

Some results of number theory

Jeremy Avigad
David Gray
Adam Kramer
Thomas M Rasmussen

June 21, 2010

Abstract

This is a collection of formalized proofs of many results of number theory. The proofs of the Chinese Remainder Theorem and Wilson's Theorem are due to Rasmussen. The proof of Gauss's law of quadratic reciprocity is due to Avigad, Gray and Kramer. Proofs can be found in most introductory number theory textbooks; Goldman's *The Queen of Mathematics: a Historically Motivated Guide to Number Theory* provides some historical context.

Avigad, Gray and Kramer have also provided library theories dealing with finite sets and finite sums, divisibility and congruences, parity and residues. The authors are engaged in redesigning and polishing these theories for more serious use. For the latest information in this respect, please see the web page <http://www.andrew.cmu.edu/~avigad/isabelle>. Other theories contain proofs of Euler's criteria, Gauss' lemma, and the law of quadratic reciprocity. The formalization follows Eisenstein's proof, which is the one most commonly found in introductory textbooks; in particular, it follows the presentation in Niven and Zuckerman, *The Theory of Numbers*.

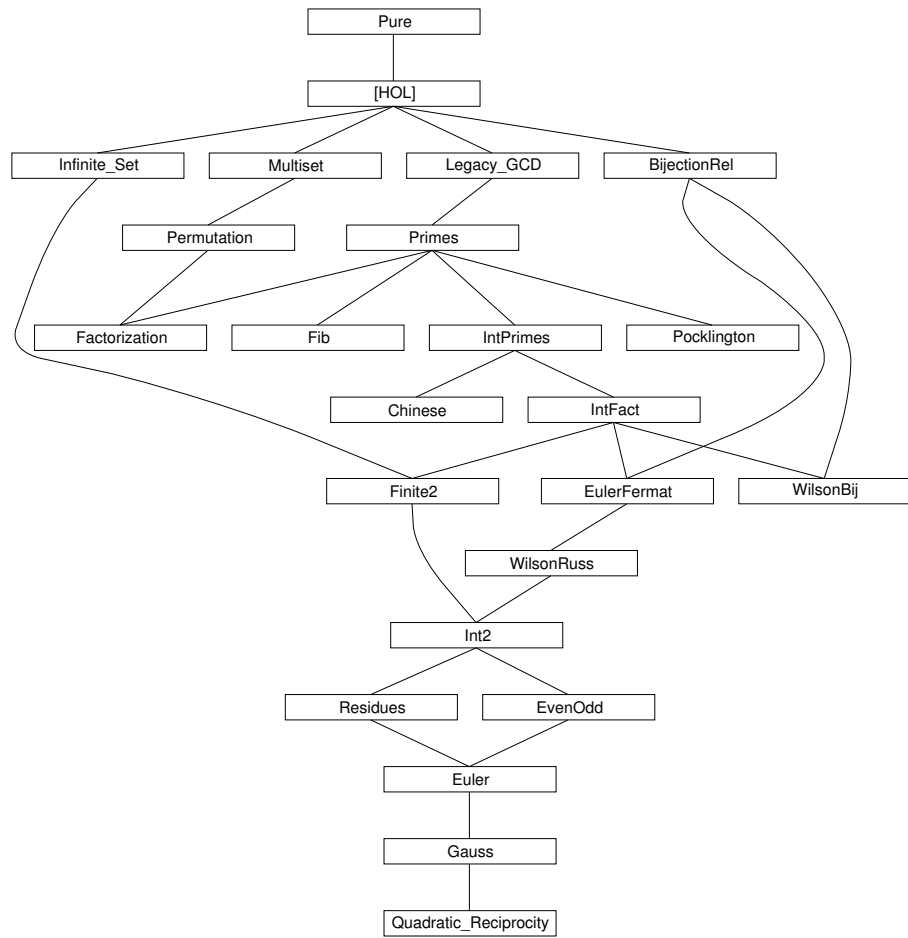
To avoid having to count roots of polynomials, however, we relied on a trick previously used by David Russinoff in formalizing quadratic reciprocity for the Boyer-Moore theorem prover; see Russinoff, David, "A mechanical proof of quadratic reciprocity," *Journal of Automated Reasoning* 8:3-21, 1992. We are grateful to Larry Paulson for calling our attention to this reference.

Contents

1	The Greatest Common Divisor	5
1.1	Specification of GCD on nats	5
1.2	GCD on nat by Euclid's algorithm	5
1.3	Derived laws for GCD	6
1.4	LCM defined by GCD	13
1.5	GCD and LCM on integers	16

2	Primality on nat	20
3	The Fibonacci function	38
4	Fundamental Theorem of Arithmetic (unique factorization into primes)	40
4.1	Definitions	41
4.2	Arithmetic	41
4.3	Prime list and product	42
4.4	Sorting	43
4.5	Permutation	44
4.6	Existence	45
4.7	Uniqueness	45
5	Divisibility and prime numbers (on integers)	47
5.1	Definitions	47
5.2	Euclid's Algorithm and GCD	48
5.3	Congruences	49
5.4	Modulo	53
5.5	Extended GCD	53
6	The Chinese Remainder Theorem	56
6.1	Definitions	56
6.2	Chinese: uniqueness	58
6.3	Chinese: existence	59
6.4	Chinese	60
7	Bijections between sets	61
8	Factorial on integers	65
9	Fermat's Little Theorem extended to Euler's Totient function	67
9.1	Definitions and lemmas	67
9.2	Fermat	72
10	Wilson's Theorem according to Russinoff	74
10.1	Definitions and lemmas	74
10.2	Wilson	79
11	Wilson's Theorem using a more abstract approach	80
11.1	Definitions and lemmas	80
11.2	Wilson	84

12 Finite Sets and Finite Sums	85
12.1 Useful properties of sums and products	85
12.2 Cardinality of explicit finite sets	86
12.3 Cardinality of finite cartesian products	89
13 Integers: Divisibility and Congruences	89
13.1 Useful lemmas about dvd and powers	90
13.2 Useful properties of congruences	91
13.3 Some properties of MultInv	93
14 Residue Sets	95
14.1 Some useful properties of StandardRes	96
14.2 Relations between StandardRes, SRStar, and SR	97
14.3 Properties relating ResSets with StandardRes	98
14.4 Property for SRStar	99
15 Parity: Even and Odd Integers	99
15.1 Some useful properties about even and odd	99
16 Euler's criterion	104
16.1 Property for MultInvPair	104
16.2 Properties of SetS	106
17 Gauss' Lemma	110
17.1 Basic properties of p	111
17.2 Basic Properties of the Gauss Sets	112
17.3 Relationships Between Gauss Sets	116
17.4 Gauss' Lemma	119
18 The law of Quadratic reciprocity	121
18.1 Stuff about S, S1 and S2	124
19 Pocklington's Theorem for Primes	134



1 The Greatest Common Divisor

theory *Legacy-GCD*
imports *Main*
begin

See [?].

1.1 Specification of GCD on nats

definition

$is_gcd :: nat \Rightarrow nat \Rightarrow nat \Rightarrow bool$ **where** — gcd as a relation
[*code del*]: $is_gcd\ m\ n\ p \iff p\ dvd\ m \wedge p\ dvd\ n \wedge$
 $(\forall d. d\ dvd\ m \longrightarrow d\ dvd\ n \longrightarrow d\ dvd\ p)$

Uniqueness

lemma $is_gcd_unique: is_gcd\ a\ b\ m \implies is_gcd\ a\ b\ n \implies m = n$
by (*simp add: is-gcd-def*) (*blast intro: dvd-antisym*)

Connection to divides relation

lemma $is_gcd_dvd: is_gcd\ a\ b\ m \implies k\ dvd\ a \implies k\ dvd\ b \implies k\ dvd\ m$
by (*auto simp add: is-gcd-def*)

Commutativity

lemma $is_gcd_commute: is_gcd\ m\ n\ k = is_gcd\ n\ m\ k$
by (*auto simp add: is-gcd-def*)

1.2 GCD on nat by Euclid's algorithm

fun

$gcd :: nat \Rightarrow nat \Rightarrow nat$

where

$gcd\ m\ n = (if\ n = 0\ then\ m\ else\ gcd\ n\ (m\ mod\ n))$

lemma gcd_induct [*case-names 0 rec*]:

fixes $m\ n :: nat$

assumes $\bigwedge m. P\ m\ 0$

and $\bigwedge m\ n. 0 < n \implies P\ n\ (m\ mod\ n) \implies P\ m\ n$

shows $P\ m\ n$

proof (*induct m n rule: gcd.induct*)

case ($1\ m\ n$) **with** *assms* **show** *?case* **by** (*cases n = 0*) *simp-all*

qed

lemma gcd_0 [*simp, algebra*]: $gcd\ m\ 0 = m$
by *simp*

lemma gcd_0_left [*simp, algebra*]: $gcd\ 0\ m = m$
by *simp*

lemma *gcd-non-0*: $n > 0 \implies \text{gcd } m \ n = \text{gcd } n \ (m \bmod n)$
by *simp*

lemma *gcd-1* [*simp*, *algebra*]: $\text{gcd } m \ (\text{Suc } 0) = \text{Suc } 0$
by *simp*

lemma *nat-gcd-1-right* [*simp*, *algebra*]: $\text{gcd } m \ 1 = 1$
unfolding *One-nat-def* **by** (*rule gcd-1*)

declare *gcd.simps* [*simp del*]

gcd m n divides *m* and *n*. The conjunctions don't seem provable separately.

lemma *gcd-dvd1* [*iff*, *algebra*]: $\text{gcd } m \ n \ \text{dvd } m$
and *gcd-dvd2* [*iff*, *algebra*]: $\text{gcd } m \ n \ \text{dvd } n$
apply (*induct m n rule: gcd-induct*)
apply (*simp-all add: gcd-non-0*)
apply (*blast dest: dvd-mod-imp-dvd*)
done

Maximality: for all *m*, *n*, *k* naturals, if *k* divides *m* and *k* divides *n* then *k* divides *gcd m n*.

lemma *gcd-greatest*: $k \ \text{dvd } m \implies k \ \text{dvd } n \implies k \ \text{dvd } \text{gcd } m \ n$
by (*induct m n rule: gcd-induct*) (*simp-all add: gcd-non-0 dvd-mod*)

Function *gcd* yields the Greatest Common Divisor.

lemma *is-gcd*: $\text{is-gcd } m \ n \ (\text{gcd } m \ n)$
by (*simp add: is-gcd-def gcd-greatest*)

1.3 Derived laws for GCD

lemma *gcd-greatest-iff* [*iff*, *algebra*]: $k \ \text{dvd } \text{gcd } m \ n \longleftrightarrow k \ \text{dvd } m \ \wedge \ k \ \text{dvd } n$
by (*blast intro!: gcd-greatest intro: dvd-trans*)

lemma *gcd-zero*[*algebra*]: $\text{gcd } m \ n = 0 \longleftrightarrow m = 0 \ \wedge \ n = 0$
by (*simp only: dvd-0-left-iff [symmetric] gcd-greatest-iff*)

lemma *gcd-commute*: $\text{gcd } m \ n = \text{gcd } n \ m$
apply (*rule is-gcd-unique*)
apply (*rule is-gcd*)
apply (*subst is-gcd-commute*)
apply (*simp add: is-gcd*)
done

lemma *gcd-assoc*: $\text{gcd } (\text{gcd } k \ m) \ n = \text{gcd } k \ (\text{gcd } m \ n)$
apply (*rule is-gcd-unique*)
apply (*rule is-gcd*)
apply (*simp add: is-gcd-def*)

```

apply (blast intro: dvd-trans)
done

```

```

lemma gcd-1-left [simp, algebra]:  $\text{gcd } (\text{Suc } 0) \ m = \text{Suc } 0$ 
by (simp add: gcd-commute)

```

```

lemma nat-gcd-1-left [simp, algebra]:  $\text{gcd } 1 \ m = 1$ 
unfolding One-nat-def by (rule gcd-1-left)

```

Multiplication laws

```

lemma gcd-mult-distrib2:  $k * \text{gcd } m \ n = \text{gcd } (k * m) \ (k * n)$ 
  — [?, page 27]
apply (induct m n rule: gcd-induct)
apply simp
apply (case-tac k = 0)
apply (simp-all add: mod-geq gcd-non-0 mod-mult-distrib2)
done

```

```

lemma gcd-mult [simp, algebra]:  $\text{gcd } k \ (k * n) = k$ 
apply (rule gcd-mult-distrib2 [of k 1 n, simplified, symmetric])
done

```

```

lemma gcd-self [simp, algebra]:  $\text{gcd } k \ k = k$ 
apply (rule gcd-mult [of k 1, simplified])
done

```

```

lemma relprime-dvd-mult:  $\text{gcd } k \ n = 1 \implies k \ \text{dvd } m * n \implies k \ \text{dvd } m$ 
apply (insert gcd-mult-distrib2 [of m k n])
apply simp
apply (erule-tac t = m in ssubst)
apply simp
done

```

```

lemma relprime-dvd-mult-iff:  $\text{gcd } k \ n = 1 \implies (k \ \text{dvd } m * n) = (k \ \text{dvd } m)$ 
by (auto intro: relprime-dvd-mult dvd-mult2)

```

```

lemma gcd-mult-cancel:  $\text{gcd } k \ n = 1 \implies \text{gcd } (k * m) \ n = \text{gcd } m \ n$ 
apply (rule dvd-antisym)
apply (rule gcd-greatest)
apply (rule-tac n = k in relprime-dvd-mult)
apply (simp add: gcd-assoc)
apply (simp add: gcd-commute)
apply (simp-all add: mult-commute)
done

```

Addition laws

```

lemma gcd-add1 [simp, algebra]:  $\text{gcd } (m + n) \ n = \text{gcd } m \ n$ 
by (cases n = 0) (auto simp add: gcd-non-0)

```

```

lemma gcd-add2 [simp, algebra]: gcd m (m + n) = gcd m n
proof -
  have gcd m (m + n) = gcd (m + n) m by (rule gcd-commute)
  also have ... = gcd (n + m) m by (simp add: add-commute)
  also have ... = gcd n m by simp
  also have ... = gcd m n by (rule gcd-commute)
  finally show ?thesis .
qed

```

```

lemma gcd-add2' [simp, algebra]: gcd m (n + m) = gcd m n
apply (subst add-commute)
apply (rule gcd-add2)
done

```

```

lemma gcd-add-mult[algebra]: gcd m (k * m + n) = gcd m n
by (induct k) (simp-all add: add-assoc)

```

```

lemma gcd-dvd-prod: gcd m n dvd m * n
using mult-dvd-mono [of 1] by auto

```

Division by gcd yields relatively primes.

```

lemma div-gcd-relprime:
  assumes nz: a ≠ 0 ∨ b ≠ 0
  shows gcd (a div gcd a b) (b div gcd a b) = 1
proof -
  let ?g = gcd a b
  let ?a' = a div ?g
  let ?b' = b div ?g
  let ?g' = gcd ?a' ?b'
  have dvdg: ?g dvd a ?g dvd b by simp-all
  have dvdg': ?g' dvd ?a' ?g' dvd ?b' by simp-all
  from dvdg dvdg' obtain ka kb ka' kb' where
    kab: a = ?g * ka b = ?g * kb ?a' = ?g' * ka' ?b' = ?g' * kb'
  unfolding dvd-def by blast
  then have ?g * ?a' = (?g * ?g') * ka' ?g * ?b' = (?g * ?g') * kb' by simp-all
  then have dvdgg': ?g * ?g' dvd a ?g * ?g' dvd b
    by (auto simp add: dvd-mult-div-cancel [OF dvdg(1)]
      dvd-mult-div-cancel [OF dvdg(2)] dvd-def)
  have ?g ≠ 0 using nz by (simp add: gcd-zero)
  then have gp: ?g > 0 by simp
  from gcd-greatest [OF dvdgg'] have ?g * ?g' dvd ?g .
  with dvd-mult-cancel1 [OF gp] show ?g' = 1 by simp
qed

```

```

lemma gcd-unique: d dvd a ∧ d dvd b ∧ (∀ e. e dvd a ∧ e dvd b ⟶ e dvd d) ⟷
d = gcd a b
proof(auto)

```


assume $H: d \text{ dvd } a \wedge d \text{ dvd } b \vee e. e \text{ dvd } a \wedge e \text{ dvd } b \longrightarrow e \text{ dvd } d$
from $H(3)[\text{rule-format}] \text{ gcd-dvd1[of } a \text{ } b] \text{ gcd-dvd2[of } a \text{ } b]$
have $th: \text{gcd } a \text{ } b \text{ dvd } d$ **by** *blast*
from $\text{dvd-antisym}[OF \text{ th gcd-greatest}[OF H(1,2)]]$ **show** $d = \text{gcd } a \text{ } b$ **by** *blast*
qed

lemma gcd-eq: assumes $H: \forall d. d \text{ dvd } x \wedge d \text{ dvd } y \longleftrightarrow d \text{ dvd } u \wedge d \text{ dvd } v$
shows $\text{gcd } x \text{ } y = \text{gcd } u \text{ } v$
proof–
from H **have** $\forall d. d \text{ dvd } x \wedge d \text{ dvd } y \longleftrightarrow d \text{ dvd } \text{gcd } u \text{ } v$ **by** *simp*
with $\text{gcd-unique}[of \text{ gcd } u \text{ } v \text{ } x \text{ } y]$ **show** *?thesis* **by** *auto*
qed

lemma ind-euclid:
assumes $c: \forall a \text{ } b. P (a::\text{nat}) \text{ } b \longleftrightarrow P \text{ } b \text{ } a$ **and** $z: \forall a. P \text{ } a \text{ } 0$
and $\text{add}: \forall a \text{ } b. P \text{ } a \text{ } b \longrightarrow P \text{ } a \text{ } (a + b)$
shows $P \text{ } a \text{ } b$
proof(*induct a + b arbitrary: a b rule: less-induct*)
case less
have $a = b \vee a < b \vee b < a$ **by** *arith*
moreover {**assume** $eq: a = b$
from $\text{add}[\text{rule-format}, OF \text{ z}[\text{rule-format}, of \text{ a}]]$ **have** $P \text{ } a \text{ } b$ **using** eq
by *simp*}
moreover
{**assume** $lt: a < b$
hence $a + b - a < a + b \vee a = 0$ **by** *arith*
moreover
{**assume** $a = 0$ **with** $z \text{ } c$ **have** $P \text{ } a \text{ } b$ **by** *blast* }
moreover
{**assume** $a + b - a < a + b$
also have $th0: a + b - a = a + (b - a)$ **using** lt **by** *arith*
finally have $a + (b - a) < a + b$.
then have $P \text{ } a \text{ } (a + (b - a))$ **by** ($\text{rule add}[\text{rule-format}, OF \text{ less}]$)
then have $P \text{ } a \text{ } b$ **by** ($\text{simp add: th0[symmetric]}$)}
ultimately have $P \text{ } a \text{ } b$ **by** *blast*}
moreover
{**assume** $lt: a > b$
hence $b + a - b < a + b \vee b = 0$ **by** *arith*
moreover
{**assume** $b = 0$ **with** $z \text{ } c$ **have** $P \text{ } a \text{ } b$ **by** *blast* }
moreover
{**assume** $b + a - b < a + b$
also have $th0: b + a - b = b + (a - b)$ **using** lt **by** *arith*
finally have $b + (a - b) < a + b$.
then have $P \text{ } b \text{ } (b + (a - b))$ **by** ($\text{rule add}[\text{rule-format}, OF \text{ less}]$)
then have $P \text{ } b \text{ } a$ **by** ($\text{simp add: th0[symmetric]}$)
hence $P \text{ } a \text{ } b$ **using** c **by** *blast* }
ultimately have $P \text{ } a \text{ } b$ **by** *blast*}
ultimately show $P \text{ } a \text{ } b$ **by** *blast*

qed

lemma bezout-lemma:

assumes $ex: \exists (d::nat) \ x \ y. \ d \ \text{dvd} \ a \wedge d \ \text{dvd} \ b \wedge (a * x = b * y + d \vee b * x = a * y + d)$
shows $\exists d \ x \ y. \ d \ \text{dvd} \ a \wedge d \ \text{dvd} \ a + b \wedge (a * x = (a + b) * y + d \vee (a + b) * x = a * y + d)$
using ex
apply $clarsimp$
apply $(rule-tac \ x=d \ \text{in} \ exI, \ simp \ add: \ dvd-add)$
apply $(case-tac \ a * x = b * y + d, \ simp-all)$
apply $(rule-tac \ x=x + y \ \text{in} \ exI)$
apply $(rule-tac \ x=y \ \text{in} \ exI)$
apply $algebra$
apply $(rule-tac \ x=x \ \text{in} \ exI)$
apply $(rule-tac \ x=x + y \ \text{in} \ exI)$
apply $algebra$
done

lemma bezout-add: $\exists (d::nat) \ x \ y. \ d \ \text{dvd} \ a \wedge d \ \text{dvd} \ b \wedge (a * x = b * y + d \vee b * x = a * y + d)$
apply $(induct \ a \ b \ \text{rule:} \ ind-euclid)$
apply $blast$
apply $clarify$
apply $(rule-tac \ x=a \ \text{in} \ exI, \ simp \ add: \ dvd-add)$
apply $clarsimp$
apply $(rule-tac \ x=d \ \text{in} \ exI)$
apply $(case-tac \ a * x = b * y + d, \ simp-all \ add: \ dvd-add)$
apply $(rule-tac \ x=x+y \ \text{in} \ exI)$
apply $(rule-tac \ x=y \ \text{in} \ exI)$
apply $algebra$
apply $(rule-tac \ x=x \ \text{in} \ exI)$
apply $(rule-tac \ x=x+y \ \text{in} \ exI)$
apply $algebra$
done

lemma bezout: $\exists (d::nat) \ x \ y. \ d \ \text{dvd} \ a \wedge d \ \text{dvd} \ b \wedge (a * x - b * y = d \vee b * x - a * y = d)$
using $bezout-add[of \ a \ b]$
apply $clarsimp$
apply $(rule-tac \ x=d \ \text{in} \ exI, \ simp)$
apply $(rule-tac \ x=x \ \text{in} \ exI)$
apply $(rule-tac \ x=y \ \text{in} \ exI)$
apply $auto$
done

We can get a stronger version with a nonzeroness assumption.

lemma divides-le: $m \ \text{dvd} \ n ==> m \leq n \vee n = (0::nat)$ **by** $(auto \ simp \ add: \ dvd-def)$

```

lemma bezout-add-strong: assumes nz:  $a \neq (0::nat)$ 
  shows  $\exists d \ x \ y. d \text{ dvd } a \wedge d \text{ dvd } b \wedge a * x = b * y + d$ 
proof -
  from nz have ap:  $a > 0$  by simp
  from bezout-add[of a b]
  have  $(\exists d \ x \ y. d \text{ dvd } a \wedge d \text{ dvd } b \wedge a * x = b * y + d) \vee (\exists d \ x \ y. d \text{ dvd } a \wedge d \text{ dvd } b \wedge b * x = a * y + d)$  by blast
  moreover
  {fix d x y assume H:  $d \text{ dvd } a \wedge d \text{ dvd } b \wedge a * x = b * y + d$ 
   from H have ?thesis by blast }
  moreover
  {fix d x y assume H:  $d \text{ dvd } a \wedge d \text{ dvd } b \wedge b * x = a * y + d$ 
   {assume b0:  $b = 0$  with H have ?thesis by simp}
   moreover
   {assume b:  $b \neq 0$  hence bp:  $b > 0$  by simp
    from divides-le[OF H(2)] b have  $d < b \vee d = b$  using le-less by blast
    moreover
    {assume db:  $d = b$ 
     from prems have ?thesis apply simp
     apply (rule exI[where x = b], simp)
     apply (rule exI[where x = b])
     by (rule exI[where x = a - 1], simp add: diff-mult-distrib2)}
    moreover
    {assume db:  $d < b$ 
     {assume x=0 hence ?thesis using prems by simp }
     moreover
     {assume x0:  $x \neq 0$  hence xp:  $x > 0$  by simp

      from db have  $d \leq b - 1$  by simp
      hence  $d * b \leq b * (b - 1)$  by simp
      with xp mult-mono[of 1 x d * b b * (b - 1)]
      have dble:  $d * b \leq x * b * (b - 1)$  using bp by simp
      from H (3) have  $a * ((b - 1) * y) + d * (b - 1 + 1) = d + x * b * (b - 1)$  by algebra
      hence  $a * ((b - 1) * y) = d + x * b * (b - 1) - d * b$  using bp by simp
      hence  $a * ((b - 1) * y) = d + (x * b * (b - 1) - d * b)$ 
       by (simp only: diff-add-assoc[OF dble, of d, symmetric])
      hence  $a * ((b - 1) * y) = b * (x * (b - 1) - d) + d$ 
       by (simp only: diff-mult-distrib2 add-commute mult-ac)
      hence ?thesis using H(1,2)
       apply -
       apply (rule exI[where x = d], simp)
       apply (rule exI[where x = (b - 1) * y])
       by (rule exI[where x = x * (b - 1) - d], simp)}
     ultimately have ?thesis by blast}
    ultimately have ?thesis by blast}
  ultimately have ?thesis by blast
  ultimately show ?thesis by blast

```

qed

lemma bezout-gcd: $\exists x y. a * x - b * y = \text{gcd } a \ b \vee b * x - a * y = \text{gcd } a \ b$
proof–
 let $?g = \text{gcd } a \ b$
 from $\text{bezout}[of \ a \ b]$ obtain $d \ x \ y$ where $d: d \ \text{dvd} \ a \ d \ \text{dvd} \ b \ a * x - b * y = d \vee b * x - a * y = d$ **by** *blast*
 from $d(1,2)$ have $d \ \text{dvd} \ ?g$ **by** *simp*
 then obtain k where $k: ?g = d * k$ **unfolding** *dvd-def* **by** *blast*
 from $d(3)$ have $(a * x - b * y) * k = d * k \vee (b * x - a * y) * k = d * k$ **by** *blast*
 hence $a * x * k - b * y * k = d * k \vee b * x * k - a * y * k = d * k$
 by (*algebra add: diff-mult-distrib*)
 hence $a * (x * k) - b * (y * k) = ?g \vee b * (x * k) - a * (y * k) = ?g$
 by (*simp add: k mult-assoc*)
 thus *?thesis* **by** *blast*
 qed

lemma bezout-gcd-strong: **assumes** $a: a \neq 0$
shows $\exists x y. a * x = b * y + \text{gcd } a \ b$
proof–
 let $?g = \text{gcd } a \ b$
 from $\text{bezout-add-strong}[OF \ a, \ of \ b]$
 obtain $d \ x \ y$ where $d: d \ \text{dvd} \ a \ d \ \text{dvd} \ b \ a * x = b * y + d$ **by** *blast*
 from $d(1,2)$ have $d \ \text{dvd} \ ?g$ **by** *simp*
 then obtain k where $k: ?g = d * k$ **unfolding** *dvd-def* **by** *blast*
 from $d(3)$ have $a * x * k = (b * y + d) * k$ **by** *algebra*
 hence $a * (x * k) = b * (y * k) + ?g$ **by** (*algebra add: k*)
 thus *?thesis* **by** *blast*
 qed

lemma gcd-mult-distrib: $\text{gcd}(a * c) (b * c) = c * \text{gcd } a \ b$
by (*simp add: gcd-mult-distrib2 mult-commute*)

lemma gcd-bezout: $(\exists x y. a * x - b * y = d \vee b * x - a * y = d) \longleftrightarrow \text{gcd } a \ b \ \text{dvd} \ d$
(is ?lhs \longleftrightarrow ?rhs)
proof–
 let $?g = \text{gcd } a \ b$
 {**assume** $H: ?rhs$ **then obtain** k where $k: d = ?g * k$ **unfolding** *dvd-def* **by** *blast*
 from $\text{bezout-gcd}[of \ a \ b]$ obtain $x \ y$ where $xy: a * x - b * y = ?g \vee b * x - a * y = ?g$
 by *blast*
 hence $(a * x - b * y) * k = ?g * k \vee (b * x - a * y) * k = ?g * k$ **by** *auto*
 hence $a * x * k - b * y * k = ?g * k \vee b * x * k - a * y * k = ?g * k$
 by (*simp only: diff-mult-distrib*)
 hence $a * (x * k) - b * (y * k) = d \vee b * (x * k) - a * (y * k) = d$
 by (*simp add: k[symmetric] mult-assoc*)

```

    hence ?lhs by blast}
  moreover
  {fix x y assume H: a * x - b * y = d ∨ b * x - a * y = d
   have dv: ?g dvd a*x ?g dvd b * y ?g dvd b*x ?g dvd a * y
   using dvd-mult2[OF gcd-dvd1[of a b]] dvd-mult2[OF gcd-dvd2[of a b]] by
simp-all
   from dvd-diff-nat[OF dv(1,2)] dvd-diff-nat[OF dv(3,4)] H
   have ?rhs by auto}
  ultimately show ?thesis by blast
qed

```

```

lemma gcd-bezout-sum: assumes H: a * x + b * y = d shows gcd a b dvd d
proof-
  let ?g = gcd a b
  have dv: ?g dvd a*x ?g dvd b * y
  using dvd-mult2[OF gcd-dvd1[of a b]] dvd-mult2[OF gcd-dvd2[of a b]] by
simp-all
  from dvd-add[OF dv] H
  show ?thesis by auto
qed

```

```

lemma gcd-mult': gcd b (a * b) = b
by (simp add: gcd-mult mult-commute[of a b])

```

```

lemma gcd-add: gcd(a + b) b = gcd a b
gcd(b + a) b = gcd a b gcd a (a + b) = gcd a b gcd a (b + a) = gcd a b
apply (simp-all add: gcd-add1)
by (simp add: gcd-commute gcd-add1)

```

```

lemma gcd-sub: b ≤ a ==> gcd(a - b) b = gcd a b a ≤ b ==> gcd a (b -
a) = gcd a b
proof-
  {fix a b assume H: b ≤ (a::nat)
   hence th: a - b + b = a by arith
   from gcd-add(1)[of a - b b] th have gcd(a - b) b = gcd a b by simp}
  note th = this
  {
    assume ab: b ≤ a
    from th[OF ab] show gcd (a - b) b = gcd a b by blast
  }
next
  assume ab: a ≤ b
  from th[OF ab] show gcd a (b - a) = gcd a b
  by (simp add: gcd-commute)}
qed

```

1.4 LCM defined by GCD

definition

$lcm :: nat \Rightarrow nat \Rightarrow nat$

where

lcm-def: $\text{lcm } m \ n = m * n \ \text{div} \ \text{gcd } m \ n$

lemma *prod-gcd-lcm*:

$m * n = \text{gcd } m \ n * \text{lcm } m \ n$

unfolding *lcm-def* by (*simp add: dvd-mult-div-cancel [OF gcd-dvd-prod]*)

lemma *lcm-0* [*simp*]: $\text{lcm } m \ 0 = 0$

unfolding *lcm-def* by *simp*

lemma *lcm-1* [*simp*]: $\text{lcm } m \ 1 = m$

unfolding *lcm-def* by *simp*

lemma *lcm-0-left* [*simp*]: $\text{lcm } 0 \ n = 0$

unfolding *lcm-def* by *simp*

lemma *lcm-1-left* [*simp*]: $\text{lcm } 1 \ m = m$

unfolding *lcm-def* by *simp*

lemma *dvd-pos*:

fixes $n \ m :: \text{nat}$

assumes $n > 0$ and $m \ \text{dvd} \ n$

shows $m > 0$

using *assms* by (*cases m*) *auto*

lemma *lcm-least*:

assumes $m \ \text{dvd} \ k$ and $n \ \text{dvd} \ k$

shows $\text{lcm } m \ n \ \text{dvd} \ k$

proof (*cases k*)

case 0 then show ?thesis by *auto*

next

case (*Suc -*) then have *pos-k*: $k > 0$ by *auto*

from *assms dvd-pos [OF this]* have *pos-mn*: $m > 0 \ n > 0$ by *auto*

with *gcd-zero [of m n]* have *pos-gcd*: $\text{gcd } m \ n > 0$ by *simp*

from *assms* obtain p where *k-m*: $k = m * p$ using *dvd-def* by *blast*

from *assms* obtain q where *k-n*: $k = n * q$ using *dvd-def* by *blast*

from *pos-k k-m* have *pos-p*: $p > 0$ by *auto*

from *pos-k k-n* have *pos-q*: $q > 0$ by *auto*

have $k * k * \text{gcd } q \ p = k * \text{gcd } (k * q) \ (k * p)$

by (*simp add: mult-ac gcd-mult-distrib2*)

also have $\dots = k * \text{gcd } (m * p * q) \ (n * q * p)$

by (*simp add: k-m [symmetric] k-n [symmetric]*)

also have $\dots = k * p * q * \text{gcd } m \ n$

by (*simp add: mult-ac gcd-mult-distrib2*)

finally have $(m * p) * (n * q) * \text{gcd } q \ p = k * p * q * \text{gcd } m \ n$

by (*simp only: k-m [symmetric] k-n [symmetric]*)

then have $p * q * m * n * \text{gcd } q \ p = p * q * k * \text{gcd } m \ n$

by (*simp add: mult-ac*)

with *pos-p pos-q* have $m * n * \text{gcd } q \ p = k * \text{gcd } m \ n$

```

    by simp
  with prod-gcd-lcm [of m n]
  have lcm m n * gcd q p * gcd m n = k * gcd m n
    by (simp add: mult-ac)
  with pos-gcd have lcm m n * gcd q p = k by simp
  then show ?thesis using dvd-def by auto
qed

lemma lcm-dvd1 [iff]:
  m dvd lcm m n
proof (cases m)
  case 0 then show ?thesis by simp
next
  case (Suc -)
  then have mpos: m > 0 by simp
  show ?thesis
  proof (cases n)
    case 0 then show ?thesis by simp
  next
    case (Suc -)
    then have npos: n > 0 by simp
    have gcd m n dvd n by simp
    then obtain k where n = gcd m n * k using dvd-def by auto
    then have m * n div gcd m n = m * (gcd m n * k) div gcd m n by (simp add:
mult-ac)
    also have ... = m * k using mpos npos gcd-zero by simp
    finally show ?thesis by (simp add: lcm-def)
  qed
qed

lemma lcm-dvd2 [iff]:
  n dvd lcm m n
proof (cases n)
  case 0 then show ?thesis by simp
next
  case (Suc -)
  then have npos: n > 0 by simp
  show ?thesis
  proof (cases m)
    case 0 then show ?thesis by simp
  next
    case (Suc -)
    then have mpos: m > 0 by simp
    have gcd m n dvd m by simp
    then obtain k where m = gcd m n * k using dvd-def by auto
    then have m * n div gcd m n = (gcd m n * k) * n div gcd m n by (simp add:
mult-ac)
    also have ... = n * k using mpos npos gcd-zero by simp
    finally show ?thesis by (simp add: lcm-def)
  qed

```

qed
qed

lemma *gcd-add1-eq*: $\text{gcd } (m + k) \ k = \text{gcd } (m + k) \ m$
by (*simp add: gcd-commute*)

lemma *gcd-diff2*: $m \leq n \implies \text{gcd } n \ (n - m) = \text{gcd } n \ m$
apply (*subgoal-tac n = m + (n - m)*)
apply (*erule ssubst, rule gcd-add1-eq, simp*)
done

1.5 GCD and LCM on integers

definition

zgcd :: $\text{int} \Rightarrow \text{int} \Rightarrow \text{int}$ **where**
zgcd *i j* = $\text{int } (\text{gcd } (\text{nat } (\text{abs } i)) \ (\text{nat } (\text{abs } j)))$

lemma *zgcd-zdvd1* [*iff, simp, algebra*]: *zgcd* *i j* *dvd* *i*
by (*simp add: zgcd-def int-dvd-iff*)

lemma *zgcd-zdvd2* [*iff, simp, algebra*]: *zgcd* *i j* *dvd* *j*
by (*simp add: zgcd-def int-dvd-iff*)

lemma *zgcd-pos*: *zgcd* *i j* ≥ 0
by (*simp add: zgcd-def*)

lemma *zgcd0* [*simp, algebra*]: (*zgcd* *i j* = 0) = (*i* = 0 \wedge *j* = 0)
by (*simp add: zgcd-def gcd-zero*)

lemma *zgcd-commute*: *zgcd* *i j* = *zgcd* *j i*
unfolding *zgcd-def* **by** (*simp add: gcd-commute*)

lemma *zgcd-zminus* [*simp, algebra*]: *zgcd* (− *i*) *j* = *zgcd* *i j*
unfolding *zgcd-def* **by** *simp*

lemma *zgcd-zminus2* [*simp, algebra*]: *zgcd* *i* (− *j*) = *zgcd* *i j*
unfolding *zgcd-def* **by** *simp*

lemma *zrelprime-dvd-mult*: *zgcd* *i j* = 1 $\implies i \text{ dvd } k * j \implies i \text{ dvd } k$
unfolding *zgcd-def*

proof −

assume $\text{int } (\text{gcd } (\text{nat } |i|) \ (\text{nat } |j|)) = 1 \ i \text{ dvd } k * j$

then have $g: \text{gcd } (\text{nat } |i|) \ (\text{nat } |j|) = 1$ **by** *simp*

from $\langle i \text{ dvd } k * j \rangle$ **obtain** *h* **where** $h: k * j = i * h$ **unfolding** *dvd-def* **by** *blast*

have $th: \text{nat } |i| \text{ dvd } \text{nat } |k| * \text{nat } |j|$

unfolding *dvd-def*

by (*rule-tac x = nat |h| in exI, simp add: h nat-abs-mult-distrib [symmetric]*)

from *relprime-dvd-mult* [*OF g th*] **obtain** *h'* **where** $h': \text{nat } |k| = \text{nat } |i| * h'$


```

    unfolding dvd-def by blast
  from h' have int (nat |k|) = int (nat |i| * h') by simp
  then have |k| = |i| * int h' by (simp add: int-mult)
  then show ?thesis
    apply (subst abs-dvd-iff [symmetric])
    apply (subst dvd-abs-iff [symmetric])
    apply (unfold dvd-def)
    apply (rule-tac x = int h' in exI, simp)
  done
qed

```

lemma int-nat-abs: int (nat (abs x)) = abs x by arith

```

lemma zgcd-greatest:
  assumes k dvd m and k dvd n
  shows k dvd zgcd m n
proof -
  let ?k' = nat |k|
  let ?m' = nat |m|
  let ?n' = nat |n|
  from ⟨k dvd m⟩ and ⟨k dvd n⟩ have dvd': ?k' dvd ?m' ?k' dvd ?n'
    unfolding zdvd-int by (simp-all only: int-nat-abs abs-dvd-iff dvd-abs-iff)
  from gcd-greatest [OF dvd'] have int (nat |k|) dvd zgcd m n
    unfolding zgcd-def by (simp only: zdvd-int)
  then have |k| dvd zgcd m n by (simp only: int-nat-abs)
  then show k dvd zgcd m n by simp
qed

```

```

lemma div-zgcd-relprime:
  assumes nz: a ≠ 0 ∨ b ≠ 0
  shows zgcd (a div (zgcd a b)) (b div (zgcd a b)) = 1
proof -
  from nz have nz': nat |a| ≠ 0 ∨ nat |b| ≠ 0 by arith
  let ?g = zgcd a b
  let ?a' = a div ?g
  let ?b' = b div ?g
  let ?g' = zgcd ?a' ?b'
  have dvdg: ?g dvd a ?g dvd b by (simp-all add: zgcd-zdvd1 zgcd-zdvd2)
  have dvdg': ?g' dvd ?a' ?g' dvd ?b' by (simp-all add: zgcd-zdvd1 zgcd-zdvd2)
  from dvdg dvdg' obtain ka kb ka' kb' where
    kab: a = ?g*ka b = ?g*kb ?a' = ?g'*ka' ?b' = ?g'*kb'
    unfolding dvd-def by blast
  then have ?g*?a' = (?g*?g')*ka'?g*?b' = (?g*?g')*kb' by simp-all
  then have dvdgg': ?g*?g' dvd a ?g*?g' dvd b
    by (auto simp add: zdvd-mult-div-cancel [OF dvdg(1)]
      zdvd-mult-div-cancel [OF dvdg(2)] dvd-def)
  have ?g ≠ 0 using nz by simp
  then have gp: ?g ≠ 0 using zgcd-pos[where i=a and j=b] by arith
  from zgcd-greatest [OF dvdgg'] have ?g*?g' dvd ?g .

```

with *zdvd-mult-cancel1* [*OF gp*] **have** $|?g'| = 1$ **by** *simp*
with *zgcd-pos* **show** $?g' = 1$ **by** *simp*
qed

lemma *zgcd-0* [*simp, algebra*]: $zgcd\ m\ 0 = abs\ m$
by (*simp add: zgcd-def abs-if*)

lemma *zgcd-0-left* [*simp, algebra*]: $zgcd\ 0\ m = abs\ m$
by (*simp add: zgcd-def abs-if*)

lemma *zgcd-non-0*: $0 < n \implies zgcd\ m\ n = zgcd\ n\ (m\ mod\ n)$
apply (*frule-tac b = n and a = m in pos-mod-sign*)
apply (*simp del: pos-mod-sign add: zgcd-def abs-if nat-mod-distrib*)
apply (*auto simp add: gcd-non-0 nat-mod-distrib [symmetric] zmod-zminus1-eq-if*)
apply (*frule-tac a = m in pos-mod-bound*)
apply (*simp del: pos-mod-bound add: nat-diff-distrib gcd-diff2 nat-le-eq-zle*)
done

lemma *zgcd-eq*: $zgcd\ m\ n = zgcd\ n\ (m\ mod\ n)$
apply (*case-tac n = 0, simp add: DIVISION-BY-ZERO*)
apply (*auto simp add: linorder-neq-iff zgcd-non-0*)
apply (*cut-tac m = -m and n = -n in zgcd-non-0, auto*)
done

lemma *zgcd-1* [*simp, algebra*]: $zgcd\ m\ 1 = 1$
by (*simp add: zgcd-def abs-if*)

lemma *zgcd-0-1-iff* [*simp, algebra*]: $zgcd\ 0\ m = 1 \longleftrightarrow |m| = 1$
by (*simp add: zgcd-def abs-if*)

lemma *zgcd-greatest-iff* [*algebra*]: $k\ dvd\ zgcd\ m\ n = (k\ dvd\ m \wedge k\ dvd\ n)$
by (*simp add: zgcd-def abs-if int-dvd-iff dvd-int-iff nat-dvd-iff*)

lemma *zgcd-1-left* [*simp, algebra*]: $zgcd\ 1\ m = 1$
by (*simp add: zgcd-def gcd-1-left*)

lemma *zgcd-assoc*: $zgcd\ (zgcd\ k\ m)\ n = zgcd\ k\ (zgcd\ m\ n)$
by (*simp add: zgcd-def gcd-assoc*)

lemma *zgcd-left-commute*: $zgcd\ k\ (zgcd\ m\ n) = zgcd\ m\ (zgcd\ k\ n)$
apply (*rule zgcd-commute [THEN trans]*)
apply (*rule zgcd-assoc [THEN trans]*)
apply (*rule zgcd-commute [THEN arg-cong]*)
done

lemmas *zgcd-ac* = *zgcd-assoc zgcd-commute zgcd-left-commute*
— addition is an AC-operator

lemma *zgcd-zmult-distrib2*: $0 \leq k \implies k * zgcd\ m\ n = zgcd\ (k * m)\ (k * n)$

by (*simp* *del*: *minus-mult-right* [*symmetric*]
add: *minus-mult-right nat-mult-distrib zgcd-def abs-if*
mult-less-0-iff gcd-mult-distrib2 [*symmetric*] *zmult-int* [*symmetric*])

lemma *zgcd-zmult-distrib2-abs*: $zgcd (k * m) (k * n) = abs\ k * zgcd\ m\ n$
by (*simp* *add*: *abs-if zgcd-zmult-distrib2*)

lemma *zgcd-self* [*simp*]: $0 \leq m \implies zgcd\ m\ m = m$
by (*cut-tac* $k = m$ **and** $m = 1$ **and** $n = 1$ **in** *zgcd-zmult-distrib2*, *simp-all*)

lemma *zgcd-zmult-eq-self* [*simp*]: $0 \leq k \implies zgcd\ k\ (k * n) = k$
by (*cut-tac* $k = k$ **and** $m = 1$ **and** $n = n$ **in** *zgcd-zmult-distrib2*, *simp-all*)

lemma *zgcd-zmult-eq-self2* [*simp*]: $0 \leq k \implies zgcd\ (k * n)\ k = k$
by (*cut-tac* $k = k$ **and** $m = n$ **and** $n = 1$ **in** *zgcd-zmult-distrib2*, *simp-all*)

definition *z lcm* $i\ j = int\ (lcm(nat(abs\ i))\ (nat(abs\ j)))$

lemma *dvd-z lcm-self1* [*simp*, *algebra*]: $i\ dvd\ z lcm\ i\ j$
by(*simp* *add*:*z lcm-def dvd-int-iff*)

lemma *dvd-z lcm-self2* [*simp*, *algebra*]: $j\ dvd\ z lcm\ i\ j$
by(*simp* *add*:*z lcm-def dvd-int-iff*)

lemma *dvd-imp-dvd-z lcm1*:
assumes $k\ dvd\ i$ **shows** $k\ dvd\ (z lcm\ i\ j)$
proof –
have $nat(abs\ k)\ dvd\ nat(abs\ i)$ **using** $\langle k\ dvd\ i \rangle$
by(*simp* *add*:*int-dvd-iff* [*symmetric*] *dvd-int-iff* [*symmetric*])
thus ?thesis **by**(*simp* *add*:*z lcm-def dvd-int-iff*)(*blast intro: dvd-trans*)
qed

lemma *dvd-imp-dvd-z lcm2*:
assumes $k\ dvd\ j$ **shows** $k\ dvd\ (z lcm\ i\ j)$
proof –
have $nat(abs\ k)\ dvd\ nat(abs\ j)$ **using** $\langle k\ dvd\ j \rangle$
by(*simp* *add*:*int-dvd-iff* [*symmetric*] *dvd-int-iff* [*symmetric*])
thus ?thesis **by**(*simp* *add*:*z lcm-def dvd-int-iff*)(*blast intro: dvd-trans*)
qed

lemma *zdvd-self-abs1*: $(d::int)\ dvd\ (abs\ d)$
by (*case-tac* $d < 0$, *simp-all*)

lemma *zdvd-self-abs2*: $(abs\ (d::int))\ dvd\ d$
by (*case-tac* $d < 0$, *simp-all*)

```

lemma lcm-pos:
  assumes mpos:  $m > 0$ 
  and npos:  $n > 0$ 
  shows  $\text{lcm } m \ n > 0$ 
proof(rule ccontr, simp add: lcm-def gcd-zero)
assume h:  $m * n \text{ div gcd } m \ n = 0$ 
from mpos npos have  $\text{gcd } m \ n \neq 0$  using gcd-zero by simp
hence gcdp:  $\text{gcd } m \ n > 0$  by simp
with h
have  $m * n < \text{gcd } m \ n$ 
  by (cases  $m * n < \text{gcd } m \ n$ ) (auto simp add: div-if[OF gcdp, where  $m = m * n$ ])
moreover
have  $\text{gcd } m \ n \text{ dvd } m$  by simp
with mpos dvd-imp-le have  $t1: \text{gcd } m \ n \leq m$  by simp
with npos have  $t1: \text{gcd } m \ n * n \leq m * n$  by simp
have  $\text{gcd } m \ n \leq \text{gcd } m \ n * n$  using npos by simp
with t1 have  $\text{gcd } m \ n \leq m * n$  by arith
ultimately show False by simp
qed

lemma zlcm-pos:
  assumes anz:  $a \neq 0$ 
  and bnz:  $b \neq 0$ 
  shows  $0 < \text{zlcm } a \ b$ 
proof-
  let ?na = nat (abs a)
  let ?nb = nat (abs b)
  have nap:  $?na > 0$  using anz by simp
  have nbp:  $?nb > 0$  using bnz by simp
  have  $0 < \text{lcm } ?na \ ?nb$  by (rule lcm-pos[OF nap nbp])
  thus ?thesis by (simp add: zlcm-def)
qed

lemma zgcd-code [code]:
   $\text{zgcd } k \ l = \text{if } l = 0 \text{ then } k \text{ else } \text{zgcd } l \ (|k| \bmod |l|)$ 
  by (simp add: zgcd-def gcd.simps [of nat |k| nat-mod-distrib])

end

```

2 Primality on nat

```

theory Primes
imports Complex-Main Legacy-GCD
begin

definition

```

```
coprime :: nat => nat => bool where
coprime m n <=> gcd m n = 1
```

definition

```
prime :: nat => bool where
[code del]: prime p <=> (1 < p & (\ m. m dvd p --> m = 1 \vee m = p))
```

```
lemma two-is-prime: prime 2
  apply (auto simp add: prime-def)
  apply (case-tac m)
  apply (auto dest!: dvd-imp-le)
done
```

```
lemma prime-imp-relprime: prime p ==> \neg p dvd n ==> gcd p n = 1
  apply (auto simp add: prime-def)
  apply (metis One-nat-def gcd-dvd1 gcd-dvd2)
done
```

This theorem leads immediately to a proof of the uniqueness of factorization.
If p divides a product of primes then it is one of those primes.

```
lemma prime-dvd-mult: prime p ==> p dvd m * n ==> p dvd m \vee p dvd n
  by (blast intro: relprime-dvd-mult prime-imp-relprime)
```

```
lemma prime-dvd-square: prime p ==> p dvd m ^ Suc (Suc 0) ==> p dvd m
  by (auto dest: prime-dvd-mult)
```

```
lemma prime-dvd-power-two: prime p ==> p dvd m^2 ==> p dvd m
  by (rule prime-dvd-square) (simp-all add: power2-eq-square)
```

```
lemma exp-eq-1: (x::nat) ^ n = 1 <=> x = 1 \vee n = 0
  by (induct n, auto)
```

```
lemma exp-mono-lt: (x::nat) ^ (Suc n) < y ^ (Suc n) <=> x < y
  by (metis linorder-not-less not-less0 power-le-imp-le-base power-less-imp-less-base)
```

```
lemma exp-mono-le: (x::nat) ^ (Suc n) \le y ^ (Suc n) <=> x \le y
  by (simp only: linorder-not-less[symmetric] exp-mono-lt)
```

```
lemma exp-mono-eq: (x::nat) ^ Suc n = y ^ Suc n <=> x = y
  using power-inject-base[of x n y] by auto
```

```
lemma even-square: assumes e: even (n::nat) shows \exists x. n ^ 2 = 4*x
proof-
  from e have 2 dvd n by presburger
  then obtain k where k: n = 2*k using dvd-def by auto
  hence n^2 = 4* (k^2) by (simp add: power2-eq-square)
```

thus ?thesis by blast
qed

lemma odd-square: assumes $e: \text{odd } (n::\text{nat})$ shows $\exists x. n^2 = 4*x + 1$
proof –
 from e have $np: n > 0$ by presburger
 from e have $2 \text{ dvd } (n - 1)$ by presburger
 then obtain k where $n - 1 = 2*k$ using dvd-def by auto
 hence $k: n = 2*k + 1$ using e by presburger
 hence $n^2 = 4 * (k^2 + k) + 1$ by algebra
 thus ?thesis by blast
qed

lemma diff-square: $(x::\text{nat})^2 - y^2 = (x+y)*(x - y)$
proof –
 have $x \leq y \vee y \leq x$ by (rule nat-le-linear)
 moreover
 {assume $le: x \leq y$
 hence $x^2 \leq y^2$ by (simp only: numeral-2-eq-2 exp-mono-le Let-def)
 with le have ?thesis by simp }
 moreover
 {assume $le: y \leq x$
 hence $le2: y^2 \leq x^2$ by (simp only: numeral-2-eq-2 exp-mono-le Let-def)
 from le have $\exists z. y + z = x$ by presburger
 then obtain z where $z: x = y + z$ by blast
 from $le2$ have $\exists z. x^2 = y^2 + z$ by presburger
 then obtain $z2$ where $z2: x^2 = y^2 + z2$ by blast
 from $z \ z2$ have ?thesis apply simp by algebra }
 ultimately show ?thesis by blast
qed

Elementary theory of divisibility

lemma divides-ge: $(a::\text{nat}) \text{ dvd } b \implies b = 0 \vee a \leq b$ unfolding dvd-def by auto
lemma divides-antisym: $(x::\text{nat}) \text{ dvd } y \wedge y \text{ dvd } x \longleftrightarrow x = y$
 using dvd-antisym[of $x \ y$] by auto

lemma divides-add-revr: assumes $da: (d::\text{nat}) \text{ dvd } a$ and $dab: d \text{ dvd } (a + b)$
 shows $d \text{ dvd } b$
proof –
 from da obtain k where $k:a = d*k$ by (auto simp add: dvd-def)
 from dab obtain k' where $k': a + b = d*k'$ by (auto simp add: dvd-def)
 from $k \ k'$ have $b = d * (k' - k)$ by (simp add: diff-mult-distrib2)
 thus ?thesis unfolding dvd-def by blast
qed

declare nat-mult-dvd-cancel-disj[presburger]
lemma nat-mult-dvd-cancel-disj'[presburger]:
 $(m::\text{nat})*k \text{ dvd } n*k \longleftrightarrow k = 0 \vee m \text{ dvd } n$ unfolding mult-commute[of $m \ k$]
 mult-commute[of $n \ k$] by presburger

```

lemma divides-mul-l: (a::nat) dvd b ==> (c * a) dvd (c * b)
  by presburger

lemma divides-mul-r: (a::nat) dvd b ==> (a * c) dvd (b * c) by presburger
lemma divides-cases: (n::nat) dvd m ==> m = 0 ∨ m = n ∨ 2 * n <= m
  by (auto simp add: dvd-def)

lemma divides-div-not: (x::nat) = (q * n) + r ==> 0 < r ==> r < n ==> ~ (n
dvd x)
proof(auto simp add: dvd-def)
  fix k assume H: 0 < r r < n q * n + r = n * k
  from H(3) have r: r = n * (k - q) by(simp add: diff-mult-distrib2 mult-commute)
  {assume k - q = 0 with r H(1) have False by simp}
  moreover
  {assume k - q ≠ 0 with r have r ≥ n by auto
    with H(2) have False by simp}
  ultimately show False by blast
qed
lemma divides-exp: (x::nat) dvd y ==> x ^ n dvd y ^ n
  by (auto simp add: power-mult-distrib dvd-def)

lemma divides-exp2: n ≠ 0 ==> (x::nat) ^ n dvd y ==> x dvd y
  by (induct n ,auto simp add: dvd-def)

fun fact :: nat => nat where
  fact 0 = 1
| fact (Suc n) = Suc n * fact n

lemma fact-lt: 0 < fact n by(induct n, simp-all)
lemma fact-le: fact n ≥ 1 using fact-lt[of n] by simp
lemma fact-mono: assumes le: m ≤ n shows fact m ≤ fact n
proof-
  from le have ∃ i. n = m + i by presburger
  then obtain i where i: n = m + i by blast
  have fact m ≤ fact (m + i)
  proof(induct m)
    case 0 thus ?case using fact-le[of i] by simp
  next
    case (Suc m)
    have fact (Suc m) = Suc m * fact m by simp
    have th1: Suc m ≤ Suc (m + i) by simp
    from mult-le-mono[of Suc m Suc (m + i) fact m fact (m + i), OF th1 Suc.hyps]
    show ?case by simp
  qed
  thus ?thesis using i by simp
qed

lemma divides-fact: 1 <= p ==> p <= n ==> p dvd fact n

```

```

proof(induct n arbitrary: p)
  case 0 thus ?case by simp
next
  case (Suc n p)
  from Suc.prems have  $p = \text{Suc } n \vee p \leq n$  by presburger
  moreover
  {assume  $p = \text{Suc } n$  hence ?case by (simp only: fact.simps dvd-triv-left)}
  moreover
  {assume  $p \leq n$ 
    with Suc.prems(1) Suc.hyps have th:  $p \text{ dvd fact } n$  by simp
    from dvd-mult[OF th] have ?case by (simp only: fact.simps) }
  ultimately show ?case by blast
qed

declare dvd-triv-left[presburger]
declare dvd-triv-right[presburger]
lemma divides-rexp:
   $x \text{ dvd } y \implies (x :: \text{nat}) \text{ dvd } (y^\wedge (\text{Suc } n))$  by (simp add: dvd-mult2[of x y])

Coprimality

lemma coprime:  $\text{coprime } a \ b \longleftrightarrow (\forall d. d \text{ dvd } a \wedge d \text{ dvd } b \longleftrightarrow d = 1)$ 
using gcd-unique[of 1 a b, simplified] by (auto simp add: coprime-def)
lemma coprime-commute:  $\text{coprime } a \ b \longleftrightarrow \text{coprime } b \ a$  by (simp add: coprime-def gcd-commute)

lemma coprime-bezout:  $\text{coprime } a \ b \longleftrightarrow (\exists x \ y. a * x - b * y = 1 \vee b * x - a * y = 1)$ 
using coprime-def gcd-bezout by auto

lemma coprime-divprod:  $d \text{ dvd } a * b \implies \text{coprime } d \ a \implies d \text{ dvd } b$ 
using relprime-dvd-mult-iff[of d a b] by (auto simp add: coprime-def mult-commute)

lemma coprime-1[simp]:  $\text{coprime } a \ 1$  by (simp add: coprime-def)
lemma coprime-1'[simp]:  $\text{coprime } 1 \ a$  by (simp add: coprime-def)
lemma coprime-Suc0[simp]:  $\text{coprime } a \ (\text{Suc } 0)$  by (simp add: coprime-def)
lemma coprime-Suc0'[simp]:  $\text{coprime } (\text{Suc } 0) \ a$  by (simp add: coprime-def)

lemma gcd-coprime:
  assumes z:  $\text{gcd } a \ b \neq 0$  and a:  $a = a' * \text{gcd } a \ b$  and b:  $b = b' * \text{gcd } a \ b$ 
  shows  $\text{coprime } a' \ b'$ 
proof–
  let ?g =  $\text{gcd } a \ b$ 
  {assume bz:  $a = 0$  from b bz z a have ?thesis by (simp add: gcd-zero coprime-def)}
  moreover
  {assume az:  $a \neq 0$ 
    from z have z':  $?g > 0$  by simp
    from bezout-gcd-strong[OF az, of b]
    obtain x y where xy:  $a * x = b * y + ?g$  by blast
    from xy a b have  $?g * a' * x = ?g * (b' * y + 1)$  by (simp add: algebra-simps)
  }

```



```

    hence ?g * (a'*x) = ?g * (b'*y + 1) by (simp add: mult-assoc)
    hence a'*x = (b'*y + 1)
      by (simp only: nat-mult-eq-cancel1[OF z'])
    hence a'*x - b'*y = 1 by simp
    with coprime-bezout[of a' b'] have ?thesis by auto}
  ultimately show ?thesis by blast
qed
lemma coprime-0: coprime d 0  $\longleftrightarrow$  d = 1 by (simp add: coprime-def)
lemma coprime-mul: assumes da: coprime d a and db: coprime d b
  shows coprime d (a * b)
proof -
  from da have th: gcd a d = 1 by (simp add: coprime-def gcd-commute)
  from gcd-mult-cancel[of a d b, OF th] db[unfolded coprime-def] have gcd d (a*b)
= 1
  by (simp add: gcd-commute)
  thus ?thesis unfolding coprime-def .
qed
lemma coprime-lmul2: assumes dab: coprime d (a * b) shows coprime d b
using prems unfolding coprime-bezout
apply clarsimp
apply (case-tac d * x - a * b * y = Suc 0 , simp-all)
apply (rule-tac x=x in exI)
apply (rule-tac x=a*y in exI)
apply (simp add: mult-ac)
apply (rule-tac x=a*x in exI)
apply (rule-tac x=y in exI)
apply (simp add: mult-ac)
done

lemma coprime-rmul2: coprime d (a * b)  $\implies$  coprime d a
unfolding coprime-bezout
apply clarsimp
apply (case-tac d * x - a * b * y = Suc 0 , simp-all)
apply (rule-tac x=x in exI)
apply (rule-tac x=b*y in exI)
apply (simp add: mult-ac)
apply (rule-tac x=b*x in exI)
apply (rule-tac x=y in exI)
apply (simp add: mult-ac)
done
lemma coprime-mul-eq: coprime d (a * b)  $\longleftrightarrow$  coprime d a  $\wedge$  coprime d b
  using coprime-rmul2[of d a b] coprime-lmul2[of d a b] coprime-mul[of d a b]
  by blast

lemma gcd-coprime-exists:
  assumes nz: gcd a b  $\neq$  0
  shows  $\exists a' b'. a = a' * \text{gcd } a \ b \wedge b = b' * \text{gcd } a \ b \wedge \text{coprime } a' \ b'$ 
proof -
  let ?g = gcd a b

```

from $\text{gcd-dvd1}[of\ a\ b]\ \text{gcd-dvd2}[of\ a\ b]$
 obtain $a'\ b'$ where $a = ?g * a'\ b = ?g * b'$ unfolding dvd-def by blast
 hence ab' : $a = a' * ?g\ b = b' * ?g$ by algebra+
 from $ab'\ \text{gcd-coprime}[OF\ nz\ ab']$ show $?thesis$ by blast
 qed

lemma coprime-exp : $\text{coprime}\ d\ a ==> \text{coprime}\ d\ (a^n)$
 by ($\text{induct}\ n, \text{simp-all}\ \text{add: coprime-mul}$)

lemma coprime-exp-imp : $\text{coprime}\ a\ b ==> \text{coprime}\ (a^n)\ (b^n)$
 by ($\text{induct}\ n, \text{simp-all}\ \text{add: coprime-mul-eq coprime-commute coprime-exp}$)
 lemma $\text{coprime-refl}[simp]$: $\text{coprime}\ n\ n \longleftrightarrow n = 1$ by ($\text{simp}\ \text{add: coprime-def}$)
 lemma $\text{coprime-plus1}[simp]$: $\text{coprime}\ (n + 1)\ n$
 apply ($\text{simp}\ \text{add: coprime-bezout}$)
 apply ($\text{rule}\ \text{exI}[\text{where}\ x=1]$)
 apply ($\text{rule}\ \text{exI}[\text{where}\ x=1]$)
 apply simp
 done

lemma coprime-minus1 : $n \neq 0 ==> \text{coprime}\ (n - 1)\ n$
 using $\text{coprime-plus1}[of\ n - 1]\ \text{coprime-commute}[of\ n - 1\ n]$ by auto

lemma bezout-gcd-pow : $\exists x\ y. a^n * x - b^n * y = \text{gcd}\ a\ b^n \vee b^n * x - a^n * y = \text{gcd}\ a\ b^n$

proof—

let $?g = \text{gcd}\ a\ b$
 {assume z : $?g = 0$ hence $?thesis$
 apply ($\text{cases}\ n, \text{simp}$)
 apply arith
 apply ($\text{simp}\ \text{only: } z\ \text{power-0-Suc}$)
 apply ($\text{rule}\ \text{exI}[\text{where}\ x=0]$)
 apply ($\text{rule}\ \text{exI}[\text{where}\ x=0]$)
 by simp }
 moreover
 {assume z : $?g \neq 0$
 from $\text{gcd-dvd1}[of\ a\ b]\ \text{gcd-dvd2}[of\ a\ b]$ obtain $a'\ b'$ where
 ab' : $a = a' * ?g\ b = b' * ?g$ unfolding dvd-def by ($\text{auto}\ \text{simp}\ \text{add: mult-ac}$)
 hence ab'' : $?g * a' = a\ ?g * b' = b$ by algebra+
 from $\text{coprime-exp-imp}[OF\ \text{gcd-coprime}[OF\ z\ ab']]$, unfolded coprime-bezout , of
 $n]$
 obtain $x\ y$ where $a'^n * x - b'^n * y = 1 \vee b'^n * x - a'^n * y = 1$ by
 blast
 hence $?g^n * (a'^n * x - b'^n * y) = ?g^n \vee ?g^n * (b'^n * x - a'^n * y) =$
 $?g^n$
 using z by auto
 then have $a^n * x - b^n * y = ?g^n \vee b^n * x - a^n * y = ?g^n$
 using $z\ ab''$ by ($\text{simp}\ \text{only: power-mult-distrib[symmetric]}$)
 $\text{diff-mult-distrib2 mult-assoc[symmetric]}$
 hence $?thesis$ by blast }
 ultimately show $?thesis$ by blast

qed

lemma gcd-exp: $\text{gcd } (a^n) (b^n) = \text{gcd } a \ b^n$

proof–

let $?g = \text{gcd } (a^n) (b^n)$
 let $?gn = \text{gcd } a \ b^n$
 {fix e assume $H: e \text{ dvd } a^n \ e \text{ dvd } b^n$
 from bezout-gcd-pow[of $a \ n \ b$] obtain $x \ y$
 where $xy: a^n * x - b^n * y = ?gn \vee b^n * x - a^n * y = ?gn$ by
 blast
 from dvd-diff-nat [OF dvd-mult2[OF $H(1)$, of x] dvd-mult2[OF $H(2)$, of y]]
 dvd-diff-nat [OF dvd-mult2[OF $H(2)$, of x] dvd-mult2[OF $H(1)$, of y]] xy
 have $e \text{ dvd } ?gn$ by (cases $a^n * x - b^n * y = \text{gcd } a \ b^n$, simp-all)}
 hence th: $\forall e. e \text{ dvd } a^n \wedge e \text{ dvd } b^n \longrightarrow e \text{ dvd } ?gn$ by blast
 from divides-exp[OF gcd-dvd1[of $a \ b$], of n] divides-exp[OF gcd-dvd2[of $a \ b$], of
 n] th
 gcd-unique have $?gn = ?g$ by blast thus ?thesis by simp
 qed

lemma coprime-exp2: $\text{coprime } (a^{Suc \ n}) (b^{Suc \ n}) \longleftrightarrow \text{coprime } a \ b$
 by (simp only: coprime-def gcd-exp exp-eq-1) simp

lemma division-decomp: assumes $dc: (a::nat) \text{ dvd } b * c$

shows $\exists b' \ c'. a = b' * c' \wedge b' \text{ dvd } b \wedge c' \text{ dvd } c$

proof–

let $?g = \text{gcd } a \ b$
 {assume $?g = 0$ with dc have ?thesis apply (simp add: gcd-zero)
 apply (rule exI[where $x=0$])
 by (rule exI[where $x=c$], simp)}
 moreover
 {assume $z: ?g \neq 0$
 from gcd-coprime-exists[OF z]
 obtain $a' \ b'$ where $ab': a = a' * ?g \ b = b' * ?g$ coprime $a' \ b'$ by blast
 from gcd-dvd2[of $a \ b$] have thb: $?g \text{ dvd } b$.
 from $ab'(1)$ have $a' \text{ dvd } a$ unfolding dvd-def by blast
 with dc have th0: $a' \text{ dvd } b * c$ using dvd-trans[of $a' \ a \ b * c$] by simp
 from $dc \ ab'(1,2)$ have $a' * ?g \text{ dvd } (b' * ?g) * c$ by auto
 hence $?g * a' \text{ dvd } ?g * (b' * c)$ by (simp add: mult-assoc)
 with z have th-1: $a' \text{ dvd } b' * c$ by simp
 from coprime-divprod[OF th-1 $ab'(\mathfrak{z})$] have thc: $a' \text{ dvd } c$.
 from ab' have $a = ?g * a'$ by algebra
 with thb thc have ?thesis by blast }
 ultimately show ?thesis by blast
 qed

lemma nat-power-eq-0-iff: $(m::nat) ^ n = 0 \longleftrightarrow n \neq 0 \wedge m = 0$ by (induct n , auto)

lemma divides-rev: assumes $ab: (a::nat) ^ n \text{ dvd } b ^ n$ and $n:n \neq 0$ shows $a \text{ dvd } b$

b
proof–
 let $?g = \gcd a b$
 from n obtain m where $m: n = \text{Suc } m$ by (cases n , simp-all)
 {assume $?g = 0$ with $ab\ n$ have $?thesis$ by (simp add: gcd-zero)}
 moreover
 {assume $z: ?g \neq 0$
 hence $zn: ?g \wedge n \neq 0$ using n by (simp add: neq0-conv)
 from gcd-coprime-exists[OF z]
 obtain $a' b'$ where $ab': a = a' * ?g\ b = b' * ?g$ coprime $a' b'$ by blast
 from ab have $(a' * ?g) \wedge n \text{ dvd } (b' * ?g) \wedge n$ by (simp add: ab'(1,2)[symmetric])
 hence $?g \wedge n * a' \wedge n \text{ dvd } ?g \wedge n * b' \wedge n$ by (simp only: power-mult-distrib mult-commute)
 with $zn\ z\ n$ have $th0: a' \wedge n \text{ dvd } b' \wedge n$ by (auto simp add: nat-power-eq-0-iff)
 have $a' \text{ dvd } a' \wedge n$ by (simp add: m)
 with $th0$ have $a' \text{ dvd } b' \wedge n$ using dvd-trans[of $a' a' \wedge n\ b' \wedge n$] by simp
 hence $th1: a' \text{ dvd } b' \wedge m * b'$ by (simp add: m mult-commute)
 from coprime-divprod[OF $th1$ coprime-exp[OF $ab'(3)$, of m]]
 have $a' \text{ dvd } b'$.
 hence $a' * ?g \text{ dvd } b' * ?g$ by simp
 with $ab'(1,2)$ have $?thesis$ by simp }
 ultimately show $?thesis$ by blast
qed

lemma divides-mul: assumes $mr: m \text{ dvd } r$ and $nr: n \text{ dvd } r$ and $mn: \text{coprime } m\ n$
 shows $m * n \text{ dvd } r$
proof–
 from $mr\ nr$ obtain $m' n'$ where $m': r = m * m'$ and $n': r = n * n'$
 unfolding dvd-def by blast
 from $mr\ n'$ have $m \text{ dvd } n' * n$ by (simp add: mult-commute)
 hence $m \text{ dvd } n'$ using relprime-dvd-mult-iff[OF mn [unfolded coprime-def]] by simp
 then obtain k where $k: n' = m * k$ unfolding dvd-def by blast
 from $n'\ k$ show $?thesis$ unfolding dvd-def by auto
qed

A binary form of the Chinese Remainder Theorem.

lemma chinese-remainder: assumes $ab: \text{coprime } a\ b$ and $a:a \neq 0$ and $b:b \neq 0$
 shows $\exists x\ q1\ q2. x = u + q1 * a \wedge x = v + q2 * b$
proof–
 from bezout-add-strong[OF a , of b] bezout-add-strong[OF b , of a]
 obtain $d1\ x1\ y1\ d2\ x2\ y2$ where $dxy1: d1 \text{ dvd } a\ d1 \text{ dvd } b\ a * x1 = b * y1 + d1$
 and $dxy2: d2 \text{ dvd } b\ d2 \text{ dvd } a\ b * x2 = a * y2 + d2$ by blast
 from gcd-unique[of $1\ a\ b$, simplified ab [unfolded coprime-def], simplified]
 $dxy1(1,2)\ dxy2(1,2)$ have $d12: d1 = 1\ d2 = 1$ by auto
 let $?x = v * a * x1 + u * b * x2$
 let $?q1 = v * x1 + u * y2$
 let $?q2 = v * y1 + u * x2$
 from $dxy2(3)$ [simplified $d12$] $dxy1(3)$ [simplified $d12$]

have $?x = u + ?q1 * a$ **?** $x = v + ?q2 * b$ **by** *algebra+*
thus *?thesis* **by** *blast*
qed

Primality

A few useful theorems about primes

lemma *prime-0[simp]: $\sim \text{prime } 0$* **by** (*simp add: prime-def*)
lemma *prime-1[simp]: $\sim \text{prime } 1$* **by** (*simp add: prime-def*)
lemma *prime-Suc0[simp]: $\sim \text{prime } (\text{Suc } 0)$* **by** (*simp add: prime-def*)

lemma *prime-ge-2: $\text{prime } p \implies p \geq 2$* **by** (*simp add: prime-def*)
lemma *prime-factor: assumes $n: n \neq 1$ shows $\exists p. \text{prime } p \wedge p \text{ dvd } n$*
using *n*

proof(*induct n rule: nat-less-induct*)
fix *n*
assume *H: $\forall m < n. m \neq 1 \longrightarrow (\exists p. \text{prime } p \wedge p \text{ dvd } m) n \neq 1$*
let *?ths = $\exists p. \text{prime } p \wedge p \text{ dvd } n$*
{assume *n=0* **hence** *?ths* **using** *two-is-prime* **by** *auto***}**
moreover
{assume *nz: $n \neq 0$*
{assume *prime n* **hence** *?ths* **by** $-(\text{rule } \text{exI}[\text{where } x=n], \text{simp}))$ **}**
moreover
{assume *n: $\neg \text{prime } n$*
with *nz H(2)*
obtain *k* **where** *k:k dvd n* *k $\neq 1$ k $\neq n$* **by** (*auto simp add: prime-def*)
from *dvd-imp-le[OF k(1)] nz k(3)* **have** *kn: $k < n$* **by** *simp*
from *H(1)[rule-format, OF kn k(2)]* **obtain** *p* **where** *p: prime p* *p dvd k* **by**
blast
from *dvd-trans[OF p(2) k(1)] p(1)* **have** *?ths* **by** *blast***}**
ultimately have *?ths* **by** *blast***}**
ultimately show *?ths* **by** *blast*
qed

lemma *prime-factor-lt: assumes $p: \text{prime } p$ and $n: n \neq 0$ and $npm: n = p * m$*
shows *m < n*

proof–
{assume *m=0* **with** *n* **have** *?thesis* **by** *simp***}**
moreover
{assume *m: m $\neq 0$*
from *npm* **have** *mn: m dvd n* **unfolding** *dvd-def* **by** *auto*
from *npm m* **have** *n $\neq m$* **using** *p* **by** *auto*
with *dvd-imp-le[OF mn] n* **have** *?thesis* **by** *simp***}**
ultimately show *?thesis* **by** *blast*
qed

lemma *euclid-bound: $\exists p. \text{prime } p \wedge n < p \wedge p \leq \text{Suc } (\text{fact } n)$*

proof–
have *f1: $\text{fact } n + 1 \neq 1$* **using** *fact-le[of n]* **by** *arith*
from *prime-factor[OF f1]* **obtain** *p* **where** *p: prime p* *p dvd fact n + 1* **by** *blast*

```

from dvd-imp-le[OF p(2)] have pfn:  $p \leq \text{fact } n + 1$  by simp
{assume np:  $p \leq n$ 
  from p(1) have p1:  $p \geq 1$  by (cases p, simp-all)
  from divides-fact[OF p1 np] have pfn':  $p \text{ dvd fact } n$  .
  from divides-add-revr[OF pfn' p(2)] p(1) have False by simp}
hence  $n < p$  by arith
with p(1) pfn show ?thesis by auto
qed

lemma euclid:  $\exists p. \text{prime } p \wedge p > n$  using euclid-bound by auto

lemma primes-infinite:  $\neg (\text{finite } \{p. \text{prime } p\})$ 
apply (simp add: finite-nat-set-iff-bounded-le)
apply (metis euclid linorder-not-le)
done

lemma coprime-prime: assumes ab: coprime a b
shows  $\sim(\text{prime } p \wedge p \text{ dvd } a \wedge p \text{ dvd } b)$ 
proof
  assume prime p  $\wedge p \text{ dvd } a \wedge p \text{ dvd } b$ 
  thus False using ab gcd-greatest[of p a b] by (simp add: coprime-def)
qed
lemma coprime-prime-eq:  $\text{coprime } a \ b \longleftrightarrow (\forall p. \sim(\text{prime } p \wedge p \text{ dvd } a \wedge p \text{ dvd } b))$ 

  (is ?lhs = ?rhs)
proof–
  {assume ?lhs with coprime-prime have ?rhs by blast}
  moreover
  {assume r: ?rhs and c:  $\neg ?lhs$ 
    then obtain g where  $g \neq 1 \wedge g \text{ dvd } a \wedge g \text{ dvd } b$  unfolding coprime-def by blast
    from prime-factor[OF g(1)] obtain p where  $p: \text{prime } p \wedge p \text{ dvd } g$  by blast
    from dvd-trans [OF p(2) g(2)] dvd-trans [OF p(2) g(3)]
    have  $p \text{ dvd } a \wedge p \text{ dvd } b$  . with p(1) r have False by blast}
  ultimately show ?thesis by blast
qed

lemma prime-coprime: assumes p: prime p
shows  $n = 1 \vee p \text{ dvd } n \vee \text{coprime } p \ n$ 
using p prime-imp-relprime[of p n] by (auto simp add: coprime-def)

lemma prime-coprime-strong:  $\text{prime } p \implies p \text{ dvd } n \vee \text{coprime } p \ n$ 
using prime-coprime[of p n] by auto

declare coprime-0[simp]

lemma coprime-0'[simp]:  $\text{coprime } 0 \ d \longleftrightarrow d = 1$  by (simp add: coprime-commute[of 0 d])
lemma coprime-bezout-strong: assumes ab: coprime a b and b:  $b \neq 1$ 
shows  $\exists x \ y. a * x = b * y + 1$ 

```

```

proof–
  from ab b have az: a ≠ 0 by – (rule ccontr, auto)
  from bezout-gcd-strong[OF az, of b] ab[unfolded coprime-def]
  show ?thesis by auto
qed

lemma bezout-prime: assumes p: prime p and pa: ¬ p dvd a
  shows  $\exists x y. a * x = p * y + 1$ 
proof–
  from p have p1: p ≠ 1 using prime-1 by blast
  from prime-coprime[OF p, of a] p1 pa have ap: coprime a p
    by (auto simp add: coprime-commute)
  from coprime-bezout-strong[OF ap p1] show ?thesis .
qed

lemma prime-divprod: assumes p: prime p and pab: p dvd a * b
  shows  $p \text{ dvd } a \vee p \text{ dvd } b$ 
proof–
  {assume a=1 hence ?thesis using pab by simp }
  moreover
  {assume p dvd a hence ?thesis by blast}
  moreover
  {assume pa: coprime p a from coprime-divprod[OF pab pa] have ?thesis .. }
  ultimately show ?thesis using prime-coprime[OF p, of a] by blast
qed

lemma prime-divprod-eq: assumes p: prime p
  shows  $p \text{ dvd } a * b \iff p \text{ dvd } a \vee p \text{ dvd } b$ 
using p prime-divprod dvd-mult dvd-mult2 by auto

lemma prime-divexp: assumes p: prime p and px: p dvd x ^ n
  shows  $p \text{ dvd } x$ 
using px
proof(induct n)
  case 0 thus ?case by simp
next
  case (Suc n)
  hence th: p dvd x * x ^ n by simp
  {assume H: p dvd x ^ n
    from Suc.hyps[OF H] have ?case .}
  with prime-divprod[OF p th] show ?case by blast
qed

lemma prime-divexp-n:  $\text{prime } p \implies p \text{ dvd } x ^ n \implies p ^ n \text{ dvd } x ^ n$ 
  using prime-divexp[of p x n] divides-exp[of p x n] by blast

lemma coprime-prime-dvd-ex: assumes xy: ¬ coprime x y
  shows  $\exists p. \text{prime } p \wedge p \text{ dvd } x \wedge p \text{ dvd } y$ 
proof–
  from xy[unfolded coprime-def] obtain g where  $g: g \neq 1 \wedge g \text{ dvd } x \wedge g \text{ dvd } y$ 

```

```

    by blast
  from prime-factor[OF g(1)] obtain p where p: prime p p dvd g by blast
  from g(2,3) dvd-trans[OF p(2)] p(1) show ?thesis by auto
qed
lemma coprime-sos: assumes xy: coprime x y
  shows coprime (x * y) (x^2 + y^2)
proof-
  {assume c: ¬ coprime (x * y) (x^2 + y^2)
   from coprime-prime-dvd-ex[OF c] obtain p
     where p: prime p p dvd x*y p dvd x^2 + y^2 by blast
   {assume px: p dvd x
    from dvd-mult[OF px, of x] p(3)
      obtain r s where x * x = p * r and x^2 + y^2 = p * s
        by (auto elim!: dvdE)
      then have y^2 = p * (s - r)
        by (auto simp add: power2-eq-square diff-mult-distrib2)
      then have p dvd y^2 ..
    with prime-divexp[OF p(1), of y 2] have py: p dvd y .
    from p(1) px py xy[unfolded coprime, rule-format, of p] prime-1
      have False by simp }
    moreover
    {assume py: p dvd y
     from dvd-mult[OF py, of y] p(3)
       obtain r s where y * y = p * r and x^2 + y^2 = p * s
         by (auto elim!: dvdE)
       then have x^2 = p * (s - r)
         by (auto simp add: power2-eq-square diff-mult-distrib2)
       then have p dvd x^2 ..
     with prime-divexp[OF p(1), of x 2] have px: p dvd x .
     from p(1) px py xy[unfolded coprime, rule-format, of p] prime-1
       have False by simp }
    ultimately have False using prime-divprod[OF p(1,2)] by blast}
  thus ?thesis by blast
qed

lemma distinct-prime-coprime: prime p ⇒ prime q ⇒ p ≠ q ⇒ coprime p q
  unfolding prime-def coprime-prime-eq by blast

lemma prime-coprime-lt: assumes p: prime p and x: 0 < x and xp: x < p
  shows coprime x p
proof-
  {assume c: ¬ coprime x p
   then obtain g where g: g ≠ 1 g dvd x g dvd p unfolding coprime-def by
    blast
   from dvd-imp-le[OF g(2)] x xp have gp: g < p by arith
   from g(2) x have g ≠ 0 by - (rule ccontr, simp)
   with g gp p[unfolded prime-def] have False by blast}
  thus ?thesis by blast
qed

```


lemma *even-dvd*[simp]: $\text{even } (n::\text{nat}) \longleftrightarrow 2 \text{ dvd } n$ **by** *presburger*
lemma *prime-odd*: $\text{prime } p \implies p = 2 \vee \text{odd } p$ **unfolding** *prime-def* **by** *auto*

One property of coprimality is easier to prove via prime factors.

lemma *prime-divprod-pow*:
assumes *p*: $\text{prime } p$ **and** *ab*: $\text{coprime } a \ b$ **and** *pab*: $p^n \text{ dvd } a * b$
shows $p^n \text{ dvd } a \vee p^n \text{ dvd } b$
proof–
 {**assume** $n = 0 \vee a = 1 \vee b = 1$ **with** *pab* **have** *?thesis*
 apply (*cases n=0, simp-all*)
 apply (*cases a=1, simp-all*) **done**}
moreover
 {**assume** $n: n \neq 0$ **and** $a: a \neq 1$ **and** $b: b \neq 1$
 then obtain *m* **where** $m: n = \text{Suc } m$ **by** (*cases n, auto*)
 from *divides-exp2*[*OF n pab*] **have** *pab'*: $p \text{ dvd } a * b$.
 from *prime-divprod*[*OF p pab'*]
 have $p \text{ dvd } a \vee p \text{ dvd } b$.
 moreover
 {**assume** *pa*: $p \text{ dvd } a$
 have *pnba*: $p^n \text{ dvd } b * a$ **using** *pab* **by** (*simp add: mult-commute*)
 from *coprime-prime*[*OF ab, of p*] *p pa* **have** $\neg p \text{ dvd } b$ **by** *blast*
 with *prime-coprime*[*OF p, of b*] *b*
 have *cpb*: $\text{coprime } b \ p$ **using** *coprime-commute* **by** *blast*
 from *coprime-exp*[*OF cpb*] **have** *pnb*: $\text{coprime } (p^n) \ b$
 by (*simp add: coprime-commute*)
 from *coprime-divprod*[*OF pnba pnb*] **have** *?thesis* **by** *blast* }
 moreover
 {**assume** *pb*: $p \text{ dvd } b$
 have *pnba*: $p^n \text{ dvd } b * a$ **using** *pab* **by** (*simp add: mult-commute*)
 from *coprime-prime*[*OF ab, of p*] *p pb* **have** $\neg p \text{ dvd } a$ **by** *blast*
 with *prime-coprime*[*OF p, of a*] *a*
 have *cpb*: $\text{coprime } a \ p$ **using** *coprime-commute* **by** *blast*
 from *coprime-exp*[*OF cpb*] **have** *pnb*: $\text{coprime } (p^n) \ a$
 by (*simp add: coprime-commute*)
 from *coprime-divprod*[*OF pab pnb*] **have** *?thesis* **by** *blast* }
 ultimately have *?thesis* **by** *blast*}
ultimately show *?thesis* **by** *blast*

qed

lemma *nat-mult-eq-one*: $(n::\text{nat}) * m = 1 \longleftrightarrow n = 1 \wedge m = 1$ (**is** *?lhs* \longleftrightarrow *?rhs*)

proof

assume *H*: *?lhs*

hence $n \text{ dvd } 1 \ m \text{ dvd } 1$ **unfolding** *dvd-def* **by** (*auto simp add: mult-commute*)

thus *?rhs* **by** *auto*

next

assume *?rhs* **then show** *?lhs* **by** *auto*

qed

```

lemma power-Suc0[simp]: Suc 0 ^ n = Suc 0
  unfolding One-nat-def[symmetric] power-one ..
lemma coprime-pow: assumes ab: coprime a b and abcn: a * b = c ^ n
  shows  $\exists r s. a = r^n \wedge b = s^n$ 
  using ab abcn
proof(induct c arbitrary: a b rule: nat-less-induct)
  fix c a b
  assume H:  $\forall m < c. \forall a b. \text{coprime } a b \longrightarrow a * b = m^n \longrightarrow (\exists r s. a = r^n \wedge b = s^n)$ 
  let ?ths =  $\exists r s. a = r^n \wedge b = s^n$ 
  {assume n: n = 0
    with H(3) power-one have a*b = 1 by simp
    hence a = 1  $\wedge$  b = 1 by simp
    hence ?ths
    apply -
    apply (rule exI[where x=1])
    apply (rule exI[where x=1])
    using power-one[of n]
    by simp}
  moreover
  {assume n: n  $\neq$  0 then obtain m where m: n = Suc m by (cases n, auto)
    {assume c: c = 0
      with H(3) m H(2) have ?ths apply simp
      apply (cases a=0, simp-all)
      apply (rule exI[where x=0], simp)
      apply (rule exI[where x=0], simp)
      done}
    moreover
    {assume c=1 with H(3) power-one have a*b = 1 by simp
      hence a = 1  $\wedge$  b = 1 by simp
      hence ?ths
      apply -
      apply (rule exI[where x=1])
      apply (rule exI[where x=1])
      using power-one[of n]
      by simp}
    moreover
    {assume c: c  $\neq$  1 c  $\neq$  0
      from prime-factor[OF c(1)] obtain p where p: prime p p dvd c by blast
      from prime-divprod-pow[OF p(1) H(2), unfolded H(3), OF divides-exp[OF p(2), of n]]
      have pnab:  $p^n \text{dvd } a \vee p^n \text{dvd } b$  .
      from p(2) obtain l where l: c = p * l unfolding dvd-def by blast
      have pn0:  $p^n \neq 0$  using n prime-ge-2 [OF p(1)] by (simp add: neq0-conv)
      {assume pa:  $p^n \text{dvd } a$ 
        then obtain k where k: a = p^n * k unfolding dvd-def by blast
        from l have l dvd c by auto
        with dvd-imp-le[of l c] c have l  $\leq$  c by auto

```

```

    moreover {assume  $l = c$  with  $l\ c$  have  $p = 1$  by simp with  $p$  have False
  by simp}
    ultimately have  $lc: l < c$  by arith
    from coprime-lmul2 [OF  $H(2)[unfolding\ k\ coprime-commute[of\ p^n * k\ b]]]$ 
    have  $kb: coprime\ k\ b$  by (simp add: coprime-commute)
    from  $H(3)\ l\ k\ pn0$  have  $kbln: k * b = l^n$ 
      by (auto simp add: power-mult-distrib)
    from  $H(1)[rule-format, OF\ lc\ kb\ kbln]$ 
    obtain  $r\ s$  where  $rs: k = r^n\ b = s^n$  by blast
    from  $k\ rs(1)$  have  $a = (p*r)^n$  by (simp add: power-mult-distrib)
    with  $rs(2)$  have ?ths by blast }
  moreover
  {assume  $pb: p^n\ dvd\ b$ 
   then obtain  $k$  where  $k: b = p^n * k$  unfolding dvd-def by blast
   from  $l$  have  $l\ dvd\ c$  by auto
   with dvd-imp-le[of  $l\ c$ ]  $c$  have  $l \leq c$  by auto
   moreover {assume  $l = c$  with  $l\ c$  have  $p = 1$  by simp with  $p$  have False
  by simp}
   ultimately have  $lc: l < c$  by arith
   from coprime-lmul2 [OF  $H(2)[unfolding\ k\ coprime-commute[of\ p^n * k\ a]]]$ 
   have  $kb: coprime\ k\ a$  by (simp add: coprime-commute)
   from  $H(3)\ l\ k\ pn0\ n$  have  $kbln: k * a = l^n$ 
     by (simp add: power-mult-distrib mult-commute)
   from  $H(1)[rule-format, OF\ lc\ kb\ kbln]$ 
   obtain  $r\ s$  where  $rs: k = r^n\ a = s^n$  by blast
   from  $k\ rs(1)$  have  $b = (p*r)^n$  by (simp add: power-mult-distrib)
   with  $rs(2)$  have ?ths by blast }
  ultimately have ?ths using pnab by blast}
  ultimately have ?ths by blast}
  ultimately show ?ths by blast
qed

```

More useful lemmas.

lemma *prime-product*:

assumes *prime* $(p * q)$

shows $p = 1 \vee q = 1$

proof –

from *assms* have

$1 < p * q$ and $P: \bigwedge m. m\ dvd\ p * q \implies m = 1 \vee m = p * q$

unfolding *prime-def* by *auto*

from $\langle 1 < p * q \rangle$ have $p \neq 0$ by (*cases p*) *auto*

then have $Q: p = p * q \iff q = 1$ by *auto*

have $p\ dvd\ p * q$ by *simp*

then have $p = 1 \vee p = p * q$ by (*rule P*)

then show *?thesis* by (*simp add: Q*)

qed

lemma *prime-exp*: $prime\ (p^n) \iff prime\ p \wedge n = 1$

proof(*induct n*)

```

    case 0 thus ?case by simp
next
case (Suc n)
{assume p = 0 hence ?case by simp}
moreover
{assume p=1 hence ?case by simp}
moreover
{assume p: p ≠ 0 p≠1
  {assume pp: prime (p^Suc n)
    hence p = 1 ∨ p^n = 1 using prime-product[of p p^n] by simp
    with p have n: n = 0
      by (simp only: exp-eq-1) simp
    with pp have prime p ∧ Suc n = 1 by simp}
  moreover
  {assume n: prime p ∧ Suc n = 1 hence prime (p^Suc n) by simp}
  ultimately have ?case by blast}
ultimately show ?case by blast
qed

```

```

lemma prime-power-mult:
  assumes p: prime p and xy: x * y = p ^ k
  shows ∃ i j. x = p ^ i ∧ y = p ^ j
  using xy
proof(induct k arbitrary: x y)
  case 0 thus ?case apply simp by (rule exI[where x=0], simp)
next
case (Suc k x y)
  from Suc.prem1 have pxy: p dvd x*y by auto
  from prime-divprod[OF p pxy] have prxc: p dvd x ∨ p dvd y .
  from p have p0: p ≠ 0 by - (rule ccontr, simp)
  {assume px: p dvd x
    then obtain d where d: x = p*d unfolding dvd-def by blast
    from Suc.prem1 d have p*d*y = p^Suc k by simp
    hence th: d*y = p^k using p0 by simp
    from Suc.hyps[OF th] obtain i j where ij: d = p^i y = p^j by blast
    with d have x = p^Suc i by simp
    with ij(2) have ?case by blast}
  moreover
  {assume py: p dvd y
    then obtain d where d: y = p*d unfolding dvd-def by blast
    from Suc.prem1 d have p*d*x = p^Suc k by (simp add: mult-commute)
    hence th: d*x = p^k using p0 by simp
    from Suc.hyps[OF th] obtain i j where ij: d = p^i x = p^j by blast
    with d have y = p^Suc i by simp
    with ij(2) have ?case by blast}
  ultimately show ?case using prxc by blast
qed

```

```

lemma prime-power-exp: assumes p: prime p and n:n ≠ 0

```

```

and xn:  $x^n = p^k$  shows  $\exists i. x = p^i$ 
using n xn
proof(induct n arbitrary: k)
  case 0 thus ?case by simp
next
  case (Suc n k) hence th:  $x * x^n = p^k$  by simp
  {assume n = 0 with prems have ?case apply simp
   by (rule exI[where x=k],simp)}
  moreover
  {assume n:  $n \neq 0$ 
   from prime-power-mult[OF p th]
   obtain i j where  $ij: x = p^i x^n = p^j$  by blast
   from Suc.hyps[OF n ij(2)] have ?case .}
  ultimately show ?case by blast
qed

```

```

lemma divides-primelow: assumes p: prime p
  shows  $d \text{ dvd } p^k \iff (\exists i. i \leq k \wedge d = p^i)$ 
proof
  assume H:  $d \text{ dvd } p^k$  then obtain e where  $e: d * e = p^k$ 
    unfolding dvd-def apply (auto simp add: mult-commute) by blast
  from prime-power-mult[OF p e] obtain i j where  $ij: d = p^i e = p^j$  by blast
  from prime-ge-2[OF p] have p1:  $p > 1$  by arith
  from e ij have  $p^{i+j} = p^k$  by (simp add: power-add)
  hence  $i + j = k$  using power-inject-exp[of p i+j k, OF p1] by simp
  hence  $i \leq k$  by arith
  with ij(1) show  $\exists i \leq k. d = p^i$  by blast
next
  {fix i assume H:  $i \leq k \wedge d = p^i$ 
   hence  $\exists j. k = i + j$  by arith
   then obtain j where  $j: k = i + j$  by blast
   hence  $p^k = p^{j+d}$  using H(2) by (simp add: power-add)
   hence  $d \text{ dvd } p^k$  unfolding dvd-def by auto}
  thus  $\exists i \leq k. d = p^i \implies d \text{ dvd } p^k$  by blast
qed

```

```

lemma coprime-divisors:  $d \text{ dvd } a \implies e \text{ dvd } b \implies \text{coprime } a \ b \implies \text{coprime } d \ e$ 
  by (auto simp add: dvd-def coprime)

```

```

lemma mult-inj-if-coprime-nat:
   $\text{inj-on } f \ A \implies \text{inj-on } g \ B \implies \text{ALL } a:A. \text{ALL } b:B. \text{coprime } (f \ a) \ (g \ b)$ 
 $\implies \text{inj-on } (\% (a,b). f \ a * g \ b :: \text{nat}) \ (A \times B)$ 
apply (auto simp add: inj-on-def)
apply (metis coprime-def dvd-triv-left gcd-proj2-if-dvd-nat gcd-semilattice-nat.inf-commute
  relprime-dvd-mult)
apply (metis coprime-commute coprime-divprod dvd.neq-le-trans dvd-triv-right)
done

```

```

declare power-Suc0[simp del]

```

```

declare even-dvd[simp del]

end

```

3 The Fibonacci function

```

theory Fib
imports Primes
begin

```

Fibonacci numbers: proofs of laws taken from: R. L. Graham, D. E. Knuth, O. Patashnik. Concrete Mathematics. (Addison-Wesley, 1989)

```

fun fib :: nat ⇒ nat
where
  fib 0 = 0
|   fib (Suc 0) = 1
| fib-2: fib (Suc (Suc n)) = fib n + fib (Suc n)

```

The difficulty in these proofs is to ensure that the induction hypotheses are applied before the definition of *fib*. Towards this end, the *fib* equations are not declared to the Simplifier and are applied very selectively at first.

We disable *fib.fib-2fib-2* for simplification ...

```

declare fib-2 [simp del]

```

...then prove a version that has a more restrictive pattern.

```

lemma fib-Suc3: fib (Suc (Suc (Suc n))) = fib (Suc n) + fib (Suc (Suc n))
  by (rule fib-2)

```

Concrete Mathematics, page 280

```

lemma fib-add: fib (Suc (n + k)) = fib (Suc k) * fib (Suc n) + fib k * fib n
proof (induct n rule: fib.induct)
  case 1 show ?case by simp
next
  case 2 show ?case by (simp add: fib-2)
next
  case 3 thus ?case by (simp add: fib-2 add-mult-distrib2)
qed

lemma fib-Suc-neq-0: fib (Suc n) ≠ 0
  apply (induct n rule: fib.induct)
  apply (simp-all add: fib-2)
  done

lemma fib-Suc-gr-0: 0 < fib (Suc n)

```

```

by (insert fib-Suc-neq-0 [of n], simp)

lemma fib-gr-0: 0 < n ==> 0 < fib n
by (case-tac n, auto simp add: fib-Suc-gr-0)

```

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) =
    (if n mod 2 = 0 then int (fib (Suc n) * fib (Suc n)) - 1
     else int (fib (Suc n) * fib (Suc n)) + 1)
proof(induct n rule: fib.induct)
  case 1 thus ?case by (simp add: fib-2)
next
  case 2 thus ?case by (simp add: fib-2 mod-Suc)
next
  case (3 x)
  have Suc 0 ≠ x mod 2 ⟶ x mod 2 = 0 by presburger
  with 3.hyps show ?case by (simp add: fib.simps add-mult-distrib add-mult-distrib2)
qed

```

We now obtain a version for the natural numbers via the coercion function *int*.

```

theorem fib-Cassini:
  fib (Suc (Suc n)) * fib n =
    (if n mod 2 = 0 then fib (Suc n) * fib (Suc n) - 1
     else fib (Suc n) * fib (Suc n) + 1)
apply (rule int-int-eq [THEN iffD1])
apply (simp add: fib-Cassini-int)
apply (subst zdiff-int [symmetric])
apply (insert fib-Suc-gr-0 [of n], simp-all)
done

```

Toward Law 6.111 of Concrete Mathematics

```

lemma gcd-fib-Suc-eq-1: gcd (fib n) (fib (Suc n)) = Suc 0
apply (induct n rule: fib.induct)
  prefer 3
  apply (simp add: gcd-commute fib-Suc3)
  apply (simp-all add: fib-2)
done

lemma gcd-fib-add: gcd (fib m) (fib (n + m)) = gcd (fib m) (fib n)
apply (simp add: gcd-commute [of fib m])
apply (case-tac m)
  apply simp
  apply (simp add: fib-add)
  apply (simp add: add-commute gcd-non-0 [OF fib-Suc-gr-0])

```

```

apply (simp add: gcd-non-0 [OF fib-Suc-gr-0, symmetric])
apply (simp add: gcd-fib-Suc-eq-1 gcd-mult-cancel)
done

lemma gcd-fib-diff:  $m \leq n \implies \text{gcd} (\text{fib } m) (\text{fib } (n - m)) = \text{gcd} (\text{fib } m) (\text{fib } n)$ 
  by (simp add: gcd-fib-add [symmetric, of - n-m])

lemma gcd-fib-mod:  $0 < m \implies \text{gcd} (\text{fib } m) (\text{fib } (n \bmod m)) = \text{gcd} (\text{fib } m) (\text{fib } n)$ 
proof (induct n rule: less-induct)
  case (less n)
  from less.premis have pos-m:  $0 < m$  .
  show  $\text{gcd} (\text{fib } m) (\text{fib } (n \bmod m)) = \text{gcd} (\text{fib } m) (\text{fib } n)$ 
  proof (cases  $m < n$ )
    case True note m-n = True
    then have m-n':  $m \leq n$  by auto
    with pos-m have pos-n:  $0 < n$  by auto
    with pos-m m-n have diff:  $n - m < n$  by auto
    have  $\text{gcd} (\text{fib } m) (\text{fib } (n \bmod m)) = \text{gcd} (\text{fib } m) (\text{fib } ((n - m) \bmod m))$ 
    by (simp add: mod-if [of n]) (insert m-n, auto)
    also have  $\dots = \text{gcd} (\text{fib } m) (\text{fib } (n - m))$  by (simp add: less.hyps diff pos-m)
    also have  $\dots = \text{gcd} (\text{fib } m) (\text{fib } n)$  by (simp add: gcd-fib-diff m-n')
    finally show  $\text{gcd} (\text{fib } m) (\text{fib } (n \bmod m)) = \text{gcd} (\text{fib } m) (\text{fib } n)$  .
  next
    case False then show  $\text{gcd} (\text{fib } m) (\text{fib } (n \bmod m)) = \text{gcd} (\text{fib } m) (\text{fib } n)$ 
    by (cases  $m = n$ ) auto
  qed
qed

lemma fib-gcd:  $\text{fib} (\text{gcd } m \ n) = \text{gcd} (\text{fib } m) (\text{fib } n)$  — Law 6.111
  apply (induct m n rule: gcd-induct)
  apply (simp-all add: gcd-non-0 gcd-commute gcd-fib-mod)
  done

theorem fib-mult-eq-setsum:
   $\text{fib} (\text{Suc } n) * \text{fib } n = (\sum k \in \{..n\}. \text{fib } k * \text{fib } k)$ 
  apply (induct n rule: fib.induct)
  apply (auto simp add: atMost-Suc fib-2)
  apply (simp add: add-mult-distrib add-mult-distrib2)
  done

end

```

4 Fundamental Theorem of Arithmetic (unique factorization into primes)

theory Factorization


```

imports Main ~~/src/HOL/Old-Number-Theory/Primes Permutation
begin

```

4.1 Definitions

definition

```

  primel :: nat list => bool where
  primel xs = (∀ p ∈ set xs. prime p)

```

consts

```

  nondec :: nat list => bool
  prod :: nat list => nat
  oinsert :: nat => nat list => nat list
  sort :: nat list => nat list

```

primrec

```

  nondec [] = True
  nondec (x # xs) = (case xs of [] => True | y # ys => x ≤ y ∧ nondec xs)

```

primrec

```

  prod [] = Suc 0
  prod (x # xs) = x * prod xs

```

primrec

```

  oinsert x [] = [x]
  oinsert x (y # ys) = (if x ≤ y then x # y # ys else y # oinsert x ys)

```

primrec

```

  sort [] = []
  sort (x # xs) = oinsert x (sort xs)

```

4.2 Arithmetic

```

lemma one-less-m: (m::nat) ≠ m * k ==> m ≠ Suc 0 ==> Suc 0 < m
apply (cases m)
apply auto
done

```

```

lemma one-less-k: (m::nat) ≠ m * k ==> Suc 0 < m * k ==> Suc 0 < k
apply (cases k)
apply auto
done

```

```

lemma mult-left-cancel: (0::nat) < k ==> k * n = k * m ==> n = m
apply auto
done

```

```

lemma mn-eq-m-one: (0::nat) < m ==> m * n = m ==> n = Suc 0
apply (cases n)
apply auto

```

done

lemma *prod-mn-less-k*:

$(0::nat) < n ==> 0 < k ==> Suc\ 0 < m ==> m * n = k ==> n < k$

apply (*induct m*)

apply *auto*

done

4.3 Prime list and product

lemma *prod-append*: $prod\ (xs\ @\ ys) = prod\ xs * prod\ ys$

apply (*induct xs*)

apply (*simp-all add: mult-assoc*)

done

lemma *prod-xy-prod*:

$prod\ (x\ \#\ xs) = prod\ (y\ \#\ ys) ==> x * prod\ xs = y * prod\ ys$

apply *auto*

done

lemma *primel-append*: $primel\ (xs\ @\ ys) = (primel\ xs \wedge primel\ ys)$

apply (*unfold primel-def*)

apply *auto*

done

lemma *prime-primel*: $prime\ n ==> primel\ [n] \wedge prod\ [n] = n$

apply (*unfold primel-def*)

apply *auto*

done

lemma *prime-nd-one*: $prime\ p ==> \neg p\ dvd\ Suc\ 0$

apply (*unfold prime-def dvd-def*)

apply *auto*

done

lemma *hd-dvd-prod*: $prod\ (x\ \#\ xs) = prod\ ys ==> x\ dvd\ (prod\ ys)$

by (*metis dvd-mult-left dvd-refl prod.simps(2)*)

lemma *primel-tl*: $primel\ (x\ \#\ xs) ==> primel\ xs$

apply (*unfold primel-def*)

apply *auto*

done

lemma *primel-hd-tl*: $(primel\ (x\ \#\ xs)) = (prime\ x \wedge primel\ xs)$

apply (*unfold primel-def*)

apply *auto*

done

lemma *primes-eq*: $prime\ p ==> prime\ q ==> p\ dvd\ q ==> p = q$

```

apply (unfold prime-def)
apply auto
done

lemma primel-one-empty: primel xs ==> prod xs = Suc 0 ==> xs = []
apply (cases xs)
apply (simp-all add: primel-def prime-def)
done

lemma prime-g-one: prime p ==> Suc 0 < p
apply (unfold prime-def)
apply auto
done

lemma prime-g-zero: prime p ==> 0 < p
apply (unfold prime-def)
apply auto
done

lemma primel-nempty-g-one:
  primel xs ==> xs ≠ [] ==> Suc 0 < prod xs
apply (induct xs)
apply simp
apply (fastsimp simp: primel-def prime-def elim: one-less-mult)
done

lemma primel-prod-gz: primel xs ==> 0 < prod xs
apply (induct xs)
apply (auto simp: primel-def prime-def)
done

#### 4.4 Sorting

lemma nondec-oinsert: nondec xs ==> nondec (oinsert x xs)
apply (induct xs)
apply simp
apply (case-tac xs)
apply (simp-all cong del: list.weak-case-cong)
done

lemma nondec-sort: nondec (sort xs)
apply (induct xs)
apply simp-all
apply (erule nondec-oinsert)
done

lemma x-less-y-oinsert: x ≤ y ==> l = y # ys ==> x # l = oinsert x l
apply simp-all
done

```

```

lemma nondec-sort-eq [rule-format]: nondec xs  $\longrightarrow$  xs = sort xs
  apply (induct xs)
  apply safe
  apply simp-all
  apply (case-tac xs)
  apply simp-all
  apply (case-tac xs)
  apply simp
  apply (rule-tac y = aa and ys = list in x-less-y-oinsert)
  apply simp-all
done

```

```

lemma oinsert-x-y: oinsert x (oinsert y l) = oinsert y (oinsert x l)
  apply (induct l)
  apply auto
done

```

4.5 Permutation

```

lemma perm-primel [rule-format]: xs <~~> ys ==> primel xs --> primel ys
  apply (unfold primel-def)
  apply (induct set: perm)
  apply simp
  apply simp
  apply (simp (no-asm))
  apply blast
  apply blast
done

```

```

lemma perm-prod: xs <~~> ys ==> prod xs = prod ys
  apply (induct set: perm)
  apply (simp-all add: mult-ac)
done

```

```

lemma perm-subst-oinsert: xs <~~> ys ==> oinsert a xs <~~> oinsert a ys
  apply (induct set: perm)
  apply auto
done

```

```

lemma perm-oinsert: x # xs <~~> oinsert x xs
  apply (induct xs)
  apply auto
done

```

```

lemma perm-sort: xs <~~> sort xs
  apply (induct xs)
  apply (auto intro: perm-oinsert elim: perm-subst-oinsert)
done

```

```

lemma perm-sort-eq:  $xs <\sim\sim> ys \implies \text{sort } xs = \text{sort } ys$ 
  apply (induct set: perm)
    apply (simp-all add: oinsert-x-y)
  done

```

4.6 Existence

```

lemma ex-nondec-lemma:
   $\text{primel } xs \implies \exists ys. \text{primel } ys \wedge \text{nondec } ys \wedge \text{prod } ys = \text{prod } xs$ 
  apply (blast intro: nondec-sort perm-prod perm-primel perm-sort perm-sym)
  done

```

```

lemma not-prime-ex-mk:
   $\text{Suc } 0 < n \wedge \neg \text{prime } n \implies$ 
     $\exists m k. \text{Suc } 0 < m \wedge \text{Suc } 0 < k \wedge m < n \wedge k < n \wedge n = m * k$ 
  apply (unfold prime-def dvd-def)
  apply (auto intro: n-less-m-mult-n n-less-n-mult-m one-less-m one-less-k)
  done

```

```

lemma split-primel:
   $\text{primel } xs \implies \text{primel } ys \implies \exists l. \text{primel } l \wedge \text{prod } l = \text{prod } xs * \text{prod } ys$ 
  apply (rule exI)
  apply safe
    apply (rule-tac [2] prod-append)
  apply (simp add: primel-append)
  done

```

```

lemma factor-exists [rule-format]:  $\text{Suc } 0 < n \dashv\dashv (\exists l. \text{primel } l \wedge \text{prod } l = n)$ 
  apply (induct n rule: nat-less-induct)
  apply (rule impI)
  apply (case-tac prime n)
  apply (rule exI)
  apply (erule prime-primel)
  apply (cut-tac n = n in not-prime-ex-mk)
  apply (auto intro!: split-primel)
  done

```

```

lemma nondec-factor-exists:  $\text{Suc } 0 < n \implies \exists l. \text{primel } l \wedge \text{nondec } l \wedge \text{prod } l = n$ 
  apply (erule factor-exists [THEN exE])
  apply (blast intro!: ex-nondec-lemma)
  done

```

4.7 Uniqueness

```

lemma prime-dvd-mult-list [rule-format]:
   $\text{prime } p \implies p \text{ dvd } (\text{prod } xs) \dashv\dashv (\exists m. m:\text{set } xs \wedge p \text{ dvd } m)$ 
  apply (induct xs)
  apply (force simp add: prime-def)

```

```

apply (force dest: prime-dvd-mult)
done

lemma hd-xs-dvd-prod:
  primel (x # xs) ==> primel ys ==> prod (x # xs) = prod ys
  ==>  $\exists m. m \in \text{set } ys \wedge x \text{ dvd } m$ 
apply (rule prime-dvd-mult-list)
apply (simp add: primel-hd-tl)
apply (erule hd-dvd-prod)
done

lemma prime-dvd-eq: primel (x # xs) ==> primel ys ==>  $m \in \text{set } ys \implies x \text{ dvd } m \implies x = m$ 
apply (rule primes-eq)
apply (auto simp add: primel-def primel-hd-tl)
done

lemma hd-xs-eq-prod:
  primel (x # xs) ==>
  primel ys ==> prod (x # xs) = prod ys ==>  $x \in \text{set } ys$ 
apply (frule hd-xs-dvd-prod)
apply auto
apply (drule prime-dvd-eq)
apply auto
done

lemma perm-primel-ex:
  primel (x # xs) ==>
  primel ys ==> prod (x # xs) = prod ys ==>  $\exists l. ys <\sim\sim> (x \# l)$ 
apply (rule exI)
apply (rule perm-remove)
apply (erule hd-xs-eq-prod)
apply simp-all
done

lemma primel-prod-less:
  primel (x # xs) ==>
  primel ys ==> prod (x # xs) = prod ys ==>  $\text{prod } xs < \text{prod } ys$ 
by (metis less-asym linorder-neqE-nat mult-less-cancel2 nat-0-less-mult-iff
  nat-less-le nat-mult-1 prime-def primel-hd-tl primel-prod-gz prod.simps(2))

lemma prod-one-empty:
  primel xs ==>  $p * \text{prod } xs = p \implies \text{prime } p \implies xs = []$ 
apply (auto intro: primel-one-empty simp add: prime-def)
done

lemma uniq-ex-aux:
   $\forall m. m < \text{prod } ys \longrightarrow (\forall xs \text{ } ys. \text{primel } xs \wedge \text{primel } ys \wedge$ 
   $\text{prod } xs = \text{prod } ys \wedge \text{prod } xs = m \longrightarrow xs <\sim\sim> ys) \implies$ 

```

```

    primel list ==> primel x ==> prod list = prod x ==> prod x < prod ys
    ==> x <~~> list
  apply simp
done

lemma factor-unique [rule-format]:
  ∀ xs ys. primel xs ∧ primel ys ∧ prod xs = prod ys ∧ prod xs = n
    --> xs <~~> ys
  apply (induct n rule: nat-less-induct)
  apply safe
  apply (case-tac xs)
  apply (force intro: primel-one-empty)
  apply (rule perm-primel-ex [THEN exE])
  apply simp-all
  apply (rule perm.trans [THEN perm-sym])
  apply assumption
  apply (rule perm.Cons)
  apply (case-tac x = [])
  apply (metis perm-prod perm-refl prime-primel primel-hd-tl primel-tl prod-one-empty)
  apply (metis nat-0-less-mult-iff nat-mult-eq-cancel1 perm-primel perm-prod primel-prod-gz
    primel-prod-less primel-tl prod.simps(2))
  done

lemma perm-nondec-unique:
  xs <~~> ys ==> nondec xs ==> nondec ys ==> xs = ys
  by (metis nondec-sort-eq perm-sort-eq)

theorem unique-prime-factorization [rule-format]:
  ∀ n. Suc 0 < n --> (∃ !l. primel l ∧ nondec l ∧ prod l = n)
  by (metis factor-unique nondec-factor-exists perm-nondec-unique)

end

```

5 Divisibility and prime numbers (on integers)

```

theory IntPrimes
imports Main Primes
begin

```

The *dvd* relation, GCD, Euclid's extended algorithm, primes, congruences (all on the Integers). Comparable to theory *Primes*, but *dvd* is included here as it is not present in main HOL. Also includes extended GCD and congruences not present in *Primes*.

5.1 Definitions

```

fun

```

```

xzgcda :: int => int => int => int => int => int => int => int => (int * int * int)
where
  xzgcda m n r' r s' s t' t =
    (if r ≤ 0 then (r', s', t')
     else xzgcda m n r (r' mod r)
        s (s' - (r' div r) * s)
        t (t' - (r' div r) * t))

```

definition

```

zprime :: int => bool where
zprime p = (1 < p ∧ (∀ m. 0 <= m & m dvd p ==> m = 1 ∨ m = p))

```

definition

```

xzgcd :: int => int => int * int * int where
xzgcd m n = xzgcda m n m n 1 0 0 1

```

definition

```

zcong :: int => int => int => bool ((1[- = -] '(mod -))) where
[a = b] (mod m) = (m dvd (a - b))

```

5.2 Euclid's Algorithm and GCD

lemma *zrelprime-zdvd-zmult-aux*:

```

  zgcd n k = 1 ==> k dvd m * n ==> 0 ≤ m ==> k dvd m
  by (metis abs-of-nonneg dvd-triv-right zgcd-greatest-iff zgcd-zmult-distrib2-abs
    zmult-1-right)

```

lemma *zrelprime-zdvd-zmult*: $zgcd\ n\ k = 1 \implies k\ dvd\ m * n \implies k\ dvd\ m$

```

apply (case-tac 0 ≤ m)
apply (blast intro: zrelprime-zdvd-zmult-aux)
apply (subgoal-tac k dvd -m)
apply (rule-tac [2] zrelprime-zdvd-zmult-aux, auto)
done

```

lemma *zgcd-geq-zero*: $0 \leq zgcd\ x\ y$

```

by (auto simp add: zgcd-def)

```

This is merely a sanity check on *zprime*, since the previous version denoted the empty set.

lemma *zprime 2*

```

apply (auto simp add: zprime-def)
apply (frule zdvd-imp-le, simp)
apply (auto simp add: order-le-less dvd-def)
done

```

lemma *zprime-imp-zrelprime*:

```

  zprime p ==> ¬ p dvd n ==> zgcd n p = 1
  apply (auto simp add: zprime-def)
  apply (metis zgcd-geq-zero zgcd-zdvd1 zgcd-zdvd2)

```


done

lemma *zless-zprime-imp-zrelprime*:

$zprime\ p \implies 0 < n \implies n < p \implies zgcd\ n\ p = 1$

apply (*erule* *zprime-imp-zrelprime*)

apply (*erule* *zdvd-not-zless*, *assumption*)

done

lemma *zprime-zdvd-zmult*:

$0 \leq (m::int) \implies zprime\ p \implies p\ dvd\ m * n \implies p\ dvd\ m \vee p\ dvd\ n$

by (*metis* *zgcd-zdvd1* *zgcd-zdvd2* *zgcd-pos* *zprime-def* *zrelprime-dvd-mult*)

lemma *zgcd-zadd-zmult* [*simp*]: $zgcd\ (m + n * k)\ n = zgcd\ m\ n$

apply (*rule* *zgcd-eq* [*THEN* *trans*])

apply (*simp* *add: mod-add-eq*)

apply (*rule* *zgcd-eq* [*symmetric*])

done

lemma *zgcd-zdvd-zgcd-zmult*: $zgcd\ m\ n\ dvd\ zgcd\ (k * m)\ n$

by (*simp* *add: zgcd-greatest-iff*)

lemma *zgcd-zmult-zdvd-zgcd*:

$zgcd\ k\ n = 1 \implies zgcd\ (k * m)\ n\ dvd\ zgcd\ m\ n$

apply (*simp* *add: zgcd-greatest-iff*)

apply (*rule-tac* $n = k$ **in** *zrelprime-zdvd-zmult*)

prefer 2

apply (*simp* *add: zmult-commute*)

apply (*metis* *zgcd-1* *zgcd-commute* *zgcd-left-commute*)

done

lemma *zgcd-zmult-cancel*: $zgcd\ k\ n = 1 \implies zgcd\ (k * m)\ n = zgcd\ m\ n$

by (*simp* *add: zgcd-def* *nat-abs-mult-distrib* *gcd-mult-cancel*)

lemma *zgcd-zgcd-zmult*:

$zgcd\ k\ m = 1 \implies zgcd\ n\ m = 1 \implies zgcd\ (k * n)\ m = 1$

by (*simp* *add: zgcd-zmult-cancel*)

lemma *zdvd-iff-zgcd*: $0 < m \implies m\ dvd\ n \longleftrightarrow zgcd\ n\ m = m$

by (*metis* *abs-of-pos* *zdvd-mult-div-cancel* *zgcd-0* *zgcd-commute* *zgcd-geq-zero* *zgcd-zdvd2* *zgcd-zmult-eq-self*)

5.3 Congruences

lemma *zcong-1* [*simp*]: $[a = b] \pmod{1}$

by (*unfold* *zcong-def*, *auto*)

lemma *zcong-refl* [*simp*]: $[k = k] \pmod{m}$

by (*unfold* *zcong-def*, *auto*)

lemma *zcong-sym*: $[a = b] \text{ (mod } m) = [b = a] \text{ (mod } m)$
unfolding *zcong-def minus-diff-eq* [of *a*, *symmetric*] *dvd-minus-iff* ..

lemma *zcong-zadd*:
 $[a = b] \text{ (mod } m) ==> [c = d] \text{ (mod } m) ==> [a + c = b + d] \text{ (mod } m)$
apply (*unfold zcong-def*)
apply (*rule-tac* $s = (a - b) + (c - d)$ **in** *subst*)
apply (*rule-tac* [2] *dvd-add*, *auto*)
done

lemma *zcong-zdiff*:
 $[a = b] \text{ (mod } m) ==> [c = d] \text{ (mod } m) ==> [a - c = b - d] \text{ (mod } m)$
apply (*unfold zcong-def*)
apply (*rule-tac* $s = (a - b) - (c - d)$ **in** *subst*)
apply (*rule-tac* [2] *dvd-diff*, *auto*)
done

lemma *zcong-trans*:
 $[a = b] \text{ (mod } m) ==> [b = c] \text{ (mod } m) ==> [a = c] \text{ (mod } m)$
unfolding *zcong-def* **by** (*auto elim!*: *dvdE simp add: algebra-simps*)

lemma *zcong-zmult*:
 $[a = b] \text{ (mod } m) ==> [c = d] \text{ (mod } m) ==> [a * c = b * d] \text{ (mod } m)$
apply (*rule-tac* $b = b * c$ **in** *zcong-trans*)
apply (*unfold zcong-def*)
apply (*metis zdiff-zmult-distrib2 dvd-mult zmult-commute*)
apply (*metis zdiff-zmult-distrib2 dvd-mult*)
done

lemma *zcong-scalar*: $[a = b] \text{ (mod } m) ==> [a * k = b * k] \text{ (mod } m)$
by (*rule zcong-zmult, simp-all*)

lemma *zcong-scalar2*: $[a = b] \text{ (mod } m) ==> [k * a = k * b] \text{ (mod } m)$
by (*rule zcong-zmult, simp-all*)

lemma *zcong-zmult-self*: $[a * m = b * m] \text{ (mod } m)$
apply (*unfold zcong-def*)
apply (*rule dvd-diff, simp-all*)
done

lemma *zcong-square*:
 $[| \text{zprime } p; 0 < a; [a * a = 1] \text{ (mod } p) |]$
 $==> [a = 1] \text{ (mod } p) \vee [a = p - 1] \text{ (mod } p)$
apply (*unfold zcong-def*)
apply (*rule zprime-zdvd-zmult*)
apply (*rule-tac* [3] $s = a * a - 1 + p * (1 - a)$ **in** *subst*)
prefer 4
apply (*simp add: zdvd-reduce*)
apply (*simp-all add: zdiff-zmult-distrib zmult-commute zdiff-zmult-distrib2*)

done

lemma *zcong-cancel*:

$0 \leq m \implies$
 $zgcd\ k\ m = 1 \implies [a * k = b * k] \text{ (mod } m) = [a = b] \text{ (mod } m)$
apply *safe*
prefer 2
apply (*blast intro: zcong-scalar*)
apply (*case-tac b < a*)
prefer 2
apply (*subst zcong-sym*)
apply (*unfold zcong-def*)
apply (*rule-tac [!] zrelprime-zdvd-zmult*)
apply (*simp-all add: zdvd-zmult-distrib*)
apply (*subgoal-tac m dvd -(a * k - b * k)*)
apply *simp*
apply (*subst dvd-minus-iff, assumption*)
done

lemma *zcong-cancel2*:

$0 \leq m \implies$
 $zgcd\ k\ m = 1 \implies [k * a = k * b] \text{ (mod } m) = [a = b] \text{ (mod } m)$
by (*simp add: zmult-commute zcong-cancel*)

lemma *zcong-zgcd-zmult-zmod*:

$[a = b] \text{ (mod } m) \implies [a = b] \text{ (mod } n) \implies zgcd\ m\ n = 1$
 $\implies [a = b] \text{ (mod } m * n)$
apply (*auto simp add: zcong-def dvd-def*)
apply (*subgoal-tac m dvd n * ka*)
apply (*subgoal-tac m dvd ka*)
apply (*case-tac [2] 0 ≤ ka*)
apply (*metis zdvd-mult-div-cancel dvd-refl dvd-mult-left zmult-commute zrelprime-zdvd-zmult*)
apply (*metis abs-dvd-iff abs-of-nonneg zadd-0 zgcd-0-left zgcd-commute zgcd-zadd-zmult*
 $zgcd-zdvd-zgcd-zmult\ zgcd-zmult-distrib2-abs\ zmult-1-right\ zmult-commute$)
apply (*metis mult-le-0-iff zdvd-mono zdvd-mult-cancel dvd-triv-left zero-le-mult-iff*
 $zle-antisym\ zle-linear\ zle-refl\ zmult-commute\ zrelprime-zdvd-zmult$)
apply (*metis dvd-triv-left*)
done

lemma *zcong-zless-imp-eq*:

$0 \leq a \implies$
 $a < m \implies 0 \leq b \implies b < m \implies [a = b] \text{ (mod } m) \implies a = b$
apply (*unfold zcong-def dvd-def, auto*)
apply (*drule-tac f = λz. z mod m in arg-cong*)
apply (*metis diff-add-cancel mod-pos-pos-trivial zadd-0 zadd-commute zmod-eq-0-iff*
 $mod-add-right-eq$)
done

lemma *zcong-square-zless*:

```

zprime p ==> 0 < a ==> a < p ==>
  [a * a = 1] (mod p) ==> a = 1 ∨ a = p - 1
apply (cut-tac p = p and a = a in zcong-square)
  apply (simp add: zprime-def)
  apply (auto intro: zcong-zless-imp-eq)
done

lemma zcong-not:
  0 < a ==> a < m ==> 0 < b ==> b < a ==> ¬ [a = b] (mod m)
apply (unfold zcong-def)
apply (rule zdvd-not-zless, auto)
done

lemma zcong-zless-0:
  0 ≤ a ==> a < m ==> [a = 0] (mod m) ==> a = 0
apply (unfold zcong-def dvd-def, auto)
apply (metis div-pos-pos-trivial linorder-not-less div-mult-self1-is-id)
done

lemma zcong-zless-unique:
  0 < m ==> (∃!b. 0 ≤ b ∧ b < m ∧ [a = b] (mod m))
apply auto
  prefer 2 apply (metis zcong-sym zcong-trans zcong-zless-imp-eq)
apply (unfold zcong-def dvd-def)
apply (rule-tac x = a mod m in exI, auto)
apply (metis zmult-div-cancel)
done

lemma zcong-iff-lin: ([a = b] (mod m)) = (∃ k. b = a + m * k)
  unfolding zcong-def
  apply (auto elim!: dvdE simp add: algebra-simps)
  apply (rule-tac x = -k in exI) apply simp
done

lemma zgcd-zcong-zgcd:
  0 < m ==>
    zgcd a m = 1 ==> [a = b] (mod m) ==> zgcd b m = 1
  by (auto simp add: zcong-iff-lin)

lemma zcong-zmod-aux:
  a - b = (m::int) * (a div m - b div m) + (a mod m - b mod m)
  by (simp add: zdiff-zmult-distrib2 add-diff-eq eq-diff-eq add-ac)

lemma zcong-zmod: [a = b] (mod m) = [a mod m = b mod m] (mod m)
  apply (unfold zcong-def)
  apply (rule-tac t = a - b in ssubst)
  apply (rule-tac m = m in zcong-zmod-aux)
  apply (rule trans)
  apply (rule-tac [2] k = m and m = a div m - b div m in zdvd-reduce)

```

```

apply (simp add: zadd-commute)
done

lemma zcong-zmod-eq:  $0 < m \implies [a = b] \text{ (mod } m) = (a \text{ mod } m = b \text{ mod } m)$ 
apply auto
apply (metis pos-mod-conj zcong-zless-imp-eq zcong-zmod)
apply (metis zcong-refl zcong-zmod)
done

lemma zcong-zminus [iff]:  $[a = b] \text{ (mod } -m) = [a = b] \text{ (mod } m)$ 
by (auto simp add: zcong-def)

lemma zcong-zero [iff]:  $[a = b] \text{ (mod } 0) = (a = b)$ 
by (auto simp add: zcong-def)

lemma  $[a = b] \text{ (mod } m) = (a \text{ mod } m = b \text{ mod } m)$ 
apply (case-tac  $m = 0$ , simp add: DIVISION-BY-ZERO)
apply (simp add: linorder-neg-iff)
apply (erule disjE)
prefer 2 apply (simp add: zcong-zmod-eq)

Remaining case:  $m < 0$ 

apply (rule-tac  $t = m$  in zminus-zminus [THEN subst])
apply (subst zcong-zminus)
apply (subst zcong-zmod-eq, arith)
apply (frule neg-mod-bound [of - a], frule neg-mod-bound [of - b])
apply (simp add: zmod-zminus2-eq-if del: neg-mod-bound)
done

```

5.4 Modulo

```

lemma zmod-zdvd-zmod:
 $0 < (m::int) \implies m \text{ dvd } b \implies (a \text{ mod } b \text{ mod } m) = (a \text{ mod } m)$ 
by (rule mod-mod-cancel)

```

5.5 Extended GCD

```

declare xzgcda.simps [simp del]

```

```

lemma xzgcd-correct-aux1:
 $zgcd\ r'\ r = k \implies 0 < r \implies$ 
 $(\exists sn\ tn. xzgcda\ m\ n\ r'\ r\ s'\ s\ t'\ t = (k, sn, tn))$ 
apply (induct  $m\ n\ r'\ r\ s'\ s\ t'\ t$  rule: xzgcda.induct)
apply (subst zgcd-eq)
apply (subst xzgcda.simps, auto)
apply (case-tac  $r' \text{ mod } r = 0$ )
prefer 2
apply (frule-tac  $a = r'$  in pos-mod-sign, auto)
apply (rule exI)
apply (rule exI)

```

```

apply (subst xzgcda.simps, auto)
done

lemma xzgcd-correct-aux2:
  ( $\exists sn\ tn. xzgcda\ m\ n\ r'\ r\ s'\ s\ t'\ t = (k, sn, tn)$ )  $\implies 0 < r \implies$ 
     $zgcd\ r'\ r = k$ 
apply (induct m n r' r s' s t' t rule: xzgcda.induct)
apply (subst zgcd-eq)
apply (subst xzgcda.simps)
apply (auto simp add: linorder-not-le)
apply (case-tac r' mod r = 0)
prefer 2
apply (frule-tac a = r' in pos-mod-sign, auto)
apply (metis Pair-eq xzgcda.simps zle-refl)
done

lemma xzgcd-correct:
   $0 < n \implies (zgcd\ m\ n = k) = (\exists s\ t. xzgcd\ m\ n = (k, s, t))$ 
apply (unfold xzgcd-def)
apply (rule iffI)
apply (rule-tac [2] xzgcd-correct-aux2 [THEN mp, THEN mp])
apply (rule xzgcd-correct-aux1 [THEN mp, THEN mp], auto)
done

xzgcd linear

lemma xzgcda-linear-aux1:
   $(a - r * b) * m + (c - r * d) * (n::int) =$ 
   $(a * m + c * n) - r * (b * m + d * n)$ 
by (simp add: zdiff-zmult-distrib zadd-zmult-distrib2 zmult-assoc)

lemma xzgcda-linear-aux2:
   $r' = s' * m + t' * n \implies r = s * m + t * n$ 
   $\implies (r' \bmod r) = (s' - (r' \text{ div } r) * s) * m + (t' - (r' \text{ div } r) * t) * (n::int)$ 
apply (rule trans)
apply (rule-tac [2] xzgcda-linear-aux1 [symmetric])
apply (simp add: eq-diff-eq mult-commute)
done

lemma order-le-neq-implies-less:  $(x::'a::order) \leq y \implies x \neq y \implies x < y$ 
by (rule iffD2 [OF order-less-le conjI])

lemma xzgcda-linear [rule-format]:
   $0 < r \implies xzgcda\ m\ n\ r'\ r\ s'\ s\ t'\ t = (rn, sn, tn) \implies$ 
   $r' = s' * m + t' * n \implies r = s * m + t * n \implies rn = sn * m + tn * n$ 
apply (induct m n r' r s' s t' t rule: xzgcda.induct)
apply (subst xzgcda.simps)
apply (simp (no-asm))
apply (rule impI)+
apply (case-tac r' mod r = 0)

```

```

    apply (simp add: xzgcda.simps, clarify)
    apply (subgoal-tac  $0 < r' \bmod r$ )
    apply (rule-tac [2] order-le-neq-implies-less)
    apply (rule-tac [2] pos-mod-sign)
    apply (cut-tac  $m = m$  and  $n = n$  and  $r' = r'$  and  $r = r$  and  $s' = s'$  and
       $s = s$  and  $t' = t'$  and  $t = t$  in xzgcda-linear-aux2, auto)
  done

lemma xzgcd-linear:
   $0 < n \implies \text{xzgcd } m \ n = (r, s, t) \implies r = s * m + t * n$ 
  apply (unfold xzgcd-def)
  apply (erule xzgcda-linear, assumption, auto)
  done

lemma zgcd-ex-linear:
   $0 < n \implies \text{zgcd } m \ n = k \implies (\exists s \ t. k = s * m + t * n)$ 
  apply (simp add: xzgcd-correct, safe)
  apply (rule exI)+
  apply (erule xzgcd-linear, auto)
  done

lemma zcong-lineq-ex:
   $0 < n \implies \text{zgcd } a \ n = 1 \implies \exists x. [a * x = 1] \pmod n$ 
  apply (cut-tac  $m = a$  and  $n = n$  and  $k = 1$  in zgcd-ex-linear, safe)
  apply (rule-tac  $x = s$  in exI)
  apply (rule-tac  $b = s * a + t * n$  in zcong-trans)
  prefer 2
  apply simp
  apply (unfold zcong-def)
  apply (simp (no-asm) add: zmult-commute)
  done

lemma zcong-lineq-unique:
   $0 < n \implies$ 
   $\text{zgcd } a \ n = 1 \implies \exists! x. 0 \leq x \wedge x < n \wedge [a * x = b] \pmod n$ 
  apply auto
  apply (rule-tac [2] zcong-zless-imp-eq)
    apply (tactic << stac (thm zcong-cancel2 RS sym) 6 >>>)
    apply (rule-tac [8] zcong-trans)
    apply (simp-all (no-asm-simp))
  prefer 2
  apply (simp add: zcong-sym)
  apply (cut-tac  $a = a$  and  $n = n$  in zcong-lineq-ex, auto)
  apply (rule-tac  $x = x * b \bmod n$  in exI, safe)
    apply (simp-all (no-asm-simp))
  apply (metis zcong-scalar zcong-zmod zmod-zmult1-eq zmult-1 zmult-assoc)
  done

end

```

6 The Chinese Remainder Theorem

```
theory Chinese
imports IntPrimes
begin
```

The Chinese Remainder Theorem for an arbitrary finite number of equations. (The one-equation case is included in theory *IntPrimes*. Uses functions for indexing.¹

6.1 Definitions

consts

```
funprod :: (nat => int) => nat => nat => int
funsum :: (nat => int) => nat => nat => int
```

primrec

```
funprod f i 0 = f i
funprod f i (Suc n) = f (Suc (i + n)) * funprod f i n
```

primrec

```
funsum f i 0 = f i
funsum f i (Suc n) = f (Suc (i + n)) + funsum f i n
```

definition

```
m-cond :: nat => (nat => int) => bool where
m-cond n mf =
  (( $\forall i. i \leq n \longrightarrow 0 < mf\ i$ )  $\wedge$ 
   ( $\forall i\ j. i \leq n \wedge j \leq n \wedge i \neq j \longrightarrow zgcd\ (mf\ i)\ (mf\ j) = 1$ ))
```

definition

```
km-cond :: nat => (nat => int) => (nat => int) => bool where
km-cond n kf mf = ( $\forall i. i \leq n \longrightarrow zgcd\ (kf\ i)\ (mf\ i) = 1$ )
```

definition

```
lincong-sol ::
  nat => (nat => int) => (nat => int) => (nat => int) => int => bool
```

where

```
lincong-sol n kf bf mf x = ( $\forall i. i \leq n \longrightarrow zcong\ (kf\ i * x)\ (bf\ i)\ (mf\ i)$ )
```

definition

```
mhf :: (nat => int) => nat => nat => int where
mhf mf n i =
  (if i = 0 then funprod mf (Suc 0) (n - Suc 0)
   else if i = n then funprod mf 0 (n - Suc 0))
```

¹Maybe *funprod* and *funsum* should be based on general *fold* on indices?

*else funprod mf 0 (i - Suc 0) * funprod mf (Suc i) (n - Suc 0 - i))*

definition

xilin-sol ::

nat => nat => (nat => int) => (nat => int) => (nat => int) => int

where

xilin-sol i n kf bf mf =

(if 0 < n ∧ i ≤ n ∧ m-cond n mf ∧ km-cond n kf mf then

*(SOME x. 0 ≤ x ∧ x < mf i ∧ zcong (kf i * mhf mf n i * x) (bf i) (mf i))*

else 0)

definition

x-sol :: nat => (nat => int) => (nat => int) => (nat => int) => int **where**

*x-sol n kf bf mf = funsum (λi. xilin-sol i n kf bf mf * mhf mf n i) 0 n*

funprod and *funsum*

lemma *funprod-pos*: $(\forall i. i \leq n \longrightarrow 0 < mf\ i) \implies 0 < funprod\ mf\ 0\ n$

apply *(induct n)*

apply *auto*

apply *(simp add: zero-less-mult-iff)*

done

lemma *funprod-zgcd* [*rule-format* (*no-asm*)]:

$(\forall i. k \leq i \wedge i \leq k + l \longrightarrow zgcd\ (mf\ i)\ (mf\ m) = 1) \longrightarrow$

$zgcd\ (funprod\ mf\ k\ l)\ (mf\ m) = 1$

apply *(induct l)*

apply *simp-all*

apply *(rule impI)+*

apply *(subst zgcd-zmult-cancel)*

apply *auto*

done

lemma *funprod-zdvd* [*rule-format*]:

$k \leq i \longrightarrow i \leq k + l \longrightarrow mf\ i\ dvd\ funprod\ mf\ k\ l$

apply *(induct l)*

apply *auto*

apply *(subgoal-tac i = Suc (k + l))*

apply *(simp-all (no-asm-simp))*

done

lemma *funsum-mod*:

$funsum\ f\ k\ l\ mod\ m = funsum\ (\lambda i. (f\ i)\ mod\ m)\ k\ l\ mod\ m$

apply *(induct l)*

apply *auto*

apply *(rule trans)*

apply *(rule mod-add-eq)*

apply *simp*

apply *(rule mod-add-right-eq [symmetric])*

done

```

lemma funsum-zero [rule-format (no-asm)]:
  ( $\forall i. k \leq i \wedge i \leq k + l \longrightarrow f\ i = 0$ )  $\longrightarrow$  (funsum f k l) = 0
apply (induct l)
apply auto
done

```

```

lemma funsum-oneelem [rule-format (no-asm)]:
   $k \leq j \longrightarrow j \leq k + l \longrightarrow$ 
  ( $\forall i. k \leq i \wedge i \leq k + l \wedge i \neq j \longrightarrow f\ i = 0$ )  $\longrightarrow$ 
  funsum f k l = f j
apply (induct l)
prefer 2
apply clarify
defer
apply clarify
apply (subgoal-tac k = j)
apply (simp-all (no-asm-simp))
apply (case-tac Suc (k + l) = j)
apply (subgoal-tac funsum f k l = 0)
apply (rule-tac [2] funsum-zero)
apply (subgoal-tac [3] f (Suc (k + l)) = 0)
apply (subgoal-tac [3] j ≤ k + l)
prefer 4
apply arith
apply auto
done

```

6.2 Chinese: uniqueness

```

lemma zcong-funprod-aux:
  m-cond n mf  $\implies$  km-cond n kf mf
   $\implies$  lincong-sol n kf bf mf x  $\implies$  lincong-sol n kf bf mf y
   $\implies$  [x = y] (mod mf n)
apply (unfold m-cond-def km-cond-def lincong-sol-def)
apply (rule iffD1)
apply (rule-tac k = kf n in zcong-cancel2)
apply (rule-tac [3] b = bf n in zcong-trans)
prefer 4
apply (subst zcong-sym)
defer
apply (rule order-less-imp-le)
apply simp-all
done

```

```

lemma zcong-funprod [rule-format]:
  m-cond n mf  $\longrightarrow$  km-cond n kf mf  $\longrightarrow$ 
  lincong-sol n kf bf mf x  $\longrightarrow$  lincong-sol n kf bf mf y  $\longrightarrow$ 
  [x = y] (mod funprod mf 0 n)

```

```

apply (induct n)
apply (simp-all (no-asm))
apply (blast intro: zcong-funprod-aux)
apply (rule impI)+
apply (rule zcong-zgcd-zmult-zmod)
  apply (blast intro: zcong-funprod-aux)
  prefer 2
  apply (subst zgcd-commute)
  apply (rule funprod-zgcd)
apply (auto simp add: m-cond-def km-cond-def lincong-sol-def)
done

```

6.3 Chinese: existence

lemma unique-xi-sol:

```

0 < n ==> i ≤ n ==> m-cond n mf ==> km-cond n kf mf
==> ∃!x. 0 ≤ x ∧ x < mf i ∧ [kf i * mhf mf n i * x = bf i] (mod mf i)
apply (rule zcong-lineq-unique)
apply (tactic ⟨ stac (thm zgcd-zmult-cancel) 2 ⟩)
  apply (unfold m-cond-def km-cond-def mhf-def)
  apply (simp-all (no-asm-simp))
apply safe
  apply (tactic ⟨ stac (thm zgcd-zmult-cancel) 3 ⟩)
  apply (rule-tac [!] funprod-zgcd)
  apply safe
  apply simp-all
apply (subgoal-tac i < n)
  prefer 2
  apply arith
  apply (case-tac [2] i)
  apply simp-all
done

```

lemma x-sol-lin-aux:

```

0 < n ==> i ≤ n ==> j ≤ n ==> j ≠ i ==> mf j dvd mhf mf n i
apply (unfold mhf-def)
apply (case-tac i = 0)
  apply (case-tac [2] i = n)
  apply (simp-all (no-asm-simp))
  apply (case-tac [3] j < i)
  apply (rule-tac [3] dvd-mult2)
  apply (rule-tac [4] dvd-mult)
  apply (rule-tac [!] funprod-zdvd)
  apply arith
  apply arith
  apply arith
  apply arith
  apply arith

```

```

    apply arith
    apply arith
done

```

```

lemma x-sol-lin:
  0 < n ==> i ≤ n
  ==> x-sol n kf bf mf mod mf i =
    xilin-sol i n kf bf mf * mhf mf n i mod mf i
  apply (unfold x-sol-def)
  apply (subst funsum-mod)
  apply (subst funsum-oneelem)
  apply auto
  apply (subst dvd-eq-mod-eq-0 [symmetric])
  apply (rule dvd-mult)
  apply (rule x-sol-lin-aux)
  apply auto
done

```

6.4 Chinese

```

lemma chinese-remainder:
  0 < n ==> m-cond n mf ==> km-cond n kf mf
  ==> ∃!x. 0 ≤ x ∧ x < funprod mf 0 n ∧ lincong-sol n kf bf mf x
  apply safe
  apply (rule-tac [2] m = funprod mf 0 n in zcong-zless-imp-eq)
  apply (rule-tac [6] zcong-funprod)
  apply auto
  apply (rule-tac x = x-sol n kf bf mf mod funprod mf 0 n in exI)
  apply (unfold lincong-sol-def)
  apply safe
  apply (tactic ⟨ stac (thm zcong-zmod) 3 ⟩)
  apply (tactic ⟨ stac (thm mod-mult-eq) 3 ⟩)
  apply (tactic ⟨ stac (thm mod-mod-cancel) 3 ⟩)
  apply (tactic ⟨ stac (thm x-sol-lin) 4 ⟩)
  apply (tactic ⟨ stac (thm mod-mult-eq RS sym) 6 ⟩)
  apply (tactic ⟨ stac (thm zcong-zmod RS sym) 6 ⟩)
  apply (subgoal-tac [6]
    0 ≤ xilin-sol i n kf bf mf ∧ xilin-sol i n kf bf mf < mf i
    ∧ [kf i * mhf mf n i * xilin-sol i n kf bf mf = bf i] (mod mf i))
  prefer 6
  apply (simp add: zmult-ac)
  apply (unfold xilin-sol-def)
  apply (tactic ⟨ asm-simp-tac @ {simpset} 6 ⟩)
  apply (rule-tac [6] ex1-implies-ex [THEN someI-ex])
  apply (rule-tac [6] unique-xi-sol)
  apply (rule-tac [3] funprod-zdvd)
  apply (unfold m-cond-def)
  apply (rule funprod-pos [THEN pos-mod-sign])
  apply (rule-tac [2] funprod-pos [THEN pos-mod-bound])

```

```

    apply auto
done

end

```

7 Bijections between sets

theory *BijectionRel* **imports** *Main* **begin**

Inductive definitions of bijections between two different sets and between the same set. Theorem for relating the two definitions.

inductive-set

```

  bijR :: ('a => 'b => bool) => ('a set * 'b set) set
  for P :: 'a => 'b => bool
where
    empty [simp]: ({}, {}) ∈ bijR P
  | insert: P a b ==> a ∉ A ==> b ∉ B ==> (A, B) ∈ bijR P
    ==> (insert a A, insert b B) ∈ bijR P

```

Add extra condition to *insert*: $\forall b \in B. \neg P a b$ (and similar for *A*).

definition

```

  bijP :: ('a => 'a => bool) => 'a set => bool where
  bijP P F = (∀ a b. a ∈ F ∧ P a b --> b ∈ F)

```

definition

```

  uniqP :: ('a => 'a => bool) => bool where
  uniqP P = (∀ a b c d. P a b ∧ P c d --> (a = c) = (b = d))

```

definition

```

  symP :: ('a => 'a => bool) => bool where
  symP P = (∀ a b. P a b = P b a)

```

inductive-set

```

  bijER :: ('a => 'a => bool) => 'a set set
  for P :: 'a => 'a => bool
where
    empty [simp]: {} ∈ bijER P
  | insert1: P a a ==> a ∉ A ==> A ∈ bijER P ==> insert a A ∈ bijER P
  | insert2: P a b ==> a ≠ b ==> a ∉ A ==> b ∉ A ==> A ∈ bijER P
    ==> insert a (insert b A) ∈ bijER P

```

bijR

lemma *fin-bijRl*: $(A, B) \in \text{bijR } P \implies \text{finite } A$

```

  apply (erule bijR.induct)
  apply auto

```

```

done

lemma fin-bijRr:  $(A, B) \in \text{bijR } P \implies \text{finite } B$ 
  apply (erule bijR.induct)
  apply auto
done

lemma aux-induct:
  assumes major:  $\text{finite } F$ 
  and subs:  $F \subseteq A$ 
  and cases:  $P \ \{\}$ 
  !! $F \ a. F \subseteq A \implies a \in A \implies a \notin F \implies P \ F \implies P \ (\text{insert } a \ F)$ 
  shows  $P \ F$ 
  using major subs
  apply (induct set: finite)
  apply (blast intro: cases)+
done

lemma inj-func-bijR-aux1:
   $A \subseteq B \implies a \notin A \implies a \in B \implies \text{inj-on } f \ B \implies f \ a \notin f \ ' \ A$ 
  apply (unfold inj-on-def)
  apply auto
done

lemma inj-func-bijR-aux2:
   $\forall a. a \in A \dashrightarrow P \ a \ (f \ a) \implies \text{inj-on } f \ A \implies \text{finite } A \implies F \leq A$ 
   $\implies (F, f \ ' \ F) \in \text{bijR } P$ 
  apply (rule-tac  $F = F$  and  $A = A$  in aux-induct)
  apply (rule finite-subset)
  apply auto
  apply (rule bijR.insert)
  apply (rule-tac [3] inj-func-bijR-aux1)
  apply auto
done

lemma inj-func-bijR:
   $\forall a. a \in A \dashrightarrow P \ a \ (f \ a) \implies \text{inj-on } f \ A \implies \text{finite } A$ 
   $\implies (A, f \ ' \ A) \in \text{bijR } P$ 
  apply (rule inj-func-bijR-aux2)
  apply auto
done

bijER

lemma fin-bijER:  $A \in \text{bijER } P \implies \text{finite } A$ 
  apply (erule bijER.induct)
  apply auto
done

```

```

lemma aux1:
   $a \notin A \implies a \notin B \implies F \subseteq \text{insert } a \ A \implies F \subseteq \text{insert } a \ B \implies a \in F$ 
   $\implies \exists C. F = \text{insert } a \ C \wedge a \notin C \wedge C \leq A \wedge C \leq B$ 
  apply (rule-tac  $x = F - \{a\}$  in exI)
  apply auto
  done

lemma aux2:  $a \neq b \implies a \notin A \implies b \notin B \implies a \in F \implies b \in F$ 
   $\implies F \subseteq \text{insert } a \ A \implies F \subseteq \text{insert } b \ B$ 
   $\implies \exists C. F = \text{insert } a \ (\text{insert } b \ C) \wedge a \notin C \wedge b \notin C \wedge C \subseteq A \wedge C \subseteq B$ 
  apply (rule-tac  $x = F - \{a, b\}$  in exI)
  apply auto
  done

lemma aux-uniq:  $\text{uniqP } P \implies P \ a \ b \implies P \ c \ d \implies (a = c) = (b = d)$ 
  apply (unfold uniqP-def)
  apply auto
  done

lemma aux-sym:  $\text{symP } P \implies P \ a \ b = P \ b \ a$ 
  apply (unfold symP-def)
  apply auto
  done

lemma aux-in1:
   $\text{uniqP } P \implies b \notin C \implies P \ b \ b \implies \text{bijP } P \ (\text{insert } b \ C) \implies \text{bijP } P \ C$ 
  apply (unfold bijP-def)
  apply auto
  apply (subgoal-tac  $b \neq a$ )
  prefer 2
  apply clarify
  apply (simp add: aux-uniq)
  apply auto
  done

lemma aux-in2:
   $\text{symP } P \implies \text{uniqP } P \implies a \notin C \implies b \notin C \implies a \neq b \implies P \ a \ b$ 
   $\implies \text{bijP } P \ (\text{insert } a \ (\text{insert } b \ C)) \implies \text{bijP } P \ C$ 
  apply (unfold bijP-def)
  apply auto
  apply (subgoal-tac  $aa \neq a$ )
  prefer 2
  apply clarify
  apply (subgoal-tac  $aa \neq b$ )
  prefer 2
  apply clarify
  apply (simp add: aux-uniq)
  apply (subgoal-tac  $ba \neq a$ )
  apply auto

```

```

apply (subgoal-tac  $P\ a\ aa$ )
prefer 2
apply (simp add: aux-sym)
apply (subgoal-tac  $b = aa$ )
apply (rule-tac [2] iffD1)
apply (rule-tac [2]  $a = a$  and  $c = a$  and  $P = P$  in aux-uniq)
apply auto
done

lemma aux-foo:  $\forall a\ b. Q\ a \wedge P\ a\ b \dashv\vdash R\ b \implies P\ a\ b \implies Q\ a \implies R\ b$ 
apply auto
done

lemma aux-bij:  $bijP\ P\ F \implies symP\ P \implies P\ a\ b \implies (a \in F) = (b \in F)$ 
apply (unfold bijP-def)
apply (rule iffI)
apply (erule-tac [!] aux-foo)
apply simp-all
apply (rule iffD2)
apply (rule-tac  $P = P$  in aux-sym)
apply simp-all
done

lemma aux-bijRER:
 $(A, B) \in bijR\ P \implies uniqP\ P \implies symP\ P$ 
 $\implies \forall F. bijP\ P\ F \wedge F \subseteq A \wedge F \subseteq B \dashv\vdash F \in bijER\ P$ 
apply (erule bijR.induct)
apply simp
apply (case-tac  $a = b$ )
apply clarify
apply (case-tac  $b \in F$ )
prefer 2
apply (simp add: subset-insert)
apply (cut-tac  $F = F$  and  $a = b$  and  $A = A$  and  $B = B$  in aux1)
prefer 6
apply clarify
apply (rule bijER.insert1)
apply simp-all
apply (subgoal-tac  $bijP\ P\ C$ )
apply simp
apply (rule aux-in1)
apply simp-all
apply clarify
apply (case-tac  $a \in F$ )
apply (case-tac [!]  $b \in F$ )
apply (cut-tac  $F = F$  and  $a = a$  and  $b = b$  and  $A = A$  and  $B = B$ 
in aux2)
apply (simp-all add: subset-insert)

```



```

    apply clarify
    apply (rule bijER.insert2)
      apply simp-all
    apply (subgoal-tac bijP P C)
    apply simp
    apply (rule aux-in2)
      apply simp-all
    apply (subgoal-tac b ∈ F)
    apply (rule-tac [2] iffD1)
      apply (rule-tac [2] a = a and F = F and P = P in aux-bij)
        apply (simp-all (no-asm-simp))
    apply (subgoal-tac [2] a ∈ F)
    apply (rule-tac [3] iffD2)
      apply (rule-tac [3] b = b and F = F and P = P in aux-bij)
    apply auto
  done

lemma bijR-bijER:
  (A, A) ∈ bijR P ==>
    bijP P A ==> uniqP P ==> symP P ==> A ∈ bijER P
  apply (cut-tac A = A and B = A and P = P in aux-bijRER)
    apply auto
  done

end

```

8 Factorial on integers

theory *IntFact* **imports** *IntPrimes* **begin**

Factorial on integers and recursively defined set including all Integers from 2 up to a . Plus definition of product of finite set.

```

fun
  zfact :: int => int
where
  zfact n = (if n ≤ 0 then 1 else n * zfact (n - 1))

fun
  d22set :: int => int set
where
  d22set a = (if 1 < a then insert a (d22set (a - 1)) else {})

```

d22set — recursively defined set including all integers from 2 up to a

declare *d22set.simps* [*simp del*]

```

lemma d22set-induct:
  assumes !!a. P {} a
    and !!a. 1 < (a::int) ==> P (d22set (a - 1)) (a - 1) ==> P (d22set a) a
  shows P (d22set u) u
  apply (rule d22set.induct)
  apply (case-tac 1 < a)
  apply (rule-tac assms)
  apply (simp-all (no-asm-simp))
  apply (simp-all (no-asm-simp) add: d22set.simps assms)
  done

lemma d22set-g-1 [rule-format]: b ∈ d22set a --> 1 < b
  apply (induct a rule: d22set-induct)
  apply simp
  apply (subst d22set.simps)
  apply auto
  done

lemma d22set-le [rule-format]: b ∈ d22set a --> b ≤ a
  apply (induct a rule: d22set-induct)
  apply simp
  apply (subst d22set.simps)
  apply auto
  done

lemma d22set-le-swap: a < b ==> b ∉ d22set a
  by (auto dest: d22set-le)

lemma d22set-mem: 1 < b ==> b ≤ a ==> b ∈ d22set a
  apply (induct a rule: d22set-induct)
  apply auto
  apply (subst d22set.simps)
  apply (case-tac b < a, auto)
  done

lemma d22set-fin: finite (d22set a)
  apply (induct a rule: d22set-induct)
  prefer 2
  apply (subst d22set.simps)
  apply auto
  done

declare zfact.simps [simp del]

lemma d22set-prod-zfact:  $\prod (d22set a) = zfact a$ 
  apply (induct a rule: d22set-induct)
  apply (subst d22set.simps)
  apply (subst zfact.simps)

```

```

    apply (case-tac 1 < a)
    prefer 2
    apply (simp add: d22set.simps zfact.simps)
    apply (simp add: d22set-fin d22set-le-swap)
    done

end

```

9 Fermat's Little Theorem extended to Euler's Totient function

```

theory EulerFermat
imports BijectionRel IntFact
begin

```

Fermat's Little Theorem extended to Euler's Totient function. More abstract approach than Boyer-Moore (which seems necessary to achieve the extended version).

9.1 Definitions and lemmas

```

inductive-set
  RsetR :: int => int set set
  for m :: int
  where
    empty [simp]: {} ∈ RsetR m
  | insert: A ∈ RsetR m ==> zgcd a m = 1 ==>
    ∀ a'. a' ∈ A --> ¬ zcong a a' m ==> insert a A ∈ RsetR m

```

```

fun
  BnorRset :: int ⇒ int => int set
where
  BnorRset a m =
    (if 0 < a then
      let na = BnorRset (a - 1) m
      in (if zgcd a m = 1 then insert a na else na)
    else {})

```

```

definition
  norRRset :: int => int set where
  norRRset m = BnorRset (m - 1) m

```

```

definition
  noXRRset :: int => int => int set where
  noXRRset m x = (λa. a * x) ' norRRset m

```

```

definition

```

```

phi :: int => nat where
phi m = card (norRRset m)

```

definition

```

is-RRset :: int set => int => bool where
is-RRset A m = (A ∈ RsetR m ∧ card A = phi m)

```

definition

```

RRset2norRR :: int set => int => int => int where
RRset2norRR A m a =
  (if 1 < m ∧ is-RRset A m ∧ a ∈ A then
    SOME b. zcong a b m ∧ b ∈ norRRset m
  else 0)

```

definition

```

zcong m :: int => int => int => bool where
zcong m = (λa b. zcong a b m)

```

lemma *abs-eq-1-iff* [iff]: $(\text{abs } z = (1::\text{int})) = (z = 1 \vee z = -1)$
— LCP: not sure why this lemma is needed now
by (*auto simp add: abs-if*)

norRRset

declare *BnorRset.simps* [*simp del*]

lemma *BnorRset-induct*:

```

assumes !!a m. P {} a m
and !!a m :: int. 0 < a ==> P (BnorRset (a - 1) m) (a - 1) m
==> P (BnorRset a m) a m
shows P (BnorRset u v) u v
apply (rule BnorRset.induct)
apply (case-tac 0 < a)
apply (rule-tac assms)
apply simp-all
apply (simp-all add: BnorRset.simps assms)
done

```

lemma *Bnor-mem-zle* [rule-format]: $b \in \text{BnorRset } a \ m \longrightarrow b \leq a$
apply (induct a m rule: BnorRset-induct)
apply *simp*
apply (subst BnorRset.simps)
apply (unfold Let-def, auto)
done

lemma *Bnor-mem-zle-swap*: $a < b ==> b \notin \text{BnorRset } a \ m$
by (*auto dest: Bnor-mem-zle*)

lemma *Bnor-mem-zg* [rule-format]: $b \in \text{BnorRset } a \ m \dashrightarrow 0 < b$
apply (induct a m rule: BnorRset-induct)

```

prefer 2
apply (subst BnorRset.simps)
apply (unfold Let-def, auto)
done

lemma Bnor-mem-if [rule-format]:
  zgcd b m = 1 --> 0 < b --> b ≤ a --> b ∈ BnorRset a m
apply (induct a m rule: BnorRset.induct, auto)
apply (subst BnorRset.simps)
defer
apply (subst BnorRset.simps)
apply (unfold Let-def, auto)
done

lemma Bnor-in-RsetR [rule-format]: a < m --> BnorRset a m ∈ RsetR m
apply (induct a m rule: BnorRset-induct, simp)
apply (subst BnorRset.simps)
apply (unfold Let-def, auto)
apply (rule RsetR.insert)
apply (rule-tac [3] allI)
apply (rule-tac [3] impI)
apply (rule-tac [3] zcong-not)
apply (subgoal-tac [6] a' ≤ a - 1)
apply (rule-tac [7] Bnor-mem-zle)
apply (rule-tac [5] Bnor-mem-zg, auto)
done

lemma Bnor-fin: finite (BnorRset a m)
apply (induct a m rule: BnorRset-induct)
prefer 2
apply (subst BnorRset.simps)
apply (unfold Let-def, auto)
done

lemma norR-mem-unique-aux: a ≤ b - 1 ==> a < (b::int)
apply auto
done

lemma norR-mem-unique:
  1 < m ==>
    zgcd a m = 1 ==> ∃!b. [a = b] (mod m) ∧ b ∈ norRRset m
apply (unfold norRRset-def)
apply (cut-tac a = a and m = m in zcong-zless-unique, auto)
apply (rule-tac [2] m = m in zcong-zless-imp-eq)
apply (auto intro: Bnor-mem-zle Bnor-mem-zg zcong-trans
    order-less-imp-le norR-mem-unique-aux simp add: zcong-sym)
apply (rule-tac x = b in exI, safe)
apply (rule Bnor-mem-if)
apply (case-tac [2] b = 0)

```

```

    apply (auto intro: order-less-le [THEN iffD2])
  prefer 2
  apply (simp only: zcong-def)
  apply (subgoal-tac zgcd a m = m)
  prefer 2
  apply (subst zdvd-iff-zgcd [symmetric])
  apply (rule-tac [4] zgcd-zcong-zgcd)
  apply (simp-all add: zcong-sym)
done

```

noXRRset

```

lemma RRset-gcd [rule-format]:
  is-RRset A m ==> a ∈ A --> zgcd a m = 1
  apply (unfold is-RRset-def)
  apply (rule RsetR.induct [where P=%A. a ∈ A --> zgcd a m = 1], auto)
done

```

```

lemma RsetR-zmult-mono:
  A ∈ RsetR m ==>
    0 < m ==> zgcd x m = 1 ==> (λa. a * x) ' A ∈ RsetR m
  apply (erule RsetR.induct, simp-all)
  apply (rule RsetR.insert, auto)
  apply (blast intro: zgcd-zgcd-zmult)
  apply (simp add: zcong-cancel)
done

```

```

lemma card-nor-eq-noX:
  0 < m ==>
    zgcd x m = 1 ==> card (noXRRset m x) = card (norRRset m)
  apply (unfold norRRset-def noXRRset-def)
  apply (rule card-image)
  apply (auto simp add: inj-on-def Bnor-fin)
  apply (simp add: BnorRset.simps)
done

```

```

lemma noX-is-RRset:
  0 < m ==> zgcd x m = 1 ==> is-RRset (noXRRset m x) m
  apply (unfold is-RRset-def phi-def)
  apply (auto simp add: card-nor-eq-noX)
  apply (unfold noXRRset-def norRRset-def)
  apply (rule RsetR-zmult-mono)
  apply (rule Bnor-in-RsetR, simp-all)
done

```

```

lemma aux-some:
  1 < m ==> is-RRset A m ==> a ∈ A
  ==> zcong a (SOME b. [a = b] (mod m) ∧ b ∈ norRRset m) m ∧
    (SOME b. [a = b] (mod m) ∧ b ∈ norRRset m) ∈ norRRset m
  apply (rule norR-mem-unique [THEN ex1-implies-ex, THEN someI-ex])

```

```

apply (rule-tac [2] RRset-gcd, simp-all)
done

lemma RRset2norRR-correct:
  1 < m ==> is-RRset A m ==> a ∈ A ==>
    [a = RRset2norRR A m a] (mod m) ∧ RRset2norRR A m a ∈ norRRset m
apply (unfold RRset2norRR-def, simp)
apply (rule aux-some, simp-all)
done

lemmas RRset2norRR-correct1 =
  RRset2norRR-correct [THEN conjunct1, standard]
lemmas RRset2norRR-correct2 =
  RRset2norRR-correct [THEN conjunct2, standard]

lemma RsetR-fin: A ∈ RsetR m ==> finite A
by (induct set: RsetR) auto

lemma RRset-zcong-eq [rule-format]:
  1 < m ==>
    is-RRset A m ==> [a = b] (mod m) ==> a ∈ A --> b ∈ A --> a = b
apply (unfold is-RRset-def)
apply (rule RsetR.induct [where P=%A. a ∈ A --> b ∈ A --> a = b])
apply (auto simp add: zcong-sym)
done

lemma aux:
  P (SOME a. P a) ==> Q (SOME a. Q a) ==>
    (SOME a. P a) = (SOME a. Q a) ==> ∃ a. P a ∧ Q a
apply auto
done

lemma RRset2norRR-inj:
  1 < m ==> is-RRset A m ==> inj-on (RRset2norRR A m) A
apply (unfold RRset2norRR-def inj-on-def, auto)
apply (subgoal-tac ∃ b. ([x = b] (mod m) ∧ b ∈ norRRset m) ∧
  ([y = b] (mod m) ∧ b ∈ norRRset m))
apply (rule-tac [2] aux)
apply (rule-tac [3] aux-some)
apply (rule-tac [2] aux-some)
apply (rule RRset-zcong-eq, auto)
apply (rule-tac b = b in zcong-trans)
apply (simp-all add: zcong-sym)
done

lemma RRset2norRR-eq-norR:
  1 < m ==> is-RRset A m ==> RRset2norRR A m ' A = norRRset m
apply (rule card-seteq)
prefer 3

```

```

apply (subst card-image)
apply (rule-tac RRset2norRR-inj, auto)
apply (rule-tac [3] RRset2norRR-correct2, auto)
apply (unfold is-RRset-def phi-def norRRset-def)
apply (auto simp add: Bnor-fin)
done

```

lemma *Bnor-prod-power-aux*: $a \notin A \implies \text{inj } f \implies f a \notin f^{-1} A$
by (unfold inj-on-def, auto)

lemma *Bnor-prod-power* [rule-format]:
 $x \neq 0 \implies a < m \longrightarrow \prod ((\lambda a. a * x)^{-1} \text{BnorRset } a \ m) =$
 $\prod (\text{BnorRset } a \ m) * x^{\text{card } (\text{BnorRset } a \ m)}$
apply (induct a m rule: BnorRset-induct)
prefer 2
apply (simplesubst BnorRset.simps) — multiple redexes
apply (unfold Let-def, auto)
apply (simp add: Bnor-fin Bnor-mem-zle-swap)
apply (subst setprod-insert)
apply (rule-tac [2] Bnor-prod-power-aux)
apply (unfold inj-on-def)
apply (simp-all add: zmult-ac Bnor-fin finite-imageI
Bnor-mem-zle-swap)
done

9.2 Fermat

lemma *bijzcong-zcong-prod*:
 $(A, B) \in \text{bijR } (\text{zcong } m) \implies [\prod A = \prod B] \pmod{m}$
apply (unfold zcong-def)
apply (erule bijR.induct)
apply (subgoal-tac [2] $a \notin A \wedge b \notin B \wedge \text{finite } A \wedge \text{finite } B$)
apply (auto intro: fin-bijRl fin-bijRr zcong-zmult)
done

lemma *Bnor-prod-zgcd* [rule-format]:
 $a < m \longrightarrow \text{zgcd } (\prod (\text{BnorRset } a \ m)) \ m = 1$
apply (induct a m rule: BnorRset-induct)
prefer 2
apply (subst BnorRset.simps)
apply (unfold Let-def, auto)
apply (simp add: Bnor-fin Bnor-mem-zle-swap)
apply (blast intro: zgcd-zgcd-zmult)
done

theorem *Euler-Fermat*:
 $0 < m \implies \text{zgcd } x \ m = 1 \implies [x^{\text{phi } m} = 1] \pmod{m}$
apply (unfold norRRset-def phi-def)


```

apply (case-tac  $x = 0$ )
apply (case-tac [2]  $m = 1$ )
apply (rule-tac [3] iffD1)
apply (rule-tac [3]  $k = \prod (BnorRset (m - 1) m)$ 
  in zcong-cancel2)
prefer 5
apply (subst Bnor-prod-power [symmetric])
apply (rule-tac [7] Bnor-prod-zgcd, simp-all)
apply (rule bijzcong-zcong-prod)
apply (fold norRRset-def, fold noXRRset-def)
apply (subst RRset2norRR-eq-norR [symmetric])
apply (rule-tac [3] inj-func-bijR, auto)
apply (unfold zcongm-def)
apply (rule-tac [2] RRset2norRR-correct1)
apply (rule-tac [5] RRset2norRR-inj)
apply (auto intro: order-less-le [THEN iffD2]
  simp add: noX-is-RRset)
apply (unfold noXRRset-def norRRset-def)
apply (rule finite-imageI)
apply (rule Bnor-fin)
done

```

lemma *Bnor-prime*:

```

 $\llbracket \text{zprime } p; a < p \rrbracket \implies \text{card } (BnorRset a p) = \text{nat } a$ 
apply (induct  $a$   $p$  rule: BnorRset.induct)
apply (subst BnorRset.simps)
apply (unfold Let-def, auto simp add: zless-zprime-imp-zrelprime)
apply (subgoal-tac finite ( $BnorRset (a - 1) m$ ))
apply (subgoal-tac  $a \sim: BnorRset (a - 1) m$ )
apply (auto simp add: card-insert-disjoint Suc-nat-eq-nat-zadd1)
apply (frule Bnor-mem-zle, arith)
apply (frule Bnor-fin)
done

```

lemma *phi-prime*: $\text{zprime } p \implies \text{phi } p = \text{nat } (p - 1)$

```

apply (unfold phi-def norRRset-def)
apply (rule Bnor-prime, auto)
done

```

theorem *Little-Fermat*:

```

 $\text{zprime } p \implies \neg p \text{ dvd } x \implies [x^{(\text{nat } (p - 1))} = 1] \pmod{p}$ 
apply (subst phi-prime [symmetric])
apply (rule-tac [2] Euler-Fermat)
apply (erule-tac [3] zprime-imp-zrelprime)
apply (unfold zprime-def, auto)
done

```

end

10 Wilson's Theorem according to Russinoff

theory *WilsonRuss* **imports** *EulerFermat* **begin**

Wilson's Theorem following quite closely Russinoff's approach using Boyer-Moore (using finite sets instead of lists, though).

10.1 Definitions and lemmas

definition

inv :: *int* => *int* => *int* **where**
inv *p* *a* = (*a*^(nat (*p* - 2))) mod *p*

fun

wset :: *int* => *int* => *int* set

where

wset *a* *p* =
 (if 1 < *a* then
 let *ws* = *wset* (*a* - 1) *p*
 in (if *a* ∈ *ws* then *ws* else insert *a* (insert (*inv* *p* *a*) *ws*)) else {})

inv

lemma *inv-is-inv-aux*: 1 < *m* ==> Suc (nat (*m* - 2)) = nat (*m* - 1)
by (subst *int-int-eq* [*symmetric*], *auto*)

lemma *inv-is-inv*:

zprime *p* ==> 0 < *a* ==> *a* < *p* ==> [*a* * *inv* *p* *a* = 1] (mod *p*)
apply (*unfold inv-def*)
apply (*subst zcong-zmod*)
apply (*subst zmod-zmult1-eq* [*symmetric*])
apply (*subst zcong-zmod* [*symmetric*])
apply (*subst power-Suc* [*symmetric*])
apply (*subst inv-is-inv-aux*)
apply (*erule-tac* [2] *Little-Fermat*)
apply (*erule-tac* [2] *zdvd-not-zless*)
apply (*unfold zprime-def*, *auto*)
done

lemma *inv-distinct*:

zprime *p* ==> 1 < *a* ==> *a* < *p* - 1 ==> *a* ≠ *inv* *p* *a*
apply *safe*
apply (*cut-tac* *a* = *a* **and** *p* = *p* **in** *zcong-square*)
apply (*cut-tac* [3] *a* = *a* **and** *p* = *p* **in** *inv-is-inv*, *auto*)
apply (*subgoal-tac* *a* = 1)
apply (*rule-tac* [2] *m* = *p* **in** *zcong-zless-imp-eq*)
apply (*subgoal-tac* [7] *a* = *p* - 1)
apply (*rule-tac* [8] *m* = *p* **in** *zcong-zless-imp-eq*, *auto*)
done

```

lemma inv-not-0:
   $zprime\ p \implies 1 < a \implies a < p - 1 \implies inv\ p\ a \neq 0$ 
  apply safe
  apply (cut-tac  $a = a$  and  $p = p$  in inv-is-inv)
    apply (unfold zcong-def, auto)
  apply (subgoal-tac  $\neg p\ dvd\ 1$ )
  apply (rule-tac [2] zdvd-not-zless)
  apply (subgoal-tac  $p\ dvd\ 1$ )
  prefer 2
  apply (subst dvd-minus-iff [symmetric], auto)
done

lemma inv-not-1:
   $zprime\ p \implies 1 < a \implies a < p - 1 \implies inv\ p\ a \neq 1$ 
  apply safe
  apply (cut-tac  $a = a$  and  $p = p$  in inv-is-inv)
    prefer 4
    apply simp
  apply (subgoal-tac  $a = 1$ )
  apply (rule-tac [2] zcong-zless-imp-eq, auto)
done

lemma inv-not-p-minus-1-aux:
   $[a * (p - 1) = 1] \pmod{p} = [a = p - 1] \pmod{p}$ 
  apply (unfold zcong-def)
  apply (simp add: diff-diff-eq diff-diff-eq2 zdiff-zmult-distrib2)
  apply (rule-tac  $s = p\ dvd\ -((a + 1) + (p * -a))$  in trans)
    apply (simp add: algebra-simps)
  apply (subst dvd-minus-iff)
  apply (subst zdvd-reduce)
  apply (rule-tac  $s = p\ dvd\ (a + 1) + (p * -1)$  in trans)
    apply (subst zdvd-reduce, auto)
done

lemma inv-not-p-minus-1:
   $zprime\ p \implies 1 < a \implies a < p - 1 \implies inv\ p\ a \neq p - 1$ 
  apply safe
  apply (cut-tac  $a = a$  and  $p = p$  in inv-is-inv, auto)
  apply (simp add: inv-not-p-minus-1-aux)
  apply (subgoal-tac  $a = p - 1$ )
  apply (rule-tac [2] zcong-zless-imp-eq, auto)
done

lemma inv-g-1:
   $zprime\ p \implies 1 < a \implies a < p - 1 \implies 1 < inv\ p\ a$ 
  apply (case-tac  $0 \leq inv\ p\ a$ )
  apply (subgoal-tac  $inv\ p\ a \neq 1$ )
  apply (subgoal-tac  $inv\ p\ a \neq 0$ )
  apply (subst order-less-le)

```

```

    apply (subst zle-add1-eq-le [symmetric])
    apply (subst order-less-le)
    apply (rule-tac [2] inv-not-0)
    apply (rule-tac [5] inv-not-1, auto)
  apply (unfold inv-def zprime-def, simp)
done

lemma inv-less-p-minus-1:
  zprime p  $\implies$   $1 < a \implies a < p - 1 \implies \text{inv } p \ a < p - 1$ 
  apply (case-tac inv p a < p)
  apply (subst order-less-le)
  apply (simp add: inv-not-p-minus-1, auto)
  apply (unfold inv-def zprime-def, simp)
done

lemma inv-inv-aux:  $5 \leq p \implies$ 
  nat (p - 2) * nat (p - 2) = Suc (nat (p - 1) * nat (p - 3))
  apply (subst int-int-eq [symmetric])
  apply (simp add: zmult-int [symmetric])
  apply (simp add: zdif-zmult-distrib zdif-zmult-distrib2)
done

lemma zcong-zpower-zmult:
   $[x^y = 1] \pmod p \implies [x^{(y * z)} = 1] \pmod p$ 
  apply (induct z)
  apply (auto simp add: zpower-zadd-distrib)
  apply (subgoal-tac zcong (x^y * x^{(y * z)}) (1 * 1) p)
  apply (rule-tac [2] zcong-zmult, simp-all)
done

lemma inv-inv: zprime p  $\implies$ 
   $5 \leq p \implies 0 < a \implies a < p \implies \text{inv } p (\text{inv } p \ a) = a$ 
  apply (unfold inv-def)
  apply (subst zpower-zmod)
  apply (subst zpower-zpower)
  apply (rule zcong-zless-imp-eq)
  prefer 5
  apply (subst zcong-zmod)
  apply (subst mod-mod-trivial)
  apply (subst zcong-zmod [symmetric])
  apply (subst inv-inv-aux)
  apply (subgoal-tac [2]
    zcong (a * a^{(nat (p - 1) * nat (p - 3))}) (a * 1) p)
  apply (rule-tac [3] zcong-zmult)
  apply (rule-tac [4] zcong-zpower-zmult)
  apply (erule-tac [4] Little-Fermat)
  apply (rule-tac [4] zdvd-not-zless, simp-all)
done

```

wset

declare *wset.simps* [*simp del*]

lemma *wset-induct*:

assumes $!!a\ p.\ P\ \{\}\ a\ p$
and $!!a\ p.\ 1 < (a::int) \implies$
 $P\ (wset\ (a - 1)\ p)\ (a - 1)\ p \implies P\ (wset\ a\ p)\ a\ p$
shows $P\ (wset\ u\ v)\ u\ v$
apply (rule *wset.induct*)
apply (case-tac $1 < a$)
apply (rule *assms*)
apply (simp-all add: *wset.simps assms*)
done

lemma *wset-mem-imp-or* [*rule-format*]:

$1 < a \implies b \notin wset\ (a - 1)\ p$
 $\implies b \in wset\ a\ p \dashv\vdash b = a \vee b = inv\ p\ a$
apply (subst *wset.simps*)
apply (unfold *Let-def*, *simp*)
done

lemma *wset-mem-mem* [*simp*]: $1 < a \implies a \in wset\ a\ p$

apply (subst *wset.simps*)
apply (unfold *Let-def*, *simp*)
done

lemma *wset-subset*: $1 < a \implies b \in wset\ (a - 1)\ p \implies b \in wset\ a\ p$

apply (subst *wset.simps*)
apply (unfold *Let-def*, *auto*)
done

lemma *wset-g-1* [*rule-format*]:

$zprime\ p \dashv\vdash a < p - 1 \dashv\vdash b \in wset\ a\ p \dashv\vdash 1 < b$
apply (induct *a p* rule: *wset-induct*, *auto*)
apply (case-tac $b = a$)
apply (case-tac [2] $b = inv\ p\ a$)
apply (subgoal-tac [3] $b = a \vee b = inv\ p\ a$)
apply (rule-tac [4] *wset-mem-imp-or*)
prefer 2
apply *simp*
apply (rule *inv-g-1*, *auto*)
done

lemma *wset-less* [*rule-format*]:

$zprime\ p \dashv\vdash a < p - 1 \dashv\vdash b \in wset\ a\ p \dashv\vdash b < p - 1$
apply (induct *a p* rule: *wset-induct*, *auto*)
apply (case-tac $b = a$)
apply (case-tac [2] $b = inv\ p\ a$)
apply (subgoal-tac [3] $b = a \vee b = inv\ p\ a$)

```

    apply (rule-tac [4] wset-mem-imp-or)
    prefer 2
    apply simp
    apply (rule inv-less-p-minus-1, auto)
done

lemma wset-mem [rule-format]:
  zprime p -->
    a < p - 1 --> 1 < b --> b ≤ a --> b ∈ wset a p
  apply (induct a p rule: wset.induct, auto)
  apply (rule-tac wset-subset)
  apply (simp (no-asm-simp))
  apply auto
done

lemma wset-mem-inv-mem [rule-format]:
  zprime p --> 5 ≤ p --> a < p - 1 --> b ∈ wset a p
    --> inv p b ∈ wset a p
  apply (induct a p rule: wset-induct, auto)
  apply (case-tac b = a)
  apply (subst wset.simps)
  apply (unfold Let-def)
  apply (rule-tac [3] wset-subset, auto)
  apply (case-tac b = inv p a)
  apply (simp (no-asm-simp))
  apply (subst inv-inv)
    apply (subgoal-tac [6] b = a ∨ b = inv p a)
    apply (rule-tac [7] wset-mem-imp-or, auto)
done

lemma wset-inv-mem-mem:
  zprime p ==> 5 ≤ p ==> a < p - 1 ==> 1 < b ==> b < p - 1
    ==> inv p b ∈ wset a p ==> b ∈ wset a p
  apply (rule-tac s = inv p (inv p b) and t = b in subst)
  apply (rule-tac [2] wset-mem-inv-mem)
    apply (rule inv-inv, simp-all)
done

lemma wset-fin: finite (wset a p)
  apply (induct a p rule: wset-induct)
  prefer 2
  apply (subst wset.simps)
  apply (unfold Let-def, auto)
done

lemma wset-zcong-prod-1 [rule-format]:
  zprime p -->
    5 ≤ p --> a < p - 1 --> [(∏ x ∈ wset a p. x) = 1] (mod p)
  apply (induct a p rule: wset-induct)

```

```

prefer 2
apply (subst wset.simps)
apply (auto, unfold Let-def, auto)
apply (subst setprod-insert)
  apply (tactic << stac (thm setprod-insert) 3 >>)
    apply (subgoal-tac [5]
      zcong (a * inv p a * ( $\prod_{x \in wset (a - 1) p} x$ )) (1 * 1) p)
      prefer 5
      apply (simp add: zmult-assoc)
      apply (rule-tac [5] zcong-zmult)
      apply (rule-tac [5] inv-is-inv)
      apply (tactic clarify-tac @ {claset} 4)
      apply (subgoal-tac [4] a  $\in$  wset (a - 1) p)
      apply (rule-tac [5] wset-inv-mem-mem)
      apply (simp-all add: wset-fin)
    apply (rule inv-distinct, auto)
done

```

```

lemma d22set-eq-wset: zprime p ==> d22set (p - 2) = wset (p - 2) p
apply safe
  apply (erule wset-mem)
    apply (rule-tac [2] d22set-g-1)
    apply (rule-tac [3] d22set-le)
    apply (rule-tac [4] d22set-mem)
    apply (erule-tac [4] wset-g-1)
    prefer 6
    apply (subst zle-add1-eq-le [symmetric])
    apply (subgoal-tac p - 2 + 1 = p - 1)
    apply (simp (no-asm-simp))
    apply (erule wset-less, auto)
  done

```

10.2 Wilson

```

lemma prime-g-5: zprime p ==> p  $\neq$  2 ==> p  $\neq$  3 ==> 5  $\leq$  p
apply (unfold zprime-def dvd-def)
apply (case-tac p = 4, auto)
apply (rule notE)
  prefer 2
  apply assumption
  apply (simp (no-asm))
  apply (rule-tac x = 2 in exI)
  apply (safe, arith)
  apply (rule-tac x = 2 in exI, auto)
done

```

theorem *Wilson-Russ:*

```

  zprime p ==> [zfact (p - 1) = -1] (mod p)
  apply (subgoal-tac [(p - 1) * zfact (p - 2) = -1 * 1] (mod p))

```

```

apply (rule-tac [2] zcong-zmult)
apply (simp only: zprime-def)
apply (subst zfact.simps)
apply (rule-tac t = p - 1 - 1 and s = p - 2 in subst, auto)
apply (simp only: zcong-def)
apply (simp (no-asm-simp))
apply (case-tac p = 2)
apply (simp add: zfact.simps)
apply (case-tac p = 3)
apply (simp add: zfact.simps)
apply (subgoal-tac 5 ≤ p)
apply (erule-tac [2] prime-g-5)
apply (subst d22set-prod-zfact [symmetric])
apply (subst d22set-eq-wset)
apply (rule-tac [2] wset-zcong-prod-1, auto)
done

end

```

11 Wilson’s Theorem using a more abstract approach

theory *WilsonBij* **imports** *BijectionRel IntFact* **begin**

Wilson’s Theorem using a more “abstract” approach based on bijections between sets. Does not use Fermat’s Little Theorem (unlike Russinoff).

11.1 Definitions and lemmas

definition

```

reciR :: int => int => int => bool where
reciR p = (λa b. zcong (a * b) 1 p ∧ 1 < a ∧ a < p - 1 ∧ 1 < b ∧ b < p - 1)

```

definition

```

inv :: int => int => int where
inv p a =
  (if zprime p ∧ 0 < a ∧ a < p then
    (SOME x. 0 ≤ x ∧ x < p ∧ zcong (a * x) 1 p)
  else 0)

```

Inverse

lemma *inv-correct*:

```

zprime p ==> 0 < a ==> a < p
==> 0 ≤ inv p a ∧ inv p a < p ∧ [a * inv p a = 1] (mod p)
apply (unfold inv-def)
apply (simp (no-asm-simp))

```



```

apply (rule zcong-lineq-unique [THEN ex1-implies-ex, THEN someI-ex])
apply (erule-tac [2] zless-zprime-imp-zrelprime)
apply (unfold zprime-def)
apply auto
done

lemmas inv-ge = inv-correct [THEN conjunct1, standard]
lemmas inv-less = inv-correct [THEN conjunct2, THEN conjunct1, standard]
lemmas inv-is-inv = inv-correct [THEN conjunct2, THEN conjunct2, standard]

lemma inv-not-0:
  zprime p ==> 1 < a ==> a < p - 1 ==> inv p a ≠ 0
  — same as WilsonRuss
apply safe
apply (cut-tac a = a and p = p in inv-is-inv)
apply (unfold zcong-def)
apply auto
apply (subgoal-tac ¬ p dvd 1)
apply (rule-tac [2] zdvd-not-zless)
apply (subgoal-tac p dvd 1)
prefer 2
apply (subst dvd-minus-iff [symmetric])
apply auto
done

lemma inv-not-1:
  zprime p ==> 1 < a ==> a < p - 1 ==> inv p a ≠ 1
  — same as WilsonRuss
apply safe
apply (cut-tac a = a and p = p in inv-is-inv)
prefer 4
apply simp
apply (subgoal-tac a = 1)
apply (rule-tac [2] zcong-zless-imp-eq)
apply auto
done

lemma aux: [a * (p - 1) = 1] (mod p) = [a = p - 1] (mod p)
  — same as WilsonRuss
apply (unfold zcong-def)
apply (simp add: diff-diff-eq diff-diff-eq2 zdiff-zmult-distrib2)
apply (rule-tac s = p dvd -((a + 1) + (p * -a)) in trans)
apply (simp add: algebra-simps)
apply (subst dvd-minus-iff)
apply (subst zdvd-reduce)
apply (rule-tac s = p dvd (a + 1) + (p * -1) in trans)
apply (subst zdvd-reduce)
apply auto
done

```

```

lemma inv-not-p-minus-1:
  zprime p ==> 1 < a ==> a < p - 1 ==> inv p a ≠ p - 1
  — same as WilsonRuss
  apply safe
  apply (cut-tac a = a and p = p in inv-is-inv)
    apply auto
  apply (simp add: aux)
  apply (subgoal-tac a = p - 1)
  apply (rule-tac [2] zcong-zless-imp-eq)
    apply auto
  done

```

Below is slightly different as we don't expand *inv* but use “*correct*” theorems.

```

lemma inv-g-1: zprime p ==> 1 < a ==> a < p - 1 ==> 1 < inv p a
  apply (subgoal-tac inv p a ≠ 1)
  apply (subgoal-tac inv p a ≠ 0)
  apply (subst order-less-le)
  apply (subst zle-add1-eq-le [symmetric])
  apply (subst order-less-le)
  apply (rule-tac [2] inv-not-0)
  apply (rule-tac [5] inv-not-1)
    apply auto
  apply (rule inv-ge)
  apply auto
  done

```

```

lemma inv-less-p-minus-1:
  zprime p ==> 1 < a ==> a < p - 1 ==> inv p a < p - 1
  — ditto
  apply (subst order-less-le)
  apply (simp add: inv-not-p-minus-1 inv-less)
  done

```

Bijection

```

lemma aux1: 1 < x ==> 0 ≤ (x::int)
  apply auto
  done

```

```

lemma aux2: 1 < x ==> 0 < (x::int)
  apply auto
  done

```

```

lemma aux3: x ≤ p - 2 ==> x < (p::int)
  apply auto
  done

```

```

lemma aux4: x ≤ p - 2 ==> x < (p::int) - 1
  apply auto

```

```

done

lemma inv-inj: zprime p ==> inj-on (inv p) (d22set (p - 2))
  apply (unfold inj-on-def)
  apply auto
  apply (rule zcong-zless-imp-eq)
    apply (tactic << stac (thm zcong-cancel RS sym) 5 >>)
    apply (rule-tac [7] zcong-trans)
    apply (tactic << stac (thm zcong-sym) 8 >>)
    apply (erule-tac [7] inv-is-inv)
    apply (tactic asm-simp-tac @{simpset} 9)
    apply (erule-tac [9] inv-is-inv)
    apply (rule-tac [6] zless-zprime-imp-zrelprime)
    apply (rule-tac [8] inv-less)
    apply (rule-tac [7] inv-g-1 [THEN aux2])
    apply (unfold zprime-def)
    apply (auto intro: d22set-g-1 d22set-le
      aux1 aux2 aux3 aux4)
done

lemma inv-d22set-d22set:
  zprime p ==> inv p ' d22set (p - 2) = d22set (p - 2)
  apply (rule endo-inj-surj)
  apply (rule d22set-fin)
  apply (erule-tac [2] inv-inj)
  apply auto
  apply (rule d22set-mem)
  apply (erule inv-g-1)
  apply (subgoal-tac [3] inv p xa < p - 1)
  apply (erule-tac [4] inv-less-p-minus-1)
  apply (auto intro: d22set-g-1 d22set-le aux4)
done

lemma d22set-d22set-bij:
  zprime p ==> (d22set (p - 2), d22set (p - 2)) ∈ bijR (reciR p)
  apply (unfold reciR-def)
  apply (rule-tac s = (d22set (p - 2), inv p ' d22set (p - 2)) in subst)
  apply (simp add: inv-d22set-d22set)
  apply (rule inj-func-bijR)
  apply (rule-tac [3] d22set-fin)
  apply (erule-tac [2] inv-inj)
  apply auto
  apply (erule inv-is-inv)
  apply (erule-tac [5] inv-g-1)
  apply (erule-tac [7] inv-less-p-minus-1)
  apply (auto intro: d22set-g-1 d22set-le aux2 aux3 aux4)
done

lemma reciP-bijP: zprime p ==> bijP (reciR p) (d22set (p - 2))

```

```

apply (unfold reciR-def bijP-def)
apply auto
apply (rule d22set-mem)
apply auto
done

lemma reciP-uniq:  $zprime\ p \implies uniqP\ (reciR\ p)$ 
apply (unfold reciR-def uniqP-def)
apply auto
apply (rule zcong-zless-imp-eq)
  apply (tactic << stac (thm zcong-cancel2 RS sym) 5 >>)
  apply (rule-tac [7] zcong-trans)
  apply (tactic << stac (thm zcong-sym) 8 >>)
  apply (rule-tac [6] zless-zprime-imp-zrelprime)
  apply auto
apply (rule zcong-zless-imp-eq)
  apply (tactic << stac (thm zcong-cancel RS sym) 5 >>)
  apply (rule-tac [7] zcong-trans)
  apply (tactic << stac (thm zcong-sym) 8 >>)
  apply (rule-tac [6] zless-zprime-imp-zrelprime)
  apply auto
done

lemma reciP-sym:  $zprime\ p \implies symP\ (reciR\ p)$ 
apply (unfold reciR-def symP-def)
apply (simp add: zmult-commute)
apply auto
done

lemma bijER-d22set:  $zprime\ p \implies d22set\ (p - 2) \in bijER\ (reciR\ p)$ 
apply (rule bijR-bijER)
  apply (erule d22set-d22set-bij)
  apply (erule reciP-bijP)
  apply (erule reciP-uniq)
  apply (erule reciP-sym)
done

```

11.2 Wilson

```

lemma bijER-zcong-prod-1:
   $zprime\ p \implies A \in bijER\ (reciR\ p) \implies [\prod A = 1] \pmod{p}$ 
apply (unfold reciR-def)
apply (erule bijER.induct)
  apply (subgoal-tac [2]  $a = 1 \vee a = p - 1$ )
  apply (rule-tac [3] zcong-square-zless)
  apply auto
apply (subst setprod-insert)
prefer 3
apply (subst setprod-insert)

```

```

    apply (auto simp add: fin-bijER)
  apply (subgoal-tac zcong ((a * b) *  $\prod A$ ) (1 * 1) p)
  apply (simp add: zmult-assoc)
  apply (rule zcong-zmult)
  apply auto
done

theorem Wilson-Bij: zprime p ==> [zfact (p - 1) = -1] (mod p)
  apply (subgoal-tac zcong ((p - 1) * zfact (p - 2)) (-1 * 1) p)
  apply (rule-tac [2] zcong-zmult)
  apply (simp add: zprime-def)
  apply (subst zfact.simps)
  apply (rule-tac t = p - 1 - 1 and s = p - 2 in subst)
  apply auto
  apply (simp add: zcong-def)
  apply (subst d22set-prod-zfact [symmetric])
  apply (rule bijER-zcong-prod-1)
  apply (rule-tac [2] bijER-d22set)
  apply auto
done

end

```

12 Finite Sets and Finite Sums

```

theory Finite2
imports Main IntFact Infinite-Set
begin

```

These are useful for combinatorial and number-theoretic counting arguments.

12.1 Useful properties of sums and products

```

lemma setsum-same-function-zcong:
  assumes a:  $\forall x \in S. [f x = g x](\text{mod } m)$ 
  shows [setsum f S = setsum g S] (mod m)
proof cases
  assume finite S
  thus ?thesis using a by induct (simp-all add: zcong-zadd)
next
  assume infinite S thus ?thesis by (simp add: setsum-def)
qed

lemma setprod-same-function-zcong:
  assumes a:  $\forall x \in S. [f x = g x](\text{mod } m)$ 
  shows [setprod f S = setprod g S] (mod m)
proof cases

```

```

    assume finite S
    thus ?thesis using a by induct (simp-all add: zcong-zmult)
next
    assume infinite S thus ?thesis by (simp add: setprod-def)
qed

lemma setsum-const: finite X ==> setsum (%x. (c :: int)) X = c * int(card X)
  apply (induct set: finite)
  apply (auto simp add: left-distrib right-distrib int-eq-of-nat)
  done

lemma setsum-const2: finite X ==> int (setsum (%x. (c :: nat)) X) =
  int(c) * int(card X)
  apply (induct set: finite)
  apply (auto simp add: zadd-zmult-distrib2)
  done

lemma setsum-const-mult: finite A ==> setsum (%x. c * ((f x)::int)) A =
  c * setsum f A
  by (induct set: finite) (auto simp add: zadd-zmult-distrib2)



## 12.2 Cardinality of explicit finite sets



lemma finite-surjI:  $[[ B \subseteq f \text{ ` } A; \text{ finite } A ]] \implies \text{finite } B$ 
  by (simp add: finite-subset finite-imageI)

lemma bdd-nat-set-l-finite: finite {y::nat . y < x}
  by (rule bounded-nat-set-is-finite) blast

lemma bdd-nat-set-le-finite: finite {y::nat . y ≤ x}
proof -
  have  $\{y::nat . y \leq x\} = \{y::nat . y < \text{Suc } x\}$  by auto
  then show ?thesis by (auto simp add: bdd-nat-set-l-finite)
qed

lemma bdd-int-set-l-finite: finite {x::int. 0 ≤ x & x < n}
  apply (subgoal-tac  $\{(x :: int). 0 \leq x \ \& \ x < n\} \subseteq$ 
    int ` {(x :: nat). x < nat n})
  apply (erule finite-surjI)
  apply (auto simp add: bdd-nat-set-l-finite image-def)
  apply (rule-tac  $x = \text{nat } x$  in exI, simp)
  done

lemma bdd-int-set-le-finite: finite {x::int. 0 ≤ x & x ≤ n}
  apply (subgoal-tac  $\{x. 0 \leq x \ \& \ x \leq n\} = \{x. 0 \leq x \ \& \ x < n + 1\}$ )
  apply (erule ssubst)
  apply (rule bdd-int-set-l-finite)
  apply auto
  done

```

```

lemma bdd-int-set-l-l-finite: finite {x::int.  $0 < x \ \& \ x < n$ }
proof -
  have {x::int.  $0 < x \ \& \ x < n$ }  $\subseteq$  {x::int.  $0 \leq x \ \& \ x < n$ }
    by auto
  then show ?thesis by (auto simp add: bdd-int-set-l-finite finite-subset)
qed

lemma bdd-int-set-l-le-finite: finite {x::int.  $0 < x \ \& \ x \leq n$ }
proof -
  have {x::int.  $0 < x \ \& \ x \leq n$ }  $\subseteq$  {x::int.  $0 \leq x \ \& \ x \leq n$ }
    by auto
  then show ?thesis by (auto simp add: bdd-int-set-le-finite finite-subset)
qed

lemma card-bdd-nat-set-l: card {y::nat .  $y < x$ } = x
proof (induct x)
  case 0
    show card {y::nat .  $y < 0$ } = 0 by simp
  next
    case (Suc n)
      have {y.  $y < \text{Suc } n$ } = insert n {y.  $y < n$ }
        by auto
      then have card {y.  $y < \text{Suc } n$ } = card (insert n {y.  $y < n$ })
        by auto
      also have ... = Suc (card {y.  $y < n$ })
        by (rule card-insert-disjoint) (auto simp add: bdd-nat-set-l-finite)
      finally show card {y.  $y < \text{Suc } n$ } = Suc n
        using {card {y.  $y < n$ } = n} by simp
qed

lemma card-bdd-nat-set-le: card { y::nat.  $y \leq x$ } = Suc x
proof -
  have {y::nat.  $y \leq x$ } = { y::nat.  $y < \text{Suc } x$ }
    by auto
  then show ?thesis by (auto simp add: card-bdd-nat-set-l)
qed

lemma card-bdd-int-set-l:  $0 \leq (n::\text{int}) \implies \text{card } \{y. 0 \leq y \ \& \ y < n\} = \text{nat } n$ 
proof -
  assume  $0 \leq n$ 
  have inj-on (%y. int y) {y.  $y < \text{nat } n$ }
    by (auto simp add: inj-on-def)
  hence card (int ‘ {y.  $y < \text{nat } n$ }) = card {y.  $y < \text{nat } n$ }
    by (rule card-image)
  also from { $0 \leq n$ } have int ‘ {y.  $y < \text{nat } n$ } = {y.  $0 \leq y \ \& \ y < n$ }
    apply (auto simp add: zless-nat-eq-int-zless image-def)
    apply (rule-tac x = nat x in exI)
    apply (auto simp add: nat-0-le)

```

done
also have $\text{card } \{y. y < \text{nat } n\} = \text{nat } n$
by (rule card-bdd-nat-set-l)
finally show $\text{card } \{y. 0 \leq y \ \& \ y < n\} = \text{nat } n$.
qed

lemma card-bdd-int-set-le: $0 \leq (n::\text{int}) \implies \text{card } \{y. 0 \leq y \ \& \ y \leq n\} = \text{nat } n + 1$
proof –
assume $0 \leq n$
moreover have $\{y. 0 \leq y \ \& \ y \leq n\} = \{y. 0 \leq y \ \& \ y < n+1\}$ **by** auto
ultimately show ?thesis
using card-bdd-int-set-l [of $n + 1$]
by (auto simp add: nat-add-distrib)
qed

lemma card-bdd-int-set-l-le: $0 \leq (n::\text{int}) \implies \text{card } \{x. 0 < x \ \& \ x \leq n\} = \text{nat } n$
proof –
assume $0 \leq n$
have inj-on $(\%x. x+1)$ $\{x. 0 \leq x \ \& \ x < n\}$
by (auto simp add: inj-on-def)
hence $\text{card } ((\%x. x+1) \text{ ` } \{x. 0 \leq x \ \& \ x < n\}) = \text{card } \{x. 0 \leq x \ \& \ x < n\}$
by (rule card-image)
also from $\langle 0 \leq n \rangle$ **have** $\dots = \text{nat } n$
by (rule card-bdd-int-set-l)
also have $(\%x. x + 1) \text{ ` } \{x. 0 \leq x \ \& \ x < n\} = \{x. 0 < x \ \& \ x \leq n\}$
apply (auto simp add: image-def)
apply (rule-tac $x = x - 1$ in exI)
apply arith
done
finally show $\text{card } \{x. 0 < x \ \& \ x \leq n\} = \text{nat } n$.
qed

lemma card-bdd-int-set-l-l: $0 < (n::\text{int}) \implies \text{card } \{x. 0 < x \ \& \ x < n\} = \text{nat } n - 1$
proof –
assume $0 < n$
moreover have $\{x. 0 < x \ \& \ x < n\} = \{x. 0 < x \ \& \ x \leq n - 1\}$
by simp
ultimately show ?thesis
using insert card-bdd-int-set-l-le [of $n - 1$]
by (auto simp add: nat-diff-distrib)
qed

lemma int-card-bdd-int-set-l-l: $0 < n \implies \text{int}(\text{card } \{x. 0 < x \ \& \ x < n\}) = n - 1$
apply (auto simp add: card-bdd-int-set-l-l)

done

lemma *int-card-bdd-int-set-l-le*: $0 \leq n \implies$
 $\text{int}(\text{card } \{x. 0 < x \ \& \ x \leq n\}) = n$
by (*auto simp add: card-bdd-int-set-l-le*)

12.3 Cardinality of finite cartesian products

Lemmas for counting arguments.

lemma *setsum-bij-eq*: $[| \text{finite } A; \text{finite } B; f ' A \subseteq B; \text{inj-on } f \ A;$
 $g ' B \subseteq A; \text{inj-on } g \ B |] \implies \text{setsum } g \ B = \text{setsum } (g \circ f) \ A$
apply (*frule-tac h = g and f = f in setsum-reindex*)
apply (*subgoal-tac setsum g B = setsum g (f ' A)*)
apply (*simp add: inj-on-def*)
apply (*subgoal-tac card A = card B*)
apply (*drule-tac A = f ' A and B = B in card-seteq*)
apply (*auto simp add: card-image*)
apply (*frule-tac A = A and B = B and f = f in card-inj-on-le, auto*)
apply (*frule-tac A = B and B = A and f = g in card-inj-on-le*)
apply *auto*
done

lemma *setprod-bij-eq*: $[| \text{finite } A; \text{finite } B; f ' A \subseteq B; \text{inj-on } f \ A;$
 $g ' B \subseteq A; \text{inj-on } g \ B |] \implies \text{setprod } g \ B = \text{setprod } (g \circ f) \ A$
apply (*frule-tac h = g and f = f in setprod-reindex*)
apply (*subgoal-tac setprod g B = setprod g (f ' A)*)
apply (*simp add: inj-on-def*)
apply (*subgoal-tac card A = card B*)
apply (*drule-tac A = f ' A and B = B in card-seteq*)
apply (*auto simp add: card-image*)
apply (*frule-tac A = A and B = B and f = f in card-inj-on-le, auto*)
apply (*frule-tac A = B and B = A and f = g in card-inj-on-le, auto*)
done

end

13 Integers: Divisibility and Congruences

theory *Int2*
imports *Finite2 WilsonRuss*
begin

definition
 $\text{MultInv} :: \text{int} \implies \text{int} \implies \text{int}$ **where**
 $\text{MultInv } p \ x = x \wedge \text{nat } (p - 2)$

13.1 Useful lemmas about dvd and powers

lemma *zpower-zdvd-prop1*:

$0 < n \implies p \text{ dvd } y \implies p \text{ dvd } ((y::\text{int}) ^ n)$
by (*induct* n) (*auto simp add: dvd-mult2 [of p y]*)

lemma *zdvd-bounds*: $n \text{ dvd } m \implies m \leq (0::\text{int}) \mid n \leq m$

proof –

assume $n \text{ dvd } m$
then have $\sim(0 < m \ \& \ m < n)$
using *zdvd-not-zless [of m n]* **by** *auto*
then show *?thesis* **by** *auto*

qed

lemma *zprime-zdvd-zmult-better*: $[[\text{zprime } p; \ p \text{ dvd } (m * n)]] \implies$

$(p \text{ dvd } m) \mid (p \text{ dvd } n)$
apply (*cases* $0 \leq m$)
apply (*simp add: zprime-zdvd-zmult*)
apply (*insert zprime-zdvd-zmult [of -m p n]*)
apply *auto*
done

lemma *zpower-zdvd-prop2*:

$\text{zprime } p \implies p \text{ dvd } ((y::\text{int}) ^ n) \implies 0 < n \implies p \text{ dvd } y$
apply (*induct* n)
apply *simp*
apply (*frule zprime-zdvd-zmult-better*)
apply *simp*
apply (*force simp del: dvd-mult*)
done

lemma *div-prop1*: $[[0 < z; \ (x::\text{int}) < y * z]] \implies x \text{ div } z < y$

proof –

assume $0 < z$ **then have** *modth*: $x \text{ mod } z \geq 0$ **by** *simp*
have $(x \text{ div } z) * z \leq (x \text{ div } z) * z$ **by** *simp*
then have $(x \text{ div } z) * z \leq (x \text{ div } z) * z + x \text{ mod } z$ **using** *modth* **by** *arith*
also have $\dots = x$
by (*auto simp add: zmod-zdiv-equality [symmetric] zmult-ac*)
also assume $x < y * z$
finally show *?thesis*
by (*auto simp add: prems mult-less-cancel-right, insert prems, arith*)

qed

lemma *div-prop2*: $[[0 < z; \ (x::\text{int}) < (y * z) + z]] \implies x \text{ div } z \leq y$

proof –

assume $0 < z$ **and** $x < (y * z) + z$
then have $x < (y + 1) * z$ **by** (*auto simp add: int-distrib*)
then have $x \text{ div } z < y + 1$
apply –
apply (*rule-tac* $y = y + 1$ **in** *div-prop1*)

```

    apply (auto simp add: prems)
  done
  then show ?thesis by auto
qed

```

```

lemma zdiv-leq-prop: [| 0 < y |] ==> y * (x div y) ≤ (x::int)
proof -
  assume 0 < y
  from zmod-zdiv-equality have x = y * (x div y) + x mod y by auto
  moreover have 0 ≤ x mod y
    by (auto simp add: prems pos-mod-sign)
  ultimately show ?thesis
    by arith
qed

```

13.2 Useful properties of congruences

```

lemma zcong-eq-zdvd-prop: [x = 0](mod p) = (p dvd x)
  by (auto simp add: zcong-def)

```

```

lemma zcong-id: [m = 0] (mod m)
  by (auto simp add: zcong-def)

```

```

lemma zcong-shift: [a = b] (mod m) ==> [a + c = b + c] (mod m)
  by (auto simp add: zcong-refl zcong-zadd)

```

```

lemma zcong-zpower: [x = y](mod m) ==> [x^z = y^z](mod m)
  by (induct z) (auto simp add: zcong-zmult)

```

```

lemma zcong-eq-trans: [| [a = b](mod m); b = c; [c = d](mod m) |] ==>
  [a = d](mod m)
  apply (erule zcong-trans)
  apply simp
  done

```

```

lemma aux1: a - b = (c::int) ==> a = c + b
  by auto

```

```

lemma zcong-zmult-prop1: [a = b](mod m) ==> ([c = a * d](mod m) =
  [c = b * d] (mod m))
  apply (auto simp add: zcong-def dvd-def)
  apply (rule-tac x = ka + k * d in exI)
  apply (drule aux1)+
  apply (auto simp add: int-distrib)
  apply (rule-tac x = ka - k * d in exI)
  apply (drule aux1)+
  apply (auto simp add: int-distrib)
  done

```

```

lemma zcong-zmult-prop2:  $[a = b](\text{mod } m) \implies$ 
   $([c = d * a](\text{mod } m) = [c = d * b](\text{mod } m))$ 
by (auto simp add: zmult-ac zcong-zmult-prop1)

lemma zcong-zmult-prop3:  $[\mid \text{zprime } p; \sim[x = 0](\text{mod } p);$ 
   $\sim[y = 0](\text{mod } p) \mid] \implies \sim[x * y = 0](\text{mod } p)$ 
apply (auto simp add: zcong-def)
apply (drule zprime-zdvd-zmult-better, auto)
done

lemma zcong-less-eq:  $[\mid 0 < x; 0 < y; 0 < m; [x = y](\text{mod } m);$ 
   $x < m; y < m \mid] \implies x = y$ 
by (metis zcong-not zcong-sym zless-linear)

lemma zcong-neg-1-impl-ne-1:  $[\mid 2 < p; [x = -1](\text{mod } p) \mid] \implies$ 
   $\sim([x = 1](\text{mod } p))$ 
proof
  assume  $2 < p$  and  $[x = 1](\text{mod } p)$  and  $[x = -1](\text{mod } p)$ 
  then have  $[1 = -1](\text{mod } p)$ 
    apply (auto simp add: zcong-sym)
    apply (drule zcong-trans, auto)
    done
  then have  $[1 + 1 = -1 + 1](\text{mod } p)$ 
    by (simp only: zcong-shift)
  then have  $[2 = 0](\text{mod } p)$ 
    by auto
  then have  $p \text{ dvd } 2$ 
    by (auto simp add: dvd-def zcong-def)
  with prems show False
    by (auto simp add: zdvd-not-zless)
qed

lemma zcong-zero-equiv-div:  $[a = 0](\text{mod } m) = (m \text{ dvd } a)$ 
by (auto simp add: zcong-def)

lemma zcong-zprime-prod-zero:  $[\mid \text{zprime } p; 0 < a \mid] \implies$ 
   $[a * b = 0](\text{mod } p) \implies [a = 0](\text{mod } p) \mid [b = 0](\text{mod } p)$ 
by (auto simp add: zcong-zero-equiv-div zprime-zdvd-zmult)

lemma zcong-zprime-prod-zero-contr:  $[\mid \text{zprime } p; 0 < a \mid] \implies$ 
   $\sim[a = 0](\text{mod } p) \ \& \ \sim[b = 0](\text{mod } p) \implies \sim[a * b = 0](\text{mod } p)$ 
apply auto
apply (frule-tac a = a and b = b and p = p in zcong-zprime-prod-zero)
apply auto
done

lemma zcong-not-zero:  $[\mid 0 < x; x < m \mid] \implies \sim[x = 0](\text{mod } m)$ 
by (auto simp add: zcong-zero-equiv-div zdvd-not-zless)

```

```

lemma zcong-zero: [| 0 ≤ x; x < m; [x = 0](mod m) |] ==> x = 0
  apply (drule order-le-imp-less-or-eq, auto)
  apply (frule-tac m = m in zcong-not-zero)
  apply auto
  done

```

```

lemma all-relprime-prod-relprime: [| finite A; ∀ x ∈ A. zgcd x y = 1 |]
  ==> zgcd (setprod id A) y = 1
  by (induct set: finite) (auto simp add: zgcd-zgcd-zmult)

```

13.3 Some properties of MultInv

```

lemma MultInv-prop1: [| 2 < p; [x = y] (mod p) |] ==>
  [(MultInv p x) = (MultInv p y)] (mod p)
  by (auto simp add: MultInv-def zcong-zpower)

```

```

lemma MultInv-prop2: [| 2 < p; zprime p; ~([x = 0](mod p)) |] ==>
  [(x * (MultInv p x)) = 1] (mod p)
proof (simp add: MultInv-def zcong-eq-zdvd-prop)
  assume 2 < p and zprime p and ~ p dvd x
  have x * x ^ nat (p - 2) = x ^ (nat (p - 2) + 1)
    by auto
  also from prems have nat (p - 2) + 1 = nat (p - 2 + 1)
    by (simp only: nat-add-distrib)
  also have p - 2 + 1 = p - 1 by arith
  finally have [x * x ^ nat (p - 2) = x ^ nat (p - 1)] (mod p)
    by (rule ssubst, auto)
  also from prems have [x ^ nat (p - 1) = 1] (mod p)
    by (auto simp add: Little-Fermat)
  finally (zcong-trans) show [x * x ^ nat (p - 2) = 1] (mod p) .
qed

```

```

lemma MultInv-prop2a: [| 2 < p; zprime p; ~([x = 0](mod p)) |] ==>
  [(MultInv p x) * x = 1] (mod p)
  by (auto simp add: MultInv-prop2 zmult-ac)

```

```

lemma aux-1: 2 < p ==> ((nat p) - 2) = (nat (p - 2))
  by (simp add: nat-diff-distrib)

```

```

lemma aux-2: 2 < p ==> 0 < nat (p - 2)
  by auto

```

```

lemma MultInv-prop3: [| 2 < p; zprime p; ~([x = 0](mod p)) |] ==>
  ~([MultInv p x = 0](mod p))
  apply (auto simp add: MultInv-def zcong-eq-zdvd-prop aux-1)
  apply (drule aux-2)
  apply (drule zpower-zdvd-prop2, auto)
  done

```

```

lemma aux--1: [| 2 < p; zprime p; ~([x = 0](mod p)) |] ==>
  [(MultInv p (MultInv p x)) = (x * (MultInv p x) *
    (MultInv p (MultInv p x)))] (mod p)
apply (drule MultInv-prop2, auto)
apply (drule-tac k = MultInv p (MultInv p x) in zcong-scalar, auto)
apply (auto simp add: zcong-sym)
done

lemma aux--2: [| 2 < p; zprime p; ~([x = 0](mod p)) |] ==>
  [(x * (MultInv p x) * (MultInv p (MultInv p x))) = x] (mod p)
apply (frule MultInv-prop3, auto)
apply (insert MultInv-prop2 [of p MultInv p x], auto)
apply (drule MultInv-prop2, auto)
apply (drule-tac k = x in zcong-scalar2, auto)
apply (auto simp add: zmult-ac)
done

lemma MultInv-prop4: [| 2 < p; zprime p; ~([x = 0](mod p)) |] ==>
  [(MultInv p (MultInv p x)) = x] (mod p)
apply (frule aux--1, auto)
apply (drule aux--2, auto)
apply (drule zcong-trans, auto)
done

lemma MultInv-prop5: [| 2 < p; zprime p; ~([x = 0](mod p));
  ~([y = 0](mod p)); [(MultInv p x) = (MultInv p y)] (mod p) |] ==>
  [x = y] (mod p)
apply (drule-tac a = MultInv p x and b = MultInv p y and
  m = p and k = x in zcong-scalar)
apply (insert MultInv-prop2 [of p x], simp)
apply (auto simp only: zcong-sym [of MultInv p x * x])
apply (auto simp add: zmult-ac)
apply (drule zcong-trans, auto)
apply (drule-tac a = x * MultInv p y and k = y in zcong-scalar, auto)
apply (insert MultInv-prop2a [of p y], auto simp add: zmult-ac)
apply (insert zcong-zmult-prop2 [of y * MultInv p y 1 p y x])
apply (auto simp add: zcong-sym)
done

lemma MultInv-zcong-prop1: [| 2 < p; [j = k] (mod p) |] ==>
  [a * MultInv p j = a * MultInv p k] (mod p)
by (drule MultInv-prop1, auto simp add: zcong-scalar2)

lemma aux--1: [j = a * MultInv p k] (mod p) ==>
  [j * k = a * MultInv p k * k] (mod p)
by (auto simp add: zcong-scalar)

lemma aux--2: [| 2 < p; zprime p; ~([k = 0](mod p));
  [j * k = a * MultInv p k * k] (mod p) |] ==> [j * k = a] (mod p)

```

```

apply (insert MultInv-prop2a [of p k] zcong-zmult-prop2
  [of MultInv p k * k 1 p j * k a])
apply (auto simp add: zmult-ac)
done

lemma aux---3: [j * k = a] (mod p) ==> [(MultInv p j) * j * k =
  (MultInv p j) * a] (mod p)
by (auto simp add: zmult-assoc zcong-scalar2)

lemma aux---4: [| 2 < p; zprime p; ~([j = 0](mod p));
  [(MultInv p j) * j * k = (MultInv p j) * a] (mod p) |]
  ==> [k = a * (MultInv p j)] (mod p)
apply (insert MultInv-prop2a [of p j] zcong-zmult-prop1
  [of MultInv p j * j 1 p MultInv p j * a k])
apply (auto simp add: zmult-ac zcong-sym)
done

lemma MultInv-zcong-prop2: [| 2 < p; zprime p; ~([k = 0](mod p));
  ~([j = 0](mod p)); [j = a * MultInv p k] (mod p) |] ==>
  [k = a * MultInv p j] (mod p)
apply (drule aux---1)
apply (frule aux---2, auto)
by (drule aux---3, drule aux---4, auto)

lemma MultInv-zcong-prop3: [| 2 < p; zprime p; ~([a = 0](mod p));
  ~([k = 0](mod p)); ~([j = 0](mod p));
  [a * MultInv p j = a * MultInv p k] (mod p) |] ==>
  [j = k] (mod p)
apply (auto simp add: zcong-eq-zdvd-prop [of a p])
apply (frule zprime-imp-zrelprime, auto)
apply (insert zcong-cancel2 [of p a MultInv p j MultInv p k], auto)
apply (drule MultInv-prop5, auto)
done

end

```

14 Residue Sets

theory *Residues* **imports** *Int2* **begin**

Define the residue of a set, the standard residue, quadratic residues, and prove some basic properties.

definition

```

  ResSet    :: int => int set => bool where
  ResSet m X = (∀ y1 y2. (y1 ∈ X & y2 ∈ X & [y1 = y2] (mod m) --> y1 =
    y2))

```

definition

StandardRes :: *int* => *int* => *int* **where**
StandardRes *m* *x* = *x mod m*

definition

QuadRes :: *int* => *int* => *bool* **where**
QuadRes *m* *x* = ($\exists y. [(y \wedge 2) = x] \text{ (mod } m)\text{)$)

definition

Legendre :: *int* => *int* => *int* **where**
Legendre *a* *p* = (if ($[a = 0] \text{ (mod } p)\text{)$) then 0
 else if (*QuadRes* *p* *a*) then 1
 else -1)

definition

SR :: *int* => *int* set **where**
SR *p* = {*x*. ($0 \leq x$) & ($x < p$)}

definition

SRStar :: *int* => *int* set **where**
SRStar *p* = {*x*. ($0 < x$) & ($x < p$)}

14.1 Some useful properties of StandardRes

lemma *StandardRes-prop1*: $[x = \text{StandardRes } m \ x] \text{ (mod } m)\text{)$
by (*auto simp add: StandardRes-def zcong-zmod*)

lemma *StandardRes-prop2*: $0 < m \implies (\text{StandardRes } m \ x1 = \text{StandardRes } m \ x2)$
 $= ([x1 = x2] \text{ (mod } m)\text{)$
by (*auto simp add: StandardRes-def zcong-zmod-eq*)

lemma *StandardRes-prop3*: $(\sim[x = 0] \text{ (mod } p)\text{) = } (\sim(\text{StandardRes } p \ x = 0)\text{)$
by (*auto simp add: StandardRes-def zcong-def dvd-eq-mod-eq-0*)

lemma *StandardRes-prop4*: $2 < m$
 $\implies [\text{StandardRes } m \ x * \text{StandardRes } m \ y = (x * y)] \text{ (mod } m)\text{)$
by (*auto simp add: StandardRes-def zcong-zmod-eq*
mod-mult-eq [of x y m])

lemma *StandardRes-lbound*: $0 < p \implies 0 \leq \text{StandardRes } p \ x$
by (*auto simp add: StandardRes-def pos-mod-sign*)

lemma *StandardRes-ubound*: $0 < p \implies \text{StandardRes } p \ x < p$
by (*auto simp add: StandardRes-def pos-mod-bound*)

lemma *StandardRes-eq-zcong*:
 $(\text{StandardRes } m \ x = 0) = ([x = 0] \text{ (mod } m)\text{)$
by (*auto simp add: StandardRes-def zcong-eq-zdvd-prop dvd-def*)

14.2 Relations between StandardRes, SRStar, and SR

lemma *SRStar-SR-prop*: $x \in \text{SRStar } p \implies x \in \text{SR } p$
by (*auto simp add: SRStar-def SR-def*)

lemma *StandardRes-SR-prop*: $x \in \text{SR } p \implies \text{StandardRes } p \ x = x$
by (*auto simp add: SR-def StandardRes-def mod-pos-pos-trivial*)

lemma *StandardRes-SRStar-prop1*: $2 < p \implies (\text{StandardRes } p \ x \in \text{SRStar } p)$
 $= (\sim[x = 0] \ (\text{mod } p))$
apply (*auto simp add: StandardRes-prop3 StandardRes-def*
 $\text{SRStar-def pos-mod-bound}$)
apply (*subgoal-tac* $0 < p$)
apply (*drule-tac* $a = x$ **in** *pos-mod-sign, arith, simp*)
done

lemma *StandardRes-SRStar-prop1a*: $x \in \text{SRStar } p \implies \sim([x = 0] \ (\text{mod } p))$
by (*auto simp add: SRStar-def zcong-def zdvd-not-zless*)

lemma *StandardRes-SRStar-prop2*: $[\ 2 < p; \text{zprime } p; x \in \text{SRStar } p \]$
 $\implies \text{StandardRes } p \ (\text{MultInv } p \ x) \in \text{SRStar } p$
apply (*frule-tac* $x = (\text{MultInv } p \ x)$ **in** *StandardRes-SRStar-prop1, simp*)
apply (*rule MultInv-prop3*)
apply (*auto simp add: SRStar-def zcong-def zdvd-not-zless*)
done

lemma *StandardRes-SRStar-prop3*: $x \in \text{SRStar } p \implies \text{StandardRes } p \ x = x$
by (*auto simp add: SRStar-SR-prop StandardRes-SR-prop*)

lemma *StandardRes-SRStar-prop4*: $[\text{zprime } p; 2 < p; x \in \text{SRStar } p \]$
 $\implies \text{StandardRes } p \ x \in \text{SRStar } p$
by (*frule StandardRes-SRStar-prop3, auto*)

lemma *SRStar-mult-prop1*: $[\text{zprime } p; 2 < p; x \in \text{SRStar } p; y \in \text{SRStar } p \]$
 $\implies (\text{StandardRes } p \ (x * y)) : \text{SRStar } p$
apply (*frule-tac* $x = x$ **in** *StandardRes-SRStar-prop4, auto*)
apply (*frule-tac* $x = y$ **in** *StandardRes-SRStar-prop4, auto*)
apply (*auto simp add: StandardRes-SRStar-prop1 zcong-zmult-prop3*)
done

lemma *SRStar-mult-prop2*: $[\text{zprime } p; 2 < p; \sim([a = 0] \ (\text{mod } p));$
 $x \in \text{SRStar } p \]$
 $\implies \text{StandardRes } p \ (a * \text{MultInv } p \ x) \in \text{SRStar } p$
apply (*frule-tac* $x = x$ **in** *StandardRes-SRStar-prop2, auto*)
apply (*frule-tac* $x = \text{MultInv } p \ x$ **in** *StandardRes-SRStar-prop1*)
apply (*auto simp add: StandardRes-SRStar-prop1 zcong-zmult-prop3*)
done

lemma *SRStar-card*: $2 < p \implies \text{int}(\text{card}(\text{SRStar } p)) = p - 1$
by (*auto simp add: SRStar-def int-card-bdd-int-set-l-l*)

lemma *SRStar-finite*: $2 < p \implies \text{finite}(\text{SRStar } p)$
by (*auto simp add: SRStar-def bdd-int-set-l-l-finite*)

14.3 Properties relating ResSets with StandardRes

lemma *aux*: $x \bmod m = y \bmod m \implies [x = y] (\bmod m)$
apply (*subgoal-tac* $x = y \implies [x = y] (\bmod m)$)
apply (*subgoal-tac* $[x \bmod m = y \bmod m] (\bmod m) \implies [x = y] (\bmod m)$)
apply (*auto simp add: zcong-zmod [of x y m]*)
done

lemma *StandardRes-inj-on-ResSet*: $\text{ResSet } m \ X \implies (\text{inj-on } (\text{StandardRes } m) \ X)$
apply (*auto simp add: ResSet-def StandardRes-def inj-on-def*)
apply (*drule-tac* $m = m$ **in** *aux*, *auto*)
done

lemma *StandardRes-Sum*: $[\text{finite } X; 0 < m] \implies [\text{setsum } f \ X = \text{setsum } (\text{StandardRes } m \circ f) \ X] (\bmod m)$
apply (*rule-tac* $F = X$ **in** *finite-induct*)
apply (*auto intro!: zcong-zadd simp add: StandardRes-prop1*)
done

lemma *SR-pos*: $0 < m \implies (\text{StandardRes } m \text{ ' } X) \subseteq \{x. 0 \leq x \ \& \ x < m\}$
by (*auto simp add: StandardRes-ubound StandardRes-lbound*)

lemma *ResSet-finite*: $0 < m \implies \text{ResSet } m \ X \implies \text{finite } X$
apply (*rule-tac* $f = \text{StandardRes } m$ **in** *finite-imageD*)
apply (*rule-tac* $B = \{x. (0 :: \text{int}) \leq x \ \& \ x < m\}$ **in** *finite-subset*)
apply (*auto simp add: StandardRes-inj-on-ResSet bdd-int-set-l-finite SR-pos*)
done

lemma *mod-mod-is-mod*: $[x = x \bmod m] (\bmod m)$
by (*auto simp add: zcong-zmod*)

lemma *StandardRes-prod*: $[\text{finite } X; 0 < m] \implies [\text{setprod } f \ X = \text{setprod } (\text{StandardRes } m \circ f) \ X] (\bmod m)$
apply (*rule-tac* $F = X$ **in** *finite-induct*)
apply (*auto intro!: zcong-zmult simp add: StandardRes-prop1*)
done

lemma *ResSet-image*:
 $[0 < m; \text{ResSet } m \ A; \forall x \in A. \forall y \in A. ([f \ x = f \ y] (\bmod m) \implies x = y)] \implies$
 $\text{ResSet } m \ (f \text{ ' } A)$
by (*auto simp add: ResSet-def*)

14.4 Property for SRStar

```

lemma ResSet-SRStar-prop: ResSet p (SRStar p)
  by (auto simp add: SRStar-def ResSet-def zcong-zless-imp-eq)

end

```

15 Parity: Even and Odd Integers

```

theory EvenOdd
imports Int2
begin

```

```

definition
  zOdd :: int set where
    zOdd = {x.  $\exists k. x = 2 * k + 1$ }

```

```

definition
  zEven :: int set where
    zEven = {x.  $\exists k. x = 2 * k$ }

```

15.1 Some useful properties about even and odd

```

lemma zOddI [intro?]:  $x = 2 * k + 1 \implies x \in zOdd$ 
  and zOddE [elim?]:  $x \in zOdd \implies (!k. x = 2 * k + 1 \implies C) \implies C$ 
  by (auto simp add: zOdd-def)

```

```

lemma zEvenI [intro?]:  $x = 2 * k \implies x \in zEven$ 
  and zEvenE [elim?]:  $x \in zEven \implies (!k. x = 2 * k \implies C) \implies C$ 
  by (auto simp add: zEven-def)

```

```

lemma one-not-even:  $\sim(1 \in zEven)$ 
proof
  assume  $1 \in zEven$ 
  then obtain k :: int where  $1 = 2 * k$  ..
  then show False by arith
qed

```

```

lemma even-odd-conj:  $\sim(x \in zOdd \ \&\ x \in zEven)$ 
proof –
  {
    fix a b
    assume  $2 * (a::int) = 2 * (b::int) + 1$ 
    then have  $2 * (a::int) - 2 * (b::int) = 1$ 
      by arith
    then have  $2 * (a - b) = 1$ 
      by (auto simp add: zdiff-zmult-distrib)
    moreover have  $(2 * (a - b)) \in zEven$ 

```

```

    by (auto simp only: zEven-def)
  ultimately have False
    by (auto simp add: one-not-even)
}
then show ?thesis
  by (auto simp add: zOdd-def zEven-def)
qed

lemma even-odd-disj:  $(x \in \text{zOdd} \mid x \in \text{zEven})$ 
  by (simp add: zOdd-def zEven-def) arith

lemma not-odd-impl-even:  $\sim(x \in \text{zOdd}) \implies x \in \text{zEven}$ 
  using even-odd-disj by auto

lemma odd-mult-odd-prop:  $(x*y):\text{zOdd} \implies x \in \text{zOdd}$ 
proof (rule classical)
  assume  $\neg ?thesis$ 
  then have  $x \in \text{zEven}$  by (rule not-odd-impl-even)
  then obtain  $a$  where  $a: x = 2 * a$  ..
  assume  $x * y : \text{zOdd}$ 
  then obtain  $b$  where  $x * y = 2 * b + 1$  ..
  with  $a$  have  $2 * a * y = 2 * b + 1$  by simp
  then have  $2 * a * y - 2 * b = 1$ 
    by arith
  then have  $2 * (a * y - b) = 1$ 
    by (auto simp add: zdiff-zmult-distrib)
  moreover have  $(2 * (a * y - b)):\text{zEven}$ 
    by (auto simp only: zEven-def)
  ultimately have False
    by (auto simp add: one-not-even)
  then show ?thesis ..
qed

lemma odd-minus-one-even:  $x \in \text{zOdd} \implies (x - 1):\text{zEven}$ 
  by (auto simp add: zOdd-def zEven-def)

lemma even-div-2-prop1:  $x \in \text{zEven} \implies (x \bmod 2) = 0$ 
  by (auto simp add: zEven-def)

lemma even-div-2-prop2:  $x \in \text{zEven} \implies (2 * (x \text{ div } 2)) = x$ 
  by (auto simp add: zEven-def)

lemma even-plus-even:  $[| x \in \text{zEven}; y \in \text{zEven} |] \implies x + y \in \text{zEven}$ 
  apply (auto simp add: zEven-def)
  apply (auto simp only: zadd-zmult-distrib2 [symmetric])
  done

lemma even-times-either:  $x \in \text{zEven} \implies x * y \in \text{zEven}$ 
  by (auto simp add: zEven-def)

```

```

lemma even-minus-even: [|  $x \in zEven$ ;  $y \in zEven$  |] ==>  $x - y \in zEven$ 
  apply (auto simp add: zEven-def)
  apply (auto simp only: zdiff-zmult-distrib2 [symmetric])
  done

lemma odd-minus-odd: [|  $x \in zOdd$ ;  $y \in zOdd$  |] ==>  $x - y \in zEven$ 
  apply (auto simp add: zOdd-def zEven-def)
  apply (auto simp only: zdiff-zmult-distrib2 [symmetric])
  done

lemma even-minus-odd: [|  $x \in zEven$ ;  $y \in zOdd$  |] ==>  $x - y \in zOdd$ 
  apply (auto simp add: zOdd-def zEven-def)
  apply (rule-tac  $x = k - ka - 1$  in exI)
  apply auto
  done

lemma odd-minus-even: [|  $x \in zOdd$ ;  $y \in zEven$  |] ==>  $x - y \in zOdd$ 
  apply (auto simp add: zOdd-def zEven-def)
  apply (auto simp only: zdiff-zmult-distrib2 [symmetric])
  done

lemma odd-times-odd: [|  $x \in zOdd$ ;  $y \in zOdd$  |] ==>  $x * y \in zOdd$ 
  apply (auto simp add: zOdd-def zadd-zmult-distrib zadd-zmult-distrib2)
  apply (rule-tac  $x = 2 * ka * k + ka + k$  in exI)
  apply (auto simp add: zadd-zmult-distrib)
  done

lemma odd-iff-not-even: ( $x \in zOdd$ ) = ( $\sim (x \in zEven)$ )
  using even-odd-conj even-odd-disj by auto

lemma even-product:  $x * y \in zEven$  ==>  $x \in zEven \mid y \in zEven$ 
  using odd-iff-not-even odd-times-odd by auto

lemma even-diff:  $x - y \in zEven = ((x \in zEven) = (y \in zEven))$ 
proof
  assume  $xy$ :  $x - y \in zEven$ 
  {
    assume  $x$ :  $x \in zEven$ 
    have  $y \in zEven$ 
    proof (rule classical)
      assume  $\neg ?thesis$ 
      then have  $y \in zOdd$ 
        by (simp add: odd-iff-not-even)
      with  $x$  have  $x - y \in zOdd$ 
        by (simp add: even-minus-odd)
      with  $xy$  have  $False$ 
        by (auto simp add: odd-iff-not-even)
      then show  $?thesis$  ..
    }

```

```

    qed
  } moreover {
    assume  $y: y \in \text{zEven}$ 
    have  $x \in \text{zEven}$ 
    proof (rule classical)
      assume  $\neg ?thesis$ 
      then have  $x \in \text{zOdd}$ 
        by (auto simp add: odd-iff-not-even)
      with  $y$  have  $x - y \in \text{zOdd}$ 
        by (simp add: odd-minus-even)
      with  $xy$  have  $\text{False}$ 
        by (auto simp add: odd-iff-not-even)
      then show  $?thesis$  ..
    qed
  }
  ultimately show  $(x \in \text{zEven}) = (y \in \text{zEven})$ 
    by (auto simp add: odd-iff-not-even even-minus-even odd-minus-odd
      even-minus-odd odd-minus-even)
next
  assume  $(x \in \text{zEven}) = (y \in \text{zEven})$ 
  then show  $x - y \in \text{zEven}$ 
    by (auto simp add: odd-iff-not-even even-minus-even odd-minus-odd
      even-minus-odd odd-minus-even)
qed

```

lemma *neg-one-even-power*: $[[x \in \text{zEven}; 0 \leq x]] \implies (-1::\text{int})^{\text{nat } x} = 1$
proof –

```

  assume  $x \in \text{zEven}$  and  $0 \leq x$ 
  from  $\langle x \in \text{zEven} \rangle$  obtain  $a$  where  $x = 2 * a$  ..
  with  $\langle 0 \leq x \rangle$  have  $0 \leq a$  by simp
  from  $\langle 0 \leq x \rangle$  and  $\langle x = 2 * a \rangle$  have  $\text{nat } x = \text{nat } (2 * a)$ 
    by simp
  also from  $\langle x = 2 * a \rangle$  have  $\text{nat } (2 * a) = 2 * \text{nat } a$ 
    by (simp add: nat-mult-distrib)
  finally have  $(-1::\text{int})^{\text{nat } x} = (-1)^{(2 * \text{nat } a)}$ 
    by simp
  also have  $\dots = ((-1::\text{int})^2)^{\text{nat } a}$ 
    by (simp add: zpower-zpower [symmetric])
  also have  $(-1::\text{int})^2 = 1$ 
    by simp
  finally show  $?thesis$ 
    by simp
qed

```

lemma *neg-one-odd-power*: $[[x \in \text{zOdd}; 0 \leq x]] \implies (-1::\text{int})^{\text{nat } x} = -1$
proof –

```

  assume  $x \in \text{zOdd}$  and  $0 \leq x$ 
  from  $\langle x \in \text{zOdd} \rangle$  obtain  $a$  where  $x = 2 * a + 1$  ..
  with  $\langle 0 \leq x \rangle$  have  $a: 0 \leq a$  by simp

```

```

with  $\langle 0 \leq x \rangle$  and  $\langle x = 2 * a + 1 \rangle$  have  $\text{nat } x = \text{nat } (2 * a + 1)$ 
  by simp
also from  $a$  have  $\text{nat } (2 * a + 1) = 2 * \text{nat } a + 1$ 
  by (auto simp add: nat-mult-distrib nat-add-distrib)
finally have  $(-1::\text{int})^{\text{nat } x} = (-1)^{(2 * \text{nat } a + 1)}$ 
  by simp
also have  $\dots = ((-1::\text{int})^2)^{(\text{nat } a) * (-1)^1}$ 
  by (auto simp add: zpower-zpower [symmetric] zpower-zadd-distrib)
also have  $(-1::\text{int})^2 = 1$ 
  by simp
finally show ?thesis
  by simp
qed

```

```

lemma neg-one-power-parity:  $[[0 \leq x; 0 \leq y; (x \in \text{zEven}) = (y \in \text{zEven})]] \implies$ 
   $(-1::\text{int})^{\text{nat } x} = (-1::\text{int})^{\text{nat } y}$ 
  using even-odd-disj [of x] even-odd-disj [of y]
  by (auto simp add: neg-one-even-power neg-one-odd-power)

```

```

lemma one-not-neg-one-mod-m:  $2 < m \implies \sim([1 = -1] \pmod m)$ 
  by (auto simp add: zcong-def zdvd-not-zless)

```

```

lemma even-div-2-l:  $[[y \in \text{zEven}; x < y]] \implies x \text{ div } 2 < y \text{ div } 2$ 
proof –
  assume  $y \in \text{zEven}$  and  $x < y$ 
  from  $\langle y \in \text{zEven} \rangle$  obtain  $k$  where  $k: y = 2 * k$  ..
  with  $\langle x < y \rangle$  have  $x < 2 * k$  by simp
  then have  $x \text{ div } 2 < k$  by (auto simp add: div-prop1)
  also have  $k = (2 * k) \text{ div } 2$  by simp
  finally have  $x \text{ div } 2 < 2 * k \text{ div } 2$  by simp
  with  $k$  show ?thesis by simp
qed

```

```

lemma even-sum-div-2:  $[[x \in \text{zEven}; y \in \text{zEven}]] \implies (x + y) \text{ div } 2 = x \text{ div } 2$ 
   $+ y \text{ div } 2$ 
  by (auto simp add: zEven-def)

```

```

lemma even-prod-div-2:  $[[x \in \text{zEven}]] \implies (x * y) \text{ div } 2 = (x \text{ div } 2) * y$ 
  by (auto simp add: zEven-def)

```

```

lemma zprime-zOdd-eq-grt-2:  $\text{zprime } p \implies (p \in \text{zOdd}) = (2 < p)$ 
  apply (auto simp add: zOdd-def zprime-def)
  apply (drule-tac x = 2 in allE)
  using odd-iff-not-even [of p]
  apply (auto simp add: zOdd-def zEven-def)
  done

```

```

lemma neg-one-special: finite A ==>
   $((-1 :: \text{int})^{\text{card } A} * (-1^{\text{card } A}) = 1$ 
  by (induct set: finite) auto

lemma neg-one-power:  $(-1 :: \text{int})^n = 1 \mid (-1 :: \text{int})^n = -1$ 
  by (induct n) auto

lemma neg-one-power-eq-mod-m:  $[2 < m; [(-1 :: \text{int})^j = (-1 :: \text{int})^k] \pmod m]$ 
  by (auto simp add: one-not-neg-one-mod-m zcong-sym)

end

```

16 Euler's criterion

theory *Euler* **imports** *Residues EvenOdd* **begin**

definition

```

MultiInvPair :: int ==> int ==> int ==> int set where
MultiInvPair a p j = {StandardRes p j, StandardRes p (a * (MultiInv p j))}

```

definition

```

SetS :: int ==> int ==> int set set where
SetS a p = (MultiInvPair a p ‘ SRStar p)

```

16.1 Property for MultiInvPair

lemma *MultiInvPair-prop1a*:

```

   $[z\text{prime } p; 2 < p; \sim([a = 0] \pmod p);$ 
   $X \in (\text{SetS } a \text{ } p); Y \in (\text{SetS } a \text{ } p);$ 
   $\sim((X \cap Y) = \{\}) \implies X = Y$ 
  apply (auto simp add: SetS-def)
  apply (drule StandardRes-SRStar-prop1a) + defer 1
  apply (drule StandardRes-SRStar-prop1a) +
  apply (auto simp add: MultiInvPair-def StandardRes-prop2 zcong-sym)
  apply (drule notE, rule MultiInv-zcong-prop1, auto)[]
  apply (drule notE, rule MultiInv-zcong-prop2, auto simp add: zcong-sym)[]
  apply (drule MultiInv-zcong-prop2, auto simp add: zcong-sym)[]
  apply (drule MultiInv-zcong-prop3, auto simp add: zcong-sym)[]
  apply (drule MultiInv-zcong-prop1, auto)[]
  apply (drule MultiInv-zcong-prop2, auto simp add: zcong-sym)[]
  apply (drule MultiInv-zcong-prop2, auto simp add: zcong-sym)[]
  apply (drule MultiInv-zcong-prop3, auto simp add: zcong-sym)[]

```



```

done

lemma MultInvPair-prop1b:
  [| zprime p; 2 < p; ~([a = 0](mod p));
    X ∈ (SetS a p); Y ∈ (SetS a p);
    X ≠ Y |] ==> X ∩ Y = {}
  apply (rule notnotD)
  apply (rule notI)
  apply (drule MultInvPair-prop1a, auto)
done

lemma MultInvPair-prop1c: [| zprime p; 2 < p; ~([a = 0](mod p)) |] ==>
  ∀ X ∈ SetS a p. ∀ Y ∈ SetS a p. X ≠ Y --> X ∩ Y = {}
  by (auto simp add: MultInvPair-prop1b)

lemma MultInvPair-prop2: [| zprime p; 2 < p; ~([a = 0](mod p)) |] ==>
  Union ( SetS a p ) = SRStar p
  apply (auto simp add: SetS-def MultInvPair-def StandardRes-SRStar-prop4
    SRStar-mult-prop2)
  apply (frule StandardRes-SRStar-prop3)
  apply (rule bexI, auto)
done

lemma MultInvPair-distinct: [| zprime p; 2 < p; ~([a = 0] (mod p));
  ~([j = 0] (mod p));
  ~ (QuadRes p a) |] ==>
  ~([j = a * MultInv p j] (mod p))

proof
  assume zprime p and 2 < p and ~([a = 0] (mod p)) and
    ~([j = 0] (mod p)) and ~ (QuadRes p a)
  assume [j = a * MultInv p j] (mod p)
  then have [j * j = (a * MultInv p j) * j] (mod p)
    by (auto simp add: zcong-scalar)
  then have a:[j * j = a * (MultInv p j * j)] (mod p)
    by (auto simp add: zmult-ac)
  have [j * j = a] (mod p)
  proof -
    from prems have b: [MultInv p j * j = 1] (mod p)
      by (simp add: MultInv-prop2a)
    from b a show ?thesis
      by (auto simp add: zcong-zmult-prop2)
  qed
  then have [j^2 = a] (mod p)
    by (metis number-of-is-id power2-eq-square succ-bin-simps)
  with prems show False
    by (simp add: QuadRes-def)
qed

lemma MultInvPair-card-two: [| zprime p; 2 < p; ~([a = 0] (mod p));

```

```

      ~(QuadRes p a); ~([j = 0] (mod p)) ]] ==>
      card (MultInvPair a p j) = 2
  apply (auto simp add: MultInvPair-def)
  apply (subgoal-tac ~ (StandardRes p j = StandardRes p (a * MultInv p j)))
  apply auto
  apply (metis MultInvPair-distinct Pls-def StandardRes-def aux number-of-is-id
one-is-num-one)
done

```

16.2 Properties of SetS

```

lemma SetS-finite: 2 < p ==> finite (SetS a p)
  by (auto simp add: SetS-def SRStar-finite [of p] finite-imageI)

```

```

lemma SetS-elems-finite:  $\forall X \in \text{SetS } a \ p. \text{ finite } X$ 
  by (auto simp add: SetS-def MultInvPair-def)

```

```

lemma SetS-elems-card: [[ zprime p; 2 < p; ~([a = 0] (mod p));
      ~(QuadRes p a) ]] ==>
       $\forall X \in \text{SetS } a \ p. \text{ card } X = 2$ 
  apply (auto simp add: SetS-def)
  apply (frule StandardRes-SRStar-prop1a)
  apply (rule MultInvPair-card-two, auto)
done

```

```

lemma Union-SetS-finite: 2 < p ==> finite (Union (SetS a p))
  by (auto simp add: SetS-finite SetS-elems-finite finite-Union)

```

```

lemma card-setsum-aux: [[ finite S;  $\forall X \in S. \text{ finite } (X::\text{int set})$ ;
       $\forall X \in S. \text{ card } X = n$  ]] ==> setsum card S = setsum (%x. n) S
  by (induct set: finite) auto

```

```

lemma SetS-card: [[ zprime p; 2 < p; ~([a = 0] (mod p)); ~(QuadRes p a) ]]
==>

```

$$\text{int}(\text{card}(\text{SetS } a \ p)) = (p - 1) \text{ div } 2$$

proof –

```

  assume zprime p and 2 < p and ~([a = 0] (mod p)) and ~(QuadRes p a)
  then have (p - 1) = 2 * int(card(SetS a p))

```

proof –

```

  have p - 1 = int(card(Union (SetS a p)))

```

```

    by (auto simp add: prems MultInvPair-prop2 SRStar-card)

```

```

  also have ... = int (setsum card (SetS a p))

```

```

    by (auto simp add: prems SetS-finite SetS-elems-finite

```

```

      MultInvPair-prop1c [of p a] card-Union-disjoint)

```

```

  also have ... = int(setsum (%x.2) (SetS a p))

```

```

    using prems

```

```

    by (auto simp add: SetS-elems-card SetS-finite SetS-elems-finite
      card-setsum-aux simp del: setsum-constant)

```

```

  also have ... = 2 * int(card(SetS a p))

```

```

    by (auto simp add: prems SetS-finite setsum-const2)
  finally show ?thesis .
qed
from this show ?thesis
  by auto
qed

lemma SetS-setprod-prop: [| zprime p; 2 < p;  $\sim([a = 0] \pmod p)$ ;
 $\sim(\text{QuadRes } p \ a)$ ;  $x \in (\text{SetS } a \ p)$  |] ==>
  [|  $\prod x = a$  |]  $\pmod p$ 
  apply (auto simp add: SetS-def MultInvPair-def)
  apply (frule StandardRes-SRStar-prop1a)
  apply (subgoal-tac StandardRes p x  $\neq$  StandardRes p (a * MultInv p x))
  apply (auto simp add: StandardRes-prop2 MultInvPair-distinct)
  apply (frule-tac m = p and x = x and y = (a * MultInv p x) in
    StandardRes-prop4)
  apply (subgoal-tac [x * (a * MultInv p x) = a * (x * MultInv p x)]  $\pmod p$ )
  apply (drule-tac a = StandardRes p x * StandardRes p (a * MultInv p x) and
    b = x * (a * MultInv p x) and
    c = a * (x * MultInv p x) in zcong-trans, force)
  apply (frule-tac p = p and x = x in MultInv-prop2, auto)
apply (metis StandardRes-SRStar-prop3 mult-1-right mult-commute zcong-sym zcong-zmult-prop1)
  apply (auto simp add: zmult-ac)
  done

lemma aux1: [| 0 < x; (x::int) < a; x  $\neq$  (a - 1) |] ==> x < a - 1
  by arith

lemma aux2: [| (a::int) < c; b < c |] ==> (a  $\leq$  b | b  $\leq$  a)
  by auto

lemma d22set-induct-old: ( $\bigwedge a::int. 1 < a \longrightarrow P(a - 1) \implies P a$ )  $\implies P x$ 
  using d22set.induct by blast

lemma SRStar-d22set-prop: 2 < p  $\implies$  (SRStar p) = {1}  $\cup$  (d22set (p - 1))
  apply (induct p rule: d22set-induct-old)
  apply auto
  apply (simp add: SRStar-def d22set.simps)
  apply (simp add: SRStar-def d22set.simps, clarify)
  apply (frule aux1)
  apply (frule aux2, auto)
  apply (simp-all add: SRStar-def)
  apply (simp add: d22set.simps)
  apply (frule d22set-le)
  apply (frule d22set-g-1, auto)
  done

lemma Union-SetS-setprod-prop1: [| zprime p; 2 < p;  $\sim([a = 0] \pmod p)$ ;  $\sim(\text{QuadRes } p \ a)$  |] ==>

```

$$[\prod (Union (SetS a p)) = a \wedge nat ((p - 1) div 2)] (mod$$

$p)$

proof –

assume $zprime\ p$ **and** $2 < p$ **and** $\sim([a = 0] (mod\ p))$ **and** $\sim(QuadRes\ p\ a)$

then have $[\prod (Union (SetS a p)) =$

$setprod\ (setprod\ (\%x.\ x))\ (SetS\ a\ p)] (mod\ p)$

by (*auto simp add: SetS-finite SetS-elems-finite*

MultInvPair-prop1c setprod-Union-disjoint)

also have $[setprod\ (setprod\ (\%x.\ x))\ (SetS\ a\ p) =$

$setprod\ (\%x.\ a)\ (SetS\ a\ p)] (mod\ p)$

by (*rule setprod-same-function-zcong*)

 (*auto simp add: prems SetS-setprod-prop SetS-finite*)

also (*zcong-trans*) **have** $[setprod\ (\%x.\ a)\ (SetS\ a\ p) =$

$a^{card\ (SetS\ a\ p)}] (mod\ p)$

by (*auto simp add: prems SetS-finite setprod-constant*)

finally (*zcong-trans*) **show** *?thesis*

apply (*rule zcong-trans*)

apply (*subgoal-tac card(SetS a p) = nat((p - 1) div 2), auto*)

apply (*subgoal-tac nat(int(card(SetS a p))) = nat((p - 1) div 2), force*)

apply (*auto simp add: prems SetS-card*)

done

qed

lemma *Union-SetS-setprod-prop2*: $[\mid zprime\ p; 2 < p; \sim([a = 0](mod\ p)) \mid] ==>$

$$\prod (Union (SetS a p)) = zfact\ (p - 1)$$

proof –

assume $zprime\ p$ **and** $2 < p$ **and** $\sim([a = 0](mod\ p))$

then have $\prod (Union (SetS a p)) = \prod (SRStar\ p)$

by (*auto simp add: MultInvPair-prop2*)

also have $\dots = \prod (\{1\} \cup (d22set\ (p - 1)))$

by (*auto simp add: prems SRStar-d22set-prop*)

also have $\dots = zfact(p - 1)$

proof –

have $\sim(1 \in d22set\ (p - 1)) \ \&\ finite\ (d22set\ (p - 1))$

by (*metis d22set-fin d22set-g-1 linorder-neq-iff*)

then have $\prod (\{1\} \cup (d22set\ (p - 1))) = \prod (d22set\ (p - 1))$

by *auto*

then show *?thesis*

by (*auto simp add: d22set-prod-zfact*)

qed

finally show *?thesis* .

qed

lemma *zfact-prop*: $[\mid zprime\ p; 2 < p; \sim([a = 0] (mod\ p)); \sim(QuadRes\ p\ a) \mid]$

$==>$

$$[zfact\ (p - 1) = a \wedge nat ((p - 1) div 2)] (mod\ p)$$

apply (*frule Union-SetS-setprod-prop1*)

apply (*auto simp add: Union-SetS-setprod-prop2*)

done

Prove the first part of Euler's Criterion:

lemma *Euler-part1*: $[[2 < p; \text{zprime } p; \sim([x = 0](\text{mod } p))];$
 $\sim(\text{QuadRes } p \ x) \] \implies$
 $[x^{\text{nat } ((p) - 1 \text{ div } 2)} = -1](\text{mod } p)$
by (*metis Wilson-Russ number-of-is-id zcong-sym zcong-trans zfact-prop*)

Prove another part of Euler Criterion:

lemma *aux-1*: $0 < p \implies (a::\text{int})^{\text{nat } (p)} = a * a^{\text{nat } (p) - 1}$
proof –
assume $0 < p$
then have $a^{\text{nat } (p)} = a^{\text{nat } (1 + (\text{nat } p - 1))}$
by (*auto simp add: diff-add-assoc*)
also have $\dots = (a^1) * a^{\text{nat } (p) - 1}$
by (*simp only: zpower-zadd-distrib*)
also have $\dots = a * a^{\text{nat } (p) - 1}$
by *auto*
finally show *?thesis* .
qed

lemma *aux-2*: $[[2::\text{int}] < p; p \in \text{zOdd}] \implies 0 < ((p - 1) \text{ div } 2)$
proof –
assume $2 < p$ **and** $p \in \text{zOdd}$
then have $(p - 1) \in \text{zEven}$
by (*auto simp add: zEven-def zOdd-def*)
then have *aux-1*: $2 * ((p - 1) \text{ div } 2) = (p - 1)$
by (*auto simp add: even-div-2-prop2*)
with $2 < p$ **have** $1 < (p - 1)$
by *auto*
then have $1 < (2 * ((p - 1) \text{ div } 2))$
by (*auto simp add: aux-1*)
then have $0 < (2 * ((p - 1) \text{ div } 2)) \text{ div } 2$
by *auto*
then show *?thesis* **by** *auto*
qed

lemma *Euler-part2*:
 $[[2 < p; \text{zprime } p; [a = 0](\text{mod } p)]] \implies [0 = a^{\text{nat } ((p - 1) \text{ div } 2)}](\text{mod } p)$
apply (*frule zprime-zOdd-eq-grt-2*)
apply (*frule aux-2, auto*)
apply (*frule-tac a = a in aux-1, auto*)
apply (*frule zcong-zmult-prop1, auto*)
done

Prove the final part of Euler's Criterion:

lemma *aux-1*: $[[\sim([x = 0](\text{mod } p)); [y^2 = x](\text{mod } p)]] \implies \sim(p \text{ dvd } y)$

```

by (metis dvdI power2-eq-square zcong-sym zcong-trans zcong-zero-equiv-div dvd-trans)

lemma aux-2: 2 * nat((p - 1) div 2) = nat (2 * ((p - 1) div 2))
by (auto simp add: nat-mult-distrib)

lemma Euler-part3: [| 2 < p; zprime p; ~([x = 0](mod p)); QuadRes p x |] ==>
  [x^(nat (((p) - 1) div 2)) = 1](mod p)
  apply (subgoal-tac p ∈ zOdd)
  apply (auto simp add: QuadRes-def)
  prefer 2
  apply (metis number-of-is-id numeral-1-eq-1 zprime-zOdd-eq-grt-2)
  apply (frule aux-1, auto)
  apply (drule-tac z = nat ((p - 1) div 2) in zcong-zpower)
  apply (auto simp add: zpower-zpower)
  apply (rule zcong-trans)
  apply (auto simp add: zcong-sym [of x ^ nat ((p - 1) div 2)])
  apply (metis Little-Fermat even-div-2-prop2 mult-Bit0 number-of-is-id odd-minus-one-even
one-is-num-one zmult-1 aux-2)
  done

```

Finally show Euler's Criterion:

```

theorem Euler-Criterion: [| 2 < p; zprime p |] ==> [(Legendre a p) =
  a^(nat (((p) - 1) div 2))](mod p)
  apply (auto simp add: Legendre-def Euler-part2)
  apply (frule Euler-part3, auto simp add: zcong-sym)[]
  apply (frule Euler-part1, auto simp add: zcong-sym)[]
  done

```

end

17 Gauss' Lemma

```

theory Gauss
imports Euler
begin

```

```

locale GAUSS =
  fixes p :: int
  fixes a :: int

  assumes p-prime: zprime p
  assumes p-g-2: 2 < p
  assumes p-a-relprime: ~[a = 0](mod p)
  assumes a-nonzero: 0 < a
begin

```

definition

$A :: \text{int set}$ **where**
 $A = \{(x :: \text{int}). 0 < x \ \& \ x \leq ((p - 1) \text{ div } 2)\}$

definition

$B :: \text{int set}$ **where**
 $B = (\%x. x * a) \text{ ' } A$

definition

$C :: \text{int set}$ **where**
 $C = \text{StandardRes } p \text{ ' } B$

definition

$D :: \text{int set}$ **where**
 $D = C \cap \{x. x \leq ((p - 1) \text{ div } 2)\}$

definition

$E :: \text{int set}$ **where**
 $E = C \cap \{x. ((p - 1) \text{ div } 2) < x\}$

definition

$F :: \text{int set}$ **where**
 $F = (\%x. (p - x)) \text{ ' } E$

17.1 Basic properties of p

lemma $p\text{-odd}$: $p \in \text{zOdd}$

by (*auto simp add: p-prime p-g-2 zprime-zOdd-eq-grt-2*)

lemma $p\text{-g-0}$: $0 < p$

using $p\text{-g-2}$ **by** *auto*

lemma int-nat : $\text{int } (\text{nat } ((p - 1) \text{ div } 2)) = (p - 1) \text{ div } 2$

using *ListMem.insert p-g-2* **by** (*auto simp add: pos-imp-zdiv-nonneg-iff*)

lemma $p\text{-minus-one-l}$: $(p - 1) \text{ div } 2 < p$

proof –

have $(p - 1) \text{ div } 2 \leq (p - 1) \text{ div } 1$

by (*rule zdiv-mono2*) (*auto simp add: p-g-0*)

also have $\dots = p - 1$ **by** *simp*

finally show *?thesis* **by** *simp*

qed

lemma $p\text{-eq}$: $p = (2 * (p - 1) \text{ div } 2) + 1$

using *div-mult-self1-is-id [of 2 p - 1]* **by** *auto*

lemma (*in* $-$) zodd-imp-zdiv-eq : $x \in \text{zOdd} ==> 2 * (x - 1) \text{ div } 2 = 2 * ((x - 1) \text{ div } 2)$

```

apply (frule odd-minus-one-even)
apply (simp add: zEven-def)
apply (subgoal-tac  $2 \neq 0$ )
apply (frule-tac  $b = 2 :: \text{int}$  and  $a = x - 1$  in div-mult-self1-is-id)
apply (auto simp add: even-div-2-prop2)
done

```

```

lemma p-eq2:  $p = (2 * ((p - 1) \text{div } 2)) + 1$ 
apply (insert p-eq p-prime p-g-2 zprime-zOdd-eq-grt-2 [of p], auto)
apply (frule zodd-imp-zdiv-eq, auto)
done

```

17.2 Basic Properties of the Gauss Sets

```

lemma finite-A: finite (A)
apply (auto simp add: A-def)
apply (subgoal-tac  $\{x. 0 < x \ \& \ x \leq (p - 1) \text{div } 2\} \subseteq \{x. 0 \leq x \ \& \ x < 1 + (p - 1) \text{div } 2\}$ )
apply (auto simp add: bdd-int-set-l-finite finite-subset)
done

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma A-card-eq:  $\text{card } A = \text{nat } ((p - 1) \text{div } 2)$ 
apply (auto simp add: A-def)
apply (insert int-nat)
apply (erule subst)
apply (auto simp add: card-bdd-int-set-l-le)
done

```

```

lemma inj-on-xa-A: inj-on ( $\%x. x * a$ ) A
using a-nonzero by (simp add: A-def inj-on-def)

```



```

lemma A-res: ResSet p A
  apply (auto simp add: A-def ResSet-def)
  apply (rule-tac m = p in zcong-less-eq)
  apply (insert p-g-2, auto)
  done

lemma B-res: ResSet p B
  apply (insert p-g-2 p-a-relprime p-minus-one-l)
  apply (auto simp add: B-def)
  apply (rule ResSet-image)
  apply (auto simp add: A-res)
  apply (auto simp add: A-def)
proof -
  fix x fix y
  assume a:  $[x * a = y * a] \pmod p$ 
  assume b:  $0 < x$ 
  assume c:  $x \leq (p - 1) \text{ div } 2$ 
  assume d:  $0 < y$ 
  assume e:  $y \leq (p - 1) \text{ div } 2$ 
  from a p-a-relprime p-prime a-nonzero zcong-cancel [of p a x y]
  have  $[x = y] \pmod p$ 
  by (simp add: zprime-imp-zrelprime zcong-def p-g-0 order-le-less)
  with zcong-less-eq [of x y p] p-minus-one-l
    order-le-less-trans [of x (p - 1) div 2 p]
    order-le-less-trans [of y (p - 1) div 2 p] show x = y
  by (simp add: prems p-minus-one-l p-g-0)
qed

lemma SR-B-inj: inj-on (StandardRes p) B
  apply (auto simp add: B-def StandardRes-def inj-on-def A-def prems)
proof -
  fix x fix y
  assume a:  $x * a \text{ mod } p = y * a \text{ mod } p$ 
  assume b:  $0 < x$ 
  assume c:  $x \leq (p - 1) \text{ div } 2$ 
  assume d:  $0 < y$ 
  assume e:  $y \leq (p - 1) \text{ div } 2$ 
  assume f:  $x \neq y$ 
  from a have  $[x * a = y * a] \pmod p$ 
  by (simp add: zcong-zmod-eq p-g-0)
  with p-a-relprime p-prime a-nonzero zcong-cancel [of p a x y]
  have  $[x = y] \pmod p$ 
  by (simp add: zprime-imp-zrelprime zcong-def p-g-0 order-le-less)
  with zcong-less-eq [of x y p] p-minus-one-l
    order-le-less-trans [of x (p - 1) div 2 p]
    order-le-less-trans [of y (p - 1) div 2 p] have x = y
  by (simp add: prems p-minus-one-l p-g-0)
  then have False

```

```

    by (simp add: f)
  then show a = 0
    by simp
qed

```

```

lemma inj-on-pminusx-E: inj-on (%x. p - x) E
  apply (auto simp add: E-def C-def B-def A-def)
  apply (rule-tac g = %x. -1 * (x - p) in inj-on-inverseI)
  apply auto
done

```

```

lemma A-ncong-p: x ∈ A ==> ~[x = 0](mod p)
  apply (auto simp add: A-def)
  apply (frule-tac m = p in zcong-not-zero)
  apply (insert p-minus-one-l)
  apply auto
done

```

```

lemma A-greater-zero: x ∈ A ==> 0 < x
  by (auto simp add: A-def)

```

```

lemma B-ncong-p: x ∈ B ==> ~[x = 0](mod p)
  apply (auto simp add: B-def)
  apply (frule A-ncong-p)
  apply (insert p-a-relprime p-prime a-nonzero)
  apply (frule-tac a = x and b = a in zcong-zprime-prod-zero-contr)
  apply (auto simp add: A-greater-zero)
done

```

```

lemma B-greater-zero: x ∈ B ==> 0 < x
  using a-nonzero by (auto simp add: B-def mult-pos-pos A-greater-zero)

```

```

lemma C-ncong-p: x ∈ C ==> ~[x = 0](mod p)
  apply (auto simp add: C-def)
  apply (frule B-ncong-p)
  apply (subgoal-tac [x = StandardRes p x](mod p))
  defer apply (simp add: StandardRes-prop1)
  apply (frule-tac a = x and b = StandardRes p x and c = 0 in zcong-trans)
  apply auto
done

```

```

lemma C-greater-zero: y ∈ C ==> 0 < y
  apply (auto simp add: C-def)
proof -
  fix x
  assume a: x ∈ B
  from p-g-0 have 0 ≤ StandardRes p x
    by (simp add: StandardRes-lbound)
  moreover have ~[x = 0] (mod p)

```

```

    by (simp add: a B-ncong-p)
  then have StandardRes p x  $\neq$  0
    by (simp add: StandardRes-prop3)
  ultimately show 0 < StandardRes p x
    by (simp add: order-le-less)
qed

```

```

lemma D-ncong-p:  $x \in D \implies \sim[x = 0](\text{mod } p)$ 
  by (auto simp add: D-def C-ncong-p)

```

```

lemma E-ncong-p:  $x \in E \implies \sim[x = 0](\text{mod } p)$ 
  by (auto simp add: E-def C-ncong-p)

```

```

lemma F-ncong-p:  $x \in F \implies \sim[x = 0](\text{mod } p)$ 
  apply (auto simp add: F-def)
proof -
  fix x assume a:  $x \in E$  assume b:  $[p - x = 0] (\text{mod } p)$ 
  from E-ncong-p have  $\sim[x = 0] (\text{mod } p)$ 
    by (simp add: a)
  moreover from a have 0 < x
    by (simp add: a E-def C-greater-zero)
  moreover from a have  $x < p$ 
    by (auto simp add: E-def C-def p-g-0 StandardRes-ubound)
  ultimately have  $\sim[p - x = 0] (\text{mod } p)$ 
    by (simp add: zcong-not-zero)
  from this show False by (simp add: b)
qed

```

```

lemma F-subset:  $F \subseteq \{x. 0 < x \ \& \ x \leq ((p - 1) \text{ div } 2)\}$ 
  apply (auto simp add: F-def E-def)
  apply (insert p-g-0)
  apply (frule-tac x = xa in StandardRes-ubound)
  apply (frule-tac x = x in StandardRes-ubound)
  apply (subgoal-tac xa = StandardRes p xa)
  apply (auto simp add: C-def StandardRes-prop2 StandardRes-prop1)
proof -
  from zodd-imp-zdiv-eq p-prime p-g-2 zprime-zOdd-eq-grt-2 have
     $2 * (p - 1) \text{ div } 2 = 2 * ((p - 1) \text{ div } 2)$ 
    by simp
  with p-eq2 show  $\forall x. [(p - 1) \text{ div } 2 < \text{StandardRes } p \ x; x \in B] \implies p - \text{StandardRes } p \ x \leq (p - 1) \text{ div } 2$ 
    by simp
qed

```

```

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

```

```

lemma F-eq:  $F = \{x. \exists y \in A. (x = p - (\text{StandardRes } p (y * a)) \ \& \ (p - 1) \text{ div } 2 < \text{StandardRes } p (y * a))\}$ 

```

```

by (auto simp add: F-def E-def D-def C-def B-def A-def)

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

lemma D-leq:  $x \in D \implies x \leq (p - 1) \text{ div } 2$ 
by (auto simp add: D-eq)

lemma F-ge:  $x \in F \implies x \leq (p - 1) \text{ div } 2$ 
  apply (auto simp add: F-eq A-def)
proof -
  fix y
  assume  $(p - 1) \text{ div } 2 < \text{StandardRes } p (y * a)$ 
  then have  $p - \text{StandardRes } p (y * a) < p - ((p - 1) \text{ div } 2)$ 
    by arith
  also from p-eq2 have  $\dots = 2 * ((p - 1) \text{ div } 2) + 1 - ((p - 1) \text{ div } 2)$ 
    by auto
  also have  $2 * ((p - 1) \text{ div } 2) + 1 - (p - 1) \text{ div } 2 = (p - 1) \text{ div } 2 + 1$ 
    by arith
  finally show  $p - \text{StandardRes } p (y * a) \leq (p - 1) \text{ div } 2$ 
    using zless-add1-eq [of  $p - \text{StandardRes } p (y * a)$   $(p - 1) \text{ div } 2$ ] by auto
qed

lemma all-A-relprime:  $\forall x \in A. \text{zgcd } x \ p = 1$ 
  using p-prime p-minus-one-l by (auto simp add: A-def zless-zprime-imp-zrelprime)

lemma A-prod-relprime:  $\text{zgcd } (\text{setprod id } A) \ p = 1$ 
by (rule all-relprime-prod-relprime[OF finite-A all-A-relprime])



### 17.3 Relationships Between Gauss Sets



lemma B-card-eq-A:  $\text{card } B = \text{card } A$ 
  using finite-A by (simp add: finite-A B-def inj-on-xa-A card-image)

lemma B-card-eq:  $\text{card } B = \text{nat } ((p - 1) \text{ div } 2)$ 
  by (simp add: B-card-eq-A A-card-eq)

lemma F-card-eq-E:  $\text{card } F = \text{card } E$ 
  using finite-E by (simp add: F-def inj-on-pminusx-E card-image)

lemma C-card-eq-B:  $\text{card } C = \text{card } B$ 
  apply (insert finite-B)
  apply (subgoal-tac inj-on (StandardRes p) B)
  apply (simp add: B-def C-def card-image)
  apply (rule StandardRes-inj-on-ResSet)
  apply (simp add: B-res)
  done

```

```

lemma D-E-disj:  $D \cap E = \{\}$ 
  by (auto simp add: D-def E-def)

lemma C-card-eq-D-plus-E:  $\text{card } C = \text{card } D + \text{card } E$ 
  by (auto simp add: C-eq card-Un-disjoint D-E-disj finite-D finite-E)

lemma C-prod-eq-D-times-E:  $\text{setprod id } E * \text{setprod id } D = \text{setprod id } C$ 
  apply (insert D-E-disj finite-D finite-E C-eq)
  apply (frule setprod-Un-disjoint [of D E id])
  apply auto
  done

lemma C-B-zcong-prod:  $[\text{setprod id } C = \text{setprod id } B] \pmod{p}$ 
  apply (auto simp add: C-def)
  apply (insert finite-B SR-B-inj)
  apply (frule-tac f = StandardRes p in setprod-reindex-id [symmetric], auto)
  apply (rule setprod-same-function-zcong)
  apply (auto simp add: StandardRes-prop1 zcong-sym p-g-0)
  done

lemma F-Un-D-subset:  $(F \cup D) \subseteq A$ 
  apply (rule Un-least)
  apply (auto simp add: A-def F-subset D-subset)
  done

lemma F-D-disj:  $(F \cap D) = \{\}$ 
  apply (simp add: F-eq D-eq)
  apply (auto simp add: F-eq D-eq)
proof –
  fix y fix ya
  assume p –  $\text{StandardRes } p (y * a) = \text{StandardRes } p (ya * a)$ 
  then have  $p = \text{StandardRes } p (y * a) + \text{StandardRes } p (ya * a)$ 
    by arith
  moreover have p dvd p
    by auto
  ultimately have p dvd  $(\text{StandardRes } p (y * a) + \text{StandardRes } p (ya * a))$ 
    by auto
  then have a:  $[\text{StandardRes } p (y * a) + \text{StandardRes } p (ya * a) = 0] \pmod{p}$ 
    by (auto simp add: zcong-def)
  have  $[y * a = \text{StandardRes } p (y * a)] \pmod{p}$ 
    by (simp only: zcong-sym StandardRes-prop1)
  moreover have  $[ya * a = \text{StandardRes } p (ya * a)] \pmod{p}$ 
    by (simp only: zcong-sym StandardRes-prop1)
  ultimately have  $[y * a + ya * a = \text{StandardRes } p (y * a) + \text{StandardRes } p (ya * a)] \pmod{p}$ 
    by (rule zcong-zadd)
  with a have  $[y * a + ya * a = 0] \pmod{p}$ 
    apply (elim zcong-trans)
    by (simp only: zcong-refl)

```

also have $y * a + ya * a = a * (y + ya)$
by (*simp add: zadd-zmult-distrib2 zmult-commute*)
finally have $[a * (y + ya) = 0] \text{ (mod } p)$.
with $p\text{-prime } a\text{-nonzero } zcong\text{-zprime-prod-zero [of } p \text{ } a \text{ } y + ya]$
 $p\text{-a-relprime}$
have $a: [y + ya = 0] \text{ (mod } p)$
by *auto*
assume $b: y \in A$ **and** $c: ya: A$
with $A\text{-def}$ **have** $0 < y + ya$
by *auto*
moreover from $b \ c \ A\text{-def}$ **have** $y + ya \leq (p - 1) \text{ div } 2 + (p - 1) \text{ div } 2$
by *auto*
moreover from $b \ c \ p\text{-eq2 } A\text{-def}$ **have** $y + ya < p$
by *auto*
ultimately show *False*
apply *simp*
apply (*frule-tac m = p in zcong-not-zero*)
apply (*auto simp add: a*)
done
qed

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

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

lemma $\text{prod-D-F-eq-prod-A:}$
 $(\text{setprod id } D) * (\text{setprod id } F) = \text{setprod id } A$
apply (*insert F-D-disj finite-D finite-F*)
apply (*frule setprod-Un-disjoint [of F D id]*)
apply (*auto simp add: F-Un-D-eq-A*)
done

lemma prod-F-zcong:
 $[\text{setprod id } F = ((-1) ^ (\text{card } E)) * (\text{setprod id } E)] \text{ (mod } p)$
proof –
have $\text{setprod id } F = \text{setprod id } (op - p \text{ ` } E)$
by (*auto simp add: F-def*)
then have $\text{setprod id } F = \text{setprod } (op - p) \ E$
apply *simp*

apply (*insert finite-E inj-on-pminusx-E*)
apply (*frule-tac f = op - p in setprod-reindex-id, auto*)
done
then have one:

$$[\text{setprod id } F = \text{setprod } (\text{StandardRes } p \text{ o } (op - p)) \ E] \ (\text{mod } p)$$
apply *simp*
apply (*insert p-g-0 finite-E StandardRes-prod*)
by (*auto*)
moreover have $a: \forall x \in E. [p - x = 0 - x] \ (\text{mod } p)$
apply *clarify*
apply (*insert zcong-id [of p]*)
apply (*rule-tac a = p and m = p and c = x and d = x in zcong-zdiff, auto*)
done
moreover have $b: \forall x \in E. [\text{StandardRes } p \ (p - x) = p - x] (\text{mod } p)$
apply *clarify*
apply (*simp add: StandardRes-prop1 zcong-sym*)
done
moreover have $\forall x \in E. [\text{StandardRes } p \ (p - x) = -x] (\text{mod } p)$
apply *clarify*
apply (*insert a b*)
apply (*rule-tac b = p - x in zcong-trans, auto*)
done
ultimately have $c:$

$$[\text{setprod } (\text{StandardRes } p \text{ o } (op - p)) \ E = \text{setprod } (\text{uminus}) \ E] (\text{mod } p)$$
apply *simp*
using *finite-E p-g-0*

$$\text{setprod-same-function-zcong [of E StandardRes p o (op - p) uminus p]}$$
by *auto*
then have two: $[\text{setprod id } F = \text{setprod } (\text{uminus}) \ E] (\text{mod } p)$
apply (*insert one c*)
apply (*rule zcong-trans [of setprod id F*

$$\text{setprod } (\text{StandardRes } p \text{ o } op - p) \ E \ p$$

$$\text{setprod uminus } E], \text{ auto}$$
)
done
also have $\text{setprod uminus } E = (\text{setprod id } E) * (-1) ^ {(\text{card } E)}$
using *finite-E by (induct set: finite) auto*
then have $\text{setprod uminus } E = (-1) ^ {(\text{card } E)} * (\text{setprod id } E)$
by (*simp add: zmult-commute*)
with two show *?thesis*
by *simp*
qed

17.4 Gauss' Lemma

lemma *aux:* $\text{setprod id } A * -1 ^ {\text{card } E} * a ^ {\text{card } A} * -1 ^ {\text{card } E} = \text{setprod id } A * a ^ {\text{card } A}$
by (*auto simp add: finite-E neg-one-special*)

theorem *pre-gauss-lemma:*

```

[ $a^{\wedge \text{nat}((p-1) \text{ div } 2)} = (-1)^{\wedge (\text{card } E)}$ ](mod p)
proof -
  have [setprod id A = setprod id F * setprod id D](mod p)
    by (auto simp add: prod-D-F-eq-prod-A zmult-commute cong del:setprod-cong)
  then have [setprod id A =  $((-1)^{\wedge (\text{card } E)} * \text{setprod id E}) * \text{setprod id D}$ ](mod p)
    apply (rule zcong-trans)
    apply (auto simp add: prod-F-zcong zcong-scalar cong del: setprod-cong)
    done
  then have [setprod id A =  $((-1)^{\wedge (\text{card } E)} * \text{setprod id C})$ ](mod p)
    apply (rule zcong-trans)
    apply (insert C-prod-eq-D-times-E, erule subst)
    apply (subst zmult-assoc, auto)
    done
  then have [setprod id A =  $((-1)^{\wedge (\text{card } E)} * \text{setprod id B})$ ](mod p)
    apply (rule zcong-trans)
    apply (simp add: C-B-zcong-prod zcong-scalar2 cong del:setprod-cong)
    done
  then have [setprod id A =  $((-1)^{\wedge (\text{card } E)} * (\text{setprod id } (\%x. x * a) ' A))$ ](mod p)
    by (simp add: B-def)
  then have [setprod id A =  $((-1)^{\wedge (\text{card } E)} * (\text{setprod } (\%x. x * a) A))$ ](mod p)
    by (simp add: finite-A inj-on-xa-A setprod-reindex-id[symmetric] cong del:setprod-cong)
  moreover have setprod  $(\%x. x * a) A = \text{setprod } (\%x. a) A * \text{setprod id A}$ 
    using finite-A by (induct set: finite) auto
  ultimately have [setprod id A =  $((-1)^{\wedge (\text{card } E)} * (\text{setprod } (\%x. a) A * \text{setprod id A}))$ ](mod p)
    by simp
  then have [setprod id A =  $((-1)^{\wedge (\text{card } E)} * a^{\wedge (\text{card } A)} * \text{setprod id A})$ ](mod p)
    apply (rule zcong-trans)
    apply (simp add: zcong-scalar2 zcong-scalar finite-A setprod-constant zmult-assoc)
    done
  then have a: [setprod id A *  $(-1)^{\wedge (\text{card } E)} = ((-1)^{\wedge (\text{card } E)} * a^{\wedge (\text{card } A)} * \text{setprod id A} * (-1)^{\wedge (\text{card } E)})$ ](mod p)
    by (rule zcong-scalar)
  then have [setprod id A *  $(-1)^{\wedge (\text{card } E)} = \text{setprod id A} * (-1)^{\wedge (\text{card } E)} * a^{\wedge (\text{card } A)} * (-1)^{\wedge (\text{card } E)}$ ](mod p)
    apply (rule zcong-trans)
    apply (simp add: a mult-commute mult-left-commute)
    done
  then have [setprod id A *  $(-1)^{\wedge (\text{card } E)} = \text{setprod id A} * a^{\wedge (\text{card } A)}$ ](mod p)
    apply (rule zcong-trans)
    apply (simp add: aux cong del:setprod-cong)
    done
  with this zcong-cancel2 [of p setprod id A  $-1^{\wedge \text{card } E} a^{\wedge \text{card } A}$ ]

```



```

      p-g-0 A-prod-relprime have  $[-1 \wedge \text{card } E = a \wedge \text{card } A](\text{mod } p)$ 
    by (simp add: order-less-imp-le)
  from this show ?thesis
    by (simp add: A-card-eq zcong-sym)
qed

theorem gauss-lemma:  $(\text{Legendre } a \ p) = (-1) \wedge (\text{card } E)$ 
proof -
  from Euler-Criterion p-prime p-g-2 have
     $[(\text{Legendre } a \ p) = a^{\wedge(\text{nat } ((p) - 1) \text{ div } 2))}] (\text{mod } p)$ 
  by auto
  moreover note pre-gauss-lemma
  ultimately have  $[(\text{Legendre } a \ p) = (-1) \wedge (\text{card } E)] (\text{mod } p)$ 
  by (rule zcong-trans)
  moreover from p-a-relprime have  $(\text{Legendre } a \ p) = 1 \mid (\text{Legendre } a \ p) = (-1)$ 
  by (auto simp add: Legendre-def)
  moreover have  $(-1::\text{int}) \wedge (\text{card } E) = 1 \mid (-1::\text{int}) \wedge (\text{card } E) = -1$ 
  by (rule neg-one-power)
  ultimately show ?thesis
    by (auto simp add: p-g-2 one-not-neg-one-mod-m zcong-sym)
qed

end

end

```

18 The law of Quadratic reciprocity

```

theory Quadratic-Reciprocity
imports Gauss
begin

```

Lemmas leading up to the proof of theorem 3.3 in Niven and Zuckerman's presentation.

```

context GAUSS
begin

```

```

lemma QRLemma1:  $a * \text{setsum id } A =$ 
   $p * \text{setsum } (\%x. ((x * a) \text{ div } p)) \ A + \text{setsum id } D + \text{setsum id } E$ 
proof -
  from finite-A have  $a * \text{setsum id } A = \text{setsum } (\%x. a * x) \ A$ 
  by (auto simp add: setsum-const-mult id-def)
  also have  $\text{setsum } (\%x. a * x) = \text{setsum } (\%x. x * a)$ 
  by (auto simp add: zmult-commute)
  also have  $\text{setsum } (\%x. x * a) \ A = \text{setsum id } B$ 
  by (simp add: B-def setsum-reindex-id[OF inj-on-xa-A])
  also have  $\dots = \text{setsum } (\%x. p * (x \text{ div } p) + \text{StandardRes } p \ x) \ B$ 
  by (auto simp add: StandardRes-def zmod-zdiv-equality)

```

also have $\dots = \text{setsum } (\%x. p * (x \text{ div } p)) B + \text{setsum } (\text{StandardRes } p) B$
by (*rule setsum-addf*)
also have $\text{setsum } (\text{StandardRes } p) B = \text{setsum id } C$
by (*auto simp add: C-def setsum-reindex-id[OF SR-B-inj]*)
also from *C-eq* **have** $\dots = \text{setsum id } (D \cup E)$
by *auto*
also from *finite-D finite-E* **have** $\dots = \text{setsum id } D + \text{setsum id } E$
by (*rule setsum-Un-disjoint*) (*auto simp add: D-def E-def*)
also have $\text{setsum } (\%x. p * (x \text{ div } p)) B =$
 $\text{setsum } ((\%x. p * (x \text{ div } p)) o (\%x. (x * a))) A$
by (*auto simp add: B-def setsum-reindex inj-on-xa-A*)
also have $\dots = \text{setsum } (\%x. p * ((x * a) \text{ div } p)) A$
by (*auto simp add: o-def*)
also from *finite-A* **have** $\text{setsum } (\%x. p * ((x * a) \text{ div } p)) A =$
 $p * \text{setsum } (\%x. ((x * a) \text{ div } p)) A$
by (*auto simp add: setsum-const-mult*)
finally show *?thesis* **by** *arith*
qed

lemma *QRLemma2*: $\text{setsum id } A = p * \text{int } (\text{card } E) - \text{setsum id } E + \text{setsum id } D$

proof –
from *F-Un-D-eq-A* **have** $\text{setsum id } A = \text{setsum id } (D \cup F)$
by (*simp add: Un-commute*)
also from *F-D-disj finite-D finite-F*
have $\dots = \text{setsum id } D + \text{setsum id } F$
by (*auto simp add: Int-commute intro: setsum-Un-disjoint*)
also from *F-def* **have** $F = (\%x. (p - x)) ' E$
by *auto*
also from *finite-E inj-on-pminusx-E* **have** $\text{setsum id } ((\%x. (p - x)) ' E) =$
 $\text{setsum } (\%x. (p - x)) E$
by (*auto simp add: setsum-reindex*)
also from *finite-E* **have** $\text{setsum } (op - p) E = \text{setsum } (\%x. p) E - \text{setsum id } E$
by (*auto simp add: setsum-subtractf id-def*)
also from *finite-E* **have** $\text{setsum } (\%x. p) E = p * \text{int}(\text{card } E)$
by (*intro setsum-const*)
finally show *?thesis*
by *arith*
qed

lemma *QRLemma3*: $(a - 1) * \text{setsum id } A =$

$p * (\text{setsum } (\%x. ((x * a) \text{ div } p)) A - \text{int}(\text{card } E)) + 2 * \text{setsum id } E$

proof –
have $(a - 1) * \text{setsum id } A = a * \text{setsum id } A - \text{setsum id } A$
by (*auto simp add: zdiff-zmult-distrib*)
also note *QRLemma1*
also from *QRLemma2* **have** $p * (\sum x \in A. x * a \text{ div } p) + \text{setsum id } D +$
 $\text{setsum id } E - \text{setsum id } A =$
 $p * (\sum x \in A. x * a \text{ div } p) + \text{setsum id } D +$

$setsum\ id\ E - (p * int\ (card\ E) - setsum\ id\ E + setsum\ id\ D)$
 by *auto*
 also have $\dots = p * (\sum x \in A. x * a\ div\ p) -$
 $p * int\ (card\ E) + 2 * setsum\ id\ E$
 by *arith*
 finally show *?thesis*
 by (*auto simp only: zdiff-zmult-distrib2*)
 qed

lemma *QRLemma4*: $a \in zOdd ==>$
 $(setsum\ (\%x. ((x * a)\ div\ p))\ A \in zEven) = (int(card\ E): zEven)$
proof –
 assume *a-odd*: $a \in zOdd$
 from *QRLemma3* have *a*: $p * (setsum\ (\%x. ((x * a)\ div\ p))\ A - int(card\ E))$
 $=$
 $(a - 1) * setsum\ id\ A - 2 * setsum\ id\ E$
 by *arith*
 from *a-odd* have $a - 1 \in zEven$
 by (*rule odd-minus-one-even*)
 hence $(a - 1) * setsum\ id\ A \in zEven$
 by (*rule even-times-either*)
 moreover have $2 * setsum\ id\ E \in zEven$
 by (*auto simp add: zEven-def*)
 ultimately have $(a - 1) * setsum\ id\ A - 2 * setsum\ id\ E \in zEven$
 by (*rule even-minus-even*)
 with *a* have $p * (setsum\ (\%x. ((x * a)\ div\ p))\ A - int(card\ E)): zEven$
 by *simp*
 hence $p \in zEven \mid (setsum\ (\%x. ((x * a)\ div\ p))\ A - int(card\ E)): zEven$
 by (*rule EvenOdd.even-product*)
 with *p-odd* have $(setsum\ (\%x. ((x * a)\ div\ p))\ A - int(card\ E)): zEven$
 by (*auto simp add: odd-iff-not-even*)
 thus *?thesis*
 by (*auto simp only: even-diff [symmetric]*)
 qed

lemma *QRLemma5*: $a \in zOdd ==>$
 $(-1::int)^(card\ E) = (-1::int)^(nat(setsum\ (\%x. ((x * a)\ div\ p))\ A))$
proof –
 assume $a \in zOdd$
 from *QRLemma4* [*OF this*] have
 $(int(card\ E): zEven) = (setsum\ (\%x. ((x * a)\ div\ p))\ A \in zEven) ..$
 moreover have $0 \leq int(card\ E)$
 by *auto*
 moreover have $0 \leq setsum\ (\%x. ((x * a)\ div\ p))\ A$
proof (*intro setsum-nonneg*)
 show $\forall x \in A. 0 \leq x * a\ div\ p$
proof
 fix *x*
 assume $x \in A$

```

    then have  $0 \leq x$ 
      by (auto simp add: A-def)
    with a-nonzero have  $0 \leq x * a$ 
      by (auto simp add: zero-le-mult-iff)
    with p-g-2 show  $0 \leq x * a \text{ div } p$ 
      by (auto simp add: pos-imp-zdiv-nonneg-iff)
  qed
qed
ultimately have  $(-1::\text{int})^{\text{nat}((\text{int } (\text{card } E)))} =$ 
   $(-1)^{\text{nat}((\sum x \in A. x * a \text{ div } p))}$ 
  by (intro neg-one-power-parity, auto)
also have  $\text{nat } (\text{int}(\text{card } E)) = \text{card } E$ 
  by auto
finally show ?thesis .
qed

end

lemma MainQRLemma:  $[| a \in \text{zOdd}; 0 < a; \sim([a = 0] \text{ (mod } p)); \text{zprime } p; 2 <$ 
 $p;$ 
 $A = \{x. 0 < x \ \& \ x \leq (p - 1) \text{ div } 2\} \ |] ==>$ 
 $(\text{Legendre } a \ p) = (-1::\text{int})^{\text{nat}(\text{setsum } (\%x. ((x * a) \text{ div } p)) \ A)}$ 
  apply (subst GAUSS.gauss-lemma)
  apply (auto simp add: GAUSS-def)
  apply (subst GAUSS.QRLemma5)
  apply (auto simp add: GAUSS-def)
  apply (simp add: GAUSS.A-def [OF GAUSS.intro] GAUSS-def)
done

```

18.1 Stuff about S, S1 and S2

```

locale QRTEMP =
  fixes p      :: int
  fixes q      :: int

  assumes p-prime: zprime p
  assumes p-g-2: 2 < p
  assumes q-prime: zprime q
  assumes q-g-2: 2 < q
  assumes p-neq-q: p  $\neq$  q
begin

definition
  P-set :: int set where
  P-set =  $\{x. 0 < x \ \& \ x \leq ((p - 1) \text{ div } 2)\}$ 

definition
  Q-set :: int set where
  Q-set =  $\{x. 0 < x \ \& \ x \leq ((q - 1) \text{ div } 2)\}$ 

```

definition

$S :: (int * int) \text{ set } \mathbf{where}$
 $S = P\text{-set} <*> Q\text{-set}$

definition

$S1 :: (int * int) \text{ set } \mathbf{where}$
 $S1 = \{ (x, y). (x, y):S \ \& \ ((p * y) < (q * x)) \}$

definition

$S2 :: (int * int) \text{ set } \mathbf{where}$
 $S2 = \{ (x, y). (x, y):S \ \& \ ((q * x) < (p * y)) \}$

definition

$f1 :: int \Rightarrow (int * int) \text{ set } \mathbf{where}$
 $f1 \ j = \{ (j1, y). (j1, y):S \ \& \ j1 = j \ \& \ (y \leq (q * j) \text{ div } p) \}$

definition

$f2 :: int \Rightarrow (int * int) \text{ set } \mathbf{where}$
 $f2 \ j = \{ (x, j1). (x, j1):S \ \& \ j1 = j \ \& \ (x \leq (p * j) \text{ div } q) \}$

lemma $p\text{-fact}: 0 < (p - 1) \text{ div } 2$

proof –

from $p\text{-g-2}$ have $2 \leq p - 1$ by *arith*
then have $2 \text{ div } 2 \leq (p - 1) \text{ div } 2$ by (rule *zdiv-mono1*, *auto*)
then show ?thesis by *auto*

qed

lemma $q\text{-fact}: 0 < (q - 1) \text{ div } 2$

proof –

from $q\text{-g-2}$ have $2 \leq q - 1$ by *arith*
then have $2 \text{ div } 2 \leq (q - 1) \text{ div } 2$ by (rule *zdiv-mono1*, *auto*)
then show ?thesis by *auto*

qed

lemma $pb\text{-neg-qa}: [| 1 \leq b; b \leq (q - 1) \text{ div } 2 |] \Rightarrow$

$(p * b \neq q * a)$

proof

assume $p * b = q * a$ and $1 \leq b$ and $b \leq (q - 1) \text{ div } 2$
then have $q \text{ dvd } (p * b)$ by (auto simp add: *dvd-def*)
with $q\text{-prime}$ $p\text{-g-2}$ have $q \text{ dvd } p \mid q \text{ dvd } b$
by (auto simp add: *zprime-zdvd-zmult*)
moreover have $\sim (q \text{ dvd } p)$

proof

assume $q \text{ dvd } p$
with $p\text{-prime}$ have $q = 1 \mid q = p$
apply (auto simp add: *zprime-def* *QRTEMP-def*)
apply (drule-tac $x = q$ and $R = \text{False}$ in *allE*)
apply (simp add: *QRTEMP-def*)

```

    apply (subgoal-tac  $0 \leq q$ , simp add: QRTEMP-def)
    apply (insert prems)
    apply (auto simp add: QRTEMP-def)
    done
  with q-g-2 p-neq-q show False by auto
qed
ultimately have  $q \text{ dvd } b$  by auto
then have  $q \leq b$ 
proof -
  assume  $q \text{ dvd } b$ 
  moreover from prems have  $0 < b$  by auto
  ultimately show ?thesis using zdvd-bounds [of  $q \ b$ ] by auto
qed
with prems have  $q \leq (q - 1) \text{ div } 2$  by auto
then have  $2 * q \leq 2 * ((q - 1) \text{ div } 2)$  by arith
then have  $2 * q \leq q - 1$ 
proof -
  assume  $2 * q \leq 2 * ((q - 1) \text{ div } 2)$ 
  with prems have  $q \in \text{zOdd}$  by (auto simp add: QRTEMP-def zprime-zOdd-eq-grt-2)
  with odd-minus-one-even have  $(q - 1) : \text{zEven}$  by auto
  with even-div-2-prop2 have  $(q - 1) = 2 * ((q - 1) \text{ div } 2)$  by auto
  with prems show ?thesis by auto
qed
then have  $p1: q \leq -1$  by arith
with q-g-2 show False by auto
qed

lemma P-set-finite: finite (P-set)
  using p-fact by (auto simp add: P-set-def bdd-int-set-l-le-finite)

lemma Q-set-finite: finite (Q-set)
  using q-fact by (auto simp add: Q-set-def bdd-int-set-l-le-finite)

lemma S-finite: finite S
  by (auto simp add: S-def P-set-finite Q-set-finite finite-cartesian-product)

lemma S1-finite: finite S1
proof -
  have finite S by (auto simp add: S-finite)
  moreover have  $S1 \subseteq S$  by (auto simp add: S1-def S-def)
  ultimately show ?thesis by (auto simp add: finite-subset)
qed

lemma S2-finite: finite S2
proof -
  have finite S by (auto simp add: S-finite)
  moreover have  $S2 \subseteq S$  by (auto simp add: S2-def S-def)
  ultimately show ?thesis by (auto simp add: finite-subset)
qed

```

```

lemma P-set-card:  $(p - 1) \text{ div } 2 = \text{int } (\text{card } (P\text{-set}))$ 
  using p-fact by (auto simp add: P-set-def card-bdd-int-set-l-le)

lemma Q-set-card:  $(q - 1) \text{ div } 2 = \text{int } (\text{card } (Q\text{-set}))$ 
  using q-fact by (auto simp add: Q-set-def card-bdd-int-set-l-le)

lemma S-card:  $((p - 1) \text{ div } 2) * ((q - 1) \text{ div } 2) = \text{int } (\text{card}(S))$ 
  using P-set-card Q-set-card P-set-finite Q-set-finite
  by (auto simp add: S-def zmult-int setsum-constant)

lemma S1-Int-S2-prop:  $S1 \cap S2 = \{\}$ 
  by (auto simp add: S1-def S2-def)

lemma S1-Union-S2-prop:  $S = S1 \cup S2$ 
  apply (auto simp add: S-def P-set-def Q-set-def S1-def S2-def)
proof -
  fix a and b
  assume  $\sim q * a < p * b$  and b1:  $0 < b$  and b2:  $b \leq (q - 1) \text{ div } 2$ 
  with zless-linear have  $(p * b < q * a) \mid (p * b = q * a)$  by auto
  moreover from pb-neq-qa b1 b2 have  $(p * b \neq q * a)$  by auto
  ultimately show  $p * b < q * a$  by auto
qed

lemma card-sum-S1-S2:  $((p - 1) \text{ div } 2) * ((q - 1) \text{ div } 2) =$ 
   $\text{int}(\text{card}(S1)) + \text{int}(\text{card}(S2))$ 
proof -
  have  $((p - 1) \text{ div } 2) * ((q - 1) \text{ div } 2) = \text{int } (\text{card}(S))$ 
  by (auto simp add: S-card)
  also have  $\dots = \text{int } (\text{card}(S1) + \text{card}(S2))$ 
  apply (insert S1-finite S2-finite S1-Int-S2-prop S1-Union-S2-prop)
  apply (drule card-Un-disjoint, auto)
  done
  also have  $\dots = \text{int}(\text{card}(S1)) + \text{int}(\text{card}(S2))$  by auto
  finally show ?thesis .
qed

lemma aux1a:  $\llbracket 0 < a; a \leq (p - 1) \text{ div } 2;$ 
   $0 < b; b \leq (q - 1) \text{ div } 2 \rrbracket \implies$ 
   $(p * b < q * a) = (b \leq q * a \text{ div } p)$ 
proof -
  assume  $0 < a$  and  $a \leq (p - 1) \text{ div } 2$  and  $0 < b$  and  $b \leq (q - 1) \text{ div } 2$ 
  have  $p * b < q * a \implies b \leq q * a \text{ div } p$ 
  proof -
  assume  $p * b < q * a$ 
  then have  $p * b \leq q * a$  by auto
  then have  $(p * b) \text{ div } p \leq (q * a) \text{ div } p$ 
  by (rule zdiv-mono1) (insert p-g-2, auto)
  then show  $b \leq (q * a) \text{ div } p$ 

```

```

    apply (subgoal-tac p ≠ 0)
    apply (frule div-mult-self1-is-id, force)
    apply (insert p-g-2, auto)
  done
qed
moreover have b ≤ q * a div p ==> p * b < q * a
proof -
  assume b ≤ q * a div p
  then have p * b ≤ p * ((q * a) div p)
    using p-g-2 by (auto simp add: mult-le-cancel-left)
  also have ... ≤ q * a
    by (rule zdiv-leq-prop) (insert p-g-2, auto)
  finally have p * b ≤ q * a .
  then have p * b < q * a | p * b = q * a
    by (simp only: order-le-imp-less-or-eq)
  moreover have p * b ≠ q * a
    by (rule pb-neq-qa) (insert prems, auto)
  ultimately show ?thesis by auto
qed
ultimately show ?thesis ..
qed

lemma aux1b: [| 0 < a; a ≤ (p - 1) div 2;
               0 < b; b ≤ (q - 1) div 2 |] ==>
  (q * a < p * b) = (a ≤ p * b div q)

proof -
  assume 0 < a and a ≤ (p - 1) div 2 and 0 < b and b ≤ (q - 1) div 2
  have q * a < p * b ==> a ≤ p * b div q
  proof -
    assume q * a < p * b
    then have q * a ≤ p * b by auto
    then have (q * a) div q ≤ (p * b) div q
      by (rule zdiv-mono1) (insert q-g-2, auto)
    then show a ≤ (p * b) div q
      apply (subgoal-tac q ≠ 0)
      apply (frule div-mult-self1-is-id, force)
      apply (insert q-g-2, auto)
    done
  qed
  moreover have a ≤ p * b div q ==> q * a < p * b
  proof -
    assume a ≤ p * b div q
    then have q * a ≤ q * ((p * b) div q)
      using q-g-2 by (auto simp add: mult-le-cancel-left)
    also have ... ≤ p * b
      by (rule zdiv-leq-prop) (insert q-g-2, auto)
    finally have q * a ≤ p * b .
    then have q * a < p * b | q * a = p * b
      by (simp only: order-le-imp-less-or-eq)

```



```

    moreover have  $p * b \neq q * a$ 
      by (rule pb-neq-qa) (insert prems, auto)
    ultimately show ?thesis by auto
  qed
  ultimately show ?thesis ..
qed

lemma (in -) aux2: [| zprime p; zprime q; 2 < p; 2 < q |] ==>
  ( $q * ((p - 1) \text{ div } 2) \text{ div } p \leq (q - 1) \text{ div } 2$ )
proof -
  assume zprime p and zprime q and 2 < p and 2 < q

  then have  $p \in \text{zOdd} \ \& \ q \in \text{zOdd}$ 
    by (auto simp add: zprime-zOdd-eq-grt-2)
  then have even1:  $(p - 1) : \text{zEven} \ \& \ (q - 1) : \text{zEven}$ 
    by (auto simp add: odd-minus-one-even)
  then have even2:  $(2 * p) : \text{zEven} \ \& \ ((q - 1) * p) : \text{zEven}$ 
    by (auto simp add: zEven-def)
  then have even3:  $((q - 1) * p) + (2 * p) : \text{zEven}$ 
    by (auto simp: EvenOdd.even-plus-even)

  from prems have  $q * (p - 1) < ((q - 1) * p) + (2 * p)$ 
    by (auto simp add: int-distrib)
  then have  $((p - 1) * q) \text{ div } 2 < (((q - 1) * p) + (2 * p)) \text{ div } 2$ 
    apply (rule-tac  $x = ((p - 1) * q)$  in even-div-2-l)
    by (auto simp add: even3, auto simp add: zmult-ac)
  also have  $((p - 1) * q) \text{ div } 2 = q * ((p - 1) \text{ div } 2)$ 
    by (auto simp add: even1 even-prod-div-2)
  also have  $((q - 1) * p) + (2 * p) \text{ div } 2 = (((q - 1) \text{ div } 2) * p) + p$ 
    by (auto simp add: even1 even2 even-prod-div-2 even-sum-div-2)
  finally show ?thesis
    apply (rule-tac  $x = q * ((p - 1) \text{ div } 2)$  and
       $y = (q - 1) \text{ div } 2$  in div-prop2)
    using prems by auto
qed

lemma aux3a:  $\forall j \in P\text{-set}. \text{int} (\text{card } (f1 \ j)) = (q * j) \text{ div } p$ 
proof
  fix j
  assume j-fact:  $j \in P\text{-set}$ 
  have int (card (f1 j)) = int (card {y. y ∈ Q-set & y ≤ (q * j) div p})
  proof -
    have finite (f1 j)
    proof -
      have  $(f1 \ j) \subseteq S$  by (auto simp add: f1-def)
      with S-finite show ?thesis by (auto simp add: finite-subset)
    qed
    moreover have inj-on  $(\%(x,y). y) (f1 \ j)$ 
      by (auto simp add: f1-def inj-on-def)
  qed

```

```

ultimately have card ((%(x,y). y) ‘ (f1 j)) = card (f1 j)
  by (auto simp add: f1-def card-image)
moreover have ((%(x,y). y) ‘ (f1 j)) = {y. y ∈ Q-set & y ≤ (q * j) div p}
  using prems by (auto simp add: f1-def S-def Q-set-def P-set-def image-def)
ultimately show ?thesis by (auto simp add: f1-def)
qed
also have ... = int (card {y. 0 < y & y ≤ (q * j) div p})
proof -
  have {y. y ∈ Q-set & y ≤ (q * j) div p} =
    {y. 0 < y & y ≤ (q * j) div p}
  apply (auto simp add: Q-set-def)
proof -
  fix x
  assume 0 < x and x ≤ q * j div p
  with j-fact P-set-def have j ≤ (p - 1) div 2 by auto
  with q-g-2 have q * j ≤ q * ((p - 1) div 2)
    by (auto simp add: mult-le-cancel-left)
  with p-g-2 have q * j div p ≤ q * ((p - 1) div 2) div p
    by (auto simp add: zdiv-mono1)
  also from prems P-set-def have ... ≤ (q - 1) div 2
  apply simp
  apply (insert aux2)
  apply (simp add: QRTEMP-def)
  done
  finally show x ≤ (q - 1) div 2 using prems by auto
qed
then show ?thesis by auto
qed
also have ... = (q * j) div p
proof -
  from j-fact P-set-def have 0 ≤ j by auto
  with q-g-2 have q * 0 ≤ q * j by (auto simp only: mult-left-mono)
  then have 0 ≤ q * j by auto
  then have 0 div p ≤ (q * j) div p
  apply (rule-tac a = 0 in zdiv-mono1)
  apply (insert p-g-2, auto)
  done
  also have 0 div p = 0 by auto
  finally show ?thesis by (auto simp add: card-bdd-int-set-l-le)
qed
finally show int (card (f1 j)) = q * j div p .
qed

lemma aux3b: ∀ j ∈ Q-set. int (card (f2 j)) = (p * j) div q
proof
  fix j
  assume j-fact: j ∈ Q-set
  have int (card (f2 j)) = int (card {y. y ∈ P-set & y ≤ (p * j) div q})
  proof -

```

```

have finite (f2 j)
proof -
  have (f2 j)  $\subseteq$  S by (auto simp add: f2-def)
  with S-finite show ?thesis by (auto simp add: finite-subset)
qed
moreover have inj-on (%(x,y). x) (f2 j)
  by (auto simp add: f2-def inj-on-def)
ultimately have card ((%(x,y). x) ` (f2 j)) = card (f2 j)
  by (auto simp add: f2-def card-image)
moreover have ((%(x,y). x) ` (f2 j)) = {y. y  $\in$  P-set & y  $\leq$  (p * j) div q}
  using prems by (auto simp add: f2-def S-def Q-set-def P-set-def image-def)
ultimately show ?thesis by (auto simp add: f2-def)
qed
also have ... = int (card {y. 0 < y & y  $\leq$  (p * j) div q})
proof -
  have {y. y  $\in$  P-set & y  $\leq$  (p * j) div q} =
    {y. 0 < y & y  $\leq$  (p * j) div q}
  apply (auto simp add: P-set-def)
proof -
  fix x
  assume 0 < x and x  $\leq$  p * j div q
  with j-fact Q-set-def have j  $\leq$  (q - 1) div 2 by auto
  with p-g-2 have p * j  $\leq$  p * ((q - 1) div 2)
    by (auto simp add: mult-le-cancel-left)
  with q-g-2 have p * j div q  $\leq$  p * ((q - 1) div 2) div q
    by (auto simp add: zdiv-mono1)
  also from prems have ...  $\leq$  (p - 1) div 2
    by (auto simp add: aux2 QRTEMP-def)
  finally show x  $\leq$  (p - 1) div 2 using prems by auto
qed
then show ?thesis by auto
qed
also have ... = (p * j) div q
proof -
  from j-fact Q-set-def have 0  $\leq$  j by auto
  with p-g-2 have p * 0  $\leq$  p * j by (auto simp only: mult-left-mono)
  then have 0  $\leq$  p * j by auto
  then have 0 div q  $\leq$  (p * j) div q
    apply (rule-tac a = 0 in zdiv-mono1)
    apply (insert q-g-2, auto)
  done
  also have 0 div q = 0 by auto
  finally show ?thesis by (auto simp add: card-bdd-int-set-l-le)
qed
finally show int (card (f2 j)) = p * j div q .
qed

lemma S1-card: int (card(S1)) = setsum (%j. (q * j) div p) P-set
proof -

```

```

have  $\forall x \in P\text{-set}. \text{finite } (f1\ x)$ 
proof
  fix  $x$ 
  have  $f1\ x \subseteq S$  by (auto simp add: f1-def)
  with S-finite show finite (f1 x) by (auto simp add: finite-subset)
qed
moreover have  $(\forall x \in P\text{-set}. \forall y \in P\text{-set}. x \neq y \longrightarrow (f1\ x) \cap (f1\ y) = \{\})$ 
  by (auto simp add: f1-def)
moreover note P-set-finite
ultimately have  $\text{int}(\text{card } (\text{UNION } P\text{-set } f1)) =$ 
   $\text{setsum } (\%x. \text{int}(\text{card } (f1\ x)))\ P\text{-set}$ 
  by (simp add: card-UN-disjoint int-setsum o-def)
moreover have  $S1 = \text{UNION } P\text{-set } f1$ 
  by (auto simp add: f1-def S-def S1-def S2-def P-set-def Q-set-def aux1a)
ultimately have  $\text{int}(\text{card } (S1)) = \text{setsum } (\%j. \text{int}(\text{card } (f1\ j)))\ P\text{-set}$ 
  by auto
also have  $\dots = \text{setsum } (\%j. q * j \text{ div } p)\ P\text{-set}$ 
  using aux3a by (fastsimp intro: setsum-cong)
finally show ?thesis .
qed

```

```

lemma S2-card:  $\text{int } (\text{card}(S2)) = \text{setsum } (\%j. (p * j) \text{ div } q)\ Q\text{-set}$ 
proof -
  have  $\forall x \in Q\text{-set}. \text{finite } (f2\ x)$ 
  proof
    fix  $x$ 
    have  $f2\ x \subseteq S$  by (auto simp add: f2-def)
    with S-finite show finite (f2 x) by (auto simp add: finite-subset)
  qed
  moreover have  $(\forall x \in Q\text{-set}. \forall y \in Q\text{-set}. x \neq y \longrightarrow$ 
     $(f2\ x) \cap (f2\ y) = \{\})$ 
    by (auto simp add: f2-def)
  moreover note Q-set-finite
  ultimately have  $\text{int}(\text{card } (\text{UNION } Q\text{-set } f2)) =$ 
     $\text{setsum } (\%x. \text{int}(\text{card } (f2\ x)))\ Q\text{-set}$ 
    by (simp add: card-UN-disjoint int-setsum o-def)
  moreover have  $S2 = \text{UNION } Q\text{-set } f2$ 
    by (auto simp add: f2-def S-def S1-def S2-def P-set-def Q-set-def aux1b)
  ultimately have  $\text{int}(\text{card } (S2)) = \text{setsum } (\%j. \text{int}(\text{card } (f2\ j)))\ Q\text{-set}$ 
    by auto
  also have  $\dots = \text{setsum } (\%j. p * j \text{ div } q)\ Q\text{-set}$ 
    using aux3b by (fastsimp intro: setsum-cong)
  finally show ?thesis .
qed

```

```

lemma S1-carda:  $\text{int } (\text{card}(S1)) =$ 
   $\text{setsum } (\%j. (j * q) \text{ div } p)\ P\text{-set}$ 
  by (auto simp add: S1-card zmult-ac)

```

```

lemma S2-carda: int (card(S2)) =
  setsum (%j. (j * p) div q) Q-set
by (auto simp add: S2-card zmult-ac)

lemma pq-sum-prop: (setsum (%j. (j * p) div q) Q-set) +
  (setsum (%j. (j * q) div p) P-set) = ((p - 1) div 2) * ((q - 1) div 2)
proof -
  have (setsum (%j. (j * p) div q) Q-set) +
    (setsum (%j. (j * q) div p) P-set) = int (card S2) + int (card S1)
  by (auto simp add: S1-carda S2-carda)
  also have ... = int (card S1) + int (card S2)
  by auto
  also have ... = ((p - 1) div 2) * ((q - 1) div 2)
  by (auto simp add: card-sum-S1-S2)
  finally show ?thesis .
qed

lemma (in -) pq-prime-neg: [| zprime p; zprime q; p ≠ q |] ==> (~[p = 0] (mod
q))
apply (auto simp add: zcong-eq-zdvd-prop zprime-def)
apply (drule-tac x = q in allE)
apply (drule-tac x = p in allE)
apply auto
done

lemma QR-short: (Legendre p q) * (Legendre q p) =
  (-1::int) ^ nat(((p - 1) div 2) * ((q - 1) div 2))
proof -
  from prems have ~([p = 0] (mod q))
  by (auto simp add: pq-prime-neg QRTEMP-def)
  with prems Q-set-def have a1: (Legendre p q) = (-1::int) ^
    nat(setsum (%x. ((x * p) div q)) Q-set)
  apply (rule-tac p = q in MainQRLemma)
  apply (auto simp add: zprime-zOdd-eq-grt-2 QRTEMP-def)
  done
  from prems have ~([q = 0] (mod p))
  apply (rule-tac p = q and q = p in pq-prime-neg)
  apply (simp add: QRTEMP-def)
  done
  with prems P-set-def have a2: (Legendre q p) =
    (-1::int) ^ nat(setsum (%x. ((x * q) div p)) P-set)
  apply (rule-tac p = p in MainQRLemma)
  apply (auto simp add: zprime-zOdd-eq-grt-2 QRTEMP-def)
  done
  from a1 a2 have (Legendre p q) * (Legendre q p) =
    (-1::int) ^ nat(setsum (%x. ((x * p) div q)) Q-set) *
    (-1::int) ^ nat(setsum (%x. ((x * q) div p)) P-set)

```

```

    by auto
  also have ... = (-1::int) ^ (nat(setsum (%x. ((x * p) div q)) Q-set) +
    nat(setsum (%x. ((x * q) div p)) P-set))
    by (auto simp add: zpower-zadd-distrib)
  also have nat(setsum (%x. ((x * p) div q)) Q-set) +
    nat(setsum (%x. ((x * q) div p)) P-set) =
    nat((setsum (%x. ((x * p) div q)) Q-set) +
      (setsum (%x. ((x * q) div p)) P-set))
  apply (rule-tac z = setsum (%x. ((x * p) div q)) Q-set in
    nat-add-distrib [symmetric])
  apply (auto simp add: S1-carda [symmetric] S2-carda [symmetric])
  done
  also have ... = nat(((p - 1) div 2) * ((q - 1) div 2))
    by (auto simp add: pq-sum-prop)
  finally show ?thesis .
qed

```

end

theorem Quadratic-Reciprocity:

```

  [| p ∈ zOdd; zprime p; q ∈ zOdd; zprime q;
    p ≠ q |]
  ==> (Legendre p q) * (Legendre q p) =
    (-1::int) ^ nat(((p - 1) div 2) * ((q - 1) div 2))
  by (auto simp add: QRTEMP.QR-short zprime-zOdd-eq-grt-2 [symmetric]
    QRTEMP-def)

```

end

19 Pocklington's Theorem for Primes

```

theory Pocklington
imports Main Primes
begin

```

```

definition modeq:: nat => nat => nat => bool  ((1[- = -] '(mod -)))
  where [a = b] (mod p) == ((a mod p) = (b mod p))

```

```

definition modneq:: nat => nat => nat => bool  ((1[- ≠ -] '(mod -)))
  where [a ≠ b] (mod p) == ((a mod p) ≠ (b mod p))

```

lemma modeq-trans:

```

  [| [a = b] (mod p); [b = c] (mod p) |] ==> [a = c] (mod p)
  by (simp add: modeq-def)

```

```

lemma nat-mod-lemma: assumes xyn: [x = y] (mod n) and xy:y ≤ x
  shows ∃ q. x = y + n * q

```

using *xyn xy unfolding modeq-def using nat-mod-eq-lemma by blast*

lemma *nat-mod[algebra]: $[x = y] \text{ (mod } n) \longleftrightarrow (\exists q1\ q2. x + n * q1 = y + n * q2)$*
unfolding *modeq-def nat-mod-eq-iff ..*

lemma *prime: $\text{prime } p \longleftrightarrow p \neq 0 \wedge p \neq 1 \wedge (\forall m. 0 < m \wedge m < p \longrightarrow \text{coprime } p\ m)$*

(is *?lhs \longleftrightarrow ?rhs*

proof –

{assume *p=0 \vee p=1 hence ?thesis using prime-0 prime-1 by (cases p=0, simp-all)}*

moreover

{assume *p0: p \neq 0 p \neq 1*

{assume *H: ?lhs*

{fix *m assume m: m > 0 m < p*

{assume *m=1 hence coprime p m by simp}*

moreover

{assume *p dvd m hence p \leq m using dvd-imp-le m by blast with m(2) have coprime p m by simp}*

ultimately have *coprime p m using prime-coprime[OF H, of m] by blast*
hence *?rhs using p0 by auto}*

moreover

{ assume *H: $\forall m. 0 < m \wedge m < p \longrightarrow \text{coprime } p\ m$*

from *prime-factor[OF p0(2)] obtain q where q: prime q q dvd p by blast*

from *prime-ge-2[OF q(1)] have q0: q > 0 by arith*

from *dvd-imp-le[OF q(2)] p0 have qp: q \leq p by arith*

{assume *q = p hence ?lhs using q(1) by blast}*

moreover

{assume *q \neq p with qp have qplt: q < p by arith*

from *H[rule-format, of q] qplt q0 have coprime p q by arith*

with *coprime-prime[of p q q] q have False by simp hence ?lhs by blast}*

ultimately have *?lhs by blast}*

ultimately have *?thesis by blast}*

ultimately show *?thesis by (cases p=0 \vee p=1, auto)*

qed

lemma *finite-number-segment: $\text{card } \{ m. 0 < m \wedge m < n \} = n - 1$*

proof –

have *$\{ m. 0 < m \wedge m < n \} = \{1..<n\}$ by auto*

thus *?thesis by simp*

qed

lemma *coprime-mod: assumes n: n \neq 0 shows $\text{coprime } (a \text{ mod } n)\ n \longleftrightarrow \text{coprime } a\ n$*

using *n dvd-mod-iff[of - n a] by (auto simp add: coprime)*

lemma *cong-mod-01* [*simp,presburger*]:

$[x = y] \text{ (mod } 0) \longleftrightarrow x = y$ $[x = y] \text{ (mod } 1) \longleftrightarrow [x = 0] \text{ (mod } n) \longleftrightarrow n \text{ dvd } x$
by (*simp-all add: modeq-def, presburger*)

lemma *cong-sub-cases*:

$[x = y] \text{ (mod } n) \longleftrightarrow (\text{if } x \leq y \text{ then } [y - x = 0] \text{ (mod } n) \text{ else } [x - y = 0] \text{ (mod } n))$

apply (*auto simp add: nat-mod*)
apply (*rule-tac x=q2 in exI*)
apply (*rule-tac x=q1 in exI, simp*)
apply (*rule-tac x=q2 in exI*)
apply (*rule-tac x=q1 in exI, simp*)
apply (*rule-tac x=q1 in exI*)
apply (*rule-tac x=q2 in exI, simp*)
apply (*rule-tac x=q1 in exI*)
apply (*rule-tac x=q2 in exI, simp*)
done

lemma *cong-mult-lcancel*: **assumes** *an: coprime a n* **and** *axy:[a * x = a * y]* $\text{ (mod } n)$

shows $[x = y] \text{ (mod } n)$

proof–

{**assume** $a = 0$ **with** *an axy coprime-0'* $\text{[of } n]$ **have** *?thesis* **by** (*simp add: modeq-def*) }

moreover

{**assume** *az: a ≠ 0*

{**assume** *xy: x ≤ y* **hence** *axy': a*x ≤ a*y* **by** *simp*
with *axy cong-sub-cases* $\text{[of } a*x \ a*y \ n]$ **have** $[a*(y - x) = 0] \text{ (mod } n)$
by (*simp only: if-True diff-mult-distrib2*)
hence *th: n dvd a*(y - x)* **by** *simp*
from *coprime-divprod* [OF th] *an* **have** $n \text{ dvd } y - x$
by (*simp add: coprime-commute*)
hence *?thesis* **using** *xy cong-sub-cases* $\text{[of } x \ y \ n]$ **by** *simp*}

moreover

{**assume** *H: ¬x ≤ y* **hence** *xy: y ≤ x* **by** *arith*
from *H az* **have** *axy': ¬ a*x ≤ a*y* **by** *auto*
with *axy H cong-sub-cases* $\text{[of } a*x \ a*y \ n]$ **have** $[a*(x - y) = 0] \text{ (mod } n)$
by (*simp only: if-False diff-mult-distrib2*)
hence *th: n dvd a*(x - y)* **by** *simp*
from *coprime-divprod* [OF th] *an* **have** $n \text{ dvd } x - y$
by (*simp add: coprime-commute*)
hence *?thesis* **using** *xy cong-sub-cases* $\text{[of } x \ y \ n]$ **by** *simp*}

ultimately have *?thesis* **by** *blast*}

ultimately show *?thesis* **by** *blast*

qed

lemma *cong-mult-rcancel*: **assumes** *an: coprime a n* **and** *axy:[x*a = y*a]* $\text{ (mod } n)$

n)
shows $[x = y] \text{ (mod } n)$
using *cong-mult-lcancel*[*OF an axy[unfolded mult-commute[of -a]]*] .

lemma *cong-refl*: $[x = x] \text{ (mod } n)$ **by** (*simp add: modeq-def*)

lemma *eq-imp-cong*: $a = b \implies [a = b] \text{ (mod } n)$ **by** (*simp add: cong-refl*)

lemma *cong-commute*: $[x = y] \text{ (mod } n) \longleftrightarrow [y = x] \text{ (mod } n)$
by (*auto simp add: modeq-def*)

lemma *cong-trans*[*trans*]: $[x = y] \text{ (mod } n) \implies [y = z] \text{ (mod } n) \implies [x = z] \text{ (mod } n)$
by (*simp add: modeq-def*)

lemma *cong-add*: **assumes** $xx': [x = x'] \text{ (mod } n)$ **and** $yy': [y = y'] \text{ (mod } n)$
shows $[x + y = x' + y'] \text{ (mod } n)$
proof –
have $(x + y) \text{ mod } n = (x \text{ mod } n + y \text{ mod } n) \text{ mod } n$
by (*simp add: mod-add-left-eq[of x y n] mod-add-right-eq[of x mod n y n]*)
also have $\dots = (x' \text{ mod } n + y' \text{ mod } n) \text{ mod } n$ **using** $xx' yy'$ *modeq-def* **by** *simp*
also have $\dots = (x' + y') \text{ mod } n$
by (*simp add: mod-add-left-eq[of x' y' n] mod-add-right-eq[of x' mod n y' n]*)
finally show *?thesis* **unfolding** *modeq-def* .
qed

lemma *cong-mult*: **assumes** $xx': [x = x'] \text{ (mod } n)$ **and** $yy': [y = y'] \text{ (mod } n)$
shows $[x * y = x' * y'] \text{ (mod } n)$
proof –
have $(x * y) \text{ mod } n = (x \text{ mod } n) * (y \text{ mod } n) \text{ mod } n$
by (*simp add: mod-mult-left-eq[of x y n] mod-mult-right-eq[of x mod n y n]*)
also have $\dots = (x' \text{ mod } n) * (y' \text{ mod } n) \text{ mod } n$ **using** xx' [*unfolded modeq-def*]
 yy' [*unfolded modeq-def*] **by** *simp*
also have $\dots = (x' * y') \text{ mod } n$
by (*simp add: mod-mult-left-eq[of x' y' n] mod-mult-right-eq[of x' mod n y' n]*)
finally show *?thesis* **unfolding** *modeq-def* .
qed

lemma *cong-exp*: $[x = y] \text{ (mod } n) \implies [x^k = y^k] \text{ (mod } n)$
by (*induct k, auto simp add: cong-refl cong-mult*)

lemma *cong-sub*: **assumes** $xx': [x = x'] \text{ (mod } n)$ **and** $yy': [y = y'] \text{ (mod } n)$
and $yx: y \leq x$ **and** $yx': y' \leq x'$
shows $[x - y = x' - y'] \text{ (mod } n)$
proof –
{ fix $x a x' a' y b y' b'$
have $(x::nat) + a = x' + a' \implies y + b = y' + b' \implies y \leq x \implies y' \leq x'$
 $\implies (x - y) + (a + b') = (x' - y') + (a' + b)$ **by** *arith*
note $th = this$
from $xx' yy'$ **obtain** $q1 q2 q1' q2'$ **where** $q12: x + n*q1 = x' + n*q2$

and $q12'$: $y + n * q1' = y' + n * q2'$ **unfolding** *nat-mod* **by** *blast* +
from $th[OF\ q12\ q12'\ yx\ yx']$
have $(x - y) + n * (q1 + q2') = (x' - y') + n * (q2 + q1')$
by (*simp add: right-distrib*)
thus $?thesis$ **unfolding** *nat-mod* **by** *blast*
qed

lemma *cong-mult-lcancel-eq*: **assumes** an : *coprime a n*
shows $[a * x = a * y] \pmod n \longleftrightarrow [x = y] \pmod n$ (**is** $?lhs \longleftrightarrow ?rhs$)
proof
assume H : $?rhs$ **from** *cong-mult*[*OF cong-refl*[*of a n*] H] **show** $?lhs$.
next
assume H : $?lhs$ **hence** H' : $[x * a = y * a] \pmod n$ **by** (*simp add: mult-commute*)
from *cong-mult-rcancel*[*OF an H'*] **show** $?rhs$.
qed

lemma *cong-mult-rcancel-eq*: **assumes** an : *coprime a n*
shows $[x * a = y * a] \pmod n \longleftrightarrow [x = y] \pmod n$
using *cong-mult-lcancel-eq*[*OF an, of x y*] **by** (*simp add: mult-commute*)

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

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

lemma *cong-add-rcancel*: $[x + a = y + a] \pmod n \implies [x = y] \pmod n$
by (*simp add: nat-mod*)

lemma *cong-add-lcancel*: $[a + x = a + y] \pmod n \implies [x = y] \pmod n$
by (*simp add: nat-mod*)

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

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

lemma *cong-imp-eq*: **assumes** xn : $x < n$ **and** yn : $y < n$ **and** xy : $[x = y] \pmod n$
shows $x = y$
using xy [*unfolded modeq-def mod-less*[*OF xn*] *mod-less*[*OF yn*]] .

lemma *cong-divides-modulus*: $[x = y] \pmod m \implies n \text{ dvd } m \implies [x = y] \pmod n$
apply (*auto simp add: nat-mod dvd-def*)
apply (*rule-tac x=k*q1 in exI*)
apply (*rule-tac x=k*q2 in exI*)
by *simp*

lemma *cong-0-divides*: $[x = 0] \text{ (mod } n) \longleftrightarrow n \text{ dvd } x$ **by** *simp*

lemma *cong-1-divides*: $[x = 1] \text{ (mod } n) \implies n \text{ dvd } x - 1$
apply (*cases* $x \leq 1$, *simp-all*)
using *cong-sub-cases*[*of* $x \ 1 \ n$] **by** *auto*

lemma *cong-divides*: $[x = y] \text{ (mod } n) \implies n \text{ dvd } x \longleftrightarrow n \text{ dvd } y$
apply (*auto simp add: nat-mod dvd-def*)
apply (*rule-tac* $x=k + q1 - q2$ **in** *exI*, *simp add: add-mult-distrib2 diff-mult-distrib2*)
apply (*rule-tac* $x=k + q2 - q1$ **in** *exI*, *simp add: add-mult-distrib2 diff-mult-distrib2*)
done

lemma *cong-coprime*: **assumes** $xy: [x = y] \text{ (mod } n)$
shows $\text{coprime } n \ x \longleftrightarrow \text{coprime } n \ y$
proof –
 {**assume** $n=0$ **hence** *?thesis* **using** xy **by** *simp*}
 moreover
 {**assume** $nz: n \neq 0$
 have $\text{coprime } n \ x \longleftrightarrow \text{coprime } (x \text{ mod } n) \ n$
 by (*simp add: coprime-mod*[*OF* nz , *of* x] *coprime-commute*[*of* $n \ x$])
 also have $\dots \longleftrightarrow \text{coprime } (y \text{ mod } n) \ n$ **using** xy [*unfolded modeq-def*] **by** *simp*
 also have $\dots \longleftrightarrow \text{coprime } y \ n$ **by** (*simp add: coprime-mod*[*OF* nz , *of* y])
 finally have *?thesis* **by** (*simp add: coprime-commute*) }
ultimately show *?thesis* **by** *blast*
qed

lemma *cong-mod*: $\sim(n = 0) \implies [a \text{ mod } n = a] \text{ (mod } n)$ **by** (*simp add: modeq-def*)

lemma *mod-mult-cong*: $\sim(a = 0) \implies \sim(b = 0)$
 $\implies [x \text{ mod } (a * b) = y] \text{ (mod } a) \longleftrightarrow [x = y] \text{ (mod } a)$
by (*simp add: modeq-def mod-mult2-eq mod-add-left-eq*)

lemma *cong-mod-mult*: $[x = y] \text{ (mod } n) \implies m \text{ dvd } n \implies [x = y] \text{ (mod } m)$
apply (*auto simp add: nat-mod dvd-def*)
apply (*rule-tac* $x=k*q1$ **in** *exI*)
apply (*rule-tac* $x=k*q2$ **in** *exI*, *simp*)
done

lemma *cong-le*: $y \leq x \implies [x = y] \text{ (mod } n) \longleftrightarrow (\exists q. x = q * n + y)$
using *nat-mod-lemma*[*of* $x \ y \ n$]
apply *auto*
apply (*simp add: nat-mod*)
apply (*rule-tac* $x=q$ **in** *exI*)
apply (*rule-tac* $x=q + q$ **in** *exI*)
by (*auto simp: algebra-simps*)

lemma *cong-to-1*: $[a = 1] \text{ (mod } n) \longleftrightarrow a = 0 \wedge n = 1 \vee (\exists m. a = 1 + m * n)$
proof–
 {**assume** $n = 0 \vee n = 1 \vee a = 0 \vee a = 1$ **hence** *?thesis*
 apply (*cases n=0, simp-all add: cong-commute*)
 apply (*cases n=1, simp-all add: cong-commute modeq-def*)
 apply *arith*
 by (*cases a=1, simp-all add: modeq-def cong-commute*)}
moreover
 {**assume** $n: n \neq 0 \wedge n \neq 1$ **and** $a: a \neq 0 \wedge a \neq 1$ **hence** $a': a \geq 1$ **by** *simp*
 hence *?thesis* **using** *cong-le[OF a', of n]* **by** *auto* }
ultimately show *?thesis* **by** *auto*
qed

lemma *cong-solve*: **assumes** *an: coprime a n* **shows** $\exists x. [a * x = b] \text{ (mod } n)$
proof–
 {**assume** $a=0$ **hence** *?thesis* **using** *an* **by** (*simp add: modeq-def*)}
moreover
 {**assume** $az: a \neq 0$
 from *bezout-add-strong[OF az, of n]*
 obtain $d \ x \ y$ **where** $dxy: d \text{ dvd } a \wedge d \text{ dvd } n \wedge a * x = n * y + d$ **by** *blast*
 from *an[unfolded coprime, rule-format, of d] dxy(1,2)* **have** $d1: d = 1$ **by** *blast*
 hence $a * x * b = (n * y + 1) * b$ **using** $dxy(3)$ **by** *simp*
 hence $a * (x * b) = n * (y * b) + b$ **by** *algebra*
 hence $a * (x * b) \text{ mod } n = (n * (y * b) + b) \text{ mod } n$ **by** *simp*
 hence $a * (x * b) \text{ mod } n = b \text{ mod } n$ **by** (*simp add: mod-add-left-eq*)
 hence $[a * (x * b) = b] \text{ (mod } n)$ **unfolding** *modeq-def* .
 hence *?thesis* **by** *blast* }
ultimately show *?thesis* **by** *blast*
qed

lemma *cong-solve-unique*: **assumes** *an: coprime a n* **and** $nz: n \neq 0$
shows $\exists! x. x < n \wedge [a * x = b] \text{ (mod } n)$
proof–
let $?P = \lambda x. x < n \wedge [a * x = b] \text{ (mod } n)$
from *cong-solve[OF an]* **obtain** x **where** $x: [a * x = b] \text{ (mod } n)$ **by** *blast*
let $?x = x \text{ mod } n$
from x **have** $th: [a * ?x = b] \text{ (mod } n)$
 by (*simp add: modeq-def mod-mult-right-eq[of a x n]*)
from *mod-less-divisor[of n x] nz th* **have** $Px: ?P \ ?x$ **by** *simp*
{fix y **assume** $Py: y < n \wedge [a * y = b] \text{ (mod } n)$
 from $Py(2)$ **th** **have** $[a * y = a * ?x] \text{ (mod } n)$ **by** (*simp add: modeq-def*)
 hence $[y = ?x] \text{ (mod } n)$ **by** (*simp add: cong-mult-lcancel-eq[OF an]*)
 with *mod-less[OF Py(1)] mod-less-divisor[of n x] nz*
 have $y = ?x$ **by** (*simp add: modeq-def*) }
with Px **show** *?thesis* **by** *blast*
qed

lemma *cong-solve-unique-nontrivial*:

assumes p : prime p and pa : coprime p a and $x0$: $0 < x$ and xp : $x < p$
 shows $\exists!y. 0 < y \wedge y < p \wedge [x * y = a] \pmod{p}$
proof–
 from p have $p1$: $p > 1$ using *prime-ge-2*[*OF* p] by *arith*
 hence $p01$: $p \neq 0$ $p \neq 1$ by *arith+*
 from pa have ap : coprime a p by (*simp add: coprime-commute*)
 from *prime-coprime*[*OF* p , *of* x] *dvd-imp-le*[*of* p x] $x0$ xp have px : coprime x p
 by (*auto simp add: coprime-commute*)
 from *cong-solve-unique*[*OF* px $p01(1)$]
 obtain y where y : $y < p$ $[x * y = a] \pmod{p}$ $\forall z. z < p \wedge [x * z = a] \pmod{p}$
 $p) \longrightarrow z = y$ by *blast*
 {assume $y0$: $y = 0$
 with $y(2)$ have th : p *dvd* a by (*simp add: cong-commute*[*of* 0 a p])
 with p *coprime-prime*[*OF* pa , *of* p] have *False* by *simp*}
 with y show *?thesis* unfolding *Ex1-def* using *neg0-conv* by *blast*
qed
lemma *cong-unique-inverse-prime*:
 assumes p : prime p and $x0$: $0 < x$ and xp : $x < p$
 shows $\exists!y. 0 < y \wedge y < p \wedge [x * y = 1] \pmod{p}$
 using *cong-solve-unique-nontrivial*[*OF* p *coprime-1*[*of* p] $x0$ xp] .

lemma *cong-chinese*:

assumes ab : coprime a b and xya : $[x = y] \pmod{a}$
 and xyb : $[x = y] \pmod{b}$
 shows $[x = y] \pmod{a*b}$
 using ab xya xyb
 by (*simp add: cong-sub-cases*[*of* x y a] *cong-sub-cases*[*of* x y b]
cong-sub-cases[*of* x y $a*b$])
(cases $x \leq y$, *simp-all add: divides-mul*[*of* $a - b$])

lemma *chinese-remainder-unique*:

assumes ab : coprime a b and az : $a \neq 0$ and bz : $b \neq 0$
 shows $\exists!x. x < a * b \wedge [x = m] \pmod{a} \wedge [x = n] \pmod{b}$
proof–
 from az bz have $abpos$: $a*b > 0$ by *simp*
 from *chinese-remainder*[*OF* ab az bz] obtain x $q1$ $q2$ where
 $xq12$: $x = m + q1 * a$ $x = n + q2 * b$ by *blast*
 let $?w = x \pmod{a*b}$
 have wab : $?w < a*b$ by (*simp add: mod-less-divisor*[*OF* $abpos$])
 from $xq12(1)$ have $?w \pmod{a} = ((m + q1 * a) \pmod{a*b}) \pmod{a}$ by *simp*
 also have $\dots = m \pmod{a}$ apply (*simp add: mod-mult2-eq*)
 apply (*subst mod-add-left-eq*)
 by *simp*
 finally have $th1$: $[?w = m] \pmod{a}$ by (*simp add: modeq-def*)
 from $xq12(2)$ have $?w \pmod{b} = ((n + q2 * b) \pmod{a*b}) \pmod{b}$ by *simp*

also have $\dots = ((n + q2 * b) \text{ mod } (b*a)) \text{ mod } b$ **by** (*simp add: mult-commute*)
also have $\dots = n \text{ mod } b$ **apply** (*simp add: mod-mult2-eq*)
apply (*subst mod-add-left-eq*)
by *simp*
finally have $th2: [?w = n] \text{ (mod } b)$ **by** (*simp add: modeq-def*)
{fix y assume $H: y < a*b$ $[y = m] \text{ (mod } a)$ $[y = n] \text{ (mod } b)$
with $th1$ $th2$ **have** $H': [y = ?w] \text{ (mod } a)$ $[y = ?w] \text{ (mod } b)$
by (*simp-all add: modeq-def*)
from *cong-chinese*[*OF ab H*] *mod-less*[*OF H(1)*] *mod-less*[*OF wab*]
have $y = ?w$ **by** (*simp add: modeq-def*)
with $th1$ $th2$ wab **show** $?thesis$ **by** *blast*
qed

lemma *chinese-remainder-coprime-unique*:
assumes $ab: \text{coprime } a \ b$ **and** $az: a \neq 0$ **and** $bz: b \neq 0$
and $ma: \text{coprime } m \ a$ **and** $nb: \text{coprime } n \ b$
shows $\exists!x. \text{coprime } x \ (a * b) \wedge x < a * b \wedge [x = m] \text{ (mod } a) \wedge [x = n] \text{ (mod } b)$
proof–
let $?P = \lambda x. x < a * b \wedge [x = m] \text{ (mod } a) \wedge [x = n] \text{ (mod } b)$
from *chinese-remainder-unique*[*OF ab az bz*]
obtain x **where** $x: x < a * b$ $[x = m] \text{ (mod } a)$ $[x = n] \text{ (mod } b)$
 $\forall y. ?P \ y \longrightarrow y = x$ **by** *blast*
from $ma \ nb$ *cong-coprime*[*OF x(2)*] *cong-coprime*[*OF x(3)*]
have $\text{coprime } x \ a$ $\text{coprime } x \ b$ **by** (*simp-all add: coprime-commute*)
with *coprime-mul*[*of x a b*] **have** $\text{coprime } x \ (a*b)$ **by** *simp*
with x **show** $?thesis$ **by** *blast*
qed

definition *phi-def*: $\varphi \ n = \text{card } \{ m. 0 < m \wedge m \leq n \wedge \text{coprime } m \ n \}$

lemma *phi-0*[*simp*]: $\varphi \ 0 = 0$
unfolding *phi-def* **by** *auto*

lemma *phi-finite*[*simp*]: *finite* ($\{ m. 0 < m \wedge m \leq n \wedge \text{coprime } m \ n \}$)
proof–
have $\{ m. 0 < m \wedge m \leq n \wedge \text{coprime } m \ n \} \subseteq \{0..n\}$ **by** *auto*
thus $?thesis$ **by** (*auto intro: finite-subset*)
qed

declare *coprime-1*[*presburger*]
lemma *phi-1*[*simp*]: $\varphi \ 1 = 1$
proof–
{fix m
have $0 < m \wedge m \leq 1 \wedge \text{coprime } m \ 1 \longleftrightarrow m = 1$ **by** *presburger* }
thus $?thesis$ **by** (*simp add: phi-def*)
qed

```

lemma [simp]:  $\varphi(\text{Suc } 0) = \text{Suc } 0$  using phi-1 by simp

lemma phi-alt:  $\varphi(n) = \text{card } \{ m. \text{coprime } m \ n \wedge m < n \}$ 
proof –
  {assume  $n=0 \vee n=1$  hence ?thesis by (cases  $n=0$ , simp-all)}
  moreover
  {assume  $n: n \neq 0 \ n \neq 1$ 
    {fix  $m$ 
      from  $n$  have  $0 < m \wedge m \leq n \wedge \text{coprime } m \ n \longleftrightarrow \text{coprime } m \ n \wedge m < n$ 
        apply (cases  $m = 0$ , simp-all)
        apply (cases  $m = 1$ , simp-all)
        apply (cases  $m = n$ , auto)
        done }
      hence ?thesis unfolding phi-def by simp}
  ultimately show ?thesis by auto
qed

lemma phi-finite-lemma[simp]:  $\text{finite } \{ m. \text{coprime } m \ n \wedge m < n \}$  (is finite ?S)
by (rule finite-subset[of ?S {0..n}], auto)

lemma phi-another: assumes  $n: n \neq 1$ 
  shows  $\varphi \ n = \text{card } \{ m. 0 < m \wedge m < n \wedge \text{coprime } m \ n \}$ 
proof –
  {fix  $m$ 
    from  $n$  have  $0 < m \wedge m < n \wedge \text{coprime } m \ n \longleftrightarrow \text{coprime } m \ n \wedge m < n$ 
      by (cases  $m=0$ , auto)}
  thus ?thesis unfolding phi-alt by auto
qed

lemma phi-limit:  $\varphi \ n \leq n$ 
proof –
  have  $\{ m. \text{coprime } m \ n \wedge m < n \} \subseteq \{ 0 ..<n \}$  by auto
  with card-mono[of {0 ..<n} {  $m. \text{coprime } m \ n \wedge m < n$  } ]
  show ?thesis unfolding phi-alt by auto
qed

lemma stupid[simp]:  $\{ m. (0::\text{nat}) < m \wedge m < n \} = \{ 1 ..<n \}$ 
by auto

lemma phi-limit-strong: assumes  $n: n \neq 1$ 
  shows  $\varphi(n) \leq n - 1$ 
proof –
  show ?thesis
    unfolding phi-another[OF  $n$ ] finite-number-segment[of  $n$ , symmetric]
    by (rule card-mono[of {  $m. 0 < m \wedge m < n$  } {  $m. 0 < m \wedge m < n \wedge \text{coprime } m \ n$  } ], auto)
qed

```

```

lemma phi-lowerbound-1-strong: assumes  $n: n \geq 1$ 
  shows  $\varphi(n) \geq 1$ 
proof –
  let  $?S = \{ m. 0 < m \wedge m \leq n \wedge \text{coprime } m \ n \}$ 
  from card-0-eq[of  $?S$ ]  $n$  have  $\varphi \ n \neq 0$  unfolding phi-alt
    apply auto
    apply (cases  $n=1$ , simp-all)
    apply (rule exI[where  $x=1$ ], simp)
    done
  thus ?thesis by arith
qed

lemma phi-lowerbound-1:  $2 \leq n \implies 1 \leq \varphi(n)$ 
  using phi-lowerbound-1-strong[of  $n$ ] by auto

lemma phi-lowerbound-2: assumes  $n: 3 \leq n$  shows  $2 \leq \varphi(n)$ 
proof –
  let  $?S = \{ m. 0 < m \wedge m \leq n \wedge \text{coprime } m \ n \}$ 
  have  $\text{inS}: \{1, n-1\} \subseteq ?S$  using n coprime-plus1[of  $n-1$ ]
    by (auto simp add: coprime-commute)
  from  $n$  have  $c2: \text{card } \{1, n-1\} = 2$  by (auto simp add: card-insert-if)
  from card-mono[of  $?S \ \{1, n-1\}$ , simplified inS c2] show ?thesis
    unfolding phi-def by auto
qed

lemma phi-prime:  $\varphi \ n = n - 1 \wedge n \neq 0 \wedge n \neq 1 \iff \text{prime } n$ 
proof –
  {assume  $n=0 \vee n=1$  hence ?thesis by (cases  $n=1$ , simp-all)}
  moreover
  {assume  $n: n \neq 0 \ n \neq 1$ 
    let  $?S = \{m. 0 < m \wedge m < n\}$ 
    have  $fS: \text{finite } ?S$  by simp
    let  $?S' = \{m. 0 < m \wedge m < n \wedge \text{coprime } m \ n\}$ 
    have  $fS': \text{finite } ?S'$  apply (rule finite-subset[of  $?S' \ ?S$ ]) by auto
    {assume  $H: \varphi \ n = n - 1 \wedge n \neq 0 \wedge n \neq 1$ 
      hence ceq:  $\text{card } ?S' = \text{card } ?S$ 
      using n finite-number-segment[of  $n$ ] phi-another[OF  $n(2)$ ] by simp
      {fix  $m$  assume  $m: 0 < m \wedge m < n \wedge \neg \text{coprime } m \ n$ 
        hence  $mS': m \notin ?S'$  by auto
        have  $\text{insert } m \ ?S' \leq ?S$  using  $m$  by auto
        from  $m$  have  $\text{card } (\text{insert } m \ ?S') \leq \text{card } ?S$ 
        by – (rule card-mono[of  $?S \ \text{insert } m \ ?S'$ ], auto)
        hence False
        unfolding card-insert-disjoint[of  $?S' \ m$ , OF  $fS' \ mS'$ ] ceq
        by simp }
      hence  $\forall m. 0 < m \wedge m < n \implies \text{coprime } m \ n$  by blast
      hence prime n unfolding prime using  $n$  by (simp add: coprime-commute)}
    moreover
    {assume  $H: \text{prime } n$ 
```


hence $?S = ?S'$ **unfolding prime using n**
 by (auto simp add: coprime-commute)
 hence $\text{card } ?S = \text{card } ?S'$ **by simp**
 hence $\varphi n = n - 1$ **unfolding phi-another[OF $n(2)$] by simp**
 ultimately have $?thesis$ **using n by blast**
 ultimately show $?thesis$ **by (cases $n=0$) blast+**
 qed

lemma phi-multiplicative: assumes ab : coprime a b
shows $\varphi (a * b) = \varphi a * \varphi b$
proof –
 {assume $a = 0 \vee b = 0 \vee a = 1 \vee b = 1$
 hence $?thesis$
 by (cases $a=0$, simp, cases $b=0$, simp, cases $a=1$, simp-all) }
moreover
 {assume a : $a \neq 0$ $a \neq 1$ and b : $b \neq 0$ $b \neq 1$
 hence $ab0$: $a * b \neq 0$ **by simp**
 let $?S = \lambda k. \{m. \text{coprime } m \ k \wedge m < k\}$
 let $?f = \lambda x. (x \bmod a, x \bmod b)$
 have eq : $?f ' (?S (a * b)) = (?S a \times ?S b)$
proof –
 {fix x assume $x: x \in ?S (a * b)$
 hence x' : coprime $x (a * b)$ $x < a * b$ **by simp-all**
 hence xab : coprime x a coprime x b **by (simp-all add: coprime-mul-eq)**
 from mod-less-divisor a b have xab' : $x \bmod a < a$ $x \bmod b < b$ **by auto**
 from xab xab' have $?f x \in (?S a \times ?S b)$
 by (simp add: coprime-mod[OF $a(1)$] coprime-mod[OF $b(1)$]) }
moreover
 {fix x y assume $x: x \in ?S a$ and $y: y \in ?S b$
 hence x' : coprime x a $x < a$ and y' : coprime y b $y < b$ **by simp-all**
 from chinese-remainder-coprime-unique[OF ab $a(1)$ $b(1)$ $x'(1)$ $y'(1)$]
 obtain z where z : coprime $z (a * b)$ $z < a * b$ $[z = x] \pmod{a}$
 $[z = y] \pmod{b}$ **by blast**
 hence $(x, y) \in ?f ' (?S (a * b))$
 using $y'(2)$ mod-less-divisor[of b y] $x'(2)$ mod-less-divisor[of a x]
 by (auto simp add: image-iff modeq-def) }
 ultimately show $?thesis$ **by auto**
 qed
 have $finj$: $\text{inj-on } ?f (?S (a * b))$
 unfolding inj-on-def
proof (clarify)
 fix x y assume H : coprime $x (a * b)$ $x < a * b$ coprime $y (a * b)$
 $y < a * b$ $x \bmod a = y \bmod a$ $x \bmod b = y \bmod b$
 hence cp : coprime x a coprime x b coprime y a coprime y b
 by (simp-all add: coprime-mul-eq)
 from chinese-remainder-coprime-unique[OF ab $a(1)$ $b(1)$ $cp(3,4)$] H
 show $x = y$ **unfolding modeq-def by blast**

qed
from *card-image*[*OF finj*, *unfolded eq*] **have** *?thesis*
unfolding *phi-alt* **by** *simp* }
ultimately show *?thesis* **by** *auto*
qed

lemma *nproduct-mod*:
assumes *fS*: *finite S* **and** *n0*: $n \neq 0$
shows $[\text{setprod } (\lambda m. a(m) \bmod n) S = \text{setprod } a S] \bmod n$
proof –
have *th1*: $[1 = 1] \bmod n$ **by** (*simp add: modeq-def*)
from *cong-mult*
have *th3*: $\forall x1\ y1\ x2\ y2. [x1 = x2] \bmod n \wedge [y1 = y2] \bmod n \longrightarrow [x1 * y1 = x2 * y2] \bmod n$
by *blast*
have *th4*: $\forall x \in S. [a\ x \bmod n = a\ x] \bmod n$ **by** (*simp add: modeq-def*)
from *fold-image-related* **where** *h* = $(\lambda m. a(m) \bmod n)$ **and** *g* = *a*, *OF th1 th3 fS*,
OF th4] **show** *?thesis* **unfolding** *setprod-def* **by** (*simp add: fS*)
qed

lemma *nproduct-cmul*:
assumes *fS*: *finite S*
shows $\text{setprod } (\lambda m. (c::'a::\{\text{comm-monoid-mult}\}) * a(m)) S = c ^ (\text{card } S) * \text{setprod } a S$
unfolding *setprod-timesf setprod-constant*[*OF fS*, *of c*] ..

lemma *coprime-nproduct*:
assumes *fS*: *finite S* **and** *Sn*: $\forall x \in S. \text{coprime } n (a\ x)$
shows $\text{coprime } n (\text{setprod } a S)$
using *fS* **unfolding** *setprod-def* **by** (*rule finite-subset-induct*)
(insert Sn, auto simp add: coprime-mul)

lemma *fermat-little*: **assumes** *an*: $\text{coprime } a\ n$
shows $[a ^ (\varphi\ n) = 1] \bmod n$
proof –
{assume *n=0* **hence** *?thesis* **by** *simp* }
moreover
{assume *n=1* **hence** *?thesis* **by** (*simp add: modeq-def*) }
moreover
{assume *nz*: $n \neq 0$ **and** *n1*: $n \neq 1$
let *?S* = $\{m. \text{coprime } m\ n \wedge m < n\}$
let *?P* = $\prod ?S$
have *fS*: *finite ?S* **by** *simp*
have *cardfS*: $\varphi\ n = \text{card } ?S$ **unfolding** *phi-alt* ..
{fix *m* **assume** *m*: $m \in ?S$
hence $\text{coprime } m\ n$ **by** *simp*

```

with coprime-mul[of n a m] an have coprime (a*m) n
  by (simp add: coprime-commute)}
hence Sn:  $\forall m \in ?S. \text{coprime } (a*m) \ n$  by blast
from coprime-nproduct[OF fS, of n  $\lambda m. m$ ] have nP: coprime ?P n
  by (simp add: coprime-commute)
have Paphi:  $[?P*a \wedge (\varphi \ n) = ?P*1] \pmod n$ 
proof-
  let ?h =  $\lambda m. m \pmod n$ 
  {fix m assume mS:  $m \in ?S$ 
   hence ?h m  $\in ?S$  by simp}
  hence hS:  $?h \circ ?S = ?S$  by (auto simp add: image-iff)
  have a $\neq 0$  using an n1 nz apply- apply (rule ccontr) by simp
  hence inj: inj-on (op * a) ?S unfolding inj-on-def by simp

  have eq0: fold-image op * ( $?h \circ op * a$ ) 1  $\{m. \text{coprime } m \ n \wedge m < n\} =$ 
    fold-image op * ( $\lambda m. m$ ) 1  $\{m. \text{coprime } m \ n \wedge m < n\}$ 
  proof (rule fold-image-eq-general[where h= $?h \circ (op * a)$ ])
    show finite ?S using fS .
  next
    {fix y assume yS:  $y \in ?S$  hence y: coprime y n  $y < n$  by simp-all
     from cong-solve-unique[OF an nz, of y]
     obtain x where x:  $x < n \ [a * x = y] \pmod n \ \forall z. z < n \wedge [a * z = y]$ 
    (mod n)  $\longrightarrow z=x$  by blast
     from cong-coprime[OF x(2)] y(1)
     have xm: coprime x n by (simp add: coprime-mul-eq coprime-commute)
     {fix z assume z  $\in ?S$  ( $?h \circ op * a$ )  $z = y$ 
      hence z: coprime z n  $z < n$  ( $?h \circ op * a$ )  $z = y$  by simp-all
      from x(3)[rule-format, of z] z(2,3) have z=x
        unfolding modeq-def mod-less[OF y(2)] by simp}
     with xm x(1,2) have  $\exists!x. x \in ?S \wedge (?h \circ op * a) \ x = y$ 
     unfolding modeq-def mod-less[OF y(2)] by auto }
    thus  $\forall y \in \{m. \text{coprime } m \ n \wedge m < n\}. \exists!x. x \in \{m. \text{coprime } m \ n \wedge m < n\} \wedge ((\lambda m. m \pmod n) \circ op * a) \ x = y$ 
  by blast
  next
    {fix x assume xS:  $x \in ?S$ 
     hence x: coprime x n  $x < n$  by simp-all
     with an have coprime (a*x) n
       by (simp add: coprime-mul-eq[of n a x] coprime-commute)
     hence ?h (a*x)  $\in ?S$  using nz
       by (simp add: coprime-mod[OF nz] mod-less-divisor)}
    thus  $\forall x \in \{m. \text{coprime } m \ n \wedge m < n\}. ((\lambda m. m \pmod n) \circ op * a) \ x \in \{m. \text{coprime } m \ n \wedge m < n\} \wedge$ 
       $((\lambda m. m \pmod n) \circ op * a) \ x = ((\lambda m. m \pmod n) \circ op * a) \ x$  by simp
  qed
  from nproduct-mod[OF fS nz, of op * a]
  have  $[(\text{setprod } (op * a) \ ?S) = (\text{setprod } (?h \circ (op * a)) \ ?S)] \pmod n$ 
  unfolding o-def
  by (simp add: cong-commute)

```

```

    also have [setprod (?h o (op * a)) ?S = ?P ] (mod n)
    using eq0 fS an by (simp add: setprod-def modeq-def o-def)
    finally show [?P*a^(φ n) = ?P*1] (mod n)
    unfolding cardfS mult-commute[of ?P a^(card ?S)]
    nproduct-cmul[OF fS, symmetric] mult-1-right by simp
  qed
  from cong-mult-lcancel[OF nP Paphi] have ?thesis . }
  ultimately show ?thesis by blast
qed

```

```

lemma fermat-little-prime: assumes p: prime p and ap: coprime a p
  shows [a^(p - 1) = 1] (mod p)
  using fermat-little[OF ap] p[unfolded phi-prime[symmetric]]
  by simp

```

```

lemma lucas-coprime-lemma:
  assumes m: m≠0 and am: [a^m = 1] (mod n)
  shows coprime a n
proof-
  {assume n=1 hence ?thesis by simp}
  moreover
  {assume n = 0 hence ?thesis using am m exp-eq-1[of a m] by simp}
  moreover
  {assume n: n≠0 n≠1
    from m obtain m' where m': m = Suc m' by (cases m, blast+)
    {fix d
      assume d: d dvd a d dvd n
      from n have n1: 1 < n by arith
      from am mod-less[OF n1] have am1: a^m mod n = 1 unfolding modeq-def
    by simp
      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
      have d = 1 by simp }
    hence ?thesis unfolding coprime by auto
  }
  ultimately show ?thesis by blast
qed

```

```

lemma lucas-weak:
  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-
  from n have n1: n ≠ 1 n≠0 n - 1 ≠ 0 n - 1 > 0 n - 1 < n by arith+
  from lucas-coprime-lemma[OF n1(3) an] have can: coprime a n .
  from fermat-little[OF can] have afn: [a^φ n = 1] (mod n) .

```

```

{assume  $\varphi\ n \neq n - 1$ 
  with phi-limit-strong[OF n1(1)] phi-lowerbound-1[OF n]
  have  $c:\varphi\ n > 0 \wedge \varphi\ n < n - 1$  by arith
  from nm[rule-format, OF c] afn have False ..}
hence  $\varphi\ n = n - 1$  by blast
with phi-prime[of n] n1(1,2) show ?thesis by simp
qed

```

```

lemma nat-exists-least-iff:  $(\exists (n::nat). P\ n) \longleftrightarrow (\exists n. P\ n \wedge (\forall m < n. \neg P\ m))$ 
(is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume ?rhs thus ?lhs by blast
next
  assume H: ?lhs then obtain n where  $n: P\ n$  by blast
  let ?x = Least P
  {fix m assume  $m: m < ?x$ 
    from not-less-Least[OF m] have  $\neg P\ m$  .}
  with LeastI-ex[OF H] show ?rhs by blast
qed

```

```

lemma nat-exists-least-iff':  $(\exists (n::nat). P\ n) \longleftrightarrow (P\ (Least\ P) \wedge (\forall m < (Least\ P). \neg P\ m))$ 
(is ?lhs  $\longleftrightarrow$  ?rhs)
proof-
  {assume ?rhs hence ?lhs by blast}
  moreover
  { assume H: ?lhs then obtain n where  $n: P\ n$  by blast
    let ?x = Least P
    {fix m assume  $m: m < ?x$ 
      from not-less-Least[OF m] have  $\neg P\ m$  .}
    with LeastI-ex[OF H] have ?rhs by blast}
  ultimately show ?thesis by blast
qed

```

```

lemma power-mod:  $((x::nat)\ mod\ m)^n\ mod\ m = x^n\ mod\ m$ 
proof(induct n)
  case 0 thus ?case by simp
next
  case (Suc n)
  have  $(x\ mod\ m)^{(Suc\ n)\ mod\ m} = ((x\ mod\ m) * ((x\ mod\ m)^n\ mod\ m))\ mod\ m$ 
  by (simp add: mod-mult-right-eq[symmetric])
  also have  $\dots = ((x\ mod\ m) * (x^n\ mod\ m))\ mod\ m$  using Suc.hyps by simp
  also have  $\dots = x^n\ mod\ m$ 
  by (simp add: mod-mult-left-eq[symmetric] mod-mult-right-eq[symmetric])
  finally show ?case .
qed

```

```

lemma lucas:

```

```

assumes n2:  $n \geq 2$  and an1:  $[a^{(n-1)} = 1] \pmod n$ 
and pn:  $\forall p. \text{prime } p \wedge p \text{ dvd } n - 1 \longrightarrow \neg [a^{((n-1) \text{ div } p)} = 1] \pmod n$ 
shows prime n
proof -
  from n2 have n01:  $n \neq 0 \ n \neq 1 \ n - 1 \neq 0$  by arith+
  from mod-less-divisor[of n 1] n01 have onen:  $1 \pmod n = 1$  by simp
  from lucas-coprime-lemma[OF n01(3) an1] cong-coprime[OF an1]
  have an: coprime a n coprime (a(n-1)) n by (simp-all add: coprime-commute)
  {assume H0:  $\exists m. 0 < m \wedge m < n - 1 \wedge [a^m = 1] \pmod n$  (is EX m. ?P
m)
    from H0[unfolded nat-exists-least-iff[of ?P]] obtain m where
      m:  $0 < m \wedge m < n - 1 \wedge [a^m = 1] \pmod n \wedge \forall k < m. \neg ?P k$  by blast
    {assume nm1:  $(n - 1) \pmod m > 0$ 
      from mod-less-divisor[OF m(1)] have th0:  $(n - 1) \pmod m < m$  by blast
      let ?y =  $a^{((n-1) \text{ div } m * m)}$ 
      note mdeq = mod-div-equality[of (n - 1) m]
      from coprime-exp[OF an(1)[unfolded coprime-commute[of a n]],
        of (n - 1) div m * m]
      have yn: coprime ?y n by (simp add: coprime-commute)
      have ?y mod n =  $(a^m)^{((n-1) \text{ div } m)} \pmod n$ 
        by (simp add: algebra-simps power-mult)
      also have ... =  $(a^m \pmod n)^{((n-1) \text{ div } m)} \pmod n$ 
        using power-mod[of a^m n (n - 1) div m] by simp
      also have ... = 1 using m(3)[unfolded modeq-def onen] onen
        by (simp add: power-Suc0)
      finally have th3: ?y mod n = 1 .
      have th2:  $[?y * a^{((n-1) \text{ div } m)} = ?y * 1] \pmod n$ 
        using an1[unfolded modeq-def onen] onen
        mod-div-equality[of (n - 1) m, symmetric]
        by (simp add: power-add[symmetric] modeq-def th3 del: One-nat-def)
      from cong-mult-1cancel[of ?y n a((n-1) div m) 1, OF yn th2]
      have th1:  $[a^{((n-1) \text{ div } m)} = 1] \pmod n$  .
      from m(4)[rule-format, OF th0] nm1
        less-trans[OF mod-less-divisor[OF m(1), of n - 1] m(2)] th1
      have False by blast }
    hence  $(n - 1) \pmod m = 0$  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:  $r \neq 0 \ r \neq 1$  by - (rule ccontr, simp)+
    from prime-factor[OF r01(2)] obtain p where p: prime p p dvd r by blast
    hence th: prime p  $\wedge p \text{ dvd } n - 1$  unfolding r by (auto intro: dvd-mult)
    have  $(a^{((n-1) \text{ div } p)}) \pmod n = (a^{(m*r \text{ div } p)}) \pmod n$  using r
      by (simp add: power-mult)
    also have ... =  $(a^{(m*(r \text{ div } p))}) \pmod n$  using div-mult1-eq[of m r p]
p(2)[unfolded dvd-eq-mod-eq-0] by simp
    also have ... =  $((a^m)^{(r \text{ div } p)}) \pmod n$  by (simp add: power-mult)
    also have ... =  $((a^m \pmod n)^{(r \text{ div } p)}) \pmod n$  using power-mod[of a^m n r
div p] ..
    also have ... = 1 using m(3) onen by (simp add: modeq-def power-Suc0)

```

finally have $[(a \wedge ((n - 1) \text{ div } p)) = 1] \pmod n$
using *onen* **by** (*simp add: modeq-def*)
with *pn[rule-format, OF th]* **have** *False* **by** *blast*
hence *th: $\forall m. 0 < m \wedge m < n - 1 \longrightarrow \neg [a \wedge m = 1] \pmod n$* **by** *blast*
from *lucas-weak[OF n2 an1 th]* **show** *?thesis* .
qed

definition *ord n a = (if coprime n a then Least ($\lambda d. d > 0 \wedge [a \wedge d = 1] \pmod n$) else 0)*

lemma *coprime-ord:*

assumes *na: coprime n a*
shows *ord n a > 0 $\wedge [a \wedge (\text{ord } n \ a) = 1] \pmod n \wedge (\forall m. 0 < m \wedge m < \text{ord } n \ a \longrightarrow \neg [a \wedge m = 1] \pmod n)$*
proof –
let *?P = $\lambda d. 0 < d \wedge [a \wedge d = 1] \pmod n$*
from *euclid[of a]* **obtain** *p* **where** *p: prime p a < p* **by** *blast*
from *na* **have** *o: ord n a = Least ?P* **by** (*simp add: ord-def*)
{assume *n=0 \vee n=1* **with** *na* **have** $\exists m > 0. ?P \ m$ **apply** *auto* **apply** (*rule exI[where x=1]*) **by** (*simp add: modeq-def*)}
moreover
{assume *n \neq 0 \wedge n \neq 1* **hence** *n2:n \geq 2* **by** *arith*
from *na* **have** *na': coprime a n* **by** (*simp add: coprime-commute*)
from *phi-lowerbound-1[OF n2] fermat-little[OF na']*
have *ex: $\exists m > 0. ?P \ m$* **by** – (*rule exI[where x= $\varphi \ n$], auto*) }
ultimately have *ex: $\exists m > 0. ?P \ m$* **by** *blast*
from *nat-exists-least-iff'[of ?P] ex na* **show** *?thesis*
unfolding *o[symmetric]* **by** *auto*
qed

lemma *ord-works:*

$[a \wedge (\text{ord } n \ a) = 1] \pmod n \wedge (\forall m. 0 < m \wedge m < \text{ord } n \ a \longrightarrow \neg [a \wedge m = 1] \pmod n)$
apply (*cases coprime n a*)
using *coprime-ord[of n a]*
by (*blast, simp add: ord-def modeq-def*)

lemma *ord: $[a \wedge (\text{ord } n \ a) = 1] \pmod n$ using ord-works by blast*

lemma *ord-minimal: $0 < m \implies m < \text{ord } n \ a \implies \neg [a \wedge m = 1] \pmod n$*

using *ord-works* **by** *blast*

lemma *ord-eq-0: $\text{ord } n \ a = 0 \iff \neg \text{coprime } n \ a$*

by (*cases coprime n a, simp add: neq0-conv coprime-ord, simp add: neq0-conv ord-def*)

lemma *ord-divides:*

```

[ $a \wedge d = 1$ ] (mod  $n$ )  $\longleftrightarrow$  ord  $n$   $a$  dvd  $d$  (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume rh: ?rhs
  then obtain  $k$  where  $d = \text{ord } n \ a * k$  unfolding dvd-def by blast
  hence [ $a \wedge d = (a \wedge (\text{ord } n \ a) \text{ mod } n) \wedge k$ ] (mod  $n$ )
    by (simp add: modeq-def power-mult power-mod)
  also have [ $(a \wedge (\text{ord } n \ a) \text{ mod } n) \wedge k = 1$ ] (mod  $n$ )
    using ord[of  $a$   $n$ , unfolded modeq-def]
    by (simp add: modeq-def power-mod power-Suc0)
  finally show ?lhs .
next
assume lh: ?lhs
{ assume  $H: \neg \text{coprime } n \ a$ 
  hence  $o: \text{ord } n \ a = 0$  by (simp add: ord-def)
  {assume  $d: d=0$  with  $o \ H$  have ?rhs by (simp add: modeq-def)}
  moreover
  {assume  $d0: d \neq 0$  then obtain  $d'$  where  $d': d = \text{Suc } d'$  by (cases  $d$ , auto)
    from  $H[\text{unfolded coprime}]$ 
    obtain  $p$  where  $p: p \text{ dvd } n \ p \text{ dvd } a \ p \neq 1$  by auto
    from  $lh[\text{unfolded nat-mod}]$ 
    obtain  $q1 \ q2$  where  $q12: a \wedge d + n * q1 = 1 + n * q2$  by blast
    hence  $a \wedge d + n * q1 - n * q2 = 1$  by simp
    with dvd-diff-nat [OF dvd-add [OF divides-rexp [OF  $p(2)$ , of  $d$ ] dvd-mult2 [OF
 $p(1)$ , of  $q1$ ] dvd-mult2 [OF  $p(1)$ , of  $q2$ ]]  $d'$  have  $p \text{ dvd } 1$  by simp
    with  $p(3)$  have False by simp
    hence ?rhs ..}
  ultimately have ?rhs by blast}
moreover
{assume  $H: \text{coprime } n \ a$ 
  let ?o = ord  $n \ a$ 
  let ?q =  $d \text{ div ord } n \ a$ 
  let ?r =  $d \text{ mod ord } n \ a$ 
  from cong-exp [OF ord[of  $a \ n$ ], of ?q]
  have eqo: [ $(a \wedge ?o) \wedge ?q = 1$ ] (mod  $n$ ) by (simp add: modeq-def power-Suc0)
  from  $H$  have onz: ?o  $\neq 0$  by (simp add: ord-eq-0)
  hence op: ?o  $> 0$  by simp
  from mod-div-equality [of  $d$  ord  $n \ a$ ] lh
  have [ $a \wedge (?o * ?q + ?r) = 1$ ] (mod  $n$ ) by (simp add: modeq-def mult-commute)
  hence [ $(a \wedge ?o) \wedge ?q * (a \wedge ?r) = 1$ ] (mod  $n$ )
    by (simp add: modeq-def power-mult[symmetric] power-add[symmetric])
  hence th: [ $a \wedge ?r = 1$ ] (mod  $n$ )
    using eqo mod-mult-left-eq [of  $(a \wedge ?o) \wedge ?q \ a \wedge ?r \ n$ ]
    apply (simp add: modeq-def del: One-nat-def)
    by (simp add: mod-mult-left-eq[symmetric])
  {assume  $r: ?r = 0$  hence ?rhs by (simp add: dvd-eq-mod-eq-0)}
  moreover
  {assume  $r: ?r \neq 0$ 
    with mod-less-divisor [OF op, of  $d$ ] have  $r0o: ?r > 0 \wedge ?r < ?o$  by simp
    from conjunct2 [OF ord-works [of  $a \ n$ ], rule-format, OF  $r0o$ ] th

```



```

      have ?rhs by blast}
    ultimately have ?rhs by blast}
  ultimately show ?rhs by blast
qed

```

lemma *order-divides-phi*: $\text{coprime } n \ a \implies \text{ord } n \ a \ \text{dvd } \varphi \ n$

using *ord-divides-fermat-little coprime-commute* **by** *simp*

lemma *order-divides-expdiff*:

assumes *na*: $\text{coprime } n \ a$

shows $[a^d = a^e] \ (\text{mod } n) \longleftrightarrow [d = e] \ (\text{mod } (\text{ord } n \ a))$

proof–

{**fix** *n a d e*

assume *na*: $\text{coprime } n \ a$ **and** *ed*: $(e::\text{nat}) \leq d$

hence $\exists c. d = e + c$ **by** *arith*

then obtain *c* **where** $c: d = e + c$ **by** *arith*

from *na* **have** *an*: $\text{coprime } a \ n$ **by** (*simp add: coprime-commute*)

from *coprime-exp*[*OF na*, *of e*]

have *aen*: $\text{coprime } (a^e) \ n$ **by** (*simp add: coprime-commute*)

from *coprime-exp*[*OF na*, *of c*]

have *acn*: $\text{coprime } (a^c) \ n$ **by** (*simp add: coprime-commute*)

have $[a^d = a^e] \ (\text{mod } n) \longleftrightarrow [a^{(e+c)} = a^{(e+0)}] \ (\text{mod } n)$

using *c* **by** *simp*

also have $\dots \longleftrightarrow [a^e * a^c = a^e * a^0] \ (\text{mod } n)$ **by** (*simp add: power-add*)

also have $\dots \longleftrightarrow [a^c = 1] \ (\text{mod } n)$

using *cong-mult-lcancel-eq*[*OF aen*, *of a^c a^0*] **by** *simp*

also have $\dots \longleftrightarrow \text{ord } n \ a \ \text{dvd } c$ **by** (*simp only: ord-divides*)

also have $\dots \longleftrightarrow [e + c = e + 0] \ (\text{mod } \text{ord } n \ a)$

using *cong-add-lcancel-eq*[*of e c 0 ord n a*, *simplified cong-0-divides*]

by *simp*

finally have $[a^d = a^e] \ (\text{mod } n) \longleftrightarrow [d = e] \ (\text{mod } (\text{ord } n \ a))$

using *c* **by** *simp* }

note *th* = *this*

have $e \leq d \vee d \leq e$ **by** *arith*

moreover

{**assume** *ed*: $e \leq d$ **from** *th*[*OF na ed*] **have** ?thesis .}

moreover

{**assume** *de*: $d \leq e$

from *th*[*OF na de*] **have** ?thesis **by** (*simp add: cong-commute*) }

ultimately show ?thesis **by** *blast*

qed

lemma *prime-prime-factor*:

$\text{prime } n \longleftrightarrow n \neq 1 \wedge (\forall p. \text{prime } p \wedge p \ \text{dvd } n \longrightarrow p = n)$

proof–

{**assume** *n*: $n=0 \vee n=1$ **hence** ?thesis **using** *prime-0 two-is-prime* **by** *auto*}

moreover

{**assume** *n*: $n \neq 0 \ n \neq 1$

```

{assume pn: prime n

  from pn[unfolded prime-def] have  $\forall p. \text{prime } p \wedge p \text{ dvd } n \longrightarrow p = n$ 
    using n
    apply (cases n = 0  $\vee$  n=1, simp)
    by (clarsimp, erule-tac x=p in allE, auto)}
moreover
{assume H:  $\forall p. \text{prime } p \wedge p \text{ dvd } n \longrightarrow p = n$ 
  from n have n1:  $n > 1$  by arith
  {fix m assume m:  $m \text{ dvd } n \wedge m \neq 1$ 
    from prime-factor[OF m(2)] obtain p where
      p: prime p  $\wedge$  p dvd m by blast
    from dvd-trans[OF p(2) m(1)] p(1) H have p = n by blast
    with p(2) have n dvd m by simp
    hence m=n using dvd-antisym[OF m(1)] by simp }
  with n1 have prime n unfolding prime-def by auto }
ultimately have ?thesis using n by blast}
ultimately show ?thesis by auto
qed

lemma prime-divisor-sqrt:
  prime n  $\longleftrightarrow$   $n \neq 1 \wedge (\forall d. d \text{ dvd } n \wedge d^2 \leq n \longrightarrow d = 1)$ 
proof-
{assume n=0  $\vee$  n=1 hence ?thesis using prime-0 prime-1
  by (auto simp add: nat-power-eq-0-iff)}
moreover
{assume n:  $n \neq 0 \wedge n \neq 1$ 
  hence np:  $n > 1$  by arith
  {fix d assume d:  $d \text{ dvd } n \wedge d^2 \leq n$  and H:  $\forall m. m \text{ dvd } n \longrightarrow m=1 \vee m=n$ 
    from H d have d1n:  $d = 1 \vee d=n$  by blast
    {assume dn: d=n
      have n^2 > n*1 using n
        by (simp add: power2-eq-square mult-less-cancel1)
      with dn d(2) have d=1 by simp}
    with d1n have d = 1 by blast }
moreover
{fix d assume d:  $d \text{ dvd } n$  and H:  $\forall d'. d' \text{ dvd } n \wedge d'^2 \leq n \longrightarrow d' = 1$ 
  from d n have d  $\neq$  0 apply - apply (rule ccontr) by simp
  hence 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
  have d^2  $\leq$  n  $\vee$  e^2  $\leq$  n using dp ep
    by (auto simp add: e power2-eq-square mult-le-cancel-left)
moreover
{assume h:  $d^2 \leq n$ 
  from H[rule-format, of d] h d have d = 1 by blast}
moreover
{assume h:  $e^2 \leq n$ 
  from e have e dvd n unfolding dvd-def by (simp add: mult-commute)

```

with $H[\text{rule-format}, \text{of } e] \ h$ have $e=1$ by *simp*
 with e have $d = n$ by *simp*
 ultimately have $d=1 \vee d=n$ by *blast*
 ultimately have *?thesis* unfolding *prime-def* using *np n(2)* by *blast*
 ultimately show *?thesis* by *auto*
 qed
 lemma *prime-prime-factor-sqrt*:
 $\text{prime } n \longleftrightarrow n \neq 0 \wedge n \neq 1 \wedge \neg (\exists p. \text{prime } p \wedge p \text{ dvd } n \wedge p^2 \leq n)$
 (is *?lhs* \longleftrightarrow *?rhs*)
 proof –
 {assume $n=0 \vee n=1$ hence *?thesis* using *prime-0 prime-1* by *auto*}
 moreover
 {assume $n: n \neq 0 \ n \neq 1$
 {assume $H: ?lhs$
 from $H[\text{unfolded prime-divisor-sqrt}] \ n$
 have *?rhs* apply *clarsimp* by (erule-tac $x=p$ in *allE*, *simp* add: *prime-1*)
 }
 moreover
 {assume $H: ?rhs$
 {fix d assume $d: d \text{ dvd } n \ d^2 \leq n \ d \neq 1$
 from *prime-factor* [OF $d(3)$]
 obtain p where $p: \text{prime } p \ p \text{ dvd } d$ by *blast*
 from n have $np: n > 0$ by *arith*
 from $d(1) \ n$ have $d \neq 0$ by – (rule *ccontr*, *auto*)
 hence $dp: d > 0$ by *arith*
 from *mult-mono* [OF $dvd\text{-}imp\text{-}le$ [OF $p(2) \ dp$] $dvd\text{-}imp\text{-}le$ [OF $p(2) \ dp$]] $d(2)$
 have $p^2 \leq n$ unfolding *power2-eq-square* by *arith*
 with $H \ n \ p(1) \ dvd\text{-}trans$ [OF $p(2) \ d(1)$] have *False* by *blast*}
 with $n \ prime\text{-}divisor\text{-}sqrt$ have *?lhs* by *auto*}
 ultimately have *?thesis* by *blast* }
 ultimately show *?thesis* by (cases $n=0 \vee n=1$, *auto*)
 qed

lemma *pocklington-lemma*:
 assumes $n: n \geq 2$ and $nqr: n - 1 = q*r$ and $an: [a^{(n-1)} = 1] \ (\text{mod } n)$
 and $aq: \forall p. \text{prime } p \wedge p \text{ dvd } q \longrightarrow \text{coprime } (a^{((n-1) \text{ div } p)} - 1) \ n$
 and $pp: \text{prime } p$ and $pn: p \text{ dvd } n$
 shows $[p = 1] \ (\text{mod } q)$
 proof –
 from $pp \ prime\text{-}0 \ prime\text{-}1$ have $p01: p \neq 0 \ p \neq 1$ by – (rule *ccontr*, *simp*) +
 from *cong-1-divides* [OF an , *unfolded nqr*, *unfolded dvd-def*]
 obtain k where $k: a^{(q * r)} - 1 = n*k$ by *blast*
 from pn [OF *unfolded dvd-def*] obtain l where $l: n = p*l$ by *blast*
 {assume $a0: a = 0$
 hence $a^{(n-1)} = 0$ using n by (*simp* add: *power-0-left*)
 with $n \ an \ mod\text{-}less$ [of $1 \ n$] have *False* by (*simp* add: *power-0-left modeq-def*)}
 hence $a0: a \neq 0$..
 from $n \ nqr$ have $aqr0: a^{(q * r)} \neq 0$ using $a0$ by (*simp* add: *neg0-conv*)

hence $(a \wedge (q * r) - 1) + 1 = a \wedge (q * r)$ **by** *simp*
 with $k \ l$ **have** $a \wedge (q * r) = p * l * k + 1$ **by** *simp*
 hence $a \wedge (r * q) + p * 0 = 1 + p * (l * k)$ **by** (*simp add: mult-ac*)
 hence $odq: \text{ord } p \ (a \wedge r) \ \text{dvd } q$
 unfolding *ord-divides[symmetric] power-mult[symmetric] nat-mod* **by** *blast*
 from $odq[\text{unfolded dvd-def}]$ **obtain** d **where** $d: q = \text{ord } p \ (a \wedge r) * d$ **by** *blast*
 {**assume** $d1: d \neq 1$
 from *prime-factor[OF d1]* **obtain** P **where** $P: \text{prime } P \ P \ \text{dvd } d$ **by** *blast*
 from $d \ \text{dvd-mult}[OF \ P(2), \text{ of } \text{ord } p \ (a \wedge r)]$ **have** $Pq: P \ \text{dvd } q$ **by** *simp*
 from $aq \ P(1) \ Pq$ **have** $caP: \text{coprime } (a \wedge ((n - 1) \ \text{div } P) - 1) \ n$ **by** *blast*
 from Pq **obtain** s **where** $s: q = P * s$ **unfolding** *dvd-def* **by** *blast*
 have $P0: P \neq 0$ **using** $P(1)$ *prime-0* **by** $-(\text{rule ccontr, simp})$
 from $P(2)$ **obtain** t **where** $t: d = P * t$ **unfolding** *dvd-def* **by** *blast*
 from $d \ s \ t \ P0$ **have** $s': \text{ord } p \ (a \wedge r) * t = s$ **by** *algebra*
 have $\text{ord } p \ (a \wedge r) * t * r = r * \text{ord } p \ (a \wedge r) * t$ **by** *algebra*
 hence $\text{exps}: a \wedge (\text{ord } p \ (a \wedge r) * t * r) = ((a \wedge r) \wedge \text{ord } p \ (a \wedge r)) \wedge t$
by (*simp only: power-mult*)
 have $[((a \wedge r) \wedge \text{ord } p \ (a \wedge r)) \wedge t = 1 \wedge t] \ (\text{mod } p)$
by (*rule cong-exp, rule ord*)
 then **have** $th: [((a \wedge r) \wedge \text{ord } p \ (a \wedge r)) \wedge t = 1] \ (\text{mod } p)$
by (*simp add: power-Suc0*)
 from *cong-1-divides[OF th] exps* **have** $pd0: p \ \text{dvd } a \wedge (\text{ord } p \ (a \wedge r) * t * r) - 1$
by *simp*
 from $nqr \ s \ s'$ **have** $(n - 1) \ \text{div } P = \text{ord } p \ (a \wedge r) * t * r$ **using** $P0$ **by** *simp*
 with caP **have** $\text{coprime } (a \wedge (\text{ord } p \ (a \wedge r) * t * r) - 1) \ n$ **by** *simp*
 with $p01 \ pn \ pd0$ **have** *False* **unfolding** *coprime* **by** *auto*}
 hence $d1: d = 1$ **by** *blast*
 hence $o: \text{ord } p \ (a \wedge r) = q$ **using** d **by** *simp*
 from $pp \ \text{phi-prime}[of \ p]$ **have** $\text{phip}: \varphi \ p = p - 1$ **by** *simp*
 {**fix** d **assume** $d: d \ \text{dvd } p \ d \ \text{dvd } a \ d \neq 1$
 from $pp[\text{unfolded prime-def}] \ d$ **have** $dp: d = p$ **by** *blast*
 from n **have** $n12: \text{Suc } (n - 2) = n - 1$ **by** *arith*
 with *divides-rexp[OF d(2)[unfolded dp], of n - 2]*
 have $th0: p \ \text{dvd } a \wedge (n - 1)$ **by** *simp*
 from n **have** $n0: n \neq 0$ **by** *simp*
 from $d(2) \ an \ n12[\text{symmetric}]$ **have** $a0: a \neq 0$
by $-(\text{rule ccontr, simp add: modeq-def})$
 have $th1: a \wedge (n - 1) \neq 0$ **using** $n \ d(2) \ dp \ a0$ **by** (*auto simp add: neq0-conv*)
 from *coprime-minus1[OF th1, unfolded coprime]*
dvd-trans[OF pn cong-1-divides[OF an]] th0 d(3) dp
have *False* **by** *auto*}
 hence $cpa: \text{coprime } p \ a$ **using** *coprime* **by** *auto*
 from *coprime-exp[OF cpa, of r] coprime-commute*
 have $\text{arp}: \text{coprime } (a \wedge r) \ p$ **by** *blast*
 from *fermat-little[OF arp, simplified ord-divides]* $o \ \text{phip}$
 have $q \ \text{dvd } (p - 1)$ **by** *simp*
 then **obtain** d **where** $d: p - 1 = q * d$ **unfolding** *dvd-def* **by** *blast*
 from *prime-0 pp* **have** $p0: p \neq 0$ **by** $-(\text{rule ccontr, auto})$
 from $p0 \ d$ **have** $p + q * 0 = 1 + q * d$ **by** *simp*

with *nat-mod*[of $p \mid q$, *symmetric*]
 show *?thesis* by *blast*
 qed

lemma *pocklington*:

assumes $n: n \geq 2$ and $nqr: n - 1 = q * r$ and $sqr: n \leq q^2$
 and $an: [a^{(n-1)} = 1] \pmod n$
 and $aq: \forall p. \text{prime } p \wedge p \mid q \longrightarrow \text{coprime } (a^{((n-1) \div p)} - 1) \ n$
 shows *prime* n

unfolding *prime-prime-factor-sqrt*[of n]

proof –

let *?ths* = $n \neq 0 \wedge n \neq 1 \wedge \neg (\exists p. \text{prime } p \wedge p \mid n \wedge p^2 \leq n)$
 from n have $n01: n \neq 0 \wedge n \neq 1$ by *arith+*
 {fix p assume $p: \text{prime } p \wedge p \mid n \wedge p^2 \leq n$
 from $p(3)$ *sqr* have $p^{(Suc\ 1)} \leq q^{(Suc\ 1)}$ by (*simp add: power2-eq-square*)
 hence $pq: p \leq q$ unfolding *exp-mono-le* .
 from *pocklington-lemma*[OF n *nqr* *an* *aq* $p(1,2)$] *cong-1-divides*
 have *th*: $q \mid p - 1$ by *blast*
 have $p - 1 \neq 0$ using *prime-ge-2*[OF $p(1)$] by *arith*
 with *divides-ge*[OF *th*] *pq* have *False* by *arith* }
 with $n01$ show *?ths* by *blast*

qed

lemma *pocklington-alt*:

assumes $n: n \geq 2$ and $nqr: n - 1 = q * r$ and $sqr: n \leq q^2$
 and $an: [a^{(n-1)} = 1] \pmod n$
 and $aq: \forall p. \text{prime } p \wedge p \mid q \longrightarrow (\exists b. [a^{((n-1) \div p)} = b] \pmod n \wedge \text{coprime } (b - 1) \ n)$
 shows *prime* n

proof –

{fix p assume $p: \text{prime } p \wedge p \mid q$
 from *aq*[*rule-format*] p obtain b where
 $b: [a^{((n-1) \div p)} = b] \pmod n \wedge \text{coprime } (b - 1) \ n$ by *blast*
 {assume $a0: a = 0$
 from n *an* have $[0 = 1] \pmod n$ unfolding *a0 power-0-left* by *auto*
 hence *False* using n by (*simp add: modeq-def dvd-eq-mod-eq-0*[*symmetric*])}
 hence $a0: a \neq 0$..
 hence $a1: a \geq 1$ by *arith*
 from *one-le-power*[OF $a1$] have *ath*: $1 \leq a^{((n-1) \div p)}$.
 {assume $b0: b = 0$
 from $p(2)$ *nqr* have $(n - 1) \bmod p = 0$
 apply (*simp only: dvd-eq-mod-eq-0*[*symmetric*]) by (*rule dvd-mult2, simp*)
 with *mod-div-equality*[of $n - 1$ p]
 have $(n - 1) \div p * p = n - 1$ by *auto*
 hence *eq*: $(a^{((n-1) \div p)})^p = a^{(n-1)}$
 by (*simp only: power-mult*[*symmetric*])
 from *prime-ge-2*[OF $p(1)$] have $pS: \text{Suc } (p - 1) = p$ by *arith*
 from $b(1)$ have *d*: $n \mid a^{((n-1) \div p)}$ unfolding *b0 cong-0-divides* .

```

    from divides-rexp[OF d, of p - 1] pS eq cong-divides[OF an] n
    have False by simp}
  then have b0: b ≠ 0 ..
  hence b1: b ≥ 1 by arith
  from cong-coprime[OF cong-sub[OF b(1) cong-refl[of 1] ath b1]] b(2) nqr
  have coprime (a ^ ((n - 1) div p) - 1) n by (simp add: coprime-commute)}
  hence th: ∀ p. prime p ∧ p dvd q ⟶ coprime (a ^ ((n - 1) div p) - 1) n
    by blast
  from pocklington[OF n nqr sqr an th] show ?thesis .
qed

```

definition *primefact* ps n = (foldr op * ps 1 = n ∧ (∀ p ∈ set ps. prime p))

```

lemma primefact: assumes n: n ≠ 0
  shows ∃ ps. primefact ps n
using n
proof(induct n rule: nat-less-induct)
  fix n assume H: ∀ m < n. m ≠ 0 ⟶ (∃ ps. primefact ps m) and n: n ≠ 0
  let ?ths = ∃ ps. primefact ps n
  {assume n = 1
    hence primefact [] n by (simp add: primefact-def)
    hence ?ths by blast }
  moreover
  {assume n1: n ≠ 1
    with n have n2: n ≥ 2 by arith
    from prime-factor[OF n1] obtain p where p: prime p p dvd n by blast
    from p(2) obtain m where m: n = p*m unfolding dvd-def by blast
    from n m have m0: m > 0 m ≠ 0 by auto
    from prime-ge-2[OF p(1)] have 1 < p by arith
    with m0 m have mn: m < n by auto
    from H[rule-format, OF mn m0(2)] obtain ps where ps: primefact ps m ..
    from ps m p(1) have primefact (p#ps) n by (simp add: primefact-def)
    hence ?ths by blast}
  ultimately show ?ths by blast
qed

```

```

lemma primefact-contains:
  assumes pf: primefact ps n and p: prime p and pn: p dvd n
  shows p ∈ set ps
  using pf p pn
proof(induct ps arbitrary: p n)
  case Nil thus ?case by (auto simp add: primefact-def)
next
  case (Cons q qs p n)
  from Cons.premis[unfolded primefact-def]
  have q: prime q q * foldr op * qs 1 = n ∀ p ∈ set qs. prime p and p: prime p p
    dvd q * foldr op * qs 1 by simp-all

```

```

{assume p dvd q
 with p(1) q(1) have p = q unfolding prime-def by auto
 hence ?case by simp}
moreover
{ assume h: p dvd foldr op * qs 1
 from q(3) have pqs: primefact qs (foldr op * qs 1)
 by (simp add: primefact-def)
 from Cons.hyps[OF pqs p(1) h] have ?case by simp}
ultimately show ?case using prime-divprod[OF p] by blast
qed

```

lemma primefact-variant: $\text{primefact } ps \ n \longleftrightarrow \text{foldr } op \ * \ ps \ 1 = n \wedge \text{list-all prime } ps$ by (auto simp add: primefact-def list-all-iff)

lemma lucas-primefact:
assumes $n: n \geq 2$ **and** $an: [a^{(n-1)} = 1] \pmod n$
and $psn: \text{foldr } op \ * \ ps \ 1 = n - 1$
and $psp: \text{list-all } (\lambda p. \text{prime } p \wedge \neg [a^{((n-1) \text{ div } p)} = 1] \pmod n) \ ps$
shows $\text{prime } n$
proof–
{fix p **assume** $p: \text{prime } p \ p \text{ dvd } n - 1 \ [a^{((n-1) \text{ div } p)} = 1] \pmod n$
from $psn \ psp$ **have** $psn1: \text{primefact } ps \ (n - 1)$
by (auto simp add: list-all-iff primefact-variant)
from $p(3) \text{ primefact-contains}[OF \ psn1 \ p(1,2)] \ psp$
have False **by** (induct ps, auto)}
with $\text{lucas}[OF \ n \ an]$ **show** ?thesis **by** blast
qed

lemma mod-le: **assumes** $n: n \neq (0::nat)$ **shows** $m \text{ mod } n \leq m$
proof–
from $\text{mod-div-equality}[of \ m \ n]$
have $\exists x. x + m \text{ mod } n = m$ **by** blast
then show ?thesis **by** auto
qed

lemma pocklington-primefact:
assumes $n: n \geq 2$ **and** $qrn: q*r = n - 1$ **and** $nq2: n \leq q^2$
and $arnb: (a^r) \text{ mod } n = b$ **and** $psq: \text{foldr } op \ * \ ps \ 1 = q$
and $bqn: (b^q) \text{ mod } n = 1$
and $psp: \text{list-all } (\lambda p. \text{prime } p \wedge \text{coprime } ((b^{(q \text{ div } p)}) \text{ mod } n - 1) \ n) \ ps$
shows $\text{prime } n$
proof–
from $bqn \ psp \ qrn$
have $bqn: a^{(n-1)} \text{ mod } n = 1$

```

    and psp: list-all ( $\lambda p. \text{prime } p \wedge \text{coprime } (a^{(r * (q \text{ div } p))} \bmod n - 1) \ n) \ ps$ 
unfolding arnb[symmetric] power-mod
    by (simp-all add: power-mult[symmetric] algebra-simps)
from n have n0:  $n > 0$  by arith
from mod-div-equality[of  $a^{(n - 1)}$  n]
    mod-less-divisor[OF n0, of  $a^{(n - 1)}$ ]
have an1:  $[a^{(n - 1)} = 1] \ (\bmod \ n)$ 
unfolding nat-mod bqn
apply -
apply (rule exI[where  $x=0$ ])
apply (rule exI[where  $x=a^{(n - 1) \text{ div } n}$ ])
by (simp add: algebra-simps)
{fix p assume p: prime p p dvd q
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 ( $a^{(r * (q \text{ div } p))} \bmod n - 1) \ n$ 
    by (simp add: list-all-iff)
from prime-ge-2[OF p(1)] have p01:  $p \neq 0 \ p \neq 1 \ p = \text{Suc}(p - 1)$  by arith+
from div-mult1-eq[of r q p] p(2)
have eq1:  $r * (q \text{ div } p) = (n - 1) \text{ div } p$ 
    unfolding grn[symmetric] dvd-eq-mod-eq-0 by (simp add: mult-commute)
have ath:  $\bigwedge a \ (b::\text{nat}). \ a \leq b \implies a \neq 0 \implies 1 \leq a \wedge 1 \leq b$  by arith
from n0 have n00:  $n \neq 0$  by arith
from mod-le[OF n00]
have th10:  $a^{((n - 1) \text{ div } p) \bmod n} \leq a^{((n - 1) \text{ div } p)}$  .
{assume a  $a^{((n - 1) \text{ div } p) \bmod n} = 0$ 
    then obtain s where s:  $a^{((n - 1) \text{ div } p)} = n * s$ 
    unfolding mod-eq-0-iff by blast
    hence eq0:  $(a^{((n - 1) \text{ div } p)})^p = (n * s)^p$  by simp
    from grn[symmetric] have qn1:  $q \text{ dvd } n - 1$  unfolding dvd-def by auto
    from dvd-trans[OF p(2) qn1] div-mod-equality'[of n - 1 p]
    have npp:  $(n - 1) \text{ div } p * p = n - 1$  by (simp add: dvd-eq-mod-eq-0)
    with eq0 have  $a^{(n - 1)} = (n * s)^p$ 
        by (simp add: power-mult[symmetric])
    hence  $1 = (n * s)^{(\text{Suc } (p - 1)) \bmod n}$  using bqn p01 by simp
    also have  $\dots = 0$  by (simp add: mult-assoc)
    finally have False by simp }
then have th11:  $a^{((n - 1) \text{ div } p) \bmod n} \neq 0$  by auto
have th1:  $[a^{((n - 1) \text{ div } p) \bmod n} = a^{((n - 1) \text{ div } p)}] \ (\bmod \ n)$ 
    unfolding modeq-def by simp
from cong-sub[OF th1 cong-refl[of 1]] ath[OF th10 th11]
have th:  $[a^{((n - 1) \text{ div } p) \bmod n - 1} = a^{((n - 1) \text{ div } p)} - 1] \ (\bmod \ n)$ 
    by blast
from cong-coprime[OF th] p'[unfolded eq1]
have coprime ( $a^{((n - 1) \text{ div } p)} - 1) \ n$  by (simp add: coprime-commute) }
with pocklington[OF n grn[symmetric] nq2 an1]
show ?thesis by blast
qed

```


end