

Polynomial Interpolation*

René Thiemann and Akihisa Yamada

December 14, 2021

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	5
4.1	<i>divmod-integer</i>	5
4.2	<i>divmod-nat</i>	6
4.3	<i>(choose)</i>	7
5	Several Locales for Homomorphisms Between Types.	7
5.1	Basic Homomorphism Locales	7
5.2	Commutativity	8
5.3	Division	10
5.4	(Partial) Injectivity	10

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

5.5	Surjectivity and Isomorphisms	13
5.6	Example Interpretations	15
6	Missing Unsorted	16
7	Missing Polynomial	22
7.1	Basic Properties	22
7.2	Polynomial Composition	24
7.3	Monic Polynomials	25
7.4	Roots	26
7.5	Divisibility	27
7.6	Map over Polynomial Coefficients	31
7.7	Morphismic properties of $pCons (0::'a)$	31
7.8	Misc	32
8	Connecting Polynomials with Homomorphism Locales	36
8.1	<i>map-poly</i> of Homomorphisms	37
	8.1.1 Injectivity	38
8.2	Example Interpretations	42
9	Newton Interpolation	43
10	Lagrange Interpolation	49
11	Neville Aitken Interpolation	50
12	Polynomial Interpolation	52

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

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

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

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 \Rightarrow bool

and *to-rat* :: 'a \Rightarrow rat

assumes *is-rat[simp]*: *is-rat* $x = (x \in \mathbb{Q})$

and *to-rat*: *to-rat* $x = (if\ x \in \mathbb{Q}\ then\ (THE\ y.\ x = of-rat\ y)\ else\ 0)$

lemma *of-rat-to-rat[simp]*: $x \in \mathbb{Q} \implies of-rat\ (to-rat\ x) = x$
<proof>

lemma *to-rat-of-rat[simp]*: *to-rat* (*of-rat* x) = x *<proof>*

instantiation *rat* :: *is-rat*

begin

definition *is-rat-rat* ($x :: rat$) = *True*

definition *to-rat-rat* ($x :: rat$) = x

instance

<proof>
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* ($x :: \text{real}$) = ($x \in \mathbb{Q}$)
definition *to-rat-real* ($x :: \text{real}$) = (if $x \in \mathbb{Q}$ then (THE y . $x = \text{of-rat } y$) else 0)
instance *<proof>*
end

lemma *of-nat-complex*: $\text{of-nat } n = \text{Complex } (\text{of-nat } n) 0$
<proof>

lemma *of-int-complex*: $\text{of-int } z = \text{Complex } (\text{of-int } z) 0$
<proof>

lemma *of-rat-complex*: $\text{of-rat } q = \text{Complex } (\text{of-rat } q) 0$
<proof>

lemma *complex-of-real-of-rat[simp]*: $\text{complex-of-real } (\text{real-of-rat } q) = \text{of-rat } q$
<proof>

lemma *is-rat-complex-iff*: $x \in \mathbb{Q} \iff \text{Re } x \in \mathbb{Q} \wedge \text{Im } x = 0$
<proof>

instantiation *complex :: is-rat*
begin
definition *is-rat-complex* ($x :: \text{complex}$) = ($\text{is-rat } (\text{Re } x) \wedge \text{Im } x = 0$)
definition *to-rat-complex* ($x :: \text{complex}$) = (if $\text{is-rat } (\text{Re } x) \wedge \text{Im } x = 0$ then *to-rat* ($\text{Re } x$) else 0)

instance *<proof>*
end

lemma *[code-unfold]*: $(x \in \mathbb{Q}) = (\text{is-rat } x)$ *<proof>*

definition *is-int-rat* :: $\text{rat} \Rightarrow \text{bool}$ **where**
is-int-rat $x \equiv \text{snd } (\text{quotient-of } x) = 1$

definition *int-of-rat* :: $\text{rat} \Rightarrow \text{int}$ **where**
int-of-rat $x \equiv \text{fst } (\text{quotient-of } x)$

lemma *is-int-rat[simp]*: $\text{is-int-rat } x = (x \in \mathbb{Z})$
<proof>

lemma *int-of-rat[simp]*: $\text{int-of-rat } (\text{rat-of-int } x) = x \wedge x \in \mathbb{Z} \implies \text{rat-of-int } (\text{int-of-rat } x)$

$z) = z$
 $\langle proof \rangle$

lemma *int-of-rat-0[simp]*: $(int-of-rat\ x = 0) = (x = 0)$ $\langle proof \rangle$

end

3 Divmod-Int

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

theory *Divmod-Int*
imports *Main*
begin

definition *divmod-int* :: $int \Rightarrow int \Rightarrow int \times int$ **where**
 $divmod-int\ n\ m = (n\ div\ m, n\ mod\ m)$

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

context
includes *integer.lifting*
begin

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

$(divmod-integer\ (integer-of-int\ m)\ (integer-of-int\ n))$
 $\langle proof \rangle$
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 $(0::'a) \leq x \wedge (0::'b) \leq y$, so that application can be ensured on non-negative values. Hence, one can drop "abs" in target language setup.

definition *divmod-abs'* **where**

$x \geq 0 \Rightarrow y \geq 0 \Rightarrow \text{divmod-abs}' \ x \ y = \text{Code-Numeral.divmod-abs} \ x \ y$

lemma *divmod-integer-code'*[code]: *divmod-integer* $k \ l =$

(if $k = 0$ then $(0, 0)$
 else if $l > 0$ then
 (if $k > 0$ then *divmod-abs'* $k \ l$
 else case *divmod-abs'* $(-k) \ l$ of $(r, s) \Rightarrow$
 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'* $(-k) \ (-l)$
 else case *divmod-abs'* $k \ (-l)$ of $(r, s) \Rightarrow$
 if $s = 0$ then $(-r, 0)$ else $(-r - 1, -l - s)$))

<proof>

code-printing — FIXME illusion of partiality

constant *divmod-abs'* \leftarrow
 (*SML*) *IntInf.divMod* / $(- / -)$
and (*Eval*) *Integer.div'-mod* / $(- / -)$
and (*OCaml*) *Z.div'-rem*
and (*Haskell*) *divMod* / $(- / -)$
and (*Scala*) $!((k: \text{BigInt}) => (l: \text{BigInt}) => / \text{if } (l == 0) / (\text{BigInt}(0), k) \text{ else} / (k \ ' / \% \ l))$

4.2 *divmod-nat*.

We implement *divmod-nat* via *divmod-integer* instead of invoking both division and modulo separately, and we further simplify the case-analysis which is performed in *divmod-integer* $?k \ ?l =$ (if $?k = 0$ then $(0, 0)$ else if $0 < ?l$ then if $0 < ?k$ then *divmod-abs'* $?k \ ?l$ else case *divmod-abs'* $(- ?k) \ ?l$ of $(r, s) \Rightarrow$ 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'* $(- ?k) \ (- ?l)$ 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]: *Divides.divmod-nat* $m \ n =$ ($m \ \text{div} \ n$)

let $k = \text{integer-of-nat } m; l = \text{integer-of-nat } n$
 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))
  ⟨proof⟩

```

4.3 (choose)

```

lemma binomial-code[code]:
  n choose k = (if k ≤ n then fact n div (fact k * fact (n - k)) else 0)
  ⟨proof⟩

```

end

5 Several Locales for Homomorphisms Between Types.

```

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

```

```

hide-const (open) mult

```

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

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

```

named-theorems hom-distrib

```

5.1 Basic Homomorphism Locales

```

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

```

```

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

```

```

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

```

```

locale plus-hom =
  fixes hom :: 'a :: plus ⇒ 'b :: plus

```

```

assumes hom-add[hom-distrib]: hom (x + y) = hom x + hom y

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

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

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

  Homomorphism distributes over product:

  lemma hom-prod-list: hom (prod-list xs) = prod-list (map hom xs)
    ⟨proof⟩

  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-distrib]: hom (x ^ n) = hom x ^ n
    ⟨proof⟩
end

locale monoid-add-hom = zero-hom hom + semigroup-add-hom hom
  for hom :: 'a :: monoid-add ⇒ 'b :: monoid-add
begin
  lemma hom-sum-list: hom (sum-list xs) = sum-list (map hom xs)
    ⟨proof⟩
  lemmas sum-list-map-hom[simp] = hom-sum-list[symmetric]
  lemma hom-add-eq-zero: assumes x + y = 0 shows hom x + hom y = 0
    ⟨proof⟩
end

locale group-add-hom = monoid-add-hom hom
  for hom :: 'a :: group-add ⇒ 'b :: group-add
begin
  lemma hom-uminus[hom-distrib]: hom (-x) = - hom x
    ⟨proof⟩
  lemma hom-minus [hom-distrib]: hom (x - y) = hom x - hom y
    ⟨proof⟩
end

```

5.2 Commutativity

```

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

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

```



```

locale comm-monoid-mult-hom = monoid-mult-hom hom
  for hom :: 'a :: comm-monoid-mult  $\Rightarrow$  'b :: comm-monoid-mult
begin
  sublocale ab-semigroup-mult-hom <proof>
  lemma hom-prod[hom-distrib]: hom (prod f X) = ( $\prod$  x  $\in$  X. hom (f x))
    <proof>
  lemma hom-prod-mset: hom (prod-mset X) = prod-mset (image-mset hom X)
    <proof>
  lemmas prod-mset-image[simp] = hom-prod-mset[symmetric]
  lemma hom-dvd[intro,simp]: assumes p dvd q shows hom p dvd hom q
    <proof>
  lemma hom-dvd-1[simp]: x dvd 1  $\implies$  hom x dvd 1 <proof>
end

locale comm-monoid-add-hom = monoid-add-hom hom
  for hom :: 'a :: comm-monoid-add  $\Rightarrow$  'b :: comm-monoid-add
begin
  sublocale ab-semigroup-add-hom <proof>
  lemma hom-sum[hom-distrib]: hom (sum f X) = ( $\sum$  x  $\in$  X. hom (f x))
    <proof>
  lemma hom-sum-mset[hom-distrib,simp]: hom (sum-mset X) = sum-mset (image-mset
hom X)
    <proof>
end

locale ab-group-add-hom = group-add-hom hom
  for hom :: 'a :: ab-group-add  $\Rightarrow$  'b :: ab-group-add
begin
  sublocale comm-monoid-add-hom <proof>
end

locale semiring-hom = comm-monoid-add-hom hom + monoid-mult-hom hom
  for hom :: 'a :: semiring-1  $\Rightarrow$  'b :: semiring-1
begin
  lemma hom-mult-eq-zero: assumes x * y = 0 shows hom x * hom y = 0
    <proof>
end

locale ring-hom = semiring-hom hom
  for hom :: 'a :: ring-1  $\Rightarrow$  'b :: ring-1
begin
  sublocale ab-group-add-hom hom <proof>
end

locale comm-semiring-hom = semiring-hom hom
  for hom :: 'a :: comm-semiring-1  $\Rightarrow$  'b :: comm-semiring-1
begin
  sublocale comm-monoid-mult-hom <proof>

```

end

```
locale comm-ring-hom = ring-hom hom
  for hom :: 'a :: comm-ring-1  $\Rightarrow$  'b :: comm-ring-1
begin
  sublocale comm-semiring-hom <proof>
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-distrib]: 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-distrib]: hom (inverse x) = inverse (hom x)
    <proof>
```

```
  sublocale idom-divide-hom hom
    <proof>
```

end

```
locale field-char-0-hom = field-hom hom
  for hom :: 'a :: field-char-0  $\Rightarrow$  'b :: field-char-0
```

5.4 (Partial) Injectivity

```
locale zero-hom-0 = zero-hom +
  assumes hom-0:  $\bigwedge x. \text{hom } x = 0 \implies x = 0$ 
begin
  lemma hom-0-iff[iff]: hom x = 0  $\longleftrightarrow$  x = 0 <proof>
end
```

```
locale one-hom-1 = one-hom +
  assumes hom-1:  $\bigwedge x. \text{hom } x = 1 \implies x = 1$ 
begin
  lemma hom-1-iff[iff]: hom x = 1  $\longleftrightarrow$  x = 1 <proof>
end
```

Next locales are at this point not interesting. They will retain some

results when we think of polynomials.

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

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

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

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

locale *injective* =
fixes *f* :: 'a \Rightarrow 'b **assumes** *injectivity*: $\bigwedge x y. f x = f y \implies x = y$
begin
lemma *eq-iff[simp]*: $f x = f y \longleftrightarrow x = y$ *<proof>*
lemma *inj-f*: *inj f* *<proof>*
lemma *inv-f-f[simp]*: *inv f (f x) = x* *<proof>*
end

locale *inj-zero-hom* = *zero-hom* + *injective hom*
begin
sublocale *zero-hom-0* *<proof>*
end

locale *inj-one-hom* = *one-hom* + *injective hom*
begin
sublocale *one-hom-1* *<proof>*
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* *<proof>*
sublocale *monoid-mult-hom-1* *<proof>*
end

locale *inj-monoid-add-hom* = *monoid-add-hom* + *inj-semigroup-add-hom*
begin
sublocale *inj-zero-hom* *<proof>*
sublocale *monoid-add-hom-0* *<proof>*
end

locale *inj-comm-monoid-mult-hom* = *comm-monoid-mult-hom* + *inj-monoid-mult-hom*
begin
sublocale *comm-monoid-mult-hom-1* *<proof>*

end

locale *inj-comm-monoid-add-hom* = *comm-monoid-add-hom* + *inj-monoid-add-hom*
begin
 sublocale *comm-monoid-add-hom-0*⟨*proof*⟩
end

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

locale *inj-comm-semiring-hom* = *comm-semiring-hom* + *inj-semiring-hom*
begin
 sublocale *inj-comm-monoid-mult-hom*⟨*proof*⟩
end

For groups, injectivity is easily ensured.

locale *inj-group-add-hom* = *group-add-hom* + *zero-hom-0*
begin
 sublocale *injective hom*
 ⟨*proof*⟩
 sublocale *inj-monoid-add-hom*⟨*proof*⟩
end

locale *inj-ab-group-add-hom* = *ab-group-add-hom* + *inj-group-add-hom*
begin
 sublocale *inj-comm-monoid-add-hom*⟨*proof*⟩
end

locale *inj-ring-hom* = *ring-hom* + *zero-hom-0*
begin
 sublocale *inj-ab-group-add-hom*⟨*proof*⟩
 sublocale *inj-semiring-hom*⟨*proof*⟩
end

locale *inj-comm-ring-hom* = *comm-ring-hom* + *zero-hom-0*
begin
 sublocale *inj-ring-hom*⟨*proof*⟩
 sublocale *inj-comm-semiring-hom*⟨*proof*⟩
end

locale *inj-idom-hom* = *idom-hom* + *zero-hom-0*
begin
 sublocale *inj-comm-ring-hom*⟨*proof*⟩
end

Field homomorphism is always injective.

context *field-hom* **begin**

```

sublocale zero-hom-0
  ⟨proof⟩
sublocale inj-idom-hom⟨proof⟩
end

```

5.5 Surjectivity and Isomorphisms

```

locale surjective =
  fixes f :: 'a ⇒ 'b
  assumes surj: surj f
begin
  lemma f-inv-f[simp]: f (inv f x) = x
    ⟨proof⟩
end

```

```

locale bijective = injective + surjective

```

```

lemma bijective-eq-bij: bijective f = bij f
  ⟨proof⟩

```

```

context bijective
begin
  lemmas bij = bijective-axioms[unfolded bijective-eq-bij]
  interpretation inv: bijective inv f
    ⟨proof⟩
  sublocale inv: surjective inv f⟨proof⟩
  sublocale inv: injective inv f⟨proof⟩
  lemma inv-inv-f-eq[simp]: inv (inv f) = f ⟨proof⟩
  lemma f-eq-iff[simp]: f x = y ⟷ x = inv f y ⟨proof⟩
  lemma inv-f-eq-iff[simp]: inv f x = y ⟷ x = f y ⟨proof⟩
end

```

```

locale monoid-mult-isom = inj-monoid-mult-hom + bijective hom
begin
  sublocale inv: bijective inv hom⟨proof⟩
  sublocale inv: inj-monoid-mult-hom inv hom
    ⟨proof⟩
end

```

```

locale monoid-add-isom = inj-monoid-add-hom + bijective hom
begin
  sublocale inv: bijective inv hom⟨proof⟩
  sublocale inv: inj-monoid-add-hom inv hom
    ⟨proof⟩
end

```

```

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

```

```

sublocale inv: monoid-mult-isom inv hom⟨proof⟩
sublocale inj-comm-monoid-mult-hom⟨proof⟩

lemma hom-dvd-hom[simp]: hom x dvd hom y  $\longleftrightarrow$  x dvd y
  ⟨proof⟩

lemma hom-dvd-simp[simp]:
  shows hom x dvd y'  $\longleftrightarrow$  x dvd inv hom y'
  ⟨proof⟩

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 ⟨proof⟩
  sublocale inj-comm-monoid-add-hom⟨proof⟩
end

locale semiring-isom = inj-semiring-hom hom + bijective hom for hom
begin
  sublocale inv: inj-semiring-hom inv hom ⟨proof⟩
  sublocale inv: bijective inv hom⟨proof⟩
  sublocale monoid-mult-isom⟨proof⟩
  sublocale comm-monoid-add-isom⟨proof⟩
end

locale comm-semiring-isom = semiring-isom hom
  for hom :: 'a :: comm-semiring-1  $\Rightarrow$  'b :: comm-semiring-1
begin
  sublocale inv: semiring-isom inv hom ⟨proof⟩
  sublocale comm-monoid-mult-isom⟨proof⟩
  sublocale inj-comm-semiring-hom⟨proof⟩
end

locale ring-isom = inj-ring-hom + surjective hom
begin
  sublocale semiring-isom⟨proof⟩
  sublocale inv: inj-ring-hom inv hom ⟨proof⟩
end

locale comm-ring-isom = ring-isom hom
  for hom :: 'a :: comm-ring-1  $\Rightarrow$  'b :: comm-ring-1
begin
  sublocale comm-semiring-isom⟨proof⟩
  sublocale inj-comm-ring-hom⟨proof⟩
  sublocale inv: ring-isom inv hom ⟨proof⟩
end

```

```

locale idom-isom = comm-ring-isom + inj-idom-hom
begin
  sublocale inv: comm-ring-isom inv hom ⟨proof⟩
  sublocale inv: inj-idom-hom inv hom⟨proof⟩
end

locale field-isom = field-hom + surjective hom
begin
  sublocale idom-isom⟨proof⟩
  sublocale inv: field-hom inv hom ⟨proof⟩
end

locale inj-idom-divide-hom = idom-divide-hom hom + inj-idom-hom hom
  for hom :: 'a :: idom-divide ⇒ 'b :: idom-divide
begin
lemma hom-dvd-iff[simp]: (hom p dvd hom q) = (p dvd q)
  ⟨proof⟩
end

context field-hom
begin
sublocale inj-idom-divide-hom ⟨proof⟩
end

```

5.6 Example Interpretations

```

interpretation of-int-hom: ring-hom of-int ⟨proof⟩
interpretation of-int-hom: comm-ring-hom of-int ⟨proof⟩
interpretation of-int-hom: idom-hom of-int ⟨proof⟩
interpretation of-int-hom: inj-ring-hom of-int :: int ⇒ 'a :: {ring-1,ring-char-0}
  ⟨proof⟩
interpretation of-int-hom: inj-comm-ring-hom of-int :: int ⇒ 'a :: {comm-ring-1,ring-char-0}
  ⟨proof⟩
interpretation of-int-hom: inj-idom-hom of-int :: int ⇒ 'a :: {idom,ring-char-0}
  ⟨proof⟩

```

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

```

interpretation of-rat-hom: field-char-0-hom of-rat
  ⟨proof⟩

interpretation of-real-hom: inj-ring-hom of-real ⟨proof⟩
interpretation of-real-hom: inj-comm-ring-hom of-real ⟨proof⟩
interpretation of-real-hom: inj-idom-hom of-real ⟨proof⟩
interpretation of-real-hom: field-hom of-real ⟨proof⟩
interpretation of-real-hom: field-char-0-hom of-real ⟨proof⟩

```

Constant multiplication in a semiring is only a monoid homomorphism.

```

interpretation mult-hom: comm-monoid-add-hom λx. c * x for c :: 'a :: semiring-1
  ⟨proof⟩

```

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** $x: -1 \leq (x :: 'a :: \text{linordered-field})$

shows $1 + \text{of-nat } n * x \leq (1 + x) ^ n$

<proof>

context

fixes $b :: 'a :: \text{archimedean-field}$

assumes $b: 0 < b \ b < 1$

begin

private lemma *pow-one*: $b ^ x \leq 1$ *<proof>* **lemma** *pow-zero*: $0 < b ^ x$ *<proof>*

lemma *exp-tends-to-zero*: **assumes** $c: c > 0$

shows $\exists x. b ^ x \leq c$

<proof>

lemma *linear-exp-bound*: $\exists p. \forall x. b ^ x * \text{of-nat } x \leq p$

<proof>

lemma *poly-exp-bound*: $\exists p. \forall x. b ^ x * \text{of-nat } x ^ \text{deg} \leq p$

<proof>

end

lemma *prod-list-replicate[simp]*: $\text{prod-list } (\text{replicate } n \ a) = a ^ n$

<proof>

lemma *prod-list-power*: **fixes** $xs :: 'a :: \text{comm-monoid-mult list}$

shows $\text{prod-list } xs ^ n = (\prod x \leftarrow xs. x ^ n)$

<proof>

lemma *set-upt-Suc*: $\{0 ..< \text{Suc } i\} = \text{insert } i \ \{0 ..< i\}$

<proof>

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

<proof>

lemma *dvd-abs-mult-left-int* [simp]:

$|a| * y \text{ dvd } x \longleftrightarrow a * y \text{ dvd } x$ **for** $x \ y \ a :: \text{int}$
(proof)

lemma *gcd-abs-mult-right-int* [simp]:

$\text{gcd } x \ (|a| * y) = \text{gcd } x \ (a * y)$ **for** $x \ y \ a :: \text{int}$
(proof)

lemma *lcm-abs-mult-right-int* [simp]:

$\text{lcm } x \ (|a| * y) = \text{lcm } x \ (a * y)$ **for** $x \ y \ a :: \text{int}$
(proof)

lemma *gcd-abs-mult-left-int* [simp]:

$\text{gcd } x \ (a * |y|) = \text{gcd } x \ (a * y)$ **for** $x \ y \ a :: \text{int}$
(proof)

lemma *lcm-abs-mult-left-int* [simp]:

$\text{lcm } x \ (a * |y|) = \text{lcm } x \ (a * y)$ **for** $x \ y \ a :: \text{int}$
(proof)

abbreviation (input) *list-gcd* :: 'a :: semiring-gcd list \Rightarrow 'a **where**
list-gcd \equiv *gcd-list*

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 # xs) = *gcd* x (*list-gcd* xs)
(proof)

lemma *list-gcd*: $x \in \text{set } xs \Longrightarrow \text{list-gcd } xs \text{ dvd } x$
(proof)

lemma *list-gcd-greatest*: $(\bigwedge x. x \in \text{set } xs \Longrightarrow y \text{ dvd } x) \Longrightarrow y \text{ dvd } (\text{list-gcd } xs)$
(proof)

lemma *list-gcd-mult-int* [simp]:

fixes $xs :: \text{int list}$

shows *list-gcd* (map (times a) xs) = $|a| * \text{list-gcd } xs$
(proof)

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

lemma *list-lcm*: $x \in \text{set } xs \Longrightarrow x \text{ dvd } \text{list-lcm } xs$
(proof)

lemma *list-lcm-least*: $(\bigwedge x. x \in \text{set } xs \implies x \text{ dvd } y) \implies \text{list-lcm } xs \text{ dvd } y$
 ⟨proof⟩

lemma *lcm-mult-distrib-nat*: $(k :: \text{nat}) * \text{lcm } m \ n = \text{lcm } (k * m) \ (k * n)$
 ⟨proof⟩

lemma *lcm-mult-distrib-int*: $\text{abs } (k :: \text{int}) * \text{lcm } m \ n = \text{lcm } (k * m) \ (k * n)$
 ⟨proof⟩

lemma *list-lcm-mult-int [simp]*:
fixes $xs :: \text{int list}$
shows $\text{list-lcm } (\text{map } (\text{times } a) \ xs) = (\text{if } xs = [] \ \text{then } 1 \ \text{else } |a| * \text{list-lcm } xs)$
 ⟨proof⟩

lemma *list-lcm-pos*:
 $\text{list-lcm } xs \geq (0 :: \text{int})$
 $0 \notin \text{set } xs \implies \text{list-lcm } xs \neq 0$
 $0 \notin \text{set } xs \implies \text{list-lcm } xs > 0$
 ⟨proof⟩

lemma *quotient-of-nonzero*: $\text{snd } (\text{quotient-of } r) > 0 \ \text{snd } (\text{quotient-of } r) \neq 0$
 ⟨proof⟩

lemma *quotient-of-int-div*: **assumes** $q: \text{quotient-of } (\text{of-int } x / \text{of-int } y) = (a, b)$
and $y \neq 0$
shows $\exists z. z \neq 0 \wedge x = a * z \wedge y = b * z$
 ⟨proof⟩

fun *max-list-non-empty* :: $(\text{'a} :: \text{linorder}) \ \text{list} \Rightarrow \text{'a}$ **where**
 $\text{max-list-non-empty } [x] = x$
 $|\ \text{max-list-non-empty } (x \# xs) = \text{max } x \ (\text{max-list-non-empty } xs)$

lemma *max-list-non-empty*: $x \in \text{set } xs \implies x \leq \text{max-list-non-empty } xs$
 ⟨proof⟩

lemma *cnj-reals[simp]*: $(\text{cnj } c \in \mathbb{R}) = (c \in \mathbb{R})$
 ⟨proof⟩

lemma *sgn-real-mono*: $x \leq y \implies \text{sgn } x \leq \text{sgn } (y :: \text{real})$
 ⟨proof⟩

lemma *sgn-minus-rat*: $\text{sgn } (- (x :: \text{rat})) = - \text{sgn } x$
 ⟨proof⟩

lemma *real-of-rat-sgn*: $\text{sgn } (\text{of-rat } x) = \text{real-of-rat } (\text{sgn } x)$
 ⟨proof⟩

lemma *inverse-le-iff-sgn*: **assumes** $\text{sgn}: \text{sgn } x = \text{sgn } y$

shows $(\text{inverse } (x :: \text{real}) \leq \text{inverse } y) = (y \leq x)$
<proof>

lemma *inverse-le-sgn*: **assumes** $\text{sgn } x = \text{sgn } y$ **and** $xy: x \leq (y :: \text{real})$
shows $\text{inverse } y \leq \text{inverse } x$
<proof>

lemma *set-list-update*: $\text{set } (xs [i := k]) =$
 $(\text{if } i < \text{length } xs \text{ then insert } k (\text{set } (\text{take } i \text{ } xs) \cup \text{set } (\text{drop } (\text{Suc } i) \text{ } xs)) \text{ else set } xs)$
<proof>

lemma *prod-list-dvd*: **assumes** $(x :: 'a :: \text{comm-monoid-mult}) \in \text{set } xs$
shows $x \text{ dvd prod-list } xs$
<proof>

lemma *dvd-prod*:
fixes $A :: 'b \text{ set}$
assumes $\exists b \in A. a \text{ dvd } f \ b \text{ finite } A$
shows $a \text{ dvd prod } f \ A$
<proof>

context

fixes $xs :: 'a :: \text{comm-monoid-mult list}$

begin

lemma *prod-list-filter*: $\text{prod-list } (\text{filter } f \ xs) * \text{prod-list } (\text{filter } (\lambda x. \neg f \ x) \ xs) =$
 $\text{prod-list } xs$
<proof>

lemma *prod-list-partition*: **assumes** $\text{partition } f \ xs = (ys, zs)$
shows $\text{prod-list } xs = \text{prod-list } ys * \text{prod-list } zs$
<proof>

end

lemma *dvd-imp-mult-div-cancel-left*[*simp*]:
assumes $(a :: 'a :: \text{semidom-divide}) \text{ dvd } b$
shows $a * (b \text{ div } a) = b$
<proof>

lemma (**in** *semidom*) *prod-list-zero-iff*[*simp*]:
 $\text{prod-list } xs = 0 \iff 0 \in \text{set } xs$ *<proof>*

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

lemma *unit-prod* [*intro*]:
shows $a \text{ dvd } 1 \implies b \text{ dvd } 1 \implies (a * b) \text{ dvd } 1$
<proof>

lemma *is-unit-mult-iff*[*simp*]:

```

shows  $(a * b) \text{ dvd } 1 \iff a \text{ dvd } 1 \wedge b \text{ dvd } 1$ 
<proof>

end

context comm-semiring-1
begin
lemma irreducibleE[elim]:
  assumes irreducible p
  and  $p \neq 0 \implies \neg p \text{ dvd } 1 \implies (\bigwedge a b. p = a * b \implies a \text{ dvd } 1 \vee b \text{ dvd } 1) \implies$ 
thesis
  shows thesis <proof>

lemma not-irreducibleE:
  assumes  $\neg \text{irreducible } x$ 
  and  $x = 0 \implies \text{thesis}$ 
  and  $x \text{ dvd } 1 \implies \text{thesis}$ 
  and  $\bigwedge a b. x = a * b \implies \neg a \text{ dvd } 1 \implies \neg b \text{ dvd } 1 \implies \text{thesis}$ 
  shows thesis <proof>

lemma prime-elem-dvd-prod-list:
  assumes p: prime-elem p and pA: p dvd prod-list A shows  $\exists a \in \text{set } A. p \text{ dvd } a$ 
  <proof>

lemma prime-elem-dvd-prod-mset:
  assumes p: prime-elem p and pA: p dvd prod-mset A shows  $\exists a \in \# A. p \text{ dvd } a$ 
  <proof>

lemma mult-unit-dvd-iff[simp]:
  assumes  $b \text{ dvd } 1$ 
  shows  $a * b \text{ dvd } c \iff a \text{ dvd } c$ 
  <proof>

lemma mult-unit-dvd-iff'[simp]:  $a \text{ dvd } 1 \implies (a * b) \text{ dvd } c \iff b \text{ dvd } c$ 
  <proof>

lemma irreducibleD':
  assumes irreducible a b dvd a
  shows  $a \text{ dvd } b \vee b \text{ dvd } 1$ 
  <proof>

end

context idom
begin

```

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

lemma *dvd-times-left-cancel-iff* [simp]:

assumes $a \neq 0$

shows $a * b \text{ dvd } a * c \longleftrightarrow b \text{ dvd } c$

(is ?lhs \longleftrightarrow ?rhs)

<proof>

lemma *dvd-times-right-cancel-iff* [simp]:

assumes $a \neq 0$

shows $b * a \text{ dvd } c * a \longleftrightarrow b \text{ dvd } c$

<proof>

lemma *irreducibleI'*:

assumes $a \neq 0 \wedge \neg a \text{ dvd } 1 \wedge b. b \text{ dvd } a \implies a \text{ dvd } b \vee b \text{ dvd } 1$

shows *irreducible* a

<proof>

lemma *irreducible-altdef*:

shows *irreducible* $x \longleftrightarrow x \neq 0 \wedge \neg x \text{ dvd } 1 \wedge (\forall b. b \text{ dvd } x \longrightarrow x \text{ dvd } b \vee b \text{ dvd } 1)$

<proof>

lemma *dvd-mult-unit-iff*:

assumes $b. b \text{ dvd } 1$

shows $a \text{ dvd } c * b \longleftrightarrow a \text{ dvd } c$

<proof>

lemma *dvd-mult-unit-iff'*: $b \text{ dvd } 1 \implies a \text{ dvd } b * c \longleftrightarrow a \text{ dvd } c$

<proof>

lemma *irreducible-mult-unit-left*:

shows $a \text{ dvd } 1 \implies \text{irreducible } (a * p) \longleftrightarrow \text{irreducible } p$

<proof>

lemma *irreducible-mult-unit-right*:

shows $a \text{ dvd } 1 \implies \text{irreducible } (p * a) \longleftrightarrow \text{irreducible } p$

<proof>

lemma *prime-elem-imp-irreducible*:

assumes *prime-elem* p

shows *irreducible* p

<proof>

lemma *unit-imp-dvd* [dest]: $b \text{ dvd } 1 \implies b \text{ dvd } a$

<proof>

lemma *unit-mult-left-cancel*: $a \text{ dvd } 1 \implies a * b = a * c \longleftrightarrow b = c$

<proof>

lemma *unit-mult-right-cancel*: $a \text{ dvd } 1 \implies b * a = c * a \longleftrightarrow b = c$
<proof>

New parts from here

lemma *irreducible-multD*:
assumes l : *irreducible* $(a*b)$
shows $a \text{ dvd } 1 \wedge \text{irreducible } b \vee b \text{ dvd } 1 \wedge \text{irreducible } a$
<proof>

end

lemma (**in** *field*) *irreducible-field[simp]*:
irreducible $x \longleftrightarrow \text{False}$ *<proof>*

lemma (**in** *idom*) *irreducible-mult*:
shows *irreducible* $(a*b) \longleftrightarrow a \text{ dvd } 1 \wedge \text{irreducible } b \vee b \text{ dvd } 1 \wedge \text{irreducible } a$
<proof>

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** $\text{degree } p = 0$
shows $[: \text{coeff } p \ 0 :] = p$
<proof>

lemma *degree0-coeffs*: $\text{degree } p = 0 \implies$
 $\exists a. p = [: a :]$
<proof>

lemma *degree1-coeffs*: $\text{degree } p = 1 \implies$
 $\exists a \ b. p = [: b, a :] \wedge a \neq 0$
<proof>

lemma *degree2-coeffs*: $\text{degree } p = 2 \implies$
 $\exists a \ b \ c. p = [: c, b, a :] \wedge a \neq 0$
<proof>

lemma *poly-zero*:

fixes $p :: 'a :: \text{comm-ring-1 poly}$

assumes $x: \text{poly } p \ x = 0$ **shows** $p = 0 \longleftrightarrow \text{degree } p = 0$

<proof>

lemma *coeff-monom-Suc*: $\text{coeff } (\text{monom } a \ (\text{Suc } d) * p) \ (\text{Suc } i) = \text{coeff } (\text{monom } a \ d * p) \ i$

<proof>

lemma *coeff-sum-monom*:

assumes $n: n \leq d$

shows $\text{coeff } (\sum_{i \leq d}. \text{monom } (f \ i) \ i) \ n = f \ n$ (**is ?l = -**)

<proof>

lemma *linear-poly-root*: $(a :: 'a :: \text{comm-ring-1}) \in \text{set } as \implies \text{poly } (\prod a \leftarrow as. [:- a, 1:]) \ a = 0$

<proof>

lemma *degree-lcoeff-sum*: **assumes** $\text{deg}: \text{degree } (f \ q) = n$

and fin: *finite* S **and** $q: q \in S$ **and** $\text{degle}: \bigwedge p. p \in S - \{q\} \implies \text{degree } (f \ p) < n$

and cong: $\text{coeff } (f \ q) \ n = c$

shows $\text{degree } (\text{sum } f \ S) = n \wedge \text{coeff } (\text{sum } f \ S) \ n = c$

<proof>

lemma *degree-sum-list-le*: $(\bigwedge p. p \in \text{set } ps \implies \text{degree } p \leq n)$

$\implies \text{degree } (\text{sum-list } ps) \leq n$

<proof>

lemma *degree-prod-list-le*: $\text{degree } (\text{prod-list } ps) \leq \text{sum-list } (\text{map } \text{degree } ps)$

<proof>

lemma *smult-sum*: $\text{smult } (\sum i \in S. f \ i) \ p = (\sum i \in S. \text{smult } (f \ i) \ p)$

<proof>

lemma *range-coeff*: $\text{range } (\text{coeff } p) = \text{insert } 0 \ (\text{set } (\text{coeffs } p))$

<proof>

lemma *smult-power*: $(\text{smult } a \ p) \ ^n = \text{smult } (a \ ^n) \ (p \ ^n)$

<proof>

lemma *poly-sum-list*: $\text{poly } (\text{sum-list } ps) \ x = \text{sum-list } (\text{map } (\lambda p. \text{poly } p \ x) \ ps)$

<proof>

lemma *poly-prod-list*: $\text{poly } (\text{prod-list } ps) \ x = \text{prod-list } (\text{map } (\lambda p. \text{poly } p \ x) \ ps)$

<proof>

lemma *sum-list-neutral*: $(\bigwedge x. x \in \text{set } xs \implies x = 0) \implies \text{sum-list } xs = 0$
 ⟨proof⟩

lemma *prod-list-neutral*: $(\bigwedge x. x \in \text{set } xs \implies x = 1) \implies \text{prod-list } xs = 1$
 ⟨proof⟩

lemma (in *comm-monoid-mult*) *prod-list-map-remove1*:
 $x \in \text{set } xs \implies \text{prod-list } (\text{map } f \text{ } xs) = f \ x * \text{prod-list } (\text{map } f \text{ } (\text{remove1 } x \text{ } xs))$
 ⟨proof⟩

lemma *poly-as-sum*:
 fixes $p :: 'a::\text{comm-semiring-1}$ *poly*
 shows $\text{poly } p \ x = (\sum_{i \leq \text{degree } p} x^i * \text{coeff } p \ i)$
 ⟨proof⟩

lemma *poly-prod-0*: $\text{finite } ps \implies \text{poly } (\text{prod } f \text{ } ps) \ x = (0 :: 'a :: \text{field}) \longleftrightarrow (\exists p \in ps. \text{poly } (f \ p) \ x = 0)$
 ⟨proof⟩

lemma *coeff-monom-mult*:
 shows $\text{coeff } (\text{monom } a \ d * p) \ i =$
 (if $d \leq i$ then $a * \text{coeff } p \ (i-d)$ else 0) (is ?l = ?r)
 ⟨proof⟩

lemma *poly-eqI2*:
 assumes $\text{degree } p = \text{degree } q$ and $\bigwedge i. i \leq \text{degree } p \implies \text{coeff } p \ i = \text{coeff } q \ i$
 shows $p = q$
 ⟨proof⟩

A nice extension rule for polynomials.

lemma *poly-ext[intro]*:
 fixes $p \ q :: 'a :: \{\text{ring-char-0}, \text{idom}\}$ *poly*
 assumes $\bigwedge x. \text{poly } p \ x = \text{poly } q \ x$ shows $p = q$
 ⟨proof⟩

Copied from non-negative variants.

lemma *coeff-linear-power-neg[simp]*:
 fixes $a :: 'a::\text{comm-ring-1}$
 shows $\text{coeff } ([:a, -1:]^{\wedge} n) \ n = (-1)^{\wedge} n$
 ⟨proof⟩

lemma *degree-linear-power-neg[simp]*:
 fixes $a :: 'a::\{\text{idom}, \text{comm-ring-1}\}$
 shows $\text{degree } ([:a, -1:]^{\wedge} n) = n$
 ⟨proof⟩

7.2 Polynomial Composition

lemmas [simp] = *pcompose-pCons*

lemma *pcompose-eq-0*: **fixes** $q :: 'a :: idom\ poly$
assumes $q: degree\ q \neq 0$
shows $p \circ_p q = 0 \iff p = 0$
 $\langle proof \rangle$

declare *degree-pcompose*[simp]

7.3 Monic Polynomials

abbreviation *monic where* $monic\ p \equiv coeff\ p\ (degree\ p) = 1$

lemma *unit-factor-field* [simp]:
 $unit_factor\ (x :: 'a :: \{field,normalization-semidom\}) = x$
 $\langle proof \rangle$

lemma *poly-gcd-monic*:
fixes $p :: 'a :: \{field,factorial-ring-gcd,semiring-gcd-mult-normalize\}\ poly$
assumes $p \neq 0 \vee q \neq 0$
shows $monic\ (gcd\ p\ q)$
 $\langle proof \rangle$

lemma *normalize-monic*: $monic\ p \implies normalize\ p = p$
 $\langle proof \rangle$

lemma *lcoeff-monic-mult*: **assumes** *monic*: $monic\ (p :: 'a :: comm-semiring-1\ poly)$
shows $coeff\ (p * q)\ (degree\ p + degree\ q) = coeff\ q\ (degree\ q)$
 $\langle proof \rangle$

lemma *degree-monic-mult*: **assumes** *monic*: $monic\ (p :: 'a :: comm-semiring-1\ poly)$
and $q: q \neq 0$
shows $degree\ (p * q) = degree\ p + degree\ q$
 $\langle proof \rangle$

lemma *degree-prod-sum-monic*: **assumes**
 $S: finite\ S$
and $nzd: 0 \notin (degree\ o\ f)\ 'S$
and $monic: (\bigwedge a . a \in S \implies monic\ (f\ a))$
shows $degree\ (prod\ f\ S) = (sum\ (degree\ o\ f)\ S) \wedge coeff\ (prod\ f\ S)\ (sum\ (degree\ o\ f)\ S) = 1$
 $\langle proof \rangle$

lemma *degree-prod-monic*:
assumes $\bigwedge i. i < n \implies degree\ (f\ i :: 'a :: comm-semiring-1\ poly) = 1$
and $\bigwedge i. i < n \implies coeff\ (f\ i)\ 1 = 1$
shows $degree\ (prod\ f\ \{0 ..< n\}) = n \wedge coeff\ (prod\ f\ \{0 ..< n\})\ n = 1$
 $\langle proof \rangle$

lemma *degree-prod-sum-lt-n*: **assumes** $\bigwedge i. i < n \implies \text{degree } (f\ i :: 'a :: \text{comm-semiring-1 poly}) \leq 1$

and $i: i < n$ **and** $f\ i: \text{degree } (f\ i) = 0$

shows $\text{degree } (\text{prod } f\ \{0 ..< n\}) < n$

<proof>

lemma *degree-linear-factors*: $\text{degree } (\prod a \leftarrow as. [: f\ a, 1:]) = \text{length } as$

<proof>

lemma *monic-mult*:

fixes $p\ q :: 'a :: \text{idom poly}$

assumes *monic* p *monic* q

shows *monic* $(p * q)$

<proof>

lemma *monic-factor*:

fixes $p\ q :: 'a :: \text{idom poly}$

assumes *monic* $(p * q)$ *monic* p

shows *monic* q

<proof>

lemma *monic-prod*:

fixes $f :: 'a \Rightarrow 'b :: \text{idom poly}$

assumes $\bigwedge a. a \in as \implies \text{monic } (f\ a)$

shows *monic* $(\text{prod } f\ as)$ *<proof>*

lemma *monic-prod-list*:

fixes $as :: 'a :: \text{idom poly list}$

assumes $\bigwedge a. a \in \text{set } as \implies \text{monic } a$

shows *monic* $(\text{prod-list } as)$ *<proof>*

lemma *monic-power*:

assumes *monic* $(p :: 'a :: \text{idom poly})$

shows *monic* $(p \hat{=} n)$

<proof>

lemma *monic-prod-list-pow*: *monic* $(\prod (x :: 'a :: \text{idom}, i) \leftarrow x\ is. [: -\ x, 1:] \hat{=} \text{Suc } i)$

<proof>

lemma *monic-degree-0*: *monic* $p \implies (\text{degree } p = 0) = (p = 1)$

<proof>

7.4 Roots

The following proof structure is completely similar to the one of $?p \neq 0 \implies \text{finite } \{x. \text{poly } ?p\ x = (0 :: ?'a)\}$.

lemma *poly-roots-degree*:

fixes $p :: 'a :: \text{idom poly}$

shows $p \neq 0 \implies \text{card } \{x. \text{poly } p\ x = 0\} \leq \text{degree } p$

<proof>

lemma *poly-root-factor*: $(poly\ ([:\ r,\ 1:] * q)\ (k :: 'a :: idom) = 0) = (k = -r \vee poly\ q\ k = 0)$ (**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>

lemma *poly-root-constant*: $c \neq 0 \implies (poly\ (p * [:\ c:])\ (k :: 'a :: idom) = 0) = (poly\ p\ k = 0)$

<proof>

lemma *poly-linear-exp-linear-factors-rev*:

$([:\ b,1:] \wedge length\ (filter\ ((=)\ b)\ as))\ dvd\ (\prod\ (a :: 'a :: comm-ring-1) \leftarrow as.\ [:\ a, 1:])$

<proof>

lemma *order-max*: **assumes** *dvd*: $[:\ -a, 1:] \wedge k\ dvd\ p$ **and** *p*: $p \neq 0$

shows $k \leq order\ a\ p$

<proof>

7.5 Divisibility

context

assumes *SORT-CONSTRAINT*($'a :: idom$)

begin

lemma *poly-linear-linear-factor*: **assumes**

dvd: $[:\ b,1:]\ dvd\ (\prod\ (a :: 'a) \leftarrow as.\ [:\ a, 1:])$

shows $b \in set\ as$

<proof>

lemma *poly-linear-exp-linear-factors*:

assumes *dvd*: $([:\ b,1:] \wedge^n\ dvd\ (\prod\ (a :: 'a) \leftarrow as.\ [:\ a, 1:]$)

shows $length\ (filter\ ((=)\ b)\ as) \geq n$

<proof>

end

lemma *const-poly-dvd*: $([:\ a:]\ dvd\ [:\ b:]) = (a\ dvd\ b)$

<proof>

lemma *const-poly-dvd-1* [*simp*]:

$[:\ a:]\ dvd\ 1 \iff a\ dvd\ 1$

<proof>

lemma *poly-dvd-1*:

fixes $p :: 'a :: \{comm-semiring-1, semiring-no-zero-divisors\}$ *poly*

shows $p\ dvd\ 1 \iff degree\ p = 0 \wedge coeff\ p\ 0\ dvd\ 1$

<proof>

Degree based version of irreducibility.

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

lemma $irreducible_dI$ [intro]:

assumes 1: $degree p > 0$
and 2: $\bigwedge q r. degree q > 0 \Longrightarrow degree q < degree p \Longrightarrow degree r > 0 \Longrightarrow degree r < degree p \Longrightarrow p = q * r \Longrightarrow False$
shows $irreducible_d p$
 ⟨proof⟩

lemma $irreducible_dI2$:

fixes $p :: 'a :: \{comm-semiring-1, semiring-no-zero-divisors\} poly$
assumes $deg: degree p > 0$ **and** $ndvd: \bigwedge q. degree q > 0 \Longrightarrow degree q \leq degree p \text{ div } 2 \Longrightarrow \neg q \text{ dvd } p$
shows $irreducible_d p$
 ⟨proof⟩

lemma $reducible_dI$:

assumes $degree p > 0 \Longrightarrow \exists q r. degree q < degree p \wedge degree r < degree p \wedge p = q * r$
shows $\neg irreducible_d p$
 ⟨proof⟩

lemma $irreducible_dE$ [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 $thesis$
 ⟨proof⟩

lemma $reducible_dE$ [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 $thesis$
 ⟨proof⟩

lemma $irreducible_dD$:

assumes $irreducible_d p$
shows $degree p > 0 \bigwedge q r. degree q < degree p \Longrightarrow degree r < degree p \Longrightarrow p \neq q * r$
 ⟨proof⟩

theorem $irreducible_d$ -factorization-exists:

assumes $degree p > 0$
shows $\exists fs. fs \neq [] \wedge (\forall f \in set fs. irreducible_d f \wedge degree f \leq degree p) \wedge p =$

prod-list fs
and $\neg \text{irreducible}_d p \implies \exists fs. \text{length } fs > 1 \wedge (\forall f \in \text{set } fs. \text{irreducible}_d f \wedge \text{degree } f < \text{degree } p) \wedge p = \text{prod-list } fs$
<proof>

lemma *irreducible_d-factor*:
fixes $p :: 'a::\{\text{comm-semiring-1}, \text{semiring-no-zero-divisors}\}$ *poly*
assumes $\text{degree } p > 0$
shows $\exists q r. \text{irreducible}_d q \wedge p = q * r \wedge \text{degree } r < \text{degree } p$ *<proof>*

context *mult-zero begin*

definition *zero-divisor where zero-divisor* $a \equiv \exists b. b \neq 0 \wedge a * b = 0$

lemma *zero-divisorI[intro]*:
assumes $b \neq 0$ **and** $a * b = 0$ **shows** *zero-divisor a*
<proof>

lemma *zero-divisorE[elim]*:
assumes *zero-divisor a*
and $\bigwedge b. b \neq 0 \implies a * b = 0 \implies \text{thesis}$
shows *thesis*
<proof>

end

lemma *zero-divisor-0[simp]*:
 $\text{zero-divisor } (0 :: 'a::\{\text{mult-zero}, \text{zero-neq-one}\})$
<proof>

lemma *not-zero-divisor-1*:
 $\neg \text{zero-divisor } (1 :: 'a :: \{\text{monoid-mult}, \text{mult-zero}\})$
<proof>

lemma *zero-divisor-iff-eq-0[simp]*:
fixes $a :: 'a :: \{\text{semiring-no-zero-divisors}, \text{zero-neq-one}\}$
shows $\text{zero-divisor } a \longleftrightarrow a = 0$ *<proof>*

lemma *mult-eq-0-not-zero-divisor-left[simp]*:
fixes $a b :: 'a :: \text{mult-zero}$
assumes $\neg \text{zero-divisor } a$
shows $a * b = 0 \longleftrightarrow b = 0$
<proof>

lemma *mult-eq-0-not-zero-divisor-right[simp]*:
fixes $a b :: 'a :: \{\text{ab-semigroup-mult}, \text{mult-zero}\}$
assumes $\neg \text{zero-divisor } b$
shows $a * b = 0 \longleftrightarrow a = 0$
<proof>

lemma *degree-smult-not-zero-divisor-left*[simp]:

assumes \neg *zero-divisor* c

shows $\text{degree } (\text{smult } c \ p) = \text{degree } p$

<proof>

lemma *degree-smult-not-zero-divisor-right*[simp]:

assumes \neg *zero-divisor* (*lead-coeff* p)

shows $\text{degree } (\text{smult } c \ p) = (\text{if } c = 0 \ \text{then } 0 \ \text{else } \text{degree } p)$

<proof>

lemma *irreducible_d-smult-not-zero-divisor-left*:

assumes $c0: \neg$ *zero-divisor* c

assumes $L: \text{irreducible}_d (\text{smult } c \ p)$

shows *irreducible_d* p

<proof>

lemmas *irreducible_d-smultI* =

irreducible_d-smult-not-zero-divisor-left

[**where** $'a = 'a :: \{\text{comm-semiring-1}, \text{semiring-no-zero-divisors}\}$, *simplified*]

lemma *irreducible_d-smult-not-zero-divisor-right*:

assumes $p0: \neg$ *zero-divisor* (*lead-coeff* p) **and** $L: \text{irreducible}_d (\text{smult } c \ p)$

shows *irreducible_d* p

<proof>

lemma *zero-divisor-mult-left*:

fixes $a \ b :: 'a :: \{\text{ab-semigroup-mult}, \text{mult-zero}\}$

assumes *zero-divisor* a

shows *zero-divisor* ($a * b$)

<proof>

lemma *zero-divisor-mult-right*:

fixes $a \ b :: 'a :: \{\text{semigroup-mult}, \text{mult-zero}\}$

assumes *zero-divisor* b

shows *zero-divisor* ($a * b$)

<proof>

lemma *not-zero-divisor-mult*:

fixes $a \ b :: 'a :: \{\text{ab-semigroup-mult}, \text{mult-zero}\}$

assumes \neg *zero-divisor* ($a * b$)

shows \neg *zero-divisor* a **and** \neg *zero-divisor* b

<proof>

lemma *zero-divisor-smult-left*:

assumes *zero-divisor* a

shows *zero-divisor* ($\text{smult } a \ f$)

<proof>

lemma *unit-not-zero-divisor*:
fixes $a :: 'a :: \{\text{comm-monoid-mult}, \text{mult-zero}\}$
assumes $a \text{ dvd } 1$
shows $\neg \text{zero-divisor } a$
 $\langle \text{proof} \rangle$

lemma *linear-irreducible_a*: **assumes** $\text{degree } p = 1$
shows $\text{irreducible}_a p$
 $\langle \text{proof} \rangle$

lemma *irreducible_a-dvd-smult*:
fixes $p :: 'a :: \{\text{comm-semiring-1}, \text{semiring-no-zero-divisors}\}$ *poly*
assumes $\text{degree } p > 0$ $\text{irreducible}_a q$ $p \text{ dvd } q$
shows $\exists c. c \neq 0 \wedge q = \text{smult } c p$
 $\langle \text{proof} \rangle$

7.6 Map over Polynomial Coefficients

lemma *map-poly-simps*:
shows $\text{map-poly } f (p\text{Cons } c p) =$
 $(\text{if } c = 0 \wedge p = 0 \text{ then } 0 \text{ else } p\text{Cons } (f c) (\text{map-poly } f p))$
 $\langle \text{proof} \rangle$

lemma *map-poly-pCons[simp]*:
assumes $c \neq 0 \vee p \neq 0$
shows $\text{map-poly } f (p\text{Cons } c p) = p\text{Cons } (f c) (\text{map-poly } f p)$
 $\langle \text{proof} \rangle$

lemma *map-poly-map-poly*:
assumes $f 0 = 0$
shows $\text{map-poly } f (\text{map-poly } g p) = \text{map-poly } (f \circ g) p$
 $\langle \text{proof} \rangle$

lemma *map-poly-zero*:
assumes $f: \forall c. f c = 0 \longrightarrow c = 0$
shows $[\text{simp}]: \text{map-poly } f p = 0 \longleftrightarrow p = 0$
 $\langle \text{proof} \rangle$

lemma *map-poly-add*:
assumes $h 0 = 0$
and $h\text{-add}: \forall p q. h (p + q) = h p + h q$
shows $\text{map-poly } h (p + q) = \text{map-poly } h p + \text{map-poly } h q$
 $\langle \text{proof} \rangle$

7.7 Morphismic properties of $p\text{Cons } (0::'a)$

lemma *monom-pCons-0-monom*:

$\text{monom } (p\text{Cons } 0 \text{ (monom } a \ n)) \ d = \text{map-poly } (p\text{Cons } 0) \ (\text{monom } (\text{monom } a \ n) \ d)$
 $\langle \text{proof} \rangle$

lemma *pCons-0-add*: $p\text{Cons } 0 \ (p + q) = p\text{Cons } 0 \ p + p\text{Cons } 0 \ q \ \langle \text{proof} \rangle$

lemma *sum-pCons-0-commute*:
 $\text{sum } (\lambda i. \ p\text{Cons } 0 \ (f \ i)) \ S = p\text{Cons } 0 \ (\text{sum } f \ S)$
 $\langle \text{proof} \rangle$

lemma *pCons-0-as-mult*:
fixes $p :: 'a :: \text{comm-semiring-1} \ \text{poly}$
shows $p\text{Cons } 0 \ p = [:0,1:] * p \ \langle \text{proof} \rangle$

7.8 Misc

fun *expand-powers* :: $(\text{nat} \times 'a)\text{list} \Rightarrow 'a \ \text{list}$ **where**
 $\text{expand-powers } [] = []$
 $| \text{expand-powers } ((\text{Suc } n, a) \# ps) = a \# \text{expand-powers } ((n, a) \# ps)$
 $| \text{expand-powers } ((0, a) \# ps) = \text{expand-powers } ps$

lemma *expand-powers*: **fixes** $f :: 'a \Rightarrow 'b :: \text{comm-ring-1}$
shows $(\prod (n, a) \leftarrow n\text{-as. } f \ a \ ^n) = (\prod a \leftarrow \text{expand-powers } n\text{-as. } f \ a)$
 $\langle \text{proof} \rangle$

lemma *poly-smult-zero-iff*: **fixes** $x :: 'a :: \text{idom}$
shows $(\text{poly } (\text{smult } a \ p) \ x = 0) = (a = 0 \vee \text{poly } p \ x = 0)$
 $\langle \text{proof} \rangle$

lemma *poly-prod-list-zero-iff*: **fixes** $x :: 'a :: \text{idom}$
shows $(\text{poly } (\text{prod-list } ps) \ x = 0) = (\exists p \in \text{set } ps. \ \text{poly } p \ x = 0)$
 $\langle \text{proof} \rangle$

lemma *poly-mult-zero-iff*: **fixes** $x :: 'a :: \text{idom}$
shows $(\text{poly } (p * q) \ x = 0) = (\text{poly } p \ x = 0 \vee \text{poly } q \ x = 0)$
 $\langle \text{proof} \rangle$

lemma *poly-power-zero-iff*: **fixes** $x :: 'a :: \text{idom}$
shows $(\text{poly } (p \ ^n) \ x = 0) = (n \neq 0 \wedge \text{poly } p \ x = 0)$
 $\langle \text{proof} \rangle$

lemma *sum-monom-0-iff*: **assumes** $\text{fin}: \text{finite } S$
and $g: \bigwedge i \ j. \ g \ i = g \ j \implies i = j$
shows $\text{sum } (\lambda i. \ \text{monom } (f \ i) \ (g \ i)) \ S = 0 \iff (\forall i \in S. \ f \ i = 0) \ (\text{is } ?l = ?r)$
 $\langle \text{proof} \rangle$

lemma *degree-prod-list-eq*: **assumes** $\bigwedge p. \ p \in \text{set } ps \implies (p :: 'a :: \text{idom} \ \text{poly}) \neq 0$
shows $\text{degree } (\text{prod-list } ps) = \text{sum-list } (\text{map } \text{degree } ps) \ \langle \text{proof} \rangle$

lemma *degree-power-eq*: **assumes** $p: p \neq 0$
shows $\text{degree } (p \wedge n) = \text{degree } (p :: 'a :: \text{idom poly}) * n$
 $\langle \text{proof} \rangle$

lemma *coeff-Poly*: $\text{coeff } (\text{Poly } xs) i = (\text{nth-default } 0 \ xs \ i)$
 $\langle \text{proof} \rangle$

lemma *rsquarefree-def'*: $\text{rsquarefree } p = (p \neq 0 \wedge (\forall a. \text{order } a \ p \leq 1))$
 $\langle \text{proof} \rangle$

lemma *order-prod-list*: $(\bigwedge p. p \in \text{set } ps \implies p \neq 0) \implies \text{order } x \ (\text{prod-list } ps) =$
 $\text{sum-list } (\text{map } (\text{order } x) \ ps)$
 $\langle \text{proof} \rangle$

lemma *irreducible_a-dvd-eq*:
fixes $a \ b :: 'a :: \{\text{comm-semiring-1, semiring-no-zero-divisors}\} \ \text{poly}$
assumes $\text{irreducible}_a \ a$ **and** $\text{irreducible}_a \ b$
and $a \ \text{dvd } b$
and $\text{monic } a$ **and** $\text{monic } b$
shows $a = b$
 $\langle \text{proof} \rangle$

lemma *monic-gcd-dvd*:
assumes $fg: f \ \text{dvd } g$ **and** $mon: \text{monic } f$ **and** $gcd: \text{gcd } g \ h \in \{1, g\}$
shows $\text{gcd } f \ h \in \{1, f\}$
 $\langle \text{proof} \rangle$

lemma *monom-power*: $(\text{monom } a \ b) \wedge n = \text{monom } (a \wedge n) \ (b * n)$
 $\langle \text{proof} \rangle$

lemma *poly-const-pow*: $[:a:] \wedge b = [:a \wedge b:]$
 $\langle \text{proof} \rangle$

lemma *degree-pderiv-le*: $\text{degree } (\text{pderiv } f) \leq \text{degree } f - 1$
 $\langle \text{proof} \rangle$

lemma *map-div-is-smult-inverse*: $\text{map-poly } (\lambda x. x / (a :: 'a :: \text{field})) \ p = \text{smult}$
 $(\text{inverse } a) \ p$
 $\langle \text{proof} \rangle$

lemma *normalize-poly-old-def*:
 $\text{normalize } (f :: 'a :: \{\text{normalization-semidom, field}\} \ \text{poly}) = \text{smult } (\text{inverse } (\text{unit-factor}$
 $(\text{lead-coeff } f))) \ f$
 $\langle \text{proof} \rangle$

lemma *poly-dvd-antisym*:
fixes $p \ q :: 'b :: \text{idom poly}$

assumes *coeff*: $\text{coeff } p \text{ (degree } p) = \text{coeff } q \text{ (degree } q)$
assumes *dvd1*: $p \text{ dvd } q$ **and** *dvd2*: $q \text{ dvd } p$ **shows** $p = q$
 $\langle \text{proof} \rangle$

lemma *coeff-f-0-code*[*code-unfold*]: $\text{coeff } f \ 0 = (\text{case } \text{coeffs } f \text{ of } [] \Rightarrow 0 \mid x \# - \Rightarrow x)$
 $\langle \text{proof} \rangle$

lemma *poly-compare-0-code*[*code-unfold*]: $(f = 0) = (\text{case } \text{coeffs } f \text{ of } [] \Rightarrow \text{True} \mid - \Rightarrow \text{False})$
 $\langle \text{proof} \rangle$

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

definition *leading-coeff*
where [*code-abbrev*, *simp*]: $\text{leading-coeff} = \text{lead-coeff}$

lemma *leading-coeff-code* [*code*]:
 $\text{leading-coeff } f = (\text{let } xs = \text{coeffs } f \text{ in if } xs = [] \text{ then } 0 \text{ else last } xs)$
 $\langle \text{proof} \rangle$

lemma *nth-coeffs-coeff*: $i < \text{length } (\text{coeffs } f) \implies \text{coeffs } f ! i = \text{coeff } f \ i$
 $\langle \text{proof} \rangle$

definition *monom-mult* :: $\text{nat} \Rightarrow 'a :: \text{comm-semiring-1 } \text{poly} \Rightarrow 'a \ \text{poly}$
where $\text{monom-mult } n \ f = \text{monom } 1 \ n * f$

lemma *monom-mult-unfold* [*code-unfold*]:
 $\text{monom } 1 \ n * f = \text{monom-mult } n \ f$
 $f * \text{monom } 1 \ n = \text{monom-mult } n \ f$
 $\langle \text{proof} \rangle$

lemma *monom-mult-code* [*code abstract*]:
 $\text{coeffs } (\text{monom-mult } n \ f) = (\text{let } xs = \text{coeffs } f \text{ in if } xs = [] \text{ then } xs \text{ else replicate } n \ 0 \ @ \ xs)$
 $\langle \text{proof} \rangle$

lemma *coeff-pcompose-monom*: **fixes** $f :: 'a :: \text{comm-ring-1 } \text{poly}$
assumes $n: j < n$
shows $\text{coeff } (f \circ_p \text{monom } 1 \ n) \ (n * i + j) = (\text{if } j = 0 \text{ then } \text{coeff } f \ i \text{ else } 0)$
 $\langle \text{proof} \rangle$

lemma *coeff-pcompose-x-pow-n*: **fixes** $f :: 'a :: \text{comm-ring-1 } \text{poly}$
assumes $n: n \neq 0$
shows $\text{coeff } (f \circ_p \text{monom } 1 \ n) \ (n * i) = \text{coeff } f \ i$
 $\langle \text{proof} \rangle$

lemma *dvd-dvd-smult*: $a \text{ dvd } b \implies f \text{ dvd } g \implies \text{smult } a \ f \text{ dvd } \text{smult } b \ g$
 $\langle \text{proof} \rangle$

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

lemma *smult-map-poly*: smult a = map-poly ((*) a)
 <proof>

lemma *smult-exact-div-poly*: **assumes** \bigwedge c. c \in set (coeffs p) \implies a dvd c
shows smult a (div-poly p a) = p
 <proof>

lemma *coeff-div-poly*: coeff (div-poly f a) n = coeff f n div a
 <proof>

lemma *poly-pinfty-ge*:
fixes p :: real poly
assumes lead-coeff p > 0 degree p \neq 0
shows \exists n. \forall x \geq n. poly p x \geq b
 <proof>

lemma *pderiv-sum*: pderiv (sum f I) = sum (λ i. (pderiv (f i))) I
 <proof>

lemma *smult-sum2*: smult m (\sum i \in S. f i) = (\sum i \in S. smult m (f i))
 <proof>

lemma *degree-mult-not-eq*:
 degree (f * g) \neq degree f + degree g \implies lead-coeff f * lead-coeff g = 0
 <proof>

lemma *irreducible_d-multD*:
fixes a b :: 'a :: {comm-semiring-1, semiring-no-zero-divisors} poly
assumes l: irreducible_d (a*b)
shows degree a = 0 \wedge a \neq 0 \wedge irreducible_d b \vee degree b = 0 \wedge b \neq 0 \wedge
 irreducible_d a
 <proof>

lemma *irreducible-connect-field[simp]*:
fixes f :: 'a :: field poly
shows irreducible_d f = irreducible f (is ?l = ?r)
 <proof>

lemma *is-unit-field-poly[simp]*:
fixes p :: 'a::field poly
shows is-unit p \iff p \neq 0 \wedge degree p = 0
 <proof>

lemma *irreducible-smult-field[simp]*:
fixes c :: 'a :: field
shows irreducible (smult c p) \iff c \neq 0 \wedge irreducible p (is ?L \iff ?R)

<proof>

lemma *irreducible-monic-factor*: **fixes** $p :: 'a :: \text{field poly}$
assumes $\text{degree } p > 0$
shows $\exists q r. \text{irreducible } q \wedge p = q * r \wedge \text{monic } q$
<proof>

lemma *monic-irreducible-factorization*: **fixes** $p :: 'a :: \text{field poly}$
shows $\text{monic } p \implies$
 $\exists \text{ as } f. \text{finite as} \wedge p = \text{prod } (\lambda a. a \wedge \text{Suc } (f a)) \text{ as} \wedge \text{as} \subseteq \{q. \text{irreducible } q \wedge \text{monic } q\}$
<proof>

lemma *monic-irreducible-gcd*:
 $\text{monic } (f :: 'a :: \{\text{field, euclidean-ring-gcd, semiring-gcd-mult-normalize, normalization-euclidean-semiring-multiplicative}\} \text{ poly}) \implies$
 $\text{irreducible } f \implies \text{gcd } f \ u \in \{1, f\}$
<proof>
end

8 Connecting Polynomials with Homomorphism Locales

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

poly as a homomorphism. Note that types differ.

interpretation *poly-hom*: *comm-semiring-hom* $\lambda p. \text{poly } p \ a$ *<proof>*

interpretation *poly-hom*: *comm-ring-hom* $\lambda p. \text{poly } p \ a$ *<proof>*

interpretation *poly-hom*: *idom-hom* $\lambda p. \text{poly } p \ a$ *<proof>*

(\circ_p) as a homomorphism.

interpretation *pcompose-hom*: *comm-semiring-hom* $\lambda q. q \circ_p p$
<proof>

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

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

definition *eval-poly* :: $('a \Rightarrow 'b :: \text{comm-semiring-1}) \Rightarrow 'a :: \text{zero poly} \Rightarrow 'b \Rightarrow 'b$
where

[code del]: $eval_poly\ h\ p = poly\ (map_poly\ h\ p)$

lemma *eval-poly-code*[code]: $eval_poly\ h\ p\ x = fold_coeffs\ (\lambda\ a\ b.\ h\ a + x * b)\ p\ 0$
⟨proof⟩

lemma *eval-poly-as-sum*:
fixes $h :: 'a :: zero \Rightarrow 'b :: comm_semiring-1$
assumes $h\ 0 = 0$
shows $eval_poly\ h\ p\ x = (\sum\ i \leq degree\ p.\ x^i * h\ (coeff\ p\ i))$
⟨proof⟩

lemma *coeff-const*: $coeff\ [:\ a :]\ i = (if\ i = 0\ then\ a\ else\ 0)$
⟨proof⟩

lemma *x-as-monom*: $[:0,1:] = monom\ 1\ 1$
⟨proof⟩

lemma *x-pow-n*: $monom\ 1\ 1^{\wedge}\ n = monom\ 1\ n$
⟨proof⟩

lemma *map-poly-eval-poly*: **assumes** $h0: h\ 0 = 0$
shows $map_poly\ h\ p = eval_poly\ (\lambda\ a.\ [:\ h\ a :])\ p\ [:0,1:]$ (**is** $?mp = ?ep$)
⟨proof⟩

lemma *smult-as-map-poly*: $smult\ a = map_poly\ ((*)\ a)$
⟨proof⟩

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]: $map_poly\ hom\ (monom\ a\ i) = monom\ (hom\ a)\ i$
⟨proof⟩

lemma *coeff-map-poly-hom*[simp]: $coeff\ (map_poly\ hom\ p)\ i = hom\ (coeff\ p\ i)$
⟨proof⟩

end

locale *map-poly-zero-hom* = *base: zero-hom*

begin

sublocale *zero-hom map-poly hom* ⟨proof⟩

end

map-poly preserves homomorphisms over addition.

context *comm-monoid-add-hom*

begin

lemma *map-poly-hom-add*[*hom-distrib*]:
 $map_poly\ hom\ (p + q) = map_poly\ hom\ p + map_poly\ hom\ q$
⟨proof⟩

end

```
locale map-poly-comm-monoid-add-hom = base: comm-monoid-add-hom  
begin  
  sublocale comm-monoid-add-hom map-poly hom  $\langle$ proof $\rangle$   
end
```

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

```
context comm-semiring-hom begin  
  lemma map-poly-pCons-hom[hom-distrib]: map-poly hom (pCons a p) = pCons  
  (hom a) (map-poly hom p)  
   $\langle$ proof $\rangle$   
  lemma map-poly-hom-smult[hom-distrib]:  
  map-poly hom (smult c p) = smult (hom c) (map-poly hom p)  
   $\langle$ proof $\rangle$   
  lemma poly-map-poly[simp]: poly (map-poly hom p) (hom x) = hom (poly p x)  
   $\langle$ proof $\rangle$   
end
```

```
locale map-poly-comm-semiring-hom = base: comm-semiring-hom  
begin  
  sublocale map-poly-comm-monoid-add-hom  $\langle$ proof $\rangle$   
  sublocale comm-semiring-hom map-poly hom  
   $\langle$ proof $\rangle$   
end
```

```
locale map-poly-comm-ring-hom = base: comm-ring-hom  
begin  
  sublocale map-poly-comm-semiring-hom  $\langle$ proof $\rangle$   
  sublocale comm-ring-hom map-poly hom  $\langle$ proof $\rangle$   
end
```

```
locale map-poly-idom-hom = base: idom-hom  
begin  
  sublocale map-poly-comm-ring-hom  $\langle$ proof $\rangle$   
  sublocale idom-hom map-poly hom  $\langle$ proof $\rangle$   
end
```

8.1.1 Injectivity

```
locale map-poly-inj-zero-hom = base: inj-zero-hom  
begin  
  sublocale inj-zero-hom map-poly hom  
   $\langle$ proof $\rangle$   
end
```

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

```

sublocale map-poly-comm-monoid-add-hom⟨proof⟩
sublocale map-poly-inj-zero-hom⟨proof⟩
sublocale inj-comm-monoid-add-hom map-poly hom⟨proof⟩
end

```

```

locale map-poly-inj-comm-semiring-hom = base: inj-comm-semiring-hom
begin
  sublocale map-poly-comm-semiring-hom⟨proof⟩
  sublocale map-poly-inj-zero-hom⟨proof⟩
  sublocale inj-comm-semiring-hom map-poly hom⟨proof⟩
end

```

```

locale map-poly-inj-comm-ring-hom = base: inj-comm-ring-hom
begin
  sublocale map-poly-inj-comm-semiring-hom⟨proof⟩
  sublocale inj-comm-ring-hom map-poly hom⟨proof⟩
end

```

```

locale map-poly-inj-idom-hom = base: inj-idom-hom
begin
  sublocale map-poly-inj-comm-ring-hom⟨proof⟩
  sublocale inj-idom-hom map-poly hom⟨proof⟩
end

```

```

lemma degree-map-poly-le: degree (map-poly f p) ≤ degree p
  ⟨proof⟩

```

```

lemma coeffs-map-poly:
  assumes f (lead-coeff p) = 0 ↔ p = 0
  shows coeffs (map-poly f p) = map f (coeffs p)
  ⟨proof⟩

```

```

lemma degree-map-poly:
  assumes f (lead-coeff p) = 0 ↔ p = 0
  shows degree (map-poly f p) = degree p
  ⟨proof⟩

```

```

context zero-hom-0 begin

```

```

  lemma degree-map-poly-hom[simp]: degree (map-poly hom p) = degree p
  ⟨proof⟩

```

```

  lemma coeffs-map-poly-hom[simp]: coeffs (map-poly hom p) = map hom (coeffs p)
  ⟨proof⟩

```

```

  lemma hom-lead-coeff[simp]: lead-coeff (map-poly hom p) = hom (lead-coeff p)
  ⟨proof⟩

```

```

end

```

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

interpretation *map-poly-hom*: *map-poly-comm-semiring-hom*⟨proof⟩

lemma *poly-map-poly-0[simp]*:

poly (map-poly hom p) 0 = hom (poly p 0) (**is** ?l = ?r)
⟨proof⟩

lemma *poly-map-poly-1[simp]*:

poly (map-poly hom p) 1 = hom (poly p 1) (**is** ?l = ?r)
⟨proof⟩

lemma *map-poly-hom-as-monom-sum*:

$(\sum j \leq \text{degree } p. \text{monom } (\text{hom } (\text{coeff } p \ j)) \ j) = \text{map-poly hom } p$
⟨proof⟩

lemma *map-poly-pcompose[hom-distrib]*:

map-poly hom (f ∘_p g) = map-poly hom f ∘_p map-poly hom g
⟨proof⟩

end

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

lemma *eval-poly-0[simp]*: *eval-poly hom 0 x = 0* ⟨proof⟩

lemma *eval-poly-monom*: *eval-poly hom (monom a n) x = hom a * x ^ n*
⟨proof⟩

lemma *poly-map-poly-eval-poly*: *poly (map-poly hom p) = eval-poly hom p*
⟨proof⟩

lemma *map-poly-eval-poly*:

map-poly hom p = eval-poly (λ a. [: hom a :]) p [:0,1:]
⟨proof⟩

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

shows $(\sum i \leq \text{degree } p. x ^ i * \text{hom } (\text{coeff } p \ i))$
 $= (\sum i \leq n. x ^ i * \text{hom } (\text{coeff } p \ i))$ (**is** ?l = ?r)
⟨proof⟩

lemma *eval-poly-add[simp]*: *eval-poly hom (p + q) x = eval-poly hom p x + eval-poly hom q x*
⟨proof⟩

lemma *eval-poly-sum*: *eval-poly hom (∑ k ∈ A. p k) x = (∑ k ∈ A. eval-poly hom (p k) x)*
⟨proof⟩

lemma *eval-poly-poly*: $eval\text{-}poly\ hom\ p\ (hom\ x) = hom\ (poly\ p\ x)$
 ⟨*proof*⟩

end

context *comm-ring-hom* **begin**

interpretation *map-poly-hom*: $map\text{-}poly\text{-}comm\text{-}ring\text{-}hom$ ⟨*proof*⟩

lemma *pseudo-divmod-main-hom*:

$pseudo\text{-}divmod\text{-}main\ (hom\ lc)\ (map\text{-}poly\ hom\ q)\ (map\text{-}poly\ hom\ r)\ (map\text{-}poly\ hom\ d)\ dr\ i =$
 $map\text{-}prod\ (map\text{-}poly\ hom)\ (map\text{-}poly\ hom)\ (pseudo\text{-}divmod\text{-}main\ lc\ q\ r\ d\ dr\ i)$
 ⟨*proof*⟩

end

lemma(**in** *inj-comm-ring-hom*) *pseudo-divmod-hom*:

$pseudo\text{-}divmod\ (map\text{-}poly\ hom\ p)\ (map\text{-}poly\ hom\ q) =$
 $map\text{-}prod\ (map\text{-}poly\ hom)\ (map\text{-}poly\ hom)\ (pseudo\text{-}divmod\ p\ q)$
 ⟨*proof*⟩

lemma(**in** *inj-idom-hom*) *pseudo-mod-hom*:

$pseudo\text{-}mod\ (map\text{-}poly\ hom\ p)\ (map\text{-}poly\ hom\ q) = map\text{-}poly\ hom\ (pseudo\text{-}mod\ p\ q)$
 ⟨*proof*⟩

lemma(**in** *idom-hom*) *map-poly-pderiv[hom-distrib]*:

$map\text{-}poly\ hom\ (pderiv\ p) = pderiv\ (map\text{-}poly\ hom\ p)$
 ⟨*proof*⟩

context *field-hom*

begin

lemma *map-poly-pdivmod[hom-distrib]*:

$map\text{-}prod\ (map\text{-}poly\ hom)\ (map\text{-}poly\ hom)\ (p\ div\ q,\ p\ mod\ q) =$
 $(map\text{-}poly\ hom\ p\ div\ map\text{-}poly\ hom\ q,\ map\text{-}poly\ hom\ p\ mod\ map\text{-}poly\ hom\ q)$
 (**is** $?l = ?r$)
 ⟨*proof*⟩

lemma *map-poly-div[hom-distrib]*: $map\text{-}poly\ hom\ (p\ div\ q) = map\text{-}poly\ hom\ p\ div$

$map\text{-}poly\ hom\ q$
 ⟨*proof*⟩

lemma *map-poly-mod[hom-distrib]*: $map\text{-}poly\ hom\ (p\ mod\ q) = map\text{-}poly\ hom\ p$

$mod\ map\text{-}poly\ hom\ q$
 ⟨*proof*⟩

end

locale *field-hom'* = *field-hom hom*

for $hom :: 'a :: \{field-gcd\} \Rightarrow 'b :: \{field-gcd\}$
begin

lemma $map-poly-normalize[hom-distrib]: map-poly\ hom\ (normalize\ p) = normalize\ (map-poly\ hom\ p)$
 $\langle proof \rangle$

lemma $map-poly-gcd[hom-distrib]: map-poly\ hom\ (gcd\ p\ q) = gcd\ (map-poly\ hom\ p)\ (map-poly\ hom\ q)$
 $\langle proof \rangle$

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 $smult-div-poly: assumes\ \bigwedge\ c.\ c \in set\ (coeffs\ p) \implies a\ dvd\ c$
shows $smult\ a\ (div-poly\ a\ p) = p$
 $\langle proof \rangle$

lemma $coeff-div-poly: coeff\ (div-poly\ a\ f)\ n = coeff\ f\ n\ div\ a$
 $\langle proof \rangle$

locale $map-poly-inj-idom-divide-hom = base: inj-idom-divide-hom$
begin

sublocale $map-poly-idom-hom\ \langle proof \rangle$
sublocale $map-poly-inj-zero-hom\ \langle proof \rangle$
sublocale $inj-idom-hom\ map-poly\ hom\ \langle proof \rangle$
lemma $divide-poly-main-hom: defines\ hh \equiv map-poly\ hom$
shows $hh\ (divide-poly-main\ lc\ f\ g\ h\ i\ j) = divide-poly-main\ (hom\ lc)\ (hh\ f)\ (hh\ g)\ (hh\ h)\ i\ j$
 $\langle proof \rangle$

sublocale $inj-idom-divide-hom\ map-poly\ hom$
 $\langle proof \rangle$

lemma $order-hom: order\ (hom\ x)\ (map-poly\ hom\ f) = order\ x\ f$
 $\langle proof \rangle$
end

8.2 Example Interpretations

abbreviation $of-int-poly \equiv map-poly\ of-int$

interpretation $of-int-poly-hom: map-poly-comm-semiring-hom\ of-int\ \langle proof \rangle$

interpretation $of-int-poly-hom: map-poly-comm-ring-hom\ of-int\ \langle proof \rangle$

interpretation $of-int-poly-hom: map-poly-idom-hom\ of-int\ \langle proof \rangle$

interpretation $of-int-poly-hom:$

$map-poly-inj-comm-ring-hom\ of-int :: int \Rightarrow 'a :: \{comm-ring-1, ring-char-0\}$

<proof>

interpretation *of-int-poly-hom:*

map-poly-inj-idom-hom of-int :: int ⇒ 'a :: {idom,ring-char-0} <proof>

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

interpretation *pCons-0-hom: injective pCons 0 <proof>*

interpretation *pCons-0-hom: zero-hom-0 pCons 0 <proof>*

interpretation *pCons-0-hom: inj-comm-monoid-add-hom pCons 0 <proof>*

interpretation *pCons-0-hom: inj-ab-group-add-hom pCons 0 <proof>*

interpretation *monom-hom: injective λx. monom x d <proof>*

interpretation *monom-hom: inj-monoid-add-hom λx. monom x d <proof>*

interpretation *monom-hom: inj-comm-monoid-add-hom λx. monom x d <proof>*

end

9 Newton Interpolation

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

theory *Newton-Interpolation*

imports

HOL-Library.Monad-Syntax

Ring-Hom-Poly

Divmod-Int

Is-Rat-To-Rat

begin

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

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

fun *horner-composition* :: 'a :: comm-ring-1 list ⇒ 'a list ⇒ 'a poly **where**

horner-composition [cn] xis = [:cn:]

| *horner-composition (ci # cs) (xi # xis) = horner-composition cs xis * [:- xi, 1:]*
+ [:ci:]

| *horner-composition - - = 0*

lemma (**in** *map-poly-comm-ring-hom*) *horner-composition-hom:*

horner-composition (map hom cs) (map hom xs) = map-poly hom (horner-composition cs xs)

<proof>

lemma *horner-coeffs-ints: assumes len: length cs ≤ Suc (length ys)*

shows $(\text{set } (\text{coeffs } (\text{horner-composition } cs \ (\text{map } \text{rat-of-int } ys))) \subseteq \mathbf{Z}) = (\text{set } cs \subseteq \mathbf{Z})$
 <proof>

context

fixes

ty :: 'a :: field itself

and *xs* :: 'a list

and *fs* :: 'a list

begin

fun *divided-differences-impl* :: 'a list \Rightarrow 'a \Rightarrow '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 \Rightarrow 'a list \Rightarrow 'a list list **where**

newton-coefficients-main [*fj*] *xjs* = [[*fj*]]

| *newton-coefficients-main* (*fj* # *fjs*) (*xj* # *xjs*) = (

let *rec* = *newton-coefficients-main* *fjs* *xjs*; *row* = hd *rec*;

new-row = *divided-differences-impl* *row* *fj* *xj* *xs*

in *new-row* # *rec*)

| *newton-coefficients-main* - - = []

definition *newton-coefficients* :: 'a list **where**

newton-coefficients = map hd (*newton-coefficients-main* (rev *fs*) (rev *xs*))

definition *newton-poly-impl* :: 'a poly **where**

newton-poly-impl = horner-composition (rev *newton-coefficients*) *xs*

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

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

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

lemma [*simp*]: *xd i i* = 0 *xd i j* + *xd j k* = *xd i k* *xd i j* + *xd k i* = *xd k j*

<proof> **function** *xij-f* :: nat \Rightarrow nat \Rightarrow '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*)

<proof>

termination <proof> **definition** *c* :: nat \Rightarrow 'a **where**

c i = *xij-f* 0 *i*

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

private function *b* :: nat \Rightarrow nat \Rightarrow 'a poly **where**

$b\ i\ n = (if\ i \geq n\ then\ [:c\ n:]\ else\ b\ (Suc\ i)\ n * X\ i + [:c\ i:])$
 $\langle proof \rangle$

termination $\langle proof \rangle$

declare $b.simps[simp\ del]$

definition $newton-poly :: nat \Rightarrow 'a\ poly$ **where**
 $newton-poly\ n = b\ 0\ n$

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

private definition $N\ i = Xij\ 0\ i$

lemma $Xii-1[simp]: Xij\ i\ i = 1$ $\langle proof \rangle$

lemma $smult-1[simp]: smult\ d\ 1 = [:d:]$

$\langle proof \rangle$ **lemma** $newton-poly-sum:$

$newton-poly\ n = sum-list\ (map\ (\lambda\ i.\ smult\ (c\ i)\ (N\ i))\ [0\ ..<\ Suc\ n])$

$\langle proof \rangle$ **lemma** $poly-newton-poly: poly\ (newton-poly\ n)\ y = sum-list\ (map\ (\lambda\ i.\ c\ i * poly\ (N\ i)\ y)\ [0\ ..<\ Suc\ n])$

$\langle proof \rangle$ **definition** $pprod\ k\ i\ j = (\prod\ l \leftarrow [i..<j].\ xd\ k\ l)$

private lemma $poly-N-xi: poly\ (N\ i)\ (x\ j) = pprod\ j\ 0\ i$

$\langle proof \rangle$ **lemma** $poly-N-xi-cond: poly\ (N\ i)\ (x\ j) = (if\ j < i\ then\ 0\ else\ pprod\ j\ 0\ i)$

$\langle proof \rangle$ **lemma** $poly-newton-poly-xj: assumes\ j \leq n$

shows $poly\ (newton-poly\ n)\ (x\ j) = sum-list\ (map\ (\lambda\ i.\ c\ i * poly\ (N\ i)\ (x\ j))\ [0\ ..<\ Suc\ j])$

$\langle proof \rangle$

declare $xij-f.simps[simp\ del]$

context

fixes n

assumes $dist: \bigwedge\ i\ j.\ i < j \implies j \leq n \implies x\ i \neq x\ j$

begin

private lemma $xd-diff: i < j \implies j \leq n \implies xd\ i\ j \neq 0$

$i < j \implies j \leq n \implies xd\ j\ i \neq 0$ $\langle proof \rangle$

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

private lemma $divided-differences-main: assumes\ k \leq n\ i < k$

shows $sum-list\ (map\ (\lambda\ j.\ xij-f\ i\ (i + j) * pprod\ k\ i\ (i + j))\ [0..<Suc\ k - i]) =$

$sum-list\ (map\ (\lambda\ j.\ xij-f\ (Suc\ i)\ (Suc\ i + j) * pprod\ k\ (Suc\ i)\ (Suc\ i + j))\ [0..<Suc\ k - Suc\ i])$

$\langle proof \rangle$ **lemma** $divided-differences: assumes\ kn: k \leq n$ **and** $ik: i \leq k$

shows $sum-list\ (map\ (\lambda\ j.\ xij-f\ i\ (i + j) * pprod\ k\ i\ (i + j))\ [0..<Suc\ k - i]) = f\ k$

$\langle proof \rangle$

lemma $newton-poly-sound: assumes\ k \leq n$

shows $\text{poly } (\text{newton-poly } n) (x \ k) = f \ k$
 $\langle \text{proof} \rangle$
end

lemma *newton-poly-degree*: $\text{degree } (\text{newton-poly } n) \leq n$
 $\langle \text{proof} \rangle$

context

fixes n

assumes xs : $\text{length } xs = n$

and fs : $\text{length } fs = n$

begin

lemma *newton-coefficients-main*:

$k < n \implies \text{newton-coefficients-main } (\text{rev } (\text{map } f \ [0..<Suc \ k])) (\text{rev } (\text{map } x \ [0..<Suc \ k]))$

$= \text{rev } (\text{map } (\lambda \ i. \ \text{map } (\lambda \ j. \ x_{ij} - f \ j \ i) \ [0..<Suc \ i]) \ [0..<Suc \ k])$

$\langle \text{proof} \rangle$

lemma *newton-coefficients*: $\text{newton-coefficients} = \text{rev } (\text{map } c \ [0 \ ..< \ n])$
 $\langle \text{proof} \rangle$

lemma *newton-poly-impl*: **assumes** $n = \text{Suc } nn$

shows $\text{newton-poly-impl} = \text{newton-poly } nn$

$\langle \text{proof} \rangle$

end

end

context

fixes $xs \ fs :: \text{int list}$

begin

fun *divided-differences-impl-int* :: $\text{int list} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{int list} \Rightarrow \text{int list option}$

where

$\text{divided-differences-impl-int } (x_{i-j1} \ \# \ x_{-j1s}) \ f_j \ x_j \ (x_i \ \# \ x_{is}) = ($

$\text{case } \text{divided-differences-impl-int } x_{-j1s} \ f_j \ x_j \ x_{is} \ \text{of } \text{None} \Rightarrow \text{None}$

$| \text{Some } x_{-js} \Rightarrow \text{let } (\text{new}, m) = \text{divmod-int } (\text{hd } x_{-js} - x_{i-j1}) \ (x_j - x_i)$

$\text{in if } m = 0 \text{ then } \text{Some } (\text{new} \ \# \ x_{-js}) \ \text{else } \text{None}$)

$| \text{divided-differences-impl-int } [] \ f_j \ x_j \ x_{is} = \text{Some } [f_j]$

fun *newton-coefficients-main-int* :: $\text{int list} \Rightarrow \text{int list} \Rightarrow \text{int list list option}$ **where**

$\text{newton-coefficients-main-int } [f_j] \ x_{js} = \text{Some } [[f_j]]$

$| \text{newton-coefficients-main-int } (f_j \ \# \ f_{js}) \ (x_j \ \# \ x_{js}) = (\text{do } \{$

$\text{rec} \leftarrow \text{newton-coefficients-main-int } f_{js} \ x_{js};$

$\text{let } \text{row} = \text{hd } \text{rec};$

$\text{new-row} \leftarrow \text{divided-differences-impl-int } \text{row } f_j \ x_j \ x_{js};$

$\text{Some } (\text{new-row} \ \# \ \text{rec})\})$

$| \text{newton-coefficients-main-int } - - = \text{Some } []$

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:*

$length\ gs \leq length\ ys$
 $\implies divided-differences-impl-int\ gs\ g\ x\ ys = Some\ res$
 $\implies divided-differences-impl (map\ rat-of-int\ gs) (rat-of-int\ g) (rat-of-int\ x) (map\ rat-of-int\ ys) = map\ rat-of-int\ res$
 $\wedge length\ res = Suc (length\ gs)$
 $\langle proof \rangle$

lemma *div-Ints-mod-0:* **assumes** $rat-of-int\ a / rat-of-int\ b \in \mathbb{Z}$ $b \neq 0$

shows $a \bmod b = 0$
 $\langle proof \rangle$

lemma *divided-differences-impl-int-None:*

$length\ gs \leq length\ ys$
 $\implies divided-differences-impl-int\ gs\ g\ x\ ys = None$
 $\implies x \notin set (take (length\ gs) ys)$
 $\implies hd (divided-differences-impl (map\ rat-of-int\ gs) (rat-of-int\ g) (rat-of-int\ x) (map\ rat-of-int\ ys)) \notin \mathbb{Z}$
 $\langle proof \rangle$

lemma *newton-coefficients-main-int-Some:*

$length\ gs = length\ ys \implies length\ ys \leq length\ xs$
 $\implies newton-coefficients-main-int\ gs\ ys = Some\ res$
 $\implies newton-coefficients-main (map\ rat-of-int\ xs) (map\ rat-of-int\ gs) (map\ rat-of-int\ ys) = map (map\ rat-of-int) res$
 $\wedge (\forall x \in set\ res. x \neq [] \wedge length\ x \leq length\ ys) \wedge length\ res = length\ gs$
 $\langle proof \rangle$

lemma *newton-coefficients-main-int-None:* **assumes** *dist: distinct xs*

shows $length\ gs = length\ ys \implies length\ ys \leq length\ xs$
 $\implies newton-coefficients-main-int\ gs\ ys = None$
 $\implies ys = drop (length\ xs - length\ ys) (rev\ xs)$
 $\implies \exists row \in set (newton-coefficients-main (map\ rat-of-int\ xs) (map\ rat-of-int\ gs) (map\ rat-of-int\ ys)). hd\ row \notin \mathbb{Z}$
 $\langle proof \rangle$

lemma *newton-coefficients-int:* **assumes** *dist: distinct xs*

and *len: length xs = length fs*
shows $newton-coefficients-int = (let\ cs = newton-coefficients (map\ rat-of-int\ xs) (map\ of-int\ fs)$
 $in\ if\ set\ cs \subseteq \mathbb{Z}\ then\ Some (map\ int-of-rat\ cs) else\ None)$
 $\langle proof \rangle$

definition *newton-poly-impl-int* :: *int poly option where*

$newton-poly-impl-int \equiv case\ newton-coefficients-int\ of\ None \Rightarrow None$

| *Some nc* \Rightarrow *Some (horner-composition (rev nc) xs)*

lemma *newton-poly-impl-int*: **assumes** *len*: *length xs = length fs*
and *dist*: *distinct xs*
shows *newton-poly-impl-int* = (let *p* = *newton-poly-impl* (map *rat-of-int* *xs*) (map
of-int *fs*)
in if set (coeffs *p*) $\subseteq \mathbb{Z}$ then *Some* (map-poly *int-of-rat* *p*) else *None*)
⟨*proof*⟩
end

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

definition *newton-interpolation-poly-int* :: (int \times int)list \Rightarrow 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*⟩

lemma *degree-newton-interpolation-poly*:
shows *degree (newton-interpolation-poly xs-ys)* \leq *length xs-ys - 1*
⟨*proof*⟩

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

lemma *newton-interpolation-poly-int*: **assumes** *dist*: *distinct (map fst xs-ys)*
shows *newton-interpolation-poly-int* *xs-ys* = (let
rxs-ys = map ($\lambda (x,y).$ (*rat-of-int* *x*, *rat-of-int* *y*)) *xs-ys*;
rp = *newton-interpolation-poly* *rxs-ys*
in if ($\forall x \in$ set (coeffs *rp*). *is-int-rat* *x*) then
Some (map-poly *int-of-rat* *rp*) else *None*)
⟨*proof*⟩

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

10 Lagrange Interpolation

We formalized the Lagrange interpolation, i.e., a method to interpolate a polynomial p from a list of points $(x_1, p(x_1)), (x_2, p(x_2)), \dots$. 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 'a \Rightarrow 'a poly **where**
lagrange-basis-poly xs xj \equiv let ys = filter (λ x. x \neq xj) xs
in prod-list (map (λ xi. smult (inverse (xj - xi)) [: - xi, 1 :]) ys)

definition *lagrange-interpolation-poly* :: ('a :: field \times 'a)list \Rightarrow 'a poly **where**
lagrange-interpolation-poly xs-ys \equiv let
xs = map fst xs-ys
in sum-list (map (λ (xj,yj). smult yj (lagrange-basis-poly xs xj)) xs-ys)

lemma [code]:

lagrange-basis-poly xs xj = (let ys = filter (λ x. x \neq xj) xs
in prod-list (map (λ xi. let ii = inverse (xj - xi) in [: - ii * xi, ii :]) ys))
<proof>

lemma *degree-lagrange-basis-poly*: degree (*lagrange-basis-poly* xs xj) \leq length (filter (λ x. x \neq xj) xs)
<proof>

lemma *degree-lagrange-interpolation-poly*:

shows degree (*lagrange-interpolation-poly* xs-ys) \leq length xs-ys - 1
<proof>

lemma *lagrange-basis-poly-1*:

poly (*lagrange-basis-poly* (map fst xs-ys) x) x = 1
<proof>

lemma *lagrange-basis-poly-0*: **assumes** x' \in set (map fst xs-ys) **and** x' \neq x

shows poly (*lagrange-basis-poly* (map fst xs-ys) x) x' = 0
<proof>

lemma *lagrange-interpolation-poly*: **assumes** dist: distinct (map fst xs-ys)

and p: p = *lagrange-interpolation-poly* xs-ys

shows \bigwedge x y. (x,y) \in set xs-ys \implies poly p x = y
<proof>

end

11 Neville Aitken Interpolation

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

theory *Neville-Aitken-Interpolation*

imports

HOL-Computational-Algebra.Polynomial

begin

context

fixes $x :: \text{nat} \Rightarrow 'a :: \text{field}$

and $f :: \text{nat} \Rightarrow 'a$

begin

private definition $X :: \text{nat} \Rightarrow 'a \text{ poly}$ **where** [*code-unfold*]: $X\ i = [:-x\ i, 1:]$

function *neville-aitken-main* $:: \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ poly}$ **where**

neville-aitken-main $i\ j = (\text{if } i < j \text{ then}$
 $(\text{smult } (\text{inverse } (x\ j - x\ i)) (X\ i * \text{neville-aitken-main } (i + 1)\ j -$
 $X\ j * \text{neville-aitken-main } i\ (j - 1)))$
 $\text{else } [f\ i:])$
 $\langle \text{proof} \rangle$

termination $\langle \text{proof} \rangle$

definition *neville-aitken* $:: \text{nat} \Rightarrow 'a \text{ poly}$ **where**

neville-aitken $= \text{neville-aitken-main } 0$

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

lemma *neville-aitken-main*: **assumes** $\text{dist: } \bigwedge i\ j. i < j \implies j \leq n \implies x\ i \neq x\ j$
shows $i \leq k \implies k \leq j \implies j \leq n \implies \text{poly } (\text{neville-aitken-main } i\ j) (x\ k) = (f\ k)$
 $\langle \text{proof} \rangle$

lemma *degree-neville-aitken-main*: $\text{degree } (\text{neville-aitken-main } i\ j) \leq j - i$
 $\langle \text{proof} \rangle$

lemma *degree-neville-aitken*: $\text{degree } (\text{neville-aitken } n) \leq n$
 $\langle \text{proof} \rangle$

fun *neville-aitken-merge* $:: ('a \times 'a \times 'a \text{ poly}) \text{ list} \Rightarrow ('a \times 'a \times 'a \text{ poly}) \text{ list}$ **where**

neville-aitken-merge $((xi, xsj, p-ij) \# (xsi, xsj, p-sisj) \# \text{rest}) =$
 $(xi, xsj, \text{smult } (\text{inverse } (xsj - xi)) ([:-xi, 1:] * p-sisj$
 $+ [xsj, -1:] * p-ij)) \# \text{neville-aitken-merge } ((xsi, xsj, p-sisj) \# \text{rest})$
 $| \text{neville-aitken-merge } [] = []$
 $| \text{neville-aitken-merge } [] = []$

lemma *length-neville-aitken-merge*[*termination-simp*]: $\text{length } (\text{neville-aitken-merge } xs) = \text{length } xs - 1$

<proof>

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

lemma *neville-aitken-merge*:

$xs = \text{map } (\lambda i. (x\ i, x\ (i + j), \text{neville-aitken-main } i\ (i + j)))\ [l \dots Suc\ (l + k)]$

⇒ *neville-aitken-merge* *xs*

= $(\text{map } (\lambda i. (x\ i, x\ (i + Suc\ j), \text{neville-aitken-main } i\ (i + Suc\ j)))\ [l \dots l + k])$

<proof>

lemma *neville-aitken-impl-main*:

$xs = \text{map } (\lambda i. (x\ i, x\ (i + j), \text{neville-aitken-main } i\ (i + j)))\ [l \dots Suc\ (l + k)]$

⇒ *neville-aitken-impl-main* *xs* = *neville-aitken-main* *l* (*l* + *j* + *k*)

<proof>

lemma *neville-aitken-impl*:

$xs = \text{map } (\lambda i. (x\ i, x\ i, [f\ i:]))\ [0 \dots Suc\ k]$

⇒ *neville-aitken-impl-main* *xs* = *neville-aitken* *k*

<proof>

end

lemma *neville-aitken*: **assumes** $\bigwedge i\ j. i < j \implies j \leq n \implies x\ i \neq x\ j$

shows $j \leq n \implies \text{poly } (\text{neville-aitken } x\ f\ n)\ (x\ j) = (f\ j)$

<proof>

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

neville-aitken-interpolation-poly *x-fs* = (let

start = $\text{map } (\lambda (xi, fi). (xi, xi, [fi:]))\ x\text{-fs}$ in

neville-aitken-impl-main *start*)

lemma *neville-aitken-interpolation-impl*: **assumes** *x-fs* ≠ []

shows *neville-aitken-interpolation-poly* *x-fs* =

neville-aitken $(\lambda i. \text{fst } (x\text{-fs } !\ i))\ (\lambda i. \text{snd } (x\text{-fs } !\ i))\ (\text{length } x\text{-fs} - 1)$

<proof>

lemma *neville-aitken-interpolation-poly*: **assumes** *dist*: *distinct* (map *fst* *xs-ys*)

and *p*: *p* = *neville-aitken-interpolation-poly* *xs-ys*

and *xy*: $(x, y) \in \text{set } xs\text{-ys}$

shows *poly* *p* *x* = *y*

<proof>

```

lemma degree-neville-aitken-interpolation-poly:
  shows degree (neville-aitken-interpolation-poly xs-ys) ≤ length xs-ys - 1
  ⟨proof⟩

end

```

12 Polynomial Interpolation

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

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

```

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

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

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

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

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

```

<proof>

lemma *interpolation-poly*: **assumes** *dist*: *distinct (map fst xs-ys)*
and *p*: *p = interpolation-poly alg xs-ys*
and *xy*: $(x,y) \in \text{set } xs-ys$
shows *poly p x = y*
<proof>

lemma *degree-interpolation-poly*:
shows *degree (interpolation-poly alg xs-ys) ≤ length xs-ys - 1*
<proof>

lemma *uniqueness-of-interpolation*: **fixes** *p :: 'a :: idom poly*
assumes *cS*: *card S = Suc n*
and *degree p ≤ n and degree q ≤ n and*
id: $\bigwedge x. x \in S \implies \text{poly } p \ x = \text{poly } q \ x$
shows *p = q*
<proof>

lemma *uniqueness-of-interpolation-point-list*: **fixes** *p :: 'a :: idom poly*
assumes *dist*: *distinct (map fst xs-ys)*
and *p*: $\bigwedge x \ y. (x,y) \in \text{set } xs-ys \implies \text{poly } p \ x = y \ \text{degree } p < \text{length } xs-ys$
and *q*: $\bigwedge x \ y. (x,y) \in \text{set } xs-ys \implies \text{poly } q \ x = y \ \text{degree } q < \text{length } xs-ys$
shows *p = q*
<proof>

lemma *exactly-one-poly-interpolation*: **assumes** *xs*: *xs-ys ≠ [] and dist*: *distinct (map fst xs-ys)*
shows $\exists! p. \text{degree } p < \text{length } xs-ys \wedge (\forall x \ y. (x,y) \in \text{set } xs-ys \longrightarrow \text{poly } p \ x = y)$
(y :: 'a :: field)
<proof>

lemma *interpolation-poly-int-Some*: **assumes** *dist'*: *distinct (map fst xs-ys)*
and *p*: *interpolation-poly-int alg xs-ys = Some p*
shows $\bigwedge x \ y. (x,y) \in \text{set } xs-ys \implies \text{poly } p \ x = y \ \text{degree } p \leq \text{length } xs-ys - 1$
<proof>

lemma *interpolation-poly-int-None*: **assumes** *dist*: *distinct (map fst xs-ys)*
and *p*: *interpolation-poly-int alg xs-ys = None*
and *q*: $\bigwedge x \ y. (x,y) \in \text{set } xs-ys \implies \text{poly } q \ x = y$
and *dq*: *degree q < length xs-ys*
shows *False*
<proof>

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 points 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*: $\neg (\exists p. \text{poly } p \ 0 = 0 \wedge \text{poly } p \ 2 = 1 :: \text{int})$
(*proof*)

end

References

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