Polynomial Interpolation*

René Thiemann and Akihisa Yamada

March 17, 2025

Abstract

We formalized three algorithms for polynomial interpolation over arbitrary fields: Lagrange's explicit expression, the recursive algorithm of Neville and Aitken, and the Newton interpolation in combination with an efficient implementation of divided differences. Variants of these algorithms for integer polynomials are also available, where sometimes the interpolation can fail; e.g., there is no linear integer polynomial p such that $p(0) = 0$ and $p(2) = 1$. Moreover, for the Newton interpolation for integer polynomials, we proved that all intermediate results that are computed during the algorithm must be integers. This admits an early failure detection in the implementation. Finally, we proved the uniqueness of polynomial interpolation.

The development also contains improved code equations to speed up the division of integers in target languages.

Contents

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

1 Introduction

We formalize three basic algorithms for interpolation for univariate field polynomials and integer polynomials which can be found in various textbooks or on Wikipedia. However, this formalization covers only basic results, e.g., compared to a specialized textbook on interpolation [1], we only cover results of the first of the eight chapters.

Given distinct inputs $x_0, \ldots, x_n$ and corresponding outputs $y_0, \ldots, y_n$, *polynomial interpolation* is to provide a polynomial p (of degree at most n) such that $p(x_i) = y_i$ for every $i < n$.

The first solution we formalize is Lagrange's explicit expression:

$$p(x) = \sum_{i<n} \Big(y_i \cdot \prod_{\substack{j<n \\ j \neq i}} \frac{x - x_j}{x_i - x_j} \Big)$$

which is however expensive since the computation involves a number of multiplications and additions of polynomials. Hence we formalize other algorithms, namely, the recursive algorithms of Neville and Aitken, and the Newton interpolation. We also show that a polynomial interpolation of degree at most n is unique.

Further, we consider a variant of the interpolation problem where the base type is restricted to *int*. In this case the result must be an integer polynomial (i.e., the coefficients are integers), which does not necessarily exist even if the specified inputs and outputs are integers. For instance, there exists no linear integer polynomial p such that $p(0) = 0$ and $p(2) = 1$.

We prove that, for the Newton interpolation to produce integer polynomials, the intermediate coefficients computed in the procedure must be always integers. This result, in practice allows the implementation to detect failure as early as possible, and in theory shows that there is no integer polynomial p satisfying $p(0) = 0$ and $p(2) = 1$, regardless of the degree of the polynomial.

The formalization also contains an improved code equations for integer division.

2 Conversions to Rational Numbers

We define a class which provides tests whether a number is rational, and a conversion from to rational numbers. These conversion functions are principle the inverse functions of *of-rat*, but they can be implemented for individual types more efficiently.

Similarly, we define tests and conversions between integer and rational numbers.

theory *Is-Rat-To-Rat*
imports
  Sqrt-Babylonian.Sqrt-Babylonian-Auxiliary
begin

class *is-rat* = *field-char-0* +
  fixes *is-rat* :: *'a* ⇒ *bool*
  and *to-rat* :: *'a* ⇒ *rat*
  assumes *is-rat*[*simp*]: *is-rat x* = ($x \in \mathbb{Q}$)
  and *to-rat*: *to-rat x* = (*if* $x \in \mathbb{Q}$ *then* (*THE y. x* = *of-rat y*) *else 0*)

lemma *of-rat-to-rat*[*simp*]: $x \in \mathbb{Q} \Longrightarrow$ *of-rat* (*to-rat x*) = *x*
  unfolding *to-rat Rats-def* **by** *auto*

lemma *to-rat-of-rat*[*simp*]: *to-rat* (*of-rat x*) = *x* **unfolding** *to-rat* **by** *simp*

instantiation *rat* :: *is-rat*
begin
definition *is-rat-rat* (*x* :: *rat*) = *True*
definition *to-rat-rat* (*x* :: *rat*) = *x*
  instance

by (*intro-classes*, *auto simp*: *is-rat-rat-def to-rat-rat-def Rats-def*)
end

The definition for reals at the moment is not executable, but it will become executable after loading the real algebraic numbers theory.

instantiation *real* :: *is-rat*
begin
definition *is-rat-real* (x :: *real*) = ($x \in \mathbb{Q}$)
definition *to-rat-real* (x :: *real*) = (*if* $x \in \mathbb{Q}$ *then* (*THE y*. x = *of-rat y*) *else 0*)
instance by (*intro-classes*, *auto simp*: *is-rat-real-def to-rat-real-def*)
end

lemma *of-nat-complex*: *of-nat n* = *Complex* (*of-nat n*) *0*
by (*simp add*: *complex-eqI*)

lemma *of-int-complex*: *of-int z* = *Complex* (*of-int z*) *0*
by (*simp add*: *complex-eq-iff*)

lemma *of-rat-complex*: *of-rat q* = *Complex* (*of-rat q*) *0*
proof −
obtain *d n* **where** *dn*: *quotient-of q* = (*d*,*n*) **by** *force*
from *quotient-of-div*[*OF dn*] **have** *q*: *q* = *of-int d* / *of-int n* **by** *auto*
then have *of-rat q* = *complex-of-real* (*real-of-rat q*) ∨ (*0*::*complex*) = *of-int n* ∨ *0* = *real-of-int n*
by (*simp add*: *of-rat-divide q*)
then show *?thesis*
using *Complex-eq-0 complex-of-real-def q* **by** *auto*
qed

lemma *complex-of-real-of-rat*[*simp*]: *complex-of-real* (*real-of-rat q*) = *of-rat q*
unfolding *complex-of-real-def of-rat-complex* **by** *simp*

lemma *is-rat-complex-iff*: $x \in \mathbb{Q} \longleftrightarrow$ *Re* $x \in \mathbb{Q}$ ∧ *Im* x = *0*
proof
assume $x \in \mathbb{Q}$
then obtain *q* **where** *x*: x = *of-rat q* **unfolding** *Rats-def* **by** *auto*
let *?y* = *Complex* (*of-rat q*) *0*
have x − *?y* = *0* **unfolding** *x* **by** (*simp add*: *Complex-eq*)
hence *x*: x = *?y* **by** *simp*
show *Re* $x \in \mathbb{Q}$ ∧ *Im* x = *0* **unfolding** *x complex.sel* **by** *auto*
next
assume *Re* $x \in \mathbb{Q}$ ∧ *Im* x = *0*
then obtain *q* **where** *Re* x = *of-rat q Im* x = *0* **unfolding** *Rats-def* **by** *auto*
hence x = *Complex* (*of-rat q*) *0* **by** (*metis complex-surj*)
thus $x \in \mathbb{Q}$ **by** (*simp add*: *Complex-eq*)
qed

instantiation *complex* :: *is-rat*
begin

definition *is-rat-complex* (*x* :: *complex*) = (*is-rat* (*Re x*) ∧ *Im x* = *0*)
definition *to-rat-complex* (*x* :: *complex*) = (*if is-rat* (*Re x*) ∧ *Im x* = *0 then to-rat* (*Re x*) *else 0*)

instance proof (*intro-classes*, *auto simp*: *is-rat-complex-def to-rat-complex-def is-rat-complex-iff*)
  fix *x*
  assume *r*: *Re x* ∈ ℚ **and** *i*: *Im x* = *0*
  hence *x* ∈ ℚ **unfolding** *is-rat-complex-iff* **by** *auto*
  then obtain *y* **where** *x*: *x* = *of-rat y* **unfolding** *Rats-def* **by** *blast*
  from *this*[*unfolded of-rat-complex*] **have** *x*: *x* = *Complex* (*real-of-rat y*) *0* **by** *auto*
  show *to-rat* (*Re x*) = (*THE y*. *x* = *of-rat y*)
    by (*subst of-rat-eq-iff*[*symmetric*, **where** *′a* = *real*], *unfold of-rat-to-rat*[*OF r*] *of-rat-complex*,
    unfold x complex.sel, *auto*)
qed
end

lemma *in-rats-code-unfold*[*code-unfold*]: (*x* ∈ ℚ) = (*is-rat x*) **by** *simp*

definition *is-int-rat* :: *rat* ⇒ *bool* **where**
  is-int-rat x ≡ *snd* (*quotient-of x*) = *1*

definition *int-of-rat* :: *rat* ⇒ *int* **where**
  int-of-rat x ≡ *fst* (*quotient-of x*)

lemma *is-int-rat*[*simp*]: *is-int-rat x* = (*x* ∈ ℤ)
  unfolding *is-int-rat-def Ints-def*
  by (*metis Ints-def Ints-induct*
    quotient-of-int is-int-rat-def old.prod.exhaust quotient-of-inject rangeI snd-conv)

lemma *in-ints-code-unfold*[*code-unfold*]: (*x* ∈ ℤ) = *is-int-rat x*
  by *simp*

lemma *int-of-rat*[*simp*]: *int-of-rat* (*rat-of-int x*) = *x z* ∈ ℤ ⟹ *rat-of-int* (*int-of-rat z*) = *z*
proof (*force simp*: *int-of-rat-def*)
  assume *z* ∈ ℤ
  thus *rat-of-int* (*int-of-rat z*) = *z* **unfolding** *int-of-rat-def*
    by (*metis Ints-cases Pair-inject quotient-of-int surjective-pairing*)
qed

lemma *int-of-rat-0*[*simp*]: (*int-of-rat x* = *0*) = (*x* = *0*) **unfolding** *int-of-rat-def*
  using *quotient-of-div*[*of x*] **by** (*cases quotient-of x*, *auto*)

end

3 Divmod-Int

We provide the divmod-operation on type int for efficiency reasons.

theory *Divmod-Int*
imports *Main*
begin

definition *divmod-int* :: *int* ⇒ *int* ⇒ *int* × *int* **where**
divmod-int n m = (*n div m*, *n mod m*)

We implement *divmod-int* via *divmod-integer* instead of invoking both division and modulo separately.

context
includes *integer.lifting*
begin

lemma *divmod-int-code*[*code*]: *divmod-int m n* = *map-prod int-of-integer int-of-integer*

(*divmod-integer* (*integer-of-int m*) (*integer-of-int n*))
by (*simp add*: *Divmod-Int.divmod-int-def divmod-integer-def*)

end

end

4 Improved Code Equations

This theory contains improved code equations for certain algorithms.

theory *Improved-Code-Equations*
imports
HOL−Computational-Algebra.Polynomial
HOL−Library.Code-Target-Nat
begin

4.1 *divmod-integer*.

We improve *divmod-integer ?k ?l* = (*if ?k* = *0 then* (*0*, *0*) *else if 0* < *?l then if 0* < *?k then Code-Numeral.divmod-abs ?k ?l else case Code-Numeral.divmod-abs ?k ?l of* (*r*, *s*) ⇒ *if s* = *0 then* (− *r*, *0*) *else* (− *r* − *1*, *?l* − *s*) *else if ?l* = *0 then* (*0*, *?k*) *else apsnd uminus* (*if ?k* < *0 then Code-Numeral.divmod-abs ?k ?l else case Code-Numeral.divmod-abs ?k ?l of* (*r*, *s*) ⇒ *if s* = *0 then* (− *r*, *0*) *else* (− *r* − *1*, − *?l* − *s*))) by deleting *sgn*-expressions.

We guard the application of divmod-abs' with the condition $0 \le x \wedge 0 \le y$, so that application can be ensured on non-negative values. Hence, one can drop "abs" in target language setup.

definition *divmod-abs′* **where**

$x \geq 0 \Longrightarrow y \geq 0 \Longrightarrow$ *divmod-abs′ x y* = *Code-Numeral.divmod-abs x y*

lemma *divmod-integer-code′′*[*code*]: *divmod-integer k l* =
(*if k* = *0 then* (*0*, *0*)
else if l > *0 then*
(*if k* > *0 then divmod-abs′ k l*
else case divmod-abs′ (− *k*) *l of* (*r*, *s*) ⇒
if s = *0 then* (− *r*, *0*) *else* (− *r* − *1*, *l* − *s*))
else if l = *0 then* (*0*, *k*)
else apsnd uminus
(*if k* < *0 then divmod-abs′* (−*k*) (−*l*)
else case divmod-abs′ k (−*l*) *of* (*r*, *s*) ⇒
if s = *0 then* (− *r*, *0*) *else* (− *r* − *1*, − *l* − *s*)))
unfolding *divmod-integer-code*
by (*cases l* = *0*; *cases l* < *0*; *cases l* > *0*; *auto split*: *prod.splits simp*: *divmod-abs′-def divmod-abs-def*)

code-printing — FIXME illusion of partiality
constant *divmod-abs′* ⇀
(*SML*) *IntInf.divMod*/ (-,/ -)
and (*Eval*) *Integer.div′-mod*/ (-)/ (-)
and (*OCaml*) *Z.div′-rem*
and (*Haskell*) *divMod*/ (-)/ (-)
and (*Scala*) !((*k*: *BigInt*) => (*l*: *BigInt*) => *l* == *0 match* { *case true* => (*BigInt*(*0*), *k*) *case false* => (*k* ′/% *l*) })

4.2 *Euclidean-Rings.divmod-nat*.

We implement *Euclidean-Rings.divmod-nat* via *divmod-integer* instead of invoking both division and modulo separately, and we further simplify the case-analysis which is performed in *divmod-integer ?k ?l* = (*if ?k* = *0 then* (*0*, *0*) *else if 0* < *?l then if 0* < *?k then divmod-abs′ ?k ?l else case divmod-abs′* (− *?k*) *?l of* (*r*, *s*) ⇒ *if s* = *0 then* (− *r*, *0*) *else* (− *r* − *1*, *?l* − *s*) *else if ?l* = *0 then* (*0*, *?k*) *else apsnd uminus* (*if ?k* < *0 then divmod-abs′* (− *?k*) (− *?l*) *else case divmod-abs′ ?k* (− *?l*) *of* (*r*, *s*) ⇒ *if s* = *0 then* (− *r*, *0*) *else* (− *r* − *1*, − *?l* − *s*))).

lemma *divmod-nat-code′*[*code*]: *Euclidean-Rings.divmod-nat m n* = (
let k = *integer-of-nat m*; *l* = *integer-of-nat n*
in map-prod nat-of-integer nat-of-integer
(*if k* = *0 then* (*0*, *0*)
else if l = *0 then* (*0*,*k*) *else*
divmod-abs′ k l))
using *divmod-nat-code* [*of m n*]
by (*simp add*: *divmod-abs′-def integer-of-nat-eq-of-nat Let-def*)

4.3 (*choose*)

lemma *binomial-code*[*code*]:
  n choose k = (*if* $k \leq n$ *then fact n div* (*fact k* ∗ *fact* (*n* − *k*)) *else 0*)
  using *binomial-eq-0*[*of n k*] *binomial-altdef-nat*[*of k n*] **by** *simp*

end

5 Several Locales for Homomorphisms Between Types.

theory *Ring-Hom*
imports
  HOL.Complex
  Main
  HOL−Library.Multiset
  HOL−Computational-Algebra.Factorial-Ring
begin

hide-const (**open**) *mult*

Many standard operations can be interpreted as homomorphisms in some sense. Since declaring some lemmas as [simp] will interfere with existing simplification rules, we introduce named theorems that would be added to the simp set when necessary.

The following collects distribution lemmas for homomorphisms. Its symmetric version can often be useful.

named-theorems *hom-distribs*

5.1 Basic Homomorphism Locales

locale *zero-hom* =
  fixes *hom* :: *'a* :: *zero* ⇒ *'b* :: *zero*
  assumes *hom-zero*[*simp*]: *hom 0* = *0*

locale *one-hom* =
  fixes *hom* :: *'a* :: *one* ⇒ *'b* :: *one*
  assumes *hom-one*[*simp*]: *hom 1* = *1*

locale *times-hom* =
  fixes *hom* :: *'a* :: *times* ⇒ *'b* :: *times*
  assumes *hom-mult*[*hom-distribs*]: *hom* (*x* ∗ *y*) = *hom x* ∗ *hom y*

locale *plus-hom* =
  fixes *hom* :: *'a* :: *plus* ⇒ *'b* :: *plus*
  assumes *hom-add*[*hom-distribs*]: *hom* (*x* + *y*) = *hom x* + *hom y*

locale *semigroup-mult-hom* =

```
  times-hom hom for hom :: 'a :: semigroup-mult ⇒ 'b :: semigroup-mult

locale semigroup-add-hom =
  plus-hom hom for hom :: 'a :: semigroup-add ⇒ 'b :: semigroup-add

locale monoid-mult-hom = one-hom hom + semigroup-mult-hom hom
  for hom :: 'a :: monoid-mult ⇒ 'b :: monoid-mult
begin
```

Homomorphism distributes over product:

```
  lemma hom-prod-list: hom (prod-list xs) = prod-list (map hom xs)
    by (induct xs, auto simp: hom-distribs)
```

but since it introduces unapplied *hom*, the reverse direction would be simp.

```
  lemmas prod-list-map-hom[simp] = hom-prod-list[symmetric]
  lemma hom-power[hom-distribs]: hom (x ^ n) = hom x ^ n
    by (induct n, auto simp: hom-distribs)
end

locale monoid-add-hom = zero-hom hom + semigroup-add-hom hom
  for hom :: 'a :: monoid-add ⇒ 'b :: monoid-add
begin
  lemma hom-sum-list: hom (sum-list xs) = sum-list (map hom xs)
    by (induct xs, auto simp: hom-distribs)
  lemmas sum-list-map-hom[simp] = hom-sum-list[symmetric]
  lemma hom-add-eq-zero: assumes x + y = 0 shows hom x + hom y = 0
  proof −
    have 0 = x + y using assms..
    hence hom 0 = hom (x + y) by simp
    thus ?thesis by (auto simp: hom-distribs)
  qed
end

locale group-add-hom = monoid-add-hom hom
  for hom :: 'a :: group-add ⇒ 'b :: group-add
begin
  lemma hom-uminus[hom-distribs]: hom (−x) = − hom x
    by (simp add: eq-neg-iff-add-eq-0 hom-add-eq-zero)
  lemma hom-minus [hom-distribs]: hom (x − y) = hom x − hom y
    unfolding diff-conv-add-uminus hom-distribs..
end
```

5.2 Commutativity

```
locale ab-semigroup-mult-hom = semigroup-mult-hom hom
  for hom :: 'a :: ab-semigroup-mult ⇒ 'b :: ab-semigroup-mult

locale ab-semigroup-add-hom = semigroup-add-hom hom
```

for *hom* :: *'a* :: *ab-semigroup-add* ⇒ *'b* :: *ab-semigroup-add*

locale *comm-monoid-mult-hom* = *monoid-mult-hom hom*
for *hom* :: *'a* :: *comm-monoid-mult* ⇒ *'b* :: *comm-monoid-mult*
begin
sublocale *ab-semigroup-mult-hom*..
lemma *hom-prod*[*hom-distribs*]: *hom* (*prod f X*) = ($\prod$ *x* ∈ *X*. *hom* (*f x*))
by (*cases finite X*, *induct rule*:*finite-induct*; *simp add*: *hom-distribs*)
lemma *hom-prod-mset*: *hom* (*prod-mset X*) = *prod-mset* (*image-mset hom X*)
by (*induct X*, *auto simp*: *hom-distribs*)
lemmas *prod-mset-image*[*simp*] = *hom-prod-mset*[*symmetric*]
lemma *hom-dvd*[*intro*,*simp*]: **assumes** *p dvd q* **shows** *hom p dvd hom q*
proof −
from *assms* **obtain** *r* **where** *q* = *p* ∗ *r* **unfolding** *dvd-def* **by** *auto*
from *arg-cong*[*OF this*, *of hom*] **show** *?thesis* **unfolding** *dvd-def* **by** (*auto simp*: *hom-distribs*)
qed
lemma *hom-dvd-1*[*simp*]: *x dvd 1* ⟹ *hom x dvd 1* **using** *hom-dvd*[*of x 1*] **by** *simp*
end

locale *comm-monoid-add-hom* = *monoid-add-hom hom*
for *hom* :: *'a* :: *comm-monoid-add* ⇒ *'b* :: *comm-monoid-add*
begin
sublocale *ab-semigroup-add-hom*..
lemma *hom-sum*[*hom-distribs*]: *hom* (*sum f X*) = ($\sum$ *x* ∈ *X*. *hom* (*f x*))
by (*cases finite X*, *induct rule*:*finite-induct*; *simp add*: *hom-distribs*)
lemma *hom-sum-mset*[*hom-distribs*,*simp*]: *hom* (*sum-mset X*) = *sum-mset* (*image-mset hom X*)
by (*induct X*, *auto simp*: *hom-distribs*)
end

locale *ab-group-add-hom* = *group-add-hom hom*
for *hom* :: *'a* :: *ab-group-add* ⇒ *'b* :: *ab-group-add*
begin
sublocale *comm-monoid-add-hom*..
end

locale *semiring-hom* = *comm-monoid-add-hom hom* + *monoid-mult-hom hom*
for *hom* :: *'a* :: *semiring-1* ⇒ *'b* :: *semiring-1*
begin
lemma *hom-mult-eq-zero*: **assumes** *x* ∗ *y* = *0* **shows** *hom x* ∗ *hom y* = *0*
proof −
have *0* = *x* ∗ *y* **using** *assms*..
hence *hom 0* = *hom* (*x* ∗ *y*) **by** *simp*
thus *?thesis* **by** (*auto simp*:*hom-distribs*)
qed
end

locale *ring-hom* = *semiring-hom hom*
  for *hom* :: *'a* :: *ring-1* ⇒ *'b* :: *ring-1*
begin
  sublocale *ab-group-add-hom hom*..
end

locale *comm-semiring-hom* = *semiring-hom hom*
  for *hom* :: *'a* :: *comm-semiring-1* ⇒ *'b* :: *comm-semiring-1*
begin
  sublocale *comm-monoid-mult-hom*..
end

locale *comm-ring-hom* = *ring-hom hom*
  for *hom* :: *'a* :: *comm-ring-1* ⇒ *'b* :: *comm-ring-1*
begin
  sublocale *comm-semiring-hom*..
end

locale *idom-hom* = *comm-ring-hom hom*
  for *hom* :: *'a* :: *idom* ⇒ *'b* :: *idom*

5.3 Division

locale *idom-divide-hom* = *idom-hom hom*
  for *hom* :: *'a* :: *idom-divide* ⇒ *'b* :: *idom-divide* +
  assumes *hom-div*[*hom-distribs*]: *hom* (*x div y*) = *hom x div hom y*
begin

end

locale *field-hom* = *idom-hom hom*
  for *hom* :: *'a* :: *field* ⇒ *'b* :: *field*
begin

  lemma *hom-inverse*[*hom-distribs*]: *hom* (*inverse x*) = *inverse* (*hom x*)
   by (*metis hom-mult hom-one hom-zero inverse-unique inverse-zero right-inverse*)

  sublocale *idom-divide-hom hom*
  proof
    fix *x y*
    have *hom* (*x* / *y*) = *hom* (*x* ∗ *inverse y*) **by** (*simp add*: *field-simps*)
    thus *hom* (*x* / *y*) = *hom x* / *hom y* **unfolding** *hom-distribs* **by** (*simp add*: *field-simps*)
  qed

end

locale *field-char-0-hom* = *field-hom hom*
  for *hom* :: *'a* :: *field-char-0* ⇒ *'b* :: *field-char-0*

5.4 (Partial) Injectivitiy

locale *zero-hom-0* = *zero-hom* +
  assumes *hom-0*: $\bigwedge x$. *hom x* = *0* $\Longrightarrow$ *x* = *0*
begin
  lemma *hom-0-iff* [*iff*]: *hom x* = *0* $\longleftrightarrow$ *x* = *0* **using** *hom-0* **by** *auto*
end

locale *one-hom-1* = *one-hom* +
  assumes *hom-1*: $\bigwedge x$. *hom x* = *1* $\Longrightarrow$ *x* = *1*
begin
  lemma *hom-1-iff* [*iff*]: *hom x* = *1* $\longleftrightarrow$ *x* = *1* **using** *hom-1* **by** *auto*
end

Next locales are at this point not interesting. They will retain some results when we think of polynomials.

locale *monoid-mult-hom-1* = *monoid-mult-hom* + *one-hom-1*

locale *monoid-add-hom-0* = *monoid-add-hom* + *zero-hom-0*

locale *comm-monoid-mult-hom-1* = *monoid-mult-hom-1 hom*
  for *hom* :: *'a* :: *comm-monoid-mult* $\Rightarrow$ *'b* :: *comm-monoid-mult*

locale *comm-monoid-add-hom-0* = *monoid-add-hom-0 hom*
  for *hom* :: *'a* :: *comm-monoid-add* $\Rightarrow$ *'b* :: *comm-monoid-add*

locale *injective* =
  fixes *f* :: *'a* $\Rightarrow$ *'b* **assumes** *injectivity*: $\bigwedge x\ y$. *f x* = *f y* $\Longrightarrow$ *x* = *y*
begin
  lemma *eq-iff* [*simp*]: *f x* = *f y* $\longleftrightarrow$ *x* = *y* **using** *injectivity* **by** *auto*
  lemma *inj-f*: *inj f* **by** (*auto intro*: *injI*)
  lemma *inv-f-f* [*simp*]: *inv f* (*f x*) = *x* **by** (*fact inv-f-f* [*OF inj-f*])
end

locale *inj-zero-hom* = *zero-hom* + *injective hom*
begin
  sublocale *zero-hom-0* **by** (*unfold-locales*, *auto intro*: *injectivity*)
end

locale *inj-one-hom* = *one-hom* + *injective hom*
begin
  sublocale *one-hom-1* **by** (*unfold-locales*, *auto intro*: *injectivity*)
end

locale *inj-semigroup-mult-hom* = *semigroup-mult-hom* + *injective hom*

locale *inj-semigroup-add-hom* = *semigroup-add-hom* + *injective hom*

locale *inj-monoid-mult-hom* = *monoid-mult-hom* + *inj-semigroup-mult-hom*

begin
  sublocale *inj-one-hom*..
  sublocale *monoid-mult-hom-1*..
end

locale *inj-monoid-add-hom* = *monoid-add-hom* + *inj-semigroup-add-hom*
begin
  sublocale *inj-zero-hom*..
  sublocale *monoid-add-hom-0*..
end

locale *inj-comm-monoid-mult-hom* = *comm-monoid-mult-hom* + *inj-monoid-mult-hom*
begin
  sublocale *comm-monoid-mult-hom-1*..
end

locale *inj-comm-monoid-add-hom* = *comm-monoid-add-hom* + *inj-monoid-add-hom*
begin
  sublocale *comm-monoid-add-hom-0*..
end

locale *inj-semiring-hom* = *semiring-hom* + *injective hom*
begin
  sublocale *inj-comm-monoid-add-hom* + *inj-monoid-mult-hom*..
end

locale *inj-comm-semiring-hom* = *comm-semiring-hom* + *inj-semiring-hom*
begin
  sublocale *inj-comm-monoid-mult-hom*..
end

For groups, injectivity is easily ensured.

locale *inj-group-add-hom* = *group-add-hom* + *zero-hom-0*
begin
  sublocale *injective hom*
  proof
    fix *x y* **assume** *hom x* = *hom y*
    then have *hom* (*x*−*y*) = *0* **by** (*auto simp*: *hom-distribs*)
    then show *x* = *y* **by** *simp*
  qed
  sublocale *inj-monoid-add-hom*..
end

locale *inj-ab-group-add-hom* = *ab-group-add-hom* + *inj-group-add-hom*
begin
  sublocale *inj-comm-monoid-add-hom*..
end

locale *inj-ring-hom* = *ring-hom* + *zero-hom-0*

```
begin
  sublocale inj-ab-group-add-hom..
  sublocale inj-semiring-hom..
end

locale inj-comm-ring-hom = comm-ring-hom + zero-hom-0
begin
  sublocale inj-ring-hom..
  sublocale inj-comm-semiring-hom..
end

locale inj-idom-hom = idom-hom + zero-hom-0
begin
  sublocale inj-comm-ring-hom..
end
```

Field homomorphism is always injective.

```
context field-hom begin
  sublocale zero-hom-0
  proof (unfold-locales, rule ccontr)
    fix x
    assume hom x = 0 and x0: x ≠ 0
    then have inverse (hom x) = 0 by simp
    then have hom (inverse x) = 0 by (simp add: hom-distribs)
    then have hom (inverse x * x) = 0 by (simp add: hom-distribs)
    with x0 have hom 1 = hom 0 by simp
    then have (1 :: 'b) = 0 by simp
    then show False by auto
  qed
  sublocale inj-idom-hom..
end
```

5.5 Surjectivity and Isomorphisms

```
locale surjective =
  fixes f :: 'a ⇒ 'b
  assumes surj: surj f
begin
  lemma f-inv-f[simp]: f (inv f x) = x
    by (rule cong, auto simp: surj[unfolded surj-iff o-def id-def])
end

locale bijective = injective + surjective

lemma bijective-eq-bij: bijective f = bij f
proof(intro iffI)
  assume bijective f
  then interpret bijective f.
  show bij f using injectivity surj by (auto intro!: bijI injI)
next
```

```
  assume bij f
  from this[unfolded bij-def]
  show bijective f by (unfold-locales, auto dest: injD)
qed

context bijective
begin
  lemmas bij = bijective-axioms[unfolded bijective-eq-bij]
  interpretation inv: bijective inv f
    using bijective-axioms bij-imp-bij-inv by (unfold bijective-eq-bij)
  sublocale inv: surjective inv f..
  sublocale inv: injective inv f..
  lemma inv-inv-f-eq[simp]: inv (inv f) = f using inv-inv-eq[OF bij].
  lemma f-eq-iff[simp]: f x = y ⟷ x = inv f y by auto
  lemma inv-f-eq-iff[simp]: inv f x = y ⟷ x = f y by auto
end

locale monoid-mult-isom = inj-monoid-mult-hom + bijective hom
begin
  sublocale inv: bijective inv hom..
  sublocale inv: inj-monoid-mult-hom inv hom
  proof (unfold-locales)
    fix hx hy :: 'b
    from bij obtain x y where hx: hx = hom x and hy: hy = hom y by (meson
bij-pointE)
     show inv hom (hx*hy) = inv hom hx * inv hom hy by (unfold hx hy, fold
hom-mult, simp)
    have inv hom (hom 1) = 1 by (unfold inv-f-f, simp)
    then show inv hom 1 = 1 by simp
  qed
end

locale monoid-add-isom = inj-monoid-add-hom + bijective hom
begin
  sublocale inv: bijective inv hom..
  sublocale inv: inj-monoid-add-hom inv hom
  proof (unfold-locales)
    fix hx hy :: 'b
    from bij obtain x y where hx: hx = hom x and hy: hy = hom y by (meson
bij-pointE)
     show inv hom (hx+hy) = inv hom hx + inv hom hy by (unfold hx hy, fold
hom-add, simp)
    have inv hom (hom 0) = 0 by (unfold inv-f-f, simp)
    then show inv hom 0 = 0 by simp
  qed
end

locale comm-monoid-mult-isom = monoid-mult-isom hom
  for hom :: 'a :: comm-monoid-mult ⇒ 'b :: comm-monoid-mult
```

```
begin
  sublocale inv: monoid-mult-isom inv hom..
  sublocale inj-comm-monoid-mult-hom..

  lemma hom-dvd-hom[simp]: hom x dvd hom y ⟷ x dvd y
  proof
    assume hom x dvd hom y
    then obtain hz where hom y = hom x * hz by (elim dvdE)
    moreover obtain z where hz = hom z using bij by (elim bij-pointE)
    ultimately have hom y = hom (x * z) by (auto simp: hom-distribs)
    from this[unfolded eq-iff] have y = x * z.
    then show x dvd y by (intro dvdI)
  qed (rule hom-dvd)

  lemma hom-dvd-simp[simp]:
    shows hom x dvd y' ⟷ x dvd inv hom y'
    using hom-dvd-hom[of x inv hom y'] by simp

end

locale comm-monoid-add-isom = monoid-add-isom hom
  for hom :: 'a :: comm-monoid-add ⇒ 'b :: comm-monoid-add
begin
 sublocale inv: monoid-add-isom inv hom by (unfold-locales; simp add: hom-distribs)
  sublocale inj-comm-monoid-add-hom..
end

locale semiring-isom = inj-semiring-hom hom + bijective hom for hom
begin
 sublocale inv: inj-semiring-hom inv hom by (unfold-locales; simp add: hom-distribs)
  sublocale inv: bijective inv hom..
  sublocale monoid-mult-isom..
  sublocale comm-monoid-add-isom..
end

locale comm-semiring-isom = semiring-isom hom
  for hom :: 'a :: comm-semiring-1 ⇒ 'b :: comm-semiring-1
begin
 sublocale inv: semiring-isom inv hom by (unfold-locales; simp add: hom-distribs)
  sublocale comm-monoid-mult-isom..
  sublocale inj-comm-semiring-hom..
end

locale ring-isom = inj-ring-hom + surjective hom
begin
  sublocale semiring-isom..
  sublocale inv: inj-ring-hom inv hom by (unfold-locales; simp add: hom-distribs)
end
```

locale *comm-ring-isom* = *ring-isom hom*
  for *hom* :: *'a* :: *comm-ring-1* ⇒ *'b* :: *comm-ring-1*
begin
  sublocale *comm-semiring-isom*..
  sublocale *inj-comm-ring-hom*..
  sublocale *inv*: *ring-isom inv hom* **by** (*unfold-locales*; *simp add*: *hom-distribs*)
end

locale *idom-isom* = *comm-ring-isom* + *inj-idom-hom*
begin
 sublocale *inv*: *comm-ring-isom inv hom* **by** (*unfold-locales*; *simp add*: *hom-distribs*)
  sublocale *inv*: *inj-idom-hom inv hom*..
end

locale *field-isom* = *field-hom* + *surjective hom*
begin
  sublocale *idom-isom*..
  sublocale *inv*: *field-hom inv hom* **by** (*unfold-locales*; *simp add*: *hom-distribs*)
end

locale *inj-idom-divide-hom* = *idom-divide-hom hom* + *inj-idom-hom hom*
  for *hom* :: *'a* :: *idom-divide* ⇒ *'b* :: *idom-divide*
begin
lemma *hom-dvd-iff*[*simp*]: (*hom p dvd hom q*) = (*p dvd q*)
proof (*cases p* = *0*)
  case *False*
  show *?thesis*
  proof
    assume *hom p dvd hom q* **from** *this*[*unfolded dvd-def*] **obtain** *k* **where**
      id: *hom q* = *hom p* ∗ *k* **by** *auto*
    hence (*hom q div hom p*) = (*hom p* ∗ *k*) *div hom p* **by** *simp*
    also have ... = *k* **by** (*rule nonzero-mult-div-cancel-left*, *insert False*, *simp*)
    also have *hom q div hom p* = *hom* (*q div p*) **by** (*simp add*: *hom-div*)
    finally have *k* = *hom* (*q div p*) **by** *auto*
     from *id*[*unfolded this*] **have** *hom q* = *hom* (*p* ∗ (*q div p*)) **by** (*simp add*:
hom-mult)
    hence *q* = *p* ∗ (*q div p*) **by** *simp*
    thus *p dvd q* **unfolding** *dvd-def* ..
  qed *simp*
qed *simp*
end

context *field-hom*
begin
sublocale *inj-idom-divide-hom* ..
end

5.6 Example Interpretations

interpretation *of-int-hom*: *ring-hom of-int* **by** (*unfold-locales*, *auto*)
interpretation *of-int-hom*: *comm-ring-hom of-int* **by** (*unfold-locales*, *auto*)
interpretation *of-int-hom*: *idom-hom of-int* **by** (*unfold-locales*, *auto*)
interpretation *of-int-hom*: *inj-ring-hom of-int* :: *int* ⇒ *'a* :: {*ring-1*,*ring-char-0*}
by (*unfold-locales*, *auto*)
interpretation *of-int-hom*: *inj-comm-ring-hom of-int* :: *int* ⇒ *'a* :: {*comm-ring-1*,*ring-char-0*}
by (*unfold-locales*, *auto*)
interpretation *of-int-hom*: *inj-idom-hom of-int* :: *int* ⇒ *'a* :: {*idom*,*ring-char-0*}
by (*unfold-locales*, *auto*)

Somehow *of-rat* is defined only on *char-0*.

interpretation *of-rat-hom*: *field-char-0-hom of-rat*
by (*unfold-locales*, *auto simp*: *of-rat-add of-rat-mult of-rat-inverse of-rat-minus*)

interpretation *of-real-hom*: *inj-ring-hom of-real* **by** (*unfold-locales*, *auto*)
interpretation *of-real-hom*: *inj-comm-ring-hom of-real* **by** (*unfold-locales*, *auto*)
interpretation *of-real-hom*: *inj-idom-hom of-real* **by** (*unfold-locales*, *auto*)
interpretation *of-real-hom*: *field-hom of-real* **by** (*unfold-locales*, *auto*)
interpretation *of-real-hom*: *field-char-0-hom of-real* **by** (*unfold-locales*, *auto*)

Constant multiplication in a semiring is only a monoid homomorphism.

interpretation *mult-hom*: *comm-monoid-add-hom* $\lambda x.\ c * x$ **for** *c* :: *'a* :: *semiring-1*
by (*unfold-locales*, *auto simp*: *field-simps*)

end

6 Missing Unsorted

This theory contains several lemmas which might be of interest to the Isabelle distribution. For instance, we prove that $b^n \cdot n^k$ is bounded by a constant whenever $0 < b < 1$.

theory *Missing-Unsorted*
imports
HOL.Complex HOL−Computational-Algebra.Factorial-Ring
begin

lemma *bernoulli-inequality*:
assumes *x*: −*1* ≤ (*x* :: *'a* :: *linordered-field*)
shows *1* + *of-nat n* ∗ *x* ≤ (*1* + *x*) ^ *n*
proof (*induct n*)
case (*Suc n*)
have *1* + *of-nat* (*Suc n*) ∗ *x* = *1* + *x* + *of-nat n* ∗ *x* **by** (*simp add*: *field-simps*)
also have ... ≤ ... + *of-nat n* ∗ *x* ^ *2* **by** *simp*
also have ... = (*1* + *of-nat n* ∗ *x*) ∗ (*1* + *x*) **by** (*simp add*: *field-simps power2-eq-square*)

```
  also have ... ≤ (1 + x) ^ n * (1 + x)
    by (rule mult-right-mono[OF Suc], insert x, auto)
  also have ... = (1 + x) ^(Suc n) by simp
  finally show ?case .
qed simp

context
  fixes b :: 'a :: archimedean-field
  assumes b: 0 < b b < 1
begin
private lemma pow-one: b ^ x ≤ 1 using power-Suc-less-one[OF b, of x − 1] by
(cases x, auto)

private lemma pow-zero: 0 < b ^ x using b(1) by simp

lemma exp-tends-to-zero:
  assumes c: c > 0
  shows ∃ x. b ^ x ≤ c
proof (rule ccontr)
  assume not: ¬ ?thesis
  define bb where bb = inverse b
  define cc where cc = inverse c
  from b have bb: bb > 1 unfolding bb-def by (rule one-less-inverse)
  from c have cc: cc > 0 unfolding cc-def by simp
  define bbb where bbb = bb − 1
  have id: bb = 1 + bbb and bbb: bbb > 0 and bm1: bbb ≥ −1 unfolding bbb-def
using bb by auto
  have ∃ n. cc / bbb < of-nat n by (rule reals-Archimedean2)
  then obtain n where lt: cc / bbb < of-nat n by auto
  from not have ¬ b ^ n ≤ c by auto
  hence bnc: b ^ n > c by simp
  have bb ^ n = inverse (b ^ n) unfolding bb-def by (rule power-inverse)
  also have ... < cc unfolding cc-def
    by (rule less-imp-inverse-less[OF bnc c])
  also have ... < bbb * of-nat n using lt bbb by (metis mult.commute pos-divide-less-eq)
  also have ... ≤ bb ^ n
    using bernoulli-inequality[OF bm1, folded id, of n] by (simp add: ac-simps)
  finally show False by simp
qed

lemma linear-exp-bound: ∃ p. ∀ x. b ^ x * of-nat x ≤ p
proof −
  from b have 1 − b > 0 by simp
  from exp-tends-to-zero[OF this]
  obtain x0 where x0: b ^ x0 ≤ 1 − b ..
  {
    fix x
    assume x ≥ x0
    hence ∃ y. x = x0 + y by arith
```

```
    then obtain y where x: x = x0 + y by auto
    have b ^ x = b ^ x0 * b ^ y unfolding x by (simp add: power-add)
    also have ... ≤ b ^ x0 using pow-one[of y] pow-zero[of x0] by auto
    also have ... ≤ 1 − b by (rule x0)
    finally have b ^ x ≤ 1 − b .
  } note x0 = this
  define bs where bs = insert 1 { b ^ Suc x * of-nat (Suc x) | x . x ≤ x0}
  have bs: finite bs unfolding bs-def by auto
  define p where p = Max bs
  have bs: ⋀ b. b ∈ bs ⟹ b ≤ p unfolding p-def using bs by simp
  hence p1: p ≥ 1 unfolding bs-def by auto
  show ?thesis
  proof (rule exI[of - p], intro allI)
    fix x
    show b ^ x * of-nat x ≤ p
    proof (induct x)
      case (Suc x)
      show ?case
      proof (cases x ≤ x0)
        case True
        show ?thesis
          by (rule bs, unfold bs-def, insert True, auto)
      next
        case False
        let ?x = of-nat x :: 'a
        have b ^(Suc x) * of-nat (Suc x) = b * (b ^ x * ?x) + b ^ Suc x by (simp
add: field-simps)
        also have ... ≤ b * p + b ^ Suc x
          by (rule add-right-mono[OF mult-left-mono[OF Suc]], insert b, auto)
        also have ... = p − ((1 − b) * p − b ^(Suc x)) by (simp add: field-simps)
        also have ... ≤ p − 0
        proof −
          have b ^ Suc x ≤ 1 − b using x0[of Suc x] False by auto
          also have ... ≤ (1 − b) * p using b p1 by auto
          finally show ?thesis
            by (intro diff-left-mono, simp)
        qed
        finally show ?thesis by simp
      qed
    qed (insert p1, auto)
  qed
qed

lemma poly-exp-bound: ∃ p. ∀ x. b ^ x * of-nat x ^ deg ≤ p
proof −
  show ?thesis
  proof (induct deg)
    case 0
    show ?case
```

```
      by (rule exI[of - 1], intro allI, insert pow-one, auto)
  next
    case (Suc deg)
    then obtain q where IH: ⋀ x. b ^ x * (of-nat x) ^ deg ≤ q by auto
    define p where p = max 0 q
    from IH have IH: ⋀ x. b ^ x * (of-nat x) ^ deg ≤ p unfolding p-def using
le-max-iff-disj by blast
    have p: p ≥ 0 unfolding p-def by simp
    show ?case
    proof (cases deg = 0)
      case True
      thus ?thesis using linear-exp-bound by simp
    next
      case False note deg = this
      define p' where p' = p*p * 2 ^ Suc deg * inverse b
      let ?f = λ x. b ^ x * (of-nat x) ^ Suc deg
      define f where f = ?f
      {
        fix x
        let ?x = of-nat x :: 'a
        have f (2 * x) ≤ (2 ^ Suc deg) * (p * p)
        proof (cases x = 0)
          case False
          hence x1: ?x ≥ 1 by (cases x, auto)
          from x1 have x: ?x ^(deg − 1) ≥ 1 by simp
          from x1 have xx: ?x ^ Suc deg ≥ 1 by (rule one-le-power)
          define c where c = b ^ x * b ^ x * (2 ^ Suc deg)
          have c: c > 0 unfolding c-def using b by auto
          have f (2 * x) = ?f (2 * x) unfolding f-def by simp
          also have b ^(2 * x) = (b ^ x) * (b ^ x) by (simp add: power2-eq-square
power-even-eq)
          also have of-nat (2 * x) = 2 * ?x by simp
          also have (2 * ?x) ^ Suc deg = 2 ^ Suc deg * ?x ^ Suc deg by simp
          finally have f (2 * x) = c * ?x ^ Suc deg unfolding c-def by (simp add:
ac-simps)
          also have ... ≤ c * ?x ^ Suc deg * ?x ^(deg − 1)
          proof −
            have c * ?x ^ Suc deg > 0 using c xx by simp
            thus ?thesis unfolding mult-le-cancel-left1 using x by simp
          qed
          also have ... = c * ?x ^(Suc deg + (deg − 1)) by (simp add: power-add)
          also have Suc deg + (deg − 1) = deg + deg using deg by simp
            also have ?x ^ (deg + deg) = (?x ^ deg) * (?x ^ deg) by (simp add:
power-add)
          also have c * ... = (2 ^ Suc deg) * ((b ^ x * ?x ^ deg) * (b ^ x * ?x ^
deg))
            unfolding c-def by (simp add: ac-simps)
          also have ... ≤ (2 ^ Suc deg) * (p * p)
            by (rule mult-left-mono[OF mult-mono[OF IH IH p]], insert pow-zero[of
```

```
x], auto)
          finally show f (2 * x) ≤ (2 ^ Suc deg) * (p * p) .
        qed (auto simp: f-def)
        hence ?f (2 * x) ≤ (2 ^ Suc deg) * (p * p) unfolding f-def .
      } note even = this
      show ?thesis
      proof (rule exI[of - p'], intro allI)
        fix y
        show ?f y ≤ p'
        proof (cases even y)
          case True
          define x where x = y div 2
          have y = 2 * x unfolding x-def using True by simp
          from even[of x, folded this] have ?f y ≤ 2 ^ Suc deg * (p * p) .
          also have ... ≤ ... * inverse b
            unfolding mult-le-cancel-left1 using b p
            by (simp add: algebra-split-simps one-le-inverse)
          also have ... = p' unfolding p'-def by (simp add: ac-simps)
          finally show ?f y ≤ p' .
        next
          case False
          define x where x = y div 2
          have y = 2 * x + 1 unfolding x-def using False by simp
          hence ?f y = ?f (2 * x + 1) by simp
          also have ... ≤ b ^(2 * x + 1) * of-nat (2 * x + 2) ^ Suc deg
            by (rule mult-left-mono[OF power-mono], insert b, auto)
          also have b ^(2 * x + 1) = b ^(2 * x + 2) * inverse b using b by auto
          also have b ^(2 * x + 2) * inverse b * of-nat (2 * x + 2) ^ Suc deg =
            inverse b * ?f (2 * (x + 1)) by (simp add: ac-simps)
          also have ... ≤ inverse b * ((2 ^ Suc deg) * (p * p))
            by (rule mult-left-mono[OF even], insert b, auto)
          also have ... = p' unfolding p'-def by (simp add: ac-simps)
          finally show ?f y ≤ p' .
        qed
      qed
    qed
  qed
qed
end

lemma prod-list-replicate[simp]: prod-list (replicate n a) = a ^ n
  by (induct n, auto)

lemma prod-list-power: fixes xs :: 'a :: comm-monoid-mult list
  shows prod-list xs ^ n = (∏ x←xs. x ^ n)
  by (induct xs, auto simp: power-mult-distrib)

lemma set-upt-Suc: {0 ..< Suc i} = insert i {0 ..< i}
  by (fact atLeast0-lessThan-Suc)
```

lemma *prod-pow*[*simp*]: ($\prod$ *i* = *0*..<*n*. *p*) = (*p* :: *'a* :: *comm-monoid-mult*) ^ *n*
  by *simp*

lemma *dvd-abs-mult-left-int* [*simp*]:
  |*a*| ∗ *y* *dvd* *x* ⟷ *a* ∗ *y* *dvd* *x* **for** *x* *y* *a* :: *int*
  using *abs-dvd-iff* [*of* *a* ∗ *y*] *abs-dvd-iff* [*of* |*a*| ∗ *y*]
  by (*simp* *add*: *abs-mult*)

lemma *gcd-abs-mult-right-int* [*simp*]:
  gcd *x* (|*a*| ∗ *y*) = *gcd* *x* (*a* ∗ *y*) **for** *x* *y* *a* :: *int*
  using *gcd-abs2-int* [*of* - *a* ∗ *y*] *gcd-abs2-int* [*of* - |*a*| ∗ *y*]
  by (*simp* *add*: *abs-mult*)

lemma *lcm-abs-mult-right-int* [*simp*]:
  lcm *x* (|*a*| ∗ *y*) = *lcm* *x* (*a* ∗ *y*) **for** *x* *y* *a* :: *int*
  using *lcm-abs2-int* [*of* - *a* ∗ *y*] *lcm-abs2-int* [*of* - |*a*| ∗ *y*]
  by (*simp* *add*: *abs-mult*)

lemma *gcd-abs-mult-left-int* [*simp*]:
  gcd *x* (*a* ∗ |*y*|) = *gcd* *x* (*a* ∗ *y*) **for** *x* *y* *a* :: *int*
  using *gcd-abs2-int* [*of* - *a* ∗ |*y*|] *gcd-abs2-int* [*of* - *a* ∗ *y*]
  by (*simp* *add*: *abs-mult*)

lemma *lcm-abs-mult-left-int* [*simp*]:
  lcm *x* (*a* ∗ |*y*|) = *lcm* *x* (*a* ∗ *y*) **for** *x* *y* *a* :: *int*
  using *lcm-abs2-int* [*of* - *a* ∗ |*y*|] *lcm-abs2-int* [*of* - *a* ∗ *y*]
  by (*simp* *add*: *abs-mult*)

abbreviation (*input*) *list-gcd* :: *'a* :: *semiring-gcd* *list* ⇒ *'a* **where**
  list-gcd ≡ *gcd-list*

abbreviation (*input*) *list-lcm* :: *'a* :: *semiring-gcd* *list* ⇒ *'a* **where**
  list-lcm ≡ *lcm-list*

lemma *list-gcd-simps*: *list-gcd* [] = *0* *list-gcd* (*x* # *xs*) = *gcd* *x* (*list-gcd* *xs*)
  by *simp-all*

lemma *list-gcd*: *x* ∈ *set* *xs* ⟹ *list-gcd* *xs* *dvd* *x*
  by (*fact* *Gcd-fin-dvd*)

lemma *list-gcd-greatest*: ($\bigwedge$ *x*. *x* ∈ *set* *xs* ⟹ *y* *dvd* *x*) ⟹ *y* *dvd* (*list-gcd* *xs*)
  by (*fact* *gcd-list-greatest*)

lemma *list-gcd-mult-int* [*simp*]:
  fixes *xs* :: *int list*
  shows *list-gcd* (*map* (*times a*) *xs*) = |*a*| ∗ *list-gcd xs*
  by (*simp add*: *Gcd-mult abs-mult*)

lemma *list-lcm-simps*: *list-lcm* [] = *1* *list-lcm* (*x* # *xs*) = *lcm x* (*list-lcm xs*)
  by *simp-all*

lemma *list-lcm*: *x* ∈ *set xs* ⟹ *x dvd list-lcm xs*
  by (*fact dvd-Lcm-fin*)

lemma *list-lcm-least*: (⋀ *x*. *x* ∈ *set xs* ⟹ *x dvd y*) ⟹ *list-lcm xs dvd y*
  by (*fact lcm-list-least*)

lemma *lcm-mult-distrib-nat*: (*k* :: *nat*) ∗ *lcm m n* = *lcm* (*k* ∗ *m*) (*k* ∗ *n*)
  by (*simp add*: *lcm-mult-left*)

lemma *lcm-mult-distrib-int*: *abs* (*k*::*int*) ∗ *lcm m n* = *lcm* (*k* ∗ *m*) (*k* ∗ *n*)
  by (*simp add*: *lcm-mult-left abs-mult*)

lemma *list-lcm-mult-int* [*simp*]:
  fixes *xs* :: *int list*
  shows *list-lcm* (*map* (*times a*) *xs*) = (*if xs* = [] *then 1 else* |*a*| ∗ *list-lcm xs*)
  by (*simp add*: *Lcm-mult abs-mult*)

lemma *list-lcm-pos*:
  list-lcm xs ≥ (*0* :: *int*)
  0 ∉ *set xs* ⟹ *list-lcm xs* ≠ *0*
  0 ∉ *set xs* ⟹ *list-lcm xs* > *0*
proof −
  have *0* ≤ |*Lcm* (*set xs*)|
    by (*simp only*: *abs-ge-zero*)
  then have *0* ≤ *Lcm* (*set xs*)
    by *simp*
  then show *list-lcm xs* ≥ *0*
    by *simp*
  assume *0* ∉ *set xs*
  then show *list-lcm xs* ≠ *0*
    by (*simp add*: *Lcm-0-iff*)
  with ‹*list-lcm xs* ≥ *0*› **show** *list-lcm xs* > *0*
    by (*simp add*: *le-less*)
qed

lemma *quotient-of-nonzero*: *snd* (*quotient-of r*) > *0* *snd* (*quotient-of r*) ≠ *0*
  using *quotient-of-denom-pos'* [*of r*] **by** *simp-all*

lemma *quotient-of-int-div*:
  assumes *q*: *quotient-of* (*of-int x* / *of-int y*) = (*a*, *b*)
  and *y*: *y* ≠ *0*

```
  shows ∃ z. z ≠ 0 ∧ x = a * z ∧ y = b * z
proof −
  let ?r = rat-of-int
  define z where z = gcd x y
  define x' where x' = x div z
  define y' where y' = y div z
  have id: x = z * x' y = z * y' unfolding x'-def y'-def z-def by auto
  from y have y': y' ≠ 0 unfolding id by auto
  have z: z ≠ 0 unfolding z-def using y by auto
  have cop: coprime x' y' unfolding x'-def y'-def z-def
    using div-gcd-coprime y by blast
   have ?r x / ?r y = ?r x' / ?r y' unfolding id using z y y' by (auto simp:
field-simps)
 from assms[unfolded this] have quot: quotient-of (?r x' / ?r y') = (a, b) by auto
  from quotient-of-coprime[OF quot] have cop': coprime a b .
  hence cop: coprime b a
    by (simp add: ac-simps)
  from quotient-of-denom-pos[OF quot] have b: b > 0 b ≠ 0 by auto
  from quotient-of-div[OF quot] quotient-of-denom-pos[OF quot] y'
  have ?r x' * ?r b = ?r a * ?r y' by (auto simp: field-simps)
  hence id': x' * b = a * y' unfolding of-int-mult[symmetric] by linarith
  from id'[symmetric] have b dvd y' * a unfolding mult.commute[of y'] by auto
  with cop y' have b dvd y'
    by (simp add: coprime-dvd-mult-left-iff)
  then obtain z' where ybz: y' = b * z' unfolding dvd-def by auto
  from id[unfolded y' this] have y: y = b * (z * z') by auto
  with ‹y ≠ 0› have zz: z * z' ≠ 0 by auto
  from quotient-of-div[OF q] ‹y ≠ 0› ‹b ≠ 0›
  have ?r x * ?r b = ?r y * ?r a by (auto simp: field-simps)
  hence id': x * b = y * a unfolding of-int-mult[symmetric] by linarith
  from this[unfolded y] b have x: x = a * (z * z') by auto
  show ?thesis unfolding x y using zz by blast
qed

fun max-list-non-empty :: ('a :: linorder) list ⇒ 'a where
  max-list-non-empty [x] = x
| max-list-non-empty (x # xs) = max x (max-list-non-empty xs)

lemma max-list-non-empty: x ∈ set xs ⟹ x ≤ max-list-non-empty xs
proof (induct xs)
  case (Cons y ys) note oCons = this
  show ?case
  proof (cases ys)
    case (Cons z zs)
     hence id: max-list-non-empty (y # ys) = max y (max-list-non-empty ys) by
simp
    from oCons show ?thesis unfolding id by (auto simp: max.coboundedI2)
  qed (insert oCons, auto)
qed simp
```

```
lemma cnj-reals[simp]: (cnj c ∈ ℝ) = (c ∈ ℝ)
  using Reals-cnj-iff by fastforce

lemma sgn-real-mono: x ≤ y ⟹ sgn x ≤ sgn (y :: real)
  unfolding sgn-real-def
  by (auto split: if-splits)

lemma sgn-minus-rat: sgn (− (x :: rat)) = − sgn x
  by (fact Rings.sgn-minus)

lemma real-of-rat-sgn: sgn (of-rat x) = real-of-rat (sgn x)
  unfolding sgn-real-def sgn-rat-def by auto

lemma inverse-le-iff-sgn:
  assumes sgn: sgn x = sgn y
  shows (inverse (x :: real) ≤ inverse y) = (y ≤ x)
proof (cases x = 0)
  case True
  with sgn have sgn y = 0 by simp
  hence y = 0 unfolding sgn-real-def by (cases y = 0; cases y < 0; auto)
  thus ?thesis using True by simp
next
  case False note x = this
  show ?thesis
  proof (cases x < 0)
    case True
    with x sgn have sgn y = −1 by simp
    hence y < 0 unfolding sgn-real-def by (cases y = 0; cases y < 0, auto)
    show ?thesis
      by (rule inverse-le-iff-le-neg[OF True ‹y < 0›])
  next
    case False
    with x have x: x > 0 by auto
    with sgn have sgn y = 1 by auto
    hence y > 0 unfolding sgn-real-def by (cases y = 0; cases y < 0, auto)
    show ?thesis
      by (rule inverse-le-iff-le[OF x ‹y > 0›])
  qed
qed

lemma inverse-le-sgn:
  assumes sgn: sgn x = sgn y and xy: x ≤ (y :: real)
  shows inverse y ≤ inverse x
  using xy inverse-le-iff-sgn[OF sgn] by auto

lemma set-list-update: set (xs [i := k]) =
  (if i < length xs then insert k (set (take i xs) ∪ set (drop (Suc i) xs)) else set xs)
proof (induct xs arbitrary: i)
```

case (*Cons x xs i*)
thus *?case*
by (*cases i*, *auto*)
qed *simp*

lemma *prod-list-dvd*: **assumes** (*x* :: *'a* :: *comm-monoid-mult*) ∈ *set xs*
shows *x dvd prod-list xs*
proof −
from *assms*[*unfolded in-set-conv-decomp*] **obtain** *ys zs* **where** *xs*: *xs* = *ys* @ *x* # *zs* **by** *auto*
show *?thesis* **unfolding** *xs dvd-def* **by** (*intro exI*[*of - prod-list* (*ys* @ *zs*)], *simp add*: *ac-simps*)
qed

lemma *dvd-prod*:
fixes *A*::*'b set*
assumes ∃ *b*∈*A*. *a dvd f b finite A*
shows *a dvd prod f A*
using *assms*(*2*,*1*)
proof (*induct A*)
case (*insert x A*)
thus *?case*
using *comm-monoid-mult-class.dvd-mult dvd-mult2 insert-iff prod.insert* **by** *auto*
qed *auto*

context
fixes *xs* :: *'a* :: *comm-monoid-mult list*
begin
lemma *prod-list-filter*: *prod-list* (*filter f xs*) ∗ *prod-list* (*filter* (λ *x*. ¬ *f x*) *xs*) = *prod-list xs*
by (*induct xs*, *auto simp*: *ac-simps*)

lemma *prod-list-partition*: **assumes** *partition f xs* = (*ys*, *zs*)
shows *prod-list xs* = *prod-list ys* ∗ *prod-list zs*
using *assms* **by** (*subst prod-list-filter*[*symmetric*, *of f*], *auto simp*: *o-def*)
end

lemma *dvd-imp-mult-div-cancel-left*[*simp*]:
assumes (*a* :: *'a* :: *semidom-divide*) *dvd b*
shows *a* ∗ (*b div a*) = *b*
proof(*cases b* = *0*)
case *True* **then show** *?thesis* **by** *auto*
next
case *False*
with *dvdE*[*OF assms*] **obtain** *c* **where** ∗: *b* = *a* ∗ *c* **by** *auto*
also with *False* **have** *a* ≠ *0* **by** *auto*
then have *a* ∗ *c div a* = *c* **by** *auto*
also note ∗[*symmetric*]

finally show *?thesis***.**
qed

lemma (**in** *semidom*) *prod-list-zero-iff* [*simp*]:
prod-list xs = *0* ⟷ *0* ∈ *set xs* **by** (*induction xs*, *auto*)

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

lemma *unit-prod* [*intro*]:
shows *a dvd 1* ⟹ *b dvd 1* ⟹ (*a* ∗ *b*) *dvd 1*
by (*subst mult-1-left* [*of 1*, *symmetric*]) (*rule mult-dvd-mono*)

lemma *is-unit-mult-iff* [*simp*]:
shows (*a* ∗ *b*) *dvd 1* ⟷ *a dvd 1* ∧ *b dvd 1*
by (*auto dest*: *dvd-mult-left dvd-mult-right*)

end

context *comm-semiring-1*
begin
lemma *irreducibleE*[*elim*]:
assumes *irreducible p*
and *p* ≠ *0* ⟹ ¬ *p dvd 1* ⟹ (⋀*a b*. *p* = *a* ∗ *b* ⟹ *a dvd 1* ∨ *b dvd 1*) ⟹ *thesis*
shows *thesis* **using** *assms* **by** (*auto simp*: *irreducible-def*)

lemma *not-irreducibleE*:
assumes ¬ *irreducible x*
and *x* = *0* ⟹ *thesis*
and *x dvd 1* ⟹ *thesis*
and ⋀*a b*. *x* = *a* ∗ *b* ⟹ ¬ *a dvd 1* ⟹ ¬ *b dvd 1* ⟹ *thesis*
shows *thesis* **using** *assms* **unfolding** *irreducible-def* **by** *auto*

lemma *prime-elem-dvd-prod-list*:
assumes *p*: *prime-elem p* **and** *pA*: *p dvd prod-list A* **shows** ∃ *a* ∈ *set A*. *p dvd a*
proof(*insert pA*, *induct A*)
case *Nil*
with *p* **show** *?case* **by** (*simp add*: *prime-elem-not-unit*)
next
case (*Cons a A*)
then show *?case* **by** (*auto simp*: *prime-elem-dvd-mult-iff* [*OF p*])
qed

lemma *prime-elem-dvd-prod-mset*:
assumes *p*: *prime-elem p* **and** *pA*: *p dvd prod-mset A* **shows** ∃ *a* ∈# *A*. *p dvd a*
proof(*insert pA*, *induct A*)
case *empty*
with *p* **show** *?case* **by** (*simp add*: *prime-elem-not-unit*)

next
  case (*add a A*)
  then show *?case* **by** (*auto simp*: *prime-elem-dvd-mult-iff*[*OF p*])
qed

lemma *mult-unit-dvd-iff*[*simp*]:
  assumes *b dvd 1*
  shows *a* ∗ *b dvd c* ⟷ *a dvd c*
proof
  assume *a* ∗ *b dvd c*
  with *assms* **show** *a dvd c* **using** *dvd-mult-left*[*of a b c*] **by** *simp*
next
  assume *a dvd c*
  with *assms mult-dvd-mono* **show** *a* ∗ *b dvd c* **by** *fastforce*
qed

lemma *mult-unit-dvd-iff′*[*simp*]: *a dvd 1* ⟹ (*a* ∗ *b*) *dvd c* ⟷ *b dvd c*
  using *mult-unit-dvd-iff* [*of a b c*] **by** (*simp add*: *ac-simps*)

lemma *irreducibleD′*:
  assumes *irreducible a b dvd a*
  shows *a dvd b* ∨ *b dvd 1*
proof −
  from *assms* **obtain** *c* **where** *c*: *a* = *b* ∗ *c* **by** (*elim dvdE*)
  from *irreducibleD*[*OF assms*(*1*) *this*] **have** *b dvd 1* ∨ *c dvd 1* **.**
  thus *?thesis* **by** (*auto simp*: *c*)
qed

end

context *idom*
begin

Following lemmas are adapted and generalized so that they don't use "algebraic" classes.

lemma *dvd-times-left-cancel-iff* [*simp*]:
  assumes *a* ≠ *0*
  shows *a* ∗ *b dvd a* ∗ *c* ⟷ *b dvd c*
    (**is** *?lhs* ⟷ *?rhs*)
  using *assms local.dvd-mult-cancel-left* **by** *presburger*

lemma *dvd-times-right-cancel-iff* [*simp*]:
  assumes *a* ≠ *0*
  shows *b* ∗ *a dvd c* ∗ *a* ⟷ *b dvd c*
  using *assms local.dvd-mult-cancel-right* **by** *presburger*

lemma *irreducibleI′*:

assumes *a* ≠ *0* ¬ *a dvd 1* ⋀*b*. *b dvd a* ⟹ *a dvd b* ∨ *b dvd 1*
shows *irreducible a*
unfolding *irreducible-def*
by (*metis assms dvd-times-left-cancel-iff local.dvd-triv-left local.mult-cancel-left1*)

lemma *irreducible-altdef*:
shows *irreducible x* ⟷ *x* ≠ *0* ∧ ¬ *x dvd 1* ∧ (∀ *b*. *b dvd x* ⟶ *x dvd b* ∨ *b dvd 1*)
using *local.irreducibleD′ irreducibleI′ irreducible-def* **by** *blast*

lemma *dvd-mult-unit-iff*:
assumes *b*: *b dvd 1*
shows *a dvd c* ∗ *b* ⟷ *a dvd c*
proof−
from *b* **obtain** *b′* **where** *1*: *b* ∗ *b′* = *1* **by** (*elim dvdE*, *auto*)
then have *b0*: *b* ≠ *0* **by** *auto*
from *1* **have** *a* = (*a* ∗ *b′*) ∗ *b* **by** (*simp add*: *ac-simps*)
also have ... *dvd c* ∗ *b* ⟷ *a* ∗ *b′ dvd c* **using** *b0* **by** *auto*
finally show *?thesis* **by** (*auto intro*: *dvd-mult-left*)
qed

lemma *dvd-mult-unit-iff′*: *b dvd 1* ⟹ *a dvd b* ∗ *c* ⟷ *a dvd c*
using *dvd-mult-unit-iff* [*of b a c*] **by** (*simp add*: *ac-simps*)

lemma *irreducible-mult-unit-left*:
shows *a dvd 1* ⟹ *irreducible* (*a* ∗ *p*) ⟷ *irreducible p*
by (*auto simp*: *irreducible-altdef mult.commute*[*of a*] *dvd-mult-unit-iff*)

lemma *irreducible-mult-unit-right*:
shows *a dvd 1* ⟹ *irreducible* (*p* ∗ *a*) ⟷ *irreducible p*
by (*auto simp*: *irreducible-altdef mult.commute*[*of a*] *dvd-mult-unit-iff*)

lemma *prime-elem-imp-irreducible*:
assumes *prime-elem p*
shows *irreducible p*
proof (*rule irreducibleI*)
fix *a b*
assume *p-eq*: *p* = *a* ∗ *b*
with *assms* **have** *nz*: *a* ≠ *0* *b* ≠ *0* **by** *auto*
from *p-eq* **have** *p dvd a* ∗ *b* **by** *simp*
with ‹*prime-elem p*› **have** *p dvd a* ∨ *p dvd b* **by** (*rule prime-elem-dvd-multD*)
with ‹*p* = *a* ∗ *b*› **have** *a* ∗ *b dvd 1* ∗ *b* ∨ *a* ∗ *b dvd a* ∗ *1* **by** *auto*
with *nz* **show** *a dvd 1* ∨ *b dvd 1*
using *local.dvd-mult-cancel-right local.dvd-times-left-cancel-iff* **by** *blast*
qed (*insert assms*, *simp-all add*: *prime-elem-def*)

lemma *unit-imp-dvd* [*dest*]: *b dvd 1* ⟹ *b dvd a*
by (*rule dvd-trans* [*of - 1*]) *simp-all*

lemma *unit-mult-left-cancel*: *a dvd 1* $\Longrightarrow$ *a* $*$ *b* = *a* $*$ *c* $\longleftrightarrow$ *b* = *c*
  using *mult-cancel-left* [*of a b c*] **by** *auto*

lemma *unit-mult-right-cancel*: *a dvd 1* $\Longrightarrow$ *b* $*$ *a* = *c* $*$ *a* $\longleftrightarrow$ *b* = *c*
  using *unit-mult-left-cancel* [*of a b c*] **by** (*auto simp add*: *ac-simps*)

New parts from here

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

end

lemma (**in** *field*) *irreducible-field*[*simp*]:
  irreducible x $\longleftrightarrow$ *False* **by** (*auto simp*: *dvd-field-iff irreducible-def*)

lemma (**in** *idom*) *irreducible-mult*:
  shows *irreducible* (*a*$*$*b*) $\longleftrightarrow$ *a dvd 1* ∧ *irreducible b* ∨ *b dvd 1* ∧ *irreducible a*
  by (*auto dest*: *irreducible-multD simp*: *irreducible-mult-unit-left irreducible-mult-unit-right*)

end

7 Missing Polynomial

The theory contains some basic results on polynomials which have not been detected in the distribution, especially on linear factors and degrees.

theory *Missing-Polynomial*
imports
  HOL−Computational-Algebra.Polynomial-Factorial
  Missing-Unsorted

begin

A nice extension rule for polynomials.

declare *poly-ext*[*intro*]

7.1 Basic Properties

lemma *linear-poly-root*:
$(a :: 'a :: comm\text{-}ring\text{-}1) \in set\ as \Longrightarrow poly\ (\prod\ a \leftarrow as.\ [:-\ a,\ 1:])\ a = 0$
proof (*induct as*)
  case (*Cons b as*)
  show *?case*
  proof (*cases a = b*)
    case *False*
    with *Cons* **have** $a \in set\ as$ **by** *auto*
    from *Cons*(*1*)[*OF this*] **show** *?thesis* **by** *simp*
  qed *simp*
qed *simp*

lemma *degree-lcoeff-sum*: **assumes** *deg*: $degree\ (f\ q) = n$
  and *fin*: *finite S* **and** *q*: $q \in S$ **and** *degle*: $\bigwedge\ p\ .\ p \in S - \{q\} \Longrightarrow degree\ (f\ p) <$
n
  and *cong*: $coeff\ (f\ q)\ n = c$
  shows $degree\ (sum\ f\ S) = n \wedge coeff\ (sum\ f\ S)\ n = c$
proof ($cases\ S = \{q\}$)
  case *True*
  thus *?thesis* **using** *deg cong* **by** *simp*
next
  case *False*
  with *q* **obtain** *p* **where** $p \in S - \{q\}$ **by** *auto*
  from *degle*[*OF this*] **have** *n*: $n > 0$ **by** *auto*
  have $degree\ (sum\ f\ S) = degree\ (f\ q + sum\ f\ (S - \{q\}))$
    unfolding *sum.remove*[*OF fin q*] **..**
  also have $\ldots = degree\ (f\ q)$
  proof (*rule degree-add-eq-left*)
    have $degree\ (sum\ f\ (S - \{q\})) \le n - 1$
    proof (*rule degree-sum-le*)
      fix *p*
      show $p \in S - \{q\} \Longrightarrow degree\ (f\ p) \le n - 1$
        using *degle*[*of p*] **by** *auto*
    qed (*insert fin*, *auto*)
    also have $\ldots < n$ **using** *n* **by** *simp*
    finally show $degree\ (sum\ f\ (S - \{q\})) < degree\ (f\ q)$ **unfolding** *deg* **.**
  qed
  finally show *?thesis* **unfolding** *deg*[*symmetric*] *cong*[*symmetric*]
  proof
    have *id*: $(\sum x{\in}S - \{q\}.\ coeff\ (f\ x)\ (degree\ (f\ q))) = 0$
      by (*rule sum.neutral*, *rule ballI*, *rule coeff-eq-0*[*OF degle*[*folded deg*]])
    show $coeff\ (sum\ f\ S)\ (degree\ (f\ q)) = coeff\ (f\ q)\ (degree\ (f\ q))$
      unfolding *coeff-sum*

by (*subst sum.remove*[*OF - q*], *unfold id*, *insert fin*, *auto*)
qed
qed

lemma *poly-sum-list*: *poly* (*sum-list ps*) *x* = *sum-list* (*map* (λ *p*. *poly p x*) *ps*)
by (*induct ps*, *auto*)

lemma *poly-prod-list*: *poly* (*prod-list ps*) *x* = *prod-list* (*map* (λ *p*. *poly p x*) *ps*)
by (*induct ps*, *auto*)

lemma *sum-list-neutral*: (⋀ *x*. *x* ∈ *set xs* ⟹ *x* = *0*) ⟹ *sum-list xs* = *0*
by (*induct xs*) *auto*

lemma *prod-list-neutral*: (⋀ *x*. *x* ∈ *set xs* ⟹ *x* = *1*) ⟹ *prod-list xs* = *1*
by (*induct xs*) *auto*

lemma (**in** *comm-monoid-mult*) *prod-list-map-remove1*:
x ∈ *set xs* ⟹ *prod-list* (*map f xs*) = *f x* ∗ *prod-list* (*map f* (*remove1 x xs*))
by (*induct xs*) (*auto simp add*: *ac-simps*)

lemma *poly-as-sum*:
fixes *p* :: ′*a*::*comm-semiring-1 poly*
shows *poly p x* = (∑ *i*≤*degree p*. *x* ^ *i* ∗ *coeff p i*)
unfolding *poly-altdef* **by** (*simp add*: *ac-simps*)

lemma *poly-prod-0*: *finite ps* ⟹ *poly* (*prod f ps*) *x* = (*0* :: ′*a* :: *field*) ⟷ (∃ *p* ∈ *ps*. *poly* (*f p*) *x* = *0*)
by (*induct ps rule*: *finite-induct*, *auto*)

lemma *coeff-monom-mult*:
shows *coeff* (*monom a d* ∗ *p*) *i* =
(*if d* ≤ *i then a* ∗ *coeff p* (*i*−*d*) *else 0*) (**is** *?l* = *?r*)
proof (*cases d* ≤ *i*)
case *False* **thus** *?thesis* **unfolding** *coeff-mult* **by** *simp*
next case *True*
let *?f* = λ*j*. *coeff* (*monom a d*) *j* ∗ *coeff p* (*i* − *j*)
have ⋀*j*. *j* ∈ {*0*..*i*} − {*d*} ⟹ *?f j* = *0* **by** *auto*
hence *0* = (∑ *j* ∈ {*0*..*i*} − {*d*}. *?f j*) **by** *auto*
also have ... + *?f d* = (∑ *j* ∈ *insert d* ({*0*..*i*} − {*d*}). *?f j*)
by(*subst sum.insert*, *auto*)
also have ... = (∑ *j* ∈ {*0*..*i*}. *?f j*) **by** (*subst insert-Diff*, *insert True*, *auto*)
also have ... = (∑ *j*≤*i*. *?f j*) **by** (*rule sum.cong*, *auto*)
also have ... = *?l* **unfolding** *coeff-mult* **..**
finally show *?thesis* **using** *True* **by** *auto*
qed

7.2 Polynomial Composition

lemmas [*simp*] = *pcompose-pCons*

declare *degree-pcompose*[*simp*]

7.3 Monic Polynomials

abbreviation *monic* **where** *monic p* ≡ *coeff p* (*degree p*) = *1*

lemma *unit-factor-field* [*simp*]:
  unit-factor (*x* :: ′*a* :: {*field*,*normalization-semidom*}) = *x*
  by (*cases is-unit x*) (*auto simp*: *is-unit-unit-factor dvd-field-iff*)

lemma *poly-gcd-monic*:
  fixes *p* :: ′*a* :: {*field*,*factorial-ring-gcd*,*semiring-gcd-mult-normalize*} *poly*
  assumes *p* ≠ *0* ∨ *q* ≠ *0*
  shows *monic* (*gcd p q*)
proof −
  from *assms* **have** *1* = *unit-factor* (*gcd p q*) **by** (*auto simp*: *unit-factor-gcd*)
  also have ... = [:*lead-coeff* (*gcd p q*):] **unfolding** *unit-factor-poly-def*
    by (*simp add*: *monom-0*)
  finally show *?thesis*
    by (*metis coeff-pCons-0 degree-1 lead-coeff-1*)
qed

lemma *normalize-monic*: *monic p* ⟹ *normalize p* = *p*
  by (*simp add*: *normalize-poly-eq-map-poly is-unit-unit-factor*)

lemma *lcoeff-monic-mult*: **assumes** *monic*: *monic* (*p* :: ′*a* :: *comm-semiring-1 poly*)
  shows *coeff* (*p* ∗ *q*) (*degree p* + *degree q*) = *coeff q* (*degree q*)
proof −
  let *?pqi* = λ *i*. *coeff p i* ∗ *coeff q* (*degree p* + *degree q* − *i*)
  have *coeff* (*p* ∗ *q*) (*degree p* + *degree q*) =
    (∑ *i*≤*degree p* + *degree q*. *?pqi i*)
    unfolding *coeff-mult* **by** *simp*
  also have ... = *?pqi* (*degree p*) + (*sum ?pqi* ({.. *degree p* + *degree q*} − {*degree p*}))
    by (*subst sum.remove*[*of - degree p*], *auto*)
  also have *?pqi* (*degree p*) = *coeff q* (*degree q*) **unfolding** *monic* **by** *simp*
  also have (*sum ?pqi* ({.. *degree p* + *degree q*} − {*degree p*})) = *0*
  proof (*rule sum.neutral*, *intro ballI*)
    fix *d*
    assume *d*: *d* ∈ {.. *degree p* + *degree q*} − {*degree p*}
    show *?pqi d* = *0*
    proof (*cases d* < *degree p*)
      case *True*
      hence *degree p* + *degree q* − *d* > *degree q* **by** *auto*
      hence *coeff q* (*degree p* + *degree q* − *d*) = *0* **by** (*rule coeff-eq-0*)
      thus *?thesis* **by** *simp*
    next

case *False*
with *d* **have** *d* > *degree p* **by** *auto*
hence *coeff p d* = *0* **by** (*rule coeff-eq-0*)
thus *?thesis* **by** *simp*
qed
qed
finally show *?thesis* **by** *simp*
qed

lemma *degree-monic-mult*: **assumes** *monic*: *monic* (*p* :: ′*a* :: *comm-semiring-1 poly*)
and *q*: *q* ≠ *0*
shows *degree* (*p* ∗ *q*) = *degree p* + *degree q*
proof −
have *degree p* + *degree q* ≥ *degree* (*p* ∗ *q*) **by** (*rule degree-mult-le*)
also have *degree p* + *degree q* ≤ *degree* (*p* ∗ *q*)
proof −
from *q* **have** *cq*: *coeff q* (*degree q*) ≠ *0* **by** *auto*
hence *coeff* (*p* ∗ *q*) (*degree p* + *degree q*) ≠ *0* **unfolding** *lcoeff-monic-mult*[*OF monic*] **.**
thus *degree* (*p* ∗ *q*) ≥ *degree p* + *degree q* **by** (*rule le-degree*)
qed
finally show *?thesis* **.**
qed

lemma *degree-prod-sum-monic*: **assumes**
S: *finite S*
and *nzd*: *0* ∉ (*degree o f*) ‘ *S*
and *monic*: (⋀ *a* . *a* ∈ *S* ⟹ *monic* (*f a*))
shows *degree* (*prod f S*) = (*sum* (*degree o f*) *S*) ∧ *coeff* (*prod f S*) (*sum* (*degree o f*) *S*) = *1*
proof −
from *S nzd monic*
have *degree* (*prod f S*) = *sum* (*degree* ∘ *f*) *S*
∧ (*S* ≠ {} ⟶ *degree* (*prod f S*) ≠ *0* ∧ *prod f S* ≠ *0*) ∧ *coeff* (*prod f S*) (*sum* (*degree o f*) *S*) = *1*
proof (*induct S rule*: *finite-induct*)
case (*insert a S*)
have *IH1*: *degree* (*prod f S*) = *sum* (*degree o f*) *S*
using *insert* **by** *auto*
have *IH2*: *coeff* (*prod f S*) (*degree* (*prod f S*)) = *1*
using *insert* **by** *auto*
have *id*: *degree* (*prod f* (*insert a S*)) = *sum* (*degree* ∘ *f*) (*insert a S*)
∧ *coeff* (*prod f* (*insert a S*)) (*sum* (*degree o f*) (*insert a S*)) = *1*
proof (*cases S* = {})
case *False*
with *insert* **have** *nz*: *prod f S* ≠ *0* **by** *auto*
from *insert* **have** *monic*: *coeff* (*f a*) (*degree* (*f a*)) = *1* **by** *auto*
have *id*: (*degree* ∘ *f*) *a* = *degree* (*f a*) **by** *simp*

show *?thesis* **unfolding** *prod.insert*[*OF insert*(*1*−*2*)] *sum.insert*[*OF insert*(*1*−*2*)] *id*
unfolding *degree-monic-mult*[*OF monic nz*]
unfolding *IH1* [*symmetric*]
unfolding *lcoeff-monic-mult*[*OF monic*] *IH2* **by** *simp*
qed (*insert insert*, *auto*)
show *?case* **using** *id* **unfolding** *sum.insert*[*OF insert*(*1*−*2*)] **using** *insert* **by** *auto*
qed *simp*
thus *?thesis* **by** *auto*
qed

lemma *degree-prod-monic*:
assumes $\bigwedge$ *i*. *i* < *n* ⟹ *degree* (*f i* :: ′*a* :: *comm-semiring-1 poly*) = *1*
and $\bigwedge$ *i*. *i* < *n* ⟹ *coeff* (*f i*) *1* = *1*
shows *degree* (*prod f* {*0* ..< *n*}) = *n* ∧ *coeff* (*prod f* {*0* ..< *n*}) *n* = *1*
proof −
from *degree-prod-sum-monic*[*of* {*0* ..< *n*} *f*] **show** *?thesis* **using** *assms* **by** *force*
qed

lemma *degree-prod-sum-lt-n*: **assumes** $\bigwedge$ *i*. *i* < *n* ⟹ *degree* (*f i* :: ′*a* :: *comm-semiring-1 poly*) ≤ *1*
and *i*: *i* < *n* **and** *fi*: *degree* (*f i*) = *0*
shows *degree* (*prod f* {*0* ..< *n*}) < *n*
proof −
have *degree* (*prod f* {*0* ..< *n*}) ≤ *sum* (*degree o f*) {*0* ..< *n*}
by (*rule degree-prod-sum-le*, *auto*)
also have *sum* (*degree o f*) {*0* ..< *n*} = (*degree o f*) *i* + *sum* (*degree o f*) ({*0* ..< *n*} − {*i*})
by (*rule sum.remove*, *insert i*, *auto*)
also have (*degree o f*) *i* = *0* **using** *fi* **by** *simp*
also have *sum* (*degree o f*) ({*0* ..< *n*} − {*i*}) ≤ *sum* (λ -. *1*) ({*0* ..< *n*} − {*i*})
by (*rule sum-mono*, *insert assms*, *auto*)
also have . . . = *n* − *1* **using** *i* **by** *simp*
also have . . . < *n* **using** *i* **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *degree-linear-factors*: *degree* (∏ *a* ← *as*. [: *f a*, *1*:]) = *length as*
proof (*induct as*)
case (*Cons b as*) **note** *IH* = *this*
have *id*: (∏ *a*←*b* # *as*. [:*f a*, *1*:]) = [:*f b*,*1* :] ∗ (∏ *a*←*as*. [:*f a*, *1*:]) **by** *simp*
show *?case* **unfolding** *id*
by (*subst degree-monic-mult*, *insert IH*, *auto*)
qed *simp*

lemma *monic-mult*:
fixes *p q* :: ′*a* :: *idom poly*
assumes *monic p monic q*

shows *monic* (*p* ∗ *q*)
proof −
from *assms* **have** *nz*: *p* ≠ *0* *q* ≠ *0* **by** *auto*
show *?thesis* **unfolding** *degree-mult-eq*[*OF nz*] *coeff-mult-degree-sum*
using *assms* **by** *simp*
qed

lemma *monic-factor*:
fixes *p q* :: *'a* :: *idom poly*
assumes *monic* (*p* ∗ *q*) *monic p*
shows *monic q*
proof −
from *assms* **have** *nz*: *p* ≠ *0* *q* ≠ *0* **by** *auto*
from *assms*[*unfolded degree-mult-eq*[*OF nz*] *coeff-mult-degree-sum* ‹*monic p*›]
show *?thesis* **by** *simp*
qed

lemma *monic-prod*:
fixes *f* :: *'a* ⇒ *'b* :: *idom poly*
assumes ⋀ *a*. *a* ∈ *as* ⟹ *monic* (*f a*)
shows *monic* (*prod f as*) **using** *assms*
proof (*induct as rule*: *infinite-finite-induct*)
case (*insert a as*)
hence *id*: *prod f* (*insert a as*) = *f a* ∗ *prod f as*
and ∗: *monic* (*f a*) *monic* (*prod f as*) **by** *auto*
show *?case* **unfolding** *id* **by** (*rule monic-mult*[*OF* ∗])
qed *auto*

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

lemma *monic-power*:
assumes *monic* (*p* :: *'a* :: *idom poly*)
shows *monic* (*p* ^ *n*)
by (*induct n*, *insert assms*, *auto intro*: *monic-mult*)

lemma *monic-prod-list-pow*: *monic* (∏ (*x*::*'a*::*idom*, *i*)←*xis*. [:− *x*, *1*:] ^ *Suc i*)
proof (*rule monic-prod-list*, *goal-cases*)
case (*1 a*)
then obtain *x i* **where** *a*: *a* = [:−*x*, *1*:]^*Suc i* **by** *force*
show *monic a* **unfolding** *a*
by (*rule monic-power*, *auto*)
qed

lemma *monic-degree-0*: *monic p* ⟹ (*degree p* = *0*) = (*p* = *1*)
using *le-degree poly-eq-iff* **by** *force*

7.4 Roots

The following proof structure is completely similar to the one of *?p* ≠ *0* ⟹ *finite* {*x*. *poly ?p x* = *0*}.

lemma *poly-roots-degree*:
  fixes *p* :: *'a*::*idom poly*
  shows *p* ≠ *0* ⟹ *card* {*x*. *poly p x* = *0*} ≤ *degree p*
proof (*induct n* ≡ *degree p arbitrary*: *p*)
  case (*0 p*)
  then obtain *a* **where** *a* ≠ *0* **and** *p* = [:*a*:]
    by (*cases p*, *simp split*: *if-splits*)
  then show *?case* **by** *simp*
next
  case (*Suc n p*)
  show *?case*
  proof (*cases* ∃ *x*. *poly p x* = *0*)
    case *True*
    then obtain *a* **where** *a*: *poly p a* = *0* **..**
    then have [:−*a*, *1*:] *dvd p* **by** (*simp only*: *poly-eq-0-iff-dvd*)
    then obtain *k* **where** *k*: *p* = [:−*a*, *1*:] ∗ *k* **..**
    with ‹*p* ≠ *0*› **have** *k* ≠ *0* **by** *auto*
    with *k* **have** *degree p* = *Suc* (*degree k*)
      by (*simp add*: *degree-mult-eq del*: *mult-pCons-left*)
    with ‹*Suc n* = *degree p*› **have** *n* = *degree k* **by** *simp*
    from *Suc.hyps*(*1*)[*OF this* ‹*k* ≠ *0*›]
    have *le*: *card* {*x*. *poly k x* = *0*} ≤ *degree k* **.**
    have *card* {*x*. *poly p x* = *0*} = *card* {*x*. *poly* ([:−*a*, *1*:] ∗ *k*) *x* = *0*} **unfolding** *k* **..**
    also have {*x*. *poly* ([:−*a*, *1*:] ∗ *k*) *x* = *0*} = *insert a* {*x*. *poly k x* = *0*}
      by *auto*
    also have *card* ... ≤ *Suc* (*card* {*x*. *poly k x* = *0*})
      unfolding *card-insert-if*[*OF poly-roots-finite*[*OF* ‹*k* ≠ *0*›]] **by** *simp*
    also have ... ≤ *Suc* (*degree k*) **using** *le* **by** *auto*
    finally show *?thesis* **using** ‹*degree p* = *Suc* (*degree k*)› **by** *simp*
  qed *simp*
qed

lemma *poly-root-factor*: (*poly* ([: *r*, *1*:] ∗ *q*) (*k* :: *'a* :: *idom*) = *0*) = (*k* = −*r* ∨ *poly q k* = *0*) (**is** *?one*)
  (*poly* (*q* ∗ [: *r*, *1*:]) *k* = *0*) = (*k* = −*r* ∨ *poly q k* = *0*) (**is** *?two*)
  (*poly* [: *r*, *1* :] *k* = *0*) = (*k* = −*r*) (**is** *?three*)
proof −
  have [*simp*]: *r* + *k* = *0* ⟹ *k* = − *r* **by** (*simp add*: *minus-unique*)
  show *?one* **unfolding** *poly-mult* **by** *auto*
  show *?two* **unfolding** *poly-mult* **by** *auto*
  show *?three* **by** *auto*
qed

lemma *poly-root-constant*: *c* ≠ *0* ⟹ (*poly* (*p* ∗ [:*c*:]) (*k* :: *'a* :: *idom*) = *0*) =

(*poly p k = 0*)
  unfolding *poly-mult* **by** *auto*

lemma *poly-linear-exp-linear-factors-rev*:
  $([:b,1:])^{\frown}(length\ (filter\ ((=)\ b)\ as))\ dvd\ (\prod\ (a :: 'a :: comm\text{-}ring\text{-}1) \leftarrow as.\ [:\ a,\ 1:])$
proof (*induct as*)
  case (*Cons a as*)
  let *?ls* = *length* (*filter* ((=) *b*) (*a* # *as*))
  let *?l* = *length* (*filter* ((=) *b*) *as*)
  have *prod*: $(\prod\ a \leftarrow Cons\ a\ as.\ [:\ a,\ 1:]) = [:\ a,\ 1\ :] * (\prod\ a \leftarrow as.\ [:\ a,\ 1:])$ **by** *simp*
  show *?case*
  proof (*cases a = b*)
    case *False*
    hence *len*: *?ls* = *?l* **by** *simp*
    show *?thesis* **unfolding** *prod len* **using** *Cons* **by** (*rule dvd-mult*)
  next
    case *True*
    hence *len*: $[:\ b,\ 1\ :]\ \widehat{}\ ?ls = [:\ a,\ 1\ :] * [:\ b,\ 1\ :]\ \widehat{}\ ?l$ **by** *simp*
    show *?thesis* **unfolding** *prod len* **using** *Cons* **using** *dvd-refl mult-dvd-mono* **by** *blast*
  qed
qed *simp*

lemma *order-max*: **assumes** *dvd*: $[:\ -a,\ 1\ :]\ \widehat{}\ k\ dvd\ p$ **and** *p*: $p \neq 0$
  shows $k \leq order\ a\ p$
proof (*rule ccontr*)
  assume ¬ *?thesis*
  hence $\exists\ j.\ k = Suc\ (order\ a\ p + j)$ **by** *arith*
  then obtain *j* **where** *k*: $k = Suc\ (order\ a\ p + j)$ **by** *auto*
  have $[:\ -a,\ 1\ :]\ \widehat{}\ Suc\ (order\ a\ p)\ dvd\ p$
    by (*rule power-le-dvd*[*OF dvd*[*unfolded k*]], *simp*)
  with *order-2*[*OF p*, *of a*] **show** *False* **by** *blast*
qed

7.5 Divisibility

context
  assumes *SORT-CONSTRAINT*(*'a* :: *idom*)
begin
lemma *poly-linear-linear-factor*: **assumes**
  dvd: $[:b,1:]\ dvd\ (\prod\ (a :: 'a) \leftarrow as.\ [:\ a,\ 1:])$
  shows $b \in set\ as$
proof −
  let $?p = \lambda\ as.\ (\prod\ a \leftarrow as.\ [:\ a,\ 1:])$
  let $?b = [:b,1:]$
  from *assms*[*unfolded dvd-def*] **obtain** *p* **where** *id*: $?p\ as = ?b * p$ **..**

from *arg-cong*[*OF id*, *of* λ *p*. *poly p* (−*b*)]
have *poly* (*?p as*) (−*b*) = *0* **by** *simp*
thus *?thesis*
proof (*induct as*)
case (*Cons a as*)
have *?p* (*a* # *as*) = [:*a*,*1*:] ∗ *?p as* **by** *simp*
from *Cons*(*2*)[*unfolded this*] **have** *poly* (*?p as*) (−*b*) = *0* ∨ (*a* − *b*) = *0* **by** *simp*
with *Cons*(*1*) **show** *?case* **by** *auto*
qed *simp*
qed

lemma *poly-linear-exp-linear-factors*:
assumes *dvd*: ([:*b*,*1*:])^*n dvd* (∏ (*a* :: ′*a*) ← *as*. [: *a*, *1*:])
shows *length* (*filter* ((=) *b*) *as*) ≥ *n*
proof −
let *?p* = λ *as*. (∏ *a* ← *as*. [: *a*, *1*:])
let *?b* = [:*b*,*1*:]
from *dvd* **show** *?thesis*
proof (*induct n arbitrary*: *as*)
case (*Suc n as*)
have *bs*: *?b* ^ *Suc n* = *?b* ∗ *?b* ^ *n* **by** *simp*
from *poly-linear-linear-factor*[*OF dvd-mult-left*[*OF Suc*(*2*)[*unfolded bs*]],
unfolded in-set-conv-decomp]
obtain *as1 as2* **where** *as*: *as* = *as1* @ *b* # *as2* **by** *auto*
have *?p as* = [:*b*,*1*:] ∗ *?p* (*as1* @ *as2*) **unfolding** *as*
proof (*induct as1*)
case (*Cons a as1*)
have *?p* (*a* # *as1* @ *b* # *as2*) = [:*a*,*1*:] ∗ *?p* (*as1* @ *b* # *as2*) **by** *simp*
also have *?p* (*as1* @ *b* # *as2*) = [:*b*,*1*:] ∗ *?p* (*as1* @ *as2*) **unfolding** *Cons* **by** *simp*
also have [:*a*,*1*:] ∗ . . . = [:*b*,*1*:] ∗ ([:*a*,*1*:] ∗ *?p* (*as1* @ *as2*))
by (*metis* (*no-types*, *lifting*) *mult.left-commute*)
finally show *?case* **by** *simp*
qed *simp*
from *Suc*(*2*)[*unfolded bs this dvd-mult-cancel-left*]
have *?b* ^ *n dvd ?p* (*as1* @ *as2*) **by** *simp*
from *Suc*(*1*)[*OF this*] **show** *?case* **unfolding** *as* **by** *simp*
qed *simp*
qed
end

lemma *const-poly-dvd*: ([:*a*:] *dvd* [:*b*:]) = (*a dvd b*)
proof
assume *a dvd b*
then obtain *c* **where** *b* = *a* ∗ *c* **unfolding** *dvd-def* **by** *auto*
hence [:*b*:] = [:*a*:] ∗ [: *c*:] **by** (*auto simp*: *ac-simps*)
thus [:*a*:] *dvd* [:*b*:] **unfolding** *dvd-def* **by** *blast*
next

assume [:a:] *dvd* [:b:]
then obtain *pc* **where** [:b:] = [:a:] $*$ *pc* **unfolding** *dvd-def* **by** *blast*
from *arg-cong*[*OF this*, *of* λ *p*. *coeff p 0*, *unfolded coeff-mult*]
have $b = a * $ *coeff pc 0* **by** *auto*
thus *a dvd b* **unfolding** *dvd-def* **by** *blast*
qed

lemma *const-poly-dvd-1* [*simp*]:
[:a:] *dvd 1* $\longleftrightarrow$ *a dvd 1*
by (*metis const-poly-dvd one-poly-eq-simps*(*2*))

lemma *poly-dvd-1*:
fixes *p* :: *'a* :: {*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
shows *p dvd 1* $\longleftrightarrow$ *degree p* = *0* $\wedge$ *coeff p 0 dvd 1*
proof (*cases degree p* = *0*)
case *False*
with *divides-degree*[*of p 1*] **show** *?thesis* **by** *auto*
next
case *True*
then obtain *a* **where** *p*: *p* = [:a:]
using *degree-eq-zeroE* **by** *blast*
show *?thesis* **unfolding** *p* **by** *auto*
qed

Degree based version of irreducibility.

definition $irreducible_d$ :: *'a* :: *comm-semiring-1 poly* $\Rightarrow$ *bool* **where**
$irreducible_d$ *p* = (*degree p* > *0* $\wedge$ ($\forall$ *q r*. *degree q* < *degree p* $\longrightarrow$ *degree r* < *degree p* $\longrightarrow$ *p* $\neq$ *q* $*$ *r*))

lemma $irreducible_d I$ [*intro*]:
assumes *1*: *degree p* > *0*
and *2*: $\bigwedge$*q r*. *degree q* > *0* $\Longrightarrow$ *degree q* < *degree p* $\Longrightarrow$ *degree r* > *0* $\Longrightarrow$ *degree r* < *degree p* $\Longrightarrow$ *p* = *q* $*$ *r* $\Longrightarrow$ *False*
shows $irreducible_d$ *p*
proof (*unfold* $irreducible_d$*-def*, *intro conjI allI impI notI 1*)
fix *q r*
assume *degree q* < *degree p* **and** *degree r* < *degree p* **and** *p* = *q* $*$ *r*
with *degree-mult-le*[*of q r*]
show *False* **by** (*intro 2*, *auto*)
qed

lemma $irreducible_d I2$:
fixes *p* :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
assumes *deg*: *degree p* > *0* **and** *ndvd*: $\bigwedge$ *q*. *degree q* > *0* $\Longrightarrow$ *degree q* $\leq$ *degree p div 2* $\Longrightarrow$ $\neg$ *q dvd p*
shows $irreducible_d$ *p*
proof (*rule ccontr*)
assume $\neg$ *?thesis*
from *this*[*unfolded* $irreducible_d$*-def*] *deg* **obtain** *q r* **where** *dq*: *degree q* < *degree*

p **and** *dr*: *degree r* < *degree p*
  and *p*: *p* = *q* ∗ *r* **by** *auto*
 from *deg* **have** *p0*: *p* ≠ *0* **by** *auto*
 with *p* **have** *q* ≠ *0 r* ≠ *0* **by** *auto*
 from *degree-mult-eq*[*OF this*] *p* **have** *dp*: *degree p* = *degree q* + *degree r* **by** *simp*
 show *False*
 proof (*cases degree q* ≤ *degree p div 2*)
  case *True*
  from *ndvd*[*OF - True*] *dq dr dp p* **show** *False* **by** *auto*
 next
  case *False*
  with *dp* **have** *dr*: *degree r* ≤ *degree p div 2* **by** *auto*
  from *p* **have** *dvd*: *r dvd p* **by** *auto*
  from *ndvd*[*OF - dr*] *dvd dp dq* **show** *False* **by** *auto*
 qed
qed

lemma *reducible*$_d$*I*:
 assumes *degree p* > *0* ⟹ ∃ *q r*. *degree q* < *degree p* ∧ *degree r* < *degree p* ∧ *p* = *q* ∗ *r*
 shows ¬ *irreducible*$_d$ *p*
 using *assms* **by** (*auto simp*: *irreducible*$_d$*-def*)

lemma *irreducible*$_d$*E* [*elim*]:
 assumes *irreducible*$_d$ *p*
  and *degree p* > *0* ⟹ (⋀*q r*. *degree q* < *degree p* ⟹ *degree r* < *degree p* ⟹ *p* ≠ *q* ∗ *r*) ⟹ *thesis*
 shows *thesis*
 using *assms* **by** (*auto simp*: *irreducible*$_d$*-def*)

lemma *reducible*$_d$*E* [*elim*]:
 assumes *red*: ¬ *irreducible*$_d$ *p*
  and *1*: *degree p* = *0* ⟹ *thesis*
  and *2*: ⋀*q r*. *degree q* > *0* ⟹ *degree q* < *degree p* ⟹ *degree r* > *0* ⟹ *degree r* < *degree p* ⟹ *p* = *q* ∗ *r* ⟹ *thesis*
 shows *thesis*
 using *red*[*unfolded irreducible*$_d$*-def de-Morgan-conj not-not not-all not-imp*]
proof (*elim disjE exE conjE*)
 show ¬*degree p* > *0* ⟹ *thesis* **using** *1* **by** *auto*
next
 fix *q r*
 assume *degree q* < *degree p* **and** *degree r* < *degree p* **and** *p* = *q* ∗ *r*
 with *degree-mult-le*[*of q r*]
 show *thesis* **by** (*intro 2*, *auto*)
qed

lemma *irreducible*$_d$*D*:
 assumes *irreducible*$_d$ *p*
 shows *degree p* > *0* ⋀*q r*. *degree q* < *degree p* ⟹ *degree r* < *degree p* ⟹ *p* ≠

$q * r$
using *assms* **unfolding** *irreducible$_d$-def* **by** *auto*

theorem *irreducible$_d$-factorization-exists*:
assumes *degree* $p > 0$
shows $\exists$*fs*. *fs* $\neq$ [] $\wedge$ ($\forall f \in$ *set fs*. *irreducible$_d$* $f \wedge$ *degree* $f \leq$ *degree* p) $\wedge$ $p =$ *prod-list fs*
and $\neg$*irreducible$_d$* $p \Longrightarrow \exists$*fs*. *length fs* > 1 $\wedge$ ($\forall f \in$ *set fs*. *irreducible$_d$* $f \wedge$ *degree* $f <$ *degree* p) $\wedge$ $p =$ *prod-list fs*
proof (*atomize*(*full*), *insert assms*, *induct degree p arbitrary*:*p rule*: *less-induct*)
case *less*
then have *deg-f*: *degree* $p > 0$ **by** *auto*
show *?case*
proof (*cases irreducible$_d$ p*)
case *True*
then have *set* [p] $\subseteq$ *Collect irreducible$_d$* $p =$ *prod-list* [p] **by** *auto*
with *True* **show** *?thesis* **by** (*auto intro*: *exI*[*of* - [p]])
next
case *False*
with *deg-f* **obtain** *g h*
where *deg-g*: *degree* $g <$ *degree* p *degree* $g > 0$
and *deg-h*: *degree* $h <$ *degree* p *degree* $h > 0$
and *f-gh*: $p = g * h$ **by** *auto*
from *less.hyps*[*OF deg-g*] *less.hyps*[*OF deg-h*]
obtain *gs hs*
where *emp*: *length gs* > 0 *length hs* > 0
and $\forall f \in$ *set gs*. *irreducible$_d$* $f \wedge$ *degree* $f \leq$ *degree* g $g =$ *prod-list gs*
and $\forall f \in$ *set hs*. *irreducible$_d$* $f \wedge$ *degree* $f \leq$ *degree* h $h =$ *prod-list hs* **by** *auto*
with *f-gh deg-g deg-h*
have *len*: *length* (*gs*@*hs*) > 1
and *mem*: $\forall f \in$ *set* (*gs*@*hs*). *irreducible$_d$* $f \wedge$ *degree* $f <$ *degree* p
and *p*: $p =$ *prod-list* (*gs*@*hs*) **by** (*auto simp del*: *length-greater-0-conv*)
with *False* **show** *?thesis* **by** (*auto intro*!: *exI*[*of* - *gs*@*hs*] *simp*: *less-imp-le*)
qed
qed

lemma *irreducible$_d$-factor*:
fixes p :: $'a$::{*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
assumes *degree* $p > 0$
shows $\exists$ *q r*. *irreducible$_d$* $q \wedge p = q * r \wedge$ *degree* $r <$ *degree* p **using** *assms*
proof (*induct degree p arbitrary*: *p rule*: *less-induct*)
case (*less p*)
show *?case*
proof (*cases irreducible$_d$ p*)
case *False*
with *less*(*2*) **obtain** *q r*
where *q*: *degree* $q <$ *degree* p *degree* $q > 0$
and *r*: *degree* $r <$ *degree* p *degree* $r > 0$
and *p*: $p = q * r$

by *auto*
from *less*(*1*)[*OF q*] **obtain** *s t* **where** *IH*: $irreducible_d$ *s q* = *s* ∗ *t* **by** *auto*
from *p* **have** *p*: *p* = *s* ∗ (*t* ∗ *r*) **unfolding** *IH* **by** (*simp add*: *ac-simps*)
from *less*(*2*) **have** *p* ≠ *0* **by** *auto*
hence *degree p* = *degree s* + (*degree* (*t* ∗ *r*)) **unfolding** *p*
by (*subst degree-mult-eq*, *insert p*, *auto*)
with $irreducible_d D$[*OF IH*(*1*)] **have** *degree p* > *degree* (*t* ∗ *r*) **by** *auto*
with *p IH* **show** *?thesis* **by** *auto*
next
case *True*
show *?thesis*
by (*rule exI*[*of - p*], *rule exI*[*of - 1*], *insert True less*(*2*), *auto*)
qed
qed

context *mult-zero* **begin**

definition *zero-divisor* **where** *zero-divisor a* ≡ ∃ *b*. *b* ≠ *0* ∧ *a* ∗ *b* = *0*

lemma *zero-divisorI*[*intro*]:
assumes *b* ≠ *0* **and** *a* ∗ *b* = *0* **shows** *zero-divisor a*
using *assms* **by** (*auto simp*: *zero-divisor-def*)

lemma *zero-divisorE*[*elim*]:
assumes *zero-divisor a*
and ⋀*b*. *b* ≠ *0* ⟹ *a* ∗ *b* = *0* ⟹ *thesis*
shows *thesis*
using *assms* **by** (*auto simp*: *zero-divisor-def*)

end

lemma *zero-divisor-0*[*simp*]:
zero-divisor (*0*::'*a*::{*mult-zero*,*zero-neq-one*})
by (*auto intro*!: *zero-divisorI*[*of 1*])

lemma *not-zero-divisor-1*:
¬ *zero-divisor* (*1* :: '*a* :: {*monoid-mult*,*mult-zero*})
by *auto*

lemma *zero-divisor-iff-eq-0*[*simp*]:
fixes *a* :: '*a* :: {*semiring-no-zero-divisors*, *zero-neq-one*}
shows *zero-divisor a* ⟷ *a* = *0* **by** *auto*

lemma *mult-eq-0-not-zero-divisor-left*[*simp*]:
fixes *a b* :: '*a* :: *mult-zero*
assumes ¬ *zero-divisor a*
shows *a* ∗ *b* = *0* ⟷ *b* = *0*
using *assms* **unfolding** *zero-divisor-def* **by** *force*

lemma *mult-eq-0-not-zero-divisor-right*[*simp*]:
  fixes *a b* :: *'a* :: {*ab-semigroup-mult*,*mult-zero*}
  assumes ¬ *zero-divisor b*
  shows $a * b = 0 \longleftrightarrow a = 0$
  using *assms* **unfolding** *zero-divisor-def* **by** (*force simp*: *ac-simps*)

lemma *degree-smult-not-zero-divisor-left*[*simp*]:
  assumes ¬ *zero-divisor c*
  shows *degree* (*smult c p*) = *degree p*
proof(*cases p* = *0*)
  case *False*
  then have *coeff* (*smult c p*) (*degree p*) ≠ *0* **using** *assms* **by** *auto*
  from *le-degree*[*OF this*] *degree-smult-le*[*of c p*]
  show *?thesis* **by** *auto*
qed *auto*

lemma *degree-smult-not-zero-divisor-right*[*simp*]:
  assumes ¬ *zero-divisor* (*lead-coeff p*)
  shows *degree* (*smult c p*) = (*if c* = *0 then 0 else degree p*)
proof(*cases c* = *0*)
  case *False*
  then have *coeff* (*smult c p*) (*degree p*) ≠ *0* **using** *assms* **by** *auto*
  from *le-degree*[*OF this*] *degree-smult-le*[*of c p*]
  show *?thesis* **by** *auto*
qed *auto*

lemma $irreducible_d$*-smult-not-zero-divisor-left*:
  assumes *c0*: ¬ *zero-divisor c*
  assumes *L*: $irreducible_d$ (*smult c p*)
  shows $irreducible_d$ *p*
proof (*intro* $irreducible_dI$)
  from *L* **have** *degree* (*smult c p*) > *0* **by** *auto*
  also note *degree-smult-le*
  finally show *degree p* > *0* **by** *auto*
  fix *q r*
  assume *deg-q*: *degree q* < *degree p*
    and *deg-r*: *degree r* < *degree p*
    and *p-qr*: *p* = *q* ∗ *r*
  then have *1*: *smult c p* = *smult c q* ∗ *r* **by** *auto*
  note *degree-smult-le*[*of c q*]
  also note *deg-q*
  finally have *2*: *degree* (*smult c q*) < *degree* (*smult c p*) **using** *c0* **by** *auto*
  from *deg-r* **have** *3*: *degree r* < . . . **using** *c0* **by** *auto*
  from $irreducible_dD(2)$[*OF L 2 3*] *1* **show** *False* **by** *auto*
qed

lemmas $irreducible_d$*-smultI* =
  $irreducible_d$*-smult-not-zero-divisor-left*

```
  [where 'a = 'a :: {comm-semiring-1,semiring-no-zero-divisors}, simplified]

lemma irreducible_d-smult-not-zero-divisor-right:
  assumes p0: ¬ zero-divisor (lead-coeff p) and L: irreducible_d (smult c p)
  shows irreducible_d p
proof−
  from L have c ≠ 0 by auto
  with p0 have [simp]: degree (smult c p) = degree p by simp
  show irreducible_d p
  proof (intro iffI irreducible_dI conjI)
    from L show degree p > 0 by auto
    fix q r
    assume deg-q: degree q < degree p
      and deg-r: degree r < degree p
      and p-qr: p = q ∗ r
    then have 1: smult c p = smult c q ∗ r by auto
    note degree-smult-le[of c q]
    also note deg-q
    finally have 2: degree (smult c q) < degree (smult c p) by simp
    from deg-r have 3: degree r < . . . by simp
    from irreducible_dD(2)[OF L 2 3] 1 show False by auto
  qed
qed

lemma zero-divisor-mult-left:
  fixes a b :: 'a :: {ab-semigroup-mult, mult-zero}
  assumes zero-divisor a
  shows zero-divisor (a ∗ b)
proof−
  from assms obtain c where c0: c ≠ 0 and [simp]: a ∗ c = 0 by auto
  have a ∗ b ∗ c = a ∗ c ∗ b by (simp only: ac-simps)
  with c0 show ?thesis by auto
qed

lemma zero-divisor-mult-right:
  fixes a b :: 'a :: {semigroup-mult, mult-zero}
  assumes zero-divisor b
  shows zero-divisor (a ∗ b)
proof−
  from assms obtain c where c0: c ≠ 0 and [simp]: b ∗ c = 0 by auto
  have a ∗ b ∗ c = a ∗ (b ∗ c) by (simp only: ac-simps)
  with c0 show ?thesis by auto
qed

lemma not-zero-divisor-mult:
  fixes a b :: 'a :: {ab-semigroup-mult, mult-zero}
  assumes ¬ zero-divisor (a ∗ b)
  shows ¬ zero-divisor a and ¬ zero-divisor b
  using assms by (auto dest: zero-divisor-mult-right zero-divisor-mult-left)
```

```
lemma zero-divisor-smult-left:
  assumes zero-divisor a
  shows zero-divisor (smult a f)
proof−
  from assms obtain b where b0: b ≠ 0 and a ∗ b = 0 by auto
  then have smult a f ∗ [:b:] = 0 by (simp add: ac-simps)
  with b0 show ?thesis by (auto intro!: zero-divisorI[of [:b:]])
qed

lemma unit-not-zero-divisor:
  fixes a :: 'a :: {comm-monoid-mult, mult-zero}
  assumes a dvd 1
  shows ¬zero-divisor a
proof
  from assms obtain b where ab: 1 = a ∗ b by (elim dvdE)
  assume zero-divisor a
  then have zero-divisor (1::'a) by (unfold ab, intro zero-divisor-mult-left)
  then show False by auto
qed


lemma linear-irreducible_d: assumes degree p = 1
  shows irreducible_d p
  by (rule irreducible_dI, insert assms, auto)

lemma irreducible_d-dvd-smult:
  fixes p :: 'a::{comm-semiring-1,semiring-no-zero-divisors} poly
  assumes degree p > 0 irreducible_d q p dvd q
  shows ∃ c. c ≠ 0 ∧ q = smult c p
proof −
  from assms obtain r where q: q = p ∗ r by (elim dvdE, auto)
  from degree-mult-eq[of p r] assms(1) q
  obtain ¬ degree p < degree q and nz: p ≠ 0 q ≠ 0
    by (metis assms(2) degree-0 less-add-same-cancel2 less-irrefl reducible_dI)
  hence deg: degree p ≥ degree q by auto
  from ‹p dvd q› obtain k where q: q = k ∗ p unfolding dvd-def by (auto simp:
ac-simps)
  with nz have k ≠ 0 by auto
  from deg[unfolded q degree-mult-eq[OF ‹k ≠ 0› ‹p ≠ 0› ]] have degree k = 0
    unfolding q by auto
  then obtain c where k: k = [: c :]
    using degree-eq-zeroE by blast
  with ‹k ≠ 0› have c ≠ 0 by auto
  have q = smult c p unfolding q k by simp
  with ‹c ≠ 0› show ?thesis by auto
qed
```

7.6 Map over Polynomial Coefficients

```
lemma map-poly-simps:
  shows map-poly f (pCons c p) =
    (if c = 0 ∧ p = 0 then 0 else pCons (f c) (map-poly f p))
proof (cases c = 0)
  case True note c0 = this show ?thesis
    proof (cases p = 0)
      case True thus ?thesis using c0 unfolding map-poly-def by simp
      next case False thus ?thesis
        unfolding map-poly-def by auto
    qed
  next case False thus ?thesis
    unfolding map-poly-def by auto
qed

lemma map-poly-pCons[simp]:
  assumes c ≠ 0 ∨ p ≠ 0
  shows map-poly f (pCons c p) = pCons (f c) (map-poly f p)
  unfolding map-poly-simps using assms by auto

lemma map-poly-map-poly:
  assumes f0: f 0 = 0
  shows map-poly f (map-poly g p) = map-poly (f ∘ g) p
proof (induct p)
  case (pCons a p) show ?case
  proof(cases g a ≠ 0 ∨ map-poly g p ≠ 0)
    case True show ?thesis
      unfolding map-poly-pCons[OF pCons(1)]
      unfolding map-poly-pCons[OF True]
      unfolding pCons(2)
      by simp
  next
    case False then show ?thesis
      unfolding map-poly-pCons[OF pCons(1)]
      unfolding pCons(2)[symmetric]
      by (simp add: f0)
  qed
qed simp

lemma map-poly-zero:
  assumes f: ∀ c. f c = 0 ⟶ c = 0
  shows [simp]: map-poly f p = 0 ⟷ p = 0
  by (induct p; auto simp: map-poly-simps f)

lemma map-poly-add:
  assumes h0: h 0 = 0
      and h-add: ∀ p q. h (p + q) = h p + h q
  shows map-poly h (p + q) = map-poly h p + map-poly h q
proof (induct p arbitrary: q)
```

case (*pCons a p*) **note** *pIH* = *this*
show *?case*
proof(*induct q*)
case (*pCons b q*) **note** *qIH* = *this*
show *?case*
unfolding *map-poly-pCons*[*OF qIH*(*1*)]
unfolding *map-poly-pCons*[*OF pIH*(*1*)]
unfolding *add-pCons*
unfolding *pIH*(*2*)[*symmetric*]
unfolding *h-add*[*rule-format*,*symmetric*]
unfolding *map-poly-simps* **using** *h0* **by** *auto*
qed *auto*
qed *auto*

7.7 Morphismic properties of *pCons 0*

lemma *monom-pCons-0-monom*:
monom (*pCons 0* (*monom a n*)) *d* = *map-poly* (*pCons 0*) (*monom* (*monom a n*) *d*)
apply (*induct d*)
unfolding *monom-0* **unfolding** *map-poly-simps* **apply** *simp*
unfolding *monom-Suc map-poly-simps* **by** *auto*

lemma *pCons-0-add*: *pCons 0* (*p* + *q*) = *pCons 0 p* + *pCons 0 q* **by** *auto*

lemma *sum-pCons-0-commute*:
sum ($\lambda i.$ *pCons 0* (*f i*)) *S* = *pCons 0* (*sum f S*)
by(*induct S rule*: *infinite-finite-induct*;*simp*)

lemma *pCons-0-as-mult*:
fixes *p*:: *'a* :: *comm-semiring-1 poly*
shows *pCons 0 p* = [:*0*,*1*:] ∗ *p* **by** *auto*

7.8 Misc

fun *expand-powers* :: (*nat* × *'a*)*list* ⇒ *'a list* **where**
expand-powers [] = []
| *expand-powers* ((*Suc n*, *a*) # *ps*) = *a* # *expand-powers* ((*n*,*a*) # *ps*)
| *expand-powers* ((*0*,*a*) # *ps*) = *expand-powers ps*

lemma *expand-powers*: **fixes** *f* :: *'a* ⇒ *'b* :: *comm-ring-1*
shows ($\prod$ (*n*,*a*) ← *n-as*. *f a* ^ *n*) = ($\prod$ *a* ← *expand-powers n-as*. *f a*)
by (*rule sym*, *induct n-as rule*: *expand-powers.induct*, *auto*)

lemma *poly-smult-zero-iff*: **fixes** *x* :: *'a* :: *idom*
shows (*poly* (*smult a p*) *x* = *0*) = (*a* = *0* ∨ *poly p x* = *0*)
by *simp*

lemma *poly-prod-list-zero-iff*: **fixes** *x* :: *'a* :: *idom*
shows (*poly* (*prod-list ps*) *x* = *0*) = (∃ *p* ∈ *set ps*. *poly p x* = *0*)

by (*induct ps*, *auto*)

lemma *poly-mult-zero-iff*: **fixes** *x* :: *'a* :: *idom*
shows (*poly* (*p* ∗ *q*) *x* = *0*) = (*poly p x* = *0* ∨ *poly q x* = *0*)
by *simp*

lemma *poly-power-zero-iff*: **fixes** *x* :: *'a* :: *idom*
shows (*poly* (*p*^*n*) *x* = *0*) = (*n* ≠ *0* ∧ *poly p x* = *0*)
by (*cases n*, *auto*)

lemma *sum-monom-0-iff*: **assumes** *fin*: *finite S*
and *g*: ⋀ *i j*. *g i* = *g j* ⟹ *i* = *j*
shows *sum* (λ *i*. *monom* (*f i*) (*g i*)) *S* = *0* ⟷ (∀ *i* ∈ *S*. *f i* = *0*) (**is** *?l* = *?r*)
proof −
{
assume ¬ *?r*
then obtain *i* **where** *i*: *i* ∈ *S* **and** *fi*: *f i* ≠ *0* **by** *auto*
let *?g* = λ *i*. *monom* (*f i*) (*g i*)
have *coeff* (*sum ?g S*) (*g i*) = *f i* + *sum* (λ *j*. *coeff* (*?g j*) (*g i*)) (*S* − {*i*})
by (*unfold sum.remove*[*OF fin i*], *simp add*: *coeff-sum*)
also have *sum* (λ *j*. *coeff* (*?g j*) (*g i*)) (*S* − {*i*}) = *0*
by (*rule sum.neutral*, *insert g*, *auto*)
finally have *coeff* (*sum ?g S*) (*g i*) ≠ *0* **using** *fi* **by** *auto*
hence ¬ *?l* **by** *auto*
}
thus *?thesis* **by** *auto*
qed

lemma *degree-prod-list-eq*: **assumes** ⋀ *p*. *p* ∈ *set ps* ⟹ (*p* :: *'a* :: *idom poly*) ≠ *0*
shows *degree* (*prod-list ps*) = *sum-list* (*map degree ps*) **using** *assms*
proof (*induct ps*)
case (*Cons p ps*)
show *?case* **unfolding** *prod-list.Cons*
by (*subst degree-mult-eq*, *insert Cons*, *auto simp*: *prod-list-zero-iff*)
qed *simp*

lemma *degree-power-eq*: **assumes** *p*: *p* ≠ *0*
shows *degree* (*p* ^ *n*) = *degree* (*p* :: *'a* :: *idom poly*) ∗ *n*
proof (*induct n*)
case (*Suc n*)
from *p* **have** *pn*: *p* ^ *n* ≠ *0* **by** *auto*
show *?case* **using** *degree-mult-eq*[*OF p pn*] *Suc* **by** *auto*
qed *simp*

lemma *coeff-Poly*: *coeff* (*Poly xs*) *i* = (*nth-default 0 xs i*)
unfolding *nth-default-coeffs-eq*[*of Poly xs*, *symmetric*] *coeffs-Poly* **by** *simp*

lemma *rsquarefree-def'*: *rsquarefree p* = (*p* ≠ *0* ∧ (∀ *a*. *order a p* ≤ *1*))

proof −
  have $\bigwedge$ *a*. *order a p* ≤ *1* ⟷ *order a p* = *0* ∨ *order a p* = *1* **by** *linarith*
  thus *?thesis* **unfolding** *rsquarefree-def* **by** *auto*
qed

lemma *order-prod-list*: ($\bigwedge$ *p*. *p* ∈ *set ps* ⟹ *p* ≠ *0*) ⟹ *order x* (*prod-list ps*) = *sum-list* (*map* (*order x*) *ps*)
  by (*induct ps*, *auto*, *subst order-mult*, *auto simp*: *prod-list-zero-iff*)

lemma *irreducible*$_d$*-dvd-eq*:
  fixes *a b* :: ′*a*::{*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
  assumes *irreducible*$_d$ *a* **and** *irreducible*$_d$ *b*
    and *a dvd b*
    and *monic a* **and** *monic b*
  shows *a* = *b*
  using *assms*
   by (*metis* (*no-types*, *lifting*) *coeff-smult degree-smult-eq irreducible*$_d$*D*(*1*) *irreducible*$_d$*-dvd-smult*
    mult.right-neutral smult-1-left)

lemma *monic-gcd-dvd*:
  assumes *fg*: *f dvd g* **and** *mon*: *monic f* **and** *gcd*: *gcd g h* ∈ {*1*, *g*}
  shows *gcd f h* ∈ {*1*, *f*}
proof (*cases coprime g h*)
  case *True*
  with *dvd-refl* **have** *coprime f h*
    using *fg* **by** (*blast intro*: *coprime-divisors*)
  then show *?thesis*
    by *simp*
next
  case *False*
  with *gcd* **have** *gcd*: *gcd g h* = *g*
    by (*simp add*: *coprime-iff-gcd-eq-1*)
  with *fg* **have** *f dvd gcd g h*
    by *simp*
  then have *f dvd h*
    by *simp*
  then have *gcd f h* = *normalize f*
    by (*simp add*: *gcd-proj1-iff*)
  also have *normalize f* = *f*
    using *mon* **by** (*rule normalize-monic*)
  finally show *?thesis*
    by *simp*
qed

lemma *monom-power*: (*monom a b*)^*n* = *monom* (*a*^*n*) (*b*∗*n*)
  by (*induct n*, *auto simp add*: *mult-monom*)

lemma *poly-const-pow*: [:*a*:]^*b* = [:*a*^*b*:]

by (*metis Groups.mult-ac*(*2*) *monom-0 monom-power mult-zero-right*)

lemma *degree-pderiv-le*: *degree* (*pderiv f*) ≤ *degree f* − *1*
proof (*rule ccontr*)
assume ¬ *?thesis*
hence *ge*: *degree* (*pderiv f*) ≥ *Suc* (*degree f* − *1*) **by** *auto*
hence *pderiv f* ≠ *0* **by** *auto*
hence *coeff* (*pderiv f*) (*degree* (*pderiv f*)) ≠ *0* **by** *auto*
from *this*[*unfolded coeff-pderiv*]
have *coeff f* (*Suc* (*degree* (*pderiv f*))) ≠ *0* **by** *auto*
moreover have *Suc* (*degree* (*pderiv f*)) > *degree f* **using** *ge* **by** *auto*
ultimately show *False* **by** (*simp add*: *coeff-eq-0*)
qed

lemma *map-div-is-smult-inverse*: *map-poly* (λ*x*. *x* / (*a* :: ′*a* :: *field*)) *p* = *smult* (*inverse a*) *p*
unfolding *smult-conv-map-poly*
by (*simp add*: *divide-inverse-commute*)

lemma *normalize-poly-old-def*:
normalize (*f* :: ′*a* :: {*normalization-semidom*,*field*} *poly*) = *smult* (*inverse* (*unit-factor* (*lead-coeff f*))) *f*
by (*simp add*: *normalize-poly-eq-map-poly map-div-is-smult-inverse*)

lemma *poly-dvd-antisym*:
fixes *p q* :: ′*b*::*idom poly*
assumes *coeff*: *coeff p* (*degree p*) = *coeff q* (*degree q*)
assumes *dvd1*: *p dvd q* **and** *dvd2*: *q dvd p* **shows** *p* = *q*
proof (*cases p* = *0*)
case *True* **with** *coeff* **show** *p* = *q* **by** *simp*
next
case *False* **with** *coeff* **have** *q* ≠ *0* **by** *auto*
have *degree*: *degree p* = *degree q*
using ‹*p dvd q*› ‹*q dvd p*› ‹*p* ≠ *0*› ‹*q* ≠ *0*›
by (*intro order-antisym dvd-imp-degree-le*)

from ‹*p dvd q*› **obtain** *a* **where** *a*: *q* = *p* ∗ *a* **..**
with ‹*q* ≠ *0*› **have** *a* ≠ *0* **by** *auto*
with *degree a* ‹*p* ≠ *0*› **have** *degree a* = *0*
by (*simp add*: *degree-mult-eq*)
with *coeff a* **show** *p* = *q*
by (*cases a*, *auto split*: *if-splits*)
qed

lemma *coeff-f-0-code*[*code-unfold*]: *coeff f 0* = (*case coeffs f of* [] ⇒ *0* | *x* # - ⇒ *x*)
by (*cases f*, *auto simp*: *cCons-def*)

lemma *poly-compare-0-code*[*code-unfold*]: (*f* = *0*) = (*case coeffs f of* [] ⇒ *True* | - ⇒ *False*)
  using *coeffs-eq-Nil list.disc-eq-case*(*1*) **by** *blast*

Getting more efficient code for abbreviation *lead-coeff*"

definition *leading-coeff*
  where [*code-abbrev*, *simp*]: *leading-coeff* = *lead-coeff*

lemma *leading-coeff-code* [*code*]:
  leading-coeff f = (*let xs* = *coeffs f in if xs* = [] *then 0 else last xs*)
  by (*simp add*: *last-coeffs-eq-coeff-degree*)

lemma *nth-coeffs-coeff*: *i* < *length* (*coeffs f*) ⟹ *coeffs f* ! *i* = *coeff f i*
  by (*metis nth-default-coeffs-eq nth-default-def*)

definition *monom-mult* :: *nat* ⇒ ′*a* :: *comm-semiring-1 poly* ⇒ ′*a poly*
  where *monom-mult n f* = *monom 1 n* ∗ *f*

lemma *monom-mult-unfold* [*code-unfold*]:
  monom 1 n ∗ *f* = *monom-mult n f*
  f ∗ *monom 1 n* = *monom-mult n f*
  by (*auto simp*: *monom-mult-def ac-simps*)

lemma *monom-mult-code* [*code abstract*]:
  coeffs (*monom-mult n f*) = (*let xs* = *coeffs f in*
    if xs = [] *then xs else replicate n 0* @ *xs*)
  by (*rule coeffs-eqI*)
    (*auto simp add*: *Let-def monom-mult-def coeff-monom-mult nth-default-append nth-default-coeffs-eq*)

lemma *coeff-pcompose-monom*: **fixes** *f* :: ′*a* :: *comm-ring-1 poly*
  assumes *n*: *j* < *n*
  shows *coeff* ($f \circ_p$ *monom 1 n*) (*n* ∗ *i* + *j*) = (*if j* = *0 then coeff f i else 0*)
proof (*induct f arbitrary*: *i*)
  case (*pCons a f i*)
  note *d* = *pcompose-pCons coeff-add coeff-monom-mult coeff-pCons*
  show *?case*
  proof (*cases i*)
    case *0*
    show *?thesis* **unfolding** *d 0* **using** *n* **by** (*cases j*, *auto*)
  next
    case (*Suc ii*)
   have *id*: *n* ∗ *Suc ii* + *j* − *n* = *n* ∗ *ii* + *j* **using** *n* **by** (*simp add*: *diff-mult-distrib2*)
    have *id1*: (*n* ≤ *n* ∗ *Suc ii* + *j*) = *True* **by** *auto*
    have *id2*: (*case n* ∗ *Suc ii* + *j of 0* ⇒ *a* | *Suc x* ⇒ *coeff 0 x*) = *0* **using** *n*
      by (*cases n* ∗ *Suc ii* + *j*, *auto*)
    show *?thesis* **unfolding** *d Suc id id1 id2 pCons*(*2*) *if-True* **by** *auto*
  qed
qed *auto*

lemma *coeff-pcompose-x-pow-n*: **fixes** *f* :: *'a* :: *comm-ring-1 poly*
assumes *n*: $n \neq 0$
shows *coeff* (*f* $\circ_p$ *monom 1 n*) ($n * i$) = *coeff f i*
using *coeff-pcompose-monom*[*of 0 n f i*] *n* **by** *auto*

lemma *dvd-dvd-smult*: *a dvd b* $\Longrightarrow$ *f dvd g* $\Longrightarrow$ *smult a f dvd smult b g*
unfolding *dvd-def* **by** (*metis mult-smult-left mult-smult-right smult-smult*)

definition *sdiv-poly* :: *'a* :: *idom-divide poly* $\Rightarrow$ *'a* $\Rightarrow$ *'a poly* **where**
sdiv-poly p a = (*map-poly* (λ *c. c div a*) *p*)

lemma *smult-map-poly*: *smult a* = *map-poly* (($*$) *a*)
by (*rule ext*, *rule poly-eqI*, *subst coeff-map-poly*, *auto*)

lemma *smult-exact-sdiv-poly*: **assumes** $\bigwedge$ *c*. *c* $\in$ *set* (*coeffs p*) $\Longrightarrow$ *a dvd c*
shows *smult a* (*sdiv-poly p a*) = *p*
unfolding *smult-map-poly sdiv-poly-def*
by (*subst map-poly-map-poly*,*simp*,*rule map-poly-idI*, *insert assms*, *auto*)

lemma *coeff-sdiv-poly*: *coeff* (*sdiv-poly f a*) *n* = *coeff f n div a*
unfolding *sdiv-poly-def* **by** (*rule coeff-map-poly*, *auto*)

lemma *poly-pinfty-ge*:
fixes *p* :: *real poly*
assumes *lead-coeff p* > *0 degree p* $\neq$ *0*
shows $\exists$ *n*. $\forall$ *x* $\geq$ *n*. *poly p x* $\geq$ *b*
proof –
let *?p* = *p* – [:*b* – *lead-coeff p* :]
have *id*: *lead-coeff ?p* = *lead-coeff p* **using** *assms*(*2*)
by (*cases p*, *auto*)
with *assms*(*1*) **have** *lead-coeff ?p* > *0* **by** *auto*
from *poly-pinfty-gt-lc*[*OF this*, *unfolded id*] **obtain** *n*
where $\bigwedge$ *x*. *x* $\geq$ *n* $\Longrightarrow$ *0* $\leq$ *poly p x* – *b* **by** *auto*
thus *?thesis* **by** *auto*
qed

lemma *pderiv-sum*: *pderiv* (*sum f I*) = *sum* (λ *i*. (*pderiv* (*f i*))) *I*
by (*induct I rule*: *infinite-finite-induct*, *auto simp*: *pderiv-add*)

lemma *smult-sum2*: *smult m* ($\sum$ *i* $\in$ *S*. *f i*) = ($\sum$ *i* $\in$ *S*. *smult m* (*f i*))
by (*induct S rule*: *infinite-finite-induct*, *auto simp add*: *smult-add-right*)

lemma *degree-mult-not-eq*:
degree (*f* $*$ *g*) $\neq$ *degree f* + *degree g* $\Longrightarrow$ *lead-coeff f* $*$ *lead-coeff g* = *0*
by (*rule ccontr*, *auto simp*: *coeff-mult-degree-sum degree-mult-le le-antisym le-degree*)

lemma *irreducible$_d$-multD*:
fixes *a b* :: *'a* :: {*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*

assumes *l*: *irreducible*$_d$ (*a*∗*b*)
shows *degree a* = *0* ∧ *a* ≠ *0* ∧ *irreducible*$_d$ *b* ∨ *degree b* = *0* ∧ *b* ≠ *0* ∧ *irreducible*$_d$ *a*
proof−
from *l* **have** *a0*: *a* ≠ *0* **and** *b0*: *b* ≠ *0* **by** *auto*
note [*simp*] = *degree-mult-eq*[*OF this*]
from *l* **have** *degree a* = *0* ∨ *degree b* = *0* **apply** (*unfold irreducible*$_d$*-def*) **by** *force*
then show *?thesis*
proof(*elim disjE*)
assume *a*: *degree a* = *0*
with *l a0* **have** *irreducible*$_d$ *b*
by (*simp add*: *irreducible*$_d$*-def*)
(*metis degree-mult-eq degree-mult-eq-0 mult.left-commute plus-nat.add-0*)
with *a a0* **show** *?thesis* **by** *auto*
next
assume *b*: *degree b* = *0*
with *l b0* **have** *irreducible*$_d$ *a*
unfolding *irreducible*$_d$*-def*
by (*smt* (*verit*) *add-cancel-left-right degree-mult-eq degree-mult-eq-0 neq0-conv semiring-normalization-rules*(*16*))
with *b b0* **show** *?thesis* **by** *auto*
qed
qed

lemma *irreducible-connect-field*[*simp*]:
fixes *f* :: ′*a* :: *field poly*
shows *irreducible*$_d$ *f* = *irreducible f* (**is** *?l* = *?r*)
proof
show *?r* ⟹ *?l*
apply (*intro irreducible*$_d$*I*, *force simp*:*is-unit-iff-degree*)
by (*auto dest*!: *irreducible-multD simp*: *poly-dvd-1*)
next
assume *l*: *?l*
show *?r*
proof (*rule irreducibleI*)
from *l* **show** *f* ≠ *0* ¬ *is-unit f* **by** (*auto simp*: *poly-dvd-1*)
fix *a b* **assume** *f* = *a* ∗ *b*
from *l*[*unfolded this*]
show *a dvd 1* ∨ *b dvd 1* **by** (*auto dest*!: *irreducible*$_d$*-multD simp*:*is-unit-iff-degree*)
qed
qed

lemma *is-unit-field-poly*[*simp*]:
fixes *p* :: ′*a*::*field poly*
shows *is-unit p* ⟷ *p* ≠ *0* ∧ *degree p* = *0*
by (*cases p*=*0*, *auto simp*: *is-unit-iff-degree*)

lemma *irreducible-smult-field*[*simp*]:

fixes *c* :: *'a* :: *field*
shows *irreducible* (*smult c p*) ⟷ *c* ≠ *0* ∧ *irreducible p* (**is** *?L* ⟷ *?R*)
proof (*intro iffI conjI* $irreducible_d$*-smult-not-zero-divisor-left*[*of c p*, *simplified*])
assume *irreducible* (*smult c p*)
then show *c* ≠ *0* **by** *auto*
next
assume *?R*
then have *c0*: *c* ≠ *0* **and** *irr*: *irreducible p* **by** *auto*
show *?L*
proof (*fold irreducible-connect-field*, *intro* $irreducible_d I$, *unfold degree-smult-eq if-not-P*[*OF c0*])
show *degree p* > *0* **using** *irr* **by** *auto*
fix *q r*
from *c0* **have** *p* = *smult* (*1*/*c*) (*smult c p*) **by** *simp*
also assume *smult c p* = *q* ∗ *r*
finally have [*simp*]: *p* = *smult* (*1*/*c*) ....
assume *main*: *degree q* < *degree p degree r* < *degree p*
have ¬$irreducible_d$ *p* **by** (*rule* $reducible_d I$, *rule exI*[*of - smult* (*1*/*c*) *q*], *rule exI*[*of - r*], *insert irr c0 main*, *simp*)
with *irr* **show** *False* **by** *auto*
qed
qed *auto*

lemma *irreducible-monic-factor*: **fixes** *p* :: *'a* :: *field poly*
assumes *degree p* > *0*
shows ∃ *q r*. *irreducible q* ∧ *p* = *q* ∗ *r* ∧ *monic q*
proof −
from $irreducible_d$*-factorization-exists*[*OF assms*]
obtain *fs* **where** *fs* ≠ [] **and** *set fs* ⊆ *Collect irreducible* **and** *p* = *prod-list fs* **by** *auto*
then have *q*: *irreducible* (*hd fs*) **and** *p*: *p* = *hd fs* ∗ *prod-list* (*tl fs*) **by** (*atomize*(*full*), *cases fs*, *auto*)
define *c* **where** *c* = *coeff* (*hd fs*) (*degree* (*hd fs*))
from *q* **have** *c*: *c* ≠ *0* **unfolding** *c-def* $irreducible_d$*-def* **by** *auto*
show *?thesis*
by (*rule exI*[*of - smult* (*1*/*c*) (*hd fs*)], *rule exI*[*of - smult c* (*prod-list* (*tl fs*))], *unfold p*,
insert q c, *auto simp*: *c-def*)
qed

lemma *monic-irreducible-factorization*: **fixes** *p* :: *'a* :: *field poly*
shows *monic p* ⟹
∃ *as f*. *finite as* ∧ *p* = *prod* (λ *a*. *a* ^ *Suc* (*f a*)) *as* ∧ *as* ⊆ {*q*. *irreducible q* ∧ *monic q*}
proof (*induct degree p arbitrary*: *p rule*: *less-induct*)
case (*less p*)
show *?case*
proof (*cases degree p* > *0*)
case *False*

with *less*(*2*) **have** $p = 1$ **by** (*simp add*: *coeff-eq-0 poly-eq-iff*)
thus *?thesis* **by** (*intro exI*[*of* - {}], *auto*)
next
case *True*
from *irreducible*$_d$*-factor*[*OF this*] **obtain** *q r* **where** *p*: $p = q * r$
and *q*: *irreducible q* **and** *deg*: *degree r* < *degree p* **by** *auto*
hence *q0*: $q \neq 0$ **by** *auto*
define *c* **where** *c* = *coeff q* (*degree q*)
let *?q* = *smult* (*1*/*c*) *q*
let *?r* = *smult c r*
from *q0* **have** *c*: $c \neq 0$ *1* / $c \neq 0$ **unfolding** *c-def* **by** *auto*
hence *p*: *p* = *?q* ∗ *?r* **unfolding** *p* **by** *auto*
have *deg*: *degree ?r* < *degree p* **using** *c deg* **by** *auto*
let *?Q* = {*q*. *irreducible q* ∧ *monic* (*q* :: '*a poly*)}
have *mon*: *monic ?q* **unfolding** *c-def* **using** *q0* **by** *auto*
from *monic-factor*[*OF* ‹*monic p*›[*unfolded p*] *this*] **have** *monic ?r* **.**
from *less*(*1*)[*OF deg this*] **obtain** *f as*
where *as*: *finite as ?r* = (∏ *a* ∈*as*. *a* ^ *Suc* (*f a*))
as ⊆ *?Q* **by** *blast*
from *q c* **have** *irred*: *irreducible ?q* **by** *simp*
show *?thesis*
proof (*cases ?q* ∈ *as*)
case *False*
let *?as* = *insert ?q as*
let *?f* = λ *a*. *if a* = *?q then 0 else f a*
have *p* = *?q* ∗ (∏ *a* ∈*as*. *a* ^ *Suc* (*f a*)) **unfolding** *p as* **by** *simp*
also have (∏ *a* ∈*as*. *a* ^ *Suc* (*f a*)) = (∏ *a* ∈*as*. *a* ^ *Suc* (*?f a*))
by (*rule prod.cong*, *insert False*, *auto*)
also have *?q* ∗ . . . = (∏ *a* ∈ *?as*. *a* ^ *Suc* (*?f a*))
by (*subst prod.insert*, *insert as False*, *auto*)
finally have *p*: *p* = (∏ *a* ∈ *?as*. *a* ^ *Suc* (*?f a*)) **.**
from *as*(*1*) **have** *fin*: *finite ?as* **by** *auto*
from *as mon irred* **have** *Q*: *?as* ⊆ *?Q* **by** *auto*
from *fin p Q* **show** *?thesis*
by(*intro exI*[*of* - *?as*] *exI*[*of* - *?f*], *auto*)
next
case *True*
let *?f* = λ *a*. *if a* = *?q then Suc* (*f a*) *else f a*
have *p* = *?q* ∗ (∏ *a* ∈*as*. *a* ^ *Suc* (*f a*)) **unfolding** *p as* **by** *simp*
also have (∏ *a* ∈*as*. *a* ^ *Suc* (*f a*)) = *?q* ^ *Suc* (*f ?q*) ∗ (∏ *a* ∈(*as* − {*?q*}).
a ^ *Suc* (*f a*))
by (*subst prod.remove*[*OF* - *True*], *insert as*, *auto*)
also have (∏ *a* ∈(*as* − {*?q*}). *a* ^ *Suc* (*f a*)) = (∏ *a* ∈(*as* − {*?q*}). *a* ^ *Suc*
(*?f a*))
by (*rule prod.cong*, *auto*)
also have *?q* ∗ (*?q* ^ *Suc* (*f ?q*) ∗ . . .) = *?q* ^ *Suc* (*?f ?q*) ∗ . . .
by (*simp add*: *ac-simps*)
also have . . . = (∏ *a* ∈ *as*. *a* ^ *Suc* (*?f a*))
by (*subst prod.remove*[*OF* - *True*], *insert as*, *auto*)

finally have $p = (\prod\ a \in as.\ a \mathbin{\hat{}} Suc\ (?f\ a))$ **.**
with *as* **show** *?thesis*
by (*intro exI*[*of - as*] *exI*[*of - ?f*], *auto*)
qed
qed
qed

lemma *monic-irreducible-gcd*:
monic (*f*::*'a*::{*field*,*euclidean-ring-gcd*,*semiring-gcd-mult-normalize*,
normalization-euclidean-semiring-multiplicative} *poly*) $\Longrightarrow$
irreducible f $\Longrightarrow$ *gcd f u* $\in$ {*1*,*f*}
by (*metis gcd-dvd1 irreducible-altdef insertCI is-unit-gcd-iff poly-dvd-antisym poly-gcd-monic*)
end

8 Connecting Polynomials with Homomorphism Locales

theory *Ring-Hom-Poly*
imports
HOL−Computational-Algebra.Euclidean-Algorithm
Ring-Hom
Missing-Polynomial
begin

poly as a homomorphism. Note that types differ.

interpretation *poly-hom*: *comm-semiring-hom* $\lambda p.$ *poly p a* **by** (*unfold-locales*, *auto*)

interpretation *poly-hom*: *comm-ring-hom* $\lambda p.$ *poly p a*..

interpretation *poly-hom*: *idom-hom* $\lambda p.$ *poly p a*..

$(\circ_p)$ as a homomorphism.

interpretation *pcompose-hom*: *comm-semiring-hom* $\lambda q.\ q \circ_p p$
using *pcompose-add pcompose-mult pcompose-1* **by** (*unfold-locales*, *auto*)

interpretation *pcompose-hom*: *comm-ring-hom* $\lambda q.\ q \circ_p p$ ..

interpretation *pcompose-hom*: *idom-hom* $\lambda q.\ q \circ_p p$ ..

definition *eval-poly* :: (*'a* $\Rightarrow$ *'b* :: *comm-semiring-1*) $\Rightarrow$ *'a* :: *zero poly* $\Rightarrow$ *'b* $\Rightarrow$ *'b*
where
[*code del*]: *eval-poly h p* = *poly* (*map-poly h p*)

lemma *eval-poly-code*[*code*]: *eval-poly h p x* = *fold-coeffs* (λ *a b*. *h a* + *x* $*$ *b*) *p 0*

by (*induct p*, *auto simp*: *eval-poly-def*)

lemma *eval-poly-as-sum*:
fixes *h* :: *'a* :: *zero* ⇒ *'b* :: *comm-semiring-1*
assumes *h 0* = *0*
shows *eval-poly h p x* = (∑ *i*≤*degree p*. *x*^*i* ∗ *h* (*coeff p i*))
unfolding *eval-poly-def*
proof (*induct p*)
case *0* **show** *?case* **using** *assms* **by** *simp*
next case (*pCons a p*) **thus** *?case*
proof (*cases p* = *0*)
case *True* **show** *?thesis* **by** (*simp add*: *True map-poly-simps assms*)
next case *False* **show** *?thesis*
unfolding *degree-pCons-eq*[*OF False*]
unfolding *sum.atMost-Suc-shift*
unfolding *map-poly-pCons*[*OF pCons*(*1*)]
by (*simp add*: *pCons*(*2*) *sum-distrib-left mult.assoc*)
qed
qed

lemma *coeff-const*: *coeff* [: *a* :] *i* = (*if i* = *0 then a else 0*)
by (*metis coeff-monom monom-0*)

lemma *x-as-monom*: [:*0*,*1*:] = *monom 1 1*
by (*simp add*: *monom-0 monom-Suc*)

lemma *x-pow-n*: *monom 1 1* ^ *n* = *monom 1 n*
by (*induct n*) (*simp-all add*: *monom-0 monom-Suc*)

lemma *map-poly-eval-poly*: **assumes** *h0*: *h 0* = *0*
shows *map-poly h p* = *eval-poly* (λ *a*. [: *h a* :]) *p* [:*0*,*1*:] (**is** *?mp* = *?ep*)
proof (*rule poly-eqI*)
fix *i* :: *nat*
have *2*: (∑ *x*≤*i*. ∑ *xa*≤*degree p*. (*if xa* = *x then 1 else 0*) ∗ *coeff* [:*h* (*coeff p xa*):] (*i* − *x*))
= *h* (*coeff p i*) (**is** *sum ?f ?s* = *?r*)
proof −
have *sum ?f ?s* = *?f i* + *sum ?f* ({..*i*} − {*i*})
by (*rule sum.remove*[*of - i*], *auto*)
also have *sum ?f* ({..*i*} − {*i*}) = *0*
by (*rule sum.neutral*, *intro ballI*, *rule sum.neutral*, *auto simp*: *coeff-const*)
also have *?f i* = (∑ *xa*≤*degree p*. (*if xa* = *i then 1 else 0*) ∗ *h* (*coeff p xa*)) (**is** - = *?m*)
unfolding *coeff-const* **by** *simp*
also have ... = *?r*
proof (*cases i* ≤ *degree p*)
case *True*
show *?thesis*
by (*subst sum.remove*[*of - i*], *insert True*, *auto*)

next
case *False*
hence [*simp*]: *coeff p i* = *0* **using** *le-degree* **by** *blast*
show *?thesis*
by (*subst sum.neutral*, *auto simp*: *h0*)
qed
finally show *?thesis* **by** *simp*
qed
have *h'0*: [: *h 0* :] = *0* **using** *h0* **by** *auto*
show *coeff ?mp i* = *coeff ?ep i*
unfolding *coeff-map-poly*[*of h*, *OF h0*]
unfolding *eval-poly-as-sum*[*of* λ*a*. [: *h a* :], *OF h'0*]
unfolding *coeff-sum*
unfolding *x-as-monom x-pow-n coeff-mult*
unfolding *sum.swap*[*of - -* {..*degree p*}]
unfolding *coeff-monom* **using** *2* **by** *auto*
qed

lemma *smult-as-map-poly*: *smult a* = *map-poly* ((∗) *a*)
by (*rule ext*, *rule poly-eqI*, *subst coeff-map-poly*, *auto*)

8.1 *map-poly* of Homomorphisms

context *zero-hom* **begin**

We will consider *hom* is always simpler than *map-poly hom*.

lemma *map-poly-hom-monom*[*simp*]: *map-poly hom* (*monom a i*) = *monom* (*hom a*) *i*
by(*rule map-poly-monom*, *auto*)
lemma *coeff-map-poly-hom*[*simp*]: *coeff* (*map-poly hom p*) *i* = *hom* (*coeff p i*)
by (*rule coeff-map-poly*, *rule hom-zero*)
end

locale *map-poly-zero-hom* = *base*: *zero-hom*
begin
sublocale *zero-hom map-poly hom* **by** (*unfold-locales*, *auto*)
end

map-poly preserves homomorphisms over addition.

context *comm-monoid-add-hom*
begin
lemma *map-poly-hom-add*[*hom-distribs*]:
map-poly hom (*p* + *q*) = *map-poly hom p* + *map-poly hom q*
by (*rule map-poly-add*; *simp add*: *hom-distribs*)
end

locale *map-poly-comm-monoid-add-hom* = *base*: *comm-monoid-add-hom*
begin
sublocale *comm-monoid-add-hom map-poly hom* **by** (*unfold-locales*, *auto simp*:*hom-distribs*)
end

To preserve homomorphisms over multiplication, it demands commutative ring homomorphisms.

context *comm-semiring-hom* **begin**
lemma *map-poly-pCons-hom*[*hom-distribs*]: *map-poly hom* (*pCons a p*) = *pCons* (*hom a*) (*map-poly hom p*)
unfolding *map-poly-simps* **by** *auto*
lemma *map-poly-hom-smult*[*hom-distribs*]:
map-poly hom (*smult c p*) = *smult* (*hom c*) (*map-poly hom p*)
by (*induct p*, *auto simp*: *hom-distribs*)
lemma *poly-map-poly*[*simp*]: *poly* (*map-poly hom p*) (*hom x*) = *hom* (*poly p x*)
by (*induct p*; *simp add*: *hom-distribs*)
end

locale *map-poly-comm-semiring-hom* = *base*: *comm-semiring-hom*
begin
sublocale *map-poly-comm-monoid-add-hom***..**
sublocale *comm-semiring-hom map-poly hom*
proof
show *map-poly hom 1* = *1* **by** *simp*
fix *p q* **show** *map-poly hom* (*p* ∗ *q*) = *map-poly hom p* ∗ *map-poly hom q*
by (*induct p*, *auto simp*: *hom-distribs*)
qed
end

locale *map-poly-comm-ring-hom* = *base*: *comm-ring-hom*
begin
sublocale *map-poly-comm-semiring-hom***..**
sublocale *comm-ring-hom map-poly hom***..**
end

locale *map-poly-idom-hom* = *base*: *idom-hom*
begin
sublocale *map-poly-comm-ring-hom***..**
sublocale *idom-hom map-poly hom***..**
end

8.1.1 Injectivity

locale *map-poly-inj-zero-hom* = *base*: *inj-zero-hom*
begin
sublocale *inj-zero-hom map-poly hom*
proof (*unfold-locales*)
fix *p q* :: *'a poly* **assume** *map-poly hom p* = *map-poly hom q*
from *cong*[*of* λ*p. coeff p* -, *OF refl this*] **show** *p* = *q* **by** (*auto intro*: *poly-eqI*)
qed *simp*
end

locale *map-poly-inj-comm-monoid-add-hom* = *base*: *inj-comm-monoid-add-hom*
begin

sublocale *map-poly-comm-monoid-add-hom*..
sublocale *map-poly-inj-zero-hom*..
sublocale *inj-comm-monoid-add-hom map-poly hom*..
end

locale *map-poly-inj-comm-semiring-hom* = *base*: *inj-comm-semiring-hom*
begin
sublocale *map-poly-comm-semiring-hom*..
sublocale *map-poly-inj-zero-hom*..
sublocale *inj-comm-semiring-hom map-poly hom*..
end

locale *map-poly-inj-comm-ring-hom* = *base*: *inj-comm-ring-hom*
begin
sublocale *map-poly-inj-comm-semiring-hom*..
sublocale *inj-comm-ring-hom map-poly hom*..
end

locale *map-poly-inj-idom-hom* = *base*: *inj-idom-hom*
begin
sublocale *map-poly-inj-comm-ring-hom*..
sublocale *inj-idom-hom map-poly hom*..
end

lemma *degree-map-poly-le*: *degree* (*map-poly f p*) ≤ *degree p*
by(*induct p*;*auto*)

lemma *coeffs-map-poly*:
assumes *f* (*lead-coeff p*) = *0* ⟷ *p* = *0*
shows *coeffs* (*map-poly f p*) = *map f* (*coeffs p*)
unfolding *coeffs-map-poly* **using** *assms* **by** (*simp add*:*coeffs-def*)

lemma *degree-map-poly*:
assumes *f* (*lead-coeff p*) = *0* ⟷ *p* = *0*
shows *degree* (*map-poly f p*) = *degree p*
unfolding *degree-eq-length-coeffs* **unfolding** *coeffs-map-poly*[*of f*, *OF assms*] **by** *simp*

context *zero-hom-0* **begin**
lemma *degree-map-poly-hom*[*simp*]: *degree* (*map-poly hom p*) = *degree p*
by (*rule degree-map-poly*, *auto*)
lemma *coeffs-map-poly-hom*[*simp*]: *coeffs* (*map-poly hom p*) = *map hom* (*coeffs p*)
by (*rule coeffs-map-poly*, *auto*)
lemma *hom-lead-coeff*[*simp*]: *lead-coeff* (*map-poly hom p*) = *hom* (*lead-coeff p*)
by *simp*

end

context *comm-semiring-hom* **begin**

interpretation *map-poly-hom*: *map-poly-comm-semiring-hom*..

lemma *poly-map-poly-0*[*simp*]:
poly (*map-poly hom p*) *0* = *hom* (*poly p 0*) (**is** *?l* = *?r*)
proof−
have *?l* = *poly* (*map-poly hom p*) (*hom 0*) **by** *auto*
then show *?thesis* **unfolding** *poly-map-poly*.
qed

lemma *poly-map-poly-1*[*simp*]:
poly (*map-poly hom p*) *1* = *hom* (*poly p 1*) (**is** *?l* = *?r*)
proof−
have *?l* = *poly* (*map-poly hom p*) (*hom 1*) **by** *auto*
then show *?thesis* **unfolding** *poly-map-poly*.
qed

lemma *map-poly-hom-as-monom-sum*:
($\sum$*j*≤*degree p*. *monom* (*hom* (*coeff p j*)) *j*) = *map-poly hom p*
proof −
show *?thesis*
by (*subst*(*6*) *poly-as-sum-of-monoms'*[*OF le-refl*, *symmetric*], *simp add*: *hom-distribs*)
qed

lemma *map-poly-pcompose*[*hom-distribs*]:
map-poly hom (*f* $\circ_p$ *g*) = *map-poly hom f* $\circ_p$ *map-poly hom g*
by (*induct f arbitrary*: *g*; *auto simp*: *hom-distribs*)

end

context *comm-semiring-hom* **begin**

lemma *eval-poly-0*[*simp*]: *eval-poly hom 0 x* = *0* **unfolding** *eval-poly-def* **by** *simp*
lemma *eval-poly-monom*: *eval-poly hom* (*monom a n*) *x* = *hom a* ∗ *x* ^ *n*
unfolding *eval-poly-def*
unfolding *map-poly-monom*[*of hom*, *OF hom-zero*] **using** *poly-monom*.

lemma *poly-map-poly-eval-poly*: *poly* (*map-poly hom p*) = *eval-poly hom p*
unfolding *eval-poly-def*..

lemma *map-poly-eval-poly*:
map-poly hom p = *eval-poly* (λ *a*. [: *hom a* :]) *p* [:*0*,*1*:]
by (*rule map-poly-eval-poly*, *simp*)

lemma *degree-extension*: **assumes** *degree p* ≤ *n*

shows ($\sum$ *i*≤*degree p*. *x* ^ *i* ∗ *hom* (*coeff p i*))
= ($\sum$ *i*≤*n*. *x* ^ *i* ∗ *hom* (*coeff p i*)) (**is** *?l* = *?r*)
proof −
let *?f* = λ *i*. *x* ^ *i* ∗ *hom* (*coeff p i*)
define *m* **where** *m* = *n* − *degree p*
have *n*: *n* = *degree p* + *m* **unfolding** *m-def* **using** *assms* **by** *auto*
have *?r* = ($\sum$ *i* ≤ *degree p* + *m*. *?f i*) **unfolding** *n* **..**
also have . . . = *?l* + *sum ?f* {*Suc* (*degree p*) .. *degree p* + *m*}
by (*subst sum.union-disjoint*[*symmetric*], *auto intro*: *sum.cong*)
also have *sum ?f* {*Suc* (*degree p*) .. *degree p* + *m*} = *0*
by (*rule sum.neutral*, *auto simp*: *coeff-eq-0*)
finally show *?thesis* **by** *simp*
qed

lemma *eval-poly-add*[*simp*]: *eval-poly hom* (*p* + *q*) *x* = *eval-poly hom p x* + *eval-poly hom q x*
unfolding *eval-poly-def hom-distribs***..**

lemma *eval-poly-sum*: *eval-poly hom* ($\sum$ *k*∈*A*. *p k*) *x* = ($\sum$ *k*∈*A*. *eval-poly hom* (*p k*) *x*)
proof (*induct A rule*: *infinite-finite-induct*)
case (*insert a A*)
show *?case*
unfolding *sum.insert*[*OF insert*(*1*−*2*)] *insert*(*3*)[*symmetric*] **by** *simp*
qed (*auto simp*: *eval-poly-def*)

lemma *eval-poly-poly*: *eval-poly hom p* (*hom x*) = *hom* (*poly p x*)
unfolding *eval-poly-def* **by** *auto*

end

context *comm-ring-hom* **begin**
interpretation *map-poly-hom*: *map-poly-comm-ring-hom***..**

lemma *pseudo-divmod-main-hom*:
pseudo-divmod-main (*hom lc*) (*map-poly hom q*) (*map-poly hom r*) (*map-poly hom d*) *dr i* =
map-prod (*map-poly hom*) (*map-poly hom*) (*pseudo-divmod-main lc q r d dr i*)
proof−
show *?thesis* **by** (*induct lc q r d dr i rule*:*pseudo-divmod-main.induct*, *auto simp*: *Let-def hom-distribs*)
qed
end

lemma(**in** *inj-comm-ring-hom*) *pseudo-divmod-hom*:
pseudo-divmod (*map-poly hom p*) (*map-poly hom q*) =
map-prod (*map-poly hom*) (*map-poly hom*) (*pseudo-divmod p q*)
unfolding *pseudo-divmod-def* **using** *pseudo-divmod-main-hom*[*of* - *0*] **by** (*cases q* = *0*,*auto*)

lemma(**in** *inj-idom-hom*) *pseudo-mod-hom*:
pseudo-mod (*map-poly hom p*) (*map-poly hom q*) = *map-poly hom* (*pseudo-mod p q*)
using *pseudo-divmod-hom* **unfolding** *pseudo-mod-def* **by** *auto*

lemma(**in** *idom-hom*) *map-poly-pderiv*[*hom-distribs*]:
map-poly hom (*pderiv p*) = *pderiv* (*map-poly hom p*)
proof (*induct p rule*: *pderiv.induct*)
case (*1 a p*)
then show *?case* **unfolding** *pderiv.simps map-poly-pCons-hom* **by** (*cases p = 0*, *auto simp*: *hom-distribs*)
qed

lemma(**in** *idom-hom*) *map-poly-higher-pderiv*[*hom-distribs*]:
map-poly hom ((*pderiv* ^^ *n*) *p*) = (*pderiv* ^^ *n*) (*map-poly hom p*)
by (*induction n*) (*auto simp*: *hom-distribs*)

context *field-hom*
begin

lemma *dvd-map-poly-hom-imp-dvd*: ‹*map-poly hom x dvd map-poly hom y* ⟹ *x dvd y*›
by (*smt* (*verit*, *del-insts*) *degree-map-poly-hom hom-0 hom-div hom-lead-coeff hom-one hom-power map-poly-hom-smult map-poly-zero mod-eq-0-iff-dvd mod-poly-def pseudo-mod-hom*)

lemma *map-poly-pdivmod* [*hom-distribs*]:
‹*map-prod* (*map-poly hom*) (*map-poly hom*) (*p div q*, *p mod q*) =
(*map-poly hom p div map-poly hom q*, *map-poly hom p mod map-poly hom q*)›
proof −
let *?mp* = ‹*map-poly hom*›
interpret *map-poly-hom*: *map-poly-idom-hom* **..**
have ‹(*?mp p div ?mp q*, *?mp p mod ?mp q*) = (*?mp* (*p div q*), *?mp* (*p mod q*))›
proof (*induction rule*: *euclidean-relation-polyI*)
case *by0*
then show *?case*
by *simp*
next
case *divides*
then have ‹*q* ≠ *0*› ‹*q dvd p*›
by (*auto dest*: *dvd-map-poly-hom-imp-dvd*)
from ‹*q dvd p*› **obtain** *r* **where** ‹*p* = *q* ∗ *r*› **..**
with ‹*q* ≠ *0*› **show** *?case*
by (*simp add*: *map-poly-hom.hom-mult*)
next
case *euclidean-relation*
with *degree-mod-less-degree* [*of q p*] **show** *?case*
by (*auto simp flip*: *map-poly-hom.hom-mult map-poly-hom-add*)

qed
then show *?thesis*
by *simp*
qed

lemma *map-poly-div*[*hom-distribs*]: *map-poly hom* (*p div q*) = *map-poly hom p div map-poly hom q*
using *map-poly-pdivmod*[*of p q*] **by** *simp*

lemma *map-poly-mod*[*hom-distribs*]: *map-poly hom* (*p mod q*) = *map-poly hom p mod map-poly hom q*
using *map-poly-pdivmod*[*of p q*] **by** *simp*

end

locale *field-hom′* = *field-hom hom*
for *hom* :: ′*a* :: {*field-gcd*} ⇒ ′*b* :: {*field-gcd*}
begin

lemma *map-poly-normalize*[*hom-distribs*]: *map-poly hom* (*normalize p*) = *normalize* (*map-poly hom p*)
by (*simp add*: *normalize-poly-def hom-distribs*)

lemma *map-poly-gcd*[*hom-distribs*]: *map-poly hom* (*gcd p q*) = *gcd* (*map-poly hom p*) (*map-poly hom q*)
by (*induct p q rule*: *eucl-induct*)
(*simp-all add*: *map-poly-normalize ac-simps hom-distribs*)

end

definition *div-poly* :: ′*a* :: *euclidean-semiring* ⇒ ′*a poly* ⇒ ′*a poly* **where**
div-poly a p = *map-poly* (λ *c*. *c div a*) *p*

lemma *smult-div-poly*: **assumes** ⋀ *c*. *c* ∈ *set* (*coeffs p*) ⟹ *a dvd c*
shows *smult a* (*div-poly a p*) = *p*
unfolding *smult-as-map-poly div-poly-def*
by (*subst map-poly-map-poly*, *force*, *subst map-poly-idI*, *insert assms*, *auto*)

lemma *coeff-div-poly*: *coeff* (*div-poly a f*) *n* = *coeff f n div a*
unfolding *div-poly-def*
by (*rule coeff-map-poly*, *auto*)

locale *map-poly-inj-idom-divide-hom* = *base*: *inj-idom-divide-hom*
begin
sublocale *map-poly-idom-hom* **..**
sublocale *map-poly-inj-zero-hom* **..**
sublocale *inj-idom-hom map-poly hom* **..**
lemma *divide-poly-main-hom*: **defines** *hh* ≡ *map-poly hom*
shows *hh* (*divide-poly-main lc f g h i j*) = *divide-poly-main* (*hom lc*) (*hh f*) (*hh*

g) (*hh h*) *i j*
  unfolding *hh-def*
proof (*induct j arbitrary*: *lc f g h i*)
  case (*Suc j lc f g h i*)
  let *?h* = *map-poly hom*
  show *?case* **unfolding** *divide-poly-main.simps Let-def*
   unfolding *base.coeff-map-poly-hom base.hom-div*[*symmetric*] *base.hom-mult*[*symmetric*] *base.eq-iff*
      if-distrib[*of ?h*] *hom-zero*
      by (*rule if-cong*[*OF refl - refl*], *subst Suc*, *simp add*: *hom-minus hom-add hom-mult*)
qed *simp*

sublocale *inj-idom-divide-hom map-poly hom*
proof
  fix *f g* :: *'a poly*
  let *?h* = *map-poly hom*
  show *?h* (*f div g*) = (*?h f*) *div* (*?h g*) **unfolding** *divide-poly-def if-distrib*[*of ?h*]
    divide-poly-main-hom **by** *simp*
qed

lemma *order-hom*: *order* (*hom x*) (*map-poly hom f*) = *order x f*
  unfolding *Polynomial.order-def* **unfolding** *hom-dvd-iff* [*symmetric*]
  unfolding *hom-power* **by** (*simp add*: *base.hom-uminus*)
end

8.2 Example Interpretations

abbreviation *of-int-poly* ≡ *map-poly of-int*

interpretation *of-int-poly-hom*: *map-poly-comm-semiring-hom of-int*..
interpretation *of-int-poly-hom*: *map-poly-comm-ring-hom of-int*..
interpretation *of-int-poly-hom*: *map-poly-idom-hom of-int*..
interpretation *of-int-poly-hom*:
  map-poly-inj-comm-ring-hom of-int :: *int* ⇒ *'a* :: {*comm-ring-1*,*ring-char-0*} ..
interpretation *of-int-poly-hom*:
  map-poly-inj-idom-hom of-int :: *int* ⇒ *'a* :: {*idom*,*ring-char-0*} ..

The following operations are homomorphic w.r.t. only *monoid-add*.

interpretation *pCons-0-hom*: *injective pCons 0* **by** (*unfold-locales*, *auto*)
interpretation *pCons-0-hom*: *zero-hom-0 pCons 0* **by** (*unfold-locales*, *auto*)
interpretation *pCons-0-hom*: *inj-comm-monoid-add-hom pCons 0* **by** (*unfold-locales*, *auto*)
interpretation *pCons-0-hom*: *inj-ab-group-add-hom pCons 0* **by** (*unfold-locales*, *auto*)

interpretation *monom-hom*: *injective* λ*x*. *monom x d* **by** (*unfold-locales*, *auto*)
interpretation *monom-hom*: *inj-monoid-add-hom* λ*x*. *monom x d* **by** (*unfold-locales*, *auto simp*: *add-monom*)
interpretation *monom-hom*: *inj-comm-monoid-add-hom* λ*x*. *monom x d*..

end

9 Newton Interpolation

We proved the soundness of the Newton interpolation, i.e., a method to interpolate a polynomial p from a list of points $(x_1, p(x_1)), (x_2, p(x_2)), \ldots$. In experiments it performs much faster than the Lagrange interpolation.

theory *Newton-Interpolation*
imports
  HOL−Library.Monad-Syntax
  Ring-Hom-Poly
  Divmod-Int
  Is-Rat-To-Rat
begin

For the Newton interpolation, we start with an efficient implementation (which in prior examples we used as an uncertified oracle). Later on, a more abstract definition of the algorithm is described for which soundness is proven, and which is provably equivalent to the efficient implementation.

The implementation is based on divided differences and the Horner schema.

```
fun horner-composition :: 'a :: comm-ring-1 list ⇒ 'a list ⇒ 'a poly where
  horner-composition [cn] xis = [:cn:]
| horner-composition (ci # cs) (xi # xis) = horner-composition cs xis ∗ [:− xi, 1:]
+ [:ci:]
| horner-composition - - = 0

lemma (in map-poly-comm-ring-hom) horner-composition-hom:
 horner-composition (map hom cs) (map hom xs) = map-poly hom (horner-composition
cs xs)
  by (induct cs xs rule: horner-composition.induct, auto simp: hom-distribs)

lemma horner-coeffs-ints: assumes len: length cs ≤ Suc (length ys)
  shows (set (coeffs (horner-composition cs (map rat-of-int ys))) ⊆ ℤ) = (set cs
⊆ ℤ)
proof −
  let ?ir = int-of-rat
  let ?ri = rat-of-int
  let ?mir = map ?ir
  let ?mri = map ?ri
  show ?thesis
  proof
    define ics where ics = map ?ir cs
    assume set cs ⊆ ℤ
    hence ics: cs = ?mri ics unfolding ics-def map-map o-def
      by (simp add: map-idI subset-code(1))
```

show *set* (*coeffs* (*horner-composition cs* (*?mri ys*))) ⊆ ℤ
unfolding *ics of-int-poly-hom.horner-composition-hom* **by** *auto*
next
assume *set* (*coeffs* (*horner-composition cs* (*?mri ys*))) ⊆ ℤ
thus *set cs* ⊆ ℤ **using** *len*
proof (*induct cs arbitrary*: *ys*)
case (*Cons c cs xs*)
show *?case*
proof (*cases cs* = [] ∨ *xs* = [])
case *True*
with *Cons* **show** *?thesis* **by** (*cases c* = *0*; *cases cs*, *auto*)
next
case *False*
then obtain *d ds* **and** *y ys* **where** *cs*: *cs* = *d* # *ds* **and** *xs*: *xs* = *y* # *ys*
by (*cases cs*, *auto*, *cases xs*, *auto*)
let *?q* = *horner-composition cs* (*?mri ys*)
define *q* **where** *q* = *?q*
define *p* **where** *p* = *q* ∗ [:− *?ri y*, *1*:] + [:*c*:]
have *id*: *horner-composition* (*c* # *cs*) (*?mri xs*) = *p*
unfolding *cs xs q-def p-def* **by** *simp*
have *coeff*: *coeff p i* ∈ ℤ **for** *i*
proof (*cases coeff p i* ∈ *set* (*coeffs p*))
case *True*
with *Cons*(*2*)[*unfolded id*] **show** *?thesis* **by** *blast*
next
case *False*
hence *coeff p i* = *0* **using** *range-coeff*[*of p*] **by** *blast*
thus *?thesis* **by** *simp*
qed
{
fix *i*
let *?f* = λ *j*. *coeff* [:− *?ri y*, *1*:] *j* ∗ *coeff q* (*Suc i* − *j*)
have *coeff p* (*Suc i*) = *coeff* ([: −*?ri y*, *1* :] ∗ *q*) (*Suc i*) **unfolding** *p-def*
by *simp*
also have ... = (∑*j*≤*Suc i*. *?f j*) **unfolding** *coeff-mult* **by** *simp*
also have ... = *?f 0* + *?f 1* + (∑*j*∈{..*Suc i*} − {*0*} − {*Suc 0*}. *?f j*)
by (*subst sum.remove*[*of - 0*], *force*+, *subst sum.remove*[*of - 1*], *force*+)
also have (∑*j*∈{..*Suc i*} − {*0*} − {*Suc 0*}. *?f j*) = *0*
proof (*rule sum.neutral*, *auto*, *goal-cases*)
case (*1 x*)
thus *?case* **by** (*cases x*, *auto*, *cases x* − *1*, *auto*)
qed
also have *?f 0* = − *?ri y* ∗ *coeff q* (*Suc i*) **by** *simp*
also have *?f 1* = *coeff q i* **by** *simp*
finally have *int*: *coeff q i* − *?ri y* ∗ *coeff q* (*Suc i*) ∈ ℤ **using** *coeff*[*of Suc i*] **by** *auto*
assume *coeff q* (*Suc i*) ∈ ℤ
hence *?ri y* ∗ *coeff q* (*Suc i*) ∈ ℤ **by** *simp*
hence *coeff q i* ∈ ℤ **using** *int Ints-diff Ints-minus* **by** *force*

```
        } note coeff-q = this
        {
          fix i
          assume i ≤ degree q
          hence coeff q (degree q − i) ∈ ℤ
          proof (induct i)
            case 0
            from coeff-q[of degree q] show ?case
              by (metis Ints-0 Suc-n-not-le-n diff-zero le-degree)
          next
            case (Suc i)
            with coeff-q[of i] show ?case
              by (metis Suc-diff-Suc Suc-leD Suc-n-not-le-n coeff-q le-less)
          qed
        } note coeff-q = this
        {
          fix i
          have coeff q i ∈ ℤ
          proof (cases i ≤ degree q)
            case True
            with coeff-q[of degree q − i] show ?thesis by auto
          next
            case False
            hence coeff q i = 0 using le-degree by blast
            thus ?thesis by simp
          qed
        } note coeff-q = this
        hence set (coeffs q) ⊆ ℤ by (auto simp: coeffs-def)

        from Cons(1)[OF this[unfolded q-def]] Cons(3) xs have IH: set cs ⊆ ℤ by
auto
        define r where r = coeff q 0 ∗ (− ?ri y)
        have r: r ∈ ℤ using coeff-q[of 0] unfolding r-def by auto
        have coeff p 0 ∈ ℤ by fact
        also have coeff p 0 = r + c unfolding p-def r-def by simp
        finally have c: c ∈ ℤ using r using Ints-diff by force
        with IH show ?thesis by auto
      qed
    qed simp
  qed
qed

context
fixes
  ty :: 'a :: field itself
  and xs :: 'a list
  and fs :: 'a list
begin
```

fun *divided-differences-impl* :: *'a list* ⇒ *'a* ⇒ *'a* ⇒ *'a list* ⇒ *'a list* **where**
divided-differences-impl (*xi-j1* # *x-j1s*) *fj xj* (*xi* # *xis*) = (*let*
x-js = *divided-differences-impl x-j1s fj xj xis*;
new = (*hd x-js* − *xi-j1*) / (*xj* − *xi*)
in new # *x-js*)
| *divided-differences-impl* [] *fj xj xis* = [*fj*]

fun *newton-coefficients-main* :: *'a list* ⇒ *'a list* ⇒ *'a list list* **where**
newton-coefficients-main [*fj*] *xjs* = [[*fj*]]
| *newton-coefficients-main* (*fj* # *fjs*) (*xj* # *xjs*) = (
let rec = *newton-coefficients-main fjs xjs*; *row* = *hd rec*;
new-row = *divided-differences-impl row fj xj xs*
in new-row # *rec*)
| *newton-coefficients-main* - - = []

definition *newton-coefficients* :: *'a list* **where**
newton-coefficients = *map hd* (*newton-coefficients-main* (*rev fs*) (*rev xs*))

definition *newton-poly-impl* :: *'a poly* **where**
newton-poly-impl = *horner-composition* (*rev newton-coefficients*) *xs*

qualified definition *x i* = *xs* ! *i*
qualified definition *f i* = *fs* ! *i*

private definition *xd i j* = *x i* − *x j*

lemma [*simp*]: *xd i i* = *0 xd i j* + *xd j k* = *xd i k xd i j* + *xd k i* = *xd k j*
unfolding *xd-def* **by** *simp-all*

private function *xij-f* :: *nat* ⇒ *nat* ⇒ *'a* **where**
xij-f i j = (*if i* < *j then* (*xij-f* (*i* + *1*) *j* − *xij-f i* (*j* − *1*)) / *xd j i else f i*)
by *pat-completeness auto*

termination by (*relation measure* (λ (*i*,*j*). *j* − *i*), *auto*)

private definition *c* :: *nat* ⇒ *'a* **where**
c i = *xij-f 0 i*

private definition *X j* = [: − *x j*, *1*:]

private function *b* :: *nat* ⇒ *nat* ⇒ *'a poly* **where**
b i n = (*if i* ≥ *n then* [:*c n*:] *else b* (*Suc i*) *n* ∗ *X i* + [:*c i*:])
by *pat-completeness auto*

termination by (*relation measure* (λ (*i*,*n*). *Suc n* − *i*), *auto*)

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

```
definition newton-poly :: nat ⇒ 'a poly where
  newton-poly n = b 0 n

private definition Xij i j = prod-list (map X [i ..< j])

private definition N i = Xij 0 i

lemma Xii-1[simp]: Xij i i = 1 unfolding Xij-def by simp
lemma smult-1[simp]: smult d 1 = [:d:]
  by (fact smult-one)

private lemma newton-poly-sum:
  newton-poly n = sum-list (map (λ i. smult (c i) (N i)) [0 ..< Suc n])
  unfolding newton-poly-def N-def
proof −
  {
    fix j
    assume j ≤ n
    hence b j n = (∑ i←[j..<Suc n]. smult (c i) (Xij j i))
    proof (induct j n rule: b.induct)
      case (1 j n)
      show ?case
      proof (cases j ≥ n)
        case True
        with 1(2) have j: j = n by auto
        hence b j n = [:c n:] unfolding b.simps[of j n] by simp
        thus ?thesis unfolding j by simp
      next
        case False
        hence b: b j n = b (Suc j) n ∗ X j + [: c j:] unfolding b.simps[of j n] by
simp
        define nn where nn = Suc n
        from 1(2) have id: [j..< nn] = j # [Suc j ..< nn] unfolding nn-def by
(simp add: upt-rec)
        from False have Suc j ≤ n by auto
        note IH = 1(1)[OF False this]
        have id2: (∑ x←[Suc j..< nn]. smult (c x) (Xij (Suc j) x ∗ X j)) =
          (∑ i←[Suc j..< nn]. smult (c i) (Xij j i))
        proof (rule arg-cong[of - - sum-list], rule map-ext, intro impI, goal-cases)
          case (1 i)
          hence Xij (Suc j) i ∗ X j = Xij j i by (simp add: Xij-def upt-conv-Cons)
          thus ?case by simp
        qed
        show ?thesis unfolding b IH sum-list-mult-const[symmetric]
          unfolding nn-def[symmetric] id
          by (simp add: id2)
      qed
    qed
```

}
from *this*[*of 0*] **show** *b 0 n* = (∑ *i*←[*0*..<*Suc n*]. *smult* (*c i*) (*Xij 0 i*)) **by** *simp*
qed

private lemma *poly-newton-poly*: *poly* (*newton-poly n*) *y* = *sum-list* (*map* (λ *i*. *c i* ∗ *poly* (*N i*) *y*) [*0* ..< *Suc n*])
unfolding *newton-poly-sum poly-sum-list map-map o-def* **by** *simp*

private definition *pprod k i j* = (∏ *l*←[*i*..<*j*]. *xd k l*)

private lemma *poly-N-xi*: *poly* (*N i*) (*x j*) = *pprod j 0 i*
proof −
have *poly* (*N i*) (*x j*) = (∏ *l*←[*0*..<*i*]. *xd j l*)
unfolding *N-def Xij-def poly-prod-list X-def*[*abs-def*] *map-map o-def xd-def* **by** *simp*
also have . . . = *pprod j 0 i* **unfolding** *pprod-def* **..**
finally show *?thesis* **.**
qed

private lemma *poly-N-xi-cond*: *poly* (*N i*) (*x j*) = (*if j* < *i then 0 else pprod j 0 i*)
proof −
show *?thesis*
proof (*cases j* < *i*)
case *False*
thus *?thesis* **using** *poly-N-xi* **by** *simp*
next
case *True*
hence *j* ∈ *set* [*0* ..< *i*] **by** *auto*
from *split-list*[*OF this*] **obtain** *bef aft* **where** *id2*: [*0* ..< *i*] = *bef* @ *j* # *aft* **by** *auto*
have (∏ *k*←[*0*..<*i*]. *xd j k*) = *0* **unfolding** *id2* **by** *auto*
with *True* **show** *?thesis* **unfolding** *poly-N-xi pprod-def* **by** *auto*
qed
qed

private lemma *poly-newton-poly-xj*: **assumes** *j* ≤ *n*
shows *poly* (*newton-poly n*) (*x j*) = *sum-list* (*map* (λ *i*. *c i* ∗ *poly* (*N i*) (*x j*)) [*0* ..< *Suc j*])
proof −
from *assms* **have** *id*: [*0* ..< *Suc n*] = [*0* ..< *Suc j*] @ [*Suc j* ..< *Suc n*]
by (*metis Suc-le-mono le-Suc-ex less-eq-nat.simps*(*1*) *upt-add-eq-append*)
have *id2*: (∑ *i*←[*Suc j*..< *Suc n*]. *c i* ∗ *poly* (*N i*) (*x j*)) = *0*
by (*rule sum-list-neutral*, *unfold poly-N-xi-cond*, *auto*)
show *?thesis* **unfolding** *poly-newton-poly id map-append sum-list-append id2* **by** *simp*
qed

declare *xij-f.simps*[*simp del*]

context
fixes *n*
assumes *dist*: ⋀ *i j*. *i* < *j* ⟹ *j* ≤ *n* ⟹ *x i* ≠ *x j*
begin
private lemma *xd-diff*: *i* < *j* ⟹ *j* ≤ *n* ⟹ *xd i j* ≠ *0*
i < *j* ⟹ *j* ≤ *n* ⟹ *xd j i* ≠ *0* **using** *dist*[*of i j*] *dist*[*of j i*] **unfolding** *xd-def*
by *auto*

This is the key technical lemma for soundness of Newton interpolation.

private lemma *divided-differences-main*: **assumes** *k* ≤ *n i* < *k*
shows *sum-list* (*map* (λ *j*. *xij-f i* (*i* + *j*) ∗ *pprod k i* (*i* + *j*)) [*0*..<*Suc k* − *i*]) =
sum-list (*map* (λ *j*. *xij-f* (*Suc i*) (*Suc i* + *j*) ∗ *pprod k* (*Suc i*) (*Suc i* + *j*)) [*0*..<*Suc k* − *Suc i*])
proof −
let *?exp* = λ *i j*. *xij-f i* (*i* + *j*) ∗ *pprod k i* (*i* + *j*)
define *ei* **where** *ei* = *?exp i*
define *esi* **where** *esi* = *?exp* (*Suc i*)
let *?ki* = *k* − *i*
let *?sumi* = λ *xs*. *sum-list* (*map ei xs*)
let *?sumsi* = λ *xs*. *sum-list* (*map esi xs*)
let *?mid* = λ *j*. *xij-f i* (*k* − *j*) ∗ *pprod k* (*Suc i*) (*k* − *j*) ∗ *xd* (*k* − *j*) *i*
let *?sum* = λ *j*. *?sumi* [*0* ..< *?ki* − *j*] + *?sumsi* [*?ki* − *j* ..< *?ki*] + *?mid j*
define *fin* **where** *fin* = *?ki* − *1*
have *fin*: *fin* < *?ki* **unfolding** *fin-def* **using** *assms* **by** *auto*
have *id*: [*0* ..< *Suc k* − *i*] = [*0* ..< *?ki*] @ [*?ki*] **and**
id2: [*i*..<*k*] = *i* # [*Suc i* ..< *k*] **and**
id3: *k* − (*i* + (*k* − *Suc i*)) = *1 k* − (*?ki* − *1*) = *Suc i* **using** *assms*
by (*auto simp*: *Suc-diff-le upt-conv-Cons*)
have *neq*: *xd* (*Suc i*) *i* ≠ *0* **using** *xd-diff*[*of i Suc i*] *assms* **by** *auto*
have *sum-list* (*map* (λ *j*. *xij-f i* (*i* + *j*) ∗ *pprod k i* (*i* + *j*)) [*0*..<*Suc k* − *i*])
= *?sumi* [*0* ..< *Suc k* − *i*] **unfolding** *ei-def* **by** *simp*
also have . . . = *?sumi* [*0* ..< *?ki*] + *?sumsi* [*?ki* ..< *?ki*] + *ei ?ki*
unfolding *id* **by** *simp*
also have . . . = *?sum 0*
unfolding *ei-def* **using** *assms* **by** (*simp add*: *pprod-def id2*)
also have *?sum 0* = *?sum fin* **using** *fin*
proof (*induct fin*)
case (*Suc fin*)
from *Suc*(*2*) *assms*
have *fki*: *fin* < *?ki* **and** *ikf*: *i* < *k* − *Suc fin i* < *k* − *fin* **and** *kfn*: *k* − *fin* ≤ *n* **by** *auto*
from *xd-diff*[*OF ikf*(*2*) *kfn*] **have** *nz*: *xd* (*k* − *fin*) *i* ≠ *0* **by** *auto*
note *IH* = *Suc*(*1*)[*OF fki*]
have *id4*: [*0* ..< *?ki* − *fin*] = [*0* ..< *?ki* − *Suc fin*] @ [*?ki* − *Suc fin*]
i + (*k* − *i* − *Suc fin*) = *k* − *Suc fin*
Suc (*k* − *Suc fin*) = *k* − *fin* **using** *Suc*(*2*) *assms* ‹*fin* < *?ki*›
by (*metis Suc-diff-Suc le0 upt-Suc*) (*insert Suc*(*2*), *auto*)
from *Suc*(*2*) *assms* **have** *id5*: [*i*..<*k* − *Suc fin*] = *i* # [*Suc i* ..< *k* − *Suc fin*]

[*Suc i*..<*k* − *fin*] = [*Suc i*..<*k* − *Suc fin*] @ [*k* − *Suc fin*]
by (*force simp*: *upt-rec*) (*metis Suc-leI id4*(*3*) *ikf*(*1*) *upt-Suc*)
have *?sum 0* = *?sum fin* **by** (*rule IH*)
also have ... = *?sumi* [*0* ..< *?ki* − *Suc fin*] + *?sumsi* [*?ki* − *fin* ..< *?ki*] +
(*ei* (*?ki* − *Suc fin*) + *?mid fin*)
unfolding *id4* **by** *simp*
also have *?mid fin* = (*xij-f* (*Suc i*) (*k* − *fin*) − *xij-f i* (*k* − *Suc fin*))
∗ *pprod k* (*Suc i*) (*k* − *fin*) **unfolding** *xij-f.simps*[*of i k* − *fin*]
using *ikf nz* **by** *simp*
also have ... = *xij-f* (*Suc i*) (*k* − *fin*) ∗ *pprod k* (*Suc i*) (*k* − *fin*) −
xij-f i (*k* − *Suc fin*) ∗ *pprod k* (*Suc i*) (*k* − *fin*) **by** *algebra*
also have *xij-f* (*Suc i*) (*k* − *fin*) ∗ *pprod k* (*Suc i*) (*k* − *fin*) = *esi* (*?ki* − *Suc fin*)
unfolding *esi-def* **using** *ikf* **by** (*simp add*: *id4*)
also have *ei* (*?ki* − *Suc fin*) = *xij-f i* (*k* − *Suc fin*) ∗ *pprod k i* (*k* − *Suc fin*)

unfolding *ei-def id4* **using** *ikf* **by** (*simp add*: *ac-simps*)
finally have *?sum 0* = *?sumi* [*0* ..< *?ki* − *Suc fin*]
+ (*esi* (*?ki* − *Suc fin*) + *?sumsi* [*?ki* − *fin* ..< *?ki*])
+ (*xij-f i* (*k* − *Suc fin*) ∗ (*pprod k i* (*k* − *Suc fin*) − *pprod k* (*Suc i*) (*k* − *fin*)))
by *algebra*
also have *esi* (*?ki* − *Suc fin*) + *?sumsi* [*?ki* − *fin* ..< *?ki*]
= *?sumsi* ((*?ki* − *Suc fin*) # [*?ki* − *fin* ..< *?ki*]) **by** *simp*
also have (*?ki* − *Suc fin*) # [*?ki* − *fin* ..< *?ki*] = [*?ki* − *Suc fin* ..< *?ki*]
using *Suc*(*2*) **by** (*simp add*: *Suc-diff-Suc upt-rec*)
also have *pprod k i* (*k* − *Suc fin*) − *pprod k* (*Suc i*) (*k* − *fin*)
= (*xd k i*) ∗ *pprod k* (*Suc i*) (*k* − *Suc fin*) − (*xd k* (*k* − *Suc fin*)) ∗ *pprod k* (*Suc i*) (*k* − *Suc fin*)
unfolding *pprod-def id5* **by** *simp*
also have ... = (*xd k i* − *xd k* (*k* − *Suc fin*)) ∗ *pprod k* (*Suc i*) (*k* − *Suc fin*)
by *algebra*
also have ... = (*xd* (*k* − *Suc fin*) *i*) ∗ *pprod k* (*Suc i*) (*k* − *Suc fin*) **unfolding** *xd-def* **by** *simp*
also have *xij-f i* (*k* − *Suc fin*) ∗ ... = *?mid* (*Suc fin*) **by** *simp*
finally show *?case* **by** *simp*
qed *simp*
also have ... = (*ei 0* + *?mid* (*k* − *i* − *1*)) + *?sumsi* [*1* ..< *k* − *i*]
unfolding *fin-def* **by** (*simp add*: *id3*)
also have *ei 0* + *?mid* (*k* − *i* − *1*) = *esi 0* **unfolding** *id3*
unfolding *ei-def esi-def xij-f.simps*[*of i i*] **using** *neq assms*
by (*simp add*: *field-simps xij-f.simps pprod-def*)
also have *esi 0* + *?sumsi* [*1* ..< *k* − *i*] = *?sumsi* (*0* # [*1* ..< *k* − *i*]) **by** *simp*
also have *0* # [*1* ..< *k* − *i*] = [*0* ..< *Suc k* − *Suc i*]
using *assms* **by** (*simp add*: *upt-rec*)
also have *?sumsi* ... = *sum-list* (*map* (λ *j*. *xij-f* (*Suc i*) (*Suc i* + *j*) ∗
pprod k (*Suc i*) (*Suc i* + *j*)) [*0*..<*Suc k* − *Suc i*])
unfolding *esi-def* **using** *assms* **by** *simp*
finally show *?thesis* **.**

qed

private lemma *divided-differences*: **assumes** *kn*: $k \leq n$ **and** *ik*: $i \leq k$
shows *sum-list* (*map* (λ *j*. *xij-f i* ($i + j$) $*$ *pprod k i* ($i + j$)) [*0*..<*Suc k* $-$ *i*]) = *f k*
proof $-$
{
fix *ii*
assume $i + ii \leq k$
hence *sum-list* (*map* (λ *j*. *xij-f i* ($i + j$) $*$ *pprod k i* ($i + j$)) [*0*..<*Suc k* $-$ *i*])
= *sum-list* (*map* (λ *j*. *xij-f* ($i + ii$) ($i + ii + j$) $*$ *pprod k* ($i + ii$) ($i + ii + j$)) [*0*..<*Suc k* $-$ ($i + ii$)])
proof (*induct ii*)
case (*Suc ii*)
hence *le1*: $i + ii \leq k$ **and** *le2*: $i + ii < k$ **by** *simp-all*
show *?case* **unfolding** *Suc*(*1*)[*OF le1*] **unfolding** *divided-differences-main*[*OF kn le2*]
using *Suc*(*2*) **by** *simp*
qed *simp*
} **note** *main* = *this*
have *ik*: $i + (k - i) \leq k$ **and** *id*: $i + (k - i) = k$ **using** *ik* **by** *simp-all*
show *?thesis* **unfolding** *main*[*OF ik*] **unfolding** *id*
by (*simp add*: *xij-f.simps pprod-def*)
qed

lemma *newton-poly-sound*: **assumes** $k \leq n$
shows *poly* (*newton-poly n*) (*x k*) = *f k*
proof $-$
have *poly* (*newton-poly n*) (*x k*) =
sum-list (*map* (λ *j*. *xij-f 0* ($0 + j$) $*$ *pprod k 0* ($0 + j$)) [*0*..<*Suc k* $-$ *0*])
unfolding *poly-newton-poly-xj*[*OF assms*] *c-def poly-N-xi* **by** *simp*
also have ... = *f k*
by (*rule divided-differences*[*OF assms*], *simp*)
finally show *?thesis* **by** *simp*
qed
end

lemma *newton-poly-degree*: *degree* (*newton-poly n*) $\leq n$
proof $-$
{
fix *i*
have $i \leq n \Longrightarrow$ *degree* (*b i n*) $\leq n - i$
proof (*induct i n rule*: *b.induct*)
case (*1 i n*)
note *b* = *b.simps*[*of i n*]
show *?case*
proof (*cases* $n \leq i$)
case *True*
thus *?thesis* **unfolding** *b* **by** *auto*

```
      next
        case False
          have degree (b i n) = degree (b (Suc i) n * X i + [:c i:]) using False
unfolding b by simp
        also have ... ≤ max (degree (b (Suc i) n * X i)) (degree [:c i:])
          by (rule degree-add-le-max)
        also have ... = degree (b (Suc i) n * X i) by simp
        also have ... ≤ degree (b (Suc i) n) + degree (X i)
          by (rule degree-mult-le)
        also have ... ≤ n − Suc i + degree (X i)
          using 1(1)[OF False] 1(2) False add-le-mono1 not-less-eq-eq by blast
        also have ... = n − Suc i + 1 unfolding X-def by simp
        also have ... = n − i using 1(2) False by auto
        finally show ?thesis .
      qed
    qed
  }
  from this[of 0] show ?thesis unfolding newton-poly-def by simp
qed

context
  fixes n
  assumes xs: length xs = n
    and fs: length fs = n
begin
lemma newton-coefficients-main:
   k < n =⇒ newton-coefficients-main (rev (map f [0..<Suc k])) (rev (map x
[0..<Suc k]))
    = rev (map (λ i. map (λ j. xij-f j i) [0..<Suc i]) [0..<Suc k])
proof (induct k)
  case 0
  show ?case
    by (simp add: xij-f.simps)
next
  case (Suc k)
  hence k < n by auto
  note IH = Suc(1)[OF this]
  have id: ⋀ f. rev (map f [0..<Suc (Suc k)]) = f (Suc k) # f k # rev (map f
[0..< k])
   and id2: ⋀ f. f k # rev (map f [0..<k]) = rev (map f [0..< Suc k]) by simp-all
  show ?case unfolding id newton-coefficients-main.simps Let-def
    unfolding id2 IH
    unfolding list.simps id2[symmetric]
  proof (rule conjI, goal-cases)
    case 1
    have xs: xs = map x [0 ..< n] using xs unfolding x-def[abs-def]
      by (intro nth-equalityI, auto)
    define nn where nn = (0 :: nat)
    define m where m = Suc k − nn
```

```
    have prems: m = Suc k − nn nn < Suc (Suc k) unfolding m-def nn-def by
auto
    have ?case = (divided-differences-impl (map ((λj. xij-f j k)) [nn..< Suc k]) (f
(Suc k)) (x (Suc k)) (map x [nn ..< n]) =
      map ((λj. xij-f j (Suc k))) [nn..<Suc (Suc k)])
      unfolding nn-def xs[symmetric] by simp
    also have . . . using prems
    proof (induct m arbitrary: nn)
      case 0
      hence nn: nn = Suc k by auto
      show ?case unfolding nn by (simp add: xij-f.simps)
    next
      case (Suc m)
      with ‹Suc k < n› have nn < n and le: nn < Suc k by auto
      with Suc(2−) have id:
        [nn..<Suc k] = nn # [Suc nn..< Suc k]
        [nn..<n] = nn # [Suc nn..< n]
      and id2: [nn..<Suc (Suc k)] = nn # [Suc nn..<Suc (Suc k)]
        [Suc nn..<Suc (Suc k)] = Suc nn # [Suc (Suc nn)..<Suc (Suc k)]
        by (auto simp: upt-rec)
      from Suc(2−) have m = Suc k − Suc nn Suc nn < Suc (Suc k) by auto
      note IH = Suc(1)[OF this]
      show ?case unfolding id list.simps divided-differences-impl.simps IH Let-def
        unfolding id2 list.simps
        using le
        by (simp add: xij-f.simps[of nn Suc k] xd-def)
    qed
    finally show ?case by simp
  qed simp
qed

lemma newton-coefficients: newton-coefficients = rev (map c [0 ..< n])
proof (cases n)
  case 0
  hence xs: xs = [] fs = [] using xs fs by auto
  show ?thesis unfolding newton-coefficients-def 0
    using newton-coefficients-main.simps
    unfolding xs by simp
next
  case (Suc nn)
  hence sn: Suc nn = n and nn: nn < n by auto
  from fs have fs: map f [0..<Suc nn] = fs unfolding sn
    by (intro nth-equalityI, auto simp: f-def)
  from xs have xs: map x [0..<Suc nn] = xs unfolding sn
    by (intro nth-equalityI, auto simp: x-def)
  show ?thesis
    unfolding newton-coefficients-def
      newton-coefficients-main[OF nn, unfolded fs xs]
    unfolding sn rev-map[symmetric] map-map o-def
```

```
   by (rule arg-cong[of - - rev], subst upt-rec, intro nth-equalityI, auto simp: c-def)
qed

lemma newton-poly-impl: assumes n = Suc nn
  shows newton-poly-impl = newton-poly nn
proof −
  define i where i = (0 :: nat)
  have xs: map x [0..<n] = xs using xs
    by (intro nth-equalityI, auto simp: x-def)
  have i ≤ nn unfolding i-def by simp
  hence horner-composition (map c [i..<Suc nn]) (map x [i..<Suc nn]) = b i nn
  proof (induct i nn rule: b.induct)
    case (1 i n)
    show ?case
    proof (cases n ≤ i)
      case True
      with 1(2) have i: i = n by simp
      show ?thesis unfolding i b.simps[of n n] by simp
    next
      case False
      hence Suc i ≤ n by simp
      note IH = 1(1)[OF False this]
     have bi: b i n = b (Suc i) n ∗ X i + [:c i:] using False by (simp add: b.simps)

      from False have id: [i ..< Suc n] = i # [Suc i ..< Suc n] by (simp add:
upt-rec)
      from False have id2: [Suc i ..< Suc n] = Suc i # [Suc (Suc i) ..< Suc n]
by (simp add: upt-rec)
      show ?thesis unfolding id bi list.simps horner-composition.simps id2
        unfolding IH[unfolded id2 list.simps] by (simp add: X-def)
    qed
  qed
  thus ?thesis
   unfolding newton-poly-impl-def newton-coefficients rev-rev-ident newton-poly-def
i-def
      assms[symmetric] xs .
qed
end
end

context
  fixes xs fs :: int list
begin

fun divided-differences-impl-int :: int list ⇒ int ⇒ int ⇒ int list ⇒ int list option
where
  divided-differences-impl-int (xi-j1 # x-j1s) fj xj (xi # xis) = (
     case divided-differences-impl-int x-j1s fj xj xis of None ⇒ None
   | Some x-js ⇒ let (new,m) = divmod-int (hd x-js − xi-j1) (xj − xi)
```

in if m = 0 then Some (new # x-js) else None)
| *divided-differences-impl-int* [] *fj xj xis* = *Some* [*fj*]

fun *newton-coefficients-main-int* :: *int list* ⇒ *int list* ⇒ *int list list option* **where**
newton-coefficients-main-int [*fj*] *xjs* = *Some* [[*fj*]]
| *newton-coefficients-main-int* (*fj* # *fjs*) (*xj* # *xjs*) = (*do* {
rec ← *newton-coefficients-main-int fjs xjs*;
let row = *hd rec*;
new-row ← *divided-differences-impl-int row fj xj xs*;
Some (*new-row* # *rec*)})
| *newton-coefficients-main-int* - - = *Some* []

definition *newton-coefficients-int* :: *int list option* **where**
newton-coefficients-int = *map-option* (*map hd*) (*newton-coefficients-main-int* (*rev fs*) (*rev xs*))

lemma *divided-differences-impl-int-Some*:
length gs ≤ *length ys*
⟹ *divided-differences-impl-int gs g x ys* = *Some res*
⟹ *divided-differences-impl* (*map rat-of-int gs*) (*rat-of-int g*) (*rat-of-int x*) (*map rat-of-int ys*) = *map rat-of-int res*
∧ *length res* = *Suc* (*length gs*)
proof (*induct gs g x ys arbitrary*: *res rule*: *divided-differences-impl-int.induct*)
case (*1 xi-j1 x-j1s fj xj xi xis*)
note *some* = *1*(*3*)
from *1*(*2*) **have** *len*: *length x-j1s* ≤ *length xis* **by** *auto*
from *some* **obtain** *x-js* **where** *rec*: *divided-differences-impl-int x-j1s fj xj xis* = *Some x-js*
by (*auto split*: *option.splits*)
note *IH* = *1*(*1*)[*OF len rec*]
have *id*: *hd* (*map rat-of-int x-js*) = *rat-of-int* (*hd x-js*) **using** *IH* **by** (*cases x-js*, *auto*)
from *some*[*simplified*, *unfolded rec divmod-int-def*] **have** *dvd*: (*xj* − *xi*) *dvd* (*hd x-js* − *xi-j1*)
and *res*: *res* = (*hd x-js* − *xi-j1*) *div* (*xj* − *xi*) # *x-js* **by** (*auto split*: *if-splits*)
from *dvd* **obtain** *k* **where** ‹*hd x-js* − *xi-j1* = (*xj* − *xi*) ∗ *k*› **..**
then have ‹*hd x-js* = (*xj* − *xi*) ∗ *k* + *xi-j1*›
by *simp*
then have *rat-of-int* ((*hd x-js* − *xi-j1*) *div* (*xj* − *xi*)) = *rat-of-int* (*hd x-js* − *xi-j1*) / *rat-of-int* (*xj* − *xi*)
by *simp*
hence (*rat-of-int* (*hd x-js*) − *rat-of-int xi-j1*) / (*rat-of-int xj* − *rat-of-int xi*) =
rat-of-int ((*hd x-js* − *xi-j1*) *div* (*xj* − *xi*))
by *simp*
thus *?case* **by** (*simp add*: *IH Let-def res id*)
next
case (*2 fj xj xis res*)
hence *res*: *res* = [*fj*] **by** *simp*
thus *?case* **by** *simp*

```
qed simp

lemma div-Ints-mod-0: assumes rat-of-int a / rat-of-int b ∈ ℤ b ≠ 0
  shows a mod b = 0
proof −
  define c where c = int-of-rat (rat-of-int a / rat-of-int b)
  have rat-of-int a / rat-of-int b = rat-of-int c unfolding c-def using assms(1)
by simp
  hence rat-of-int a = rat-of-int b ∗ rat-of-int c using assms(2)
    by (metis divide-cancel-right nonzero-mult-div-cancel-left of-int-eq-0-iff)
  hence a: a = b ∗ c by (simp add: of-int-hom.injectivity)
  show a mod b = 0 unfolding a by simp
qed

lemma divided-differences-impl-int-None:
  length gs ≤ length ys
  =⇒ divided-differences-impl-int gs g x ys = None
  =⇒ x ∉ set (take (length gs) ys)
  =⇒ hd (divided-differences-impl (map rat-of-int gs) (rat-of-int g) (rat-of-int x)
(map rat-of-int ys)) ∉ ℤ
proof (induct gs g x ys rule: divided-differences-impl-int.induct)
  case (1 xi-j1 x-j1s fj xj xi xis)
  note none = 1(3)
  from 1(2,4) have len: length x-j1s ≤ length xis and xj: xj ∉ set (take (length
x-j1s) xis) and xji: xj ≠ xi by auto
  define d where d = divided-differences-impl (map rat-of-int x-j1s) (rat-of-int fj)
(rat-of-int xj) (map rat-of-int xis)
  note IH = 1(1)[OF len - xj]
  show ?case
  proof (cases divided-differences-impl-int x-j1s fj xj xis)
    case None
    from IH[OF None] have d: hd d ∉ ℤ unfolding d-def by auto
    {
      let ?x = (hd d − rat-of-int xi-j1) / (rat-of-int xj − rat-of-int xi)
      assume ?x ∈ ℤ
      hence ?x ∗ (of-int (xj − xi)) + rat-of-int xi-j1 ∈ ℤ
        using Ints-mult Ints-add Ints-of-int by blast
      also have ?x ∗ (of-int (xj − xi)) = hd d − rat-of-int xi-j1 using xji by auto
      also have . . . + rat-of-int xi-j1 = hd d by simp
      finally have False using d by simp
    }
    thus ?thesis
      by (auto simp: Let-def d-def[symmetric])
  next
    case (Some res)
    from divided-differences-impl-int-Some[OF len Some]
    have id: divided-differences-impl (map rat-of-int x-j1s) (rat-of-int fj) (rat-of-int
xj) (map rat-of-int xis) =
      map rat-of-int res and res: res ≠ [] by auto
```

have *hd*: *hd* (*map rat-of-int res*) = *of-int* (*hd res*) **using** *res* **by** (*cases res*, *auto*)
  define *a* **where** *a* = (*hd res* − *xi-j1*)
  define *b* **where** *b* = *xj* − *xi*
  from *none*[*simplified*, *unfolded Some divmod-int-def*]
  have *mod*: *a mod b* ≠ *0*
    by (*auto split*: *if-splits simp*: *a-def b-def*)
  {
    assume (*rat-of-int* (*hd res*) − *rat-of-int xi-j1*) / (*rat-of-int xj* − *rat-of-int xi*) ∈ ℤ
    hence *rat-of-int a* / *rat-of-int b* ∈ ℤ **unfolding** *a-def b-def* **by** *simp*
    moreover have *b* ≠ *0* **using** *xji* **unfolding** *b-def* **by** *simp*
    ultimately have *False* **using** *mod div-Ints-mod-0* **by** *auto*
  }
  thus *?thesis*
    by (*auto simp*: *id Let-def hd*)
qed
qed *auto*

lemma *newton-coefficients-main-int-Some*:
  length gs = *length ys* ⟹ *length ys* ≤ *length xs*
  ⟹ *newton-coefficients-main-int gs ys* = *Some res*
  ⟹ *newton-coefficients-main* (*map rat-of-int xs*) (*map rat-of-int gs*) (*map rat-of-int ys*) = *map* (*map rat-of-int*) *res*
    ∧ (∀ *x* ∈ *set res*. *x* ≠ [] ∧ *length x* ≤ *length ys*) ∧ *length res* = *length gs*
proof (*induct gs ys arbitrary*: *res rule*: *newton-coefficients-main-int.induct*)
  case (*2 fv v va xj xjs res*)
  from *2*(*2*,*3*) **have** *len*: *length* (*v* # *va*) = *length xjs length xjs* ≤ *length xs* **by** *auto*
  note *some* = *2*(*4*)
  let *?n* = *newton-coefficients-main-int* (*v* # *va*) *xjs*
  let *?ri* = *rat-of-int*
  let *?mri* = *map ?ri*
  from *some* **obtain** *rec* **where** *n*: *?n* = *Some rec*
    by (*cases ?n*, *auto*)
  note *some* = *some*[*simplified*, *unfolded n*]
  let *?d* = *divided-differences-impl-int* (*hd rec*) *fv xj xs*
  from *some* **obtain** *dd* **where** *d*: *?d* = *Some dd* **and** *res*: *res* = *dd* # *rec*
    by (*cases ?d*, *auto*)
  note *IH* = *2*(*1*)[*OF len n*]
  from *IH* **have** *lenn*: *length* (*hd rec*) ≤ *length xjs* **by** (*cases rec*, *auto*)
  with *len* **have** *length* (*hd rec*) ≤ *length xs* **by** *auto*
  note *dd* = *divided-differences-impl-int-Some*[*OF this d*]
  have *hd*: *hd* (*map ?mri rec*) = *?mri* (*hd rec*) **using** *IH* **by** (*cases rec*, *auto*)
  show *?case* **unfolding** *newton-coefficients-main.simps list.simps*
    IH[*THEN conjunct1*, *unfolded list.simps*] *Let-def hd*
    dd[*THEN conjunct1*] *res*
  proof (*intro conjI*)
    show *length* (*dd* # *rec*) = *length* (*fv* # *v* # *va*) **using** *len*

IH[*THEN conjunct2*] *dd*[*THEN conjunct2*] **by** *auto*
show ∀ *x*∈*insert dd* (*set rec*). *x* ≠ [] ∧ *length x* ≤ *length* (*xj* # *xjs*)
using *len IH*[*THEN conjunct2*] *dd*[*THEN conjunct2*] *lenn* **by** *auto*
qed *auto*
qed *auto*

lemma *newton-coefficients-main-int-None*: **assumes** *dist*: *distinct xs*
shows *length gs* = *length ys* ⟹ *length ys* ≤ *length xs*
⟹ *newton-coefficients-main-int gs ys* = *None*
⟹ *ys* = *drop* (*length xs* − *length ys*) (*rev xs*)
⟹ ∃ *row* ∈ *set* (*newton-coefficients-main* (*map rat-of-int xs*) (*map rat-of-int gs*) (*map rat-of-int ys*)). *hd row* ∉ ℤ
proof (*induct gs ys rule*: *newton-coefficients-main-int.induct*)
case (*2 fv v va xj xjs*)
from *2*(*2*,*3*) **have** *len*: *length* (*v* # *va*) = *length xjs length xjs* ≤ *length xs* **by** *auto*
from *arg-cong*[*OF 2*(*5*), *of tl*] *2*(*3*)
have *xjs*: *xjs* = *drop* (*length xs* − *length xjs*) (*rev xs*)
by (*metis 2*(*5*) *butlast-snoc butlast-take length-drop rev.simps*(*2*) *rev-drop rev-rev-ident rev-take*)
note *none* = *2*(*4*)
let *?n* = *newton-coefficients-main-int* (*v* # *va*) *xjs*
let *?n′* = *newton-coefficients-main* (*map rat-of-int xs*) (*map rat-of-int* (*v* # *va*)) (*map rat-of-int xjs*)
let *?ri* = *rat-of-int*
let *?mri* = *map ?ri*
show *?case*
proof (*cases ?n*)
case *None*
from *2*(*1*)[*OF len None xjs*] **obtain** *row* **where**
row: *row*∈*set ?n′* **and** *hd row* ∉ ℤ **by** *auto*
thus *?thesis* **by** (*intro bexI*[*of - row*], *auto simp*: *Let-def*)
next
case (*Some rec*)
note *some* = *newton-coefficients-main-int-Some*[*OF len this*]
hence *len′*: *length* (*hd rec*) ≤ *length xjs* **by** (*cases rec*, *auto*)
hence *lenn*: *length* (*hd rec*) ≤ *length xs* **using** *len* **by** *auto*
have *hd*: *hd* (*map ?mri rec*) = *?mri* (*hd rec*) **using** *some* **by** (*cases rec*, *auto*)
let *?d* = *divided-differences-impl-int* (*hd rec*) *fv xj xs*
from *none*[*simplified*, *unfolded Some*]
have *none*: *?d* = *None* **by** (*cases ?d*, *auto*)
have *xj* ∉ *set* (*take* (*length* (*hd rec*)) *xs*)
proof
assume *xj* ∈ *set* (*take* (*length* (*hd rec*)) *xs*)
then obtain *i* **where** *i* < *length* (*hd rec*) **and** *xj*: *xj* = *xs* ! *i*
unfolding *in-set-conv-nth* **by** *auto*
with *len′* **have** *i*: *i* < *length xjs* **by** *simp*
have *Suc* (*length xjs*) ≤ *length xs* **using** *2*(*3*) **by** *auto*
with *i* **have** *i0*: *i* ≠ *0*

by (*metis* *2*(*5*) *Suc-diff-Suc Suc-le-lessD diff-less dist distinct-conv-nth hd-drop-conv-nth length-Cons length-drop length-greater-0-conv length-rev less-le-trans list.sel*(*1*) *list.simps*(*3*) *nat-neq-iff rev-nth xj xjs*)
have *xj* ∈ *set xjs*
by (*subst xjs*, *unfold xj in-set-conv-nth*, *rule exI*[*of - length xjs* − *Suc i*], *insert i 2*(*3*) *i0*, *auto simp*: *rev-nth*)
hence *ndist*: ¬ *distinct* (*xj* # *xjs*) **by** *auto*
from *dist* **have** *distinct* (*rev xs*) **by** *simp*
from *distinct-drop*[*OF this*] **have** *distinct* (*xj* # *xjs*) **using** *2*(*5*) **by** *metis*
with *ndist*
show *False* **..**
qed
note *dd* = *divided-differences-impl-int-None*[*OF lenn none this*]
show *?thesis*
by (*rule bexI*, *rule dd*, *insert some hd*, *auto*)
qed
qed *auto*

lemma *newton-coefficients-int*: **assumes** *dist*: *distinct xs*
and *len*: *length xs* = *length fs*
shows *newton-coefficients-int* = (*let cs* = *newton-coefficients* (*map rat-of-int xs*) (*map of-int fs*)
in if set cs ⊆ ℤ *then Some* (*map int-of-rat cs*) *else None*)
proof −
from *len* **have** *len*: *length* (*rev fs*) = *length* (*rev xs*) *length* (*rev xs*) ≤ *length xs* **by** *auto*
show *?thesis*
proof (*cases newton-coefficients-main-int* (*rev fs*) (*rev xs*))
case (*Some res*)
have *rev*: ⋀ *xs*. *map rat-of-int* (*rev xs*) = *rev* (*map of-int xs*) **unfolding** *rev-map* **..**
note *n* = *newton-coefficients-main-int-Some*[*OF len Some*, *unfolded rev*]
{
fix *row*
assume *row* ∈ *set res*
with *n* **have** *row* ≠ [] **by** *auto*
hence *id*: *hd* (*map rat-of-int row*) = *rat-of-int* (*hd row*) **by** (*cases row*, *auto*)
also have . . . ∈ ℤ **by** *auto*
finally have *int*: *hd* (*map rat-of-int row*) ∈ ℤ **by** *auto*
have *hd row* = *int-of-rat* (*hd* (*map rat-of-int row*)) **unfolding** *id* **by** *simp*
note *this int*
}
thus *?thesis* **unfolding** *newton-coefficients-int-def Some newton-coefficients-def n*[*THEN conjunct1*] *Let-def option.simps*
by (*auto simp*: *o-def*)
next

case *None*
have *rev xs* = *drop* (*length xs* − *length* (*rev xs*)) (*rev xs*) **by** *simp*
from *newton-coefficients-main-int-None*[*OF dist len None this*]
show *?thesis* **unfolding** *newton-coefficients-int-def newton-coefficients-def None*
by (*auto simp*: *Let-def rev-map*)
qed
qed

definition *newton-poly-impl-int* :: *int poly option* **where**
newton-poly-impl-int ≡ *case newton-coefficients-int of None* ⇒ *None*
| *Some nc* ⇒ *Some* (*horner-composition* (*rev nc*) *xs*)

lemma *newton-poly-impl-int*: **assumes** *len*: *length xs* = *length fs*
and *dist*: *distinct xs*
shows *newton-poly-impl-int* = (*let p* = *newton-poly-impl* (*map rat-of-int xs*) (*map of-int fs*)
in if set (*coeffs p*) ⊆ ℤ *then Some* (*map-poly int-of-rat p*) *else None*)
proof −
let *?ir* = *int-of-rat*
let *?ri* = *rat-of-int*
let *?mir* = *map ?ir*
let *?mri* = *map ?ri*
let *?nc* = *newton-coefficients* (*?mri xs*) (*?mri fs*)
have *id*: *newton-poly-impl-int* = (*if set ?nc* ⊆ ℤ
then Some (*horner-composition* (*rev* (*?mir ?nc*)) *xs*) *else None*)
unfolding *newton-poly-impl-int-def newton-coefficients-int*[*OF dist len*] *Let-def*
by *simp*
have *len*: *length* (*rev ?nc*) ≤ *Suc* (*length xs*)
unfolding *length-rev*
by (*subst newton-coefficients*[*OF refl*], *insert len*, *auto*)
show *?thesis* **unfolding** *id*
unfolding *newton-poly-impl-def*
unfolding *Let-def set-rev rev-map horner-coeffs-ints*[*OF len*]
proof (*rule if-cong*[*OF refl - refl*], *rule arg-cong*[*of - - Some*])
define *cs* **where** *cs* = *rev ?nc*
define *ics* **where** *ics* = *map ?ir cs*
assume *set ?nc* ⊆ ℤ
hence *set cs* ⊆ ℤ **unfolding** *cs-def* **by** *auto*
hence *ics*: *cs* = *?mri ics* **unfolding** *ics-def map-map o-def*
by (*simp add*: *map-idI subset-code*(*1*))
have *id*: *horner-composition* (*rev ?nc*) (*?mri xs*) = *map-poly ?ri* (*horner-composition ics xs*)
unfolding *cs-def*[*symmetric*] *ics*
by (*rule of-int-poly-hom.horner-composition-hom*)
show *horner-composition* (*?mir* (*rev ?nc*)) *xs*
= *map-poly ?ir* (*horner-composition* (*rev ?nc*) (*?mri xs*))
unfolding *id* **unfolding** *cs-def*[*symmetric*] *ics-def*[*symmetric*]
by (*subst map-poly-map-poly*, *auto simp*: *o-def map-poly-idI*)
qed

qed
end

definition *newton-interpolation-poly* :: ('*a* :: *field* × '*a*)*list* ⇒ '*a poly* **where**
newton-interpolation-poly x-fs = (*let*
xs = *map fst x-fs*; *fs* = *map snd x-fs in*
newton-poly-impl xs fs)

definition *newton-interpolation-poly-int* :: (*int* × *int*)*list* ⇒ *int poly option* **where**
newton-interpolation-poly-int x-fs = (*let*
xs = *map fst x-fs*; *fs* = *map snd x-fs in*
newton-poly-impl-int xs fs)

lemma *newton-interpolation-poly*: **assumes** *dist*: *distinct* (*map fst xs-ys*)
and *p*: *p* = *newton-interpolation-poly xs-ys*
and *xy*: (*x*,*y*) ∈ *set xs-ys*
shows *poly p x* = *y*
proof (*cases length xs-ys*)
case *0*
thus *?thesis* **using** *xy* **by** (*cases xs-ys*, *auto*)
next
case (*Suc nn*)
let *?xs* = *map fst xs-ys* **let** *?fs* = *map snd xs-ys* **let** *?n* = *Suc nn*
from *xy*[*unfolded set-conv-nth*] **obtain** *i* **where** *xy*: *i* ≤ *nn x* = *?xs* ! *i y* = *?fs* ! *i*
using *Suc*
by (*metis* (*no-types*, *lifting*) *fst-conv in-set-conv-nth less-Suc-eq-le nth-map snd-conv xy*)
have *id*: *newton-interpolation-poly xs-ys* = *newton-poly ?xs ?fs nn*
unfolding *newton-interpolation-poly-def Let-def*
by (*rule newton-poly-impl*[*OF - - Suc*], *auto*)
show *?thesis*
unfolding *p id*
proof (*rule newton-poly-sound*[*of nn ?xs - ?fs*, *unfolded*
Newton-Interpolation.x-def Newton-Interpolation.f-def, *OF - xy*(*1*), *folded xy*(*2*−)])
fix *i j*
show *i* < *j* ⟹ *j* ≤ *nn* ⟹ *?xs* ! *i* ≠ *?xs* ! *j* **using** *dist Suc nth-eq-iff-index-eq* **by** *fastforce*
qed
qed

lemma *degree-newton-interpolation-poly*:
shows *degree* (*newton-interpolation-poly xs-ys*) ≤ *length xs-ys* − *1*
proof (*cases length xs-ys*)
case *0*
hence *id*: *xs-ys* = [] **by** (*cases xs-ys*, *auto*)
show *?thesis* **unfolding**
id newton-interpolation-poly-def Let-def list.simps newton-poly-impl-def

Newton-Interpolation.newton-coefficients-def
by *simp*
next
case (*Suc nn*)
let *?xs* = *map fst xs-ys* **let** *?fs* = *map snd xs-ys* **let** *?n* = *Suc nn*
have *id*: *newton-interpolation-poly xs-ys* = *newton-poly ?xs ?fs nn*
unfolding *newton-interpolation-poly-def Let-def*
by (*rule newton-poly-impl*[*OF - - Suc*], *auto*)
show *?thesis* **unfolding** *id* **using** *newton-poly-degree*[*of ?xs ?fs nn*] *Suc* **by** *simp*
qed

For *newton-interpolation-poly-int* at this point we just prove that it is equivalent to perfom an interpolation on the rational numbers, and then check whether all resulting coefficients are integers. That this corresponds to a sound and complete interpolation algorithm on the integers is proven in the theory Polynomial-Interpolation, cf. lemmas newton-interpolation-poly-int-Some/None.

lemma *newton-interpolation-poly-int*: **assumes** *dist*: *distinct* (*map fst xs-ys*)
shows *newton-interpolation-poly-int xs-ys* = (*let*
rxs-ys = *map* (λ (*x*,*y*). (*rat-of-int x*, *rat-of-int y*)) *xs-ys*;
rp = *newton-interpolation-poly rxs-ys*
in if (∀ *x* ∈ *set* (*coeffs rp*). *is-int-rat x*) *then*
Some (*map-poly int-of-rat rp*) *else None*)
proof −
have *id1*: *map fst* (*map* (λ(*x*, *y*). (*rat-of-int x*, *rat-of-int y*)) *xs-ys*) = *map rat-of-int* (*map fst xs-ys*)
by (*induct xs-ys*, *auto*)
have *id2*: *map snd* (*map* (λ(*x*, *y*). (*rat-of-int x*, *rat-of-int y*)) *xs-ys*) = *map rat-of-int* (*map snd xs-ys*)
by (*induct xs-ys*, *auto*)
have *id3*: *length* (*map fst xs-ys*) = *length* (*map snd xs-ys*) **by** *auto*
show *?thesis*
unfolding *newton-interpolation-poly-def newton-interpolation-poly-int-def Let-def newton-poly-impl-int*[*OF id3 dist*]
unfolding *id1 id2*
by (*rule sym*, *rule if-cong*, *auto simp*: *is-int-rat*[*abs-def*])
qed

hide-const
Newton-Interpolation.x
Newton-Interpolation.f
end

10 Lagrange Interpolation

We formalized the Lagrange interpolation, i.e., a method to interpolate a polynomial p from a list of points $(x_1, p(x_1)), (x_2, p(x_2)), \ldots$. The interpolation algorithm is proven to be sound and complete.

```
theory Lagrange-Interpolation
imports
  Missing-Polynomial
begin

definition lagrange-basis-poly :: 'a :: field list ⇒ 'a ⇒ 'a poly where
  lagrange-basis-poly xs xj ≡ let ys = filter (λ x. x ≠ xj) xs
    in prod-list (map (λ xi. smult (inverse (xj − xi)) [: − xi, 1 :]) ys)

definition lagrange-interpolation-poly :: ('a :: field × 'a)list ⇒ 'a poly where
  lagrange-interpolation-poly xs-ys ≡ let
    xs = map fst xs-ys
    in sum-list (map (λ (xj,yj). smult yj (lagrange-basis-poly xs xj)) xs-ys)

lemma [code]:
  lagrange-basis-poly xs xj = (let ys = filter (λ x. x ≠ xj) xs
    in prod-list (map (λ xi. let ii = inverse (xj − xi) in [: − ii ∗ xi, ii :]) ys))
  unfolding lagrange-basis-poly-def Let-def by simp

lemma degree-lagrange-basis-poly: degree (lagrange-basis-poly xs xj) ≤ length (filter
(λ x. x ≠ xj) xs)
  unfolding lagrange-basis-poly-def Let-def
  by (rule order.trans[OF degree-prod-list-le], rule order-trans[OF sum-list-mono[of
- - λ -. 1]],
  auto simp: o-def, induct xs, auto)

lemma degree-lagrange-interpolation-poly:
  shows degree (lagrange-interpolation-poly xs-ys) ≤ length xs-ys − 1
proof −
  {
    fix a b
    assume ab: (a,b) ∈ set xs-ys
    let ?xs = filter (λx. x≠a) (map fst xs-ys)
    from ab have a ∈ set (map fst xs-ys) by force
    hence Suc (length ?xs) ≤ length xs-ys
      by (induct xs-ys, auto)
    hence length ?xs ≤ length xs-ys − 1 by auto
  } note main = this
  show ?thesis
    unfolding lagrange-interpolation-poly-def Let-def
   by (rule degree-sum-list-le, auto, rule order-trans[OF degree-lagrange-basis-poly],
insert main, auto)
qed

lemma lagrange-basis-poly-1:
  poly (lagrange-basis-poly (map fst xs-ys) x) x = 1
  unfolding lagrange-basis-poly-def Let-def poly-prod-list
  by (rule prod-list-neutral, auto)
```

(*metis field-class.field-inverse mult.commute right-diff-distrib right-minus-eq*)

lemma *lagrange-basis-poly-0*: **assumes** $x' \in$ *set* (*map fst xs-ys*) **and** $x' \neq x$
shows *poly* (*lagrange-basis-poly* (*map fst xs-ys*) *x*) $x' = 0$
proof −
let *?f* = λ*xi*. *smult* (*inverse* (*x* − *xi*)) [:− *xi*, *1*:]
let *?xs* = *filter* (λ*c*. *c*≠*x*) (*map fst xs-ys*)
have *mem*: *?f* x' ∈ *set* (*map ?f ?xs*) **using** *assms* **by** *auto*
show *?thesis*
unfolding *lagrange-basis-poly-def Let-def poly-prod-list prod-list-map-remove1* [*OF mem*]
by *simp*
qed

lemma *lagrange-interpolation-poly*: **assumes** *dist*: *distinct* (*map fst xs-ys*)
and *p*: *p* = *lagrange-interpolation-poly xs-ys*
shows ⋀ *x y*. (*x*,*y*) ∈ *set xs-ys* ⟹ *poly p x* = *y*
proof −
let *?xs* = *map fst xs-ys*
{
fix *x y*
assume *xy*: (*x*,*y*) ∈ *set xs-ys*
show *poly p x* = *y* **unfolding** *p lagrange-interpolation-poly-def Let-def poly-sum-list map-map o-def*
proof (*subst sum-list-map-remove1* [*OF xy*], *unfold split poly-smult lagrange-basis-poly-1*, *subst sum-list-neutral*)
fix *v*
assume *v* ∈ *set* (*map* (λ*xa*. *poly* (*case xa of* (*xj*, *yj*) ⇒ *smult yj* (*lagrange-basis-poly ?xs xj*))
x)
(*remove1* (*x*, *y*) *xs-ys*)) (**is** - ∈ *set* (*map ?f ?xy*))
then obtain xy' **where** *mem*: xy' ∈ *set ?xy* **and** *v*: *v* = *?f* xy' **by** *auto*
obtain x' y' **where** xy': $xy' = (x', y')$ **by** *force*
from *v*[*unfolded this split*] **have** *v*: *v* = *poly* (*smult* y' (*lagrange-basis-poly ?xs* x')) *x* **.**
have *neq*: $x' \neq x$
proof
assume $x' = x$
with *mem*[*unfolded* xy'] **have** *mem*: (*x*,y') ∈ *set* (*remove1* (*x*,*y*) *xs-ys*) **by** *auto*
hence mem': (*x*,y') ∈ *set xs-ys* **by** (*meson notin-set-remove1*)
from *dist*[*unfolded distinct-map*] **have** *inj*: *inj-on fst* (*set xs-ys*) **by** *auto*
with mem' *xy* **have** y': $y' = y$ **unfolding** *inj-on-def* **by** *force*
from *dist* **have** *distinct xs-ys* **using** *distinct-map* **by** *blast*
hence (*x*,*y*) ∉ *set* (*remove1* (*x*,*y*) *xs-ys*) **by** *simp*
with *mem*[*unfolded* y']
show *False* **by** *auto*
qed
have *poly* (*lagrange-basis-poly ?xs* x') *x* = *0*

```
      by (rule lagrange-basis-poly-0, insert xy mem[unfolded xy'] dist neq, force+)
    thus v = 0 unfolding v by simp
  qed simp
} note sound = this
qed

end
```

11 Neville Aitken Interpolation

We prove soundness of Neville-Aitken's polynomial interpolation algorithm using the recursive formula directly. We further provide an implementation which avoids the exponential branching in the recursion.

```
theory Neville-Aitken-Interpolation
imports
  HOL−Computational-Algebra.Polynomial
begin

context
  fixes x :: nat ⇒ 'a :: field
  and f :: nat ⇒ 'a
begin

private definition X :: nat ⇒ 'a poly where [code-unfold]: X i = [:−x i, 1:]

function neville-aitken-main :: nat ⇒ nat ⇒ 'a poly where
  neville-aitken-main i j = (if i < j then
      (smult (inverse (x j − x i)) (X i ∗ neville-aitken-main (i + 1) j −
      X j ∗ neville-aitken-main i (j − 1)))
    else [:f i:])
  by pat-completeness auto

termination by (relation measure (λ (i,j). j − i), auto)

definition neville-aitken :: nat ⇒ 'a poly where
  neville-aitken = neville-aitken-main 0

declare neville-aitken-main.simps[simp del]

lemma neville-aitken-main: assumes dist: ⋀ i j. i < j =⇒ j ≤ n =⇒ x i ≠ x j
  shows i ≤ k =⇒ k ≤ j =⇒ j ≤ n =⇒ poly (neville-aitken-main i j) (x k) = (f
k)
proof (induct i j arbitrary: k rule: neville-aitken-main.induct)
  case (1 i j k)
  note neville-aitken-main.simps[of i j, simp]
  show ?case
  proof (cases i < j)
```

```
    case False
    with 1(3−) have k = i by auto
    with False show ?thesis by auto
  next
    case True note ij = this
    from dist[OF True 1(5)] have diff: x i ≠ x j by auto
    from True have id: neville-aitken-main i j =
      (smult (inverse (x j − x i)) (X i ∗ neville-aitken-main (i + 1) j − X j
        ∗ neville-aitken-main i (j − 1))) by simp
    note IH = 1(1−2)[OF True]
    show ?thesis
    proof (cases k = i)
      case True
      show ?thesis unfolding id True poly-smult using IH(2)[of i] ij 1(3−) diff
        by (simp add: X-def field-simps)
    next
      case False note ki = this
      show ?thesis
      proof (cases k = j)
        case True
        show ?thesis unfolding id True poly-smult using IH(1)[of j] ij 1(3−) diff
          by (simp add: X-def field-simps)
      next
        case False
          with ki show ?thesis unfolding id poly-smult using IH(1−2)[of k] ij
1(3−) diff
          by (simp add: X-def field-simps)
      qed
    qed
  qed
qed

lemma degree-neville-aitken-main: degree (neville-aitken-main i j) ≤ j − i
proof (induct i j rule: neville-aitken-main.induct)
  case (1 i j)
  note simp = neville-aitken-main.simps[of i j]
  show ?case
  proof (cases i < j)
    case False
    thus ?thesis unfolding simp by simp
  next
    case True
    note IH = 1[OF this]
    let ?n = neville-aitken-main
    have X: ⋀ i. degree (X i) = Suc 0 unfolding X-def by auto
    have degree (X i ∗ ?n (i + 1) j) ≤ Suc (degree (?n (i+1) j))
      by (rule order.trans[OF degree-mult-le], simp add: X)
    also have . . . ≤ Suc (j − (i+1)) using IH(1) by simp
    finally have 1: degree (X i ∗ ?n (i + 1) j) ≤ j − i using True by auto
```

have *degree* (*X j* ∗ *?n i* (*j* − *1*)) ≤ *Suc* (*degree* (*?n i* (*j* − *1*)))
by (*rule order.trans*[*OF degree-mult-le*], *simp add*: *X*)
also have . . . ≤ *Suc* ((*j* − *1*) − *i*) **using** *IH*(*2*) **by** *simp*
finally have *2*: *degree* (*X j* ∗ *?n i* (*j* − *1*)) ≤ *j* − *i* **using** *True* **by** *auto*
have *id*: *?n i j* = *smult* (*inverse* (*x j* − *x i*))
(*X i* ∗ *?n* (*i* + *1*) *j* − *X j* ∗ *?n i* (*j* − *1*)) **unfolding** *simp* **using** *True*
by *simp*
have *degree* (*?n i j*) ≤ *degree* (*X i* ∗ *?n* (*i* + *1*) *j* − *X j* ∗ *?n i* (*j* − *1*))
unfolding *id* **by** *simp*
also have . . . ≤ *max* (*degree* (*X i* ∗ *?n* (*i* + *1*) *j*)) (*degree* (*X j* ∗ *?n i* (*j* −
1)))
by (*rule degree-diff-le-max*)
also have . . . ≤ *j* − *i* **using** *1 2* **by** *auto*
finally show *?thesis* **.**
qed
qed

lemma *degree-neville-aitken*: *degree* (*neville-aitken n*) ≤ *n*
unfolding *neville-aitken-def* **using** *degree-neville-aitken-main*[*of 0 n*] **by** *simp*

fun *neville-aitken-merge* :: (*'a* × *'a* × *'a poly*) *list* ⇒ (*'a* × *'a* × *'a poly*) *list* **where**
neville-aitken-merge ((*xi*,*xj*,*p-ij*) # (*xsi*,*xsj*,*p-sisj*) # *rest*) =
(*xi*,*xsj*, *smult* (*inverse* (*xsj* − *xi*)) ([:−*xi*,*1*:] ∗ *p-sisj*
+ [:*xsj*,−*1*:] ∗ *p-ij*)) # *neville-aitken-merge* ((*xsi*,*xsj*,*p-sisj*) # *rest*)
| *neville-aitken-merge* [-] = []
| *neville-aitken-merge* [] = []

lemma *length-neville-aitken-merge*[*termination-simp*]: *length* (*neville-aitken-merge*
xs) = *length xs* − *1*
by (*induct xs rule*: *neville-aitken-merge.induct*, *auto*)

fun *neville-aitken-impl-main* :: (*'a* × *'a* × *'a poly*) *list* ⇒ *'a poly* **where**
neville-aitken-impl-main (*e1* # *e2* # *es*) =
neville-aitken-impl-main (*neville-aitken-merge* (*e1* # *e2* # *es*))
| *neville-aitken-impl-main* [(-,-,*p*)] = *p*
| *neville-aitken-impl-main* [] = *0*

lemma *neville-aitken-merge*:
xs = *map* (λ *i*. (*x i*, *x* (*i* + *j*), *neville-aitken-main i* (*i* + *j*))) [*l* ..< *Suc* (*l* + *k*)]

⟹ *neville-aitken-merge xs*
= (*map* (λ *i*. (*x i*, *x* (*i* + *Suc j*), *neville-aitken-main i* (*i* + *Suc j*))) [*l* ..< *l*
+ *k*])
proof (*induct xs arbitrary*: *l k rule*: *neville-aitken-merge.induct*)
case (*1 xi xj p-ij xsi xsj p-sisj rest l k*)
let *?n* = *neville-aitken-main*
let *?f* = λ *j i*. (*x i*, *x* (*i* + *j*), *?n i* (*i* + *j*))
define *f* **where** *f* = *?f*
let *?map* = λ *j*. *map* (*?f j*)

note *res* = *1*(*2*)
from *arg-cong*[*OF res*, *of length*] **obtain** *kk* **where** *k*: *k* = *Suc kk* **by** (*cases k*, *auto*)
hence *id*: [*l*..<*Suc* (*l* + *k*)] = *l* # [*Suc l* ..< *Suc* (*Suc l* + *kk*)]
by (*simp add*: *upt-rec*)
from *res*[*unfolded id*] **have** *id2*: (*xsi*, *xsj*, *p-sisj*) # *rest* =
?map j [*Suc l*..< *Suc* (*Suc l* + *kk*)]
and *id3*: *xi* = *x l xj* = *x* (*l* + *j*) *p-ij* = *?n l* (*l* + *j*)
xsi = *x* (*Suc l*) *xsj* = *x* (*Suc* (*l* + *j*)) *p-sisj* = *?n* (*Suc l*) (*Suc* (*l* + *j*))
by (*auto simp*: *upt-rec*)
note *IH* = *1*(*1*)[*OF id2*]
have *X*: [:*x* (*Suc* (*l* + *j*)), − *1*:] = − *X* (*Suc l* + *j*) **unfolding** *X-def* **by** *simp*
have *id4*: (*xi*, *xsj*, *smult* (*inverse* (*xsj* − *xi*)) ([:− *xi*, *1*:] ∗ *p-sisj* +
[:*xsj*, − *1*:] ∗ *p-ij*)) = (*x l*, *x* (*l* + *Suc j*), *?n l* (*l* + *Suc j*))
unfolding *id3 neville-aitken-main.simps*[*of l l* + *Suc j*]
X-def[*symmetric*] *X* **by** *simp*
have *id5*: [*l*..<*l* + *k*] = *l* # [*Suc l* ..< *Suc l* + *kk*] **unfolding** *k*
by (*simp add*: *upt-rec*)
show *?case* **unfolding** *neville-aitken-merge.simps IH id4*
unfolding *id5* **by** *simp*
qed *auto*

lemma *neville-aitken-impl-main*:
xs = *map* (λ *i*. (*x i*, *x* (*i* + *j*), *neville-aitken-main i* (*i* + *j*))) [*l* ..< *Suc* (*l* + *k*)]

⟹ *neville-aitken-impl-main xs* = *neville-aitken-main l* (*l* + *j* + *k*)
proof (*induct xs arbitrary*: *l k j rule*: *neville-aitken-impl-main.induct*)
case (*1 e1 e2 es l k j*)
note *res* = *1*(*2*)
from *res* **obtain** *kk* **where** *k*: *k* = *Suc kk* **by** (*cases k*, *auto*)
hence *id1*: *l* + *k* = *Suc* (*l* + *kk*) **by** *auto*
show *?case* **unfolding** *neville-aitken-impl-main.simps 1*(*1*)[*OF neville-aitken-merge*[*OF 1*(*2*), *unfolded id1*]]
by (*simp add*: *k*)
qed *auto*

lemma *neville-aitken-impl*:
xs = *map* (λ *i*. (*x i*, *x i*, [:*f i*:])) [*0* ..< *Suc k*]
⟹ *neville-aitken-impl-main xs* = *neville-aitken k*
unfolding *neville-aitken-def* **using** *neville-aitken-impl-main*[*of xs 0 0 k*]
by (*simp add*: *neville-aitken-main.simps*)
end

lemma *neville-aitken*: **assumes** ⋀ *i j*. *i* < *j* ⟹ *j* ≤ *n* ⟹ *x i* ≠ *x j*
shows *j* ≤ *n* ⟹ *poly* (*neville-aitken x f n*) (*x j*) = (*f j*)
unfolding *neville-aitken-def*
by (*rule neville-aitken-main*[*OF assms*, *of n*], *auto*)

definition *neville-aitken-interpolation-poly* :: (*'a* :: *field* × *'a*)*list* ⇒ *'a poly* **where**

neville-aitken-interpolation-poly x-fs = (*let*
start = *map* (λ (*xi*,*fi*). (*xi*,*xi*,[:*fi*:])) *x-fs in*
neville-aitken-impl-main start)

lemma *neville-aitken-interpolation-impl*: **assumes** *x-fs* ≠ []
shows *neville-aitken-interpolation-poly x-fs* =
neville-aitken (λ *i*. *fst* (*x-fs* ! *i*)) (λ *i*. *snd* (*x-fs* ! *i*)) (*length x-fs* − *1*)
proof −
from *assms* **have** *id*: *Suc* (*length x-fs* − *1*) = *length x-fs* **by** *auto*
show *?thesis*
unfolding *neville-aitken-interpolation-poly-def Let-def*
by (*rule neville-aitken-impl*, *unfold id*, *rule nth-equalityI*, *auto split*: *prod.splits*)
qed

lemma *neville-aitken-interpolation-poly*: **assumes** *dist*: *distinct* (*map fst xs-ys*)
and *p*: *p* = *neville-aitken-interpolation-poly xs-ys*
and *xy*: (*x*,*y*) ∈ *set xs-ys*
shows *poly p x* = *y*
proof −
have *p*: *p* = *neville-aitken* (λ *i*. *fst* (*xs-ys* ! *i*)) (λ *i*. *snd* (*xs-ys* ! *i*)) (*length xs-ys* − *1*)
unfolding *p*
by (*rule neville-aitken-interpolation-impl*, *insert xy*, *auto*)
from *xy* **obtain** *i* **where** *i*: *i* < *length xs-ys* **and** *x*: *x* = *fst* (*xs-ys* ! *i*) **and** *y*: *y* = *snd* (*xs-ys* ! *i*)
unfolding *set-conv-nth* **by** (*metis fst-conv in-set-conv-nth snd-conv xy*)
show *?thesis* **unfolding** *p x y*
proof (*rule neville-aitken*)
fix *i j*
show *i* < *j* ⟹ *j* ≤ *length xs-ys* − *1* ⟹ *fst* (*xs-ys* ! *i*) ≠ *fst* (*xs-ys* ! *j*) **using** *dist*
by (*metis* (*mono-tags*, *lifting*) *One-nat-def diff-less dual-order.strict-trans2 length-map*
length-pos-if-in-set lessI less-or-eq-imp-le neq-iff nth-eq-iff-index-eq nth-map xy)
qed (*insert i*, *auto*)
qed

lemma *degree-neville-aitken-interpolation-poly*:
shows *degree* (*neville-aitken-interpolation-poly xs-ys*) ≤ *length xs-ys* − *1*
proof (*cases length xs-ys*)
case *0*
hence *id*: *xs-ys* = [] **by** (*cases xs-ys*, *auto*)
show *?thesis* **unfolding** *id neville-aitken-interpolation-poly-def Let-def* **by** *simp*
next
case (*Suc nn*)
have *id*: *neville-aitken-interpolation-poly xs-ys* =
neville-aitken (λ *i*. *fst* (*xs-ys* ! *i*)) (λ *i*. *snd* (*xs-ys* ! *i*)) (*length xs-ys* − *1*)
by (*rule neville-aitken-interpolation-impl*, *insert Suc*, *auto*)

show *?thesis* **unfolding** *id* **by** (*rule degree-neville-aitken*)
qed

end

12 Polynomial Interpolation

We combine Newton's, Lagrange's, and Neville-Aitken's interpolation algorithms to a combined interpolation algorithm which is parametric. This parametric algorithm is then further extend from fields to also perform interpolation of integer polynomials.

In experiments it is revealed that Newton's algorithm performs better than the one of Lagrange. Moreover, on the integer numbers, only Newton's algorithm has been optimized with fast failure capabilities.

```
theory Polynomial-Interpolation
imports
  Improved-Code-Equations
  Newton-Interpolation
  Lagrange-Interpolation
  Neville-Aitken-Interpolation
begin

datatype interpolation-algorithm = Newton | Lagrange | Neville-Aitken

fun interpolation-poly :: interpolation-algorithm ⇒ ('a :: field × 'a)list ⇒ 'a poly
where
  interpolation-poly Newton = newton-interpolation-poly
| interpolation-poly Lagrange = lagrange-interpolation-poly
| interpolation-poly Neville-Aitken = neville-aitken-interpolation-poly

fun interpolation-poly-int :: interpolation-algorithm ⇒ (int × int)list ⇒ int poly
option where
  interpolation-poly-int Newton xs-ys = newton-interpolation-poly-int xs-ys
| interpolation-poly-int alg xs-ys = (let
    rxs-ys = map (λ (x,y). (of-int x, of-int y)) xs-ys;
    rp = interpolation-poly alg rxs-ys
    in if (∀ x ∈ set (coeffs rp). is-int-rat x) then
      Some (map-poly int-of-rat rp) else None)

lemma interpolation-poly-int-def: distinct (map fst xs-ys) ⟹
  interpolation-poly-int alg xs-ys = (let
    rxs-ys = map (λ (x,y). (of-int x, of-int y)) xs-ys;
    rp = interpolation-poly alg rxs-ys
    in if (∀ x ∈ set (coeffs rp). is-int-rat x) then
      Some (map-poly int-of-rat rp) else None)
  by (cases alg, auto simp: newton-interpolation-poly-int)
```

lemma *interpolation-poly*: **assumes** *dist*: *distinct* (*map fst xs-ys*)
and *p*: *p* = *interpolation-poly alg xs-ys*
and *xy*: (*x*,*y*) ∈ *set xs-ys*
shows *poly p x* = *y*
proof (*cases alg*)
case *Newton*
thus *?thesis* **using** *newton-interpolation-poly*[*OF dist - xy*] *p* **by** *simp*
next
case *Lagrange*
thus *?thesis* **using** *lagrange-interpolation-poly*[*OF dist - xy*] *p* **by** *simp*
next
case *Neville-Aitken*
thus *?thesis* **using** *neville-aitken-interpolation-poly*[*OF dist - xy*] *p* **by** *simp*
qed

lemma *degree-interpolation-poly*:
shows *degree* (*interpolation-poly alg xs-ys*) ≤ *length xs-ys* − *1*
using *degree-lagrange-interpolation-poly*[*of xs-ys*]
degree-newton-interpolation-poly[*of xs-ys*]
degree-neville-aitken-interpolation-poly[*of xs-ys*]
by (*cases alg*, *auto*)

lemma *uniqueness-of-interpolation*: **fixes** *p* :: *'a* :: *idom poly*
assumes *cS*: *card S* = *Suc n*
and *degree p* ≤ *n* **and** *degree q* ≤ *n* **and**
id: ⋀ *x*. *x* ∈ *S* ⟹ *poly p x* = *poly q x*
shows *p* = *q*
proof −
define *f* **where** *f* = *p* − *q*
let *?R* = {*x*. *poly f x* = *0*}
have *sub*: *S* ⊆ *?R* **unfolding** *f-def* **using** *id* **by** *auto*
show *?thesis*
proof (*cases f* = *0*)
case *True* **thus** *?thesis* **unfolding** *f-def* **by** *simp*
next
case *False* **note** *f* = *this*
let *?R* = {*x*. *poly f x* = *0*}
from *poly-roots-finite*[*OF f*] **have** *finite ?R* **.**
from *card-mono*[*OF this sub*] *poly-roots-degree*[*OF f*]
have *Suc n* ≤ *degree f* **unfolding** *cS* **by** *auto*
also have . . . ≤ *n* **unfolding** *f-def*
by (*rule degree-diff-le*, *insert assms*, *auto*)
finally show *?thesis* **by** *auto*
qed
qed

lemma *uniqueness-of-interpolation-point-list*: **fixes** *p* :: *'a* :: *idom poly*
assumes *dist*: *distinct* (*map fst xs-ys*)
and *p*: ⋀ *x y*. (*x*,*y*) ∈ *set xs-ys* ⟹ *poly p x* = *y degree p* < *length xs-ys*

and *q*: ⋀ *x y*. (*x*,*y*) ∈ *set xs-ys* ⟹ *poly q x* = *y degree q* < *length xs-ys*
shows *p* = *q*
proof −
let *?xs* = *map fst xs-ys*
from *q* **obtain** *n* **where** *len*: *length xs-ys* = *Suc n* **and** *dq*: *degree q* ≤ *n* **by** (*cases xs-ys*, *auto*)
from *p* **have** *dp*: *degree p* ≤ *n* **unfolding** *len* **by** *auto*
from *dist* **have** *card*: *card* (*set ?xs*) = *Suc n* **unfolding** *len*[*symmetric*]
using *distinct-card* **by** *fastforce*
show *p* = *q*
proof (*rule uniqueness-of-interpolation*[*OF card dp dq*])
fix *x*
assume *x* ∈ *set ?xs*
then obtain *y* **where** (*x*,*y*) ∈ *set xs-ys* **by** *auto*
from *p*(*1*)[*OF this*] *q*(*1*)[*OF this*] **show** *poly p x* = *poly q x* **by** *simp*
qed
qed

lemma *exactly-one-poly-interpolation*: **assumes** *xs*: *xs-ys* ≠ [] **and** *dist*: *distinct* (*map fst xs-ys*)
shows ∃! *p*. *degree p* < *length xs-ys* ∧ (∀ *x y*. (*x*,*y*) ∈ *set xs-ys* ⟶ *poly p x* = (*y* :: ′*a* :: *field*))
proof −
let *?alg* = *undefined*
let *?p* = *interpolation-poly ?alg xs-ys*
note *inter* = *interpolation-poly*[*OF dist refl*]
show *?thesis*
proof (*rule ex1I*[*of - ?p*], *intro conjI allI impI*)
show *dp*: *degree ?p* < *length xs-ys* **using** *degree-interpolation-poly*[*of ?alg xs-ys*] *xs* **by** (*cases xs-ys*, *auto*)
show ⋀*x y*. (*x*, *y*) ∈ *set xs-ys* ⟹ *poly* (*interpolation-poly ?alg xs-ys*) *x* = *y*
by (*rule inter*)
fix *q*
assume *q*: *degree q* < *length xs-ys* ∧ (∀ *x y*. (*x*, *y*) ∈ *set xs-ys* ⟶ *poly q x* = *y*)
show *q* = *?p*
by (*rule uniqueness-of-interpolation-point-list*[*OF dist - - inter dp*], *insert q*, *auto*)
qed
qed

lemma *interpolation-poly-int-Some*: **assumes** *dist*′: *distinct* (*map fst xs-ys*)
and *p*: *interpolation-poly-int alg xs-ys* = *Some p*
shows ⋀ *x y*. (*x*,*y*) ∈ *set xs-ys* ⟹ *poly p x* = *y degree p* ≤ *length xs-ys* − *1*
proof −
let *?r* = *rat-of-int*
define *rxs-ys* **where** *rxs-ys* = *map* (λ(*x*, *y*). (*?r x*, *?r y*)) *xs-ys*
have *dist*: *distinct* (*map fst rxs-ys*) **using** *dist*′ **unfolding** *distinct-map rxs-ys-def*

inj-on-def **by** *force*
obtain *rp* **where** *rp*: *rp* = *interpolation-poly alg rxs-ys* **by** *blast*
from *p*[*unfolded interpolation-poly-int-def*[*OF dist'*] *Let-def*, *folded rxs-ys-def rp*]
have *p*: *p* = *map-poly int-of-rat rp* **and** *ball*: *Ball* (*set* (*coeffs rp*)) *is-int-rat*
by (*auto split*: *if-splits*)
have *id*: *rp* = *map-poly ?r p* **unfolding** *p*
by (*rule sym*, *subst map-poly-map-poly*, *force*, *rule map-poly-idI*, *insert ball*, *auto*)
note *inter* = *interpolation-poly*[*OF dist rp*]
{
fix *x y*
assume (*x*,*y*) ∈ *set xs-ys*
hence (*?r x*, *?r y*) ∈ *set rxs-ys* **unfolding** *rxs-ys-def* **by** *auto*
from *inter*[*OF this*] **have** *poly rp* (*?r x*) = *?r y* **by** *auto*
from *this*[*unfolded id of-int-hom.poly-map-poly*] **show** *poly p x* = *y* **by** *auto*
}
show *degree p* ≤ *length xs-ys* − *1* **using** *degree-interpolation-poly*[*of alg rxs-ys*, *folded rp*]
unfolding *id rxs-ys-def* **by** *simp*
qed

lemma *interpolation-poly-int-None*: **assumes** *dist*: *distinct* (*map fst xs-ys*)
and *p*: *interpolation-poly-int alg xs-ys* = *None*
and *q*: ⋀ *x y*. (*x*,*y*) ∈ *set xs-ys* ⟹ *poly q x* = *y*
and *dq*: *degree q* < *length xs-ys*
shows *False*
proof −
let *?r* = *rat-of-int*
let *?rp* = *map-poly ?r*
define *rxs-ys* **where** *rxs-ys* = *map* (λ(*x*, *y*). (*?r x*, *?r y*)) *xs-ys*
have *dist'*: *distinct* (*map fst rxs-ys*) **using** *dist* **unfolding** *distinct-map rxs-ys-def inj-on-def* **by** *force*
obtain *rp* **where** *rp*: *rp* = *interpolation-poly alg rxs-ys* **by** *blast*
note *degrp* = *degree-interpolation-poly*[*of alg rxs-ys*, *folded rp*]
from *q* **have** *q'*: ⋀ *x y*. (*x*,*y*) ∈ *set rxs-ys* ⟹ *poly* (*?rp q*) *x* = *y* **unfolding** *rxs-ys-def*
by *auto*
have [*simp*]: *degree* (*?rp q*) = *degree q* **by** *simp*
have *id*: *rp* = *?rp q*
by (*rule uniqueness-of-interpolation-point-list*[*OF dist' interpolation-poly*[*OF dist' rp*]],
insert q' dq degrp, *auto simp*: *rxs-ys-def*)
from *p*[*unfolded interpolation-poly-int-def*[*OF dist*] *Let-def*, *folded rxs-ys-def rp*]
have ∃ *c* ∈ *set* (*coeffs rp*). *c* ∉ ℤ **by** (*auto split*: *if-splits*)
from *this*[*unfolded id*] **show** *False* **by** *auto*
qed

lemmas *newton-interpolation-poly-int-Some* =

interpolation-poly-int-Some[**where** *alg* = *Newton*, *unfolded interpolation-poly-int.simps*]

lemmas *newton-interpolation-poly-int-None* =
interpolation-poly-int-None[**where** *alg* = *Newton*, *unfolded interpolation-poly-int.simps*]

We can also use Newton's improved algorithm for integer polynomials to show that there is no polynomial p over the integers such that $p(0) = 0$ and $p(2) = 1$. The reason is that the intermediate result for computing the linear interpolant for these two point fails, and so adding further points (which corresponds to increasing the degree) will also fail. Of course, this can be generalized, showing that whenever you cannot interpolate a set of n points with an integer polynomial of degree $n - 1$, then you cannot interpolate this set of points with any integer polynomial. However, we did not formally prove this more general fact.

lemma *impossible-p-0-is-0-and-p-2-is-1*: ¬ (∃ *p*. *poly p 0* = *0* ∧ *poly p 2* = (*1* :: *int*))
proof
  assume ∃ *p*. *poly p 0* = *0* ∧ *poly p 2* = (*1* :: *int*)
  then obtain *p* **where** *p*: *poly p 0* = *0* *poly p 2* = (*1* :: *int*) **by** *auto*
  define *xs-ys* **where** *xs-ys* = *map* (λ *i*. (*int i*, *poly p* (*int i*))) [*3* ..< *3* + *degree p*]
  let *?l* = λ *xs*. (*0*,*0*) # (*2* :: *int*,*1* :: *int*) # *xs*
  let *?xs-ys* = *?l xs-ys*
  define *list* **where** *list* = *map fst ?xs-ys*
  have *dist*: *distinct* (*map fst ?xs-ys*) **unfolding** *xs-ys-def* **by** (*auto simp*: *o-def distinct-map inj-on-def*)
  have *p*: ⋀ *x y*. (*x*,*y*) ∈ *set ?xs-ys* ⟹ *poly p x* = *y* **unfolding** *xs-ys-def* **using** *p* **by** *auto*
  have *deg*: *degree p* < *length ?xs-ys* **unfolding** *xs-ys-def* **by** *simp*
  have *newton-coefficients-main-int list* (*rev* (*map snd ?xs-ys*)) (*rev* (*map fst ?xs-ys*)) = *None*
  proof (*induct xs-ys rule*: *rev-induct*)
    case *Nil*
    show *?case* **unfolding** *list-def* **by** (*simp add*: *divmod-int-def*)
  next
    case (*snoc xy xs-ys*) **note** *IH* = *this*
    obtain *x y* **where** *xy*: *xy* = (*x*,*y*) **by** *force*
    show *?case*
    proof (*cases xs-ys rule*: *rev-cases*)
      case *Nil*
      show *?thesis* **unfolding** *Nil xy*
        by (*simp add*: *list-def divmod-int-def*)
    next
      case (*snoc xs-ys′ xy′*)
      obtain *x′ y′* **where** *xy′*: *xy′* = (*x′*,*y′*) **by** *force*
      show *?thesis* **using** *IH* **unfolding** *xy′ snoc xy* **by** *simp*
    qed
  qed

```
  hence newton: newton-interpolation-poly-int ?xs-ys = None
    unfolding newton-interpolation-poly-int-def Let-def newton-poly-impl-int-def
      Newton-Interpolation.newton-coefficients-int-def list-def by simp
  from newton-interpolation-poly-int-None[OF dist newton p deg]
  show False .
qed

end
```

References

[1] G. M. Phillips. *Interpolation and Approximation by Polynomials.* Springer, 2003.