*
shows *ord n a* > *0* ∧ [*a* ^(*ord n a*) = *1*] (*mod n*) ∧ (∀ *m*. *0* < *m* ∧ *m* < *ord n a* ⟶ [*a*^ *m* ≠ *1*] (*mod n*))
proof−
let *?P* = λ*d*. *0* < *d* ∧ [*a* ^ *d* = *1*] (*mod n*)
from *bigger-prime*[*of a*] **obtain** *p* **where** *p*: *prime p a* < *p*
by *blast*
from *assms* **have** *o*: *ord n a* = *Least ?P*
by (*simp add*: *ord-def*)
have *ex*: ∃ *m*>*0*. *?P m*
proof (*cases n* ≥ *2*)
case *True*
moreover from *assms* **have** *coprime a n*
by (*simp add*: *ac-simps*)
then have [*a* ^ *totient n* = *1*] (*mod n*)
by (*rule euler-theorem*)
ultimately show *?thesis*
by (*auto intro*: *exI* [**where** *x* = *totient n*])
next
case *False*
then have *n* = *0* ∨ *n* = *1*
by *auto*
with *assms* **show** *?thesis*
by *auto*
qed
from *exists-least-iff'*[*of ?P*] *ex assms* **show** *?thesis*
unfolding *o*[*symmetric*] **by** *auto*
qed

With the special value *0* for non-coprime case, it's more convenient.

lemma *ord-works*: [*a* ^ (*ord n a*) = *1*] (*mod n*) ∧ (∀ *m*. *0* < *m* ∧ *m* < *ord n a* ⟶ ¬ [*a*^ *m* = *1*] (*mod n*))
for *n* :: *nat*
by (*cases coprime n a*) (*use coprime-ord*[*of n a*] **in** ‹*auto simp add*: *ord-def cong-def*›)

lemma *ord*: [*a*^(*ord n a*) = *1*] (*mod n*)
for *n* :: *nat*
using *ord-works* **by** *blast*

lemma *ord-minimal*: $0 < m \Longrightarrow m < ord\ n\ a \Longrightarrow \neg\ [a \hat{} m = 1]\ (mod\ n)$
for $n :: nat$
using *ord-works* **by** *blast*

lemma *ord-eq-0*: $ord\ n\ a = 0 \longleftrightarrow \neg\ coprime\ n\ a$
for $n :: nat$
by (*cases coprime n a*) (*simp add*: *coprime-ord*, *simp add*: *ord-def*)

lemma *divides-rexp*: $x\ dvd\ y \Longrightarrow x\ dvd\ (y \hat{} Suc\ n)$
for $x\ y :: nat$
by (*simp add*: *dvd-mult2*[*of x y*])

lemma *ord-divides*:$[a \hat{} d = 1]\ (mod\ n) \longleftrightarrow ord\ n\ a\ dvd\ d$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
for $n :: nat$
proof
assume *?rhs*
then obtain k **where** $d = ord\ n\ a * k$
unfolding *dvd-def* **by** *blast*
then have $[a \hat{} d = (a \hat{} (ord\ n\ a)\ mod\ n) \hat{} k]\ (mod\ n)$
by (*simp add* : *cong-def power-mult power-mod*)
also have $[(a \hat{} (ord\ n\ a)\ mod\ n) \hat{} k = 1]\ (mod\ n)$
using *ord*[*of a n*, *unfolded cong-def*]
by (*simp add*: *cong-def power-mod*)
finally show *?lhs* **.**
next
assume *?lhs*
show *?rhs*
proof (*cases coprime n a*)
case *prem*: *False*
then have *o*: $ord\ n\ a = 0$ **by** (*simp add*: *ord-def*)
show *?thesis*
proof (*cases d*)
case *0*
with *o prem* **show** *?thesis* **by** (*simp add*: *cong-def*)
next
case (*Suc d'*)
then have *d0*: $d \neq 0$ **by** *simp*
from *prem* **obtain** p **where** p: $p\ dvd\ n\ p\ dvd\ a\ p \neq 1$
by (*auto elim*: *not-coprimeE*)
from ‹*?lhs*› **obtain** *q1 q2* **where** *q12*: $a \hat{} d + n * q1 = 1 + n * q2$
using *prem d0 lucas-coprime-lemma*
by (*auto elim*: *not-coprimeE simp add*: *ac-simps*)
then have $a \hat{} d + n * q1 - n * q2 = 1$ **by** *simp*
with *dvd-diff-nat* [*OF dvd-add* [*OF divides-rexp*]] *dvd-mult2 Suc p* **have** *p dvd 1*
by *metis*
with *p*(*3*) **have** *False* **by** *simp*

```
    then show ?thesis ..
  qed
 next
  case H: True
  let ?o = ord n a
  let ?q = d div ord n a
  let ?r = d mod ord n a
  have eqo: [(a^?o)^?q = 1] (mod n)
   using cong-pow ord-works by fastforce
  from H have onz: ?o ≠ 0 by (simp add: ord-eq-0)
  then have opos: ?o > 0 by simp
  from div-mult-mod-eq[of d ord n a] ‹?lhs›
  have [a^(?o*?q + ?r) = 1] (mod n)
   by (simp add: cong-def mult.commute)
  then have [(a^?o)^?q * (a^?r) = 1] (mod n)
   by (simp add: cong-def power-mult[symmetric] power-add[symmetric])
  then have th: [a^?r = 1] (mod n)
   using eqo mod-mult-left-eq[of (a^?o)^?q a^?r n]
   by (simp add: cong-def del: One-nat-def) (metis mod-mult-left-eq nat-mult-1)
  show ?thesis
  proof (cases ?r = 0)
   case True
   then show ?thesis by (simp add: dvd-eq-mod-eq-0)
  next
   case False
   with mod-less-divisor[OF opos, of d] have r0o:?r >0 ∧ ?r < ?o by simp
   from conjunct2[OF ord-works[of a n], rule-format, OF r0o] th
   show ?thesis by blast
  qed
 qed
qed

lemma order-divides-totient:
 ord n a dvd totient n if coprime n a
 using that euler-theorem [of a n]
 by (simp add: ord-divides [symmetric] ac-simps)

lemma order-divides-expdiff:
 fixes n::nat and a::nat assumes na: coprime n a
 shows [a^d = a^e] (mod n) ⟷ [d = e] (mod (ord n a))
proof −
 have th: [a^d = a^e] (mod n) ⟷ [d = e] (mod (ord n a))
  if na: coprime n a and ed: (e::nat) ≤ d
  for n a d e :: nat
 proof −
  from na ed have ∃ c. d = e + c by presburger
  then obtain c where c: d = e + c ..
  from na have an: coprime a n
   by (simp add: ac-simps)
```

```
    then have aen: coprime (a ^ e) n
      by (cases e > 0) simp-all
    from an have acn: coprime (a ^ c) n
      by (cases c > 0) simp-all
    from c have [a^d = a^e] (mod n) ⟷ [a^(e + c) = a^(e + 0)] (mod n)
      by simp
    also have ... ⟷ [a^e* a^c = a^e *a^0] (mod n) by (simp add: power-add)
    also have  ... ⟷ [a ^ c = 1] (mod n)
      using cong-mult-lcancel-nat [OF aen, of a^c a^0] by simp
    also have ... ⟷ ord n a dvd c
      by (simp only: ord-divides)
    also have ... ⟷ [e + c = e + 0] (mod ord n a)
      by (auto simp add: cong-altdef-nat)
    finally show ?thesis
      using c by simp
  qed
  consider e ≤ d | d ≤ e by arith
  then show ?thesis
  proof cases
    case 1
    with na show ?thesis by (rule th)
  next
    case 2
    from th[OF na this] show ?thesis
      by (metis cong-sym)
  qed
qed

lemma ord-not-coprime [simp]: ¬coprime n a ⟹ ord n a = 0
  by (simp add: ord-def)

lemma ord-1 [simp]: ord 1 n = 1
proof −
  have (LEAST k. k > 0) = (1 :: nat)
    by (rule Least-equality) auto
  thus ?thesis by (simp add: ord-def)
qed

lemma ord-1-right [simp]: ord (n::nat) 1 = 1
  using ord-divides[of 1 1 n] by simp

lemma ord-Suc-0-right [simp]: ord (n::nat) (Suc 0) = 1
  using ord-divides[of 1 1 n] by simp

lemma ord-0-nat [simp]: ord 0 (n :: nat) = (if n = 1 then 1 else 0)
proof −
  have (LEAST k. k > 0) = (1 :: nat)
    by (rule Least-equality) auto
  thus ?thesis by (auto simp: ord-def)
```

qed

lemma *ord-0-right-nat* [*simp*]: *ord* (*n* :: *nat*) *0* = (*if n = 1 then 1 else 0*)
proof −
have (*LEAST k*. *k* > *0*) = (*1* :: *nat*)
by (*rule Least-equality*) *auto*
thus *?thesis* **by** (*auto simp*: *ord-def*)
qed

lemma *ord-divides′*: [*a* ^ *d* = *Suc 0*] (*mod n*) = (*ord n a dvd d*)
using *ord-divides*[*of a d n*] **by** *simp*

lemma *ord-Suc-0* [*simp*]: *ord* (*Suc 0*) *n* = *1*
using *ord-1*[**where** *′a* = *nat*] **by** (*simp del*: *ord-1*)

lemma *ord-mod* [*simp*]: *ord n* (*k mod n*) = *ord n k*
by (*cases n* = *0*) (*auto simp add*: *ord-def cong-def power-mod*)

lemma *ord-gt-0-iff* [*simp*]: *ord* (*n*::*nat*) *x* > *0* ⟷ *coprime n x*
using *ord-eq-0*[*of n x*] **by** *auto*

lemma *ord-eq-Suc-0-iff*: *ord n* (*x*::*nat*) = *Suc 0* ⟷ [*x* = *1*] (*mod n*)
using *ord-divides*[*of x 1 n*] **by** (*auto simp*: *ord-divides′*)

lemma *ord-cong*:
assumes [*k1* = *k2*] (*mod n*)
shows *ord n k1* = *ord n k2*
proof −
have *ord n* (*k1 mod n*) = *ord n* (*k2 mod n*)
by (*simp only*: *assms*[*unfolded cong-def*])
thus *?thesis* **by** *simp*
qed

lemma *ord-nat-code* [*code-unfold*]:
ord n a =
(*if n* = *0 then if a* = *1 then 1 else 0 else*
if coprime n a then Min (*Set.filter* (λ*k*. [*a* ^ *k* = *1*] (*mod n*)) {*0*<..*n*}) *else*
0)
proof (*cases coprime n a* ∧ *n* > *0*)
case *True*
define *A* **where** *A* = {*k*∈{*0*<..*n*}. [*a* ^ *k* = *1*] (*mod n*)}
define *k* **where** *k* = (*LEAST k*. *k* > *0* ∧ [*a* ^ *k* = *1*] (*mod n*))
have *totient*: *totient n* ∈ *A*
using *euler-theorem*[*of a n*] *True*
by (*auto simp*: *A-def coprime-commute intro*!: *Nat.gr0I totient-le*)
moreover have *finite A* **by** (*auto simp*: *A-def*)
ultimately have ∗: *Min A* ∈ *A* **and** ∀ *y*. *y* ∈ *A* ⟶ *Min A* ≤ *y*
by (*auto intro*: *Min-in*)

```
  have k > 0 ∧ [a ^ k = 1] (mod n)
    unfolding k-def by (rule LeastI[of - totient n]) (use totient in ‹auto simp:
A-def›)
  moreover have k ≤ totient n
    unfolding k-def by (intro Least-le) (use totient in ‹auto simp: A-def›)
  ultimately have k ∈ A using totient-le[of n] by (auto simp: A-def)
  hence Min A ≤ k by (intro Min-le) (auto simp: ‹finite A›)
  moreover from ∗ have k ≤ Min A
    unfolding k-def by (intro Least-le) (auto simp: A-def)
  ultimately show ?thesis using True
    by (simp add: ord-def k-def A-def)
qed auto

theorem ord-modulus-mult-coprime:
  fixes x :: nat
  assumes coprime m n
  shows   ord (m ∗ n) x = lcm (ord m x) (ord n x)
proof (intro dvd-antisym)
  have [x ^ lcm (ord m x) (ord n x) = 1] (mod (m ∗ n))
    using assms by (intro coprime-cong-mult-nat assms) (auto simp: ord-divides′)
  thus ord (m ∗ n) x dvd lcm (ord m x) (ord n x)
    by (simp add: ord-divides′)
next
  show lcm (ord m x) (ord n x) dvd ord (m ∗ n) x
  proof (intro lcm-least)
    show ord m x dvd ord (m ∗ n) x
      using cong-modulus-mult-nat[of x ^ ord (m ∗ n) x 1 m n] assms
      by (simp add: ord-divides′)
    show ord n x dvd ord (m ∗ n) x
      using cong-modulus-mult-nat[of x ^ ord (m ∗ n) x 1 n m] assms
      by (simp add: ord-divides′ mult.commute)
  qed
qed

corollary ord-modulus-prod-coprime:
  assumes finite A ⋀i j. i ∈ A ⟹ j ∈ A ⟹ i ≠ j ⟹ coprime (f i) (f j)
  shows   ord (∏ i∈A. f i :: nat) x = (LCM i∈A. ord (f i) x)
  using assms by (induction A rule: finite-induct)
          (simp, simp, subst ord-modulus-mult-coprime, auto intro!: prod-coprime-right)

lemma ord-power-aux:
  fixes m x k a :: nat
  defines l ≡ ord m a
  shows   ord m (a ^ k) ∗ gcd k l = l
proof (rule dvd-antisym)
  have [a ^ lcm k l = 1] (mod m)
    unfolding ord-divides by (simp add: l-def)
  also have lcm k l = k ∗ (l div gcd k l)
    by (simp add: lcm-nat-def div-mult-swap)
```

finally have *ord m (a ^ k) dvd l div gcd k l*
unfolding *ord-divides* [*symmetric*] **by** (*simp add*: *power-mult* [*symmetric*])
thus *ord m (a ^ k) * gcd k l dvd l*
by (*cases l = 0*) (*auto simp*: *dvd-div-iff-mult*)

have [*(a ^ k) ^ ord m (a ^ k) = 1*] (*mod m*)
by (*rule ord*)
also have *(a ^ k) ^ ord m (a ^ k) = a ^ (k * ord m (a ^ k))*
by (*simp add*: *power-mult*)
finally have *ord m a dvd k * ord m (a ^ k)*
by (*simp add*: *ord-divides'*)
hence *l dvd gcd (k * ord m (a ^ k)) (l * ord m (a ^ k))*
by (*intro gcd-greatest dvd-triv-left*) (*auto simp*: *l-def ord-divides'*)
also have *gcd (k * ord m (a ^ k)) (l * ord m (a ^ k)) = ord m (a ^ k) * gcd k l*
by (*subst gcd-mult-distrib-nat*) (*auto simp*: *mult-ac*)
finally show *l dvd ord m (a ^ k) * gcd k l* **.**
qed

theorem *ord-power*: *coprime m a ⟹ ord m (a ^ k :: nat) = ord m a div gcd k (ord m a)*
using *ord-power-aux*[*of m a k*] **by** (*metis div-mult-self-is-m gcd-pos-nat ord-eq-0*)

lemma *inj-power-mod*:
assumes *coprime n (a :: nat)*
shows *inj-on (λk. a ^ k mod n) {..<ord n a}*
proof
fix *k l* **assume** *: *k ∈ {..<ord n a} l ∈ {..<ord n a} a ^ k mod n = a ^ l mod n*
have *k = l* **if** *k < l l < ord n a* [*a ^ k = a ^ l*] (*mod n*) **for** *k l*
proof −
have *l = k + (l − k)* **using** *that* **by** *simp*
also have *a ^ ... = a ^ k * a ^ (l − k)*
by (*simp add*: *power-add*)
also have [*... = a ^ l * a ^ (l − k)*] (*mod n*)
using *that* **by** (*intro cong-mult*) *auto*
finally have [*a ^ l * a ^ (l − k) = a ^ l * 1*] (*mod n*)
by (*simp add*: *cong-sym-eq*)
with *assms* **have** [*a ^ (l − k) = 1*] (*mod n*)
by (*subst* (*asm*) *cong-mult-lcancel-nat*) (*auto simp*: *coprime-commute*)
hence *ord n a dvd l − k*
by (*simp add*: *ord-divides'*)
from *dvd-imp-le*[*OF this*] **and** ‹*l < ord n a*› **have** *l − k = 0*
by (*cases l − k = 0*) *auto*
with ‹*k < l*› **show** *k = l* **by** *simp*
qed
from *this*[*of k l*] **and** *this*[*of l k*] **and** * **show** *k = l*
by (*cases k l rule*: *linorder-cases*) (*auto simp*: *cong-def*)
qed

lemma *ord-eq-2-iff*: *ord n (x :: nat) = 2 ⟷ [x ≠ 1] (mod n) ∧ [x^2 = 1] (mod*

```
n)
proof
  assume x: [x ≠ 1] (mod n) ∧ [x^2 = 1] (mod n)
  hence coprime n x
    by (metis coprime-commute lucas-coprime-lemma zero-neq-numeral)
  with x have ord n x dvd 2 ord n x ≠ 1 ord n x > 0
    by (auto simp: ord-divides' ord-eq-Suc-0-iff)
  thus ord n x = 2 by (auto dest!: dvd-imp-le simp del: ord-gt-0-iff)
qed (use ord-divides[of - 2] ord-divides[of - 1] in auto)

lemma square-mod-8-eq-1-iff: [x^2 = 1] (mod 8) ⟷ odd (x :: nat)
proof −
  have [x^2 = 1] (mod 8) ⟷ ((x mod 8)^2 mod 8 = 1)
    by (simp add: power-mod cong-def)
  also have ... ⟷ x mod 8 ∈ {1, 3, 5, 7}
  proof
    assume x: (x mod 8)^2 mod 8 = 1
    have x mod 8 ∈ {..<8} by simp
    also have {..<8} = {0, 1, 2, 3, 4, 5, 6, 7::nat}
      by (simp add: lessThan-nat-numeral lessThan-Suc insert-commute)
    finally have x-cases: x mod 8 ∈ {0, 1, 2, 3, 4, 5, 6, 7} .
    from x have x mod 8 ∉ {0, 2, 4, 6}
      using x by (auto intro: Nat.gr0I)
    with x-cases show x mod 8 ∈ {1, 3, 5, 7} by simp
  qed auto
  also have ... ⟷ odd (x mod 8)
    by (auto elim!: oddE)
  also have ... ⟷ odd x
    by presburger
  finally show ?thesis .
qed

lemma ord-twopow-aux:
  assumes k ≥ 3 and odd (x :: nat)
  shows   [x ^ (2 ^ (k − 2)) = 1] (mod (2 ^ k))
  using assms(1)
proof (induction k rule: dec-induct)
  case base
  from assms have [x^2 = 1] (mod 8)
    by (subst square-mod-8-eq-1-iff) auto
  thus ?case by simp
next
  case (step k)
  define k' where k' = k − 2
  have k: k = Suc (Suc k')
    using ‹k ≥ 3› by (simp add: k'-def)
  from ‹k ≥ 3› have 2 * k ≥ Suc k by presburger

  from ‹odd x› have x > 0 by (intro Nat.gr0I) auto
```

```
  from step.IH have 2 ^ k dvd (x ^ (2 ^ (k - 2)) - 1)
    by (rule cong-to-1-nat)
  then obtain t where x ^ (2 ^ (k - 2)) - 1 = t * 2 ^ k
    by auto
  hence x ^ (2 ^ (k - 2)) = t * 2 ^ k + 1
   by (metis ‹0 < x› add.commute add-diff-inverse-nat less-one neq0-conv power-eq-0-iff)
  hence (x ^ (2 ^ (k - 2))) ^ 2 = (t * 2 ^ k + 1) ^ 2
    by (rule arg-cong)
  hence [(x ^ (2 ^ (k - 2))) ^ 2 = (t * 2 ^ k + 1) ^ 2] (mod (2 ^ Suc k))
    by simp
  also have (x ^ (2 ^ (k - 2))) ^ 2 = x ^ (2 ^ (k - 1))
    by (simp-all add: power-even-eq[symmetric] power-mult k )
  also have (t * 2 ^ k + 1) ^ 2 = t^2 * 2 ^ (2 * k) + t * 2 ^ Suc k + 1
    by (subst power2-eq-square)
       (auto simp: algebra-simps k power2-eq-square[of t]
                   power-even-eq[symmetric] power-add [symmetric])
  also have [... = 0 + 0 + 1] (mod 2 ^ Suc k)
    using ‹2 * k ≥ Suc k›
    by (intro cong-add)
     (auto simp: cong-0-iff intro: dvd-mult[OF le-imp-power-dvd] simp del: power-Suc)
  finally show ?case by simp
qed

lemma ord-twopow-3-5:
  assumes k ≥ 3 x mod 8 ∈ {3, 5 :: nat}
  shows   ord (2 ^ k) x = 2 ^ (k - 2)
  using assms(1)
proof (induction k rule: less-induct)
  have x mod 8 = 3 ∨ x mod 8 = 5 using assms by auto
  hence odd x by presburger
  case (less k)
  from ‹k ≥ 3› consider k = 3 | k = 4 | k ≥ 5 by force
  thus ?case
  proof cases
    case 1
    thus ?thesis using assms
      by (auto simp: ord-eq-2-iff cong-def simp flip: power-mod[of x])
  next
    case 2
    from assms have x mod 8 = 3 ∨ x mod 8 = 5 by auto
    then have x': x mod 16 = 3 ∨ x mod 16 = 5 ∨ x mod 16 = 11 ∨ x mod 16
= 13
      using mod-double-nat [of x 8] by auto
    hence [x ^ 4 = 1] (mod 16) using assms
      by (auto simp: cong-def simp flip: power-mod[of x])
    hence ord 16 x dvd 2^2 by (simp add: ord-divides')
    then obtain l where l: ord 16 x = 2 ^ l l ≤ 2
      by (subst (asm) divides-primepow-nat) auto
```

```
  have [x ^ 2 ≠ 1] (mod 16)
    using x' by (auto simp: cong-def simp flip: power-mod[of x])
  hence ¬ord 16 x dvd 2 by (simp add: ord-divides')
  with l have l = 2
    using le-imp-power-dvd[of l 1 2] by (cases l ≤ 1) auto
  with l show ?thesis by (simp add: ‹k = 4›)
next
  case 3
  define k' where k' = k - 2
  have k': k' ≥ 2 and [simp]: k = Suc (Suc k')
    using 3 by (simp-all add: k'-def)
  have IH: ord (2 ^ k') x = 2 ^ (k' - 2) ord (2 ^ Suc k') x = 2 ^ (k' - 1)
    using less.IH[of k'] less.IH[of Suc k'] 3 by simp-all
  from IH have cong: [x ^ (2 ^ (k' - 2)) = 1] (mod (2 ^ k'))
    by (simp-all add: ord-divides')
  have notcong: [x ^ (2 ^ (k' - 2)) ≠ 1] (mod (2 ^ Suc k'))
  proof
    assume [x ^ (2 ^ (k' - 2)) = 1] (mod (2 ^ Suc k'))
    hence ord (2 ^ Suc k') x dvd 2 ^ (k' - 2)
      by (simp add: ord-divides')
    also have ord (2 ^ Suc k') x = 2 ^ (k' - 1)
      using IH by simp
    finally have k' - 1 ≤ k' - 2
      by (rule power-dvd-imp-le) auto
    with ‹k' ≥ 2› show False by simp
  qed

  have 2 ^ k' + 1 < 2 ^ k' + (2 ^ k' :: nat)
    using one-less-power[of 2::nat k'] k' by (intro add-strict-left-mono) auto
  with cong notcong have cong': x ^ (2 ^ (k' - 2)) mod 2 ^ Suc k' = 1 + 2 ^ k'
    using mod-double-nat [of ‹x ^ 2 ^ (k' - 2)› ‹2 ^ k'›] k' by (auto simp:
cong-def)

  hence x ^ (2 ^ (k' - 2)) mod 2 ^ k = 1 + 2 ^ k' ∨
        x ^ (2 ^ (k' - 2)) mod 2 ^ k = 1 + 2 ^ k' + 2 ^ Suc k'
    using mod-double-nat [of ‹x ^ 2 ^ (k' - 2)› ‹2 ^ Suc k'›] by auto
  hence eq: [x ^ 2 ^ (k' - 1) = 1 + 2 ^ (k - 1)] (mod 2 ^ k)
  proof
    assume *: x ^ (2 ^ (k' - 2)) mod (2 ^ k) = 1 + 2 ^ k'
    have [x ^ (2 ^ (k' - 2)) = x ^ (2 ^ (k' - 2)) mod 2 ^ k] (mod 2 ^ k)
      by simp
    also have [x ^ (2 ^ (k' - 2)) mod (2 ^ k) = 1 + 2 ^ k'] (mod 2 ^ k)
      by (subst *) auto
    finally have [(x ^ 2 ^ (k' - 2)) ^ 2 = (1 + 2 ^ k') ^ 2] (mod 2 ^ k)
      by (rule cong-pow)
    hence [x ^ 2 ^ Suc (k' - 2) = (1 + 2 ^ k') ^ 2] (mod 2 ^ k)
    by (simp add: power-mult [symmetric] power-Suc2 [symmetric] del: power-Suc)
    also have Suc (k' - 2) = k' - 1
      using k' by simp
```

```
    also have (1 + 2 ^ k' :: nat)^2 = 1 + 2 ^ (k - 1) + 2 ^ (2 * k')
      by (subst power2-eq-square) (simp add: algebra-simps flip: power-add)
    also have (2 ^ k :: nat) dvd 2 ^ (2 * k')
      using k' by (intro le-imp-power-dvd) auto
    hence [1 + 2 ^ (k - 1) + 2 ^ (2 * k') = 1 + 2 ^ (k - 1) + (0 :: nat)] (mod
2 ^ k)
      by (intro cong-add) (auto simp: cong-0-iff)
    finally show [x ^ 2 ^ (k' - 1) = 1 + 2 ^ (k - 1)] (mod 2 ^ k)
      by simp
  next
    assume *: x ^ (2 ^ (k' - 2)) mod 2 ^ k = 1 + 2 ^ k' + 2 ^ Suc k'
    have [x ^ (2 ^ (k' - 2)) = x ^ (2 ^ (k' - 2)) mod 2 ^ k] (mod 2 ^ k)
      by simp
    also have [x ^ (2 ^ (k' - 2)) mod (2 ^ k) = 1 + 3 * 2 ^ k'] (mod 2 ^ k)
      by (subst *) auto
    finally have [(x ^ 2 ^ (k' - 2)) ^ 2 = (1 + 3 * 2 ^ k') ^ 2] (mod 2 ^ k)
      by (rule cong-pow)
    hence [x ^ 2 ^ Suc (k' - 2) = (1 + 3 * 2 ^ k') ^ 2] (mod 2 ^ k)
    by (simp add: power-mult [symmetric] power-Suc2 [symmetric] del: power-Suc)
    also have Suc (k' - 2) = k' - 1
      using k' by simp
    also have (1 + 3 * 2 ^ k' :: nat)^2 = 1 + 2 ^ (k - 1) + 2 ^ k + 9 * 2 ^ (2
* k')
      by (subst power2-eq-square) (simp add: algebra-simps flip: power-add)
    also have (2 ^ k :: nat) dvd 9 * 2 ^ (2 * k')
      using k' by (intro dvd-mult le-imp-power-dvd) auto
    hence [1 + 2 ^ (k - 1) + 2 ^ k + 9 * 2 ^ (2 * k') = 1 + 2 ^ (k - 1) + 0
+ (0 :: nat)]
            (mod 2 ^ k)
      by (intro cong-add) (auto simp: cong-0-iff)
    finally show [x ^ 2 ^ (k' - 1) = 1 + 2 ^ (k - 1)] (mod 2 ^ k)
      by simp
  qed

  have notcong': [x ^ 2 ^ (k - 3) ≠ 1] (mod 2 ^ k)
  proof
    assume [x ^ 2 ^ (k - 3) = 1] (mod 2 ^ k)
    hence [x ^ 2 ^ (k' - 1) - x ^ 2 ^ (k' - 1) = 1 + 2 ^ (k - 1) - 1] (mod 2
^ k)
      by (intro cong-diff-nat eq) auto
    hence [2 ^ (k - 1) = (0 :: nat)] (mod 2 ^ k)
      by (simp add: cong-sym-eq)
    hence 2 ^ k dvd 2 ^ (k - 1)
      by (simp add: cong-0-iff)
    hence k ≤ k - 1
      by (rule power-dvd-imp-le) auto
    thus False by simp
  qed
```

```
    have [x ^ 2 ^ (k - 2) = 1] (mod 2 ^ k)
      using ord-twopow-aux[of k x] ‹odd x› ‹k ≥ 3› by simp
    hence ord (2 ^ k) x dvd 2 ^ (k - 2)
      by (simp add: ord-divides')
    then obtain l where l: l ≤ k - 2 ord (2 ^ k) x = 2 ^ l
      using divides-primepow-nat[of 2 ord (2 ^ k) x k - 2] by auto

    from notcong' have ¬ord (2 ^ k) x dvd 2 ^ (k - 3)
      by (simp add: ord-divides')
    with l have l = k - 2
      using le-imp-power-dvd[of l k - 3 2] by (cases l ≤ k - 3) auto
    with l show ?thesis by simp
  qed
qed

lemma ord-4-3 [simp]: ord 4 (3::nat) = 2
proof -
  have [3 ^ 2 = (1 :: nat)] (mod 4)
    by (simp add: cong-def)
  hence ord 4 (3::nat) dvd 2
    by (subst (asm) ord-divides) auto
  hence ord 4 (3::nat) ≤ 2
    by (intro dvd-imp-le) auto
  moreover have ord 4 (3::nat) ≠ 1
    by (auto simp: ord-eq-Suc-0-iff cong-def)
  moreover have ord 4 (3::nat) ≠ 0
    by (auto simp: gcd-non-0-nat coprime-iff-gcd-eq-1)
  ultimately show ord 4 (3 :: nat) = 2
    by linarith
qed

lemma elements-with-ord-1: n > 0 ⟹ {x∈totatives n. ord n x = Suc 0} = {1}
  by (auto simp: ord-eq-Suc-0-iff cong-def totatives-less)

lemma residue-prime-has-primroot:
  fixes p :: nat
  assumes prime p
  shows ∃ a∈totatives p. ord p a = p - 1
proof -
  from residue-prime-mult-group-has-gen[OF assms]
    obtain a where a: a ∈ {1..p-1} {1..p-1} = {a ^ i mod p |i. i ∈ UNIV} by
blast
  from a have coprime p a
    using a assms by (intro prime-imp-coprime) (auto dest: dvd-imp-le)
  with a(1) have a ∈ totatives p by (auto simp: totatives-def coprime-commute)

  have p - 1 = card {1..p-1} by simp
  also have {1..p-1} = {a ^ i mod p |i. i ∈ UNIV} by fact
  also have {a ^ i mod p |i. i ∈ UNIV} = (λi. a ^ i mod p) ‘ {..<ord p a}
```

```
  proof (intro equalityI subsetI)
    fix x assume x ∈ {a ^ i mod p |i. i ∈ UNIV}
    then obtain i where [simp]: x = a ^ i mod p by auto

    have [a ^ i = a ^ (i mod ord p a)] (mod p)
      using ‹coprime p a› by (subst order-divides-expdiff) auto
    hence ∃j. a ^ i mod p = a ^ j mod p ∧ j < ord p a
      using ‹coprime p a› by (intro exI[of - i mod ord p a]) (auto simp: cong-def)
    thus x ∈ (λi. a ^ i mod p) ` {..<ord p a}
      by auto
  qed auto
  also have card ... = ord p a
    using inj-power-mod[OF ‹coprime p a›] by (subst card-image) auto
  finally show ?thesis using ‹a ∈ totatives p›
    by auto
qed
```

9.5 Another trivial primality characterization

```
lemma prime-prime-factor: prime n ⟷ n ≠ 1 ∧ (∀p. prime p ∧ p dvd n ⟶ p
= n)
  (is ?lhs ⟷ ?rhs)
  for n :: nat
proof (cases n = 0 ∨ n = 1)
  case True
  then show ?thesis
    by (metis bigger-prime dvd-0-right not-prime-1 not-prime-0)
next
  case False
  show ?thesis
  proof
    assume prime n
    then show ?rhs
      by (metis not-prime-1 prime-nat-iff)
  next
    assume ?rhs
    with False show prime n
    by (auto simp: prime-nat-iff) (metis One-nat-def prime-factor-nat prime-nat-iff)
  qed
qed

lemma prime-divisor-sqrt: prime n ⟷ n ≠ 1 ∧ (∀d. d dvd n ∧ d^2 ≤ n ⟶ d
= 1)
  for n :: nat
proof −
  consider n = 0 | n = 1 | n ≠ 0 n ≠ 1 by blast
  then show ?thesis
  proof cases
    case 1
```

```
    then show ?thesis by simp
  next
    case 2
    then show ?thesis by simp
  next
    case n: 3
    then have np: n > 1 by arith
    {
      fix d
      assume d: d dvd n d^2 ≤ n
        and H: ∀ m. m dvd n ⟶ m = 1 ∨ m = n
      from H d have d1n: d = 1 ∨ d = n by blast
      then have d = 1
      proof
        assume dn: d = n
        from n have n^2 > n * 1
          by (simp add: power2-eq-square)
        with dn d(2) show ?thesis by simp
      qed
    }
    moreover
    {
      fix d assume d: d dvd n and H: ∀ d'. d' dvd n ∧ d'^2 ≤ n ⟶ d' = 1
      from d n have d ≠ 0
        by (metis dvd-0-left-iff)
      then have dp: d > 0 by simp
      from d[unfolded dvd-def] obtain e where e: n= d*e by blast
      from n dp e have ep:e > 0 by simp
      from dp ep have d^2 ≤ n ∨ e^2 ≤ n
        by (auto simp add: e power2-eq-square mult-le-cancel-left)
      then have d = 1 ∨ d = n
      proof
        assume d^2 ≤ n
        with H[rule-format, of d] d have d = 1 by blast
        then show ?thesis ..
      next
        assume h: e^2 ≤ n
        from e have e dvd n by (simp add: dvd-def mult.commute)
        with H[rule-format, of e] h have e = 1 by simp
        with e have d = n by simp
        then show ?thesis ..
      qed
    }
    ultimately show ?thesis
      unfolding prime-nat-iff using np n(2) by blast
  qed
qed

lemma prime-prime-factor-sqrt:
```

```
  prime (n::nat) ⟷ n ≠ 0 ∧ n ≠ 1 ∧ (∄p. prime p ∧ p dvd n ∧ p^2 ≤ n)
  (is ?lhs ⟷ ?rhs)
proof −
  consider n = 0 | n = 1 | n ≠ 0 n ≠ 1
    by blast
  then show ?thesis
  proof cases
    case 1
    then show ?thesis by (metis not-prime-0)
  next
    case 2
    then show ?thesis by (metis not-prime-1)
  next
    case n: 3
    show ?thesis
    proof
      assume ?lhs
      from this[unfolded prime-divisor-sqrt] n show ?rhs
        by (metis prime-prime-factor)
    next
      assume ?rhs
      {
        fix d
        assume d: d dvd n d^2 ≤ n d ≠ 1
        then obtain p where p: prime p p dvd d
          by (metis prime-factor-nat)
        from d(1) n have dp: d > 0
          by (metis dvd-0-left neq0-conv)
        from mult-mono[OF dvd-imp-le[OF p(2) dp] dvd-imp-le[OF p(2) dp]] d(2)
        have p^2 ≤ n unfolding power2-eq-square by arith
        with ‹?rhs› n p(1) dvd-trans[OF p(2) d(1)] have False
          by blast
      }
      with n prime-divisor-sqrt show ?lhs by auto
    qed
  qed
qed
```

9.6 Pocklington theorem

```
lemma pocklington-lemma:
  fixes p :: nat
  assumes n: n ≥ 2 and nqr: n − 1 = q ∗ r
    and an: [a^(n − 1) = 1] (mod n)
    and aq: ∀p. prime p ∧ p dvd q ⟶ coprime (a ^((n − 1) div p) − 1) n
    and pp: prime p and pn: p dvd n
  shows [p = 1] (mod q)
proof −
  have p01: p ≠ 0 p ≠ 1
```

```
  using pp by (auto intro: prime-gt-0-nat)
obtain k where k: a ^ (q * r) - 1 = n * k
  by (metis an cong-to-1-nat dvd-def nqr)
from pn[unfolded dvd-def] obtain l where l: n = p * l
  by blast
have a0: a ≠ 0
proof
  assume a = 0
  with n have a^(n - 1) = 0
    by (simp add: power-0-left)
  with n an mod-less[of 1 n] show False
    by (simp add: power-0-left cong-def)
qed
with n nqr have aqr0: a ^ (q * r) ≠ 0
  by simp
then have (a ^ (q * r) - 1) + 1  = a ^ (q * r)
  by simp
with k l have a ^ (q * r) = p * l * k + 1
  by simp
then have a ^ (r * q) + p * 0 = 1 + p * (l * k)
  by (simp add: ac-simps)
then have odq: ord p (a^r) dvd q
  unfolding ord-divides[symmetric] power-mult[symmetric]
  by (metis an cong-dvd-modulus-nat mult.commute nqr pn)
from odq[unfolded dvd-def] obtain d where d: q = ord p (a^r) * d
  by blast
have d1: d = 1
proof (rule ccontr)
  assume d1: d ≠ 1
  obtain P where P: prime P P dvd d
    by (metis d1 prime-factor-nat)
  from d dvd-mult[OF P(2), of ord p (a^r)] have Pq: P dvd q by simp
  from aq P(1) Pq have caP:coprime (a^((n - 1) div P) - 1) n by blast
  from Pq obtain s where s: q = P*s unfolding dvd-def by blast
  from P(1) have P0: P ≠ 0
    by (metis not-prime-0)
  from P(2) obtain t where t: d = P*t unfolding dvd-def by blast
  from d s t P0  have s': ord p (a^r) * t = s
    by (metis mult.commute mult-cancel1 mult.assoc)
  have ord p (a^r) * t*r = r * ord p (a^r) * t
    by (metis mult.assoc mult.commute)
  then have exps: a^(ord p (a^r) * t*r) = ((a ^ r) ^ ord p (a^r)) ^ t
    by (simp only: power-mult)
  then have [((a ^ r) ^ ord p (a^r)) ^ t= 1] (mod p)
    by (metis cong-pow ord power-one)
  then have pd0: p dvd a^(ord p (a^r) * t*r) - 1
    by (metis cong-to-1-nat exps)
  from nqr s s' have (n - 1) div P = ord p (a^r) * t*r
    using P0 by simp
```

```
    with caP have coprime (a ^ (ord p (a ^ r) * t * r) - 1) n
      by simp
    with p01 pn pd0 coprime-common-divisor [of - n p] show False
      by auto
  qed
  with d have o: ord p (a^r) = q by simp
  from pp totient-prime [of p] have totient-eq: totient p = p - 1
    by simp
  {
    fix d
    assume d: d dvd p d dvd a d ≠ 1
    from pp[unfolded prime-nat-iff] d have dp: d = p by blast
    from n have n ≠ 0 by simp
    then have False using d dp pn an
      by auto (metis One-nat-def Suc-lessI
          ‹1 < p ∧ (∀m. m dvd p ⟶ m = 1 ∨ m = p)› ‹a ^ (q * r) = p *
l * k + 1› add-diff-cancel-left' dvd-diff-nat dvd-power dvd-triv-left gcd-nat.trans
nat-dvd-not-less nqr zero-less-diff zero-less-one)
  }
  then have cpa: coprime p a
    by (auto intro: coprimeI)
  then have arp: coprime (a ^ r) p
    by (cases r > 0) (simp-all add: ac-simps)
  from euler-theorem [OF arp, simplified ord-divides] o totient-eq have q dvd (p
- 1)
    by simp
  then obtain d where d:p - 1 = q * d
    unfolding dvd-def by blast
  have p ≠ 0
    by (metis p01(1))
  with d have p + q * 0 = 1 + q * d by simp
  then show ?thesis
    by (metis cong-iff-lin-nat mult.commute)
qed

theorem pocklington:
  assumes n: n ≥ 2 and nqr: n - 1 = q * r and sqr: n ≤ q^2
    and an: [a^(n - 1) = 1] (mod n)
    and aq: ∀p. prime p ∧ p dvd q ⟶ coprime (a^((n - 1) div p) - 1) n
  shows prime n
  unfolding prime-prime-factor-sqrt[of n]
proof -
  let ?ths = n ≠ 0 ∧ n ≠ 1 ∧ (∄p. prime p ∧ p dvd n ∧ p^2 ≤ n)
  from n have n01: n ≠ 0 n ≠ 1 by arith+
  {
    fix p
    assume p: prime p p dvd n p^2 ≤ n
    from p(3) sqr have p^(Suc 1) ≤ q^(Suc 1)
      by (simp add: power2-eq-square)
```

```
    then have pq: p ≤ q
      by (metis le0 power-le-imp-le-base)
    from pocklington-lemma[OF n nqr an aq p(1,2)] have *: q dvd p − 1
      by (metis cong-to-1-nat)
    have p − 1 ≠ 0
      using prime-ge-2-nat [OF p(1)] by arith
    with pq * have False
      by (simp add: nat-dvd-not-less)
  }
  with n01 show ?ths by blast
qed
```

Variant for application, to separate the exponentiation.

```
lemma pocklington-alt:
  assumes n: n ≥ 2 and nqr: n − 1 = q * r and sqr: n ≤ q^2
    and an: [a^(n − 1) = 1] (mod n)
    and aq: ∀p. prime p ∧ p dvd q ⟶ (∃b. [a^((n − 1) div p) = b] (mod n) ∧
coprime (b − 1) n)
  shows prime n
proof −
  {
    fix p
    assume p: prime p p dvd q
    from aq[rule-format] p obtain b where b: [a^((n − 1) div p) = b] (mod n)
coprime (b − 1) n
      by blast
    have a0: a ≠ 0
    proof
      assume a0: a = 0
      from n an have [0 = 1] (mod n)
        unfolding a0 power-0-left by auto
      then show False
        using n by (simp add: cong-def dvd-eq-mod-eq-0[symmetric])
    qed
    then have a1: a ≥ 1 by arith
    from one-le-power[OF a1] have ath: 1 ≤ a ^ ((n − 1) div p) .
    have b0: b ≠ 0
    proof
      assume b0: b = 0
      from p(2) nqr have (n − 1) mod p = 0
        by (metis mod-0 mod-mod-cancel mod-mult-self1-is-0)
      with div-mult-mod-eq[of n − 1 p]
      have (n − 1) div p * p= n − 1 by auto
      then have eq: (a^((n − 1) div p))^p = a^(n − 1)
        by (simp only: power-mult[symmetric])
      have p − 1 ≠ 0
        using prime-ge-2-nat [OF p(1)] by arith
      then have pS: Suc (p − 1) = p by arith
      from b have d: n dvd a^((n − 1) div p)
```

unfolding *b0* **by** *auto*
from *divides-rexp*[*OF d*, *of p − 1*] *pS eq cong-dvd-iff* [*OF an*] *n* **show** *False*
by *simp*
qed
then have *b1*: $b \geq 1$ **by** *arith*
from *cong-imp-coprime*[*OF Cong.cong-diff-nat*[*OF cong-sym* [*OF b*(*1*)] *cong-refl* [*of 1*] *b1*]]
ath b1 b nqr
have *coprime* $(a \mathbin{\hat{}} ((n - 1) \mathbin{div} p) - 1)\ n$
by *simp*
}
then have $\forall p.\ prime\ p \wedge p\ dvd\ q \longrightarrow coprime\ (a \mathbin{\hat{}} ((n - 1) \mathbin{div} p) - 1)\ n$
by *blast*
then show *?thesis* **by** (*rule pocklington*[*OF n nqr sqr an*])
qed

9.7 Prime factorizations

definition $primefact\ ps\ n \longleftrightarrow foldr\ (*)\ ps\ 1 = n \wedge (\forall p \in set\ ps.\ prime\ p)$

lemma *primefact*:
fixes *n* :: *nat*
assumes *n*: $n \neq 0$
shows $\exists ps.\ primefact\ ps\ n$
proof −
obtain *xs* **where** *xs*: *mset xs* = *prime-factorization n*
using *ex-mset* [*of prime-factorization n*] **by** *blast*
from *assms* **have** *n* = *prod-mset* (*prime-factorization n*)
by (*simp add*: *prod-mset-prime-factorization*)
also have ... = *prod-mset* (*mset xs*) **by** (*simp add*: *xs*)
also have ... = *foldr* (∗) *xs 1* **by** (*induct xs*) *simp-all*
finally have *foldr* (∗) *xs 1* = *n* **..**
moreover from *xs* **have** $\forall p \in \#mset\ xs.\ prime\ p$ **by** *auto*
ultimately have *primefact xs n* **by** (*auto simp*: *primefact-def*)
then show *?thesis* **..**
qed

lemma *primefact-contains*:
fixes *p* :: *nat*
assumes *pf*: *primefact ps n*
and *p*: *prime p*
and *pn*: *p dvd n*
shows $p \in set\ ps$
using *pf p pn*
proof (*induct ps arbitrary*: *p n*)
case *Nil*
then show *?case* **by** (*auto simp*: *primefact-def*)
next
case (*Cons q qs*)

```
  from Cons.prems[unfolded primefact-def]
  have q: prime q q * foldr (*) qs 1 = n ∀ p ∈set qs. prime p
    and p: prime p p dvd q * foldr (*) qs 1
    by simp-all
  consider p dvd q | p dvd foldr (*) qs 1
    by (metis p prime-dvd-mult-eq-nat)
  then show ?case
  proof cases
    case 1
    with p(1) q(1) have p = q
      unfolding prime-nat-iff by auto
    then show ?thesis by simp
  next
    case prem: 2
    from q(3) have pqs: primefact qs (foldr (*) qs 1)
      by (simp add: primefact-def)
    from Cons.hyps[OF pqs p(1) prem] show ?thesis by simp
  qed
qed

lemma primefact-variant: primefact ps n ⟷ foldr (*) ps 1 = n ∧ list-all prime
ps
  by (auto simp add: primefact-def list-all-iff)
```

Variant of Lucas theorem.

```
lemma lucas-primefact:
  assumes n: n ≥ 2 and an: [a^(n − 1) = 1] (mod n)
    and psn: foldr (*) ps 1 = n − 1
    and psp: list-all (λp. prime p ∧ ¬ [a^((n − 1) div p) = 1] (mod n)) ps
  shows prime n
proof −
  {
    fix p
    assume p: prime p p dvd n − 1 [a ^((n − 1) div p) = 1] (mod n)
    from psn psp have psn1: primefact ps (n − 1)
      by (auto simp add: list-all-iff primefact-variant)
    from p(3) primefact-contains[OF psn1 p(1,2)] psp
    have False by (induct ps) auto
  }
  with lucas[OF n an] show ?thesis by blast
qed
```

Variant of Pocklington theorem.

```
lemma pocklington-primefact:
  assumes n: n ≥ 2 and qrn: q*r = n − 1 and nq2: n ≤ q^2
    and arnb: (a^r) mod n = b and psq: foldr (*) ps 1 = q
    and bqn: (b^q) mod n = 1
    and psp: list-all (λp. prime p ∧ coprime ((b^(q div p)) mod n − 1) n) ps
  shows prime n
```

```
proof −
  from bqn psp qrn
  have bqn: a ^ (n − 1) mod n = 1
    and psp: list-all (λp. prime p ∧ coprime (a^(r ∗(q div p)) mod n − 1) n) ps
    unfolding arnb[symmetric] power-mod
    by (simp-all add: power-mult[symmetric] algebra-simps)
  from n have n0: n > 0 by arith
  from div-mult-mod-eq[of a^(n − 1) n]
    mod-less-divisor[OF n0, of a^(n − 1)]
  have an1: [a ^ (n − 1) = 1] (mod n)
    by (metis bqn cong-def mod-mod-trivial)
  have coprime (a ^ ((n − 1) div p) − 1) n if p: prime p p dvd q for p
  proof −
    from psp psq have pfpsq: primefact ps q
      by (auto simp add: primefact-variant list-all-iff)
    from psp primefact-contains[OF pfpsq p]
    have p′: coprime (a ^ (r ∗ (q div p)) mod n − 1) n
      by (simp add: list-all-iff)
    from p prime-nat-iff have p01: p ≠ 0 p ≠ 1 p = Suc (p − 1)
      by auto
    from div-mult1-eq[of r q p] p(2)
    have eq1: r∗ (q div p) = (n − 1) div p
      unfolding qrn[symmetric] dvd-eq-mod-eq-0 by (simp add: mult.commute)
    have ath: a ≤ b ⟹ a ≠ 0 ⟹ 1 ≤ a ∧ 1 ≤ b for a b :: nat
      by arith
    {
      assume a ^ ((n − 1) div p) mod n = 0
      then obtain s where s: a ^ ((n − 1) div p) = n ∗ s
        by blast
      then have eq0: (a^((n − 1) div p))^p = (n∗s)^p by simp
      from qrn[symmetric] have qn1: q dvd n − 1
        by (auto simp: dvd-def)
      from dvd-trans[OF p(2) qn1] have npp: (n − 1) div p ∗ p = n − 1
        by simp
      with eq0 have a ^ (n − 1) = (n ∗ s) ^ p
        by (simp add: power-mult[symmetric])
      with bqn p01 have 1 = (n ∗ s)^(Suc (p − 1)) mod n
        by simp
      also have ... = 0 by (simp add: mult.assoc)
      finally have False by simp
    }
    then have ∗: a ^ ((n − 1) div p) mod n ≠ 0 by auto
    have [a ^ ((n − 1) div p) mod n = a ^ ((n − 1) div p)] (mod n)
      by (simp add: cong-def)
    with ath[OF mod-less-eq-dividend ∗]
    have [a ^ ((n − 1) div p) mod n − 1 = a ^ ((n − 1) div p) − 1] (mod n)
      by (simp add: cong-diff-nat)
    then show ?thesis
      by (metis cong-imp-coprime eq1 p′)
```

qed
with *pocklington*[*OF n qrn*[*symmetric*] *nq2 an1*] **show** *?thesis*
by *blast*
qed

end

10 Prime powers

theory *Prime-Powers*
imports *Complex-Main HOL−Computational-Algebra.Primes HOL−Library.FuncSet*
begin

definition *aprimedivisor* :: *'a* :: *normalization-semidom* ⇒ *'a* **where**
aprimedivisor q = (*SOME p. prime p* ∧ *p dvd q*)

definition *primepow* :: *'a* :: *normalization-semidom* ⇒ *bool* **where**
primepow n ⟷ (∃ *p k. prime p* ∧ *k* > *0* ∧ *n* = *p* ^ *k*)

definition *primepow-factors* :: *'a* :: *normalization-semidom* ⇒ *'a set* **where**
primepow-factors n = {*x. primepow x* ∧ *x dvd n*}

lemma *primepow-gt-Suc-0*: *primepow n* ⟹ *n* > *Suc 0*
using *one-less-power*[*of p*::*nat* **for** *p*] **by** (*auto simp*: *primepow-def prime-nat-iff*)

lemma
assumes *prime p p dvd n*
shows *prime-aprimedivisor*: *prime* (*aprimedivisor n*)
and *aprimedivisor-dvd*: *aprimedivisor n dvd n*
proof −
from *assms* **have** ∃ *p. prime p* ∧ *p dvd n* **by** *auto*
from *someI-ex*[*OF this*] **show** *prime* (*aprimedivisor n*) *aprimedivisor n dvd n*
unfolding *aprimedivisor-def* **by** (*simp-all add*: *conj-commute*)
qed

lemma
assumes *n* ≠ *0* ¬*is-unit* (*n* :: *'a* :: *factorial-semiring*)
shows *prime-aprimedivisor'*: *prime* (*aprimedivisor n*)
and *aprimedivisor-dvd'*: *aprimedivisor n dvd n*
proof −
from *someI-ex*[*OF prime-divisor-exists*[*OF assms*]]
show *prime* (*aprimedivisor n*) *aprimedivisor n dvd n*
unfolding *aprimedivisor-def* **by** (*simp-all add*: *conj-commute*)
qed

lemma *aprimedivisor-of-prime* [*simp*]:
assumes *prime p*
shows *aprimedivisor p* = *p*
proof −

```
  from assms have ∃ q. prime q ∧ q dvd p by auto
  from someI-ex[OF this, folded aprimedivisor-def] assms show ?thesis
    by (auto intro: primes-dvd-imp-eq)
qed

lemma aprimedivisor-pos-nat: (n::nat) > 1 ⟹ aprimedivisor n > 0
  using aprimedivisor-dvd'[of n] by (auto elim: dvdE intro!: Nat.gr0I)

lemma aprimedivisor-primepow-power:
  assumes primepow n k > 0
  shows   aprimedivisor (n ^ k) = aprimedivisor n
proof −
  from assms obtain p l where l: prime p l > 0 n = p ^ l
    by (auto simp: primepow-def)
  from l assms have ∗: prime (aprimedivisor (n ^ k)) aprimedivisor (n ^ k) dvd
n ^ k
    by (intro prime-aprimedivisor[of p] aprimedivisor-dvd[of p] dvd-power;
        simp add: power-mult [symmetric])+
  from ∗ l have aprimedivisor (n ^ k) dvd p ^ (l ∗ k) by (simp add: power-mult)
  with assms ∗ l have aprimedivisor (n ^ k) dvd p
    by (subst (asm) prime-dvd-power-iff) simp-all
  with l assms have aprimedivisor (n ^ k) = p
     by (intro primes-dvd-imp-eq prime-aprimedivisor l) (auto simp: power-mult
[symmetric])
  moreover from l have aprimedivisor n dvd p ^ l
    by (auto intro: aprimedivisor-dvd simp: prime-gt-0-nat)
  with assms l have aprimedivisor n dvd p
    by (subst (asm) prime-dvd-power-iff) (auto intro!: prime-aprimedivisor simp:
prime-gt-0-nat)
  with l assms have aprimedivisor n = p
    by (intro primes-dvd-imp-eq prime-aprimedivisor l) auto
  ultimately show ?thesis by simp
qed

lemma aprimedivisor-prime-power:
  assumes prime p k > 0
  shows   aprimedivisor (p ^ k) = p
proof −
  from assms have ∗: prime (aprimedivisor (p ^ k)) aprimedivisor (p ^ k) dvd p
^ k
   by (intro prime-aprimedivisor[of p] aprimedivisor-dvd[of p]; simp add: prime-nat-iff)+
  from assms ∗ have aprimedivisor (p ^ k) dvd p
    by (subst (asm) prime-dvd-power-iff) simp-all
  with assms ∗ show aprimedivisor (p ^ k) = p by (intro primes-dvd-imp-eq)
qed

lemma prime-factorization-primepow:
  assumes primepow n
  shows   prime-factorization n =
```

replicate-mset (*multiplicity* (*aprimedivisor n*) *n*) (*aprimedivisor n*)
using *assms*
by (*auto simp*: *primepow-def aprimedivisor-prime-power prime-factorization-prime-power*)

lemma *primepow-decompose*:
fixes *n* :: *'a* :: *factorial-semiring-multiplicative*
assumes *primepow n*
shows *aprimedivisor n* ^ *multiplicity* (*aprimedivisor n*) *n* = *n*
proof −
from *assms* **have** *n* ≠ *0* **by** (*intro notI*) (*auto simp*: *primepow-def*)
hence *n* = *unit-factor n* ∗ *prod-mset* (*prime-factorization n*)
by (*subst prod-mset-prime-factorization*) *simp-all*
also from *assms* **have** *unit-factor n* = *1* **by** (*auto simp*: *primepow-def unit-factor-power*)
also have *prime-factorization n* =
replicate-mset (*multiplicity* (*aprimedivisor n*) *n*) (*aprimedivisor n*)
by (*intro prime-factorization-primepow assms*)
also have *prod-mset* ... = *aprimedivisor n* ^ *multiplicity* (*aprimedivisor n*) *n*
by *simp*
finally show *?thesis* **by** *simp*
qed

lemma *prime-power-not-one*:
assumes *prime p k* > *0*
shows *p* ^ *k* ≠ *1*
proof
assume *p* ^ *k* = *1*
hence *is-unit* (*p* ^ *k*) **by** *simp*
thus *False* **using** *assms* **by** (*simp add*: *is-unit-power-iff*)
qed

lemma *zero-not-primepow* [*simp*]: ¬*primepow 0*
by (*auto simp*: *primepow-def*)

lemma *one-not-primepow* [*simp*]: ¬*primepow 1*
by (*auto simp*: *primepow-def prime-power-not-one*)

lemma *primepow-not-unit* [*simp*]: *primepow p* ⟹ ¬*is-unit p*
by (*auto simp*: *primepow-def is-unit-power-iff*)

lemma *not-primepow-Suc-0-nat* [*simp*]: ¬*primepow* (*Suc 0*)
using *primepow-gt-Suc-0*[*of Suc 0*] **by** *auto*

lemma *primepow-gt-0-nat*: *primepow n* ⟹ *n* > (*0*::*nat*)
using *primepow-gt-Suc-0*[*of n*] **by** *simp*

lemma *unit-factor-primepow*:
fixes *p* :: *'a* :: *factorial-semiring-multiplicative*
shows *primepow p* ⟹ *unit-factor p* = *1*
by (*auto simp*: *primepow-def unit-factor-power*)

lemma *aprimedivisor-primepow*:
assumes *prime p p dvd n primepow* (*n* :: *'a* :: *factorial-semiring-multiplicative*)
shows *aprimedivisor* (*p* ∗ *n*) = *p aprimedivisor n* = *p*
proof −
from *assms* **have** [*simp*]: *n* ≠ *0* **by** *auto*
define *q* **where** *q* = *aprimedivisor n*
with *assms* **have** *q*: *prime q* **by** (*auto simp*: *q-def intro*!: *prime-aprimedivisor*)
from ‹*primepow n*› **have** *n*: *n* = *q* ^ *multiplicity q n*
by (*simp add*: *primepow-decompose q-def*)
have *nz*: *multiplicity q n* ≠ *0*
proof
assume *multiplicity q n* = *0*
with *n* **have** *n'*: *n* = *unit-factor n* **by** *simp*
have *is-unit n* **by** (*subst n'*, *rule unit-factor-is-unit*) (*insert assms*, *auto*)
with *assms* **show** *False* **by** *auto*
qed
with ‹*prime p*› ‹*p dvd n*› *q* **have** *p dvd q*
by (*subst* (*asm*) *n*) (*auto intro*: *prime-dvd-power*)
with ‹*prime p*› *q* **have** *p* = *q* **by** (*intro primes-dvd-imp-eq*)
thus *aprimedivisor n* = *p* **by** (*simp add*: *q-def*)

define *r* **where** *r* = *aprimedivisor* (*p* ∗ *n*)
with *assms* **have** *r*: *r dvd* (*p* ∗ *n*) *prime r* **unfolding** *r-def*
by (*intro aprimedivisor-dvd*[*of p*] *prime-aprimedivisor*[*of p*]; *simp*)+
hence *r dvd q* ^ *Suc* (*multiplicity q n*)
by (*subst* (*asm*) *n*) (*auto simp*: ‹*p* = *q*› *dest*: *dvd-unit-imp-unit*)
with *r* **have** *r dvd q*
by (*auto intro*: *prime-dvd-power-nat simp*: *prime-dvd-mult-iff dest*: *prime-dvd-power*)
with *r q* **have** *r* = *q* **by** (*intro primes-dvd-imp-eq*)
thus *aprimedivisor* (*p* ∗ *n*) = *p* **by** (*simp add*: *r-def* ‹*p* = *q*›)
qed

lemma *power-eq-prime-powerD*:
fixes *p* :: *'a* :: *factorial-semiring*
assumes *prime p n* > *0 x* ^ *n* = *p* ^ *k*
shows ∃ *i*. *normalize x* = *normalize* (*p* ^ *i*)
proof −
have *normalize x* = *normalize* (*p* ^ *multiplicity p x*)
proof (*rule multiplicity-eq-imp-eq*)
fix *q* :: *'a* **assume** *prime q*
from *assms* **have** *multiplicity q* (*x* ^ *n*) = *multiplicity q* (*p* ^ *k*) **by** *simp*
with ‹*prime q*› **and** *assms* **have** *n* ∗ *multiplicity q x* = *k* ∗ *multiplicity q p*
by (*subst* (*asm*) (*1 2*) *prime-elem-multiplicity-power-distrib*) (*auto simp*: *power-0-left*)
with *assms* **and** ‹*prime q*› **show** *multiplicity q x* = *multiplicity q* (*p* ^ *multiplicity p x*)
by (*cases p* = *q*) (*auto simp*: *multiplicity-distinct-prime-power prime-multiplicity-other*)
qed (*insert assms*, *auto simp*: *power-0-left*)

thus *?thesis* **by** *auto*
qed

lemma *primepow-power-iff*:
fixes $p :: 'a :: factorial\text{-}semiring\text{-}multiplicative$
assumes *unit-factor p = 1*
shows *primepow* $(p \hat{} n) \longleftrightarrow$ *primepow p* $\wedge$ $n > 0$
proof *safe*
assume *primepow* $(p \hat{} n)$
hence *n*: $n \neq 0$ **by** (*auto intro*!: *Nat.gr0I*)
thus $n > 0$ **by** *simp*
from *assms* **have** [*simp*]: *normalize p = p*
using *normalize-mult-unit-factor*[*of p*] **by** (*simp only*: *mult.right-neutral*)
from ‹*primepow* $(p \hat{} n)$› **obtain** *q k* **where** $*$: $k > 0$ *prime q p* $\hat{} n = q \hat{} k$
by (*auto simp*: *primepow-def*)
with *power-eq-prime-powerD*[*of q n p k*] *n*
obtain *i* **where** *eq*: *normalize p = normalize* $(q \hat{} i)$ **by** *auto*
with *primepow-not-unit*[*OF* ‹*primepow* $(p \hat{} n)$›] **have** $i \neq 0$
by (*intro notI*) (*simp add*: *normalize-1-iff is-unit-power-iff del*: *primepow-not-unit*)
with ‹*normalize p = normalize* $(q \hat{} i)$› ‹*prime q*› **show** *primepow p*
by (*auto simp*: *normalize-power primepow-def intro*!: *exI*[*of - q*] *exI*[*of - i*])
next
assume *primepow p n* > 0
then obtain *q k* **where** $*$: $k > 0$ *prime q p* $= q \hat{} k$ **by** (*auto simp*: *primepow-def*)
with ‹$n > 0$› **show** *primepow* $(p \hat{} n)$
by (*auto simp*: *primepow-def power-mult intro*!: *exI*[*of - q*] *exI*[*of - k* $*$ *n*])
qed

lemma *primepow-power-iff-nat*:
$p > 0 \Longrightarrow$ *primepow* $(p \hat{} n) \longleftrightarrow$ *primepow* $(p :: nat) \wedge n > 0$
by (*rule primepow-power-iff*) (*simp-all add*: *unit-factor-nat-def*)

lemma *primepow-prime* [*simp*]: *prime n* $\Longrightarrow$ *primepow n*
by (*auto simp*: *primepow-def intro*!: *exI*[*of - n*] *exI*[*of - 1*::*nat*])

lemma *primepow-prime-power* [*simp*]:
prime $(p :: 'a :: factorial\text{-}semiring\text{-}multiplicative) \Longrightarrow$ *primepow* $(p \hat{} n) \longleftrightarrow n > 0$
by (*subst primepow-power-iff*) *auto*

lemma *aprimedivisor-vimage*:
assumes *prime* $(p :: 'a :: factorial\text{-}semiring\text{-}multiplicative)$
shows *aprimedivisor* $-` \{p\} \cap$ *primepow-factors n* $= \{p \hat{} k \mid k.\ k > 0 \wedge p \hat{} k$ *dvd n*$\}$
proof *safe*
fix *q* **assume** *q*: $q \in$ *primepow-factors n*
hence *q'*: $q \neq 0$ $q \neq 1$ **by** (*auto simp*: *primepow-def primepow-factors-def prime-power-not-one*)

```
  let ?n = multiplicity (aprimedivisor q) q
  from q q' have q = aprimedivisor q ^ ?n ∧ ?n > 0 ∧ aprimedivisor q ^ ?n dvd
n
   by (auto simp: primepow-decompose primepow-factors-def prime-multiplicity-gt-zero-iff
         prime-aprimedivisor' prime-imp-prime-elem aprimedivisor-dvd')
  thus ∃k. q = aprimedivisor q ^ k ∧ k > 0 ∧ aprimedivisor q ^ k dvd n ..
next
  fix k :: nat assume k: p ^ k dvd n k > 0
  with assms show p ^ k ∈ aprimedivisor -` {p}
    by (auto simp: aprimedivisor-prime-power)
  with assms k show p ^ k ∈ primepow-factors n
    by (auto simp: primepow-factors-def primepow-def aprimedivisor-prime-power
intro: Suc-leI)
qed

lemma aprimedivisor-nat:
  assumes n ≠ (Suc 0::nat)
  shows   prime (aprimedivisor n) aprimedivisor n dvd n
proof -
  from assms have ∃p. prime p ∧ p dvd n by (intro prime-factor-nat) auto
  from someI-ex[OF this, folded aprimedivisor-def]
    show prime (aprimedivisor n) aprimedivisor n dvd n by blast+
qed

lemma aprimedivisor-gt-Suc-0:
  assumes n ≠ Suc 0
  shows   aprimedivisor n > Suc 0
proof -
  from assms have prime (aprimedivisor n) by (rule aprimedivisor-nat)
  thus aprimedivisor n > Suc 0 by (simp add: prime-nat-iff)
qed

lemma aprimedivisor-le-nat:
  assumes n > Suc 0
  shows   aprimedivisor n ≤ n
proof -
  from assms have aprimedivisor n dvd n by (intro aprimedivisor-nat) simp-all
  with assms show aprimedivisor n ≤ n
    by (intro dvd-imp-le) simp-all
qed

lemma bij-betw-primepows:
  bij-betw (λ(p,k). p ^ Suc k :: 'a :: factorial-semiring-multiplicative)
     (Collect prime × UNIV) (Collect primepow)
proof (rule bij-betwI [where ?g = (λn. (aprimedivisor n, multiplicity (aprimedivisor
n) n - 1))],
       goal-cases)
  case 1
  show (λ(p, k). p ^ Suc k :: 'a) ∈ Collect prime × UNIV → Collect primepow
```

by (*auto intro*!: *primepow-prime-power simp del*: *power-Suc*)
next
case *2*
show *?case*
by (*auto simp*: *primepow-def prime-aprimedivisor*)
next
case (*3 n*)
thus *?case*
by (*auto simp*: *aprimedivisor-prime-power simp del*: *power-Suc*)
next
case (*4 n*)
hence ∗: *0* < *multiplicity* (*aprimedivisor n*) *n*
by (*subst prime-multiplicity-gt-zero-iff*)
(*auto intro*!: *prime-imp-prime-elem aprimedivisor-dvd simp*: *primepow-def prime-aprimedivisor*)
have *aprimedivisor n* ∗ *aprimedivisor n* ^ (*multiplicity* (*aprimedivisor n*) *n* − *Suc 0*) =
aprimedivisor n ^ *Suc* (*multiplicity* (*aprimedivisor n*) *n* − *Suc 0*) **by** *simp*
also from ∗ **have** *Suc* (*multiplicity* (*aprimedivisor n*) *n* − *Suc 0*) =
multiplicity (*aprimedivisor n*) *n*
by (*subst Suc-diff-Suc*) (*auto simp*: *prime-multiplicity-gt-zero-iff*)
also have *aprimedivisor n* ^ ... = *n*
using *4* **by** (*subst primepow-decompose*) *auto*
finally show *?case* **by** *auto*
qed

lemma *primepow-multD*:
assumes *primepow* (*a* ∗ *b* :: *nat*)
shows *a* = *1* ∨ *primepow a b* = *1* ∨ *primepow b*
proof −
from *assms* **obtain** *p k* **where** *k*: *k* > *0 a* ∗ *b* = *p* ^ *k prime p*
unfolding *primepow-def* **by** *auto*
then obtain *i j* **where** *a* = *p* ^ *i b* = *p* ^ *j*
using *prime-power-mult-nat*[*of p a b*] **by** *blast*
with ‹*prime p*› **show** *a* = *1* ∨ *primepow a b* = *1* ∨ *primepow b* **by** *auto*
qed

lemma *primepow-mult-aprimedivisorI*:
assumes *primepow* (*n* :: *'a* :: *factorial-semiring-multiplicative*)
shows *primepow* (*aprimedivisor n* ∗ *n*)
by (*subst* (*2*) *primepow-decompose*[*OF assms*, *symmetric*], *subst power-Suc* [*symmetric*],
subst primepow-prime-power)
(*insert assms*, *auto intro*!: *prime-aprimedivisor'* *dest*: *primepow-gt-Suc-0*)

lemma *primepow-factors-altdef*:
fixes *x* :: *'a* :: *factorial-semiring-multiplicative*
assumes *x* ≠ *0*
shows *primepow-factors x* = {*p* ^ *k* |*p k*. *p* ∈ *prime-factors x* ∧ *k* ∈ {*0*<..

multiplicity p x}}
proof (*intro equalityI subsetI*)
fix *q* **assume** *q* ∈ *primepow-factors x*
then obtain *p k* **where** *pk*: *prime p k* > *0 q* = *p* ^ *k q dvd x*
unfolding *primepow-factors-def primepow-def* **by** *blast*
moreover have *k* ≤ *multiplicity p x* **using** *pk assms* **by** (*intro multiplicity-geI*)
auto
ultimately show *q* ∈ {*p* ^ *k* |*p k*. *p* ∈ *prime-factors x* ∧ *k* ∈ {*0*<.. *multiplicity*
p x}}
by (*auto simp*: *prime-factors-multiplicity intro*!: *exI*[*of* - *p*] *exI*[*of* - *k*])
qed (*auto simp*: *primepow-factors-def prime-factors-multiplicity multiplicity-dvd'*)

lemma *finite-primepow-factors*:
assumes *x* ≠ (*0* :: *'a* :: *factorial-semiring-multiplicative*)
shows *finite* (*primepow-factors x*)
proof −
have *finite* (*SIGMA p*:*prime-factors x*. {*0*<..*multiplicity p x*})
by (*intro finite-SigmaI*) *simp-all*
hence *finite* ((λ(*p*,*k*). *p* ^ *k*) ` ...) (**is** *finite ?A*) **by** (*rule finite-imageI*)
also have *?A* = *primepow-factors x*
using *assms* **by** (*subst primepow-factors-altdef*) *fast*+
finally show *?thesis* **.**
qed

lemma *aprimedivisor-primepow-factors-conv-prime-factorization*:
assumes [*simp*]: *n* ≠ (*0* :: *'a* :: *factorial-semiring-multiplicative*)
shows *image-mset aprimedivisor* (*mset-set* (*primepow-factors n*)) = *prime-factorization*
n
(**is** *?lhs* = *?rhs*)
proof (*intro multiset-eqI*)
fix *p* :: *'a*
show *count ?lhs p* = *count ?rhs p*
proof (*cases prime p*)
case *False*
have *p* ∉# *image-mset aprimedivisor* (*mset-set* (*primepow-factors n*))
proof
assume *p* ∈# *image-mset aprimedivisor* (*mset-set* (*primepow-factors n*))
then obtain *q* **where** *p* = *aprimedivisor q q* ∈ *primepow-factors n*
by (*auto simp*: *finite-primepow-factors*)
with *False prime-aprimedivisor'*[*of q*] **have** *q* = *0* ∨ *is-unit q* **by** *auto*
with ‹*q* ∈ *primepow-factors n*› **show** *False* **by** (*auto simp*: *primepow-factors-def*
primepow-def)
qed
hence *count ?lhs p* = *0* **by** (*simp only*: *Multiset.not-in-iff*)
with *False* **show** *?thesis* **by** (*simp add*: *count-prime-factorization*)
next
case *True*
hence *p*: *p* ≠ *0* ¬*is-unit p* **by** *auto*
have *count ?lhs p* = *card* (*aprimedivisor* −` {*p*} ∩ *primepow-factors n*)

by (*simp add*: *count-image-mset finite-primepow-factors*)
also have *aprimedivisor* −‘ {*p*} ∩ *primepow-factors n* = {*p*^*k* |*k*. *k* > *0* ∧ *p* ^ *k dvd n*}
using *True* **by** (*rule aprimedivisor-vimage*)
also from *True* **have** ... = (λ*k*. *p* ^ *k*) ‘ {*0*<..*multiplicity p n*}
by (*subst power-dvd-iff-le-multiplicity*) *auto*
also from *p True* **have** *card* ... = *multiplicity p n*
by (*subst card-image*) (*auto intro*!: *inj-onI dest*: *prime-power-inj*)
also from *True* **have** ... = *count* (*prime-factorization n*) *p*
by (*simp add*: *count-prime-factorization*)
finally show *?thesis* **.**
qed
qed

lemma *prime-elem-aprimedivisor-nat*: *d* > *Suc 0* ⟹ *prime-elem* (*aprimedivisor d*)
using *prime-aprimedivisor*′[*of d*] **by** *simp*

lemma *aprimedivisor-gt-0-nat* [*simp*]: *d* > *Suc 0* ⟹ *aprimedivisor d* > *0*
using *prime-aprimedivisor*′[*of d*] **by** (*simp add*: *prime-gt-0-nat*)

lemma *aprimedivisor-gt-Suc-0-nat* [*simp*]: *d* > *Suc 0* ⟹ *aprimedivisor d* > *Suc 0*
using *prime-aprimedivisor*′[*of d*] **by** (*simp add*: *prime-gt-Suc-0-nat*)

lemma *aprimedivisor-not-Suc-0-nat* [*simp*]: *d* > *Suc 0* ⟹ *aprimedivisor d* ≠ *Suc 0*
using *aprimedivisor-gt-Suc-0*[*of d*] **by** (*intro notI*) *auto*

lemma *multiplicity-aprimedivisor-gt-0-nat* [*simp*]:
d > *Suc 0* ⟹ *multiplicity* (*aprimedivisor d*) *d* > *0*
by (*subst multiplicity-gt-zero-iff*) (*auto intro*: *aprimedivisor-dvd*′)

lemma *primepowI*:
prime p ⟹ *k* > *0* ⟹ *p* ^ *k* = *n* ⟹ *primepow n* ∧ *aprimedivisor n* = *p*
unfolding *primepow-def* **by** (*auto simp*: *aprimedivisor-prime-power*)

lemma *not-primepowI*:
assumes *prime p prime q p* ≠ *q p dvd n q dvd n*
shows ¬*primepow n*
using *assms* **by** (*auto simp*: *primepow-def dest*!: *prime-dvd-power*[*rotated*] *dest*: *primes-dvd-imp-eq*)

lemma *sum-prime-factorization-conv-sum-primepow-factors*:
fixes *n* :: ′*a* :: *factorial-semiring-multiplicative*
assumes *n* ≠ *0*
shows (∑ *q*∈*primepow-factors n*. *f* (*aprimedivisor q*)) = (∑ *p*∈#*prime-factorization n*. *f p*)
proof −

from *assms* **have** *prime-factorization n = image-mset aprimedivisor (mset-set (primepow-factors n))*
by (*rule aprimedivisor-primepow-factors-conv-prime-factorization* [*symmetric*])
also have $(\sum p{\in}\#. \ldots\ f\ p) = (\sum q{\in}$*primepow-factors n. f (aprimedivisor q*$))$
by (*simp add*: *image-mset.compositionality sum-unfold-sum-mset o-def*)
finally show *?thesis* **..**
qed

lemma *multiplicity-aprimedivisor-Suc-0-iff*:
assumes *primepow* (*n* :: *'a* :: *factorial-semiring-multiplicative*)
shows *multiplicity* (*aprimedivisor n*) *n* = *Suc 0* $\longleftrightarrow$ *prime n*
by (*subst* (*3*) *primepow-decompose* [*OF assms*, *symmetric*])
(*insert assms*, *auto simp add*: *prime-power-iff intro*!: *prime-aprimedivisor'*)

definition *mangoldt* :: *nat* $\Rightarrow$ *'a* :: *real-algebra-1* **where**
mangoldt n = (*if primepow n then of-real* (*ln* (*real* (*aprimedivisor n*))) *else 0*)

lemma *mangoldt-0* [*simp*]: *mangoldt 0* = *0*
by (*simp add*: *mangoldt-def*)

lemma *mangoldt-Suc-0* [*simp*]: *mangoldt* (*Suc 0*) = *0*
by (*simp add*: *mangoldt-def*)

lemma *of-real-mangoldt* [*simp*]: *of-real* (*mangoldt n*) = *mangoldt n*
by (*simp add*: *mangoldt-def*)

lemma *mangoldt-sum*:
assumes *n* $\neq$ *0*
shows $(\sum d \mid d\ dvd\ n.$ *mangoldt d* :: *'a* :: *real-algebra-1*$)$ = *of-real* (*ln* (*real n*))
proof −
have $(\sum d \mid d\ dvd\ n.$ *mangoldt d* :: *'a*$)$ = *of-real* $(\sum d \mid d\ dvd\ n.$ *mangoldt d*$)$ **by** *simp*
also have $(\sum d \mid d\ dvd\ n.$ *mangoldt d*$)$ = $(\sum d \in$ *primepow-factors n. ln* (*real* (*aprimedivisor d*)))
using *assms* **by** (*intro sum.mono-neutral-cong-right*) (*auto simp*: *primepow-factors-def mangoldt-def*)
also have ... = *ln* (*real* $(\prod d \in$ *primepow-factors n. aprimedivisor d*$)$)
using *assms finite-primepow-factors*[*of n*]
by (*subst ln-prod* [*symmetric*])
(*auto simp*: *primepow-factors-def intro*!: *aprimedivisor-pos-nat*
intro: *Nat.gr0I primepow-gt-Suc-0*)
also have *primepow-factors n* =
$(\lambda(p,k).\ p\ \hat{}\ k)$ ' (*SIGMA p*:*prime-factors n.* $\{0<..$*multiplicity p n*$\}$)
(**is** - = - ' *?A*) **by** (*subst primepow-factors-altdef*[*OF assms*]) *fast*+
also have *prod aprimedivisor* ... = $(\prod (p,k){\in}$*?A. aprimedivisor* $(p\ \hat{}\ k)$)
by (*subst prod.reindex*)
(*auto simp*: *inj-on-def prime-power-inj'' prime-factors-multiplicity*
prod.Sigma [*symmetric*] *case-prod-unfold*)

```
  also have ... = (∏(p,k)∈?A. p)
   by (intro prod.cong refl) (auto simp: aprimedivisor-prime-power prime-factors-multiplicity)
  also have ... = (∏x∈prime-factors n. ∏k∈{0<..multiplicity x n}. x)
    by (rule prod.Sigma [symmetric]) auto
  also have ... = (∏x∈prime-factors n. x ^ multiplicity x n)
    by (intro prod.cong refl) (simp add: prod-constant)
  also have ... = n using assms by (intro prime-factorization-nat [symmetric])
simp
  finally show ?thesis .
qed

lemma mangoldt-primepow:
   prime p ⟹ mangoldt (p ^ k) = (if k > 0 then of-real (ln (real p)) else 0)
  by (simp add: mangoldt-def aprimedivisor-prime-power)

lemma mangoldt-primepow' [simp]: prime p ⟹ k > 0 ⟹ mangoldt (p ^ k) =
of-real (ln (real p))
  by (subst mangoldt-primepow) auto

lemma mangoldt-prime [simp]: prime p ⟹ mangoldt p = of-real (ln (real p))
  using mangoldt-primepow[of p 1] by simp

lemma mangoldt-nonneg: 0 ≤ (mangoldt d :: real)
  using aprimedivisor-gt-Suc-0-nat[of d]
  by (auto simp: mangoldt-def of-nat-le-iff[of 1 x for x, unfolded of-nat-1] Suc-le-eq
          intro!: ln-ge-zero dest: primepow-gt-Suc-0)

lemma norm-mangoldt [simp]:
  norm (mangoldt n :: 'a :: real-normed-algebra-1) = mangoldt n
proof (cases primepow n)
  case True
  hence prime (aprimedivisor n)
    by (intro prime-aprimedivisor')
       (auto simp: primepow-def prime-gt-0-nat)
  hence aprimedivisor n > 1 by (simp add: prime-gt-Suc-0-nat)
  with True show ?thesis by (auto simp: mangoldt-def abs-if)
qed (auto simp: mangoldt-def)

lemma Re-mangoldt [simp]: Re (mangoldt n) = mangoldt n
  and Im-mangoldt [simp]: Im (mangoldt n) = 0
  by (simp-all add: mangoldt-def)

lemma abs-mangoldt [simp]: abs (mangoldt n :: real) = mangoldt n
  using norm-mangoldt[of n, where ?'a = real, unfolded real-norm-def] .

lemma mangoldt-le:
  assumes n > 0
  shows   mangoldt n ≤ ln n
proof (cases primepow n)
```

case *True*
from *True* **have** *prime* (*aprimedivisor n*)
by (*intro prime-aprimedivisor'*)
(*auto simp*: *primepow-def prime-gt-0-nat*)
hence *gt-1*: *aprimedivisor n > 1* **by** (*simp add*: *prime-gt-Suc-0-nat*)
from *True* **have** *mangoldt n = ln* (*aprimedivisor n*)
by (*simp add*: *mangoldt-def*)
also have ... ≤ *ln n* **using** *True gt-1*
by (*subst ln-le-cancel-iff*) (*auto intro*!: *Nat.gr0I dvd-imp-le aprimedivisor-dvd'*)
finally show *?thesis* **.**
qed (*insert assms*, *auto simp*: *mangoldt-def*)

end

11 Primitive roots in residue rings and Carmichael's function

theory *Residue-Primitive-Roots*
imports *Pocklington*
begin

This theory develops the notions of primitive roots (generators) in residue rings. It also provides a definition and all the basic properties of Carmichael's function $\lambda(n)$, which is strongly related to this. The proofs mostly follow Apostol's presentation

11.1 Primitive roots in residue rings

A primitive root of a residue ring modulo n is an element g that *generates* the ring, i. e. such that for each x coprime to n there exists an i such that $x = g^i$. A simpler definition is that g must have the same order as the cardinality of the multiplicative group, which is $\varphi(n)$.

Note that for convenience, this definition does *not* demand $g < n$.

inductive *residue-primroot* :: *nat* ⇒ *nat* ⇒ *bool* **where**
n > 0 ⟹ *coprime n g* ⟹ *ord n g = totient n* ⟹ *residue-primroot n g*

lemma *residue-primroot-def* [*code*]:
residue-primroot n x ⟷ *n > 0* ∧ *coprime n x* ∧ *ord n x = totient n*
by (*simp add*: *residue-primroot.simps*)

lemma *not-residue-primroot-0* [*simp*]: ~*residue-primroot 0 x*
by (*auto simp*: *residue-primroot-def*)

lemma *residue-primroot-mod* [*simp*]: *residue-primroot n* (*x mod n*) = *residue-primroot n x*
by (*cases n = 0*) (*simp-all add*: *residue-primroot-def*)

lemma *residue-primroot-cong*:
assumes $[x = x']\ (mod\ n)$
shows *residue-primroot n x = residue-primroot n x'*
proof −
have *residue-primroot n x = residue-primroot n (x mod n)*
by *simp*
also have *x mod n = x' mod n*
using *assms* **by** (*simp add*: *cong-def*)
also have *residue-primroot n (x' mod n) = residue-primroot n x'*
by *simp*
finally show *?thesis* **.**
qed

lemma *not-residue-primroot-0-right* [*simp*]: *residue-primroot n 0* ⟷ *n = 1*
by (*auto simp*: *residue-primroot-def*)

lemma *residue-primroot-1-iff*: *residue-primroot n* (*Suc 0*) ⟷ *n* ∈ {*1*, *2*}
proof
assume ∗: *residue-primroot n* (*Suc 0*)
with *totient-gt-1*[*of n*] **have** *n* ≤ *2* **by** (*cases n* ≤ *2*) (*auto simp*: *residue-primroot-def*)
hence *n* ∈ {*0*, *1*, *2*} **by** *auto*
thus *n* ∈ {*1*, *2*} **using** ∗ **by** (*auto simp*: *residue-primroot-def*)
qed (*auto simp*: *residue-primroot-def*)

11.2 Primitive roots modulo a prime

For prime p, we now analyse the number of elements in the ring $\mathbb{Z}/p\mathbb{Z}$ whose order is precisely d for each d.

context
fixes *n* :: *nat* **and** *ψ*
assumes *n*: *n > 1*
defines *ψ* ≡ (λ*d*. *card* {*x*∈*totatives n*. *ord n x = d*})
begin

lemma *elements-with-ord-restrict-totatives*:
d > 0 ⟹ {*x*∈{..<*n*}. *ord n x = d*} = {*x*∈*totatives n*. *ord n x = d*}
using *n* **by** (*auto simp*: *totatives-def coprime-commute intro*!: *Nat.gr0I le-neq-trans*)

lemma *prime-elements-with-ord*:
assumes *ψ d* ≠ *0* **and** *prime n*
and *a*: *a* ∈ *totatives n ord n a = d a* ≠ *1*
shows *inj-on* (λ*k*. *a* ^ *k mod n*) {..<*d*}
and {*x*∈{..<*n*}. [*x* ^ *d = 1*] (*mod n*)} = (λ*k*. *a* ^ *k mod n*) ‘ {..<*d*}
and *bij-betw* (λ*k*. *a* ^ *k mod n*) (*totatives d*) {*x*∈{..<*n*}. *ord n x = d*}
proof −
show *inj*: *inj-on* (λ*k*. *a* ^ *k mod n*) {..<*d*}
using *inj-power-mod*[*of n a*] *a* **by** (*auto simp*: *totatives-def coprime-commute*)
from *a* **have** *d > 0* **by** (*auto simp*: *totatives-def coprime-commute*)

```
  moreover have d ≠ 1 using a n
    by (auto simp: ord-eq-Suc-0-iff totatives-less cong-def)
  ultimately have d: d > 1 by simp

  have ∗: (λk. a ^ k mod n) ‘ {..<d} = {x∈{..<n}. [x ^ d = 1] (mod n)}
  proof (rule card-seteq)
    have card {x∈{..<n}. [x ^ d = 1] (mod n)} ≤ d
      using assms a by (intro roots-mod-prime-bound) (auto simp: totatives-def
coprime-commute)
    also have ... = card ((λk. a ^ k mod n) ‘ {..<d})
      using inj by (subst card-image) auto
    finally show card {x ∈ {..<n}. [x ^ d = 1] (mod n)} ≤ ... .
  next
    show (λk. a ^ k mod n) ‘ {..<d} ⊆ {x ∈ {..<n}. [x ^ d = 1] (mod n)}
    proof safe
      fix k assume k < d
      have [(a ^ d) ^ k = 1 ^ k] (mod n)
        by (intro cong-pow) (use a in ‹auto simp: ord-divides'›)
      thus [(a ^ k mod n) ^ d = 1] (mod n)
        by (simp add: power-mult [symmetric] cong-def power-mod mult.commute)
    qed (use ‹prime n› in ‹auto dest: prime-gt-1-nat›)
  qed auto
  thus {x∈{..<n}. [x ^ d = 1] (mod n)} = (λk. a ^ k mod n) ‘ {..<d} ..

  show bij-betw (λk. a ^ k mod n) (totatives d) {x∈{..<n}. ord n x = d}
    unfolding bij-betw-def
  proof (intro conjI inj-on-subset[OF inj] equalityI subsetI)
    fix b assume b ∈ (λk. a ^ k mod n) ‘ totatives d
    then obtain k where b = a ^ k mod n k ∈ totatives d by auto
    thus b ∈ {b ∈ {..<n}. ord n b = d}
      using n a by (simp add: ord-power totatives-def coprime-commute)
  next
    fix b assume b ∈ {x ∈ {..<n}. ord n x = d}
    hence b: ord n b = d b < n by auto
    with d have coprime n b using ord-eq-0[of n b] by auto
    from b have b ∈ {x∈{..<n}. [x ^ d = 1] (mod n)}
      by (auto simp: ord-divides')
    with ∗ obtain k where k: k < d b = a ^ k mod n
      by blast
    with b(2) n a d have d div gcd k d = ord n b
      using ‹coprime n b› by (auto simp: ord-power)
    also have ord n b = d by (simp add: b)
    finally have coprime k d
     unfolding coprime-iff-gcd-eq-1 using d a by (subst (asm) div-eq-dividend-iff)
auto
    with k b d have k ∈ totatives d by (auto simp: totatives-def intro!: Nat.gr0I)
    with k show b ∈ (λk. a ^ k mod n) ‘ totatives d by blast
  qed (use d n in ‹auto simp: totatives-less›)
qed
```

lemma *prime-card-elements-with-ord*:
assumes $\psi\ d \neq 0$ **and** *prime n*
shows $\psi\ d = totient\ d$
proof (*cases* $d = 1$)
case *True*
have $\psi\ 1 = 1$
using *elements-with-ord-1*[*of n*] *n* **by** (*simp add*: ψ-*def*)
thus *?thesis* **using** *True* **by** *simp*
next
case *False*
from *assms* **obtain** *a* **where** *a*: $a \in totatives\ n\ ord\ n\ a = d$
by (*auto simp*: ψ-*def*)
from *a* **have** $d > 0$ **by** (*auto intro*!: *Nat.gr0I simp*: *ord-eq-0 totatives-def coprime-commute*)
from *a* **and** *False* **have** $a \neq 1$ **by** *auto*
from *bij-betw-same-card*[*OF prime-elements-with-ord*(*3*)[*OF assms a this*]] **show** $\psi\ d = totient\ d$
using *elements-with-ord-restrict-totatives*[*of d*] *False a* ‹$d > 0$›
by (*simp add*: ψ-*def totient-def*)
qed

lemma *prime-sum-card-elements-with-ord-eq-totient*:
$(\sum d \mid d\ dvd\ totient\ n.\ \psi\ d) = totient\ n$
proof −
have *totient n* = *card* (*totatives n*)
by (*simp add*: *totient-def*)
also have *totatives n* = $(\bigcup d \in \{d.\ d\ dvd\ totient\ n\}.\ \{x \in totatives\ n.\ ord\ n\ x = d\})$
by (*force simp*: *order-divides-totient totatives-def coprime-commute*)
also have *card* ... = $(\sum d \mid d\ dvd\ totient\ n.\ \psi\ d)$
unfolding ψ-*def* **using** *n* **by** (*subst card-UN-disjoint*) (*auto intro*!: *finite-divisors-nat*)
finally show *?thesis* **..**
qed

We can now show that the number of elements of order d is $\varphi(d)$ if $d \mid p-1$ and 0 otherwise.

theorem *prime-card-elements-with-ord-eq-totient*:
assumes *prime n*
shows $\psi\ d = (if\ d\ dvd\ n - 1\ then\ totient\ d\ else\ 0)$
proof (*cases d dvd totient n*)
case *False*
thus *?thesis* **using** *order-divides-totient*[*of n*] *assms*
by (*auto simp*: ψ-*def totient-prime totatives-def coprime-commute*[*of n*])
next
case *True*
have $\psi\ d = totient\ d$
proof (*rule ccontr*)
assume *neq*: $\psi\ d \neq totient\ d$

```
    have le: ψ d ≤ totient d if d dvd totient n for d
      using prime-card-elements-with-ord[of d] assms by (cases ψ d = 0) auto
    from neq and le[of d] and True have less: ψ d < totient d by auto

    have totient n = (∑ d | d dvd totient n. ψ d)
      using prime-sum-card-elements-with-ord-eq-totient ..
    also have ... < (∑ d | d dvd totient n. totient d)
      by (rule sum-strict-mono-ex1)
         (use n le less assms True in ‹auto intro!: finite-divisors-nat›)
    also have ... = totient n
      using totient-divisor-sum .
    finally show False by simp
  qed
  with True show ?thesis using assms by (simp add: totient-prime)
qed
```

As a corollary, we get that the number of primitive roots modulo a prime p is $\varphi(p-1)$. Since this number is positive, we also get that there *is* at least one primitive root modulo p.

```
lemma
  assumes prime n
  shows   prime-card-primitive-roots:  card {x∈totatives n. ord n x = n - 1} =
totient (n - 1)
                                  card {x∈{..<n}. ord n x = n - 1} = totient (n - 1)
  and     prime-primitive-root-exists: ∃x. residue-primroot n x
proof -
  show *: card {x∈totatives n. ord n x = n - 1} = totient (n - 1)
    using prime-card-elements-with-ord-eq-totient[of n - 1] assms
    by (auto simp: totient-prime ψ-def)
  thus card {x∈{..<n}. ord n x = n - 1} = totient (n - 1)
    using assms n elements-with-ord-restrict-totatives[of n - 1] by simp

  note *
  also have totient (n - 1) > 0 using n by auto
  finally show ∃x. residue-primroot n x using assms prime-gt-1-nat[of n]
    by (subst (asm) card-gt-0-iff)
       (auto simp: residue-primroot-def totient-prime totatives-def coprime-commute)
qed

end
```

11.3 Primitive roots modulo powers of an odd prime

Any primitive root g modulo an odd prime p is also a primitive root modulo p^k for all $k > 0$ if $[g^{p-1} \neq 1] \pmod{p^2}$. To show this, we first need the following lemma.

```
lemma residue-primroot-power-prime-power-neq-1:
  assumes k ≥ 2
```

```
  assumes p: prime p odd p and residue-primroot p g and [g ^ (p - 1) ≠ 1] (mod
p^2)
  shows   [g ^ totient (p ^ (k - 1)) ≠ 1] (mod (p ^ k))
  using assms(1)
proof (induction k rule: dec-induct)
  case base
  thus ?case using assms by (simp add: totient-prime)
next
  case (step k)
  from p have p > 2
    using prime-gt-1-nat[of p] by (cases p = 2) auto
  from assms have g: g > 0 by (auto intro!: Nat.gr0I)
  have [g ^ totient (p ^ (k - 1)) = 1] (mod p ^ (k - 1))
    using assms by (intro euler-theorem)
                   (auto simp: residue-primroot-def totatives-def coprime-commute)
  from cong-to-1-nat[OF this]
    obtain t where *: g ^ totient (p ^ (k - 1)) - 1 = p ^ (k - 1) * t by auto
  have t: g ^ totient (p ^ (k - 1)) = p ^ (k - 1) * t + 1
    using g by (subst * [symmetric]) auto

  have ¬p dvd t
  proof
    assume p dvd t
    then obtain q where [simp]: t = p * q by auto
    from t have g ^ totient (p ^ (k - 1)) = p ^ k * q + 1
      using ‹k ≥ 2› by (cases k) auto
    hence [g ^ totient (p ^ (k - 1)) = p ^ k * q + 1] (mod p ^ k)
      by simp
    also have [p ^ k * q + 1 = 0 * q + 1] (mod p ^ k)
      by (intro cong-add cong-mult) (auto simp: cong-0-iff)
    finally have [g ^ totient (p ^ (k - 1)) = 1] (mod p ^ k)
      by simp
    with step.IH show False by contradiction
  qed

  from t have (g ^ totient (p ^ (k - 1))) ^ p = (p ^ (k - 1) * t + 1) ^ p
    by (rule arg-cong)
  also have (g ^ totient (p ^ (k - 1))) ^ p = g ^ (p * totient (p ^ (k - 1)))
    by (simp add: power-mult [symmetric] mult.commute)
  also have p * totient (p ^ (k - 1)) = totient (p ^ k)
   using p ‹k ≥ 2› by (simp add: totient-prime-power Suc-diff-Suc flip: power-Suc)
  also have (p ^ (k - 1) * t + 1) ^ p = (∑ i≤p. (p choose i) * t ^ i * p ^ (i *
(k - 1)))
   by (subst binomial) (simp-all add: mult-ac power-mult-distrib power-mult [symmetric])
  finally have [g ^ totient (p ^ k) = (∑ i≤p. (p choose i) * t ^ i * p ^ (i * (k -
1)))]
                (mod (p ^ Suc k)) (is [- = ?rhs] (mod -)) by simp

  also have [?rhs = (∑ i≤p. if i ≤ 1 then (p choose i) * t ^ i * p ^ (i * (k - 1))
```

```
else 0)]
            (mod (p ^ Suc k)) (is [sum ?f - = sum ?g -] (mod -))
  proof (intro cong-sum)
    fix i assume i: i ∈ {..p}
    consider i ≤ 1 | i = 2 | i > 2 by force
    thus [?f i = ?g i] (mod (p ^ Suc k))
    proof cases
      assume i: i > 2
      have Suc k ≤ 3 * (k - 1)
        using ‹k ≥ 2› by (simp add: algebra-simps)
      also have 3 * (k - 1) ≤ i * (k - 1)
        using i by (intro mult-right-mono) auto
      finally have p ^ Suc k dvd ?f i
        by (intro dvd-mult le-imp-power-dvd)
      thus [?f i = ?g i] (mod (p ^ Suc k))
        by (simp add: cong-0-iff)
    next
      assume [simp]: i = 2
      have ?f i = p * (p - 1) div 2 * t² * p ^ (2 * (k - 1))
        using choose-two[of p] by simp
      also have p * (p - 1) div 2 = (p - 1) div 2 * p
        using ‹odd p› by (auto elim!: oddE)
      also have ... * t² * p ^ (2 * (k - 1)) = (p - 1) div 2 * t² * (p * p ^ (2 *
(k - 1)))
        by (simp add: algebra-simps)
      also have p * p ^ (2 * (k - 1)) = p ^ (2 * k - 1)
        using ‹k ≥ 2› by (cases k) auto
      also have p ^ Suc k dvd (p - 1) div 2 * t² * p ^ (2 * k - 1)
        using ‹k ≥ 2› by (intro dvd-mult le-imp-power-dvd) auto
      finally show [?f i = ?g i] (mod (p ^ Suc k))
        by (simp add: cong-0-iff)
    qed auto
  qed
  also have (∑ i≤p. ?g i) = (∑ i≤1. ?f i)
    using p prime-gt-1-nat[of p] by (intro sum.mono-neutral-cong-right) auto
  also have ... = 1 + t * p ^ k
    using choose-two[of p] ‹k ≥ 2› by (cases k) simp-all
  finally have eq: [g ^ totient (p ^ k) = 1 + t * p ^ k] (mod p ^ Suc k) .

  have [g ^ totient (p ^ k) ≠ 1] (mod p ^ Suc k)
  proof
    assume [g ^ totient (p ^ k) = 1] (mod p ^ Suc k)
    hence [g ^ totient (p ^ k) - g ^ totient (p ^ k) = 1 + t * p ^ k - 1] (mod p
^ Suc k)
      by (intro cong-diff-nat eq) auto
    hence [t * p ^ k = 0] (mod p ^ Suc k)
      by (simp add: cong-sym-eq)
    hence p * p ^ k dvd t * p ^ k
      by (simp add: cong-0-iff)
```

```
    hence p dvd t using ‹p > 2› by simp
    with ‹¬p dvd t› show False by contradiction
  qed
  thus ?case by simp
qed
```

We can now show that primitive roots modulo p with the above condition are indeed also primitive roots modulo p^k.

```
proposition residue-primroot-prime-lift-iff:
  assumes p: prime p odd p and residue-primroot p g
  shows   (∀k>0. residue-primroot (p ^ k) g) ⟷ [g ^ (p - 1) ≠ 1] (mod p²)
proof -
  from assms have g: coprime p g ord p g = p - 1
    by (auto simp: residue-primroot-def totient-prime)
  show ?thesis
  proof
    assume ∀k>0. residue-primroot (p ^ k) g
    hence residue-primroot (p²) g by auto
    hence ord (p²) g = totient (p²)
      by (simp-all add: residue-primroot-def)
    thus [g ^ (p - 1) ≠ 1] (mod p²)
      using g assms prime-gt-1-nat[of p]
      by (auto simp: ord-divides' totient-prime-power)
  next
    assume g': [g ^ (p - 1) ≠ 1] (mod p²)

    have residue-primroot (p ^ k) g if k > 0 for k
    proof (cases k = 1)
      case False
      with that have k: k > 1 by simp
      from g have coprime: coprime (p ^ k) g
        by (auto simp: totatives-def coprime-commute)

      define t where t = ord (p ^ k) g
      have [g ^ t = 1] (mod (p ^ k))
        by (simp add: t-def ord-divides')
      also have p ^ k = p * p ^ (k - 1)
        using k by (cases k) auto
      finally have [g ^ t = 1] (mod p)
        by (rule cong-modulus-mult-nat)
      hence totient p dvd t
        using g p by (simp add: ord-divides' totient-prime)
      then obtain q where t: t = totient p * q by auto

      have t dvd totient (p ^ k)
        using coprime by (simp add: t-def order-divides-totient)
      with t p k have q dvd p ^ (k - 1) using prime-gt-1-nat[of p]
        by (auto simp: totient-prime totient-prime-power)
      then obtain b where b: b ≤ k - 1 q = p ^ b
```

```
      using divides-primepow-nat[of p q k − 1] p by auto

    have b = k − 1
    proof (rule ccontr)
      assume b ≠ k − 1
      with b have b < k − 1 by simp
      have t = p ^ b ∗ (p − 1)
        using b p by (simp add: t totient-prime)
      also have ... dvd p ^ (k − 2) ∗ (p − 1)
        using ‹b < k − 1› by (intro mult-dvd-mono le-imp-power-dvd) auto
      also have ... = totient (p ^ (k − 1))
        using k p by (simp add: totient-prime-power numeral-2-eq-2)
      finally have [g ^ totient (p ^ (k − 1)) = 1] (mod (p ^ k))
        by (simp add: ord-divides′ t-def)
      with residue-primroot-power-prime-power-neq-1[of k p g] p k assms g′ show
False
        by auto
    qed
    hence t = totient (p ^ k)
      using p k by (simp add: t b totient-prime totient-prime-power)
    thus residue-primroot (p ^ k) g
      using g one-less-power[of p k] prime-gt-1-nat[of p] p k
      by (simp add: residue-primroot-def t-def)
  qed (use assms in auto)
  thus ∀ k>0. residue-primroot (p ^ k) g by blast
 qed
qed
```

If p is an odd prime, there is always a primitive root g modulo p, and if g does not fulfil the above assumption required for it to be liftable to p^k, we can use $g + p$, which is also a primitive root modulo p and *does* fulfil the assumption.

This shows that any modulus that is a power of an odd prime has a primitive root.

```
theorem residue-primroot-odd-prime-power-exists:
 assumes p: prime p odd p
 obtains g where ∀ k>0. residue-primroot (p ^ k) g
proof −
 obtain g where g: residue-primroot p g
   using prime-primitive-root-exists[of p] assms prime-gt-1-nat[of p] by auto

 have ∃ g. residue-primroot p g ∧ [g ^ (p − 1) ≠ 1] (mod p²)
 proof (cases [g ^ (p − 1) = 1] (mod p²))
   case True
   define g′ where g′ = p + g
   note g
   also have residue-primroot p g ⟷ residue-primroot p g′
     unfolding g′-def by (rule residue-primroot-cong) (auto simp: cong-def)
```

finally have g': *residue-primroot p g'* **.**

have [$g' \hat{} (p - 1) = (\sum k{\le}p{-}1.\ ((p{-}1)\ choose\ k) * g \hat{} (p - Suc\ k) * p \hat{} k)$] (*mod* p^2)
(**is** [*- = ?rhs*] (*mod -*)) **by** (*simp add*: *g'-def binomial mult-ac*)
also have [*?rhs* = $(\sum k{\le}p{-}1.$ *if* $k \le 1$ *then* $((p{-}1)\ choose\ k) * g \hat{} (p - Suc\ k) * p \hat{} k$ *else 0*)] (*mod* p^2)
(**is** [*sum ?f - = sum ?g -*] (*mod -*))
proof (*intro cong-sum*)
fix *k* **assume** $k \in \{..p{-}1\}$
show [*?f k = ?g k*] (*mod* p^2)
proof (*cases* $k \le 1$)
case *False*
have p^2 *dvd ?f k*
using *False* **by** (*intro dvd-mult le-imp-power-dvd*) *auto*
thus *?thesis* **using** *False* **by** (*simp add*: *cong-0-iff*)
qed *auto*
qed
also have *sum ?g* $\{..p{-}1\}$ = *sum ?f* $\{0, 1\}$
using *prime-gt-1-nat*[*of p*] *p* **by** (*intro sum.mono-neutral-cong-right*) *auto*
also have ... = $g \hat{} (p - 1) + p * (p - 1) * g \hat{} (p - 2)$
using *p* **by** (*simp add*: *numeral-2-eq-2*)
also have [$g \hat{} (p - 1) + p * (p - 1) * g \hat{} (p - 2) = 1 + p * (p - 1) * g \hat{} (p - 2)$] (*mod* p^2)
by (*intro cong-add True*) *auto*
finally have [$g' \hat{} (p - 1) = 1 + p * (p - 1) * g \hat{} (p - 2)$] (*mod* p^2) **.**

moreover have [$1 + p * (p - 1) * g \hat{} (p - 2) \neq 1$] (*mod* p^2)
proof
assume [$1 + p * (p - 1) * g \hat{} (p - 2) = 1$] (*mod* p^2)
hence [$1 + p * (p - 1) * g \hat{} (p - 2) - 1 = 1 - 1$] (*mod* p^2)
by (*intro cong-diff-nat*) *auto*
hence $p * p$ *dvd* $p * ((p - 1) * g \hat{} (p - 2))$
by (*auto simp*: *cong-0-iff power2-eq-square*)
hence *p dvd* $(p - 1) * g \hat{} (p - 2)$
using *p* **by** *simp*
hence *p dvd* $g \hat{} (p - 2)$
using *p dvd-imp-le*[*of p p - Suc 0*] *prime-gt-1-nat*[*of p*]
by (*auto simp*: *prime-dvd-mult-iff*)
also have ... *dvd* $g \hat{} (p - 1)$
by (*intro le-imp-power-dvd*) *auto*
finally have [$g \hat{} (p - 1) = 0$] (*mod p*)
by (*simp add*: *cong-0-iff*)
hence [$0 = g \hat{} (p - 1)$] (*mod p*)
by (*simp add*: *cong-sym-eq*)

also from ‹*residue-primroot p g*› **have** [$g \hat{} (p - 1) = 1$] (*mod p*)
using *p* **by** (*auto simp*: *residue-primroot-def ord-divides' totient-prime*)
finally have [$0 = 1$] (*mod p*) **.**

```
    thus False using prime-gt-1-nat[of p] p by (simp add: cong-def)
  qed

  ultimately have [g' ^ (p - 1) ≠ 1] (mod p^2)
    by (simp add: cong-def)
  thus ∃ g. residue-primroot p g ∧ [g ^ (p - 1) ≠ 1] (mod p^2)
    using g' by blast
qed (use g in auto)
thus ?thesis
  using residue-primroot-prime-lift-iff[OF assms] that by blast
qed
```

11.4 Carmichael's function

Carmichael's function $\lambda(n)$ gives the LCM of the orders of all elements in the residue ring modulo n – or, equivalently, the maximum order, as we will show later. Algebraically speaking, it is the exponent of the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^*$.

It is not to be confused with Liouville's function, which is also denoted by $\lambda(n)$.

```
definition Carmichael where
  Carmichael n = (LCM a ∈ totatives n. ord n a)

lemma Carmichael-0 [simp]: Carmichael 0 = 1
  by (simp add: Carmichael-def)

lemma Carmichael-1 [simp]: Carmichael 1 = 1
  by (simp add: Carmichael-def)

lemma Carmichael-Suc-0 [simp]: Carmichael (Suc 0) = 1
  by (simp add: Carmichael-def)

lemma ord-dvd-Carmichael:
  assumes n > 1 coprime n k
  shows   ord n k dvd Carmichael n
proof -
  have k mod n ∈ totatives n
    using assms by (auto simp: totatives-def coprime-commute intro!: Nat.gr0I)
  hence ord n (k mod n) dvd Carmichael n
    by (simp add: Carmichael-def del: ord-mod)
  thus ?thesis by simp
qed

lemma Carmichael-divides:
  assumes Carmichael n dvd k coprime n a
  shows   [a ^ k = 1] (mod n)
proof (cases n < 2 ∨ a = 1)
  case False
```

```
  hence ord n a dvd Carmichael n
    using False assms by (intro ord-dvd-Carmichael) auto
  also have ... dvd k by fact
  finally have ord n a dvd k .
  thus ?thesis using ord-divides by auto
next
  case True
  then consider a = 1 | n = 0 | n = 1 by force
  thus ?thesis using assms by cases auto
qed

lemma Carmichael-dvd-totient: Carmichael n dvd totient n
  unfolding Carmichael-def
proof (intro Lcm-least, safe)
  fix a assume a ∈ totatives n
  hence [a ^ totient n = 1] (mod n)
    by (intro euler-theorem) (auto simp: totatives-def)
  thus ord n a dvd totient n
    using ord-divides by blast
qed

lemma Carmichael-dvd-mono-coprime:
  assumes coprime m n m > 1 n > 1
  shows   Carmichael m dvd Carmichael (m * n)
  unfolding Carmichael-def[of m]
proof (intro Lcm-least, safe)
  fix x assume x: x ∈ totatives m
  from assms have totatives n ≠ {} by simp
  then obtain y where y: y ∈ totatives n by blast

  from binary-chinese-remainder-nat[OF assms(1), of x y]
  obtain z where z: [z = x] (mod m) [z = y] (mod n) by blast
  have z': coprime z n coprime z m
   by (rule cong-imp-coprime; use x y z in ‹force simp: totatives-def cong-sym-eq›)+

  from z have ord m x = ord m z
    by (intro ord-cong) (auto simp: cong-sym-eq)
  also have ord m z dvd ord (m * n) z
    using assms by (auto simp: ord-modulus-mult-coprime)
  also from z' assms have ... dvd Carmichael (m * n)
   by (intro ord-dvd-Carmichael) (auto simp: coprime-commute intro!:one-less-mult)
  finally show ord m x dvd Carmichael (m * n) .
qed
```

λ distributes over the product of coprime numbers similarly to φ, but with LCM instead of multiplication:

```
lemma Carmichael-mult-coprime:
  assumes coprime m n
  shows   Carmichael (m * n) = lcm (Carmichael m) (Carmichael n)
```

proof (*cases* $m \leq 1 \vee n \leq 1$)
case *True*
hence $m = 0 \vee n = 0 \vee m = 1 \vee n = 1$ **by** *force*
thus *?thesis* **using** *assms* **by** *auto*
next
case *False*
show *?thesis*
proof (*rule dvd-antisym*)
show *Carmichael* $(m * n)$ *dvd lcm* (*Carmichael m*) (*Carmichael n*)
unfolding *Carmichael-def*[*of* $m * n$]
proof (*intro Lcm-least*, *safe*)
fix x **assume** x: $x \in$ *totatives* $(m * n)$
have *ord* $(m * n)$ $x =$ *lcm* (*ord m x*) (*ord n x*)
using *assms x* **by** (*subst ord-modulus-mult-coprime*) (*auto simp*: *coprime-commute totatives-def*)
also have ... *dvd lcm* (*Carmichael m*) (*Carmichael n*)
using *False x*
by (*intro lcm-mono ord-dvd-Carmichael*) (*auto simp*: *totatives-def coprime-commute*)
finally show *ord* $(m * n)$ *x dvd* ... **.**
qed
next
show *lcm* (*Carmichael m*) (*Carmichael n*) *dvd Carmichael* $(m * n)$
using *Carmichael-dvd-mono-coprime*[*of m n*]
Carmichael-dvd-mono-coprime[*of n m*] *assms False*
by (*auto intro*!: *lcm-least simp*: *coprime-commute mult.commute*)
qed
qed

lemma *Carmichael-pos* [*simp*, *intro*]: *Carmichael* $n > 0$
by (*auto simp*: *Carmichael-def ord-eq-0 totatives-def coprime-commute intro*!: *Nat.gr0I*)

lemma *Carmichael-nonzero* [*simp*]: *Carmichael* $n \neq 0$
by *simp*

lemma *power-Carmichael-eq-1*:
assumes $n > 1$ *coprime n x*
shows [$x \mathbin{\hat{}} Carmichael\ n = 1$] (*mod n*)
using *ord-dvd-Carmichael*[*of n x*] *assms*
by (*auto simp*: *ord-divides'*)

lemma *Carmichael-2* [*simp*]: *Carmichael 2* $= 1$
using *Carmichael-dvd-totient*[*of 2*] **by** *simp*

lemma *Carmichael-4* [*simp*]: *Carmichael 4* $= 2$
proof −
have *Carmichael 4 dvd 2*
using *Carmichael-dvd-totient*[*of 4*] **by** *simp*

hence *Carmichael 4 ≤ 2* **by** (*rule dvd-imp-le*) *auto*
moreover have *Carmichael 4 ≠ 1*
using *power-Carmichael-eq-1*[*of 4*::*nat 3*]
unfolding *coprime-iff-gcd-eq-1* **by** (*auto simp*: *gcd-non-0-nat cong-def*)
ultimately show *?thesis*
using *Carmichael-pos*[*of 4*] **by** *linarith*
qed

lemma *residue-primroot-Carmichael*:
assumes *residue-primroot n g*
shows *Carmichael n = totient n*
proof (*cases n = 1*)
case *False*
show *?thesis*
proof (*intro dvd-antisym Carmichael-dvd-totient*)
have *ord n g dvd Carmichael n*
using *assms False* **by** (*intro ord-dvd-Carmichael*) (*auto simp*: *residue-primroot-def*)
thus *totient n dvd Carmichael n*
using *assms* **by** (*auto simp*: *residue-primroot-def*)
qed
qed *auto*

lemma *Carmichael-odd-prime-power*:
assumes *prime p odd p k > 0*
shows *Carmichael (p ^ k) = p ^ (k − 1) ∗ (p − 1)*
proof −
from *assms* **obtain** *g* **where** *residue-primroot (p ^ k) g*
using *residue-primroot-odd-prime-power-exists*[*of p*] *assms* **by** *metis*
hence *Carmichael (p ^ k) = totient (p ^ k)*
by (*intro residue-primroot-Carmichael*[*of p ^ k g*]) *auto*
with *assms* **show** *?thesis* **by** (*simp add*: *totient-prime-power*)
qed

lemma *Carmichael-prime*:
assumes *prime p*
shows *Carmichael p = p − 1*
proof (*cases even p*)
case *True*
with *assms* **have** *p = 2*
using *primes-dvd-imp-eq two-is-prime-nat* **by** *blast*
thus *?thesis* **by** *simp*
next
case *False*
with *Carmichael-odd-prime-power*[*of p 1*] *assms* **show** *?thesis* **by** *simp*
qed

lemma *Carmichael-twopow-ge-8*:
assumes *k ≥ 3*
shows *Carmichael (2 ^ k) = 2 ^ (k − 2)*

proof (*intro dvd-antisym*)
have $2 \hat{} (k - 2) = ord\ (2 \hat{} k)\ (3 :: nat)$
using *ord-twopow-3-5*[*of k 3*] *assms* **by** *simp*
also have ... *dvd Carmichael* $(2 \hat{} k)$
using *assms one-less-power*[*of 2*::*nat k*] **by** (*intro ord-dvd-Carmichael*) *auto*
finally show $2 \hat{} (k - 2)$ *dvd* ... **.**
next
show *Carmichael* $(2 \hat{} k)$ *dvd* $2 \hat{} (k - 2)$
unfolding *Carmichael-def*
proof (*intro Lcm-least*, *safe*)
fix *x* **assume** $x \in totatives\ (2 \hat{} k)$
hence *odd x* **by** (*auto simp*: *totatives-def*)
hence $[x \hat{} 2 \hat{} (k - 2) = 1]\ (mod\ 2 \hat{} k)$
using *assms ord-twopow-aux*[*of k x*] **by** *auto*
thus *ord* $(2 \hat{} k)$ *x dvd* $2 \hat{} (k - 2)$
by (*simp add*: *ord-divides'*)
qed
qed

lemma *Carmichael-twopow*:
Carmichael $(2 \hat{} k) = (if\ k \leq 2\ then\ 2 \hat{} (k - 1)\ else\ 2 \hat{} (k - 2))$
proof −
have $k = 0 \vee k = 1 \vee k = 2 \vee k \geq 3$ **by** *linarith*
thus *?thesis* **by** (*auto simp*: *Carmichael-twopow-ge-8*)
qed

lemma *Carmichael-prime-power*:
assumes *prime p k* > *0*
shows *Carmichael* $(p \hat{} k) =$
$(if\ p = 2 \wedge k > 2\ then\ 2 \hat{} (k - 2)\ else\ p \hat{} (k - 1) * (p - 1))$
proof (*cases p* = *2*)
case *True*
thus *?thesis* **by** (*simp add*: *Carmichael-twopow*)
next
case *False*
with *assms* **have** *odd p p* > *2*
using *prime-odd-nat*[*of p*] *prime-gt-1-nat*[*of p*] **by** *auto*
thus *?thesis*
using *assms Carmichael-odd-prime-power*[*of p k*] **by** *simp*
qed

lemma *Carmichael-prod-coprime*:
assumes *finite A* $\bigwedge i\ j.\ i \in A \Longrightarrow j \in A \Longrightarrow i \neq j \Longrightarrow coprime\ (f\ i)\ (f\ j)$
shows *Carmichael* $(\prod i \in A.\ f\ i) = (LCM\ i \in A.\ Carmichael\ (f\ i))$
using *assms* **by** (*induction A rule*: *finite-induct*)
(*simp*, *simp*, *subst Carmichael-mult-coprime*[*OF prod-coprime-right*],
auto)

Since λ distributes over coprime factors and we know the value of $\lambda(p^k)$ for

prime p, we can now give a closed formula for $\lambda(n)$ in terms of the prime factorisation of n:

theorem *Carmichael-closed-formula*:
  Carmichael n =
    (*LCM p*∈*prime-factors n*. *let k* = *multiplicity p n*
                    in if p = *2* ∧ *k* > *2 then 2* ^ (*k* − *2*) *else p* ^ (*k* − *1*) ∗
(*p* − *1*))
  (**is** - = *Lcm ?A*)
proof (*cases n* = *0*)
  case *False*
  hence *n* = (∏ *p*∈*prime-factors n*. *p* ^ *multiplicity p n*)
    using *prime-factorization-nat* **by** *blast*
  also have *Carmichael* ... =
          (*LCM p*∈*prime-factors n*. *Carmichael* (*p* ^ *multiplicity p n*))
   by (*subst Carmichael-prod-coprime*) (*auto simp*: *in-prime-factors-iff primes-coprime*)
  also have (λ*p*. *Carmichael* (*p* ^ *multiplicity p n*)) ‘ *prime-factors n* = *?A*
    by (*intro image-cong*)
      (*auto simp*: *Let-def Carmichael-prime-power prime-factors-multiplicity*)
  finally show *?thesis* **.**
qed *auto*

corollary *even-Carmichael*:
  assumes *n* > *2*
  shows *even* (*Carmichael n*)
proof (*cases* ∃ *k*. *n* = *2* ^ *k*)
  case *True*
  then obtain *k* **where** [*simp*]: *n* = *2* ^ *k* **by** *auto*
  from *assms* **have** *k* ≠ *0 k* ≠ *1* **by** (*auto intro*!: *Nat.gr0I*)
  hence *k* ≥ *2* **by** *auto*
  thus *?thesis* **by** (*auto simp*: *Carmichael-twopow*)
next
  case *False*
  from *assms* **have** *n* ≠ *0* **by** *auto*
  from *False* **have** ∃ *p*∈*prime-factors n*. *p* ≠ *2*
    using *assms Ex-other-prime-factor*[*of n 2*] **by** *auto*
  from *divide-out-primepow-ex*[*OF* ‹*n* ≠ *0*› *this*]
  obtain *p k n*′
    where *p*:
      p ≠ *2*
      prime p
      p dvd n
      ¬ *p dvd n*′
      0 < *k*
      n = *p* ^ *k* ∗ *n*′ **.**
  from *p* **have** *coprime*: *coprime* (*p* ^ *k*) *n*′
    using *p prime-imp-coprime* **by** *auto*
  have *odd p*
    using *p primes-dvd-imp-eq*[*of 2 p*] **by** *auto*
  have *even* (*Carmichael* (*p* ^ *k*))

```
    using p ‹odd p› by (auto simp: Carmichael-prime-power)
  with p coprime show ?thesis
    by (auto simp: Carmichael-mult-coprime intro!: dvd-lcmI1)
qed

lemma eval-Carmichael:
  assumes prime-factorization n = A
  shows   Carmichael n = (LCM p ∈ set-mset A.
            let k = count A p in if p = 2 ∧ k > 2 then 2 ^ (k - 2) else p ^ (k -
1) * (p - 1))
  unfolding assms [symmetric] Carmichael-closed-formula
 by (intro arg-cong[where f = Lcm] image-cong) (auto simp: Let-def count-prime-factorization)
```

Any residue ring always contains a λ-root, i. e. an element whose order is $\lambda(n)$.

```
theorem Carmichael-root-exists:
  assumes n > (0::nat)
  obtains g where g ∈ totatives n and ord n g = Carmichael n
proof (cases n = 1)
  case True
  thus ?thesis by (intro that[of 1]) (auto simp: totatives-def)
next
  case False
  have primepow: ∃ g. coprime (p ^ k) g ∧ ord (p ^ k) g = Carmichael (p ^ k)
    if pk: prime p k > 0 for p k
  proof (cases p = 2)
    case [simp]: True
    from ‹k > 0› consider k = 1 | k = 2 | k ≥ 3 by force
    thus ?thesis
    proof cases
      assume k = 1
      thus ?thesis by (intro exI[of - 1]) auto
    next
      assume [simp]: k = 2
      have coprime 4 (3::nat)
        by (auto simp: coprime-iff-gcd-eq-1 gcd-non-0-nat)
      thus ?thesis by (intro exI[of - 3]) auto
    next
      assume k: k ≥ 3
      have coprime (2 ^ k :: nat) 3 by auto
      thus ?thesis using k
        by (intro exI[of - 3]) (auto simp: ord-twopow-3-5 Carmichael-twopow)
    qed
  next
    case False
    hence odd p using ‹prime p›
      using primes-dvd-imp-eq two-is-prime-nat by blast
    then obtain g where residue-primroot (p ^ k) g
      using residue-primroot-odd-prime-power-exists[of p] pk by metis
```

```
    thus ?thesis using False pk
      by (intro exI[of - g])
       (auto simp: Carmichael-prime-power residue-primroot-def totient-prime-power)
  qed

  define P where P = prime-factors n
  define k where k = (λp. multiplicity p n)
  have ∀ p∈P. ∃ g. coprime (p ^ k p) g ∧ ord (p ^ k p) g = Carmichael (p ^ k p)
    using primepow by (auto simp: P-def k-def prime-factors-multiplicity)
  hence ∃ g. ∀ p∈P. coprime (p ^ k p) (g p) ∧ ord (p ^ k p) (g p) = Carmichael
(p ^ k p)
    by (subst (asm) bchoice-iff)
  then obtain g where g: ⋀p. p ∈ P ⟹ coprime (p ^ k p) (g p)
                         ⋀p. p ∈ P ⟹ ord (p ^ k p) (g p) = Carmichael (p ^ k p) by
metis
  have ∃ x. ∀ i∈P. [x = g i] (mod i ^ k i)
    by (intro chinese-remainder-nat)
       (auto simp: P-def k-def in-prime-factors-iff primes-coprime)
  then obtain x where x: ⋀p. p ∈ P ⟹ [x = g p] (mod p ^ k p) by metis

  have n = (∏ p∈P. p ^ k p)
    using assms unfolding P-def k-def by (rule prime-factorization-nat)
  also have ord ... x = (LCM p∈P. ord (p ^ k p) x)
     by (intro ord-modulus-prod-coprime) (auto simp: P-def in-prime-factors-iff
primes-coprime)
  also have (λp. ord (p ^ k p) x) ‘ P = (λp. ord (p ^ k p) (g p)) ‘ P
    by (intro image-cong ord-cong x) auto
  also have ... = (λp. Carmichael (p ^ k p)) ‘ P
    by (intro image-cong g) auto
  also have Lcm ... = Carmichael (∏ p∈P. p ^ k p)
    by (intro Carmichael-prod-coprime [symmetric])
       (auto simp: P-def in-prime-factors-iff primes-coprime)
  also have (∏ p∈P. p ^ k p) = n
   using assms unfolding P-def k-def by (rule prime-factorization-nat [symmetric])
  finally have ord n x = Carmichael n .
  moreover from this have coprime n x
    by (cases coprime n x) (auto split: if-splits)
  ultimately show ?thesis using assms False
    by (intro that[of x mod n])
     (auto simp: totatives-def coprime-commute coprime-absorb-left intro!: Nat.gr0I)
qed
```

This also means that the Carmichael number is not only the LCM of the orders of the elements of the residue ring, but indeed the maximum of the orders.

```
lemma Carmichael-altdef:
  Carmichael n = (if n = 0 then 1 else Max (ord n ‘ totatives n))
proof (cases n = 0)
  case False
```

```
  have Carmichael n = Max (ord n ‘ totatives n)
  proof (intro antisym Max.boundedI Max.coboundedI)
    fix k assume k: k ∈ ord n ‘ totatives n
    thus k ≤ Carmichael n
    proof (cases n = 1)
      case False
      with ‹n ≠ 0› have n > 1 by linarith
      thus ?thesis using k ‹n ≠ 0›
        by (intro dvd-imp-le)
           (auto intro!: ord-dvd-Carmichael simp: totatives-def coprime-commute)
    qed auto
  next
    obtain g where g ∈ totatives n and ord n g = Carmichael n
      using Carmichael-root-exists[of n] ‹n ≠ 0› by auto
    thus Carmichael n ∈ ord n ‘ totatives n by force
  qed (use ‹n ≠ 0› in auto)
  thus ?thesis using False by simp
qed auto
```

11.5 Existence of primitive roots for general moduli

We now related Carmichael's function to the existence of primitive roots and, in the end, use this to show precisely which moduli have primitive roots and which do not.

The first criterion for the existence of a primitive root is this: A primitive root modulo n exists iff $\lambda(n) = \varphi(n)$.

```
lemma Carmichael-eq-totient-imp-primroot:
  assumes n > 0 and Carmichael n = totient n
  shows   ∃g. residue-primroot n g
proof −
  from ‹n > 0› obtain g where g ∈ totatives n and ord n g = Carmichael n
    using Carmichael-root-exists[of n] by metis
  with assms show ?thesis by (auto simp: residue-primroot-def totatives-def co-
prime-commute)
qed

theorem residue-primroot-iff-Carmichael:
  (∃g. residue-primroot n g) ⟷ Carmichael n = totient n ∧ n > 0
proof safe
  fix g assume g: residue-primroot n g
  thus n > 0 by (auto simp: residue-primroot-def)
  have coprime n g by (rule ccontr) (use g in ‹auto simp: residue-primroot-def›)

  show Carmichael n = totient n
  proof (cases n = 1)
    case False
    with ‹n > 0› have n > 1 by auto
    with ‹coprime n g› have ord n g dvd Carmichael n
```

```
    by (intro ord-dvd-Carmichael) auto
  thus ?thesis using g by (intro dvd-antisym Carmichael-dvd-totient)
                          (auto simp: residue-primroot-def)
 qed auto
qed (use Carmichael-eq-totient-imp-primroot[of n] in auto)
```

Any primitive root modulo mn for coprime m, n is also a primitive root modulo m and n. The converse does not hold in general.

```
lemma residue-primroot-modulus-mult-coprimeD:
 assumes coprime m n and residue-primroot (m * n) g
 shows   residue-primroot m g residue-primroot n g
proof −
 have *: m > 0 n > 0 coprime m g coprime n g
       lcm (ord m g) (ord n g) = totient m * totient n
   using assms
  by (auto simp: residue-primroot-def ord-modulus-mult-coprime totient-mult-coprime)

 define a b where a = totient m div ord m g and b = totient n div ord n g
 have ab: totient m = ord m g * a totient n = ord n g * b
   using * by (auto simp: a-def b-def order-divides-totient)

 have a = 1 b = 1 coprime (ord m g) (ord n g)
  unfolding coprime-iff-gcd-eq-1 using * by (auto simp: ab lcm-nat-def dvd-div-eq-mult
ord-eq-0)
 with ab and * show residue-primroot m g residue-primroot n g
   by (auto simp: residue-primroot-def)
qed
```

If a primitive root modulo mn exists for coprime m, n, then $\lambda(m)$ and $\lambda(n)$ must also be coprime. This is helpful in establishing that there are no primitive roots modulo mn by showing e. g. that $\lambda(m)$ and $\lambda(n)$ are both even.

```
lemma residue-primroot-modulus-mult-coprime-imp-Carmichael-coprime:
 assumes coprime m n and residue-primroot (m * n) g
 shows   coprime (Carmichael m) (Carmichael n)
proof −
 from residue-primroot-modulus-mult-coprimeD[OF assms]
   have *: residue-primroot m g residue-primroot n g by auto
 hence [simp]: Carmichael m = totient m Carmichael n = totient n
   by (simp-all add: residue-primroot-Carmichael)
 from * have mn: m > 0 n > 0 by (auto simp: residue-primroot-def)

 from assms have Carmichael (m * n) = totient (m * n) ∧ n > 0
   using residue-primroot-iff-Carmichael[of m * n] by auto
 with assms have lcm (totient m) (totient n) = totient m * totient n
   by (simp add: Carmichael-mult-coprime totient-mult-coprime)
 thus ?thesis unfolding coprime-iff-gcd-eq-1 using mn
   by (simp add: lcm-nat-def dvd-div-eq-mult)
qed
```

The following moduli are precisely those that have primitive roots.

definition *cyclic-moduli* :: *nat set* **where**
  cyclic-moduli = {*1*, *2*, *4*} ∪ {*p* ^ *k* |*p k*. *prime p* ∧ *odd p* ∧ *k* > *0*} ∪
                {*2* ∗ *p* ^ *k* |*p k*. *prime p* ∧ *odd p* ∧ *k* > *0*}

theorem *residue-primroot-iff-in-cyclic-moduli*:
  (∃ *g*. *residue-primroot m g*) ⟷ *m* ∈ *cyclic-moduli*
proof −
  have (∃ *g*. *residue-primroot m g*) **if** *m* ∈ *cyclic-moduli*
  using *that* **unfolding** *cyclic-moduli-def*
  by (*intro Carmichael-eq-totient-imp-primroot*)
    (*auto dest*: *prime-gt-0-nat simp*: *Carmichael-prime-power totient-prime-power*
          Carmichael-mult-coprime totient-mult-coprime)

  moreover have ¬(∃ *g*. *residue-primroot m g*) **if** *m* ∉ *cyclic-moduli*
  proof (*cases m* = *0*)
    case *False*
    with *that* **have** [*simp*]: *m* > *0 m* ≠ *1* **by** (*auto simp*: *cyclic-moduli-def*)
    show *?thesis*
    proof (*cases* ∃ *k*. *m* = *2* ^ *k*)
      case *True*
      then obtain *k* **where** [*simp*]: *m* = *2* ^ *k* **by** *auto*
      with *that* **have** *k* ≠ *0* ∧ *k* ≠ *1* ∧ *k* ≠ *2* **by** (*auto simp*: *cyclic-moduli-def*)
      hence *k* ≥ *3* **by** *auto*
      thus *?thesis* **by** (*subst residue-primroot-iff-Carmichael*)
                (*auto simp*: *Carmichael-twopow totient-prime-power*)
    next
      case *False*
      hence ∃ *p*∈*prime-factors m*. *p* ≠ *2*
        using *Ex-other-prime-factor*[*of m 2*] **by** *auto*
      from *divide-out-primepow-ex*[*OF* ‹*m* ≠ *0*› *this*]
      obtain *p k m*′ **where** *p*: *p* ≠ *2 prime p p dvd m* ¬*p dvd m*′ *0* < *k m* = *p* ^ *k*
∗ *m*′
        by *metis*
      have *odd p*
        using *p primes-dvd-imp-eq*[*of 2 p*] **by** *auto*
      from *p* **have** *coprime*: *coprime* (*p* ^ *k*) *m*′
        using *p prime-imp-coprime* **by** *auto*
      have *m* ∈ *cyclic-moduli* **if** *m*′ = *1*
        using *that p* ‹*odd p*› **by** (*auto simp*: *cyclic-moduli-def*)
      moreover have *m* ∈ *cyclic-moduli* **if** *m*′ = *2*
      proof −
        have *m* = *2* ∗ *p* ^ *k* **using** *p that* **by** (*simp add*: *mult.commute*)
        thus *m* ∈ *cyclic-moduli* **using** *p* ‹*odd p*›
          unfolding *cyclic-moduli-def* **by** *blast*
      qed
      moreover have *m*′ ≠ *0* **using** *p* **by** (*intro notI*) *auto*
      ultimately have *m*′ ≠ *0* ∧*m*′ ≠ *1* ∧ *m*′ ≠ *2* **using** *that* **by** *auto*
      hence *m*′ > *2* **by** *linarith*

```
      show ?thesis
      proof
        assume ∃ g. residue-primroot m g
        with coprime p have coprime': coprime (p − 1) (Carmichael m')
         using residue-primroot-modulus-mult-coprime-imp-Carmichael-coprime[OF
coprime]
          by (auto simp: Carmichael-prime-power)
        moreover have even (p − 1) even (Carmichael m')
          using ‹m' > 2› ‹odd p› by (auto intro!: even-Carmichael)
        ultimately show False by force
      qed
    qed
  qed auto

  ultimately show ?thesis by metis
qed

lemma residue-primroot-is-generator:
  assumes m > 1 and residue-primroot m g
  shows   bij-betw (λi. g ^ i mod m) {..<totient m} (totatives m)
  unfolding bij-betw-def
proof
  from assms have [simp]: ord m g = totient m
    by (simp add: residue-primroot-def)
  from assms have coprime m g by (simp add: residue-primroot-def)
  hence inj-on (λi. g ^ i mod m) {..<ord m g}
    by (intro inj-power-mod)
  thus inj: inj-on (λi. g ^ i mod m) {..<totient m}
    by simp

  show (λi. g ^ i mod m) ` {..<totient m} = totatives m
  proof (rule card-subset-eq)
    show card ((λi. g ^ i mod m) ` {..<totient m}) = card (totatives m)
      using inj by (subst card-image) (auto simp: totient-def)
    show (λi. g ^ i mod m) ` {..<totient m} ⊆ totatives m
      using ‹m > 1› ‹coprime m g› power-in-totatives[of m g] by blast
  qed auto
qed
```

Given one primitive root g, all the primitive roots are powers g^i for $1 \leq i \leq \varphi(n)$ with $\gcd(i, \varphi(n)) = 1$.

```
theorem residue-primroot-bij-betw-primroots:
  assumes m > 1 and residue-primroot m g
  shows   bij-betw (λi. g ^ i mod m) (totatives (totient m))
                                {g∈totatives m. residue-primroot m g}
proof (cases m = 2)
  case [simp]: True
  have [simp]: totatives 2 = {1}
```

```
    by (auto simp: totatives-def elim!: oddE)
  from assms have odd g
    by (auto simp: residue-primroot-def)
  hence pow-eq: (λi. g ^ i mod m) = (λ-. 1)
    by (auto simp: fun-eq-iff mod-2-eq-odd)
  have {g ∈ totatives m. residue-primroot m g} = {1}
    by (auto simp: residue-primroot-def)
  thus ?thesis using pow-eq by (auto simp: bij-betw-def)
next
  case False
  thus ?thesis unfolding bij-betw-def
  proof safe
    from assms False have m > 2 by auto
    from assms ‹m > 2› have totient m > 1 by (intro totient-gt-1) auto
    from assms have [simp]: ord m g = totient m
      by (simp add: residue-primroot-def)
    from assms have coprime m g by (simp add: residue-primroot-def)
    hence inj-on (λi. g ^ i mod m) {..<ord m g}
      by (intro inj-power-mod)
    thus inj-on (λi. g ^ i mod m) (totatives (totient m))
      by (rule inj-on-subset)
       (use assms ‹totient m > 1› in ‹auto simp: totatives-less residue-primroot-def›)

    {
      fix i assume i: i ∈ totatives (totient m)
      from ‹coprime m g› and ‹m > 2› show g ^ i mod m ∈ totatives m
        by (intro power-in-totatives) auto
      show residue-primroot m (g ^ i mod m)
        using i ‹m > 2› ‹coprime m g›
      by (auto simp: residue-primroot-def coprime-commute ord-power totatives-def)
    }
    {
      fix x assume x: x ∈ totatives m residue-primroot m x
      then obtain i where i: i < totient m x = (g ^ i mod m)
      using assms residue-primroot-is-generator[of m g] by (auto simp: bij-betw-def)
    from i x ‹m > 2› have i > 0 by (intro Nat.gr0I) (auto simp: residue-primroot-1-iff)
      have totient m div gcd i (totient m) = totient m
       using x i ‹coprime m g› by (auto simp add: residue-primroot-def ord-power)
      hence coprime i (totient m)
      unfolding coprime-iff-gcd-eq-1 using assms by (subst (asm) dvd-div-eq-mult)
auto
      with i ‹i > 0› have i ∈ totatives (totient m) by (auto simp: totatives-def)
      thus x ∈ (λi. g ^ i mod m) ‘ totatives (totient m) using i by auto
    }
  qed
qed
```

It follows from the above statement that any residue ring modulo n that *has* primitive roots has exactly $\varphi(\varphi(n))$ of them.

corollary *card-residue-primroots*:
  assumes $\exists g.$ *residue-primroot m g*
  shows *card* {*g*∈*totatives m. residue-primroot m g*} = *totient* (*totient m*)
proof (*cases m = 1*)
  case [*simp*]: *True*
  have {*g* ∈ *totatives m. residue-primroot m g*} = {*1*}
    by (*auto simp*: *residue-primroot-def*)
  thus *?thesis* **by** *simp*
next
  case *False*
  from *assms* **obtain** *g* **where** *g*: *residue-primroot m g* **by** *auto*
  hence *m* ≠ *0* **by** (*intro notI*) *auto*
  with ‹*m* ≠ *1*› **have** *m* > *1* **by** *linarith*
  from *this g* **have** *bij-betw* (λ*i*. *g* ^ *i mod m*) (*totatives* (*totient m*))
                    {*g*∈*totatives m. residue-primroot m g*}
    by (*rule residue-primroot-bij-betw-primroots*)
  hence *card* (*totatives* (*totient m*)) = *card* {*g*∈*totatives m. residue-primroot m g*}
    by (*rule bij-betw-same-card*)
  thus *?thesis* **by** (*simp add*: *totient-def*)
qed

corollary *card-residue-primroots′*:
  card {*g*∈*totatives m. residue-primroot m g*} =
    (*if m* ∈ *cyclic-moduli then totient* (*totient m*) *else 0*)
 by (*simp add*: *residue-primroot-iff-in-cyclic-moduli* [*symmetric*] *card-residue-primroots*)

As an example, we evaluate $\lambda(122200)$ using the prime factorisation.

lemma *Carmichael 122200 = 1380*
proof −
 have *prime-factorization* (*2*^*3* ∗ *5*^*2* ∗ *13* ∗ *47*) = {#*2*, *2*, *2*, *5*, *5*, *13*, *47*::*nat*#}
   by (*intro prime-factorization-eqI*) *auto*
 from *eval-Carmichael*[*OF this*] **show** *Carmichael 122200 = 1380*
   by (*simp add*: *lcm-nat-def gcd-non-0-nat*)
qed

end

12 Modular Inverses

theory *Modular-Inverse*
  imports *Cong HOL−Library.FuncSet*
begin

The following returns the unique number m such that $mn \equiv 1 \pmod{p}$ if there is one, i.e. if n and p are coprime, and otherwise 0 by convention.

definition *modular-inverse* **where**

modular-inverse p n = (*if coprime p n then fst* (*bezout-coefficients n p*) *mod p else 0*)

lemma *cong-modular-inverse1*:
assumes *coprime n p*
shows [*n* ∗ *modular-inverse p n* = *1*] (*mod p*)
proof −
have [*fst* (*bezout-coefficients n p*) ∗ *n* + *snd* (*bezout-coefficients n p*) ∗ *p* = *modular-inverse p n* ∗ *n* + *0*] (*mod p*)
unfolding *modular-inverse-def* **using** *assms*
by (*intro cong-add cong-mult*) (*auto simp*: *Cong.cong-def coprime-commute*)
also have *fst* (*bezout-coefficients n p*) ∗ *n* + *snd* (*bezout-coefficients n p*) ∗ *p* = *gcd n p*
by (*simp add*: *bezout-coefficients-fst-snd*)
also have ... = *1*
using *assms* **by** *simp*
finally show *?thesis*
by (*simp add*: *cong-sym mult-ac*)
qed

lemma *cong-modular-inverse2*:
assumes *coprime n p*
shows [*modular-inverse p n* ∗ *n* = *1*] (*mod p*)
using *cong-modular-inverse1*[*OF assms*] **by** (*simp add*: *mult.commute*)

lemma *coprime-modular-inverse* [*simp*, *intro*]:
fixes *n* :: *'a* :: {*euclidean-ring-gcd*,*unique-euclidean-semiring*}
assumes *coprime n p*
shows *coprime* (*modular-inverse p n*) *p*
using *cong-modular-inverse1*[*OF assms*] *assms*
by (*meson cong-imp-coprime cong-sym coprime-1-left coprime-mult-left-iff*)

lemma *modular-inverse-int-nonneg*: *p* > *0* ⟹ *modular-inverse p* (*n* :: *int*) ≥ *0*
by (*simp add*: *modular-inverse-def*)

lemma *modular-inverse-int-less*: *p* > *0* ⟹ *modular-inverse p* (*n* :: *int*) < *p*
by (*simp add*: *modular-inverse-def*)

lemma *modular-inverse-int-eqI*:
fixes *x y* :: *int*
assumes *y* ∈ {*0*..<*m*} [*x* ∗ *y* = *1*] (*mod m*)
shows *modular-inverse m x* = *y*
proof −
from *assms* **have** *coprime x m*
using *cong-gcd-eq* **by** *force*
have [*modular-inverse m x* ∗ *1* = *modular-inverse m x* ∗ (*x* ∗ *y*)] (*mod m*)
by (*rule cong-sym*, *intro cong-mult assms cong-refl*)
also have *modular-inverse m x* ∗ (*x* ∗ *y*) = (*modular-inverse m x* ∗ *x*) ∗ *y*
by (*simp add*: *mult-ac*)

```
  also have [... = 1 * y] (mod m)
    using ‹coprime x m› by (intro cong-mult cong-refl cong-modular-inverse2)
  finally have [modular-inverse m x = y] (mod m)
    by simp
  thus modular-inverse m x = y
    using assms by (simp add: Cong.cong-def modular-inverse-def)
qed

lemma modular-inverse-1 [simp]:
  assumes m > (1 :: int)
  shows   modular-inverse m 1 = 1
  by (rule modular-inverse-int-eqI) (use assms in auto)

lemma modular-inverse-int-mult:
  fixes x y :: int
  assumes coprime x m coprime y m m > 0
  shows modular-inverse m (x * y) = (modular-inverse m y * modular-inverse m
x) mod m
proof (rule modular-inverse-int-eqI)
  show modular-inverse m y * modular-inverse m x mod m ∈ {0..<m}
    using assms by auto
next
  have [x * y * (modular-inverse m y * modular-inverse m x mod m) =
        x * y * (modular-inverse m y * modular-inverse m x)] (mod m)
    by (intro cong-mult cong-refl) auto
  also have x * y * (modular-inverse m y * modular-inverse m x) =
            (x * modular-inverse m x) * (y * modular-inverse m y)
    by (simp add: mult-ac)
  also have [... = 1 * 1] (mod m)
    by (intro cong-mult cong-modular-inverse1 assms)
  finally show [x * y * (modular-inverse m y * modular-inverse m x mod m) =
1] (mod m)
    by simp
qed

lemma bij-betw-int-remainders-mult:
  fixes a n :: int
  assumes a: coprime a n
  shows   bij-betw (λm. a * m mod n) {1..<n} {1..<n}
proof −
  define a' where a' = modular-inverse n a

  have *: a' * (a * m mod n) mod n = m ∧ a * m mod n ∈ {1..<n}
    if a: [a * a' = 1] (mod n) and m: m ∈ {1..<n} for m a a' :: int
  proof
    have [a' * (a * m mod n) = a' * (a * m)] (mod n)
      by (intro cong-mult cong-refl) (auto simp: Cong.cong-def)
    also have a' * (a * m) = (a * a') * m
      by (simp add: mult-ac)
```

also have $[(a * a') * m = 1 * m]$ $(mod\ n)$
unfolding a'*-def* **by** (*intro cong-mult cong-refl*) (*use a* **in** *auto*)
finally show $a' * (a * m\ mod\ n)\ mod\ n = m$
using m **by** (*simp add*: *Cong.cong-def*)
next
have *coprime a n*
using *a coprime-iff-invertible-int* **by** *auto*
hence $\neg n\ dvd\ (a * m)$
using m **by** (*simp add*: *coprime-commute coprime-dvd-mult-right-iff zdvd-not-zless*)
hence $a * m\ mod\ n > 0$
using m *order-le-less* **by** *fastforce*
thus $a * m\ mod\ n \in \{1..<n\}$
using m **by** *auto*
qed

have $[a * a' = 1]$ $(mod\ n)$ $[a' * a = 1]$ $(mod\ n)$
unfolding a'*-def* **by** (*rule cong-modular-inverse1 cong-modular-inverse2*; *fact*)+
from *this*[*THEN* $*$] **show** *?thesis*
by (*intro bij-betwI*[*of - - -* $\lambda m.\ a' * m\ mod\ n$]) *auto*
qed

lemma *mult-modular-inverse-of-not-coprime* [*simp*]: $\neg$*coprime a m* $\Longrightarrow$ *modular-inverse m a = 0*
by (*simp add*: *coprime-commute modular-inverse-def*)

lemma *mult-modular-inverse-eq-0-iff*:
fixes *a* :: $'a$ :: {*unique-euclidean-semiring*, *euclidean-ring-gcd*}
shows $\neg$*is-unit m* $\Longrightarrow$ *modular-inverse m a = 0* $\longleftrightarrow$ $\neg$*coprime a m*
by (*metis coprime-modular-inverse mult-modular-inverse-of-not-coprime coprime-0-left-iff*)

lemma *mult-modular-inverse-int-pos*: *m > 1* $\Longrightarrow$ *coprime a m* $\Longrightarrow$ *modular-inverse m a > (0* :: *int)*
by (*simp add*: *modular-inverse-int-nonneg mult-modular-inverse-eq-0-iff order.strict-iff-order*)

lemma *abs-mult-modular-inverse-int-less*: *m* $\neq$ *0* $\Longrightarrow$ $|$*modular-inverse m a* :: *int*$|$ $<$ $|m|$
by (*auto simp*: *modular-inverse-def intro*!: *abs-mod-less*)

lemma *modular-inverse-int-less'*: *m* $\neq$ *0* $\Longrightarrow$ (*modular-inverse m a* :: *int*) $<$ $|m|$
using *abs-mult-modular-inverse-int-less*[*of m a*] **by** *linarith*

end

13 Comprehensive number theory

theory *Number-Theory*
imports
Fib
Residues

Eratosthenes
Mod-Exp
Quadratic-Reciprocity
Pocklington
Prime-Powers
Residue-Primitive-Roots
Modular-Inverse
begin

end