

The Factorization Algorithm of Berlekamp and Zassenhaus *

Jose Divasón Sebastiaan Joosten René Thiemann
Akihisa Yamada

December 14, 2021

Abstract

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

The algorithm first performs a factorization in the prime field $\text{GF}(p)$ 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

1	Introduction	3
2	Finite Rings and Fields	5
2.1	Finite Rings	5
2.2	Nontrivial Finite Rings	7
2.3	Finite Fields	8
3	Arithmetics via Records	10
3.1	Finite Fields	13
3.1.1	Transfer Relation	17
3.1.2	Transfer Rules	17
3.2	Matrix Operations in Fields	33
3.3	Interfacing UFD properties	40

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

3.3.1	Original part	40
3.3.2	Connecting to HOL/Divisibility	42
3.4	Preservation of Irreducibility	44
3.4.1	Back to divisibility	45
3.5	Results for GCDs etc.	45
4	Unique Factorization Domain for Polynomials	49
5	Polynomials in Rings and Fields	55
5.1	Polynomials in Rings	55
5.2	Polynomials in a Finite Field	66
5.3	Transferring to class-based mod-ring	67
5.4	Karatsuba's Multiplication Algorithm for Polynomials	73
5.5	Record Based Version	75
5.5.1	Definitions	75
5.5.2	Properties	80
5.5.3	Over a Finite Field	84
5.6	Chinese Remainder Theorem for Polynomials	87
6	The Berlekamp Algorithm	90
6.1	Auxiliary lemmas	90
6.2	Previous Results	94
6.3	Definitions	97
6.4	Properties	98
7	Distinct Degree Factorization	109
8	A Combined Factorization Algorithm for Polynomials over $\mathbf{GF}(p)$	116
8.1	Type Based Version	116
8.2	Record Based Version	117
9	Hensel Lifting	122
9.1	Properties about Factors	122
9.2	Hensel Lifting in a Type-Based Setting	139
9.3	Result is Unique	147
10	Reconstructing Factors of Integer Polynomials	151
10.1	Square-Free Polynomials over Finite Fields and Integers	151
10.2	Finding a Suitable Prime	152
10.3	Maximal Degree during Reconstruction	154
10.4	Mahler Measure	156
10.5	The Mignotte Bound	165
10.6	Iteration of Subsets of Factors	166
10.7	Reconstruction of Integer Factorization	173

11 The Polynomial Factorization Algorithm	179
11.1 Factoring Square-Free Integer Polynomials	179
11.2 A fast coprimality approximation	180
11.3 Factoring Arbitrary Integer Polynomials	185
11.4 Factoring Rational Polynomials	190

1 Introduction

Modern algorithms to factor integer polynomials – following Berlekamp and Zassenhaus – work via polynomial factorization over prime fields $\text{GF}(p)$ 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, \dots, f_n such that $f = f_1 \cdot \dots \cdot f_n$.

- 4 Choose a suitable prime p depending on f .
 - 5 Factor f in $\text{GF}(p)$: $f \equiv g_1 \cdot \dots \cdot g_m \pmod{p}$.
 - 6 Determine a suitable bound d on the degree, depending on g_1, \dots, 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 $f \equiv h_1 \cdot \dots \cdot h_m \pmod{p^k}$ via the Hensel lifting.
 - 8 Construct a factorization $f = f_1 \cdot \dots \cdot f_n$ 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 $\text{int poly} \Rightarrow \text{int poly list}$, where we chose Berlekamp’s algorithm in step 5. However, the algorithm was available only as an oracle, and thus a validity check on the result factorization had to be performed.

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

Theorem 1 (Berlekamp-Zassenhaus’ Algorithm)

```

assumes square_free (f :: int poly)
and degree f ≠ 0
and berlekamp_zassenhaus_factorization f = fs
shows f = prod_list fs
and ∀fi ∈ set fs. irreducible fi

```

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

To obtain Theorem 1 we perform the following tasks.

- We introduce two formulations of $\text{GF}(p)$ 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 development, 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 $\text{GF}(p)$. 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 $\text{GF}(p)$.
- 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 $\text{GF}(p)$ 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  $\Rightarrow$  'a mod-ring  $\Rightarrow$  bool is (=) <proof>
instance <proof>
end

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

lift-definition plus-mod-ring :: 'a mod-ring  $\Rightarrow$  'a mod-ring  $\Rightarrow$  'a mod-ring is
   $\lambda x y. (x + y) \text{ mod int (CARD('a))}$  <proof>

lift-definition uminus-mod-ring :: 'a mod-ring  $\Rightarrow$  'a mod-ring is
   $\lambda x. \text{if } x = 0 \text{ then } 0 \text{ else int (CARD('a)) - } x$  <proof>

lift-definition minus-mod-ring :: 'a mod-ring  $\Rightarrow$  'a mod-ring  $\Rightarrow$  'a mod-ring is
   $\lambda x y. (x - y) \text{ mod int (CARD('a))}$  <proof>

lift-definition times-mod-ring :: 'a mod-ring  $\Rightarrow$  'a mod-ring  $\Rightarrow$  'a mod-ring is
   $\lambda x y. (x * y) \text{ mod int (CARD('a))}$  <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  $\Rightarrow$  int is  $\lambda x. x$  <proof>

lift-definition of-int-mod-ring :: int  $\Rightarrow$  'a::finite mod-ring is
   $\lambda x. x \text{ mod int (CARD('a))}$  <proof>

interpretation to-int-mod-ring-hom: inj-zero-hom to-int-mod-ring

```

<proof>

lemma *int-nat-card[simp]*: $\text{int } (\text{nat } \text{CARD}('a::\text{finite})) = \text{CARD}('a)$ *<proof>*

interpretation *of-int-mod-ring-hom*: *zero-hom of-int-mod-ring*
<proof>

lemma *of-int-mod-ring-to-int-mod-ring[simp]*:
 $\text{of-int-mod-ring } (\text{to-int-mod-ring } x) = x$ *<proof>*

lemma *to-int-mod-ring-of-int-mod-ring[simp]*: $0 \leq x \implies x < \text{int } \text{CARD}('a :: \text{finite}) \implies$
 $\text{to-int-mod-ring } (\text{of-int-mod-ring } x :: 'a \text{ mod-ring}) = x$
<proof>

lemma *range-to-int-mod-ring*:
 $\text{range } (\text{to-int-mod-ring } :: ('a :: \text{finite mod-ring} \Rightarrow \text{int})) = \{0 ..< \text{CARD}('a)\}$
<proof>

2.2 Nontrivial Finite Rings

class *nontriv* = **assumes** *nontriv*: $\text{CARD}('a) > 1$

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]*:
 $\text{of-nat} = \text{of-int-mod-ring } \circ \text{int}$
<proof>

lemma *of-nat-card-eq-0[simp]*: $(\text{of-nat } (\text{CARD}('a::\text{nontriv})) :: 'a \text{ mod-ring}) = 0$
<proof>

lemma *of-int-of-int-mod-ring[code-unfold]*: $\text{of-int} = \text{of-int-mod-ring}$
<proof>

unbundle *lifting-syntax*

lemma *pcr-mod-ring-to-int-mod-ring*: $pcr\text{-mod-ring} = (\lambda x y. x = to\text{-int-mod-ring } y)$
 ⟨proof⟩

lemma [*transfer-rule*]:
 $((=) ==> pcr\text{-mod-ring}) (\lambda x. int\ x\ mod\ int\ (CARD('a :: nontriv))) (of\text{-nat} :: nat \Rightarrow 'a\ mod\text{-ring})$
 ⟨proof⟩

lemma [*transfer-rule*]:
 $((=) ==> pcr\text{-mod-ring}) (\lambda x. x\ mod\ int\ (CARD('a :: nontriv))) (of\text{-int} :: int \Rightarrow 'a\ mod\text{-ring})$
 ⟨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*]:
 $(pcr\text{-mod-ring} ==> (=) ==> pcr\text{-mod-ring})$
 $(\lambda a :: int. \lambda n. a^n\ mod\ CARD('a :: nontriv)) ((\wedge) :: 'a\ mod\text{-ring} \Rightarrow nat \Rightarrow 'a\ mod\text{-ring})$
 ⟨proof⟩

lemma *dvd-mod-ring-transfer* [*transfer-rule*]:
 $((pcr\text{-mod-ring} :: int \Rightarrow 'a :: nontriv\ mod\text{-ring} \Rightarrow bool) ==>$
 $(pcr\text{-mod-ring} :: int \Rightarrow 'a\ mod\text{-ring} \Rightarrow bool) ==> (=))$
 $(\lambda i\ j. \exists k \in \{0..<int\ CARD('a)\}. j = i * k\ mod\ int\ CARD('a)) (dvd)$
 ⟨proof⟩

lemma *Rep-mod-ring-mod* [*simp*]: $Rep\text{-mod-ring}\ (a :: 'a :: nontriv\ mod\text{-ring})\ mod\ CARD('a) = Rep\text{-mod-ring}\ a$
 ⟨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

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

begin

definition *inverse-mod-ring* :: 'a mod-ring \Rightarrow 'a mod-ring **where**
inverse-mod-ring $x = (\text{if } x = 0 \text{ then } 0 \text{ else } x \wedge (\text{nat } (\text{CARD}('a) - 2)))$

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

instance

<proof>

end

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

definition *modulo-mod-ring* :: 'a mod-ring \Rightarrow 'a mod-ring \Rightarrow 'a mod-ring **where**
modulo-mod-ring $x y = (\text{if } y = 0 \text{ then } x \text{ else } 0)$

definition *normalize-mod-ring* :: 'a mod-ring \Rightarrow 'a mod-ring **where** *normalize-mod-ring*
 $x = (\text{if } x = 0 \text{ then } 0 \text{ else } 1)$

definition *unit-factor-mod-ring* :: 'a mod-ring \Rightarrow 'a mod-ring **where** *unit-factor-mod-ring*
 $x = x$

definition *euclidean-size-mod-ring* :: 'a mod-ring \Rightarrow nat **where** *euclidean-size-mod-ring*
 $x = (\text{if } x = 0 \text{ then } 0 \text{ else } 1)$

instance

<proof>

end

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

definition *gcd-mod-ring* :: 'a mod-ring \Rightarrow 'a mod-ring \Rightarrow 'a mod-ring **where**
gcd-mod-ring = *Euclidean-Algorithm.gcd*

definition *lcm-mod-ring* :: 'a mod-ring \Rightarrow 'a mod-ring \Rightarrow 'a mod-ring **where**
lcm-mod-ring = *Euclidean-Algorithm.lcm*

definition *Gcd-mod-ring* :: 'a mod-ring set \Rightarrow 'a mod-ring **where** *Gcd-mod-ring*
= *Euclidean-Algorithm.Gcd*

definition *Lcm-mod-ring* :: 'a mod-ring set \Rightarrow '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* ($x :: 'a \text{ mod-ring}$) = ($1 :: 'a \text{ mod-ring}$)

instance $\langle proof \rangle$

end

instance *mod-ring* :: (*prime-card*) *field-gcd*
 $\langle proof \rangle$

lemma *surj-of-nat-mod-ring*: $\exists i. i < CARD('a :: prime-card) \wedge (x :: 'a mod-ring)$
 $= of-nat\ i$
 $\langle proof \rangle$

lemma *of-nat-0-mod-ring-dvd*: **assumes** $x: of-nat\ x = (0 :: 'a :: prime-card\ mod-ring)$
shows $CARD('a)\ dvd\ x$
 $\langle proof \rangle$

end

3 Arithmetics via Records

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

theory *Arithmetic-Record-Based*

imports

HOL-Library.More-List

HOL-Computational-Algebra.Euclidean-Algorithm

begin

datatype *'a arith-ops-record* = *Arith-Ops-Record*

(*zero* : *'a*)

(*one* : *'a*)

(*plus* : *'a* \Rightarrow *'a* \Rightarrow *'a*)

(*times* : *'a* \Rightarrow *'a* \Rightarrow *'a*)

(*minus* : *'a* \Rightarrow *'a* \Rightarrow *'a*)

(*uminus* : *'a* \Rightarrow *'a*)

(*divide* : *'a* \Rightarrow *'a* \Rightarrow *'a*)

(*inverse* : *'a* \Rightarrow *'a*)

(*modulo* : *'a* \Rightarrow *'a* \Rightarrow *'a*)

(*normalize* : *'a* \Rightarrow *'a*)

(*unit-factor* : *'a* \Rightarrow *'a*)

(*of-int* : *int* \Rightarrow *'a*)

(*to-int* : *'a* \Rightarrow *int*)

(*DP* : *'a* \Rightarrow *bool*)

hide-const (**open**)

zero

one

plus

times

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

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

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

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

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

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

abbreviation (*input*) *euclid-ext-i* :: 'i ⇒ '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
  Native-Word.Code-Target-Bits-Int
  HOL-Library.Code-Target-Numeral
begin

```

definition *mod-nonneg-pos* :: integer ⇒ integer ⇒ integer **where**
 $x \geq 0 \implies y > 0 \implies \text{mod-nonneg-pos } x \ y = (x \bmod y)$

code-printing — FIXME illusion of partiality

```
constant mod-nonneg-pos  $\leftarrow$ 
  (SML) IntInf.mod / ( -, / - )
  and (Eval) IntInf.mod / ( -, / - )
  and (OCaml) Z.rem
  and (Haskell) Prelude.mod / ( - ) / ( - )
  and (Scala) !((k: BigInt) => (l: BigInt) => / (k %0 l))
```

definition *mod-nonneg-pos-int* :: int ⇒ int ⇒ int **where**
 $\text{mod-nonneg-pos-int } x \ y = \text{int-of-integer } (\text{mod-nonneg-pos } (\text{integer-of-int } x) (\text{integer-of-int } y))$

lemma *mod-nonneg-pos-int[simp]*: $x \geq 0 \implies y > 0 \implies \text{mod-nonneg-pos-int } x \ y = (x \bmod y)$
 ⟨*proof*⟩

context

fixes $p :: \text{int}$

begin

definition *plus-p* :: int ⇒ int ⇒ int **where**
 $\text{plus-p } x \ y \equiv \text{let } z = x + y \text{ in if } z \geq p \text{ then } z - p \text{ else } z$

definition *minus-p* :: int ⇒ int ⇒ int **where**
 $\text{minus-p } x \ y \equiv \text{if } y \leq x \text{ then } x - y \text{ else } x + p - y$

definition *uminus-p* :: int ⇒ int **where**
 $\text{uminus-p } x = (\text{if } x = 0 \text{ then } 0 \text{ else } p - x)$

definition *mult-p* :: int ⇒ int ⇒ int **where**
 $\text{mult-p } x \ y = (\text{mod-nonneg-pos-int } (x * y) \ p)$

fun *power-p* :: int ⇒ nat ⇒ int **where**

```
power-p  $x \ n = (\text{if } n = 0 \text{ then } 1 \text{ else}$ 
   $\text{let } (d,r) = \text{Divides.divmod-nat } n \ 2;$ 
   $\text{rec} = \text{power-p } (\text{mult-p } x \ x) \ d \text{ in}$ 
   $\text{if } r = 0 \text{ then } \text{rec} \text{ else } \text{mult-p } \text{rec } x)$ 
```

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**
 $\text{inverse-p } x = (\text{if } x = 0 \text{ then } 0 \text{ else } \text{power-p } x \ (\text{nat } (p - 2)))$

definition *divide-p* :: *int* \Rightarrow *int* \Rightarrow *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 \equiv *Arith-Ops-Record*

0
1
plus-p
mult-p
minus-p
uminus-p
divide-p
inverse-p
 $(\lambda x y . \text{if } y = 0 \text{ then } x \text{ else } 0)$
 $(\lambda x . \text{if } x = 0 \text{ then } 0 \text{ else } 1)$
 $(\lambda x . x)$
 $(\lambda x . x)$
 $(\lambda x . x)$
 $(\lambda x . 0 \leq x \wedge x < p)$

end

context

fixes *p* :: *uint32*

begin

definition *plus-p32* :: *uint32* \Rightarrow *uint32* \Rightarrow *uint32* **where**
plus-p32 *x y* \equiv *let* *z* = *x* + *y* *in* *if* *z* \geq *p* *then* *z* - *p* *else* *z*

definition *minus-p32* :: *uint32* \Rightarrow *uint32* \Rightarrow *uint32* **where**
minus-p32 *x y* \equiv *if* *y* \leq *x* *then* *x* - *y* *else* (*x* + *p*) - *y*

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

definition *mult-p32* :: *uint32* \Rightarrow *uint32* \Rightarrow *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)
 \langle *proof* \rangle

lemma *int-of-uint32-0-iff*: *int-of-uint32* *n* = 0 \longleftrightarrow *n* = 0
 \langle *proof* \rangle

lemma *int-of-uint32-0*: *int-of-uint32* 0 = 0 \langle *proof* \rangle

lemma *int-of-uint32-ge-0*: *int-of-uint32* *n* \geq 0
 \langle *proof* \rangle

lemma *two-32*: $2 \wedge \text{LENGTH}(32) = (4294967296 :: \text{int})$ *<proof>*

lemma *int-of-uint32-plus*: $\text{int-of-uint32 } (x + y) = (\text{int-of-uint32 } x + \text{int-of-uint32 } y) \text{ mod } 4294967296$
<proof>

lemma *int-of-uint32-minus*: $\text{int-of-uint32 } (x - y) = (\text{int-of-uint32 } x - \text{int-of-uint32 } y) \text{ mod } 4294967296$
<proof>

lemma *int-of-uint32-mult*: $\text{int-of-uint32 } (x * y) = (\text{int-of-uint32 } x * \text{int-of-uint32 } y) \text{ mod } 4294967296$
<proof>

lemma *int-of-uint32-mod*: $\text{int-of-uint32 } (x \text{ mod } y) = (\text{int-of-uint32 } x \text{ mod } \text{int-of-uint32 } y)$
<proof>

lemma *int-of-uint32-inv*: $0 \leq x \implies x < 4294967296 \implies \text{int-of-uint32 } (\text{uint32-of-int } x) = x$
<proof>

context

includes *bit-operations-syntax*

begin

function *power-p32* :: $\text{uint32} \Rightarrow \text{uint32} \Rightarrow \text{uint32}$ **where**
 power-p32 x $n = (\text{if } n = 0 \text{ then } 1 \text{ else}$
 let $\text{rec} = \text{power-p32 } (\text{mult-p32 } x x) (\text{drop-bit } 1 n)$ *in*
 if $n \text{ AND } 1 = 0$ *then* rec *else* $\text{mult-p32 } \text{rec } x$
<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* :: $\text{uint32} \Rightarrow \text{uint32}$ **where**
 inverse-p32 $x = (\text{if } x = 0 \text{ then } 0 \text{ else } \text{power-p32 } x (p - 2))$

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

definition *finite-field-ops32* :: uint32 *arith-ops-record* **where**
 finite-field-ops32 $\equiv \text{Arith-Ops-Record}$


```

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

```

lemma *shiftr-uint32-code* [*code-unfold*]: $\text{drop-bit } 1 \ x = (\text{uint32-shiftr } x \ 1)$
<proof>

3.1.1 Transfer Relation

```

locale mod-ring-locale =
  fixes  $p :: \text{int}$  and  $ty :: 'a$  nontriv itself
  assumes  $p = \text{int } \text{CARD}('a)$ 
begin
lemma nat-p:  $\text{nat } p = \text{CARD}('a)$  <proof>
lemma p2:  $p \geq 2$  <proof>
lemma p2-ident:  $\text{int } (\text{CARD}('a) - 2) = p - 2$  <proof>

```

definition *mod-ring-rel* :: $\text{int} \Rightarrow 'a \text{ mod-ring} \Rightarrow \text{bool}$ **where**
mod-ring-rel $x \ x' = (x = \text{to-int-mod-ring } x')$

lemma *Domainp-mod-ring-rel* [*transfer-domain-rule*]:
 $\text{Domainp } (\text{mod-ring-rel}) = (\lambda v . v \in \{0 ..< p\})$
<proof>

lemma *bi-unique-mod-ring-rel* [*transfer-rule*]:
 $\text{bi-unique } \text{mod-ring-rel } \text{left-unique } \text{mod-ring-rel } \text{right-unique } \text{mod-ring-rel}$
<proof>

lemma *right-total-mod-ring-rel* [*transfer-rule*]: $\text{right-total } \text{mod-ring-rel}$
<proof>

3.1.2 Transfer Rules

lemma *mod-ring-0* [*transfer-rule*]: $\text{mod-ring-rel } 0 \ 0$ *<proof>*
lemma *mod-ring-1* [*transfer-rule*]: $\text{mod-ring-rel } 1 \ 1$ *<proof>*

lemma *plus-p-mod-def*: **assumes** $x: x \in \{0 \dots p\}$ **and** $y: y \in \{0 \dots p\}$
shows $\text{plus-p } p \ x \ y = ((x + y) \bmod p)$
 $\langle \text{proof} \rangle$

lemma *mod-ring-plus[transfer-rule]*: $(\text{mod-ring-rel} \implies \text{mod-ring-rel} \implies \text{mod-ring-rel})$
 $(\text{plus-p } p) \ (+)$
 $\langle \text{proof} \rangle$

lemma *minus-p-mod-def*: **assumes** $x: x \in \{0 \dots p\}$ **and** $y: y \in \{0 \dots p\}$
shows $\text{minus-p } p \ x \ y = ((x - y) \bmod p)$
 $\langle \text{proof} \rangle$

lemma *mod-ring-minus[transfer-rule]*: $(\text{mod-ring-rel} \implies \text{mod-ring-rel} \implies \text{mod-ring-rel})$
 $(\text{minus-p } p) \ (-)$
 $\langle \text{proof} \rangle$

lemma *mod-ring-uminus[transfer-rule]*: $(\text{mod-ring-rel} \implies \text{mod-ring-rel})$ $(\text{uminus-p } p) \ \text{uminus}$
 $\langle \text{proof} \rangle$

lemma *mod-ring-mult[transfer-rule]*: $(\text{mod-ring-rel} \implies \text{mod-ring-rel} \implies \text{mod-ring-rel})$
 $(\text{mult-p } p) \ ((*))$
 $\langle \text{proof} \rangle$

lemma *mod-ring-eq[transfer-rule]*: $(\text{mod-ring-rel} \implies \text{mod-ring-rel} \implies (=))$
 $(=) \ (=)$
 $\langle \text{proof} \rangle$

lemma *mod-ring-power[transfer-rule]*: $(\text{mod-ring-rel} \implies (=) \implies \text{mod-ring-rel})$
 $(\text{power-p } p) \ (\wedge)$
 $\langle \text{proof} \rangle$

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

lemma *ring-finite-field-ops-int*: $\text{ring-ops } (\text{finite-field-ops-int } p) \ \text{mod-ring-rel}$
 $\langle \text{proof} \rangle$
end

locale *prime-field* = *mod-ring-locale* $p \ \text{ty}$ **for** p **and** $\text{ty} :: 'a :: \text{prime-card itself}$
begin

lemma *prime*: $\text{prime } p \ \langle \text{proof} \rangle$

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

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

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

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

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

lemma *mod-ring-rel-unsafe*: **assumes** $x < \text{CARD}(a)$
shows *mod-ring-rel* (*int* *x*) (*of-nat* *x*) $0 < x \implies \text{of-nat } x \neq 0 \text{ :: } a \text{ mod-ring}$
 ⟨*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 applications of transfer.

lifting-forget *mod-ring.lifting*

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

context *mod-ring-locale*
begin

context **fixes** *pp* :: *uint32*

assumes *ppp*: $p = \text{int-of-uint32 } pp$
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* :: $\text{uint32} \Rightarrow \text{int} \Rightarrow \text{bool}$ **where** $\text{urel32 } x \ y = (y = \text{int-of-uint32 } x \wedge y < p)$

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

lemma *urel32-0*: $\text{urel32 } 0 \ 0$ *<proof>*

lemma *urel32-1*: $\text{urel32 } 1 \ 1$ *<proof>*

lemma *le-int-of-uint32*: $(x \leq y) = (\text{int-of-uint32 } x \leq \text{int-of-uint32 } y)$
<proof>

lemma *urel32-plus*: **assumes** $\text{urel32 } x \ y \ \text{urel32 } x' \ y'$
shows $\text{urel32 } (\text{plus-p32 } pp \ x \ x') \ (\text{plus-p } p \ y \ y')$
<proof>

lemma *urel32-minus*: **assumes** $\text{urel32 } x \ y \ \text{urel32 } x' \ y'$
shows $\text{urel32 } (\text{minus-p32 } pp \ x \ x') \ (\text{minus-p } p \ y \ y')$
<proof>

lemma *urel32-uminus*: **assumes** $\text{urel32 } x \ y$
shows $\text{urel32 } (\text{uminus-p32 } pp \ x) \ (\text{uminus-p } p \ y)$
<proof>

lemma *urel32-mult*: **assumes** $\text{urel32 } x \ y \ \text{urel32 } x' \ y'$
shows $\text{urel32 } (\text{mult-p32 } pp \ x \ x') \ (\text{mult-p } p \ y \ y')$
<proof>

lemma *urel32-eq*: **assumes** $\text{urel32 } x \ y \ \text{urel32 } x' \ y'$
shows $(x = x') = (y = y')$
<proof>

lemma *urel32-normalize*:
assumes $x: \text{urel32 } x \ y$
shows $\text{urel32 } (\text{if } x = 0 \ \text{then } 0 \ \text{else } 1) \ (\text{if } y = 0 \ \text{then } 0 \ \text{else } 1)$
<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 $x' \implies$ *urel32* y (*int* y^\wedge) \implies *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* $\implies \implies \implies$ *mod-ring-rel32*) (*uminus-p32* pp) *uminus*

<proof>

lemma *mod-ring-plus32*: (*mod-ring-rel32* $\implies \implies \implies$ *mod-ring-rel32* $\implies \implies$ *mod-ring-rel32*) (*plus-p32* pp) (+)

<proof>

lemma *mod-ring-minus32*: (*mod-ring-rel32* $\implies \implies \implies$ *mod-ring-rel32* $\implies \implies$ *mod-ring-rel32*) (*minus-p32* pp) (-)

<proof>

lemma *mod-ring-mult32*: (*mod-ring-rel32* $\implies \implies \implies$ *mod-ring-rel32* $\implies \implies$ *mod-ring-rel32*) (*mult-p32* pp) ((*))

<proof>

lemma *mod-ring-eq32*: (*mod-ring-rel32* $\implies \implies \implies$ *mod-ring-rel32* $\implies \implies$ (=)) (=)

<proof>

lemma *urel32-inj*: *urel32* x $y \implies$ *urel32* x $z \implies$ $y = z$

<proof>

lemma *urel32-inj'*: *urel32* x $z \implies$ *urel32* y $z \implies$ $x = y$

<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 = ($\lambda x. 0 \leq x \wedge x <$*

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) ($\lambda x. \text{if } x$*

= 0 then 0 else 1) normalize

<proof>

lemma *mod-ring-mod32*: *(mod-ring-rel32 ==> mod-ring-rel32 ==> mod-ring-rel32)*

($\lambda x y. \text{if } y = 0 \text{ then } x \text{ else } 0) (\text{mod})$

<proof>

lemma *mod-ring-unit-factor32*: *(mod-ring-rel32 ==> mod-ring-rel32) ($\lambda x. x$*

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* \Rightarrow *uint64* \Rightarrow *uint64* **where**
plus-p64 *x y* \equiv *let* *z* = *x* + *y* *in* *if* *z* \geq *p* *then* *z* - *p* *else* *z*

definition *minus-p64* :: *uint64* \Rightarrow *uint64* \Rightarrow *uint64* **where**
minus-p64 *x y* \equiv *if* *y* \leq *x* *then* *x* - *y* *else* (*x* + *p*) - *y*

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

definition *mult-p64* :: *uint64* \Rightarrow *uint64* \Rightarrow *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* ($2^{\wedge} k$)
(*proof*)

lemma *int-of-uint64-0-iff*: *int-of-uint64* *n* = 0 \longleftrightarrow *n* = 0
(*proof*)

lemma *int-of-uint64-0*: *int-of-uint64* 0 = 0 (*proof*)

lemma *int-of-uint64-ge-0*: *int-of-uint64* *n* \geq 0
(*proof*)

lemma *two-64*: $2^{\wedge} \text{LENGTH}(64)$ = (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*: $0 \leq x \implies x < 18446744073709551616 \implies \text{int-of-uint64} (\text{uint64-of-int } x) = x$
(*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 ≤ *x* ∧ *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*: $p = \text{int-of-uint64 } pp$
and *small*: $p \leq 4294967295$
begin

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

definition *urel64* :: $\text{uint64} \Rightarrow \text{int} \Rightarrow \text{bool}$ **where** $\text{urel64 } x \ y = (y = \text{int-of-uint64 } x \wedge y < p)$

definition *mod-ring-rel64* :: $\text{uint64} \Rightarrow 'a \ \text{mod-ring} \Rightarrow \text{bool}$ **where**
 $\text{mod-ring-rel64 } x \ y = (\exists z. \text{urel64 } x \ z \wedge \text{mod-ring-rel } z \ y)$

lemma *urel64-0*: $\text{urel64 } 0 \ 0$ *<proof>*

lemma *urel64-1*: $\text{urel64 } 1 \ 1$ *<proof>*

lemma *le-int-of-uint64*: $(x \leq y) = (\text{int-of-uint64 } x \leq \text{int-of-uint64 } y)$
<proof>

lemma *urel64-plus*: **assumes** $\text{urel64 } x \ y \ \text{urel64 } x' \ y'$
shows $\text{urel64 } (\text{plus-p64 } pp \ x \ x') \ (\text{plus-p } p \ y \ y')$
<proof>

lemma *urel64-minus*: **assumes** $\text{urel64 } x \ y \ \text{urel64 } x' \ y'$
shows $\text{urel64 } (\text{minus-p64 } pp \ x \ x') \ (\text{minus-p } p \ y \ y')$
<proof>

lemma *urel64-uminus*: **assumes** $\text{urel64 } x \ y$
shows $\text{urel64 } (\text{uminus-p64 } pp \ x) \ (\text{uminus-p } p \ y)$
<proof>

lemma *urel64-mult*: **assumes** $\text{urel64 } x \ y \ \text{urel64 } x' \ y'$
shows $\text{urel64 } (\text{mult-p64 } pp \ x \ x') \ (\text{mult-p } p \ y \ y')$
<proof>

lemma *urel64-eq*: **assumes** $\text{urel64 } x \ y \ \text{urel64 } x' \ y'$
shows $(x = x') = (y = y')$
<proof>

lemma *urel64-normalize*:
assumes $x: \text{urel64 } x \ y$
shows $\text{urel64 } (\text{if } x = 0 \ \text{then } 0 \ \text{else } 1) \ (\text{if } y = 0 \ \text{then } 0 \ \text{else } 1)$
<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 $x' \implies$ *urel64* y (*int* y^\wedge) \implies *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* $\implies \implies \implies$ *mod-ring-rel64*) (*uminus-p64* pp) *uminus*

<proof>

lemma *mod-ring-plus64*: (*mod-ring-rel64* $\implies \implies \implies$ *mod-ring-rel64* $\implies \implies$ *mod-ring-rel64*) (*plus-p64* pp) (+)

<proof>

lemma *mod-ring-minus64*: (*mod-ring-rel64* $\implies \implies \implies$ *mod-ring-rel64* $\implies \implies$ *mod-ring-rel64*) (*minus-p64* pp) (-)

<proof>

lemma *mod-ring-mult64*: (*mod-ring-rel64* $\implies \implies \implies$ *mod-ring-rel64* $\implies \implies$ *mod-ring-rel64*) (*mult-p64* pp) ((*))

<proof>

lemma *mod-ring-eq64*: (*mod-ring-rel64* $\implies \implies \implies$ *mod-ring-rel64* $\implies \implies$ (=)) (=)

<proof>

lemma *urel64-inj*: *urel64* x $y \implies$ *urel64* x $z \implies$ $y = z$

<proof>

lemma *urel64-inj'*: *urel64* x $z \implies$ *urel64* y $z \implies$ $x = y$

<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 = (λx. 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) (λx. if x = 0 then 0 else 1) normalize*
⟨*proof*⟩

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

lemma *mod-ring-unit-factor64*: *(mod-ring-rel64 ==> mod-ring-rel64) (λx. 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 $x\ y \equiv \text{let } z = x + y \text{ in if } z \geq p \text{ then } z - p \text{ else } z$

definition *minus-p-integer* :: *integer* \Rightarrow *integer* \Rightarrow *integer* **where**
minus-p-integer $x\ y \equiv \text{if } y \leq x \text{ then } x - y \text{ else } (x + p) - y$

definition *uminus-p-integer* :: *integer* \Rightarrow *integer* **where**
uminus-p-integer $x = (\text{if } x = 0 \text{ then } 0 \text{ else } p - x)$

definition *mult-p-integer* :: *integer* \Rightarrow *integer* \Rightarrow *integer* **where**
mult-p-integer $x\ y = (x * y \bmod p)$

lemma *int-of-integer-0-iff*: *int-of-integer* $n = 0 \iff n = 0$
(*proof*)

lemma *int-of-integer-0*: *int-of-integer* $0 = 0$ (*proof*)

lemma *int-of-integer-plus*: *int-of-integer* $(x + y) = (\text{int-of-integer } x + \text{int-of-integer } y)$
(*proof*)

lemma *int-of-integer-minus*: *int-of-integer* $(x - y) = (\text{int-of-integer } x - \text{int-of-integer } y)$
(*proof*)

lemma *int-of-integer-mult*: *int-of-integer* $(x * y) = (\text{int-of-integer } x * \text{int-of-integer } y)$
(*proof*)

lemma *int-of-integer-mod*: *int-of-integer* $(x \bmod y) = (\text{int-of-integer } x \bmod \text{int-of-integer } y)$
(*proof*)

lemma *int-of-integer-inv*: *int-of-integer* $(\text{integer-of-int } x) = x$ (*proof*)

lemma *int-of-integer-shift*: *int-of-integer* $(\text{drop-bit } k\ n) = (\text{int-of-integer } n) \text{ div } (2^k)$
(*proof*)

context

includes *bit-operations-syntax*

begin

function *power-p-integer* :: *integer* \Rightarrow *integer* \Rightarrow *integer* **where**
power-p-integer $x\ n = (\text{if } n \leq 0 \text{ then } 1 \text{ else}$
 let *rec* = *power-p-integer* $(\text{mult-p-integer } x\ x) (\text{drop-bit } 1\ n)$ *in*
 *if } n \text{ AND } 1 = 0 \text{ then } \text{rec} \text{ else } \text{mult-p-integer } \text{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-p-integer* :: integer \Rightarrow integer **where**
inverse-p-integer $x = (\text{if } x = 0 \text{ then } 0 \text{ else } \text{power-p-integer } x (p - 2))$

definition *divide-p-integer* :: integer \Rightarrow integer \Rightarrow integer **where**
divide-p-integer $x y = \text{mult-p-integer } x (\text{inverse-p-integer } y)$

definition *finite-field-ops-integer* :: integer arith-ops-record **where**
finite-field-ops-integer \equiv *Arith-Ops-Record*

0
1
plus-p-integer
mult-p-integer
minus-p-integer
uminus-p-integer
divide-p-integer
inverse-p-integer
 $(\lambda x y . \text{if } y = 0 \text{ then } x \text{ else } 0)$
 $(\lambda x . \text{if } x = 0 \text{ then } 0 \text{ else } 1)$
 $(\lambda x . x)$
integer-of-int
int-of-integer
 $(\lambda x . 0 \leq x \wedge x < p)$

end

lemma *shiftr-integer-code* [*code-unfold*]: *drop-bit 1 x = (integer-shiftr x 1)*
<proof>

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

context *mod-ring-locale*
begin

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

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

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

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

lemma *urel-integer-0*: *urel-integer* 0 0 \langle *proof* \rangle

lemma *urel-integer-1*: *urel-integer* 1 1 \langle *proof* \rangle

lemma *le-int-of-integer*: (*x* \leq *y*) = (*int-of-integer* *x* \leq *int-of-integer* *y*)
 \langle *proof* \rangle

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'*)
 \langle *proof* \rangle

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'*)
 \langle *proof* \rangle

lemma *urel-integer-uminus*: **assumes** *urel-integer* *x y*
shows *urel-integer* (*uminus-p-integer* *pp x*) (*uminus-p* *p y*)
 \langle *proof* \rangle

lemma *pp-pos*: *int-of-integer* *pp* > 0
 \langle *proof* \rangle

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'*)
 \langle *proof* \rangle

lemma *urel-integer-eq*: **assumes** *urel-integer* *x y* *urel-integer* *x' y'*
shows (*x* = *x'*) = (*y* = *y'*)
 \langle *proof* \rangle

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)
 \langle *proof* \rangle

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)
 \langle *proof* \rangle

lemma *urel-integer-power*: *urel-integer* *x x'* \Longrightarrow *urel-integer* *y* (*int* *y'*) \Longrightarrow *urel-integer*

$(\text{power-}p\text{-integer } pp \ x \ y) \ (\text{power-}p \ p \ x' \ y')$
including *bit-operations-syntax* $\langle \text{proof} \rangle$

lemma *urel-integer-inverse*: **assumes** x : *urel-integer* $x \ x'$
shows *urel-integer* $(\text{inverse-}p\text{-integer } pp \ x) \ (\text{inverse-}p \ p \ x')$
 $\langle \text{proof} \rangle$

lemma *mod-ring-0--integer*: *mod-ring-rel-integer* $0 \ 0$
 $\langle \text{proof} \rangle$

lemma *mod-ring-1--integer*: *mod-ring-rel-integer* $1 \ 1$
 $\langle \text{proof} \rangle$

lemma *mod-ring-uminus-integer*: $(\text{mod-ring-rel-integer} \implies \text{mod-ring-rel-integer})$
 $(\text{uminus-}p\text{-integer } pp) \ \text{uminus}$
 $\langle \text{proof} \rangle$

lemma *mod-ring-plus-integer*: $(\text{mod-ring-rel-integer} \implies \text{mod-ring-rel-integer})$
 $\implies \text{mod-ring-rel-integer} \ (\text{plus-}p\text{-integer } pp) \ (+)$
 $\langle \text{proof} \rangle$

lemma *mod-ring-minus-integer*: $(\text{mod-ring-rel-integer} \implies \text{mod-ring-rel-integer})$
 $\implies \text{mod-ring-rel-integer} \ (\text{minus-}p\text{-integer } pp) \ (-)$
 $\langle \text{proof} \rangle$

lemma *mod-ring-mult-integer*: $(\text{mod-ring-rel-integer} \implies \text{mod-ring-rel-integer})$
 $\implies \text{mod-ring-rel-integer} \ (\text{mult-}p\text{-integer } pp) \ ((*))$
 $\langle \text{proof} \rangle$

lemma *mod-ring-eq-integer*: $(\text{mod-ring-rel-integer} \implies \text{mod-ring-rel-integer} \implies)$
 $(=) \ (=) \ (=)$
 $\langle \text{proof} \rangle$

lemma *urel-integer-inj*: *urel-integer* $x \ y \implies \text{urel-integer } x \ z \implies y = z$
 $\langle \text{proof} \rangle$

lemma *urel-integer-inj'*: *urel-integer* $x \ z \implies \text{urel-integer } y \ z \implies x = y$
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

lemma *right-total-mod-ring-rel-integer*: *right-total mod-ring-rel-integer*
 $\langle \text{proof} \rangle$

lemma *Domainp-mod-ring-rel-integer*: *Domainp mod-ring-rel-integer* $= (\lambda x. 0 \leq x \wedge x < pp)$

```

⟨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)
  (λx. 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) (λx y. if y = 0 then x else 0) (mod)
  ⟨proof⟩

lemma mod-ring-unit-factor-integer: (mod-ring-rel-integer ===> mod-ring-rel-integer)
  (λx. 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* $R \implies \text{left-unique } (\text{vec-rel } R)$
 ⟨proof⟩

lemma *right-unique-vec-rel*: *right-unique* $R \implies \text{right-unique } (\text{vec-rel } R)$
 ⟨proof⟩

lemma *bi-unique-vec-rel*: *bi-unique* $R \implies \text{bi-unique } (\text{vec-rel } R)$
 ⟨proof⟩

lemma *vec-rel-eq*: $((R \implies R \implies (=))) (=) (=) \implies$
 $((\text{vec-rel } R \implies \text{vec-rel } R \implies (=))) (=) (=)$
 ⟨proof⟩

lemma *multrow-transfer*[*transfer-rule*]: $((R \implies R \implies R) \implies (=) \implies R$
 $\implies \text{mat-rel } R \implies \text{mat-rel } R) \text{ mat-multrow-gen mat-multrow-gen}$
 ⟨proof⟩

lemma *swap-rows-transfer*: *mat-rel* $R \ A \ B \implies i < \text{dim-row } B \implies j < \text{dim-row}$
 $B \implies$
 $\text{mat-rel } R \ (\text{mat-swaprows } i \ j \ A) \ (\text{mat-swaprows } i \ j \ B)$
 ⟨proof⟩

lemma *pivot-positions-gen-transfer*: **assumes** [*transfer-rule*]: $(R \implies R \implies$
 $(=)) (=) (=)$
shows
 $(R \implies \text{mat-rel } R \implies (=)) \text{ pivot-positions-gen pivot-positions-gen}$
 ⟨proof⟩

lemma *set-pivot-positions-main-gen*:
 $\text{set } (\text{pivot-positions-main-gen } ze \ A \ nr \ nc \ i \ j) \subseteq \{0 \ ..< nr\} \times \{0 \ ..< nc\}$
 ⟨proof⟩

lemma *find-base-vectors-transfer*: **assumes** [*transfer-rule*]: $(R \implies R \implies$
 $(=)) (=) (=)$
shows $((R \implies R) \implies R \implies R \implies \text{mat-rel } R$
 $\implies \text{list-all2 } (\text{vec-rel } R)) \text{ find-base-vectors-gen find-base-vectors-gen}$
 ⟨proof⟩

lemma *eliminate-entries-gen-transfer*: **assumes** *[*transfer-rule*]: $(R \implies R \implies$
 $R) \text{ ad ad'}$
 $(R \implies R \implies R) \text{ mul mul'}$
and $vs: \bigwedge j. j < \text{dim-row } B' \implies R \ (vs \ j) \ (vs' \ j)$
and $i: i < \text{dim-row } B'$
and $B: \text{mat-rel } R \ B \ B'$
shows *mat-rel* R
 $(\text{eliminate-entries-gen } ad \ mul \ vs \ B \ i \ j)$

(*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 iaaj = inverse aij; A' = multrow-i i iaaj A;

$v = (\lambda i. A' \$\$ (\text{nat-of-integer } i, j))$
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 \Rightarrow 'i mat **where**
gauss-jordan-single-i A \equiv gauss-jordan-main-i (dim-row A) (dim-col A) A 0 0

definition *find-base-vectors-i* :: 'i mat \Rightarrow 'i vec list **where**
find-base-vectors-i A \equiv 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 \implies mat-rel R \implies (=)) (=)*
<proof>

lemma *multrow-i[transfer-rule]*: *((=) \implies R \implies mat-rel R \implies 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: $\bigwedge j. j < \text{dim-row } B' \implies R (vs (\text{integer-of-nat } j)) (vs' j)$
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' \implies nc = dim-col A' \implies mat-rel R A A' \implies i \leq nr \implies j \leq
nc \implies

$mat\text{-}rel\ R\ (gauss\text{-}jordan\text{-}main\text{-}i\ ops\ nr\ nc\ A\ i\ j)\ (fst\ (gauss\text{-}jordan\text{-}main\ A'\ B'\ i\ j))$
 $\langle proof \rangle$

lemma $gauss\text{-}jordan\text{-}i[transfer\text{-}rule]$:
 $(mat\text{-}rel\ R\ ==\Rightarrow\ mat\text{-}rel\ R)\ (gauss\text{-}jordan\text{-}single\text{-}i\ ops)\ gauss\text{-}jordan\text{-}single$
 $\langle proof \rangle$

lemma $find\text{-}base\text{-}vectors\text{-}i[transfer\text{-}rule]$:
 $(mat\text{-}rel\ R\ ==\Rightarrow\ list\text{-}all2\ (vec\text{-}rel\ R))\ (find\text{-}base\text{-}vectors\text{-}i\ ops)\ find\text{-}base\text{-}vectors$
 $\langle proof \rangle$

end

lemma $list\text{-}of\text{-}vec\text{-}transfer[transfer\text{-}rule]$: $(vec\text{-}rel\ A\ ==\Rightarrow\ list\text{-}all2\ A)\ list\text{-}of\text{-}vec\ list\text{-}of\text{-}vec$
 $\langle proof \rangle$

lemma $IArray\text{-}sub'[simp]$: $i < IArray.length\ a\ \Rightarrow\ IArray.sub'\ (a,\ integer\text{-}of\text{-}nat\ i) = IArray.sub\ a\ i$
 $\langle proof \rangle$

lift-definition $eliminate\text{-}entries\text{-}i2\ ::$
 $'a\ \Rightarrow\ ('a\ \Rightarrow\ 'a\ \Rightarrow\ 'a)\ \Rightarrow\ ('a\ \Rightarrow\ 'a\ \Rightarrow\ 'a)\ \Rightarrow\ (integer\ \Rightarrow\ 'a)\ \Rightarrow\ 'a\ mat\text{-}impl\ \Rightarrow\ integer\ \Rightarrow\ 'a\ mat\text{-}impl\ \mathbf{is}$
 $\lambda\ z\ mminus\ ttimes\ v\ (nr,\ nc,\ a)\ i'.$
 $(nr,\ nc,\ let\ ai' = IArray.sub'\ (a,\ i')\ in\ (IArray.tabulate\ (integer\text{-}of\text{-}nat\ nr,\ \lambda\ i.\ let\ ai = IArray.sub'\ (a,\ i)\ in\ if\ i = i'\ then\ ai\ else\ let\ vi'j = v\ i\ in\ if\ vi'j = z\ then\ ai\ else\ IArray.tabulate\ (integer\text{-}of\text{-}nat\ nc,\ \lambda\ j.\ mminus\ (IArray.sub'\ (ai,\ j))\ (ttimes\ vi'j\ (IArray.sub'\ (ai',\ j))))))$
 $\langle proof \rangle$

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

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

(if $i = I$ then $A \text{ $$ } (i,j)$ else $mminus (A \text{ $$ } (i,j)) (ttimes (v i) (A \text{ $$ } (I,j)))$)
 <proof>

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

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

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

lemma *eliminate-entries-i2*[code]: $eliminate\text{-entries-gen-zero } mm \text{ tt } z \text{ v } (mat\text{-impl } m) \text{ i } j$
 $= (if \text{ i } < dim\text{-row-impl } m$
 then $mat\text{-impl } (eliminate\text{-entries-i2 } z \text{ mm } tt \text{ v } m (integer\text{-of-nat } i))$
 else $(Code.abort (STR "index out of range in eliminate-entries")$
 $(\lambda \text{ -. } eliminate\text{-entries-gen-zero } mm \text{ tt } z \text{ v } (mat\text{-impl } m) \text{ i } j))$)

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

lemma *rel-mset-free*:
assumes $rel: rel\text{-mset } rel \text{ X } Y$ **and** $xs: mset \text{ xs} = X$
shows $\exists ys. mset \text{ ys} = Y \wedge list\text{-all2 } rel \text{ xs } ys$
 <proof>

lemma *rel-mset-split*:
assumes $rel: rel\text{-mset } rel (X1+X2) Y$
shows $\exists Y1 \text{ Y2. } Y = Y1 + Y2 \wedge rel\text{-mset } rel \text{ X1 } Y1 \wedge rel\text{-mset } rel \text{ X2 } Y2$
 <proof>

lemma *rel-mset-OO*:
assumes $AB: rel\text{-mset } R \text{ A } B$ **and** $BC: rel\text{-mset } S \text{ B } C$
shows $rel\text{-mset } (R \text{ OO } S) \text{ A } C$
 <proof>

lemma *ex-mset-zip-right*:

assumes $\text{length } xs = \text{length } ys \text{ mset } ys' = \text{mset } ys$
shows $\exists xs'. \text{length } ys' = \text{length } xs' \wedge \text{mset } (\text{zip } xs' ys') = \text{mset } (\text{zip } xs \text{ } ys)$
 $\langle \text{proof} \rangle$

lemma *list-all2-reorder-right-invariance*:
assumes $\text{rel}: \text{list-all2 } R \text{ } xs \text{ } ys$ **and** $\text{ms-y}: \text{mset } ys' = \text{mset } ys$
shows $\exists xs'. \text{list-all2 } R \text{ } xs' \text{ } ys' \wedge \text{mset } xs' = \text{mset } xs$
 $\langle \text{proof} \rangle$

lemma *rel-mset-via-perm*: $\text{rel-mset } \text{rel } (\text{mset } xs) (\text{mset } ys) \longleftrightarrow (\exists zs. \text{mset } xs = \text{mset } zs \wedge \text{list-all2 } \text{rel } zs \text{ } ys)$
 $\langle \text{proof} \rangle$

end

theory *Unique-Factorization*

imports

Polynomial-Interpolation.Ring-Hom-Poly
Polynomial-Factorization.Polynomial-Divisibility
HOL-Combinatorics.Permutations
HOL-Computational-Algebra.Euclidean-Algorithm
Containers.Containers-Auxiliary
More-Missing-Multiset
HOL-Algebra.Divisibility

begin

hide-const(**open**)

Divisibility.prime
Divisibility.irreducible

hide-fact(**open**)

Divisibility.irreducible-def
Divisibility.irreducibleI
Divisibility.irreducibleD
Divisibility.irreducibleE

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

lemma *irreducible-uminus* [*simp*]:

fixes $a::'a::\text{idom}$
shows $\text{irreducible } (-a) \longleftrightarrow \text{irreducible } a$
 $\langle \text{proof} \rangle$

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

definition *coprime* $:: 'a \Rightarrow 'a \Rightarrow \text{bool}$

where *coprime-def*: $\text{coprime } p \text{ } q \equiv \forall r. r \text{ dvd } p \longrightarrow r \text{ dvd } q \longrightarrow r \text{ dvd } 1$

lemma *coprimeI*:

assumes $\bigwedge r. r \text{ dvd } p \Longrightarrow r \text{ dvd } q \Longrightarrow r \text{ dvd } 1$

shows *coprime p q* \langle proof \rangle

lemma *coprimeE*:

assumes *coprime p q*

and $(\bigwedge r. r \text{ dvd } p \implies r \text{ dvd } q \implies r \text{ dvd } 1) \implies \textit{thesis}$

shows *thesis* \langle proof \rangle

lemma *coprime-commute* [*ac-simps*]:

coprime p q \longleftrightarrow *coprime q p*

\langle proof \rangle

lemma *not-coprime-iff-common-factor*:

$\neg \textit{coprime p q} \longleftrightarrow (\exists r. r \text{ dvd } p \wedge r \text{ dvd } q \wedge \neg r \text{ dvd } 1)$

\langle proof \rangle

end

lemma (**in** *algebraic-semidom*) *coprime-iff-coprime* [*simp, code*]:

coprime = *Rings.coprime*

\langle proof \rangle

lemma (**in** *comm-semiring-1*) *coprime-0* [*simp*]:

coprime p 0 \longleftrightarrow *p dvd 1* *coprime 0 p* \longleftrightarrow *p dvd 1*

\langle proof \rangle

lemma *dvd-rewrites*: *dvd.dvd* $((*) = (dvd))$ \langle proof \rangle

3.3 Interfacing UFD properties

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

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

lemma *coprime-hom*[*simp*]: *coprime (hom x) y'* \longleftrightarrow *coprime x* (*Hilbert-Choice.inv hom y'*)

\langle proof \rangle

lemma *coprime-inv-hom*[*simp*]: *coprime (Hilbert-Choice.inv hom x')* *y* \longleftrightarrow *coprime x'* (*hom y*)

\langle proof \rangle

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** $\exists c. p = \textit{smult c q}$

\langle proof \rangle

lemma *dvd-const*:
 assumes $pq: (p::'a::\text{semidom poly}) \text{ dvd } q$ and $q0: q \neq 0$ and $\text{deg}q: \text{degree } q = 0$
 shows $\text{degree } p = 0$
 $\langle \text{proof} \rangle$

context *Rings.dvd* **begin**
 abbreviation *ddvd* (**infix** *ddvd* 40) **where** $x \text{ ddvd } y \equiv x \text{ dvd } y \wedge y \text{ dvd } x$
 lemma *ddvd-sym[sym]*: $x \text{ ddvd } y \implies y \text{ ddvd } x$ $\langle \text{proof} \rangle$
end

context *comm-monoid-mult* **begin**
 lemma *ddvd-trans[trans]*: $x \text{ ddvd } y \implies y \text{ ddvd } z \implies x \text{ ddvd } z$ $\langle \text{proof} \rangle$
 lemma *ddvd-transp*: $\text{transp } (\text{ddvd})$ $\langle \text{proof} \rangle$
end

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

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

lemma *mset-factorsI[intro!]*:
 assumes $\bigwedge f. f \in \# F \implies \text{irreducible } f$ and $F \neq \{\#\}$ and $\text{prod-mset } F = p$
 shows $\text{mset-factors } F \text{ } p$
 $\langle \text{proof} \rangle$

lemma *mset-factorsD*:
 assumes $\text{mset-factors } F \text{ } p$
 shows $f \in \# F \implies \text{irreducible } f$ and $F \neq \{\#\}$ and $\text{prod-mset } F = p$
 $\langle \text{proof} \rangle$

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

lemma *mset-factors-imp-not-is-unit*:
 assumes $\text{mset-factors } F \text{ } p$
 shows $\neg p \text{ dvd } 1$
 $\langle \text{proof} \rangle$

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

end

lemma(**in** *semidom*) *mset-factors-imp-nonzero*:

assumes *mset-factors* F p
shows $p \neq 0$
 \langle *proof* \rangle

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

3.3.2 Connecting to HOL/Divisibility

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

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

lemma *carrier-0[simp]*: $x \in \text{carrier } \text{mk-monoid} \longleftrightarrow x \neq 0$ \langle *proof* \rangle

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

abbreviation *irred* **where** *irred* $\equiv \text{Divisibility.irreducible } \text{mk-monoid}$

abbreviation *factor* **where** *factor* $\equiv \text{Divisibility.factor } \text{mk-monoid}$

abbreviation *factors* **where** *factors* $\equiv \text{Divisibility.factors } \text{mk-monoid}$

abbreviation *properfactor* **where** *properfactor* $\equiv \text{Divisibility.properfactor } \text{mk-monoid}$

lemma *factors*: *factors* fs $y \longleftrightarrow \text{prod-list } fs = y \wedge \text{Ball } (\text{set } fs) \text{ irred}$
 \langle *proof* \rangle

lemma *factor*: *factor* x $y \longleftrightarrow (\exists z. z \neq 0 \wedge x * z = y)$ \langle *proof* \rangle

lemma *properfactor-nz*:

shows $(y :: 'a) \neq 0 \implies \text{properfactor } x y \longleftrightarrow x \text{ dvd } y \wedge \neg y \text{ dvd } x$
 \langle *proof* \rangle

lemma *mem-Units[simp]*: $y \in \text{Units } \text{mk-monoid} \longleftrightarrow y \text{ dvd } 1$
 \langle *proof* \rangle

end

context *idom* **begin**

lemma *irred-0[simp]*: *irred* $(0::'a)$ \langle *proof* \rangle

lemma *factor-idom[simp]*: *factor* $(x::'a)$ $y \longleftrightarrow (\text{if } y = 0 \text{ then } x = 0 \text{ else } x \text{ dvd } y)$
 \langle *proof* \rangle

lemma *associated-connect[simp]*: $(\sim_{\text{mk-monoid}}) = (ddvd)$ \langle *proof* \rangle

lemma *essentially-equal-connect[simp]*:

essentially-equal mk-monoid fs $gs \longleftrightarrow \text{rel-mset } (ddvd) (\text{mset } fs) (\text{mset } gs)$
 \langle *proof* \rangle

lemma *irred-idom-nz*:
assumes $x0: (x::'a) \neq 0$
shows $\text{irred } x \longleftrightarrow \text{irreducible } x$
 $\langle \text{proof} \rangle$

lemma *dvd-dvd-imp-unit-mult*:
assumes $xy: x \text{ dvd } y$ **and** $yx: y \text{ dvd } x$
shows $\exists z. z \text{ dvd } 1 \wedge y = x * z$
 $\langle \text{proof} \rangle$

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

lemma *irred-idom[simp]*: $\text{irred } x \longleftrightarrow x = 0 \vee \text{irreducible } x$
 $\langle \text{proof} \rangle$

lemma **assumes** $x \neq 0$ **and** *factors fs x* **and** $f \in \text{set } fs$ **shows** $f \neq 0$
 $\langle \text{proof} \rangle$

lemma *factors-as-mset-factors*:
assumes $x0: x \neq 0$ **and** $x1: x \neq 1$
shows $\text{factors } fs \ x \longleftrightarrow \text{mset-factors } (\text{mset } fs) \ x$ $\langle \text{proof} \rangle$

end

context *ufd* **begin**

interpretation *comm-monoid-cancel*: *comm-monoid-cancel mk-monoid::'a monoid*
 $\langle \text{proof} \rangle$

lemma *factors-exist*:
assumes $a \neq 0$
and $\neg a \text{ dvd } 1$
shows $\exists fs. \text{set } fs \subseteq \text{UNIV} - \{0\} \wedge \text{factors } fs \ a$
 $\langle \text{proof} \rangle$

lemma *factors-unique*:
assumes $fs: \text{factors } fs \ a$
and $gs: \text{factors } gs \ a$
and $a0: a \neq 0$
and $a1: \neg a \text{ dvd } 1$
shows $\text{rel-mset } (\text{dvd}) \ (\text{mset } fs) \ (\text{mset } gs)$
 $\langle \text{proof} \rangle$

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 ⇒ *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*: $\bigwedge x. \text{hom } x \text{ dvd } 1 \implies x \text{ dvd } 1$

begin

lemma *is-unit-hom-iff[simp]*: $\text{hom } x \text{ dvd } 1 \longleftrightarrow x \text{ dvd } 1$ ⟨*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]*: $\text{irreducible} (\text{hom } a) \longleftrightarrow \text{irreducible } a$
 ⟨*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 $\exists G'. G' \subseteq\# G \wedge \text{rel-mset } (dvd) F G'$
 ⟨*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 $\exists g \in\# G. a \text{ } dvd \text{ } g$
 ⟨*proof*⟩

lemma(in *idom*) *prod-mset-remove-units*:
 prod-mset *F* *dvd* *prod-mset* $\{\# f \in\# F. \neg f \text{ } dvd \text{ } 1 \#\}$
 ⟨*proof*⟩

lemma(in *comm-semiring-1*) *mset-factors-imp-dvd*:
 assumes *mset-factors* *F* *x* and $f \in\# F$ 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*: $(x :: 'b :: \text{comm-monoid-mult}) \in \text{set } xs \implies \text{prod-list } (\text{remove1 } x \text{ } xs) * x = \text{prod-list } xs$
 ⟨*proof*⟩

class *comm-monoid-gcd* = *gcd* + *comm-semiring-1* +

assumes gcd-dvd1 [*iff*]: $\text{gcd } a \ b \ \text{dvd } a$
and gcd-dvd2 [*iff*]: $\text{gcd } a \ b \ \text{dvd } b$
and gcd-greatest : $c \ \text{dvd } a \implies c \ \text{dvd } b \implies c \ \text{dvd } \text{gcd } a \ b$
begin

lemma gcd-0-0 [*simp*]: $\text{gcd } 0 \ 0 = 0$
 <proof>

lemma gcd-zero-iff [*simp*]: $\text{gcd } a \ b = 0 \iff a = 0 \wedge b = 0$
 <proof>

lemma gcd-zero-iff' [*simp*]: $0 = \text{gcd } a \ b \iff a = 0 \wedge b = 0$
 <proof>

lemma dvd-gcd-0-iff [*simp*]:
shows $x \ \text{dvd } \text{gcd } 0 \ a \iff x \ \text{dvd } a$ (**is** ?g1)
and $x \ \text{dvd } \text{gcd } a \ 0 \iff x \ \text{dvd } a$ (**is** ?g2)
 <proof>

lemma gcd-dvd-1 [*simp*]: $\text{gcd } a \ b \ \text{dvd } 1 \iff \text{coprime } a \ b$
 <proof>

lemma $\text{dvd-imp-gcd-dvd-gcd}$: $b \ \text{dvd } c \implies \text{gcd } a \ b \ \text{dvd } \text{gcd } a \ c$
 <proof>

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

lemma listgcd-simps [*simp*]: $\text{listgcd } [] = 0$ $\text{listgcd } (x \ \# \ xs) = \text{gcd } x \ (\text{listgcd } xs)$
 <proof>

lemma listgcd : $x \in \text{set } xs \implies \text{listgcd } xs \ \text{dvd } x$
 <proof>

lemma listgcd-greatest : $(\bigwedge x. x \in \text{set } xs \implies y \ \text{dvd } x) \implies y \ \text{dvd } \text{listgcd } xs$
 <proof>

end

context *Rings.dvd* **begin**

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

definition $\text{some-gcd } a \ b \equiv \text{SOME } x. \text{is-gcd } x \ a \ b$

lemma is-gcdI [*intro!*]:
assumes $x \ \text{dvd } a \ x \ \text{dvd } b \ \bigwedge y. y \ \text{dvd } a \implies y \ \text{dvd } b \implies y \ \text{dvd } x$

shows *is-gcd* $x a b$ \langle proof \rangle

lemma *is-gcdE*[*elim!*]:
assumes *is-gcd* $x a b$
and $x \text{ dvd } a \implies x \text{ dvd } b \implies (\bigwedge y. y \text{ dvd } a \implies y \text{ dvd } b \implies y \text{ dvd } x) \implies$
thesis
shows *thesis* \langle proof \rangle

lemma *is-gcd-some-gcdI*:
assumes $\exists x. \text{is-gcd } x a b$ **shows** *is-gcd* (*some-gcd* $a b$) $a b$
 \langle proof \rangle

end

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

lemma *some-gcd-0*[*intro!*]: *is-gcd* (*some-gcd* $a 0$) $a 0$ *is-gcd* (*some-gcd* $0 b$) $0 b$
 \langle proof \rangle

lemma *some-gcd-0-dvd*[*intro!*]:
some-gcd $a 0 \text{ dvd } a$ *some-gcd* $0 b \text{ dvd } b$ \langle proof \rangle

lemma *dvd-some-gcd-0*[*intro!*]:
 $a \text{ dvd } \text{some-gcd } a 0 b \text{ dvd } \text{some-gcd } 0 b$ \langle proof \rangle

end

context *idom* **begin**

lemma *is-gcd-connect*:
assumes $a \neq 0 b \neq 0$ **shows** *isgcd mk-monoid* $x a b \iff \text{is-gcd } x a b$
 \langle proof \rangle

lemma *some-gcd-connect*:
assumes $a \neq 0$ **and** $b \neq 0$ **shows** *somegcd mk-monoid* $a b = \text{some-gcd } a b$
 \langle proof \rangle

end

context *comm-monoid-gcd*
begin

lemma *is-gcd-gcd*: *is-gcd* (*gcd* $a b$) $a b$ \langle proof \rangle
lemma *is-gcd-some-gcd*: *is-gcd* (*some-gcd* $a b$) $a b$ \langle proof \rangle
lemma *gcd-dvd-some-gcd*: *gcd* $a b \text{ dvd } \text{some-gcd } a b$ \langle proof \rangle
lemma *some-gcd-dvd-gcd*: *some-gcd* $a b \text{ dvd } \text{gcd } a b$ \langle proof \rangle
lemma *some-gcd-ddvd-gcd*: *some-gcd* $a b \text{ ddvd } \text{gcd } a b$ \langle proof \rangle
lemma *some-gcd-dvd*: *some-gcd* $a b \text{ dvd } d \iff \text{gcd } a b \text{ dvd } d \text{ dvd } \text{some-gcd } a b$
 $\iff d \text{ dvd } \text{gcd } a b$
 \langle proof \rangle

```

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: (∧ x :: 'a. x ∈ set xs ⇒ y dvd x * z) ⇒ 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: ∧ a b :: 'a. ∃ x. is-gcd x a b
begin
  sublocale idom-gcd (*) 1 :: 'a (+) 0 (-) uminus some-gcd
  rewrites dvd.dvd ((*)) = (dvd)
    and comm-monoid-mult.coprime ((*)) 1 = Unique-Factorization.coprime
end

```



```

    <proof>
end

instance semiring-gcd  $\subseteq$  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  $\subseteq$ 
    gcd-condition vs. factorial-ring-gcd  $\subseteq$  gcd-condition.

sublocale as-ufd: ufd
    <proof>

end

instance int :: ufd <proof>
instance int :: idom-gcd <proof>

instance field  $\subseteq$  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]*: $\text{divides-ff } (z * x) (z * y) \longleftrightarrow z = 0 \vee \text{divides-ff } x y$

and *divides-ff-mult-cancel-right[simp]*: $\text{divides-ff } (x * z) (y * z) \longleftrightarrow z = 0 \vee \text{divides-ff } x y$

<proof>

definition *gcd-ff-list* :: 'a::ufd fract list \Rightarrow 'a fract \Rightarrow bool **where**

gcd-ff-list X g = (
 ($\forall x \in \text{set } X. \text{divides-ff } g x$) \wedge
 ($\forall d. (\forall x \in \text{set } X. \text{divides-ff } d x) \longrightarrow \text{divides-ff } d g$)

lemma *gcd-ff-list-exists*: $\exists g. \text{gcd-ff-list } (X :: 'a::\text{ufd fract list}) g$

<proof>

definition *some-gcd-ff-list* :: 'a :: ufd fract list \Rightarrow '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*: $x \in \text{set } xs \Longrightarrow \text{divides-ff } (\text{some-gcd-ff-list } xs) x$

<proof>

lemma *some-gcd-ff-list-greatest*: $(\forall x \in \text{set } xs. \text{divides-ff } d x) \Longrightarrow \text{divides-ff } d (\text{some-gcd-ff-list } xs)$

<proof>

lemma *divides-ff-refl[simp]*: *divides-ff* x x

<proof>

lemma *divides-ff-trans*:

$\text{divides-ff } x y \Longrightarrow \text{divides-ff } y z \Longrightarrow \text{divides-ff } x z$

<proof>

lemma *divides-ff-mult-right*: $a \neq 0 \Longrightarrow \text{divides-ff } (x * \text{inverse } a) y \Longrightarrow \text{divides-ff } x (a * y)$

<proof>

definition *eq-dff* :: 'a :: ufd fract \Rightarrow 'a fract \Rightarrow bool (**infix** =dff 50) **where**

$x = \text{dff } y \longleftrightarrow \text{divides-ff } x y \wedge \text{divides-ff } y x$

lemma *eq-dffI[intro]*: $\text{divides-ff } x y \Longrightarrow \text{divides-ff } y x \Longrightarrow x = \text{dff } y$

<proof>

lemma *eq-dff-refl[simp]*: $x = \text{dff } x$

<proof>

lemma *eq-dff-sym*: $x = \text{dff } y \implies y = \text{dff } x$ $\langle \text{proof} \rangle$

lemma *eq-dff-trans*[*trans*]: $x = \text{dff } y \implies y = \text{dff } z \implies x = \text{dff } z$
 $\langle \text{proof} \rangle$

lemma *eq-dff-cancel-right*[*simp*]: $x * y = \text{dff } x * z \iff x = 0 \vee y = \text{dff } z$
 $\langle \text{proof} \rangle$

lemma *eq-dff-mult-right-trans*[*trans*]: $x = \text{dff } y * z \implies z = \text{dff } u \implies x = \text{dff } y * u$
 $\langle \text{proof} \rangle$

lemma *some-gcd-ff-list-smult*: $a \neq 0 \implies \text{some-gcd-ff-list } (\text{map } ((*) a) xs) = \text{dff } a$
 $* \text{some-gcd-ff-list } xs$
 $\langle \text{proof} \rangle$

definition *content-ff* :: $'a::\text{ufd fract poly} \Rightarrow 'a \text{ fract}$ **where**
 $\text{content-ff } p = \text{some-gcd-ff-list } (\text{coeffs } p)$

lemma *content-ff-iff*: $\text{divides-ff } x (\text{content-ff } p) \iff (\forall c \in \text{set } (\text{coeffs } p). \text{divides-ff } x c)$ (**is** $?l = ?r$)
 $\langle \text{proof} \rangle$

lemma *content-ff-divides-ff*: $x \in \text{set } (\text{coeffs } p) \implies \text{divides-ff } (\text{content-ff } p) x$
 $\langle \text{proof} \rangle$

lemma *content-ff-0*[*simp*]: $\text{content-ff } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *content-ff-0-iff*[*simp*]: $(\text{content-ff } p = 0) = (p = 0)$
 $\langle \text{proof} \rangle$

lemma *content-ff-eq-dff-nonzero*: $\text{content-ff } p = \text{dff } x \implies x \neq 0 \implies p \neq 0$
 $\langle \text{proof} \rangle$

lemma *content-ff-smult*: $\text{content-ff } (\text{smult } (a::'a::\text{ufd fract}) p) = \text{dff } a * \text{content-ff } p$
 $\langle \text{proof} \rangle$

definition *normalize-content-ff*
where $\text{normalize-content-ff } (p::'a::\text{ufd fract poly}) \equiv \text{smult } (\text{inverse } (\text{content-ff } p)) p$

lemma *smult-normalize-content-ff*: $\text{smult } (\text{content-ff } p) (\text{normalize-content-ff } p) = p$
 $\langle \text{proof} \rangle$

lemma *content-ff-normalize-content-ff-1*: **assumes** $p0: p \neq 0$
shows $\text{content-ff } (\text{normalize-content-ff } p) = \text{dff } 1$

<proof>

lemma *content-ff-to-fract*: **assumes** $\text{set } (\text{coeffs } p) \subseteq \text{range to-fract}$
shows $\text{content-ff } p \in \text{range to-fract}$
<proof>

lemma *content-ff-map-poly-to-fract*: $\text{content-ff } (\text{map-poly to-fract } (p :: 'a :: \text{ufd poly})) \in \text{range to-fract}$
<proof>

lemma *range-coeffs-to-fract*: **assumes** $\text{set } (\text{coeffs } p) \subseteq \text{range to-fract}$
shows $\exists m. \text{coeff } p \ i = \text{to-fract } m$
<proof>

lemma *divides-ff-coeff*: **assumes** $\text{set } (\text{coeffs } p) \subseteq \text{range to-fract}$ **and** $\text{divides-ff } (\text{to-fract } n) (\text{coeff } p \ i)$
shows $\exists m. \text{coeff } p \ i = \text{to-fract } n * \text{to-fract } m$
<proof>

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

lemma *inv-embed[simp]*: $\text{inv-embed } (\text{to-fract } x) = x$
<proof>

lemma *inv-embed-0[simp]*: $\text{inv-embed } 0 = 0$ *<proof>*

lemma *range-to-fract-embed-poly*: **assumes** $\text{set } (\text{coeffs } p) \subseteq \text{range to-fract}$
shows $p = \text{map-poly to-fract } (\text{map-poly inv-embed } p)$
<proof>

lemma *content-ff-to-fract-coeffs-to-fract*: **assumes** $\text{content-ff } p \in \text{range to-fract}$
shows $\text{set } (\text{coeffs } p) \subseteq \text{range to-fract}$
<proof>

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

lemma *gauss-lemma*:
fixes $p \ q :: 'a :: \text{ufd fract poly}$
shows $\text{content-ff } (p * q) = \text{dff } \text{content-ff } p * \text{content-ff } q$
<proof>

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

lemma *factorization-to-fract*:
assumes $q: q \neq 0$ **and** *factor*: $\text{map-poly to-fract } (p :: 'a :: \text{ufd poly}) = q * r$
shows $\exists q' \ r' \ c. c \neq 0 \wedge q = \text{smult } c \ (\text{map-poly to-fract } q') \wedge$

$r = \text{smult } (\text{inverse } c) (\text{map-poly to-fract } r') \wedge$
 $\text{content-ff-ff } q' = \text{dff } 1 \wedge p = q' * r'$
 <proof>

lemma *irreducible-PM-M-PFM*:

assumes *irr*: *irreducible* *p*
shows $\text{degree } p = 0 \wedge \text{irreducible } (\text{coeff } p \ 0) \vee$
 $\text{degree } p \neq 0 \wedge \text{irreducible } (\text{map-poly to-fract } p) \wedge \text{content-ff-ff } p = \text{dff } 1$
 <proof>

lemma *irreducible-M-PM*:

fixes $p :: 'a :: \text{ufd poly}$ **assumes** $0: \text{degree } p = 0$ **and** *irr*: *irreducible* (*coeff* *p* 0)
shows *irreducible* *p*
 <proof>

lemma *primitive-irreducible-imp-degree*:

$\text{primitive } (p :: 'a :: \{\text{semiring-gcd, idom}\} \text{ poly}) \implies \text{irreducible } p \implies \text{degree } p > 0$
 <proof>

lemma *irreducible-degree-field*:

fixes $p :: 'a :: \text{field poly}$ **assumes** *irreducible* *p*
shows $\text{degree } p > 0$
 <proof>

lemma *irreducible-PFM-PM*: **assumes**

irr: *irreducible* (*map-poly to-fract* *p*) **and** *ct*: $\text{content-ff-ff } p = \text{dff } 1$
shows *irreducible* *p*
 <proof>

lemma *irreducible-cases*: $\text{irreducible } p \longleftrightarrow$

$\text{degree } p = 0 \wedge \text{irreducible } (\text{coeff } p \ 0) \vee$
 $\text{degree } p \neq 0 \wedge \text{irreducible } (\text{map-poly to-fract } p) \wedge \text{content-ff-ff } p = \text{dff } 1$
 <proof>

lemma *dvd-PM-iff*: $p \text{ dvd } q \longleftrightarrow \text{divides-ff } (\text{content-ff-ff } p) (\text{content-ff-ff } q) \wedge$

$\text{map-poly to-fract } p \text{ dvd } \text{map-poly to-fract } q$
 <proof>

lemma *factorial-monoid-poly*: *factorial-monoid* (*mk-monoid* :: $'a :: \text{ufd poly monoid}$)

<proof>

instance *poly* :: $(\text{ufd}) \text{ ufd}$

<proof>

lemma *primitive-iff-some-content-dvd-1*:

fixes $f :: 'a :: \text{ufd poly}$
shows $\text{primitive } f \longleftrightarrow \text{some-gcd.listgcd } (\text{coeffs } f) \text{ dvd } 1$ (**is** - $\longleftrightarrow ?c \text{ dvd } 1$)
 <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 $M :: int \Rightarrow int$ **where** $M x = x \bmod m$

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*: $M (\text{sum } (\lambda x. M (f x)) A) = M (\text{sum } f A)$

<proof>

definition *inv-M* :: $int \Rightarrow int$ **where**

$inv-M = (\lambda x. \text{if } x + x \leq m \text{ then } x \text{ else } x - m)$

lemma *M-inv-M-id[simp]*: $M (inv-M x) = M x$

<proof>

definition *Mp* :: $int \text{ poly} \Rightarrow int \text{ poly}$ **where** $Mp = \text{map-poly } M$

lemma *Mp-0[simp]*: $Mp 0 = 0$ *<proof>*

lemma *Mp-coeff*: $\text{coeff } (Mp\ f)\ i = M\ (\text{coeff } f\ i)$ *<proof>*

abbreviation *eq-m* :: $\text{int poly} \Rightarrow \text{int poly} \Rightarrow \text{bool}$ (**infixl** =*m* 50) **where**
 $f =_m g \equiv (Mp\ f = Mp\ g)$

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

abbreviation *degree-m* :: $\text{int poly} \Rightarrow \text{nat}$ **where**
 $\text{degree-}m\ f \equiv \text{degree } (Mp\ f)$

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]*: $Mp\ (\text{smult } (M\ a)\ f) = Mp\ (\text{smult } a\ f)$ $Mp\ (\text{smult } a\ (Mp\ f)) = Mp\ (\text{smult } a\ f)$
<proof>

lemma *Mp-Mp[simp]*: $Mp\ (Mp\ f) = Mp\ f$ *<proof>*

lemma *Mp-smult-m-0[simp]*: $Mp\ (\text{smult } m\ f) = 0$
<proof>

definition *dvd m* :: $\text{int poly} \Rightarrow \text{int poly} \Rightarrow \text{bool}$ (**infix** *dvd m* 50) **where**
 $f\ \text{dvd } m\ g = (\exists\ h. g =_m f * h)$

notation *dvd m* (**infix** *dvd m* 50)

lemma *dvd m E*:

assumes *fg*: $f\ \text{dvd } m\ g$

and main: $\bigwedge h. g =_m f * h \implies Mp\ h = h \implies \text{thesis}$

shows *thesis*

<proof>

lemma *Mp-dvd m[simp]*: $Mp\ f\ \text{dvd } m\ g \iff f\ \text{dvd } m\ g$

and *dvd m-Mp[simp]*: $f\ \text{dvd } m\ Mp\ g \iff f\ \text{dvd } m\ g$ *<proof>*

definition *irreducible-m*

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

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

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

lemma *degree-m-le-degree* [intro!]: *degree-m f* ≤ *degree f*
 ⟨proof⟩

lemma *irreducible_{d-m}I*:
assumes *f0*: *degree-m f* > 0
and main: $\bigwedge\ g\ h. Mp\ g = g \implies Mp\ h = h \implies \text{degree}\ g > 0 \implies \text{degree}\ g < \text{degree-}m\ f \implies \text{degree}\ h > 0 \implies \text{degree}\ h < \text{degree-}m\ f \implies f =_m g * h \implies \text{False}$
shows *irreducible_{d-m} f*
 ⟨proof⟩

lemma *irreducible_{d-m}E*:
assumes *irreducible_{d-m} f*
and $\text{degree-}m\ f > 0 \implies (\bigwedge\ g\ h. \text{degree-}m\ g < \text{degree-}m\ f \implies \text{degree-}m\ h < \text{degree-}m\ f \implies \neg f =_m g * h) \implies \text{thesis}$
shows *thesis*
 ⟨proof⟩

lemma *irreducible_{d-m}D*:
assumes *irreducible_{d-m} f*
shows $\text{degree-}m\ f > 0$ **and** $\bigwedge\ g\ h. \text{degree-}m\ g < \text{degree-}m\ f \implies \text{degree-}m\ h < \text{degree-}m\ f \implies \neg f =_m g * h$
 ⟨proof⟩

definition *square-free-m* :: *int poly* ⇒ *bool* **where**
 $\text{square-free-}m\ f = (\neg f =_m 0 \wedge (\forall\ g. \text{degree-}m\ g \neq 0 \longrightarrow \neg (g * g\ \text{dvd}_m\ f)))$

definition *coprime-m* :: *int poly* ⇒ *int poly* ⇒ *bool* **where**
 $\text{coprime-}m\ f\ g = (\forall\ h. h\ \text{dvd}_m\ f \longrightarrow h\ \text{dvd}_m\ g \longrightarrow h\ \text{dvd}_m\ 1)$

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* ⇒ *Mp f* = *Mp g* ⇒ *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*: $n > 0 \implies Mp f = f \implies poly-mod.Mp (m * n) f = f$
 ⟨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*: $coeff f (degree f) \bmod m \neq 0 \implies m > 1 \implies degree-m f = degree f$
 ⟨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*: $f =_m g \implies irreducible-m f \longleftrightarrow irreducible-m g$
 ⟨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 \subseteq {q. irreducible q \wedge monic q}
and Q: set-mset Q \subseteq {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-mset fs \wedge set-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-mset as \implies 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-Mon* f cfs
<proof>

lemma *mod-ident-iff*: m > 0 \implies (x :: int) mod m = x \longleftrightarrow x \in {0 ..< m}
<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  $\implies$   $m > 1 \implies$  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  $\implies$  degree-m f = degree f
  <proof>

lemma monic-Mp: monic f  $\implies$  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:  $m' = m * k \implies k > 0 \implies Mp\ (poly-mod.Mp\ m'\ f)$ 
  =  $Mp\ f$ 
  <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  $f0: Mp\ f \neq 0$ 
  and  $deg: degree-m\ f = degree\ f$ 

```

and $eq: f =_m g * h$
and $p: \text{prime } m$
shows $\text{degree-}m f = \text{degree-}m g + \text{degree-}m h$
 $\langle \text{proof} \rangle$

lemma *monic-smult-add-small*: **assumes** $f = 0 \vee \text{degree } f < \text{degree } g$ **and** *monic* g
shows *monic* $(g + \text{smult } q f)$
 $\langle \text{proof} \rangle$

context *poly-mod*
begin

definition *factorization-m* :: $\text{int poly} \Rightarrow (\text{int} \times \text{int poly multiset}) \Rightarrow \text{bool}$ **where**
factorization-m $f \text{ cfs} \equiv (\text{case } \text{cfs} \text{ of } (c, fs) \Rightarrow f =_m (\text{smult } c (\text{prod-mset } fs)) \wedge$
 $(\forall f \in \text{set-mset } fs. \text{irreducible-}_d\text{-}m f \wedge \text{monic } (Mp f)))$

definition *Mf* :: $\text{int} \times \text{int poly multiset} \Rightarrow \text{int} \times \text{int poly multiset}$ **where**
Mf $\text{cfs} \equiv \text{case } \text{cfs} \text{ of } (c, fs) \Rightarrow (M c, \text{image-mset } Mp fs)$

lemma *Mf-Mf[simp]*: $Mf (Mf x) = Mf x$
 $\langle \text{proof} \rangle$

definition *equivalent-fact-m* :: $\text{int} \times \text{int poly multiset} \Rightarrow \text{int} \times \text{int poly multiset} \Rightarrow \text{bool}$ **where**
equivalent-fact-m $\text{cfs } dgs = (Mf \text{ cfs} = Mf dgs)$

definition *unique-factorization-m* :: $\text{int poly} \Rightarrow (\text{int} \times \text{int poly multiset}) \Rightarrow \text{bool}$
where
unique-factorization-m $f \text{ cfs} = (Mf \text{ 'Collect (factorization-m } f) = \{Mf \text{ cfs}\})$

lemma *Mp-irreducible-d-m[simp]*: $\text{irreducible-}_d\text{-}m (Mp f) = \text{irreducible-}_d\text{-}m f$
 $\langle \text{proof} \rangle$

lemma *Mf-factorization-m[simp]*: $\text{factorization-m } f (Mf \text{ cfs}) = \text{factorization-m } f \text{ cfs}$
 $\langle \text{proof} \rangle$

lemma *unique-factorization-m-imp-factorization*: **assumes** *unique-factorization-m* $f \text{ cfs}$
shows *factorization-m* $f \text{ cfs}$
 $\langle \text{proof} \rangle$

lemma *unique-factorization-m-alt-def*: $\text{unique-factorization-m } f \text{ cfs} = (\text{factorization-m } f \text{ cfs}$
 $\wedge (\forall dgs. \text{factorization-m } f dgs \longrightarrow Mf dgs = Mf \text{ cfs}))$
 $\langle \text{proof} \rangle$

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 \implies Mp f = Mp g*
 \implies unique-factorization-m g cfs
{*proof*}

lemma *unique-factorization-mI*: **assumes** *factorization-m f (c,fs)*
and $\bigwedge d gs. \textit{factorization-m f (d,gs)} \implies \textit{Mf (d,gs)} = \textit{Mf (c,fs)}$
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 \implies lead-coeff (Mp f) = M (lead-coeff f)*
{*proof*}

lemma *unique-factorization-m-zero*: **assumes** *unique-factorization-m f (c,fs)*
shows $M\ c \neq 0$
 \langle *proof* \rangle

end

context *poly-mod*
begin

lemma *dvdm-smult*: **assumes** *f dvd m g*
shows *f dvd m smult c g*
 \langle *proof* \rangle

lemma *dvdm-factor*: **assumes** *f dvd m g*
shows *f dvd m g * h*
 \langle *proof* \rangle

lemma *square-free-m-smultD*: **assumes** *square-free-m (smult c f)*
shows *square-free-m f*
 \langle *proof* \rangle

lemma *square-free-m-smultI*: **assumes** *sf: square-free-m f*
and *inv: M (ci * c) = 1*
shows *square-free-m (smult c f)*
 \langle *proof* \rangle

lemma *square-free-m-factor*: **assumes** *square-free-m (f * g)*
shows *square-free-m f square-free-m g*
 \langle *proof* \rangle

end

context *poly-mod-2*
begin

lemma *Mp-ident-iff*: $Mp\ f = f \iff (\forall\ n.\ \text{coeff}\ f\ n \in \{0 \ ..< m\})$
 \langle *proof* \rangle

lemma *Mp-ident-iff'*: $Mp\ f = f \iff (\text{set}\ (\text{coeffs}\ f) \subseteq \{0 \ ..< m\})$
 \langle *proof* \rangle

end

lemma *Mp-Mp-pow-is-Mp*: $n \neq 0 \implies p > 1 \implies \text{poly-mod.Mp}\ p\ (\text{poly-mod.Mp}\ (p \widehat{\ } n)\ f)$
 $= \text{poly-mod.Mp}\ p\ f$
 \langle *proof* \rangle

lemma *M-M-pow-is-M*: $n \neq 0 \implies p > 1 \implies \text{poly-mod.M } p (\text{poly-mod.M } (p \wedge n) f)$
 $= \text{poly-mod.M } p f \langle \text{proof} \rangle$

definition *inverse-mod* :: $\text{int} \Rightarrow \text{int} \Rightarrow \text{int}$ **where**
inverse-mod $x m = \text{fst } (\text{bezout-coefficients } x m)$

lemma *inverse-mod*:
 $(\text{inverse-mod } x m * x) \text{ mod } m = 1$
if *coprime* $x m m > 1$
 $\langle \text{proof} \rangle$

lemma *inverse-mod-pow*:
 $(\text{inverse-mod } x (p \wedge n) * x) \text{ mod } (p \wedge n) = 1$
if *coprime* $x p p > 1 n \neq 0$
 $\langle \text{proof} \rangle$

lemma (**in** *poly-mod*) *inverse-mod-coprime*:
assumes *p*: *prime* m
and *cop*: *coprime* $x m$ **shows** $M (\text{inverse-mod } x m * x) = 1$
 $\langle \text{proof} \rangle$

lemma (**in** *poly-mod*) *inverse-mod-coprime-exp*:
assumes $m: m = p \wedge n$ **and** *p*: *prime* p
and $n: n \neq 0$ **and** *cop*: *coprime* $x p$
shows $M (\text{inverse-mod } x m * x) = 1$
 $\langle \text{proof} \rangle$

locale *poly-mod-prime* = *poly-mod* p **for** $p :: \text{int} +$
assumes *prime*: *prime* p
begin

sublocale *poly-mod-2* $p \langle \text{proof} \rangle$

lemma *square-free-m-prod-imp-coprime-m*: **assumes** *sf*: *square-free-m* $(A * B)$
shows *coprime-m* $A B$
 $\langle \text{proof} \rangle$

lemma *coprime-exp-mod*: *coprime* $lu p \implies n \neq 0 \implies lu \text{ mod } p \wedge n \neq 0$
 $\langle \text{proof} \rangle$

end

context *poly-mod*
begin

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

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 $\exists q\ r. b = q * a + Polynomial.smult\ m\ r$
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 $q: lead-coeff\ q\ mod\ m \neq 0$ 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 $q0: Mp\ q \neq 0$ and $m1: m > 1$
shows $degree\ p \leq degree\ q$
⟨*proof*⟩

lemma *dvdm-uminus* [*simp*]:
 $p\ dvdm\ -q \longleftrightarrow p\ dvdm\ q$
⟨*proof*⟩

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

lemma *dvdm-imp-div-mod*:
assumes $u \text{ dvdm } g$
shows $\exists q r. g = q * u + \text{smult } m r$
<proof>

corollary *div-mod-iff-dvdm*:
shows $a \text{ dvdm } b = (\exists q r. b = q * a + \text{Polynomial.smult } m r)$
<proof>

lemma *dvdmE'*:
assumes $p \text{ dvdm } q$ **and** $\bigwedge r. q = m p * Mp r \implies \text{thesis}$
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 $g \text{ dvdm } f$
<proof>

lemma *dvdm-degree*: $\text{monic } u \implies u \text{ dvdm } f \implies Mp f \neq 0 \implies \text{degree } u \leq \text{degree } f$
<proof>

end

lemma (**in** *poly-mod-prime*) *pl-dvdm-imp-p-dvdm*:
assumes $l0: l \neq 0$
and *pl-dvdm: poly-mod.dvdm (p^l) a b*
shows $a \text{ dvdm } b$
<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* \Rightarrow *int poly* **where**
to-int-poly \equiv *map-poly to-int-mod-ring*

interpretation *to-int-poly-hom*: *map-poly-inj-zero-hom to-int-mod-ring* <*proof*>

lemma *irreducible_a-def-0*:
fixes *f* :: 'a :: {*comm-semiring-1, semiring-no-zero-divisors*} *poly*
shows *irreducible_a 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* \Rightarrow 'a *mod-ring poly* \Rightarrow *bool*
where *MP-Rel f f'* \equiv (*Mp f* = *to-int-poly f'*)

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

definition *MF-Rel* \equiv *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]*: ($M\text{-Rel} \implies M\text{-Rel} \implies (=)$) ($\lambda x y. M x = M y$) (=)
 ⟨*proof*⟩

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

lemma *eq-MP-Rel[transfer-rule]*: ($MP\text{-Rel} \implies MP\text{-Rel} \implies (=)$) (=m) (=)
 ⟨*proof*⟩

lemma *eq-Mf-Rel[transfer-rule]*: ($MF\text{-Rel} \implies MF\text{-Rel} \implies (=)$) ($\lambda x y. Mf x = Mf y$) (=)
 ⟨*proof*⟩

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

lemma *plus-MP-Rel[transfer-rule]*: ($MP\text{-Rel} \implies MP\text{-Rel} \implies MP\text{-Rel}$) (+)
 (+)
 ⟨*proof*⟩

lemma *times-MP-Rel[transfer-rule]*: ($MP\text{-Rel} \implies MP\text{-Rel} \implies MP\text{-Rel}$)
 ((*)) ((*))
 ⟨*proof*⟩

lemma *smult-MP-Rel[transfer-rule]*: ($M\text{-Rel} \implies MP\text{-Rel} \implies MP\text{-Rel}$) *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* $\implies MP\text{-Rel}$) *prod-list*
prod-list
 ⟨*proof*⟩

lemma *prod-mset-MP-Rel[transfer-rule]*: (*rel-mset MP-Rel* $\implies MP\text{-Rel}$) *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]*: $Domainp MP-Rel = (\lambda f. True)$
<proof>

lemma *mem-MP-Rel[transfer-rule]*: $(MP-Rel ==> rel-set MP-Rel ==> (=))$
 $(\lambda x Y. \exists y \in Y. eq-m x y) (\in)$
<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]: $(MP-Rel ==> MP-Rel ==> (=)) (dvdm)$
 ⟨proof⟩

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

lemma *irreducible_a-MP-Rel* [transfer-rule]: $(MP-Rel ==> (=)) irreducible_a-m irreducible_a$
 ⟨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* \implies (=)) *prime-elem-m*
prime-elem

<proof>

end

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

lemma *non-empty*: $\{0..<m\} \neq \{\}$ *<proof>*

lemma *type-to-set*:

assumes *type-def*: $\exists (Rep :: 'b \Rightarrow int)$ *Abs. type-definition* *Rep Abs* $\{0 ..< m :: int\}$

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

<proof>

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* \implies *MF-Rel* \implies (=)) *factorization-m* (*factorization* *Irr-Mon*)

<proof>

lemma *unique-factorization-MP-Rel* [*transfer-rule*]: (*MP-Rel* \implies *MF-Rel* \implies (=))
(=)

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* \implies *PROP* *P* \implies *PROP* *Q*) \equiv (*PROP* *P* \implies *PROP* *Q*) (**is** ?l \equiv ?r)

<proof>

context *poly-mod-prime* **begin**

lemma *type-to-set*:

assumes *type-def*: $\exists (Rep :: 'b \Rightarrow int) Abs. type-definition\ Rep\ Abs\ \{0 ..< p :: int\}$
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_a-lifting*:
assumes $n: n \neq 0$
and *deg*: *poly-mod.degree-m* ($p \hat{=} n$) *f* = *degree-m f*
and *irr*: *irreducible_a-m f*
shows *poly-mod.irreducible_a-m* ($p \hat{=} n$) *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** $f :: 'a :: \text{comm-ring-1 poly}$
assumes $f = \text{monom-mult } n \ f1 + f0$ **and** $g = \text{monom-mult } n \ g1 + g0$
shows
 $\text{monom-mult } (n + n) (f1 * g1) + (\text{monom-mult } n (f1 * g1 - (f1 - f0) * (g1 - g0) + f0 * g0) + f0 * g0) = f * g$
 ⟨proof⟩

lemma karatsuba-single-sided: **fixes** $f :: 'a :: \text{comm-ring-1 poly}$
assumes $f = \text{monom-mult } n \ f1 + f0$
shows $\text{monom-mult } n (f1 * g) + f0 * g = f * g$
 ⟨proof⟩

definition split-at :: $\text{nat} \Rightarrow 'a \ \text{list} \Rightarrow 'a \ \text{list} \times 'a \ \text{list}$ **where**
 [code del]: $\text{split-at } n \ xs = (\text{take } n \ xs, \text{drop } n \ xs)$

lemma split-at-code[code]:
 $\text{split-at } n \ [] = ([], [])$
 $\text{split-at } n (x \# xs) = (\text{if } n = 0 \text{ then } ([], x \# xs) \text{ else case split-at } (n-1) \ xs \text{ of}$
 (bef, aft)
 $\Rightarrow (x \# bef, aft))$
 ⟨proof⟩

fun coeffs-minus :: $'a :: \text{ab-group-add list} \Rightarrow 'a \ \text{list} \Rightarrow 'a \ \text{list}$ **where**
 $\text{coeffs-minus } (x \# xs) (y \# ys) = ((x - y) \# \text{coeffs-minus } xs \ ys)$
 | $\text{coeffs-minus } xs \ [] = xs$
 | $\text{coeffs-minus } [] \ ys = \text{map } \text{uminus } ys$

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 \Rightarrow nat \Rightarrow 'a list \Rightarrow nat \Rightarrow 'a poly **where**

karatsuba-main f n g m = (if n \leq *karatsuba-lower-bound* \vee m \leq *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) \Rightarrow case split-at n2 g of
(g0,g1) \Rightarrow 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) \Rightarrow 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 \Rightarrow 'a poly \Rightarrow 'a poly **where**
karatsuba-mult-poly f g = (let ff = coeffs f; gg = coeffs g; n = length ff; m = length gg
in (if n \leq *karatsuba-lower-bound* \vee m \leq *karatsuba-lower-bound* then if n \leq m
then foldr (λ a p. smult a g + pCons 0 p) ff 0
else foldr (λ a p. smult a f + pCons 0 p) gg 0
else if n \leq 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::?'a) ?n * ?f = monom-mult ?n ?f*
*?f * monom (1::?'a) ?n = monom-mult ?n ?f.*

lemmas *karatsuba-monom-mult-code-unfold*[*code-unfold*] =
monom-mult-unfold[**where** *f = f :: 'a :: comm-ring-1 poly* **for** *f*, *unfolded karatsuba-mult-poly-code-unfold*]

end

5.5 Record Based Version

We provide an implementation for polynomials which may be parametrized by the ring- or field-operations. These don't have to be type-based!

5.5.1 Definitions

theory *Polynomial-Record-Based*

imports

Arithmetic-Record-Based

Karatsuba-Multiplication

begin

context

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

begin

private abbreviation (*input*) *zero* **where** *zero* \equiv *arith-ops-record.zero ops*

private abbreviation (*input*) *one* **where** *one* \equiv *arith-ops-record.one ops*

private abbreviation (*input*) *plus* **where** *plus* \equiv *arith-ops-record.plus ops*

private abbreviation (*input*) *times* **where** *times* \equiv *arith-ops-record.times ops*

private abbreviation (*input*) *minus* **where** *minus* \equiv *arith-ops-record.minus ops*

private abbreviation (*input*) *uminus* **where** *uminus* \equiv *arith-ops-record.uminus ops*

private abbreviation (*input*) *divide* **where** *divide* \equiv *arith-ops-record.divide ops*

private abbreviation (*input*) *inverse* **where** *inverse* \equiv *arith-ops-record.inverse ops*

private abbreviation (*input*) *modulo* **where** *modulo* \equiv *arith-ops-record.modulo ops*

private abbreviation (*input*) *normalize* **where** *normalize* \equiv *arith-ops-record.normalize ops*

private abbreviation (*input*) *unit-factor* **where** *unit-factor* \equiv *arith-ops-record.unit-factor ops*

private abbreviation (*input*) *DP* **where** *DP* \equiv *arith-ops-record.DP ops*

definition *is-poly* :: *'i list* \Rightarrow *bool* **where**

$is-poly\ xs \iff list-all\ DP\ xs \wedge no-trailing\ (HOL.eq\ zero)\ xs$

definition $cCons-i :: 'i \Rightarrow 'i\ list \Rightarrow 'i\ list$

where

$cCons-i\ x\ xs = (if\ xs = [] \wedge x = zero\ then\ []\ else\ x \# xs)$

fun $plus-poly-i :: 'i\ list \Rightarrow 'i\ list \Rightarrow 'i\ list$ **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 $uminus-poly-i :: 'i\ list \Rightarrow 'i\ list$ **where**

$[code-unfold]:\ uminus-poly-i = map\ uminus$

fun $minus-poly-i :: 'i\ list \Rightarrow 'i\ list \Rightarrow 'i\ list$ **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**

$zero-poly-i \equiv []$

definition $one-poly-i :: 'i\ list$ **where**

$[code-unfold]:\ one-poly-i = [one]$

definition $smult-i :: 'i \Rightarrow 'i\ list \Rightarrow 'i\ list$ **where**

$smult-i\ a\ pp = (if\ a = zero\ then\ []\ else\ strip-while\ ((=)\ zero)\ (map\ (times\ a)\ pp))$

definition $sdiv-i :: 'i\ list \Rightarrow 'i \Rightarrow 'i\ list$ **where**

$sdiv-i\ pp\ a = (strip-while\ ((=)\ zero)\ (map\ (\lambda\ c.\ divide\ c\ a)\ pp))$

definition $poly-of-list-i :: 'i\ list \Rightarrow 'i\ list$ **where**

$poly-of-list-i = strip-while\ ((=)\ zero)$

fun $coeffs-minus-i :: 'i\ list \Rightarrow 'i\ list \Rightarrow 'i\ list$ **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 $monom-mult-i :: nat \Rightarrow 'i\ list \Rightarrow 'i\ list$ **where**

$monom-mult-i\ n\ xs = (if\ xs = []\ then\ xs\ else\ replicate\ n\ zero\ @\ xs)$

fun $karatsuba-main-i :: 'i\ list \Rightarrow nat \Rightarrow 'i\ list \Rightarrow nat \Rightarrow 'i\ list$ **where**

$karatsuba-main-i\ f\ n\ g\ m = (if\ n \leq karatsuba-lower-bound \vee m \leq karatsuba-lower-bound$
 $then$

$let\ ff = poly-of-list-i\ f\ in\ foldr\ (\lambda a\ p.\ plus-poly-i\ (smult-i\ a\ ff)\ (cCons-i\ zero\ p))$

$g\ zero-poly-i$

$else\ let\ n2 = n\ div\ 2\ in$

if $m > n2$ then (case split-at $n2$ f of
 ($f0, f1$) \Rightarrow case split-at $n2$ g of
 ($g0, g1$) \Rightarrow let
 $p1 = \text{karatsuba-main-}i\ f1\ (n - n2)\ g1\ (m - n2)$;
 $p2 = \text{karatsuba-main-}i\ (\text{coeffs-minus-}i\ f1\ f0)\ n2\ (\text{coeffs-minus-}i\ g1\ g0)\ n2$;
 $p3 = \text{karatsuba-main-}i\ f0\ n2\ g0\ n2$
 in $\text{plus-poly-}i\ (\text{monom-mult-}i\ (n2 + n2)\ p1)$
 ($\text{plus-poly-}i\ (\text{monom-mult-}i\ n2\ (\text{plus-poly-}i\ (\text{minus-poly-}i\ p1\ p2)\ p3))\ p3$)
 else case split-at $n2$ f of
 ($f0, f1$) \Rightarrow let
 $p1 = \text{karatsuba-main-}i\ f1\ (n - n2)\ g\ m$;
 $p2 = \text{karatsuba-main-}i\ f0\ n2\ g\ m$
 in $\text{plus-poly-}i\ (\text{monom-mult-}i\ n2\ p1)\ p2$)

definition $\text{times-poly-}i :: 'i\ \text{list} \Rightarrow 'i\ \text{list} \Rightarrow 'i\ \text{list}$ **where**

$\text{times-poly-}i\ f\ g \equiv (\text{let } n = \text{length } f; m = \text{length } g$
 in (if $n \leq \text{karatsuba-lower-bound} \vee m \leq \text{karatsuba-lower-bound}$ then if $n \leq m$
 then

$\text{foldr } (\lambda a\ p.\ \text{plus-poly-}i\ (\text{smult-}i\ a\ g)\ (c\text{Cons-}i\ \text{zero } p))\ f\ \text{zero-poly-}i$ else
 $\text{foldr } (\lambda a\ p.\ \text{plus-poly-}i\ (\text{smult-}i\ a\ f)\ (c\text{Cons-}i\ \text{zero } p))\ g\ \text{zero-poly-}i$ else
 if $n \leq m$ then $\text{karatsuba-main-}i\ g\ m\ f\ n$ else $\text{karatsuba-main-}i\ f\ n\ g\ m$)

definition $\text{coeff-}i :: 'i\ \text{list} \Rightarrow \text{nat} \Rightarrow 'i$ **where**

$\text{coeff-}i = \text{nth-default zero}$

definition $\text{degree-}i :: 'i\ \text{list} \Rightarrow \text{nat}$ **where**

$\text{degree-}i\ pp \equiv \text{length } pp - 1$

definition $\text{lead-coeff-}i :: 'i\ \text{list} \Rightarrow 'i$ **where**

$\text{lead-coeff-}i\ pp = (\text{case } pp\ \text{of } [] \Rightarrow \text{zero} \mid - \Rightarrow \text{last } pp)$

definition $\text{monic-}i :: 'i\ \text{list} \Rightarrow \text{bool}$ **where**

$\text{monic-}i\ pp = (\text{lead-coeff-}i\ pp = \text{one})$

fun $\text{minus-poly-rev-list-}i :: 'i\ \text{list} \Rightarrow 'i\ \text{list} \Rightarrow 'i\ \text{list}$ **where**

$\text{minus-poly-rev-list-}i\ (x \# xs)\ (y \# ys) = (\text{minus } x\ y) \# (\text{minus-poly-rev-list-}i\ xs$
 $ys)$

$\mid \text{minus-poly-rev-list-}i\ xs\ [] = xs$
 $\mid \text{minus-poly-rev-list-}i\ []\ (y \# ys) = []$

fun $\text{divmod-poly-one-main-}i :: 'i\ \text{list} \Rightarrow 'i\ \text{list} \Rightarrow 'i\ \text{list}$

$\Rightarrow \text{nat} \Rightarrow 'i\ \text{list} \times 'i\ \text{list}$ **where**

$\text{divmod-poly-one-main-}i\ q\ r\ d\ (\text{Suc } n) = (\text{let}$

$a = \text{hd } r$;

$qqq = c\text{Cons-}i\ a\ q$;

$rr = \text{tl } (if\ a = \text{zero}\ \text{then } r\ \text{else } \text{minus-poly-rev-list-}i\ r\ (\text{map } (\text{times } a)\ d))$

in $\text{divmod-poly-one-main-}i\ qqq\ rr\ d\ n$)

$\mid \text{divmod-poly-one-main-}i\ q\ r\ d\ 0 = (q, r)$

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 \equiv *gcd-poly-i xs* (*pderiv-i xs*) = *one-poly-i*

end

5.5.2 Properties

definition *pdivmod-monic* :: 'a::comm-ring-1 poly \Rightarrow 'a poly \Rightarrow 'a poly \times 'a poly
where

pdivmod-monic f g \equiv *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* ($\lambda x. x \text{ div } a$)
(coeffs p)))
{*proof*}

lifting-forget *poly.lifting*

context *ring-ops*

begin

definition *poly-rel* :: 'i list \Rightarrow 'a poly \Rightarrow bool **where**
poly-rel x x' \iff *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* \implies *poly-rel x z* \implies *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]*: ($R \implies \text{list-all2 } R \implies \text{list-all2 } R$)
(*cCons-i ops*) *cCons*
<proof>

lemma *poly-rel-pCons[transfer-rule]*: ($R \implies \text{poly-rel} \implies \text{poly-rel}$) (*cCons-i ops*) *pCons*
<proof>

lemma *poly-rel-eq[transfer-rule]*: ($\text{poly-rel} \implies \text{poly-rel} \implies (=)$) (=) (=)
<proof>

lemma *poly-rel-plus[transfer-rule]*: ($\text{poly-rel} \implies \text{poly-rel} \implies \text{poly-rel}$) (*plus-poly-i ops*) (+)
<proof>

lemma *poly-rel-uminus[transfer-rule]*: ($\text{poly-rel} \implies \text{poly-rel}$) (*uminus-poly-i ops*)
Groups.uminus
<proof>

lemma *poly-rel-minus[transfer-rule]*: ($\text{poly-rel} \implies \text{poly-rel} \implies \text{poly-rel}$) (*minus-poly-i ops*) (-)
<proof>

lemma *poly-rel-smult[transfer-rule]*: ($R \implies \text{poly-rel} \implies \text{poly-rel}$) (*smult-i ops*) *smult*
<proof>

lemma *poly-rel-coeffs[transfer-rule]*: ($\text{poly-rel} \implies \text{list-all2 } R$) ($\lambda x. x$) *coeffs*
<proof>

lemma *poly-rel-poly-of-list[transfer-rule]*: ($\text{list-all2 } R \implies \text{poly-rel}$) (*poly-of-list-i ops*) *poly-of-list*
<proof>

lemma *poly-rel-monom-mult[transfer-rule]*:
($(=) \implies \text{poly-rel} \implies \text{poly-rel}$) (*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 \implies list-all2 R y1 y2 \implies*
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 \implies \implies \implies \implies \implies \implies poly-rel*) (*times-poly-i*
ops) ((***))
{*proof*}

lemma *poly-rel-coeff[transfer-rule]*: (*poly-rel \implies \implies \implies \implies \implies \implies R*) (*coeff-i ops*)
coeff
{*proof*}

lemma *poly-rel-degree[transfer-rule]*: (*poly-rel \implies \implies \implies \implies \implies \implies 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 \implies \implies \implies \implies \implies R*) (*lead-coeff-i ops*) *lead-coeff*
{*proof*}

lemma *poly-rel-minus-poly-rev-list[transfer-rule]*:
(*list-all2 R \implies \implies \implies \implies \implies \implies list-all2 R*) (*minus-poly-rev-list-i ops*) *mi-*
nus-poly-rev-list
{*proof*}

lemma *divmod-poly-one-main-i*: **assumes** *len: n \leq 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 \leq length X* **and** *rel: list-all2 R x X*
list-all2 R y Y

and $n: n = N$
shows $list-all2\ R\ (mod-poly-one-main-i\ ops\ x\ y\ n)$
 $(mod-poly-one-main-list\ X\ Y\ N)$
 $\langle proof \rangle$

lemma $poly-rel-dvd[transfer-rule]: (poly-rel\ ==\>\ poly-rel\ ==\>\ (=))\ (dvd-poly-i\ ops)\ (dvd)$
 $\langle proof \rangle$

lemma $poly-rel-monic[transfer-rule]: (poly-rel\ ==\>\ (=))\ (monic-i\ ops)\ monic$
 $\langle proof \rangle$

lemma $poly-rel-pdivmod-monic: assumes\ mon: monic\ Y$
and $x: poly-rel\ x\ X$ **and** $y: poly-rel\ y\ Y$
shows $rel-prod\ poly-rel\ poly-rel\ (pdivmod-monic-i\ ops\ x\ y)\ (pdivmod-monic\ X\ Y)$
 $\langle proof \rangle$

lemma $ring-ops-poly: ring-ops\ (poly-ops\ ops)\ poly-rel$
 $\langle proof \rangle$
end

context $idom-ops$
begin

lemma $poly-rel-pderiv [transfer-rule]: (poly-rel\ ==\>\ poly-rel)\ (pderiv-i\ ops)\ pderiv$
 $\langle proof \rangle$

lemma $poly-rel-irreducible[transfer-rule]: (poly-rel\ ==\>\ (=))\ (irreducible-i\ ops)\ irreducible_a$
 $\langle proof \rangle$

lemma $idom-ops-poly: idom-ops\ (poly-ops\ ops)\ poly-rel$
 $\langle proof \rangle$
end

context $idom-divide-ops$
begin

lemma $poly-rel-sdiv[transfer-rule]: (poly-rel\ ==\>\ R\ ==\>\ poly-rel)\ (sdiv-i\ ops)\ sdiv-poly$
 $\langle proof \rangle$
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* \implies *poly-rel* \implies *poly-rel*)
(*mod-field-poly-i ops*) (*mod*)
<proof>

lemma *poly-rel-normalize* [*transfer-rule*]: (*poly-rel* \implies *poly-rel*)
(*normalize-poly-i ops*) *Rings.normalize*
<proof>

lemma *poly-rel-unit-factor* [*transfer-rule*]: (*poly-rel* \implies *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* \implies *poly-rel* \implies *poly-rel*) (*gcd-poly-i*
ops) *gcd*
<proof>

lemma *poly-rel-euclid-ext* [*transfer-rule*]: (*poly-rel* \implies *poly-rel* \implies *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*: $f' = \text{of-int-poly-i ff-ops (Mp f)}$ $f'' = \text{of-int-poly (Mp f)}$
shows *poly-rel* $f' f''$ $\langle \text{proof} \rangle$

sublocale *arith-ops-record ff-ops p* $\langle \text{proof} \rangle$

lemma *Mp-rel-iI*: *poly-rel* $f1 f2 \implies \text{MP-Rel } f3 f2 \implies \text{Mp-rel-i } f1 f3$
 $\langle \text{proof} \rangle$

lemma *M-rel-iI*: *R* $f1 f2 \implies \text{M-Rel } f3 f2 \implies \text{M-rel-i } f1 f3$
 $\langle \text{proof} \rangle$

lemma *M-rel-iI'*: **assumes** *R* $f1 f2$
shows *M-rel-i* $f1 (\text{arith-ops-record.to-int ff-ops } f1)$
 $\langle \text{proof} \rangle$

lemma *Mp-rel-iI'*: **assumes** *poly-rel* $f1 f2$
shows *Mp-rel-i* $f1 (\text{to-int-poly-i ff-ops } f1)$
 $\langle \text{proof} \rangle$

lemma *M-rel-iD*: **assumes** *M-rel-i* $f1 f3$
shows
R $f1 (\text{of-int (M } f3))$
M-Rel $f3 (\text{of-int (M } f3))$
 $\langle \text{proof} \rangle$

lemma *Mp-rel-iD*: **assumes** *Mp-rel-i* $f1 f3$
shows
poly-rel $f1 (\text{of-int-poly (Mp } f3))$
MP-Rel $f3 (\text{of-int-poly (Mp } f3))$
 $\langle \text{proof} \rangle$
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* \implies *'a* :: *prime-card mod-ring* \implies *bool* **and** *p* :: *int*
begin

sublocale *poly-mod-prime-type p TYPE('a)*
 $\langle \text{proof} \rangle$

end

lemma (**in** *mod-ring-locale*) *mod-ring-rel-of-int*:
 $0 \leq x \implies x < p \implies \text{mod-ring-rel } x (\text{of-int } x)$
 $\langle \text{proof} \rangle$

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 ≤ 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 ≤ 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 ≤ 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

begin

lemma *cong-add-poly*:

$[(a::'b::\{\text{field-gcd}\} \text{poly}) = b] \pmod{m} \implies [c = d] \pmod{m} \implies [a + c = b + d] \pmod{m}$
 ⟨proof⟩

lemma *cong-mult-poly*:

$[(a::'b::\{\text{field-gcd}\} \text{poly}) = b] \pmod{m} \implies [c = d] \pmod{m} \implies [a * c = b * d] \pmod{m}$
 ⟨proof⟩

lemma *cong-mult-self-poly*: $[(a::'b::\{\text{field-gcd}\} \text{poly}) * m = 0] \pmod{m}$

⟨proof⟩

lemma *cong-scalar2-poly*: $[(a::'b::\{\text{field-gcd}\} \text{poly}) = b] \pmod{m} \implies [k * a = k * b] \pmod{m}$

⟨proof⟩

lemma *cong-sum-poly*:

$(\bigwedge x. x \in A \implies [(f x)::'b::\{\text{field-gcd}\} \text{poly}) = g x] \pmod{m}) \implies$
 $[(\sum_{x \in A} f x) = (\sum_{x \in A} g x)] \pmod{m}$
 ⟨proof⟩

lemma *cong-iff-lin-poly*: $[(a::'b::\{\text{field-gcd}\} \text{poly}) = b] \pmod{m} = (\exists k. b = a + m * k)$

⟨proof⟩

lemma *cong-solve-poly*: $(a::'b::\{\text{field-gcd}\} \text{poly}) \neq 0 \implies \exists x. [a * x = \text{gcd } a \ n] \pmod{n}$

⟨proof⟩

lemma *cong-solve-coprime-poly*:

assumes *coprime-an*: *coprime* $(a::'b::\{\text{field-gcd}\} \text{poly}) \ n$

shows $\exists x. [a * x = 1] \pmod{n}$

⟨proof⟩

lemma *cong-dvd-modulus-poly*:

$[x = y] \pmod{m} \implies n \text{ dvd } m \implies [x = y] \pmod{n}$ **for** $x \ y :: 'b::\{\text{field-gcd}\} \text{poly}$
 ⟨proof⟩

lemma *chinese-remainder-aux-poly*:

fixes $A :: 'a \text{ set}$

and $m :: 'a \Rightarrow 'b::\{\text{field-gcd}\} \text{poly}$

assumes *fin*: *finite* A

and *cop*: $\forall i \in A. (\forall j \in A. i \neq j \longrightarrow \text{coprime } (m \ i) \ (m \ j))$

shows $\exists b. (\forall i \in A. [b \ i = 1] \ (\text{mod } m \ i) \wedge [b \ i = 0] \ (\text{mod } (\prod_{j \in A - \{i\}} m \ j)))$
 $\langle \text{proof} \rangle$

lemma *chinese-remainder-poly*:

fixes $A :: 'a \ \text{set}$
and $m :: 'a \Rightarrow 'b :: \{\text{field-gcd}\} \ \text{poly}$
and $u :: 'a \Rightarrow 'b \ \text{poly}$
assumes $\text{fin: finite } A$
and $\text{cop: } \forall i \in A. (\forall j \in A. i \neq j \longrightarrow \text{coprime } (m \ i) \ (m \ j))$
shows $\exists x. (\forall i \in A. [x = u \ i] \ (\text{mod } m \ i))$
 $\langle \text{proof} \rangle$

lemma *cong-trans-poly*:

$[(a :: 'b :: \{\text{field-gcd}\} \ \text{poly}) = b] \ (\text{mod } m) \Longrightarrow [b = c] \ (\text{mod } m) \Longrightarrow [a = c] \ (\text{mod } m)$
 $\langle \text{proof} \rangle$

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

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

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

lemma *coprime-cong-mult-poly*:

assumes $[(a :: 'b :: \{\text{field-gcd}\} \ \text{poly}) = b] \ (\text{mod } m)$ **and** $[a = b] \ (\text{mod } n)$ **and** $\text{coprime } m \ n$
shows $[a = b] \ (\text{mod } m * n)$
 $\langle \text{proof} \rangle$

lemma *coprime-cong-prod-poly*:

$(\forall i \in A. (\forall j \in A. i \neq j \longrightarrow \text{coprime } (m \ i) \ (m \ j))) \Longrightarrow$
 $(\forall i \in A. [(x :: 'b :: \{\text{field-gcd}\} \ \text{poly}) = y] \ (\text{mod } m \ i)) \Longrightarrow$
 $[x = y] \ (\text{mod } (\prod_{i \in A} m \ i))$
 $\langle \text{proof} \rangle$

lemma *cong-less-modulus-unique-poly*:

$[(x :: 'b :: \{\text{field-gcd}\} \ \text{poly}) = y] \ (\text{mod } m) \Longrightarrow \text{degree } x < \text{degree } m \Longrightarrow \text{degree } y < \text{degree } m \Longrightarrow x = y$
 $\langle \text{proof} \rangle$

```

lemma chinese-remainder-unique-poly:
  fixes A :: 'a set
    and m :: 'a ⇒ 'b::{field-gcd} poly
    and u :: 'a ⇒ 'b poly
  assumes nz: ∀ i∈A. (m i) ≠ 0
    and cop: ∀ i∈A. (∀ j∈A. i ≠ j → coprime (m i) (m j))

    and not-constant: 0 < degree (prod m A)
  shows ∃!x. degree x < (∑ i∈A. degree (m i)) ∧ (∀ i∈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 ⇒ 'a :: comm-monoid-mult
begin
lemma prod-list-map-filter: prod-list (map g (filter f xs)) * prod-list (map g (filter
  (λ x. ¬ f x) xs))
  = prod-list (map g xs)
  ⟨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 $a::'b::\text{semiring-gcd}$
shows $\text{coprime } a \ a \longleftrightarrow \text{is-unit } a$
 $\langle \text{proof} \rangle$

lemma $\text{dim-vec-of-list}[\text{simp}]$: $\text{dim-vec } (\text{vec-of-list } x) = \text{length } x$
 $\langle \text{proof} \rangle$

lemma $\text{length-list-of-vec}[\text{simp}]$: $\text{length } (\text{list-of-vec } A) = \text{dim-vec } A$
 $\langle \text{proof} \rangle$

lemma $\text{list-of-vec-vec-of-list}[\text{simp}]$: $\text{list-of-vec } (\text{vec-of-list } a) = a$
 $\langle \text{proof} \rangle$

context
assumes $\text{SORT-CONSTRAINT}('a::\text{finite})$
begin

lemma $\text{inj-Poly-list-of-vec}'$: $\text{inj-on } (\text{Poly } \circ \text{list-of-vec}) \{v. \text{dim-vec } v = n\}$
 $\langle \text{proof} \rangle$

corollary $\text{inj-Poly-list-of-vec}$: $\text{inj-on } (\text{Poly } \circ \text{list-of-vec}) (\text{carrier-vec } n)$
 $\langle \text{proof} \rangle$

lemma $\text{list-of-vec-rw-map}$: $\text{list-of-vec } m = \text{map } (\lambda n. m \ \$ \ n) [0..<\text{dim-vec } m]$
 $\langle \text{proof} \rangle$

lemma $\text{degree-Poly}'$:
assumes $xs: xs \neq []$
shows $\text{degree } (\text{Poly } xs) < \text{length } xs$
 $\langle \text{proof} \rangle$

lemma $\text{vec-of-list-list-of-vec}[\text{simp}]$: $\text{vec-of-list } (\text{list-of-vec } a) = a$
 $\langle \text{proof} \rangle$

lemma $\text{row-mat-of-rows-list}$:
assumes $b: b < \text{length } A$
and $nc: \forall i. i < \text{length } A \longrightarrow \text{length } (A ! i) = nc$
shows $(\text{row } (\text{mat-of-rows-list } nc \ A) \ b) = \text{vec-of-list } (A ! b)$
 $\langle \text{proof} \rangle$

lemma $\text{degree-Poly-list-of-vec}$:
assumes $n: x \in \text{carrier-vec } n$
and $n0: n > 0$
shows $\text{degree } (\text{Poly } (\text{list-of-vec } x)) < n$
 $\langle \text{proof} \rangle$

lemma list-of-vec-nth :
assumes $i: i < \text{dim-vec } x$

shows $list\text{-of-vec } x ! i = x \$ i$
<proof>

lemma *coeff-Poly-list-of-vec-nth'*:
assumes $i < dim\text{-vec } x$
shows $coeff (Poly (list\text{-of-vec } x)) i = x \$ i$
<proof>

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

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

lemma *inj-on-list-of-vec*: $inj\text{-on } list\text{-of-vec (carrier\text{-vec } n)$
<proof>

lemma *vec-of-list-carrier[simp]*: $vec\text{-of-list } x \in carrier\text{-vec (length } x)$
<proof>

lemma *card-carrier-vec*: $card (carrier\text{-vec } n :: 'b::finite\text{ vec set}) = CARD('b) ^ n$
<proof>

lemma *finite-carrier-vec[simp]*: $finite (carrier\text{-vec } n :: 'b::finite\text{ vec set})$
<proof>

lemma *row-echelon-form-dim0-row*:
assumes $A \in carrier\text{-mat } 0\ n$
shows $row\text{-echelon-form } A$
<proof>

lemma *row-echelon-form-dim0-col*:
assumes $A \in carrier\text{-mat } n\ 0$
shows $row\text{-echelon-form } A$
<proof>

lemma *row-echelon-form-one-dim0[simp]*: $row\text{-echelon-form } (1_m\ 0)$
<proof>

lemma *Poly-list-of-vec-0[simp]*: $Poly (list\text{-of-vec } (0_v\ 0)) = [:0:]$
<proof>

lemma *monic-normalize*:

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

lemma *exists-factorization-prod-list:*
fixes $P :: 'b :: \text{field poly list}$
assumes $\text{degree (prod-list } P) > 0$
and $\bigwedge u. u \in \text{set } P \implies \text{degree } u > 0 \wedge \text{monic } u$
and *square-free (prod-list P)*
shows $\exists Q. \text{prod-list } Q = \text{prod-list } P \wedge \text{length } P \leq \text{length } Q$
 $\wedge (\forall u. u \in \text{set } Q \longrightarrow \text{irreducible } u \wedge \text{monic } u)$
 ⟨proof⟩

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

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

lemma *prod-list-dvd-prod-list-subset:*
fixes $A :: 'b :: \text{comm-monoid-mult list}$
assumes $dA: \text{distinct } A$
and $dB: \text{distinct } B$
and $s: \text{set } A \subseteq \text{set } B$
shows $\text{prod-list } A \text{ dvd prod-list } B$
 ⟨proof⟩

end

lemma *gcd-monic-constant:*
 $\text{gcd } f g \in \{1, f\}$ **if** *monic f* **and** $\text{degree } g = 0$
for $f g :: 'a :: \{\text{field-gcd}\} \text{poly}$
 ⟨proof⟩

lemma *distinct-find-base-vectors:*
fixes $A :: 'a :: \text{field mat}$
assumes *ref: row-echelon-form A*
and $A: A \in \text{carrier-mat nr nc}$
shows *distinct (find-base-vectors A)*
 ⟨proof⟩

lemma *length-find-base-vectors:*

fixes $A::'a::\text{field mat}$
assumes $\text{ref}: \text{row-echelon-form } A$
and $A: A \in \text{carrier-mat nr nc}$
shows $\text{length } (\text{find-base-vectors } A) = \text{card } (\text{set } (\text{find-base-vectors } A))$
 $\langle \text{proof} \rangle$

6.2 Previous Results

definition $\text{power-poly-f-mod} :: 'a::\text{field poly} \Rightarrow 'a \text{ poly} \Rightarrow \text{nat} \Rightarrow 'a \text{ poly}$ **where**
 $\text{power-poly-f-mod modulus} = (\lambda a n. a \wedge n \text{ mod modulus})$

lemma $\text{power-poly-f-mod-binary}: \text{power-poly-f-mod } m \ a \ n = (\text{if } n = 0 \text{ then } 1 \text{ mod } m$
 m
 $\text{else let } (d, r) = \text{Divides.divmod-nat } n \ 2;$
 $\text{rec} = \text{power-poly-f-mod } m \ ((a * a) \text{ mod } m) \ d \ \text{in}$
 $\text{if } r = 0 \text{ then } \text{rec} \ \text{else } (\text{rec} * a) \text{ mod } m)$
for $m \ a :: 'a :: \{\text{field-gcd}\} \text{ poly}$
 $\langle \text{proof} \rangle$

fun power-polys **where**
 $\text{power-polys mul-p } u \ \text{curr-p } (\text{Suc } i) = \text{curr-p } \#$
 $\text{power-polys mul-p } u \ ((\text{curr-p} * \text{mul-p}) \text{ mod } u) \ i$
 $| \text{power-polys mul-p } u \ \text{curr-p } 0 = []$

context
assumes $\text{SORT-CONSTRAINT}('a::\text{prime-card})$
begin

lemma $\text{fermat-theorem-mod-ring}$ $[\text{simp}]$:
fixes $a::'a \text{ mod-ring}$
shows $a \wedge \text{CARD}('a) = a$
 $\langle \text{proof} \rangle$

lemma $\text{mod-eq-dvd-iff-poly}: ((x::'a \text{ mod-ring poly}) \text{ mod } n = y \text{ mod } n) = (n \text{ dvd } x$
 $- y)$
 $\langle \text{proof} \rangle$

lemma cong-gcd-eq-poly :
 $\text{gcd } a \ m = \text{gcd } b \ m \ \text{if } [(a::'a \text{ mod-ring poly}) = b] \ (\text{mod } m)$
 $\langle \text{proof} \rangle$

lemma coprime-h-c-poly :
fixes $h::'a \text{ mod-ring poly}$
assumes $c1 \neq c2$
shows $\text{coprime } (h - [:c1:]) \ (h - [:c2:])$
 $\langle \text{proof} \rangle$

lemma *coprime-h-c-poly2*:
fixes $h::'a \text{ mod-ring poly}$
assumes $\text{coprime } (h - [:c1:]) (h - [:c2:])$
and $\neg \text{is-unit } (h - [:c1:])$
shows $c1 \neq c2$
 $\langle \text{proof} \rangle$

lemma *degree-minus-eq-right*:
fixes $p::'b::\text{ab-group-add poly}$
shows $\text{degree } q < \text{degree } p \implies \text{degree } (p - q) = \text{degree } p$
 $\langle \text{proof} \rangle$

lemma *coprime-prod*:
fixes $A::'a \text{ mod-ring set}$ **and** $g::'a \text{ mod-ring} \implies 'a \text{ mod-ring poly}$
assumes $\forall x \in A. \text{coprime } (g \ a) (g \ x)$
shows $\text{coprime } (g \ a) (\text{prod } (\lambda x. g \ x) \ A)$
 $\langle \text{proof} \rangle$

lemma *coprime-prod2*:
fixes $A::'b::\text{semiring-gcd set}$
assumes $\forall x \in A. \text{coprime } (a) (x)$ **and** $f: \text{finite } A$
shows $\text{coprime } (a) (\text{prod } (\lambda x. x) \ A)$
 $\langle \text{proof} \rangle$

lemma *divides-prod*:
fixes $g::'a \text{ mod-ring} \implies 'a \text{ mod-ring poly}$
assumes $\forall c1 \ c2. c1 \in A \wedge c2 \in A \wedge c1 \neq c2 \longrightarrow \text{coprime } (g \ c1) (g \ c2)$
assumes $\forall c \in A. g \ c \ \text{dvd } f$
shows $(\prod c \in A. g \ c) \ \text{dvd } f$
 $\langle \text{proof} \rangle$

lemma *poly-monom-identity-mod-p*:
 $\text{monom } (1::'a \text{ mod-ring}) (\text{CARD}('a)) - \text{monom } 1 \ 1 = \text{prod } (\lambda x. [:0,1:] - [:x:])$
 $(\text{UNIV}::'a \text{ mod-ring set})$
(is ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

lemma *poly-identity-mod-p*:

$v \widehat{(\text{CARD}('a))} - v = \text{prod } (\lambda x. v - [x:])$ (*UNIV::'a mod-ring set*)
 ⟨proof⟩

lemma *coprime-gcd*:

fixes $h::'a \text{ mod-ring poly}$
assumes $\text{Rings.coprime } (h-[c1:]) (h-[c2:])$
shows $\text{Rings.coprime } (\text{gcd } f(h-[c1:])) (\text{gcd } f(h-[c2:]))$
 ⟨proof⟩

lemma *divides-prod-gcd*:

fixes $h::'a \text{ mod-ring poly}$
assumes $\forall c1 c2. c1 \in A \wedge c2 \in A \wedge c1 \neq c2 \longrightarrow \text{coprime } (h-[c1:]) (h-[c2:])$
shows $(\prod c \in A. \text{gcd } f(h - [c:])) \text{ dvd } f$
 ⟨proof⟩

lemma *monic-prod-gcd*:

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

lemma *coprime-not-unit-not-dvd*:

fixes $a::'b::\text{semiring-gcd}$
assumes $a \text{ dvd } b$
and $\text{coprime } b c$
and $\neg \text{is-unit } a$
shows $\neg a \text{ dvd } c$
 ⟨proof⟩

lemma *divides-prod2*:

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

lemma *coprime-polynomial-factorization*:

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

theorem *Berlekamp-gcd-step*:

fixes $f::'a$ mod-ring poly **and** $h::'a$ mod-ring poly
assumes $hq\text{-mod-}f: [h \wedge (CARD('a)) = h]$ (mod f) **and** $monic\text{-}f: \text{monic } f$ **and** $sf\text{-}f: \text{square-free } f$
shows $f = \text{prod } (\lambda c. \text{gcd } f (h - [c:]))$ (UNIV:: $'a$ mod-ring set) (**is** ?lhs = ?rhs)
 ⟨proof⟩

6.3 Definitions

definition $berlekamp\text{-}mat :: 'a$ mod-ring poly $\Rightarrow 'a$ mod-ring mat **where**

$berlekamp\text{-}mat\ u = (\text{let } n = \text{degree } u;$
 $\text{mul-}p = \text{power-poly-f-mod } u\ [:0,1:] (CARD('a));$
 $xks = \text{power-polys mul-}p\ u\ 1\ n$
 in
 $\text{mat-of-rows-list } n\ (\text{map } (\lambda\ cs. \text{let } coeffs\text{-}cs = (coeffs\ cs);$
 $k = n - \text{length } (coeffs\ cs)$
 in $(coeffs\ cs) @ \text{replicate } k\ 0) xks)$

definition $berlekamp\text{-}resulting\text{-}mat :: ('a$ mod-ring) poly $\Rightarrow 'a$ mod-ring mat **where**

$berlekamp\text{-}resulting\text{-}mat\ u = (\text{let } Q = berlekamp\text{-}mat\ u;$
 $n = \text{dim-row } Q;$
 $QI = \text{mat } n\ n\ (\lambda\ (i,j). \text{if } i = j \text{ then } Q\ \$\$ (i,j) - 1 \text{ else } Q\ \$\$ (i,j))$
 in $(\text{gauss-jordan-single } (\text{transpose-mat } QI)))$

definition $berlekamp\text{-}basis :: 'a$ mod-ring poly $\Rightarrow 'a$ mod-ring poly list **where**

$berlekamp\text{-}basis\ u = (\text{map } (Poly\ o\ \text{list-of-vec}) (\text{find-base-vectors } (berlekamp\text{-}resulting\text{-}mat\ u)))$

lemma $berlekamp\text{-}basis\text{-}code[\text{code}]: berlekamp\text{-}basis\ u =$

$(\text{map } (poly\text{-of-list } o\ \text{list-of-vec}) (\text{find-base-vectors } (berlekamp\text{-}resulting\text{-}mat\ u)))$
 ⟨proof⟩

primrec $berlekamp\text{-}factorization\text{-}main :: nat \Rightarrow 'a$ mod-ring poly list $\Rightarrow 'a$ mod-ring poly list $\Rightarrow nat \Rightarrow 'a$ mod-ring poly list **where**

$berlekamp\text{-}factorization\text{-}main\ i\ \text{divs } (v \# vs)\ n = (\text{if } v = 1 \text{ then } berlekamp\text{-}factorization\text{-}main\ i\ \text{divs } vs\ n \text{ else}$

$\text{if } \text{length } \text{divs} = n \text{ then } \text{divs} \text{ else}$

$\text{let } \text{facts} = [w . u \leftarrow \text{divs}, s \leftarrow [0 ..< CARD('a)], w \leftarrow [\text{gcd } u (v - [:of-int\ s:]), w \neq 1];$

$(lin, nonlin) = \text{List.partition } (\lambda\ q. \text{degree } q = i) \text{ facts}$

$\text{in } lin @ berlekamp\text{-}factorization\text{-}main\ i\ nonlin\ vs\ (n - \text{length } lin)$

$| berlekamp\text{-}factorization\text{-}main\ i\ \text{divs } []\ n = \text{divs}$

definition $berlekamp\text{-}monic\text{-}factorization :: nat \Rightarrow 'a$ mod-ring poly $\Rightarrow 'a$ mod-ring poly list **where**

$berlekamp\text{-}monic\text{-}factorization\ d\ f = (\text{let}$
 $vs = berlekamp\text{-}basis\ f;$
 $n = \text{length } vs;$
 $fs = berlekamp\text{-}factorization\text{-}main\ d\ [f]\ vs\ n$

in fs)

6.4 Properties

lemma *power-polys-works*:

fixes $u::'b::\text{unique-euclidean-semiring}$

assumes $i: i < n$ **and** $c: \text{curr-p} = \text{curr-p mod } u$

shows $\text{power-polys mult-p } u \text{ curr-p } n ! i = \text{curr-p} * \text{mult-p} ^ i \text{ mod } u$

<proof>

lemma *length-power-polys[simp]*: $\text{length} (\text{power-polys mult-p } u \text{ curr-p } n) = n$

<proof>

lemma *Poly-berlekamp-mat*:

assumes $k: k < \text{degree } u$

shows $\text{Poly} (\text{list-of-vec} (\text{row} (\text{berlekamp-mat } u) k)) = [:0,1:] ^ (\text{CARD}('a) * k) \text{ mod } u$

<proof>

corollary *Poly-berlekamp-cong-mat*:

assumes $k: k < \text{degree } u$

shows $[\text{Poly} (\text{list-of-vec} (\text{row} (\text{berlekamp-mat } u) k))] = [:0,1:] ^ (\text{CARD}('a) * k) \text{ mod } u$

<proof>

lemma *mat-of-rows-list-dim[simp]*:

$\text{mat-of-rows-list } n \text{ vs} \in \text{carrier-mat} (\text{length } \text{vs}) n$

$\text{dim-row} (\text{mat-of-rows-list } n \text{ vs}) = \text{length } \text{vs}$

$\text{dim-col} (\text{mat-of-rows-list } n \text{ vs}) = n$

<proof>

lemma *berlekamp-mat-closed[simp]*:

$\text{berlekamp-mat } u \in \text{carrier-mat} (\text{degree } u) (\text{degree } u)$

$\text{dim-row} (\text{berlekamp-mat } u) = \text{degree } u$

$\text{dim-col} (\text{berlekamp-mat } u) = \text{degree } u$

<proof>

lemma *vec-of-list-coeffs-nth*:

assumes $i: i \in \{.. \text{degree } h\}$ **and** $h\text{-not0}: h \neq 0$

shows $\text{vec-of-list} (\text{coeffs } h) \$ i = \text{coeff } h i$

<proof>

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

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

lemma *dvd-choose-prime*:
assumes $kn: k < n$ **and** $k: k \neq 0$ **and** $n: n \neq 0$ **and** $prime-n: prime\ n$
shows $n\ dvd\ (n\ choose\ k)$
 $\langle proof \rangle$

lemma *add-power-poly-mod-ring*:
fixes $x :: 'a\ mod-ring\ poly$
shows $(x + y) \wedge CARD('a) = x \wedge CARD('a) + y \wedge CARD('a)$
 $\langle proof \rangle$

lemma *power-poly-sum-mod-ring*:
fixes $f :: 'b \Rightarrow 'a\ mod-ring\ poly$
assumes $f: finite\ A$
shows $(sum\ f\ A) \wedge CARD('a) = sum\ (\lambda i. (f\ i) \wedge CARD('a))\ A$
 $\langle proof \rangle$

lemma *poly-power-card-as-sum-of-monom*s:
fixes $h :: 'a\ mod-ring\ poly$
shows $h \wedge CARD('a) = (\sum\ i \leq degree\ h. monom\ (coeff\ h\ i)\ (CARD('a)*i))$
 $\langle proof \rangle$

lemma *degree-Poly-berlekamp-le*:
assumes $i: i < degree\ u$
shows $degree\ (Poly\ (list-of-vec\ (row\ (berlekamp-mat\ u)\ i))) < degree\ u$
 $\langle proof \rangle$

lemma *monom-card-pow-mod-sum-berlekamp*:
assumes $i: i < degree\ u$

shows *monom 1* ($CARD('a) * i \text{ mod } u = (\sum j < \text{degree } u. \text{monom } ((\text{berlekamp-mat } u) \text{ \$(i,j)) } j)$)
 <proof>

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

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

lemma *poly-as-sum-eq-monom*:
assumes $\text{ss-eq: } (\sum i < n. \text{monom } (f \text{ } i) \text{ } i) = (\sum i < n. \text{monom } (g \text{ } i) \text{ } i)$
and $a\text{-less-}n: a < n$
shows $f \text{ } a = g \text{ } a$
 <proof>

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

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

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

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

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 $[h \wedge CARD('a) = h] (mod\ u) \longleftrightarrow (transpose-mat\ (berlekamp-mat\ u)) *_{\mathbb{V}} H = H$
(is $?lhs = ?rhs)$
 $\langle proof \rangle$

end

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

lemma *exists-s-factor-dvd-h-s*:
fixes $fi::'a\ mod-ring\ poly$
assumes $finite-P: finite\ P$
and $f-desc-square-free: f = (\prod_{a \in P} a)$
and $P: P \subseteq \{q. irreducible\ q \wedge monic\ q\}$
and $fi-P: fi \in P$
and $h: h \in \{v. [v \wedge CARD('a) = v] (mod\ f)\}$
shows $\exists s. fi\ dvd\ (h - [:s:])$
 $\langle proof \rangle$

corollary *exists-unique-s-factor-dvd-h-s*:
fixes $fi::'a\ mod-ring\ poly$
assumes $finite-P: finite\ P$
and $f-desc-square-free: f = (\prod_{a \in P} a)$
and $P: P \subseteq \{q. irreducible\ q \wedge monic\ q\}$
and $fi-P: fi \in P$
and $h: h \in \{v. [v \wedge CARD('a) = v] (mod\ f)\}$
shows $\exists! s. fi\ dvd\ (h - [:s:])$
 $\langle proof \rangle$

lemma *exists-two-distint*: $\exists a b::'a \text{ mod-ring. } a \neq b$
 <proof>

lemma *coprime-cong-mult-factorization-poly*:

fixes $f::'b::\{\text{field}\}$ *poly*
and $a b p :: 'c :: \{\text{field-gcd}\}$ *poly*
assumes *finite-P*: *finite P*
and $P: P \subseteq \{q. \text{irreducible } q\}$
and $p: \forall p \in P. [a=b] \text{ (mod } p)$
and *coprime-P*: $\forall p1 p2. p1 \in P \wedge p2 \in P \wedge p1 \neq p2 \longrightarrow \text{coprime } p1 p2$
shows $[a = b] \text{ (mod } (\prod_{a \in P. a})$
 <proof>

end

context

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

begin

lemma *W-eq-berlekamp-mat*:

fixes $u::'a \text{ mod-ring poly}$
shows $\{v. [v \hat{C}ARD('a) = v] \text{ (mod } u) \wedge \text{degree } v < \text{degree } u\}$
 $= \{h. \text{let } H = \text{vec-of-list } ((\text{coeffs } h) @ \text{replicate } (\text{degree } u - \text{length } (\text{coeffs } h)) 0)$
in
 $(\text{transpose-mat } (\text{berlekamp-mat } u)) *_v H = H \wedge \text{degree } h < \text{degree } u\}$
 <proof>

lemma *transpose-minus-1*:

assumes $\text{dim-row } Q = \text{dim-col } Q$
shows $\text{transpose-mat } (Q - (1_m (\text{dim-row } Q))) = (\text{transpose-mat } Q - (1_m$
 $(\text{dim-row } Q)))$
 <proof>

lemma *system-iff*:

fixes $v::'b::\text{comm-ring-1 vec}$
assumes $\text{sq-}Q: \text{dim-row } Q = \text{dim-col } Q$ **and** $v: \text{dim-row } Q = \text{dim-vec } v$
shows $(\text{transpose-mat } Q *_v v = v) \longleftrightarrow ((\text{transpose-mat } Q - 1_m (\text{dim-row } Q)) *_v$
 $v = 0_v (\text{dim-vec } v))$
 <proof>

lemma *system-if-mat-kernel*:

assumes $\text{sq-}Q: \text{dim-row } Q = \text{dim-col } Q$ **and** $v: \text{dim-row } Q = \text{dim-vec } v$
shows $(\text{transpose-mat } Q *_v v = v) \longleftrightarrow v \in \text{mat-kernel } (\text{transpose-mat } (Q - (1_m$
 $(\text{dim-row } Q))))$

<proof>

lemma *degree-u-mod-irreducible_a-factor-0*:

fixes *v* **and** *u*::'a mod-ring poly

defines *W*: $W \equiv \{v. [v \wedge \text{CARD}('a) = v] \pmod{u}\}$

assumes *v*: $v \in W$

and *finite-U*: finite *U* **and** *u-U*: $u = \prod U$ **and** *U-irr-monic*: $U \subseteq \{q. \text{irreducible } q \wedge \text{monic } q\}$

and *fi-U*: $fi \in U$

shows $\text{degree}(v \pmod{fi}) = 0$

<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*]: $\mathbf{0}_{\text{poly-abelian-monoid}} = 0$

and [*simp*]: $\mathbf{1}_{\text{poly-abelian-monoid}} = 1$

and [*simp*]: $(\oplus_{\text{poly-abelian-monoid}}) = (+)$

and [*simp*]: $(\otimes_{\text{poly-abelian-monoid}}) = (*)$

and [*simp*]: *carrier poly-abelian-monoid* = *UNIV*

and [*simp*]: $(\odot_{\text{poly-abelian-monoid}}) = \text{smult}$

<proof>

lemma *subspace-Berlekamp*:

assumes *f*: $\text{degree } f \neq 0$

shows *subspace* (*class-ring* :: 'a mod-ring ring)

$\{v. [v \wedge (\text{CARD}('a)) = v] \pmod{f} \wedge (\text{degree } v < \text{degree } f)\}$ *poly-abelian-monoid*

<proof>

lemma *berlekamp-resulting-mat-closed*[*simp*]:

berlekamp-resulting-mat *u* \in *carrier-mat* ($\text{degree } u$) ($\text{degree } u$)

dim-row (*berlekamp-resulting-mat* *u*) = $\text{degree } u$

dim-col (*berlekamp-resulting-mat* *u*) = $\text{degree } u$

<proof>

lemma *berlekamp-resulting-mat-basis*:

kernel.basis ($\text{degree } u$) (*berlekamp-resulting-mat* *u*) (*set* (*find-base-vectors* (*berlekamp-resulting-mat* *u*)))

<proof>

lemma *set-berlekamp-basis-eq*: $(\text{set } (\text{berlekamp-basis } u))$
 $= (\text{Poly } \circ \text{list-of-vec})' (\text{set } (\text{find-base-vectors } (\text{berlekamp-resulting-mat } u)))$
 $\langle \text{proof} \rangle$

lemma *berlekamp-resulting-mat-constant*:
assumes *deg-u*: $\text{degree } u = 0$
shows *berlekamp-resulting-mat* $u = 1_m \ 0$
 $\langle \text{proof} \rangle$

context
fixes *u::'a::prime-card mod-ring poly*
begin

lemma *set-berlekamp-basis-constant*:
assumes *deg-u*: $\text{degree } u = 0$
shows $\text{set } (\text{berlekamp-basis } u) = \{\}$
 $\langle \text{proof} \rangle$

lemma *row-echelon-form-berlekamp-resulting-mat*: *row-echelon-form* $(\text{berlekamp-resulting-mat } u)$
 $\langle \text{proof} \rangle$

lemma *mat-kernel-berlekamp-resulting-mat-degree-0*:
assumes *d*: $\text{degree } u = 0$
shows *mat-kernel* $(\text{berlekamp-resulting-mat } u) = \{0_v \ 0\}$
 $\langle \text{proof} \rangle$

lemma *in-mat-kernel-berlekamp-resulting-mat*:
assumes *x*: $\text{transpose-mat } (\text{berlekamp-mat } u) *_v x = x$
and *x-dim*: $x \in \text{carrier-vec } (\text{degree } u)$
shows $x \in \text{mat-kernel } (\text{berlekamp-resulting-mat } u)$
 $\langle \text{proof} \rangle$ **abbreviation** $V \equiv \text{kernel.VK } (\text{degree } u) (\text{berlekamp-resulting-mat } u)$
private abbreviation $W \equiv \text{vector-space-poly.vs } \{v. [v \wedge (\text{CARD}('a)) = v] (\text{mod } u) \wedge (\text{degree } v < \text{degree } u)\}$

interpretation V : *vectorspace class-ring* V
 $\langle \text{proof} \rangle$

lemma *linear-Poly-list-of-vec*:
shows $(\text{Poly } \circ \text{list-of-vec}) \in \text{module-hom class-ring } V (\text{vector-space-poly.vs } \{v. [v \wedge (\text{CARD}('a)) = v] (\text{mod } u)\})$
 $\langle \text{proof} \rangle$

lemma *linear-Poly-list-of-vec'*:
assumes *degree u > 0*
shows $(Poly \circ list-of-vec) \in module-hom R V W$
 $\langle proof \rangle$

lemma *berlekamp-basis-eq-8*:
assumes $v: v \in set (berlekamp-basis u)$
shows $[v \wedge CARD('a) = v] (mod u)$
 $\langle proof \rangle$

lemma *surj-Poly-list-of-vec*:
assumes *deg-u: degree u > 0*
shows $(Poly \circ list-of-vec)' (carrier V) = carrier W$
 $\langle proof \rangle$

lemma *card-set-berlekamp-basis*: $card (set (berlekamp-basis u)) = length (berlekamp-basis u)$
 $\langle proof \rangle$

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

interpretation *Berlekamp-subspace: vectorspace class-ring W*
 $\langle proof \rangle$

lemma *linear-map-Poly-list-of-vec'*: $linear-map class-ring V W (Poly \circ list-of-vec)$
 $\langle proof \rangle$

lemma *berlekamp-basis-basis*:
 $Berlekamp-subspace.basis (set (berlekamp-basis u))$
 $\langle proof \rangle$

lemma *finsum-sum*:
fixes $f::'a mod-ring poly$
assumes $f: finite B$
and $a-Pi: a \in B \rightarrow carrier R$
and $V: B \subseteq carrier W$
shows $(\bigoplus_{Wv \in B. a v \odot_W v) = sum (\lambda v. smult (a v) v) B$
 $\langle proof \rangle$

lemma *exists-vector-in-Berlekamp-subspace-dvd*:
fixes $p-i::'a mod-ring poly$
assumes $finite-P: finite P$

and *f-desc-square-free*: $u = (\prod_{a \in P} a)$
and $P: P \subseteq \{q. \text{irreducible } q \wedge \text{monic } q\}$
and $pi: p-i \in P$ **and** $pj: p-j \in P$ **and** $pi-pj: p-i \neq p-j$
and *monic-f*: *monic* u **and** *sf-f*: *square-free* u
and *not-irr-w*: \neg *irreducible* w
and *w-dvd-f*: $w \text{ dvd } u$ **and** *monic-w*: *monic* w
and *pi-dvd-w*: $p-i \text{ dvd } w$ **and** *pj-dvd-w*: $p-j \text{ dvd } w$
shows $\exists v. v \in \{h. [h \wedge \text{CARD}(a) = h] \pmod{u} \wedge \text{degree } h < \text{degree } u\}$
 $\wedge v \pmod{p-i} \neq v \pmod{p-j}$
 $\wedge \text{degree } (v \pmod{p-i}) = 0$
 $\wedge \text{degree } (v \pmod{p-j}) = 0$
— This implies that the algorithm decreases the degree of the reducible polynomials
in each step:
 $\wedge (\exists s. \text{gcd } w (v - [:s]) \neq w \wedge \text{gcd } w (v - [:s]) \neq 1)$
 $\langle \text{proof} \rangle$

lemma *exists-vector-in-Berlekamp-basis-dvd-aux*:
assumes *basis-V*: *Berlekamp-subspace.basis* B
and *finite-V*: *finite* B
assumes *finite-P*: *finite* P
and *f-desc-square-free*: $u = (\prod_{a \in P} a)$
and $P: P \subseteq \{q. \text{irreducible } q \wedge \text{monic } q\}$
and $pi: p-i \in P$ **and** $pj: p-j \in P$ **and** $pi-pj: p-i \neq p-j$
and *monic-f*: *monic* u **and** *sf-f*: *square-free* u
and *not-irr-w*: \neg *irreducible* w
and *w-dvd-f*: $w \text{ dvd } u$ **and** *monic-w*: *monic* w
and *pi-dvd-w*: $p-i \text{ dvd } w$ **and** *pj-dvd-w*: $p-j \text{ dvd } w$
shows $\exists v \in B. v \pmod{p-i} \neq v \pmod{p-j}$
 $\langle \text{proof} \rangle$

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*: $u = (\prod_{a \in P} a)$
and $P: P \subseteq \{q. \text{irreducible } q \wedge \text{monic } q\}$
and $pi: p-i \in P$ **and** $pj: p-j \in P$ **and** $pi-pj: p-i \neq p-j$
and *monic-f*: *monic* u **and** *sf-f*: *square-free* u
and *not-irr-w*: \neg *irreducible* w
and *w-dvd-f*: $w \text{ dvd } u$ **and** *monic-w*: *monic* w
and *pi-dvd-w*: $p-i \text{ dvd } w$ **and** *pj-dvd-w*: $p-j \text{ dvd } w$
shows $\exists v \in B. v \pmod{p-i} \neq v \pmod{p-j}$
 $\wedge \text{degree } (v \pmod{p-i}) = 0$
 $\wedge \text{degree } (v \pmod{p-j}) = 0$
— This implies that the algorithm decreases the degree of the reducible polynomials
in each step:

$\wedge (\exists s. \text{gcd } w (v - [s:]) \neq w \wedge \neg \text{coprime } w (v - [s:]))$
 $\langle \text{proof} \rangle$

lemma *exists-bijective-linear-map-W-vec*:

assumes *finite-P*: *finite P*

and *u-desc-square-free*: $u = (\prod_{a \in P}. a)$

and *P*: $P \subseteq \{q. \text{irreducible } q \wedge \text{monic } q\}$

shows $\exists f. \text{linear-map class-ring } W (\text{module-vec TYPE('a mod-ring) (card P)}) f$
 $\wedge \text{bij-betw } f (\text{carrier } W) (\text{carrier-vec (card P)::'a mod-ring vec set})$

$\langle \text{proof} \rangle$

lemma *fin-dim-kernel-berlekamp*: $V.\text{fin-dim}$

$\langle \text{proof} \rangle$

lemma *Berlekamp-subspace-fin-dim*: *Berlekamp-subspace.fin-dim*

$\langle \text{proof} \rangle$

context

fixes *P*

assumes *finite-P*: *finite P*

and *u-desc-square-free*: $u = (\prod_{a \in P}. a)$

and *P*: $P \subseteq \{q. \text{irreducible } q \wedge \text{monic } q\}$

begin

interpretation *RV*: *vec-space TYPE('a mod-ring) card P* $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *Berlekamp-subspace-dim*: *Berlekamp-subspace.dim = card P*

$\langle \text{proof} \rangle$

corollary *card-berlekamp-basis-number-factors*: $\text{card (set (berlekamp-basis } u)) = \text{card } P$

$\langle \text{proof} \rangle$

lemma *length-berlekamp-basis-numbers-factors*: $\text{length (berlekamp-basis } u) = \text{card } P$

$\langle \text{proof} \rangle$

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:  $\bigwedge u. u \in \text{set } us \implies \text{degree } u > 0$ 
  shows length us  $\leq$  n
  <proof>

lemma berlekamp-basis-irreducible: assumes f: f = prod-list us
  and n-us: length us = n
  and us:  $\bigwedge u. u \in \text{set } us \implies \text{degree } u > 0$ 
  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-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  $\exists pi pj. pi \in P \wedge pj \in P \wedge pi \neq pj \wedge pi \text{ dvd } u \wedge pj \text{ dvd } u$ 
  <proof>

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

lemma berlekamp-monic-factorization:
  fixes f::'a mod-ring poly

```

```

assumes sf-f: square-free f
and us: berlekamp-monic-factorization d f = us
and d:  $\bigwedge g. g \text{ dvd } f \implies \text{degree } g = d \implies \text{irreducible } g$ 
and deg: degree f > 0
and mon: monic f
shows f = prod-list us  $\wedge$  ( $\forall u \in \text{set us. monic } u \wedge \text{irreducible } u$ )
<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 \Rightarrow 'a :: field poly \Rightarrow bool* **where**
factors-of-same-degree i f = (i \neq 0 \wedge degree f \neq 0 \wedge monic f \wedge ($\forall g. \text{irreducible } g \longrightarrow g \text{ dvd } f \longrightarrow \text{degree } g = i$))

lemma *factors-of-same-degreeD: assumes factors-of-same-degree i f*
shows *i \neq 0 degree f \neq 0 monic f g dvd f \implies 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 $\exists a \in \text{carrier } (mult\text{-of } R). \text{group.ord } (mult\text{-of } R) a = \text{order } (mult\text{-of } R)$

$\wedge \text{carrier } (mult\text{-of } R) = \{a[\wedge]i \mid i::nat . i \in UNIV\}$

<proof>

lemma *add-power-prime-poly-mod-ring[simp]:*

fixes *x :: 'a::{prime-card} mod-ring poly*

shows $(x + y) \wedge \text{CARD}'a \wedge n = x \wedge (\text{CARD}'a \wedge n) + y \wedge \text{CARD}'a \wedge n$

<proof>

lemma *fermat-theorem-mod-ring2[simp]:*

fixes $a::'a::\{prime-card\} mod-ring$
shows $a \wedge (CARD('a) \wedge n) = a$
 $\langle proof \rangle$

lemma *fermat-theorem-power-poly*[simp]:
fixes $a::'a::prime-card mod-ring$
shows $[:a:] \wedge CARD('a::prime-card) \wedge n = [:a:]$
 $\langle proof \rangle$

lemma *degree-prod-monom*: $degree (\prod i = 0..<n. monom 1 1) = n$
 $\langle proof \rangle$

lemma *degree-monom0*[simp]: $degree (monom a 0) = 0$ $\langle proof \rangle$
lemma *degree-monom0'*[simp]: $degree (monom 0 b) = 0$ $\langle proof \rangle$

lemma *sum-monom-mod*:
assumes $b < degree f$
shows $(\sum i \leq b. monom (g i) i) mod f = (\sum i \leq b. monom (g i) i)$
 $\langle proof \rangle$

lemma *x-power-aq-minus-1-rw*:
fixes $x::nat$
assumes $x: x > 1$
and $a: a > 0$
and $b: b > 0$
shows $x \wedge (a * q) - 1 = ((x \wedge a) - 1) * sum ((\wedge) (x \wedge a)) \{..<q\}$
 $\langle proof \rangle$

lemma *dvd-power-minus-1-conv1*:
fixes $x::nat$
assumes $x: x > 1$
and $a: a > 0$
and $xa-dvd: x \wedge a - 1 dvd x \wedge b - 1$
and $b0: b > 0$
shows $a dvd b$
 $\langle proof \rangle$

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 $x \wedge a - 1 dvd x \wedge b - 1$
 $\langle proof \rangle$

corollary *dvd-power-minus-1-conv*:

```

fixes  $x::\text{nat}$ 
assumes  $x: x > 1$ 
  and  $a: a > 0$ 
  and  $b0: b > 0$ 
shows  $a \text{ dvd } b = (x^a - 1 \text{ dvd } x^b - 1)$ 
   $\langle \text{proof} \rangle$ 

locale  $\text{poly-mod-type-irr} = \text{poly-mod-type } m \text{ TYPE}('a::\text{prime-card})$  for  $m +$ 
  fixes  $f::'a::\{\text{prime-card}\} \text{ mod-ring poly}$ 
  assumes  $\text{irr-f}: \text{irreducible}_a f$ 
begin

definition  $\text{plus-irr} :: 'a \text{ mod-ring poly} \Rightarrow 'a \text{ mod-ring poly} \Rightarrow 'a \text{ mod-ring poly}$ 
  where  $\text{plus-irr } a \ b = (a + b) \text{ mod } f$ 

definition  $\text{minus-irr} :: 'a \text{ mod-ring poly} \Rightarrow 'a \text{ mod-ring poly} \Rightarrow 'a \text{ mod-ring poly}$ 
  where  $\text{minus-irr } x \ y \equiv (x - y) \text{ mod } f$ 

definition  $\text{uminus-irr} :: 'a \text{ mod-ring poly} \Rightarrow 'a \text{ mod-ring poly}$ 
  where  $\text{uminus-irr } x = -x$ 

definition  $\text{mult-irr} :: 'a \text{ mod-ring poly} \Rightarrow 'a \text{ mod-ring poly} \Rightarrow 'a \text{ mod-ring poly}$ 
  where  $\text{mult-irr } x \ y = ((x*y) \text{ mod } f)$ 

definition  $\text{carrier-irr} :: 'a \text{ mod-ring poly set}$ 
  where  $\text{carrier-irr} = \{x. \text{degree } x < \text{degree } f\}$ 

definition  $\text{power-irr} :: 'a \text{ mod-ring poly} \Rightarrow \text{nat} \Rightarrow 'a \text{ mod-ring poly}$ 
  where  $\text{power-irr } p \ n = ((p^{\wedge}n) \text{ mod } f)$ 

definition  $R = (\text{carrier} = \text{carrier-irr}, \text{monoid.mult} = \text{mult-irr}, \text{one} = 1, \text{zero} =$ 
   $0, \text{add} = \text{plus-irr})$ 

lemma  $\text{degree-f[simp]}: \text{degree } f > 0$ 
   $\langle \text{proof} \rangle$ 

lemma  $\text{element-in-carrier}: (a \in \text{carrier } R) = (\text{degree } a < \text{degree } f)$ 
   $\langle \text{proof} \rangle$ 

lemma  $f\text{-dvd-ab}:$ 
   $a = 0 \vee b = 0$  if  $f \text{ dvd } a * b$ 
  and  $a: \text{degree } a < \text{degree } f$ 
  and  $b: \text{degree } b < \text{degree } f$ 
   $\langle \text{proof} \rangle$ 

```

lemma *ab-mod-f0*:

$a = 0 \vee b = 0$ **if** $a * b \bmod f = 0$

and a : *degree a < degree f*

and b : *degree b < degree f*

<proof>

lemma *irreducible_dD2*:

fixes $p\ q :: 'b::\{\text{comm-semiring-1, semiring-no-zero-divisors}\}$ *poly*

assumes *irreducible_d p*

and *degree q < degree p* **and** *degree q \neq 0*

shows $\neg q \text{ dvd } p$

<proof>

lemma *times-mod-f-1-imp-0*:

assumes x : *degree x < degree f*

and $x2$: $\forall xa. x * xa \bmod f = 1 \longrightarrow \neg \text{degree } xa < \text{degree } f$

shows $x = 0$

<proof>

sublocale *field-R*: *field R*

<proof>

lemma *zero-in-carrier[simp]*: $0 \in \text{carrier-irr}$ *<proof>*

lemma *card-carrier-irr[simp]*: $\text{card carrier-irr} = \text{CARD}('a)^{\wedge}(\text{degree } f)$

<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]*: $[:a:] \in \text{carrier } R$

<proof>

lemma *mod-in-carrier[simp]*: $a \bmod f \in \text{carrier } R$

<proof>

lemma *order-irr*: $\text{Coset.order (mult-of } R) = \text{CARD}('a)^{\wedge}(\text{degree } f) - 1$

<proof>

lemma *element-power-order-eq-1*:

assumes x : $x \in \text{carrier (mult-of } R)$

shows $x [^{\wedge}]_{(\text{mult-of } R)} \text{Coset.order (mult-of } R) = \mathbf{1}_{(\text{mult-of } R)}$

<proof>

corollary *element-power-order-eq-1'*:

assumes $x: x \in \text{carrier (mult-of } R)$

shows $x \lceil_{(\text{mult-of } R)} \text{CARD}'a \wedge \text{degree } f = x$

<proof>

lemma *pow-irr[simp]*: $x \lceil_{(R)} n = x \wedge n \text{ mod } f$

<proof>

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

<proof>

lemma *fermat-theorem-power-poly-R[simp]*: $[:a:] \lceil_R \text{CARD}'a \wedge n = [:a:]$

<proof>

lemma *times-mod-expand*:

$(a \otimes_{(R)} b) = ((a \text{ mod } f) \otimes_{(R)} (b \text{ mod } f))$

<proof>

lemma *mult-closed-power*:

assumes $x: x \in \text{carrier } R$ **and** $y: y \in \text{carrier } R$

and $x \lceil_{(R)} \text{CARD}'a \wedge m' = x$

and $y \lceil_{(R)} \text{CARD}'a \wedge m' = y$

shows $(x \otimes_{(R)} y) \lceil_{(R)} \text{CARD}'a \wedge m' = (x \otimes_{(R)} y)$

<proof>

lemma *add-closed-power*:

assumes $x1: x \lceil_{(R)} \text{CARD}'a \wedge m' = x$

and $y1: y \lceil_{(R)} \text{CARD}'a \wedge m' = y$

shows $(x \oplus_{(R)} y) \lceil_{(R)} \text{CARD}'a \wedge m' = (x \oplus_{(R)} y)$

<proof>

lemma *x-power-pm-minus-1*:

assumes $x: x \in \text{carrier (mult-of } R)$

and $x \lceil_{(R)} \text{CARD}'a \wedge m' = x$

shows $x \lceil_{(R)} (\text{CARD}'a \wedge m' - 1) = \mathbf{1}_{(R)}$

<proof>

context

begin

private lemma *monom-a-1-P*:

assumes $m: \text{monom } 1 \ 1 \in \text{carrier } R$

and $eq: \text{monom } 1 \ 1 \lceil_{(R)} (\text{CARD}'a \wedge m') = \text{monom } 1 \ 1$

shows $\text{monom } a \ 1 \lceil_{(R)} (\text{CARD}'a \wedge m') = \text{monom } a \ 1$

<proof> **lemma** *prod-monom-1-1*:
defines $P == (\lambda x n. (x[\wedge]_{(R)} (CARD('a) \wedge n) = x))$
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 $P == (\lambda x n. (x[\wedge]_{(R)} (CARD('a) \wedge n) = x))$
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 $P == (\lambda x n. (x[\wedge]_{(R)} (CARD('a) \wedge n) = x))$
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-monom-P*:
defines $P == (\lambda x n. (x[\wedge]_{(R)} (CARD('a) \wedge n) = x))$
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 $P \equiv (\lambda x n. (x[\wedge]_{(R)} (CARD('a) \wedge n) = x))$
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

lemma *degree-divisor1*:
assumes $f: irreducible\ (f :: 'a :: prime-card\ mod-ring\ poly)$
and $d: degree\ f = d$
shows $f\ dvd\ (monom\ 1\ 1) \wedge (CARD('a) \wedge d) - monom\ 1\ 1$
 <proof>

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

<proof>

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

context

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

begin

function *dist-degree-factorize-main* ::

$'a \text{ mod-ring poly} \Rightarrow 'a \text{ mod-ring poly} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times 'a \text{ mod-ring poly}) \text{ list}$

$\Rightarrow (\text{nat} \times 'a \text{ mod-ring poly}) \text{ list}$ **where**

dist-degree-factorize-main $v \ w \ d \ \text{res} = (\text{if } v = 1 \ \text{then } \text{res} \ \text{else if } d + d > \text{degree } v$

then $(\text{degree } v, v) \# \text{res}$ *else let*

$w = w \wedge (\text{CARD}('a)) \text{ mod } v;$

$d = \text{Suc } d;$

$gd = \text{gcd } (w - \text{monom } 1 \ 1) \ v$

in if $gd = 1$ *then* *dist-degree-factorize-main* $v \ w \ d \ \text{res}$ *else*

let $v' = v \ \text{div } gd$ *in*

dist-degree-factorize-main $v' \ (w \ \text{mod } v') \ d \ ((d, gd) \# \text{res})$

<proof>

termination

<proof>

declare *dist-degree-factorize-main.simps*[*simp del*]

lemma *dist-degree-factorize-main*: **assumes**

dist: *dist-degree-factorize-main* $v \ w \ d \ \text{res} = \text{facts}$ **and**

$w = (\text{monom } 1 \ 1) \wedge (\text{CARD}('a) \wedge d) \text{ mod } v$ **and**

sf: *square-free* u **and**

mon: *monic* u **and**

prod: $u = v * \text{prod-list } (\text{map } \text{snd } \text{res})$ **and**

deg: $\bigwedge f. \text{irreducible } f \implies f \text{ dvd } v \implies \text{degree } f > d$ **and**

res: $\bigwedge i f. (i, f) \in \text{set } \text{res} \implies i \neq 0 \wedge \text{degree } f \neq 0 \wedge \text{monic } f \wedge (\forall g. \text{irreducible } g \longrightarrow g \text{ dvd } f \longrightarrow \text{degree } g = i)$

shows $u = \text{prod-list } (\text{map } \text{snd } \text{facts}) \wedge (\forall i f. (i, f) \in \text{set } \text{facts} \longrightarrow \text{factors-of-same-degree } i \ f)$

<proof>

definition *distinct-degree-factorization*

$:: 'a \text{ mod-ring poly} \Rightarrow (\text{nat} \times 'a \text{ mod-ring poly}) \text{ list}$ **where**

distinct-degree-factorization $f =$

$(\text{if } \text{degree } f = 1 \ \text{then } [(1, f)] \ \text{else } \text{dist-degree-factorize-main } f \ (\text{monom } 1 \ 1) \ 0$

\square)

```

lemma distinct-degree-factorization: assumes
  dist: distinct-degree-factorization  $f = \text{facts}$  and
  u: square-free  $f$  and
  mon: monic  $f$ 
shows  $f = \text{prod-list } (\text{map snd facts}) \wedge (\forall i f. (i,f) \in \text{set facts} \longrightarrow \text{factors-of-same-degree } i f)$ 
 $\langle \text{proof} \rangle$ 
end

end

```

8 A Combined Factorization Algorithm for Polynomials over $\text{GF}(p)$

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 $\text{GF}(p)$.

```

theory Finite-Field-Factorization
imports Berlekamp-Type-Based
  Distinct-Degree-Factorization
begin

```

We prove soundness of the finite field factorization, independent 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  $\Rightarrow$  'a mod-ring  $\times$  'a mod-ring
poly list where

```

```

  finite-field-factorization  $f = (\text{if degree } f = 0 \text{ then } (\text{lead-coeff } f, []) \text{ else let}$ 
     $a = \text{lead-coeff } f;$ 
     $u = \text{smult } (\text{inverse } a) f;$ 
     $gs = (\text{if use-distinct-degree-factorization then distinct-degree-factorization } u \text{ else}$ 
 $[(1, u)]);$ 
     $(\text{irr}, \text{hs}) = \text{List.partition } (\lambda (i, f). \text{degree } f = i) \text{ gs}$ 
     $\text{in } (a, \text{map snd irr } @ \text{concat } (\text{map } (\lambda (i, g). \text{berlekamp-monic-factorization } i g) \text{ hs}))$ 

```

```

lemma finite-field-factorization-explicit:
fixes  $f :: 'a \text{ mod-ring poly}$ 
assumes  $\text{sf-f}: \text{square-free } f$ 
and  $\text{us}: \text{finite-field-factorization } f = (c, \text{us})$ 

```

shows $f = \text{smult } c \text{ (prod-list us)} \wedge (\forall u \in \text{set us. monic } u \wedge \text{irreducible } u)$
 <proof>

lemma *finite-field-factorization*:

fixes $f :: 'a \text{ mod-ring poly}$

assumes $\text{sf-f: square-free } f$

and $\text{us: finite-field-factorization } f = (c, \text{us})$

shows *unique-factorization Irr-Mon* $f \text{ (c, mset us)}$

<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* \equiv *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 $\llbracket \text{square-free } ?f; \text{finite-field-factorization } ?f = (?c, ?us) \rrbracket \implies \text{unique-factorization Irr-Mon } ?f \text{ (?c, mset ?us)}$ provides a result for a polynomial over $\text{GF}(p)$, we now develop a theorem which speaks about integer polynomials modulo p .

lemma (**in** *poly-mod-prime-type*) *finite-field-factorization-modulo-ring*:

assumes $g: (g :: 'a \text{ mod-ring poly}) = \text{of-int-poly } f$

and $\text{sf: square-free-m } f$

and $\text{fact: finite-field-factorization } g = (d, \text{gs})$

and $c: c = \text{to-int-mod-ring } d$

and $\text{fs: fs} = \text{map to-int-poly } \text{gs}$

shows *unique-factorization-m* $f \text{ (c, mset fs)}$

<proof>

We now have to implement *finite-field-factorization*.

context

fixes $p :: \text{int}$
and $\text{ff-ops} :: 'i \text{ arith-ops-record}$
begin

fun $\text{power-poly-f-mod-}i :: ('i \text{ list} \Rightarrow 'i \text{ list}) \Rightarrow 'i \text{ list} \Rightarrow \text{nat} \Rightarrow 'i \text{ list}$ **where**
 $\text{power-poly-f-mod-}i \text{ modulus } a \ n = (\text{if } n = 0 \text{ then modulus (one-poly-}i \ \text{ff-ops)}$
 $\text{else let } (d,r) = \text{Divides.divmod-nat } n \ 2;$
 $\text{rec} = \text{power-poly-f-mod-}i \text{ modulus (modulus (times-poly-}i \ \text{ff-ops } a \ a)) \ d \ \text{in}$
 $\text{if } r = 0 \text{ then rec else modulus (times-poly-}i \ \text{ff-ops } \text{rec } a))$

declare $\text{power-poly-f-mod-}i.\text{simps}[simp \ \text{del}]$

fun $\text{power-polys-}i :: 'i \text{ list} \Rightarrow 'i \text{ list} \Rightarrow 'i \text{ list} \Rightarrow \text{nat} \Rightarrow 'i \text{ list list}$ **where**
 $\text{power-polys-}i \ \text{mul-p } u \ \text{curr-p } (\text{Suc } i) = \text{curr-p } \#$
 $\text{power-polys-}i \ \text{mul-p } u \ (\text{mod-field-poly-}i \ \text{ff-ops } (\text{times-poly-}i \ \text{ff-ops } \text{curr-p } \text{mul-p})$
 $u) \ i$
 $| \ \text{power-polys-}i \ \text{mul-p } u \ \text{curr-p } 0 = []$

lemma $\text{length-power-polys-}i[\text{simp}]$: $\text{length } (\text{power-polys-}i \ x \ y \ z \ n) = n$
 $\langle \text{proof} \rangle$

definition $\text{berlekamp-mat-}i :: 'i \text{ list} \Rightarrow 'i \text{ mat}$ **where**

$\text{berlekamp-mat-}i \ u = (\text{let } n = \text{degree-}i \ u;$
 $ze = \text{arith-ops-record.zero } \text{ff-ops}; \ on = \text{arith-ops-record.one } \text{ff-ops};$
 $\text{mul-p} = \text{power-poly-f-mod-}i \ (\lambda \ v. \ \text{mod-field-poly-}i \ \text{ff-ops } v \ u)$
 $[ze, \ on] \ (\text{nat } p);$
 $xks = \text{power-polys-}i \ \text{mul-p } u \ [on] \ n$
 $\text{in } \text{mat-of-rows-list } n \ (\text{map } (\lambda \ cs. \ cs \ @ \ \text{replicate } (n - \text{length } cs) \ ze) \ xks))$

definition $\text{berlekamp-resulting-mat-}i :: 'i \text{ list} \Rightarrow 'i \text{ mat}$ **where**

$\text{berlekamp-resulting-mat-}i \ u = (\text{let } Q = \text{berlekamp-mat-}i \ u;$
 $n = \text{dim-row } Q;$
 $QI = \text{mat } n \ n \ (\lambda \ (i,j). \ \text{if } i = j \ \text{then } \text{arith-ops-record.minus } \text{ff-ops } (Q \ \text{\$} \ \$ \ (i,j))$
 $(\text{arith-ops-record.one } \text{ff-ops}) \ \text{else } Q \ \text{\$} \ \$ \ (i,j))$
 $\text{in } (\text{gauss-jordan-single-}i \ \text{ff-ops } (\text{transpose-mat } QI)))$

definition $\text{berlekamp-basis-}i :: 'i \text{ list} \Rightarrow 'i \text{ list list}$ **where**

$\text{berlekamp-basis-}i \ u = (\text{map } (\text{poly-of-list-}i \ \text{ff-ops } o \ \text{list-of-vec})$
 $(\text{find-base-vectors-}i \ \text{ff-ops } (\text{berlekamp-resulting-mat-}i \ u)))$

primrec $\text{berlekamp-factorization-main-}i :: 'i \Rightarrow 'i \Rightarrow \text{nat} \Rightarrow 'i \text{ list list} \Rightarrow 'i \text{ list list}$
 $\Rightarrow \text{nat} \Rightarrow 'i \text{ list list}$ **where**

$\text{berlekamp-factorization-main-}i \ ze \ on \ d \ \text{divs } (v \ \# \ vs) \ n = ($
 $\text{if } v = [on] \ \text{then } \text{berlekamp-factorization-main-}i \ ze \ on \ d \ \text{divs } vs \ n \ \text{else}$
 $\text{if } \text{length } \text{divs} = n \ \text{then } \text{divs} \ \text{else}$
 $\text{let } \text{of-int} = \text{arith-ops-record.of-int } \text{ff-ops};$
 $\text{facts} = \text{filter } (\lambda \ w. \ w \neq [on])$

```

[ 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 : $\text{poly-rel } h \ h'$
shows $\text{poly-rel } (\text{power-polys-}i \ \text{ff-ops } g \ f \ h \ n \ ! \ i) \ (\text{power-polys } g' \ f' \ h' \ n \ ! \ i)$
 $\langle \text{proof} \rangle$

lemma $\text{power-poly-f-mod-}i$: **assumes** m : $(\text{poly-rel } == => \text{poly-rel}) \ m \ (\lambda \ x'. \ x' \ \text{mod } m \wedge)$
shows $\text{poly-rel } f \ f' \implies \text{poly-rel } (\text{power-poly-f-mod-}i \ \text{ff-ops } m \ f \ n) \ (\text{power-poly-f-mod } m' \ f' \ n)$
 $\langle \text{proof} \rangle$

lemma $\text{berlekamp-mat-}i$ [transfer-rule]: $(\text{poly-rel } == => \text{mat-rel } R)$
 $(\text{berlekamp-mat-}i \ p \ \text{ff-ops}) \ \text{berlekamp-mat}$
 $\langle \text{proof} \rangle$

lemma $\text{berlekamp-resulting-mat-}i$ [transfer-rule]: $(\text{poly-rel } == => \text{mat-rel } R)$
 $(\text{berlekamp-resulting-mat-}i \ p \ \text{ff-ops}) \ \text{berlekamp-resulting-mat}$
 $\langle \text{proof} \rangle$

lemma $\text{berlekamp-basis-}i$ [transfer-rule]: $(\text{poly-rel } == => \text{list-all2 } \text{poly-rel})$
 $(\text{berlekamp-basis-}i \ p \ \text{ff-ops}) \ \text{berlekamp-basis}$
 $\langle \text{proof} \rangle$

lemma $\text{berlekamp-factorization-main-}i$ [transfer-rule]:
 $((=) == => \text{list-all2 } \text{poly-rel } == => \text{list-all2 } \text{poly-rel } == => (=) == => \text{list-all2 } \text{poly-rel})$
 $(\text{berlekamp-factorization-main-}i \ p \ \text{ff-ops} \ (\text{arith-ops-record.zero } \text{ff-ops})$
 $(\text{arith-ops-record.one } \text{ff-ops}))$
 $\text{berlekamp-factorization-main}$
 $\langle \text{proof} \rangle$

lemma $\text{berlekamp-monic-factorization-}i$ [transfer-rule]:
 $((=) == => \text{poly-rel } == => \text{list-all2 } \text{poly-rel})$
 $(\text{berlekamp-monic-factorization-}i \ p \ \text{ff-ops}) \ \text{berlekamp-monic-factorization}$
 $\langle \text{proof} \rangle$

lemma $\text{dist-degree-factorize-main-}i$:
 $\text{poly-rel } F \ f \implies \text{poly-rel } G \ g \implies \text{list-all2 } (\text{rel-prod } (=) \ \text{poly-rel}) \ \text{Res } \text{res}$
 $\implies \text{list-all2 } (\text{rel-prod } (=) \ \text{poly-rel})$
 $(\text{dist-degree-factorize-main-}i \ p \ \text{ff-ops}$
 $(\text{arith-ops-record.zero } \text{ff-ops}) \ (\text{arith-ops-record.one } \text{ff-ops}) \ (\text{degree-}i \ F) \ F \ G$
 $d \ \text{Res})$
 $(\text{dist-degree-factorize-main } f \ g \ d \ \text{res})$
 $\langle \text{proof} \rangle$

lemma $\text{distinct-degree-factorization-}i$ [transfer-rule]: $(\text{poly-rel } == => \text{list-all2 } (\text{rel-prod } (=) \ \text{poly-rel}))$
 $(\text{distinct-degree-factorization-}i \ p \ \text{ff-ops}) \ \text{distinct-degree-factorization}$
 $\langle \text{proof} \rangle$

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' : $f' = \text{of-int-poly-i ff-ops } (Mp\ f)$
and berl-i : *finite-field-factorization-i* p ff-ops $f' = (c',fs')$
and sq : *square-free-m* f
and fs : $\text{fs} = \text{map } (\text{to-int-poly-i ff-ops})\ \text{fs}'$
and c : $c = \text{arith-ops-record.to-int ff-ops } c'$
shows *unique-factorization-m* f (c, mset fs)
 $\wedge c \in \{0 \dots p\}$
 $\wedge (\forall fi \in \text{set fs. set (coeffs fi)} \subseteq \{0 \dots p\})$
 <proof>
end

definition *finite-field-factorization-main* :: int \Rightarrow 'i arith-ops-record \Rightarrow int poly \Rightarrow int \times int poly list **where**
finite-field-factorization-main p f-ops f \equiv
 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)
 $\wedge c \in \{0 \dots p\}$
 $\wedge (\forall fi \in \text{set fs. set (coeffs fi)} \subseteq \{0 \dots p\})$
 <proof>

definition *finite-field-factorization-int* :: int \Rightarrow int poly \Rightarrow int \times int poly list **where**
finite-field-factorization-int p = (
 if p \leq 65535
 then *finite-field-factorization-main* p (finite-field-ops32 (uint32-of-int p))
 else if p \leq 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-wint32* = *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-wint64* = *prime-field-gen.finite-field-factorization-main*
 [*OF prime-field.prime-field-finite-field-ops64*, *unfolded prime-field-def mod-ring-locale-def*,
unfolded poly-mod-type-simps, *internalize-sort 'a :: prime-card*, *OF type-to-set*,
unfolded remove-duplicate-premise, *cancel-type-definition*, *OF non-empty*]

lemma *finite-field-factorization-int*:

assumes *sq*: *poly-mod.square-free-m p f*

and *result*: *finite-field-factorization-int p f = (c,fs)*

shows *poly-mod.unique-factorization-m p f (c, mset fs)*

$\wedge c \in \{0 \dots p\}$

$\wedge (\forall fi \in \text{set } fs. \text{set } (\text{coeffs } fi) \subseteq \{0 \dots p\})$

<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*: $B = 0 \vee \text{degree } B < \text{degree } f$ $B' = 0 \vee \text{degree } B' < \text{degree } f$

and *f*: $f \neq 0$ **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 $f = g * q + r$ $r = 0 \vee \text{degree } r < \text{degree } g$
 ⟨proof⟩

definition *dupe-monic* :: 'a :: comm-ring-1 poly \Rightarrow 'a poly \Rightarrow 'a poly \Rightarrow 'a poly
 \Rightarrow 'a poly \Rightarrow
 'a poly * 'a poly **where**
dupe-monic D H S T U = (case *pdivmod-monic* (T * U) D of (q,r) \Rightarrow
 (S * U + H * q, r))

lemma *dupe-monic*: **assumes** 1: $D*S + H*T = 1$
and mon: monic D
and dupe: *dupe-monic* D H S T U = (A,B)
shows $A * D + B * H = U$ $B = 0 \vee \text{degree } B < \text{degree } D$
 ⟨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: $A' * D + B' * H = U$ $B' = 0 \vee \text{degree } B' < \text{degree } D$
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* \implies *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*: $A = \text{to-int-poly-i ff-ops } a$
and *B*: $B = \text{to-int-poly-i ff-ops } b$
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
 $A * D + B * H = m\ U$
 $B = 0 \vee \text{degree } B < \text{degree } D$
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
 $A * D + B * H = m\ U$
 $B = 0 \vee \text{degree } B < \text{degree } D$
Mp A = A
Mp B = B
 ⟨proof⟩

end

definition *dupe-monic-dynamic*

:: $\text{int} \Rightarrow \text{int poly} \Rightarrow \text{int poly} \Rightarrow \text{int poly} \Rightarrow \text{int poly} \Rightarrow \text{int poly} \Rightarrow \text{int poly} \times \text{int poly}$ **where**

dupe-monic-dynamic $p = ($
 if $p \leq 65535$
 then *dupe-monic-i-int* (*finite-field-ops32* (*uint32-of-int* p))
 else if $p \leq 4294967295$
 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

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

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

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

<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

$A * D + B * H = m \ U$

$B = 0 \vee \text{degree } B < \text{degree } D$

$Mp \ A = A$

$Mp \ B = B$

<proof>

end

context *poly-mod*

begin

definition *dupe-monic-int* :: *int poly* \Rightarrow *int poly* \Rightarrow *int poly* \Rightarrow *int poly* \Rightarrow *int poly*
 \Rightarrow

int poly * *int poly* **where**

*dupe-monic-int D H S T U = (case pdivmod-monic (Mp (T * U)) D of (q,r) \Rightarrow*
*(Mp (S * U + H * q), Mp r))*

end

declare *poly-mod.dupe-monic-int-def*[code]

Old direct proof on int poly. It does not permit to change implementation. This proof is still present, since we did not export the uniqueness part from the type-based uniqueness result $\llbracket ?D * ?S + ?H * ?T = 1; \text{monic } ?D; \text{dupe-monic } ?D ?H ?S ?T ?U = (?A, ?B); \text{comm-monoid-mult-class.coprime } ?D ?H; ?A' * ?D + ?B' * ?H = ?U; ?B' = 0 \vee \text{degree } ?B' < \text{degree } ?D \rrbracket \Longrightarrow ?A' = ?A$

$\llbracket ?D * ?S + ?H * ?T = 1; \text{monic } ?D; \text{dupe-monic } ?D ?H ?S ?T ?U = (?A, ?B); \text{comm-monoid-mult-class.coprime } ?D ?H; ?A' * ?D + ?B' * ?H = ?U; ?B' = 0 \vee \text{degree } ?B' < \text{degree } ?D \rrbracket \Longrightarrow ?B' = ?B$ via the various relations.

lemma (in *poly-mod-2*) *dupe-monic-int*: **assumes** *1*: $D * S + H * T = m \ 1$

and *mon*: *monic D*

and *dupe*: *dupe-monic-int D H S T U = (A,B)*

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

$\text{coprime-m } D \ H \Longrightarrow A' * D + B' * H = m \ U \Longrightarrow B' = 0 \vee \text{degree } B' < \text{degree } D \Longrightarrow Mp \ D = D$

$\Longrightarrow Mp \ A' = A' \Longrightarrow Mp \ B' = B' \Longrightarrow \text{prime } m$

$\Longrightarrow A' = A \wedge B' = B$

<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*: $a = \text{to-int-poly } A$
and *b*: $b = \text{to-int-poly } B$
shows $f * a + g * b = m \ 1$
<proof>

definition *bezout-coefficients-i* :: $'i$ *arith-ops-record* \Rightarrow $'i$ *list* \Rightarrow $'i$ *list* \Rightarrow $'i$ *list* \times
 $'i$ *list* **where**

bezout-coefficients-i ff-ops f g = fst (euclid-ext-poly-i ff-ops f g)

definition *euclid-ext-poly-mod-main* :: $int \Rightarrow 'a$ *arith-ops-record* \Rightarrow int *poly* \Rightarrow int
poly \Rightarrow int *poly* \times int *poly* **where**

euclid-ext-poly-mod-main p ff-ops f g = (case bezout-coefficients-i ff-ops (of-int-poly-i
ff-ops f) (of-int-poly-i ff-ops g) of
 $(a,b) \Rightarrow (\text{to-int-poly-i ff-ops } a, \text{to-int-poly-i ff-ops } b))$

definition *euclid-ext-poly-dynamic* :: $int \Rightarrow int$ *poly* \Rightarrow int *poly* \Rightarrow int *poly* \times int
poly **where**

euclid-ext-poly-dynamic p = (
if $p \leq 65535$
then *euclid-ext-poly-mod-main p (finite-field-ops32 (uint32-of-int p))*
else if $p \leq 4294967295$
then *euclid-ext-poly-mod-main p (finite-field-ops64 (uint64-of-int p))*
else *euclid-ext-poly-mod-main p (finite-field-ops-integer (integer-of-int p))*

context *prime-field-gen*

begin

lemma *bezout-coefficients-i[transfer-rule]*:

$(poly-rel \implies poly-rel \implies rel-prod poly-rel poly-rel)$
 $(bezout-coefficients-i ff-ops) \text{ bezout-coefficients}$
<proof>

lemma *bezout-coefficients-i-sound*: **assumes** *f*: $f' = \text{of-int-poly-i ff-ops } f \text{ Mp } f = f$

and *g*: $g' = \text{of-int-poly-i ff-ops } g \text{ Mp } g = g$
and *cop*: *coprime-m f g*
and *res*: *bezout-coefficients-i ff-ops f' g' = (a',b')*

and a : $a = \text{to-int-poly-i ff-ops } a'$
and b : $b = \text{to-int-poly-i ff-ops } b'$
shows $f * a + g * b = m \ 1$
 $Mp \ a = a \ Mp \ b = b$
 $\langle \text{proof} \rangle$

lemma *euclid-ext-poly-mod-main*: **assumes** $\text{cop: coprime-}m \ f \ g$
and f : $Mp \ f = f$ **and** g : $Mp \ g = g$
and $\text{res: euclid-ext-poly-mod-main } m \ \text{ff-ops } f \ g = (a, b)$
shows $f * a + g * b = m \ 1$
 $Mp \ a = a \ Mp \ b = b$
 $\langle \text{proof} \rangle$

end

context *poly-mod-prime* **begin**

lemmas *euclid-ext-poly-mod-integer* = *prime-field-gen.euclid-ext-poly-mod-main*
 $[OF \ \text{prime-field.prime-field-finite-field-ops-integer},$
 $\text{unfolded prime-field-def mod-ring-locale-def poly-mod-type-simps, internalize-sort}$
 $'a :: \text{prime-card}, OF \ \text{type-to-set, unfolded remove-duplicate-premise, cancel-type-definition},$
 $OF \ \text{non-empty}]$

lemmas *euclid-ext-poly-mod-uint32* = *prime-field-gen.euclid-ext-poly-mod-main*
 $[OF \ \text{prime-field.prime-field-finite-field-ops32},$
 $\text{unfolded prime-field-def mod-ring-locale-def poly-mod-type-simps, internalize-sort}$
 $'a :: \text{prime-card}, OF \ \text{type-to-set, unfolded remove-duplicate-premise, cancel-type-definition},$
 $OF \ \text{non-empty}]$

lemmas *euclid-ext-poly-mod-uint64* = *prime-field-gen.euclid-ext-poly-mod-main* $[OF$
 $\text{prime-field.prime-field-finite-field-ops64},$
 $\text{unfolded prime-field-def mod-ring-locale-def poly-mod-type-simps, internalize-sort}$
 $'a :: \text{prime-card}, OF \ \text{type-to-set, unfolded remove-duplicate-premise, cancel-type-definition},$
 $OF \ \text{non-empty}]$

lemma *euclid-ext-poly-dynamic*:
assumes $\text{cop: coprime-}m \ f \ g$ **and** f : $Mp \ f = f$ **and** g : $Mp \ g = g$
and $\text{res: euclid-ext-poly-dynamic } p \ f \ g = (a, b)$
shows $f * a + g * b = m \ 1$
 $Mp \ a = a \ Mp \ b = b$
 $\langle \text{proof} \rangle$

end

lemma *range-sum-prod*: **assumes** $xy: x \in \{0..<q\} \ (y :: \text{int}) \in \{0..<p\}$
shows $x + q * y \in \{0..<p * q\}$
 $\langle \text{proof} \rangle$

context

fixes $C :: \text{int poly}$
begin

context

fixes $p :: \text{int and } S T D1 H1 :: \text{int poly}$
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 ;
 $q = p^{\wedge} n$;
 $U = \text{poly-mod.Mp } p (\text{sdiv-poly } (C - D * H) q)$; — $H2 + H3$
 (*A*,*B*) = *poly-mod.dupe-monic-int* $p D1 H1 S T U$
 in ($D + \text{smult } q B, H + \text{smult } q A$) — $H4$
| *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* $p D1 = D1 \text{ poly-mod.Mp } p H1 = H1$

and *res*: *linear-hensel-main* $n = (D, H)$

and $n: n \neq 0$

and *prime*: *prime* $p - p > 1$ suffices if one does not need uniqueness

and *cop*: *poly-mod.coprime-m* $p D1 H1$

shows *poly-mod.eq-m* $(p^{\wedge} n) (D * H) C$

\wedge *monic* *D*

\wedge *poly-mod.eq-m* $p D D1 \wedge$ *poly-mod.eq-m* $p H H1$

\wedge *poly-mod.Mp* $(p^{\wedge} n) D = D$

\wedge *poly-mod.Mp* $(p^{\wedge} n) H = H \wedge$

 (*poly-mod.eq-m* $(p^{\wedge} n) (D' * H')$ $C \longrightarrow$

poly-mod.eq-m $p D' D1 \longrightarrow$

poly-mod.eq-m $p H' H1 \longrightarrow$

poly-mod.Mp $(p^{\wedge} n) D' = D' \longrightarrow$

poly-mod.Mp $(p^{\wedge} n) H' = H' \longrightarrow$ *monic* $D' \longrightarrow D' = D \wedge H' = H$)

 ⟨*proof*⟩

end

end

definition *linear-hensel-binary* :: $\text{int} \Rightarrow \text{nat} \Rightarrow \text{int poly} \Rightarrow \text{int poly} \Rightarrow \text{int poly} \Rightarrow$

$\text{int poly} \times \text{int poly}$ **where**

linear-hensel-binary $p n C D H = (\text{let}$

 (*S*,*T*) = *euclid-ext-poly-dynamic* $p D H$

 in *linear-hensel-main* $C p S T D H n$)

lemma (in *poly-mod-prime*) *unique-hensel-binary*:

assumes *prime*: *prime* p

and *cop*: *coprime-m* $D H$ **and** *eq*: *eq-m* $(D * H) C$

and *normalized-input*: $Mp\ D = D\ Mp\ H = H$
and *monic-input*: *monic* D
and $n: n \neq 0$
shows $\exists! (D', H')$. — D', H' are computed via *linear-hensel-binary*
 $poly-mod.eq-m\ (p \hat{n})\ (D' * H')\ C$ — the main result: equivalence mod $p \hat{n}$
 \wedge *monic* D' — monic output
 $\wedge eq-m\ D\ D' \wedge eq-m\ H\ H'$ — apply ‘*mod* p ’ on D' and H' yields D and H again
 $\wedge poly-mod.Mp\ (p \hat{n})\ D' = D' \wedge poly-mod.Mp\ (p \hat{n})\ H' = H'$ — output is
normalized
 $\langle proof \rangle$

context

fixes $C :: int\ poly$
begin

lemma *hensel-step-main*: **assumes**

$one-q: poly-mod.eq-m\ q\ (D * S + H * T)\ 1$
and $one-p: poly-mod.eq-m\ p\ (D1 * S1 + H1 * T1)\ 1$
and $CDHq: poly-mod.eq-m\ q\ C\ (D * H)$
and $D1D: poly-mod.eq-m\ p\ D1\ D$
and $H1H: poly-mod.eq-m\ p\ H1\ H$
and $S1S: poly-mod.eq-m\ p\ S1\ S$
and $T1T: poly-mod.eq-m\ p\ T1\ T$
and *mon*: *monic* D
and $mon1: monic\ D1$
and $q: q > 1$
and $p: p > 1$
and $D1: poly-mod.Mp\ p\ D1 = D1$
and $H1: poly-mod.Mp\ p\ H1 = H1$
and $S1: poly-mod.Mp\ p\ S1 = S1$
and $T1: poly-mod.Mp\ p\ T1 = T1$
and $D: poly-mod.Mp\ q\ D = D$
and $H: poly-mod.Mp\ q\ H = H$
and $S: poly-mod.Mp\ q\ S = S$
and $T: poly-mod.Mp\ q\ T = T$
and $U1: U1 = poly-mod.Mp\ p\ (sdiv-poly\ (C - D * H)\ q)$
and $dupe1: dupe-monic-dynamic\ p\ D1\ H1\ S1\ T1\ U1 = (A, B)$
and $D': D' = D + smult\ q\ B$
and $H': H' = H + smult\ q\ A$
and $U2: U2 = poly-mod.Mp\ q\ (sdiv-poly\ (S * D' + T * H' - 1)\ p)$
and $dupe2: dupe-monic-dynamic\ q\ D\ H\ S\ T\ U2 = (A', B')$
and $rq: r = p * q$
and $pq: p\ dvd\ q$
and $S': S' = poly-mod.Mp\ r\ (S - smult\ p\ A')$
and $T': T' = poly-mod.Mp\ r\ (T - smult\ p\ B')$
shows $poly-mod.eq-m\ r\ C\ (D' * H')$
 $poly-mod.Mp\ r\ D' = D'$
 $poly-mod.Mp\ r\ H' = H'$

poly-mod.Mp r S' = S'
poly-mod.Mp r T' = T'
*poly-mod.eq-m r (D' * S' + H' * T') 1*
monic D'
 ⟨proof⟩

definition *hensel-step where*

hensel-step p q S1 T1 D1 H1 S T D H = (
*let U = poly-mod.Mp p (sdiv-poly (C - D * H) q); — Z2 and Z3*
(A,B) = dupe-monic-dynamic p D1 H1 S1 T1 U;
D' = D + smult q B; — Z4
H' = H + smult q A;
*U' = poly-mod.Mp q (sdiv-poly (S*D' + T*H' - 1) p); — Z5 + Z6*
(A',B') = dupe-monic-dynamic q D H S T U';
*q' = p * q;*
S' = poly-mod.Mp q' (S - smult p A'); — Z7
T' = poly-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-mod.Mp q (sdiv-poly (C - D * H) q);*
(A, B) = dupe U;
D' = D + Polynomial.smult q B;
H' = H + Polynomial.smult q A;
*U' = poly-mod.Mp q (sdiv-poly (S * D' + T * H' - 1) q);*
(A', B') = dupe U';
*q' = q * q;*
S' = poly-mod.Mp q' (S - Polynomial.smult q A');
T' = poly-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-mod.Mp q (sdiv-poly (C - D * H) q); — Z2 + Z3*
(A,B) = dupe-monic-dynamic q D H S T U;
D' = D + smult q B; — Z4
H' = H + smult q A
in (D',H'))

lemma *hensel-step: assumes step: hensel-step p q S1 T1 D1 H1 S T D H = (S', T', D', H')*

and *one-p: poly-mod.eq-m p (D1 * S1 + H1 * T1) 1*

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

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]: ¬ j ≤ 1 ⇒ odd j ⇒ Suc (j div 2)
< j ⟨proof⟩

```

fun quadratic-hensel-loop **where**

```

quadratic-hensel-loop (j :: nat) = (
  if j ≤ 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) ⇒
      let qq = q * q in
      (case quadratic-hensel-step q S T D H of — quadratic step
        (S', T', D', H') ⇒ (qq, S', T', D', H')))
    else — odd j
      (case quadratic-hensel-loop (j div 2 + 1) of
        (q, S, T, D, H) ⇒
        (case quadratic-hensel-step q S T D H of — quadratic step
          (S', T', D', H') ⇒
          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) ⇒ (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 ≤ 1 then (D1, H1)
  else if even j
  then (case quadratic-hensel-loop (j div 2) of
    (q, S, T, D, H) ⇒
    simple-quadratic-hensel-step q S T D H)
  else (case quadratic-hensel-loop (j div 2 + 1) of
    (q, S, T, D, H) ⇒
    (case simple-quadratic-hensel-step q S T D H of

```

$(D', H') \Rightarrow \text{let } \text{down} = \text{poly-mod.Mp } (q * q \text{ div } p) \text{ in } (\text{down } D', \text{down } H')\rangle\rangle\rangle\rangle$
 ⟨proof⟩

context

fixes $j :: \text{nat}$
assumes $1: \text{poly-mod.eq-m } p (D1 * S1 + H1 * T1) 1$
and $CDH1: \text{poly-mod.eq-m } p C (D1 * H1)$
and $\text{mon1: monic } D1$
and $p: p > 1$
and $D1: \text{poly-mod.Mp } p D1 = D1$
and $H1: \text{poly-mod.Mp } p H1 = H1$
and $S1: \text{poly-mod.Mp } p S1 = S1$
and $T1: \text{poly-mod.Mp } p T1 = T1$
and $j: j \geq 1$
begin

lemma *quadratic-hensel-loop*:

assumes *quadratic-hensel-loop* $j = (q, S, T, D, H)$
shows $(\text{poly-mod.eq-m } q C (D * H) \wedge \text{monic } D$
 $\wedge \text{poly-mod.eq-m } p D1 D \wedge \text{poly-mod.eq-m } p H1 H$
 $\wedge \text{poly-mod.eq-m } q (D * S + H * T) 1$
 $\wedge \text{poly-mod.Mp } q D = D \wedge \text{poly-mod.Mp } q H = H$
 $\wedge \text{poly-mod.Mp } q S = S \wedge \text{poly-mod.Mp } q T = T$
 $\wedge q = p^{\wedge}j)$
 ⟨proof⟩

lemma *quadratic-hensel-main*: **assumes** *res: quadratic-hensel-main* $j = (D, H)$

shows $\text{poly-mod.eq-m } (p^{\wedge}j) C (D * H)$
 $\text{monic } D$
 $\text{poly-mod.eq-m } p D1 D$
 $\text{poly-mod.eq-m } p H1 H$
 $\text{poly-mod.Mp } (p^{\wedge}j) D = D$
 $\text{poly-mod.Mp } (p^{\wedge}j) H = H$
 ⟨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* \Rightarrow *'a* **where**

$\text{factor-node-info } (\text{Factor-Leaf } i x) = i$
 $|\text{factor-node-info } (\text{Factor-Node } i l r) = i$

fun *factors-of-factor-tree* :: *'a factor-tree* \Rightarrow *int poly multiset* **where**

$\text{factors-of-factor-tree } (\text{Factor-Leaf } i x) = \{\#x\# \}$

| *factors-of-factor-tree* (*Factor-Node* *i l r*) = *factors-of-factor-tree* *l* + *factors-of-factor-tree* *r*

fun *product-factor-tree* :: *int* \Rightarrow '*a* *factor-tree* \Rightarrow *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* \Rightarrow '*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* \cup *sub-trees* *r*)

lemma *sub-trees-refl[simp]*: *t* \in *sub-trees* *t* *<proof>*

lemma *product-factor-tree*: **assumes** $\bigwedge x. x \in \#$ *factors-of-factor-tree* *t* \Longrightarrow *poly-mod.Mp* *p* *x* = *x*
shows *u* \in *sub-trees* (*product-factor-tree* *p* *t*) \Longrightarrow *factor-node-info* *u* = *f* \Longrightarrow
poly-mod.Mp *p* *f* = *f* \wedge *f* = *poly-mod.Mp* *p* (*prod-mset* (*factors-of-factor-tree* *u*))
 \wedge
factors-of-factor-tree (*product-factor-tree* *p* *t*) = *factors-of-factor-tree* *t*
<proof>

fun *create-factor-tree-simple* :: *int* *poly list* \Rightarrow *unit factor-tree* **where**
create-factor-tree-simple *xs* = (let *n* = *length* *xs* in if *n* \leq 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* \neq [] \Longrightarrow *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* \Rightarrow ('*a* \times *nat*) *list* \Rightarrow ('*a* \times *nat*) *list* \times ('*a* \times *nat*) *list* **where**
partition-factors-main *s* [] = ([], [])

| *partition-factors-main* $s ((f,d) \# xs) = (if\ d \leq s\ then\ case\ partition-factors-main\ (s - d)\ xs\ of$
 $(l,r) \Rightarrow ((f,d) \# l, r)$ *else case partition-factors-main* $d\ xs\ of$
 $(l,r) \Rightarrow (l, (f,d) \# r)$)

lemma *partition-factors-main*: *partition-factors-main* $s\ xs = (a,b) \implies mset\ xs = mset\ a + mset\ b$
 ⟨*proof*⟩

definition *partition-factors* :: $('a \times nat)\ list \Rightarrow ('a \times nat)\ list \times ('a \times nat)\ list$
where

partition-factors $xs = (let\ n = sum-list\ (map\ snd\ xs)\ div\ 2\ in$
 $case\ partition-factors-main\ n\ xs\ of$
 $([], x \# y \# ys) \Rightarrow ([x], y \# ys)$
 | $(x \# y \# ys, []) \Rightarrow ([x], y \# ys)$
 | *pair* $\Rightarrow pair$)

lemma *partition-factors*: *partition-factors* $xs = (a,b) \implies mset\ xs = mset\ a + mset\ b$
 ⟨*proof*⟩

lemma *partition-factors-length*: **assumes** $\neg\ length\ xs \leq 1\ (a,b) = partition-factors\ xs$

shows [*termination-simp*]: $length\ a < length\ xs\ length\ b < length\ xs$ **and** $a \neq []$
 $b \neq []$
 ⟨*proof*⟩

fun *create-factor-tree-balanced* :: $(int\ poly \times nat)\ list \Rightarrow unit\ factor-tree$ **where**
create-factor-tree-balanced $xs = (if\ length\ xs \leq 1\ then\ Factor-Leaf\ ()\ (fst\ (hd\ xs)))$
else
 $case\ partition-factors\ xs\ of\ (l,r) \Rightarrow Factor-Node\ ()$
 $(create-factor-tree-balanced\ l)$
 $(create-factor-tree-balanced\ r)$)

definition *create-factor-tree* :: $int\ poly\ list \Rightarrow unit\ factor-tree$ **where**

create-factor-tree $xs = (let\ ys = map\ (\lambda\ f.\ (f,\ degree\ f))\ xs;$
 $zs = rev\ (sort-key\ snd\ ys)$
 in *create-factor-tree-balanced* $zs)$

lemma *create-factor-tree-balanced*: $xs \neq [] \implies factors-of-factor-tree\ (create-factor-tree-balanced\ xs) = mset\ (map\ fst\ xs)$
 ⟨*proof*⟩

lemma *create-factor-tree*: **assumes** $xs \neq []$

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* \Rightarrow *int poly* \Rightarrow *int poly* \Rightarrow *int poly* \times *int poly* **where**

quadratic-hensel-binary *C D H* = (
 case euclid-ext-poly-dynamic *p D H* of
 (*S,T*) \Rightarrow *quadratic-hensel-main* *C p S T D H n*)

fun *hensel-lifting-main* :: *int poly* \Rightarrow *int poly factor-tree* \Rightarrow *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* \Rightarrow *int poly list* \Rightarrow *int poly list* **where**

hensel-lifting-monic *u vs* = (if *vs* = [] then [] else let
 pn = $p^{\wedge}n$;
 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* \Rightarrow *int poly list* \Rightarrow *int poly list* **where**

hensel-lifting *f gs* = (let *lc* = *lead-coeff* *f*;
 ilc = *inverse-mod* *lc (p^{\wedge}n)*;
 g = *smult* *ilc f*
 in *hensel-lifting-monic* *g gs*)

end

context *poly-mod-prime* **begin**

context

fixes *n* :: *nat*
 assumes *n*: *n* \neq 0

begin

abbreviation *hensel-binary* \equiv *quadratic-hensel-binary* *p n*

abbreviation *hensel-main* \equiv *hensel-lifting-main* *p n*

lemma *hensel-binary*:

assumes *cop*: *coprime-m* *D H* **and** *eq*: *eq-m* *C (D * H)*

and *normalized-input*: *Mp* *D* = *D Mp* *H* = *H*

and *monic-input*: *monic* *D*

and *hensel-result*: *hensel-binary* *C D H* = (*D',H'*)

shows *poly-mod.eq-m* ($p^{\wedge}n$) *C (D' * H')* — the main result: equivalence mod

$p^{\widehat{n}}$
 \wedge *monic* D' — monic output
 \wedge *eq-m* $D D' \wedge$ *eq-m* $H H'$ — apply ‘*mod p*’ on D' and H' yields D and H again
 \wedge *poly-mod.Mp* $(p^{\widehat{n}}) D' = D' \wedge$ *poly-mod.Mp* $(p^{\widehat{n}}) H' = H'$ — output is normalized
 ⟨*proof*⟩

lemma *hensel-main*:

assumes *eq*: *eq-m* C (*prod-mset* (*factors-of-factor-tree* Fs))
and $\bigwedge F. F \in \#$ *factors-of-factor-tree* $Fs \implies Mp F = F \wedge$ *monic* F
and *hensel-result*: *hensel-main* $C Fs = Gs$
and C : *monic* C *poly-mod.Mp* $(p^{\widehat{n}}) C = C$
and *sf*: *square-free-m* C
and $\bigwedge f t. t \in$ *sub-trees* $Fs \implies$ *factor-node-info* $t = f \implies f = Mp$ (*prod-mset* (*factors-of-factor-tree* t))
shows *poly-mod.eq-m* $(p^{\widehat{n}}) C$ (*prod-list* Gs) — the main result: equivalence mod $p^{\widehat{n}}$
 \wedge *factors-of-factor-tree* $Fs =$ *mset* (*map* Mp Gs)
 \wedge ($\forall G. G \in$ *set* $Gs \longrightarrow$ *monic* $G \wedge$ *poly-mod.Mp* $(p^{\widehat{n}}) G = G$)
 ⟨*proof*⟩

lemma *hensel-lifting-monic*:

assumes *eq*: *poly-mod.eq-m* $p C$ (*prod-list* Fs)
and Fs : $\bigwedge F. F \in$ *set* $Fs \implies$ *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* $(p^{\widehat{n}}) C$)
and *sf*: *poly-mod.square-free-m* $p C$
shows *poly-mod.eq-m* $(p^{\widehat{n}}) C$ (*prod-list* Gs)
 \wedge *mset* (*map* (*poly-mod.Mp* p) Gs) = *mset* Fs
 $G \in$ *set* $Gs \implies$ *monic* $G \wedge$ *poly-mod.Mp* $(p^{\widehat{n}}) G = G$
 ⟨*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* $(p^{\widehat{n}}) f$ (*lead-coeff* $f, mset gs$) — factorization
mod p
 $sort$ (*map* *degree* fs) = $sort$ (*map* *degree* gs) — degrees stay the
 same
 $\bigwedge g. g \in$ *set* $gs \implies$ *monic* $g \wedge$ *poly-mod.Mp* $(p^{\widehat{n}}) g = g \wedge$ — 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*:

$\text{degree } (\text{smult } a \ p) = \text{degree } p \iff \text{degree } p = 0 \vee a * \text{lead-coeff } p \neq 0$

<proof>

lemma *degree-smult-eqI[intro!]*:

assumes $\text{degree } p \neq 0 \implies a * \text{lead-coeff } p \neq 0$

shows $\text{degree } (\text{smult } a \ p) = \text{degree } p$

<proof>

lemma *degree-mult-eq2*:

assumes $lc: \text{lead-coeff } p * \text{lead-coeff } q \neq 0$

shows $\text{degree } (p * q) = \text{degree } p + \text{degree } q$ (**is - = ?r**)

<proof>

lemma *degree-mult-eq-left-unit*:

fixes $p \ q :: 'a :: \text{comm-semiring-1} \ \text{poly}$

assumes *unit*: $\text{lead-coeff } p \ \text{dvd } 1$ **and** $q0: q \neq 0$

shows $\text{degree } (p * q) = \text{degree } p + \text{degree } q$

<proof>

context *ring-hom* **begin**

lemma *monic-degree-map-poly-hom*: $\text{monic } p \implies \text{degree } (\text{map-poly } \text{hom } p) = \text{degree } p$

<proof>

lemma *monic-map-poly-hom*: $\text{monic } p \implies \text{monic } (\text{map-poly } \text{hom } p)$

<proof>

end

lemma *of-nat-zero*:

assumes $\text{CARD}('a :: \text{nontriv}) \ \text{dvd } n$

shows $(\text{of-nat } n :: 'a \ \text{mod-ring}) = 0$

<proof>

abbreviation *rebase* :: $'a :: \text{nontriv mod-ring} \implies 'b :: \text{nontriv mod-ring} \ (@- [100]100)$

where $@x \equiv \text{of-int } (\text{to-int-mod-ring } x)$

abbreviation $\text{rebase-poly} :: 'a :: \text{nontriv mod-ring poly} \Rightarrow 'b :: \text{nontriv mod-ring poly} \ (\#- \ [100]100)$
where $\#x \equiv \text{of-int-poly } (\text{to-int-poly } x)$

lemma $\text{rebase-self} \ [simp]:$
 $@x = x$
 $\langle \text{proof} \rangle$

lemma $\text{map-poly-rebase} \ [simp]:$
 $\text{map-poly rebase } p = \#p$
 $\langle \text{proof} \rangle$

lemma $\text{rebase-poly-0}: \#0 = 0$
 $\langle \text{proof} \rangle$

lemma $\text{rebase-poly-1}: \#1 = 1$
 $\langle \text{proof} \rangle$

lemma $\text{rebase-poly-pCons} [simp]: \#pCons \ a \ p = pCons \ (@a) \ (\#p)$
 $\langle \text{proof} \rangle$

lemma $\text{rebase-poly-self} [simp]: \#p = p \ \langle \text{proof} \rangle$

lemma $\text{degree-rebase-poly-le}: \text{degree } (\#p) \leq \text{degree } p$
 $\langle \text{proof} \rangle$

lemma(**in** comm-ring-hom) $\text{degree-map-poly-unit}: \text{assumes } \text{lead-coeff } p \ \text{dvd } 1$
shows $\text{degree } (\text{map-poly hom } p) = \text{degree } p$
 $\langle \text{proof} \rangle$

lemma $\text{rebase-poly-eq-0-iff}:$
 $(\#p :: 'a :: \text{nontriv mod-ring poly}) = 0 \iff (\forall i. (@coeff \ p \ i :: 'a \ \text{mod-ring}) = 0)$ (**is** $?l \iff ?r$)
 $\langle \text{proof} \rangle$

lemma $\text{mod-mod-le}:$
assumes $ab: (a::\text{int}) \leq b$ **and** $a0: 0 < a$ **and** $c0: c \geq 0$ **shows** $(c \ \text{mod } a) \ \text{mod } b = c \ \text{mod } a$
 $\langle \text{proof} \rangle$

locale $\text{rebase-ge} =$
fixes $ty1 :: 'a :: \text{nontriv itself}$ **and** $ty2 :: 'b :: \text{nontriv itself}$
assumes $\text{card}: \text{CARD}('a) \leq \text{CARD}('b)$
begin

lemma $ab: \text{int } \text{CARD}('a) \leq \text{CARD}('b) \ \langle \text{proof} \rangle$

lemma *rebase-eq-0[simp]*:
shows $(@x :: 'a \text{ mod-ring}) :: 'b \text{ mod-ring} = 0 \longleftrightarrow x = 0$
 $\langle \text{proof} \rangle$

lemma *degree-rebase-poly-eq[simp]*:
shows $\text{degree } (\#(p :: 'a \text{ mod-ring poly}) :: 'b \text{ mod-ring poly}) = \text{degree } p$
 $\langle \text{proof} \rangle$

lemma *lead-coeff-rebase-poly[simp]*:
lead-coeff $(\#(p :: 'a \text{ mod-ring poly}) :: 'b \text{ mod-ring poly}) = @\text{lead-coeff } p$
 $\langle \text{proof} \rangle$

lemma *to-int-mod-ring-rebase*: $\text{to-int-mod-ring}(@x :: 'a \text{ mod-ring}) :: 'b \text{ mod-ring}$
 $= \text{to-int-mod-ring } x$
 $\langle \text{proof} \rangle$

lemma *rebase-id[simp]*: $@(@x :: 'a \text{ mod-ring}) :: 'b \text{ mod-ring} = @x$
 $\langle \text{proof} \rangle$

lemma *rebase-poly-id[simp]*: $\#(\#(p :: 'a \text{ mod-ring poly}) :: 'b \text{ mod-ring poly}) = \#p$
 $\langle \text{proof} \rangle$

end

locale *rebase-dvd* =
fixes $ty1 :: 'a :: \text{nontriv itself}$ **and** $ty2 :: 'b :: \text{nontriv itself}$
assumes $\text{dvd}: \text{CARD}('b) \text{ dvd } \text{CARD}('a)$
begin

lemma *ab*: $\text{CARD}('a) \geq \text{CARD}('b)$ $\langle \text{proof} \rangle$

lemma *rebase-id[simp]*: $@(@x :: 'b \text{ mod-ring}) :: 'a \text{ mod-ring} = x$ $\langle \text{proof} \rangle$

lemma *rebase-poly-id[simp]*: $\#(\#(p :: 'b \text{ mod-ring poly}) :: 'a \text{ mod-ring poly}) = p$
 $\langle \text{proof} \rangle$

lemma *rebase-of-nat[simp]*: $(@(\text{of-nat } n :: 'a \text{ mod-ring}) :: 'b \text{ mod-ring}) = \text{of-nat } n$
 $\langle \text{proof} \rangle$

lemma *mod-1-lift-nat*:
assumes $(\text{of-int } (\text{int } x) :: 'a \text{ mod-ring}) = 1$
shows $(\text{of-int } (\text{int } x) :: 'b \text{ mod-ring}) = 1$
 $\langle \text{proof} \rangle$

sublocale *comm-ring-hom rebase* :: $'a \text{ mod-ring} \Rightarrow 'b \text{ mod-ring}$
 $\langle \text{proof} \rangle$

lemma *of-nat-CARD-eq-0[simp]*: $(\text{of-nat } \text{CARD}('a) :: 'b \text{ mod-ring}) = 0$

$\langle \text{proof} \rangle$

interpretation *map-poly-hom*: *map-poly-comm-ring-hom rebase* :: 'a mod-ring \Rightarrow 'b mod-ring $\langle \text{proof} \rangle$

sublocale *poly*: *comm-ring-hom rebase-poly* :: 'a mod-ring poly \Rightarrow 'b mod-ring poly $\langle \text{proof} \rangle$

lemma *poly-rebase[simp]*: $\text{@poly } p \ x = \text{poly } (\#(p :: 'a \text{ mod-ring poly}) :: 'b \text{ mod-ring poly})$ ($\text{@}(x :: 'a \text{ mod-ring}) :: 'b \text{ mod-ring}$) $\langle \text{proof} \rangle$

lemma *rebase-poly-smult[simp]*: $(\#(\text{smult } a \ p :: 'a \text{ mod-ring poly}) :: 'b \text{ mod-ring poly}) = \text{smult } (\text{@}a) (\#p)$ $\langle \text{proof} \rangle$

end

locale *rebase-mult* =
fixes *ty1* :: 'a :: nontriv itself
and *ty2* :: 'b :: nontriv itself
and *ty3* :: 'd :: nontriv itself
assumes *d*: $\text{CARD}('a) = \text{CARD}('b) * \text{CARD}('d)$
begin

sublocale *rebase-dvd* *ty1 ty2* $\langle \text{proof} \rangle$

lemma *rebase-mult-eq[simp]*: $(\text{of-nat } \text{CARD}('d) * a :: 'a \text{ mod-ring}) = \text{of-nat } \text{CARD}('d) * a'$ \longleftrightarrow ($\text{@}a :: 'b \text{ mod-ring}$) = $\text{@}a'$ $\langle \text{proof} \rangle$

lemma *rebase-poly-smult-eq[simp]*:
fixes *a a'* :: 'a mod-ring poly
defines *d* \equiv *of-nat* $\text{CARD}('d) :: 'a \text{ mod-ring}$
shows $\text{smult } d \ a = \text{smult } d \ a' \longleftrightarrow (\#a :: 'b \text{ mod-ring poly}) = \#a'$ (**is** ?l \longleftrightarrow ?r) $\langle \text{proof} \rangle$

lemma *rebase-eq-0-imp-ex-mult*:
 $(\text{@}(a :: 'a \text{ mod-ring}) :: 'b \text{ mod-ring}) = 0 \implies (\exists c :: 'd \text{ mod-ring. } a = \text{of-nat } \text{CARD}('b) * \text{@}c)$ (**is** ?l \implies ?r) $\langle \text{proof} \rangle$

lemma *rebase-poly-eq-0-imp-ex-smult*:
 $(\#(p :: 'a \text{ mod-ring poly}) :: 'b \text{ mod-ring poly}) = 0 \implies$
 $(\exists p' :: 'd \text{ mod-ring poly. } (p = 0 \longleftrightarrow p' = 0) \wedge \text{degree } p' \leq \text{degree } p \wedge p = \text{smult } (\text{of-nat } \text{CARD}('b)) (\#p'))$
(is ?l \implies ?r) $\langle \text{proof} \rangle$

end

lemma *mod-mod-nat[simp]*: $a \bmod b \bmod (b * c :: nat) = a \bmod b$ *<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*: $CARD('p) \text{ dvd } CARD('q)$

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* \equiv *if* (*ty-p* :: 'p *itself*) = *ty-p* *then* $CARD('p)$ *else* *undefined*

lemma *p[simp]*: $p \equiv CARD('p)$ *<proof>*

definition *q* \equiv *if* (*ty-q* :: 'q *itself*) = *ty-q* *then* $CARD('q)$ *else* *undefined*

lemma *q[simp]*: $q \equiv CARD('q)$ *<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]*: $of\ nat\ q :: 'q\ mod\ ring \equiv 0$ *<proof>*

lemma *rebase-rebase[simp]*: $(@(@(x::'pq\ mod\ ring) :: 'q\ mod\ ring) :: 'p\ mod\ ring)$

$= @x$

<proof>

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

end

definition *dupe-monic* **where**

dupe-monic $D H S T U = (\text{case } p\text{divmod-monic } (T * U) D \text{ of } (q,r) \Rightarrow (S * U + H * q, r))$

lemma *dupe-monic*:

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

assumes $1: D*S + H*T = 1$

and *mon*: *monic* D

and *dupe*: *dupe-monic* $D H S T U = (A,B)$

shows $A * D + B * H = U \ B = 0 \vee \text{degree } B < \text{degree } D$

coprime $D H \Rightarrow A' * D + B' * H = U \Rightarrow B' = 0 \vee \text{degree } B' < \text{degree } D$
 $\Rightarrow A' = A \wedge B' = B$

⟨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 $a \ b :: 'p \text{ mod-ring poly}$ **and** $u :: 'pq \text{ mod-ring poly}$ **and** $v \ w :: 'q \text{ mod-ring poly}$

assumes *uvw*: $(\#u :: 'q \text{ mod-ring poly}) = v * w$

and *degu*: $\text{degree } u = \text{degree } v + \text{degree } w$

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

and *monic-v*: *monic* v

and *bv*: $\text{degree } b < \text{degree } v$

begin

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

⟨proof⟩

lemma *u0*: $u \neq 0$ ⟨proof⟩

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

⟨proof⟩

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

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

⟨proof⟩

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

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

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'*: $\text{degree } v' < \text{degree } v$ ⟨proof⟩

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

lemma *degw'*: $\text{degree } w' \leq \text{degree } w$
 ⟨proof⟩

abbreviation $qv' \equiv \text{smult } (\text{of-nat } q) (\#v') :: 'pq \text{ mod-ring poly}$

abbreviation $qw' \equiv \text{smult } (\text{of-nat } q) (\#w') :: 'pq \text{ mod-ring poly}$

abbreviation $V \equiv \#v + qv'$

abbreviation $W \equiv \#w + qw'$

lemma *vV*: $v = \#V$ ⟨proof⟩

lemma *wW*: $w = \#W$ ⟨proof⟩

lemma *uVW*: $u = V * W$
 ⟨proof⟩

lemma *degV*: $\text{degree } V = \text{degree } v$

and *lcV*: $\text{lead-coeff } V = @\text{lead-coeff } v$

and *degW*: $\text{degree } W = \text{degree } w$

⟨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* ($\#v :: 'p \text{ mod-ring poly}$) ($\#w$)

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* $v'' < \text{degree } v$
shows $v'' = v' \wedge w'' = w'$
<proof>

lemma *unique*:

assumes *vV2*: $v = \#V2$ **and** *wW2*: $w = \#W2$ **and** *uVW2*: $u = V2 * W2$
and *degV2*: *degree* $V2 = \text{degree } v$ **and** *degW2*: *degree* $W2 = \text{degree } w$
and *lc*: *lead-coeff* $V2 = @\text{lead-coeff } v$
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) \equiv
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 *uw*: $\#u = v * w$
and *degv*: *degree* $u = \text{degree } v + \text{degree } w$
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 $u = V' * W' \wedge v = \#V' \wedge w = \#W' \wedge \text{degree } V' = \text{degree } v \wedge \text{degree } W' = \text{degree } w \wedge$
 $\text{monic } V' \wedge \text{coprime } (\#V' :: 'p \text{ mod-ring poly}) (\#W')$ (**is** ?main)
and $(\forall V'' W''. u = V'' * W'' \longrightarrow v = \#V'' \longrightarrow w = \#W'' \longrightarrow$
 $\text{degree } V'' = \text{degree } v \longrightarrow \text{degree } W'' = \text{degree } w \longrightarrow \text{lead-coeff } V'' =$
 $@\text{lead-coeff } v \longrightarrow$
 $V'' = V' \wedge W'' = W')$ (**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* \Rightarrow *nat* \Rightarrow *int poly* \Rightarrow *int poly list* **where**

berlekamp-hensel p n f = (*case finite-field-factorization-int p f of*
(-,fs) \Rightarrow *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*: $n \neq 0$

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 $\bigwedge g. g \in \text{set } gs \implies \text{monic } g \wedge \text{poly-mod.Mp } (p^n) g = g \wedge$ — monic and normalized

poly-mod.irreducible-m p g \wedge — 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*: $n \neq 0$

shows *poly-mod.factorization-m (p ^ n) f (lead-coeff f, mset gs)* — factorization mod p^n

and $\bigwedge g. g \in \text{set } gs \implies \text{poly-mod.Mp } (p \hat{=} n) g = g \wedge \text{poly-mod.irreducible-m } p$
 g
— normalized and *irreducible* even mod p
 $\langle \text{proof} \rangle$

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)
 $\langle \text{proof} \rangle$

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* $(p \hat{=} n) c \in \{1 ..< p \hat{=} n\}$
 $\langle \text{proof} \rangle$

context *poly-mod-2*
begin

context
fixes $p :: \text{int}$
assumes *prime*: *prime* p
begin

interpretation p : *poly-mod-prime* p $\langle \text{proof} \rangle$

lemma *coprime-lead-coeff-factor*: **assumes** *coprime* (lead-coeff $(f * g)$) p
shows *coprime* (lead-coeff f) p *coprime* (lead-coeff g) p
 $\langle \text{proof} \rangle$

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 $\exists fs\ gs. \text{unique-factorization-m } f \text{ (lead-coeff } f, fs)$
 $\wedge \text{unique-factorization-m } g \text{ (lead-coeff } g, gs)$
 $\wedge Mf (c,hs) = Mf \text{ (lead-coeff } f * \text{lead-coeff } g, fs + gs)$
 $\wedge \text{image-mset } Mp\ fs = fs \wedge \text{image-mset } Mp\ gs = gs$
 $\langle \text{proof} \rangle$

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 $F \implies$ *unique-factorization-m* f F
and *factorization-m* g $G \implies$ *unique-factorization-m* g G
<proof>

end

lemma *monic-Mp-prod-mset*: **assumes** *fs*: $\bigwedge f. f \in \# fs \implies$ *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 $f \neq 0$
shows *degree-m* f = *sum-mset* (*image-mset* *degree-m* *fs*)
<proof>

lemma *degree-m-mult-le*: *degree-m* ($f * g$) \leq *degree-m* f + *degree-m* g
<proof>

lemma *degree-m-prod-mset-le*: *degree-m* (*prod-mset* *fs*) \leq *sum-mset* (*image-mset* *degree-m* *fs*)
<proof>

end

context *poly-mod-prime*
begin

lemma *unique-factorization-m-factor-partition*: **assumes** *l0*: $l \neq 0$
and *uf*: *poly-mod.unique-factorization-m* ($p \hat{\ } l$) 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* ($\lambda gi. gi$ *dvd* m $f1$) *gs* = (*gs1*, *gs2*)
shows *poly-mod.unique-factorization-m* ($p \hat{\ } l$) $f1$ (*lead-coeff* $f1$, *mset* *gs1*)
poly-mod.unique-factorization-m ($p \hat{\ } l$) $f2$ (*lead-coeff* $f2$, *mset* *gs2*)
<proof>

lemma *factorization-pn-to-factorization-p*: **assumes** *fact*: *poly-mod.factorization-m* ($p \hat{\ } n$) C (*c,fs*)

and *sf*: square-free-m *C*
and *n*: $n \neq 0$
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*: $n \neq 0$
shows \exists *Gs*. poly-mod.unique-factorization-m ($p^{\wedge}n$) *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* *f* = *gs*
and *n*: $n \neq 0$
shows poly-mod.unique-factorization-m ($p^{\wedge}n$) *f* (lead-coeff *f*, *mset* *gs*) — unique factorization mod $p^{\wedge}n$
 \bigwedge *g*. *g* ∈ *set* *gs* \implies poly-mod.Mp ($p^{\wedge}n$) *g* = *g* — normalized
 ⟨*proof*⟩

lemma hensel-lifting-unique:
assumes *n*: $n \neq 0$
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*: $c \in \{0..<p\}$
and *norm*: $(\forall fi \in \text{set } fs. \text{set } (\text{coeffs } fi) \subseteq \{0..<p\})$
shows poly-mod.unique-factorization-m ($p^{\wedge}n$) *f* (lead-coeff *f*, *mset* *gs*) — unique factorization mod $p^{\wedge}n$
 $\text{sort } (\text{map } \text{degree } fs) = \text{sort } (\text{map } \text{degree } gs)$ — degrees stay the same
 \bigwedge *g*. *g* ∈ *set* *gs* \implies monic *g* \wedge poly-mod.Mp ($p^{\wedge}n$) *g* = *g* \wedge — monic and normalized
 poly-mod.irreducible-m *p* *g* \wedge — 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 \longleftrightarrow degree p = 0*

<proof>

lemma *square-free-imp-resultant-non-zero*: **assumes** *sf: square-free (f :: int poly)*

shows *resultant f (pderiv f) \neq 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 \Rightarrow int* **where**

separable-bound f = max (abs (resultant f (pderiv f)))

(max (abs (lead-coeff f)) (abs (lead-coeff (pderiv f))))

lemma *square-free-int-imp-resultant-non-zero-mod-ring*: **assumes** *sf: square-free f*

and *large: int CARD('a) > separable-bound f*

shows *resultant (map-poly of-int f :: 'a :: prime-card mod-ring poly) (pderiv (map-poly of-int f)) \neq 0*

\wedge map-poly of-int f \neq (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** $f :: 'a :: \text{prime-card mod-ring poly}$

assumes $\text{card}: \text{CARD}('a) > \text{degree } f$

and $\text{sf}: \text{square-free } f$

shows $\text{separable } f$

<proof>

lemma *square-free-iff-separable-GFp*: **assumes** $\text{degree } f < \text{CARD}('a)$

shows $\text{square-free } (f :: 'a :: \text{prime-card mod-ring poly}) = \text{separable } f$

<proof>

definition *separable-impl-main* $:: \text{int} \Rightarrow 'i \text{ arith-ops-record} \Rightarrow \text{int poly} \Rightarrow \text{bool}$

where

$\text{separable-impl-main } p \text{ ff-ops } f = \text{separable-iff-ops } (\text{of-int-poly-iff-ops } (\text{poly-mod.Mp } p \ f))$

lemma (*in prime-field-gen*) *separable-impl*:

shows $\text{separable-impl-main } p \text{ ff-ops } f \Longrightarrow \text{square-free-m } f$

$p > \text{degree-m } f \Longrightarrow p > \text{separable-bound } f \Longrightarrow \text{square-free } f$

$\Longrightarrow \text{separable-impl-main } p \text{ ff-ops } f$ *<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* \Rightarrow *int poly* \Rightarrow *bool* **where**
separable-impl p = (
 if p \leq 65535
 then *separable-impl-main* p (*finite-field-ops32* (*uint32-of-int* p))
 else if p \leq 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 \impl *square-free-m* f

nat p > *degree-m* f \impl *nat* p > *separable-bound* f \impl *square-free* f

\impl *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 \neq 0

shows *coprime* (*lead-coeff* f) p

<proof>

lemma *prime-for-berlekamp-zassenhaus-exists*: **assumes** *sf*: *square-free* f

shows \exists p. *prime* p \wedge (*coprime* (*lead-coeff* f) p \wedge *separable-impl* p f)

<proof>

definition *next-primes* :: *nat* \Rightarrow *nat* \times *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* \Rightarrow *bool*) \Rightarrow *nat* \Rightarrow *nat list* \Rightarrow *nat* **where**

[code]: $\text{find-prime-main } f \text{ } np \text{ } ps = (\text{case } ps \text{ of } [] \Rightarrow$
 $\text{let } (np', ps') = \text{next-primes } np$
 $\text{in } \text{find-prime-main } f \text{ } np' \text{ } ps'$
 $| (p \# ps) \Rightarrow \text{if } f \text{ } p \text{ then } p \text{ else } \text{find-prime-main } f \text{ } np \text{ } ps)$

definition $\text{find-prime} :: (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{nat}$ **where**
 $\text{find-prime } f = \text{find-prime-main } f \text{ } 0 \text{ } []$

lemma next-primes : **assumes** $\text{res}: \text{next-primes } n = (m, ps)$
and n : $\text{candidate-invariant } n$
shows $\text{candidate-invariant } m$ $\text{sorted } ps$ $\text{distinct } ps$ $n < m$
 $\text{set } ps = \{i. \text{prime } i \wedge n \leq i \wedge i < m\}$
 $\langle \text{proof} \rangle$

lemma find-prime : **assumes** $\exists n. \text{prime } n \wedge f \text{ } n$
shows $\text{prime } (\text{find-prime } f) \wedge f \text{ } (\text{find-prime } f)$
 $\langle \text{proof} \rangle$

definition $\text{suitable-prime-bz} :: \text{int poly} \Rightarrow \text{int}$ **where**
 $\text{suitable-prime-bz } f \equiv \text{let } lc = \text{lead-coeff } f \text{ in } \text{int } (\text{find-prime } (\lambda n. \text{let } p = \text{int } n$
 in
 $\text{coprime } lc \text{ } p \wedge \text{separable-impl } p \text{ } f))$

lemma suitable-prime-bz : **assumes** $\text{sf}: \text{square-free } f$ **and** $p: p = \text{suitable-prime-bz}$
 f
shows $\text{prime } p$ $\text{coprime } (\text{lead-coeff } f) \text{ } p$ $\text{poly-mod.square-free-m } p \text{ } f$
 $\langle \text{proof} \rangle$

definition $\text{square-free-heuristic} :: \text{int poly} \Rightarrow \text{int option}$ **where**
 $\text{square-free-heuristic } f = (\text{let } lc = \text{lead-coeff } f \text{ in}$
 $\text{find } (\lambda p. \text{coprime } lc \text{ } p \wedge \text{separable-impl } p \text{ } f) [2, 3, 5, 7, 11, 13, 17, 19, 23])$

lemma find-Some-D : $\text{find } f \text{ } xs = \text{Some } y \Rightarrow y \in \text{set } xs \wedge f \text{ } y \langle \text{proof} \rangle$

lemma $\text{square-free-heuristic}$: **assumes** $\text{square-free-heuristic } f = \text{Some } p$
shows $\text{coprime } (\text{lead-coeff } f) \text{ } p \wedge \text{separable-impl } p \text{ } f \wedge \text{prime } p$
 $\langle \text{proof} \rangle$

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, \dots, 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* \Rightarrow *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*))
 \leq *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*)) \leq *max-factor-degree* (*map degree vs*)
<proof>

lemma *mset-sub-decompose*: *mset ds* $\subseteq\#$ *mset bs* + *as* \implies *length ds* < *length bs*
 $\implies \exists$ *b1 b b2*.

bs = *b1 @ b # b2* \wedge *mset ds* $\subseteq\#$ *mset* (*b1 @ b2*) + *as*
<proof>

lemma *max-factor-degree-aux*: **fixes** *es* :: *nat list*

assumes *sub*: *mset ds* $\subseteq\#$ *mset es*

and *len*: *length ds* + *length ds* \leq *length es* **and** *sort*: *sorted es*

shows *sum-list ds* \leq *sum-list* (*drop* (*length es div 2*) *es*)

<proof>

lemma *max-factor-degree*: **assumes** *sub*: *mset ws* $\subseteq\#$ *mset vs*

and *len*: *length ws* + *length ws* \leq *length vs*

shows *degree* (*prod-list ws*) \leq *max-factor-degree* (*map degree vs*)

<proof>

lemma *degree-bound*: **assumes** *sub*: $mset\ ws \subseteq\# mset\ vs$
and *len*: $length\ ws + length\ vs \leq length\ vs$
shows $degree\ (prod\text{-}list\ ws) \leq degree\text{-}bound\ vs$
 $\langle proof \rangle$

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 $\bigwedge a\ r. a \in set\ lst \implies P\ (f\ (h\ a)\ r)\ (f\ (g\ a)\ r)$

$\bigwedge x\ y\ z. P\ x\ y \implies P\ y\ z \implies P\ x\ z$

$P\ (F\ (map\ g\ lst))\ (F\ (map\ g\ lst))$

shows $P\ (F\ (map\ h\ lst))\ (F\ (map\ g\ lst))$

$\langle proof \rangle$

end

lemma *prod-induct-gen*:

assumes $\bigwedge a\ r. f\ (h\ a\ * r) :: 'a :: \{comm\text{-}monoid\text{-}mult\} = f\ (g\ a\ * r)$

shows $f\ (\prod v \leftarrow lst. h\ v) = f\ (\prod v \leftarrow lst. g\ v)$

$\langle proof \rangle$

abbreviation *complex-of-int::int* \Rightarrow *complex* **where**

complex-of-int \equiv *of-int*

definition *l2norm-list* :: *int list* \Rightarrow *int* **where**

l2norm-list *lst* = $\lfloor \sqrt{\text{sum-list}\ (map\ (\lambda a. a\ * a)\ lst)} \rfloor$

abbreviation *l2norm* :: *int poly* \Rightarrow *int* **where**

l2norm *p* \equiv *l2norm-list* (*coeffs* *p*)

abbreviation *norm2* *p* \equiv $\sum a \leftarrow coeffs\ p. (cmod\ a)^2$

abbreviation *l2norm-complex* **where**

l2norm-complex *p* \equiv *sqrt* (*norm2* *p*)

abbreviation $height :: int \text{ poly} \Rightarrow int$ **where**
 $height \ p \equiv max\text{-list} \ (map \ (nat \circ \text{abs}) \ (\text{coeffs} \ p))$

definition $complex\text{-roots}\text{-complex}$ **where**
 $complex\text{-roots}\text{-complex} \ (p :: complex \text{ poly}) = (SOME \ as. \ smult \ (\text{coeff} \ p \ (\text{degree} \ p))$
 $(\prod \ a \leftarrow as. \ [:- \ a, \ 1:]) = p \wedge \text{length} \ as = \text{degree} \ p)$

lemma $complex\text{-roots}$:
 $smult \ (\text{lead}\text{-coeff} \ p) \ (\prod \ a \leftarrow complex\text{-roots}\text{-complex} \ p. \ [:- \ a, \ 1:]) = p$
 $\text{length} \ (complex\text{-roots}\text{-complex} \ p) = \text{degree} \ p$
 $\langle proof \rangle$

lemma $complex\text{-roots}\text{-c}$ $[simp]$:
 $complex\text{-roots}\text{-complex} \ [:- \ c] = []$
 $\langle proof \rangle$

declare $complex\text{-roots}(2)[simp]$

lemma $complex\text{-roots}\text{-1}$ $[simp]$:
 $complex\text{-roots}\text{-complex} \ 1 = []$
 $\langle proof \rangle$

lemma $linear\text{-term}\text{-irreducible}_d[simp]$: $irreducible_d \ [:- \ a, \ 1:]$
 $\langle proof \rangle$

definition $complex\text{-roots}\text{-int}$ **where**
 $complex\text{-roots}\text{-int} \ (p :: int \text{ poly}) = complex\text{-roots}\text{-complex} \ (map\text{-poly} \ of\text{-int} \ p)$

lemma $complex\text{-roots}\text{-int}$:
 $smult \ (\text{lead}\text{-coeff} \ p) \ (\prod \ a \leftarrow complex\text{-roots}\text{-int} \ p. \ [:- \ a, \ 1:]) = map\text{-poly} \ of\text{-int} \ p$
 $\text{length} \ (complex\text{-roots}\text{-int} \ p) = \text{degree} \ p$
 $\langle proof \rangle$

The measure for polynomials, after K. Mahler

definition $mahler\text{-measure}\text{-poly}$ **where**
 $mahler\text{-measure}\text{-poly} \ p = cmod \ (\text{lead}\text{-coeff} \ p) * (\prod \ a \leftarrow complex\text{-roots}\text{-complex} \ p. \ (max \ 1 \ (cmod \ a)))$

definition $mahler\text{-measure}$ **where**
 $mahler\text{-measure} \ p = mahler\text{-measure}\text{-poly} \ (map\text{-poly} \ complex\text{-of}\text{-int} \ p)$

definition $mahler\text{-measure}\text{-monic}$ **where**
 $mahler\text{-measure}\text{-monic} \ p = (\prod \ a \leftarrow complex\text{-roots}\text{-complex} \ p. \ (max \ 1 \ (cmod \ a)))$

lemma $mahler\text{-measure}\text{-poly}\text{-via}\text{-monic}$:
 $mahler\text{-measure}\text{-poly} \ p = cmod \ (\text{lead}\text{-coeff} \ p) * mahler\text{-measure}\text{-monic} \ p$
 $\langle proof \rangle$

lemma $smult\text{-inj}[simp]$: **assumes** $(a :: 'a :: idom) \neq 0$ **shows** $inj \ (smult \ a)$

<proof>

definition *reconstruct-poly*::'a::idom \Rightarrow 'a list \Rightarrow 'a poly **where**
reconstruct-poly c roots = smult c (\prod a \leftarrow 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-poly of-int f
<proof>

lemma *reconstruct-prod*:

shows *reconstruct-poly* (a::complex) as * *reconstruct-poly* b bs
= *reconstruct-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) \neq 0
reconstruct-poly a as = *reconstruct-poly* a bs
shows mset as = mset bs
<proof>

lemma *complex-roots-complex-prod [simp]*:

assumes f \neq 0 g \neq 0
shows mset (complex-roots-complex (f * g))
= mset (complex-roots-complex f) + mset (complex-roots-complex g)
<proof>

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 $f \neq 0 \ g \neq 0$

shows $\text{mahler-measure-monic } (f * g) = \text{mahler-measure-monic } f * \text{mahler-measure-monic } g$
<proof>

lemma *mahler-measure-poly-0[simp]*: $\text{mahler-measure-poly } 0 = 0$ *<proof>*

lemma *measure-eq-prod*:

$\text{mahler-measure-poly } (f * g) = \text{mahler-measure-poly } f * \text{mahler-measure-poly } g$
<proof>

lemma *prod-cmod[simp]*:

$\text{cmod } (\prod a \leftarrow \text{lst. } f \ a) = (\prod a \leftarrow \text{lst. } \text{cmod } (f \ a))$
<proof>

lemma *lead-coeff-of-prod[simp]*:

$\text{lead-coeff } (\prod a \leftarrow \text{lst. } f \ a::'a::\text{idom poly}) = (\prod a \leftarrow \text{lst. } \text{lead-coeff } (f \ a))$
<proof>

lemma *ineq-about-squares*: **assumes** $x \leq (y::\text{real})$ **shows** $x \leq c^{\wedge}2 + y$ *<proof>*

lemma *first-coeff-le-tail*: $(\text{cmod } (\text{lead-coeff } g))^{\wedge}2 \leq (\sum a \leftarrow \text{coeffs } g. (\text{cmod } a)^{\wedge}2)$
<proof>

lemma *square-prod-cmod[simp]*:

$(\text{cmod } (a * b))^{\wedge}2 = \text{cmod } a^{\wedge}2 * \text{cmod } b^{\wedge}2$
<proof>

lemma *sum-coeffs-smult-cmod*:

$(\sum a \leftarrow \text{coeffs } (\text{smult } v \ p). (\text{cmod } a)^{\wedge}2) = (\text{cmod } v)^{\wedge}2 * (\sum a \leftarrow \text{coeffs } p. (\text{cmod } a)^{\wedge}2)$
(is ?l = ?r)
<proof>

abbreviation $\text{linH } a \equiv \text{if } (\text{cmod } a > 1) \text{ then } [:- 1, \text{cnj } a:] \text{ else } [:- a, 1:]$

lemma *coeffs-cong-1[simp]*: $\text{cCons } a \ v = \text{cCons } b \ v \longleftrightarrow a = b$ *<proof>*

lemma *strip-while-singleton[simp]*:

$\text{strip-while } ((=) \ 0) \ [v * a] = \text{cCons } (v * a) \ []$ *<proof>*

lemma *coeffs-times-linterm*:

shows $\text{coeffs } (p \ \text{Cons } 0 \ (\text{smult } a \ p) + \text{smult } b \ p) = \text{strip-while } (\text{HOL.eq } (0::'a::\{\text{comm-ring-1}\}))$
 $(\text{map } (\lambda(c,d). b*d+c*a) \ (\text{zip } (0 \ \# \ \text{coeffs } p) \ (\text{coeffs } p \ @ \ [0])))$ *<proof>*

lemma *filter-distr-rev[simp]*:

shows $filter\ f\ (rev\ lst) = rev\ (filter\ f\ lst)$
 ⟨proof⟩

lemma *strip-while-filter*:

shows $filter\ ((\neq)\ 0)\ (strip_while\ ((=)\ 0)\ (lst::'a::zero\ list)) = filter\ ((\neq)\ 0)\ lst$
 ⟨proof⟩

lemma *sum-stripwhile[simp]*:

assumes $f\ 0 = 0$
shows $(\sum\ a \leftarrow strip_while\ ((=)\ 0)\ lst.\ f\ a) = (\sum\ a \leftarrow lst.\ f\ a)$
 ⟨proof⟩

lemma *complex-split* : $Complex\ a\ b = c \longleftrightarrow (a = Re\ c \wedge b = Im\ c)$
 ⟨proof⟩

lemma *norm-times-const*: $(\sum\ y \leftarrow lst.\ (cmod\ (a * y))^2) = (cmod\ a)^2 * (\sum\ y \leftarrow lst.\ (cmod\ y)^2)$
 ⟨proof⟩

fun *bisumTail* **where**

$bisumTail\ f\ (Cons\ a\ (Cons\ b\ bs)) = f\ a\ b + bisumTail\ f\ (Cons\ b\ bs) \mid$
 $bisumTail\ f\ (Cons\ a\ Nil) = f\ a\ 0 \mid$
 $bisumTail\ f\ Nil = f\ 1\ 0$

fun *bisum* **where**

$bisum\ f\ (Cons\ a\ as) = f\ 0\ a + bisumTail\ f\ (Cons\ a\ as) \mid$
 $bisum\ f\ Nil = f\ 0\ 0$

lemma *bisumTail-is-map-zip*:

$(\sum\ x \leftarrow zip\ (v\ \# \ l1)\ (l1\ @\ [0]).\ f\ x) = bisumTail\ (\lambda x\ y.\ f\ (x,y))\ (v\ \# \ l1)$
 ⟨proof⟩

lemma *bisum-is-map-zip*:

$(\sum\ x \leftarrow zip\ (0\ \# \ l1)\ (l1\ @\ [0]).\ f\ x) = bisum\ (\lambda x\ y.\ f\ (x,y))\ l1$
 ⟨proof⟩

lemma *map-zip-is-bisum*:

$bisum\ f\ l1 = (\sum\ (x,y) \leftarrow zip\ (0\ \# \ l1)\ (l1\ @\ [0]).\ f\ x\ y)$
 ⟨proof⟩

lemma *bisum-outside* :

$(bisum\ (\lambda x\ y.\ f1\ x - f2\ x\ y + f3\ y)\ lst :: 'a :: field)$
 $= sum_list\ (map\ f1\ lst) + f1\ 0 - bisum\ f2\ lst + sum_list\ (map\ f3\ lst) + f3\ 0$
 ⟨proof⟩

lemma *Landau-lemma*:

$(\sum\ a \leftarrow coeffs\ (\prod\ a \leftarrow lst.\ [:-\ a,\ 1:]).\ (cmod\ a)^2) = (\sum\ a \leftarrow coeffs\ (\prod\ a \leftarrow lst.\ linH\ a).\ (cmod\ a)^2)$
 (is *norm2 ?l = norm2 ?r*)
 ⟨proof⟩

lemma *Landau-inequality*:
mahler-measure-poly $f \leq l2norm\text{-}complex\ f$
 $\langle proof \rangle$

lemma *prod-list-ge1*:
assumes *Ball* (*set* x) $(\lambda (a::real). a \geq 1)$
shows *prod-list* $x \geq 1$
 $\langle proof \rangle$

lemma *mahler-measure-monic-ge-1*: *mahler-measure-monic* $p \geq 1$
 $\langle proof \rangle$

lemma *mahler-measure-monic-ge-0*: *mahler-measure-monic* $p \geq 0$
 $\langle proof \rangle$

lemma *mahler-measure-ge-0*: $0 \leq mahler\text{-}measure\ h$ $\langle proof \rangle$

lemma *mahler-measure-constant[simp]*: *mahler-measure-poly* $[:c:] = c\ mod\ c$
 $\langle proof \rangle$

lemma *mahler-measure-factor[simplified,simp]*: *mahler-measure-poly* $[:-\ a, 1:] =$
 $max\ 1\ (c\ mod\ a)$
 $\langle proof \rangle$

lemma *mahler-measure-poly-explicit*: *mahler-measure-poly* $(smult\ c\ (\prod a \leftarrow as.\ [:-\ a, 1:]))$
 $=\ c\ mod\ c\ * (\prod a \leftarrow as.\ (max\ 1\ (c\ mod\ a)))$
 $\langle proof \rangle$

lemma *mahler-measure-poly-ge-1*:
assumes $h \neq 0$
shows $(1::real) \leq mahler\text{-}measure\ h$
 $\langle proof \rangle$

lemma *mahler-measure-dvd*: **assumes** $f \neq 0$ **and** $h\ dvd\ f$
shows $mahler\text{-}measure\ h \leq mahler\text{-}measure\ f$
 $\langle proof \rangle$

definition *graeffe-poly* :: $'a \Rightarrow 'a :: comm\text{-}ring\text{-}1\ list \Rightarrow nat \Rightarrow 'a\ poly$ **where**
graeffe-poly $c\ as\ m = smult\ (c\ ^\ (2^\ m))\ (\prod a \leftarrow as.\ [:-\ (a\ ^\ (2^\ m)), 1:])$

context
fixes $f :: complex\ poly$ **and** $c\ as$
assumes $f = smult\ c\ (\prod a \leftarrow as.\ [:-\ a, 1:])$
begin
lemma *mahler-graeffe*: *mahler-measure-poly* $(graeffe\text{-}poly\ c\ as\ m) = (mahler\text{-}measure\text{-}poly\ f)^\ (2^\ m)$

<proof>
end

fun *drop-half* :: 'a list \Rightarrow 'a list **where**
drop-half (x # y # ys) = x # *drop-half* ys
| *drop-half* xs = xs

fun *alternate* :: 'a list \Rightarrow 'a list \times 'a list **where**
alternate (x # y # ys) = (case *alternate* ys of (evn, od) \Rightarrow (x # evn, y # od))
| *alternate* xs = (xs,[])

definition *poly-square-subst* :: 'a :: comm-ring-1 poly \Rightarrow 'a poly **where**
poly-square-subst f = *poly-of-list* (*drop-half* (*coeffs* f))

definition *poly-even-odd* :: 'a :: comm-ring-1 poly \Rightarrow 'a poly \times 'a poly **where**
poly-even-odd f = (case *alternate* (*coeffs* f) of (evn, od) \Rightarrow (*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 \circ_p (*monom* 1 2)) = f
<proof>

lemma *poly-even-odd*: **assumes** *poly-even-odd* f = (g, h)
shows f = g \circ_p *monom* 1 2 + *monom* 1 1 * (h \circ_p *monom* 1 2)
<proof>

context
fixes f :: 'a :: idom poly
begin

lemma *graeffe-0*: f = *smult* c (\prod a \leftarrow as. [:- a, 1:]) \Longrightarrow *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 \circ_p [:-0, -1:]))
<proof>
end

definition *graeffe-one-step* :: 'a \Rightarrow 'a :: idom poly \Rightarrow 'a poly **where**
graeffe-one-step c f = *smult* c (*poly-square-subst* (f * f \circ_p [:-0, -1:]))

lemma *graeffe-one-step-code*[code]: **fixes** $c :: 'a :: idom$
shows $graeffe-one-step\ c\ f = (case\ poly-even-odd\ f\ of\ (g,h)$
 $\Rightarrow\ smult\ c\ (g * g - monom\ 1\ 1 * h * h))$
 $\langle proof \rangle$

fun *graeffe-poly-impl-main* :: $'a \Rightarrow 'a :: idom\ poly \Rightarrow nat \Rightarrow 'a\ poly$ **where**
 $graeffe-poly-impl-main\ c\ f\ 0 = f$
 $| graeffe-poly-impl-main\ c\ f\ (Suc\ m) = graeffe-one-step\ c\ (graeffe-poly-impl-main\ c\ f\ m)$

lemma *graeffe-poly-impl-main*: **assumes** $f = smult\ c\ (\prod a \leftarrow as.\ [:-\ a,\ 1:])$
shows $graeffe-poly-impl-main\ ((-1)^{degree\ f})\ f\ m = graeffe-poly\ c\ as\ m$
 $\langle proof \rangle$

definition *graeffe-poly-impl* :: $'a :: idom\ poly \Rightarrow nat \Rightarrow 'a\ poly$ **where**
 $graeffe-poly-impl\ f = graeffe-poly-impl-main\ ((-1)^{degree\ f})\ f$

lemma *graeffe-poly-impl*: **assumes** $f = smult\ c\ (\prod a \leftarrow as.\ [:-\ a,\ 1:])$
shows $graeffe-poly-impl\ f\ m = graeffe-poly\ c\ as\ m$
 $\langle proof \rangle$

lemma *drop-half-map*: $drop-half\ (map\ f\ xs) = map\ f\ (drop-half\ xs)$
 $\langle proof \rangle$

lemma (**in** *inj-comm-ring-hom*) *map-poly-poly-square-subst*:
 $map-poly\ hom\ (poly-square-subst\ f) = poly-square-subst\ (map-poly\ hom\ f)$
 $\langle proof \rangle$

context *inj-idom-hom*
begin

lemma *graeffe-poly-impl-hom*:
 $map-poly\ hom\ (graeffe-poly-impl\ f\ m) = graeffe-poly-impl\ (map-poly\ hom\ f)\ m$
 $\langle proof \rangle$
end

lemma *graeffe-poly-impl-mahler*: $mahler-measure\ (graeffe-poly-impl\ f\ m) = mahler-measure\ f\ ^2\ ^m$
 $\langle proof \rangle$

definition *mahler-landau-graeffe-approximation* :: $nat \Rightarrow nat \Rightarrow int\ poly \Rightarrow int$
where

$mahler-landau-graeffe-approximation\ kk\ dd\ f = (let$
 $no = sum-list\ (map\ (\lambda a.\ a * a)\ (coeffs\ f))$
 $in\ root-int-floor\ kk\ (dd * no))$

lemma *mahler-landau-graeffe-approximation-core*:
assumes $g: g = graeffe-poly-impl\ f\ k$
shows $mahler-measure\ f \leq root\ (2\ ^{Suc\ k})\ (real-of-int\ (\sum a \leftarrow coeffs\ g.\ a * a))$

<proof>

lemma *Landau-inequality-mahler-measure: mahler-measure $f \leq \text{sqrt} (\text{real-of-int} (\sum a \leftarrow \text{coeffs } f. a * a))$*
<proof>

lemma *mahler-landau-graeffe-approximation:*

assumes *g: $g = \text{graeffe-poly-impl } f \ k \ dd = d^{\wedge}(2^{\wedge}(\text{Suc } k)) \ kk = 2^{\wedge}(\text{Suc } k)$*

shows *$\lfloor \text{real } d * \text{mahler-measure } f \rfloor \leq \text{mahler-landau-graeffe-approximation } kk \ dd$*

g

<proof>

context

fixes *bnd :: nat*

begin

function *mahler-approximation-main :: nat \Rightarrow int \Rightarrow int poly \Rightarrow int \Rightarrow nat \Rightarrow nat \Rightarrow 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 \geq 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 $k \neq 0 \implies \lfloor \text{real } d * \text{mahler-measure } f \rfloor \leq mm$*

and *$c = (-1)^{\wedge}(\text{degree } f)$*

and *$g = \text{graeffe-poly-impl-main } c \ f \ k \ dd = d^{\wedge}(2^{\wedge}(\text{Suc } k)) \ kk = 2^{\wedge}(\text{Suc } k)$*

shows *$\lfloor \text{real } d * \text{mahler-measure } f \rfloor \leq \text{mahler-approximation-main } dd \ c \ g \ mm \ k \ kk$*

<proof>

definition *mahler-approximation :: nat \Rightarrow int poly \Rightarrow int* **where**

*mahler-approximation d f = mahler-approximation-main (d * d) ((-1)^{\wedge}(\text{degree } f)) f (-1) 0 2*

lemma *mahler-approximation: $\lfloor \text{real } d * \text{mahler-measure } f \rfloor \leq \text{mahler-approximation } d \ f$*

<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*: $n \leq N \implies n \text{ choose } k \leq N \text{ choose } k$

<proof>

definition *choose-int* **where** *choose-int* $m\ n = (\text{if } n < 0 \text{ then } 0 \text{ else } m \text{ choose } (\text{nat } n))$

lemma *choose-int-suc*[*simp*]:

choose-int (*Suc* n) $i = \text{choose-int } n\ (i-1) + \text{choose-int } n\ i$

<proof>

lemma *sum-le-1-prod*: **assumes** $d: 1 \leq d$ **and** $c: 1 \leq c$

shows $c + d \leq 1 + c * (d :: \text{real})$

<proof>

lemma *mignotte-helper-coeff-int*: $\text{cmod } (\text{coeff-int } (\prod a \leftarrow \text{lst. } [:- a, 1:])\ i)$
 $\leq \text{choose-int } (\text{length } \text{lst} - 1)\ i * (\prod a \leftarrow \text{lst. } (\text{max } 1\ (\text{cmod } a)))$
 $+ \text{choose-int } (\text{length } \text{lst} - 1)\ (i - 1)$

<proof>

lemma *mignotte-helper-coeff-int'*: $\text{cmod } (\text{coeff-int } (\prod a \leftarrow \text{lst. } [:- a, 1:])\ i)$
 $\leq ((\text{length } \text{lst} - 1)\ \text{choose } i) * (\prod a \leftarrow \text{lst. } (\text{max } 1\ (\text{cmod } a)))$
 $+ \text{min } i\ 1 * ((\text{length } \text{lst} - 1)\ \text{choose } (\text{nat } (i - 1)))$

<proof>

lemma *mignotte-helper-coeff*:

$\text{cmod } (\text{coeff } h\ i) \leq (\text{degree } h - 1\ \text{choose } i) * \text{mahler-measure-poly } h$
 $+ \text{min } i\ 1 * (\text{degree } h - 1\ \text{choose } (i - 1)) * \text{cmod } (\text{lead-coeff } h)$

<proof>

lemma *mignotte-coeff-helper*:

$\text{abs } (\text{coeff } h\ i) \leq$
 $(\text{degree } h - 1\ \text{choose } i) * \text{mahler-measure } h +$
 $(\text{min } i\ 1 * (\text{degree } h - 1\ \text{choose } (i - 1))) * \text{abs } (\text{lead-coeff } h)$

<proof>

lemma *cmod-through-lead-coeff*[*simp*]:

$\text{cmod } (\text{lead-coeff } (\text{of-int-poly } h)) = \text{abs } (\text{lead-coeff } h)$

<proof>

lemma *choose-approx*: $n \leq N \implies n \text{ choose } k \leq N \text{ choose } (N \text{ div } 2)$

<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* \Rightarrow *nat* \Rightarrow *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* \neq 0 *g* dvd *f* degree *g* \leq *n*

shows |coeff *g k*| \leq [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*| \leq (degree *g* choose *k*) * mahler-measure *g*

<proof>

lemma *mignotte-bound*:

assumes *f* \neq 0 *g* dvd *f* degree *g* \leq *n*

shows |coeff *g k*| \leq *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* \Rightarrow *nat* \Rightarrow *int* **where**

factor-bound = *mignotte-bound*

lemma *factor-bound*: **assumes** *f* \neq 0 *g* dvd *f* degree *g* \leq *n*

shows |coeff *g k*| \leq *factor-bound f n*

<proof>

We further prove a result for factor bounds and scalar multiplication.

lemma *factor-bound-ge-0*: *f* \neq 0 \implies *factor-bound f n* \geq 0

<proof>

lemma *factor-bound-smult*: **assumes** *f*: *f* \neq 0 **and** *d*: *d* \neq 0

and *dvd*: *g* dvd *smult d f* **and** *deg*: degree *g* \leq *n*

shows |coeff *g k*| \leq |*d*| * *factor-bound f n*

<proof>

end

10.6 Iteration of Subsets of Factors

theory *Sublist-Iteration*

imports

Polynomial-Factorization.Missing-Multiset

Polynomial-Factorization.Missing-List

HOL-Library.IArray

begin

Misc lemmas **lemma** *mem-snd-map*: $(\exists x. (x, y) \in S) \longleftrightarrow y \in \text{snd } S$ *<proof>*

lemma *filter-upt*: **assumes** $l \leq m$ $m < n$ **shows** $\text{filter } ((\leq) m) [l..<n] = [m..<n]$ *<proof>*

lemma *upt-append*: $i < j \implies j < k \implies [i..<j]@[j..<k] = [i..<k]$ *<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]*: $\text{subseqs } (x\#xs) = \text{map } (Cons\ x) (\text{subseqs } xs) @ \text{subseqs } xs$ *<proof>*

declare *subseqs.simps(2) [simp del]*

lemma *singleton-mem-set-subseqs [simp]*: $[x] \in \text{set } (\text{subseqs } xs) \longleftrightarrow x \in \text{set } xs$ *<proof>*

lemma *Cons-mem-set-subseqsD*: $y\#ys \in \text{set } (\text{subseqs } xs) \implies y \in \text{set } xs$ *<proof>*

lemma *subseqs-subset*: $ys \in \text{set } (\text{subseqs } xs) \implies \text{set } ys \subseteq \text{set } xs$ *<proof>*

lemma *Cons-mem-set-subseqs-Cons*:

$y\#ys \in \text{set } (\text{subseqs } (x\#xs)) \longleftrightarrow (y = x \wedge ys \in \text{set } (\text{subseqs } xs)) \vee y\#ys \in \text{set } (\text{subseqs } xs)$ *<proof>*

lemma *sorted-subseqs-sorted*:

$\text{sorted } xs \implies ys \in \text{set } (\text{subseqs } xs) \implies \text{sorted } ys$ *<proof>*

lemma *subseqs-of-subseq*: $ys \in \text{set } (\text{subseqs } xs) \implies \text{set } (\text{subseqs } ys) \subseteq \text{set } (\text{subseqs } xs)$ *<proof>*

lemma *mem-set-subseqs-append*: $xs \in \text{set } (\text{subseqs } ys) \implies xs \in \text{set } (\text{subseqs } (zs @ ys))$ *<proof>*

lemma *Cons-mem-set-subseqs-append*:

$x \in \text{set } ys \implies xs \in \text{set } (\text{subseqs } zs) \implies x\#xs \in \text{set } (\text{subseqs } (ys@zs))$ *<proof>*

lemma *Cons-mem-set-subseqs-sorted*:

$sorted\ xs \implies y\#\!ys \in set\ (subseqs\ xs) \implies y\#\!ys \in set\ (subseqs\ (filter\ (\lambda x. y \leq x)\ xs))$
 $\langle proof \rangle$

lemma *subseqs-map[simp]*: $subseqs\ (map\ f\ xs) = map\ (map\ f)\ (subseqs\ xs)$ $\langle proof \rangle$

lemma *subseqs-of-indices*: $map\ (map\ (nth\ xs))\ (subseqs\ [0..<length\ xs]) = subseqs\ xs$
 $\langle proof \rangle$

Specification definition *subseq-of-length* $n\ xs\ ys \equiv ys \in set\ (subseqs\ xs) \wedge length\ ys = n$

lemma *subseq-of-lengthI[intro]*:
assumes $ys \in set\ (subseqs\ xs)\ length\ ys = n$
shows *subseq-of-length* $n\ xs\ ys$
 $\langle proof \rangle$

lemma *subseq-of-lengthD[dest]*:
assumes *subseq-of-length* $n\ xs\ ys$
shows $ys \in set\ (subseqs\ xs)\ length\ ys = n$
 $\langle proof \rangle$

lemma *subseq-of-length0[simp]*: *subseq-of-length* $0\ xs\ ys \longleftrightarrow ys = []$ $\langle proof \rangle$

lemma *subseq-of-length-Nil[simp]*: *subseq-of-length* $n\ []\ ys \longleftrightarrow n = 0 \wedge ys = []$
 $\langle proof \rangle$

lemma *subseq-of-length-Suc-upt*:
subseq-of-length $(Suc\ n)\ [0..<m]\ xs \longleftrightarrow$
 (if $n = 0$ then $length\ xs = Suc\ 0 \wedge hd\ xs < m$
 else $hd\ xs < hd\ (tl\ xs) \wedge subseq-of-length\ n\ [0..<m]\ (tl\ xs)$) (is $?l \longleftrightarrow ?r$)
 $\langle proof \rangle$

lemma *subseqs-of-length-of-indices*:
 $\{ ys. subseq-of-length\ n\ xs\ ys \} = \{ map\ (nth\ xs)\ is \mid is. subseq-of-length\ n\ [0..<length\ xs]\ is \}$
 $\langle proof \rangle$

lemma *subseqs-of-length-Suc-Cons*:
 $\{ ys. subseq-of-length\ (Suc\ n)\ (x\#\!xs)\ ys \} =$
 $Cons\ x\ ' \{ ys. subseq-of-length\ n\ xs\ ys \} \cup \{ ys. subseq-of-length\ (Suc\ n)\ xs\ ys \}$
 $\langle proof \rangle$

datatype $('a, 'b, 'state)subseqs-impl = Sublists-Impl$
create-subseqs: $'b \Rightarrow 'a\ list \Rightarrow nat \Rightarrow ('b \times 'a\ list)list \times 'state$
next-subseqs: $'state \Rightarrow ('b \times 'a\ list)list \times 'state$


```

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)

```

($\lambda a xs n \text{ triple. } (a, xs, n) = \text{triple}$)
 <proof>

Improved Implementation `datatype ('a,'b,'state) subseqs-foldr-impl = Sublists-Foldr-Impl`

(`subseqs-foldr: 'b \Rightarrow 'a list \Rightarrow nat \Rightarrow 'b list \times 'state`)
 (`next-subseqs-foldr: 'state \Rightarrow 'b list \times 'state`)

locale `subseqs-foldr-impl =`

fixes `f :: 'a \Rightarrow 'b \Rightarrow 'b`

and `impl :: ('a,'b,'state) subseqs-foldr-impl`

begin

definition `S where S base elements n \equiv { foldr f ys base | ys. subseq-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 \Rightarrow 'a list \Rightarrow nat \Rightarrow 'state \Rightarrow bool`

assumes `subseqs-foldr:`

`subseqs-foldr impl base elements n = (out, state) \impl`

`invariant base elements n state \wedge set out = S base elements n`

and `next-subseqs-foldr:`

`next-subseqs-foldr impl state = (out, state') \impl invariant base elements n state`

\impl

`invariant base elements (Suc n) state' \wedge set out = S base elements (Suc n)`

locale `my-subseqs =`

fixes `f :: 'a \Rightarrow 'b \Rightarrow 'b`

begin

context **fixes** `head :: 'a` **and** `tail :: 'a iarray`

begin

fun `next-subseqs1` **and** `next-subseqs2`

where `next-subseqs1 ret0 ret1 [] = (ret0, (head, tail, ret1))`

| `next-subseqs1 ret0 ret1 ((i,v)#prevs) = next-subseqs2 (f head v # ret0) ret1 prevs v [0..`

| `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 \equiv { (j, f (tail !! j) v) | j. j \in set js }`

definition `out-subseqs2-set v js \equiv { f (tail !! j) v | j. j \in set js }`

definition `next-subseqs1-set prevs \equiv \bigcup { next-subseqs2-set v [0..\in set prevs }`

definition *out-subseqs1-set* *prevs* \equiv
 $(f \text{ head} \circ \text{snd}) \text{ ` set } prevs \cup (\bigcup \{ \text{out-subseqs2-set } v [0..<i] \mid v \text{ i. } (i,v) \in \text{set } prevs$
 $\})$

fun *next-subseqs1-spec* **where**
next-subseqs1-spec *out* *nexts* *prevs* (*out'*, (*head'*,*tail'*,*nexts'*)) \longleftrightarrow
set *nexts'* = *set* *nexts* \cup *next-subseqs1-set* *prevs* \wedge
set *out'* = *set* *out* \cup *out-subseqs1-set* *prevs*

fun *next-subseqs2-spec* **where**
next-subseqs2-spec *out* *nexts* *prevs* *v* *js* (*out'*, (*head'*,*tail'*,*nexts'*)) \longleftrightarrow
set *nexts'* = *set* *nexts* \cup *next-subseqs1-set* *prevs* \cup *next-subseqs2-set* *v* *js* \wedge
set *out'* = *set* *out* \cup *out-subseqs1-set* *prevs* \cup *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*)
 \langle *proof* \rangle

lemma *out-subseqs2-Cons*:
out-subseqs2-set *v* (*j*#*js*) = *insert* (*f* (*tail*!!*j*) *v*) (*out-subseqs2-set* *v* *js*)
 \langle *proof* \rangle

lemma *next-subseqs1-set-as-next-subseqs2-set*:
next-subseqs1-set ((*i*,*v*) # *prevs*) = *next-subseqs1-set* *prevs* \cup *next-subseqs2-set* *v*
 $[0..<i]$
 \langle *proof* \rangle

lemma *out-subseqs1-set-as-out-subseqs2-set*:
out-subseqs1-set ((*i*,*v*) # *prevs*) =
 $\{ f \text{ head } v \} \cup \text{out-subseqs1-set } prevs \cup \text{out-subseqs2-set } v [0..<i]$
 \langle *proof* \rangle

lemma *next-subseqs1-spec*:
shows \bigwedge *out* *nexts*. *next-subseqs1-spec* *out* *nexts* *prevs* (*next-subseqs1* *out* *nexts*
prevs)
and \bigwedge *out* *nexts*. *next-subseqs2-spec* *out* *nexts* *prevs* *v* *js* (*next-subseqs2* *out* *nexts*
prevs *v* *js*)
 \langle *proof* \rangle

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 next: next-subseqs state = (out, state')
shows set out = S base elements (Suc n)
⟨proof⟩

```

```

lemma create-subseqs:
  create-subseqs base elements n = (out, state)  $\implies$ 
    invariant base elements n state  $\wedge$  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 $f \bmod p^n$, 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 ( $\lambda x. \text{map-prod } (f x) (g x)$ ) 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* \Rightarrow *int poly* **where**
inv-Mp = *map-poly inv-M*

definition *mul-const* :: *int poly* \Rightarrow *int* \Rightarrow *int* **where**
mul-const p c = (*coeff p 0* * *c*) mod *m*

fun *prod-list-m* :: *int poly list* \Rightarrow *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* \times *int poly list*, *'state*)*subseqs-foldr-impl*
and *m2* :: *int*

begin

definition *inv-M2* :: *int* \Rightarrow *int* **where**
inv-M2 = (λx . if $x \leq m2$ then x else $x - m$)

definition *inv-Mp2* :: *int poly* \Rightarrow *int poly* **where**
inv-Mp2 = *map-poly inv-M2*

partial-function (*tailrec*) *reconstruction* :: *'state* \Rightarrow *int poly* \Rightarrow *int poly*
 \Rightarrow *int* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *int poly list* \Rightarrow *int poly list*
 \Rightarrow (*int* \times (*int poly list*)) *list* \Rightarrow *int poly list* **where**
*reconstruction state u luu lu d r vs res cand*s = (*case cand*s of *Nil*
 \Rightarrow let $d' = \text{Suc } d$
in if $d' + d' > r$ then ($u \# \text{res}$) else
(*case next-subseqs-foldr sl-impl state of (cands, state')* \Rightarrow
*reconstruction state' u luu lu d' r vs res cand*s)
| (*lv', ws*) # *cands'* \Rightarrow 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*;
 $u' = u \text{ div } pp\text{-}vb$;
 $r' = r - \text{length } ws$;
 $\text{res}' = pp\text{-}vb \# \text{res}$
in if $d + d > r'$
then $u' \# \text{res}'$
else let
 $lu' = \text{lead-coeff } u'$;
 $vs' = \text{fold remove1 } ws \text{ vs}$;
(*cands''*, *state'*) = *subseqs-foldr sl-impl (lu', []) vs' d*
in *reconstruction state' u' (smult lu' u') lu' d r' vs' res' cand*s''
else *reconstruction state u luu lu d r vs res cand*s'
else *reconstruction state u luu lu d r vs res cand*s')

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 = $p^{\widehat{n}}$;
 (-, 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** \neg *irreducible_d (u :: 'a :: idom*
poly) degree u ≠ 0
shows $\exists f g. smult (lead-coeff u) u = f * g \wedge lead-coeff f = lead-coeff u \wedge$
lead-coeff g = lead-coeff u
 $\wedge degree f < degree u \wedge degree g < degree u$
 ⟨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** $bnd: \bigwedge n. 2 * abs (coeff f n) < m$
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 $sl-impl :: (int poly, int \times int poly list, 'state)subseqs-foldr-impl$

and $sli :: int \times int poly list \Rightarrow int poly list \Rightarrow nat \Rightarrow 'state \Rightarrow bool$

assumes *prime*: $prime p$

and $m: m = p^{\wedge}n$

and $n: n \neq 0$

and $sl-impl: correct-subseqs-foldr-impl (\lambda x. map-prod (mul-const x) (Cons x))$
 $sl-impl sli$

begin

private definition *test-dvd-exec* $lu u ws = (\neg inv-Mp (Mp (smult lu (prod-mset ws)))) dvd smult lu u$

private definition *test-dvd* $u ws = (\forall v l. v dvd u \longrightarrow 0 < degree v \longrightarrow degree v < degree u \longrightarrow \neg v =m smult l (prod-mset ws))$

private definition *large-m* $u vs = (\forall v n. v dvd u \longrightarrow degree v \leq degree-bound vs \longrightarrow 2 * abs (coeff v n) < m)$

lemma *large-m-factor*: $large-m u vs \Longrightarrow v dvd u \Longrightarrow large-m v vs$
<proof>

lemma *test-dvd-factor*: **assumes** $u: u \neq 0$ **and** *test*: $test-dvd u ws$ **and** *vu*: $v dvd u$

shows $test-dvd v ws$

<proof>

lemma *coprime-exp-mod*: $coprime lu p \Longrightarrow prime p \Longrightarrow n \neq 0 \Longrightarrow lu mod p^{\wedge}n \neq 0$

<proof>

interpretation *correct-subseqs-foldr-impl* $\lambda x. map-prod (mul-const x) (Cons x)$
 $sl-impl sli$ *<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: ∧ v. v ∈ set vs ⇒ Mp v = v*
and *tests: ∧ ws. ws ⊆ # mset vs ⇒ ws ≠ {#} ⇒*
size ws < d ∨ size ws = d ∧ ws ∉ (mset o snd) ‘ set candS
⇒ test-dvd u ws
and *irr: ∧ f. f ∈ set res ⇒ irreducible_d f*
and *deg: degree u > 0*
and *candS-ne: candS ≠ [] ⇒ d < r*
and *large: ∀ v n. v dvd smult lu u → degree v ≤ degree-bound vs*
*→ 2 * abs (coeff v n) < m*
and *f0: f ≠ 0*
and *state: sli (lu, []) vs d state*
and *m2: m2 = m div 2*
shows *f = prod-list fs ∧ (∀ fi ∈ set fs. irreducible_d fi)*
<proof>
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 (pⁿ);
sl-impl = my-subseqs.impl (λx. 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) < pⁿ*
begin

private lemma n: n ≠ 0

<proof>

interpretation p : *poly-mod-prime* p \langle proof \rangle

lemma *zassenhaus-reconstruction-generic*:

assumes *sl-impl*: *correct-subseqs-foldr-impl* $(\lambda v. \text{map-prod } (\text{poly-mod.mul-const } (p \hat{=} n) v) (\text{Cons } v)) \text{ sl-impl } sli$
and *res*: *zassenhaus-reconstruction-generic* *sl-impl* hs p n $f = fs$
shows $f = \text{prod-list } fs \wedge (\forall fi \in \text{set } fs. \text{irreducible}_d fi)$
 \langle proof \rangle

lemma *zassenhaus-reconstruction-irreducible_d*:

assumes *res*: *zassenhaus-reconstruction* hs p n $f = fs$
shows $f = \text{prod-list } fs \wedge (\forall fi \in \text{set } fs. \text{irreducible}_d fi)$
 \langle proof \rangle

corollary *zassenhaus-reconstruction*:

assumes *pr*: *primitive* f
assumes *res*: *zassenhaus-reconstruction* hs p n $f = fs$
shows $f = \text{prod-list } fs \wedge (\forall fi \in \text{set } fs. \text{irreducible } fi)$
 \langle proof \rangle

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 $[[\text{code abort: dummy-Gcd}]]$

lemma *dummy-Gcd-Lcm*: Gcd $x = \text{dummy-Gcd } x$ Lcm $x = \text{dummy-Lcm } x$
 \langle proof \rangle

lemmas *dummy-Gcd-Lcm-poly* $[code] = \text{dummy-Gcd-Lcm}$

$[\text{where } ?'a = 'a :: \{\text{factorial-ring-gcd, semiring-gcd-mult-normalize}\} \text{ poly}]$

lemmas *dummy-Gcd-Lcm-int* $[code] = \text{dummy-Gcd-Lcm}$ $[\text{where } ?'a = \text{int}]$

lemmas *dummy-Gcd-Lcm-nat* $[code] = \text{dummy-Gcd-Lcm}$ $[\text{where } ?'a = \text{nat}]$

declare $[[\text{code abort: Euclidean-Algorithm.Gcd Euclidean-Algorithm.Lcm}]]$

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 $p \wedge \text{find-exponent } p \text{ bnd} > \text{bnd}$ *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 $p^k > \text{bnd}$

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* \vee *coprime (lead-coeff g) p* **and**
cnt: *content f = 1* \vee *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* \in *set gcd-primes32* \implies *prime p* \wedge *p* \leq 65535
<proof>

definition *gcd-primes64* :: *int list* **where**
gcd-primes64 = [383, 21984191, 50329901, 80329901, 219849193]

lemma *gcd-primes64*: *p* \in *set gcd-primes64* \implies *prime p* \wedge *p* \leq 4294967295
<proof>

definition *coprime-heuristic* :: *int poly* \Rightarrow *int poly* \Rightarrow *bool* **where**
coprime-heuristic f g = (*let lcf = lead-coeff f*; *lcg = lead-coeff g* *in*
find ($\lambda p. (coprime lcf p \vee coprime lcg p) \wedge coprime-approx-main p (finite-field-ops64$
(uint64-of-int p)) f g)
gcd-primes64 \neq *None*)

lemma *coprime-heuristic*: **assumes** *coprime-heuristic f g*
and *content f = 1* \vee *content g = 1*
shows *coprime f g*
<proof>

definition *gcd-int-poly* :: *int poly* \Rightarrow *int poly* \Rightarrow *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]: $gcd = gcd\text{-int-poly}$
⟨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* :: $int\ poly \Rightarrow rat \Rightarrow rat\ poly \Rightarrow bool$ **where**
yun-wrel $F\ c\ f = (map\text{-poly}\ rat\text{-of-int}\ F = smult\ c\ f)$

definition *yun-rel* :: $int\ poly \Rightarrow rat \Rightarrow rat\ poly \Rightarrow bool$ **where**
yun-rel $F\ c\ f = (yun\text{-wrel}\ F\ c\ f$
 $\wedge\ content\ F = 1 \wedge lead\text{-coeff}\ F > 0 \wedge monic\ f)$

definition *yun-erel* :: $int\ poly \Rightarrow rat\ poly \Rightarrow bool$ **where**
yun-erel $F\ f = (\exists\ c. yun\text{-rel}\ F\ c\ f)$

lemma *yun-wrelD*: **assumes** $yun\text{-wrel}\ F\ c\ f$
shows $map\text{-poly}\ rat\text{-of-int}\ F = smult\ c\ f$
⟨proof⟩

lemma *yun-relD*: **assumes** $yun\text{-rel}\ F\ c\ f$
shows $yun\text{-wrel}\ F\ c\ f\ map\text{-poly}\ rat\text{-of-int}\ F = smult\ c\ f$
 $degree\ F = degree\ f\ F \neq 0\ lead\text{-coeff}\ F > 0\ monic\ f$
 $f = 1 \iff F = 1\ content\ F = 1$
⟨proof⟩

lemma *yun-erel-1-eq*: **assumes** $yun\text{-erel}\ F\ f$
shows $(F = 1) \iff (f = 1)$
⟨proof⟩

lemma *yun-rel-1[simp]*: $yun\text{-rel}\ 1\ 1\ 1$
⟨proof⟩

lemma *yun-erel-1[simp]*: $yun\text{-erel}\ 1\ 1$ ⟨proof⟩

lemma *yun-rel-mult*: $yun\text{-rel}\ F\ c\ f \implies yun\text{-rel}\ G\ d\ g \implies yun\text{-rel}\ (F * G)\ (c * d)$

$(f * g)$
 $\langle \text{proof} \rangle$

lemma *yun-erel-mult*: $\text{yun-erel } F f \implies \text{yun-erel } G g \implies \text{yun-erel } (F * G) (f * g)$

$\langle \text{proof} \rangle$

lemma *yun-rel-pow*: **assumes** $\text{yun-rel } F c f$
shows $\text{yun-rel } (F^{\wedge} n) (c^{\wedge} n) (f^{\wedge} n)$
 $\langle \text{proof} \rangle$

lemma *yun-erel-pow*: $\text{yun-erel } F f \implies \text{yun-erel } (F^{\wedge} n) (f^{\wedge} n)$
 $\langle \text{proof} \rangle$

lemma *yun-wrel-pderiv*: **assumes** $\text{yun-wrel } F c f$
shows $\text{yun-wrel } (\text{pderiv } F) c (\text{pderiv } f)$
 $\langle \text{proof} \rangle$

lemma *yun-wrel-minus*: **assumes** $\text{yun-wrel } F c f \text{ yun-wrel } G c g$
shows $\text{yun-wrel } (F - G) c (f - g)$
 $\langle \text{proof} \rangle$

lemma *yun-wrel-div*: **assumes** f : $\text{yun-wrel } F c f$ **and** g : $\text{yun-wrel } G d g$
and dvd : $G \text{ dvd } F g \text{ dvd } f$
and $G0$: $G \neq 0$
shows $\text{yun-wrel } (F \text{ div } G) (c / d) (f \text{ div } g)$
 $\langle \text{proof} \rangle$

lemma *yun-rel-div*: **assumes** f : $\text{yun-rel } F c f$ **and** g : $\text{yun-rel } G d g$
and dvd : $G \text{ dvd } F g \text{ dvd } f$
shows $\text{yun-rel } (F \text{ div } G) (c / d) (f \text{ div } g)$
 $\langle \text{proof} \rangle$

lemma *yun-wrel-gcd*: **assumes** $\text{yun-wrel } F c' f \text{ yun-wrel } G c g$ **and** c : $c' \neq 0 \ c \neq 0$
and d : $d = \text{rat-of-int } (\text{lead-coeff } (\text{gcd } F G)) \ d \neq 0$
shows $\text{yun-wrel } (\text{gcd } F G) d (\text{gcd } f g)$
 $\langle \text{proof} \rangle$

lemma *yun-rel-gcd*: **assumes** f : $\text{yun-rel } F c f$ **and** g : $\text{yun-wrel } G c' g$ **and** c' : $c' \neq 0$
and d : $d = \text{rat-of-int } (\text{lead-coeff } (\text{gcd } F G))$
shows $\text{yun-rel } (\text{gcd } F G) d (\text{gcd } f g)$
 $\langle \text{proof} \rangle$

lemma *yun-factorization-main-int*: **assumes** $f = p \text{ div gcd } p \text{ (pderiv } p)$
and $g = \text{pderiv } p \text{ div gcd } p \text{ (pderiv } p) \text{ monic } p$
and $\text{yun-gcd.yun-factorization-main gcd } f \ g \ i \ \text{hs} = \text{res}$
and $\text{yun-gcd.yun-factorization-main gcd } F \ G \ i \ \text{Hs} = \text{Res}$
and $\text{yun-rel } F \ c \ f \ \text{yun-wrel } G \ c \ g \ \text{list-all2 (rel-prod yun-erel (=)) Hs hs}$
shows $\text{list-all2 (rel-prod yun-erel (=)) Res res}$
 $\langle \text{proof} \rangle$

lemma *yun-monic-factorization-int-yun-rel*: **assumes**
 $\text{res: yun-gcd.yun-monic-factorization gcd } f = \text{res}$
and $\text{Res: yun-gcd.yun-monic-factorization gcd } F = \text{Res}$
and $f: \text{yun-rel } F \ c \ f$
shows $\text{list-all2 (rel-prod yun-erel (=)) Res res}$
 $\langle \text{proof} \rangle$

lemma *yun-rel-same-right*: **assumes** $\text{yun-rel } f \ c \ G \ \text{yun-rel } g \ d \ G$
shows $f = g$
 $\langle \text{proof} \rangle$

definition *square-free-factorization-int-main* :: $\text{int poly} \Rightarrow (\text{int poly} \times \text{nat}) \text{ list}$
where
 $\text{square-free-factorization-int-main } f = (\text{case square-free-heuristic } f \ \text{of None} \Rightarrow$
 $\text{yun-gcd.yun-monic-factorization gcd } f \ | \ \text{Some } p \Rightarrow [(f, 0)])$

lemma *square-free-factorization-int-main*: **assumes** $\text{res: square-free-factorization-int-main}$
 $f = \text{fs}$
and $\text{ct: content } f = 1$ **and** $\text{lc: lead-coeff } f > 0$
and $\text{deg: degree } f \neq 0$
shows $\text{square-free-factorization } f \ (1, \text{fs}) \wedge (\forall \text{ fi } i. (\text{fi}, i) \in \text{set } \text{fs} \longrightarrow \text{content } \text{fi} =$
 $1 \wedge \text{lead-coeff } \text{fi} > 0) \wedge$
 $\text{distinct (map snd } \text{fs})$
 $\langle \text{proof} \rangle$

definition *square-free-factorization-int'* :: $\text{int poly} \Rightarrow \text{int} \times (\text{int poly} \times \text{nat}) \text{ list}$
where
 $\text{square-free-factorization-int'} \ f = (\text{if degree } f = 0$
 $\text{then (lead-coeff } f, []) \ \text{else (let — content factorization}$
 $\ c = \text{content } f;$
 $\ d = (\text{sgn (lead-coeff } f) * c);$
 $\ g = \text{sdiv-poly } f \ d$
 $\ \text{— and square-free factorization}$
 $\ \text{in } (d, \text{square-free-factorization-int-main } g)))$

lemma *square-free-factorization-int'*: **assumes** $\text{res: square-free-factorization-int'} \ f$
 $= (d, \text{fs})$
shows $\text{square-free-factorization } f \ (d, \text{fs})$

$(fi, i) \in set\ fs \implies content\ fi = 1 \wedge lead-coeff\ fi > 0$
 $distinct\ (map\ snd\ fs)$
 <proof>

definition $x-split :: 'a :: semiring-0\ poly \Rightarrow nat \times 'a\ poly$ **where**
 $x-split\ f = (let\ fs = coeffs\ f; zs = takeWhile\ ((=)\ 0)\ fs$
 $in\ case\ zs\ of\ [] \Rightarrow (0, f) \mid - \Rightarrow (length\ zs, poly-of-list\ (dropWhile\ ((=)\ 0)\ fs))$

lemma $x-split$: **assumes** $x-split\ f = (n, g)$
shows $f = monom\ 1\ n * g\ n \neq 0 \vee f \neq 0 \implies \neg\ monom\ 1\ 1\ dvd\ g$
 <proof>

definition $square-free-factorization-int :: int\ poly \Rightarrow int \times (int\ poly \times nat)list$
where
 $square-free-factorization-int\ f = (case\ x-split\ f\ of\ (n, g) \text{ --- } extract\ x\ \hat{n}$
 $\Rightarrow case\ square-free-factorization-int'\ g\ of\ (d, fs)$
 $\Rightarrow if\ n = 0\ then\ (d, fs)\ else\ (d, (monom\ 1\ 1, n - 1) \# fs))$

lemma $square-free-factorization-int$: **assumes** $res: square-free-factorization-int\ f$
 $= (d, fs)$
shows $square-free-factorization\ f\ (d, fs)$
 $(fi, i) \in set\ fs \implies primitive\ fi \wedge lead-coeff\ fi > 0$
 <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 $int-poly-factorization-algorithm = \{alg.$
 $\forall\ (f :: int\ poly)\ fs. square-free\ f \longrightarrow degree\ f > 0 \longrightarrow alg\ f = fs \longrightarrow$
 $(f = prod-list\ fs \wedge (\forall\ fi \in set\ fs. irreducible_d\ fi))\}$
 <proof>

setup-lifting *type-definition-int-poly-factorization-algorithm*

lift-definition *int-poly-factorization-algorithm* :: *int-poly-factorization-algorithm*
 \Rightarrow

(*int poly* \Rightarrow *int poly list*) **is** $\lambda x. x$ *<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 \wedge ($\forall fi \in set fs. irreducible_d fi$)*

<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 \wedge ($\forall fi \in set fs. irreducible fi \wedge degree fi > 0 \wedge primitive fi$)*

<proof>

lemma *irreducible-imp-square-free*:

assumes *irr: irreducible (p::'a::idom poly)* **shows** *square-free p*

<proof>

lemma *not-mem-set-dropWhileD*: *$x \notin set (dropWhile P xs) \Longrightarrow x \in set xs \Longrightarrow P x$*

<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 \subseteq 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 $\text{smult } c \ f = \text{smult } c \ g \iff c = 0 \vee f = g$
(proof)

lemma *primitive-part-reflect-poly*:

fixes $f :: 'a :: \{\text{semiring-gcd}, \text{idom}\}$ poly

shows $\text{primitive-part } (\text{reflect-poly } f) = \text{reflect-poly } (\text{primitive-part } f)$ (is ?l = ?r)

(proof)

lemma *reflect-poly-eq-zero[simp]*:

$\text{reflect-poly } f = 0 \iff f = 0$

(proof)

lemma *irreducible_a-reflect-poly-main*:

fixes $f :: 'a :: \{\text{idom}, \text{semiring-gcd}\}$ poly

assumes $\text{nz}: \text{coeff } f \ 0 \neq 0$

and $\text{irr}: \text{irreducible}_a (\text{reflect-poly } f)$

shows $\text{irreducible}_a f$

(proof)

lemma *irreducible_a-reflect-poly*:

fixes $f :: 'a :: \{\text{idom}, \text{semiring-gcd}\}$ poly

assumes $\text{nz}: \text{coeff } f \ 0 \neq 0$

shows $\text{irreducible}_a (\text{reflect-poly } f) = \text{irreducible}_a f$

(proof)

lemma *irreducible-reflect-poly*:

fixes $f :: 'a :: \{\text{idom}, \text{semiring-gcd}\}$ poly

assumes $\text{nz}: \text{coeff } f \ 0 \neq 0$

shows $\text{irreducible } (\text{reflect-poly } f) = \text{irreducible } f$ (is ?l = ?r)

(proof)

lemma *reflect-poly-dvd*: $(f :: 'a :: \text{idom poly}) \text{ dvd } g \implies \text{reflect-poly } f \text{ dvd } \text{reflect-poly } g$

(proof)

lemma *square-free-reflect-poly*: **fixes** $f :: 'a :: \text{idom poly}$

assumes $\text{sf}: \text{square-free } f$

and $\text{nz}: \text{coeff } f \ 0 \neq 0$

shows $\text{square-free } (\text{reflect-poly } f)$ (proof)

lemma *gcd-reflect-poly*: **fixes** $f :: 'a :: \{\text{factorial-ring-gcd}, \text{semiring-gcd-mult-normalize}\}$ poly

assumes $\text{nz}: \text{coeff } f \ 0 \neq 0 \ \text{coeff } g \ 0 \neq 0$

shows $\text{gcd } (\text{reflect-poly } f) (\text{reflect-poly } g) = \text{normalize } (\text{reflect-poly } (\text{gcd } f \ g))$

(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
 $\langle proof \rangle$

lemma *square-free-factorization-last-coeff-nz*:
assumes $sff: square-free-factorization\ f\ (a, fs)$
and $mem: (fi, i) \in set\ fs$
and $nz: coeff\ f\ 0 \neq 0$
shows $coeff\ fi\ 0 \neq 0$
 $\langle proof \rangle$

context

fixes $alg :: int-poly-factorization-algorithm$
begin

definition *main-int-poly-factorization* $:: int\ poly \Rightarrow int\ poly\ list$ **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* $:: int\ poly \Rightarrow int \times (int\ poly \times nat)$ *list*
where
internal-int-poly-factorization $f = ($
case *square-free-factorization-int* f *of*
 $(a, gis) \Rightarrow (a, [(h, i) . (g, i) \leftarrow gis, h \leftarrow main-int-poly-factorization\ g])$
 $)$

lemma *internal-int-poly-factorization-code* $[code]: internal-int-poly-factorization\ f =$
 $($
case *square-free-factorization-int* f *of* $(a, gis) \Rightarrow$
 $(a, concat\ (map\ (\lambda\ (g, i). (map\ (\lambda\ f. (f, i))\ (main-int-poly-factorization\ g)))\ gis)))$
 $\langle proof \rangle$

definition *factorize-int-last-nz-poly* $:: int\ poly \Rightarrow int \times (int\ poly \times nat)$ *list* **where**
factorize-int-last-nz-poly $f = (let\ df = degree\ f$
in if $df = 0$ *then* $(coeff\ f\ 0, [])$ *else if* $df = 1$ *then* $(content\ f, [(primitive-part\ f, 0)])$ *else*
internal-int-poly-factorization $f)$

definition *factorize-int-poly-generic* $:: int\ poly \Rightarrow int \times (int\ poly \times nat)$ *list* **where**
factorize-int-poly-generic $f = (case\ x-split\ f\ of\ (n, g) \text{ --- extract } x^n$

\Rightarrow if $g = 0$ then $(0, \square)$ else case factorize-int-last-nz-poly g of (a, fs)
 \Rightarrow if $n = 0$ then (a, fs) else $(a, (\text{monom } 1 \ 1, n - 1) \# fs)$

lemma factorize-int-poly-0[simp]: factorize-int-poly-generic $0 = (0, \square)$
 <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 $f \ 0 \neq 0$
shows $f = \text{prod-list } fs \wedge (\forall fi \in \text{set } fs. \text{irreducible}_d \ fi)$
 <proof>

lemma internal-int-poly-factorization-mem:

assumes f : coeff $f \ 0 \neq 0$
and res: internal-int-poly-factorization $f = (c, fs)$
and mem: $(fi, i) \in \text{set } fs$
shows irreducible fi irreducible _{d} fi **and** primitive fi **and** degree $fi \neq 0$
 <proof>

lemma internal-int-poly-factorization:

assumes f : coeff $f \ 0 \neq 0$
and res: internal-int-poly-factorization $f = (c, fs)$
shows square-free-factorization $f \ (c, fs)$
 <proof>

lemma factorize-int-last-nz-poly: **assumes** res: factorize-int-last-nz-poly $f = (c, fs)$

and nz: coeff $f \ 0 \neq 0$
shows square-free-factorization $f \ (c, fs)$
 $(fi, i) \in \text{set } fs \implies \text{irreducible } fi$
 $(fi, i) \in \text{set } fs \implies \text{degree } fi \neq 0$
 <proof>

lemma factorize-int-poly: **assumes** res: factorize-int-poly-generic $f = (c, fs)$

shows square-free-factorization $f \ (c, fs)$
 $(fi, i) \in \text{set } fs \implies \text{irreducible } fi$
 $(fi, i) \in \text{set } fs \implies \text{degree } fi \neq 0$
 <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 \equiv 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 \Rightarrow -

<proof>

lemma *prod-dvd-1-imp-all-dvd-1:*

assumes finite X and prod f X dvd 1 and x \in X shows f x dvd 1

<proof>

context

fixes alg :: int-poly-factorization-algorithm

begin

definition *factorize-rat-poly-generic :: rat poly \Rightarrow rat \times (rat poly \times nat) list where*

factorize-rat-poly-generic f = (case rat-to-normalized-int-poly f of

*(c,g) \Rightarrow case factorize-int-poly-generic alg g of (d,fs) \Rightarrow (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) \in set fs \implies irreducible fi

<proof>

end

abbreviation *factorize-rat-poly where*

factorize-rat-poly \equiv factorize-rat-poly-generic berlekamp-zassenhaus-factorization-algorithm

end

References

- [1] G. Barthe, B. Grégoire, S. Heraud, F. Olmedo, and S. Z. Béguelin. Verified indifferentiable hashing into elliptic curves. In *POST 2012*, volume 7215 of *LNCS*, pages 209–228, 2012.
- [2] E. R. Berlekamp. Factoring polynomials over finite fields. *Bell System Technical Journal*, 46:1853–1859, 1967.
- [3] D. G. Cantor and H. Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Math. Comput.*, 36(154):587–592, 1981.
- [4] J. R. Cowles and R. Gamboa. Unique factorization in ACL2: Euclidean domains. In *Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 21–27. ACM, 2006.
- [5] B. Huffman and O. Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *CPP 2013*, volume 8307 of *LNCS*, pages 131–146, 2013.
- [6] B. Kirkels. *Irreducibility Certificates for Polynomials with Integer Coefficients*. PhD thesis, Radboud Universiteit Nijmegen, 2004.
- [7] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981.
- [8] H. Kobayashi, H. Suzuki, and Y. Ono. Formalization of Hensel’s lemma. In *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*, volume 1, pages 114–118, 2005.
- [9] O. Kunčar and A. Popescu. From types to sets by local type definitions in higher-order logic. In *ITP 2016*, volume 9807 of *LNCS*, pages 200–218, 2016.
- [10] É. Martin-Dorel, G. Hanrot, M. Mayero, and L. Théry. Formally verified certificate checkers for hardest-to-round computation. *Journal of Automated Reasoning*, 54(1):1–29, 2015.
- [11] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [12] R. Thiemann and A. Yamada. Algebraic numbers in Isabelle/HOL. In *ITP 2016*, volume 9807 of *LNCS*, pages 391–408, 2016.
- [13] H. Zassenhaus. On Hensel factorization, I. *Journal of Number Theory*, 1(3):291–311, 1969.