The Factorization Algorithm of Berlekamp and Zassenhaus *

Jose Divasón Sebastiaan Joosten René Thiemann
Akihisa Yamada

March 19, 2025

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* ⇒ *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.

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

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

To obtain Theorem 1 we perform the following tasks.

- We introduce two formulations of 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 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 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*)} ⟨*proof*⟩

setup-lifting *type-definition-mod-ring*

lemma *CARD-mod-ring*[*simp*]: *CARD*(*'a mod-ring*) = *CARD*(*'a*::*finite*)
⟨*proof*⟩

instance *mod-ring* :: (*finite*) *finite*
⟨*proof*⟩

instantiation *mod-ring* :: (*finite*) *equal*
begin
lift-definition *equal-mod-ring* :: *'a mod-ring* ⇒ *'a mod-ring* ⇒ *bool* **is** (=) ⟨*proof*⟩
instance ⟨*proof*⟩
end

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

lift-definition *plus-mod-ring* :: *'a mod-ring* ⇒ *'a mod-ring* ⇒ *'a mod-ring* **is**
λ *x y*. (*x* + *y*) *mod int* (*CARD*(*'a*)) ⟨*proof*⟩

lift-definition *uminus-mod-ring* :: *'a mod-ring* ⇒ *'a mod-ring* **is**
λ *x*. *if x* = *0 then 0 else int* (*CARD*(*'a*)) − *x* ⟨*proof*⟩

lift-definition *minus-mod-ring* :: *'a mod-ring* ⇒ *'a mod-ring* ⇒ *'a mod-ring* **is**
λ *x y*. (*x* − *y*) *mod int* (*CARD*(*'a*)) ⟨*proof*⟩

lift-definition *times-mod-ring* :: *'a mod-ring* ⇒ *'a mod-ring* ⇒ *'a mod-ring* **is**
λ *x y*. (*x* ∗ *y*) *mod int* (*CARD*(*'a*)) ⟨*proof*⟩

lift-definition *zero-mod-ring* :: *'a mod-ring* **is** *0* ⟨*proof*⟩

instance
⟨*proof*⟩

end

lift-definition *to-int-mod-ring* :: *'a*::*finite mod-ring* ⇒ *int* **is** λ *x*. *x* ⟨*proof*⟩

lift-definition *of-int-mod-ring* :: *int* ⇒ *'a*::*finite mod-ring* **is**
λ *x*. *x mod int* (*CARD*(*'a*)) ⟨*proof*⟩

interpretation *to-int-mod-ring-hom*: *inj-zero-hom to-int-mod-ring*
⟨*proof*⟩

lemma *int-nat-card*[*simp*]: *int* (*nat CARD*(*'a*::*finite*)) = *CARD*(*'a*) ⟨*proof*⟩

interpretation *of-int-mod-ring-hom*: *zero-hom of-int-mod-ring*
⟨*proof*⟩

lemma *of-int-mod-ring-to-int-mod-ring*[*simp*]:
of-int-mod-ring (*to-int-mod-ring x*) = *x* ⟨*proof*⟩

lemma *to-int-mod-ring-of-int-mod-ring*[*simp*]: *0* ≤ *x* ⟹ *x* < *int CARD*(*'a* :: *finite*) ⟹
to-int-mod-ring (*of-int-mod-ring x* :: *'a mod-ring*) = *x*
⟨*proof*⟩

lemma *range-to-int-mod-ring*:
range (*to-int-mod-ring* :: (*'a* :: *finite mod-ring* ⇒ *int*)) = {*0* ..< *CARD*(*'a*)}
⟨*proof*⟩

2.2 Nontrivial Finite Rings

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

subclass(**in** *nontriv*) *finite* ⟨*proof*⟩

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

lift-definition *one-mod-ring* :: *'a mod-ring* **is** *1* ⟨*proof*⟩

instance ⟨*proof*⟩

end

interpretation *to-int-mod-ring-hom*: *inj-one-hom to-int-mod-ring*
⟨*proof*⟩

lemma *of-nat-of-int-mod-ring* [*code-unfold*]:
of-nat = *of-int-mod-ring o int*
⟨*proof*⟩

lemma *of-nat-card-eq-0*[*simp*]: (*of-nat* (*CARD*(*'a*::*nontriv*)) :: *'a mod-ring*) = *0*
⟨*proof*⟩

lemma *of-int-of-int-mod-ring*[*code-unfold*]: *of-int* = *of-int-mod-ring*
⟨*proof*⟩

unbundle *lifting-syntax*

lemma *pcr-mod-ring-to-int-mod-ring*: *pcr-mod-ring* = (λ*x y*. *x* = *to-int-mod-ring y*)
⟨*proof*⟩

lemma [*transfer-rule*]:
((=) ===> *pcr-mod-ring*) (λ *x*. *int x mod int* (*CARD*('*a* :: *nontriv*))) (*of-nat* :: *nat* ⇒ '*a mod-ring*)
⟨*proof*⟩

lemma [*transfer-rule*]:
((=) ===> *pcr-mod-ring*) (λ *x*. *x mod int* (*CARD*('*a* :: *nontriv*))) (*of-int* :: *int* ⇒ '*a mod-ring*)
⟨*proof*⟩

lemma *one-mod-card* [*simp*]: *1 mod CARD*('*a*::*nontriv*) = *1*
⟨*proof*⟩

lemma *Suc-0-mod-card* [*simp*]: *Suc 0 mod CARD*('*a*::*nontriv*) = *1*
⟨*proof*⟩

lemma *one-mod-card-int* [*simp*]: *1 mod int CARD*('*a*::*nontriv*) = *1*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *Rep-mod-ring-mod*[*simp*]: *Rep-mod-ring* (*a* :: '*a* :: *nontriv mod-ring*) *mod CARD*('*a*) = *Rep-mod-ring a*
⟨*proof*⟩

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*))) ⟨*proof*⟩

subclass *nontriv* ⟨*proof*⟩
end

instance *bool* :: *prime-card*
⟨*proof*⟩

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* ⇒ '*a mod-ring* ⇒ '*a mod-ring* **where**
divide-mod-ring x y = *x* ∗ ((λ*c*. *if c* = *0 then 0 else c* ^ (*nat* (*CARD*('*a*) − *2*)))
y)

instance
⟨*proof*⟩
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*⟩
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 ⟨*proof*⟩
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 ⟨*proof*⟩

end

instance *mod-ring* :: (*prime-card*) *field-gcd*
⟨*proof*⟩

lemma *surj-of-nat-mod-ring*: ∃ *i*. *i* < *CARD*(*'a* :: *prime-card*) ∧ (*x* :: *'a mod-ring*) = *of-nat i*
⟨*proof*⟩

lemma *of-nat-0-mod-ring-dvd*: **assumes** *x*: *of-nat x* = (*0* :: *'a* ::*prime-card mod-ring*)
shows *CARD*(*'a*) *dvd x*
⟨*proof*⟩

lemma *semiring-char-mod-ring* [*simp*]:
CHAR(*'n* :: *nontriv mod-ring*) = *CARD*(*'n*)
⟨*proof*⟩

The following Material was contributed by Manuel Eberl

instance *mod-ring* :: (*prime-card*) *finite-field*
⟨*proof*⟩

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

definition *enum-finite-field-mod-ring* :: *nat* ⇒ *'a mod-ring* **where**
enum-finite-field-mod-ring n = *of-int-mod-ring* (*int n*)

instance ⟨*proof*⟩

end

typedef (**overloaded**) *'a* :: *semiring-1 ring-char* = *if CHAR*(*'a*) = *0 then UNIV else* {*0*..<*CHAR*(*'a*)}
⟨*proof*⟩

lemma *CARD-ring-char* [*simp*]: *CARD* (*'a* :: *semiring-1 ring-char*) = *CHAR*(*'a*)
⟨*proof*⟩

instance *ring-char* :: (*semiring-prime-char*) *nontriv*
⟨*proof*⟩

instance *ring-char* :: (*semiring-prime-char*) *prime-card*
⟨*proof*⟩

lemma *to-int-mod-ring-add*:
to-int-mod-ring (*x* + *y* :: 'a :: *finite mod-ring*) = (*to-int-mod-ring x* + *to-int-mod-ring y*) *mod CARD*('a)
⟨*proof*⟩

lemma *to-int-mod-ring-mult*:
to-int-mod-ring (*x* ∗ *y* :: 'a :: *finite mod-ring*) = (*to-int-mod-ring x* ∗ *to-int-mod-ring y*) *mod CARD*('a)
⟨*proof*⟩

lemma *of-nat-mod-CHAR* [*simp*]: *of-nat* (*x mod CHAR*('a :: *semiring-1*)) = (*of-nat x* :: 'a)
⟨*proof*⟩

lemma *of-int-mod-CHAR* [*simp*]: *of-int* (*x mod int CHAR*('a :: *ring-1*)) = (*of-int x* :: 'a)
⟨*proof*⟩

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 ⇒ 'a ⇒ 'a)
(*times* : 'a ⇒ 'a ⇒ 'a)
(*minus* : 'a ⇒ 'a ⇒ 'a)
(*uminus* : 'a ⇒ 'a)
(*divide* : 'a ⇒ 'a ⇒ 'a)
(*inverse* : 'a ⇒ 'a)
(*modulo* : 'a ⇒ 'a ⇒ 'a)
(*normalize* : 'a ⇒ 'a)
(*unit-factor* : 'a ⇒ 'a)

(*of-int* : *int* ⇒ *'a*)
(*to-int* : *'a* ⇒ *int*)
(*DP* : *'a* ⇒ *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* ⇒ *′i* ⇒ (*′i* × *′i*) × *′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*
⟨*proof*⟩

lemma *listprod-i*[*transfer-rule*]: (*list-all2 R* ===> *R*) (*listprod-i ops*) *prod-list*
⟨*proof*⟩
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*⟩
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*⟩

lemma *euclid-ext-i* [*transfer-rule*]:
(*R* ===> *R* ===> *rel-prod* (*rel-prod R R*) *R*) *euclid-ext-i euclid-ext*
⟨*proof*⟩

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*⟩

lemma *strip-while-rel*[*transfer-rule*]:
((*A* ===> (=)) ===> *list-all2 A* ===> *list-all2 A*) *strip-while strip-while*
⟨*proof*⟩

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

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
HOL−Library.Code-Target-Numeral
Native-Word.Code-Target-Int-Bit
begin

definition *mod-nonneg-pos* :: *integer* ⇒ *integer* ⇒ *integer* **where**
x ≥ *0* ⟹ *y* > *0* ⟹ *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* ≥ *0* ⟹ *y* > *0* ⟹ *mod-nonneg-pos-int x y* = (*x mod y*)
⟨*proof*⟩

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*) = *Euclidean-Rings.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*)
⟨*proof*⟩

lemma *int-of-uint32-0-iff*: *int-of-uint32 n = 0 ⟷ n = 0*
⟨*proof*⟩

lemma *int-of-uint32-0*: *int-of-uint32 0 = 0* ⟨*proof*⟩

lemma *int-of-uint32-ge-0*: *int-of-uint32 n ≥ 0*
⟨*proof*⟩

lemma *two-32*: *2 ^ LENGTH(32) = (4294967296 :: int)* ⟨*proof*⟩

lemma *int-of-uint32-plus*: *int-of-uint32 (x + y) = (int-of-uint32 x + int-of-uint32 y) mod 4294967296*
⟨*proof*⟩

lemma *int-of-uint32-minus*: *int-of-uint32 (x − y) = (int-of-uint32 x − int-of-uint32 y) mod 4294967296*
⟨*proof*⟩

lemma *int-of-uint32-mult*: *int-of-uint32 (x * y) = (int-of-uint32 x * int-of-uint32 y) mod 4294967296*
⟨*proof*⟩

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

lemma *int-of-uint32-inv*: *0 ≤ x ⟹ x < 4294967296 ⟹ int-of-uint32 (uint32-of-int x) = x*
⟨*proof*⟩

context
includes *bit-operations-syntax*
begin

function *power-p32* :: *uint32 ⇒ uint32 ⇒ 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)
⟨*proof*⟩

termination
⟨*proof*⟩

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* ⇒ *uint32* **where**
inverse-p32 x = (*if x* = *0 then 0 else power-p32 x* (*p* − *2*))

definition *divide-p32* :: *uint32* ⇒ *uint32* ⇒ *uint32* **where**
divide-p32 x y = *mult-p32 x* (*inverse-p32 y*)

definition *finite-field-ops32* :: *uint32 arith-ops-record* **where**
finite-field-ops32 ≡ *Arith-Ops-Record*
0
1
plus-p32
mult-p32
minus-p32
uminus-p32
divide-p32
inverse-p32
(λ *x y* . *if y* = *0 then x else 0*)
(λ *x* . *if x* = *0 then 0 else 1*)
(λ *x* . *x*)
uint32-of-int
int-of-uint32
(λ *x*. *0* ≤ *x* ∧ *x* < *p*)
end

lemma *shiftr-uint32-code* [*code-unfold*]: *drop-bit 1 x* = (*uint32-shiftr x 1*)
⟨*proof*⟩

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*) ⟨*proof*⟩
lemma *p2*: *p* ≥ *2* ⟨*proof*⟩
lemma *p2-ident*: *int* (*CARD*(*'a*) − *2*) = *p* − *2* ⟨*proof*⟩

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*⟩

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

⟨*proof*⟩

lemma *right-total-mod-ring-rel* [*transfer-rule*]: *right-total mod-ring-rel*
⟨*proof*⟩

3.1.2 Transfer Rules

lemma *mod-ring-0*[*transfer-rule*]: *mod-ring-rel 0 0* ⟨*proof*⟩
lemma *mod-ring-1*[*transfer-rule*]: *mod-ring-rel 1 1* ⟨*proof*⟩

lemma *plus-p-mod-def*: **assumes** *x*: $x \in \{0 ..< p\}$ **and** *y*: $y \in \{0 ..< p\}$
shows *plus-p p x y* = $((x + y) \bmod p)$
⟨*proof*⟩

lemma *mod-ring-plus*[*transfer-rule*]: (*mod-ring-rel* ===> *mod-ring-rel* ===> *mod-ring-rel*) (*plus-p p*) (+)
⟨*proof*⟩

lemma *minus-p-mod-def*: **assumes** *x*: $x \in \{0 ..< p\}$ **and** *y*: $y \in \{0 ..< p\}$
shows *minus-p p x y* = $((x - y) \bmod p)$
⟨*proof*⟩

lemma *mod-ring-minus*[*transfer-rule*]: (*mod-ring-rel* ===> *mod-ring-rel* ===> *mod-ring-rel*) (*minus-p p*) (−)
⟨*proof*⟩

lemma *mod-ring-uminus*[*transfer-rule*]: (*mod-ring-rel* ===> *mod-ring-rel*) (*uminus-p p*) *uminus*
⟨*proof*⟩

lemma *mod-ring-mult*[*transfer-rule*]: (*mod-ring-rel* ===> *mod-ring-rel* ===> *mod-ring-rel*) (*mult-p p*) ((∗))
⟨*proof*⟩

lemma *mod-ring-eq*[*transfer-rule*]: (*mod-ring-rel* ===> *mod-ring-rel* ===> (=)) (=) (=)
⟨*proof*⟩

lemma *mod-ring-power*[*transfer-rule*]: (*mod-ring-rel* ===> (=) ===> *mod-ring-rel*) (*power-p p*) (^)
⟨*proof*⟩

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

lemma *ring-finite-field-ops-int*: *ring-ops* (*finite-field-ops-int p*) *mod-ring-rel*
⟨*proof*⟩
end

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

lemma *prime*: *prime p* ⟨*proof*⟩

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*⟩

lemma *mod-ring-normalize*[*transfer-rule*]: (*mod-ring-rel* ===> *mod-ring-rel*) ((λ *x*. *if x* = *0 then 0 else 1*)) *normalize*
⟨*proof*⟩

lemma *mod-ring-unit-factor*[*transfer-rule*]: (*mod-ring-rel* ===> *mod-ring-rel*) (λ *x*. *x*) *unit-factor*
⟨*proof*⟩

lemma *mod-ring-inverse*[*transfer-rule*]: (*mod-ring-rel* ===> *mod-ring-rel*) (*inverse-p p*) *inverse*
⟨*proof*⟩

lemma *mod-ring-divide*[*transfer-rule*]: (*mod-ring-rel* ===> *mod-ring-rel* ===> *mod-ring-rel*)
(*divide-p p*) (/)
⟨*proof*⟩

lemma *mod-ring-rel-unsafe*: **assumes** *x* < *CARD*(*'a*)
shows *mod-ring-rel* (*int x*) (*of-nat x*) *0* < *x* ⟹ *of-nat x* ≠ (*0* :: *'a mod-ring*)
⟨*proof*⟩

lemma *finite-field-ops-int*: *field-ops* (*finite-field-ops-int p*) *mod-ring-rel*
⟨*proof*⟩

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 appli-

cations 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* $\Rightarrow$ *int* $\Rightarrow$ *bool* **where** *urel32 x y* = (*y* = *int-of-uint32* $x \wedge y < p$)

definition *mod-ring-rel32* :: *uint32* $\Rightarrow$ *'a mod-ring* $\Rightarrow$ *bool* **where**
mod-ring-rel32 x y = ($\exists$ *z*. *urel32 x z* $\wedge$ *mod-ring-rel z y*)

lemma *urel32-0*: *urel32 0 0* ⟨*proof*⟩

lemma *urel32-1*: *urel32 1 1* ⟨*proof*⟩

lemma *le-int-of-uint32*: ($x \leq y$) = (*int-of-uint32 x* $\leq$ *int-of-uint32 y*)
⟨*proof*⟩

lemma *urel32-plus*: **assumes** *urel32 x y urel32 x' y'*
shows *urel32* (*plus-p32 pp x x'*) (*plus-p p y y'*)
⟨*proof*⟩

lemma *urel32-minus*: **assumes** *urel32 x y urel32 x' y'*
shows *urel32* (*minus-p32 pp x x'*) (*minus-p p y y'*)
⟨*proof*⟩

lemma *urel32-uminus*: **assumes** *urel32 x y*
shows *urel32* (*uminus-p32 pp x*) (*uminus-p p y*)
⟨*proof*⟩

lemma *urel32-mult*: **assumes** *urel32 x y urel32 x' y'*
shows *urel32* (*mult-p32 pp x x'*) (*mult-p p y y'*)
⟨*proof*⟩

lemma *urel32-eq*: **assumes** *urel32 x y urel32 x′ y′*
shows $(x = x') = (y = y')$
⟨*proof*⟩

lemma *urel32-normalize*:
assumes *x*: *urel32 x y*
shows *urel32* (*if x = 0 then 0 else 1*) (*if y = 0 then 0 else 1*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *urel32-inverse*: **assumes** *x*: *urel32 x x′*
shows *urel32* (*inverse-p32 pp x*) (*inverse-p p x′*)
⟨*proof*⟩

lemma *mod-ring-0-32*: *mod-ring-rel32 0 0*
⟨*proof*⟩

lemma *mod-ring-1-32*: *mod-ring-rel32 1 1*
⟨*proof*⟩

lemma *mod-ring-uminus32*: (*mod-ring-rel32* ===> *mod-ring-rel32*) (*uminus-p32 pp*) *uminus*
⟨*proof*⟩

lemma *mod-ring-plus32*: (*mod-ring-rel32* ===> *mod-ring-rel32* ===> *mod-ring-rel32*) (*plus-p32 pp*) (+)
⟨*proof*⟩

lemma *mod-ring-minus32*: (*mod-ring-rel32* ===> *mod-ring-rel32* ===> *mod-ring-rel32*) (*minus-p32 pp*) (−)
⟨*proof*⟩

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

lemma *mod-ring-eq32*: (*mod-ring-rel32* ===> *mod-ring-rel32* ===> (=)) (=) (=)
⟨*proof*⟩

lemma *urel32-inj*: *urel32 x y* $\Longrightarrow$ *urel32 x z* $\Longrightarrow$ $y = z$
⟨*proof*⟩

lemma *urel32-inj'*: *urel32 x z* $\Longrightarrow$ *urel32 y z* $\Longrightarrow$ $x = y$
⟨*proof*⟩

lemma *bi-unique-mod-ring-rel32*:
bi-unique mod-ring-rel32 left-unique mod-ring-rel32 right-unique mod-ring-rel32
⟨*proof*⟩

lemma *right-total-mod-ring-rel32*: *right-total mod-ring-rel32*
⟨*proof*⟩

lemma *Domainp-mod-ring-rel32*: *Domainp mod-ring-rel32* = (λ*x*. *0* ≤ *x* ∧ *x* < *pp*)
⟨*proof*⟩

lemma *ring-finite-field-ops32*: *ring-ops* (*finite-field-ops32 pp*) *mod-ring-rel32*
⟨*proof*⟩
end
end

context *prime-field*
begin
context fixes *pp* :: *uint32*
assumes ∗: *p* = *int-of-uint32 pp p* ≤ *65535*
begin

lemma *mod-ring-normalize32*: (*mod-ring-rel32* ===> *mod-ring-rel32*) (λ*x*. *if x = 0 then 0 else 1*) *normalize*
⟨*proof*⟩

lemma *mod-ring-mod32*: (*mod-ring-rel32* ===> *mod-ring-rel32* ===> *mod-ring-rel32*) (λ*x y*. *if y = 0 then x else 0*) (*mod*)
⟨*proof*⟩

lemma *mod-ring-unit-factor32*: (*mod-ring-rel32* ===> *mod-ring-rel32*) (λ*x*. *x*) *unit-factor*
⟨*proof*⟩

lemma *mod-ring-inverse32*: (*mod-ring-rel32* ===> *mod-ring-rel32*) (*inverse-p32 pp*) *inverse*
⟨*proof*⟩

lemma *mod-ring-divide32*: (*mod-ring-rel32* ===> *mod-ring-rel32* ===> *mod-ring-rel32*) (*divide-p32 pp*) (/)
⟨*proof*⟩

lemma *finite-field-ops32*: *field-ops* (*finite-field-ops32 pp*) *mod-ring-rel32*

⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *int-of-uint64-0-iff*: *int-of-uint64 n* = *0* ⟷ *n* = *0*
⟨*proof*⟩

lemma *int-of-uint64-0*: *int-of-uint64 0* = *0* ⟨*proof*⟩

lemma *int-of-uint64-ge-0*: *int-of-uint64 n* ≥ *0*
⟨*proof*⟩

lemma *two-64*: *2* ^ *LENGTH*(*64*) = (*18446744073709551616* :: *int*) ⟨*proof*⟩

lemma *int-of-uint64-plus*: *int-of-uint64* (*x* + *y*) = (*int-of-uint64 x* + *int-of-uint64 y*) *mod 18446744073709551616*
⟨*proof*⟩

lemma *int-of-uint64-minus*: *int-of-uint64* (*x* − *y*) = (*int-of-uint64 x* − *int-of-uint64 y*) *mod 18446744073709551616*
⟨*proof*⟩

lemma *int-of-uint64-mult*: *int-of-uint64* (*x* ∗ *y*) = (*int-of-uint64 x* ∗ *int-of-uint64 y*) *mod 18446744073709551616*
⟨*proof*⟩

lemma *int-of-uint64-mod*: *int-of-uint64* (*x mod y*) = (*int-of-uint64 x mod int-of-uint64 y*)

⟨*proof*⟩

lemma *int-of-uint64-inv*: $0 \leq x \Longrightarrow x < 18446744073709551616 \Longrightarrow$ *int-of-uint64* (*uint64-of-int* x) $= x$
⟨*proof*⟩

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*)
⟨*proof*⟩

termination
⟨*proof*⟩

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* ⇒ *uint64* **where**
inverse-p64 x = (*if x* = *0 then 0 else power-p64 x* (*p* − *2*))

definition *divide-p64* :: *uint64* ⇒ *uint64* ⇒ *uint64* **where**
divide-p64 x y = *mult-p64 x* (*inverse-p64 y*)

definition *finite-field-ops64* :: *uint64 arith-ops-record* **where**
finite-field-ops64 ≡ *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*)
⟨*proof*⟩

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* ≤ *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* ⟨*proof*⟩

lemma *urel64-1*: *urel64 1 1* ⟨*proof*⟩

lemma *le-int-of-uint64*: (*x* ≤ *y*) = (*int-of-uint64 x* ≤ *int-of-uint64 y*)
⟨*proof*⟩

lemma *urel64-plus*: **assumes** *urel64 x y urel64 x' y'*
shows *urel64* (*plus-p64 pp x x'*) (*plus-p p y y'*)
⟨*proof*⟩

lemma *urel64-minus*: **assumes** *urel64 x y urel64 x' y'*
shows *urel64* (*minus-p64 pp x x'*) (*minus-p p y y'*)
⟨*proof*⟩

lemma *urel64-uminus*: **assumes** *urel64 x y*
shows *urel64* (*uminus-p64 pp x*) (*uminus-p p y*)
⟨*proof*⟩

lemma *urel64-mult*: **assumes** *urel64 x y urel64 x' y'*
shows *urel64* (*mult-p64 pp x x'*) (*mult-p p y y'*)
⟨*proof*⟩

lemma *urel64-eq*: **assumes** *urel64 x y urel64 x' y'*
shows $(x = x') = (y = y')$
⟨*proof*⟩

lemma *urel64-normalize*:
assumes *x*: *urel64 x y*
shows *urel64* (*if x = 0 then 0 else 1*) (*if y = 0 then 0 else 1*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *urel64-power*: *urel64 x x'* ⟹ *urel64 y* (*int y'*) ⟹ *urel64* (*power-p64 pp x y*) (*power-p p x' y'*)
including *bit-operations-syntax* ⟨*proof*⟩

lemma *urel64-inverse*: **assumes** *x*: *urel64 x x'*
shows *urel64* (*inverse-p64 pp x*) (*inverse-p p x'*)
⟨*proof*⟩

lemma *mod-ring-0-64*: *mod-ring-rel64 0 0*
⟨*proof*⟩

lemma *mod-ring-1-64*: *mod-ring-rel64 1 1*
⟨*proof*⟩

lemma *mod-ring-uminus64*: (*mod-ring-rel64* ===> *mod-ring-rel64*) (*uminus-p64 pp*) *uminus*
⟨*proof*⟩

lemma *mod-ring-plus64*: (*mod-ring-rel64* ===> *mod-ring-rel64* ===> *mod-ring-rel64*) (*plus-p64 pp*) (+)
⟨*proof*⟩

lemma *mod-ring-minus64*: (*mod-ring-rel64* ===> *mod-ring-rel64* ===> *mod-ring-rel64*) (*minus-p64 pp*) (−)
⟨*proof*⟩

lemma *mod-ring-mult64*: (*mod-ring-rel64* ===> *mod-ring-rel64* ===> *mod-ring-rel64*) (*mult-p64 pp*) ((∗))
⟨*proof*⟩

lemma *mod-ring-eq64*: (*mod-ring-rel64* ===> *mod-ring-rel64* ===> (=)) (=) (=)
⟨*proof*⟩

lemma *urel64-inj*: *urel64 x y* $\Longrightarrow$ *urel64 x z* $\Longrightarrow$ *y = z*
⟨*proof*⟩

lemma *urel64-inj'*: *urel64 x z* $\Longrightarrow$ *urel64 y z* $\Longrightarrow$ *x = y*
⟨*proof*⟩

lemma *bi-unique-mod-ring-rel64*:
bi-unique mod-ring-rel64 left-unique mod-ring-rel64 right-unique mod-ring-rel64
⟨*proof*⟩

lemma *right-total-mod-ring-rel64*: *right-total mod-ring-rel64*
⟨*proof*⟩

lemma *Domainp-mod-ring-rel64*: *Domainp mod-ring-rel64* = (λ*x*. *0* ≤ *x* ∧ *x* < *pp*)
⟨*proof*⟩

lemma *ring-finite-field-ops64*: *ring-ops* (*finite-field-ops64 pp*) *mod-ring-rel64*
⟨*proof*⟩
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*
⟨*proof*⟩

lemma *mod-ring-mod64*: (*mod-ring-rel64* ===> *mod-ring-rel64* ===> *mod-ring-rel64*) (λ*x y*. *if y = 0 then x else 0*) (*mod*)
⟨*proof*⟩

lemma *mod-ring-unit-factor64*: (*mod-ring-rel64* ===> *mod-ring-rel64*) (λ*x*. *x*) *unit-factor*
⟨*proof*⟩

lemma *mod-ring-inverse64*: (*mod-ring-rel64* ===> *mod-ring-rel64*) (*inverse-p64 pp*) *inverse*
⟨*proof*⟩

lemma *mod-ring-divide64*: (*mod-ring-rel64* ===> *mod-ring-rel64* ===> *mod-ring-rel64*) (*divide-p64 pp*) (/)
⟨*proof*⟩

lemma *finite-field-ops64*: *field-ops* (*finite-field-ops64 pp*) *mod-ring-rel64*

⟨*proof*⟩
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*)

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*)
⟨*proof*⟩

termination
⟨*proof*⟩
include *integer.lifting*
⟨*proof*⟩

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* ≤ *x* ∧ *x* < *p*)
end

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

lemma *integer-simps*:
‹*int-of-integer 0* = *0*›
‹*int-of-integer* (*x* + *y*) = *int-of-integer x* + *int-of-integer y*›
‹*int-of-integer* (*x* − *y*) = *int-of-integer x* − *int-of-integer y*›
‹*int-of-integer* (*x* ∗ *y*) = *int-of-integer x* ∗ *int-of-integer y*›
⟨*proof*⟩

definition *urel-integer* :: *integer* ⇒ *int* ⇒ *bool* **where** *urel-integer x y* = (*y* = *int-of-integer x* ∧ *y* ≥ *0* ∧ *y* < *p*)

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

lemma *urel-integer-0*: *urel-integer 0 0* ⟨*proof*⟩

lemma *urel-integer-1*: *urel-integer 1 1* ⟨*proof*⟩

lemma *le-int-of-integer*: (*x* ≤ *y*) = (*int-of-integer x* ≤ *int-of-integer y*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *urel-integer-uminus*: **assumes** *urel-integer x y*
shows *urel-integer* (*uminus-p-integer pp x*) (*uminus-p p y*)
⟨*proof*⟩
include *integer.lifting*
⟨*proof*⟩

lemma *pp-pos*: *int-of-integer pp* > *0*
⟨*proof*⟩

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*⟩

lemma *urel-integer-eq*: **assumes** *urel-integer x y urel-integer x′ y′*
shows (*x* = *x′*) = (*y* = *y′*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *urel-integer-inverse*: **assumes** *x*: *urel-integer x x′*
shows *urel-integer* (*inverse-p-integer pp x*) (*inverse-p p x′*)
⟨*proof*⟩

lemma *mod-ring-0--integer*: *mod-ring-rel-integer 0 0*
⟨*proof*⟩

lemma *mod-ring-1--integer*: *mod-ring-rel-integer 1 1*
⟨*proof*⟩

lemma *mod-ring-uminus-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer*)
(*uminus-p-integer pp*) *uminus*

⟨*proof*⟩

lemma *mod-ring-plus-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer* ===> *mod-ring-rel-integer*) (*plus-p-integer pp*) (+)
⟨*proof*⟩

lemma *mod-ring-minus-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer* ===> *mod-ring-rel-integer*) (*minus-p-integer pp*) (−)
⟨*proof*⟩

lemma *mod-ring-mult-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer* ===> *mod-ring-rel-integer*) (*mult-p-integer pp*) ((∗))
⟨*proof*⟩

lemma *mod-ring-eq-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer* ===> (=)) (=) (=)
⟨*proof*⟩

lemma *urel-integer-inj*: *urel-integer x y* ⟹ *urel-integer x z* ⟹ *y* = *z*
⟨*proof*⟩

lemma *urel-integer-inj'*: *urel-integer x z* ⟹ *urel-integer y z* ⟹ *x* = *y*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *right-total-mod-ring-rel-integer*: *right-total mod-ring-rel-integer*
⟨*proof*⟩

lemma *Domainp-mod-ring-rel-integer*: *Domainp mod-ring-rel-integer* = (λ*x*. *0* ≤ *x* ∧ *x* < *pp*)
⟨*proof*⟩

lemma *ring-finite-field-ops-integer*: *ring-ops* (*finite-field-ops-integer pp*) *mod-ring-rel-integer*
⟨*proof*⟩
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*) (λ*x*. *if x* = *0 then 0 else 1*) *normalize*
⟨*proof*⟩

lemma *mod-ring-mod-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer* ===> *mod-ring-rel-integer*) (λ*x y*. *if y = 0 then x else 0*) (*mod*)
⟨*proof*⟩

lemma *mod-ring-unit-factor-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer*) (λ*x*. *x*) *unit-factor*
⟨*proof*⟩

lemma *mod-ring-inverse-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer*) (*inverse-p-integer pp*) *inverse*
⟨*proof*⟩

lemma *mod-ring-divide-integer*: (*mod-ring-rel-integer* ===> *mod-ring-rel-integer* ===> *mod-ring-rel-integer*) (*divide-p-integer pp*) (/)
⟨*proof*⟩

lemma *finite-field-ops-integer*: *field-ops* (*finite-field-ops-integer pp*) *mod-ring-rel-integer*
⟨*proof*⟩
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* ⇒ '*b* ⇒ *bool*) ⇒ '*a mat* ⇒ '*b mat* ⇒ *bool* **where**
mat-rel R A B ≡ *dim-row A* = *dim-row B* ∧ *dim-col A* = *dim-col B* ∧
(∀ *i j*. *i* < *dim-row B* ⟶ *j* < *dim-col B* ⟶ *R* (*A* $$ (*i*,*j*)) (*B* $$ (*i*,*j*)))

lemma *right-total-mat-rel*: *right-total R* ⟹ *right-total* (*mat-rel R*)
⟨*proof*⟩

lemma *left-unique-mat-rel*: *left-unique R* ⟹ *left-unique* (*mat-rel R*)
⟨*proof*⟩

lemma *right-unique-mat-rel*: *right-unique R* ⟹ *right-unique* (*mat-rel R*)
⟨*proof*⟩

lemma *bi-unique-mat-rel*: *bi-unique R* ⟹ *bi-unique* (*mat-rel R*)
⟨*proof*⟩

lemma *mat-rel-eq*: ((*R* ===> *R* ===> (=))) (=) (=) ⟹
((*mat-rel R* ===> *mat-rel R* ===> (=))) (=) (=)
⟨*proof*⟩

definition *vec-rel* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a vec* ⇒ '*b vec* ⇒ *bool* **where**
vec-rel R A B ≡ *dim-vec A* = *dim-vec B* ∧ (∀ *i*. *i* < *dim-vec B* ⟶ *R* (*A* $ *i*)
(*B* $ *i*))

lemma *right-total-vec-rel*: *right-total R* ⟹ *right-total* (*vec-rel R*)
⟨*proof*⟩

lemma *left-unique-vec-rel*: *left-unique R* ⟹ *left-unique* (*vec-rel R*)
⟨*proof*⟩

lemma *right-unique-vec-rel*: *right-unique R* ⟹ *right-unique* (*vec-rel R*)
⟨*proof*⟩

lemma *bi-unique-vec-rel*: *bi-unique R* ⟹ *bi-unique* (*vec-rel R*)
⟨*proof*⟩

lemma *vec-rel-eq*: ((*R* ===> *R* ===> (=))) (=) (=) ⟹
((*vec-rel R* ===> *vec-rel R* ===> (=))) (=) (=)
⟨*proof*⟩

lemma *multrow-transfer*[*transfer-rule*]: ((*R* ===> *R* ===> *R*) ===> (=) ===>
R
===> *mat-rel R* ===> *mat-rel R*) *mat-multrow-gen mat-multrow-gen*

⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *pivot-positions-gen-transfer*: **assumes** [*transfer-rule*]: (*R* ===> *R* ===> (=)) (=) (=)
shows
(*R* ===> *mat-rel R* ===> (=)) *pivot-positions-gen pivot-positions-gen*
⟨*proof*⟩

lemma *set-pivot-positions-main-gen*:
set (*pivot-positions-main-gen ze A nr nc i j*) ⊆ {*0* ..< *nr*} × {*0* ..< *nc*}
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bi-unique-poly-rel*[*transfer-rule*]: *bi-unique* (*mat-rel R*)
⟨*proof*⟩

lemma *eq-mat-rel*[*transfer-rule*]: (*mat-rel R* ===> *mat-rel R* ===> (=)) (=) (=)
⟨*proof*⟩

lemma *multrow-i*[*transfer-rule*]: ((=) ===> *R* ===> *mat-rel R* ===> *mat-rel R*)
(*multrow-i ops*) *multrow*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *eliminate-entries-i*: **assumes**
vs: ⋀ *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*)
⟨*proof*⟩

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*⟩

lemma *gauss-jordan-i*[*transfer-rule*]:
(*mat-rel R* ===> *mat-rel R*) (*gauss-jordan-single-i ops*) *gauss-jordan-single*
⟨*proof*⟩

lemma *find-base-vectors-i*[*transfer-rule*]:
(*mat-rel R* ===> *list-all2* (*vec-rel R*)) (*find-base-vectors-i ops*) *find-base-vectors*
⟨*proof*⟩

end

lemma *list-of-vec-transfer*[*transfer-rule*]: (*vec-rel A* ===> *list-all2 A*) *list-of-vec list-of-vec*

⟨*proof*⟩

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

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**
$\lambda\ z\ mminus\ ttimes\ v\ (nr,\ nc,\ a)\ i'.$
$(nr,nc,let\ ai' = IArray.sub'\ (a,\ i')\ in\ (IArray.tabulate\ (integer\text{-}of\text{-}nat\ nr,\ \lambda\ i.\ let\ ai = IArray.sub'\ (a,\ i)\ in$
if $i = i'$ *then ai else*
let $vi'j = v\ i$
in if $vi'j = z$ *then ai*
else
$IArray.tabulate\ (integer\text{-}of\text{-}nat\ nc,\ \lambda\ j.\ mminus\ (IArray.sub'\ (ai,\ j))\ (ttimes\ vi'j\ (IArray.sub'\ (ai',\ j))))$
)))
⟨*proof*⟩

lemma *eliminate-entries-gen-zero* [*simp*]:
assumes $i<(dim\text{-}row\ A)$ $j<(dim\text{-}col\ A)$ **shows**
eliminate-entries-gen-zero mminus ttimes z v A I J $\$\$\ (i,\ j) =$
$(if\ v\ (integer\text{-}of\text{-}nat\ i) = z \vee i = I\ then\ A\ \$\$\ (i,j)\ else\ mminus\ (A\ \$\$\ (i,j))\ (ttimes\ (v\ (integer\text{-}of\text{-}nat\ i))\ (A\ \$\$\ (I,j))))$
⟨*proof*⟩

lemma *eliminate-entries-gen* [*simp*]:
assumes $i<(dim\text{-}row\ A)$ $j<(dim\text{-}col\ A)$ **shows**
eliminate-entries-gen mminus ttimes v A I J $\$\$\ (i,\ j) =$
$(if\ i = I\ then\ A\ \$\$\ (i,j)\ else\ mminus\ (A\ \$\$\ (i,j))\ (ttimes\ (v\ i)\ (A\ \$\$\ (I,j))))$
⟨*proof*⟩

lemma *dim-mat-impl* [*simp*]:
$dim\text{-}row\ (mat\text{-}impl\ x) = dim\text{-}row\text{-}impl\ x$
$dim\text{-}col\ (mat\text{-}impl\ x) = dim\text{-}col\text{-}impl\ x$
⟨*proof*⟩

lemma *dim-eliminate-entries-i2* [*simp*]:
$dim\text{-}row\text{-}impl\ (eliminate\text{-}entries\text{-}i2\ z\ mm\ tt\ v\ m\ i) = dim\text{-}row\text{-}impl\ m$
$dim\text{-}col\text{-}impl\ (eliminate\text{-}entries\text{-}i2\ z\ mm\ tt\ v\ m\ i) = dim\text{-}col\text{-}impl\ m$
⟨*proof*⟩

lemma *tabulate-nth*: $i < n \Longrightarrow IArray.tabulate\ (integer\text{-}of\text{-}nat\ n,\ f)\ !!\ i = f\ (integer\text{-}of\text{-}nat\ i)$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *ex-mset-zip-right*:
assumes *length xs* = *length ys mset ys*′ = *mset ys*
shows ∃ *xs*′. *length ys*′ = *length xs*′ ∧ *mset* (*zip xs*′ *ys*′) = *mset* (*zip xs ys*)
⟨*proof*⟩

lemma *list-all2-reorder-right-invariance*:
assumes *rel*: *list-all2 R xs ys* **and** *ms-y*: *mset ys*′ = *mset ys*
shows ∃ *xs*′. *list-all2 R xs*′ *ys*′ ∧ *mset xs*′ = *mset xs*
⟨*proof*⟩

lemma *rel-mset-via-perm*: *rel-mset rel* (*mset xs*) (*mset ys*) ⟷ (∃ *zs*. *mset xs* = *mset zs* ∧ *list-all2 rel zs ys*)
⟨*proof*⟩

end
theory *Unique-Factorization*
imports

Polynomial-Interpolation.Ring-Hom-Poly
Polynomial-Factorization.Polynomial-Irreducibility
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*) ⟷ *irreducible a*
⟨*proof*⟩

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

definition *coprime* :: *'a* ⇒ *'a* ⇒ *bool*
where *coprime-def'*: *coprime p q* ≡ ∀ *r*. *r dvd p* ⟶ *r dvd q* ⟶ *r dvd 1*

lemma *coprimeI*:
assumes ⋀*r*. *r dvd p* ⟹ *r dvd q* ⟹ *r dvd 1*
shows *coprime p q* ⟨*proof*⟩

lemma *coprimeE*:
assumes *coprime p q*
and (⋀*r*. *r dvd p* ⟹ *r dvd q* ⟹ *r dvd 1*) ⟹ *thesis*
shows *thesis* ⟨*proof*⟩

lemma *coprime-commute* [*ac-simps*]:
coprime p q ⟷ *coprime q p*
⟨*proof*⟩

lemma *not-coprime-iff-common-factor*:
¬ *coprime p q* ⟷ (∃ *r*. *r dvd p* ∧ *r dvd q* ∧ ¬ *r dvd 1*)
⟨*proof*⟩

end

lemma (**in** *algebraic-semidom*) *coprime-iff-coprime* [*simp*, *code*]:
coprime = *Rings.coprime*
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *coprime-0* [*simp*]:
coprime p 0 ⟷ *p dvd 1 coprime 0 p* ⟷ *p dvd 1*
⟨*proof*⟩

lemma *dvd-rewrites*: *dvd.dvd* ((∗)) = (*dvd*) ⟨*proof*⟩

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*⟩
lemma *coprime-inv-hom*[*simp*]: *coprime* (*Hilbert-Choice.inv hom x′*) *y* ⟷ *coprime x′* (*hom y*)
⟨*proof*⟩
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*⟩

lemma *dvd-const*:
assumes *pq*: (*p*::*′a*::*semidom poly*) *dvd q* **and** *q0*: *q* ≠ *0* **and** *degq*: *degree q* = *0*
shows *degree p* = *0*
⟨*proof*⟩

context *Rings.dvd* **begin**
abbreviation *ddvd* (**infix** ‹*ddvd*› *40*) **where** *x ddvd y* ≡ *x dvd y* ∧ *y dvd x*
lemma *ddvd-sym*[*sym*]: *x ddvd y* ⟹ *y ddvd x* ⟨*proof*⟩
end

context *comm-monoid-mult* **begin**
lemma *ddvd-trans*[*trans*]: *x ddvd y* ⟹ *y ddvd z* ⟹ *x ddvd z* ⟨*proof*⟩
lemma *ddvd-transp*: *transp* (*ddvd*) ⟨*proof*⟩
end

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

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

lemma *mset-factorsI*[*intro*!]:
  assumes $\bigwedge f.\ f \in\# F \Longrightarrow \textit{irreducible}\ f$ **and** $F \neq \{\#\}$ **and** *prod-mset F* = *p*
  shows *mset-factors F p*
  ⟨*proof*⟩

lemma *mset-factorsD*:
  assumes *mset-factors F p*
  shows $f \in\# F \Longrightarrow \textit{irreducible}\ f$ **and** $F \neq \{\#\}$ **and** *prod-mset F* = *p*
  ⟨*proof*⟩

lemma *mset-factorsE*[*elim*]:
  assumes *mset-factors F p*
    and $(\bigwedge f.\ f \in\# F \Longrightarrow \textit{irreducible}\ f) \Longrightarrow F \neq \{\#\} \Longrightarrow \textit{prod-mset}\ F = p \Longrightarrow$ *thesis*
  shows *thesis*
  ⟨*proof*⟩

lemma *mset-factors-imp-not-is-unit*:
  assumes *mset-factors F p*
  shows ¬ *p dvd 1*
⟨*proof*⟩

definition *primitive-poly* **where** *primitive-poly f* ≡ ∀ *d*. (∀ *i*. *d dvd coeff f i*) ⟶ *d dvd 1*

end

lemma(**in** *semidom*) *mset-factors-imp-nonzero*:
  assumes *mset-factors F p*
  shows *p* ≠ *0*
⟨*proof*⟩

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

3.3.2 Connecting to HOL/Divisibility

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

  abbreviation *mk-monoid* ≡ ⦇*carrier* = *UNIV* − {*0*}, *mult* = (∗), *one* = *1*⦈

  lemma *carrier-0*[*simp*]: *x* ∈ *carrier mk-monoid* ⟷ *x* ≠ *0* ⟨*proof*⟩

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

abbreviation *irred* **where** *irred* ≡ *Divisibility.irreducible mk-monoid*
abbreviation *factor* **where** *factor* ≡ *Divisibility.factor mk-monoid*
abbreviation *factors* **where** *factors* ≡ *Divisibility.factors mk-monoid*
abbreviation *properfactor* **where** *properfactor* ≡ *Divisibility.properfactor mk-monoid*

lemma *factors*: *factors fs y* ⟷ *prod-list fs* = *y* ∧ *Ball* (*set fs*) *irred*
⟨*proof*⟩

lemma *factor*: *factor x y* ⟷ (∃ *z*. *z* ≠ *0* ∧ *x* ∗ *z* = *y*) ⟨*proof*⟩

lemma *properfactor-nz*:
shows (*y* :: ′*a*) ≠ *0* ⟹ *properfactor x y* ⟷ *x dvd y* ∧ ¬ *y dvd x*
⟨*proof*⟩

lemma *mem-Units*[*simp*]: *y* ∈ *Units mk-monoid* ⟷ *y dvd 1*
⟨*proof*⟩

end

context *idom* **begin**
lemma *irred-0*[*simp*]: *irred* (*0*::′*a*) ⟨*proof*⟩
lemma *factor-idom*[*simp*]: *factor* (*x*::′*a*) *y* ⟷ (*if y* = *0 then x* = *0 else x dvd y*)
⟨*proof*⟩

lemma *associated-connect*[*simp*]: (∼$_{mk\text{-}monoid}$) = (*ddvd*) ⟨*proof*⟩

lemma *essentially-equal-connect*[*simp*]:
essentially-equal mk-monoid fs gs ⟷ *rel-mset* (*ddvd*) (*mset fs*) (*mset gs*)
⟨*proof*⟩

lemma *irred-idom-nz*:
assumes *x0*: (*x*::′*a*) ≠ *0*
shows *irred x* ⟷ *irreducible x*
⟨*proof*⟩

lemma *dvd-dvd-imp-unit-mult*:
assumes *xy*: *x dvd y* **and** *yx*: *y dvd x*
shows ∃ *z*. *z dvd 1* ∧ *y* = *x* ∗ *z*
⟨*proof*⟩

lemma *irred-inner-nz*:
assumes *x0*: *x* ≠ *0*
shows (∀ *b*. *b dvd x* ⟶ ¬ *x dvd b* ⟶ *b dvd 1*) ⟷ (∀ *a b*. *x* = *a* ∗ *b* ⟶ *a*

dvd 1 ∨ *b dvd 1*) (**is** *?l* ⟷ *?r*)
⟨*proof*⟩

lemma *irred-idom*[*simp*]: *irred x* ⟷ *x* = *0* ∨ *irreducible x*
⟨*proof*⟩

lemma assumes *x* ≠ *0* **and** *factors fs x* **and** *f* ∈ *set fs* **shows** *f* ≠ *0*
⟨*proof*⟩

lemma *factors-as-mset-factors*:
assumes *x0*: *x* ≠ *0* **and** *x1*: *x* ≠ *1*
shows *factors fs x* ⟷ *mset-factors* (*mset fs*) *x* ⟨*proof*⟩

end

context *ufd* **begin**
interpretation *comm-monoid-cancel*: *comm-monoid-cancel mk-monoid*::′*a monoid*
⟨*proof*⟩
lemma *factors-exist*:
assumes *a* ≠ *0*
and ¬ *a dvd 1*
shows ∃*fs*. *set fs* ⊆ *UNIV* − {*0*} ∧ *factors fs a*
⟨*proof*⟩

lemma *factors-unique*:
assumes *fs*: *factors fs a*
and *gs*: *factors gs a*
and *a0*: *a* ≠ *0*
and *a1*: ¬ *a dvd 1*
shows *rel-mset* (*ddvd*) (*mset fs*) (*mset gs*)
⟨*proof*⟩

lemma *factorial-monoid*: *factorial-monoid* (*mk-monoid* :: ′*a monoid*)
⟨*proof*⟩

end

lemma (**in** *idom*) *factorial-monoid-imp-ufd*:
assumes *factorial-monoid* (*mk-monoid* :: ′*a monoid*)
shows *class.ufd* ((∗) :: ′*a* ⇒ -) *1* (+) *0* (−) *uminus*
⟨*proof*⟩

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* $\Longrightarrow$ *irreducible* (*hom a*)
begin
lemma *hom-mset-factors*:
assumes *F*: *mset-factors F p*
shows *mset-factors* (*image-mset hom F*) (*hom p*)
⟨*proof*⟩
end

locale *unit-preserving-hom* = *comm-semiring-1-hom* +
assumes *is-unit-hom-if*: $\bigwedge x$. *hom x dvd 1* $\Longrightarrow$ *x dvd 1*
begin
lemma *is-unit-hom-iff*[*simp*]: *hom x dvd 1* $\longleftrightarrow$ *x dvd 1* ⟨*proof*⟩

lemma *irreducible-hom-imp-irreducible*:
assumes *irr*: *irreducible* (*hom a*) **shows** *irreducible a*
⟨*proof*⟩
end

locale *factor-preserving-hom* = *unit-preserving-hom* + *irreducibility-hom*
begin
lemma *irreducible-hom*[*simp*]: *irreducible* (*hom a*) $\longleftrightarrow$ *irreducible a*
⟨*proof*⟩
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*⟩

context *comm-semiring-isom* **begin**
sublocale *unit-preserving-hom* ⟨*proof*⟩
sublocale *factor-preserving-hom*
⟨*proof*⟩
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*⟩

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*⟩

lemma(**in** *idom*) *irreducible-factor-singleton*:
assumes *a*: *irreducible a*
shows *mset-factors F a* $\longleftrightarrow$ *F* = {#*a*#}
⟨*proof*⟩

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*⟩

lemma(**in** *idom*) *prod-mset-remove-units*:
prod-mset F ddvd prod-mset {# *f* ∈# *F*. ¬*f dvd 1* #}
⟨*proof*⟩

lemma(**in** *comm-semiring-1*) *mset-factors-imp-dvd*:
assumes *mset-factors F x* **and** *f* ∈# *F* **shows** *f dvd x*
⟨*proof*⟩

lemma(**in** *ufd*) *prime-elem-iff-irreducible*[*iff*]:
prime-elem x $\longleftrightarrow$ *irreducible x*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *gcd-zero-iff*[*simp*]: *gcd a b* = *0* $\longleftrightarrow$ *a* = *0* ∧ *b* = *0*
⟨*proof*⟩

lemma *gcd-zero-iff'*[*simp*]: *0* = *gcd a b* $\longleftrightarrow$ *a* = *0* ∧ *b* = *0*
⟨*proof*⟩

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*⟩

lemma *gcd-dvd-1*[*simp*]: *gcd a b dvd 1* ⟷ *coprime a b*
⟨*proof*⟩

lemma *dvd-imp-gcd-dvd-gcd*: *b dvd c* ⟹ *gcd a b dvd gcd a c*
⟨*proof*⟩

definition *listgcd* :: *'a list* ⇒ *'a* **where**
listgcd xs = *foldr gcd xs 0*

lemma *listgcd-simps*[*simp*]: *listgcd* [] = *0 listgcd* (*x* # *xs*) = *gcd x* (*listgcd xs*)
⟨*proof*⟩

lemma *listgcd*: *x* ∈ *set xs* ⟹ *listgcd xs dvd x*
⟨*proof*⟩

lemma *listgcd-greatest*: (⋀ *x*. *x* ∈ *set xs* ⟹ *y dvd x*) ⟹ *y dvd listgcd xs*
⟨*proof*⟩

end

context *Rings.dvd* **begin**

definition *is-gcd x a b* ≡ *x dvd a* ∧ *x dvd b* ∧ (∀ *y*. *y dvd a* ⟶ *y dvd b* ⟶ *y dvd x*)

definition *some-gcd a b* ≡ *SOME x*. *is-gcd x a b*

lemma *is-gcdI*[*intro*!]:
assumes *x dvd a x dvd b* ⋀*y*. *y dvd a* ⟹ *y dvd b* ⟹ *y dvd x*
shows *is-gcd x a b* ⟨*proof*⟩

lemma *is-gcdE*[*elim*!]:
assumes *is-gcd x a b*
and *x dvd a* ⟹ *x dvd b* ⟹ (⋀*y*. *y dvd a* ⟹ *y dvd b* ⟹ *y dvd x*) ⟹ *thesis*
shows *thesis* ⟨*proof*⟩

lemma *is-gcd-some-gcdI*:
assumes ∃ *x*. *is-gcd x a b* **shows** *is-gcd* (*some-gcd a b*) *a b*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *some-gcd-0-dvd*[*intro*!]:
some-gcd a 0 dvd a some-gcd 0 b dvd b ⟨*proof*⟩

lemma *dvd-some-gcd-0*[*intro*!]:
a dvd some-gcd a 0 b dvd some-gcd 0 b ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *some-gcd-connect*:
assumes $a \neq 0$ **and** $b \neq 0$ **shows** *somegcd mk-monoid a b* = *some-gcd a b*
⟨*proof*⟩
end

context *comm-monoid-gcd*
begin
lemma *is-gcd-gcd*: *is-gcd* (*gcd a b*) *a b* ⟨*proof*⟩
lemma *is-gcd-some-gcd*: *is-gcd* (*some-gcd a b*) *a b* ⟨*proof*⟩
lemma *gcd-dvd-some-gcd*: *gcd a b dvd some-gcd a b* ⟨*proof*⟩
lemma *some-gcd-dvd-gcd*: *some-gcd a b dvd gcd a b* ⟨*proof*⟩
lemma *some-gcd-ddvd-gcd*: *some-gcd a b ddvd gcd a b* ⟨*proof*⟩
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*
⟨*proof*⟩

end

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

interpretation *raw*: *comm-monoid-cancel mk-monoid* :: *'a monoid*
⟨*proof*⟩

interpretation *raw*: *gcd-condition-monoid mk-monoid* :: *'a monoid*
⟨*proof*⟩

lemma *gcd-mult-ddvd*:
d ∗ *gcd a b ddvd gcd* (*d* ∗ *a*) (*d* ∗ *b*)
⟨*proof*⟩

lemma *gcd-greatest-mult*: **assumes** *cad*: *c dvd a * d* **and** *cbd*: *c dvd b * d*
shows *c dvd gcd a b * d*
⟨*proof*⟩

lemma *listgcd-greatest-mult*: $(\bigwedge x :: 'a.\ x \in set\ xs \Longrightarrow y\ dvd\ x * z) \Longrightarrow y\ dvd\ listgcd\ xs * z$
⟨*proof*⟩

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*⟩

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*⟩

end

subclass (**in** *ring-gcd*) *idom-gcd* ⟨*proof*⟩

lemma *coprime-rewrites*: *comm-monoid-mult.coprime ((*)) 1 = coprime*
⟨*proof*⟩

locale *gcd-condition* =
fixes *ty* :: *'a* :: *idom itself*
assumes *gcd-exists*: $\bigwedge a\ b :: 'a.\ \exists x.$ *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*⟩
end

instance *semiring-gcd* ⊆ *comm-monoid-gcd* ⟨*proof*⟩

lemma *listgcd-connect*: *listgcd = gcd-list*
⟨*proof*⟩

interpretation *some-gcd*: *gcd-condition TYPE('a::ufd)*
⟨*proof*⟩

lemma *some-gcd-listgcd-dvd-listgcd*: *some-gcd.listgcd xs dvd listgcd xs*
⟨*proof*⟩

lemma *listgcd-dvd-some-gcd-listgcd*: *listgcd xs dvd some-gcd.listgcd xs*

⟨*proof*⟩

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

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

sublocale *as-ufd*: *ufd*
⟨*proof*⟩

end

instance *int* :: *ufd* ⟨*proof*⟩
instance *int* :: *idom-gcd* ⟨*proof*⟩

instance *field* ⊆ *ufd* ⟨*proof*⟩

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 ⟨*proof*⟩

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
⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩
end

instance *fract* :: (*idom*) *field-gcd* ⟨*proof*⟩

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*⟩

lemma *divides-ff-to-fract*[*simp*]: *divides-ff* (*to-fract x*) (*to-fract y*) ⟷ *x dvd y*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *some-gcd-ff-list-divides*: *x* ∈ *set xs* ⟹ *divides-ff* (*some-gcd-ff-list xs*) *x*
⟨*proof*⟩

lemma *some-gcd-ff-list-greatest*: (∀ *x* ∈ *set xs*. *divides-ff d x*) ⟹ *divides-ff d* (*some-gcd-ff-list xs*)
⟨*proof*⟩

lemma *divides-ff-refl*[*simp*]: *divides-ff x x*
⟨*proof*⟩

lemma *divides-ff-trans*:
divides-ff x y ⟹ *divides-ff y z* ⟹ *divides-ff x z*
⟨*proof*⟩

lemma *divides-ff-mult-right*: *a* ≠ *0* ⟹ *divides-ff* (*x* ∗ *inverse a*) *y* ⟹ *divides-ff x* (*a* ∗ *y*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *eq-dff-refl*[*simp*]: *x* =*dff x*
⟨*proof*⟩

lemma *eq-dff-sym*: *x* =*dff y* ⟹ *y* =*dff x* ⟨*proof*⟩

lemma *eq-dff-trans*[*trans*]: *x* =*dff y* ⟹ *y* =*dff z* ⟹ *x* =*dff z*
⟨*proof*⟩

lemma *eq-dff-cancel-right*[*simp*]: *x* ∗ *y* =*dff x* ∗ *z* ⟷ *x* = *0* ∨ *y* =*dff z*
⟨*proof*⟩

lemma *eq-dff-mult-right-trans*[*trans*]: *x* =*dff y* ∗ *z* ⟹ *z* =*dff u* ⟹ *x* =*dff y* ∗ *u*
⟨*proof*⟩

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

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*⟩

lemma *content-ff-divides-ff*: *x* ∈ *set* (*coeffs p*) ⟹ *divides-ff* (*content-ff p*) *x*
⟨*proof*⟩

lemma *content-ff-0*[*simp*]: *content-ff 0* = *0*
⟨*proof*⟩

lemma *content-ff-0-iff*[*simp*]: (*content-ff p* = *0*) = (*p* = *0*)
⟨*proof*⟩

lemma *content-ff-eq-dff-nonzero*: *content-ff p* =*dff x* ⟹ *x* ≠ *0* ⟹ *p* ≠ *0*
⟨*proof*⟩

lemma *content-ff-smult*: *content-ff* (*smult* (*a::'a::ufd fract*) *p*) =*dff a* ∗ *content-ff p*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *content-ff-normalize-content-ff-1*: **assumes** *p0*: *p* ≠ *0*
shows *content-ff* (*normalize-content-ff p*) =*dff 1*
⟨*proof*⟩

lemma *content-ff-to-fract*: **assumes** *set* (*coeffs p*) ⊆ *range to-fract*
shows *content-ff p* ∈ *range to-fract*
⟨*proof*⟩

lemma *content-ff-map-poly-to-fract*: *content-ff* (*map-poly to-fract* (*p* :: *'a* :: *ufd poly*)) ∈ *range to-fract*
⟨*proof*⟩

lemma *range-coeffs-to-fract*: **assumes** *set* (*coeffs p*) ⊆ *range to-fract*
shows ∃ *m*. *coeff p i* = *to-fract m*
⟨*proof*⟩

lemma *divides-ff-coeff*: **assumes** *set* (*coeffs p*) ⊆ *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*⟩

definition *inv-embed* :: *'a* :: *ufd fract* $\Rightarrow$ *'a* **where**
inv-embed = *the-inv to-fract*

lemma *inv-embed*[*simp*]: *inv-embed* (*to-fract x*) = *x*
⟨*proof*⟩

lemma *inv-embed-0*[*simp*]: *inv-embed 0* = *0* ⟨*proof*⟩

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*⟩

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*⟩

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

lemma *gauss-lemma*:
fixes *p q* :: *'a* :: *ufd fract poly*
shows *content-ff* (*p* $*$ *q*) =*dff content-ff p* $*$ *content-ff q*
⟨*proof*⟩

abbreviation (*input*) *content-ff-ff p* $\equiv$ *content-ff* (*map-poly to-fract p*)

lemma *factorization-to-fract*:
assumes *q*: *q* $\neq$ *0* **and** *factor*: *map-poly to-fract* (*p* :: *'a* :: *ufd poly*) = *q* $*$ *r*
shows $\exists$ *q' r' c*. *c* $\neq$ *0* $\wedge$ *q* = *smult c* (*map-poly to-fract q'*) $\wedge$
r = *smult* (*inverse c*) (*map-poly to-fract r'*) $\wedge$
content-ff-ff q' =*dff 1* $\wedge$ *p* = *q'* $*$ *r'*
⟨*proof*⟩

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-poly to-fract p*) $\wedge$ *content-ff-ff p* =*dff 1*
⟨*proof*⟩

lemma *irreducible-M-PM*:
fixes *p* :: *'a* :: *ufd poly* **assumes** *0*: *degree p* = *0* **and** *irr*: *irreducible* (*coeff p 0*)
shows *irreducible p*
⟨*proof*⟩

lemma *primitive-irreducible-imp-degree*:
primitive (*p*::'*a*::{*semiring-gcd*,*idom*} *poly*) ⟹ *irreducible p* ⟹ *degree p* > *0*
⟨*proof*⟩

lemma *irreducible-degree-field*:
fixes *p* :: '*a* :: *field poly* **assumes** *irreducible p*
shows *degree p* > *0*
⟨*proof*⟩

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

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*
⟨*proof*⟩

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*⟩

lemma *factorial-monoid-poly*: *factorial-monoid* (*mk-monoid* :: '*a* :: *ufd poly monoid*)
⟨*proof*⟩

instance *poly* :: (*ufd*) *ufd*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *M-M*[*simp*]: *M* (*M x*) = *M x*
⟨*proof*⟩

lemma *M-plus*[*simp*]: *M* (*M x* + *y*) = *M* (*x* + *y*) *M* (*x* + *M y*) = *M* (*x* + *y*)
⟨*proof*⟩

lemma *M-minus*[*simp*]: *M* (*M x* − *y*) = *M* (*x* − *y*) *M* (*x* − *M y*) = *M* (*x* − *y*)
⟨*proof*⟩

lemma *M-times*[*simp*]: *M* (*M x* ∗ *y*) = *M* (*x* ∗ *y*) *M* (*x* ∗ *M y*) = *M* (*x* ∗ *y*)
⟨*proof*⟩

lemma *M-sum*: *M* (*sum* (λ *x*. *M* (*f x*)) *A*) = *M* (*sum f A*)
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *Mp-0*[*simp*]: *Mp 0* = *0* ⟨*proof*⟩

lemma *Mp-coeff*: *coeff* (*Mp f*) *i* = *M* (*coeff f i*) ⟨*proof*⟩

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*⟩

lemma *plus-Mp*[*simp*]: *Mp* (*Mp f* + *g*) = *Mp* (*f* + *g*) *Mp* (*f* + *Mp g*) = *Mp* (*f* +

g)
⟨*proof*⟩

lemma *minus-Mp*[*simp*]: $Mp\ (Mp\ f\ -\ g) = Mp\ (f\ -\ g)\ Mp\ (f\ -\ Mp\ g) = Mp\ (f\ -\ g)$
⟨*proof*⟩

lemma *Mp-smult*[*simp*]: $Mp\ (smult\ (M\ a)\ f) = Mp\ (smult\ a\ f)\ Mp\ (smult\ a\ (Mp\ f)) = Mp\ (smult\ a\ f)$
⟨*proof*⟩

lemma *Mp-Mp*[*simp*]: $Mp\ (Mp\ f) = Mp\ f$ ⟨*proof*⟩

lemma *Mp-smult-m-0*[*simp*]: $Mp\ (smult\ m\ f) = 0$
⟨*proof*⟩

definition *dvdm* :: *int poly* ⇒ *int poly* ⇒ *bool* (**infix** ‹*dvdm*› *50*) **where**
$f\ dvdm\ g = (\exists\ h.\ g =_m f * h)$
notation *dvdm* (**infix** ‹*dvdm*› *50*)

lemma *dvdmE*:
assumes *fg*: $f\ dvdm\ g$
and *main*: $\bigwedge h.\ g =_m f * h \Longrightarrow Mp\ h = h \Longrightarrow thesis$
shows *thesis*
⟨*proof*⟩

lemma *Mp-dvdm*[*simp*]: $Mp\ f\ dvdm\ g \longleftrightarrow f\ dvdm\ g$
and *dvdm-Mp*[*simp*]: $f\ dvdm\ Mp\ g \longleftrightarrow f\ dvdm\ g$ ⟨*proof*⟩

definition *irreducible-m*
where $irreducible\text{-}m\ f = (\neg f =_m 0\ \wedge\ \neg\ f\ dvdm\ 1\ \wedge\ (\forall\ a\ b.\ f =_m a * b \longrightarrow a\ dvdm\ 1\ \vee\ b\ dvdm\ 1))$

definition $irreducible_d\text{-}m$ :: *int poly* ⇒ *bool* **where** $irreducible_d\text{-}m\ f \equiv$
$degree\text{-}m\ f > 0\ \wedge$
$(\forall\ g\ h.\ degree\text{-}m\ g < degree\text{-}m\ f \longrightarrow degree\text{-}m\ h < degree\text{-}m\ f \longrightarrow \neg\ f =_m g * h)$

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

lemma *degree-m-le-degree* [*intro*!]: $degree\text{-}m\ f \leq degree\ f$
⟨*proof*⟩

lemma $irreducible_d\text{-}mI$:
assumes *f0*: $degree\text{-}m\ f > 0$
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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *square-free-m-cong*: *square-free-m f* $\Longrightarrow$ *Mp f* = *Mp g* $\Longrightarrow$ *square-free-m g*
⟨*proof*⟩

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

lemma *Mp-prod-list*: *Mp* (*prod-list* (*map Mp b*)) = *Mp* (*prod-list b*)
⟨*proof*⟩

Polynomial evaluation modulo

definition *M-poly p x* $\equiv$ *M* (*poly p x*)

lemma *M-poly-Mp*[*simp*]: *M-poly* (*Mp p*) = *M-poly p*
⟨*proof*⟩

lemma *Mp-lift-modulus*: **assumes** *f* $=_m$ *g*
shows *poly-mod.eq-m* (*m* $*$ *k*) (*smult k f*) (*smult k g*)
⟨*proof*⟩

lemma *Mp-ident-product*: *n* $>$ *0* $\Longrightarrow$ *Mp f* = *f* $\Longrightarrow$ *poly-mod.Mp* (*m* $*$ *n*) *f* = *f*
⟨*proof*⟩

lemma *Mp-shrink-modulus*: **assumes** *poly-mod.eq-m* $(m * k)$ f g $k \neq 0$
shows $f =m\ g$
⟨*proof*⟩

lemma *degree-m-le*: *degree-m* $f \leq$ *degree* f ⟨*proof*⟩

lemma *degree-m-eq*: *coeff* f (*degree* f) *mod* $m \neq 0 \Longrightarrow m > 1 \Longrightarrow$ *degree-m* $f =$ *degree* f
⟨*proof*⟩

lemma *degree-m-mult-le*:
assumes *eq*: $f =m\ g * h$
shows *degree-m* $f \leq$ *degree-m* $g +$ *degree-m* h
⟨*proof*⟩

lemma *degree-m-smult-le*: *degree-m* (*smult* c f) $\leq$ *degree-m* f
⟨*proof*⟩

lemma *irreducible-m-Mp*[*simp*]: *irreducible-m* (*Mp* f) $\longleftrightarrow$ *irreducible-m* f ⟨*proof*⟩

lemma *eq-m-irreducible-m*: $f =m\ g \Longrightarrow$ *irreducible-m* $f \longleftrightarrow$ *irreducible-m* g
⟨*proof*⟩

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

definition *factorization* :: $'a$ :: *comm-semiring-1 poly set* $\Rightarrow$ $'a$ *poly* $\Rightarrow$ ($'a \times 'a$ *poly multiset*) $\Rightarrow$ *bool* **where**
factorization Factors f cfs $\equiv$ (*case cfs of* (*c*,*fs*) $\Rightarrow$ $f =$ (*smult c* (*prod-mset fs*)) $\wedge$ (*set-mset fs* $\subseteq$ *Factors*))

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

lemma *irreducible-multD*:
assumes *l*: *irreducible* $(a*b)$
shows *a dvd 1* $\wedge$ *irreducible b* $\vee$ *b dvd 1* $\wedge$ *irreducible a*

⟨*proof*⟩

lemma *irreducible-dvd-prod-mset*:
fixes $p :: 'a :: field\ poly$
assumes *irr*: *irreducible p* **and** *dvd*: *p dvd prod-mset as*
shows $\exists\ a \in\# as.\ p\ dvd\ a$
⟨*proof*⟩

lemma *monic-factorization-unique-mset*:
fixes $P::'a::field\ poly\ multiset$
assumes *eq*: *prod-mset P* = *prod-mset Q*
and *P*: *set-mset P* ⊆ $\{q.\ irreducible\ q \wedge monic\ q\}$
and *Q*: *set-mset Q* ⊆ $\{q.\ irreducible\ q \wedge monic\ q\}$
shows *P* = *Q*
⟨*proof*⟩

lemma *exactly-one-monic-factorization*:
assumes *mon*: *monic* $(f :: 'a :: field\ poly)$
shows $\exists!\ fs.\ f = prod\text{-}mset\ fs \wedge set\text{-}mset\ fs \subseteq \{q.\ irreducible\ q \wedge monic\ q\}$
⟨*proof*⟩

lemma *monic-prod-mset*:
fixes $as :: 'a :: idom\ poly\ multiset$
assumes $\bigwedge a.\ a \in set\text{-}mset\ as \Longrightarrow monic\ a$
shows *monic* (*prod-mset as*) ⟨*proof*⟩

lemma *exactly-one-factorization*:
assumes *f*: $f \neq (0 :: 'a :: field\ poly)$
shows $\exists!\ cfs.\ factorization\ Irr\text{-}Mon\ f\ cfs$
⟨*proof*⟩

lemma *mod-ident-iff*:
‹(*x* :: *int*) *mod m* = *x* ⟷ *x* ∈ {*0* ..< *m*}›
if ‹*m* > *0*›
⟨*proof*⟩

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

lemma *mult-1-is-id*[*simp*]: (∗) (*1* :: *'a* :: *ring-1*) = *id* ⟨*proof*⟩

context *poly-mod*
begin

lemma *degree-m-eq-monic*: *monic f* ⟹ *m* > *1* ⟹ *degree-m f* = *degree f*
⟨*proof*⟩

lemma *monic-degree-m-lift*: **assumes** *monic f k* > *1 m* > *1*
shows *monic* (*poly-mod.Mp* (*m* ∗ *k*) *f*)

⟨*proof*⟩

end

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

lemma *M-1*[*simp*]: *M 1* = *1* ⟨*proof*⟩

lemma *Mp-1*[*simp*]: *Mp 1* = *1* ⟨*proof*⟩

lemma *monic-degree-m*[*simp*]: *monic f* ⟹ *degree-m f* = *degree f*
  ⟨*proof*⟩

lemma *monic-Mp*: *monic f* ⟹ *monic* (*Mp f*)
  ⟨*proof*⟩

lemma *Mp-0-smult-sdiv-poly*: **assumes** *Mp f* = *0*
  shows *smult m* (*sdiv-poly f m*) = *f*
⟨*proof*⟩

lemma *Mp-product-modulus*: *m'* = *m* ∗ *k* ⟹ *k* > *0* ⟹ *Mp* (*poly-mod.Mp m' f*)
= *Mp f*
  ⟨*proof*⟩

lemma *inv-M-rev*: **assumes** *bnd*: *2* ∗ *abs c* < *m*
  shows *inv-M* (*M c*) = *c*
⟨*proof*⟩

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*⟩

lemma *monic-smult-add-small*: **assumes** *f* = *0* ∨ *degree f* < *degree g* **and** *mon*:
monic g
  shows *monic* (*g* + *smult q f*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *Mf-factorization-m*[*simp*]: *factorization-m f* (*Mf cfs*) = *factorization-m f*
cfs
⟨*proof*⟩

lemma *unique-factorization-m-imp-factorization*: **assumes** *unique-factorization-m*
f cfs
shows *factorization-m f cfs*
⟨*proof*⟩

lemma *unique-factorization-m-alt-def*: *unique-factorization-m f cfs* = (*factorization-m*
f cfs
∧ (∀ *dgs*. *factorization-m f dgs* ⟶ *Mf dgs* = *Mf cfs*))
⟨*proof*⟩

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*⟩

lemma *factorization-m-smult*: **assumes** *factorization-m f* (*c*,*fs*)
shows *factorization-m* (*smult d f*) (*c* ∗ *d*,*fs*)
⟨*proof*⟩

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*⟩

lemma *Mp-factorization-m*[*simp*]: *factorization-m* (*Mp f*) *cfs* = *factorization-m f cfs*
⟨*proof*⟩

lemma *Mp-unique-factorization-m*[*simp*]:
unique-factorization-m (*Mp f*) *cfs* = *unique-factorization-m f cfs*
⟨*proof*⟩

lemma *unique-factorization-m-cong*: *unique-factorization-m f cfs* ⟹ *Mp f* = *Mp g*
⟹ *unique-factorization-m g cfs*
⟨*proof*⟩

lemma *unique-factorization-mI*: **assumes** *factorization-m f* (*c*,*fs*)
and ⋀ *d gs*. *factorization-m f* (*d*,*gs*) ⟹ *Mf* (*d*,*gs*) = *Mf* (*c*,*fs*)
shows *unique-factorization-m f* (*c*,*fs*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *degree-m-eq-lead-coeff*: *degree-m f* = *degree f* ⟹ *lead-coeff* (*Mp f*) = *M* (*lead-coeff f*)
⟨*proof*⟩

lemma *unique-factorization-m-zero*: **assumes** *unique-factorization-m f* (*c*,*fs*)
shows *M c* ≠ *0*
⟨*proof*⟩

end

context *poly-mod*
begin

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

⟨*proof*⟩

lemma *dvdm-factor*: **assumes** *f dvdm g*
shows *f dvdm g ∗ h*
⟨*proof*⟩

lemma *square-free-m-smultD*: **assumes** *square-free-m* (*smult c f*)
shows *square-free-m f*
⟨*proof*⟩

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*⟩

lemma *square-free-m-factor*: **assumes** *square-free-m* (*f ∗ g*)
shows *square-free-m f square-free-m g*
⟨*proof*⟩

end

context *poly-mod-2*
begin

lemma *Mp-ident-iff*: *Mp f* = *f* ⟷ (∀ *n*. *coeff f n* ∈ {*0* ..< *m*})
⟨*proof*⟩

lemma *Mp-ident-iff′*: *Mp f* = *f* ⟷ (*set* (*coeffs f*) ⊆ {*0* ..< *m*})
⟨*proof*⟩
end

lemma *Mp-Mp-pow-is-Mp*: *n* ≠ *0* ⟹ *p* > *1* ⟹ *poly-mod.Mp p* (*poly-mod.Mp* (*p*^*n*) *f*)
= *poly-mod.Mp p f*
⟨*proof*⟩

lemma *M-M-pow-is-M*: *n* ≠ *0* ⟹ *p* > *1* ⟹ *poly-mod.M p* (*poly-mod.M* (*p*^*n*) *f*)
= *poly-mod.M p f* ⟨*proof*⟩

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*⟩

lemma *inverse-mod-pow*:
  (*inverse-mod x* (*p* ^ *n*) ∗ *x*) *mod* (*p* ^ *n*) = *1*
  if *coprime x p p* > *1 n* ≠ *0*
  ⟨*proof*⟩

lemma (**in** *poly-mod*) *inverse-mod-coprime*:
  assumes *p*: *prime m*
  and *cop*: *coprime x m* **shows** *M* (*inverse-mod x m* ∗ *x*) = *1*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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

sublocale *poly-mod-2 p* ⟨*proof*⟩

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

lemma *coprime-exp-mod*: *coprime lu p* ⟹ *n* ≠ *0* ⟹ *lu mod p* ^ *n* ≠ *0*
  ⟨*proof*⟩

end

context *poly-mod*
begin

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

lemma *Dp-Mp-eq*: *f* = *Mp f* + *smult m* (*Dp f*)
  ⟨*proof*⟩

lemma *dvd-imp-dvdm*:
  assumes *a dvd b* **shows** *a dvdm b*
  ⟨*proof*⟩

lemma *dvdm-add*:
  assumes *a*: *u dvdm a*
  and *b*: *u dvdm b*
  shows *u dvdm* (*a*+*b*)
⟨*proof*⟩

lemma *monic-dvdm-constant*:
assumes *uk*: *u dvdm* [:*k*:]
and *u1*: *monic u* **and** *u2*: *degree u* > *0*
shows *k mod m* = *0*
⟨*proof*⟩

lemma *div-mod-imp-dvdm*:
assumes ∃ *q r*. *b* = *q* ∗ *a* + *Polynomial.smult m r*
shows *a dvdm b*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

lemma *dvdm-uminus* [*simp*]:
p dvdm −*q* ⟷ *p dvdm q*
⟨*proof*⟩

lemma *Mp-const-poly*: *Mp* [:*a*:] = [:*a mod m*:]
⟨*proof*⟩

lemma *dvdm-imp-div-mod*:
assumes *u dvdm g*
shows ∃ *q r*. *g* = *q*∗*u* + *smult m r*
⟨*proof*⟩

corollary *div-mod-iff-dvdm*:
shows *a dvdm b* = (∃ *q r*. *b* = *q* ∗ *a* + *Polynomial.smult m r*)
⟨*proof*⟩

lemma *dvdmE'*:
assumes *p dvdm q* **and** ⋀*r*. *q* =*m p* ∗ *Mp r* ⟹ *thesis*
shows *thesis*

⟨*proof*⟩

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*⟩

lemma *dvdm-degree*: *monic u* ⟹ *u dvdm f* ⟹ *Mp f* ≠ *0* ⟹ *degree u* ≤ *degree f*
⟨*proof*⟩

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*⟩

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*⟩

abbreviation *to-int-poly* :: *'a* :: *finite mod-ring poly* ⇒ *int poly* **where**
to-int-poly ≡ *map-poly to-int-mod-ring*

interpretation *to-int-poly-hom*: *map-poly-inj-zero-hom to-int-mod-ring* ⟨*proof*⟩

lemma *irreducible$_d$-def-0*:
fixes $f :: 'a :: \{comm\text{-}semiring\text{-}1, semiring\text{-}no\text{-}zero\text{-}divisors\}$ *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*⟩

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*: $m > 1$ ⟨*proof*⟩

sublocale *poly-mod-2* ⟨*proof*⟩

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

definition *M-Rel* :: *int* ⇒ $'a$ *mod-ring* ⇒ *bool*
where *M-Rel x x'* ≡ (*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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *degree-MP-Rel* [*transfer-rule*]: (*MP-Rel* ===> (=)) *degree-m degree*
⟨*proof*⟩

lemma *eq-M-Rel*[*transfer-rule*]: (*M-Rel* ===> *M-Rel* ===> (=)) (λ *x y*. *M x* = *M y*) (=)
⟨*proof*⟩

interpretation *to-int-mod-ring-hom*: *map-poly-inj-zero-hom to-int-mod-ring*⟨*proof*⟩

lemma *eq-MP-Rel*[*transfer-rule*]: (*MP-Rel* ===> *MP-Rel* ===> (=)) (=*m*) (=)
⟨*proof*⟩

lemma *eq-Mf-Rel*[*transfer-rule*]: (*MF-Rel* ===> *MF-Rel* ===> (=)) (λ *x y*. *Mf x* = *Mf y*) (=)
⟨*proof*⟩

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*) (+) (+)
⟨*proof*⟩

lemma *times-MP-Rel*[*transfer-rule*]: (*MP-Rel* ===> *MP-Rel* ===> *MP-Rel*) ((∗)) ((∗))
⟨*proof*⟩

lemma *smult-MP-Rel*[*transfer-rule*]: (*M-Rel* ===> *MP-Rel* ===> *MP-Rel*) *smult smult*
⟨*proof*⟩

lemma *one-M-Rel*[*transfer-rule*]: *M-Rel 1 1*
⟨*proof*⟩

lemma *one-MP-Rel*[*transfer-rule*]: *MP-Rel 1 1*
⟨*proof*⟩

lemma *zero-M-Rel*[*transfer-rule*]: *M-Rel 0 0*
⟨*proof*⟩

lemma *zero-MP-Rel*[*transfer-rule*]: *MP-Rel 0 0*
⟨*proof*⟩

lemma *listprod-MP-Rel*[*transfer-rule*]: (*list-all2 MP-Rel* ===> *MP-Rel*) *prod-list prod-list*
⟨*proof*⟩

lemma *prod-mset-MP-Rel*[*transfer-rule*]: (*rel-mset MP-Rel* ===> *MP-Rel*) *prod-mset prod-mset*
⟨*proof*⟩

lemma *right-unique-MP-Rel*[*transfer-rule*]: *right-unique MP-Rel*
⟨*proof*⟩

lemma *M-to-int-mod-ring*: *M* (*to-int-mod-ring* (*x* :: *'a mod-ring*)) = *to-int-mod-ring x*
⟨*proof*⟩

lemma *Mp-to-int-poly*: *Mp* (*to-int-poly* (*f* :: *'a mod-ring poly*)) = *to-int-poly f*
⟨*proof*⟩

lemma *right-total-M-Rel*[*transfer-rule*]: *right-total M-Rel*
⟨*proof*⟩

lemma *left-total-M-Rel*[*transfer-rule*]: *left-total M-Rel*

⟨*proof*⟩

lemma *bi-total-M-Rel*[*transfer-rule*]: *bi-total M-Rel*
⟨*proof*⟩

lemma *right-total-MP-Rel*[*transfer-rule*]: *right-total MP-Rel*
⟨*proof*⟩

lemma *to-int-mod-ring-of-int-M*: *to-int-mod-ring* (*of-int x* :: *'a mod-ring*) = *M x*
⟨*proof*⟩

lemma *Mp-f-representative*: *Mp f* = *to-int-poly* (*map-poly of-int f* :: *'a mod-ring poly*)
⟨*proof*⟩

lemma *left-total-MP-Rel*[*transfer-rule*]: *left-total MP-Rel*
⟨*proof*⟩

lemma *bi-total-MP-Rel*[*transfer-rule*]: *bi-total MP-Rel*
⟨*proof*⟩

lemma *bi-total-MF-Rel*[*transfer-rule*]: *bi-total MF-Rel*
⟨*proof*⟩

lemma *right-total-MF-Rel*[*transfer-rule*]: *right-total MF-Rel*
⟨*proof*⟩

lemma *left-total-MF-Rel*[*transfer-rule*]: *left-total MF-Rel*
⟨*proof*⟩

lemma *domain-RT-rel*[*transfer-domain-rule*]: *Domainp MP-Rel* = (λ *f*. *True*)
⟨*proof*⟩

lemma *mem-MP-Rel*[*transfer-rule*]: (*MP-Rel* ===> *rel-set MP-Rel* ===> (=)) (λ *x Y*. ∃ *y* ∈ *Y*. *eq-m x y*) (∈)
⟨*proof*⟩

lemma *conversep-MP-Rel-OO-MP-Rel* [*simp*]: *MP-Rel*$^{-1-1}$ *OO MP-Rel* = (=)
⟨*proof*⟩

lemma *MP-Rel-OO-conversep-MP-Rel* [*simp*]: *MP-Rel OO MP-Rel*$^{-1-1}$ = *eq-m*
⟨*proof*⟩

lemma *conversep-MP-Rel-OO-eq-m* [*simp*]: *MP-Rel*$^{-1-1}$ *OO eq-m* = *MP-Rel*$^{-1-1}$
⟨*proof*⟩

lemma *eq-m-OO-MP-Rel* [*simp*]: *eq-m OO MP-Rel* = *MP-Rel*
⟨*proof*⟩

lemma *eq-mset-MP-Rel* [*transfer-rule*]: (*rel-mset MP-Rel* ===> *rel-mset MP-Rel* ===> (=)) (*rel-mset eq-m*) (=)
⟨*proof*⟩

lemma *dvd-MP-Rel*[*transfer-rule*]: (*MP-Rel* ===> *MP-Rel* ===> (=)) (*dvdm*) (*dvd*)
⟨*proof*⟩

lemma *irreducible-MP-Rel* [*transfer-rule*]: (*MP-Rel* ===> (=)) *irreducible-m irreducible*
⟨*proof*⟩

lemma *irreducible$_d$-MP-Rel* [*transfer-rule*]: (*MP-Rel* ===> (=)) *irreducible$_d$-m irreducible$_d$*
⟨*proof*⟩

lemma *UNIV-M-Rel*[*transfer-rule*]: *rel-set M-Rel* {*0..<m*} *UNIV*
⟨*proof*⟩

lemma *coeff-MP-Rel* [*transfer-rule*]: (*MP-Rel* ===> (=) ===> *M-Rel*) *coeff coeff*
⟨*proof*⟩

lemma *M-1-1*: *M 1 = 1* ⟨*proof*⟩

lemma *square-free-MP-Rel* [*transfer-rule*]: (*MP-Rel* ===> (=)) *square-free-m square-free*
⟨*proof*⟩

lemma *mset-factors-m-MP-Rel* [*transfer-rule*]: (*rel-mset MP-Rel* ===> *MP-Rel* ===> (=)) *mset-factors-m mset-factors*
⟨*proof*⟩

lemma *coprime-MP-Rel* [*transfer-rule*]: (*MP-Rel* ===> *MP-Rel* ===> (=)) *coprime-m coprime*
⟨*proof*⟩

lemma *prime-elem-MP-Rel* [*transfer-rule*]: (*MP-Rel* ===> (=)) *prime-elem-m prime-elem*
⟨*proof*⟩

end

context *poly-mod-2* **begin**

lemma *non-empty*: {*0..<m*} ≠ {} ⟨*proof*⟩

lemma *type-to-set*:
assumes *type-def*: ∃(*Rep* :: *'b* ⇒ *int*) *Abs*. *type-definition Rep Abs* {*0* ..< *m* :: *int*}

shows *class.nontriv* (*TYPE*('*b*)) (**is** *?a*) **and** *m* = *int CARD*('*b*) (**is** *?b*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *unique-factorization-MP-Rel* [*transfer-rule*]: (*MP-Rel* ===> *MF-Rel* ===> (=))
unique-factorization-m (*unique-factorization Irr-Mon*)
⟨*proof*⟩

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*)
⟨*proof*⟩ **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*)

⟨*proof*⟩

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*⟩

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*⟩
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* ≠ *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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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* (λ*a 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*⟩

lemma *coeffs-minus*: *poly-of-list (coeffs-minus f1 f0) = poly-of-list f1 − poly-of-list f0*
⟨*proof*⟩

lemma *karatsuba-main*: *karatsuba-main f n g m = poly-of-list f ∗ poly-of-list g*
⟨*proof*⟩

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*⟩

lemma *karatsuba-mult-poly-code-unfold*[*code-unfold*]: *(∗) = karatsuba-mult-poly*
⟨*proof*⟩

The following declaration will resolve a race-conflict between *(∗) = karatsuba-mult-poly* and *monom 1 ?n ∗ ?f = monom-mult ?n ?f*
?f ∗ monom 1 ?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* ≡ *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*
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*)))
⟨*proof*⟩

lemma *coeffs-sdiv*: *coeffs* (*sdiv-poly p a*) = (*strip-while* ((=) *0*) (*map* (λ *x*. *x div a*) (*coeffs p*)))
⟨*proof*⟩

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
⟨*proof*⟩

lemma *poly-rel-inj*: *poly-rel x y* ⟹ *poly-rel x z* ⟹ *y = z*
⟨*proof*⟩

lemma *bi-unique-poly-rel*[*transfer-rule*]: *bi-unique poly-rel*
⟨*proof*⟩

lemma *Domainp-is-poly* [*transfer-domain-rule*]:
Domainp poly-rel = is-poly ops
⟨*proof*⟩

lemma *poly-rel-zero*[*transfer-rule*]: *poly-rel zero-poly-i 0*
⟨*proof*⟩

lemma *poly-rel-one*[*transfer-rule*]: *poly-rel* (*one-poly-i ops*) *1*
⟨*proof*⟩

lemma *poly-rel-cCons*[*transfer-rule*]: (*R* ===> *list-all2 R* ===> *list-all2 R*) (*cCons-i ops*) *cCons*
⟨*proof*⟩

lemma *poly-rel-pCons*[*transfer-rule*]: (*R* ===> *poly-rel* ===> *poly-rel*) (*cCons-i ops*) *pCons*
⟨*proof*⟩

lemma *poly-rel-eq*[*transfer-rule*]: (*poly-rel* ===> *poly-rel* ===> (=)) (=) (=)
⟨*proof*⟩

lemma *poly-rel-plus*[*transfer-rule*]: (*poly-rel* ===> *poly-rel* ===> *poly-rel*) (*plus-poly-i ops*) (+)
⟨*proof*⟩

lemma *poly-rel-uminus*[*transfer-rule*]: (*poly-rel* ===> *poly-rel*) (*uminus-poly-i ops*) *Groups.uminus*
⟨*proof*⟩

lemma *poly-rel-minus*[*transfer-rule*]: (*poly-rel* ===> *poly-rel* ===> *poly-rel*) (*minus-poly-i ops*) (−)
⟨*proof*⟩

lemma *poly-rel-smult*[*transfer-rule*]: (*R* ===> *poly-rel* ===> *poly-rel*) (*smult-i ops*) *smult*
⟨*proof*⟩

lemma *poly-rel-coeffs*[*transfer-rule*]: (*poly-rel* ===> *list-all2 R*) (λ *x*. *x*) *coeffs*
⟨*proof*⟩

lemma *poly-rel-poly-of-list*[*transfer-rule*]: (*list-all2 R* ===> *poly-rel*) (*poly-of-list-i ops*) *poly-of-list*
⟨*proof*⟩

lemma *poly-rel-monom-mult*[*transfer-rule*]:
((=) ===> *poly-rel* ===> *poly-rel*) (*monom-mult-i ops*) *monom-mult*
⟨*proof*⟩

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*⟩

lemma *poly-rel-karatsuba-main*: *list-all2 R x1 x2* ⟹ *list-all2 R y1 y2* ⟹
poly-rel (*karatsuba-main-i ops x1 n y1 m*) (*karatsuba-main x2 n y2 m*)
⟨*proof*⟩

lemma *poly-rel-times*[*transfer-rule*]: (*poly-rel* ===> *poly-rel* ===> *poly-rel*) (*times-poly-i ops*) ((∗))
⟨*proof*⟩

lemma *poly-rel-coeff*[*transfer-rule*]: (*poly-rel* ===> (=) ===> *R*) (*coeff-i ops*) *coeff*
⟨*proof*⟩

lemma *poly-rel-degree*[*transfer-rule*]: (*poly-rel* ===> (=)) *degree-i degree*
⟨*proof*⟩

lemma *lead-coeff-i-def'*: *lead-coeff-i ops x* = (*coeff-i ops*) *x* (*degree-i x*)
⟨*proof*⟩

lemma *poly-rel-lead-coeff*[*transfer-rule*]: (*poly-rel* ===> *R*) (*lead-coeff-i ops*) *lead-coeff*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *mod-poly-one-main-i*: **assumes** *len*: *n* ≤ *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*)
⟨*proof*⟩

lemma *poly-rel-dvd*[*transfer-rule*]: (*poly-rel* ===> *poly-rel* ===> (=)) (*dvd-poly-i ops*) (*dvd*)
⟨*proof*⟩

lemma *poly-rel-monic*[*transfer-rule*]: (*poly-rel* ===> (=)) (*monic-i ops*) *monic*
⟨*proof*⟩

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*⟩

lemma *ring-ops-poly*: *ring-ops* (*poly-ops ops*) *poly-rel*
⟨*proof*⟩
end

context *idom-ops*
begin

lemma *poly-rel-pderiv* [*transfer-rule*]: (*poly-rel* ===> *poly-rel*) (*pderiv-i ops*) *pderiv*
⟨*proof*⟩

lemma *poly-rel-irreducible*[*transfer-rule*]: (*poly-rel* ===> (=)) (*irreducible-i ops*) $irreducible_d$
⟨*proof*⟩

lemma *idom-ops-poly*: *idom-ops* (*poly-ops ops*) *poly-rel*
⟨*proof*⟩
end

context *idom-divide-ops*
begin

lemma *poly-rel-sdiv*[*transfer-rule*]: (*poly-rel* ===> *R* ===> *poly-rel*) (*sdiv-i ops*) *sdiv-poly*
⟨*proof*⟩
end

context *field-ops*
begin

lemma *poly-rel-div*[*transfer-rule*]: (*poly-rel* ===> *poly-rel* ===> *poly-rel*)
(*div-field-poly-i ops*) (*div*)
⟨*proof*⟩

lemma *poly-rel-mod*[*transfer-rule*]: (*poly-rel* ===> *poly-rel* ===> *poly-rel*)
(*mod-field-poly-i ops*) (*mod*)
⟨*proof*⟩

lemma *poly-rel-normalize* [*transfer-rule*]: (*poly-rel* ===> *poly-rel*)
(*normalize-poly-i ops*) *Rings.normalize*
⟨*proof*⟩

lemma *poly-rel-unit-factor* [*transfer-rule*]: (*poly-rel* ===> *poly-rel*)
(*unit-factor-poly-i ops*) *Rings.unit-factor*
⟨*proof*⟩

lemma *idom-divide-ops-poly*: *idom-divide-ops* (*poly-ops ops*) *poly-rel*
⟨*proof*⟩

lemma *euclidean-ring-ops-poly*: *euclidean-ring-ops* (*poly-ops ops*) *poly-rel*
⟨*proof*⟩

lemma *poly-rel-gcd* [*transfer-rule*]: (*poly-rel* ===> *poly-rel* ===> *poly-rel*) (*gcd-poly-i ops*) *gcd*
⟨*proof*⟩

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*⟩

end

context *ring-ops*
begin
notepad
begin
⟨*proof*⟩
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* ⟨*proof*⟩

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*
⟨*proof*⟩
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*) ⟨*proof*⟩

sublocale *poly-mod-type p TYPE*(*'a*)
⟨*proof*⟩

lemma *coeffs-to-int-poly*: *coeffs* (*to-int-poly* (*x* :: *'a mod-ring poly*)) = *map to-int-mod-ring* (*coeffs x*)
⟨*proof*⟩

lemma *coeffs-of-int-poly*: *coeffs* (*of-int-poly* (*Mp x*) :: *'a mod-ring poly*) = *map of-int* (*coeffs* (*Mp x*))
⟨*proof*⟩

lemma *to-int-poly-i*: **assumes** *poly-rel f g* **shows** *to-int-poly-i ff-ops f* = *to-int-poly g*
⟨*proof*⟩

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''* ⟨*proof*⟩

sublocale *arith-ops-record ff-ops p* ⟨*proof*⟩

lemma *Mp-rel-iI*: *poly-rel f1 f2* ⟹ *MP-Rel f3 f2* ⟹ *Mp-rel-i f1 f3*
⟨*proof*⟩

lemma *M-rel-iI*: *R f1 f2* ⟹ *M-Rel f3 f2* ⟹ *M-rel-i f1 f3*
⟨*proof*⟩

lemma *M-rel-iI'*: **assumes** *R f1 f2*
shows *M-rel-i f1* (*arith-ops-record.to-int ff-ops f1*)
⟨*proof*⟩

lemma *Mp-rel-iI'*: **assumes** *poly-rel f1 f2*
shows *Mp-rel-i f1* (*to-int-poly-i ff-ops f1*)
⟨*proof*⟩

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*⟩

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*⟩
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*)
⟨*proof*⟩

end

lemma (**in** *mod-ring-locale*) *mod-ring-rel-of-int*:
0 ≤ *x* ⟹ *x* < *p* ⟹ *mod-ring-rel x* (*of-int x*)
⟨*proof*⟩

context *prime-field*
begin

lemma *prime-field-finite-field-ops-int*: *prime-field-gen* (*finite-field-ops-int p*) *mod-ring-rel p*
⟨*proof*⟩

lemma *prime-field-finite-field-ops-integer*: *prime-field-gen* (*finite-field-ops-integer* (*integer-of-int p*)) *mod-ring-rel-integer p*
⟨*proof*⟩

lemma *prime-field-finite-field-ops32*: **assumes** *small*: *p* ≤ *65535*

shows *prime-field-gen* (*finite-field-ops32* (*uint32-of-int p*)) *mod-ring-rel32 p*
⟨*proof*⟩

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*⟩
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*⟩

lemma *mod-ring-finite-field-ops-integer*: *mod-ring-gen* (*finite-field-ops-integer* (*integer-of-int p*)) *mod-ring-rel-integer p*
⟨*proof*⟩

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*⟩

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*⟩
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-Irreducibility
Polynomial-Interpolation.Missing-Polynomial
begin

lemma *cong-add-poly*:
$[(a::'b::\{field\text{-}gcd\}\ poly) = b]\ (mod\ m) \Longrightarrow [c = d]\ (mod\ m) \Longrightarrow [a + c = b + d]\ (mod\ m)$
⟨*proof*⟩

lemma *cong-mult-poly*:
$[(a::'b::\{field\text{-}gcd\}\ poly) = b]\ (mod\ m) \Longrightarrow [c = d]\ (mod\ m) \Longrightarrow [a * c = b * d]\ (mod\ m)$

⟨*proof*⟩

lemma *cong-mult-self-poly*: [(*a*::'*b*::{*field-gcd*} *poly*) ∗ *m* = *0*] (*mod m*)
⟨*proof*⟩

lemma *cong-scalar2-poly*: [(*a*::'*b*::{*field-gcd*} *poly*)= *b*] (*mod m*) ⟹ [*k* ∗ *a* = *k* ∗ *b*] (*mod m*)
⟨*proof*⟩

lemma *cong-sum-poly*:
(⋀*x*. *x* ∈ *A* ⟹ [((*f x*)::'*b*::{*field-gcd*} *poly*) = *g x*] (*mod m*)) ⟹
[(∑ *x*∈*A*. *f x*) = (∑ *x*∈*A*. *g x*)] (*mod m*)
⟨*proof*⟩

lemma *cong-iff-lin-poly*: ([(*a*::'*b*::{*field-gcd*} *poly*) = *b*] (*mod m*)) = (∃ *k*. *b* = *a* + *m* ∗ *k*)
⟨*proof*⟩

lemma *cong-solve-poly*: (*a*::'*b*::{*field-gcd*} *poly*) ≠ *0* ⟹ ∃ *x*. [*a* ∗ *x* = *gcd a n*] (*mod n*)
⟨*proof*⟩

lemma *cong-solve-coprime-poly*:
assumes *coprime-an*:*coprime* (*a*::'*b*::{*field-gcd*} *poly*) *n*
shows ∃ *x*. [*a* ∗ *x* = *1*] (*mod n*)
⟨*proof*⟩

lemma *cong-dvd-modulus-poly*:
[*x* = *y*] (*mod m*) ⟹ *n dvd m* ⟹ [*x* = *y*] (*mod n*) **for** *x y* :: '*b*::{*field-gcd*} *poly*
⟨*proof*⟩

lemma *chinese-remainder-aux-poly*:
fixes *A* :: '*a set*
and *m* :: '*a* ⇒ '*b*::{*field-gcd*} *poly*
assumes *fin*: *finite A*
and *cop*: ∀ *i* ∈ *A*. (∀ *j* ∈ *A*. *i* ≠ *j* ⟶ *coprime* (*m i*) (*m j*))
shows ∃ *b*. (∀ *i* ∈ *A*. [*b i* = *1*] (*mod m i*) ∧ [*b i* = *0*] (*mod* (∏ *j* ∈ *A* − {*i*}. *m j*)))
⟨*proof*⟩

lemma *chinese-remainder-poly*:
fixes *A* :: '*a set*
and *m* :: '*a* ⇒ '*b*::{*field-gcd*} *poly*
and *u* :: '*a* ⇒ '*b poly*
assumes *fin*: *finite A*
and *cop*: ∀ *i*∈*A*. (∀ *j*∈*A*. *i* ≠ *j* ⟶ *coprime* (*m i*) (*m j*))

shows $\exists x.\ (\forall i{\in}A.\ [x = u\ i]\ (mod\ m\ i))$
⟨*proof*⟩

lemma *cong-trans-poly*:
$[(a::'b::\{field\text{-}gcd\}\ poly) = b]\ (mod\ m) \Longrightarrow [b = c]\ (mod\ m) \Longrightarrow [a = c]\ (mod\ m)$
⟨*proof*⟩

lemma *cong-mod-poly*: $(n::'b::\{field\text{-}gcd\}\ poly)\ ^\sim= 0 \Longrightarrow [a\ mod\ n = a]\ (mod\ n)$
⟨*proof*⟩

lemma *cong-sym-poly*: $[(a::'b::\{field\text{-}gcd\}\ poly) = b]\ (mod\ m) \Longrightarrow [b = a]\ (mod\ m)$
⟨*proof*⟩

lemma *cong-1-poly*: $[(a::'b::\{field\text{-}gcd\}\ poly) = b]\ (mod\ 1)$
⟨*proof*⟩

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 $[a = b]\ (mod\ m * n)$
⟨*proof*⟩

lemma *coprime-cong-prod-poly*:
$(\forall i{\in}A.\ (\forall j{\in}A.\ i \neq j \longrightarrow coprime\ (m\ i)\ (m\ j))) \Longrightarrow$
$(\forall i{\in}A.\ [(x::'b::\{field\text{-}gcd\}\ poly) = y]\ (mod\ m\ i)) \Longrightarrow$
$[x = y]\ (mod\ (\prod i{\in}A.\ m\ i))$
⟨*proof*⟩

lemma *cong-less-modulus-unique-poly*:
$[(x::'b::\{field\text{-}gcd\}\ poly) = y]\ (mod\ m) \Longrightarrow degree\ x < degree\ m \Longrightarrow degree\ y < degree\ m \Longrightarrow x = y$
⟨*proof*⟩

lemma *chinese-remainder-unique-poly*:
fixes $A :: 'a\ set$
and $m :: 'a \Rightarrow 'b::\{field\text{-}gcd\}\ poly$
and $u :: 'a \Rightarrow 'b\ poly$
assumes *nz*: $\forall i{\in}A.\ (m\ i) \neq 0$
and *cop*: $\forall i{\in}A.\ (\forall j{\in}A.\ i \neq j \longrightarrow coprime\ (m\ i)\ (m\ j))$

and *not-constant*: $0 < degree\ (prod\ m\ A)$
shows $\exists!x.\ degree\ x < (\sum i{\in}A.\ degree\ (m\ i)) \wedge (\forall i{\in}A.\ [x = u\ i]\ (mod\ m\ i))$
⟨*proof*⟩

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 $g :: 'b \Rightarrow 'a :: comm\text{-}monoid\text{-}mult$
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*)
⟨*proof*⟩

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*)
⟨*proof*⟩
end

lemma *coprime-id-is-unit*:
fixes *a*::'*b*::*semiring-gcd*
shows *coprime a a* ⟷ *is-unit a*
⟨*proof*⟩

lemma *dim-vec-of-list*[*simp*]: *dim-vec* (*vec-of-list x*) = *length x*
⟨*proof*⟩

lemma *length-list-of-vec*[*simp*]: *length* (*list-of-vec A*) = *dim-vec A*
⟨*proof*⟩

lemma *list-of-vec-vec-of-list*[*simp*]: *list-of-vec* (*vec-of-list a*) = *a*
⟨*proof*⟩

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*⟩

corollary *inj-Poly-list-of-vec*: *inj-on* (*Poly* ∘ *list-of-vec*) (*carrier-vec n*)
⟨*proof*⟩

lemma *list-of-vec-rw-map*: *list-of-vec m* = *map* (λ*n*. *m* $ *n*) [*0*..<*dim-vec m*]
⟨*proof*⟩

lemma *degree-Poly'*:
assumes *xs*: *xs* ≠ []
shows *degree* (*Poly xs*) < *length xs*
⟨*proof*⟩

lemma *vec-of-list-list-of-vec*[*simp*]: *vec-of-list* (*list-of-vec a*) = *a*
⟨*proof*⟩

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*⟩

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*⟩

lemma *list-of-vec-nth*:
assumes *i*: *i* < *dim-vec x*
shows *list-of-vec x* ! *i* = *x* $ *i*
⟨*proof*⟩

lemma *coeff-Poly-list-of-vec-nth'*:
assumes *i*: *i* < *dim-vec x*
shows *coeff* (*Poly* (*list-of-vec x*)) *i* = *x* $ *i*
⟨*proof*⟩

lemma *list-of-vec-row-nth*:
assumes *x*: *x* < *dim-col A*
shows *list-of-vec* (*row A i*) ! *x* = *A* $$ (*i*, *x*)
⟨*proof*⟩

lemma *coeff-Poly-list-of-vec-nth*:
assumes x: $x < dim\text{-}col\ A$
shows *coeff* (*Poly* (*list-of-vec* (*row A i*))) $x = A\ \$\$\ (i, x)$
⟨*proof*⟩

lemma *inj-on-list-of-vec*: *inj-on list-of-vec* (*carrier-vec n*)
⟨*proof*⟩

lemma *vec-of-list-carrier*[*simp*]: *vec-of-list x* ∈ *carrier-vec* (*length x*)
⟨*proof*⟩

lemma *card-carrier-vec*: *card* (*carrier-vec n*:: *'b*::*finite vec set*) = *CARD*(*'b*) ^ *n*
⟨*proof*⟩

lemma *finite-carrier-vec*[*simp*]: *finite* (*carrier-vec n*:: *'b*::*finite vec set*)
⟨*proof*⟩

lemma *row-echelon-form-dim0-row*:
assumes *A* ∈ *carrier-mat 0 n*
shows *row-echelon-form A*
⟨*proof*⟩

lemma *row-echelon-form-dim0-col*:
assumes *A* ∈ *carrier-mat n 0*
shows *row-echelon-form A*
⟨*proof*⟩

lemma *row-echelon-form-one-dim0*[*simp*]: *row-echelon-form* (1_m *0*)
⟨*proof*⟩

lemma *Poly-list-of-vec-0*[*simp*]: *Poly* (*list-of-vec* (0_v *0*)) = [:*0*:]
⟨*proof*⟩

lemma *monic-normalize*:
assumes (*p* :: *'b* :: {*field*,*euclidean-ring-gcd*} *poly*) ≠ *0* **shows** *monic* (*normalize p*)
⟨*proof*⟩

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*)

⟨*proof*⟩

lemma *normalize-eq-imp-smult*:
  fixes $p :: 'b :: \{euclidean\text{-}ring\text{-}gcd\}$ *poly*
  assumes *n*: *normalize p* = *normalize q*
  shows $\exists\ c.\ c \neq 0 \wedge q = smult\ c\ p$
⟨*proof*⟩

lemma *prod-list-normalize*:
  fixes $P :: 'b :: \{idom\text{-}divide, normalization\text{-}semidom\text{-}multiplicative\}$ *poly list*
  shows *normalize* (*prod-list P*) = *prod-list* (*map normalize P*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*))
⟨*proof*⟩

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*) = *Euclidean-Rings.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*⟩

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*⟩

lemma *mod-eq-dvd-iff-poly*: ((*x*::*'a mod-ring poly*) *mod n* = *y mod n*) = (*n dvd x* − *y*)
⟨*proof*⟩

lemma *cong-gcd-eq-poly*:
gcd a m = *gcd b m* **if** [(*a*::*'a mod-ring poly*) = *b*] (*mod m*)
⟨*proof*⟩

lemma *coprime-h-c-poly*:
fixes *h*::*'a mod-ring poly*
assumes *c1* ≠ *c2*
shows *coprime* (*h* − [:*c1*:]) (*h* − [:*c2*:])
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *degree-minus-eq-right*:
fixes *p*::*'b*::*ab-group-add poly*
shows *degree q* < *degree p* ⟹ *degree* (*p* − *q*) = *degree p*

⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *poly-identity-mod-p*:
v^(*CARD*(*'a*)) − *v* = *prod* (λ*x*. *v* − [:*x*:]) (*UNIV*::*'a mod-ring set*)
⟨*proof*⟩

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*:]))
⟨*proof*⟩

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*⟩

lemma *monic-prod-gcd*:
assumes *f*: *finite A* **and** *f0*: (*f* :: *'b* :: {*field-gcd*} *poly*) $\neq$ *0*
shows *monic* ($\prod$ *c*$\in$*A*. *gcd f* (*h* − [:*c*:]))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *divides-prod2*:
  fixes *A*::*'b*::*semiring-gcd set*
  assumes *f*: *finite A*
  and $\forall$ *a*$\in$*A*. *a dvd c*
  and $\forall$ *a1 a2*. *a1* $\in$ *A* $\wedge$ *a2* $\in$ *A* $\wedge$ *a1* $\neq$ *a2* $\longrightarrow$ *coprime a1 a2*
  shows $\prod$ *A dvd c*
⟨*proof*⟩

lemma *coprime-polynomial-factorization*:
  fixes *a1* :: *'b* :: {*field-gcd*} *poly*
  assumes *irr*: *as* $\subseteq$ {*q*. *irreducible q* $\wedge$ *monic q*}
  and *finite as* **and** *a1*: *a1* $\in$ *as* **and** *a2*: *a2* $\in$ *as* **and** *a1-not-a2*: *a1* $\neq$ *a2*
  shows *coprime a1 a2*
⟨*proof*⟩

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*⟩

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)))
  ⟨proof⟩

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
⟨proof⟩


lemma length-power-polys[simp]: length (power-polys mult-p u curr-p n) = n
```

⟨*proof*⟩

lemma *Poly-berlekamp-mat*:
assumes *k*: $k < degree\ u$
shows $Poly\ (list\text{-}of\text{-}vec\ (row\ (berlekamp\text{-}mat\ u)\ k)) = [:0,1:]\hat{}(CARD('a) * k)\ mod\ u$
⟨*proof*⟩

corollary *Poly-berlekamp-cong-mat*:
assumes *k*: $k < degree\ u$
shows $[Poly\ (list\text{-}of\text{-}vec\ (row\ (berlekamp\text{-}mat\ u)\ k)) = [:0,1:]\hat{}(CARD('a) * k)]\ (mod\ u)$
⟨*proof*⟩

lemma *mat-of-rows-list-dim*[*simp*]:
$mat\text{-}of\text{-}rows\text{-}list\ n\ vs \in carrier\text{-}mat\ (length\ vs)\ n$
$dim\text{-}row\ (mat\text{-}of\text{-}rows\text{-}list\ n\ vs) = length\ vs$
$dim\text{-}col\ (mat\text{-}of\text{-}rows\text{-}list\ n\ vs) = n$
⟨*proof*⟩

lemma *berlekamp-mat-closed*[*simp*]:
$berlekamp\text{-}mat\ u \in carrier\text{-}mat\ (degree\ u)\ (degree\ u)$
$dim\text{-}row\ (berlekamp\text{-}mat\ u) = degree\ u$
$dim\text{-}col\ (berlekamp\text{-}mat\ u) = degree\ u$
⟨*proof*⟩

lemma *vec-of-list-coeffs-nth*:
assumes *i*: $i \in \{..degree\ h\}$ **and** *h-not0*: $h \neq 0$
shows $vec\text{-}of\text{-}list\ (coeffs\ h)\ \$\ i = coeff\ h\ i$
⟨*proof*⟩

lemma *poly-mod-sum*:
fixes *x y z* :: *'b*::*field poly*
assumes *f*: *finite A*
shows $sum\ f\ A\ mod\ z = sum\ (\lambda i.\ f\ i\ mod\ z)\ A$
⟨*proof*⟩

lemma *prime-not-dvd-fact*:
assumes *kn*: $k < n$ **and** *prime-n*: *prime n*
shows $\neg\ n\ dvd\ fact\ k$
⟨*proof*⟩

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*⟩

lemma *add-power-poly-mod-ring*:
fixes $x :: 'a$ *mod-ring poly*
shows $(x + y) \hat{} CARD('a) = x \hat{} CARD('a) + y \hat{} CARD('a)$
⟨*proof*⟩

lemma *power-poly-sum-mod-ring*:
fixes $f :: 'b \Rightarrow 'a$ *mod-ring poly*
assumes *f*: *finite A*
shows $(sum\ f\ A) \hat{} CARD('a) = sum\ (\lambda i.\ (f\ i) \hat{} CARD('a))\ A$
⟨*proof*⟩

lemma *poly-power-card-as-sum-of-monoms*:
fixes $h :: 'a$ *mod-ring poly*
shows $h \hat{} CARD('a) = (\sum i \leq degree\ h.\ monom\ (coeff\ h\ i)\ (CARD('a) * i))$
⟨*proof*⟩

lemma *degree-Poly-berlekamp-le*:
assumes *i*: $i < degree\ u$
shows $degree\ (Poly\ (list\text{-}of\text{-}vec\ (row\ (berlekamp\text{-}mat\ u)\ i))) < degree\ u$
⟨*proof*⟩

lemma *monom-card-pow-mod-sum-berlekamp*:
assumes *i*: $i < degree\ u$
shows $monom\ 1\ (CARD('a) * i)\ mod\ u = (\sum j < degree\ u.\ monom\ ((berlekamp\text{-}mat\ u)\ \$\$\ (i,j))\ j)$
⟨*proof*⟩

lemma *col-scalar-prod-as-sum*:
assumes *dim-vec v* = *dim-row A*
shows $col\ A\ j \cdot v = (\sum i = 0..<dim\text{-}vec\ v.\ A\ \$\$\ (i,j) * v\ \$\ i)$
⟨*proof*⟩

lemma *row-transpose-scalar-prod-as-sum*:

assumes *j*: $j < dim\text{-}col\ A$ **and** *dim-v*: $dim\text{-}vec\ v = dim\text{-}row\ A$
shows $row\ (transpose\text{-}mat\ A)\ j \cdot v = (\sum i = 0..<dim\text{-}vec\ v.\ A\ \$\$\ (i,j) * v\ \$\ i)$
⟨*proof*⟩

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*⟩

lemma *dim-vec-of-list-h*:
assumes $degree\ h < degree\ u$
shows $dim\text{-}vec\ (vec\text{-}of\text{-}list\ ((coeffs\ h)\ @\ replicate\ (degree\ u - length\ (coeffs\ h))\ 0))$
$= degree\ u$
⟨*proof*⟩

lemma *vec-of-list-coeffs-nth'*:
assumes *i*: $i \in \{..degree\ h\}$ **and** *h-not0*: $h \neq 0$
assumes $degree\ h < degree\ u$
shows $vec\text{-}of\text{-}list\ ((coeffs\ h)\ @\ replicate\ (degree\ u - length\ (coeffs\ h))\ 0)\ \$\ i = coeff\ h\ i$
⟨*proof*⟩

lemma *vec-of-list-coeffs-replicate-nth-0*:
assumes *i*: $i \in \{..<degree\ u\}$
shows $vec\text{-}of\text{-}list\ (coeffs\ 0\ @\ replicate\ (degree\ u - length\ (coeffs\ 0))\ 0)\ \$\ i = coeff\ 0\ i$
⟨*proof*⟩

lemma *vec-of-list-coeffs-replicate-nth*:
assumes *i*: $i \in \{..<degree\ u\}$
assumes $degree\ h < degree\ u$
shows $vec\text{-}of\text{-}list\ ((coeffs\ h)\ @\ replicate\ (degree\ u - length\ (coeffs\ h))\ 0)\ \$\ i = coeff\ h\ i$
⟨*proof*⟩

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 [$h \hat{} CARD('a) = h$] (*mod u*) $\longleftrightarrow$ (*transpose-mat* (*berlekamp-mat u*)) $*_v$ H = H
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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* ∧ *monic q*$\}$
and *fi-P*: *fi* ∈ *P*
and *h*: $h \in \{v.\ [v \hat{} (CARD('a)) = v]\ (mod\ f)\}$
shows ∃ *s*. *fi dvd* (*h* − [:*s*:])
⟨*proof*⟩

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* ∧ *monic q*$\}$
and *fi-P*: *fi* ∈ *P*
and *h*: $h \in \{v.\ [v \hat{} (CARD('a)) = v]\ (mod\ f)\}$
shows ∃!*s*. *fi dvd* (*h* − [:*s*:])
⟨*proof*⟩

lemma *exists-two-distint*: ∃ *a b*::'*a mod-ring*. *a* ≠ *b*
⟨*proof*⟩

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*: ∀ *p*∈*P*. [*a*=*b*] (*mod p*)
and *coprime-P*: ∀ *p1 p2*. *p1* ∈ *P* ∧ *p2* ∈ *P* ∧ *p1* ≠ *p2* ⟶ *coprime p1 p2*

shows $[a = b]\ (mod\ (\prod a{\in}P.\ a))$
⟨*proof*⟩

end

context
assumes *SORT-CONSTRAINT*('a::*prime-card*)
begin

lemma *W-eq-berlekamp-mat*:
fixes *u*::'a *mod-ring poly*
shows {*v*. [*v*^*CARD*('a) = *v*] (*mod u*) ∧ *degree v* < *degree u*}
= {*h*. *let H* = *vec-of-list* ((*coeffs h*) @ *replicate* (*degree u* − *length* (*coeffs h*)) *0*)
in
(*transpose-mat* (*berlekamp-mat u*)) $*_v$ *H* = *H* ∧ *degree h* < *degree u*}
⟨*proof*⟩

lemma *transpose-minus-1*:
assumes *dim-row Q* = *dim-col Q*
shows *transpose-mat* (*Q* − (*1*$_m$ (*dim-row Q*))) = (*transpose-mat Q* − (*1*$_m$ (*dim-row Q*)))
⟨*proof*⟩

lemma *system-iff*:
fixes *v*::'b::*comm-ring-1 vec*
assumes *sq-Q*: *dim-row Q* = *dim-col Q* **and** *v*: *dim-row Q* = *dim-vec v*
shows (*transpose-mat Q* $*_v$ *v* = *v*) ⟷ ((*transpose-mat Q* − *1*$_m$ (*dim-row Q*)) $*_v$ *v* = *0*$_v$ (*dim-vec v*))
⟨*proof*⟩

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*⟩

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*⟩

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*]: $(\odot_{poly\text{-}abelian\text{-}monoid}) = smult$
⟨*proof*⟩

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*⟩

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*⟩

lemma *berlekamp-resulting-mat-basis*:
kernel.basis (*degree u*) (*berlekamp-resulting-mat u*) (*set* (*find-base-vectors* (*berlekamp-resulting-mat u*)))
⟨*proof*⟩

lemma *set-berlekamp-basis-eq*: (*set* (*berlekamp-basis u*))
= (*Poly* ∘ *list-of-vec*)‘ (*set* (*find-base-vectors* (*berlekamp-resulting-mat u*)))
⟨*proof*⟩

lemma *berlekamp-resulting-mat-constant*:
assumes *deg-u*: *degree u* = *0*
shows *berlekamp-resulting-mat u* = 1_m *0*
⟨*proof*⟩

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*⟩

lemma *row-echelon-form-berlekamp-resulting-mat*: *row-echelon-form* (*berlekamp-resulting-mat u*)
⟨*proof*⟩

lemma *mat-kernel-berlekamp-resulting-mat-degree-0*:
assumes *d*: *degree u = 0*
shows *mat-kernel* (*berlekamp-resulting-mat u*) = {0_v 0}
⟨*proof*⟩

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*⟩ **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*⟩

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*⟩

lemma *linear-Poly-list-of-vec'*:
assumes *degree u > 0*
shows (*Poly* ∘ *list-of-vec*) ∈ *module-hom R V W*
⟨*proof*⟩

lemma *berlekamp-basis-eq-8*:
assumes *v*: *v* ∈ *set* (*berlekamp-basis u*)
shows [*v* ^ *CARD*(*'a*) = *v*] (*mod u*)
⟨*proof*⟩

lemma *surj-Poly-list-of-vec*:
  assumes *deg-u*: *degree u > 0*
  shows *(Poly ∘ list-of-vec)‘ (carrier V) = carrier W*
⟨*proof*⟩

lemma *card-set-berlekamp-basis*: *card (set (berlekamp-basis u)) = length (berlekamp-basis u)*
⟨*proof*⟩

context
  assumes *deg-u0*[*simp*]: *degree u > 0*
begin

interpretation *Berlekamp-subspace*: *vectorspace class-ring W*
  ⟨*proof*⟩

lemma *linear-map-Poly-list-of-vec'*: *linear-map class-ring V W (Poly ∘ list-of-vec)*
⟨*proof*⟩

lemma *berlekamp-basis-basis*:
  Berlekamp-subspace.basis (set (berlekamp-basis u))
⟨*proof*⟩

lemma *finsum-sum*:
fixes *f*::*'a mod-ring poly*
assumes *f*: *finite B*
and *a-Pi*: *a ∈ B → carrier R*
and *V*: *B ⊆ carrier W*
shows *(⨁ W v∈B. a v ⊙ W v) = sum (λv. smult (a v) v) B*
⟨*proof*⟩

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 = (∏ 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. v ∈ {h. [h^(CARD('a)) = h] (mod u) ∧ degree h < degree u}*
  ∧ 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:
  $\wedge$ ($\exists$ *s*. *gcd w* (*v* − [:*s*:]) ≠ *w* $\wedge$ *gcd w* (*v* − [:*s*:]) ≠ *1*)
⟨*proof*⟩

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* = (∏ *a*∈*P*. *a*)
      and *P*: *P* ⊆ {*q*. *irreducible q* $\wedge$ *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*
⟨*proof*⟩

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* $\wedge$ *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*
  $\wedge$ *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*:]) ≠ *w* $\wedge$ ¬ *coprime w* (*v* − [:*s*:]))
⟨*proof*⟩

lemma *exists-bijective-linear-map-W-vec*:
  assumes *finite-P*: *finite P*
      and *u-desc-square-free*: *u* = (∏ *a*∈*P*. *a*)
      and *P*: *P* ⊆ {*q*. *irreducible q* $\wedge$ *monic q*}
  shows ∃ *f*. *linear-map class-ring W* (*module-vec TYPE*('*a mod-ring*) (*card P*)) *f*
  $\wedge$ *bij-betw f* (*carrier W*) (*carrier-vec* (*card P*)::'*a mod-ring vec set*)
⟨*proof*⟩

lemma *fin-dim-kernel-berlekamp*: *V.fin-dim*

⟨*proof*⟩

lemma *Berlekamp-subspace-fin-dim*: *Berlekamp-subspace.fin-dim*
⟨*proof*⟩

context
  fixes *P*
  assumes *finite-P*: *finite P*
  and *u-desc-square-free*: $u = (\prod a{\in}P.\ a)$
  and *P*: $P \subseteq \{q.\ irreducible\ q \wedge monic\ q\}$
begin

interpretation *RV*: *vec-space TYPE*('*a mod-ring*) *card P* ⟨*proof*⟩

lemma *Berlekamp-subspace-eq-dim-vec*: *Berlekamp-subspace.dim* = *RV.dim*
⟨*proof*⟩

lemma *Berlekamp-subspace-dim*: *Berlekamp-subspace.dim* = *card P*
  ⟨*proof*⟩

corollary *card-berlekamp-basis-number-factors*: *card* (*set* (*berlekamp-basis u*)) = *card P*
  ⟨*proof*⟩

lemma *length-berlekamp-basis-numbers-factors*: *length* (*berlekamp-basis u*) = *card P*
  ⟨*proof*⟩

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*: $\bigwedge u.\ u \in set\ us \Longrightarrow degree\ u > 0$
  shows *length us* ≤ *n*

⟨*proof*⟩

lemma *berlekamp-basis-irreducible*: **assumes** *f*: $f = prod\text{-}list\ us$
and *n-us*: $length\ us = n$
and *us*: $\bigwedge u.\ u \in set\ us \Longrightarrow degree\ u > 0$
and *u*: $u \in set\ us$
shows *irreducible u*
⟨*proof*⟩
end

lemma *not-irreducible-factor-yields-prime-factors*:
assumes *uf*: *u dvd* $(f :: 'b :: \{field\text{-}gcd\}\ poly)$ **and** *fin*: *finite P*
and *fP*: $f = \prod P$ **and** *P*: $P \subseteq \{q.\ irreducible\ q \wedge monic\ q\}$
and *u*: $degree\ u > 0\ \neg\ irreducible\ u$
shows $\exists\ pi\ pj.\ pi \in P \wedge pj \in P \wedge pi \neq pj \wedge pi\ dvd\ u \wedge pj\ dvd\ u$
⟨*proof*⟩

lemma *berlekamp-factorization-main*:
fixes $f::'a\ mod\text{-}ring\ poly$
assumes *sf-f*: *square-free f*
and *vs*: $vs = vs1\ @\ vs2$
and *vsf*: $vs = berlekamp\text{-}basis\ f$
and *n-bb*: $n = length\ (berlekamp\text{-}basis\ f)$
and *n*: $n = length\ us1 + n2$
and *us*: $us = us1\ @\ berlekamp\text{-}factorization\text{-}main\ d\ divs\ vs2\ n2$
and *us1*: $\bigwedge u.\ u \in set\ us1 \Longrightarrow monic\ u \wedge irreducible\ u$
and *divs*: $\bigwedge d.\ d \in set\ divs \Longrightarrow monic\ d \wedge degree\ d > 0$
and *vs1*: $\bigwedge u\ v\ i.\ v \in set\ vs1 \Longrightarrow u \in set\ us1 \cup set\ divs$
$\Longrightarrow i < CARD('a) \Longrightarrow gcd\ u\ (v - [:of\text{-}nat\ i:]) \in \{1,u\}$
and *f*: $f = prod\text{-}list\ (us1\ @\ divs)$
and *deg-f*: $degree\ f > 0$
and *d*: $\bigwedge g.\ g\ dvd\ f \Longrightarrow degree\ g = d \Longrightarrow irreducible\ g$
shows $f = prod\text{-}list\ us \wedge (\forall\ u \in set\ us.\ monic\ u \wedge irreducible\ u)$
⟨*proof*⟩

lemma *berlekamp-monic-factorization*:
fixes $f::'a\ mod\text{-}ring\ poly$
assumes *sf-f*: *square-free f*
and *us*: $berlekamp\text{-}monic\text{-}factorization\ d\ f = us$
and *d*: $\bigwedge g.\ g\ dvd\ f \Longrightarrow degree\ g = d \Longrightarrow irreducible\ g$
and *deg*: $degree\ f > 0$
and *mon*: *monic f*
shows $f = prod\text{-}list\ us \wedge (\forall\ u \in set\ us.\ monic\ u \wedge irreducible\ u)$
⟨*proof*⟩
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*⟩

hide-const *order*
hide-const *up-ring.monom*

theorem (**in** *field*) *finite-field-mult-group-has-gen2*:
assumes *finite*:*finite* (*carrier R*)
shows ∃ *a* ∈ *carrier* (*mult-of R*). *group.ord* (*mult-of R*) *a* = *order* (*mult-of R*)
∧ *carrier* (*mult-of R*) = {*a*[^]*i* | *i*::*nat* . *i* ∈ *UNIV*}
⟨*proof*⟩

lemma *add-power-prime-poly-mod-ring*[*simp*]:
fixes *x* :: *'a*::{*prime-card*} *mod-ring poly*
shows (*x* + *y*) ^ *CARD*(*'a*)^*n* = *x* ^ (*CARD*(*'a*)^*n*) + *y* ^ *CARD*(*'a*)^*n*
⟨*proof*⟩

lemma *fermat-theorem-mod-ring2*[*simp*]:
fixes *a*::*'a*::{*prime-card*} *mod-ring*
shows *a* ^ (*CARD*(*'a*)^*n*) = *a*
⟨*proof*⟩

lemma *fermat-theorem-power-poly*[*simp*]:
fixes *a*::*'a*::*prime-card mod-ring*
shows [:*a*:] ^ *CARD*(*'a*::*prime-card*) ^ *n* = [:*a*:]
⟨*proof*⟩

lemma *degree-prod-monom*: *degree* (∏ *i* = *0*..<*n*. *monom 1 1*) = *n*

⟨*proof*⟩

lemma *degree-monom0*[*simp*]: *degree* (*monom a 0*) = *0* ⟨*proof*⟩
lemma *degree-monom0'*[*simp*]: *degree* (*monom 0 b*) = *0* ⟨*proof*⟩

lemma *sum-monom-mod*:
  assumes *b* < *degree f*
  shows (∑ *i*≤*b*. *monom* (*g i*) *i*) *mod f* = (∑ *i*≤*b*. *monom* (*g i*) *i*)
  ⟨*proof*⟩

lemma *x-power-aq-minus-1-rw*:
  fixes *x*::*nat*
  assumes *x*: *x* > *1*
    and *a*: *a* > *0*
    and *b*: *b* > *0*
  shows *x* ^ (*a* * *q*) − *1* = ((*x*^*a*) − *1*) * *sum* ((^) (*x*^*a*)) {..<*q*}
⟨*proof*⟩

lemma *dvd-power-minus-1-conv1*:
  fixes *x*::*nat*
  assumes *x*: *x* > *1*
    and *a*: *a* > *0*
    and *xa-dvd*: *x* ^ *a* − *1* *dvd* *x*^*b* − *1*
    and *b0*: *b* > *0*
  shows *a* *dvd* *b*
⟨*proof*⟩

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*⟩

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*)
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *element-in-carrier*: (*a* ∈ *carrier R*) = (*degree a* < *degree f*)
  ⟨*proof*⟩

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*⟩

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*
  ⟨*proof*⟩

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 $\neg$ *q dvd p*
⟨*proof*⟩

lemma *times-mod-f-1-imp-0*:
assumes *x*: *degree x* < *degree f*
and *x2*: $\forall$ *xa*. *x* ∗ *xa mod f* = *1* $\longrightarrow$ $\neg$ *degree xa* < *degree f*
shows *x* = *0*
⟨*proof*⟩

sublocale *field-R*: *field R*
⟨*proof*⟩

lemma *zero-in-carrier*[*simp*]: *0* ∈ *carrier-irr* ⟨*proof*⟩

lemma *card-carrier-irr*[*simp*]: *card carrier-irr* = *CARD*('*a*)^(*degree f*)
⟨*proof*⟩

lemma *finite-carrier-irr*[*simp*]: *finite* (*carrier-irr*)
⟨*proof*⟩

lemma *finite-carrier-R*[*simp*]: *finite* (*carrier R*) ⟨*proof*⟩

lemma *finite-carrier-mult-of*[*simp*]: *finite* (*carrier* (*mult-of R*))
⟨*proof*⟩

lemma *constant-in-carrier*[*simp*]: [:*a*:] ∈ *carrier R*
⟨*proof*⟩

lemma *mod-in-carrier*[*simp*]: *a mod f* ∈ *carrier R*
⟨*proof*⟩

lemma *order-irr*: *Coset.order* (*mult-of R*) = *CARD*('*a*)^*degree f* − *1*
⟨*proof*⟩

lemma *element-power-order-eq-1*:
assumes *x*: *x* ∈ *carrier* (*mult-of R*)
shows *x* [^]$_{(mult\text{-}of\ R)}$ *Coset.order* (*mult-of R*) = **1**$_{(mult\text{-}of\ R)}$
⟨*proof*⟩

corollary *element-power-order-eq-1'*:
assumes *x*: *x* ∈ *carrier* (*mult-of R*)
shows*x* [^]$_{(mult\text{-}of\ R)}$ *CARD*('*a*)^*degree f* = *x*
⟨*proof*⟩

lemma *pow-irr*[*simp*]: *x* [^]$_{(R)}$ *n*= *x*^*n mod f*
⟨*proof*⟩

lemma *pow-irr-mult-of*[*simp*]: *x* [^]$_{(mult\text{-}of\ R)}$ *n*= *x*^*n mod f*

⟨*proof*⟩

lemma *fermat-theorem-power-poly-R*[*simp*]: $[:a:]\ [\hat{}]_R\ CARD('a)\ \hat{}\ n = [:a:]$
⟨*proof*⟩

lemma *times-mod-expand*:
$(a \otimes_{(R)} b) = ((a\ mod\ f) \otimes_{(R)} (b\ mod\ f))$
⟨*proof*⟩

lemma *mult-closed-power*:
assumes *x*: $x \in carrier\ R$ **and** *y*: $y \in carrier\ R$
and $x\ [\hat{}]_{(R)}\ CARD('a)\ \hat{}\ m' = x$
and $y\ [\hat{}]_{(R)}\ CARD('a)\ \hat{}\ m' = y$
shows $(x \otimes_{(R)} y)\ [\hat{}]_{(R)}\ CARD('a)\ \hat{}\ m' = (x \otimes_{(R)} y)$
⟨*proof*⟩

lemma *add-closed-power*:
assumes *x1*: $x\ [\hat{}]_{(R)}\ CARD('a)\ \hat{}\ m' = x$
and *y1*: $y\ [\hat{}]_{(R)}\ CARD('a)\ \hat{}\ m' = y$
shows $(x \oplus_{(R)} y)\ [\hat{}]_{(R)}\ CARD('a)\ \hat{}\ m' = (x \oplus_{(R)} y)$
⟨*proof*⟩

lemma *x-power-pm-minus-1*:
assumes *x*: $x \in carrier\ (mult\text{-}of\ R)$
and $x\ [\hat{}]_{(R)}\ CARD('a)\ \hat{}\ m' = x$
shows $x\ [\hat{}]_{(R)}\ (CARD('a)\ \hat{}\ m' - 1) = \mathbf{1}_{(R)}$
⟨*proof*⟩

context
begin

private lemma *monom-a-1-P*:
assumes *m*: $monom\ 1\ 1 \in carrier\ R$
and *eq*: $monom\ 1\ 1\ [\hat{}]_{(R)}\ (CARD('a)\ \hat{}\ m') = monom\ 1\ 1$
shows $monom\ a\ 1\ [\hat{}]_{(R)}\ (CARD('a)\ \hat{}\ m') = monom\ a\ 1$
⟨*proof*⟩ **lemma** *prod-monom-1-1*:
defines $P == (\lambda\ x\ n.\ (x[\hat{}]_{(R)}\ (CARD('a)\ \hat{}\ n) = x))$
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*⟩ **lemma** *monom-1-b*:
defines $P == (\lambda\ x\ n.\ (x[\hat{}]_{(R)}\ (CARD('a)\ \hat{}\ n) = x))$
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*⟩ **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*⟩ **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 $P\ ((\sum i{\leq}b.\ monom\ (g\ i)\ i))\ n$
⟨*proof*⟩

lemma *element-carrier-P*:
defines $P \equiv (\lambda\ x\ n.\ (x[\widehat{\ }]_{(R)}\ (CARD('a)\ \widehat{\ }\ n) = x))$
assumes *m*: $monom\ 1\ 1 \in carrier\ R$
and *monom-1-1*: $P\ (monom\ 1\ 1)\ m'$
and *a*: $a \in carrier\ R$
shows $P\ a\ m'$
⟨*proof*⟩
end

end

lemma *degree-divisor1*:
assumes *f*: *irreducible* $(f :: 'a :: prime\text{-}card\ mod\text{-}ring\ poly)$
and *d*: $degree\ f = d$
shows $f\ dvd\ (monom\ 1\ 1)\widehat{\ }(CARD('a)\widehat{\ }d) - monom\ 1\ 1$
⟨*proof*⟩

lemma *degree-divisor2*:
assumes *f*: *irreducible* $(f :: 'a :: prime\text{-}card\ mod\text{-}ring\ poly)$
and *d*: $degree\ f = d$
and *c-ge-1*: $1 \leq c$ **and** *cd*: $c < d$
shows $\neg\ f\ dvd\ monom\ 1\ 1\ \widehat{\ }\ CARD('a)\ \widehat{\ }\ c - monom\ 1\ 1$
⟨*proof*⟩

lemma *degree-divisor*: **assumes** *irreducible* $(f :: 'a :: prime\text{-}card\ mod\text{-}ring\ poly)$
$degree\ f = d$
shows $f\ dvd\ (monom\ 1\ 1)\widehat{\ }(CARD('a)\widehat{\ }d) - monom\ 1\ 1$
and $1 \leq c \Longrightarrow c < d \Longrightarrow \neg\ f\ dvd\ (monom\ 1\ 1)\widehat{\ }(CARD('a)\widehat{\ }c) - monom\ 1\ 1$
⟨*proof*⟩

context
assumes $SORT\text{-}CONSTRAINT('a :: prime\text{-}card)$
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*))
⟨*proof*⟩

termination
⟨*proof*⟩

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)
⟨*proof*⟩

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*⟩
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*⟩

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*⟩
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*⟩

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*) = *Euclidean-Rings.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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *power-poly-f-mod-i*: **assumes** *m*: (*poly-rel* ===> *poly-rel*) *m* (λ *x'*. *x' mod m'*)
shows *poly-rel f f'* ⟹ *poly-rel* (*power-poly-f-mod-i ff-ops m f n*) (*power-poly-f-mod m' f' n*)
⟨*proof*⟩

lemma *berlekamp-mat-i*[*transfer-rule*]: (*poly-rel* ===> *mat-rel R*)

(*berlekamp-mat-i p ff-ops*) *berlekamp-mat*
⟨*proof*⟩

lemma *berlekamp-resulting-mat-i*[*transfer-rule*]: (*poly-rel* ===> *mat-rel R*)
(*berlekamp-resulting-mat-i p ff-ops*) *berlekamp-resulting-mat*
⟨*proof*⟩

lemma *berlekamp-basis-i*[*transfer-rule*]: (*poly-rel* ===> *list-all2 poly-rel*)
(*berlekamp-basis-i p ff-ops*) *berlekamp-basis*
⟨*proof*⟩

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*⟩

lemma *berlekamp-monic-factorization-i*[*transfer-rule*]:
((=) ===> *poly-rel* ===> *list-all2 poly-rel*)
(*berlekamp-monic-factorization-i p ff-ops*) *berlekamp-monic-factorization*
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩
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*⟩

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*)
∧ *c* ∈ {*0* ..< *p*}
∧ (∀ *fi* ∈ *set fs*. *set* (*coeffs fi*) ⊆ {*0* ..< *p*})
⟨*proof*⟩

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* ∨ *degree B* < *degree f B'* = *0* ∨ *degree B'* < *degree f*
and *f*: *f* ≠ *0* **and** *eq*: *A* ∗ *f* + *B* ∗ *g* = *A'* ∗ *f* + *B'* ∗ *g*
shows *A* = *A'* *B* = *B'*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *pdivmod-monic-pseudo-divmod*: **assumes** *g*: *monic g* **shows** *pdivmod-monic f g* = *pseudo-divmod f g*
⟨*proof*⟩

lemma *pdivmod-monic*: **assumes** *g*: *monic g* **and** *res*: *pdivmod-monic f g* = (*q*, *r*)
shows *f* = *g* ∗ *q* + *r r* = *0* ∨ *degree r* < *degree g*
⟨*proof*⟩

definition *dupe-monic* :: *'a* :: *comm-ring-1 poly* ⇒ *'a poly* ⇒ *'a poly* ⇒ *'a poly* ⇒ *'a poly* ⇒
'a poly ∗ *'a poly* **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*: **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* ∨ *degree B* < *degree D*
⟨*proof*⟩

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* ∨ *degree B′* < *degree D*
shows *A′* = *A B′* = *B*
⟨*proof*⟩

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*⟩
end

context *mod-ring-gen*
begin

lemma *monic-of-int-poly*: *monic D* ⟹ *monic* (*of-int-poly* (*Mp D*) :: *'a mod-ring poly*)
⟨*proof*⟩

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*⟩

lemma *Mp-rel-i-of-int-poly-i*: **assumes** *Mp F = F*
shows *Mp-rel-i* (*of-int-poly-i ff-ops F*) *F*
⟨*proof*⟩

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*⟩

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
$A * D + B * H =m\ U$
$B = 0 \vee degree\ B < degree\ D$
$Mp\ A = A$
$Mp\ B = B$
⟨*proof*⟩

lemma *dupe-monic-i-int-finite-field-ops32*: **assumes**
m: $m \leq 65535$
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
$A * D + B * H =m\ U$
$B = 0 \vee degree\ B < degree\ D$
$Mp\ A = A$
$Mp\ B = B$
⟨*proof*⟩

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
$A * D + B * H =m\ U$
$B = 0 \vee degree\ B < degree\ D$
$Mp\ A = A$
$Mp\ B = B$
⟨*proof*⟩

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
$A * D + B * H =m\ U$
$B = 0 \vee degree\ B < degree\ D$
$Mp\ A = A$
$Mp\ B = B$

⟨*proof*⟩
end

context *poly-mod*
begin

definition *dupe-monic-int* :: *int poly* ⇒ *int poly* ⇒ *int poly* ⇒ *int poly* ⇒ *int poly* ⇒
int poly ∗ *int poly* **where**
dupe-monic-int D H S T U = (*case pdivmod-monic* (*Mp* (*T* ∗ *U*)) *D of* (*q*,*r*) ⇒
(*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* ∨ *degree ?B′* < *degree ?D*⟧ ⟹ *?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* ∨ *degree ?B′* < *degree ?D*⟧ ⟹ *?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* ∨ *degree B* < *degree D Mp A* = *A Mp B* = *B*
coprime-m D H ⟹ *A′* ∗ *D* + *B′* ∗ *H* =*m* *U* ⟹ *B′* = *0* ∨ *degree B′* < *degree D* ⟹ *Mp D* = *D*
⟹ *Mp A′* = *A′* ⟹ *Mp B′* = *B′* ⟹ *prime m*
⟹ *A′* = *A* ∧ *B′* = *B*
⟨*proof*⟩

lemma *coprime-bezout-coefficients*:
assumes *cop*: *coprime f g*
and *ext*: *bezout-coefficients f g* = (*a*, *b*)
shows *a* ∗ *f* + *b* ∗ *g* = *1*
⟨*proof*⟩

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*⟩

definition *bezout-coefficients-i* :: *'i arith-ops-record* ⇒ *'i list* ⇒ *'i list* ⇒ *'i list* × *'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* ⇒ *'a arith-ops-record* ⇒ *int poly* ⇒ *int poly* ⇒ *int poly* × *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*) ⇒ (*to-int-poly-i ff-ops a*, *to-int-poly-i ff-ops b*))

definition *euclid-ext-poly-dynamic* :: *int* ⇒ *int poly* ⇒ *int poly* ⇒ *int poly* × *int poly* **where**
euclid-ext-poly-dynamic p = (
if p ≤ *65535*
then euclid-ext-poly-mod-main p (*finite-field-ops32* (*uint32-of-int p*))
else if p ≤ *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*
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

end

lemma *range-sum-prod*: **assumes** *xy*: *x* ∈ {0..<*q*} (*y* :: *int*) ∈ {0..<*p*}
shows *x* + *q* ∗ *y* ∈ {0..<*p* ∗ *q*}
⟨*proof*⟩

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 ≠ 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^n) (D * H) C
    ∧ monic D
    ∧ poly-mod.eq-m p D D1 ∧ poly-mod.eq-m p H H1
    ∧ poly-mod.Mp (p^n) D = D
    ∧ poly-mod.Mp (p^n) H = H ∧
    (poly-mod.eq-m (p^n) (D' * H') C ⟶
     poly-mod.eq-m p D' D1 ⟶
     poly-mod.eq-m p H' H1 ⟶
     poly-mod.Mp (p^n) D' = D' ⟶
     poly-mod.Mp (p^n) H' = H' ⟶ monic D' ⟶ D' = D ∧ H' = H)

  ⟨proof⟩
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 ≠ 0
shows ∃! (D',H'). — D', H' are computed via linear-hensel-binary
      poly-mod.eq-m (p^n) (D' * H') C — the main result: equivalence mod p^n
    ∧ monic D' — monic output
    ∧ eq-m D D' ∧ eq-m H H' — apply ‘mod p‘ on D' and H' yields D and H again
     ∧ poly-mod.Mp (p^n) D' = D' ∧ poly-mod.Mp (p^n) H' = H' — output is
normalized
⟨proof⟩
```

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* $(sdiv\text{-}poly\ (C - D * H)\ q)$
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* $(sdiv\text{-}poly\ (S*D' + T*H' - 1)\ p)$
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': $S' = poly\text{-}mod.Mp\ r\ (S - smult\ p\ A')$
and T': $T' = poly\text{-}mod.Mp\ r\ (T - smult\ p\ B')$
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'
⟨*proof*⟩

definition *hensel-step* **where**
hensel-step p q S1 T1 D1 H1 S T D H $=$ (
let $U =$ *poly-mod.Mp p* $(sdiv\text{-}poly\ (C - D * H)\ q)$; — *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'))
⟨*proof*⟩

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*: $poly\text{-}mod.eq\text{-}m\ p\ (D1 * S1 + H1 * T1)\ 1$
and *mon1*: *monic D1*
and *p*: $p > 1$
and *CDHq*: $poly\text{-}mod.eq\text{-}m\ q\ C\ (D * H)$
and *one-q*: $poly\text{-}mod.eq\text{-}m\ q\ (D * S + H * T)\ 1$
and *D1D*: $poly\text{-}mod.eq\text{-}m\ p\ D1\ D$
and *H1H*: $poly\text{-}mod.eq\text{-}m\ p\ H1\ H$
and *S1S*: $poly\text{-}mod.eq\text{-}m\ p\ S1\ S$
and *T1T*: $poly\text{-}mod.eq\text{-}m\ p\ T1\ T$
and *mon*: *monic D*
and *q*: $q > 1$
and *D1*: $poly\text{-}mod.Mp\ p\ D1 = D1$
and *H1*: $poly\text{-}mod.Mp\ p\ H1 = H1$

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*⟩

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*⟩

context
fixes *p* :: *int* **and** *S1 T1 D1 H1* :: *int poly*

begin
private lemma *decrease*[*termination-simp*]: $\neg\ j \leq 1 \Longrightarrow$ *odd* $j \Longrightarrow$ *Suc* $(j$ *div* $2)$ $< j$ ⟨*proof*⟩

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* $H'))))$
⟨*proof*⟩

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)
  ⟨proof⟩

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⟩
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* ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*⟩

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*
  ⟨*proof*⟩

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
  ⟨proof⟩

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⟩

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⟩

lemma create-factor-tree: assumes xs ≠ []
  shows factors-of-factor-tree (create-factor-tree xs) = mset xs
⟨proof⟩

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 ≠ 0
begin

abbreviation hensel-binary ≡ quadratic-hensel-binary p n

abbreviation hensel-main ≡ 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^n) C (D' * H') — the main result: equivalence mod
p^n
    ∧ monic D' — monic output
    ∧ eq-m D D' ∧ eq-m H H' — apply ‘mod p‘ on D' and H' yields D and H again
     ∧ poly-mod.Mp (p^n) D' = D' ∧ poly-mod.Mp (p^n) H' = H' — output is
normalized
⟨proof⟩

lemma hensel-main:
  assumes eq: eq-m C (prod-mset (factors-of-factor-tree Fs))
  and ⋀ F. F ∈# factors-of-factor-tree Fs ⟹ Mp F = F ∧ monic F
  and hensel-result: hensel-main C Fs = Gs
  and C: monic C poly-mod.Mp (p^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* ($p\hat{}n$) *C* (*prod-list Gs*) — the main result: equivalence mod $p\hat{}n$
$\wedge$ *factors-of-factor-tree Fs* = *mset* (*map Mp Gs*)
$\wedge$ ($\forall$ *G*. *G* $\in$ *set Gs* $\longrightarrow$ *monic G* $\wedge$ *poly-mod.Mp* ($p\hat{}n$) *G* = *G*)
⟨*proof*⟩

lemma *hensel-lifting-monic*:
assumes *eq*: *poly-mod.eq-m p C* (*prod-list Fs*)
and *Fs*: $\bigwedge F.\ F \in$ *set Fs* $\Longrightarrow$ *poly-mod.Mp p F* = *F* $\wedge$ *monic F*
and *res*: *hensel-lifting-monic p n C Fs* = *Gs*
and *mon*: *monic* (*poly-mod.Mp* ($p\hat{}n$) *C*)
and *sf*: *poly-mod.square-free-m p C*
shows *poly-mod.eq-m* ($p\hat{}n$) *C* (*prod-list Gs*)
mset (*map* (*poly-mod.Mp p*) *Gs*) = *mset Fs*
G $\in$ *set Gs* $\Longrightarrow$ *monic G* $\wedge$ *poly-mod.Mp* ($p\hat{}n$) *G* = *G*
⟨*proof*⟩

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* $\in \{0..<p\}$
and *norm*: ($\forall$ *fi*$\in$*set fs*. *set* (*coeffs fi*) $\subseteq \{0..<p\}$)
shows *poly-mod.factorization-m* ($p\hat{}n$) *f* (*lead-coeff f*, *mset gs*) — factorization mod $p\hat{}n$
sort (*map degree fs*) = *sort* (*map degree gs*) — degrees stay the same
$\bigwedge g.\ g \in$ *set gs* $\Longrightarrow$ *monic g* $\wedge$ *poly-mod.Mp* ($p\hat{}n$) *g* = *g* $\wedge$ — monic and normalized
irreducible-m g $\wedge$ — irreducibility even mod *p*
degree-m g = *degree g* — mod *p* does not change degree of *g*
⟨*proof*⟩

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* ⟷ *degree p* = *0* ∨ *a* ∗ *lead-coeff p* ≠ *0*
⟨*proof*⟩

lemma *degree-smult-eqI*[*intro*!]:
assumes *degree p* ≠ *0* ⟹ *a* ∗ *lead-coeff p* ≠ *0*
shows *degree* (*smult a p*) = *degree p*
⟨*proof*⟩

lemma *degree-mult-eq2*:
assumes *lc*: *lead-coeff p* ∗ *lead-coeff q* ≠ *0*
shows *degree* (*p* ∗ *q*) = *degree p* + *degree q* (**is** - = *?r*)
⟨*proof*⟩

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*⟩

context *ring-hom* **begin**
lemma *monic-degree-map-poly-hom*: *monic p* ⟹ *degree* (*map-poly hom p*) = *degree p*
⟨*proof*⟩

lemma *monic-map-poly-hom*: *monic p* ⟹ *monic* (*map-poly hom p*)
⟨*proof*⟩

end

lemma *of-nat-zero*:
assumes *CARD*(*'a*::*nontriv*) *dvd n*
shows (*of-nat n* :: *'a mod-ring*) = *0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *map-poly-rebase* [*simp*]:

map-poly rebase p = #p
⟨*proof*⟩

lemma *rebase-poly-0*: *#0 = 0*
⟨*proof*⟩

lemma *rebase-poly-1*: *#1 = 1*
⟨*proof*⟩

lemma *rebase-poly-pCons*[*simp*]: *#pCons a p = pCons (@a) (#p)*
⟨*proof*⟩

lemma *rebase-poly-self*[*simp*]: *#p = p* ⟨*proof*⟩

lemma *degree-rebase-poly-le*: *degree (#p) ≤ degree p*
⟨*proof*⟩

lemma(**in** *comm-ring-hom*) *degree-map-poly-unit*: **assumes** *lead-coeff p dvd 1*
shows *degree (map-poly hom p) = degree p*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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)* ⟨*proof*⟩

lemma *rebase-eq-0*[*simp*]:
shows *(@(x :: 'a mod-ring) :: 'b mod-ring) = 0 ⟷ x = 0*
⟨*proof*⟩

lemma *degree-rebase-poly-eq*[*simp*]:
shows *degree (#(p :: 'a mod-ring poly) :: 'b mod-ring poly) = degree p*
⟨*proof*⟩

lemma *lead-coeff-rebase-poly*[*simp*]:
lead-coeff (#(p::'a mod-ring poly) :: 'b mod-ring poly) = @lead-coeff p
⟨*proof*⟩

lemma *to-int-mod-ring-rebase*: *to-int-mod-ring*(@(*x* :: *'a mod-ring*)::*'b mod-ring*) = *to-int-mod-ring x*
⟨*proof*⟩

lemma *rebase-id*[*simp*]: @(@(*x*::*'a mod-ring*) :: *'b mod-ring*) = @*x*
⟨*proof*⟩

lemma *rebase-poly-id*[*simp*]: #(#(*p*::*'a mod-ring poly*) :: *'b mod-ring poly*) = #*p*
⟨*proof*⟩

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*) ⟨*proof*⟩

lemma *rebase-id*[*simp*]: @(@(*x*::*'b mod-ring*) :: *'a mod-ring*) = *x* ⟨*proof*⟩

lemma *rebase-poly-id*[*simp*]: #(#(*p*::*'b mod-ring poly*) :: *'a mod-ring poly*) = *p*
⟨*proof*⟩

lemma *rebase-of-nat*[*simp*]: (@(*of-nat n* :: *'a mod-ring*) :: *'b mod-ring*) = *of-nat n*
⟨*proof*⟩

lemma *mod-1-lift-nat*:
assumes (*of-int* (*int x*) :: *'a mod-ring*) = *1*
shows (*of-int* (*int x*) :: *'b mod-ring*) = *1*
⟨*proof*⟩

sublocale *comm-ring-hom rebase* :: *'a mod-ring* ⇒ *'b mod-ring*
⟨*proof*⟩

lemma *of-nat-CARD-eq-0*[*simp*]: (*of-nat CARD*(*'a*) :: *'b mod-ring*) = *0*
⟨*proof*⟩

interpretation *map-poly-hom*: *map-poly-comm-ring-hom rebase* :: *'a mod-ring* ⇒ *'b mod-ring*⟨*proof*⟩

sublocale *poly*: *comm-ring-hom rebase-poly* :: *'a mod-ring poly* ⇒ *'b mod-ring poly*
⟨*proof*⟩

lemma *poly-rebase*[*simp*]: @*poly p x* = *poly* (#(*p* :: *'a mod-ring poly*) :: *'b mod-ring poly*) (@(*x*::*'a mod-ring*) :: *'b mod-ring*)
⟨*proof*⟩

lemma *rebase-poly-smult*[*simp*]: (#(*smult a p* :: *'a mod-ring poly*) :: *'b mod-ring poly*) = *smult* (@*a*) (#*p*)
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *rebase-mult-eq*[*simp*]: (*of-nat CARD*(*'d*) ∗ *a* :: *'a mod-ring*) = *of-nat CARD*(*'d*) ∗ *a'* ⟷ (@*a* :: *'b mod-ring*) = @*a'*
⟨*proof*⟩

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*⟩

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*⟩

lemma *rebase-poly-eq-0-imp-ex-smult*:
(#(*p* :: *'a mod-ring poly*) :: *'b mod-ring poly*) = *0* ⟹
(∃ *p'* :: *'d mod-ring poly*. (*p* = *0* ⟷ *p'* = *0*) ∧ *degree p'* ≤ *degree p* ∧ *p* = *smult* (*of-nat CARD*(*'b*)) (#*p'*))
(**is** *?l* ⟹ *?r*)
⟨*proof*⟩

end

lemma *mod-mod-nat*[*simp*]: *a mod b mod* (*b* ∗ *c* :: *nat*) = *a mod b*
⟨*proof*⟩

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)* ⟨*proof*⟩
sublocale *rebase-pq-to-p*: *rebase-mult TYPE('pq) TYPE('p) TYPE('q)* ⟨*proof*⟩
sublocale *rebase-pq-to-q*: *rebase-mult TYPE('pq) TYPE('q) TYPE('p)* ⟨*proof*⟩

sublocale *rebase-p-to-q*: *rebase-ge TYPE('p) TYPE ('q)* ⟨*proof*⟩
sublocale *rebase-p-to-pq*: *rebase-ge TYPE('p) TYPE ('pq)* ⟨*proof*⟩
sublocale *rebase-q-to-pq*: *rebase-ge TYPE('q) TYPE ('pq)* ⟨*proof*⟩

definition *p* ≡ *if* (*ty-p* :: *'p itself*) = *ty-p then CARD('p) else undefined*
lemma *p*[*simp*]: *p* ≡ *CARD('p)* ⟨*proof*⟩

definition *q* ≡ *if* (*ty-q* :: *'q itself*) = *ty-q then CARD('q) else undefined*
lemma *q*[*simp*]: *q* = *CARD('q)* ⟨*proof*⟩

lemma *p1*: *int p* > *1*
⟨*proof*⟩
lemma *q1*: *int q* > *1*
⟨*proof*⟩
lemma *q0*: *int q* > *0*
⟨*proof*⟩

lemma *pq2*[*simp*]: *CARD('pq)* = *p* * *q* ⟨*proof*⟩

lemma *qq-eq-0*[*simp*]: (*of-nat CARD('q)* * *of-nat CARD('q)* :: *'pq mod-ring*) = *0*
⟨*proof*⟩

lemma *of-nat-q*[*simp*]: *of-nat q* :: *'q mod-ring* ≡ *0* ⟨*proof*⟩

lemma *rebase-rebase*[*simp*]: (@(@(*x*::*'pq mod-ring*) :: *'q mod-ring*) :: *'p mod-ring*)
= @*x*
⟨*proof*⟩

lemma *rebase-rebase-poly*[*simp*]: (#(#(*f*::*'pq mod-ring poly*) :: *'q mod-ring poly*) ::
'p mod-ring poly) = #*f*
⟨*proof*⟩

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 $D :: 'a :: prime\text{-}card\ mod\text{-}ring\ poly$
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*⟩

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*: $(\#u :: 'q\ mod\text{-}ring\ poly) = v * w$
and *degu*: *degree u* = *degree v* + *degree w*
and *avbw*: $(a * \#v + b * \#w :: 'p\ mod\text{-}ring\ poly) = 1$
and *monic-v*: *monic v*

and *bv*: *degree b* < *degree v*
begin

lemma *deg-v*: $degree\ (\#v :: 'p\ mod\text{-}ring\ poly) = degree\ v$
⟨*proof*⟩

lemma *u0*: $u \neq 0$ ⟨*proof*⟩

lemma *ex-f*: $\exists f :: 'p\ mod\text{-}ring\ poly.\ u = \#v * \#w + smult\ (of\text{-}nat\ q)\ (\#f)$
⟨*proof*⟩

definition $f :: 'p\ mod\text{-}ring\ poly \equiv SOME\ f.\ u = \#v * \#w + smult\ (of\text{-}nat\ q)\ (\#f)$

lemma *u*: $u = \#v * \#w + smult\ (of\text{-}nat\ q)\ (\#f)$
⟨*proof*⟩

lemma *t-ex*: $\exists t :: 'p\ mod\text{-}ring\ poly.\ degree\ (b * f - t * \#v) < degree\ v$
⟨*proof*⟩

definition *t* **where** $t \equiv SOME\ t :: 'p\ mod\text{-}ring\ poly.\ degree\ (b * f - t * \#v) <$ *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*⟩

lemma *degv'*: *degree v'* < *degree v* ⟨*proof*⟩

lemma *degqf*[*simp*]: *degree* (*smult* (*of-nat CARD*('*q*)) (#*f* :: '*pq mod-ring poly*))
= *degree* (#*f* :: '*pq mod-ring poly*)
⟨*proof*⟩

lemma *degw'*: *degree w'* ≤ *degree w*
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *wW*: *w* = # *W* ⟨*proof*⟩

lemma *uVW*: *u* = *V* ∗ *W*
⟨*proof*⟩

lemma *degV*: *degree V* = *degree v*
and *lcV*: *lead-coeff V* = @*lead-coeff v*
and *degW*: *degree W* = *degree w*
⟨*proof*⟩

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-ring poly*) (#*w*)

begin

lemma *coprime-preserves*: *coprime* (# *V* :: '*p mod-ring poly*) (# *W*)
⟨*proof*⟩

lemma *pre-unique*:
assumes *f2*: *w''* ∗ #*v* + *v''* ∗ #*w* = *f*
and *degv''*: *degree v''* < *degree v*
shows *v''* = *v'* ∧ *w''* = *w'*
⟨*proof*⟩

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*⟩

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 $u = V' * W' \wedge v = \#V' \wedge w = \#W' \wedge$ *degree V'* = *degree v* ∧ *degree W'* = *degree w* ∧
monic V' ∧ *coprime* (#*V'* :: '*p mod-ring poly*) (#*W'*) (**is** *?main*)
and $(\forall V''\ W''.\ u = V'' * W'' \longrightarrow v = \#V'' \longrightarrow w = \#W'' \longrightarrow$
degree V'' = *degree v* ⟶ *degree W''* = *degree w* ⟶ *lead-coeff V''* = @*lead-coeff v* ⟶
$V'' = V' \wedge W'' = W'$) (**is** *?unique*)
⟨*proof*⟩

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*⟩

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*⟩

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: $n \neq 0$
shows *berlekamp-hensel* p n f $= gs$
and *sort* (*map degree fs*) $=$ *sort* (*map degree gs*)
⟨*proof*⟩

end

lemma *prime-cop-exp-poly-mod*:
assumes *prime*: *prime* p **and** *cop*: *coprime* c p **and** n: $n \neq 0$
shows *poly-mod.M* ($p\hat{}n$) $c \in \{1 ..< p\hat{}n\}$
⟨*proof*⟩

context *poly-mod-2*
begin

context
fixes p :: *int*
assumes *prime*: *prime* p
begin

interpretation p: *poly-mod-prime* p ⟨*proof*⟩

lemma *coprime-lead-coeff-factor*: **assumes** *coprime* (*lead-coeff* ($f * g$)) p
shows *coprime* (*lead-coeff f*) p *coprime* (*lead-coeff g*) p
⟨*proof*⟩

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*⟩

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: $m = p\hat{}n$
shows *factorization-m f F* $\Longrightarrow$ *unique-factorization-m f F*
and *factorization-m g G* $\Longrightarrow$ *unique-factorization-m g G*
⟨*proof*⟩

end

lemma *monic-Mp-prod-mset*: **assumes** *fs*: $\bigwedge f.\ f \in\# fs \Longrightarrow monic\ (Mp\ f)$
shows *monic* (*Mp* (*prod-mset fs*))
⟨*proof*⟩

lemma *degree-Mp-mult-monic*: **assumes** *monic f monic g*
shows *degree* (*Mp* (*f* ∗ *g*)) = *degree f* + *degree g*
⟨*proof*⟩

lemma *factorization-m-degree*: **assumes** *factorization-m f* (*c*,*fs*)
and *0*: *Mp f* ≠ *0*
shows *degree-m f* = *sum-mset* (*image-mset degree-m fs*)
⟨*proof*⟩

lemma *degree-m-mult-le*: *degree-m* (*f* ∗ *g*) ≤ *degree-m f* + *degree-m g*
⟨*proof*⟩

lemma *degree-m-prod-mset-le*: *degree-m* (*prod-mset fs*) ≤ *sum-mset* (*image-mset degree-m fs*)
⟨*proof*⟩

end

context *poly-mod-prime*
begin

lemma *unique-factorization-m-factor-partition*: **assumes** *l0*: *l* ≠ *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*⟩

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* ≠ *0*
shows *factorization-m C* (*c*,*fs*)
⟨*proof*⟩

lemma *unique-monic-hensel-factorization*:
assumes *ufact*: *unique-factorization-m C* (*1*,*Fs*)
and *C*: *monic C square-free-m C*
and *n*: *n* ≠ *0*
shows ∃ *Gs*. *poly-mod.unique-factorization-m* (*p*^*n*) *C* (*1*, *Gs*)

⟨*proof*⟩

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* ≠ *0*
shows *poly-mod.unique-factorization-m* (*p*^*n*) *f* (*lead-coeff f*, *mset gs*) — unique factorization mod *p*^*n*
⋀ *g*. *g* ∈ *set gs* ⟹ *poly-mod.Mp* (*p*^*n*) *g* = *g* — normalized
⟨*proof*⟩

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*⟩

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*⟩

lemma *content-free-unit*:
assumes *content* (*p*::*'a*::{*idom*,*semiring-gcd*} *poly*) = *1*
shows *p dvd 1* ⟷ *degree p* = *0*
⟨*proof*⟩

lemma *square-free-imp-resultant-non-zero*: **assumes** *sf*: *square-free* (*f* :: *int poly*)
shows *resultant f* (*pderiv f*) ≠ *0*
⟨*proof*⟩

lemma *large-mod-0*: **assumes** (*n* :: *int*) > *1* |*k*| < *n k mod n* = *0* **shows** *k* = *0*
⟨*proof*⟩

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*)) ≠ *0*
∧ *map-poly of-int f* ≠ (*0* :: *'a mod-ring poly*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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** $f :: 'a :: prime\text{-}card\ mod\text{-}ring\ poly$
assumes *card*: $CARD('a) > degree\ f$
and *sf*: *square-free f*
shows *separable f*
⟨*proof*⟩

lemma *square-free-iff-separable-GFp*: **assumes** $degree\ f < CARD('a)$
shows $square\text{-}free\ (f :: 'a :: prime\text{-}card\ mod\text{-}ring\ poly) = separable\ f$
⟨*proof*⟩

definition $separable\text{-}impl\text{-}main :: int \Rightarrow 'i\ arith\text{-}ops\text{-}record \Rightarrow int\ poly \Rightarrow bool$
where
$separable\text{-}impl\text{-}main\ p\ ff\text{-}ops\ f = separable\text{-}i\ ff\text{-}ops\ (of\text{-}int\text{-}poly\text{-}i\ ff\text{-}ops\ (poly\text{-}mod.Mp\ p\ f))$

lemma (**in** *prime-field-gen*) *separable-impl*:
shows $separable\text{-}impl\text{-}main\ p\ ff\text{-}ops\ f \Longrightarrow square\text{-}free\text{-}m\ f$
$p > degree\text{-}m\ f \Longrightarrow p > separable\text{-}bound\ f \Longrightarrow square\text{-}free\ f$
$\Longrightarrow separable\text{-}impl\text{-}main\ p\ ff\text{-}ops\ f$ ⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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*) ⇒ *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* ∧ *n* ≤ *i* ∧ *i* < *m*}

⟨*proof*⟩

lemma *find-prime*: **assumes** ∃ *n*. *prime n* ∧ *f n*
shows *prime* (*find-prime f*) ∧ *f* (*find-prime f*)
⟨*proof*⟩

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*⟩

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* ⟨*proof*⟩

lemma *square-free-heuristic*: **assumes** *square-free-heuristic f* = *Some p*
shows *coprime* (*lead-coeff f*) *p* ∧ *separable-impl p f* ∧ *prime p*
⟨*proof*⟩

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))*
⟨*proof*⟩

lemma *sum-list-insort*[*simp*]:
sum-list (insort (d :: 'a :: {comm-monoid-add,linorder}) xs) = d + sum-list xs
⟨*proof*⟩

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*⟩

lemma *max-factor-degree-mono*:
max-factor-degree (map degree (fold remove1 ws vs)) ≤ max-factor-degree (map degree vs)
⟨*proof*⟩

lemma *mset-sub-decompose*: *mset ds ⊆# mset bs + as ⟹ length ds < length bs ⟹ ∃ b1 b b2.*
bs = b1 @ b # b2 ∧ mset ds ⊆# mset (b1 @ b2) + as
⟨*proof*⟩

lemma *max-factor-degree-aux*: **fixes** *es :: nat list*
assumes *sub*: *mset ds ⊆# mset es*
and *len*: *length ds + length ds ≤ length es* **and** *sort*: *sorted es*
shows *sum-list ds ≤ sum-list (drop (length es div 2) es)*
⟨*proof*⟩

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*⟩

lemma *degree-bound*: **assumes** *sub*: *mset ws ⊆# mset vs*
and *len*: *length ws + length ws ≤ length vs*
shows *degree (prod-list ws) ≤ degree-bound vs*
⟨*proof*⟩

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 $P\ (F\ (map\ h\ lst))\ (F\ (map\ g\ lst))$
⟨*proof*⟩
end

lemma *prod-induct-gen*:
assumes $\bigwedge\ a\ r.\ f\ (h\ a * r :: 'a :: \{comm\text{-}monoid\text{-}mult\}) = f\ (g\ a * r)$
shows $f\ (\prod v{\leftarrow}lst.\ h\ v) = f\ (\prod v{\leftarrow}lst.\ g\ v)$
⟨*proof*⟩

abbreviation *complex-of-int*::*int* ⇒ *complex* **where**
complex-of-int ≡ *of-int*

definition *l2norm-list* :: *int list* ⇒ *int* **where**
l2norm-list lst = $\lfloor sqrt\ (sum\text{-}list\ (map\ (\lambda\ a.\ a * a)\ lst))\rfloor$

abbreviation *l2norm* :: *int poly* ⇒ *int* **where**
l2norm p ≡ *l2norm-list* (*coeffs p*)

abbreviation *norm2 p* ≡ $\sum a{\leftarrow}coeffs\ p.\ (cmod\ a)^2$

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*))
$(\prod a{\leftarrow}as.\ [:-\ a,\ 1:]) = p \wedge length\ as = degree\ p)$

lemma *complex-roots*:
smult (*lead-coeff p*) $(\prod a{\leftarrow}complex\text{-}roots\text{-}complex\ p.\ [:-\ a,\ 1:]) = p$
length (*complex-roots-complex p*) = *degree p*
⟨*proof*⟩

lemma *complex-roots-c* [*simp*]:
complex-roots-complex [:*c*:] = []
⟨*proof*⟩

declare *complex-roots*(*2*)[*simp*]

lemma *complex-roots-1* [*simp*]:
complex-roots-complex 1 = []
⟨*proof*⟩

lemma *linear-term-irreducible*$_d$[*simp*]: *irreducible*$_d$ [: *a*, *1*:]
⟨*proof*⟩

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*) (∏ *a*←*complex-roots-int p*. [:− *a*, *1*:]) = *map-poly of-int p*
length (*complex-roots-int p*) = *degree p*
⟨*proof*⟩

The measure for polynomials, after K. Mahler

definition *mahler-measure-poly* **where**
mahler-measure-poly p = *cmod* (*lead-coeff p*) ∗ (∏ *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 = (∏ *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*
⟨*proof*⟩

lemma *smult-inj*[*simp*]: **assumes** (*a*::′*a*::*idom*) ≠ *0* **shows** *inj* (*smult a*)
⟨*proof*⟩

definition *reconstruct-poly*::′*a*::*idom* ⇒ ′*a list* ⇒ ′*a poly* **where**
reconstruct-poly c roots = *smult c* (∏ *a*←*roots*. [:− *a*, *1*:])

lemma *reconstruct-is-original-poly*:
reconstruct-poly (*lead-coeff p*) (*complex-roots-complex p*) = *p*
⟨*proof*⟩

lemma *reconstruct-with-type-conversion*:
smult (*lead-coeff* (*map-poly of-int f*)) (*prod-list* (*map* (λ *a*. [:− *a*, *1*:]) (*complex-roots-int f*)))

$= map\text{-}poly\ of\text{-}int\ f$
⟨*proof*⟩

lemma *reconstruct-prod*:
shows *reconstruct-poly* (*a*::*complex*) *as* ∗ *reconstruct-poly b bs*
$= reconstruct\text{-}poly\ (a * b)\ (as\ @\ bs)$
⟨*proof*⟩

lemma *linear-term-inj*[*simplified*,*simp*]: *inj* (λ *a*. [:− *a*, *1*::′*a*::*idom*:])
⟨*proof*⟩

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*⟩

lemma *reconstruct-poly-defines-mset-of-argument*:
assumes (*a*::′*a*::*field*) ≠ *0*
reconstruct-poly a as = *reconstruct-poly a bs*
shows *mset as* = *mset bs*
⟨*proof*⟩

lemma *complex-roots-complex-prod* [*simp*]:
assumes *f* ≠ *0 g* ≠ *0*
shows *mset* (*complex-roots-complex* (*f* ∗ *g*))
$= mset\ (complex\text{-}roots\text{-}complex\ f) + mset\ (complex\text{-}roots\text{-}complex\ g)$
⟨*proof*⟩

lemma *mset-mult-add*:
assumes *mset* (*a*::′*a*::*field list*) = *mset b* + *mset c*
shows *prod-list a* = *prod-list b* ∗ *prod-list c*
⟨*proof*⟩

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*⟩

lemma *measure-mono-eq-prod*:
assumes *f* ≠ *0 g* ≠ *0*
shows *mahler-measure-monic* (*f* ∗ *g*) = *mahler-measure-monic f* ∗ *mahler-measure-monic g*
⟨*proof*⟩

lemma *mahler-measure-poly-0*[*simp*]: *mahler-measure-poly 0* = *0* ⟨*proof*⟩

lemma *measure-eq-prod*:
mahler-measure-poly (*f* ∗ *g*) = *mahler-measure-poly f* ∗ *mahler-measure-poly g*
⟨*proof*⟩

lemma *prod-cmod*[*simp*]:
  cmod (∏ *a*←*lst*. *f a*) = (∏ *a*←*lst*. *cmod* (*f a*))
  ⟨*proof*⟩

lemma *lead-coeff-of-prod*[*simp*]:
  lead-coeff (∏ *a*←*lst*. *f a*::′*a*::*idom poly*) = (∏ *a*←*lst*. *lead-coeff* (*f a*))
⟨*proof*⟩

lemma *ineq-about-squares*:**assumes** *x* ≤ (*y*::*real*) **shows** *x* ≤ *c*^2 + *y* ⟨*proof*⟩

lemma *first-coeff-le-tail*:(*cmod* (*lead-coeff g*))^2 ≤ (∑ *a*←*coeffs g*. (*cmod a*)^2)
⟨*proof*⟩

lemma *square-prod-cmod*[*simp*]:
  (*cmod* (*a* ∗ *b*))^2 = *cmod a* ^ 2 ∗ *cmod b* ^ 2
⟨*proof*⟩

lemma *sum-coeffs-smult-cmod*:
  (∑ *a*←*coeffs* (*smult v p*). (*cmod a*)^2) = (*cmod v*)^2 ∗ (∑ *a*←*coeffs p*. (*cmod a*)^2)
  (**is** *?l* = *?r*)
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *strip-while-singleton*[*simp*]:
  strip-while ((=) *0*) [*v* ∗ *a*] = *cCons* (*v* ∗ *a*) [] ⟨*proof*⟩

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*⟩

lemma *filter-distr-rev*[*simp*]:
  shows *filter f* (*rev lst*) = *rev* (*filter f lst*)
⟨*proof*⟩

lemma *strip-while-filter*:
  shows *filter* ((≠) *0*) (*strip-while* ((=) *0*) (*lst*::′*a*::*zero list*)) = *filter* ((≠) *0*) *lst*
⟨*proof*⟩

lemma *sum-stripwhile*[*simp*]:
  assumes *f 0* = *0*
  shows (∑ *a*←*strip-while* ((=) *0*) *lst*. *f a*) = (∑ *a*←*lst*. *f a*)
⟨*proof*⟩

lemma *complex-split* : *Complex a b = c* $\longleftrightarrow$ *(a = Re c ∧ b = Im c)*
⟨*proof*⟩

lemma *norm-times-const*:*(*$\sum$ *y←lst. (cmod (a * y))²) = (cmod a)² * (*$\sum$ *y←lst. (cmod y)²)*
⟨*proof*⟩

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←zip (v # l1) (l1 @ [0]). f x) = bisumTail (λx y .f (x,y)) (v#l1)*
⟨*proof*⟩

lemma *bisum-is-map-zip*:
($\sum$ *x←zip (0 # l1) (l1 @ [0]). f x) = bisum (λx y. f (x,y)) l1*
⟨*proof*⟩
lemma *map-zip-is-bisum*:
bisum f l1 = ($\sum$ *(x,y)←zip (0 # l1) (l1 @ [0]). f x y)*
⟨*proof*⟩

lemma *bisum-outside* :
(bisum (λ x y. f1 x − f2 x y + f3 y) lst :: 'a :: field)
= sum-list (map f1 lst) + f1 0 − bisum f2 lst + sum-list (map f3 lst) + f3 0
⟨*proof*⟩

lemma *Landau-lemma*:
($\sum$ *a←coeffs (*$\prod$ *a←lst. [:− a, 1:]). (cmod a)²) = (*$\sum$ *a←coeffs (*$\prod$ *a←lst. linH a). (cmod a)²)*
*(***is** *norm2 ?l = norm2 ?r)*
⟨*proof*⟩

lemma *Landau-inequality*:
mahler-measure-poly f ≤ l2norm-complex f
⟨*proof*⟩

lemma *prod-list-ge1*:
assumes *Ball (set x) (λ (a::real). a ≥ 1)*
shows *prod-list x ≥ 1*
⟨*proof*⟩

lemma *mahler-measure-monic-ge-1*: *mahler-measure-monic p ≥ 1*
⟨*proof*⟩

lemma *mahler-measure-monic-ge-0*: *mahler-measure-monic p* $\geq$ *0*
⟨*proof*⟩

lemma *mahler-measure-ge-0*: *0* $\leq$ *mahler-measure h* ⟨*proof*⟩

lemma *mahler-measure-constant*[*simp*]: *mahler-measure-poly* [:*c*:] = *cmod c*
⟨*proof*⟩

lemma *mahler-measure-factor*[*simplified*,*simp*]: *mahler-measure-poly* [:− *a*, *1*:] = *max 1* (*cmod a*)
⟨*proof*⟩

lemma *mahler-measure-poly-explicit*: *mahler-measure-poly* (*smult c* (∏ *a*←*as*. [:− *a*, *1*:]))
= *cmod c* ∗ (∏ *a*←*as*. (*max 1* (*cmod a*)))
⟨*proof*⟩

lemma *mahler-measure-poly-ge-1*:
assumes *h* ≠ *0*
shows (*1*::*real*) $\leq$ *mahler-measure h*
⟨*proof*⟩

lemma *mahler-measure-dvd*: **assumes** *f* ≠ *0* **and** *h dvd f*
shows *mahler-measure h* $\leq$ *mahler-measure f*
⟨*proof*⟩

definition *graeffe-poly* :: *'a* ⇒ *'a* :: *comm-ring-1 list* ⇒ *nat* ⇒ *'a poly* **where**
graeffe-poly c as m = *smult* (*c* ^ (*2*^*m*)) (∏ *a*←*as*. [:− (*a* ^ (*2*^*m*)), *1*:])

context
fixes *f* :: *complex poly* **and** *c as*
assumes *f*: *f* = *smult c* (∏ *a*←*as*. [:− *a*, *1*:])
begin
lemma *mahler-graeffe*: *mahler-measure-poly* (*graeffe-poly c as m*) = (*mahler-measure-poly f*)^(*2*^*m*)
⟨*proof*⟩
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*⟩

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*⟩

lemma *poly-square-subst: poly-square-subst (f ∘p (monom 1 2)) = f*
⟨*proof*⟩

lemma *poly-even-odd*: **assumes** *poly-even-odd f = (g,h)*
shows *f = g ∘p monom 1 2 + monom 1 1 * (h ∘p monom 1 2)*
⟨*proof*⟩

context
fixes *f :: 'a :: idom poly*
begin

lemma *graeffe-0: f = smult c (∏ a←as. [:− a, 1:]) ⟹ graeffe-poly c as 0 = f*
⟨*proof*⟩

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*⟩
end

definition *graeffe-one-step :: 'a ⇒ 'a :: idom poly ⇒ 'a poly* **where**
*graeffe-one-step c f = smult c (poly-square-subst (f * f ∘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*⟩

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 (∏ a←as. [:− a, 1:])*
shows *graeffe-poly-impl-main ((−1)^degree f) f m = graeffe-poly c as m*

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *drop-half-map*: *drop-half* (*map f xs*) = *map f* (*drop-half xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩
end

lemma *graeffe-poly-impl-mahler*: *mahler-measure* (*graeffe-poly-impl f m*) = *mahler-measure f* ^ *2* ^ *m*
⟨*proof*⟩

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*⟩

lemma *Landau-inequality-mahler-measure*: *mahler-measure f* ≤ *sqrt* (*real-of-int* (∑ *a*←*coeffs f*. *a* ∗ *a*))
⟨*proof*⟩

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*⟩

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*)))
⟨*proof*⟩

termination ⟨*proof*⟩
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*
⟨*proof*⟩

definition *mahler-approximation* :: *nat* ⇒ *int poly* ⇒ *int* **where**
mahler-approximation d f = *mahler-approximation-main* (*d* ∗ *d*) ((−*1*)^(*degree f*)) *f* (−*1*) *0 2*

lemma *mahler-approximation*: ⌊*real d* ∗ *mahler-measure f*⌋ ≤ *mahler-approximation d f*
⟨*proof*⟩
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* ≤ *N* ⟹ *n choose k* ≤ *N choose k*
⟨*proof*⟩

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*⟩

lemma *sum-le-1-prod*: **assumes** *d*: *1* ≤ *d* **and** *c*: *1* ≤ *c*
shows *c* + *d* ≤ *1* + *c* ∗ (*d* :: *real*)
⟨*proof*⟩

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*⟩

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*)))
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

lemma *cmod-through-lead-coeff*[*simp*]:
cmod (*lead-coeff* (*of-int-poly h*)) = *abs* (*lead-coeff h*)
⟨*proof*⟩

lemma *choose-approx*: *n* ≤ *N* ⟹ *n choose k* ≤ *N choose* (*N div 2*)
⟨*proof*⟩

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*⟩

lemma *Mignotte-bound*:
shows *of-int* |*coeff g k*| ≤ (*degree g choose k*) ∗ *mahler-measure g*
⟨*proof*⟩

lemma *mignotte-bound*:
assumes *f* ≠ *0 g dvd f degree g* ≤ *n*
shows |*coeff g k*| ≤ *mignotte-bound f n*
⟨*proof*⟩

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* ⇒ *nat* ⇒ *int* **where**
factor-bound = *mignotte-bound*

lemma *factor-bound*: **assumes** *f* ≠ *0 g dvd f degree g* ≤ *n*
shows |*coeff g k*| ≤ *factor-bound f n*
⟨*proof*⟩

We further prove a result for factor bounds and scalar multiplication.

lemma *factor-bound-ge-0*: *f* ≠ *0* ⟹ *factor-bound f n* ≥ *0*
⟨*proof*⟩

lemma *factor-bound-smult*: **assumes** *f*: *f* ≠ *0* **and** *d*: *d* ≠ *0*
and *dvd*: *g dvd smult d f* **and** *deg*: *degree g* ≤ *n*
shows |*coeff g k*| ≤ |*d*| ∗ *factor-bound f n*
⟨*proof*⟩

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*: (∃ *x*. (*x*, *y*) ∈ *S*) ⟷ *y* ∈ *snd* ‘ *S* ⟨*proof*⟩

lemma *filter-upt*: **assumes** *l* ≤ *m m* < *n* **shows** *filter* ((≤) *m*) [*l*..<*n*] = [*m*..<*n*]
⟨*proof*⟩

lemma *upt-append*: *i* < *j* ⟹ *j* < *k* ⟹ [*i*..<*j*]@[*j*..<*k*] = [*i*..<*k*]
⟨*proof*⟩

lemma *IArray-sub*[*simp*]: (!!) *as* = (!) (*IArray.list-of as*) ⟨*proof*⟩

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*
⟨*proof*⟩

declare *subseqs.simps*(*2*) [*simp del*]

lemma *singleton-mem-set-subseqs* [*simp*]: [*x*] ∈ *set* (*subseqs xs*) ⟷ *x* ∈ *set xs* ⟨*proof*⟩

lemma *Cons-mem-set-subseqsD*: *y*#*ys* ∈ *set* (*subseqs xs*) ⟹ *y* ∈ *set xs* ⟨*proof*⟩

lemma *subseqs-subset*: *ys* ∈ *set* (*subseqs xs*) ⟹ *set ys* ⊆ *set xs*
⟨*proof*⟩

lemma *Cons-mem-set-subseqs-Cons*:
y#*ys* ∈ *set* (*subseqs* (*x*#*xs*)) ⟷ (*y* = *x* ∧ *ys* ∈ *set* (*subseqs xs*)) ∨ *y*#*ys* ∈ *set* (*subseqs xs*)
⟨*proof*⟩

lemma *sorted-subseqs-sorted*:
sorted xs ⟹ *ys* ∈ *set* (*subseqs xs*) ⟹ *sorted ys*
⟨*proof*⟩

lemma *subseqs-of-subseq*: *ys* ∈ *set* (*subseqs xs*) ⟹ *set* (*subseqs ys*) ⊆ *set* (*subseqs xs*)
⟨*proof*⟩

lemma *mem-set-subseqs-append*: *xs* ∈ *set* (*subseqs ys*) ⟹ *xs* ∈ *set* (*subseqs* (*zs* @ *ys*))
⟨*proof*⟩

lemma *Cons-mem-set-subseqs-append*:
x ∈ *set ys* ⟹ *xs* ∈ *set* (*subseqs zs*) ⟹ *x*#*xs* ∈ *set* (*subseqs* (*ys*@*zs*))
⟨*proof*⟩

lemma *Cons-mem-set-subseqs-sorted*:
sorted xs ⟹ *y*#*ys* ∈ *set* (*subseqs xs*) ⟹ *y*#*ys* ∈ *set* (*subseqs* (*filter* (λ*x*. *y* ≤ *x*) *xs*))
⟨*proof*⟩

lemma *subseqs-map*[*simp*]: *subseqs* (*map f xs*) = *map* (*map f*) (*subseqs xs*) ⟨*proof*⟩

lemma *subseqs-of-indices*: *map* (*map* (*nth xs*)) (*subseqs* [*0*..<*length xs*]) = *subseqs xs*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *subseq-of-lengthD*[*dest*]:
assumes *subseq-of-length n xs ys*
shows *ys* ∈ *set* (*subseqs xs*) *length ys* = *n*
⟨*proof*⟩

lemma *subseq-of-length0*[*simp*]: *subseq-of-length 0 xs ys* ⟷ *ys* = [] ⟨*proof*⟩

lemma *subseq-of-length-Nil*[*simp*]: *subseq-of-length n* [] *ys* ⟷ *n* = *0* ∧ *ys* = []
⟨*proof*⟩

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*⟩

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* }
⟨*proof*⟩

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* }
⟨*proof*⟩

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*⟩

definition *basic-subseqs-impl* :: (*'a* ⇒ *'b* ⇒ *'b*) ⇒ (*'a*, *'b*, *'b* × *'a list* × *nat*)*subseqs-impl* **where**
basic-subseqs-impl f = *Sublists-Impl*
(λ *a xs n*. (*subseqs-length f a n xs*, (*a*,*xs*,*n*)))
(λ (*a*,*xs*,*n*). (*subseqs-length f a* (*Suc n*) *xs*, (*a*,*xs*,*Suc n*)))

lemma *basic-subseqs-impl*: **assumes** *f-ac*: ⋀ *x y z*. *f x* (*f y z*) = *f y* (*f x z*)
shows *correct-subseqs-impl f* (*basic-subseqs-impl f*)
(λ *a xs n triple*. (*a*,*xs*,*n*) = *triple*)
⟨*proof*⟩

Improved Implementation **datatype** (*'a*,*'b*,*'state*) *subseqs-foldr-impl* = *Sublists-Foldr-Impl*
(*subseqs-foldr*: *'b* ⇒ *'a list* ⇒ *nat* ⇒ *'b list* × *'state*)
(*next-subseqs-foldr*: *'state* ⇒ *'b list* × *'state*)

locale *subseqs-foldr-impl* =
fixes *f* :: *'a* ⇒ *'b* ⇒ *'b*

and *impl* :: ('a,'b,'state) *subseqs-foldr-impl*
begin
definition *S* **where** *S base elements n* ≡ { *foldr f ys base* | *ys. subseq-of-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 ⇒ 'a *list* ⇒ *nat* ⇒ 'state ⇒ *bool*
assumes *subseqs-foldr*:
subseqs-foldr impl base elements n = (*out*, *state*) ⟹
invariant base elements n state ∧ *set out* = *S base elements n*
and *next-subseqs-foldr*:
next-subseqs-foldr impl state = (*out*, *state'*) ⟹ *invariant base elements n state* ⟹
invariant base elements (*Suc n*) *state'* ∧ *set out* = *S base elements* (*Suc n*)

locale *my-subseqs* =
fixes *f* :: 'a ⇒ 'b ⇒ 'b
begin

context fixes *head* :: 'a **and** *tail* :: 'a *iarray*
begin

fun *next-subseqs1* **and** *next-subseqs2*
where *next-subseqs1 ret0 ret1* [] = (*ret0*, (*head*, *tail*, *ret1*))
| *next-subseqs1 ret0 ret1* ((*i*,*v*)#*prevs*) = *next-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*)
⟨*proof*⟩

lemma *out-subseqs2-Cons*:
out-subseqs2-set v (*j*#*js*) = *insert* (*f* (*tail*!!*j*) *v*) (*out-subseqs2-set v js*)
⟨*proof*⟩

lemma *next-subseqs1-set-as-next-subseqs2-set*:
next-subseqs1-set ((*i*,*v*) # *prevs*) = *next-subseqs1-set prevs* ∪ *next-subseqs2-set v* [*0*..<*i*]
⟨*proof*⟩

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*]
⟨*proof*⟩

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*⟩

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* ⟨*proof*⟩

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) = (λas. foldr f as base) ‘ { as. subseq-of-length n tail as }
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *create-subseqs*:
create-subseqs base elements n = (out, state) ⟹
invariant base elements n state ∧ set out = S base elements n
⟨*proof*⟩

sublocale *correct-subseqs-foldr-impl f impl invariant*

⟨*proof*⟩

lemma *impl-correct*: *correct-subseqs-foldr-impl f impl invariant* ⟨*proof*⟩
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* ⟨*proof*⟩

lemma *foldr-map-prod*[*simp*]:
foldr (λ*x*. *map-prod* (*f x*) (*g x*)) *xs base* = (*foldr f xs* (*fst base*), *foldr g xs* (*snd base*))
⟨*proof*⟩

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-foldr sl-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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *mset-subseqs-size*: *mset* ‘ {*ys*. *ys* ∈ *set* (*subseqs xs*) ∧ *length ys* = *n*} =
{*ws*. *ws* ⊆# *mset xs* ∧ *size ws* = *n*}
⟨*proof*⟩

context *poly-mod-2*
begin
lemma *prod-list-m*[*simp*]: *prod-list-m fs* = *Mp* (*prod-list fs*)
⟨*proof*⟩

lemma *inv-Mp-coeff*: *coeff* (*inv-Mp f*) *n* = *inv-M* (*coeff f n*)
⟨*proof*⟩

lemma *Mp-inv-Mp-id*[*simp*]: *Mp* (*inv-Mp f*) = *Mp f*
⟨*proof*⟩

lemma *inv-Mp-rev*: **assumes** *bnd*: ⋀ *n*. *2* ∗ *abs* (*coeff f n*) < *m*
shows *inv-Mp* (*Mp f*) = *f*
⟨*proof*⟩

lemma *mul-const-commute-below*: *mul-const x* (*mul-const y z*) = *mul-const y* (*mul-const x z*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *coprime-exp-mod*: *coprime lu p* ⟹ *prime p* ⟹ *n* ≠ *0* ⟹ *lu mod p* ^ *n* ≠ *0*
⟨*proof*⟩

interpretation *correct-subseqs-foldr-impl* λ*x*. *map-prod* (*mul-const x*) (*Cons x*) *sl-impl sli* ⟨*proof*⟩

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*: $\bigwedge$ *v*. *v* ∈ *set vs* ⟹ *Mp v* = *v*
and *tests*: $\bigwedge$ *ws*. *ws* ⊆# *mset vs* ⟹ *ws* ≠ {#} ⟹
size ws < *d* ∨ *size ws* = *d* ∧ *ws* ∉ (*mset o snd*) ‘ *set cands*
⟹ *test-dvd u ws*
and *irr*: $\bigwedge$ *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*⟩
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*^*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* ≠ 0
⟨*proof*⟩

interpretation *p*: *poly-mod-prime p* ⟨*proof*⟩

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*⟩

lemma *zassenhaus-reconstruction-irreducible*$_d$:
assumes *res*: *zassenhaus-reconstruction hs p n f* = *fs*
shows *f* = *prod-list fs* ∧ (∀ *fi* ∈ *set fs*. *irreducible*$_d$ *fi*)
⟨*proof*⟩

corollary *zassenhaus-reconstruction*:
assumes *pr*: *primitive f*
assumes *res*: *zassenhaus-reconstruction hs p n f* = *fs*
shows *f* = *prod-list fs* ∧ (∀ *fi* ∈ *set fs*. *irreducible fi*)
⟨*proof*⟩
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*
⟨*proof*⟩

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*⟩

end

definition *berlekamp-zassenhaus-factorization* :: *int poly* ⇒ *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*^*k* > *bnd*
k = *find-exponent p bnd*;
— perform hensel lifting to lift factorization to mod *p*^*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*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

lemma *yun-erel-1-eq*: **assumes** *yun-erel F f*
shows (*F* = *1*) ⟷ (*f* = *1*)
⟨*proof*⟩

lemma *yun-rel-1*[*simp*]: *yun-rel 1 1 1*
⟨*proof*⟩

lemma *yun-erel-1*[*simp*]: *yun-erel 1 1* ⟨*proof*⟩

lemma *yun-rel-mult*: *yun-rel F c f* ⟹ *yun-rel G d g* ⟹ *yun-rel* (*F* ∗ *G*) (*c* ∗ *d*)
(*f* ∗ *g*)
⟨*proof*⟩

lemma *yun-erel-mult*: *yun-erel F f* ⟹ *yun-erel G g* ⟹ *yun-erel* (*F* ∗ *G*) (*f* ∗ *g*)

⟨*proof*⟩

lemma *yun-rel-pow*: **assumes** *yun-rel F c f*
shows *yun-rel* (*F*^*n*) (*c*^*n*) (*f*^*n*)
⟨*proof*⟩

lemma *yun-erel-pow*: *yun-erel F f* $\Longrightarrow$ *yun-erel* (*F*^*n*) (*f*^*n*)
⟨*proof*⟩

lemma *yun-wrel-pderiv*: **assumes** *yun-wrel F c f*
shows *yun-wrel* (*pderiv F*) *c* (*pderiv f*)
⟨*proof*⟩

lemma *yun-wrel-minus*: **assumes** *yun-wrel F c f yun-wrel G c g*
shows *yun-wrel* (*F* − *G*) *c* (*f* − *g*)
⟨*proof*⟩

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* ≠ *0*
shows *yun-wrel* (*F div G*) (*c* / *d*) (*f div g*)
⟨*proof*⟩

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*⟩

lemma *yun-wrel-gcd*: **assumes** *yun-wrel F c′ f yun-wrel G c g* **and** *c*: *c′* ≠ *0 c* ≠ *0*
and *d*: *d* = *rat-of-int* (*lead-coeff* (*gcd F G*)) *d* ≠ *0*
shows *yun-wrel* (*gcd F G*) *d* (*gcd f g*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *yun-rel-same-right*: **assumes** *yun-rel f c G yun-rel g d G*
shows *f = g*
⟨*proof*⟩

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*,*1*)])

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*⟩

definition *square-free-factorization-int′* :: *int poly* ⇒ *int* × (*int poly* × *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*⟩

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 $f = monom\ 1\ n * g\ n \neq 0 \vee f \neq 0 \Longrightarrow \neg\ monom\ 1\ 1\ dvd\ g$
⟨*proof*⟩

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 $x\hat{}n$
⇒ *case square-free-factorization-int′ g of* (*d*,*fs*)
⇒ *if n* = *0 then* (*d*,*fs*) *else* (*d*, (*monom 1 1*, *n*) # *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*⟩

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*))}
⟨*proof*⟩

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* ⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *irreducible-imp-square-free*:
assumes *irr*: *irreducible* (*p*::*'a*::*idom poly*) **shows** *square-free p*
⟨*proof*⟩

lemma *not-mem-set-dropWhileD*: *x ∉ set* (*dropWhile P xs*) ⟹ *x ∈ set xs* ⟹ *P x*
⟨*proof*⟩

lemma *primitive-reflect-poly*:
fixes *f* :: *'a* :: *comm-semiring-1 poly*
shows *primitive* (*reflect-poly f*) = *primitive f*
⟨*proof*⟩

lemma *gcd-list-sub*:
assumes *set xs* ⊆ *set ys* **shows** *gcd-list ys dvd gcd-list xs*
⟨*proof*⟩

lemma *content-reflect-poly*:
content (*reflect-poly f*) = *content f* (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *coeff-primitive-part*: *content f* ∗ *coeff* (*primitive-part f*) *i* = *coeff f i*
⟨*proof*⟩

lemma *smult-cancel*[*simp*]:
fixes *c* :: *'a* :: *idom*
shows *smult c f = smult c g* ⟷ *c = 0 ∨ f = g*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *reflect-poly-eq-zero*[*simp*]:

reflect-poly f = 0 ⟷ f = 0
⟨*proof*⟩

lemma *irreducible$_d$-reflect-poly-main*:
fixes *f* :: *'a* :: {*idom*, *semiring-gcd*} *poly*
assumes *nz*: *coeff f 0 ≠ 0*
and *irr*: *irreducible$_d$* (*reflect-poly f*)
shows *irreducible$_d$ f*
⟨*proof*⟩

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*⟩

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*⟩

lemma *reflect-poly-dvd*: (*f* :: *'a* :: *idom poly*) *dvd g* ⟹ *reflect-poly f dvd reflect-poly g*
⟨*proof*⟩

lemma *square-free-reflect-poly*: **fixes** *f* :: *'a* :: *idom poly*
assumes *sf*: *square-free f*
and *nz*: *coeff f 0 ≠ 0*
shows *square-free* (*reflect-poly f*) ⟨*proof*⟩

lemma *gcd-reflect-poly*: **fixes** *f* :: *'a* :: {*factorial-ring-gcd*, *semiring-gcd-mult-normalize*} *poly*
assumes *nz*: *coeff f 0 ≠ 0 coeff g 0 ≠ 0*
shows *gcd* (*reflect-poly f*) (*reflect-poly g*) = *normalize* (*reflect-poly* (*gcd f g*))
⟨*proof*⟩

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*⟩

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*⟩

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*)))
⟨*proof*⟩

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*,*1*)]) *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 $x \hat{} n$
⇒ *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*) # *fs*))

lemma *factorize-int-poly-0*[*simp*]: *factorize-int-poly-generic 0* = (*0*,[])
⟨*proof*⟩

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*⟩

lemma *internal-int-poly-factorization-mem*:
assumes *f*: *coeff f 0* ≠ *0*
and *res*: *internal-int-poly-factorization f* = (*c*,*fs*)
and *mem*: (*fi*,*i*) ∈ *set fs*
shows *irreducible fi irreducible*$_d$ *fi* **and** *primitive fi* **and** *degree fi* ≠ *0 i* ≠ *0*
⟨*proof*⟩

lemma *internal-int-poly-factorization*:
assumes *f*: *coeff f 0* ≠ *0*
and *res*: *internal-int-poly-factorization f* = (*c*,*fs*)
shows *square-free-factorization f* (*c*,*fs*)
⟨*proof*⟩

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*⟩

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*⟩
end

lift-definition *berlekamp-zassenhaus-factorization-algorithm* :: *int-poly-factorization-algorithm*
is *berlekamp-zassenhaus-factorization*
⟨*proof*⟩

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 ⇒ -
⟨*proof*⟩

lemma *prod-dvd-1-imp-all-dvd-1*:
assumes *finite X* **and** *prod f X dvd 1* **and** *x* ∈ *X* **shows** *f x dvd 1*
⟨*proof*⟩

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*,[])
⟨*proof*⟩

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*⟩

end

abbreviation *factorize-rat-poly* **where**
factorize-rat-poly ≡ *factorize-rat-poly-generic berlekamp-zassenhaus-factorization-algorithm*

end

12 External Interface

We provide two functions for external usage that work on lists and integers only, so that they can easily be accessed via these primitive datatypes.

theory *Factorization-External-Interface*
imports
Factorize-Rat-Poly
Factorize-Int-Poly
begin

declare *Lcm-fin.set-eq-fold*[*code-unfold*]

definition *factor-int-poly* :: *integer list* ⇒ *integer* × (*integer list* × *integer*) *list*
where
factor-int-poly p = *map-prod integer-of-int* (*map* (*map-prod* (*map integer-of-int*
o coeffs) *integer-of-nat*))
(*factorize-int-poly* (*poly-of-list* (*map int-of-integer p*)))

Just for clarifying the representation, we present a part of the soundness statement of the factorization algorithm with conversions included

lemma *factor-int-poly*: **assumes** *factor-int-poly p* = (*c*, *qes*)
shows *poly-of-list* (*map int-of-integer p*) = *smult* (*int-of-integer c*)
(∏ (*q*, *e*)←*qes*. *poly-of-list* (*map int-of-integer q*) ^ *nat-of-integer e*)
(**is** *?p* = *?prod*)
⟨*proof*⟩

Note that coefficients are listed with lowest coefficient as head of list

value *coeffs* (*monom 1 3*) :: *int list*
value *factor-int-poly* [*0*,*0*,*0*,*5*]
value *factor-int-poly* [*0*,*1*,−*2*,*1*]

definition *integers-of-rat* **where** *integers-of-rat x* = *map-prod integer-of-int integer-of-int* (*quotient-of x*)
fun *rat-of-integers* **where** *rat-of-integers* (*n*,*d*) = (*rat-of-int* (*int-of-integer n*) / *rat-of-int* (*int-of-integer d*))

definition *integer-of-rat* **where** *integer-of-rat x* = *integer-of-int* (*fst* (*quotient-of x*))
definition *rat-of-integer* **where** *rat-of-integer x* = *rat-of-int* (*int-of-integer x*)

lemma *integers-of-rat*[*simp*]: *rat-of-integers* (*integers-of-rat x*) = *x*
⟨*proof*⟩

lemma *integer-of-rat*[*simp*]: **assumes** *x* ∈ ℤ
shows *rat-of-integer* (*integer-of-rat x*) = *x*
⟨*proof*⟩

definition *factor-rat-poly* :: (*integer* × *integer*) *list* ⇒ (*integer* × *integer*) × (*integer list* × *integer*) *list* **where**
factor-rat-poly p = *map-prod integers-of-rat* (*map* (*map-prod* (*map integer-of-rat*
o coeffs) *integer-of-nat*))
(*factorize-rat-poly* (*poly-of-list* (*map rat-of-integers p*)))

lemma *factor-rat-poly*: **assumes** *factor-rat-poly p* = (*c*, *qes*)
shows *poly-of-list* (*map rat-of-integers p*) = *smult* (*rat-of-integers c*)
(∏ (*q*, *e*)←*qes*. *poly-of-list* (*map rat-of-integer q*) ^ *nat-of-integer e*)
(**is** *?p* = *?prod*)
⟨*proof*⟩

Note that rational numbers in the input are encoded as pairs, whereas the polynomials in the output are just integer polynomials, i.e., only the

constant factor is a rational number

value *factor-rat-poly* [(*1*,*6*),(*−1*,*3*),(*1*,*6*)]

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.