# Polynomial Interpolation* 

René Thiemann and Akihisa Yamada

September 13, 2023


#### 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

1 Introduction ..... 2
2 Conversions to Rational Numbers ..... 3
3 Divmod-Int ..... 5
4 Improved Code Equations ..... 6
4.1 divmod-integer. ..... 6
4.2 Euclidean-Rings.divmod-nat. ..... 7
4.3 (choose) ..... 8
5 Several Locales for Homomorphisms Between Types. ..... 8
5.1 Basic Homomorphism Locales ..... 8
5.2 Commutativity ..... 9
5.3 Division ..... 11
5.4 (Partial) Injectivitiy ..... 12

[^0]5.5 Surjectivity and Isomorphisms ..... 14
5.6 Example Interpretations ..... 18
6 Missing Unsorted ..... 18
7 Missing Polynomial ..... 32
7.1 Basic Properties ..... 32
7.2 Polynomial Composition ..... 36
7.3 Monic Polynomials ..... 37
7.4 Roots ..... 41
7.5 Divisibility ..... 42
7.6 Map over Polynomial Coefficients ..... 51
7.7 Morphismic properties of $p \operatorname{Cons}\left(0::^{\prime} a\right)$ ..... 52
7.8 Misc ..... 52
8 Connecting Polynomials with Homomorphism Locales ..... 61
8.1 map-poly of Homomorphisms ..... 63
8.1.1 Injectivity ..... 64
8.2 Example Interpretations ..... 70
9 Newton Interpolation ..... 71
10 Lagrange Interpolation ..... 90
11 Neville Aitken Interpolation ..... 93
12 Polynomial Interpolation ..... 98

## 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 $p\left(x_{i}\right)=y_{i}$ for every $i<n$.

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

$$
p(x)=\sum_{i<n}\left(y_{i} \cdot \prod_{\substack{j<n \\ j \neq i}} \frac{x-x_{j}}{x_{i}-x_{j}}\right)
$$

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=>\mathrm{ bool
    and to-rat :: ' }a=>\mathrm{ rat
    assumes is-rat[simp]: is-rat x = (x\in\mathbb{Q})
```



```
lemma of-rat-to-rat[simp]: x }\in\mathbb{Q}\Longrightarrow\mathrm{ 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 $y . x=o f$-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 $(o f-r a t q) 0$
proof -
obtain $d n$ where $d n$ : quotient-of $q=(d, n)$ by force
from quotient-of-div[OF $d n]$ have $q: q=o f$-int $d /$ of-int $n$ by auto
then have of-rat $q=$ complex-of-real (real-of-rat $q) \vee(0:: c o m p l e x)=o f-i n t ~ n \vee$
$0=$ 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: $x \in \mathbb{Q} \longleftrightarrow \operatorname{Re} x \in \mathbb{Q} \wedge \operatorname{Im} x=0$
proof
assume $x \in \mathbb{Q}$
then obtain $q$ where $x: x=o f-r a t ~ q u n f o l d i n g$ Rats-def by auto
let $? y=$ Complex $(o f-r a t q) 0$
have $x-? y=0$ unfolding $x$ by (simp add: Complex-eq)
hence $x$ : $x=$ ? $y$ by simp
show Re $x \in \mathbb{Q} \wedge \operatorname{Im} x=0$ unfolding $x$ complex.sel by auto
next
assume $\operatorname{Re} x \in \mathbb{Q} \wedge \operatorname{Im} x=0$
then obtain $q$ where Rex=of-rat $q \operatorname{Im} x=0$ 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 $(\operatorname{Re} x) \wedge \operatorname{Im} x=0)$
definition to-rat-complex ( $x::$ complex $)=($ if is-rat $(\operatorname{Re} x) \wedge \operatorname{Im} x=0$ 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 $r: \operatorname{Re} x \in \mathbb{Q}$ and $i: \operatorname{Im} x=0$
hence $x \in \mathbb{Q}$ unfolding is-rat-complex-iff by auto
then obtain $y$ where $x: x=o f-$ 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 $y . x=o f$-rat $y)$
by (subst of-rat-eq-iff[symmetric, where 'a $a$ real], unfold of-rat-to-rat $[O F r]$ of-rat-complex,
unfold $x$ complex.sel, auto)
qed
end
lemma [code-unfold]: $(x \in \mathbb{Q})=($ is-rat $x)$ by simp
definition is-int-rat :: rat $\Rightarrow$ bool where
is-int-rat $x \equiv$ snd (quotient-of $x)=1$
definition int-of-rat :: rat $\Rightarrow$ int where
int-of-rat $x \equiv f s t$ (quotient-of $x)$
lemma is-int-rat $[$ simp $]$ : is-int-rat $x=(x \in \mathbb{Z})$
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 int-of-rat $[$ simp $]$ : int-of-rat (rat-of-int $x)=x z \in \mathbb{Z} \Longrightarrow$ rat-of-int (int-of-rat $z)=z$
proof (force simp: int-of-rat-def)
assume $z \in \mathbb{Z}$
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 }\times\mathrm{ int where
    divmod-int n m}=(n\mathrm{ 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 $n$ ))
unfolding divmod-int-def divmod-integer-def map-prod-def split prod.simps
proof
show $m$ div $n=$ int-of-integer
(integer-of-int m div integer-of-int $n$ )
by (transfer, simp)
show $m$ mod $n=$ int-of-integer
(integer-of-int m mod integer-of-int $n$ )
by (transfer, simp)
qed
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 $(r, s) \Rightarrow$ 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 $(r, s) \Rightarrow$ 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 ( $0::^{\prime} a$ ) $\leq$
$x \wedge\left(0::^{\prime} b\right) \leq y$, so that application can be ensured on non-negative values. Hence, one can drop "abs" in target language setup.
definition divmod-abs ${ }^{\prime}$ where $x \geq 0 \Longrightarrow y \geq 0 \Longrightarrow$ divmod-abs ${ }^{\prime} x y=$ Code-Numeral.divmod-abs $x y$
lemma divmod-integer-code ${ }^{\prime \prime}[$ code $]$ : divmod-integer $k l=$
(if $k=0$ then ( 0,0 )
else if $l>0$ then
(if $k>0$ then divmod-abs ${ }^{\prime} k l$
else case divmod-abs ${ }^{\prime}(-k) l$ of $(r, s) \Rightarrow$
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 ${ }^{\prime}(-k)(-l)$ else case divmod-abs' $k(-l)$ of $(r, s) \Rightarrow$
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: div-mod-abs'-def divmod-abs-def)
code-printing - FIXME illusion of partiality
constant divmod-abs ${ }^{\prime}-$
(SML) IntInf.divMod/ ( -,/ - )
and (Eval) Integer.div'-mod/( - )/( - )
and (OCaml) Z.div'-rem
and (Haskell) divMod/ ( - )/( - )
and $(S c a l a)!((k$ : BigInt $)=>(l: \operatorname{BigInt})=>/$ if $(l==0) /(\operatorname{BigInt}(0), k)$ else $/$ ( $\left.k^{\prime} / \% l\right)$ )

### 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 $0<?$ then if $0<? k$ then divmod-abs ${ }^{\prime} ? k$ ?l else case div-mod-abs $(-? k)$ ?l of $(r, s) \Rightarrow$ 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 ${ }^{\prime}$ $(-? k)(-? l)$ else case divmod-abs' ? $k(-? l)$ of $(r, s) \Rightarrow$ 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 $n \operatorname{div}($ 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<br>imports<br>HOL.Complex<br>Main<br>HOL-Library.Multiset<br>HOL-Computational-Algebra.Factorial-Ring<br>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 $\Rightarrow$ ' $b::$ zero
assumes hom-zero $[\operatorname{simp}]$ : hom $0=0$
locale one-hom =
fixes hom :: ' $a$ :: one $\Rightarrow$ ' $b::$ one
assumes hom-one $[$ simp $]$ : hom $1=1$
locale times-hom $=$
fixes hom :: ' $a$ :: times $\Rightarrow$ ' $b::$ times
assumes hom-mult[hom-distribs]: hom $(x * y)=$ hom $x *$ hom $y$
locale plus-hom $=$
fixes hom :: ' $a$ :: plus $\Rightarrow$ ' $b::$ 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 $\Rightarrow$ ' $b::$ semigroup-mult
locale semigroup-add-hom $=$
plus-hom hom for hom :: ' $a$ :: semigroup-add $\Rightarrow{ }^{\prime} b$ :: semigroup-add
locale monoid-mult-hom $=$ one-hom hom + semigroup-mult-hom hom
for hom :: ' $a$ :: monoid-mult $\Rightarrow$ ' $b::$ 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 $(x \wedge n)=$ hom $x \wedge n$ by (induct $n$, auto simp: hom-distribs)
end
locale monoid-add-hom $=$ zero-hom hom + semigroup-add-hom hom
for hom :: ' $a$ :: monoid-add $\Rightarrow$ ' $b$ :: 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 $x+h o m y=0$
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 $\Rightarrow$ '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 $x-h o m y$ 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 $\Rightarrow$ ' $b::$ 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 }=>\mp@subsup{}{}{\prime}'b :: comm-monoid-mult
begin
    sublocale ab-semigroup-mult-hom..
    lemma hom-prod[hom-distribs]: hom (prod f X) = (\prodx\inX.hom (fx))
        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 pdvd q shows hom p dvd hom q
    proof -
        from assms obtain r where q}=p*r\mathrm{ 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 \Longrightarrow 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)=(\sumx\inX.hom (fx))
        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 吘'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 $\Rightarrow{ }^{\prime} b$ :: comm-semiring-1
begin
sublocale comm-monoid-mult-hom..
end
locale comm-ring-hom $=$ ring-hom hom
for hom :: ' $a$ :: comm-ring-1 $\Rightarrow^{\prime} b$ :: comm-ring-1
begin
sublocale comm-semiring-hom..
end
locale idom-hom $=$ comm-ring-hom hom
for hom :: 'a :: idom $\Rightarrow$ ' $b::$ idom

### 5.3 Division

```
locale idom-divide-hom \(=\) idom-hom hom
    for hom :: ' \(a\) :: idom-divide \(\Rightarrow\) ' \(b\) :: 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 \(\Rightarrow\) ' \(b::\) 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 \(\operatorname{hom}(x / y)=\) 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 $\Rightarrow$ ' $b::$ 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 $f::{ }^{\prime} a \Rightarrow$ ' $b$ 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 $x 0: x \neq 0$
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 $x 0$ 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 $f::^{\prime} a \Rightarrow{ }^{\prime} b$
assumes surj: surj $f$
begin
lemma $f$-inv-f[simp]: $f(\operatorname{inv} f x)=x$
by (rule cong, auto simp: surj[unfolded surj-iff o-def id-def])
end
locale bijective $=$ injective + surjective
lemma bijective-eq-bij: bijective $f=b i j f$
proof (intro iffI)

```
    assume bijective f
    then interpret bijective f.
    show bijf 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]: fx=y\longleftrightarrowx=inv f y by auto
    lemma inv-f-eq-iff[simp]: inv f}x=y\longleftrightarrowx=fy\mathrm{ 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 }=>\mp@subsup{}{}{\prime}'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 \longleftrightarrowx 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\mathrm{ dvd y by (intro dvdI)
    qed (rule hom-dvd)
    lemma hom-dvd-simp[simp]:
        shows hom x dvd y'}\longleftrightarrowx 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 $\Rightarrow$ ' $b$ :: 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 $\Rightarrow^{\prime} b::$ 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 $\Rightarrow^{\prime} b::$ 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 $\Rightarrow$ ' $b$ :: 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
$i d$ : hom $q=$ hom $p * k$ by auto
hence (hom $q$ div hom $p)=($ hom $p * k)$ div hom $p$ by simp
also have $\ldots=k$ 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 $k=h o m(q$ div $p$ ) by auto
from id[unfolded this] have hom $q=\operatorname{hom}(p *(q$ div $p))$ by (simp add:

## hom-mult)

hence $q=p *(q$ div $p)$ by simp
thus $p d v d q$ unfolding $d v d$-def ..
qed $\operatorname{simp}$
qed $\operatorname{simp}$
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\mathrm{ :: {ring-1,ring-char-0}
    by (unfold-locales, auto)
interpretation of-int-hom: inj-comm-ring-hom of-int :: int 和 }a::{\mathrm{ {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 $c::{ }^{\prime} a$ :: semir-ing-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 $x:-1 \leq(x::$ ' $a$ :: linordered-field $)$
shows $1+$ of-nat $n * x \leq(1+x) \wedge n$
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 $\ldots \leq \ldots+$ of-nat $n * x \wedge 2$ by $\operatorname{simp}$

```
    also have ... =(1 + of-nat n*x)* (1 +x) by (simp add: field-simps
power2-eq-square)
    also have ... \leq (1+x)^ n* (1+x)
        by (rule mult-right-mono[OF Suc], insert x, auto)
    also have ... = (1+x) ^
    finally show ?case .
qed simp
context
    fixes b :: 'a :: archimedean-field
    assumes b: 0<bb<1
begin
private lemma pow-one: }\mp@subsup{b}{}{\wedge}x\leq1 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 \exists x. b^ }x\leq
proof (rule ccontr)
    assume not: \neg?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\mathrm{ and }bbb:bbb>0\mathrm{ and bm1: bbb }\geq-1\mathrm{ unfolding bbb-def
using bb by auto
    have \exists n.cc / bbb<of-nat n by (rule reals-Archimedean2)
    then obtain n}\mathrm{ where lt:cc / bbb<of-nat n by auto
    from not have }\neg\mp@subsup{b}{}{`}n\leqc\mathrm{ by auto
    hence bnc: b ^ n>c by simp
    have bb` n = inverse ( }b\mp@subsup{}{}{`}n)\mathrm{ 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 ... \leq 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: \exists p.\forallx.b^}x*\mathrm{ of-nat }x\leq
proof -
    from b have 1-b>0 by simp
    from exp-tends-to-zero[OF this]
    obtain x0 where x0: b^ x0 \leq 1-b ..
    {
        fix }
        assume }x\geqx
```

```
    hence }\existsy.x=x0+y\mathrm{ by arith
    then obtain }y\mathrm{ where }x:x=x0+y\mathrm{ by auto
    have b^x = b^x x * b^y unfolding x by (simp add: power-add)
    also have ... \leq b ^x0 using pow-one[of y] pow-zero[of x0] by auto
    also have ... \leq1-b by (rule x0)
    finally have b}\mp@subsup{}{}{`}x\leq1-b
    } note x0 = this
    define bs where bs=insert 1 { b^`Suc x*of-nat (Suc x)|x.x\leqx0}
    have bs: finite bs unfolding bs-def by auto
    define p}\mathrm{ where p=Max bs
    have bs: \bigwedge b. b\inbs\Longrightarrowb\leqp unfolding p-def using bs by simp
    hence p1:p\geq1 unfolding bs-def by auto
    show ?thesis
    proof (rule exI[of-p], intro allI)
        fix }
        show b` x * of-nat x\leqp
        proof (induct x)
            case (Suc x)
            show ?case
            proof (cases x \leqx0)
                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 .. S 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 ... \leqp-0
            proof -
                have b^ Suc x s 1 - b using x0[of Suc x] False by auto
                    also have \ldots\leq (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: \exists p.\forallx. b^ x* of-nat x^deg \leq 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 \leqq by auto
    define p where p=max 0q
    from IH have IH:^x. 觙 x* (of-nat x)^deg \leqpunfolding p-def using
le-max-iff-disj by blast
    have p:p\geq0 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 = \lambdax.b ^}x*(of-nat x)^ Suc de
        define f}\mathrm{ where f=?f
        {
        fix }
        let ?x = of-nat x :: 'a
        have f(2*x)\leq(2 ^Suc deg)*(p*p)
        proof (cases x = 0)
            case False
            hence x1: ? }x\geq1\mathrm{ by (cases }x\mathrm{ , auto)
            from x1 have x: ? x ^ (deg - 1) \geq 1 by simp
            from x1 have xx: ? x ^ Suc deg \geq1 by (rule one-le-power)
            define }c\mathrm{ where }c=\mp@subsup{b}{}{\wedge}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)=( ( ^ 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 ... \leqc*?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*? ? ^(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)
```



```
deg))
            unfolding c-def by (simp add: ac-simps)
            also have .. . \leq (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 $f(2 * x) \leq\left(2^{\wedge}\right.$ Suc deg $) *(p * p)$.
qed (auto simp: $f$-def)
hence ?f $(2 * x) \leq\left(\mathcal{Z}^{\wedge}\right.$ Suc deg $) *(p * p)$ unfolding $f$-def.
$\}$ note even $=$ this
show ?thesis
proof (rule exI $[o f-p]$, intro allI)
fix $y$
show ?f $y \leq p^{\prime}$
proof (cases even $y$ )
case True
define $x$ where $x=y \operatorname{div} 2$
have $y=2 * x$ unfolding $x$-def using True by simp
from even[of $x$, folded this] have ?f $y \leq 2^{\wedge}$ Suc deg $*(p * p)$.
also have $\ldots \leq \ldots *$ inverse $b$
unfolding mult-le-cancel-left1 using $b p$
by (simp add: algebra-split-simps one-le-inverse)
also have $\ldots=p^{\prime}$ unfolding $p^{\prime}$-def by (simp add: ac-simps)
finally show ?f $y \leq p^{\prime}$.
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 $\ldots \leq b^{\wedge}(2 * x+1) *$ of-nat $(2 * x+2){ }^{\wedge}$ Suc deg
by (rule mult-left-mono[OF power-mono], insert b, auto)
also have $b^{\wedge}(2 * x+1)=b^{\wedge}(2 * x+2) *$ inverse $b$ using $b$ by auto also have $b^{\wedge}(2 * x+2) *$ inverse $b *$ of-nat $(2 * x+2) \wedge$ Suc deg $=$ inverse $b *$ ? $(2 *(x+1))$ by (simp add: ac-simps)
also have $\ldots \leq$ inverse $b *\left(\left(2{ }^{\text {^Suc deg })} *(p * p)\right)\right.$
by (rule mult-left-mono[OF even], insert b, auto)
also have $\ldots=p^{\prime}$ unfolding $p^{\prime}$-def by (simp add: ac-simps)
finally show ?f $y \leq p^{\prime}$.
qed
qed
qed
qed
qed
end
lemma prod-list-replicate[simp]: prod-list (replicate $n$ a) $=a{ }^{\wedge} n$ by (induct $n$, auto)
lemma prod-list-power: fixes $x s$ :: ' $a$ :: comm-monoid-mult list
shows prod-list $x{ }^{\wedge} n=\left(\prod x \leftarrow x s, x^{\wedge} n\right)$
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]:(\prodi=0..<n.p) = ( p :: 'a :: comm-monoid-mult) ^ n
    by (induct n, auto simp: set-upt-Suc)
```

lemma dvd-abs-mult-left-int [simp]:
$|a| * y d v d x \longleftrightarrow a * y d v d x$ for $x y a::$ int
using abs-dvd-iff [of $a * y$ ] abs-dvd-iff $[o f|a| * y]$
by (simp add: abs-mult)
lemma gcd-abs-mult-right-int [simp]:
$\operatorname{gcd} x(|a| * y)=\operatorname{gcd} x(a * y)$ 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 $x(|a| * y)=\operatorname{lcm} x(a * y)$ 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 $x(a *|y|)=\operatorname{gcd} x(a * y)$ 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 $\Rightarrow{ }^{\prime} a$ where
list-gcd $\equiv g c d-l i s t$
abbreviation (input) list-lcm :: ' $a$ :: semiring-gcd list $\Rightarrow$ ' $a$ where
list-lcm $\equiv$ lcm-list
lemma list-gcd-simps: list-gcd []$=0$ list-gcd $(x \# x s)=g c d x(l i s t-g c d x s)$
by simp-all
lemma list-gcd: $x \in$ set $x s \Longrightarrow$ list-gcd xs dvd $x$
by (fact Gcd-fin-dvd)
lemma list-gcd-greatest: $(\bigwedge x . x \in$ set $x s \Longrightarrow y d v d x) \Longrightarrow y d v d$ (list-gcd $x s)$
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 fet xs \Longrightarrowx dvd list-lcm xs
    by (fact dvd-Lcm-fin)
lemma list-lcm-least: (\bigwedge x. x 的 xs \Longrightarrow x dvd y)\Longrightarrowlist-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 \geq (0 :: int)
    0}\not\in\mathrm{ set xs \ list-lcm xs }\not=
    0\not\in set xs \Longrightarrow list-lcm xs > 0
proof -
    have 0\leq LLcm (set xs)|
        by (simp only: abs-ge-zero)
    then have 0\leqLcm (set xs)
        by simp
    then show list-lcm xs \geq0
        by simp
    assume 0 & set xs
    then show list-lcm xs \not=0
        by (simp add: Lcm-0-iff)
    with <list-lcm xs \geq0\rangle show list-lcm xs > 0
        by (simp add: le-less)
qed
lemma quotient-of-nonzero: snd (quotient-of r)>0 snd (quotient-of r) }=
    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\not=0
```

```
    shows \existsz. z\not=0\wedgex=a*z\wedge y=b*z
proof -
    let ?r = rat-of-int
    define z where z=gcd x y
    define }\mp@subsup{x}{}{\prime}\mathrm{ where }\mp@subsup{x}{}{\prime}=x\operatorname{div}
    define }\mp@subsup{y}{}{\prime}\mathrm{ where }\mp@subsup{y}{}{\prime}=y\mathrm{ div z
    have id: x=z* x'y=z* y' unfolding x'-def y'-def z-def by auto
    from }y\mathrm{ have }\mp@subsup{y}{}{\prime}:\mp@subsup{y}{}{\prime}\not=0\mathrm{ unfolding id by auto
    have z:z\not=0 unfolding z-def using y by auto
    have cop: coprime }\mp@subsup{x}{}{\prime}\mp@subsup{y}{}{\prime}\mathrm{ unfolding }\mp@subsup{x}{}{\prime}\mathrm{ -def }\mp@subsup{y}{}{\prime}\mathrm{ -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\not=0 by auto
    from quotient-of-div[OF quot] quotient-of-denom-pos[OF quot] y'
    have ?r 和* ?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 }\mp@subsup{z}{}{\prime}\mathrm{ where ybz: y'}=b*\mp@subsup{z}{}{\prime}\mathrm{ unfolding dvd-def by auto
    from id[unfolded y' this] have y:y=b*(z*\mp@subsup{z}{}{\prime})\mathrm{ by auto}
    with }\langley\not=0\rangle\mathrm{ have }zz:z*\mp@subsup{z}{}{\prime}\not=0\mathrm{ by auto
    from quotient-of-div[OF q] }\langley\not=0\rangle\langleb\not=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*\mp@subsup{z}{}{\prime})\mathrm{ by auto}
    show ?thesis unfolding x y using zz by blast
qed
fun max-list-non-empty :: ('a :: linorder) list }=>\mp@subsup{)}{}{\prime}a\mathrm{ 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\in\mathrm{ set xs }\Longrightarrowx\leqmax-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.coboundedIL)
    qed (insert oCons, auto)
qed simp
```

```
lemma cnj-reals[simp]:(cnj c\in\mathbb{R})=(c\in\mathbb{R})
    using Reals-cnj-iff by fastforce
lemma sgn-real-mono: }x\leqy\Longrightarrow\operatorname{sgn}x\leq\operatorname{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) \leqinverse y)}=(y\leqx
proof (cases x=0)
    case True
    with sgn have sgn y=0 by simp
    hence }y=0\mathrm{ 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\mathrm{ sgn have sgn }y=-1\mathrm{ 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\mathrm{ have }x:x>0\mathrm{ 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}\langley>0\rangle]
    qed
qed
lemma inverse-le-sgn: assumes sgn: sgn x = sgn y and xy:x\leq(y :: real)
    shows inverse }y\leq\mathrm{ inverse }
    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 ixs) \cup 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) \in set xs
    shows x dvd prod-list xs
proof -
    from assms[unfolded in-set-conv-decomp] obtain ys zs where xs:xs=ys@ @
# 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 }\existsb\inA. a dvd f b finite 
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 (\lambda x. \negf 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\mathrm{ 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\not=0 by auto
    then have }a*c\mathrm{ div }a=c\mathrm{ by auto
    also note *[symmetric]
    finally show ?thesis.
qed
```

```
lemma (in semidom) prod-list-zero-iff [simp]:
    prod-list xs=0\longleftrightarrow0\in set xs by (induction xs,auto)
context comm-monoid-mult begin
lemma unit-prod [intro]:
    shows a dvd 1 \Longrightarrowb dvd 1 \Longrightarrow(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\longleftrightarrowa dvd 1^b dvd 
    by (auto dest: dvd-mult-left dvd-mult-right)
end
context comm-semiring-1
begin
lemma irreducibleE[elim]:
    assumes irreducible p
        and p\not=0\Longrightarrow\negp dvd 1\Longrightarrow(\bigwedgeab.p=a*b\Longrightarrowadvd 1\veeb dvd 1) \Longrightarrow
thesis
    shows thesis using assms by (auto simp: irreducible-def)
lemma not-irreducibleE:
    assumes \neg irreducible x
        and }x=0\Longrightarrow\mathrm{ thesis
        and x dvd 1\Longrightarrow thesis
        and }\bigwedgeab.x=a*b\Longrightarrow\nega dvd 1\Longrightarrow\negb dvd 1\Longrightarrow thesi
        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 \existsa\in 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 \existsa\in# 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*bdvdc\longleftrightarrowa dvd 
proof
    assume a*bdvd 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\mathrm{ dvd c by fastforce
qed
lemma mult-unit-dvd-iff '[simp]: a dvd 1 \Longrightarrow(a*b)dvd c\longleftrightarrowb dvd c
    using mult-unit-dvd-iff [of a blc] by (simp add: ac-simps)
lemma irreducibleD':
    assumes irreducible a b dvd a
    shows a dvd b\vee 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 \vee c dvd 1.
    thus ?thesis by (auto simp: c)
qed
end
```


## context idom <br> begin

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

```
lemma dvd-times-left-cancel-iff [simp]:
    assumes \(a \neq 0\)
    shows \(a * b d v d a * c \longleftrightarrow b d v d c\)
        (is ?lhs \(\longleftrightarrow\) ? rhs)
proof
    assume ?lhs
    then obtain \(d\) where \(a * c=a * b * d\)..
    with assms have \(c=b * d\) by (auto simp add: ac-simps)
    then show ?rhs ..
next
    assume ?rhs
    then obtain \(d\) where \(c=b * d\)..
    then have \(a * c=a * b * d\) by (simp add: ac-simps)
    then show? lhs ..
```

```
qed
lemma dvd-times-right-cancel-iff [simp]:
    assumes a\not=0
    shows b*advd c*a\longleftrightarrowb dvd c
    using dvd-times-left-cancel-iff [of a b c] assms by (simp add: ac-simps)
lemma irreducibleI':
    assumes }a\not=0\nega\mathrm{ dvd }1\bigwedgeb.bdvd a\Longrightarrowa dvd b\veeb dvd 1
    shows irreducible a
proof (rule irreducibleI)
    fix b c assume a-eq: a = b*c
    hence a dvd b\veeb dvd 1 by (intro assms) simp-all
    thus b dvd 1\vee c dvd 1
    proof
        assume a dvd b
        hence b*cdvd b*1 by (simp add: a-eq)
        moreover from <a\not=0\rangle a-eq have b\not=0 by auto
        ultimately show ?thesis using dvd-times-left-cancel-iff by fastforce
    qed blast
qed (simp-all add: assms(1,2))
lemma irreducible-altdef:
    shows irreducible }x\longleftrightarrowx\not=0\wedge\negx\mathrm{ dvd 1 ^( }\forall\textrm{b}.b\mathrm{ dvd x }\longrightarrowx\mathrm{ dvd b}\veebdv
1)
    using irreducibleI'[of x] irreducibleD'[of x] irreducible-not-unit[of x] by auto
lemma dvd-mult-unit-iff:
    assumes b: b dvd 1
    shows a dvd c*b\longleftrightarrowa dvd c
proof-
    from b obtain b' where 1:b* b
    then have b0: b\not=0 by auto
    from 1 have }a=(a*\mp@subsup{b}{}{\prime})*b\mathrm{ by (simp add: ac-simps)
    also have ...dvd c*b\longleftrightarrowa<a* b
    finally show ?thesis by (auto intro: dvd-mult-left)
qed
lemma dvd-mult-unit-iff':}b\mathrm{ dvd 1 >advd 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 \Longrightarrow irreducible ( a*p) \longleftrightarrow irreducible p
    by (auto simp: irreducible-altdef mult.commute[of a] dvd-mult-unit-iff)
lemma irreducible-mult-unit-right:
    shows a dvd 1 \Longrightarrow irreducible ( p*a) \longleftrightarrow 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 ab
    assume p-eq: p=a*b
    with assms have nz:a\not=0 b\not=0 by auto
    from p-eq have p dvd a*b by simp
    with 〈prime-elem p> have p dvd a\veep dvd b by (rule prime-elem-dvd-multD)
    with <p=a*b\rangle have }a*bdvd 1*b\veea*bdvd a*1 by aut
    thus a dvd 1\veeb dvd 1
    by (simp only:dvd-times-left-cancel-iff[OF nz(1)] dvd-times-right-cancel-iff[OF
nz(2)])
qed (insert assms, simp-all add: prime-elem-def)
lemma unit-imp-dvd [dest]: b dvd 1\Longrightarrowb dvd a
    by (rule dvd-trans [of-1]) simp-all
lemma unit-mult-left-cancel: a dvd 1 \Longrightarrowa*b=a*c\longleftrightarrowb=c
    using mult-cancel-left [of a b c] by auto
lemma unit-mult-right-cancel: a dvd 1 \Longrightarrow b*a=c*a\longleftrightarrowb=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 \vee b dvd 1 ^ irreducible a
proof-
    from l have a dvd 1 \vee 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 $1 \wedge$ irreducible $b \vee b$ dvd $1 \wedge$ 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
```


### 7.1 Basic Properties

lemma degree-0-id: assumes degree $p=0$
shows [: coeff $p$ 0 : $0=p$
proof -
have $\wedge x .0 \neq$ Suc $x$ by auto
thus ?thesis using assms
by (metis coeff-pCons-0 degree-pCons-eq-if pCons-cases)
qed
lemma degree0-coeffs: degree $p=0 \Longrightarrow$
$\exists$ a. $p=[: a:]$
by (metis degree-pCons-eq-if old.nat.distinct(2) pCons-cases)
lemma degree1-coeffs: degree $p=1 \Longrightarrow$
$\exists a b . p=[: b, a:] \wedge a \neq 0$
by (metis One-nat-def degree-pCons-eq-if nat.inject old.nat.distinct(2) pCons-0-0
pCons-cases)
lemma degree2-coeffs: degree $p=2 \Longrightarrow$
$\exists a b c . p=[: c, b, a:] \wedge a \neq 0$
by (metis Suc-1 Suc-neq-Zero degree1-coeffs degree-pCons-eq-if nat.inject pCons-cases)
lemma poly-zero:
fixes $p::$ ' $a$ :: comm-ring-1 poly
assumes $x$ : poly $p x=0$ shows $p=0 \longleftrightarrow$ degree $p=0$
proof
assume degp: degree $p=0$
hence poly $p x=$ coeff $p$ (degree $p$ ) by (subst degree-0-id[OF degp,symmetric], simp)
hence coeff $p($ degree $p)=0$ using $x$ by auto
thus $p=0$ by auto
qed auto
lemma coeff-monom-Suc: coeff (monom a $($ Suc d) $* p)($ Suc $i)=$ coeff (monom ad*p) $i$
by (simp add: monom-Suc)
lemma coeff-sum-monom:
assumes $n$ : $n \leq d$
shows coeff $\left(\sum i \leq d\right.$. monom $\left.(f i) i\right) n=f n($ is $? l=-)$
proof -
have ?l $=\left(\sum i \leq d\right.$. coeff $\left.(\operatorname{monom}(f i) i) n\right)($ is $-=s u m ? c m f-)$ using coeff-sum.
also have $\{. . d\}=$ insert $n(\{. . d\}-\{n\})$ using $n$ by auto
hence sum? cmf $\{. . d\}=$ sum ?cmf $\ldots$ by auto
also have $\ldots=\operatorname{sum}$ ?cmf $(\{. . d\}-\{n\})+$ ?cmf $n$ by (subst sum.insert,auto)
also have sum ? cmf $(\{. . d\}-\{n\})=0$ by (subst sum.neutral, auto)
finally show? ?thesis by simp
qed
lemma linear-poly-root: ( $a::$ ' $a::$ comm-ring-1) $\in$ set $a s \Longrightarrow \operatorname{poly}\left(\prod a \leftarrow a s .[\right.$ : $-a, 1:]) a=0$
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 $\operatorname{simp}$
qed $\operatorname{simp}$
lemma degree-lcoeff-sum: assumes deg: degree $(f q)=n$
and fin: finite $S$ and $q: q \in S$ and degle: $\wedge p . p \in S-\{q\} \Longrightarrow$ degree $(f p)<$
$n$
and cong: coeff $(f q) n=c$
shows degree $(\operatorname{sum} f S)=n \wedge$ coeff $(\operatorname{sum} f S) n=c$
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 $(\operatorname{sum} f S)=$ degree $(f q+\operatorname{sum} f(S-\{q\}))$
unfolding sum.remove $[O F$ fin $q]$..

```
    also have ... = degree ( f q)
    proof (rule degree-add-eq-left)
    have degree (sum f (S-{q})) \leqn-1
    proof (rule degree-sum-le)
        fix p
        show }p\inS-{q}\Longrightarrow\mathrm{ degree (f p) sn-1
            using degle[of p] by auto
    qed (insert fin, auto)
    also have ...<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 (rule conjI)
    have id: (\sumx\inS - {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 fS) (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 degree-sum-list-le:(\ p.p set ps \Longrightarrow degree p\leqn)
    C degree (sum-list ps) \leqn
proof (induct ps)
    case (Cons p ps)
    hence degree (sum-list ps)\leqn degree p \leqn by auto
    thus ?case unfolding sum-list.Cons by (metis degree-add-le)
qed simp
lemma degree-prod-list-le: degree (prod-list ps)\leqsum-list (map degree ps)
proof (induct ps)
    case (Cons p ps)
    show ?case unfolding prod-list.Cons
        by (rule order.trans[OF degree-mult-le], insert Cons, auto)
qed simp
lemma smult-sum: smult (\sumi\inS.fi) p=(\sumi\inS.smult (fi)p)
    by (induct S rule: infinite-finite-induct, auto simp: smult-add-left)
lemma range-coeff: range (coeff p)= insert 0 (set (coeffs p))
    by (metis nth-default-coeffs-eq range-nth-default)
lemma smult-power: (smult a p) ^n = smult (a` n) ( }\mp@subsup{p}{}{`}n
    by (induct n, auto simp: field-simps)
lemma poly-sum-list: poly (sum-list ps) x = sum-list (map ( }\lambda\mathrm{ p.poly p x) ps)
    by (induct ps,auto)
```

```
lemma poly-prod-list: poly (prod-list ps) \(x=\) prod-list ( \(\operatorname{map}(\lambda p . p o l y p x) p s)\)
    by (induct ps, auto)
lemma sum-list-neutral: \((\bigwedge x . x \in\) set \(x s \Longrightarrow x=0) \Longrightarrow\) sum-list \(x s=0\)
    by (induct xs, auto)
lemma prod-list-neutral: ( \(\bigwedge x . x \in\) set \(x s \Longrightarrow x=1\) ) \(\Longrightarrow\) prod-list \(x s=1\)
    by (induct xs, auto)
lemma (in comm-monoid-mult) prod-list-map-remove1:
    \(x \in\) set \(x s \Longrightarrow\) prod-list \((\operatorname{map} f x s)=f x * \operatorname{prod}\)-list \((\operatorname{map} f(\) remove1 \(x x s))\)
    by (induct xs) (auto simp add: ac-simps)
lemma poly-as-sum:
    fixes \(p::\) ' \(a::\) :comm-semiring-1 poly
    shows poly \(p x=\left(\sum i \leq\right.\) degree \(p . x^{\wedge} i *\) coeff \(\left.p i\right)\)
    unfolding poly-altdef by (simp add: ac-simps)
lemma poly-prod-0: finite \(p s \Longrightarrow\) poly \((\) prod \(f p s) x=\left(0::{ }^{\prime} a::\right.\) field \() \longleftrightarrow(\exists p\)
\(\in\) 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 \(d \leq i\) then \(a *\) coeff \(p(i-d)\) else 0\()(\) is ?l \(=? r\) )
proof (cases \(d \leq i\) )
    case False thus ?thesis unfolding coeff-mult by simp
    next case True
        let ?f \(=\lambda j\). coeff (monom a d) \(j *\) coeff \(p(i-j)\)
        have \(\wedge j . j \in\{0 . . i\}-\{d\} \Longrightarrow\) ?f \(j=0\) by auto
        hence \(0=\left(\sum j \in\{0 . . i\}-\{d\}\right.\). ?f \(\left.j\right)\) by auto
        also have..+ ?f \(d=\left(\sum j \in\right.\) insert \(d(\{0 . . i\}-\{d\})\). ?f \(\left.j\right)\)
            by(subst sum.insert, auto)
    also have \(\ldots=\left(\sum j \in\{0 . . i\}\right.\). ?f \(j\) ) by (subst insert-Diff, insert True, auto)
    also have \(\ldots=\left(\sum j \leq i\right.\). ?f \(\left.j\right)\) by (rule sum.cong, auto)
    also have \(\ldots=\) ?l unfolding coeff-mult ..
    finally show ?thesis using True by auto
qed
lemma poly-eqI2:
    assumes degree \(p=\) degree \(q\) and \(\bigwedge i . i \leq\) degree \(p \Longrightarrow\) coeff \(p i=\operatorname{coeff} q i\)
    shows \(p=q\)
    apply(rule poly-eqI) by (metis assms le-degree)
        A nice extension rule for polynomials.
lemma poly-ext[intro]:
    fixes \(p q::{ }^{\prime} a::\{\) ring-char-0, idom \(\}\) poly
    assumes \(\bigwedge x\). poly \(p x=\) poly \(q x\) shows \(p=q\)
    unfolding poly-eq-poly-eq-iff[symmetric]
```

using assms by (rule ext)
Copied from non-negative variants.

```
lemma coeff-linear-power-neg[simp]:
    fixes a :: 'a::comm-ring-1
    shows coeff ([:a, -1:] ^ n) n=(-1)`n
apply (induct n, simp-all)
apply (subst coeff-eq-0)
apply (auto intro: le-less-trans degree-power-le)
done
lemma degree-linear-power-neg[simp]:
    fixes a :: 'a::{idom,comm-ring-1}
    shows degree ([:a,-1:] ^ n)=n
apply (rule order-antisym)
apply (rule ord-le-eq-trans [OF degree-power-le], simp)
apply (rule le-degree)
unfolding coeff-linear-power-neg
apply (auto)
done
```


### 7.2 Polynomial Composition

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

lemma pcompose-eq-0: fixes $q$ :: ' $a$ :: idom poly
assumes $q$ : degree $q \neq 0$
shows $p \circ_{p} q=0 \longleftrightarrow p=0$
proof (induct p)
case 0
show ?case by auto
next
case ( $p$ Cons a $p$ )
have $i d:(p$ Cons a $p) \circ_{p} q=[: a:]+q *\left(p \circ_{p} q\right)$ by simp
show ? case
proof (cases $p=0$ )
case True
show ?thesis unfolding id unfolding True by simp
next
case False
with $p \operatorname{Cons}(2)$ have $p \circ_{p} q \neq 0$ by auto
from degree-mult-eq[OF-this, of $q] q$ have degree $\left(q *\left(p \circ_{p} q\right)\right) \neq 0$ by force
hence deg: degree $\left([: a:]+q *\left(p \circ_{p} q\right)\right) \neq 0$
by (subst degree-add-eq-right, auto)
show ?thesis unfolding id using False deg by auto
qed
qed
declare degree-pcompose[simp]

### 7.3 Monic Polynomials

```
abbreviation monic where monic p\equiv 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\not=0\veeq\not=0
    shows monic (gcd pq)
proof -
    from assms have 1 = unit-factor (gcd pq) 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\Longrightarrow 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-
poly)
    shows coeff ( }p*q)(\mathrm{ degree }p+\mathrm{ degree }q)=\mathrm{ coeff q (degree q)
proof -
    let ?pqi=\lambda i. coeff pi* coeff q(degree p d degree q - i)
    have coeff (p*q) (degree p + degree q) =
        (\sumi\leqdegree 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\in{.. 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\not=0
    shows degree ( }p*q)=\mathrm{ degree }p+\mathrm{ degree }
proof -
    have degree p+degree q\geq degree ( }p*q)\mathrm{ by (rule degree-mult-le)
    also have degree p+degree q}\leq\mathrm{ degree ( }p*q
    proof -
    from q have cq: coeff q (degree q)}\not=0\mathrm{ by auto
    hence coeff (p*q) (degree p+degree q) =0 unfolding lcoeff-monic-mult[OF
monic] .
    thus degree ( }p*q)\geq\mathrm{ degree }p+\mathrm{ degree q by (rule le-degree)
    qed
    finally show ?thesis .
qed
lemma degree-prod-sum-monic: assumes
    S: finite S
    and nzd: 0 & (degree of)'S
    and monic: (\bigwedge \ . a \inS\Longrightarrow monic (f a))
    shows degree (prod fS)=(sum (degree of)S)^ coeff (prod f S) (sum (degree
of)S)=1
proof -
    from S nzd monic
    have degree (prod fS)= sum (degree ○f)S
    \wedge(S\not={}\longrightarrow degree (prod f S)}\not=0\wedge\operatorname{prod}fS\not=0)\wedge coeff (prod fS) (sum
(degree of)S)=1
    proof (induct S rule: finite-induct)
    case (insert a S)
    have IH1: degree (prod fS)=sum (degree of)S
        using insert by auto
    have IH2:coeff (prod fS) (degree (prod f S)) =1
        using insert by auto
    have id: degree (prod f(insert a S))}=\mathrm{ sum (degree ○f) (insert a S)
        ^coeff (prod f (insert a S)) (sum (degree of) (insert a S))}=
    proof (cases S={})
        case False
        with insert have nz: prod fS\not=0 by auto
        from insert have monic: coeff (f a) (degree (f a)) = 1 by auto
        have id: (degree of) a = degree (f a) by simp
            show ?thesis unfolding prod.insert[OF insert(1-2)] sum.insert[OF in-
sert(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 $\operatorname{simp}$
thus ?thesis by auto
qed
lemma degree-prod-monic:
assumes $\bigwedge i . i<n \Longrightarrow$ degree $\left(f i::{ }^{\prime} a::\right.$ comm-semiring-1 poly $)=1$
and $\wedge i . i<n \Longrightarrow$ coeff $(f i) 1=1$
shows degree $(\operatorname{prod} f\{0 . .<n\})=n \wedge \operatorname{coeff}(\operatorname{prod} f\{0 . .<n\}) n=1$
proof -
from degree-prod-sum-monic $[$ of $\{0 . .<n\} f]$ show ?thesis using assms by force qed
lemma degree-prod-sum-lt-n: assumes $\bigwedge i . i<n \Longrightarrow$ degree ( $f i::$ ' $a$ :: comm-semiring-1
poly) $\leq 1$
and $i$ : $i<n$ and $f$ : degree $(f i)=0$
shows degree $(\operatorname{prod} f\{0 . .<n\})<n$
proof -
have degree $(\operatorname{prod} f\{0 . .<n\}) \leq \operatorname{sum}($ degree of $)\{0 . .<n\}$
by (rule degree-prod-sum-le, auto)
also have sum $($ degree of $)\{0 . .<n\}=($ degree of $) i+$ sum $($ degree of $)(\{0$
$. .<n\}-\{i\})$
by (rule sum.remove, insert $i$, auto)
also have (degree of) $i=0$ using $f i$ by simp
also have sum $($ degree of $)(\{0 . .<n\}-\{i\}) \leq \operatorname{sum}(\lambda-.1)(\{0 . .<n\}-\{i\})$
by (rule sum-mono, insert assms, auto)
also have $\ldots=n-1$ using $i$ by simp
also have $\ldots<n$ using $i$ by simp
finally show? thesis by simp
qed
lemma degree-linear-factors: degree $\left(\prod a \leftarrow a s .[: f a, 1:]\right)=$ length as proof (induct as)
case (Cons $b$ as) note $I H=$ this
have $i d:\left(\prod a \leftarrow b \#\right.$ as. [:f $\left.\left.a, 1:\right]\right)=[: f b, 1:] *\left(\prod a \leftarrow a s\right.$. [:f $\left.\left.a, 1:\right]\right)$ 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 $n z: p \neq 0 q \neq 0$ 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 $n z: p \neq 0 q \neq 0$ 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 $f::{ }^{\prime} a \Rightarrow$ ' $b::$ idom poly
assumes $\bigwedge a . a \in a s \Longrightarrow$ monic $(f a)$
shows monic (prod fas) using assms
proof (induct as rule: infinite-finite-induct)
case (insert a as)
hence $i d$ : prod $f($ insert a as $)=f a * \operatorname{prod} f$ 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 $\bigwedge a . a \in \operatorname{set}$ as $\Longrightarrow$ 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 ( $p^{\text {^ }} n$ )
by (induct $n$, insert assms, auto intro: monic-mult)
lemma monic-prod-list-pow: monic $\left(\prod\left(x::^{\prime} a::\right.\right.$ idom, $\left.i\right) \leftarrow x i s .[:-x, 1:]{ }^{\wedge}$ Suc $\left.i\right)$
proof (rule monic-prod-list, goal-cases)
case (1 a)
then obtain $x i$ where $a: a=[:-x, 1:] \uparrow$ Suc $i$ by force
show monic a unfolding $a$
by (rule monic-power, auto)
qed
lemma monic-degree- $0:$ monic $p \Longrightarrow($ 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 $? p \neq 0 \Longrightarrow$ finite $\left\{x\right.$. poly ? $\left.p x=\left(0:: ?^{\prime} a\right)\right\}$.

```
lemma poly-roots-degree:
    fixes \(p\) :: ' \(a:\) :idom poly
    shows \(p \neq 0 \Longrightarrow\) card \(\{x\). poly \(p x=0\} \leq\) degree \(p\)
proof (induct \(n \equiv\) degree \(p\) arbitrary: \(p\) )
    case ( \(0 p\) )
    then obtain \(a\) where \(a \neq 0\) and \(p=[: a:]\)
        by (cases \(p\), simp split: if-splits)
    then show? case by simp
next
    case (Suc \(n\) p)
    show ? case
    proof (cases \(\exists x\). poly \(p x=0\) )
        case True
        then obtain \(a\) where \(a\) : poly p \(a=0\)..
        then have \([:-a, 1:] d v d p\) by (simp only: poly-eq-0-iff-dvd)
    then obtain \(k\) where \(k: p=[:-a, 1:] * k\)..
    with \(\langle p \neq 0\rangle\) have \(k \neq 0\) by auto
    with \(k\) have degree \(p=\operatorname{Suc}\) (degree \(k\) )
        by (simp add: degree-mult-eq del: mult-pCons-left)
    with \(\langle\) Suc \(n=\) degree \(p\rangle\) have \(n=\) degree \(k\) by simp
    from Suc.hyps (1)[OF this \(\langle k \neq 0\rangle\) ]
    have le: card \(\{x\). poly \(k x=0\} \leq\) 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 \(\ldots \leq \operatorname{Suc}\) (card \(\{x\). poly \(k x=0\}\) )
            unfolding card-insert-if[OF poly-roots-finite[OF \(\langle k \neq 0\rangle]]\) by simp
    also have \(\ldots \leq\) Suc (degree \(k\) ) using le by auto
    finally show ?thesis using 〈degree \(p=\) Suc (degree \(k\) ) 〉 by simp
    qed \(\operatorname{simp}\)
qed
```

lemma poly-root-factor: $\left(\right.$ poly $\left.([: r, 1:] * q)\left(k::{ }^{\prime} a:: i d o m\right)=0\right)=(k=-r \vee$
poly $q k=0$ ) (is ?one)
$($ poly $(q *[: r, 1:]) k=0)=(k=-r \vee$ poly $q k=0)($ is ?two $)$
(poly [: $r, 1:] k=0)=(k=-r)$ (is ?three)
proof -
have $[$ simp $]: r+k=0 \Longrightarrow k=-r$ 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: $c \neq 0 \Longrightarrow(p o l y(p *[: c:])(k:: ' a ~::$ idom $)=0)=$

```
(poly p k=0)
    unfolding poly-mult by auto
lemma poly-linear-exp-linear-factors-rev:
    ([:b,1:])^(length (filter ((=)b) as)) dvd (П (a ::' 'a :: comm-ring-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\leftarrowCons a as. [:a, 1:]) = [:a, 1:] * (П a\leftarrowas. [: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 :] ^ ?ls = [: a, 1 :]*[:b, 1 :] ^ ?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 :] ^ k dvd p and p:p\not=0
    shows k\leq order a p
proof (rule ccontr)
    assume \neg? ?thesis
    hence }\existsj.k=Suc (order a p+j) by arith
    then obtain j where k: k=Suc (order a p +j) by auto
    have [:-a,1 :] ^ 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
$d v d:[: b, 1:] \operatorname{dvd}\left(\prod(a:: ' a) \leftarrow a s .[: a, 1:]\right)$
shows $b \in$ set as
proof -
let $? p=\lambda a s .\left(\prod a \leftarrow a s .[: a, 1:]\right)$
let $? b=[: b, 1:]$
from assms[unfolded dvd-def] obtain $p$ where $i d: ? p$ as $=? b * p .$.

```
    from arg-cong[OF id, of \lambda 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\vee (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 (\prod(a::' 'a)\leftarrowas. [: a, 1:])
    shows length (filter ((=) b) as) \geqn
proof -
    let ?p = \lambdaas. (П a\leftarrowas. [: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:]*\ldots=[: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 \lambda 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 \longleftrightarrowa dvd 1
    by (metis const-poly-dvd one-poly-eq-simps(2))
lemma poly-dvd-1:
    fixes p :: ' }a\mathrm{ :: {comm-semiring-1,semiring-no-zero-divisors} poly
    shows p dvd 1 \longleftrightarrow degree p=0^ 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
    from degree0-coeffs[OF this] obtain a where p: p=[:a:] by auto
    show ?thesis unfolding p by auto
qed
    Degree based version of irreducibility.
definition irreducible ed :: 'a :: comm-semiring-1 poly }=>\mathrm{ bool where
    irreducible e}p=(\mathrm{ degree }p>0\wedge(\forallqr\mathrm{ . degree }q<\mathrm{ degree }p\longrightarrow\mathrm{ degree }r
degree p\longrightarrowp\not=q*r))
lemma irreducible d I [intro]:
    assumes 1: degree p>0
    and 2: \qr.degree q>0\Longrightarrow degree q< degree p\Longrightarrow degree r>0\Longrightarrowdegree
r< degree p\Longrightarrowp=q*r\Longrightarrow False
    shows irreducible, p
proof (unfold irreducible d
    fix qr
    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 }\mp@subsup{\mp@code{d}}{\mathrm{ 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\mathrm{ degree
p div 2 \Longrightarrow ᄀq dvd p
    shows irreducible d p
proof (rule ccontr)
    assume \neg? ?thesis
    from this[unfolded irreducible e}\mp@subsup{d}{}{-}\mathrm{ def] deg obtain qr where dq: degree q< degree
p and dr: degree r < degree p
```

and $p: p=q * r$ by auto
from deg have $p 0: p \neq 0$ by auto
with $p$ have $q \neq 0 r \neq 0$ by auto
from degree-mult-eq[OF this] $p$ have dp: degree $p=$ degree $q+$ degree $r$ by simp
show False
proof (cases degree $q \leq$ degree $p$ div 2)
case True
from $n d v d[O F-T r u e] d q d r d p p$ show False by auto
next
case False
with $d p$ have $d r$ : degree $r \leq$ degree $p$ div 2 by auto
from $p$ have $d v d: r d v d p$ by auto
from $n d v d[O F-d r] d v d d p d q$ show False by auto
qed
qed
lemma reducible ${ }_{d} I$ :
assumes degree $p>0 \Longrightarrow \exists q r$. degree $q<$ degree $p \wedge$ degree $r<$ degree $p \wedge p$
$=q * r$
shows $\neg$ irreducible $_{d} p$
using assms by (auto simp: irreducible $_{d}$-def)
lemma irreducible $_{d} E[$ elim] :
assumes irreducible $_{d} p$
and degree $p>0 \Longrightarrow(\bigwedge q r$. degree $q<$ degree $p \Longrightarrow$ degree $r<$ degree $p \Longrightarrow$ $p \neq q * r) \Longrightarrow$ thesis
shows thesis
using assms by (auto simp: irreducible $d_{d}$-def)
lemma reducible ${ }_{d} E[$ elim]:
assumes red: $\neg$ irreducible $_{d} p$
and 1: degree $p=0 \Longrightarrow$ thesis
and 2: $\bigwedge q r$. degree $q>0 \Longrightarrow$ degree $q<$ degree $p \Longrightarrow$ degree $r>0 \Longrightarrow$ degree
$r<$ degree $p \Longrightarrow p=q * r \Longrightarrow$ thesis
shows thesis
using red[unfolded irreducible $d_{d^{-}}$def de-Morgan-conj not-not not-all not-imp]
proof (elim disjE exE conjE)
show $\neg$ degree $p>0 \Longrightarrow$ 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 $p>0 \bigwedge q r$. degree $q<$ degree $p \Longrightarrow$ degree $r<$ degree $p \Longrightarrow p \neq$ $q * r$
using assms unfolding irreducible $_{d^{-}}$def by auto
theorem irreducible $_{d}$-factorization-exists:
assumes degree $p>0$
shows $\exists$ fs. $f s \neq[] \wedge\left(\forall f \in\right.$ set fs. irreducible $e_{d} f \wedge$ degree $f \leq$ degree $\left.p\right) \wedge p=$ prod-list fs
and $\neg$ irreducible $_{d} p \Longrightarrow \exists$ s. length $f s>1 \wedge\left(\forall f \in\right.$ set fs. irreducible ${ }_{d} f \wedge$ degree $f<$ degree $p) \wedge p=$ prod-list $f_{s}$
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 $\left(\right.$ cases irreducible $\left.d_{d} p\right)$
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 $[O F$ deg-g] less.hyps $[O F$ deg-h]
obtain $g s h s$
where emp: length gs $>0$ length $h s>0$
and $\forall f \in$ set gs. irreducible $d_{d} f \wedge$ degree $f \leq$ degree $g g=$ prod-list gs and $\forall f \in$ set hs. irreducible $e_{d} f \wedge$ degree $f \leq$ degree $h h=$ prod-list $h$ s by auto with $f$-gh deg-g deg-h
have len: length $(g s @ h s)>1$
and mem: $\forall f \in \operatorname{set}(g s @ h s)$. irreducible $d f \wedge$ degree $f<$ degree $p$
and $p: p=p r o d-l i s t ~(g s @ h s)$ 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 st where IH: irreducible e}s sq=s*t\mathrm{ by auto
    from p have p:p=s*(t*r) unfolding IH by (simp add: ac-simps)
    from less(2) have p}\not=0\mathrm{ by auto
    hence degree p=degree s+(degree (t*r)) unfolding p
            by (subst degree-mult-eq, insert p, auto)
    with irreducible }D[\mp@code{OF IH(1)] have degree p> degree ( }t*r\mathrm{ ) 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\equiv\existsb.b\not=0\wedgea*b=
lemma zero-divisorI[intro]:
    assumes }b\not=0\mathrm{ and }a*b=0\mathrm{ shows zero-divisor a
    using assms by (auto simp: zero-divisor-def)
lemma zero-divisorE[elim]:
    assumes zero-divisor a
        and \}\b.b\not=0\Longrightarrowa*b=0\Longrightarrowthesi
    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:
    \neg 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\longleftrightarrowa=0\mathrm{ by auto
lemma mult-eq-0-not-zero-divisor-left[simp]:
    fixes a b :: 'a :: mult-zero
    assumes \neg zero-divisor a
    shows }a*b=0\longleftrightarrowb=
    using assms unfolding zero-divisor-def by force
lemma mult-eq-0-not-zero-divisor-right[simp]:
```

```
    fixes a b :: ' }a\mathrm{ :: {ab-semigroup-mult,mult-zero}
    assumes \neg zero-divisor b
    shows }a*b=0\longleftrightarrowa=
    using assms unfolding zero-divisor-def by (force simp: ac-simps)
lemma degree-smult-not-zero-divisor-left[simp]:
    assumes ᄀ zero-divisor c
    shows degree (smult c p)= degree p
proof(cases p=0)
    case False
    then have coeff (smult c p) (degree p) \not=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 \neg 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: ᄀ zero-divisor c
    assumes L: irreducible ( smult c p)
    shows irreducible d}
proof (intro irreducible d}I\mathrm{ )
    from L have degree (smult c p)>0 by auto
    also note degree-smult-le
    finally show degree p>0 by auto
    fix }q
    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<\ldots. using c0 by auto
    from irreducible dD(2)[OF L 2 3] 1 show False by auto
qed
lemmas irreducible d}\mp@subsup{\mp@code{-smultI =}}{}{\prime
    irreducible d
    [where ' }a='='a::{comm-semiring-1,semiring-no-zero-divisors}, simplified]
```

```
lemma irreducible e-smult-not-zero-divisor-right:
    assumes p0:\neg zero-divisor (lead-coeff p) and L: irreducible (smult c p)
    shows \mp@subsup{\mathrm{ irreducible }}{d}{}p
proof-
    from L have c\not=0 by auto
    with p0 have [simp]: degree (smult c p) = degree p by simp
    show irreducible e}
    proof (intro iffI irreducible d I conjI)
        from L show degree p>0 by auto
        fix qr
        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<\ldots. by simp
        from irreducible }D\mathrm{ D(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\not=0 and [simp]:a*c=0 by auto
    have }a*b*c=a*c*b\mathrm{ 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\mathrm{ )
proof-
    from assms obtain c where c0:c\not=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 \neg zero-divisor ( }a*b\mathrm{ )
    shows }\neg\mathrm{ zero-divisor a and }\neg\mathrm{ 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\not=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 \negzero-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-irreducibled}\mp@subsup{}{d}{}\mathrm{ : assumes degree p=1
    shows \mp@subsup{\mathrm{ irreducible }}{d}{}p
    by (rule irreducible d}I\mathrm{ , insert assms, auto)
lemma irreducible d
    fixes p ::'a::{comm-semiring-1,semiring-no-zero-divisors} poly
    assumes degree p>0 irreducible d q p dvd q
    shows }\existsc.c\not=0\wedgeq= smult c
proof -
    from assms obtain r where q: q= p*r by (elim dvdE, auto)
    from degree-mult-eq[of pr] assms(1) q
    have }\neg\mathrm{ degree }p<\mathrm{ degree q and nz: p}=0q\not=
            apply (metis assms(2) degree-mult-eq-0 gr-implies-not-zero irreducible dD(2)
less-add-same-cancel2)
            using assms by auto
    hence deg: degree p degree q by auto
    from \langlep dvd q\rangle obtain k where q: q=k*p unfolding dvd-def by (auto simp:
ac-simps)
    with nz have k\not=0 by auto
    from deg[unfolded q degree-mult-eq[OF \langlek\not=0\rangle\langlep\not=0\rangle]] have degree k=0
            unfolding q by auto
    then obtain c where k:k=[:c:] by (metis degree-0-id)
    with }\langlek\not=0\rangle have c\not=0 by aut
    have q= smult c p unfolding qk by simp
    with }\langlec\not=0\rangle\mathrm{ 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\not=0\vee p\not=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: f0=0
    shows map-poly f(map-poly g p)= map-poly (f\circg)p
proof (induct p)
    case (pCons a p) show ?case
    proof(cases g a\not=0\vee map-poly g p}\not=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:\forallc.fc=0\longrightarrowc=0
    shows [simp]: map-poly f p=0 \longleftrightarrowp=0
    by (induct p; auto simp: map-poly-simps f)
lemma map-poly-add:
    assumes h0:h 0=0
            and h-add: }\forallp\mathrm{ q. }h(p+q)=hp+h
    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 $p$ Cons ( $0::^{\prime} a$ )

lemma monom-pCons-0-monom:
monom ( $p$ Cons 0 (monom a $n$ )) $d=$ map-poly ( $p$ Cons 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 $p$ Cons- 0 -add: $p$ Cons $0(p+q)=p$ Cons $0 p+p$ Cons $0 q$ by auto
lemma sum-pCons-0-commute:
sum ( $\lambda$ i. pCons $0(f i)) S=p$ Cons $0(\operatorname{sum} f S)$
by (induct $S$ rule: infinite-finite-induct;simp)
lemma pCons-0-as-mult:
fixes $p$ :: ' $a$ :: comm-semiring-1 poly
shows $p$ Cons $0 p=[: 0,1:] * p$ by auto

### 7.8 Misc

fun expand-powers $::($ nat $\times$ ' $a)$ list $\Rightarrow$ ' $a$ list where
expand-powers [] = []
| expand-powers $((S u c ~ n, a) \# p s)=a \#$ expand-powers $((n, a) \# p s)$
| expand-powers $((0, a) \# p s)=$ expand-powers $p s$
lemma expand-powers: fixes $f:: ' a \Rightarrow$ ' $b::$ comm-ring-1
shows $\left(\prod(n, a) \leftarrow n\right.$-as. f $\left.a^{\wedge} n\right)=\left(\prod a \leftarrow\right.$ expand-powers $n$-as. $\left.f a\right)$
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) $x=0)=(a=0 \vee$ poly $p x=0)$
by simp
lemma poly-prod-list-zero-iff: fixes $x$ :: ' $a$ :: idom
shows (poly (prod-list ps) $x=0)=(\exists p \in$ set ps. poly $p x=0)$

```
    by (induct ps,auto)
lemma poly-mult-zero-iff: fixes }x\mathrm{ :: ' }a\mathrm{ :: idom
    shows (poly (p*q)x=0)=(poly px=0\vee poly qx=0)
    by simp
lemma poly-power-zero-iff: fixes }x:: ' a :: idom
    shows (poly ( 
    by (cases n, auto)
lemma sum-monom-0-iff: assumes fin: finite S
    and g: \bigwedgeij.gi=gj\Longrightarrowi=j
    shows sum (\lambda i. monom (fi) (gi))S=0\longleftrightarrow(\foralli\inS.fi=0) (is ?l = ?r)
proof -
    {
        assume }\neg\mathrm{ ? }
        then obtain }i\mathrm{ where }i:i\inS\mathrm{ and fi:fi}=0\mathrm{ by auto
        let ? g=\lambdai.monom (fi)(gi)
        have coeff (sum?g S) (gi) = fi + sum (\lambda j. coeff (?g j) (gi)) (S - {i})
            by (unfold sum.remove[OF fin i], simp add: coeff-sum)
        also have sum (\lambda j. coeff (?g j) (gi)) (S - {i}) = 0
            by (rule sum.neutral, insert g, auto)
        finally have coeff (sum ?g S)(gi)}\not=0\mathrm{ using fi by auto
        hence }\neg\mathrm{ ?l by auto
    }
    thus ?thesis by auto
qed
lemma degree-prod-list-eq: assumes }\bigwedgep.p\in\operatorname{set}ps\Longrightarrow(p:: 'a a: idom poly)\not=
    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\not=0
    shows degree ( }\mp@subsup{p}{}{`}n)=\mathrm{ degree ( }p:: 'a :: idom poly)*
proof (induct n)
    case (Suc n)
    from p have pn: p^ n\not=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\not=0^(\foralla. order a p \leq 1))
```

```
proof -
    have }\bigwedgea. order a p\leq1\longleftrightarrow order a p=0 \vee order a p=1 by linarith
    thus ?thesis unfolding rsquarefree-def by auto
qed
lemma order-prod-list: (\bigwedge p.p set ps \Longrightarrowp\not=0)\Longrightarroworder 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 }\mp@subsup{e}{d}{}-dvd-eq
    fixes a b ::'a::{comm-semiring-1,semiring-no-zero-divisors} poly
    assumes irreducible e}d\mathrm{ a and irreducible }\mp@subsup{|}{d}{}
        and a dvd b
    and monic a and monic b
    shows }a=
    using assms
    by (metis (no-types, lifting) coeff-smult degree-smult-eq irreducible }D\mathrm{ D(1) irre-
ducible }\mp@subsup{d}{d}{}\mathrm{ -dvd-smult
    mult.right-neutral smult-1-left)
lemma monic-gcd-dvd:
    assumes fg: fdvd g and mon: monic f and gcd: gcd gh}\{{1,g
    shows gcd f h\in{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}=
        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)\leqdegree f - 1
proof (rule ccontr)
    assume ᄀ?thesis
    hence ge:degree (pderiv f) \geqSuc (degree f - 1) by auto
    hence pderiv f}\not=0\mathrm{ by auto
    hence coeff (pderiv f) (degree (pderiv f)) \not=0 by auto
    from this[unfolded coeff-pderiv]
    have coeff f (Suc (degree (pderiv f))) \not=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 ( }\lambdax.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\mathrm{ by auto
    have degree: degree p= degree q
        using \langlepdvd q>\langleqdvd p\rangle\langlep\not=0\rangle\langleq\not=0\rangle
        by (intro order-antisym dvd-imp-degree-le)
    from }\langlepdvdq\rangle\mathrm{ obtain a where a: q=p*a..
    with }\langleq\not=0\rangle\mathrm{ have }a\not=0\mathrm{ by auto
    with degree a <p\not=0\rangle 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 []$\Rightarrow$ True $\mid$ - $\Rightarrow$ 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 $x s=$ coeffs $f$ in if $x s=[]$ then 0 else last xs $)$
by (simp add: last-coeffs-eq-coeff-degree)
lemma nth-coeffs-coeff: $i<$ length (coeffs $f) \Longrightarrow$ coeffs $f!i=$ coeff $f i$ by (metis nth-default-coeffs-eq nth-default-def)
definition monom-mult :: nat $\Rightarrow$ ' $a$ :: comm-semiring-1 poly $\Rightarrow$ '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 $x s=$ coeffs $f$ in
if $x s=[]$ then $x s$ else replicate $n 0 @ x s)$
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 $f::{ }^{\prime} a$ :: comm-ring-1 poly assumes $n: j<n$
shows coeff $\left(f \circ_{p}\right.$ monom $\left.1 n\right)(n * i+j)=($ if $j=0$ then coeff $f i$ else 0$)$
proof (induct $f$ arbitrary: $i$ )
case ( $p$ Cons 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 $i d$ : $n * S u c i i+j-n=n * i i+j$ using $n$ by (simp add: diff-mult-distrib2)
have id1: $(n \leq n * S u c i i+j)=$ True by auto
have id2: $($ case $n * S u c i i+j$ of $0 \Rightarrow a \mid$ Suc $x \Rightarrow$ coeff $0 x)=0$ using $n$ by (cases $n * S u c i i+j$, 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\not=0
    shows coeff (f 牛 monom 1 n) ( n*i)= coeff fi
    using coeff-pcompose-monom[of 0 nfi] n by auto
lemma dvd-dvd-smult: a dvd b\Longrightarrowfdvd 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\mathrm{ :: idom-divide poly }=>\mp@subsup{}{}{\prime}a=>\mp@subsup{|}{}{\prime}a poly wher
    sdiv-poly p a = (map-poly (\lambda 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 }\Lambdac.c\in\operatorname{set}(\mathrm{ coeffs p) }\Longrightarrowadvd 
    shows smult a (sdiv-poly pa) = 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}=
    shows \existsn.\forallx\geqn. poly p x \geqb
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 }\x.x\geqn\Longrightarrow0\leq poly p x - b by aut
    thus ?thesis by auto
qed
lemma pderiv-sum: pderiv (sum fI) = sum (\lambda i.(pderiv (f i))) I
    by (induct I rule: infinite-finite-induct, auto simp: pderiv-add)
lemma smult-sum2: smult m (\sumi\inS.fi)=(\sumi\inS.smult m (fi))
    by (induct S rule: infinite-finite-induct, auto simp add: smult-add-right)
lemma degree-mult-not-eq:
    degree }(f*g)\not=\mathrm{ degree }f+\mathrm{ degree }g\Longrightarrow\mathrm{ lead-coeff }f*\mathrm{ lead-coeff g}=
    by (rule ccontr, auto simp: coeff-mult-degree-sum degree-mult-le le-antisym le-degree)
lemma irreducible }\mp@subsup{d}{d}{}-multD
    fixes }ab\mathrm{ :: 'a :: {comm-semiring-1,semiring-no-zero-divisors} poly
```

```
    assumes l: irreducible e}(a*b
    shows degree a=0^a\not=0^\mp@subsup{\mathrm{ irreducible }}{d}{}b\vee\mathrm{ degree b=0^b}=0\wedge
irreducible }\mp@subsup{|}{|}{}
proof-
    from l have a0:a\not=0 and b0:b\not=0 by auto
    note [simp] = degree-mult-eq[OF this]
    from l have degree a=0 \vee 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}
            by (simp add: irreducible d}\mp@subsup{|}{d}{}\mathrm{ -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}\mp@subsup{|}{}{-def
            by (smt add-cancel-left-right degree-mult-eq degree-mult-eq-0 neq0-conv semir-
ing-normalization-rules(16))
    with b b0 show ?thesis by auto
    qed
qed
lemma irreducible-connect-field[simp]:
    fixes f :: 'a :: field poly
    shows irreducible e}f=\mathrm{ irreducible f(is ?l = ?r)
proof
    show ?r \Longrightarrow ?l
            apply (intro irreducible }\mp@subsup{|}{d}{}\mathrm{ 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\not=0 \neg is-unit f by (auto simp: poly-dvd-1)
            fix ab assume f=a*b
            from l[unfolded this]
    show a dvd 1 \vee 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\longleftrightarrowp\not=0^ degree p=0
    by (cases p=0, auto simp: is-unit-iff-degree)
lemma irreducible-smult-field[simp]:
```

```
    fixes c :: ' }a\mathrm{ :: field
    shows irreducible (smult c p) \longleftrightarrowc\not=0\wedge irreducible p (is ?L \longleftrightarrow?R)
proof (intro iffI conjI irreducible d-smult-not-zero-divisor-left[of c p, simplified])
    assume irreducible (smult c p)
    then show c\not=0 by auto
next
    assume ?R
    then have c0:c\not=0 and irr: irreducible p by auto
    show ?L
    proof (fold irreducible-connect-field, intro irreducible }\mp@subsup{|}{d}{}\mathrm{ , unfold degree-smult-eq
if-not-P[OFc0])
    show degree p>0 using irr by auto
    fix }q
    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) \ldots..
    assume main: degree q< degree p degree r < degree p
    have }\neg\mp@subsup{\mathrm{ irreducible }}{d}{}p\mathrm{ by (rule reducible }\mp@subsup{|}{d}{}I\mathrm{ , 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 \existsqr. irreducible q\wedgep=q*r^ monic q
proof -
    from irreducible d}\mp@subsup{\mathrm{ -factorization-exists[OF assms]}}{}{\prime
    obtain fs where fs =[] and set fs\subseteqCollect 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\not=0 unfolding c-def irreducible d
    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\mathrm{ :: field poly
    shows monic p\Longrightarrow
    \exists as f. finite as }\wedge p=prod (\lambda a. a^ Suc (fa)) as ^ as\subseteq{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 \(q 0: q \neq 0\) by auto
    define \(c\) where \(c=\) coeff \(q\) (degree \(q\) )
    let ? \(q=\operatorname{smult}(1 / c) q\)
    let \(? r=\) smult \(c r\)
    from \(q 0\) have \(c: c \neq 01 / c \neq 0\) 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 \(q \wedge\) monic ( \(q::\) 'a poly) \(\}\)
    have mon: monic ? \(q\) unfolding \(c\)-def using \(q 0\) by auto
    from monic-factor \([O F\langle m o n i c ~ p\rangle[\) unfolded \(p]\) this] have monic ? \(r\).
    from less (1)[OF deg this] obtain \(f\) as
    where as: finite as ? \(r=\left(\prod a \in a s . a^{\wedge} S u c(f a)\right)\)
        as \(\subseteq ? Q\) by blast
    from \(q c\) have irred: irreducible ? \(q\) by simp
    show ?thesis
    proof (cases ? \(q \in a s\) )
        case False
        let ?as \(=\) insert \(? q\) as
        let ?f \(=\lambda a\). if \(a=\) ? \(q\) then 0 else \(f a\)
        have \(p=? q *\left(\Pi a \in a s . a{ }^{\wedge} S u c(f a)\right)\) unfolding \(p\) as by simp
        also have (Пa<as. \(\left.a^{\wedge} \operatorname{Suc}(f a)\right)=\left(\prod a \in a s . a{ }^{\wedge} S u c(? f a)\right)\)
        by (rule prod.cong, insert False, auto)
    also have ? \(q * \ldots=\left(\prod a \in\right.\) ? as. \(\left.a^{\wedge} S u c(? f a)\right)\)
        by (subst prod.insert, insert as False, auto)
    finally have \(p: p=\left(\prod a \in\right.\) ? as. \(\left.a^{\wedge} S u c(? f a)\right)\).
    from \(a s(1)\) have fin: finite? as by auto
    from as mon irred have \(Q: ?\) as \(\subseteq ? Q\) by auto
    from fin \(p Q\) show ?thesis
        by (intro exI[of - ?as \(]\) exI [of - ?f], auto)
    next
    case True
    let ?f \(=\lambda a\). if \(a=\) ? \(q\) then Suc \((f a)\) else \(f a\)
    have \(p=? q *\left(\prod a \in a s . a{ }^{\wedge} S u c(f a)\right)\) unfolding \(p\) as by simp
    also have \(\left(\prod a \in a s . a^{\wedge} S u c(f a)\right)=? q \wedge S u c(f ? q) *\left(\prod a \in(a s-\{? q\})\right.\).
\(\left.a^{\wedge} S u c(f a)\right)\)
        by (subst prod.remove[OF - True], insert as, auto)
    also have (П \(\left.a \in(a s-\{? q\}) . a^{\wedge} S u c(f a)\right)=\left(\prod a \in(a s-\{? q\}) . a^{\wedge} S u c\right.\)
(?f a))
    by (rule prod.cong, auto)
    also have \(? q *(? q \wedge \operatorname{Suc}(f ? q) * \ldots)=? q \wedge \operatorname{Suc}(? f ? q) * \ldots\)
        by (simp add: ac-simps)
        also have \(\ldots=\left(\prod a \in a s . a{ }^{\wedge}\right.\) Suc (?f \(\left.a\right)\) )
            by (subst prod.remove[OF - True], insert as, auto)
```

```
        finally have p=(\proda\inas.a^ Suc (?f a)).
        with as show ?thesis
        by (intro exI[of - as] exI[of - ?f],auto)
        qed
    qed
qed
lemma monic-irreducible-gcd:
monic ( \(f:\) :' \(^{\prime}\) :::\{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<br>imports<br>HOL-Computational-Algebra.Euclidean-Algorithm<br>Ring-Hom<br>Missing-Polynomial<br>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 pa..
interpretation poly-hom: idom-hom $\lambda$ p. poly $p$ a.. $\left(\circ_{p}\right)$ 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 :: (' $a \neq$ ' $b::$ comm-semiring-1) $\Rightarrow^{\prime} a::$ zero poly $\Rightarrow^{\prime} b \Rightarrow^{\prime} b$ where
[code del]: eval-poly $h p=$ poly (map-poly $h p$ )
lemma eval-poly-code[code]: eval-poly $h$ p $x=$ fold-coeffs $(\lambda a b . h a+x * b) p 0$

```
    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 = (\sumi\leqdegree p. x`i*h(coeff pi))
    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 11
    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 (\lambda a.[: ha:]) p [:0,1:] (is ?mp = ?ep)
proof (rule poly-eqI)
    fix i :: nat
    have 2:(\sumx\leqi. \sumxa\leqdegree 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}+\mathrm{ 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=(\sumxa\leqdegree 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\leq 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 \lambdaa.[:ha 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 pi) 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)=\mathrm{ 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 pq show map-poly hom ( }p*q)=\mathrm{ 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 $\lambda p$. coeff $p-$, OF refl this] show $p=q$ by (auto intro: poly-eqI)
qed $\operatorname{simp}$
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 $p$ ) $\leq$ degree $p$
by (induct $p ;$ auto)
lemma coeffs-map-poly:
assumes $f$ (lead-coeff $p)=0 \longleftrightarrow p=0$
shows coeffs (map-poly f $p$ ) $=\operatorname{map} f$ (coeffs $p$ )
unfolding coeffs-map-poly using assms by (simp add:coeffs-def)
lemma degree-map-poly:
assumes $f$ (lead-coeff $p)=0 \longleftrightarrow p=0$
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 $\operatorname{simp}$
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 $p) 1=\operatorname{hom}($ 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:
$\left(\sum j \leq\right.$ degree $p$. monom $($ hom $($ coeff $\left.p j)) j\right)=$ 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 $\left(f \circ_{p} g\right)=$ 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- $O[$ 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 $a * x \wedge n$ 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 $(\lambda a$. [: hom a :]) $p[: 0,1:]$
by (rule map-poly-eval-poly, simp)
lemma degree-extension: assumes degree $p \leq n$

```
    shows (\sumi\leqdegree p. x^ i * hom (coeff p i))
        =(\sumi\leqn. x^ i* hom (coeff pi))(is ?l = ?r)
proof -
    let ?f = \lambdai. x^ i* hom (coeff pi)
    define m}\mathrm{ where m=n- degree p
    have n: n = degree p+m unfolding m-def using assms by auto
    have ?r = (\sum i\leq 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=\mathrm{ eval-poly hom p x +
eval-poly hom q x
    unfolding eval-poly-def hom-distribs..
lemma eval-poly-sum: eval-poly hom (\sumk\inA.pk)x=(\sumk\inA. eval-poly hom ( }
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
context field-hom
begin
lemma dvd-map-poly-hom-imp-dvd:«map-poly hom x dvd map-poly hom y }\Longrightarrow
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\operatorname{div}q),?mp (p\operatorname{mod}q))\rangle
    proof (induction rule: euclidean-relation-polyI)
        case by0
        then show ?case
            by simp
    next
        case divides
        then have }\langleq\not=0\rangle\langleq dvd p
            by (auto dest:dvd-map-poly-hom-imp-dvd)
        from }\langleqdvd p\rangle\mathrm{ obtain }r\mathrm{ where }\langlep=q*r\rangle.
        with }\langleq\not=0\rangle\mathrm{ 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 $p \quad q]$ by simp
lemma map-poly-mod[hom-distribs]: map-poly hom $(p \bmod q)=$ map-poly hom $p$ mod map-poly hom $q$
using map-poly-pdivmod[of $p q$ ] by simp
end
locale field-hom ${ }^{\prime}=$ field-hom hom for hom :: ' $a::\{$ field-gcd $\} \Rightarrow{ }^{\prime} b::\{$ 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 pq)=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 $\Rightarrow$ ' $a$ poly $\Rightarrow$ ' $a$ poly where
div-poly a $p=$ map-poly $(\lambda c . c$ div a) $p$
lemma smult-div-poly: assumes $\wedge c . c \in \operatorname{set}($ coeffs $p) \Longrightarrow a d v d c$ 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) $n=\operatorname{coeff} f n$ 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 $h h \equiv$ map-poly hom
shows $h h$ (divide-poly-main lc fghij)=divide-poly-main (hom lc) (hhf) (hh
g) $(h h h) i j$
unfolding $h h$-def
proof (induct $j$ arbitrary: lc $f g h i$ )
case (Suc jlc fghi)

```
    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 $\equiv$ 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 $\Rightarrow{ }^{\prime} a::\{$ comm-ring-1,ring-char-0\} ..
interpretation of-int-poly-hom:
map-poly-inj-idom-hom of-int :: int $\Rightarrow^{\prime} a::\{$ idom,ring-char-0 $\} ..$
The following operations are homomorphic w.r.t. only monoid-add.
interpretation $p$ Cons- 0 -hom: injective pCons 0 by (unfold-locales, auto)
interpretation pCons-0-hom: zero-hom-0 pCons 0 by (unfold-locales, auto)
interpretation $p$ Cons-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 $\lambda$ x. monom $x d$ by (unfold-locales, auto)
interpretation monom-hom: inj-monoid-add-hom $\lambda x$. monom x $d$ by (unfold-locales, auto simp: add-monom)
interpretation monom-hom: inj-comm-monoid-add-hom $\lambda x$. 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 $\left(x_{1}, p\left(x_{1}\right)\right),\left(x_{2}, p\left(x_{2}\right)\right), \ldots$.. 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 \(\Rightarrow\) ' \(a\) list \(\Rightarrow\) 'a poly where
    horner-composition [cn] xis \(=\) [:cn:]
\(\mid\) horner-composition \((c i \# c s)(x i \#\) xis \()=\) horner-composition cs xis * [:- xi, 1:]
\(+[: c i:]\)
| 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 \(c s \leq\) Suc (length ys)
    shows (set (coeffs (horner-composition cs (map rat-of-int ys))) \(\subseteq \mathbb{Z})=(\) set cs
\(\subseteq \mathbb{Z}\) )
proof -
    let \(?\) ir \(=\) int-of-rat
    let ?ri = rat-of-int
    let ?mir = map ? ir
    let ? mri = map ? \(r i\)
    show?thesis
    proof
        define \(i c s\) where \(i c s=\) map ?ir cs
        assume set \(c s \subseteq \mathbb{Z}\)
        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) \()\) ) \(\subseteq \mathbb{Z}\)
            unfolding ics of-int-poly-hom.horner-composition-hom by auto
    next
        assume set (coeffs (horner-composition cs \((\) ?mri ys) \()\) ) \(\subseteq \mathbb{Z}\)
```

```
thus set cs \subseteq\mathbb{Z}\mathrm{ using len}
proof (induct cs arbitrary: ys)
    case (Cons c cs xs)
    show ?case
    proof (cases cs=[] \vee 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}\mathrm{ where }q=
    define p}\mathrm{ 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\in\mathbb{Z}\mathrm{ 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 }
        let ?f = \lambda 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 ... =(\sumj\leqSuc i. ?f j) unfolding coeff-mult by simp
    also have ... = ?f 0 + ?f 1 + (\sumj\in{..Suc i} - {0} - {Suc 0}. ?f j)
            by (subst sum.remove[of-0], force+, subst sum.remove[of-1], force+)
        also have (\sumj\in{..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 qi by simp
```



```
i] by auto
    assume coeff q(Suc i)\in\mathbb{Z}
    hence ?ri y* coeff q(Suc i)\in\mathbb{Z by simp}
    hence coeff q}i\in\mathbb{Z}\mathrm{ using int Ints-diff Ints-minus by force
    } note coeff-q = this
    {
    fix }
    assume i\leq degree q
```

```
            hence coeff \(q(\) degree \(q-i) \in \mathbb{Z}\)
            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 \(q i \in \mathbb{Z}\)
            proof (cases \(i \leq\) 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 \(q\) ) \(\subseteq \mathbb{Z}\) by (auto simp: coeffs-def)
                            from Cons(1)[OF this[unfolded \(q\)-def]] Cons(3) xs have \(I H\) : set \(c s \subseteq \mathbb{Z}\) by
auto
            define \(r\) where \(r=\) coeff \(q 0 *(-\) ?ri \(y)\)
            have \(r: r \in \mathbb{Z}\) using coeff- \(q[\) of 0\(]\) unfolding \(r\)-def by auto
            have coeff \(p 0 \in \mathbb{Z}\) by fact
            also have coeff \(p 0=r+c\) unfolding \(p\)-def \(r\)-def by simp
            finally have \(c: c \in \mathbb{Z}\) using \(r\) using Ints-diff by force
            with \(I H\) show ?thesis by auto
        qed
    qed \(\operatorname{simp}\)
    qed
qed
context
fixes
    ty :: 'a :: field itself
    and \(x s::\) ' \(a\) list
    and \(f s::\) 'a list
begin
fun divided-differences-impl :: 'a list \(\Rightarrow{ }^{\prime} a \Rightarrow{ }^{\prime} a \Rightarrow\) ' \(a\) list \(\Rightarrow\) ' \(a\) 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 }=>\mathrm{ ' 'a list }=>\mathrm{ ' '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 xi= xs!i
qualified definition fi=fs!i
private definition xd ij=xi-xj
lemma [simp]: xd i i= 0 xd ij + xd jk=xd ikxd ij + xd ki=xd kj
    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 ( }\lambda(i,j).j-i),auto
private definition c:: nat => ' }a\mathrm{ where
    ci= xij-f 0 i
private definition X j=[:-xj,1:]
private function b :: nat => nat }=>\mathrm{ 'a poly where
    b in = (if i\geqn then [:c n:] else b (Suc i) n*X i +[:c i:])
    by pat-completeness auto
termination by (relation measure ( }\lambda(i,n).Suc n-i),auto
declare b.simps[simp del]
definition newton-poly :: nat }=>\mathrm{ ' 'a poly where
    newton-poly n = b 0 n
```

```
private definition Xij ij = prod-list (map X [i..<j])
private definition Ni= Xij 0i
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 (\lambda i. smult (c i) (N i)) [0 ..< Suc n])
    unfolding newton-poly-def N-def
proof -
    {
        fix }
        assume j\leqn
        hence b j n = (\sumi\leftarrow[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\geqn)
            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\leqn by auto
            note IH=1(1)[OF False this]
            have id2: (\sumx\leftarrow[Suc j..<nn]. smult (c x) (Xij (Suc j) x*X j))=
                (\sumi\leftarrow[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: idQ)
            qed
        qed
    }
    from this[of 0] show b 0 n = (\sumi\leftarrow[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 ( $\lambda$ i. c $i * \operatorname{poly}(N i) y)[0 . .<S u c n])$
unfolding newton-poly-sum poly-sum-list map-map o-def by simp
private definition pprod $k i j=\left(\prod l \leftarrow[i . .<j] . x d k l\right)$
private lemma poly- $N$-xi: poly $(N i)(x j)=\operatorname{pprod} j 0 i$
proof -
have poly $(N i)(x j)=\left(\prod l \leftarrow[0 . .<i] . x d j l\right)$
unfolding $N$-def Xij-def poly-prod-list X-def[abs-def] map-map o-def $x d$-def by simp
also have ... = pprod j 0 i unfolding pprod-def ..
finally show ?thesis .
qed
private lemma poly- $N$-xi-cond: poly $(N i)(x j)=(i f j<i$ 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 \(j \in \operatorname{set}[0 . .<i]\) by auto
            from split-list \([O F\) this \(]\) obtain bef aft where id2: \([0\).. \(<i]=\) bef @ \(j \#\) aft
    by auto
have $(\Pi k \leftarrow[0 . .<i] . x d j k)=0$ unfolding $i d 2$ by auto
with True show ?thesis unfolding poly-N-xi pprod-def by auto
qed
qed
private lemma poly-newton-poly-xj: assumes $j \leq n$
shows poly (newton-poly $n)(x j)=$ sum-list $(\operatorname{map}(\lambda i . c i * \operatorname{poly}(N i)(x j))[0$
.. $<$ Suc $j$ ])
proof -
from assms have id: $[0$.. $<$ Suc $n]=[0$.. $<$ Suc $j] @[S u c j$.. $<$ Suc n]
by (metis Suc-le-mono le-Suc-ex less-eq-nat.simps(1) upt-add-eq-append)
have $i d 2:\left(\sum i \leftarrow[S u c j . .<S u c n] . c i * \operatorname{poly}(N i)(x j)\right)=0$
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: $\bigwedge i j . i<j \Longrightarrow j \leq n \Longrightarrow x i \neq x j$

## begin

private lemma $x d$-diff: $i<j \Longrightarrow j \leq n \Longrightarrow x d i j \neq 0$ $i<j \Longrightarrow j \leq n \Longrightarrow x d j i \neq 0$ using dist[of $i j] \operatorname{dist}[$ of $j i]$ unfolding $x d$-def by auto

This is the key technical lemma for soundness of Newton interpolation.
private lemma divided-differences-main: assumes $k \leq n i<k$
shows sum-list (map $(\lambda j$. xij-f $i(i+j) * \operatorname{pprod} k i(\bar{i}+j))[0 . .<S u c k-i])=$ sum-list (map $(\lambda j$. xij-f (Suc $i)(S u c i+j) * \operatorname{prod} k($ Suc $i)(S u c i+j))$ $[0 . .<$ Suc $k-S u c i])$
proof -
let ? $\exp =\lambda i j$. xij-f $i(i+j) * \operatorname{prod} k i(i+j)$
define $e i$ where $e i=? \exp i$
define esi where esi $=?$ exp $(S u c i)$
let ? $k i=k-i$
let ?sumi $=\lambda$ xs. sum-list (map ei xs)
let ?sumsi $=\lambda$ xs.sum-list (map esi xs)
let ?mid $=\lambda j$. xij-f $i(k-j) * \operatorname{pprod} k(S u c i)(k-j) * x d(k-j) i$
let ?sum $=\lambda j$. ?sumi $[0 . .<$ ? $k i-j]+$ ?sumsi $[$ ? $k i-j . .<$ ? $k i]+$ ?mid $j$
define fin where fin $=? k i-1$
have fin: fin $<$ ? $k i$ unfolding fin-def using assms by auto
have $i d:[0 . .<S u c k-i]=[0 . .<? k i] @[? k i]$ and
id2: $[i . .<k]=i \#[$ Suc $i . .<k]$ and
id3: $k-(i+(k-$ Suc $i))=1 k-(? k i-1)=$ Suc $i$ using assms
by (auto simp: Suc-diff-le upt-conv-Cons)
have neq: $x d$ (Suc i) $i \neq 0$ using $x d$-diff $[$ of $i$ Suc $i]$ assms by auto
have sum-list (map $(\lambda j$. xij-f $i(i+j) * \operatorname{pprod} k i(i+j))[0 . .<S u c k-i])$
$=$ ? sumi $[0 . .<$ Suc $k-i]$ unfolding ei-def by simp
also have $\ldots=$ ? sumi $[0 . .<$ ? $k i]+$ ?sumsi $[$ ? $k i . .<$ ? $k i]+$ ei ? $k i$
unfolding $i d$ by $\operatorname{simp}$
also have $\ldots=$ ? 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 $<$ ? $k i$ and $i k f: i<k-S u c f i n ~ i<k-f i n$ and $k f n: k-f i n \leq$ $n$ by auto
from $x d$-diff $[O F \operatorname{ikf}(2) k f n]$ have $n z: x d(k-f i n) i \neq 0$ by auto
note $I H=\operatorname{Suc}(1)[O F f k i]$
have id4: $[0 . .<? k i-f i n]=[0 . .<? k i-S u c f i n] @[? k i-S u c f i n]$
$i+(k-i-$ Suc fin $)=k-$ Suc fin
Suc ( $k-$ Suc fin) $=k-$ fin using Suc(2) assms 〈fin $<$ ? $k i$ 〉
by (metis Suc-diff-Suc le0 upt-Suc) (insert Suc(2), auto)
from Suc(2) assms have id5: $[i . .<k-S u c f i n]=i \#[S u c i \quad . .<k-S u c f i n]$
$[$ Suc $i . .<k-f i n]=[$ Suc $i . .<k-S u c f i n] @[k-S u c$ fin $]$
by (force simp: upt-rec) (metis Suc-leI id4 (3) ikf(1) upt-Suc)
have ?sum $0=$ ? sum fin by (rule $I H$ )
also have $\ldots=$ ? sumi $[0 . .<$ ?ki - Suc fin $]+$ ?sumsi $[$ ? $k i-$ fin..$<$ ? $k i]+$

$$
(e i(? k i-S u c f i n)+? m i d f i n)
$$

unfolding $i d_{4}$ by simp
also have ? mid fin $=(x i j-f($ Suc $i)(k-f i n)-x i j-f i(k-S u c f i n))$

* pprod $k$ (Suc $i$ ) ( $k$ - fin) unfolding xij-f.simps[of $i k-f i n]$
using $i k f n z$ by simp
also have $\ldots=x i j-f(S u c i)(k-f i n) * \operatorname{pprod} k(S u c i)(k-f i n)-$ xij-f $i(k-S u c$ fin) * pprod $k$ (Suc $i)(k-f i n)$ by algebra
also have xij-f (Suc i) $(k-f i n) * \operatorname{pprod} k(S u c i)(k-f i n)=$ esi $(? k i-S u c$ fin)
unfolding esi-def using ikf by (simp add: id4)
also have $e i(? k i-S u c f i n)=x i j-f i(k-S u c f i n) * \operatorname{prod} k i(k-S u c f i n)$
unfolding ei-def id4 using ikf by (simp add: ac-simps)
finally have ?sum $0=$ ? sumi $[0 . .<$ ? $k i-S u c f i n]$

$$
+(\text { esi }(? k i-\text { Suc fin })+? \text { sumsi }[? k i-f i n . .<? k i])
$$

$+($ xij-f $i(k-$ Suc fin $) *($ pprod $k i(k-S u c f i n)-\operatorname{pprod} k(S u c i)(k-$
fin)))
by algebra
also have esi $(? k i-S u c f i n)+? s u m s i[? k i-f i n . .<? k i]$
$=$ ? sumsi $((? k i-S u c$ fin $) \#[? k i-f i n . .<? k i])$ by $\operatorname{simp}$
also have $(? k i-S u c ~ f i n) ~ \# ~[? k i-f i n ~ . .<? k i]=[? k i-S u c f i n ~ . .<? k i]$
using Suc(2) by (simp add: Suc-diff-Suc upt-rec)
also have pprod $k i(k-S u c f i n)-\operatorname{pprod} k$ (Suc $i)(k-f i n)$
$=(x d k i) * \operatorname{pprod} k($ Suc $i)(k-$ Suc fin $)-(x d k(k-S u c$ fin $)) * \operatorname{prod} k$
(Suc i) ( $k-$ Suc fin)
unfolding pprod-def id5 by simp
also have $\ldots=(x d k i-x d k(k-S u c f i n)) * \operatorname{prod} k(S u c i)(k-S u c f i n)$ by algebra
also have $\ldots=(x d(k-S u c f i n) i) * \operatorname{prod} k$ (Suc $i)(k-S u c$ fin $)$ unfolding $x d-\operatorname{def}$ by simp
also have xij-f $i(k-S u c f i n) * \ldots=$ ? mid (Suc fin) by simp
finally show ?case by simp
qed $\operatorname{simp}$
also have $\ldots=($ 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 $i d 3$
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 $\ldots=$ sum-list (map $(\lambda j$. xij-f (Suc $i)(S u c i+j) *$ pprod $k$ (Suc $i)($ Suc $i+j))[0 . .<$ Suc $k-S u c i])$
unfolding esi-def using assms by simp
finally show ?thesis.
qed
private lemma divided-differences: assumes $k n: k \leq n$ and $i k: i \leq k$
shows sum-list (map $(\lambda j$. xij-f $i(i+j) * \operatorname{prod} k i(i+j))[0 . .<S u c k-i])=$

```
fk
proof -
    {
    fix ii
    assume i+ ii \leqk
    hence sum-list (map ( }\lambda\mathrm{ j. xij-f i (i+j)* pprod ki (i+j)) [0..<Suc k - i])
            =sum-list (map (\lambda 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\leqk 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)\leqk 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
    shows poly (newton-poly n) (x k) =fk
proof -
    have poly (newton-poly n) (x k)=
        sum-list (map (\lambda 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 ... =fk
        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 }
        have i\leqn\Longrightarrowdegree (b i n) \leqn-i
    proof (induct i n rule: b.induct)
        case (1 i n)
        note b = b.simps[of in]
        show ?case
        proof (cases n \leqi)
            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 ... \leqmax (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 ... \leq degree (b (Suc i) n) + degree (X i)
                    by (rule degree-mult-le)
            also have ... \leqn-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-Suci+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\Longrightarrow newton-coefficients-main (rev (map f [0..<Suc k])) (rev (map x
[0..<Suc k]))
    = rev (map (\lambda i.map (\lambda j. xij-fj 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\mathrm{ by auto
    note IH = Suc(1)[OF this]
    have id: \f.rev (map f[0..<Suc (Suc k)]) =f(Suc k)#fk# rev (map f
[0..<k])
    and id2: \f.fk# rev (map f[0..<k])=rev (map f [0..<Suck]) 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}\mathrm{ 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 ((\lambdaj. xij-fj k)) [nn..< Suc k]) (f
(Suc k)) (x (Suc k)) (map x [nn ..< n]) =
```

```
        map ((\lambdaj. 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\rangle have nn<n and le: nn<Suck 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 \(n n\)
```

shows newton-poly-impl $=$ newton-poly $n n$ proof -
define $i$ where $i=(0::$ nat $)$
have $x s$ : map $x[0 . .<n]=x s$ using $x s$
by (intro nth-equalityI, auto simp: $x$-def)
have $i \leq n n$ unfolding $i$-def by simp
hence horner-composition (map c $[i . .<$ Suc nn $])(\operatorname{map} x[i . .<S u c n n])=b i n n$ proof (induct $i$ nn rule: b.induct)
case (1in)
show ?case
proof (cases $n \leq i$ )
case True
with 1 (2) have $i: i=n$ by simp
show ?thesis unfolding $i$ b.simps $[o f n n]$ by simp
next
case False
hence Suc $i \leq n$ by simp
note $I H=1(1)[$ OF False this]
have $b i$ : $b i n=b(S u c i) n * X i+[: c i:]$ using False by (simp add: b.simps)
from False have $i d:\left[\begin{array}{l}i \\ . .\end{array}\right.$ Suc $\left.n\right]=i \#[$ Suc $i$.. $<$ Suc $n]$ by (simp add: upt-rec)
from False have id2: [Suc $i \quad . .<$ 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 $I H[$ 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] $x s$.
qed
end
end
context
fixes $x s$ fs :: int list
begin
fun divided-differences-impl-int $::$ int list $\Rightarrow$ int $\Rightarrow$ int $\Rightarrow$ int list $\Rightarrow$ int list option where
divided-differences-impl-int (xi-j1 \# x-j1s) fj xj $(x i \# x i s)=($
case divided-differences-impl-int $x$-j1s fj xj xis of None $\Rightarrow$ None
$\mid$ Some $x$ - $j s \Rightarrow$ let $(n e w, m)=$ divmod-int $(h d x-j s-x i-j 1)(x j-x i)$
in if $m=0$ then Some (new \# x-js) else None)
$\mid$ divided-differences-impl-int [] fj xj xis $=$ Some [fj]
fun newton-coefficients-main-int $::$ int list $\Rightarrow$ int list $\Rightarrow$ int list list option where

```
    newton-coefficients-main-int [fj] xjs = Some [[fj]]
| newton-coefficients-main-int (fj # fjs) (xj # xjs) = (do {
    rec }\leftarrow\mathrm{ newton-coefficients-main-int fjs xjs;
    let row = hd rec;
    new-row \leftarrow 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 \leq length ys
    \Longrightarrow ~ d i v i d e d - d i f f e r e n c e s - i m p l - i n t ~ g s ~ g ~ x ~ y s ~ = ~ S o m e ~ r e s ~
    \Longrightarrow ~ d i v i d e d - d i f f e r e n c e s - i m p l ~ ( m a p ~ r a t - o f - i n t ~ g s ) ~ ( r a t - o f - i n t ~ g ) ~ ( r a t - o f - i n t ~ x ) ~ ( m a p ~
rat-of-int ys) = map rat-of-int res
    \wedge ~ l e n g t h ~ r e s ~ = ~ S u c ~ ( l e n g t h ~ g s )
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 \leq 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 mod:(hd x-js - xi-j1)
mod (xj - xi) = 0
            and res: res = (hd x-js - xi-j1) div (xj - xi) # x-js by (auto split: if-splits)
    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)
    using mod by force
    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\in\mathbb{Z}b\not=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 \leq length ys
    \Longrightarrow ~ d i v i d e d - d i f f e r e n c e s - i m p l - i n t ~ g s ~ g ~ x ~ y s ~ = ~ N o n e ~
    \Longrightarrow # 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)) &\mathbb{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 \leq length xis and xj: xj & set (take (length
x-j1s) xis) and xji: xj \not= xi by auto
    define d}\mathrm{ 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 &\mathbb{Z}\mathrm{ unfolding d-def by auto}
        {
            let ?x = (hd d - rat-of-int xi-j1) / (rat-of-int xj - rat-of-int xi)
            assume ?x }\in\mathbb{Z
            hence ?x * (of-int (xj - xi)) + rat-of-int xi-j1 \in\mathbb{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 }\not=[] by aut
            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}\not=
                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)
\in\mathbb{Z}
    hence rat-of-int a / rat-of-int b\in\mathbb{Z unfolding a-def b-def by simp}
    moreover have }b\not=0\mathrm{ 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 \(y s \Longrightarrow\) length \(y s \leq\) length \(x s\)
\(\Longrightarrow\) newton-coefficients-main-int gs ys \(=\) Some res
\(\Longrightarrow\) newton-coefficients-main (map rat-of-int xs) (map rat-of-int gs) (map rat-of-int \(y s)=\operatorname{map}(\) map rat-of-int) res
\(\wedge(\forall x \in\) set res. \(x \neq[] \wedge\) length \(x \leq\) length \(y s) \wedge\) length res \(=\) length \(g s\) 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 ( \(v \# v a\) ) = length \(x j s\) length \(x j s \leq\) length \(x s\) by auto
note some \(=2(4)\)
let ? \(n=\) newton-coefficients-main-int \((v \# v a) x j s\)
let \(? r i=r a t-o f-i n t\)
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 \(d d\) where \(d: ? d=\) Some \(d d\) and res: res \(=d d \#\) rec
by (cases? \(d\), auto)
note \(I H=2(1)[\) OF len \(n]\)
```



```
with len have length ( \(h d r e c\) ) \(\leq\) length \(x s\) by auto
note \(d d=\) divided-differences-impl-int-Some[OF this \(d]\)
have \(h d\) : 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 \((d d \#\) rec \()=\) length \((f v \# v \# v a)\) using len
IH[THEN conjunct2] dd[THEN conjunct2] by auto
show \(\forall x \in\) insert dd (set rec). \(x \neq[] \wedge\) length \(x \leq\) length \((x j \# x j s)\)
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 \Longrightarrow length ys \leq length xs
    \Longrightarrow ~ n e w t o n - c o e f f i c i e n t s - m a i n - i n t ~ g s ~ y s ~ = ~ N o n e ~
    \Longrightarrow y s = d r o p ~ ( l e n g t h ~ x s ~ - ~ l e n g t h ~ y s ) ~ ( r e v ~ x s ) ~
    \Longrightarrow row \in set (newton-coefficients-main (map rat-of-int xs) (map rat-of-int gs)
(map rat-of-int ys)). hd row }\not\in\mathbb{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 \leqlength 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 ? }\mp@subsup{n}{}{\prime}=\mathrm{ 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\inset ? n' and hd row }\not\in\mathbb{Z}\mathrm{ 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) \leq length xjs by (cases rec, auto)
        hence lenn: length (hd rec)\leqlength 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 \in 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) \leq length xs using 2(3) by auto
        with i have i0: i\not=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 \in 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: \(\neg\) distinct ( \(x j \# x j s\) ) by auto
            from dist have distinct (rev xs) by simp
            from distinct-drop[OF this] have distinct ( \(x j \#\) xjs) using 2(5) by metis
            with ndist
            show False ..
    qed
    note \(d d=\) 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 $x s=$ length $f s$
shows newton-coefficients-int $=($ let cs $=$ newton-coefficients (map rat-of-int xs)
(map of-int fs)
in if set cs $\subseteq \mathbb{Z}$ then Some (map int-of-rat cs) else None)
proof -
from len have len: length (rev $f_{s}$ ) = length (rev xs) length (rev xs) $\leq$ length $x s$
by auto
show ?thesis
proof (cases newton-coefficients-main-int (rev fs) (rev xs))
case (Some res)
have rev: $\bigwedge$ 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 $\in$ set res
with $n$ have row $\neq[]$ by auto
hence $i d$ : $h d$ (map rat-of-int row) $=$ rat-of-int ( $h d$ row) by (cases row, auto)
also have $\ldots \in \mathbb{Z}$ by auto
finally have int: $h d$ (map rat-of-int row) $\in \mathbb{Z}$ by auto
have $h d$ row $=$ int-of-rat ( $h d$ ( map rat-of-int row)) unfolding id by simp
note this int
\}
thus ?thesis unfolding newton-coefficients-int-def Some newton-coefficients-def
$n[T H E N$ conjunct1] Let-def option.simps
by (auto simp: o-def)
next
case None
have rev $x s=$ drop (length $x s-$ 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 \(\equiv\) case newton-coefficients-int of None \(\Rightarrow\) None
        \(\mid\) Some \(n c \Rightarrow\) Some (horner-composition (rev nc) xs)
lemma newton-poly-impl-int: assumes len: length \(x s=\) length \(f s\)
    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 \(p) \subseteq \mathbb{Z}\) then Some (map-poly int-of-rat p) else None)
proof -
    let ? ir = int-of-rat
    let ? \(\mathrm{ri}=\) 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 ? \(n c \subseteq \mathbb{Z}\)
    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 \(\operatorname{simp}\)
    have len: length (rev ?nc) \(\leq\) Suc (length \(x s\) )
        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 \(c s\) where \(c s=r e v\) ? \(n c\)
    define ics where ics = map ? ir cs
    assume set ? \(n c \subseteq \mathbb{Z}\)
    hence set \(c s \subseteq \mathbb{Z}\) 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 \(x s\) )
            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 :: ( \({ }^{\prime} a\) :: field \(\left.\times{ }^{\prime} a\right)\) list \(\Rightarrow{ }^{\prime} a\) poly where
    newton-interpolation-poly \(x\) - \(f s=\) (let
        \(x s=\) map fst \(x\) - \(f s ; f s=\) map snd \(x\) - \(f s\) in
        newton-poly-impl xs fs)
```

```
definition newton-interpolation-poly-int :: (int }\times\mathrm{ int)list }=>\mathrm{ 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)\in 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\leqnn 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 ij
        show }i<j\Longrightarrowj\leqnn\Longrightarrow\mathrm{ ?xs ! i}\not=\mathrm{ ?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) \leqlength 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 \(=\operatorname{map}(\lambda(x, y) .(\) rat-of-int \(x\), rat-of-int \(y)) x s\)-ys;
        \(r p=\) newton-interpolation-poly rxs-ys
        in if \((\forall x \in \operatorname{set}(\) coeffs rp). is-int-rat \(x)\) then
            Some (map-poly int-of-rat rp) else None)
proof -
    have id1: map fst \((\operatorname{map}(\lambda(x, y) .(\) rat-of-int \(x\), rat-of-int \(y)) x s\) - \(y s)=\) map
rat-of-int (map fst xs-ys)
            by (induct xs-ys, auto)
    have id2: map snd (map \((\lambda(x, y)\). (rat-of-int x, rat-of-int \(y)) x s-y s)=m a p\)
rat-of-int (map snd xs-ys)
            by (induct xs-ys, auto)
    have id3: length (map fst \(x s-y s\) ) \(=\) length (map snd \(x s\)-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 \([a b s-d e f])\)
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 $\left(x_{1}, p\left(x_{1}\right)\right),\left(x_{2}, p\left(x_{2}\right)\right), \ldots$. The interpolation algorithm is proven to be sound and complete.

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

definition lagrange-basis-poly :: ' $a$ :: field list $\Rightarrow{ }^{\prime} a \Rightarrow$ ' $a$ poly where
lagrange-basis-poly xs $x j \equiv$ let $y s=$ filter $(\lambda x . x \neq x j) x s$
in prod-list (map ( $\lambda$ xi.smult (inverse $(x j-x i))[:-x i, 1:]) y s)$
definition lagrange-interpolation-poly :: ('a :: field $\times$ ' $a$ ) list $\Rightarrow$ ' $a$ poly where
lagrange-interpolation-poly xs-ys $\equiv$ let
$x s=m a p f s t x s-y s$
in sum-list ( $\operatorname{map}(\lambda(x j, y j)$. smult yj (lagrange-basis-poly $x s x j)) x s-y s)$
lemma [code]:
lagrange-basis-poly xs $x j=($ let $y s=$ filter $(\lambda x . x \neq x j) x s$
in prod-list $(\operatorname{map}(\lambda x i$. let $i i=$ inverse $(x j-x i)$ in $[:-i i * x i, i i:]) y s))$
unfolding lagrange-basis-poly-def Let-def by simp
lemma degree-lagrange-basis-poly: degree (lagrange-basis-poly xs xj) $\leq$ length (filter $(\lambda x . x \neq x j) x s)$
unfolding lagrange-basis-poly-def Let-def
by (rule order.trans[OF degree-prod-list-le], rule order-trans[OF sum-list-mono[of - - $\lambda$-. 1]],
auto simp: o-def, induct xs, auto)
lemma degree-lagrange-interpolation-poly:
shows degree (lagrange-interpolation-poly xs-ys) $\leq$ length $x s-y s-1$
proof -
\{
fix $a b$
assume $a b:(a, b) \in$ set $x s$-ys
let ? $x s=$ filter $(\lambda x . x \neq a)($ map fst $x s$ - $y s)$
from $a b$ have $a \in$ set (map fst xs-ys) by force
hence Suc (length ?xs) $\leq$ length $x s$-ys
by (induct xs-ys, auto)
hence length ? $x s \leq$ length $x s$-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 $x^{\prime} \in \operatorname{set}($ map fst $x s-y s)$ and $x^{\prime} \neq x$
shows poly (lagrange-basis-poly (map fst xs-ys) $x$ ) $x^{\prime}=0$
proof -
let $? f=\lambda x i$. smult (inverse $(x-x i))[:-x i, 1:]$

```
    let ?xs = filter (\lambdac. c\not=x) (map fst xs-ys)
    have mem:?f }\mp@subsup{x}{}{\prime}\in\mathrm{ 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 \}\xy.(x,y)\in\mathrm{ set xs-ys C poly p x = y
proof -
    let ?xs = map fst xs-ys
    {
        fix }x
        assume xy: (x,y)\in set xs-ys
        show poly px=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}\in\operatorname{set (map ( \lambdaxa.poly (case xa of (xj, yj) => smult yj (lagrange-basis-poly
?xs xj))
                    x)
                        (remove1 (x,y) xs-ys)) (is - \in set (map ?f ?xy))
            then obtain }x\mp@subsup{y}{}{\prime}\mathrm{ where mem: xy' G set ?xy and v:v=?f xy' by auto
            obtain }\mp@subsup{x}{}{\prime}\mp@subsup{y}{}{\prime}\mathrm{ where }x\mp@subsup{y}{}{\prime}:x\mp@subsup{y}{}{\prime}=(\mp@subsup{x}{}{\prime},\mp@subsup{y}{}{\prime})\mathrm{ by force
            from v[unfolded this split] have v:v=poly (smult y' (lagrange-basis-poly?xs
x')) x.
            have neq: 和 }=
            proof
                assume x' = x
                with mem[unfolded xy'] have mem: (x,\mp@subsup{y}{}{\prime})\in\operatorname{set (remove1 (x,y) xs-ys) by}
auto
            hence mem': (x,y')\in 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)\not\in 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=
                    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
```


## 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 }=>\mathrm{ nat }=>\mathrm{ ' 'a poly where
    neville-aitken-main i j= (if i<j then
        (smult (inverse (xj-xi)) (Xi* neville-aitken-main (i+1)j-
        Xj* neville-aitken-main i (j-1)))
        else [:f i:])
    by pat-completeness auto
termination by (relation measure ( }\lambda(i,j).j-i),auto
definition neville-aitken :: nat }=>\mathrm{ 'a poly where
    neville-aitken = neville-aitken-main 0
declare neville-aitken-main.simps[simp del]
lemma neville-aitken-main: assumes dist: \ ij. i<j\Longrightarrowj\leqn\Longrightarrowx 
    shows i\leqk\Longrightarrowk\leqj\Longrightarrowj\leqn\Longrightarrow poly (neville-aitken-main ij) (x k)=(f
k)
proof (induct i j arbitrary: k rule: neville-aitken-main.induct)
    case (1 ijk)
    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\not=xj by auto
```

from True have $i d$ : 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 $I H=1(1-2)[$ 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 $k i=$ 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 $k i$ show ?thesis unfolding id poly-smult using $\operatorname{IH}(1-2)[o f k] i j$
1 (3-) diff by (simp add: X-def field-simps)
qed
qed
qed
qed
lemma degree-neville-aitken-main: degree (neville-aitken-main $i j) \leq j-i$
proof (induct ij rule: neville-aitken-main.induct)
case ( $1 i j$ )
note $\operatorname{simp}=$ 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 $I H=1$ [OF this]
let $? n=$ neville-aitken-main
have $X$ : $\wedge$ i. degree $(X i)=$ Suc 0 unfolding $X$-def by auto
have degree $(X i * ? n(i+1) j) \leq \operatorname{Suc}($ degree $(? n(i+1) j))$
by (rule order.trans $[$ OF degree-mult-le], simp add: $X$ )
also have $\ldots \leq \operatorname{Suc}(j-(i+1))$ using $I H(1)$ by simp
finally have 1: degree $(X i * ? n(i+1) j) \leq j-i$ using True by auto
have degree $(X j * ? n i(j-1)) \leq \operatorname{Suc}($ degree $(? n i(j-1)))$
by (rule order.trans $[$ OF degree-mult-le], simp add: X)
also have $\ldots \leq$ Suc $((j-1)-i)$ using $I H(2)$ by simp
finally have 2: degree $(X j *$ ? $n i(j-1)) \leq j-i$ using True by auto have $i d$ : ?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 $\operatorname{simp}$
have degree $(? n i j) \leq$ degree $(X i * ? n(i+1) j-X j * ? n i(j-1))$ unfolding id by simp also have $\ldots \leq \max ($ degree $(X i * ? n(i+1) j))($ degree $(X j * ? n i(j-$ 1))) by (rule degree-diff-le-max)
also have $\ldots \leq j-i$ using 12 by auto finally show ?thesis .
qed
qed
lemma degree-neville-aitken: degree (neville-aitken $n$ ) $\leq n$ unfolding neville-aitken-def using degree-neville-aitken-main $[$ of $0 n]$ by simp
fun neville-aitken-merge $::\left({ }^{\prime} a \times{ }^{\prime} a \times\right.$ 'a poly) list $\Rightarrow\left({ }^{\prime} a \times{ }^{\prime} a \times\right.$ 'a poly) list where neville-aitken-merge $((x i, x j, p-i j) \#(x s i, x s j, p-s i s j) \#$ rest $)=$
(xi,xsj, smult (inverse $(x s j-x i))([:-x i, 1:] * p-s i s j$
$+[: x s j,-1:] * p-i j)) \#$ neville-aitken-merge $((x s i, x s j, p-s i s j) \#$ rest $)$
| neville-aitken-merge [-] = []
| neville-aitken-merge [] = []
lemma length-neville-aitken-merge[termination-simp]: length (neville-aitken-merge $x s)=$ length $x s-1$
by (induct xs rule: neville-aitken-merge.induct, auto)
fun neville-aitken-impl-main :: ( ${ }^{\prime} a \times$ ' $a \times$ ' $a$ poly) list $\Rightarrow$ ' $a$ poly where
neville-aitken-impl-main (e1 \# e2 \# es) =
neville-aitken-impl-main (neville-aitken-merge (e1 \# e2 \# es))
$\mid$ neville-aitken-impl-main $[(-,-, p)]=p$
| neville-aitken-impl-main [] = 0
lemma neville-aitken-merge:

```
xs=map (\lambdai.(xi,x(i+j),neville-aitken-main i (i+j))) [l..<Suc (l+k)]
    \Longrightarrow ~ n e v i l l e - a i t k e n - m e r g e ~ x s ~
        =(map (\lambda 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 = \lambda ji. (xi,x (i+j), ?n i (i+j))
    define f}\mathrm{ where f=?f
    let ?map = j 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 =? ? l l l + j)
        xsi=x(Sucl) 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)) =(xl,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 # [Sucl ..<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 (\lambdai.(xi,x(i+j),neville-aitken-main i (i+j))) [l..<Suc (l+k)]
    \Longrightarrow ~ n e v i l l e - a i t k e n - i m p l - m a i n ~ x s ~ = ~ n e v i l l e - a i t k e n - m a i n ~ l ~ ( ~ l ~ + ~ j + k )
proof (induct xs arbitrary:l k j rule: neville-aitken-impl-main.induct)
    case (1 e1 e2 es l kj)
    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 (\lambda i. (x i, x i,[:f i:])) [0 ..< Suc k]
    \Longrightarrow ~ n e v i l l e - a i t k e n - i m p l - m a i n ~ x s ~ = ~ n e v i l l e - a i t k e n ~ 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 \(\wedge i j . i<j \Longrightarrow j \leq n \Longrightarrow x i \neq x j\)
shows \(j \leq n \Longrightarrow\) 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 \(\times\) 'a)list \(\Rightarrow\) ' \(a\) poly where
neville-aitken-interpolation-poly \(x\) - \(f s=(\) let
start \(=\operatorname{map}(\lambda(x i, f i) .(x i, x i,[: f i:])) x\)-fs in
neville-aitken-impl-main start)
lemma neville-aitken-interpolation-impl: assumes \(x\) - \(f s \neq[]\)
shows neville-aitken-interpolation-poly \(x\) - \(f s=\)
```

```
    neville-aitken (\lambda i.fst (x-fs!i)) (\lambda 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) \in set xs-ys
    shows poly p }x=
proof -
    have p:p= neville-aitken ( }\lambda\mathrm{ i.fst (xs-ys!i)) ( }\lambda\mathrm{ 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 px y
    proof (rule neville-aitken)
    fix ij
    show }i<j\Longrightarrowj\leqlength xs-ys - 1\Longrightarrowfst (xs-ys!i)\not=fst (xs-ys!j) usin
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) \leqlength 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 (\lambda i.fst (xs-ys!i)) (\lambda 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 }=>\mathrm{ (int }\times\mathrm{ int)list }=>\mathrm{ 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 (\lambda (x,y). (of-int x, of-int y)) xs-ys;
        rp = interpolation-poly alg rxs-ys
        in if ( }\forallx\in\mathrm{ 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) \Longrightarrow
    interpolation-poly-int alg xs-ys = (let
        rxs-ys=map (\lambda (x,y). (of-int x, of-int y)) xs-ys;
        rp = interpolation-poly alg rxs-ys
        in if ( }\forallx\in\mathrm{ 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)\in 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) \leq 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\mathrm{ :: idom poly
    assumes cS: card S=Suc n
    and degree p}\leqn\mathrm{ and degree }q\leqn\mathrm{ and
    id: \ x. x }\inS\Longrightarrow\mathrm{ poly p x = poly q }
    shows p=q
proof -
    define f}\mathrm{ where f=p-q
    let ?R={x. poly fx=0}
    have sub: S\subseteq?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 fx=0
        from poly-roots-finite[OF f] have finite ?R .
        from card-mono[OF this sub] poly-roots-degree[OF f]
        have Suc n \leq degree f unfolding cS by auto
        also have ... \leqn 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:\bigwedgexy.(x,y)\in set xs-ys \Longrightarrow poly p x = y degree p < length xs-ys
    and q: \bigwedgexy.(x,y)\in set xs-ys \Longrightarrow 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}\leqn\mathrm{ by
(cases xs-ys, auto)
```

from $p$ have $d p$ : degree $p \leq n$ 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 $x \in$ set ? $x s$
then obtain $y$ where $(x, y) \in$ set $x s$ - $y s$ 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 $\neq[]$ and dist: distinct (map fst $x s$-ys)
shows $\exists$ ! p. degree $p<$ length $x s$ - $y s \wedge(\forall x y .(x, y) \in$ set $x s$ - $y s \longrightarrow$ poly $p x=$ ( $y$ :: ' $a$ :: field $)$ )
proof -
let ?alg $=$ undefined
let ? $p=$ interpolation-poly ?alg $x s$ - $y s$
note inter $=$ interpolation-poly $[O F$ dist refl $]$
show ?thesis
proof (rule ex1I[of - ?p], intro conjI allI impI)
show dp: degree ? $p<$ length $x s$-ys using degree-interpolation-poly[of ?alg xs-ys] xs by (cases xs-ys, auto)
show $\bigwedge x y .(x, y) \in$ set $x s$ - $y s \Longrightarrow$ poly (interpolation-poly ?alg $x s-y s) x=y$
by (rule inter)
fix $q$
assume $q$ : degree $q<$ length $x s$-ys $\wedge(\forall x y .(x, y) \in$ set $x s$ - $y s \longrightarrow$ 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 $\bigwedge x y .(x, y) \in$ set $x s$-ys $\Longrightarrow$ poly $p x=y$ degree $p \leq$ length $x s-y s-1$
proof -
let $? r=r a t-o f$-int
define rxs-ys where rxs-ys $=\operatorname{map}(\lambda(x, y) .(? r x$, ?r $y)) x s-y s$
have dist: distinct (map fst rxs-ys) using dist' unfolding distinct-map rxs-ys-def inj-on-def by force
obtain $r p$ where $r p: r p=$ 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 $i d: r p=$ 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
        assume (x,y) \in set xs-ys
        hence (?r x, ?r y) \in 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\mathrm{ by auto
    }
    show degree p s 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 $x s$ - $y s=$ None
and $q: \bigwedge x y .(x, y) \in$ set $x s$ - $y s \Longrightarrow$ poly $q x=y$
and $d q$ : degree $q<$ length xs-ys
shows False
proof -
let $? r=r a t-o f-i n t$
let ? $r p=$ map-poly ? $r$
define rxs-ys where rxs-ys $=\operatorname{map}(\lambda(x, y)$. (?r $x$, ?r $y)) x s$ - $y s$
have dist': distinct (map fst rxs-ys) using dist unfolding distinct-map rxs-ys-def
inj-on-def by force
obtain $r p$ where $r p$ : $r p=$ interpolation-poly alg rxs-ys by blast
note degrp $=$ degree-interpolation-poly[of alg rxs-ys, folded rp]
from $q$ have $q^{\prime}: \bigwedge x y .(x, y) \in$ set rxs-ys $\Longrightarrow$ poly $(? r p q) x=y$ unfolding
rxs-ys-def
by auto
have $[$ simp $]$ : degree $($ ?rp $q)=$ degree $q$ by simp
have $i d: r p=? r p q$
by (rule uniqueness-of-interpolation-point-list[OF dist' interpolation-poly[OF
dist' $r p 1]$,
insert $q^{\prime}$ dq degrp, auto simp: rxs-ys-def)
from $p$ [unfolded interpolation-poly-int-def[OF dist] Let-def, folded rxs-ys-def rp]
have $\exists c \in$ set (coeffs rp). $c \notin \mathbb{Z}$ 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:\neg (\exists p. poly p 0 = 0 ^ poly p 2 = (1 ::
int))
proof
    assume \exists 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 (\lambda i.(int i, poly p (int i))) [ 3 ..< 3 + degree
p]
    let ?l = \lambda 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:\xy.(x,y)\in set ?xs-ys \Longrightarrow 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' x\mp@subsup{y}{}{\prime})
                obtain }\mp@subsup{x}{}{\prime}\mp@subsup{y}{}{\prime}\mathrm{ where }x\mp@subsup{y}{}{\prime}:x\mp@subsup{y}{}{\prime}=(\mp@subsup{x}{}{\prime},\mp@subsup{y}{}{\prime})\mathrm{ 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.


[^0]:    *Supported by FWF (Austrian Science Fund) project Y757.

