The Factorization Algorithm of Berlekamp and Zassenhaus *

Jose Divasón Sebastiaan Joosten René Thiemann
Akihisa Yamada

December 14, 2021

Abstract

We formalize the Berlekamp-Zassenhaus algorithm for factoring square-free integer polynomials in Isabelle/HOL. We further adapt an existing formalization of Yun's square-free factorization algorithm to integer polynomials, and thus provide an efficient and certified factorization algorithm for arbitrary univariate polynomials.

The algorithm first performs a factorization in the prime field GF(p) and then performs computations in the integer ring modulo p^k, where both p and k are determined at runtime. Since a natural modeling of these structures via dependent types is not possible in Isabelle/HOL, we formalize the whole algorithm using Isabelle's recent addition of local type definitions.

Through experiments we verify that our algorithm factors polynomials of degree 100 within seconds.

Contents

*Supported by FWF (Austrian Science Fund) project Y757.

1 Introduction

Modern algorithms to factor integer polynomials – following Berlekamp and Zassenhaus – work via polynomial factorization over prime fields GF(p) and quotient rings $\mathbb{Z}/p^k\mathbb{Z}$ [2, 3]. Algorithm 1 illustrates the basic structure of such an algorithm.[1]

Algorithm 1: A modern factorization algorithm

Input: Square-free integer polynomial f.
Output: Irreducible factors $f_1, \ldots, f_n$ such that $f = f_1 \cdot \ldots \cdot f_n$.

4 Choose a suitable prime p depending on f.
5 Factor f in GF(p): $f \equiv g_1 \cdot \ldots \cdot g_m \pmod{p}$.
6 Determine a suitable bound d on the degree, depending on $g_1, \ldots, g_m$. Choose an exponent k such that every coefficient of a factor of a given multiple of f in $\mathbb{Z}$ with degree at most d can be uniquely represent by a number below p^k.
7 From step 5 compute the unique factorization $f \equiv h_1 \cdot \ldots \cdot h_m \pmod{p^k}$ via the Hensel lifting.
8 Construct a factorization $f = f_1 \cdot \ldots \cdot f_n$ over the integers where each f_i corresponds to the product of one or more h_j.

In previous work on algebraic numbers [12], we implemented Algorithm 1 in Isabelle/HOL [11] as a function of type *int poly* $\Rightarrow$ *int poly list*, where we chose Berlekamp's algorithm in step 5. However, the algorithm was available only as an oracle, and thus a validity check on the result factorization had to be performed.

In this work we fully formalize the correctness of our implementation.

Theorem 1 (Berlekamp-Zassenhaus' Algorithm)

assumes *square_free* (f :: *int poly*)
  and *degree* $f \neq 0$
  and *berlekamp_zassenhaus_factorization* f = *fs*
shows f = *prod_list fs*
  and $\forall f_i \in$ *set fs*. *irreducible* f_i

[1] Our algorithm starts with step 4, so that section numbers and step-numbers coincide.

To obtain Theorem 1 we perform the following tasks.

- We introduce two formulations of $\mathrm{GF}(p)$ and $\mathbb{Z}/p^k\mathbb{Z}$. We first define a type to represent these domains, employing ideas from HOL multivariate analysis. This is essential for reusing many type-based algorithms from the Isabelle distribution and the AFP (archive of formal proofs). At some points in our developement, the type-based setting is still too restrictive. Hence we also introduce a second formulation which is *locale-based*.

- The prime p in step 4 must be chosen so that f remains square-free in $\mathrm{GF}(p)$. For the termination of the algorithm, we prove that such a prime always exists.

- We explain Berlekamp's algorithm that factors polynomials over prime fields, and formalize its correctness using the type-based representation. Since Isabelle's code generation does not work for the type-based representation of prime fields, we define an implementation of Berlekamp's algorithm which avoids type-based polynomial algorithms and type-based prime fields. The soundness of this implementation is proved via the transfer package [5]: we transform the type-based soundness statement of Berlekamp's algorithm into a statement which speaks solely about integer polynomials. Here, we crucially rely upon local type definitions [9] to eliminate the presence of the type for the prime field $\mathrm{GF}(p)$.

- For step 6 we need to find a bound on the coefficients of the factors of a polynomial. For this purpose, we formalize Mignotte's factor bound. During this formalization task we detected a bug in our previous oracle implementation, which computed improper bounds on the degrees of factors.

- We formalize the Hensel lifting. As for Berlekamp's algorithm, we first formalize basic operations in the type-based setting. Unfortunately, however, this result cannot be extended to the full Hensel lifting. Therefore, we model the Hensel lifting in a locale-based way so that modulo operation is explicitly applied on polynomials.

- For the reconstruction in step 8 we closely follow the description of Knuth [7, page 452]. Here, we use the same representation of polynomials over $\mathbb{Z}/p^k\mathbb{Z}$ as for the Hensel lifting.

- We adapt an existing square-free factorization algorithm from $\mathbb{Q}$ to $\mathbb{Z}$. In combination with the previous results this leads to a factorization algorithm for arbitrary integer and rational polynomials.

To our knowledge, this is the first formalization of the Berlekamp-Zassenhaus algorithm. For instance, Barthe et al. report that there is no formalization of an efficient factorization algorithm over GF(p) available in Coq [1, Section 6, note 3 on formalization].

Some key theorems leading to the algorithm have already been formalized in Isabelle or other proof assistants. In ACL2, for instance, polynomials over a field are shown to be a unique factorization domain (UFD) [4]. A more general result, namely that polynomials over UFD are also UFD, was already developed in Isabelle/HOL for implementing algebraic numbers [12] and an independent development by Eberl is now available in the Isabelle distribution.

An Isabelle formalization of Hensel's lemma is provided by Kobayashi et al. [8], who defined the valuations of polynomials via Cauchy sequences, and used this setup to prove the lemma. Consequently, their result requires a 'valuation ring' as precondition in their formalization. While this extra precondition is theoretically met in our setting, we did not attempt to reuse their results, because the type of polynomials in their formalization (from HOL-Algebra) differs from the polynomials in our development (from HOL/Library). Instead, we formalize a direct proof for Hensel's lemma. Our formalizations are incomparable: On the one hand, Kobayashi et al. did not consider only integer polynomials as we do. On the other hand, we additionally formalize the quadratic Hensel lifting [13], extend the lifting from binary to n-ary factorizations, and prove a uniqueness result, which is required for proving the soundness of Theorem 1.

A Coq formalization of Hensel's lemma is also available, which is used for certifying integral roots and 'hardest-to-round computation' [10]. If one is interested in certifying a factorization, rather than a certified algorithm that performs it, it suffices to test that all the found factors are irreducible. Kirkels [6] formalized a sufficient criterion for this test in Coq: when a polynomial is irreducible modulo some prime, it is also irreducible in $\mathbb{Z}$. Both formalizations are in Coq, and we did not attempt to reuse them.

2 Finite Rings and Fields

We start by establishing some preliminary results about finite rings and finite fields

2.1 Finite Rings

theory *Finite-Field*
imports
  HOL−Computational-Algebra.Primes
  HOL−Number-Theory.Residues
  HOL−Library.Cardinality

```
  Subresultants.Binary-Exponentiation
  Polynomial-Interpolation.Ring-Hom-Poly
begin

typedef ('a::finite) mod-ring = {0..<int CARD('a)} by auto

setup-lifting type-definition-mod-ring

lemma CARD-mod-ring[simp]: CARD('a mod-ring) = CARD('a::finite)
proof −
  have card {y. ∃x∈{0..<int CARD('a)}. (y::'a mod-ring) = Abs-mod-ring x} =
card {0..<int CARD('a)}
  proof (rule bij-betw-same-card)
    have inj-on Rep-mod-ring {y. ∃x∈{0..<int CARD('a)}. y = Abs-mod-ring x}
      by (meson Rep-mod-ring-inject inj-onI)
    moreover have Rep-mod-ring ‘ {y. ∃x∈{0..<int CARD('a)}. (y::'a mod-ring)
= Abs-mod-ring x} = {0..<int CARD('a)}
    proof (auto simp add: image-def Rep-mod-ring-inject)
      fix xb show 0 ≤ Rep-mod-ring (Abs-mod-ring xb)
        using Rep-mod-ring atLeastLessThan-iff by blast
      assume xb1: 0 ≤ xb and xb2: xb < int CARD('a)
      thus  Rep-mod-ring (Abs-mod-ring xb) < int CARD('a)
      by (metis Abs-mod-ring-inverse Rep-mod-ring atLeastLessThan-iff le-less-trans
linear)
      have xb: xb ∈ {0..<int CARD('a)} using xb1 xb2 by simp
      show ∃xa::'a mod-ring. (∃x∈{0..<int CARD('a)}. xa = Abs-mod-ring x) ∧
xb = Rep-mod-ring xa
     by (rule exI[of - Abs-mod-ring xb], auto simp add: xb1 xb2, rule Abs-mod-ring-inverse[OF
xb, symmetric])
    qed
    ultimately show bij-betw Rep-mod-ring
      {y. ∃x∈{0..<int CARD('a)}. (y::'a mod-ring) = Abs-mod-ring x}
      {0..<int CARD('a)}
      by (simp add: bij-betw-def)
  qed
  thus ?thesis
    unfolding type-definition.univ[OF type-definition-mod-ring]
    unfolding image-def by auto
qed

instance mod-ring :: (finite) finite
proof (intro-classes)
  show finite (UNIV::'a mod-ring set)
    unfolding type-definition.univ[OF type-definition-mod-ring]
    using finite by simp
qed


instantiation mod-ring :: (finite) equal
```

begin
lift-definition *equal-mod-ring* :: $'a$ *mod-ring* $\Rightarrow$ $'a$ *mod-ring* $\Rightarrow$ *bool* **is** $(=)$ **.**
instance by (*intro-classes*, *transfer*, *auto*)
end

instantiation *mod-ring* :: (*finite*) *comm-ring*
begin

lift-definition *plus-mod-ring* :: $'a$ *mod-ring* $\Rightarrow$ $'a$ *mod-ring* $\Rightarrow$ $'a$ *mod-ring* **is**
λ *x y*. $(x + y)$ *mod int* $(CARD('a))$ **by** *simp*

lift-definition *uminus-mod-ring* :: $'a$ *mod-ring* $\Rightarrow$ $'a$ *mod-ring* **is**
λ *x*. *if* $x = 0$ *then* 0 *else int* $(CARD('a)) - x$ **by** *simp*

lift-definition *minus-mod-ring* :: $'a$ *mod-ring* $\Rightarrow$ $'a$ *mod-ring* $\Rightarrow$ $'a$ *mod-ring* **is**
λ *x y*. $(x - y)$ *mod int* $(CARD('a))$ **by** *simp*

lift-definition *times-mod-ring* :: $'a$ *mod-ring* $\Rightarrow$ $'a$ *mod-ring* $\Rightarrow$ $'a$ *mod-ring* **is**
λ *x y*. $(x * y)$ *mod int* $(CARD('a))$ **by** *simp*

lift-definition *zero-mod-ring* :: $'a$ *mod-ring* **is** 0 **by** *simp*

instance
by *standard*
(*transfer*; *auto simp add*: *mod-simps algebra-simps intro*: *mod-diff-cong*)+

end

lift-definition *to-int-mod-ring* :: $'a$::*finite mod-ring* $\Rightarrow$ *int* **is** λ *x*. *x* **.**

lift-definition *of-int-mod-ring* :: *int* $\Rightarrow$ $'a$::*finite mod-ring* **is**
λ *x*. *x mod int* $(CARD('a))$ **by** *simp*

interpretation *to-int-mod-ring-hom*: *inj-zero-hom to-int-mod-ring*
by (*unfold-locales*; *transfer*, *auto*)

lemma *int-nat-card*[*simp*]: *int* (*nat* $CARD('a$::*finite*$)) = CARD('a)$ **by** *auto*

interpretation *of-int-mod-ring-hom*: *zero-hom of-int-mod-ring*
by (*unfold-locales*, *transfer*, *auto*)

lemma *of-int-mod-ring-to-int-mod-ring*[*simp*]:
of-int-mod-ring (*to-int-mod-ring x*) $= x$ **by** (*transfer*, *auto*)

lemma *to-int-mod-ring-of-int-mod-ring*[*simp*]: $0 \leq x \Longrightarrow x <$ *int* $CARD('a$ :: *finite*$) \Longrightarrow$
to-int-mod-ring (*of-int-mod-ring x* :: $'a$ *mod-ring*) $= x$
by (*transfer*, *auto*)

lemma *range-to-int-mod-ring*:
range (*to-int-mod-ring* :: ('*a* :: *finite mod-ring* ⇒ *int*)) = {*0* ..< *CARD*('*a*)}
apply (*intro equalityI subsetI*)
apply (*elim rangeE*, *transfer*, *force*)
by (*auto intro*!: *range-eqI to-int-mod-ring-of-int-mod-ring*[*symmetric*])

2.2 Nontrivial Finite Rings

class *nontriv* = **assumes** *nontriv*: *CARD*('*a*) > *1*

subclass(**in** *nontriv*) *finite* **by**(*intro-classes*,*insert nontriv*,*auto intro*:*card-ge-0-finite*)

instantiation *mod-ring* :: (*nontriv*) *comm-ring-1*
begin

lift-definition *one-mod-ring* :: '*a mod-ring* **is** *1* **using** *nontriv*[**where** *?'a*='*a*] **by** *auto*

instance by (*intro-classes*; *transfer*, *simp*)

end

interpretation *to-int-mod-ring-hom*: *inj-one-hom to-int-mod-ring*
by (*unfold-locales*, *transfer*, *simp*)

lemma *of-nat-of-int-mod-ring* [*code-unfold*]:
of-nat = *of-int-mod-ring o int*
proof (*rule ext*, *unfold o-def*)
show *of-nat n* = *of-int-mod-ring* (*int n*) **for** *n*
proof (*induct n*)
case (*Suc n*)
show *?case*
by (*simp only*: *of-nat-Suc Suc*, *transfer*) (*simp add*: *mod-simps*)
qed *simp*
qed

lemma *of-nat-card-eq-0*[*simp*]: (*of-nat* (*CARD*('*a*::*nontriv*)) :: '*a mod-ring*) = *0*
by (*unfold of-nat-of-int-mod-ring*, *transfer*, *auto*)

lemma *of-int-of-int-mod-ring*[*code-unfold*]: *of-int* = *of-int-mod-ring*
proof (*rule ext*)
fix *x* :: *int*
obtain *n1 n2* **where** *x*: *x* = *int n1* − *int n2* **by** (*rule int-diff-cases*)
show *of-int x* = *of-int-mod-ring x*
unfolding *x of-int-diff of-int-of-nat-eq of-nat-of-int-mod-ring o-def*
by (*transfer*, *simp add*: *mod-diff-right-eq mod-diff-left-eq*)
qed

unbundle *lifting-syntax*

lemma *pcr-mod-ring-to-int-mod-ring*: *pcr-mod-ring* = ($\lambda x\ y.\ x$ = *to-int-mod-ring* *y*)
unfolding *mod-ring.pcr-cr-eq* **unfolding** *cr-mod-ring-def to-int-mod-ring.rep-eq*
..

lemma [*transfer-rule*]:
((=) ===> *pcr-mod-ring*) (λ *x. int x mod int* (*CARD*(*'a* :: *nontriv*))) (*of-nat* :: *nat* ⇒ *'a mod-ring*)
by (*intro rel-funI*, *unfold pcr-mod-ring-to-int-mod-ring of-nat-of-int-mod-ring*, *transfer*, *auto*)

lemma [*transfer-rule*]:
((=) ===> *pcr-mod-ring*) (λ *x. x mod int* (*CARD*(*'a* :: *nontriv*))) (*of-int* :: *int* ⇒ *'a mod-ring*)
by (*intro rel-funI*, *unfold pcr-mod-ring-to-int-mod-ring of-int-of-int-mod-ring*, *transfer*, *auto*)

lemma *one-mod-card* [*simp*]: *1 mod CARD*(*'a*::*nontriv*) = *1*
using *mod-less nontriv* **by** *blast*

lemma *Suc-0-mod-card* [*simp*]: *Suc 0 mod CARD*(*'a*::*nontriv*) = *1*
using *one-mod-card* **by** *simp*

lemma *one-mod-card-int* [*simp*]: *1 mod int CARD*(*'a*::*nontriv*) = *1*
proof −
from *nontriv* [**where** *?'a* = *'a*] **have** *int* (*1 mod CARD*(*'a*::*nontriv*)) = *1*
by *simp*
then show *?thesis*
using *of-nat-mod* [*of 1 CARD*(*'a*), **where** *?'a* = *int*] **by** *simp*
qed

lemma *pow-mod-ring-transfer*[*transfer-rule*]:
(*pcr-mod-ring* ===> (=) ===> *pcr-mod-ring*)
(λ*a*::*int*. λ*n*. *a*^*n mod CARD*(*'a*::*nontriv*)) ((^)::*'a mod-ring* ⇒ *nat* ⇒ *'a mod-ring*)
unfolding *pcr-mod-ring-to-int-mod-ring*
proof (*intro rel-funI*,*simp*)
fix *x*::*'a mod-ring* **and** *n*
show *to-int-mod-ring x* ^ *n mod int CARD*(*'a*) = *to-int-mod-ring* (*x* ^ *n*)
proof (*induct n*)
case *0*
thus *?case* **by** *auto*
next
case (*Suc n*)
have *to-int-mod-ring* (*x* ^ *Suc n*) = *to-int-mod-ring* (*x* ∗ *x* ^ *n*) **by** *auto*
also have ... = *to-int-mod-ring x* ∗ *to-int-mod-ring* (*x* ^ *n*) *mod CARD*(*'a*)
unfolding *to-int-mod-ring-def* **using** *times-mod-ring.rep-eq* **by** *auto*
also have ... = *to-int-mod-ring x* ∗ (*to-int-mod-ring x* ^ *n mod CARD*(*'a*)) *mod CARD*(*'a*)

using *Suc.hyps* **by** *auto*
also have ... = *to-int-mod-ring x ^ Suc n mod int CARD('a)*
by (*simp add*: *mod-simps*)
finally show *?case* **..**
qed
qed

lemma *dvd-mod-ring-transfer*[*transfer-rule*]:
((*pcr-mod-ring* :: *int* ⇒ *'a* :: *nontriv mod-ring* ⇒ *bool*) ===>
(*pcr-mod-ring* :: *int* ⇒ *'a mod-ring* ⇒ *bool*) ===> (=))
(λ *i j*. ∃ *k* ∈ {*0..<int CARD('a)*}. *j* = *i* ∗ *k mod int CARD('a)*) (*dvd*)
proof (*unfold pcr-mod-ring-to-int-mod-ring*, *intro rel-funI iffI*)
fix *x y* :: *'a mod-ring* **and** *i j*
assume *i*: *i* = *to-int-mod-ring x* **and** *j*: *j* = *to-int-mod-ring y*
{ assume *x dvd y*
then obtain *z* **where** *y* = *x* ∗ *z* **by** (*elim dvdE*, *auto*)
then have *j* = *i* ∗ *to-int-mod-ring z mod CARD('a)* **by** (*unfold i j*, *transfer*)
with *range-to-int-mod-ring*
show ∃ *k* ∈ {*0..<int CARD('a)*}. *j* = *i* ∗ *k mod CARD('a)* **by** *auto*
}
assume ∃ *k* ∈ {*0..<int CARD('a)*}. *j* = *i* ∗ *k mod CARD('a)*
then obtain *k* **where** *k*: *k* ∈ {*0..<int CARD('a)*} **and** *dvd*: *j* = *i* ∗ *k mod CARD('a)* **by** *auto*
from *k* **have** *to-int-mod-ring* (*of-int k* :: *'a mod-ring*) = *k* **by** (*transfer*, *auto*)
also from *dvd* **have** *j* = *i* ∗ ... *mod CARD('a)* **by** *auto*
finally have *y* = *x* ∗ (*of-int k* :: *'a mod-ring*) **unfolding** *i j* **using** *k* **by** (*transfer*, *auto*)
then show *x dvd y* **by** *auto*
qed

lemma *Rep-mod-ring-mod*[*simp*]: *Rep-mod-ring* (*a* :: *'a* :: *nontriv mod-ring*) *mod CARD('a)* = *Rep-mod-ring a*
using *Rep-mod-ring*[**where** *'a* = *'a*] **by** *auto*

2.3 Finite Fields

When the domain is prime, the ring becomes a field

class *prime-card* = **assumes** *prime-card*: *prime* (*CARD('a)*)
begin
lemma *prime-card-int*: *prime* (*int* (*CARD('a)*)) **using** *prime-card* **by** *auto*

subclass *nontriv* **using** *prime-card prime-gt-1-nat* **by** (*intro-classes*,*auto*)
end

instantiation *mod-ring* :: (*prime-card*) *field*
begin

definition *inverse-mod-ring* :: *'a mod-ring* ⇒ *'a mod-ring* **where**
inverse-mod-ring x = (*if x* = *0 then 0 else x ^* (*nat* (*CARD('a)* − *2*)))

definition *divide-mod-ring* :: $'a$ *mod-ring* $\Rightarrow$ $'a$ *mod-ring* $\Rightarrow$ $'a$ *mod-ring* **where**
divide-mod-ring x y = $x * ((\lambda c.$ *if* $c = 0$ *then* 0 *else* c ^ (*nat* (*CARD*($'a$) − 2))) y)

instance
proof
fix *a b c*::$'a$ *mod-ring*
show *inverse 0* = (*0*::$'a$ *mod-ring*) **by** (*simp add*: *inverse-mod-ring-def*)
show *a div b* = $a *$ *inverse b*
unfolding *inverse-mod-ring-def* **by** (*transfer'*, *simp add*: *divide-mod-ring-def*)
show $a \neq 0 \Longrightarrow$ *inverse a* $* a = 1$
proof (*unfold inverse-mod-ring-def*, *transfer*)
let *?p*=*CARD*($'a$)
fix *x*
assume *x*: $x \in$ {*0*..<*int CARD*($'a$)} **and** *x0*: $x \neq 0$
have *p0'*: *0*≤*?p* **by** *auto*
have ¬ *?p dvd x*
using *x x0 zdvd-imp-le* **by** *fastforce*
then have ¬ *CARD*($'a$) *dvd nat* |*x*|
by *simp*
with *x* **have** ¬ *CARD*($'a$) *dvd nat x*
by *simp*
have *rw*: *x* ^ *nat* (*int* (*?p* − *2*)) ∗ *x* = *x* ^ *nat* (*?p* − *1*)
proof −
have *p2*: *0* ≤ *int* (*?p*−*2*) **using** *x* **by** *simp*
have *card-rw*: (*CARD*($'a$) − *Suc 0*) = *nat* (*1* + *int* (*CARD*($'a$) − *2*))
using *nat-eq-iff x x0* **by** *auto*
have *x* ^ *nat* (*?p* − *2*)∗*x* = *x* ^ (*Suc* (*nat* (*?p* − *2*))) **by** *simp*
also have ... = *x* ^ (*nat* (*?p* − *1*))
using *Suc-nat-eq-nat-zadd1*[*OF p2*] *card-rw* **by** *auto*
finally show *?thesis* **.**
qed
have [*int* (*nat x* ^ (*CARD*($'a$) − *1*)) = *int 1*] (*mod CARD*($'a$))
using *fermat-theorem* [*OF prime-card* ‹¬ *CARD*($'a$) *dvd nat x*›]
by (*simp only*: *cong-def cong-def of-nat-mod* [*symmetric*])
then have ∗: [*x* ^ (*CARD*($'a$) − *1*) = *1*] (*mod CARD*($'a$))
using *x* **by** *auto*
have *x* ^ (*CARD*($'a$) − *2*) *mod CARD*($'a$) ∗ *x mod CARD*($'a$)
= (*x* ^ *nat* (*CARD*($'a$) − *2*) ∗ *x*) *mod CARD*($'a$) **by** (*simp add*: *mod-simps*)
also have ... = (*x* ^ *nat* (*?p* − *1*) *mod ?p*) **unfolding** *rw* **by** *simp*
also have ... = (*x* ^ (*nat ?p* − *1*) *mod ?p*) **using** *p0'* **by** (*simp add*: *nat-diff-distrib'*)
also have ... = *1*
using ∗ **by** (*simp add*: *cong-def*)
finally show (*if x* = *0 then 0 else x* ^ *nat* (*int* (*CARD*($'a$) − *2*)) *mod CARD*($'a$)) ∗ *x mod CARD*($'a$) = *1*
using *x0* **by** *auto*
qed
qed

end

instantiation *mod-ring* :: (*prime-card*) {*normalization-euclidean-semiring*, *euclidean-ring*}
begin

definition *modulo-mod-ring* :: *'a mod-ring* ⇒ *'a mod-ring* ⇒ *'a mod-ring* **where**
modulo-mod-ring x y = (*if y* = *0 then x else 0*)
definition *normalize-mod-ring* :: *'a mod-ring* ⇒ *'a mod-ring* **where** *normalize-mod-ring*
x = (*if x* = *0 then 0 else 1*)
definition *unit-factor-mod-ring* :: *'a mod-ring* ⇒ *'a mod-ring* **where** *unit-factor-mod-ring*
x = *x*
definition *euclidean-size-mod-ring* :: *'a mod-ring* ⇒ *nat* **where** *euclidean-size-mod-ring*
x = (*if x* = *0 then 0 else 1*)

instance
proof (*intro-classes*)
  fix *a* :: *'a mod-ring* **show** *a* ≠ *0* ⟹ *unit-factor a dvd 1*
    unfolding *dvd-def unit-factor-mod-ring-def* **by** (*intro exI*[*of - inverse a*], *auto*)
qed (*auto simp*: *normalize-mod-ring-def unit-factor-mod-ring-def modulo-mod-ring-def*
    euclidean-size-mod-ring-def field-simps)
end

instantiation *mod-ring* :: (*prime-card*) *euclidean-ring-gcd*
begin

definition *gcd-mod-ring* :: *'a mod-ring* ⇒ *'a mod-ring* ⇒ *'a mod-ring* **where**
gcd-mod-ring = *Euclidean-Algorithm.gcd*
definition *lcm-mod-ring* :: *'a mod-ring* ⇒ *'a mod-ring* ⇒ *'a mod-ring* **where**
lcm-mod-ring = *Euclidean-Algorithm.lcm*
definition *Gcd-mod-ring* :: *'a mod-ring set* ⇒ *'a mod-ring* **where** *Gcd-mod-ring*
= *Euclidean-Algorithm.Gcd*
definition *Lcm-mod-ring* :: *'a mod-ring set* ⇒ *'a mod-ring* **where** *Lcm-mod-ring*
= *Euclidean-Algorithm.Lcm*

instance by (*intro-classes*, *auto simp*: *gcd-mod-ring-def lcm-mod-ring-def Gcd-mod-ring-def*
Lcm-mod-ring-def)
end

instantiation *mod-ring* :: (*prime-card*) *unique-euclidean-ring*
begin

definition [*simp*]: *division-segment-mod-ring* (*x* :: *'a mod-ring*) = (*1* :: *'a mod-ring*)

instance by *intro-classes* (*auto simp*: *euclidean-size-mod-ring-def split*: *if-splits*)

end

instance *mod-ring* :: (*prime-card*) *field-gcd*
  by *intro-classes auto*

lemma *surj-of-nat-mod-ring*: $\exists\ i.\ i < CARD('a :: prime\text{-}card) \wedge (x :: 'a\ mod\text{-}ring) = of\text{-}nat\ i$
  by (*rule exI*[*of - nat* (*to-int-mod-ring x*)], *unfold of-nat-of-int-mod-ring o-def*, *subst nat-0-le*, *transfer*, *simp*, *simp*, *transfer*, *auto*)

lemma *of-nat-0-mod-ring-dvd*: **assumes** *x*: $of\text{-}nat\ x = (0 :: 'a :: prime\text{-}card\ mod\text{-}ring)$
  shows $CARD('a)\ dvd\ x$
proof −
  let $?x = of\text{-}nat\ x :: int$
  from *x* **have** $of\text{-}int\text{-}mod\text{-}ring\ ?x = (0 :: 'a\ mod\text{-}ring)$ **by** (*fold of-int-of-int-mod-ring*, *simp*)
  hence $?x\ mod\ CARD('a) = 0$ **by** (*transfer*, *auto*)
  hence $x\ mod\ CARD('a) = 0$ **by** *presburger*
  thus *?thesis* **unfolding** *mod-eq-0-iff-dvd* **.**
qed

end

3 Arithmetics via Records

We create a locale for rings and fields based on a record that includes all the necessary operations.

theory *Arithmetic-Record-Based*
imports
  HOL−Library.More-List
  HOL−Computational-Algebra.Euclidean-Algorithm
begin
datatype *'a arith-ops-record* = *Arith-Ops-Record*
  ($zero : 'a$)
  ($one : 'a$)
  ($plus : 'a \Rightarrow 'a \Rightarrow 'a$)
  ($times : 'a \Rightarrow 'a \Rightarrow 'a$)
  ($minus : 'a \Rightarrow 'a \Rightarrow 'a$)
  ($uminus : 'a \Rightarrow 'a$)
  ($divide : 'a \Rightarrow 'a \Rightarrow 'a$)
  ($inverse : 'a \Rightarrow 'a$)
  ($modulo : 'a \Rightarrow 'a \Rightarrow 'a$)
  ($normalize : 'a \Rightarrow 'a$)
  ($unit\text{-}factor : 'a \Rightarrow 'a$)
  ($of\text{-}int : int \Rightarrow 'a$)
  ($to\text{-}int : 'a \Rightarrow int$)
  ($DP : 'a \Rightarrow bool$)

hide-const (**open**)
  zero
  one

plus
times
minus
uminus
divide
inverse
modulo
normalize
unit-factor
of-int
to-int
DP

fun *listprod-i* :: *'i arith-ops-record* ⇒ *'i list* ⇒ *'i* **where**
listprod-i ops (*x* # *xs*) = *arith-ops-record.times ops x* (*listprod-i ops xs*)
| *listprod-i ops* [] = *arith-ops-record.one ops*

locale *arith-ops* = **fixes** *ops* :: *'i arith-ops-record* (**structure**)
begin

abbreviation (*input*) *zero* **where** *zero* ≡ *arith-ops-record.zero ops*
abbreviation (*input*) *one* **where** *one* ≡ *arith-ops-record.one ops*
abbreviation (*input*) *plus* **where** *plus* ≡ *arith-ops-record.plus ops*
abbreviation (*input*) *times* **where** *times* ≡ *arith-ops-record.times ops*
abbreviation (*input*) *minus* **where** *minus* ≡ *arith-ops-record.minus ops*
abbreviation (*input*) *uminus* **where** *uminus* ≡ *arith-ops-record.uminus ops*
abbreviation (*input*) *divide* **where** *divide* ≡ *arith-ops-record.divide ops*
abbreviation (*input*) *inverse* **where** *inverse* ≡ *arith-ops-record.inverse ops*
abbreviation (*input*) *modulo* **where** *modulo* ≡ *arith-ops-record.modulo ops*
abbreviation (*input*) *normalize* **where** *normalize* ≡ *arith-ops-record.normalize ops*
abbreviation (*input*) *unit-factor* **where** *unit-factor* ≡ *arith-ops-record.unit-factor ops*
abbreviation (*input*) *DP* **where** *DP* ≡ *arith-ops-record.DP ops*

partial-function (*tailrec*) *gcd-eucl-i* :: *'i* ⇒ *'i* ⇒ *'i* **where**
gcd-eucl-i a b = (*if b* = *zero*
then normalize a else gcd-eucl-i b (*modulo a b*))

partial-function (*tailrec*) *euclid-ext-aux-i* :: *'i* ⇒ *'i* ⇒ *'i* ⇒ *'i* ⇒ *'i* ⇒ *'i* ⇒ (*'i* × *'i*) × *'i* **where**
euclid-ext-aux-i s' s t' t r' r = (
if r = *zero then let c* = *divide one* (*unit-factor r'*) *in* ((*times s' c*, *times t' c*), *normalize r'*)
else let q = *divide r' r*
in euclid-ext-aux-i s (*minus s'* (*times q s*)) *t* (*minus t'* (*times q t*)) *r* (*modulo r' r*))

abbreviation (*input*) *euclid-ext-i* :: $'i \Rightarrow 'i \Rightarrow ('i \times 'i) \times 'i$ **where**
euclid-ext-i ≡ *euclid-ext-aux-i one zero zero one*

end

declare *arith-ops.gcd-eucl-i.simps*[*code*]
declare *arith-ops.euclid-ext-aux-i.simps*[*code*]

unbundle *lifting-syntax*

locale *ring-ops* = *arith-ops ops* **for** *ops* :: *'i arith-ops-record* +
fixes *R* :: *'i* ⇒ *'a* :: *comm-ring-1* ⇒ *bool*
assumes *bi-unique*[*transfer-rule*]: *bi-unique R*
and *right-total*[*transfer-rule*]: *right-total R*
and *zero*[*transfer-rule*]: *R zero 0*
and *one*[*transfer-rule*]: *R one 1*
and *plus*[*transfer-rule*]: (*R* ===> *R* ===> *R*) *plus* (+)
and *minus*[*transfer-rule*]: (*R* ===> *R* ===> *R*) *minus* (−)
and *uminus*[*transfer-rule*]: (*R* ===> *R*) *uminus Groups.uminus*
and *times*[*transfer-rule*]: (*R* ===> *R* ===> *R*) *times* ((∗))
and *eq*[*transfer-rule*]: (*R* ===> *R* ===> (=)) (=) (=)
and *DPR*[*transfer-domain-rule*]: *Domainp R* = *DP*
begin
lemma *left-right-unique*[*transfer-rule*]: *left-unique R right-unique R*
using *bi-unique* **unfolding** *bi-unique-def left-unique-def right-unique-def* **by** *auto*

lemma *listprod-i*[*transfer-rule*]: (*list-all2 R* ===> *R*) (*listprod-i ops*) *prod-list*
proof (*intro rel-funI*, *goal-cases*)
case (*1 xs ys*)
thus *?case*
proof (*induct xs ys rule*: *list-all2-induct*)
case (*Cons x xs y ys*)
note [*transfer-rule*] = *this*
show *?case* **by** *simp transfer-prover*
qed (*simp add*: *one*)
qed
end

locale *idom-ops* = *ring-ops ops R* **for** *ops* :: *'i arith-ops-record* **and**
R :: *'i* ⇒ *'a* :: *idom* ⇒ *bool*

locale *idom-divide-ops* = *idom-ops ops R* **for** *ops* :: *'i arith-ops-record* **and**
R :: *'i* ⇒ *'a* :: *idom-divide* ⇒ *bool* +
assumes *divide*[*transfer-rule*]: (*R* ===> *R* ===> *R*) *divide Rings.divide*

locale *euclidean-semiring-ops* = *idom-ops ops R* **for** *ops* :: *'i arith-ops-record* **and**
R :: *'i* ⇒ *'a* :: {*idom*,*normalization-euclidean-semiring*} ⇒ *bool* +
assumes *modulo*[*transfer-rule*]: (*R* ===> *R* ===> *R*) *modulo* (*mod*)
and *normalize*[*transfer-rule*]: (*R* ===> *R*) *normalize Rings.normalize*

and *unit-factor*[*transfer-rule*]: (*R* ===> *R*) *unit-factor Rings.unit-factor*
begin
lemma *gcd-eucl-i* [*transfer-rule*]: (*R* ===> *R* ===> *R*) *gcd-eucl-i Euclidean-Algorithm.gcd*

proof (*intro rel-funI*, *goal-cases*)
case (*1 x X y Y*)
thus *?case*
proof (*induct X Y arbitrary*: *x y rule*: *Euclidean-Algorithm.gcd.induct*)
case (*1 X Y x y*)
note [*transfer-rule*] = *1*(*2*−)
note *simps* = *gcd-eucl-i.simps*[*of x y*] *Euclidean-Algorithm.gcd.simps*[*of X Y*]
have *eq*: (*y* = *zero*) = (*Y* = *0*) **by** *transfer-prover*
show *?case*
proof (*cases Y* = *0*)
case *True*
hence ∗: *y* = *zero* **using** *eq* **by** *simp*
have *R* (*normalize x*) (*Rings.normalize X*) **by** *transfer-prover*
thus *?thesis* **unfolding** *simps* **unfolding** *True* ∗ **by** *simp*
next
case *False*
with *eq* **have** *yz*: *y* ≠ *zero* **by** *simp*
have *R* (*gcd-eucl-i y* (*modulo x y*)) (*Euclidean-Algorithm.gcd Y* (*X mod Y*))
by (*rule 1*(*1*)[*OF False*], *transfer-prover*+)
thus *?thesis* **unfolding** *simps* **using** *False yz* **by** *simp*
qed
qed
qed
end

locale *euclidean-ring-ops* = *euclidean-semiring-ops ops R* **for** *ops* :: *'i arith-ops-record*
and
R :: *'i* ⇒ *'a* :: {*idom*,*euclidean-ring-gcd*} ⇒ *bool* +
assumes *divide*[*transfer-rule*]: (*R* ===> *R* ===> *R*) *divide* (*div*)
begin
lemma *euclid-ext-aux-i*[*transfer-rule*]:
(*R* ===> *R* ===> *R* ===> *R* ===> *R* ===> *R* ===> *rel-prod* (*rel-prod R R*) *R*) *euclid-ext-aux-i euclid-ext-aux*
proof (*intro rel-funI*, *goal-cases*)
case (*1 z Z a A b B c C x X y Y*)
thus *?case*
proof (*induct Z A B C X Y arbitrary*: *z a b c x y rule*: *euclid-ext-aux.induct*)
case (*1 Z A B C X Y z a b c x y*)
note [*transfer-rule*] = *1*(*2*−)
note *simps* = *euclid-ext-aux-i.simps*[*of z a b c x y*] *euclid-ext-aux.simps*[*of Z A B C X Y*]
have *eq*: (*y* = *zero*) = (*Y* = *0*) **by** *transfer-prover*
show *?case*
proof (*cases Y* = *0*)
case *True*

hence $*$: (y = *zero*) = *True* (Y = *0*) = *True* **using** *eq* **by** *auto*
show *?thesis* **unfolding** *simps* **unfolding** $*$ *if-True*
by *transfer-prover*
next
case *False*
hence $*$: (y = *zero*) = *False* (Y = *0*) = *False* **using** *eq* **by** *auto*
have *XY*: *R* (*modulo x y*) (*X mod Y*) **by** *transfer-prover*
have *YA*: *R* (*minus z* (*times* (*divide x y*) *a*)) (*Z* − *X div Y* $*$ *A*) **by** *transfer-prover*
have *YC*: *R* (*minus b* (*times* (*divide x y*) *c*)) (*B* − *X div Y* $*$ *C*) **by** *transfer-prover*
note [*transfer-rule*] = *1*(*1*)[*OF False refl 1*(*3*) *YA 1*(*5*) *YC 1*(*7*) *XY*]

show *?thesis* **unfolding** *simps* $*$ *if-False Let-def* **by** *transfer-prover*
qed
qed
qed

lemma *euclid-ext-i* [*transfer-rule*]:
(*R* ===> *R* ===> *rel-prod* (*rel-prod R R*) *R*) *euclid-ext-i euclid-ext*
by *transfer-prover*

end

locale *field-ops* = *idom-divide-ops ops R* + *euclidean-semiring-ops ops R* **for** *ops* :: *'i arith-ops-record* **and**
R :: *'i* ⇒ *'a* :: {*field-gcd*} ⇒ *bool* +
assumes *inverse*[*transfer-rule*]: (*R* ===> *R*) *inverse Fields.inverse*

lemma *nth-default-rel*[*transfer-rule*]: (*S* ===> *list-all2 S* ===> (=) ===> *S*) *nth-default nth-default*
proof (*intro rel-funI*, *clarify*, *goal-cases*)
case (*1 x y xs ys - n*)
from *1*(*2*) **show** *?case*
proof (*induct arbitrary*: *n*)
case *Nil*
thus *?case* **using** *1*(*1*) **by** *simp*
next
case (*Cons x y xs ys n*)
thus *?case* **by** (*cases n*, *auto*)
qed
qed

lemma *strip-while-rel*[*transfer-rule*]:
((*A* ===> (=)) ===> *list-all2 A* ===> *list-all2 A*) *strip-while strip-while*
unfolding *strip-while-def*[*abs-def*] **by** *transfer-prover*

lemma *list-all2-last*[*simp*]: *list-all2 A* (*xs* @ [*x*]) (*ys* @ [*y*]) ⟷ *list-all2 A xs ys* ∧

A x y
proof (*cases length xs = length ys*)
  case *True*
  show *?thesis* **by** (*simp add*: *list-all2-append*[*OF True*])
next
  case *False*
  note *len* = *list-all2-lengthD*[*of A*]
  from *len*[*of xs ys*] *len*[*of xs* @ [*x*] *ys* @ [*y*]] *False*
  show *?thesis* **by** *auto*
qed

end

3.1 Finite Fields

We provide four implementations for $GF(p)$ – the field with p elements for some prime p – one by int, one by integers, one by 32-bit numbers and one 64-bit implementation. Correctness of the implementations is proven by transfer rules to the type-based version of $GF(p)$.

theory *Finite-Field-Record-Based*
imports
  Finite-Field
  Arithmetic-Record-Based
  Native-Word.Uint32
  Native-Word.Uint64
  Native-Word.Code-Target-Bits-Int
  HOL−Library.Code-Target-Numeral
begin

definition *mod-nonneg-pos* :: *integer* ⇒ *integer* ⇒ *integer* **where**
  $x \geq 0 \Longrightarrow y > 0 \Longrightarrow$ *mod-nonneg-pos x y* = (*x mod y*)

code-printing — FIXME illusion of partiality
  constant *mod-nonneg-pos* ⇀
        (*SML*) *IntInf.mod*/ (-,/ -)
    and (*Eval*) *IntInf.mod*/ (-,/ -)
    and (*OCaml*) *Z.rem*
    and (*Haskell*) *Prelude.mod*/ (-)/ (-)
    and (*Scala*) !((*k*: *BigInt*) => (*l*: *BigInt*) =>/ (*k* '% *l*))

definition *mod-nonneg-pos-int* :: *int* ⇒ *int* ⇒ *int* **where**
  mod-nonneg-pos-int x y = *int-of-integer* (*mod-nonneg-pos* (*integer-of-int x*) (*integer-of-int y*))

lemma *mod-nonneg-pos-int*[*simp*]: $x \geq 0 \Longrightarrow y > 0 \Longrightarrow$ *mod-nonneg-pos-int x y* = (*x mod y*)
  unfolding *mod-nonneg-pos-int-def* **using** *mod-nonneg-pos-def* **by** *simp*

context
  fixes *p* :: *int*
begin
definition *plus-p* :: *int* ⇒ *int* ⇒ *int* **where**
  plus-p x y ≡ *let z* = *x* + *y in if z* ≥ *p then z* − *p else z*

definition *minus-p* :: *int* ⇒ *int* ⇒ *int* **where**
  minus-p x y ≡ *if y* ≤ *x then x* − *y else x* + *p* − *y*

definition *uminus-p* :: *int* ⇒ *int* **where**
  uminus-p x = (*if x* = *0 then 0 else p* − *x*)

definition *mult-p* :: *int* ⇒ *int* ⇒ *int* **where**
  mult-p x y = (*mod-nonneg-pos-int* (*x* ∗ *y*) *p*)

fun *power-p* :: *int* ⇒ *nat* ⇒ *int* **where**
  power-p x n = (*if n* = *0 then 1 else*
    let (*d*,*r*) = *Divides.divmod-nat n 2*;
      rec = *power-p* (*mult-p x x*) *d in*
    if r = *0 then rec else mult-p rec x*)

In experiments with Berlekamp-factorization (where the prime *p* is usually small), it turned out that taking the below implementation of inverse via exponentiation is faster than the one based on the extended Euclidean algorithm.

definition *inverse-p* :: *int* ⇒ *int* **where**
  inverse-p x = (*if x* = *0 then 0 else power-p x* (*nat* (*p* − *2*)))

definition *divide-p* :: *int* ⇒ *int* ⇒ *int* **where**
  divide-p x y = *mult-p x* (*inverse-p y*)

definition *finite-field-ops-int* :: *int arith-ops-record* **where**
  finite-field-ops-int ≡ *Arith-Ops-Record*
      0
      1
      plus-p
      mult-p
      minus-p
      uminus-p
      divide-p
      inverse-p
      (λ *x y* . *if y* = *0 then x else 0*)
      (λ *x* . *if x* = *0 then 0 else 1*)
      (λ *x* . *x*)
      (λ *x* . *x*)
      (λ *x* . *x*)
      (λ *x*. *0* ≤ *x* ∧ *x* < *p*)

end

context
  fixes *p* :: *uint32*
begin
definition *plus-p32* :: *uint32* ⇒ *uint32* ⇒ *uint32* **where**
  plus-p32 x y ≡ *let z* = *x* + *y in if z* ≥ *p then z* − *p else z*

definition *minus-p32* :: *uint32* ⇒ *uint32* ⇒ *uint32* **where**
  minus-p32 x y ≡ *if y* ≤ *x then x* − *y else* (*x* + *p*) − *y*

definition *uminus-p32* :: *uint32* ⇒ *uint32* **where**
  uminus-p32 x = (*if x* = *0 then 0 else p* − *x*)

definition *mult-p32* :: *uint32* ⇒ *uint32* ⇒ *uint32* **where**
  mult-p32 x y = (*x* ∗ *y mod p*)

lemma *int-of-uint32-shift*: *int-of-uint32* (*drop-bit k n*) = (*int-of-uint32 n*) *div* (*2* ^ *k*)
  apply *transfer*
  apply *transfer*
  apply (*simp add*: *take-bit-drop-bit min-def*)
  apply (*simp add*: *drop-bit-eq-div*)
  done

lemma *int-of-uint32-0-iff*: *int-of-uint32 n* = *0* ⟷ *n* = *0*
  by (*transfer*, *rule uint-0-iff*)

lemma *int-of-uint32-0*: *int-of-uint32 0* = *0* **unfolding** *int-of-uint32-0-iff* **by** *simp*

lemma *int-of-uint32-ge-0*: *int-of-uint32 n* ≥ *0*
  by (*transfer*, *auto*)

lemma *two-32*: *2* ^ *LENGTH*(*32*) = (*4294967296* :: *int*) **by** *simp*

lemma *int-of-uint32-plus*: *int-of-uint32* (*x* + *y*) = (*int-of-uint32 x* + *int-of-uint32 y*) *mod 4294967296*
  by (*transfer*, *unfold uint-word-ariths two-32*, *rule refl*)

lemma *int-of-uint32-minus*: *int-of-uint32* (*x* − *y*) = (*int-of-uint32 x* − *int-of-uint32 y*) *mod 4294967296*
  by (*transfer*, *unfold uint-word-ariths two-32*, *rule refl*)

lemma *int-of-uint32-mult*: *int-of-uint32* (*x* ∗ *y*) = (*int-of-uint32 x* ∗ *int-of-uint32 y*) *mod 4294967296*
  by (*transfer*, *unfold uint-word-ariths two-32*, *rule refl*)

lemma *int-of-uint32-mod*: *int-of-uint32* (*x mod y*) = (*int-of-uint32 x mod int-of-uint32 y*)

by (*transfer*, *unfold uint-mod two-32*, *rule refl*)

lemma *int-of-uint32-inv*: $0 \leq x \Longrightarrow x < 4294967296 \Longrightarrow$ *int-of-uint32* (*uint32-of-int* x) $= x$
by *transfer* (*simp add*: *take-bit-int-eq-self unsigned-of-int*)

context
includes *bit-operations-syntax*
begin

function *power-p32* :: *uint32* $\Rightarrow$ *uint32* $\Rightarrow$ *uint32* **where**
power-p32 x n $=$ (*if n* $=$ *0 then 1 else*
let rec $=$ *power-p32* (*mult-p32 x x*) (*drop-bit 1 n*) *in*
if n AND 1 $=$ *0 then rec else mult-p32 rec x*)
by *pat-completeness auto*

termination
proof $-$
{
fix *n* :: *uint32*
assume $n \neq 0$
with *int-of-uint32-ge-0*[*of n*] *int-of-uint32-0-iff*[*of n*] **have** *int-of-uint32 n* > 0 **by** *auto*
hence $0 <$ *int-of-uint32 n int-of-uint32 n div 2* $<$ *int-of-uint32 n* **by** *auto*
} **note** $* =$ *this*
show *?thesis*
by (*relation measure* (λ (*x*,*n*). *nat* (*int-of-uint32 n*)), *auto simp*: *int-of-uint32-shift* $*$)
qed

end

In experiments with Berlekamp-factorization (where the prime p is usually small), it turned out that taking the below implementation of inverse via exponentiation is faster than the one based on the extended Euclidean algorithm.

definition *inverse-p32* :: *uint32* $\Rightarrow$ *uint32* **where**
inverse-p32 x $=$ (*if x* $=$ *0 then 0 else power-p32 x* ($p - 2$))

definition *divide-p32* :: *uint32* $\Rightarrow$ *uint32* $\Rightarrow$ *uint32* **where**
divide-p32 x y $=$ *mult-p32 x* (*inverse-p32 y*)

definition *finite-field-ops32* :: *uint32 arith-ops-record* **where**
finite-field-ops32 $\equiv$ *Arith-Ops-Record*
0
1
plus-p32
mult-p32
minus-p32

uminus-p32
divide-p32
inverse-p32
$(\lambda\ x\ y\ .\ \mathit{if}\ y = 0\ \mathit{then}\ x\ \mathit{else}\ 0)$
$(\lambda\ x\ .\ \mathit{if}\ x = 0\ \mathit{then}\ 0\ \mathit{else}\ 1)$
$(\lambda\ x\ .\ x)$
uint32-of-int
int-of-uint32
$(\lambda\ x.\ 0 \leq x \wedge x < p)$
end

lemma *shiftr-uint32-code* [*code-unfold*]: *drop-bit 1 x* = (*uint32-shiftr x 1*)
by (*simp add*: *uint32-shiftr-def*)

3.1.1 Transfer Relation

locale *mod-ring-locale* =
fixes *p* :: *int* **and** *ty* :: *'a* :: *nontriv itself*
assumes *p*: *p* = *int CARD('a)*
begin
lemma *nat-p*: *nat p* = *CARD('a)* **unfolding** *p* **by** *simp*
lemma *p2*: $p \geq 2$ **unfolding** *p* **using** *nontriv*[**where** *'a* = *'a*] **by** *auto*
lemma *p2-ident*: *int* (*CARD('a)* − *2*) = *p* − *2* **using** *p2* **unfolding** *p* **by** *simp*

definition *mod-ring-rel* :: *int* ⇒ *'a mod-ring* ⇒ *bool* **where**
mod-ring-rel x x' = (*x* = *to-int-mod-ring x'*)

lemma *Domainp-mod-ring-rel* [*transfer-domain-rule*]:
Domainp (*mod-ring-rel*) = (λ *v*. *v* ∈ {*0* ..< *p*})
proof −
{
fix *v* :: *int*
assume ∗: *0* ≤ *v v* < *p*
have *Domainp mod-ring-rel v*
proof
show *mod-ring-rel v* (*of-int-mod-ring v*) **unfolding** *mod-ring-rel-def* **using** ∗ *p* **by** *auto*
qed
} **note** ∗ = *this*
show *?thesis*
by (*intro ext iffI*, *insert range-to-int-mod-ring*[**where** *'a* = *'a*] ∗, *auto simp*: *mod-ring-rel-def p*)
qed

lemma *bi-unique-mod-ring-rel* [*transfer-rule*]:
bi-unique mod-ring-rel left-unique mod-ring-rel right-unique mod-ring-rel
unfolding *mod-ring-rel-def bi-unique-def left-unique-def right-unique-def*

by *auto*

lemma *right-total-mod-ring-rel* [*transfer-rule*]: *right-total mod-ring-rel*
unfolding *mod-ring-rel-def right-total-def* **by** *simp*

3.1.2 Transfer Rules

lemma *mod-ring-0*[*transfer-rule*]: *mod-ring-rel 0 0* **unfolding** *mod-ring-rel-def* **by** *simp*
lemma *mod-ring-1*[*transfer-rule*]: *mod-ring-rel 1 1* **unfolding** *mod-ring-rel-def* **by** *simp*

lemma *plus-p-mod-def*: **assumes** *x*: $x \in \{0 ..< p\}$ **and** *y*: $y \in \{0 ..< p\}$
shows *plus-p p x y* = ((*x* + *y*) *mod p*)
proof (*cases* $p \leq x + y$)
case *False*
thus *?thesis* **using** *x y* **unfolding** *plus-p-def Let-def* **by** *auto*
next
case *True*
from *True x y* **have** ∗: $p > 0$ $0 \leq x + y - p$ $x + y - p < p$ **by** *auto*
from *True* **have** *id*: *plus-p p x y* = $x + y - p$ **unfolding** *plus-p-def* **by** *auto*
show *?thesis* **unfolding** *id* **using** ∗ **using** *mod-pos-pos-trivial* **by** *fastforce*
qed

lemma *mod-ring-plus*[*transfer-rule*]: (*mod-ring-rel* ===> *mod-ring-rel* ===> *mod-ring-rel*) (*plus-p p*) (+)
proof −
{
fix *x y* :: *'a mod-ring*
have *plus-p p* (*to-int-mod-ring x*) (*to-int-mod-ring y*) = *to-int-mod-ring* (*x* + *y*)
by (*transfer*, *subst plus-p-mod-def*, *auto*, *auto simp*: *p*)
} **note** ∗ = *this*
show *?thesis*
by (*intro rel-funI*, *auto simp*: *mod-ring-rel-def* ∗)
qed

lemma *minus-p-mod-def*: **assumes** *x*: $x \in \{0 ..< p\}$ **and** *y*: $y \in \{0 ..< p\}$
shows *minus-p p x y* = ((*x* − *y*) *mod p*)
proof (*cases* $x - y < 0$)
case *False*
thus *?thesis* **using** *x y* **unfolding** *minus-p-def Let-def* **by** *auto*
next
case *True*
from *True x y* **have** ∗: $p > 0$ $0 \leq x - y + p$ $x - y + p < p$ **by** *auto*
from *True* **have** *id*: *minus-p p x y* = $x - y + p$ **unfolding** *minus-p-def* **by** *auto*

show *?thesis* **unfolding** *id* **using** ∗ **using** *mod-pos-pos-trivial* **by** *fastforce*
qed

lemma *mod-ring-minus*[*transfer-rule*]: (*mod-ring-rel* ===> *mod-ring-rel* ===> *mod-ring-rel*) (*minus-p p*) (−)
proof −
{
fix *x y* :: *'a mod-ring*
have *minus-p p* (*to-int-mod-ring x*) (*to-int-mod-ring y*) = *to-int-mod-ring* (*x* − *y*)
by (*transfer*, *subst minus-p-mod-def*, *auto simp*: *p*)
} **note** ∗ = *this*
show *?thesis*
by (*intro rel-funI*, *auto simp*: *mod-ring-rel-def* ∗)
qed

lemma *mod-ring-uminus*[*transfer-rule*]: (*mod-ring-rel* ===> *mod-ring-rel*) (*uminus-p p*) *uminus*
proof −
{
fix *x* :: *'a mod-ring*
have *uminus-p p* (*to-int-mod-ring x*) = *to-int-mod-ring* (*uminus x*)
by (*transfer*, *auto simp*: *uminus-p-def p*)
} **note** ∗ = *this*
show *?thesis*
by (*intro rel-funI*, *auto simp*: *mod-ring-rel-def* ∗)
qed

lemma *mod-ring-mult*[*transfer-rule*]: (*mod-ring-rel* ===> *mod-ring-rel* ===> *mod-ring-rel*) (*mult-p p*) ((∗))
proof −
{
fix *x y* :: *'a mod-ring*
have *mult-p p* (*to-int-mod-ring x*) (*to-int-mod-ring y*) = *to-int-mod-ring* (*x* ∗ *y*)
by (*transfer*, *auto simp*: *mult-p-def p*)
} **note** ∗ = *this*
show *?thesis*
by (*intro rel-funI*, *auto simp*: *mod-ring-rel-def* ∗)
qed

lemma *mod-ring-eq*[*transfer-rule*]: (*mod-ring-rel* ===> *mod-ring-rel* ===> (=)) (=) (=)
by (*intro rel-funI*, *auto simp*: *mod-ring-rel-def*)

lemma *mod-ring-power*[*transfer-rule*]: (*mod-ring-rel* ===> (=) ===> *mod-ring-rel*) (*power-p p*) (^)
proof (*intro rel-funI*, *clarify*, *unfold binary-power*[*symmetric*], *goal-cases*)
  fix *x y n*
  assume *xy*: *mod-ring-rel x y*
  from *xy* **show** *mod-ring-rel* (*power-p p x n*) (*binary-power y n*)
  proof (*induct y n arbitrary*: *x rule*: *binary-power.induct*)
    case (*1 x n y*)
    note *1*(*2*)[*transfer-rule*]
    show *?case*
    proof (*cases n* = *0*)
      case *True*
      thus *?thesis* **by** (*simp add*: *mod-ring-1*)
    next
      case *False*
      obtain *d r* **where** *id*: *Divides.divmod-nat n 2* = (*d*,*r*) **by** *force*
      let *?int* = *power-p p* (*mult-p p y y*) *d*
      let *?gfp* = *binary-power* (*x* ∗ *x*) *d*
      from *False* **have** *id′*: *?thesis* = (*mod-ring-rel*
        (*if r* = *0 then ?int else mult-p p ?int y*)
        (*if r* = *0 then ?gfp else ?gfp* ∗ *x*))
       unfolding *power-p.simps*[*of - - n*] *binary-power.simps*[*of - n*] *Let-def id split* **by** *simp*
      have [*transfer-rule*]: *mod-ring-rel ?int ?gfp*
        by (*rule 1*(*1*)[*OF False refl id*[*symmetric*]], *transfer-prover*)
      show *?thesis* **unfolding** *id′* **by** *transfer-prover*
    qed
  qed
qed

declare *power-p.simps*[*simp del*]

lemma *ring-finite-field-ops-int*: *ring-ops* (*finite-field-ops-int p*) *mod-ring-rel*
  by (*unfold-locales*, *auto simp*:
  finite-field-ops-int-def
  bi-unique-mod-ring-rel
  right-total-mod-ring-rel
  mod-ring-plus
  mod-ring-minus
  mod-ring-uminus
  mod-ring-mult
  mod-ring-eq
  mod-ring-0
  mod-ring-1
  Domainp-mod-ring-rel)
end

locale *prime-field* = *mod-ring-locale p ty* **for** *p* **and** *ty* :: ′*a* :: *prime-card itself*
begin

lemma *prime*: *prime p* **unfolding** *p* **using** *prime-card*[**where** $'a = 'a$] **by** *simp*

lemma *mod-ring-mod*[*transfer-rule*]:
(*mod-ring-rel* ===> *mod-ring-rel* ===> *mod-ring-rel*) ((λ *x y*. *if y* = *0 then x else 0*)) (*mod*)
proof −
  {
    fix *x y* :: $'a$ *mod-ring*
    have (*if to-int-mod-ring y* = *0 then to-int-mod-ring x else 0*) = *to-int-mod-ring* (*x mod y*)
      unfolding *modulo-mod-ring-def* **by** *auto*
  } **note** ∗ = *this*
  show *?thesis*
    by (*intro rel-funI*, *auto simp*: *mod-ring-rel-def* ∗[*symmetric*])
qed

lemma *mod-ring-normalize*[*transfer-rule*]: (*mod-ring-rel* ===> *mod-ring-rel*) ((λ *x*. *if x* = *0 then 0 else 1*)) *normalize*
proof −
  {
    fix *x* :: $'a$ *mod-ring*
    have (*if to-int-mod-ring x* = *0 then 0 else 1*) = *to-int-mod-ring* (*normalize x*)
      unfolding *normalize-mod-ring-def* **by** *auto*
  } **note** ∗ = *this*
  show *?thesis*
    by (*intro rel-funI*, *auto simp*: *mod-ring-rel-def* ∗[*symmetric*])
qed

lemma *mod-ring-unit-factor*[*transfer-rule*]: (*mod-ring-rel* ===> *mod-ring-rel*) (λ *x*. *x*) *unit-factor*
proof −
  {
    fix *x* :: $'a$ *mod-ring*
    have *to-int-mod-ring x* = *to-int-mod-ring* (*unit-factor x*)
      unfolding *unit-factor-mod-ring-def* **by** *auto*
  } **note** ∗ = *this*
  show *?thesis*
    by (*intro rel-funI*, *auto simp*: *mod-ring-rel-def* ∗[*symmetric*])
qed

lemma *mod-ring-inverse*[*transfer-rule*]: (*mod-ring-rel* ===> *mod-ring-rel*) (*inverse-p p*) *inverse*
proof (*intro rel-funI*)
  fix *x y*

assume [*transfer-rule*]: *mod-ring-rel x y*
show *mod-ring-rel* (*inverse-p p x*) (*inverse y*)
unfolding *inverse-p-def inverse-mod-ring-def*
apply (*transfer-prover-start*)
apply (*transfer-step*)+
apply (*unfold p2-ident*)
apply (*rule refl*)
done
qed

lemma *mod-ring-divide*[*transfer-rule*]: (*mod-ring-rel* ===> *mod-ring-rel* ===> *mod-ring-rel*)
(*divide-p p*) (/)
unfolding *divide-p-def*[*abs-def*] *divide-mod-ring-def*[*abs-def*] *inverse-mod-ring-def*[*symmetric*]
by *transfer-prover*

lemma *mod-ring-rel-unsafe*: **assumes** $x < CARD('a)$
shows *mod-ring-rel* (*int x*) (*of-nat x*) $0 < x \Longrightarrow$ *of-nat* $x \neq (0 :: 'a\ mod\text{-}ring)$
proof −
have *id*: *of-nat* $x =$ (*of-int* (*int x*) :: *'a mod-ring*) **by** *simp*
show *mod-ring-rel* (*int x*) (*of-nat x*) $0 < x \Longrightarrow$ *of-nat* $x \neq (0 :: 'a\ mod\text{-}ring)$
unfolding *id*
unfolding *mod-ring-rel-def*
proof (*auto simp add*: *assms of-int-of-int-mod-ring*)
assume $0 < x$ **with** *assms*
have *of-int-mod-ring* (*int x*) $\neq (0 :: 'a\ mod\text{-}ring)$
by (*metis* (*no-types*) *less-imp-of-nat-less less-irrefl of-nat-0-le-iff of-nat-0-less-iff to-int-mod-ring-hom.hom-zero to-int-mod-ring-of-int-mod-ring*)
thus *of-int-mod-ring* (*int x*) $= (0 :: 'a\ mod\text{-}ring) \Longrightarrow$ *False* **by** *blast*
qed
qed

lemma *finite-field-ops-int*: *field-ops* (*finite-field-ops-int p*) *mod-ring-rel*
by (*unfold-locales*, *auto simp*:
finite-field-ops-int-def
bi-unique-mod-ring-rel
right-total-mod-ring-rel
mod-ring-divide
mod-ring-plus
mod-ring-minus
mod-ring-uminus
mod-ring-inverse
mod-ring-mod
mod-ring-unit-factor
mod-ring-normalize
mod-ring-mult
mod-ring-eq
mod-ring-0

mod-ring-1
Domainp-mod-ring-rel)

end

Once we have proven the soundness of the implementation, we do not care any longer that *'a mod-ring* has been defined internally via lifting. Disabling the transfer-rules will hide the internal definition in further applications of transfer.

lifting-forget *mod-ring.lifting*

For soundness of the 32-bit implementation, we mainly prove that this implementation implements the int-based implementation of the mod-ring.

context *mod-ring-locale*
begin

context fixes *pp* :: *uint32*
assumes *ppp*: $p = $ *int-of-uint32 pp*
and *small*: $p \leq 65535$
begin

lemmas *uint32-simps* =
int-of-uint32-0
int-of-uint32-plus
int-of-uint32-minus
int-of-uint32-mult

definition *urel32* :: *uint32* ⇒ *int* ⇒ *bool* **where** *urel32 x y* = (*y* = *int-of-uint32 x* ∧ $y < p$)

definition *mod-ring-rel32* :: *uint32* ⇒ *'a mod-ring* ⇒ *bool* **where**
mod-ring-rel32 x y = (∃ *z*. *urel32 x z* ∧ *mod-ring-rel z y*)

lemma *urel32-0*: *urel32 0 0* **unfolding** *urel32-def* **using** *p2* **by** (*simp*, *transfer*, *simp*)

lemma *urel32-1*: *urel32 1 1* **unfolding** *urel32-def* **using** *p2* **by** (*simp*, *transfer*, *simp*)

lemma *le-int-of-uint32*: ($x \leq y$) = (*int-of-uint32 x* ≤ *int-of-uint32 y*)
by (*transfer*, *simp add*: *word-le-def*)

lemma *urel32-plus*: **assumes** *urel32 x y urel32 x' y'*
shows *urel32* (*plus-p32 pp x x'*) (*plus-p p y y'*)
proof −
let *?x* = *int-of-uint32 x*
let *?x'* = *int-of-uint32 x'*
let *?p* = *int-of-uint32 pp*

from *assms int-of-uint32-ge-0* **have** *id*: $y = ?x$ $y' = ?x'$
and *rel*: $0 \leq ?x$ $?x < p$
$0 \leq ?x'$ $?x' \leq p$ **unfolding** *urel32-def* **by** *auto*
have *le*: $(pp \leq x + x') = (?p \leq ?x + ?x')$ **unfolding** *le-int-of-uint32*
using *rel small* **by** (*auto simp*: *uint32-simps*)
show *?thesis*
proof (*cases* $?p \leq ?x + ?x'$)
case *True*
hence *True*: $(?p \leq ?x + ?x') = True$ **by** *simp*
show *?thesis* **unfolding** *id*
using *small rel* **unfolding** *plus-p32-def plus-p-def Let-def urel32-def*
unfolding *ppp le True if-True*
using *True* **by** (*auto simp*: *uint32-simps*)
next
case *False*
hence *False*: $(?p \leq ?x + ?x') = False$ **by** *simp*
show *?thesis* **unfolding** *id*
using *small rel* **unfolding** *plus-p32-def plus-p-def Let-def urel32-def*
unfolding *ppp le False if-False*
using *False* **by** (*auto simp*: *uint32-simps*)
qed
qed

lemma *urel32-minus*: **assumes** *urel32 x y urel32 x' y'*
shows *urel32* (*minus-p32 pp x x'*) (*minus-p p y y'*)
proof −
let $?x = int\text{-}of\text{-}uint32\ x$
let $?x' = int\text{-}of\text{-}uint32\ x'$
from *assms int-of-uint32-ge-0* **have** *id*: $y = ?x$ $y' = ?x'$
and *rel*: $0 \leq ?x$ $?x < p$
$0 \leq ?x'$ $?x' \leq p$ **unfolding** *urel32-def* **by** *auto*
have *le*: $(x' \leq x) = (?x' \leq ?x)$ **unfolding** *le-int-of-uint32*
using *rel small* **by** (*auto simp*: *uint32-simps*)
show *?thesis*
proof (*cases* $?x' \leq ?x$)
case *True*
hence *True*: $(?x' \leq ?x) = True$ **by** *simp*
show *?thesis* **unfolding** *id*
using *small rel* **unfolding** *minus-p32-def minus-p-def Let-def urel32-def*
unfolding *ppp le True if-True*
using *True* **by** (*auto simp*: *uint32-simps*)
next
case *False*
hence *False*: $(?x' \leq ?x) = False$ **by** *simp*
show *?thesis* **unfolding** *id*
using *small rel* **unfolding** *minus-p32-def minus-p-def Let-def urel32-def*
unfolding *ppp le False if-False*
using *False* **by** (*auto simp*: *uint32-simps*)
qed

```
qed

lemma urel32-uminus: assumes urel32 x y
  shows urel32 (uminus-p32 pp x) (uminus-p p y)
proof –
  let ?x = int-of-uint32 x
  from assms int-of-uint32-ge-0 have id: y = ?x
    and rel: 0 ≤ ?x ?x < p
      unfolding urel32-def by auto
  have le: (x = 0) = (?x = 0) unfolding int-of-uint32-0-iff
    using rel small by (auto simp: uint32-simps)
  show ?thesis
  proof (cases ?x = 0)
    case True
    hence True: (?x = 0) = True by simp
    show ?thesis unfolding id
      using small rel unfolding uminus-p32-def uminus-p-def Let-def urel32-def
      unfolding ppp le True if-True
      using True by (auto simp: uint32-simps)
  next
    case False
    hence False: (?x = 0) = False by simp
    show ?thesis unfolding id
      using small rel unfolding uminus-p32-def uminus-p-def Let-def urel32-def
      unfolding ppp le False if-False
      using False by (auto simp: uint32-simps)
  qed
qed

lemma urel32-mult: assumes urel32 x y urel32 x' y'
  shows urel32 (mult-p32 pp x x') (mult-p p y y')
proof –
  let ?x = int-of-uint32 x
  let ?x' = int-of-uint32 x'
  from assms int-of-uint32-ge-0 have id: y = ?x y' = ?x'
    and rel: 0 ≤ ?x ?x < p
      0 ≤ ?x' ?x' < p unfolding urel32-def by auto
  from rel have ?x * ?x' < p * p by (metis mult-strict-mono')
  also have ... ≤ 65536 * 65536
    by (rule mult-mono, insert p2 small, auto)
  finally have le: ?x * ?x' < 4294967296 by simp
  show ?thesis unfolding id
      using small rel unfolding mult-p32-def mult-p-def Let-def urel32-def
      unfolding ppp
    by (auto simp: uint32-simps, unfold int-of-uint32-mod int-of-uint32-mult,
        subst mod-pos-pos-trivial[of - 4294967296], insert le, auto)
qed

lemma urel32-eq: assumes urel32 x y urel32 x' y'
```

shows $(x = x') = (y = y')$
proof −
let *?x* = *int-of-uint32 x*
let *?x′* = *int-of-uint32 x′*
from *assms int-of-uint32-ge-0* **have** *id*: *y* = *?x y′* = *?x′*
unfolding *urel32-def* **by** *auto*
show *?thesis* **unfolding** *id* **by** (*transfer*, *transfer*) *rule*
qed

lemma *urel32-normalize*:
assumes *x*: *urel32 x y*
shows *urel32* (*if x* = *0 then 0 else 1*) (*if y* = *0 then 0 else 1*)
unfolding *urel32-eq*[*OF x urel32-0*] **using** *urel32-0 urel32-1* **by** *auto*

lemma *urel32-mod*:
assumes *x*: *urel32 x x′* **and** *y*: *urel32 y y′*
shows *urel32* (*if y* = *0 then x else 0*) (*if y′* = *0 then x′ else 0*)
unfolding *urel32-eq*[*OF y urel32-0*] **using** *urel32-0 x* **by** *auto*

lemma *urel32-power*: *urel32 x x′* ⟹ *urel32 y* (*int y′*) ⟹ *urel32* (*power-p32 pp x y*) (*power-p p x′ y′*)
including *bit-operations-syntax* **proof** (*induct x′ y′ arbitrary*: *x y rule*: *power-p.induct*[*of - p*])
case (*1 x′ y′ x y*)
note *x* = *1*(*2*) **note** *y* = *1*(*3*)
show *?case*
proof (*cases y′* = *0*)
case *True*
hence *y*: *y* = *0* **using** *urel32-eq*[*OF y urel32-0*] **by** *auto*
show *?thesis* **unfolding** *y True* **by** (*simp add*: *power-p.simps urel32-1*)
next
case *False*
hence *id*: (*y* = *0*) = *False* (*y′* = *0*) = *False* **using** *urel32-eq*[*OF y urel32-0*] **by** *auto*
from *y* **have** ‹*int y′* = *int-of-uint32 y*› ‹*int y′* < *p*›
by (*simp-all add*: *urel32-def*)
obtain *d′ r′* **where** *dr′*: *Divides.divmod-nat y′ 2* = (*d′*,*r′*) **by** *force*
from *divmod-nat-def*[*of y′ 2*, *unfolded dr′*]
have *r′*: *r′* = *y′ mod 2* **and** *d′*: *d′* = *y′ div 2* **by** *auto*
have *urel32* (*y AND 1*) *r′*
using ‹*int y′* < *p*› *small*
apply (*simp add*: *urel32-def and-one-eq r′*)
apply (*auto simp add*: *ppp and-one-eq*)
apply (*simp add*: *of-nat-mod int-of-uint32.rep-eq modulo-uint32.rep-eq uint-mod* ‹*int y′* = *int-of-uint32 y*›)
done
from *urel32-eq*[*OF this urel32-0*]
have *rem*: (*y AND 1* = *0*) = (*r′* = *0*) **by** *simp*
have *div*: *urel32* (*drop-bit 1 y*) (*int d′*) **unfolding** *d′* **using** *y* **unfolding**

urel32-def **using** *small*
unfolding *ppp*
apply *transfer*
apply *transfer*
apply (*auto simp add*: *drop-bit-Suc take-bit-int-eq-self*)
done
note *IH* = *1*(*1*)[*OF False refl dr'*[*symmetric*] *urel32-mult*[*OF x x*] *div*]
show *?thesis* **unfolding** *power-p.simps*[*of - - y'*] *power-p32.simps*[*of - - y*] *dr' id if-False rem*
using *IH urel32-mult*[*OF IH x*] **by** (*auto simp*: *Let-def*)
qed
qed

lemma *urel32-inverse*: **assumes** *x*: *urel32 x x'*
shows *urel32* (*inverse-p32 pp x*) (*inverse-p p x'*)
proof −
have *p*: *urel32* (*pp* − *2*) (*int* (*nat* (*p* − *2*))) **using** *p2 small* **unfolding** *urel32-def* **unfolding** *ppp*
by (*simp add*: *int-of-uint32.rep-eq minus-uint32.rep-eq uint-sub-if'*)
show *?thesis*
unfolding *inverse-p32-def inverse-p-def urel32-eq*[*OF x urel32-0*] **using** *urel32-0 urel32-power*[*OF x p*]
by *auto*
qed

lemma *mod-ring-0-32*: *mod-ring-rel32 0 0*
using *urel32-0 mod-ring-0* **unfolding** *mod-ring-rel32-def* **by** *blast*

lemma *mod-ring-1-32*: *mod-ring-rel32 1 1*
using *urel32-1 mod-ring-1* **unfolding** *mod-ring-rel32-def* **by** *blast*

lemma *mod-ring-uminus32*: (*mod-ring-rel32* ===> *mod-ring-rel32*) (*uminus-p32 pp*) *uminus*
using *urel32-uminus mod-ring-uminus* **unfolding** *mod-ring-rel32-def rel-fun-def* **by** *blast*

lemma *mod-ring-plus32*: (*mod-ring-rel32* ===> *mod-ring-rel32* ===> *mod-ring-rel32*) (*plus-p32 pp*) (+)
using *urel32-plus mod-ring-plus* **unfolding** *mod-ring-rel32-def rel-fun-def* **by** *blast*

lemma *mod-ring-minus32*: (*mod-ring-rel32* ===> *mod-ring-rel32* ===> *mod-ring-rel32*) (*minus-p32 pp*) (−)
using *urel32-minus mod-ring-minus* **unfolding** *mod-ring-rel32-def rel-fun-def* **by** *blast*

lemma *mod-ring-mult32*: (*mod-ring-rel32* ===> *mod-ring-rel32* ===> *mod-ring-rel32*) (*mult-p32 pp*) ((∗))

using *urel32-mult mod-ring-mult* **unfolding** *mod-ring-rel32-def rel-fun-def* **by** *blast*

lemma *mod-ring-eq32*: $(mod\text{-}ring\text{-}rel32 \implies mod\text{-}ring\text{-}rel32 \implies (=))\ (=)\ (=)$
using *urel32-eq mod-ring-eq* **unfolding** *mod-ring-rel32-def rel-fun-def* **by** *blast*

lemma *urel32-inj*: $urel32\ x\ y \Longrightarrow urel32\ x\ z \Longrightarrow y = z$
using *urel32-eq*[*of x y x z*] **by** *auto*

lemma *urel32-inj'*: $urel32\ x\ z \Longrightarrow urel32\ y\ z \Longrightarrow x = y$
using *urel32-eq*[*of x z y z*] **by** *auto*

lemma *bi-unique-mod-ring-rel32*:
bi-unique mod-ring-rel32 left-unique mod-ring-rel32 right-unique mod-ring-rel32
using *bi-unique-mod-ring-rel urel32-inj'*
unfolding *mod-ring-rel32-def bi-unique-def left-unique-def right-unique-def*
by (*auto simp*: *urel32-def*)

lemma *right-total-mod-ring-rel32*: *right-total mod-ring-rel32*
unfolding *mod-ring-rel32-def right-total-def*
proof
fix *y* :: *'a mod-ring*
from *right-total-mod-ring-rel*[*unfolded right-total-def*, *rule-format*, *of y*]
obtain *z* **where** *zy*: *mod-ring-rel z y* **by** *auto*
hence *zp*: $0 \le z\ z < p$ **unfolding** *mod-ring-rel-def p* **using** *range-to-int-mod-ring*[**where** $'a = 'a$] **by** *auto*
hence *urel32* (*uint32-of-int z*) *z* **unfolding** *urel32-def* **using** *small* **unfolding** *ppp*
by (*auto simp*: *int-of-uint32-inv*)
with *zy* **show** $\exists\ x\ z.\ urel32\ x\ z \wedge mod\text{-}ring\text{-}rel\ z\ y$ **by** *blast*
qed

lemma *Domainp-mod-ring-rel32*: $Domainp\ mod\text{-}ring\text{-}rel32 = (\lambda x.\ 0 \le x \wedge x < pp)$
proof
fix *x*
show $Domainp\ mod\text{-}ring\text{-}rel32\ x = (0 \le x \wedge x < pp)$
unfolding *Domainp.simps*
unfolding *mod-ring-rel32-def*
proof
let $?i = int\text{-}of\text{-}uint32$
assume $*$: $0 \le x \wedge x < pp$
hence $0 \le ?i\ x \wedge ?i\ x < p$ **using** *small* **unfolding** *ppp*
by (*transfer*, *auto simp*: *word-less-def*)
hence $?i\ x \in \{0\ ..< p\}$ **by** *auto*
with *Domainp-mod-ring-rel*
have *Domainp mod-ring-rel* (*?i x*) **by** *auto*
from *this*[*unfolded Domainp.simps*]

obtain *b* **where** *b*: *mod-ring-rel* (*?i x*) *b* **by** *auto*
show $\exists$ *a b*. *x* = *a* $\wedge$ ($\exists$ *z*. *urel32 a z* $\wedge$ *mod-ring-rel z b*)
proof (*intro exI*, *rule conjI*[*OF refl*], *rule exI*, *rule conjI*[*OF - b*])
show *urel32 x* (*?i x*) **unfolding** *urel32-def* **using** *small* $*$ **unfolding** *ppp*
by (*transfer*, *auto simp*: *word-less-def*)
qed
next
assume $\exists$ *a b*. *x* = *a* $\wedge$ ($\exists$ *z*. *urel32 a z* $\wedge$ *mod-ring-rel z b*)
then obtain *b z* **where** *xz*: *urel32 x z* **and** *zb*: *mod-ring-rel z b* **by** *auto*
hence *Domainp mod-ring-rel z* **by** *auto*
with *Domainp-mod-ring-rel* **have** *0* $\le$ *z z* < *p* **by** *auto*
with *xz* **show** *0* $\le$ *x* $\wedge$ *x* < *pp* **unfolding** *urel32-def* **using** *small* **unfolding** *ppp*
by (*transfer*, *auto simp*: *word-less-def*)
qed
qed

lemma *ring-finite-field-ops32*: *ring-ops* (*finite-field-ops32 pp*) *mod-ring-rel32*
by (*unfold-locales*, *auto simp*:
finite-field-ops32-def
bi-unique-mod-ring-rel32
right-total-mod-ring-rel32
mod-ring-plus32
mod-ring-minus32
mod-ring-uminus32
mod-ring-mult32
mod-ring-eq32
mod-ring-0-32
mod-ring-1-32
Domainp-mod-ring-rel32)
end
end

context *prime-field*
begin
context fixes *pp* :: *uint32*
assumes $*$: *p* = *int-of-uint32 pp p* $\le$ *65535*
begin

lemma *mod-ring-normalize32*: (*mod-ring-rel32* ===> *mod-ring-rel32*) (λ*x*. *if x* = *0 then 0 else 1*) *normalize*
using *urel32-normalize*[*OF* $*$] *mod-ring-normalize* **unfolding** *mod-ring-rel32-def*[*OF* $*$] *rel-fun-def* **by** *blast*

lemma *mod-ring-mod32*: (*mod-ring-rel32* ===> *mod-ring-rel32* ===> *mod-ring-rel32*) (λ*x y*. *if y* = *0 then x else 0*) (*mod*)
using *urel32-mod*[*OF* $*$] *mod-ring-mod* **unfolding** *mod-ring-rel32-def*[*OF* $*$] *rel-fun-def* **by** *blast*

lemma *mod-ring-unit-factor32*: (*mod-ring-rel32* ===> *mod-ring-rel32*) ($\lambda x.\ x$) *unit-factor*
  using *mod-ring-unit-factor* **unfolding** *mod-ring-rel32-def*[*OF* ∗] *rel-fun-def* **by** *blast*

lemma *mod-ring-inverse32*: (*mod-ring-rel32* ===> *mod-ring-rel32*) (*inverse-p32* *pp*) *inverse*
  using *urel32-inverse*[*OF* ∗] *mod-ring-inverse* **unfolding** *mod-ring-rel32-def*[*OF* ∗] *rel-fun-def* **by** *blast*

lemma *mod-ring-divide32*: (*mod-ring-rel32* ===> *mod-ring-rel32* ===> *mod-ring-rel32*) (*divide-p32* *pp*) (/)
  using *mod-ring-inverse32* *mod-ring-mult32*[*OF* ∗]
  unfolding *divide-p32-def* *divide-mod-ring-def* *inverse-mod-ring-def*[*symmetric*]
    rel-fun-def **by** *blast*

lemma *finite-field-ops32*: *field-ops* (*finite-field-ops32* *pp*) *mod-ring-rel32*
  by (*unfold-locales*, *insert* *ring-finite-field-ops32*[*OF* ∗], *auto* *simp*:
  ring-ops-def
  finite-field-ops32-def
  mod-ring-divide32
  mod-ring-inverse32
  mod-ring-mod32
  mod-ring-normalize32)

end
end

context
  fixes *p* :: *uint64*
begin
definition *plus-p64* :: *uint64* ⇒ *uint64* ⇒ *uint64* **where**
  plus-p64 *x* *y* ≡ *let* *z* = *x* + *y* *in* *if* *z* ≥ *p* *then* *z* − *p* *else* *z*

definition *minus-p64* :: *uint64* ⇒ *uint64* ⇒ *uint64* **where**
  minus-p64 *x* *y* ≡ *if* *y* ≤ *x* *then* *x* − *y* *else* (*x* + *p*) − *y*

definition *uminus-p64* :: *uint64* ⇒ *uint64* **where**
  uminus-p64 *x* = (*if* *x* = *0* *then* *0* *else* *p* − *x*)

definition *mult-p64* :: *uint64* ⇒ *uint64* ⇒ *uint64* **where**
  mult-p64 *x* *y* = (*x* ∗ *y* *mod* *p*)

lemma *int-of-uint64-shift*: *int-of-uint64* (*drop-bit* *k* *n*) = (*int-of-uint64* *n*) *div* (*2* ^ *k*)
  apply *transfer*
  apply *transfer*
  apply (*simp* *add*: *take-bit-drop-bit* *min-def*)

apply (*simp add*: *drop-bit-eq-div*)
done

lemma *int-of-uint64-0-iff*: *int-of-uint64 n* = *0* ⟷ *n* = *0*
by (*transfer*, *rule uint-0-iff*)

lemma *int-of-uint64-0*: *int-of-uint64 0* = *0* **unfolding** *int-of-uint64-0-iff* **by** *simp*

lemma *int-of-uint64-ge-0*: *int-of-uint64 n* ≥ *0*
by (*transfer*, *auto*)

lemma *two-64*: *2* ^ *LENGTH*(*64*) = (*18446744073709551616* :: *int*) **by** *simp*

lemma *int-of-uint64-plus*: *int-of-uint64* (*x* + *y*) = (*int-of-uint64 x* + *int-of-uint64 y*) *mod 18446744073709551616*
by (*transfer*, *unfold uint-word-ariths two-64*, *rule refl*)

lemma *int-of-uint64-minus*: *int-of-uint64* (*x* − *y*) = (*int-of-uint64 x* − *int-of-uint64 y*) *mod 18446744073709551616*
by (*transfer*, *unfold uint-word-ariths two-64*, *rule refl*)

lemma *int-of-uint64-mult*: *int-of-uint64* (*x* ∗ *y*) = (*int-of-uint64 x* ∗ *int-of-uint64 y*) *mod 18446744073709551616*
by (*transfer*, *unfold uint-word-ariths two-64*, *rule refl*)

lemma *int-of-uint64-mod*: *int-of-uint64* (*x mod y*) = (*int-of-uint64 x mod int-of-uint64 y*)
by (*transfer*, *unfold uint-mod two-64*, *rule refl*)

lemma *int-of-uint64-inv*: *0* ≤ *x* ⟹ *x* < *18446744073709551616* ⟹ *int-of-uint64* (*uint64-of-int x*) = *x*
by *transfer* (*simp add*: *take-bit-int-eq-self unsigned-of-int*)

context
includes *bit-operations-syntax*
begin

function *power-p64* :: *uint64* ⇒ *uint64* ⇒ *uint64* **where**
power-p64 x n = (*if n* = *0 then 1 else*
let rec = *power-p64* (*mult-p64 x x*) (*drop-bit 1 n*) *in*
if n AND 1 = *0 then rec else mult-p64 rec x*)
by *pat-completeness auto*

termination
proof −
{
fix *n* :: *uint64*
assume *n* ≠ *0*
with *int-of-uint64-ge-0*[*of n*] *int-of-uint64-0-iff*[*of n*] **have** *int-of-uint64 n* > *0*

by *auto*
hence $0 < \textit{int-of-uint64 n int-of-uint64 n div 2} < \textit{int-of-uint64 n}$ **by** *auto*
} **note** $* = \textit{this}$
show *?thesis*
by (*relation measure* (λ (*x*,*n*). *nat* (*int-of-uint64 n*)), *auto simp*: *int-of-uint64-shift* $*$)
qed

end

In experiments with Berlekamp-factorization (where the prime *p* is usually small), it turned out that taking the below implementation of inverse via exponentiation is faster than the one based on the extended Euclidean algorithm.

definition *inverse-p64* :: *uint64* $\Rightarrow$ *uint64* **where**
inverse-p64 x = (*if x* = *0 then 0 else power-p64 x* (*p* − *2*))

definition *divide-p64* :: *uint64* $\Rightarrow$ *uint64* $\Rightarrow$ *uint64* **where**
divide-p64 x y = *mult-p64 x* (*inverse-p64 y*)

definition *finite-field-ops64* :: *uint64 arith-ops-record* **where**
finite-field-ops64 $\equiv$ *Arith-Ops-Record*
0
1
plus-p64
mult-p64
minus-p64
uminus-p64
divide-p64
inverse-p64
(λ *x y* . *if y* = *0 then x else 0*)
(λ *x* . *if x* = *0 then 0 else 1*)
(λ *x* . *x*)
uint64-of-int
int-of-uint64
(λ *x*. $0 \leq x \wedge x < p$)
end

lemma *shiftr-uint64-code* [*code-unfold*]: *drop-bit 1 x* = (*uint64-shiftr x 1*)
by (*simp add*: *uint64-shiftr-def*)

For soundness of the 64-bit implementation, we mainly prove that this implementation implements the int-based implementation of GF(p).

context *mod-ring-locale*
begin

context fixes *pp* :: *uint64*
assumes *ppp*: *p* = *int-of-uint64 pp*
and *small*: $p \leq 4294967295$

begin

lemmas *uint64-simps* =
  int-of-uint64-0
  int-of-uint64-plus
  int-of-uint64-minus
  int-of-uint64-mult

definition *urel64* :: *uint64* ⇒ *int* ⇒ *bool* **where** *urel64 x y* = (*y* = *int-of-uint64 x* ∧ *y* < *p*)

definition *mod-ring-rel64* :: *uint64* ⇒ *'a mod-ring* ⇒ *bool* **where**
  mod-ring-rel64 x y = (∃ *z*. *urel64 x z* ∧ *mod-ring-rel z y*)

lemma *urel64-0*: *urel64 0 0* **unfolding** *urel64-def* **using** *p2* **by** (*simp*, *transfer*, *simp*)

lemma *urel64-1*: *urel64 1 1* **unfolding** *urel64-def* **using** *p2* **by** (*simp*, *transfer*, *simp*)

lemma *le-int-of-uint64*: (*x* ≤ *y*) = (*int-of-uint64 x* ≤ *int-of-uint64 y*)
  by (*transfer*, *simp add*: *word-le-def*)

lemma *urel64-plus*: **assumes** *urel64 x y urel64 x′ y′*
  shows *urel64* (*plus-p64 pp x x′*) (*plus-p p y y′*)
proof −
  let *?x* = *int-of-uint64 x*
  let *?x′* = *int-of-uint64 x′*
  let *?p* = *int-of-uint64 pp*
  from *assms int-of-uint64-ge-0* **have** *id*: *y* = *?x y′* = *?x′*
    and *rel*: *0* ≤ *?x ?x* < *p*
      0 ≤ *?x′ ?x′* ≤ *p* **unfolding** *urel64-def* **by** *auto*
  have *le*: (*pp* ≤ *x* + *x′*) = (*?p* ≤ *?x* + *?x′*) **unfolding** *le-int-of-uint64*
    using *rel small* **by** (*auto simp*: *uint64-simps*)
  show *?thesis*
  proof (*cases ?p* ≤ *?x* + *?x′*)
    case *True*
    hence *True*: (*?p* ≤ *?x* + *?x′*) = *True* **by** *simp*
    show *?thesis* **unfolding** *id*
      using *small rel* **unfolding** *plus-p64-def plus-p-def Let-def urel64-def*
      unfolding *ppp le True if-True*
      using *True* **by** (*auto simp*: *uint64-simps*)
  next
    case *False*
    hence *False*: (*?p* ≤ *?x* + *?x′*) = *False* **by** *simp*
    show *?thesis* **unfolding** *id*
      using *small rel* **unfolding** *plus-p64-def plus-p-def Let-def urel64-def*
      unfolding *ppp le False if-False*

```
    using False by (auto simp: uint64-simps)
  qed
qed

lemma urel64-minus: assumes urel64 x y urel64 x' y'
  shows urel64 (minus-p64 pp x x') (minus-p p y y')
proof -
  let ?x = int-of-uint64 x
  let ?x' = int-of-uint64 x'
  from assms int-of-uint64-ge-0 have id: y = ?x y' = ?x'
    and rel: 0 ≤ ?x ?x < p
      0 ≤ ?x' ?x' ≤ p unfolding urel64-def by auto
  have le: (x' ≤ x) = (?x' ≤ ?x) unfolding le-int-of-uint64
    using rel small by (auto simp: uint64-simps)
  show ?thesis
  proof (cases ?x' ≤ ?x)
    case True
    hence True: (?x' ≤ ?x) = True by simp
    show ?thesis unfolding id
      using small rel unfolding minus-p64-def minus-p-def Let-def urel64-def
      unfolding ppp le True if-True
      using True by (auto simp: uint64-simps)
  next
    case False
    hence False: (?x' ≤ ?x) = False by simp
    show ?thesis unfolding id
      using small rel unfolding minus-p64-def minus-p-def Let-def urel64-def
      unfolding ppp le False if-False
      using False by (auto simp: uint64-simps)
  qed
qed

lemma urel64-uminus: assumes urel64 x y
  shows urel64 (uminus-p64 pp x) (uminus-p p y)
proof -
  let ?x = int-of-uint64 x
  from assms int-of-uint64-ge-0 have id: y = ?x
    and rel: 0 ≤ ?x ?x < p
      unfolding urel64-def by auto
  have le: (x = 0) = (?x = 0) unfolding int-of-uint64-0-iff
    using rel small by (auto simp: uint64-simps)
  show ?thesis
  proof (cases ?x = 0)
    case True
    hence True: (?x = 0) = True by simp
    show ?thesis unfolding id
      using small rel unfolding uminus-p64-def uminus-p-def Let-def urel64-def
      unfolding ppp le True if-True
      using True by (auto simp: uint64-simps)
```

next
case *False*
hence *False*: $(?x = 0) = False$ **by** *simp*
show *?thesis* **unfolding** *id*
using *small rel* **unfolding** *uminus-p64-def uminus-p-def Let-def urel64-def*
unfolding *ppp le False if-False*
using *False* **by** (*auto simp*: *uint64-simps*)
qed
qed

lemma *urel64-mult*: **assumes** *urel64 x y urel64 x′ y′*
shows *urel64* (*mult-p64 pp x x′*) (*mult-p p y y′*)
proof −
let $?x = int\text{-}of\text{-}uint64\ x$
let $?x' = int\text{-}of\text{-}uint64\ x'$
from *assms int-of-uint64-ge-0* **have** *id*: $y = ?x\ y' = ?x'$
and *rel*: $0 \leq ?x\ ?x < p$
$0 \leq ?x'\ ?x' < p$ **unfolding** *urel64-def* **by** *auto*
from *rel* **have** $?x * ?x' < p * p$ **by** (*metis mult-strict-mono′*)
also have $\ldots \leq 4294967296 * 4294967296$
by (*rule mult-mono*, *insert p2 small*, *auto*)
finally have *le*: $?x * ?x' < 18446744073709551616$ **by** *simp*
show *?thesis* **unfolding** *id*
using *small rel* **unfolding** *mult-p64-def mult-p-def Let-def urel64-def*
unfolding *ppp*
by (*auto simp*: *uint64-simps*, *unfold int-of-uint64-mod int-of-uint64-mult*,
subst mod-pos-pos-trivial[*of - 18446744073709551616*], *insert le*, *auto*)
qed

lemma *urel64-eq*: **assumes** *urel64 x y urel64 x′ y′*
shows $(x = x') = (y = y')$
proof −
let $?x = int\text{-}of\text{-}uint64\ x$
let $?x' = int\text{-}of\text{-}uint64\ x'$
from *assms int-of-uint64-ge-0* **have** *id*: $y = ?x\ y' = ?x'$
unfolding *urel64-def* **by** *auto*
show *?thesis* **unfolding** *id* **by** (*transfer*, *transfer*) *rule*
qed

lemma *urel64-normalize*:
assumes *x*: *urel64 x y*
shows *urel64* (*if x = 0 then 0 else 1*) (*if y = 0 then 0 else 1*)
unfolding *urel64-eq*[*OF x urel64-0*] **using** *urel64-0 urel64-1* **by** *auto*

lemma *urel64-mod*:
assumes *x*: *urel64 x x′* **and** *y*: *urel64 y y′*
shows *urel64* (*if y = 0 then x else 0*) (*if y′ = 0 then x′ else 0*)
unfolding *urel64-eq*[*OF y urel64-0*] **using** *urel64-0 x* **by** *auto*

lemma *urel64-power*: *urel64 x x′* $\Longrightarrow$ *urel64 y* (*int y′*) $\Longrightarrow$ *urel64* (*power-p64 pp x y*) (*power-p p x′ y′*)
including *bit-operations-syntax* **proof** (*induct x′ y′ arbitrary*: *x y rule*: *power-p.induct*[*of - p*])
  case (*1 x′ y′ x y*)
  note *x* = *1*(*2*) **note** *y* = *1*(*3*)
  show *?case*
  proof (*cases y′* = *0*)
    case *True*
    hence *y*: *y* = *0* **using** *urel64-eq*[*OF y urel64-0*] **by** *auto*
    show *?thesis* **unfolding** *y True* **by** (*simp add*: *power-p.simps urel64-1*)
  next
    case *False*
    hence *id*: (*y* = *0*) = *False* (*y′* = *0*) = *False* **using** *urel64-eq*[*OF y urel64-0*] **by** *auto*
    from *y* **have** ‹*int y′* = *int-of-uint64 y*› ‹*int y′* < *p*›
      by (*simp-all add*: *urel64-def*)
    obtain *d′ r′* **where** *dr′*: *Divides.divmod-nat y′ 2* = (*d′*,*r′*) **by** *force*
    from *divmod-nat-def*[*of y′ 2*, *unfolded dr′*]
    have *r′*: *r′* = *y′ mod 2* **and** *d′*: *d′* = *y′ div 2* **by** *auto*
    have *urel64* (*y AND 1*) *r′*
      using ‹*int y′* < *p*› *small*
      apply (*simp add*: *urel64-def and-one-eq r′*)
      apply (*auto simp add*: *ppp and-one-eq*)
      apply (*simp add*: *of-nat-mod int-of-uint64.rep-eq modulo-uint64.rep-eq uint-mod* ‹*int y′* = *int-of-uint64 y*›)
      done
    from *urel64-eq*[*OF this urel64-0*]
    have *rem*: (*y AND 1* = *0*) = (*r′* = *0*) **by** *simp*
    have *div*: *urel64* (*drop-bit 1 y*) (*int d′*) **unfolding** *d′* **using** *y* **unfolding** *urel64-def* **using** *small*
      unfolding *ppp*
      apply *transfer*
      apply *transfer*
      apply (*auto simp add*: *drop-bit-Suc take-bit-int-eq-self*)
      done
    note *IH* = *1*(*1*)[*OF False refl dr′*[*symmetric*] *urel64-mult*[*OF x x*] *div*]
    show *?thesis* **unfolding** *power-p.simps*[*of - - y′*] *power-p64.simps*[*of - - y*] *dr′ id if-False rem*
      using *IH urel64-mult*[*OF IH x*] **by** (*auto simp*: *Let-def*)
  qed
qed

lemma *urel64-inverse*: **assumes** *x*: *urel64 x x′*
  shows *urel64* (*inverse-p64 pp x*) (*inverse-p p x′*)
proof −
  have *p*: *urel64* (*pp* − *2*) (*int* (*nat* (*p* − *2*))) **using** *p2 small* **unfolding** *urel64-def* **unfolding** *ppp*

by (*simp add*: *int-of-uint64.rep-eq minus-uint64.rep-eq uint-sub-if'*)
show *?thesis*
unfolding *inverse-p64-def inverse-p-def urel64-eq*[*OF x urel64-0*] **using** *urel64-0 urel64-power*[*OF x p*]
by *auto*
qed

lemma *mod-ring-0-64*: *mod-ring-rel64 0 0*
using *urel64-0 mod-ring-0* **unfolding** *mod-ring-rel64-def* **by** *blast*

lemma *mod-ring-1-64*: *mod-ring-rel64 1 1*
using *urel64-1 mod-ring-1* **unfolding** *mod-ring-rel64-def* **by** *blast*

lemma *mod-ring-uminus64*: (*mod-ring-rel64* ===> *mod-ring-rel64*) (*uminus-p64 pp*) *uminus*
using *urel64-uminus mod-ring-uminus* **unfolding** *mod-ring-rel64-def rel-fun-def* **by** *blast*

lemma *mod-ring-plus64*: (*mod-ring-rel64* ===> *mod-ring-rel64* ===> *mod-ring-rel64*) (*plus-p64 pp*) (+)
using *urel64-plus mod-ring-plus* **unfolding** *mod-ring-rel64-def rel-fun-def* **by** *blast*

lemma *mod-ring-minus64*: (*mod-ring-rel64* ===> *mod-ring-rel64* ===> *mod-ring-rel64*) (*minus-p64 pp*) (−)
using *urel64-minus mod-ring-minus* **unfolding** *mod-ring-rel64-def rel-fun-def* **by** *blast*

lemma *mod-ring-mult64*: (*mod-ring-rel64* ===> *mod-ring-rel64* ===> *mod-ring-rel64*) (*mult-p64 pp*) ((∗))
using *urel64-mult mod-ring-mult* **unfolding** *mod-ring-rel64-def rel-fun-def* **by** *blast*

lemma *mod-ring-eq64*: (*mod-ring-rel64* ===> *mod-ring-rel64* ===> (=)) (=) (=)
using *urel64-eq mod-ring-eq* **unfolding** *mod-ring-rel64-def rel-fun-def* **by** *blast*

lemma *urel64-inj*: *urel64 x y* $\Longrightarrow$ *urel64 x z* $\Longrightarrow$ *y* = *z*
using *urel64-eq*[*of x y x z*] **by** *auto*

lemma *urel64-inj'*: *urel64 x z* $\Longrightarrow$ *urel64 y z* $\Longrightarrow$ *x* = *y*
using *urel64-eq*[*of x z y z*] **by** *auto*

lemma *bi-unique-mod-ring-rel64*:
bi-unique mod-ring-rel64 left-unique mod-ring-rel64 right-unique mod-ring-rel64
using *bi-unique-mod-ring-rel urel64-inj'*
unfolding *mod-ring-rel64-def bi-unique-def left-unique-def right-unique-def*
by (*auto simp*: *urel64-def*)

lemma *right-total-mod-ring-rel64*: *right-total mod-ring-rel64*
unfolding *mod-ring-rel64-def right-total-def*
proof
fix *y* :: *'a mod-ring*
from *right-total-mod-ring-rel*[*unfolded right-total-def*, *rule-format*, *of y*]
obtain *z* **where** *zy*: *mod-ring-rel z y* **by** *auto*
hence *zp*: $0 \leq z$ $z < p$ **unfolding** *mod-ring-rel-def p* **using** *range-to-int-mod-ring*[**where** $'a = 'a$] **by** *auto*
hence *urel64* (*uint64-of-int z*) *z* **unfolding** *urel64-def* **using** *small* **unfolding** *ppp*
by (*auto simp*: *int-of-uint64-inv*)
with *zy* **show** $\exists$ *x z*. *urel64 x z* $\wedge$ *mod-ring-rel z y* **by** *blast*
qed

lemma *Domainp-mod-ring-rel64*: *Domainp mod-ring-rel64* = (λx. $0 \leq x \wedge x < pp$)
proof
fix *x*
show *Domainp mod-ring-rel64* $x = (0 \leq x \wedge x < pp)$
unfolding *Domainp.simps*
unfolding *mod-ring-rel64-def*
proof
let *?i* = *int-of-uint64*
assume $*$: $0 \leq x \wedge x < pp$
hence $0 \leq$ *?i x* $\wedge$ *?i x* $< p$ **using** *small* **unfolding** *ppp*
by (*transfer*, *auto simp*: *word-less-def*)
hence *?i x* $\in \{0 ..< p\}$ **by** *auto*
with *Domainp-mod-ring-rel*
have *Domainp mod-ring-rel* (*?i x*) **by** *auto*
from *this*[*unfolded Domainp.simps*]
obtain *b* **where** *b*: *mod-ring-rel* (*?i x*) *b* **by** *auto*
show $\exists$ *a b*. $x = a \wedge (\exists z$. *urel64 a z* $\wedge$ *mod-ring-rel z b*)
proof (*intro exI*, *rule conjI*[*OF refl*], *rule exI*, *rule conjI*[*OF - b*])
show *urel64 x* (*?i x*) **unfolding** *urel64-def* **using** *small* $*$ **unfolding** *ppp*
by (*transfer*, *auto simp*: *word-less-def*)
qed
next
assume $\exists$ *a b*. $x = a \wedge (\exists z$. *urel64 a z* $\wedge$ *mod-ring-rel z b*)
then obtain *b z* **where** *xz*: *urel64 x z* **and** *zb*: *mod-ring-rel z b* **by** *auto*
hence *Domainp mod-ring-rel z* **by** *auto*
with *Domainp-mod-ring-rel* **have** $0 \leq z$ $z < p$ **by** *auto*
with *xz* **show** $0 \leq x \wedge x < pp$ **unfolding** *urel64-def* **using** *small* **unfolding** *ppp*
by (*transfer*, *auto simp*: *word-less-def*)
qed
qed

lemma *ring-finite-field-ops64*: *ring-ops* (*finite-field-ops64 pp*) *mod-ring-rel64*
by (*unfold-locales*, *auto simp*:

finite-field-ops64-def
bi-unique-mod-ring-rel64
right-total-mod-ring-rel64
mod-ring-plus64
mod-ring-minus64
mod-ring-uminus64
mod-ring-mult64
mod-ring-eq64
mod-ring-0-64
mod-ring-1-64
Domainp-mod-ring-rel64)
end
end

context *prime-field*
begin
context fixes *pp* :: *uint64*
assumes ∗: *p* = *int-of-uint64 pp p* ≤ *4294967295*
begin

lemma *mod-ring-normalize64*: (*mod-ring-rel64* ===> *mod-ring-rel64*) (λ*x*. *if x* = *0 then 0 else 1*) *normalize*
using *urel64-normalize*[*OF* ∗] *mod-ring-normalize* **unfolding** *mod-ring-rel64-def*[*OF* ∗] *rel-fun-def* **by** *blast*

lemma *mod-ring-mod64*: (*mod-ring-rel64* ===> *mod-ring-rel64* ===> *mod-ring-rel64*) (λ*x y*. *if y* = *0 then x else 0*) (*mod*)
using *urel64-mod*[*OF* ∗] *mod-ring-mod* **unfolding** *mod-ring-rel64-def*[*OF* ∗] *rel-fun-def* **by** *blast*

lemma *mod-ring-unit-factor64*: (*mod-ring-rel64* ===> *mod-ring-rel64*) (λ*x*. *x*) *unit-factor*
using *mod-ring-unit-factor* **unfolding** *mod-ring-rel64-def*[*OF* ∗] *rel-fun-def* **by** *blast*

lemma *mod-ring-inverse64*: (*mod-ring-rel64* ===> *mod-ring-rel64*) (*inverse-p64 pp*) *inverse*
using *urel64-inverse*[*OF* ∗] *mod-ring-inverse* **unfolding** *mod-ring-rel64-def*[*OF* ∗] *rel-fun-def* **by** *blast*

lemma *mod-ring-divide64*: (*mod-ring-rel64* ===> *mod-ring-rel64* ===> *mod-ring-rel64*) (*divide-p64 pp*) (/)
using *mod-ring-inverse64 mod-ring-mult64*[*OF* ∗]
unfolding *divide-p64-def divide-mod-ring-def inverse-mod-ring-def*[*symmetric*]
rel-fun-def **by** *blast*

lemma *finite-field-ops64*: *field-ops* (*finite-field-ops64 pp*) *mod-ring-rel64*
by (*unfold-locales*, *insert ring-finite-field-ops64*[*OF* ∗], *auto simp*:
ring-ops-def

finite-field-ops64-def
mod-ring-divide64
mod-ring-inverse64
mod-ring-mod64
mod-ring-normalize64)
end
end

context
fixes p :: *integer*
begin
definition *plus-p-integer* :: *integer* ⇒ *integer* ⇒ *integer* **where**
plus-p-integer x y ≡ *let z* = *x* + *y in if z* ≥ *p then z* − *p else z*

definition *minus-p-integer* :: *integer* ⇒ *integer* ⇒ *integer* **where**
minus-p-integer x y ≡ *if y* ≤ *x then x* − *y else* (*x* + *p*) − *y*

definition *uminus-p-integer* :: *integer* ⇒ *integer* **where**
uminus-p-integer x = (*if x* = *0 then 0 else p* − *x*)

definition *mult-p-integer* :: *integer* ⇒ *integer* ⇒ *integer* **where**
mult-p-integer x y = (*x* ∗ *y mod p*)

lemma *int-of-integer-0-iff*: *int-of-integer n* = *0* ⟷ *n* = *0*
using *integer-eqI* **by** *auto*

lemma *int-of-integer-0*: *int-of-integer 0* = *0* **unfolding** *int-of-integer-0-iff* **by** *simp*

lemma *int-of-integer-plus*: *int-of-integer* (*x* + *y*) = (*int-of-integer x* + *int-of-integer y*)
by *simp*

lemma *int-of-integer-minus*: *int-of-integer* (*x* − *y*) = (*int-of-integer x* − *int-of-integer y*)
by *simp*

lemma *int-of-integer-mult*: *int-of-integer* (*x* ∗ *y*) = (*int-of-integer x* ∗ *int-of-integer y*)
by *simp*

lemma *int-of-integer-mod*: *int-of-integer* (*x mod y*) = (*int-of-integer x mod int-of-integer y*)
by *simp*

lemma *int-of-integer-inv*: *int-of-integer* (*integer-of-int x*) = *x* **by** *simp*

lemma *int-of-integer-shift*: *int-of-integer* (*drop-bit k n*) = (*int-of-integer n*) *div* (*2*

^ *k*)
  by *transfer* (*simp add*: *int-of-integer-pow shiftr-integer-conv-div-pow2*)

context
  includes *bit-operations-syntax*
begin

function *power-p-integer* :: *integer* ⇒ *integer* ⇒ *integer* **where**
  power-p-integer x n = (*if n* ≤ *0 then 1 else*
    let rec = *power-p-integer* (*mult-p-integer x x*) (*drop-bit 1 n*) *in*
    if n AND 1 = *0 then rec else mult-p-integer rec x*)
  by *pat-completeness auto*

termination
proof −
  {
    fix *n* :: *integer*
    assume ¬ (*n* ≤ *0*)
    hence *n* > *0* **by** *auto*
    hence *int-of-integer n* > *0*
      by (*simp add*: *less-integer.rep-eq*)
    hence *0* < *int-of-integer n int-of-integer n div 2* < *int-of-integer n* **by** *auto*
  } **note** ∗ = *this*
  show *?thesis*
   by (*relation measure* (λ (*x*,*n*). *nat* (*int-of-integer n*)), *auto simp*: ∗ *int-of-integer-shift*)

qed

end

In experiments with Berlekamp-factorization (where the prime p is usually small), it turned out that taking the below implementation of inverse via exponentiation is faster than the one based on the extended Euclidean algorithm.

definition *inverse-p-integer* :: *integer* ⇒ *integer* **where**
  inverse-p-integer x = (*if x* = *0 then 0 else power-p-integer x* (*p* − *2*))

definition *divide-p-integer* :: *integer* ⇒ *integer* ⇒ *integer* **where**
  divide-p-integer x y = *mult-p-integer x* (*inverse-p-integer y*)

definition *finite-field-ops-integer* :: *integer arith-ops-record* **where**
  finite-field-ops-integer ≡ *Arith-Ops-Record*
      0
      1
      plus-p-integer
      mult-p-integer
      minus-p-integer
      uminus-p-integer
      divide-p-integer

inverse-p-integer
(λ *x y* . *if y* = *0 then x else 0*)
(λ *x* . *if x* = *0 then 0 else 1*)
(λ *x* . *x*)
integer-of-int
int-of-integer
(λ *x*. *0* $\leq$ *x* $\wedge$ *x* < *p*)
end

lemma *shiftr-integer-code* [*code-unfold*]: *drop-bit 1 x* = (*integer-shiftr x 1*)
unfolding *shiftr-integer-code* **using** *integer-of-nat-1* **by** *auto*

For soundness of the integer implementation, we mainly prove that this implementation implements the int-based implementation of GF(p).

context *mod-ring-locale*
begin

context fixes *pp* :: *integer*
assumes *ppp*: *p* = *int-of-integer pp*
begin

lemmas *integer-simps* =
int-of-integer-0
int-of-integer-plus
int-of-integer-minus
int-of-integer-mult

definition *urel-integer* :: *integer* $\Rightarrow$ *int* $\Rightarrow$ *bool* **where** *urel-integer x y* = (*y* = *int-of-integer x* $\wedge$ *y* $\geq$ *0* $\wedge$ *y* < *p*)

definition *mod-ring-rel-integer* :: *integer* $\Rightarrow$ *'a mod-ring* $\Rightarrow$ *bool* **where**
mod-ring-rel-integer x y = ($\exists$ *z*. *urel-integer x z* $\wedge$ *mod-ring-rel z y*)

lemma *urel-integer-0*: *urel-integer 0 0* **unfolding** *urel-integer-def* **using** *p2* **by** *simp*

lemma *urel-integer-1*: *urel-integer 1 1* **unfolding** *urel-integer-def* **using** *p2* **by** *simp*

lemma *le-int-of-integer*: (*x* $\leq$ *y*) = (*int-of-integer x* $\leq$ *int-of-integer y*)
by (*rule less-eq-integer.rep-eq*)

lemma *urel-integer-plus*: **assumes** *urel-integer x y urel-integer x' y'*
shows *urel-integer* (*plus-p-integer pp x x'*) (*plus-p p y y'*)
proof −
let *?x* = *int-of-integer x*
let *?x'* = *int-of-integer x'*
let *?p* = *int-of-integer pp*

from *assms* **have** *id*: $y = ?x\ y' = ?x'$
  and *rel*: $0 \leq ?x\ ?x < p$
    $0 \leq ?x'\ ?x' \leq p$ **unfolding** *urel-integer-def* **by** *auto*
have *le*: $(pp \leq x + x') = (?p \leq ?x + ?x')$ **unfolding** *le-int-of-integer*
  using *rel* **by** *auto*
show *?thesis*
proof (*cases* $?p \leq ?x + ?x'$)
  case *True*
  hence *True*: $(?p \leq ?x + ?x') = True$ **by** *simp*
  show *?thesis* **unfolding** *id*
    using *rel* **unfolding** *plus-p-integer-def plus-p-def Let-def urel-integer-def*
    unfolding *ppp le True if-True*
    using *True* **by** *auto*
next
  case *False*
  hence *False*: $(?p \leq ?x + ?x') = False$ **by** *simp*
  show *?thesis* **unfolding** *id*
    using *rel* **unfolding** *plus-p-integer-def plus-p-def Let-def urel-integer-def*
    unfolding *ppp le False if-False*
    using *False* **by** *auto*
qed
qed

lemma *urel-integer-minus*: **assumes** *urel-integer x y urel-integer x' y'*
  shows *urel-integer* (*minus-p-integer pp x x'*) (*minus-p p y y'*)
proof −
  let $?x = int\text{-}of\text{-}integer\ x$
  let $?x' = int\text{-}of\text{-}integer\ x'$
  from *assms* **have** *id*: $y = ?x\ y' = ?x'$
    and *rel*: $0 \leq ?x\ ?x < p$
      $0 \leq ?x'\ ?x' \leq p$ **unfolding** *urel-integer-def* **by** *auto*
  have *le*: $(x' \leq x) = (?x' \leq ?x)$ **unfolding** *le-int-of-integer*
    using *rel* **by** *auto*
  show *?thesis*
  proof (*cases* $?x' \leq ?x$)
    case *True*
    hence *True*: $(?x' \leq ?x) = True$ **by** *simp*
    show *?thesis* **unfolding** *id*
      using *rel* **unfolding** *minus-p-integer-def minus-p-def Let-def urel-integer-def*
      unfolding *ppp le True if-True*
      using *True* **by** *auto*
  next
    case *False*
    hence *False*: $(?x' \leq ?x) = False$ **by** *simp*
    show *?thesis* **unfolding** *id*
      using *rel* **unfolding** *minus-p-integer-def minus-p-def Let-def urel-integer-def*
      unfolding *ppp le False if-False*
      using *False* **by** *auto*
  qed

qed

lemma *urel-integer-uminus*: **assumes** *urel-integer x y*
  shows *urel-integer* (*uminus-p-integer pp x*) (*uminus-p p y*)
proof −
  let *?x* = *int-of-integer x*
  from *assms* **have** *id*: $y = ?x$
    and *rel*: $0 \leq ?x$ $?x < p$
      unfolding *urel-integer-def* **by** *auto*
  have *le*: $(x = 0) = (?x = 0)$ **unfolding** *int-of-integer-0-iff*
    using *rel* **by** *auto*
  show *?thesis*
  proof (*cases* $?x = 0$)
    case *True*
    hence *True*: $(?x = 0) = True$ **by** *simp*
    show *?thesis* **unfolding** *id*
     using *rel* **unfolding** *uminus-p-integer-def uminus-p-def Let-def urel-integer-def*

      unfolding *ppp le True if-True*
      using *True* **by** *auto*
  next
    case *False*
    hence *False*: $(?x = 0) = False$ **by** *simp*
    show *?thesis* **unfolding** *id*
     using *rel* **unfolding** *uminus-p-integer-def uminus-p-def Let-def urel-integer-def*

      unfolding *ppp le False if-False*
      using *False* **by** *auto*
  qed
qed

lemma *pp-pos*: *int-of-integer pp* > *0*
  using *ppp nontriv*[**where** $'a = 'a$] **unfolding** *p*
  by (*simp add*: *less-integer.rep-eq*)

lemma *urel-integer-mult*: **assumes** *urel-integer x y urel-integer x′ y′*
  shows *urel-integer* (*mult-p-integer pp x x′*) (*mult-p p y y′*)
proof −
  let *?x* = *int-of-integer x*
  let *?x′* = *int-of-integer x′*
  from *assms* **have** *id*: $y = ?x$ $y' = ?x'$
    and *rel*: $0 \leq ?x$ $?x < p$
      $0 \leq ?x'$ $?x' < p$ **unfolding** *urel-integer-def* **by** *auto*
  from *rel*(*1*,*3*) **have** *xx*: $0 \leq ?x * ?x'$ **by** *simp*
  show *?thesis* **unfolding** *id*
    using *rel* **unfolding** *mult-p-integer-def mult-p-def Let-def urel-integer-def*
     unfolding *ppp mod-nonneg-pos-int*[*OF xx pp-pos*] **using** *xx pp-pos* **by** *simp*

qed

lemma *urel-integer-eq*: **assumes** *urel-integer x y urel-integer x′ y′*
  shows $(x = x') = (y = y')$
proof −
  let *?x* = *int-of-integer x*
  let *?x′* = *int-of-integer x′*
  from *assms* **have** *id*: *y* = *?x y′* = *?x′*
    unfolding *urel-integer-def* **by** *auto*
  show *?thesis* **unfolding** *id integer-eq-iff* **..**
qed

lemma *urel-integer-normalize*:
assumes *x*: *urel-integer x y*
shows *urel-integer* (*if x* = *0 then 0 else 1*) (*if y* = *0 then 0 else 1*)
 unfolding *urel-integer-eq*[*OF x urel-integer-0*] **using** *urel-integer-0 urel-integer-1*
by *auto*

lemma *urel-integer-mod*:
assumes *x*: *urel-integer x x′* **and** *y*: *urel-integer y y′*
shows *urel-integer* (*if y* = *0 then x else 0*) (*if y′* = *0 then x′ else 0*)
  unfolding *urel-integer-eq*[*OF y urel-integer-0*] **using** *urel-integer-0 x* **by** *auto*

lemma *urel-integer-power*: *urel-integer x x′* ⟹ *urel-integer y* (*int y′*) ⟹ *urel-integer* (*power-p-integer pp x y*) (*power-p p x′ y′*)
including *bit-operations-syntax* **proof** (*induct x′ y′ arbitrary*: *x y rule*: *power-p.induct*[*of - p*])
  case (*1 x′ y′ x y*)
  note *x* = *1*(*2*) **note** *y* = *1*(*3*)
  show *?case*
  proof (*cases y′* ≤ *0*)
    case *True*
    hence *y*: *y* = *0 y′* = *0* **using** *urel-integer-eq*[*OF y urel-integer-0*] **by** *auto*
    show *?thesis* **unfolding** *y True* **by** (*simp add*: *power-p.simps urel-integer-1*)
  next
    case *False*
    hence *id*: (*y* ≤ *0*) = *False* (*y′* = *0*) = *False* **using** *False y*
     by (*auto simp add*: *urel-integer-def not-le*) (*metis of-int-integer-of of-int-of-nat-eq of-nat-0-less-iff*)
    obtain *d′ r′* **where** *dr′*: *Divides.divmod-nat y′ 2* = (*d′*,*r′*) **by** *force*
    from *divmod-nat-def*[*of y′ 2*, *unfolded dr′*]
    have *r′*: *r′* = *y′ mod 2* **and** *d′*: *d′* = *y′ div 2* **by** *auto*
    have *aux*: ⋀ *y′*. *int* (*y′ mod 2*) = *int y′ mod 2* **by** *presburger*
   have *urel-integer* (*y AND 1*) *r′* **unfolding** *r′* **using** *y* **unfolding** *urel-integer-def*

      unfolding *ppp*
      apply (*auto simp add*: *and-one-eq*)
      apply (*simp add*: *of-nat-mod*)
      done
    from *urel-integer-eq*[*OF this urel-integer-0*]

have *rem*: (*y AND 1* = *0*) = (*r′* = *0*) **by** *simp*
have *div*: *urel-integer* (*drop-bit 1 y*) (*int d′*) **unfolding** *d′* **using** *y* **unfolding** *urel-integer-def*
unfolding *ppp shiftr-integer-conv-div-pow2* **by** *auto*
from *id* **have** *y′* ≠ *0* **by** *auto*
note *IH* = *1*(*1*)[*OF this refl dr′*[*symmetric*] *urel-integer-mult*[*OF x x*] *div*]
show *?thesis* **unfolding** *power-p.simps*[*of - - y′*] *power-p-integer.simps*[*of - - y*] *dr′ id if-False rem*
using *IH urel-integer-mult*[*OF IH x*] **by** (*auto simp*: *Let-def*)
qed
qed

lemma *urel-integer-inverse*: **assumes** *x*: *urel-integer x x′*
shows *urel-integer* (*inverse-p-integer pp x*) (*inverse-p p x′*)
proof −
have *p*: *urel-integer* (*pp* − *2*) (*int* (*nat* (*p* − *2*))) **using** *p2* **unfolding** *urel-integer-def* **unfolding** *ppp*
by *auto*
show *?thesis*
unfolding *inverse-p-integer-def inverse-p-def urel-integer-eq*[*OF x urel-integer-0*] **using** *urel-integer-0 urel-integer-power*[*OF x p*]
by *auto*
qed

lemma *mod-ring-0--integer*: *mod-ring-rel-integer 0 0*
using *urel-integer-0 mod-ring-0* **unfolding** *mod-ring-rel-integer-def* **by** *blast*

lemma *mod-ring-1--integer*: *mod-ring-rel-integer 1 1*
using *urel-integer-1 mod-ring-1* **unfolding** *mod-ring-rel-integer-def* **by** *blast*

lemma *mod-ring-uminus-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer*) (*uminus-p-integer pp*) *uminus*
using *urel-integer-uminus mod-ring-uminus* **unfolding** *mod-ring-rel-integer-def rel-fun-def* **by** *blast*

lemma *mod-ring-plus-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer* ===> *mod-ring-rel-integer*) (*plus-p-integer pp*) (+)
using *urel-integer-plus mod-ring-plus* **unfolding** *mod-ring-rel-integer-def rel-fun-def* **by** *blast*

lemma *mod-ring-minus-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer* ===> *mod-ring-rel-integer*) (*minus-p-integer pp*) (−)
using *urel-integer-minus mod-ring-minus* **unfolding** *mod-ring-rel-integer-def rel-fun-def* **by** *blast*

lemma *mod-ring-mult-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer* ===> *mod-ring-rel-integer*) (*mult-p-integer pp*) ((∗))
using *urel-integer-mult mod-ring-mult* **unfolding** *mod-ring-rel-integer-def rel-fun-def*

by *blast*

lemma *mod-ring-eq-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer* ===> (=)) (=) (=)
  using *urel-integer-eq mod-ring-eq* **unfolding** *mod-ring-rel-integer-def rel-fun-def*
by *blast*

lemma *urel-integer-inj*: *urel-integer x y* $\Longrightarrow$ *urel-integer x z* $\Longrightarrow$ $y = z$
  using *urel-integer-eq*[*of x y x z*] **by** *auto*

lemma *urel-integer-inj'*: *urel-integer x z* $\Longrightarrow$ *urel-integer y z* $\Longrightarrow$ $x = y$
  using *urel-integer-eq*[*of x z y z*] **by** *auto*

lemma *bi-unique-mod-ring-rel-integer*:
  bi-unique mod-ring-rel-integer left-unique mod-ring-rel-integer right-unique mod-ring-rel-integer
  using *bi-unique-mod-ring-rel urel-integer-inj'*
  unfolding *mod-ring-rel-integer-def bi-unique-def left-unique-def right-unique-def*
  by (*auto simp*: *urel-integer-def*)

lemma *right-total-mod-ring-rel-integer*: *right-total mod-ring-rel-integer*
  unfolding *mod-ring-rel-integer-def right-total-def*
proof
  fix *y* :: *'a mod-ring*
  from *right-total-mod-ring-rel*[*unfolded right-total-def*, *rule-format*, *of y*]
  obtain *z* **where** *zy*: *mod-ring-rel z y* **by** *auto*
  hence *zp*: $0 \le z\, z < p$ **unfolding** *mod-ring-rel-def p* **using** *range-to-int-mod-ring*[**where** *'a* = *'a*] **by** *auto*
  hence *urel-integer* (*integer-of-int z*) *z* **unfolding** *urel-integer-def* **unfolding** *ppp*

    by *auto*
  with *zy* **show** $\exists$ *x z*. *urel-integer x z* $\wedge$ *mod-ring-rel z y* **by** *blast*
qed

lemma *Domainp-mod-ring-rel-integer*: *Domainp mod-ring-rel-integer* = (λx. $0 \le x \wedge x < pp$)
proof
  fix *x*
  show *Domainp mod-ring-rel-integer x* = ($0 \le x \wedge x < pp$)
    unfolding *Domainp.simps*
    unfolding *mod-ring-rel-integer-def*
  proof
    let *?i* = *int-of-integer*
    assume $*$: $0 \le x \wedge x < pp$
    hence $0 \le$ *?i x* $\wedge$ *?i x* $< p$ **unfolding** *ppp*
      by (*simp add*: *le-int-of-integer less-integer.rep-eq*)
    hence *?i x* $\in \{0\ ..< p\}$ **by** *auto*
    with *Domainp-mod-ring-rel*
    have *Domainp mod-ring-rel* (*?i x*) **by** *auto*
    from *this*[*unfolded Domainp.simps*]

obtain *b* **where** *b*: *mod-ring-rel* (*?i x*) *b* **by** *auto*
show $\exists$ *a b*. $x = a \wedge (\exists z.$ *urel-integer a z* $\wedge$ *mod-ring-rel z b*)
proof (*intro exI*, *rule conjI*[*OF refl*], *rule exI*, *rule conjI*[*OF - b*])
show *urel-integer x* (*?i x*) **unfolding** *urel-integer-def* **using** $*$ **unfolding** *ppp*
by (*simp add*: *le-int-of-integer less-integer.rep-eq*)
qed
next
assume $\exists$ *a b*. $x = a \wedge (\exists z.$ *urel-integer a z* $\wedge$ *mod-ring-rel z b*)
then obtain *b z* **where** *xz*: *urel-integer x z* **and** *zb*: *mod-ring-rel z b* **by** *auto*
hence *Domainp mod-ring-rel z* **by** *auto*
with *Domainp-mod-ring-rel* **have** $0 \leq z$ $z < p$ **by** *auto*
with *xz* **show** $0 \leq x \wedge x < pp$ **unfolding** *urel-integer-def* **unfolding** *ppp*
by (*simp add*: *le-int-of-integer less-integer.rep-eq*)
qed
qed

lemma *ring-finite-field-ops-integer*: *ring-ops* (*finite-field-ops-integer pp*) *mod-ring-rel-integer*
by (*unfold-locales*, *auto simp*:
finite-field-ops-integer-def
bi-unique-mod-ring-rel-integer
right-total-mod-ring-rel-integer
mod-ring-plus-integer
mod-ring-minus-integer
mod-ring-uminus-integer
mod-ring-mult-integer
mod-ring-eq-integer
mod-ring-0--integer
mod-ring-1--integer
Domainp-mod-ring-rel-integer)
end
end

context *prime-field*
begin
context fixes *pp* :: *integer*
assumes $*$: $p =$ *int-of-integer pp*
begin

lemma *mod-ring-normalize-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer*) ($\lambda x.$ *if* $x = 0$ *then* 0 *else* 1) *normalize*
using *urel-integer-normalize*[*OF* $*$] *mod-ring-normalize* **unfolding** *mod-ring-rel-integer-def*[*OF* $*$] *rel-fun-def* **by** *blast*

lemma *mod-ring-mod-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer* ===> *mod-ring-rel-integer*) ($\lambda x\ y.$ *if* $y = 0$ *then* x *else* 0) (*mod*)
using *urel-integer-mod*[*OF* $*$] *mod-ring-mod* **unfolding** *mod-ring-rel-integer-def*[*OF* $*$] *rel-fun-def* **by** *blast*

lemma *mod-ring-unit-factor-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer*)

($\lambda x.\ x$) *unit-factor*
using *mod-ring-unit-factor* **unfolding** *mod-ring-rel-integer-def*[*OF* $*$] *rel-fun-def*
by *blast*

lemma *mod-ring-inverse-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer*) (*inverse-p-integer pp*) *inverse*
using *urel-integer-inverse*[*OF* $*$] *mod-ring-inverse* **unfolding** *mod-ring-rel-integer-def*[*OF* $*$] *rel-fun-def* **by** *blast*

lemma *mod-ring-divide-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer* ===> *mod-ring-rel-integer*) (*divide-p-integer pp*) (/)
using *mod-ring-inverse-integer mod-ring-mult-integer*[*OF* $*$]
unfolding *divide-p-integer-def divide-mod-ring-def inverse-mod-ring-def*[*symmetric*]
rel-fun-def **by** *blast*

lemma *finite-field-ops-integer*: *field-ops* (*finite-field-ops-integer pp*) *mod-ring-rel-integer*
by (*unfold-locales*, *insert ring-finite-field-ops-integer*[*OF* $*$], *auto simp*:
ring-ops-def
finite-field-ops-integer-def
mod-ring-divide-integer
mod-ring-inverse-integer
mod-ring-mod-integer
mod-ring-normalize-integer)
end
end

context *prime-field*
begin

thm
finite-field-ops64
finite-field-ops32
finite-field-ops-integer
finite-field-ops-int
end

context *mod-ring-locale*
begin

thm
ring-finite-field-ops64
ring-finite-field-ops32
ring-finite-field-ops-integer
ring-finite-field-ops-int
end

end

3.2 Matrix Operations in Fields

We use our record based description of a field to perform matrix operations.

theory *Matrix-Record-Based*
imports
  Jordan-Normal-Form.Gauss-Jordan-Elimination
  Jordan-Normal-Form.Gauss-Jordan-IArray-Impl
  Arithmetic-Record-Based
begin

definition *mat-rel* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ mat \Rightarrow 'b\ mat \Rightarrow bool$ **where**
  $mat\text{-}rel\ R\ A\ B \equiv dim\text{-}row\ A = dim\text{-}row\ B \wedge dim\text{-}col\ A = dim\text{-}col\ B \wedge$
    $(\forall\ i\ j.\ i < dim\text{-}row\ B \longrightarrow j < dim\text{-}col\ B \longrightarrow R\ (A\ \$\$\ (i,j))\ (B\ \$\$\ (i,j)))$

lemma *right-total-mat-rel*: $right\text{-}total\ R \Longrightarrow right\text{-}total\ (mat\text{-}rel\ R)$
  unfolding *right-total-def*
proof
  fix *B*
  assume $\forall\ y.\ \exists\ x.\ R\ x\ y$
  from *choice*[*OF this*] **obtain** *f* **where** *f*: $\bigwedge x.\ R\ (f\ x)\ x$ **by** *auto*
  show $\exists\ A.\ mat\text{-}rel\ R\ A\ B$
    by (*rule exI*[*of - map-mat f B*], *unfold mat-rel-def*, *auto simp*: *f*)
qed

lemma *left-unique-mat-rel*: $left\text{-}unique\ R \Longrightarrow left\text{-}unique\ (mat\text{-}rel\ R)$
  unfolding *left-unique-def mat-rel-def mat-eq-iff* **by** (*auto*, *blast*)

lemma *right-unique-mat-rel*: $right\text{-}unique\ R \Longrightarrow right\text{-}unique\ (mat\text{-}rel\ R)$
  unfolding *right-unique-def mat-rel-def mat-eq-iff* **by** (*auto*, *blast*)

lemma *bi-unique-mat-rel*: $bi\text{-}unique\ R \Longrightarrow bi\text{-}unique\ (mat\text{-}rel\ R)$
  using *left-unique-mat-rel*[*of R*] *right-unique-mat-rel*[*of R*]
  unfolding *bi-unique-def left-unique-def right-unique-def* **by** *blast*

lemma *mat-rel-eq*: $((R ===> R ===> (=)))\ (=)\ (=) \Longrightarrow$
  $((mat\text{-}rel\ R ===> mat\text{-}rel\ R ===> (=)))\ (=)\ (=)$
  unfolding *mat-rel-def rel-fun-def mat-eq-iff* **by** (*auto*, *blast+*)

definition *vec-rel* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ vec \Rightarrow 'b\ vec \Rightarrow bool$ **where**
  $vec\text{-}rel\ R\ A\ B \equiv dim\text{-}vec\ A = dim\text{-}vec\ B \wedge (\forall\ i.\ i < dim\text{-}vec\ B \longrightarrow R\ (A\ \$\ i)$
$(B\ \$\ i))$

lemma *right-total-vec-rel*: $right\text{-}total\ R \Longrightarrow right\text{-}total\ (vec\text{-}rel\ R)$
  unfolding *right-total-def*
proof
  fix *B*
  assume $\forall\ y.\ \exists\ x.\ R\ x\ y$
  from *choice*[*OF this*] **obtain** *f* **where** *f*: $\bigwedge x.\ R\ (f\ x)\ x$ **by** *auto*

show ∃ *A*. *vec-rel R A B*
by (*rule exI*[*of - map-vec f B*], *unfold vec-rel-def*, *auto simp*: *f*)
qed

lemma *left-unique-vec-rel*: *left-unique R* ⟹ *left-unique* (*vec-rel R*)
unfolding *left-unique-def vec-rel-def vec-eq-iff* **by** *auto*

lemma *right-unique-vec-rel*: *right-unique R* ⟹ *right-unique* (*vec-rel R*)
unfolding *right-unique-def vec-rel-def vec-eq-iff* **by** *auto*

lemma *bi-unique-vec-rel*: *bi-unique R* ⟹ *bi-unique* (*vec-rel R*)
using *left-unique-vec-rel*[*of R*] *right-unique-vec-rel*[*of R*]
unfolding *bi-unique-def left-unique-def right-unique-def* **by** *blast*

lemma *vec-rel-eq*: ((*R* ===> *R* ===> (=))) (=) (=) ⟹
((*vec-rel R* ===> *vec-rel R* ===> (=))) (=) (=)
unfolding *vec-rel-def rel-fun-def vec-eq-iff* **by** (*auto*, *blast*+)

lemma *multrow-transfer*[*transfer-rule*]: ((*R* ===> *R* ===> *R*) ===> (=) ===> *R*
===> *mat-rel R* ===> *mat-rel R*) *mat-multrow-gen mat-multrow-gen*
unfolding *mat-rel-def*[*abs-def*] *mat-multrow-gen-def*[*abs-def*]
by (*intro rel-funI conjI allI impI eq-matI*, *auto simp*: *rel-fun-def*)

lemma *swap-rows-transfer*: *mat-rel R A B* ⟹ *i* < *dim-row B* ⟹ *j* < *dim-row B* ⟹
mat-rel R (*mat-swaprows i j A*) (*mat-swaprows i j B*)
unfolding *mat-rel-def mat-swaprows-def*
by (*intro rel-funI conjI allI impI eq-matI*, *auto*)

lemma *pivot-positions-gen-transfer*: **assumes** [*transfer-rule*]: (*R* ===> *R* ===> (=)) (=) (=)
shows
(*R* ===> *mat-rel R* ===> (=)) *pivot-positions-gen pivot-positions-gen*
proof (*intro rel-funI*, *goal-cases*)
case (*1 ze ze′ A A′*)
note *trans*[*transfer-rule*] = *1*
from *1* **have** *dim*: *dim-row A* = *dim-row A′ dim-col A* = *dim-col A′* **unfolding** *mat-rel-def* **by** *auto*
obtain *i j* **where** *id*: *i* = *0 j* = *0* **and** *ij*: *i* ≤ *dim-row A′ j* ≤ *dim-col A′* **by** *auto*
have *pivot-positions-main-gen ze A* (*dim-row A*) (*dim-col A*) *i j* =
pivot-positions-main-gen ze′ A′ (*dim-row A′*) (*dim-col A′*) *i j*
using *ij*
proof (*induct i j rule*: *pivot-positions-main-gen.induct*[*of dim-row A′ dim-col A′ A′ ze′*])
case (*1 i j*)
note *simps*[*simp*] = *pivot-positions-main-gen.simps*[*of - - - - i j*]

```
    show ?case
    proof (cases i < dim-row A′ ∧ j < dim-col A′)
      case False
      with dim show ?thesis by auto
    next
      case True
      hence ij: i < dim-row A′ j < dim-col A′ and j: Suc j ≤ dim-col A′ by auto
      note IH = 1(1−2)[OF ij - - j]
      from ij True trans have [transfer-rule]:R (A $$ (i,j)) (A′ $$ (i,j))
        unfolding mat-rel-def by auto
      have eq: (A $$ (i,j) = ze) = (A′ $$ (i,j) = ze′) by transfer-prover
      show ?thesis
      proof (cases A′ $$ (i,j) = ze′)
        case True
        from ij have i ≤ dim-row A′ by auto
        note IH = IH(1)[OF True this]
        thus ?thesis using True ij dim eq by simp
      next
        case False
        from ij have Suc i ≤ dim-row A′ by auto
        note IH = IH(2)[OF False this]
        thus ?thesis using False ij dim eq by simp
      qed
    qed
  qed
  thus pivot-positions-gen ze A = pivot-positions-gen ze′ A′
    unfolding pivot-positions-gen-def id .
qed

lemma set-pivot-positions-main-gen:
  set (pivot-positions-main-gen ze A nr nc i j) ⊆ {0 ..< nr} × {0 ..< nc}
proof (induct i j rule: pivot-positions-main-gen.induct[of nr nc A ze])
  case (1 i j)
  note [simp] = pivot-positions-main-gen.simps[of - - - - i j]
  from 1 show ?case
    by (cases i < nr ∧ j < nc, auto)
qed

lemma find-base-vectors-transfer: assumes [transfer-rule]: (R ===> R ===>
(=)) (=) (=)
  shows ((R ===> R) ===> R ===> R ===> mat-rel R
  ===> list-all2 (vec-rel R)) find-base-vectors-gen find-base-vectors-gen
proof (intro rel-funI, goal-cases)
  case (1 um um′ ze ze′ on on′ A A′)
  note trans[transfer-rule] = 1 pivot-positions-gen-transfer[OF assms]
 from 1(4) have dim: dim-row A = dim-row A′ dim-col A = dim-col A′ unfolding
mat-rel-def by auto
  have id: pivot-positions-gen ze A = pivot-positions-gen ze′ A′ by transfer-prover
  obtain xs where xs: map snd (pivot-positions-gen ze′ A′) = xs by auto
```

obtain *ys* **where** *ys*: [*j*←[*0*..<*dim-col A'*] . *j* ∉ *set xs*] = *ys* **by** *auto*
show *list-all2* (*vec-rel R*) (*find-base-vectors-gen um ze on A*)
(*find-base-vectors-gen um' ze' on' A'*)
unfolding *find-base-vectors-gen-def Let-def id xs list-all2-conv-all-nth length-map ys dim*
proof (*intro conjI*[*OF refl*] *allI impI*)
fix *i*
assume *i*: *i* < *length ys*
define *y* **where** *y* = *ys* ! *i*
from *i* **have** *y*: *y* < *dim-col A'* **unfolding** *y-def ys*[*symmetric*] **using** *nth-mem* **by** *fastforce*
let *?map* = *map-of* (*map prod.swap* (*pivot-positions-gen ze' A'*))
{
fix *i*
assume *i*: *i* < *dim-col A'*
and *neq*: *i* ≠ *y*
have *R* (*case ?map i of None* ⇒ *ze* | *Some j* ⇒ *um* (*A* $$ (*j*, *y*)))
(*case ?map i of None* ⇒ *ze'* | *Some j* ⇒ *um'* (*A'* $$ (*j*, *y*)))
proof (*cases ?map i*)
case *None*
with *trans*(*2*) **show** *?thesis* **by** *auto*
next
case (*Some j*)
from *map-of-SomeD*[*OF this*] **have** (*j*,*i*) ∈ *set* (*pivot-positions-gen ze' A'*) **by** *auto*
from *subsetD*[*OF set-pivot-positions-main-gen this*[*unfolded pivot-positions-gen-def*]]
have *j*: *j* < *dim-row A'* **by** *auto*
with *trans*(*4*) *y* **have** [*transfer-rule*]: *R* (*A* $$ (*j*,*y*)) (*A'* $$ (*j*,*y*)) **unfolding** *mat-rel-def* **by** *auto*
show *?thesis* **unfolding** *Some* **by** (*simp*, *transfer-prover*)
qed
} **note** *main* = *this*
show *vec-rel R* (*map* (*non-pivot-base-gen um ze on A* (*pivot-positions-gen ze' A'*)) *ys* ! *i*)
(*map* (*non-pivot-base-gen um' ze' on' A'* (*pivot-positions-gen ze' A'*)) *ys* ! *i*)
unfolding *y-def*[*symmetric*] *nth-map*[*OF i*]
unfolding *non-pivot-base-gen-def Let-def dim vec-rel-def*
by (*intro conjI allI impI*, *force*, *insert main*, *auto simp*: *trans*(*3*))
qed
qed

lemma *eliminate-entries-gen-transfer*: **assumes** ∗[*transfer-rule*]: (*R* ===> *R* ===> *R*) *ad ad'*
(*R* ===> *R* ===> *R*) *mul mul'*
and *vs*: ⋀ *j*. *j* < *dim-row B'* ⟹ *R* (*vs j*) (*vs' j*)
and *i*: *i* < *dim-row B'*
and *B*: *mat-rel R B B'*

shows *mat-rel R*
(*eliminate-entries-gen ad mul vs B i j*)
(*eliminate-entries-gen ad′ mul′ vs′ B′ i j*)
proof −
note *BB* = *B*[*unfolded mat-rel-def*]
show *?thesis* **unfolding** *mat-rel-def dim-eliminate-entries-gen*
proof (*intro conjI impI allI*)
fix *i′ j′*
assume *ij′*: *i′* < *dim-row B′ j′* < *dim-col B′*
with *BB* **have** *ij*: *i′*< *dim-row B j′* < *dim-col B* **by** *auto*
have [*transfer-rule*]: *R* (*B* $$ (*i′*, *j′*)) (*B′* $$ (*i′*, *j′*)) **using** *BB ij′* **by** *auto*
have [*transfer-rule*]: *R* (*B* $$ (*i*, *j′*)) (*B′* $$ (*i*, *j′*)) **using** *BB ij′ i* **by** *auto*
have [*transfer-rule*]: *R* (*vs i′*) (*vs′ i′*) **using** *ij′ vs*[*of i′*] **by** *auto*
show *R* (*eliminate-entries-gen ad mul vs B i j* $$ (*i′*, *j′*))
(*eliminate-entries-gen ad′ mul′ vs′ B′ i j* $$ (*i′*, *j′*))
unfolding *eliminate-entries-gen-def index-mat*(*1*)[*OF ij*] *index-mat*(*1*)[*OF ij′*]
split
by *transfer-prover*
qed (*insert BB*, *auto*)
qed

context
fixes *ops* :: *′i arith-ops-record* (**structure**)
begin
private abbreviation (*input*) *zero* **where** *zero* ≡ *arith-ops-record.zero ops*
private abbreviation (*input*) *one* **where** *one* ≡ *arith-ops-record.one ops*
private abbreviation (*input*) *plus* **where** *plus* ≡ *arith-ops-record.plus ops*
private abbreviation (*input*) *times* **where** *times* ≡ *arith-ops-record.times ops*
private abbreviation (*input*) *minus* **where** *minus* ≡ *arith-ops-record.minus ops*
private abbreviation (*input*) *uminus* **where** *uminus* ≡ *arith-ops-record.uminus ops*
private abbreviation (*input*) *divide* **where** *divide* ≡ *arith-ops-record.divide ops*
private abbreviation (*input*) *inverse* **where** *inverse* ≡ *arith-ops-record.inverse ops*
private abbreviation (*input*) *modulo* **where** *modulo* ≡ *arith-ops-record.modulo ops*
private abbreviation (*input*) *normalize* **where** *normalize* ≡ *arith-ops-record.normalize ops*

definition *eliminate-entries-gen-zero* :: (*′a* ⇒ *′a* ⇒ *′a*) ⇒ (*′a* ⇒ *′a* ⇒ *′a*) ⇒ *′a* ⇒ (*integer* ⇒ *′a*) ⇒ *′a mat* ⇒ *nat* ⇒ *nat* ⇒ *′a mat* **where**
eliminate-entries-gen-zero minu time z v A I J = *mat* (*dim-row A*) (*dim-col A*) (λ (*i*, *j*).
if v (*integer-of-nat i*) ≠ *z* ∧ *i* ≠ *I then minu* (*A* $$ (*i*,*j*)) (*time* (*v* (*integer-of-nat i*)) (*A* $$ (*I*,*j*))) *else A* $$ (*i*,*j*))

definition *eliminate-entries-i* **where** *eliminate-entries-i* ≡ *eliminate-entries-gen-zero minus times zero*
definition *multrow-i* **where** *multrow-i* ≡ *mat-multrow-gen times*

lemma *dim-eliminate-entries-gen-zero*[*simp*]:
  dim-row (*eliminate-entries-gen-zero mm tt z v B i as*) = *dim-row B*
  dim-col (*eliminate-entries-gen-zero mm tt z v B i as*) = *dim-col B*
  unfolding *eliminate-entries-gen-zero-def* **by** *auto*

partial-function (*tailrec*) *gauss-jordan-main-i* :: *nat* ⇒ *nat* ⇒ *'i mat* ⇒ *nat* ⇒ *nat* ⇒ *'i mat* **where**
  [*code*]: *gauss-jordan-main-i nr nc A i j* = (
    if i < *nr* ∧ *j* < *nc then let aij* = *A* $$ (*i*,*j*) *in if aij* = *zero then*
      (*case* [*i'* . *i'* <− [*Suc i* ..< *nr*], *A* $$ (*i'*,*j*) ≠ *zero*]
        of [] ⇒ *gauss-jordan-main-i nr nc A i* (*Suc j*)
        | (*i'* # -) ⇒ *gauss-jordan-main-i nr nc* (*swaprows i i' A*) *i j*)
      else if aij = *one then let*
        v = (λ *i*. *A* $$ (*nat-of-integer i*,*j*)) *in*
        gauss-jordan-main-i nr nc
        (*eliminate-entries-i v A i j*) (*Suc i*) (*Suc j*)
      else let iaij = *inverse aij*; *A'* = *multrow-i i iaij A*;
        v = (λ *i*. *A'* $$ (*nat-of-integer i*,*j*))
        in gauss-jordan-main-i nr nc (*eliminate-entries-i v A' i j*) (*Suc i*) (*Suc j*)
    else A)

definition *gauss-jordan-single-i* :: *'i mat* ⇒ *'i mat* **where**
  gauss-jordan-single-i A ≡ *gauss-jordan-main-i* (*dim-row A*) (*dim-col A*) *A 0 0*

definition *find-base-vectors-i* :: *'i mat* ⇒ *'i vec list* **where**
  find-base-vectors-i A ≡ *find-base-vectors-gen uminus zero one A*
end

context *field-ops*
begin

lemma *right-total-poly-rel*[*transfer-rule*]: *right-total* (*mat-rel R*)
  using *right-total-mat-rel*[*of R*] *right-total* **.**

lemma *bi-unique-poly-rel*[*transfer-rule*]: *bi-unique* (*mat-rel R*)
  using *bi-unique-mat-rel*[*of R*] *bi-unique* **.**

lemma *eq-mat-rel*[*transfer-rule*]: (*mat-rel R* ===> *mat-rel R* ===> (=)) (=) (=)
  by (*rule mat-rel-eq*[*OF eq*])

lemma *multrow-i*[*transfer-rule*]: ((=) ===> *R* ===> *mat-rel R* ===> *mat-rel R*)
  (*multrow-i ops*) *multrow*
  using *multrow-transfer*[*of R*] *times* **unfolding** *multrow-i-def rel-fun-def* **by** *blast*

lemma *eliminate-entries-gen-zero*[*simp*]:
  assumes *mat-rel R A A′ I* < *dim-row A′* **shows**
  eliminate-entries-gen-zero minus times zero v A I J = *eliminate-entries-gen minus times* (*v o integer-of-nat*) *A I J*
  unfolding *eliminate-entries-gen-def eliminate-entries-gen-zero-def*
proof(*standard*,*goal-cases*)
  case (*1 i j*)
  have *d1*:*DP* (*A* $$ (*I*, *j*)) **and** *d2*:*DP* (*A* $$ (*i*, *j*)) **using** *assms DPR 1*
    unfolding *mat-rel-def dim-col-mat dim-row-mat*
    by (*metis Domainp.DomainI*)+
  have *e1*:$\bigwedge$ *x*. (*0*::′*a*) ∗ *x* = *0* **and** *e2*:$\bigwedge$ *x*. *x* − (*0*::′*a*) = *x* **by** *auto*
  from *e1*[*untransferred*,*OF d1*] *e2*[*untransferred*,*OF d2*] *1* **show** *?case* **by** *auto*
qed *auto*

lemma *eliminate-entries-i*: **assumes**
  vs: $\bigwedge$ *j*. *j* < *dim-row B′* ⟹ *R* (*vs* (*integer-of-nat j*)) (*vs′ j*)
  and *i*: *i* < *dim-row B′*
  and *B*: *mat-rel R B B′*
  shows *mat-rel R* (*eliminate-entries-i ops vs B i j*)
    (*eliminate-entries vs′ B′ i j*)
  unfolding *eliminate-entries-i-def eliminate-entries-gen-zero*[*OF B i*]
  by (*rule eliminate-entries-gen-transfer*, *insert assms*, *auto simp*: *plus times minus*)

lemma *gauss-jordan-main-i*:
  nr = *dim-row A′* ⟹ *nc* = *dim-col A′* ⟹ *mat-rel R A A′* ⟹ *i* ≤ *nr* ⟹ *j* ≤ *nc* ⟹
    mat-rel R (*gauss-jordan-main-i ops nr nc A i j*) (*fst* (*gauss-jordan-main A′ B′ i j*))
proof −
  obtain *P* **where** *P*: *P* = (*A′*,*i*,*j*) **by** *auto*
  let *?Rel* = *measures* [λ (*A′* :: ′*a mat*,*i*,*j*). *nc* − *j*, λ (*A′*,*i*,*j*). *if A′* $$ (*i*,*j*) = *0 then 1 else 0*]
  have *wf*: *wf ?Rel* **by** *simp*
  show *nr* = *dim-row A′* ⟹ *nc* = *dim-col A′* ⟹ *mat-rel R A A′* ⟹ *i* ≤ *nr* ⟹ *j* ≤ *nc* ⟹
    mat-rel R (*gauss-jordan-main-i ops nr nc A i j*) (*fst* (*gauss-jordan-main A′ B′ i j*))
    using *P*
  proof (*induct P arbitrary*: *A′ B′ A i j rule*: *wf-induct*[*OF wf*])
    case (*1 P A′ B′ A i j*)
    note *prems* = *1*(*2*−*6*)
    note *P* = *1*(*7*)
    note *A*[*transfer-rule*] = *prems*(*3*)
    note *IH* = *1*(*1*)[*rule-format*, *OF* - - - - - - *refl*]
    note *simps* = *gauss-jordan-main-code*[*of A′ B′ i j*, *unfolded Let-def*, *folded prems*(*1*−*2*)]

gauss-jordan-main-i.simps[*of ops nr nc A i j*] *Let-def if-True if-False*
show *?case*
proof (*cases* $i < nr \wedge j < nc$)
case *False*
hence *id*: $(i < nr \wedge j < nc) = False$ **by** *simp*
show *?thesis* **unfolding** *simps id* **by** *simp transfer-prover*
next
case *True* **note** $ij' = this$
hence *id*: $(i < nr \wedge j < nc) = True$ $\bigwedge x\ y\ z.$ (*if* $x = x$ *then* y *else* z) $= y$ **by** *auto*
from *True prems* **have** *ij* [*transfer-rule*]:*R* (*A* $$ (*i*,*j*)) (*A'* $$ (*i*,*j*))
unfolding *mat-rel-def* **by** *auto*
from *True prems* **have** *i*: $i <$ *dim-row* A' $j <$ *dim-col* A' **and** i': $i < nr$ $j < nc$ **by** *auto*
{
fix *i*
assume $i <$ *dim-row* A'
with *i True prems* **have** *R*[*transfer-rule*]:*R* (*A* $$ (*i*,*j*)) (*A'* $$ (*i*,*j*))
unfolding *mat-rel-def* **by** *auto*
have (*A* $$ (*i*,*j*) = *zero*) = (*A'* $$ (*i*,*j*) = *0*) **by** *transfer-prover*
note *this R*
} **note** *eq-gen* = *this*
have *eq*: (*A* $$ (*i*,*j*) = *zero*) = (*A'* $$ (*i*,*j*) = *0*)
(*A* $$ (*i*,*j*) = *one*) = (*A'* $$ (*i*,*j*) = *1*)
by *transfer-prover+*
show *?thesis*
proof (*cases A'* $$ (*i*, *j*) = *0*)
case *True*
hence *eq*: *A* $$ (*i*,*j*) = *zero* **using** *eq* **by** *auto*
let *?is* = [*i'* . *i'* <− [*Suc i* ..< *nr*], *A* $$ (*i'*,*j*) ≠ *zero*]
let *?is'* = [*i'* . *i'* <− [*Suc i* ..< *nr*], *A'* $$ (*i'*,*j*) ≠ *0*]
define *xs* **where** *xs* = [*Suc i*..<*nr*]
have *xs*: *set xs* ⊆ {*0* ..< *dim-row A'*} **unfolding** *xs-def* **using** *prems* **by** *auto*
hence *id'*: *?is* = *?is'* **unfolding** *xs-def*[*symmetric*]
by (*induct xs*, *insert eq-gen*, *auto*)
show *?thesis*
proof (*cases ?is'*)
case *Nil*
have *?thesis* = (*mat-rel R* (*gauss-jordan-main-i ops nr nc A i* (*Suc j*))
(*fst* (*gauss-jordan-main A' B' i* (*Suc j*))))
unfolding *True simps id eq* **unfolding** *Nil id'*[*unfolded Nil*] **by** *simp*
also have . . .
by (*rule IH*, *insert i prems P*, *auto*)
finally show *?thesis* **.**
next
case (*Cons i' idx'*)
from *arg-cong*[*OF this*, *of set*] *i*
have *i'*: *i'* < *nr A'* $$ (*i'*, *j*) ≠ *0* **by** *auto*

with *ij′ prems*(*1*−*2*) **have** ∗: $i' <$ *dim-row* A' $i <$ *dim-row* A' $j <$ *dim-col* A' **by** *auto*
have *rel*: ((*swaprows i i′ A′*, *i*, *j*), *P*) ∈ *?Rel*
by (*simp add*: *P True* ∗ *i′*)
have *?thesis* = (*mat-rel R* (*gauss-jordan-main-i ops nr nc* (*swaprows i i′ A*) *i j*)
(*fst* (*gauss-jordan-main* (*swaprows i i′ A′*) (*swaprows i i′ B′*) *i j*)))
unfolding *True simps id eq Cons id′*[*unfolded Cons*] **by** *simp*
also have . . .
by (*rule IH*[*OF rel - - swap-rows-transfer*], *insert i i′ prems P True*, *auto*)
finally show *?thesis* **.**
qed
next
case *False*
from *False eq* **have** *neq*: (*A* \$\$ (*i*, *j*) = *zero*) = *False* (*A′* \$\$ (*i*, *j*) = *0*) = *False* **by** *auto*
{
fix *B B′ i*
assume *B*[*transfer-rule*]: *mat-rel R B B′* **and** *dim*: *dim-col B′* = *nc* **and** *i*: $i <$ *dim-row* B'
from *dim i True* **have** $j <$ *dim-col* B' **by** *simp*
with *B i* **have** *R* (*B* \$\$ (*i*,*j*)) (*B′* \$\$ (*i*,*j*))
by (*simp add*: *mat-rel-def*)
} **note** *vec-rel* = *this*
from *prems* **have** *dim*: *dim-row A* = *dim-row A′* **unfolding** *mat-rel-def* **by** *auto*
show *?thesis*
proof (*cases A′* \$\$ (*i*, *j*) = *1*)
case *True*
from *True eq* **have** *eq*: (*A* \$\$ (*i*,*j*) = *one*) = *True* (*A′* \$\$ (*i*,*j*) = *1*) = *True* **by** *auto*
note *rel* = *vec-rel*[*OF A*]
show *?thesis* **unfolding** *simps id neq eq*
by (*rule IH*[*OF - - - eliminate-entries-i*], *insert rel prems ij i P dim*, *auto*)
next
case *False*
from *False eq* **have** *eq*: (*A* \$\$ (*i*,*j*) = *one*) = *False* (*A′* \$\$ (*i*,*j*) = *1*) = *False* **by** *auto*
show *?thesis* **unfolding** *simps id neq eq*
proof (*rule IH*, *goal-cases*)
case *4*
have *A′*: *mat-rel R* (*multrow-i ops i* (*inverse* (*A* \$\$ (*i*, *j*))) *A*)
(*multrow i* (*inverse-class.inverse* (*A′* \$\$ (*i*, *j*))) *A′*) **by** *transfer-prover*
note *rel* = *vec-rel*[*OF A′*]
show *?case*
by (*rule eliminate-entries-i*[*OF - - A′*], *insert rel prems i dim*, *auto*)
qed (*insert prems i P*, *auto*)
qed

qed
qed
qed
qed

lemma *gauss-jordan-i*[*transfer-rule*]:
(*mat-rel R* ===> *mat-rel R*) (*gauss-jordan-single-i ops*) *gauss-jordan-single*
proof (*intro rel-funI*)
fix *A A′*
assume *A*: *mat-rel R A A′*
show *mat-rel R* (*gauss-jordan-single-i ops A*) (*gauss-jordan-single A′*)
unfolding *gauss-jordan-single-def gauss-jordan-single-i-def gauss-jordan-def*
by (*rule gauss-jordan-main-i*[*OF - - A*], *insert A*, *auto simp*: *mat-rel-def*)
qed

lemma *find-base-vectors-i*[*transfer-rule*]:
(*mat-rel R* ===> *list-all2* (*vec-rel R*)) (*find-base-vectors-i ops*) *find-base-vectors*
unfolding *find-base-vectors-i-def*[*abs-def*]
using *find-base-vectors-transfer*[*OF eq*] *uminus zero one*
unfolding *rel-fun-def* **by** *blast*

end

lemma *list-of-vec-transfer*[*transfer-rule*]: (*vec-rel A* ===> *list-all2 A*) *list-of-vec list-of-vec*
unfolding *rel-fun-def vec-rel-def vec-eq-iff list-all2-conv-all-nth*
by *auto*

lemma *IArray-sub′*[*simp*]: $i < IArray.length\ a \Longrightarrow IArray.sub'\ (a,\ integer\text{-}of\text{-}nat\ i) = IArray.sub\ a\ i$
by *auto*

lift-definition *eliminate-entries-i2* ::
$'a \Rightarrow ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow (integer \Rightarrow 'a) \Rightarrow 'a\ mat\text{-}impl \Rightarrow integer \Rightarrow 'a\ mat\text{-}impl$ **is**
λ *z mminus ttimes v* (*nr*, *nc*, *a*) *i′*.
(*nr*,*nc*,*let ai′* = *IArray.sub′* (*a*, *i′*) *in* (*IArray.tabulate* (*integer-of-nat nr*, λ *i*. *let ai* = *IArray.sub′* (*a*, *i*) *in*
if i = *i′ then ai else*
let vi′j = *v i*
in if vi′j = *z then ai*
else
IArray.tabulate (*integer-of-nat nc*, λ *j*. *mminus* (*IArray.sub′* (*ai*, *j*)) (*ttimes vi′j*
(*IArray.sub′* (*ai′*, *j*))))
)))
proof(*goal-cases*)
case (*1 z mm tt vec prod nat2*)
thus *?case* **by**(*cases prod*;*cases snd* (*snd prod*);*auto simp*:*Let-def*)

```
qed

lemma eliminate-entries-gen-zero [simp]:
  assumes i<(dim-row A) j<(dim-col A) shows
  eliminate-entries-gen-zero mminus ttimes z v A I J $$ (i, j) =
    (if v (integer-of-nat i) = z ∨ i = I then A $$ (i,j) else mminus (A $$ (i,j))
(ttimes (v (integer-of-nat i)) (A $$ (I,j))))
using assms unfolding eliminate-entries-gen-zero-def by auto


lemma eliminate-entries-gen [simp]:
  assumes i<(dim-row A) j<(dim-col A) shows
  eliminate-entries-gen mminus ttimes v A I J $$ (i, j) =
   (if i = I then A $$ (i,j) else mminus (A $$ (i,j)) (ttimes (v i) (A $$ (I,j))))
using assms unfolding eliminate-entries-gen-def by auto

lemma dim-mat-impl [simp]:
  dim-row (mat-impl x) = dim-row-impl x
  dim-col (mat-impl x) = dim-col-impl x
   by (cases Rep-mat-impl x;auto simp:mat-impl.rep-eq dim-row-def dim-col-def
dim-row-impl.rep-eq dim-col-impl.rep-eq)+

lemma dim-eliminate-entries-i2 [simp]:
  dim-row-impl (eliminate-entries-i2 z mm tt v m i) = dim-row-impl m
  dim-col-impl (eliminate-entries-i2 z mm tt v m i) = dim-col-impl m
  by (transfer, auto)+

lemma tabulate-nth: i < n ⟹ IArray.tabulate (integer-of-nat n, f) !! i = f
(integer-of-nat i)
  using of-fun-nth[of i n] by auto

lemma eliminate-entries-i2[code]:eliminate-entries-gen-zero mm tt z v (mat-impl
m) i j
   = (if i < dim-row-impl m
     then mat-impl (eliminate-entries-i2 z mm tt v m (integer-of-nat i))
     else (Code.abort (STR ''index out of range in eliminate-entries'')
       (λ -. eliminate-entries-gen-zero mm tt z v (mat-impl m) i j)))
proof (cases i < dim-row-impl m)
  case True
  hence id: (i < dim-row-impl m) = True by simp
  show ?thesis unfolding id if-True
  proof (standard;goal-cases)
    case (1 i j)
    have dims: i < dim-row (mat-impl m) j < dim-col (mat-impl m) using 1 by
(auto simp:eliminate-entries-i2.rep-eq)
    then show ?case unfolding eliminate-entries-gen-zero[OF dims] using True
    proof(transfer, goal-cases)
      case (1 i m j ia v z mm tt)
      obtain nr nc M where m: m = (nr,nc,M) by (cases m)
```

note *1* = *1*[*unfolded m*, *simplified*]
have *mk*: $\bigwedge$ *f*. *mk-mat nr nc f* (*i*,*j*) = *f* (*i*,*j*)
$\bigwedge$ *f*. *mk-mat nr nc f* (*ia*,*j*) = *f* (*ia*,*j*)
using *1* **unfolding** *mk-mat-def mk-vec-def* **by** *auto*
note *of-fun* = *of-fun-nth*[*OF 1*(*2*)] *of-fun-nth*[*OF 1*(*3*)] *tabulate-nth*[*OF 1*(*2*)] *tabulate-nth*[*OF 1*(*3*)]
let *?c1* = *v* (*integer-of-nat i*) = *z*
show *?case*
proof (*cases ?c1* ∨ *i* = *ia*)
case *True*
hence *id*: (*if ?c1* ∨ *i* = *ia then x else y*) = *x*
(*if integer-of-nat i* = *integer-of-nat ia then x else if ?c1 then x else y*) = *x* **for** *x y*
by *auto*
show *?thesis* **unfolding** *id m o-def Let-def split snd-conv mk of-fun* **by** (*auto simp*: *1*)
next
case *False*
hence *id*: *?c1* = *False* (*integer-of-nat i* = *integer-of-nat ia*) = *False* (*False* ∨ *i* = *ia*) = *False*
by (*auto simp add*: *integer-of-nat-eq-of-nat*)
show *?thesis* **unfolding** *m o-def Let-def split snd-conv mk of-fun id if-False*
by (*auto simp*: *1*)
qed
qed
qed (*auto simp*:*eliminate-entries-i2*.*rep-eq*)
qed *auto*

end
theory *More-Missing-Multiset*
imports
HOL−Combinatorics.*Permutations*
Polynomial-Factorization.*Missing-Multiset*
begin

lemma *rel-mset-free*:
assumes *rel*: *rel-mset rel X Y* **and** *xs*: *mset xs* = *X*
shows ∃ *ys*. *mset ys* = *Y* ∧ *list-all2 rel xs ys*
proof−
from *rel*[*unfolded rel-mset-def*] **obtain** *xs′ ys′*
where *xs′*: *mset xs′* = *X* **and** *ys′*: *mset ys′* = *Y* **and** *xsys′*: *list-all2 rel xs′ ys′* **by** *auto*
from *xs′ xs* **have** *mset xs* = *mset xs′* **by** *auto*
from *mset-eq-permutation*[*OF this*]
obtain *f* **where** *perm*: *f permutes* {..<*length xs′*} **and** *xs′*: *permute-list f xs′* = *xs*.
then have [*simp*]: *length xs′* = *length xs* **by** *auto*
from *permute-list-nth*[*OF perm*, *unfolded xs′*] **have** ∗: $\bigwedge$*i*. *i* < *length xs* $\Longrightarrow$ *xs* ! *i* = *xs′* ! *f i* **by** *auto*

note [*simp*] = *list-all2-lengthD*[*OF xsys'*,*symmetric*]
note [*simp*] = *atLeast0LessThan*[*symmetric*]
note *bij* = *permutes-bij*[*OF perm*]
define *ys* **where** *ys* ≡ *map* (*nth ys'* ∘ *f*) [*0..<length ys'*]
then have [*simp*]: *length ys* = *length ys'* **by** *auto*
have *mset ys* = *mset* (*map* (*nth ys'*) (*map f* [*0..<length ys'*]))
unfolding *ys-def* **by** *auto*
also have ... = *image-mset* (*nth ys'*) (*image-mset f* (*mset* [*0..<length ys'*]))
by (*simp add*: *multiset.map-comp*)
also have (*mset* [*0..<length ys'*]) = *mset-set* {*0..<length ys'*}
by (*metis mset-sorted-list-of-multiset sorted-list-of-mset-set sorted-list-of-set-range*)

also have *image-mset f* (...) = *mset-set* (*f* ' {..<*length ys'*})
using *subset-inj-on*[*OF bij-is-inj*[*OF bij*]] **by** (*subst image-mset-mset-set*, *auto*)
also have ... = *mset* [*0..<length ys'*] **using** *perm* **by** (*simp add*: *permutes-image*)
also have *image-mset* (*nth ys'*) ... = *mset ys'* **by**(*fold mset-map*, *unfold map-nth*, *auto*)
finally have *mset ys* = *Y* **using** *ys'* **by** *auto*
moreover have *list-all2 rel xs ys*
proof(*rule list-all2-all-nthI*)
fix *i* **assume** *i*: *i* < *length xs*
with ∗ **have** *xs* ! *i* = *xs'* ! *f i* **by** *auto*
also from *i permutes-in-image*[*OF perm*]
have *rel* (*xs'* ! *f i*) (*ys'* ! *f i*) **by** (*intro list-all2-nthD*[*OF xsys'*], *auto*)
finally show *rel* (*xs* ! *i*) (*ys* ! *i*) **unfolding** *ys-def* **using** *i* **by** *simp*
qed *simp*
ultimately show *?thesis* **by** *auto*
qed

lemma *rel-mset-split*:
assumes *rel*: *rel-mset rel* (*X1*+*X2*) *Y*
shows ∃ *Y1 Y2*. *Y* = *Y1* + *Y2* ∧ *rel-mset rel X1 Y1* ∧ *rel-mset rel X2 Y2*
proof−
obtain *xs1* **where** *xs1*: *mset xs1* = *X1* **using** *ex-mset* **by** *auto*
obtain *xs2* **where** *xs2*: *mset xs2* = *X2* **using** *ex-mset* **by** *auto*
from *xs1 xs2* **have** *mset* (*xs1* @ *xs2*) = *X1* + *X2* **by** *auto*
from *rel-mset-free*[*OF rel this*] **obtain** *ys*
where *ys*: *mset ys* = *Y list-all2 rel* (*xs1* @ *xs2*) *ys* **by** *auto*
then obtain *ys1 ys2*
where *ys12*: *ys* = *ys1* @ *ys2*
and *xs1ys1*: *list-all2 rel xs1 ys1*
and *xs2ys2*: *list-all2 rel xs2 ys2*
using *list-all2-append1* **by** *blast*
from *ys12 ys* **have** *Y* = *mset ys1* + *mset ys2* **by** *auto*
moreover from *xs1 xs1ys1* **have** *rel-mset rel X1* (*mset ys1*) **unfolding** *rel-mset-def* **by** *auto*
moreover from *xs2 xs2ys2* **have** *rel-mset rel X2* (*mset ys2*) **unfolding** *rel-mset-def* **by** *auto*
ultimately show *?thesis* **by** (*subst exI*[*of - mset ys1*], *subst exI*[*of - mset*

ys2],*auto*)
qed

lemma *rel-mset-OO*:
assumes *AB*: *rel-mset R A B* **and** *BC*: *rel-mset S B C*
shows *rel-mset* (*R OO S*) *A C*
proof−
from *AB* **obtain** *as bs* **where** *A-as*: $A = mset\ as$ **and** *B-bs*: $B = mset\ bs$ **and** *as-bs*: *list-all2 R as bs*
by (*auto simp*: *rel-mset-def*)
from *rel-mset-free*[*OF BC*] *B-bs* **obtain** *cs* **where** *C-cs*: $C = mset\ cs$ **and** *bs-cs*: *list-all2 S bs cs*
by *auto*
from *list-all2-trans*[*OF - as-bs bs-cs*, *of R OO S*] *A-as C-cs*
show *?thesis* **by** (*auto simp*: *rel-mset-def*)
qed

lemma *ex-mset-zip-right*:
assumes $length\ xs = length\ ys$ $mset\ ys' = mset\ ys$
shows $\exists xs'.\ length\ ys' = length\ xs' \wedge mset\ (zip\ xs'\ ys') = mset\ (zip\ xs\ ys)$
using *assms*
proof (*induct xs ys arbitrary*: *ys' rule*: *list-induct2*)
case *Nil*
thus *?case*
by *auto*
next
case (*Cons x xs y ys ys'*)
obtain *j* **where** *j-len*: $j < length\ ys'$ **and** *nth-j*: $ys' ! j = y$
by (*metis Cons.prems in-set-conv-nth list.set-intros*(*1*) *mset-eq-setD*)

define *ysa* **where** $ysa = take\ j\ ys'\ @\ drop\ (Suc\ j)\ ys'$
have $mset\ ys' = \{\#y\#\} + mset\ ysa$
unfolding *ysa-def* **using** *j-len nth-j*
by (*metis Cons-nth-drop-Suc union-mset-add-mset-right add-mset-remove-trivial add-diff-cancel-left'*
append-take-drop-id mset.simps(*2*) *mset-append*)
hence *ms-y*: $mset\ ysa = mset\ ys$
by (*simp add*: *Cons.prems*)
then obtain *xsa* **where**
len-a: $length\ ysa = length\ xsa$ **and** *ms-a*: $mset\ (zip\ xsa\ ysa) = mset\ (zip\ xs\ ys)$
using *Cons.hyps*(*2*) **by** *blast*

define *xs'* **where** $xs' = take\ j\ xsa\ @\ x\ \#\ drop\ j\ xsa$
have *ys'*: $ys' = take\ j\ ysa\ @\ y\ \#\ drop\ j\ ysa$
using *ms-y j-len nth-j Cons.prems ysa-def*
by (*metis append-eq-append-conv append-take-drop-id diff-Suc-Suc Cons-nth-drop-Suc length-Cons*
length-drop size-mset)

have *j-len'*: $j \leq$ *length ysa*
using *j-len ys' ysa-def*
by (*metis add-Suc-right append-take-drop-id length-Cons length-append less-eq-Suc-le not-less*)
have *length ys'* = *length xs'*
unfolding *xs'-def* **using** *Cons.prems len-a ms-y*
by (*metis add-Suc-right append-take-drop-id length-Cons length-append mset-eq-length*)
moreover have *mset* (*zip xs' ys'*) = *mset* (*zip* (*x* # *xs*) (*y* # *ys*))
unfolding *ys' xs'-def*
apply (*rule HOL.trans*[*OF mset-zip-take-Cons-drop-twice*])
using *j-len'* **by** (*auto simp*: *len-a ms-a*)
ultimately show *?case*
by *blast*
qed

lemma *list-all2-reorder-right-invariance*:
assumes *rel*: *list-all2 R xs ys* **and** *ms-y*: *mset ys'* = *mset ys*
shows $\exists$ *xs'*. *list-all2 R xs' ys'* $\wedge$ *mset xs'* = *mset xs*
proof −
have *len*: *length xs* = *length ys*
using *rel list-all2-conv-all-nth* **by** *auto*
obtain *xs'* **where**
len': *length xs'* = *length ys'* **and** *ms-xy*: *mset* (*zip xs' ys'*) = *mset* (*zip xs ys*)
using *len ms-y* **by** (*metis ex-mset-zip-right*)
have *list-all2 R xs' ys'*
using *assms*(*1*) *len' ms-xy* **unfolding** *list-all2-iff* **by** (*blast dest*: *mset-eq-setD*)
moreover have *mset xs'* = *mset xs*
using *len len' ms-xy map-fst-zip mset-map* **by** *metis*
ultimately show *?thesis*
by *blast*
qed

lemma *rel-mset-via-perm*: *rel-mset rel* (*mset xs*) (*mset ys*) $\longleftrightarrow$ ($\exists$ *zs*. *mset xs* = *mset zs* $\wedge$ *list-all2 rel zs ys*)
proof (*unfold rel-mset-def*, *intro iffI*, *goal-cases*)
case *1*
then obtain *zs ws* **where** *zs*: *mset zs* = *mset xs* **and** *ws*: *mset ws* = *mset ys* **and** *zsws*: *list-all2 rel zs ws* **by** *auto*
note *list-all2-reorder-right-invariance*[*OF zsws ws*[*symmetric*], *unfolded zs*]
then show *?case* **by** (*auto dest*: *sym*)
next
case *2*
from *this* **show** *?case* **by** *force*
qed

end
theory *Unique-Factorization*
imports
Polynomial-Interpolation.Ring-Hom-Poly

Polynomial-Factorization.Polynomial-Divisibility
HOL−Combinatorics.Permutations
HOL−Computational-Algebra.Euclidean-Algorithm
Containers.Containers-Auxiliary
More-Missing-Multiset
HOL−Algebra.Divisibility
begin

hide-const(**open**)
Divisibility.prime
Divisibility.irreducible

hide-fact(**open**)
Divisibility.irreducible-def
Divisibility.irreducibleI
Divisibility.irreducibleD
Divisibility.irreducibleE

hide-const (**open**) *Rings.coprime*

lemma *irreducible-uminus* [*simp*]:
fixes $a::'a::idom$
shows *irreducible* $(-a) \longleftrightarrow$ *irreducible* a
using *irreducible-mult-unit-left*[*of* $-1::'a$] **by** *auto*

context *comm-monoid-mult* **begin**

definition *coprime* :: $'a \Rightarrow 'a \Rightarrow bool$
where *coprime-def′*: $coprime\ p\ q \equiv \forall r.\ r\ dvd\ p \longrightarrow r\ dvd\ q \longrightarrow r\ dvd\ 1$

lemma *coprimeI*:
assumes $\bigwedge r.\ r\ dvd\ p \Longrightarrow r\ dvd\ q \Longrightarrow r\ dvd\ 1$
shows *coprime p q* **using** *assms* **by** (*auto simp*: *coprime-def′*)

lemma *coprimeE*:
assumes *coprime p q*
and $(\bigwedge r.\ r\ dvd\ p \Longrightarrow r\ dvd\ q \Longrightarrow r\ dvd\ 1) \Longrightarrow thesis$
shows *thesis* **using** *assms* **by** (*auto simp*: *coprime-def′*)

lemma *coprime-commute* [*ac-simps*]:
$coprime\ p\ q \longleftrightarrow coprime\ q\ p$
by (*auto simp add*: *coprime-def′*)

lemma *not-coprime-iff-common-factor*:
$\neg\ coprime\ p\ q \longleftrightarrow (\exists r.\ r\ dvd\ p \wedge r\ dvd\ q \wedge \neg\ r\ dvd\ 1)$
by (*auto simp add*: *coprime-def′*)

end

lemma (**in** *algebraic-semidom*) *coprime-iff-coprime* [*simp*, *code*]:
coprime = *Rings.coprime*
by (*simp add*: *fun-eq-iff coprime-def coprime-def'*)

lemma (**in** *comm-semiring-1*) *coprime-0* [*simp*]:
coprime p 0 ⟷ *p dvd 1 coprime 0 p* ⟷ *p dvd 1*
by (*auto intro*: *coprimeI elim*: *coprimeE dest*: *dvd-trans*)

lemma *dvd-rewrites*: *dvd.dvd* ((∗)) = (*dvd*) **by** (*unfold dvd.dvd-def dvd-def*, *rule*)

3.3 Interfacing UFD properties

hide-const (**open**) *Divisibility.irreducible*

context *comm-monoid-mult-isom* **begin**
lemma *coprime-hom*[*simp*]: *coprime* (*hom x*) *y'* ⟷ *coprime x* (*Hilbert-Choice.inv hom y'*)
proof−
show *?thesis* **by** (*unfold coprime-def'*, *fold ball-UNIV*, *subst surj*[*symmetric*], *simp*)
qed
lemma *coprime-inv-hom*[*simp*]: *coprime* (*Hilbert-Choice.inv hom x'*) *y* ⟷ *coprime x'* (*hom y*)
proof−
interpret *inv*: *comm-monoid-mult-isom Hilbert-Choice.inv hom*..
show *?thesis* **by** *simp*
qed
end

3.3.1 Original part

lemma *dvd-dvd-imp-smult*:
fixes *p q* :: *'a* :: *idom poly*
assumes *pq*: *p dvd q* **and** *qp*: *q dvd p* **shows** ∃ *c*. *p* = *smult c q*
proof (*cases p* = *0*)
case *True* **then show** *?thesis* **by** *auto*
next
case *False*
from *qp* **obtain** *r* **where** *r*: *p* = *q* ∗ *r* **by** (*elim dvdE*, *auto*)
with *False qp* **have** *r0*: *r* ≠ *0* **and** *q0*: *q* ≠ *0* **by** *auto*
with *divides-degree*[*OF pq*] *divides-degree*[*OF qp*] *False*
have *degree p* = *degree q* **by** *auto*
with *r degree-mult-eq*[*OF q0 r0*] **have** *degree r* = *0* **by** *auto*
from *degree-0-id*[*OF this*] **obtain** *c* **where** *r* = [:*c*:] **by** *metis*
from *r*[*unfolded this*] **show** *?thesis* **by** *auto*
qed

lemma *dvd-const*:
  assumes *pq*: (*p*::′*a*::*semidom poly*) *dvd q* **and** *q0*: $q \neq 0$ **and** *degq*: *degree* $q = 0$
  shows *degree* $p = 0$
proof−
  from *dvdE*[*OF pq*] **obtain** *r* **where** ∗: $q = p * r$.
  with *q0* **have** $p \neq 0$ $r \neq 0$ **by** *auto*
  from *degree-mult-eq*[*OF this*] *degq* ∗ **show** *degree* $p = 0$ **by** *auto*
qed

context *Rings.dvd* **begin**
  abbreviation *ddvd* (**infix** *ddvd 40*) **where** x *ddvd* $y \equiv x$ *dvd* $y \wedge y$ *dvd* x
  lemma *ddvd-sym*[*sym*]: x *ddvd* $y \Longrightarrow y$ *ddvd* x **by** *auto*
end

context *comm-monoid-mult* **begin**
  lemma *ddvd-trans*[*trans*]: x *ddvd* $y \Longrightarrow y$ *ddvd* $z \Longrightarrow x$ *ddvd* z **using** *dvd-trans*
by *auto*
  lemma *ddvd-transp*: *transp* (*ddvd*) **by** (*intro transpI*, *fact ddvd-trans*)
end

context *comm-semiring-1* **begin**

definition *mset-factors* **where** *mset-factors F p* ≡
  $F \neq \{\#\} \wedge (\forall f.\ f \in\# F \longrightarrow$ *irreducible* $f) \wedge p =$ *prod-mset* F

lemma *mset-factorsI*[*intro*!]:
  assumes $\bigwedge f.\ f \in\# F \Longrightarrow$ *irreducible* f **and** $F \neq \{\#\}$ **and** *prod-mset* $F = p$
  shows *mset-factors F p*
  unfolding *mset-factors-def* **using** *assms* **by** *auto*

lemma *mset-factorsD*:
  assumes *mset-factors F p*
  shows $f \in\# F \Longrightarrow$ *irreducible* f **and** $F \neq \{\#\}$ **and** *prod-mset* $F = p$
  using *assms*[*unfolded mset-factors-def*] **by** *auto*

lemma *mset-factorsE*[*elim*]:
  assumes *mset-factors F p*
    and $(\bigwedge f.\ f \in\# F \Longrightarrow$ *irreducible* $f) \Longrightarrow F \neq \{\#\} \Longrightarrow$ *prod-mset* $F = p \Longrightarrow$
thesis
  shows *thesis*
  using *assms*[*unfolded mset-factors-def*] **by** *auto*

lemma *mset-factors-imp-not-is-unit*:
  assumes *mset-factors F p*
  shows ¬ *p dvd 1*
proof(*cases F*)
  case *empty* **with** *assms* **show** *?thesis* **by** *auto*
next

case (*add f F*)
with *assms* **have** $\neg$ *f dvd 1 p* = *f* $*$ *prod-mset F* **by** (*auto intro*!: *irreducible-not-unit*)
then show *?thesis* **by** *auto*
qed

definition *primitive-poly* **where** *primitive-poly f* $\equiv \forall$ *d.* ($\forall$ *i. d dvd coeff f i*) $\longrightarrow$ *d dvd 1*

end

lemma(**in** *semidom*) *mset-factors-imp-nonzero*:
assumes *mset-factors F p*
shows *p* $\neq$ *0*
proof
assume *p* = *0*
moreover from *assms* **have** *prod-mset F* = *p* **by** *auto*
ultimately obtain *f* **where** *f* $\in\#$ *F f* = *0* **by** *auto*
with *assms* **show** *False* **by** *auto*
qed

class *ufd* = *idom* +
assumes *mset-factors-exist*: $\bigwedge$*x. x* $\neq$ *0* $\Longrightarrow \neg$ *x dvd 1* $\Longrightarrow \exists$ *F. mset-factors F x*
and *mset-factors-unique*: $\bigwedge$*x F G. mset-factors F x* $\Longrightarrow$ *mset-factors G x* $\Longrightarrow$ *rel-mset* (*ddvd*) *F G*

3.3.2 Connecting to HOL/Divisibility

context *comm-semiring-1* **begin**

abbreviation *mk-monoid* $\equiv$ (|*carrier* = *UNIV* $-$ {*0*}, *mult* = ($*$), *one* = *1*|)

lemma *carrier-0*[*simp*]: *x* $\in$ *carrier mk-monoid* $\longleftrightarrow$ *x* $\neq$ *0* **by** *auto*

lemmas *mk-monoid-simps* = *carrier-0 monoid.simps*

abbreviation *irred* **where** *irred* $\equiv$ *Divisibility.irreducible mk-monoid*
abbreviation *factor* **where** *factor* $\equiv$ *Divisibility.factor mk-monoid*
abbreviation *factors* **where** *factors* $\equiv$ *Divisibility.factors mk-monoid*
abbreviation *properfactor* **where** *properfactor* $\equiv$ *Divisibility.properfactor mk-monoid*

lemma *factors*: *factors fs y* $\longleftrightarrow$ *prod-list fs* = *y* $\wedge$ *Ball* (*set fs*) *irred*
proof $-$
have *prod-list fs* = *foldr* ($*$) *fs 1* **by** (*induct fs*, *auto*)
thus *?thesis* **unfolding** *factors-def* **by** *auto*
qed

lemma *factor*: *factor x y* $\longleftrightarrow$ ($\exists$ *z. z* $\neq$ *0* $\wedge$ *x* $*$ *z* = *y*) **unfolding** *factor-def* **by** *auto*

lemma *properfactor-nz*:
shows $(y :: 'a) \neq 0 \Longrightarrow$ *properfactor x y* $\longleftrightarrow$ *x dvd y* $\wedge \neg$ *y dvd x*
by (*auto simp*: *properfactor-def factor-def dvd-def*)

lemma *mem-Units*[*simp*]: $y \in$ *Units mk-monoid* $\longleftrightarrow$ *y dvd 1*
unfolding *dvd-def Units-def* **by** (*auto simp*: *ac-simps*)

end

context *idom* **begin**
lemma *irred-0*[*simp*]: *irred* $(0::'a)$ **by** (*unfold Divisibility.irreducible-def*, *auto simp*: *factor properfactor-def*)
lemma *factor-idom*[*simp*]: *factor* $(x::'a)$ $y \longleftrightarrow$ (*if* $y = 0$ *then* $x = 0$ *else* *x dvd y*)
by (*cases* $y = 0$; *auto intro*: *exI*[*of* - *1*] *elim*: *dvdE simp*: *factor*)

lemma *associated-connect*[*simp*]: $(\sim_{mk\text{-}monoid}) = (ddvd)$ **by** (*intro ext*, *unfold associated-def*, *auto*)

lemma *essentially-equal-connect*[*simp*]:
essentially-equal mk-monoid fs gs $\longleftrightarrow$ *rel-mset* (*ddvd*) (*mset fs*) (*mset gs*)
by (*auto simp*: *essentially-equal-def rel-mset-via-perm*)

lemma *irred-idom-nz*:
assumes *x0*: $(x::'a) \neq 0$
shows *irred x* $\longleftrightarrow$ *irreducible x*
using *x0* **by** (*auto simp*: *irreducible-altdef Divisibility.irreducible-def properfactor-nz*)

lemma *dvd-dvd-imp-unit-mult*:
assumes *xy*: *x dvd y* **and** *yx*: *y dvd x*
shows $\exists z.$ *z dvd 1* $\wedge$ $y = x * z$
proof(*cases* $x = 0$)
case *True* **with** *xy* **show** *?thesis* **by** (*auto intro*: *exI*[*of* - *1*])
next
case *x0*: *False*
from *xy* **obtain** *z* **where** *z*: $y = x * z$ **by** (*elim dvdE*, *auto*)
from *yx* **obtain** *w* **where** *w*: $x = y * w$ **by** (*elim dvdE*, *auto*)
from *z w* **have** $x * (z * w) = x$ **by** (*auto simp*: *ac-simps*)
then have $z * w = 1$ **using** *x0* **by** *auto*
with *z* **show** *?thesis* **by** (*auto intro*: *exI*[*of* - *z*])
qed

lemma *irred-inner-nz*:
assumes *x0*: $x \neq 0$
shows ($\forall b.$ *b dvd x* $\longrightarrow \neg$ *x dvd b* $\longrightarrow$ *b dvd 1*) $\longleftrightarrow$ ($\forall a\ b.$ $x = a * b \longrightarrow$ *a dvd 1* $\vee$ *b dvd 1*) (**is** *?l* $\longleftrightarrow$ *?r*)

```
proof (intro iffI allI impI)
  assume l: ?l
  fix a b
  assume xab: x = a * b
  then have ax: a dvd x and bx: b dvd x by auto
  { assume a1: ¬ a dvd 1
    with l ax have xa: x dvd a by auto
   from dvd-dvd-imp-unit-mult[OF ax xa] obtain z where z1: z dvd 1 and xaz:
x = a * z by auto
    from xab x0 have a ≠ 0 by auto
    with xab xaz have b = z by auto
    with z1 have b dvd 1 by auto
  }
  then show a dvd 1 ∨ b dvd 1 by auto
next
  assume r: ?r
  fix b assume bx: b dvd x and xb: ¬ x dvd b
  then obtain a where xab: x = a * b by (elim dvdE, auto simp: ac-simps)
  with r consider a dvd 1 | b dvd 1 by auto
  then show b dvd 1
  proof(cases)
    case 2 then show ?thesis by auto
  next
    case 1
    then obtain c where ac1: a * c = 1 by (elim dvdE, auto)
    from xab have x * c = b * (a * c) by (auto simp: ac-simps)
    with ac1 have x * c = b by auto
    then have x dvd b by auto
    with xb show ?thesis by auto
  qed
qed

lemma irred-idom[simp]: irred x ⟷ x = 0 ∨ irreducible x
by (cases x = 0; simp add: irred-idom-nz irred-inner-nz irreducible-def)

lemma assumes x ≠ 0 and factors fs x and f ∈ set fs shows f ≠ 0
  using assms by (auto simp: factors)

lemma factors-as-mset-factors:
  assumes x0: x ≠ 0 and x1: x ≠ 1
  shows factors fs x ⟷ mset-factors (mset fs) x using assms
  by (auto simp: factors prod-mset-prod-list)


end

context ufd begin
 interpretation comm-monoid-cancel: comm-monoid-cancel mk-monoid::'a monoid
  apply (unfold-locales)
```

apply *simp-all*
using *mult-left-cancel*
apply (*auto simp*: *ac-simps*)
done
lemma *factors-exist*:
assumes $a \neq 0$
and $\neg$ *a dvd 1*
shows $\exists$*fs*. *set fs* $\subseteq$ *UNIV* $-$ $\{0\}$ $\wedge$ *factors fs a*
proof−
from *mset-factors-exist*[*OF assms*]
obtain *F* **where** *mset-factors F a* **by** *auto*
also from *ex-mset* **obtain** *fs* **where** *F* = *mset fs* **by** *metis*
finally have *fs*: *mset-factors* (*mset fs*) *a*.
then have *factors fs a* **using** *assms* **by** (*subst factors-as-mset-factors*, *auto*)
moreover have *set fs* $\subseteq$ *UNIV* $-$ $\{0\}$ **using** *fs* **by** (*auto elim*!: *mset-factorsE*)
ultimately show *?thesis* **by** *auto*
qed

lemma *factors-unique*:
assumes *fs*: *factors fs a*
and *gs*: *factors gs a*
and *a0*: $a \neq 0$
and *a1*: $\neg$ *a dvd 1*
shows *rel-mset* (*ddvd*) (*mset fs*) (*mset gs*)
proof−
from *a1* **have** $a \neq 1$ **by** *auto*
with *a0 fs gs* **have** *mset-factors* (*mset fs*) *a mset-factors* (*mset gs*) *a* **by** (*unfold factors-as-mset-factors*)
from *mset-factors-unique*[*OF this*] **show** *?thesis*.
qed

lemma *factorial-monoid*: *factorial-monoid* (*mk-monoid* :: *'a monoid*)
by (*unfold-locales*; *auto simp add*: *factors-exist factors-unique*)

end

lemma (**in** *idom*) *factorial-monoid-imp-ufd*:
assumes *factorial-monoid* (*mk-monoid* :: *'a monoid*)
shows *class.ufd* ((∗) :: *'a* ⇒ -) *1* (+) *0* (−) *uminus*
proof (*unfold-locales*)
interpret *factorial-monoid mk-monoid* :: *'a monoid* **by** (*fact assms*)
{
fix *x* **assume** *x*: $x \neq 0$ $\neg$ *x dvd 1*
note ∗ = *factors-exist*[*simplified*, *OF this*]
with *x* **show** $\exists$*F*. *mset-factors F x* **by** (*subst*(*asm*) *factors-as-mset-factors*, *auto*)
}
fix *x F G* **assume** *FG*: *mset-factors F x mset-factors G x*
with *mset-factors-imp-not-is-unit* **have** *x1*: $\neg$ *x dvd 1* **by** *auto*

from *FG*(*1*) **have** *x0*: $x \neq 0$ **by** (*rule mset-factors-imp-nonzero*)
obtain *fs gs* **where** *fsgs*: *F* = *mset fs G* = *mset gs* **using** *ex-mset* **by** *metis*
note *FG* = *FG*[*unfolded this*]
then have *0*: $0 \notin$ *set fs* $0 \notin$ *set gs* **by** (*auto elim*!: *mset-factorsE*)
from *x1* **have** $x \neq 1$ **by** *auto*
note *FG*[*folded factors-as-mset-factors*[*OF x0 this*]]
from *factors-unique*[*OF this*, *simplified*, *OF x0 x1*, *folded fsgs*] *0*
show *rel-mset* (*ddvd*) *F G* **by** *auto*
qed

3.4 Preservation of Irreducibility

locale *comm-semiring-1-hom* = *comm-monoid-mult-hom hom* + *zero-hom hom*
for *hom* :: *'a* :: *comm-semiring-1* ⇒ *'b* :: *comm-semiring-1*

locale *irreducibility-hom* = *comm-semiring-1-hom* +
assumes *irreducible-imp-irreducible-hom*: *irreducible a* ⟹ *irreducible* (*hom a*)
begin
lemma *hom-mset-factors*:
assumes *F*: *mset-factors F p*
shows *mset-factors* (*image-mset hom F*) (*hom p*)
proof (*unfold mset-factors-def*, *intro conjI allI impI*)
from *F* **show** *hom p* = *prod-mset* (*image-mset hom F*) *image-mset hom F* ≠ {#} **by** (*auto simp*: *hom-distribs*)
fix *f'* **assume** *f'* ∈# *image-mset hom F*
then obtain *f* **where** *f*: *f* ∈# *F* **and** *f'f*: *f'* = *hom f* **by** *auto*
with *F irreducible-imp-irreducible-hom* **show** *irreducible f'* **unfolding** *f'f* **by** *auto*
qed
end

locale *unit-preserving-hom* = *comm-semiring-1-hom* +
assumes *is-unit-hom-if*: ⋀*x*. *hom x dvd 1* ⟹ *x dvd 1*
begin
lemma *is-unit-hom-iff*[*simp*]: *hom x dvd 1* ⟷ *x dvd 1* **using** *is-unit-hom-if hom-dvd* **by** *force*

lemma *irreducible-hom-imp-irreducible*:
assumes *irr*: *irreducible* (*hom a*) **shows** *irreducible a*
proof (*intro irreducibleI*)
from *irr* **show** $a \neq 0$ **by** *auto*
from *irr* **show** ¬ *a dvd 1* **by** (*auto dest*: *irreducible-not-unit*)
fix *b c* **assume** $a = b * c$
then have *hom a* = *hom b* ∗ *hom c* **by** (*simp add*: *hom-distribs*)
with *irr* **have** *hom b dvd 1* ∨ *hom c dvd 1* **by** (*auto dest*: *irreducibleD*)
then show *b dvd 1* ∨ *c dvd 1* **by** *simp*
qed
end

```
locale factor-preserving-hom = unit-preserving-hom + irreducibility-hom
begin
  lemma irreducible-hom[simp]: irreducible (hom a) ⟷ irreducible a
    using irreducible-hom-imp-irreducible irreducible-imp-irreducible-hom by metis
end

lemma factor-preserving-hom-comp:
  assumes f: factor-preserving-hom f and g: factor-preserving-hom g
  shows factor-preserving-hom (f o g)
proof−
  interpret f: factor-preserving-hom f by (rule f)
  interpret g: factor-preserving-hom g by (rule g)
  show ?thesis by (unfold-locales, auto simp: hom-distribs)
qed

context comm-semiring-isom begin
  sublocale unit-preserving-hom by (unfold-locales, auto)
  sublocale factor-preserving-hom
  proof (standard)
    fix a :: 'a
    assume irreducible a
    note a = this[unfolded irreducible-def]
    show irreducible (hom a)
    proof (rule ccontr)
      assume ¬ irreducible (hom a)
      from this[unfolded Factorial-Ring.irreducible-def,simplified] a
       obtain hb hc where eq: hom a = hb ∗ hc and nu: ¬ hb dvd 1 ¬ hc dvd 1
by auto
      from bij obtain b where hb: hb = hom b by (elim bij-pointE)
      from bij obtain c where hc: hc = hom c by (elim bij-pointE)
      from eq[unfolded hb hc, folded hom-mult] have a = b ∗ c by auto
      with nu hb hc have a = b ∗ c ¬ b dvd 1 ¬ c dvd 1 by auto
      with a show False by auto
    qed
  qed
end
```

3.4.1 Back to divisibility

```
lemma(in comm-semiring-1) mset-factors-mult:
  assumes F: mset-factors F a
      and G: mset-factors G b
  shows mset-factors (F+G) (a∗b)
proof(intro mset-factorsI)
  fix f assume f ∈# F + G
  then consider f ∈# F | f ∈# G by auto
  then show irreducible f by(cases, insert F G, auto)
qed (insert F G, auto)
```

lemma(**in** *ufd*) *dvd-imp-subset-factors*:
  assumes *ab*: *a dvd b*
    and *F*: *mset-factors F a*
    and *G*: *mset-factors G b*
  shows $\exists$ *G'*. *G'* $\subseteq$# *G* $\wedge$ *rel-mset* (*ddvd*) *F G'*
proof−
  from *F G* **have** *a0*: *a* $\neq$ *0* **and** *b0*: *b* $\neq$ *0* **by** (*simp-all add*: *mset-factors-imp-nonzero*)
  from *ab* **obtain** *c* **where** *c*: *b* = *a* $*$ *c* **by** (*elim dvdE*, *auto*)
  with *b0* **have** *c0*: *c* $\neq$ *0* **by** *auto*
  show *?thesis*
  proof(*cases c dvd 1*)
    case *True*
    show *?thesis*
    proof(*cases F*)
      case *empty* **with** *F* **show** *?thesis* **by** *auto*
    next
      case (*add f F'*)
        with *F*
        have *a*: *f* $*$ *prod-mset F'* = *a*
         and *F'*: $\bigwedge$*f*. *f* $\in$# *F'* $\Longrightarrow$ *irreducible f*
         and *irrf*: *irreducible f* **by** *auto*
            from *irrf* **have** *f0*: *f* $\neq$ *0* **and** *f1*: $\neg$*f dvd 1* **by** (*auto dest*: *irreducible-not-unit*)
        from *a c* **have** (*f* $*$ *c*) $*$ *prod-mset F'* = *b* **by** (*auto simp*: *ac-simps*)
        moreover **{**
       have *irreducible* (*f* $*$ *c*) **using** *True irrf* **by** (*subst irreducible-mult-unit-right*)
           with *F' irrf* **have** $\bigwedge$*f'*. *f'* $\in$# *F'* + {#*f* $*$ *c*#} $\Longrightarrow$ *irreducible f'* **by** *auto*
        }
       ultimately have *mset-factors* (*F'* + {#*f* $*$ *c*#}) *b* **by** (*intro mset-factorsI*, *auto*)
        from *mset-factors-unique*[*OF this G*]
        have *F'G*: *rel-mset* (*ddvd*) (*F'* + {#*f* $*$ *c*#}) *G*.
        from *True add* **have** *FF'*: *rel-mset* (*ddvd*) *F* (*F'* + {#*f* $*$ *c*#})
          by (*auto simp add*: *multiset.rel-refl intro*!: *rel-mset-Plus*)
        have *rel-mset* (*ddvd*) *F G*
          apply(*rule transpD*[*OF multiset.rel-transp*[*OF transpI*] *FF' F'G*])
          using *ddvd-trans*.
        then show *?thesis* **by** *auto*
    qed
  next
    case *False*
      from *mset-factors-exist*[*OF c0 this*] **obtain** *H* **where** *H*: *mset-factors H c* **by** *auto*
    from *c mset-factors-mult*[*OF F H*] **have** *mset-factors* (*F* + *H*) *b* **by** *auto*
    note *mset-factors-unique*[*OF this G*]
    from *rel-mset-split*[*OF this*] **obtain** *G1 G2*
      where *G* = *G1* + *G2* *rel-mset* (*ddvd*) *F G1* *rel-mset* (*ddvd*) *H G2* **by** *auto*
    then show *?thesis* **by** (*intro exI*[*of - G1*], *auto*)

```
  qed
qed

lemma(in idom) irreducible-factor-singleton:
  assumes a: irreducible a
  shows mset-factors F a ⟷ F = {#a#}
proof(cases F)
  case empty with mset-factorsD show ?thesis by auto
next
  case (add f F')
  show ?thesis
  proof
    assume F: mset-factors F a
    from add mset-factorsD[OF F] have *: a = f * prod-mset F' by auto
    then have fa: f dvd a by auto
    from * a have f0: f ≠ 0 by auto
    from add have f ∈# F by auto
    with F have f: irreducible f by auto
    from add have F' ⊆# F by auto
    then have unitemp: prod-mset F' dvd 1 ⟹ F' = {#}
    proof(induct F')
      case empty then show ?case by auto
    next
      case (add f F')
        from add have f ∈# F by (simp add: mset-subset-eq-insertD)
        with F irreducible-not-unit have ¬ f dvd 1 by auto
        then have ¬ (prod-mset F' * f) dvd 1 by simp
        with add show ?case by auto
    qed
    show F = {#a#}
    proof(cases a dvd f)
      case True
        then obtain r where f = a * r by (elim dvdE, auto)
        with * have f = (r * prod-mset F') * f by (auto simp: ac-simps)
        with f0 have r * prod-mset F' = 1 by auto
        then have prod-mset F' dvd 1 by (metis dvd-triv-right)
        with unitemp * add show ?thesis by auto
    next
      case False with fa a f show ?thesis by (auto simp: irreducible-altdef)
    qed
  qed (insert a, auto)
qed


lemma(in ufd) irreducible-dvd-imp-factor:
  assumes ab: a dvd b
      and a: irreducible a
      and G: mset-factors G b
  shows ∃ g ∈# G. a ddvd g
```

```
proof−
  from a have mset-factors {#a#} a by auto
  from dvd-imp-subset-factors[OF ab this G]
  obtain G′ where G′G: G′ ⊆# G and rel: rel-mset (ddvd) {#a#} G′ by auto
  with rel-mset-size size-1-singleton-mset size-single
  obtain g where gG′: G′ = {#g#} by fastforce
  from rel[unfolded this rel-mset-def]
  have a ddvd g by auto
  with gG′ G′G show ?thesis by auto
qed

lemma(in idom) prod-mset-remove-units:
  prod-mset F ddvd prod-mset {# f ∈# F. ¬f dvd 1 #}
proof(induct F)
  case (add f F) then show ?case by (cases f = 0, auto)
qed auto

lemma(in comm-semiring-1) mset-factors-imp-dvd:
  assumes mset-factors F x and f ∈# F shows f dvd x
  using assms by (simp add: dvd-prod-mset mset-factors-def)

lemma(in ufd) prime-elem-iff-irreducible[iff]:
  prime-elem x ⟷ irreducible x
proof (intro iffI, fact prime-elem-imp-irreducible, rule prime-elemI)
  assume r: irreducible x
  then show x0: x ≠ 0 and x1: ¬ x dvd 1 by (auto dest: irreducible-not-unit)
  from irreducible-factor-singleton[OF r]
  have ∗: mset-factors {#x#} x by auto
  fix a b
  assume x dvd a ∗ b
  then obtain c where abxc: a ∗ b = x ∗ c by (elim dvdE, auto)
  show x dvd a ∨ x dvd b
  proof(cases c = 0 ∨ a = 0 ∨ b = 0)
    case True with abxc show ?thesis by auto
  next
    case False
    then have a0: a ≠ 0 and b0: b ≠ 0 and c0: c ≠ 0 by auto
    from x0 c0 have xc0: x ∗ c ≠ 0 by auto
    from x1 have xc1: ¬ x ∗ c dvd 1 by auto
    show ?thesis
    proof (cases a dvd 1 ∨ b dvd 1)
      case False
      then have a1: ¬ a dvd 1 and b1: ¬ b dvd 1 by auto
      from mset-factors-exist[OF a0 a1]
      obtain F where Fa: mset-factors F a by auto
      then have F0: F ≠ {#} by auto
      from mset-factors-exist[OF b0 b1]
      obtain G where Gb: mset-factors G b by auto
      then have G0: G ≠ {#} by auto
```

from *mset-factors-mult*[*OF Fa Gb*]
have *FGxc*: *mset-factors* (*F* + *G*) (*x* ∗ *c*) **by** (*simp add*: *abxc*)
show *?thesis*
proof (*cases c dvd 1*)
case *True*
from *r irreducible-mult-unit-right*[*OF this*] **have** *irreducible* (*x*∗*c*) **by** *simp*
note *irreducible-factor-singleton*[*OF this*] *FGxc*
with *F0 G0* **have** *False* **by** (*cases F*; *cases G*; *auto*)
then show *?thesis* **by** *auto*
next
case *False*
from *mset-factors-exist*[*OF c0 this*] **obtain** *H* **where** *mset-factors H c* **by** *auto*
with ∗ **have** *xHxc*: *mset-factors* (*add-mset x H*) (*x* ∗ *c*) **by** *force*
note *rel* = *mset-factors-unique*[*OF this FGxc*]
obtain *hs* **where** *mset hs* = *H* **using** *ex-mset* **by** *auto*
then have *mset* (*x*#*hs*) = *add-mset x H* **by** *auto*
from *rel-mset-free*[*OF rel this*]
obtain *jjs* **where** *jjsGH*: *mset jjs* = *F* + *G* **and** *rel*: *list-all2* (*ddvd*) (*x* # *hs*) *jjs* **by** *auto*
then obtain *j js* **where** *jjs*: *jjs* = *j* # *js* **by** (*cases jjs*, *auto*)
with *rel* **have** *xj*: *x ddvd j* **by** *auto*
from *jjs jjsGH* **have** *j*: *j* ∈ *set-mset* (*F* + *G*) **by** (*intro union-single-eq-member*, *auto*)
from *j* **consider** *j* ∈# *F* | *j* ∈# *G* **by** *auto*
then show *?thesis*
proof(*cases*)
case *1*
with *Fa* **have** *j dvd a* **by** (*auto intro*: *mset-factors-imp-dvd*)
with *xj dvd-trans* **have** *x dvd a* **by** *auto*
then show *?thesis* **by** *auto*
next
case *2*
with *Gb* **have** *j dvd b* **by** (*auto intro*: *mset-factors-imp-dvd*)
with *xj dvd-trans* **have** *x dvd b* **by** *auto*
then show *?thesis* **by** *auto*
qed
qed
next
case *True*
then consider *a dvd 1* | *b dvd 1* **by** *auto*
then show *?thesis*
proof(*cases*)
case *1*
then obtain *d* **where** *ad*: *a* ∗ *d* = *1* **by** (*elim dvdE*, *auto*)
from *abxc* **have** *x* ∗ (*c* ∗ *d*) = *a* ∗ *b* ∗ *d* **by** (*auto simp*: *ac-simps*)
also have ... = *a* ∗ *d* ∗ *b* **by** (*auto simp*: *ac-simps*)
finally have *x dvd b* **by** (*intro dvdI*, *auto simp*: *ad*)
then show *?thesis* **by** *auto*

```
      next
        case 2
        then obtain d where bd: b * d = 1 by (elim dvdE, auto)
        from abxc have x * (c * d) = a * b * d by (auto simp: ac-simps)
        also have ... = (b * d) * a by (auto simp: ac-simps)
        finally have x dvd a by (intro dvdI, auto simp:bd)
        then show ?thesis by auto
      qed
    qed
  qed
qed
```

3.5 Results for GCDs etc.

```
lemma prod-list-remove1: (x :: 'b :: comm-monoid-mult) ∈ set xs ⟹ prod-list
(remove1 x xs) * x = prod-list xs
  by (induct xs, auto simp: ac-simps)


class comm-monoid-gcd = gcd + comm-semiring-1 +
  assumes gcd-dvd1[iff]: gcd a b dvd a
      and gcd-dvd2[iff]: gcd a b dvd b
      and gcd-greatest: c dvd a ⟹ c dvd b ⟹ c dvd gcd a b
begin

  lemma gcd-0-0[simp]: gcd 0 0 = 0
    using gcd-greatest[OF dvd-0-right dvd-0-right, of 0] by auto

  lemma gcd-zero-iff[simp]: gcd a b = 0 ⟷ a = 0 ∧ b = 0
  proof
    assume gcd a b = 0
    from gcd-dvd1[of a b, unfolded this] gcd-dvd2[of a b, unfolded this]
    show a = 0 ∧ b = 0 by auto
  qed auto

  lemma gcd-zero-iff'[simp]: 0 = gcd a b ⟷ a = 0 ∧ b = 0
    using gcd-zero-iff by metis

  lemma dvd-gcd-0-iff[simp]:
    shows x dvd gcd 0 a ⟷ x dvd a (is ?g1)
      and x dvd gcd a 0 ⟷ x dvd a (is ?g2)
  proof−
    have a dvd gcd a 0 a dvd gcd 0 a by (auto intro: gcd-greatest)
    with dvd-refl show ?g1 ?g2 by (auto dest: dvd-trans)
  qed

  lemma gcd-dvd-1[simp]: gcd a b dvd 1 ⟷ coprime a b
    using dvd-trans[OF gcd-greatest[of - a b], of - 1]
    by (cases a = 0 ∧ b = 0) (auto intro!: coprimeI elim: coprimeE)
```

lemma *dvd-imp-gcd-dvd-gcd*: b *dvd* $c \Longrightarrow$ *gcd* a b *dvd* *gcd* a c
  by (*meson gcd-dvd1 gcd-dvd2 gcd-greatest dvd-trans*)

definition *listgcd* :: *'a list* $\Rightarrow$ *'a* **where**
  listgcd xs = *foldr gcd xs 0*

lemma *listgcd-simps*[*simp*]: *listgcd* [] = *0* *listgcd* (x # *xs*) = *gcd* x (*listgcd xs*)
  by (*auto simp*: *listgcd-def*)

lemma *listgcd*: $x \in$ *set xs* $\Longrightarrow$ *listgcd xs dvd* x
proof (*induct xs*)
  case (*Cons y ys*)
  show *?case*
  proof (*cases* $x = y$)
    case *False*
    with *Cons* **have** *dvd*: *listgcd ys dvd* x **by** *auto*
    thus *?thesis* **unfolding** *listgcd-simps* **using** *dvd-trans* **by** *blast*
  next
    case *True*
    thus *?thesis* **unfolding** *listgcd-simps* **using** *dvd-trans* **by** *blast*
  qed
qed *simp*

lemma *listgcd-greatest*: ($\bigwedge x$. $x \in$ *set xs* $\Longrightarrow$ y *dvd* x) $\Longrightarrow$ y *dvd listgcd xs*
  by (*induct xs arbitrary*:y, *auto intro*: *gcd-greatest*)

end

context *Rings.dvd* **begin**

definition *is-gcd* x a b $\equiv$ x *dvd* a $\wedge$ x *dvd* b $\wedge$ ($\forall y$. y *dvd* a $\longrightarrow$ y *dvd* b $\longrightarrow$ y *dvd* x)

definition *some-gcd* a b $\equiv$ *SOME* x. *is-gcd* x a b

lemma *is-gcdI*[*intro*!]:
  assumes x *dvd* a x *dvd* b $\bigwedge y$. y *dvd* $a \Longrightarrow y$ *dvd* $b \Longrightarrow y$ *dvd* x
  shows *is-gcd* x a b **by** (*insert assms*, *auto simp*: *is-gcd-def*)

lemma *is-gcdE*[*elim*!]:
  assumes *is-gcd* x a b
    and x *dvd* $a \Longrightarrow x$ *dvd* $b \Longrightarrow$ ($\bigwedge y$. y *dvd* $a \Longrightarrow y$ *dvd* $b \Longrightarrow y$ *dvd* x) $\Longrightarrow$ *thesis*
  shows *thesis* **by** (*insert assms*, *auto simp*: *is-gcd-def*)

lemma *is-gcd-some-gcdI*:
  assumes $\exists x$. *is-gcd* x a b **shows** *is-gcd* (*some-gcd* a b) a b

by (*unfold some-gcd-def*, *rule someI-ex*[*OF assms*])

end

context *comm-semiring-1* **begin**

lemma *some-gcd-0*[*intro*!]: *is-gcd* (*some-gcd a 0*) *a 0 is-gcd* (*some-gcd 0 b*) *0 b*
by (*auto intro*!: *is-gcd-some-gcdI intro*: *exI*[*of - a*] *exI*[*of - b*])

lemma *some-gcd-0-dvd*[*intro*!]:
some-gcd a 0 dvd a some-gcd 0 b dvd b **using** *some-gcd-0* **by** *auto*

lemma *dvd-some-gcd-0*[*intro*!]:
a dvd some-gcd a 0 b dvd some-gcd 0 b **using** *some-gcd-0*[*of a*] *some-gcd-0*[*of b*] **by** *auto*

end

context *idom* **begin**

lemma *is-gcd-connect*:
assumes $a \neq 0$ $b \neq 0$ **shows** *isgcd mk-monoid x a b* $\longleftrightarrow$ *is-gcd x a b*
using *assms* **by** (*force simp*: *isgcd-def*)

lemma *some-gcd-connect*:
assumes $a \neq 0$ **and** $b \neq 0$ **shows** *somegcd mk-monoid a b* = *some-gcd a b*
using *assms* **by** (*auto intro*!: *arg-cong*[*of - - Eps*] *simp*: *is-gcd-connect some-gcd-def somegcd-def*)
end

context *comm-monoid-gcd*
begin
lemma *is-gcd-gcd*: *is-gcd* (*gcd a b*) *a b* **using** *gcd-greatest* **by** *auto*
lemma *is-gcd-some-gcd*: *is-gcd* (*some-gcd a b*) *a b* **by** (*insert is-gcd-gcd*, *auto intro*!: *is-gcd-some-gcdI*)
lemma *gcd-dvd-some-gcd*: *gcd a b dvd some-gcd a b* **using** *is-gcd-some-gcd* **by** *auto*
lemma *some-gcd-dvd-gcd*: *some-gcd a b dvd gcd a b* **using** *is-gcd-some-gcd* **by** (*auto intro*: *gcd-greatest*)
lemma *some-gcd-ddvd-gcd*: *some-gcd a b ddvd gcd a b* **by** (*auto intro*: *gcd-dvd-some-gcd some-gcd-dvd-gcd*)
lemma *some-gcd-dvd*: *some-gcd a b dvd d* $\longleftrightarrow$ *gcd a b dvd d d dvd some-gcd a b* $\longleftrightarrow$ *d dvd gcd a b*
using *some-gcd-ddvd-gcd*[*of a b*] **by** (*auto dest*:*dvd-trans*)

end

class *idom-gcd* = *comm-monoid-gcd* + *idom*
begin

interpretation *raw*: *comm-monoid-cancel mk-monoid* :: *'a monoid*
by (*unfold-locales*, *auto intro*: *mult-commute mult-assoc*)

interpretation *raw*: *gcd-condition-monoid mk-monoid* :: *'a monoid*
by (*unfold-locales*, *auto simp*: *is-gcd-connect intro*!: *exI*[*of - gcd - -*] *dest*: *gcd-greatest*)

lemma *gcd-mult-ddvd*:
d ∗ *gcd a b ddvd gcd* (*d* ∗ *a*) (*d* ∗ *b*)
proof (*cases d* = *0*)
case *True* **then show** *?thesis* **by** *auto*
next
case *d0*: *False*
show *?thesis*
proof (*cases a* = *0* ∨ *b* = *0*)
case *False*
note *some-gcd-ddvd-gcd*[*of a b*]
with *d0* **have** *d* ∗ *gcd a b ddvd d* ∗ *some-gcd a b* **by** *auto*
also have *d* ∗ *some-gcd a b ddvd some-gcd* (*d* ∗ *a*) (*d* ∗ *b*)
using *False d0 raw.gcd-mult* **by** (*simp add*: *some-gcd-connect*)
also note *some-gcd-ddvd-gcd*
finally show *?thesis*.
next
case *True*
with *d0* **show** *?thesis*
apply (*elim disjE*)
apply (*rule ddvd-trans*[*of - d* ∗ *b*]; *force*)
apply (*rule ddvd-trans*[*of - d* ∗ *a*]; *force*)
done
qed
qed

lemma *gcd-greatest-mult*: **assumes** *cad*: *c dvd a* ∗ *d* **and** *cbd*: *c dvd b* ∗ *d*
shows *c dvd gcd a b* ∗ *d*
proof−
from *gcd-greatest*[*OF assms*] **have** *c*: *c dvd gcd* (*d* ∗ *a*) (*d* ∗ *b*) **by** (*auto simp*: *ac-simps*)
note *gcd-mult-ddvd*[*of d a b*]
then have *gcd* (*d* ∗ *a*) (*d* ∗ *b*) *dvd gcd a b* ∗ *d* **by** (*auto simp*: *ac-simps*)
from *dvd-trans*[*OF c this*] **show** *?thesis* **.**
qed

lemma *listgcd-greatest-mult*: ($\bigwedge$ *x* :: *'a*. *x* ∈ *set xs* ⟹ *y dvd x* ∗ *z*) ⟹ *y dvd listgcd xs* ∗ *z*
proof (*induct xs*)
case (*Cons x xs*)
from *Cons* **have** *y dvd x* ∗ *z y dvd listgcd xs* ∗ *z* **by** *auto*
thus *?case* **unfolding** *listgcd-simps* **by** (*rule gcd-greatest-mult*)

```
qed (simp)

lemma dvd-factor-mult-gcd:
  assumes dvd: k dvd p * q k dvd p * r
    and q0: q ≠ 0 and r0: r ≠ 0
  shows k dvd p * gcd q r
proof -
  from dvd gcd-greatest[of k p * q p * r]
  have k dvd gcd (p * q) (p * r) by simp
  also from gcd-mult-ddvd[of p q r]
  have ... dvd (p * gcd q r) by auto
  finally show ?thesis .
qed

lemma coprime-mult-cross-dvd:
  assumes coprime: coprime p q and eq: p' * p = q' * q
  shows p dvd q' (is ?g1) and q dvd p' (is ?g2)
proof (atomize(full), cases p = 0 ∨ q = 0)
  case True
  then show ?g1 ∧ ?g2
  proof
    assume p0: p = 0 with coprime have q dvd 1 by auto
    with eq p0 show ?thesis by auto
  next
    assume q0: q = 0 with coprime have p dvd 1 by auto
    with eq q0 show ?thesis by auto
  qed
next
  case False
  {
    fix p q r p' q' :: 'a
    assume cop: coprime p q and eq: p' * p = q' * q and p: p ≠ 0 and q: q ≠ 0
      and r: r dvd p r dvd q
    let ?gcd = gcd q p
    from eq have p' * p dvd q' * q by auto
    hence d1: p dvd q' * q by (rule dvd-mult-right)
    have d2: p dvd q' * p by auto
    from dvd-factor-mult-gcd[OF d1 d2 q p] have 1: p dvd q' * ?gcd .
    from q p have 2: ?gcd dvd q by auto
    from q p have 3: ?gcd dvd p by auto
    from cop[unfolded coprime-def', rule-format, OF 3 2] have ?gcd dvd 1 .
    from 1 dvd-mult-unit-iff[OF this] have p dvd q' by auto
  } note main = this
   from main[OF coprime eq,of 1] False coprime coprime-commute main[OF -
eq[symmetric], of 1]
  show ?g1 ∧ ?g2 by auto
qed

end
```

```
subclass (in ring-gcd) idom-gcd by (unfold-locales, auto)

lemma coprime-rewrites: comm-monoid-mult.coprime ((∗)) 1 = coprime
  apply (intro ext)
  apply (subst comm-monoid-mult.coprime-def')
  apply (unfold-locales)
  apply (unfold dvd-rewrites)
  apply (fold coprime-def') ..


locale gcd-condition =
  fixes ty :: 'a :: idom itself
  assumes gcd-exists: ⋀a b :: 'a. ∃ x. is-gcd x a b
begin
  sublocale idom-gcd (∗) 1 :: 'a (+) 0 (−) uminus some-gcd
    rewrites dvd.dvd ((∗)) = (dvd)
        and comm-monoid-mult.coprime ((∗) ) 1 = Unique-Factorization.coprime
  proof−
   have is-gcd (some-gcd a b) a b for a b :: 'a by (intro is-gcd-some-gcdI gcd-exists)
    from this[unfolded is-gcd-def]
   show class.idom-gcd (∗) (1 :: 'a) (+) 0 (−) uminus some-gcd by (unfold-locales,
auto simp: dvd-rewrites)
  qed (simp-all add: dvd-rewrites coprime-rewrites)
end

instance semiring-gcd ⊆ comm-monoid-gcd by (intro-classes, auto)

lemma listgcd-connect: listgcd = gcd-list
proof (intro ext)
  fix xs :: 'a list
  show listgcd xs = gcd-list xs by(induct xs, auto)
qed

interpretation some-gcd: gcd-condition TYPE('a::ufd)
proof(unfold-locales, intro exI)
  interpret factorial-monoid mk-monoid :: 'a monoid by (fact factorial-monoid)
  note d = dvd.dvd-def some-gcd-def carrier-0
  fix a b :: 'a
  show is-gcd (some-gcd a b) a b
  proof (cases a = 0 ∨ b = 0)
    case True
    thus ?thesis using some-gcd-0 by auto
  next
    case False
    with gcdof-exists[of a b]
   show ?thesis by (auto intro!: is-gcd-some-gcdI simp add: is-gcd-connect some-gcd-connect)
  qed
qed
```

lemma *some-gcd-listgcd-dvd-listgcd*: *some-gcd.listgcd xs dvd listgcd xs*
  by (*induct xs*, *auto simp*:*some-gcd-dvd intro*:*dvd-imp-gcd-dvd-gcd*)

lemma *listgcd-dvd-some-gcd-listgcd*: *listgcd xs dvd some-gcd.listgcd xs*
  by (*induct xs*, *auto simp*:*some-gcd-dvd intro*:*dvd-imp-gcd-dvd-gcd*)

context *factorial-ring-gcd* **begin**

Do not declare the following as subclass, to avoid conflict in *field* $\subseteq$ *gcd-condition* vs. *factorial-ring-gcd* $\subseteq$ *gcd-condition*.

sublocale *as-ufd*: *ufd*
proof(*unfold-locales*, *goal-cases*)
  case (*1 x*)
  from *prime-factorization-exists*[*OF* ‹$x \neq 0$›]
  obtain *F* **where** *f*: $\bigwedge f.\ f \in\# F \Longrightarrow$ *prime-elem f*
      and *Fx*: *normalize* (*prod-mset F*) = *normalize x* **by** *auto*
  from *associatedE2*[*OF Fx*] **obtain** *u* **where** *u*: *is-unit u x* = *u* $*$ *prod-mset F*
    by *blast*
  from ‹¬ *is-unit x*› *Fx* **have** $F \neq \{\#\}$ **by** *auto*
  then obtain *g G* **where** *F*: *F* = *add-mset g G* **by** (*cases F*, *auto*)
  then have $g \in\# F$ **by** *auto*
  with *f*[*OF this*]*prime-elem-iff-irreducible*
    irreducible-mult-unit-left[*OF unit-factor-is-unit*[*OF* ‹$x \neq 0$›]]
  have *g*: *irreducible* ($u * g$) **using** *u*(*1*)
    by (*subst irreducible-mult-unit-left*) *simp-all*
  show *?case*
  proof (*intro exI conjI mset-factorsI*)
    show *prod-mset* (*add-mset* ($u * g$) *G*) = *x*
      using ‹$x \neq 0$› **by** (*simp add*: *F ac-simps u*)
    fix *f* **assume** $f \in\#$ *add-mset* ($u * g$) *G*
    with *f*[*unfolded F*] *g prime-elem-iff-irreducible*
    show *irreducible f* **by** *auto*
  qed *auto*
next
  case (*2 x F G*)
  note *transpD*[*OF multiset.rel-transp*[*OF ddvd-transp*],*trans*]
  obtain *fs* **where** *F*: *F* = *mset fs* **by** (*metis ex-mset*)
  have *list-all2* (*ddvd*) *fs* (*map normalize fs*) **by** (*intro list-all2-all-nthI*, *auto*)
  then have *FH*: *rel-mset* (*ddvd*) *F* (*image-mset normalize F*) **by** (*unfold rel-mset-def F*, *force*)
  also
  have *FG*: *image-mset normalize F* = *image-mset normalize G*
  proof (*intro prime-factorization-unique*$''$)
    from *2* **have** *xF*: *x* = *prod-mset F* **and** *xG*: *x* = *prod-mset G* **by** *auto*
    from *xF* **have** *normalize x* = *normalize* (*prod-mset* (*image-mset normalize F*))
      by (*simp add*: *normalize-prod-mset-normalize*)
    with *xG* **have** *nFG*: ... = *normalize* (*prod-mset* (*image-mset normalize G*))
      by (*simp-all add*: *normalize-prod-mset-normalize*)

then show *normalize* ($\prod$ *i*∈#*image-mset normalize F*. *i*) =
normalize ($\prod$ *i*∈#*image-mset normalize G*. *i*) **by** *auto*
next
from *2 prime-elem-iff-irreducible* **have** *f* ∈# *F* ⟹ *prime-elem f g* ∈# *G* ⟹ *prime-elem g* **for** *f g*
by (*auto intro*: *prime-elemI*)
then show *Multiset.Ball* (*image-mset normalize F*) *prime*
Multiset.Ball (*image-mset normalize G*) *prime* **by** *auto*
qed
also
obtain *gs* **where** *G*: *G* = *mset gs* **by** (*metis ex-mset*)
have *list-all2* ((*ddvd*)$^{-1-1}$) *gs* (*map normalize gs*) **by** (*intro list-all2-all-nthI*, *auto*)
then have *rel-mset* (*ddvd*) (*image-mset normalize G*) *G*
by (*subst multiset.rel-flip*[*symmetric*], *unfold rel-mset-def G*, *force*)
finally show *?case*.
qed

end

instance *int* :: *ufd* **by** (*intro class.ufd.of-class.intro as-ufd.ufd-axioms*)
instance *int* :: *idom-gcd* **by** (*intro-classes*, *auto*)

instance *field* ⊆ *ufd* **by** (*intro-classes*, *auto simp*: *dvd-field-iff*)

end

4 Unique Factorization Domain for Polynomials

In this theory we prove that the polynomials over a unique factorization domain (UFD) form a UFD.

theory *Unique-Factorization-Poly*
imports
Unique-Factorization
Polynomial-Factorization.Missing-Polynomial-Factorial
Subresultants.More-Homomorphisms
HOL−Computational-Algebra.Field-as-Ring
begin

hide-const (**open**) *module.smult*
hide-const (**open**) *Divisibility.irreducible*

instantiation *fract* :: (*idom*) {*normalization-euclidean-semiring*, *euclidean-ring*}
begin

definition [*simp*]: *normalize-fract* ≡ (*normalize-field* :: *'a fract* ⇒ -)
definition [*simp*]: *unit-factor-fract* = (*unit-factor-field* :: *'a fract* ⇒ -)
definition [*simp*]: *euclidean-size-fract* = (*euclidean-size-field* :: *'a fract* ⇒ -)

definition [*simp*]: *modulo-fract* = (*mod-field* :: *'a fract* ⇒ -)

instance by *standard* (*simp-all add*: *dvd-field-iff divide-simps*)

end

instantiation *fract* :: (*idom*) *euclidean-ring-gcd*
begin

definition *gcd-fract* :: *'a fract* ⇒ *'a fract* ⇒ *'a fract* **where**
gcd-fract ≡ *Euclidean-Algorithm.gcd*
definition *lcm-fract* :: *'a fract* ⇒ *'a fract* ⇒ *'a fract* **where**
lcm-fract ≡ *Euclidean-Algorithm.lcm*
definition *Gcd-fract* :: *'a fract set* ⇒ *'a fract* **where**
Gcd-fract ≡ *Euclidean-Algorithm.Gcd*
definition *Lcm-fract* :: *'a fract set* ⇒ *'a fract* **where**
Lcm-fract ≡ *Euclidean-Algorithm.Lcm*

instance
by (*standard*, *simp-all add*: *gcd-fract-def lcm-fract-def Gcd-fract-def Lcm-fract-def*)

end

instantiation *fract* :: (*idom*) *unique-euclidean-ring*
begin

definition [*simp*]: *division-segment-fract* (*x* :: *'a fract*) = (*1* :: *'a fract*)

instance by *standard* (*auto split*: *if-splits*)
end

instance *fract* :: (*idom*) *field-gcd* **by** *standard auto*

definition *divides-ff* :: *'a::idom fract* ⇒ *'a fract* ⇒ *bool*
where *divides-ff x y* ≡ ∃ *r*. *y* = *x* ∗ *to-fract r*

lemma *ff-list-pairs*:
∃ *xs*. *X* = *map* (λ (*x*,*y*). *Fraction-Field.Fract x y*) *xs* ∧ *0* ∉ *snd* ‘ *set xs*
proof (*induct X*)
case (*Cons a X*)
from *Cons*(*1*) **obtain** *xs* **where** *X*: *X* = *map* (λ (*x*,*y*). *Fraction-Field.Fract x y*) *xs* **and** *xs*: *0* ∉ *snd* ‘ *set xs*
by *auto*
obtain *x y* **where** *a*: *a* = *Fraction-Field.Fract x y* **and** *y*: *y* ≠ *0* **by** (*cases a*, *auto*)
show *?case* **unfolding** *X a* **using** *xs y*
by (*intro exI*[*of* - (*x*,*y*) # *xs*], *auto*)

qed *auto*

lemma *divides-ff-to-fract*[*simp*]: *divides-ff* (*to-fract x*) (*to-fract y*) ⟷ *x dvd y*
  unfolding *divides-ff-def dvd-def*
  by (*simp add*: *to-fract-def eq-fract*(*1*) *mult.commute*)

lemma
  shows *divides-ff-mult-cancel-left*[*simp*]: *divides-ff* (*z* ∗ *x*) (*z* ∗ *y*) ⟷ *z* = *0* ∨ *divides-ff x y*
    and *divides-ff-mult-cancel-right*[*simp*]: *divides-ff* (*x* ∗ *z*) (*y* ∗ *z*) ⟷ *z* = *0* ∨ *divides-ff x y*
  unfolding *divides-ff-def* **by** *auto*

definition *gcd-ff-list* :: *'a::ufd fract list* ⇒ *'a fract* ⇒ *bool* **where**
  gcd-ff-list X g = (
    (∀ *x* ∈ *set X*. *divides-ff g x*) ∧
    (∀ *d*. (∀ *x* ∈ *set X*. *divides-ff d x*) ⟶ *divides-ff d g*))

lemma *gcd-ff-list-exists*: ∃ *g*. *gcd-ff-list* (*X* :: *'a::ufd fract list*) *g*
proof −
  interpret *some-gcd*: *idom-gcd* (∗) *1* :: *'a* (+) *0* (−) *uminus some-gcd*
    rewrites *dvd.dvd* ((∗)) = (*dvd*) **by** (*unfold-locales*, *auto simp*: *dvd-rewrites*)
  from *ff-list-pairs*[*of X*] **obtain** *xs* **where** *X*: *X* = *map* (λ (*x*,*y*). *Fraction-Field.Fract x y*) *xs*
    and *xs*: *0* ∉ *snd* ‘ *set xs* **by** *auto*
  define *r* **where** *r* ≡ *prod-list* (*map snd xs*)
  have *r*: *r* ≠ *0* **unfolding** *r-def prod-list-zero-iff* **using** *xs* **by** *auto*
  define *ys* **where** *ys* ≡ *map* (λ (*x*,*y*). *x* ∗ *prod-list* (*remove1 y* (*map snd xs*))) *xs*
  {
    fix *i*
    assume *i* < *length X*
    hence *i*: *i* < *length xs* **unfolding** *X* **by** *auto*
    obtain *x y* **where** *xsi*: *xs* ! *i* = (*x*,*y*) **by** *force*
    with *i* **have** (*x*,*y*) ∈ *set xs* **unfolding** *set-conv-nth* **by** *force*
    hence *y-mem*: *y* ∈ *set* (*map snd xs*) **by** *force*
    with *xs* **have** *y*: *y* ≠ *0* **by** *force*
    from *i* **have** *id1*: *ys* ! *i* = *x* ∗ *prod-list* (*remove1 y* (*map snd xs*)) **unfolding** *ys-def* **using** *xsi* **by** *auto*
    from *i xsi* **have** *id2*: *X* ! *i* = *Fraction-Field.Fract x y* **unfolding** *X* **by** *auto*
    have *lp*: *prod-list* (*remove1 y* (*map snd xs*)) ∗ *y* = *r* **unfolding** *r-def*
      by (*rule prod-list-remove1*[*OF y-mem*])
    have *ys* ! *i* ∈ *set ys* **using** *i* **unfolding** *ys-def* **by** *auto*
    moreover have *to-fract* (*ys* ! *i*) = *to-fract r* ∗ (*X* ! *i*)
      unfolding *id1 id2 to-fract-def mult-fract*
      by (*subst eq-fract*(*1*), *force*, *force simp*: *y*, *simp add*: *lp*)
    ultimately have *ys* ! *i* ∈ *set ys to-fract* (*ys* ! *i*) = *to-fract r* ∗ (*X* ! *i*) **.**
  } **note** *ys* = *this*
  define *G* **where** *G* ≡ *some-gcd.listgcd ys*
  define *g* **where** *g* ≡ *to-fract G* ∗ *Fraction-Field.Fract 1 r*

have *len*: *length X* = *length ys* **unfolding** *X ys-def* **by** *auto*
show *?thesis*
proof (*rule exI*[*of - g*], *unfold gcd-ff-list-def*, *intro ballI conjI impI allI*)
fix *x*
assume *x* ∈ *set X*
then obtain *i* **where** *i*: *i* < *length X* **and** *x*: *x* = *X* ! *i* **unfolding** *set-conv-nth* **by** *auto*
from *ys*[*OF i*] **have** *id*: *to-fract* (*ys* ! *i*) = *to-fract r* ∗ *x*
and *ysi*: *ys* ! *i* ∈ *set ys* **unfolding** *x* **by** *auto*
from *some-gcd.listgcd*[*OF ysi*] **have** *G dvd ys* ! *i* **unfolding** *G-def* **.**
then obtain *d* **where** *ysi*: *ys* ! *i* = *G* ∗ *d* **unfolding** *dvd-def* **by** *auto*
have *to-fract d* ∗ (*to-fract G* ∗ *Fraction-Field.Fract 1 r*) = *x* ∗ (*to-fract r* ∗ *Fraction-Field.Fract 1 r*)
using *id*[*unfolded ysi*]
by (*simp add*: *ac-simps*)
also have ... = *x* **using** *r* **unfolding** *to-fract-def* **by** (*simp add*: *eq-fract One-fract-def*)
finally have *to-fract d* ∗ (*to-fract G* ∗ *Fraction-Field.Fract 1 r*) = *x* **by** *simp*
thus *divides-ff g x* **unfolding** *divides-ff-def g-def*
by (*intro exI*[*of - d*], *auto*)
next
fix *d*
assume ∀ *x* ∈ *set X*. *divides-ff d x*
hence *Ball* ((λ *x*. *to-fract r* ∗ *x*) ' *set X*) (*divides-ff* (*to-fract r* ∗ *d*)) **by** *simp*
also have (λ *x*. *to-fract r* ∗ *x*) ' *set X* = *to-fract* ' *set ys*
unfolding *set-conv-nth* **using** *ys len* **by** *force*
finally have *dvd*: *Ball* (*set ys*) (λ *y*. *divides-ff* (*to-fract r* ∗ *d*) (*to-fract y*)) **by** *auto*
obtain *nd dd* **where** *d*: *d* = *Fraction-Field.Fract nd dd* **and** *dd*: *dd* ≠ *0* **by** (*cases d*, *auto*)
{
fix *y*
assume *y* ∈ *set ys*
hence *divides-ff* (*to-fract r* ∗ *d*) (*to-fract y*) **using** *dvd* **by** *auto*
from *this*[*unfolded divides-ff-def d to-fract-def mult-fract*]
obtain *ra* **where** *Fraction-Field.Fract y 1* = *Fraction-Field.Fract* (*r* ∗ *nd* ∗ *ra*) *dd* **by** *auto*
hence *y* ∗ *dd* = *ra* ∗ (*r* ∗ *nd*) **by** (*simp add*: *eq-fract dd*)
hence *r* ∗ *nd dvd y* ∗ *dd* **by** *auto*
}
hence *r* ∗ *nd dvd some-gcd.listgcd ys* ∗ *dd* **by** (*rule some-gcd.listgcd-greatest-mult*)
hence *divides-ff* (*to-fract r* ∗ *d*) (*to-fract G*) **unfolding** *to-fract-def d mult-fract G-def divides-ff-def* **by** (*auto simp add*: *eq-fract dd dvd-def*)
also have *to-fract G* = *to-fract r* ∗ *g* **unfolding** *g-def* **using** *r*
by (*auto simp*: *to-fract-def eq-fract*)
finally show *divides-ff d g* **using** *r* **by** *simp*
qed
qed

definition *some-gcd-ff-list* :: *'a* :: *ufd fract list* ⇒ *'a fract* **where**
some-gcd-ff-list xs = (*SOME g*. *gcd-ff-list xs g*)

lemma *some-gcd-ff-list*: *gcd-ff-list xs* (*some-gcd-ff-list xs*)
unfolding *some-gcd-ff-list-def* **using** *gcd-ff-list-exists*[*of xs*]
by (*rule someI-ex*)

lemma *some-gcd-ff-list-divides*: *x* ∈ *set xs* ⟹ *divides-ff* (*some-gcd-ff-list xs*) *x*
using *some-gcd-ff-list*[*of xs*] **unfolding** *gcd-ff-list-def* **by** *auto*

lemma *some-gcd-ff-list-greatest*: (∀ *x* ∈ *set xs*. *divides-ff d x*) ⟹ *divides-ff d* (*some-gcd-ff-list xs*)
using *some-gcd-ff-list*[*of xs*] **unfolding** *gcd-ff-list-def* **by** *auto*

lemma *divides-ff-refl*[*simp*]: *divides-ff x x*
unfolding *divides-ff-def*
by (*rule exI*[*of - 1*], *auto simp*: *to-fract-def One-fract-def*)

lemma *divides-ff-trans*:
divides-ff x y ⟹ *divides-ff y z* ⟹ *divides-ff x z*
unfolding *divides-ff-def*
by (*auto simp del*: *to-fract-hom.hom-mult simp add*: *to-fract-hom.hom-mult*[*symmetric*])

lemma *divides-ff-mult-right*: *a* ≠ *0* ⟹ *divides-ff* (*x* ∗ *inverse a*) *y* ⟹ *divides-ff x* (*a* ∗ *y*)
unfolding *divides-ff-def divide-inverse*[*symmetric*] **by** *auto*

definition *eq-dff* :: *'a* :: *ufd fract* ⇒ *'a fract* ⇒ *bool* (**infix** *=dff 50*) **where**
x =dff y ⟷ *divides-ff x y* ∧ *divides-ff y x*

lemma *eq-dffI*[*intro*]: *divides-ff x y* ⟹ *divides-ff y x* ⟹ *x =dff y*
unfolding *eq-dff-def* **by** *auto*

lemma *eq-dff-refl*[*simp*]: *x =dff x*
by (*intro eq-dffI*, *auto*)

lemma *eq-dff-sym*: *x =dff y* ⟹ *y =dff x* **unfolding** *eq-dff-def* **by** *auto*

lemma *eq-dff-trans*[*trans*]: *x =dff y* ⟹ *y =dff z* ⟹ *x =dff z*
unfolding *eq-dff-def* **using** *divides-ff-trans* **by** *auto*

lemma *eq-dff-cancel-right*[*simp*]: *x* ∗ *y =dff x* ∗ *z* ⟷ *x* = *0* ∨ *y =dff z*
unfolding *eq-dff-def* **by** *auto*

lemma *eq-dff-mult-right-trans*[*trans*]: *x =dff y* ∗ *z* ⟹ *z =dff u* ⟹ *x =dff y* ∗ *u*
using *eq-dff-trans* **by** *force*

lemma *some-gcd-ff-list-smult*: *a* ≠ *0* ⟹ *some-gcd-ff-list* (*map* ((∗) *a*) *xs*) *=dff a* ∗ *some-gcd-ff-list xs*

proof
  let *?g* = *some-gcd-ff-list* (*map* ((∗) *a*) *xs*)
  show *divides-ff* (*a* ∗ *some-gcd-ff-list xs*) *?g*
    by (*rule some-gcd-ff-list-greatest*, *insert some-gcd-ff-list-divides*[*of - xs*], *auto simp*: *divides-ff-def*)
  assume *a*: *a* ≠ *0*
  show *divides-ff ?g* (*a* ∗ *some-gcd-ff-list xs*)
  proof (*rule divides-ff-mult-right*[*OF a some-gcd-ff-list-greatest*], *intro ballI*)
    fix *x*
    assume *x*: *x* ∈ *set xs*
    have *divides-ff* (*?g* ∗ *inverse a*) *x* = *divides-ff* (*inverse a* ∗ *?g*) (*inverse a* ∗ (*a* ∗ *x*))
      using *a* **by** (*simp add*: *field-simps*)
    also have . . . **using** *a x* **by** (*auto intro*: *some-gcd-ff-list-divides*)
    finally show *divides-ff* (*?g* ∗ *inverse a*) *x* **.**
  qed
qed

definition *content-ff* :: ′*a*::*ufd fract poly* ⇒ ′*a fract* **where**
  content-ff p = *some-gcd-ff-list* (*coeffs p*)

lemma *content-ff-iff*: *divides-ff x* (*content-ff p*) ⟷ (∀ *c* ∈ *set* (*coeffs p*). *divides-ff x c*) (**is** *?l* = *?r*)
proof
  assume *?l*
  from *divides-ff-trans*[*OF this*, *unfolded content-ff-def*, *OF some-gcd-ff-list-divides*]
show *?r* **..**
next
  assume *?r*
  thus *?l* **unfolding** *content-ff-def* **by** (*intro some-gcd-ff-list-greatest*, *auto*)
qed

lemma *content-ff-divides-ff*: *x* ∈ *set* (*coeffs p*) ⟹ *divides-ff* (*content-ff p*) *x*
  unfolding *content-ff-def* **by** (*rule some-gcd-ff-list-divides*)

lemma *content-ff-0*[*simp*]: *content-ff 0* = *0*
  using *content-ff-iff*[*of 0 0*] **by** (*auto simp*: *divides-ff-def*)

lemma *content-ff-0-iff*[*simp*]: (*content-ff p* = *0*) = (*p* = *0*)
proof (*cases p* = *0*)
  case *False*
  define *a* **where** *a* ≡ *last* (*coeffs p*)
  define *xs* **where** *xs* ≡ *coeffs p*
  from *False*
  have *mem*: *a* ∈ *set* (*coeffs p*) **and** *a*: *a* ≠ *0*
    unfolding *a-def last-coeffs-eq-coeff-degree*[*OF False*] *coeffs-def* **by** *auto*
  from *content-ff-divides-ff*[*OF mem*] **have** *divides-ff* (*content-ff p*) *a* **.**
  with *a* **have** *content-ff p* ≠ *0* **unfolding** *divides-ff-def* **by** *auto*
  with *False* **show** *?thesis* **by** *auto*

qed *auto*

lemma *content-ff-eq-dff-nonzero*: *content-ff p =dff x* $\Longrightarrow$ *x* $\neq$ *0* $\Longrightarrow$ *p* $\neq$ *0*
  using *divides-ff-def eq-dff-def* **by** *force*

lemma *content-ff-smult*: *content-ff* (*smult* (*a*::′*a*::*ufd fract*) *p*) *=dff a* ∗ *content-ff p*
proof (*cases a* = *0*)
  case *False* **note** *a* = *this*
  have *id*: *coeffs* (*smult a p*) = *map* ((∗) *a*) (*coeffs p*)
    unfolding *coeffs-smult* **using** *a* **by** (*simp add*: *Polynomial.coeffs-smult*)
  show *?thesis* **unfolding** *content-ff-def id* **using** *some-gcd-ff-list-smult*[*OF a*] **.**
qed *simp*

definition *normalize-content-ff*
  where *normalize-content-ff* (*p*::′*a*::*ufd fract poly*) ≡ *smult* (*inverse* (*content-ff p*)) *p*

lemma *smult-normalize-content-ff*: *smult* (*content-ff p*) (*normalize-content-ff p*) = *p*
  unfolding *normalize-content-ff-def*
  by (*cases p* = *0*, *auto*)

lemma *content-ff-normalize-content-ff-1*: **assumes** *p0*: *p* $\neq$ *0*
  shows *content-ff* (*normalize-content-ff p*) *=dff 1*
proof −
  have *content-ff p* = *content-ff* (*smult* (*content-ff p*) (*normalize-content-ff p*)) **unfolding** *smult-normalize-content-ff* **..**
  also have . . . *=dff content-ff p* ∗ *content-ff* (*normalize-content-ff p*) **by** (*rule content-ff-smult*)
  finally show *?thesis* **unfolding** *eq-dff-def divides-ff-def* **using** *p0* **by** *auto*
qed

lemma *content-ff-to-fract*: **assumes** *set* (*coeffs p*) ⊆ *range to-fract*
  shows *content-ff p* ∈ *range to-fract*
proof −
  have *divides-ff 1* (*content-ff p*) **using** *assms*
    unfolding *content-ff-iff* **unfolding** *divides-ff-def*[*abs-def*] **by** *auto*
  thus *?thesis* **unfolding** *divides-ff-def* **by** *auto*
qed

lemma *content-ff-map-poly-to-fract*: *content-ff* (*map-poly to-fract* (*p* :: ′*a* :: *ufd poly*)) ∈ *range to-fract*
  by (*rule content-ff-to-fract*, *subst coeffs-map-poly*, *auto*)

lemma *range-coeffs-to-fract*: **assumes** *set* (*coeffs p*) ⊆ *range to-fract*
  shows ∃ *m*. *coeff p i* = *to-fract m*
proof −
  from *assms*(*1*) *to-fract-0* **have** *coeff p i* ∈ *range to-fract* **using** *range-coeff* [*of*

p]
  by *auto* (*metis contra-subsetD to-fract-hom.hom-zero insertE range-eqI*)
 thus *?thesis* **by** *auto*
qed

lemma *divides-ff-coeff*: **assumes** *set* (*coeffs p*) $\subseteq$ *range to-fract* **and** *divides-ff* (*to-fract n*) (*coeff p i*)
 shows $\exists$ *m*. *coeff p i* = *to-fract n* $*$ *to-fract m*
proof −
 from *range-coeffs-to-fract*[*OF assms*(*1*)] **obtain** *k* **where** *pi*: *coeff p i* = *to-fract k* **by** *auto*
 from *assms*(*2*)[*unfolded this*] **have** *n dvd k* **by** *simp*
 then obtain *j* **where** *k*: *k* = *n* $*$ *j* **unfolding** *Rings.dvd-def* **by** *auto*
 show *?thesis* **unfolding** *pi k* **by** *auto*
qed

definition *inv-embed* :: *'a* :: *ufd fract* ⇒ *'a* **where**
 inv-embed = *the-inv to-fract*

lemma *inv-embed*[*simp*]: *inv-embed* (*to-fract x*) = *x*
 unfolding *inv-embed-def*
 by (*rule the-inv-f-f*, *auto simp*: *inj-on-def*)

lemma *inv-embed-0*[*simp*]: *inv-embed 0* = *0* **unfolding** *to-fract-0*[*symmetric*] *inv-embed* **by** *simp*

lemma *range-to-fract-embed-poly*: **assumes** *set* (*coeffs p*) $\subseteq$ *range to-fract*
 shows *p* = *map-poly to-fract* (*map-poly inv-embed p*)
proof −
 have *p* = *map-poly* (*to-fract o inv-embed*) *p*
  by (*rule sym*, *rule map-poly-idI*, *insert assms*, *auto*)
 also have ... = *map-poly to-fract* (*map-poly inv-embed p*)
  by (*subst map-poly-map-poly*, *auto*)
 finally show *?thesis* **.**
qed

lemma *content-ff-to-fract-coeffs-to-fract*: **assumes** *content-ff p* $\in$ *range to-fract*
 shows *set* (*coeffs p*) $\subseteq$ *range to-fract*
proof
 fix *x*
 assume *x* $\in$ *set* (*coeffs p*)
 from *content-ff-divides-ff*[*OF this*] *assms*[*unfolded eq-dff-def*] **show** *x* $\in$ *range to-fract*
  unfolding *divides-ff-def* **by** (*auto simp del*: *to-fract-hom.hom-mult simp*: *to-fract-hom.hom-mult*[*symmetric*])
qed

lemma *content-ff-1-coeffs-to-fract*: **assumes** *content-ff p* =*dff* *1*
 shows *set* (*coeffs p*) $\subseteq$ *range to-fract*
proof

fix *x*
assume *x* ∈ *set* (*coeffs p*)
from *content-ff-divides-ff*[*OF this*] *assms*[*unfolded eq-dff-def*] **show** *x* ∈ *range to-fract*
unfolding *divides-ff-def* **by** (*auto simp del*: *to-fract-hom.hom-mult simp*: *to-fract-hom.hom-mult*[*symmetric*])
qed

lemma *gauss-lemma*:
fixes *p q* :: *'a* :: *ufd fract poly*
shows *content-ff* (*p* ∗ *q*) =*dff content-ff p* ∗ *content-ff q*
proof (*cases p* = *0* ∨ *q* = *0*)
case *False*
hence *p*: *p* ≠ *0* **and** *q*: *q* ≠ *0* **by** *auto*
let *?c* = *content-ff* :: *'a fract poly* ⇒ *'a fract*
{
fix *p q* :: *'a fract poly*
assume *cp1*: *?c p* =*dff 1* **and** *cq1*: *?c q* =*dff 1*
define *ip* **where** *ip* ≡ *map-poly inv-embed p*
define *iq* **where** *iq* ≡ *map-poly inv-embed q*
interpret *map-poly-hom*: *map-poly-comm-ring-hom to-fract*..
from *content-ff-1-coeffs-to-fract*[*OF cp1*] **have** *cp*: *set* (*coeffs p*) ⊆ *range to-fract* .
from *content-ff-1-coeffs-to-fract*[*OF cq1*] **have** *cq*: *set* (*coeffs q*) ⊆ *range to-fract* .
have *ip*: *p* = *map-poly to-fract ip* **unfolding** *ip-def*
by (*rule range-to-fract-embed-poly*[*OF cp*])
have *iq*: *q* = *map-poly to-fract iq* **unfolding** *iq-def*
by (*rule range-to-fract-embed-poly*[*OF cq*])
have *cpq0*: *?c* (*p* ∗ *q*) ≠ *0*
unfolding *content-ff-0-iff* **using** *cp1 cq1 content-ff-eq-dff-nonzero*[*of - 1*] **by** *auto*
have *cpq*: *set* (*coeffs* (*p* ∗ *q*)) ⊆ *range to-fract* **unfolding** *ip iq*
unfolding *map-poly-hom.hom-mult*[*symmetric*] *to-fract-hom.coeffs-map-poly-hom* **by** *auto*
have *ctnt*: *?c* (*p* ∗ *q*) ∈ *range to-fract* **using** *content-ff-to-fract*[*OF cpq*] .
then obtain *cpq* **where** *id*: *?c* (*p* ∗ *q*) = *to-fract cpq* **by** *auto*
have *dvd*: *divides-ff 1* (*?c* (*p* ∗ *q*)) **using** *ctnt* **unfolding** *divides-ff-def* **by** *auto*
from *cpq0*[*unfolded id*] **have** *cpq0*: *cpq* ≠ *0* **unfolding** *to-fract-def Zero-fract-def* **by** *auto*
hence *cpqM*: *cpq* ∈ *carrier mk-monoid* **by** *auto*
have *?c* (*p* ∗ *q*) =*dff 1*
proof (*rule ccontr*)
assume ¬ *?c* (*p* ∗ *q*) =*dff 1*
with *dvd* **have** ¬ *divides-ff* (*?c* (*p* ∗ *q*)) *1*
unfolding *eq-dff-def* **by** *auto*
from *this*[*unfolded id divides-ff-def*] **have** *cpq*: ⋀ *r*. *cpq* ∗ *r* ≠ *1*
by (*auto simp*: *to-fract-def One-fract-def eq-fract*)
then have *cpq1*: ¬ *cpq dvd 1* **by** (*auto elim*:*dvdE simp*:*ac-simps*)
from *mset-factors-exist*[*OF cpq0 cpq1*]

obtain *F* **where** *F*: *mset-factors F cpq* **by** *auto*
have $F \neq \{\#\}$ **using** *F* **by** *auto*
then obtain *f* **where** *f*: $f \in\# F$ **by** *auto*
with *F* **have** *irrf*: *irreducible f* **and** *f0*: $f \neq 0$ **by** (*auto dest*: *mset-factorsD*)
from *irrf* **have** *pf*: *prime-elem f* **by** *simp*
note $* =$ *this*[*unfolded prime-elem-def*]
from $*$ **have** *no-unit*: $\neg$ *f dvd 1* **by** *auto*
from $*$ *f0* **have** *prime*: $\bigwedge$ *a b. f dvd a* $*$ *b* $\Longrightarrow$ *f dvd a* $\vee$ *f dvd b* **unfolding** *dvd-def* **by** *force*
let *?f* = *to-fract f*
from *F f*
have *fdvd*: *f dvd cpq* **by** (*auto intro*:*mset-factors-imp-dvd*)
hence *divides-ff ?f* (*to-fract cpq*) **by** *simp*
from *divides-ff-trans*[*OF this*, *folded id*, *OF content-ff-divides-ff*]
have *dvd*: $\bigwedge$ *z. z* $\in$ *set* (*coeffs* (*p* $*$ *q*)) $\Longrightarrow$ *divides-ff ?f z* **.**
{
fix *p* :: *'a fract poly*
assume *cp*: *?c p =dff 1*
let *?P* = λ *i.* $\neg$ *divides-ff ?f* (*coeff p i*)
{
assume $\forall$ *c* $\in$ *set* (*coeffs p*). *divides-ff ?f c*
hence *n*: *divides-ff ?f* (*?c p*) **unfolding** *content-ff-iff* **by** *auto*
from *divides-ff-trans*[*OF this*] *cp*[*unfolded eq-dff-def*] **have** *divides-ff ?f 1* **by** *auto*
also have *1* = *to-fract 1* **by** *simp*
finally have *f dvd 1* **by** (*unfold divides-ff-to-fract*)
hence *False* **using** *no-unit* **unfolding** *dvd-def* **by** (*auto simp*: *ac-simps*)
}
then obtain *cp* **where** *cp*: *cp* $\in$ *set* (*coeffs p*) **and** *ncp*: $\neg$ *divides-ff ?f cp* **by** *auto*
hence *cp* $\in$ *range* (*coeff p*) **unfolding** *range-coeff* **by** *auto*
with *ncp* **have** $\exists$ *i. ?P i* **by** *auto*
from *LeastI-ex*[*OF this*] *not-less-Least*[*of - ?P*]
have $\exists$ *i. ?P i* $\wedge$ ($\forall$ *j. j* $<$ *i* $\longrightarrow$ *divides-ff ?f* (*coeff p j*)) **by** *blast*
} note *cont* = *this*
from *cont*[*OF cp1*] **obtain** *r* **where**
r: $\neg$ *divides-ff ?f* (*coeff p r*) **and** *r'*: $\bigwedge$ *i. i* $<$ *r* $\Longrightarrow$ *divides-ff ?f* (*coeff p i*) **by** *auto*
have $\forall$ *i.* $\exists$ *k. i* $<$ *r* $\longrightarrow$ *coeff p i* = *?f* $*$ *to-fract k* **using** *divides-ff-coeff*[*OF cp r'*] **by** *blast*
from *choice*[*OF this*] **obtain** *rr* **where** *r'*: $\bigwedge$ *i. i* $<$ *r* $\Longrightarrow$ *coeff p i* = *?f* $*$ *to-fract* (*rr i*) **by** *blast*
let *?r* = *coeff p r*
from *cont*[*OF cq1*] **obtain** *s* **where**
s: $\neg$ *divides-ff ?f* (*coeff q s*) **and** *s'*: $\bigwedge$ *i. i* $<$ *s* $\Longrightarrow$ *divides-ff ?f* (*coeff q i*) **by** *auto*
have $\forall$ *i.* $\exists$ *k. i* $<$ *s* $\longrightarrow$ *coeff q i* = *?f* $*$ *to-fract k* **using** *divides-ff-coeff*[*OF cq s'*] **by** *blast*
from *choice*[*OF this*] **obtain** *ss* **where** *s'*: $\bigwedge$ *i. i* $<$ *s* $\Longrightarrow$ *coeff q i* = *?f* $*$

to-fract (*ss i*) **by** *blast*
from *range-coeffs-to-fract*[*OF cp*] **have** ∀ *i*. ∃ *m*. *coeff p i* = *to-fract m* **..**
from *choice*[*OF this*] **obtain** *pi* **where** *pi*: ⋀ *i*. *coeff p i* = *to-fract* (*pi i*) **by** *blast*
from *range-coeffs-to-fract*[*OF cq*] **have** ∀ *i*. ∃ *m*. *coeff q i* = *to-fract m* **..**
from *choice*[*OF this*] **obtain** *qi* **where** *qi*: ⋀ *i*. *coeff q i* = *to-fract* (*qi i*) **by** *blast*
let *?s* = *coeff q s*
let *?g* = λ *i*. *coeff p i* ∗ *coeff q* (*r* + *s* − *i*)
define *a* **where** *a* = (∑ *i*∈{..<*r*}. (*rr i* ∗ *qi* (*r* + *s* − *i*)))
define *b* **where** *b* = (∑ *i* ∈ {*Suc r*..*r* + *s*}. *pi i* ∗ (*ss* (*r* + *s* − *i*)))
have *coeff* (*p* ∗ *q*) (*r* + *s*) = (∑ *i*≤*r* + *s*. *?g i*) **unfolding** *coeff-mult* **..**
also have {..*r*+*s*} = {..< *r*} ∪ {*r* .. *r*+*s*} **by** *auto*
also have (∑ *i*∈{..<*r*} ∪ {*r*..*r* + *s*}. *?g i*)
= (∑ *i*∈{..<*r*}. *?g i*) + (∑ *i* ∈ {*r*..*r* + *s*}. *?g i*)
by (*rule sum.union-disjoint*, *auto*)
also have (∑ *i*∈{..<*r*}. *?g i*) = (∑ *i*∈{..<*r*}. *?f* ∗ (*to-fract* (*rr i*) ∗ *to-fract* (*qi* (*r* + *s* − *i*))))
by (*rule sum.cong*[*OF refl*], *insert r′ qi*, *auto*)
also have ... = *to-fract* (*f* ∗ *a*) **by** (*simp add*: *a-def sum-distrib-left*)
also have (∑ *i* ∈ {*r*..*r* + *s*}. *?g i*) = *?g r* + (∑ *i* ∈ {*Suc r*..*r* + *s*}. *?g i*)
by (*subst sum.remove*[*of - r*], *auto intro*: *sum.cong*)
also have (∑ *i* ∈ {*Suc r*..*r* + *s*}. *?g i*) = (∑ *i* ∈ {*Suc r*..*r* + *s*}. *?f* ∗ (*to-fract* (*pi i*) ∗ *to-fract* (*ss* (*r* + *s* − *i*))))
by (*rule sum.cong*[*OF refl*], *insert s′ pi*, *auto*)
also have ... = *to-fract* (*f* ∗ *b*) **by** (*simp add*: *sum-distrib-left b-def*)
finally have *cpq*: *coeff* (*p* ∗ *q*) (*r* + *s*) = *to-fract* (*f* ∗ (*a* + *b*)) + *?r* ∗ *?s* **by** (*simp add*: *field-simps*)
{
fix *i*
from *dvd*[*of coeff* (*p* ∗ *q*) *i*] **have** *divides-ff ?f* (*coeff* (*p* ∗ *q*) *i*) **using** *range-coeff*[*of p* ∗ *q*]
by (*cases coeff* (*p* ∗ *q*) *i* = *0*, *auto simp*: *divides-ff-def*)
}
from *this*[*of r* + *s*, *unfolded cpq*] **have** *divides-ff ?f* (*to-fract* (*f* ∗ (*a* + *b*) + *pi r* ∗ *qi s*))
unfolding *pi qi* **by** *simp*
from *this*[*unfolded divides-ff-to-fract*] **have** *f dvd pi r* ∗ *qi s*
by (*metis dvd-add-times-triv-left-iff mult.commute*)
from *prime*[*OF this*] **have** *f dvd pi r* ∨ *f dvd qi s* **by** *auto*
with *r s* **show** *False* **unfolding** *pi qi* **by** *auto*
qed
} **note** *main* = *this*
define *n* **where** *n* ≡ *normalize-content-ff* :: ′*a fract poly* ⇒ ′*a fract poly*
let *?s* = λ *p*. *smult* (*content-ff p*) (*n p*)
have *?c* (*p* ∗ *q*) = *?c* (*?s p* ∗ *?s q*) **unfolding** *smult-normalize-content-ff n-def* **by** *simp*
also have *?s p* ∗ *?s q* = *smult* (*?c p* ∗ *?c q*) (*n p* ∗ *n q*) **by** (*simp add*: *mult.commute*)

also have *?c* (...) *=dff* (*?c p* ∗ *?c q*) ∗ *?c* (*n p* ∗ *n q*) **by** (*rule content-ff-smult*)
also have *?c* (*n p* ∗ *n q*) *=dff 1* **unfolding** *n-def*
by (*rule main*, *insert p q*, *auto simp*: *content-ff-normalize-content-ff-1*)
finally show *?thesis* **by** *simp*
qed *auto*

abbreviation (*input*) *content-ff-ff p* ≡ *content-ff* (*map-poly to-fract p*)

lemma *factorization-to-fract*:
assumes *q*: *q* ≠ *0* **and** *factor*: *map-poly to-fract* (*p* :: *'a* :: *ufd poly*) = *q* ∗ *r*
shows ∃ *q' r' c*. *c* ≠ *0* ∧ *q* = *smult c* (*map-poly to-fract q'*) ∧
r = *smult* (*inverse c*) (*map-poly to-fract r'*) ∧
content-ff-ff q' *=dff 1* ∧ *p* = *q'* ∗ *r'*
proof −
let *?c* = *content-ff*
let *?p* = *map-poly to-fract p*
interpret *map-poly-inj-comm-ring-hom to-fract* :: *'a* ⇒ *-*..
define *cq* **where** *cq* ≡ *normalize-content-ff q*
define *cr* **where** *cr* ≡ *smult* (*content-ff q*) *r*
define *q'* **where** *q'* ≡ *map-poly inv-embed cq*
define *r'* **where** *r'* ≡ *map-poly inv-embed cr*
from *content-ff-map-poly-to-fract* **have** *cp-ff*: *?c ?p* ∈ *range to-fract* **by** *auto*
from *smult-normalize-content-ff*[*of q*] **have** *cqs*: *q* = *smult* (*content-ff q*) *cq* **unfolding** *cq-def* **..**
from *content-ff-normalize-content-ff-1*[*OF q*] **have** *c-cq*: *content-ff cq* *=dff 1* **unfolding** *cq-def* **.**
from *content-ff-1-coeffs-to-fract*[*OF this*] **have** *cq-ff*: *set* (*coeffs cq*) ⊆ *range to-fract* **.**
have *factor*: *?p* = *cq* ∗ *cr* **unfolding** *factor cr-def* **using** *cqs*
by (*metis mult-smult-left mult-smult-right*)
from *gauss-lemma*[*of cq cr*] **have** *cp*: *?c ?p* *=dff* *?c cq* ∗ *?c cr* **unfolding** *factor* **.**
with *c-cq* **have** *?c ?p* *=dff* *?c cr*
by (*metis eq-dff-mult-right-trans mult.commute mult.right-neutral*)
with *cp-ff* **have** *?c cr* ∈ *range to-fract*
by (*metis divides-ff-def to-fract-hom.hom-mult eq-dff-def image-iff range-eqI*)
from *content-ff-to-fract-coeffs-to-fract*[*OF this*] **have** *cr-ff*: *set* (*coeffs cr*) ⊆ *range to-fract* **by** *auto*
have *cq*: *cq* = *map-poly to-fract q'* **unfolding** *q'-def*
by (*rule range-to-fract-embed-poly*[*OF cq-ff*])
have *cr*: *cr* = *map-poly to-fract r'* **unfolding** *r'-def*
by (*rule range-to-fract-embed-poly*[*OF cr-ff*])
from *factor*[*unfolded cq cr*]
have *p*: *p* = *q'* ∗ *r'* **by** (*simp add*: *injectivity*)
from *c-cq* **have** *ctnt*: *content-ff-ff q'* *=dff 1* **using** *cq q'-def* **by** *force*
from *cqs* **have** *idq*: *q* = *smult* (*?c q*) (*map-poly to-fract q'*) **unfolding** *cq* **.**
with *q* **have** *cq*: *?c q* ≠ *0* **by** *auto*
have *r* = *smult* (*inverse* (*?c q*)) *cr* **unfolding** *cr-def* **using** *cq* **by** *auto*
also have *cr* = *map-poly to-fract r'* **by** (*rule cr*)

finally have *idr*: $r = smult\ (inverse\ (?c\ q))\ (map\text{-}poly\ to\text{-}fract\ r')$ **by** *auto*
from *cq p ctnt idq idr* **show** *?thesis* **by** *blast*
qed

lemma *irreducible-PM-M-PFM*:
assumes *irr*: *irreducible p*
shows $degree\ p = 0 \wedge irreducible\ (coeff\ p\ 0) \vee$
$degree\ p \neq 0 \wedge irreducible\ (map\text{-}poly\ to\text{-}fract\ p) \wedge content\text{-}ff\text{-}ff\ p =dff\ 1$
proof−
interpret *map-poly-inj-idom-hom to-fract*..
from *irr*[*unfolded irreducible-altdef*]
have *p0*: $p \neq 0$ **and** *irr*: $\neg\ p\ dvd\ 1\ \bigwedge b.\ b\ dvd\ p \Longrightarrow \neg\ p\ dvd\ b \Longrightarrow b\ dvd\ 1$ **by** *auto*
show *?thesis*
proof (*cases* $degree\ p = 0$)
case *True*
from *degree0-coeffs*[*OF True*] **obtain** *a* **where** *p*: $p = [:a:]$ **by** *auto*
note $irr = irr[unfolded\ p]$
from *p p0* **have** *a0*: $a \neq 0$ **by** *auto*
moreover have $\neg\ a\ dvd\ 1$ **using** *irr*(*1*) **by** *simp*
moreover {
fix *b*
assume *b dvd a* $\neg$ *a dvd b*
hence $[:b:]\ dvd\ [:a:]\ \neg\ [:a:]\ dvd\ [:b:]$ **unfolding** *const-poly-dvd* **.**
from *irr*(*2*)[*OF this*] **have** *b dvd 1* **unfolding** *const-poly-dvd-1* **.**
}
ultimately have *irreducible a* **unfolding** *irreducible-altdef* **by** *auto*
with *True* **show** *?thesis* **unfolding** *p* **by** *auto*
next
case *False*
let $?E = map\text{-}poly\ to\text{-}fract$
let $?p = ?E\ p$
have *dp*: $degree\ ?p \neq 0$ **using** *False* **by** *simp*
from *p0* **have** *p'*: $?p \neq 0$ **by** *simp*
moreover have $\neg\ ?p\ dvd\ 1$
proof
assume *?p dvd 1* **then obtain** *q* **where** *id*: $?p * q = 1$ **unfolding** *dvd-def* **by** *auto*
have *deg*: $degree\ (?p * q) = degree\ ?p + degree\ q$
by (*rule degree-mult-eq*, *insert id*, *auto*)
from *arg-cong*[*OF id*, *of degree*, *unfolded deg*] *dp* **show** *False* **by** *auto*
qed
moreover {
fix *q*
assume *q dvd ?p* **and** *ndvd*: $\neg$ *?p dvd q*
then obtain *r* **where** *fact*: $?p = q * r$ **unfolding** *dvd-def* **by** *auto*
with *p'* **have** *q0*: $q \neq 0$ **by** *auto*
from *factorization-to-fract*[*OF this fact*] **obtain** *q' r' c* **where** $*$: $c \neq 0\ q = smult\ c\ (?E\ q')$

$r = smult\ (inverse\ c)\ (?E\ r')\ content\text{-}ff\text{-}ff\ q' =dff\ 1$
$p = q' * r'$ **by** *auto*
hence q' *dvd* p **unfolding** *dvd-def* **by** *auto*
note *irr* = *irr*(*2*)[*OF this*]
have ¬ *p dvd* q'
proof
assume *p dvd* q'
then obtain *u* **where** q': $q' = p * u$ **unfolding** *dvd-def* **by** *auto*
from *arg-cong*[*OF this*, *of* λ *x*. *smult c* (*?E x*), *unfolded* ∗(*2*)[*symmetric*]]
have *q* = *?p* ∗ *smult c* (*?E u*) **by** *simp*
hence *?p dvd q* **unfolding** *dvd-def* **by** *blast*
with *ndvd* **show** *False* **..**
qed
from *irr*[*OF this*] **have** q' *dvd 1* **.**
from *divides-degree*[*OF this*] **have** *degree* $q' = 0$ **by** *auto*
from *degree0-coeffs*[*OF this*] **obtain** a' **where** $q' = [:a':]$ **by** *auto*
from ∗(*2*)[*unfolded this*] **obtain** *a* **where** *q*: *q* = [:*a*:]
by (*simp add*: *to-fract-hom.map-poly-pCons-hom*)
with *q0* **have** *a*: *a* ≠ *0* **by** *auto*
have *q dvd 1* **unfolding** *q const-poly-dvd-1* **using** *a* **unfolding** *dvd-def*
by (*intro exI*[*of - inverse a*], *auto*)
}
ultimately have *irr-p'*: *irreducible ?p* **unfolding** *irreducible-altdef* **by** *auto*
let *?c* = *content-ff*
have *?c ?p* ∈ *range to-fract*
by (*rule content-ff-to-fract*, *unfold to-fract-hom.coeffs-map-poly-hom*, *auto*)
then obtain *c* **where** *cp*: *?c ?p* = *to-fract c* **by** *auto*
from p' *cp* **have** *c*: *c* ≠ *0* **by** *auto*
have *?c ?p* =*dff 1* **unfolding** *cp*
proof (*rule ccontr*)
define *cp* **where** *cp* = *normalize-content-ff ?p*
from *smult-normalize-content-ff*[*of ?p*] **have** *cps*: *?p* = *smult* (*to-fract c*) *cp* **unfolding** *cp-def cp* **..**
from *content-ff-normalize-content-ff-1*[*OF p'*] **have** *c-cp*: *content-ff cp* =*dff 1* **unfolding** *cp-def* **.**
from *range-to-fract-embed-poly*[*OF content-ff-1-coeffs-to-fract*[*OF c-cp*]] **obtain** cp' **where** *cp* = *?E* cp' **by** *auto*
from *cps*[*unfolded this*] **have** *p* = *smult c* cp' **by** (*simp add*: *injectivity*)
hence *dvd*: [: *c* :] *dvd p* **unfolding** *dvd-def* **by** *auto*
have ¬ *p dvd* [: *c* :] **using** *divides-degree*[*of p* [: *c* :]] *c False* **by** *auto*
from *irr*(*2*)[*OF dvd this*] **have** *c dvd 1* **by** *simp*
assume ¬ *to-fract c* =*dff 1*
from *this*[*unfolded eq-dff-def One-fract-def to-fract-def*[*symmetric*] *divides-ff-def to-fract-mult*]
have *c1*: ⋀ *r*. *1* ≠ *c* ∗ *r* **by** (*auto simp*: *ac-simps simp del*: *to-fract-hom.hom-mult simp*: *to-fract-hom.hom-mult*[*symmetric*])
with ‹*c dvd 1*› **show** *False* **unfolding** *dvd-def* **by** *blast*
qed
with *False irr-p'* **show** *?thesis* **by** *auto*

qed
qed

lemma *irreducible-M-PM*:
fixes *p* :: *'a* :: *ufd poly* **assumes** *0*: *degree p* = *0* **and** *irr*: *irreducible* (*coeff p 0*)
shows *irreducible p*
proof (*cases p* = *0*)
case *True*
thus *?thesis* **using** *assms* **by** *auto*
next
case *False*
from *degree0-coeffs*[*OF 0*] **obtain** *a* **where** *p*: *p* = [:*a*:] **by** *auto*
with *False* **have** *a0*: *a* ≠ *0* **by** *auto*
from *p irr* **have** *irreducible a* **by** *auto*
from *this*[*unfolded irreducible-altdef*]
have *a1*: ¬ *a dvd 1* **and** *irr*: ⋀ *b*. *b dvd a* ⟹ ¬ *a dvd b* ⟹ *b dvd 1* **by** *auto*
{
fix *b*
assume ∗: *b dvd* [:*a*:] ¬ [:*a*:] *dvd b*
from *divides-degree*[*OF this*(*1*)] *a0* **have** *degree b* = *0* **by** *auto*
from *degree0-coeffs*[*OF this*] **obtain** *bb* **where** *b*: *b* = [: *bb* :] **by** *auto*
from ∗ *irr*[*of bb*] **have** *b dvd 1* **unfolding** *b const-poly-dvd* **by** *auto*
}
with *a0 a1* **show** *?thesis* **by** (*auto simp*: *irreducible-altdef p*)
qed

lemma *primitive-irreducible-imp-degree*:
primitive (*p*::*'a*::{*semiring-gcd*,*idom*} *poly*) ⟹ *irreducible p* ⟹ *degree p* > *0*
by (*unfold irreducible-primitive-connect*[*symmetric*], *auto*)

lemma *irreducible-degree-field*:
fixes *p* :: *'a* :: *field poly* **assumes** *irreducible p*
shows *degree p* > *0*
proof−
{
assume *degree p* = *0*
from *degree0-coeffs*[*OF this*] *assms* **obtain** *a* **where** *p*: *p* = [:*a*:] **and** *a*: *a* ≠ *0*
by *auto*
hence *1* = *p* ∗ [:*inverse a*:] **by** *auto*
hence *p dvd 1* **..**
hence *p* ∈ *Units mk-monoid* **by** *simp*
with *assms* **have** *False* **unfolding** *irreducible-def* **by** *auto*
} **then show** *?thesis* **by** *auto*
qed

lemma *irreducible-PFM-PM*: **assumes**
irr: *irreducible* (*map-poly to-fract p*) **and** *ct*: *content-ff-ff p* =*dff 1*
shows *irreducible p*
proof −

let *?E* = *map-poly to-fract*
let *?p* = *?E p*
from *ct* **have** *p0*: $p \neq 0$ **by** (*auto simp*: *eq-dff-def divides-ff-def*)
moreover
from *irreducible-degree-field*[*OF irr*] **have** *deg*: *degree p* $\neq 0$ **by** *simp*
from *irr*[*unfolded irreducible-altdef*]
have *irr*: $\bigwedge$ *b*. *b dvd ?p* $\Longrightarrow \neg$ *?p dvd b* $\Longrightarrow$ *b dvd 1* **by** *auto*
have $\neg$ *p dvd 1* **using** *deg divides-degree*[*of p 1*] **by** *auto*
moreover {
fix *q* :: *'a poly*
assume *dvd*: *q dvd p* **and** *ndvd*: $\neg$ *p dvd q*
from *dvd* **obtain** *r* **where** *pqr*: $p = q * r$ **..**
from *arg-cong*[*OF this*, *of ?E*] **have** *pqr'*: *?p* = *?E q* $*$ *?E r* **by** *simp*
from *p0 pqr* **have** *q*: $q \neq 0$ **and** *r*: $r \neq 0$ **by** *auto*
have *dp*: *degree p* = *degree q* + *degree r* **unfolding** *pqr*
by (*subst degree-mult-eq*, *insert q r*, *auto*)
from *eq-dff-trans*[*OF eq-dff-sym*[*OF gauss-lemma*[*of ?E q ?E r*, *folded pqr'*]] *ct*]
have *ct*: *content-ff* (*?E q*) $*$ *content-ff* (*?E r*) =*dff 1* **.**
from *content-ff-map-poly-to-fract* **obtain** *cq* **where** *cq*: *content-ff* (*?E q*) = *to-fract cq* **by** *auto*
from *content-ff-map-poly-to-fract* **obtain** *cr* **where** *cr*: *content-ff* (*?E r*) = *to-fract cr* **by** *auto*
note *ct*[*unfolded cq cr to-fract-mult eq-dff-def divides-ff-def*]
from *this*[*folded hom-distribs*]
obtain *c* **where** *c*: $cq * cr * c = 1$ **by** (*auto simp del*: *to-fract-hom.hom-mult simp*: *to-fract-hom.hom-mult*[*symmetric*])
hence *one*: $1 = cq * (c * cr)$ $1 = cr * (c * cq)$ **by** (*auto simp*: *ac-simps*)
{
assume $*$: *degree q* $\neq 0 \wedge$ *degree r* $\neq 0$
with *dp* **have** *degree q* < *degree p* **by** *auto*
hence *degree* (*?E q*) < *degree* (*?E p*) **by** *simp*
hence *ndvd*: $\neg$ *?p dvd ?E q* **using** *divides-degree*[*of ?p ?E q*] *q* **by** *auto*
have *?E q dvd ?p* **unfolding** *pqr'* **by** *auto*
from *irr*[*OF this ndvd*] **have** *?E q dvd 1* **.**
from *divides-degree*[*OF this*] $*$ **have** *False* **by** *auto*
}
hence *degree q* = 0 $\vee$ *degree r* = 0 **by** *blast*
then have *q dvd 1*
proof
assume *degree q* = 0
from *degree0-coeffs*[*OF this*] *q* **obtain** *a* **where** *q*: *q* = [:*a*:] **and** *a*: $a \neq 0$ **by** *auto*
hence *id*: *set* (*coeffs* (*?E q*)) = {*to-fract a*} **by** *auto*
have *divides-ff* (*to-fract a*) (*content-ff* (*?E q*)) **unfolding** *content-ff-iff id* **by** *auto*
from *this*[*unfolded cq divides-ff-def*, *folded hom-distribs*]
obtain *rr* **where** *cq*: $cq = a * rr$ **by** (*auto simp del*: *to-fract-hom.hom-mult simp*: *to-fract-hom.hom-mult*[*symmetric*])
with *one*(*1*) **have** $1 = a * (rr * c * cr)$ **by** (*auto simp*: *ac-simps*)

hence *a dvd 1* **..**
thus *?thesis* **by** (*simp add*: *q*)
next
assume *degree r* = *0*
from *degree0-coeffs*[*OF this*] *r* **obtain** *a* **where** *r*: *r* = [:*a*:] **and** *a*: *a* ≠ *0* **by** *auto*
hence *id*: *set* (*coeffs* (*?E r*)) = {*to-fract a*} **by** *auto*
have *divides-ff* (*to-fract a*) (*content-ff* (*?E r*)) **unfolding** *content-ff-iff id* **by** *auto*
note *this*[*unfolded cr divides-ff-def to-fract-mult*]
note *this*[*folded hom-distribs*]
then obtain *rr* **where** *cr*: *cr* = *a* ∗ *rr* **by** (*auto simp del*: *to-fract-hom.hom-mult simp*: *to-fract-hom.hom-mult*[*symmetric*])
with *one*(*2*) **have** *one*: *1* = *a* ∗ (*rr* ∗ *c* ∗ *cq*) **by** (*auto simp*: *ac-simps*)
from *arg-cong*[*OF pqr*[*unfolded r*], *of* λ *p*. *p* ∗ [:*rr* ∗ *c* ∗ *cq*:]]
have *p* ∗ [:*rr* ∗ *c* ∗ *cq*:] = *q* ∗ [:*a* ∗ (*rr* ∗ *c* ∗ *cq*):] **by** (*simp add*: *ac-simps*)
also have . . . = *q* **unfolding** *one*[*symmetric*] **by** *auto*
finally obtain *r* **where** *q* = *p* ∗ *r* **by** *blast*
hence *p dvd q* **..**
with *ndvd* **show** *?thesis* **by** *auto*
qed
}
ultimately show *?thesis* **by** (*auto simp*:*irreducible-altdef*)
qed

lemma *irreducible-cases*: *irreducible p* ⟷
degree p = *0* ∧ *irreducible* (*coeff p 0*) ∨
degree p ≠ *0* ∧ *irreducible* (*map-poly to-fract p*) ∧ *content-ff-ff p* =*dff 1*
using *irreducible-PM-M-PFM irreducible-M-PM irreducible-PFM-PM*
by *blast*

lemma *dvd-PM-iff*: *p dvd q* ⟷ *divides-ff* (*content-ff-ff p*) (*content-ff-ff q*) ∧
map-poly to-fract p dvd map-poly to-fract q
proof −
interpret *map-poly-inj-idom-hom to-fract*..
let *?E* = *map-poly to-fract*
show *?thesis* (**is** *?l* = *?r*)
proof
assume *p dvd q*
then obtain *r* **where** *qpr*: *q* = *p* ∗ *r* **..**
from *arg-cong*[*OF this*, *of ?E*]
have *dvd*: *?E p dvd ?E q* **by** *auto*
from *content-ff-map-poly-to-fract* **obtain** *cq* **where** *cq*: *content-ff-ff q* = *to-fract cq* **by** *auto*
from *content-ff-map-poly-to-fract* **obtain** *cp* **where** *cp*: *content-ff-ff p* = *to-fract cp* **by** *auto*
from *content-ff-map-poly-to-fract* **obtain** *cr* **where** *cr*: *content-ff-ff r* = *to-fract cr* **by** *auto*
from *gauss-lemma*[*of ?E p ?E r*, *folded hom-distribs qpr*, *unfolded cq cp cr*]

have *divides-ff* (*content-ff-ff p*) (*content-ff-ff q*) **unfolding** *cq cp eq-dff-def*
by (*metis divides-ff-def divides-ff-trans*)
with *dvd* **show** *?r* **by** *blast*
next
assume *?r*
show *?l*
proof (*cases* $q = 0$)
case *True*
with ‹*?r*› **show** *?l* **by** *auto*
next
case *False* **note** q = *this*
hence q': $?E\ q \neq 0$ **by** *auto*
from ‹*?r*› **obtain** *rr* **where** *qpr*: $?E\ q = ?E\ p * rr$ **unfolding** *dvd-def* **by** *auto*
with *q* **have** *p*: $p \neq 0$ **and** *Ep*: $?E\ p \neq 0$ **and** *rr*: $rr \neq 0$ **by** *auto*
from *gauss-lemma*[*of ?E p rr*, *folded qpr*]
have *ct*: *content-ff-ff q =dff content-ff-ff p* ∗ *content-ff rr*
by *auto*
from *content-ff-map-poly-to-fract*[*of p*] **obtain** *cp* **where** *cp*: *content-ff-ff p* = *to-fract cp* **by** *auto*
from *content-ff-map-poly-to-fract*[*of q*] **obtain** *cq* **where** *cq*: *content-ff-ff q* = *to-fract cq* **by** *auto*
from ‹*?r*›[*unfolded cp cq*] **have** *divides-ff* (*to-fract cp*) (*to-fract cq*) **..**
with *ct*[*unfolded cp cq eq-dff-def*] **have** *content-ff rr* ∈ *range to-fract*
by (*metis* (*no-types*, *lifting*) *Ep content-ff-0-iff cp divides-ff-def*
divides-ff-trans mult.commute mult-right-cancel range-eqI)
from *range-to-fract-embed-poly*[*OF content-ff-to-fract-coeffs-to-fract*[*OF this*]] **obtain** *r*
where *rr*: $rr = ?E\ r$ **by** *auto*
from *qpr*[*unfolded rr*, *folded hom-distribs*]
have $q = p * r$ **by** (*rule injectivity*)
thus *p dvd q* **..**
qed
qed
qed

lemma *factorial-monoid-poly*: *factorial-monoid* (*mk-monoid* :: *'a* :: *ufd poly monoid*)
proof (*fold factorial-condition-one*, *intro conjI*)
interpret *M*: *factorial-monoid mk-monoid* :: *'a monoid* **by** (*fact factorial-monoid*)
interpret *PFM*: *factorial-monoid mk-monoid* :: *'a fract poly monoid*
by (*rule as-ufd.factorial-monoid*)
interpret *PM*: *comm-monoid-cancel mk-monoid* :: *'a poly monoid* **by** (*unfold-locales*, *auto*)
let *?E* = *map-poly to-fract*
show *divisor-chain-condition-monoid* (*mk-monoid*::*'a poly monoid*)
proof (*unfold-locales*, *unfold mk-monoid-simps*)
let *?rel'* = $\{(x::'a\ poly,\ y).\ x \neq 0 \wedge y \neq 0 \wedge properfactor\ x\ y\}$
let *?rel''* = $\{(x::'a,\ y).\ x \neq 0 \wedge y \neq 0 \wedge properfactor\ x\ y\}$
let *?relPM* = $\{(x,\ y).\ x \neq 0 \wedge y \neq 0 \wedge x\ dvd\ y \wedge \neg\ y\ dvd\ (x :: 'a\ poly)\}$

let *?relM* = {(*x*, *y*). *x* ≠ *0* ∧ *y* ≠ *0* ∧ *x dvd y* ∧ ¬ *y dvd* (*x* :: *'a*)}
have *id*: *?rel'* = *?relPM* **using** *properfactor-nz* **by** *auto*
have *id'*: *?rel''* = *?relM* **using** *properfactor-nz* **by** *auto*
have *wf ?rel''* **using** *M.division-wellfounded* **by** *auto*
hence *wfM*: *wf ?relM* **using** *id'* **by** *auto*
let *?c* = λ *p*. *inv-embed* (*content-ff-ff p*)
let *?f* = λ *p*. (*degree p*, *?c p*)
note *wf* = *wf-inv-image*[*OF wf-lex-prod*[*OF wf-less wfM*], *of ?f*]
show *wf ?rel'* **unfolding** *id*
proof (*rule wf-subset*[*OF wf*], *clarify*)
fix *p q* :: *'a poly*
assume *p*: *p* ≠ *0* **and** *q*: *q* ≠ *0* **and** *dvd*: *p dvd q* **and** *ndvd*: ¬ *q dvd p*
from *dvd* **obtain** *r* **where** *qpr*: *q* = *p* ∗ *r* **..**
from *degree-mult-eq*[*of p r*, *folded qpr*] *q qpr* **have** *r*: *r* ≠ *0*
and *deg*: *degree q* = *degree p* + *degree r* **by** *auto*
show (*p*,*q*) ∈ *inv-image* ({(*x*, *y*). *x* < *y*} <∗*lex*∗> *?relM*) *?f*
proof (*cases degree p* = *degree q*)
case *False*
with *deg* **have** *degree p* < *degree q* **by** *auto*
thus *?thesis* **by** *auto*
next
case *True*
with *deg* **have** *degree r* = *0* **by** *simp*
from *degree0-coeffs*[*OF this*] *r* **obtain** *a* **where** *ra*: *r* = [:*a*:] **and** *a*: *a* ≠ *0* **by** *auto*
from *arg-cong*[*OF qpr*, *of* λ *p*. *?E p* ∗ [:*inverse* (*to-fract a*):]] *a*
have *?E p* = *?E q* ∗ [:*inverse* (*to-fract a*):]
by (*auto simp*: *ac-simps ra*)
hence *?E q dvd ?E p* **..**
with *ndvd dvd-PM-iff* **have** *ndvd*: ¬ *divides-ff* (*content-ff-ff q*) (*content-ff-ff p*) **by** *auto*
from *content-ff-map-poly-to-fract* **obtain** *cq* **where** *cq*: *content-ff-ff q* = *to-fract cq* **by** *auto*
from *content-ff-map-poly-to-fract* **obtain** *cp* **where** *cp*: *content-ff-ff p* = *to-fract cp* **by** *auto*
from *ndvd*[*unfolded cp cq*] **have** *ndvd*: ¬ *cq dvd cp* **by** *simp*
from *iffD1*[*OF dvd-PM-iff*,*OF dvd*,*unfolded cq cp*]
have *dvd*: *cp dvd cq* **by** *simp*
have *c-p*: *?c p* = *cp* **unfolding** *cp* **by** *simp*
have *c-q*: *?c q* = *cq* **unfolding** *cq* **by** *simp*
from *q cq* **have** *cq0*: *cq* ≠ *0* **by** *auto*
from *p cp* **have** *cp0*: *cp* ≠ *0* **by** *auto*
from *ndvd cq0 cp0 dvd* **have** (*?c p*, *?c q*) ∈ *?relM* **unfolding** *c-p c-q* **by** *auto*
with *True* **show** *?thesis* **by** *auto*
qed
qed
qed
show *primeness-condition-monoid* (*mk-monoid*::*'a poly monoid*)

proof (*unfold-locales*, *unfold mk-monoid-simps*)
fix *p* :: *'a poly*
assume *p*: *p* ≠ *0* **and** *irred p*
then have *irr*: *irreducible p* **by** *auto*
from *p* **have** *p'*: *?E p* ≠ *0* **by** *auto*
from *irreducible-PM-M-PFM*[*OF irr*] **have** *choice*: *degree p* = *0* ∧ *irred* (*coeff p 0*)
∨ *degree p* ≠ *0* ∧ *irred* (*?E p*) ∧ *content-ff-ff p* =*dff 1* **by** *auto*
show *Divisibility.prime mk-monoid p*
proof (*rule Divisibility.primeI*, *unfold mk-monoid-simps mem-Units*)
show ¬ *p dvd 1*
proof
assume *p dvd 1*
from *divides-degree*[*OF this*] **have** *dp*: *degree p* = *0* **by** *auto*
from *degree0-coeffs*[*OF this*] *p* **obtain** *a* **where** *p*: *p* = [:*a*:] **and** *a*: *a* ≠ *0* **by** *auto*
with *choice* **have** *irr*: *irreducible a* **by** *auto*
from ‹*p dvd 1*›[*unfolded p*] **have** *a dvd 1* **by** *auto*
with *irr* **show** *False* **unfolding** *irreducible-def* **by** *auto*
qed
fix *q r* :: *'a poly*
assume *q*: *q* ≠ *0* **and** *r*: *r* ≠ *0* **and** *factor p* (*q* ∗ *r*)
from *this*[*unfolded factor-idom*] **have** *p dvd q* ∗ *r* **by** *auto*
from *iffD1*[*OF dvd-PM-iff this*] **have** *dvd-ct*: *divides-ff* (*content-ff-ff p*) (*content-ff* (*?E* (*q* ∗ *r*)))
and *dvd-E*: *?E p dvd ?E q* ∗ *?E r* **by** *auto*
from *gauss-lemma*[*of ?E q ?E r*] *divides-ff-trans*[*OF dvd-ct*, *of content-ff-ff q* ∗ *content-ff-ff r*]
have *dvd-ct*: *divides-ff* (*content-ff-ff p*) (*content-ff-ff q* ∗ *content-ff-ff r*)
unfolding *eq-dff-def* **by** *auto*
from *choice*
have *p dvd q* ∨ *p dvd r*
proof
assume *degree p* ≠ *0* ∧ *irred* (*?E p*) ∧ *content-ff-ff p* =*dff 1*
hence *deg*: *degree p* ≠ *0* **and** *irr*: *irred* (*?E p*) **and** *ct*: *content-ff-ff p* =*dff 1* **by** *auto*
from *PFM.irreducible-prime*[*OF irr*] *p* **have** *prime*: *Divisibility.prime mk-monoid* (*?E p*) **by** *auto*
from *q r* **have** *Eq*: *?E q* ∈ *carrier mk-monoid* **and** *Er*: *?E r* ∈ *carrier mk-monoid*
and *q'*: *?E q* ≠ *0* **and** *r'*: *?E r* ≠ *0* **and** *qr'*: *?E q* ∗ *?E r* ≠ *0* **by** *auto*
from *PFM.prime-divides*[*OF Eq Er prime*] *q' r' qr' dvd-E*
have *dvd-E*: *?E p dvd ?E q* ∨ *?E p dvd ?E r* **by** *simp*
from *ct* **have** *ct*: *divides-ff* (*content-ff-ff p*) *1* **unfolding** *eq-dff-def* **by** *auto*
moreover have ⋀ *q*. *divides-ff 1* (*content-ff-ff q*) **using** *content-ff-map-poly-to-fract*
unfolding *divides-ff-def* **by** *auto*
from *divides-ff-trans*[*OF ct this*] **have** *ct*: ⋀ *q*. *divides-ff* (*content-ff-ff p*) (*content-ff-ff q*) **.**
with *dvd-E* **show** *?thesis* **using** *dvd-PM-iff* **by** *blast*

next
assume *degree p = 0 ∧ irred (coeff p 0)*
hence *deg*: *degree p = 0* **and** *irr*: *irred (coeff p 0)* **by** *auto*
from *degree0-coeffs[OF deg] p* **obtain** *a* **where** *p*: *p = [:a:]* **and** *a*: *a ≠ 0* **by** *auto*
with *irr* **have** *irr*: *irred a* **and** *aM*: *a ∈ carrier mk-monoid* **by** *auto*
from *M.irreducible-prime[OF irr aM]* **have** *prime*: *Divisibility.prime mk-monoid a* **.**
from *content-ff-map-poly-to-fract* **obtain** *cq* **where** *cq*: *content-ff-ff q = to-fract cq* **by** *auto*
from *content-ff-map-poly-to-fract* **obtain** *cp* **where** *cp*: *content-ff-ff p = to-fract cp* **by** *auto*
from *content-ff-map-poly-to-fract* **obtain** *cr* **where** *cr*: *content-ff-ff r = to-fract cr* **by** *auto*
have *divides-ff (to-fract a) (content-ff-ff p)* **unfolding** *p content-ff-iff* **using** *a* **by** *auto*
from *divides-ff-trans[OF this[unfolded cp] dvd-ct[unfolded cp cq cr]]*
have *divides-ff (to-fract a) (to-fract (cq ∗ cr))* **by** *simp*
hence *dvd*: *a dvd cq ∗ cr* **by** *(auto simp add: divides-ff-def simp del: to-fract-hom.hom-mult simp: to-fract-hom.hom-mult[symmetric])*
from *content-ff-divides-ff[of to-fract a ?E p]* **have** *divides-ff (to-fract cp) (to-fract a)*
using *cp a p* **by** *auto*
hence *cpa*: *cp dvd a* **by** *simp*
from *a q r cq cr* **have** *aM*: *a ∈ carrier mk-monoid* **and** *qM*: *cq ∈ carrier mk-monoid* **and** *rM*: *cr ∈ carrier mk-monoid*
and *q′*: *cq ≠ 0* **and** *r′*: *cr ≠ 0* **and** *qr′*: *cq ∗ cr ≠ 0*
by *auto*
from *M.prime-divides[OF qM rM prime] q′ r′ qr′ dvd*
have *a dvd cq ∨ a dvd cr* **by** *simp*
with *dvd-trans[OF cpa]* **have** *dvd*: *cp dvd cq ∨ cp dvd cr* **by** *auto*
have *⋀ q. ?E p ∗ (smult (inverse (to-fract a)) q) = q* **unfolding** *p* **using** *a* **by** *(auto simp: one-poly-def)*
hence *Edvd*: *⋀ q. ?E p dvd q* **unfolding** *dvd-def* **by** *metis*
from *dvd Edvd* **show** *?thesis* **apply** *(subst(1 2) dvd-PM-iff)* **unfolding** *cp cq cr* **by** *auto*
qed
thus *factor p q ∨ factor p r* **unfolding** *factor-idom* **using** *p q r* **by** *auto*
qed
qed
qed

instance *poly* :: *(ufd) ufd*
by *(intro class.ufd.of-class.intro factorial-monoid-imp-ufd factorial-monoid-poly)*

lemma *primitive-iff-some-content-dvd-1*:
fixes *f* :: *′a* :: *ufd poly*
shows *primitive f ⟷ some-gcd.listgcd (coeffs f) dvd 1* (**is** *- ⟷ ?c dvd 1*)

proof(*intro iffI primitiveI*)
  fix *x*
  assume ($\bigwedge y.\ y \in set\ (coeffs\ f) \Longrightarrow x\ dvd\ y$)
  from *some-gcd.listgcd-greatest*[*of coeffs f*, *OF this*]
  have *x dvd ?c* **by** *simp*
  also assume *?c dvd 1*
  finally show *x dvd 1* **.**
next
  assume *primitive f*
  from *primitiveD*[*OF this some-gcd.listgcd*[*of - coeffs f*]]
  show *?c dvd 1* **by** *auto*
qed

end

5 Polynomials in Rings and Fields

5.1 Polynomials in Rings

We use a locale to work with polynomials in some integer-modulo ring.

theory *Poly-Mod*
  imports
  HOL−Computational-Algebra.Primes
  Polynomial-Factorization.Square-Free-Factorization
  Unique-Factorization-Poly
begin

locale *poly-mod* = **fixes** *m* :: *int*
begin

definition *M* :: *int* ⇒ *int* **where** *M x* = *x mod m*

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

lemma *M-M*[*simp*]: *M* (*M x*) = *M x*
  by (*auto simp add*: *M-def*)

lemma *M-plus*[*simp*]: $M\ (M\ x + y) = M\ (x + y)$ $M\ (x + M\ y) = M\ (x + y)$
  by (*auto simp add*: *M-def mod-simps*)

lemma *M-minus*[*simp*]: $M\ (M\ x - y) = M\ (x - y)$ $M\ (x - M\ y) = M\ (x - y)$
  by (*auto simp add*: *M-def mod-simps*)

lemma *M-times*[*simp*]: $M\ (M\ x * y) = M\ (x * y)$ $M\ (x * M\ y) = M\ (x * y)$
  by (*auto simp add*: *M-def mod-simps*)

lemma *M-sum*: $M\ (sum\ (\lambda\ x.\ M\ (f\ x))\ A) = M\ (sum\ f\ A)$
proof (*induct A rule*: *infinite-finite-induct*)

case (*insert x A*)
from *insert*(*1−2*) **have** M ($\sum x{\in}insert\ x\ A.\ M\ (f\ x)$) = M ($f\,x + M$ (($\sum x{\in}A.$ $M\ (f\ x)$))) **by** *simp*
also have M (($\sum x{\in}A.\ M\ (f\ x)$)) = M (($\sum x{\in}A.\ f\ x$)) **using** *insert* **by** *simp*
finally show *?case* **using** *insert* **by** *simp*
qed *auto*

definition *inv-M* :: *int* ⇒ *int* **where**
inv-M = (λ *x. if x + x* ≤ *m then x else x* − *m*)

lemma *M-inv-M-id*[*simp*]: *M* (*inv-M x*) = *M x*
unfolding *inv-M-def M-def* **by** *simp*

definition *Mp* :: *int poly* ⇒ *int poly* **where** *Mp* = *map-poly M*

lemma *Mp-0*[*simp*]: *Mp 0* = *0* **unfolding** *Mp-def* **by** *auto*

lemma *Mp-coeff*: *coeff* (*Mp f*) *i* = *M* (*coeff f i*) **unfolding** *Mp-def*
by (*simp add*: *M-def coeff-map-poly*)

abbreviation *eq-m* :: *int poly* ⇒ *int poly* ⇒ *bool* (**infixl** =*m 50*) **where**
f =*m g* ≡ (*Mp f* = *Mp g*)

notation *eq-m* (**infixl** =*m 50*)

abbreviation *degree-m* :: *int poly* ⇒ *nat* **where**
degree-m f ≡ *degree* (*Mp f*)

lemma *mult-Mp*[*simp*]: *Mp* (*Mp f* ∗ *g*) = *Mp* (*f* ∗ *g*) *Mp* (*f* ∗ *Mp g*) = *Mp* (*f* ∗ *g*)
proof −
{
fix *f g*
have *Mp* (*Mp f* ∗ *g*) = *Mp* (*f* ∗ *g*)
unfolding *poly-eq-iff Mp-coeff* **unfolding** *coeff-mult Mp-coeff*
proof
fix *n*
show M ($\sum i{\le}n.\ M$ (*coeff f i*) ∗ *coeff g* (*n* − *i*)) = M ($\sum i{\le}n.$ *coeff f i* ∗ *coeff g* (*n* − *i*))
by (*subst M-sum*[*symmetric*], *rule sym*, *subst M-sum*[*symmetric*], *unfold M-times*, *simp*)
qed
}
from *this*[*of f g*] *this*[*of g f*] **show** *Mp* (*Mp f* ∗ *g*) = *Mp* (*f* ∗ *g*) *Mp* (*f* ∗ *Mp g*) = *Mp* (*f* ∗ *g*)
by (*auto simp*: *ac-simps*)
qed

lemma *plus-Mp*[*simp*]: *Mp* (*Mp f* + *g*) = *Mp* (*f* + *g*) *Mp* (*f* + *Mp g*) = *Mp* (*f* + *g*)
unfolding *poly-eq-iff Mp-coeff* **unfolding** *coeff-mult Mp-coeff* **by** (*auto simp add*: *Mp-coeff*)

lemma *minus-Mp*[*simp*]: *Mp* (*Mp f* − *g*) = *Mp* (*f* − *g*) *Mp* (*f* − *Mp g*) = *Mp* (*f* − *g*)
unfolding *poly-eq-iff Mp-coeff* **unfolding** *coeff-mult Mp-coeff* **by** (*auto simp add*: *Mp-coeff*)

lemma *Mp-smult*[*simp*]: *Mp* (*smult* (*M a*) *f*) = *Mp* (*smult a f*) *Mp* (*smult a* (*Mp f*)) = *Mp* (*smult a f*)
unfolding *Mp-def smult-as-map-poly*
by (*rule poly-eqI*, *auto simp*: *coeff-map-poly*)+

lemma *Mp-Mp*[*simp*]: *Mp* (*Mp f*) = *Mp f* **unfolding** *Mp-def*
by (*intro poly-eqI*, *auto simp*: *coeff-map-poly*)

lemma *Mp-smult-m-0*[*simp*]: *Mp* (*smult m f*) = *0*
by (*intro poly-eqI*, *auto simp*: *Mp-coeff*, *auto simp*: *M-def*)

definition *dvdm* :: *int poly* ⇒ *int poly* ⇒ *bool* (**infix** *dvdm 50*) **where**
f dvdm g = (∃ *h*. *g* =*m f* ∗ *h*)
notation *dvdm* (**infix** *dvdm 50*)

lemma *dvdmE*:
assumes *fg*: *f dvdm g*
and *main*: ⋀*h*. *g* =*m f* ∗ *h* ⟹ *Mp h* = *h* ⟹ *thesis*
shows *thesis*
proof−
from *fg* **obtain** *h* **where** *g* =*m f* ∗ *h* **by** (*auto simp*: *dvdm-def*)
then have *g* =*m f* ∗ *Mp h* **by** *auto*
from *main*[*OF this*] **show** *thesis* **by** *auto*
qed

lemma *Mp-dvdm*[*simp*]: *Mp f dvdm g* ⟷ *f dvdm g*
and *dvdm-Mp*[*simp*]: *f dvdm Mp g* ⟷ *f dvdm g* **by** (*auto simp*: *dvdm-def*)

definition *irreducible-m*
where *irreducible-m f* = (¬*f* =*m 0* ∧ ¬ *f dvdm 1* ∧ (∀ *a b*. *f* =*m a* ∗ *b* ⟶ *a dvdm 1* ∨ *b dvdm 1*))

definition $irreducible_d$*-m* :: *int poly* ⇒ *bool* **where** $irreducible_d$*-m f* ≡
degree-m f > *0* ∧
(∀ *g h*. *degree-m g* < *degree-m f* ⟶ *degree-m h* < *degree-m f* ⟶ ¬ *f* =*m g* ∗ *h*)

definition *prime-elem-m*

where *prime-elem-m f* $\equiv \neg$ *f* =*m 0* $\wedge \neg$ *f dvdm 1* $\wedge$ ($\forall$ *g h*. *f dvdm g* $*$ *h* $\longrightarrow$ *f dvdm g* $\vee$ *f dvdm h*)

lemma *degree-m-le-degree* [*intro*!]: *degree-m f* $\leq$ *degree f*
by (*simp add*: *Mp-def degree-map-poly-le*)

lemma *irreducible*$_d$*-mI*:
assumes *f0*: *degree-m f* $>$ *0*
and *main*: $\bigwedge$*g h*. *Mp g* = *g* $\Longrightarrow$ *Mp h* = *h* $\Longrightarrow$ *degree g* $>$ *0* $\Longrightarrow$ *degree g* $<$ *degree-m f* $\Longrightarrow$ *degree h* $>$ *0* $\Longrightarrow$ *degree h* $<$ *degree-m f* $\Longrightarrow$ *f* =*m g* $*$ *h* $\Longrightarrow$ *False*
shows *irreducible*$_d$*-m f*
proof (*unfold irreducible*$_d$*-m-def*, *intro conjI allI impI f0 notI*)
fix *g h*
assume *deg*: *degree-m g* $<$ *degree-m f degree-m h* $<$ *degree-m f* **and** *f* =*m g* $*$ *h*
then have *f*: *f* =*m Mp g* $*$ *Mp h* **by** *simp*
have *degree-m f* $\leq$ *degree-m g* + *degree-m h*
unfolding *f* **using** *degree-mult-le order.trans* **by** *blast*
with *main*[*of Mp g Mp h*] *deg f* **show** *False* **by** *auto*
qed

lemma *irreducible*$_d$*-mE*:
assumes *irreducible*$_d$*-m f*
and *degree-m f* $>$ *0* $\Longrightarrow$ ($\bigwedge$*g h*. *degree-m g* $<$ *degree-m f* $\Longrightarrow$ *degree-m h* $<$ *degree-m f* $\Longrightarrow \neg$ *f* =*m g* $*$ *h*) $\Longrightarrow$ *thesis*
shows *thesis*
using *assms* **by** (*unfold irreducible*$_d$*-m-def*, *auto*)

lemma *irreducible*$_d$*-mD*:
assumes *irreducible*$_d$*-m f*
shows *degree-m f* $>$ *0* **and** $\bigwedge$*g h*. *degree-m g* $<$ *degree-m f* $\Longrightarrow$ *degree-m h* $<$ *degree-m f* $\Longrightarrow \neg$ *f* =*m g* $*$ *h*
using *assms* **by** (*auto elim*: *irreducible*$_d$*-mE*)

definition *square-free-m* :: *int poly* $\Rightarrow$ *bool* **where**
square-free-m f = ($\neg$ *f* =*m 0* $\wedge$ ($\forall$ *g*. *degree-m g* $\neq$ *0* $\longrightarrow \neg$ (*g* $*$ *g dvdm f*)))

definition *coprime-m* :: *int poly* $\Rightarrow$ *int poly* $\Rightarrow$ *bool* **where**
coprime-m f g = ($\forall$ *h*. *h dvdm f* $\longrightarrow$ *h dvdm g* $\longrightarrow$ *h dvdm 1*)

lemma *Mp-square-free-m*[*simp*]: *square-free-m* (*Mp f*) = *square-free-m f*
unfolding *square-free-m-def dvdm-def* **by** *simp*

lemma *square-free-m-cong*: *square-free-m f* $\Longrightarrow$ *Mp f* = *Mp g* $\Longrightarrow$ *square-free-m g*
unfolding *square-free-m-def dvdm-def* **by** *simp*

lemma *Mp-prod-mset*[*simp*]: *Mp* (*prod-mset* (*image-mset Mp b*)) = *Mp* (*prod-mset b*)
proof (*induct b*)

case (*add x b*)
have *Mp* (*prod-mset* (*image-mset Mp* ({#*x*#}+*b*))) = *Mp* (*Mp x* ∗ *prod-mset* (*image-mset Mp b*)) **by** *simp*
also have … = *Mp* (*Mp x* ∗ *Mp* (*prod-mset* (*image-mset Mp b*))) **by** *simp*
also have … = *Mp* (*Mp x* ∗ *Mp* (*prod-mset b*)) **unfolding** *add* **by** *simp*
finally show *?case* **by** *simp*
qed *simp*

lemma *Mp-prod-list*: *Mp* (*prod-list* (*map Mp b*)) = *Mp* (*prod-list b*)
proof (*induct b*)
case (*Cons b xs*)
have *Mp* (*prod-list* (*map Mp* (*b* # *xs*))) = *Mp* (*Mp b* ∗ *prod-list* (*map Mp xs*)) **by** *simp*
also have … = *Mp* (*Mp b* ∗ *Mp* (*prod-list* (*map Mp xs*))) **by** *simp*
also have … = *Mp* (*Mp b* ∗ *Mp* (*prod-list xs*)) **unfolding** *Cons* **by** *simp*
finally show *?case* **by** *simp*
qed *simp*

Polynomial evaluation modulo

definition *M-poly p x* ≡ *M* (*poly p x*)

lemma *M-poly-Mp*[*simp*]: *M-poly* (*Mp p*) = *M-poly p*
proof(*intro ext*, *induct p*)
case *0* **show** *?case* **by** *auto*
next
case *IH*: (*pCons a p*)
from *IH*(*1*) **have** *M-poly* (*Mp* (*pCons a p*)) *x* = *M* (*a* + *M*(*x* ∗ *M-poly* (*Mp p*) *x*))
by (*simp add*: *M-poly-def Mp-def*)
also note *IH*(*2*)[*of x*]
finally show *?case* **by** (*simp add*: *M-poly-def*)
qed

lemma *Mp-lift-modulus*: **assumes** *f* =*m g*
shows *poly-mod.eq-m* (*m* ∗ *k*) (*smult k f*) (*smult k g*)
using *assms* **unfolding** *poly-eq-iff poly-mod.Mp-coeff coeff-smult*
unfolding *poly-mod.M-def* **by** *simp*

lemma *Mp-ident-product*: *n* > *0* ⟹ *Mp f* = *f* ⟹ *poly-mod.Mp* (*m* ∗ *n*) *f* = *f*
unfolding *poly-eq-iff poly-mod.Mp-coeff poly-mod.M-def*
by (*auto simp add*: *zmod-zmult2-eq*) (*metis mod-div-trivial mod-0*)

lemma *Mp-shrink-modulus*: **assumes** *poly-mod.eq-m* (*m* ∗ *k*) *f g k* ≠ *0*
shows *f* =*m g*
proof −
from *assms* **have** *a*: ⋀ *n. coeff f n mod* (*m* ∗ *k*) = *coeff g n mod* (*m* ∗ *k*)
unfolding *poly-eq-iff poly-mod.Mp-coeff* **unfolding** *poly-mod.M-def* **by** *auto*
show *?thesis* **unfolding** *poly-eq-iff poly-mod.Mp-coeff* **unfolding** *poly-mod.M-def*
proof

fix *n*
show *coeff f n mod m = coeff g n mod m* **using** *a*[*of n*] ‹*k ≠ 0*›
by (*metis mod-mult-right-eq mult.commute mult-cancel-left mult-mod-right*)
qed
qed

lemma *degree-m-le*: *degree-m f ≤ degree f* **unfolding** *Mp-def* **by** (*rule degree-map-poly-le*)

lemma *degree-m-eq*: *coeff f* (*degree f*) *mod m ≠ 0 ⟹ m > 1 ⟹ degree-m f = degree f*
using *degree-m-le*[*of f*] **unfolding** *Mp-def*
by (*auto intro*: *degree-map-poly simp*: *Mp-def poly-mod.M-def*)

lemma *degree-m-mult-le*:
assumes *eq*: *f =m g ∗ h*
shows *degree-m f ≤ degree-m g + degree-m h*
proof −
have *degree-m f = degree-m* (*Mp g ∗ Mp h*) **using** *eq* **by** *simp*
also have . . . *≤ degree* (*Mp g ∗ Mp h*) **by** (*rule degree-m-le*)
also have . . . *≤ degree-m g + degree-m h* **by** (*rule degree-mult-le*)
finally show *?thesis* **by** *auto*
qed

lemma *degree-m-smult-le*: *degree-m* (*smult c f*) *≤ degree-m f*
by (*metis Mp-0 coeff-0 degree-le degree-m-le degree-smult-eq poly-mod.Mp-smult*(*2*) *smult-eq-0-iff*)

lemma *irreducible-m-Mp*[*simp*]: *irreducible-m* (*Mp f*) *⟷ irreducible-m f* **by** (*simp add*: *irreducible-m-def*)

lemma *eq-m-irreducible-m*: *f =m g ⟹ irreducible-m f ⟷ irreducible-m g*
using *irreducible-m-Mp* **by** *metis*

definition *mset-factors-m* **where** *mset-factors-m F p ≡*
F ≠ {#} ∧ (∀ f. f ∈# F ⟶ irreducible-m f) ∧ p =m prod-mset F

end

declare *poly-mod.M-def*[*code*]
declare *poly-mod.Mp-def*[*code*]
declare *poly-mod.inv-M-def*[*code*]

definition *Irr-Mon* :: *'a* :: *comm-semiring-1 poly set*
where *Irr-Mon* = {*x. irreducible x ∧ monic x*}

definition *factorization* :: *'a* :: *comm-semiring-1 poly set ⇒ 'a poly ⇒* (*'a × 'a poly multiset*) *⇒ bool* **where**
factorization Factors f cfs ≡ (*case cfs of* (*c,fs*) *⇒ f =* (*smult c* (*prod-mset fs*)) *∧*

(*set-mset fs* ⊆ *Factors*))

definition *unique-factorization* :: *'a* :: *comm-semiring-1 poly set* ⇒ *'a poly* ⇒ (*'a* × *'a poly multiset*) ⇒ *bool* **where**
unique-factorization Factors f cfs = (*Collect* (*factorization Factors f*) = {*cfs*})

lemma *irreducible-multD*:
assumes *l*: *irreducible* (*a*∗*b*)
shows *a dvd 1* ∧ *irreducible b* ∨ *b dvd 1* ∧ *irreducible a*
proof−
from *l* **have** *a dvd 1* ∨ *b dvd 1* **by** *auto*
then show *?thesis*
proof(*elim disjE*)
assume *a*: *a dvd 1*
with *l* **have** *irreducible b*
unfolding *irreducible-def*
by (*meson is-unit-mult-iff mult.left-commute mult-not-zero*)
with *a* **show** *?thesis* **by** *auto*
next
assume *a*: *b dvd 1*
with *l* **have** *irreducible a*
unfolding *irreducible-def*
by (*meson is-unit-mult-iff mult-not-zero semiring-normalization-rules*(*16*))
with *a* **show** *?thesis* **by** *auto*
qed
qed

lemma *irreducible-dvd-prod-mset*:
fixes *p* :: *'a* :: *field poly*
assumes *irr*: *irreducible p* **and** *dvd*: *p dvd prod-mset as*
shows ∃ *a* ∈# *as*. *p dvd a*
proof −
from *irr*[*unfolded irreducible-def*] **have** *deg*: *degree p* ≠ *0* **by** *auto*
hence *p1*: ¬ *p dvd 1* **unfolding** *dvd-def*
by (*metis degree-1 nonzero-mult-div-cancel-left div-poly-less linorder-neqE-nat mult-not-zero not-less0 zero-neq-one*)
from *dvd* **show** *?thesis*
proof (*induct as*)
case (*add a as*)
hence *prod-mset* (*add-mset a as*) = *a* ∗ *prod-mset as* **by** *auto*
from *add*(*2*)[*unfolded this*] *add*(*1*) *irr*
show *?case* **by** *auto*
qed (*insert p1*, *auto*)
qed

lemma *monic-factorization-unique-mset*:
fixes *P*::*'a*::*field poly multiset*
assumes *eq*: *prod-mset P* = *prod-mset Q*
and *P*: *set-mset P* ⊆ {*q*. *irreducible q* ∧ *monic q*}

  and Q: *set-mset* $Q \subseteq \{q.$ *irreducible* $q \wedge$ *monic* $q\}$
 shows $P = Q$
proof −
 {
  fix P Q :: *'a poly multiset*
  assume *id*: *prod-mset* P = *prod-mset* Q
  and P: *set-mset* $P \subseteq \{q.$ *irreducible* $q \wedge$ *monic* $q\}$
  and Q: *set-mset* $Q \subseteq \{q.$ *irreducible* $q \wedge$ *monic* $q\}$
  hence $P \subseteq\# Q$
  proof (*induct P arbitrary*: Q)
   case (*add x P Q'*)
   from *add*(*3*) **have** *irr*: *irreducible x* **and** *mon*: *monic x* **by** *auto*
   have $\exists\ a \in\# Q'.\ x$ *dvd a*
   proof (*rule irreducible-dvd-prod-mset*[*OF irr*])
    show *x dvd prod-mset Q'* **unfolding** *add*(*2*)[*symmetric*] **by** *simp*
   qed
   then obtain *y Q* **where** *Q'*: $Q' =$ *add-mset y Q* **and** *xy*: *x dvd y* **by** (*meson mset-add*)
   from *add*(*4*) *Q'* **have** *irr'*: *irreducible y* **and** *mon'*: *monic y* **by** *auto*
   have $x = y$ **using** *irr irr' xy mon mon'*
    by (*metis irreducibleD' irreducible-not-unit poly-dvd-antisym*)
   hence *Q'*: $Q' = Q + \{\#x\#\}$ **using** *Q'* **by** *auto*
   from *mon* **have** *x0*: $x \neq 0$ **by** *auto*
   from *arg-cong*[*OF add*(*2*)[*unfolded Q'*], *of* λ *z. z div x*]
   have *eq*: *prod-mset P* = *prod-mset Q* **using** *x0* **by** *auto*
   from *add*(*3*−*4*)[*unfolded Q'*]
   have *set-mset* $P \subseteq \{q.$ *irreducible* $q \wedge$ *monic* $q\}$ *set-mset* $Q \subseteq \{q.$ *irreducible* $q \wedge$ *monic* $q\}$
    by *auto*
   from *add*(*1*)[*OF eq this*] **show** *?case* **unfolding** *Q'* **by** *auto*
  qed *auto*
 }
 from *this*[*OF eq P Q*] *this*[*OF eq*[*symmetric*] *Q P*]
 show *?thesis* **by** *auto*
qed

lemma *exactly-one-monic-factorization*:
 assumes *mon*: *monic* (*f* :: *'a* :: *field poly*)
 shows $\exists!$ *fs*. $f =$ *prod-mset fs* $\wedge$ *set-mset fs* $\subseteq \{q.$ *irreducible* $q \wedge$ *monic* $q\}$
proof −
 from *monic-irreducible-factorization*[*OF mon*]
 obtain *gs g* **where** *fin*: *finite gs* **and** *f*: $f = (\prod a \in gs.\ a$ ^ *Suc* $(g\ a))$
  and *gs*: $gs \subseteq \{q.$ *irreducible* $q \wedge$ *monic* $q\}$
  by *blast*
 from *fin*
 have $\exists$ *fs*. *set-mset fs* $\subseteq gs \wedge$ *prod-mset fs* $= (\prod a \in gs.\ a$ ^ *Suc* $(g\ a))$
 proof (*induct gs*)
  case (*insert a gs*)

from *insert*(*3*) **obtain** *fs* **where** ∗: *set-mset fs* ⊆ *gs prod-mset fs* = (∏ *a*∈*gs*. *a* ^ *Suc* (*g a*)) **by** *auto*
let *?fs* = *fs* + *replicate-mset* (*Suc* (*g a*)) *a*
show *?case*
proof (*rule exI*[*of* - *fs* + *replicate-mset* (*Suc* (*g a*)) *a*], *intro conjI*)
show *set-mset ?fs* ⊆ *insert a gs* **using** ∗(*1*) **by** *auto*
show *prod-mset ?fs* = (∏ *a*∈*insert a gs*. *a* ^ *Suc* (*g a*))
by (*subst prod.insert*[*OF insert*(*1*−*2*)], *auto simp*: ∗(*2*))
qed
qed *simp*
then obtain *fs* **where** *set-mset fs* ⊆ *gs prod-mset fs* = (∏ *a*∈*gs*. *a* ^ *Suc* (*g a*)) **by** *auto*
with *gs f* **have** *ex*: ∃ *fs*. *f* = *prod-mset fs* ∧ *set-mset fs* ⊆ {*q*. *irreducible q* ∧ *monic q*}
by (*intro exI*[*of* - *fs*], *auto*)
thus *?thesis* **using** *monic-factorization-unique-mset* **by** *blast*
qed

lemma *monic-prod-mset*:
fixes *as* :: ′*a* :: *idom poly multiset*
assumes ⋀ *a*. *a* ∈ *set-mset as* ⟹ *monic a*
shows *monic* (*prod-mset as*) **using** *assms*
by (*induct as*, *auto intro*: *monic-mult*)

lemma *exactly-one-factorization*:
assumes *f*: *f* ≠ (*0* :: ′*a* :: *field poly*)
shows ∃! *cfs*. *factorization Irr-Mon f cfs*
proof −
let *?a* = *coeff f* (*degree f*)
let *?b* = *inverse ?a*
let *?g* = *smult ?b f*
define *g* **where** *g* = *?g*
from *f* **have** *a*: *?a* ≠ *0 ?b* ≠ *0* **by** (*auto simp*: *field-simps*)
hence *monic g* **unfolding** *g-def* **by** *simp*
note *ex1* = *exactly-one-monic-factorization*[*OF this*, *folded Irr-Mon-def*]
then obtain *fs* **where** *g*: *g* = *prod-mset fs set-mset fs* ⊆ *Irr-Mon* **by** *auto*
let *?cfs* = (*?a*,*fs*)
have *cfs*: *factorization Irr-Mon f ?cfs* **unfolding** *factorization-def split g*(*1*)[*symmetric*]
using *g*(*2*) **unfolding** *g-def* **by** (*simp add*: *a field-simps*)
show *?thesis*
proof (*rule*, *rule cfs*)
fix *dgs*
assume *fact*: *factorization Irr-Mon f dgs*
obtain *d gs* **where** *dgs*: *dgs* = (*d*,*gs*) **by** *force*
from *fact*[*unfolded factorization-def dgs split*]
have *fd*: *f* = *smult d* (*prod-mset gs*) **and** *gs*: *set-mset gs* ⊆ *Irr-Mon* **by** *auto*
have *monic* (*prod-mset gs*) **by** (*rule monic-prod-mset*, *insert gs*[*unfolded Irr-Mon-def*], *auto*)
hence *d*: *d* = *?a* **unfolding** *fd* **by** *auto*

from *arg-cong*[*OF fd*, *of* λ *x*. *smult ?b x*, *unfolded d g-def*[*symmetric*]]
have *g* = *prod-mset gs* **using** *a* **by** (*simp add*: *field-simps*)
with *ex1 g gs* **have** *gs* = *fs* **by** *auto*
thus *dgs* = *?cfs* **unfolding** *dgs d* **by** *auto*
qed
qed

lemma *mod-ident-iff*: $m > 0 \Longrightarrow (x :: int)\ mod\ m = x \longleftrightarrow x \in \{0 ..< m\}$
by (*metis Divides.pos-mod-bound Divides.pos-mod-sign atLeastLessThan-iff mod-pos-pos-trivial*)

declare *prod-mset-prod-list*[*simp*]

lemma *mult-1-is-id*[*simp*]: $(*)\ (1 :: 'a :: ring\text{-}1) = id$ **by** *auto*

context *poly-mod*
begin

lemma *degree-m-eq-monic*: *monic f* $\Longrightarrow m > 1 \Longrightarrow$ *degree-m f* = *degree f*
by (*rule degree-m-eq*) *auto*

lemma *monic-degree-m-lift*: **assumes** *monic f* $k > 1$ $m > 1$
shows *monic* (*poly-mod.Mp* $(m * k)$ *f*)
proof −
have *deg*: *degree* (*poly-mod.Mp* $(m * k)$ *f*) = *degree f*
by (*rule poly-mod.degree-m-eq-monic*[*of f* $m * k$], *insert assms*, *auto simp*: *less-1-mult*)
show *?thesis* **unfolding** *poly-mod.Mp-coeff deg assms poly-mod.M-def* **using** *assms*(*2*−)
by (*simp add*: *less-1-mult*)
qed

end

locale *poly-mod-2* = *poly-mod m* **for** *m* +
assumes *m1*: $m > 1$
begin

lemma *M-1*[*simp*]: *M 1* = *1* **unfolding** *M-def* **using** *m1*
by *auto*

lemma *Mp-1*[*simp*]: *Mp 1* = *1* **unfolding** *Mp-def* **by** *simp*

lemma *monic-degree-m*[*simp*]: *monic f* $\Longrightarrow$ *degree-m f* = *degree f*
using *degree-m-eq-monic*[*of f*] **using** *m1* **by** *auto*

lemma *monic-Mp*: *monic f* $\Longrightarrow$ *monic* (*Mp f*)
by (*auto simp*: *Mp-coeff*)

lemma *Mp-0-smult-sdiv-poly*: **assumes** *Mp f = 0*
  shows *smult m (sdiv-poly f m) = f*
proof (*intro poly-eqI*, *unfold Mp-coeff coeff-smult sdiv-poly-def*, *subst coeff-map-poly*, *force*)
  fix *n*
  from *assms* **have** *coeff (Mp f) n = 0* **by** *simp*
  hence *0*: *coeff f n mod m = 0* **unfolding** *Mp-coeff M-def* **.**
  thus *m ∗ (coeff f n div m) = coeff f n* **by** *auto*
qed

lemma *Mp-product-modulus*: *m′ = m ∗ k ⟹ k > 0 ⟹ Mp (poly-mod.Mp m′ f) = Mp f*
  by (*intro poly-eqI*, *unfold poly-mod.Mp-coeff poly-mod.M-def*, *auto simp*: *mod-mod-cancel*)

lemma *inv-M-rev*: **assumes** *bnd*: *2 ∗ abs c < m*
  shows *inv-M (M c) = c*
proof (*cases c ≥ 0*)
  case *True*
  with *bnd* **show** *?thesis* **unfolding** *M-def inv-M-def* **by** *auto*
next
  case *False*
  have *2*: *⋀ v :: int. 2 ∗ v = v + v* **by** *auto*
  from *False* **have** *c*: *c < 0* **by** *auto*
  from *bnd c* **have** *c + m > 0 c + m < m* **by** *auto*
  with *c* **have** *cm*: *c mod m = c + m*
    by (*metis le-less mod-add-self2 mod-pos-pos-trivial*)
  from *c bnd* **have** *2 ∗ (c mod m) > m* **unfolding** *cm* **by** *auto*
  with *bnd c* **show** *?thesis* **unfolding** *M-def inv-M-def cm* **by** *auto*
qed

end

lemma (**in** *poly-mod*) *degree-m-eq-prime*:
  assumes *f0*: *Mp f ≠ 0*
  and *deg*: *degree-m f = degree f*
  and *eq*: *f =m g ∗ h*
  and *p*: *prime m*
  shows *degree-m f = degree-m g + degree-m h*
proof −
  interpret *poly-mod-2 m* **using** *prime-ge-2-int*[*OF p*] **unfolding** *poly-mod-2-def* **by** *simp*
  from *f0 eq* **have** *Mp (Mp g ∗ Mp h) ≠ 0* **by** *auto*
  hence *Mp g ∗ Mp h ≠ 0* **using** *Mp-0* **by** (*cases Mp g ∗ Mp h*, *auto*)
  hence *g0*: *Mp g ≠ 0* **and** *h0*: *Mp h ≠ 0* **by** *auto*
  have *degree (Mp (g ∗ h)) = degree-m (Mp g ∗ Mp h)* **by** *simp*
  also have ... *= degree (Mp g ∗ Mp h)*
  proof (*rule degree-m-eq*[*OF - m1*], *rule*)
    have *id*: *⋀ g. coeff (Mp g) (degree (Mp g)) mod m = coeff (Mp g) (degree (Mp*

g))
  unfolding *M-def*[*symmetric*] *Mp-coeff* **by** *simp*
  from *p* **have** *p'*: *prime m* **unfolding** *prime-int-nat-transfer* **unfolding** *prime-nat-iff* **by** *auto*
  assume *coeff* (*Mp g* ∗ *Mp h*) (*degree* (*Mp g* ∗ *Mp h*)) *mod m* = *0*
  from *this*[*unfolded coeff-degree-mult*]
  have *coeff* (*Mp g*) (*degree* (*Mp g*)) *mod m* = *0* ∨ *coeff* (*Mp h*) (*degree* (*Mp h*)) *mod m* = *0*
  unfolding *dvd-eq-mod-eq-0*[*symmetric*] **using** *m1 prime-dvd-mult-int*[*OF p'*] **by** *auto*
  with *g0 h0* **show** *False* **unfolding** *id* **by** *auto*
qed
also have ... = *degree* (*Mp g*) + *degree* (*Mp h*)
  by (*rule degree-mult-eq*[*OF g0 h0*])
finally show *?thesis* **using** *eq* **by** *simp*
qed

lemma *monic-smult-add-small*: **assumes** *f* = *0* ∨ *degree f* < *degree g* **and** *mon*: *monic g*
shows *monic* (*g* + *smult q f*)
proof (*cases f* = *0*)
case *True*
thus *?thesis* **using** *mon* **by** *auto*
next
case *False*
with *assms* **have** *degree f* < *degree g* **by** *auto*
hence *degree* (*smult q f*) < *degree g* **by** (*meson degree-smult-le not-less order-trans*)
thus *?thesis* **using** *mon* **using** *coeff-eq-0 degree-add-eq-left* **by** *fastforce*
qed

context *poly-mod*
begin

definition *factorization-m* :: *int poly* ⇒ (*int* × *int poly multiset*) ⇒ *bool* **where**
factorization-m f cfs ≡ (*case cfs of* (*c*,*fs*) ⇒ *f* =*m* (*smult c* (*prod-mset fs*)) ∧
(∀ *f* ∈ *set-mset fs*. *irreducible*$_d$-*m f* ∧ *monic* (*Mp f*)))

definition *Mf* :: *int* × *int poly multiset* ⇒ *int* × *int poly multiset* **where**
Mf cfs ≡ *case cfs of* (*c*,*fs*) ⇒ (*M c*, *image-mset Mp fs*)

lemma *Mf-Mf*[*simp*]: *Mf* (*Mf x*) = *Mf x*
proof (*cases x*, *auto simp*: *Mf-def*, *goal-cases*)
case (*1 c fs*)
show *?case* **by** (*induct fs*, *auto*)
qed

definition *equivalent-fact-m* :: *int* × *int poly multiset* ⇒ *int* × *int poly multiset* ⇒ *bool* **where**

equivalent-fact-m cfs dgs = (*Mf cfs* = *Mf dgs*)

definition *unique-factorization-m* :: *int poly* ⇒ (*int* × *int poly multiset*) ⇒ *bool*
where
unique-factorization-m f cfs = (*Mf* ‘ *Collect* (*factorization-m f*) = {*Mf cfs*})

lemma *Mp-irreducible*$_d$*-m*[*simp*]: *irreducible*$_d$*-m* (*Mp f*) = *irreducible*$_d$*-m f*
unfolding *irreducible*$_d$*-m-def dvdm-def* **by** *simp*

lemma *Mf-factorization-m*[*simp*]: *factorization-m f* (*Mf cfs*) = *factorization-m f cfs*
unfolding *factorization-m-def Mf-def*
proof (*cases cfs*, *simp*, *goal-cases*)
case (*1 c fs*)
have *Mp* (*smult c* (*prod-mset fs*)) = *Mp* (*smult* (*M c*) (*Mp* (*prod-mset fs*))) **by** *simp*
also have . . . = *Mp* (*smult* (*M c*) (*Mp* (*prod-mset* (*image-mset Mp fs*))))
unfolding *Mp-prod-mset* **by** *simp*
also have . . . = *Mp* (*smult* (*M c*) (*prod-mset* (*image-mset Mp fs*))) **unfolding** *Mp-smult* **..**
finally show *?case* **by** *auto*
qed

lemma *unique-factorization-m-imp-factorization*: **assumes** *unique-factorization-m f cfs*
shows *factorization-m f cfs*
proof −
from *assms*[*unfolded unique-factorization-m-def*] **obtain** *dfs* **where**
fact: *factorization-m f dfs* **and** *id*: *Mf cfs* = *Mf dfs* **by** *blast*
from *fact* **have** *factorization-m f* (*Mf dfs*) **by** *simp*
from *this*[*folded id*] **show** *?thesis* **by** *simp*
qed

lemma *unique-factorization-m-alt-def*: *unique-factorization-m f cfs* = (*factorization-m f cfs*
∧ (∀ *dgs*. *factorization-m f dgs* ⟶ *Mf dgs* = *Mf cfs*))
using *unique-factorization-m-imp-factorization*[*of f cfs*]
unfolding *unique-factorization-m-def* **by** *auto*

end

context *poly-mod-2*
begin

lemma *factorization-m-lead-coeff*: **assumes** *factorization-m f* (*c*,*fs*)
shows *lead-coeff* (*Mp f*) = *M c*
proof −
note ∗ = *assms*[*unfolded factorization-m-def split*]
have *monic* (*prod-mset* (*image-mset Mp fs*)) **by** (*rule monic-prod-mset*, *insert* ∗,

auto)
hence *monic* (*Mp* (*prod-mset* (*image-mset Mp fs*))) **by** (*rule monic-Mp*)
from *this*[*unfolded Mp-prod-mset*] **have** *monic*: *monic* (*Mp* (*prod-mset fs*)) **by** *simp*
from ∗ **have** *lead-coeff* (*Mp f*) = *lead-coeff* (*Mp* (*smult c* (*prod-mset fs*))) **by** *simp*
also have *Mp* (*smult c* (*prod-mset fs*)) = *Mp* (*smult* (*M c*) (*Mp* (*prod-mset fs*))) **by** *simp*
finally show *?thesis*
using *monic* ‹*smult c* (*prod-mset fs*) =*m smult* (*M c*) (*Mp* (*prod-mset fs*))›
by (*metis M-M M-def Mp-0 Mp-coeff lead-coeff-smult m1 mult-cancel-left2 poly-mod.degree-m-eq smult-eq-0-iff*)
qed

lemma *factorization-m-smult*: **assumes** *factorization-m f* (*c*,*fs*)
shows *factorization-m* (*smult d f*) (*c* ∗ *d*,*fs*)
proof −
note ∗ = *assms*[*unfolded factorization-m-def split*]
from ∗ **have** *f*: *Mp f* = *Mp* (*smult c* (*prod-mset fs*)) **by** *simp*
have *Mp* (*smult d f*) = *Mp* (*smult d* (*Mp f*)) **by** *simp*
also have ... = *Mp* (*smult* (*c* ∗ *d*) (*prod-mset fs*)) **unfolding** *f* **by** (*simp add*: *ac-simps*)
finally show *?thesis* **using** *assms*
unfolding *factorization-m-def split* **by** *auto*
qed

lemma *factorization-m-prod*: **assumes** *factorization-m f* (*c*,*fs*) *factorization-m g* (*d*,*gs*)
shows *factorization-m* (*f* ∗ *g*) (*c* ∗ *d*, *fs* + *gs*)
proof −
note ∗ = *assms*[*unfolded factorization-m-def split*]
have *Mp* (*f* ∗ *g*) = *Mp* (*Mp f* ∗ *Mp g*) **by** *simp*
also have *Mp f* = *Mp* (*smult c* (*prod-mset fs*)) **using** ∗ **by** *simp*
also have *Mp g* = *Mp* (*smult d* (*prod-mset gs*)) **using** ∗ **by** *simp*
finally have *Mp* (*f* ∗ *g*) = *Mp* (*smult* (*c* ∗ *d*) (*prod-mset* (*fs* + *gs*))) **unfolding** *mult-Mp*
by (*simp add*: *ac-simps*)
with ∗ **show** *?thesis* **unfolding** *factorization-m-def split* **by** *auto*
qed

lemma *Mp-factorization-m*[*simp*]: *factorization-m* (*Mp f*) *cfs* = *factorization-m f cfs*
unfolding *factorization-m-def* **by** *simp*

lemma *Mp-unique-factorization-m*[*simp*]:
unique-factorization-m (*Mp f*) *cfs* = *unique-factorization-m f cfs*
unfolding *unique-factorization-m-alt-def* **by** *simp*

lemma *unique-factorization-m-cong*: *unique-factorization-m f cfs* ⟹ *Mp f* = *Mp*

g
  $\Longrightarrow$ *unique-factorization-m g cfs*
  unfolding *Mp-unique-factorization-m*[*of f*, *symmetric*] **by** *simp*

lemma *unique-factorization-mI*: **assumes** *factorization-m f* (*c*,*fs*)
  and $\bigwedge$ *d gs*. *factorization-m f* (*d*,*gs*) $\Longrightarrow$ *Mf* (*d*,*gs*) = *Mf* (*c*,*fs*)
  shows *unique-factorization-m f* (*c*,*fs*)
  unfolding *unique-factorization-m-alt-def*
    by (*intro conjI*[*OF assms*(*1*)] *allI impI*, *insert assms*(*2*), *auto*)

lemma *unique-factorization-m-smult*: **assumes** *uf*: *unique-factorization-m f* (*c*,*fs*)
  and *d*: *M* (*di* ∗ *d*) = *1*
  shows *unique-factorization-m* (*smult d f*) (*c* ∗ *d*,*fs*)
proof (*rule unique-factorization-mI*[*OF factorization-m-smult*])
  show *factorization-m f* (*c*, *fs*) **using** *uf*[*unfolded unique-factorization-m-alt-def*]
by *auto*
  fix *e gs*
  assume *fact*: *factorization-m* (*smult d f*) (*e*,*gs*)
  from *factorization-m-smult*[*OF this*, *of di*]
  have *factorization-m* (*Mp* (*smult di* (*smult d f*))) (*e* ∗ *di*, *gs*) **by** *simp*
  also have *Mp* (*smult di* (*smult d f*)) = *Mp* (*smult* (*M* (*di* ∗ *d*)) *f*) **by** *simp*
  also have ... = *Mp f* **unfolding** *d* **by** *simp*
  finally have *fact*: *factorization-m f* (*e* ∗ *di*, *gs*) **by** *simp*
  with *uf*[*unfolded unique-factorization-m-alt-def*] **have** *eq*: *Mf* (*e* ∗ *di*, *gs*) = *Mf*
(*c*, *fs*) **by** *blast*
  from *eq*[*unfolded Mf-def*] **have** *M* (*e* ∗ *di*) = *M c* **by** *simp*
  from *arg-cong*[*OF this*, *of* λ *x*. *M* (*x* ∗ *d*)]
  have *M* (*e* ∗ *M* (*di* ∗ *d*)) = *M* (*c* ∗ *d*) **by** (*simp add*: *ac-simps*)
  from *this*[*unfolded d*] **have** *e*: *M e* = *M* (*c* ∗ *d*) **by** *simp*
  with *eq*
  show *Mf* (*e*,*gs*) = *Mf* (*c* ∗ *d*, *fs*) **unfolding** *Mf-def split* **by** *simp*
qed

lemma *unique-factorization-m-smultD*: **assumes** *uf*: *unique-factorization-m* (*smult*
d f) (*c*,*fs*)
  and *d*: *M* (*di* ∗ *d*) = *1*
  shows *unique-factorization-m f* (*c* ∗ *di*,*fs*)
proof −
  from *d* **have** *d'*: *M* (*d* ∗ *di*) = *1* **by** (*simp add*: *ac-simps*)
  show *?thesis*
  proof (*rule unique-factorization-m-cong*[*OF unique-factorization-m-smult*[*OF uf*
d']],
    rule poly-eqI, *unfold Mp-coeff coeff-smult*)
    fix *n*
     have *M* (*di* ∗ (*d* ∗ *coeff f n*)) = *M* (*M* (*di* ∗ *d*) ∗ *coeff f n*) **by** (*auto simp*:
ac-simps)
    from *this*[*unfolded d*] **show** *M* (*di* ∗ (*d* ∗ *coeff f n*)) = *M* (*coeff f n*) **by** *simp*
  qed
qed

lemma *degree-m-eq-lead-coeff*: *degree-m f* = *degree f* ⟹ *lead-coeff* (*Mp f*) = *M* (*lead-coeff f*)
  by (*simp add*: *Mp-coeff*)

lemma *unique-factorization-m-zero*: **assumes** *unique-factorization-m f* (*c*,*fs*)
  shows *M c* ≠ *0*
proof
  assume *c*: *M c* = *0*
  from *unique-factorization-m-imp-factorization*[*OF assms*]
  have *Mp f* = *Mp* (*smult* (*M c*) (*prod-mset fs*)) **unfolding** *factorization-m-def split*
    by *simp*
  from *this*[*unfolded c*] **have** *f*: *Mp f* = *0* **by** *simp*
  have *factorization-m f* (*0*,{#})
    unfolding *factorization-m-def split f* **by** *auto*
  moreover have *Mf* (*0*,{#}) = (*0*,{#}) **unfolding** *Mf-def* **by** *auto*
  ultimately have *fact1*: (*0*, {#}) ∈ *Mf* ‘ *Collect* (*factorization-m f*) **by** *force*
  define *g* :: *int poly* **where** *g* = [:*0*,*1*:]
  have *mpg*: *Mp g* = [:*0*,*1*:] **unfolding** *Mp-def*
    by (*auto simp*: *g-def*)
  {
    fix *g h*
    assume ∗: *degree* (*Mp g*) = *0 degree* (*Mp h*) = *0* [:*0*, *1*:] = *Mp* (*g* ∗ *h*)
    from *arg-cong*[*OF* ∗(*3*), *of degree*] **have** *1* = *degree-m* (*Mp g* ∗ *Mp h*) **by** *simp*
    also have . . . ≤ *degree* (*Mp g* ∗ *Mp h*) **by** (*rule degree-m-le*)
    also have . . . ≤ *degree* (*Mp g*) + *degree* (*Mp h*) **by** (*rule degree-mult-le*)
    also have . . . ≤ *0* **using** ∗ **by** *simp*
    finally have *False* **by** *simp*
  } **note** *irr* = *this*
  have *factorization-m f* (*0*,{# *g* #})
    unfolding *factorization-m-def split* **using** *irr*
    by (*auto simp*: *irreducible*$_d$*-m-def f mpg*)
  moreover have *Mf* (*0*,{# *g* #}) = (*0*,{# *g* #}) **unfolding** *Mf-def* **by** (*auto simp*: *mpg*, *simp add*: *g-def*)
  ultimately have *fact2*: (*0*, {#*g*#}) ∈ *Mf* ‘ *Collect* (*factorization-m f*) **by** *force*
  note [*simp*] = *assms*[*unfolded unique-factorization-m-def*]
  from *fact1*[*simplified*, *folded fact2*[*simplified*]] **show** *False* **by** *auto*
qed

end

context *poly-mod*
begin

lemma *dvdm-smult*: **assumes** *f dvdm g*
  shows *f dvdm smult c g*
proof −

from *assms*[*unfolded dvdm-def*] **obtain** *h* **where** *g*: *g* =*m f* ∗ *h* **by** *auto*
show *?thesis* **unfolding** *dvdm-def*
proof (*intro exI*[*of - smult c h*])
have *Mp* (*smult c g*) = *Mp* (*smult c* (*Mp g*)) **by** *simp*
also have *Mp g* = *Mp* (*f* ∗ *h*) **using** *g* **by** *simp*
finally show *Mp* (*smult c g*) = *Mp* (*f* ∗ *smult c h*) **by** *simp*
qed
qed

lemma *dvdm-factor*: **assumes** *f dvdm g*
shows *f dvdm g* ∗ *h*
proof −
from *assms*[*unfolded dvdm-def*] **obtain** *k* **where** *g*: *g* =*m f* ∗ *k* **by** *auto*
show *?thesis* **unfolding** *dvdm-def*
proof (*intro exI*[*of - h* ∗ *k*])
have *Mp* (*g* ∗ *h*) = *Mp* (*Mp g* ∗ *h*) **by** *simp*
also have *Mp g* = *Mp* (*f* ∗ *k*) **using** *g* **by** *simp*
finally show *Mp* (*g* ∗ *h*) = *Mp* (*f* ∗ (*h* ∗ *k*)) **by** (*simp add*: *ac-simps*)
qed
qed

lemma *square-free-m-smultD*: **assumes** *square-free-m* (*smult c f*)
shows *square-free-m f*
unfolding *square-free-m-def*
proof (*intro conjI allI impI*)
fix *g*
assume *degree-m g* ≠ *0*
with *assms*[*unfolded square-free-m-def*] **have** ¬ *g* ∗ *g dvdm smult c f* **by** *auto*
thus ¬ *g* ∗ *g dvdm f* **using** *dvdm-smult*[*of g* ∗ *g f c*] **by** *blast*
next
from *assms*[*unfolded square-free-m-def*] **have** ¬ *smult c f* =*m 0* **by** *simp*
thus ¬ *f* =*m 0*
by (*metis Mp-smult*(*2*) *smult-0-right*)
qed

lemma *square-free-m-smultI*: **assumes** *sf*: *square-free-m f*
and *inv*: *M* (*ci* ∗ *c*) = *1*
shows *square-free-m* (*smult c f*)
proof −
have *square-free-m* (*smult ci* (*smult c f*))
proof (*rule square-free-m-cong*[*OF sf*], *rule poly-eqI*, *unfold Mp-coeff coeff-smult*)
fix *n*
have *M* (*ci* ∗ (*c* ∗ *coeff f n*)) = *M* (*M* (*ci* ∗ *c*) ∗ *coeff f n*) **by** (*simp add*: *ac-simps*)
from *this*[*unfolded inv*] **show** *M* (*coeff f n*) = *M* (*ci* ∗ (*c* ∗ *coeff f n*)) **by** *simp*
qed
from *square-free-m-smultD*[*OF this*] **show** *?thesis* **.**
qed

lemma *square-free-m-factor*: **assumes** *square-free-m* $(f * g)$
  shows *square-free-m f square-free-m g*
proof −
  {
    fix *f g*
    assume *sf*: *square-free-m* $(f * g)$
    have *square-free-m f*
      unfolding *square-free-m-def*
    proof (*intro conjI allI impI*)
      fix *h*
      assume *degree-m h* $\neq 0$
      with *sf*[*unfolded square-free-m-def*] **have** $\neg$ *h* $*$ *h dvdm f* $*$ *g* **by** *auto*
      thus $\neg$ *h* $*$ *h dvdm f* **using** *dvdm-factor*[*of h* $*$ *h f g*] **by** *blast*
    next
      from *sf*[*unfolded square-free-m-def*] **have** $\neg$ *f* $*$ *g* =*m 0* **by** *simp*
      thus $\neg$ *f* =*m 0*
        by (*metis mult.commute mult-zero-right poly-mod.mult-Mp*(*2*))
    qed
  }
  from *this*[*of f g*] *this*[*of g f*] *assms*
  show *square-free-m f square-free-m g* **by** (*auto simp*: *ac-simps*)
qed

end

context *poly-mod-2*
begin

lemma *Mp-ident-iff*: *Mp f* = *f* $\longleftrightarrow$ ($\forall$ *n*. *coeff f n* $\in$ {*0* ..< *m*})
proof −
  have *m0*: *m* > *0* **using** *m1* **by** *simp*
  show *?thesis* **unfolding** *poly-eq-iff Mp-coeff M-def mod-ident-iff*[*OF m0*] **by** *simp*
qed

lemma *Mp-ident-iff'*: *Mp f* = *f* $\longleftrightarrow$ (*set* (*coeffs f*) $\subseteq$ {*0* ..< *m*})
proof −
  have *0*: *0* $\in$ {*0* ..< *m*} **using** *m1* **by** *auto*
  have *ran*: ($\forall$ *n*. *coeff f n* $\in$ {*0*..<*m*}) $\longleftrightarrow$ *range* (*coeff f*) $\subseteq$ {*0* ..< *m*} **by** *blast*
  show *?thesis* **unfolding** *Mp-ident-iff ran* **using** *range-coeff*[*of f*] *0* **by** *auto*
qed
end

lemma *Mp-Mp-pow-is-Mp*: *n* $\neq$ *0* $\Longrightarrow$ *p* > *1* $\Longrightarrow$ *poly-mod.Mp p* (*poly-mod.Mp* (*p*^*n*) *f*)
  = *poly-mod.Mp p f*
  using *poly-mod-2.Mp-product-modulus poly-mod-2-def* **by**(*subst power-eq-if*, *auto*)

lemma *M-M-pow-is-M*: *n* $\neq$ *0* $\Longrightarrow$ *p* > *1* $\Longrightarrow$ *poly-mod.M p* (*poly-mod.M* (*p*^*n*)

f)
  = *poly-mod.M p f* **using** *Mp-Mp-pow-is-Mp*[*of n p* [:*f*:]]
  by (*metis coeff-pCons-0 poly-mod.Mp-coeff*)

definition *inverse-mod* :: *int* ⇒ *int* ⇒ *int* **where**
  inverse-mod x m = *fst* (*bezout-coefficients x m*)

lemma *inverse-mod*:
  (*inverse-mod x m* ∗ *x*) *mod m* = *1*
  if *coprime x m m* > *1*
proof −
  from *bezout-coefficients* [*of x m inverse-mod x m snd* (*bezout-coefficients x m*)]
  have *inverse-mod x m* ∗ *x* + *snd* (*bezout-coefficients x m*) ∗ *m* = *gcd x m*
    by (*simp add*: *inverse-mod-def*)
  with *that* **have** *inverse-mod x m* ∗ *x* + *snd* (*bezout-coefficients x m*) ∗ *m* = *1*
    by *simp*
  then have (*inverse-mod x m* ∗ *x* + *snd* (*bezout-coefficients x m*) ∗ *m*) *mod m* =
1 mod m
    by *simp*
  with ‹*m* > *1*› **show** *?thesis*
    by *simp*
qed

lemma *inverse-mod-pow*:
  (*inverse-mod x* (*p* ^ *n*) ∗ *x*) *mod* (*p* ^ *n*) = *1*
  if *coprime x p p* > *1 n* ≠ *0*
  using *that* **by** (*auto intro*: *inverse-mod*)

lemma (**in** *poly-mod*) *inverse-mod-coprime*:
  assumes *p*: *prime m*
  and *cop*: *coprime x m* **shows** *M* (*inverse-mod x m* ∗ *x*) = *1*
  unfolding *M-def* **using** *inverse-mod-pow*[*OF cop*, *of 1*] *p*
  by (*auto simp*: *prime-int-iff*)

lemma (**in** *poly-mod*) *inverse-mod-coprime-exp*:
  assumes *m*: *m* = *p*^*n* **and** *p*: *prime p*
  and *n*: *n* ≠ *0* **and** *cop*: *coprime x p*
  shows *M* (*inverse-mod x m* ∗ *x*) = *1*
  unfolding *M-def* **unfolding** *m* **using** *inverse-mod-pow*[*OF cop - n*] *p*
  by (*auto simp*: *prime-int-iff*)

locale *poly-mod-prime* = *poly-mod p* **for** *p* :: *int* +
  assumes *prime*: *prime p*
begin

sublocale *poly-mod-2 p* **using** *prime* **unfolding** *poly-mod-2-def*
  using *prime-gt-1-int* **by** *force*

lemma *square-free-m-prod-imp-coprime-m*: **assumes** *sf*: *square-free-m* (*A* ∗ *B*)

```
  shows coprime-m A B
  unfolding coprime-m-def
proof (intro allI impI)
  fix h
  assume dvd: h dvdm A h dvdm B
  then obtain ha hb where *: Mp A = Mp (h * ha) Mp B = Mp (h * hb)
    unfolding dvdm-def by auto
  have AB: Mp (A * B) = Mp (Mp A * Mp B) by simp
  from this[unfolded *, simplified]
  have eq: Mp (A * B) = Mp (h * h * (ha * hb)) by (simp add: ac-simps)
  hence dvd-hh: (h * h) dvdm (A * B) unfolding dvdm-def by auto
  {
    assume degree-m h ≠ 0
    from sf[unfolded square-free-m-def, THEN conjunct2, rule-format, OF this]
    have ¬ h * h dvdm A * B .
    with dvd-hh have False by simp
  }
  hence degree (Mp h) = 0 by auto
  then obtain c where hc: Mp h = [: c :] by (rule degree-eq-zeroE)
  {
    assume c = 0
    hence Mp h = 0 unfolding hc by auto
    with *(1) have Mp A = 0
      by (metis Mp-0 mult-zero-left poly-mod.mult-Mp(1))
    with sf[unfolded square-free-m-def, THEN conjunct1] have False
      by (simp add: AB)
  }
  hence c0: c ≠ 0 by auto
  with arg-cong[OF hc[symmetric], of λ f. coeff f 0, unfolded Mp-coeff M-def] m1
  have c ≥ 0 c < p by auto
  with c0 have c-props:c > 0 c < p by auto
  with prime have prime p by simp
  with c-props have coprime p c
    by (auto intro: prime-imp-coprime dest: zdvd-not-zless)
  then have coprime c p
    by (simp add: ac-simps)
  from inverse-mod-coprime[OF prime this]
  obtain d where d: M (c * d) = 1 by (auto simp: ac-simps)
  show h dvdm 1 unfolding dvdm-def
  proof (intro exI[of - [:d:]])
    have Mp (h * [: d :]) = Mp (Mp h * [: d :]) by simp
    also have ... = Mp ([: c * d :]) unfolding hc by (auto simp: ac-simps)
    also have ... = [: M (c * d) :] unfolding Mp-def
      by (metis (no-types) M-0 map-poly-pCons Mp-0 Mp-def d zero-neq-one)
    also have ... = 1 unfolding d by simp
    finally show Mp 1 = Mp (h * [:d:]) by simp
  qed
qed
```

lemma *coprime-exp-mod*: *coprime lu p* $\Longrightarrow$ *n* $\neq$ *0* $\Longrightarrow$ *lu mod p* ^ *n* $\neq$ *0*
  using *prime* **by** *fastforce*

end

context *poly-mod*
begin

definition *Dp* :: *int poly* $\Rightarrow$ *int poly* **where**
  Dp f = *map-poly* (λ *a*. *a div m*) *f*

lemma *Dp-Mp-eq*: *f* = *Mp f* + *smult m* (*Dp f*)
  by (*rule poly-eqI*, *auto simp*: *Mp-coeff M-def Dp-def coeff-map-poly*)

lemma *dvd-imp-dvdm*:
  assumes *a dvd b* **shows** *a dvdm b*
  by (*metis assms dvd-def dvdm-def*)

lemma *dvdm-add*:
  assumes *a*: *u dvdm a*
  and *b*: *u dvdm b*
  shows *u dvdm* (*a*+*b*)
proof −
  obtain *a'* **where** *a*: *a* =*m* *u*∗*a'* **using** *a* **unfolding** *dvdm-def* **by** *auto*
  obtain *b'* **where** *b*: *b* =*m* *u*∗*b'* **using** *b* **unfolding** *dvdm-def* **by** *auto*
  have *Mp* (*a* + *b*) = *Mp* (*u*∗*a'*+*u*∗*b'*) **using** *a b*
    by (*metis poly-mod.plus-Mp*(*1*) *poly-mod.plus-Mp*(*2*))
  also have ... = *Mp* (*u* ∗ (*a'*+ *b'*))
    by (*simp add*: *distrib-left*)
  finally show *?thesis* **unfolding** *dvdm-def* **by** *auto*
qed

lemma *monic-dvdm-constant*:
  assumes *uk*: *u dvdm* [:*k*:]
  and *u1*: *monic u* **and** *u2*: *degree u* > *0*
  shows *k mod m* = *0*
proof −
  have *d1*: *degree-m* [:*k*:] = *degree* [:*k*:]
    by (*metis degree-pCons-0 le-zero-eq poly-mod.degree-m-le*)
  obtain *h* **where** *h*: *Mp* [:*k*:] = *Mp* (*u* ∗ *h*)
    using *uk* **unfolding** *dvdm-def* **by** *auto*
  have *d2*: *degree-m* [:*k*:] = *degree-m* (*u*∗*h*) **using** *h* **by** *metis*
  have *d3*: *degree* (*map-poly M* (*u* ∗ *map-poly M h*)) = *degree* (*u* ∗ *map-poly M h*)

    by (*rule degree-map-poly*)
      (*metis coeff-degree-mult leading-coeff-0-iff mult.right-neutral M-M Mp-coeff*
Mp-def u1)
  thus *?thesis* **using** *assms d1 d2 d3*

```
    by (auto, metis M-def map-poly-pCons degree-mult-right-le h leD map-poly-0
        mult-poly-0-right pCons-eq-0-iff M-0 Mp-def mult-Mp(2))
qed

lemma div-mod-imp-dvdm:
  assumes ∃ q r. b = q * a + Polynomial.smult m r
  shows a dvdm b
proof −
  from assms obtain q r where b:b = a * q + smult m r
    by (metis mult.commute)
  have a: Mp (Polynomial.smult m r) = 0 by auto
  show ?thesis
  proof (unfold dvdm-def, rule exI[of - q])
    have Mp (a * q + smult m r) = Mp (a * q + Mp (smult m r))
      using plus-Mp(2)[of a*q smult m r] by auto
    also have ... = Mp (a*q) by auto
    finally show eq-m b (a * q) using b by auto
  qed
qed

lemma lead-coeff-monic-mult:
  fixes p :: 'a :: {comm-semiring-1,semiring-no-zero-divisors} poly
  assumes monic p shows lead-coeff (p * q) = lead-coeff q
  using assms by (simp add: lead-coeff-mult)

lemma degree-m-mult-eq:
  assumes p: monic p and q: lead-coeff q mod m ≠ 0 and m1: m > 1
  shows degree (Mp (p * q)) = degree p + degree q
proof−
  have lead-coeff (p * q) mod m ≠ 0
    using q p by (auto simp: lead-coeff-monic-mult)
  with m1 show ?thesis
    by (auto simp: degree-m-eq intro!: degree-mult-eq)
qed

lemma dvdm-imp-degree-le:
  assumes pq: p dvdm q and p: monic p and q0: Mp q ≠ 0 and m1: m > 1
  shows degree p ≤ degree q
proof−
  from q0
  have q: lead-coeff (Mp q) mod m ≠ 0
    by (metis Mp-Mp Mp-coeff leading-coeff-neq-0 M-def)
  from pq obtain r where Mpq: Mp q = Mp (p * Mp r) by (auto elim: dvdmE)
  with p q have lead-coeff (Mp r) mod m ≠ 0
    by (metis Mp-Mp Mp-coeff leading-coeff-0-iff mult-poly-0-right M-def)
  from degree-m-mult-eq[OF p this m1] Mpq
  have degree p ≤ degree-m q by simp
  thus ?thesis using degree-m-le le-trans by blast
qed
```

```
lemma dvdm-uminus [simp]:
  p dvdm −q ⟷ p dvdm q
  by (metis add.inverse-inverse dvdm-smult smult-1-left smult-minus-left)

lemma Mp-const-poly: Mp [:a:] = [:a mod m:]
  by (simp add: Mp-def M-def Polynomial.map-poly-pCons)

lemma dvdm-imp-div-mod:
  assumes u dvdm g
  shows ∃ q r. g = q∗u + smult m r
proof −
  obtain q where q: Mp g = Mp (u∗q)
    using assms unfolding dvdm-def by fast
  have (u∗q) = Mp (u∗q) + smult m (Dp (u∗q))
    by (simp add: poly-mod.Dp-Mp-eq[of u∗q])
  hence uq: Mp (u∗q) = (u∗q) − smult m (Dp (u∗q))
    by auto
  have g: g = Mp g + smult m (Dp g)
    by (simp add: poly-mod.Dp-Mp-eq[of g])
  also have ... = poly-mod.Mp m (u∗q) + smult m (Dp g) using q by simp
  also have ... = u ∗ q − smult m (Dp (u ∗ q)) + smult m (Dp g)
    unfolding uq by auto
  also have ... = u ∗ q + smult m (−Dp (u∗q)) + smult m (Dp g) by auto
  also have ... = u ∗ q + smult m (−Dp (u∗q) + Dp g)
    unfolding smult-add-right by auto
  also have ... = q ∗ u + smult m (−Dp (u∗q) + Dp g) by auto
  finally show ?thesis by auto
qed

corollary div-mod-iff-dvdm:
  shows a dvdm b = (∃ q r. b = q ∗ a + Polynomial.smult m r)
  using div-mod-imp-dvdm dvdm-imp-div-mod by blast

lemma dvdmE':
  assumes p dvdm q and ⋀r. q =m p ∗ Mp r ⟹ thesis
  shows thesis
  using assms by (auto simp: dvdm-def)

end

context poly-mod-2
begin
lemma factorization-m-mem-dvdm: assumes fact: factorization-m f (c,fs)
  and mem: Mp g ∈# image-mset Mp fs
shows g dvdm f
proof −
```

from *fact* **have** *factorization-m f* (*Mf* (*c*, *fs*)) **by** *auto*
then obtain *l* **where** *f*: *factorization-m f* (*l*, *image-mset Mp fs*) **by** (*auto simp*: *Mf-def*)
from *multi-member-split*[*OF mem*] **obtain** *ls* **where**
fs: *image-mset Mp fs* = {# *Mp g* #} + *ls* **by** *auto*
from *f*[*unfolded fs split factorization-m-def*] **show** *g dvdm f*
unfolding *dvdm-def*
by (*intro exI*[*of - smult l* (*prod-mset ls*)], *auto simp del*: *Mp-smult*
simp add: *Mp-smult*(*2*)[*of - Mp g* ∗ *prod-mset ls*, *symmetric*], *simp*)
qed

lemma *dvdm-degree*: *monic u* ⟹ *u dvdm f* ⟹ *Mp f* ≠ *0* ⟹ *degree u* ≤ *degree f*
using *dvdm-imp-degree-le m1* **by** *blast*

end

lemma (**in** *poly-mod-prime*) *pl-dvdm-imp-p-dvdm*:
assumes *l0*: *l* ≠ *0*
and *pl-dvdm*: *poly-mod.dvdm* (*p*^*l*) *a b*
shows *a dvdm b*
proof −
from *l0* **have** *l-gt-0*: *l* > *0* **by** *auto*
with *m1* **interpret** *pl*: *poly-mod-2 p*^*l* **by** (*unfold-locales*, *auto*)
from *l-gt-0* **have** *p-rw*: *p* ∗ *p* ^ (*l* − *1*) = *p* ^ *l*
by (*cases l*) *simp-all*
obtain *q r* **where** *b*: *b* = *q* ∗ *a* + *smult* (*p*^*l*) *r* **using** *pl.dvdm-imp-div-mod*[*OF pl-dvdm*] **by** *auto*
have *smult* (*p*^*l*) *r* = *smult p* (*smult* (*p* ^ (*l* − *1*)) *r*) **unfolding** *smult-smult p-rw* **..**
hence *b2*: *b* = *q* ∗ *a* + *smult p* (*smult* (*p* ^ (*l* − *1*)) *r*) **using** *b* **by** *auto*
show *?thesis*
by (*rule div-mod-imp-dvdm*, *rule exI*[*of - q*],
rule exI[*of -* (*smult* (*p* ^ (*l* − *1*)) *r*)], *auto simp add*: *b2*)
qed

end

5.2 Polynomials in a Finite Field

We connect polynomials in a prime field with integer polynomials modulo some prime.

theory *Poly-Mod-Finite-Field*
imports
Finite-Field
Polynomial-Interpolation.Ring-Hom-Poly
HOL−Types-To-Sets.Types-To-Sets
More-Missing-Multiset
Poly-Mod

begin

declare *rel-mset-Zero*[*transfer-rule*]

lemma *mset-transfer*[*transfer-rule*]: (*list-all2 rel* ===> *rel-mset rel*) *mset mset*
proof (*intro rel-funI*)
  show *list-all2 rel xs ys* $\Longrightarrow$ *rel-mset rel* (*mset xs*) (*mset ys*) **for** *xs ys*
  proof (*induct xs arbitrary*: *ys*)
    case *Nil*
    then show *?case* **by** *auto*
  next
    case *IH*: (*Cons x xs*)
   then show *?case* **by** (*auto dest*!:*msed-rel-invL simp*: *list-all2-Cons1 intro*!:*rel-mset-Plus*)
  qed
qed

abbreviation *to-int-poly* :: *'a* :: *finite mod-ring poly* $\Rightarrow$ *int poly* **where**
  to-int-poly $\equiv$ *map-poly to-int-mod-ring*

interpretation *to-int-poly-hom*: *map-poly-inj-zero-hom to-int-mod-ring* **..**

lemma *irreducible$_d$-def-0*:
  fixes *f* :: *'a* :: {*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
  shows *irreducible$_d$ f* = (*degree f* $\neq$ *0* $\wedge$
  ($\forall$ *g h*. *degree g* $\neq$ *0* $\longrightarrow$ *degree h* $\neq$ *0* $\longrightarrow$ *f* $\neq$ *g* $*$ *h*))
proof−
  have *degree g* $\neq$ *0* $\Longrightarrow$ *g* $\neq$ *0* **for** *g* :: *'a poly* **by** *auto*
  note *1* = *degree-mult-eq*[*OF this this*, *simplified*]
  then show *?thesis* **by** (*force elim*!: *irreducible$_d$E*)
qed

5.3 Transferring to class-based mod-ring

locale *poly-mod-type* = *poly-mod m*
  for *m* **and** *ty* :: *'a* :: *nontriv itself* +
  assumes *m*: *m* = *CARD*(*'a*)
begin

lemma *m1*: *m* > *1* **using** *nontriv*[**where** *'a* = *'a*] **by** (*auto simp*:*m*)

sublocale *poly-mod-2* **using** *m1* **by** *unfold-locales*

definition *MP-Rel* :: *int poly* $\Rightarrow$ *'a mod-ring poly* $\Rightarrow$ *bool*
  where *MP-Rel f f'* $\equiv$ (*Mp f* = *to-int-poly f'*)

definition *M-Rel* :: *int* $\Rightarrow$ *'a mod-ring* $\Rightarrow$ *bool*
  where *M-Rel x x'* $\equiv$ (*M x* = *to-int-mod-ring x'*)

definition *MF-Rel* ≡ *rel-prod M-Rel* (*rel-mset MP-Rel*)

lemma *to-int-mod-ring-plus*: *to-int-mod-ring* ((*x* :: *'a mod-ring*) + *y*) = *M* (*to-int-mod-ring x* + *to-int-mod-ring y*)
  unfolding *M-def* **using** *m* **by** (*transfer*, *auto*)

lemma *to-int-mod-ring-times*: *to-int-mod-ring* ((*x* :: *'a mod-ring*) ∗ *y*) = *M* (*to-int-mod-ring x* ∗ *to-int-mod-ring y*)
  unfolding *M-def* **using** *m* **by** (*transfer*, *auto*)

lemma *degree-MP-Rel* [*transfer-rule*]: (*MP-Rel* ===> (=)) *degree-m degree*
  unfolding *MP-Rel-def rel-fun-def*
  by (*auto intro*!: *degree-map-poly*)

lemma *eq-M-Rel*[*transfer-rule*]: (*M-Rel* ===> *M-Rel* ===> (=)) (λ *x y*. *M x* = *M y*) (=)
  unfolding *M-Rel-def rel-fun-def* **by** *auto*

interpretation *to-int-mod-ring-hom*: *map-poly-inj-zero-hom to-int-mod-ring*..

lemma *eq-MP-Rel*[*transfer-rule*]: (*MP-Rel* ===> *MP-Rel* ===> (=)) (=*m*) (=)
  unfolding *MP-Rel-def rel-fun-def* **by** *auto*

lemma *eq-Mf-Rel*[*transfer-rule*]: (*MF-Rel* ===> *MF-Rel* ===> (=)) (λ *x y*. *Mf x* = *Mf y*) (=)
proof (*intro rel-funI*, *goal-cases*)
  case (*1 cfs Cfs dgs Dgs*)
  obtain *c fs* **where** *cfs*: *cfs* = (*c*,*fs*) **by** *force*
  obtain *C Fs* **where** *Cfs*: *Cfs* = (*C*,*Fs*) **by** *force*
  obtain *d gs* **where** *dgs*: *dgs* = (*d*,*gs*) **by** *force*
  obtain *D Gs* **where** *Dgs*: *Dgs* = (*D*,*Gs*) **by** *force*
  note *pairs* = *cfs Cfs dgs Dgs*
  from *1*[*unfolded pairs MF-Rel-def rel-prod.simps*]
  have ∗[*transfer-rule*]: *M-Rel c C M-Rel d D rel-mset MP-Rel fs Fs rel-mset MP-Rel gs Gs*
    by *auto*
  have *eq1*: (*M c* = *M d*) = (*C* = *D*) **by** *transfer-prover*
  from ∗(*3*)[*unfolded rel-mset-def*] **obtain** *fs' Fs'* **where** *fs-eq*: *mset fs'* = *fs mset Fs'* = *Fs*
    and *rel-f*: *list-all2 MP-Rel fs' Fs'* **by** *auto*
  from ∗(*4*)[*unfolded rel-mset-def*] **obtain** *gs' Gs'* **where** *gs-eq*: *mset gs'* = *gs mset Gs'* = *Gs*
    and *rel-g*: *list-all2 MP-Rel gs' Gs'* **by** *auto*
  have *eq2*: (*image-mset Mp fs* = *image-mset Mp gs*) = (*Fs* = *Gs*)
    using ∗(*3*−*4*)
  proof (*induct fs arbitrary*: *Fs gs Gs*)
    case (*empty Fs gs Gs*)
    from *empty*(*1*) **have** *Fs*: *Fs* = {#} **unfolding** *rel-mset-def* **by** *auto*

with *empty* **show** *?case* **by** (*cases gs*; *cases Gs*; *auto simp*: *rel-mset-def*)
next
case (*add f fs Fs′ gs′ Gs′*)
note [*transfer-rule*] = *add*(*3*)
from *msed-rel-invL*[*OF add*(*2*)]
obtain *Fs F* **where** *Fs′*: *Fs′* = *Fs* + {#*F*#} **and** *rel*[*transfer-rule*]:
MP-Rel f F rel-mset MP-Rel fs Fs **by** *auto*
note *IH* = *add*(*1*)[*OF rel*(*2*)]
{
from *add*(*3*)[*unfolded rel-mset-def*] **obtain** *gs Gs* **where** *id*: *mset gs* = *gs′*
mset Gs = *Gs′*
and *rel*: *list-all2 MP-Rel gs Gs* **by** *auto*
have *Mp f* ∈# *image-mset Mp gs′* ⟷ *F* ∈# *Gs′*
proof −
have *?thesis* = ((*Mp f* ∈ *Mp* ‘ *set gs*) = (*F* ∈ *set Gs*))
unfolding *id*[*symmetric*] **by** *simp*
also have . . . **using** *rel*
proof (*induct gs Gs rule*: *list-all2-induct*)
case (*Cons g gs G Gs*)
note [*transfer-rule*] = *Cons*(*1*−*2*)
have *id*: (*Mp g* = *Mp f*) = (*F* = *G*) **by** (*transfer*, *auto*)
show *?case* **using** *id Cons*(*3*) **by** *auto*
qed *auto*
finally show *?thesis* **by** *simp*
qed
} **note** *id* = *this*
show *?case*
proof (*cases Mp f* ∈# *image-mset Mp gs′*)
case *False*
have *Mp f* ∈# *image-mset Mp* (*fs* + {#*f*#}) **by** *auto*
with *False* **have** *F*: *image-mset Mp* (*fs* + {#*f*#}) ≠ *image-mset Mp gs′* **by**
metis
with *False*[*unfolded id*] **show** *?thesis* **unfolding** *Fs′* **by** *auto*
next
case *True*
then obtain *g* **where** *fg*: *Mp f* = *Mp g* **and** *g*: *g* ∈# *gs′* **by** *auto*
from *g* **obtain** *gs* **where** *gs′*: *gs′* = *add-mset g gs* **by** (*rule mset-add*)
from *msed-rel-invL*[*OF add*(*3*)[*unfolded gs′*]]
obtain *Gs G* **where** *Gs′*: *Gs′* = *Gs* + {# *G* #} **and** *gG*[*transfer-rule*]:
MP-Rel g G **and**
gsGs: *rel-mset MP-Rel gs Gs* **by** *auto*
have *FG*: *F* = *G* **by** (*transfer*, *simp add*: *fg*)
note *IH* = *IH*[*OF gsGs*]
show *?thesis* **unfolding** *gs′ Fs′ Gs′* **by** (*simp add*: *fg IH FG*)
qed
qed
show (*Mf cfs* = *Mf dgs*) = (*Cfs* = *Dgs*) **unfolding** *pairs Mf-def split*
by (*simp add*: *eq1 eq2*)
qed

lemmas *coeff-map-poly-of-int* = *coeff-map-poly*[*of of-int*, *OF of-int-0*]

lemma *plus-MP-Rel*[*transfer-rule*]: (*MP-Rel* ===> *MP-Rel* ===> *MP-Rel*) (+) (+)
  unfolding *MP-Rel-def*
proof (*intro rel-funI*, *goal-cases*)
  case (*1 x f y g*)
  have *Mp* (*x* + *y*) = *Mp* (*Mp x* + *Mp y*) **by** *simp*
  also have ... = *Mp* (*map-poly to-int-mod-ring f* + *map-poly to-int-mod-ring g*) **unfolding** *1* **..**
  also have ... = *map-poly to-int-mod-ring* (*f* + *g*) **unfolding** *poly-eq-iff Mp-coeff*

      by (*auto simp*: *to-int-mod-ring-plus*)
  finally show *?case* **.**
qed

lemma *times-MP-Rel*[*transfer-rule*]: (*MP-Rel* ===> *MP-Rel* ===> *MP-Rel*) ((∗)) ((∗))
  unfolding *MP-Rel-def*
proof (*intro rel-funI*, *goal-cases*)
  case (*1 x f y g*)
  have *Mp* (*x* ∗ *y*) = *Mp* (*Mp x* ∗ *Mp y*) **by** *simp*
  also have ... = *Mp* (*map-poly to-int-mod-ring f* ∗ *map-poly to-int-mod-ring g*) **unfolding** *1* **..**
  also have ... = *map-poly to-int-mod-ring* (*f* ∗ *g*)
  proof −
    { fix *n* :: *nat*
      define *A* **where** *A* = {.. *n*}
      have *finite A* **unfolding** *A-def* **by** *auto*
      then have *M* (∑ *i*≤*n*. *to-int-mod-ring* (*coeff f i*) ∗ *to-int-mod-ring* (*coeff g* (*n* − *i*))) =
          to-int-mod-ring (∑ *i*≤*n*. *coeff f i* ∗ *coeff g* (*n* − *i*))
        unfolding *A-def*[*symmetric*]
      proof (*induct A*)
        case (*insert a A*)
        have *?case* = *?case* (**is** (*?l* = *?r*) = -) **by** *simp*
        have *?r* = *to-int-mod-ring* (*coeff f a* ∗ *coeff g* (*n* − *a*) + (∑ *i*∈ *A*. *coeff f i* ∗ *coeff g* (*n* − *i*)))
          using *insert*(*1*−*2*) **by** *auto*
        note *r* = *this*[*unfolded to-int-mod-ring-plus to-int-mod-ring-times*]
        from *insert*(*1*−*2*) **have** *?l* = *M* (*to-int-mod-ring* (*coeff f a*) ∗ *to-int-mod-ring* (*coeff g* (*n* − *a*))
          + *M* (∑ *i*∈*A*. *to-int-mod-ring* (*coeff f i*) ∗ *to-int-mod-ring* (*coeff g* (*n* − *i*))))
          by *simp*
        also have *M* (∑ *i*∈*A*. *to-int-mod-ring* (*coeff f i*) ∗ *to-int-mod-ring* (*coeff g* (*n* − *i*))) = *to-int-mod-ring* (∑ *i*∈*A*. *coeff f i* ∗ *coeff g* (*n* − *i*))
          unfolding *insert* **..**

finally
show *?case* **unfolding** *r* **by** *simp*
qed *auto*
}
then show *?thesis* **by** (*auto intro*!:*poly-eqI simp*: *coeff-mult Mp-coeff*)
qed
finally show *?case* **.**
qed

lemma *smult-MP-Rel*[*transfer-rule*]: (*M-Rel* ===> *MP-Rel* ===> *MP-Rel*) *smult smult*
unfolding *MP-Rel-def M-Rel-def*
proof (*intro rel-funI*, *goal-cases*)
case (*1 x x' f f'*)
thus *?case* **unfolding** *poly-eq-iff coeff Mp-coeff coeff-smult M-def*
proof (*intro allI*, *goal-cases*)
case (*1 n*)
have $x * coeff\ f\ n\ mod\ m = (x\ mod\ m) * (coeff\ f\ n\ mod\ m)\ mod\ m$
by (*simp add*: *mod-simps*)
also have ... $= to\text{-}int\text{-}mod\text{-}ring\ x' * (to\text{-}int\text{-}mod\text{-}ring\ (coeff\ f'\ n))\ mod\ m$
using *1* **by** *auto*
also have ... $= to\text{-}int\text{-}mod\text{-}ring\ (x' * coeff\ f'\ n)$
unfolding *to-int-mod-ring-times M-def* **by** *simp*
finally show *?case* **by** *auto*
qed
qed

lemma *one-M-Rel*[*transfer-rule*]: *M-Rel 1 1*
unfolding *M-Rel-def M-def*
unfolding *m* **by** *auto*

lemma *one-MP-Rel*[*transfer-rule*]: *MP-Rel 1 1*
unfolding *MP-Rel-def poly-eq-iff Mp-coeff M-def*
unfolding *m* **by** *auto*

lemma *zero-M-Rel*[*transfer-rule*]: *M-Rel 0 0*
unfolding *M-Rel-def M-def*
unfolding *m* **by** *auto*

lemma *zero-MP-Rel*[*transfer-rule*]: *MP-Rel 0 0*
unfolding *MP-Rel-def poly-eq-iff Mp-coeff M-def*
unfolding *m* **by** *auto*

lemma *listprod-MP-Rel*[*transfer-rule*]: (*list-all2 MP-Rel* ===> *MP-Rel*) *prod-list prod-list*
proof (*intro rel-funI*, *goal-cases*)
case (*1 xs ys*)
thus *?case*

proof (*induct xs ys rule*: *list-all2-induct*)
case (*Cons x xs y ys*)
note [*transfer-rule*] = *this*
show *?case* **by** *simp transfer-prover*
qed (*simp add*: *one-MP-Rel*)
qed

lemma *prod-mset-MP-Rel*[*transfer-rule*]: (*rel-mset MP-Rel* ===> *MP-Rel*) *prod-mset prod-mset*
proof (*intro rel-funI*, *goal-cases*)
case (*1 xs ys*)
have (*MP-Rel* ===> *MP-Rel* ===> *MP-Rel*) ((∗)) ((∗)) *MP-Rel 1 1* **by** *transfer-prover*+
from *1 this* **show** *?case*
proof (*induct xs ys rule*: *rel-mset-induct*)
case (*add R x xs y ys*)
note [*transfer-rule*] = *this*
show *?case* **by** *simp transfer-prover*
qed (*simp add*: *one-MP-Rel*)
qed

lemma *right-unique-MP-Rel*[*transfer-rule*]: *right-unique MP-Rel*
unfolding *right-unique-def MP-Rel-def* **by** *auto*

lemma *M-to-int-mod-ring*: *M* (*to-int-mod-ring* (*x* :: ′*a mod-ring*)) = *to-int-mod-ring x*
unfolding *M-def* **unfolding** *m* **by** (*transfer*, *auto*)

lemma *Mp-to-int-poly*: *Mp* (*to-int-poly* (*f* :: ′*a mod-ring poly*)) = *to-int-poly f*
by (*auto simp*: *poly-eq-iff Mp-coeff M-to-int-mod-ring*)

lemma *right-total-M-Rel*[*transfer-rule*]: *right-total M-Rel*
unfolding *right-total-def M-Rel-def* **using** *M-to-int-mod-ring* **by** *blast*

lemma *left-total-M-Rel*[*transfer-rule*]: *left-total M-Rel*
unfolding *left-total-def M-Rel-def*[*abs-def*]
proof
fix *x*
show ∃ *x*′ :: ′*a mod-ring*. *M x* = *to-int-mod-ring x*′ **unfolding** *M-def* **unfolding** *m*
by (*rule exI*[*of - of-int x*], *transfer*, *simp*)
qed

lemma *bi-total-M-Rel*[*transfer-rule*]: *bi-total M-Rel*
using *right-total-M-Rel left-total-M-Rel* **by** (*metis bi-totalI*)

lemma *right-total-MP-Rel*[*transfer-rule*]: *right-total MP-Rel*
unfolding *right-total-def MP-Rel-def*
proof

fix *f* :: *'a mod-ring poly*
show ∃ *x*. *Mp x* = *to-int-poly f*
by (*intro exI*[*of - to-int-poly f*], *simp add*: *Mp-to-int-poly*)
qed

lemma *to-int-mod-ring-of-int-M*: *to-int-mod-ring* (*of-int x* :: *'a mod-ring*) = *M x*
unfolding *M-def*
unfolding *m* **by** *transfer auto*

lemma *Mp-f-representative*: *Mp f* = *to-int-poly* (*map-poly of-int f* :: *'a mod-ring poly*)
unfolding *Mp-def* **by** (*auto intro*: *poly-eqI simp*: *coeff-map-poly to-int-mod-ring-of-int-M*)

lemma *left-total-MP-Rel*[*transfer-rule*]: *left-total MP-Rel*
unfolding *left-total-def MP-Rel-def*[*abs-def*] **using** *Mp-f-representative* **by** *blast*

lemma *bi-total-MP-Rel*[*transfer-rule*]: *bi-total MP-Rel*
using *right-total-MP-Rel left-total-MP-Rel* **by** (*metis bi-totalI*)

lemma *bi-total-MF-Rel*[*transfer-rule*]: *bi-total MF-Rel*
unfolding *MF-Rel-def*[*abs-def*]
by (*intro prod.bi-total-rel multiset.bi-total-rel bi-total-MP-Rel bi-total-M-Rel*)

lemma *right-total-MF-Rel*[*transfer-rule*]: *right-total MF-Rel*
using *bi-total-MF-Rel* **unfolding** *bi-total-alt-def* **by** *auto*

lemma *left-total-MF-Rel*[*transfer-rule*]: *left-total MF-Rel*
using *bi-total-MF-Rel* **unfolding** *bi-total-alt-def* **by** *auto*

lemma *domain-RT-rel*[*transfer-domain-rule*]: *Domainp MP-Rel* = (λ *f*. *True*)
proof
fix *f* :: *int poly*
show *Domainp MP-Rel f* = *True* **unfolding** *MP-Rel-def*[*abs-def*] *Domainp.simps*
by (*auto simp*: *Mp-f-representative*)
qed

lemma *mem-MP-Rel*[*transfer-rule*]: (*MP-Rel* ===> *rel-set MP-Rel* ===> (=)) (λ *x Y*. ∃ *y* ∈ *Y*. *eq-m x y*) (∈)
proof (*intro rel-funI iffI*)
fix *x y X Y* **assume** *xy*: *MP-Rel x y* **and** *XY*: *rel-set MP-Rel X Y*
{ assume ∃ *x'* ∈ *X*. *x* =*m x'*
then obtain *x'* **where** *x'X*: *x'* ∈ *X* **and** *xx'*: *x* =*m x'* **by** *auto*
with *xy* **have** *x'y*: *MP-Rel x' y* **by** (*auto simp*: *MP-Rel-def*)
from *rel-setD1*[*OF XY x'X*] **obtain** *y'* **where** *MP-Rel x' y'* **and** *y'* ∈ *Y* **by** *auto*
with *x'y*
show *y* ∈ *Y* **by** (*auto simp*: *MP-Rel-def*)
}
assume *y* ∈ *Y*

from *rel-setD2*[*OF XY this*] **obtain** x' **where** $x'X$: $x' \in X$ **and** $x'y$: *MP-Rel* x' y **by** *auto*
from *xy* $x'y$ **have** $x =m\ x'$ **by** (*auto simp*: *MP-Rel-def*)
with $x'X$ **show** $\exists x' \in X.\ x =m\ x'$ **by** *auto*
qed

lemma *conversep-MP-Rel-OO-MP-Rel* [*simp*]: *MP-Rel*$^{-1-1}$ *OO MP-Rel* = (=)
using *Mp-to-int-poly* **by** (*intro ext*, *auto simp*: *OO-def MP-Rel-def*)

lemma *MP-Rel-OO-conversep-MP-Rel* [*simp*]: *MP-Rel OO MP-Rel*$^{-1-1}$ = *eq-m*
by (*intro ext*, *auto simp*: *OO-def MP-Rel-def Mp-f-representative*)

lemma *conversep-MP-Rel-OO-eq-m* [*simp*]: *MP-Rel*$^{-1-1}$ *OO eq-m* = *MP-Rel*$^{-1-1}$
by (*intro ext*, *auto simp*: *OO-def MP-Rel-def*)

lemma *eq-m-OO-MP-Rel* [*simp*]: *eq-m OO MP-Rel* = *MP-Rel*
by (*intro ext*, *auto simp*: *OO-def MP-Rel-def*)

lemma *eq-mset-MP-Rel* [*transfer-rule*]: (*rel-mset MP-Rel* ===> *rel-mset MP-Rel* ===> (=)) (*rel-mset eq-m*) (=)
proof (*intro rel-funI iffI*)
fix *A B X Y*
assume *AX*: *rel-mset MP-Rel A X* **and** *BY*: *rel-mset MP-Rel B Y*
{
assume *AB*: *rel-mset eq-m A B*
from *AX* **have** *rel-mset MP-Rel*$^{-1-1}$ *X A* **by** (*simp add*: *multiset.rel-flip*)
note *rel-mset-OO*[*OF this AB*]
note *rel-mset-OO*[*OF this BY*]
then show *X* = *Y* **by** (*simp add*: *multiset.rel-eq*)
}
assume *X* = *Y*
with *BY* **have** *rel-mset MP-Rel*$^{-1-1}$ *X B* **by** (*simp add*: *multiset.rel-flip*)
from *rel-mset-OO*[*OF AX this*]
show *rel-mset eq-m A B* **by** *simp*
qed

lemma *dvd-MP-Rel*[*transfer-rule*]: (*MP-Rel* ===> *MP-Rel* ===> (=)) (*dvdm*) (*dvd*)
unfolding *dvdm-def*[*abs-def*] *dvd-def*[*abs-def*]
by *transfer-prover*

lemma *irreducible-MP-Rel* [*transfer-rule*]: (*MP-Rel* ===> (=)) *irreducible-m irreducible*
unfolding *irreducible-m-def irreducible-def*
by *transfer-prover*

lemma *irreducible$_d$-MP-Rel* [*transfer-rule*]: (*MP-Rel* ===> (=)) *irreducible$_d$-m irreducible$_d$*
unfolding *irreducible$_d$-m-def*[*abs-def*] *irreducible$_d$-def*[*abs-def*]

by *transfer-prover*

lemma *UNIV-M-Rel*[*transfer-rule*]: *rel-set M-Rel* {*0*..<*m*} *UNIV*
unfolding *rel-set-def M-Rel-def*[*abs-def*] *M-def*
by (*auto simp*: *M-def m*, *goal-cases*, *metis to-int-mod-ring-of-int-mod-ring*, (*transfer*, *auto*)+)

lemma *coeff-MP-Rel* [*transfer-rule*]: (*MP-Rel* ===> (=) ===> *M-Rel*) *coeff coeff*
unfolding *rel-fun-def M-Rel-def MP-Rel-def Mp-coeff*[*symmetric*] **by** *auto*

lemma *M-1-1*: *M 1* = *1* **unfolding** *M-def* **unfolding** *m* **by** *simp*

lemma *square-free-MP-Rel* [*transfer-rule*]: (*MP-Rel* ===> (=)) *square-free-m square-free*
unfolding *square-free-m-def*[*abs-def*] *square-free-def*[*abs-def*]
by (*transfer-prover-start*, *transfer-step*+, *auto*)

lemma *mset-factors-m-MP-Rel* [*transfer-rule*]: (*rel-mset MP-Rel* ===> *MP-Rel* ===> (=)) *mset-factors-m mset-factors*
unfolding *mset-factors-def mset-factors-m-def*
by (*transfer-prover-start*, *transfer-step*+, *auto dest*:*eq-m-irreducible-m*)

lemma *coprime-MP-Rel* [*transfer-rule*]: (*MP-Rel* ===> *MP-Rel* ===> (=)) *coprime-m coprime*
unfolding *coprime-m-def*[*abs-def*] *coprime-def'* [*abs-def*]
by (*transfer-prover-start*, *transfer-step*+, *auto*)

lemma *prime-elem-MP-Rel* [*transfer-rule*]: (*MP-Rel* ===> (=)) *prime-elem-m prime-elem*
unfolding *prime-elem-m-def prime-elem-def* **by** *transfer-prover*

end

context *poly-mod-2* **begin**

lemma *non-empty*: {*0*..<*m*} $\neq$ {} **using** *m1* **by** *auto*

lemma *type-to-set*:
assumes *type-def*: $\exists$(*Rep* :: *'b* $\Rightarrow$ *int*) *Abs*. *type-definition Rep Abs* {*0* ..< *m* :: *int*}
shows *class.nontriv* (*TYPE*(*'b*)) (**is** *?a*) **and** *m* = *int CARD*(*'b*) (**is** *?b*)
proof −
from *type-def* **obtain** *rep* :: *'b* $\Rightarrow$ *int* **and** *abs* :: *int* $\Rightarrow$ *'b* **where** *t*: *type-definition rep abs* {*0* ..< *m*} **by** *auto*
have *card* (*UNIV* :: *'b set*) = *card* {*0* ..< *m*} **using** *t* **by** (*rule type-definition.card*)
also have ... = *m* **using** *m1* **by** *auto*
finally show *?b* ..
then show *?a* **unfolding** *class.nontriv-def* **using** *m1* **by** *auto*
qed

end

locale *poly-mod-prime-type* = *poly-mod-type m ty* **for** *m* :: *int* **and**
ty :: *'a* :: *prime-card itself*
begin

lemma *factorization-MP-Rel* [*transfer-rule*]:
(*MP-Rel* ===> *MF-Rel* ===> (=)) *factorization-m* (*factorization Irr-Mon*)
unfolding *rel-fun-def*
proof (*intro allI impI*, *goal-cases*)
case (*1 f F cfs Cfs*)
note [*transfer-rule*] = *1*(*1*)
obtain *c fs* **where** *cfs*: *cfs* = (*c*,*fs*) **by** *force*
obtain *C Fs* **where** *Cfs*: *Cfs* = (*C*,*Fs*) **by** *force*
from *1*(*2*)[*unfolded rel-prod.simps cfs Cfs MF-Rel-def*]
have *tr*[*transfer-rule*]: *M-Rel c C rel-mset MP-Rel fs Fs* **by** *auto*
have *eq*: (*f* =*m smult c* (*prod-mset fs*) = (*F* = *smult C* (*prod-mset Fs*)))
by *transfer-prover*
have *set-mset Fs* ⊆ *Irr-Mon* = (∀ *x* ∈# *Fs*. $irreducible_d$ *x* ∧ *monic x*) **unfolding**
Irr-Mon-def **by** *auto*
also have ... = (∀ *f*∈#*fs*. $irreducible_d$-*m f* ∧ *monic* (*Mp f*))
proof (*rule sym*, *transfer-prover-start*, *transfer-step*+)
{
fix *f*
assume *f* ∈# *fs*
have *monic* (*Mp f*) ⟷ *M* (*coeff f* (*degree-m f*)) = *M 1*
unfolding *Mp-coeff*[*symmetric*] **by** *simp*
}
thus (∀ *f*∈#*fs*. $irreducible_d$-*m f* ∧ *monic* (*Mp f*)) =
(∀ *x*∈#*fs*. $irreducible_d$-*m x* ∧ *M* (*coeff x* (*degree-m x*)) = *M 1*) **by** *auto*
qed
finally
show *factorization-m f cfs* = *factorization Irr-Mon F Cfs* **unfolding** *cfs Cfs*
factorization-m-def factorization-def split eq **by** *simp*
qed

lemma *unique-factorization-MP-Rel* [*transfer-rule*]: (*MP-Rel* ===> *MF-Rel* ===>
(=))
unique-factorization-m (*unique-factorization Irr-Mon*)
unfolding *rel-fun-def*
proof (*intro allI impI*, *goal-cases*)
case (*1 f F cfs Cfs*)
note [*transfer-rule*] = *1*(*1*,*2*)
let *?F* = *factorization Irr-Mon F*
let *?f* = *factorization-m f*
let *?R* = *Collect ?F*
let *?L* = *Mf* ‘ *Collect ?f*
note *X-to-x* = *right-total-MF-Rel*[*unfolded right-total-def*, *rule-format*]

{
fix *X*
assume $X \in ?R$
hence *F*: *?F X* by *simp*
from *X-to-x*[*of X*] obtain *x* where *rel*[*transfer-rule*]: *MF-Rel x X* by *blast*
from *F*[*untransferred*] have $Mf\ x \in ?L$ by *blast*
with *rel* have $\exists\ x.\ Mf\ x \in ?L \wedge MF\text{-}Rel\ x\ X$ by *blast*
} note *R-to-L* = *this*
show *unique-factorization-m f cfs* = *unique-factorization Irr-Mon F Cfs* unfolding
unique-factorization-m-def unique-factorization-def
proof −
have *fF*: *?F Cfs* = *?f cfs* by *transfer simp*
have $(?L = \{Mf\ cfs\}) = (?L \subseteq \{Mf\ cfs\} \wedge Mf\ cfs \in ?L)$ by *blast*
also have $?L \subseteq \{Mf\ cfs\} = (\forall\ dfs.\ ?f\ dfs \longrightarrow Mf\ dfs = Mf\ cfs)$ by *blast*
also have $\ldots = (\forall\ y.\ ?F\ y \longrightarrow y = Cfs)$ (is *?left* = *?right*)
proof (*rule*; *intro allI impI*)
fix *Dfs*
assume ∗: *?left* and *F*: *?F Dfs*
from *X-to-x*[*of Dfs*] obtain *dfs* where [*transfer-rule*]: *MF-Rel dfs Dfs* by *auto*
from *F*[*untransferred*] have *f*: *?f dfs* .
from ∗[*rule-format*, *OF f*] have *eq*: *Mf dfs* = *Mf cfs* by *simp*
have (*Mf dfs* = *Mf cfs*) = (*Dfs* = *Cfs*) by (*transfer-prover-start*, *transfer-step*+, *simp*)
thus *Dfs* = *Cfs* using *eq* by *simp*
next
fix *dfs*
assume ∗: *?right* and *f*: *?f dfs*
from *left-total-MF-Rel* obtain *Dfs* where
rel[*transfer-rule*]: *MF-Rel dfs Dfs* unfolding *left-total-def* by *blast*
have *?F Dfs* by (*transfer*, *rule f*)
from ∗[*rule-format*, *OF this*] have *eq*: *Dfs* = *Cfs* .
have (*Mf dfs* = *Mf cfs*) = (*Dfs* = *Cfs*) by (*transfer-prover-start*, *transfer-step*+, *simp*)
thus *Mf dfs* = *Mf cfs* using *eq* by *simp*
qed
also have $Mf\ cfs \in ?L = (\exists\ dfs.\ ?f\ dfs \wedge Mf\ cfs = Mf\ dfs)$ by *auto*
also have $\ldots = ?F\ Cfs$ unfolding *fF*
proof
assume $\exists\ dfs.\ ?f\ dfs \wedge Mf\ cfs = Mf\ dfs$
then obtain *dfs* where *f*: *?f dfs* and *id*: *Mf dfs* = *Mf cfs* by *auto*
from *f* have *?f* (*Mf dfs*) by *simp*
from *this*[*unfolded id*] show *?f cfs* by *simp*
qed *blast*
finally show $(?L = \{Mf\ cfs\}) = (?R = \{Cfs\})$ by *auto*
qed
qed

end

context begin
private lemma *1*: *poly-mod-type TYPE*(*'a* :: *nontriv*) *m* = (*m* = *int CARD*(*'a*))
and *2*: *class.nontriv TYPE*(*'a*) = (*CARD*(*'a*) ≥ *2*)
unfolding *poly-mod-type-def class.prime-card-def class.nontriv-def poly-mod-prime-type-def*
by *auto*

private lemma *3*: *poly-mod-prime-type TYPE*(*'b*) *m* = (*m* = *int CARD*(*'b*))
and *4*: *class.prime-card TYPE*(*'b* :: *prime-card*) = *prime CARD*(*'b* :: *prime-card*)

unfolding *poly-mod-type-def class.prime-card-def class.nontriv-def poly-mod-prime-type-def*
by *auto*

lemmas *poly-mod-type-simps* = *1 2 3 4*
end

lemma *remove-duplicate-premise*: (*PROP P* ⟹ *PROP P* ⟹ *PROP Q*) ≡ (*PROP P* ⟹ *PROP Q*) (**is** *?l* ≡ *?r*)
proof (*intro Pure.equal-intr-rule*)
assume *p*: *PROP P* **and** *ppq*: *PROP ?l*
from *ppq*[*OF p p*] **show** *PROP Q*.
next
assume *p*: *PROP P* **and** *pq*: *PROP ?r*
from *pq*[*OF p*] **show** *PROP Q*.
qed

context *poly-mod-prime* **begin**

lemma *type-to-set*:
assumes *type-def*: ∃(*Rep* :: *'b* ⇒ *int*) *Abs*. *type-definition Rep Abs* {*0* ..< *p* :: *int*}
shows *class.prime-card* (*TYPE*(*'b*)) (**is** *?a*) **and** *p* = *int CARD*(*'b*) (**is** *?b*)
proof −
from *prime* **have** *p2*: *p* ≥ *2* **by** (*rule prime-ge-2-int*)
from *type-def* **obtain** *rep* :: *'b* ⇒ *int* **and** *abs* :: *int* ⇒ *'b* **where** *t*: *type-definition rep abs* {*0* ..< *p*} **by** *auto*
have *card* (*UNIV* :: *'b set*) = *card* {*0* ..< *p*} **using** *t* **by** (*rule type-definition.card*)
also have ... = *p* **using** *p2* **by** *auto*
finally show *?b* ..
then show *?a* **unfolding** *class.prime-card-def* **using** *prime p2* **by** *auto*
qed
end

lemmas (**in** *poly-mod-type*) *prime-elem-m-dvdm-multD* = *prime-elem-dvd-multD*
[**where** *'a* = *'a mod-ring poly*,*untransferred*]

lemmas (**in** *poly-mod-2*) *prime-elem-m-dvdm-multD* = *poly-mod-type.prime-elem-m-dvdm-multD*
[*unfolded poly-mod-type-simps*, *internalize-sort* $'a$ :: *nontriv*, *OF type-to-set*, *unfolded remove-duplicate-premise*, *cancel-type-definition*, *OF non-empty*]

lemmas(**in** *poly-mod-prime-type*) *degree-m-mult-eq* = *degree-mult-eq*
[**where** $'a$ = $'a$ *mod-ring*, *untransferred*]
lemmas(**in** *poly-mod-prime*) *degree-m-mult-eq* = *poly-mod-prime-type.degree-m-mult-eq*
[*unfolded poly-mod-type-simps*, *internalize-sort* $'a$ :: *prime-card*, *OF type-to-set*, *unfolded remove-duplicate-premise*, *cancel-type-definition*, *OF non-empty*]

lemma(**in** *poly-mod-prime*) *irreducible*$_d$*-lifting*:
assumes *n*: $n \neq 0$
and *deg*: *poly-mod.degree-m* (p^n) *f* = *degree-m f*
and *irr*: *irreducible*$_d$*-m f*
shows *poly-mod.irreducible*$_d$*-m* (p^n) *f*
proof −
interpret *q*: *poly-mod-2 p*^*n* **unfolding** *poly-mod-2-def* **using** *n m1* **by** *auto*
show *q.irreducible*$_d$*-m f*
proof (*rule q.irreducible*$_d$*-mI*)
from *deg irr* **show** *q.degree-m f* > *0* **by** (*auto elim*: *irreducible*$_d$*-mE*)
then have *pdeg-f*: *degree-m f* ≠ *0* **by** (*simp add*: *deg*)
note *pMp-Mp* = *Mp-Mp-pow-is-Mp*[*OF n m1*]
fix *g h*
assume *deg-g*: *degree g* < *q.degree-m f* **and** *deg-h*: *degree h* < *q.degree-m f*
and *eq*: *q.eq-m f* (*g* ∗ *h*)
from *eq* **have** *p-f*: *f* =*m* (*g* ∗ *h*) **using** *pMp-Mp* **by** *metis*
have ¬*g* =*m 0* **and** ¬*h* =*m 0*
apply (*metis degree-0 mult-zero-left Mp-0 p-f pdeg-f poly-mod.mult-Mp*(*1*))
by (*metis degree-0 mult-eq-0-iff Mp-0 mult-Mp*(*2*) *p-f pdeg-f*)
note [*simp*] = *degree-m-mult-eq*[*OF this*]
from *degree-m-le*[*of g*] *deg-g*
have *2*: *degree-m g* < *degree-m f* **by** (*fold deg*, *auto*)
from *degree-m-le*[*of h*] *deg-h*
have *3*: *degree-m h* < *degree-m f* **by** (*fold deg*, *auto*)
from *irreducible*$_d$*-mD*(*2*)[*OF irr 2 3*] *p-f*
show *False* **by** *auto*
qed
qed

lemmas (**in** *poly-mod-prime-type*) *mset-factors-exist* =
mset-factors-exist[**where** $'a$ = $'a$ *mod-ring poly*,*untransferred*]
lemmas (**in** *poly-mod-prime*) *mset-factors-exist* = *poly-mod-prime-type.mset-factors-exist*
[*unfolded poly-mod-type-simps*, *internalize-sort* $'a$ :: *prime-card*, *OF type-to-set*, *unfolded remove-duplicate-premise*, *cancel-type-definition*, *OF non-empty*]

lemmas (**in** *poly-mod-prime-type*) *mset-factors-unique* =
mset-factors-unique[**where** $'a$ = $'a$ *mod-ring poly*,*untransferred*]
lemmas (**in** *poly-mod-prime*) *mset-factors-unique* = *poly-mod-prime-type.mset-factors-unique*

[*unfolded poly-mod-type-simps*, *internalize-sort 'a :: prime-card*, *OF type-to-set*, *unfolded remove-duplicate-premise*, *cancel-type-definition*, *OF non-empty*]

lemmas (**in** *poly-mod-prime-type*) *prime-elem-iff-irreducible* =
prime-elem-iff-irreducible[**where** *'a = 'a mod-ring poly*,*untransferred*]
lemmas (**in** *poly-mod-prime*) *prime-elem-iff-irreducible*[*simp*] = *poly-mod-prime-type.prime-elem-iff-irreducible*
[*unfolded poly-mod-type-simps*, *internalize-sort 'a :: prime-card*, *OF type-to-set*, *unfolded remove-duplicate-premise*, *cancel-type-definition*, *OF non-empty*]

lemmas (**in** *poly-mod-prime-type*) *irreducible-connect* =
irreducible-connect-field[**where** *'a = 'a mod-ring*, *untransferred*]
lemmas (**in** *poly-mod-prime*) *irreducible-connect*[*simp*] = *poly-mod-prime-type.irreducible-connect*
[*unfolded poly-mod-type-simps*, *internalize-sort 'a :: prime-card*, *OF type-to-set*, *unfolded remove-duplicate-premise*, *cancel-type-definition*, *OF non-empty*]

lemmas (**in** *poly-mod-prime-type*) *irreducible-degree* =
irreducible-degree-field[**where** *'a = 'a mod-ring*, *untransferred*]
lemmas (**in** *poly-mod-prime*) *irreducible-degree* = *poly-mod-prime-type.irreducible-degree*
[*unfolded poly-mod-type-simps*, *internalize-sort 'a :: prime-card*, *OF type-to-set*, *unfolded remove-duplicate-premise*, *cancel-type-definition*, *OF non-empty*]

end

5.4 Karatsuba's Multiplication Algorithm for Polynomials

theory *Karatsuba-Multiplication*
imports
Polynomial-Interpolation.Missing-Polynomial
begin

lemma *karatsuba-main-step*: **fixes** *f :: 'a :: comm-ring-1 poly*
assumes *f*: *f = monom-mult n f1 + f0* **and** *g*: *g = monom-mult n g1 + g0*
shows
monom-mult (n + n) (f1 ∗ g1) + (monom-mult n (f1 ∗ g1 − (f1 − f0) ∗ (g1 − g0) + f0 ∗ g0) + f0 ∗ g0) = f ∗ g
unfolding *assms*
by (*auto simp*: *field-simps mult-monom monom-mult-def*)

lemma *karatsuba-single-sided*: **fixes** *f :: 'a :: comm-ring-1 poly*
assumes *f = monom-mult n f1 + f0*
shows *monom-mult n (f1 ∗ g) + f0 ∗ g = f ∗ g*
unfolding *assms* **by** (*auto simp*: *field-simps mult-monom monom-mult-def*)

definition *split-at :: nat ⇒ 'a list ⇒ 'a list × 'a list* **where**
[*code del*]: *split-at n xs = (take n xs, drop n xs)*

lemma *split-at-code*[*code*]:

split-at n [] = ([],[])
split-at n (x # xs) = (if n = 0 then ([], x # xs) else case split-at (n−1) xs of (bef,aft)
⇒ (x # bef, aft))
unfolding *split-at-def* **by** (*force*, *cases n*, *auto*)

fun *coeffs-minus* :: *'a* :: *ab-group-add list ⇒ 'a list ⇒ 'a list* **where**
coeffs-minus (x # xs) (y # ys) = ((x − y) # coeffs-minus xs ys)
| *coeffs-minus xs [] = xs*
| *coeffs-minus [] ys = map uminus ys*

The following constant determines at which size we will switch to the standard multiplication algorithm.

definition *karatsuba-lower-bound* **where** [*termination-simp*]: *karatsuba-lower-bound = (7 :: nat)*

fun *karatsuba-main* :: *'a* :: *comm-ring-1 list ⇒ nat ⇒ 'a list ⇒ nat ⇒ 'a poly* **where**
karatsuba-main f n g m = (if n ≤ karatsuba-lower-bound ∨ m ≤ karatsuba-lower-bound then
let ff = poly-of-list f in foldr (λa p. smult a ff + pCons 0 p) g 0
else let n2 = n div 2 in
if m > n2 then (case split-at n2 f of
(f0,f1) ⇒ case split-at n2 g of
(g0,g1) ⇒ let
p1 = karatsuba-main f1 (n − n2) g1 (m − n2);
p2 = karatsuba-main (coeffs-minus f1 f0) n2 (coeffs-minus g1 g0) n2;
p3 = karatsuba-main f0 n2 g0 n2
in monom-mult (n2 + n2) p1 + (monom-mult n2 (p1 − p2 + p3) + p3))
else case split-at n2 f of
(f0,f1) ⇒ let
p1 = karatsuba-main f1 (n − n2) g m;
p2 = karatsuba-main f0 n2 g m
in monom-mult n2 p1 + p2)

declare *karatsuba-main.simps*[*simp del*]

lemma *poly-of-list-split-at*: **assumes** *split-at n f = (f0,f1)*
shows *poly-of-list f = monom-mult n (poly-of-list f1) + poly-of-list f0*
proof −
from *assms* **have** *id*: *f1 = drop n f f0 = take n f* **unfolding** *split-at-def* **by** *auto*
show *?thesis* **unfolding** *id*
proof (*rule poly-eqI*)
fix *i*
show *coeff (poly-of-list f) i =*
coeff (monom-mult n (poly-of-list (drop n f)) + poly-of-list (take n f)) i
unfolding *monom-mult-def coeff-monom-mult coeff-add poly-of-list-def coeff-Poly*
by (*cases n ≤ i*; *cases i ≥ length f*, *auto simp*: *nth-default-nth nth-default-beyond*)

  qed
qed

lemma *coeffs-minus*: *poly-of-list* (*coeffs-minus f1 f0*) = *poly-of-list f1* − *poly-of-list f0*
proof (*rule poly-eqI*, *unfold poly-of-list-def coeff-diff coeff-Poly*)
  fix *i*
  show *nth-default 0* (*coeffs-minus f1 f0*) *i* = *nth-default 0 f1 i* − *nth-default 0 f0 i*
  proof (*induct f1 f0 arbitrary*: *i rule*: *coeffs-minus.induct*)
    case (*1 x xs y ys*)
    thus *?case* **by** (*cases i*, *auto*)
  next
    case (*3 x xs*)
    thus *?case* **unfolding** *coeffs-minus.simps*
      by (*subst nth-default-map-eq*[*of uminus 0 0*], *auto*)
  qed *auto*
qed

lemma *karatsuba-main*: *karatsuba-main f n g m* = *poly-of-list f* ∗ *poly-of-list g*
proof (*induct n arbitrary*: *f g m rule*: *less-induct*)
  case (*less n f g m*)
  note *simp*[*simp*] = *karatsuba-main.simps*[*of f n g m*]
  show *?case* (**is** *?lhs* = *?rhs*)
  proof (*cases* (*n* ≤ *karatsuba-lower-bound* ∨ *m* ≤ *karatsuba-lower-bound*) = *False*)
    case *False*
    hence *lhs*: *?lhs* = *foldr* (λ*a p*. *smult a* (*poly-of-list f*) + *pCons 0 p*) *g 0* **by** *simp*
    have *rhs*: *?rhs* = *poly-of-list g* ∗ *poly-of-list f* **by** *simp*
    also have ... = *foldr* (λ*a p*. *smult a* (*poly-of-list f*) + *pCons 0 p*) (*strip-while* ((=) *0*) *g*) *0*
      unfolding *times-poly-def fold-coeffs-def poly-of-list-impl* **..**
    also have ... = *?lhs* **unfolding** *lhs*
    proof (*induct g*)
      case (*Cons x xs*)
      have ∀ *x*∈*set xs*. *x* = *0* ⟹ *foldr* (λ*a p*. *smult a* (*Poly f*) + *pCons 0 p*) *xs 0* = *0*
        by (*induct xs*, *auto*)
      thus *?case* **using** *Cons* **by** (*auto simp*: *cCons-def Cons*)
    qed *auto*
    finally show *?thesis* **by** *simp*
  next
    case *True*
    let *?n2* = *n div 2*
    have *?n2* < *n n* − *?n2* < *n* **using** *True* **unfolding** *karatsuba-lower-bound-def* **by** *auto*
    note *IH* = *less*[*OF this*(*1*)] *less*[*OF this*(*2*)]
    obtain *f1 f0* **where** *f*: *split-at ?n2 f* = (*f0*,*f1*) **by** *force*
    obtain *g1 g0* **where** *g*: *split-at ?n2 g* = (*g0*,*g1*) **by** *force*

note *fsplit* = *poly-of-list-split-at*[*OF f*]
note *gsplit* = *poly-of-list-split-at*[*OF g*]
show *?lhs* = *?rhs* **unfolding** *simp Let-def f g split IH True if-False coeffs-minus karatsuba-single-sided*[*OF fsplit*] *karatsuba-main-step*[*OF fsplit gsplit*] **by** *auto*
qed
qed

definition *karatsuba-mult-poly* :: *'a* :: *comm-ring-1 poly* ⇒ *'a poly* ⇒ *'a poly* **where**
karatsuba-mult-poly f g = (*let ff* = *coeffs f*; *gg* = *coeffs g*; *n* = *length ff*; *m* = *length gg*
in (*if n* ≤ *karatsuba-lower-bound* ∨ *m* ≤ *karatsuba-lower-bound then if n* ≤ *m*
then foldr (*λa p. smult a g* + *pCons 0 p*) *ff 0*
else foldr (*λa p. smult a f* + *pCons 0 p*) *gg 0*
else if n ≤ *m*
then karatsuba-main gg m ff n
else karatsuba-main ff n gg m))

lemma *karatsuba-mult-poly*: *karatsuba-mult-poly f g* = *f* ∗ *g*
proof −
note *d* = *karatsuba-mult-poly-def Let-def*
let *?len* = *length* (*coeffs f*) ≤ *length* (*coeffs g*)
show *?thesis* (**is** *?lhs* = *?rhs*)
proof (*cases length* (*coeffs f*) ≤ *karatsuba-lower-bound* ∨ *length* (*coeffs g*) ≤ *karatsuba-lower-bound*)
case *True* **note** *outer* = *this*
show *?thesis*
proof (*cases ?len*)
case *True*
with *outer* **have** *?lhs* = *foldr* (*λa p. smult a g* + *pCons 0 p*) (*coeffs f*) *0* **unfolding** *d* **by** *auto*
also have ... = *?rhs* **unfolding** *times-poly-def fold-coeffs-def* **by** *auto*
finally show *?thesis* **.**
next
case *False*
with *outer* **have** *?lhs* = *foldr* (*λa p. smult a f* + *pCons 0 p*) (*coeffs g*) *0* **unfolding** *d* **by** *auto*
also have ... = *g* ∗ *f* **unfolding** *times-poly-def fold-coeffs-def* **by** *auto*
also have ... = *?rhs* **by** *simp*
finally show *?thesis* **.**
qed
next
case *False* **note** *outer* = *this*
show *?thesis*
proof (*cases ?len*)
case *True*
with *outer* **have** *?lhs* = *karatsuba-main* (*coeffs g*) (*length* (*coeffs g*)) (*coeffs f*) (*length* (*coeffs f*))
unfolding *d* **by** *auto*

also have ... $=$ *g* $*$ *f* **unfolding** *karatsuba-main* **by** *auto*
also have ... $=$ *?rhs* **by** *auto*
finally show *?thesis* **.**
next
case *False*
with *outer* **have** *?lhs* $=$ *karatsuba-main* (*coeffs f*) (*length* (*coeffs f*)) (*coeffs g*) (*length* (*coeffs g*))
unfolding *d* **by** *auto*
also have ... $=$ *?rhs* **unfolding** *karatsuba-main* **by** *auto*
finally show *?thesis* **.**
qed
qed
qed

lemma *karatsuba-mult-poly-code-unfold*[*code-unfold*]: $(*)$ $=$ *karatsuba-mult-poly*
by (*intro ext*, *unfold karatsuba-mult-poly*, *auto*)

The following declaration will resolve a race-conflict between $(*)$ = *karatsuba-mult-poly* and *monom* (*1*::*?'a*) *?n* $*$ *?f* $=$ *monom-mult ?n ?f*
?f $*$ *monom* (*1*::*?'a*) *?n* $=$ *monom-mult ?n ?f*.

lemmas *karatsuba-monom-mult-code-unfold*[*code-unfold*] $=$
monom-mult-unfold[**where** *f* $=$ *f* :: *'a* :: *comm-ring-1 poly* **for** *f*, *unfolded karatsuba-mult-poly-code-unfold*]

end

5.5 Record Based Version

We provide an implementation for polynomials which may be parametrized by the ring- or field-operations. These don't have to be type-based!

5.5.1 Definitions

theory *Polynomial-Record-Based*
imports
Arithmetic-Record-Based
Karatsuba-Multiplication
begin

context
fixes *ops* :: *'i arith-ops-record* (**structure**)
begin
private abbreviation (*input*) *zero* **where** *zero* $\equiv$ *arith-ops-record.zero ops*
private abbreviation (*input*) *one* **where** *one* $\equiv$ *arith-ops-record.one ops*
private abbreviation (*input*) *plus* **where** *plus* $\equiv$ *arith-ops-record.plus ops*
private abbreviation (*input*) *times* **where** *times* $\equiv$ *arith-ops-record.times ops*
private abbreviation (*input*) *minus* **where** *minus* $\equiv$ *arith-ops-record.minus ops*
private abbreviation (*input*) *uminus* **where** *uminus* $\equiv$ *arith-ops-record.uminus ops*

private abbreviation (*input*) *divide* **where** *divide* ≡ *arith-ops-record.divide ops*
private abbreviation (*input*) *inverse* **where** *inverse* ≡ *arith-ops-record.inverse ops*
private abbreviation (*input*) *modulo* **where** *modulo* ≡ *arith-ops-record.modulo ops*
private abbreviation (*input*) *normalize* **where** *normalize* ≡ *arith-ops-record.normalize ops*
private abbreviation (*input*) *unit-factor* **where** *unit-factor* ≡ *arith-ops-record.unit-factor ops*
private abbreviation (*input*) *DP* **where** *DP* ≡ *arith-ops-record.DP ops*

definition *is-poly* :: *'i list* ⇒ *bool* **where**
is-poly xs ⟷ *list-all DP xs* ∧ *no-trailing* (*HOL.eq zero*) *xs*

definition *cCons-i* :: *'i* ⇒ *'i list* ⇒ *'i list*
where
cCons-i x xs = (*if xs* = [] ∧ *x* = *zero then* [] *else x* # *xs*)

fun *plus-poly-i* :: *'i list* ⇒ *'i list* ⇒ *'i list* **where**
plus-poly-i (*x* # *xs*) (*y* # *ys*) = *cCons-i* (*plus x y*) (*plus-poly-i xs ys*)
| *plus-poly-i xs* [] = *xs*
| *plus-poly-i* [] *ys* = *ys*

definition *uminus-poly-i* :: *'i list* ⇒ *'i list* **where**
[*code-unfold*]: *uminus-poly-i* = *map uminus*

fun *minus-poly-i* :: *'i list* ⇒ *'i list* ⇒ *'i list* **where**
minus-poly-i (*x* # *xs*) (*y* # *ys*) = *cCons-i* (*minus x y*) (*minus-poly-i xs ys*)
| *minus-poly-i xs* [] = *xs*
| *minus-poly-i* [] *ys* = *uminus-poly-i ys*

abbreviation (*input*) *zero-poly-i* :: *'i list* **where**
zero-poly-i ≡ []

definition *one-poly-i* :: *'i list* **where**
[*code-unfold*]: *one-poly-i* = [*one*]

definition *smult-i* :: *'i* ⇒ *'i list* ⇒ *'i list* **where**
smult-i a pp = (*if a* = *zero then* [] *else strip-while* ((=) *zero*) (*map* (*times a*) *pp*))

definition *sdiv-i* :: *'i list* ⇒ *'i* ⇒ *'i list* **where**
sdiv-i pp a = (*strip-while* ((=) *zero*) (*map* (λ *c*. *divide c a*) *pp*))

definition *poly-of-list-i* :: *'i list* ⇒ *'i list* **where**
poly-of-list-i = *strip-while* ((=) *zero*)

fun *coeffs-minus-i* :: *'i list* ⇒ *'i list* ⇒ *'i list* **where**
coeffs-minus-i (*x* # *xs*) (*y* # *ys*) = (*minus x y* # *coeffs-minus-i xs ys*)

| *coeffs-minus-i xs* [] = *xs*
| *coeffs-minus-i* [] *ys* = *map uminus ys*

definition *monom-mult-i* :: *nat* ⇒ *'i list* ⇒ *'i list* **where**
monom-mult-i n xs = (*if xs* = [] *then xs else replicate n zero* @ *xs*)

fun *karatsuba-main-i* :: *'i list* ⇒ *nat* ⇒ *'i list* ⇒ *nat* ⇒ *'i list* **where**
karatsuba-main-i f n g m = (*if n* ≤ *karatsuba-lower-bound* ∨ *m* ≤ *karatsuba-lower-bound then*
let ff = *poly-of-list-i f in foldr* (*λa p. plus-poly-i* (*smult-i a ff*) (*cCons-i zero p*)) *g zero-poly-i*
else let n2 = *n div 2 in*
if m > *n2 then* (*case split-at n2 f of*
(*f0*,*f1*) ⇒ *case split-at n2 g of*
(*g0*,*g1*) ⇒ *let*
p1 = *karatsuba-main-i f1* (*n* − *n2*) *g1* (*m* − *n2*);
p2 = *karatsuba-main-i* (*coeffs-minus-i f1 f0*) *n2* (*coeffs-minus-i g1 g0*) *n2*;
p3 = *karatsuba-main-i f0 n2 g0 n2*
in plus-poly-i (*monom-mult-i* (*n2* + *n2*) *p1*)
(*plus-poly-i* (*monom-mult-i n2* (*plus-poly-i* (*minus-poly-i p1 p2*) *p3*)) *p3*))
else case split-at n2 f of
(*f0*,*f1*) ⇒ *let*
p1 = *karatsuba-main-i f1* (*n* − *n2*) *g m*;
p2 = *karatsuba-main-i f0 n2 g m*
in plus-poly-i (*monom-mult-i n2 p1*) *p2*)

definition *times-poly-i* :: *'i list* ⇒ *'i list* ⇒ *'i list* **where**
times-poly-i f g ≡ (*let n* = *length f*; *m* = *length g*
in (*if n* ≤ *karatsuba-lower-bound* ∨ *m* ≤ *karatsuba-lower-bound then if n* ≤ *m then*
foldr (*λa p. plus-poly-i* (*smult-i a g*) (*cCons-i zero p*)) *f zero-poly-i else*
foldr (*λa p. plus-poly-i* (*smult-i a f*) (*cCons-i zero p*)) *g zero-poly-i else*
if n ≤ *m then karatsuba-main-i g m f n else karatsuba-main-i f n g m*))

definition *coeff-i* :: *'i list* ⇒ *nat* ⇒ *'i* **where**
coeff-i = *nth-default zero*

definition *degree-i* :: *'i list* ⇒ *nat* **where**
degree-i pp ≡ *length pp* − *1*

definition *lead-coeff-i* :: *'i list* ⇒ *'i* **where**
lead-coeff-i pp = (*case pp of* [] ⇒ *zero* | - ⇒ *last pp*)

definition *monic-i* :: *'i list* ⇒ *bool* **where**
monic-i pp = (*lead-coeff-i pp* = *one*)

fun *minus-poly-rev-list-i* :: *'i list* ⇒ *'i list* ⇒ *'i list* **where**
minus-poly-rev-list-i (*x* # *xs*) (*y* # *ys*) = (*minus x y*) # (*minus-poly-rev-list-i xs ys*)

| *minus-poly-rev-list-i xs* [] = *xs*
| *minus-poly-rev-list-i* [] (*y* # *ys*) = []

fun *divmod-poly-one-main-i* :: *'i list* ⇒ *'i list* ⇒ *'i list*
⇒ *nat* ⇒ *'i list* × *'i list* **where**
divmod-poly-one-main-i q r d (*Suc n*) = (*let*
a = *hd r*;
qqq = *cCons-i a q*;
rr = *tl* (*if a* = *zero then r else minus-poly-rev-list-i r* (*map* (*times a*) *d*))
in divmod-poly-one-main-i qqq rr d n)
| *divmod-poly-one-main-i q r d 0* = (*q*,*r*)

fun *mod-poly-one-main-i* :: *'i list* ⇒ *'i list*
⇒ *nat* ⇒ *'i list* **where**
mod-poly-one-main-i r d (*Suc n*) = (*let*
a = *hd r*;
rr = *tl* (*if a* = *zero then r else minus-poly-rev-list-i r* (*map* (*times a*) *d*))
in mod-poly-one-main-i rr d n)
| *mod-poly-one-main-i r d 0* = *r*

definition *pdivmod-monic-i* :: *'i list* ⇒ *'i list* ⇒ *'i list* × *'i list* **where**
pdivmod-monic-i cf cg ≡ *case*
divmod-poly-one-main-i [] (*rev cf*) (*rev cg*) (*1* + *length cf* − *length cg*)
of (*q*,*r*) ⇒ (*poly-of-list-i q*, *poly-of-list-i* (*rev r*))

definition *dupe-monic-i* :: *'i list* ⇒ *'i list* ⇒ *'i list* ⇒ *'i list* ⇒ *'i list* ⇒ *'i list* × *'i list* **where**
dupe-monic-i D H S T U = (*case pdivmod-monic-i* (*times-poly-i T U*) *D of* (*Q*,*R*) ⇒
(*plus-poly-i* (*times-poly-i S U*) (*times-poly-i H Q*), *R*))

definition *of-int-poly-i* :: *int poly* ⇒ *'i list* **where**
of-int-poly-i f = *map* (*arith-ops-record.of-int ops*) (*coeffs f*)

definition *to-int-poly-i* :: *'i list* ⇒ *int poly* **where**
to-int-poly-i f = *poly-of-list* (*map* (*arith-ops-record.to-int ops*) *f*)

definition *dupe-monic-i-int* :: *int poly* ⇒ *int poly* ⇒ *int poly* ⇒ *int poly* ⇒ *int poly* ⇒ *int poly* × *int poly* **where**
dupe-monic-i-int D H S T = (*let*
d = *of-int-poly-i D*;
h = *of-int-poly-i H*;
s = *of-int-poly-i S*;
t = *of-int-poly-i T*
in (λ *U*. *case dupe-monic-i d h s t* (*of-int-poly-i U*) *of*
(*D'*,*H'*) ⇒ (*to-int-poly-i D'*, *to-int-poly-i H'*)))

definition *div-field-poly-i* :: *'i list* ⇒ *'i list* ⇒ *'i list* **where**
div-field-poly-i cf cg = (

if cg = [] then zero-poly-i
else let ilc = inverse (last cg); ch = map (times ilc) cg;
q = fst (divmod-poly-one-main-i [] (rev cf) (rev ch) (1 + length cf − length cg))
in poly-of-list-i ((map (times ilc) q)))

definition *mod-field-poly-i* :: *′i list ⇒ ′i list ⇒ ′i list* **where**
mod-field-poly-i cf cg = (
if cg = [] then cf
else let ilc = inverse (last cg); ch = map (times ilc) cg;
r = mod-poly-one-main-i (rev cf) (rev ch) (1 + length cf − length cg)
in poly-of-list-i (rev r))

definition *normalize-poly-i* :: *′i list ⇒ ′i list* **where**
normalize-poly-i xs = smult-i (inverse (unit-factor (lead-coeff-i xs))) xs

definition *unit-factor-poly-i* :: *′i list ⇒ ′i list* **where**
unit-factor-poly-i xs = cCons-i (unit-factor (lead-coeff-i xs)) []

fun *pderiv-main-i* :: *′i ⇒ ′i list ⇒ ′i list* **where**
pderiv-main-i f (x # xs) = cCons-i (times f x) (pderiv-main-i (plus f one) xs)
| *pderiv-main-i f [] = []*

definition *pderiv-i* :: *′i list ⇒ ′i list* **where**
pderiv-i xs = pderiv-main-i one (tl xs)

definition *dvd-poly-i* :: *′i list ⇒ ′i list ⇒ bool* **where**
dvd-poly-i xs ys = (∃ zs. is-poly zs ∧ ys = times-poly-i xs zs)

definition *irreducible-i* :: *′i list ⇒ bool* **where**
irreducible-i xs = (degree-i xs ≠ 0 ∧
(∀ q r. is-poly q ⟶ is-poly r ⟶ degree-i q < degree-i xs ⟶ degree-i r < degree-i xs
⟶ xs ≠ times-poly-i q r))

definition *poly-ops* :: *′i list arith-ops-record* **where**
poly-ops ≡ Arith-Ops-Record
zero-poly-i
one-poly-i
plus-poly-i
times-poly-i
minus-poly-i
uminus-poly-i
div-field-poly-i
(λ -. []) — not defined
mod-field-poly-i
normalize-poly-i
unit-factor-poly-i

(λ *i*. *if i* = *0 then* [] *else* [*arith-ops-record.of-int ops i*])
(λ -. *0*) — not defined
is-poly

definition *gcd-poly-i* :: *'i list* ⇒ *'i list* ⇒ *'i list* **where**
gcd-poly-i = *arith-ops.gcd-eucl-i poly-ops*

definition *euclid-ext-poly-i* :: *'i list* ⇒ *'i list* ⇒ (*'i list* × *'i list*) × *'i list* **where**
euclid-ext-poly-i = *arith-ops.euclid-ext-i poly-ops*

definition *separable-i* :: *'i list* ⇒ *bool* **where**
separable-i xs ≡ *gcd-poly-i xs* (*pderiv-i xs*) = *one-poly-i*

end

5.5.2 Properties

definition *pdivmod-monic* :: *'a*::*comm-ring-1 poly* ⇒ *'a poly* ⇒ *'a poly* × *'a poly* **where**
pdivmod-monic f g ≡ *let cg* = *coeffs g*; *cf* = *coeffs f*;
(*q*, *r*) = *divmod-poly-one-main-list* [] (*rev cf*) (*rev cg*) (*1* + *length cf* − *length cg*)
in (*poly-of-list q*, *poly-of-list* (*rev r*))

lemma *coeffs-smult'*: *coeffs* (*smult a p*) = (*if a* = *0 then* [] *else strip-while* ((=) *0*) (*map* (*Groups.times a*) (*coeffs p*)))
by (*simp add*: *coeffs-map-poly smult-conv-map-poly*)

lemma *coeffs-sdiv*: *coeffs* (*sdiv-poly p a*) = (*strip-while* ((=) *0*) (*map* (λ *x*. *x div a*) (*coeffs p*)))
unfolding *sdiv-poly-def* **by** (*rule coeffs-map-poly*)

lifting-forget *poly.lifting*

context *ring-ops*
begin

definition *poly-rel* :: *'i list* ⇒ *'a poly* ⇒ *bool* **where**
poly-rel x x' ⟷ *list-all2 R x* (*coeffs x'*)

lemma *right-total-poly-rel*[*transfer-rule*]:
right-total poly-rel
using *list.right-total-rel*[*of R*] *right-total* **unfolding** *poly-rel-def right-total-def* **by** *auto*

lemma *poly-rel-inj*: *poly-rel x y* ⟹ *poly-rel x z* ⟹ *y* = *z*
using *list.bi-unique-rel*[*OF bi-unique*] **unfolding** *poly-rel-def coeffs-eq-iff bi-unique-def* **by** *auto*

lemma *bi-unique-poly-rel*[*transfer-rule*]: *bi-unique poly-rel*
using *list.bi-unique-rel*[*OF bi-unique*] **unfolding** *poly-rel-def bi-unique-def coeffs-eq-iff* **by** *auto*

lemma *Domainp-is-poly* [*transfer-domain-rule*]:
Domainp poly-rel = *is-poly ops*
unfolding *poly-rel-def* [*abs-def*] *is-poly-def* [*abs-def*]
proof (*intro ext iffI*, *unfold Domainp-iff*)
note *DPR* = *fun-cong* [*OF list.Domainp-rel* [*of R*, *unfolded DPR*], *unfolded Domainp-iff*]
let *?no-trailing* = *no-trailing* (*HOL.eq zero*)
fix *xs*
have *no-trailing*: *no-trailing* (*HOL.eq 0*) xs' $\longleftrightarrow$ *?no-trailing xs*
if *list-all2 R xs* xs' **for** xs'
proof (*cases xs rule*: *rev-cases*)
case *Nil*
with *that* **show** *?thesis*
by *simp*
next
case (*snoc ys y*)
with *that* **have** $xs' \neq []$
by *auto*
then obtain ys' y' **where** $xs' = ys'$ @ $[y']$
by (*cases* xs' *rule*: *rev-cases*) *simp-all*
with *that snoc* **show** *?thesis*
by *simp* (*meson bi-unique bi-unique-def zero*)
qed
let *?DPR* = *arith-ops-record.DP ops*
{
assume $\exists x'$. *list-all2 R xs* (*coeffs* x')
then obtain xs' **where** ∗: *list-all2 R xs* (*coeffs* xs') **by** *auto*
with *DPR* [*of xs*] **have** *list-all ?DPR xs* **by** *auto*
then show *list-all ?DPR xs* ∧ *?no-trailing xs*
using *no-trailing* [*OF* ∗] **by** *simp*
}
{
assume *list-all ?DPR xs* ∧ *?no-trailing xs*
with *DPR* [*of xs*] **obtain** xs' **where** ∗: *list-all2 R xs* xs' **and** *?no-trailing xs*
by *auto*
from *no-trailing* [*OF* ∗] *this*(*2*) **have** *no-trailing* (*HOL.eq 0*) xs'
by *simp*
hence *coeffs* (*poly-of-list* xs') = xs' **unfolding** *poly-of-list-impl* **by** *auto*
with ∗ **show** $\exists x'$. *list-all2 R xs* (*coeffs* x') **by** *metis*
}
qed

lemma *poly-rel-zero*[*transfer-rule*]: *poly-rel zero-poly-i 0*

unfolding *poly-rel-def* **by** *auto*

lemma *poly-rel-one*[*transfer-rule*]: *poly-rel* (*one-poly-i ops*) *1*
unfolding *poly-rel-def one-poly-i-def* **by** (*simp add*: *one*)

lemma *poly-rel-cCons*[*transfer-rule*]: (*R* ===> *list-all2 R* ===> *list-all2 R*) (*cCons-i ops*) *cCons*
unfolding *cCons-i-def*[*abs-def*] *cCons-def*[*abs-def*]
by *transfer-prover*

lemma *poly-rel-pCons*[*transfer-rule*]: (*R* ===> *poly-rel* ===> *poly-rel*) (*cCons-i ops*) *pCons*
unfolding *rel-fun-def poly-rel-def coeffs-pCons-eq-cCons cCons-def*[*symmetric*]
using *poly-rel-cCons*[*unfolded rel-fun-def*] **by** *auto*

lemma *poly-rel-eq*[*transfer-rule*]: (*poly-rel* ===> *poly-rel* ===> (=)) (=) (=)
unfolding *poly-rel-def*[*abs-def*] *coeffs-eq-iff*[*abs-def*] *rel-fun-def*
by (*metis bi-unique bi-uniqueDl bi-uniqueDr list.bi-unique-rel*)

lemma *poly-rel-plus*[*transfer-rule*]: (*poly-rel* ===> *poly-rel* ===> *poly-rel*) (*plus-poly-i ops*) (+)
proof (*intro rel-funI*)
fix *x1 y1 x2 y2*
assume *poly-rel x1 x2* **and** *poly-rel y1 y2*
thus *poly-rel* (*plus-poly-i ops x1 y1*) (*x2* + *y2*)
unfolding *poly-rel-def coeffs-eq-iff coeffs-plus-eq-plus-coeffs*
proof (*induct x1 y1 arbitrary*: *x2 y2 rule*: *plus-poly-i.induct*)
case (*1 x1 xs1 y1 ys1 X2 Y2*)
from *1*(*2*) **obtain** *x2 xs2* **where** *X2*: *coeffs X2* = *x2* # *coeffs xs2*
by (*cases X2*, *auto simp*: *cCons-def split*: *if-splits*)
from *1*(*3*) **obtain** *y2 ys2* **where** *Y2*: *coeffs Y2* = *y2* # *coeffs ys2*
by (*cases Y2*, *auto simp*: *cCons-def split*: *if-splits*)
from *1*(*2*) *1*(*3*) **have** [*transfer-rule*]: *R x1 x2 R y1 y2*
and ∗: *list-all2 R xs1* (*coeffs xs2*) *list-all2 R ys1* (*coeffs ys2*) **unfolding** *X2 Y2* **by** *auto*
note [*transfer-rule*] = *1*(*1*)[*OF* ∗]
show *?case* **unfolding** *X2 Y2* **by** *simp transfer-prover*
next
case (*2 xs1 xs2 ys2*)
thus *?case* **by** (*cases coeffs xs2*, *auto*)
next
case (*3 xs2 y1 ys1 Y2*)
thus *?case* **by** (*cases Y2*, *auto simp*: *cCons-def*)
qed

qed

lemma *poly-rel-uminus*[*transfer-rule*]: (*poly-rel* ===> *poly-rel*) (*uminus-poly-i ops*) *Groups.uminus*
proof (*intro rel-funI*)
  fix *x y*
  assume *poly-rel x y*
  hence [*transfer-rule*]: *list-all2 R x* (*coeffs y*) **unfolding** *poly-rel-def* **.**
  show *poly-rel* (*uminus-poly-i ops x*) (−*y*)
    unfolding *poly-rel-def coeffs-uminus uminus-poly-i-def* **by** *transfer-prover*
qed

lemma *poly-rel-minus*[*transfer-rule*]: (*poly-rel* ===> *poly-rel* ===> *poly-rel*) (*minus-poly-i ops*) (−)
proof (*intro rel-funI*)
  fix *x1 y1 x2 y2*
  assume *poly-rel x1 x2* **and** *poly-rel y1 y2*
  thus *poly-rel* (*minus-poly-i ops x1 y1*) (*x2* − *y2*)
    unfolding *diff-conv-add-uminus*
    unfolding *poly-rel-def coeffs-eq-iff coeffs-plus-eq-plus-coeffs coeffs-uminus*
  proof (*induct x1 y1 arbitrary*: *x2 y2 rule*: *minus-poly-i.induct*)
    case (*1 x1 xs1 y1 ys1 X2 Y2*)
    from *1*(*2*) **obtain** *x2 xs2* **where** *X2*: *coeffs X2* = *x2* # *coeffs xs2*
      by (*cases X2*, *auto simp*: *cCons-def split*: *if-splits*)
    from *1*(*3*) **obtain** *y2 ys2* **where** *Y2*: *coeffs Y2* = *y2* # *coeffs ys2*
      by (*cases Y2*, *auto simp*: *cCons-def split*: *if-splits*)
    from *1*(*2*) *1*(*3*) **have** [*transfer-rule*]: *R x1 x2 R y1 y2*
      and ∗: *list-all2 R xs1* (*coeffs xs2*) *list-all2 R ys1* (*coeffs ys2*) **unfolding** *X2 Y2* **by** *auto*
    note [*transfer-rule*] = *1*(*1*)[*OF* ∗]
    show *?case* **unfolding** *X2 Y2* **by** *simp transfer-prover*
  next
    case (*2 xs1 xs2 ys2*)
    thus *?case* **by** (*cases coeffs xs2*, *auto*)
  next
    case (*3 xs2 y1 ys1 Y2*)
    from *3*(*1*) **have** *id0*: *coeffs ys1* = *coeffs 0* **by** (*cases ys1*, *auto*)
    have *id1*: *minus-poly-i ops* [] (*xs2* # *y1*) = *uminus-poly-i ops* (*xs2* # *y1*) **by** *simp*
    from *3*(*2*) **have** [*transfer-rule*]: *poly-rel* (*xs2* # *y1*) *Y2* **unfolding** *poly-rel-def* **by** *simp*
    show *?case* **unfolding** *id0 id1 coeffs-uminus*[*symmetric*] *coeffs-plus-eq-plus-coeffs*[*symmetric*] *poly-rel-def*[*symmetric*] **by** *simp transfer-prover*
  qed
qed

lemma *poly-rel-smult*[*transfer-rule*]: (*R* ===> *poly-rel* ===> *poly-rel*) (*smult-i ops*) *smult*
  unfolding *rel-fun-def poly-rel-def coeffs-smult′ smult-i-def*
proof (*intro allI impI*, *goal-cases*)
  case (*1 x y xs ys*)
  note [*transfer-rule*] = *1*
  show *?case* **by** *transfer-prover*
qed

lemma *poly-rel-coeffs*[*transfer-rule*]: (*poly-rel* ===> *list-all2 R*) (λ *x*. *x*) *coeffs*
  unfolding *rel-fun-def poly-rel-def* **by** *auto*

lemma *poly-rel-poly-of-list*[*transfer-rule*]: (*list-all2 R* ===> *poly-rel*) (*poly-of-list-i ops*) *poly-of-list*
  unfolding *rel-fun-def poly-of-list-i-def poly-rel-def poly-of-list-impl*
proof (*intro allI impI*, *goal-cases*)
  case (*1 x y*)
  note [*transfer-rule*] = *this*
  show *?case* **by** *transfer-prover*
qed

lemma *poly-rel-monom-mult*[*transfer-rule*]:
  ((=) ===> *poly-rel* ===> *poly-rel*) (*monom-mult-i ops*) *monom-mult*
  unfolding *rel-fun-def monom-mult-i-def poly-rel-def monom-mult-code Let-def*
proof (*auto*, *goal-cases*)
  case (*1 x xs y*)
  show *?case* **by** (*induct x*, *auto simp*: *1*(*3*) *zero*)
qed

declare *karatsuba-main-i.simps*[*simp del*]

lemma *list-rel-coeffs-minus-i*: **assumes** *list-all2 R x1 x2 list-all2 R y1 y2*
  shows *list-all2 R* (*coeffs-minus-i ops x1 y1*) (*coeffs-minus x2 y2*)
proof −
  note *simps* = *coeffs-minus-i.simps coeffs-minus.simps*
  show *?thesis* **using** *assms*
  proof (*induct x1 y1 arbitrary*: *x2 y2 rule*: *coeffs-minus-i.induct*)
    case (*1 x xs y ys*)
   from *1*(*2*−) **obtain** *Y Ys* **where** *y2*: *y2* = *Y* # *Ys* **unfolding** *list-all2-conv-all-nth*
by (*cases y2*, *auto*)
    with *1*(*2*−) **have** *y*: *R y Y list-all2 R ys Ys* **by** *auto*
   from *1*(*2*−) **obtain** *X Xs* **where** *x2*: *x2* = *X* # *Xs* **unfolding** *list-all2-conv-all-nth*
by (*cases x2*, *auto*)
    with *1*(*2*−) **have** *x*: *R x X list-all2 R xs Xs* **by** *auto*
    from *1*(*1*)[*OF x*(*2*) *y*(*2*)] *x*(*1*) *y*(*1*)
    show *?case* **unfolding** *x2 y2 simps* **using** *minus*[*unfolded rel-fun-def*] **by** *auto*
  next

case (*3 y ys*)
from *3* **have** *x2*: *x2* = [] **by** *auto*
from *3* **obtain** *Y Ys* **where** *y2*: *y2* = *Y* # *Ys* **unfolding** *list-all2-conv-all-nth* **by** (*cases y2*, *auto*)
obtain *y1* **where** *y1*: *y* # *ys* = *y1* **by** *auto*
show *?case* **unfolding** *y2 simps x2* **unfolding** *y2*[*symmetric*] *list-all2-map2 list-all2-map1*
using *3*(*2*) **unfolding** *y1* **using** *uminus*[*unfolded rel-fun-def*]
unfolding *list-all2-conv-all-nth* **by** *auto*
qed *auto*
qed

lemma *poly-rel-karatsuba-main*: *list-all2 R x1 x2* $\Longrightarrow$ *list-all2 R y1 y2* $\Longrightarrow$
poly-rel (*karatsuba-main-i ops x1 n y1 m*) (*karatsuba-main x2 n y2 m*)
proof (*induct n arbitrary*: *x1 y1 x2 y2 m rule*: *less-induct*)
case (*less n f g F G m*)
note *simp*[*simp*] = *karatsuba-main.simps*[*of F n G m*] *karatsuba-main-i.simps*[*of ops f n g m*]
note *IH* = *less*(*1*)
note *rel*[*transfer-rule*] = *less*(*2*−*3*)
show *?case* (**is** *poly-rel ?lhs ?rhs*)
proof (*cases* (*n* ≤ *karatsuba-lower-bound* ∨ *m* ≤ *karatsuba-lower-bound*) = *False*)
case *False*
from *False*
have *lhs*: *?lhs* = *foldr* (λ*a p. plus-poly-i ops* (*smult-i ops a* (*poly-of-list-i ops f*))
(*cCons-i ops zero p*)) *g* [] **by** *simp*
from *False* **have** *rhs*: *?rhs* = *foldr* (λ*a p. smult a* (*poly-of-list F*) + *pCons 0 p*) *G 0* **by** *simp*
show *?thesis* **unfolding** *lhs rhs* **by** *transfer-prover*
next
case *True* **note** ∗ = *this*
let *?n2* = *n div 2*
have *?n2* < *n n* − *?n2* < *n* **using** *True* **unfolding** *karatsuba-lower-bound-def* **by** *auto*
note *IH* = *IH*[*OF this*(*1*)] *IH*[*OF this*(*2*)]
obtain *f1 f0* **where** *f*: *split-at ?n2 f* = (*f0*,*f1*) **by** *force*
obtain *g1 g0* **where** *g*: *split-at ?n2 g* = (*g0*,*g1*) **by** *force*
obtain *F1 F0* **where** *F*: *split-at ?n2 F* = (*F0*,*F1*) **by** *force*
obtain *G1 G0* **where** *G*: *split-at ?n2 G* = (*G0*,*G1*) **by** *force*
from *rel f F* **have** *relf*[*transfer-rule*]: *list-all2 R f0 F0 list-all2 R f1 F1*
unfolding *split-at-def* **by** *auto*
from *rel g G* **have** *relg*[*transfer-rule*]: *list-all2 R g0 G0 list-all2 R g1 G1*
unfolding *split-at-def* **by** *auto*
show *?thesis*
proof (*cases ?n2* < *m*)
case *True*
obtain *p1 P1* **where** *p1*: *p1* = *karatsuba-main-i ops f1* (*n* − *n div 2*) *g1* (*m* − *n div 2*)

P1 = *karatsuba-main F1* (*n* − *n div 2*) *G1* (*m* − *n div 2*) **by** *auto*
obtain *p2 P2* **where** *p2*: *p2* = *karatsuba-main-i ops* (*coeffs-minus-i ops f1 f0*) (*n div 2*)
(*coeffs-minus-i ops g1 g0*) (*n div 2*)
P2 = *karatsuba-main* (*coeffs-minus F1 F0*) (*n div 2*)
(*coeffs-minus G1 G0*) (*n div 2*) **by** *auto*
obtain *p3 P3* **where** *p3*: *p3* = *karatsuba-main-i ops f0* (*n div 2*) *g0* (*n div 2*)
P3 = *karatsuba-main F0* (*n div 2*) *G0* (*n div 2*) **by** *auto*
from ∗ *True* **have** *lhs*: *?lhs* = *plus-poly-i ops* (*monom-mult-i ops* (*n div 2* + *n div 2*) *p1*)
(*plus-poly-i ops*
(*monom-mult-i ops* (*n div 2*)
(*plus-poly-i ops* (*minus-poly-i ops p1 p2*) *p3*)) *p3*)
unfolding *simp Let-def f g split p1 p2 p3* **by** *auto*
have [*transfer-rule*]: *poly-rel p1 P1* **using** *IH*(*2*)[*OF relf*(*2*) *relg*(*2*)] **unfolding** *p1* **.**
have [*transfer-rule*]: *poly-rel p3 P3* **using** *IH*(*1*)[*OF relf*(*1*) *relg*(*1*)] **unfolding** *p3* **.**
have [*transfer-rule*]: *poly-rel p2 P2* **unfolding** *p2*
by (*rule IH*(*1*)[*OF list-rel-coeffs-minus-i list-rel-coeffs-minus-i*], *insert relf relg*)
from *True* ∗ **have** *rhs*: *?rhs* = *monom-mult* (*n div 2* + *n div 2*) *P1* +
(*monom-mult* (*n div 2*) (*P1* − *P2* + *P3*) + *P3*)
unfolding *simp Let-def F G split p1 p2 p3* **by** *auto*
show *?thesis* **unfolding** *lhs rhs* **by** *transfer-prover*
next
case *False*
obtain *p1 P1* **where** *p1*: *p1* = *karatsuba-main-i ops f1* (*n* − *n div 2*) *g m*
P1 = *karatsuba-main F1* (*n* − *n div 2*) *G m* **by** *auto*
obtain *p2 P2* **where** *p2*: *p2* = *karatsuba-main-i ops f0* (*n div 2*) *g m*
P2 = *karatsuba-main F0* (*n div 2*) *G m* **by** *auto*
from ∗ *False* **have** *lhs*: *?lhs* = *plus-poly-i ops* (*monom-mult-i ops* (*n div 2*) *p1*) *p2*
unfolding *simp Let-def f split p1 p2* **by** *auto*
from ∗ *False* **have** *rhs*: *?rhs* = *monom-mult* (*n div 2*) *P1* + *P2*
unfolding *simp Let-def F split p1 p2* **by** *auto*
have [*transfer-rule*]: *poly-rel p1 P1* **using** *IH*(*2*)[*OF relf*(*2*) *rel*(*2*)] **unfolding** *p1* **.**
have [*transfer-rule*]: *poly-rel p2 P2* **using** *IH*(*1*)[*OF relf*(*1*) *rel*(*2*)] **unfolding** *p2* **.**
show *?thesis* **unfolding** *lhs rhs* **by** *transfer-prover*
qed
qed
qed

lemma *poly-rel-times*[*transfer-rule*]: (*poly-rel* ===> *poly-rel* ===> *poly-rel*) (*times-poly-i ops*) ((∗))

proof (*intro rel-funI*)
fix *x1 y1 x2 y2*
assume *x12*[*transfer-rule*]: *poly-rel x1 x2* **and** *y12* [*transfer-rule*]: *poly-rel y1 y2*
hence *X12*[*transfer-rule*]: *list-all2 R x1* (*coeffs x2*) **and** *Y12*[*transfer-rule*]: *list-all2 R y1* (*coeffs y2*)
unfolding *poly-rel-def* **by** *auto*
hence *len*: *length* (*coeffs x2*) = *length x1 length* (*coeffs y2*) = *length y1*
unfolding *list-all2-conv-all-nth* **by** *auto*
let *?cond1* = *length x1* $\leq$ *karatsuba-lower-bound* $\vee$ *length y1* $\leq$ *karatsuba-lower-bound*

let *?cond2* = *length x1* $\leq$ *length y1*
note *d* = *karatsuba-mult-poly*[*symmetric*] *karatsuba-mult-poly-def Let-def times-poly-i-def len if-True if-False*
consider (*TT*) *?cond1* = *True ?cond2* = *True* | (*TF*) *?cond1* = *True ?cond2* = *False*
| (*FT*) *?cond1* = *False ?cond2* = *True* | (*FF*) *?cond1* = *False ?cond2* = *False*
by *auto*
thus *poly-rel* (*times-poly-i ops x1 y1*) (*x2* $*$ *y2*)
proof (*cases*)
case *TT*
show *?thesis* **unfolding** *d TT*
unfolding *poly-rel-def coeffs-eq-iff times-poly-def times-poly-i-def fold-coeffs-def*
by *transfer-prover*
next
case *TF*
show *?thesis* **unfolding** *d TF*
unfolding *poly-rel-def coeffs-eq-iff times-poly-def times-poly-i-def fold-coeffs-def*
by *transfer-prover*
next
case *FT*
show *?thesis* **unfolding** *d FT*
by (*rule poly-rel-karatsuba-main*[*OF Y12 X12*])
next
case *FF*
show *?thesis* **unfolding** *d FF*
by (*rule poly-rel-karatsuba-main*[*OF X12 Y12*])
qed
qed

lemma *poly-rel-coeff*[*transfer-rule*]: (*poly-rel* ===> (=) ===> *R*) (*coeff-i ops*) *coeff*
unfolding *poly-rel-def rel-fun-def coeff-i-def nth-default-coeffs-eq*[*symmetric*]
proof (*intro allI impI*, *clarify*)
fix *x y n*
assume [*transfer-rule*]: *list-all2 R x* (*coeffs y*)
show *R* (*nth-default zero x n*) (*nth-default 0* (*coeffs y*) *n*) **by** *transfer-prover*
qed

lemma *poly-rel-degree*[*transfer-rule*]: (*poly-rel* ===> (=)) *degree-i degree*
  unfolding *poly-rel-def rel-fun-def degree-i-def degree-eq-length-coeffs*
  by (*simp add*: *list-all2-lengthD*)

lemma *lead-coeff-i-def'*: *lead-coeff-i ops x* = (*coeff-i ops*) *x* (*degree-i x*)
  unfolding *lead-coeff-i-def degree-i-def coeff-i-def*
proof (*cases x*, *auto*, *goal-cases*)
  case (*1 a xs*)
  hence *id*: *last xs* = *last* (*a* # *xs*) **by** *auto*
  show *?case* **unfolding** *id* **by** (*subst last-conv-nth-default*, *auto*)
qed

lemma *poly-rel-lead-coeff*[*transfer-rule*]: (*poly-rel* ===> *R*) (*lead-coeff-i ops*) *lead-coeff*
  unfolding *lead-coeff-i-def'* [*abs-def*] **by** *transfer-prover*

lemma *poly-rel-minus-poly-rev-list*[*transfer-rule*]:
  (*list-all2 R* ===> *list-all2 R* ===> *list-all2 R*) (*minus-poly-rev-list-i ops*) *minus-poly-rev-list*
proof (*intro rel-funI*, *goal-cases*)
  case (*1 x1 x2 y1 y2*)
  thus *?case*
  proof (*induct x1 y1 arbitrary*: *x2 y2 rule*: *minus-poly-rev-list-i.induct*)
    case (*1 x1 xs1 y1 ys1 X2 Y2*)
    from *1*(*2*) **obtain** *x2 xs2* **where** *X2*: *X2* = *x2* # *xs2* **by** (*cases X2*, *auto*)
    from *1*(*3*) **obtain** *y2 ys2* **where** *Y2*: *Y2* = *y2* # *ys2* **by** (*cases Y2*, *auto*)
    from *1*(*2*) *1*(*3*) **have** [*transfer-rule*]: *R x1 x2 R y1 y2*
      and ∗: *list-all2 R xs1 xs2 list-all2 R ys1 ys2* **unfolding** *X2 Y2* **by** *auto*
    note [*transfer-rule*] = *1*(*1*)[*OF* ∗]
    show *?case* **unfolding** *X2 Y2* **by** (*simp*, *intro conjI*, *transfer-prover*+)
  next
    case (*2 xs1 xs2 ys2*)
    thus *?case* **by** (*cases xs2*, *auto*)
  next
    case (*3 xs2 y1 ys1 Y2*)
    thus *?case* **by** (*cases Y2*, *auto*)
  qed
qed

lemma *divmod-poly-one-main-i*: **assumes** *len*: *n* ≤ *length Y* **and** *rel*: *list-all2 R x X list-all2 R y Y*
    list-all2 R z Z **and** *n*: *n* = *N*
 shows *rel-prod* (*list-all2 R*) (*list-all2 R*) (*divmod-poly-one-main-i ops x y z n*)
    (*divmod-poly-one-main-list X Y Z N*)
   using *len rel* **unfolding** *n*
proof (*induct N arbitrary*: *x X y Y z Z*)

case (*Suc n x X y Y z Z*)
from *Suc*(*2*,*4*) **have** [*transfer-rule*]: *R* (*hd y*) (*hd Y*) **by** (*cases y*; *cases Y*, *auto*)

note [*transfer-rule*] = *Suc*(*3*−*5*)
have *id*: *?case* = (*rel-prod* (*list-all2 R*) (*list-all2 R*)
(*divmod-poly-one-main-i ops* (*cCons-i ops* (*hd y*) *x*)
(*tl* (*if hd y* = *zero then y else minus-poly-rev-list-i ops y* (*map* (*times* (*hd y*)) *z*))) *z n*)
(*divmod-poly-one-main-list* (*cCons* (*hd Y*) *X*)
(*tl* (*if hd Y* = *0 then Y else minus-poly-rev-list Y* (*map* ((∗) (*hd Y*)) *Z*))) *Z n*))
by (*simp add*: *Let-def*)
show *?case* **unfolding** *id*
proof (*rule Suc*(*1*), *goal-cases*)
case *1*
show *?case* **using** *Suc*(*2*) **by** *simp*
qed (*transfer-prover*+)
qed *simp*

lemma *mod-poly-one-main-i*: **assumes** *len*: $n \leq$ *length X* **and** *rel*: *list-all2 R x X list-all2 R y Y*
and *n*: $n = N$
shows *list-all2 R* (*mod-poly-one-main-i ops x y n*)
(*mod-poly-one-main-list X Y N*)
using *len rel* **unfolding** *n*
proof (*induct N arbitrary*: *x X y Y*)
case (*Suc n y Y z Z*)
from *Suc*(*2*,*3*) **have** [*transfer-rule*]: *R* (*hd y*) (*hd Y*) **by** (*cases y*; *cases Y*, *auto*)

note [*transfer-rule*] = *Suc*(*3*−*4*)
have *id*: *?case* = (*list-all2 R*
(*mod-poly-one-main-i ops*
(*tl* (*if hd y* = *zero then y else minus-poly-rev-list-i ops y* (*map* (*times* (*hd y*)) *z*))) *z n*)
(*mod-poly-one-main-list*
(*tl* (*if hd Y* = *0 then Y else minus-poly-rev-list Y* (*map* ((∗) (*hd Y*)) *Z*))) *Z n*))
by (*simp add*: *Let-def*)
show *?case* **unfolding** *id*
proof (*rule Suc*(*1*), *goal-cases*)
case *1*
show *?case* **using** *Suc*(*2*) **by** *simp*
qed (*transfer-prover*+)
qed *simp*

lemma *poly-rel-dvd*[*transfer-rule*]: (*poly-rel* ===> *poly-rel* ===> (=)) (*dvd-poly-i ops*) (*dvd*)
unfolding *dvd-poly-i-def*[*abs-def*] *dvd-def*[*abs-def*]

by (*transfer-prover-start*, *transfer-step*+, *auto*)

lemma *poly-rel-monic*[*transfer-rule*]: (*poly-rel* ===> (=)) (*monic-i ops*) *monic*
unfolding *monic-i-def lead-coeff-i-def′* **by** *transfer-prover*

lemma *poly-rel-pdivmod-monic*: **assumes** *mon*: *monic Y*
and *x*: *poly-rel x X* **and** *y*: *poly-rel y Y*
shows *rel-prod poly-rel poly-rel* (*pdivmod-monic-i ops x y*) (*pdivmod-monic X Y*)
proof −
note [*transfer-rule*] = *x y*
note *listall* = *this*[*unfolded poly-rel-def*]
note *defs* = *pdivmod-monic-def pdivmod-monic-i-def Let-def*
from *mon* **obtain** *k* **where** *len*: *length* (*coeffs Y*) = *Suc k* **unfolding** *poly-rel-def list-all2-iff*
by (*cases coeffs Y*, *auto*)
have [*transfer-rule*]:
rel-prod (*list-all2 R*) (*list-all2 R*)
(*divmod-poly-one-main-i ops* [] (*rev x*) (*rev y*) (*1* + *length x* − *length y*))
(*divmod-poly-one-main-list* [] (*rev* (*coeffs X*)) (*rev* (*coeffs Y*)) (*1* + *length* (*coeffs X*) − *length* (*coeffs Y*)))
by (*rule divmod-poly-one-main-i*, *insert x y listall*, *auto*, *auto simp*: *poly-rel-def list-all2-iff len*)
show *?thesis* **unfolding** *defs* **by** *transfer-prover*
qed

lemma *ring-ops-poly*: *ring-ops* (*poly-ops ops*) *poly-rel*
by (*unfold-locales*, *auto simp*: *poly-ops-def*
bi-unique-poly-rel
right-total-poly-rel
poly-rel-times
poly-rel-zero
poly-rel-one
poly-rel-minus
poly-rel-uminus
poly-rel-plus
poly-rel-eq
Domainp-is-poly)
end

context *idom-ops*
begin

lemma *poly-rel-pderiv* [*transfer-rule*]: (*poly-rel* ===> *poly-rel*) (*pderiv-i ops*) *pderiv*
proof (*intro rel-funI*, *unfold poly-rel-def coeffs-pderiv-code pderiv-i-def pderiv-coeffs-def*)
fix *xs xs′*
assume *list-all2 R xs* (*coeffs xs′*)
then obtain *ys ys′ y y′* **where** *id*: *tl xs* = *ys tl* (*coeffs xs′*) = *ys′ one* = *y 1* = *y′* **and**

R: *list-all2 R ys ys′ R y y′*
by (*cases xs*; *cases coeffs xs′*; *auto simp*: *one*)
show *list-all2 R* (*pderiv-main-i ops one* (*tl xs*))
(*pderiv-coeffs-code 1* (*tl* (*coeffs xs′*)))
unfolding *id* **using** *R*
proof (*induct ys ys′ arbitrary*: *y y′ rule*: *list-all2-induct*)
case (*Cons x xs x′ xs′ y y′*)
note [*transfer-rule*] = *Cons*(*1*,*2*,*4*)
have *R* (*plus y one*) (*y′* + *1*) **by** *transfer-prover*
note [*transfer-rule*] = *Cons*(*3*)[*OF this*]
show *?case* **by** (*simp*, *transfer-prover*)
qed *simp*
qed

lemma *poly-rel-irreducible*[*transfer-rule*]: (*poly-rel* ===> (=)) (*irreducible-i ops*) $irreducible_d$
unfolding *irreducible-i-def*[*abs-def*] $irreducible_d$-*def*[*abs-def*]
by (*transfer-prover-start*, *transfer-step*+, *auto*)

lemma *idom-ops-poly*: *idom-ops* (*poly-ops ops*) *poly-rel*
using *ring-ops-poly* **unfolding** *ring-ops-def idom-ops-def* **by** *auto*
end

context *idom-divide-ops*
begin

lemma *poly-rel-sdiv*[*transfer-rule*]: (*poly-rel* ===> *R* ===> *poly-rel*) (*sdiv-i ops*) *sdiv-poly*
unfolding *rel-fun-def poly-rel-def coeffs-sdiv sdiv-i-def*
proof (*intro allI impI*, *goal-cases*)
case (*1 x y xs ys*)
note [*transfer-rule*] = *1*
show *?case* **by** *transfer-prover*
qed
end

context *field-ops*
begin

lemma *poly-rel-div*[*transfer-rule*]: (*poly-rel* ===> *poly-rel* ===> *poly-rel*)
(*div-field-poly-i ops*) (*div*)
proof (*intro rel-funI*, *goal-cases*)
case (*1 x X y Y*)
note [*transfer-rule*] = *this*
note *listall* = *this*[*unfolded poly-rel-def*]
note *defs* = *div-field-poly-impl div-field-poly-impl-def div-field-poly-i-def Let-def*
show *?case*
proof (*cases y* = [])
case *True*

with *1*(*2*) **have** *nil*: *coeffs Y* = [] **unfolding** *poly-rel-def* **by** *auto*
show *?thesis* **unfolding** *defs True nil poly-rel-def* **by** *auto*
next
case *False*
from *append-butlast-last-id*[*OF False*] **obtain** *ys yl* **where** *y*: *y* = *ys* @ [*yl*] **by** *metis*
from *False listall* **have** *coeffs Y* ≠ [] **by** *auto*
from *append-butlast-last-id*[*OF this*] **obtain** *Ys Yl* **where** *Y*: *coeffs Y* = *Ys* @ [*Yl*] **by** *metis*
from *listall* **have** [*transfer-rule*]: *R yl Yl* **by** (*simp add*: *y Y*)
have *id*: *last* (*coeffs Y*) = *Yl last* (*y*) = *yl*
⋀ *t e*. (*if y* = [] *then t else e*) = *e*
⋀ *t e*. (*if coeffs Y* = [] *then t else e*) = *e* **unfolding** *y Y* **by** *auto*
have [*transfer-rule*]: (*rel-prod* (*list-all2 R*) (*list-all2 R*))
(*divmod-poly-one-main-i ops* [] (*rev x*) (*rev* (*map* (*times* (*inverse yl*)) *y*))
(*1* + *length x* − *length y*))
(*divmod-poly-one-main-list* [] (*rev* (*coeffs X*))
(*rev* (*map* ((∗) (*Fields.inverse Yl*)) (*coeffs Y*)))
(*1* + *length* (*coeffs X*) − *length* (*coeffs Y*)))
proof (*rule divmod-poly-one-main-i*, *goal-cases*)
case *5*
from *listall* **show** *?case* **by** (*simp add*: *list-all2-lengthD*)
next
case *1*
from *listall* **show** *?case* **by** (*simp add*: *list-all2-lengthD Y*)
qed *transfer-prover*+
show *?thesis* **unfolding** *defs id* **by** *transfer-prover*
qed
qed

lemma *poly-rel-mod*[*transfer-rule*]: (*poly-rel* ===> *poly-rel* ===> *poly-rel*)
(*mod-field-poly-i ops*) (*mod*)
proof (*intro rel-funI*, *goal-cases*)
case (*1 x X y Y*)
note [*transfer-rule*] = *this*
note *listall* = *this*[*unfolded poly-rel-def*]
note *defs* = *mod-poly-code mod-field-poly-i-def Let-def*
show *?case*
proof (*cases y* = [])
case *True*
with *1*(*2*) **have** *nil*: *coeffs Y* = [] **unfolding** *poly-rel-def* **by** *auto*
show *?thesis* **unfolding** *defs True nil poly-rel-def* **by** (*simp add*: *listall*)
next
case *False*
from *append-butlast-last-id*[*OF False*] **obtain** *ys yl* **where** *y*: *y* = *ys* @ [*yl*] **by** *metis*
from *False listall* **have** *coeffs Y* ≠ [] **by** *auto*
from *append-butlast-last-id*[*OF this*] **obtain** *Ys Yl* **where** *Y*: *coeffs Y* = *Ys*

```
@ [Yl] by metis
    from listall have [transfer-rule]: R yl Yl by (simp add: y Y)
    have id: last (coeffs Y) = Yl last (y) = yl
      ⋀ t e. (if y = [] then t else e) = e
      ⋀ t e. (if coeffs Y = [] then t else e) = e unfolding y Y by auto
    have [transfer-rule]: list-all2 R
      (mod-poly-one-main-i ops (rev x) (rev (map (times (inverse yl)) y))
        (1 + length x − length y))
      (mod-poly-one-main-list (rev (coeffs X))
                (rev (map ((∗) (Fields.inverse Yl)) (coeffs Y)))
                (1 + length (coeffs X) − length (coeffs Y)))
    proof (rule mod-poly-one-main-i, goal-cases)
      case 4
      from listall show ?case by (simp add: list-all2-lengthD)
    next
      case 1
      from listall show ?case by (simp add: list-all2-lengthD Y)
    qed transfer-prover+
    show ?thesis unfolding defs id by transfer-prover
  qed
qed


lemma poly-rel-normalize [transfer-rule]: (poly-rel ===> poly-rel)
  (normalize-poly-i ops) Rings.normalize
  unfolding normalize-poly-old-def normalize-poly-i-def lead-coeff-i-def′
  by transfer-prover


lemma poly-rel-unit-factor [transfer-rule]: (poly-rel ===> poly-rel)
  (unit-factor-poly-i ops) Rings.unit-factor
  unfolding unit-factor-poly-def unit-factor-poly-i-def lead-coeff-i-def′
  unfolding monom-0 by transfer-prover

lemma idom-divide-ops-poly: idom-divide-ops (poly-ops ops) poly-rel
proof −
  interpret poly: idom-ops poly-ops ops poly-rel by (rule idom-ops-poly)
  show ?thesis
    by (unfold-locales, simp add: poly-rel-div poly-ops-def)
qed

lemma euclidean-ring-ops-poly: euclidean-ring-ops (poly-ops ops) poly-rel
proof −
  interpret poly: idom-ops poly-ops ops poly-rel by (rule idom-ops-poly)
  have id: arith-ops-record.normalize (poly-ops ops) = normalize-poly-i ops
    arith-ops-record.unit-factor (poly-ops ops) = unit-factor-poly-i ops
    unfolding poly-ops-def by simp-all
  show ?thesis
    by (unfold-locales, simp add: poly-rel-mod poly-ops-def, unfold id,
```

simp add: *poly-rel-normalize*, *insert poly-rel-div poly-rel-unit-factor*,
auto simp: *poly-ops-def*)
qed

lemma *poly-rel-gcd* [*transfer-rule*]: (*poly-rel* ===> *poly-rel* ===> *poly-rel*) (*gcd-poly-i ops*) *gcd*
proof −
interpret *poly*: *euclidean-ring-ops poly-ops ops poly-rel* **by** (*rule euclidean-ring-ops-poly*)
show *?thesis* **using** *poly.gcd-eucl-i* **unfolding** *gcd-poly-i-def gcd-eucl* **.**
qed

lemma *poly-rel-euclid-ext* [*transfer-rule*]: (*poly-rel* ===> *poly-rel* ===>
rel-prod (*rel-prod poly-rel poly-rel*) *poly-rel*) (*euclid-ext-poly-i ops*) *euclid-ext*
proof −
interpret *poly*: *euclidean-ring-ops poly-ops ops poly-rel* **by** (*rule euclidean-ring-ops-poly*)
show *?thesis* **using** *poly.euclid-ext-i* **unfolding** *euclid-ext-poly-i-def* **.**
qed

end

context *ring-ops*
begin
notepad
begin
fix *xs x ys y*
assume [*transfer-rule*]: *poly-rel xs x poly-rel ys y*
have $x * y = y * x$ **by** *simp*
from *this*[*untransferred*]
have *times-poly-i ops xs ys* = *times-poly-i ops ys xs* **.**
end
end
end

5.5.3 Over a Finite Field

theory *Poly-Mod-Finite-Field-Record-Based*
imports
Poly-Mod-Finite-Field
Finite-Field-Record-Based
Polynomial-Record-Based
begin

locale *arith-ops-record* = *arith-ops ops* + *poly-mod m* **for** *ops* :: *'i arith-ops-record*
and *m* :: *int*
begin

definition *M-rel-i* :: *'i* ⇒ *int* ⇒ *bool* **where**
M-rel-i f F = (*arith-ops-record.to-int ops f* = *M F*)

definition *Mp-rel-i* :: *'i list* ⇒ *int poly* ⇒ *bool* **where**
Mp-rel-i f F = (*map* (*arith-ops-record.to-int ops*) *f* = *coeffs* (*Mp F*))

lemma *Mp-rel-i-Mp*[*simp*]: *Mp-rel-i f* (*Mp F*) = *Mp-rel-i f F* **unfolding** *Mp-rel-i-def*
by *auto*

lemma *Mp-rel-i-Mp-to-int-poly-i*: *Mp-rel-i f F* ⟹ *Mp* (*to-int-poly-i ops f*) = *to-int-poly-i ops f*
unfolding *Mp-rel-i-def to-int-poly-i-def* **by** *simp*
end

locale *mod-ring-gen* = *ring-ops ff-ops R* **for** *ff-ops* :: *'i arith-ops-record* **and**
R :: *'i* ⇒ *'a* :: *nontriv mod-ring* ⇒ *bool* +
fixes *p* :: *int*
assumes *p*: *p* = *int CARD*(*'a*)
and *of-int*: *0* ≤ *x* ⟹ *x* < *p* ⟹ *R* (*arith-ops-record.of-int ff-ops x*) (*of-int x*)
and *to-int*: *R y z* ⟹ *arith-ops-record.to-int ff-ops y* = *to-int-mod-ring z*
and *to-int'*: *0* ≤ *arith-ops-record.to-int ff-ops y* ⟹ *arith-ops-record.to-int ff-ops y* < *p* ⟹
R y (*of-int* (*arith-ops-record.to-int ff-ops y*))
begin

lemma *nat-p*: *nat p* = *CARD*(*'a*) **unfolding** *p* **by** *simp*

sublocale *poly-mod-type p TYPE*(*'a*)
by (*unfold-locales*, *rule p*)

lemma *coeffs-to-int-poly*: *coeffs* (*to-int-poly* (*x* :: *'a mod-ring poly*)) = *map to-int-mod-ring* (*coeffs x*)
by (*rule coeffs-map-poly*, *auto*)

lemma *coeffs-of-int-poly*: *coeffs* (*of-int-poly* (*Mp x*) :: *'a mod-ring poly*) = *map of-int* (*coeffs* (*Mp x*))
apply (*rule coeffs-map-poly*)
by (*metis M-0 M-M Mp-coeff leading-coeff-0-iff of-int-hom.hom-zero to-int-mod-ring-of-int-M*)

lemma *to-int-poly-i*: **assumes** *poly-rel f g* **shows** *to-int-poly-i ff-ops f* = *to-int-poly g*
proof −
have ∗: *map* (*arith-ops-record.to-int ff-ops*) *f* = *coeffs* (*to-int-poly g*)
unfolding *coeffs-to-int-poly*
by (*rule nth-equalityI*, *insert assms*, *auto simp*: *list-all2-conv-all-nth poly-rel-def to-int*)
show *?thesis* **unfolding** *coeffs-eq-iff to-int-poly-i-def poly-of-list-def coeffs-Poly* ∗ *strip-while-coeffs*..
qed

lemma *poly-rel-of-int-poly*: **assumes** *id*: $f' = $ *of-int-poly-i ff-ops* (*Mp f*) $f'' =$ *of-int-poly* (*Mp f*)
  shows *poly-rel* f' f'' **unfolding** *id poly-rel-def*
  unfolding *list-all2-conv-all-nth coeffs-of-int-poly of-int-poly-i-def length-map*
  by (*rule conjI*[*OF refl*], *intro allI impI*, *simp add*: *nth-coeffs-coeff Mp-coeff M-def*, *rule of-int*,
    insert p, *auto*)

sublocale *arith-ops-record ff-ops p* **.**

lemma *Mp-rel-iI*: *poly-rel f1 f2* $\Longrightarrow$ *MP-Rel f3 f2* $\Longrightarrow$ *Mp-rel-i f1 f3*
  unfolding *Mp-rel-i-def MP-Rel-def poly-rel-def*
  by (*auto simp add*: *list-all2-conv-all-nth to-int intro*: *nth-equalityI*)

lemma *M-rel-iI*: *R f1 f2* $\Longrightarrow$ *M-Rel f3 f2* $\Longrightarrow$ *M-rel-i f1 f3*
  unfolding *M-rel-i-def M-Rel-def* **by** (*simp add*: *to-int*)

lemma *M-rel-iI'*: **assumes** *R f1 f2*
  shows *M-rel-i f1* (*arith-ops-record.to-int ff-ops f1*)
  by (*rule M-rel-iI*[*OF assms*], *simp add*: *to-int*[*OF assms*] *M-Rel-def M-to-int-mod-ring*)

lemma *Mp-rel-iI'*: **assumes** *poly-rel f1 f2*
  shows *Mp-rel-i f1* (*to-int-poly-i ff-ops f1*)
proof (*rule Mp-rel-iI*[*OF assms*], *unfold to-int-poly-i*[*OF assms*])
  show *MP-Rel* (*to-int-poly f2*) *f2* **unfolding** *MP-Rel-def* **by** (*simp add*: *Mp-to-int-poly*)
qed

lemma *M-rel-iD*: **assumes** *M-rel-i f1 f3*
  shows
    R f1 (*of-int* (*M f3*))
    M-Rel f3 (*of-int* (*M f3*))
proof −
  show *M-Rel f3* (*of-int* (*M f3*))
    using *M-Rel-def to-int-mod-ring-of-int-M* **by** *auto*
  from *assms* **show** *R f1* (*of-int* (*M f3*))
    unfolding *M-rel-i-def*
     by (*metis int-one-le-iff-zero-less leD linear m1 poly-mod.M-def pos-mod-conj to-int'*)
qed

lemma *Mp-rel-iD*: **assumes** *Mp-rel-i f1 f3*
  shows
    poly-rel f1 (*of-int-poly* (*Mp f3*))
    MP-Rel f3 (*of-int-poly* (*Mp f3*))
proof −
  show *Rel*: *MP-Rel f3* (*of-int-poly* (*Mp f3*))
    using *MP-Rel-def Mp-Mp Mp-f-representative* **by** *auto*
  let *?ti* = *arith-ops-record.to-int ff-ops*

from *assms*[*unfolded Mp-rel-i-def*] **have**
∗: *coeffs* (*Mp f3*) = *map ?ti f1* **by** *auto*
{
fix *x*
assume $x \in$ *set f1*
hence *?ti* $x \in$ *set* (*map ?ti f1*) **by** *auto*
from *this*[*folded ∗*] **have** *?ti* $x \in$ *range M*
by (*metis* (*no-types*, *lifting*) *MP-Rel-def M-to-int-mod-ring Rel coeffs-to-int-poly ex-map-conv range-eqI*)
hence *?ti* $x \geq 0$ *?ti* $x < p$ **unfolding** *M-def* **using** *m1* **by** *auto*
hence *R x* (*of-int* (*?ti x*))
by (*rule to-int′*)
}
thus *poly-rel f1* (*of-int-poly* (*Mp f3*)) **using** *∗*
unfolding *poly-rel-def coeffs-of-int-poly*
by (*auto simp*: *list-all2-map2 list-all2-same*)
qed
end

locale *prime-field-gen* = *field-ops ff-ops R* + *mod-ring-gen ff-ops R p* **for** *ff-ops* :: *′i arith-ops-record* **and**
R :: *′i* ⇒ *′a* :: *prime-card mod-ring* ⇒ *bool* **and** *p* :: *int*
begin

sublocale *poly-mod-prime-type p TYPE*(*′a*)
by (*unfold-locales*, *rule p*)

end

lemma (**in** *mod-ring-locale*) *mod-ring-rel-of-int*:
$0 \leq x \Longrightarrow x < p \Longrightarrow$ *mod-ring-rel x* (*of-int x*)
unfolding *mod-ring-rel-def*
by (*transfer*, *auto simp*: *p*)

context *prime-field*
begin

lemma *prime-field-finite-field-ops-int*: *prime-field-gen* (*finite-field-ops-int p*) *mod-ring-rel p*
proof −
interpret *field-ops finite-field-ops-int p mod-ring-rel* **by** (*rule finite-field-ops-int*)
show *?thesis*
by (*unfold-locales*, *rule p*,
auto simp: *finite-field-ops-int-def p mod-ring-rel-def of-int-of-int-mod-ring*)
qed

lemma *prime-field-finite-field-ops-integer*: *prime-field-gen* (*finite-field-ops-integer* (*integer-of-int p*)) *mod-ring-rel-integer p*

proof −
interpret *field-ops finite-field-ops-integer* (*integer-of-int p*) *mod-ring-rel-integer*
by (*rule finite-field-ops-integer*, *simp*)
have *pp*: $p = $ *int-of-integer* (*integer-of-int p*) **by** *auto*
interpret *int*: *prime-field-gen finite-field-ops-int p mod-ring-rel*
by (*rule prime-field-finite-field-ops-int*)
show *?thesis*
by (*unfold-locales*, *rule p*, *auto simp*: *finite-field-ops-integer-def*
mod-ring-rel-integer-def[*OF pp*] *urel-integer-def*[*OF pp*] *mod-ring-rel-of-int*
int.to-int[*symmetric*] *finite-field-ops-int-def*)
qed

lemma *prime-field-finite-field-ops32*: **assumes** *small*: $p \leq 65535$
shows *prime-field-gen* (*finite-field-ops32* (*uint32-of-int p*)) *mod-ring-rel32 p*
proof −
let *?pp* = *uint32-of-int p*
have *ppp*: *p* = *int-of-uint32 ?pp*
by (*subst int-of-uint32-inv*, *insert small p2*, *auto*)
note ∗ = *ppp small*
interpret *field-ops finite-field-ops32 ?pp mod-ring-rel32*
by (*rule finite-field-ops32*, *insert* ∗)
interpret *int*: *prime-field-gen finite-field-ops-int p mod-ring-rel*
by (*rule prime-field-finite-field-ops-int*)
show *?thesis*
proof (*unfold-locales*, *rule p*, *auto simp*: *finite-field-ops32-def*)
fix *x*
assume *x*: $0 \leq x$ $x < p$
from *int.of-int*[*OF this*] **have** *mod-ring-rel x* (*of-int x*) **by** (*simp add*: *finite-field-ops-int-def*)
thus *mod-ring-rel32* (*uint32-of-int x*) (*of-int x*) **unfolding** *mod-ring-rel32-def*[*OF* ∗]
by (*intro exI*[*of - x*], *auto simp*: *urel32-def*[*OF* ∗], *subst int-of-uint32-inv*, *insert* ∗ *x*, *auto*)
next
fix *y z*
assume *mod-ring-rel32 y z*
from *this*[*unfolded mod-ring-rel32-def*[*OF* ∗]] **obtain** *x* **where** *yx*: *urel32 y x* **and** *xz*: *mod-ring-rel x z* **by** *auto*
from *int.to-int*[*OF xz*] **have** *zx*: *to-int-mod-ring z* = *x* **by** (*simp add*: *finite-field-ops-int-def*)
show *int-of-uint32 y* = *to-int-mod-ring z* **unfolding** *zx* **using** *yx* **unfolding** *urel32-def*[*OF* ∗] **by** *simp*
next
fix *y*
show $0 \leq$ *int-of-uint32 y* $\Longrightarrow$ *int-of-uint32 y* $< p \Longrightarrow$ *mod-ring-rel32 y* (*of-int* (*int-of-uint32 y*))
unfolding *mod-ring-rel32-def*[*OF* ∗] *urel32-def*[*OF* ∗]
by (*intro exI*[*of - int-of-uint32 y*], *auto simp*: *mod-ring-rel-of-int*)
qed

qed

lemma *prime-field-finite-field-ops64*: **assumes** *small*: $p \leq 4294967295$
  shows *prime-field-gen* (*finite-field-ops64* (*uint64-of-int p*)) *mod-ring-rel64 p*
proof −
  let *?pp* = *uint64-of-int p*
  have *ppp*: *p* = *int-of-uint64 ?pp*
    by (*subst int-of-uint64-inv*, *insert small p2*, *auto*)
  note ∗ = *ppp small*
  interpret *field-ops finite-field-ops64 ?pp mod-ring-rel64*
    by (*rule finite-field-ops64*, *insert* ∗)
  interpret *int*: *prime-field-gen finite-field-ops-int p mod-ring-rel*
    by (*rule prime-field-finite-field-ops-int*)
  show *?thesis*
  proof (*unfold-locales*, *rule p*, *auto simp*: *finite-field-ops64-def*)
    fix *x*
    assume *x*: $0 \leq x$ $x < p$
    from *int.of-int*[*OF this*] **have** *mod-ring-rel x* (*of-int x*) **by** (*simp add*: *finite-field-ops-int-def*)
    thus *mod-ring-rel64* (*uint64-of-int x*) (*of-int x*) **unfolding** *mod-ring-rel64-def*[*OF* ∗]
      by (*intro exI*[*of - x*], *auto simp*: *urel64-def*[*OF* ∗], *subst int-of-uint64-inv*, *insert* ∗ *x*, *auto*)
  next
    fix *y z*
    assume *mod-ring-rel64 y z*
    from *this*[*unfolded mod-ring-rel64-def*[*OF* ∗]] **obtain** *x* **where** *yx*: *urel64 y x* **and** *xz*: *mod-ring-rel x z* **by** *auto*
    from *int.to-int*[*OF xz*] **have** *zx*: *to-int-mod-ring z* = *x* **by** (*simp add*: *finite-field-ops-int-def*)
    show *int-of-uint64 y* = *to-int-mod-ring z* **unfolding** *zx* **using** *yx* **unfolding** *urel64-def*[*OF* ∗] **by** *simp*
  next
    fix *y*
    show $0 \leq$ *int-of-uint64 y* $\Longrightarrow$ *int-of-uint64 y* $< p \Longrightarrow$ *mod-ring-rel64 y* (*of-int* (*int-of-uint64 y*))
      unfolding *mod-ring-rel64-def*[*OF* ∗] *urel64-def*[*OF* ∗]
      by (*intro exI*[*of - int-of-uint64 y*], *auto simp*: *mod-ring-rel-of-int*)
  qed
qed
end

context *mod-ring-locale*
begin
lemma *mod-ring-finite-field-ops-int*: *mod-ring-gen* (*finite-field-ops-int p*) *mod-ring-rel p*
proof −
  interpret *ring-ops finite-field-ops-int p mod-ring-rel* **by** (*rule ring-finite-field-ops-int*)
  show *?thesis*

by (*unfold-locales*, *rule p*,
auto simp: *finite-field-ops-int-def p mod-ring-rel-def of-int-of-int-mod-ring*)
qed

lemma *mod-ring-finite-field-ops-integer*: *mod-ring-gen* (*finite-field-ops-integer* (*integer-of-int p*)) *mod-ring-rel-integer p*
proof −
interpret *ring-ops finite-field-ops-integer* (*integer-of-int p*) *mod-ring-rel-integer*
by (*rule ring-finite-field-ops-integer*, *simp*)
have *pp*: *p* = *int-of-integer* (*integer-of-int p*) **by** *auto*
interpret *int*: *mod-ring-gen finite-field-ops-int p mod-ring-rel*
by (*rule mod-ring-finite-field-ops-int*)
show *?thesis*
by (*unfold-locales*, *rule p*, *auto simp*: *finite-field-ops-integer-def*
mod-ring-rel-integer-def[*OF pp*] *urel-integer-def*[*OF pp*] *mod-ring-rel-of-int*
int.to-int[*symmetric*] *finite-field-ops-int-def*)
qed

lemma *mod-ring-finite-field-ops32*: **assumes** *small*: $p \leq 65535$
shows *mod-ring-gen* (*finite-field-ops32* (*uint32-of-int p*)) *mod-ring-rel32 p*
proof −
let *?pp* = *uint32-of-int p*
have *ppp*: *p* = *int-of-uint32 ?pp*
by (*subst int-of-uint32-inv*, *insert small p2*, *auto*)
note ∗ = *ppp small*
interpret *ring-ops finite-field-ops32 ?pp mod-ring-rel32*
by (*rule ring-finite-field-ops32*, *insert* ∗)
interpret *int*: *mod-ring-gen finite-field-ops-int p mod-ring-rel*
by (*rule mod-ring-finite-field-ops-int*)
show *?thesis*
proof (*unfold-locales*, *rule p*, *auto simp*: *finite-field-ops32-def*)
fix *x*
assume *x*: $0 \leq x$ $x < p$
from *int.of-int*[*OF this*] **have** *mod-ring-rel x* (*of-int x*) **by** (*simp add*: *finite-field-ops-int-def*)
thus *mod-ring-rel32* (*uint32-of-int x*) (*of-int x*) **unfolding** *mod-ring-rel32-def*[*OF* ∗]
by (*intro exI*[*of - x*], *auto simp*: *urel32-def*[*OF* ∗], *subst int-of-uint32-inv*, *insert* ∗ *x*, *auto*)
next
fix *y z*
assume *mod-ring-rel32 y z*
from *this*[*unfolded mod-ring-rel32-def*[*OF* ∗]] **obtain** *x* **where** *yx*: *urel32 y x* **and** *xz*: *mod-ring-rel x z* **by** *auto*
from *int.to-int*[*OF xz*] **have** *zx*: *to-int-mod-ring z* = *x* **by** (*simp add*: *finite-field-ops-int-def*)
show *int-of-uint32 y* = *to-int-mod-ring z* **unfolding** *zx* **using** *yx* **unfolding** *urel32-def*[*OF* ∗] **by** *simp*

next
fix *y*
show $0 \leq$ *int-of-uint32 y* $\Longrightarrow$ *int-of-uint32 y* $< p \Longrightarrow$ *mod-ring-rel32 y* (*of-int* (*int-of-uint32 y*))
unfolding *mod-ring-rel32-def*[*OF* ∗] *urel32-def*[*OF* ∗]
by (*intro exI*[*of* - *int-of-uint32 y*], *auto simp*: *mod-ring-rel-of-int*)
qed
qed

lemma *mod-ring-finite-field-ops64*: **assumes** *small*: $p \leq 4294967295$
shows *mod-ring-gen* (*finite-field-ops64* (*uint64-of-int p*)) *mod-ring-rel64 p*
proof −
let *?pp* = *uint64-of-int p*
have *ppp*: *p* = *int-of-uint64 ?pp*
by (*subst int-of-uint64-inv*, *insert small p2*, *auto*)
note ∗ = *ppp small*
interpret *ring-ops finite-field-ops64 ?pp mod-ring-rel64*
by (*rule ring-finite-field-ops64*, *insert* ∗)
interpret *int*: *mod-ring-gen finite-field-ops-int p mod-ring-rel*
by (*rule mod-ring-finite-field-ops-int*)
show *?thesis*
proof (*unfold-locales*, *rule p*, *auto simp*: *finite-field-ops64-def*)
fix *x*
assume *x*: $0 \leq x$ $x < p$
from *int.of-int*[*OF this*] **have** *mod-ring-rel x* (*of-int x*) **by** (*simp add*: *finite-field-ops-int-def*)
thus *mod-ring-rel64* (*uint64-of-int x*) (*of-int x*) **unfolding** *mod-ring-rel64-def*[*OF* ∗]
by (*intro exI*[*of* - *x*], *auto simp*: *urel64-def*[*OF* ∗], *subst int-of-uint64-inv*, *insert* ∗ *x*, *auto*)
next
fix *y z*
assume *mod-ring-rel64 y z*
from *this*[*unfolded mod-ring-rel64-def*[*OF* ∗]] **obtain** *x* **where** *yx*: *urel64 y x* **and** *xz*: *mod-ring-rel x z* **by** *auto*
from *int.to-int*[*OF xz*] **have** *zx*: *to-int-mod-ring z* = *x* **by** (*simp add*: *finite-field-ops-int-def*)
show *int-of-uint64 y* = *to-int-mod-ring z* **unfolding** *zx* **using** *yx* **unfolding** *urel64-def*[*OF* ∗] **by** *simp*
next
fix *y*
show $0 \leq$ *int-of-uint64 y* $\Longrightarrow$ *int-of-uint64 y* $< p \Longrightarrow$ *mod-ring-rel64 y* (*of-int* (*int-of-uint64 y*))
unfolding *mod-ring-rel64-def*[*OF* ∗] *urel64-def*[*OF* ∗]
by (*intro exI*[*of* - *int-of-uint64 y*], *auto simp*: *mod-ring-rel-of-int*)
qed
qed
end

end

5.6 Chinese Remainder Theorem for Polynomials

We prove the Chinese Remainder Theorem, and strengthen it by showing uniqueness

theory *Chinese-Remainder-Poly*
imports
  HOL−Number-Theory.Residues
  Polynomial-Factorization.Polynomial-Divisibility
  Polynomial-Interpolation.Missing-Polynomial
begin

lemma *cong-add-poly*:
  $[(a::'b::\{field\text{-}gcd\}\ poly) = b]\ (mod\ m) \Longrightarrow [c = d]\ (mod\ m) \Longrightarrow [a + c = b + d]\ (mod\ m)$
  by (*fact cong-add*)

lemma *cong-mult-poly*:
  $[(a::'b::\{field\text{-}gcd\}\ poly) = b]\ (mod\ m) \Longrightarrow [c = d]\ (mod\ m) \Longrightarrow [a * c = b * d]\ (mod\ m)$
  by (*fact cong-mult*)

lemma *cong-mult-self-poly*: $[(a::'b::\{field\text{-}gcd\}\ poly) * m = 0]\ (mod\ m)$
  by (*fact cong-mult-self-right*)

lemma *cong-scalar2-poly*: $[(a::'b::\{field\text{-}gcd\}\ poly)= b]\ (mod\ m) \Longrightarrow [k * a = k * b]\ (mod\ m)$
  by (*fact cong-scalar-left*)

lemma *cong-sum-poly*:
    $(\bigwedge x.\ x \in A \Longrightarrow [((f\ x)::'b::\{field\text{-}gcd\}\ poly) = g\ x]\ (mod\ m)) \Longrightarrow$
      $[(\sum x{\in}A.\ f\ x) = (\sum x{\in}A.\ g\ x)]\ (mod\ m)$
  by (*rule cong-sum*)

lemma *cong-iff-lin-poly*: $([(a::'b::\{field\text{-}gcd\}\ poly) = b]\ (mod\ m)) = (\exists k.\ b = a + m * k)$
  using *cong-diff-iff-cong-0* [*of b a m*] **by** (*auto simp add*: *cong-0-iff dvd-def algebra-simps dest*: *cong-sym*)

lemma *cong-solve-poly*: $(a::'b::\{field\text{-}gcd\}\ poly) \neq 0 \Longrightarrow \exists x.\ [a * x = gcd\ a\ n]\ (mod\ n)$
proof (*cases* $n = 0$)
  case *True*
  note *n0=True*
  show *?thesis*
  proof (*cases monic a*)
    case *True*
    have *n*: *normalize* $a = a$ **by** (*rule normalize-monic*[*OF True*])

show *?thesis*
by (*rule exI*[*of - 1*], *auto simp add*: *n0 n cong-def*)
next
case *False*
show *?thesis*
by (*auto simp add*: *True cong-def normalize-poly-old-def map-div-is-smult-inverse*)
(*metis mult.right-neutral mult-smult-right*)
qed
next
case *False*
note *n-not-0* = *False*
show *?thesis*
using *bezout-coefficients-fst-snd* [*of a n*, *symmetric*]
by (*auto simp add*: *cong-iff-lin-poly mult.commute* [*of a*] *mult.commute* [*of n*])
qed

lemma *cong-solve-coprime-poly*:
assumes *coprime-an*:*coprime* (*a*::′*b*::{*field-gcd*} *poly*) *n*
shows $\exists$ *x*. [*a* $*$ *x* = *1*] (*mod n*)
proof (*cases a* = *0*)
case *True*
show *?thesis* **unfolding** *cong-def*
using *True coprime-an* **by** *auto*
next
case *False*
show *?thesis*
using *coprime-an cong-solve-poly*[*OF False*, *of n*]
unfolding *cong-def*
by *presburger*
qed

lemma *cong-dvd-modulus-poly*:
[*x* = *y*] (*mod m*) $\Longrightarrow$ *n dvd m* $\Longrightarrow$ [*x* = *y*] (*mod n*) **for** *x y* :: ′*b*::{*field-gcd*} *poly*
by (*auto simp add*: *cong-iff-lin-poly elim*!: *dvdE*)

lemma *chinese-remainder-aux-poly*:
fixes *A* :: ′*a set*
and *m* :: ′*a* $\Rightarrow$ ′*b*::{*field-gcd*} *poly*
assumes *fin*: *finite A*
and *cop*: $\forall$ *i* $\in$ *A*. ($\forall$ *j* $\in$ *A*. *i* $\neq$ *j* $\longrightarrow$ *coprime* (*m i*) (*m j*))
shows $\exists$ *b*. ($\forall$ *i* $\in$ *A*. [*b i* = *1*] (*mod m i*) $\wedge$ [*b i* = *0*] (*mod* ($\prod$ *j* $\in$ *A* − {*i*}. *m j*)))
proof (*rule finite-set-choice*, *rule fin*, *rule ballI*)
fix *i*
assume *i* : *A*
with *cop* **have** *coprime* ($\prod$ *j* $\in$ *A* − {*i*}. *m j*) (*m i*)
by (*auto intro*: *prod-coprime-left*)
then have $\exists$ *x*. [($\prod$ *j* $\in$ *A* − {*i*}. *m j*) $*$ *x* = *1*] (*mod m i*)

by (*elim cong-solve-coprime-poly*)
then obtain x **where** $[(\prod j \in A - \{i\}.\ m\ j) * x = 1]$ $(mod\ m\ i)$
by *auto*
moreover have $[(\prod j \in A - \{i\}.\ m\ j) * x = 0]$
$(mod\ (\prod j \in A - \{i\}.\ m\ j))$
by (*subst mult.commute*, *rule cong-mult-self-poly*)
ultimately show $\exists\, a.\ [a = 1]\ (mod\ m\ i) \wedge [a = 0]$
$(mod\ prod\ m\ (A - \{i\}))$
by *blast*
qed

lemma *chinese-remainder-poly*:
fixes A :: *'a set*
and m :: $'a \Rightarrow 'b$::{*field-gcd*} *poly*
and u :: $'a \Rightarrow 'b$ *poly*
assumes *fin*: *finite A*
and *cop*: $\forall\, i{\in}A.\ (\forall\, j{\in}A.\ i \neq j \longrightarrow coprime\ (m\ i)\ (m\ j))$
shows $\exists\, x.\ (\forall\, i{\in}A.\ [x = u\ i]\ (mod\ m\ i))$
proof −
from *chinese-remainder-aux-poly* [*OF fin cop*] **obtain** b **where**
bprop: $\forall\, i{\in}A.\ [b\ i = 1]\ (mod\ m\ i) \wedge$
$[b\ i = 0]\ (mod\ (\prod j \in A - \{i\}.\ m\ j))$
by *blast*
let $?x = \sum i{\in}A.\ (u\ i) * (b\ i)$
show *?thesis*
proof (*rule exI*, *clarify*)
fix i
assume a: $i : A$
show $[?x = u\ i]\ (mod\ m\ i)$
proof −
from *fin a* **have** $?x = (\sum j \in \{i\}.\ u\ j * b\ j) +$
$(\sum j \in A - \{i\}.\ u\ j * b\ j)$
by (*subst sum.union-disjoint* [*symmetric*], *auto intro*: *sum.cong*)
then have $[?x = u\ i * b\ i + (\sum j \in A - \{i\}.\ u\ j * b\ j)]\ (mod\ m\ i)$
unfolding *cong-def*
by *auto*
also have $[u\ i * b\ i + (\sum j \in A - \{i\}.\ u\ j * b\ j) =$
$u\ i * 1 + (\sum j \in A - \{i\}.\ u\ j * 0)]\ (mod\ m\ i)$
apply (*rule cong-add-poly*)
apply (*rule cong-scalar2-poly*)
using *bprop a* **apply** *blast*
apply (*rule cong-sum*)
apply (*rule cong-scalar2-poly*)
using *bprop* **apply** *auto*
apply (*rule cong-dvd-modulus-poly*)
apply (*drule* (*1*) *bspec*)
apply (*erule conjE*)

apply *assumption*
apply *rule*
using *fin a* **apply** *auto*
done
thus *?thesis*
by (*metis* (*no-types*, *lifting*) *a add.right-neutral fin mult-cancel-left1 mult-cancel-right1*

sum.not-neutral-contains-not-neutral sum.remove)
qed
qed
qed

lemma *cong-trans-poly*:
$[(a::'b::\{field\text{-}gcd\}\ poly) = b]\ (mod\ m) \Longrightarrow [b = c]\ (mod\ m) \Longrightarrow [a = c]\ (mod\ m)$
by (*fact cong-trans*)

lemma *cong-mod-poly*: $(n::'b::\{field\text{-}gcd\}\ poly)\ {}^{\sim}\!= 0 \Longrightarrow [a\ mod\ n = a]\ (mod\ n)$
by *auto*

lemma *cong-sym-poly*: $[(a::'b::\{field\text{-}gcd\}\ poly) = b]\ (mod\ m) \Longrightarrow [b = a]\ (mod\ m)$
by (*fact cong-sym*)

lemma *cong-1-poly*: $[(a::'b::\{field\text{-}gcd\}\ poly) = b]\ (mod\ 1)$
by (*fact cong-1*)

lemma *coprime-cong-mult-poly*:
assumes $[(a::'b::\{field\text{-}gcd\}\ poly) = b]\ (mod\ m)$ **and** $[a = b]\ (mod\ n)$ **and** *coprime m n*
shows $[a = b]\ (mod\ m * n)$
using *divides-mult assms*
by (*metis* (*no-types*, *opaque-lifting*) *cong-dvd-modulus-poly cong-iff-lin-poly dvd-mult2 dvd-refl minus-add-cancel mult.right-neutral*)

lemma *coprime-cong-prod-poly*:
$(\forall i\in A.\ (\forall j\in A.\ i \neq j \longrightarrow coprime\ (m\ i)\ (m\ j))) \Longrightarrow$
$(\forall i\in A.\ [(x::'b::\{field\text{-}gcd\}\ poly) = y]\ (mod\ m\ i)) \Longrightarrow$
$[x = y]\ (mod\ (\prod i\in A.\ m\ i))$
apply (*induct A rule*: *infinite-finite-induct*)
apply *auto*
apply (*metis coprime-cong-mult-poly prod-coprime-right*)
done

lemma *cong-less-modulus-unique-poly*:
$[(x::'b::\{field\text{-}gcd\}\ poly) = y]\ (mod\ m) \Longrightarrow degree\ x < degree\ m \Longrightarrow degree\ y < degree\ m \Longrightarrow x = y$

by (*simp add*: *cong-def mod-poly-less*)

lemma *chinese-remainder-unique-poly*:
fixes A :: $'a$ *set*
and m :: $'a \Rightarrow 'b$::{*field-gcd*} *poly*
and u :: $'a \Rightarrow 'b$ *poly*
assumes *nz*: $\forall i{\in}A.\ (m\ i) \neq 0$
and *cop*: $\forall i{\in}A.\ (\forall j{\in}A.\ i \neq j \longrightarrow \mathit{coprime}\ (m\ i)\ (m\ j))$

and *not-constant*: $0 < \mathit{degree}\ (\mathit{prod}\ m\ A)$
shows $\exists!x.\ \mathit{degree}\ x < (\sum i{\in}A.\ \mathit{degree}\ (m\ i)) \wedge (\forall i{\in}A.\ [x = u\ i]\ (\mathit{mod}\ m\ i))$
proof −
from *not-constant* **have** *fin*: *finite A*
by (*metis degree-1 gr-implies-not0 prod.infinite*)
from *chinese-remainder-poly* [*OF fin cop*]
obtain *y* **where** *one*: $(\forall i{\in}A.\ [y = u\ i]\ (\mathit{mod}\ m\ i))$
by *blast*
let $?x = y\ \mathit{mod}\ (\prod i{\in}A.\ m\ i)$
have *degree-prod-sum*: $\mathit{degree}\ (\mathit{prod}\ m\ A) = (\sum i{\in}A.\ \mathit{degree}\ (m\ i))$
by (*rule degree-prod-eq-sum-degree*[*OF nz*])
from *fin nz* **have** *prodnz*: $(\prod i{\in}A.\ (m\ i)) \neq 0$
by *auto*

have *less*: $\mathit{degree}\ ?x < (\sum i{\in}A.\ \mathit{degree}\ (m\ i))$
unfolding *degree-prod-sum*[*symmetric*]
using *degree-mod-less*[*OF prodnz*, *of y*]
using *not-constant*
by *auto*
have *cong*: $\forall i{\in}A.\ [?x = u\ i]\ (\mathit{mod}\ m\ i)$
apply *auto*
apply (*rule cong-trans-poly*)
prefer *2*
using *one* **apply** *auto*
apply (*rule cong-dvd-modulus-poly*)
apply (*rule cong-mod-poly*)
using *prodnz* **apply** *auto*
apply *rule*
apply (*rule fin*)
apply *assumption*
done
have *unique*: $\forall z.\ \mathit{degree}\ z < (\sum i{\in}A.\ \mathit{degree}\ (m\ i)) \wedge$
$(\forall i{\in}A.\ [z = u\ i]\ (\mathit{mod}\ m\ i)) \longrightarrow z = ?x$
proof (*clarify*)
fix z::$'b$ *poly*
assume *zless*: $\mathit{degree}\ z < (\sum i{\in}A.\ \mathit{degree}\ (m\ i))$
assume *zcong*: $(\forall i{\in}A.\ [z = u\ i]\ (\mathit{mod}\ m\ i))$
have *deg1*: $\mathit{degree}\ z < \mathit{degree}\ (\mathit{prod}\ m\ A)$
using *degree-prod-sum zless* **by** *simp*

have *deg2*: *degree ?x* < *degree* (*prod m A*)
  by (*metis deg1 degree-0 degree-mod-less gr0I gr-implies-not0*)
have ∀ *i*∈*A*. [*?x* = *z*] (*mod m i*)
  apply *clarify*
  apply (*rule cong-trans-poly*)
  using *cong* **apply** (*erule bspec*)
  apply (*rule cong-sym-poly*)
  using *zcong* **by** *auto*
with *fin cop* **have** [*?x* = *z*] (*mod* (∏ *i*∈*A*. *m i*))
  by (*intro coprime-cong-prod-poly*) *auto*
with *zless* **show** *z* = *?x*
  apply (*intro cong-less-modulus-unique-poly*)
  apply (*erule cong-sym-poly*)
  apply (*auto simp add*: *deg1 deg2*)
  done
qed
from *less cong unique* **show** *?thesis* **by** *blast*
qed

end

6 The Berlekamp Algorithm

theory *Berlekamp-Type-Based*
imports
Jordan-Normal-Form.Matrix-Kernel
Jordan-Normal-Form.Gauss-Jordan-Elimination
Jordan-Normal-Form.Missing-VectorSpace
Polynomial-Factorization.Square-Free-Factorization
Polynomial-Factorization.Missing-Multiset
Finite-Field
Chinese-Remainder-Poly
Poly-Mod-Finite-Field
HOL−Computational-Algebra.Field-as-Ring
begin

hide-const (**open**) *up-ring.coeff up-ring.monom Modules.module subspace*
Modules.module-hom

6.1 Auxiliary lemmas

context
fixes *g* :: *'b* ⇒ *'a* :: *comm-monoid-mult*
begin
lemma *prod-list-map-filter*: *prod-list* (*map g* (*filter f xs*)) ∗ *prod-list* (*map g* (*filter* (λ *x*. ¬ *f x*) *xs*))
= *prod-list* (*map g xs*)
by (*induct xs*, *auto simp*: *ac-simps*)

lemma *prod-list-map-partition*:
  assumes *List.partition f xs* = (*ys*, *zs*)
  shows *prod-list* (*map g xs*) = *prod-list* (*map g ys*) ∗ *prod-list* (*map g zs*)
  using *assms* **by** (*subst prod-list-map-filter*[*symmetric*, *of - f*], *auto simp*: *o-def*)
end

lemma *coprime-id-is-unit*:
  fixes *a*::′*b*::*semiring-gcd*
  shows *coprime a a* ⟷ *is-unit a*
  using *dvd-unit-imp-unit* **by** *auto*

lemma *dim-vec-of-list*[*simp*]: *dim-vec* (*vec-of-list x*) = *length x*
  by (*transfer*, *auto*)

lemma *length-list-of-vec*[*simp*]: *length* (*list-of-vec A*) = *dim-vec A*
  by (*transfer*′, *auto*)

lemma *list-of-vec-vec-of-list*[*simp*]: *list-of-vec* (*vec-of-list a*) = *a*
proof −
  {
  fix *aa* :: ′*a list*
   have *map* (λ*n*. *if n* < *length aa then aa* ! *n else undef-vec* (*n* − *length aa*)) [*0*..<*length aa*]
    = *map* ((!) *aa*) [*0*..<*length aa*]
    by *simp*
   hence *map* (λ*n*. *if n* < *length aa then aa* ! *n else undef-vec* (*n* − *length aa*)) [*0*..<*length aa*] = *aa*
    by (*simp add*: *map-nth*)
  }
  thus *?thesis* **by** (*transfer*, *simp add*: *mk-vec-def*)
qed

context
assumes *SORT-CONSTRAINT*(′*a*::*finite*)
begin

lemma *inj-Poly-list-of-vec*′: *inj-on* (*Poly* ∘ *list-of-vec*) {*v*. *dim-vec v* = *n*}
proof (*rule comp-inj-on*)
  show *inj-on list-of-vec* {*v*. *dim-vec v* = *n*}
    by (*auto simp add*: *inj-on-def*, *transfer*, *auto simp add*: *mk-vec-def*)
  show *inj-on Poly* (*list-of-vec* ‘ {*v*. *dim-vec v* = *n*})
  proof (*auto simp add*: *inj-on-def*)
    fix *x y*::′*c vec* **assume** *n* = *dim-vec x* **and** *dim-xy*: *dim-vec y* = *dim-vec x*
    and *Poly-eq*: *Poly* (*list-of-vec x*) = *Poly* (*list-of-vec y*)
    note [*simp del*] = *nth-list-of-vec*
    show *list-of-vec x* = *list-of-vec y*
    proof (*rule nth-equalityI*, *auto simp*: *dim-xy*)
      have *length-eq*: *length* (*list-of-vec x*) = *length* (*list-of-vec y*)

```
        using dim-xy by (transfer, auto)
      fix i assume i < dim-vec x
       thus list-of-vec x ! i = list-of-vec y ! i using Poly-eq unfolding poly-eq-iff
coeff-Poly-eq
        using dim-xy unfolding nth-default-def by (auto, presburger)
     qed
  qed
qed

corollary inj-Poly-list-of-vec: inj-on (Poly ∘ list-of-vec) (carrier-vec n)
  using inj-Poly-list-of-vec' unfolding carrier-vec-def .

lemma list-of-vec-rw-map: list-of-vec m = map (λn. m $ n) [0..<dim-vec m]
    by (transfer, auto simp add: mk-vec-def)

lemma degree-Poly':
assumes xs: xs ≠ []
shows degree (Poly xs) < length xs
using xs
by (induct xs, auto intro: Poly.simps(1))

lemma vec-of-list-list-of-vec[simp]: vec-of-list (list-of-vec a) = a
by (transfer, auto simp add: mk-vec-def)

lemma row-mat-of-rows-list:
assumes b: b < length A
and nc: ∀ i. i < length A ⟶ length (A ! i) = nc
shows (row (mat-of-rows-list nc A) b) = vec-of-list (A ! b)
proof (auto simp add: vec-eq-iff)
  show dim-col (mat-of-rows-list nc A) = length (A ! b)
    unfolding mat-of-rows-list-def using b nc by auto
  fix i assume i: i < length (A ! b)
  show row (mat-of-rows-list nc A) b $ i = vec-of-list (A ! b) $ i
    using i b nc
    unfolding mat-of-rows-list-def row-def
    by (transfer, auto simp add: mk-vec-def mk-mat-def)
qed

lemma degree-Poly-list-of-vec:
assumes n: x ∈ carrier-vec n
and n0: n > 0
shows degree (Poly (list-of-vec x)) < n
proof −
  have x-dim: dim-vec x = n using n by auto
  have l: (list-of-vec x) ≠ []
    by (auto simp add: list-of-vec-rw-map vec-of-dim-0[symmetric] n0 n x-dim)
 have degree (Poly (list-of-vec x)) < length (list-of-vec x) by (rule degree-Poly'[OF
l])
  also have ... = n using x-dim by auto
```

```
  finally show ?thesis .
qed

lemma list-of-vec-nth:
  assumes i: i < dim-vec x
  shows list-of-vec x ! i = x $ i
  using i
  by (transfer, auto simp add: mk-vec-def)

lemma coeff-Poly-list-of-vec-nth':
 assumes i: i < dim-vec x
 shows coeff (Poly (list-of-vec x)) i = x $ i
 using i
 by (auto simp add: list-of-vec-nth nth-default-def)

lemma list-of-vec-row-nth:
assumes x: x < dim-col A
shows list-of-vec (row A i) ! x = A $$ (i, x)
using x unfolding row-def by (transfer', auto simp add: mk-vec-def)

lemma coeff-Poly-list-of-vec-nth:
assumes x: x < dim-col A
shows coeff (Poly (list-of-vec (row A i))) x = A $$ (i, x)
proof -
  have coeff (Poly (list-of-vec (row A i))) x  = nth-default 0 (list-of-vec (row A
i)) x
    unfolding coeff-Poly-eq  by simp
  also have ... = A $$ (i, x) using x list-of-vec-row-nth
    unfolding nth-default-def by (auto simp del: nth-list-of-vec)
  finally show ?thesis .
qed

lemma inj-on-list-of-vec: inj-on list-of-vec (carrier-vec n)
 unfolding inj-on-def unfolding list-of-vec-rw-map by auto

lemma vec-of-list-carrier[simp]: vec-of-list x ∈ carrier-vec (length x)
  unfolding carrier-vec-def by simp

lemma card-carrier-vec: card (carrier-vec n:: 'b::finite vec set) = CARD('b) ^ n
proof -
  let ?A = UNIV::'b set
  let ?B = {xs. set xs ⊆ ?A ∧ length xs = n}
  let ?C = (carrier-vec n:: 'b::finite vec set)
  have card ?C = card ?B
  proof -
    have bij-betw (list-of-vec) ?C ?B
    proof (unfold bij-betw-def, auto)
      show inj-on list-of-vec (carrier-vec n) by (rule inj-on-list-of-vec)
      fix x::'b list
```

    assume *n*: *n* = *length x*
    thus *x* ∈ *list-of-vec* ‘ *carrier-vec* (*length x*)
      unfolding *image-def*
      by *auto* (*rule bexI*[*of* - *vec-of-list x*], *auto*)
  qed
  thus *?thesis* **using** *bij-betw-same-card* **by** *blast*
qed
also have ... = *card ?A* ^ *n*
  by (*rule card-lists-length-eq*, *simp*)
finally show *?thesis* **.**
qed

lemma *finite-carrier-vec*[*simp*]: *finite* (*carrier-vec n*:: *'b*::*finite vec set*)
  by (*rule card-ge-0-finite*, *unfold card-carrier-vec*, *auto*)

lemma *row-echelon-form-dim0-row*:
assumes *A* ∈ *carrier-mat 0 n*
shows *row-echelon-form A*
using *assms*
unfolding *row-echelon-form-def pivot-fun-def Let-def* **by** *auto*

lemma *row-echelon-form-dim0-col*:
assumes *A* ∈ *carrier-mat n 0*
shows *row-echelon-form A*
using *assms*
unfolding *row-echelon-form-def pivot-fun-def Let-def* **by** *auto*

lemma *row-echelon-form-one-dim0*[*simp*]: *row-echelon-form* (1_m *0*)
  unfolding *row-echelon-form-def pivot-fun-def Let-def* **by** *auto*

lemma *Poly-list-of-vec-0*[*simp*]: *Poly* (*list-of-vec* (0_v *0*)) = [:*0*:]
  by (*simp add*: *poly-eq-iff nth-default-def*)

lemma *monic-normalize*:
assumes (*p* :: *'b* :: {*field*,*euclidean-ring-gcd*} *poly*) ≠ *0* **shows** *monic* (*normalize p*)
by (*simp add*: *assms normalize-poly-old-def*)

lemma *exists-factorization-prod-list*:
fixes *P*::*'b*::*field poly list*
assumes *degree* (*prod-list P*) > *0*
  and ⋀ *u*. *u* ∈ *set P* ⟹ *degree u* > *0* ∧ *monic u*
  and *square-free* (*prod-list P*)
shows ∃ *Q*. *prod-list Q* = *prod-list P* ∧ *length P* ≤ *length Q*
      ∧ (∀ *u*. *u* ∈ *set Q* ⟶ *irreducible u* ∧ *monic u*)
using *assms*

proof (*induct P*)
  case *Nil*
  thus *?case* **by** *auto*
next
  case (*Cons x P*)
  have *sf-P*: *square-free* (*prod-list P*)
   by (*metis Cons.prems*(*3*) *dvd-triv-left prod-list.Cons mult.commute square-free-factor*)
  have *deg-x*: *degree x* > *0* **using** *Cons.prems* **by** *auto*
  have *distinct-P*: *distinct P*
   by (*meson Cons.prems*(*2*) *Cons.prems*(*3*) *distinct.simps*(*2*) *square-free-prod-list-distinct*)
  have $\exists$ *A*. *finite A* $\wedge$ *x* = $\prod$ *A* $\wedge$ *A* $\subseteq$ {*q*. *irreducible q* $\wedge$ *monic q*}
    proof (*rule monic-square-free-irreducible-factorization*)
      show *monic x* **by** (*simp add*: *Cons.prems*(*2*))
      show *square-free x*
        by (*metis Cons.prems*(*3*) *dvd-triv-left prod-list.Cons square-free-factor*)
    qed
    from *this* **obtain** *A* **where** *fin-A*: *finite A*
    and *xA*: *x* = $\prod$ *A*
    and *A*: *A* $\subseteq$ {*q*. *irreducible*$_d$ *q* $\wedge$ *monic q*}
    by *auto*
    obtain *A'* **where** *s*: *set A'* = *A* **and** *length-A'*: *length A'* = *card A*
      using ‹*finite A*› *distinct-card finite-distinct-list* **by** *force*
  have *A-not-empty*: *A* $\neq$ {} **using** *xA deg-x* **by** *auto*
  have *x-prod-list-A'*: *x* = *prod-list A'*
  proof −
    have *x* = $\prod$ *A* **using** *xA* **by** *simp*
    also have ... = *prod id A* **by** *simp*
    also have ... = *prod id* (*set A'*) **unfolding** *s* **by** *simp*
    also have ... = *prod-list* (*map id A'*)
      by (*rule prod.distinct-set-conv-list*, *simp add*: *card-distinct length-A' s*)
    also have ... = *prod-list A'* **by** *auto*
    finally show *?thesis* **.**
  qed
  show *?case*
  proof (*cases P* = [])
    case *True*
    show *?thesis*
    proof (*rule exI*[*of* - *A'*], *auto simp add*: *True*)
      show *prod-list A'* = *x* **using** *x-prod-list-A'* **by** *simp*
      show *Suc 0* $\leq$ *length A'* **using** *A-not-empty* **using** *s length-A'*
        by (*simp add*: *Suc-leI card-gt-0-iff fin-A*)
      show $\bigwedge$*u*. *u* $\in$ *set A'* $\Longrightarrow$ *irreducible u* **using** *s A* **by** *auto*
      show $\bigwedge$*u*. *u* $\in$ *set A'* $\Longrightarrow$ *monic u* **using** *s A* **by** *auto*
    qed
 next
  case *False*
  have *hyp*: $\exists$ *Q*. *prod-list Q* = *prod-list P*
    $\wedge$ *length P* $\leq$ *length Q* $\wedge$ ($\forall$ *u*. *u* $\in$ *set Q* $\longrightarrow$ *irreducible u* $\wedge$ *monic u*)
  proof (*rule Cons.hyps*[*OF* - - *sf-P*])

have *set-P*: *set P* $\neq$ {} **using** *False* **by** *auto*
have *prod-list P* = *prod-list* (*map id P*) **by** *simp*
also have ... = *prod id* (*set P*)
using *prod.distinct-set-conv-list*[*OF distinct-P*, *of id*] **by** *simp*
also have ... = $\prod$(*set P*) **by** *simp*
finally have *prod-list P* = $\prod$(*set P*) **.**
hence *degree* (*prod-list P*) = *degree* ($\prod$(*set P*)) **by** *simp*
also have ... = *degree* (*prod id* (*set P*)) **by** *simp*
also have ... = ($\sum$ *i*$\in$(*set P*). *degree* (*id i*))
proof (*rule degree-prod-eq-sum-degree*)
show $\forall$ *i*$\in$*set P*. *id i* $\neq$ *0* **using** *Cons.prems*(*2*) **by** *force*
qed
also have ... > *0*
by (*metis Cons.prems*(*2*) *List.finite-set set-P gr0I id-apply insert-iff list.set*(*2*) *sum-pos*)
finally show *degree* (*prod-list P*) > *0* **by** *simp*
show $\bigwedge$*u*. *u* $\in$ *set P* $\Longrightarrow$ *degree u* > *0* $\wedge$ *monic u* **using** *Cons.prems* **by** *auto*
qed
from *this* **obtain** *Q* **where** *QP*: *prod-list Q* = *prod-list P* **and** *length-PQ*: *length P* $\leq$ *length Q*
and *monic-irr-Q*: ($\forall$ *u*. *u* $\in$ *set Q* $\longrightarrow$ *irreducible u* $\wedge$ *monic u*) **by** *blast*
show *?thesis*
proof (*rule exI*[*of - A′* @ *Q*], *auto simp add*: *monic-irr-Q*)
show *prod-list A′* ∗ *prod-list Q* = *x* ∗ *prod-list P* **unfolding** *QP x-prod-list-A′* **by** *auto*
have *length A′* $\neq$ *0* **using** *A-not-empty* **using** *s length-A′* **by** *auto*
thus *Suc* (*length P*) $\leq$ *length A′* + *length Q* **using** *QP length-PQ* **by** *linarith*
show $\bigwedge$*u*. *u* $\in$ *set A′* $\Longrightarrow$ *irreducible u* **using** *s A* **by** *auto*
show $\bigwedge$*u*. *u* $\in$ *set A′* $\Longrightarrow$ *monic u* **using** *s A* **by** *auto*
qed
qed
qed

lemma *normalize-eq-imp-smult*:
fixes *p* :: *′b* :: {*euclidean-ring-gcd*} *poly*
assumes *n*: *normalize p* = *normalize q*
shows $\exists$ *c*. *c* $\neq$ *0* $\wedge$ *q* = *smult c p*
proof(*cases p* = *0*)
case *True* **with** *n* **show** *?thesis* **by** (*auto intro*:*exI*[*of - 1*])
next
case *p0*: *False*
have *degree-eq*: *degree p* = *degree q* **using** *n degree-normalize* **by** *metis*
hence *q0*: *q* $\neq$ *0* **using** *p0 n* **by** *auto*
have *p-dvd-q*: *p dvd q* **using** *n* **by** (*simp add*: *associatedD1*)
from *p-dvd-q* **obtain** *k* **where** *q*: *q* = *k* ∗ *p* **unfolding** *dvd-def* **by** (*auto simp*: *ac-simps*)
with *q0* **have** *k* $\neq$ *0* **by** *auto*
then have *degree k* = *0*
using *degree-eq degree-mult-eq p0 q* **by** *fastforce*

then obtain *c* **where** *k*: *k* = [: *c* :] **by** (*metis degree-0-id*)
with ‹*k* ≠ *0*› **have** *c* ≠ *0* **by** *auto*
have *q* = *smult c p* **unfolding** *q k* **by** *simp*
with ‹*c* ≠ *0*› **show** *?thesis* **by** *auto*
qed

lemma *prod-list-normalize*:
fixes *P* :: *'b* :: {*idom-divide*,*normalization-semidom-multiplicative*} *poly list*
shows *normalize* (*prod-list P*) = *prod-list* (*map normalize P*)
proof (*induct P*)
case *Nil*
show *?case* **by** *auto*
next
case (*Cons p P*)
have *normalize* (*prod-list* (*p* # *P*)) = *normalize p* ∗ *normalize* (*prod-list P*)
using *normalize-mult* **by** *auto*
also have ... = *normalize p* ∗ *prod-list* (*map normalize P*) **using** *Cons.hyps* **by** *auto*
also have ... = *prod-list* (*normalize p* # (*map normalize P*)) **by** *auto*
also have ... = *prod-list* (*map normalize* (*p* # *P*)) **by** *auto*
finally show *?case* **.**
qed

lemma *prod-list-dvd-prod-list-subset*:
fixes *A*::*'b*::*comm-monoid-mult list*
assumes *dA*: *distinct A*
and *dB*: *distinct B*
and *s*: *set A* ⊆ *set B*
shows *prod-list A dvd prod-list B*
proof −
have *prod-list A* = *prod-list* (*map id A*) **by** *auto*
also have ... = *prod id* (*set A*)
by (*rule prod.distinct-set-conv-list*[*symmetric*, *OF dA*])
also have ... *dvd prod id* (*set B*)
by (*rule prod-dvd-prod-subset*[*OF - s*], *auto*)
also have ... = *prod-list* (*map id B*)
by (*rule prod.distinct-set-conv-list*[*OF dB*])
also have ... = *prod-list B* **by** *simp*
finally show *?thesis* **.**
qed

end

lemma *gcd-monic-constant*:
gcd f g ∈ {*1*, *f*} **if** *monic f* **and** *degree g* = *0*
for *f g* :: *'a* :: {*field-gcd*} *poly*
proof (*cases g* = *0*)
case *True*

moreover from ‹*monic f*› **have** *normalize f* = *f*
by (*rule normalize-monic*)
ultimately show *?thesis*
by *simp*
next
case *False*
with ‹*degree g* = *0*› **have** *is-unit g*
by *simp*
then have *Rings.coprime f g*
by (*rule is-unit-right-imp-coprime*)
then show *?thesis*
by *simp*
qed

lemma *distinct-find-base-vectors*:
fixes *A*::′*a*::*field mat*
assumes *ref*: *row-echelon-form A*
and *A*: *A* ∈ *carrier-mat nr nc*
shows *distinct* (*find-base-vectors A*)
proof −
note *non-pivot-base* = *non-pivot-base*[*OF ref A*]
let *?pp* = *set* (*pivot-positions A*)
from *A* **have** *dim*: *dim-row A* = *nr dim-col A* = *nc* **by** *auto*
{
fix *j j*′
assume *j*: *j* < *nc j* ∉ *snd* ‘ *?pp* **and** *j*′: *j*′ < *nc j*′ ∉ *snd* ‘ *?pp* **and** *neq*: *j*′ ≠ *j*
from *non-pivot-base*(*2*)[*OF j*] *non-pivot-base*(*4*)[*OF j*′ *j neq*]
have *non-pivot-base A* (*pivot-positions A*) *j* ≠ *non-pivot-base A* (*pivot-positions A*) *j*′ **by** *auto*
}
hence *inj*: *inj-on* (*non-pivot-base A* (*pivot-positions A*))
(*set* [*j*←[*0*..<*nc*] . *j* ∉ *snd* ‘ *?pp*]) **unfolding** *inj-on-def* **by** *auto*
thus *?thesis* **unfolding** *find-base-vectors-def Let-def* **unfolding** *distinct-map dim* **by** *auto*
qed

lemma *length-find-base-vectors*:
fixes *A*::′*a*::*field mat*
assumes *ref*: *row-echelon-form A*
and *A*: *A* ∈ *carrier-mat nr nc*
shows *length* (*find-base-vectors A*) = *card* (*set* (*find-base-vectors A*))
using *distinct-card*[*OF distinct-find-base-vectors*[*OF ref A*]] **by** *auto*

6.2 Previous Results

definition *power-poly-f-mod* :: ′*a*::*field poly* ⇒ ′*a poly* ⇒ *nat* ⇒ ′*a poly* **where**
power-poly-f-mod modulus = (λ*a n. a* ^ *n mod modulus*)

lemma *power-poly-f-mod-binary*: *power-poly-f-mod m a n* = (*if n* = *0 then 1 mod*

m
  else let (*d*, *r*) = *Divides.divmod-nat n 2*;
    rec = *power-poly-f-mod m* ((*a* ∗ *a*) *mod m*) *d in*
  if r = *0 then rec else* (*rec* ∗ *a*) *mod m*)
for *m a* :: *'a* :: {*field-gcd*} *poly*
proof −
  note *d* = *power-poly-f-mod-def*
  show *?thesis*
  proof (*cases n* = *0*)
    case *True*
    thus *?thesis* **unfolding** *d* **by** *simp*
  next
    case *False*
    obtain *q r* **where** *div*: *Divides.divmod-nat n 2* = (*q*,*r*) **by** *force*
    hence *n*: *n* = *2* ∗ *q* + *r* **and** *r*: *r* = *0* ∨ *r* = *1* **unfolding** *divmod-nat-def* **by**
auto
    have *id*: *a* ^ (*2* ∗ *q*) = (*a* ∗ *a*) ^ *q*
      by (*simp add*: *power-mult-distrib semiring-normalization-rules*)
    show *?thesis*
    proof (*cases r* = *0*)
      case *True*
      show *?thesis*
        using *power-mod* [*of a* ∗ *a m q*]
        by (*auto simp add*: *divmod-nat-def Let-def True n d div id*)
    next
      case *False*
      with *r* **have** *r*: *r* = *1* **by** *simp*
      show *?thesis*
        by (*auto simp add*: *d r div Let-def mod-simps*)
        (*simp add*: *n r mod-simps ac-simps power-mult-distrib power-mult power2-eq-square*)
    qed
  qed
qed

fun *power-polys* **where**
  power-polys mul-p u curr-p (*Suc i*) = *curr-p* #
    power-polys mul-p u ((*curr-p* ∗ *mul-p*) *mod u*) *i*
| *power-polys mul-p u curr-p 0* = []

context
assumes *SORT-CONSTRAINT*(*'a*::*prime-card*)
begin

lemma *fermat-theorem-mod-ring* [*simp*]:
  fixes *a*::*'a mod-ring*
  shows *a* ^ *CARD*(*'a*) = *a*
proof (*cases a* = *0*)
  case *True*

then show *?thesis* **by** *auto*
next
case *False*
then show *?thesis*
proof *transfer*
fix *a*
assume $a \in \{0..<int\ CARD('a)\}$ **and** $a \neq 0$
then have *a*: $1 \leq a\ a < int\ CARD('a)$
by *simp-all*
then have [*simp*]: $a\ mod\ int\ CARD('a) = a$
by *simp*
from *a* **have** $\neg\ int\ CARD('a)\ dvd\ a$
by (*auto simp add*: *zdvd-not-zless*)
then have $\neg\ CARD('a)\ dvd\ nat\ |a|$
by *simp*
with *a* **have** $\neg\ CARD('a)\ dvd\ nat\ a$
by *simp*
with *prime-card* **have** $[nat\ a\ \widehat{}\ (CARD('a) - 1) = 1]\ (mod\ CARD('a))$
by (*rule fermat-theorem*)
with *a* **have** $int\ (nat\ a\ \widehat{}\ (CARD('a) - 1)\ mod\ CARD('a)) = 1$
by (*simp add*: *cong-def*)
with *a* **have** $a\ \widehat{}\ (CARD('a) - 1)\ mod\ CARD('a) = 1$
by (*simp add*: *of-nat-mod*)
then have $a * (a\ \widehat{}\ (CARD('a) - 1)\ mod\ int\ CARD('a)) = a$
by *simp*
then have $(a * (a\ \widehat{}\ (CARD('a) - 1)\ mod\ int\ CARD('a)))\ mod\ int\ CARD('a) = a\ mod\ int\ CARD('a)$
by (*simp only*:)
then show $a\ \widehat{}\ CARD('a)\ mod\ int\ CARD('a) = a$
by (*simp add*: *mod-simps semiring-normalization-rules*(*27*))
qed
qed

lemma *mod-eq-dvd-iff-poly*: $((x::'a\ mod\text{-}ring\ poly)\ mod\ n = y\ mod\ n) = (n\ dvd\ x - y)$
proof
assume *H*: $x\ mod\ n = y\ mod\ n$
hence $x\ mod\ n - y\ mod\ n = 0$ **by** *simp*
hence $(x\ mod\ n - y\ mod\ n)\ mod\ n = 0$ **by** *simp*
hence $(x - y)\ mod\ n = 0$ **by** (*simp add*: *mod-diff-eq*)
thus $n\ dvd\ x - y$ **by** (*simp add*: *dvd-eq-mod-eq-0*)
next
assume *H*: $n\ dvd\ x - y$
then obtain *k* **where** *k*: $x-y = n*k$ **unfolding** *dvd-def* **by** *blast*
hence $x = n*k + y$ **using** *diff-eq-eq* **by** *blast*
hence $x\ mod\ n = (n*k + y)\ mod\ n$ **by** *simp*
thus $x\ mod\ n = y\ mod\ n$ **by** (*simp add*: *mod-add-left-eq*)
qed

lemma *cong-gcd-eq-poly*:
  gcd a m = *gcd b m* **if** [(*a*::′*a mod-ring poly*) = *b*] (*mod m*)
  using *that* **by** (*simp add*: *cong-def*) (*metis gcd-mod-left mod-by-0*)

lemma *coprime-h-c-poly*:
fixes *h*::′*a mod-ring poly*
assumes *c1* ≠ *c2*
shows *coprime* (*h* − [:*c1*:]) (*h* − [:*c2*:])
proof (*intro coprimeI*)
  fix *d* **assume** *d dvd h* − [:*c1*:]
  and *d dvd h* − [:*c2*:]
  hence *h mod d* = [:*c1*:] *mod d* **and** *h mod d* = [:*c2*:] *mod d*
    using *mod-eq-dvd-iff-poly* **by** *simp+*
  hence [:*c1*:] *mod d* = [:*c2*:] *mod d* **by** *simp*
  hence *d dvd* [:*c2* − *c1*:]
    by (*metis* (*no-types*) *mod-eq-dvd-iff-poly diff-pCons right-minus-eq*)
  thus *is-unit d*
   by (*metis* (*no-types*) *assms dvd-trans is-unit-monom-0 monom-0 right-minus-eq*)
qed

lemma *coprime-h-c-poly2*:
fixes *h*::′*a mod-ring poly*
assumes *coprime* (*h* − [:*c1*:]) (*h* − [:*c2*:])
and ¬ *is-unit* (*h* − [:*c1*:])
shows *c1* ≠ *c2*
using *assms coprime-id-is-unit* **by** *blast*

lemma *degree-minus-eq-right*:
fixes *p*::′*b*::*ab-group-add poly*
shows *degree q* < *degree p* ⟹ *degree* (*p* − *q*) = *degree p*
using *degree-add-eq-left*[*of* −*q p*] *degree-minus* **by** *auto*

lemma *coprime-prod*:
  fixes *A*::′*a mod-ring set* **and** *g*::′*a mod-ring* ⇒ ′*a mod-ring poly*
  assumes ∀ *x*∈*A*. *coprime* (*g a*) (*g x*)
  shows *coprime* (*g a*) (*prod* (λ*x*. *g x*) *A*)
proof −
  have *f*: *finite A* **by** *simp*
  show *?thesis*
  using *f* **using** *assms*
  proof (*induct A*)
    case (*insert x A*)
    have (∏ *c*∈*insert x A*. *g c*) = (*g x*) ∗ (∏ *c*∈*A*. *g c*)
      by (*simp add*: *insert.hyps*(*2*))
    with *insert.prems* **show** *?case*
      by (*auto simp*: *insert.hyps*(*3*) *prod-coprime-right*)

  qed *auto*
qed

lemma *coprime-prod2*:
  fixes *A*::*'b*::*semiring-gcd set*
  assumes ∀ *x*∈*A*. *coprime* (*a*) (*x*) **and** *f*: *finite A*
  shows *coprime* (*a*) (*prod* (λ*x*. *x*) *A*)
  using *f* **using** *assms*
proof (*induct A*)
  case (*insert x A*)
  have (∏ *c*∈*insert x A*. *c*) = (*x*) ∗ (∏ *c*∈*A*. *c*)
    by (*simp add*: *insert.hyps*)
  with *insert.prems* **show** *?case*
    by (*simp add*: *insert.hyps prod-coprime-right*)
qed *auto*

lemma *divides-prod*:
  fixes *g*::*'a mod-ring* ⇒ *'a mod-ring poly*
  assumes ∀ *c1 c2*. *c1* ∈ *A* ∧ *c2* ∈ *A* ∧ *c1* ≠ *c2* ⟶ *coprime* (*g c1*) (*g c2*)
  assumes ∀ *c*∈ *A*. *g c dvd f*
  shows (∏ *c*∈*A*. *g c*) *dvd f*
proof −
  have *finite-A*: *finite A* **using** *finite*[*of A*] **.**
  thus *?thesis* **using** *assms*
  proof (*induct A*)
    case (*insert x A*)
    have (∏ *c*∈*insert x A*. *g c*) = *g x* ∗ (∏ *c*∈ *A*. *g c*)
      by (*simp add*: *insert.hyps*(*2*))
    also have ... *dvd f*
    proof (*rule divides-mult*)
      show *g x dvd f* **using** *insert.prems* **by** *auto*
      show *prod g A dvd f* **using** *insert.hyps*(*3*) *insert.prems* **by** *auto*
      from *insert* **show** *Rings.coprime* (*g x*) (*prod g A*)
        by (*auto intro*: *prod-coprime-right*)
    qed
    finally show *?case* **.**
   qed *auto*
qed

lemma *poly-monom-identity-mod-p*:
  monom (*1*::*'a mod-ring*) (*CARD*(*'a*)) − *monom 1 1* = *prod* (λ*x*. [:*0*,*1*:] − [:*x*:])
(*UNIV*::*'a mod-ring set*)
  (**is** *?lhs* = *?rhs*)
proof −

let *?f*=(*λx*::*'a mod-ring*. [:*0*,*1*:] − [:*x*:])
have *?rhs dvd ?lhs*
proof (*rule divides-prod*)
{
fix *a*::*'a mod-ring*
have *poly ?lhs a* = *0*
by (*simp add*: *poly-monom*)
hence ([:*0*,*1*:] − [:*a*:]) *dvd ?lhs*
using *poly-eq-0-iff-dvd* **by** *fastforce*
}
thus ∀ *x*∈*UNIV*::*'a mod-ring set*. [:*0*, *1*:] − [:*x*:] *dvd monom 1 CARD*(*'a*) − *monom 1 1* **by** *fast*
show ∀ *c1 c2*. *c1* ∈ *UNIV* ∧ *c2* ∈ *UNIV* ∧ *c1* ≠ (*c2* :: *'a mod-ring*) ⟶ *coprime* ([:*0*, *1*:] − [:*c1*:]) ([:*0*, *1*:] − [:*c2*:])
by (*auto dest*!: *coprime-h-c-poly*[*of* - - [:*0*,*1*:]])
qed
from *this* **obtain** *g* **where** *g*: *?lhs* = *?rhs* ∗ *g* **using** *dvdE* **by** *blast*
have *degree-lhs-card*: *degree ?lhs* = *CARD*(*'a*)
proof −
have *degree* (*monom* (*1*::*'a mod-ring*) *1*) = *1* **by** (*simp add*: *degree-monom-eq*)
moreover have *d-c*: *degree* (*monom* (*1*::*'a mod-ring*) *CARD*(*'a*)) = *CARD*(*'a*)
by (*simp add*: *degree-monom-eq*)
ultimately have *degree* (*monom* (*1*::*'a mod-ring*) *1*) < *degree* (*monom* (*1*::*'a mod-ring*) *CARD*(*'a*))
using *prime-card* **unfolding** *prime-nat-iff* **by** *auto*
hence *degree ?lhs* = *degree* (*monom* (*1*::*'a mod-ring*) *CARD*(*'a*))
by (*rule degree-minus-eq-right*)
thus *?thesis* **unfolding** *d-c* **.**
qed
have *degree-rhs-card*: *degree ?rhs* = *CARD*(*'a*)
proof −
have *degree* (*prod ?f UNIV*) = *sum* (*degree* ∘ *?f*) *UNIV*
∧ *coeff* (*prod ?f UNIV*) (*sum* (*degree* ∘ *?f*) *UNIV*) = *1*
by (*rule degree-prod-sum-monic*, *auto*)
moreover have *sum* (*degree* ∘ *?f*) *UNIV* = *CARD*(*'a*) **by** *auto*
ultimately show *?thesis* **by** *presburger*
qed
have *monic-lhs*: *monic ?lhs* **using** *degree-lhs-card* **by** *auto*
have *monic-rhs*: *monic ?rhs* **by** (*rule monic-prod*, *simp*)
have *degree-eq*: *degree ?rhs* = *degree ?lhs* **unfolding** *degree-lhs-card degree-rhs-card* **..**
have *g-not-0*: *g* ≠ *0* **using** *g monic-lhs* **by** *auto*
have *degree-g0*: *degree g* = *0*
proof −
have *degree* (*?rhs* ∗ *g*) = *degree ?rhs* + *degree g*
by (*rule degree-monic-mult*[*OF monic-rhs g-not-0*])
thus *?thesis* **using** *degree-eq g* **by** *simp*
qed
have *monic-g*: *monic g* **using** *monic-factor g monic-lhs monic-rhs* **by** *auto*

have $g = 1$ **using** *monic-degree-0*[*OF monic-g*] *degree-g0* **by** *simp*
thus *?thesis* **using** *g* **by** *auto*
qed

lemma *poly-identity-mod-p*:
$v\hat{}(CARD('a)) - v = prod\ (\lambda x.\ v - [:x:])\ (UNIV::'a\ mod\text{-}ring\ set)$
proof −
have *id*: $monom\ 1\ 1 \circ_p v = v\ [:0,\ 1:] \circ_p v = v$ **unfolding** *pcompose-def*
apply (*auto*)
by (*simp add*: *fold-coeffs-def*)
have *id2*: $monom\ 1\ (CARD('a)) \circ_p v = v\ \hat{}\ (CARD('a))$ **by** (*metis id*(*1*) *pcompose-hom.hom-power x-pow-n*)
show *?thesis* **using** *arg-cong*[*OF poly-monom-identity-mod-p*, *of* $\lambda f.\ f \circ_p v$]
unfolding *pcompose-hom.hom-minus pcompose-hom.hom-prod id pcompose-const id2* **.**
qed

lemma *coprime-gcd*:
fixes *h*::*'a mod-ring poly*
assumes *Rings.coprime* $(h-[:c1:])\ (h-[:c2:])$
shows *Rings.coprime* $(gcd\ f(h-[:c1:]))\ (gcd\ f\ (h-[:c2:]))$
using *assms coprime-divisors* **by** *blast*

lemma *divides-prod-gcd*:
fixes *h*::*'a mod-ring poly*
assumes $\forall\ c1\ c2.\ c1 \in A \wedge c2 \in A \wedge c1 \neq c2 \longrightarrow coprime\ (h-[:c1:])\ (h-[:c2:])$
shows $(\prod c{\in}A.\ gcd\ f\ (h - [:c:]))\ dvd\ f$
proof −
have *finite-A*: *finite A* **using** *finite*[*of A*] **.**
thus *?thesis* **using** *assms*
proof (*induct A*)
case (*insert x A*)
have $(\prod c{\in}insert\ x\ A.\ gcd\ f\ (h - [:c:])) = (gcd\ f\ (h - [:x:])) * (\prod c{\in}\ A.\ gcd\ f\ (h - [:c:]))$
by (*simp add*: *insert.hyps*(*2*))
also have ... *dvd f*
proof (*rule divides-mult*)
show $gcd\ f\ (h - [:x:])\ dvd\ f$ **by** *simp*
show $(\prod c{\in}A.\ gcd\ f\ (h - [:c:]))\ dvd\ f$ **using** *insert.hyps*(*3*) *insert.prems* **by** *auto*
show *Rings.coprime* $(gcd\ f\ (h - [:x:]))\ (\prod c{\in}A.\ gcd\ f\ (h - [:c:]))$
by (*rule prod-coprime-right*)

(*metis Berlekamp-Type-Based.coprime-h-c-poly coprime-gcd coprime-iff-coprime insert.hyps*(*2*))
qed
finally show *?case* **.**
qed *auto*
qed

lemma *monic-prod-gcd*:
assumes *f*: *finite A* **and** *f0*: (*f* :: *'b* :: {*field-gcd*} *poly*) ≠ *0*
shows *monic* (∏ *c*∈*A*. *gcd f* (*h* − [:*c*:]))
using *f*
proof (*induct A*)
case (*insert x A*)
have *rw*: (∏ *c*∈*insert x A*. *gcd f* (*h* − [:*c*:]))
= (*gcd f* (*h* − [:*x*:])) ∗ (∏ *c*∈ *A*. *gcd f* (*h* − [:*c*:]))
by (*simp add*: *insert.hyps*)
show *?case*
proof (*unfold rw*, *rule monic-mult*)
show *monic* (*gcd f* (*h* − [:*x*:]))
using *poly-gcd-monic*[*of f*] *f0*
using *insert.prems insert-iff* **by** *blast*
show *monic* (∏ *c*∈*A*. *gcd f* (*h* − [:*c*:]))
using *insert.hyps*(*3*) *insert.prems* **by** *blast*
qed
qed *auto*

lemma *coprime-not-unit-not-dvd*:
fixes *a*::*'b*::*semiring-gcd*
assumes *a dvd b*
and *coprime b c*
and ¬ *is-unit a*
shows ¬ *a dvd c*
using *assms coprime-divisors coprime-id-is-unit* **by** *fastforce*

lemma *divides-prod2*:
fixes *A*::*'b*::*semiring-gcd set*
assumes *f*: *finite A*
and ∀ *a*∈*A*. *a dvd c*
and ∀ *a1 a2*. *a1* ∈ *A* ∧ *a2* ∈ *A* ∧ *a1* ≠ *a2* ⟶ *coprime a1 a2*
shows ∏ *A dvd c*
using *assms*
proof (*induct A*)
case (*insert x A*)
have ∏ (*insert x A*) = *x* ∗ ∏ *A* **by** (*simp add*: *insert.hyps*(*1*) *insert.hyps*(*2*))
also have ... *dvd c*
proof (*rule divides-mult*)
show *x dvd c* **by** (*simp add*: *insert.prems*)
show ∏ *A dvd c* **using** *insert* **by** *auto*
from *insert* **show** *Rings.coprime x* (∏ *A*)

    by (*auto intro*: *prod-coprime-right*)
  qed
  finally show *?case* **.**
qed *auto*

lemma *coprime-polynomial-factorization*:
  fixes *a1* :: *'b* :: {*field-gcd*} *poly*
  assumes *irr*: *as* ⊆ {*q*. *irreducible q* ∧ *monic q*}
  and *finite as* **and** *a1*: *a1* ∈ *as* **and** *a2*: *a2* ∈ *as* **and** *a1-not-a2*: *a1* ≠ *a2*
  shows *coprime a1 a2*
proof (*rule ccontr*)
  assume *not-coprime*: ¬ *coprime a1 a2*
  let *?b*= *gcd a1 a2*
  have *b-dvd-a1*: *?b dvd a1* **and** *b-dvd-a2*: *?b dvd a2* **by** *simp*+
  have *irr-a1*: *irreducible a1* **using** *a1 irr* **by** *blast*
  have *irr-a2*: *irreducible a2* **using** *a2 irr* **by** *blast*
  have *a2-not0*: *a2* ≠ *0* **using** *a2 irr* **by** *auto*
  have *degree-a1*: *degree a1* ≠ *0* **using** *irr-a1* **by** *auto*
  have *degree-a2*: *degree a2* ≠ *0* **using** *irr-a2* **by** *auto*
  have *not-a2-dvd-a1*: ¬ *a2 dvd a1*
  proof (*rule ccontr*, *simp*)
    assume *a2-dvd-a1*: *a2 dvd a1*
    from *this* **obtain** *k* **where** *k*: *a1* = *a2* ∗ *k* **unfolding** *dvd-def* **by** *auto*
    have *k-not0*: *k* ≠ *0* **using** *degree-a1 k* **by** *auto*
    show *False*
    proof (*cases degree a2* = *degree a1*)
      case *False*
      have *degree a2* < *degree a1*
        using *False a2-dvd-a1 degree-a1 divides-degree*
        by *fastforce*
      hence ¬ *irreducible a1*
        using *degree-a2 a2-dvd-a1 degree-a2*
      by (*metis degree-a1 irreducible*$_d$*D*(*2*) *irreducible*$_d$*-multD irreducible-connect-field k neq0-conv*)
      thus *False* **using** *irr-a1* **by** *contradiction*
    next
      case *True*
      have *degree a1* = *degree a2* + *degree k*
        unfolding *k* **using** *degree-mult-eq*[*OF a2-not0 k-not0*] **by** *simp*
      hence *degree k* = *0* **using** *True* **by** *simp*
      hence *k* = *1* **using** *monic-factor a1 a2 irr k monic-degree-0* **by** *auto*
      hence *a1* = *a2* **using** *k* **by** *simp*
      thus *False* **using** *a1-not-a2* **by** *contradiction*
    qed
  qed
  have *b-not0*: *?b* ≠ *0* **by** (*simp add*: *a2-not0*)
  have *degree-b*: *degree ?b* > *0*
    using *not-coprime*[*simplified*] *b-not0 is-unit-gcd is-unit-iff-degree* **by** *blast*

have *degree ?b* < *degree a2*
by (*meson b-dvd-a1 b-dvd-a2 irreducibleD′ dvd-trans gcd-dvd-1 irr-a2 not-a2-dvd-a1 not-coprime*)
hence ¬ $irreducible_d$ *a2* **using** *degree-a2 b-dvd-a2 degree-b*
by (*metis degree-smult-eq* $irreducible_d$*-dvd-smult less-not-refl3*)
thus *False* **using** *irr-a2* **by** *auto*
qed

theorem *Berlekamp-gcd-step*:
fixes *f*::*′a mod-ring poly* **and** *h*::*′a mod-ring poly*
assumes *hq-mod-f*: [*h*^(*CARD*(*′a*)) = *h*] (*mod f*) **and** *monic-f*: *monic f* **and** *sf-f*: *square-free f*
shows *f* = *prod* (λ*c. gcd f* (*h* − [:*c*:])) (*UNIV*::*′a mod-ring set*) (**is** *?lhs* = *?rhs*)
proof (*cases f=0*)
case *True*
thus *?thesis* **using** *coeff-0 monic-f zero-neq-one* **by** *auto*
next
case *False* **note** *f-not-0* = *False*
show *?thesis*
proof (*rule poly-dvd-antisym*)
show *?rhs dvd f*
using *coprime-h-c-poly* **by** (*intro divides-prod-gcd*, *auto*)
have *monic ?rhs* **by** (*rule monic-prod-gcd*[*OF - f-not-0*], *simp*)
thus *coeff f* (*degree f*) = *coeff ?rhs* (*degree ?rhs*)
using *monic-f* **by** *auto*
next
show *f dvd ?rhs*
proof −
let *?p* = *CARD*(*′a*)
obtain *P* **where** *finite-P*: *finite P*
and *f-desc-square-free*: *f* = (∏ *a*∈*P*. *a*)
and *P*: *P* ⊆ {*q. irreducible q* ∧ *monic q*}
using *monic-square-free-irreducible-factorization*[*OF monic-f sf-f*] **by** *auto*
have *f-dvd-hqh*: *f dvd* (*h*^*?p* − *h*) **using** *hq-mod-f* **unfolding** *cong-def*
using *mod-eq-dvd-iff-poly* **by** *blast*
also have *hq-h-rw*: ... = *prod* (λ*c. h* − [:*c*:]) (*UNIV*::*′a mod-ring set*)
by (*rule poly-identity-mod-p*)
finally have *f-dvd-hc*: *f dvd prod* (λ*c. h* − [:*c*:]) (*UNIV*::*′a mod-ring set*) **by** *simp*
have *f* = ∏ *P* **using** *f-desc-square-free* **by** *simp*
also have ... *dvd ?rhs*
proof (*rule divides-prod2*[*OF finite-P*])
show ∀ *a1 a2*. *a1* ∈ *P* ∧ *a2* ∈ *P* ∧ *a1* ≠ *a2* ⟶ *coprime a1 a2*
using *coprime-polynomial-factorization*[*OF P finite-P*] **by** *simp*
show ∀ *a*∈*P*. *a dvd* (∏ *c*∈*UNIV*. *gcd f* (*h* − [:*c*:]))
proof
fix *fi* **assume** *fi-P*: *fi* ∈ *P*
show *fi dvd ?rhs*

```
        proof (rule dvd-prod, auto)
          show fi dvd f using f-desc-square-free fi-P
           using dvd-prod-eqI finite-P by blast
          hence fi dvd (h^?p − h) using dvd-trans f-dvd-hqh by auto
          also have ... = prod (λc. h − [:c:]) (UNIV::'a mod-ring set)
            unfolding hq-h-rw by simp
        finally have fi-dvd-prod-hc: fi dvd prod (λc. h − [:c:]) (UNIV::'a mod-ring
set) .
          have irr-fi: irreducible (fi) using fi-P P by blast
         have fi-not-unit: ¬ is-unit fi using irr-fi by (simp add: irreducible_d D(1)
poly-dvd-1)
          have fi-dvd-hc: ∃ c∈UNIV::'a mod-ring set. fi dvd (h−[:c:])
            by (rule irreducible-dvd-prod[OF - fi-dvd-prod-hc], simp add: irr-fi)
          thus ∃ c. fi dvd h − [:c:] by simp
        qed
      qed
    qed
    finally show f dvd ?rhs .
  qed
 qed
qed
```

6.3 Definitions

```
definition berlekamp-mat :: 'a mod-ring poly ⇒ 'a mod-ring mat where
 berlekamp-mat u = (let n = degree u;
   mul-p = power-poly-f-mod u [:0,1:] (CARD('a));
   xks = power-polys mul-p u 1 n
  in
   mat-of-rows-list n (map (λ cs. let coeffs-cs = (coeffs cs);
                                     k = n − length (coeffs cs)
                                in (coeffs cs) @ replicate k 0) xks))


definition berlekamp-resulting-mat :: ('a mod-ring) poly ⇒ 'a mod-ring mat where
berlekamp-resulting-mat u = (let Q = berlekamp-mat u;
   n = dim-row Q;
   QI = mat n n (λ (i,j). if i = j then Q $$ (i,j) − 1 else Q $$ (i,j))
   in (gauss-jordan-single (transpose-mat QI)))

definition berlekamp-basis :: 'a mod-ring poly ⇒ 'a mod-ring poly list where
 berlekamp-basis u = (map (Poly o list-of-vec) (find-base-vectors (berlekamp-resulting-mat
u)))

lemma berlekamp-basis-code[code]: berlekamp-basis u =
 (map (poly-of-list o list-of-vec) (find-base-vectors (berlekamp-resulting-mat u)))
 unfolding berlekamp-basis-def poly-of-list-def ..

primrec berlekamp-factorization-main :: nat ⇒ 'a mod-ring poly list ⇒ 'a mod-ring
```

poly list ⇒ *nat* ⇒ *'a mod-ring poly list* **where**
berlekamp-factorization-main i divs (*v* # *vs*) *n* = (*if v* = *1 then berlekamp-factorization-main i divs vs n else*
if length divs = *n then divs else*
let facts = [*w* . *u* ← *divs*, *s* ← [*0* ..< *CARD*(*'a*)], *w* ← [*gcd u* (*v* − [:*of-int s*:])], *w* ≠ *1*];
(*lin*,*nonlin*) = *List.partition* (λ *q*. *degree q* = *i*) *facts*
in lin @ *berlekamp-factorization-main i nonlin vs* (*n* − *length lin*))
| *berlekamp-factorization-main i divs* [] *n* = *divs*

definition *berlekamp-monic-factorization* :: *nat* ⇒ *'a mod-ring poly* ⇒ *'a mod-ring poly list* **where**
berlekamp-monic-factorization d f = (*let*
vs = *berlekamp-basis f*;
n = *length vs*;
fs = *berlekamp-factorization-main d* [*f*] *vs n*
in fs)

6.4 Properties

lemma *power-polys-works*:
fixes *u*::*'b*::*unique-euclidean-semiring*
assumes *i*: *i* < *n* **and** *c*: *curr-p* = *curr-p mod u*
shows *power-polys mult-p u curr-p n* ! *i* = *curr-p* ∗ *mult-p* ^ *i mod u*
using *i c*
proof (*induct n arbitrary*: *curr-p i*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n*)
have *p-rw*: *power-polys mult-p u curr-p* (*Suc n*) ! *i*
= (*curr-p* # *power-polys mult-p u* (*curr-p* ∗ *mult-p mod u*) *n*) ! *i*
by *simp*
show *?case*
proof (*cases i*=*0*)
case *True*
show *?thesis* **using** *Suc.prems* **unfolding** *p-rw True* **by** *auto*
next
case *False* **note** *i-not-0* = *False*
show *?thesis*
proof (*cases i* < *n*)
case *True* **note** *i-less-n* = *True*
have *power-polys mult-p u curr-p* (*Suc n*) ! *i* = *power-polys mult-p u* (*curr-p* ∗ *mult-p mod u*) *n* ! (*i* − *1*)
unfolding *p-rw* **using** *nth-Cons-pos False* **by** *auto*
also have ... = (*curr-p* ∗ *mult-p mod u*) ∗ *mult-p* ^ (*i*−*1*) *mod u*
by (*rule Suc.hyps*) (*auto simp add*: *i-less-n less-imp-diff-less*)
also have ... = *curr-p* ∗ *mult-p* ^ *i mod u*
using *False* **by** (*cases i*) (*simp-all add*: *algebra-simps mod-simps*)
finally show *?thesis* **.**

next
case *False*
hence *i-n*: $i = n$ **using** *Suc.prems* **by** *auto*
have *power-polys mult-p u curr-p* (*Suc n*) ! $i =$ *power-polys mult-p u* (*curr-p* $*$ *mult-p mod u*) *n* ! ($n - 1$)
unfolding *p-rw* **using** *nth-Cons-pos i-n i-not-0* **by** *auto*
also have ... = (*curr-p* $*$ *mult-p mod u*) $*$ *mult-p* ^ ($n-1$) *mod u*
proof (*rule Suc.hyps*)
show $n - 1 < n$ **using** *i-n i-not-0* **by** *linarith*
show *curr-p* $*$ *mult-p mod u* = *curr-p* $*$ *mult-p mod u mod u* **by** *simp*
qed
also have ... = *curr-p* $*$ *mult-p* ^ *i mod u*
using *i-n* [*symmetric*] *i-not-0* **by** (*cases i*) (*simp-all add*: *algebra-simps mod-simps*)
finally show *?thesis* **.**
qed
qed
qed

lemma *length-power-polys*[*simp*]: *length* (*power-polys mult-p u curr-p n*) = *n*
by (*induct n arbitrary*: *curr-p*, *auto*)

lemma *Poly-berlekamp-mat*:
assumes *k*: $k <$ *degree u*
shows *Poly* (*list-of-vec* (*row* (*berlekamp-mat u*) *k*)) = [:*0*,*1*:]^(*CARD*('*a*) $*$ *k*) *mod u*
proof −
let *?map* =(*map* (λ*cs. coeffs cs* @ *replicate* (*degree u* − *length* (*coeffs cs*)) *0*)
(*power-polys* (*power-poly-f-mod u* [:*0*, *1*:] (*nat* (*int CARD*('*a*)))) *u 1* (*degree u*)))
have *row* (*berlekamp-mat u*) *k* = *row* (*mat-of-rows-list* (*degree u*) *?map*) *k*
by (*simp add*: *berlekamp-mat-def Let-def*)
also have ... = *vec-of-list* (*?map* ! *k*)
proof−
{
fix *i* **assume** *i*: $i <$ *degree u*
let *?c*= *power-polys* (*power-poly-f-mod u* [:*0*, *1*:] *CARD*('*a*)) *u 1* (*degree u*) ! *i*
let *?coeffs-c*=(*coeffs ?c*)
have *?c* = *1*$*$([:*0*, *1*:] ^ *CARD*('*a*) *mod u*)^*i mod u*
proof (*unfold power-poly-f-mod-def*, *rule power-polys-works*[*OF i*])
show *1* = *1 mod u* **using** *k mod-poly-less* **by** *force*
qed
also have ... = [:*0*, *1*:] ^ (*CARD*('*a*) $*$ *i*) *mod u* **by** (*simp add*: *power-mod power-mult*)

finally have *c-rw*: *?c* = [:*0*, *1*:] ^ (*CARD*(′*a*) ∗ *i*) *mod u* **.**
have *length ?coeffs-c* ≤ *degree u*
proof −
show *?thesis*
proof (*cases ?c* = *0*)
case *True* **thus** *?thesis* **by** *auto*
next
case *False*
have *length ?coeffs-c* = *degree* (*?c*) + *1* **by** (*rule length-coeffs*[*OF False*])
also have ... = *degree* ([:*0*, *1*:] ^ (*CARD*(′*a*) ∗ *i*) *mod u*) + *1* **using** *c-rw*
by *simp*
also have ... ≤ *degree u*
by (*metis One-nat-def add.right-neutral add-Suc-right c-rw calculation*
coeffs-def degree-0
degree-mod-less discrete gr-implies-not0 k list.size(*3*) *one-neq-zero*)
finally show *?thesis* **.**
qed
qed
then have *length ?coeffs-c* + (*degree u* − *length ?coeffs-c*) = *degree u* **by** *auto*
}
with *k* **show** *?thesis* **by** (*intro row-mat-of-rows-list*, *auto*)
qed
finally have *row-rw*: *row* (*berlekamp-mat u*) *k* = *vec-of-list* (*?map* ! *k*) **.**
have *Poly* (*list-of-vec* (*row* (*berlekamp-mat u*) *k*)) = *Poly* (*list-of-vec* (*vec-of-list*
(*?map* ! *k*)))
unfolding *row-rw* **..**
also have ... = *Poly* (*?map* ! *k*) **by** *simp*
also have ... = [:*0*,*1*:]^(*CARD*(′*a*) ∗ *k*) *mod u*
proof −
let *?cs* = (*power-polys* (*power-poly-f-mod u* [:*0*, *1*:] (*nat* (*int CARD*(′*a*)))) *u 1*
(*degree u*)) ! *k*
let *?c* = *coeffs ?cs* @ *replicate* (*degree u* − *length* (*coeffs ?cs*)) *0*
have *map-k-c*: *?map* ! *k* = *?c* **by** (*rule nth-map*, *simp add*: *k*)
have (*Poly* (*?map* ! *k*)) = *Poly* (*coeffs ?cs*) **unfolding** *map-k-c Poly-append-replicate-0*
..
also have ... = *?cs* **by** *simp*
also have ... = *power-polys* ([:*0*, *1*:] ^ *CARD*(′*a*) *mod u*) *u 1* (*degree u*) ! *k*
by (*simp add*: *power-poly-f-mod-def*)
also have ... = *1*∗ ([:*0*,*1*:]^(*CARD*(′*a*)) *mod u*) ^ *k mod u*
proof (*rule power-polys-works*[*OF k*])
show *1* = *1 mod u* **using** *k mod-poly-less* **by** *force*
qed
also have ... = ([:*0*,*1*:]^(*CARD*(′*a*)) *mod u*) ^ *k mod u* **by** *auto*
also have ... = [:*0*,*1*:]^(*CARD*(′*a*) ∗ *k*) *mod u* **by** (*simp add*: *power-mod*
power-mult)
finally show *?thesis* **.**
qed
finally show *?thesis* **.**
qed

corollary *Poly-berlekamp-cong-mat*:
assumes *k*: $k < degree\ u$
shows $[Poly\ (list\text{-}of\text{-}vec\ (row\ (berlekamp\text{-}mat\ u)\ k)) = [:0,1:]\widehat{\ }(CARD('a) * k)]$ $(mod\ u)$
using *Poly-berlekamp-mat*[*OF k*] **unfolding** *cong-def* **by** *auto*

lemma *mat-of-rows-list-dim*[*simp*]:
mat-of-rows-list n vs ∈ *carrier-mat* (*length vs*) *n*
dim-row (*mat-of-rows-list n vs*) = *length vs*
dim-col (*mat-of-rows-list n vs*) = *n*
unfolding *mat-of-rows-list-def* **by** *auto*

lemma *berlekamp-mat-closed*[*simp*]:
berlekamp-mat u ∈ *carrier-mat* (*degree u*) (*degree u*)
dim-row (*berlekamp-mat u*) = *degree u*
dim-col (*berlekamp-mat u*) = *degree u*
unfolding *carrier-mat-def berlekamp-mat-def Let-def* **by** *auto*

lemma *vec-of-list-coeffs-nth*:
assumes *i*: *i* ∈ {..*degree h*} **and** *h-not0*: *h* ≠ *0*
shows *vec-of-list* (*coeffs h*) $ *i* = *coeff h i*
proof −
have *vec-of-list* (*map* (*coeff h*) [*0*..<*degree h*] @ [*coeff h* (*degree h*)]) $ *i* = *coeff h i*
using *i*
by (*transfer′*, *auto simp add*: *mk-vec-def*)
(*metis* (*no-types*, *lifting*) *Cons-eq-append-conv coeffs-def coeffs-nth degree-0 diff-zero length-upt less-eq-nat.simps*(*1*) *list.simps*(*8*) *list.simps*(*9*) *map-append nth-Cons-0 upt-Suc upt-eq-Nil-conv*)
thus *vec-of-list* (*coeffs h*) $ *i* = *coeff h i*
using *i h-not0*
unfolding *coeffs-def* **by** *simp*
qed

lemma *poly-mod-sum*:
fixes *x y z* :: ′*b*::*field poly*
assumes *f*: *finite A*
shows *sum f A mod z* = *sum* (λ*i*. *f i mod z*) *A*
using *f*
by (*induct*, *auto simp add*: *poly-mod-add-left*)

lemma *prime-not-dvd-fact*:
assumes *kn*: *k* < *n* **and** *prime-n*: *prime n*
shows ¬ *n dvd fact k*

using *kn*
proof (*induct k*)
  case *0*
  thus *?case* **using** *prime-n* **unfolding** *prime-nat-iff* **by** *auto*
next
  case (*Suc k*)
  show *?case*
  proof (*rule ccontr*, *unfold not-not*)
    assume *n dvd fact* (*Suc k*)
    also have ... = *Suc k* $*$ $\prod$ {*1..k*} **unfolding** *fact-Suc* **unfolding** *fact-prod* **by** *simp*
    finally have *n dvd Suc k* $*$ $\prod$ {*1..k*} **.**
    hence *n dvd Suc k* $\vee$ *n dvd* $\prod$ {*1..k*} **using** *prime-dvd-mult-eq-nat*[*OF prime-n*] **by** *blast*
    moreover have $\neg$ *n dvd Suc k* **by** (*simp add*: *Suc.prems*(*1*) *nat-dvd-not-less*)
    moreover hence $\neg$ *n dvd* $\prod$ {*1..k*} **using** *Suc.hyps Suc.prems*
      using *Suc-lessD fact-prod*[*of k*] **by** (*metis of-nat-id*)
    ultimately show *False* **by** *simp*
  qed
qed

lemma *dvd-choose-prime*:
assumes *kn*: $k < n$ **and** *k*: $k \neq 0$ **and** *n*: $n \neq 0$ **and** *prime-n*: *prime n*
shows *n dvd* (*n choose k*)
proof −
  have *n dvd* (*fact n*) **by** (*simp add*: *fact-num-eq-if n*)
  moreover have $\neg$ *n dvd* (*fact k* $*$ *fact* (*n*−*k*))
  proof (*rule ccontr*, *simp*)
    assume *n dvd fact k* $*$ *fact* (*n* − *k*)
    hence *n dvd fact k* $\vee$ *n dvd fact* (*n* − *k*) **using** *prime-dvd-mult-eq-nat*[*OF prime-n*] **by** *simp*
    moreover have $\neg$ *n dvd* (*fact k*) **by** (*rule prime-not-dvd-fact*[*OF kn prime-n*])
    moreover have $\neg$ *n dvd fact* (*n* − *k*) **using** *prime-not-dvd-fact*[*OF - prime-n*] *kn k* **by** *simp*
    ultimately show *False* **by** *simp*
  qed
  moreover have (*fact n*::*nat*) = *fact k* $*$ *fact* (*n*−*k*) $*$ (*n choose k*)
    using *binomial-fact-lemma kn* **by** *auto*
  ultimately show *?thesis* **using** *prime-n*
    by (*auto simp add*: *prime-dvd-mult-iff*)
qed

lemma *add-power-poly-mod-ring*:
fixes *x* :: *'a mod-ring poly*
shows $(x + y)$ ^ *CARD*(*'a*) = *x* ^ *CARD*(*'a*) + *y* ^ *CARD*(*'a*)
proof −

let *?A*={*0*..*CARD*(*'a*)}
let *?f*=λ*k*. *of-nat* (*CARD*(*'a*) *choose k*) ∗ *x* ^ *k* ∗ *y* ^ (*CARD*(*'a*) − *k*)
have *A-rw*: *?A* = *insert CARD*(*'a*) (*insert 0* (*?A* − {*0*} − {*CARD*(*'a*)}))
by *fastforce*
have *sum0*: *sum ?f* (*?A* − {*0*} − {*CARD*(*'a*)}) = *0*
proof (*rule sum.neutral*, *rule*)
fix *xa* **assume** *xa*: *xa* ∈ {*0*..*CARD*(*'a*)} − {*0*} − {*CARD*(*'a*)}
have *card-dvd-choose*: *CARD*(*'a*) *dvd* (*CARD*(*'a*) *choose xa*)
proof (*rule dvd-choose-prime*)
show *xa* < *CARD*(*'a*) **using** *xa* **by** *simp*
show *xa* ≠ *0* **using** *xa* **by** *simp*
show *CARD*(*'a*) ≠ *0* **by** *simp*
show *prime CARD*(*'a*) **by** (*rule prime-card*)
qed
hence *rw0*: *of-int* (*CARD*(*'a*) *choose xa*) = (*0* :: *'a mod-ring*)
by *transfer simp*
have *of-nat* (*CARD*(*'a*) *choose xa*) = [:*of-int* (*CARD*(*'a*) *choose xa*) :: *'a mod-ring*:]
by (*simp add*: *of-nat-poly*)
also have ... = [:*0*:] **using** *rw0* **by** *simp*
finally show *of-nat* (*CARD*(*'a*) *choose xa*) ∗ *x* ^ *xa* ∗ *y* ^ (*CARD*(*'a*) − *xa*) = *0* **by** *auto*
qed
have (*x* + *y*)^*CARD*(*'a*)
= (∑ *k* = *0*..*CARD*(*'a*). *of-nat* (*CARD*(*'a*) *choose k*) ∗ *x* ^ *k* ∗ *y* ^ (*CARD*(*'a*) − *k*))
unfolding *binomial-ring* **by** (*rule sum.cong*, *auto*)
also have ... = *sum ?f* (*insert CARD*(*'a*) (*insert 0* (*?A* − {*0*} − {*CARD*(*'a*)})))
using *A-rw* **by** *simp*
also have ... = *?f 0* + *?f CARD*(*'a*) + *sum ?f* (*?A* − {*0*} − {*CARD*(*'a*)}) **by** *auto*
also have ... = *x*^*CARD*(*'a*) + *y*^*CARD*(*'a*) **unfolding** *sum0* **by** *auto*
finally show *?thesis* **.**
qed

lemma *power-poly-sum-mod-ring*:
fixes *f* :: *'b* ⇒ *'a mod-ring poly*
assumes *f*: *finite A*
shows (*sum f A*) ^ *CARD*(*'a*) = *sum* (λ*i*. (*f i*) ^ *CARD*(*'a*)) *A*
using *f* **by** (*induct*, *auto simp add*: *add-power-poly-mod-ring*)

lemma *poly-power-card-as-sum-of-monoms*:
fixes *h* :: *'a mod-ring poly*
shows *h* ^ *CARD*(*'a*) = (∑ *i*≤*degree h*. *monom* (*coeff h i*) (*CARD*(*'a*)∗*i*))
proof −
have *h* ^ *CARD*(*'a*) = (∑ *i*≤*degree h*. *monom* (*coeff h i*) *i*) ^ *CARD*(*'a*)
by (*simp add*: *poly-as-sum-of-monoms*)

also have ... = ($\sum i \le degree\ h.$ (monom (coeff h i) i) ^ CARD('a))
  by (simp add: power-poly-sum-mod-ring)
also have ... = ($\sum i \le degree\ h.$ monom (coeff h i) (CARD('a)∗i))
proof (rule sum.cong, rule)
  fix x assume x: x ∈ {..degree h}
  show monom (coeff h x) x ^ CARD('a) = monom (coeff h x) (CARD('a) ∗ x)
    by (unfold poly-eq-iff, auto simp add: monom-power)
qed
finally show ?thesis .
qed

lemma *degree-Poly-berlekamp-le*:
assumes *i*: $i < degree\ u$
shows *degree* (*Poly* (*list-of-vec* (*row* (*berlekamp-mat u*) *i*))) < *degree u*
by (*metis Poly-berlekamp-mat degree-0 degree-mod-less gr-implies-not0 i linorder-neqE-nat*)

lemma *monom-card-pow-mod-sum-berlekamp*:
assumes *i*: $i < degree\ u$
shows *monom 1* (*CARD*('*a*) ∗ *i*) *mod u* = ($\sum j<degree\ u.$ *monom* ((*berlekamp-mat u*) $$ (*i*,*j*)) *j*)
proof −
  let *?p* = *Poly* (*list-of-vec* (*row* (*berlekamp-mat u*) *i*))
  have *degree-not-0*: *degree u* ≠ *0* **using** *i* **by** *simp*
  hence *set-rw*: {..*degree u* − *1*} = {..<*degree u*} **by** *auto*
  have *degree-le*: *degree ?p* < *degree u*
    by (*rule degree-Poly-berlekamp-le*[*OF i*])
  hence *degree-le2*: *degree ?p* ≤ *degree u* − *1* **by** *auto*
  have *monom 1* (*CARD*('*a*) ∗ *i*) *mod u* = [:*0*, *1*:] ^ (*CARD*('*a*) ∗ *i*) *mod u*
    using *x-as-monom x-pow-n* **by** *metis*
  also have ... = *?p*
    unfolding *Poly-berlekamp-mat*[*OF i*] **by** *simp*
  also have ... = ($\sum i \le degree\ u - 1.$ *monom* (*coeff ?p i*) *i*)
      using *degree-le2 poly-as-sum-of-monoms'* **by** *fastforce*
  also have ... = ($\sum i<degree\ u.$ *monom* (*coeff ?p i*) *i*) **using** *set-rw* **by** *auto*
  also have ... = ($\sum j<degree\ u.$ *monom* ((*berlekamp-mat u*) $$ (*i*,*j*)) *j*)
  proof (*rule sum.cong*, *rule*)
    fix *x* **assume** *x*: *x* ∈ {..<*degree u*}
    have *coeff ?p x* = *berlekamp-mat u* $$ (*i*, *x*)
    proof (*rule coeff-Poly-list-of-vec-nth*)
      show *x* < *dim-col* (*berlekamp-mat u*) **using** *x* **by** *auto*
    qed
    thus *monom* (*coeff ?p x*) *x* = *monom* (*berlekamp-mat u* $$ (*i*, *x*)) *x*
      by (*simp add*: *poly-eq-iff*)
  qed
  finally show *?thesis* **.**

qed

lemma *col-scalar-prod-as-sum*:
assumes *dim-vec v = dim-row A*
shows *col A j* $\cdot$ *v* = ($\sum$ *i = 0..<dim-vec v. A $$ (i,j) * v $ i*)
  using *assms*
  unfolding *col-def scalar-prod-def*
  by *transfer'* (*rule sum.cong*, *transfer'*, *auto simp add*: *mk-vec-def mk-mat-def*)

lemma *row-transpose-scalar-prod-as-sum*:
assumes *j*: *j < dim-col A* **and** *dim-v*: *dim-vec v = dim-row A*
shows *row* (*transpose-mat A*) *j* $\cdot$ *v* = ($\sum$ *i = 0..<dim-vec v. A $$ (i,j) * v $ i*)
proof –
  have *row* (*transpose-mat A*) *j* $\cdot$ *v = col A j* $\cdot$ *v* **using** *j row-transpose* **by** *auto*
  also have ... = ($\sum$ *i = 0..<dim-vec v. A $$ (i,j) * v $ i*)
    by (*rule col-scalar-prod-as-sum*[*OF dim-v*])
  finally show *?thesis* **.**
qed

lemma *poly-as-sum-eq-monoms*:
assumes *ss-eq*: ($\sum$ *i<n. monom* (*f i*) *i*) = ($\sum$ *i<n. monom* (*g i*) *i*)
and *a-less-n*: *a<n*
shows *f a = g a*
proof –
  let *?f=λi. if i = a then f i else 0*
  let *?g=λi. if i = a then g i else 0*
  have *sum-f-0*: *sum ?f* ({..<*n*} – {*a*}) = *0* **by** (*rule sum.neutral*, *auto*)
  have *coeff* ($\sum$ *i<n. monom* (*f i*) *i*) *a = coeff* ($\sum$ *i<n. monom* (*g i*) *i*) *a*
    using *ss-eq* **unfolding** *poly-eq-iff* **by** *simp*
  hence ($\sum$ *i<n. coeff* (*monom* (*f i*) *i*) *a*) = ($\sum$ *i<n. coeff* (*monom* (*g i*) *i*) *a*)
    by (*simp add*: *coeff-sum*)
  hence *1*: ($\sum$ *i<n. if i = a then f i else 0*) = ($\sum$ *i<n. if i = a then g i else 0*)
    unfolding *coeff-monom* **by** *auto*
  have *set-rw*: {..<*n*} = (*insert a* ({..<*n*} – {*a*})) **using** *a-less-n* **by** *auto*
  have ($\sum$ *i<n. if i = a then f i else 0*) = *sum ?f* (*insert a* ({..<*n*} – {*a*}))
    using *set-rw* **by** *auto*
  also have ... = *?f a + sum ?f* ({..<*n*} – {*a*})
    by (*simp add*: *sum.insert-remove*)
  also have ... = *?f a* **using** *sum-f-0* **by** *simp*
  finally have *2*: ($\sum$ *i<n. if i = a then f i else 0*) = *?f a* **.**
  have *sum ?g* {..<*n*} = *sum ?g* (*insert a* ({..<*n*} – {*a*}))
    using *set-rw* **by** *auto*
  also have ... = *?g a + sum ?g* ({..<*n*} – {*a*})
    by (*simp add*: *sum.insert-remove*)
  also have ... = *?g a* **using** *sum-f-0* **by** *simp*
  finally have *3*: ($\sum$ *i<n. if i = a then g i else 0*) = *?g a* **.**

show *?thesis* **using** *1 2 3* **by** *auto*
qed

lemma *dim-vec-of-list-h*:
assumes *degree h* $<$ *degree u*
shows *dim-vec* (*vec-of-list* ((*coeffs h*) @ *replicate* (*degree u* $-$ *length* (*coeffs h*)) *0*)) $=$ *degree u*
proof $-$
  have *length* (*coeffs h*) $\leq$ *degree u*
    by (*metis Suc-leI assms coeffs-0-eq-Nil degree-0 length-coeffs-degree list.size*(*3*) *not-le-imp-less order.asym*)
  thus *?thesis* **by** *simp*
qed

lemma *vec-of-list-coeffs-nth$'$*:
assumes *i*: *i* $\in$ {..*degree h*} **and** *h-not0*: *h* $\neq$ *0*
assumes *degree h* $<$ *degree u*
shows *vec-of-list* ((*coeffs h*) @ *replicate* (*degree u* $-$ *length* (*coeffs h*)) *0*) \$ *i* $=$ *coeff h i*
using *assms*
by (*transfer$'$*, *auto simp add*: *mk-vec-def coeffs-nth length-coeffs-degree nth-append*)

lemma *vec-of-list-coeffs-replicate-nth-0*:
assumes *i*: *i* $\in$ {..<*degree u*}
shows *vec-of-list* (*coeffs 0* @ *replicate* (*degree u* $-$ *length* (*coeffs 0*)) *0*) \$ *i* $=$ *coeff 0 i*
using *assms*
by (*transfer$'$*, *auto simp add*: *mk-vec-def*)

lemma *vec-of-list-coeffs-replicate-nth*:
assumes *i*: *i* $\in$ {..<*degree u*}
assumes *degree h* $<$ *degree u*
shows *vec-of-list* ((*coeffs h*) @ *replicate* (*degree u* $-$ *length* (*coeffs h*)) *0*) \$ *i* $=$ *coeff h i*
proof (*cases h* $=$ *0*)
  case *True*
  thus *?thesis* **using** *vec-of-list-coeffs-replicate-nth-0 i* **by** *auto*
next
  case *False* **note** *h-not0* $=$ *False*
  show *?thesis*
  proof (*cases i* $\in${..*degree h*})
    case *True* **thus** *?thesis* **using** *assms vec-of-list-coeffs-nth$'$ h-not0* **by** *simp*

next
case *False*
have *c0*: *coeff h i* = *0* **using** *False le-degree* **by** *auto*
thus *?thesis*
using *assms False h-not0*
by (*transfer'*, *auto simp add*: *mk-vec-def length-coeffs-degree nth-append c0*)
qed
qed

lemma *equation-13*:
fixes *u h*
defines *H*: $H \equiv$ *vec-of-list* ((*coeffs h*) @ *replicate* (*degree u* − *length* (*coeffs h*)) *0*)
assumes *deg-le*: *degree h* < *degree u*
shows [*h*^*CARD*(*'a*) = *h*] (*mod u*) $\longleftrightarrow$ (*transpose-mat* (*berlekamp-mat u*)) $*_v$ *H* = *H*
(**is** *?lhs* = *?rhs*)
proof −
have *f*: *finite* {..*degree u*} **by** *auto*
have [*simp*]: *dim-vec H* = *degree u* **unfolding** *H* **using** *dim-vec-of-list-h deg-le* **by** *simp*
let *?B* = (*berlekamp-mat u*)
let *?f* = λ*i*. (*transpose-mat ?B* $*_v$ *H*) $ *i*
show *?thesis*
proof
assume *rhs*: *?rhs*
have *dimv-h-dimr-B*: *dim-vec H* = *dim-row ?B*
by (*metis berlekamp-mat-closed*(*2*) *berlekamp-mat-closed*(*3*)
dim-mult-mat-vec index-transpose-mat(*2*) *rhs*)
have *degree-h-less-dim-H*: *degree h* < *dim-vec H* **by** (*auto simp add*: *deg-le*)
have *set-rw*: {..*degree u* − *1*} = {..<*degree u*} **using** *deg-le* **by** *auto*
have *degree h* ≤ *degree u* − *1* **using** *deg-le* **by** *simp*
hence *h* = ($\sum$*j*≤*degree u* − *1*. *monom* (*coeff h j*) *j*) **using** *poly-as-sum-of-monoms'* **by** *fastforce*
also have ... = ($\sum$*j*<*degree u*. *monom* (*coeff h j*) *j*) **using** *set-rw* **by** *simp*
also have ... = ($\sum$*j*<*degree u*. *monom* (*?f j*) *j*)
proof (*rule sum.cong*, *rule*+)
fix *j* **assume** *i*: *j* ∈ {..<*degree u*}
have (*coeff h j*) = *?f j*
using *rhs vec-of-list-coeffs-replicate-nth*[*OF i deg-le*]
unfolding *H* **by** *presburger*
thus *monom* (*coeff h j*) *j* = *monom* (*?f j*) *j*
by *simp*
qed
also have ... = ($\sum$*j*<*degree u*. *monom* (*row* (*transpose-mat ?B*) *j* · *H*) *j*)
by (*rule sum.cong*, *auto*)

also have ... = ($\sum j$<*degree u*. *monom* ($\sum i$ = *0*..<*dim-vec H*. *?B* $$ (*i*,*j*) ∗ *H* $ *i*) *j*)
proof (*rule sum.cong*, *rule*)
fix *x* **assume** *x*: *x* ∈ {..<*degree u*}
show *monom* (*row* (*transpose-mat ?B*) *x* · *H*) *x* =
monom ($\sum i$ = *0*..<*dim-vec H*. *?B* $$ (*i*, *x*) ∗ *H* $ *i*) *x*
proof (*unfold monom-eq-iff*, *rule row-transpose-scalar-prod-as-sum*[*OF* - *dimv-h-dimr-B*])
show *x* < *dim-col ?B* **using** *x deg-le* **by** *auto*
qed
qed
also have ... = ($\sum j$<*degree u*. $\sum i$ = *0*..<*dim-vec H*. *monom* (*?B* $$ (*i*,*j*) ∗ *H* $ *i*) *j*)
by (*auto simp add*: *monom-sum*)
also have ... = ($\sum i$ = *0*..<*dim-vec H*. $\sum j$<*degree u*. *monom* (*?B* $$ (*i*,*j*) ∗ *H* $ *i*) *j*)
by (*rule sum.swap*)
also have ... = ($\sum i$ = *0*..<*dim-vec H*. $\sum j$<*degree u*. *monom* (*H* $ *i*) *0* ∗ *monom* (*?B* $$ (*i*,*j*)) *j*)
proof (*rule sum.cong*, *rule*, *rule sum.cong*, *rule*)
fix *x xa*
show *monom* (*?B* $$ (*x*, *xa*) ∗ *H* $ *x*) *xa* = *monom* (*H* $ *x*) *0* ∗ *monom* (*?B* $$ (*x*, *xa*)) *xa*
by (*simp add*: *mult-monom*)
qed
also have ... = ($\sum i$ = *0*..<*dim-vec H*. (*monom* (*H* $ *i*) *0*) ∗ ($\sum j$<*degree u*. *monom* (*?B* $$ (*i*,*j*)) *j*))
by (*rule sum.cong*, *auto simp*: *sum-distrib-left*)
also have ... = ($\sum i$ = *0*..<*dim-vec H*. (*monom* (*H* $ *i*) *0*) ∗ (*monom 1* (*CARD*(′*a*) ∗ *i*) *mod u*))
proof (*rule sum.cong*, *rule*)
fix *x* **assume** *x*: *x* ∈ {*0*..<*dim-vec H*}
have ($\sum j$<*degree u*. *monom* (*?B* $$ (*x*, *j*)) *j*) = (*monom 1* (*CARD*(′*a*) ∗ *x*) *mod u*)
proof (*rule monom-card-pow-mod-sum-berlekamp*[*symmetric*])
show *x* < *degree u* **using** *x dimv-h-dimr-B* **by** *auto*
qed
thus *monom* (*H* $ *x*) *0* ∗ ($\sum j$<*degree u*. *monom* (*?B* $$ (*x*, *j*)) *j*) =
monom (*H* $ *x*) *0* ∗ (*monom 1* (*CARD*(′*a*) ∗ *x*) *mod u*) **by** *presburger*
qed
also have ... = ($\sum i$ = *0*..<*dim-vec H*. *monom* (*H* $ *i*) (*CARD*(′*a*) ∗ *i*) *mod u*)
proof (*rule sum.cong*, *rule*)
fix *x*
have *h-rw*: *monom* (*H* $ *x*) *0 mod u* = *monom* (*H* $ *x*) *0*
by (*metis deg-le degree-pCons-eq-if gr-implies-not-zero linorder-neqE-nat mod-poly-less monom-0*)
have *monom* (*H* $ *x*) (*CARD*(′*a*) ∗ *x*) = *monom* (*H* $ *x*) *0* ∗ *monom 1* (*CARD*(′*a*) ∗ *x*)

unfolding *mult-monom* **by** *simp*
also have ... = *smult* (*H* \$ *x*) (*monom 1* (*CARD*('*a*) ∗ *x*))
by (*simp add*: *monom-0*)
also have ... *mod u* = *Polynomial.smult* (*H* \$ *x*) (*monom 1* (*CARD*('*a*) ∗ *x*) *mod u*)
using *mod-smult-left* **by** *auto*
also have ... = *monom* (*H* \$ *x*) *0* ∗ (*monom 1* (*CARD*('*a*) ∗ *x*) *mod u*)
by (*simp add*: *monom-0*)
finally show *monom* (*H* \$ *x*) *0* ∗ (*monom 1* (*CARD*('*a*) ∗ *x*) *mod u*)
= *monom* (*H* \$ *x*) (*CARD*('*a*) ∗ *x*) *mod u* **..**
qed
also have ... = ($\sum$ *i* = *0*..<*dim-vec H*. *monom* (*H* \$ *i*) (*CARD*('*a*) ∗ *i*)) *mod u*
by (*simp add*: *poly-mod-sum*)
also have ... = ($\sum$ *i* = *0*..<*dim-vec H*. *monom* (*coeff h i*) (*CARD*('*a*) ∗ *i*)) *mod u*
proof (*rule arg-cong*[*of* - - λ*x*. *x mod u*], *rule sum.cong*, *rule*)
fix *x* **assume** *x*: *x* ∈ {*0*..<*dim-vec H*}
have *H* \$ *x* = (*coeff h x*)
proof (*unfold H*, *rule vec-of-list-coeffs-replicate-nth*[*OF* - *deg-le*])
show *x* ∈ {..<*degree u*} **using** *x* **by** *auto*
qed
thus *monom* (*H* \$ *x*) (*CARD*('*a*) ∗ *x*) = *monom* (*coeff h x*) (*CARD*('*a*) ∗ *x*)
by *simp*
qed
also have ... = ($\sum$ *i*≤*degree h*. *monom* (*coeff h i*) (*CARD*('*a*) ∗ *i*)) *mod u*
proof (*rule arg-cong*[*of* - - λ*x*. *x mod u*])
let *?f*=λ*i*. *monom* (*coeff h i*) (*CARD*('*a*) ∗ *i*)
have *ss0*: ($\sum$ *i* = *degree h* + *1* ..< *dim-vec H*. *?f i*) = *0*
by (*rule sum.neutral*, *simp add*: *coeff-eq-0*)
have *set-rw*: {*0*..< *dim-vec H*} = {*0*..*degree h*} ∪ {*degree h* + *1* ..< *dim-vec H*}
using *degree-h-less-dim-H* **by** *auto*
have ($\sum$ *i* = *0*..<*dim-vec H*. *?f i*) = ($\sum$ *i* = *0*..*degree h*. *?f i*) + ($\sum$ *i* = *degree h* + *1* ..< *dim-vec H*. *?f i*)
unfolding *set-rw* **by** (*rule sum.union-disjoint*, *auto*)
also have ... = ($\sum$ *i* = *0*..*degree h*. *?f i*) **unfolding** *ss0* **by** *auto*
finally show ($\sum$ *i* = *0*..<*dim-vec H*. *?f i*) = ($\sum$ *i*≤*degree h*. *?f i*)
by (*simp add*: *atLeast0AtMost*)
qed
also have ... = *h*^*CARD*('*a*) *mod u*
using *poly-power-card-as-sum-of-monoms* **by** *auto*
finally show *?lhs*
unfolding *cong-def*
using *deg-le*
by (*simp add*: *mod-poly-less*)
next
assume *lhs*: *?lhs*
have *deg-le'*: *degree h* ≤ *degree u* − *1* **using** *deg-le* **by** *auto*

have *set-rw*: {..<*degree u*} = {..*degree u* −*1*} **using** *deg-le* **by** *auto*
hence (∑ *i*<*degree u*. *monom* (*coeff h i*) *i*) = (∑ *i* ≤ *degree u* − *1*. *monom* (*coeff h i*) *i*) **by** *simp*
also have ... = (∑ *i*≤*degree h*. *monom* (*coeff h i*) *i*)
unfolding *poly-as-sum-of-monoms*
using *poly-as-sum-of-monoms′ deg-le′* **by** *auto*
also have ... = (∑ *i*≤*degree h*. *monom* (*coeff h i*) *i*) *mod u*
by (*simp add*: *deg-le mod-poly-less poly-as-sum-of-monoms*)
also have ... = (∑ *i*≤*degree h*. *monom* (*coeff h i*) (*CARD*(′*a*)∗*i*)) *mod u*
using *lhs*
unfolding *cong-def poly-as-sum-of-monoms poly-power-card-as-sum-of-monoms*
by *auto*
also have ... = (∑ *i*≤*degree h*. *monom* (*coeff h i*) *0* ∗ *monom 1* (*CARD*(′*a*)∗*i*)) *mod u*
by (*rule arg-cong*[*of - - λx. x mod u*], *rule sum.cong*, *simp-all add*: *mult-monom*)
also have ... = (∑ *i*≤*degree h*. *monom* (*coeff h i*) *0* ∗ *monom 1* (*CARD*(′*a*)∗*i*) *mod u*)
by (*simp add*: *poly-mod-sum*)
also have ... = (∑ *i*≤*degree h*. *monom* (*coeff h i*) *0* ∗ (*monom 1* (*CARD*(′*a*)∗*i*) *mod u*))
proof (*rule sum.cong*, *rule*)
fix *x* **assume** *x*: *x* ∈ {..*degree h*}
have *h-rw*: *monom* (*coeff h x*) *0 mod u* = *monom* (*coeff h x*) *0*
by (*metis deg-le degree-pCons-eq-if gr-implies-not-zero*
linorder-neqE-nat mod-poly-less monom-0)
have *monom* (*coeff h x*) *0* ∗ *monom 1* (*CARD*(′*a*) ∗ *x*) = *smult* (*coeff h x*) (*monom 1* (*CARD*(′*a*) ∗ *x*))
by (*simp add*: *monom-0*)
also have ... *mod u* = *Polynomial.smult* (*coeff h x*) (*monom 1* (*CARD*(′*a*) ∗ *x*) *mod u*)
using *mod-smult-left* **by** *auto*
also have ... = *monom* (*coeff h x*) *0* ∗ (*monom 1* (*CARD*(′*a*) ∗ *x*) *mod u*)
by (*simp add*: *monom-0*)
finally show *monom* (*coeff h x*) *0* ∗ *monom 1* (*CARD*(′*a*) ∗ *x*) *mod u*
= *monom* (*coeff h x*) *0* ∗ (*monom 1* (*CARD*(′*a*) ∗ *x*) *mod u*) **.**
qed
also have ... = (∑ *i*≤*degree h*. *monom* (*coeff h i*) *0* ∗ (∑ *j*<*degree u*. *monom* (*?B* $$ (*i*, *j*)) *j*))
proof (*rule sum.cong*, *rule*)
fix *x* **assume** *x*: *x* ∈ {..*degree h*}
have (*monom 1* (*CARD*(′*a*) ∗ *x*) *mod u*) = (∑ *j*<*degree u*. *monom* (*?B* $$ (*x*, *j*)) *j*)
proof (*rule monom-card-pow-mod-sum-berlekamp*)
show *x* < *degree u* **using** *x deg-le* **by** *auto*
qed
thus *monom* (*coeff h x*) *0* ∗ (*monom 1* (*CARD*(′*a*) ∗ *x*) *mod u*) =
monom (*coeff h x*) *0* ∗ (∑ *j*<*degree u*. *monom* (*?B* $$ (*x*, *j*)) *j*) **by** *simp*
qed
also have ... = (∑ *i*<*degree u*. *monom* (*coeff h i*) *0* ∗ (∑ *j*<*degree u*. *monom*

```
(?B $$ (i, j)) j))
  proof -
    let ?f=λi. monom (coeff h i) 0 * (∑j<degree u. monom (?B $$ (i, j)) j)
    have ss0: (∑i=degree h+1 ..< degree u. ?f i) = 0
      by (rule sum.neutral, simp add: coeff-eq-0)
    have set-rw: {0..<degree u} = {0..degree h} ∪ {degree h+1..<degree u} using
deg-le by auto
    have (∑i=0..<degree u. ?f i) = (∑i=0..degree h. ?f i) + (∑i=degree h+1
..< degree u. ?f i)
    unfolding set-rw by (rule sum.union-disjoint, auto)
    also have ... = (∑i=0..degree h. ?f i) using ss0 by simp
    finally show ?thesis
      by (simp add: atLeast0AtMost atLeast0LessThan)
  qed
  also have ... = (∑i<degree u. (∑j<degree u. monom (coeff h i) 0 * monom
(?B $$ (i, j)) j))
    by (simp add: sum-distrib-left)
  also have ... = (∑i<degree u. (∑j<degree u. monom (coeff h i * ?B $$ (i, j))
j))
    by (simp add: mult-monom)
  also have ... = (∑j<degree u. (∑i<degree u. monom (coeff h i * ?B $$ (i, j))
j))
    using sum.swap by auto
  also have ... = (∑j<degree u. monom (∑i<degree u. (coeff h i * ?B $$ (i,
j))) j)
    by (simp add: monom-sum)
  finally have ss-rw: (∑i<degree u. monom (coeff h i) i)
    = (∑j<degree u. monom (∑i<degree u. coeff h i * ?B $$ (i, j)) j) .
  have coeff-eq-sum: ∀i. i < degree u ⟶ coeff h i = (∑j<degree u. coeff h j *
?B $$ (j, i))
    using poly-as-sum-eq-monoms[OF ss-rw] by fast
  have coeff-eq-sum': ∀i. i < degree u ⟶ H $ i = (∑j<degree u. H $ j * ?B $$
(j, i))
  proof (rule+)
    fix i assume i: i < degree u
    have H $ i = coeff h i by (simp add: H deg-le i vec-of-list-coeffs-replicate-nth)
    also have ... = (∑j<degree u. coeff h j * ?B $$ (j, i)) using coeff-eq-sum i
by blast
    also have ... = (∑j<degree u. H $ j * ?B $$ (j, i))
      by (rule sum.cong, auto simp add: H deg-le vec-of-list-coeffs-replicate-nth)
    finally show H $ i = (∑j<degree u. H $ j * ?B $$ (j, i)) .
  qed
  show (transpose-mat (?B)) *v H = H
  proof (rule eq-vecI)
    fix i
    show dim-vec (transpose-mat ?B *v H) = dim-vec (H) by auto
    assume i: i < dim-vec (H)
    have (transpose-mat ?B *v H) $ i = row (transpose-mat ?B) i · H using i by
simp
```

also have ... = $(\sum j = 0..<dim\text{-}vec\ H.\ ?B\ \$\$\ (j,\ i) * H\ \$\ j)$
proof (*rule row-transpose-scalar-prod-as-sum*)
show $i < dim\text{-}col\ ?B$ **using** *i* **by** *simp*
show $dim\text{-}vec\ H = dim\text{-}row\ ?B$ **by** *simp*
qed
also have ... = $(\sum j<degree\ u.\ H\ \$\ j * ?B\ \$\$\ (j,\ i))$ **by** (*rule sum.cong*, *auto*)
also have ... = $H\ \$\ i$ **using** *coeff-eq-sum'*[*rule-format*, *symmetric*, *of i*] *i* **by** *simp*
finally show $(transpose\text{-}mat\ ?B *_v H)\ \$\ i = H\ \$\ i$ **.**
qed
qed
qed

end

context
assumes *SORT-CONSTRAINT*(*'a*::*prime-card*)
begin

lemma *exists-s-factor-dvd-h-s*:
fixes *fi*::*'a mod-ring poly*
assumes *finite-P*: *finite P*
and *f-desc-square-free*: $f = (\prod a{\in}P.\ a)$
and *P*: $P \subseteq \{q.\ irreducible\ q \wedge monic\ q\}$
and *fi-P*: $fi \in P$
and *h*: $h \in \{v.\ [v\widehat{\ }(CARD('a)) = v]\ (mod\ f)\}$
shows $\exists\, s.\ fi\ dvd\ (h - [:s:])$
proof −
let $?p = CARD('a)$
have *f-dvd-hqh*: $f\ dvd\ (h\widehat{\ }?p - h)$ **using** *h* **unfolding** *cong-def*
using *mod-eq-dvd-iff-poly* **by** *blast*
also have *hq-h-rw*: ... = $prod\ (\lambda c.\ h - [:c:])\ (UNIV::'a\ mod\text{-}ring\ set)$
by (*rule poly-identity-mod-p*)
finally have *f-dvd-hc*: $f\ dvd\ prod\ (\lambda c.\ h - [:c:])\ (UNIV::'a\ mod\text{-}ring\ set)$ **by** *simp*
have *fi dvd f* **using** *f-desc-square-free fi-P*
using *dvd-prod-eqI finite-P* **by** *blast*
hence $fi\ dvd\ (h\widehat{\ }?p - h)$ **using** *dvd-trans f-dvd-hqh* **by** *auto*
also have ... = $prod\ (\lambda c.\ h - [:c:])\ (UNIV::'a\ mod\text{-}ring\ set)$ **unfolding** *hq-h-rw* **by** *simp*
finally have *fi-dvd-prod-hc*: $fi\ dvd\ prod\ (\lambda c.\ h - [:c:])\ (UNIV::'a\ mod\text{-}ring\ set)$ **.**
have *irr-fi*: *irreducible fi* **using** *fi-P P* **by** *blast*
have *fi-not-unit*: ¬ *is-unit fi* **using** *irr-fi* **by** (*simp add*: $irreducible_dD(1)$ *poly-dvd-1*)
show *?thesis* **using** *irreducible-dvd-prod*[*OF - fi-dvd-prod-hc*] *irr-fi* **by** *auto*
qed

corollary *exists-unique-s-factor-dvd-h-s*:
fixes *fi*::'*a mod-ring poly*
assumes *finite-P*: *finite P*
and *f-desc-square-free*: $f = (\prod a{\in}P.\ a)$
and *P*: $P \subseteq \{q.\ irreducible\ q \wedge monic\ q\}$
and *fi-P*: $fi \in P$
and *h*: $h \in \{v.\ [v\widehat{\ }(CARD('a)) = v]\ (mod\ f)\}$
shows $\exists!s.\ fi\ dvd\ (h - [:s:])$
proof −
obtain *c* **where** *fi-dvd*: *fi dvd* $(h - [:c:])$ **using** *assms exists-s-factor-dvd-h-s* **by** *blast*
have *irr-fi*: *irreducible fi* **using** *fi-P P* **by** *blast*
have *fi-not-unit*: ¬ *is-unit fi*
by (*simp add*: *irr-fi* $irreducible_d D(1)$ *poly-dvd-1*)
show *?thesis*
proof (*rule ex1I*[*of - c*], *auto simp add*: *fi-dvd*)
fix *c2* **assume** *fi-dvd-hc2*: *fi dvd* $h - [:c2:]$
have ∗: *fi dvd* $(h - [:c:]) * (h - [:c2:])$ **using** *fi-dvd* **by** *auto*
hence *fi dvd* $(h - [:c:]) \vee$ *fi dvd* $(h - [:c2:])$
using *irr-fi* **by** *auto*
thus $c2 = c$
using *coprime-h-c-poly coprime-not-unit-not-dvd fi-dvd fi-dvd-hc2 fi-not-unit* **by** *blast*
qed
qed

lemma *exists-two-distint*: $\exists\ a\ b$::'*a mod-ring*. $a \neq b$
by (*rule exI*[*of - 0*], *rule exI*[*of - 1*], *auto*)

lemma *coprime-cong-mult-factorization-poly*:
fixes *f*::'*b*::{*field*} *poly*
and *a b p* :: '*c* :: {*field-gcd*} *poly*
assumes *finite-P*: *finite P*
and *P*: $P \subseteq \{q.\ irreducible\ q\}$
and *p*: $\forall p{\in}P.\ [a{=}b]\ (mod\ p)$
and *coprime-P*: $\forall p1\ p2.\ p1 \in P \wedge p2 \in P \wedge p1 \neq p2 \longrightarrow coprime\ p1\ p2$
shows $[a = b]\ (mod\ (\prod a{\in}P.\ a))$
using *finite-P P p coprime-P*
proof (*induct P*)
case *empty*
thus *?case* **by** *simp*
next
case (*insert p P*)
have *ab-mod-pP*: $[a{=}b]\ (mod\ (p * \prod P))$
proof (*rule coprime-cong-mult-poly*)

show $[a = b]$ $(mod\ p)$ **using** *insert.prems* **by** *auto*
show $[a = b]$ $(mod\ \prod P)$ **using** *insert.prems insert.hyps* **by** *auto*
from *insert* **show** *Rings.coprime p* $(\prod P)$
by (*auto intro*: *prod-coprime-right*)
qed
thus *?case* **by** (*simp add*: *insert.hyps*(*1*) *insert.hyps*(*2*))
qed

end

context
assumes *SORT-CONSTRAINT*(*'a*::*prime-card*)
begin

lemma *W-eq-berlekamp-mat*:
fixes *u*::*'a mod-ring poly*
shows $\{v.\ [v \hat{} CARD('a) = v]\ (mod\ u) \wedge degree\ v < degree\ u\}$
$= \{h.\ let\ H = vec\text{-}of\text{-}list\ ((coeffs\ h)\ @\ replicate\ (degree\ u - length\ (coeffs\ h))\ 0)$ *in*
$(transpose\text{-}mat\ (berlekamp\text{-}mat\ u)) *_v H = H \wedge degree\ h < degree\ u\}$
using *equation-13* **by** (*auto simp add*: *Let-def*)

lemma *transpose-minus-1*:
assumes $dim\text{-}row\ Q = dim\text{-}col\ Q$
shows $transpose\text{-}mat\ (Q - (1_m\ (dim\text{-}row\ Q))) = (transpose\text{-}mat\ Q - (1_m\ (dim\text{-}row\ Q)))$
using *assms*
unfolding *mat-eq-iff* **by** *auto*

lemma *system-iff*:
fixes *v*::*'b*::*comm-ring-1 vec*
assumes *sq-Q*: $dim\text{-}row\ Q = dim\text{-}col\ Q$ **and** *v*: $dim\text{-}row\ Q = dim\text{-}vec\ v$
shows $(transpose\text{-}mat\ Q *_v v = v) \longleftrightarrow ((transpose\text{-}mat\ Q - 1_m\ (dim\text{-}row\ Q)) *_v v = 0_v\ (dim\text{-}vec\ v))$
proof –
have *t1*: $transpose\text{-}mat\ Q *_v v - v = 0_v\ (dim\text{-}vec\ v) \Longrightarrow (transpose\text{-}mat\ Q - 1_m\ (dim\text{-}row\ Q)) *_v v = 0_v\ (dim\text{-}vec\ v)$
by (*subst minus-mult-distrib-mat-vec*, *insert sq-Q*[*symmetric*] *v*, *auto*)
have *t2*: $(transpose\text{-}mat\ Q - 1_m\ (dim\text{-}row\ Q)) *_v v = 0_v\ (dim\text{-}vec\ v) \Longrightarrow transpose\text{-}mat\ Q *_v v - v = 0_v\ (dim\text{-}vec\ v)$
by (*subst* (*asm*) *minus-mult-distrib-mat-vec*, *insert sq-Q*[*symmetric*] *v*, *auto*)
have $transpose\text{-}mat\ Q *_v v - v = v - v \Longrightarrow transpose\text{-}mat\ Q *_v v = v$
proof –
assume *a1*: $transpose\text{-}mat\ Q *_v v - v = v - v$
have *f2*: $transpose\text{-}mat\ Q *_v v \in carrier\text{-}vec\ (dim\text{-}vec\ v)$
by (*metis dim-mult-mat-vec index-transpose-mat*(*2*) *sq-Q v carrier-vec-dim-vec*)
then have *f3*: $0_v\ (dim\text{-}vec\ v) + transpose\text{-}mat\ Q *_v v = transpose\text{-}mat\ Q *_v v$

by (*meson left-zero-vec*)
have *f4*: 0_v (*dim-vec v*) = *transpose-mat Q* $*_v$ *v* − *v*
using *a1* **by** *auto*
have *f5*: − *v* ∈ *carrier-vec* (*dim-vec v*)
by *simp*
then have *f6*: − *v* + *transpose-mat Q* $*_v$ *v* = *v* − *v*
using *f2 a1* **using** *comm-add-vec minus-add-uminus-vec* **by** *fastforce*
have *v* − *v* = − *v* + *v* **by** *auto*
then have *transpose-mat Q* $*_v$ *v* = *transpose-mat Q* $*_v$ *v* − *v* + *v*
using *f6 f4 f3 f2* **by** (*metis* (*no-types*, *lifting*) *a1 assoc-add-vec comm-add-vec f5 carrier-vec-dim-vec*)
then show *?thesis*
using *a1* **by** *auto*
qed
hence (*transpose-mat Q* $*_v$ *v* = *v*) = ((*transpose-mat Q* $*_v$ *v*) − *v* = *v* − *v*) **by** *auto*
also have ... = ((*transpose-mat Q* $*_v$ *v*) − *v* = 0_v (*dim-vec v*)) **by** *auto*
also have ... = ((*transpose-mat Q* − 1_m (*dim-row Q*)) $*_v$ *v* = 0_v (*dim-vec v*))
using *t1 t2* **by** *auto*
finally show *?thesis***.**
qed

lemma *system-if-mat-kernel*:
assumes *sq-Q*: *dim-row Q* = *dim-col Q* **and** *v*: *dim-row Q* = *dim-vec v*
shows (*transpose-mat Q* $*_v$ *v* = *v*) ⟷ *v* ∈ *mat-kernel* (*transpose-mat* (*Q* − (1_m (*dim-row Q*))))
proof −
have (*transpose-mat Q* $*_v$ *v* = *v*) = ((*transpose-mat Q* − 1_m (*dim-row Q*)) $*_v$ *v* = 0_v (*dim-vec v*))
using *assms system-iff* **by** *blast*
also have ... = (*v* ∈ *mat-kernel* (*transpose-mat* (*Q* − (1_m (*dim-row Q*)))))
unfolding *mat-kernel-def* **unfolding** *transpose-minus-1*[*OF sq-Q*] **unfolding** *v* **by** *auto*
finally show *?thesis* **.**
qed

lemma *degree-u-mod-irreducible*$_d$*-factor-0*:
fixes *v* **and** *u*::′*a mod-ring poly*
defines *W*: *W* ≡ {*v*. [*v* ^ *CARD*(′*a*) = *v*] (*mod u*)}
assumes *v*: *v* ∈ *W*
and *finite-U*: *finite U* **and** *u-U*: *u* = ∏ *U* **and** *U-irr-monic*: *U* ⊆ {*q*. *irreducible q* ∧ *monic q*}
and *fi-U*: *fi* ∈ *U*
shows *degree* (*v mod fi*) = *0*
proof −
have *deg-fi*: *degree fi* > *0*

using *U-irr-monic*
using *fi-U irreducible*$_d$*D*[*of fi*] **by** *auto*
have *fi dvd u*
using *u-U U-irr-monic finite-U dvd-prod-eqI fi-U* **by** *blast*
moreover have *u dvd* (v^*CARD*('*a*) − *v*)
using *v* **unfolding** *W cong-def*
by (*simp add*: *mod-eq-dvd-iff-poly*)
ultimately have *fi dvd* (v^*CARD*('*a*) − *v*)
by (*rule dvd-trans*)
then have *fi-dvd-prod-vc*: *fi dvd prod* (λ*c*. *v* − [:*c*:]) (*UNIV*::'*a mod-ring set*)
by (*simp add*: *poly-identity-mod-p*)
have *irr-fi*: *irreducible fi* **using** *fi-U U-irr-monic* **by** *blast*
have *fi-not-unit*: ¬ *is-unit fi*
using *irr-fi*
by (*auto simp*: *poly-dvd-1*)
have *fi-dvd-vc*: ∃ *c*. *fi dvd v* − [:*c*:]
using *irreducible-dvd-prod*[*OF - fi-dvd-prod-vc*] *irr-fi* **by** *auto*
from *this* **obtain** *a* **where** *fi dvd v* − [:*a*:] **by** *blast*
hence *v mod fi* = [:*a*:] *mod fi* **using** *mod-eq-dvd-iff-poly* **by** *blast*
also have ... = [:*a*:] **by** (*simp add*: *deg-fi mod-poly-less*)
finally show *?thesis* **by** *simp*
qed

definition *poly-abelian-monoid*
= (|*carrier* = *UNIV*::'*a mod-ring poly set*, *monoid.mult* = ((∗)), *one* = *1*, *zero* = *0*, *add* = (+), *module.smult* = *smult*|)

interpretation *vector-space-poly*: *vectorspace class-ring poly-abelian-monoid*
rewrites [*simp*]: $\mathbf{0}_{poly\text{-}abelian\text{-}monoid}$ = *0*
and [*simp*]: $\mathbf{1}_{poly\text{-}abelian\text{-}monoid}$ = *1*
and [*simp*]: ($\oplus_{poly\text{-}abelian\text{-}monoid}$) = (+)
and [*simp*]: ($\otimes_{poly\text{-}abelian\text{-}monoid}$) = (∗)
and [*simp*]: *carrier poly-abelian-monoid* = *UNIV*
and [*simp*]: ($\odot_{poly\text{-}abelian\text{-}monoid}$) = *smult*
apply *unfold-locales*
apply (*auto simp*: *poly-abelian-monoid-def class-field-def smult-add-left smult-add-right Units-def*)
by (*metis add.commute add.right-inverse*)

lemma *subspace-Berlekamp*:
assumes *f*: *degree f* ≠ *0*
shows *subspace* (*class-ring* :: '*a mod-ring ring*)
{*v*. [*v*^(*CARD*('*a*)) = *v*] (*mod f*) ∧ (*degree v* < *degree f*)} *poly-abelian-monoid*
proof −
{ **fix** *v* :: '*a mod-ring poly* **and** *w* :: '*a mod-ring poly*
assume *a1*: *v* ^ *card* (*UNIV*::'*a set*) *mod f* = *v mod f*

assume *w* ^ *card* (*UNIV*::*'a set*) *mod f* = *w mod f*
then have (*v* ^ *card* (*UNIV*::*'a set*) + *w* ^ *card* (*UNIV*::*'a set*)) *mod f* = (*v* + *w*) *mod f*
using *a1* **by** (*meson mod-add-cong*)
then have (*v* + *w*) ^ *card* (*UNIV*::*'a set*) *mod f* = (*v* + *w*) *mod f*
by (*simp add*: *add-power-poly-mod-ring*)
} **note** *r*=*this*
thus *?thesis* **using** *f*
by (*unfold-locales*, *auto simp*: *zero-power mod-smult-left smult-power cong-def degree-add-less*)
qed

lemma *berlekamp-resulting-mat-closed*[*simp*]:
berlekamp-resulting-mat u ∈ *carrier-mat* (*degree u*) (*degree u*)
dim-row (*berlekamp-resulting-mat u*) = *degree u*
dim-col (*berlekamp-resulting-mat u*) = *degree u*
proof −
let *?A*=(*transpose-mat* (*mat* (*degree u*) (*degree u*)
(λ(*i*, *j*). *if i* = *j then berlekamp-mat u* $$ (*i*, *j*) − *1 else berlekamp-mat u* $$ (*i*, *j*))))
let *?G*=(*gauss-jordan-single ?A*)
have *?G* ∈*carrier-mat* (*degree u*) (*degree u*)
by (*rule gauss-jordan-single*(*2*)[*of ?A*], *auto*)
thus
berlekamp-resulting-mat u ∈ *carrier-mat* (*degree u*) (*degree u*)
dim-row (*berlekamp-resulting-mat u*) = *degree u*
dim-col (*berlekamp-resulting-mat u*) = *degree u*
unfolding *berlekamp-resulting-mat-def Let-def* **by** *auto*
qed

lemma *berlekamp-resulting-mat-basis*:
kernel.basis (*degree u*) (*berlekamp-resulting-mat u*) (*set* (*find-base-vectors* (*berlekamp-resulting-mat u*)))
proof (*rule find-base-vectors*(*3*))
show *berlekamp-resulting-mat u* ∈ *carrier-mat* (*degree u*) (*degree u*) **by** *simp*
let *?A*=(*transpose-mat* (*mat* (*degree u*) (*degree u*)
(λ(*i*, *j*). *if i* = *j then berlekamp-mat u* $$ (*i*, *j*) − *1 else berlekamp-mat u* $$ (*i*, *j*))))
have *row-echelon-form* (*gauss-jordan-single ?A*)
by (*rule gauss-jordan-single*(*3*)[*of ?A*], *auto*)
thus *row-echelon-form* (*berlekamp-resulting-mat u*)
unfolding *berlekamp-resulting-mat-def Let-def* **by** *auto*
qed

lemma *set-berlekamp-basis-eq*: (*set* (*berlekamp-basis u*))

$= (Poly \circ list\text{-}of\text{-}vec)$ ‘ (*set* (*find-base-vectors* (*berlekamp-resulting-mat u*)))
by (*auto simp add*: *image-def o-def berlekamp-basis-def*)

lemma *berlekamp-resulting-mat-constant*:
assumes *deg-u*: *degree u* = *0*
shows *berlekamp-resulting-mat u* = 1_m *0*
by (*unfold mat-eq-iff*, *auto simp add*: *deg-u*)

context
fixes *u*::′*a*::*prime-card mod-ring poly*
begin

lemma *set-berlekamp-basis-constant*:
assumes *deg-u*: *degree u* = *0*
shows *set* (*berlekamp-basis u*) = {}
proof −
have *one-carrier*: 1_m *0* ∈ *carrier-mat 0 0* **by** *auto*
have *m*: *mat-kernel* (1_m *0*) = {(0_v *0*) :: ′*a mod-ring vec*} **unfolding** *mat-kernel-def*
by *auto*
have *r*: *row-echelon-form* (1_m *0* :: ′*a mod-ring mat*)
unfolding *row-echelon-form-def pivot-fun-def Let-def* **by** *auto*
have *set* (*find-base-vectors* (1_m *0*)) ⊆ {0_v *0* :: ′*a mod-ring vec*}
using *find-base-vectors*(*1*)[*OF r one-carrier*] **unfolding** *m* **.**
hence *set* (*find-base-vectors* (1_m *0*) :: ′*a mod-ring vec list*) = {}
using *find-base-vectors*(*2*)[*OF r one-carrier*]
using *subset-singletonD* **by** *fastforce*
thus *?thesis*
unfolding *set-berlekamp-basis-eq* **unfolding** *berlekamp-resulting-mat-constant*[*OF deg-u*] **by** *auto*
qed

lemma *row-echelon-form-berlekamp-resulting-mat*: *row-echelon-form* (*berlekamp-resulting-mat u*)
by (*rule gauss-jordan-single*(*3*), *auto simp add*: *berlekamp-resulting-mat-def Let-def*)

lemma *mat-kernel-berlekamp-resulting-mat-degree-0*:
assumes *d*: *degree u* = *0*
shows *mat-kernel* (*berlekamp-resulting-mat u*) = {0_v *0*}
by (*auto simp add*: *mat-kernel-def mult-mat-vec-def d*)

lemma *in-mat-kernel-berlekamp-resulting-mat*:
assumes *x*: *transpose-mat* (*berlekamp-mat u*) $*_v$ *x* = *x*
and *x-dim*: *x* ∈ *carrier-vec* (*degree u*)
shows *x* ∈ *mat-kernel* (*berlekamp-resulting-mat u*)
proof −
let *?QI*=(*mat*(*dim-row* (*berlekamp-mat u*)) (*dim-row* (*berlekamp-mat u*))

(λ(*i*, *j*). *if i* = *j then berlekamp-mat u* $$ (*i*, *j*) − *1 else berlekamp-mat u* $$ (*i*, *j*)))
have ∗: (*transpose-mat* (*berlekamp-mat u*) − 1_m (*degree u*)) = *transpose-mat ?QI* **by** *auto*
have (*transpose-mat* (*berlekamp-mat u*) − 1_m (*dim-row* (*berlekamp-mat u*))) $*_v$ *x* = 0_v (*dim-vec x*)
using *system-iff*[*of berlekamp-mat u x*] *x-dim x* **by** *auto*
hence *transpose-mat ?QI* $*_v$ *x* = 0_v (*degree u*) **using** *x-dim* ∗ **by** *auto*
hence *berlekamp-resulting-mat u* $*_v$ *x* = 0_v (*degree u*)
unfolding *berlekamp-resulting-mat-def Let-def*
using *gauss-jordan-single*(*1*)[*of transpose-mat ?QI degree u degree u - x*] *x-dim* **by** *auto*
thus *?thesis* **by** (*auto simp add*: *mat-kernel-def x-dim*)
qed

private abbreviation *V* ≡ *kernel.VK* (*degree u*) (*berlekamp-resulting-mat u*)
private abbreviation *W* ≡ *vector-space-poly.vs*
{*v*. [*v*^(*CARD*(*'a*)) = *v*] (*mod u*) ∧ (*degree v* < *degree u*)}

interpretation *V*: *vectorspace class-ring V*
proof −
interpret *k*: *kernel* (*degree u*) (*degree u*) (*berlekamp-resulting-mat u*)
by (*unfold-locales*; *auto*)
show *vectorspace class-ring V* **by** *intro-locales*
qed

lemma *linear-Poly-list-of-vec*:
shows (*Poly* ∘ *list-of-vec*) ∈ *module-hom class-ring V* (*vector-space-poly.vs* {*v*. [*v*^(*CARD*(*'a*)) = *v*] (*mod u*)})
proof (*auto simp add*: *LinearCombinations.module-hom-def Matrix.module-vec-def*)
fix *m1 m2*:: *'a mod-ring vec*
assume *m1*: *m1* ∈ *mat-kernel* (*berlekamp-resulting-mat u*)
and *m2*: *m2* ∈ *mat-kernel* (*berlekamp-resulting-mat u*)
have *m1-rw*: *list-of-vec m1* = *map* (λ*n*. *m1* $ *n*) [*0*..<*dim-vec m1*]
by (*transfer*, *auto simp add*: *mk-vec-def*)
have *m2-rw*: *list-of-vec m2* = *map* (λ*n*. *m2* $ *n*) [*0*..<*dim-vec m2*]
by (*transfer*, *auto simp add*: *mk-vec-def*)
have *m1* ∈ *carrier-vec* (*degree u*) **by** (*rule mat-kernelD*(*1*)[*OF - m1*], *auto*)
moreover have *m2* ∈ *carrier-vec* (*degree u*) **by** (*rule mat-kernelD*(*1*)[*OF - m2*], *auto*)
ultimately have *dim-eq*: *dim-vec m1* = *dim-vec m2* **by** *auto*
show *Poly* (*list-of-vec* (*m1* + *m2*)) = *Poly* (*list-of-vec m1*) + *Poly* (*list-of-vec m2*)
unfolding *poly-eq-iff m1-rw m2-rw plus-vec-def*
using *dim-eq*
by (*transfer'*, *auto simp add*: *mk-vec-def nth-default-def*)
next
fix *r m* **assume** *m*: *m* ∈ *mat-kernel* (*berlekamp-resulting-mat u*)
show *Poly* (*list-of-vec* (*r* $\cdot_v$ *m*)) = *smult r* (*Poly* (*list-of-vec m*))

unfolding *poly-eq-iff list-of-vec-rw-map*[*of m*] *smult-vec-def*
by (*transfer′*, *auto simp add*: *mk-vec-def nth-default-def*)
next
fix x **assume** x: $x \in$ *mat-kernel* (*berlekamp-resulting-mat u*)
show [*Poly* (*list-of-vec x*) ^ *CARD*(′*a*) = *Poly* (*list-of-vec x*)] (*mod u*)
proof (*cases degree u* = *0*)
case *True*
have *mat-kernel* (*berlekamp-resulting-mat u*) = $\{0_v\ 0\}$
by (*rule mat-kernel-berlekamp-resulting-mat-degree-0*[*OF True*])
hence *x-0*: $x = 0_v\ 0$ **using** x **by** *blast*
show *?thesis* **by** (*auto simp add*: *zero-power x-0 cong-def*)
next
case *False* **note** *deg-u* = *False*
show *?thesis*
proof −
let *?QI*=(*mat* (*degree u*) (*degree u*)
(λ(*i*, *j*). *if i* = *j then berlekamp-mat u* \$\$ (*i*, *j*) − *1 else berlekamp-mat u* \$\$ (*i*, *j*)))
let *?H*=*vec-of-list* (*coeffs* (*Poly* (*list-of-vec x*)) @ *replicate* (*degree u* − *length* (*coeffs* (*Poly* (*list-of-vec x*)))) *0*)
have *x-dim*: *dim-vec x* = *degree u* **using** x **unfolding** *mat-kernel-def* **by** *auto*
hence *x-carrier*[*simp*]: $x \in$ *carrier-vec* (*degree u*) **by** (*metis carrier-vec-dim-vec*)
have *x-kernel*: *berlekamp-resulting-mat u* $*_v$ *x* = 0_v (*degree u*)
using x **unfolding** *mat-kernel-def* **by** *auto*
have *t-QI-x-0*: (*transpose-mat ?QI*) $*_v$ *x* = 0_v (*degree u*)
using *gauss-jordan-single*(*1*)[*of* (*transpose-mat ?QI*) *degree u degree u gauss-jordan-single* (*transpose-mat ?QI*) *x*]
using *x-kernel* **unfolding** *berlekamp-resulting-mat-def Let-def* **by** *auto*
have *l*: (*list-of-vec x*) $\neq$ []
by (*auto simp add*: *list-of-vec-rw-map vec-of-dim-0*[*symmetric*] *deg-u x-dim*)
have *deg-le*: *degree* (*Poly* (*list-of-vec x*)) < *degree u*
using *degree-Poly-list-of-vec*
using *x-carrier deg-u* **by** *blast*
show [*Poly* (*list-of-vec x*) ^ *CARD*(′*a*) = *Poly* (*list-of-vec x*)] (*mod u*)
proof (*unfold equation-13*[*OF deg-le*])
have *QR-rw*: *?QI* = *berlekamp-mat u* − 1_m (*dim-row* (*berlekamp-mat u*)) **by** *auto*
have *dim-row* (*berlekamp-mat u*) = *dim-vec ?H*
by (*auto*, *metis le-add-diff-inverse length-list-of-vec length-strip-while-le x-dim*)
moreover have *?H* $\in$ *mat-kernel* (*transpose-mat* (*berlekamp-mat u* − 1_m (*dim-row* (*berlekamp-mat u*))))
proof −
have *Hx*: *?H* = *x*
proof (*unfold vec-eq-iff*, *auto*)
let *?H′*=*vec-of-list* (*strip-while* ((=) *0*) (*list-of-vec x*)
@ *replicate* (*degree u* − *length* (*strip-while* ((=) *0*) (*list-of-vec x*))) *0*)
show *length* (*strip-while* ((=) *0*) (*list-of-vec x*))
+ (*degree u* − *length* (*strip-while* ((=) *0*) (*list-of-vec x*))) = *dim-vec x*

by (*metis le-add-diff-inverse length-list-of-vec length-strip-while-le x-dim*)
fix *i* **assume** *i*: $i < dim\text{-}vec\ x$
have *?H* $ *i* = *coeff* (*Poly* (*list-of-vec x*)) *i*
proof (*rule vec-of-list-coeffs-replicate-nth*[*OF - deg-le*])
show $i \in \{..<degree\ u\}$ **using** *x-dim i* **by** (*auto*, *linarith*)
qed
also have ... = *x* $ *i* **by** (*rule coeff-Poly-list-of-vec-nth'*[*OF i*])
finally show *?H'* $ *i* = *x* $ *i* **by** *auto*
qed
have $?H \in carrier\text{-}vec\ (degree\ u)$ **using** *deg-le dim-vec-of-list-h Hx* **by** *auto*
moreover have *transpose-mat* (*berlekamp-mat u* $- 1_m$ (*degree u*)) $*_v$ *?H* $= 0_v$ (*degree u*)
using *t-QI-x-0 Hx QR-rw* **by** *auto*
ultimately show *?thesis*
by (*auto simp add*: *mat-kernel-def*)
qed
ultimately show *transpose-mat* (*berlekamp-mat u*) $*_v$ *?H* = *?H*
using *system-if-mat-kernel*[*of berlekamp-mat u ?H*]
by *auto*
qed
qed
qed
qed

lemma *linear-Poly-list-of-vec'*:
assumes *degree u* > *0*
shows (*Poly* ∘ *list-of-vec*) ∈ *module-hom R V W*
proof (*auto simp add*: *LinearCombinations.module-hom-def Matrix.module-vec-def*)
fix *m1 m2*:: *'a mod-ring vec*
assume *m1*: *m1* ∈ *mat-kernel* (*berlekamp-resulting-mat u*)
and *m2*: *m2* ∈ *mat-kernel* (*berlekamp-resulting-mat u*)
have *m1-rw*: *list-of-vec m1* = *map* (λ*n*. *m1* $ *n*) [*0*..<*dim-vec m1*]
by (*transfer*, *auto simp add*: *mk-vec-def*)
have *m2-rw*: *list-of-vec m2* = *map* (λ*n*. *m2* $ *n*) [*0*..<*dim-vec m2*]
by (*transfer*, *auto simp add*: *mk-vec-def*)
have *m1* ∈ *carrier-vec* (*degree u*) **by** (*rule mat-kernelD*(*1*)[*OF - m1*], *auto*)
moreover have *m2* ∈ *carrier-vec* (*degree u*) **by** (*rule mat-kernelD*(*1*)[*OF - m2*], *auto*)
ultimately have *dim-eq*: *dim-vec m1* = *dim-vec m2* **by** *auto*
show *Poly* (*list-of-vec* (*m1* + *m2*)) = *Poly* (*list-of-vec m1*) + *Poly* (*list-of-vec m2*)
unfolding *poly-eq-iff m1-rw m2-rw plus-vec-def*
using *dim-eq*
by (*transfer'*, *auto simp add*: *mk-vec-def nth-default-def*)
next
fix *r m* **assume** *m*: *m* ∈ *mat-kernel* (*berlekamp-resulting-mat u*)

show *Poly* (*list-of-vec* ($r \cdot_v m$)) = *smult r* (*Poly* (*list-of-vec m*))
unfolding *poly-eq-iff list-of-vec-rw-map*[*of m*] *smult-vec-def*
by (*transfer'*, *auto simp add*: *mk-vec-def nth-default-def*)
next
fix *x* **assume** *x*: $x \in$ *mat-kernel* (*berlekamp-resulting-mat u*)
show [*Poly* (*list-of-vec x*) ^ *CARD*('*a*) = *Poly* (*list-of-vec x*)] (*mod u*)
proof (*cases degree u* = *0*)
case *True*
have *mat-kernel* (*berlekamp-resulting-mat u*) = {0_v *0*}
by (*rule mat-kernel-berlekamp-resulting-mat-degree-0*[*OF True*])
hence *x-0*: $x = 0_v$ *0* **using** *x* **by** *blast*
show *?thesis* **by** (*auto simp add*: *zero-power x-0 cong-def*)
next
case *False* **note** *deg-u* = *False*
show *?thesis*
proof −
let *?QI*=(*mat* (*degree u*) (*degree u*)
(λ(*i*, *j*). *if i* = *j then berlekamp-mat u* $$ (*i*, *j*) − *1 else berlekamp-mat u* $$ (*i*, *j*)))
let *?H*=*vec-of-list* (*coeffs* (*Poly* (*list-of-vec x*)) @ *replicate* (*degree u* − *length* (*coeffs* (*Poly* (*list-of-vec x*)))) *0*)
have *x-dim*: *dim-vec x* = *degree u* **using** *x* **unfolding** *mat-kernel-def* **by** *auto*
hence *x-carrier*[*simp*]: $x \in$ *carrier-vec* (*degree u*) **by** (*metis carrier-vec-dim-vec*)
have *x-kernel*: *berlekamp-resulting-mat u* $*_v$ *x* = 0_v (*degree u*)
using *x* **unfolding** *mat-kernel-def* **by** *auto*
have *t-QI-x-0*: (*transpose-mat ?QI*) $*_v$ *x* = 0_v (*degree u*)
using *gauss-jordan-single*(*1*)[*of* (*transpose-mat ?QI*) *degree u degree u gauss-jordan-single* (*transpose-mat ?QI*) *x*]
using *x-kernel* **unfolding** *berlekamp-resulting-mat-def Let-def* **by** *auto*
have *l*: (*list-of-vec x*) ≠ []
by (*auto simp add*: *list-of-vec-rw-map vec-of-dim-0*[*symmetric*] *deg-u x-dim*)
have *deg-le*: *degree* (*Poly* (*list-of-vec x*)) < *degree u*
using *degree-Poly-list-of-vec*
using *x-carrier deg-u* **by** *blast*
show [*Poly* (*list-of-vec x*) ^ *CARD*('*a*) = *Poly* (*list-of-vec x*)] (*mod u*)
proof (*unfold equation-13*[*OF deg-le*])
have *QR-rw*: *?QI* = *berlekamp-mat u* − 1_m (*dim-row* (*berlekamp-mat u*)) **by** *auto*
have *dim-row* (*berlekamp-mat u*) = *dim-vec ?H*
by (*auto*, *metis le-add-diff-inverse length-list-of-vec length-strip-while-le x-dim*)
moreover have *?H* ∈ *mat-kernel* (*transpose-mat* (*berlekamp-mat u* − 1_m (*dim-row* (*berlekamp-mat u*))))
proof −
have *Hx*: *?H* = *x*
proof (*unfold vec-eq-iff*, *auto*)
let *?H'*=*vec-of-list* (*strip-while* ((=) *0*) (*list-of-vec x*)
@ *replicate* (*degree u* − *length* (*strip-while* ((=) *0*) (*list-of-vec x*))) *0*)
show *length* (*strip-while* ((=) *0*) (*list-of-vec x*))

+ (*degree u* − *length* (*strip-while* ((=) *0*) (*list-of-vec x*))) = *dim-vec x*
by (*metis le-add-diff-inverse length-list-of-vec length-strip-while-le x-dim*)
fix *i* **assume** *i*: *i* < *dim-vec x*
have *?H* \$ *i* = *coeff* (*Poly* (*list-of-vec x*)) *i*
proof (*rule vec-of-list-coeffs-replicate-nth*[*OF* - *deg-le*])
show *i* ∈ {..<*degree u*} **using** *x-dim i* **by** (*auto*, *linarith*)
qed
also have ... = *x* \$ *i* **by** (*rule coeff-Poly-list-of-vec-nth′*[*OF i*])
finally show *?H′* \$ *i* = *x* \$ *i* **by** *auto*
qed
have *?H* ∈ *carrier-vec* (*degree u*) **using** *deg-le dim-vec-of-list-h Hx* **by** *auto*
moreover have *transpose-mat* (*berlekamp-mat u* − 1_m (*degree u*)) $*_v$ *?H* = 0_v (*degree u*)
using *t-QI-x-0 Hx QR-rw* **by** *auto*
ultimately show *?thesis*
by (*auto simp add*: *mat-kernel-def*)
qed
ultimately show *transpose-mat* (*berlekamp-mat u*) $*_v$ *?H* = *?H*
using *system-if-mat-kernel*[*of berlekamp-mat u ?H*]
by *auto*
qed
qed
qed
next
fix *x* **assume** *x*: *x* ∈ *mat-kernel* (*berlekamp-resulting-mat u*)
show *degree* (*Poly* (*list-of-vec x*)) < *degree u*
by (*rule degree-Poly-list-of-vec*, *insert assms x*, *auto simp*: *mat-kernel-def*)
qed

lemma *berlekamp-basis-eq-8*:
assumes *v*: *v* ∈ *set* (*berlekamp-basis u*)
shows [*v* ^ *CARD*(′*a*) = *v*] (*mod u*)
proof −
{
fix *x* **assume** *x*: *x* ∈ *set* (*find-base-vectors* (*berlekamp-resulting-mat u*))
have *set* (*find-base-vectors* (*berlekamp-resulting-mat u*)) ⊆ *mat-kernel* (*berlekamp-resulting-mat u*)
proof (*rule find-base-vectors*(*1*))
show *row-echelon-form* (*berlekamp-resulting-mat u*)
by (*rule row-echelon-form-berlekamp-resulting-mat*)
show *berlekamp-resulting-mat u* ∈ *carrier-mat* (*degree u*) (*degree u*) **by** *simp*
qed
hence *x* ∈ *mat-kernel* (*berlekamp-resulting-mat u*) **using** *x* **by** *auto*
hence [*Poly* (*list-of-vec x*) ^ *CARD*(′*a*) = *Poly* (*list-of-vec x*)] (*mod u*)
using *linear-Poly-list-of-vec*
unfolding *LinearCombinations.module-hom-def Matrix.module-vec-def* **by**

auto
}
thus [v ^ $CARD('a)$ = v] (*mod u*) **using** *v* **unfolding** *set-berlekamp-basis-eq* **by** *auto*
qed

lemma *surj-Poly-list-of-vec*:
assumes *deg-u*: *degree u* > *0*
shows (*Poly* ∘ *list-of-vec*)‘ (*carrier V*) = *carrier W*
proof (*auto simp add*: *image-def*)
fix *xa*
assume *xa*: *xa* ∈ *mat-kernel* (*berlekamp-resulting-mat u*)
thus [*Poly* (*list-of-vec xa*) ^ $CARD('a)$ = *Poly* (*list-of-vec xa*)] (*mod u*)
using *linear-Poly-list-of-vec*
unfolding *LinearCombinations.module-hom-def Matrix.module-vec-def* **by** *auto*
show *degree* (*Poly* (*list-of-vec xa*)) < *degree u*
proof (*rule degree-Poly-list-of-vec*[*OF - deg-u*])
show *xa* ∈ *carrier-vec* (*degree u*) **using** *xa* **unfolding** *mat-kernel-def* **by** *simp*
qed
next
fix *x* **assume** *x*: [*x* ^ $CARD('a)$ = *x*] (*mod u*)
and *deg-x*: *degree x* < *degree u*
show ∃ *xa* ∈ *mat-kernel* (*berlekamp-resulting-mat u*). *x* = *Poly* (*list-of-vec xa*)
proof (*rule bexI*[*of - vec-of-list* (*coeffs x* @ *replicate* (*degree u* − *length* (*coeffs x*)) *0*)])
let *?X* = *vec-of-list* (*coeffs x* @ *replicate* (*degree u* − *length* (*coeffs x*)) *0*)
show *x* = *Poly* (*list-of-vec* (*vec-of-list* (*coeffs x* @ *replicate* (*degree u* − *length* (*coeffs x*)) *0*)))
by *auto*
have *X*: *?X* ∈ *carrier-vec* (*degree u*) **unfolding** *carrier-vec-def*
by (*auto*, *metis Suc-leI coeffs-0-eq-Nil deg-x degree-0 le-add-diff-inverse length-coeffs-degree linordered-semidom-class.add-diff-inverse list.size*(*3*) *order.asym*)
have *t*: *transpose-mat* (*berlekamp-mat u*) $*_v$ *?X* = *?X*
using *equation-13*[*OF deg-x*] *x* **by** *auto*
show *vec-of-list* (*coeffs x* @ *replicate* (*degree u* − *length* (*coeffs x*)) *0*)
∈ *mat-kernel* (*berlekamp-resulting-mat u*) **by** (*rule in-mat-kernel-berlekamp-resulting-mat*[*OF t X*])
qed
qed

lemma *card-set-berlekamp-basis*: *card* (*set* (*berlekamp-basis u*)) = *length* (*berlekamp-basis u*)
proof −
have *b*: *berlekamp-resulting-mat u* ∈ *carrier-mat* (*degree u*) (*degree u*) **by** *auto*
have (*set* (*berlekamp-basis u*)) = (*Poly* ∘ *list-of-vec*) ‘ *set* (*find-base-vectors* (*berlekamp-resulting-mat u*))

unfolding *set-berlekamp-basis-eq* **..**
also have *card* ... = *card* (*set* (*find-base-vectors* (*berlekamp-resulting-mat* *u*)))
proof (*rule* *card-image*, *rule* *subset-inj-on*[*OF* *inj-Poly-list-of-vec*])
show *set* (*find-base-vectors* (*berlekamp-resulting-mat* *u*)) ⊆ *carrier-vec* (*degree* *u*)
using *find-base-vectors*(*1*)[*OF* *row-echelon-form-berlekamp-resulting-mat* *b*]
unfolding *carrier-vec-def* *mat-kernel-def*
by *auto*
qed
also have ... = *length* (*find-base-vectors* (*berlekamp-resulting-mat* *u*))
by (*rule* *length-find-base-vectors*[*symmetric*, *OF* *row-echelon-form-berlekamp-resulting-mat* *b*])
finally show *?thesis* **unfolding** *berlekamp-basis-def* **by** *auto*
qed

context
assumes *deg-u0*[*simp*]: *degree* *u* > *0*
begin

interpretation *Berlekamp-subspace*: *vectorspace* *class-ring* *W*
by (*rule* *vector-space-poly.subspace-is-vs*[*OF* *subspace-Berlekamp*], *simp*)

lemma *linear-map-Poly-list-of-vec'*: *linear-map* *class-ring* *V* *W* (*Poly* ∘ *list-of-vec*)
proof (*auto* *simp* *add*: *linear-map-def*)
show *vectorspace* *class-ring* *V* **by** *intro-locales*
show *vectorspace* *class-ring* *W* **by** (*rule* *Berlekamp-subspace.vectorspace-axioms*)
show *mod-hom* *class-ring* *V* *W* (*Poly* ∘ *list-of-vec*)
proof (*rule* *mod-hom.intro*, *unfold* *mod-hom-axioms-def*)
show *module* *class-ring* *V* **by** *intro-locales*
show *module* *class-ring* *W* **using** *Berlekamp-subspace.vectorspace-axioms* **by** *intro-locales*
show *Poly* ∘ *list-of-vec* ∈ *module-hom* *class-ring* *V* *W*
by (*rule* *linear-Poly-list-of-vec'*[*OF* *deg-u0*])
qed
qed

lemma *berlekamp-basis-basis*:
Berlekamp-subspace.basis (*set* (*berlekamp-basis* *u*))
proof (*unfold* *set-berlekamp-basis-eq*, *rule* *linear-map.linear-inj-image-is-basis*)
show *linear-map* *class-ring* *V* *W* (*Poly* ∘ *list-of-vec*)
by (*rule* *linear-map-Poly-list-of-vec'*)
show *inj-on* (*Poly* ∘ *list-of-vec*) (*carrier* *V*)
proof (*rule* *subset-inj-on*[*OF* *inj-Poly-list-of-vec*])
show *carrier* *V* ⊆ *carrier-vec* (*degree* *u*)
by (*auto* *simp* *add*: *mat-kernel-def*)
qed
show (*Poly* ∘ *list-of-vec*) ‘ *carrier* *V* = *carrier* *W*
using *surj-Poly-list-of-vec*[*OF* *deg-u0*] **by** *auto*
show *b*: *V.basis* (*set* (*find-base-vectors* (*berlekamp-resulting-mat* *u*)))

by (*rule berlekamp-resulting-mat-basis*)
show *V.fin-dim*
proof −
have *finite* (*set* (*find-base-vectors* (*berlekamp-resulting-mat u*))) **by** *auto*
moreover have *set* (*find-base-vectors* (*berlekamp-resulting-mat u*)) $\subseteq$ *carrier V*
and *V.gen-set* (*set* (*find-base-vectors* (*berlekamp-resulting-mat u*)))
using *b* **unfolding** *V.basis-def* **by** *auto*
ultimately show *?thesis* **unfolding** *V.fin-dim-def* **by** *auto*
qed
qed

lemma *finsum-sum*:
fixes *f*::'*a mod-ring poly*
assumes *f*: *finite B*
and *a-Pi*: $a \in B \to$ *carrier R*
and *V*: $B \subseteq$ *carrier W*
shows ($\bigoplus_W v{\in}B.\ a\ v \odot_W v$) = *sum* ($\lambda v.$ *smult* ($a\ v$) v) *B*
using *f a-Pi V*
proof (*induct B*)
case *empty*
thus *?case* **unfolding** *Berlekamp-subspace.module.M.finsum-empty* **by** *auto*
next
case (*insert x V*)
have *hyp*: ($\bigoplus_W v \in V.\ a\ v \odot_W v$) = *sum* ($\lambda v.$ *smult* ($a\ v$) v) *V*
proof (*rule insert.hyps*)
show $a \in V \to$ *carrier R*
using *insert.prems* **unfolding** *class-field-def* **by** *auto*
show $V \subseteq$ *carrier W* **using** *insert.prems* **by** *simp*
qed
have ($\bigoplus_W v{\in}$*insert x V*. $a\ v \odot_W v$) = ($a\ x \odot_W x$) $\oplus_W$ ($\bigoplus_W v \in V.\ a\ v \odot_W v$)
proof (*rule abelian-monoid.finsum-insert*)
show *abelian-monoid W* **by** (*unfold-locales*)
show *finite V* **by** *fact*
show $x \notin V$ **by** *fact*
show ($\lambda v.\ a\ v \odot_W v$) $\in V \to$ *carrier W*
proof (*unfold Pi-def*, *rule*, *rule allI*, *rule impI*)
fix *v* **assume** *v*: $v{\in}V$
show $a\ v \odot_W v \in$ *carrier W*
proof (*rule Berlekamp-subspace.smult-closed*)
show $a\ v \in$ *carrier class-ring* **using** *insert.prems v* **unfolding** *Pi-def*
by (*simp add*: *class-field-def*)
show $v \in$ *carrier W* **using** *v insert.prems* **by** *auto*
qed
qed
show $a\ x \odot_W x \in$ *carrier W*
proof (*rule Berlekamp-subspace.smult-closed*)

show $a\ x \in$ *carrier class-ring* **using** *insert.prems* **unfolding** *Pi-def*
by (*simp add*: *class-field-def*)
show $x \in$ *carrier W* **using** *insert.prems* **by** *auto*
qed
qed
also have ... $= (a\ x \odot_W x) + (\bigoplus_W v \in V.\ a\ v \odot_W v)$ **by** *auto*
also have ... $= (a\ x \odot_W x) +$ *sum* $(\lambda v.$ *smult* $(a\ v)\ v)\ V$ **unfolding** *hyp* **by** *simp*
also have ... $=$ (*smult* $(a\ x)\ x) +$ *sum* $(\lambda v.$ *smult* $(a\ v)\ v)\ V$ **by** *simp*
also have ... $=$ *sum* $(\lambda v.$ *smult* $(a\ v)\ v)$ (*insert* $x\ V$)
by (*simp add*: *insert.hyps*(*1*) *insert.hyps*(*2*))
finally show *?case* **.**
qed

lemma *exists-vector-in-Berlekamp-subspace-dvd*:
fixes *p-i*::*'a mod-ring poly*
assumes *finite-P*: *finite P*
and *f-desc-square-free*: $u = (\prod a{\in}P.\ a)$
and *P*: $P \subseteq \{q.$ *irreducible* $q \wedge$ *monic* $q\}$
and *pi*: *p-i* $\in P$ **and** *pj*: *p-j* $\in P$ **and** *pi-pj*: *p-i* $\neq$ *p-j*
and *monic-f*: *monic u* **and** *sf-f*: *square-free u*
and *not-irr-w*: $\neg$ *irreducible w*
and *w-dvd-f*: *w dvd u* **and** *monic-w*: *monic w*
and *pi-dvd-w*: *p-i dvd w* **and** *pj-dvd-w*: *p-j dvd w*
shows $\exists v.\ v \in \{h.\ [h\widehat{\ }(CARD('a)) = h]\ (mod\ u) \wedge$ *degree h* $<$ *degree u*$\}$
$\wedge$ *v mod p-i* $\neq$ *v mod p-j*
$\wedge$ *degree* (*v mod p-i*) $= 0$
$\wedge$ *degree* (*v mod p-j*) $= 0$
— This implies that the algorithm decreases the degree of the reducible polynomials in each step:
$\wedge$ ($\exists s.$ *gcd w* $(v - [:s:]) \neq w \wedge$ *gcd w* $(v - [:s:]) \neq 1$)
proof −
have *f-not-0*: $u \neq 0$ **using** *monic-f* **by** *auto*
have *irr-pi*: *irreducible p-i* **using** *pi P* **by** *auto*
have *irr-pj*: *irreducible p-j* **using** *pj P* **by** *auto*
obtain *m* **and** *n*::*nat* **where** *P-m*: $P = m\ `\ \{i.\ i < n\}$ **and** *inj-on-m*: *inj-on m* $\{i.\ i < n\}$
using *finite-imp-nat-seg-image-inj-on*[*OF finite-P*] **by** *blast*
hence $n =$ *card P* **by** (*simp add*: *card-image*)
have *degree-prod*: *degree* (*prod m* $\{i.\ i < n\}$) $=$ *degree u*
by (*metis P-m f-desc-square-free inj-on-m prod.reindex-cong*)
have *not-zero*: $\forall i{\in}\{i.\ i < n\}.\ m\ i \neq 0$
using *P-m f-desc-square-free f-not-0* **by** *auto*
obtain *i* **where** *mi*: $m\ i =$ *p-i* **and** *i*: $i < n$ **using** *P-m pi* **by** *blast*
obtain *j* **where** *mj*: $m\ j =$ *p-j* **and** *j*: $j < n$ **using** *P-m pj* **by** *blast*
have *ij*: $i \neq j$ **using** *mi mj pi-pj* **by** *auto*
obtain *s-i* **and** *s-j*::*'a mod-ring* **where** *si-sj*: *s-i* $\neq$ *s-j* **using** *exists-two-distint* **by** *blast*

let $?u = \lambda x.\ if\ x = i\ then\ [:s\text{-}i:]\ else\ if\ x = j\ then\ [:s\text{-}j:]\ else\ [:0:]$
have *degree-si*: $degree\ [:s\text{-}i:] = 0$ **by** *auto*
have *degree-sj*: $degree\ [:s\text{-}j:] = 0$ **by** *auto*
have $\exists!v.\ degree\ v < (\sum i\in\{i.\ i < n\}.\ degree\ (m\ i)) \land (\forall a\in\{i.\ i < n\}.\ [v = ?u\ a]\ (mod\ m\ a))$
proof (*rule chinese-remainder-unique-poly*)
show $\forall a\in\{i.\ i < n\}.\ \forall b\in\{i.\ i < n\}.\ a \neq b \longrightarrow Rings.coprime\ (m\ a)\ (m\ b)$
proof (*rule*+)
fix *a b* **assume** $a \in \{i.\ i < n\}$ **and** $b \in \{i.\ i < n\}$ **and** $a \neq b$
thus $Rings.coprime\ (m\ a)\ (m\ b)$
using *coprime-polynomial-factorization*[*OF P finite-P*, *simplified*] *P-m*
by (*metis image-eqI inj-onD inj-on-m*)
qed
show $\forall i\in\{i.\ i < n\}.\ m\ i \neq 0$ **by** (*rule not-zero*)
show $0 < degree\ (prod\ m\ \{i.\ i < n\})$ **unfolding** *degree-prod* **using** *deg-u0* **by** *blast*
qed
from *this* **obtain** *v* **where** *v*: $\forall a\in\{i.\ i < n\}.\ [v = ?u\ a]\ (mod\ m\ a)$
and *degree-v*: $degree\ v < (\sum i\in\{i.\ i < n\}.\ degree\ (m\ i))$ **by** *blast*
show *?thesis*
proof (*rule exI*[*of - v*], *auto*)
show *vp-v-mod*: $[v\ \hat{}\ CARD('a) = v]\ (mod\ u)$
proof (*unfold f-desc-square-free*, *rule coprime-cong-mult-factorization-poly*[*OF finite-P*])
show $P \subseteq \{q.\ irreducible\ q\}$ **using** *P* **by** *blast*
show $\forall p\in P.\ [v\ \hat{}\ CARD('a) = v]\ (mod\ p)$
proof (*rule ballI*)
fix *p* **assume** *p*: $p \in P$
hence *irr-p*: $irreducible_d\ p$ **using** *P* **by** *auto*
obtain *k* **where** *mk*: $m\ k = p$ **and** *k*: $k < n$ **using** *P-m p* **by** *blast*
have $[v = ?u\ k]\ (mod\ p)$ **using** *v mk k* **by** *auto*
moreover have $?u\ k\ mod\ p = ?u\ k$
apply (*rule mod-poly-less*) **using** $irreducible_dD(1)$[*OF irr-p*] **by** *auto*
ultimately obtain *s* **where** *v-mod-p*: $v\ mod\ p = [:s:]$ **unfolding** *cong-def* **by** *force*
hence *deg-v-p*: $degree\ (v\ mod\ p) = 0$ **by** *auto*
have $v\ mod\ p = [:s:]$ **by** (*rule v-mod-p*)
also have $\ldots = [:s:]\hat{}CARD('a)$ **unfolding** *poly-const-pow* **by** *auto*
also have $\ldots = (v\ mod\ p)\ \hat{}\ CARD('a)$ **using** *v-mod-p* **by** *auto*
also have $\ldots = (v\ mod\ p)\ \hat{}\ CARD('a)\ mod\ p$ **using** *calculation* **by** *auto*
also have $\ldots = v\hat{}CARD('a)\ mod\ p$ **using** *power-mod* **by** *blast*
finally show $[v\ \hat{}\ CARD('a) = v]\ (mod\ p)$ **unfolding** *cong-def* **..**
qed
show $\forall p1\ p2.\ p1 \in P \land p2 \in P \land p1 \neq p2 \longrightarrow coprime\ p1\ p2$
using *P coprime-polynomial-factorization finite-P* **by** *auto*
qed
have $[v = ?u\ i]\ (mod\ m\ i)$ **using** *v i* **by** *auto*
hence *v-pi-si-mod*: $v\ mod\ p\text{-}i = [:s\text{-}i:]\ mod\ p\text{-}i$ **unfolding** *cong-def mi* **by** *auto*
also have $\ldots = [:s\text{-}i:]$ **apply** (*rule mod-poly-less*) **using** *irr-pi* **by** *auto*

finally have *v-pi-si*: *v mod p-i* = [:*s-i*:] **.**

have [*v* = *?u j*] (*mod m j*) **using** *v j* **by** *auto*
hence *v-pj-sj-mod*: *v mod p-j* = [:*s-j*:] *mod p-j* **unfolding** *cong-def mj* **using** *ij* **by** *auto*
also have ... = [:*s-j*:] **apply** (*rule mod-poly-less*) **using** *irr-pj* **by** *auto*
finally have *v-pj-sj*: *v mod p-j* = [:*s-j*:] **.**
show *v mod p-i* = *v mod p-j* $\Longrightarrow$ *False* **using** *si-sj v-pi-si v-pj-sj* **by** *auto*
show *degree* (*v mod p-i*) = *0* **unfolding** *v-pi-si* **by** *simp*
show *degree* (*v mod p-j*) = *0* **unfolding** *v-pj-sj* **by** *simp*
show $\exists$ *s*. *gcd w* (*v* − [:*s*:]) $\neq$ *w* $\wedge$ *gcd w* (*v* − [:*s*:]) $\neq$ *1*
proof (*rule exI*[*of - s-i*], *rule conjI*)
have *pi-dvd-v-si*: *p-i dvd v* − [:*s-i*:] **using** *v-pi-si-mod mod-eq-dvd-iff-poly* **by** *blast*
have *pj-dvd-v-sj*: *p-j dvd v* − [:*s-j*:] **using** *v-pj-sj-mod mod-eq-dvd-iff-poly* **by** *blast*
have *w-eq*: *w* = *prod* (λ*c*. *gcd w* (*v* − [:*c*:])) (*UNIV*::$'$*a mod-ring set*)
proof (*rule Berlekamp-gcd-step*)
show [*v* ^ *CARD*($'$*a*) = *v*] (*mod w*) **using** *vp-v-mod cong-dvd-modulus-poly w-dvd-f* **by** *blast*
show *square-free w* **by** (*rule square-free-factor*[*OF w-dvd-f sf-f*])
show *monic w* **by** (*rule monic-w*)
qed
show *gcd w* (*v* − [:*s-i*:]) $\neq$ *w*
proof (*rule ccontr*, *simp*)
assume *gcd-w*: *gcd w* (*v* − [:*s-i*:]) = *w*
show *False* **apply** (*rule* ‹*v mod p-i* = *v mod p-j* $\Longrightarrow$ *False*›)
by (*metis irreducibleE* ‹*degree* (*v mod p-i*) = *0*› *gcd-greatest-iff gcd-w irr-pj is-unit-field-poly mod-eq-dvd-iff-poly mod-poly-less neq0-conv pj-dvd-w v-pi-si*)
qed
show *gcd w* (*v* − [:*s-i*:]) $\neq$ *1*
by (*metis irreducibleE gcd-greatest-iff irr-pi pi-dvd-v-si pi-dvd-w*)
qed
show *degree v* < *degree u*
proof −
have ($\sum$ *i* | *i* < *n*. *degree* (*m i*)) = *degree* (*prod m* {*i*. *i* < *n*})
by (*rule degree-prod-eq-sum-degree*[*symmetric*, *OF not-zero*])
thus *?thesis* **using** *degree-v* **unfolding** *degree-prod* **by** *auto*
qed
qed
qed

lemma *exists-vector-in-Berlekamp-basis-dvd-aux*:
assumes *basis-V*: *Berlekamp-subspace.basis B*
and *finite-V*: *finite B*
assumes *finite-P*: *finite P*
and *f-desc-square-free*: *u* = ($\prod$ *a*$\in$*P*. *a*)

and *P*: $P \subseteq \{q.\ irreducible\ q \wedge monic\ q\}$
and *pi*: $p\text{-}i \in P$ **and** *pj*: $p\text{-}j \in P$ **and** *pi-pj*: $p\text{-}i \neq p\text{-}j$
and *monic-f*: *monic u* **and** *sf-f*: *square-free u*
and *not-irr-w*: ¬ *irreducible w*
and *w-dvd-f*: *w dvd u* **and** *monic-w*: *monic w*
and *pi-dvd-w*: *p-i dvd w* **and** *pj-dvd-w*: *p-j dvd w*
shows $\exists\, v \in B.\ v\ mod\ p\text{-}i \neq v\ mod\ p\text{-}j$
proof (*rule ccontr*, *auto*)
have *V-in-carrier*: $B \subseteq carrier\ W$
using *basis-V* **unfolding** *Berlekamp-subspace.basis-def* **by** *auto*
assume *all-eq*: $\forall\, v{\in}B.\ v\ mod\ p\text{-}i = v\ mod\ p\text{-}j$
obtain *x* **where** *x*: $x \in \{h.\ [h \mathbin{\char`\^} CARD('a) = h]\ (mod\ u) \wedge degree\ h < degree\ u\}$
and *x-pi-pj*: $x\ mod\ p\text{-}i \neq x\ mod\ p\text{-}j$ **and** $degree\ (x\ mod\ p\text{-}i) = 0$ **and** $degree\ (x\ mod\ p\text{-}j) = 0$
$(\exists\, s.\ gcd\ w\ (x - [:s:]) \neq w \wedge gcd\ w\ (x - [:s:]) \neq 1)$
using *exists-vector-in-Berlekamp-subspace-dvd*[*OF - - - pi pj - - - - w-dvd-f monic-w pi-dvd-w*]
assms **by** *meson*
have *x-in*: $x \in carrier\ W$ **using** *x* **by** *auto*
hence $(\exists!a.\ a \in B \rightarrow_E carrier\ class\text{-}ring \wedge Berlekamp\text{-}subspace.lincomb\ a\ B = x)$
using *Berlekamp-subspace.basis-criterion*[*OF finite-V V-in-carrier*] **using** *basis-V*
by (*simp add*: *class-field-def*)
from *this* **obtain** *a* **where** *a-Pi*: $a \in B \rightarrow_E carrier\ class\text{-}ring$
and *lincomb-x*: $Berlekamp\text{-}subspace.lincomb\ a\ B = x$
by *blast*
have *fs-ss*: $(\bigoplus_W v{\in}B.\ a\ v \odot_W v) = sum\ (\lambda v.\ smult\ (a\ v)\ v)\ B$
proof (*rule finsum-sum*)
show *finite B* **by** *fact*
show $a \in B \rightarrow carrier\ class\text{-}ring$ **using** *a-Pi* **by** *auto*
show $B \subseteq carrier\ W$ **by** (*rule V-in-carrier*)
qed
have $x\ mod\ p\text{-}i = Berlekamp\text{-}subspace.lincomb\ a\ B\ mod\ p\text{-}i$ **using** *lincomb-x* **by** *simp*
also have ... $= (\bigoplus_W v{\in}B.\ a\ v \odot_W v)\ mod\ p\text{-}i$ **unfolding** *Berlekamp-subspace.lincomb-def* **..**
also have ... $= (sum\ (\lambda v.\ smult\ (a\ v)\ v)\ B)\ mod\ p\text{-}i$ **unfolding** *fs-ss* **..**
also have ... $= sum\ (\lambda v.\ smult\ (a\ v)\ v\ mod\ p\text{-}i)\ B$ **using** *finite-V poly-mod-sum* **by** *blast*
also have ... $= sum\ (\lambda v.\ smult\ (a\ v)\ (v\ mod\ p\text{-}i))\ B$ **by** (*meson mod-smult-left*)
also have ... $= sum\ (\lambda v.\ smult\ (a\ v)\ (v\ mod\ p\text{-}j))\ B$ **using** *all-eq* **by** *auto*
also have ... $= sum\ (\lambda v.\ smult\ (a\ v)\ v\ mod\ p\text{-}j)\ B$ **by** (*metis mod-smult-left*)
also have ... $= (sum\ (\lambda v.\ smult\ (a\ v)\ v)\ B)\ mod\ p\text{-}j$
by (*metis* (*mono-tags*, *lifting*) *finite-V poly-mod-sum sum.cong*)
also have ... $= (\bigoplus_W v{\in}B.\ a\ v \odot_W v)\ mod\ p\text{-}j$ **unfolding** *fs-ss* **..**
also have ... $= Berlekamp\text{-}subspace.lincomb\ a\ B\ mod\ p\text{-}j$
unfolding *Berlekamp-subspace.lincomb-def* **..**
also have ... $= x\ mod\ p\text{-}j$ **using** *lincomb-x* **by** *simp*

finally have *x mod p-i = x mod p-j* **.**
thus *False* **using** *x-pi-pj* **by** *contradiction*
qed

lemma *exists-vector-in-Berlekamp-basis-dvd*:
assumes *basis-V*: *Berlekamp-subspace.basis B*
and *finite-V*: *finite B*
assumes *finite-P*: *finite P*
and *f-desc-square-free*: *u* = (∏ *a*∈*P*. *a*)
and *P*: *P* ⊆ {*q*. *irreducible q* ∧ *monic q*}
and *pi*: *p-i* ∈ *P* **and** *pj*: *p-j* ∈ *P* **and** *pi-pj*: *p-i* ≠ *p-j*
and *monic-f*: *monic u* **and** *sf-f*: *square-free u*
and *not-irr-w*: ¬ *irreducible w*
and *w-dvd-f*: *w dvd u* **and** *monic-w*: *monic w*
and *pi-dvd-w*: *p-i dvd w* **and** *pj-dvd-w*: *p-j dvd w*
shows ∃ *v* ∈ *B*. *v mod p-i* ≠ *v mod p-j*
∧ *degree* (*v mod p-i*) = *0*
∧ *degree* (*v mod p-j*) = *0*
— This implies that the algorithm decreases the degree of the reducible polynomials in each step:
∧ (∃ *s*. *gcd w* (*v* − [:*s*:]) ≠ *w* ∧ ¬ *coprime w* (*v* − [:*s*:]))
proof −
have *f-not-0*: *u* ≠ *0* **using** *monic-f* **by** *auto*
have *irr-pi*: *irreducible p-i* **using** *pi P* **by** *fast*
have *irr-pj*: *irreducible p-j* **using** *pj P* **by** *fast*
obtain *v* **where** *vV*: *v* ∈ *B* **and** *v-pi-pj*: *v mod p-i* ≠ *v mod p-j*
using *assms exists-vector-in-Berlekamp-basis-dvd-aux* **by** *blast*
have *v*: *v* ∈ {*v*. [*v* ^ *CARD*(*'a*) = *v*] (*mod u*)}
using *basis-V vV* **unfolding** *Berlekamp-subspace.basis-def* **by** *auto*
have *deg-v-pi*: *degree* (*v mod p-i*) = *0*
by (*rule degree-u-mod-irreducible*$_d$*-factor-0*[*OF v finite-P f-desc-square-free P pi*])
from *this* **obtain** *s-i* **where** *v-pi-si*: *v mod p-i* = [:*s-i*:] **using** *degree-eq-zeroE* **by** *blast*
have *deg-v-pj*: *degree* (*v mod p-j*) = *0*
by (*rule degree-u-mod-irreducible*$_d$*-factor-0*[*OF v finite-P f-desc-square-free P pj*])
from *this* **obtain** *s-j* **where** *v-pj-sj*: *v mod p-j* = [:*s-j*:] **using** *degree-eq-zeroE* **by** *blast*
have *si-sj*: *s-i* ≠ *s-j* **using** *v-pi-si v-pj-sj v-pi-pj* **by** *auto*
have (∃ *s*. *gcd w* (*v* − [:*s*:]) ≠ *w* ∧ ¬ *Rings.coprime w* (*v* − [:*s*:]))
proof (*rule exI*[*of - s-i*], *rule conjI*)
have *pi-dvd-v-si*: *p-i dvd v* − [:*s-i*:] **by** (*metis mod-eq-dvd-iff-poly mod-mod-trivial v-pi-si*)
have *pj-dvd-v-sj*: *p-j dvd v* − [:*s-j*:] **by** (*metis mod-eq-dvd-iff-poly mod-mod-trivial v-pj-sj*)
have *w-eq*: *w* = *prod* (λ*c*. *gcd w* (*v* − [:*c*:])) (*UNIV*::*'a mod-ring set*)
proof (*rule Berlekamp-gcd-step*)

show [v ^ $CARD('a)$ = v] (mod w) **using** *v cong-dvd-modulus-poly w-dvd-f*
by *blast*
show *square-free w* **by** (*rule square-free-factor*[*OF w-dvd-f sf-f*])
show *monic w* **by** (*rule monic-w*)
qed
show *gcd w* (v − [:s-i:]) ≠ w
by (*metis irreducibleE deg-v-pi gcd-greatest-iff irr-pj is-unit-field-poly mod-eq-dvd-iff-poly mod-poly-less neq0-conv pj-dvd-w v-pi-pj v-pi-si*)
show ¬ *Rings.coprime w* (v − [:s-i:])
using *irr-pi pi-dvd-v-si pi-dvd-w*
by (*simp add*: *irreducible$_d$D*(*1*) *not-coprimeI*)
qed
thus *?thesis* **using** *v-pi-pj vV deg-v-pi deg-v-pj* **by** *auto*
qed

lemma *exists-bijective-linear-map-W-vec*:
assumes *finite-P*: *finite P*
and *u-desc-square-free*: u = ($\prod$ a∈P. a)
and *P*: P ⊆ {q. *irreducible q* ∧ *monic q*}
shows ∃f. *linear-map class-ring W* (*module-vec TYPE*(*'a mod-ring*) (*card P*)) f
∧ *bij-betw f* (*carrier W*) (*carrier-vec* (*card P*)::*'a mod-ring vec set*)
proof −
let *?B*=*carrier-vec* (*card P*)::*'a mod-ring vec set*
have *u-not-0*: u ≠ *0* **using** *deg-u0 degree-0* **by** *force*
obtain m **and** n::*nat* **where** *P-m*: P = m ‘ {i. i < n} **and** *inj-on-m*: *inj-on m*
{i. i < n}
using *finite-imp-nat-seg-image-inj-on*[*OF finite-P*] **by** *blast*
hence n: n = *card P* **by** (*simp add*: *card-image*)
have *degree-prod*: *degree* (*prod m* {i. i < n}) = *degree u*
by (*metis P-m u-desc-square-free inj-on-m prod.reindex-cong*)
have *not-zero*: ∀ i∈{i. i < n}. m i ≠ *0*
using *P-m u-desc-square-free u-not-0* **by** *auto*
have *deg-sum-eq*: ($\sum$ i∈{i. i < n}. *degree* (m i)) = *degree u*
by (*metis degree-prod degree-prod-eq-sum-degree not-zero*)
have *coprime-mi-mj*:∀ i∈{i. i < n}. ∀ j∈{i. i < n}. i ≠ j ⟶ *coprime* (m i) (m
j)
proof (*rule*+)
fix i j **assume** i: i ∈ {i. i < n}
and j: j ∈ {i. i < n} **and** *ij*: i ≠ j
show *coprime* (m i) (m j)
proof (*rule coprime-polynomial-factorization*[*OF P finite-P*])
show m i ∈ P **using** *i P-m* **by** *auto*
show m j ∈ P **using** *j P-m* **by** *auto*
show m i ≠ m j **using** *inj-on-m i ij j* **unfolding** *inj-on-def* **by** *blast*
qed
qed
let *?f* = λv. *vec n* (λi. *coeff* (v *mod* (m i)) *0*)
interpret *vec-VS*: *vectorspace class-ring* (*module-vec TYPE*(*'a mod-ring*) n)
by (*rule VS-Connect.vec-vs*)

interpret *linear-map class-ring W* (*module-vec TYPE*(*'a mod-ring*) *n*) *?f*
by (*intro-locales*, *unfold mod-hom-axioms-def LinearCombinations.module-hom-def*, *auto simp add*: *vec-eq-iff module-vec-def mod-smult-left poly-mod-add-left*)
have *linear-map class-ring W* (*module-vec TYPE*(*'a mod-ring*) *n*) *?f*
by (*intro-locales*)
moreover have *inj-f*: *inj-on ?f* (*carrier W*)
proof (*rule Ke0-imp-inj*, *auto simp add*: *mod-hom.ker-def*)
show $[0 \,\widehat{}\, CARD('a) = 0]$ (*mod u*) **by** (*simp add*: *cong-def zero-power*)
show *vec n* ($\lambda i.\ 0$) = $\mathbf{0}_{module\text{-}vec\ TYPE('a\ mod\text{-}ring)\ n}$ **by** (*auto simp add*: *module-vec-def*)
fix *x* **assume** *x*: $[x \,\widehat{}\, CARD('a) = x]$ (*mod u*) **and** *deg-x*: *degree x* < *degree u*
and *v*: *vec n* ($\lambda i.$ *coeff* (*x mod m i*) *0*) = $\mathbf{0}_{module\text{-}vec\ TYPE('a\ mod\text{-}ring)\ n}$
have *cong-0*: $\forall i \in \{i.\ i < n\}$. [*x* = ($\lambda i.\ 0$) *i*] (*mod m i*)
proof (*rule*, *unfold cong-def*)
fix *i* **assume** *i*: $i \in \{i.\ i < n\}$
have *deg-x-mod-mi*: *degree* (*x mod m i*) = *0*
proof (*rule degree-u-mod-irreducible$_d$-factor-0*[*OF - finite-P u-desc-square-free P*])
show $x \in \{v.\ [v \,\widehat{}\, CARD('a) = v]$ (*mod u*)} **using** *x* **by** *auto*
show *m i* ∈ *P* **using** *P-m i* **by** *auto*
qed
thus *x mod m i* = *0 mod m i*
using *v*
unfolding *module-vec-def*
by (*auto*, *metis i leading-coeff-neq-0 mem-Collect-eq index-vec index-zero-vec*(*1*))
qed
moreover have *deg-x2*: *degree x* < ($\sum i \in \{i.\ i < n\}$. *degree* (*m i*))
using *deg-sum-eq deg-x* **by** *simp*
moreover have $\forall i \in \{i.\ i < n\}$. [*0* = ($\lambda i.\ 0$) *i*] (*mod m i*)
by (*auto simp add*: *cong-def*)
moreover have *degree 0* < ($\sum i \in \{i.\ i < n\}$. *degree* (*m i*))
using *degree-prod deg-sum-eq deg-u0* **by** *force*
moreover have $\exists!x$. *degree x* < ($\sum i \in \{i.\ i < n\}$. *degree* (*m i*))
∧ ($\forall i \in \{i.\ i < n\}$. [*x* = ($\lambda i.\ 0$) *i*] (*mod m i*))
proof (*rule chinese-remainder-unique-poly*[*OF not-zero*])
show *0* < *degree* (*prod m* {*i. i* < *n*})
using *deg-u0 degree-prod* **by** *linarith*
qed (*insert coprime-mi-mj*, *auto*)
ultimately show *x* = *0* **by** *blast*
qed
moreover have *?f* ‘ (*carrier W*) = *?B*
proof (*auto simp add*: *image-def*)
fix *xa*
show *n* = *card P* **by** (*auto simp add*: *n*)
next
fix *x*::*'a mod-ring vec* **assume** *x*: *x* ∈ *carrier-vec* (*card P*)
have $\exists!v$. *degree v* < ($\sum i \in \{i.\ i < n\}$. *degree* (*m i*)) ∧ ($\forall i \in \{i.\ i < n\}$. [*v* = ($\lambda i.$ [:*x* $ *i*:]) *i*] (*mod m i*))
proof (*rule chinese-remainder-unique-poly*[*OF not-zero*])

```
      show 0 < degree (prod m {i. i < n})
        using deg-u0 degree-prod by linarith
    qed (insert coprime-mi-mj, auto)
    from this obtain v where deg-v: degree v < (∑ i∈{i. i < n}. degree (m i))
      and v-x-cong: (∀ i ∈ {i. i < n}. [v = (λi. [:x $ i:]) i] (mod m i)) by auto
    show ∃ xa. [xa ^ CARD('a) = xa] (mod u) ∧ degree xa < degree u
      ∧ x = vec n (λi. coeff (xa mod m i) 0)
    proof (rule exI[of - v], auto)
      show v: [v ^ CARD('a) = v] (mod u)
     proof (unfold u-desc-square-free, rule coprime-cong-mult-factorization-poly[OF
finite-P], auto)
        fix y assume y: y ∈ P thus irreducible y using P by blast
        obtain i where i: i ∈ {i. i < n} and mi: y = m i using P-m y by blast
        have irreducible (m i) using i P-m P by auto
        moreover have [v = [:x $ i:]] (mod m i) using v-x-cong i by auto
        ultimately have v-mi-eq-xi: v mod m i = [:x $ i:]
          by (auto simp: cong-def intro!: mod-poly-less)
       have xi-pow-xi: [:x $ i:]^CARD('a) = [:x $ i:] by (simp add: poly-const-pow)
        hence (v mod m i)^CARD('a) = v mod m i using v-mi-eq-xi by auto
        hence (v mod m i)^CARD('a) = (v^CARD('a) mod m i)
          by (metis mod-mod-trivial power-mod)
      thus [v ^ CARD('a) = v] (mod y) unfolding mi cong-def v-mi-eq-xi xi-pow-xi
by simp
      next
        fix p1 p2 assume p1 ∈ P and p2 ∈ P and p1 ≠ p2
        then show Rings.coprime p1 p2
          using coprime-polynomial-factorization[OF P finite-P] by auto
      qed
      show degree v < degree u using deg-v deg-sum-eq degree-prod by presburger
      show x = vec n (λi. coeff (v mod m i) 0)
      proof (unfold vec-eq-iff, rule conjI)
        show dim-vec x = dim-vec (vec n (λi. coeff (v mod m i) 0)) using x n by
simp
        show ∀ i<dim-vec (vec n (λi. coeff (v mod m i) 0)). x $ i = vec n (λi.
coeff (v mod m i) 0) $ i
        proof (auto)
          fix i assume i: i < n
          have deg-mi: irreducible (m i) using i P-m P by auto
          have deg-v-mi: degree (v mod m i) = 0
        proof (rule degree-u-mod-irreducible_d-factor-0[OF - finite-P u-desc-square-free
P])
            show v ∈ {v. [v ^ CARD('a) = v] (mod u)} using v by fast
            show m i ∈ P using P-m i by auto
          qed
          have v mod m i = [:x $ i:] mod m i using v-x-cong i unfolding cong-def
by auto
          also have ... = [:x $ i:] using deg-mi by (auto intro!: mod-poly-less)
          finally show x $ i = coeff (v mod m i) 0 by simp
        qed
```

qed
qed
qed
ultimately show *?thesis* **unfolding** *bij-betw-def n* **by** *auto*
qed

lemma *fin-dim-kernel-berlekamp*: *V.fin-dim*
proof −
have *finite* (*set* (*find-base-vectors* (*berlekamp-resulting-mat u*))) **by** *auto*
moreover have *set* (*find-base-vectors* (*berlekamp-resulting-mat u*)) ⊆ *carrier V*
and *V.gen-set* (*set* (*find-base-vectors* (*berlekamp-resulting-mat u*)))
using *berlekamp-resulting-mat-basis*[*of u*] **unfolding** *V.basis-def* **by** *auto*
ultimately show *?thesis* **unfolding** *V.fin-dim-def* **by** *auto*
qed

lemma *Berlekamp-subspace-fin-dim*: *Berlekamp-subspace.fin-dim*
proof (*rule linear-map.surj-fin-dim*[*OF linear-map-Poly-list-of-vec′*])
show (*Poly* ∘ *list-of-vec*) ‘ *carrier V* = *carrier W*
using *surj-Poly-list-of-vec*[*OF deg-u0*] **by** *auto*
show *V.fin-dim* **by** (*rule fin-dim-kernel-berlekamp*)
qed

context
fixes *P*
assumes *finite-P*: *finite P*
and *u-desc-square-free*: *u* = (∏ *a*∈*P*. *a*)
and *P*: *P* ⊆ {*q*. *irreducible q* ∧ *monic q*}
begin

interpretation *RV*: *vec-space TYPE*(′*a mod-ring*) *card P* **.**

lemma *Berlekamp-subspace-eq-dim-vec*: *Berlekamp-subspace.dim* = *RV.dim*
proof −
obtain *f* **where** *lm-f*: *linear-map class-ring W* (*module-vec TYPE*(′*a mod-ring*) (*card P*)) *f*
and *bij-f*: *bij-betw f* (*carrier W*) (*carrier-vec* (*card P*)::′*a mod-ring vec set*)
using *exists-bijective-linear-map-W-vec*[*OF finite-P u-desc-square-free P*] **by** *blast*
show *?thesis*
proof (*rule linear-map.dim-eq*[*OF lm-f Berlekamp-subspace-fin-dim*])
show *inj-on f* (*carrier W*) **by** (*rule bij-betw-imp-inj-on*[*OF bij-f*])
show *f* ‘ *carrier W* = *carrier RV.V* **using** *bij-f* **unfolding** *bij-betw-def* **by** *auto*
qed
qed

lemma *Berlekamp-subspace-dim*: *Berlekamp-subspace.dim* = *card P*
using *Berlekamp-subspace-eq-dim-vec RV.dim-is-n* **by** *simp*

corollary *card-berlekamp-basis-number-factors*: *card* (*set* (*berlekamp-basis* *u*)) = *card* *P*
  unfolding *Berlekamp-subspace-dim*[*symmetric*]
  by (*rule Berlekamp-subspace.dim-basis*[*symmetric*], *auto simp add*: *berlekamp-basis-basis*)

lemma *length-berlekamp-basis-numbers-factors*: *length* (*berlekamp-basis* *u*) = *card* *P*
  using *card-set-berlekamp-basis card-berlekamp-basis-number-factors* **by** *auto*

end
end
end
end

context
  assumes *SORT-CONSTRAINT*(*'a* :: *prime-card*)
begin

context
  fixes *f* :: *'a mod-ring poly* **and** *n*
  assumes *sf*: *square-free f*
  and *n*: *n* = *length* (*berlekamp-basis f*)
  and *monic-f*: *monic f*
begin
lemma *berlekamp-basis-length-factorization*: **assumes** *f*: *f* = *prod-list us*
  and *d*: $\bigwedge$ *u*. *u* ∈ *set us* ⟹ *degree u* > *0*
  shows *length us* ≤ *n*
proof (*cases degree f* = *0*)
  case *True*
  have *us* = []
  proof (*rule ccontr*)
    assume *us* ≠ []
    from *this* **obtain** *u* **where** *u*: *u* ∈ *set us* **by** *fastforce*
    hence *deg-u*: *degree u* > *0* **using** *d* **by** *auto*
    have *degree f* = *degree* (*prod-list us*) **unfolding** *f* **..**
    also have ... = *sum-list* (*map degree us*)
    proof (*rule degree-prod-list-eq*)
      fix *p* **assume** *p*: *p* ∈ *set us*
      show *p* ≠ *0* **using** *d*[*OF p*] *degree-0* **by** *auto*
    qed
    also have ... ≥ *degree u* **by** (*simp add*: *member-le-sum-list u*)
    finally have *degree f* > *0* **using** *deg-u* **by** *auto*
    thus *False* **using** *True* **by** *auto*
  qed
  thus *?thesis* **by** *simp*
next

case *False*
hence *f-not-0*: $f \neq 0$ **using** *degree-0* **by** *fastforce*
obtain *P* **where** *fin-P*: *finite P* **and** *f-P*: $f = \prod P$ **and** *P*: $P \subseteq \{p.$ *irreducible* $p \wedge$ *monic* $p\}$
using *monic-square-free-irreducible-factorization*[*OF monic-f sf*] **by** *auto*
have *n-card-P*: $n =$ *card P*
using *P False f-P fin-P length-berlekamp-basis-numbers-factors n* **by** *blast*
have *distinct-us*: *distinct us* **using** *d f sf square-free-prod-list-distinct* **by** *blast*
let *?us'*=(*map normalize us*)
have *distinct-us'*: *distinct ?us'*
proof (*auto simp add*: *distinct-map*)
show *distinct us* **by** (*rule distinct-us*)
show *inj-on normalize* (*set us*)
proof (*auto simp add*: *inj-on-def*, *rule ccontr*)
fix *x y* **assume** *x*: $x \in$ *set us* **and** *y*: $y \in$ *set us* **and** *n*: *normalize* $x =$ *normalize* y
and *x-not-y*: $x \neq y$
from *normalize-eq-imp-smult*[*OF n*]
obtain *c* **where** *c0*: $c \neq 0$ **and** *y-smult*: $y =$ *smult c x* **by** *blast*
have *sf-xy*: *square-free* ($x{*}y$)
proof (*rule square-free-factor*[*OF - sf*])
have $x{*}y =$ *prod-list* [*x*,*y*] **by** *simp*
also have ... *dvd prod-list us*
by (*rule prod-list-dvd-prod-list-subset*, *auto simp add*: *x y x-not-y distinct-us*)
also have ... $= f$ **unfolding** *f* **..**
finally show $x * y$ *dvd f* **.**
qed
have $x * y =$ *smult c* ($x{*}x$) **using** *y-smult mult-smult-right* **by** *auto*
hence *sf-smult*: *square-free* (*smult c* ($x{*}x$)) **using** *sf-xy* **by** *auto*
have $x{*}x$ *dvd* (*smult c* ($x{*}x$)) **by** (*simp add*: *dvd-smult*)
hence ¬ *square-free* (*smult c* ($x{*}x$))
by (*metis d square-free-def x*)
thus *False* **using** *sf-smult* **by** *contradiction*
qed
qed
have *length-us-us'*: *length us* = *length ?us'* **by** *simp*
have *f-us'*: $f =$ *prod-list ?us'*
proof −
have $f =$ *normalize f* **using** *monic-f f-not-0* **by** (*simp add*: *normalize-monic*)
also have ... = *prod-list ?us'* **by** (*unfold f*, *rule prod-list-normalize*[*of us*])
finally show *?thesis* **.**
qed
have ∃ *Q*. *prod-list Q* = *prod-list ?us'* ∧ *length ?us'* ≤ *length Q*
∧ (∀ *u*. *u* ∈ *set Q* ⟶ *irreducible u* ∧ *monic u*)
proof (*rule exists-factorization-prod-list*)
show *degree* (*prod-list ?us'*) > *0* **using** *False f-us'* **by** *auto*
show *square-free* (*prod-list ?us'*) **using** *f-us' sf* **by** *auto*
fix *u* **assume** *u*: *u* ∈ *set ?us'*
have *u-not0*: $u \neq 0$ **using** *d u degree-0* **by** *fastforce*

have *degree u > 0* **using** *d u* **by** *auto*
moreover have *monic u* **using** *u monic-normalize*[*OF u-not0*] **by** *auto*
ultimately show *degree u > 0* ∧ *monic u* **by** *simp*
qed
from *this* **obtain** *Q*
where *Q-us'*: *prod-list Q = prod-list ?us'*
and *length-us'-Q*: *length ?us'* ≤ *length Q*
and *Q*: (∀ *u*. *u* ∈ *set Q* ⟶ *irreducible u* ∧ *monic u*)
by *blast*
have *distinct-Q*: *distinct Q*
proof (*rule square-free-prod-list-distinct*)
show *square-free* (*prod-list Q*) **using** *Q-us' f-us' sf* **by** *auto*
show ⋀*u*. *u* ∈ *set Q* ⟹ *degree u > 0* **using** *Q irreducible-degree-field* **by** *auto*
qed
have *set-Q-P*: *set Q = P*
proof (*rule monic-factorization-uniqueness*)
show ∏(*set Q*) = ∏*P* **using** *Q-us'*
by (*metis distinct-Q f-P f-us' list.map-ident prod.distinct-set-conv-list*)
qed (*insert P Q fin-P*, *auto*)
hence *length Q = card P* **using** *distinct-Q distinct-card* **by** *fastforce*
have *length us = length ?us'* **by** (*rule length-us-us'*)
also have ... ≤ *length Q* **using** *length-us'-Q* **by** *auto*
also have ... = *card* (*set Q*) **using** *distinct-card*[*OF distinct-Q*] **by** *simp*
also have ... = *card P* **using** *set-Q-P* **by** *simp*
finally show *?thesis* **using** *n-card-P* **by** *simp*
qed

lemma *berlekamp-basis-irreducible*: **assumes** *f*: *f = prod-list us*
and *n-us*: *length us = n*
and *us*: ⋀ *u*. *u* ∈ *set us* ⟹ *degree u > 0*
and *u*: *u* ∈ *set us*
shows *irreducible u*
proof (*fold irreducible-connect-field*, *intro* $irreducible_d I$[*OF us*[*OF u*]])
fix *q r* :: *'a mod-ring poly*
assume *dq*: *degree q > 0* **and** *qu*: *degree q < degree u* **and** *dr*: *degree r > 0* **and** *uqr*: *u = q * r*
with *us*[*OF u*] **have** *q*: *q ≠ 0* **and** *r*: *r ≠ 0* **by** *auto*
from *split-list*[*OF u*] **obtain** *xs ys* **where** *id*: *us = xs @ u # ys* **by** *auto*
let *?us = xs @ q # r # ys*
have *f*: *f = prod-list ?us* **unfolding** *f id uqr* **by** *simp*
{
fix *x*
assume *x* ∈ *set ?us*
with *us*[*unfolded id*] *dr dq* **have** *degree x > 0* **by** *auto*
}
from *berlekamp-basis-length-factorization*[*OF f this*]
have *length ?us* ≤ *n* **by** *simp*
also have . . . = *length us* **unfolding** *n-us* **by** *simp*
also have . . . < *length ?us* **unfolding** *id* **by** *simp*

finally show *False* **by** *simp*
qed
end

lemma *not-irreducible-factor-yields-prime-factors*:
  assumes *uf*: *u dvd* (*f* :: *'b* :: {*field-gcd*} *poly*) **and** *fin*: *finite P*
    and *fP*: $f = \prod P$ **and** *P*: $P \subseteq \{q.\ irreducible\ q \wedge monic\ q\}$
   and *u*: *degree u* > *0* ¬ *irreducible u*
  shows $\exists\ pi\ pj.\ pi \in P \wedge pj \in P \wedge pi \neq pj \wedge pi\ dvd\ u \wedge pj\ dvd\ u$
proof −
  from *finite-distinct-list*[*OF fin*] **obtain** *ps* **where** *Pps*: *P* = *set ps* **and** *dist*: *distinct ps* **by** *auto*
  have *fP*: *f* = *prod-list ps* **unfolding** *fP Pps* **using** *dist*
   by (*simp add*: *prod.distinct-set-conv-list*)
  note *P* = *P*[*unfolded Pps*]
  have *set ps* ⊆ *P* **unfolding** *Pps* **by** *auto*
  from *uf*[*unfolded fP*] *P dist this*
  show *?thesis*
  proof (*induct ps*)
   case *Nil*
   with *u* **show** *?case* **using** *divides-degree*[*of u 1*] **by** *auto*
  next
   case (*Cons p ps*)
   from *Cons*(*3*) **have** *ps*: *set ps* ⊆ {*q*. *irreducible q* ∧ *monic q*} **by** *auto*
   from *Cons*(*2*) **have** *dvd*: *u dvd p* ∗ *prod-list ps* **by** *simp*
   obtain *k* **where** *gcd*: *u* = *gcd p u* ∗ *k* **by** (*meson dvd-def gcd-dvd2*)
   from *Cons*(*3*) **have** ∗: *monic p irreducible p p* ≠ *0* **by** *auto*
   from *monic-irreducible-gcd*[*OF* ∗(*1*), *of u*] ∗(*2*)
   have *gcd p u* = *1* ∨ *gcd p u* = *p* **by** *auto*
   thus *?case*
   proof
    assume *gcd p u* = *1*
    then have *Rings.coprime p u*
     by (*rule gcd-eq-1-imp-coprime*)
    with *dvd* **have** *u dvd prod-list ps*
     using *coprime-dvd-mult-right-iff coprime-imp-coprime* **by** *blast*
    from *Cons*(*1*)[*OF this ps*] *Cons*(*4*−*5*) **show** *?thesis* **by** *auto*
   next
    assume *gcd p u* = *p*
    with *gcd* **have** *upk*: *u* = *p* ∗ *k* **by** *auto*
    hence *p*: *p dvd u* **by** *auto*
    from *dvd*[*unfolded upk*] ∗(*3*) **have** *kps*: *k dvd prod-list ps* **by** *auto*
    from *dvd u* ∗ **have** *dk*: *degree k* > *0*
      by (*metis gr0I irreducible-mult-unit-right is-unit-iff-degree mult-zero-right upk*)
    from *ps kps* **have** ∃ *q* ∈ *set ps*. *q dvd k*
    proof (*induct ps*)
     case *Nil*
     with *dk* **show** *?case* **using** *divides-degree*[*of k 1*] **by** *auto*

next
case (*Cons p ps*)
from *Cons*(*3*) **have** *dvd*: *k dvd p* ∗ *prod-list ps* **by** *simp*
obtain *l* **where** *gcd*: *k* = *gcd p k* ∗ *l* **by** (*meson dvd-def gcd-dvd2*)
from *Cons*(*2*) **have** ∗: *monic p irreducible p p* ≠ *0* **by** *auto*
from *monic-irreducible-gcd*[*OF* ∗(*1*), *of k*] ∗(*2*)
have *gcd p k* = *1* ∨ *gcd p k* = *p* **by** *auto*
thus *?case*
proof
assume *gcd p k* = *1*
with *dvd* **have** *k dvd prod-list ps*
by (*metis dvd-triv-left gcd-greatest-mult mult.left-neutral*)
from *Cons*(*1*)[*OF - this*] *Cons*(*2*) **show** *?thesis* **by** *auto*
next
assume *gcd p k* = *p*
with *gcd* **have** *upk*: *k* = *p* ∗ *l* **by** *auto*
hence *p*: *p dvd k* **by** *auto*
thus *?thesis* **by** *auto*
qed
qed
then obtain *q* **where** *q*: *q* ∈ *set ps* **and** *dvd*: *q dvd k* **by** *auto*
from *dvd upk* **have** *qu*: *q dvd u* **by** *auto*
from *Cons*(*4*) *q* **have** *p* ≠ *q* **by** *auto*
thus *?thesis* **using** *q p qu Cons*(*5*) **by** *auto*
qed
qed
qed

lemma *berlekamp-factorization-main*:
fixes *f*::*'a mod-ring poly*
assumes *sf-f*: *square-free f*
and *vs*: *vs* = *vs1* @ *vs2*
and *vsf*: *vs* = *berlekamp-basis f*
and *n-bb*: *n* = *length* (*berlekamp-basis f*)
and *n*: *n* = *length us1* + *n2*
and *us*: *us* = *us1* @ *berlekamp-factorization-main d divs vs2 n2*
and *us1*: ⋀ *u*. *u* ∈ *set us1* ⟹ *monic u* ∧ *irreducible u*
and *divs*: ⋀ *d*. *d* ∈ *set divs* ⟹ *monic d* ∧ *degree d* > *0*
and *vs1*: ⋀ *u v i*. *v* ∈ *set vs1* ⟹ *u* ∈ *set us1* ∪ *set divs*
⟹ *i* < *CARD*(*'a*) ⟹ *gcd u* (*v* − [:*of-nat i*:]) ∈ {*1*,*u*}
and *f*: *f* = *prod-list* (*us1* @ *divs*)
and *deg-f*: *degree f* > *0*
and *d*: ⋀ *g*. *g dvd f* ⟹ *degree g* = *d* ⟹ *irreducible g*
shows *f* = *prod-list us* ∧ (∀ *u* ∈ *set us*. *monic u* ∧ *irreducible u*)
proof −
have *mon-f*: *monic f* **unfolding** *f*
by (*rule monic-prod-list*, *insert divs us1*, *auto*)
from *monic-square-free-irreducible-factorization*[*OF mon-f sf-f*] **obtain** *P* **where**
P: *finite P f* = ∏ *P P* ⊆ {*q*. *irreducible q* ∧ *monic q*} **by** *auto*

hence *f0*: $f \neq 0$ **by** *auto*
show *?thesis*
using *vs n us divs f us1 vs1*
proof (*induct vs2 arbitrary*: *divs n2 us1 vs1*)
case (*Cons v vs2*)
show *?case*
proof (*cases* $v = 1$)
case *False*
from *Cons*(*2*) *vsf* **have** *v*: $v \in$ *set* (*berlekamp-basis f*) **by** *auto*
from *berlekamp-basis-eq-8*[*OF this*] **have** *vf*: [v ^ *CARD*('*a*) = v] (*mod f*) **.**
let *?gcd* = λ *u i. gcd u* (v − [:*of-int i*:])
let *?gcdn* = λ *u i. gcd u* (v − [:*of-nat i*:])
let *?map* = λ *u.* (*map* (λ *i. ?gcd u i*) [*0* ..< *CARD*('*a*)])
define *udivs* **where** *udivs* ≡ λ *u. filter* (λ *w.* $w \neq 1$) (*?map u*)
{
obtain *xs* **where** *xs*: [*0*..<*CARD*('*a*)] = *xs* **by** *auto*
have *udivs* = (λ *u.* [*w. i* ← [*0* ..< *CARD*('*a*)], *w* ← [*?gcd u i*], $w \neq 1$])
unfolding *udivs-def xs*
by (*intro ext*, *auto simp*: *o-def*, *induct xs*, *auto*)
} **note** *udivs-def'* = *this*
define *facts* **where** *facts* ≡ [*w* . *u* ← *divs*, *w* ← *udivs u*]
{
fix *u*
assume *u*: $u \in$ *set divs*
then obtain *bef aft* **where** *divs*: *divs* = *bef* @ *u* # *aft* **by** (*meson split-list*)
from *Cons*(*5*)[*OF u*] **have** *mon-u*: *monic u* **by** *simp*
have *uf*: *u dvd f* **unfolding** *Cons*(*6*) *divs* **by** *auto*
from *vf uf* **have** *vu*: [v ^ *CARD*('*a*) = v] (*mod u*) **by** (*rule cong-dvd-modulus-poly*)
from *square-free-factor*[*OF uf sf-f*] **have** *sf-u*: *square-free u* **.**
let *?g* = *?gcd u*
from *mon-u* **have** *u0*: $u \neq 0$ **by** *auto*
have *u* = ($\prod$ *c*∈*UNIV*. *gcd u* (v − [:*c*:]))
using *Berlekamp-gcd-step*[*OF vu mon-u sf-u*] **.**
also have ... = ($\prod$ *i* ∈ {*0*..< *int CARD*('*a*)}. *?g i*)
by (*rule sym*, *rule prod.reindex-cong*[*OF to-int-mod-ring-hom.inj-f range-to-int-mod-ring*[*symmetric*]],
simp add: *of-int-of-int-mod-ring*)
finally have *u-prod*: *u* = ($\prod$ *i* ∈ {*0*..< *int CARD*('*a*)}. *?g i*) **.**
let *?S* = {*0*..<*int CARD*('*a*)} − {*i. ?g i* = *1*}
{
fix *i*
assume *i* ∈ *?S*
hence *?g i* ≠ *1* **by** *auto*
moreover have *mgi*: *monic* (*?g i*) **by** (*rule poly-gcd-monic*, *insert u0*,
auto)
ultimately have *degree* (*?g i*) > *0*
using *monic-degree-0* **by** *blast*
note *this mgi*
} **note** *gS* = *this*

have *int-set*: *int* ‘ *set* [*0*..<*CARD*(′*a*)] = {*0* ..< *int CARD*(′*a*)}
  by (*simp add*: *image-int-atLeastLessThan*)

have *inj*: *inj-on ?g ?S* **unfolding** *inj-on-def*
proof (*intro ballI impI*)
  fix *i j*
  assume *i*: $i \in$ *?S* **and** *j*: $j \in$ *?S* **and** *gij*: *?g i* = *?g j*
  show $i = j$
  proof (*rule ccontr*)
    define *S* **where** *S* = {*0*..<*int CARD*(′*a*)} − {*i*,*j*}
    have *id*: {*0*..<*int CARD*(′*a*)} = (*insert i* (*insert j S*)) **and** *S*: $i \notin S$ $j \notin S$ *finite S*
      using *i j* **unfolding** *S-def* **by** *auto*
    assume *ij*: $i \neq j$
    have *u* = ($\prod$ $i \in$ {*0*..< *int CARD*(′*a*)}. *?g i*) **by** *fact*
    also have ... = *?g i* ∗ *?g j* ∗ ($\prod$ $i \in S$. *?g i*)
      unfolding *id* **using** *S ij* **by** *auto*
    also have ... = *?g i* ∗ *?g i* ∗ ($\prod$ $i \in S$. *?g i*) **unfolding** *gij* **by** *simp*
    finally have *dvd*: *?g i* ∗ *?g i dvd u* **unfolding** *dvd-def* **by** *auto*
      with *sf-u*[*unfolded square-free-def*, *THEN conjunct2*, *rule-format*, *OF gS*(*1*)[*OF i*]]
    show *False* **by** *simp*
  qed
qed

have *u* = ($\prod$ $i \in$ {*0*..< *int CARD*(′*a*)}. *?g i*) **by** *fact*
also have ... = ($\prod$ $i \in$ *?S*. *?g i*)
  by (*rule sym*, *rule prod.setdiff-irrelevant*, *auto*)
also have ... = $\prod$ (*set* (*udivs u*)) **unfolding** *udivs-def set-filter set-map*
    by (*rule sym*, *rule prod.reindex-cong*[*of ?g*, *OF inj* - *refl*], *auto simp*: *int-set*[*symmetric*])
finally have *u-udivs*: *u* = $\prod$ (*set* (*udivs u*)) **.**
{
  fix *w*
  assume *mem*: $w \in$ *set* (*udivs u*)
  then obtain *i* **where** *w*: *w* = *?g i* **and** *i*: $i \in$ *?S*
    unfolding *udivs-def set-filter set-map int-set* **by** *auto*
  have *wu*: *w dvd u* **by** (*simp add*: *w*)
  let *?v* = λ *j*. *v* − [:*of-nat j*:]
  define *j* **where** *j* = *nat i*
from *i* **have** *j*: *of-int i* = (*of-nat j* :: ′*a mod-ring*) $j <$ *CARD*(′*a*) **unfolding** *j-def* **by** *auto*
  from *gS*[*OF i*, *folded w*] **have** ∗: *degree w* > *0 monic w* $w \neq 0$ **by** *auto*
  from *w* **have** *w dvd ?v j* **using** *j* **by** *simp*
  hence *gcdj*: *?gcdn w j* = *w* **by** (*metis gcd.commute gcd-left-idem j*(*1*) *w*)
  {
    fix *j*′
    assume *j*′: *j*′ < *CARD*(′*a*)
    have *?gcdn w j*′ ∈ {*1*,*w*}

proof (*rule ccontr*)
assume *not*: *?gcdn w j'* $\notin$ {*1*,*w*}
with *gcdj* **have** *neq*: *int j'* $\neq$ *int j* **by** *auto*

let *?h* = *?gcdn w j'*
from $*$(*3*) *not* **have** *deg*: *degree ?h* > *0*
using *monic-degree-0 poly-gcd-monic* **by** *auto*
have *hw*: *?h dvd w* **by** *auto*
have *?h dvd ?gcdn u j'* **using** *wu* **using** *dvd-trans* **by** *auto*
also have *?gcdn u j'* = *?g j'* **by** *simp*
finally have *hj'*: *?h dvd ?g j'* **by** *auto*
from *divides-degree*[*OF this*] *deg u0* **have** *degj'*: *degree* (*?g j'*) > *0* **by** *auto*
hence *j'1*: *?g j'* $\neq$ *1* **by** *auto*
with *j'* **have** *mem'*: *?g j'* $\in$ *set* (*udivs u*) **unfolding** *udivs-def* **by** *auto*
from *degj' j'* **have** *j'S*: *int j'* $\in$ *?S* **by** *auto*
from *i j* **have** *jS*: *int j* $\in$ *?S* **by** *auto*
from *inj-on-contraD*[*OF inj neq j'S jS*]
have *neq*: *w* $\neq$ *?g j'* **using** *w j* **by** *auto*
have *cop*: $\neg$ *coprime w* (*?g j'*) **using** *hj' hw deg*
by (*metis coprime-not-unit-not-dvd poly-dvd-1 Nat.neq0-conv*)
obtain *w'* **where** *w'*: *?g j'* = *w'* **by** *auto*
from *u-udivs sf-u* **have** *square-free* ($\prod$(*set* (*udivs u*))) **by** *simp*
from *square-free-prodD*[*OF this finite-set mem mem'*] *cop neq*
show *False* **by** *simp*
qed
}
from *gS*[*OF i*, *folded w*] *i this*
have *degree w* > *0 monic w* $\bigwedge$ *j*. *j* < *CARD*(*'a*) $\Longrightarrow$ *?gcdn w j* $\in$ {*1*,*w*} **by** *auto*
} **note** *udivs* = *this*
let *?is* = *filter* (λ *i*. *?g i* $\neq$ *1*) (*map int* [*0* ..< *CARD*(*'a*)])
have *id*: *udivs u* = *map ?g ?is*
unfolding *udivs-def filter-map o-def* **..**
have *dist*: *distinct* (*udivs u*) **unfolding** *id distinct-map*
proof (*rule conjI*[*OF distinct-filter*], *unfold distinct-map*)
have *?S* = *set ?is* **unfolding** *int-set*[*symmetric*] **by** *auto*
thus *inj-on ?g* (*set ?is*) **using** *inj* **by** *auto*
qed (*auto simp*: *inj-on-def*)
from *u-udivs prod.distinct-set-conv-list*[*OF dist*, *of id*]
have *prod-list* (*udivs u*) = *u* **by** *auto*
note *udivs this dist*
} **note** *udivs* = *this*
have *facts*: *facts* = *concat* (*map udivs divs*)
unfolding *facts-def* **by** *auto*
obtain *lin nonlin* **where** *part*: *List.partition* (λ *q*. *degree q* = *d*) *facts* = (*lin*,*nonlin*)
by *force*
from *Cons*(*6*) **have** *f* = *prod-list us1* $*$ *prod-list divs* **by** *auto*

also have *prod-list divs = prod-list facts* **unfolding** *facts* **using** *udivs(4)*
by (*induct divs, auto*)
finally have *f*: *f = prod-list us1 ∗ prod-list facts* **.**
note *facts′ = facts*
{
fix *u*
assume *u*: *u ∈ set facts*
from *u[unfolded facts]* **obtain** *u′* **where** *u′*: *u′ ∈ set divs* **and** *u*: *u ∈ set (udivs u′)* **by** *auto*
from *u′ udivs(1−2)[OF u′ u] prod-list-dvd[OF u, unfolded udivs(4)[OF u′]]*
have *degree u > 0 monic u ∃ u′ ∈ set divs. u dvd u′* **by** *auto*
} **note** *facts = this*
have *not1*: (*v = 1*) = *False* **using** *False* **by** *auto*
have *us = us1 @ (if length divs = n2 then divs*
else let (lin, nonlin) = List.partition (λq. degree q = d) facts
in lin @ berlekamp-factorization-main d nonlin vs2 (n2 − length lin))
unfolding *Cons(4) facts-def udivs-def′ berlekamp-factorization-main.simps Let-def not1 if-False*
by (*rule arg-cong*[**where** *f = λ x. us1 @ x*], *rule if-cong*, *simp-all*)
hence *res*: *us = us1 @ (if length divs = n2 then divs else*
lin @ berlekamp-factorization-main d nonlin vs2 (n2 − length lin))
unfolding *part* **by** *auto*
show *?thesis*
proof (*cases length divs = n2*)
case *False*
with *res* **have** *us*: *us = (us1 @ lin) @ berlekamp-factorization-main d nonlin vs2 (n2 − length lin)*
by *auto*
from *Cons(2)* **have** *vs*: *vs = (vs1 @ [v]) @ vs2* **by** *auto*
have *f*: *f = prod-list ((us1 @ lin) @ nonlin)*
unfolding *f* **using** *prod-list-partition[OF part]* **by** *simp*
{
fix *u*
assume *u ∈ set ((us1 @ lin) @ nonlin)*
with *part* **have** *u ∈ set facts ∪ set us1* **by** *auto*
with *facts Cons(7)* **have** *degree u > 0* **by** (*auto simp*: *irreducible-degree-field*)
} **note** *deg = this*
from *berlekamp-basis-length-factorization[OF sf-f n-bb mon-f f deg, unfolded Cons(3)]*
have *n2 ≥ length lin* **by** *auto*
hence *n*: *n = length (us1 @ lin) + (n2 − length lin)*
unfolding *Cons(3)* **by** *auto*
show *?thesis*
proof (*rule Cons(1)[OF vs n us - f]*)
fix *u*
assume *u ∈ set nonlin*
with *part* **have** *u ∈ set facts* **by** *auto*
from *facts[OF this]* **show** *monic u ∧ degree u > 0* **by** *auto*
next

fix *u*
assume *u*: $u \in$ *set* (*us1* @ *lin*)
{
assume $*$: $\neg$ (*monic u* $\wedge$ $irreducible_d$ *u*)
with *Cons*(*7*) *u* **have** $u \in$ *set lin* **by** *auto*
with *part* **have** *uf*: $u \in$ *set facts* **and** *deg*: *degree* $u = d$ **by** *auto*
from *facts*[*OF uf*] **obtain** u' **where** $u' \in$ *set divs* **and** uu': *u dvd* u' **by** *auto*
from *this*(*1*) **have** u' *dvd f* **unfolding** *Cons*(*6*) **using** *prod-list-dvd*[*of* u'] **by** *auto*
with uu' **have** *u dvd f* **by** (*rule dvd-trans*)
from *facts*[*OF uf*] *d*[*OF this deg*] $*$ **have** *False* **by** *auto*
}
thus *monic u* $\wedge$ *irreducible u* **by** *auto*
next
fix *w u i*
assume *w*: $w \in$ *set* (*vs1* @ [*v*])
and *u*: $u \in$ *set* (*us1* @ *lin*) $\cup$ *set nonlin*
and *i*: $i <$ *CARD*($'a$)
from *u part* **have** *u*: $u \in$ *set us1* $\cup$ *set facts* **by** *auto*
show *gcd u* ($w -$ [:*of-nat i*:]) $\in \{1,\ u\}$
proof (*cases* $u \in$ *set us1*)
case *True*
from *Cons*(*7*)[*OF this*] **have** *monic u irreducible u* **by** *auto*
thus *?thesis* **by** (*rule monic-irreducible-gcd*)
next
case *False*
with *u* **have** *u*: $u \in$ *set facts* **by** *auto*
show *?thesis*
proof (*cases* $w = v$)
case *True*
from *u*[*unfolded facts'*] **obtain** u' **where** *u*: $u \in$ *set* (*udivs* u')
and u': $u' \in$ *set divs* **by** *auto*
from *udivs*(*3*)[*OF* u' *u i*] **show** *?thesis* **unfolding** *True* **.**
next
case *False*
with *w* **have** *w*: $w \in$ *set vs1* **by** *auto*
from *u* **obtain** u' **where** u': $u' \in$ *set divs* **and** *dvd*: *u dvd* u'
using *facts*(*3*)[*of u*] *dvd-refl*[*of u*] **by** *blast*
from *w* **have** $w \in$ *set vs1* $\vee$ $w = v$ **by** *auto*
from *facts*(*1*$-$*2*)[*OF u*] **have** *u*: *monic u* **by** *auto*
from *Cons*(*8*)[*OF w - i*] u'
have *gcd* u' ($w -$ [:*of-nat i*:]) $\in \{1,\ u'\}$ **by** *auto*
with *dvd u* **show** *?thesis* **by** (*rule monic-gcd-dvd*)
qed
qed
qed
next
case *True*

```
        with res have us: us = us1 @ divs by auto
        from Cons(3) True have n: n = length us unfolding us by auto
        show ?thesis unfolding us[symmetric]
        proof (intro conjI ballI)
          show f: f = prod-list us unfolding us using Cons(6) by simp
          {
            fix u
            assume u ∈ set us
            hence degree u > 0 using Cons(5) Cons(7)[unfolded irreducible_d-def]
              unfolding us by (auto simp: irreducible-degree-field)
          } note deg = this
          fix u
          assume u: u ∈ set us
          thus monic u unfolding us using Cons(5) Cons(7) by auto
          show irreducible u
             by (rule berlekamp-basis-irreducible[OF sf-f n-bb mon-f f n[symmetric]
deg u])
        qed
      qed
    next
      case True
      with Cons(4) have us: us = us1 @ berlekamp-factorization-main d divs vs2
n2 by simp
      from Cons(2) True have vs: vs = (vs1 @ [1]) @ vs2 by auto
      show ?thesis
      proof (rule Cons(1)[OF vs Cons(3) us Cons(5−7)], goal-cases)
        case (3 v u i)
        show ?case
        proof (cases v = 1)
          case False
          with 3 Cons(8)[of v u i] show ?thesis by auto
        next
          case True
          hence deg: degree (v − [: of-nat i :]) = 0
             by (metis (no-types, opaque-lifting) degree-pCons-0 diff-pCons diff-zero
pCons-one)
          from 3(2) Cons(5,7)[of u] have monic u by auto
          from gcd-monic-constant[OF this deg] show ?thesis .
        qed
      qed
    qed
  next
    case Nil
    with vsf have vs1: vs1 = berlekamp-basis f by auto
    from Nil(3) have us: us = us1 @ divs by auto
    from Nil(4,6) have md: ⋀ u. u ∈ set us ⟹ monic u ∧ degree u > 0
      unfolding us by (auto simp: irreducible-degree-field)
    from Nil(7)[unfolded vs1] us
    have no-further-splitting-possible:
```

$\bigwedge$ *u v i. v* ∈ *set* (*berlekamp-basis f*) ⟹ *u* ∈ *set us*
⟹ *i* < *CARD*(*'a*) ⟹ *gcd u* (*v* − [:*of-nat i*:]) ∈ {*1*, *u*} **by** *auto*
from *Nil*(*5*) *us* **have** *prod*: *f* = *prod-list us* **by** *simp*
show *?case*
proof (*intro conjI ballI*)
fix *u*
assume *u*: *u* ∈ *set us*
from *md*[*OF this*] **have** *mon-u*: *monic u* **and** *deg-u*: *degree u* > *0* **by** *auto*
from *prod u* **have** *uf*: *u dvd f* **by** (*simp add*: *prod-list-dvd*)
from *monic-square-free-irreducible-factorization*[*OF mon-f sf-f*] **obtain** *P* **where**
P: *finite P f* = ∏ *P P* ⊆ {*q. irreducible q* ∧ *monic q*} **by** *auto*
show *irreducible u*
proof (*rule ccontr*)
assume *irr-u*: ¬ *irreducible u*
from *not-irreducible-factor-yields-prime-factors*[*OF uf P deg-u this*]
obtain *pi pj* **where** *pij*: *pi* ∈ *P pj* ∈ *P pi* ≠ *pj pi dvd u pj dvd u* **by** *blast*
from *exists-vector-in-Berlekamp-basis-dvd*[*OF*
deg-f berlekamp-basis-basis[*OF deg-f*, *folded vs1*] *finite-set*
P pij(*1*−*3*) *mon-f sf-f irr-u uf mon-u pij*(*4*−*5*), *unfolded vs1*]
obtain *v s* **where** *v*: *v* ∈ *set* (*berlekamp-basis f*)
and *gcd*: *gcd u* (*v* − [:*s*:]) ∉ {*1*,*u*} **using** *is-unit-gcd* **by** *auto*
from *surj-of-nat-mod-ring*[*of s*] **obtain** *i* **where** *i*: *i* < *CARD*(*'a*) **and** *s*: *s* = *of-nat i* **by** *auto*
from *no-further-splitting-possible*[*OF v u i*] *gcd*[*unfolded s*]
show *False* **by** *auto*
qed
qed (*insert prod md*, *auto*)
qed
qed

lemma *berlekamp-monic-factorization*:
fixes *f*::*'a mod-ring poly*
assumes *sf-f*: *square-free f*
and *us*: *berlekamp-monic-factorization d f* = *us*
and *d*: $\bigwedge$ *g. g dvd f* ⟹ *degree g* = *d* ⟹ *irreducible g*
and *deg*: *degree f* > *0*
and *mon*: *monic f*
shows *f* = *prod-list us* ∧ (∀ *u* ∈ *set us. monic u* ∧ *irreducible u*)
proof −
from *us*[*unfolded berlekamp-monic-factorization-def Let-def*] *deg*
have *us*: *us* = [] @ *berlekamp-factorization-main d* [*f*] (*berlekamp-basis f*) (*length* (*berlekamp-basis f*))
by (*auto*)
have *id*: *berlekamp-basis f* = [] @ *berlekamp-basis f*
length (*berlekamp-basis f*) = *length* [] + *length* (*berlekamp-basis f*)
f = *prod-list* ([] @ [*f*])
by *auto*
show *f* = *prod-list us* ∧ (∀ *u* ∈ *set us. monic u* ∧ *irreducible u*)

by (*rule berlekamp-factorization-main*[*OF sf-f id*(*1*) *refl refl id*(*2*) *us - - - id*(*3*)], *insert mon deg d*, *auto*)
qed
end

end

7 Distinct Degree Factorization

theory *Distinct-Degree-Factorization*
imports
Finite-Field
Polynomial-Factorization.Square-Free-Factorization
Berlekamp-Type-Based
begin

definition *factors-of-same-degree* :: *nat* ⇒ *'a* :: *field poly* ⇒ *bool* **where**
factors-of-same-degree i f = (*i* ≠ *0* ∧ *degree f* ≠ *0* ∧ *monic f* ∧ (∀ *g*. *irreducible g* ⟶ *g dvd f* ⟶ *degree g* = *i*))

lemma *factors-of-same-degreeD*: **assumes** *factors-of-same-degree i f*
shows *i* ≠ *0 degree f* ≠ *0 monic f g dvd f* ⟹ *irreducible g* = (*degree g* = *i*)
proof −
note ∗ = *assms*[*unfolded factors-of-same-degree-def*]
show *i*: *i* ≠ *0* **and** *f*: *degree f* ≠ *0 monic f* **using** ∗ **by** *auto*
assume *gf*: *g dvd f*
with ∗ **have** *irreducible g* ⟹ *degree g* = *i* **by** *auto*
moreover
{
assume ∗∗: *degree g* = *i* ¬ *irreducible g*
with *irreducible*$_d$*-factor*[*of g*] *i* **obtain** *h1 h2* **where** *irr*: *irreducible h1* **and** *gh*: *g* = *h1* ∗ *h2*
and *deg-h2*: *degree h2* < *degree g* **by** *auto*
from ∗∗ *i* **have** *g0*: *g* ≠ *0* **by** *auto*
from *gf gh g0* **have** *h1 dvd f* **using** *dvd-mult-left* **by** *blast*
from ∗ *f this irr* **have** *deg-h*: *degree h1* = *i* **by** *auto*
from *arg-cong*[*OF gh*, *of degree*] *g0* **have** *degree g* = *degree h1* + *degree h2*
by (*simp add*: *degree-mult-eq gh*)
with ∗∗(*1*) *deg-h* **have** *degree h2* = *0* **by** *auto*
from *degree0-coeffs*[*OF this*] **obtain** *c* **where** *h2*: *h2* = [:*c*:] **by** *auto*
with *gh g0* **have** *g*: *g* = *smult c h1 c* ≠ *0* **by** *auto*
with *irr* ∗∗(*2*) *irreducible-smult-field*[*of c h1*] **have** *False* **by** *auto*
}
ultimately show *irreducible g* = (*degree g* = *i*) **by** *auto*
qed

hide-const *order*

hide-const *up-ring.monom*

theorem (**in** *field*) *finite-field-mult-group-has-gen2*:
assumes *finite*:*finite* (*carrier R*)
shows $\exists\, a \in$ *carrier* (*mult-of R*). *group.ord* (*mult-of R*) $a =$ *order* (*mult-of R*) $\wedge$ *carrier* (*mult-of R*) $= \{a[\widehat{\ }]i \mid i{::}nat\ .\ i \in UNIV\}$
proof −
note *mult-of-simps*[*simp*]
have *finite'*: *finite* (*carrier* (*mult-of R*)) **using** *finite* **by** (*rule finite-mult-of*)

interpret *G*: *group mult-of R* **rewrites**
$([\widehat{\ }]_{mult\text{-}of\ R}) = (([\widehat{\ }]) :: - \Rightarrow nat \Rightarrow -)$ **and** $\mathbf{1}_{mult\text{-}of\ R} = \mathbf{1}$
by (*rule field-mult-group*) (*simp-all add*: *fun-eq-iff nat-pow-def*)

let $?N = \lambda\ x\ .\ card\ \{a \in carrier\ (mult\text{-}of\ R).\ group.ord\ (mult\text{-}of\ R)\ a\ = x\}$
have $0 < order\ R - 1$ **unfolding** *Coset.order-def* **using** *card-mono*[*OF finite*, *of* {**0**, **1**}] **by** *simp*
then have $*$: $0 < order\ (mult\text{-}of\ R)$ **using** *assms* **by** (*simp add*: *order-mult-of*)
have *fin*: *finite* {*d*. *d dvd order* (*mult-of R*) } **using** *dvd-nat-bounds*[*OF* $*$] **by** *force*

have $(\sum d \mid d\ dvd\ order\ (mult\text{-}of\ R).\ ?N\ d)$
$= card\ (UN\ d{:}\{d\ .\ d\ dvd\ order\ (mult\text{-}of\ R)\ \}.\ \{a \in carrier\ (mult\text{-}of\ R).\ group.ord\ (mult\text{-}of\ R)\ a\ = d\})$
(**is** - = *card ?U*)
using *fin finite* **by** (*subst card-UN-disjoint*) *auto*
also have *?U* = *carrier* (*mult-of R*)
proof
{ **fix** *x* **assume** *x*:$x \in carrier\ (mult\text{-}of\ R)$
hence *x'*:$x{\in}carrier\ (mult\text{-}of\ R)$ **by** *simp*
then have *group.ord* (*mult-of R*) *x dvd order* (*mult-of R*)
using *finite' G.ord-dvd-group-order*[*OF x'*] **by** (*simp add*: *order-mult-of*)
hence $x \in ?U$ **using** *dvd-nat-bounds*[*of order* (*mult-of R*) *group.ord* (*mult-of R*) *x*] *x* **by** *blast*
} **thus** *carrier* (*mult-of R*) $\subseteq$ *?U* **by** *blast*
qed *auto*
also have *card* ... = *Coset.order* (*mult-of R*)
using *order-mult-of finite'* **by** (*simp add*: *Coset.order-def*)
finally have *sum-Ns-eq*: $(\sum d \mid d\ dvd\ order\ (mult\text{-}of\ R).\ ?N\ d) = order\ (mult\text{-}of\ R)$.

{ **fix** *d* **assume** *d*:*d dvd order* (*mult-of R*)
have $card\ \{a \in carrier\ (mult\text{-}of\ R).\ group.ord\ (mult\text{-}of\ R)\ a = d\} \le phi'\ d$
proof *cases*
assume $card\ \{a \in carrier\ (mult\text{-}of\ R).\ group.ord\ (mult\text{-}of\ R)\ a = d\} = 0$ **thus** *?thesis* **by** *presburger*
next
assume $card\ \{a \in carrier\ (mult\text{-}of\ R).\ group.ord\ (mult\text{-}of\ R)\ a = d\} \neq 0$

hence $\exists\, a \in$ *carrier* (*mult-of R*). *group.ord* (*mult-of R*) $a = d$ **by** (*auto simp*: *card-eq-0-iff*)
thus *?thesis* **using** *num-elems-of-ord-eq-phi'*[*OF finite d*] **by** *auto*
qed
}
hence *all-le*:$\bigwedge i.\ i \in \{d.\ d\ dvd\ order\ (mult\text{-}of\ R)\ \}$
$\Longrightarrow (\lambda i.\ card\ \{a \in carrier\ (mult\text{-}of\ R).\ group.ord\ (mult\text{-}of\ R)\ a = i\})\ i \leq (\lambda i.\ phi'\ i)\ i$ **by** *fast*
hence *le*:$(\sum i \mid i\ dvd\ order\ (mult\text{-}of\ R).\ ?N\ i)$
$\leq (\sum i \mid i\ dvd\ order\ (mult\text{-}of\ R).\ phi'\ i)$
using *sum-mono*[*of* $\{d\ .\ \ d\ dvd\ order\ (mult\text{-}of\ R)\}$
$\lambda i.\ card\ \{a \in carrier\ \ (mult\text{-}of\ R).\ group.ord\ (mult\text{-}of\ R)\ a = i\}$] **by** *presburger*
have *order* (*mult-of R*) $= (\sum d \mid d\ dvd\ order\ (mult\text{-}of\ R).\ phi'\ d)$ **using** $*$
by (*simp add*: *sum-phi'-factors*)
hence *eq*:$(\sum i \mid i\ dvd\ order\ (mult\text{-}of\ R).\ ?N\ i)$
$= (\sum i \mid i\ dvd\ order\ (mult\text{-}of\ R).\ phi'\ i)$ **using** *le sum-Ns-eq* **by** *presburger*
have $\bigwedge i.\ i \in \{d.\ d\ dvd\ order\ (mult\text{-}of\ R)\ \} \Longrightarrow ?N\ i = (\lambda i.\ phi'\ i)\ i$
proof (*rule ccontr*)
fix *i*
assume *i1*:$i \in \{d.\ d\ dvd\ order\ (mult\text{-}of\ R)\}$ **and** $?N\ i \neq phi'\ i$
hence $?N\ i = 0$
using *num-elems-of-ord-eq-phi'*[*OF finite*, *of i*] **by** (*auto simp*: *card-eq-0-iff*)
moreover **have** $0 < i$ **using** $*$ *i1* **by** (*simp add*: *dvd-nat-bounds*[*of order* (*mult-of R*) *i*])
ultimately have $?N\ i < phi'\ i$ **using** *phi'-nonzero* **by** *presburger*
hence $(\sum i \mid i\ dvd\ order\ (mult\text{-}of\ R).\ ?N\ i)$
$< (\sum i \mid i\ dvd\ order\ (mult\text{-}of\ R).\ phi'\ i)$
using *sum-strict-mono-ex1*[*OF fin*, *of ?N* $\lambda\ i\ .\ phi'\ i$]
i1 all-le **by** *auto*
thus *False* **using** *eq* **by** *force*
qed
hence *?N* (*order* (*mult-of R*)) > 0 **using** $*$ **by** (*simp add*: *phi'-nonzero*)
then obtain *a* **where** *a*:$a \in$ *carrier* (*mult-of R*) **and** *a-ord*:*group.ord* (*mult-of R*) $a =$ *order* (*mult-of R*)
by (*auto simp add*: *card-gt-0-iff*)
hence *set-eq*:$\{a[\widehat{\ }]i \mid i{::}nat.\ i \in UNIV\} = (\lambda x.\ a[\widehat{\ }]x)\ `\ \{0\ ..\ group.ord\ (mult\text{-}of\ R)\ a - 1\}$
using *G.ord-elems*[*OF finite'*] **by** *auto*
have *card-eq*:$card\ ((\lambda x.\ a[\widehat{\ }]x)\ `\ \{0\ ..\ group.ord\ (mult\text{-}of\ R)\ a - 1\}) = card\ \{0\ ..\ group.ord\ (mult\text{-}of\ R)\ a - 1\}$
by (*intro card-image G.ord-inj finite' a*)
hence $card\ ((\lambda\ x\ .\ a[\widehat{\ }]x)\ `\ \{0\ ..\ group.ord\ (mult\text{-}of\ R)\ a - 1\}) = card\ \{0\ ..order\ (mult\text{-}of\ R) - 1\}$
using *assms* **by** (*simp add*: *card-eq a-ord*)
hence *card-R-minus-1*:$card\ \{a[\widehat{\ }]i \mid i{::}nat.\ i \in UNIV\} =\ order\ (mult\text{-}of\ R)$
using $*$ **by** (*subst set-eq*) *auto*
have $**$:$\{a[\widehat{\ }]i \mid i{::}nat.\ i \in UNIV\} \subseteq carrier\ (mult\text{-}of\ R)$
using *G.nat-pow-closed*[*OF a*] **by** *auto*

with - **have** *carrier* (*mult-of R*) = $\{a[\widehat{\ }]i|i::nat.\ i \in UNIV\}$
by (*rule card-seteq*[*symmetric*]) (*simp-all add*: *card-R-minus-1 finite Coset.order-def del*: *UNIV-I*)
thus *?thesis* **using** *a a-ord* **by** *blast*
qed

lemma *add-power-prime-poly-mod-ring*[*simp*]:
fixes x :: $'a$::{*prime-card*} *mod-ring poly*
shows $(x + y)\ \widehat{}\ CARD('a)\widehat{}n = x\ \widehat{}\ (CARD('a)\widehat{}n) + y\ \widehat{}\ CARD('a)\widehat{}n$
proof (*induct n arbitrary*: *x y*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n*)
define *p* **where** *p*: $p = CARD('a)$
have $(x + y)\ \widehat{}\ p\ \widehat{}\ Suc\ n = (x + y)\ \widehat{}\ (p * p\widehat{}n)$ **by** *simp*
also have ... $= ((x + y)\ \widehat{}\ p)\ \widehat{}\ (p\widehat{}n)$
by (*simp add*: *power-mult*)
also have ... $= (x\widehat{}p + y\widehat{}p)\widehat{}\ (p\widehat{}n)$
by (*simp add*: *add-power-poly-mod-ring p*)
also have ... $= (x\widehat{}p)\widehat{}(p\widehat{}n) + (y\widehat{}p)\widehat{}(p\widehat{}n)$ **using** *Suc.hyps* **unfolding** *p* **by** *auto*
also have ... $= x\widehat{}(p\widehat{}(n+1)) + y\widehat{}(p\widehat{}(n+1))$ **by** (*simp add*: *power-mult*)
finally show *?case* **by** (*simp add*: *p*)
qed

lemma *fermat-theorem-mod-ring2*[*simp*]:
fixes $a::'a$::{*prime-card*} *mod-ring*
shows $a\ \widehat{}\ (CARD('a)\widehat{}n) = a$
proof (*induct n arbitrary*: *a*)
case (*Suc n*)
define *p* **where** $p = CARD('a)$
have $a\ \widehat{}\ p\ \widehat{}\ Suc\ n = a\ \widehat{}\ (p * (p\ \widehat{}\ n))$ **by** *simp*
also have ... $= (a\ \widehat{}\ p)\ \widehat{}(p\ \widehat{}\ n)$ **by** (*simp add*: *power-mult*)
also have ... $= a\widehat{}(p\ \widehat{}\ n)$ **using** *fermat-theorem-mod-ring*[*of* $a\widehat{}p$] **unfolding** *p-def* **by** *auto*
also have ... $= a$ **using** *Suc.hyps p-def* **by** *auto*
finally show *?case* **by** (*simp add*: *p-def*)
qed *auto*

lemma *fermat-theorem-power-poly*[*simp*]:
fixes $a::'a$::*prime-card mod-ring*
shows [:*a*:] $\widehat{}\ CARD('a$::*prime-card*$)\ \widehat{}\ n$ = [:*a*:]
by (*auto simp add*: *Missing-Polynomial.poly-const-pow mod-poly-less*)

lemma *degree-prod-monom*: *degree* $(\prod i = 0..<n.\ monom\ 1\ 1) = n$
  by (*metis degree-monom-eq prod-pow x-pow-n zero-neq-one*)

lemma *degree-monom0*[*simp*]: *degree* (*monom a 0*) = *0* **using** *degree-monom-le*
by *auto*
lemma *degree-monom0'*[*simp*]: *degree* (*monom 0 b*) = *0* **by** *auto*

lemma *sum-monom-mod*:
  assumes $b <$ *degree f*
  shows $(\sum i{\le}b.\ monom\ (g\ i)\ i)\ mod\ f = (\sum i{\le}b.\ monom\ (g\ i)\ i)$
  using *assms*
proof (*induct b*)
  case *0*
  then show *?case* **by** (*auto simp add*: *mod-poly-less*)
next
  case (*Suc b*)
  have *hyp*: $(\sum i{\le}b.\ monom\ (g\ i)\ i)\ mod\ f = (\sum i{\le}b.\ monom\ (g\ i)\ i)$
    using *Suc.prems Suc.hyps* **by** *simp*
  have *rw-monom*: *monom* (*g* (*Suc b*)) (*Suc b*) *mod f* = *monom* (*g* (*Suc b*)) (*Suc*
b)
   by (*metis Suc.prems degree-monom-eq mod-0 mod-poly-less monom-hom.hom-0-iff*)
  have *rw*: $(\sum i{\le}Suc\ b.\ monom\ (g\ i)\ i) = (monom\ (g\ (Suc\ b))\ (Suc\ b) + (\sum i{\le}b.$
$monom\ (g\ i)\ i))$
    by *auto*
  have $(\sum i{\le}Suc\ b.\ monom\ (g\ i)\ i)\ mod\ f$
    $= (monom\ (g\ (Suc\ b))\ (Suc\ b) + (\sum i{\le}b.\ monom\ (g\ i)\ i))\ mod\ f$ **using** *rw* **by**
presburger
  also have ... $=((monom\ (g\ (Suc\ b))\ (Suc\ b))\ mod\ f) + ((\sum i{\le}b.\ monom\ (g\ i)$
$i)\ mod\ f)$
    using *poly-mod-add-left* **by** *auto*
  also have ... $= monom\ (g\ (Suc\ b))\ (Suc\ b) + (\sum i{\le}b.\ monom\ (g\ i)\ i)$
    using *hyp rw-monom* **by** *presburger*
  also have ... $= (\sum i{\le}Suc\ b.\ monom\ (g\ i)\ i)$ **using** *rw* **by** *auto*
  finally show *?case* **.**
qed

lemma *x-power-aq-minus-1-rw*:
  fixes *x*::*nat*
  assumes *x*: $x > 1$
    and *a*: $a > 0$
    and *b*: $b > 0$
  shows $x\ \hat{}\ (a * q) - 1 = ((x\hat{}a) - 1) * sum\ ((\hat{})\ (x\hat{}a))\ \{..<q\}$
proof −
  have *xa*: $(x\ \hat{}\ a) > 0$ **using** *x* **by** *auto*
  have *int-rw1*: $int\ (x\ \hat{}\ a) - 1 = int\ ((x\ \hat{}\ a) - 1)$
    using *xa* **by** *linarith*
  have *int-rw2*: $sum\ ((\hat{})\ (int\ (x\ \hat{}\ a)))\ \{..<q\} = int\ (sum\ ((\hat{})\ ((x\ \hat{}\ a)))\ \{..<q\})$
    unfolding *int-sum* **by** *simp*
  have $int\ (x\ \hat{}\ a)\ \hat{}\ q = int\ (Suc\ ((x\ \hat{}\ a)\ \hat{}\ q - 1))$ **using** *xa* **by** *auto*

hence *int* $((x \hat{}\ a) \hat{}\ q - 1) = int\ (x \hat{}\ a) \hat{}\ q - 1$ **using** *xa* **by** *presburger*
also have ... $= (int\ (x \hat{}\ a) - 1) * sum\ ((\hat{})\ (int\ (x \hat{}\ a)))\ \{..<q\}$
by (*rule power-diff-1-eq*)
also have ... $= (int\ ((x \hat{}\ a) - 1)) * int\ (sum\ ((\hat{})\ (\ (x \hat{}\ a)))\ \{..<q\})$
unfolding *int-rw1 int-rw2* **by** *simp*
also have ... $= int\ (((x \hat{}\ a) - 1) * (sum\ ((\hat{})\ (\ (x \hat{}\ a)))\ \{..<q\}))$ **by** *auto*
finally have *aux*: $int\ ((x \hat{}\ a) \hat{}\ q - 1) = int\ (((x \hat{}\ a) - 1) * sum\ ((\hat{})\ (x \hat{}\ a))\ \{..<q\})$ **.**
have $x \hat{}(a * q) - 1 = (x\hat{}a)\hat{}q - 1$
by (*simp add*: *power-mult*)
also have ... $= ((x\hat{}a) - 1) * sum\ ((\hat{})\ (x\hat{}a))\ \{..<q\}$
using *aux* **unfolding** *int-int-eq* **.**
finally show *?thesis* **.**
qed

lemma *dvd-power-minus-1-conv1*:
fixes *x*::*nat*
assumes *x*: $x > 1$
and *a*: $a > 0$
and *xa-dvd*: $x \hat{}\ a - 1\ dvd\ x\hat{}b - 1$
and *b0*: $b > 0$
shows *a dvd b*
proof −
define *r* **where** *r*[*simp*]: $r = b\ mod\ a$
define *q* **where** *q*[*simp*]: $q = b\ div\ a$
have *b*: $b = a * q + r$ **by** *auto*
have *ra*: $r < a$ **by** (*simp add*: *a*)
hence *xr-less-xa*: $x \hat{}\ r - 1 < x \hat{}\ a - 1$
using *x power-strict-increasing-iff diff-less-mono x* **by** *simp*
have *dvd*: $x \hat{}\ a - 1\ dvd\ x \hat{}(a * q) - 1$
using *x-power-aq-minus-1-rw*[*OF x a b0*] **unfolding** *dvd-def* **by** *auto*
have $x\hat{}b - 1 = x\hat{}b - x\hat{}r + x\hat{}r - 1$
using *assms*(*1*) *assms*(*4*) **by** *auto*
also have ... $= x\hat{}r * (x\hat{}(a*q) - 1) + x\hat{}r - 1$
by (*metis* (*no-types*, *lifting*) *b diff-mult-distrib2 mult.commute nat-mult-1-right power-add*)
finally have $x\hat{}b - 1 = x\hat{}r * (x\hat{}(a*q) - 1) + x\hat{}r - 1$ **.**
hence $x \hat{}\ a - 1\ dvd\ x \hat{}\ r * (x \hat{}(a * q) - 1) + x \hat{}\ r - 1$ **using** *xa-dvd* **by** *presburger*
hence $x\hat{}a - 1\ dvd\ x\hat{}r - 1$
by (*metis* (*no-types*) *diff-add-inverse diff-commute dvd dvd-diff-nat dvd-trans dvd-triv-right*)
hence $r = 0$
using *xr-less-xa*
by (*meson nat-dvd-not-less neq0-conv one-less-power x zero-less-diff*)
thus *?thesis* **by** *auto*
qed

lemma *dvd-power-minus-1-conv2*:
  fixes x::*nat*
  assumes x: $x > 1$
    and a: $a > 0$
    and *a-dvd-b*: a *dvd* b
    and *b0*: $b > 0$
  shows x ^ $a - 1$ *dvd* x^$b - 1$
proof −
  define q **where** q[*simp*]: $q = b$ *div* a
  have b: $b = a * q$ **using** *a-dvd-b* **by** *auto*
  have x^$b - 1 = ((x$ ^ $a) - 1) *$ *sum* $(($^$)$ $(x$ ^ $a))$ $\{..<q\}$
    unfolding b **by** (*rule x-power-aq-minus-1-rw*[*OF x a b0*])
  thus *?thesis* **unfolding** *dvd-def* **by** *auto*
qed

corollary *dvd-power-minus-1-conv*:
  fixes x::*nat*
  assumes x: $x > 1$
    and a: $a > 0$
    and *b0*: $b > 0$
  shows a *dvd* $b = (x$ ^ $a - 1$ *dvd* x^$b - 1)$
  using *assms dvd-power-minus-1-conv1 dvd-power-minus-1-conv2* **by** *blast*

locale *poly-mod-type-irr* = *poly-mod-type m TYPE*('*a*::*prime-card*) **for** *m* +
  fixes *f*::'*a*::{*prime-card*} *mod-ring poly*
  assumes *irr-f*: *irreducible*$_d$ *f*
begin

definition *plus-irr* :: '*a mod-ring poly* ⇒'*a mod-ring poly* ⇒ '*a mod-ring poly*
  where *plus-irr a b* = (*a* + *b*) *mod f*

definition *minus-irr* :: '*a mod-ring poly* ⇒'*a mod-ring poly* ⇒ '*a mod-ring poly*
  where *minus-irr x y* ≡ (*x* − *y*) *mod f*

definition *uminus-irr* :: '*a mod-ring poly* ⇒'*a mod-ring poly*
  where *uminus-irr x* = −*x*

definition *mult-irr* :: '*a mod-ring poly* ⇒'*a mod-ring poly* ⇒ '*a mod-ring poly*
  where *mult-irr x y* = ((*x*∗*y*) *mod f*)

definition *carrier-irr* :: '*a mod-ring poly set*
  where *carrier-irr* = {*x*. *degree x* < *degree f*}

definition *power-irr* :: '*a mod-ring poly* ⇒ *nat* ⇒ '*a mod-ring poly*

where *power-irr p n* = ((*p*^*n*) *mod f*)

definition *R* = (|*carrier* = *carrier-irr*, *monoid.mult* = *mult-irr*, *one* = *1*, *zero* = *0*, *add* = *plus-irr*|)

lemma *degree-f*[*simp*]: *degree f* > *0*
  using *irr-f* $irreducible_d D(1)$ **by** *blast*

lemma *element-in-carrier*: (*a* ∈ *carrier R*) = (*degree a* < *degree f*)
  unfolding *R-def carrier-irr-def* **by** *auto*

lemma *f-dvd-ab*:
  a = *0* ∨ *b* = *0* **if** *f dvd a* ∗ *b*
    and *a*: *degree a* < *degree f*
    and *b*: *degree b* < *degree f*
proof (*rule ccontr*)
  assume ¬ (*a* = *0* ∨ *b* = *0*)
  then have *a* ≠ *0* **and** *b* ≠ *0*
    by *simp-all*
  with *a b* **have** ¬ *f dvd a* **and** ¬ *f dvd b*
    by (*auto simp add*: *mod-poly-less dvd-eq-mod-eq-0*)
  moreover from ‹*f dvd a* ∗ *b*› *irr-f* **have** *f dvd a* ∨ *f dvd b*
    by *auto*
  ultimately show *False*
    by *simp*
qed

lemma *ab-mod-f0*:
  a = *0* ∨ *b* = *0* **if** *a* ∗ *b mod f* = *0*
    and *a*: *degree a* < *degree f*
    and *b*: *degree b* < *degree f*
  using *that f-dvd-ab* **by** *auto*

lemma $irreducible_d D2$:
  fixes *p q* :: ′*b*::{*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
  assumes $irreducible_d$ *p*
  and *degree q* < *degree p* **and** *degree q* ≠ *0*
  shows ¬ *q dvd p*
  using *assms* $irreducible_d$-*dvd-smult* **by** *force*

lemma *times-mod-f-1-imp-0*:
  assumes *x*: *degree x* < *degree f*
    and *x2*: ∀ *xa*. *x* ∗ *xa mod f* = *1* ⟶ ¬ *degree xa* < *degree f*
  shows *x* = *0*
proof (*rule ccontr*)
  assume *x3*: *x* ≠ *0*
  let *?u* = *fst* (*bezout-coefficients f x*)
  let *?v* = *snd* (*bezout-coefficients f x*)

have *?u* ∗ *f* + *?v* ∗ *x* = *gcd f x* **using** *bezout-coefficients-fst-snd* **by** *auto*
also have ... = *1*
proof (*rule ccontr*)
assume *g*: *gcd f x* ≠ *1*
have *degree* (*gcd f x*) < *degree f*
by (*metis degree-0 dvd-eq-mod-eq-0 gcd-dvd1 gcd-dvd2 irr-f irreducible$_d$D*(*1*) *mod-poly-less nat-neq-iff x x3*)
have ¬ *gcd f x dvd f*
proof (*rule irreducible$_d$D2*[*OF irr-f*])
show *degree* (*gcd f x*) < *degree f*
by (*metis degree-0 dvd-eq-mod-eq-0 gcd-dvd1 gcd-dvd2 irr-f irreducible$_d$D*(*1*) *mod-poly-less nat-neq-iff x x3*)
show *degree* (*gcd f x*) ≠ *0*
by (*metis* (*no-types, opaque-lifting*) *g degree-mod-less′ gcd.bottom-left-bottom gcd-eq-0-iff gcd-left-idem gcd-mod-left gr-implies-not0 x*)
qed
moreover have *gcd f x dvd f* **by** *auto*
ultimately show *False* **by** *contradiction*
qed
finally have *?v*∗*x mod f* = *1*
by (*metis degree-1 degree-f mod-mult-self3 mod-poly-less*)
hence (*x*∗(*?v mod f*)) *mod f* = *1*
by (*simp add*: *mod-mult-right-eq mult.commute*)
moreover have *degree* (*?v mod f*) < *degree f*
by (*metis degree-0 degree-f degree-mod-less′ not-gr-zero*)
ultimately show *False* **using** *x2* **by** *auto*
qed

sublocale *field-R*: *field R*
proof −
have ∗: ∃ *y*. *degree y* < *degree f* ∧ *f dvd x* + *y* **if** *degree x* < *degree f*
for *x* :: ′*a mod-ring poly*
proof −
from *that* **have** *degree* (− *x*) < *degree f*
by *simp*
moreover have *f dvd* (*x* + − *x*)
by *simp*
ultimately show *?thesis*
by *blast*
qed
have ∗∗: *degree* (*x* ∗ *y mod f*) < *degree f*
if *degree x* < *degree f* **and** *degree y* < *degree f*
for *x y* :: ′*a mod-ring poly*
using *that* **by** (*cases x* = *0* ∨ *y* = *0*)
(*auto intro*: *degree-mod-less′ dest*: *f-dvd-ab*)
show *field R*
by *standard* (*auto simp add*: *R-def carrier-irr-def plus-irr-def mult-irr-def Units-def algebra-simps degree-add-less mod-poly-less mod-add-eq mult-poly-add-left*

mod-mult-left-eq mod-mult-right-eq mod-eq-0-iff-dvd ab-mod-f0 $*$ $**$ *dest*: *times-mod-f-1-imp-0*)
qed

lemma *zero-in-carrier*[*simp*]: *0* ∈ *carrier-irr* **unfolding** *carrier-irr-def* **by** *auto*

lemma *card-carrier-irr*[*simp*]: *card carrier-irr* = *CARD*(*'a*)^(*degree f*)
proof −
  let *?A* = (*carrier-vec* (*degree f*):: *'a mod-ring vec set*)
  have *bij-A-carrier*: *bij-betw* (*Poly* ∘ *list-of-vec*) *?A carrier-irr*
  proof (*unfold bij-betw-def*, *rule conjI*)
    show *inj-on* (*Poly* ∘ *list-of-vec*) *?A* **by** (*rule inj-Poly-list-of-vec*)
    show (*Poly* ∘ *list-of-vec*) ' *?A* = *carrier-irr*
    proof (*unfold image-def o-def carrier-irr-def*, *auto*)
      fix *xa* **assume** *xa* ∈ *?A* **thus** *degree* (*Poly* (*list-of-vec xa*)) < *degree f*
        using *degree-Poly-list-of-vec irr-f* **by** *blast*
    next
      fix *x*::*'a mod-ring poly*
      assume *deg-x*: *degree x* < *degree f*
      let *?xa* = *vec-of-list* (*coeffs x* @ *replicate* (*degree f* − *length* (*coeffs x*)) *0*)
      show ∃ *xa*∈*carrier-vec* (*degree f*). *x* = *Poly* (*list-of-vec xa*)
        by (*rule bexI*[*of - ?xa*], *unfold carrier-vec-def*, *insert deg-x*)
          (*auto simp add*: *degree-eq-length-coeffs*)
    qed
  qed
  have *CARD*(*'a*)^(*degree f*) = *card ?A*
    by (*simp add*: *card-carrier-vec*)
  also have ... = *card carrier-irr* **using** *bij-A-carrier bij-betw-same-card* **by** *blast*
  finally show *?thesis* **..**
qed

lemma *finite-carrier-irr*[*simp*]: *finite* (*carrier-irr*)
proof −
  have *degree f* > *degree 0* **using** *degree-0* **by** *auto*
  hence *carrier-irr* ≠ {} **using** *degree-0* **unfolding** *carrier-irr-def*
    by *blast*
  moreover have *card carrier-irr* ≠ *0* **by** *auto*
  ultimately show *?thesis* **using** *card-eq-0-iff* **by** *metis*
qed

lemma *finite-carrier-R*[*simp*]: *finite* (*carrier R*) **unfolding** *R-def* **by** *simp*

lemma *finite-carrier-mult-of*[*simp*]: *finite* (*carrier* (*mult-of R*))
  unfolding *carrier-mult-of* **by** *auto*

lemma *constant-in-carrier*[*simp*]: [:*a*:] ∈ *carrier R*
  unfolding *R-def carrier-irr-def* **by** *auto*

lemma *mod-in-carrier*[*simp*]: *a mod f* ∈ *carrier R*
  unfolding *R-def carrier-irr-def*

by (*auto*, *metis degree-0 degree-f degree-mod-less′ less-not-refl*)

lemma *order-irr*: *Coset.order* (*mult-of R*) = *CARD*(′*a*)^*degree f* − *1*
by (*simp add*: *card-Diff-singleton Coset.order-def carrier-mult-of R-def*)

lemma *element-power-order-eq-1*:
assumes *x*: $x \in$ *carrier* (*mult-of R*)
shows *x* [^]$_{(mult\text{-}of\ R)}$ *Coset.order* (*mult-of R*) = $\mathbf{1}_{(mult\text{-}of\ R)}$
by (*meson field-R.field-mult-group finite-carrier-mult-of group.pow-order-eq-1 x*)

corollary *element-power-order-eq-1′*:
assumes *x*: $x \in$ *carrier* (*mult-of R*)
shows*x* [^]$_{(mult\text{-}of\ R)}$ *CARD*(′*a*)^*degree f* = *x*
proof −
have *x* [^]$_{(mult\text{-}of\ R)}$ *CARD*(′*a*)^*degree f*
= *x* $\otimes_{(mult\text{-}of\ R)}$ *x* [^]$_{(mult\text{-}of\ R)}$ (*CARD*(′*a*)^*degree f* − *1*)
by (*metis Diff-iff One-nat-def Suc-pred field-R.m-comm field-R.nat-pow-Suc field-R.nat-pow-closed*
mult-of-simps(*1*) *mult-of-simps*(*2*) *nat-pow-mult-of neq0-conv power-eq-0-iff x zero-less-card-finite*)
also have *x* $\otimes_{(mult\text{-}of\ R)}$ *x* [^]$_{(mult\text{-}of\ R)}$ (*CARD*(′*a*)^*degree f* − *1*) = *x*
by (*metis carrier-mult-of element-power-order-eq-1 field-R.Units-closed field-R.field-Units*

field-R.r-one monoid.simps(*2*) *mult-mult-of mult-of-def order-irr x*)
finally show *?thesis* **.**
qed

lemma *pow-irr*[*simp*]: *x* [^]$_{(R)}$ *n*= *x*^*n mod f*
by (*induct n*, *auto simp add*: *mod-poly-less nat-pow-def R-def mult-of-def mult-irr-def*

carrier-irr-def mod-mult-right-eq mult.commute)

lemma *pow-irr-mult-of*[*simp*]: *x* [^]$_{(mult\text{-}of\ R)}$ *n*= *x*^*n mod f*
by (*induct n*, *auto simp add*: *mod-poly-less nat-pow-def R-def mult-of-def mult-irr-def*

carrier-irr-def mod-mult-right-eq mult.commute)

lemma *fermat-theorem-power-poly-R*[*simp*]: [:*a*:] [^]$_R$ *CARD*(′*a*) ^ *n* = [:*a*:]
by (*auto simp add*: *Missing-Polynomial.poly-const-pow mod-poly-less*)

lemma *times-mod-expand*:
(*a* $\otimes_{(R)}$ *b*) = ((*a mod f*) $\otimes_{(R)}$ (*b mod f*))
by (*simp add*: *mod-mult-eq R-def mult-irr-def*)

lemma *mult-closed-power*:
assumes *x*: $x \in$ *carrier R* **and** *y*: $y \in$ *carrier R*
and *x* [^]$_{(R)}$ *CARD*(′*a*) ^ *m′* = *x*

and $y\ [^\frown]_{(R)}\ CARD('a)\ ^\frown m' = y$
shows $(x \otimes_{(R)} y)\ [^\frown]_{(R)}\ CARD('a)\ ^\frown m' = (x \otimes_{(R)} y)$
using *assms assms field-R.nat-pow-distrib* **by** *auto*

lemma *add-closed-power*:
assumes *x1*: $x\ [^\frown]_{(R)}\ CARD('a)\ ^\frown m' = x$
and *y1*: $y\ [^\frown]_{(R)}\ CARD('a)\ ^\frown m' = y$
shows $(x \oplus_{(R)} y)\ [^\frown]_{(R)}\ CARD('a)\ ^\frown m' = (x \oplus_{(R)} y)$
proof −
have $(x + y)\ ^\frown CARD('a)\ ^\frown m' = x^\frown(CARD('a)\ ^\frown m') + y\ ^\frown(CARD('a)\ ^\frown m')$
by *auto*
hence $(x + y)\ ^\frown CARD('a)\ ^\frown m'\ mod\ f = (x^\frown(CARD('a)\ ^\frown m') + y\ ^\frown(CARD('a)$
$^\frown m'))\ mod\ f$ **by** *auto*
hence $(x \oplus_{(R)} y)\ [^\frown]_{(R)}\ CARD('a)\ ^\frown m'$
$= (x\ [^\frown]_{(R)}\ CARD('a)^\frown m') \oplus_{(R)} (y\ [^\frown]_{(R)}\ CARD('a)^\frown m')$
by (*auto*, *unfold R-def plus-irr-def*, *auto simp add*: *mod-add-eq power-mod*)
also have ... $= x \oplus_{(R)} y$ **unfolding** *x1 y1* **by** *simp*
finally show *?thesis* **.**
qed

lemma *x-power-pm-minus-1*:
assumes *x*: $x \in carrier\ (mult\text{-}of\ R)$
and $x\ [^\frown]_{(R)}\ CARD('a)\ ^\frown m' = x$
shows $x\ [^\frown]_{(R)}\ (CARD('a)\ ^\frown m' - 1) = \mathbf{1}_{(R)}$
by (*metis* (*no-types*, *lifting*) *One-nat-def Suc-pred assms*(*2*) *carrier-mult-of field-R.Units-closed*

field-R.Units-l-cancel field-R.field-Units field-R.l-one field-R.m-rcancel field-R.nat-pow-Suc

field-R.nat-pow-closed field-R.one-closed field-R.r-null field-R.r-one x zero-less-card-finite

zero-less-power)

context
begin

private lemma *monom-a-1-P*:
assumes *m*: *monom 1 1* $\in$ *carrier R*
and *eq*: *monom 1 1* $[^\frown]_{(R)}\ (CARD('a)\ ^\frown m')$ = *monom 1 1*
shows *monom a 1* $[^\frown]_{(R)}\ (CARD('a)\ ^\frown m')$ = *monom a 1*
proof −
have *monom a 1* = [:*a*:] ∗ (*monom 1 1*)
by (*metis One-nat-def monom-0 monom-Suc mult.commute pCons-0-as-mult*)
also have ... = [:*a*:] $\otimes_{(R)}$ (*monom 1 1*)
by (*auto simp add*: *R-def mult-irr-def*)
(*metis One-nat-def assms*(*2*) *mod-mod-trivial mod-smult-left pow-irr*)
finally have *eq2*: *monom a 1* = [:*a*:] $\otimes_R$ *monom 1 1* **.**
show *?thesis* **unfolding** *eq2*

by (*rule mult-closed-power*[*OF - m - eq*], *insert fermat-theorem-power-poly-R*, *auto*)
qed

private lemma *prod-monom-1-1*:
defines $P == (\lambda\ x\ n.\ (x[\widehat{\ }]_{(R)}\ (CARD('a)\ \widehat{\ }\ n) = x))$
assumes *m*: $monom\ 1\ 1 \in carrier\ R$
and *eq*: $P\ (monom\ 1\ 1)\ n$
shows $P\ ((\prod i = 0..<b::nat.\ monom\ 1\ 1)\ mod\ f)\ n$
proof (*induct b*)
case *0*
then show *?case* **unfolding** *P-def*
by (*simp add*: *power-mod*)
next
case (*Suc b*)
let $?N = (\prod i = 0..<b.\ monom\ 1\ 1)$
have *eq2*: $(\prod i = 0..<Suc\ b.\ monom\ 1\ 1)\ mod\ f = monom\ 1\ 1 \otimes_{(R)} (\prod i = 0..<b.\ monom\ 1\ 1)$
by (*metis field-R.m-comm field-R.nat-pow-Suc mod-in-carrier mod-mod-trivial pow-irr prod-pow times-mod-expand*)
also have ... $= (monom\ 1\ 1\ mod\ f) \otimes_{(R)} ((\prod i = 0..<b.\ monom\ 1\ 1)\ mod\ f)$
by (*rule times-mod-expand*)
finally have *eq2*: $(\prod i = 0..<Suc\ b.\ monom\ 1\ 1)\ mod\ f$
$= (monom\ 1\ 1\ mod\ f) \otimes_{(R)} ((\prod i = 0..<b.\ monom\ 1\ 1)\ mod\ f)$ **.**
show *?case*
unfolding *eq2 P-def*
proof (*rule mult-closed-power*)
show $(monom\ 1\ 1\ mod\ f)\ [\widehat{\ }]_R\ CARD('a)\ \widehat{\ }\ n = monom\ 1\ 1\ mod\ f$
using *P-def element-in-carrier eq m mod-poly-less* **by** *force*
show $((\prod i = 0..<b.\ monom\ 1\ 1)\ mod\ f)\ [\widehat{\ }]_R\ CARD('a)\ \widehat{\ }\ n = (\prod i = 0..<b.\ monom\ 1\ 1)\ mod\ f$
using *P-def Suc.hyps* **by** *blast*
qed (*auto*)
qed

private lemma *monom-1-b*:
defines $P == (\lambda\ x\ n.\ (x[\widehat{\ }]_{(R)}\ (CARD('a)\ \widehat{\ }\ n) = x))$
assumes *m*: $monom\ 1\ 1 \in carrier\ R$
and *monom-1-1*: $P\ (monom\ 1\ 1)\ m'$
and *b*: $b < degree\ f$
shows $P\ (monom\ 1\ b)\ m'$
proof −
have $monom\ 1\ b = (\prod i = 0..<b.\ monom\ 1\ 1)$
by (*metis prod-pow x-pow-n*)
also have ... $= (\prod i = 0..<b.\ monom\ 1\ 1)\ mod\ f$
by (*rule mod-poly-less*[*symmetric*], *auto*)
(*metis One-nat-def b degree-linear-power x-as-monom*)
finally have *eq2*: $monom\ 1\ b = (\prod i = 0..<b.\ monom\ 1\ 1)\ mod\ f$ **.**

show *?thesis* **unfolding** *eq2 P-def*
by (*rule prod-monom-1-1*[*OF m monom-1-1*[*unfolded P-def*]])
qed

private lemma *monom-a-b*:
defines $P == (\lambda\ x\ n.\ (x[\widehat{\ }]_{(R)}\ (CARD('a)\ \widehat{\ }\ n) = x))$
assumes *m*: $monom\ 1\ 1 \in carrier\ R$
and *m1*: $P\ (monom\ 1\ 1)\ m'$
and *b*: $b < degree\ f$
shows $P\ (monom\ a\ b)\ m'$
proof $-$
have $monom\ a\ b = smult\ a\ (monom\ 1\ b)$
by (*simp add*: *smult-monom*)
also have ... $= [:a:] * (monom\ 1\ b)$ **by** *auto*
also have ... $= [:a:] \otimes_{(R)} (monom\ 1\ b)$
unfolding *R-def mult-irr-def*
by (*simp add*: *b degree-monom-eq mod-poly-less*)
finally have *eq*: $monom\ a\ b = [:a:] \otimes_{(R)} (monom\ 1\ b)$ **.**
show *?thesis* **unfolding** *eq P-def*
proof (*rule mult-closed-power*)
show $[:a:]\ [\widehat{\ }]_R\ CARD('a)\ \widehat{\ }\ m' = [:a:]$ **by** (*rule fermat-theorem-power-poly-R*)
show $monom\ 1\ b\ [\widehat{\ }]_R\ CARD('a)\ \widehat{\ }\ m' = monom\ 1\ b$
unfolding *P-def* **by** (*rule monom-1-b*[*OF m m1*[*unfolded P-def*] *b*])
show $monom\ 1\ b \in carrier\ R$ **unfolding** *element-in-carrier* **using** *b*
by (*simp add*: *degree-monom-eq*)
qed (*auto*)
qed

private lemma *sum-monoms-P*:
defines $P == (\lambda\ x\ n.\ (x[\widehat{\ }]_{(R)}\ (CARD('a)\ \widehat{\ }\ n) = x))$
assumes *m*: $monom\ 1\ 1 \in carrier\ R$
and *monom-1-1*: $P\ (monom\ 1\ 1)\ n$
and *b*: $b < degree\ f$
shows $P\ ((\sum i{\le}b.\ monom\ (g\ i)\ i))\ n$
using *b*
proof (*induct b*)
case *0*
then show *?case* **unfolding** *P-def*
by (*simp add*: *poly-const-pow mod-poly-less monom-0*)
next
case (*Suc b*)
have *b*: $b < degree\ f$ **using** *Suc.prems* **by** *auto*
have *rw*: $(\sum i{\le}b.\ monom\ (g\ i)\ i)\ mod\ f = (\sum i{\le}b.\ monom\ (g\ i)\ i)$ **by** (*rule sum-monom-mod*[*OF b*])
have *rw2*: $(monom\ (g\ (Suc\ b))\ (Suc\ b)\ mod\ f) = monom\ (g\ (Suc\ b))\ (Suc\ b)$
by (*metis Suc.prems field-R.nat-pow-eone m monom-a-b pow-irr power-0 power-one-right*)

have *hyp*: P ($\sum i{\le}b.$ *monom* (g i) i) n **using** *Suc.prems Suc.hyps* **by** *auto*
have ($\sum i{\le}Suc\ b.$ *monom* (g i) i) = *monom* (g (*Suc* b)) (*Suc* b) + ($\sum i{\le}b.$ *monom* (g i) i)
by *simp*
also have ... = (*monom* (g (*Suc* b)) (*Suc* b) *mod* f) + (($\sum i{\le}b.$ *monom* (g i) i) *mod* f)
using *rw rw2* **by** *argo*
also have ... = *monom* (g (*Suc* b)) (*Suc* b) $\oplus_R$ ($\sum i{\le}b.$ *monom* (g i) i)
unfolding *R-def plus-irr-def*
by (*simp add*: *poly-mod-add-left*)
finally have *eq*: ($\sum i{\le}Suc\ b.$ *monom* (g i) i)
= *monom* (g (*Suc* b)) (*Suc* b) $\oplus_R$ ($\sum i{\le}b.$ *monom* (g i) i) **.**
show *?case* **unfolding** *eq P-def*
proof (*rule add-closed-power*)
show *monom* (g (*Suc* b)) (*Suc* b) $[\urcorner]_R$ $CARD('a)$ ^ n = *monom* (g (*Suc* b)) (*Suc* b)
by (*rule monom-a-b*[*OF m monom-1-1*[*unfolded P-def*] *Suc.prems*])
show ($\sum i{\le}b.$ *monom* (g i) i) $[\urcorner]_R$ $CARD('a)$ ^ n = ($\sum i{\le}b.$ *monom* (g i) i)
using *hyp* **unfolding** *P-def* **by** *simp*
qed
qed

lemma *element-carrier-P*:
defines $P \equiv (\lambda\ x\ n.\ (x[\urcorner]_{(R)}\ (CARD('a)$ ^ $n) = x))$
assumes *m*: *monom 1 1* $\in$ *carrier R*
and *monom-1-1*: P (*monom 1 1*) m'
and *a*: $a \in$ *carrier R*
shows $P\ a\ m'$
proof –
have *degree-a*: *degree a* < *degree f* **using** *a element-in-carrier* **by** *simp*
have P ($\sum i{\le}degree\ a.$ *monom* (*poly.coeff a i*) i) m'
unfolding *P-def*
by (*rule sum-monoms-P*[*OF m monom-1-1*[*unfolded P-def*] *degree-a*])
thus *?thesis* **unfolding** *poly-as-sum-of-monoms* **by** *simp*
qed
end

end

lemma *degree-divisor1*:
assumes *f*: *irreducible* (f :: $'a$ :: *prime-card mod-ring poly*)
and *d*: *degree f* = d
shows *f dvd* (*monom 1 1*)^($CARD('a)$^d) – *monom 1 1*
proof –
interpret *poly-mod-type-irr* $CARD('a)$ f **by** (*unfold-locales*, *auto simp add*: f)
show *?thesis*
proof (*cases* $d = 1$)
case *True*

show *?thesis*
proof (*cases monom 1 1 mod f* = *0*)
case *True*
then show *?thesis*
by (*metis Suc-pred dvd-diff dvd-mult2 mod-eq-0-iff-dvd power.simps*(*2*)
zero-less-card-finite zero-less-power)
next
case *False* **note** *mod-f-not0* = *False*
have *monom 1* (*CARD*(*'a*)) *mod f* = *monom 1 1 mod f*
proof −
let *?g1* = (*monom 1* (*CARD*(*'a*))) *mod f*
let *?g2* = (*monom 1 1*) *mod f*
have *deg-g1*: *degree ?g1* < *degree f* **and** *deg-g2*: *degree ?g2* < *degree f*
by (*metis True card-UNIV-unit d degree-0 degree-mod-less' zero-less-card-finite zero-neq-one*)+
have *g2*: *?g2* $[^\frown]_{(mult\text{-}of\ R)}$ *CARD*(*'a*)^*degree f* = *?g2* ^ (*CARD*(*'a*)^*degree f*) *mod f*
by (*rule pow-irr-mult-of*)
have *?g2* $[^\frown]_{(mult\text{-}of\ R)}$ *CARD*(*'a*)^*degree f* = *?g2*
by (*rule element-power-order-eq-1'*, *insert mod-f-not0 deg-g2*,
auto simp add: *carrier-mult-of R-def carrier-irr-def*)
hence *?g2* ^ *CARD*(*'a*) *mod f* = *?g2 mod f* **using** *True d* **by** *auto*
hence *?g1 mod f* = *?g2 mod f* **by** (*metis mod-mod-trivial power-mod x-pow-n*)
thus *?thesis* **by** *simp*
qed
thus *?thesis* **by** (*metis True mod-eq-dvd-iff-poly power-one-right x-pow-n*)
qed
next
case *False*
have *deg-f1*: *1* < *degree f*
using *False d degree-f* **by** *linarith*
have *monom 1 1* $[^\frown]_{(mult\text{-}of\ R)}$ *CARD*(*'a*)^*degree f* = *monom 1 1*
by (*rule element-power-order-eq-1'*, *insert deg-f1*)
(*auto simp add*: *carrier-mult-of R-def carrier-irr-def degree-monom-eq*)
hence *monom 1 1*^*CARD*(*'a*)^*degree f mod f* = *monom 1 1 mod f*
using *deg-f1* **by** (*auto*, *metis mod-mod-trivial*)
thus *?thesis* **using** *d mod-eq-dvd-iff-poly* **by** *blast*
qed
qed

lemma *degree-divisor2*:
assumes *f*: *irreducible* (*f* :: *'a* :: *prime-card mod-ring poly*)
and *d*: *degree f* = *d*
and *c-ge-1*: *1* ≤ *c* **and** *cd*: *c* < *d*
shows ¬ *f dvd monom 1 1* ^ *CARD*(*'a*) ^ *c* − *monom 1 1*
proof (*rule ccontr*)
interpret *poly-mod-type-irr CARD*(*'a*) *f* **by** (*unfold-locales*, *auto simp add*: *f*)
have *field-R*: *field R*

by (*simp add*: *field-R.field-axioms*)
assume ¬ ¬ *f dvd monom 1 1* ^ *CARD*(′*a*) ^ *c* − *monom 1 1*
hence *f-dvd*: *f dvd monom 1 1* ^ *CARD*(′*a*) ^ *c* − *monom 1 1* **by** *simp*
obtain *a* **where** *a-R*: *a* ∈ *carrier* (*mult-of R*)
and *ord-a*: *group.ord* (*mult-of R*) *a* = *order* (*mult-of R*)
and *gen*: *carrier* (*mult-of R*) = {*a* [^]$_R$ *i* |*i*. *i* ∈ (*UNIV*::*nat set*)}
using *field.finite-field-mult-group-has-gen2*[*OF field-R*] **by** *auto*
have *d-not1*: *d*>*1* **using** *c-ge-1 cd* **by** *auto*
have *monom-in-carrier*: *monom 1 1* ∈ *carrier* (*mult-of R*)
using *d-not1* **unfolding** *carrier-mult-of R-def carrier-irr-def*
by (*simp add*: *d degree-monom-eq*)
then have *monom 1 1* ∉ {$\mathbf{0}_R$}
by *auto*
then obtain *k* **where** *monom 1 1* = *a* ^ *k mod f*
using *gen monom-in-carrier* **by** *auto*
then have *k*: *a* [^]$_R$ *k* = *monom 1 1*
by *simp*
have *a-m-1*: *a* [^]$_R$ (*CARD*(′*a*)^*c* − *1*) = $\mathbf{1}_R$
proof (*rule x-power-pm-minus-1*[*OF a-R*])
let *?x* = *monom 1 1*::′*a mod-ring poly*
show *a* [^]$_R$ *CARD*(′*a*) ^ *c* = *a*
proof (*rule element-carrier-P*)
show *?x* ∈ *carrier R*
by (*metis k mod-in-carrier pow-irr*)
have *?x* ^ *CARD*(′*a*)^ *c mod f* = *?x mod f* **using** *f-dvd*
using *mod-eq-dvd-iff-poly* **by** *blast*
thus *?x* [^]$_R$ *CARD*(′*a*)^ *c* = *?x*
by (*metis d d-not1 degree-monom-eq mod-poly-less one-neq-zero pow-irr*)
show *a* ∈ *carrier R* **using** *a-R* **unfolding** *carrier-mult-of* **by** *auto*
qed
qed
have *Group.group* (*mult-of R*)
by (*simp add*: *field-R.field-mult-group*)
moreover have *finite* (*carrier* (*mult-of R*)) **by** *auto*
moreover have *a* ∈ *carrier* (*mult-of R*) **by** (*rule a-R*)
moreover have *a* [^]$_{mult\text{-}of\ R}$ (*CARD*(′*a*) ^ *c* − *1*) = $\mathbf{1}_{mult\text{-}of\ R}$
using *a-m-1* **unfolding** *mult-of-def*
by (*auto*, *metis mult-of-def pow-irr-mult-of nat-pow-mult-of*)
ultimately have *ord-dvd*: *group.ord* (*mult-of R*) *a dvd* (*CARD*(′*a*)^*c* − *1*)
by (*meson group.pow-eq-id*)
have *d dvd c*
proof (*rule dvd-power-minus-1-conv1*[*OF nontriv*])
show *0* < *d* **using** *cd* **by** *auto*
show *CARD*(′*a*) ^ *d* − *1 dvd CARD*(′*a*) ^ *c* − *1*
using *ord-dvd* **by** (*simp add*: *d ord-a order-irr*)
show *0* < *c* **using** *c-ge-1* **by** *auto*
qed
thus *False* **using** *c-ge-1 cd*
using *nat-dvd-not-less* **by** *auto*

qed

lemma *degree-divisor*: **assumes** *irreducible* (*f* :: *'a* :: *prime-card mod-ring poly*)
degree f = *d*
  shows *f dvd* (*monom 1 1*)^(*CARD*(*'a*)^*d*) − *monom 1 1*
  and *1* ≤ *c* ⟹ *c* < *d* ⟹ ¬ *f dvd* (*monom 1 1*)^(*CARD*(*'a*)^*c*) − *monom 1 1*
    using *assms degree-divisor1 degree-divisor2* **by** *blast+*

context
  assumes *SORT-CONSTRAINT*(*'a* :: *prime-card*)
begin

function *dist-degree-factorize-main* ::
  'a mod-ring poly ⇒ *'a mod-ring poly* ⇒ *nat* ⇒ (*nat* × *'a mod-ring poly*) *list*
  ⇒ (*nat* × *'a mod-ring poly*) *list* **where**
  dist-degree-factorize-main v w d res = (*if v* = *1 then res else if d* + *d* > *degree v*
    then (*degree v*, *v*) # *res else let*
      w = *w*^(*CARD*(*'a*)) *mod v*;
      d = *Suc d*;
      gd = *gcd* (*w* − *monom 1 1*) *v*
      in if gd = *1 then dist-degree-factorize-main v w d res else*
      let v' = *v div gd in*
      dist-degree-factorize-main v' (*w mod v'*) *d* ((*d*,*gd*) # *res*))
  by *pat-completeness auto*

termination
proof (*relation measure* (λ (*v*,*w*,*d*,*res*). *Suc* (*degree v*) − *d*), *goal-cases*)
  case (*3 v w d res x xa xb xc*)
  have *xb dvd v* **unfolding** *3* **by** *auto*
  hence *xc dvd v* **unfolding** *3* **by** (*metis dvd-def dvd-div-mult-self*)
  from *divides-degree*[*OF this*] *3*
  show *?case* **by** *auto*
qed *auto*

declare *dist-degree-factorize-main.simps*[*simp del*]

lemma *dist-degree-factorize-main*: **assumes**
  dist: *dist-degree-factorize-main v w d res* = *facts* **and**
  w: *w* = (*monom 1 1*)^(*CARD*(*'a*)^*d*) *mod v* **and**
  sf: *square-free u* **and**
  mon: *monic u* **and**
  prod: *u* = *v* ∗ *prod-list* (*map snd res*) **and**
  deg: ⋀ *f*. *irreducible f* ⟹ *f dvd v* ⟹ *degree f* > *d* **and**
  res: ⋀ *i f*. (*i*,*f*) ∈ *set res* ⟹ *i* ≠ *0* ∧ *degree f* ≠ *0* ∧ *monic f* ∧ (∀ *g*. *irreducible*
g ⟶ *g dvd f* ⟶ *degree g* = *i*)
shows *u* = *prod-list* (*map snd facts*) ∧ (∀ *i f*. (*i*,*f*) ∈ *set facts* ⟶ *factors-of-same-degree*
i f)
  using *dist w prod res deg* **unfolding** *factors-of-same-degree-def*

proof (*induct v w d res rule*: *dist-degree-factorize-main.induct*)
  case (*1 v w d res*)
  note *IH* = *1*(*1*−*2*)
  note *result* = *1*(*3*)
  note *w* = *1*(*4*)
  note *u* = *1*(*5*)
  note *res* = *1*(*6*)
  note *fact* = *1*(*7*)
  note [*simp*] = *dist-degree-factorize-main.simps*[*of* - - *d*]
  let *?x* = *monom 1 1* :: ′*a mod-ring poly*
  show *?case*
  proof (*cases v* = *1*)
    case *True*
    thus *?thesis* **using** *result u mon res* **by** *auto*
  next
    case *False* **note** *v* = *this*
    note *IH* = *IH*[*OF this*]
   have *mon-prod*: *monic* (*prod-list* (*map snd res*)) **by** (*rule monic-prod-list*, *insert res*, *auto*)
    with *mon*[*unfolded u*] **have** *mon-v*: *monic v* **by** (*simp add*: *coeff-degree-mult*)
    with *False* **have** *deg-v*: *degree v* ≠ *0* **by** (*simp add*: *monic-degree-0*)
    show *?thesis*
    proof (*cases degree v* < *d* + *d*)
      case *True*
      with *result False* **have** *facts*: *facts* = (*degree v*, *v*) # *res* **by** *simp*
      show *?thesis*
      proof (*intro allI conjI impI*)
        fix *i f g*
        assume ∗: (*i*,*f*) ∈ *set facts irreducible g g dvd f*
        show *degree g* = *i*
        proof (*cases* (*i*,*f*) ∈ *set res*)
          case *True*
          from *res*[*OF this*] ∗ **show** *?thesis* **by** *auto*
        next
          case *False*
          with ∗ *facts* **have** *id*: *i* = *degree v f* = *v* **by** *auto*
          note ∗ = ∗(*2*−*3*)[*unfolded id*]
          from *fact*[*OF* ∗] **have** *dg*: *d* < *degree g* **by** *auto*
          from *divides-degree*[*OF* ∗(*2*)] *mon-v* **have** *deg-gv*: *degree g* ≤ *degree v* **by** *auto*
          from ∗(*2*) **obtain** *h* **where** *vgh*: *v* = *g* ∗ *h* **unfolding** *dvd-def* **by** *auto*
           from *arg-cong*[*OF this*, *of degree*] *mon-v* **have** *dvgh*: *degree v* = *degree g* + *degree h*
            by (*metis deg-v degree-mult-eq degree-mult-eq-0*)
          with *dg deg-gv dg True* **have** *deg-h*: *degree h* < *d* **by** *auto*
          {
            assume *degree h* = *0*
            with *dvgh* **have** *degree g* = *degree v* **by** *simp*
          }

moreover
{
assume *deg-h0*: *degree h* ≠ *0*
hence ∃ *k*. *irreducible*$_d$ *k* ∧ *k dvd h*
using *dvd-triv-left irreducible*$_d$*-factor* **by** *blast*
then obtain *k* **where** *irr*: *irreducible k* **and** *k dvd h* **by** *auto*
from *dvd-trans*[*OF this*(*2*), *of v*] *vgh* **have** *k dvd v* **by** *auto*
from *fact*[*OF irr this*] **have** *dk*: *d* < *degree k* **.**
from *divides-degree*[*OF* ‹*k dvd h*›] *deg-h0* **have** *degree k* ≤ *degree h* **by** *auto*
with *deg-h* **have** *degree k* < *d* **by** *auto*
with *dk* **have** *False* **by** *auto*
}
ultimately have *degree g* = *degree v* **by** *auto*
thus *?thesis* **unfolding** *id* **by** *auto*
qed
qed (*insert v mon-v deg-v u facts res*, *force+*)
next
case *False*
note *IH* = *IH*[*OF this refl refl refl*]
let *?p* = *CARD*(*'a*)
let *?w* = *w* ^ *?p mod v*
let *?g* = *gcd* (*?w* − *?x*) *v*
let *?v* = *v div ?g*
let *?d* = *Suc d*
from *result*[*simplified*] *v False*
have *result*: (*if ?g* = *1 then dist-degree-factorize-main v ?w ?d res*
else dist-degree-factorize-main ?v (*?w mod ?v*) *?d* ((*?d*, *?g*) # *res*)) = *facts*
by (*auto simp*: *Let-def*)
from *mon-v* **have** *mon-g*: *monic ?g* **by** (*metis deg-v degree-0 poly-gcd-monic*)
have *ww*: *?w* = *?x* ^ *?p* ^ *?d mod v* **unfolding** *w*
by *simp* (*metis* (*mono-tags*, *opaque-lifting*) *One-nat-def mult.commute power-Suc power-mod power-mult x-pow-n*)
have *gv*: *?g dvd v* **by** *auto*
hence *gv'*: *v div ?g dvd v*
by (*metis dvd-def dvd-div-mult-self*)
{
fix *f*
assume *irr*: *irreducible f* **and** *fv*: *f dvd v* **and** *degree f* = *?d*
from *degree-divisor*(*1*)[*OF this*(*1*,*3*)]
have *f dvd ?x* ^ *?p* ^ *?d* − *?x* **by** *auto*
hence *f dvd* (*?x* ^ *?p* ^ *?d* − *?x*) *mod v* **using** *fv* **by** (*rule dvd-mod*)
also have (*?x* ^ *?p* ^ *?d* − *?x*) *mod v* = *?x* ^ *?p* ^ *?d mod v* − *?x mod v* **by** (*rule poly-mod-diff-left*)
also have *?x* ^ *?p* ^ *?d mod v* = *?w mod v* **unfolding** *ww* **by** *auto*
also have . . . − *?x mod v* = (*w* ^ *?p mod v* − *?x*) *mod v* **by** (*metis poly-mod-diff-left*)
finally have *f dvd* (*w*^*?p mod v* − *?x*) **using** *fv* **by** (*rule dvd-mod-imp-dvd*)

```
        with fv have f dvd ?g by auto
      } note deg-d-dvd-g = this
      show ?thesis
      proof (cases ?g = 1)
        case True
          with result have dist: dist-degree-factorize-main v ?w ?d res = facts by
auto
        show ?thesis
        proof (rule IH(1)[OF True dist ww u res])
          fix f
          assume irr: irreducible f and fv: f dvd v
          from fact[OF this] have d < degree f .
          moreover have degree f ≠ ?d
          proof
            assume degree f = ?d
            from divides-degree[OF deg-d-dvd-g[OF irr fv this]] mon-v
            have degree f ≤ degree ?g by auto
            with irr have degree ?g ≠ 0 unfolding irreducible_d-def by auto
            with True show False by auto
          qed
          ultimately show ?d < degree f by auto
        qed
      next
        case False
        with result
        have result: dist-degree-factorize-main ?v (?w mod ?v) ?d ((?d, ?g) # res)
= facts
          by auto
       from False mon-g have deg-g: degree ?g ≠ 0 by (simp add: monic-degree-0)
        have www: ?w mod ?v = monom 1 1 ^ ?p ^ ?d mod ?v using gv'
          by (simp add: mod-mod-cancel ww)
        from square-free-factor[OF - sf, of v] u have sfv: square-free v by auto
        have u: u = ?v * prod-list (map snd ((?d, ?g) # res))
          unfolding u by simp
        show ?thesis
        proof (rule IH(2)[OF False refl result www u], goal-cases)
          case (1 i f)
          show ?case
          proof (cases (i,f) ∈ set res)
            case True
            from res[OF this] show ?thesis by auto
          next
            case False
            with 1 have id: i = ?d f = ?g by auto
            show ?thesis unfolding id
            proof (intro conjI impI allI)
              fix g
              assume *: irreducible g g dvd ?g
              hence gv: g dvd v using dvd-trans[of g ?g v] by simp
```

from *fact*[*OF* ∗(*1*) *this*] **have** *dg*: *d* < *degree g* **.**
{
assume *degree g* > *?d*
from *degree-divisor*(*2*)[*OF* ∗(*1*) *refl - this*]
have *ndvd*: ¬ *g dvd ?x* ^ *?p* ^ *?d* − *?x* **by** *auto*
from ∗(*2*) **have** *g dvd ?w* − *?x* **by** *simp*
from *this*[*unfolded ww*]
have *g dvd ?x* ^ *?p* ^ *?d mod v* − *?x* **.**
with *gv* **have** *g dvd* (*?x* ^ *?p* ^ *?d mod v* − *?x*) *mod v* **by** (*metis dvd-mod*)
also have (*?x* ^ *?p* ^ *?d mod v* − *?x*) *mod v* = (*?x* ^ *?p* ^ *?d* − *?x*) *mod v*
by (*metis mod-diff-left-eq*)
finally have *g dvd ?x* ^ *?p* ^ *?d* − *?x* **using** *gv* **by** (*rule dvd-mod-imp-dvd*)
with *ndvd* **have** *False* **by** *auto*
}
with *dg* **show** *degree g* = *?d* **by** *presburger*
qed (*insert mon-g deg-g*, *auto*)
qed
next
case (*2 f*)
note *irr* = *2*(*1*)
from *dvd-trans*[*OF 2*(*2*) *gv'*] **have** *fv*: *f dvd v* **.**
from *fact*[*OF irr fv*] **have** *df*: *d* < *degree f degree f* ≠ *0* **by** *auto*
{
assume *degree f* = *?d*
from *deg-d-dvd-g*[*OF irr fv this*] **have** *fg*: *f dvd ?g* **.**
from *gv* **have** *id*: *v* = (*v div ?g*) ∗ *?g* **by** *simp*
from *sfv id* **have** *square-free* (*v div ?g* ∗ *?g*) **by** *simp*
from *square-free-multD*(*1*)[*OF this 2*(*2*) *fg*] **have** *degree f* = *0* **.**
with *df* **have** *False* **by** *auto*
}
with *df* **show** *?d* < *degree f* **by** *presburger*
qed
qed
qed
qed
qed

definition *distinct-degree-factorization*
:: *'a mod-ring poly* ⇒ (*nat* × *'a mod-ring poly*) *list* **where**
distinct-degree-factorization f =
(*if degree f* = *1 then* [(*1*,*f*)] *else dist-degree-factorize-main f* (*monom 1 1*) *0* [])

lemma *distinct-degree-factorization*: **assumes**
dist: *distinct-degree-factorization f* = *facts* **and**
u: *square-free f* **and**

mon: *monic f*
shows *f* = *prod-list* (*map snd facts*) ∧ (∀ *i f*. (*i*,*f*) ∈ *set facts* ⟶ *factors-of-same-degree i f*)
proof −
note *dist* = *dist*[*unfolded distinct-degree-factorization-def*]
show *?thesis*
proof (*cases degree f* ≤ *1*)
case *False*
hence *degree f* > *1* **and** *dist*: *dist-degree-factorize-main f* (*monom 1 1*) *0* [] = *facts*
using *dist* **by** *auto*
hence ∗: *monom 1* (*Suc 0*) = *monom 1* (*Suc 0*) *mod f*
by (*simp add*: *degree-monom-eq mod-poly-less*)
show *?thesis*
by (*rule dist-degree-factorize-main*[*OF dist - u mon*], *insert* ∗, *auto simp*: *irreducible*$_d$*-def*)
next
case *True*
hence *degree f* = *0* ∨ *degree f* = *1* **by** *auto*
thus *?thesis*
proof
assume *degree f* = *0*
with *mon* **have** *f*: *f* = *1* **using** *monic-degree-0* **by** *blast*
hence *facts* = [] **using** *dist* **unfolding** *dist-degree-factorize-main.simps*[*of - - 0*]
by *auto*
thus *?thesis* **using** *f* **by** *auto*
next
assume *deg*: *degree f* = *1*
hence *facts*: *facts* = [(*1*,*f*)] **using** *dist* **by** *auto*
show *?thesis* **unfolding** *facts factors-of-same-degree-def*
proof (*intro conjI allI impI*; *clarsimp*)
fix *g*
assume *irreducible g g dvd f*
thus *degree g* = *Suc 0* **using** *deg divides-degree*[*of g f*] **by** (*auto simp*: *irreducible*$_d$*-def*)
qed (*insert mon deg*, *auto*)
qed
qed
qed
end

end

8 A Combined Factorization Algorithm for Polynomials over GF(p)

8.1 Type Based Version

We combine Berlekamp's algorithm with the distinct degree factorization to obtain an efficient factorization algorithm for square-free polynomials in GF(p).

theory *Finite-Field-Factorization*
imports *Berlekamp-Type-Based*
Distinct-Degree-Factorization
begin

We prove soundness of the finite field factorization, indepedendent on whether distinct-degree-factorization is applied as preprocessing or not.

consts *use-distinct-degree-factorization* :: *bool*

context
assumes *SORT-CONSTRAINT*('*a*::*prime-card*)
begin

definition *finite-field-factorization* :: '*a mod-ring poly* ⇒ '*a mod-ring* × '*a mod-ring poly list* **where**
finite-field-factorization f = (*if degree f* = *0 then* (*lead-coeff f*,[]) *else let*
a = *lead-coeff f*;
u = *smult* (*inverse a*) *f*;
gs = (*if use-distinct-degree-factorization then distinct-degree-factorization u else* [(*1*,*u*)]);
(*irr*,*hs*) = *List.partition* (λ (*i*,*f*). *degree f* = *i*) *gs*
in (*a*,*map snd irr* @ *concat* (*map* (λ (*i*,*g*). *berlekamp-monic-factorization i g*) *hs*)))

lemma *finite-field-factorization-explicit*:
fixes *f*::'*a mod-ring poly*
assumes *sf-f*: *square-free f*
and *us*: *finite-field-factorization f* = (*c*,*us*)
shows *f* = *smult c* (*prod-list us*) ∧ (∀ *u* ∈ *set us*. *monic u* ∧ *irreducible u*)
proof (*cases degree f* = *0*)
case *False* **note** *f* = *this*
define *g* **where** *g* = *smult* (*inverse c*) *f*
obtain *gs* **where** *dist*: (*if use-distinct-degree-factorization then distinct-degree-factorization g else* [(*1*,*g*)]) = *gs* **by** *auto*
note *us* = *us*[*unfolded finite-field-factorization-def Let-def*]
from *us f* **have** *c*: *c* = *lead-coeff f* **by** *auto*
obtain *irr hs* **where** *part*: *List.partition* (λ (*i*, *f*). *degree f* = *i*) *gs* = (*irr*,*hs*) **by** *force*
from *arg-cong*[*OF this*, *of fst*] **have** *irr*: *irr* = *filter* (λ (*i*, *f*). *degree f* = *i*) *gs* **by** *auto*

from *us*[*folded c*, *folded g-def*, *unfolded dist part split*] *f*
have *us*: *us* = *map snd irr* @ *concat* (*map* (λ(*x*, *y*). *berlekamp-monic-factorization x y*) *hs*) **by** *auto*
from *f c* **have** *c0*: *c* ≠ *0* **by** *auto*
from *False c0* **have** *deg-g*: *degree g* ≠ *0* **unfolding** *g-def* **by** *auto*
have *mon-g*: *monic g* **unfolding** *g-def*
by (*metis c c0 field-class.field-inverse lead-coeff-smult*)
from *sf-f* **have** *sf-g*: *square-free g* **unfolding** *g-def* **by** (*simp add*: *c0*)
from *c0* **have** *f*: *f* = *smult c g* **unfolding** *g-def* **by** *auto*
have *g* = *prod-list* (*map snd gs*) ∧ (∀ (*i*,*f*) ∈ *set gs*. *degree f* > *0* ∧ *monic f* ∧ (∀ *h*. *h dvd f* ⟶ *degree h* = *i* ⟶ *irreducible h*))
proof (*cases use-distinct-degree-factorization*)
case *True*
with *dist* **have** *distinct-degree-factorization g* = *gs* **by** *auto*
note *dist* = *distinct-degree-factorization*[*OF this sf-g mon-g*]
from *dist* **have** *g*: *g* = *prod-list* (*map snd gs*) **by** *auto*
show *?thesis*
proof (*intro conjI*[*OF g*] *ballI*, *clarify*)
fix *i f*
assume (*i*,*f*) ∈ *set gs*
with *dist* **have** *factors-of-same-degree i f* **by** *auto*
from *factors-of-same-degreeD*[*OF this*]
show *degree f* > *0* ∧ *monic f* ∧ (∀ *h*. *h dvd f* ⟶ *degree h* = *i* ⟶ *irreducible h*) **by** *auto*
qed
next
case *False*
with *dist* **have** *gs*: *gs* = [(*1*,*g*)] **by** *auto*
show *?thesis* **unfolding** *gs* **using** *deg-g mon-g linear-irreducible*$_d$[**where** ′*a* = ′*a mod-ring*] **by** *auto*
qed
hence *g-gs*: *g* = *prod-list* (*map snd gs*)
and *mon-gs*: ⋀ *i f*. (*i*, *f*) ∈ *set gs* ⟹ *monic f* ∧ *degree f* > *0*
and *irrI*: ⋀ *i f h* . (*i*, *f*) ∈ *set gs* ⟹ *h dvd f* ⟹ *degree h* = *i* ⟹ *irreducible h* **by** *auto*
have *g*: *g* = *prod-list* (*map snd irr*) ∗ *prod-list* (*map snd hs*) **unfolding** *g-gs*
using *prod-list-map-partition*[*OF part*] **.**
{
fix *f*
assume *f* ∈ *snd* ‘ *set irr*
from *this*[*unfolded irr*] **obtain** *i* **where** ∗: (*i*,*f*) ∈ *set gs degree f* = *i* **by** *auto*
have *f dvd f* **by** *auto*
from *irrI*[*OF* ∗(*1*) *this* ∗(*2*)] *mon-gs*[*OF* ∗(*1*)] **have** *monic f irreducible f* **by** *auto*
} **note** *irr* = *this*
let *?berl* = λ *hs*. *concat* (*map* (λ(*x*, *y*). *berlekamp-monic-factorization x y*) *hs*)
have *set hs* ⊆ *set gs* **using** *part* **by** *auto*
hence *prod-list* (*map snd hs*) = *prod-list* (*?berl hs*)
∧ (∀ *f* ∈ *set* (*?berl hs*). *monic f* ∧ *irreducible*$_d$ *f*)

proof (*induct hs*)
case (*Cons ih hs*)
obtain *i h* **where** *ih*: *ih* = (*i*,*h*) **by** *force*
have *?berl* (*Cons ih hs*) = *berlekamp-monic-factorization i h* @ *?berl hs* **unfolding** *ih* **by** *auto*
from *Cons*(*2*)[*unfolded ih*] **have** *mem*: (*i*,*h*) ∈ *set gs* **and** *sub*: *set hs* ⊆ *set gs* **by** *auto*
note *IH* = *Cons*(*1*)[*OF sub*]
from *mem* **have** *h* ∈ *set* (*map snd gs*) **by** *force*
from *square-free-factor*[*OF prod-list-dvd*[*OF this*], *folded g-gs*, *OF sf-g*] **have** *sf*: *square-free h* **.**
from *mon-gs*[*OF mem*] *irrI*[*OF mem*] **have** ∗: *degree h* > *0 monic h*
⋀ *g*. *g dvd h* ⟹ *degree g* = *i* ⟹ *irreducible g* **by** *auto*
from *berlekamp-monic-factorization*[*OF sf refl* ∗(*3*) ∗(*1*−*2*), *of i*]
have *berl*: *prod-list* (*berlekamp-monic-factorization i h*) = *h*
and *irr*: ⋀ *f*. *f* ∈ *set* (*berlekamp-monic-factorization i h*) ⟹ *monic f* ∧ *irreducible f* **by** *auto*
have *prod-list* (*map snd* (*Cons ih hs*)) = *h* ∗ *prod-list* (*map snd hs*) **unfolding** *ih* **by** *simp*
also have *prod-list* (*map snd hs*) = *prod-list* (*?berl hs*) **using** *IH* **by** *auto*
finally have *prod-list* (*map snd* (*Cons ih hs*)) = *prod-list* (*?berl* (*Cons ih hs*))
unfolding *ih* **using** *berl* **by** *auto*
thus *?case* **using** *IH irr* **unfolding** *ih* **by** *auto*
qed *auto*
with *g irr* **have** *main*: *g* = *prod-list us* ∧ (∀ *u* ∈ *set us*. *monic u* ∧ $irreducible_d$ *u*) **unfolding** *us*
by *auto*
thus *?thesis* **unfolding** *f* **using** *sf-g* **by** *auto*
next
case *True*
with *us*[*unfolded finite-field-factorization-def*] **have** *c* = *lead-coeff f* **and** *us*: *us* = [] **by** *auto*
with *degree0-coeffs*[*OF True*] **have** *f*: *f* = [:*c*:] **by** *auto*
show *?thesis* **unfolding** *us f* **by** (*auto simp*: *normalize-poly-def*)
qed

lemma *finite-field-factorization*:
fixes *f*::′*a mod-ring poly*
assumes *sf-f*: *square-free f*
and *us*: *finite-field-factorization f* = (*c*,*us*)
shows *unique-factorization Irr-Mon f* (*c*, *mset us*)
proof −
from *finite-field-factorization-explicit*[*OF sf-f us*]
have *fact*: *factorization Irr-Mon f* (*c*, *mset us*)
unfolding *factorization-def split Irr-Mon-def* **by** (*auto simp*: *prod-mset-prod-list*)
from *sf-f*[*unfolded square-free-def*] **have** *f* ≠ *0* **by** *auto*
from *exactly-one-factorization*[*OF this*] *fact*
show *?thesis* **unfolding** *unique-factorization-def* **by** *auto*
qed

end

Experiments revealed that preprocessing via distinct-degree-factorization slows down the factorization algorithm (statement for implementation in AFP 2017)

overloading *use-distinct-degree-factorization* ≡ *use-distinct-degree-factorization*
begin
  definition *use-distinct-degree-factorization*
    where [*code-unfold*]: *use-distinct-degree-factorization* = *False*
end
end

8.2 Record Based Version

theory *Finite-Field-Factorization-Record-Based*
imports
  Finite-Field-Factorization
  Matrix-Record-Based
  Poly-Mod-Finite-Field-Record-Based
  HOL−Types-To-Sets.Types-To-Sets
  Jordan-Normal-Form.Matrix-IArray-Impl
  Jordan-Normal-Form.Gauss-Jordan-IArray-Impl
  Polynomial-Interpolation.Improved-Code-Equations
  Polynomial-Factorization.Missing-List
begin

hide-const(**open**) *monom coeff*

Whereas ⟦*square-free ?f*; *finite-field-factorization ?f* = (*?c*, *?us*)⟧ ⟹ *unique-factorization Irr-Mon ?f* (*?c*, *mset ?us*) provides a result for a polynomials over GF(p), we now develop a theorem which speaks about integer polynomials modulo p.

lemma (**in** *poly-mod-prime-type*) *finite-field-factorization-modulo-ring*:
  assumes *g*: (*g* :: *'a mod-ring poly*) = *of-int-poly f*
  and *sf*: *square-free-m f*
  and *fact*: *finite-field-factorization g* = (*d*,*gs*)
  and *c*: *c* = *to-int-mod-ring d*
  and *fs*: *fs* = *map to-int-poly gs*
  shows *unique-factorization-m f* (*c*, *mset fs*)
proof −
 have [*transfer-rule*]: *MP-Rel f g* **unfolding** *g MP-Rel-def* **by** (*simp add*: *Mp-f-representative*)
 have *sg*: *square-free g* **by** (*transfer*, *rule sf*)
 have [*transfer-rule*]: *M-Rel c d* **unfolding** *M-Rel-def c* **by** (*rule M-to-int-mod-ring*)
 have *fs-gs*[*transfer-rule*]: *list-all2 MP-Rel fs gs*
   unfolding *fs list-all2-map1 MP-Rel-def*[*abs-def*] *Mp-to-int-poly* **by** (*simp add*:
list.rel-refl)
 have [*transfer-rule*]: *rel-mset MP-Rel* (*mset fs*) (*mset gs*)
   using *fs-gs* **using** *rel-mset-def* **by** *blast*

have [*transfer-rule*]: *MF-Rel* (*c*,*mset fs*) (*d*,*mset gs*) **unfolding** *MF-Rel-def* **by** *transfer-prover*
from *finite-field-factorization*[*OF sg fact*]
have *uf*: *unique-factorization Irr-Mon g* (*d*,*mset gs*) **by** *auto*
from *uf*[*untransferred*] **show** *unique-factorization-m f* (*c*, *mset fs*) **.**
qed

We now have to implement *finite-field-factorization*.

context
fixes *p* :: *int*
and *ff-ops* :: *'i arith-ops-record*
begin

fun *power-poly-f-mod-i* :: (*'i list* ⇒ *'i list*) ⇒ *'i list* ⇒ *nat* ⇒ *'i list* **where**
power-poly-f-mod-i modulus a n = (*if n* = *0 then modulus* (*one-poly-i ff-ops*)
else let (*d*,*r*) = *Divides.divmod-nat n 2*;
rec = *power-poly-f-mod-i modulus* (*modulus* (*times-poly-i ff-ops a a*)) *d in*
if r = *0 then rec else modulus* (*times-poly-i ff-ops rec a*))

declare *power-poly-f-mod-i.simps*[*simp del*]

fun *power-polys-i* :: *'i list* ⇒ *'i list* ⇒ *'i list* ⇒ *nat* ⇒ *'i list list* **where**
power-polys-i mul-p u curr-p (*Suc i*) = *curr-p* #
power-polys-i mul-p u (*mod-field-poly-i ff-ops* (*times-poly-i ff-ops curr-p mul-p*) *u*) *i*
| *power-polys-i mul-p u curr-p 0* = []

lemma *length-power-polys-i*[*simp*]: *length* (*power-polys-i x y z n*) = *n*
by (*induct n arbitrary*: *x y z*, *auto*)

definition *berlekamp-mat-i* :: *'i list* ⇒ *'i mat* **where**
berlekamp-mat-i u = (*let n* = *degree-i u*;
ze = *arith-ops-record.zero ff-ops*; *on* = *arith-ops-record.one ff-ops*;
mul-p = *power-poly-f-mod-i* (λ *v*. *mod-field-poly-i ff-ops v u*)
[*ze*, *on*] (*nat p*);
xks = *power-polys-i mul-p u* [*on*] *n*
in mat-of-rows-list n (*map* (λ *cs*. *cs* @ *replicate* (*n* − *length cs*) *ze*) *xks*))

definition *berlekamp-resulting-mat-i* :: *'i list* ⇒ *'i mat* **where**
berlekamp-resulting-mat-i u = (*let Q* = *berlekamp-mat-i u*;
n = *dim-row Q*;
QI = *mat n n* (λ (*i*,*j*). *if i* = *j then arith-ops-record.minus ff-ops* (*Q* $$ (*i*,*j*)) (*arith-ops-record.one ff-ops*) *else Q* $$ (*i*,*j*))
in (*gauss-jordan-single-i ff-ops* (*transpose-mat QI*)))

definition *berlekamp-basis-i* :: *'i list* ⇒ *'i list list* **where**
berlekamp-basis-i u = (*map* (*poly-of-list-i ff-ops o list-of-vec*)
(*find-base-vectors-i ff-ops* (*berlekamp-resulting-mat-i u*)))

primrec *berlekamp-factorization-main-i* :: *'i* ⇒ *'i* ⇒ *nat* ⇒ *'i list list* ⇒ *'i list list* ⇒ *nat* ⇒ *'i list list* **where**
berlekamp-factorization-main-i ze on d divs (*v* # *vs*) *n* = (
if v = [*on*] *then berlekamp-factorization-main-i ze on d divs vs n else*
if length divs = *n then divs else*
let of-int = *arith-ops-record.of-int ff-ops*;
facts = *filter* (λ *w*. *w* ≠ [*on*])
[*gcd-poly-i ff-ops u* (*minus-poly-i ff-ops v* (*if s* = *0 then* [] *else* [*of-int* (*int s*)])) .
u ← *divs*, *s* ← [*0* ..< *nat p*]];
(*lin*,*nonlin*) = *List.partition* (λ *q*. *degree-i q* = *d*) *facts*
in lin @ *berlekamp-factorization-main-i ze on d nonlin vs* (*n* − *length lin*))
| *berlekamp-factorization-main-i ze on d divs* [] *n* = *divs*

definition *berlekamp-monic-factorization-i* :: *nat* ⇒ *'i list* ⇒ *'i list list* **where**
berlekamp-monic-factorization-i d f = (*let*
vs = *berlekamp-basis-i f*
in berlekamp-factorization-main-i (*arith-ops-record.zero ff-ops*) (*arith-ops-record.one ff-ops*) *d* [*f*] *vs* (*length vs*))

partial-function (*tailrec*) *dist-degree-factorize-main-i* ::
'i ⇒ *'i* ⇒ *nat* ⇒ *'i list* ⇒ *'i list* ⇒ *nat* ⇒ (*nat* × *'i list*) *list*
⇒ (*nat* × *'i list*) *list* **where**
[*code*]: *dist-degree-factorize-main-i ze on dv v w d res* = (*if v* = [*on*] *then res else if d* + *d* > *dv*
then (*dv*, *v*) # *res else let*
w = *power-poly-f-mod-i* (λ *f*. *mod-field-poly-i ff-ops f v*) *w* (*nat p*);
d = *Suc d*;
gd = *gcd-poly-i ff-ops* (*minus-poly-i ff-ops w* [*ze*,*on*]) *v*
in if gd = [*on*] *then dist-degree-factorize-main-i ze on dv v w d res else*
let v' = *div-field-poly-i ff-ops v gd*
in dist-degree-factorize-main-i ze on (*degree-i v'*) *v'* (*mod-field-poly-i ff-ops w v'*) *d* ((*d*,*gd*) # *res*))

definition *distinct-degree-factorization-i*
:: *'i list* ⇒ (*nat* × *'i list*) *list* **where**
distinct-degree-factorization-i f = (*let ze* = *arith-ops-record.zero ff-ops*;
on = *arith-ops-record.one ff-ops in if degree-i f* = *1 then* [(*1*,*f*)] *else*
dist-degree-factorize-main-i ze on (*degree-i f*) *f* [*ze*,*on*] *0* [])

definition *finite-field-factorization-i* :: *'i list* ⇒ *'i* × *'i list list* **where**
finite-field-factorization-i f = (*if degree-i f* = *0 then* (*lead-coeff-i ff-ops f*,[]) *else let*
a = *lead-coeff-i ff-ops f*;
u = *smult-i ff-ops* (*arith-ops-record.inverse ff-ops a*) *f*;
gs = (*if use-distinct-degree-factorization then distinct-degree-factorization-i u else* [(*1*,*u*)]);
(*irr*,*hs*) = *List.partition* (λ (*i*,*f*). *degree-i f* = *i*) *gs*
in (*a*,*map snd irr* @ *concat* (*map* (λ (*i*,*g*). *berlekamp-monic-factorization-i i g*)

hs)))
end

context *prime-field-gen*
begin

lemma *power-polys-i*: **assumes** *i*: $i < n$ **and** [*transfer-rule*]: *poly-rel f f′ poly-rel g g′*
  and *h*: *poly-rel h h′*
  shows *poly-rel* (*power-polys-i ff-ops g f h n* ! *i*) (*power-polys g′ f′ h′ n* ! *i*)
  using *i h*
proof (*induct n arbitrary*: *h h′ i*)
  case (*Suc n h h′ i*) **note** $*$ = *this*
  note [*transfer-rule*] = $*(3)$
  show *?case*
  proof (*cases i*)
    case *0*
    with *Suc* **show** *?thesis* **by** *auto*
  next
    case (*Suc j*)
    with $*(2-)$ **have** $j < n$ **by** *auto*
    note *IH* = $*(1)$[*OF this*]
    show *?thesis* **unfolding** *Suc* **by** (*simp*, *rule IH*, *transfer-prover*)
  qed
qed *simp*

lemma *power-poly-f-mod-i*: **assumes** *m*: (*poly-rel ===> poly-rel*) *m* (λ *x′*. *x′ mod m′*)
  shows *poly-rel f f′* $\Longrightarrow$ *poly-rel* (*power-poly-f-mod-i ff-ops m f n*) (*power-poly-f-mod m′ f′ n*)
proof −
  from *m* **have** *m*: $\bigwedge$ *x x′*. *poly-rel x x′* $\Longrightarrow$ *poly-rel* (*m x*) (*x′ mod m′*)
    unfolding *rel-fun-def* **by** *auto*
  show *poly-rel f f′* $\Longrightarrow$ *poly-rel* (*power-poly-f-mod-i ff-ops m f n*) (*power-poly-f-mod m′ f′ n*)
  proof (*induct n arbitrary*: *f f′ rule*: *less-induct*)
    case (*less n f f′*)
    note *f*[*transfer-rule*] = *less*(*2*)
    show *?case*
    proof (*cases* $n = 0$)
      case *True*
      show *?thesis*
        by (*simp add*: *True power-poly-f-mod-i.simps power-poly-f-mod-binary*,
          rule m[*OF poly-rel-one*])
    next
      case *False*
      hence *n*: ($n = 0$) = *False* **by** *simp*
      obtain *q r* **where** *div*: *Divides.divmod-nat n 2* = (*q*,*r*) **by** *force*
      from *this*[*unfolded divmod-nat-def*] *n* **have** $q < n$ **by** *auto*

```
      note IH = less(1)[OF this]
      have rec: poly-rel (power-poly-f-mod-i ff-ops m (m (times-poly-i ff-ops f f)) q)

        (power-poly-f-mod m′ (f′ ∗ f′ mod m′) q)
        by (rule IH, rule m, transfer-prover)
      have other: poly-rel
        (m (times-poly-i ff-ops (power-poly-f-mod-i ff-ops m (m (times-poly-i ff-ops
f f)) q) f))
        (power-poly-f-mod m′ (f′ ∗ f′ mod m′) q ∗ f′ mod m′)
        by (rule m, rule poly-rel-times[unfolded rel-fun-def, rule-format, OF rec f])
      show ?thesis unfolding power-poly-f-mod-i.simps[of - - - n] Let-def
        power-poly-f-mod-binary[of - - n] div split n if-False using rec other by auto
    qed
  qed
qed

lemma berlekamp-mat-i[transfer-rule]: (poly-rel ===> mat-rel R)
  (berlekamp-mat-i p ff-ops) berlekamp-mat
proof (intro rel-funI)
  fix f f′
  let ?ze = arith-ops-record.zero ff-ops
  let ?on = arith-ops-record.one ff-ops
  assume f[transfer-rule]: poly-rel f f′
  have deg: degree-i f = degree f′ by transfer-prover
  {
    fix i j
    assume i: i < degree f′ and j: j < degree f′
    define cs where cs = (λcs :: ′i list. cs @ replicate (degree f′ − length cs) ?ze)
    define cs′ where cs′ = (λcs :: ′a mod-ring poly. coeffs cs @ replicate (degree
f′ − length (coeffs cs)) 0)
    define poly where poly = power-polys-i ff-ops
        (power-poly-f-mod-i ff-ops (λv. mod-field-poly-i ff-ops v f) [?ze, ?on] (nat
p)) f [?on]
        (degree f′)
    define poly′ where poly′ = (power-polys (power-poly-f-mod f′ [:0, 1:] (nat p))
f′ 1 (degree f′))
    have ∗: poly-rel (power-poly-f-mod-i ff-ops (λv. mod-field-poly-i ff-ops v f) [?ze,
?on] (nat p))
      (power-poly-f-mod f′ [:0, 1:] (nat p))
      by (rule power-poly-f-mod-i, transfer-prover, simp add: poly-rel-def one zero)
    have [transfer-rule]: poly-rel (poly ! i) (poly′ ! i)
      unfolding poly-def poly′-def
      by (rule power-polys-i[OF i f ∗], simp add: poly-rel-def one)
    have ∗: list-all2 R (cs (poly ! i)) (cs′ (poly′ ! i))
      unfolding cs-def cs′-def by transfer-prover
    from list-all2-nthD[OF ∗[unfolded poly-rel-def], of j] j
    have R (cs (poly ! i) ! j) (cs′ (poly′ ! i) ! j) unfolding cs-def by auto
    hence R
        (mat-of-rows-list (degree f′)
```

(*map* (*λcs*. *cs* @ *replicate* (*degree* *f'* − *length* *cs*) *?ze*)
(*power-polys-i* *ff-ops*
(*power-poly-f-mod-i* *ff-ops* (*λv*. *mod-field-poly-i* *ff-ops* *v* *f*) [*?ze*, *?on*]
(*nat* *p*)) *f* [*?on*]
(*degree* *f'*))) $$
(*i*, *j*))
(*mat-of-rows-list* (*degree* *f'*)
(*map* (*λcs*. *coeffs* *cs* @ *replicate* (*degree* *f'* − *length* (*coeffs* *cs*)) *0*)
(*power-polys* (*power-poly-f-mod* *f'* [:*0*, *1*:] (*nat* *p*)) *f'* *1* (*degree* *f'*))) $$
(*i*, *j*))
unfolding *mat-of-rows-list-def length-map length-power-polys-i power-polys-works length-power-polys index-mat*[*OF i j*] *split*
unfolding *poly-def cs-def poly'-def cs'-def* **using** *i*
by *auto*
} **note** *main* = *this*
show *mat-rel R* (*berlekamp-mat-i p ff-ops f*) (*berlekamp-mat f'*)
unfolding *berlekamp-mat-i-def berlekamp-mat-def Let-def nat-p*[*symmetric*] *deg*
unfolding *mat-rel-def*
by (*intro conjI allI impI*, *insert main*, *auto*)
qed

lemma *berlekamp-resulting-mat-i*[*transfer-rule*]: (*poly-rel* ===> *mat-rel R*)
(*berlekamp-resulting-mat-i p ff-ops*) *berlekamp-resulting-mat*
proof (*intro rel-funI*)
fix *f f'*
assume *poly-rel f f'*
from *berlekamp-mat-i*[*unfolded rel-fun-def*, *rule-format*, *OF this*]
have *bmi*: *mat-rel R* (*berlekamp-mat-i p ff-ops f*) (*berlekamp-mat f'*) **.**
show *mat-rel R* (*berlekamp-resulting-mat-i p ff-ops f*) (*berlekamp-resulting-mat*
f')
unfolding *berlekamp-resulting-mat-def Let-def berlekamp-resulting-mat-i-def*
by (*rule gauss-jordan-i*[*unfolded rel-fun-def*, *rule-format*],
insert bmi, *auto simp*: *mat-rel-def one intro*!: *minus*[*unfolded rel-fun-def*, *rule-format*])
qed

lemma *berlekamp-basis-i*[*transfer-rule*]: (*poly-rel* ===> *list-all2 poly-rel*)
(*berlekamp-basis-i p ff-ops*) *berlekamp-basis*
unfolding *berlekamp-basis-i-def*[*abs-def*] *berlekamp-basis-code*[*abs-def*] *o-def*
by *transfer-prover*

lemma *berlekamp-factorization-main-i*[*transfer-rule*]:
((=) ===> *list-all2 poly-rel* ===> *list-all2 poly-rel* ===> (=) ===> *list-all2*
poly-rel)
(*berlekamp-factorization-main-i p ff-ops* (*arith-ops-record.zero ff-ops*)
(*arith-ops-record.one ff-ops*))
berlekamp-factorization-main
proof (*intro rel-funI*, *clarify*, *goal-cases*)
case (*1* - *d xs xs'* *ys ys'* - *n*)
let *?ze* = *arith-ops-record.zero ff-ops*

let *?on* = *arith-ops-record.one ff-ops*
let *?of-int* = *arith-ops-record.of-int ff-ops*
from *1*(*2*) *1*(*1*) **show** *?case*
proof (*induct ys ys′ arbitrary*: *xs xs′ n rule*: *list-all2-induct*)
case (*Cons y ys y′ ys′ xs xs′ n*)
note *trans*[*transfer-rule*] = *Cons*(*1*,*2*,*4*)
obtain *clar0 clar1 clar2* **where** *clarify*: $\bigwedge$ *s u*. *gcd-poly-i ff-ops u*
(*minus-poly-i ff-ops y*
(*if s* = *0 then* [] *else* [*?of-int* (*int s*)])) = *clar0 s u*
[*0*..<*nat p*] = *clar1*
[*?on*] = *clar2* **by** *auto*
define *facts* **where** *facts* = *concat* (*map* (λ*u*. *concat*
(*map* (λ*s*. *if gcd-poly-i ff-ops u*
(*minus-poly-i ff-ops y* (*if s* = *0 then* [] *else* [*?of-int*
(*int s*)])) ≠
[*?on*]
then [*gcd-poly-i ff-ops u*
(*minus-poly-i ff-ops y* (*if s* = *0 then* [] *else* [*?of-int*
(*int s*)]))]
else [])
[*0*..<*nat p*])) *xs*)
define *Facts* **where** *Facts* = [*w*←*concat*
(*map* (λ*u*. *map* (λ*s*. *gcd-poly-i ff-ops u*
(*minus-poly-i ff-ops y*
(*if s* = *0 then* [] *else* [*?of-int* (*int s*)])))
[*0*..<*nat p*])
xs) . *w* ≠ [*?on*]]
have *Facts*: *Facts* = *facts*
unfolding *Facts-def facts-def clarify*
proof (*induct xs*)
case (*Cons x xs*)
show *?case* **by** (*simp add*: *Cons*, *induct clar1*, *auto*)
qed *simp*
define *facts′* **where** *facts′* = *concat*
(*map* (λ*u*. *concat*
(*map* (λ*x*. *if gcd u* (*y′* − [:*of-nat x*:]) ≠ *1*
then [*gcd u* (*y′* − [:*of-int* (*int x*):])] *else* [])
[*0*..<*nat p*]))
xs′)
have *id*: $\bigwedge$ *x*. *of-int* (*int x*) = *of-nat x* [*?on*] = *one-poly-i ff-ops*
by (*auto simp*: *one-poly-i-def*)
have *facts*[*transfer-rule*]: *list-all2 poly-rel facts facts′*
unfolding *facts-def facts′-def*
apply (*rule concat-transfer*[*unfolded rel-fun-def*, *rule-format*])
apply (*rule list.map-transfer*[*unfolded rel-fun-def*, *rule-format*, *OF* - *trans*(*3*)])
apply (*rule concat-transfer*[*unfolded rel-fun-def*, *rule-format*])
apply (*rule list-all2-map-map*)
proof (*unfold id*)
fix *f f′ x*

assume [*transfer-rule*]: *poly-rel f f′* **and** *x*: *x* ∈ *set* [*0*..<*nat p*]
hence ∗: *0* ≤ *int x int x* < *p* **by** *auto*
from *of-int*[*OF this*] **have** *rel*[*transfer-rule*]: *R* (*?of-int* (*int x*)) (*of-nat x*) **by** *auto*
{
assume *0* < *x*
with ∗ **have** ∗: *0* < *int x int x* < *p* **by** *auto*
have (*of-nat x* :: ′*a mod-ring*) = *of-int* (*int x*) **by** *simp*
also have ... ≠ *0* **unfolding** *of-int-of-int-mod-ring* **using** ∗ **unfolding** *p*
by (*transfer*′, *auto*)
}
with *rel* **have** [*transfer-rule*]: *poly-rel* (*if x* = *0 then* [] *else* [*?of-int* (*int x*)]) [:*of-nat x*:]
unfolding *poly-rel-def* **by** (*auto simp add*: *cCons-def p*)
show *list-all2 poly-rel*
(*if gcd-poly-i ff-ops f* (*minus-poly-i ff-ops y* (*if x* = *0 then* [] *else* [*?of-int* (*int x*)])) ≠ *one-poly-i ff-ops*
then [*gcd-poly-i ff-ops f* (*minus-poly-i ff-ops y* (*if x* = *0 then* [] *else* [*?of-int* (*int x*)]))]
else [])
(*if gcd f*′ (*y*′ − [:*of-nat x*:]) ≠ *1 then* [*gcd f*′ (*y*′ − [:*of-nat x*:])] *else* [])
by *transfer-prover*
qed
have *id1*: *berlekamp-factorization-main-i p ff-ops ?ze ?on d xs* (*y* # *ys*) *n* = (
if y = [*?on*] *then berlekamp-factorization-main-i p ff-ops ?ze ?on d xs ys n else*
if length xs = *n then xs else*
(*let fac* = *facts*;
(*lin*, *nonlin*) = *List.partition* (λ*q*. *degree-i q* = *d*) *fac*
in lin @ *berlekamp-factorization-main-i p ff-ops ?ze ?on d nonlin ys* (*n* − *length lin*)))
unfolding *berlekamp-factorization-main-i.simps Facts*[*symmetric*]
by (*simp add*: *o-def Facts-def Let-def*)
have *id2*: *berlekamp-factorization-main d xs*′ (*y*′ # *ys*′) *n* = (
if y′ = *1 then berlekamp-factorization-main d xs*′ *ys*′ *n*
else if length xs′ = *n then xs*′ *else*
(*let fac* = *facts*′;
(*lin*, *nonlin*) = *List.partition* (λ*q*. *degree q* = *d*) *fac*
in lin @ *berlekamp-factorization-main d nonlin ys*′ (*n* − *length lin*)))
by (*simp add*: *o-def facts*′*-def nat-p*)
have *len*: *length xs* = *length xs*′ **by** *transfer-prover*
have *id3*: (*y* = [*?on*]) = (*y*′ = *1*)
by (*transfer-prover-start*, *transfer-step*+, *simp add*: *one-poly-i-def finite-field-ops-int-def*)
show *?case*
proof (*cases y*′ = *1*)
case *True*
hence *id4*: (*y*′ = *1*) = *True* **by** *simp*
show *?thesis* **unfolding** *id1 id2 id3 id4 if-True*
by (*rule Cons*(*3*), *transfer-prover*)
next

case *False*
hence *id4*: $(y' = 1) = False$ **by** *simp*
note *id1* = *id1*[*unfolded id3 id4 if-False*]
note *id2* = *id2*[*unfolded id4 if-False*]
show *?thesis*
proof (*cases length* $xs' = n$)
case *True*
thus *?thesis* **unfolding** *id1 id2 Let-def len* **using** *trans* **by** *simp*
next
case *False*
hence *id*: (*length* $xs' = n$) = *False* **by** *simp*
have *id'*: *length* [q←*facts* . *degree-i* $q = d$] = *length* [q←*facts'*. *degree* q = d]
by *transfer-prover*
have [*transfer-rule*]: *list-all2 poly-rel* (*berlekamp-factorization-main-i p ff-ops ?ze ?on d* [x←*facts* . *degree-i* $x \neq d$] *ys*
(n − *length* [q←*facts* . *degree-i* $q = d$]))
(*berlekamp-factorization-main d* [x←*facts'* . *degree* $x \neq d$] *ys'*
(n − *length* [q←*facts'* . *degree* $q = d$]))
unfolding *id'*
by (*rule Cons*(*3*), *transfer-prover*)
show *?thesis* **unfolding** *id1 id2 Let-def len id if-False*
unfolding *partition-filter-conv o-def split* **by** *transfer-prover*
qed
qed
qed *simp*
qed

lemma *berlekamp-monic-factorization-i*[*transfer-rule*]:
((=) ===> *poly-rel* ===> *list-all2 poly-rel*)
(*berlekamp-monic-factorization-i p ff-ops*) *berlekamp-monic-factorization*
unfolding *berlekamp-monic-factorization-i-def*[*abs-def*] *berlekamp-monic-factorization-def*[*abs-def*] *Let-def*
by *transfer-prover*

lemma *dist-degree-factorize-main-i*:
poly-rel F f ⟹ *poly-rel G g* ⟹ *list-all2* (*rel-prod* (=) *poly-rel*) *Res res*
⟹ *list-all2* (*rel-prod* (=) *poly-rel*)
(*dist-degree-factorize-main-i p ff-ops*
(*arith-ops-record.zero ff-ops*) (*arith-ops-record.one ff-ops*) (*degree-i F*) *F G d Res*)
(*dist-degree-factorize-main f g d res*)
proof (*induct f g d res arbitrary*: *F G Res rule*: *dist-degree-factorize-main.induct*)
case (*1 v w d res V W Res*)
let *?ze* = *arith-ops-record.zero ff-ops*
let *?on* = *arith-ops-record.one ff-ops*
note *simp* = *dist-degree-factorize-main.simps*[*of v w d*]
dist-degree-factorize-main-i.simps[*of p ff-ops ?ze ?on degree-i V V W d*]
have *v*[*transfer-rule*]: *poly-rel V v* **by** (*rule 1*)

have *w*[*transfer-rule*]: *poly-rel W w* **by** (*rule 1*)
have *res*[*transfer-rule*]: *list-all2* (*rel-prod* (=) *poly-rel*) *Res res* **by** (*rule 1*)
have [*transfer-rule*]: *poly-rel* [*?on*] *1*
by (*simp add*: *one poly-rel-def*)
have *id1*: (V = [*?on*]) = (v = *1*) **unfolding** *finite-field-ops-int-def* **by** *transfer-prover*
have *id2*: *degree-i V* = *degree v* **by** *transfer-prover*
note *simp* = *simp*[*unfolded id1 id2*]
note *IH* = *1*(*1*,*2*)
show *?case*
proof (*cases v* = *1*)
case *True*
with *res* **show** *?thesis* **unfolding** *id2 simp* **by** *simp*
next
case *False*
with *id1* **have** (v = *1*) = *False* **by** *auto*
note *simp* = *simp*[*unfolded this if-False*]
note *IH* = *IH*[*OF False*]
show *?thesis*
proof (*cases degree v* < *d* + *d*)
case *True*
thus *?thesis* **unfolding** *id2 simp* **using** *res v* **by** *auto*
next
case *False*
hence (*degree v* < *d* + *d*) = *False* **by** *auto*
note *simp* = *simp*[*unfolded this if-False*]
let *?P* = *power-poly-f-mod-i ff-ops* (λf. *mod-field-poly-i ff-ops f V*) *W* (*nat p*)
let *?G* = *gcd-poly-i ff-ops* (*minus-poly-i ff-ops ?P* [*?ze*, *?on*]) *V*
let *?g* = *gcd* (*w* ^ *CARD*(*'a*) *mod v* − *monom 1 1*) *v*
define *G* **where** *G* = *?G*
define *g* **where** *g* = *?g*
note *simp* = *simp*[*unfolded Let-def*, *folded G-def g-def*]
note *IH* = *IH*[*OF False refl refl refl*]
have [*transfer-rule*]: *poly-rel* [*?ze*,*?on*] (*monom 1 1*) **unfolding** *poly-rel-def*
by (*auto simp*: *coeffs-monom one zero*)
have *id*: *w* ^ *CARD*(*'a*) *mod v* = *power-poly-f-mod v w* (*nat p*)
unfolding *power-poly-f-mod-def* **by** (*simp add*: *p*)
have *P*[*transfer-rule*]: *poly-rel ?P* (*w* ^ *CARD*(*'a*) *mod v*) **unfolding** *id*
by (*rule power-poly-f-mod-i*[*OF - w*], *transfer-prover*)
have *g*[*transfer-rule*]: *poly-rel G g* **unfolding** *G-def g-def* **by** *transfer-prover*
have *id3*: (*G* = [*?on*]) = (*g* = *1*) **by** *transfer-prover*
note *simp* = *simp*[*unfolded id3*]
show *?thesis*
proof (*cases g* = *1*)
case *True*
from *IH*(*1*)[*OF this*[*unfolded g-def*] *v P res*] *True*
show *?thesis* **unfolding** *id2 simp* **by** *simp*
next

case *False*
have *vg*: *poly-rel* (*div-field-poly-i ff-ops V G*) (*v div g*) **by** *transfer-prover*
have *poly-rel* (*mod-field-poly-i ff-ops ?P*
(*div-field-poly-i ff-ops V G*)) (*w* ^ *CARD*(*'a*) *mod v mod* (*v div g*)) **by** *transfer-prover*
note *IH* = *IH*(*2*)[*OF False*[*unfolded g-def*] *refl vg*[*unfolded G-def g-def*] *this*[*unfolded G-def g-def*],
folded g-def G-def]
have *list-all2* (*rel-prod* (=) *poly-rel*) ((*Suc d*, *G*) # *Res*) ((*Suc d*, *g*) # *res*)
using *g res* **by** *auto*
note *IH* = *IH*[*OF this*]
from *False* **have** (*g* = *1*) = *False* **by** *simp*
note *simp* = *simp*[*unfolded this if-False*]
show *?thesis* **unfolding** *id2 simp* **using** *IH* **by** *simp*
qed
qed
qed
qed

lemma *distinct-degree-factorization-i*[*transfer-rule*]: (*poly-rel* ===> *list-all2* (*rel-prod* (=) *poly-rel*))
(*distinct-degree-factorization-i p ff-ops*) *distinct-degree-factorization*
proof
fix *F f*
assume *f*[*transfer-rule*]: *poly-rel F f*
have *id*: (*degree-i F* = *1*) = (*degree f* = *1*) **by** *transfer-prover*
note *d* = *distinct-degree-factorization-i-def distinct-degree-factorization-def*
let *?ze* = *arith-ops-record.zero ff-ops*
let *?on* = *arith-ops-record.one ff-ops*
show *list-all2* (*rel-prod* (=) *poly-rel*) (*distinct-degree-factorization-i p ff-ops F*)
(*distinct-degree-factorization f*)
proof (*cases degree f* = *1*)
case *True*
with *id f* **show** *?thesis* **unfolding** *d* **by** *auto*
next
case *False*
from *False id* **have** *?thesis* = (*list-all2* (*rel-prod* (=) *poly-rel*)
(*dist-degree-factorize-main-i p ff-ops ?ze ?on* (*degree-i F*) *F* [*?ze*, *?on*] *0* [])
(*dist-degree-factorize-main f* (*monom 1 1*) *0* [])) **unfolding** *d Let-def* **by** *simp*

also have ...
by (*rule dist-degree-factorize-main-i*[*OF f*], *auto simp*: *poly-rel-def*
coeffs-monom one zero)
finally show *?thesis* **.**
qed
qed

lemma *finite-field-factorization-i*[*transfer-rule*]:
(*poly-rel* ===> *rel-prod R* (*list-all2 poly-rel*))

(*finite-field-factorization-i p ff-ops*) *finite-field-factorization*
unfolding *finite-field-factorization-i-def finite-field-factorization-def Let-def lead-coeff-i-def'*
by *transfer-prover*

Since the implementation is sound, we can now combine it with the soundness result of the finite field factorization.

lemma *finite-field-i-sound*:
assumes *f'*: *f' = of-int-poly-i ff-ops* (*Mp f*)
and *berl-i*: *finite-field-factorization-i p ff-ops f'* = (*c'*,*fs'*)
and *sq*: *square-free-m f*
and *fs*: *fs = map* (*to-int-poly-i ff-ops*) *fs'*
and *c*: *c = arith-ops-record.to-int ff-ops c'*
shows *unique-factorization-m f* (*c*, *mset fs*)
∧ *c* ∈ {*0* ..< *p*}
∧ (∀ *fi* ∈ *set fs*. *set* (*coeffs fi*) ⊆ {*0* ..< *p*})
proof −
define *f''* :: *'a mod-ring poly* **where** *f'' = of-int-poly* (*Mp f*)
have *rel-f*[*transfer-rule*]: *poly-rel f' f''*
by (*rule poly-rel-of-int-poly*[*OF f'*], *simp add*: *f''-def*)
interpret *pff*: *idom-ops poly-ops ff-ops poly-rel*
by (*rule idom-ops-poly*)
obtain *c'' fs''* **where** *berl*: *finite-field-factorization f''* = (*c''*,*fs''*) **by** *force*
from *rel-funD*[*OF finite-field-factorization-i rel-f*, *unfolded rel-prod-conv assms*(*2*) *split berl*]
have *rel*[*transfer-rule*]: *R c' c'' list-all2 poly-rel fs' fs''* **by** *auto*
from *to-int*[*OF rel*(*1*)] **have** *cc'*: *c = to-int-mod-ring c''* **unfolding** *c* **by** *simp*
have *c*: *c* ∈ {*0* ..< *p*} **unfolding** *cc'*
by (*metis Divides.pos-mod-bound Divides.pos-mod-sign M-to-int-mod-ring atLeast-LessThan-iff*
gr-implies-not-zero nat-le-0 nat-p not-le poly-mod.M-def zero-less-card-finite)
{
fix *f*
assume *f* ∈ *set fs'*
with *rel*(*2*) **obtain** *f'* **where** *poly-rel f f'* **unfolding** *list-all2-conv-all-nth set-conv-nth*
by *auto*
hence *is-poly ff-ops f* **using** *fun-cong*[*OF Domainp-is-poly*, *of f*]
unfolding *Domainp-iff*[*abs-def*] **by** *auto*
}
hence *fs'*: *Ball* (*set fs'*) (*is-poly ff-ops*) **by** *auto*
define *mon* :: *'a mod-ring poly* ⇒ *bool* **where** *mon = monic*
have [*transfer-rule*]: (*poly-rel* ===> (=)) (*monic-i ff-ops*) *mon* **unfolding** *mon-def*
by (*rule poly-rel-monic*)
have *len*: *length fs' = length fs''* **by** *transfer-prover*
have *fs'*: *fs = map to-int-poly fs''* **unfolding** *fs*
proof (*rule nth-map-conv*[*OF len*], *intro allI impI*)
fix *i*
assume *i*: *i* < *length fs'*

obtain *f g* **where** *id*: *fs′* ! *i* = *f fs′′* ! *i* = *g* **by** *auto*
from *i rel*(*2*)[*unfolded list-all2-conv-all-nth*[*of - fs′ fs′′*]] *id*
have *poly-rel f g* **by** *auto*
from *to-int-poly-i*[*OF this*] **have** *to-int-poly-i ff-ops f* = *to-int-poly g* **.**
thus *to-int-poly-i ff-ops* (*fs′* ! *i*) = *to-int-poly* (*fs′′* ! *i*) **unfolding** *id* **.**
qed
have *f*: *f′′* = *of-int-poly f* **unfolding** *poly-eq-iff f′′-def*
by (*simp add*: *to-int-mod-ring-hom.injectivity to-int-mod-ring-of-int-M Mp-coeff*)
have ∗: *unique-factorization-m f* (*c*, *mset fs*)
using *finite-field-factorization-modulo-ring*[*OF f sq berl cc′ fs′*] **by** *auto*
have *fs′*: (∀ *fi*∈*set fs. set* (*coeffs fi*) ⊆ {*0*..<*p*}) **unfolding** *fs′*
using *range-to-int-mod-ring*[**where** *′a* = *′a*]
by (*auto simp*: *coeffs-to-int-poly p*)
with *c fs* ∗
show *?thesis* **by** *blast*
qed
end

definition *finite-field-factorization-main* :: *int* ⇒ *′i arith-ops-record* ⇒ *int poly* ⇒ *int* × *int poly list* **where**
finite-field-factorization-main p f-ops f ≡
let (*c′*,*fs′*) = *finite-field-factorization-i p f-ops* (*of-int-poly-i f-ops* (*poly-mod.Mp p f*))
in (*arith-ops-record.to-int f-ops c′*, *map* (*to-int-poly-i f-ops*) *fs′*)

lemma(**in** *prime-field-gen*) *finite-field-factorization-main*:
assumes *res*: *finite-field-factorization-main p ff-ops f* = (*c*,*fs*)
and *sq*: *square-free-m f*
shows *unique-factorization-m f* (*c*, *mset fs*)
∧ *c* ∈ {*0* ..< *p*}
∧ (∀ *fi* ∈ *set fs. set* (*coeffs fi*) ⊆ {*0* ..< *p*})
proof −
obtain *c′ fs′* **where**
res′: *finite-field-factorization-i p ff-ops* (*of-int-poly-i ff-ops* (*Mp f*)) = (*c′*, *fs′*)
by *force*
show *?thesis*
by (*rule finite-field-i-sound*[*OF refl res′ sq*],
insert res[*unfolded finite-field-factorization-main-def res′*], *auto*)
qed

definition *finite-field-factorization-int* :: *int* ⇒ *int poly* ⇒ *int* × *int poly list* **where**
finite-field-factorization-int p = (
if p ≤ *65535*
then finite-field-factorization-main p (*finite-field-ops32* (*uint32-of-int p*))
else if p ≤ *4294967295*
then finite-field-factorization-main p (*finite-field-ops64* (*uint64-of-int p*))
else finite-field-factorization-main p (*finite-field-ops-integer* (*integer-of-int p*)))

context *poly-mod-prime* **begin**
lemmas *finite-field-factorization-main-integer* = *prime-field-gen.finite-field-factorization-main*
[*OF prime-field.prime-field-finite-field-ops-integer*, *unfolded prime-field-def mod-ring-locale-def*,
unfolded poly-mod-type-simps, *internalize-sort* $'a$ :: *prime-card*, *OF type-to-set*,
unfolded remove-duplicate-premise, *cancel-type-definition*, *OF non-empty*]

lemmas *finite-field-factorization-main-uint32* = *prime-field-gen.finite-field-factorization-main*
[*OF prime-field.prime-field-finite-field-ops32*, *unfolded prime-field-def mod-ring-locale-def*,
unfolded poly-mod-type-simps, *internalize-sort* $'a$ :: *prime-card*, *OF type-to-set*,
unfolded remove-duplicate-premise, *cancel-type-definition*, *OF non-empty*]

lemmas *finite-field-factorization-main-uint64* = *prime-field-gen.finite-field-factorization-main*
[*OF prime-field.prime-field-finite-field-ops64*, *unfolded prime-field-def mod-ring-locale-def*,
unfolded poly-mod-type-simps, *internalize-sort* $'a$ :: *prime-card*, *OF type-to-set*,
unfolded remove-duplicate-premise, *cancel-type-definition*, *OF non-empty*]

lemma *finite-field-factorization-int*:
assumes *sq*: *poly-mod.square-free-m p f*
and *result*: *finite-field-factorization-int p f* = (*c*,*fs*)
shows *poly-mod.unique-factorization-m p f* (*c*, *mset fs*)
$\wedge$ $c \in \{0 ..< p\}$
$\wedge$ ($\forall$ *fi* $\in$ *set fs*. *set* (*coeffs fi*) $\subseteq \{0 ..< p\}$)
using *finite-field-factorization-main-integer*[*OF - sq*, *of c fs*]
finite-field-factorization-main-uint32[*OF - - sq*, *of c fs*]
finite-field-factorization-main-uint64[*OF - - sq*, *of c fs*]
result[*unfolded finite-field-factorization-int-def*]
by (*auto split*: *if-splits*)

end

end

9 Hensel Lifting

9.1 Properties about Factors

We define and prove properties of Hensel-lifting. Here, we show the result that Hensel-lifting can lift a factorization mod p to a factorization mod p^n. For the lifting we have proofs for both versions, the original linear Hensel-lifting or the quadratic approach from Zassenhaus. Via the linear version, we also show a uniqueness result, however only in the binary case, i.e., where $f = g \cdot h$. Uniqueness of the general case will later be shown in theory Berlekamp-Hensel by incorporating the factorization algorithm for finite fields algorithm.

theory *Hensel-Lifting*
imports
HOL−Computational-Algebra.Euclidean-Algorithm
Poly-Mod-Finite-Field-Record-Based

Polynomial-Factorization.Square-Free-Factorization
begin

lemma *uniqueness-poly-equality*:
fixes *f g* :: *'a* :: {*factorial-ring-gcd*,*semiring-gcd-mult-normalize*} *poly*
assumes *cop*: *coprime f g*
and *deg*: $B = 0 \vee degree\ B < degree\ f\ B' = 0 \vee degree\ B' < degree\ f$
and *f*: $f \neq 0$ **and** *eq*: $A * f + B * g = A' * f + B' * g$
shows $A = A'\ B = B'$
proof −
from *eq* **have** $*$: $(A - A') * f = (B' - B) * g$ **by** (*simp add*: *field-simps*)
hence *f dvd* $(B' - B) * g$ **unfolding** *dvd-def* **by** (*intro exI*[*of - A* − *A'*], *auto simp*: *field-simps*)
with *cop*[*simplified*] **have** *dvd*: *f dvd* $(B' - B)$
by (*simp add*: *coprime-dvd-mult-right-iff ac-simps*)
from *divides-degree*[*OF this*] **have** *degree f* $\leq$ *degree* $(B' - B) \vee B = B'$ **by** *auto*
with *degree-diff-le-max*[*of B' B*] *deg*
show $B = B'$ **by** *auto*
with $*$ *f* **show** $A = A'$ **by** *auto*
qed

lemmas (**in** *poly-mod-prime-type*) *uniqueness-poly-equality* =
uniqueness-poly-equality[**where** *'a='a mod-ring*, *untransferred*]
lemmas (**in** *poly-mod-prime*) *uniqueness-poly-equality* = *poly-mod-prime-type.uniqueness-poly-equality*
[*unfolded poly-mod-type-simps*, *internalize-sort 'a* :: *prime-card*, *OF type-to-set*, *unfolded remove-duplicate-premise*, *cancel-type-definition*, *OF non-empty*]

lemma *pseudo-divmod-main-list-1-is-divmod-poly-one-main-list*:
pseudo-divmod-main-list (*1* :: *'a* :: *comm-ring-1*) *q f g n* = *divmod-poly-one-main-list q f g n*
by (*induct n arbitrary*: *q f g*, *auto simp*: *Let-def*)

lemma *pdivmod-monic-pseudo-divmod*: **assumes** *g*: *monic g* **shows** *pdivmod-monic f g* = *pseudo-divmod f g*
proof −
from *g* **have** *id*: (*coeffs g* = []) = *False* **by** *auto*
from *g* **have** *mon*: *hd* (*rev* (*coeffs g*)) = *1* **by** (*metis coeffs-eq-Nil hd-rev id last-coeffs-eq-coeff-degree*)
show *?thesis*
unfolding *pseudo-divmod-impl pseudo-divmod-list-def id if-False pdivmod-monic-def Let-def mon*
pseudo-divmod-main-list-1-is-divmod-poly-one-main-list **by** (*auto split*: *prod.splits*)
qed

lemma *pdivmod-monic*: **assumes** *g*: *monic g* **and** *res*: *pdivmod-monic f g* = (*q*, *r*)
shows $f = g * q + r\ r = 0 \vee degree\ r < degree\ g$
proof −
from *g* **have** *g0*: $g \neq 0$ **by** *auto*

from *pseudo-divmod*[*OF g0 res*[*unfolded pdivmod-monic-pseudo-divmod*[*OF g*]], *unfolded g*]
show $f = g * q + r$ $r = 0 \vee \mathit{degree}\ r < \mathit{degree}\ g$ **by** *auto*
qed

definition *dupe-monic* :: $'a$ :: *comm-ring-1 poly* $\Rightarrow$ $'a$ *poly* $\Rightarrow$ $'a$ *poly* $\Rightarrow$ $'a$ *poly* $\Rightarrow$ $'a$ *poly* $\Rightarrow$
$'a$ *poly* $*$ $'a$ *poly* **where**
dupe-monic D H S T U = (*case pdivmod-monic* ($T * U$) *D of* (*q*,*r*) $\Rightarrow$
($S * U + H * q$, *r*))

lemma *dupe-monic*: **assumes** *1*: $D{*}S + H{*}T = 1$
and *mon*: *monic D*
and *dupe*: *dupe-monic D H S T U* = (*A*,*B*)
shows $A * D + B * H = U$ $B = 0 \vee \mathit{degree}\ B < \mathit{degree}\ D$
proof −
obtain *Q R* **where** *div*: *pdivmod-monic* (($T * U$)) *D* = (*Q*,*R*) **by** *force*
from *dupe*[*unfolded dupe-monic-def div split*]
have *A*: $A = (S * U + H * Q)$ **and** *B*: $B = R$ **by** *auto*
from *pdivmod-monic*[*OF mon div*] **have** *TU*: $T * U = D * Q + R$ **and**
deg: $R = 0 \vee \mathit{degree}\ R < \mathit{degree}\ D$ **by** *auto*
hence *R*: $R = T * U - D * Q$ **by** *simp*
have $A * D + B * H = (D * S + H * T) * U$ **unfolding** *A B R* **by** (*simp add*: *field-simps*)
also have $\ldots = U$ **unfolding** *1* **by** *simp*
finally show *eq*: $A * D + B * H = U$ **.**
show $B = 0 \vee \mathit{degree}\ B < \mathit{degree}\ D$ **using** *deg* **unfolding** *B* **.**
qed

lemma *dupe-monic-unique*: **fixes** *D* :: $'a$ :: {*factorial-ring-gcd*,*semiring-gcd-mult-normalize*} *poly*
assumes *1*: $D{*}S + H{*}T = 1$
and *mon*: *monic D*
and *dupe*: *dupe-monic D H S T U* = (*A*,*B*)
and *cop*: *coprime D H*
and *other*: $A' * D + B' * H = U$ $B' = 0 \vee \mathit{degree}\ B' < \mathit{degree}\ D$
shows $A' = A$ $B' = B$
proof −
from *dupe-monic*[*OF 1 mon dupe*] **have** *one*: $A * D + B * H = U$ $B = 0 \vee \mathit{degree}\ B < \mathit{degree}\ D$ **by** *auto*
from *mon* **have** *D0*: $D \neq 0$ **by** *auto*
from *uniqueness-poly-equality*[*OF cop one*(*2*) *other*(*2*) *D0*, *of A A'*, *unfolded other*, *OF one*(*1*)]
show $A' = A$ $B' = B$ **by** *auto*
qed

context *ring-ops*
begin
lemma *poly-rel-dupe-monic-i*: **assumes** *mon*: *monic D*

and *rel*: *poly-rel d D poly-rel h H poly-rel s S poly-rel t T poly-rel u U*
shows *rel-prod poly-rel poly-rel* (*dupe-monic-i ops d h s t u*) (*dupe-monic D H S T U*)
proof −
note *defs* = *dupe-monic-i-def dupe-monic-def*
note [*transfer-rule*] = *rel*
have [*transfer-rule*]: *rel-prod poly-rel poly-rel*
(*pdivmod-monic-i ops* (*times-poly-i ops t u*) *d*)
(*pdivmod-monic* (*T* ∗ *U*) *D*)
by (*rule poly-rel-pdivmod-monic*[*OF mon*], *transfer-prover*+)
show *?thesis* **unfolding** *defs* **by** *transfer-prover*
qed
end

context *mod-ring-gen*
begin

lemma *monic-of-int-poly*: *monic D* ⟹ *monic* (*of-int-poly* (*Mp D*) :: *'a mod-ring poly*)
using *Mp-f-representative Mp-to-int-poly monic-Mp* **by** *auto*

lemma *dupe-monic-i*: **assumes** *dupe-i*: *dupe-monic-i ff-ops d h s t u* = (*a*,*b*)
and *1*: *D*∗*S* + *H*∗*T* =*m 1*
and *mon*: *monic D*
and *A*: *A* = *to-int-poly-i ff-ops a*
and *B*: *B* = *to-int-poly-i ff-ops b*
and *d*: *Mp-rel-i d D*
and *h*: *Mp-rel-i h H*
and *s*: *Mp-rel-i s S*
and *t*: *Mp-rel-i t T*
and *u*: *Mp-rel-i u U*
shows
A ∗ *D* + *B* ∗ *H* =*m U*
B = *0* ∨ *degree B* < *degree D*
Mp-rel-i a A
Mp-rel-i b B
proof −
let *?I* = λ *f*. *of-int-poly* (*Mp f*) :: *'a mod-ring poly*
let *?i* = *to-int-poly-i ff-ops*
note *dd* = *Mp-rel-iD*[*OF d*]
note *hh* = *Mp-rel-iD*[*OF h*]
note *ss* = *Mp-rel-iD*[*OF s*]
note *tt* = *Mp-rel-iD*[*OF t*]
note *uu* = *Mp-rel-iD*[*OF u*]
obtain *A′ B′* **where** *dupe*: *dupe-monic* (*?I D*) (*?I H*) (*?I S*) (*?I T*) (*?I U*) = (*A′*,*B′*) **by** *force*
from *poly-rel-dupe-monic-i*[*OF monic-of-int-poly*[*OF mon*] *dd*(*1*) *hh*(*1*) *ss*(*1*) *tt*(*1*) *uu*(*1*), *unfolded dupe-i dupe*]
have *a*: *poly-rel a A′* **and** *b*: *poly-rel b B′* **by** *auto*

show *aa*: *Mp-rel-i a A* **by** (*rule Mp-rel-iI'*[*OF a*, *folded A*])
show *bb*: *Mp-rel-i b B* **by** (*rule Mp-rel-iI'*[*OF b*, *folded B*])
note *Aa* = *Mp-rel-iD*[*OF aa*]
note *Bb* = *Mp-rel-iD*[*OF bb*]
from *poly-rel-inj*[*OF a Aa*(*1*)] *A* **have** *A*: *A'* = *?I A* **by** *simp*
from *poly-rel-inj*[*OF b Bb*(*1*)] *B* **have** *B*: *B'* = *?I B* **by** *simp*
note *Mp* = *dd*(*2*) *hh*(*2*) *ss*(*2*) *tt*(*2*) *uu*(*2*)
note [*transfer-rule*] = *Mp*
have (=) (*D* ∗ *S* + *H* ∗ *T* =*m 1*) (*?I D* ∗ *?I S* + *?I H* ∗ *?I T* = *1*) **by** *transfer-prover*
with *1* **have** *11*: *?I D* ∗ *?I S* + *?I H* ∗ *?I T* = *1* **by** *simp*
from *dupe-monic*[*OF 11 monic-of-int-poly*[*OF mon*] *dupe*, *unfolded A B*]
have *res*: *?I A* ∗ *?I D* + *?I B* ∗ *?I H* = *?I U ?I B* = *0* ∨ *degree* (*?I B*) < *degree* (*?I D*) **by** *auto*
note [*transfer-rule*] = *Aa*(*2*) *Bb*(*2*)
have (=) (*A* ∗ *D* + *B* ∗ *H* =*m U*) (*?I A* ∗ *?I D* + *?I B* ∗ *?I H* = *?I U*)
(=) (*B* =*m 0* ∨ *degree-m B* < *degree-m D*) (*?I B* = *0* ∨ *degree* (*?I B*) < *degree* (*?I D*)) **by** *transfer-prover+*
with *res* **have** ∗: *A* ∗ *D* + *B* ∗ *H* =*m U B* =*m 0* ∨ *degree-m B* < *degree-m D* **by** *auto*
show *A* ∗ *D* + *B* ∗ *H* =*m U* **by** *fact*
have *B*: *Mp B* = *B* **using** *Mp-rel-i-Mp-to-int-poly-i assms*(*5*) *bb* **by** *blast*
from ∗(*2*) **show** *B* = *0* ∨ *degree B* < *degree D* **unfolding** *B* **using** *degree-m-le*[*of D*] **by** *auto*
qed

lemma *Mp-rel-i-of-int-poly-i*: **assumes** *Mp F* = *F*
shows *Mp-rel-i* (*of-int-poly-i ff-ops F*) *F*
by (*metis Mp-f-representative Mp-rel-iI' assms poly-rel-of-int-poly to-int-poly-i*)

lemma *dupe-monic-i-int*: **assumes** *dupe-i*: *dupe-monic-i-int ff-ops D H S T U* = (*A*,*B*)
and *1*: *D*∗*S* + *H*∗*T* =*m 1*
and *mon*: *monic D*
and *norm*: *Mp D* = *D Mp H* = *H Mp S* = *S Mp T* = *T Mp U* = *U*
shows
A ∗ *D* + *B* ∗ *H* =*m U*
B = *0* ∨ *degree B* < *degree D*
Mp A = *A*
Mp B = *B*
proof −
let *?oi* = *of-int-poly-i ff-ops*
let *?ti* = *to-int-poly-i ff-ops*
note *rel* = *norm*[*THEN Mp-rel-i-of-int-poly-i*]
obtain *a b* **where** *dupe*: *dupe-monic-i ff-ops* (*?oi D*) (*?oi H*) (*?oi S*) (*?oi T*) (*?oi U*) = (*a*,*b*) **by** *force*
from *dupe-i*[*unfolded dupe-monic-i-int-def this Let-def*] **have** *AB*: *A* = *?ti a B* = *?ti b* **by** *auto*
from *dupe-monic-i*[*OF dupe 1 mon AB rel*] *Mp-rel-i-Mp-to-int-poly-i*

show $A * D + B * H =m\ U$
$B = 0 \vee degree\ B < degree\ D$
$Mp\ A = A$
$Mp\ B = B$
unfolding *AB* **by** *auto*
qed

end

definition *dupe-monic-dynamic*
:: *int* ⇒ *int poly* ⇒ *int poly* ⇒ *int poly* ⇒ *int poly* ⇒ *int poly* ⇒ *int poly* × *int poly* **where**
dupe-monic-dynamic p = (
if p ≤ *65535*
then dupe-monic-i-int (*finite-field-ops32* (*uint32-of-int p*))
else if p ≤ *4294967295*
then dupe-monic-i-int (*finite-field-ops64* (*uint64-of-int p*))
else dupe-monic-i-int (*finite-field-ops-integer* (*integer-of-int p*)))

context *poly-mod-2*
begin

lemma *dupe-monic-i-int-finite-field-ops-integer*: **assumes**
dupe-i: *dupe-monic-i-int* (*finite-field-ops-integer* (*integer-of-int m*)) *D H S T U* = (*A*,*B*)
and *1*: $D*S + H*T =m\ 1$
and *mon*: *monic D*
and *norm*: $Mp\ D = D\ Mp\ H = H\ Mp\ S = S\ Mp\ T = T\ Mp\ U = U$
shows
$A * D + B * H =m\ U$
$B = 0 \vee degree\ B < degree\ D$
$Mp\ A = A$
$Mp\ B = B$
using *m1 mod-ring-gen.dupe-monic-i-int*[*OF*
mod-ring-locale.mod-ring-finite-field-ops-integer[*unfolded mod-ring-locale-def*],

internalize-sort 'a :: *nontriv*, *OF type-to-set*, *unfolded remove-duplicate-premise*,

cancel-type-definition, *OF* - *assms*] **by** *auto*

lemma *dupe-monic-i-int-finite-field-ops32*: **assumes**
m: *m* ≤ *65535*
and *dupe-i*: *dupe-monic-i-int* (*finite-field-ops32* (*uint32-of-int m*)) *D H S T U* = (*A*,*B*)
and *1*: $D*S + H*T =m\ 1$
and *mon*: *monic D*
and *norm*: $Mp\ D = D\ Mp\ H = H\ Mp\ S = S\ Mp\ T = T\ Mp\ U = U$
shows
$A * D + B * H =m\ U$

$B = 0 \vee$ *degree* $B <$ *degree* D
Mp $A = A$
Mp $B = B$
using *m1 mod-ring-gen.dupe-monic-i-int*[*OF*
mod-ring-locale.mod-ring-finite-field-ops32[*unfolded mod-ring-locale-def*],
internalize-sort $'a$ :: *nontriv*, *OF type-to-set*, *unfolded remove-duplicate-premise*,

cancel-type-definition, *OF - assms*] **by** *auto*

lemma *dupe-monic-i-int-finite-field-ops64*: **assumes**
m: $m \leq 4294967295$
and *dupe-i*: *dupe-monic-i-int* (*finite-field-ops64* (*uint64-of-int m*)) *D H S T U* = (*A*,*B*)
and *1*: $D*S + H*T$ =*m* *1*
and *mon*: *monic D*
and *norm*: *Mp* $D = D$ *Mp* $H = H$ *Mp* $S = S$ *Mp* $T = T$ *Mp* $U = U$
shows
$A * D + B * H$ =*m* U
$B = 0 \vee$ *degree* $B <$ *degree* D
Mp $A = A$
Mp $B = B$
using *m1 mod-ring-gen.dupe-monic-i-int*[*OF*
mod-ring-locale.mod-ring-finite-field-ops64[*unfolded mod-ring-locale-def*],
internalize-sort $'a$ :: *nontriv*, *OF type-to-set*, *unfolded remove-duplicate-premise*,

cancel-type-definition, *OF - assms*] **by** *auto*

lemma *dupe-monic-dynamic*: **assumes** *dupe*: *dupe-monic-dynamic m D H S T U* = (*A*,*B*)
and *1*: $D*S + H*T$ =*m* *1*
and *mon*: *monic D*
and *norm*: *Mp* $D = D$ *Mp* $H = H$ *Mp* $S = S$ *Mp* $T = T$ *Mp* $U = U$
shows
$A * D + B * H$ =*m* U
$B = 0 \vee$ *degree* $B <$ *degree* D
Mp $A = A$
Mp $B = B$
using *dupe*
dupe-monic-i-int-finite-field-ops32[*OF - - 1 mon norm*, *of A B*]
dupe-monic-i-int-finite-field-ops64[*OF - - 1 mon norm*, *of A B*]
dupe-monic-i-int-finite-field-ops-integer[*OF - 1 mon norm*, *of A B*]
unfolding *dupe-monic-dynamic-def* **by** (*auto split*: *if-splits*)
end

context *poly-mod*
begin

definition *dupe-monic-int* :: *int poly* ⇒ *int poly* ⇒ *int poly* ⇒ *int poly* ⇒ *int poly*

$\Rightarrow$
int poly $*$ *int poly* **where**
dupe-monic-int D H S T U = (*case pdivmod-monic* (*Mp* (*T* $*$ *U*)) *D of* (*q*,*r*) $\Rightarrow$
(*Mp* (*S* $*$ *U* + *H* $*$ *q*), *Mp r*))

end

declare *poly-mod.dupe-monic-int-def*[*code*]

Old direct proof on int poly. It does not permit to change implementation. This proof is still present, since we did not export the uniqueness part from the type-based uniqueness result $[\![?D * ?S + ?H * ?T = 1$; *monic* $?D$; *dupe-monic* $?D\ ?H\ ?S\ ?T\ ?U = (?A, ?B)$; *comm-monoid-mult-class.coprime* $?D\ ?H$; $?A' * ?D + ?B' * ?H = ?U$; $?B' = 0 \lor$ *degree* $?B' <$ *degree* $?D]\!] \Longrightarrow ?A' = ?A$

$[\![?D * ?S + ?H * ?T = 1$; *monic* $?D$; *dupe-monic* $?D\ ?H\ ?S\ ?T\ ?U = (?A, ?B)$; *comm-monoid-mult-class.coprime* $?D\ ?H$; $?A' * ?D + ?B' * ?H = ?U$; $?B' = 0 \lor$ *degree* $?B' <$ *degree* $?D]\!] \Longrightarrow ?B' = ?B$ via the various relations.

lemma (**in** *poly-mod-2*) *dupe-monic-int*: **assumes** *1*: *D*$*$*S* + *H*$*$*T* =*m 1*
and *mon*: *monic D*
and *dupe*: *dupe-monic-int D H S T U* = (*A*,*B*)
shows *A* $*$ *D* + *B* $*$ *H* =*m U B* = *0* $\lor$ *degree B* < *degree D Mp A* = *A Mp B* = *B*
coprime-m D H $\Longrightarrow$ *A'* $*$ *D* + *B'* $*$ *H* =*m U* $\Longrightarrow$ *B'* = *0* $\lor$ *degree B'* < *degree D* $\Longrightarrow$ *Mp D* = *D*
$\Longrightarrow$ *Mp A'* = *A'* $\Longrightarrow$ *Mp B'* = *B'* $\Longrightarrow$ *prime m*
$\Longrightarrow$ *A'* = *A* $\land$ *B'* = *B*
proof −
obtain *Q R* **where** *div*: *pdivmod-monic* (*Mp* (*T* $*$ *U*)) *D* = (*Q*,*R*) **by** *force*
from *dupe*[*unfolded dupe-monic-int-def div split*]
have *A*: *A* = *Mp* (*S* $*$ *U* + *H* $*$ *Q*) **and** *B*: *B* = *Mp R* **by** *auto*
from *pdivmod-monic*[*OF mon div*] **have** *TU*: *Mp* (*T* $*$ *U*) = *D* $*$ *Q* + *R* **and**
deg: *R* = *0* $\lor$ *degree R* < *degree D* **by** *auto*
hence *Mp R* = *Mp* (*Mp* (*T* $*$ *U*) − *D* $*$ *Q*) **by** *simp*
also have ... = *Mp* (*T* $*$ *U* − *Mp* (*Mp* (*Mp D* $*$ *Q*))) **unfolding** *Mp-Mp* **unfolding** *minus-Mp*
using *minus-Mp mult-Mp* **by** *metis*
also have ... = *Mp* (*T* $*$ *U* − *D* $*$ *Q*) **by** *simp*
finally have *r*: *Mp R* = *Mp* (*T* $*$ *U* − *D* $*$ *Q*) **by** *simp*
have *Mp* (*A* $*$ *D* + *B* $*$ *H*) = *Mp* (*Mp* (*A* $*$ *D*) + *Mp* (*B* $*$ *H*)) **by** *simp*
also have *Mp* (*A* $*$ *D*) = *Mp* ((*S* $*$ *U* + *H* $*$ *Q*) $*$ *D*) **unfolding** *A* **by** *simp*
also have *Mp* (*B* $*$ *H*) = *Mp* (*Mp R* $*$ *Mp H*) **unfolding** *B* **by** *simp*
also have ... = *Mp* ((*T* $*$ *U* − *D* $*$ *Q*) $*$ *H*) **unfolding** *r* **by** *simp*
also have *Mp* (*Mp* ((*S* $*$ *U* + *H* $*$ *Q*) $*$ *D*) + *Mp* ((*T* $*$ *U* − *D* $*$ *Q*) $*$ *H*)) =
Mp ((*S* $*$ *U* + *H* $*$ *Q*) $*$ *D* + (*T* $*$ *U* − *D* $*$ *Q*) $*$ *H*) **by** *simp*
also have (*S* $*$ *U* + *H* $*$ *Q*) $*$ *D* + (*T* $*$ *U* − *D* $*$ *Q*) $*$ *H* = (*D* $*$ *S* + *H* $*$ *T*) $*$ *U*

by (*simp add*: *field-simps*)
also have *Mp* ... = *Mp* (*Mp* (*D* ∗ *S* + *H* ∗ *T*) ∗ *U*) **by** *simp*
also have *Mp* (*D* ∗ *S* + *H* ∗ *T*) = *1* **using** *1* **by** *simp*
finally show *eq*: *A* ∗ *D* + *B* ∗ *H* =*m* *U* **by** *simp*
have *id*: *degree-m* (*Mp R*) = *degree-m R* **by** *simp*
have *id′*: *degree D* = *degree-m D* **using** *mon* **by** *simp*
show *degB*: *B* = *0* ∨ *degree B* < *degree D* **using** *deg* **unfolding** *B id id′*
using *degree-m-le*[*of R*] **by** (*cases R* = *0*, *auto*)
show *Mp*: *Mp A* = *A Mp B* = *B* **unfolding** *A B* **by** *auto*
assume *another*: *A′* ∗ *D* + *B′* ∗ *H* =*m* *U* **and** *degB′*: *B′* = *0* ∨ *degree B′* < *degree D*
and *norm*: *Mp A′* = *A′ Mp B′* = *B′* **and** *cop*: *coprime-m D H* **and** *D*: *Mp D* = *D*
and *prime*: *prime m*
from *degB Mp D* **have** *degB*: *B* =*m* *0* ∨ *degree-m B* < *degree-m D* **by** *auto*
from *degB′ Mp D norm* **have** *degB′*: *B′* =*m* *0* ∨ *degree-m B′* < *degree-m D* **by** *auto*
from *mon D* **have** *D0*: ¬ (*D* =*m* *0*) **by** *auto*
from *prime* **interpret** *poly-mod-prime m* **by** *unfold-locales*
from *another eq* **have** *A′* ∗ *D* + *B′* ∗ *H* =*m* *A* ∗ *D* + *B* ∗ *H* **by** *simp*
from *uniqueness-poly-equality*[*OF cop degB′ degB D0 this*]
show *A′* = *A* ∧ *B′* = *B* **unfolding** *norm Mp* **by** *auto*
qed

lemma *coprime-bezout-coefficients*:
assumes *cop*: *coprime f g*
and *ext*: *bezout-coefficients f g* = (*a*, *b*)
shows *a* ∗ *f* + *b* ∗ *g* = *1*
using *assms bezout-coefficients* [*of f g a b*]
by *simp*

lemma (**in** *poly-mod-prime-type*) *bezout-coefficients-mod-int*: **assumes** *f*: (*F* :: *′a mod-ring poly*) = *of-int-poly f*
and *g*: (*G* :: *′a mod-ring poly*) = *of-int-poly g*
and *cop*: *coprime-m f g*
and *fact*: *bezout-coefficients F G* = (*A*,*B*)
and *a*: *a* = *to-int-poly A*
and *b*: *b* = *to-int-poly B*
shows *f* ∗ *a* + *g* ∗ *b* =*m* *1*
proof −
have *f*[*transfer-rule*]: *MP-Rel f F* **unfolding** *f MP-Rel-def* **by** (*simp add*: *Mp-f-representative*)
have *g*[*transfer-rule*]: *MP-Rel g G* **unfolding** *g MP-Rel-def* **by** (*simp add*: *Mp-f-representative*)
have [*transfer-rule*]: *MP-Rel a A* **unfolding** *a MP-Rel-def* **by** (*rule Mp-to-int-poly*)
have [*transfer-rule*]: *MP-Rel b B* **unfolding** *b MP-Rel-def* **by** (*rule Mp-to-int-poly*)
from *cop* **have** *coprime F G* **using** *coprime-MP-Rel*[*unfolded rel-fun-def*] *f g* **by** *auto*
from *coprime-bezout-coefficients* [*OF this fact*]

have $A * F + B * G = 1$ **.**
from *this* [*untransferred*]
show *?thesis* **by** (*simp add*: *ac-simps*)
qed

definition *bezout-coefficients-i* :: $'i$ *arith-ops-record* $\Rightarrow$ $'i$ *list* $\Rightarrow$ $'i$ *list* $\Rightarrow$ $'i$ *list* $\times$ $'i$ *list* **where**
bezout-coefficients-i ff-ops f g = *fst* (*euclid-ext-poly-i ff-ops f g*)

definition *euclid-ext-poly-mod-main* :: *int* $\Rightarrow$ $'a$ *arith-ops-record* $\Rightarrow$ *int poly* $\Rightarrow$ *int poly* $\Rightarrow$ *int poly* $\times$ *int poly* **where**
euclid-ext-poly-mod-main p ff-ops f g = (*case bezout-coefficients-i ff-ops* (*of-int-poly-i ff-ops f*) (*of-int-poly-i ff-ops g*) *of*
(*a*,*b*) $\Rightarrow$ (*to-int-poly-i ff-ops a*, *to-int-poly-i ff-ops b*))

definition *euclid-ext-poly-dynamic* :: *int* $\Rightarrow$ *int poly* $\Rightarrow$ *int poly* $\Rightarrow$ *int poly* $\times$ *int poly* **where**
euclid-ext-poly-dynamic p = (
if p $\leq$ *65535*
then euclid-ext-poly-mod-main p (*finite-field-ops32* (*uint32-of-int p*))
else if p $\leq$ *4294967295*
then euclid-ext-poly-mod-main p (*finite-field-ops64* (*uint64-of-int p*))
else euclid-ext-poly-mod-main p (*finite-field-ops-integer* (*integer-of-int p*)))

context *prime-field-gen*
begin
lemma *bezout-coefficients-i*[*transfer-rule*]:
(*poly-rel* ===> *poly-rel* ===> *rel-prod poly-rel poly-rel*)
(*bezout-coefficients-i ff-ops*) *bezout-coefficients*
unfolding *bezout-coefficients-i-def bezout-coefficients-def*
by *transfer-prover*

lemma *bezout-coefficients-i-sound*: **assumes** *f*: f' = *of-int-poly-i ff-ops f Mp f* = *f*
and *g*: g' = *of-int-poly-i ff-ops g Mp g* = *g*
and *cop*: *coprime-m f g*
and *res*: *bezout-coefficients-i ff-ops* f' g' = (a',b')
and *a*: *a* = *to-int-poly-i ff-ops* a'
and *b*: *b* = *to-int-poly-i ff-ops* b'
shows $f * a + g * b$ =m *1*
Mp a = *a Mp b* = *b*
proof −
from *f* **have** f': f' = *of-int-poly-i ff-ops* (*Mp f*) **by** *simp*
define f'' **where** $f'' \equiv$ *of-int-poly* (*Mp f*) :: $'a$ *mod-ring poly*
have f'': f'' = *of-int-poly f* **unfolding** f''*-def f* **by** *simp*
have *rel-f*[*transfer-rule*]: *poly-rel* f' f''
by (*rule poly-rel-of-int-poly*[*OF* f'], *simp add*: f'' *f*)
from *g* **have** g': g' = *of-int-poly-i ff-ops* (*Mp g*) **by** *simp*
define g'' **where** $g'' \equiv$ *of-int-poly* (*Mp g*) :: $'a$ *mod-ring poly*
have g'': g'' = *of-int-poly g* **unfolding** g''*-def g* **by** *simp*

have *rel-g*[*transfer-rule*]: *poly-rel g′ g′′*
by (*rule poly-rel-of-int-poly*[*OF g′*], *simp add*: *g′′ g*)
obtain *a′′ b′′* **where** *eucl*: *bezout-coefficients f′′ g′′* = (*a′′*,*b′′*) **by** *force*
from *bezout-coefficients-i*[*unfolded rel-fun-def rel-prod-conv*, *rule-format*, *OF rel-f rel-g*,
unfolded res split eucl]
have *rel*[*transfer-rule*]: *poly-rel a′ a′′ poly-rel b′ b′′* **by** *auto*
with *to-int-poly-i* **have** *a*: *a* = *to-int-poly a′′*
and *b*: *b* = *to-int-poly b′′* **unfolding** *a b* **by** *auto*
from *bezout-coefficients-mod-int* [*OF f′′ g′′ cop eucl a b*]
show *f* ∗ *a* + *g* ∗ *b* =*m 1* **.**
show *Mp a* = *a Mp b* = *b* **unfolding** *a b* **by** (*auto simp*: *Mp-to-int-poly*)
qed

lemma *euclid-ext-poly-mod-main*: **assumes** *cop*: *coprime-m f g*
and *f*: *Mp f* = *f* **and** *g*: *Mp g* = *g*
and *res*: *euclid-ext-poly-mod-main m ff-ops f g* = (*a*,*b*)
shows *f* ∗ *a* + *g* ∗ *b* =*m 1*
Mp a = *a Mp b* = *b*
proof −
obtain *a′ b′* **where** *res′*: *bezout-coefficients-i ff-ops* (*of-int-poly-i ff-ops f*)
(*of-int-poly-i ff-ops g*) = (*a′*, *b′*) **by** *force*
show *f* ∗ *a* + *g* ∗ *b* =*m 1*
Mp a = *a Mp b* = *b*
by (*insert bezout-coefficients-i-sound*[*OF refl f refl g cop res′*]
res [*unfolded euclid-ext-poly-mod-main-def res′*], *auto*)
qed

end

context *poly-mod-prime* **begin**

lemmas *euclid-ext-poly-mod-integer* = *prime-field-gen.euclid-ext-poly-mod-main*
[*OF prime-field.prime-field-finite-field-ops-integer*,
unfolded prime-field-def mod-ring-locale-def poly-mod-type-simps, *internalize-sort ′a* :: *prime-card*, *OF type-to-set*, *unfolded remove-duplicate-premise*, *cancel-type-definition*, *OF non-empty*]

lemmas *euclid-ext-poly-mod-uint32* = *prime-field-gen.euclid-ext-poly-mod-main*
[*OF prime-field.prime-field-finite-field-ops32*,
unfolded prime-field-def mod-ring-locale-def poly-mod-type-simps, *internalize-sort ′a* :: *prime-card*, *OF type-to-set*, *unfolded remove-duplicate-premise*, *cancel-type-definition*, *OF non-empty*]

lemmas *euclid-ext-poly-mod-uint64* = *prime-field-gen.euclid-ext-poly-mod-main*[*OF prime-field.prime-field-finite-field-ops64*,
unfolded prime-field-def mod-ring-locale-def poly-mod-type-simps, *internalize-sort ′a* :: *prime-card*, *OF type-to-set*, *unfolded remove-duplicate-premise*, *cancel-type-definition*, *OF non-empty*]

lemma *euclid-ext-poly-dynamic*:
  assumes *cop*: *coprime-m f g* **and** *f*: *Mp f = f* **and** *g*: *Mp g = g*
    and *res*: *euclid-ext-poly-dynamic p f g = (a,b)*
  shows *f ∗ a + g ∗ b =m 1*
    Mp a = a Mp b = b
  using *euclid-ext-poly-mod-integer[OF cop f g, of p a b]*
    euclid-ext-poly-mod-uint32[OF - cop f g, of p a b]
    euclid-ext-poly-mod-uint64[OF - cop f g, of p a b]
    res[unfolded euclid-ext-poly-dynamic-def] **by** (*auto split*: *if-splits*)

end

lemma *range-sum-prod*: **assumes** *xy*: *x ∈ {0..<q} (y :: int) ∈ {0..<p}*
  shows *x + q ∗ y ∈ {0..<p ∗ q}*
proof −
  {
    fix *x q :: int*
    have *x ∈ {0 ..< q} ⟷ 0 ≤ x ∧ x < q* **by** *auto*
  } **note** *id = this*
  from *xy* **have** *0*: *0 ≤ x + q ∗ y* **by** *auto*
  have *x + q ∗ y ≤ q − 1 + q ∗ y* **using** *xy* **by** *simp*
  also have *q ∗ y ≤ q ∗ (p − 1)* **using** *xy* **by** *auto*
  finally have *x + q ∗ y ≤ q − 1 + q ∗ (p − 1)* **by** *auto*
  also have ... *= p ∗ q − 1* **by** (*simp add*: *field-simps*)
  finally show *?thesis* **using** *0* **by** *auto*
qed

context
  fixes *C :: int poly*
begin

context
  fixes *p :: int* **and** *S T D1 H1 :: int poly*
begin

fun *linear-hensel-main* **where**
  linear-hensel-main (Suc 0) = (D1,H1)
| *linear-hensel-main (Suc n) = (*
      let (D,H) = linear-hensel-main n;
        q = p ^ n;
        U = poly-mod.Mp p (sdiv-poly (C − D ∗ H) q); — *H2 + H3*
        (A,B) = poly-mod.dupe-monic-int p D1 H1 S T U
      in (D + smult q B, H + smult q A)) — *H4*
    | *linear-hensel-main 0 = (D1,H1)*

lemma *linear-hensel-main*: **assumes** *1*: *poly-mod.eq-m p (D1 ∗ S + H1 ∗ T) 1*
  and *equiv*: *poly-mod.eq-m p (D1 ∗ H1) C*
  and *monD1*: *monic D1*

and *normDH1*: *poly-mod.Mp p D1 = D1 poly-mod.Mp p H1 = H1*
and *res*: *linear-hensel-main n = (D,H)*
and *n*: $n \neq 0$
and *prime*: *prime p* — $p > 1$ suffices if one does not need uniqueness
and *cop*: *poly-mod.coprime-m p D1 H1*
shows *poly-mod.eq-m* $(p\hat{\ }n)$ $(D * H)$ *C*
$\wedge$ *monic D*
$\wedge$ *poly-mod.eq-m p D D1* $\wedge$ *poly-mod.eq-m p H H1*
$\wedge$ *poly-mod.Mp* $(p\hat{\ }n)$ $D = D$
$\wedge$ *poly-mod.Mp* $(p\hat{\ }n)$ $H = H$ $\wedge$
(*poly-mod.eq-m* $(p\hat{\ }n)$ $(D' * H')$ $C \longrightarrow$
poly-mod.eq-m p D' *D1* $\longrightarrow$
poly-mod.eq-m p H' *H1* $\longrightarrow$
poly-mod.Mp $(p\hat{\ }n)$ $D' = D' \longrightarrow$
poly-mod.Mp $(p\hat{\ }n)$ $H' = H' \longrightarrow$ *monic* $D' \longrightarrow D' = D \wedge H' = H$)

using *res n*
proof (*induct n arbitrary*: $D\ H\ D'\ H'$)
case (*Suc n* $D'\ H'\ D''\ H''$)
show *?case*
proof (*cases* $n = 0$)
case *True*
with *Suc equiv monD1 normDH1* **show** *?thesis* **by** *auto*
next
case *False*
hence *n*: $n \neq 0$ **by** *auto*
let $?q = p\hat{\ }n$
let $?pq = p * p\hat{\ }n$
from *prime* **have** *p*: $p > 1$ **using** *prime-gt-1-int* **by** *force*
from *n p* **have** *q*: $?q > 1$ **by** *auto*
from *n p* **have** *pq*: $?pq > 1$ **by** (*metis power-gt1-lemma*)
interpret *p*: *poly-mod-2 p* **using** *p* **unfolding** *poly-mod-2-def* **.**
interpret *q*: *poly-mod-2 ?q* **using** *q* **unfolding** *poly-mod-2-def* **.**
interpret *pq*: *poly-mod-2 ?pq* **using** *pq* **unfolding** *poly-mod-2-def* **.**
obtain *D H* **where** *rec*: *linear-hensel-main n = (D,H)* **by** *force*
obtain *V* **where** *V*: *sdiv-poly* $(C - D * H)$ $?q = V$ **by** *force*
obtain *U* **where** *U*: *p.Mp* (*sdiv-poly* $(C - D * H)$ $?q$) $= U$ **by** *auto*
obtain *A B* **where** *dupe*: *p.dupe-monic-int D1 H1 S T U = (A,B)* **by** *force*
note *IH = Suc(1)[OF rec n]*
from *IH*
have *CDH*: *q.eq-m* $(D * H)$ *C*
and *monD*: *monic D*
and *p-eq*: *p.eq-m D D1 p.eq-m H H1*
and *norm*: *q.Mp D = D q.Mp H = H* **by** *auto*
from *n* **obtain** *k* **where** *n*: *n = Suc k* **by** (*cases n*, *auto*)
have *qq*: $?q * ?q = ?pq * p\hat{\ }k$ **unfolding** *n* **by** *simp*
from *Suc(2)[unfolded n linear-hensel-main.simps, folded n, unfolded rec split Let-def U dupe]*
have D': $D' = D +$ *smult ?q B* **and** H': $H' = H +$ *smult ?q A* **by** *auto*

note *dupe* = *p.dupe-monic-int*[*OF 1 monD1 dupe*]
from *CDH* **have** *q.Mp C* − *q.Mp* (*D* ∗ *H*) = *0* **by** *simp*
hence *q.Mp* (*q.Mp C* − *q.Mp* (*D* ∗ *H*)) = *0* **by** *simp*
hence *q.Mp* (*C* − *D*∗*H*) = *0* **by** *simp*
from *q.Mp-0-smult-sdiv-poly*[*OF this*] **have** *CDHq*: *smult ?q* (*sdiv-poly* (*C* − *D* ∗ *H*) *?q*) = *C* − *D* ∗ *H* **.**
have *ADBHU*: *p.eq-m* (*A* ∗ *D* + *B* ∗ *H*) *U* **using** *p-eq dupe*(*1*)
by (*metis* (*mono-tags*, *lifting*) *p.mult-Mp*(*2*) *poly-mod.plus-Mp*)
have *pq.Mp* (*D′* ∗ *H′*) = *pq.Mp* ((*D* + *smult ?q B*) ∗ (*H* + *smult ?q A*))
unfolding *D′ H′* **by** *simp*
also have (*D* + *smult ?q B*) ∗ (*H* + *smult ?q A*) = (*D* ∗ *H* + *smult ?q* (*A* ∗ *D* + *B* ∗ *H*)) + *smult* (*?q* ∗ *?q*) (*A* ∗ *B*)
by (*simp add*: *field-simps smult-distribs*)
also have *pq.Mp* ... = *pq.Mp* (*D* ∗ *H* + *pq.Mp* (*smult ?q* (*A* ∗ *D* + *B* ∗ *H*)) + *pq.Mp* (*smult* (*?q* ∗ *?q*) (*A* ∗ *B*)))
using *pq.plus-Mp* **by** *metis*
also have *pq.Mp* (*smult* (*?q* ∗ *?q*) (*A* ∗ *B*)) = *0* **unfolding** *qq*
by (*metis pq.Mp-smult-m-0 smult-smult*)
finally have *DH′*: *pq.Mp* (*D′* ∗ *H′*) = *pq.Mp* (*D* ∗ *H* + *pq.Mp* (*smult ?q* (*A* ∗ *D* + *B* ∗ *H*))) **by** *simp*
also have *pq.Mp* (*smult ?q* (*A* ∗ *D* + *B* ∗ *H*)) = *pq.Mp* (*smult ?q U*)
using *p.Mp-lift-modulus*[*OF ADBHU*, *of ?q*] **by** *simp*
also have ... = *pq.Mp* (*C* − *D* ∗ *H*)
unfolding *arg-cong*[*OF CDHq*, *of pq.Mp*, *symmetric*] *U*[*symmetric*] *V*
by (*rule p.Mp-lift-modulus*[*of - - ?q*], *auto*)
also have *pq.Mp* (*D* ∗ *H* + *pq.Mp* (*C* − *D* ∗ *H*)) = *pq.Mp C* **by** *simp*
finally have *CDH*: *pq.eq-m C* (*D′* ∗ *H′*) **by** *simp*

have *deg*: *degree D1* = *degree D* **using** *p-eq*(*1*) *monD1 monD*
by (*metis p.monic-degree-m*)
have *mon*: *monic D′* **unfolding** *D′* **using** *dupe*(*2*) *monD* **unfolding** *deg* **by** (*rule monic-smult-add-small*)
have *normD′*: *pq.Mp D′* = *D′*
unfolding *D′ pq.Mp-ident-iff poly-mod.Mp-coeff plus-poly.rep-eq coeff-smult*
proof
fix *i*
from *norm*(*1*) *dupe*(*4*) **have** *coeff D i* ∈ {*0..*<*?q*} *coeff B i* ∈ {*0..*<*p*}
unfolding *p.Mp-ident-iff q.Mp-ident-iff* **by** *auto*
thus *coeff D i* + *?q* ∗ *coeff B i* ∈ {*0..*< *?pq*} **by** (*rule range-sum-prod*)
qed
have *normH′*: *pq.Mp H′* = *H′*
unfolding *H′ pq.Mp-ident-iff poly-mod.Mp-coeff plus-poly.rep-eq coeff-smult*
proof
fix *i*
from *norm*(*2*) *dupe*(*3*) **have** *coeff H i* ∈ {*0..*<*?q*} *coeff A i* ∈ {*0..*<*p*}
unfolding *p.Mp-ident-iff q.Mp-ident-iff* **by** *auto*
thus *coeff H i* + *?q* ∗ *coeff A i* ∈ {*0..*< *?pq*} **by** (*rule range-sum-prod*)
qed
have *eq*: *p.eq-m D D′ p.eq-m H H′* **unfolding** *D′ H′ n*

poly-eq-iff p.Mp-coeff p.M-def **by** (*auto simp*: *field-simps*)
with *p-eq* **have** *eq*: *p.eq-m D′ D1 p.eq-m H′ H1* **by** *auto*
{
assume *CDH′′*: *pq.eq-m C* (*D′′* ∗ *H′′*)
and *DH1′′*: *p.eq-m D1 D′′ p.eq-m H1 H′′*
and *norm′′*: *pq.Mp D′′* = *D′′ pq.Mp H′′* = *H′′*
and *monD′′*: *monic D′′*
from *q.Dp-Mp-eq*[*of D′′*] **obtain** *d B′* **where** *D′′*: *D′′* = *q.Mp d* + *smult ?q B′* **by** *auto*
from *q.Dp-Mp-eq*[*of H′′*] **obtain** *h A′* **where** *H′′*: *H′′* = *q.Mp h* + *smult ?q A′* **by** *auto*
{
fix *A B*
assume ∗: *pq.Mp* (*q.Mp A* + *smult ?q B*) = *q.Mp A* + *smult ?q B*
have *p.Mp B* = *B* **unfolding** *p.Mp-ident-iff*
proof
fix *i*
from *arg-cong*[*OF* ∗, *of* λ *f*. *coeff f i*, *unfolded pq.Mp-coeff pq.M-def*]
have *coeff* (*q.Mp A* + *smult ?q B*) *i* ∈ {*0* ..< *?pq*} **using** ∗ *pq.Mp-ident-iff* **by** *blast*
hence *sum*: *coeff* (*q.Mp A*) *i* + *?q* ∗ *coeff B i* ∈ {*0* ..< *?pq*} **by** *auto*
have *q.Mp* (*q.Mp A*) = *q.Mp A* **by** *auto*
from *this*[*unfolded q.Mp-ident-iff*] **have** *A*: *coeff* (*q.Mp A*) *i* ∈ {*0* ..< *p^n*} **by** *auto*
{
assume *coeff B i* < *0* **hence** *coeff B i* ≤ −*1* **by** *auto*
from *mult-left-mono*[*OF this*, *of ?q*] *q.m1* **have** *?q* ∗ *coeff B i* ≤ −*?q* **by** *simp*
with *A sum* **have** *False* **by** *auto*
} **hence** *coeff B i* ≥ *0* **by** *force*
moreover
{
assume *coeff B i* ≥ *p*
from *mult-left-mono*[*OF this*, *of ?q*] *q.m1* **have** *?q* ∗ *coeff B i* ≥ *?pq* **by** *simp*
with *A sum* **have** *False* **by** *auto*
} **hence** *coeff B i* < *p* **by** *force*
ultimately show *coeff B i* ∈ {*0* ..< *p*} **by** *auto*
qed
} **note** *norm-convert* = *this*
from *norm-convert*[*OF norm′′*(*1*)[*unfolded D′′*]] **have** *normB′*: *p.Mp B′* = *B′*
.
from *norm-convert*[*OF norm′′*(*2*)[*unfolded H′′*]] **have** *normA′*: *p.Mp A′* = *A′*
.
let *?d* = *q.Mp d*
let *?h* = *q.Mp h*
{
assume *lt*: *degree ?d* < *degree B′*
hence *eq*: *degree D′′* = *degree B′* **unfolding** *D′′* **using** *q.m1 p.m1*

by (*subst degree-add-eq-right*, *auto*)
from *lt* **have** [*simp*]: *coeff ?d* (*degree B'*) = *0* **by** (*rule coeff-eq-0*)
from *monD''*[*unfolded eq*, *unfolded D''*, *simplified*] *False q.m1 lt* **have** *False*
by (*metis mod-mult-self1-is-0 poly-mod.M-def q.M-1 zero-neq-one*)
}
hence *deg-dB'*: *degree ?d* ≥ *degree B'* **by** *presburger*
{
assume *eq*: *degree ?d* = *degree B'* **and** *B'*: *B'* ≠ *0*
let *?B* = *coeff B'* (*degree B'*)
from *normB'*[*unfolded p.Mp-ident-iff*, *rule-format*, *of degree B'*] *B'*
have *?B* ∈ {*0..<p*} − {*0*} **by** *simp*
hence *bnds*: *?B* > *0 ?B* < *p* **by** *auto*
have *degD''*: *degree D''* ≤ *degree ?d* **unfolding** *D''* **using** *eq* **by** (*simp add*: *degree-add-le*)
have *?q* ∗ *?B* ≥ *1* ∗ *1* **by** (*rule mult-mono*, *insert q.m1 bnds*, *auto*)
moreover have *coeff D''* (*degree ?d*) = *1* + *?q* ∗ *?B* **using** *monD''*
unfolding *D''* **using** *eq*
by (*metis D'' coeff-smult monD'' plus-poly.rep-eq poly-mod.Dp-Mp-eq*
poly-mod.degree-m-eq-monic poly-mod.plus-Mp(*1*)
q.Mp-smult-m-0 q.m1 q.monic-Mp q.plus-Mp(*2*))
ultimately have *gt*: *coeff D''* (*degree ?d*) > *1* **by** *auto*
hence *coeff D''* (*degree ?d*) ≠ *0* **by** *auto*
hence *degree D''* ≥ *degree ?d* **by** (*rule le-degree*)
with *degree-add-le-max*[*of ?d smult ?q B'*, *folded D''*] *eq*
have *deg*: *degree D''* = *degree ?d* **using** *degD''* **by** *linarith*
from *gt*[*folded this*] **have** ¬ *monic D''* **by** *auto*
with *monD''* **have** *False* **by** *auto*
}
with *deg-dB'* **have** *deg-dB2*: *B'* = *0* ∨ *degree B'* < *degree ?d* **by** *fastforce*
have *d*: *q.Mp D''* = *?d* **unfolding** *D''*
by (*metis add.right-neutral poly-mod.Mp-smult-m-0 poly-mod.plus-Mp*)
have *h*: *q.Mp H''* = *?h* **unfolding** *H''*
by (*metis add.right-neutral poly-mod.Mp-smult-m-0 poly-mod.plus-Mp*)
from *CDH''* **have** *pq.Mp C* = *pq.Mp* (*D''* ∗ *H''*) **by** *simp*
from *arg-cong*[*OF this*, *of q.Mp*]
have *q.Mp C* = *q.Mp* (*D''* ∗ *H''*)
using *p.m1 q.Mp-product-modulus* **by** *auto*
also have ... = *q.Mp* (*q.Mp D''* ∗ *q.Mp H''*) **by** *simp*
also have ... = *q.Mp* (*?d* ∗ *?h*) **unfolding** *d h* **by** *simp*
finally have *eqC*: *q.eq-m* (*?d* ∗ *?h*) *C* **by** *auto*
have *d1*: *p.eq-m ?d D1* **unfolding** *d*[*symmetric*] **using** *DH1''*
using *assms*(*4*) *n p.Mp-product-modulus p.m1* **by** *auto*
have *h1*: *p.eq-m ?h H1* **unfolding** *h*[*symmetric*] **using** *DH1''*
using *assms*(*5*) *n p.Mp-product-modulus p.m1* **by** *auto*
have *mond*: *monic* (*q.Mp d*) **using** *monD'' deg-dB2* **unfolding** *D''*
using *d q.monic-Mp*[*OF monD''*] **by** *simp*
from *eqC d1 h1 mond IH*[*of q.Mp d q.Mp h*] **have** *IH*: *?d* = *D ?h* = *H* **by** *auto*
from *deg-dB2*[*unfolded IH*] **have** *degB'*: *B'* = *0* ∨ *degree B'* < *degree D* **by**

auto
  from *IH* **have** D'': $D'' = D + smult\ ?q\ B'$ **and** H'': $H'' = H + smult\ ?q\ A'$
    unfolding $D''\ H''$ **by** *auto*
  have $pq.Mp\ (D'' * H'') = pq.Mp\ (D' * H')$ **using** CDH'' *CDH* **by** *simp*
  also have $pq.Mp\ (D'' * H'') = pq.Mp\ ((D + smult\ ?q\ B') * (H + smult\ ?q\ A'))$
    unfolding $D''\ H''$ **by** *simp*
  also have $(D + smult\ ?q\ B') * (H + smult\ ?q\ A') = (D * H + smult\ ?q\ (A' * D + B' * H)) + smult\ (?q * ?q)\ (A' * B')$
    by (*simp add*: *field-simps smult-distribs*)
  also have $pq.Mp \ldots = pq.Mp\ (D * H + pq.Mp\ (smult\ ?q\ (A' * D + B' * H)) + pq.Mp\ (smult\ (?q * ?q)\ (A' * B')))$
    using *pq.plus-Mp* **by** *metis*
  also have $pq.Mp\ (smult\ (?q * ?q)\ (A' * B')) = 0$ **unfolding** *qq*
    by (*metis pq.Mp-smult-m-0 smult-smult*)
  finally have $pq.Mp\ (D * H + pq.Mp\ (smult\ ?q\ (A' * D + B' * H)))$
    $= pq.Mp\ (D * H + pq.Mp\ (smult\ ?q\ (A * D + B * H)))$ **unfolding** DH' **by** *simp*
  hence $pq.Mp\ (smult\ ?q\ (A' * D + B' * H)) = pq.Mp\ (smult\ ?q\ (A * D + B * H))$
    by (*metis* (*no-types*, *lifting*) *add-diff-cancel-left'* *poly-mod.minus-Mp*(*1*) *poly-mod.plus-Mp*(*2*))
  hence $p.Mp\ (A' * D + B' * H) = p.Mp\ (A * D + B * H)$ **unfolding** *poly-eq-iff p.Mp-coeff pq.Mp-coeff coeff-smult*
    by (*insert p*, *auto simp*: *p.M-def pq.M-def*)
  hence $p.Mp\ (A' * D1 + B' * H1) = p.Mp\ (A * D1 + B * H1)$ **using** *p-eq*
    by (*metis p.mult-Mp*(*2*) *poly-mod.plus-Mp*)
  hence *eq*: $p.eq\text{-}m\ (A' * D1 + B' * H1)\ U$ **using** *dupe*(*1*) **by** *auto*
  have *degree D* = *degree D1* **using** *monD monD1*
    arg-cong[*OF p-eq*(*1*), *of degree*]
    p.degree-m-eq-monic[*OF - p.m1*] **by** *auto*
  hence $B' = 0 \lor degree\ B' < degree\ D1$ **using** $degB'$ **by** *simp*
  from *dupe*(*5*)[*OF cop eq this normDH1*(*1*) $normA'$ $normB'$ *prime*] **have** $A' = A\ B' = B$ **by** *auto*
  hence $D'' = D'\ H'' = H'$ **unfolding** $D''\ H''\ D'\ H'$ **by** *auto*
 }
 thus *?thesis* **using** $normD'$ $normH'$ *CDH mon eq* **by** *simp*
qed
qed *simp*
end
end

definition *linear-hensel-binary* :: *int* ⇒ *nat* ⇒ *int poly* ⇒ *int poly* ⇒ *int poly* ⇒ *int poly* × *int poly* **where**
 linear-hensel-binary p n C D H = (*let*
  (S,T) = *euclid-ext-poly-dynamic p D H*
  in linear-hensel-main C p S T D H n)

lemma (**in** *poly-mod-prime*) *unique-hensel-binary*:

assumes *prime*: *prime p*
and *cop*: *coprime-m D H* **and** *eq*: *eq-m* $(D * H)$ *C*
and *normalized-input*: *Mp D* $=$ *D Mp H* $=$ *H*
and *monic-input*: *monic D*
and *n*: $n \neq 0$
shows $\exists!$ (D',H'). — D', H' are computed via *linear-hensel-binary*
poly-mod.eq-m $(p\hat{}n)$ $(D' * H')$ C — the main result: equivalence mod $p\hat{}n$
$\wedge$ *monic* D' — monic output
$\wedge$ *eq-m D* D' $\wedge$ *eq-m H* H' — apply *'mod p'* on D' and H' yields D and H again
$\wedge$ *poly-mod.Mp* $(p\hat{}n)$ $D' = D'$ $\wedge$ *poly-mod.Mp* $(p\hat{}n)$ $H' = H'$ — output is normalized
proof −
obtain D' H' **where** *hensel-result*: *linear-hensel-binary p n C D H* $= (D',H')$ **by** *force*
from *m1* **have** *p*: $p > 1$ **.**
obtain *S T* **where** *ext*: *euclid-ext-poly-dynamic p D H* $= (S,T)$ **by** *force*
obtain *D1 H1* **where** *main*: *linear-hensel-main C p S T D H n* $= (D1,H1)$ **by** *force*
from *hensel-result*[*unfolded linear-hensel-binary-def ext split Let-def main*]
have *id*: $D1 = D'$ $H1 = H'$ **by** *auto*
note *eucl* $=$ *euclid-ext-poly-dynamic* [*OF cop normalized-input ext*]
from *linear-hensel-main* [*OF eucl*(*1*)
eq monic-input normalized-input main [*unfolded id*] *n prime cop*]
show *?thesis* **by** (*intro ex1I*, *auto*)
qed

context
fixes *C* :: *int poly*
begin

lemma *hensel-step-main*: **assumes**
one-q: *poly-mod.eq-m q* $(D * S + H * T)$ *1*
and *one-p*: *poly-mod.eq-m p* $(D1 * S1 + H1 * T1)$ *1*
and *CDHq*: *poly-mod.eq-m q C* $(D * H)$
and *D1D*: *poly-mod.eq-m p D1 D*
and *H1H*: *poly-mod.eq-m p H1 H*
and *S1S*: *poly-mod.eq-m p S1 S*
and *T1T*: *poly-mod.eq-m p T1 T*
and *mon*: *monic D*
and *mon1*: *monic D1*
and *q*: $q > 1$
and *p*: $p > 1$
and *D1*: *poly-mod.Mp p D1* $=$ *D1*
and *H1*: *poly-mod.Mp p H1* $=$ *H1*
and *S1*: *poly-mod.Mp p S1* $=$ *S1*
and *T1*: *poly-mod.Mp p T1* $=$ *T1*
and *D*: *poly-mod.Mp q D* $=$ *D*
and *H*: *poly-mod.Mp q H* $=$ *H*

and *S*: *poly-mod.Mp q S = S*
and *T*: *poly-mod.Mp q T = T*
and *U1*: *U1 = poly-mod.Mp p (sdiv-poly (C − D ∗ H) q)*
and *dupe1*: *dupe-monic-dynamic p D1 H1 S1 T1 U1 = (A,B)*
and *D′*: *D′ = D + smult q B*
and *H′*: *H′ = H + smult q A*
and *U2*: *U2 = poly-mod.Mp q (sdiv-poly (S∗D′ + T∗H′ − 1) p)*
and *dupe2*: *dupe-monic-dynamic q D H S T U2 = (A′,B′)*
and *rq*: *r = p ∗ q*
and *pq*: *p dvd q*
and *S′*: *S′ = poly-mod.Mp r (S − smult p A′)*
and *T′*: *T′ = poly-mod.Mp r (T − smult p B′)*
shows *poly-mod.eq-m r C (D′ ∗ H′)*
poly-mod.Mp r D′ = D′
poly-mod.Mp r H′ = H′
poly-mod.Mp r S′ = S′
poly-mod.Mp r T′ = T′
poly-mod.eq-m r (D′ ∗ S′ + H′ ∗ T′) 1
monic D′
unfolding *rq*
proof −
from *pq* **obtain** *k* **where** *qp*: *q = p ∗ k* **unfolding** *dvd-def* **by** *auto*
from *arg-cong[OF qp, of sgn] q p* **have** *k0*: *k > 0* **unfolding** *sgn-mult* **by** *(auto simp: sgn-1-pos)*
from *qp* **have** *qq*: *q ∗ q = p ∗ q ∗ k* **by** *auto*
let *?r = p ∗ q*
interpret *poly-mod-2 p* **by** *(standard, insert p, auto)*
interpret *q*: *poly-mod-2 q* **by** *(standard, insert q, auto)*
from *p q* **have** *r*: *?r > 1* **by** *(simp add: less-1-mult)*
interpret *r*: *poly-mod-2 ?r* **using** *r* **unfolding** *poly-mod-2-def* **.**
have *Mp-conv*: *Mp (q.Mp x) = Mp x* **for** *x* **unfolding** *qp*
by *(rule Mp-product-modulus[OF refl k0])*
from *arg-cong[OF CDHq, of Mp, unfolded Mp-conv]* **have** *Mp C = Mp (Mp D ∗ Mp H)*
by *simp*
also have *Mp D = Mp D1* **using** *D1D* **by** *simp*
also have *Mp H = Mp H1* **using** *H1H* **by** *simp*
finally have *CDHp*: *eq-m C (D1 ∗ H1)* **by** *simp*
have *Mp U1 = U1* **unfolding** *U1* **by** *simp*
note *dupe1 = dupe-monic-dynamic[OF dupe1 one-p mon1 D1 H1 S1 T1 this]*
have *q.Mp U2 = U2* **unfolding** *U2* **by** *simp*
note *dupe2 = q.dupe-monic-dynamic[OF dupe2 one-q mon D H S T this]*
from *CDHq* **have** *q.Mp C − q.Mp (D ∗ H) = 0* **by** *simp*
hence *q.Mp (q.Mp C − q.Mp (D ∗ H)) = 0* **by** *simp*
hence *q.Mp (C − D∗H) = 0* **by** *simp*
from *q.Mp-0-smult-sdiv-poly[OF this]* **have** *CDHq*: *smult q (sdiv-poly (C − D ∗ H) q) = C − D ∗ H* **.**
{
fix *A B*

have *Mp* (*A* ∗ *D1* + *B* ∗ *H1*) = *Mp* (*Mp* (*A* ∗ *D1*) + *Mp* (*B* ∗ *H1*)) **by** *simp*
also have *Mp* (*A* ∗ *D1*) = *Mp* (*A* ∗ *Mp D1*) **by** *simp*
also have . . . = *Mp* (*A* ∗ *D*) **unfolding** *D1D* **by** *simp*
also have *Mp* (*B* ∗ *H1*) = *Mp* (*B* ∗ *Mp H1*) **by** *simp*
also have . . . = *Mp* (*B* ∗ *H*) **unfolding** *H1H* **by** *simp*
finally have *Mp* (*A* ∗ *D1* + *B* ∗ *H1*) = *Mp* (*A* ∗ *D* + *B* ∗ *H*) **by** *simp*
} **note** *D1H1* = *this*
have *r.Mp* (*D′* ∗ *H′*) = *r.Mp* ((*D* + *smult q B*) ∗ (*H* + *smult q A*))
unfolding *D′ H′* **by** *simp*
also have (*D* + *smult q B*) ∗ (*H* + *smult q A*) = (*D* ∗ *H* + *smult q* (*A* ∗ *D* + *B* ∗ *H*)) + *smult* (*q* ∗ *q*) (*A* ∗ *B*)
by (*simp add*: *field-simps smult-distribs*)
also have *r.Mp* . . . = *r.Mp* (*D* ∗ *H* + *r.Mp* (*smult q* (*A* ∗ *D* + *B* ∗ *H*)) + *r.Mp* (*smult* (*q* ∗ *q*) (*A* ∗ *B*)))
using *r.plus-Mp* **by** *metis*
also have *r.Mp* (*smult* (*q* ∗ *q*) (*A* ∗ *B*)) = *0* **unfolding** *qq*
by (*metis r.Mp-smult-m-0 smult-smult*)
also have *r.Mp* (*smult q* (*A* ∗ *D* + *B* ∗ *H*)) = *r.Mp* (*smult q U1*)
proof (*rule Mp-lift-modulus*[*of - - q*])
show *Mp* (*A* ∗ *D* + *B* ∗ *H*) = *Mp U1* **using** *dupe1*(*1*) **unfolding** *D1H1* **by** *simp*
qed
also have . . . = *r.Mp* (*C* − *D* ∗ *H*)
unfolding *arg-cong*[*OF CDHq*, *of r.Mp*, *symmetric*]
using *Mp-lift-modulus*[*of U1 sdiv-poly* (*C* − *D* ∗ *H*) *q q*] **unfolding** *U1*
by *simp*
also have *r.Mp* (*D* ∗ *H* + *r.Mp* (*C* − *D* ∗ *H*) + *0*) = *r.Mp C* **by** *simp*
finally show *CDH*: *r.eq-m C* (*D′* ∗ *H′*) **by** *simp*
have *degree D1* = *degree* (*Mp D1*) **using** *mon1* **by** *simp*
also have . . . = *degree D* **unfolding** *D1D* **using** *mon* **by** *simp*
finally have *deg-eq*: *degree D1* = *degree D* **by** *simp*
show *mon*: *monic D′* **unfolding** *D′* **using** *dupe1*(*2*) *mon* **unfolding** *deg-eq* **by** (*rule monic-smult-add-small*)
have *Mp* (*S* ∗ *D′* + *T* ∗ *H′* − *1*) = *Mp* (*Mp* (*D* ∗ *S* + *H* ∗ *T*) + (*smult q* (*S* ∗ *B* + *T* ∗ *A*) − *1*))
unfolding *D′ H′ plus-Mp* **by** (*simp add*: *field-simps smult-distribs*)
also have *Mp* (*D* ∗ *S* + *H* ∗ *T*) = *Mp* (*Mp* (*D1* ∗ *Mp S*) + *Mp* (*H1* ∗ *Mp T*)) **using** *D1H1*[*of S T*] **by** (*simp add*: *ac-simps*)
also have . . . = *1* **using** *one-p* **unfolding** *S1S*[*symmetric*] *T1T*[*symmetric*] **by** *simp*
also have *Mp* (*1* + (*smult q* (*S* ∗ *B* + *T* ∗ *A*) − *1*)) = *Mp* (*smult q* (*S* ∗ *B* + *T* ∗ *A*)) **by** *simp*
also have . . . = *0* **unfolding** *qp* **by** (*metis Mp-smult-m-0 smult-smult*)
finally have *Mp* (*S* ∗ *D′* + *T* ∗ *H′* − *1*) = *0* **.**
from *Mp-0-smult-sdiv-poly*[*OF this*]
have *SDTH*: *smult p* (*sdiv-poly* (*S* ∗ *D′* + *T* ∗ *H′* − *1*) *p*) = *S* ∗ *D′* + *T* ∗ *H′* − *1* **.**
have *swap*: *q* ∗ *p* = *p* ∗ *q* **by** *simp*
have *r.Mp* (*D′* ∗ *S′* + *H′* ∗ *T′*) =

r.Mp $((D + \mathit{smult}\ q\ B) * (S - \mathit{smult}\ p\ A') + (H + \mathit{smult}\ q\ A) * (T - \mathit{smult}\ p\ B'))$
unfolding $D'\ S'\ H'\ T'$ *rq* **using** *r.plus-Mp r.mult-Mp* **by** *metis*
also have ... $=$ *r.Mp* $((D * S + H * T +$
smult $q\ (B * S + A * T)) -$ *smult* $p\ (A' * D + B' * H) -$ *smult* $?r\ (A * B' + B * A'))$
by (*simp add*: *field-simps smult-distribs*)
also have ... $=$ *r.Mp* $((D * S + H * T +$
smult $q\ (B * S + A * T)) -$ *r.Mp* (*smult* $p\ (A' * D + B' * H)) -$ *r.Mp* (*smult* $?r\ (A * B' + B * A')))$
using *r.plus-Mp r.minus-Mp* **by** *metis*
also have *r.Mp* (*smult* $?r\ (A * B' + B * A')) = 0$ **by** *simp*
also have *r.Mp* (*smult* $p\ (A' * D + B' * H)) =$ *r.Mp* (*smult* p *U2*)
using *q.Mp-lift-modulus*[*OF dupe2*(*1*), *of p*] **unfolding** *swap* **.**
also have ... $=$ *r.Mp* $(S * D' + T * H' - 1)$
unfolding *arg-cong*[*OF SDTH*, *of r.Mp*, *symmetric*]
using *q.Mp-lift-modulus*[*of U2 sdiv-poly* $(S * D' + T * H' - 1)$ *p p*]
unfolding *U2 swap* **by** *simp*
also have $S * D' + T * H' - 1 = S * D + T * H +$ *smult* $q\ (B * S + A * T) - 1$
unfolding $D'\ H'$ **by** (*simp add*: *field-simps smult-distribs*)
also have *r.Mp* $(D * S + H * T +$ *smult* $q\ (B * S + A * T) -$
r.Mp $(S * D + T * H +$ *smult* $q\ (B * S + A * T) - 1) - 0)$
$= 1$ **by** *simp*
finally show *1*: *r.eq-m* $(D' * S' + H' * T')$ *1* **by** *simp*
show D': *r.Mp* $D' = D'$ **unfolding** D' *r.Mp-ident-iff poly-mod.Mp-coeff plus-poly.rep-eq coeff-smult*
proof
fix *n*
from *D dupe1*(*4*) **have** *coeff D n* $\in \{0..<q\}$ *coeff B n* $\in \{0..<p\}$
unfolding *q.Mp-ident-iff Mp-ident-iff* **by** *auto*
thus *coeff D n* $+ q *$ *coeff B n* $\in \{0..<?r\}$ **by** (*metis range-sum-prod*)
qed
show H': *r.Mp* $H' = H'$ **unfolding** H' *r.Mp-ident-iff poly-mod.Mp-coeff plus-poly.rep-eq coeff-smult*
proof
fix *n*
from *H dupe1*(*3*) **have** *coeff H n* $\in \{0..<q\}$ *coeff A n* $\in \{0..<p\}$
unfolding *q.Mp-ident-iff Mp-ident-iff* **by** *auto*
thus *coeff H n* $+ q *$ *coeff A n* $\in \{0..<?r\}$ **by** (*metis range-sum-prod*)
qed
show *poly-mod.Mp* $?r\ S' = S'$ *poly-mod.Mp* $?r\ T' = T'$
unfolding $S'\ T'$ *rq* **by** *auto*
qed

definition *hensel-step* **where**
hensel-step p q S1 T1 D1 H1 S T D H = (
let U = *poly-mod.Mp p* (*sdiv-poly* $(C - D * H)$ *q*); — *Z2 and Z3*
(*A*,*B*) = *dupe-monic-dynamic p D1 H1 S1 T1 U*;

$D' = D + smult\ q\ B$; — $Z4$
$H' = H + smult\ q\ A$;
$U' = poly\text{-}mod.Mp\ q\ (sdiv\text{-}poly\ (S{*}D' + T{*}H' - 1)\ p)$; — $Z5 + Z6$
$(A',B') = dupe\text{-}monic\text{-}dynamic\ q\ D\ H\ S\ T\ U'$;
$q' = p * q$;
$S' = poly\text{-}mod.Mp\ q'\ (S - smult\ p\ A')$; — $Z7$
$T' = poly\text{-}mod.Mp\ q'\ (T - smult\ p\ B')$
in (S',T',D',H'))

definition *quadratic-hensel-step q S T D H = hensel-step q q S T D H S T D H*

lemma *quadratic-hensel-step-code*[*code*]:
quadratic-hensel-step q S T D H =
(*let dupe = dupe-monic-dynamic q D H S T*; — this will share the conversions of *D H S T*
$U = poly\text{-}mod.Mp\ q\ (sdiv\text{-}poly\ (C - D * H)\ q)$;
$(A, B) = dupe\ U$;
$D' = D + Polynomial.smult\ q\ B$;
$H' = H + Polynomial.smult\ q\ A$;
$U' = poly\text{-}mod.Mp\ q\ (sdiv\text{-}poly\ (S * D' + T * H' - 1)\ q)$;
$(A', B') = dupe\ U'$;
$q' = q * q$;
$S' = poly\text{-}mod.Mp\ q'\ (S - Polynomial.smult\ q\ A')$;
$T' = poly\text{-}mod.Mp\ q'\ (T - Polynomial.smult\ q\ B')$
in (S', T', D', H'))
unfolding *quadratic-hensel-step-def*[*unfolded hensel-step-def*] *Let-def* **..**

definition *simple-quadratic-hensel-step* **where** — do not compute new values S' and T'
simple-quadratic-hensel-step q S T D H = (
let $U = poly\text{-}mod.Mp\ q\ (sdiv\text{-}poly\ (C - D * H)\ q)$; — $Z2 + Z3$
$(A,B) = dupe\text{-}monic\text{-}dynamic\ q\ D\ H\ S\ T\ U$;
$D' = D + smult\ q\ B$; — $Z4$
$H' = H + smult\ q\ A$
in (D',H'))

lemma *hensel-step*: **assumes** *step*: *hensel-step p q S1 T1 D1 H1 S T D H* $= (S', T', D', H')$
and *one-p*: *poly-mod.eq-m p* $(D1 * S1 + H1 * T1)$ *1*
and *mon1*: *monic D1*
and *p*: $p > 1$
and *CDHq*: *poly-mod.eq-m q C* $(D * H)$
and *one-q*: *poly-mod.eq-m q* $(D * S + H * T)$ *1*
and *D1D*: *poly-mod.eq-m p D1 D*
and *H1H*: *poly-mod.eq-m p H1 H*
and *S1S*: *poly-mod.eq-m p S1 S*
and *T1T*: *poly-mod.eq-m p T1 T*
and *mon*: *monic D*
and *q*: $q > 1$

and $D1$: *poly-mod.Mp p D1* = *D1*
and $H1$: *poly-mod.Mp p H1* = *H1*
and $S1$: *poly-mod.Mp p S1* = *S1*
and $T1$: *poly-mod.Mp p T1* = *T1*
and D: *poly-mod.Mp q D* = *D*
and H: *poly-mod.Mp q H* = *H*
and S: *poly-mod.Mp q S* = *S*
and T: *poly-mod.Mp q T* = *T*
and *rq*: $r = p * q$
and *pq*: *p dvd q*
shows
poly-mod.eq-m r C $(D' * H')$
poly-mod.eq-m r $(D' * S' + H' * T')$ *1*
poly-mod.Mp r $D' = D'$
poly-mod.Mp r $H' = H'$
poly-mod.Mp r $S' = S'$
poly-mod.Mp r $T' = T'$
poly-mod.Mp p D1 = *poly-mod.Mp p* D'
poly-mod.Mp p H1 = *poly-mod.Mp p* H'
poly-mod.Mp p S1 = *poly-mod.Mp p* S'
poly-mod.Mp p T1 = *poly-mod.Mp p* T'
monic D'
proof −
define *U* **where** *U*: *U* = *poly-mod.Mp p* (*sdiv-poly* $(C - D * H)$ *q*)
note *step* = *step*[*unfolded hensel-step-def Let-def*, *folded U*]
obtain *A B* **where** *dupe1*: *dupe-monic-dynamic p D1 H1 S1 T1 U* = (*A*,*B*) **by** *force*
note *step* = *step*[*unfolded dupe1 split*]
from *step* **have** D': $D' = D +$ *smult q B* **and** H': $H' = H +$ *smult q A*
by (*auto split*: *prod.splits*)
define U' **where** U': U' = *poly-mod.Mp q* (*sdiv-poly* $(S * D' + T * H' - 1)$ *p*)
obtain A' B' **where** *dupe2*: *dupe-monic-dynamic q D H S T* U' = (A',B') **by** *force*
from *step*[*folded* D' H', *folded* U', *unfolded dupe2 split*, *folded rq*]
have S': S' = *poly-mod.Mp r* (*S* − *Polynomial.smult p* A') **and**
T': T' = *poly-mod.Mp r* (*T* − *Polynomial.smult p* B') **by** *auto*
from *hensel-step-main*[*OF one-q one-p CDHq D1D H1H S1S T1T mon mon1 q p D1 H1 S1 T1 D H S T U*
dupe1 D' H' U' *dupe2 rq pq* S' T']
show *poly-mod.eq-m r* $(D' * S' + H' * T')$ *1*
poly-mod.eq-m r C $(D' * H')$
poly-mod.Mp r $D' = D'$
poly-mod.Mp r $H' = H'$
poly-mod.Mp r $S' = S'$
poly-mod.Mp r $T' = T'$
monic D' **by** *auto*
from *pq* **obtain** *s* **where** *q*: $q = p * s$ **by** (*metis dvdE*)
show *poly-mod.Mp p D1* = *poly-mod.Mp p* D'

poly-mod.Mp p H1 = *poly-mod.Mp p H′*
unfolding *q D′ D1D H′ H1H*
by (*metis add.right-neutral poly-mod.Mp-smult-m-0 poly-mod.plus-Mp*(*2*) *smult-smult*)+

from ‹*q* > *1*› **have** *q0*: *q* > *0* **by** *auto*
show *poly-mod.Mp p S1* = *poly-mod.Mp p S′*
poly-mod.Mp p T1 = *poly-mod.Mp p T′*
unfolding *S′ S1S T′ T1T poly-mod-2.Mp-product-modulus*[*OF poly-mod-2.intro*[*OF* ‹*p* > *1*›] *rq q0*]
by (*metis group-add-class.diff-0-right poly-mod.Mp-smult-m-0 poly-mod.minus-Mp*(*2*))+

qed

lemma *quadratic-hensel-step*: **assumes** *step*: *quadratic-hensel-step q S T D H* = (*S′*, *T′*, *D′*, *H′*)
and *CDH*: *poly-mod.eq-m q C* (*D* ∗ *H*)
and *one*: *poly-mod.eq-m q* (*D* ∗ *S* + *H* ∗ *T*) *1*
and *D*: *poly-mod.Mp q D* = *D*
and *H*: *poly-mod.Mp q H* = *H*
and *S*: *poly-mod.Mp q S* = *S*
and *T*: *poly-mod.Mp q T* = *T*
and *mon*: *monic D*
and *q*: *q* > *1*
and *rq*: *r* = *q* ∗ *q*
shows
poly-mod.eq-m r C (*D′* ∗ *H′*)
poly-mod.eq-m r (*D′* ∗ *S′* + *H′* ∗ *T′*) *1*
poly-mod.Mp r D′ = *D′*
poly-mod.Mp r H′ = *H′*
poly-mod.Mp r S′ = *S′*
poly-mod.Mp r T′ = *T′*
poly-mod.Mp q D = *poly-mod.Mp q D′*
poly-mod.Mp q H = *poly-mod.Mp q H′*
poly-mod.Mp q S = *poly-mod.Mp q S′*
poly-mod.Mp q T = *poly-mod.Mp q T′*
monic D′
proof (*atomize*(*full*), *goal-cases*)
case *1*
from *hensel-step*[*OF step*[*unfolded quadratic-hensel-step-def*] *one mon q CDH one refl refl refl refl mon q D H S T D H S T rq*]
show *?case* **by** *auto*
qed

context
fixes *p* :: *int* **and** *S1 T1 D1 H1* :: *int poly*
begin
private lemma *decrease*[*termination-simp*]: ¬ *j* ≤ *1* ⟹ *odd j* ⟹ *Suc* (*j div 2*) < *j* **by** *presburger*

fun *quadratic-hensel-loop* **where**
quadratic-hensel-loop $(j :: nat) = ($
if $j \leq 1$ *then* $(p, S1, T1, D1, H1)$ *else*
if even j *then*
$($*case quadratic-hensel-loop* $(j$ *div* $2)$ *of*
$(q, S, T, D, H) \Rightarrow$
let $qq = q * q$ *in*
$($*case quadratic-hensel-step* $q\ S\ T\ D\ H$ *of* — quadratic step
$(S', T', D', H') \Rightarrow (qq, S', T', D', H')))$
else — odd j
$($*case quadratic-hensel-loop* $(j$ *div* $2 + 1)$ *of*
$(q, S, T, D, H) \Rightarrow$
$($*case quadratic-hensel-step* $q\ S\ T\ D\ H$ *of* — quadratic step
$(S', T', D', H') \Rightarrow$
let $qq = q * q$; $pj = qq$ *div* p; *down* = *poly-mod.Mp* pj *in*
$(pj$, *down* S', *down* T', *down* D', *down* $H'))))$

definition *quadratic-hensel-main* $j = ($*case quadratic-hensel-loop* j *of*
$(qq, S, T, D, H) \Rightarrow (D, H))$

declare *quadratic-hensel-loop.simps*[*simp del*]

— unroll the definition of *hensel-loop* so that in outermost iteration we can use *simple-hensel-step*
lemma *quadratic-hensel-main-code*[*code*]: *quadratic-hensel-main* $j = ($
if $j \leq 1$ *then* $(D1, H1)$
else if even j
then $($*case quadratic-hensel-loop* $(j$ *div* $2)$ *of*
$(q, S, T, D, H) \Rightarrow$
simple-quadratic-hensel-step $q\ S\ T\ D\ H)$
else $($*case quadratic-hensel-loop* $(j$ *div* $2 + 1)$ *of*
$(q, S, T, D, H) \Rightarrow$
$($*case simple-quadratic-hensel-step* $q\ S\ T\ D\ H$ *of*
$(D', H') \Rightarrow$ *let down* = *poly-mod.Mp* $(q * q$ *div* $p)$ *in* $($*down* D', *down* $H'))))$
unfolding *quadratic-hensel-loop.simps*[*of j*] *quadratic-hensel-main-def Let-def*
by (*simp split*: *if-splits prod.splits option.splits sum.splits*
add: *quadratic-hensel-step-code simple-quadratic-hensel-step-def Let-def*)

context
fixes $j :: nat$
assumes *1*: *poly-mod.eq-m* $p\ (D1 * S1 + H1 * T1)\ 1$
and *CDH1*: *poly-mod.eq-m* $p\ C\ (D1 * H1)$
and *mon1*: *monic* $D1$
and p: $p > 1$
and $D1$: *poly-mod.Mp* $p\ D1 = D1$
and $H1$: *poly-mod.Mp* $p\ H1 = H1$
and $S1$: *poly-mod.Mp* $p\ S1 = S1$

```
  and T1: poly-mod.Mp p T1 = T1
  and j: j ≥ 1
begin

lemma quadratic-hensel-loop:
  assumes quadratic-hensel-loop j = (q, S, T, D, H)
  shows (poly-mod.eq-m q C (D ∗ H) ∧ monic D
    ∧ poly-mod.eq-m p D1 D ∧ poly-mod.eq-m p H1 H
    ∧ poly-mod.eq-m q (D ∗ S + H ∗ T) 1
    ∧ poly-mod.Mp q D = D ∧ poly-mod.Mp q H = H
    ∧ poly-mod.Mp q S = S ∧ poly-mod.Mp q T = T
    ∧ q = p^j)
  using j assms
proof (induct j arbitrary: q S T D H rule: less-induct)
  case (less j q' S' T' D' H')
  note res = less(3)
  interpret poly-mod-2 p using p by (rule poly-mod-2.intro)
  let ?hens = quadratic-hensel-loop
  note simp[simp] = quadratic-hensel-loop.simps[of j]
  show ?case
  proof (cases j = 1)
    case True
    show ?thesis using res simp unfolding True using CDH1 1 mon1 D1 H1 S1
T1 by auto
  next
    case False
    with less(2) have False: (j ≤ 1) = False by auto
     have mod-2: k ≥ 1 ⟹ poly-mod-2 (p^k) for k by (intro poly-mod-2.intro,
insert p, auto)
    {
      fix k D
      assume ∗: k ≥ 1 k ≤ j poly-mod.Mp (p ^ k) D = D
      from ∗(2) have {0..<p ^ k} ⊆ {0..<p ^ j} using p by auto
      hence poly-mod.Mp (p ^ j) D = D
        unfolding poly-mod-2.Mp-ident-iff[OF mod-2[OF less(2)]]
        using ∗(3)[unfolded poly-mod-2.Mp-ident-iff[OF mod-2[OF ∗(1)]]] by blast
    } note lift-norm = this
    show ?thesis
    proof (cases even j)
      case True
      let ?j2 = j div 2
      from False have lt: ?j2 < j 1 ≤ ?j2 by auto
      obtain q S T D H where rec: ?hens ?j2 = (q, S, T, D, H) by (cases ?hens
?j2, auto)
      note IH = less(1)[OF lt rec]
      from IH
      have ∗: poly-mod.eq-m q C (D ∗ H)
        poly-mod.eq-m q (D ∗ S + H ∗ T) 1
        monic D
```

eq-m D1 D
eq-m H1 H
poly-mod.Mp q D = D
poly-mod.Mp q H = H
poly-mod.Mp q S = S
poly-mod.Mp q T = T
q = p ^ ?j2
by auto
hence norm: poly-mod.Mp (p ^ j) D = D poly-mod.Mp (p ^ j) H = H
poly-mod.Mp (p ^ j) S = S poly-mod.Mp (p ^ j) T = T
using lift-norm[OF lt(2)] **by** auto
from lt p **have** q: q > 1 **unfolding** * **by** simp
let ?step = quadratic-hensel-step q S T D H
obtain S2 T2 D2 H2 **where** step-res: ?step = (S2, T2, D2, H2) **by** (cases ?step, auto)
note step = quadratic-hensel-step[OF step-res *(1,2,6−9,3) q refl]
let ?qq = q * q
{
fix D D2
assume poly-mod.Mp q D = poly-mod.Mp q D2
from arg-cong[OF this, of Mp] Mp-Mp-pow-is-Mp[of ?j2, OF - p, folded *(10)] lt
have Mp D = Mp D2 **by** simp
} **note** shrink = this
have **: poly-mod.eq-m ?qq C (D2 * H2)
poly-mod.eq-m ?qq (D2 * S2 + H2 * T2) 1
monic D2
eq-m D1 D2
eq-m H1 H2
poly-mod.Mp ?qq D2 = D2
poly-mod.Mp ?qq H2 = H2
poly-mod.Mp ?qq S2 = S2
poly-mod.Mp ?qq T2 = T2
using step shrink[of H H2] shrink[of D D2] *(4−7) **by** auto
note simp = simp False if-False rec split Let-def step-res option.simps
from True **have** j: p ^ j = p ^ (2 * ?j2) **by** auto
with *(10) **have** qq: q * q = p ^ j
by (simp add: power-mult-distrib semiring-normalization-rules(30−))
from res[unfolded simp] True **have** id′: q′ = ?qq S′ = S2 T′ = T2 D′ = D2 H′ = H2 **by** auto
show ?thesis **unfolding** id′ **using** ** **by** (auto simp: qq)
next
case odd: False
hence False′: (even j) = False **by** auto
let ?j2 = j div 2 + 1
from False odd **have** lt: ?j2 < j 1 ≤ ?j2 **by** presburger+
obtain q S T D H **where** rec: ?hens ?j2 = (q, S, T, D, H) **by** (cases ?hens ?j2, auto)
note IH = less(1)[OF lt rec]

note *simp* = *simp False if-False rec sum.simps split Let-def False′ option.simps*
from *IH* **have** ∗: *poly-mod.eq-m q C* (*D* ∗ *H*)
poly-mod.eq-m q (*D* ∗ *S* + *H* ∗ *T*) *1*
monic D
eq-m D1 D
eq-m H1 H
poly-mod.Mp q D = *D*
poly-mod.Mp q H = *H*
poly-mod.Mp q S = *S*
poly-mod.Mp q T = *T*
q = *p* ^ *?j2*
by *auto*
hence *norm*: *poly-mod.Mp* (*p* ^ *j*) *D* = *D poly-mod.Mp* (*p* ^ *j*) *H* = *H*
using *lift-norm*[*OF lt*(*2*)] *lt* **by** *auto*
from *lt p* **have** *q*: *q* > *1* **unfolding** ∗
using *mod-2 poly-mod-2.m1* **by** *blast*
let *?step* = *quadratic-hensel-step q S T D H*
obtain *S2 T2 D2 H2* **where** *step-res*: *?step* = (*S2*, *T2*, *D2*, *H2*) **by** (*cases ?step*, *auto*)
have *dvd*: *q dvd q* **by** *auto*
note *step* = *quadratic-hensel-step*[*OF step-res* ∗(*1*,*2*,*6*−*9*,*3*) *q refl*]
let *?qq* = *q* ∗ *q*
{
fix *D D2*
assume *poly-mod.Mp q D* = *poly-mod.Mp q D2*
from *arg-cong*[*OF this*, *of Mp*] *Mp-Mp-pow-is-Mp*[*of ?j2*, *OF - p*, *folded* ∗(*10*)] *lt*
have *Mp D* = *Mp D2* **by** *simp*
} **note** *shrink* = *this*
have ∗∗: *poly-mod.eq-m ?qq C* (*D2* ∗ *H2*)
poly-mod.eq-m ?qq (*D2* ∗ *S2* + *H2* ∗ *T2*) *1*
monic D2
eq-m D1 D2
eq-m H1 H2
poly-mod.Mp ?qq D2 = *D2*
poly-mod.Mp ?qq H2 = *H2*
poly-mod.Mp ?qq S2 = *S2*
poly-mod.Mp ?qq T2 = *T2*
using *step shrink*[*of H H2*] *shrink*[*of D D2*] ∗(*4*−*7*) **by** *auto*
note *simp* = *simp False if-False rec split Let-def step-res option.simps*
from *odd* **have** *j*: *Suc j* = *2* ∗ *?j2* **by** *auto*
from *arg-cong*[*OF this*, *of* λ *j*. *p* ^ *j div p*]
have *pj*: *p* ^ *j* = *q* ∗ *q div p* **and** *qq*: *q* ∗ *q* = *p* ^ *j* ∗ *p* **unfolding** ∗(*10*) **using** *p*
by (*simp add*: *power-mult-distrib semiring-normalization-rules*(*30*−))+
let *?pj* = *p* ^ *j*
from *res*[*unfolded simp*] *pj*
have *id*:
$q' = p\hat{}j$

```
        S′ = poly-mod.Mp ?pj S2
        T′ = poly-mod.Mp ?pj T2
        D′ = poly-mod.Mp ?pj D2
        H′ = poly-mod.Mp ?pj H2
        by auto
      interpret pj: poly-mod-2 ?pj by (rule mod-2[OF ‹1 ≤ j›])
      have norm: pj.Mp D′ = D′ pj.Mp H′ = H′
        unfolding id by (auto simp: poly-mod.Mp-Mp)
      have mon: monic D′ using pj.monic-Mp[OF step(11)] unfolding id .
      have id′: Mp (pj.Mp D) = Mp D for D using ‹1 ≤ j›
        by (simp add: Mp-Mp-pow-is-Mp p)
      have eq: eq-m D1 D2 ⟹ eq-m D1 (pj.Mp D2) for D1 D2
        unfolding id′ by auto
      have id′′: pj.Mp (poly-mod.Mp (q ∗ q) D) = pj.Mp D for D
        unfolding qq by (rule pj.Mp-product-modulus[OF refl], insert p, auto)
      {
        fix D1 D2
        assume poly-mod.eq-m (q ∗ q) D1 D2
        hence poly-mod.Mp (q ∗ q) D1 = poly-mod.Mp (q ∗ q) D2 by simp
        from arg-cong[OF this, of pj.Mp]
        have pj.Mp D1 = pj.Mp D2 unfolding id′′ .
      } note eq′ = this
      from eq′[OF step(1)] have eq1: pj.eq-m C (D′ ∗ H′) unfolding id by simp
      from eq′[OF step(2)] have eq2: pj.eq-m (D′ ∗ S′ + H′ ∗ T′) 1
        unfolding id by (metis pj.mult-Mp pj.plus-Mp)
      from ∗∗(4−5) have eq3: eq-m D1 D′ eq-m H1 H′
        unfolding id by (auto intro: eq)
      from norm mon eq1 eq2 eq3
      show ?thesis unfolding id by simp
    qed
  qed
qed

lemma quadratic-hensel-main: assumes res: quadratic-hensel-main j = (D,H)
  shows poly-mod.eq-m (p^j) C (D ∗ H)
  monic D
  poly-mod.eq-m p D1 D
  poly-mod.eq-m p H1 H
  poly-mod.Mp (p^j) D = D
  poly-mod.Mp (p^j) H = H
proof (atomize(full), goal-cases)
  case 1
  let ?hen = quadratic-hensel-loop j
  from res obtain q S T where hen: ?hen = (q, S, T, D, H)
    by (cases ?hen, auto simp: quadratic-hensel-main-def)
  from quadratic-hensel-loop[OF hen] show ?case by auto
qed
end
end
```

end

datatype $'a$ *factor-tree* = *Factor-Leaf* $'a$ *int poly* | *Factor-Node* $'a$ $'a$ *factor-tree* $'a$ *factor-tree*

fun *factor-node-info* :: $'a$ *factor-tree* ⇒ $'a$ **where**
factor-node-info (*Factor-Leaf i x*) = *i*
| *factor-node-info* (*Factor-Node i l r*) = *i*

fun *factors-of-factor-tree* :: $'a$ *factor-tree* ⇒ *int poly multiset* **where**
factors-of-factor-tree (*Factor-Leaf i x*) = {#*x*#}
| *factors-of-factor-tree* (*Factor-Node i l r*) = *factors-of-factor-tree l* + *factors-of-factor-tree r*

fun *product-factor-tree* :: *int* ⇒ $'a$ *factor-tree* ⇒ *int poly factor-tree* **where**
product-factor-tree p (*Factor-Leaf i x*) = (*Factor-Leaf x x*)
| *product-factor-tree p* (*Factor-Node i l r*) = (*let*
L = *product-factor-tree p l*;
R = *product-factor-tree p r*;
f = *factor-node-info L*;
g = *factor-node-info R*;
fg = *poly-mod.Mp p* (*f* ∗ *g*)
in Factor-Node fg L R)

fun *sub-trees* :: $'a$ *factor-tree* ⇒ $'a$ *factor-tree set* **where**
sub-trees (*Factor-Leaf i x*) = {*Factor-Leaf i x*}
| *sub-trees* (*Factor-Node i l r*) = *insert* (*Factor-Node i l r*) (*sub-trees l* ∪ *sub-trees r*)

lemma *sub-trees-refl*[*simp*]: *t* ∈ *sub-trees t* **by** (*cases t*, *auto*)

lemma *product-factor-tree*: **assumes** ⋀ *x*. *x* ∈# *factors-of-factor-tree t* ⟹ *poly-mod.Mp p x* = *x*
shows *u* ∈ *sub-trees* (*product-factor-tree p t*) ⟹ *factor-node-info u* = *f* ⟹ *poly-mod.Mp p f* = *f* ∧ *f* = *poly-mod.Mp p* (*prod-mset* (*factors-of-factor-tree u*)) ∧
factors-of-factor-tree (*product-factor-tree p t*) = *factors-of-factor-tree t*
using *assms*
proof (*induct t arbitrary*: *u f*)
case (*Factor-Node i l r u f*)
interpret *poly-mod p* **.**
let *?L* = *product-factor-tree p l*
let *?R* = *product-factor-tree p r*
let *?f* = *factor-node-info ?L*
let *?g* = *factor-node-info ?R*
let *?fg* = *Mp* (*?f* ∗ *?g*)
have *Mp ?f* = *?f* ∧ *?f* = *Mp* (*prod-mset* (*factors-of-factor-tree ?L*)) ∧
(*factors-of-factor-tree ?L*) = (*factors-of-factor-tree l*)
by (*rule Factor-Node*(*1*)[*OF sub-trees-refl refl*], *insert Factor-Node*(*5*), *auto*)

hence *IH1*: *?f* = *Mp* (*prod-mset* (*factors-of-factor-tree ?L*))
    (*factors-of-factor-tree ?L*) = (*factors-of-factor-tree l*) **by** *blast+*
have *Mp ?g* = *?g* ∧ *?g* = *Mp* (*prod-mset* (*factors-of-factor-tree ?R*)) ∧
    (*factors-of-factor-tree ?R*) = (*factors-of-factor-tree r*)
    by (*rule Factor-Node*(*2*)[*OF sub-trees-refl refl*], *insert Factor-Node*(*5*), *auto*)
hence *IH2*: *?g* = *Mp* (*prod-mset* (*factors-of-factor-tree ?R*))
    (*factors-of-factor-tree ?R*) = (*factors-of-factor-tree r*) **by** *blast+*
have *id*: (*factors-of-factor-tree* (*product-factor-tree p* (*Factor-Node i l r*))) =
  (*factors-of-factor-tree* (*Factor-Node i l r*)) **by** (*simp add*: *Let-def IH1 IH2*)
from *Factor-Node*(*3*) **consider** (*root*) *u* = *Factor-Node ?fg ?L ?R*
  | (*l*) *u* ∈ *sub-trees ?L* | (*r*) *u* ∈ *sub-trees ?R*
  by (*auto simp*: *Let-def*)
thus *?case*
proof *cases*
  case *root*
  with *Factor-Node* **have** *f*: *f* = *?fg* **by** *auto*
  show *?thesis* **unfolding** *f root id* **by** (*simp add*: *Let-def ac-simps IH1 IH2*)
next
  case *l*
  have *Mp f* = *f* ∧ *f* = *Mp* (*prod-mset* (*factors-of-factor-tree u*))
    using *Factor-Node*(*1*)[*OF l Factor-Node*(*4*)] *Factor-Node*(*5*) **by** *auto*
  thus *?thesis* **unfolding** *id* **by** *blast*
next
  case *r*
  have *Mp f* = *f* ∧ *f* = *Mp* (*prod-mset* (*factors-of-factor-tree u*))
    using *Factor-Node*(*2*)[*OF r Factor-Node*(*4*)] *Factor-Node*(*5*) **by** *auto*
  thus *?thesis* **unfolding** *id* **by** *blast*
qed
qed *auto*

fun *create-factor-tree-simple* :: *int poly list* ⇒ *unit factor-tree* **where**
  create-factor-tree-simple xs = (*let n* = *length xs in if n* ≤ *1 then Factor-Leaf* ()
(*hd xs*)
    else let i = *n div 2*;
      xs1 = *take i xs*;
      xs2 = *drop i xs*
     in Factor-Node () (*create-factor-tree-simple xs1*) (*create-factor-tree-simple xs2*)
)

declare *create-factor-tree-simple.simps*[*simp del*]

lemma *create-factor-tree-simple*: *xs* ≠ [] ⟹ *factors-of-factor-tree* (*create-factor-tree-simple xs*) = *mset xs*
proof (*induct xs rule*: *wf-induct*[*OF wf-measure*[*of length*]])
  case (*1 xs*)
  from *1*(*2*) **have** *xs*: *length xs* ≠ *0* **by** *auto*
  then consider (*base*) *length xs* = *1* | (*step*) *length xs* > *1* **by** *linarith*
  thus *?case*
  proof *cases*

case *base*
then obtain x **where** *xs*: $xs = [x]$ **by** (*cases xs*; *cases tl xs*; *auto*)
thus *?thesis* **by** (*auto simp*: *create-factor-tree-simple.simps*)
next
case *step*
let *?i* = *length xs div 2*
let *?xs1* = *take ?i xs*
let *?xs2* = *drop ?i xs*
from *step* **have** *xs1*: (*?xs1*, *xs*) ∈ *measure length ?xs1* ≠ [] **by** *auto*
from *step* **have** *xs2*: (*?xs2*, *xs*) ∈ *measure length ?xs2* ≠ [] **by** *auto*
from *step* **have** *id*: *create-factor-tree-simple xs* = *Factor-Node* () (*create-factor-tree-simple* (*take ?i xs*))
(*create-factor-tree-simple* (*drop ?i xs*)) **unfolding** *create-factor-tree-simple.simps*[*of xs*] *Let-def* **by** *auto*
have *xs*: *xs* = *?xs1* @ *?xs2* **by** *auto*
show *?thesis* **unfolding** *id arg-cong*[*OF xs*, *of mset*] *mset-append*
using *1*(*1*)[*rule-format*, *OF xs1*] *1*(*1*)[*rule-format*, *OF xs2*]
by *auto*
qed
qed

We define a better factorization tree which balances the trees according to their degree., cf. Modern Computer Algebra, Chapter 15.5 on Multifactor Hensel lifting.

fun *partition-factors-main* :: *nat* ⇒ ('*a* × *nat*) *list* ⇒ ('*a* × *nat*) *list* × ('*a* × *nat*) *list* **where**
partition-factors-main s [] = ([], [])
| *partition-factors-main s* ((*f*,*d*) # *xs*) = (*if d* ≤ *s then case partition-factors-main* (*s* − *d*) *xs of*
(*l*,*r*) ⇒ ((*f*,*d*) # *l*, *r*) *else case partition-factors-main d xs of*
(*l*,*r*) ⇒ (*l*, (*f*,*d*) # *r*))

lemma *partition-factors-main*: *partition-factors-main s xs* = (*a*,*b*) ⟹ *mset xs* = *mset a* + *mset b*
by (*induct s xs arbitrary*: *a b rule*: *partition-factors-main.induct*, *auto split*: *if-splits prod.splits*)

definition *partition-factors* :: ('*a* × *nat*) *list* ⇒ ('*a* × *nat*) *list* × ('*a* × *nat*) *list* **where**
partition-factors xs = (*let n* = *sum-list* (*map snd xs*) *div 2 in*
case partition-factors-main n xs of
([], *x* # *y* # *ys*) ⇒ ([*x*], *y* # *ys*)
| (*x* # *y* # *ys*, []) ⇒ ([*x*], *y* # *ys*)
| *pair* ⇒ *pair*)

lemma *partition-factors*: *partition-factors xs* = (*a*,*b*) ⟹ *mset xs* = *mset a* + *mset b*
unfolding *partition-factors-def Let-def*
by (*cases partition-factors-main* (*sum-list* (*map snd xs*) *div 2*) *xs*, *auto split*:

list.splits
simp: *partition-factors-main*)

lemma *partition-factors-length*: **assumes** ¬ *length xs* ≤ *1* (*a*,*b*) = *partition-factors xs*
shows [*termination-simp*]: *length a* < *length xs length b* < *length xs* **and** *a* ≠ [] *b* ≠ []
proof −
obtain *ys zs* **where** *main*: *partition-factors-main* (*sum-list* (*map snd xs*) *div 2*) *xs* = (*ys*,*zs*) **by** *force*
note *res* = *assms*(*2*)[*unfolded partition-factors-def Let-def main split*]
from *arg-cong*[*OF partition-factors-main*[*OF main*], *of size*] **have** *len*: *length xs* = *length ys* + *length zs* **by** *auto*
with *assms*(*1*) **have** *len2*: *length ys* + *length zs* ≥ *2* **by** *auto*
from *res len2* **have** *length a* < *length xs* ∧ *length b* < *length xs* ∧ *a* ≠ [] ∧ *b* ≠ [] **unfolding** *len*
by (*cases ys*; *cases zs*; *cases tl ys*; *cases tl zs*; *auto*)
thus *length a* < *length xs length b* < *length xs a* ≠ [] *b* ≠ [] **by** *blast*+
qed

fun *create-factor-tree-balanced* :: (*int poly* × *nat*)*list* ⇒ *unit factor-tree* **where**
create-factor-tree-balanced xs = (*if length xs* ≤ *1 then Factor-Leaf* () (*fst* (*hd xs*)) *else*
case partition-factors xs of (*l*,*r*) ⇒ *Factor-Node* ()
(*create-factor-tree-balanced l*)
(*create-factor-tree-balanced r*))

definition *create-factor-tree* :: *int poly list* ⇒ *unit factor-tree* **where**
create-factor-tree xs = (*let ys* = *map* (λ *f*. (*f*, *degree f*)) *xs*;
zs = *rev* (*sort-key snd ys*)
in create-factor-tree-balanced zs)

lemma *create-factor-tree-balanced*: *xs* ≠ [] ⟹ *factors-of-factor-tree* (*create-factor-tree-balanced xs*) = *mset* (*map fst xs*)
proof (*induct xs rule*: *create-factor-tree-balanced.induct*)
case (*1 xs*)
show *?case*
proof (*cases length xs* ≤ *1*)
case *True*
with *1*(*3*) **obtain** *x* **where** *xs*: *xs* = [*x*] **by** (*cases xs*; *cases tl xs*, *auto*)
show *?thesis* **unfolding** *xs* **by** *auto*
next
case *False*
obtain *a b* **where** *part*: *partition-factors xs* = (*a*,*b*) **by** *force*
note *abp* = *this*[*symmetric*]
note *nonempty* = *partition-factors-length*(*3*−*4*)[*OF False abp*]
note *IH* = *1*(*1*)[*OF False abp nonempty*(*1*)] *1*(*2*)[*OF False abp nonempty*(*2*)]
show *?thesis* **unfolding** *create-factor-tree-balanced.simps*[*of xs*] *part split* **using**

False IH partition-factors[*OF part*] **by** *auto*
qed
qed

lemma *create-factor-tree*: **assumes** $xs \neq []$
shows *factors-of-factor-tree* (*create-factor-tree xs*) = *mset xs*
proof −
let *?xs* = *rev* (*sort-key snd* (*map* (λf. (*f*, *degree f*)) *xs*))
from *assms* **have** *set xs* $\neq \{\}$ **by** *auto*
hence *set ?xs* $\neq \{\}$ **by** *auto*
hence *xs*: *?xs* $\neq []$ **by** *blast*
show *?thesis* **unfolding** *create-factor-tree-def Let-def create-factor-tree-balanced*[*OF xs*]
by (*auto*, *induct xs*, *auto*)
qed

context
fixes *p* :: *int* **and** *n* :: *nat*
begin

definition *quadratic-hensel-binary* :: *int poly* ⇒ *int poly* ⇒ *int poly* ⇒ *int poly* × *int poly* **where**
quadratic-hensel-binary C D H = (
case euclid-ext-poly-dynamic p D H of
(*S*,*T*) ⇒ *quadratic-hensel-main C p S T D H n*)

fun *hensel-lifting-main* :: *int poly* ⇒ *int poly factor-tree* ⇒ *int poly list* **where**
hensel-lifting-main U (*Factor-Leaf - -*) = [*U*]
| *hensel-lifting-main U* (*Factor-Node - l r*) = (*let*
v = *factor-node-info l*;
w = *factor-node-info r*;
(*V*,*W*) = *quadratic-hensel-binary U v w*
in hensel-lifting-main V l @ *hensel-lifting-main W r*)

definition *hensel-lifting-monic* :: *int poly* ⇒ *int poly list* ⇒ *int poly list* **where**
hensel-lifting-monic u vs = (*if vs* = [] *then* [] *else let*
pn = *p*^*n*;
C = *poly-mod.Mp pn u*;
tree = *product-factor-tree p* (*create-factor-tree vs*)
in hensel-lifting-main C tree)

definition *hensel-lifting* :: *int poly* ⇒ *int poly list* ⇒ *int poly list* **where**
hensel-lifting f gs = (*let lc* = *lead-coeff f*;
ilc = *inverse-mod lc* (*p*^*n*);
g = *smult ilc f*
in hensel-lifting-monic g gs)

end

context *poly-mod-prime* **begin**

context
fixes n :: *nat*
assumes n: $n \neq 0$
begin

abbreviation *hensel-binary* $\equiv$ *quadratic-hensel-binary p n*

abbreviation *hensel-main* $\equiv$ *hensel-lifting-main p n*

lemma *hensel-binary*:
assumes *cop*: *coprime-m D H* **and** *eq*: *eq-m C* $(D * H)$
and *normalized-input*: *Mp D* = *D Mp H* = *H*
and *monic-input*: *monic D*
and *hensel-result*: *hensel-binary C D H* = (D',H')
shows *poly-mod.eq-m* $(p\widehat{\ }n)$ C $(D' * H')$ — the main result: equivalence mod $p\widehat{\ }n$
$\wedge$ *monic* D' — monic output
$\wedge$ *eq-m D* D' $\wedge$ *eq-m H* H' — apply *'mod p'* on D' and H' yields D and H again
$\wedge$ *poly-mod.Mp* $(p\widehat{\ }n)$ $D' = D'$ $\wedge$ *poly-mod.Mp* $(p\widehat{\ }n)$ $H' = H'$ — output is normalized
proof −
from *m1* **have** *p*: $p > 1$ **.**
obtain *S T* **where** *ext*: *euclid-ext-poly-dynamic p D H* = (S,T) **by** *force*
obtain *D1 H1* **where** *main*: *quadratic-hensel-main C p S T D H n* = $(D1,H1)$ **by** *force*
note *hen* = *hensel-result*[*unfolded quadratic-hensel-binary-def ext split Let-def main*]
from *n* **have** *n*: $n \geq 1$ **by** *simp*
note *eucl* = *euclid-ext-poly-dynamic*[*OF cop normalized-input ext*]
note *main* = *quadratic-hensel-main*[*OF eucl*(*1*) *eq monic-input p normalized-input eucl*(*2*−) *n main*]
show *?thesis* **using** *hen main* **by** *auto*
qed

lemma *hensel-main*:
assumes *eq*: *eq-m C* (*prod-mset* (*factors-of-factor-tree Fs*))
and $\bigwedge$ *F*. *F* $\in\#$ *factors-of-factor-tree Fs* $\Longrightarrow$ *Mp F* = *F* $\wedge$ *monic F*
and *hensel-result*: *hensel-main C Fs* = *Gs*
and *C*: *monic C poly-mod.Mp* $(p\widehat{\ }n)$ *C* = *C*
and *sf*: *square-free-m C*
and $\bigwedge$ *f t*. *t* $\in$ *sub-trees Fs* $\Longrightarrow$ *factor-node-info t* = *f* $\Longrightarrow$ *f* = *Mp* (*prod-mset* (*factors-of-factor-tree t*))
shows *poly-mod.eq-m* $(p\widehat{\ }n)$ *C* (*prod-list Gs*) — the main result: equivalence mod $p\widehat{\ }n$
$\wedge$ *factors-of-factor-tree Fs* = *mset* (*map Mp Gs*)
$\wedge$ ($\forall$ *G*. *G* $\in$ *set Gs* $\longrightarrow$ *monic G* $\wedge$ *poly-mod.Mp* $(p\widehat{\ }n)$ *G* = *G*)

using *assms*
proof (*induct Fs arbitrary*: *C Gs*)
case (*Factor-Leaf f fs C Gs*)
thus *?case* **by** *auto*
next
case (*Factor-Node f l r C Gs*) **note** $*$ = *this*
note *simps* = *hensel-lifting-main.simps*
note *IH1* = $*$(*1*)[*rule-format*]
note *IH2* = $*$(*2*)[*rule-format*]
note *res* = $*$(*5*)[*unfolded simps Let-def*]
note *eq* = $*$(*3*)
note *Fs* = $*$(*4*)
note *C* = $*$(*6*,*7*)
note *sf* = $*$(*8*)
note *inv* = $*$(*9*)
interpret *pn*: *poly-mod-2 p*^*n* **apply** (*unfold-locales*) **using** *m1 n* **by** *auto*
let *?Mp* = *pn.Mp*
define *D* **where** *D* ≡ *prod-mset* (*factors-of-factor-tree l*)
define *H* **where** *H* ≡ *prod-mset* (*factors-of-factor-tree r*)
let *?D* = *Mp D*
let *?H* = *Mp H*
let *?D′* = *factor-node-info l*
let *?H′* = *factor-node-info r*
obtain *A B* **where** *hen*: *hensel-binary C ?D′ ?H′* = (*A*,*B*) **by** *force*
note *res* = *res*[*unfolded hen split*]
obtain *AD* **where** *AD′*: *AD* = *hensel-main A l* **by** *auto*
obtain *BH* **where** *BH′*: *BH* = *hensel-main B r* **by** *auto*
from *inv*[*of l*, *OF* - *refl*] **have** *D′*: *?D′* = *?D* **unfolding** *D-def* **by** *auto*
from *inv*[*of r*, *OF* - *refl*] **have** *H′*: *?H′* = *?H* **unfolding** *H-def* **by** *auto*
from *eq*[*simplified*]
have *eq′*: *Mp C* = *Mp* (*?D* $*$ *?H*) **unfolding** *D-def H-def* **by** *simp*
from *square-free-m-cong*[*OF sf*, *of ?D* $*$ *?H*, *OF eq′*]
have *sf′*: *square-free-m* (*?D* $*$ *?H*) **.**
from *poly-mod-prime.square-free-m-prod-imp-coprime-m*[*OF* - *this*]
have *cop′*: *coprime-m ?D ?H* **unfolding** *poly-mod-prime-def* **using** *prime* **.**
from *eq′* **have** *eq′*: *eq-m C* (*?D* $*$ *?H*) **by** *simp*
have *monD*: *monic D* **unfolding** *D-def* **by** (*rule monic-prod-mset*, *insert Fs*, *auto*)
from *hensel-binary*[*OF* - - - - - *hen*, *unfolded D′ H′*, *OF cop′ eq′ Mp-Mp Mp-Mp monic-Mp*[*OF monD*]]
have *step*: *poly-mod.eq-m* (p ^ n) *C* (*A* $*$ *B*) ∧ *monic A* ∧ *eq-m ?D A* ∧
eq-m ?H B ∧ *?Mp A* = *A* ∧ *?Mp B* = *B* **.**
from *res* **have** *Gs*: *Gs* = *AD* @ *BH* **by** (*simp add*: *AD′ BH′*)
have *AD*: *eq-m A ?D ?Mp A* = *A eq-m A* (*prod-mset* (*factors-of-factor-tree l*))
and *monA*: *monic A*
using *step* **by** (*auto simp*: *D-def*)
note *sf-fact* = *square-free-m-factor*[*OF sf′*]
from *square-free-m-cong*[*OF sf-fact*(*1*)] *AD* **have** *sfA*: *square-free-m A* **by** *auto*
have *IH1*: *poly-mod.eq-m* (p ^ n) *A* (*prod-list AD*) ∧

factors-of-factor-tree l = mset (map Mp AD) ∧
(∀ *G. G* ∈ *set AD* ⟶ *monic G* ∧ *?Mp G = G*)
by (*rule IH1*[*OF AD*(*3*) *Fs AD'*[*symmetric*] *monA AD*(*2*) *sfA inv*], *auto*)
have *BH*: *eq-m B ?H pn.Mp B = B eq-m B* (*prod-mset* (*factors-of-factor-tree r*))
using *step* **by** (*auto simp*: *H-def*)
from *step* **have** *pn.eq-m C* (*A* ∗ *B*) **by** *simp*
hence *?Mp C = ?Mp* (*A* ∗ *B*) **by** *simp*
with *C AD*(*2*) **have** *pn.Mp C = pn.Mp* (*A* ∗ *pn.Mp B*) **by** *simp*
from *arg-cong*[*OF this*, *of lead-coeff*] *C*
have *monic* (*pn.Mp* (*A* ∗ *B*)) **by** *simp*
then have *lead-coeff* (*pn.Mp A*) ∗ *lead-coeff* (*pn.Mp B*) = *1*
by (*metis lead-coeff-mult leading-coeff-neq-0 local.step mult-cancel-right2 pn.degree-m-eq pn.m1 poly-mod.M-def poly-mod.Mp-coeff*)
with *monA AD*(*2*) *BH*(*2*) **have** *monB*: *monic B* **by** *simp*
from *square-free-m-cong*[*OF sf-fact*(*2*)] *BH* **have** *sfB*: *square-free-m B* **by** *auto*
have *IH2*: *poly-mod.eq-m* (*p* ^ *n*) *B* (*prod-list BH*) ∧
factors-of-factor-tree r = mset (*map Mp BH*) ∧
(∀ *G. G* ∈ *set BH* ⟶ *monic G* ∧ *?Mp G = G*)
by (*rule IH2*[*OF BH*(*3*) *Fs BH'*[*symmetric*] *monB BH*(*2*) *sfB inv*], *auto*)
from *step* **have** *?Mp C = ?Mp* (*?Mp A* ∗ *?Mp B*) **by** *auto*
also have *?Mp A = ?Mp* (*prod-list AD*) **using** *IH1* **by** *auto*
also have *?Mp B = ?Mp* (*prod-list BH*) **using** *IH2* **by** *auto*
finally have *poly-mod.eq-m* (*p* ^ *n*) *C* (*prod-list AD* ∗ *prod-list BH*)
by (*auto simp*: *poly-mod.mult-Mp*)
thus *?case* **unfolding** *Gs* **using** *IH1 IH2* **by** *auto*
qed

lemma *hensel-lifting-monic*:
assumes *eq*: *poly-mod.eq-m p C* (*prod-list Fs*)
and *Fs*: ⋀ *F. F* ∈ *set Fs* ⟹ *poly-mod.Mp p F = F* ∧ *monic F*
and *res*: *hensel-lifting-monic p n C Fs = Gs*
and *mon*: *monic* (*poly-mod.Mp* (*p*^*n*) *C*)
and *sf*: *poly-mod.square-free-m p C*
shows *poly-mod.eq-m* (*p*^*n*) *C* (*prod-list Gs*)
mset (*map* (*poly-mod.Mp p*) *Gs*) = *mset Fs*
G ∈ *set Gs* ⟹ *monic G* ∧ *poly-mod.Mp* (*p*^*n*) *G = G*
proof −
note *res = res*[*unfolded hensel-lifting-monic-def Let-def*]
let *?Mp = poly-mod.Mp* (*p* ^ *n*)
let *?C = ?Mp C*
interpret *poly-mod-prime p*
by (*unfold-locales*, *insert n prime*, *auto*)
interpret *pn*: *poly-mod-2 p*^*n* **using** *m1 n poly-mod-2.intro* **by** *auto*
from *eq n* **have** *eq*: *eq-m* (*?Mp C*) (*prod-list Fs*)
using *Mp-Mp-pow-is-Mp eq m1 n* **by** *force*
have *poly-mod.eq-m* (*p*^*n*) *C* (*prod-list Gs*) ∧ *mset* (*map* (*poly-mod.Mp p*) *Gs*) = *mset Fs*
∧ (*G* ∈ *set Gs* ⟶ *monic G* ∧ *poly-mod.Mp* (*p*^*n*) *G = G*)
proof (*cases Fs* = [])

case *True*
with *res* **have** *Gs*: *Gs* = [] **by** *auto*
from *eq* **have** *Mp ?C* = *1* **unfolding** *True* **by** *simp*
hence *degree* (*Mp ?C*) = *0* **by** *simp*
with *degree-m-eq-monic*[*OF mon m1*] **have** *degree ?C* = *0* **by** *simp*
with *mon* **have** *?C* = *1* **using** *monic-degree-0* **by** *blast*
thus *?thesis* **unfolding** *True Gs* **by** *auto*
next
case *False*
let *?t* = *create-factor-tree Fs*
note *tree* = *create-factor-tree*[*OF False*]
from *False res* **have** *hen*: *hensel-main ?C* (*product-factor-tree p ?t*) = *Gs* **by** *auto*
have *tree1*: *x* ∈# *factors-of-factor-tree ?t* ⟹ *Mp x* = *x* **for** *x* **unfolding** *tree* **using** *Fs* **by** *auto*
from *product-factor-tree*[*OF tree1 sub-trees-refl refl*, *of ?t*]
have *id*: (*factors-of-factor-tree* (*product-factor-tree p ?t*)) =
(*factors-of-factor-tree ?t*) **by** *auto*
have *eq*: *eq-m ?C* (*prod-mset* (*factors-of-factor-tree* (*product-factor-tree p ?t*)))
unfolding *id tree* **using** *eq* **by** *auto*
have *id'*: *Mp C* = *Mp ?C* **using** *n* **by** (*simp add*: *Mp-Mp-pow-is-Mp m1*)
have *pn.eq-m ?C* (*prod-list Gs*) ∧ *mset Fs* = *mset* (*map Mp Gs*) ∧ (∀ *G*. *G* ∈ *set Gs* ⟶ *monic G* ∧ *pn.Mp G* = *G*)
by (*rule hensel-main*[*OF eq Fs hen mon pn.Mp-Mp square-free-m-cong*[*OF sf id'*], *unfolded id tree*],
insert product-factor-tree[*OF tree1*], *auto*)
thus *?thesis* **by** *auto*
qed
thus *poly-mod.eq-m* (*p*^*n*) *C* (*prod-list Gs*)
mset (*map* (*poly-mod.Mp p*) *Gs*) = *mset Fs*
G ∈ *set Gs* ⟹ *monic G* ∧ *poly-mod.Mp* (*p*^*n*) *G* = *G* **by** *blast+*
qed

lemma *hensel-lifting*:
assumes *res*: *hensel-lifting p n f fs* = *gs* — result of hensel is fact. *gs*
and *cop*: *coprime* (*lead-coeff f*) *p*
and *sf*: *poly-mod.square-free-m p f*
and *fact*: *poly-mod.factorization-m p f* (*c*, *mset fs*) — input is fact. *fs mod p*
and *c*: *c* ∈ {*0*..<*p*}
and *norm*: (∀ *fi*∈*set fs*. *set* (*coeffs fi*) ⊆ {*0*..<*p*})
shows *poly-mod.factorization-m* (*p*^*n*) *f* (*lead-coeff f*, *mset gs*) — factorization mod *p*^*n*
sort (*map degree fs*) = *sort* (*map degree gs*) — degrees stay the same
⋀ *g*. *g* ∈ *set gs* ⟹ *monic g* ∧ *poly-mod.Mp* (*p*^*n*) *g* = *g* ∧ — monic and normalized
irreducible-m g ∧ — irreducibility even mod *p*

degree-m g = *degree g* — mod *p* does not change degree of *g*
proof −
  interpret *poly-mod-prime p* **using** *prime* **by** *unfold-locales*
  interpret *q*: *poly-mod-2 p^n* **using** *m1 n* **unfolding** *poly-mod-2-def* **by** *auto*
  from *fact* **have** *eq*: *eq-m f* (*smult c* (*prod-list fs*))
    and *mon-fs*: (∀ *fi*∈*set fs*. *monic* (*Mp fi*) ∧ *irreducible*$_d$*-m fi*)
    unfolding *factorization-m-def* **by** *auto*
  {
    fix *f*
    assume *f* ∈ *set fs*
    with *mon-fs norm* **have** *set* (*coeffs f*) ⊆ {*0*..<*p*} **and** *monic* (*Mp f*) **by** *auto*
    hence *monic f* **using** *Mp-ident-iff′* **by** *force*
  } **note** *mon-fs′* = *this*
  have *Mp-id*: ⋀ *f*. *Mp* (*q.Mp f*) = *Mp f* **by** (*simp add*: *Mp-Mp-pow-is-Mp m1 n*)
  let *?lc* = *lead-coeff f*
  let *?q* = *p* ^ *n*
  define *ilc* **where** *ilc* ≡ *inverse-mod ?lc ?q*
  define *F* **where** *F* ≡ *smult ilc f*
  from *res*[*unfolded hensel-lifting-def Let-def*]
  have *hen*: *hensel-lifting-monic p n F fs* = *gs*
    unfolding *ilc-def F-def* **.**
  from *m1 n cop* **have** *inv*: *q.M* (*ilc* ∗ *?lc*) = *1*
    by (*auto simp add*: *q.M-def inverse-mod-pow ilc-def*)
  hence *ilc0*: *ilc* ≠ *0* **by** (*cases ilc* = *0*, *auto*)
  {
    fix *q*
    assume *ilc* ∗ *?lc* = *?q* ∗ *q*
    from *arg-cong*[*OF this*, *of q.M*] **have** *q.M* (*ilc* ∗ *?lc*) = *0*
      unfolding *q.M-def* **by** *auto*
    with *inv* **have** *False* **by** *auto*
  } **note** *not-dvd* = *this*
  have *mon*: *monic* (*q.Mp F*) **unfolding** *F-def q.Mp-coeff coeff-smult*
    by (*subst q.degree-m-eq* [*OF - q.m1*]) (*auto simp*: *inv ilc0* [*symmetric*] *intro*: *not-dvd*)
  have *q.Mp f* = *q.Mp* (*smult* (*q.M* (*?lc* ∗ *ilc*)) *f*) **using** *inv* **by** (*simp add*: *ac-simps*)
  also have … = *q.Mp* (*smult ?lc F*) **by** (*simp add*: *F-def*)
  finally have *f*: *q.Mp f* = *q.Mp* (*smult ?lc F*) **.**
  from *arg-cong*[*OF f*, *of Mp*]
  have *f-p*: *Mp f* = *Mp* (*smult ?lc F*)
    by (*simp add*: *Mp-Mp-pow-is-Mp n m1*)
  from *arg-cong*[*OF this*, *of square-free-m*, *unfolded Mp-square-free-m*] *sf*
  have *square-free-m* (*smult ?lc F*) **by** *simp*
  from *square-free-m-smultD*[*OF this*] **have** *sf*: *square-free-m F* **.**

  define *c′* **where** *c′* ≡ *M* (*c* ∗ *ilc*)
  from *factorization-m-smult*[*OF fact*, *of ilc*, *folded F-def*]
  have *fact*: *factorization-m F* (*c′*, *mset fs*) **unfolding** *c′-def factorization-m-def* **by** *auto*

hence *eq*: *eq-m F* (*smult c′* (*prod-list fs*)) **unfolding** *factorization-m-def* **by** *auto*
from *factorization-m-lead-coeff*[*OF fact*] *monic-Mp*[*OF mon*, *unfolded Mp-id*]
have *M c′* = *1*
by *auto*
hence *c′*: *c′* = *1* **unfolding** *c′-def* **by** *auto*
with *eq* **have** *eq*: *eq-m F* (*prod-list fs*) **by** *auto*
{
fix *f*
assume *f* ∈ *set fs*
with *mon-fs′ norm* **have** *Mp f* = *f* ∧ *monic f* **unfolding** *Mp-ident-iff′*
by *auto*
} **note** *fs* = *this*
note *hen* = *hensel-lifting-monic*[*OF eq fs hen mon sf*]
from *hen*(*2*) **have** *gs-fs*: *mset* (*map Mp gs*) = *mset fs* **by** *auto*
have *eq*: *q.eq-m f* (*smult ?lc* (*prod-list gs*))
unfolding *f* **using** *arg-cong*[*OF hen*(*1*), *of* λ *f*. *q.Mp* (*smult ?lc f*)] **by** *simp*
{
fix *g*
assume *g*: *g* ∈ *set gs*
from *hen*(*3*)[*OF - g*] **have** *mon-g*: *monic g* **and** *Mp-g*: *q.Mp g* = *g* **by** *auto*
from *g* **have** *Mp g* ∈# *mset* (*map Mp gs*) **by** *auto*
from *this*[*unfolded gs-fs*] **obtain** *f* **where** *f*: *f* ∈ *set fs* **and** *fg*: *eq-m f g* **by** *auto*
from *mon-fs f fs* **have** *irr-f*: *irreducible$_d$-m f* **and** *mon-f*: *monic f* **and** *Mp-f*: *Mp f* = *f* **by** *auto*
have *deg*: *degree-m g* = *degree g*
by (*rule degree-m-eq-monic*[*OF mon-g m1*])
from *irr-f fg* **have** *irr-g*: *irreducible$_d$-m g*
unfolding *irreducible$_d$-m-def dvdm-def* **by** *simp*
have *q.irreducible$_d$-m g*
by (*rule irreducible$_d$-lifting*[*OF* n *- irr-g*], *unfold deg*, *rule q.degree-m-eq-monic*[*OF mon-g q.m1*])
note *mon-g Mp-g deg irr-g this*
} **note** *g* = *this*
{
fix *g*
assume *g* ∈ *set gs*
from *g*[*OF this*]
show *monic g* ∧ *q.Mp g* = *g* ∧ *irreducible-m g* ∧ *degree-m g* = *degree g* **by** *auto*
}
show *sort* (*map degree fs*) = *sort* (*map degree gs*)
proof (*rule sort-key-eq-sort-key*)
have *mset* (*map degree fs*) = *image-mset degree* (*mset fs*) **by** *auto*
also have ... = *image-mset degree* (*mset* (*map Mp gs*)) **unfolding** *gs-fs* **..**
also have ... = *mset* (*map degree* (*map Mp gs*)) **unfolding** *mset-map* **..**
also have *map degree* (*map Mp gs*) = *map degree-m gs* **by** *auto*
also have ... = *map degree gs* **using** *g*(*3*) **by** *auto*
finally show *mset* (*map degree fs*) = *mset* (*map degree gs*) **.**

  qed *auto*
  show *q.factorization-m f* (*lead-coeff f*, *mset gs*)
    using *eq g* **unfolding** *q.factorization-m-def* **by** *auto*
qed

end

end
end

theory *Hensel-Lifting-Type-Based*
imports *Hensel-Lifting*
begin

9.2 Hensel Lifting in a Type-Based Setting

lemma *degree-smult-eq-iff*:
  degree (*smult a p*) = *degree p* $\longleftrightarrow$ *degree p* = *0* $\vee$ *a* $*$ *lead-coeff p* $\neq$ *0*
  by (*metis* (*no-types*, *lifting*) *coeff-smult degree-0 degree-smult-le le-antisym*
    le-degree le-zero-eq leading-coeff-0-iff)

lemma *degree-smult-eqI*[*intro*!]:
  assumes *degree p* $\neq$ *0* $\Longrightarrow$ *a* $*$ *lead-coeff p* $\neq$ *0*
  shows *degree* (*smult a p*) = *degree p*
  using *assms degree-smult-eq-iff* **by** *auto*

lemma *degree-mult-eq2*:
  assumes *lc*: *lead-coeff p* $*$ *lead-coeff q* $\neq$ *0*
  shows *degree* ($p * q$) = *degree p* + *degree q* (**is** - = *?r*)
proof(*intro antisym*[*OF degree-mult-le*] *le-degree*, *unfold coeff-mult*)
  let *?f* = λi. *coeff p i* $*$ *coeff q* (*?r* $-$ *i*)
  have ($\sum i \leq ?r.\ ?f\ i$) = *sum ?f* {..*degree p*} + *sum ?f* {*Suc* (*degree p*)..*?r*}
    by (*rule sum-up-index-split*)
  also have *sum ?f* {*Suc* (*degree p*)..*?r*} = *0*
    proof−
      { **fix** *x* **assume** *x* > *degree p*
        then have *coeff p x* = *0* **by** (*rule coeff-eq-0*)
        then have *?f x* = *0* **by** *auto*
      }
      then show *?thesis* **by** (*intro sum.neutral*, *auto*)
    qed
  also have *sum ?f* {..*degree p*} = *sum ?f* {..<*degree p*} + *?f* (*degree p*)
    by(*fold lessThan-Suc-atMost*, *unfold sum.lessThan-Suc*, *auto*)
  also have *sum ?f* {..<*degree p*} = *0*
    proof−
      {**fix** *x* **assume** *x* < *degree p*
        then have *coeff q* (*?r* − *x*) = *0* **by** (*intro coeff-eq-0*, *auto*)
        then have *?f x* = *0* **by** *auto*
      }

then show *?thesis* **by** (*intro sum.neutral*, *auto*)
qed
finally show ($\sum$ *i*≤*?r*. *?f i*) ≠ *0* **using** *assms* **by** (*auto simp*:)
qed

lemma *degree-mult-eq-left-unit*:
fixes *p q* :: *'a* :: *comm-semiring-1 poly*
assumes *unit*: *lead-coeff p dvd 1* **and** *q0*: *q* ≠ *0*
shows *degree* (*p* ∗ *q*) = *degree p* + *degree q*
proof(*intro degree-mult-eq2 notI*)
from *unit* **obtain** *c* **where** *lead-coeff p* ∗ *c* = *1* **by** (*elim dvdE*,*auto*)
then have *c* ∗ *lead-coeff p* = *1* **by** (*auto simp*: *ac-simps*)
moreover assume *lead-coeff p* ∗ *lead-coeff q* = *0*
then have *c* ∗ *lead-coeff p* ∗ *lead-coeff q* = *0* **by** (*auto simp*: *ac-simps*)
ultimately have *lead-coeff q* = *0* **by** *auto*
with *q0* **show** *False* **by** *auto*
qed

context *ring-hom* **begin**
lemma *monic-degree-map-poly-hom*: *monic p* ⟹ *degree* (*map-poly hom p*) = *degree p*
by (*auto intro*: *degree-map-poly*)

lemma *monic-map-poly-hom*: *monic p* ⟹ *monic* (*map-poly hom p*)
by (*simp add*: *monic-degree-map-poly-hom*)

end

lemma *of-nat-zero*:
assumes *CARD*(*'a*::*nontriv*) *dvd n*
shows (*of-nat n* :: *'a mod-ring*) = *0*
apply (*transfer fixing*: *n*) **using** *assms* **by** (*presburger*)

abbreviation *rebase* :: *'a* :: *nontriv mod-ring* ⇒ *'b* :: *nontriv mod-ring* (@- [*100*]*100*)
where @*x* ≡ *of-int* (*to-int-mod-ring x*)

abbreviation *rebase-poly* :: *'a* :: *nontriv mod-ring poly* ⇒ *'b* :: *nontriv mod-ring poly* (#- [*100*]*100*)
where #*x* ≡ *of-int-poly* (*to-int-poly x*)

lemma *rebase-self* [*simp*]:
@*x* = *x*
by (*simp add*: *of-int-of-int-mod-ring*)

lemma *map-poly-rebase* [*simp*]:
map-poly rebase p = #*p*
by (*induct p*) *simp-all*

lemma *rebase-poly-0*: #*0* = *0*

by *simp*

lemma *rebase-poly-1*: *#1* = *1*
by *simp*

lemma *rebase-poly-pCons*[*simp*]: *#pCons a p* = *pCons* (*@a*) (*#p*)
by(*cases a* = *0* ∧ *p* = *0*, *simp*, *fold map-poly-rebase*, *subst map-poly-pCons*, *auto*)

lemma *rebase-poly-self*[*simp*]: *#p* = *p* **by** (*induct p*, *auto*)

lemma *degree-rebase-poly-le*: *degree* (*#p*) ≤ *degree p*
by (*fold map-poly-rebase*, *subst degree-map-poly-le*, *auto*)

lemma(**in** *comm-ring-hom*) *degree-map-poly-unit*: **assumes** *lead-coeff p dvd 1*
shows *degree* (*map-poly hom p*) = *degree p*
using *hom-dvd-1*[*OF assms*] **by** (*auto intro*: *degree-map-poly*)

lemma *rebase-poly-eq-0-iff*:
(*#p* :: *'a* :: *nontriv mod-ring poly*) = *0* ⟷ (∀ *i*. (*@coeff p i* :: *'a mod-ring*) = *0*) (**is** *?l* ⟷ *?r*)
proof(*intro iffI*)
assume *?l*
then have *coeff* (*#p* :: *'a mod-ring poly*) *i* = *0* **for** *i* **by** *auto*
then show *?r* **by** *auto*
next
assume *?r*
then have *coeff* (*#p* :: *'a mod-ring poly*) *i* = *0* **for** *i* **by** *auto*
then show *?l* **by** (*intro poly-eqI*, *auto*)
qed

lemma *mod-mod-le*:
assumes *ab*: (*a*::*int*) ≤ *b* **and** *a0*: *0* < *a* **and** *c0*: *c* ≥ *0* **shows** (*c mod a*) *mod b* = *c mod a*
by (*meson Divides.pos-mod-bound Divides.pos-mod-sign a0 ab less-le-trans mod-pos-pos-trivial*)

locale *rebase-ge* =
fixes *ty1* :: *'a* :: *nontriv itself* **and** *ty2* :: *'b* :: *nontriv itself*
assumes *card*: *CARD*(*'a*) ≤ *CARD*(*'b*)
begin

lemma *ab*: *int CARD*(*'a*) ≤ *CARD*(*'b*) **using** *card* **by** *auto*

lemma *rebase-eq-0*[*simp*]:
shows (*@*(*x* :: *'a mod-ring*) :: *'b mod-ring*) = *0* ⟷ *x* = *0*
using *card* **by** (*transfer*, *auto*)

lemma *degree-rebase-poly-eq*[*simp*]:
shows *degree* (*#*(*p* :: *'a mod-ring poly*) :: *'b mod-ring poly*) = *degree p*
by (*subst degree-map-poly*; *simp*)

lemma *lead-coeff-rebase-poly*[*simp*]:
lead-coeff (#(*p*::′*a mod-ring poly*) :: ′*b mod-ring poly*) = @*lead-coeff p*
by *simp*

lemma *to-int-mod-ring-rebase*: *to-int-mod-ring*(@(*x* :: ′*a mod-ring*)::′*b mod-ring*) = *to-int-mod-ring x*
using *card* **by** (*transfer*, *auto*)

lemma *rebase-id*[*simp*]: @(@(*x*::′*a mod-ring*) :: ′*b mod-ring*) = @*x*
using *card* **by** (*transfer*, *auto*)

lemma *rebase-poly-id*[*simp*]: #(#(*p*::′*a mod-ring poly*) :: ′*b mod-ring poly*) = #*p*
by (*induct p*, *auto*)

end

locale *rebase-dvd* =
fixes *ty1* :: ′*a* :: *nontriv itself* **and** *ty2* :: ′*b* :: *nontriv itself*
assumes *dvd*: *CARD*(′*b*) *dvd CARD*(′*a*)
begin

lemma *ab*: *CARD*(′*a*) ≥ *CARD*(′*b*) **by** (*rule dvd-imp-le*[*OF dvd*], *auto*)

lemma *rebase-id*[*simp*]: @(@(*x*::′*b mod-ring*) :: ′*a mod-ring*) = *x* **using** *ab* **by** (*transfer*, *auto*)

lemma *rebase-poly-id*[*simp*]: #(#(*p*::′*b mod-ring poly*) :: ′*a mod-ring poly*) = *p* **by** (*induct p*, *auto*)

lemma *rebase-of-nat*[*simp*]: (@(*of-nat n* :: ′*a mod-ring*) :: ′*b mod-ring*) = *of-nat n*
apply *transfer* **apply** (*rule mod-mod-cancel*) **using** *dvd* **by** *presburger*

lemma *mod-1-lift-nat*:
assumes (*of-int* (*int x*) :: ′*a mod-ring*) = *1*
shows (*of-int* (*int x*) :: ′*b mod-ring*) = *1*
proof −
from *assms* **have** *int x mod CARD*(′*a*) = *1*
by *transfer*
then have *x mod CARD*(′*a*) = *1*
by (*simp add*: *of-nat-mod* [*symmetric*])
then have *x mod CARD*(′*b*) = *1*
by (*metis dvd mod-mod-cancel one-mod-card*)
then have *int x mod CARD*(′*b*) = *1*
by (*simp add*: *of-nat-mod* [*symmetric*])
then show *?thesis*
by *transfer*
qed

sublocale *comm-ring-hom rebase* :: *'a mod-ring* ⇒ *'b mod-ring*
proof
  fix *x y* :: *'a mod-ring*
  show *hom-add*: (@(*x*+*y*) :: *'b mod-ring*) = @*x* + @*y*
    by *transfer* (*simp add*: *mod-simps dvd mod-mod-cancel*)
  show (@(*x*∗*y*) :: *'b mod-ring*) = @*x* ∗ @*y*
    by *transfer* (*simp add*: *mod-simps dvd mod-mod-cancel*)
qed *auto*

lemma *of-nat-CARD-eq-0*[*simp*]: (*of-nat CARD*(*'a*) :: *'b mod-ring*) = *0*
  using *dvd* **by** (*transfer*, *presburger*)

interpretation *map-poly-hom*: *map-poly-comm-ring-hom rebase* :: *'a mod-ring* ⇒ *'b mod-ring*..

sublocale *poly*: *comm-ring-hom rebase-poly* :: *'a mod-ring poly* ⇒ *'b mod-ring poly*
  by (*fold map-poly-rebase*, *unfold-locales*)

lemma *poly-rebase*[*simp*]: @*poly p x* = *poly* (#(*p* :: *'a mod-ring poly*) :: *'b mod-ring poly*) (@(*x*::*'a mod-ring*) :: *'b mod-ring*)
  by (*fold map-poly-rebase poly-map-poly*, *rule*)

lemma *rebase-poly-smult*[*simp*]: (#(*smult a p* :: *'a mod-ring poly*) :: *'b mod-ring poly*) = *smult* (@*a*) (#*p*)
  by(*induct p*, *auto simp*: *hom-distribs*)

end

locale *rebase-mult* =
  fixes *ty1* :: *'a* :: *nontriv itself*
    and *ty2* :: *'b* :: *nontriv itself*
    and *ty3* :: *'d* :: *nontriv itself*
  assumes *d*: *CARD*(*'a*) = *CARD*(*'b*) ∗ *CARD*(*'d*)
begin

sublocale *rebase-dvd ty1 ty2* **using** *d* **by** (*unfold-locales*, *auto*)

lemma *rebase-mult-eq*[*simp*]: (*of-nat CARD*(*'d*) ∗ *a* :: *'a mod-ring*) = *of-nat CARD*(*'d*) ∗ *a'* ⟷ (@*a* :: *'b mod-ring*) = @*a'*
proof−
  from *dvd* **obtain** *d'* **where** *CARD*(*'a*) = *d'* ∗ *CARD*(*'b*) **by** (*elim dvdE*, *auto*)
  then show *?thesis* **by** (*transfer*, *auto simp*:*d*)
qed

lemma *rebase-poly-smult-eq*[*simp*]:
  fixes *a a'* :: *'a mod-ring poly*
  defines *d* ≡ *of-nat CARD*(*'d*) :: *'a mod-ring*
  shows *smult d a* = *smult d a'* ⟷ (#*a* :: *'b mod-ring poly*) = #*a'* (**is** *?l* ⟷

?r)
proof (*intro iffI*)
  assume *l*: *?l* **show** *?r*
  proof (*intro poly-eqI*)
    fix *n*
    from *l* **have** *coeff* (*smult d a*) *n* = *coeff* (*smult d a'*) *n* **by** *auto*
    then have *d* ∗ *coeff a n* = *d* ∗ *coeff a' n* **by** *auto*
    from *this*[*unfolded d-def rebase-mult-eq*]
    show *coeff* (#*a* :: *'b mod-ring poly*) *n* = *coeff* (#*a'*) *n* **by** *auto*
  qed
next
  assume *r*: *?r* **show** *?l*
  proof(*intro poly-eqI*)
    fix *n*
    from *r* **have** *coeff* (#*a* :: *'b mod-ring poly*) *n* = *coeff* (#*a'*) *n* **by** *auto*
    then have (@*coeff a n* :: *'b mod-ring*) = @*coeff a' n* **by** *auto*
    from *this*[*folded d-def rebase-mult-eq*]
    show *coeff* (*smult d a*) *n* = *coeff* (*smult d a'*) *n* **by** *auto*
  qed
qed

lemma *rebase-eq-0-imp-ex-mult*:
  (@(*a* :: *'a mod-ring*) :: *'b mod-ring*) = *0* ⟹ (∃ *c* :: *'d mod-ring*. *a* = *of-nat CARD*(*'b*) ∗ @*c*) (**is** *?l* ⟹ *?r*)
proof(*cases CARD*(*'a*) = *CARD*(*'b*))
  case *True* **then show** *?l* ⟹ *?r*
    by (*transfer*, *auto*)
next
  case *False*
  have [*simp*]: *int CARD*(*'b*) *mod int CARD*(*'a*) = *int CARD*(*'b*)
    by(*rule mod-pos-pos-trivial*, *insert ab False*, *auto*)
  {
    fix *a*
    assume *a*: *0* ≤ *a a* < *int CARD*(*'a*) **and** *mod*: *a mod int CARD*(*'b*) = *0*
    from *mod* **have** *int CARD*(*'b*) *dvd a* **by** *auto*
    then obtain *i* **where** ∗: *a* = *int CARD*(*'b*) ∗ *i* **by** (*elim dvdE*, *auto*)
    from ∗ *a* **have** *i* < *int CARD*(*'d*) **by** (*simp add*:*d*)
    moreover
      hence (*i mod int CARD*(*'a*)) = *i*
        by (*metis dual-order.order-iff-strict less-le-trans not-le of-nat-less-iff* ∗ *a*(*1*) *a*(*2*)
            mod-pos-pos-trivial mult-less-cancel-right1 nat-neq-iff nontriv of-nat-1)
      with ∗ *a* **have** *a* = *int CARD*(*'b*) ∗ (*i mod int CARD*(*'a*)) *mod int CARD*(*'a*)
        by (*auto simp*:*d*)
    moreover from ∗ *a* **have** *0* ≤ *i*
    using *linordered-semiring-strict-class.mult-pos-neg of-nat-0-less-iff zero-less-card-finite*
      by (*simp add*: *zero-le-mult-iff*)
    ultimately have ∃ *i*≥*0*. *i* < *int CARD*(*'d*) ∧ *a* = *int CARD*(*'b*) ∗ (*i mod int CARD*(*'a*)) *mod int CARD*(*'a*)

by (*auto intro*: *exI*[*of* - *i*])
}
then show *?l* $\Longrightarrow$ *?r* **by** (*transfer*, *auto simp*:*d*)
qed

lemma *rebase-poly-eq-0-imp-ex-smult*:
(#(*p* :: *'a mod-ring poly*) :: *'b mod-ring poly*) = *0* $\Longrightarrow$
($\exists$ *p'* :: *'d mod-ring poly*. (*p* = *0* $\longleftrightarrow$ *p'* = *0*) $\wedge$ *degree p'* $\leq$ *degree p* $\wedge$ *p* = *smult* (*of-nat CARD*(*'b*)) (#*p'*))
(**is** *?l* $\Longrightarrow$ *?r*)
proof(*induct p*)
case *0*
then show *?case* **by** (*intro exI*[*of* - *0*],*auto*)
next
case *IH*: (*pCons a p*)
from *IH*(*3*) **have** (#*p* :: *'b mod-ring poly*) = *0* **by** *auto*
from *IH*(*2*)[*OF this*] **obtain** *p'* :: *'d mod-ring poly*
where ∗: *p* = *0* $\longleftrightarrow$ *p'* = *0* *degree p'* $\leq$ *degree p p* = *smult* (*of-nat CARD*(*'b*)) (#*p'*) **by** (*elim exE conjE*)
from *IH* **have** (@*a* :: *'b mod-ring*) = *0* **by** *auto*
from *rebase-eq-0-imp-ex-mult*[*OF this*]
obtain *a'* :: *'d mod-ring* **where** *a'*: *of-nat CARD*(*'b*) ∗ (@*a'*) = *a* **by** *auto*
from *IH*(*1*) **have** *pCons a p* $\neq$ *0* **by** *auto*
moreover from ∗(*1*,*2*) **have** *degree* (*pCons a' p'*) $\leq$ *degree* (*pCons a p*) **by** *auto*
moreover from *a'* ∗(*3*)
have *pCons a p* = *smult* (*of-nat CARD*(*'b*)) (#*pCons a' p'*) **by** *auto*
ultimately show *?case* **by** (*intro exI*[*of* - *pCons a' p'*], *auto*)
qed

end

lemma *mod-mod-nat*[*simp*]: *a mod b mod* (*b* ∗ *c* :: *nat*) = *a mod b* **by** (*simp add*: *Divides.mod-mult2-eq*)

locale *Knuth-ex-4-6-2-22-base* =
fixes *ty-p* :: *'p* :: *nontriv itself*
and *ty-q* :: *'q* :: *nontriv itself*
and *ty-pq* :: *'pq* :: *nontriv itself*
assumes *pq*: *CARD*(*'pq*) = *CARD*(*'p*) ∗ *CARD*(*'q*)
and *p-dvd-q*: *CARD*(*'p*) *dvd CARD*(*'q*)
begin

sublocale *rebase-q-to-p*: *rebase-dvd TYPE*(*'q*) *TYPE*(*'p*) **using** *p-dvd-q* **by** (*unfold-locales*, *auto*)
sublocale *rebase-pq-to-p*: *rebase-mult TYPE*(*'pq*) *TYPE*(*'p*) *TYPE*(*'q*) **using** *pq*
by (*unfold-locales*, *auto*)
sublocale *rebase-pq-to-q*: *rebase-mult TYPE*(*'pq*) *TYPE*(*'q*) *TYPE*(*'p*) **using** *pq*

by (*unfold-locales*, *auto*)

sublocale *rebase-p-to-q*: *rebase-ge TYPE*(*'p*) *TYPE* (*'q*) **by** (*unfold-locales*, *insert p-dvd-q*, *simp add*: *dvd-imp-le*)
sublocale *rebase-p-to-pq*: *rebase-ge TYPE*(*'p*) *TYPE* (*'pq*) **by** (*unfold-locales*, *simp add*: *pq*)
sublocale *rebase-q-to-pq*: *rebase-ge TYPE*(*'q*) *TYPE* (*'pq*) **by** (*unfold-locales*, *simp add*: *pq*)

definition $p \equiv$ *if* (*ty-p* :: *'p itself*) = *ty-p then CARD*(*'p*) *else undefined*
lemma *p*[*simp*]: $p \equiv$ *CARD*(*'p*) **unfolding** *p-def* **by** *auto*

definition $q \equiv$ *if* (*ty-q* :: *'q itself*) = *ty-q then CARD*(*'q*) *else undefined*
lemma *q*[*simp*]: *q* = *CARD*(*'q*) **unfolding** *q-def* **by** *auto*

lemma *p1*: *int p* > *1*
  using *nontriv* [**where** *?'a* = *'p*] *p* **by** *simp*
lemma *q1*: *int q* > *1*
  using *nontriv* [**where** *?'a* = *'q*] *q* **by** *simp*
lemma *q0*: *int q* > *0*
  using *q1* **by** *auto*

lemma *pq2*[*simp*]: *CARD*(*'pq*) = *p* ∗ *q* **using** *pq* **by** *simp*

lemma *qq-eq-0*[*simp*]: (*of-nat CARD*(*'q*) ∗ *of-nat CARD*(*'q*) :: *'pq mod-ring*) = *0*
proof−
  have (*of-nat* (*q* ∗ *q*) :: *'pq mod-ring*) = *0* **by** (*rule of-nat-zero*, *auto simp*: *p-dvd-q*)
  then show *?thesis* **by** *auto*
qed

lemma *of-nat-q*[*simp*]: *of-nat q* :: *'q mod-ring* $\equiv$ *0* **by** (*fold of-nat-card-eq-0*, *auto*)

lemma *rebase-rebase*[*simp*]: (@(@(*x*::*'pq mod-ring*) :: *'q mod-ring*) :: *'p mod-ring*) = @*x*
  using *p-dvd-q* **by** (*transfer*) (*simp add*: *mod-mod-cancel*)

lemma *rebase-rebase-poly*[*simp*]: (#(#(*f*::*'pq mod-ring poly*) :: *'q mod-ring poly*) :: *'p mod-ring poly*) = #*f*
  by (*induct f*, *auto*)

end

definition *dupe-monic* **where**
  dupe-monic D H S T U = (*case pdivmod-monic* (*T* ∗ *U*) *D of* (*q*,*r*) ⇒ (*S* ∗ *U* + *H* ∗ *q*, *r*))

lemma *dupe-monic*:

fixes D :: $'a$ :: *prime-card mod-ring poly*
assumes *1*: $D{*}S + H{*}T = 1$
and *mon*: *monic D*
and *dupe*: *dupe-monic D H S T U* $= (A,B)$
shows $A * D + B * H = U$ $B = 0 \vee$ *degree* $B <$ *degree* D
coprime D H $\Longrightarrow A' * D + B' * H = U \Longrightarrow B' = 0 \vee$ *degree* $B' <$ *degree* D
$\Longrightarrow A' = A \wedge B' = B$
proof −
obtain *q r* **where** *div*: *pdivmod-monic* $(T * U)$ $D = (q,r)$ **by** *force*
from *dupe*[*unfolded dupe-monic-def div split*]
have *A*: $A = (S * U + H * q)$ **and** *B*: $B = r$ **by** *auto*
from *pdivmod-monic*[*OF mon div*] **have** *TU*: $T * U = D * q + r$ **and**
deg: $r = 0 \vee$ *degree* $r <$ *degree* D **by** *auto*
hence *r*: $r = T * U - D * q$ **by** *simp*
have $A * D + B * H = (S * U + H * q) * D + (T * U - D * q) * H$ **unfolding** *A B r* **by** *simp*
also have ... $= (D * S + H * T) * U$ **by** (*simp add*: *field-simps*)
also have $D * S + H * T = 1$ **using** *1* **by** *simp*
finally show *eq*: $A * D + B * H = U$ **by** *simp*
show *degB*: $B = 0 \vee$ *degree* $B <$ *degree* D **using** *deg* **unfolding** *B* **by** (*cases* $r = 0$, *auto*)
assume *another*: $A' * D + B' * H = U$ **and** *degB'*: $B' = 0 \vee$ *degree* $B' <$ *degree* D
and *cop*: *coprime D H*
from *degB* **have** *degB*: $B = 0 \vee$ *degree* $B <$ *degree* D **by** *auto*
from *degB'* **have** *degB'*: $B' = 0 \vee$ *degree* $B' <$ *degree* D **by** *auto*
from *mon* **have** *D0*: $D \neq 0$ **by** *auto*
from *another eq* **have** $A' * D + B' * H = A * D + B * H$ **by** *simp*
from *uniqueness-poly-equality*[*OF cop degB' degB D0 this*]
show $A' = A \wedge B' = B$ **by** *auto*
qed

locale *Knuth-ex-4-6-2-22-main* = *Knuth-ex-4-6-2-22-base p-ty q-ty pq-ty*
for *p-ty* :: $'p$::*nontriv itself*
and *q-ty* :: $'q$::*nontriv itself*
and *pq-ty* :: $'pq$::*nontriv itself* +
fixes *a b* :: $'p$ *mod-ring poly* **and** *u* :: $'pq$ *mod-ring poly* **and** *v w* :: $'q$ *mod-ring poly*
assumes *uvw*: $(\#u$ :: $'q$ *mod-ring poly*$) = v * w$
and *degu*: *degree u* = *degree v* + *degree w*
and *avbw*: $(a * \#v + b * \#w$ :: $'p$ *mod-ring poly*$) = 1$
and *monic-v*: *monic v*

and *bv*: *degree b* < *degree v*
begin

lemma *deg-v*: *degree* $(\#v$ :: $'p$ *mod-ring poly*$)$ = *degree v*
using *monic-v* **by** (*simp add*: *of-int-hom.monic-degree-map-poly-hom*)

lemma *u0*: $u \neq 0$ **using** *degu bv* **by** *auto*

lemma *ex-f*: $\exists f$ :: *'p mod-ring poly*. $u = \#v * \#w + smult\ (of\text{-}nat\ q)\ (\#f)$
proof−
  from *uvw* **have** $(\#(u - \#v * \#w)$ :: *'q mod-ring poly*$) = 0$ **by** (*auto simp*:*hom-distribs*)
  from *rebase-pq-to-q.rebase-poly-eq-0-imp-ex-smult*[*OF this*]
  obtain f :: *'p mod-ring poly* **where** $u - \#v * \#w = smult\ (of\text{-}nat\ q)\ (\#f)$ **by** *force*
  then have $u = \#v * \#w + smult\ (of\text{-}nat\ q)\ (\#f)$ **by** (*metis add-diff-cancel-left' add-diff-eq*)
  then show *?thesis* **by** (*intro exI*[*of - f*], *auto*)
qed

definition f :: *'p mod-ring poly* $\equiv$ *SOME f*. $u = \#v * \#w + smult\ (of\text{-}nat\ q)\ (\#f)$

lemma *u*: $u = \#v * \#w + smult\ (of\text{-}nat\ q)\ (\#f)$
  using *ex-f*[*folded some-eq-ex*] *f-def* **by** *auto*

lemma *t-ex*: $\exists t$ :: *'p mod-ring poly*. *degree* $(b * f - t * \#v) <$ *degree v*
proof−
  define v' **where** $v' \equiv \#v$ :: *'p mod-ring poly*
  from *monic-v*
  have *1*: *lead-coeff* $v' = 1$ **by** (*simp add*: *v'-def deg-v*)
  then have *4*: $v' \neq 0$ **by** *auto*
  obtain *t rem* :: *'p mod-ring poly*
  where *pseudo-divmod* $(b * f)\ v' = (t, rem)$ **by** *force*
  from *pseudo-divmod*[*OF 4 this*, *folded*, *unfolded 1*]
  have $b * f = v' * t + rem$ **and** *deg*: $rem = 0 \lor$ *degree rem* $<$ *degree* v' **by** *auto*
  then have $rem = b * f - t * v'$ **by**(*auto simp*: *ac-simps*)
  also have ... $= b * f - \#(\#t$ :: *'p mod-ring poly*$) * v'$ (**is** - = - − *?t* $* v'$) **by** *simp*
  also have ... $= b * f -$ *?t* $* \#v$
    by (*unfold v'-def*, *rule*)
  finally have *degree rem* = *degree* ... **by** *auto*
  with *deg bv* **have** *degree* $(b * f -$ *?t* $* \#v$ :: *'p mod-ring poly*$) <$ *degree v* **by** (*auto simp*: *v'-def deg-v*)
  then show *?thesis* **by** (*rule exI*)
qed

definition t **where** $t \equiv$ *SOME t* :: *'p mod-ring poly*. *degree* $(b * f - t * \#v) <$ *degree v*

definition $v' \equiv b * f - t * \#v$
definition $w' \equiv a * f + t * \#w$

lemma *f*: $w' * \#v + v' * \#w = f$ (**is** *?l* = -)
proof−

have *?l* = *f* ∗ (*a* ∗ #*v* + *b* ∗ #*w* :: *′p mod-ring poly*) **by** (*simp add*: *v′-def w′-def ring-distribs ac-simps*)
also
from *avbw* **have** (#(*a* ∗ #*v* + *b* ∗ #*w*) :: *′p mod-ring poly*) = *1* **by** *auto*
then have (*a* ∗ #*v* + *b* ∗ #*w* :: *′p mod-ring poly*) = *1* **by** *auto*
finally show *?thesis* **by** *auto*
qed

lemma *degv′*: *degree v′* < *degree v* **by** (*unfold v′-def t-def*, *rule someI-ex*, *rule t-ex*)

lemma *degqf*[*simp*]: *degree* (*smult* (*of-nat CARD*(*′q*)) (#*f* :: *′pq mod-ring poly*)) = *degree* (#*f* :: *′pq mod-ring poly*)
proof (*intro degree-smult-eqI*)
assume *degree* (#*f* :: *′pq mod-ring poly*) ≠ *0*
then have *f0*: *degree f* ≠ *0* **by** *simp*
moreover define *l* **where** *l* ≡ *lead-coeff f*
ultimately have *l0*: *l* ≠ *0* **by** *auto*
then show *of-nat CARD*(*′q*) ∗ *lead-coeff* (#*f*::*′pq mod-ring poly*) ≠ *0*
apply (*unfold rebase-p-to-pq.lead-coeff-rebase-poly*, *fold l-def*)
apply (*transfer*)
using *q1* **by** (*simp add*: *pq mod-mod-cancel*)
qed

lemma *degw′*: *degree w′* ≤ *degree w*
proof(*rule ccontr*)
let *?f* = #*f* :: *′pq mod-ring poly*
let *?qf* = *smult* (*of-nat q*) (#*f*) :: *′pq mod-ring poly*

have *degree* (#*w*::*′p mod-ring poly*) ≤ *degree w* **by** (*rule degree-rebase-poly-le*)
also assume ¬ *degree w′* ≤ *degree w*
then have *1*: *degree w* < *degree w′* **by** *auto*
finally have *2*: *degree* (#*w* :: *′p mod-ring poly*) < *degree w′* **by** *auto*
then have *w′0*: *w′* ≠ *0* **by** *auto*

have *3*: *degree* (#*v* ∗ *w′*) = *degree* (#*v* :: *′p mod-ring poly*) + *degree w′*
using *monic-v*[*unfolded*] **by** (*intro degree-monic-mult*[*OF - w′0*], *auto simp*: *deg-v*)

have *degree f* ≤ *degree u*
proof(*rule ccontr*)
assume ¬*?thesis*
then have ∗: *degree u* < *degree f* **by** *auto*
with *degu* **have** *1*: *degree v* + *degree w* < *degree f* **by** *auto*
define *lcf* **where** *lcf* ≡ *lead-coeff f*
with *1* **have** *lcf0*: *lcf* ≠ *0* **by** (*unfold*, *auto*)
have *degree f* = *degree ?qf* **by** *simp*
also have ... = *degree* (#*v* ∗ #*w* + *?qf*)
proof(*rule sym*, *rule degree-add-eq-right*)
from *1 degree-mult-le*[*of* #*v*::*′pq mod-ring poly* #*w*]

show degree (#v ∗ #w :: 'pq mod-ring poly) < degree ?qf **by** simp
qed
also have ... < degree f **using** ∗ u **by** auto
finally show False **by** auto
qed
with degu **have** degree f ≤ degree v + degree w **by** auto
also note f[symmetric]
finally have degree (w' ∗ #v + v' ∗ #w) ≤ degree v + degree w**.**
moreover have degree (w' ∗ #v + v' ∗ #w) = degree (w' ∗ #v)
proof(rule degree-add-eq-left)
have degree (v' ∗ #w) ≤ degree v' + degree (#w :: 'p mod-ring poly)
by(rule degree-mult-le)
also have ... < degree v + degree (#w :: 'p mod-ring poly) **using** degv' **by** auto
also have ... < degree (#v :: 'p mod-ring poly) + degree w' **using** 2 **by** (auto simp: deg-v)
also have ... = degree (#v ∗ w') **using** 3 **by** auto
finally show degree (v' ∗ #w) < degree (w' ∗ #v) **by** (auto simp: ac-simps)
qed
ultimately have degree (w' ∗ #v) ≤ degree v + degree w **by** auto
moreover
from 3 **have** degree (w' ∗ #v) = degree w' + degree v **by** (auto simp: ac-simps deg-v)
with 1 **have** degree w + degree v < degree (w' ∗ #v) **by** auto
ultimately show False **by** auto
qed

abbreviation qv' ≡ smult (of-nat q) (#v') :: 'pq mod-ring poly
abbreviation qw' ≡ smult (of-nat q) (#w') :: 'pq mod-ring poly

abbreviation V ≡ #v + qv'
abbreviation W ≡ #w + qw'

lemma vV: v = #V **by** (auto simp: v'-def hom-distribs)

lemma wW: w = #W **by** (auto simp: w'-def hom-distribs)

lemma uVW: u = V ∗ W
by (subst u, fold f, simp add: ring-distribs add.left-cancel smult-add-right[symmetric] hom-distribs)

lemma degV: degree V = degree v
and lcV: lead-coeff V = @lead-coeff v
and degW: degree W = degree w
proof−
from p1 q1 **have** int p < int p ∗ int q **by** auto
from less-trans[OF - this]
have 1: l < int p ⟹ l < int p ∗ int q **for** l **by** auto
have degree qv' = degree (#v' :: 'pq mod-ring poly)
proof (rule degree-smult-eqI, safe, unfold rebase-p-to-pq.degree-rebase-poly-eq)

define l **where** $l \equiv$ *lead-coeff* v'
assume *degree* $v' > 0$
then have *lead-coeff* $v' \neq 0$ **by** *auto*
then have $(@l :: 'pq\ mod\text{-}ring) \neq 0$ **by** (*simp add*: *l-def*)
then have $(of\text{-}nat\ q * @l :: 'pq\ mod\text{-}ring) \neq 0$
apply (*transfer fixing*:*q-ty*) **using** *p-dvd-q p1 q1 1* **by** *auto*
moreover assume $of\text{-}nat\ q * coeff\ (\#v')\ (degree\ v') = (0 :: 'pq\ mod\text{-}ring)$
ultimately show *False* **by** (*auto simp*: *l-def*)
qed
also from *degv'* **have** ... $<$ *degree* $(\#v::'pq\ mod\text{-}ring\ poly)$ **by** *simp*
finally have $*$: *degree* $qv' <$ *degree* $(\#v :: 'pq\ mod\text{-}ring\ poly)$**.**
from *degree-add-eq-left*[*OF* $*$]
show $**$: *degree* $V =$ *degree* v **by** (*simp add*: v'-*def*)

from $*$ **have** *coeff* qv' (*degree* v) $= 0$ **by** (*intro coeff-eq-0*, *auto*)
then show *lead-coeff* $V = @$*lead-coeff* v **by** (*unfold* $**$, *auto simp*: v'-*def*)

with *u0 uVW* **have** *degree* $(V * W) =$ *degree* $V +$ *degree* W
by (*intro degree-mult-eq-left-unit*, *auto simp*: *monic-v*)
from *this*[*folded uVW*, *unfolded degu* $**$] **show** *degree* $W =$ *degree* w **by** *auto*
qed

end

locale *Knuth-ex-4-6-2-22-prime* = *Knuth-ex-4-6-2-22-main ty-p ty-q ty-pq a b u v w*
for *ty-p* :: $'p$ :: *prime-card itself*
and *ty-q* :: $'q$ :: *nontriv itself*
and *ty-pq* :: $'pq$ :: *nontriv itself*
and *a b u v w* +
assumes *coprime*: *coprime* $(\#v :: 'p\ mod\text{-}ring\ poly)$ $(\#w)$

begin

lemma *coprime-preserves*: *coprime* $(\#V :: 'p\ mod\text{-}ring\ poly)$ $(\#W)$
apply (*intro coprimeI*,*simp add*: *rebase-q-to-p.of-nat-CARD-eq-0*[*simplified*] *hom-distribs*)
using *coprime* **by** (*elim coprimeE*, *auto*)

lemma *pre-unique*:
assumes *f2*: $w'' * \#v + v'' * \#w = f$
and *degv''*: *degree* $v'' <$ *degree* v
shows $v'' = v' \wedge w'' = w'$
proof(*intro conjI*)
from *f f2*
have $w' * \#v + v' * \#w = w'' * \#v + v'' * \#w$ **by** *auto*
also have ... $- w'' * \#v = v'' * \#w$ **by** *auto*
also have ... $- v' * \#w = (v'' - v') * \#w$ **by** (*auto simp*: *left-diff-distrib*)
finally have $*$: $(w' - w'') * \#v = (v'' - v') * \#w$ **by** (*auto simp*: *left-diff-distrib*)
then have $\#v$ *dvd* $(v'' - v') * \#w$ **by** (*auto intro*: *dvdI*[*of - -* $w' - w''$] *simp*:

ac-simps)
  with *coprime* **have** $\#v$ *dvd* $v'' - v'$
    by (*simp add*: *coprime-dvd-mult-left-iff*)
  moreover have *degree* $(v'' - v') <$ *degree* v **by** (*rule degree-diff-less*[*OF degv''* *degv'*])
  ultimately have $v'' - v' = 0$
    by (*metis deg-v degree-0 gr-implies-not-zero poly-divides-conv0*)
  then show $v'' = v'$ **by** *auto*
  with $*$ **have** $(w' - w'') * \#v = 0$ **by** *auto*
  with *bv* **have** $w' - w'' = 0$
    by (*metis deg-v degree-0 gr-implies-not-zero mult-eq-0-iff*)
  then show $w'' = w'$ **by** *auto*
qed

lemma *unique*:
  assumes *vV2*: $v = \#V2$ **and** *wW2*: $w = \#W2$ **and** *uVW2*: $u = V2 * W2$
    and *degV2*: *degree* $V2 =$ *degree* v **and** *degW2*: *degree* $W2 =$ *degree* w
    and *lc*: *lead-coeff* $V2 =$ @*lead-coeff* v
  shows $V2 = V$ $W2 = W$
proof−
  from *vV2* **have** $(\#(V2 - \#v) :: 'q$ *mod-ring poly*$) = 0$ **by** (*auto simp*: *hom-distribs*)
  from *rebase-pq-to-q.rebase-poly-eq-0-imp-ex-smult*[*OF this*]
  obtain $v'' :: 'p$ *mod-ring poly*
  where *deg*: *degree* $v'' \leq$ *degree* $(V2 - \#v)$
    and v'': $V2 - \#v =$ *smult* (*of-nat CARD*$('q)$) $(\#v'')$ **by** (*elim exE conjE*)
  then have *V2*: $V2 = \#v + ...$ **by** (*metis add-diff-cancel-left' diff-add-cancel*)

  from *lc*[*unfolded degV2*, *unfolded V2*]
  have *of-nat* $q *$ (@*coeff* v'' (*degree* v) :: $'pq$ *mod-ring*) $=$ *of-nat* $q * 0$ **by** *auto*
  from *this*[*unfolded q rebase-pq-to-p.rebase-mult-eq*]
  have *coeff* v'' (*degree* v) $= 0$ **by** *simp*
  moreover have *degree* $v'' \leq$ *degree* v **using** *deg degV2*
   by (*metis degree-diff-le le-antisym nat-le-linear rebase-q-to-pq.degree-rebase-poly-eq*)
  ultimately have *degv''*: *degree* $v'' <$ *degree* v
    using *bv eq-zero-or-degree-less* **by** *fastforce*

  from *wW2* **have** $(\#(W2 - \#w) :: 'q$ *mod-ring poly*$) = 0$ **by** (*auto simp*: *hom-distribs*)
  from *rebase-pq-to-q.rebase-poly-eq-0-imp-ex-smult*[*OF this*] *pq*
  obtain $w'' :: 'p$ *mod-ring poly* **where** w'': $W2 - \#w =$ *smult* (*of-nat* q) $(\#w'')$ **by** *force*
  then have *W2*: $W2 = \#w + ...$ **by** (*metis add-diff-cancel-left' diff-add-cancel*)

  have $u = \#v * \#w +$ *smult* (*of-nat* q) $(\#w'' * \#v + \#v'' * \#w) +$ *smult* (*of-nat* $(q * q)$) $(\#v'' * \#w'')$
    by(*simp add*: *uVW2 V2 W2 ring-distribs smult-add-right ac-simps*)
  also have *smult* (*of-nat* $(q * q)$) $(\#v'' * \#w'' :: 'pq$ *mod-ring poly*$) = 0$ **by** *simp*
  finally have $u - \#v * \#w =$ *smult* (*of-nat* q) $(\#w'' * \#v + \#v'' * \#w)$ **by** *auto*

also have $u - \#v * \#w = smult\ (of\text{-}nat\ q)\ (\#f)$ **by** (*subst u*, *simp*)
finally have $w'' * \#v + v'' * \#w = f$ **by** (*simp add*: *hom-distribs*)
from *pre-unique*[*OF this degv''*]
have *pre*: $v'' = v'\ w'' = w'$ **by** *auto*
with *V2 W2* **show** $V2 = V\ W2 = W$ **by** *auto*
qed

end

definition
hensel-1 (*ty* ::′*p* :: *prime-card itself*)
(*u* :: ′*pq* :: *nontriv mod-ring poly*) (*v* :: ′*q* :: *nontriv mod-ring poly*) (*w* :: ′*q* *mod-ring poly*) ≡
if $v = 1$ *then* (*1*,*u*) *else*
let (*s*, *t*) = *bezout-coefficients* (#*v* :: ′*p mod-ring poly*) (#*w*) *in*
let (*a*, *b*) = *dupe-monic* (#*v*::′*p mod-ring poly*) (#*w*) *s t 1 in*
(*Knuth-ex-4-6-2-22-main.V TYPE*(′*q*) *b u v w*, *Knuth-ex-4-6-2-22-main.W TYPE*(′*q*) *a b u v w*)

lemma *hensel-1*:
fixes *u* :: ′*pq* :: *nontriv mod-ring poly*
and *v w* :: ′*q* :: *nontriv mod-ring poly*
assumes *CARD*(′*pq*) = *CARD*(′*p* :: *prime-card*) ∗ *CARD*(′*q*)
and *CARD*(′*p*) *dvd CARD*(′*q*)
and *uvw*: #*u* = *v* ∗ *w*
and *degu*: *degree u* = *degree v* + *degree w*
and *monic*: *monic v*
and *coprime*: *coprime* (#*v* :: ′*p mod-ring poly*) (#*w*)
and *out*: *hensel-1 TYPE*(′*p*) *u v w* = (*V*′,*W*′)
shows *u* = *V*′ ∗ *W*′ ∧ *v* = #*V*′ ∧ *w* = #*W*′ ∧ *degree V*′ = *degree v* ∧ *degree* *W*′ = *degree w* ∧
monic V′ ∧ *coprime* (#*V*′ :: ′*p mod-ring poly*) (#*W*′) (**is** *?main*)
and (∀ *V*″ *W*″. *u* = *V*″ ∗ *W*″ ⟶ *v* = #*V*″ ⟶ *w* = #*W*″ ⟶
degree V″ = *degree v* ⟶ *degree W*″ = *degree w* ⟶ *lead-coeff V*″ = @*lead-coeff v* ⟶
V″ = *V*′ ∧ *W*″ = *W*′) (**is** *?unique*)
proof−
from *monic*
have *degv*: *degree* (#*v* :: ′*p mod-ring poly*) = *degree v*
by (*simp add*: *of-int-hom.monic-degree-map-poly-hom*)
from *monic*
have *monic2*: *monic* (#*v* :: ′*p mod-ring poly*)
by (*auto simp*: *degv*)
obtain *s t* **where** *bezout*: *bezout-coefficients* (#*v* :: ′*p mod-ring poly*) (#*w*) = (*s*, *t*)
by (*auto simp add*: *prod-eq-iff*)
then have *s* ∗ #*v* + *t* ∗ #*w* = *gcd* (#*v* :: ′*p mod-ring poly*) (#*w*)
by (*rule bezout-coefficients*)
with *coprime* **have** *vswt*: #*v* ∗ *s* + #*w* ∗ *t* = *1*

```
    by (simp add: ac-simps)
  obtain a b where dupe: dupe-monic (#v) (#w) s t 1 = (a, b) by force
  from dupe-monic(1,2)[OF vswt monic2, where U=1, unfolded this]
  have avbw: a * #v + b * #w = 1 and degb: b = 0 ∨ degree b < degree (#v::'p
mod-ring poly) by auto
  have ?main ∧ ?unique
  proof (cases b = 0)
    case b0: True
    with avbw have a * #v = 1 by auto
    then have degree (#v :: 'p mod-ring poly) = 0
      by (metis degree-1 degree-mult-eq-0 mult-zero-left one-neq-zero)
    from this[unfolded degv] monic-degree-0[OF monic[unfolded]]
    have 1: v = 1 by auto
    with b0 out uvw have 2: V' = 1 W' = u
      by (unfold split hensel-1-def Let-def dupe) auto
    have 3: ?unique apply (simp add: 1 2) by (metis monic-degree-0 mult.left-neutral)
    with uvw degu show ?thesis unfolding 1 2 by auto
  next
    case b0: False
    with degb degv have degb: degree b < degree v by auto
    then have v1: v ≠ 1 by auto
    interpret Knuth-ex-4-6-2-22-prime TYPE('p) TYPE('q) TYPE('pq) a b
      by (unfold-locales; fact assms degb avbw)
    show ?thesis
    proof (intro conjI)
      from out [unfolded hensel-1-def] v1
      have 1 [simp]: V' = V W' = W by (auto simp: bezout dupe)
      from uVW show u = V' * W' by auto
      from degV show [simp]: degree V' = degree v by simp
      from degW show [simp]: degree W' = degree w by simp
      from lcV have lead-coeff V' = @lead-coeff v by simp
      with monic-v show monic V' by (simp add:)
      from vV show v = #V' by simp
      from wW show w = #W' by simp
      from coprime-preserves show coprime (#V' :: 'p mod-ring poly) (#W') by
simp
      show 9: ?unique by (unfold 1, intro allI conjI impI; rule unique)
    qed
  qed
  then show ?main ?unique by (fact conjunct1, fact conjunct2)
qed

end
```

9.3 Result is Unique

We combine the finite field factorization algorithm with Hensel-lifting to obtain factorizations mod p^n. Moreover, we prove results on unique-factorizations in mod p^n which admit to extend the uniqueness result for binary Hensel-lifting to the general case. As a consequence, our factorization algorithm will produce unique factorizations mod p^n.

theory *Berlekamp-Hensel*
imports
  Finite-Field-Factorization-Record-Based
  Hensel-Lifting
begin

hide-const *coeff monom*

definition *berlekamp-hensel* :: *int* ⇒ *nat* ⇒ *int poly* ⇒ *int poly list* **where**
  berlekamp-hensel p n f = (*case finite-field-factorization-int p f of*
    (-,*fs*) ⇒ *hensel-lifting p n f fs*)

Finite field factorization in combination with Hensel-lifting delivers factorization modulo p^k where factors are irreducible modulo p. Assumptions: input polynomial is square-free modulo p.

context *poly-mod-prime* **begin**

lemma *berlekamp-hensel-main*:
  assumes *n*: *n* ≠ *0*
    and *res*: *berlekamp-hensel p n f* = *gs*
    and *cop*: *coprime* (*lead-coeff f*) *p*
    and *sf*: *square-free-m f*
    and *berl*: *finite-field-factorization-int p f* = (*c*,*fs*)
  shows *poly-mod.factorization-m* (*p* ^ *n*) *f* (*lead-coeff f*, *mset gs*) — factorization mod *p*^*n*
    and *sort* (*map degree fs*) = *sort* (*map degree gs*)
    and ⋀ *g*. *g* ∈ *set gs* ⟹ *monic g* ∧ *poly-mod.Mp* (*p*^*n*) *g* = *g* ∧ — monic and normalized
      poly-mod.irreducible-m p g ∧ — irreducibility even mod *p*
      poly-mod.degree-m p g = *degree g* — mod *p* does not change degree of *g*
proof −
  from *res*[*unfolded berlekamp-hensel-def berl split*]
  have *hen*: *hensel-lifting p n f fs* = *gs* **.**
  note *bh* = *finite-field-factorization-int*[*OF sf berl*]
  from *bh* **have** *poly-mod.factorization-m p f* (*c*, *mset fs*) *c* ∈ {*0*..<*p*} (∀ *fi*∈*set fs*. *set* (*coeffs fi*) ⊆ {*0*..<*p*})
    by (*auto simp*: *poly-mod.unique-factorization-m-alt-def*)
  note *hen* = *hensel-lifting*[*OF n hen cop sf*, *OF this*]
  show *poly-mod.factorization-m* (*p* ^ *n*) *f* (*lead-coeff f*, *mset gs*)
    sort (*map degree fs*) = *sort* (*map degree gs*)
    ⋀ *g*. *g* ∈ *set gs* ⟹ *monic g* ∧ *poly-mod.Mp* (*p*^*n*) *g* = *g* ∧
      poly-mod.irreducible-m p g ∧
      poly-mod.degree-m p g = *degree g* **using** *hen* **by** *auto*
qed

theorem *berlekamp-hensel*:
  assumes *cop*: *coprime* (*lead-coeff f*) *p*

and *sf*: *square-free-m f*
and *res*: *berlekamp-hensel p n f* = *gs*
and *n*: *n* ≠ *0*
shows *poly-mod.factorization-m* (*p*^*n*) *f* (*lead-coeff f*, *mset gs*) — factorization mod *p*^*n*
and ⋀ *g*. *g* ∈ *set gs* ⟹ *poly-mod.Mp* (*p*^*n*) *g* = *g* ∧ *poly-mod.irreducible-m p g*
— normalized and *irreducible* even mod *p*
proof −
obtain *c fs* **where** *finite-field-factorization-int p f* = (*c*,*fs*) **by** *force*
from *berlekamp-hensel-main*[*OF n res cop sf this*]
show *poly-mod.factorization-m* (*p*^*n*) *f* (*lead-coeff f*, *mset gs*)
⋀ *g*. *g* ∈ *set gs* ⟹ *poly-mod.Mp* (*p*^*n*) *g* = *g* ∧ *poly-mod.irreducible-m p g* **by** *auto*
qed

lemma *berlekamp-and-hensel-separated*:
assumes *cop*: *coprime* (*lead-coeff f*) *p*
and *sf*: *square-free-m f*
and *res*: *hensel-lifting p n f fs* = *gs*
and *berl*: *finite-field-factorization-int p f* = (*c*,*fs*)
and *n*: *n* ≠ *0*
shows *berlekamp-hensel p n f* = *gs*
and *sort* (*map degree fs*) = *sort* (*map degree gs*)
proof −
show *berlekamp-hensel p n f* = *gs* **unfolding** *res*[*symmetric*]
berlekamp-hensel-def hensel-lifting-def berl split Let-def **..**
from *berlekamp-hensel-main*[*OF n this cop sf berl*] **show** *sort* (*map degree fs*) = *sort* (*map degree gs*)
by *auto*
qed

end

lemma *prime-cop-exp-poly-mod*:
assumes *prime*: *prime p* **and** *cop*: *coprime c p* **and** *n*: *n* ≠ *0*
shows *poly-mod.M* (*p*^*n*) *c* ∈ {*1* ..< *p*^*n*}
proof −
from *prime* **have** *p1*: *p* > *1* **by** (*simp add*: *prime-int-iff*)
interpret *poly-mod-2 p*^*n* **unfolding** *poly-mod-2-def* **using** *p1 n* **by** *simp*
from *cop p1 m1* **have** *M c* ≠ *0*
by (*auto simp add*: *M-def*)
moreover have *M c* < *p*^*n M c* ≥ *0* **unfolding** *M-def* **using** *m1* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

context *poly-mod-2*
begin

context
fixes *p* :: *int*
assumes *prime*: *prime p*
begin

interpretation *p*: *poly-mod-prime p* **using** *prime* **by** *unfold-locales*

lemma *coprime-lead-coeff-factor*: **assumes** *coprime* (*lead-coeff* (*f* ∗ *g*)) *p*
shows *coprime* (*lead-coeff f*) *p coprime* (*lead-coeff g*) *p*
proof −
{
fix *f g*
assume *cop*: *coprime* (*lead-coeff* (*f* ∗ *g*)) *p*
from *this*[*unfolded lead-coeff-mult*]
have *coprime* (*lead-coeff f*) *p* **using** *prime*
by *simp*
}
from *this*[*OF assms*] *this*[*of g f*] *assms*
show *coprime* (*lead-coeff f*) *p coprime* (*lead-coeff g*) *p* **by** (*auto simp*: *ac-simps*)
qed

lemma *unique-factorization-m-factor*: **assumes** *uf*: *unique-factorization-m* (*f* ∗ *g*) (*c*,*hs*)
and *cop*: *coprime* (*lead-coeff* (*f* ∗ *g*)) *p*
and *sf*: *p.square-free-m* (*f* ∗ *g*)
and *n*: $n \neq 0$
and *m*: $m = p\hat{\ }n$
shows ∃ *fs gs*. *unique-factorization-m f* (*lead-coeff f*,*fs*)
∧ *unique-factorization-m g* (*lead-coeff g*,*gs*)
∧ *Mf* (*c*,*hs*) = *Mf* (*lead-coeff f* ∗ *lead-coeff g*, *fs* + *gs*)
∧ *image-mset Mp fs* = *fs* ∧ *image-mset Mp gs* = *gs*
proof −
from *prime* **have** *p1*: *1* < *p* **by** (*simp add*: *prime-int-iff*)
interpret *p*: *poly-mod-2 p* **by** (*standard*, *rule p1*)
note *sf* = *p.square-free-m-factor*[*OF sf*]
note *cop* = *coprime-lead-coeff-factor*[*OF cop*]
from *cop* **have** *copm*: *coprime* (*lead-coeff f*) *m coprime* (*lead-coeff g*) *m*
by (*simp-all add*: *m*)
have *df*: *degree-m f* = *degree f*
by (*rule degree-m-eq*[*OF - m1*], *insert copm*(*1*) *m1*, *auto*)
have *dg*: *degree-m g* = *degree g*
by (*rule degree-m-eq*[*OF - m1*], *insert copm*(*2*) *m1*, *auto*)
define *fs* **where** *fs* ≡ *mset* (*berlekamp-hensel p n f*)
define *gs* **where** *gs* ≡ *mset* (*berlekamp-hensel p n g*)
from *p.berlekamp-hensel*[*OF cop*(*1*) *sf*(*1*) *refl n*, *folded m*]
have *f*: *factorization-m f* (*lead-coeff f*,*fs*)
and *f-id*: ⋀ *f*. *f* ∈# *fs* ⟹ *Mp f* = *f* **unfolding** *fs-def* **by** *auto*
from *p.berlekamp-hensel*[*OF cop*(*2*) *sf*(*2*) *refl n*, *folded m*]
have *g*: *factorization-m g* (*lead-coeff g*,*gs*)

and *g-id*: $\bigwedge f.\ f \in\#\ gs \Longrightarrow Mp\ f = f$ **unfolding** *gs-def* **by** *auto*
from *factorization-m-prod*[*OF f g*] *uf*[*unfolded unique-factorization-m-alt-def*]
have *eq*: *Mf* (*lead-coeff f* ∗ *lead-coeff g*, *fs* + *gs*) = *Mf* (*c*,*hs*) **by** *blast*
have *uff*: *unique-factorization-m f* (*lead-coeff f*,*fs*)
proof (*rule unique-factorization-mI*[*OF f*])
fix *e ks*
assume *factorization-m f* (*e*,*ks*)
from *factorization-m-prod*[*OF this g*] *uf*[*unfolded unique-factorization-m-alt-def*]
factorization-m-lead-coeff[*OF this*, *unfolded degree-m-eq-lead-coeff*[*OF df*]]
have *Mf* (*e* ∗ *lead-coeff g*, *ks* + *gs*) = *Mf* (*c*,*hs*) **and** *e*: *M* (*lead-coeff f*) = *M e* **by** *blast*+
from *this*[*folded eq*, *unfolded Mf-def split*]
have *ks*: *image-mset Mp ks* = *image-mset Mp fs* **by** *auto*
show *Mf* (*e*, *ks*) = *Mf* (*lead-coeff f*, *fs*) **unfolding** *Mf-def split ks e* **by** *simp*
qed
have *idf*: *image-mset Mp fs* = *fs* **using** *f-id* **by** (*induct fs*, *auto*)
have *idg*: *image-mset Mp gs* = *gs* **using** *g-id* **by** (*induct gs*, *auto*)
have *ufg*: *unique-factorization-m g* (*lead-coeff g*,*gs*)
proof (*rule unique-factorization-mI*[*OF g*])
fix *e ks*
assume *factorization-m g* (*e*,*ks*)
from *factorization-m-prod*[*OF f this*] *uf*[*unfolded unique-factorization-m-alt-def*]
factorization-m-lead-coeff[*OF this*, *unfolded degree-m-eq-lead-coeff*[*OF dg*]]
have *Mf* (*lead-coeff f* ∗ *e*, *fs* + *ks*) = *Mf* (*c*,*hs*) **and** *e*: *M* (*lead-coeff g*) = *M e* **by** *blast*+
from *this*[*folded eq*, *unfolded Mf-def split*]
have *ks*: *image-mset Mp ks* = *image-mset Mp gs* **by** *auto*
show *Mf* (*e*, *ks*) = *Mf* (*lead-coeff g*, *gs*) **unfolding** *Mf-def split ks e* **by** *simp*
qed
from *uff ufg eq*[*symmetric*] *idf idg* **show** *?thesis* **by** *auto*
qed

lemma *unique-factorization-factorI*:
assumes *ufact*: *unique-factorization-m* (*f* ∗ *g*) *FG*
and *cop*: *coprime* (*lead-coeff* (*f* ∗ *g*)) *p*
and *sf*: *poly-mod.square-free-m p* (*f* ∗ *g*)
and *n*: $n \neq 0$
and *m*: $m = p\widehat{\ }n$
shows *factorization-m f F* $\Longrightarrow$ *unique-factorization-m f F*
and *factorization-m g G* $\Longrightarrow$ *unique-factorization-m g G*
proof −
obtain *c fg* **where** *FG*: *FG* = (*c*,*fg*) **by** *force*
from *unique-factorization-m-factor*[*OF ufact*[*unfolded FG*] *cop sf n m*]
obtain *fs gs* **where** *ufact*: *unique-factorization-m f* (*lead-coeff f*, *fs*)
unique-factorization-m g (*lead-coeff g*, *gs*) **by** *auto*
from *ufact*(*1*) **show** *factorization-m f F* $\Longrightarrow$ *unique-factorization-m f F*
by (*metis unique-factorization-m-alt-def*)
from *ufact*(*2*) **show** *factorization-m g G* $\Longrightarrow$ *unique-factorization-m g G*
by (*metis unique-factorization-m-alt-def*)

qed

end

lemma *monic-Mp-prod-mset*: **assumes** *fs*: $\bigwedge f.\ f \in\# fs \Longrightarrow$ *monic* (*Mp f*)
  shows *monic* (*Mp* (*prod-mset fs*))
proof −
  have *monic* (*prod-mset* (*image-mset Mp fs*))
    by (*rule monic-prod-mset*, *insert fs*, *auto*)
  from *monic-Mp*[*OF this*] **have** *monic* (*Mp* (*prod-mset* (*image-mset Mp fs*))) **.**
  also have *Mp* (*prod-mset* (*image-mset Mp fs*)) = *Mp* (*prod-mset fs*) **by** (*rule Mp-prod-mset*)
  finally show *?thesis* **.**
qed

lemma *degree-Mp-mult-monic*: **assumes** *monic f monic g*
  shows *degree* (*Mp* ($f * g$)) = *degree f* + *degree g*
  by (*metis zero-neq-one assms degree-monic-mult leading-coeff-0-iff monic-degree-m monic-mult*)

lemma *factorization-m-degree*: **assumes** *factorization-m f* (*c*,*fs*)
  and *0*: *Mp f* $\neq$ *0*
  shows *degree-m f* = *sum-mset* (*image-mset degree-m fs*)
proof −
  note *a* = *assms*[*unfolded factorization-m-def split*]
  hence *deg*: *degree-m f* = *degree-m* (*smult c* (*prod-mset fs*))
    and *fs*: $\bigwedge f.\ f \in\# fs \Longrightarrow$ *monic* (*Mp f*) **by** *auto*
  define *gs* **where** *gs* $\equiv$ *Mp* (*prod-mset fs*)
  from *monic-Mp-prod-mset*[*OF fs*] **have** *mon-gs*: *monic gs* **unfolding** *gs-def* **.**
  have *d*:*degree* (*Mp* (*Polynomial.smult c gs*)) = *degree gs*
  proof −
    have *f1*: *0* $\neq$ *c* **by** (*metis 0 Mp-0 a*(*1*) *smult-eq-0-iff*)
    then have *M c* $\neq$ *0* **by** (*metis* (*no-types*) *0 assms*(*1*) *factorization-m-lead-coeff leading-coeff-0-iff*)
    then show *degree* (*Mp* (*Polynomial.smult c gs*)) = *degree gs*
      unfolding *monic-degree-m*[*OF mon-gs*,*symmetric*]
    using *f1* **by** (*metis coeff-smult degree-m-eq degree-smult-eq m1 mon-gs monic-degree-m mult-cancel-left1 poly-mod.M-def*)
  qed
  note *deg*
  also have *degree-m* (*smult c* (*prod-mset fs*)) = *degree-m* (*smult c gs*)
    unfolding *gs-def* **by** *simp*
  also have ... = *degree gs* **using** *d***.**
  also have ... = *sum-mset* (*image-mset degree-m fs*) **unfolding** *gs-def*
    using *fs*
  proof (*induct fs*)
    case (*add f fs*)
  have *mon*: *monic* (*Mp f*) *monic* (*Mp* (*prod-mset fs*)) **using** *monic-Mp-prod-mset*[*of fs*]

add(*2*) **by** *auto*
have *degree* (*Mp* (*prod-mset* (*add-mset f fs*))) = *degree* (*Mp* (*Mp f* ∗ *Mp* (*prod-mset fs*)))
by (*auto simp*: *ac-simps*)
also have ... = *degree* (*Mp f*) + *degree* (*Mp* (*prod-mset fs*))
by (*rule degree-Mp-mult-monic*[*OF mon*])
also have *degree* (*Mp* (*prod-mset fs*)) = *sum-mset* (*image-mset degree-m fs*)
by (*rule add*(*1*), *insert add*(*2*), *auto*)
finally show *?case* **by** (*simp add*: *ac-simps*)
qed *simp*
finally show *?thesis* **.**
qed

lemma *degree-m-mult-le*: *degree-m* (*f* ∗ *g*) ≤ *degree-m f* + *degree-m g*
using *degree-m-mult-le* **by** *auto*

lemma *degree-m-prod-mset-le*: *degree-m* (*prod-mset fs*) ≤ *sum-mset* (*image-mset degree-m fs*)
proof (*induct fs*)
case *empty*
show *?case* **by** *simp*
next
case (*add f fs*)
then show *?case* **using** *degree-m-mult-le*[*of f prod-mset fs*] **by** *auto*
qed

end

context *poly-mod-prime*
begin

lemma *unique-factorization-m-factor-partition*: **assumes** *l0*: $l \neq 0$
and *uf*: *poly-mod.unique-factorization-m* (*p*^*l*) *f* (*lead-coeff f*, *mset gs*)
and *f*: *f* = *f1* ∗ *f2*
and *cop*: *coprime* (*lead-coeff f*) *p*
and *sf*: *square-free-m f*
and *part*: *List.partition* (λ*gi*. *gi dvdm f1*) *gs* = (*gs1*, *gs2*)
shows *poly-mod.unique-factorization-m* (*p*^*l*) *f1* (*lead-coeff f1*, *mset gs1*)
poly-mod.unique-factorization-m (*p*^*l*) *f2* (*lead-coeff f2*, *mset gs2*)
proof −
interpret *pl*: *poly-mod-2 p*^*l* **by** (*standard*, *insert m1 l0*, *auto*)
let *?I* = *image-mset pl.Mp*
note *Mp-pow* [*simp*] = *Mp-Mp-pow-is-Mp*[*OF l0 m1*]
have [*simp*]: *pl.Mp x dvdm u* = (*x dvdm u*) **for** *x u* **unfolding** *dvdm-def* **using** *Mp-pow*[*of x*]
by (*metis poly-mod.mult-Mp*(*1*))
have *gs-split*: *set gs* = *set gs1* ∪ *set gs2* **using** *part* **by** *auto*
from *pl.unique-factorization-m-factor*[*OF prime uf*[*unfolded f*] - - *l0 refl*, *folded*

f, *OF cop sf*]
  obtain *hs1 hs2* **where** *uf'*: *pl.unique-factorization-m f1* (*lead-coeff f1*, *hs1*)
      pl.unique-factorization-m f2 (*lead-coeff f2*, *hs2*)
    and *gs-hs*: *?I* (*mset gs*) = *hs1* + *hs2*
    unfolding *pl.Mf-def split* **by** *auto*
  have *gs-gs*: *?I* (*mset gs*) = *?I* (*mset gs1*) + *?I* (*mset gs2*) **using** *part*
    by (*auto*, *induct gs arbitrary*: *gs1 gs2*, *auto*)
  with *gs-hs* **have** *gs-hs12*: *?I* (*mset gs1*) + *?I* (*mset gs2*) = *hs1* + *hs2* **by** *auto*
  note *pl-dvdm-imp-p-dvdm* = *pl-dvdm-imp-p-dvdm*[*OF l0*]
  note *fact* = *pl.unique-factorization-m-imp-factorization*[*OF uf*]
  have *gs1*: *?I* (*mset gs1*) = {#*x* ∈# *?I* (*mset gs*). *x dvdm f1*#}
    using *part* **by** (*auto*, *induct gs arbitrary*: *gs1 gs2*, *auto*)
  also have ... = {#*x* ∈# *hs1*. *x dvdm f1*#} + {#*x* ∈# *hs2*. *x dvdm f1*#}
unfolding *gs-hs* **by** *simp*
  also have {#*x* ∈# *hs2*. *x dvdm f1*#} = {#}
  proof (*rule ccontr*)
    assume ¬ *?thesis*
    then obtain *x* **where** *x*: *x* ∈# *hs2* **and** *dvd*: *x dvdm f1* **by** *fastforce*
    from *x gs-hs* **have** *x* ∈# *?I* (*mset gs*) **by** *auto*
    with *fact*[*unfolded pl.factorization-m-def*]
    have *xx*: *pl.irreducible$_d$-m x monic x* **by** *auto*
    from *square-free-m-prod-imp-coprime-m*[*OF sf*[*unfolded f*]]
    have *cop-h-f*: *coprime-m f1 f2* **by** *auto*
   from *pl.factorization-m-mem-dvdm*[*OF pl.unique-factorization-m-imp-factorization*[*OF uf'*(*2*)], *of x*] *x*
    have *pl.dvdm x f2* **by** *auto*
    hence *x dvdm f2* **by** (*rule pl-dvdm-imp-p-dvdm*)
    from *cop-h-f*[*unfolded coprime-m-def*, *rule-format*, *OF dvd this*]
    have *x dvdm 1* **by** *auto*
    from *dvdm-imp-degree-le*[*OF this xx*(*2*) - *m1*] **have** *degree x* = *0* **by** *auto*
    with *xx* **show** *False* **unfolding** *pl.irreducible$_d$-m-def* **by** *auto*
  qed
  also have {#*x* ∈# *hs1*. *x dvdm f1*#} = *hs1*
  proof (*rule ccontr*)
    assume ¬ *?thesis*
    from *filter-mset-inequality*[*OF this*]
    obtain *x* **where** *x*: *x* ∈# *hs1* **and** *dvd*: ¬ *x dvdm f1* **by** *blast*
   from *pl.factorization-m-mem-dvdm*[*OF pl.unique-factorization-m-imp-factorization*[*OF uf'*(*1*)],
      of x] *x dvd*
    have *pl.dvdm x f1* **by** *auto*
    from *pl-dvdm-imp-p-dvdm*[*OF this*] *dvd* **show** *False* **by** *auto*
  qed
  finally have *gs-hs1*: *?I* (*mset gs1*) = *hs1* **by** *simp*
  with *gs-hs12* **have** *?I* (*mset gs2*) = *hs2* **by** *auto*
  with *uf' gs-hs1* **have** *pl.unique-factorization-m f1* (*lead-coeff f1*, *?I* (*mset gs1*))
    pl.unique-factorization-m f2 (*lead-coeff f2*, *?I* (*mset gs2*)) **by** *auto*
  thus *pl.unique-factorization-m f1* (*lead-coeff f1*, *mset gs1*)
    pl.unique-factorization-m f2 (*lead-coeff f2*, *mset gs2*)

unfolding *pl.unique-factorization-m-def*
by (*auto simp*: *pl.Mf-def image-mset.compositionality o-def*)
qed

lemma *factorization-pn-to-factorization-p*: **assumes** *fact*: *poly-mod.factorization-m* (p^n) *C* (*c*,*fs*)
and *sf*: *square-free-m C*
and *n*: $n \neq 0$
shows *factorization-m C* (*c*,*fs*)
proof −
let $?q = p^n$
from *n m1* **have** *q*: $?q > 1$ **by** *simp*
interpret *q*: *poly-mod-2 ?q* **by** (*standard*, *insert q*, *auto*)
from *fact*[*unfolded q.factorization-m-def*]
have *eq*: *q.Mp C* = *q.Mp* (*Polynomial.smult c* (*prod-mset fs*))
and *irr*: $\bigwedge$ *f*. *f* ∈# *fs* ⟹ *q.irreducible*$_d$*-m f*
and *mon*: $\bigwedge$ *f*. *f* ∈# *fs* ⟹ *monic* (*q.Mp f*)
by *auto*
from *arg-cong*[*OF eq*, *of Mp*]
have *eq*: *eq-m C* (*smult c* (*prod-mset fs*))
by (*simp add*: *Mp-Mp-pow-is-Mp m1 n*)
show *?thesis* **unfolding** *factorization-m-def split*
proof (*rule conjI*[*OF eq*], *intro ballI conjI*)
fix *f*
assume *f*: *f* ∈# *fs*
from *mon*[*OF this*] **have** *mon-qf*: *monic* (*q.Mp f*) **.**
hence *lc*: *lead-coeff* (*q.Mp f*) = *1* **by** *auto*
from *mon-qf* **show** *mon-f*: *monic* (*Mp f*)
by (*metis Mp-Mp-pow-is-Mp m1 monic-Mp n*)
from *irr*[*OF f*] **have** *irr*: *q.irreducible*$_d$*-m f* **.**
hence *q.degree-m f* ≠ *0* **unfolding** *q.irreducible*$_d$*-m-def* **by** *auto*
also have *q.degree-m f* = *degree-m f* **using** *mon*[*OF f*]
by (*metis Mp-Mp-pow-is-Mp m1 monic-degree-m n*)
finally have *deg*: *degree-m f* ≠ *0* **by** *auto*
from *f* **obtain** *gs* **where** *fs*: *fs* = {#*f*#} + *gs*
by (*metis mset-subset-eq-single subset-mset.add-diff-inverse*)
from *eq*[*unfolded fs*] **have** *Mp C* = *Mp* (*f* ∗ *smult c* (*prod-mset gs*)) **by** *auto*
from *square-free-m-factor*[*OF square-free-m-cong*[*OF sf this*]]
have *sf-f*: *square-free-m f* **by** *simp*
have *sf-Mf*: *square-free-m* (*q.Mp f*)
by (*rule square-free-m-cong*[*OF sf-f*], *auto simp*: *Mp-Mp-pow-is-Mp n m1*)
have *coprime* (*lead-coeff* (*q.Mp f*)) *p* **using** *mon*[*OF f*] *prime* **by** *simp*
from *berlekamp-hensel*[*OF this sf-Mf refl n*, *unfolded lc*] **obtain** *gs* **where**
qfact: *q.factorization-m* (*q.Mp f*) (*1*, *mset gs*)
and $\bigwedge$ *g*. *g* ∈ *set gs* ⟹ *irreducible-m g* **by** *blast*
hence *fact*: *q.Mp f* = *q.Mp* (*prod-list gs*)
and *gs*: $\bigwedge$ *g*. *g*∈ *set gs* ⟹ *irreducible*$_d$*-m g* ∧ *q.irreducible*$_d$*-m g* ∧ *monic* (*q.Mp g*)
unfolding *q.factorization-m-def* **by** *auto*

from *q.factorization-m-degree*[*OF qfact*]
have *deg*: *q.degree-m* (*q.Mp f*) = *sum-mset* (*image-mset q.degree-m* (*mset gs*))
using *mon-qf* **by** *fastforce*
from *irr*[*unfolded q.irreducible*$_d$*-m-def*]
have *sum-mset* (*image-mset q.degree-m* (*mset gs*)) $\neq$ *0* **by** (*fold deg*, *auto*)
then obtain *g gs'* **where** *gs1*: *gs* = *g* # *gs'* **by** (*cases gs*, *auto*)
{
assume *gs'* $\neq$ []
then obtain *h hs* **where** *gs2*: *gs'* = *h* # *hs* **by** (*cases gs'*, *auto*)
from *deg gs*[*unfolded q.irreducible*$_d$*-m-def*]
have *small*: *q.degree-m g* < *q.degree-m f*
q.degree-m h + *sum-mset* (*image-mset q.degree-m* (*mset hs*)) < *q.degree-m f*
unfolding *gs1 gs2* **by** *auto*
have *q.eq-m f* (*g* ∗ (*h* ∗ *prod-list hs*))
using *fact* **unfolding** *gs1 gs2* **by** *simp*
with *irr*[*unfolded q.irreducible*$_d$*-m-def*, *THEN conjunct2*, *rule-format*, *of g h* ∗ *prod-list hs*]
small(*1*) **have** ¬ *q.degree-m* (*h* ∗ *prod-list hs*) < *q.degree-m f* **by** *auto*
hence *q.degree-m f* ≤ *q.degree-m* (*h* ∗ *prod-list hs*) **by** *simp*
also have ... = *q.degree-m* (*prod-mset* ({#*h*#} + *mset hs*)) **by** *simp*
also have ... ≤ *sum-mset* (*image-mset q.degree-m* ({#*h*#} + *mset hs*))
by (*rule q.degree-m-prod-mset-le*)
also have ... < *q.degree-m f* **using** *small*(*2*) **by** *simp*
finally have *False* **by** *simp*
}
hence *gs1*: *gs* = [*g*] **unfolding** *gs1* **by** (*cases gs'*, *auto*)
with *fact* **have** *q.Mp f* = *q.Mp g* **by** *auto*
from *arg-cong*[*OF this*, *of Mp*] **have** *eq*: *Mp f* = *Mp g*
by (*simp add*: *Mp-Mp-pow-is-Mp m1 n*)
from *gs*[*unfolded gs1*] **have** *g*: *irreducible*$_d$*-m g* **by** *auto*
with *eq* **show** *irreducible*$_d$*-m f* **unfolding** *irreducible*$_d$*-m-def* **by** *auto*
qed
qed

lemma *unique-monic-hensel-factorization*:
assumes *ufact*: *unique-factorization-m C* (*1*,*Fs*)
and *C*: *monic C square-free-m C*
and *n*: *n* $\neq$ *0*
shows ∃ *Gs*. *poly-mod.unique-factorization-m* (*p*^*n*) *C* (*1*, *Gs*)
using *ufact C*
proof (*induct Fs arbitrary*: *C rule*: *wf-induct*[*OF wf-measure*[*of size*]])
case (*1 Fs C*)
let *?q* = *p*^*n*
from *n m1* **have** *q*: *?q* > *1* **by** *simp*
interpret *q*: *poly-mod-2 ?q* **by** (*standard*, *insert q*, *auto*)
note [*simp*] = *Mp-Mp-pow-is-Mp*[*OF n m1*]
note *IH* = *1*(*1*)[*rule-format*]
note *ufact* = *1*(*2*)

hence *fact*: *factorization-m C* (*1*, *Fs*) **unfolding** *unique-factorization-m-alt-def* **by** *auto*
  note *monC* = *1*(*3*)
  note *sf* = *1*(*4*)
  let *?n* = *size Fs*
  {
    fix *d gs*
    assume *qfact*: *q.factorization-m C* (*d*,*gs*)
    from *q.factorization-m-lead-coeff*[*OF this*] *q.monic-Mp*[*OF monC*]
    have *d1*: *q.M d* = *1* **by** *auto*

    from *factorization-pn-to-factorization-p*[*OF qfact sf n*]
    have *factorization-m C* (*d*,*gs*) **.**
    with *ufact d1* **have** *q.M d* = *1 M d* = *1 image-mset Mp gs* = *image-mset Mp Fs*
      unfolding *unique-factorization-m-alt-def Mf-def* **by** *auto*
  } **note** *pre-unique* = *this*
  show *?case*
  proof (*cases Fs*)
    case *empty*
    with *fact C* **have** *Mp C* = *1* **unfolding** *factorization-m-def* **by** *auto*
    hence *degree* (*Mp C*) = *0* **by** *simp*
    with *degree-m-eq-monic*[*OF monC m1*] **have** *degree C* = *0* **by** *simp*
    with *monC* **have** *C1*: *C* = *1* **using** *monic-degree-0* **by** *blast*
    with *fact* **have** *fact*: *q.factorization-m C* (*1*,{#})
      by (*auto simp*: *q.factorization-m-def*)
    show *?thesis*
    proof (*rule exI*, *rule q.unique-factorization-mI*[*OF fact*])
      fix *d gs*
      assume *fact*: *q.factorization-m C* (*d*,*gs*)
      from *pre-unique*[*OF this*, *unfolded empty*]
      show *q.Mf* (*d*, *gs*) = *q.Mf* (*1*, {#}) **by** (*auto simp*: *q.Mf-def*)
    qed
  next
    case (*add D H*) **note** *FDH* = *this*
    let *?D* = *Mp D*
    let *?H* = *Mp* (*prod-mset H*)
    from *fact* **have** *monFs*: $\bigwedge$ *F*. *F* ∈# *Fs* ⟹ *monic* (*Mp F*)
      and *prod*: *eq-m C* (*prod-mset Fs*) **unfolding** *factorization-m-def* **by** *auto*
    hence *monD*: *monic ?D* **unfolding** *FDH* **by** *auto*
    from *square-free-m-cong*[*OF sf*, *of D* ∗ *prod-mset H*] *prod*[*unfolded FDH*]
    have *square-free-m* (*D* ∗ *prod-mset H*) **by** (*auto simp*: *ac-simps*)
    from *square-free-m-prod-imp-coprime-m*[*OF this*]
    have *coprime-m D* (*prod-mset H*) **.**
    hence *cop'*: *coprime-m ?D ?H* **unfolding** *coprime-m-def dvdm-def Mp-Mp* **by** *simp*
    from *fact* **have** *eq'*: *eq-m* (*?D* ∗ *?H*) *C*
      unfolding *FDH* **by** (*simp add*: *factorization-m-def ac-simps*)
    note *unique-hensel-binary*[*OF prime cop' eq' Mp-Mp Mp-Mp monD n*]

from *ex1-implies-ex*[*OF this*] *this*
obtain *A B* **where** *CAB*: *q.eq-m* $(A * B)$ *C* **and** *monA*: *monic A* **and** *DA*: *eq-m ?D A*
and *HB*: *eq-m ?H B* **and** *norm*: *q.Mp A* = *A q.Mp B* = *B*
and *unique*: $\bigwedge$ D' H'. *q.eq-m* $(D' * H')$ *C* $\Longrightarrow$
monic D' $\Longrightarrow$
eq-m (*Mp D*) D' $\Longrightarrow$ *eq-m* (*Mp* (*prod-mset H*)) H' $\Longrightarrow$ *q.Mp* $D' = D'$ $\Longrightarrow$ *q.Mp* $H' = H'$
$\Longrightarrow$ $D' = A \wedge H' = B$ **by** *blast*
note *hensel-bin-wit* = *CAB monA DA HB norm*
from *monA* **have** *monA′*: *monic* (*q.Mp A*) **by** (*rule q.monic-Mp*)
from *q.monic-Mp*[*OF monC*] *CAB* **have** *monicP*:*monic* (*q.Mp* $(A * B)$) **by** *auto*
have *f4*: $\bigwedge$*p*. *coeff* $(A * p)$ (*degree* $(A * p)$) = *coeff p* (*degree p*)
by (*simp add*: *coeff-degree-mult monA*)
have *f2*: $\bigwedge$*p n i*. *coeff p n mod i* = *coeff* (*poly-mod.Mp i p*) *n*
using *poly-mod.M-def poly-mod.Mp-coeff* **by** *presburger*
hence *coeff B* (*degree B*) = *0* $\vee$ *monic B*
using *monicP f4* **by** (*metis* (*no-types*) *norm*(*2*) *q.degree-m-eq q.m1*)
hence *monB*: *monic B*
using *f4 monicP* **by** (*metis norm*(*2*) *leading-coeff-0-iff*)
from *monA monB* **have** *lcAB*: *lead-coeff* $(A * B)$ = *1* **by** (*rule monic-mult*)
hence *copAB*: *coprime* (*lead-coeff* $(A * B)$) *p* **by** *auto*
from *arg-cong*[*OF CAB*, *of Mp*]
have *CAB′*: *eq-m C* $(A * B)$ **by** *auto*
from *sf CAB′* **have** *sfAB*: *square-free-m* $(A * B)$ **using** *square-free-m-cong* **by** *blast*
from *CAB′ ufact* **have** *ufact*: *unique-factorization-m* $(A * B)$ (*1*, *Fs*)
using *unique-factorization-m-cong* **by** *blast*
have (*1* :: *nat*) $\neq$ *0 p* = *p* ^ *1* **by** *auto*
note *u-factor* = *unique-factorization-factorI*[*OF prime ufact copAB sfAB this*]
from *fact DA* **have** *irreducible*$_d$*-m D eq-m A D* **unfolding** *add factorization-m-def* **by** *auto*
hence *irreducible*$_d$*-m A* **using** *Mp-irreducible*$_d$*-m* **by** *fastforce*
from *irreducible*$_d$*-lifting*[*OF n - this*] **have** *irrA*: *q.irreducible*$_d$*-m A* **using** *monA*
by (*simp add*: *m1 poly-mod.degree-m-eq-monic q.m1*)

from *add* **have** *lenH*: (*H*,*Fs*) $\in$ *measure size* **by** *auto*
from *HB fact* **have** *factB*: *factorization-m B* (*1*, *H*)
unfolding *FDH factorization-m-def* **by** *auto*
from *u-factor*(*2*)[*OF factB*] **have** *ufactB*: *unique-factorization-m B* (*1*, *H*) **.**

from *sfAB* **have** *sfB*: *square-free-m B* **by** (*rule square-free-m-factor*)
from *IH*[*OF lenH ufactB monB sfB*] **obtain** *Bs* **where**
IH2: *q.unique-factorization-m B* (*1*, *Bs*) **by** *auto*

from *CAB* **have** *q.Mp C* = *q.Mp* (*q.Mp A* $*$ *q.Mp B*) **by** *simp*
also have *q.Mp A* $*$ *q.Mp B* = *q.Mp A* $*$ *q.Mp* (*prod-mset Bs*)
using *IH2* **unfolding** *q.unique-factorization-m-alt-def q.factorization-m-def*

by *auto*
  also have *q.Mp* ... = *q.Mp* (*A* ∗ *prod-mset Bs*) **by** *simp*
  finally have *factC*: *q.factorization-m C* (*1*, {# *A* #} + *Bs*) **using** *IH2 monA′ irrA*
    by (*auto simp*: *q.unique-factorization-m-alt-def q.factorization-m-def*)
  show *?thesis*
  proof (*rule exI*, *rule q.unique-factorization-mI*[*OF factC*])
    fix *d gs*
    assume *dgs*: *q.factorization-m C* (*d*,*gs*)
    from *pre-unique*[*OF dgs*, *unfolded add*] **have** *d1*: *q.M d* = *1* **and**
      gs-fs: *image-mset Mp gs* = {# *Mp D* #} + *image-mset Mp H* **by** (*auto simp*: *ac-simps*)
    have ∀ *f m p ma*. *image-mset f m* ≠ *add-mset* (*p*::*int poly*) *ma* ∨
          (∃ *mb pa*. *m* = *add-mset* (*pa*::*int poly*) *mb* ∧ *f pa* = *p* ∧ *image-mset f mb* = *ma*)
      by (*simp add*: *msed-map-invR*)
    then obtain *g hs* **where** *gs*: *gs* = {# *g* #} + *hs* **and** *gD*: *Mp g* = *Mp D*
      and *hsH*: *image-mset Mp hs* = *image-mset Mp H*
      using *gs-fs* **by** (*metis add-mset-add-single union-commute*)
    from *dgs*[*unfolded q.factorization-m-def split*]
    have *eq*: *q.Mp C* = *q.Mp* (*smult d* (*prod-mset gs*))
      and *irr-mon*: ⋀ *g*. *g*∈#*gs* ⟹ *q.irreducible*$_d$*-m g* ∧ *monic* (*q.Mp g*)
      using *d1* **by** *auto*
    note *eq*
    also have *q.Mp* (*smult d* (*prod-mset gs*)) = *q.Mp* (*smult* (*q.M d*) (*prod-mset gs*))
      by *simp*
    also have ... = *q.Mp* (*prod-mset gs*) **unfolding** *d1* **by** *simp*
    finally have *eq*: *q.eq-m* (*q.Mp g* ∗ *q.Mp* (*prod-mset hs*)) *C* **unfolding** *gs* **by** *simp*
    from *gD* **have** *Dg*: *eq-m* (*Mp D*) (*q.Mp g*) **by** *simp*
    have *Mp* (*prod-mset H*) = *Mp* (*prod-mset* (*image-mset Mp H*)) **by** *simp*
    also have ... = *Mp* (*prod-mset hs*) **unfolding** *hsH*[*symmetric*] **by** *simp*
    finally have *Hhs*: *eq-m* (*Mp* (*prod-mset H*)) (*q.Mp* (*prod-mset hs*)) **by** *simp*
    from *irr-mon*[*of g*, *unfolded gs*] **have** *mon-g*: *monic* (*q.Mp g*) **by** *auto*
    from *unique*[*OF eq mon-g Dg Hhs q.Mp-Mp q.Mp-Mp*]
    have *gA*: *q.Mp g* = *A* **and** *hsB*: *q.Mp* (*prod-mset hs*) = *B* **by** *auto*
    have *q.factorization-m B* (*1*, *hs*) **unfolding** *q.factorization-m-def split*
      by (*simp add*: *hsB norm irr-mon*[*unfolded gs*])
  with *IH2* **have** *hsBs*: *q.Mf* (*1*,*hs*) = *q.Mf* (*1*,*Bs*) **unfolding** *q.unique-factorization-m-alt-def* **by** *blast*
    show *q.Mf* (*d*, *gs*) = *q.Mf* (*1*, {# *A* #} + *Bs*)
      using *gA hsBs d1* **unfolding** *gs q.Mf-def* **by** *auto*
  qed
 qed
qed

theorem *berlekamp-hensel-unique*:
 assumes *cop*: *coprime* (*lead-coeff f*) *p*

and *sf*: *poly-mod.square-free-m p f*
and *res*: *berlekamp-hensel p n f = gs*
and *n*: $n \neq 0$
shows *poly-mod.unique-factorization-m* $(p\hat{\ }n)$ *f* (*lead-coeff f*, *mset gs*) — unique factorization mod $p\hat{\ }n$
$\bigwedge$ *g*. *g* $\in$ *set gs* $\Longrightarrow$ *poly-mod.Mp* $(p\hat{\ }n)$ *g = g* — normalized
proof −
let *?q* = $p\hat{\ }n$
interpret *q*: *poly-mod-2 ?q* **unfolding** *poly-mod-2-def* **using** *m1 n* **by** *simp*
from *berlekamp-hensel*[*OF assms*]
have *bh-fact*: *q.factorization-m f* (*lead-coeff f*, *mset gs*) **by** *auto*
from *berlekamp-hensel*[*OF assms*]
show $\bigwedge$ *g*. *g* $\in$ *set gs* $\Longrightarrow$ *poly-mod.Mp* $(p\hat{\ }n)$ *g = g* **by** *blast*
from *prime* **have** *p1*: *p* > *1* **by** (*simp add*: *prime-int-iff*)
let *?lc* = *coeff f* (*degree f*)
define *ilc* **where** *ilc* ≡ *inverse-mod ?lc* $(p\ \hat{}\ n)$
from *cop p1 n* **have** *inv*: *q.M* (*ilc* ∗ *?lc*) = *1*
by (*auto simp add*: *q.M-def ilc-def inverse-mod-pow*)
hence *ilc0*: *ilc* ≠ *0* **by** (*cases ilc* = *0*, *auto*)
{
fix *q*
assume *ilc* ∗ *?lc* = *?q* ∗ *q*
from *arg-cong*[*OF this*, *of q.M*] **have** *q.M* (*ilc* ∗ *?lc*) = *0*
unfolding *q.M-def* **by** *auto*
with *inv* **have** *False* **by** *auto*
} **note** *not-dvd* = *this*
let *?in* = *q.Mp* (*smult ilc f*)
have *mon*: *monic ?in* **unfolding** *q.Mp-coeff coeff-smult*
by (*subst q.degree-m-eq*[*OF - q.m1*], *insert not-dvd*, *auto simp*: *inv ilc0*)
have *q.Mp f* = *q.Mp* (*smult* (*q.M* (*?lc* ∗ *ilc*)) *f*) **using** *inv* **by** (*simp add*: *ac-simps*)
also have . . . = *q.Mp* (*smult ?lc* (*smult ilc f*)) **by** *simp*
finally have *f*: *q.Mp f* = *q.Mp* (*smult ?lc* (*smult ilc f*)) **.**
from *arg-cong*[*OF f*, *of Mp*]
have *Mp f* = *Mp* (*smult ?lc* (*smult ilc f*))
by (*simp add*: *Mp-Mp-pow-is-Mp n p1*)
from *arg-cong*[*OF this*, *of square-free-m*, *unfolded Mp-square-free-m*] *sf*
have *square-free-m* (*smult* (*coeff f* (*degree f*)) (*smult ilc f*)) **by** *simp*
from *square-free-m-smultD*[*OF this*] **have** *sf*: *square-free-m* (*smult ilc f*) **.**
have *Mp-in*: *Mp ?in* = *Mp* (*smult ilc f*)
by (*simp add*: *Mp-Mp-pow-is-Mp n p1*)
from *Mp-square-free-m*[*of ?in*, *unfolded Mp-in*] *sf* **have** *sf*: *square-free-m ?in*
unfolding *Mp-square-free-m* **by** *simp*
obtain *a b* **where** *finite-field-factorization-int p ?in* = (*a*,*b*) **by** *force*
from *finite-field-factorization-int*[*OF sf this*]
have *ufact*: *unique-factorization-m ?in* (*a*, *mset b*) **by** *auto*
from *unique-factorization-m-imp-factorization*[*OF this*]
have *fact*: *factorization-m ?in* (*a*, *mset b*) **.**
from *factorization-m-lead-coeff*[*OF this*] *monic-Mp*[*OF mon*]

have *M a* = *1* **by** *auto*
with *ufact* **have** *unique-factorization-m ?in* (*1*, *mset b*)
unfolding *unique-factorization-m-def Mf-def* **by** *auto*
from *unique-monic-hensel-factorization*[*OF this mon sf n*]
obtain *hs* **where** *q.unique-factorization-m ?in* (*1*, *hs*) **by** *auto*
hence *unique*: *q.unique-factorization-m* (*smult ilc f*) (*1*, *hs*)
unfolding *unique-factorization-m-def Mf-def* **by** *auto*
from *q.factorization-m-smult*[*OF q.unique-factorization-m-imp-factorization*[*OF unique*], *of ?lc*]
have *q.factorization-m* (*smult* (*ilc* ∗ *?lc*) *f*) (*?lc*, *hs*) **by** (*simp add*: *ac-simps*)
moreover have *q.Mp* (*smult* (*q.M* (*ilc* ∗ *?lc*)) *f*) = *q.Mp f* **unfolding** *inv* **by** *simp*
ultimately have *fact*: *q.factorization-m f* (*?lc*, *hs*)
unfolding *q.factorization-m-def* **by** *auto*
have *q.unique-factorization-m f* (*?lc*, *hs*)
proof (*rule q.unique-factorization-mI*[*OF fact*])
fix *d us*
assume *other-fact*: *q.factorization-m f* (*d*,*us*)
from *q.factorization-m-lead-coeff*[*OF this*] **have** *lc*: *q.M d* = *lead-coeff* (*q.Mp f*) **..**
have *lc*: *q.M d* = *q.M ?lc* **unfolding** *lc*
by (*metis bh-fact q.factorization-m-lead-coeff*)
from *q.factorization-m-smult*[*OF other-fact*, *of ilc*] *unique*
have *eq*: *q.Mf* (*d* ∗ *ilc*, *us*) = *q.Mf* (*1*, *hs*) **unfolding** *q.unique-factorization-m-def* **by** *auto*
thus *q.Mf* (*d*, *us*) = *q.Mf* (*?lc*, *hs*) **using** *lc* **unfolding** *q.Mf-def* **by** *auto*
qed
with *bh-fact* **show** *q.unique-factorization-m f* (*lead-coeff f*, *mset gs*)
unfolding *q.unique-factorization-m-alt-def* **by** *metis*
qed

lemma *hensel-lifting-unique*:
assumes *n*: *n* ≠ *0*
and *res*: *hensel-lifting p n f fs* = *gs* — result of hensel is fact. *gs*
and *cop*: *coprime* (*lead-coeff f*) *p*
and *sf*: *poly-mod.square-free-m p f*
and *fact*: *poly-mod.factorization-m p f* (*c*, *mset fs*) — input is fact. *fs mod p*
and *c*: *c* ∈ {*0*..<*p*}
and *norm*: (∀ *fi*∈*set fs*. *set* (*coeffs fi*) ⊆ {*0*..<*p*})
shows *poly-mod.unique-factorization-m* (*p*^*n*) *f* (*lead-coeff f*, *mset gs*) — unique factorization mod *p*^*n*
sort (*map degree fs*) = *sort* (*map degree gs*) — degrees stay the same
⋀ *g*. *g* ∈ *set gs* ⟹ *monic g* ∧ *poly-mod.Mp* (*p*^*n*) *g* = *g* ∧ — monic and normalized
poly-mod.irreducible-m p g ∧ — irreducibility even mod *p*
poly-mod.degree-m p g = *degree g* — mod *p* does not change degree of *g*

proof −
note *hensel* = *hensel-lifting*[*OF assms*]
show *sort* (*map degree fs*) = *sort* (*map degree gs*)
$\bigwedge$ *g*. *g* ∈ *set gs* ⟹ *monic g* ∧ *poly-mod.Mp* (*p*^*n*) *g* = *g* ∧
poly-mod.irreducible-m p g ∧
poly-mod.degree-m p g = *degree g* **using** *hensel* **by** *auto*
from *berlekamp-hensel-unique*[*OF cop sf refl n*]
have *poly-mod.unique-factorization-m* (*p* ^ *n*) *f* (*lead-coeff f*, *mset* (*berlekamp-hensel p n f*)) **by** *auto*
with *hensel*(*1*) **show** *poly-mod.unique-factorization-m* (*p*^*n*) *f* (*lead-coeff f*, *mset gs*)
by (*metis poly-mod.unique-factorization-m-alt-def*)
qed

end

end

10 Reconstructing Factors of Integer Polynomials

10.1 Square-Free Polynomials over Finite Fields and Integers

theory *Square-Free-Int-To-Square-Free-GFp*
imports
Subresultants.Subresultant-Gcd
Polynomial-Factorization.Rational-Factorization
Finite-Field
Polynomial-Factorization.Square-Free-Factorization
begin

lemma *square-free-int-rat*: **assumes** *sf*: *square-free f*
shows *square-free* (*map-poly rat-of-int f*)
proof −
let *?r* = *map-poly rat-of-int*
from *sf*[*unfolded square-free-def*] **have** *f0*: *f* ≠ *0* $\bigwedge$ *q*. *degree q* ≠ *0* ⟹ ¬ *q* ∗ *q dvd f* **by** *auto*
show *?thesis*
proof (*rule square-freeI*)
show *?r f* ≠ *0* **using** *f0* **by** *auto*
fix *q*
assume *dq*: *degree q* > *0* **and** *dvd*: *q* ∗ *q dvd ?r f*
hence *q0*: *q* ≠ *0* **by** *auto*
obtain *q′ c* **where** *norm*: *rat-to-normalized-int-poly q* = (*c*,*q′*) **by** *force*
from *rat-to-normalized-int-poly*[*OF norm*] **have** *c0*: *c* ≠ *0* **by** *auto*
note *q* = *rat-to-normalized-int-poly*(*1*)[*OF norm*]
from *dvd* **obtain** *k* **where** *rf*: *?r f* = *q* ∗ (*q* ∗ *k*) **unfolding** *dvd-def* **by** (*auto simp*: *ac-simps*)
from *rat-to-int-factor-explicit*[*OF this norm*] **obtain** *s* **where**

f: *f* = *q′* ∗ *smult* (*content f*) *s* **by** *auto*
let *?s* = *smult* (*content f*) *s*
from *arg-cong*[*OF f*, *of ?r*] *c0*
have *?r f* = *q* ∗ (*smult* (*inverse c*) (*?r ?s*))
by (*simp add*: *field-simps q hom-distribs*)
from *arg-cong*[*OF this*[*unfolded rf*], *of* λ *f*. *f div q*] *q0*
have *q* ∗ *k* = *smult* (*inverse c*) (*?r ?s*)
by (*metis nonzero-mult-div-cancel-left*)
from *arg-cong*[*OF this*, *of smult c*] **have** *?r ?s* = *q* ∗ *smult c k* **using** *c0*
by (*auto simp*: *field-simps*)
from *rat-to-int-factor-explicit*[*OF this norm*] **obtain** *t* **where** *?s* = *q′* ∗ *t* **by** *blast*
from *f*[*unfolded this*] *sf*[*unfolded square-free-def*] *f0* **have** *degree q′* = *0* **by** *auto*
with *rat-to-normalized-int-poly*(*4*)[*OF norm*] *dq* **show** *False* **by** *auto*
qed
qed

lemma *content-free-unit*:
assumes *content* (*p*::′*a*::{*idom*,*semiring-gcd*} *poly*) = *1*
shows *p dvd 1* ⟷ *degree p* = *0*
by (*insert assms*, *auto dest*!:*degree0-coeffs simp*: *normalize-1-iff poly-dvd-1*)

lemma *square-free-imp-resultant-non-zero*: **assumes** *sf*: *square-free* (*f* :: *int poly*)
shows *resultant f* (*pderiv f*) ≠ *0*
proof (*cases degree f* = *0*)
case *True*
from *degree0-coeffs*[*OF this*] **obtain** *c* **where** *f*: *f* = [:*c*:] **by** *auto*
with *sf* **have** *c*: *c* ≠ *0* **unfolding** *square-free-def* **by** *auto*
show *?thesis* **unfolding** *f* **by** *simp*
next
case *False* **note** *deg* = *this*
define *pp* **where** *pp* = *primitive-part f*
define *c* **where** *c* = *content f*
from *sf* **have** *f0*: *f* ≠ *0* **unfolding** *square-free-def* **by** *auto*
hence *c0*: *c* ≠ *0* **unfolding** *c-def* **by** *auto*
have *f*: *f* = *smult c pp* **unfolding** *c-def pp-def* **unfolding** *content-times-primitive-part*[*of f*] ..
from *sf*[*unfolded f*] *c0* **have** *sf′*: *square-free pp* **by** (*metis dvd-smult smult-0-right square-free-def*)
from *deg*[*unfolded f*] *c0* **have** *deg′*: ⋀ *x*. *degree pp* > *0* ∨ *x* **by** *auto*
from *content-primitive-part*[*OF f0*] **have** *cp*: *content pp* = *1* **unfolding** *pp-def* .
let *?p′* = *pderiv pp*
{
assume *resultant pp ?p′* = *0*
from *this*[*unfolded resultant-0-gcd*] **have** ¬ *coprime pp ?p′* **by** *auto*
then obtain *r* **where** *r*: *r dvd pp r dvd ?p′* ¬ *r dvd 1*
by (*blast elim*: *not-coprimeE*)

from $r(1)$ **obtain** k **where** $pp = r * k$ **..**
from *pos-zmult-eq-1-iff-lemma*[*OF arg-cong*[*OF this*,
of content, *unfolded content-mult cp*, *symmetric*]] *content-ge-0-int*[*of r*]
have *cr*: *content r* $=$ *1* **by** *auto*
with $r(3)$ *content-free-unit* **have** *dr*: *degree r* $\neq$ *0* **by** *auto*
let *?r* $=$ *map-poly rat-of-int*
from $r(1)$ **have** *dvd*: *?r r dvd ?r pp* **unfolding** *dvd-def* **by** (*auto simp*: *hom-distribs*)
from $r(2)$ **have** *?r r dvd ?r ?p′* **apply** (*intro of-int-poly-hom.hom-dvd*) **by** *auto*
also have *?r ?p′* $=$ *pderiv* (*?r pp*) **unfolding** *of-int-hom.map-poly-pderiv* **..**
finally have *dvd′*: *?r r dvd pderiv* (*?r pp*) **by** *auto*
from *dr* **have** *dr′*: *degree* (*?r r*) $\neq$ *0* **by** *simp*
from *square-free-imp-separable*[*OF square-free-int-rat*[*OF sf′*]]
have *separable* (*?r pp*) **.**
hence *cop*: *coprime* (*?r pp*) (*pderiv* (*?r pp*)) **unfolding** *separable-def* **.**
from *f0 f* **have** *pp0*: *pp* $\neq$ *0* **by** *auto*
from *dvd dvd′* **have** *?r r dvd gcd* (*?r pp*) (*pderiv* (*?r pp*)) **by** *auto*
from *divides-degree*[*OF this*] *pp0* **have** *degree* (*?r r*) $\leq$ *degree* (*gcd* (*?r pp*) (*pderiv* (*?r pp*)))
by *auto*
with *dr′* **have** *degree* (*gcd* (*?r pp*) (*pderiv* (*?r pp*))) $\neq$ *0* **by** *auto*
hence ¬ *coprime* (*?r pp*) (*pderiv* (*?r pp*)) **by** *auto*
with *cop* **have** *False* **by** *auto*
}
hence *resultant pp ?p′* $\neq$ *0* **by** *auto*
with *resultant-smult-left*[*OF c0*, *of pp ?p′*, *folded f*] *c0*
have *resultant f ?p′* $\neq$ *0* **by** *auto*
with *resultant-smult-right*[*OF c0*, *of f ?p′*, *folded pderiv-smult f*] *c0*
show *resultant f* (*pderiv f*) $\neq$ *0* **by** *auto*
qed

lemma *large-mod-0*: **assumes** (*n* :: *int*) $>$ *1* $|k| < n$ *k mod n* $=$ *0* **shows** $k = 0$
proof −
from ‹*k mod n* $=$ *0*› **have** *n dvd k*
by *auto*
then obtain *m* **where** $k = n * m$ **..**
with ‹$n > 1$› ‹$|k| < n$› **show** *?thesis*
by (*auto simp add*: *abs-mult*)
qed

definition *separable-bound* :: *int poly* ⇒ *int* **where**
separable-bound f $=$ *max* (*abs* (*resultant f* (*pderiv f*)))
(*max* (*abs* (*lead-coeff f*)) (*abs* (*lead-coeff* (*pderiv f*))))

lemma *square-free-int-imp-resultant-non-zero-mod-ring*: **assumes** *sf*: *square-free f*

and *large*: *int CARD*(*′a*) $>$ *separable-bound f*
shows *resultant* (*map-poly of-int f* :: *′a* :: *prime-card mod-ring poly*) (*pderiv* (*map-poly of-int f*)) $\neq$ *0*

∧ *map-poly of-int f* ≠ (*0* :: *'a mod-ring poly*)
proof (*intro conjI*, *rule notI*)
let *?i* = *of-int* :: *int* ⇒ *'a mod-ring*
let *?m* = *of-int-poly* :: - ⇒ *'a mod-ring poly*
let *?f* = *?m f*
from *sf*[*unfolded square-free-def*] **have** *f0*: *f* ≠ *0* **by** *auto*
hence *lf*: *lead-coeff f* ≠ *0* **by** *auto*
{
fix *k* :: *int*
have *C1*: *int CARD*(*'a*) > *1* **using** *prime-card*[**where** *'a* = *'a*] **by** (*auto simp*: *prime-nat-iff*)
assume *abs k* < *CARD*(*'a*) **and** *?i k* = *0*
hence *k* = *0* **unfolding** *of-int-of-int-mod-ring*
by (*transfer*) (*rule large-mod-0*[*OF C1*])
} **note** *of-int-0* = *this*
from *square-free-imp-resultant-non-zero*[*OF sf*]
have *non0*: *resultant f* (*pderiv f*) ≠ *0* **.**
{
fix *g* :: *int poly*
assume *abs*: *abs* (*lead-coeff g*) < *CARD*(*'a*)
have *degree* (*?m g*) = *degree g* **by** (*rule degree-map-poly*, *insert of-int-0*[*OF abs*], *auto*)
} **note** *deg* = *this*
note *large* = *large*[*unfolded separable-bound-def*]
from *of-int-0*[*of lead-coeff f*] *large lf* **have** *?i* (*lead-coeff f*) ≠ *0* **by** *auto*
thus *f0*: *?f* ≠ *0* **unfolding** *poly-eq-iff* **by** *auto*
assume *0*: *resultant ?f* (*pderiv ?f*) = *0*
have *resultant ?f* (*pderiv ?f*) = *?i* (*resultant f* (*pderiv f*))
unfolding *of-int-hom.map-poly-pderiv*[*symmetric*]
by (*subst of-int-hom.resultant-map-poly*(*1*)[*OF deg deg*], *insert large*, *auto simp*: *hom-distribs*)
from *of-int-0*[*OF - this*[*symmetric*, *unfolded 0*]] *non0*
show *False* **using** *large* **by** *auto*
qed

lemma *square-free-int-imp-separable-mod-ring*: **assumes** *sf*: *square-free f*
and *large*: *int CARD*(*'a*) > *separable-bound f*
shows *separable* (*map-poly of-int f* :: *'a* :: *prime-card mod-ring poly*)
proof −
define *g* **where** *g* = *map-poly* (*of-int* :: *int* ⇒ *'a mod-ring*) *f*
from *square-free-int-imp-resultant-non-zero-mod-ring*[*OF sf large*]
have *res*: *resultant g* (*pderiv g*) ≠ *0* **and** *g*: *g* ≠ *0* **unfolding** *g-def* **by** *auto*
from *res*[*unfolded resultant-0-gcd*] **have** *degree* (*gcd g* (*pderiv g*)) = *0* **by** *auto*
from *degree0-coeffs*[*OF this*]
have *separable g* **unfolding** *separable-def*
by (*metis degree-pCons-0 g gcd-eq-0-iff is-unit-gcd is-unit-iff-degree*)
thus *?thesis* **unfolding** *g-def* **.**
qed

lemma *square-free-int-imp-square-free-mod-ring*: **assumes** *sf*: *square-free f*
  and *large*: *int CARD('a) > separable-bound f*
shows *square-free (map-poly of-int f :: 'a :: prime-card mod-ring poly)*
 using *separable-imp-square-free[OF square-free-int-imp-separable-mod-ring[OF assms]]*
.

end

10.2 Finding a Suitable Prime

The Berlekamp-Zassenhaus algorithm demands for an input polynomial f to determine a prime p such that f is square-free mod p and such that p and the leading coefficient of f are coprime. To this end, we first prove that such a prime always exists, provided that f is square-free over the integers. Second, we provide a generic algorithm which searches for primes have a certain property P. Combining both results gives us the suitable prime for the Berlekamp-Zassenhaus algorithm.

theory *Suitable-Prime*
imports
  Poly-Mod
  Finite-Field-Record-Based
  HOL−Types-To-Sets.Types-To-Sets
  Poly-Mod-Finite-Field-Record-Based
  Polynomial-Record-Based
  Square-Free-Int-To-Square-Free-GFp
begin

lemma *square-free-separable-GFp*: **fixes** *f :: 'a :: prime-card mod-ring poly*
  assumes *card*: *CARD('a) > degree f*
  and *sf*: *square-free f*
  shows *separable f*
proof (*rule ccontr*)
  assume ¬ *separable f*
  with *square-free-separable-main[OF sf]*
  obtain *g k* **where** *: $f = g * k$ *degree g* $\neq 0$ **and** *g0*: *pderiv g* $= 0$ **by** *auto*
  from *assms* **have** *f*: $f \neq 0$ **unfolding** *square-free-def* **by** *auto*
  have *degree f = degree g + degree k* **using** *f* **unfolding** *(1)
    by (*subst degree-mult-eq, auto*)
  with *card* **have** *card*: *degree g < CARD('a)* **by** *auto*
  from *(2) **obtain** *n* **where** *deg*: *degree g = Suc n* **by** (*cases degree g, auto*)
  from *(2) **have** *cg*: *coeff g (degree g)* $\neq 0$ **by** *auto*
  from *g0* **have** *coeff (pderiv g) n = 0* **by** *auto*
  from *this[unfolded coeff-pderiv, folded deg] cg*
  have *of-nat (degree g) = (0 :: 'a mod-ring)* **by** *auto*
  from *of-nat-0-mod-ring-dvd[OF this]* **have** *CARD('a) dvd degree g* .
  with *card* **show** *False* **by** (*simp add: deg nat-dvd-not-less*)
qed

lemma *square-free-iff-separable-GFp*: **assumes** *degree f* $<$ *CARD*($'a$)
  shows *square-free* (f :: $'a$ :: *prime-card mod-ring poly*) = *separable f*
  using *separable-imp-square-free*[*of f*] *square-free-separable-GFp*[*OF assms*] **by** *auto*

definition *separable-impl-main* :: *int* ⇒ $'i$ *arith-ops-record* ⇒ *int poly* ⇒ *bool*
where
  separable-impl-main p ff-ops f = *separable-i ff-ops* (*of-int-poly-i ff-ops* (*poly-mod.Mp p f*))

lemma (**in** *prime-field-gen*) *separable-impl*:
  shows *separable-impl-main p ff-ops f* ⟹ *square-free-m f*
  p > *degree-m f* ⟹ *p* > *separable-bound f* ⟹ *square-free f*
  ⟹ *separable-impl-main p ff-ops f* **unfolding** *separable-impl-main-def*
proof −
  define *F* **where** *F*: (*F* :: $'a$ *mod-ring poly*) = *of-int-poly* (*Mp f*)
  let *?f'* = *of-int-poly-i ff-ops* (*Mp f*)
  define *f''* **where** *f''* ≡ *of-int-poly* (*Mp f*) :: $'a$ *mod-ring poly*
  have *rel-f*[*transfer-rule*]: *poly-rel ?f' f''*
    by (*rule poly-rel-of-int-poly*[*OF refl*], *simp add*: *f''-def*)
  have *separable-i ff-ops ?f'* ⟷ *gcd f''* (*pderiv f''*) = *1*
    unfolding *separable-i-def* **by** *transfer-prover*
  also have ... ⟷ *coprime f''* (*pderiv f''*)
    by (*auto simp add*: *gcd-eq-1-imp-coprime*)
  finally have *id*: *separable-i ff-ops ?f'* ⟷ *separable f''* **unfolding** *separable-def coprime-iff-coprime* **.**
  have *Mprel* [*transfer-rule*]: *MP-Rel* (*Mp f*) *F* **unfolding** *F MP-Rel-def*
    by (*simp add*: *Mp-f-representative*)
  have *square-free f''* = *square-free F* **unfolding** *f''-def F* **by** *simp*
  also have ... = *square-free-m* (*Mp f*)
    by (*transfer*, *simp*)
  also have ... = *square-free-m f* **by** *simp*
  finally have *id2*: *square-free f''* = *square-free-m f* **.**
  from *separable-imp-square-free*[*of f''*]
  show *separable-i ff-ops ?f'* ⟹ *square-free-m f*
    unfolding *id id2* **by** *auto*
  let *?m* = *map-poly* (*of-int* :: *int* ⇒ $'a$ *mod-ring*)
  let *?f* = *?m f*
  assume *p* > *degree-m f* **and** *bnd*: *p* > *separable-bound f* **and** *sf*: *square-free f*
  with *rel-funD*[*OF degree-MP-Rel Mprel*, *folded p*]
  have *p* > *degree F* **by** *simp*
  hence *CARD*($'a$) > *degree f''* **unfolding** *f''-def F p* **by** *simp*
  from *square-free-iff-separable-GFp*[*OF this*]
  have *separable-i ff-ops ?f'* = *square-free f''* **unfolding** *id id2* **by** *simp*
  also have ... = *square-free F* **unfolding** *f''-def F* **by** *simp*
  also have *F* = *?f* **unfolding** *F*
    by (*rule poly-eqI*, (*subst coeff-map-poly*, *force*)+, *unfold Mp-coeff*,
    auto simp: *M-def*, *transfer*, *auto simp*: *p*)
  also have *square-free ?f* **using** *square-free-int-imp-square-free-mod-ring*[**where**

$'a = 'a$, *OF sf*] *bnd m* **by** *auto*
finally
show *separable-i ff-ops ?f'* **.**
qed

context *poly-mod-prime* **begin**

lemmas *separable-impl-integer* = *prime-field-gen.separable-impl*
[*OF prime-field.prime-field-finite-field-ops-integer*, *unfolded prime-field-def mod-ring-locale-def*,
unfolded poly-mod-type-simps, *internalize-sort* $'a$ :: *prime-card*, *OF type-to-set*,
unfolded remove-duplicate-premise, *cancel-type-definition*, *OF non-empty*]

lemmas *separable-impl-uint32* = *prime-field-gen.separable-impl*
[*OF prime-field.prime-field-finite-field-ops32*, *unfolded prime-field-def mod-ring-locale-def*,
unfolded poly-mod-type-simps, *internalize-sort* $'a$ :: *prime-card*, *OF type-to-set*,
unfolded remove-duplicate-premise, *cancel-type-definition*, *OF non-empty*]

lemmas *separable-impl-uint64* = *prime-field-gen.separable-impl*
[*OF prime-field.prime-field-finite-field-ops64*, *unfolded prime-field-def mod-ring-locale-def*,
unfolded poly-mod-type-simps, *internalize-sort* $'a$ :: *prime-card*, *OF type-to-set*,
unfolded remove-duplicate-premise, *cancel-type-definition*, *OF non-empty*]

end

definition *separable-impl* :: *int* ⇒ *int poly* ⇒ *bool* **where**
separable-impl p = (
if p ≤ *65535*
then separable-impl-main p (*finite-field-ops32* (*uint32-of-int p*))
else if p ≤ *4294967295*
then separable-impl-main p (*finite-field-ops64* (*uint64-of-int p*))
else separable-impl-main p (*finite-field-ops-integer* (*integer-of-int p*)))

lemma *square-free-mod-imp-square-free*: **assumes**
p: *prime p* **and** *sf*: *poly-mod.square-free-m p f*
and *cop*: *coprime* (*lead-coeff f*) *p*
shows *square-free f*
proof −
interpret *poly-mod p* **.**
from *sf*[*unfolded square-free-m-def*] **have** *f0*: *Mp f* ≠ *0* **and** *ndvd*: ⋀ *g*. *degree-m g* > *0* ⟹ ¬ (*g* ∗ *g*) *dvdm f*
by *auto*
from *f0* **have** *ff0*: *f* ≠ *0* **by** *auto*
show *square-free f* **unfolding** *square-free-def*
proof (*intro conjI*[*OF ff0*] *allI impI notI*)
fix *g*
assume *deg*: *degree g* > *0* **and** *dvd*: *g* ∗ *g dvd f*
then obtain *h* **where** *f*: *f* = *g* ∗ *g* ∗ *h* **unfolding** *dvd-def* **by** *auto*
from *arg-cong*[*OF this*, *of Mp*] **have** (*g* ∗ *g*) *dvdm f* **unfolding** *dvdm-def* **by** *auto*

with *ndvd*[*of g*] **have** *deg0*: *degree-m g* = *0* **by** *auto*
hence *g0*: *M* (*lead-coeff g*) = *0* **unfolding** *Mp-def* **using** *deg*
by (*metis M-def deg0 p poly-mod.degree-m-eq prime-gt-1-int neq0-conv*)
from *p* **have** *p0*: *p* ≠ *0* **by** *auto*
from *arg-cong*[*OF f*, *of lead-coeff*] **have** *lead-coeff f* = *lead-coeff g* ∗ *lead-coeff g* ∗ *lead-coeff h*
by (*auto simp*: *lead-coeff-mult*)
hence *lead-coeff g dvd lead-coeff f* **by** *auto*
with *cop* **have** *cop*: *coprime* (*lead-coeff g*) *p*
by (*auto elim*: *coprime-imp-coprime intro*: *dvd-trans*)
with *p0* **have** *coprime* (*lead-coeff g mod p*) *p* **by** *simp*
also have *lead-coeff g mod p* = *0*
using *M-def g0* **by** *simp*
finally show *False* **using** *p*
unfolding *prime-int-iff*
by (*simp add*: *prime-int-iff*)
qed
qed

lemma(**in** *poly-mod-prime*) *separable-impl*:
shows *separable-impl p f* ⟹ *square-free-m f*
nat p > *degree-m f* ⟹ *nat p* > *separable-bound f* ⟹ *square-free f*
⟹ *separable-impl p f*
using
separable-impl-integer[*of f*]
separable-impl-uint32[*of f*]
separable-impl-uint64[*of f*]
unfolding *separable-impl-def* **by** (*auto split*: *if-splits*)

lemma *coprime-lead-coeff-large-prime*: **assumes** *prime*: *prime* (*p* :: *int*)
and *large*: *p* > *abs* (*lead-coeff f*)
and *f*: *f* ≠ *0*
shows *coprime* (*lead-coeff f*) *p*
proof −
{
fix *lc*
assume *0* < *lc lc* < *p*
then have ¬ *p dvd lc*
by (*simp add*: *zdvd-not-zless*)
with ‹*prime p*› **have** *coprime p lc*
by (*auto intro*: *prime-imp-coprime*)
then have *coprime lc p*
by (*simp add*: *ac-simps*)
} **note** *main* = *this*
define *lc* **where** *lc* = *lead-coeff f*
from *f* **have** *lc0*: *lc* ≠ *0* **unfolding** *lc-def* **by** *auto*
from *large* **have** *large*: *p* > *abs lc* **unfolding** *lc-def* **by** *auto*
have *coprime lc p*
proof (*cases lc* > *0*)

```
    case True
    from large have p > lc by auto
    from main[OF True this] show ?thesis .
  next
    case False
    let ?mlc = − lc
    from large False lc0 have ?mlc > 0 p > ?mlc by auto
    from main[OF this] show ?thesis by simp
  qed
  thus ?thesis unfolding lc-def by auto
qed

lemma prime-for-berlekamp-zassenhaus-exists: assumes sf: square-free f
  shows ∃ p. prime p ∧ (coprime (lead-coeff f) p ∧ separable-impl p f)
proof (rule ccontr)
  from assms have f0: f ≠ 0 unfolding square-free-def by auto
  define n where n = max (max (abs (lead-coeff f)) (degree f)) (separable-bound
f)
  assume contr: ¬ ?thesis
  {
    fix p :: int
    assume prime: prime p and n: p > n
    then interpret poly-mod-prime p by unfold-locales
     from n have large: p > abs (lead-coeff f) nat p > degree f nat p > separa-
ble-bound f
      unfolding n-def by auto
    from coprime-lead-coeff-large-prime[OF prime large(1) f0]
    have cop: coprime (lead-coeff f) p by auto
    with prime contr have nsf: ¬ separable-impl p f by auto
    from large(2) have nat p > degree-m f using degree-m-le[of f] by auto
    from separable-impl(2)[OF this large(3) sf] nsf have False by auto
  }
  hence no-large-prime: ⋀ p. prime p ⟹ p > n ⟹ False by auto
  from bigger-prime[of nat n] obtain p where ∗: prime p p > nat n by auto
  define q where q ≡ int p
  from ∗ have prime q q > n unfolding q-def by auto
  from no-large-prime[OF this]
  show False .
qed

definition next-primes :: nat ⇒ nat × nat list where
  next-primes n = (if n = 0 then next-candidates 0 else
    let (m,ps) = next-candidates n in (m,filter prime ps))

partial-function (tailrec) find-prime-main ::
  (nat ⇒ bool) ⇒ nat ⇒ nat list ⇒ nat where
  [code]: find-prime-main f np ps = (case ps of [] ⇒
    let (np',ps') = next-primes np
      in find-prime-main f np' ps'
```

| $(p \# ps) \Rightarrow$ *if f p then p else find-prime-main f np ps*)

definition *find-prime* :: (*nat* ⇒ *bool*) ⇒ *nat* **where**
find-prime f = *find-prime-main f 0* []

lemma *next-primes*: **assumes** *res*: *next-primes n* = (*m*,*ps*)
and *n*: *candidate-invariant n*
shows *candidate-invariant m sorted ps distinct ps* $n < m$
set ps = $\{i.$ *prime* $i \wedge n \leq i \wedge i < m\}$
proof −
have *candidate-invariant m* ∧ *sorted ps* ∧ *distinct ps* ∧ $n < m$ ∧
set ps = $\{i.$ *prime* $i \wedge n \leq i \wedge i < m\}$
proof (*cases n* = *0*)
case *True*
with *res*[*unfolded next-primes-def*] **have** *nc*: *next-candidates 0* = (*m*,*ps*) **by** *auto*
from *this*[*unfolded next-candidates-def*] **have** *ps*: *ps* = *primes-1000* **and** *m*: *m* = *1001* **by** *auto*
have *ps*: *set ps* = $\{i.$ *prime* $i \wedge n \leq i \wedge i < m\}$ **unfolding** *m True ps*
by (*auto simp*: *primes-1000*)
with *next-candidates*[*OF nc n*[*unfolded True*]] *True*
show *?thesis* **by** *simp*
next
case *False*
with *res*[*unfolded next-primes-def Let-def*] **obtain** *qs* **where** *nc*: *next-candidates n* = (*m*, *qs*)
and *ps*: *ps* = *filter prime qs* **by** (*cases next-candidates n*, *auto*)
have *sorted qs* ⟹ *sorted ps* **unfolding** *ps* **using** *sorted-filter*[*of id qs prime*] **by** *auto*
with *next-candidates*[*OF nc n*] **show** *?thesis* **unfolding** *ps* **by** *auto*
qed
thus *candidate-invariant m sorted ps distinct ps* $n < m$
set ps = $\{i.$ *prime* $i \wedge n \leq i \wedge i < m\}$ **by** *auto*
qed

lemma *find-prime*: **assumes** ∃ *n*. *prime n* ∧ *f n*
shows *prime* (*find-prime f*) ∧ *f* (*find-prime f*)
proof −
from *assms* **obtain** *n* **where** *fn*: *prime n f n* **by** *auto*
{
fix *i ps m*
assume *candidate-invariant i*
and $n \in$ *set ps* ∨ $n \geq i$
and *m* = (*Suc n* − *i*, *length ps*)
and ⋀ *p*. *p* ∈ *set ps* ⟹ *prime p*
hence *prime* (*find-prime-main f i ps*) ∧ *f* (*find-prime-main f i ps*)
proof (*induct m arbitrary*: *i ps rule*: *wf-induct*[*OF wf-measures*[*of* [*fst*, *snd*]]])
case (*1 m i ps*)

note *IH* = *1*(*1*)[*rule-format*]
note *can* = *1*(*2*)
note *n* = *1*(*3*)
note *m* = *1*(*4*)
note *ps* = *1*(*5*)
note *simps* [*simp*] = *find-prime-main.simps*[*of f i ps*]
show *?case*
proof (*cases ps*)
case *Nil*
with *n* **have** *i-n*: $i \leq n$ **by** *auto*
obtain *j qs* **where** *np*: *next-primes i* = (*j*,*qs*) **by** *force*
note *j* = *next-primes*[*OF np can*]
from *j*(*4*) *i-n m* **have** *meas*: ((*Suc n* − *j*, *length qs*), *m*) ∈ *measures* [*fst*, *snd*] **by** *auto*
have *n*: *n* ∈ *set qs* ∨ $j \leq n$ **unfolding** *j*(*5*) **using** *i-n fn* **by** *auto*
show *?thesis* **unfolding** *simps* **using** *IH*[*OF meas j*(*1*) *n refl*] *j*(*5*) **by** (*simp add*: *Nil np*)
next
case (*Cons p qs*)
show *?thesis*
proof (*cases f p*)
case *True*
thus *?thesis* **unfolding** *simps* **using** *ps* **unfolding** *Cons* **by** *simp*
next
case *False*
have *m*: ((*Suc n* − *i*, *length qs*), *m*) ∈ *measures* [*fst*, *snd*] **using** *m* **unfolding** *Cons* **by** *simp*
have *n*: *n* ∈ *set qs* ∨ $i \leq n$ **using** *False n fn* **by** (*auto simp*: *Cons*)
from *IH*[*OF m can n refl ps*]
show *?thesis* **unfolding** *simps* **using** *Cons False* **by** *simp*
qed
qed
qed
} **note** *main* = *this*
have *candidate-invariant 0* **by** (*simp add*: *candidate-invariant-def*)
from *main*[*OF this - refl*, *of Nil*] **show** *?thesis* **unfolding** *find-prime-def* **by** *auto*
qed

definition *suitable-prime-bz* :: *int poly* ⇒ *int* **where**
suitable-prime-bz f ≡ *let lc* = *lead-coeff f in int* (*find-prime* (λ *n*. *let p* = *int n in*
coprime lc p ∧ *separable-impl p f*))

lemma *suitable-prime-bz*: **assumes** *sf*: *square-free f* **and** *p*: *p* = *suitable-prime-bz f*
shows *prime p coprime* (*lead-coeff f*) *p poly-mod.square-free-m p f*
proof −
let *?lc* = *lead-coeff f*

from *prime-for-berlekamp-zassenhaus-exists*[*OF sf*, *unfolded Let-def*]
obtain *P* **where** ∗: *prime P* ∧ *coprime ?lc P* ∧ *separable-impl P f*
by *auto*
hence *prime* (*nat P*) **using** *prime-int-nat-transfer* **by** *blast*
with ∗ **have** ∃ *p*. *prime p* ∧ *coprime ?lc* (*int p*) ∧ *separable-impl p f*
by (*intro exI* [*of - nat P*]) (*auto dest*: *prime-gt-0-int*)
from *find-prime*[*OF this*]
have *prime*: *prime p* **and** *cop*: *coprime ?lc p* **and** *sf*: *separable-impl p f*
unfolding *p suitable-prime-bz-def Let-def* **by** *auto*
then interpret *poly-mod-prime p* **by** *unfold-locales*
from *prime cop separable-impl*(*1*)[*OF sf*]
show *prime p coprime ?lc p square-free-m f* **by** *auto*
qed

definition *square-free-heuristic* :: *int poly* ⇒ *int option* **where**
square-free-heuristic f = (*let lc* = *lead-coeff f in*
find (λ *p*. *coprime lc p* ∧ *separable-impl p f*) [*2*, *3*, *5*, *7*, *11*, *13*, *17*, *19*, *23*])

lemma *find-Some-D*: *find f xs* = *Some y* ⟹ *y* ∈ *set xs* ∧ *f y* **unfolding** *find-Some-iff*
by *auto*

lemma *square-free-heuristic*: **assumes** *square-free-heuristic f* = *Some p*
shows *coprime* (*lead-coeff f*) *p* ∧ *separable-impl p f* ∧ *prime p*
proof −
from *find-Some-D*[*OF assms*[*unfolded square-free-heuristic-def Let-def*]]
show *?thesis* **by** *auto*
qed

end

10.3 Maximal Degree during Reconstruction

We define a function which computes an upper bound on the degree of a factor for which we have to reconstruct the integer values of the coefficients. This degree will determine how large the second parameter of the factor-bound will be.

In essence, if the Berlekamp-factorization will produce n factors with degrees $d_1, \ldots, d_n$, then our bound will be the sum of the $\frac{n}{2}$ largest degrees. The reason is that we will combine at most $\frac{n}{2}$ factors before reconstruction.

Soundness of the bound is proven, as well as a monotonicity property.

theory *Degree-Bound*
imports *Containers.Set-Impl*
HOL−Library.Multiset
Polynomial-Interpolation.Missing-Polynomial
Efficient−Mergesort.Efficient-Sort
begin

definition *max-factor-degree* :: *nat list* ⇒ *nat* **where**

max-factor-degree degs = (*let*
ds = *sort degs*
in sum-list (*drop* (*length ds div 2*) *ds*))

definition *degree-bound* **where** *degree-bound vs* = *max-factor-degree* (*map degree vs*)

lemma *insort-middle*: *sort* (*xs* @ *x* # *ys*) = *insort x* (*sort* (*xs* @ *ys*))
by (*metis append.assoc sort-append-Cons-swap sort-snoc*)

lemma *sum-list-insort*[*simp*]:
sum-list (*insort* (*d* :: *'a* :: {*comm-monoid-add*,*linorder*}) *xs*) = *d* + *sum-list xs*
proof (*induct xs*)
case (*Cons x xs*)
thus *?case* **by** (*cases d* ≤ *x*, *auto simp*: *ac-simps*)
qed *simp*

lemma *half-largest-elements-mono*: *sum-list* (*drop* (*length ds div 2*) (*sort ds*))
≤ *sum-list* (*drop* (*Suc* (*length ds*) *div 2*) (*insort* (*d* :: *nat*) (*sort ds*)))
proof −
define *n* **where** *n* = *length ds div 2*
define *m* **where** *m* = *Suc* (*length ds*) *div 2*
define *xs* **where** *xs* = *sort ds*
have *xs*: *sorted xs* **unfolding** *xs-def* **by** *auto*
have *nm*: *m* ∈ {*n*, *Suc n*} **unfolding** *n-def m-def* **by** *auto*
show *?thesis* **unfolding** *n-def*[*symmetric*] *m-def*[*symmetric*] *xs-def*[*symmetric*]
using *nm xs*
proof (*induct xs arbitrary*: *n m d*)
case (*Cons x xs n m d*)
show *?case*
proof (*cases n*)
case *0*
with *Cons*(*2*) **have** *m*: *m* = *0* ∨ *m* = *1* **by** *auto*
show *?thesis*
proof (*cases d* ≤ *x*)
case *True*
hence *ins*: *insort d* (*x* # *xs*) = *d* # *x* # *xs* **by** *auto*
show *?thesis* **unfolding** *ins 0* **using** *True m* **by** *auto*
next
case *False*
hence *ins*: *insort d* (*x* # *xs*) = *x* # *insort d xs* **by** *auto*
show *?thesis* **unfolding** *ins 0* **using** *False m* **by** *auto*
qed
next
case (*Suc nn*)
with *Cons*(*2*) **obtain** *mm* **where** *m*: *m* = *Suc mm* **and** *mm*: *mm* ∈ {*nn*, *Suc nn*} **by** *auto*
from *Cons*(*3*) **have** *sort*: *sorted xs* **by** (*simp*)
note *IH* = *Cons*(*1*)[*OF mm*]

show *?thesis*
proof (*cases d* ≤ *x*)
case *True*
with *Cons*(*3*) have *ins*: *insort d* (*x* # *xs*) = *d* # *insort x xs*
by (*cases xs*, *auto*)
show *?thesis* unfolding *ins Suc m* using *IH*[*OF sort*] by *auto*
next
case *False*
hence *ins*: *insort d* (*x* # *xs*) = *x* # *insort d xs* by *auto*
show *?thesis* unfolding *ins Suc m* using *IH*[*OF sort*] *Cons*(*3*) by *auto*
qed
qed
qed *auto*
qed

lemma *max-factor-degree-mono*:
max-factor-degree (*map degree* (*fold remove1 ws vs*)) ≤ *max-factor-degree* (*map degree vs*)
unfolding *max-factor-degree-def Let-def length-sort length-map*
proof (*induct ws arbitrary*: *vs*)
case (*Cons w ws vs*)
show *?case*
proof (*cases w* ∈ *set vs*)
case *False*
hence *remove1 w vs* = *vs* by (*rule remove1-idem*)
thus *?thesis* using *Cons*[*of vs*] by *auto*
next
case *True*
then obtain *bef aft* where *vs*: *vs* = *bef* @ *w* # *aft* and *rem1*: *remove1 w vs* = *bef* @ *aft*
by (*metis remove1.simps*(*2*) *remove1-append split-list-first*)
let *?exp* = λ *ws vs*. *sum-list* (*drop* (*length* (*fold remove1 ws vs*) *div 2*)
(*sort* (*map degree* (*fold remove1 ws vs*))))
let *?bnd* = λ *vs*. *sum-list* (*drop* (*length vs div 2*) (*sort* (*map degree vs*)))
let *?bd* = λ *vs*. *sum-list* (*drop* (*length vs div 2*) (*sort vs*))
define *ba* where *ba* = *bef* @ *aft*
define *ds* where *ds* = *map degree ba*
define *d* where *d* = *degree w*
have *?exp* (*w* # *ws*) *vs* = *?exp ws* (*bef* @ *aft*) by (*auto simp*: *rem1*)
also have ... ≤ *?bnd ba* unfolding *ba-def* by (*rule Cons*)
also have ... = *?bd ds* unfolding *ds-def* by *simp*
also have ... ≤ *sum-list* (*drop* (*Suc* (*length ds*) *div 2*) (*insort d* (*sort ds*)))
by (*rule half-largest-elements-mono*)
also have ... = *?bnd vs* unfolding *vs ds-def d-def* by (*simp add*: *ba-def insort-middle*)
finally show *?exp* (*w* # *ws*) *vs* ≤ *?bnd vs* by *simp*
qed
qed *auto*

lemma *mset-sub-decompose*: $mset\ ds \subseteq\# \ mset\ bs + as \Longrightarrow length\ ds < length\ bs \Longrightarrow \exists\ b1\ b\ b2.$
$bs = b1\ @\ b\ \#\ b2 \wedge mset\ ds \subseteq\# \ mset\ (b1\ @\ b2) + as$
proof (*induct ds arbitrary*: *bs as*)
case *Nil*
hence $bs = []\ @\ hd\ bs\ \#\ tl\ bs$ **by** *auto*
thus *?case* **by** *fastforce*
next
case (*Cons d ds bs as*)
have $d \in\# \ mset\ (d\ \#\ ds)$ **by** *auto*
with *Cons*(*2*) **have** *d*: $d \in\# \ mset\ bs + as$ **by** (*rule mset-subset-eqD*)
hence $d \in set\ bs \vee d \in\# \ as$ **by** *auto*
thus *?case*
proof
assume $d \in set\ bs$
from *this*[*unfolded in-set-conv-decomp*] **obtain** *b1 b2* **where** *bs*: $bs = b1\ @\ d\ \#\ b2$ **by** *auto*
from *Cons*(*2*) *Cons*(*3*)
have $mset\ ds \subseteq\# \ mset\ (b1\ @\ b2) + as\ length\ ds < length\ (b1\ @\ b2)$ **by** (*auto simp*: *ac-simps bs*)
from *Cons*(*1*)[*OF this*] **obtain** *b1' b b2'* **where** *split*: $b1\ @\ b2 = b1'\ @\ b\ \#\ b2'$
and *sub*: $mset\ ds \subseteq\# \ mset\ (b1'\ @\ b2') + as$ **by** *auto*
from *split*[*unfolded append-eq-append-conv2*]
obtain *us* **where** $b1 = b1'\ @\ us \wedge us\ @\ b2 = b\ \#\ b2' \vee b1\ @\ us = b1' \wedge b2 = us\ @\ b\ \#\ b2'$ **..**
thus *?thesis*
proof
assume $b1\ @\ us = b1' \wedge b2 = us\ @\ b\ \#\ b2'$
hence $*$: $b1\ @\ us = b1'\ b2 = us\ @\ b\ \#\ b2'$ **by** *auto*
hence *bs*: $bs = (b1\ @\ d\ \#\ us)\ @\ b\ \#\ b2'$ **unfolding** *bs* **by** *auto*
show *?thesis*
by (*intro exI conjI*, *rule bs*, *insert* $*$ *sub*, *auto simp*: *ac-simps*)
next
assume $b1 = b1'\ @\ us \wedge us\ @\ b2 = b\ \#\ b2'$
hence $*$: $b1 = b1'\ @\ us\ us\ @\ b2 = b\ \#\ b2'$ **by** *auto*
show *?thesis*
proof (*cases us*)
case *Nil*
with $*$ **have** $*$: $b1 = b1'\ b2 = b\ \#\ b2'$ **by** *auto*
hence *bs*: $bs = (b1'\ @\ [d])\ @\ b\ \#\ b2'$ **unfolding** *bs* **by** *simp*
show *?thesis*
by (*intro exI conjI*, *rule bs*, *insert* $*$ *sub*, *auto simp*: *ac-simps*)
next
case (*Cons u vs*)
with $*$ **have** $*$: $b1 = b1'\ @\ b\ \#\ vs\ vs\ @\ b2 = b2'$ **by** *auto*
hence *bs*: $bs = b1'\ @\ b\ \#\ (vs\ @\ d\ \#\ b2)$ **unfolding** *bs* **by** *auto*
show *?thesis*
by (*intro exI conjI*, *rule bs*, *insert* $*$ *sub*, *auto simp*: *ac-simps*)

qed
qed
next
define as' **where** $as' = as - \{\#d\#\}$
assume $d \in\# as$
hence as': $as = \{\#d\#\} + as'$ **unfolding** as'-*def* **by** *auto*
from *Cons*(*2*)[*unfolded* as'] *Cons*(*3*) **have** *mset ds* $\subseteq\#$ *mset bs* $+ as'$ *length ds* $<$ *length bs*
by (*auto simp*: *ac-simps*)
from *Cons*(*1*)[*OF this*] **obtain** *b1 b b2* **where** *bs*: *bs* = *b1* @ *b* # *b2* **and**
sub: *mset ds* $\subseteq\#$ *mset* (*b1* @ *b2*) + as' **by** *auto*
show *?thesis*
by (*intro exI conjI*, *rule bs*, *insert sub*, *auto simp*: as' *ac-simps*)
qed
qed

lemma *max-factor-degree-aux*: **fixes** *es* :: *nat list*
assumes *sub*: *mset ds* $\subseteq\#$ *mset es*
and *len*: *length ds* + *length ds* $\leq$ *length es* **and** *sort*: *sorted es*
shows *sum-list ds* $\leq$ *sum-list* (*drop* (*length es div 2*) *es*)
proof −
define *bef* **where** *bef* = *take* (*length es div 2*) *es*
define *aft* **where** *aft* = *drop* (*length es div 2*) *es*
have *es*: *es* = *bef* @ *aft* **unfolding** *bef-def aft-def* **by** *auto*
from *len* **have** *len*: *length ds* $\leq$ *length bef length ds* $\leq$ *length aft* **unfolding** *bef-def aft-def*
by *auto*
from *sub* **have** *sub*: *mset ds* $\subseteq\#$ *mset bef* + *mset aft* **unfolding** *es* **by** *auto*
from *sort* **have** *sort*: *sorted* (*bef* @ *aft*) **unfolding** *es* **.**
show *?thesis* **unfolding** *aft-def*[*symmetric*] **using** *sub len sort*
proof (*induct ds arbitrary*: *bef aft*)
case (*Cons d ds bef aft*)
have *d* $\in\#$ *mset* (*d* # *ds*) **by** *auto*
with *Cons*(*2*) **have** *d* $\in\#$ *mset bef* + *mset aft* **by** (*rule mset-subset-eqD*)
hence *d* $\in$ *set bef* $\vee$ *d* $\in$ *set aft* **by** *auto*
thus *?case*
proof
assume *d* $\in$ *set aft*
from *this*[*unfolded in-set-conv-decomp*] **obtain** *a1 a2* **where** *aft*: *aft* = *a1* @ *d* # *a2* **by** *auto*
from *Cons*(*4*) **have** *len-a*: *length ds* $\leq$ *length* (*a1* @ *a2*) **unfolding** *aft* **by** *auto*
from *Cons*(*2*)[*unfolded aft*] *Cons*(*3*)
have *mset ds* $\subseteq\#$ *mset bef* + (*mset* (*a1* @ *a2*)) *length ds* $<$ *length bef* **by** *auto*
from *mset-sub-decompose*[*OF this*]
obtain *b b1 b2*
where *bef*: *bef* = *b1* @ *b* # *b2* **and** *sub*: *mset ds* $\subseteq\#$ (*mset* (*b1* @ *b2*) +

mset (*a1* @ *a2*)) **by** *auto*
from *Cons*(*3*) **have** *len-b*: *length ds* ≤ *length* (*b1* @ *b2*) **unfolding** *bef* **by** *auto*
from *Cons*(*5*)[*unfolded bef aft*] **have** *sort*: *sorted* ((*b1* @ *b2*) @ (*a1* @ *a2*))
unfolding *sorted-append* **by** *auto*
note *IH* = *Cons*(*1*)[*OF sub len-b len-a sort*]
show *?thesis* **using** *IH* **unfolding** *aft* **by** *simp*
next
assume *d* ∈ *set bef*
from *this*[*unfolded in-set-conv-decomp*] **obtain** *b1 b2* **where** *bef*: *bef* = *b1* @ *d* # *b2* **by** *auto*
from *Cons*(*3*) **have** *len-b*: *length ds* ≤ *length* (*b1* @ *b2*) **unfolding** *bef* **by** *auto*
from *Cons*(*2*)[*unfolded bef*] *Cons*(*4*)
have *mset ds* ⊆# *mset aft* + (*mset* (*b1* @ *b2*)) *length ds* < *length aft* **by** (*auto simp*: *ac-simps*)
from *mset-sub-decompose*[*OF this*]
obtain *a a1 a2*
where *aft*: *aft* = *a1* @ *a* # *a2* **and** *sub*: *mset ds* ⊆# (*mset* (*b1* @ *b2*) + *mset* (*a1* @ *a2*))
by (*auto simp*: *ac-simps*)
from *Cons*(*4*) **have** *len-a*: *length ds* ≤ *length* (*a1* @ *a2*) **unfolding** *aft* **by** *auto*
from *Cons*(*5*)[*unfolded bef aft*] **have** *sort*: *sorted* ((*b1* @ *b2*) @ (*a1* @ *a2*)) **and** *ad*: *d* ≤ *a*
unfolding *sorted-append* **by** *auto*
note *IH* = *Cons*(*1*)[*OF sub len-b len-a sort*]
show *?thesis* **using** *IH ad* **unfolding** *aft* **by** *simp*
qed
qed *auto*
qed

lemma *max-factor-degree*: **assumes** *sub*: *mset ws* ⊆# *mset vs*
and *len*: *length ws* + *length ws* ≤ *length vs*
shows *degree* (*prod-list ws*) ≤ *max-factor-degree* (*map degree vs*)
proof −
define *ds* **where** *ds* ≡ *map degree ws*
define *es* **where** *es* ≡ *sort* (*map degree vs*)
from *sub len* **have** *sub*: *mset ds* ⊆# *mset es* **and** *len*: *length ds* + *length ds* ≤ *length es*
and *es*: *sorted es*
unfolding *ds-def es-def*
by (*auto simp*: *image-mset-subseteq-mono*)
have *degree* (*prod-list ws*) ≤ *sum-list* (*map degree ws*) **by** (*rule degree-prod-list-le*)
also have . . . ≤ *max-factor-degree* (*map degree vs*)
unfolding *max-factor-degree-def Let-def ds-def*[*symmetric*] *es-def*[*symmetric*]
using *sub len es* **by** (*rule max-factor-degree-aux*)
finally show *?thesis* **.**
qed

lemma *degree-bound*: **assumes** *sub*: *mset ws* $\subseteq\#$ *mset vs*
  and *len*: *length ws* + *length ws* $\leq$ *length vs*
shows *degree* (*prod-list ws*) $\leq$ *degree-bound vs*
  using *max-factor-degree*[*OF sub len*] **unfolding** *degree-bound-def* **by** *auto*

end

10.4 Mahler Measure

This part contains a definition of the Mahler measure, it contains Landau's inequality and the Graeffe-transformation. We also assemble a heuristic to approximate the Mahler's measure.

theory *Mahler-Measure*
imports
  Sqrt-Babylonian.Sqrt-Babylonian
  Poly-Mod-Finite-Field-Record-Based
  Polynomial-Factorization.Fundamental-Theorem-Algebra-Factorized
  Polynomial-Factorization.Missing-Multiset
begin

context *comm-monoid-list* **begin**
  lemma *induct-gen-abs*:
    assumes $\bigwedge$ *a r. a*$\in$*set lst* $\Longrightarrow$ *P* (*f* (*h a*) *r*) (*f* (*g a*) *r*)
            $\bigwedge$ *x y z. P x y* $\Longrightarrow$ *P y z* $\Longrightarrow$ *P x z*
            P (*F* (*map g lst*)) (*F* (*map g lst*))
    shows *P* (*F* (*map h lst*)) (*F* (*map g lst*))
  using *assms* **proof**(*induct lst arbitrary*:*P*)
    case (*Cons a as P*)
    have *inl*:*a*$\in$*set* (*a*#*as*) **by** *auto*
    let *?uf* = λ *v w. P* (*f* (*g a*) *v*) (*f* (*g a*) *w*)
    have *p-suc*:*?uf* (*F* (*map g as*)) (*F* (*map g as*))
      using *Cons.prems*(*3*) **by** *auto*
    { fix *r aa* **assume** *aa* $\in$ *set as* **hence** *ins*:*aa* $\in$ *set* (*a*#*as*) **by** *auto*
      have *P* (*f* (*g a*) (*f* (*h aa*) *r*)) (*f* (*g a*) (*f* (*g aa*) *r*))
        using *Cons.prems*(*1*)[*of aa f r* (*g a*),*OF ins*]
        by (*auto simp*: *assoc commute left-commute*)
    } note *h* = *this*
    from *Cons.hyps*(*1*)[*of ?uf*, *OF h Cons.prems*(*2*)[*simplified*] *p-suc*]
    have *e1*:*P* (*f* (*g a*) (*F* (*map h as*))) (*f* (*g a*) (*F* (*map g as*))) **by** *simp*
    have *e2*:*P* (*f* (*h a*) (*F* (*map h as*))) (*f* (*g a*) (*F* (*map h as*)))
      using *Cons.prems*(*1*)[*OF inl*] **by** *blast*
    from *Cons*(*3*)[*OF e2 e1*] **show** *?case* **by** *auto* **next**
  qed *auto*
end

lemma *prod-induct-gen*:
  assumes $\bigwedge$ *a r. f* (*h a* $*$ *r* :: *'a* :: {*comm-monoid-mult*}) = *f* (*g a* $*$ *r*)
  shows *f* ($\prod$ *v*$\leftarrow$*lst. h v*) = *f* ($\prod$ *v*$\leftarrow$*lst. g v*)

proof − **let** *?P x y* = *f x* = *f y*
show *?thesis* **using** *comm-monoid-mult-class.prod-list.induct-gen-abs*[*of - ?P*,*OF assms*] **by** *auto*
qed

abbreviation *complex-of-int*::*int* ⇒ *complex* **where**
complex-of-int ≡ *of-int*

definition *l2norm-list* :: *int list* ⇒ *int* **where**
l2norm-list lst = ⌊*sqrt* (*sum-list* (*map* (λ *a*. *a* ∗ *a*) *lst*))⌋

abbreviation *l2norm* :: *int poly* ⇒ *int* **where**
l2norm p ≡ *l2norm-list* (*coeffs p*)

abbreviation *norm2 p* ≡ $\sum$ *a*←*coeffs p*. (*cmod a*)^2

abbreviation *l2norm-complex* **where**
l2norm-complex p ≡ *sqrt* (*norm2 p*)

abbreviation *height* :: *int poly* ⇒ *int* **where**
height p ≡ *max-list* (*map* (*nat* ∘ *abs*) (*coeffs p*))

definition *complex-roots-complex* **where**
complex-roots-complex (*p*::*complex poly*) = (*SOME as*. *smult* (*coeff p* (*degree p*)) ($\prod$ *a*←*as*. [:− *a*, *1*:]) = *p* ∧ *length as* = *degree p*)

lemma *complex-roots*:
smult (*lead-coeff p*) ($\prod$ *a*←*complex-roots-complex p*. [:− *a*, *1*:]) = *p*
length (*complex-roots-complex p*) = *degree p*
using *someI-ex*[*OF fundamental-theorem-algebra-factorized*]
unfolding *complex-roots-complex-def* **by** *simp-all*

lemma *complex-roots-c* [*simp*]:
complex-roots-complex [:*c*:] = []
using *complex-roots*(*2*) [*of* [:*c*:]] **by** *simp*

declare *complex-roots*(*2*)[*simp*]

lemma *complex-roots-1* [*simp*]:
complex-roots-complex 1 = []
using *complex-roots-c* [*of 1*] **by** (*simp add*: *pCons-one*)

lemma *linear-term-irreducible*$_d$[*simp*]: *irreducible*$_d$ [: *a*, *1*:]
by (*rule linear-irreducible*$_d$, *simp*)

definition *complex-roots-int* **where**
complex-roots-int (*p*::*int poly*) = *complex-roots-complex* (*map-poly of-int p*)

lemma *complex-roots-int*:

smult (*lead-coeff p*) ($\prod$ *a*←*complex-roots-int p*. [:− *a*, *1*:]) = *map-poly of-int p*
length (*complex-roots-int p*) = *degree p*
proof −
show *smult* (*lead-coeff p*) ($\prod$ *a*←*complex-roots-int p*. [:− *a*, *1*:]) = *map-poly of-int p*
length (*complex-roots-int p*) = *degree p*
using *complex-roots*[*of map-poly of-int p*] **unfolding** *complex-roots-int-def* **by** *auto*
qed

The measure for polynomials, after K. Mahler

definition *mahler-measure-poly* **where**
mahler-measure-poly p = *cmod* (*lead-coeff p*) ∗ ($\prod$ *a*←*complex-roots-complex p*. (*max 1* (*cmod a*)))

definition *mahler-measure* **where**
mahler-measure p = *mahler-measure-poly* (*map-poly complex-of-int p*)

definition *mahler-measure-monic* **where**
mahler-measure-monic p = ($\prod$ *a*←*complex-roots-complex p*. (*max 1* (*cmod a*)))

lemma *mahler-measure-poly-via-monic* :
mahler-measure-poly p = *cmod* (*lead-coeff p*) ∗ *mahler-measure-monic p*
unfolding *mahler-measure-poly-def mahler-measure-monic-def* **by** *simp*

lemma *smult-inj*[*simp*]: **assumes** (*a*::′*a*::*idom*) ≠ *0* **shows** *inj* (*smult a*)
proof−
interpret *map-poly-inj-zero-hom* (∗) *a* **using** *assms* **by** (*unfold-locales*, *auto*)
show *?thesis* **unfolding** *smult-as-map-poly* **by** (*rule inj-f*)
qed

definition *reconstruct-poly*::′*a*::*idom* ⇒ ′*a list* ⇒ ′*a poly* **where**
reconstruct-poly c roots = *smult c* ($\prod$ *a*←*roots*. [:− *a*, *1*:])

lemma *reconstruct-is-original-poly*:
reconstruct-poly (*lead-coeff p*) (*complex-roots-complex p*) = *p*
using *complex-roots*(*1*) **by** (*simp add*: *reconstruct-poly-def*)

lemma *reconstruct-with-type-conversion*:
smult (*lead-coeff* (*map-poly of-int f*)) (*prod-list* (*map* (λ *a*. [:− *a*, *1*:]) (*complex-roots-int f*)))
= *map-poly of-int f*
unfolding *complex-roots-int-def complex-roots*(*1*) **by** *simp*

lemma *reconstruct-prod*:
shows *reconstruct-poly* (*a*::*complex*) *as* ∗ *reconstruct-poly b bs*
= *reconstruct-poly* (*a* ∗ *b*) (*as* @ *bs*)
unfolding *reconstruct-poly-def* **by** *auto*

lemma *linear-term-inj*[*simplified*,*simp*]: *inj* (λ *a*. [:$-$ *a*, *1*::'*a*::*idom*:])
  unfolding *inj-on-def* **by** *simp*

lemma *reconstruct-poly-monic-defines-mset*:
  assumes ($\prod$ *a*$\leftarrow$*as*. [:$-$ *a*, *1*:]) = ($\prod$ *a*$\leftarrow$*bs*. [:$-$ *a*, *1*::'*a*::*field*:])
  shows *mset as* = *mset bs*
proof $-$
  let *?as* = *mset* (*map* (λ *a*. [:$-$ *a*, *1*:]) *as*)
  let *?bs* = *mset* (*map* (λ *a*. [:$-$ *a*, *1*:]) *bs*)
  have *eq-smult*:*prod-mset ?as* = *prod-mset ?bs* **using** *assms* **by** (*metis prod-mset-prod-list*)
  have *irr*:$\bigwedge$ *as*::'*a list*. *set-mset* (*mset* (*map* (λ *a*. [:$-$ *a*, *1*:]) *as*)) $\subseteq$ {*q*. *irreducible*
q $\wedge$ *monic q*}
    by (*auto intro*!: *linear-term-irreducible*$_d$[*of* $-$-::'*a*, *simplified*])
  from *monic-factorization-unique-mset*[*OF eq-smult irr irr*]
  show *?thesis* **apply** (*subst inj-eq*[*OF multiset.inj-map*,*symmetric*]) **by** *auto*
qed

lemma *reconstruct-poly-defines-mset-of-argument*:
  assumes (*a*::'*a*::*field*) $\neq$ *0*
    reconstruct-poly a as = *reconstruct-poly a bs*
  shows *mset as* = *mset bs*
proof $-$
  have *eq-smult*:*smult a* ($\prod$ *a*$\leftarrow$*as*. [:$-$ *a*, *1*:]) = *smult a* ($\prod$ *a*$\leftarrow$*bs*. [:$-$ *a*, *1*:])
    using *assms*(*2*) **by** (*auto simp*:*reconstruct-poly-def*)
  from *reconstruct-poly-monic-defines-mset*[*OF Fun.injD*[*OF smult-inj*[*OF assms*(*1*)]
eq-smult]]
  show *?thesis* **by** *simp*
qed

lemma *complex-roots-complex-prod* [*simp*]:
  assumes *f* $\neq$ *0 g* $\neq$ *0*
  shows *mset* (*complex-roots-complex* (*f* $*$ *g*))
    = *mset* (*complex-roots-complex f*) + *mset* (*complex-roots-complex g*)
proof $-$
  let *?p* = *f* $*$ *g*
  let *?lc v* = (*lead-coeff* (*v*:: *complex poly*))
  have *nonzero-prod*:*?lc ?p* $\neq$ *0* **using** *assms* **by** *auto*
  from *reconstruct-prod*[*of ?lc f complex-roots-complex f ?lc g complex-roots-complex*
g]
  have *reconstruct-poly* (*?lc ?p*) (*complex-roots-complex ?p*)
    = *reconstruct-poly* (*?lc ?p*) (*complex-roots-complex f* @ *complex-roots-complex*
g)
    unfolding *lead-coeff-mult*[*symmetric*] *reconstruct-is-original-poly* **by** *auto*
  from *reconstruct-poly-defines-mset-of-argument*[*OF nonzero-prod this*]
  show *?thesis* **by** *simp*
qed

lemma *mset-mult-add*:
  assumes *mset* (*a*::'*a*::*field list*) = *mset b* + *mset c*

shows *prod-list a* = *prod-list b* ∗ *prod-list c*
unfolding *prod-mset-prod-list*[*symmetric*]
using *prod-mset-Un*[*of mset b mset c*,*unfolded assms*[*symmetric*]]**.**

lemma *mset-mult-add-2*:
assumes *mset a* = *mset b* + *mset c*
shows *prod-list* (*map i a*::′*b*::*field list*) = *prod-list* (*map i b*) ∗ *prod-list* (*map i c*)
proof −
have *r*:*mset* (*map i a*) = *mset* (*map i b*) + *mset* (*map i c*) **using** *assms*
by (*metis map-append mset-append mset-map*)
show *?thesis* **using** *mset-mult-add*[*OF r*] **by** *auto*
qed

lemma *measure-mono-eq-prod*:
assumes *f* ≠ *0 g* ≠ *0*
shows *mahler-measure-monic* (*f* ∗ *g*) = *mahler-measure-monic f* ∗ *mahler-measure-monic g*
unfolding *mahler-measure-monic-def*
using *mset-mult-add-2*[*OF complex-roots-complex-prod*[*OF assms*],*of* λ *a*. *max 1* (*cmod a*)] **by** *simp*

lemma *mahler-measure-poly-0*[*simp*]: *mahler-measure-poly 0* = *0* **unfolding** *mahler-measure-poly-via-monic* **by** *auto*

lemma *measure-eq-prod*:
mahler-measure-poly (*f* ∗ *g*) = *mahler-measure-poly f* ∗ *mahler-measure-poly g*
proof −
consider *f* = *0* | *g* = *0* | (*both*) *f* ≠ *0 g* ≠ *0* **by** *auto*
thus *?thesis* **proof**(*cases*)
case *both* **show** *?thesis* **unfolding** *mahler-measure-poly-via-monic norm-mult lead-coeff-mult*
by (*auto simp*: *measure-mono-eq-prod*[*OF both*])
qed (*simp-all*)
qed

lemma *prod-cmod*[*simp*]:
cmod (∏ *a*←*lst*. *f a*) = (∏ *a*←*lst*. *cmod* (*f a*))
by(*induct lst*,*auto simp*:*real-normed-div-algebra-class.norm-mult*)

lemma *lead-coeff-of-prod*[*simp*]:
lead-coeff (∏ *a*←*lst*. *f a*::′*a*::*idom poly*) = (∏ *a*←*lst*. *lead-coeff* (*f a*))
by(*induct lst*,*auto simp*:*lead-coeff-mult*)

lemma *ineq-about-squares*:**assumes** *x* ≤ (*y*::*real*) **shows** *x* ≤ *c*ˆ*2* + *y* **using** *assms*
by (*simp add*: *add.commute add-increasing2*)

lemma *first-coeff-le-tail*:(*cmod* (*lead-coeff g*))ˆ*2* ≤ (∑ *a*←*coeffs g*. (*cmod a*)ˆ*2*)
proof(*induct g*)
case (*pCons a p*)

thus *?case* **proof**(*cases p = 0*) **case** *False*
show *?thesis* **using** *pCons* **unfolding** *lead-coeff-pCons*(*1*)[*OF False*]
by(*cases a = 0*,*simp-all add*:*ineq-about-squares*)
qed *simp*
qed *simp*

lemma *square-prod-cmod*[*simp*]:
(*cmod* (*a* ∗ *b*))^*2* = *cmod a* ^ *2* ∗ *cmod b* ^ *2*
by (*simp add*: *norm-mult power-mult-distrib*)

lemma *sum-coeffs-smult-cmod*:
($\sum$ *a*←*coeffs* (*smult v p*). (*cmod a*)^*2*) = (*cmod v*)^*2* ∗ ($\sum$ *a*←*coeffs p*. (*cmod a*)^*2*)
(**is** *?l* = *?r*)
proof −
have *?l* = ($\sum$ *a*←*coeffs p*. (*cmod v*)^*2* ∗ (*cmod a*)^*2*) **by**(*cases v*=*0*;*induct p*,*auto*)
thus *?thesis* **by** (*auto simp*:*sum-list-const-mult*)
qed

abbreviation *linH a* ≡ *if* (*cmod a* > *1*) *then* [:− *1*,*cnj a*:] *else* [:− *a*,*1*:]

lemma *coeffs-cong-1*[*simp*]: *cCons a v* = *cCons b v* ⟷ *a* = *b* **unfolding** *cCons-def*
by *auto*

lemma *strip-while-singleton*[*simp*]:
strip-while ((=) *0*) [*v* ∗ *a*] = *cCons* (*v* ∗ *a*) [] **unfolding** *cCons-def strip-while-def*
by *auto*

lemma *coeffs-times-linterm*:
shows *coeffs* (*pCons 0* (*smult a p*) + *smult b p*) = *strip-while* (*HOL.eq* (*0*::'*a*::{*comm-ring-1*}))
(*map* (λ(*c*,*d*).*b*∗*d*+*c*∗*a*) (*zip* (*0* # *coeffs p*) (*coeffs p* @ [*0*]))) **proof** −
{**fix** *v*
have *coeffs* (*smult b p* + *pCons* (*a*∗ *v*) (*smult a p*)) = *strip-while* (*HOL.eq 0*) (*map* (λ(*c*,*d*).*b*∗*d*+*c*∗*a*) (*zip* ([*v*] @ *coeffs p*) (*coeffs p* @ [*0*])))
proof(*induct p arbitrary*:*v*) **case** (*pCons pa ps*) **thus** *?case* **by** *auto* **qed** *auto*
}
from *this*[*of 0*] **show** *?thesis* **by** (*simp add*: *add.commute*)
qed

lemma *filter-distr-rev*[*simp*]:
shows *filter f* (*rev lst*) = *rev* (*filter f lst*)
by(*induct lst*;*auto*)

lemma *strip-while-filter*:
shows *filter* ((≠) *0*) (*strip-while* ((=) *0*) (*lst*::'*a*::*zero list*)) = *filter* ((≠) *0*) *lst*
proof − {**fix** *lst*::'*a list*
have *filter* ((≠) *0*) (*dropWhile* ((=) *0*) *lst*) = *filter* ((≠) *0*) *lst* **by** (*induct*

lst;*auto*)
  hence (*filter* $((\neq)\ 0)$ (*strip-while* $((=)\ 0)$ (*rev lst*))) = *filter* $((\neq)\ 0)$ (*rev lst*)
  unfolding *strip-while-def* **by**(*simp*)**}**
  from *this*[*of rev lst*] **show** *?thesis* **by** *simp*
qed

lemma *sum-stripwhile*[*simp*]:
  assumes $f\ 0 = 0$
  shows $(\sum a \leftarrow strip\text{-}while\ ((=)\ 0)\ lst.\ f\ a) = (\sum a \leftarrow lst.\ f\ a)$
proof −
  {fix *lst*
    have $(\sum a \leftarrow filter\ ((\neq)\ 0)\ lst.\ f\ a) = (\sum a \leftarrow lst.\ f\ a)$ **by**(*induct lst*,*auto simp*:*assms*)**}**
  note *f*=*this*
  have *sum-list* (*map f* (*filter* $((\neq)\ 0)$ (*strip-while* $((=)\ 0)$ *lst*)))
     = *sum-list* (*map f* (*filter* $((\neq)\ 0)$ *lst*))
  using *strip-while-filter*[*of lst*] **by**(*simp*)
  thus *?thesis* **unfolding** *f*.
qed

lemma *complex-split* : $Complex\ a\ b = c \longleftrightarrow (a = Re\ c \land b = Im\ c)$
  using *complex-surj* **by** *auto*

lemma *norm-times-const*:$(\sum y \leftarrow lst.\ (cmod\ (a * y))^2) = (cmod\ a)^2 * (\sum y \leftarrow lst.\ (cmod\ y)^2)$
by(*induct lst*,*auto simp*:*ring-distribs*)

fun *bisumTail* **where**
  bisumTail f (*Cons a* (*Cons b bs*)) = *f a b* + *bisumTail f* (*Cons b bs*) |
  bisumTail f (*Cons a Nil*) = *f a 0* |
  bisumTail f Nil = *f 1 0*
fun *bisum* **where**
  bisum f (*Cons a as*) = *f 0 a* + *bisumTail f* (*Cons a as*) |
  bisum f Nil = *f 0 0*

lemma *bisumTail-is-map-zip*:
  $(\sum x \leftarrow zip\ (v\ \#\ l1)\ (l1\ @\ [0]).\ f\ x) = bisumTail\ (\lambda x\ y\ .f\ (x,y))\ \ (v\#l1)$
by(*induct l1 arbitrary*:*v*,*auto*)

lemma *bisum-is-map-zip*:
  $(\sum x \leftarrow zip\ (0\ \#\ l1)\ (l1\ @\ [0]).\ f\ x) = bisum\ (\lambda x\ y.\ f\ (x,y))\ l1$
using *bisumTail-is-map-zip*[*of f hd l1 tl l1*] **by**(*cases l1*,*auto*)
lemma *map-zip-is-bisum*:
  $bisum\ f\ l1 = (\sum (x,y) \leftarrow zip\ (0\ \#\ l1)\ (l1\ @\ [0]).\ f\ x\ y)$
using *bisum-is-map-zip*[*of* $\lambda(x,y).\ f\ x\ y$] **by** *auto*

lemma *bisum-outside* :
  $(bisum\ (\lambda\ x\ y.\ f1\ x - f2\ x\ y + f3\ y)\ lst :: 'a :: field)$
  $= sum\text{-}list\ (map\ f1\ lst) + f1\ 0 - bisum\ f2\ lst + sum\text{-}list\ (map\ f3\ lst) + f3\ 0$

proof(*cases lst*)
 case (*Cons a lst*) **show** *?thesis* **unfolding** *map-zip-is-bisum Cons* **by**(*induct lst arbitrary*:*a*,*auto*)
qed *auto*

lemma *Landau-lemma*:
 $(\sum a{\leftarrow}coeffs\ (\prod a{\leftarrow}lst.\ [{:}-\ a,\ 1{:}]).\ (cmod\ a)^2) = (\sum a{\leftarrow}coeffs\ (\prod a{\leftarrow}lst.\ linH\ a).\ (cmod\ a)^2)$
 (**is** *norm2 ?l* = *norm2 ?r*)
proof −
 have *a*:$\bigwedge$ *a*. $(cmod\ a)^2 = Re\ (a * cnj\ a)$ **using** *complex-norm-square*
  unfolding *complex-split complex-of-real-def* **by** *simp*
 have *b*:$\bigwedge$ *x a y*. $(cmod\ (x - a * y))\hat{}2$
  $= (cmod\ x)^2 - Re\ (a * y * cnj\ x + x * cnj\ (a * y)) + (cmod\ (a * y))\hat{}2$
  unfolding *left-diff-distrib right-diff-distrib a complex-cnj-diff* **by** *simp*
 have *c*:$\bigwedge$ *y a x*. $(cmod\ (cnj\ a * x - y))^2$
  $= (cmod\ (a * x))^2 - Re\ (a * y * cnj\ x + x * cnj\ (a * y)) + (cmod\ y)\hat{}2$
  unfolding *left-diff-distrib right-diff-distrib a complex-cnj-diff*
  by (*simp add*: *mult.assoc mult.left-commute*)
 { fix *f1 a*
  have *norm2* $([{:}-\ a,\ 1\ {:}] * f1)$ = *bisum* $(\lambda x\ y.\ cmod\ (x - a * y)\hat{}2)$ (*coeffs f1*)
   by(*simp add*: *bisum-is-map-zip*[*of - coeffs f1*] *coeffs-times-linterm*[*of 1 - −a*,*simplified*])
  also have . . . = *norm2 f1* + *cmod a*ˆ*2*∗*norm2 f1*
   − *bisum* $(\lambda x\ y.\ Re\ (a * y * cnj\ x + x * cnj\ (a * y)))$ (*coeffs f1*)
   unfolding *b bisum-outside norm-times-const* **by** *simp*
  also have . . . = *bisum* $(\lambda x\ y.\ cmod\ (cnj\ a * x - y)\hat{}2)$ (*coeffs f1*)
   unfolding *c bisum-outside norm-times-const* **by** *auto*
  also have . . . = *norm2* $([{:}-\ 1,\ cnj\ a\ {:}] * f1)$
   using *coeffs-times-linterm*[*of cnj a - −1*]
   by(*simp add*: *bisum-is-map-zip*[*of - coeffs f1*] *mult.commute*)
  finally have *norm2* $([{:}-\ a,\ 1\ {:}] * f1)$ = ...**}**
 hence *h*:$\bigwedge$ *a f1*. *norm2* $([{:}-\ a,\ 1\ {:}] * f1)$ = *norm2* (*linH a* ∗ *f1*) **by** *auto*
 show *?thesis* **by**(*rule prod-induct-gen*[*OF h*])
qed

lemma *Landau-inequality*:
 mahler-measure-poly f ≤ *l2norm-complex f*
proof −
 let *?f* = *reconstruct-poly* (*lead-coeff f*) (*complex-roots-complex f*)
 let *?roots* = (*complex-roots-complex f*)
 let *?g* = $\prod$ *a*←*?roots*. *linH a*

 have *max*:$\bigwedge$*a*. *cmod* (*if 1* < *cmod a then cnj a else 1*) = *max 1* (*cmod a*)
  by *simp*
 have $\bigwedge$*a*. *1* < *cmod a* $\Longrightarrow$ *a* ≠ *0* **by** *auto*
 hence $\bigwedge$*a*. *lead-coeff* (*linH a*) = (*if* (*cmod a* > *1*) *then cnj a else 1*) **by**(*auto*

simp:*if-split*)
hence *lead-coeff-g*:*cmod* (*lead-coeff ?g*) = ($\prod$ *a*←*?roots*. *max 1* (*cmod a*)) **by**(*auto simp*:*max*)

have *norm2 f* = ($\sum$ *a*←*coeffs ?f*. (*cmod a*)^*2*) **unfolding** *reconstruct-is-original-poly*..
also have ... = *cmod* (*lead-coeff f*)^*2* ∗ ($\sum$ *a*←*coeffs* ($\prod$ *a*←*?roots*. [:− *a*, *1*:]). (*cmod a*)^2)
unfolding *reconstruct-poly-def* **using** *sum-coeffs-smult-cmod*.
finally have *fg-norm*:*norm2 f* = *cmod* (*lead-coeff f*)^*2* ∗ ($\sum$ *a*←*coeffs ?g*. (*cmod a*)^*2*)
unfolding *Landau-lemma* **by** *auto*

have (*cmod* (*lead-coeff ?g*))^*2* ≤ ($\sum$ *a*←*coeffs ?g*. (*cmod a*)^*2*)
using *first-coeff-le-tail* **by** *blast*
from *ordered-comm-semiring-class.comm-mult-left-mono*[*OF this*]
have (*cmod* (*lead-coeff f*) ∗ *cmod* (*lead-coeff ?g*))^*2* ≤ ($\sum$ *a*←*coeffs f*. (*cmod a*)^*2*)
unfolding *fg-norm* **by** (*simp add*:*power-mult-distrib*)
hence *cmod* (*lead-coeff f*) ∗ ($\prod$ *a*←*?roots*. *max 1* (*cmod a*)) ≤ *sqrt* (*norm2 f*)
using *NthRoot.real-le-rsqrt lead-coeff-g* **by** *auto*
thus *mahler-measure-poly f* ≤ *sqrt* (*norm2 f*)
using *reconstruct-with-type-conversion*[*unfolded complex-roots-int-def*]
by (*simp add*: *mahler-measure-poly-via-monic mahler-measure-monic-def complex-roots-int-def*)
qed

lemma *prod-list-ge1*:
assumes *Ball* (*set x*) (λ (*a*::*real*). *a* ≥ *1*)
shows *prod-list x* ≥ *1*
using *assms* **proof**(*induct x*)
case (*Cons a as*)
have ∀ *a*∈*set as*. *1* ≤ *a 1* ≤ *a* **using** *Cons*(*2*) **by** *auto*
thus *?case* **using** *Cons.hyps mult-mono′* **by** *fastforce*
qed *auto*

lemma *mahler-measure-monic-ge-1*: *mahler-measure-monic p* ≥ *1*
unfolding *mahler-measure-monic-def* **by**(*rule prod-list-ge1*,*simp*)

lemma *mahler-measure-monic-ge-0*: *mahler-measure-monic p* ≥ *0*
using *mahler-measure-monic-ge-1 le-numeral-extra*(*1*) *order-trans* **by** *blast*

lemma *mahler-measure-ge-0*: *0* ≤ *mahler-measure h* **unfolding** *mahler-measure-def mahler-measure-poly-via-monic*
by (*simp add*: *mahler-measure-monic-ge-0*)

lemma *mahler-measure-constant*[*simp*]: *mahler-measure-poly* [:*c*:] = *cmod c*
proof −
have *main*: *complex-roots-complex* [:*c*:] = [] **unfolding** *complex-roots-complex-def*
by (*rule some-equality*, *auto*)

show *?thesis* **unfolding** *mahler-measure-poly-def main* **by** *auto*
qed

lemma *mahler-measure-factor*[*simplified*,*simp*]: *mahler-measure-poly* [:− *a*, *1*:] = *max 1* (*cmod a*)
proof −
have *main*: *complex-roots-complex* [:− *a*, *1*:] = [*a*] **unfolding** *complex-roots-complex-def*
proof (*rule some-equality*, *auto*, *goal-cases*)
case (*1 as*)
thus *?case* **by** (*cases as*, *auto*)
qed
show *?thesis* **unfolding** *mahler-measure-poly-def main* **by** *auto*
qed

lemma *mahler-measure-poly-explicit*: *mahler-measure-poly* (*smult c* (∏ *a*←*as*. [:− *a*, *1*:]))
= *cmod c* ∗ (∏ *a*←*as*. (*max 1* (*cmod a*)))
proof (*cases c* = *0*)
case *True*
thus *?thesis* **by** *auto*
next
case *False* **note** *c* = *this*
show *?thesis*
proof (*induct as*)
case (*Cons a as*)
have *mahler-measure-poly* (*smult c* (∏ *a*←*a* # *as*. [:− *a*, *1*:]))
= *mahler-measure-poly* (*smult c* (∏ *a*←*as*. [:− *a*, *1*:]) ∗ [: −*a*, *1* :])
by (*rule arg-cong*[*of* - - *mahler-measure-poly*], *unfold list.simps prod-list.Cons mult-smult-left*, *simp*)
also have . . . = *mahler-measure-poly* (*smult c* (∏ *a*←*as*. [:− *a*, *1*:])) ∗ *mahler-measure-poly* ([:− *a*, *1*:])
(**is** - = *?l* ∗ *?r*) **by** (*rule measure-eq-prod*)
also have *?l* = *cmod c* ∗ (∏ *a*←*as*. *max 1* (*cmod a*)) **unfolding** *Cons* **by** *simp*
also have *?r* = *max 1* (*cmod a*) **by** *simp*
finally show *?case* **by** *simp*
next
case *Nil*
show *?case* **by** *simp*
qed
qed

lemma *mahler-measure-poly-ge-1*:
assumes *h* ≠ *0*
shows (*1*::*real*) ≤ *mahler-measure h*
proof −
have *rc*: |*real-of-int i*| = *of-int* |*i*| **for** *i* **by** *simp*
from *assms* **have** *cmod* (*lead-coeff* (*map-poly complex-of-int h*)) > *0* **by** *simp*
hence *cmod* (*lead-coeff* (*map-poly complex-of-int h*)) ≥ *1*
by(*cases lead-coeff h* = *0*, *auto simp del*: *leading-coeff-0-iff*)

from *mult-mono*[*OF this mahler-measure-monic-ge-1 norm-ge-zero*]
show *?thesis* **unfolding** *mahler-measure-def mahler-measure-poly-via-monic*
by *auto*
qed

lemma *mahler-measure-dvd*: **assumes** $f \neq 0$ **and** *h dvd f*
shows *mahler-measure h* $\leq$ *mahler-measure f*
proof −
from *assms* **obtain** *g* **where** *f*: $f = g * h$ **unfolding** *dvd-def* **by** *auto*
from *f assms* **have** *g0*: $g \neq 0$ **by** *auto*
hence *mg*: *mahler-measure g* ≥ 1 **by** (*rule mahler-measure-poly-ge-1*)
have $1 *$ *mahler-measure h* $\leq$ *mahler-measure f*
unfolding *mahler-measure-def f measure-eq-prod*
of-int-poly-hom.hom-mult **unfolding** *mahler-measure-def*[*symmetric*]
by (*rule mult-right-mono*[*OF mg mahler-measure-ge-0*])
thus *?thesis* **by** *simp*
qed

definition *graeffe-poly* :: $'a \Rightarrow 'a$ :: *comm-ring-1 list* $\Rightarrow$ *nat* $\Rightarrow$ *'a poly* **where**
graeffe-poly c as m = *smult* (*c* ^ (*2*^*m*)) ($\prod$ *a*←*as*. [:− (*a* ^ (*2*^*m*)), *1*:])

context
fixes *f* :: *complex poly* **and** *c as*
assumes *f*: *f* = *smult c* ($\prod$ *a*←*as*. [:− *a*, *1*:])
begin
lemma *mahler-graeffe*: *mahler-measure-poly* (*graeffe-poly c as m*) = (*mahler-measure-poly f*)^(*2*^*m*)
proof −
have *graeffe*: *graeffe-poly c as m* = *smult* (*c* ^ *2* ^ *m*) ($\prod$ *a*←(*map* (λ *a*. *a* ^ *2* ^ *m*) *as*). [:− *a*, *1*:])
unfolding *graeffe-poly-def*
by (*rule arg-cong*[*of - - smult* (*c* ^ *2* ^ *m*)], *induct as*, *auto*)
{
fix *n* :: *nat*
assume *n*: *n* > *0*
have *id*: *max 1* (*cmod a* ^ *n*) = *max 1* (*cmod a*) ^ *n* **for** *a*
proof (*cases cmod a* $\leq$ *1*)
case *True*
hence *cmod a* ^ *n* $\leq$ *1* **by** (*simp add*: *power-le-one*)
with *True* **show** *?thesis* **by** (*simp add*: *max-def*)
qed (*auto simp*: *max-def*)
have ($\prod$ *x*←*as*. *max 1* (*cmod x* ^ *n*)) = ($\prod$ *a*←*as*. *max 1* (*cmod a*)) ^ *n*
by (*induct as*, *auto simp*: *field-simps n id*)
}
thus *?thesis* **unfolding** *f mahler-measure-poly-explicit graeffe*
by (*auto simp*: *o-def field-simps norm-power*)
qed

end

fun *drop-half* :: *'a list* ⇒ *'a list* **where**
  drop-half (*x* # *y* # *ys*) = *x* # *drop-half ys*
| *drop-half xs* = *xs*

fun *alternate* :: *'a list* ⇒ *'a list* × *'a list* **where**
  alternate (*x* # *y* # *ys*) = (*case alternate ys of* (*evn*, *od*) ⇒ (*x* # *evn*, *y* # *od*))
| *alternate xs* = (*xs*,[])

definition *poly-square-subst* :: *'a* :: *comm-ring-1 poly* ⇒ *'a poly* **where**
  poly-square-subst f = *poly-of-list* (*drop-half* (*coeffs f*))

definition *poly-even-odd* :: *'a* :: *comm-ring-1 poly* ⇒ *'a poly* × *'a poly* **where**
  poly-even-odd f = (*case alternate* (*coeffs f*) *of* (*evn*,*od*) ⇒ (*poly-of-list evn*, *poly-of-list od*))

lemma *poly-square-subst-coeff*: *coeff* (*poly-square-subst f*) *i* = *coeff f* (*2* ∗ *i*)
proof −
  have *id*: *coeff f* (*2* ∗ *i*) = *coeff* (*Poly* (*coeffs f*)) (*2* ∗ *i*) **by** *simp*
  obtain *xs* **where** *xs*: *coeffs f* = *xs* **by** *auto*
  show *?thesis* **unfolding** *poly-square-subst-def poly-of-list-def coeff-Poly-eq id xs*
  proof (*induct xs arbitrary*: *i rule*: *drop-half.induct*)
    case (*1 x y ys i*) **thus** *?case* **by** (*cases i*, *auto*)
  next
    case (*2-2 x i*) **thus** *?case* **by** (*cases i*, *auto*)
  qed *auto*
qed

lemma *poly-even-odd-coeff*: **assumes** *poly-even-odd f* = (*ev*,*od*)
  shows *coeff ev i* = *coeff f* (*2* ∗ *i*) *coeff od i* = *coeff f* (*2* ∗ *i* + *1*)
proof −
  have *id*: ⋀ *i*. *coeff f i* = *coeff* (*Poly* (*coeffs f*)) *i* **by** *simp*
  obtain *xs* **where** *xs*: *coeffs f* = *xs* **by** *auto*
  from *assms*[*unfolded poly-even-odd-def*]
  have *ev-od*: *ev* = *Poly* (*fst* (*alternate xs*)) *od* = *Poly* (*snd* (*alternate xs*))
    by (*auto simp*: *xs split*: *prod.splits*)
  have *coeff ev i* = *coeff f* (*2* ∗ *i*) ∧ *coeff od i* = *coeff f* (*2* ∗ *i* + *1*)
    unfolding *poly-of-list-def coeff-Poly-eq id xs ev-od*
  proof (*induct xs arbitrary*: *i rule*: *alternate.induct*)
    case (*1 x y ys i*) **thus** *?case* **by** (*cases alternate ys*; *cases i*, *auto*)
  next
    case (*2-2 x i*) **thus** *?case* **by** (*cases i*, *auto*)
  qed *auto*
  thus *coeff ev i* = *coeff f* (*2* ∗ *i*) *coeff od i* = *coeff f* (*2* ∗ *i* + *1*) **by** *auto*
qed

lemma *poly-square-subst*: *poly-square-subst* (*f* $\circ_p$ (*monom 1 2*)) = *f* by (*rule poly-eqI*, *unfold poly-square-subst-coeff*, *subst coeff-pcompose-x-pow-n*, *auto*)

lemma *poly-even-odd*: **assumes** *poly-even-odd f* = (*g*,*h*)
shows *f* = *g* $\circ_p$ *monom 1 2* + *monom 1 1* ∗ (*h* $\circ_p$ *monom 1 2*)
proof −
note *id* = *poly-even-odd-coeff*[*OF assms*]
show *?thesis*
proof (*rule poly-eqI*, *unfold coeff-add coeff-monom-mult*)
fix *n* :: *nat*
obtain *m i* **where** *mi*: *m* = *n div 2 i* = *n mod 2* **by** *auto*
have *nmi*: *n* = *2* ∗ *m* + *i i* < *2 0* < (*2* :: *nat*) *1* < (*2* :: *nat*) **unfolding** *mi* **by** *auto*
have (*2* :: *nat*) ≠ *0* **by** *auto*
show *coeff f n* = *coeff* (*g* $\circ_p$ *monom 1 2*) *n* + (*if 1* ≤ *n then 1* ∗ *coeff* (*h* $\circ_p$ *monom 1 2*) (*n* − *1*) *else 0*)
proof (*cases i* = *1*)
case *True*
hence *id1*: *2* ∗ *m* + *i* − *1* = *2* ∗ *m* + *0* **by** *auto*
show *?thesis* **unfolding** *nmi id id1 coeff-pcompose-monom*[*OF nmi*(*2*)] *coeff-pcompose-monom*[*OF nmi*(*3*)]
unfolding *True* **by** *auto*
next
case *False*
with *nmi* **have** *i0*: *i* = *0* **by** *auto*
show *?thesis*
proof (*cases m*)
case (*Suc k*)
hence *id1*: *2* ∗ *m* + *i* − *1* = *2* ∗ *k* + *1* **using** *i0* **by** *auto*
show *?thesis* **unfolding** *nmi id coeff-pcompose-monom*[*OF nmi*(*2*)]
coeff-pcompose-monom[*OF nmi*(*4*)] *id1* **unfolding** *Suc i0* **by** *auto*
next
case *0*
show *?thesis* **unfolding** *nmi id coeff-pcompose-monom*[*OF nmi*(*2*)] **unfolding** *i0 0* **by** *auto*
qed
qed
qed
qed

context
fixes *f* :: *'a* :: *idom poly*
begin

lemma *graeffe-0*: *f* = *smult c* (∏ *a*←*as*. [:− *a*, *1*:]) ⟹ *graeffe-poly c as 0* = *f*
unfolding *graeffe-poly-def* **by** *auto*

lemma *graeffe-recursion*: **assumes** *graeffe-poly c as m* = *f*
shows *graeffe-poly c as* (*Suc m*) = *smult* ((−*1*)^(*degree f*)) (*poly-square-subst* (*f*

∗ *f* ∘ₚ [:*0*,−*1*:]))
proof −
  let *?g* = *graeffe-poly c as m*
  have *f* ∗ *f* ∘ₚ [:*0*,−*1*:] = *?g* ∗ *?g* ∘ₚ [:*0*,−*1*:] **unfolding** *assms* **by** *simp*
  also have *?g* ∘ₚ [:*0*,−*1*:] = *smult* ((− *1*) ^ *length as*) (*smult* (*c* ^ *2* ^ *m*) (∏ *a*←*as*. [:*a* ^ *2* ^ *m*, *1*:]))
    unfolding *graeffe-poly-def*
  proof (*induct as*)
    case (*Cons a as*)
    have *?case* = ((*smult* (*c* ^ *2* ^ *m*) ([:− (*a* ^ *2* ^ *m*), *1*:] ∘ₚ [:*0*, − *1*:] ∗ (∏ *a*←*as*. [:− (*a* ^ *2* ^ *m*), *1*:]) ∘ₚ [:*0*, − *1*:]) =
      smult (−*1* ∗ (− *1*) ^ *length as*)
      (*smult* (*c* ^ *2* ^ *m*) ([: *a* ^ *2* ^ *m*, *1*:] ∗ (∏ *a*←*as*. [:*a* ^ *2* ^ *m*, *1*:])))))
      unfolding *list.simps prod-list.Cons pcompose-smult pcompose-mult* **by** *simp*
    also have *smult* (*c* ^ *2* ^ *m*) ([:− (*a* ^ *2* ^ *m*), *1*:] ∘ₚ [:*0*, − *1*:] ∗ (∏ *a*←*as*. [:− (*a* ^ *2* ^ *m*), *1*:]) ∘ₚ [:*0*, − *1*:])
      = *smult* (*c* ^ *2* ^ *m*) ((∏ *a*←*as*. [:− (*a* ^ *2* ^ *m*), *1*:]) ∘ₚ [:*0*, − *1*:]) ∗ [:− (*a* ^ *2* ^ *m*), *1*:] ∘ₚ [:*0*, − *1*:]
      unfolding *mult-smult-left* **by** *simp*
    also have *smult* (*c* ^ *2* ^ *m*) ((∏ *a*←*as*. [:− (*a* ^ *2* ^ *m*), *1*:]) ∘ₚ [:*0*, − *1*:]) =
      smult ((− *1*) ^ *length as*) (*smult* (*c* ^ *2* ^ *m*) (∏ *a*←*as*. [:*a* ^ *2* ^ *m*, *1*:]))
      unfolding *pcompose-smult*[*symmetric*] *Cons* **..**
    also have [:− (*a* ^ *2* ^ *m*), *1*:] ∘ₚ [:*0*, − *1*:] = *smult* (−*1*) [: *a*^*2*^*m*, *1*:] **by** *simp*
    finally have *id*: *?case* = (*smult* ((− *1*) ^ *length as*) (*smult* (*c* ^ *2* ^ *m*) (∏ *a*←*as*. [:*a* ^ *2* ^ *m*, *1*:])) ∗ *smult* (− *1*) [:*a* ^ *2* ^ *m*, *1*:] =
      smult (− *1* ∗ (− *1*) ^ *length as*) (*smult* (*c* ^ *2* ^ *m*) ([:*a* ^ *2* ^ *m*, *1*:] ∗ (∏ *a*←*as*. [:*a* ^ *2* ^ *m*, *1*:])))) **by** *simp*
    obtain *c d* **where** *id'*: (∏ *a*←*as*. [:*a* ^ *2* ^ *m*, *1*:]) = *c* [:*a* ^ *2* ^ *m*, *1*:] = *d* **by** *auto*
    show *?case* **unfolding** *id* **unfolding** *id'* **by** (*simp add*: *ac-simps*)
  qed *simp*
  finally have *f* ∗ *f* ∘ₚ [:*0*, − *1*:] =
    smult ((− *1*) ^ *length as* ∗ (*c* ^ *2* ^ *m* ∗ *c* ^ *2* ^ *m*))
    ((∏ *a*←*as*. [:− (*a* ^ *2* ^ *m*), *1*:]) ∗ (∏ *a*←*as*. [:*a* ^ *2* ^ *m*, *1*:]))
    unfolding *graeffe-poly-def* **by** (*simp add*: *ac-simps*)
  also have *c* ^ *2* ^ *m* ∗ *c* ^ *2* ^ *m* = *c* ^ *2* ^ (*Suc m*) **by** (*simp add*: *semiring-normalization-rules*(*36*))
  also have (∏ *a*←*as*. [:− (*a* ^ *2* ^ *m*), *1*:]) ∗ (∏ *a*←*as*. [:*a* ^ *2* ^ *m*, *1*:]) =
    (∏ *a*←*as*. [:− (*a* ^ *2* ^ (*Suc m*)), *1*:]) ∘ₚ *monom 1 2*
  proof (*induct as*)
    case (*Cons a as*)
    have *id*: (*monom 1 2* :: *'a poly*) = [:*0*,*0*,*1*:]
      by (*metis monom-altdef pCons-0-as-mult power2-eq-square smult-1-left*)
    have (∏ *a*←*a* # *as*. [:− (*a* ^ *2* ^ *m*), *1*:]) ∗ (∏ *a*←*a* # *as*. [:*a* ^ *2* ^ *m*, *1*:])
      = ([:− (*a* ^ *2* ^ *m*), *1*:] ∗ [: *a* ^ *2* ^ *m*, *1*:]) ∗ ((∏ *a*← *as*. [:− (*a* ^ *2* ^ *m*), *1*:]) ∗ (∏ *a*← *as*. [:*a* ^ *2* ^ *m*, *1*:]))
        (**is** - = *?a* ∗ *?b*)
      unfolding *list.simps prod-list.Cons* **by** (*simp only*: *ac-simps*)

also have *?b* = ($\prod$*a*←*as*. [:− (*a* ^ *2* ^ *Suc m*), *1*:]) $\circ_p$ *monom 1 2* **unfolding** *Cons* **by** *simp*
also have *?a* = [: − (*a* ^ *2* ^ (*Suc m*)), *0* , *1*:] **by** (*simp add*: *semiring-normalization-rules*(*36*))
also have ... = [: − (*a* ^ *2* ^ (*Suc m*)), *1*:] $\circ_p$ *monom 1 2* **by** (*simp add*: *id*)
also have [: − (*a* ^ *2* ^ (*Suc m*)), *1*:] $\circ_p$ *monom 1 2* ∗ ($\prod$*a*←*as*. [:− (*a* ^ *2* ^ *Suc m*), *1*:]) $\circ_p$ *monom 1 2* =
($\prod$*a*←*a* # *as*. [:− (*a* ^ *2* ^ *Suc m*), *1*:]) $\circ_p$ *monom 1 2* **unfolding** *pcompose-mult*[*symmetric*] **by** *simp*
finally show *?case* **.**
qed *simp*
finally have *f* ∗ *f* $\circ_p$ [:*0*, − *1*:] = (*smult* ((− *1*) ^ *length as*) (*graeffe-poly c as* (*Suc m*)) $\circ_p$ *monom 1 2*)
unfolding *graeffe-poly-def pcompose-smult* **by** *simp*
from *arg-cong*[*OF this*, *of* λ *f*. *smult* ((− *1*) ^ *length as*) (*poly-square-subst f*), *unfolded poly-square-subst*]
have *graeffe-poly c as* (*Suc m*) = *smult* ((− *1*) ^ *length as*) (*poly-square-subst* (*f* ∗ *f* $\circ_p$ [:*0*, − *1*:])) **by** *simp*
also have ... = *smult* ((− *1*) ^ *degree f*) (*poly-square-subst* (*f* ∗ *f* $\circ_p$ [:*0*, − *1*:]))

proof (*cases f* = *0*)
case *True*
thus *?thesis* **by** (*auto simp*: *poly-square-subst-def*)
next
case *False*
with *assms* **have** *c0*: *c* ≠ *0* **unfolding** *graeffe-poly-def* **by** *auto*
from *arg-cong*[*OF assms*, *of degree*]
have *degree f* = *degree* (*smult* (*c* ^ *2* ^ *m*) ($\prod$*a*←*as*. [:− (*a* ^ *2* ^ *m*), *1*:])) **unfolding** *graeffe-poly-def* **by** *auto*
also have ... = *degree* ($\prod$*a*←*as*. [:− (*a* ^ *2* ^ *m*), *1*:]) **unfolding** *degree-smult-eq* **using** *c0* **by** *auto*
also have ... = *length as* **unfolding** *degree-linear-factors* **by** *simp*
finally show *?thesis* **by** *simp*
qed
finally show *?thesis* **.**
qed
end

definition *graeffe-one-step* :: *'a* ⇒ *'a* :: *idom poly* ⇒ *'a poly* **where**
graeffe-one-step c f = *smult c* (*poly-square-subst* (*f* ∗ *f* $\circ_p$ [:*0*,−*1*:]))

lemma *graeffe-one-step-code*[*code*]: **fixes** *c* :: *'a* :: *idom*
shows *graeffe-one-step c f* = (*case poly-even-odd f of* (*g*,*h*)
⇒ *smult c* (*g* ∗ *g* − *monom 1 1* ∗ *h* ∗ *h*))
proof −
obtain *g h* **where** *eo*: *poly-even-odd f* = (*g*,*h*) **by** *force*
from *poly-even-odd*[*OF eo*] **have** *fgh*: *f* = *g* $\circ_p$ *monom 1 2* + *monom 1 1* ∗ *h* $\circ_p$ *monom 1 2* **by** *auto*
have *m2*: *monom* (*1* :: *'a*) *2* = [:*0*,*0*,*1*:] *monom* (*1* :: *'a*) *1* = [:*0*,*1*:]

unfolding *coeffs-eq-iff coeffs-monom*
by (*auto simp add*: *numeral-2-eq-2*)
show *?thesis* **unfolding** *eo split graeffe-one-step-def*
proof (*rule arg-cong*[*of - - smult c*])
let $?g = g \circ_p$ *monom 1 2*
let $?h = h \circ_p$ *monom 1 2*
let $?x$ = *monom* (*1* :: *'a*) *1*
have *2*: *2* = *Suc* (*Suc 0*) **by** *simp*
have $f * f \circ_p [:0, -1:] = (g \circ_p$ *monom 1 2* + *monom 1 1* $* h \circ_p$ *monom 1 2*) $*$
($g \circ_p$ *monom 1 2* + *monom 1 1* $* h \circ_p$ *monom 1 2*) $\circ_p [:0, -1:]$ **unfolding** *fgh* **by** *simp*
also have ($g \circ_p$ *monom 1 2* + *monom 1 1* $* h \circ_p$ *monom 1 2*) $\circ_p [:0, -1:]$
$= g \circ_p$ (*monom 1 2* $\circ_p [:0, -1:]$) + *monom 1 1* $\circ_p [:0, -1:] * h \circ_p$ (*monom 1 2* $\circ_p [:0, -1:]$)
unfolding *pcompose-add pcompose-mult pcompose-assoc* **by** *simp*
also have *monom* (*1* :: *'a*) *2* $\circ_p [:0, -1:]$ = *monom 1 2* **unfolding** *m2* **by** *auto*
also have $?x \circ_p [:0, -1:] = [:0, -1:]$ **unfolding** *m2* **by** *auto*
also have $[:0, -1:] * h \circ_p$ *monom 1 2* = $(-?x) * ?h$ **unfolding** *m2* **by** *simp*
also have $(?g + ?x * ?h) * (?g + (-\ ?x) * ?h) = (?g * ?g - (?x * ?x) * ?h * ?h)$
by (*auto simp*: *field-simps*)
also have $?x * ?x = ?x \circ_p$ *monom 1 2* **unfolding** *mult-monom* **by** (*insert m2*, *simp add*: *2*)
also have $(?g * ?g - \ldots * ?h * ?h) = (g * g - ?x * h * h) \circ_p$ *monom 1 2*
unfolding *pcompose-diff pcompose-mult* **by** *auto*
finally have *poly-square-subst* ($f * f \circ_p [:0, -1:]$)
= *poly-square-subst* (($g * g - ?x * h * h) \circ_p$ *monom 1 2*) **by** *simp*
also have $\ldots = g * g - ?x * h * h$ **unfolding** *poly-square-subst* **by** *simp*
finally show *poly-square-subst* ($f * f \circ_p [:0, -1:]$) = $g * g - ?x * h * h$ **.**
qed
qed

fun *graeffe-poly-impl-main* :: *'a* ⇒ *'a* :: *idom poly* ⇒ *nat* ⇒ *'a poly* **where**
graeffe-poly-impl-main c f 0 = f
| *graeffe-poly-impl-main c f* (*Suc m*) = *graeffe-one-step c* (*graeffe-poly-impl-main c f m*)

lemma *graeffe-poly-impl-main*: **assumes** *f* = *smult c* ($\prod$ *a*←*as*. [:− *a*, *1*:])
shows *graeffe-poly-impl-main* ((*−1*)^*degree f*) *f m* = *graeffe-poly c as m*
proof (*induct m*)
case *0*
show *?case* **using** *graeffe-0*[*OF assms*] **by** *simp*
next
case (*Suc m*)
have [*simp*]: *degree* (*graeffe-poly c as m*) = *degree f* **unfolding** *graeffe-poly-def degree-smult-eq assms*
degree-linear-factors **by** *auto*

from *arg-cong*[*OF Suc*, *of degree*]
show *?case* **unfolding** *graeffe-recursion*[*OF Suc*[*symmetric*]]
by (*simp add*: *graeffe-one-step-def*)
qed

definition *graeffe-poly-impl* :: *'a* :: *idom poly* ⇒ *nat* ⇒ *'a poly* **where**
graeffe-poly-impl f = *graeffe-poly-impl-main* ((*−1*)^(*degree f*)) *f*

lemma *graeffe-poly-impl*: **assumes** *f* = *smult c* (∏ *a*←*as*. [:− *a*, *1*:])
shows *graeffe-poly-impl f m* = *graeffe-poly c as m*
using *graeffe-poly-impl-main*[*OF assms*] **unfolding** *graeffe-poly-impl-def* **.**

lemma *drop-half-map*: *drop-half* (*map f xs*) = *map f* (*drop-half xs*)
by (*induct xs rule*: *drop-half.induct*, *auto*)

lemma (**in** *inj-comm-ring-hom*) *map-poly-poly-square-subst*:
map-poly hom (*poly-square-subst f*) = *poly-square-subst* (*map-poly hom f*)
unfolding *poly-square-subst-def coeffs-map-poly-hom drop-half-map poly-of-list-def*
by (*rule poly-eqI*, *auto simp*: *nth-default-map-eq*)

context *inj-idom-hom*
begin

lemma *graeffe-poly-impl-hom*:
map-poly hom (*graeffe-poly-impl f m*) = *graeffe-poly-impl* (*map-poly hom f*) *m*
proof −
interpret *mh*: *map-poly-inj-idom-hom*..
obtain *c* **where** *c*: (((− *1*) ^ *degree f*) :: *'a*) = *c* **by** *auto*
have *c'*: (((− *1*) ^ *degree f*) :: *'b*) = *hom c* **unfolding** *c*[*symmetric*] **by** (*simp add*:*hom-distribs*)
show *?thesis* **unfolding** *graeffe-poly-impl-def degree-map-poly-hom c c'*
apply (*induct m arbitrary*: *f*; *simp*)
by (*unfold graeffe-one-step-def hom-distribs map-poly-poly-square-subst map-poly-pcompose*,*simp*)
qed
end

lemma *graeffe-poly-impl-mahler*: *mahler-measure* (*graeffe-poly-impl f m*) = *mahler-measure f* ^ *2* ^ *m*
proof −
let *?c* = *complex-of-int*
let *?cc* = *map-poly ?c*
let *?f* = *?cc f*
note *eq* = *complex-roots*(*1*)[*of ?f*]
interpret *inj-idom-hom complex-of-int* **by** (*standard*, *auto*)
show *?thesis*
unfolding *mahler-measure-def mahler-graeffe*[*OF eq*[*symmetric*], *symmetric*]
graeffe-poly-impl[*OF eq*[*symmetric*], *symmetric*] **by** (*simp add*: *of-int-hom.graeffe-poly-impl-hom*)
qed

definition *mahler-landau-graeffe-approximation* :: *nat* ⇒ *nat* ⇒ *int poly* ⇒ *int*
where
mahler-landau-graeffe-approximation kk dd f = (*let*
no = *sum-list* (*map* (λ *a*. *a* ∗ *a*) (*coeffs f*))
in root-int-floor kk (*dd* ∗ *no*))

lemma *mahler-landau-graeffe-approximation-core*:
assumes *g*: *g* = *graeffe-poly-impl f k*
shows *mahler-measure f* ≤ *root* (*2* ^ *Suc k*) (*real-of-int* (∑ *a*←*coeffs g*. *a* ∗ *a*))
proof −
have *mahler-measure f* = *root* (*2*^*k*) (*mahler-measure f* ^ (*2*^*k*))
by (*simp add*: *real-root-power-cancel mahler-measure-ge-0*)
also have ... = *root* (*2*^*k*) (*mahler-measure g*)
unfolding *graeffe-poly-impl-mahler g* **by** *simp*
also have ... = *root* (*2*^*k*) (*root 2* (((*mahler-measure g*)^*2*)))
by (*simp add*: *real-root-power-cancel mahler-measure-ge-0*)
also have ... = *root* (*2*^*Suc k*) (((*mahler-measure g*)^*2*))
by (*metis power-Suc2 real-root-mult-exp*)
also have ... ≤ *root* (*2* ^ *Suc k*) (*real-of-int* (∑ *a*←*coeffs g*. *a* ∗ *a*))
proof (*rule real-root-le-mono*, *force*)
have *square-mono*: *0* ≤ (*x* :: *real*) ⟹ *x* ≤ *y* ⟹ *x* ∗ *x* ≤ *y* ∗ *y* **for** *x y*
by (*simp add*: *mult-mono'*)
obtain *gs* **where** *gs*: *coeffs g* = *gs* **by** *auto*
have (*mahler-measure g*)^2 ≤ *real-of-int* |∑ *a*←*coeffs g*. *a* ∗ *a*|
using *square-mono*[*OF mahler-measure-ge-0 Landau-inequality*[*of of-int-poly g*, *folded mahler-measure-def*]]
by (*auto simp*: *power2-eq-square coeffs-map-poly o-def of-int-hom.hom-sum-list*)
also have |∑ *a*←*coeffs g*. *a* ∗ *a*| = (∑ *a*←*coeffs g*. *a* ∗ *a*) **unfolding** *gs*
by (*induct gs*, *auto*)
finally show (*mahler-measure g*)^2 ≤ *real-of-int* (∑ *a*←*coeffs g*. *a* ∗ *a*) **.**
qed
finally show *mahler-measure f* ≤ *root* (*2* ^ *Suc k*) (*real-of-int* (∑ *a*←*coeffs g*. *a* ∗ *a*)) **.**
qed

lemma *Landau-inequality-mahler-measure*: *mahler-measure f* ≤ *sqrt* (*real-of-int* (∑ *a*←*coeffs f*. *a* ∗ *a*))
by (*rule order.trans*[*OF mahler-landau-graeffe-approximation-core*[*OF refl*, *of - 0*]],
auto simp: *graeffe-poly-impl-def sqrt-def*)

lemma *mahler-landau-graeffe-approximation*:
assumes *g*: *g* = *graeffe-poly-impl f k dd* = *d*^(*2*^(*Suc k*)) *kk* = *2*^(*Suc k*)
shows ⌊*real d* ∗ *mahler-measure f*⌋ ≤ *mahler-landau-graeffe-approximation kk dd g*
proof −
have *id1*: *real-of-int* (*int* (*d* ^ *2* ^ *Suc k*)) = (*real d*) ^ *2* ^ *Suc k* **by** *simp*
have *id2*: *root* (*2* ^ *Suc k*) (*real d* ^ *2* ^ *Suc k*) = *real d*
by (*simp add*: *real-root-power-cancel*)

show *?thesis* **unfolding** *mahler-landau-graeffe-approximation-def Let-def root-int-floor of-int-mult g(2−3)*
by (*rule floor-mono*, *unfold real-root-mult id1 id2*, *rule mult-left-mono*, *rule mahler-landau-graeffe-approximation-core*[*OF g*(*1*)], *auto*)
qed

context
fixes *bnd* :: *nat*
begin

function *mahler-approximation-main* :: *nat* ⇒ *int* ⇒ *int poly* ⇒ *int* ⇒ *nat* ⇒ *nat* ⇒ *int* **where**
mahler-approximation-main dd c g mm k kk = (*let mmm* = *mahler-landau-graeffe-approximation kk dd g*;
new-mm = (*if k* = *0 then mmm else min mm mmm*)
in (*if k* ≥ *bnd then new-mm else*
— abort after *bnd* iterations of Graeffe transformation
mahler-approximation-main (*dd* ∗ *dd*) *c* (*graeffe-one-step c g*) *new-mm* (*Suc k*) (*2* ∗ *kk*)))
by *pat-completeness auto*

termination by (*relation measure* (λ (*dd*,*c*,*f*,*mm*,*k*,*kk*). *Suc bnd* − *k*), *auto*)
declare *mahler-approximation-main.simps*[*simp del*]

lemma *mahler-approximation-main*: **assumes** *k* ≠ *0* ⟹ ⌊*real d* ∗ *mahler-measure f*⌋ ≤ *mm*
and *c* = (−*1*)^(*degree f*)
and *g* = *graeffe-poly-impl-main c f k dd* = *d*^(*2*^(*Suc k*)) *kk* = *2*^(*Suc k*)
shows ⌊*real d* ∗ *mahler-measure f*⌋ ≤ *mahler-approximation-main dd c g mm k kk*
using *assms*
proof (*induct c g mm k kk rule*: *mahler-approximation-main.induct*)
case (*1 dd c g mm k kk*)
let *?df* = ⌊*real d* ∗ *mahler-measure f*⌋
note *dd* = *1*(*5*)
note *kk* = *1*(*6*)
note *g* = *1*(*4*)
note *c* = *1*(*3*)
note *mm* = *1*(*2*)
note *IH* = *1*(*1*)
note *mahl* = *mahler-approximation-main.simps*[*of dd c g mm k kk*]
define *mmm* **where** *mmm* = *mahler-landau-graeffe-approximation kk dd g*
define *new-mm* **where** *new-mm* = (*if k* = *0 then mmm else min mm mmm*)
let *?cond* = *bnd* ≤ *k*
have *id*: *mahler-approximation-main dd c g mm k kk* = (*if ?cond then new-mm else mahler-approximation-main* (*dd* ∗ *dd*) *c* (*graeffe-one-step c g*) *new-mm* (*Suc k*) (*2* ∗ *kk*))
unfolding *mahl mmm-def*[*symmetric*] *Let-def new-mm-def*[*symmetric*] **by** *simp*
have *gg*: *g* = (*graeffe-poly-impl f k*) **unfolding** *g graeffe-poly-impl-def c* **..**

from *mahler-landau-graeffe-approximation*[*OF gg dd kk*, *folded mmm-def*]
have *mmm*: *?df* $\leq$ *mmm* **.**
with *mm* **have** *new-mm*: *?df* $\leq$ *new-mm* **unfolding** *new-mm-def* **by** *auto*
show *?case*
proof (*cases ?cond*)
case *True*
show *?thesis* **unfolding** *id* **using** *True new-mm* **by** *auto*
next
case *False*
hence *id*: *mahler-approximation-main dd c g mm k kk* =
mahler-approximation-main (*dd* $*$ *dd*) *c* (*graeffe-one-step c g*) *new-mm* (*Suc k*) (*2* $*$ *kk*)
unfolding *id* **by** *auto*
have *id'*: *graeffe-one-step c g* = *graeffe-poly-impl-main c f* (*Suc k*)
unfolding *g* **by** *simp*
have *dd* $*$ *dd* = *d* ^ *2* ^ *Suc* (*Suc k*) *2* $*$ *kk* = *2* ^ *Suc* (*Suc k*) **unfolding** *dd kk*
semiring-normalization-rules(*26*) **by** *auto*
from *IH*[*OF mmm-def new-mm-def False new-mm c id' this*]
show *?thesis* **unfolding** *id* **.**
qed
qed

definition *mahler-approximation* :: *nat* $\Rightarrow$ *int poly* $\Rightarrow$ *int* **where**
mahler-approximation d f = *mahler-approximation-main* (*d* $*$ *d*) ((*−1*)^(*degree f*)) *f* (*−1*) *0 2*

lemma *mahler-approximation*: $\lfloor$*real d* $*$ *mahler-measure f*$\rfloor$ $\leq$ *mahler-approximation d f*
unfolding *mahler-approximation-def*
by (*rule mahler-approximation-main*, *auto simp*: *semiring-normalization-rules*(*29*))

end

end

10.5 The Mignotte Bound

theory *Factor-Bound*
imports
Mahler-Measure
Polynomial-Factorization.Gauss-Lemma
Subresultants.Coeff-Int
begin

lemma *binomial-mono-left*: *n* $\leq$ *N* $\Longrightarrow$ *n choose k* $\leq$ *N choose k*
proof (*induct n arbitrary*: *k N*)
case (*0 k N*)
thus *?case* **by** (*cases k*, *auto*)

next
  case (*Suc n k N*) **note** *IH* = *this*
  show *?case*
  proof (*cases k*)
    case (*Suc kk*)
    from *IH* **obtain** *NN* **where** *N*: *N* = *Suc NN* **and** *le*: *n* ≤ *NN* **by** (*cases N*, *auto*)
    show *?thesis* **unfolding** *N Suc* **using** *IH*(*1*)[*OF le*]
      by (*simp add*: *add-le-mono*)
  qed *auto*
qed

definition *choose-int* **where** *choose-int m n* = (*if n* < *0 then 0 else m choose* (*nat n*))

lemma *choose-int-suc*[*simp*]:
  choose-int (*Suc n*) *i* = *choose-int n* (*i*−*1*) + *choose-int n i*
proof(*cases nat i*)
  case *0* **thus** *?thesis* **by** (*simp add*:*choose-int-def*) **next**
  case (*Suc v*) **hence** *nat* (*i* − *1*) = *v i*≠*0* **by** *simp-all*
    thus *?thesis* **unfolding** *choose-int-def Suc* **by** *simp*
qed

lemma *sum-le-1-prod*: **assumes** *d*: *1* ≤ *d* **and** *c*: *1* ≤ *c*
  shows *c* + *d* ≤ *1* + *c* ∗ (*d* :: *real*)
proof −
  from *d c* **have** (*c* − *1*) ∗ (*d* − *1*) ≥ *0* **by** *auto*
  thus *?thesis* **by** (*auto simp*: *field-simps*)
qed

lemma *mignotte-helper-coeff-int*: *cmod* (*coeff-int* (∏ *a*←*lst*. [:− *a*, *1*:]) *i*)
    ≤ *choose-int* (*length lst* − *1*) *i* ∗ (∏ *a*←*lst*. (*max 1* (*cmod a*)))
    + *choose-int* (*length lst* − *1*) (*i* − *1*)
proof(*induct lst arbitrary*:*i*)
  case *Nil* **thus** *?case* **by** (*auto simp*:*coeff-int-def choose-int-def*)
  case (*Cons v xs i*)
  show *?case*
  proof (*cases xs* = [])
    case *True*
    show *?thesis* **unfolding** *True*
      by (*cases nat i*, *cases nat* (*i* − *1*), *auto simp*: *coeff-int-def choose-int-def*)
  next
    case *False*
    hence *id*: *length* (*v* # *xs*) − *1* = *Suc* (*length xs* − *1*) **by** *auto*
    have *id'*: *choose-int* (*length xs*) *i* = *choose-int* (*Suc* (*length xs* − *1*)) *i* **for** *i*
      using *False* **by** (*cases xs*, *auto*)
    let *?r* = (∏ *a*←*xs*. [:− *a*, *1*:])
    let *?mv* = (∏ *a*←*xs*. (*max 1* (*cmod a*)))
    let *?c1* = *real* (*choose-int* (*length xs* − *1*) (*i* − *1* − *1*))

let *?c2* = *real* (*choose-int* (*length* (*v* # *xs*) − *1*) *i* − *choose-int* (*length xs* − *1*) *i*)
let *?m xs n* = *choose-int* (*length xs* − *1*) *n* ∗ (∏ *a*←*xs*. (*max 1* (*cmod a*)))
have *le1*:*1* ≤ *max 1* (*cmod v*) **by** *auto*
have *le2*:*cmod v* ≤ *max 1* (*cmod v*) **by** *auto*
have *mv-ge-1*:*1* ≤ *?mv* **by** (*rule prod-list-ge1*, *auto*)
obtain *a b c d* **where** *abcd* :
  a = *real* (*choose-int* (*length xs* − *1*) *i*)
  b = *real* (*choose-int* (*length xs* − *1*) (*i* − *1*))
  c = (∏ *a*←*xs*. *max 1* (*cmod a*))
  d = *cmod v* **by** *auto*
{
  have *c1*: *c* ≥ *1* **unfolding** *abcd* **by** (*rule mv-ge-1*)
  have *b*: *b* = *0* ∨ *b* ≥ *1* **unfolding** *abcd* **by** *auto*
  have *a*: *a* = *0* ∨ *a* ≥ *1* **unfolding** *abcd* **by** *auto*
  hence *a0*: *a* ≥ *0* **by** *auto*
  have *acd*: *a* ∗ (*c* ∗ *d*) ≤ *a* ∗ (*c* ∗ *max 1 d*) **using** *a0 c1*
    by (*simp add*: *mult-left-mono*)
  from *b* **have** *b* ∗ (*c* + *d*) ≤ *b* ∗ (*1* + (*c* ∗ *max 1 d*))
  proof
    assume *b* ≥ *1*
    hence *?thesis* = (*c* + *d* ≤ *1* + *c* ∗ *max 1 d*) **by** *simp*
    also have . . .
    proof (*cases d* ≥ *1*)
      case *False*
      hence *id*: *max 1 d* = *1* **by** *simp*
      show *?thesis* **using** *False* **unfolding** *id* **by** *simp*
    next
      case *True*
      hence *id*: *max 1 d* = *d* **by** *simp*
      show *?thesis* **using** *True c1* **unfolding** *id* **by** (*rule sum-le-1-prod*)
    qed
    finally show *?thesis* **.**
  qed *auto*
  with *acd* **have** *b* ∗ *c* + (*b* ∗ *d* + *a* ∗ (*c* ∗ *d*)) ≤ *b* + (*a* ∗ (*c* ∗ *max 1 d*) + *b* ∗ (*c* ∗ *max 1 d*))
    by (*auto simp*: *field-simps*)
} **note** *abcd-main* = *this*
have *cmod* (*coeff-int* ([:− *v*, *1*:] ∗ *?r*) *i*) ≤ *cmod* (*coeff-int ?r* (*i* − *1*)) + *cmod* (*coeff-int* (*smult v ?r*) *i*)
  using *norm-triangle-ineq4* **by** *auto*
also have . . . ≤ *?m xs* (*i* − *1*) + (*choose-int* (*length xs* − *1*) (*i* − *1* − *1*)) + *cmod* (*coeff-int* (*smult v ?r*) *i*)
  using *Cons*[*of i*−*1*] **by** *auto*
also have *choose-int* (*length xs* − *1*) (*i* − *1*) = *choose-int* (*length* (*v* # *xs*) − *1*) *i* − *choose-int* (*length xs* − *1*) *i*
  unfolding *id choose-int-suc* **by** *auto*
also have *?c2* ∗ (∏ *a*←*xs*. *max 1* (*cmod a*)) + *?c1* +
  cmod (*coeff-int* (*smult v* (∏ *a*←*xs*. [:− *a*, *1*:])) *i*) ≤

?c2 ∗ (∏ *a*←*xs*. *max 1* (*cmod a*)) + *?c1* + *cmod v* ∗ (
real (*choose-int* (*length xs* − *1*) *i*) ∗ (∏ *a*←*xs*. *max 1* (*cmod a*)) +
real (*choose-int* (*length xs* − *1*) (*i* − *1*)))
using *mult-mono′*[*OF order-refl Cons*, *of cmod v i*, *simplified*] **by** (*auto simp*: *norm-mult*)
also have ... ≤ *?m* (*v* # *xs*) *i* + (*choose-int* (*length xs*) (*i* − *1*)) **using** *abcd-main*[*unfolded abcd*]
by (*simp add*: *field-simps id′*)
finally show *?thesis* **by** *simp*
qed
qed

lemma *mignotte-helper-coeff-int′*: *cmod* (*coeff-int* (∏ *a*←*lst*. [:− *a*, *1*:]) *i*)
≤ ((*length lst* − *1*) *choose i*) ∗ (∏ *a*←*lst*. (*max 1* (*cmod a*)))
+ *min i 1* ∗ ((*length lst* − *1*) *choose* (*nat* (*i* − *1*)))
by (*rule order.trans*[*OF mignotte-helper-coeff-int*], *auto simp*: *choose-int-def min-def*)

lemma *mignotte-helper-coeff*:
cmod (*coeff h i*) ≤ (*degree h* − *1 choose i*) ∗ *mahler-measure-poly h*
+ *min i 1* ∗ (*degree h* − *1 choose* (*i* − *1*)) ∗ *cmod* (*lead-coeff h*)
proof −
let *?r* = *complex-roots-complex h*
have *cmod* (*coeff h i*) = *cmod* (*coeff* (*smult* (*lead-coeff h*) (∏ *a*←*?r*. [:− *a*, *1*:])) *i*)
unfolding *complex-roots* **by** *auto*
also have ... = *cmod* (*lead-coeff h*) ∗ *cmod* (*coeff* (∏ *a*←*?r*. [:− *a*, *1*:]) *i*) **by**(*simp add*:*norm-mult*)
also have ... ≤ *cmod* (*lead-coeff h*) ∗ ((*degree h* − *1 choose i*) ∗ *mahler-measure-monic h* +
(*min i 1* ∗ ((*degree h* − *1*) *choose nat* (*int i* − *1*))))
unfolding *mahler-measure-monic-def*
by (*rule mult-left-mono*, *insert mignotte-helper-coeff-int′*[*of ?r i*], *auto*)
also have ... = (*degree h* − *1 choose i*) ∗ *mahler-measure-poly h* + *cmod* (*lead-coeff h*) ∗ (
min i 1 ∗ ((*degree h* − *1*) *choose nat* (*int i* − *1*)))
unfolding *mahler-measure-poly-via-monic* **by** (*simp add*: *field-simps*)
also have *nat* (*int i* − *1*) = *i* − *1* **by** (*cases i*, *auto*)
finally show *?thesis* **by** (*simp add*: *ac-simps split*: *if-splits*)
qed

lemma *mignotte-coeff-helper*:
abs (*coeff h i*) ≤
(*degree h* − *1 choose i*) ∗ *mahler-measure h* +
(*min i 1* ∗ (*degree h* − *1 choose* (*i* − *1*)) ∗ *abs* (*lead-coeff h*))
unfolding *mahler-measure-def*
using *mignotte-helper-coeff*[*of of-int-poly h i*]
by *auto*

lemma *cmod-through-lead-coeff*[*simp*]:

cmod (*lead-coeff* (*of-int-poly h*)) = *abs* (*lead-coeff h*)
by *simp*

lemma *choose-approx*: $n \leq N \Longrightarrow n$ *choose* $k \leq N$ *choose* (*N div 2*)
by (*rule order.trans*[*OF binomial-mono-left binomial-maximum*])

For Mignotte's factor bound, we currently do not support queries for individual coefficients, as we do not have a combined factor bound algorithm.

definition *mignotte-bound* :: *int poly* ⇒ *nat* ⇒ *int* **where**
mignotte-bound f d = (*let d′* = *d* − *1*; *d2* = *d′ div 2*; *binom* = (*d′ choose d2*) *in*
(*mahler-approximation 2 binom f* + *binom* ∗ *abs* (*lead-coeff f*)))

lemma *mignotte-bound-main*:
assumes *f* ≠ *0 g dvd f degree g* ≤ *n*
shows |*coeff g k*| ≤ ⌊*real* (*n* − *1 choose k*) ∗ *mahler-measure f*⌋ +
int (*min k 1* ∗ (*n* − *1 choose* (*k* − *1*))) ∗ |*lead-coeff f*|
proof−
let *?bnd* = *2*
let *?n* = (*n* − *1*) *choose k*
let *?n′* = *min k 1* ∗ ((*n* − *1*) *choose* (*k* − *1*))
let *?approx* = *mahler-approximation ?bnd ?n f*
obtain *h* **where** *gh*:*g* ∗ *h* = *f* **using** *assms* **by** (*metis dvdE*)
have *nz*:*g*≠*0 h*≠*0* **using** *gh assms*(*1*) **by** *auto*
have *g1*:(*1*::*real*) ≤ *mahler-measure h* **using** *mahler-measure-poly-ge-1 gh assms*(*1*)
by *auto*
note *g0* = *mahler-measure-ge-0*
have *to-n*: (*degree g* − *1 choose k*) ≤ *real ?n*
using *binomial-mono-left*[*of degree g* − *1 n* − *1 k*] *assms*(*3*) **by** *auto*
have *to-n′*: *min k 1* ∗ (*degree g* − *1 choose* (*k* − *1*)) ≤ *real ?n′*
using *binomial-mono-left*[*of degree g* − *1 n* − *1 k* − *1*] *assms*(*3*)
by (*simp add*: *min-def*)
have |*coeff g k*| ≤ (*degree g* − *1 choose k*) ∗ *mahler-measure g*
+ (*real* (*min k 1* ∗ (*degree g* − *1 choose* (*k* − *1*))) ∗ |*lead-coeff g*|)
using *mignotte-coeff-helper*[*of g k*] **by** *simp*
also have ... ≤ *?n* ∗ *mahler-measure f* + *real ?n′* ∗ |*lead-coeff f*|
proof (*rule add-mono*[*OF mult-mono*[*OF to-n*] *mult-mono*[*OF to-n′*]])
have *mahler-measure g* ≤ *mahler-measure g* ∗ *mahler-measure h* **using** *g1*
g0[*of g*]
using *mahler-measure-poly-ge-1 nz*(*1*) **by** *force*
thus *mahler-measure g* ≤ *mahler-measure f*
using *measure-eq-prod*[*of of-int-poly g of-int-poly h*]
unfolding *mahler-measure-def gh*[*symmetric*] **by** (*auto simp*: *hom-distribs*)
have ∗: *lead-coeff f* = *lead-coeff g* ∗ *lead-coeff h*
unfolding *arg-cong*[*OF gh*, *of lead-coeff*, *symmetric*] **by** (*rule lead-coeff-mult*)
have |*lead-coeff h*| ≠ *0* **using** *nz*(*2*) **by** *auto*
hence *lh*: |*lead-coeff h*| ≥ *1* **by** *linarith*
have |*lead-coeff f*| = |*lead-coeff g*| ∗ |*lead-coeff h*| **unfolding** ∗ **by** (*rule abs-mult*)
also have ... ≥ |*lead-coeff g*| ∗ *1*
by (*rule mult-mono*, *insert lh*, *auto*)

finally have $|$*lead-coeff g*$|$ $\leq$ $|$*lead-coeff f*$|$ **by** *simp*
thus *real-of-int* $|$*lead-coeff g*$|$ $\leq$ *real-of-int* $|$*lead-coeff f*$|$ **by** *simp*
qed (*auto simp*: *g0*)
finally have $|$*coeff g k*$|$ $\leq$ *?n* $*$ *mahler-measure f* + *real-of-int* (*?n'* $*$ $|$*lead-coeff f*$|$) **by** *simp*
from *floor-mono*[*OF this*, *folded floor-add-int*]
have $|$*coeff g k*$|$ $\leq$ *floor* (*?n* $*$ *mahler-measure f*) + *?n'* $*$ $|$*lead-coeff f*$|$ **by** *linarith*
thus *?thesis* **unfolding** *mignotte-bound-def Let-def* **using** *mahler-approximation*[*of ?n f ?bnd*] **by** *auto*
qed

lemma *Mignotte-bound*:
shows *of-int* $|$*coeff g k*$|$ $\leq$ (*degree g choose k*) $*$ *mahler-measure g*
proof (*cases k* $\leq$ *degree g* $\wedge$ *g* $\neq$ *0*)
case *False*
hence *coeff g k* = *0* **using** *le-degree* **by** (*cases g* = *0*, *auto*)
thus *?thesis* **using** *mahler-measure-ge-0*[*of g*] **by** *auto*
next
case *kg*: *True*
hence *g*: *g* $\neq$ *0 g dvd g* **by** *auto*
from *mignotte-bound-main*[*OF g le-refl*, *of k*]
have *real-of-int* $|$*coeff g k*$|$
$\leq$ *of-int* $\lfloor$*real* (*degree g* $-$ *1 choose k*) $*$ *mahler-measure g*$\rfloor$ +
of-int (*int* (*min k 1* $*$ (*degree g* $-$ *1 choose* (*k* $-$ *1*))) $*$ $|$*lead-coeff g*$|$) **by** *linarith*
also have ... $\leq$ *real* (*degree g* $-$ *1 choose k*) $*$ *mahler-measure g*
+ *real* (*min k 1* $*$ (*degree g* $-$ *1 choose* (*k* $-$ *1*))) $*$ (*of-int* $|$*lead-coeff g*$|$ $*$ *1*)
by (*rule add-mono*, *force*, *auto*)
also have ... $\leq$ *real* (*degree g* $-$ *1 choose k*) $*$ *mahler-measure g*
+ *real* (*min k 1* $*$ (*degree g* $-$ *1 choose* (*k* $-$ *1*))) $*$ *mahler-measure g*
by (*rule add-left-mono*[*OF mult-left-mono*],
unfold mahler-measure-def mahler-measure-poly-def,
rule mult-mono, *auto intro*!: *prod-list-ge1*)
also have ... =
(*real* ((*degree g* $-$ *1 choose k*) + (*min k 1* $*$ (*degree g* $-$ *1 choose* (*k* $-$ *1*))))) $*$ *mahler-measure g*
by (*auto simp*: *field-simps*)
also have (*degree g* $-$ *1 choose k*) + (*min k 1* $*$ (*degree g* $-$ *1 choose* (*k* $-$ *1*))) = *degree g choose k*
proof (*cases k* = *0*)
case *False*
then obtain *kk* **where** *k*: *k* = *Suc kk* **by** (*cases k*, *auto*)
with *kg* **obtain** *gg* **where** *g*: *degree g* = *Suc gg* **by** (*cases degree g*, *auto*)
show *?thesis* **unfolding** *k g* **by** *auto*
qed *auto*
finally show *?thesis* **.**
qed

lemma *mignotte-bound*:

assumes $f \neq 0$ g *dvd* f *degree* $g \leq n$
shows $|$*coeff* g $k|$ $\leq$ *mignotte-bound* f n
proof $-$
let *?bnd* $= 2$
let *?n* $= (n - 1)$ *choose* $((n - 1)$ *div* $2)$
have *to-n*: $(n - 1$ *choose* $k) \leq$ *real* *?n* **for** k
using *choose-approx*[*OF le-refl*] **by** *auto*
from *mignotte-bound-main*[*OF assms*, *of k*]
have $|$*coeff* g $k| \leq$
$\lfloor$*real* $(n - 1$ *choose* $k) *$ *mahler-measure* $f\rfloor$ $+$
int $($*min* k $1 * (n - 1$ *choose* $(k - 1))) * |$*lead-coeff* $f|$ **.**
also have $\ldots \leq \lfloor$*real* $(n - 1$ *choose* $k) *$ *mahler-measure* $f\rfloor$ $+$
int $((n - 1$ *choose* $(k - 1))) * |$*lead-coeff* $f|$
by (*rule add-left-mono*[*OF mult-right-mono*], *cases k*, *auto*)
also have $\ldots \leq$ *mignotte-bound* f n
unfolding *mignotte-bound-def Let-def*
by (*rule add-mono*[*OF order.trans*[*OF floor-mono*[*OF mult-right-mono*]
mahler-approximation[*of ?n f ?bnd*]] *mult-right-mono*], *insert to-n mahler-measure-ge-0*,
auto)
finally show *?thesis* **.**
qed

As indicated before, at the moment the only available factor bound is Mignotte's one. As future work one might use a combined bound.

definition *factor-bound* :: *int poly* $\Rightarrow$ *nat* $\Rightarrow$ *int* **where**
factor-bound $=$ *mignotte-bound*

lemma *factor-bound*: **assumes** $f \neq 0$ g *dvd* f *degree* $g \leq n$
shows $|$*coeff* g $k| \leq$ *factor-bound* f n
unfolding *factor-bound-def* **by** (*rule mignotte-bound*[*OF assms*])

We further prove a result for factor bounds and scalar multiplication.

lemma *factor-bound-ge-0*: $f \neq 0 \Longrightarrow$ *factor-bound* f $n \geq 0$
using *factor-bound*[*of f 1 n 0*] **by** *auto*

lemma *factor-bound-smult*: **assumes** f: $f \neq 0$ **and** d: $d \neq 0$
and *dvd*: g *dvd* *smult* d f **and** *deg*: *degree* $g \leq n$
shows $|$*coeff* g $k| \leq |d| *$ *factor-bound* f n
proof $-$
let *?nf* $=$ *primitive-part* f **let** *?cf* $=$ *content* f
let *?ng* $=$ *primitive-part* g **let** *?cg* $=$ *content* g
from *content-dvd-contentI*[*OF dvd*] **have** *?cg dvd abs* d $*$ *?cf*
unfolding *content-smult-int* **.**
hence *dvd-c*: *?cg dvd* d $*$ *?cf* **using** d
by (*metis abs-content-int abs-mult dvd-abs-iff*)
from *primitive-part-dvd-primitive-partI*[*OF dvd*] **have** *?ng dvd smult* (*sgn d*) *?nf*
unfolding *primitive-part-smult-int* **.**
hence *dvd-n*: *?ng dvd ?nf* **using** d
by (*metis content-eq-zero-iff dvd dvd-smult-int f mult-eq-0-iff content-times-primitive-part smult-smult*)

define *gc* **where** *gc* = *gcd ?cf ?cg*
define *cg* **where** *cg* = *?cg div gc*
from *dvd d f* **have** *g*: *g* ≠ *0* **by** *auto*
from *f* **have** *cf*: *?cf* ≠ *0* **by** *auto*
from *g* **have** *cg*: *?cg* ≠ *0* **by** *auto*
hence *gc*: *gc* ≠ *0* **unfolding** *gc-def* **by** *auto*
have *cg-dvd*: *cg dvd ?cg* **unfolding** *cg-def gc-def* **using** *g* **by** (*simp add*: *div-dvd-iff-mult*)
have *cg-id*: *?cg* = *cg* ∗ *gc* **unfolding** *gc-def cg-def* **using** *g cf* **by** *simp*
from *dvd-smult-int*[*OF d dvd*] **have** *ngf*: *?ng dvd f* **.**
have *gcf*: |*gc*| *dvd content f* **unfolding** *gc-def* **by** *auto*
have *dvd-f*: *smult gc ?ng dvd f*
proof (*rule dvd-content-dvd*,
  unfold content-smult-int content-primitive-part[*OF g*]
  primitive-part-smult-int primitive-part-idemp)
 show |*gc*| ∗ *1 dvd content f* **using** *gcf* **by** *auto*
 show *smult* (*sgn gc*) (*primitive-part g*) *dvd primitive-part f*
  using *dvd-n cf gc* **using** *zsgn-def* **by** *force*
qed
have *cg dvd d* **using** *dvd-c* **unfolding** *gc-def cg-def* **using** *cf cg d*
 by (*simp add*: *div-dvd-iff-mult dvd-gcd-mult*)
then obtain *h* **where** *dcg*: *d* = *cg* ∗ *h* **unfolding** *dvd-def* **by** *auto*
with *d* **have** *h* ≠ *0* **by** *auto*
hence *h1*: |*h*| ≥ *1* **by** *simp*
have *degree* (*smult gc* (*primitive-part g*)) = *degree g*
 using *gc* **by** *auto*
from *factor-bound*[*OF f dvd-f*, *unfolded this*, *OF deg*, *of k*, *unfolded coeff-smult*]
have *le*: |*gc* ∗ *coeff ?ng k*| ≤ *factor-bound f n* **.**
note *f0* = *factor-bound-ge-0*[*OF f*, *of n*]
from *mult-left-mono*[*OF le*, *of abs cg*]
have |*cg* ∗ *gc* ∗ *coeff ?ng k*| ≤ |*cg*| ∗ *factor-bound f n*
 unfolding *abs-mult*[*symmetric*] **by** *simp*
also have *cg* ∗ *gc* ∗ *coeff ?ng k* = *coeff* (*smult ?cg ?ng*) *k* **unfolding** *cg-id* **by** *simp*
also have . . . = *coeff g k* **unfolding** *content-times-primitive-part* **by** *simp*
finally have |*coeff g k*| ≤ *1* ∗ (|*cg*| ∗ *factor-bound f n*) **by** *simp*
also have . . . ≤ |*h*| ∗ (|*cg*| ∗ *factor-bound f n*)
 by (*rule mult-right-mono*[*OF h1*], *insert f0*, *auto*)
also have . . . = (|*cg* ∗ *h*|) ∗ *factor-bound f n* **by** (*simp add*: *abs-mult*)
finally show *?thesis* **unfolding** *dcg* **.**
qed

end

10.6 Iteration of Subsets of Factors

theory *Sublist-Iteration*
imports
 Polynomial-Factorization.Missing-Multiset
 Polynomial-Factorization.Missing-List

HOL−Library.IArray
begin

Misc lemmas **lemma** *mem-snd-map*: $(\exists x.\ (x,\ y) \in S) \longleftrightarrow y \in \mathit{snd}\ `\ S$ **by** *force*

lemma *filter-upt*: **assumes** $l \le m$ $m < n$ **shows** $\mathit{filter}\ ((\le)\ m)\ [l..<n] = [m..<n]$
proof(*insert assms*, *induct n*)
  case *0* **then show** *?case* **by** *auto*
next
  case (*Suc n*) **then show** *?case* **by** (*cases* $m = n$, *auto*)
qed

lemma *upt-append*: $i < j \Longrightarrow j < k \Longrightarrow [i..<j]@[j..<k] = [i..<k]$
proof(*induct k arbitrary*: *j*)
  case *0* **then show** *?case* **by** *auto*
next
  case (*Suc k*) **then show** *?case* **by** (*cases* $j = k$, *auto*)
qed

lemma *IArray-sub*[*simp*]: (!!) *as* = (!) (*IArray.list-of as*) **by** *auto*
declare *IArray.sub-def*[*simp del*]

Following lemmas in this section are for *subseqs*

lemma *subseqs-Cons*[*simp*]: *subseqs* (*x#xs*) = *map* (*Cons x*) (*subseqs xs*) @ *subseqs xs*
  by (*simp add*: *Let-def*)

declare *subseqs.simps*(*2*) [*simp del*]

lemma *singleton-mem-set-subseqs* [*simp*]: $[x] \in \mathit{set}\ (\mathit{subseqs}\ \mathit{xs}) \longleftrightarrow x \in \mathit{set}\ \mathit{xs}$ **by** (*induct xs*, *auto*)

lemma *Cons-mem-set-subseqsD*: $y\#ys \in \mathit{set}\ (\mathit{subseqs}\ \mathit{xs}) \Longrightarrow y \in \mathit{set}\ \mathit{xs}$ **by** (*induct xs*, *auto*)

lemma *subseqs-subset*: $ys \in \mathit{set}\ (\mathit{subseqs}\ \mathit{xs}) \Longrightarrow \mathit{set}\ ys \subseteq \mathit{set}\ \mathit{xs}$
  by (*metis Pow-iff image-eqI subseqs-powset*)

lemma *Cons-mem-set-subseqs-Cons*:
  $y\#ys \in \mathit{set}\ (\mathit{subseqs}\ (x\#xs)) \longleftrightarrow (y = x \wedge ys \in \mathit{set}\ (\mathit{subseqs}\ xs)) \vee y\#ys \in \mathit{set}\ (\mathit{subseqs}\ xs)$
  by *auto*

lemma *sorted-subseqs-sorted*:
  $\mathit{sorted}\ xs \Longrightarrow ys \in \mathit{set}\ (\mathit{subseqs}\ xs) \Longrightarrow \mathit{sorted}\ ys$
proof(*induct xs arbitrary*: *ys*)
  case *Nil* **thus** *?case* **by** *simp*
next

case *Cons* **thus** *?case* **using** *subseqs-subset* **by** *fastforce*
qed

lemma *subseqs-of-subseq*: $ys \in set\ (subseqs\ xs) \Longrightarrow set\ (subseqs\ ys) \subseteq set\ (subseqs\ xs)$
proof(*induct xs arbitrary*: *ys*)
  case *Nil* **then show** *?case* **by** *auto*
next
  case *IHx*: (*Cons x xs*)
  from *IHx.prems* **show** *?case*
  proof(*induct ys*)
    case *Nil* **then show** *?case* **by** *auto*
  next
    case *IHy*: (*Cons y ys*)
    from *IHy.prems*[*unfolded subseqs-Cons*]
    consider $y = x\ ys \in set\ (subseqs\ xs) \mid y\ \#\ ys \in set\ (subseqs\ xs)$ **by** *auto*
    then show *?case*
    proof(*cases*)
      case *1* **with** *IHx.hyps* **show** *?thesis* **by** *auto*
    next
      case *2* **from** *IHx.hyps*[*OF this*] **show** *?thesis* **by** *auto*
    qed
  qed
qed

lemma *mem-set-subseqs-append*: $xs \in set\ (subseqs\ ys) \Longrightarrow xs \in set\ (subseqs\ (zs\ @\ ys))$
  by (*induct zs*, *auto*)

lemma *Cons-mem-set-subseqs-append*:
  $x \in set\ ys \Longrightarrow xs \in set\ (subseqs\ zs) \Longrightarrow x\#xs \in set\ (subseqs\ (ys@zs))$
proof(*induct ys*)
  case *Nil* **then show** *?case* **by** *auto*
next
  case *IH*: (*Cons y ys*)
  then consider $x = y \mid x \in set\ ys$ **by** *auto*
  then show *?case*
  proof(*cases*)
    case *1* **with** *IH* **show** *?thesis* **by** (*auto intro*: *mem-set-subseqs-append*)
  next
    case *2* **from** *IH.hyps*[*OF this IH.prems*(*2*)] **show** *?thesis* **by** *auto*
  qed
qed

lemma *Cons-mem-set-subseqs-sorted*:
  $sorted\ xs \Longrightarrow y\#ys \in set\ (subseqs\ xs) \Longrightarrow y\#ys \in set\ (subseqs\ (filter\ (\lambda x.\ y \le x)\ xs))$
by (*induct xs*) (*auto simp*: *Let-def*)

lemma *subseqs-map*[*simp*]: *subseqs* (*map f xs*) = *map* (*map f*) (*subseqs xs*) **by** (*induct xs*, *auto*)

lemma *subseqs-of-indices*: *map* (*map* (*nth xs*)) (*subseqs* [*0*..<*length xs*]) = *subseqs xs*
proof (*induct xs*)
  case *Nil* **then show** *?case* **by** *auto*
next
  case (*Cons x xs*)
  from *this*[*symmetric*]
  have *subseqs xs* = *map* (*map* ((!) (*x#xs*))) (*subseqs* [*Suc 0*..<*Suc* (*length xs*)])
    by (*fold map-Suc-upt*, *simp*)
  then show *?case* **by** (*unfold length-Cons upt-conv-Cons*[*OF zero-less-Suc*], *simp*)
qed

Specification **definition** *subseq-of-length n xs ys* ≡ *ys* ∈ *set* (*subseqs xs*) ∧ *length ys* = *n*

lemma *subseq-of-lengthI*[*intro*]:
  assumes *ys* ∈ *set* (*subseqs xs*) *length ys* = *n*
  shows *subseq-of-length n xs ys*
by (*insert assms*, *unfold subseq-of-length-def*, *auto*)

lemma *subseq-of-lengthD*[*dest*]:
  assumes *subseq-of-length n xs ys*
  shows *ys* ∈ *set* (*subseqs xs*) *length ys* = *n*
  by (*insert assms*, *unfold subseq-of-length-def*, *auto*)

lemma *subseq-of-length0*[*simp*]: *subseq-of-length 0 xs ys* ⟷ *ys* = [] **by** *auto*

lemma *subseq-of-length-Nil*[*simp*]: *subseq-of-length n* [] *ys* ⟷ *n* = *0* ∧ *ys* = []
  by (*auto simp*: *subseq-of-length-def*)

lemma *subseq-of-length-Suc-upt*:
  subseq-of-length (*Suc n*) [*0*..<*m*] *xs* ⟷
   (*if n* = *0 then length xs* = *Suc 0* ∧ *hd xs* < *m*
    else hd xs < *hd* (*tl xs*) ∧ *subseq-of-length n* [*0*..<*m*] (*tl xs*)) (**is** *?l* ⟷ *?r*)
proof(*cases n*)
  case *0*
  show *?thesis*
  proof(*intro iffI*)
    assume *l*: *?l*
    with *0* **have** *1*: *length xs* = *Suc 0* **by** *auto*
    then have *xs*: *xs* = [*hd xs*] **by** (*metis length-0-conv length-Suc-conv list.sel*(*1*))
    with *l* **have** [*hd xs*] ∈ *set* (*subseqs* [*0*..<*m*]) **by** *auto*
    with *1* **show** *?r* **by** (*unfold 0*, *auto*)
  next
    assume *?r*

with *0* **have** *1*: *length xs* = *Suc 0* **and** *2*: *hd xs* < *m* **by** *auto*
then have *xs*: *xs* = [*hd xs*] **by** (*metis length-0-conv length-Suc-conv list.sel*(*1*))
from *2* **show** *?l* **by** (*subst xs*, *auto simp*: *0*)
qed
next
case *n*: (*Suc n'*)
show *?thesis*
proof (*intro iffI*)
assume *?l*
with *n* **have** *1*: *length xs* = *Suc* (*Suc n'*) **and** *2*: *xs* ∈ *set* (*subseqs* [*0*..<*m*]) **by** *auto*
from *1*[*unfolded length-Suc-conv*]
obtain *x y ys* **where** *xs*: *xs* = *x*#*y*#*ys* **and** *n'*: *length ys* = *n'* **by** *auto*
have *sorted xs* **by**(*rule sorted-subseqs-sorted*[*OF* - *2*], *auto*)
from *this*[*unfolded xs*] **have** *x* ≤ *y* **by** *auto*
moreover
from *2* **have** *distinct xs* **by** (*rule subseqs-distinctD*, *auto*)
from *this*[*unfolded xs*] **have** *x* ≠ *y* **by** *auto*
ultimately have *x* < *y* **by** *auto*
moreover
from *2* **have** *y*#*ys* ∈ *set* (*subseqs* [*0*..<*m*]) **by** (*unfold xs*, *auto dest*: *Cons-in-subseqsD*)
with *n n'* **have** *subseq-of-length n* [*0*..<*m*] (*y*#*ys*) **by** *auto*
ultimately show *?r* **by** (*auto simp*: *xs*)
next
assume *r*: *?r*
with *n* **have** *len*: *length xs* = *Suc* (*Suc n'*)
and ∗: *hd xs* < *hd* (*tl xs*) *tl xs* ∈ *set* (*subseqs* [*0*..<*m*]) **by** *auto*
from *len*[*unfolded length-Suc-conv*] **obtain** *x y ys*
where *xs*: *xs* = *x*#*y*#*ys* **and** *n'*: *length ys* = *n'* **by** *auto*
with ∗ **have** *xy*: *x* < *y* **and** *yys*: *y*#*ys* ∈ *set* (*subseqs* [*0*..<*m*]) **by** *auto*
from *Cons-mem-set-subseqs-sorted*[*OF* - *yys*]
have *y*#*ys* ∈ *set* (*subseqs* (*filter* ((≤) *y*) [*0*..<*m*])) **by** *auto*
also from *Cons-mem-set-subseqsD*[*OF yys*] **have** *ym*: *y* < *m* **by** *auto*
then have *filter* ((≤) *y*) [*0*..<*m*] = [*y*..<*m*] **by** (*auto intro*: *filter-upt*)
finally have *y*#*ys* ∈ *set* (*subseqs* [*y*..<*m*]) **by** *auto*
with *xy* **have** *x*#*y*#*ys* ∈ *set* (*subseqs* (*x*#[*y*..<*m*])) **by** *auto*
also from *xy* **have** ... ⊆ *set* (*subseqs* ([*0*..<*y*] @ [*y*..<*m*]))
by (*intro subseqs-of-subseq Cons-mem-set-subseqs-append*, *auto intro*: *subseqs-refl*)
also from *xy ym* **have** [*0*..<*y*] @ [*y*..<*m*] = [*0*..<*m*] **by** (*auto intro*: *upt-append*)
finally have *xs* ∈ *set* (*subseqs* [*0*..<*m*]) **by** (*unfold xs*)
with *len*[*folded n*] **show** *?l* **by** *auto*
qed
qed

lemma *subseqs-of-length-of-indices*:
{ *ys*. *subseq-of-length n xs ys* } = { *map* (*nth xs*) *is* | *is*. *subseq-of-length n* [*0*..<*length xs*] *is* }

by(*unfold subseq-of-length-def*, *subst subseqs-of-indices*[*symmetric*], *auto*)

lemma *subseqs-of-length-Suc-Cons*:
{ *ys. subseq-of-length* (*Suc n*) (*x#xs*) *ys* } =
Cons x ' { *ys. subseq-of-length n xs ys* } ∪ { *ys. subseq-of-length* (*Suc n*) *xs ys* }
by (*unfold subseq-of-length-def*, *auto*)

datatype ('*a*,'*b*,'*state*)*subseqs-impl* = *Sublists-Impl*
(*create-subseqs*: '*b* ⇒ '*a list* ⇒ *nat* ⇒ ('*b* × '*a list*)*list* × '*state*)
(*next-subseqs*: '*state* ⇒ ('*b* × '*a list*)*list* × '*state*)

locale *subseqs-impl* =
fixes *f* :: '*a* ⇒ '*b* ⇒ '*b*
and *sl-impl* :: ('*a*,'*b*,'*state*)*subseqs-impl*
begin

definition *S* :: '*b* ⇒ '*a list* ⇒ *nat* ⇒ ('*b* × '*a list*)*set* **where**
S base elements n = { (*foldr f ys base*, *ys*) | *ys. subseq-of-length n elements ys* }

end

locale *correct-subseqs-impl* = *subseqs-impl f sl-impl*
for *f* :: '*a* ⇒ '*b* ⇒ '*b*
and *sl-impl* :: ('*a*,'*b*,'*state*)*subseqs-impl* +
fixes *invariant* :: '*b* ⇒ '*a list* ⇒ *nat* ⇒ '*state* ⇒ *bool*
assumes *create-subseqs*: *create-subseqs sl-impl base elements n* = (*out*, *state*) ⟹
invariant base elements n state ∧ *set out* = *S base elements n*
and *next-subseqs*:
invariant base elements n state ⟹
next-subseqs sl-impl state = (*out*, *state*') ⟹
invariant base elements (*Suc n*) *state*' ∧ *set out* = *S base elements* (*Suc n*)

Basic Implementation **fun** *subseqs-i-n-main* :: ('*a* ⇒ '*b* ⇒ '*b*) ⇒ '*b* ⇒ '*a list* ⇒ *nat* ⇒ *nat* ⇒ ('*b* × '*a list*) *list* **where**
subseqs-i-n-main f b xs i n = (*if i* = *0 then* [(*b*,[])] *else if i* = *n then* [(*foldr f xs b*, *xs*)]
else case xs of
(*y* # *ys*) ⇒ *map* (λ (*c*,*zs*) ⇒ (*c*,*y* # *zs*)) (*subseqs-i-n-main f* (*f y b*) *ys* (*i* − *1*) (*n* − *1*))
@ *subseqs-i-n-main f b ys i* (*n* − *1*))
declare *subseqs-i-n-main.simps*[*simp del*]

definition *subseqs-length* :: ('*a* ⇒ '*b* ⇒ '*b*) ⇒ '*b* ⇒ *nat* ⇒ '*a list* ⇒ ('*b* × '*a list*) *list* **where**
subseqs-length f b i xs = (
let n = *length xs in if i* > *n then* [] *else subseqs-i-n-main f b xs i n*)

lemma *subseqs-length*: **assumes** *f-ac*: ⋀ *x y z*. *f x* (*f y z*) = *f y* (*f x z*)

shows *set* (*subseqs-length f a n xs*) =
{ (*foldr f ys a*, *ys*) | *ys*. *ys* ∈ *set* (*subseqs xs*) ∧ *length ys* = *n*}
proof −
show *?thesis*
proof (*cases length xs* < *n*)
case *True*
thus *?thesis* **unfolding** *subseqs-length-def Let-def*
using *length-subseqs*[*of xs*] *subseqs-length-simple-False* **by** *auto*
next
case *False*
hence *id*: (*length xs* < *n*) = *False* **and** *n* ≤ *length xs* **by** *auto*
from *this*(*2*) **show** *?thesis* **unfolding** *subseqs-length-def Let-def id if-False*
proof (*induct xs arbitrary*: *n a rule*: *length-induct*[*rule-format*])
case (*1 xs n a*)
note *n* = *1*(*2*)
note *IH* = *1*(*1*)
note *simp*[*simp*] = *subseqs-i-n-main.simps*[*of f - xs n*]
show *?case*
proof (*cases n* = *0*)
case *True*
thus *?thesis* **unfolding** *simp* **by** *simp*
next
case *False* **note** *0* = *this*
show *?thesis*
proof (*cases n* = *length xs*)
case *True*
have *?thesis* = ({(*foldr f xs a*, *xs*)} = (λ *ys*. (*foldr f ys a*, *ys*)) ‘ {*ys*. *ys* ∈
set (*subseqs xs*) ∧ *length ys* = *length xs*})
unfolding *simp* **using** *0 True* **by** *auto*
from *this*[*unfolded full-list-subseqs*] **show** *?thesis* **by** *auto*
next
case *False*
with *n* **have** *n*: *n* < *length xs* **by** *auto*
from *0* **obtain** *m* **where** *m*: *n* = *Suc m* **by** (*cases n*, *auto*)
from *n 0* **obtain** *y ys* **where** *xs*: *xs* = *y* # *ys* **by** (*cases xs*, *auto*)
from *n m xs* **have** *le*: *m* ≤ *length ys n* ≤ *length ys* **by** *auto*
from *xs* **have** *lt*: *length ys* < *length xs* **by** *auto*
have *sub*: *set* (*subseqs-i-n-main f a xs n* (*length xs*)) =
(λ(*c*, *zs*). (*c*, *y* # *zs*)) ‘ *set* (*subseqs-i-n-main f* (*f y a*) *ys m* (*length ys*))
∪
set (*subseqs-i-n-main f a ys n* (*length ys*))
unfolding *simp* **using** *0 False* **by** (*simp add*: *xs m*)
have *fold*: ⋀ *ys*. *foldr f ys* (*f y a*) = *f y* (*foldr f ys a*)
by (*induct-tac ys*, *auto simp*: *f-ac*)
show *?thesis* **unfolding** *sub IH*[*OF lt le*(*1*)] *IH*[*OF lt le*(*2*)]
unfolding *m xs* **by** (*auto simp*: *Let-def fold*)
qed
qed
qed

qed
qed

definition *basic-subseqs-impl* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a, 'b, 'b \times 'a\ list \times nat)subseqs\text{-}impl$
where
basic-subseqs-impl f = *Sublists-Impl*
$(\lambda\ a\ xs\ n.\ (subseqs\text{-}length\ f\ a\ n\ xs,\ (a,xs,n)))$
$(\lambda\ (a,xs,n).\ (subseqs\text{-}length\ f\ a\ (Suc\ n)\ xs,\ (a,xs,Suc\ n)))$

lemma *basic-subseqs-impl*: **assumes** *f-ac*: $\bigwedge\ x\ y\ z.\ f\ x\ (f\ y\ z) = f\ y\ (f\ x\ z)$
shows *correct-subseqs-impl f* (*basic-subseqs-impl f*)
$(\lambda\ a\ xs\ n\ triple.\ (a,xs,n) = triple)$
by (*unfold-locales*; *unfold subseqs-impl.S-def basic-subseqs-impl-def subseq-of-length-def*, *insert subseqs-length*[*of f*, *OF f-ac*], *auto*)

Improved Implementation **datatype** $('a,'b,'state)$ *subseqs-foldr-impl* = *Sublists-Foldr-Impl*
(*subseqs-foldr*: $'b \Rightarrow 'a\ list \Rightarrow nat \Rightarrow 'b\ list \times 'state$)
(*next-subseqs-foldr*: $'state \Rightarrow 'b\ list \times 'state$)

locale *subseqs-foldr-impl* =
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'b$
and *impl* :: $('a,'b,'state)$ *subseqs-foldr-impl*
begin
definition *S* **where** $S\ base\ elements\ n \equiv \{\ foldr\ f\ ys\ base\ |\ ys.\ subseq\text{-}of\text{-}length\ n\ elements\ ys\ \}$
end

locale *correct-subseqs-foldr-impl* = *subseqs-foldr-impl f impl*
for *f* **and** *impl* :: $('a,'b,'state)$ *subseqs-foldr-impl* +
fixes *invariant* :: $'b \Rightarrow 'a\ list \Rightarrow nat \Rightarrow 'state \Rightarrow bool$
assumes *subseqs-foldr*:
$subseqs\text{-}foldr\ impl\ base\ elements\ n = (out,\ state) \Longrightarrow$
$invariant\ base\ elements\ n\ state \wedge set\ out = S\ base\ elements\ n$
and *next-subseqs-foldr*:
$next\text{-}subseqs\text{-}foldr\ impl\ state = (out,\ state') \Longrightarrow invariant\ base\ elements\ n\ state \Longrightarrow$
$invariant\ base\ elements\ (Suc\ n)\ state' \wedge set\ out = S\ base\ elements\ (Suc\ n)$

locale *my-subseqs* =
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'b$
begin

context fixes *head* :: $'a$ **and** *tail* :: $'a\ iarray$
begin

fun *next-subseqs1* **and** *next-subseqs2*
where $next\text{-}subseqs1\ ret0\ ret1\ [] = (ret0,\ (head,\ tail,\ ret1))$
| $next\text{-}subseqs1\ ret0\ ret1\ ((i,v)\#prevs) = next\text{-}subseqs2\ (f\ head\ v\ \#\ ret0)\ ret1$

prevs v [*0*..<*i*]
  | *next-subseqs2 ret0 ret1 prevs v* [] = *next-subseqs1 ret0 ret1 prevs*
  | *next-subseqs2 ret0 ret1 prevs v* (*j*#*js*) =
    (*let v′* = *f* (*tail* !! *j*) *v in next-subseqs2* (*v′* # *ret0*) ((*j*,*v′*) # *ret1*) *prevs v js*)

definition *next-subseqs2-set v js* ≡ { (*j*, *f* (*tail* !! *j*) *v*) | *j*. *j* ∈ *set js* }

definition *out-subseqs2-set v js* ≡ { *f* (*tail* !! *j*) *v* | *j*. *j* ∈ *set js* }

definition *next-subseqs1-set prevs* ≡ ⋃ { *next-subseqs2-set v* [*0*..<*i*] | *v i*. (*i*,*v*) ∈ *set prevs* }

definition *out-subseqs1-set prevs* ≡
  (*f head* ∘ *snd*) ' *set prevs* ∪ (⋃ { *out-subseqs2-set v* [*0*..<*i*] | *v i*. (*i*,*v*) ∈ *set prevs* })

fun *next-subseqs1-spec* **where**
  next-subseqs1-spec out nexts prevs (*out′*, (*head′*,*tail′*,*nexts′*)) ⟷
   set nexts′ = *set nexts* ∪ *next-subseqs1-set prevs* ∧
   set out′ = *set out* ∪ *out-subseqs1-set prevs*

fun *next-subseqs2-spec* **where**
  next-subseqs2-spec out nexts prevs v js (*out′*, (*head′*,*tail′*,*nexts′*)) ⟷
   set nexts′ = *set nexts* ∪ *next-subseqs1-set prevs* ∪ *next-subseqs2-set v js* ∧
   set out′ = *set out* ∪ *out-subseqs1-set prevs* ∪ *out-subseqs2-set v js*

lemma *next-subseqs2-Cons*:
  next-subseqs2-set v (*j*#*js*) = *insert* (*j*, *f* (*tail*!!*j*) *v*) (*next-subseqs2-set v js*)
  by (*auto simp*: *next-subseqs2-set-def*)

lemma *out-subseqs2-Cons*:
  out-subseqs2-set v (*j*#*js*) = *insert* (*f* (*tail*!!*j*) *v*) (*out-subseqs2-set v js*)
  by (*auto simp*: *out-subseqs2-set-def*)

lemma *next-subseqs1-set-as-next-subseqs2-set*:
  next-subseqs1-set ((*i*,*v*) # *prevs*) = *next-subseqs1-set prevs* ∪ *next-subseqs2-set v* [*0*..<*i*]
  by (*auto simp*: *next-subseqs1-set-def*)

lemma *out-subseqs1-set-as-out-subseqs2-set*:
  out-subseqs1-set ((*i*,*v*) # *prevs*) =
   { *f head v* } ∪ *out-subseqs1-set prevs* ∪ *out-subseqs2-set v* [*0*..<*i*]
  by (*auto simp*: *out-subseqs1-set-def*)

lemma *next-subseqs1-spec*:
  shows ⋀*out nexts*. *next-subseqs1-spec out nexts prevs* (*next-subseqs1 out nexts prevs*)
   and ⋀*out nexts*. *next-subseqs2-spec out nexts prevs v js* (*next-subseqs2 out nexts prevs v js*)

```
proof(induct rule: next-subseqs1-next-subseqs2.induct)
  case (1 ret0 ret1)
  then show ?case by (simp add: next-subseqs1-set-def out-subseqs1-set-def)
next
  case (2 ret0 ret1 i v prevs)
  show ?case
  proof(cases next-subseqs1 out nexts ((i, v) # prevs))
    case split: (fields out' head' tail' nexts')
    have next-subseqs2-spec (f head v # out) nexts prevs v [0..<i] (out', (head',tail',nexts'))
      by (fold split, unfold next-subseqs1.simps, rule 2)
    then show ?thesis
      apply (unfold next-subseqs2-spec.simps split)
     by (auto simp: next-subseqs1-set-as-next-subseqs2-set out-subseqs1-set-as-out-subseqs2-set)
  qed
next
  case (3 ret0 ret1 prevs v)
  show ?case
  proof (cases next-subseqs1 out nexts prevs)
    case split: (fields out' head' tail' nexts')
     from 3[of out nexts] show ?thesis by(simp add: split next-subseqs2-set-def
out-subseqs2-set-def)
  qed
next
  case (4 ret0 ret1 prevs v j js)
  define tj where tj = tail !! j
  define nexts'' where nexts'' = (j, f tj v) # nexts
  define out'' where out'' = (f tj v) # out
  let ?n = next-subseqs2 out'' nexts'' prevs v js
  show ?case
  proof (cases ?n)
    case split: (fields out' head' tail' nexts')
    show ?thesis
      apply (unfold next-subseqs2.simps Let-def)
      apply (fold tj-def)
      apply (fold out''-def nexts''-def)
     apply (unfold split next-subseqs2-spec.simps next-subseqs2-Cons out-subseqs2-Cons)
      using 4[OF refl, of out'' nexts'', unfolded split]
      apply (auto simp: tj-def nexts''-def out''-def)
      done
  qed
qed

end

fun next-subseqs where next-subseqs (head,tail,prevs) = next-subseqs1 head tail []
[] prevs

fun create-subseqs
where create-subseqs base elements 0 = (
```

if elements = [] *then* ([*base*],(*undefined*, *IArray* [], []))
else let head = *hd elements*; *tail* = *IArray* (*tl elements*) *in*
([*base*], (*head*, *tail*, [(*IArray.length tail*, *base*)])))
| *create-subseqs base elements* (*Suc n*) =
next-subseqs (*snd* (*create-subseqs base elements n*))

definition *impl* **where** *impl* = *Sublists-Foldr-Impl create-subseqs next-subseqs*

sublocale *subseqs-foldr-impl f impl* **.**

definition *set-prevs* **where** *set-prevs base tail n* ≡
{ (*i*, *foldr f* (*map* ((!) *tail*) *is*) *base*) | *i is*.
subseq-of-length n [*0*..<*length tail*] *is* ∧ *i* = (*if n* = *0 then length tail else hd is*)
}

lemma *snd-set-prevs*:
snd ‘ (*set-prevs base tail n*) = (λ*as*. *foldr f as base*) ‘ { *as*. *subseq-of-length n tail as* }
by (*subst subseqs-of-length-of-indices*, *auto simp*: *set-prevs-def image-Collect*)

fun *invariant* **where** *invariant base elements n* (*head*,*tail*,*prevs*) =
(*if elements* = [] *then prevs* = []
else head = *hd elements* ∧ *tail* = *IArray* (*tl elements*) ∧ *set prevs* = *set-prevs base* (*tl elements*) *n*)

lemma *next-subseq-preserve*:
assumes *next-subseqs* (*head*,*tail*,*prevs*) = (*out*, (*head′*,*tail′*,*prevs′*))
shows *head′* = *head tail′* = *tail*
proof−
define *P* :: ′*b list* × - × - × (*nat* × ′*b*) *list* ⇒ *bool*
where *P* ≡ λ (*out*, (*head′*,*tail′*,*prevs′*)). *head′* = *head* ∧ *tail′* = *tail*
{ fix *ret0 ret1 v js*
have ∗: *P* (*next-subseqs1 head tail ret0 ret1 prevs*)
and *P* (*next-subseqs2 head tail ret0 ret1 prevs v js*)
by(*induct rule*: *next-subseqs1-next-subseqs2.induct*, *simp add*: *P-def*, *auto simp*: *Let-def*)
}
from *this*(*1*)[*unfolded P-def*, *of* [] [], *folded next-subseqs.simps*] *assms*
show *head′* = *head tail′* = *tail* **by** *auto*
qed

lemma *next-subseqs-spec*:
assumes *nxt*: *next-subseqs* (*head*,*tail*,*prevs*) = (*out*, (*head′*,*tail′*,*prevs′*))
shows *set prevs′* = { (*j*, *f* (*tail* !! *j*) *v*) | *v i j*. (*i*,*v*) ∈ *set prevs* ∧ *j* < *i* } (**is** *?g1*)
and *set out* = (*f head* ∘ *snd*) ‘ *set prevs* ∪ *snd* ‘ *set prevs′* (**is** *?g2*)
proof−
note *next-subseqs1-spec*(*1*)[*of head tail Nil Nil prevs*]

note *this*[*unfolded nxt*[*simplified*]]
note *this*[*unfolded next-subseqs1-spec.simps*]
note *this*[*unfolded next-subseqs1-set-def out-subseqs1-set-def*]
note $*$ = *this*[*unfolded next-subseqs2-set-def out-subseqs2-set-def*]
then show *g1*: *?g1* **by** *auto*
also have *snd* ' ... = $(\bigcup \{\{(f\ (tail\ !!\ j)\ v) \mid j.\ j < i\} \mid v\ i.\ (i,\ v) \in set\ prevs\})$
by (*unfold image-Collect*, *auto*)
finally have $**$: *snd* ' *set prevs'* = ...
with *conjunct2*[*OF* $*$] **show** *?g2* **by** *simp*
qed

lemma *next-subseq-prevs*:
assumes *nxt*: *next-subseqs* (*head*,*tail*,*prevs*) = (*out*, (*head'*,*tail'*,*prevs'*))
and *inv-prevs*: *set prevs* = *set-prevs base* (*IArray.list-of tail*) *n*
shows *set prevs'* = *set-prevs base* (*IArray.list-of tail*) (*Suc n*) (**is** *?l* = *?r*)
proof(*intro equalityI subsetI*)
fix *t*
assume *r*: $t \in$ *?r*
from *this*[*unfolded set-prevs-def*] **obtain** *iis*
where *t*: *t* = (*hd iis*, *foldr f* (*map* ((!!) *tail*) *iis*) *base*)
and *sl*: *subseq-of-length* (*Suc n*) [*0*..<*IArray.length tail*] *iis* **by** *auto*
from *sl* **have** *length iis* > *0* **by** *auto*
then obtain *i is* **where** *iis*: *iis* = *i*#*is* **by** (*meson list.set-cases nth-mem*)
define *v* **where** *v* = *foldr f* (*map* ((!!) *tail*) *is*) *base*
note *sl*[*unfolded subseq-of-length-Suc-upt*]
note *nxt* = *next-subseqs-spec*[*OF nxt*]
show $t \in$ *?l*
proof(*cases n* = *0*)
case *True*
from *sl*[*unfolded subseq-of-length-Suc-upt*] *t*
show *?thesis* **by** (*unfold nxt*[*unfolded inv-prevs*] *True set-prevs-def length-Suc-conv*, *auto*)
next
case [*simp*]: *False*
from *sl*[*unfolded subseq-of-length-Suc-upt iis*,*simplified*]
have *i*: *i* < *hd is* **and** *is*: *subseq-of-length n* [*0*..<*IArray.length tail*] *is* **by** *auto*
then have $*$: (*hd is*, *v*) $\in$ *set-prevs base* (*IArray.list-of tail*) *n*
by (*unfold set-prevs-def*, *auto intro*!: *exI*[*of - is*] *simp*: *v-def*)
with *i* **have** $(i,\ f\ (tail\ !!\ i)\ v) \in \{(j,\ f\ (tail\ !!\ j)\ v) \mid j.\ j < hd\ is\}$ **by** *auto*
with *t*[*unfolded iis*] **have** $t \in$ ... **by** (*auto simp*: *v-def*)
with $*$ **show** *?thesis* **by** (*unfold nxt*[*unfolded inv-prevs*], *auto*)
qed
next
fix *t*
assume *l*: $t \in$ *?l*
from *l*[*unfolded next-subseqs-spec*(*1*)[*OF nxt*]]
obtain *j v i*
where *t*: *t* = (*j*, *f* (*tail*!!*j*) *v*)
and *j*: *j* < *i*

```
    and iv: (i,v) ∈ set prevs by auto
  from iv[unfolded inv-prevs set-prevs-def, simplified]
  obtain is
  where v: v = foldr f (map ((!!) tail) is) base
    and is: subseq-of-length n [0..<IArray.length tail] is
    and i: if n = 0 then i = IArray.length tail else i = hd is by auto
  from is j i have jis: subseq-of-length (Suc n) [0..<IArray.length tail] (j#is)
    by (unfold subseq-of-length-Suc-upt, auto)
  then show t ∈ ?r by (auto intro!: exI[of - j#is] simp: set-prevs-def t v)
qed

lemma invariant-next-subseqs:
  assumes inv: invariant base elements n state
      and nxt: next-subseqs state = (out, state′)
  shows invariant base elements (Suc n) state′
proof(cases elements = [])
  case True with inv nxt show ?thesis by(cases state, auto)
next
  case False with inv nxt show ?thesis
  proof (cases state)
    case state: (fields head tail prevs)
    note inv = inv[unfolded state]
    show ?thesis
    proof (cases state′)
      case state′: (fields head′ tail′ prevs′)
      note nxt = nxt[unfolded state state′]
      note [simp] = next-subseq-preserve[OF nxt]
      from False inv
      have set prevs = set-prevs base (IArray.list-of tail) n by auto
      from False next-subseq-prevs[OF nxt this] inv
      show ?thesis by(auto simp: state′)
    qed
  qed
qed

lemma out-next-subseqs:
  assumes inv: invariant base elements n state
      and nxt: next-subseqs state = (out, state′)
  shows set out = S base elements (Suc n)
proof (cases state)
  case state: (fields head tail prevs)
  show ?thesis
  proof(cases elements = [])
    case True
    with inv nxt show ?thesis by (auto simp: state S-def)
  next
    case elements: False
    show ?thesis
    proof(cases state′)
```

case *state'*: (*fields head' tail' prevs'*)
from *elements inv*[*unfolded state*,*simplified*]
have *head* = *hd elements*
and *tail* = *IArray* (*tl elements*)
and *prevs*: *set prevs* = *set-prevs base* (*tl elements*) *n* **by** *auto*
with *elements* **have** *elements2*: *elements* = *head* # *IArray.list-of tail* **by** *auto*
let *?f* = λ*as.* (*foldr f as base*)
have *set out* = *?f* ` {*ys. subseq-of-length* (*Suc n*) *elements ys*}
proof−
from *invariant-next-subseqs*[*OF inv nxt*, *unfolded state' invariant.simps if-not-P*[*OF elements*]]
have *tail'*: *tail'* = *IArray* (*tl elements*)
and *prevs'*: *set prevs'* = *set-prevs base* (*tl elements*) (*Suc n*) **by** *auto*
note *next-subseqs-spec*(*2*)[*OF nxt*[*unfolded state state'*], *unfolded this*]
note *this*[*folded image-comp*, *unfolded snd-set-prevs*]
also note *prevs*
also note *snd-set-prevs*
also have *f head* ` *?f* ` { *as. subseq-of-length n* (*tl elements*) *as* } =
?f ` *Cons head* ` { *as. subseq-of-length n* (*tl elements*) *as* } **by** (*auto simp*: *image-def*)
also note *image-Un*[*symmetric*]
also have
((#) *head* ` {*as. subseq-of-length n* (*tl elements*) *as*} ∪
{*as. subseq-of-length* (*Suc n*) (*tl elements*) *as*}) =
{*as. subseq-of-length* (*Suc n*) *elements as*}
by (*unfold subseqs-of-length-Suc-Cons elements2*, *auto*)
finally show *?thesis***.**
qed
then show *?thesis* **by** (*auto simp*: *S-def*)
qed
qed
qed

lemma *create-subseqs*:
create-subseqs base elements n = (*out*, *state*) $\Longrightarrow$
invariant base elements n state $\wedge$ *set out* = *S base elements n*
proof(*induct n arbitrary*: *out state*)
case *0* **then show** *?case* **by** (*cases elements*, *cases state*, *auto simp*: *S-def Let-def set-prevs-def*)
next
case (*Suc n*) **show** *?case*
proof (*cases create-subseqs base elements n*)
case *1*: (*fields out'' head tail prevs*)
show *?thesis*
proof (*cases next-subseqs* (*head*, *tail*, *prevs*))
case (*fields out' head' tail' prevs'*)
note *2* = *this*[*unfolded next-subseq-preserve*[*OF this*]]
from *Suc*(*2*)[*unfolded create-subseqs.simps 1 snd-conv 2*]

    have *3*: *out′* = *out state* = (*head*,*tail*,*prevs′*) **by** *auto*
    from *Suc*(*1*)[*OF 1*]
    have *inv*: *invariant base elements n* (*head*, *tail*, *prevs*) **by** *auto*
    from *out-next-subseqs*[*OF inv 2*] *invariant-next-subseqs*[*OF inv 2*]
    show *?thesis* **by** (*auto simp*: *3*)
  qed
 qed
qed

sublocale *correct-subseqs-foldr-impl f impl invariant*
 by (*unfold-locales*; *auto simp*: *impl-def invariant-next-subseqs out-next-subseqs create-subseqs*)

lemma *impl-correct*: *correct-subseqs-foldr-impl f impl invariant* **..**
end

lemmas [*code*] =
 my-subseqs.next-subseqs.simps
 my-subseqs.next-subseqs1.simps
 my-subseqs.next-subseqs2.simps
 my-subseqs.create-subseqs.simps
 my-subseqs.impl-def

end

10.7 Reconstruction of Integer Factorization

We implemented Zassenhaus reconstruction-algorithm, i.e., given a factorization of f mod p^n, the aim is to reconstruct a factorization of f over the integers.

theory *Reconstruction*
imports
 Berlekamp-Hensel
 Polynomial-Factorization.Gauss-Lemma
 Polynomial-Factorization.Dvd-Int-Poly
 Polynomial-Factorization.Gcd-Rat-Poly
 Degree-Bound
 Factor-Bound
 Sublist-Iteration
 Poly-Mod
begin

hide-const *coeff monom*

Misc lemmas **lemma** *foldr-of-Cons*[*simp*]: *foldr Cons xs ys* = *xs* @ *ys* **by** (*induct xs*, *auto*)

lemma *foldr-map-prod*[*simp*]:

foldr ($\lambda x.$ *map-prod* (*f x*) (*g x*)) *xs base* = (*foldr f xs* (*fst base*), *foldr g xs* (*snd base*))
by (*induct xs*, *auto*)

The main part **context** *poly-mod*
begin

definition *inv-Mp* :: *int poly* ⇒ *int poly* **where**
inv-Mp = *map-poly inv-M*

definition *mul-const* :: *int poly* ⇒ *int* ⇒ *int* **where**
mul-const p c = (*coeff p 0* ∗ *c*) *mod m*

fun *prod-list-m* :: *int poly list* ⇒ *int poly* **where**
prod-list-m (*f* # *fs*) = *Mp* (*f* ∗ *prod-list-m fs*)
| *prod-list-m* [] = *1*

context
fixes *sl-impl* :: (*int poly*, *int* × *int poly list*, ′*state*)*subseqs-foldr-impl*
and *m2* :: *int*
begin
definition *inv-M2* :: *int* ⇒ *int* **where**
inv-M2 = (λ *x*. *if x* ≤ *m2 then x else x* − *m*)

definition *inv-Mp2* :: *int poly* ⇒ *int poly* **where**
inv-Mp2 = *map-poly inv-M2*

partial-function (*tailrec*) *reconstruction* :: ′*state* ⇒ *int poly* ⇒ *int poly*
⇒ *int* ⇒ *nat* ⇒ *nat* ⇒ *int poly list* ⇒ *int poly list*
⇒ (*int* × (*int poly list*)) *list* ⇒ *int poly list* **where**
reconstruction state u luu lu d r vs res cands = (*case cands of Nil*
⇒ *let d*′ = *Suc d*
in if d′ + *d*′ > *r then* (*u* # *res*) *else*
(*case next-subseqs-foldr sl-impl state of* (*cands*,*state*′) ⇒
reconstruction state′ *u luu lu d*′ *r vs res cands*)
| (*lv*′,*ws*) # *cands*′ ⇒ *let*
lv = *inv-M2 lv*′ — *lv* is last coefficient of *vb* below
in if lv dvd coeff luu 0 then let
vb = *inv-Mp2* (*Mp* (*smult lu* (*prod-list-m ws*)))
in if vb dvd luu then
let pp-vb = *primitive-part vb*;
u′ = *u div pp-vb*;
r′ = *r* − *length ws*;
res′ = *pp-vb* # *res*
in if d + *d* > *r*′
then u′ # *res*′
else let
lu′ = *lead-coeff u*′;
vs′ = *fold remove1 ws vs*;

$(cands'', state') = subseqs\text{-}foldr\ sl\text{-}impl\ (lu',[])\ vs'\ d$
in reconstruction state′ u′ (smult lu′ u′) lu′ d r′ vs′ res′ cands′′
else reconstruction state u luu lu d r vs res cands′
else reconstruction state u luu lu d r vs res cands′)
end
end

declare *poly-mod.reconstruction.simps*[*code*]
declare *poly-mod.prod-list-m.simps*[*code*]
declare *poly-mod.mul-const-def* [*code*]
declare *poly-mod.inv-M2-def* [*code*]
declare *poly-mod.inv-Mp2-def* [*code-unfold*]
declare *poly-mod.inv-Mp-def* [*code-unfold*]

definition *zassenhaus-reconstruction-generic* ::
(*int poly*, *int* × *int poly list*, ′*state*) *subseqs-foldr-impl*
⇒ *int poly list* ⇒ *int* ⇒ *nat* ⇒ *int poly* ⇒ *int poly list* **where**
zassenhaus-reconstruction-generic sl-impl vs p n f = (*let*
lf = *lead-coeff f*;
pn = *p*^*n*;
(-, *state*) = *subseqs-foldr sl-impl* (*lf*,[]) *vs 0*
in
poly-mod.reconstruction pn sl-impl (*pn div 2*) *state f* (*smult lf f*) *lf 0* (*length vs*) *vs* [] [])

lemma *coeff-mult-0*: *coeff* (*f* ∗ *g*) *0* = *coeff f 0* ∗ *coeff g 0*
by (*metis poly-0-coeff-0 poly-mult*)

lemma *lead-coeff-factor*: **assumes** *u*: *u* = *v* ∗ (*w* :: ′*a* ::*idom poly*)
shows *smult* (*lead-coeff u*) *u* = (*smult* (*lead-coeff w*) *v*) ∗ (*smult* (*lead-coeff v*) *w*)
lead-coeff (*smult* (*lead-coeff w*) *v*) = *lead-coeff u lead-coeff* (*smult* (*lead-coeff v*) *w*) = *lead-coeff u*
unfolding *u* **by** (*auto simp*: *lead-coeff-mult lead-coeff-smult*)

lemma *not-irreducible*$_d$*-lead-coeff-factors*: **assumes** ¬ *irreducible*$_d$ (*u* :: ′*a* :: *idom poly*) *degree u* ≠ *0*
shows ∃ *f g*. *smult* (*lead-coeff u*) *u* = *f* ∗ *g* ∧ *lead-coeff f* = *lead-coeff u* ∧ *lead-coeff g* = *lead-coeff u*
∧ *degree f* < *degree u* ∧ *degree g* < *degree u*
proof −
from *assms*[*unfolded irreducible*$_d$*-def*, *simplified*]
obtain *v w* **where** *deg*: *degree v* < *degree u degree w* < *degree u* **and** *u*: *u* = *v* ∗ *w* **by** *auto*
define *f* **where** *f* = *smult* (*lead-coeff w*) *v*
define *g* **where** *g* = *smult* (*lead-coeff v*) *w*
note *lf* = *lead-coeff-factor*[*OF u*, *folded f-def g-def*]
show *?thesis*

proof (*intro exI conjI*, (*rule lf*)+)
show *degree f* < *degree u degree g* < *degree u* **unfolding** *f-def g-def* **using** *deg u* **by** *auto*
qed
qed

lemma *mset-subseqs-size*: *mset* ‘ {*ys*. *ys* ∈ *set* (*subseqs xs*) ∧ *length ys* = *n*} =
{*ws*. *ws* ⊆# *mset xs* ∧ *size ws* = *n*}
proof (*induct xs arbitrary*: *n*)
case (*Cons x xs n*)
show *?case* (**is** *?l* = *?r*)
proof (*cases n*)
case *0*
thus *?thesis* **by** (*auto simp*: *Let-def*)
next
case (*Suc m*)
have *?r* = {*ws*. *ws* ⊆# *mset* (*x* # *xs*)} ∩ {*ps*. *size ps* = *n*} **by** *auto*
also have {*ws*. *ws* ⊆# *mset* (*x* # *xs*)} = {*ps*. *ps* ⊆# *mset xs*} ∪ ((λ *ps*. *ps* + {#*x*#}) ‘ {*ps*. *ps* ⊆# *mset xs*})
by (*simp add*: *multiset-subset-insert*)
also have ... ∩ {*ps*. *size ps* = *n*} = {*ps*. *ps* ⊆# *mset xs* ∧ *size ps* = *n*}
∪ ((λ *ps*. *ps* + {#*x*#}) ‘ {*ps*. *ps* ⊆# *mset xs* ∧ *size ps* = *m*}) **unfolding** *Suc* **by** *auto*
finally have *id*: *?r* =
{*ps*. *ps* ⊆# *mset xs* ∧ *size ps* = *n*} ∪ (λ*ps*. *ps* + {#*x*#}) ‘ {*ps*. *ps* ⊆# *mset xs* ∧ *size ps* = *m*} **.**
have *?l* = *mset* ‘ {*ys* ∈ *set* (*subseqs xs*). *length ys* = *Suc m*}
∪ *mset* ‘ {*ys* ∈ (#) *x* ‘ *set* (*subseqs xs*). *length ys* = *Suc m*}
unfolding *Suc* **by** (*auto simp*: *Let-def*)
also have *mset* ‘ {*ys* ∈ (#) *x* ‘ *set* (*subseqs xs*). *length ys* = *Suc m*}
= (λ*ps*. *ps* + {#*x*#}) ‘ *mset* ‘ {*ys* ∈ *set* (*subseqs xs*). *length ys* = *m*} **by** *force*
finally have *id'*: *?l* = *mset* ‘ {*ys* ∈ *set* (*subseqs xs*). *length ys* = *Suc m*} ∪
(λ*ps*. *ps* + {#*x*#}) ‘ *mset* ‘ {*ys* ∈ *set* (*subseqs xs*). *length ys* = *m*} **.**
show *?thesis* **unfolding** *id id'* *Cons*[*symmetric*] **unfolding** *Suc* **by** *simp*
qed
qed *auto*

context *poly-mod-2*
begin
lemma *prod-list-m*[*simp*]: *prod-list-m fs* = *Mp* (*prod-list fs*)
by (*induct fs*, *auto*)

lemma *inv-Mp-coeff*: *coeff* (*inv-Mp f*) *n* = *inv-M* (*coeff f n*)
unfolding *inv-Mp-def*
by (*rule coeff-map-poly*, *insert m1*, *auto simp*: *inv-M-def*)

lemma *Mp-inv-Mp-id*[*simp*]: *Mp* (*inv-Mp f*) = *Mp f*
unfolding *poly-eq-iff Mp-coeff inv-Mp-coeff* **by** *simp*

lemma *inv-Mp-rev*: **assumes** *bnd*: $\bigwedge$ *n*. *2* $*$ *abs* (*coeff f n*) < *m*
shows *inv-Mp* (*Mp f*) = *f*
proof (*rule poly-eqI*)
fix *n*
define *c* **where** *c* = *coeff f n*
from *bnd*[*of n*, *folded c-def*] **have** *bnd*: *2* $*$ *abs c* < *m* **by** *auto*
show *coeff* (*inv-Mp* (*Mp f*)) *n* = *coeff f n* **unfolding** *inv-Mp-coeff Mp-coeff c-def*[*symmetric*]
using *inv-M-rev*[*OF bnd*] **.**
qed

lemma *mul-const-commute-below*: *mul-const x* (*mul-const y z*) = *mul-const y* (*mul-const x z*)
unfolding *mul-const-def* **by** (*metis mod-mult-right-eq mult.left-commute*)

context
fixes *p n*
and *sl-impl* :: (*int poly*, *int* × *int poly list*, *'state*)*subseqs-foldr-impl*
and *sli* :: *int* × *int poly list* ⇒ *int poly list* ⇒ *nat* ⇒ *'state* ⇒ *bool*
assumes *prime*: *prime p*
and *m*: *m* = *p*^*n*
and *n*: *n* ≠ *0*
and *sl-impl*: *correct-subseqs-foldr-impl* (λ*x*. *map-prod* (*mul-const x*) (*Cons x*)) *sl-impl sli*
begin
private definition *test-dvd-exec lu u ws* = (¬ *inv-Mp* (*Mp* (*smult lu* (*prod-mset ws*))) *dvd smult lu u*)

private definition *test-dvd u ws* = (∀ *v l*. *v dvd u* ⟶ *0* < *degree v* ⟶ *degree v* < *degree u*
⟶ ¬ *v* =*m smult l* (*prod-mset ws*))

private definition *large-m u vs* = (∀ *v n*. *v dvd u* ⟶ *degree v* ≤ *degree-bound vs* ⟶ *2* $*$ *abs* (*coeff v n*) < *m*)

lemma *large-m-factor*: *large-m u vs* ⟹ *v dvd u* ⟹ *large-m v vs*
unfolding *large-m-def* **using** *dvd-trans* **by** *auto*

lemma *test-dvd-factor*: **assumes** *u*: *u* ≠ *0* **and** *test*: *test-dvd u ws* **and** *vu*: *v dvd u*
shows *test-dvd v ws*
proof −
from *vu* **obtain** *w* **where** *uv*: *u* = *v* $*$ *w* **unfolding** *dvd-def* **by** *auto*
from *u* **have** *deg*: *degree u* = *degree v* + *degree w* **unfolding** *uv*
by (*subst degree-mult-eq*, *auto*)
show *?thesis* **unfolding** *test-dvd-def*
proof (*intro allI impI*, *goal-cases*)
case (*1 f l*)

from *1*(*1*) **have** *fu*: *f dvd u* **unfolding** *uv* **by** *auto*
from *1*(*3*) **have** *deg*: *degree f* < *degree u* **unfolding** *deg* **by** *auto*
from *test*[*unfolded test-dvd-def*, *rule-format*, *OF fu 1*(*2*) *deg*]
show *?case* **.**
qed
qed

lemma *coprime-exp-mod*: *coprime lu p* ⟹ *prime p* ⟹ *n* ≠ *0* ⟹ *lu mod p* ^ *n* ≠ *0*
by (*auto simp add*: *abs-of-pos prime-gt-0-int*)

interpretation *correct-subseqs-foldr-impl λx. map-prod* (*mul-const x*) (*Cons x*) *sl-impl sli* **by** *fact*

lemma *reconstruction*: **assumes**
res: *reconstruction sl-impl m2 state u* (*smult lu u*) *lu d r vs res cands* = *fs*
and *f*: *f* = *u* ∗ *prod-list res*
and *meas*: *meas* = (*r* − *d*, *cands*)
and *dr*: *d* + *d* ≤ *r*
and *r*: *r* = *length vs*
and *cands*: *set cands* ⊆ *S* (*lu*,[]) *vs d*
and *d0*: *d* = *0* ⟹ *cands* = []
and *lu*: *lu* = *lead-coeff u*
and *factors*: *unique-factorization-m u* (*lu*,*mset vs*)
and *sf*: *poly-mod.square-free-m p u*
and *cop*: *coprime lu p*
and *norm*: ⋀ *v*. *v* ∈ *set vs* ⟹ *Mp v* = *v*
and *tests*: ⋀ *ws*. *ws* ⊆# *mset vs* ⟹ *ws* ≠ {#} ⟹
size ws < *d* ∨ *size ws* = *d* ∧ *ws* ∉ (*mset o snd*) ‘ *set cands*
⟹ *test-dvd u ws*
and *irr*: ⋀ *f*. *f* ∈ *set res* ⟹ $irreducible_d$ *f*
and *deg*: *degree u* > *0*
and *cands-ne*: *cands* ≠ [] ⟹ *d* < *r*
and *large*: ∀ *v n*. *v dvd smult lu u* ⟶ *degree v* ≤ *degree-bound vs*
⟶ *2* ∗ *abs* (*coeff v n*) < *m*
and *f0*: *f* ≠ *0*
and *state*: *sli* (*lu*,[]) *vs d state*
and *m2*: *m2* = *m div 2*
shows *f* = *prod-list fs* ∧ (∀ *fi* ∈ *set fs*. $irreducible_d$ *fi*)
proof −
from *large* **have** *large*: *large-m* (*smult lu u*) *vs* **unfolding** *large-m-def* **by** *auto*
interpret *p*: *poly-mod-prime p* **using** *prime* **by** *unfold-locales*
define *R* **where** *R* ≡ *measures* [
λ (*n* :: *nat*,*cds* :: (*int* × *int poly list*) *list*). *n*,
λ (*n*,*cds*). *length cds*]
have *wf*: *wf R* **unfolding** *R-def* **by** *simp*
have *mset-snd-S*: ⋀ *vs lu d*. (*mset* ∘ *snd*) ‘ *S* (*lu*,[]) *vs d* =
{ *ws*. *ws* ⊆# *mset vs* ∧ *size ws* = *d*}
by (*fold mset-subseqs-size image-comp*, *unfold S-def image-Collect*, *auto*)

have *inv-M2*[*simp*]: *inv-M2 m2* = *inv-M* **unfolding** *inv-M2-def m2 inv-M-def*
by (*intro ext*, *auto*)
have *inv-Mp2*[*simp*]: *inv-Mp2 m2* = *inv-Mp* **unfolding** *inv-Mp2-def inv-Mp-def*
by *simp*
have *p-Mp*[*simp*]: $\bigwedge$ *f*. *p.Mp* (*Mp f*) = *p.Mp f* **using** *m p.m1 n Mp-Mp-pow-is-Mp*
by *blast*
{
fix *u lu vs*
assume *eq*: *Mp u* = *Mp* (*smult lu* (*prod-mset vs*)) **and** *cop*: *coprime lu p* **and**
size: *size vs* $\neq$ *0*
and *mi*: $\bigwedge$ *v*. *v* $\in$# *vs* $\Longrightarrow$ *irreducible*$_d$*-m v* $\wedge$ *monic v*
from *cop p.m1* **have** *lu0*: *lu* $\neq$ *0* **by** *auto*
from *size* **have** *vs* $\neq$ {#} **by** *auto*
then obtain *v vs'* **where** *vs-v*: *vs* = *vs'* + {#*v*#} **by** (*cases vs*, *auto*)
have *mon*: *monic* (*prod-mset vs*)
by (*rule monic-prod-mset*, *insert mi*, *auto*)
hence *vs0*: *prod-mset vs* $\neq$ *0* **by** (*metis coeff-0 zero-neq-one*)
from *mon* **have** *l-vs*: *lead-coeff* (*prod-mset vs*) = *1* **.**
have *deg-ws*: *degree-m* (*smult lu* (*prod-mset vs*)) = *degree* (*smult lu* (*prod-mset*
vs))
by (*rule degree-m-eq*[*OF - m1*], *unfold lead-coeff-smult*,
insert cop n p.m1 l-vs, *auto simp*: *m*)
with *eq* **have** *degree-m u* = *degree* (*smult lu* (*prod-mset vs*)) **by** *auto*
also have ... = *degree* (*prod-mset vs'* ∗ *v*) **unfolding** *degree-smult-eq vs-v*
using *lu0* **by** (*simp add*:*ac-simps*)
also have ... = *degree* (*prod-mset vs'*) + *degree v*
by (*rule degree-mult-eq*, *insert vs0*[*unfolded vs-v*], *auto*)
also have ... $\geq$ *degree v* **by** *simp*
finally have *deg-v*: *degree v* $\leq$ *degree-m u* **.**
from *mi*[*unfolded vs-v*, *of v*] **have** *irreducible*$_d$*-m v* **by** *auto*
hence *0* < *degree-m v* **unfolding** *irreducible*$_d$*-m-def* **by** *auto*
also have ... $\leq$ *degree v* **by** (*rule degree-m-le*)
also have ... $\leq$ *degree-m u* **by** (*rule deg-v*)
also have ... $\leq$ *degree u* **by** (*rule degree-m-le*)
finally have *degree u* > *0* **by** *auto*
} **note** *deg-non-zero* = *this*
{
fix *u* :: *int poly* **and** *vs* :: *int poly list* **and** *d* :: *nat*
assume *deg-u*: *degree u* > *0*
and *cop*: *coprime* (*lead-coeff u*) *p*
and *uf*: *unique-factorization-m u* (*lead-coeff u*, *mset vs*)
and *sf*: *p.square-free-m u*
and *norm*: $\bigwedge$ *v*. *v* $\in$ *set vs* $\Longrightarrow$ *Mp v* = *v*
and *d*: *size* (*mset vs*) < *d* + *d*
and *tests*: $\bigwedge$ *ws*. *ws* $\subseteq$# *mset vs* $\Longrightarrow$ *ws* $\neq$ {#} $\Longrightarrow$ *size ws* < *d* $\Longrightarrow$ *test-dvd*
u ws
from *deg-u* **have** *u0*: *u* $\neq$ *0* **by** *auto*
have *irreducible*$_d$ *u*
proof (*rule irreducible*$_d$*I*[*OF deg-u*])

fix *q q′* :: *int poly*
assume *deg*: *degree q* > *0 degree q* < *degree u degree q′* > *0 degree q′* < *degree u*
and *uq*: *u* = *q* ∗ *q′*
then have *qu*: *q dvd u* **and** *q′u*: *q′ dvd u* **by** *auto*
from *u0* **have** *deg-u*: *degree u* = *degree q* + *degree q′* **unfolding** *uq*
by (*subst degree-mult-eq*, *auto*)
from *coprime-lead-coeff-factor*[*OF prime cop*[*unfolded uq*]]
have *cop-q*: *coprime* (*lead-coeff q*) *p coprime* (*lead-coeff q′*) *p* **by** *auto*
from *unique-factorization-m-factor*[*OF prime uf*[*unfolded uq*] - - *n m*, *folded uq*,
OF cop sf]
obtain *fs gs l* **where** *uf-q*: *unique-factorization-m q* (*lead-coeff q*, *fs*)
and *uf-q′*: *unique-factorization-m q′* (*lead-coeff q′*, *gs*)
and *Mf-eq*: *Mf* (*l*, *mset vs*) = *Mf* (*lead-coeff q* ∗ *lead-coeff q′*, *fs* + *gs*)
and *fs-id*: *image-mset Mp fs* = *fs*
and *gs-id*: *image-mset Mp gs* = *gs* **by** *auto*
from *Mf-eq fs-id gs-id* **have** *image-mset Mp* (*mset vs*) = *fs* + *gs*
unfolding *Mf-def* **by** *auto*
also have *image-mset Mp* (*mset vs*) = *mset vs* **using** *norm* **by** (*induct vs*, *auto*)
finally have *eq*: *mset vs* = *fs* + *gs* **by** *simp*
from *uf-q*[*unfolded unique-factorization-m-alt-def factorization-m-def split*]
have *q-eq*: *q* =*m smult* (*lead-coeff q*) (*prod-mset fs*) **by** *auto*
have *degree-m q* = *degree q*
by (*rule degree-m-eq*[*OF* - *m1*], *insert cop-q*(*1*) *n p.m1*, *unfold m*,
auto simp:)
with *q-eq* **have** *degm-q*: *degree q* = *degree* (*Mp* (*smult* (*lead-coeff q*) (*prod-mset fs*))) **by** *auto*
with *deg* **have** *fs-nempty*: *fs* ≠ {#}
by (*cases fs*; *cases lead-coeff q* = *0*; *auto simp*: *Mp-def*)
from *uf-q′*[*unfolded unique-factorization-m-alt-def factorization-m-def split*]
have *q′-eq*: *q′* =*m smult* (*lead-coeff q′*) (*prod-mset gs*) **by** *auto*
have *degree-m q′* = *degree q′*
by (*rule degree-m-eq*[*OF* - *m1*], *insert cop-q*(*2*) *n p.m1*, *unfold m*,
auto simp:)
with *q′-eq* **have** *degm-q′*: *degree q′* = *degree* (*Mp* (*smult* (*lead-coeff q′*) (*prod-mset gs*))) **by** *auto*
with *deg* **have** *gs-nempty*: *gs* ≠ {#}
by (*cases gs*; *cases lead-coeff q′* = *0*; *auto simp*: *Mp-def*)

from *eq* **have** *size*: *size fs* + *size gs* = *size* (*mset vs*) **by** *auto*
with *d* **have** *choice*: *size fs* < *d* ∨ *size gs* < *d* **by** *auto*
from *choice* **show** *False*
proof
assume *fs*: *size fs* < *d*
from *eq* **have** *sub*: *fs* ⊆# *mset vs* **using** *mset-subset-eq-add-left*[*of fs gs*] **by** *auto*
have *test-dvd u fs*

by (*rule tests*[*OF sub fs-nempty*, *unfolded Nil*], *insert fs*, *auto*)
from *this*[*unfolded test-dvd-def*] *uq deg q-eq* **show** *False* **by** *auto*
next
assume *gs*: *size gs* < *d*
from *eq* **have** *sub*: *gs* ⊆# *mset vs* **using** *mset-subset-eq-add-left*[*of fs gs*] **by** *auto*
have *test-dvd u gs*
by (*rule tests*[*OF sub gs-nempty*, *unfolded Nil*], *insert gs*, *auto*)
from *this*[*unfolded test-dvd-def*] *uq deg q′-eq* **show** *False* **unfolding** *uq* **by** *auto*
qed
qed
} **note** *irreducible*$_d$*-via-tests* = *this*
show *?thesis* **using** *assms*(*1*−*16*) *large state*
proof (*induct meas arbitrary*: *u lu d r vs res cands state rule*: *wf-induct*[*OF wf*])
case (*1 meas u lu d r vs res cands state*)
note *IH* = *1*(*1*)[*rule-format*]
note *res* = *1*(*2*)[*unfolded reconstruction.simps*[**where** *cands* = *cands*]]
note *f* = *1*(*3*)
note *meas* = *1*(*4*)
note *dr* = *1*(*5*)
note *r* = *1*(*6*)
note *cands* = *1*(*7*)
note *d0* = *1*(*8*)
note *lu* = *1*(*9*)
note *factors* = *1*(*10*)
note *sf* = *1*(*11*)
note *cop* = *1*(*12*)
note *norm* = *1*(*13*)
note *tests* = *1*(*14*)
note *irr* = *1*(*15*)
note *deg-u* = *1*(*16*)
note *cands-empty* = *1*(*17*)
note *large* = *1*(*18*)
note *state* = *1*(*19*)
from *unique-factorization-m-zero*[*OF factors*]
have *Mlu0*: *M lu* ≠ *0* **by** *auto*
from *Mlu0* **have** *lu0*: *lu* ≠ *0* **by** *auto*
from *this*[*unfolded lu*] **have** *u0*: *u* ≠ *0* **by** *auto*
from *unique-factorization-m-imp-factorization*[*OF factors*]
have *fact*: *factorization-m u* (*lu*,*mset vs*) **by** *auto*
from *this*[*unfolded factorization-m-def split*] *norm*
have *vs*: *u* =*m smult lu* (*prod-list vs*) **and**
vs-mi: ⋀ *f*. *f*∈#*mset vs* ⟹ *irreducible*$_d$*-m f* ∧ *monic f* **by** *auto*
let *?luu* = *smult lu u*
show *?case*
proof (*cases cands*)
case *Nil*
note *res* = *res*[*unfolded this*]

let *?d′ = Suc d*
show *?thesis*
proof (*cases r < ?d′ + ?d′*)
case *True*
with *res* **have** *fs*: *fs = u # res* **by** (*simp add*: *Let-def*)
from *True*[*unfolded r*] **have** *size*: *size* (*mset vs*) < *?d′ + ?d′* **by** *auto*
have *irreducible$_d$ u*
by (*rule irreducible$_d$-via-tests*[*OF deg-u cop*[*unfolded lu*] *factors*(*1*)[*unfolded lu*]
sf norm size tests], *auto simp*: *Nil*)
with *fs f irr* **show** *?thesis* **by** *simp*
next
case *False*
with *dr* **have** *dr*: *?d′ + ?d′ ≤ r* **and** *dr′*: *?d′ < r* **by** *auto*
obtain *state′ cands′* **where** *sln*: *next-subseqs-foldr sl-impl state* = (*cands′*,*state′*) **by** *force*
from *next-subseqs-foldr*[*OF sln state*] **have** *state′*: *sli* (*lu*,[]) *vs* (*Suc d*) *state′*
and *cands′*: *set cands′ = S* (*lu*,[]) *vs* (*Suc d*) **by** *auto*
let *?new = subseqs-length mul-const lu ?d′ vs*
have *R*: ((*r − Suc d*, *cands′*), *meas*) ∈ *R* **unfolding** *meas R-def* **using** *False* **by** *auto*
from *res False sln*
have *fact*: *reconstruction sl-impl m2 state′ u ?luu lu ?d′ r vs res cands′ = fs* **by** *auto*
show *?thesis*
proof (*rule IH*[*OF R fact f refl dr r - - lu factors sf cop norm - irr deg-u dr′ large state′*], *goal-cases*)
case (*4 ws*)
show *?case*
proof (*cases size ws = Suc d*)
case *False*
with *4* **have** *size ws < Suc d* **by** *auto*
thus *?thesis* **by** (*intro tests*[*OF 4*(*1−2*)], *unfold Nil*, *auto*)
next
case *True*
from *4*(*3*)[*unfolded cands′ mset-snd-S*] *True 4*(*1*) **show** *?thesis* **by** *auto*
qed
qed (*auto simp*: *cands′*)
qed
next
case (*Cons c cds*)
with *d0* **have** *d0*: *d > 0* **by** *auto*
obtain *lv′ ws* **where** *c*: *c* = (*lv′*,*ws*) **by** *force*
let *?lv = inv-M lv′*
define *vb* **where** *vb ≡ inv-Mp* (*Mp* (*smult lu* (*prod-list ws*)))
note *res = res*[*unfolded Cons c list.simps split*]
from *cands*[*unfolded Cons c S-def*] **have** *ws*: *ws ∈ set* (*subseqs vs*) *length ws = d*
and *lv″*: *lv′ = foldr mul-const ws lu* **by** *auto*

from *subseqs-sub-mset*[*OF ws*(*1*)] **have** *ws-vs*: *mset ws* ⊆# *mset vs set ws* ⊆ *set vs*
using *set-mset-mono subseqs-length-simple-False* **by** *auto fastforce*
have *mon-ws*: *monic* (*prod-mset* (*mset ws*))
by (*rule monic-prod-mset*, *insert ws-vs vs-mi*, *auto*)
have *l-ws*: *lead-coeff* (*prod-mset* (*mset ws*)) = *1* **using** *mon-ws* **.**
have *lv*′: *M lv*′ = *M* (*coeff* (*smult lu* (*prod-list ws*)) *0*)
unfolding *lv*′′ *coeff-smult*
by (*induct ws arbitrary*: *lu*, *auto simp*: *mul-const-def M-def coeff-mult-0*)
(*metis mod-mult-right-eq mult.left-commute*)
show *?thesis*
proof (*cases ?lv dvd coeff ?luu 0* ∧ *vb dvd ?luu*)
case *False*
have *ndvd*: ¬ *vb dvd ?luu*
proof
assume *dvd*: *vb dvd ?luu*
hence *coeff vb 0 dvd coeff ?luu 0* **by** (*metis coeff-mult-0 dvd-def*)
with *dvd False* **have** *?lv* ≠ *coeff vb 0* **by** *auto*
also have *lv*′ = *M lv*′ **using** *ws*(*2*) *d0* **unfolding** *lv*′′
by (*cases ws*, *force*, *simp add*: *M-def mul-const-def*)
also have *inv-M* (*M lv*′) = *coeff vb 0* **unfolding** *vb-def inv-Mp-coeff Mp-coeff lv*′ **by** *simp*
finally show *False* **by** *simp*
qed
from *False res*
have *res*: *reconstruction sl-impl m2 state u ?luu lu d r vs res cds* = *fs*
unfolding *vb-def Let-def* **by** *auto*
have *R*: ((*r* − *d*, *cds*), *meas*) ∈ *R* **unfolding** *meas Cons R-def* **by** *auto*
from *cands* **have** *cands*: *set cds* ⊆ *S* (*lu*,[]) *vs d*
unfolding *Cons* **by** *auto*
show *?thesis*
proof (*rule IH*[*OF R res f refl dr r cands - lu factors sf cop norm - irr deg-u - large state*], *goal-cases*)
case (*3 ws*′)
show *?case*
proof (*cases ws*′ = *mset ws*)
case *False*
show *?thesis*
by (*rule tests*[*OF 3*(*1*−*2*)], *insert 3*(*3*) *False*, *force simp*: *Cons c*)
next
case *True*
have *test*: *test-dvd-exec lu u ws*′
unfolding *True test-dvd-exec-def* **using** *ndvd* **unfolding** *vb-def* **by** *simp*
show *?thesis* **unfolding** *test-dvd-def*
proof (*intro allI impI notI*, *goal-cases*)
case (*1 v l*)
note *deg-v* = *1*(*2*−*3*)
from *1*(*1*) **obtain** *w* **where** *u*: *u* = *v* ∗ *w* **unfolding** *dvd-def* **by** *auto*

from *u0* **have** *deg*: *degree u* = *degree v* + *degree w* **unfolding** *u*
by (*subst degree-mult-eq*, *auto*)
define v' **where** v' = *smult* (*lead-coeff w*) *v*
define w' **where** w' = *smult* (*lead-coeff v*) *w*
let *?ws* = *smult* (*lead-coeff w* ∗ *l*) (*prod-mset ws*$'$)
from *arg-cong*[*OF 1*(*4*), *of* λ *f*. *Mp* (*smult* (*lead-coeff w*) *f*)]
have v'*-ws*$'$: *Mp* v' = *Mp ?ws* **unfolding** v'*-def*
by *simp*
from *lead-coeff-factor*[*OF u*, *folded* v'*-def* w'*-def*]
have *prod*: *?luu* = v' ∗ w' **and** *lc*: *lead-coeff* v' = *lu* **and** *lead-coeff* w' = *lu*
unfolding *lu* **by** *auto*
with *lu0* **have** *lc0*: *lead-coeff v* ≠ *0* *lead-coeff w* ≠ *0* **unfolding** v'*-def* w'*-def* **by** *auto*
from *deg-v* **have** *deg-w*: *0* < *degree w* *degree w* < *degree u* **unfolding** *deg* **by** *auto*
from *deg-v deg-w lc0*
have *deg*: *0* < *degree* v' *degree* v' < *degree u* *0* < *degree* w' *degree* w' < *degree u*
unfolding v'*-def* w'*-def* **by** *auto*
from *prod* **have** *v-dvd*: v' *dvd ?luu* **by** *auto*
with *test*[*unfolded test-dvd-exec-def*]
have *neq*: v' ≠ *inv-Mp* (*Mp* (*smult lu* (*prod-mset ws*$'$))) **by** *auto*
have *deg-m-v*$'$: *degree-m* v' = *degree* v'
by (*rule degree-m-eq*[*OF - m1*], *unfold lc m*,
insert cop prime n coprime-exp-mod, *auto*)
with v'*-ws*$'$ **have** *degree* v' = *degree-m ?ws* **by** *simp*
also have ... ≤ *degree-m* (*prod-mset ws*$'$) **by** (*rule degree-m-smult-le*)
also have ... = *degree-m* (*prod-list ws*) **unfolding** *True* **by** *simp*
also have ... ≤ *degree* (*prod-list ws*) **by** (*rule degree-m-le*)
also have ... ≤ *degree-bound vs*
using *ws-vs*(*1*) *ws*(*2*) *dr*[*unfolded r*] *degree-bound* **by** *auto*
finally have *degree* v' ≤ *degree-bound vs* **.**
from *inv-Mp-rev*[*OF large*[*unfolded large-m-def*, *rule-format*, *OF v-dvd this*]]
have *inv*: *inv-Mp* (*Mp* v') = v' **by** *simp*
from *arg-cong*[*OF* v'*-ws*$'$, *of inv-Mp*, *unfolded inv*]
have v': v' = *inv-Mp* (*Mp ?ws*) **by** *auto*
have *deg-ws*: *degree-m ?ws* = *degree ?ws*
proof (*rule degree-m-eq*[*OF - m1*],
unfold lead-coeff-smult True l-ws, *rule*)
assume *lead-coeff w* ∗ *l* ∗ *1 mod m* = *0*
hence *0*: *M* (*lead-coeff w* ∗ *l*) = *0* **unfolding** *M-def* **by** *simp*
have *Mp ?ws* = *Mp* (*smult* (*M* (*lead-coeff w* ∗ *l*)) (*prod-mset ws*$'$)) **by** *simp*
also have ... = *0* **unfolding** *0* **by** *simp*
finally have *Mp ?ws* = *0* **by** *simp*
hence v' = *0* **unfolding** v' **by** (*simp add*: *inv-Mp-def*)
with *deg* **show** *False* **by** *auto*

qed
from *arg-cong*[*OF v′*, *of* λ *f*. *lead-coeff* (*Mp f*), *simplified*]
have *M lu* = *M* (*lead-coeff v′*) **using** *lc* **by** *simp*
also have ... = *lead-coeff* (*Mp v′*)
by (*rule degree-m-eq-lead-coeff*[*OF deg-m-v′*, *symmetric*])
also have ... = *lead-coeff* (*Mp ?ws*)
using *arg-cong*[*OF v′*, *of* λ *f*. *lead-coeff* (*Mp f*)] **by** *simp*
also have ... = *M* (*lead-coeff ?ws*)
by (*rule degree-m-eq-lead-coeff*[*OF deg-ws*])
also have ... = *M* (*lead-coeff w* ∗ *l*) **unfolding** *lead-coeff-smult True l-ws* **by** *simp*
finally have *id*: *M lu* = *M* (*lead-coeff w* ∗ *l*) **.**
note *v′*
also have *Mp ?ws* = *Mp* (*smult* (*M* (*lead-coeff w* ∗ *l*)) (*prod-mset ws′*)) **by** *simp*
also have ... = *Mp* (*smult lu* (*prod-mset ws′*)) **unfolding** *id*[*symmetric*] **by** *simp*
finally show *False* **using** *neq* **by** *simp*
qed
qed
qed (*insert d0 Cons cands-empty*, *auto*)
next
case *True*
define *pp-vb* **where** *pp-vb* ≡ *primitive-part vb*
define *u′* **where** *u′* ≡ *u div pp-vb*
define *lu′* **where** *lu′* ≡ *lead-coeff u′*
let *?luu′* = *smult lu′ u′*
define *vs′* **where** *vs′* ≡ *fold remove1 ws vs*
obtain *state′ cands′* **where** *slc*: *subseqs-foldr sl-impl* (*lu′*,[]) *vs′ d* = (*cands′*, *state′*) **by** *force*
from *subseqs-foldr*[*OF slc*] **have** *state′*: *sli* (*lu′*,[]) *vs′ d state′*
and *cands′*: *set cands′* = *S* (*lu′*,[]) *vs′ d* **by** *auto*
let *?res′* = *pp-vb* # *res*
let *?r′* = *r* − *length ws*
note *defs* = *vb-def pp-vb-def u′-def lu′-def vs′-def slc*
from *fold-remove1-mset*[*OF subseqs-sub-mset*[*OF ws*(*1*)]]
have *vs-split*: *mset vs* = *mset vs′* + *mset ws* **unfolding** *vs′-def* **by** *auto*
hence *vs′-diff*: *mset vs′* = *mset vs* − *mset ws* **and** *ws-sub*: *mset ws* ⊆# *mset vs* **by** *auto*
from *arg-cong*[*OF vs-split*, *of size*]
have *r′*: *?r′* = *length vs′* **unfolding** *defs r* **by** *simp*
from *arg-cong*[*OF vs-split*, *of prod-mset*]
have *prod-vs*: *prod-list vs* = *prod-list vs′* ∗ *prod-list ws* **by** *simp*
from *arg-cong*[*OF vs-split*, *of set-mset*] **have** *set-vs*: *set vs* = *set vs′* ∪ *set ws* **by** *auto*
note *inv* = *inverse-mod-coprime-exp*[*OF m prime n*]
note *p-inv* = *p.inverse-mod-coprime*[*OF prime*]
from *True res slc*
have *res*: (*if ?r′* < *d* + *d then u′* # *?res′ else reconstruction sl-impl m2*

state'
u' ?luu' lu' d ?r' vs' ?res' cands') = *fs*
unfolding *Let-def defs* **by** *auto*
from *True* **have** *dvd*: *vb dvd ?luu* **by** *simp*
from *dvd-smult-int*[*OF lu0 this*] **have** *ppu*: *pp-vb dvd u* **unfolding** *defs* **by** *simp*
hence *u*: *u* = *pp-vb* ∗ *u'* **unfolding** *u'-def*
by (*metis dvdE mult-eq-0-iff nonzero-mult-div-cancel-left*)
hence *uu'*: *u' dvd u* **unfolding** *dvd-def* **by** *auto*
have *f*: *f* = *u'* ∗ *prod-list ?res'* **using** *f u* **by** *auto*
let *?fact* = *smult lu* (*prod-mset* (*mset ws*))
have *Mp-vb*: *Mp vb* = *Mp* (*smult lu* (*prod-list ws*)) **unfolding** *vb-def* **by** *simp*
have *pp-vb-vb*: *smult* (*content vb*) *pp-vb* = *vb* **unfolding** *pp-vb-def* **by** (*rule content-times-primitive-part*)
{
have *smult* (*content vb*) *u* = (*smult* (*content vb*) *pp-vb*) ∗ *u'* **unfolding** *u* **by** *simp*
also have *smult* (*content vb*) *pp-vb* = *vb* **by** *fact*
finally have *smult* (*content vb*) *u* = *vb* ∗ *u'* **by** *simp*
from *arg-cong*[*OF this*, *of Mp*]
have *Mp* (*Mp vb* ∗ *u'*) = *Mp* (*smult* (*content vb*) *u*) **by** *simp*
hence *Mp* (*smult* (*content vb*) *u*) = *Mp* (*?fact* ∗ *u'*) **unfolding** *Mp-vb* **by** *simp*
} **note** *prod* = *this*
from *arg-cong*[*OF this*, *of p.Mp*]
have *prod'*: *p.Mp* (*smult* (*content vb*) *u*) = *p.Mp* (*?fact* ∗ *u'*) **by** *simp*
from *dvd* **have** *lead-coeff vb dvd lead-coeff* (*smult lu u*)
by (*metis dvd-def lead-coeff-mult*)
hence *ldvd*: *lead-coeff vb dvd lu* ∗ *lu* **unfolding** *lead-coeff-smult lu* **by** *simp*
from *cop* **have** *cop-lu*: *coprime* (*lu* ∗ *lu*) *p*
by *simp*
from *coprime-divisors* [*OF ldvd dvd-refl*] *cop-lu*
have *cop-lvb*: *coprime* (*lead-coeff vb*) *p* **by** *simp*
then have *cop-vb*: *coprime* (*content vb*) *p*
by (*auto intro*: *coprime-divisors*[*OF content-dvd-coeff dvd-refl*])
from *u* **have** *u' dvd u* **unfolding** *dvd-def* **by** *auto*
hence *lead-coeff u' dvd lu* **unfolding** *lu* **by** (*metis dvd-def lead-coeff-mult*)
from *coprime-divisors*[*OF this dvd-refl*] *cop*
have *coprime* (*lead-coeff u'*) *p* **by** *simp*
hence *coprime* (*lu* ∗ *lead-coeff u'*) *p* **and** *cop-lu'*: *coprime lu' p*
using *cop* **by** (*auto simp*: *lu'-def*)
hence *cop'*: *coprime* (*lead-coeff* (*?fact* ∗ *u'*)) *p*
unfolding *lead-coeff-mult lead-coeff-smult l-ws* **by** *simp*
have *p.square-free-m* (*smult* (*content vb*) *u*) **using** *cop-vb sf p-inv*
by (*auto intro*!: *p.square-free-m-smultI*)
from *p.square-free-m-cong*[*OF this prod'*]
have *sf'*: *p.square-free-m* (*?fact* ∗ *u'*) **by** *simp*
from *p.square-free-m-factor*[*OF this*]

have *sf-u′*: *p.square-free-m u′* **by** *simp*
have *unique-factorization-m* (*smult* (*content vb*) *u*) (*lu* ∗ *content vb*, *mset vs*)
using *cop-vb factors inv* **by** (*auto intro*: *unique-factorization-m-smult*)
from *unique-factorization-m-cong*[*OF this prod*]
have *uf*: *unique-factorization-m* (*?fact* ∗ *u′*) (*lu* ∗ *content vb*, *mset vs*) **.**
{
from *unique-factorization-m-factor*[*OF prime uf cop′ sf′ n m*]
obtain *fs gs* **where** *uf1*: *unique-factorization-m ?fact* (*lu*, *fs*)
and *uf2*: *unique-factorization-m u′* (*lu′*, *gs*)
and *eq*: *Mf* (*lu* ∗ *content vb*, *mset vs*) = *Mf* (*lu* ∗ *lead-coeff u′*, *fs* + *gs*)
unfolding *lead-coeff-smult l-ws lu′-def*
by *auto*
have *factorization-m ?fact* (*lu*, *mset ws*)
unfolding *factorization-m-def split* **using** *set-vs vs-mi norm* **by** *auto*
with *uf1*[*unfolded unique-factorization-m-alt-def*] **have** *Mf* (*lu*,*mset ws*) = *Mf* (*lu*, *fs*)
by *blast*
hence *fs-ws*: *image-mset Mp fs* = *image-mset Mp* (*mset ws*) **unfolding** *Mf-def split* **by** *auto*
from *eq*[*unfolded Mf-def split*]
have *image-mset Mp* (*mset vs*) = *image-mset Mp fs* + *image-mset Mp gs* **by** *auto*
from *this*[*unfolded fs-ws vs-split*] **have** *gs*: *image-mset Mp gs* = *image-mset Mp* (*mset vs′*)
by (*simp add*: *ac-simps*)
from *uf1* **have** *uf1*: *unique-factorization-m ?fact* (*lu*, *mset ws*)
unfolding *unique-factorization-m-def Mf-def split fs-ws* **by** *simp*
from *uf2* **have** *uf2*: *unique-factorization-m u′* (*lu′*, *mset vs′*)
unfolding *unique-factorization-m-def Mf-def split gs* **by** *simp*
note *uf1 uf2*
}
hence *factors*: *unique-factorization-m u′* (*lu′*, *mset vs′*)
unique-factorization-m ?fact (*lu*, *mset ws*) **by** *auto*
have *lu′*: *lu′* = *lead-coeff u′* **unfolding** *lu′-def* **by** *simp*
have *vb0*: *vb* ≠ *0* **using** *dvd lu0 u0* **by** *auto*
from *ws*(*2*) **have** *size-ws*: *size* (*mset ws*) = *d* **by** *auto*
with *d0* **have** *size-ws0*: *size* (*mset ws*) ≠ *0* **by** *auto*
then obtain *w ws′* **where** *ws-w*: *ws* = *w* # *ws′* **by** (*cases ws*, *auto*)
from *Mp-vb* **have** *Mp-vb′*: *Mp vb* = *Mp* (*smult lu* (*prod-mset* (*mset ws*))) **by** *auto*
have *deg-vb*: *degree vb* > *0*
by (*rule deg-non-zero*[*OF Mp-vb′ cop size-ws0 vs-mi*], *insert vs-split*, *auto*)
also have *degree vb* = *degree pp-vb* **using** *arg-cong*[*OF pp-vb-vb*, *of degree*]
unfolding *degree-smult-eq* **using** *vb0* **by** *auto*
finally have *deg-pp*: *degree pp-vb* > *0* **by** *auto*
hence *pp-vb0*: *pp-vb* ≠ *0* **by** *auto*
from *factors*(*1*)[*unfolded unique-factorization-m-alt-def factorization-m-def*]
have *eq-u′*: *Mp u′* = *Mp* (*smult lu′* (*prod-mset* (*mset vs′*))) **by** *auto*

from *r'*[*unfolded ws*(*2*)] *dr* **have** *length vs'* + *d* = *r* **by** *auto*
from *this cands-empty*[*unfolded Cons*] **have** *size* (*mset vs'*) ≠ *0* **by** *auto*
from *deg-non-zero*[*OF eq-u' cop-lu' this vs-mi*]
have *deg-u'*: *degree u'* > *0* **unfolding** *vs-split* **by** *auto*
have *irr-pp*: $irreducible_d$ *pp-vb*
proof (*rule* $irreducible_dI$[*OF deg-pp*])
fix *q r* :: *int poly*
assume *deg-q*: *degree q* > *0 degree q* < *degree pp-vb*
and *deg-r*: *degree r* > *0 degree r* < *degree pp-vb*
and *pp-qr*: *pp-vb* = *q* ∗ *r*
then have *qvb*: *q dvd pp-vb* **by** *auto*
from *dvd-trans*[*OF qvb ppu*] **have** *qu*: *q dvd u* **.**
have *degree pp-vb* = *degree q* + *degree r* **unfolding** *pp-qr*
by (*subst degree-mult-eq*, *insert pp-qr pp-vb0*, *auto*)
have *uf*: *unique-factorization-m* (*smult* (*content vb*) *pp-vb*) (*lu*, *mset ws*)
unfolding *pp-vb-vb*
by (*rule unique-factorization-m-cong*[*OF factors*(*2*)], *insert Mp-vb*, *auto*)
from *unique-factorization-m-smultD*[*OF uf inv*] *cop-vb*
have *uf*: *unique-factorization-m pp-vb* (*lu* ∗ *inverse-mod* (*content vb*) *m*, *mset ws*) **by** *auto*
from *ppu* **have** *lead-coeff pp-vb dvd lu* **unfolding** *lu* **by** (*metis dvd-def lead-coeff-mult*)
from *coprime-divisors*[*OF this dvd-refl*] *cop*
have *cop-pp*: *coprime* (*lead-coeff pp-vb*) *p* **by** *simp*
from *coprime-lead-coeff-factor*[*OF prime cop-pp*[*unfolded pp-qr*]]
have *cop-qr*: *coprime* (*lead-coeff q*) *p coprime* (*lead-coeff r*) *p* **by** *auto*
from *p.square-free-m-factor*[*OF sf*[*unfolded u*]]
have *sf-pp*: *p.square-free-m pp-vb* **by** *simp*
from *unique-factorization-m-factor*[*OF prime uf*[*unfolded pp-qr*] - - *n m*, *folded pp-qr*, *OF cop-pp sf-pp*]
obtain *fs gs l* **where** *uf-q*: *unique-factorization-m q* (*lead-coeff q*, *fs*)
and *uf-r*: *unique-factorization-m r* (*lead-coeff r*, *gs*)
and *Mf-eq*: *Mf* (*l*, *mset ws*) = *Mf* (*lead-coeff q* ∗ *lead-coeff r*, *fs* + *gs*)
and *fs-id*: *image-mset Mp fs* = *fs*
and *gs-id*: *image-mset Mp gs* = *gs* **by** *auto*
from *Mf-eq* **have** *image-mset Mp* (*mset ws*) = *image-mset Mp fs* + *image-mset Mp gs*
unfolding *Mf-def* **by** *auto*
also have *image-mset Mp* (*mset ws*) = *mset ws* **using** *norm ws-vs*(*2*) **by** (*induct ws*, *auto*)
finally have *eq*: *mset ws* = *image-mset Mp fs* + *image-mset Mp gs* **by** *simp*
from *arg-cong*[*OF this*, *of size*, *unfolded size-ws*] **have** *size*: *size fs* + *size gs* = *d* **by** *auto*
from *uf-q*[*unfolded unique-factorization-m-alt-def factorization-m-def split*]
have *q-eq*: *q* =*m smult* (*lead-coeff q*) (*prod-mset fs*) **by** *auto*
have *degree-m q* = *degree q*
by (*rule degree-m-eq*[*OF* - *m1*], *insert cop-qr*(*1*) *n p.m1*, *unfold m*, *auto simp*:)

with *q-eq* **have** *degm-q*: *degree q* = *degree* (*Mp* (*smult* (*lead-coeff q*) (*prod-mset fs*))) **by** *auto*
with *deg-q* **have** *fs-nempty*: *fs* ≠ {#}
by (*cases fs*; *cases lead-coeff q* = *0*; *auto simp*: *Mp-def*)
from *uf-r*[*unfolded unique-factorization-m-alt-def factorization-m-def split*]
have *r-eq*: *r* =*m smult* (*lead-coeff r*) (*prod-mset gs*) **by** *auto*
have *degree-m r* = *degree r*
by (*rule degree-m-eq*[*OF - m1*], *insert cop-qr*(*2*) *n p.m1*, *unfold m*,
auto simp:)
with *r-eq* **have** *degm-r*: *degree r* = *degree* (*Mp* (*smult* (*lead-coeff r*) (*prod-mset gs*))) **by** *auto*
with *deg-r* **have** *gs-nempty*: *gs* ≠ {#}
by (*cases gs*; *cases lead-coeff r* = *0*; *auto simp*: *Mp-def*)
from *gs-nempty* **have** *size gs* ≠ *0* **by** *auto*
with *size* **have** *size-fs*: *size fs* < *d* **by** *linarith*
note ∗ = *tests*[*unfolded test-dvd-def*, *rule-format*, *OF - fs-nempty - qu*, *of lead-coeff q*]
from *ppu* **have** *degree pp-vb* ≤ *degree u*
using *dvd-imp-degree-le u0* **by** *blast*
with *deg-q q-eq size-fs*
have ¬ *fs* ⊆# *mset vs* **by** (*auto dest*!:∗)
thus *False* **unfolding** *vs-split eq fs-id gs-id* **using** *mset-subset-eq-add-left*[*of fs mset vs′* + *gs*]
by (*auto simp*: *ac-simps*)
qed
{
fix *ws′*
assume ∗: *ws′* ⊆# *mset vs′ ws′* ≠ {#}
size ws′ < *d* ∨ *size ws′* = *d* ∧ *ws′* ∉ (*mset* ∘ *snd*) ‘ *set cands′*
from ∗(*1*) **have** *ws′* ⊆# *mset vs* **unfolding** *vs-split*
by (*simp add*: *subset-mset.add-increasing2*)
from *tests*[*OF this* ∗(*2*)] ∗(*3*)[*unfolded cands′ mset-snd-S*] ∗(*1*)
have *test-dvd u ws′* **by** *auto*
from *test-dvd-factor*[*OF u0 this*[*unfolded lu*] *uu′*]
have *test-dvd u′ ws′* **.**
} **note** *tests′* = *this*
show *?thesis*
proof (*cases ?r′* < *d* + *d*)
case *True*
with *res* **have** *res*: *fs* = *u′* # *?res′* **by** *auto*
from *True r′* **have** *size*: *size* (*mset vs′*) < *d* + *d* **by** *auto*
have *irreducible$_d$ u′*
by (*rule irreducible$_d$-via-tests*[*OF deg-u′ cop-lu′*[*unfolded lu′*] *factors*(*1*)[*unfolded lu′*]
sf-u′ norm size tests′], *insert set-vs*, *auto*)
with *f res irr irr-pp* **show** *?thesis* **by** *auto*
next
case *False*
have *res*: *reconstruction sl-impl m2 state′ u′ ?luu′ lu′ d ?r′ vs′ ?res′ cands′*

= *fs*
using *False res* **by** *auto*
from *False* **have** *dr*: $d + d \leq ?r'$ **by** *auto*
from *False dr r r' d0 ws Cons* **have** *le*: $?r' - d < r - d$ **by** (*cases ws*, *auto*)
hence *R*: $((?r' - d, cands'), meas) \in R$ **unfolding** *meas R-def* **by** *simp*
have *dr'*: $d < ?r'$ **using** *le False ws*(*2*) **by** *linarith*
have *luu'*: *lu' dvd lu* **using** ‹*lead-coeff u' dvd lu*› **unfolding** *lu'* **.**
have *large-m* (*smult lu' u'*) *vs*
by (*rule large-m-factor*[*OF large dvd-dvd-smult*], *insert uu' luu'*)
moreover have *degree-bound vs'* ≤ *degree-bound vs*
unfolding *vs'-def degree-bound-def* **by** (*rule max-factor-degree-mono*)
ultimately have *large'*: *large-m* (*smult lu' u'*) *vs'* **unfolding** *large-m-def* **by** *auto*
show *?thesis*
by (*rule IH*[*OF R res f refl dr r' - - lu' factors*(*1*) *sf-u' cop-lu' norm tests' - deg-u'*
dr' large' state'], *insert irr irr-pp d0 Cons set-vs*, *auto simp*: *cands'*)
qed
qed
qed
qed
qed
end
end

definition *zassenhaus-reconstruction* ::
int poly list ⇒ *int* ⇒ *nat* ⇒ *int poly* ⇒ *int poly list* **where**
zassenhaus-reconstruction vs p n f = (*let*
mul = *poly-mod.mul-const* ($p \hat{~} n$);
sl-impl = *my-subseqs.impl* (λ*x*. *map-prod* (*mul x*) (*Cons x*))
in zassenhaus-reconstruction-generic sl-impl vs p n f)

context
fixes *p n f hs*
assumes *prime*: *prime p*
and *cop*: *coprime* (*lead-coeff f*) *p*
and *sf*: *poly-mod.square-free-m p f*
and *deg*: *degree f* > *0*
and *bh*: *berlekamp-hensel p n f* = *hs*
and *bnd*: *2* ∗ |*lead-coeff f*| ∗ *factor-bound f* (*degree-bound hs*) < *p* ^ *n*
begin

private lemma *n*: $n \neq 0$
proof
assume *n*: $n = 0$
hence *pn*: $p \hat{~} n = 1$ **by** *auto*
let *?f* = *smult* (*lead-coeff f*) *f*

let *?d* = *degree-bound hs*
have *f*: *f* ≠ *0* **using** *deg* **by** *auto*
hence *lead-coeff f* ≠ *0* **by** *auto*
hence *lf*: *abs* (*lead-coeff f*) > *0* **by** *auto*
obtain *c d* **where** *c*: *factor-bound f* (*degree-bound hs*) = *c abs* (*lead-coeff f*) = *d* **by** *auto*
{
assume ∗: *1* ≤ *c 2* ∗ *d* ∗ *c* < *1 0* < *d*
hence *1* ≤ *d* **by** *auto*
from *mult-mono*[*OF this* ∗(*1*)] ∗ **have** *1* ≤ *d* ∗ *c* **by** *auto*
hence *2* ∗ *d* ∗ *c* ≥ *2* **by** *auto*
with ∗ **have** *False* **by** *auto*
} **note** *tedious* = *this*
have *1* ≤ *factor-bound f ?d*
using *factor-bound*[*OF f*, *of 1 ?d 0*] **by** *auto*
also have ... = *0* **using** *bnd* **unfolding** *pn*
using *factor-bound-ge-0*[*of f degree-bound hs*, *OF f*] *lf* **unfolding** *c*
by (*cases c* ≥ *1*; *insert tedious*, *auto*)
finally show *False* **by** *simp*
qed

interpretation *p*: *poly-mod-prime p* **using** *prime* **by** *unfold-locales*

lemma *zassenhaus-reconstruction-generic*:
assumes *sl-impl*: *correct-subseqs-foldr-impl* (λ*v*. *map-prod* (*poly-mod.mul-const* (*p*^*n*) *v*) (*Cons v*)) *sl-impl sli*
and *res*: *zassenhaus-reconstruction-generic sl-impl hs p n f* = *fs*
shows *f* = *prod-list fs* ∧ (∀ *fi* ∈ *set fs*. *irreducible$_d$ fi*)
proof −
let *?lc* = *lead-coeff f*
let *?ff* = *smult ?lc f*
let *?q* = *p*^*n*
have *p1*: *p* > *1* **using** *prime* **unfolding** *prime-int-iff* **by** *simp*
interpret *poly-mod-2 p*^*n* **using** *p1 n* **unfolding** *poly-mod-2-def* **by** *simp*
obtain *cands state* **where** *slc*: *subseqs-foldr sl-impl* (*lead-coeff f*, []) *hs 0* = (*cands*, *state*) **by** *force*
interpret *correct-subseqs-foldr-impl* λ*x*. *map-prod* (*mul-const x*) (*Cons x*) *sl-impl sli* **by** *fact*
from *subseqs-foldr*[*OF slc*] **have** *state*: *sli* (*lead-coeff f*, []) *hs 0 state* **by** *auto*
from *res*[*unfolded zassenhaus-reconstruction-generic-def bh split Let-def slc fst-conv*]
have *res*: *reconstruction sl-impl* (*?q div 2*) *state f ?ff ?lc 0* (*length hs*) *hs* [] [] = *fs* **by** *auto*
from *p.berlekamp-hensel-unique*[*OF cop sf bh n*]
have *ufact*: *unique-factorization-m f* (*?lc*, *mset hs*) **by** *simp*
note *bh* = *p.berlekamp-hensel*[*OF cop sf bh n*]
from *deg* **have** *f0*: *f* ≠ *0* **and** *lf0*: *?lc* ≠ *0* **by** *auto*
hence *ff0*: *?ff* ≠ *0* **by** *auto*
have *bnd*: ∀ *g k*. *g dvd ?ff* ⟶ *degree g* ≤ *degree-bound hs* ⟶ *2* ∗ |*coeff g k*| < *p* ^ *n*

proof (*intro allI impI*, *goal-cases*)
  case (*1 g k*)
  from *factor-bound-smult*[*OF f0 lf0 1*, *of k*]
  have $|coeff\ g\ k| \leq |?lc| * factor\text{-}bound\ f\ (degree\text{-}bound\ hs)$ **.**
  hence $2 * |coeff\ g\ k| \leq 2 * |?lc| * factor\text{-}bound\ f\ (degree\text{-}bound\ hs)$ **by** *auto*
  also have $\ldots < p\hat{\ }n$ **using** *bnd* **.**
  finally show *?case* **.**
qed
note $bh' = bh$[*unfolded factorization-m-def split*]
have *deg-f*: *degree-m f* = *degree f*
  using *cop unique-factorization-m-zero* [*OF ufact*] *n*
  by (*auto simp add*: *M-def intro*: *degree-m-eq* [*OF - m1*])
have *mon-hs*: *monic* (*prod-list hs*) **using** bh' **by** (*auto intro*: *monic-prod-list*)
have *Mlc*: *M ?lc* $\in$ {*1* ..< $p\hat{\ }n$}
  by (*rule prime-cop-exp-poly-mod*[*OF prime cop n*])
hence *?lc* $\neq$ *0* **by** *auto*
hence *f0*: *f* $\neq$ *0* **by** *auto*
have *degm*: *degree-m* (*smult ?lc* (*prod-list hs*)) = *degree* (*smult ?lc* (*prod-list hs*))

  by (*rule degree-m-eq*[*OF - m1*], *insert n bh mon-hs Mlc*, *auto simp*: *M-def*)
from *reconstruction*[*OF prime refl n sl-impl res - refl - refl - refl refl ufact sf*
    cop - - - deg - bnd f0] *bh*(*2*) *state*
show *?thesis* **by** *simp*
qed

lemma *zassenhaus-reconstruction-irreducible*$_d$:
  assumes *res*: *zassenhaus-reconstruction hs p n f* = *fs*
  shows *f* = *prod-list fs* $\wedge$ ($\forall$ *fi* $\in$ *set fs*. *irreducible*$_d$ *fi*)
  by (*rule zassenhaus-reconstruction-generic*[*OF my-subseqs.impl-correct*
    res[*unfolded zassenhaus-reconstruction-def Let-def*]])

corollary *zassenhaus-reconstruction*:
  assumes *pr*: *primitive f*
  assumes *res*: *zassenhaus-reconstruction hs p n f* = *fs*
  shows *f* = *prod-list fs* $\wedge$ ($\forall$ *fi* $\in$ *set fs*. *irreducible fi*)
  using *zassenhaus-reconstruction-irreducible*$_d$[*OF res*] *pr*
  irreducible-primitive-connect[*OF primitive-prod-list*]
  by *auto*
end

end

theory *Code-Abort-Gcd*
imports
  HOL−Computational-Algebra.Polynomial-Factorial
begin

Dummy code-setup for *Gcd* and *Lcm* in the presence of Container.

definition *dummy-Gcd* **where** *dummy-Gcd x* = *Gcd x*

definition *dummy-Lcm* **where** *dummy-Lcm* x = *Lcm* x
declare [[*code abort*: *dummy-Gcd*]]

lemma *dummy-Gcd-Lcm*: *Gcd* x = *dummy-Gcd* x *Lcm* x = *dummy-Lcm* x
  unfolding *dummy-Gcd-def dummy-Lcm-def* **by** *auto*

lemmas *dummy-Gcd-Lcm-poly* [*code*] = *dummy-Gcd-Lcm*
  [**where** *?'a* = *'a* :: {*factorial-ring-gcd*,*semiring-gcd-mult-normalize*} *poly*]
lemmas *dummy-Gcd-Lcm-int* [*code*] = *dummy-Gcd-Lcm* [**where** *?'a* = *int*]
lemmas *dummy-Gcd-Lcm-nat* [*code*] = *dummy-Gcd-Lcm* [**where** *?'a* = *nat*]

declare [[*code abort*: *Euclidean-Algorithm.Gcd Euclidean-Algorithm.Lcm*]]

end

11 The Polynomial Factorization Algorithm

11.1 Factoring Square-Free Integer Polynomials

We combine all previous results, i.e., Berlekamp's algorithm, Hensel-lifting, the reconstruction of Zassenhaus, Mignotte-bounds, etc., to eventually assemble the factorization algorithm for integer polynomials.

theory *Berlekamp-Zassenhaus*
imports
  Berlekamp-Hensel
  Polynomial-Factorization.Gauss-Lemma
  Polynomial-Factorization.Dvd-Int-Poly
  Reconstruction
  Suitable-Prime
  Degree-Bound
  Code-Abort-Gcd
begin

context
begin
private partial-function (*tailrec*) *find-exponent-main* :: *int* ⇒ *int* ⇒ *nat* ⇒ *int* ⇒ *nat* **where**
  [*code*]: *find-exponent-main p pm m bnd* = (*if pm* > *bnd then m*
    else find-exponent-main p (*pm* ∗ *p*) (*Suc m*) *bnd*)

definition *find-exponent* :: *int* ⇒ *int* ⇒ *nat* **where**
  find-exponent p bnd = *find-exponent-main p p 1 bnd*

lemma *find-exponent*: **assumes** *p*: *p* > *1*
  shows *p* ^ *find-exponent p bnd* > *bnd find-exponent p bnd* ≠ *0*
proof −
  {
    fix *m* **and** *n*

assume $n = nat\ (1 + bnd - p\hat{\ }m)$ **and** $m \geq 1$
hence $bnd < p\ \hat{\ }$ *find-exponent-main* $p\ (p\hat{\ }m)\ m\ bnd \wedge$ *find-exponent-main* $p\ (p\hat{\ }m)\ m\ bnd \geq 1$
proof (*induct n arbitrary*: *m rule*: *less-induct*)
case (*less n m*)
note *simp* = *find-exponent-main.simps*[*of p p^m*]
show *?case*
proof (*cases* $bnd < p\ \hat{\ }\ m$)
case *True*
thus *?thesis* **using** *less* **unfolding** *simp* **by** *simp*
next
case *False*
hence *id*: *find-exponent-main* $p\ (p\ \hat{\ }\ m)\ m\ bnd =$ *find-exponent-main* $p\ (p\ \hat{\ }\ Suc\ m)\ (Suc\ m)\ bnd$
unfolding *simp* **by** (*simp add*: *ac-simps*)
show *?thesis* **unfolding** *id*
by (*rule less*(*1*)[*OF - refl*], *unfold less*(*2*), *insert False p*, *auto*)
qed
qed
}
from *this*[*OF refl*, *of 1*]
show $p\ \hat{\ }$ *find-exponent* $p\ bnd > bnd$ *find-exponent* $p\ bnd \neq 0$
unfolding *find-exponent-def* **by** *auto*
qed

end

definition *berlekamp-zassenhaus-factorization* :: *int poly* $\Rightarrow$ *int poly list* **where**
berlekamp-zassenhaus-factorization f = (*let*
— find suitable prime
p = *suitable-prime-bz f*;
— compute finite field factorization
(-, *fs*) = *finite-field-factorization-int p f*;
— determine maximal degree that we can build by multiplying at most half of the factors
max-deg = *degree-bound fs*;
— determine a number large enough to represent all coefficients of every
— factor of $lc * f$ that has at most degree most *max-deg*
$bnd = 2 * |$*lead-coeff f*$| *$ *factor-bound f max-deg*;
— determine k such that $p\hat{\ }k > bnd$
k = *find-exponent p bnd*;
— perform hensel lifting to lift factorization to mod $p\hat{\ }k$
vs = *hensel-lifting p k f fs*
— reconstruct integer factors
in zassenhaus-reconstruction vs p k f)

theorem *berlekamp-zassenhaus-factorization-irreducible*$_d$:
assumes *res*: *berlekamp-zassenhaus-factorization f* = *fs*
and *sf*: *square-free f*

and *deg*: *degree f* > *0*
shows *f* = *prod-list fs* ∧ (∀ *fi* ∈ *set fs*. *irreducible*$_d$ *fi*)
proof −
let *?lc* = *lead-coeff f*
define *p* **where** *p* ≡ *suitable-prime-bz f*
obtain *c gs* **where** *berl*: *finite-field-factorization-int p f* = (*c*,*gs*) **by** *force*
let *?degs* = *map degree gs*
note *res* = *res*[*unfolded berlekamp-zassenhaus-factorization-def Let-def*, *folded p-def*,
unfolded berl split, *folded*]
from *suitable-prime-bz*[*OF sf refl*]
have *prime*: *prime p* **and** *cop*: *coprime ?lc p* **and** *sf*: *poly-mod.square-free-m p f*
unfolding *p-def* **by** *auto*
from *prime* **interpret** *poly-mod-prime p* **by** *unfold-locales*
define *n* **where** *n* = *find-exponent p* (*2* ∗ *abs ?lc* ∗ *factor-bound f* (*degree-bound gs*))
note *n* = *find-exponent*[*OF m1*, *of 2* ∗ *abs ?lc* ∗ *factor-bound f* (*degree-bound gs*),
folded n-def]
note *bh* = *berlekamp-and-hensel-separated*[*OF cop sf refl berl n*(*2*)]
have *db*: *degree-bound* (*berlekamp-hensel p n f*) = *degree-bound gs* **unfolding** *bh*
degree-bound-def max-factor-degree-def **by** *simp*
note *res* = *res*[*folded n-def bh*(*1*)]
show *?thesis*
by (*rule zassenhaus-reconstruction-irreducible*$_d$[*OF prime cop sf deg refl - res*], *insert n db*, *auto*)
qed

corollary *berlekamp-zassenhaus-factorization-irreducible*:
assumes *res*: *berlekamp-zassenhaus-factorization f* = *fs*
and *sf*: *square-free f*
and *pr*: *primitive f*
and *deg*: *degree f* > *0*
shows *f* = *prod-list fs* ∧ (∀ *fi* ∈ *set fs*. *irreducible fi*)
using *pr irreducible-primitive-connect*[*OF primitive-prod-list*]
berlekamp-zassenhaus-factorization-irreducible$_d$[*OF res sf deg*] **by** *auto*

end

11.2 A fast coprimality approximation

We adapt the integer polynomial gcd algorithm so that it first tests whether f and g are coprime modulo a few primes. If so, we are immediately done.

theory *Gcd-Finite-Field-Impl*
imports
Suitable-Prime
Code-Abort-Gcd
HOL−Library.Code-Target-Int
begin

definition *coprime-approx-main* :: *int* ⇒ *'i arith-ops-record* ⇒ *int poly* ⇒ *int poly* ⇒ *bool* **where**
coprime-approx-main p ff-ops f g = (*gcd-poly-i ff-ops* (*of-int-poly-i ff-ops* (*poly-mod.Mp p f*))
(*of-int-poly-i ff-ops* (*poly-mod.Mp p g*)) = *one-poly-i ff-ops*)

lemma (**in** *prime-field-gen*) *coprime-approx-main*:
shows *coprime-approx-main p ff-ops f g* ⟹ *coprime-m f g*
proof −
define *F* **where** *F*: (*F* :: *'a mod-ring poly*) = *of-int-poly* (*Mp f*)
define *G* **where** *G*: (*G* :: *'a mod-ring poly*) = *of-int-poly* (*Mp g*) **let** *?f'* = *of-int-poly-i ff-ops* (*Mp f*)
let *?g'* = *of-int-poly-i ff-ops* (*Mp g*)
define *f''* **where** *f''* ≡ *of-int-poly* (*Mp f*) :: *'a mod-ring poly*
define *g''* **where** *g''* ≡ *of-int-poly* (*Mp g*) :: *'a mod-ring poly*
have *rel-f*[*transfer-rule*]: *poly-rel ?f' f''*
by (*rule poly-rel-of-int-poly*[*OF refl*], *simp add*: *f''-def*)
have *rel-f*[*transfer-rule*]: *poly-rel ?g' g''*
by (*rule poly-rel-of-int-poly*[*OF refl*], *simp add*: *g''-def*)
have *id*: (*gcd-poly-i ff-ops* (*of-int-poly-i ff-ops* (*Mp f*)) (*of-int-poly-i ff-ops* (*Mp g*)) = *one-poly-i ff-ops*)
= *coprime f'' g''* (**is** *?P* ⟷ *?Q*)
proof −
have *?P* ⟷ *gcd f'' g''* = *1*
unfolding *separable-i-def* **by** *transfer-prover*
also have ... ⟷ *?Q*
by (*simp add*: *coprime-iff-gcd-eq-1*)
finally show *?thesis* **.**
qed
have *fF*: *MP-Rel* (*Mp f*) *F* **unfolding** *F MP-Rel-def*
by (*simp add*: *Mp-f-representative*)
have *gG*: *MP-Rel* (*Mp g*) *G* **unfolding** *G MP-Rel-def*
by (*simp add*: *Mp-f-representative*)
have *coprime f'' g''* = *coprime F G* **unfolding** *f''-def F g''-def G* **by** *simp*
also have ... = *coprime-m* (*Mp f*) (*Mp g*)
using *coprime-MP-Rel*[*unfolded rel-fun-def*, *rule-format*, *OF fF gG*] **by** *simp*
also have ... = *coprime-m f g* **unfolding** *coprime-m-def dvdm-def* **by** *simp*
finally have *id2*: *coprime f'' g''* = *coprime-m f g* **.**
show *coprime-approx-main p ff-ops f g* ⟹ *coprime-m f g* **unfolding** *coprime-approx-main-def*
id id2 **by** *auto*
qed

context *poly-mod-prime* **begin**

lemmas *coprime-approx-main-uint32* = *prime-field-gen.coprime-approx-main*[*OF*

prime-field.prime-field-finite-field-ops32, *unfolded prime-field-def mod-ring-locale-def*
poly-mod-type-simps, *internalize-sort 'a* :: *prime-card*, *OF type-to-set*, *unfolded*

remove-duplicate-premise, *cancel-type-definition*, *OF non-empty*]

lemmas *coprime-approx-main-uint64* = *prime-field-gen.coprime-approx-main*[*OF*

prime-field.prime-field-finite-field-ops64, *unfolded prime-field-def mod-ring-locale-def poly-mod-type-simps*, *internalize-sort 'a* :: *prime-card*, *OF type-to-set*, *unfolded remove-duplicate-premise*, *cancel-type-definition*, *OF non-empty*]

end

lemma *coprime-mod-imp-coprime*: **assumes**
p: *prime p* **and**
cop-m: *poly-mod.coprime-m p f g* **and**
cop: *coprime* (*lead-coeff f*) *p* ∨ *coprime* (*lead-coeff g*) *p* **and**
cnt: *content f* = *1* ∨ *content g* = *1*
shows *coprime f g*
proof −
interpret *poly-mod-prime p* **by** (*standard*, *rule p*)
from *cop-m*[*unfolded coprime-m-def*] **have** *cop-m*: ⋀ *h*. *h dvdm f* ⟹ *h dvdm g* ⟹ *h dvdm 1* **by** *auto*
show *?thesis*
proof (*rule coprimeI*)
fix *h*
assume *dvd*: *h dvd f h dvd g*
hence *h dvdm f h dvdm g* **unfolding** *dvdm-def dvd-def* **by** *auto*
from *cop-m*[*OF this*] **obtain** *k* **where** *unit*: *Mp* (*h* ∗ *Mp k*) = *1* **unfolding** *dvdm-def* **by** *auto*
from *content-dvd-contentI*[*OF dvd*(*1*)] *content-dvd-contentI*[*OF dvd*(*2*)] *cnt*
have *cnt*: *content h* = *1* **by** *auto*
let *?k* = *Mp k*
from *unit* **have** *h0*: *h* ≠ *0* **by** *auto*
from *unit* **have** *k0*: *?k* ≠ *0* **by** *fastforce*
from *p* **have** *p0*: *p* ≠ *0* **by** *auto*
from *dvd* **have** *lead-coeff h dvd lead-coeff f lead-coeff h dvd lead-coeff g*
by (*metis dvd-def lead-coeff-mult*)+
with *cop* **have** *coph*: *coprime* (*lead-coeff h*) *p*
by (*meson dvd-trans not-coprime-iff-common-factor*)
let *?k* = *Mp k*
from *arg-cong*[*OF unit*, *of degree*] **have** *degm0*: *degree-m* (*h* ∗ *?k*) = *0* **by** *simp*
have *lead-coeff ?k* ∈ {*0* ..< *p*} **unfolding** *Mp-coeff M-def* **using** *m1* **by** *simp*
with *k0* **have** *lk*: *lead-coeff ?k* ≥ *1 lead-coeff ?k* < *p*
by (*auto simp add*: *int-one-le-iff-zero-less order.not-eq-order-implies-strict*)
have *id*: *lead-coeff* (*h* ∗ *?k*) = *lead-coeff h* ∗ *lead-coeff ?k* **unfolding** *lead-coeff-mult* ..
from *coph prime lk* **have** *coprime* (*lead-coeff h* ∗ *lead-coeff ?k*) *p*
by (*simp add*: *ac-simps prime-imp-coprime zdvd-not-zless*)
with *id* **have** *cop-prod*: *coprime* (*lead-coeff* (*h* ∗ *?k*)) *p* **by** *simp*
from *h0 k0* **have** *lc0*: *lead-coeff* (*h* ∗ *?k*) ≠ *0*
unfolding *lead-coeff-mult* **by** *auto*

from *p* **have** *lcp*: *lead-coeff* (*h* ∗ *?k*) *mod p* ≠ *0*
  using *M-1 M-def cop-prod* **by** *auto*
have *deg-eq*: *degree-m* (*h* ∗ *?k*) = *degree* (*h* ∗ *Mp k*)
  by (*rule degree-m-eq*[*OF - m1*], *insert lcp*)
from *this*[*unfolded degm0*] **have** *degree* (*h* ∗ *Mp k*) = *0* **by** *simp*
with *degree-mult-eq*[*OF h0 k0*] **have** *deg0*: *degree h* = *0* **by** *auto*
from *degree0-coeffs*[*OF this*] **obtain** *h0* **where** *h*: *h* = [:*h0*:] **by** *auto*
have *content h* = *abs h0* **unfolding** *content-def h* **by** (*cases h0* = *0*, *auto*)
hence *abs h0* = *1* **using** *cnt* **by** *auto*
hence *h0* ∈ {−*1*,*1*} **by** *auto*
hence *h* = *1* ∨ *h* = −*1* **unfolding** *h* **by** (*auto*)
thus *is-unit h* **by** *auto*
qed
qed

We did not try to optimize the set of chosen primes. They have just been picked randomly from a list of primes.

definition *gcd-primes32* :: *int list* **where**
gcd-primes32 = [*383*, *1409*, *19213*, *22003*, *41999*]

lemma *gcd-primes32*: *p* ∈ *set gcd-primes32* ⟹ *prime p* ∧ *p* ≤ *65535*
proof −
have *list-all* (λ *p*. *prime p* ∧ *p* ≤ *65535*) *gcd-primes32* **by** *eval*
thus *p* ∈ *set gcd-primes32* ⟹ *prime p* ∧ *p* ≤ *65535* **by** (*auto simp*: *list-all-iff*)
qed

definition *gcd-primes64* :: *int list* **where**
gcd-primes64 = [*383*, *21984191*, *50329901*, *80329901*, *219849193*]

lemma *gcd-primes64*: *p* ∈ *set gcd-primes64* ⟹ *prime p* ∧ *p* ≤ *4294967295*
proof −
have *list-all* (λ *p*. *prime p* ∧ *p* ≤ *4294967295*) *gcd-primes64* **by** *eval*
thus *p* ∈ *set gcd-primes64* ⟹ *prime p* ∧ *p* ≤ *4294967295* **by** (*auto simp*: *list-all-iff*)
qed

definition *coprime-heuristic* :: *int poly* ⇒ *int poly* ⇒ *bool* **where**
coprime-heuristic f g = (*let lcf* = *lead-coeff f*; *lcg* = *lead-coeff g in*
find (λ *p*. (*coprime lcf p* ∨ *coprime lcg p*) ∧ *coprime-approx-main p* (*finite-field-ops64* (*uint64-of-int p*)) *f g*)
gcd-primes64 ≠ *None*)

lemma *coprime-heuristic*: **assumes** *coprime-heuristic f g*
and *content f* = *1* ∨ *content g* = *1*
shows *coprime f g*
proof (*cases find* (λ*p*. (*coprime* (*lead-coeff f*) *p* ∨ *coprime* (*lead-coeff g*) *p*) ∧
coprime-approx-main p (*finite-field-ops64* (*uint64-of-int p*)) *f g*)
gcd-primes64)
case (*Some p*)

from *find-Some-D*[*OF Some*] *gcd-primes64* **have** *p*: *prime p* **and** *small*: $p \leq 4294967295$
and *cop*: *coprime* (*lead-coeff f*) *p* ∨ *coprime* (*lead-coeff g*) *p*
and *copp*: *coprime-approx-main p* (*finite-field-ops64* (*uint64-of-int p*)) *f g* **by** *auto*
interpret *poly-mod-prime p* **using** *p* **by** *unfold-locales*
from *coprime-approx-main-uint64*[*OF small copp*] **have** *poly-mod.coprime-m p f g* **by** *auto*
from *coprime-mod-imp-coprime*[*OF p this cop assms*(*2*)] **show** *coprime f g* **.**
qed (*insert assms*(*1*)[*unfolded coprime-heuristic-def*], *auto simp*: *Let-def*)

definition *gcd-int-poly* :: *int poly* ⇒ *int poly* ⇒ *int poly* **where**
gcd-int-poly f g =
(*if f* = *0 then normalize g*
else if g = *0 then normalize f*
else let
cf = *Polynomial.content f*;
cg = *Polynomial.content g*;
ct = *gcd cf cg*;
ff = *map-poly* (λ *x*. *x div cf*) *f*;
gg = *map-poly* (λ *x*. *x div cg*) *g*
in if coprime-heuristic ff gg then [:*ct*:] *else smult ct* (*gcd-poly-code-aux ff gg*))

lemma *gcd-int-poly-code*[*code-unfold*]: *gcd* = *gcd-int-poly*
proof (*intro ext*)
fix *f g* :: *int poly*
let *?ff* = *primitive-part f*
let *?gg* = *primitive-part g*
note *d* = *gcd-int-poly-def gcd-poly-code gcd-poly-code-def*
show *gcd f g* = *gcd-int-poly f g*
proof (*cases f* = *0* ∨ *g* = *0* ∨ ¬ *coprime-heuristic ?ff ?gg*)
case *True*
thus *?thesis* **unfolding** *d* **by** (*auto simp*: *Let-def primitive-part-def*)
next
case *False*
hence *cop*: *coprime-heuristic ?ff ?gg* **by** *simp*
from *False* **have** *f* ≠ *0* **by** *auto*
from *content-primitive-part*[*OF this*] *coprime-heuristic*[*OF cop*]
have *id*: *gcd ?ff ?gg* = *1* **by** *auto*
show *?thesis* **unfolding** *gcd-poly-decompose*[*of f g*] **unfolding** *gcd-int-poly-def Let-def id*
using *False* **by** (*auto simp*: *primitive-part-def*)
qed
qed

end

theory *Square-Free-Factorization-Int*

imports
Square-Free-Int-To-Square-Free-GFp
Suitable-Prime
Code-Abort-Gcd
Gcd-Finite-Field-Impl
begin

definition *yun-wrel* :: *int poly* ⇒ *rat* ⇒ *rat poly* ⇒ *bool* **where**
yun-wrel F c f = (*map-poly rat-of-int F* = *smult c f*)

definition *yun-rel* :: *int poly* ⇒ *rat* ⇒ *rat poly* ⇒ *bool* **where**
yun-rel F c f = (*yun-wrel F c f*
∧ *content F* = *1* ∧ *lead-coeff F* > *0* ∧ *monic f*)

definition *yun-erel* :: *int poly* ⇒ *rat poly* ⇒ *bool* **where**
yun-erel F f = (∃ *c*. *yun-rel F c f*)

lemma *yun-wrelD*: **assumes** *yun-wrel F c f*
shows *map-poly rat-of-int F* = *smult c f*
using *assms* **unfolding** *yun-wrel-def* **by** *auto*

lemma *yun-relD*: **assumes** *yun-rel F c f*
shows *yun-wrel F c f map-poly rat-of-int F* = *smult c f*
degree F = *degree f F* ≠ *0 lead-coeff F* > *0 monic f*
f = *1* ⟷ *F* = *1 content F* = *1*
proof −
note ∗ = *assms*[*unfolded yun-rel-def yun-wrel-def*, *simplified*]
then have *degree* (*map-poly rat-of-int F*) = *degree f* **by** *auto*
then show *deg*: *degree F* = *degree f* **by** *simp*
show *F* ≠ *0 lead-coeff F* > *0 monic f content F* = *1*
map-poly rat-of-int F = *smult c f*
yun-wrel F c f **using** ∗ **by** (*auto simp*: *yun-wrel-def*)
{
assume *f* = *1*
with *deg* **have** *degree F* = *0* **by** *auto*
from *degree0-coeffs*[*OF this*] **obtain** *c* **where** *F*: *F* = [:*c*:] **and** *c*: *c* = *lead-coeff F* **by** *auto*
from *c* ∗ **have** *c0*: *c* > *0* **by** *auto*
hence *cF*: *content F* = *c* **unfolding** *F content-def* **by** *auto*
with ∗ **have** *c* = *1* **by** *auto*
with *F* **have** *F* = *1* **by** *simp*
}
moreover
{
assume *F* = *1*
with *deg* **have** *degree f* = *0* **by** *auto*
with ‹*monic f*› **have** *f* = *1*
using *monic-degree-0* **by** *blast*
}

ultimately show $(f = 1) \longleftrightarrow (F = 1)$ **by** *auto*
qed

lemma *yun-erel-1-eq*: **assumes** *yun-erel F f*
shows $(F = 1) \longleftrightarrow (f = 1)$
proof −
from *assms*[*unfolded yun-erel-def*] **obtain** *c* **where** *yun-rel F c f* **by** *auto*
from *yun-relD*[*OF this*] **show** *?thesis* **by** *simp*
qed

lemma *yun-rel-1*[*simp*]: *yun-rel 1 1 1*
by (*auto simp*: *yun-rel-def yun-wrel-def content-def*)

lemma *yun-erel-1*[*simp*]: *yun-erel 1 1* **unfolding** *yun-erel-def* **using** *yun-rel-1* **by** *blast*

lemma *yun-rel-mult*: *yun-rel F c f* $\Longrightarrow$ *yun-rel G d g* $\Longrightarrow$ *yun-rel* $(F * G)$ $(c * d)$ $(f * g)$
unfolding *yun-rel-def yun-wrel-def content-mult lead-coeff-mult*
by (*auto simp*: *monic-mult hom-distribs*)

lemma *yun-erel-mult*: *yun-erel F f* $\Longrightarrow$ *yun-erel G g* $\Longrightarrow$ *yun-erel* $(F * G)$ $(f * g)$

unfolding *yun-erel-def* **using** *yun-rel-mult*[*of F - f G - g*] **by** *blast*

lemma *yun-rel-pow*: **assumes** *yun-rel F c f*
shows *yun-rel* $(F\hat{\ }n)$ $(c\hat{\ }n)$ $(f\hat{\ }n)$
by (*induct n*, *insert assms yun-rel-mult*, *auto*)

lemma *yun-erel-pow*: *yun-erel F f* $\Longrightarrow$ *yun-erel* $(F\hat{\ }n)$ $(f\hat{\ }n)$
using *yun-rel-pow* **unfolding** *yun-erel-def* **by** *blast*

lemma *yun-wrel-pderiv*: **assumes** *yun-wrel F c f*
shows *yun-wrel* (*pderiv F*) *c* (*pderiv f*)
by (*unfold yun-wrel-def*, *simp add*: *yun-wrelD*[*OF assms*] *pderiv-smult hom-distribs*)

lemma *yun-wrel-minus*: **assumes** *yun-wrel F c f yun-wrel G c g*
shows *yun-wrel* $(F - G)$ *c* $(f - g)$
using *assms* **unfolding** *yun-wrel-def* **by** (*auto simp*: *smult-diff-right hom-distribs*)

lemma *yun-wrel-div*: **assumes** *f*: *yun-wrel F c f* **and** *g*: *yun-wrel G d g*
and *dvd*: *G dvd F g dvd f*
and *G0*: $G \neq 0$
shows *yun-wrel* (*F div G*) $(c \ / \ d)$ (*f div g*)
proof −
let *?r* = *rat-of-int*
let *?rp* = *map-poly ?r*
from *dvd* **obtain** *H h* **where** *fgh*: $F = G * H$ $f = g * h$ **unfolding** *dvd-def* **by** *auto*

from *G0 yun-wrelD*[*OF g*] **have** *g0*: $g \neq 0$ **and** *d0*: $d \neq 0$ **by** *auto*
from *arg-cong*[*OF fgh*(*1*), *of* λ *x*. *x div G*] **have** *H*: *H* = *F div G* **using** *G0* **by** *simp*
from *arg-cong*[*OF fgh*(*1*), *of ?rp*] **have** *?rp F* = *?rp G* ∗ *?rp H* **by** (*auto simp*: *hom-distribs*)
from *arg-cong*[*OF this*, *of* λ *x*. *x div ?rp G*] *G0* **have** *id*: *?rp H* = *?rp F div ?rp G* **by** *auto*
have *?rp* (*F div G*) = *?rp F div ?rp G* **unfolding** *H*[*symmetric*] *id* **by** *simp*
also have ... = *smult c f div smult d g* **using** *f g* **unfolding** *yun-wrel-def* **by** *auto*
also have ... = *smult* (*c* / *d*) (*f div g*) **unfolding** *div-smult-right*[*OF d0*] *div-smult-left*
by (*simp add*: *field-simps*)
finally show *?thesis* **unfolding** *yun-wrel-def* **by** *simp*
qed

lemma *yun-rel-div*: **assumes** *f*: *yun-rel F c f* **and** *g*: *yun-rel G d g*
and *dvd*: *G dvd F g dvd f*
shows *yun-rel* (*F div G*) (*c* / *d*) (*f div g*)
proof −
note *ff* = *yun-relD*[*OF f*]
note *gg* = *yun-relD*[*OF g*]
show *?thesis* **unfolding** *yun-rel-def*
proof (*intro conjI*)
from *yun-wrel-div*[*OF ff*(*1*) *gg*(*1*) *dvd gg*(*4*)]
show *yun-wrel* (*F div G*) (*c* / *d*) (*f div g*) **by** *auto*
from *dvd* **have** *fg*: *f* = *g* ∗ (*f div g*) **by** *auto*
from *arg-cong*[*OF fg*, *of monic*] *ff*(*6*) *gg*(*6*)
show *monic* (*f div g*) **using** *monic-factor* **by** *blast*
from *dvd* **have** *FG*: *F* = *G* ∗ (*F div G*) **by** *auto*
from *arg-cong*[*OF FG*, *of content*, *unfolded content-mult*] *ff*(*8*) *gg*(*8*)
show *content* (*F div G*) = *1* **by** *simp*
from *arg-cong*[*OF FG*, *of lead-coeff*, *unfolded lead-coeff-mult*] *ff*(*5*) *gg*(*5*)
show *lead-coeff* (*F div G*) > *0* **by** (*simp add*: *zero-less-mult-iff*)
qed
qed

lemma *yun-wrel-gcd*: **assumes** *yun-wrel F c′ f yun-wrel G c g* **and** *c*: $c' \neq 0$ $c \neq 0$
and *d*: *d* = *rat-of-int* (*lead-coeff* (*gcd F G*)) $d \neq 0$
shows *yun-wrel* (*gcd F G*) *d* (*gcd f g*)
proof −
let *?r* = *rat-of-int*
let *?rp* = *map-poly ?r*
have *smult d* (*gcd f g*) = *smult d* (*gcd* (*smult c′ f*) (*smult c g*))
by (*simp add*: *c gcd-smult-left gcd-smult-right*)
also have ... = *smult d* (*gcd* (*?rp F*) (*?rp G*)) **using** *assms*(*1*−*2*)[*unfolded*

yun-wrel-def] **by** *simp*
  also have ... = *smult* (*d* ∗ *inverse* *d*) (*?rp* (*gcd* *F* *G*))
    unfolding *gcd-rat-to-gcd-int* *d* **by** *simp*
  also have *d* ∗ *inverse* *d* = *1* **using** *d* **by** *auto*
  finally show *?thesis* **unfolding** *yun-wrel-def* **by** *simp*
qed

lemma *yun-rel-gcd*: **assumes** *f*: *yun-rel* *F* *c* *f* **and** *g*: *yun-wrel* *G* *c′* *g* **and** *c′*: *c′* ≠ *0*
  and *d*: *d* = *rat-of-int* (*lead-coeff* (*gcd* *F* *G*))
shows *yun-rel* (*gcd* *F* *G*) *d* (*gcd* *f* *g*)
  unfolding *yun-rel-def*
proof (*intro* *conjI*)
  note *ff* = *yun-relD*[*OF* *f*]
  from *ff* **have** *c0*: *c* ≠ *0* **by** *auto*
  from *ff* *d* **have** *d0*: *d* ≠ *0* **by** *auto*
  from *yun-wrel-gcd*[*OF* *ff*(*1*) *g* *c0* *c′* *d* *d0*]
  show *yun-wrel* (*gcd* *F* *G*) *d* (*gcd* *f* *g*) **by** *auto*
  from *ff* **have** *gcd* *f* *g* ≠ *0* **by** *auto*
  thus *monic* (*gcd* *f* *g*) **by** (*simp* *add*: *poly-gcd-monic*)
  obtain *H* **where** *H*: *gcd* *F* *G* = *H* **by** *auto*
  obtain *lc* **where** *lc*: *coeff* *H* (*degree* *H*) = *lc* **by** *auto*
  from *ff* **have** *gcd* *F* *G* ≠ *0* **by** *auto*
  hence *H* ≠ *0* *lc* ≠ *0* **unfolding** *H*[*symmetric*] *lc*[*symmetric*] **by** *auto*
  thus *0* < *lead-coeff* (*gcd* *F* *G*) **unfolding**
    arg-cong[*OF* *normalize-gcd*[*of* *F* *G*], *of* *lead-coeff*, *symmetric*]
    unfolding *normalize-poly-eq-map-poly* *H*
    by (*auto*, *subst* *Polynomial.coeff-map-poly*, *auto*,
    subst *Polynomial.degree-map-poly*, *auto* *simp*: *sgn-if*)
  have *H* *dvd* *F* **unfolding** *H*[*symmetric*] **by** *auto*
  then obtain *K* **where** *F*: *F* = *H* ∗ *K* **unfolding** *dvd-def* **by** *auto*
  from *arg-cong*[*OF* *this*, *of* *content*, *unfolded* *content-mult* *ff*(*8*)]
    content-ge-0-int[*of* *H*] **have** *content* *H* = *1*
    by (*auto* *simp* *add*: *zmult-eq-1-iff*)
  thus *content* (*gcd* *F* *G*) = *1* **unfolding** *H* **.**
qed

lemma *yun-factorization-main-int*: **assumes** *f*: *f* = *p* *div* *gcd* *p* (*pderiv* *p*)
    and *g* = *pderiv* *p* *div* *gcd* *p* (*pderiv* *p*) *monic* *p*
    and *yun-gcd.yun-factorization-main* *gcd* *f* *g* *i* *hs* = *res*
    and *yun-gcd.yun-factorization-main* *gcd* *F* *G* *i* *Hs* = *Res*
    and *yun-rel* *F* *c* *f* *yun-wrel* *G* *c* *g* *list-all2* (*rel-prod* *yun-erel* (=)) *Hs* *hs*
  shows *list-all2* (*rel-prod* *yun-erel* (=)) *Res* *res*
proof −
  let *?P* = λ *f* *g*. ∀ *i* *hs* *res* *F* *G* *Hs* *Res* *c*.
    yun-gcd.yun-factorization-main *gcd* *f* *g* *i* *hs* = *res*
    ⟶ *yun-gcd.yun-factorization-main* *gcd* *F* *G* *i* *Hs* = *Res*

⟶ *yun-rel F c f* ⟶ *yun-wrel G c g* ⟶ *list-all2* (*rel-prod yun-erel* (=)) *Hs hs*
⟶ *list-all2* (*rel-prod yun-erel* (=)) *Res res*
note *simps* = *yun-gcd.yun-factorization-main.simps*
note *rel* = *yun-relD*
let *?rel* = λ *F f*. *map-poly rat-of-int F* = *smult* (*rat-of-int* (*lead-coeff F*)) *f*
show *?thesis*
proof (*induct rule*: *yun-factorization-induct*[*of ?P*, *rule-format*, *OF* - - *assms*])
case (*1 f g i hs res F G Hs Res c*)
from *rel*[*OF 1*(*4*)] *1*(*1*) **have** *f* = *1 F* = *1* **by** *auto*
from *1*(*2*−*3*)[*unfolded simps*[*of* - *1*] *this*] **have** *res* = *hs Res* = *Hs* **by** *auto*
with *1*(*6*) **show** *?case* **by** *simp*
next
case (*2 f g i hs res F G Hs Res c*)
define *d* **where** *d* = *g* − *pderiv f*
define *a* **where** *a* = *gcd f d*
define *D* **where** *D* = *G* − *pderiv F*
define *A* **where** *A* = *gcd F D*
note *f* = *2*(*5*)
note *g* = *2*(*6*)
note *hs* = *2*(*7*)
note *f1* = *2*(*1*)
from *f1 rel*[*OF f*] **have** ∗: (*f* = *1*) = *False* (*F* = *1*) = *False* **and** *c*: *c* ≠ *0* **by** *auto*
note *res* = *2*(*3*)[*unfolded simps*[*of* - *f*] ∗ *if-False Let-def*, *folded d-def a-def*]
note *Res* = *2*(*4*)[*unfolded simps*[*of* - *F*] ∗ *if-False Let-def*, *folded D-def A-def*]
note *IH* = *2*(*2*)[*folded d-def a-def*, *OF res Res*]
obtain *c′* **where** *c′*: *c′* = *rat-of-int* (*lead-coeff* (*gcd F D*)) **by** *auto*
show *?case*
proof (*rule IH*)
from *yun-wrel-minus*[*OF g yun-wrel-pderiv*[*OF rel*(*1*)[*OF f*]]]
have *d*: *yun-wrel D c d* **unfolding** *D-def d-def* **.**
have *a*: *yun-rel A c′ a* **unfolding** *A-def a-def*
by (*rule yun-rel-gcd*[*OF f d c c′*])
hence *yun-erel A a* **unfolding** *yun-erel-def* **by** *auto*
thus *list-all2* (*rel-prod yun-erel* (=)) ((*A*, *i*) # *Hs*) ((*a*, *i*) # *hs*)
using *hs* **by** *auto*
have *A0*: *A* ≠ *0* **by** (*rule rel*(*4*)[*OF a*])
have *A dvd D a dvd d* **unfolding** *A-def a-def* **by** *auto*
from *yun-wrel-div*[*OF d rel*(*1*)[*OF a*] *this A0*]
show *yun-wrel* (*D div A*) (*c* / *c′*) (*d div a*) **.**
have *A dvd F a dvd f* **unfolding** *A-def a-def* **by** *auto*
from *yun-rel-div*[*OF f a this*]
show *yun-rel* (*F div A*) (*c* / *c′*) (*f div a*) **.**
qed
qed
qed

lemma *yun-monic-factorization-int-yun-rel*: **assumes**
res: *yun-gcd.yun-monic-factorization gcd f* = *res*

and *Res*: *yun-gcd.yun-monic-factorization gcd F = Res*
and *f*: *yun-rel F c f*
shows *list-all2* (*rel-prod yun-erel* (=)) *Res res*
proof −
note *ff* = *yun-relD*[*OF f*]
let *?g* = *gcd f* (*pderiv f*)
let *?yf* = *yun-gcd.yun-factorization-main gcd* (*f div ?g*) (*pderiv f div ?g*) *0* []
let *?G* = *gcd F* (*pderiv F*)
let *?yF* = *yun-gcd.yun-factorization-main gcd* (*F div ?G*) (*pderiv F div ?G*) *0* []

obtain *r R* **where** *r*: *?yf* = *r* **and** *R*: *?yF* = *R* **by** *blast*
from *res*[*unfolded yun-gcd.yun-monic-factorization-def Let-def r*]
have *res*: *res* = [(*a*, *i*)←*r* . *a* ≠ *1*] **by** *simp*
from *Res*[*unfolded yun-gcd.yun-monic-factorization-def Let-def R*]
have *Res*: *Res* = [(*A*, *i*)←*R* . *A* ≠ *1*] **by** *simp*
from *yun-wrel-pderiv*[*OF ff*(*1*)] **have** *f'*: *yun-wrel* (*pderiv F*) *c* (*pderiv f*) **.**
from *ff* **have** *c*: *c* ≠ *0* **by** *auto*
from *yun-rel-gcd*[*OF f f' c refl*] **obtain** *d* **where** *g*: *yun-rel ?G d ?g* **..**
from *yun-rel-div*[*OF f g*] **have** *1*: *yun-rel* (*F div ?G*) (*c* / *d*) (*f div ?g*) **by** *auto*
from *yun-wrel-div*[*OF f' yun-relD*(*1*)[*OF g*] - - *yun-relD*(*4*)[*OF g*]]
have *2*: *yun-wrel* (*pderiv F div ?G*) (*c* / *d*) (*pderiv f div ?g*) **by** *auto*
from *yun-factorization-main-int*[*OF refl refl ff*(*6*) *r R 1 2*]
have *list-all2* (*rel-prod yun-erel* (=)) *R r* **by** *simp*
thus *?thesis* **unfolding** *res Res*
by (*induct R r rule*: *list-all2-induct*, *auto dest*: *yun-erel-1-eq*)
qed

lemma *yun-rel-same-right*: **assumes** *yun-rel f c G yun-rel g d G*
shows *f* = *g*
proof −
note *f* = *yun-relD*[*OF assms*(*1*)]
note *g* = *yun-relD*[*OF assms*(*2*)]
let *?r* = *rat-of-int*
let *?rp* = *map-poly ?r*
from *g* **have** *d*: *d* ≠ *0* **by** *auto*
obtain *a b* **where** *quot*: *quotient-of* (*c* / *d*) = (*a*,*b*) **by** *force*
from *quotient-of-nonzero*[*of c*/*d*, *unfolded quot*] **have** *b*: *b* ≠ *0* **by** *simp*
note *f*(*2*)
also have *smult c G* = *smult* (*c* / *d*) (*smult d G*) **using** *d* **by** (*auto simp*: *field-simps*)
also have *smult d G* = *?rp g* **using** *g*(*2*) **by** *simp*
also have *cd*: *c* / *d* = (*?r a* / *?r b*) **using** *quotient-of-div*[*OF quot*] **.**
finally have *fg*: *?rp f* = *smult* (*?r a* / *?r b*) (*?rp g*) **by** *simp*
from *f* **have** *c* ≠ *0* **by** *auto*
with *cd d* **have** *a*: *a* ≠ *0* **by** *auto*
from *arg-cong*[*OF fg*, *of* λ *x*. *smult* (*?r b*) *x*]
have *smult* (*?r b*) (*?rp f*) = *smult* (*?r a*) (*?rp g*) **using** *b* **by** *auto*
hence *?rp* (*smult b f*) = *?rp* (*smult a g*) **by** (*auto simp*: *hom-distribs*)
then have *fg*: [:*b*:] ∗ *f* = [:*a*:] ∗ *g* **by** *auto*

from *arg-cong*[*OF this*, *of content*, *unfolded content-mult* $f(8)$ $g(8)$]
have *content* [: *b* :] = *content* [: *a* :] **by** *simp*
hence *abs*: *abs a* = *abs b* **unfolding** *content-def* **using** *b a* **by** *auto*
from *arg-cong*[*OF fg*, *of* λ *x*. *lead-coeff x* > *0*, *unfolded lead-coeff-mult*] $f(5)$ $g(5)$ *a b*
have (*a* > *0*) = (*b* > *0*) **by** (*simp add*: *zero-less-mult-iff*)
with *a b abs* **have** *a* = *b* **by** *auto*
with *arg-cong*[*OF fg*, *of* λ *x*. *x div* [:*b*:]] *b* **show** *?thesis*
by (*metis nonzero-mult-div-cancel-left pCons-eq-0-iff*)
qed

definition *square-free-factorization-int-main* :: *int poly* ⇒ (*int poly* × *nat*) *list*
where
square-free-factorization-int-main f = (*case square-free-heuristic f of None* ⇒
yun-gcd.yun-monic-factorization gcd f | *Some p* ⇒ [(*f*,*0*)])

lemma *square-free-factorization-int-main*: **assumes** *res*: *square-free-factorization-int-main f* = *fs*
and *ct*: *content f* = *1* **and** *lc*: *lead-coeff f* > *0*
and *deg*: *degree f* ≠ *0*
shows *square-free-factorization f* (*1*,*fs*) ∧ (∀ *fi i*. (*fi*, *i*) ∈ *set fs* ⟶ *content fi* = *1* ∧ *lead-coeff fi* > *0*) ∧
distinct (*map snd fs*)
proof (*cases square-free-heuristic f*)
case *None*
from *lc* **have** *f0*: *f* ≠ *0* **by** *auto*
from *res None* **have** *fs*: *yun-gcd.yun-monic-factorization gcd f* = *fs*
unfolding *square-free-factorization-int-main-def* **by** *auto*
let *?r* = *rat-of-int*
let *?rp* = *map-poly ?r*
define *G* **where** *G* = *smult* (*inverse* (*lead-coeff* (*?rp f*))) (*?rp f*)
have *?rp f* ≠ *0* **using** *f0* **by** *auto*
hence *mon*: *monic G* **unfolding** *G-def coeff-smult* **by** *simp*
obtain *Fs* **where** *Fs*: *yun-gcd.yun-monic-factorization gcd G* = *Fs* **by** *blast*
from *lc* **have** *lg*: *lead-coeff* (*?rp f*) ≠ *0* **by** *auto*
let *?c* = *lead-coeff* (*?rp f*)
define *c* **where** *c* = *?c*
have *rp*: *?rp f* = *smult c G* **unfolding** *G-def c-def* **by** (*simp add*: *field-simps*)
have *in-rel*: *yun-rel f c G* **unfolding** *yun-rel-def yun-wrel-def*
using *rp mon lc ct* **by** *auto*
from *yun-monic-factorization-int-yun-rel*[*OF Fs fs in-rel*]
have *out-rel*: *list-all2* (*rel-prod yun-erel* (=)) *fs Fs* **by** *auto*
from *yun-monic-factorization*[*OF Fs mon*]
have *square-free-factorization G* (*1*, *Fs*) **and** *dist*: *distinct* (*map snd Fs*) **by** *auto*
note *sff* = *square-free-factorizationD*[*OF this*(*1*)]
from *out-rel* **have** *map snd fs* = *map snd Fs* **by** (*induct fs Fs rule*: *list-all2-induct*, *auto*)

with *dist* **have** *dist′*: *distinct* (*map snd fs*) **by** *auto*
have *main*: *square-free-factorization f* (*1*, *fs*) ∧ (∀ *fi i*. (*fi*, *i*) ∈ *set fs* ⟶ *content fi* = *1* ∧ *lead-coeff fi* > *0*)
unfolding *square-free-factorization-def split*
proof (*intro conjI allI impI*)
from *ct* **have** *f* ≠ *0* **by** *auto*
thus *f* = *0* ⟹ *1* = *0* *f* = *0* ⟹ *fs* = [] **by** *auto*
from *dist′* **show** *distinct fs* **by** (*simp add*: *distinct-map*)
{
fix *a i*
assume *a*: (*a*,*i*) ∈ *set fs*
with *out-rel* **obtain** *bj* **where** *bj* ∈ *set Fs* **and** *rel-prod yun-erel* (=) (*a*,*i*) *bj*
unfolding *list-all2-conv-all-nth set-conv-nth* **by** *fastforce*
then obtain *b* **where** *b*: (*b*,*i*) ∈ *set Fs* **and** *ab*: *yun-erel a b* **by** (*cases bj*, *auto simp*: *rel-prod.simps*)
from *sff*(*2*)[*OF b*] **have** *b′*: *square-free b degree b* ≠ *0* **by** *auto*
from *ab* **obtain** *c* **where** *rel*: *yun-rel a c b* **unfolding** *yun-erel-def* **by** *auto*
note *aa* = *yun-relD*[*OF this*]
from *aa* **have** *c0*: *c* ≠ *0* **by** *auto*
from *b′ aa*(*3*) **show** *degree a* > *0* **by** *simp*
from *square-free-smult*[*OF c0 b′*(*1*), *folded aa*(*2*)]
show *square-free a* **unfolding** *square-free-def* **by** (*force simp*: *dvd-def hom-distribs*)
show *cnt*: *content a* = *1* **and** *lc*: *lead-coeff a* > *0* **using** *aa* **by** *auto*
fix *A I*
assume *A*: (*A*,*I*) ∈ *set fs* **and** *diff*: (*a*,*i*) ≠ (*A*,*I*)
from *a*[*unfolded set-conv-nth*] **obtain** *k* **where** *k*: *fs* ! *k* = (*a*,*i*) *k* < *length fs* **by** *auto*
from *A*[*unfolded set-conv-nth*] **obtain** *K* **where** *K*: *fs* ! *K* = (*A*,*I*) *K* < *length fs* **by** *auto*
from *diff k K* **have** *kK*: *k* ≠ *K* **by** *auto*
from *dist′*[*unfolded distinct-conv-nth length-map*, *rule-format*, *OF k*(*2*) *K*(*2*) *kK*]
have *iI*: *i* ≠ *I* **using** *k K* **by** *simp*
from *A out-rel* **obtain** *Bj* **where** *Bj* ∈ *set Fs* **and** *rel-prod yun-erel* (=) (*A*,*I*) *Bj*
unfolding *list-all2-conv-all-nth set-conv-nth* **by** *fastforce*
then obtain *B* **where** *B*: (*B*,*I*) ∈ *set Fs* **and** *AB*: *yun-erel A B* **by** (*cases Bj*, *auto simp*: *rel-prod.simps*)
then obtain *C* **where** *Rel*: *yun-rel A C B* **unfolding** *yun-erel-def* **by** *auto*
note *AA* = *yun-relD*[*OF this*]
from *iI* **have** (*b*,*i*) ≠ (*B*,*I*) **by** *auto*
from *sff*(*3*)[*OF b B this*] **have** *cop*: *coprime b B* **by** *simp*
from *AA* **have** *C*: *C* ≠ *0* **by** *auto*
from *yun-rel-gcd*[*OF rel AA*(*1*) *C refl*] **obtain** *c* **where** *yun-rel* (*gcd a A*) *c* (*gcd b B*) **by** *auto*
note *rel* = *yun-relD*[*OF this*]
from *rel*(*2*) *cop* **have** *?rp* (*gcd a A*) = [: *c* :] **by** *simp*
from *arg-cong*[*OF this*, *of degree*] **have** *degree* (*gcd a A*) = *0* **by** *simp*
from *degree0-coeffs*[*OF this*] **obtain** *c* **where** *gcd*: *gcd a A* = [: *c* :] **by** *auto*

from *rel*(*8*) *rel*(*5*) **show** *Rings.coprime a A*
by (*auto intro*!: *gcd-eq-1-imp-coprime simp add*: *gcd*)
}
let *?prod* = λ *fs*. ($\prod$(*a*, *i*)$\in$*set fs*. *a* ^ *Suc i*)
let *?pr* = λ *fs*. ($\prod$(*a*, *i*)$\leftarrow$*fs*. *a* ^ *Suc i*)
define *pr* **where** *pr* = *?prod fs*
from ‹*distinct fs*› **have** *pfs*: *?prod fs* = *?pr fs* **by** (*rule prod.distinct-set-conv-list*)
from ‹*distinct Fs*› **have** *pFs*: *?prod Fs* = *?pr Fs* **by** (*rule prod.distinct-set-conv-list*)
from *out-rel* **have** *yun-erel* (*?prod fs*) (*?prod Fs*) **unfolding** *pfs pFs*
proof (*induct fs Fs rule*: *list-all2-induct*)
case (*Cons ai fs Ai Fs*)
obtain *a i* **where** *ai*: *ai* = (*a*,*i*) **by** *force*
from *Cons*(*1*) *ai* **obtain** *A* **where** *Ai*: *Ai* = (*A*,*i*)
and *rel*: *yun-erel a A* **by** (*cases Ai*, *auto simp*: *rel-prod.simps*)
show *?case* **unfolding** *ai Ai* **using** *yun-erel-mult*[*OF yun-erel-pow*[*OF rel*, *of Suc i*] *Cons*(*3*)]
by *auto*
qed *simp*
also have *?prod Fs* = *G* **using** *sff*(*1*) **by** *simp*
finally obtain *d* **where** *rel*: *yun-rel pr d G* **unfolding** *yun-erel-def pr-def* **by** *auto*
with *in-rel* **have** *f* = *pr* **by** (*rule yun-rel-same-right*)
thus *f* = *smult 1* (*?prod fs*) **unfolding** *pr-def* **by** *simp*
qed
from *main dist'* **show** *?thesis* **by** *auto*
next
case (*Some p*)
from *res*[*unfolded square-free-factorization-int-main-def Some*] **have** *fs*: *fs* = [(*f*,*0*)] **by** *auto*
from *lc* **have** *f0*: *f* $\neq$ *0* **by** *auto*
from *square-free-heuristic*[*OF Some*] *poly-mod-prime.separable-impl*(*1*)[*of p f*] *square-free-mod-imp-square-free*[*of p f*] *deg*
show *?thesis* **unfolding** *fs*
by (*auto simp*: *ct lc square-free-factorization-def f0 poly-mod-prime-def*)
qed

definition *square-free-factorization-int'* :: *int poly* $\Rightarrow$ *int* $\times$ (*int poly* $\times$ *nat*)*list* **where**
square-free-factorization-int' f = (*if degree f* = *0*
then (*lead-coeff f*,[]) *else* (*let* — content factorization
c = *content f*;
d = (*sgn* (*lead-coeff f*) $*$ *c*);
g = *sdiv-poly f d*
— and *square-free* factorization
in (*d*, *square-free-factorization-int-main g*)))

lemma *square-free-factorization-int'*: **assumes** *res*: *square-free-factorization-int' f* = (*d*, *fs*)

shows *square-free-factorization f* (*d*,*fs*)
(*fi*, *i*) ∈ *set fs* ⟹ *content fi* = *1* ∧ *lead-coeff fi* > *0*
distinct (*map snd fs*)
proof −
note *res* = *res*[*unfolded square-free-factorization-int′-def Let-def*]
have *square-free-factorization f* (*d*,*fs*)
∧ ((*fi*, *i*) ∈ *set fs* ⟶ *content fi* = *1* ∧ *lead-coeff fi* > *0*)
∧ *distinct* (*map snd fs*)
proof (*cases degree f* = *0*)
case *True*
from *degree0-coeffs*[*OF True*] **obtain** *c* **where** *f*: *f* = [: *c* :] **by** *auto*
thus *?thesis* **using** *res* **by** (*simp add*: *square-free-factorization-def*)
next
case *False*
let *?s* = *sgn* (*lead-coeff f*)
have *s*: *?s* ∈ {−*1*,*1*} **using** *False* **unfolding** *sgn-if* **by** *auto*
define *g* **where** *g* = *smult ?s f*
let *?d* = *?s* ∗ *content f*
have *content g* = *content* ([:*?s*:] ∗ *f*) **unfolding** *g-def* **by** *simp*
also have ... = *content* [:*?s*:] ∗ *content f* **unfolding** *content-mult* **by** *simp*
also have *content* [:*?s*:] = *1* **using** *s* **by** (*auto simp*: *content-def*)
finally have *cg*: *content g* = *content f* **by** *simp*
from *False res*
have *d*: *d* = *?d* **and** *fs*: *fs* = *square-free-factorization-int-main* (*sdiv-poly f ?d*) **by** *auto*
let *?g* = *primitive-part g*
define *ng* **where** *ng* = *primitive-part g*
note *fs*
also have *sdiv-poly f ?d* = *sdiv-poly g* (*content g*) **unfolding** *cg* **unfolding** *g-def*
by (*rule poly-eqI*, *unfold coeff-sdiv-poly coeff-smult*, *insert s*, *auto simp*: *div-minus-right*)
finally have *fs*: *square-free-factorization-int-main ng* = *fs*
unfolding *primitive-part-alt-def ng-def* **by** *simp*
have *lead-coeff f* ≠ *0* **using** *False* **by** *auto*
hence *lg*: *lead-coeff g* > *0* **unfolding** *g-def lead-coeff-smult*
by (*meson linorder-neqE-linordered-idom sgn-greater sgn-less zero-less-mult-iff*)
hence *g0*: *g* ≠ *0* **by** *auto*
from *g0* **have** *content g* ≠ *0* **by** *simp*
from *arg-cong*[*OF content-times-primitive-part*[*of g*], *of lead-coeff*, *unfolded lead-coeff-smult*]
lg content-ge-0-int[*of g*] **have** *lg′*: *lead-coeff ng* > *0* **unfolding** *ng-def*
by (*metis* ‹*content g* ≠ *0*› *dual-order.antisym dual-order.strict-implies-order zero-less-mult-iff*)
from *content-primitive-part*[*OF g0*] **have** *c-ng*: *content ng* = *1* **unfolding** *ng-def* **.**
have *degree ng* = *degree f* **using** ‹*content* [:*sgn* (*lead-coeff f*):] = *1*› *g-def ng-def*
by (*auto simp add*: *sgn-eq-0-iff*)
with *False* **have** *degree ng* ≠ *0* **by** *auto*

note *main* = *square-free-factorization-int-main*[*OF fs c-ng lg′ this*]
show *?thesis*
proof (*intro conjI impI*)
{
assume (*fi*, *i*) ∈ *set fs*
with *main* **show** *content fi* = *1 0* < *lead-coeff fi* **by** *auto*
}
have *d0*: *d* ≠ *0* **using** ‹*content* [:*?s*:] = *1*› *d* **by** (*auto simp*:*sgn-eq-0-iff*)
have *smult d ng* = *smult ?s* (*smult* (*content g*) (*primitive-part g*))
unfolding *ng-def d cg* **by** *simp*
also have *smult* (*content g*) (*primitive-part g*) = *g* **using** *content-times-primitive-part*
.
also have *smult ?s g* = *f* **unfolding** *g-def* **using** *s* **by** *auto*
finally have *id*: *smult d ng* = *f* **.**
from *main* **have** *square-free-factorization ng* (*1*, *fs*) **by** *auto*
from *square-free-factorization-smult*[*OF d0 this*]
show *square-free-factorization f* (*d*,*fs*) **unfolding** *id* **by** *simp*
show *distinct* (*map snd fs*) **using** *main* **by** *auto*
qed
qed
thus *square-free-factorization f* (*d*,*fs*)
(*fi*, *i*) ∈ *set fs* ⟹ *content fi* = *1* ∧ *lead-coeff fi* > *0 distinct* (*map snd fs*) **by**
auto
qed

definition *x-split* :: ′*a* :: *semiring-0 poly* ⇒ *nat* × ′*a poly* **where**
x-split f = (*let fs* = *coeffs f*; *zs* = *takeWhile* ((=) *0*) *fs*
in case zs of [] ⇒ (*0*,*f*) | *-* ⇒ (*length zs*, *poly-of-list* (*dropWhile* ((=) *0*) *fs*)))

lemma *x-split*: **assumes** *x-split f* = (*n*, *g*)
shows *f* = *monom 1 n* ∗ *g n* ≠ *0* ∨ *f* ≠ *0* ⟹ ¬ *monom 1 1 dvd g*
proof −
define *zs* **where** *zs* = *takeWhile* ((=) *0*) (*coeffs f*)
note *res* = *assms*[*unfolded zs-def*[*symmetric*] *x-split-def Let-def*]
have *f* = *monom 1 n* ∗ *g* ∧ ((*n* ≠ *0* ∨ *f* ≠ *0*) ⟶ ¬ (*monom 1 1 dvd g*)) (**is** *-*
∧ (*-* ⟶ ¬ (*?x dvd -*)))
proof (*cases f* = *0*)
case *True*
with *res* **have** *n* = *0 g* = *0* **unfolding** *zs-def* **by** *auto*
thus *?thesis* **using** *True* **by** *auto*
next
case *False* **note** *f* = *this*
show *?thesis*
proof (*cases zs* = [])
case *True*
hence *choice*: *coeff f 0* ≠ *0* **using** *f* **unfolding** *zs-def coeff-f-0-code poly-compare-0-code*
by (*cases coeffs f*, *auto*)
have *dvd*: *?x dvd h* ⟷ *coeff h 0* = *0* **for** *h* **by** (*simp add*: *monom-1-dvd-iff′*)

from *True choice res f* **show** *?thesis* **unfolding** *dvd* **by** *auto*
next
case *False*
define *ys* **where** *ys* = *dropWhile* ((=) *0*) (*coeffs f*)
have *dvd*: *?x dvd h* ⟷ *coeff h 0* = *0* **for** *h* **by** (*simp add*: *monom-1-dvd-iff'*)
from *res False* **have** *n*: *n* = *length zs* **and** *g*: *g* = *poly-of-list ys* **unfolding** *ys-def*
by (*cases zs*, *auto*)+
obtain *xx* **where** *xx*: *coeffs f* = *xx* **by** *auto*
have *coeffs f* = *zs* @ *ys* **unfolding** *zs-def ys-def* **by** *auto*
also have *zs* = *replicate n 0* **unfolding** *zs-def n xx* **by** (*induct xx*, *auto*)
finally have *ff*: *coeffs f* = *replicate n 0* @ *ys* **by** *auto*
from *f* **have** *lead-coeff f* ≠ *0* **by** *auto*
then have *nz*: *coeffs f* ≠ [] *last* (*coeffs f*) ≠ *0*
by (*simp-all add*: *last-coeffs-eq-coeff-degree*)
have *ys*: *ys* ≠ [] **using** *nz*[*unfolded ff*] **by** *auto*
with *ys-def* **have** *hd*: *hd ys* ≠ *0* **by** (*metis* (*full-types*) *hd-dropWhile*)
hence *coeff* (*poly-of-list ys*) *0* ≠ *0* **unfolding** *poly-of-list-def coeff-Poly* **using** *ys* **by** (*cases ys*, *auto*)
moreover have *coeffs* (*Poly ys*) = *ys*
by (*simp add*: *ys-def strip-while-dropWhile-commute*)
then have *coeffs* (*monom-mult n* (*Poly ys*)) = *replicate n 0* @ *ys*
by (*simp add*: *coeffs-eq-iff monom-mult-def* [*symmetric*] *ff ys monom-mult-code*)
ultimately show *?thesis* **unfolding** *dvd g*
by (*auto simp add*: *coeffs-eq-iff monom-mult-def* [*symmetric*] *ff*)
qed
qed
thus *f* = *monom 1 n* ∗ *g n* ≠ *0* ∨ *f* ≠ *0* ⟹ ¬ *monom 1 1 dvd g* **by** *auto*
qed

definition *square-free-factorization-int* :: *int poly* ⇒ *int* × (*int poly* × *nat*)*list* **where**
square-free-factorization-int f = (*case x-split f of* (*n*,*g*) — extract *x^n*
⇒ *case square-free-factorization-int' g of* (*d*,*fs*)
⇒ *if n* = *0 then* (*d*,*fs*) *else* (*d*, (*monom 1 1*, *n* − *1*) # *fs*))

lemma *square-free-factorization-int*: **assumes** *res*: *square-free-factorization-int f* = (*d*, *fs*)
shows *square-free-factorization f* (*d*,*fs*)
(*fi*, *i*) ∈ *set fs* ⟹ *primitive fi* ∧ *lead-coeff fi* > *0*
proof −
obtain *n g* **where** *xs*: *x-split f* = (*n*,*g*) **by** *force*
obtain *c hs* **where** *sf*: *square-free-factorization-int' g* = (*c*,*hs*) **by** *force*
from *res*[*unfolded square-free-factorization-int-def xs sf split*]
have *d*: *d* = *c* **and** *fs*: *fs* = (*if n* = *0 then hs else* (*monom 1 1*, *n* − *1*) # *hs*) **by** (*cases n*, *auto*)
note *sff* = *square-free-factorization-int'*(*1*−*2*)[*OF sf*]
note *xs* = *x-split*[*OF xs*]

let *?x* = *monom 1 1* :: *int poly*
have *x*: *primitive ?x* ∧ *lead-coeff ?x* = *1* ∧ *degree ?x* = *1*
by (*auto simp add*: *degree-monom-eq content-def monom-Suc*)
thus (*fi*, *i*) ∈ *set fs* ⟹ *primitive fi* ∧ *lead-coeff fi* > *0* **using** *sff*(*2*) **unfolding** *fs*
by (*cases n*, *auto*)
show *square-free-factorization f* (*d*,*fs*)
proof (*cases n*)
case *0*
with *d fs sff xs* **show** *?thesis* **by** *auto*
next
case (*Suc m*)
with *xs* **have** *fg*: *f* = *monom 1* (*Suc m*) ∗ *g* **and** *dvd*: ¬ *?x dvd g* **by** *auto*
from *Suc* **have** *fs*: *fs* = (*?x*,*m*) # *hs* **unfolding** *fs* **by** *auto*
have *degx*: *degree ?x* = *1* **by** *code-simp*
from *irreducible$_d$-square-free*[*OF linear-irreducible$_d$*[*OF this*]] **have** *sfx*: *square-free ?x* **by** *auto*
have *fg*: *f* = *?x* ^ *n* ∗ *g* **unfolding** *fg Suc* **by** (*metis x-pow-n*)
have *eq0*: *?x* ^ *n* ∗ *g* = *0* ⟷ *g* = *0* **by** *simp*
note *sf* = *square-free-factorizationD*[*OF sff*(*1*)]
{
fix *a i*
assume *ai*: (*a*,*i*) ∈ *set hs*
with *sf*(*4*) **have** *g0*: *g* ≠ *0* **by** *auto*
from *split-list*[*OF ai*] **obtain** *ys zs* **where** *hs*: *hs* = *ys* @ (*a*,*i*) # *zs* **by** *auto*
have *a dvd g* **unfolding** *square-free-factorization-prod-list*[*OF sff*(*1*)] *hs*
by (*rule dvd-smult*, *simp add*: *ac-simps*)
moreover have ¬ *?x dvd g* **using** *xs*[*unfolded Suc*] **by** *auto*
ultimately have *dvd*: ¬ *?x dvd a* **using** *dvd-trans* **by** *blast*
from *sf*(*2*)[*OF ai*] **have** *a* ≠ *0* **by** *auto*
have *1* = *gcd ?x a*
proof (*rule gcdI*)
fix *d*
assume *d*: *d dvd ?x d dvd a*
from *content-dvd-contentI*[*OF d*(*1*)] *x* **have** *cnt*: *is-unit* (*content d*) **by** *auto*
show *is-unit d*
proof (*cases degree d* = *1*)
case *False*
with *divides-degree*[*OF d*(*1*), *unfolded degx*] **have** *degree d* = *0* **by** *auto*
from *degree0-coeffs*[*OF this*] **obtain** *c* **where** *dc*: *d* = [:*c*:] **by** *auto*
from *cnt*[*unfolded dc*] **have** *is-unit c* **by** (*auto simp*: *content-def*, *cases c* = *0*, *auto*)
hence *d* ∗ *d* = *1* **unfolding** *dc* **by** (*cases c* = −*1*; *cases c* = *1*, *auto*)
thus *is-unit d* **by** (*metis dvd-triv-right*)
next
case *True*
from *d*(*1*) **obtain** *e* **where** *xde*: *?x* = *d* ∗ *e* **unfolding** *dvd-def* **by** *auto*
from *arg-cong*[*OF this*, *of degree*] *degx* **have** *degree d* + *degree e* = *1*
by (*metis True add.right-neutral degree-0 degree-mult-eq one-neq-zero*)

with *True* **have** *degree e* = *0* **by** *auto*
from *degree0-coeffs*[*OF this*] *xde* **obtain** *e* **where** *xde*: *?x* = [:*e*:] ∗ *d* **by** *auto*
from *arg-cong*[*OF this*, *of content*, *unfolded content-mult*] *x*
have *content* [:*e*:] ∗ *content d* = *1* **by** *auto*
also have *content* [:*e* :] = *abs e* **by** (*auto simp*: *content-def*, *cases e* = *0*, *auto*)
finally have |*e*| ∗ *content d* = *1* **.**
from *pos-zmult-eq-1-iff-lemma*[*OF this*] **have** *e* ∗ *e* = *1* **by** (*cases e* = *1*; *cases e* = −*1*, *auto*)
with *arg-cong*[*OF xde*, *of smult e*] **have** *d* = *?x* ∗ [:*e*:] **by** *auto*
hence *?x dvd d* **unfolding** *dvd-def* **by** *blast*
with *d*(*2*) **have** *?x dvd a* **by** (*metis dvd-trans*)
with *dvd* **show** *?thesis* **by** *auto*
qed
qed *auto*
hence *coprime ?x a*
by (*simp add*: *gcd-eq-1-imp-coprime*)
note *this dvd*
} **note** *hs-dvd-x* = *this*
from *hs-dvd-x*[*of ?x m*]
have *nmem*: (*?x*,*m*) ∉ *set hs* **by** *auto*
hence *eq*: *?x* ^ *n* ∗ *g* = *smult c* (∏(*a*, *i*)∈*set fs*. *a* ^ *Suc i*)
unfolding *sf*(*1*) **unfolding** *fs Suc* **by** *simp*
show *?thesis* **unfolding** *fg d* **unfolding** *square-free-factorization-def split eq0* **unfolding** *eq*
proof (*intro conjI allI impI*, *rule refl*)
fix *a i*
assume *ai*: (*a*,*i*) ∈ *set fs*
thus *square-free a degree a* > *0* **using** *sf*(*2*) *sfx degx* **unfolding** *fs* **by** *auto*
fix *b j*
assume *bj*: (*b*,*j*) ∈ *set fs* **and** *diff*: (*a*,*i*) ≠ (*b*,*j*)
consider (*hs-hs*) (*a*,*i*) ∈ *set hs* (*b*,*j*) ∈ *set hs*
| (*hs-x*) (*a*,*i*) ∈ *set hs b* = *?x*
| (*x-hs*) (*b*,*j*) ∈ *set hs a* = *?x*
using *ai bj diff* **unfolding** *fs* **by** *auto*
then show *Rings.coprime a b*
proof *cases*
case *hs-hs*
from *sf*(*3*)[*OF this diff*] **show** *?thesis* **.**
next
case *hs-x*
from *hs-dvd-x*(*1*)[*OF hs-x*(*1*)] **show** *?thesis* **unfolding** *hs-x*(*2*) **by** (*simp add*: *ac-simps*)
next
case *x-hs*
from *hs-dvd-x*(*1*)[*OF x-hs*(*1*)] **show** *?thesis* **unfolding** *x-hs*(*2*) **by** *simp*
qed
next

    show *g* = *0* ⟹ *c* = *0* **using** *sf*(*4*) **by** *auto*
    show *g* = *0* ⟹ *fs* = [] **using** *sf*(*4*) *xs Suc* **by** *auto*
    show *distinct fs* **using** *sf*(*5*) *nmem* **unfolding** *fs* **by** *auto*
  qed
 qed
qed

end

11.3 Factoring Arbitrary Integer Polynomials

We combine the factorization algorithm for square-free integer polynomials with a square-free factorization algorithm to a factorization algorithm for integer polynomials which does not make any assumptions.

theory *Factorize-Int-Poly*
imports
 Berlekamp-Zassenhaus
 Square-Free-Factorization-Int
begin

hide-const *coeff monom*
lifting-forget *poly.lifting*

typedef *int-poly-factorization-algorithm* = {*alg*.
 ∀ (*f* :: *int poly*) *fs*. *square-free f* ⟶ *degree f* > *0* ⟶ *alg f* = *fs* ⟶
 (*f* = *prod-list fs* ∧ (∀ *fi* ∈ *set fs*. *irreducible*$_d$ *fi*))}
 by (*rule exI*[*of* - *berlekamp-zassenhaus-factorization*],
   insert berlekamp-zassenhaus-factorization-irreducible$_d$, *auto*)

setup-lifting *type-definition-int-poly-factorization-algorithm*

lift-definition *int-poly-factorization-algorithm* :: *int-poly-factorization-algorithm* ⇒
 (*int poly* ⇒ *int poly list*) **is** λ *x*. *x* **.**

lemma *int-poly-factorization-algorithm-irreducible*$_d$:
 assumes *int-poly-factorization-algorithm alg f* = *fs*
 and *square-free f*
 and *degree f* > *0*
shows *f* = *prod-list fs* ∧ (∀ *fi* ∈ *set fs*. *irreducible*$_d$ *fi*)
 using *assms* **by** (*transfer*, *auto*)

corollary *int-poly-factorization-algorithm-irreducible*:
 assumes *res*: *int-poly-factorization-algorithm alg f* = *fs*
 and *sf*: *square-free f*
 and *deg*: *degree f* > *0*
 and *pr*: *primitive f*
 shows *f* = *prod-list fs* ∧ (∀ *fi* ∈ *set fs*. *irreducible fi* ∧ *degree fi* > *0* ∧ *primitive fi*)

proof (*intro conjI ballI*)
  note $*$ = *int-poly-factorization-algorithm-irreducible*$_d$[*OF res sf deg*]
  from $*$ **show** *f*: *f* = *prod-list fs* **by** *auto*
  fix *fi* **assume** *fi*: *fi* ∈ *set fs*
  with *primitive-prod-list*[*OF pr*[*unfolded f*]] **show** *primitive fi* **by** *auto*
  from *irreducible-primitive-connect*[*OF this*] $*$ *pr*[*unfolded f*] *fi*
  show *irreducible fi* **by** *auto*
  from $*$ *fi* **show** *degree fi* > *0* **by** (*auto*)
qed

lemma *irreducible-imp-square-free*:
  assumes *irr*: *irreducible* (*p*::'*a*::*idom poly*) **shows** *square-free p*
proof(*intro square-freeI*)
  from *irr* **show** *p0*: *p* ≠ *0* **by** *auto*
  fix *a* **assume** *a* $*$ *a dvd p*
  then obtain *b* **where** *paab*: *p* = *a* $*$ (*a* $*$ *b*) **by** (*elim dvdE*, *auto*)
  assume *degree a* > *0*
  then have *a1*: ¬ *a dvd 1* **by** (*auto simp*: *poly-dvd-1*)
  then have *ab1*: ¬ *a* $*$ *b dvd 1* **using** *dvd-mult-left* **by** *auto*
  from *paab irr a1 ab1* **show** *False* **by** *force*
qed

lemma *not-mem-set-dropWhileD*: *x* ∉ *set* (*dropWhile P xs*) ⟹ *x* ∈ *set xs* ⟹ *P x*
  by (*metis dropWhile-append3 in-set-conv-decomp*)

lemma *primitive-reflect-poly*:
  fixes *f* :: '*a* :: *comm-semiring-1 poly*
  shows *primitive* (*reflect-poly f*) = *primitive f*
proof−
  have (∀ *a* ∈ *set* (*coeffs f*). *x dvd a*) ⟷ (∀ *a* ∈ *set* (*dropWhile* ((=) *0*) (*coeffs f*)). *x dvd a*) **for** *x*
    by (*auto dest*: *not-mem-set-dropWhileD set-dropWhileD*)
  then show *?thesis* **by** (*auto simp*: *primitive-def coeffs-reflect-poly*)
qed

lemma *gcd-list-sub*:
  assumes *set xs* ⊆ *set ys* **shows** *gcd-list ys dvd gcd-list xs*
  by (*metis Gcd-fin.subset assms semiring-gcd-class.gcd-dvd1*)

lemma *content-reflect-poly*:
  content (*reflect-poly f*) = *content f* (**is** *?l* = *?r*)
proof−
  have *l*: *?l* = *gcd-list* (*dropWhile* ((=) *0*) (*coeffs f*)) (**is** - = *gcd-list ?xs*)
    by (*simp add*: *content-def reflect-poly-def*)
  have *set ?xs* ⊆ *set* (*coeffs f*) **by** (*auto dest*: *set-dropWhileD*)
  from *gcd-list-sub*[*OF this*]

have *?r dvd gcd-list ?xs* **by** (*simp add*: *content-def*)
with *l* **have** *rl*: *?r dvd ?l* **by** *auto*
have *set* (*coeffs f*) $\subseteq$ *set* (*0 # ?xs*) **by** (*auto dest*: *not-mem-set-dropWhileD*)
from *gcd-list-sub*[*OF this*]
have *gcd-list ?xs dvd ?r* **by** (*simp add*: *content-def*)
with *l* **have** *lr*: *?l dvd ?r* **by** *auto*
from *rl lr* **show** *?l* = *?r* **by** (*simp add*: *associated-eqI*)
qed

lemma *coeff-primitive-part*: *content f* $*$ *coeff* (*primitive-part f*) *i* = *coeff f i*
using *arg-cong*[*OF content-times-primitive-part*[*of f*], *of* λf. *coeff f -*, *unfolded coeff-smult*].

lemma *smult-cancel*[*simp*]:
fixes *c* :: *'a* :: *idom*
shows *smult c f* = *smult c g* $\longleftrightarrow$ *c* = *0* $\vee$ *f* = *g*
proof−
have *l*: *smult c f* = [:*c*:] $*$ *f* **by** *simp*
have *r*: *smult c g* = [:*c*:] $*$ *g* **by** *simp*
show *?thesis* **unfolding** *l r mult-cancel-left* **by** *simp*
qed

lemma *primitive-part-reflect-poly*:
fixes *f* :: *'a* :: {*semiring-gcd*,*idom*} *poly*
shows *primitive-part* (*reflect-poly f*) = *reflect-poly* (*primitive-part f*) (**is** *?l* = *?r*)
using *content-times-primitive-part*[*of reflect-poly f*]
proof−
note *content-reflect-poly*[*of f*, *symmetric*]
also have *smult* (*content* (*reflect-poly f*)) *?l* = *reflect-poly f* **by** *simp*
also have ... = *reflect-poly* (*smult* (*content f*) (*primitive-part f*)) **by** *simp*
finally show *?thesis* **unfolding** *reflect-poly-smult smult-cancel* **by** *auto*
qed

lemma *reflect-poly-eq-zero*[*simp*]:
reflect-poly f = *0* $\longleftrightarrow$ *f* = *0*
proof
assume *reflect-poly f* = *0*
then have *coeff* (*reflect-poly f*) *0* = *0* **by** *simp*
then have *lead-coeff f* = *0* **by** *simp*
then show *f* = *0* **by** *simp*
qed *simp*

lemma *irreducible$_d$-reflect-poly-main*:
fixes *f* :: *'a* :: {*idom*, *semiring-gcd*} *poly*
assumes *nz*: *coeff f 0* $\neq$ *0*
and *irr*: *irreducible$_d$* (*reflect-poly f*)
shows *irreducible$_d$ f*

proof
  let *?r* = *reflect-poly*
  from *irr degree-reflect-poly-eq*[*OF nz*] **show** *degree f* > *0* **by** *auto*
  fix *g h*
  assume *deg*: *degree g* < *degree f degree h* < *degree f* **and** *fgh*: *f* = *g* ∗ *h*
  from *arg-cong*[*OF fgh*, *of* λ *f*. *coeff f 0*] *nz*
  have *nz′*: *coeff g 0* ≠ *0* **by** (*auto simp*: *coeff-mult-0*)
  note *rfgh* = *arg-cong*[*OF fgh*, *of reflect-poly*, *unfolded reflect-poly-mult*[*of g h*]]
 from *deg degree-reflect-poly-le*[*of g*] *degree-reflect-poly-le*[*of h*] *degree-reflect-poly-eq*[*OF nz*]
  have *degree* (*?r h*) < *degree* (*?r f*) *degree* (*?r g*) < *degree* (*?r f*) **by** *auto*
  with *irr rfgh* **show** *False* **by** *auto*
qed

lemma *irreducible$_d$-reflect-poly*:
  fixes *f* :: *′a* :: {*idom*, *semiring-gcd*} *poly*
  assumes *nz*: *coeff f 0* ≠ *0*
  shows *irreducible$_d$* (*reflect-poly f*) = *irreducible$_d$ f*
proof
  assume *irreducible$_d$* (*reflect-poly f*)
  from *irreducible$_d$-reflect-poly-main*[*OF nz this*] **show** *irreducible$_d$ f* **.**
next
  from *nz* **have** *nzr*: *coeff* (*reflect-poly f*) *0* ≠ *0* **by** *auto*
  assume *irreducible$_d$ f*
  with *nz* **have** *irreducible$_d$* (*reflect-poly* (*reflect-poly f*)) **by** *simp*
  from *irreducible$_d$-reflect-poly-main*[*OF nzr this*]
  show *irreducible$_d$* (*reflect-poly f*) **.**
qed

lemma *irreducible-reflect-poly*:
  fixes *f* :: *′a* :: {*idom*,*semiring-gcd*} *poly*
  assumes *nz*: *coeff f 0* ≠ *0*
  shows *irreducible* (*reflect-poly f*) = *irreducible f* (**is** *?l* = *?r*)
proof (*cases degree f* = *0*)
  case *True* **then obtain** *f0* **where** *f* = [:*f0*:] **by** (*auto dest*: *degree0-coeffs*)
  then show *?thesis* **by** *simp*
next
  case *deg*: *False*
  show *?thesis*
  proof (*cases primitive f*)
    case *False*
    with *deg irreducible-imp-primitive*[*of f*] *irreducible-imp-primitive*[*of reflect-poly f*] *nz*
    show *?thesis* **unfolding** *primitive-reflect-poly* **by** *auto*
  next
    case *cf*: *True*
    let *?r* = *reflect-poly*
    from *nz* **have** *nz′*: *coeff* (*?r f*) *0* ≠ *0* **by** *auto*
    let *?ir* = *irreducible$_d$*

from *irreducible*$_d$*-reflect-poly*[*OF nz*] *irreducible*$_d$*-reflect-poly*[*OF nz'*] *nz*
have *?ir f* $\longleftrightarrow$ *?ir* (*reflect-poly f*) **by** *auto*
also have ... $\longleftrightarrow$ *irreducible* (*reflect-poly f*)
by (*rule irreducible-primitive-connect*, *unfold primitive-reflect-poly*, *fact cf*)
finally show *?thesis*
by (*unfold irreducible-primitive-connect*[*OF cf*], *auto*)
qed
qed

lemma *reflect-poly-dvd*: (*f* :: *'a* :: *idom poly*) *dvd g* $\Longrightarrow$ *reflect-poly f dvd reflect-poly g*
unfolding *dvd-def* **by** (*auto simp*: *reflect-poly-mult*)

lemma *square-free-reflect-poly*: **fixes** *f* :: *'a* :: *idom poly*
assumes *sf*: *square-free f*
and *nz*: *coeff f 0* $\neq$ *0*
shows *square-free* (*reflect-poly f*) **unfolding** *square-free-def*
proof (*intro allI conjI impI notI*)
let *?r* = *reflect-poly*
from *sf*[*unfolded square-free-def*]
have *f0*: *f* $\neq$ *0* **and** *sf*: $\bigwedge$ *q*. *0* < *degree q* $\Longrightarrow$ *q* $*$ *q dvd f* $\Longrightarrow$ *False* **by** *auto*
from *f0 nz* **show** *?r f* = *0* $\Longrightarrow$ *False* **by** *auto*
fix *q*
assume *0*: *0* < *degree q* **and** *dvd*: *q* $*$ *q dvd ?r f*
from *dvd* **have** *q dvd ?r f* **by** *auto*
then obtain *x* **where** *id*: *?r f* = *q* $*$ *x* **by** *fastforce*
{
assume *coeff q 0* = *0*
hence *coeff* (*?r f*) *0* = *0* **using** *id* **by** (*auto simp*: *coeff-mult*)
with *nz* **have** *False* **by** *auto*
}
hence *nzq*: *coeff q 0* $\neq$ *0* **by** *auto*
from *dvd* **have** *?r* (*q* $*$ *q*) *dvd ?r* (*?r f*) **by** (*rule reflect-poly-dvd*)
also have *?r* (*?r f*) = *f* **using** *nz* **by** *auto*
also have *?r* (*q* $*$ *q*) = *?r q* $*$ *?r q* **by** (*rule reflect-poly-mult*)
finally have *?r q* $*$ *?r q dvd f* **.**
from *sf*[*OF - this*] *0 nzq* **show** *False* **by** *simp*
qed

lemma *gcd-reflect-poly*: **fixes** *f* :: *'a* :: {*factorial-ring-gcd*, *semiring-gcd-mult-normalize*} *poly*
assumes *nz*: *coeff f 0* $\neq$ *0 coeff g 0* $\neq$ *0*
shows *gcd* (*reflect-poly f*) (*reflect-poly g*) = *normalize* (*reflect-poly* (*gcd f g*))
proof (*rule sym*, *rule gcdI*)
have *gcd f g dvd f* **by** *auto*
from *reflect-poly-dvd*[*OF this*]
show *normalize* (*reflect-poly* (*gcd f g*)) *dvd reflect-poly f* **by** *simp*
have *gcd f g dvd g* **by** *auto*

from *reflect-poly-dvd*[*OF this*]
show *normalize* (*reflect-poly* (*gcd f g*)) *dvd reflect-poly g* **by** *simp*
show *normalize* (*normalize* (*reflect-poly* (*gcd f g*))) = *normalize* (*reflect-poly* (*gcd f g*)) **by** *auto*
fix *h*
assume *hf*: *h dvd reflect-poly f* **and** *hg*: *h dvd reflect-poly g*
from *hf* **obtain** *k* **where** *reflect-poly f* = *h* ∗ *k* **unfolding** *dvd-def* **by** *auto*
from *arg-cong*[*OF this*, *of* λ *f*. *coeff f 0*, *unfolded coeff-mult-0*] *nz*(*1*) **have** *h*: *coeff h 0* ≠ *0* **by** *auto*
from *reflect-poly-dvd*[*OF hf*] *reflect-poly-dvd*[*OF hg*]
have *reflect-poly h dvd f reflect-poly h dvd g* **using** *nz* **by** *auto*
hence *reflect-poly h dvd gcd f g* **by** *auto*
from *reflect-poly-dvd*[*OF this*] *h* **have** *h dvd reflect-poly* (*gcd f g*) **by** *auto*
thus *h dvd normalize* (*reflect-poly* (*gcd f g*)) **by** *auto*
qed

lemma *linear-primitive-irreducible*:
fixes *f* :: *'a* :: {*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
assumes *deg*: *degree f* = *1* **and** *cf*: *primitive f*
shows *irreducible f*
proof (*intro irreducibleI*)
fix *a b* **assume** *fab*: *f* = *a* ∗ *b*
with *deg* **have** *a0*: *a* ≠ *0* **and** *b0*: *b* ≠ *0* **by** *auto*
from *deg*[*unfolded fab*] *degree-mult-eq*[*OF this*] **have** *degree a* = *0* ∨ *degree b* = *0* **by** *auto*
then show *a dvd 1* ∨ *b dvd 1*
proof
assume *degree a* = *0*
then obtain *a0* **where** *a*: *a* = [:*a0*:] **by** (*auto dest*:*degree0-coeffs*)
with *fab* **have** *c* ∈ *set* (*coeffs f*) ⟹ *a0 dvd c* **for** *c* **by** (*cases a0* = *0*, *auto simp*: *coeffs-smult*)
with *cf* **show** *?thesis* **by** (*auto dest*: *primitiveD simp*: *a*)
next
assume *degree b* = *0*
then obtain *b0* **where** *b*: *b* = [:*b0*:] **by** (*auto dest*:*degree0-coeffs*)
with *fab* **have** *c* ∈ *set* (*coeffs f*) ⟹ *b0 dvd c* **for** *c* **by** (*cases b0* = *0*, *auto simp*: *coeffs-smult*)
with *cf* **show** *?thesis* **by** (*auto dest*: *primitiveD simp*: *b*)
qed
qed (*insert deg*, *auto simp*: *poly-dvd-1*)

lemma *square-free-factorization-last-coeff-nz*:
assumes *sff*: *square-free-factorization f* (*a*, *fs*)
and *mem*: (*fi*,*i*) ∈ *set fs*
and *nz*: *coeff f 0* ≠ *0*
shows *coeff fi 0* ≠ *0*
proof
assume *fi*: *coeff fi 0* = *0*
note *sff-list* = *square-free-factorization-prod-list*[*OF sff*]

note *sff* = *square-free-factorizationD*[*OF sff*]
from *sff-list* **have** *coeff f 0* = *a* ∗ *coeff* ($\prod$(*a*, *i*)←*fs*. *a* ^ *Suc i*) *0* **by** *simp*
with *split-list*[*OF mem*] *fi* **have** *coeff f 0* = *0*
by (*auto simp*: *coeff-mult*)
with *nz* **show** *False* **by** *simp*
qed

context
fixes *alg* :: *int-poly-factorization-algorithm*
begin

definition *main-int-poly-factorization* :: *int poly* ⇒ *int poly list* **where**
main-int-poly-factorization f = (*let df* = *degree f*
in if df = *1 then* [*f*] *else*
if abs (*coeff f 0*) < *abs* (*coeff f df*) — take reciprocal polynomial, if *f*(*0*) < *lc*(*f*)
then map reflect-poly (*int-poly-factorization-algorithm alg* (*reflect-poly f*))
else int-poly-factorization-algorithm alg f)

definition *internal-int-poly-factorization* :: *int poly* ⇒ *int* × (*int poly* × *nat*) *list*
where
internal-int-poly-factorization f = (
case square-free-factorization-int f of
(*a*,*gis*) ⇒ (*a*, [(*h*,*i*) . (*g*,*i*) ← *gis*, *h* ← *main-int-poly-factorization g*])
)

lemma *internal-int-poly-factorization-code*[*code*]: *internal-int-poly-factorization f* =
(
case square-free-factorization-int f of (*a*,*gis*) ⇒
(*a*, *concat* (*map* (λ (*g*,*i*). (*map* (λ *f*. (*f*,*i*)) (*main-int-poly-factorization g*))) *gis*)))
unfolding *internal-int-poly-factorization-def* **by** *auto*

definition *factorize-int-last-nz-poly* :: *int poly* ⇒ *int* × (*int poly* × *nat*) *list* **where**
factorize-int-last-nz-poly f = (*let df* = *degree f*
in if df = *0 then* (*coeff f 0*, []) *else if df* = *1 then* (*content f*,[(*primitive-part f*,*0*)]) *else*
internal-int-poly-factorization f)

definition *factorize-int-poly-generic* :: *int poly* ⇒ *int* × (*int poly* × *nat*) *list* **where**
factorize-int-poly-generic f = (*case x-split f of* (*n*,*g*) — extract *x*^*n*
⇒ *if g* = *0 then* (*0*,[]) *else case factorize-int-last-nz-poly g of* (*a*,*fs*)
⇒ *if n* = *0 then* (*a*,*fs*) *else* (*a*, (*monom 1 1*, *n* − *1*) # *fs*))

lemma *factorize-int-poly-0*[*simp*]: *factorize-int-poly-generic 0* = (*0*,[])

```
  unfolding factorize-int-poly-generic-def x-split-def by simp

lemma main-int-poly-factorization:
  assumes res: main-int-poly-factorization f = fs
  and sf: square-free f
  and df: degree f > 0
  and nz: coeff f 0 ≠ 0
shows f = prod-list fs ∧ (∀ fi ∈ set fs. irreducible_d fi)
proof (cases degree f = 1)
  case True
  with res[unfolded main-int-poly-factorization-def Let-def]
  have fs = [f] by auto
  with True show ?thesis by auto
next
  case False
  hence *: (if degree f = 1 then t :: int poly list else e) = e for t e by auto
  note res = res[unfolded main-int-poly-factorization-def Let-def *]
  show ?thesis
  proof (cases abs (coeff f 0) < abs (coeff f (degree f)))
    case False
    with res have int-poly-factorization-algorithm alg f = fs by auto
    from int-poly-factorization-algorithm-irreducible_d[OF this sf df] show ?thesis .
  next
    case True
    let ?f = reflect-poly f
    from square-free-reflect-poly[OF sf nz] have sf: square-free ?f .
    from nz df have df: degree ?f > 0 by simp
    from True res obtain gs where fs: fs = map reflect-poly gs
      and gs: int-poly-factorization-algorithm alg (reflect-poly f) = gs
      by auto
    from int-poly-factorization-algorithm-irreducible_d[OF gs sf df]
    have id: reflect-poly ?f = reflect-poly (prod-list gs) ?f = prod-list gs
      and irr: ⋀ gi. gi ∈ set gs ⟹ irreducible_d gi by auto
    from id(1) have f-fs: f = prod-list fs unfolding fs using nz
      by (simp add: reflect-poly-prod-list)
    {
      fix fi
      assume fi ∈ set fs
     from this[unfolded fs] obtain gi where gi: gi ∈ set gs and fi: fi = reflect-poly
gi by auto
      {
        assume coeff gi 0 = 0
        with id(2) split-list[OF gi] have coeff ?f 0 = 0
          by (auto simp: coeff-mult)
        with nz have False by auto
      }
      hence nzg: coeff gi 0 ≠ 0 by auto
     from irreducible_d-reflect-poly[OF nzg] irr[OF gi] have irreducible_d fi unfold-
ing fi by simp
```

}
with *f-fs* **show** *?thesis* **by** *auto*
qed
qed

lemma *internal-int-poly-factorization-mem*:
assumes *f*: *coeff f 0* $\neq$ *0*
and *res*: *internal-int-poly-factorization f* = (*c*,*fs*)
and *mem*: (*fi*,*i*) $\in$ *set fs*
shows *irreducible fi* $irreducible_d$ *fi* **and** *primitive fi* **and** *degree fi* $\neq$ *0*
proof −
obtain *a psi* **where** *a-psi*: *square-free-factorization-int f* = (*a*, *psi*)
by *force*
from *square-free-factorization-int*[*OF this*]
have *sff*: *square-free-factorization f* (*a*, *psi*)
and *cnt*: $\bigwedge$ *fi i*. (*fi*, *i*) $\in$ *set psi* $\Longrightarrow$ *primitive fi* **by** *blast+*
from *square-free-factorization-last-coeff-nz*[*OF sff* - *f*]
have *nz-fi*: $\bigwedge$ *fi i*. (*fi*, *i*) $\in$ *set psi* $\Longrightarrow$ *coeff fi 0* $\neq$ *0* **by** *auto*
note *res* = *res*[*unfolded internal-int-poly-factorization-def a-psi Let-def split*]
obtain *fact* **where** *fact*: *fact* = (λ (*q*,*i* :: *nat*). (*map* (λ *f*. (*f*,*i*)) (*main-int-poly-factorization q*))) **by** *auto*
from *res*[*unfolded split Let-def*]
have *c*: *c* = *a* **and** *fs*: *fs* = *concat* (*map fact psi*)
unfolding *fact* **by** *auto*
note *sff'* = *square-free-factorizationD*[*OF sff*]
from *mem*[*unfolded fs*, *simplified*] **obtain** *d j* **where** *psi*: (*d*,*j*) $\in$ *set psi*
and *fi*: (*fi*, *i*) $\in$ *set* (*fact* (*d*,*j*)) **by** *auto*
obtain *hs* **where** *d*: *main-int-poly-factorization d* = *hs* **by** *force*
from *fi*[*unfolded d split fact*] **have** *fi*: *fi* $\in$ *set hs* **by** *auto*
from *main-int-poly-factorization*[*OF d* - - *nz-fi*[*OF psi*]] *sff'*(*2*)[*OF psi*] *cnt*[*OF psi*]
have *main*: *d* = *prod-list hs* $\bigwedge$ *fi*. *fi* $\in$ *set hs* $\Longrightarrow$ $irreducible_d$ *fi* **by** *auto*
from *main split-list*[*OF fi*] **have** *content fi dvd content d* **by** *auto*
with *cnt*[*OF psi*] **show** *cnt*: *primitive fi* **by** *simp*
from *main*(*2*)[*OF fi*] **show** *irr*: $irreducible_d$ *fi* **.**
show *irreducible fi*
using *irreducible-primitive-connect*[*OF cnt*] *irr* **by** *blast*
from *irr* **show** *degree fi* $\neq$ *0* **by** *auto*
qed

lemma *internal-int-poly-factorization*:
assumes *f*: *coeff f 0* $\neq$ *0*
and *res*: *internal-int-poly-factorization f* = (*c*,*fs*)
shows *square-free-factorization f* (*c*,*fs*)
proof −
obtain *a psi* **where** *a-psi*: *square-free-factorization-int f* = (*a*, *psi*)
by *force*
from *square-free-factorization-int*[*OF this*]
have *sff*: *square-free-factorization f* (*a*, *psi*)

and *pr*: $\bigwedge$ *fi i*. (*fi*, *i*) ∈ *set psi* ⟹ *primitive fi* **by** *blast+*
obtain *fact* **where** *fact*: *fact* = (λ (*q*,*i* :: *nat*). (*map* (λ *f*. (*f*,*i*)) (*main-int-poly-factorization q*))) **by** *auto*
from *res*[*unfolded split Let-def*]
have *c*: *c* = *a* **and** *fs*: *fs* = *concat* (*map fact psi*)
unfolding *fact internal-int-poly-factorization-def a-psi* **by** *auto*
note *sff'* = *square-free-factorizationD*[*OF sff*]
show *?thesis* **unfolding** *square-free-factorization-def split*
proof (*intro conjI impI allI*)
show *f* = *0* ⟹ *c* = *0 f* = *0* ⟹ *fs* = [] **using** *sff'*(*4*) **unfolding** *c fs* **by** *auto*
{
fix *a i*
assume (*a*,*i*) ∈ *set fs*
from *irreducible-imp-square-free internal-int-poly-factorization-mem*[*OF f res this*]
show *square-free a degree a* > *0* **by** *auto*
}
from *square-free-factorization-last-coeff-nz*[*OF sff - f*]
have *nz*: $\bigwedge$ *fi i*. (*fi*, *i*) ∈ *set psi* ⟹ *coeff fi 0* ≠ *0* **by** *auto*
have *eq*: *f* = *smult c* (∏(*a*, *i*)←*fs*. *a* ^ *Suc i*) **unfolding**
prod.distinct-set-conv-list[*OF sff'*(*5*)]
sff'(*1*) *c*
proof (*rule arg-cong*[**where** *f* = *smult a*], *unfold fs*, *insert sff'*(*2*) *nz*, *induct psi*)
case (*Cons pi psi*)
obtain *p i* **where** *pi*: *pi* = (*p*,*i*) **by** *force*
obtain *gs* **where** *gs*: *main-int-poly-factorization p* = *gs* **by** *auto*
from *Cons*(*2*)[*of p i*] **have** *p*: *square-free p degree p* > *0* **unfolding** *pi* **by** *auto*
from *Cons*(*3*)[*of p i*] **have** *nz*: *coeff p 0* ≠ *0* **unfolding** *pi* **by** *auto*
from *main-int-poly-factorization*[*OF gs p nz*] **have** *pgs*: *p* = *prod-list gs* **by** *auto*
have *fact*: *fact* (*p*,*i*) = *map* (λ *g*. (*g*,*i*)) *gs* **unfolding** *fact split gs* **by** *auto*
have *cong*: $\bigwedge$ *x y X Y*. *x* = *X* ⟹ *y* = *Y* ⟹ *x* ∗ *y* = *X* ∗ *Y* **by** *auto*
show *?case* **unfolding** *pi list.simps prod-list.Cons split fact concat.simps prod-list.append*
map-append
proof (*rule cong*)
show *p* ^ *Suc i* = (∏(*a*, *i*)←*map* (λ*g*. (*g*, *i*)) *gs*. *a* ^ *Suc i*) **unfolding** *pgs*
by (*induct gs*, *auto simp*: *ac-simps power-mult-distrib*)
show (∏(*a*, *i*)←*psi*. *a* ^ *Suc i*) = (∏(*a*, *i*)←*concat* (*map fact psi*). *a* ^ *Suc i*)
by (*rule Cons*(*1*), *insert Cons*(*2*−*3*), *auto*)
qed
qed *simp*
{
fix *i j l fi*
assume ∗: *j* < *length psi l* < *length* (*fact* (*psi* ! *j*)) *fact* (*psi* ! *j*) ! *l* = (*fi*, *i*)
from ∗ **have** *psi*: *psi* ! *j* ∈ *set psi* **by** *auto*

obtain *d k* **where** *dk*: *psi* ! *j* = (*d*,*k*) **by** *force*
with ∗ **have** *psij*: *psi* ! *j* = (*d*,*i*) **unfolding** *fact split* **by** *auto*
from *sff'*(*2*)[*OF psi*[*unfolded psij*]] **have** *d*: *square-free d degree d* > *0* **by** *auto*
from *nz*[*OF psi*[*unfolded psij*]] **have** *d0*: *coeff d 0* ≠ *0* **.**
from ∗ *psij fact*
have *bz*: *main-int-poly-factorization d* = *map fst* (*fact* (*psi* ! *j*)) **by** (*auto simp*: *o-def*)
from *main-int-poly-factorization*[*OF bz d d0*] *pr*[*OF psi*[*unfolded dk*]]
have *dhs*: *d* = *prod-list* (*map fst* (*fact* (*psi* ! *j*))) **by** *auto*
from ∗ **have** *mem*: *fi* ∈ *set* (*map fst* (*fact* (*psi* ! *j*)))
by (*metis fst-conv image-eqI nth-mem set-map*)
from *mem dhs psij d* **have** ∃ *d*. *fi* ∈ *set* (*map fst* (*fact* (*psi* ! *j*))) ∧
d = *prod-list* (*map fst* (*fact* (*psi* ! *j*))) ∧
psi ! *j* = (*d*, *i*) ∧
square-free d **by** *blast*
} **note** *deconstruct* = *this*
{
fix *k K fi i Fi I*
assume *k*: *k* < *length fs K* < *length fs* **and** *f*: *fs* ! *k* = (*fi*, *i*) *fs* ! *K* = (*Fi*, *I*)
and *diff*: *k* ≠ *K*
from *nth-concat-diff*[*OF k*[*unfolded fs*] *diff*, *folded fs*, *unfolded length-map*]
obtain *j l J L* **where** *diff*: (*j*, *l*) ≠ (*J*, *L*)
and *j*: *j* < *length psi J* < *length psi*
and *l*: *l* < *length* (*map fact psi* ! *j*) *L* < *length* (*map fact psi* ! *J*)
and *fs*: *fs* ! *k* = *map fact psi* ! *j* ! *l fs* ! *K* = *map fact psi* ! *J* ! *L* **by** *blast*+
hence *psij*: *psi* ! *j* ∈ *set psi* **by** *auto*
from *j* **have** *id*: *map fact psi* ! *j* = *fact* (*psi* ! *j*) *map fact psi* ! *J* = *fact* (*psi* ! *J*) **by** *auto*
note *l* = *l*[*unfolded id*] **note** *fs* = *fs*[*unfolded id*]
from *j* **have** *psi*: *psi* ! *j* ∈ *set psi psi* ! *J* ∈ *set psi* **by** *auto*
from *deconstruct*[*OF j*(*1*) *l*(*1*) *fs*(*1*)[*unfolded f*, *symmetric*]]
obtain *d* **where** *mem*: *fi* ∈ *set* (*map fst* (*fact* (*psi* ! *j*)))
and *d*: *d* = *prod-list* (*map fst* (*fact* (*psi* ! *j*))) *psi* ! *j* = (*d*, *i*) *square-free d* **by** *blast*
from *deconstruct*[*OF j*(*2*) *l*(*2*) *fs*(*2*)[*unfolded f*, *symmetric*]]
obtain *D* **where** *Mem*: *Fi* ∈ *set* (*map fst* (*fact* (*psi* ! *J*)))
and *D*: *D* = *prod-list* (*map fst* (*fact* (*psi* ! *J*))) *psi* ! *J* = (*D*, *I*) *square-free D* **by** *blast*
from *pr*[*OF psij*[*unfolded d*(*2*)]] **have** *cnt*: *primitive d* **.**
have *coprime fi Fi*
proof (*cases J* = *j*)
case *False*
from *sff'*(*5*) *False j* **have** (*d*,*i*) ≠ (*D*,*I*)
unfolding *distinct-conv-nth d*(*2*)[*symmetric*] *D*(*2*)[*symmetric*] **by** *auto*
from *sff'*(*3*)[*OF psi*[*unfolded d*(*2*) *D*(*2*)] *this*]
have *cop*: *coprime d D* **by** *auto*
from *prod-list-dvd*[*OF mem*, *folded d*(*1*)] **have** *fid*: *fi dvd d* **by** *auto*
from *prod-list-dvd*[*OF Mem*, *folded D*(*1*)] **have** *FiD*: *Fi dvd D* **by** *auto*

from *coprime-divisors*[*OF fid FiD*] *cop* **show** *?thesis* **by** *simp*
next
case *True* **note** *id* = *this*
from *id diff* **have** *diff*: $l \neq L$ **by** *auto*
obtain *bz* **where** *bz*: *bz* = *map fst* (*fact* (*psi* ! *j*)) **by** *auto*
from *fs*[*unfolded f*] *l*
have *fi*: *fi* = *bz* ! *l Fi* = *bz* ! *L*
unfolding *id bz* **by** (*metis fst-conv nth-map*)+
from *d*[*folded bz*] **have** *sf*: *square-free* (*prod-list bz*) **by** *auto*
from *d*[*folded bz*] *cnt* **have** *cnt*: *content* (*prod-list bz*) = *1* **by** *auto*
from *l* **have** *l*: *l* < *length bz L* < *length bz* **unfolding** *bz id* **by** *auto*
from *l fi* **have** *fi* ∈ *set bz* **by** *auto*
from *content-dvd-1*[*OF cnt prod-list-dvd*[*OF this*]] **have** *cnt*: *content fi* = *1*
.
obtain *g* **where** *g*: *g* = *gcd fi Fi* **by** *auto*
have *g′*: *g dvd fi g dvd Fi* **unfolding** *g* **by** *auto*
define *bef* **where** *bef* = *take l bz*
define *aft* **where** *aft* = *drop* (*Suc l*) *bz*
from *id-take-nth-drop*[*OF l*(*1*)] *l* **have** *bz*: *bz* = *bef* @ *fi* # *aft* **and** *bef*: *length bef* = *l*
unfolding *bef-def aft-def fi* **by** *auto*
with *l diff* **have** *mem*: *Fi* ∈ *set* (*bef* @ *aft*) **unfolding** *fi*(*2*) **by** (*auto simp*: *nth-append*)
from *split-list*[*OF this*] **obtain** *Bef Aft* **where** *ba*: *bef* @ *aft* = *Bef* @ *Fi* # *Aft* **by** *auto*
have *prod-list bz* = *fi* ∗ *prod-list* (*bef* @ *aft*) **unfolding** *bz* **by** *simp*
also have *prod-list* (*bef* @ *aft*) = *Fi* ∗ *prod-list* (*Bef* @ *Aft*) **unfolding** *ba* **by** *auto*
finally have *fi* ∗ *Fi dvd prod-list bz* **by** *auto*
with *g′* **have** *g* ∗ *g dvd prod-list bz* **by** (*meson dvd-trans mult-dvd-mono*)
with *sf*[*unfolded square-free-def*] **have** *deg*: *degree g* = *0* **by** *auto*
from *content-dvd-1*[*OF cnt g′*(*1*)] **have** *cnt*: *content g* = *1* **.**
from *degree0-coeffs*[*OF deg*] **obtain** *c* **where** *gc*: *g* = [: *c* :] **by** *auto*
from *cnt*[*unfolded gc content-def*, *simplified*] **have** *abs c* = *1*
by (*cases c* = *0*, *auto*)
with *g gc* **have** *gcd fi Fi* ∈ {*1*,−*1*} **by** *fastforce*
thus *coprime fi Fi*
by (*auto intro*!: *gcd-eq-1-imp-coprime*)
(*metis dvd-minus-iff dvd-refl is-unit-gcd-iff one-neq-neg-one*)
qed
} **note** *cop* = *this*
show *dist*: *distinct fs* **unfolding** *distinct-conv-nth*
proof (*intro impI allI*)
fix *k K*
assume *k*: *k* < *length fs K* < *length fs* **and** *diff*: $k \neq K$
obtain *fi i Fi I* **where** *f*: *fs* ! *k* = (*fi*,*i*) *fs* ! *K* = (*Fi*,*I*) **by** *force*+
from *cop*[*OF k f diff*] **have** *cop*: *coprime fi Fi* **.**
from *k*(*1*) *f*(*1*) **have** (*fi*,*i*) ∈ *set fs* **unfolding** *set-conv-nth* **by** *force*
from *internal-int-poly-factorization-mem*[*OF assms*(*1*) *res this*] **have** *degree*

fi > *0* **by** *auto*
hence ¬ *is-unit fi* **by** (*simp add*: *poly-dvd-1*)
with *cop coprime-id-is-unit*[*of fi*] **have** *fi* ≠ *Fi* **by** *auto*
thus *fs* ! *k* ≠ *fs* ! *K* **unfolding** *f* **by** *auto*
qed
show *f* = *smult c* (∏(*a*, *i*)∈*set fs*. *a* ^ *Suc i*) **unfolding** *eq*
prod.distinct-set-conv-list[*OF dist*] **by** *simp*
fix *fi i Fi I*
assume *mem*: (*fi*, *i*) ∈ *set fs* (*Fi*,*I*) ∈ *set fs* **and** *diff*: (*fi*, *i*) ≠ (*Fi*, *I*)
then obtain *k K* **where** *k*: *k* < *length fs K* < *length fs*
and *f*: *fs* ! *k* = (*fi*, *i*) *fs* ! *K* = (*Fi*, *I*) **unfolding** *set-conv-nth* **by** *auto*
with *diff* **have** *diff*: *k* ≠ *K* **by** *auto*
from *cop*[*OF k f diff*] **show** *Rings.coprime fi Fi* **by** *auto*
qed
qed

lemma *factorize-int-last-nz-poly*: **assumes** *res*: *factorize-int-last-nz-poly f* = (*c*,*fs*)
and *nz*: *coeff f 0* ≠ *0*
shows *square-free-factorization f* (*c*,*fs*)
(*fi*,*i*) ∈ *set fs* ⟹ *irreducible fi*
(*fi*,*i*) ∈ *set fs* ⟹ *degree fi* ≠ *0*
proof (*atomize*(*full*))
from *nz* **have** *lz*: *lead-coeff f* ≠ *0* **by** *auto*
note *res* = *res*[*unfolded factorize-int-last-nz-poly-def Let-def*]
consider (*0*) *degree f* = *0*
| (*1*) *degree f* = *1*
| (*2*) *degree f* > *1* **by** *linarith*
then show *square-free-factorization f* (*c*,*fs*) ∧ ((*fi*,*i*) ∈ *set fs* ⟶ *irreducible fi*)
∧ ((*fi*,*i*) ∈ *set fs* ⟶ *degree fi* ≠ *0*)
proof *cases*
case *0*
from *degree0-coeffs*[*OF 0*] **obtain** *a* **where** *f*: *f* = [:*a*:] **by** *auto*
from *res* **show** *?thesis* **unfolding** *square-free-factorization-def f* **by** *auto*
next
case *1*
then have *irr*: *irreducible* (*primitive-part f*)
by (*auto intro*!: *linear-primitive-irreducible content-primitive-part*)
from *irreducible-imp-square-free*[*OF irr*] **have** *sf*: *square-free* (*primitive-part f*)
.
from *1* **have** *f0*: *f* ≠ *0* **by** *auto*
from *res irr sf f0* **show** *?thesis* **unfolding** *square-free-factorization-def* **by**
(*auto simp*: *1*)
next
case *2*
with *res* **have** *internal-int-poly-factorization f* = (*c*,*fs*) **by** *auto*
from *internal-int-poly-factorization*[*OF nz this*] *internal-int-poly-factorization-mem*[*OF*
nz this]
show *?thesis* **by** *auto*
qed

qed

lemma *factorize-int-poly*: **assumes** *res*: *factorize-int-poly-generic f* = (*c*,*fs*)
shows *square-free-factorization f* (*c*,*fs*)
  (*fi*,*i*) ∈ *set fs* ⟹ *irreducible fi*
  (*fi*,*i*) ∈ *set fs* ⟹ *degree fi* ≠ *0*
proof (*atomize*(*full*))
  obtain *n g* **where** *xs*: *x-split f* = (*n*,*g*) **by** *force*
  obtain *d hs* **where** *fact*: *factorize-int-last-nz-poly g* = (*d*,*hs*) **by** *force*
  from *res*[*unfolded factorize-int-poly-generic-def xs split fact*]
  have *res*: (*if g* = *0 then* (*0*, []) *else if n* = *0 then* (*d*, *hs*) *else* (*d*, (*monom 1 1*, *n* − *1*) # *hs*)) = (*c*, *fs*) **.**
  note *xs* = *x-split*[*OF xs*]
  show *square-free-factorization f* (*c*,*fs*) ∧ ((*fi*,*i*) ∈ *set fs* ⟶ *irreducible fi*) ∧ ((*fi*,*i*) ∈ *set fs* ⟶ *degree fi* ≠ *0*)
  proof (*cases g* = *0*)
    case *True*
    hence *f* = *0 c* = *0 fs* = [] **using** *res xs* **by** *auto*
    thus *?thesis* **unfolding** *square-free-factorization-def* **by** *auto*
  next
    case *False*
    with *xs* **have** ¬ *monom 1 1 dvd g* **by** *auto*
    hence *coeff g 0* ≠ *0* **by** (*simp add*: *monom-1-dvd-iff'*)
    note *fact* = *factorize-int-last-nz-poly*[*OF fact this*]
    let *?x* = *monom 1 1* :: *int poly*
    have *x*: *content ?x* = *1* ∧ *lead-coeff ?x* = *1* ∧ *degree ?x* = *1*
      by (*auto simp add*: *degree-monom-eq monom-Suc content-def*)
    from *res False* **have** *res*: (*if n* = *0 then* (*d*, *hs*) *else* (*d*, (*?x*, *n* − *1*) # *hs*)) = (*c*, *fs*) **by** *auto*
    show *?thesis*
    proof (*cases n*)
      case *0*
      with *res xs* **have** *id*: *fs* = *hs c* = *d f* = *g* **by** *auto*
      from *fact* **show** *?thesis* **unfolding** *id* **by** *auto*
    next
      case (*Suc m*)
      with *res* **have** *id*: *c* = *d fs* = (*?x*,*m*) # *hs* **by** *auto*
      from *Suc xs* **have** *fg*: *f* = *monom 1* (*Suc m*) ∗ *g* **and** *dvd*: ¬ *?x dvd g* **by** *auto*
      from *x linear-primitive-irreducible*[*of ?x*] **have** *irr*: *irreducible ?x* **by** *auto*
      from *irreducible-imp-square-free*[*OF this*] **have** *sfx*: *square-free ?x* **.**
      from *irr fact* **have** *one*: (*fi*, *i*) ∈ *set fs* ⟶ *irreducible fi* ∧ *degree fi* ≠ *0* **unfolding** *id* **by** (*auto simp*: *degree-monom-eq*)
      have *fg*: *f* = *?x* ^ *n* ∗ *g* **unfolding** *fg Suc* **by** (*metis x-pow-n*)
      from *x* **have** *degx*: *degree ?x* = *1* **by** *simp*
      note *sf* = *square-free-factorizationD*[*OF fact*(*1*)]
      {
        fix *a i*
        assume *ai*: (*a*,*i*) ∈ *set hs*

with *sf*(*4*) **have** *g0*: *g* ≠ *0* **by** *auto*
from *split-list*[*OF ai*] **obtain** *ys zs* **where** *hs*: *hs* = *ys* @ (*a*,*i*) # *zs* **by** *auto*
have *a dvd g* **unfolding** *square-free-factorization-prod-list*[*OF fact*(*1*)] *hs*
by (*rule dvd-smult*, *simp add*: *ac-simps*)
moreover have ¬ *?x dvd g* **using** *xs*[*unfolded Suc*] **by** *auto*
ultimately have *dvd*: ¬ *?x dvd a* **using** *dvd-trans* **by** *blast*
from *sf*(*2*)[*OF ai*] **have** *a* ≠ *0* **by** *auto*
have *1* = *gcd ?x a*
proof (*rule gcdI*)
fix *d*
assume *d*: *d dvd ?x d dvd a*
from *content-dvd-contentI*[*OF d*(*1*)] *x* **have** *cnt*: *is-unit* (*content d*) **by** *auto*
show *is-unit d*
proof (*cases degree d* = *1*)
case *False*
with *divides-degree*[*OF d*(*1*), *unfolded degx*] **have** *degree d* = *0* **by** *auto*
from *degree0-coeffs*[*OF this*] **obtain** *c* **where** *dc*: *d* = [:*c*:] **by** *auto*
from *cnt*[*unfolded dc*] **have** *is-unit c* **by** (*auto simp*: *content-def*, *cases c* = *0*, *auto*)
hence *d* ∗ *d* = *1* **unfolding** *dc* **by** (*auto*, *cases c* = −*1*; *cases c* = *1*, *auto*)
thus *is-unit d* **by** (*metis dvd-triv-right*)
next
case *True*
from *d*(*1*) **obtain** *e* **where** *xde*: *?x* = *d* ∗ *e* **unfolding** *dvd-def* **by** *auto*
from *arg-cong*[*OF this*, *of degree*] *degx* **have** *degree d* + *degree e* = *1*
by (*metis True add.right-neutral degree-0 degree-mult-eq one-neq-zero*)
with *True* **have** *degree e* = *0* **by** *auto*
from *degree0-coeffs*[*OF this*] *xde* **obtain** *e* **where** *xde*: *?x* = [:*e*:] ∗ *d* **by** *auto*
from *arg-cong*[*OF this*, *of content*, *unfolded content-mult*] *x*
have *content* [:*e*:] ∗ *content d* = *1* **by** *auto*
also have *content* [:*e* :] = *abs e* **by** (*auto simp*: *content-def*, *cases e* = *0*, *auto*)
finally have |*e*| ∗ *content d* = *1* **.**
from *pos-zmult-eq-1-iff-lemma*[*OF this*] **have** *e* ∗ *e* = *1* **by** (*cases e* = *1*; *cases e* = −*1*, *auto*)
with *arg-cong*[*OF xde*, *of smult e*] **have** *d* = *?x* ∗ [:*e*:] **by** *auto*
hence *?x dvd d* **unfolding** *dvd-def* **by** *blast*
with *d*(*2*) **have** *?x dvd a* **by** (*metis dvd-trans*)
with *dvd* **show** *?thesis* **by** *auto*
qed
qed *auto*
hence *coprime ?x a*
by (*simp add*: *gcd-eq-1-imp-coprime*)
note *this dvd*
} **note** *hs-dvd-x* = *this*
from *hs-dvd-x*[*of ?x m*]

```
      have nmem: (?x,m) ∉ set hs by auto
      hence eq: ?x ^ n * g = smult d (∏(a, i)∈set fs. a ^ Suc i)
        unfolding sf(1) unfolding id Suc by simp
      have eq0: ?x ^ n * g = 0 ⟷ g = 0 by simp
    have square-free-factorization f (d,fs) unfolding fg id(1) square-free-factorization-def
split eq0 unfolding eq
      proof (intro conjI allI impI, rule refl)
        fix a i
        assume ai: (a,i) ∈ set fs
        thus square-free a degree a > 0 using sf(2) sfx degx unfolding id by auto
        fix b j
        assume bj: (b,j) ∈ set fs and diff: (a,i) ≠ (b,j)
        consider (hs-hs) (a,i) ∈ set hs (b,j) ∈ set hs
          | (hs-x) (a,i) ∈ set hs b = ?x
          | (x-hs) (b,j) ∈ set hs a = ?x
          using ai bj diff unfolding id by auto
        thus Rings.coprime a b
        proof cases
          case hs-hs
          from sf(3)[OF this diff] show ?thesis .
        next
          case hs-x
          from hs-dvd-x(1)[OF hs-x(1)] show ?thesis unfolding hs-x(2)
            by (simp add: ac-simps)
        next
          case x-hs
          from hs-dvd-x(1)[OF x-hs(1)] show ?thesis unfolding x-hs(2)
            by simp
        qed
      next
        show g = 0 ⟹ d = 0 using sf(4) by auto
        show g = 0 ⟹ fs = [] using sf(4) xs Suc by auto
        show distinct fs using sf(5) nmem unfolding id by auto
      qed
      thus ?thesis using one unfolding id by auto
    qed
  qed
qed
end

lift-definition berlekamp-zassenhaus-factorization-algorithm :: int-poly-factorization-algorithm
  is berlekamp-zassenhaus-factorization
  using berlekamp-zassenhaus-factorization-irreducible_d by blast

abbreviation factorize-int-poly where
 factorize-int-poly ≡ factorize-int-poly-generic berlekamp-zassenhaus-factorization-algorithm

end
```

11.4 Factoring Rational Polynomials

We combine the factorization algorithm for integer polynomials with Gauss Lemma to a factorization algorithm for rational polynomials.

theory *Factorize-Rat-Poly*
imports
  Factorize-Int-Poly
begin

interpretation *content-hom*: *monoid-mult-hom*
  content::$'a$::{*factorial-semiring*, *semiring-gcd*, *normalization-semidom-multiplicative*}
poly ⇒ -
by (*unfold-locales*, *auto simp*: *content-mult*)

lemma *prod-dvd-1-imp-all-dvd-1*:
  assumes *finite X* **and** *prod f X dvd 1* **and** $x \in X$ **shows** *f x dvd 1*
proof (*insert assms*, *induct rule*:*finite-induct*)
  case *IH*: (*insert* x' *X*)
  show *?case*
  proof (*cases* $x = x'$)
    case *True*
    with *IH* **show** *?thesis* **using** *dvd-trans*[*of f* x' *f* x' ∗ - *1*]
      by (*metis dvd-triv-left prod.insert*)
  next
    case *False*
    then show *?thesis* **using** *IH* **by** (*auto intro*!: *IH*(*3*) *dvd-trans*[*of prod f X* - ∗
prod f X 1])
  qed
qed *simp*

context
  fixes *alg* :: *int-poly-factorization-algorithm*
begin
definition *factorize-rat-poly-generic* :: *rat poly* ⇒ *rat* × (*rat poly* × *nat*) *list* **where**
  factorize-rat-poly-generic f = (*case rat-to-normalized-int-poly f of*
    (*c*,*g*) ⇒ *case factorize-int-poly-generic alg g of* (*d*,*fs*) ⇒ (*c* ∗ *rat-of-int d*,
    map (λ (*fi*,*i*). (*map-poly rat-of-int fi*, *i*)) *fs*))

lemma *factorize-rat-poly-0*[*simp*]: *factorize-rat-poly-generic 0* = (*0*,[])
  unfolding *factorize-rat-poly-generic-def rat-to-normalized-int-poly-def* **by** *simp*

lemma *factorize-rat-poly*:
  assumes *res*: *factorize-rat-poly-generic f* = (*c*,*fs*)
  shows *square-free-factorization f* (*c*,*fs*)
    and (*fi*,*i*) ∈ *set fs* ⟹ *irreducible fi*
proof(*atomize*(*full*), *cases f*=*0*, *goal-cases*)
  case *1* **with** *res* **show** *?case* **by** (*auto simp*: *square-free-factorization-def*)
next

case *2* **show** *?case*
proof (*unfold square-free-factorization-def split*, *intro conjI impI allI*)
let *?r* = *rat-of-int*
let *?rp* = *map-poly ?r*
obtain *d g* **where** *ri*: *rat-to-normalized-int-poly f* = (*d*,*g*) **by** *force*
obtain *e gs* **where** *fi*: *factorize-int-poly-generic alg g* = (*e*,*gs*) **by** *force*
from *res*[*unfolded factorize-rat-poly-generic-def ri fi split*]
have *c*: *c* = *d* ∗ *?r e* **and** *fs*: *fs* = *map* (λ (*fi*,*i*). (*?rp fi*, *i*)) *gs* **by** *auto*
from *factorize-int-poly*[*OF fi*]
have *irr*: (*fi*, *i*) ∈ *set gs* ⟹ *irreducible fi* ∧ *content fi* = *1* **for** *fi i*
using *irreducible-imp-primitive*[*of fi*] **by** *auto*
note *sff* = *factorize-int-poly*(*1*)[*OF fi*]
note *sff'* = *square-free-factorizationD*[*OF sff*]
{
fix *n f*
have *?rp* (*f* ^ *n*) = (*?rp f*) ^ *n*
by (*induct n*, *auto simp*: *hom-distribs*)
} **note** *exp* = *this*
show *dist*: *distinct fs* **using** *sff'*(*5*) **unfolding** *fs distinct-map inj-on-def* **by** *auto*
interpret *mh*: *map-poly-inj-idom-hom rat-of-int*..
have *f* = *smult d* (*?rp g*) **using** *rat-to-normalized-int-poly*[*OF ri*] **by** *auto*
also have ... = *smult d* (*?rp* (*smult e* (∏(*a*, *i*)∈*set gs*. *a* ^ *Suc i*))) **using** *sff'*(*1*) **by** *simp*
also have ... = *smult c* (*?rp* (∏(*a*, *i*)∈*set gs*. *a* ^ *Suc i*)) **unfolding** *c* **by** (*simp add*: *hom-distribs*)
also have *?rp* (∏(*a*, *i*)∈*set gs*. *a* ^ *Suc i*) = (∏(*a*, *i*)∈*set fs*. *a* ^ *Suc i*)
unfolding *prod.distinct-set-conv-list*[*OF sff'*(*5*)] *prod.distinct-set-conv-list*[*OF dist*]
unfolding *fs*
by (*insert exp*, *auto intro*!: *arg-cong*[*of* - - λ*x*. *prod-list* (*map x gs*)] *simp*: *hom-distribs of-int-poly-hom.hom-prod-list*)
finally show *f*: *f* = *smult c* (∏(*a*, *i*)∈*set fs*. *a* ^ *Suc i*) **by** *auto*
{
fix *a i*
assume *ai*: (*a*,*i*) ∈ *set fs*
from *ai* **obtain** *A* **where** *a*: *a* = *?rp A* **and** *A*: (*A*,*i*) ∈ *set gs* **unfolding** *fs* **by** *auto*
fix *b j*
assume (*b*,*j*) ∈ *set fs* **and** *diff*: (*a*,*i*) ≠ (*b*,*j*)
from *this*(*1*) **obtain** *B* **where** *b*: *b* = *?rp B* **and** *B*: (*B*,*j*) ∈ *set gs* **unfolding** *fs* **by** *auto*
from *diff*[*unfolded a b*] **have** (*A*,*i*) ≠ (*B*,*j*) **by** *auto*
from *sff'*(*3*)[*OF A B this*]
show *Rings.coprime a b*
by (*auto simp add*: *coprime-iff-gcd-eq-1 gcd-rat-to-gcd-int a b*)
}
{
fix *fi i*

assume (*fi*,*i*) ∈ *set fs*
then obtain *gi* **where** *fi*: *fi* = *?rp gi* **and** *gi*: (*gi*,*i*) ∈ *set gs* **unfolding** *fs*
by *auto*
from *irr*[*OF gi*] **have** *cf-gi*: *primitive gi* **by** *auto*
then have *primitive* (*?rp gi*) **by** (*auto simp*: *content-field-poly*)
note [*simp*] = *irreducible-primitive-connect*[*OF cf-gi*] *irreducible-primitive-connect*[*OF this*]
show *irreducible fi*
using *irr*[*OF gi*] *fi* $irreducible_d$*-int-rat*[*of gi*,*simplified*] **by** *auto*
then show *degree fi* > *0 square-free fi* **unfolding** *fi*
by (*auto intro*: *irreducible-imp-square-free*)
}
{
assume *f* = *0* **with** *ri* **have** ∗: *d* = *1 g* = *0* **unfolding** *rat-to-normalized-int-poly-def*
by *auto*
with *sff'*(*4*)[*OF* ∗(*2*)] **show** *c* = *0 fs* = [] **unfolding** *c fs* **by** *auto*
}
qed
qed

end

abbreviation *factorize-rat-poly* **where**
factorize-rat-poly ≡ *factorize-rat-poly-generic berlekamp-zassenhaus-factorization-algorithm*

end

References

[1] G. Barthe, B. Grégoire, S. Heraud, F. Olmedo, and S. Z. Béguelin. Verified indifferentiable hashing into elliptic curves. In *POST 2012*, volume 7215 of *LNCS*, pages 209–228, 2012.

[2] E. R. Berlekamp. Factoring polynomials over finite fields. *Bell System Technical Journal*, 46:1853–1859, 1967.

[3] D. G. Cantor and H. Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Math. Comput.*, 36(154):587–592, 1981.

[4] J. R. Cowles and R. Gamboa. Unique factorization in ACL2: Euclidean domains. In *Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 21–27. ACM, 2006.

[5] B. Huffman and O. Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *CPP 2013*, volume 8307 of *LNCS*, pages 131–146, 2013.

[6] B. Kirkels. *Irreducibility Certificates for Polynomials with Integer Coefficients*. PhD thesis, Radboud Universiteit Nijmegen, 2004.

[7] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981.

[8] H. Kobayashi, H. Suzuki, and Y. Ono. Formalization of Hensel's lemma. In *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*, volume 1, pages 114–118, 2005.

[9] O. Kunčar and A. Popescu. From types to sets by local type definitions in higher-order logic. In *ITP 2016*, volume 9807 of *LNCS*, pages 200–218, 2016.

[10] É. Martin-Dorel, G. Hanrot, M. Mayero, and L. Théry. Formally verified certificate checkers for hardest-to-round computation. *Journal of Automated Reasoning*, 54(1):1–29, 2015.

[11] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[12] R. Thiemann and A. Yamada. Algebraic numbers in Isabelle/HOL. In *ITP 2016*, volume 9807 of *LNCS*, pages 391–408, 2016.

[13] H. Zassenhaus. On Hensel factorization, I. *Journal of Number Theory*, 1(3):291–311, 1969.