Some results of number theory

Jeremy Avigad
David Gray
Adam Kramer
Thomas M Rasmussen

November 22, 2007

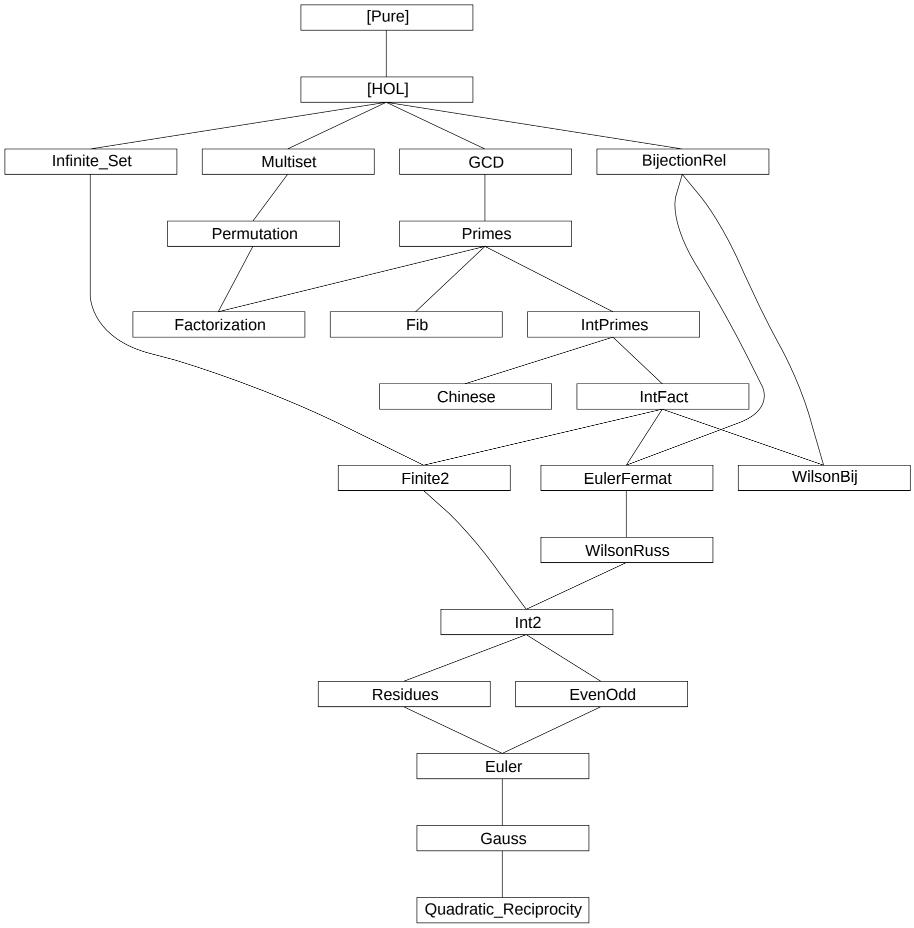
Abstract

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


[Pure]
[HOL]
Infinite_Set
Multiset
GCD
BijectionRel
Permutation
Primes
Factorization
Fib
IntPrimes
Chinese
IntFact
Finite2
EulerFermat
WilsonBij
WilsonRuss
Int2
Residues
EvenOdd
Euler
Gauss
Quadratic_Reciprocity

1 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* ≤ *n* ==> *gcd* (*fib m*, *fib* (*n* − *m*)) = *gcd* (*fib m*, *fib n*)
by (*simp add*: *gcd-fib-add* [*symmetric*, *of* - *n*−*m*])

lemma *gcd-fib-mod*: *0* < *m* ==> *gcd* (*fib m*, *fib* (*n mod m*)) = *gcd* (*fib m*, *fib n*)

proof (*induct n rule*: *less-induct*)
  case (*less n*)
  from *less.prems* **have** *pos-m*: $0 < m$ **.**
  show *gcd* (*fib m*, *fib* (*n mod m*)) = *gcd* (*fib m*, *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 *gcd* (*fib m*, *fib* (*n mod m*)) = *gcd* (*fib m*, *fib* ((*n* − *m*) *mod m*))
    by (*simp add*: *mod-if* [*of n*]) (*insert m-n*, *auto*)
    also have ... = *gcd* (*fib m*, *fib* (*n* − *m*)) **by** (*simp add*: *less.hyps diff pos-m*)
    also have ... = *gcd* (*fib m*, *fib n*) **by** (*simp add*: *gcd-fib-diff m-n'*)
    finally show *gcd* (*fib m*, *fib* (*n mod m*)) = *gcd* (*fib m*, *fib n*) **.**
  next
    case *False* **then show** *gcd* (*fib m*, *fib* (*n mod m*)) = *gcd* (*fib m*, *fib n*)
    by (*cases* $m = n$) *auto*
  qed
qed

lemma *fib-gcd*: *fib* (*gcd* (*m*, *n*)) = *gcd* (*fib m*, *fib n*) — Law 6.111
  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*:
    fib (*Suc n*) ∗ *fib n* = ($\sum$ *k* ∈ {..*n*}. *fib k* ∗ *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

2 Fundamental Theorem of Arithmetic (unique factorization into primes)

theory *Factorization* **imports** *Primes Permutation* **begin**

2.1 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 => $x \leq y \wedge nondec\ xs$)

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

primrec
$oinsert\ x$ [] = [x]
$oinsert\ x$ (y # ys) = (if $x \leq y$ $then$ x # y # ys $else$ y # $oinsert\ x\ ys$)

primrec
$sort$ [] = []
$sort$ (x # xs) = $oinsert\ x$ ($sort\ xs$)

2.2 Arithmetic

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

lemma *one-less-k*: (m::nat) $\neq m * k$ ==> $Suc\ 0 < m * k$ ==> $Suc\ 0 < k$
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

2.3 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* ∧ *primel ys*)
apply (*unfold primel-def*)
apply *auto*
done

lemma *prime-primel*: *prime n* ==> *primel* [*n*] ∧ *prod* [*n*] = *n*
apply (*unfold primel-def*)
apply *auto*
done

lemma *prime-nd-one*: *prime p* ==> ¬ *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* ∧ *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
```

2.4 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 ⟶ 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
```

2.5 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 <~~> ys ==> sort xs = sort ys
  apply (induct set: perm)
     apply (simp-all add: oinsert-x-y)
  done
```

2.6 Existence

```
lemma ex-nondec-lemma:
    primel xs ==> ∃ ys. primel ys ∧ nondec ys ∧ prod ys = prod xs
  apply (blast intro: nondec-sort perm-prod perm-primel perm-sort perm-sym)
  done
```

lemma *not-prime-ex-mk*:
  Suc 0 $<$ *n* $\wedge$ $\neg$ *prime n* ==>
    $\exists$ *m k*. *Suc 0* $<$ *m* $\wedge$ *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*:
    primel xs ==> *primel ys* ==> $\exists$ *l*. *primel l* $\wedge$ *prod l* $=$ *prod xs* $*$ *prod ys*
  by (*metis primel-append prod.simps*(*2*) *prod-append*)

lemma *factor-exists* [*rule-format*]: *Suc 0* $<$ *n* −−> ($\exists$ *l*. *primel l* $\wedge$ *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*: *Suc 0* $<$ *n* ==> $\exists$ *l*. *primel l* $\wedge$ *nondec l* $\wedge$ *prod l* $=$
n
  apply (*erule factor-exists* [*THEN exE*])
  apply (*blast intro*!: *ex-nondec-lemma*)
  done

2.7 Uniqueness

lemma *prime-dvd-mult-list* [*rule-format*]:
    prime p ==> *p dvd* (*prod xs*) −−> ($\exists$ *m*. *m*:*set xs* $\wedge$ *p 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* $\in$ *set ys* $\wedge$ *x 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* $\in$ *set ys* ==> *x*
dvd m ==> *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 ∈ 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 ==> ∃ l. ys <~~> (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 ==> prod xs < prod ys
  by (metis Nat.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 ∗ prod xs = p ==> prime p ==> xs = []
  apply (auto intro: primel-one-empty simp add: prime-def)
  done

lemma uniq-ex-aux:
  ∀ m. m < prod ys --> (∀ xs ys. primel xs ∧ primel ys ∧
      prod xs = prod ys ∧ prod xs = m --> xs <~~> ys) ==>
    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 (simp add: perm-sing-eq primel-hd-tl)
   apply (rule-tac p = a in prod-one-empty)
     apply simp-all
  apply (erule uniq-ex-aux)
     apply (auto intro: primel-tl perm-primel simp add: primel-hd-tl)
   apply (rule-tac k = a and n = prod list and m = prod x in mult-left-cancel)
    apply (rule-tac [3] x = a in primel-prod-less)
      apply (rule-tac [2] prod-xy-prod)
      apply (rule-tac [2] s = prod ys in HOL.trans)
       apply (erule-tac [3] perm-prod)
      apply (erule-tac [5] perm-prod [symmetric])
     apply (auto intro: perm-primel prime-g-zero)
  done

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


lemma unique-prime-factorization [rule-format]:
    ∀n. Suc 0 < n --> (∃!l. primel l ∧ nondec l ∧ prod l = n)
  apply safe
   apply (erule nondec-factor-exists)
  apply (rule perm-nondec-unique)
    apply (rule factor-unique)
    apply simp-all
  done

end
```

3 Divisibility and prime numbers (on integers)

theory *IntPrimes* **imports** *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*.

3.1 Definitions

```
consts
  xzgcda :: int * int * int * int * int * int * int * int => int * int * int
```

recdef *xzgcda*
  measure $((\lambda(m, n, r', r, s', s, t', t).\ nat\ r)$
    $::\ int * int * int * int *int * int * int * int => nat)$
  $xzgcda\ (m, n, r', r, s', s, t', t) =$
    $(if\ r \leq 0\ then\ (r', s', t')$
     $else\ xzgcda\ (m, n, r, r'\ mod\ r,$
       $s, s' - (r'\ div\ r) * s,$
       $t, t' - (r'\ div\ r) * t))$

definition
  $zgcd :: int * int => int$ **where**
  $zgcd = (\lambda(x,y).\ int\ (gcd\ (nat\ (abs\ x), nat\ (abs\ y))))$

definition
  $zprime :: int \Rightarrow bool$ **where**
  $zprime\ p = (1 < p \wedge (\forall m.\ 0 <= m\ \&\ m\ dvd\ p\ --> m = 1 \vee m = p))$

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**
  $[a = b]\ (mod\ m) = (m\ dvd\ (a - b))$

gcd lemmas

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

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

3.2 Euclid's Algorithm and GCD

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

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

lemma *zgcd-zminus* [*simp*]: $zgcd\ (-m, n) = zgcd\ (m, n)$
  by (*simp add*: *zgcd-def*)

lemma *zgcd-zminus2* [*simp*]: $zgcd\ (m, -n) = zgcd\ (m, n)$
  by (*simp add*: *zgcd-def*)

lemma *zgcd-non-0*: $0 < n ==> zgcd\ (m, n) = zgcd\ (n, m\ mod\ n)$

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*]: *zgcd* (m, *1*) = *1*
by (*simp add*: *zgcd-def abs-if*)

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

lemma *zgcd-zdvd1* [*iff*]: *zgcd* (m, n) *dvd* m
by (*simp add*: *zgcd-def abs-if int-dvd-iff*)

lemma *zgcd-zdvd2* [*iff*]: *zgcd* (m, n) *dvd* n
by (*simp add*: *zgcd-def abs-if int-dvd-iff*)

lemma *zgcd-greatest-iff*: 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-commute*: *zgcd* (m, n) = *zgcd* (n, m)
by (*simp add*: *zgcd-def gcd-commute*)

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

lemma *zrelprime-zdvd-zmult-aux*:
zgcd (*n*, *k*) = *1* ==> *k dvd m* $*$ *n* ==> *0* $\leq$ *m* ==> *k dvd m*
by (*metis abs-of-nonneg zdvd-triv-right zgcd-greatest-iff zgcd-zmult-distrib2-abs zmult-1-right*)

lemma *zrelprime-zdvd-zmult*: *zgcd* (*n*, *k*) = *1* ==> *k dvd m* $*$ *n* ==> *k dvd m*
apply (*case-tac 0* $\leq$ *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* <= *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 ==> $\neg$ *p dvd n* ==> *zgcd* (*n*, *p*) = *1*
apply (*auto simp add*: *zprime-def*)
apply (*metis zgcd-commute zgcd-geq-zero zgcd-zdvd1 zgcd-zdvd2*)
done

lemma *zless-zprime-imp-zrelprime*:
zprime p ==> *0* < *n* ==> *n* < *p* ==> *zgcd* (*n*, *p*) = *1*
apply (*erule zprime-imp-zrelprime*)
apply (*erule zdvd-not-zless*, *assumption*)
done

lemma *zprime-zdvd-zmult*:
  $0 \leq (m::int) ==> zprime\ p ==> p\ dvd\ m * n ==> p\ dvd\ m \vee p\ dvd\ n$
 by (*metis igcd-dvd1 igcd-dvd2 igcd-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*: *zmod-zadd1-eq*)
 apply (*rule zgcd-eq* [*symmetric*])
 done

lemma *zgcd-zdvd-zgcd-zmult*: $zgcd\ (m,\ n)\ dvd\ zgcd\ (k * m,\ n)$
 apply (*simp add*: *zgcd-greatest-iff*)
 apply (*blast intro*: *zdvd-trans*)
 done

lemma *zgcd-zmult-zdvd-zgcd*:
  $zgcd\ (k,\ n) = 1 ==> zgcd\ (k * m,\ n)\ dvd\ zgcd\ (m,\ n)$
 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*: $zgcd\ (k,\ n) = 1 ==> zgcd\ (k * m,\ n) = zgcd\ (m,\ n)$
 by (*simp add*: *zgcd-def nat-abs-mult-distrib gcd-mult-cancel*)

lemma *zgcd-zgcd-zmult*:
  $zgcd\ (k,\ m) = 1 ==> zgcd\ (n,\ m) = 1 ==> zgcd\ (k * n,\ m) = 1$
 by (*simp add*: *zgcd-zmult-cancel*)

lemma *zdvd-iff-zgcd*: $0 < m ==> (m\ dvd\ n) = (zgcd\ (n,\ m) = m)$
 by (*metis abs-of-pos zdvd-mult-div-cancel zgcd-0 zgcd-commute zgcd-geq-zero zgcd-zdvd2 zgcd-zmult-eq-self*)

3.3 Congruences

lemma *zcong-1* [*simp*]: $[a = b]\ (mod\ 1)$
 by (*unfold zcong-def*, *auto*)

lemma *zcong-refl* [*simp*]: $[k = k]\ (mod\ m)$
 by (*unfold zcong-def*, *auto*)

lemma *zcong-sym*: $[a = b]\ (mod\ m) = [b = a]\ (mod\ m)$
 apply (*unfold zcong-def dvd-def*, *auto*)
  apply (*rule-tac* [!] $x = -k$ **in** *exI*, *auto*)
 done

lemma *zcong-zadd*:

```
    [a = b] (mod m) ==> [c = d] (mod m) ==> [a + c = b + d] (mod m)
  apply (unfold zcong-def)
  apply (rule-tac s = (a - b) + (c - d) in subst)
   apply (rule-tac [2] zdvd-zadd, auto)
  done

lemma zcong-zdiff:
    [a = b] (mod m) ==> [c = d] (mod m) ==> [a - c = b - d] (mod m)
  apply (unfold zcong-def)
  apply (rule-tac s = (a - b) - (c - d) in subst)
   apply (rule-tac [2] zdvd-zdiff, auto)
  done

lemma zcong-trans:
    [a = b] (mod m) ==> [b = c] (mod m) ==> [a = c] (mod m)
  apply (unfold zcong-def dvd-def, auto)
  apply (rule-tac x = k + ka in exI)
  apply (simp add: zadd-ac zadd-zmult-distrib2)
  done

lemma zcong-zmult:
    [a = b] (mod m) ==> [c = d] (mod m) ==> [a * c = b * d] (mod m)
  apply (rule-tac b = b * c in zcong-trans)
   apply (unfold zcong-def)
  apply (metis zdiff-zmult-distrib2 zdvd-zmult zmult-commute)
  apply (metis zdiff-zmult-distrib2 zdvd-zmult)
  done

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

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

lemma zcong-zmult-self: [a * m = b * m] (mod m)
  apply (unfold zcong-def)
  apply (rule zdvd-zdiff, simp-all)
  done

lemma zcong-square:
   [| zprime p;  0 < a;  [a * a = 1] (mod p)|]
    ==> [a = 1] (mod p) ∨ [a = p - 1] (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*:
  $0 \leq m ==>$
    $zgcd\ (k,\ m) = 1 ==> [a * k = b * k]\ (mod\ m) = [a = b]\ (mod\ m)$
  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*: *zdiff-zmult-distrib*)
  apply (*subgoal-tac* $m\ dvd\ (-(a * k - b * k))$)
   apply *simp*
  apply (*subst zdvd-zminus-iff*, *assumption*)
  done

lemma *zcong-cancel2*:
  $0 \leq m ==>$
    $zgcd\ (k,\ m) = 1 ==> [k * a = k * b]\ (mod\ m) = [a = b]\ (mod\ m)$
  by (*simp add*: *zmult-commute zcong-cancel*)

lemma *zcong-zgcd-zmult-zmod*:
  $[a = b]\ (mod\ m) ==> [a = b]\ (mod\ n) ==> zgcd\ (m,\ n) = 1$
    $==> [a = b]\ (mod\ m * n)$
  apply (*unfold zcong-def dvd-def*, *auto*)
  apply (*subgoal-tac* $m\ dvd\ n * ka$)
   apply (*subgoal-tac* $m\ dvd\ ka$)
    apply (*case-tac* [*2*] $0 \leq ka$)
 apply (*metis zdvd-mult-div-cancel zdvd-refl zdvd-zminus2-iff zdvd-zmultD2 zgcd-zminus zmult-commute zmult-zminus zrelprime-zdvd-zmult*)
   apply (*metis IntDiv.zdvd-abs1 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 abs-eq-0 int-0-neq-1 mult-le-0-iff zdvd-mono zdvd-mult-cancel zdvd-mult-cancel1 zdvd-refl zdvd-triv-left zdvd-zmult2 zero-le-mult-iff zgcd-greatest-iff zle-anti-sym zle-linear zle-refl zmult-commute zrelprime-zdvd-zmult*)
  apply (*metis zdvd-triv-left*)
  done

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

lemma *zcong-square-zless*:
  $zprime\ p \Longrightarrow 0 < a \Longrightarrow a < p \Longrightarrow$
  $[a * a = 1]\ (mod\ p) \Longrightarrow a = 1 \lor a = p - 1$
  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*:
  $0 < a \Longrightarrow a < m \Longrightarrow 0 < b \Longrightarrow b < a \Longrightarrow \neg [a = b]\ (mod\ m)$
  apply (*unfold zcong-def*)
  apply (*rule zdvd-not-zless*, *auto*)
  done

lemma *zcong-zless-0*:
  $0 \le a \Longrightarrow a < m \Longrightarrow [a = 0]\ (mod\ m) \Longrightarrow a = 0$
  apply (*unfold zcong-def dvd-def*, *auto*)
  apply (*metis div-pos-pos-trivial linorder-not-less zdiv-zmult-self2 zle-refl zle-trans*)
  done

lemma *zcong-zless-unique*:
  $0 < m \Longrightarrow (\exists! b.\ 0 \le b \land b < m \land [a = b]\ (mod\ m))$
  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*: $([a = b]\ (mod\ m)) = (\exists k.\ b = a + m * k)$
  apply (*unfold zcong-def dvd-def*, *auto*)
   apply (*rule-tac* [!] $x = -k$ **in** *exI*, *auto*)
  done

lemma *zgcd-zcong-zgcd*:
  $0 < m \Longrightarrow$
  $zgcd\ (a, m) = 1 \Longrightarrow [a = b]\ (mod\ m) \Longrightarrow zgcd\ (b, m) = 1$
  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*: $0 < m ==> [a = b]\ (mod\ m) = (a\ mod\ m = b\ mod\ m)$
apply *auto*
apply (*metis pos-mod-conj zcong-zless-imp-eq zcong-zmod*)
apply (*metis zcong-refl zcong-zmod*)
done

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

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

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

Remainding 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

3.4 Modulo

lemma *zmod-zdvd-zmod*:
$0 < (m::int) ==> m\ dvd\ b ==> (a\ mod\ b\ mod\ m) = (a\ mod\ m)$
apply (*unfold dvd-def*, *auto*)
apply (*subst zcong-zmod-eq* [*symmetric*])
prefer *2*
apply (*subst zcong-iff-lin*)
apply (*rule-tac* $x = k * (a\ div\ (m * k))$ **in** *exI*)
apply (*simp add*:*zmult-assoc* [*symmetric*], *assumption*)
done

3.5 Extended GCD

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

lemma *xzgcd-correct-aux1*:
$zgcd\ (r', r) = k --> 0 < r -->$
$(\exists\, sn\ tn.\ xzgcda\ (m, n, r', r, s', s, t', t) = (k, sn, tn))$
apply (*rule-tac* $u = m$ **and** $v = n$ **and** $w = r'$ **and** $x = r$ **and** $y = s'$ **and**
$z = s$ **and** $aa = t'$ **and** $ab = t$ **in** *xzgcda.induct*)

apply (*subst zgcd-eq*)
apply (*subst xzgcda.simps*, *auto*)
apply (*case-tac r′ 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*:
(∃ *sn tn*. *xzgcda* (*m*, *n*, *r′*, *r*, *s′*, *s*, *t′*, *t*) = (*k*, *sn*, *tn*)) *−−> 0* < *r −−>*
zgcd (*r′*, *r*) = *k*
apply (*rule-tac u = m* **and** *v = n* **and** *w = r′* **and** *x = r* **and** *y = s′* **and**
z = s **and** *aa = t′* **and** *ab = t* **in** *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 simps zle-refl*)
done

lemma *xzgcd-correct*:
0 < *n ==>* (*zgcd* (*m*, *n*) = *k*) = (∃ *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 ==> r = s ∗ m + t ∗ n
==> (*r′ mod r*) = (*s′ −* (*r′ div r*) *∗ s*) *∗ m* + (*t′ −* (*r′ 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*) ≤ *y ==> x* ≠ *y ==> x* < *y*
by (*rule iffD2* [*OF order-less-le conjI*])

lemma *xzgcda-linear* [*rule-format*]:
$0 < r \text{ --> } xzgcda\ (m,\ n,\ r',\ r,\ s',\ s,\ t',\ t) = (rn,\ sn,\ tn) \text{ --> }$
$r' = s' * m + t' * n \text{ --> } r = s * m + t * n \text{ --> } rn = sn * m + tn * n$
apply (*rule-tac* $u = m$ **and** $v = n$ **and** $w = r'$ **and** $x = r$ **and** $y = s'$ **and** $z = s$ **and** $aa = t'$ **and** $ab = t$ **in** *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* $0 < r'$ *mod* 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*:
$0 < n \text{ ==> } xzgcd\ m\ n = (r,\ s,\ t) \text{ ==> } r = s * m + t * n$
apply (*unfold xzgcd-def*)
apply (*erule xzgcda-linear*, *assumption*, *auto*)
done

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

lemma *zcong-lineq-ex*:
$0 < n \text{ ==> } zgcd\ (a,\ n) = 1 \text{ ==> } \exists x.\ [a * x = 1]\ (mod\ n)$
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 zdvd-zminus-iff*)
done

lemma *zcong-lineq-unique*:
$0 < n \text{ ==> }$
$zgcd\ (a,\ n) = 1 \text{ ==> } \exists!x.\ 0 \le x \wedge x < n \wedge [a * x = b]\ (mod\ n)$
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* $x = x * b$ *mod* 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

4 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]

4.1 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 =
(($\forall i.\ i \le n$ −−> $0 <$ *mf i*) $\wedge$
($\forall i\ j.\ i \le n \wedge j \le n \wedge i \ne j$ −−> *zgcd* (*mf i*, *mf j*) = *1*))

definition
km-cond :: *nat* => (*nat* => *int*) => (*nat* => *int*) => *bool* **where**
km-cond n kf mf = ($\forall i.\ i \le n$ −−> *zgcd* (*kf i*, *mf i*) = *1*)

definition
lincong-sol ::
nat => (*nat* => *int*) => (*nat* => *int*) => (*nat* => *int*) => *int* => *bool*
where

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

lincong-sol n kf bf mf x = (∀ *i*. *i* ≤ *n* --> *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*)
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* (λ*i*. *xilin-sol i n kf bf mf* ∗ *mhf mf n i*) *0 n*

funprod and *funsum*

lemma *funprod-pos*: (∀ *i*. *i* ≤ *n* --> *0* < *mf i*) ==> *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*)]:
(∀ *i*. *k* ≤ *i* ∧ *i* ≤ *k* + *l* --> *zgcd* (*mf i*, *mf m*) = *1*) -->
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 ≤ *i* --> *i* ≤ *k* + *l* --> *mf i dvd funprod mf k l*
apply (*induct l*)
apply *auto*
apply (*rule-tac* [*1*] *zdvd-zmult2*)
apply (*rule-tac* [*2*] *zdvd-zmult*)
apply (*subgoal-tac i* = *Suc* (*k* + *l*))
apply (*simp-all* (*no-asm-simp*))
done

lemma *funsum-mod*:
$\quad$ $funsum\ f\ k\ l\ mod\ m = funsum\ (\lambda i.\ (f\ i)\ mod\ m)\ k\ l\ mod\ m$
apply (*induct l*)
apply *auto*
apply (*rule trans*)
apply (*rule zmod-zadd1-eq*)
apply *simp*
apply (*rule zmod-zadd-right-eq* [*symmetric*])
done

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

lemma *funsum-oneelem* [*rule-format* (*no-asm*)]:
$k \leq j \ {-}{-}{>}\ j \leq k + l \ {-}{-}{>}$
$\quad$ $(\forall i.\ k \leq i \wedge i \leq k + l \wedge i \neq j \ {-}{-}{>}\ f\ i = 0) \ {-}{-}{>}$
$\quad$ $funsum\ f\ k\ l = f\ j$
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*] $j \leq k + l$)
prefer *4*
apply *arith*
apply *auto*
done

4.2 Chinese: uniqueness

lemma *zcong-funprod-aux*:
$m\text{-}cond\ n\ mf \ {=}{=}{>}\ km\text{-}cond\ n\ kf\ mf$
$\quad$ ${=}{=}{>}\ lincong\text{-}sol\ n\ kf\ bf\ mf\ x \ {=}{=}{>}\ lincong\text{-}sol\ n\ kf\ bf\ mf\ y$
$\quad$ ${=}{=}{>}\ [x = y]\ (mod\ mf\ n)$
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 --> km-cond n kf mf -->
    lincong-sol n kf bf mf x --> lincong-sol n kf bf mf y -->
    [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
```

4.3 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] zdvd-zmult2)
    apply (rule-tac [4] zdvd-zmult)
    apply (rule-tac [!] funprod-zdvd)
    apply arith
    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 zdvd-iff-zmod-eq-0 [symmetric])
  apply (rule zdvd-zmult)
  apply (rule x-sol-lin-aux)
  apply auto
  done
```

4.4 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 zmod-zmult-distrib) 3 ⟫)
    apply (tactic ⟪ stac (thm zmod-zdvd-zmod) 3 ⟫)
      apply (tactic ⟪ stac (thm x-sol-lin) 5 ⟫)
        apply (tactic ⟪ stac (thm zmod-zmult-distrib RS sym) 7 ⟫)
        apply (tactic ⟪ stac (thm zcong-zmod RS sym) 7 ⟫)
        apply (subgoal-tac [7]
          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 7
```

```
    apply (simp add: zmult-ac)
   apply (unfold xilin-sol-def)
   apply (tactic ⟨⟨ asm-simp-tac @{simpset} 7 ⟩⟩)
   apply (rule-tac [7] ex1-implies-ex [THEN someI-ex])
   apply (rule-tac [7] unique-xi-sol)
      apply (rule-tac [4] 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
```

5 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*: $P\ a\ a$ ==> $a \notin A$ ==> $A \in$ *bijER* P ==> *insert* $a\ A \in$ *bijER* P
| *insert2*: $P\ a\ b$ ==> $a \neq b$ ==> $a \notin A$ ==> $b \notin A$ ==> $A \in$ *bijER* P
  ==> *insert* a (*insert* b A) $\in$ *bijER* P

bijR

lemma *fin-bijRl*: $(A,\ B) \in$ *bijR* P ==> *finite* A
  apply (*erule bijR.induct*)
  apply *auto*
  done

lemma *fin-bijRr*: $(A,\ B) \in$ *bijR* P ==> *finite* B
  apply (*erule bijR.induct*)
  apply *auto*
  done

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

lemma *inj-func-bijR-aux1*:
    $A \subseteq B$ ==> $a \notin A$ ==> $a \in B$ ==> *inj-on* $f\ B$ ==> $f\ a \notin f$ ‘ A
  apply (*unfold inj-on-def*)
  apply *auto*
  done

lemma *inj-func-bijR-aux2*:
  $\forall a.\ a \in A$ --> $P\ a\ (f\ a)$ ==> *inj-on* $f\ A$ ==> *finite* A ==> F <= A
    ==> $(F,\ f$ ‘ $F) \in$ *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*:
  $\forall a.\ a \in A$ --> $P\ a\ (f\ a)$ ==> *inj-on* $f\ A$ ==> *finite* A
    ==> $(A,\ f$ ‘ $A) \in$ *bijR* P
  apply (*rule inj-func-bijR-aux2*)
     apply *auto*

done

bijER

lemma *fin-bijER*: $A \in bijER\ P \Longrightarrow finite\ A$
apply (*erule bijER.induct*)
apply *auto*
done

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

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

lemma *aux-uniq*: $uniqP\ P \Longrightarrow P\ a\ b \Longrightarrow P\ c\ d \Longrightarrow (a = c) = (b = d)$
apply (*unfold uniqP-def*)
apply *auto*
done

lemma *aux-sym*: $symP\ P \Longrightarrow P\ a\ b = P\ b\ a$
apply (*unfold symP-def*)
apply *auto*
done

lemma *aux-in1*:
$uniqP\ P \Longrightarrow b \notin C \Longrightarrow P\ b\ b \Longrightarrow bijP\ P\ (insert\ b\ C) \Longrightarrow bijP\ P\ C$
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*:
$symP\ P \Longrightarrow uniqP\ P \Longrightarrow a \notin C \Longrightarrow b \notin C \Longrightarrow a \neq b \Longrightarrow P\ a\ b$
$\Longrightarrow bijP\ P\ (insert\ a\ (insert\ b\ C)) \Longrightarrow bijP\ P\ C$
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* −−> *R b* ==> *P a b* ==> *Q a* ==> *R b*
apply *auto*
done

lemma *aux-bij*: *bijP P F* ==> *symP P* ==> *P a b* ==> (*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* ==> *uniqP P* ==> *symP P*
==> $\forall$ *F*. *bijP P F* $\wedge$ *F* $\subseteq$ *A* $\wedge$ *F* $\subseteq$ *B* −−> *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 ∈ F)
   apply (case-tac [!] b ∈ 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
```

6 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.

```
consts
  zfact :: int => int
  d22set :: int => int set

recdef zfact  measure ((λn. nat n) :: int => nat)
  zfact n = (if n ≤ 0 then 1 else n ∗ zfact (n − 1))
```

```
recdef d22set measure ((λa. nat a) :: int => nat)
  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 safe
   prefer 2
   apply (case-tac 1 < a)
    apply (rule-tac prems)
     apply (simp-all (no-asm-simp))
   apply (simp-all (no-asm-simp) add: d22set.simps prems)
  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 (simp-all add: d22set.simps)
  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 *safe*
   apply (*simp add*: *d22set.simps zfact.simps*)
  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

7 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).

7.1 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*

consts
  BnorRset :: *int* ∗ *int* => *int set*

recdef *BnorRset*
  measure ((λ(*a*, *m*). *nat a*) :: *int* ∗ *int* => *nat*)
  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 = (λ*a*. *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
zcongm :: *int* => *int* => *int* => *bool* **where**
zcongm m = (λ*a b*. *zcong a b m*)

lemma *abs-eq-1-iff* [*iff*]: (*abs z* = (*1*::*int*)) = (*z* = *1* ∨ *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*. *0* < (*a*::*int*) ==> *P* (*BnorRset* (*a* − *1*, *m*::*int*)) (*a* − *1*) *m*
    ==> *P* (*BnorRset*(*a*,*m*)) *a m*
shows *P* (*BnorRset*(*u*,*v*)) *u v*
apply (*rule BnorRset.induct*)
apply *safe*
 apply (*case-tac* [*2*] *0* < *a*)
  apply (*rule-tac* [*2*] *prems*)
   apply *simp-all*
 apply (*simp-all add*: *BnorRset.simps prems*)
done

lemma *Bnor-mem-zle* [*rule-format*]: *b* ∈ *BnorRset* (*a*, *m*) ⟶ *b* ≤ *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 ∉ BnorRset (a, m)
  by (auto dest: Bnor-mem-zle)

lemma Bnor-mem-zg [rule-format]: b ∈ BnorRset (a, m) --> 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: zdvd-zminus-iff 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, 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*)
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*:
$1 < m ==> is\text{-}RRset\ A\ m ==> RRset2norRR\ A\ m\ `\ A = norRRset\ m$
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*: $a \notin A ==> inj\ f ==> f\ a \notin f\ `\ A$
by (*unfold inj-on-def*, *auto*)

lemma *Bnor-prod-power* [*rule-format*]:
$x \neq 0 ==> a < m --> \prod((\lambda a.\ a * x)\ `\ BnorRset\ (a, m)) =$
$\prod(BnorRset(a, m)) * x\hat{}card\ (BnorRset\ (a, m))$
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

7.2 Fermat

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

lemma *Bnor-prod-zgcd* [*rule-format*]:
$a < m --> zgcd\ (\prod(BnorRset(a, m)), m) = 1$

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*:
  $0 < m$ ==> *zgcd* (x, m) = *1* ==> [x^(*phi* m) = *1*] (*mod* 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 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*:
$[\![$ *zprime* p; $a < p$ $]\!] \Longrightarrow$ *card* (*BnorRset* (a, p)) = *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 ~: *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*: *zprime* p ==> *phi* p = *nat* ($p - 1$)
apply (*unfold phi-def norRRset-def*)
apply (*rule Bnor-prime*, *auto*)
done

```
theorem Little-Fermat:
    zprime p ==> ¬ p dvd x ==> [x^(nat (p − 1)) = 1] (mod 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
```

8 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).

8.1 Definitions and lemmas

```
definition
  inv :: int => int => int where
  inv p a = (a^(nat (p − 2))) mod p

consts
  wset :: int ∗ int => int set

recdef wset
  measure ((λ(a, p). nat a) :: int ∗ int => nat)
  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*:
$zprime\ p \Longrightarrow 1 < a \Longrightarrow a < p - 1 ==> a \neq inv\ p\ a$
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*:
$zprime\ p \Longrightarrow 1 < a \Longrightarrow a < p - 1 ==> inv\ p\ a \neq 0$
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 zdvd-zminus-iff* [*symmetric*], *auto*)
done

lemma *inv-not-1*:
$zprime\ p \Longrightarrow 1 < a \Longrightarrow a < p - 1 ==> inv\ p\ a \neq 1$
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*:
$[a * (p - 1) = 1]\ (mod\ p) = [a = p - 1]\ (mod\ p)$
apply (*unfold zcong-def*)
apply (*simp add*: *OrderedGroup.diff-diff-eq diff-diff-eq2 zdiff-zmult-distrib2*)
apply (*rule-tac* $s = p\ dvd\ -((a + 1) + (p * -a))$ **in** *trans*)
apply (*simp add*: *mult-commute*)
apply (*subst zdvd-zminus-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 $\Longrightarrow$ *1* < *a* $\Longrightarrow$ *a* < *p* − *1* ==> *inv p a* ≠ *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 $\Longrightarrow$ *1* < *a* $\Longrightarrow$ *a* < *p* − *1* ==> *1* < *inv p a*
 apply (*case-tac 0*≤ *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*, *auto*)
 apply (*unfold inv-def zprime-def*, *simp*)
 done

lemma *inv-less-p-minus-1*:
  zprime p $\Longrightarrow$ *1* < *a* $\Longrightarrow$ *a* < *p* − *1* ==> *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* ≤ *p* ==>
  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*: *zdiff-zmult-distrib zdiff-zmult-distrib2*)
 done

lemma *zcong-zpower-zmult*:
  [*x*ˆ*y* = *1*] (*mod p*) $\Longrightarrow$ [*x*ˆ(*y* ∗ *z*) = *1*] (*mod p*)
 apply (*induct z*)
  apply (*auto simp add*: *zpower-zadd-distrib*)
 apply (*subgoal-tac zcong* (*x*ˆ*y* ∗ *x*ˆ(*y* ∗ *z*)) (*1* ∗ *1*) *p*)
  apply (*rule-tac* [*2*] *zcong-zmult*, *simp-all*)
 done

lemma *inv-inv*: *zprime p* $\Longrightarrow$
  5 ≤ *p* $\Longrightarrow$ *0* < *a* $\Longrightarrow$ *a* < *p* ==> *inv p* (*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 $(a * a\hat{}(nat\ (p - 1) * nat\ (p - 3)))\ (a * 1)\ p$)
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*. $1 < (a::int) \Longrightarrow$
$P\ (wset\ (a - 1,\ p))\ (a - 1)\ p$ ==> $P\ (wset\ (a,\ p))\ a\ p$
shows $P\ (wset\ (u,\ v))\ u\ v$
apply (*rule wset.induct*, *safe*)
prefer *2*
apply (*case-tac* $1 < a$)
apply (*rule prems*)
apply *simp-all*
apply (*simp-all add*: *wset.simps prems*)
done

lemma *wset-mem-imp-or* [*rule-format*]:
$1 < a \Longrightarrow b \notin wset\ (a - 1,\ p)$
==> $b \in wset\ (a,\ p)$ −−> $b = a \vee b = inv\ p\ a$
apply (*subst wset.simps*)
apply (*unfold Let-def*, *simp*)
done

lemma *wset-mem-mem* [*simp*]: $1 < a$ ==> $a \in wset\ (a,\ p)$
apply (*subst wset.simps*)
apply (*unfold Let-def*, *simp*)
done

lemma *wset-subset*: $1 < a \Longrightarrow b \in wset\ (a - 1,\ p)$ ==> $b \in wset\ (a,\ p)$
apply (*subst wset.simps*)
apply (*unfold Let-def*, *auto*)
done

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

zprime p --> a < p − 1 --> b ∈ wset (a, p) --> 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 ∨ 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 --> a < p − 1 --> b ∈ wset (a, p) --> 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 ∨ 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

$\Longrightarrow$ *inv p b* $\in$ *wset* (*a*, *p*) $\Longrightarrow$ *b* $\in$ *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 −−>
$5 \le p$ −−> $a < p - 1$ −−> [($\prod x \in wset(a,\ p).\ x$) = *1*] (*mod p*)
apply (*induct a p rule*: *wset-induct*)
prefer *2*
apply (*subst wset.simps*)
apply (*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

8.2 Wilson

lemma *prime-g-5*: *zprime p* $\Longrightarrow$ *p* $\neq$ *2* $\Longrightarrow$ *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* $\leq$ *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

9 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).

9.1 Definitions and lemmas

definition
  reciR :: *int* => *int* => *int* => *bool* **where**

reciR p = (λ*a b*. *zcong* (*a* $*$ *b*) *1 p* $\wedge$ *1* < *a* $\wedge$ *a* < *p* − *1* $\wedge$ *1* < *b* $\wedge$ *b* < *p* − *1*)

definition
inv :: *int* => *int* => *int* **where**
inv p a =
(*if zprime p* $\wedge$ *0* < *a* $\wedge$ *a* < *p then*
(*SOME x*. *0* $\leq$ *x* $\wedge$ *x* < *p* $\wedge$ *zcong* (*a* $*$ *x*) *1 p*)
else 0)

Inverse

lemma *inv-correct*:
zprime p ==> *0* < *a* ==> *a* < *p*
==> *0* $\leq$ *inv p a* $\wedge$ *inv p a* < *p* $\wedge$ [*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* $\neq$ *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* $\neg$ *p dvd 1*)
apply (*rule-tac* [*2*] *zdvd-not-zless*)
apply (*subgoal-tac p dvd 1*)
prefer *2*
apply (*subst zdvd-zminus-iff* [*symmetric*])
apply *auto*
done

lemma *inv-not-1*:
zprime p ==> *1* < *a* ==> *a* < *p* − *1* ==> *inv p a* $\neq$ *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*: *OrderedGroup.diff-diff-eq diff-diff-eq2 zdiff-zmult-distrib2*)
apply (*rule-tac* $s = p\ dvd\ -((a + 1) + (p * -a))$ **in** *trans*)
apply (*simp add*: *mult-commute*)
apply (*subst zdvd-zminus-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*:
$zprime\ p ==> 1 < a ==> a < p - 1 ==> inv\ p\ a \neq p - 1$
— 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* $inv\ p\ a \neq 1$)
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*)
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 ==> 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 ==> symP (reciR p)*
apply (*unfold reciR-def symP-def*)
apply (*simp add: zmult-commute*)
apply *auto*
done

lemma *bijER-d22set: zprime p ==> d22set (p − 2) ∈ 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

9.2 Wilson

lemma *bijER-zcong-prod-1*:
zprime p ==> A ∈ bijER (reciR p) ==> [$\prod$ *A = 1*] (*mod p*)
apply (*unfold reciR-def*)
apply (*erule bijER.induct*)
apply (*subgoal-tac* [*2*] *a = 1 ∨ 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

10 Finite Sets and Finite Sums

theory *Finite2*
imports *IntFact Infinite-Set*
begin

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

10.1 Useful properties of sums and products

lemma *setsum-same-function-zcong*:
 assumes *a*: $\forall x \in S$. $[f\ x = g\ x](mod\ m)$
 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*: $\forall x \in S$. $[f\ x = g\ x](mod\ m)$
 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*)

10.2 Cardinality of explicit finite sets

lemma *finite-surjI*: [| $B \subseteq f$ ‘ *A*; *finite A* |] ==> *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* $\{y::nat\ .\ y \le x\}$
proof −
 have $\{y::nat\ .\ y \le x\} = \{y::nat\ .\ y < Suc\ x\}$ **by** *auto*
 then show *?thesis* **by** (*auto simp add*: *bdd-nat-set-l-finite*)
qed

lemma *bdd-int-set-l-finite*: *finite* $\{x::int.\ 0 \leq x\ \&\ x < n\}$
  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* $x = nat\ x$ **in** *exI*, *simp*)
  done

lemma *bdd-int-set-le-finite*: *finite* $\{x::int.\ 0 \leq x\ \&\ x \leq n\}$
  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 $\{y.\ y < Suc\ n\} = insert\ n\ \{y.\ y < n\}$
    by *auto*
  then have *card* $\{y.\ y < Suc\ n\} = card\ (insert\ n\ \{y.\ y < n\})$
    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* $\{y.\ y < Suc\ n\} = Suc\ n$
    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 $\{y::nat.\ y \leq x\} = \{\ y::nat.\ y < Suc\ x\}$

by *auto*
then show *?thesis* **by** (*auto simp add*: *card-bdd-nat-set-l*)
qed

lemma *card-bdd-int-set-l*: $0 \leq (n::int)$ ==> *card* $\{y.\ 0 \leq y\ \&\ y < n\} = nat\ n$
proof −
assume $0 \leq n$
have *inj-on* (%*y*. *int y*) $\{y.\ y < nat\ n\}$
by (*auto simp add*: *inj-on-def*)
hence *card* (*int* ' $\{y.\ y < nat\ n\}$) = *card* $\{y.\ y < nat\ n\}$
by (*rule card-image*)
also from ‹$0 \leq n$› **have** *int* ' $\{y.\ y < nat\ n\} = \{y.\ 0 \leq y\ \&\ y < n\}$
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 *card* $\{y.\ y < nat\ n\} = nat\ n$
by (*rule card-bdd-nat-set-l*)
finally show *card* $\{y.\ 0 \leq y\ \&\ y < n\} = nat\ n$ **.**
qed

lemma *card-bdd-int-set-le*: $0 \leq (n::int)$ ==> *card* $\{y.\ 0 \leq y\ \&\ y \leq n\}$ =
$nat\ n + 1$
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*: $0 \leq (n::int)$ ==>
card $\{x.\ 0 < x\ \&\ x \leq n\} = nat\ n$
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 *card* ((%*x*. $x+1$) ' $\{x.\ 0 \leq x\ \&\ x < n\}$) =
card $\{x.\ 0 \leq x\ \&\ x < n\}$
by (*rule card-image*)
also from ‹$0 \leq n$› **have** ... = *nat n*
by (*rule card-bdd-int-set-l*)
also have (%*x*. $x + 1$) ' $\{x.\ 0 \leq x\ \&\ x < n\} = \{x.\ 0 < x\ \&\ x$<= $n\}$
apply (*auto simp add*: *image-def*)
apply (*rule-tac* $x = x - 1$ **in** *exI*)
apply *arith*
done
finally show *card* $\{x.\ 0 < x\ \&\ x \leq n\} = nat\ n$ **.**
qed

lemma *card-bdd-int-set-l-l*: *0* < (*n*::*int*) ==>
  card {*x*. *0* < *x* & *x* < *n*} = *nat n* − *1*
proof −
  assume *0* < *n*
  moreover have {*x*. *0* < *x* & *x* < *n*} = {*x*. *0* < *x* & *x* ≤ *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* ==>
    int(*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* ≤ *n* ==>
    int(*card* {*x*. *0* < *x* & *x* ≤ *n*}) = *n*
  by (*auto simp add*: *card-bdd-int-set-l-le*)

10.3 Cardinality of finite cartesian products

Lemmas for counting arguments.

lemma *setsum-bij-eq*: [| *finite A*; *finite B*; *f* ‘ *A* ⊆ *B*; *inj-on f A*;
    g ‘ *B* ⊆ *A*; *inj-on g B* |] ==> *setsum g B* = *setsum* (*g* ∘ *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*: [| *finite A*; *finite B*; *f* ‘ *A* ⊆ *B*; *inj-on f A*;
    g ‘ *B* ⊆ *A*; *inj-on g B* |] ==> *setprod g B* = *setprod* (*g* ∘ *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

11 Integers: Divisibility and Congruences

theory *Int2* **imports** *Finite2 WilsonRuss* **begin**

definition
  MultInv :: *int* => *int* => *int* **where**
  MultInv p x = *x* ˆ *nat* (*p* − *2*)

11.1 Useful lemmas about dvd and powers

lemma *zpower-zdvd-prop1*:
  0 < *n* ⟹ *p dvd y* ⟹ *p dvd* ((*y*::*int*) ˆ *n*)
  by (*induct n*) (*auto simp add*: *zdvd-zmult zdvd-zmult2* [*of p y*])

lemma *zdvd-bounds*: *n dvd m* ==> *m* ≤ (*0*::*int*) | *n* ≤ *m*
proof −
  assume *n dvd m*
  then have ~(*0* < *m* & *m* < *n*)
    using *zdvd-not-zless* [*of m n*] **by** *auto*
  then show *?thesis* **by** *auto*
qed

lemma *zprime-zdvd-zmult-better*: [| *zprime p*; *p dvd* (*m* ∗ *n*) |] ==>
    (*p dvd m*) | (*p dvd n*)
  apply (*cases 0* ≤ *m*)
  apply (*simp add*: *zprime-zdvd-zmult*)
  apply (*insert zprime-zdvd-zmult* [*of* −*m p n*])
  apply *auto*
  done

lemma *zpower-zdvd-prop2*:
    zprime p ⟹ *p dvd* ((*y*::*int*) ˆ *n*) ⟹ *0* < *n* ⟹ *p dvd y*
  apply (*induct n*)
   apply *simp*
  apply (*frule zprime-zdvd-zmult-better*)
   apply *simp*
  apply *force*
  done

lemma *div-prop1*: [| *0* < *z*; (*x*::*int*) < *y* ∗ *z* |] ==> *x div z* < *y*
proof −
  assume *0* < *z* **then have** *modth*: *x mod z* ≥ *0* **by** *simp*
  have (*x div z*) ∗ *z* ≤ (*x div z*) ∗ *z* **by** *simp*
  then have (*x div z*) ∗ *z* ≤ (*x div z*) ∗ *z* + *x mod z* **using** *modth* **by** *arith*
  also have . . . = *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*: [| $0 < z$; $(x::int) < (y * z) + z$ |] ==> x *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 *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* $y) \leq (x::int)$
proof−
assume $0 < y$
from *zmod-zdiv-equality* **have** $x = y * (x$ *div* $y) + x$ *mod* y **by** *auto*
moreover have $0 \leq x$ *mod* y
by (*auto simp add*: *prems pos-mod-sign*)
ultimately show *?thesis*
by *arith*
qed

11.2 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 zdvd-0-right*)

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

lemma *zcong-zpower*: $[x = y](mod\ m)$ ==> $[x\hat{}z = y\hat{}z](mod\ m)$
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*: $[a = b](mod\ m) ==>$ $([c = d * a](mod\ m) = [c = d * b]\ (mod\ m))$
by (*auto simp add*: *zmult-ac zcong-zmult-prop1*)

lemma *zcong-zmult-prop3*: [| *zprime p*; ~$[x = 0]\ (mod\ p)$; ~$[y = 0]\ (mod\ p)$ |] ==> ~$[x * y = 0]\ (mod\ p)$
apply (*auto simp add*: *zcong-def*)
apply (*drule zprime-zdvd-zmult-better*, *auto*)
done

lemma *zcong-less-eq*: [| $0 < x$; $0 < y$; $0 < m$; $[x = y]\ (mod\ m)$; $x < m$; $y < m$ |] ==> $x = y$
apply (*simp add*: *zcong-zmod-eq*)
apply (*subgoal-tac* $(x\ mod\ m) = x$)
apply (*subgoal-tac* $(y\ mod\ m) = y$)
apply *simp*
apply (*rule-tac* [1−2] *mod-pos-pos-trivial*)
apply *auto*
done

lemma *zcong-neg-1-impl-ne-1*: [| $2 < p$; $[x = -1]\ (mod\ p)$ |] ==> ~$([x = 1]\ (mod\ p))$
proof
assume $2 < p$ **and** $[x = 1]\ (mod\ p)$ **and** $[x = -1]\ (mod\ p)$
then have $[1 = -1]\ (mod\ p)$
apply (*auto simp add*: *zcong-sym*)
apply (*drule zcong-trans*, *auto*)
done
then have $[1 + 1 = -1 + 1]\ (mod\ p)$
by (*simp only*: *zcong-shift*)
then have $[2 = 0]\ (mod\ p)$
by *auto*
then have *p 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] (*mod m*) = (*m dvd a*)
by (*auto simp add*: *zcong-def*)

lemma *zcong-zprime-prod-zero*: [| *zprime p*; $0 < a$ |] ==>
[$a * b = 0$] (*mod p*) ==> [$a = 0$] (*mod p*) | [$b = 0$] (*mod p*)
by (*auto simp add*: *zcong-zero-equiv-div zprime-zdvd-zmult*)

lemma *zcong-zprime-prod-zero-contra*: [| *zprime p*; $0 < a$ |] ==>
~[$a = 0$](*mod p*) & ~[$b = 0$](*mod p*) ==> ~[$a * b = 0$] (*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*: [| $0 < x$; $x < m$ |] ==> ~[$x = 0$] (*mod m*)
by (*auto simp add*: *zcong-zero-equiv-div zdvd-not-zless*)

lemma *zcong-zero*: [| $0 \leq x$; $x < m$; [$x = 0$](*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*; $\forall x \in A$. (*zgcd*(x,y) = 1) |]
==> *zgcd* (*setprod id A*,y) = 1
by (*induct set*: *finite*) (*auto simp add*: *zgcd-zgcd-zmult*)

11.3 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

12 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 = ($\forall$ *y1 y2*. (*y1* $\in$ *X* & *y2* $\in$ *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* ^ *2*) = *x*] (*mod m*)))

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

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

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

12.1 Some useful properties of StandardRes

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

lemma *StandardRes-prop2*: *0* < *m* ==> (*StandardRes m x1* = *StandardRes m x2*)
= ([*x1* = *x2*] (*mod m*))
by (*auto simp add*: *StandardRes-def zcong-zmod-eq*)

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

lemma *StandardRes-prop4*: $2 < m$
$==> [StandardRes \ m \ x * StandardRes \ m \ y = (x * y)] \ (mod \ m)$
by (*auto simp add*: *StandardRes-def zcong-zmod-eq*
zmod-zmult-distrib [*of x y m*])

lemma *StandardRes-lbound*: $0 < p ==> 0 \leq StandardRes \ p \ x$
by (*auto simp add*: *StandardRes-def pos-mod-sign*)

lemma *StandardRes-ubound*: $0 < p ==> StandardRes \ p \ x < p$
by (*auto simp add*: *StandardRes-def pos-mod-bound*)

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

12.2 Relations between StandardRes, SRStar, and SR

lemma *SRStar-SR-prop*: $x \in SRStar \ p ==> x \in SR \ p$
by (*auto simp add*: *SRStar-def SR-def*)

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

lemma *StandardRes-SRStar-prop1*: $2 < p ==> (StandardRes \ p \ x \in SRStar \ p)$
$= (^{\sim}[x = 0] \ (mod \ p))$
apply (*auto simp add*: *StandardRes-prop3 StandardRes-def*
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*: $x \in SRStar \ p ==> {}^{\sim}([x = 0] \ (mod \ p))$
by (*auto simp add*: *SRStar-def zcong-def zdvd-not-zless*)

lemma *StandardRes-SRStar-prop2*: $[| \ 2 < p; \ zprime \ p; \ x \in SRStar \ p \ |]$
$==> StandardRes \ p \ (MultInv \ p \ x) \in SRStar \ p$
apply (*frule-tac* $x = (MultInv \ p \ x)$ **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*: $x \in SRStar \ p ==> StandardRes \ p \ x = x$
by (*auto simp add*: *SRStar-SR-prop StandardRes-SR-prop*)

lemma *StandardRes-SRStar-prop4*: $[| \ zprime \ p; \ 2 < p; \ x \in SRStar \ p \ |]$

==> *StandardRes p x* ∈ *SRStar p*
by (*frule StandardRes-SRStar-prop3*, *auto*)

lemma *SRStar-mult-prop1*: [| *zprime p*; *2* < *p*; *x* ∈ *SRStar p*; *y* ∈ *SRStar p*|]
==> (*StandardRes p* (*x* ∗ *y*)):*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*: [| *zprime p*; *2* < *p*; ~([*a* = *0*](*mod p*));
x ∈ *SRStar p* |]
==> *StandardRes p* (*a* ∗ *MultInv p x*) ∈ *SRStar p*
apply (*frule-tac x* = *x* **in** *StandardRes-SRStar-prop2*, *auto*)
apply (*frule-tac x* = *MultInv p x* **in** *StandardRes-SRStar-prop1*)
apply (*auto simp add*: *StandardRes-SRStar-prop1 zcong-zmult-prop3*)
done

lemma *SRStar-card*: *2* < *p* ==> *int*(*card*(*SRStar p*)) = *p* − *1*
by (*auto simp add*: *SRStar-def int-card-bdd-int-set-l-l*)

lemma *SRStar-finite*: *2* < *p* ==> *finite*(*SRStar p*)
by (*auto simp add*: *SRStar-def bdd-int-set-l-l-finite*)

12.3 Properties relating ResSets with StandardRes

lemma *aux*: *x mod m* = *y mod m* ==> [*x* = *y*] (*mod m*)
apply (*subgoal-tac x* = *y* ==> [*x* = *y*](*mod m*))
apply (*subgoal-tac* [*x mod m* = *y mod m*] (*mod m*) ==> [*x* = *y*] (*mod m*))
apply (*auto simp add*: *zcong-zmod* [*of x y m*])
done

lemma *StandardRes-inj-on-ResSet*: *ResSet m X* ==> (*inj-on* (*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*: [| *finite X*; *0* < *m* |]
==> [*setsum f X* = *setsum* (*StandardRes m o f*) *X*](*mod m*)
apply (*rule-tac F* = *X* **in** *finite-induct*)
apply (*auto intro*!: *zcong-zadd simp add*: *StandardRes-prop1*)
done

lemma *SR-pos*: *0* < *m* ==> (*StandardRes m* ‘ *X*) ⊆ {*x*. *0* ≤ *x* & *x* < *m*}
by (*auto simp add*: *StandardRes-ubound StandardRes-lbound*)

lemma *ResSet-finite*: *0* < *m* ==> *ResSet m X* ==> *finite X*
apply (*rule-tac f* = *StandardRes m* **in** *finite-imageD*)

apply (*rule-tac B* = {*x*. (*0* :: *int*) ≤ *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 mod m*](*mod m*)
by (*auto simp add*: *zcong-zmod*)

lemma *StandardRes-prod*: [| *finite X*; *0* < *m* |]
==> [*setprod f X* = *setprod* (*StandardRes m o f*) *X*] (*mod m*)
apply (*rule-tac F* = *X* **in** *finite-induct*)
apply (*auto intro*!: *zcong-zmult simp add*: *StandardRes-prop1*)
done

lemma *ResSet-image*:
[| *0* < *m*; *ResSet m A*; ∀ *x* ∈ *A*. ∀ *y* ∈ *A*. ([*f x* = *f y*](*mod m*) --> *x* = *y*) |]
==>
ResSet m (*f* ' *A*)
by (*auto simp add*: *ResSet-def*)

12.4 Property for SRStar

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

end

13 Parity: Even and Odd Integers

theory *EvenOdd* **imports** *Int2* **begin**

definition
zOdd :: *int set* **where**
zOdd = {*x*. ∃ *k*. *x* = *2* ∗ *k* + *1*}

definition
zEven :: *int set* **where**
zEven = {*x*. ∃ *k*. *x* = *2* ∗ *k*}

13.1 Some useful properties about even and odd

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

lemma *zEvenI* [*intro?*]: *x* = *2* ∗ *k* ⟹ *x* ∈ *zEven*
and *zEvenE* [*elim?*]: *x* ∈ *zEven* ⟹ (!!*k*. *x* = *2* ∗ *k* ⟹ *C*) ⟹ *C*
by (*auto simp add*: *zEven-def*)

```
lemma one-not-even: ~(1 ∈ zEven)
proof
  assume 1 ∈ zEven
  then obtain k :: int where 1 = 2 * k ..
  then show False by arith
qed

lemma even-odd-conj: ~(x ∈ zOdd & x ∈ 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)):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 ∈ zOdd | x ∈ zEven)
  by (simp add: zOdd-def zEven-def) arith

lemma not-odd-impl-even: ~(x ∈ zOdd) ==> x ∈ zEven
  using even-odd-disj by auto

lemma odd-mult-odd-prop: (x*y):zOdd ==> x ∈ zOdd
proof (rule classical)
  assume ¬ ?thesis
  then have x ∈ zEven by (rule not-odd-impl-even)
  then obtain a where a: x = 2 * a ..
  assume x * y : 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)):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*: $x \in zOdd ==> (x - 1)$:*zEven*
by (*auto simp add*: *zOdd-def zEven-def*)

lemma *even-div-2-prop1*: $x \in zEven ==> (x \ mod \ 2) = 0$
by (*auto simp add*: *zEven-def*)

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

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

lemma *even-times-either*: $x \in zEven ==> x * y \in zEven$
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*: $x * y \in zEven ==> x \in zEven \ | \ y \in zEven$

```
  using odd-iff-not-even odd-times-odd by auto

lemma even-diff: x − y ∈ zEven = ((x ∈ zEven) = (y ∈ zEven))
proof
  assume xy: x − y ∈ zEven
  {
    assume x: x ∈ zEven
    have y ∈ zEven
    proof (rule classical)
      assume ¬ ?thesis
      then have y ∈ zOdd
        by (simp add: odd-iff-not-even)
      with x have x − y ∈ 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 ∈ zEven
    have x ∈ zEven
    proof (rule classical)
      assume ¬ ?thesis
      then have x ∈ zOdd
        by (auto simp add: odd-iff-not-even)
      with y have x − y ∈ zOdd
        by (simp add: odd-minus-even)
      with xy have False
        by (auto simp add: odd-iff-not-even)
      then show ?thesis ..
    qed
  }
  ultimately show (x ∈ zEven) = (y ∈ zEven)
    by (auto simp add: odd-iff-not-even even-minus-even odd-minus-odd
      even-minus-odd odd-minus-even)
next
  assume (x ∈ zEven) = (y ∈ zEven)
  then show x − y ∈ 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 ∈ zEven; 0 ≤ x |] ==> (−1::int)^(nat x) = 1
proof −
  assume x ∈ zEven and 0 ≤ x
  from ‹x ∈ zEven› obtain a where x = 2 ∗ a ..
  with ‹0 ≤ x› have 0 ≤ a by simp
  from ‹0 ≤ x› and ‹x = 2 ∗ a› have nat x = nat (2 ∗ a)
    by simp
```

also from ‹$x = 2 * a$› **have** $nat\ (2 * a) = 2 * nat\ a$
  by (*simp add*: *nat-mult-distrib*)
finally have $(-1::int)\,\hat{}\,nat\ x = (-1)\,\hat{}\,(2 * nat\ a)$
  by *simp*
also have ... $= ((-1::int)\,\hat{}\,2)\,\hat{}\ (nat\ a)$
  by (*simp add*: *zpower-zpower* [*symmetric*])
also have $(-1::int)\,\hat{}\,2 = 1$
  by *simp*
finally show *?thesis*
  by *simp*
qed

lemma *neg-one-odd-power*: [| $x \in zOdd$; $0 \le x$ |] ==> $(-1::int)\,\hat{}\,(nat\ x) = -1$
proof −
  assume $x \in zOdd$ **and** $0 \le x$
  from ‹$x \in zOdd$› **obtain** a **where** $x = 2 * a + 1$ **..**
  with ‹$0 \le x$› **have** a: $0 \le a$ **by** *simp*
  with ‹$0 \le x$› **and** ‹$x = 2 * a + 1$› **have** $nat\ x = nat\ (2 * a + 1)$
    by *simp*
  also from a **have** $nat\ (2 * a + 1) = 2 * nat\ a + 1$
    by (*auto simp add*: *nat-mult-distrib nat-add-distrib*)
  finally have $(-1::int)\,\hat{}\,nat\ x = (-1)\,\hat{}\,(2 * nat\ a + 1)$
    by *simp*
  also have ... $= ((-1::int)\,\hat{}\,2)\,\hat{}\ (nat\ a) * (-1)\,\hat{}\,1$
    by (*auto simp add*: *zpower-zpower* [*symmetric*] *zpower-zadd-distrib*)
  also have $(-1::int)\,\hat{}\,2 = 1$
    by *simp*
  finally show *?thesis*
    by *simp*
qed

lemma *neg-one-power-parity*: [| $0 \le x$; $0 \le y$; $(x \in zEven) = (y \in zEven)$ |] ==>
    $(-1::int)\,\hat{}\,(nat\ x) = (-1::int)\,\hat{}\,(nat\ y)$
  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*: $2 < m$ ==> ~([$1 = -1$] ($mod\ m$))
  by (*auto simp add*: *zcong-def zdvd-not-zless*)

lemma *even-div-2-l*: [| $y \in zEven$; $x < y$ |] ==> $x\ div\ 2 < y\ div\ 2$
proof −
  assume $y \in zEven$ **and** $x < y$
  from ‹$y \in zEven$› **obtain** k **where** k: $y = 2 * k$ **..**
  with ‹$x < y$› **have** $x < 2 * k$ **by** *simp*
  then have $x\ div\ 2 < k$ **by** (*auto simp add*: *div-prop1*)
  also have $k = (2 * k)\ div\ 2$ **by** *simp*
  finally have $x\ div\ 2 < 2 * k\ div\ 2$ **by** *simp*
  with k **show** *?thesis* **by** *simp*

qed

lemma *even-sum-div-2*: [| $x \in$ *zEven*; $y \in$ *zEven* |] ==> $(x + y)$ *div 2* = x *div 2* + y *div 2*
  by (*auto simp add*: *zEven-def*, *auto simp add*: *zdiv-zadd1-eq*)

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

lemma *zprime-zOdd-eq-grt-2*: *zprime p* ==> ($p \in$ *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 :: *int*) ^ *card A*) $*$ (-1 ^ *card A*) = *1*
  by (*induct set*: *finite*) *auto*

lemma *neg-one-power*: (-1::*int*)^*n* = *1* | (-1::*int*)^*n* = -1
  by (*induct n*) *auto*

lemma *neg-one-power-eq-mod-m*: [| $2 < m$; [(-1::*int*)^*j* = (-1::*int*)^*k*] (*mod m*) |]
    ==> ((-1::*int*)^*j* = (-1::*int*)^*k*)
  using *neg-one-power* [*of j*] **and** *insert neg-one-power* [*of k*]
  by (*auto simp add*: *one-not-neg-one-mod-m zcong-sym*)

end

14 Euler's criterion

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

definition
  MultInvPair :: *int* => *int* => *int* => *int set* **where**
  MultInvPair a p j = {*StandardRes p j*, *StandardRes p* (a $*$ (*MultInv p j*))}

definition
  SetS :: *int* => *int* => *int set set* **where**
  SetS *a p* = (*MultInvPair a p* ‘ *SRStar p*)

14.1 Property for MultInvPair

lemma *MultInvPair-prop1a*:
  [| *zprime p*; *2* < *p*; ~([*a* = *0*](*mod p*));
    X ∈ (*SetS a p*); *Y* ∈ (*SetS a p*);
    ~((*X* ∩ *Y*) = {}) |] ==> *X* = *Y*
  apply (*auto simp add*: *SetS-def*)
  apply (*drule StandardRes-SRStar-prop1a*)+ **defer** *1*
  apply (*drule StandardRes-SRStar-prop1a*)+
  apply (*auto simp add*: *MultInvPair-def StandardRes-prop2 zcong-sym*)
  apply (*drule notE*, *rule MultInv-zcong-prop1*, *auto*)[]
  apply (*drule notE*, *rule MultInv-zcong-prop2*, *auto simp add*: *zcong-sym*)[]
  apply (*drule MultInv-zcong-prop2*, *auto simp add*: *zcong-sym*)[]
  apply (*drule MultInv-zcong-prop3*, *auto simp add*: *zcong-sym*)[]
  apply (*drule MultInv-zcong-prop1*, *auto*)[]
  apply (*drule MultInv-zcong-prop2*, *auto simp add*: *zcong-sym*)[]
  apply (*drule MultInv-zcong-prop2*, *auto simp add*: *zcong-sym*)[]
  apply (*drule MultInv-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)
    apply(subgoal-tac 2 = Suc(Suc(0)))
    apply (erule ssubst)
    apply (auto simp only: power-Suc power-0)
    by auto
  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 (simp only: StandardRes-prop2)
  apply (drule MultInvPair-distinct)
  apply auto back
  done
```

14.2 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: ∀ X ∈ SetS a p. 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) |]  ==>
                        ∀ X ∈ SetS a p. 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*; $\forall X \in S$. *finite* (*X*::*int set*);
$\forall X \in S$. *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*) |]
==>
int(*card*(*SetS a p*)) = (*p* − *1*) *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*; ~([*a* = *0*] (*mod p*));
~(*QuadRes p a*); *x* ∈ (*SetS a p*) |] ==>
[$\prod$ *x* = *a*] (*mod p*)
apply (*auto simp add*: *SetS-def MultInvPair-def*)
apply (*frule StandardRes-SRStar-prop1a*)
apply (*subgoal-tac StandardRes p x* ≠ *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*)] (*mod 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 (*drule-tac a* = *x* ∗ *MultInv p x* **and** *b* = *1* **in** *zcong-zmult-prop2*)
apply (*auto simp add*: *zmult-ac*)
done

lemma *aux1*: [| *0* < *x*; (*x*::*int*) < *a*; *x* ≠ (*a* − *1*) |] ==> *x* < *a* − *1*
by *arith*

lemma *aux2*: [| (*a*::*int*) < *c*; *b* < *c* |] ==> (*a* ≤ *b* | *b* ≤ *a*)
  by *auto*

lemma *SRStar-d22set-prop*: *2* < *p* ⟹ (*SRStar p*) = {*1*} ∪ (*d22set* (*p* − *1*))
  apply (*induct p rule*: *d22set.induct*)
  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*; ~([*a* = *0*] (*mod p*)); ~(*QuadRes p a*) |] ==>
[∏ (*Union* (*SetS a p*)) = *a* ˆ *nat* ((*p* − *1*) *div 2*)] (*mod p*)
proof −
  assume *zprime p* **and** *2* < *p* **and** ~([*a* = *0*] (*mod p*)) **and** ~(*QuadRes p a*)
  then have [∏ (*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*ˆ(*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*: [| *zprime p*; *2* < *p*; ~([*a* = *0*](*mod p*)) |] ==>
∏ (*Union* (*SetS a p*)) = *zfact* (*p* − *1*)
proof −
  assume *zprime p* **and** *2* < *p* **and** ~([*a* = *0*](*mod p*))
  then have ∏ (*Union* (*SetS a p*)) = ∏ (*SRStar p*)
    by (*auto simp add*: *MultInvPair-prop2*)
  also have ... = ∏ ({*1*} ∪ (*d22set* (*p* − *1*)))

by (*auto simp add*: *prems SRStar-d22set-prop*)
also have ... = *zfact*(*p* − *1*)
proof −
have ~(*1* ∈ *d22set* (*p* − *1*)) & *finite*(*d22set* (*p* − *1*))
apply (*insert prems*, *auto*)
apply (*drule d22set-g-1*)
apply (*auto simp add*: *d22set-fin*)
done
then have ∏({*1*} ∪ (*d22set* (*p* − *1*))) = ∏(*d22set* (*p* − *1*))
by *auto*
then show *?thesis*
by (*auto simp add*: *d22set-prod-zfact*)
qed
finally show *?thesis* .
qed

```
lemma zfact-prop: [| zprime p; 2 < p; ~([a = 0] (mod p)); ~(QuadRes p a) |]
==>
                  [zfact (p − 1) = a ^ 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; zprime p; ~([x = 0](mod p));
    ~(QuadRes p x) |] ==>
      [x^(nat (((p) − 1) div 2)) = −1](mod p)
  apply (frule zfact-prop, auto)
  apply (frule Wilson-Russ)
  apply (auto simp add: zcong-sym)
  apply (rule zcong-trans, auto)
  done
```

Prove another part of Euler Criterion:

```
lemma aux-1: 0 < p ==> (a::int) ^ nat (p) = a ∗ a ^ (nat (p) − 1)
proof −
  assume 0 < p
  then have a ^ (nat p) =  a ^ (1 + (nat p − 1))
    by (auto simp add: diff-add-assoc)
  also have ... = (a ^ 1) ∗ a ^ (nat(p) − 1)
    by (simp only: zpower-zadd-distrib)
  also have ... = a ∗ a ^ (nat(p) − 1)
    by auto
  finally show ?thesis .
qed

lemma aux-2: [| (2::int) < p; p ∈ zOdd |] ==> 0 < ((p − 1) div 2)
proof −
```

```
  assume 2 < p and p ∈ zOdd
  then have (p - 1):zEven
    by (auto simp add: zEven-def zOdd-def)
  then have aux-1: 2 * ((p - 1) 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) div 2))
    by (auto simp add: aux-1)
  then have 0 < (2 * ((p - 1) div 2)) div 2
    by auto
  then show ?thesis by auto
qed

lemma Euler-part2:
    [| 2 < p; zprime p; [a = 0] (mod p) |] ==> [0 = a ^ nat ((p - 1) div 2)] (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: [| ~([x = 0] (mod p)); [y ^ 2 = x] (mod p)|] ==> ~(p dvd y)
  apply (subgoal-tac [| ~([x = 0] (mod p)); [y ^ 2 = x] (mod p)|] ==>
    ~([y ^ 2 = 0] (mod p)))
  apply (auto simp add: zcong-sym [of y^2 x p] intro: zcong-trans)
  apply (auto simp add: zcong-eq-zdvd-prop intro: zpower-zdvd-prop1)
  done

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)
  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 (simp add: aux--2)
  apply (frule odd-minus-one-even)
  apply (frule even-div-2-prop2)
  apply (auto intro: Little-Fermat simp add: zprime-zOdd-eq-grt-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

15 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 :: *int set* **where**
  A = {(*x*::*int*). *0* < *x* & *x* ≤ ((*p* − *1*) *div 2*)}

definition
  B :: *int set* **where**
  B = (%*x*. *x* ∗ *a*) ‘ *A*

definition
  C :: *int set* **where**
  C = *StandardRes p* ‘ *B*

definition
  D :: *int set* **where**
  D = *C* ∩ {*x*. *x* ≤ ((*p* − *1*) *div 2*)}

definition
  E :: *int set* **where**
  E = *C* ∩ {*x*. ((*p* − *1*) *div 2*) < *x*}

definition
  F :: *int set* **where**
  F = (%*x*. (*p* − *x*)) ‘ *E*

15.1 Basic properties of p

lemma *p-odd*: $p \in zOdd$
  by (*auto simp add*: *p-prime p-g-2 zprime-zOdd-eq-grt-2*)

lemma *p-g-0*: $0 < p$
  using *p-g-2* **by** *auto*

lemma *int-nat*: $int\ (nat\ ((p - 1)\ div\ 2)) = (p - 1)\ div\ 2$
  using *insert p-g-2* **by** (*auto simp add*: *pos-imp-zdiv-nonneg-iff*)

lemma *p-minus-one-l*: $(p - 1)\ div\ 2 < p$
proof −
  have $(p - 1)\ div\ 2 \leq (p - 1)\ div\ 1$
    by (*rule zdiv-mono2*) (*auto simp add*: *p-g-0*)
  also have $\ldots = p - 1$ **by** *simp*
  finally show *?thesis* **by** *simp*
qed

lemma *p-eq*: $p = (2 * (p - 1)\ div\ 2) + 1$
  using *zdiv-zmult-self2* [*of* $2\ p - 1$] **by** *auto*

lemma (**in** −) *zodd-imp-zdiv-eq*: $x \in zOdd ==> 2 * (x - 1)\ div\ 2 = 2 * ((x - 1)\ div\ 2)$
  apply (*frule odd-minus-one-even*)
  apply (*simp add*: *zEven-def*)
  apply (*subgoal-tac* $2 \neq 0$)
  apply (*frule-tac* $b = 2 :: int$ **and** $a = x - 1$ **in** *zdiv-zmult-self2*)
  apply (*auto simp add*: *even-div-2-prop2*)
  done

lemma *p-eq2*: $p = (2 * ((p - 1)\ div\ 2)) + 1$
  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

15.2 Basic Properties of the Gauss Sets

lemma *finite-A*: *finite* (*A*)
  apply (*auto simp add*: *A-def*)
  apply (*subgoal-tac* $\{x.\ 0 < x\ \&\ x \leq (p - 1)\ div\ 2\} \subseteq \{x.\ 0 \leq x\ \&\ x < 1 + (p - 1)\ div\ 2\}$)
  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*: *card* A = *nat* (($p - 1$) *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*: [$x * a = y * a$] (*mod p*)
  assume *b*: $0 < x$
  assume *c*: $x \leq (p - 1)$ *div* 2
  assume *d*: $0 < y$
  assume *e*: $y \leq (p - 1)$ *div* 2
  from *a p-a-relprime p-prime a-nonzero zcong-cancel* [*of p a x y*]
  have [$x = y$](*mod 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 mod p* = *y* ∗ *a mod p*
assume *b*: *0* < *x*
assume *c*: *x* ≤ (*p* − *1*) *div 2*
assume *d*: *0* < *y*
assume *e*: *y* ≤ (*p* − *1*) *div 2*
assume *f*: *x* ≠ *y*
from *a* **have** [*x* ∗ *a* = *y* ∗ *a*](*mod 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*](*mod 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-contra*)
apply (*auto simp add*: *A-greater-zero*)
done

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

lemma *C-ncong-p*: $x \in C$ ==> $\sim[x = 0](mod\ p)$
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*: $y \in C$ ==> $0 < y$
apply (*auto simp add*: *C-def*)
proof −
fix *x*
assume *a*: $x \in B$
from *p-g-0* **have** $0 \leq StandardRes\ p\ x$
by (*simp add*: *StandardRes-lbound*)
moreover have $\sim[x = 0]\ (mod\ p)$
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*: $x \in D$ ==> $\sim[x = 0](mod\ p)$
by (*auto simp add*: *D-def C-ncong-p*)

lemma *E-ncong-p*: $x \in E$ ==> $\sim[x = 0](mod\ p)$
by (*auto simp add*: *E-def C-ncong-p*)

lemma *F-ncong-p*: $x \in F$ ==> $\sim[x = 0](mod\ p)$
apply (*auto simp add*: *F-def*)
proof −
fix *x* **assume** *a*: $x \in E$ **assume** *b*: $[p - x = 0]\ (mod\ p)$
from *E-ncong-p* **have** $\sim[x = 0]\ (mod\ p)$
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 $\sim[p - x = 0]\ (mod\ p)$
by (*simp add*: *zcong-not-zero*)

from *this* **show** *False* **by** (*simp add*: *b*)
qed

lemma *F-subset*: $F \subseteq \{x.\ 0 < x\ \&\ x \leq ((p - 1)\ div\ 2)\}$
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**
$2 * (p - 1)\ div\ 2 = 2 * ((p - 1)\ div\ 2)$
by *simp*
with *p-eq2* **show** $!!x.\ [|\ (p - 1)\ div\ 2 < StandardRes\ p\ x;\ x \in B\ |]$
$==> p - StandardRes\ p\ x \leq (p - 1)\ div\ 2$
by *simp*
qed

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

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

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

lemma *D-leq*: $x \in D ==> x \leq (p - 1)\ div\ 2$
by (*auto simp add*: *D-eq*)

lemma *F-ge*: $x \in F ==> x \leq (p - 1)\ div\ 2$
apply (*auto simp add*: *F-eq A-def*)
proof −
fix *y*
assume $(p - 1)\ div\ 2 < StandardRes\ p\ (y * a)$
then have $p - StandardRes\ p\ (y * a) < p - ((p - 1)\ div\ 2)$
by *arith*
also from *p-eq2* **have** $... = 2 * ((p - 1)\ div\ 2) + 1 - ((p - 1)\ div\ 2)$
by *auto*
also have $2 * ((p - 1)\ div\ 2) + 1 - (p - 1)\ div\ 2 = (p - 1)\ div\ 2 + 1$
by *arith*
finally show $p - StandardRes\ p\ (y * a) \leq (p - 1)\ div\ 2$
using *zless-add1-eq* [*of* $p - StandardRes\ p\ (y * a)\ (p - 1)\ div\ 2$] **by** *auto*
qed

lemma *all-A-relprime*: $\forall\, x \in A.\ zgcd(x,\ p) = 1$

using *p-prime p-minus-one-l* **by** (*auto simp add*: *A-def zless-zprime-imp-zrelprime*)

lemma *A-prod-relprime*: *zgcd*((*setprod id A*),*p*) = *1*
using *all-A-relprime finite-A* **by** (*simp add*: *all-relprime-prod-relprime*)

15.3 Relationships Between Gauss Sets

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

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

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

lemma *C-card-eq-B*: *card C* = *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* ∩ *E* = {}
by (*auto simp add*: *D-def E-def*)

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

lemma *C-prod-eq-D-times-E*: *setprod id E* ∗ *setprod id D* = *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*: [*setprod id C* = *setprod id B*] (*mod 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* ∪ *D*) ⊆ *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 $p - StandardRes\ p\ (y * a) = StandardRes\ p\ (ya * a)$
  then have $p = StandardRes\ p\ (y * a) + StandardRes\ p\ (ya * a)$
    by *arith*
  moreover have *p dvd p*
    by *auto*
  ultimately have $p\ dvd\ (StandardRes\ p\ (y * a) + StandardRes\ p\ (ya * a))$
    by *auto*
  then have *a*: $[StandardRes\ p\ (y * a) + StandardRes\ p\ (ya * a) = 0]\ (mod\ p)$
    by (*auto simp add*: *zcong-def*)
  have $[y * a = StandardRes\ p\ (y * a)]\ (mod\ p)$
    by (*simp only*: *zcong-sym StandardRes-prop1*)
  moreover have $[ya * a = StandardRes\ p\ (ya * a)]\ (mod\ p)$
    by (*simp only*: *zcong-sym StandardRes-prop1*)
  ultimately have $[y * a + ya * a =$
    $StandardRes\ p\ (y * a) + StandardRes\ p\ (ya * a)]\ (mod\ p)$
    by (*rule zcong-zadd*)
  with *a* **have** $[y * a + ya * a = 0]\ (mod\ p)$
    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 $[a * (y + ya) = 0]\ (mod\ p)$ **.**
  with *p-prime a-nonzero zcong-zprime-prod-zero* [*of p a y + ya*]
    p-a-relprime
  have *a*: $[y + ya = 0]\ (mod\ p)$
    by *auto*
  assume *b*: $y \in A$ **and** *c*: *ya*: *A*
  with *A-def* **have** $0 < y + ya$
    by *auto*
  moreover from *b c A-def* **have** $y + ya \leq (p - 1)\ div\ 2 + (p - 1)\ div\ 2$
    by *auto*
  moreover from *b c p-eq2 A-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-Un-D-card*: $card\ (F \cup D) = nat\ ((p - 1)\ div\ 2)$
proof −
  have $card\ (F \cup D) = card\ E + card\ D$
    by (*auto simp add*: *finite-F finite-D F-D-disj*

card-Un-disjoint F-card-eq-E)
then have *card* $(F \cup D) = $ *card C*
by (*simp add*: *C-card-eq-D-plus-E*)
from *this* **show** *card* $(F \cup D) =$ *nat* $((p - 1)$ *div* $2)$
by (*simp add*: *C-card-eq-B B-card-eq*)
qed

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

lemma *prod-D-F-eq-prod-A*:
(*setprod id D*) $*$ (*setprod id F*) = *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*:
[*setprod id F* = $((-1)$ ^ (*card E*)) $*$ (*setprod id E*)] (*mod p*)
proof −
have *setprod id F* = *setprod id* (*op* − *p* ‘ *E*)
by (*auto simp add*: *F-def*)
then have *setprod id F* = *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*:
[*setprod id F* = *setprod* (*StandardRes p o* (*op* − *p*)) *E*] (*mod p*)
apply *simp*
apply (*insert p-g-0 finite-E*)
by (*auto simp add*: *StandardRes-prod*)
moreover have *a*: $\forall x \in E$. [$p - x = 0 - x$] (*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*: $\forall x \in E$. [*StandardRes p* $(p - x) = p - x$](*mod p*)
apply *clarify*
apply (*simp add*: *StandardRes-prop1 zcong-sym*)
done
moreover have $\forall x \in E$. [*StandardRes p* $(p - x) = - x$](*mod p*)
apply *clarify*
apply (*insert a b*)
apply (*rule-tac b* = $p - x$ **in** *zcong-trans*, *auto*)
done
ultimately have *c*:
[*setprod* (*StandardRes p o* (*op* − *p*)) *E* = *setprod* (*uminus*) *E*](*mod p*)
apply *simp*

apply (*insert finite-E p-g-0*)
apply (*rule setprod-same-function-zcong*
[*of E StandardRes p o* (*op* $-$ *p*) *uminus p*], *auto*)
done
then have *two*: [*setprod id F* = *setprod* (*uminus*) *E*](*mod p*)
apply (*insert one c*)
apply (*rule zcong-trans* [*of setprod id F*
setprod (*StandardRes p o op* $-$ *p*) *E p*
setprod uminus E], *auto*)
done
also have *setprod uminus E* = (*setprod id E*) $*$ (-1) ^(*card E*)
using *finite-E* **by** (*induct set*: *finite*) *auto*
then have *setprod uminus E* = (-1) ^ (*card E*) $*$ (*setprod id E*)
by (*simp add*: *zmult-commute*)
with *two* **show** *?thesis*
by *simp*
qed

15.4 Gauss' Lemma

lemma *aux*: *setprod id A* $*$ -1 ^ *card E* $*$ *a* ^ *card A* $*$ -1 ^ *card E* = *setprod id A* $*$ *a* ^ *card A*
by (*auto simp add*: *finite-E neg-one-special*)

theorem *pre-gauss-lemma*:
[*a* ^ *nat*((*p* $-$ *1*) *div 2*) = (-1) ^ (*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*)
then have [*setprod id A* = ((-1) ^(*card E*) $*$ *setprod id E*) $*$
setprod id D] (*mod p*)
apply (*rule zcong-trans*)
apply (*auto simp add*: *prod-F-zcong zcong-scalar*)
done
then have [*setprod id A* = ((-1) ^(*card E*) $*$ *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) ^(*card E*) $*$ *setprod id B*)] (*mod p*)
apply (*rule zcong-trans*)
apply (*simp add*: *C-B-zcong-prod zcong-scalar2*)
done
then have [*setprod id A* = ((-1) ^(*card E*) $*$
(*setprod id* ((%*x*. *x* $*$ *a*) ' *A*)))] (*mod p*)
by (*simp add*: *B-def*)
then have [*setprod id A* = ((-1) ^(*card E*) $*$ (*setprod* (%*x*. *x* $*$ *a*) *A*))]
(*mod p*)
by (*simp add*:*finite-A inj-on-xa-A setprod-reindex-id*[*symmetric*])

```
  moreover have setprod (%x. x * a) A =
    setprod (%x. a) A * setprod id A
    using finite-A by (induct set: finite) auto
  ultimately have [setprod id A = ((-1)^(card E) * (setprod (%x. a) A *
    setprod id A))] (mod p)
    by simp
  then have [setprod id A = ((-1)^(card E) * a^(card A) *
      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)^(card E) =
      ((-1)^(card E) * a^(card A) * setprod id A * (-1)^(card E))](mod p)
    by (rule zcong-scalar)
  then have [setprod id A * (-1)^(card E) = setprod id A *
      (-1)^(card E) * a^(card A) * (-1)^(card E)](mod p)
    apply (rule zcong-trans)
    apply (simp add: a mult-commute mult-left-commute)
    done
  then have [setprod id A * (-1)^(card E) = setprod id A *
      a^(card A)](mod p)
    apply (rule zcong-trans)
    apply (simp add: aux)
    done
  with this zcong-cancel2 [of p setprod id A -1 ^ card E a ^ card A]
      p-g-0 A-prod-relprime have [-1 ^ card E = a ^ card A](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: (Legendre a p) = (-1) ^ (card E)
proof -
  from Euler-Criterion p-prime p-g-2 have
      [(Legendre a p) = a^(nat (((p) - 1) div 2))] (mod p)
    by auto
  moreover note pre-gauss-lemma
  ultimately have [(Legendre a p) = (-1) ^ (card E)] (mod p)
    by (rule zcong-trans)
  moreover from p-a-relprime have (Legendre a p) = 1 | (Legendre a p) = (-1)
    by (auto simp add: Legendre-def)
  moreover have (-1::int) ^ (card E) = 1 | (-1::int) ^ (card E) = -1
    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

16 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*: $a * setsum\ id\ A =$
  $p * setsum\ (\%x.\ ((x * a)\ div\ p))\ A + setsum\ id\ D + setsum\ id\ E$
proof −
  from *finite-A* **have** $a * setsum\ id\ A = setsum\ (\%x.\ a * x)\ A$
    by (*auto simp add*: *setsum-const-mult id-def*)
  also have $setsum\ (\%x.\ a * x) = setsum\ (\%x.\ x * a)$
    by (*auto simp add*: *zmult-commute*)
  also have $setsum\ (\%x.\ x * a)\ A = setsum\ id\ B$
    by (*simp add*: *B-def setsum-reindex-id*[*OF inj-on-xa-A*])
  also have ... $= setsum\ (\%x.\ p * (x\ div\ p) + StandardRes\ p\ x)\ B$
    by (*auto simp add*: *StandardRes-def zmod-zdiv-equality*)
  also have ... $= setsum\ (\%x.\ p * (x\ div\ p))\ B + setsum\ (StandardRes\ p)\ B$
    by (*rule setsum-addf*)
  also have $setsum\ (StandardRes\ p)\ B = setsum\ id\ C$
    by (*auto simp add*: *C-def setsum-reindex-id*[*OF SR-B-inj*])
  also from *C-eq* **have** ... $= setsum\ id\ (D \cup E)$
    by *auto*
  also from *finite-D finite-E* **have** ... $= setsum\ id\ D + setsum\ id\ E$
    by (*rule setsum-Un-disjoint*) (*auto simp add*: *D-def E-def*)
  also have $setsum\ (\%x.\ p * (x\ div\ p))\ B =$
      $setsum\ ((\%x.\ p * (x\ div\ p))\ o\ (\%x.\ (x * a)))\ A$
    by (*auto simp add*: *B-def setsum-reindex inj-on-xa-A*)
  also have ... $= setsum\ (\%x.\ p * ((x * a)\ div\ p))\ A$
    by (*auto simp add*: *o-def*)
  also from *finite-A* **have** $setsum\ (\%x.\ p * ((x * a)\ div\ p))\ A =$
    $p * setsum\ (\%x.\ ((x * a)\ div\ p))\ A$
    by (*auto simp add*: *setsum-const-mult*)
  finally show *?thesis* **by** *arith*
qed

lemma *QRLemma2*: $setsum\ id\ A = p * int\ (card\ E) - setsum\ id\ E +$
  $setsum\ id\ D$
proof −
  from *F-Un-D-eq-A* **have** $setsum\ id\ A = setsum\ id\ (D \cup F)$
    by (*simp add*: *Un-commute*)

also from *F-D-disj finite-D finite-F*
have ... $=$ *setsum id D* $+$ *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** *setsum id* $((\%x.\ (p - x))$ ‘ $E) =$
setsum $(\%x.\ (p - x))$ E
by (*auto simp add*: *setsum-reindex*)
also from *finite-E* **have** *setsum* $(op - p)$ $E =$ *setsum* $(\%x.\ p)$ $E -$ *setsum id E*
by (*auto simp add*: *setsum-subtractf id-def*)
also from *finite-E* **have** *setsum* $(\%x.\ p)$ $E = p * int(card\ E)$
by (*intro setsum-const*)
finally show *?thesis*
by *arith*
qed

lemma *QRLemma3*: $(a - 1) *$ *setsum id A* $=$
$p * (setsum\ (\%x.\ ((x * a)\ div\ p))\ A - int(card\ E)) + 2 * setsum\ id\ E$
proof $-$
have $(a - 1) *$ *setsum id A* $= a *$ *setsum id A* $-$ *setsum id A*
by (*auto simp add*: *zdiff-zmult-distrib*)
also note *QRLemma1*
also from *QRLemma2* **have** $p * (\sum x \in A.\ x * a\ div\ p) + setsum\ id\ D +$
$setsum\ id\ E - setsum\ id\ A =$
$p * (\sum x \in A.\ x * a\ div\ p) + setsum\ id\ D +$
$setsum\ id\ E - (p * int\ (card\ E) - setsum\ id\ E + setsum\ id\ D)$
by *auto*
also have ... $= p * (\sum x \in A.\ x * a\ div\ p) -$
$p * int\ (card\ E) + 2 * setsum\ id\ E$
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*: $a \in zOdd$ ==>
$(-1::int)\hat{}(card\ E) = (-1::int)\hat{}(nat(setsum\ (\%x.\ ((x * a)\ div\ p))\ A))$
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 $0 \le int(card\ E)$
by *auto*
moreover have $0 \le setsum\ (\%x.\ ((x * a)\ div\ p))\ A$
proof (*intro setsum-nonneg*)
show $\forall x \in A.\ 0 \le x * a\ div\ p$
proof
fix x
assume $x \in A$
then have $0 \le x$
by (*auto simp add*: *A-def*)
with *a-nonzero* **have** $0 \le x * a$
by (*auto simp add*: *zero-le-mult-iff*)
with *p-g-2* **show** $0 \le x * a\ div\ p$
by (*auto simp add*: *pos-imp-zdiv-nonneg-iff*)
qed
qed
ultimately have $(-1::int)\hat{}nat((int\ (card\ E))) =$
$(-1)\hat{}nat(((\sum x \in A.\ x * a\ div\ p)))$
by (*intro neg-one-power-parity*, *auto*)
also have $nat\ (int(card\ E)) = card\ E$
by *auto*
finally show *?thesis* **.**
qed

end

lemma *MainQRLemma*: $[|\ a \in zOdd;\ 0 < a;\ \sim([a = 0]\ (mod\ p));\ zprime\ p;\ 2 <$
$p;$
$A = \{x.\ 0 < x\ \&\ x \le (p - 1)\ div\ 2\}\ |]$ ==>
$(Legendre\ a\ p) = (-1::int)\hat{}(nat(setsum\ (\%x.\ ((x * a)\ div\ p))\ A))$
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

16.1 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 = $\{x.\ 0 < x\ \&\ x \leq ((p - 1)\ div\ 2)\ \}$

definition
Q-set :: *int set* **where**
Q-set = $\{x.\ 0 < x\ \&\ x \leq ((q - 1)\ div\ 2)\ \}$

definition
S :: (*int* ∗ *int*) *set* **where**
S = *P-set* <∗> *Q-set*

definition
S1 :: (*int* ∗ *int*) *set* **where**
S1 = $\{\ (x, y).\ (x, y){:}S\ \&\ ((p * y) < (q * x))\ \}$

definition
S2 :: (*int* ∗ *int*) *set* **where**
S2 = $\{\ (x, y).\ (x, y){:}S\ \&\ ((q * x) < (p * y))\ \}$

definition
f1 :: *int* => (*int* ∗ *int*) *set* **where**
f1 j = $\{\ (j1, y).\ (j1, y){:}S\ \&\ j1 = j\ \&\ (y \leq (q * j)\ div\ p)\ \}$

definition
f2 :: *int* => (*int* ∗ *int*) *set* **where**
f2 j = $\{\ (x, j1).\ (x, j1){:}S\ \&\ j1 = j\ \&\ (x \leq (p * j)\ div\ q)\ \}$

lemma *p-fact*: $0 < (p - 1)\ div\ 2$
proof −
from *p-g-2* **have** $2 \leq p - 1$ **by** *arith*

```
  then have 2 div 2 ≤ (p − 1) div 2 by (rule zdiv-mono1, auto)
  then show ?thesis by auto
qed

lemma q-fact: 0 < (q − 1) div 2
proof −
  from q-g-2 have 2 ≤ q − 1 by arith
  then have 2 div 2 ≤ (q − 1) div 2 by (rule zdiv-mono1, auto)
  then show ?thesis by auto
qed

lemma pb-neq-qa: [|1 ≤ b; b ≤ (q − 1) div 2 |] ==>
    (p ∗ b ≠ q ∗ a)
proof
  assume p ∗ b = q ∗ a and 1 ≤ b and b ≤ (q − 1) div 2
  then have q dvd (p ∗ b) by (auto simp add: dvd-def)
  with q-prime p-g-2 have q dvd p | q dvd b
    by (auto simp add: zprime-zdvd-zmult)
  moreover have ~ (q dvd p)
  proof
    assume q dvd p
    with p-prime have q = 1 | q = p
      apply (auto simp add: zprime-def QRTEMP-def)
      apply (drule-tac x = q and R = False in allE)
      apply (simp add: QRTEMP-def)
      apply (subgoal-tac 0 ≤ 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 dvd b by auto
  then have q ≤ b
  proof −
    assume q 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 ≤ (q − 1) div 2 by auto
  then have 2 ∗ q ≤ 2 ∗ ((q − 1) div 2) by arith
  then have 2 ∗ q ≤ q − 1
  proof −
    assume 2 ∗ q ≤ 2 ∗ ((q − 1) div 2)
    with prems have q ∈ zOdd by (auto simp add: QRTEMP-def zprime-zOdd-eq-grt-2)
    with odd-minus-one-even have (q − 1):zEven by auto
    with even-div-2-prop2 have (q − 1) = 2 ∗ ((q − 1) div 2) by auto
    with prems show ?thesis by auto
  qed
  then have p1: q ≤ −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*: $(p - 1)$ *div* $2 =$ *int* (*card* (*P-set*))
using *p-fact* **by** (*auto simp add*: *P-set-def card-bdd-int-set-l-le*)

lemma *Q-set-card*: $(q - 1)$ *div* $2 =$ *int* (*card* (*Q-set*))
using *q-fact* **by** (*auto simp add*: *Q-set-def card-bdd-int-set-l-le*)

lemma *S-card*: $((p - 1)$ *div* $2) * ((q - 1)$ *div* $2) =$ *int* (*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*: $b \leq (q - 1)$ *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*: $((p - 1)\ div\ 2) * ((q - 1)\ div\ 2) =$
  $int(card(S1)) + int(card(S2))$
proof −
  have $((p - 1)\ div\ 2) * ((q - 1)\ div\ 2) = int\ (card(S))$
    by (*auto simp add*: *S-card*)
  also have ... $= int(\ card(S1) + card(S2))$
    apply (*insert S1-finite S2-finite S1-Int-S2-prop S1-Union-S2-prop*)
    apply (*drule card-Un-disjoint*, *auto*)
    done
  also have ... $= int(card(S1)) + int(card(S2))$ **by** *auto*
  finally show *?thesis* **.**
qed

lemma *aux1a*: $[|\ 0 < a;\ a \leq (p - 1)\ div\ 2;$
      $0 < b;\ b \leq (q - 1)\ div\ 2\ |] ==>$
      $(p * b < q * a) = (b \leq q * a\ div\ p)$
proof −
  assume $0 < a$ **and** $a \leq (p - 1)\ div\ 2$ **and** $0 < b$ **and** $b \leq (q - 1)\ div\ 2$
  have $p * b < q * a ==> b \leq q * a\ div\ p$
  proof −
    assume $p * b < q * a$
    then have $p * b \leq q * a$ **by** *auto*
    then have $(p * b)\ div\ p \leq (q * a)\ div\ p$
      by (*rule zdiv-mono1*) (*insert p-g-2*, *auto*)
    then show $b \leq (q * a)\ div\ p$
      apply (*subgoal-tac* $p \neq 0$)
      apply (*frule zdiv-zmult-self2*, *force*)
      apply (*insert p-g-2*, *auto*)
      done
  qed
  moreover have $b \leq q * a\ div\ p ==> p * b < q * a$
  proof −
    assume $b \leq q * a\ div\ p$
    then have $p * b \leq p * ((q * a)\ div\ p)$
      using *p-g-2* **by** (*auto simp add*: *mult-le-cancel-left*)
    also have ... $\leq q * a$
      by (*rule zdiv-leq-prop*) (*insert p-g-2*, *auto*)
    finally have $p * b \leq q * a$ **.**
    then have $p * b < q * a \mid p * b = q * a$
      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 *aux1b*: $[|\ 0 < a;\ a \leq (p - 1)\ div\ 2;$
      $0 < b;\ b \leq (q - 1)\ div\ 2\ |] ==>$

```
                    (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 zdiv-zmult-self2, 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 ≠ 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) div 2)) div p ≤ (q - 1) div 2
proof-
  assume zprime p and zprime q and 2 < p and 2 < q

  then have p ∈ zOdd & q ∈ zOdd
    by (auto simp add:  zprime-zOdd-eq-grt-2)
  then have even1: (p - 1):zEven & (q - 1):zEven
    by (auto simp add: odd-minus-one-even)
  then have even2: (2 * p):zEven & ((q - 1) * p):zEven
    by (auto simp add: zEven-def)
  then have even3: (((q - 1) * p) + (2 * p)):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) div 2 < (((q - 1) * p) + (2 * p)) 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 $((p - 1) * q)$ *div 2* $= q * ((p - 1)$ *div 2*$)$
by (*auto simp add*: *even1 even-prod-div-2*)
also have $(((q - 1) * p) + (2 * p))$ *div 2* $= (((q - 1)$ *div 2*$) * p) + p$
by (*auto simp add*: *even1 even2 even-prod-div-2 even-sum-div-2*)
finally show *?thesis*
apply (*rule-tac* $x = q * ((p - 1)$ *div 2*$)$ **and**
$y = (q - 1)$ *div 2* **in** *div-prop2*)
using *prems* **by** *auto*
qed

lemma *aux3a*: $\forall j \in$ *P-set*. *int* (*card* (*f1 j*)) $= (q * j)$ *div p*
proof
fix *j*
assume *j-fact*: $j \in$ *P-set*
have *int* (*card* (*f1 j*)) $=$ *int* (*card* $\{y.\ y \in$ *Q-set* & $y \leq (q * j)$ *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*)
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*)) $= \{y.\ y \in$ *Q-set* & $y \leq (q * j)$ *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* $\{y.\ 0 < y$ & $y \leq (q * j)$ *div p*$\}$)
proof −
have $\{y.\ y \in$ *Q-set* & $y \leq (q * j)$ *div p*$\} =$
$\{y.\ 0 < y$ & $y \leq (q * j)$ *div p*$\}$
apply (*auto simp add*: *Q-set-def*)
proof −
fix *x*
assume $0 < x$ **and** $x \leq q * j$ *div p*
with *j-fact P-set-def* **have** $j \leq (p - 1)$ *div 2* **by** *auto*
with *q-g-2* **have** $q * j \leq q * ((p - 1)$ *div 2*$)$
by (*auto simp add*: *mult-le-cancel-left*)
with *p-g-2* **have** $q * j$ *div p* $\leq q * ((p - 1)$ *div 2*$)$ *div p*
by (*auto simp add*: *zdiv-mono1*)
also from *prems P-set-def* **have** ... $\leq (q - 1)$ *div 2*
apply *simp*
apply (*insert aux2*)
apply (*simp add*: *QRTEMP-def*)
done

finally show $x \le (q - 1)$ *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** $0 \le j$ **by** *auto*
with *q-g-2* **have** $q * 0 \le q * j$ **by** (*auto simp only*: *mult-left-mono*)
then have $0 \le q * j$ **by** *auto*
then have 0 *div* $p \le (q * j)$ *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*: $\forall j \in$ *Q-set*. *int* (*card* (*f2 j*)) $= (p * j)$ *div* q
proof
fix j
assume *j-fact*: $j \in$ *Q-set*
have *int* (*card* (*f2 j*)) $=$ *int* (*card* $\{y.\ y \in$ *P-set* & $y \le (p * j)$ *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* & $y \le (p * j)$ *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* $\{y.\ 0 < y$ & $y \le (p * j)$ *div* $q\}$)
proof $-$
have $\{y.\ y \in$ *P-set* & $y \le (p * j)$ *div* $q\} =$
$\{y.\ 0 < y$ & $y \le (p * j)$ *div* $q\}$
apply (*auto simp add*: *P-set-def*)
proof $-$
fix x
assume $0 < x$ **and** $x \le p * j$ *div* q
with *j-fact Q-set-def* **have** $j \le (q - 1)$ *div* 2 **by** *auto*
with *p-g-2* **have** $p * j \le p * ((q - 1)$ *div* $2)$
by (*auto simp add*: *mult-le-cancel-left*)

```
      with q-g-2 have p * j div q ≤ p * ((q - 1) div 2) div q
        by (auto simp add: zdiv-mono1)
      also from prems have ... ≤ (p - 1) div 2
        by (auto simp add: aux2 QRTEMP-def)
      finally show x ≤ (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 ≤ j by auto
    with p-g-2 have p * 0 ≤ p * j by (auto simp only: mult-left-mono)
    then have 0 ≤ p * j by auto
    then have 0 div q ≤ (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 ∀ x ∈ P-set. finite (f1 x)
  proof
    fix x
    have f1 x ⊆ S by (auto simp add: f1-def)
    with S-finite show finite (f1 x) by (auto simp add: finite-subset)
  qed
  moreover have (∀ x ∈ P-set. ∀ y ∈ P-set. x ≠ y --> (f1 x) ∩ (f1 y) = {})
    by (auto simp add: f1-def)
  moreover note P-set-finite
  ultimately have int(card (UNION P-set f1)) =
      setsum (%x. int(card (f1 x))) P-set
    by(simp add:card-UN-disjoint int-setsum o-def)
  moreover have S1 = UNION P-set f1
    by (auto simp add: f1-def S-def S1-def S2-def P-set-def Q-set-def aux1a)
  ultimately have int(card (S1)) = setsum (%j. int(card (f1 j))) P-set
    by auto
  also have ... = setsum (%j. q * j div p) P-set
    using aux3a by(fastsimp intro: setsum-cong)
  finally show ?thesis .
qed

lemma S2-card: int (card(S2)) = setsum (%j. (p * j) div q) Q-set
proof -
  have ∀ x ∈ Q-set. finite (f2 x)
```

```
  proof
    fix x
    have f2 x ⊆ S by (auto simp add: f2-def)
    with S-finite show finite (f2 x) by (auto simp add: finite-subset)
  qed
  moreover have (∀ x ∈ Q-set. ∀ y ∈ Q-set. x ≠ y -->
      (f2 x) ∩ (f2 y) = {})
    by (auto simp add: f2-def)
  moreover note Q-set-finite
  ultimately have int(card (UNION Q-set f2)) =
      setsum (%x. int(card (f2 x))) Q-set
    by(simp add:card-UN-disjoint int-setsum o-def)
  moreover have S2 = UNION Q-set f2
    by (auto simp add: f2-def S-def S1-def S2-def P-set-def Q-set-def aux1b)
  ultimately have int(card (S2)) = setsum (%j. int(card (f2 j))) Q-set
    by auto
  also have ... = setsum (%j. p * j div q) Q-set
    using aux3b by(fastsimp intro: setsum-cong)
  finally show ?thesis .
qed

lemma S1-carda: int (card(S1)) =
    setsum (%j. (j * q) div p) P-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-neq: [| 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-neq 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-neq*)
 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