Polynomial Interpolation*

René Thiemann and Akihisa Yamada

April 20, 2020

Abstract

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

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

Contents

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

1 Introduction

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

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

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

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

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

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

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

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

2 Conversions to Rational Numbers

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

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

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

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

lemma *of-rat-to-rat*[*simp*]: $x \in \mathbb{Q} \Longrightarrow$ *of-rat* (*to-rat x*) = *x*
  ⟨*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 :: *real*) = ($x \in \mathbb{Q}$)
definition *to-rat-real* (x :: *real*) = (*if* $x \in \mathbb{Q}$ *then* (*THE* y. x = *of-rat* y) *else* *0*)
instance ⟨*proof*⟩
end

lemma *of-nat-complex*: *of-nat* n = *Complex* (*of-nat* n) *0*
⟨*proof*⟩

lemma *of-int-complex*: *of-int* z = *Complex* (*of-int* z) *0*
⟨*proof*⟩

lemma *of-rat-complex*: *of-rat* q = *Complex* (*of-rat* q) *0*
⟨*proof*⟩

lemma *complex-of-real-of-rat*[*simp*]: *complex-of-real* (*real-of-rat* q) = *of-rat* q
⟨*proof*⟩

lemma *is-rat-complex-iff*: $x \in \mathbb{Q} \longleftrightarrow$ *Re* $x \in \mathbb{Q} \wedge$ *Im* x = *0*
⟨*proof*⟩

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

instance ⟨*proof*⟩
end

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

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

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

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

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

z) = *z*
⟨*proof*⟩

lemma *int-of-rat-0*[*simp*]: (*int-of-rat x* = *0*) = (*x* = *0*) ⟨*proof*⟩

end

3 Divmod-Int

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

theory *Divmod-Int*
imports *Main*
begin

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

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

context
includes *integer.lifting*
begin

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

(*divmod-integer* (*integer-of-int m*) (*integer-of-int n*))
⟨*proof*⟩
end

end

4 Improved Code Equations

This theory contains improved code equations for certain algorithms.

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

4.1 *divmod-integer*.

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

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

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

definition *divmod-abs'* **where**
x ≥ 0 ⟹ y ≥ 0 ⟹ divmod-abs' x y = Code-Numeral.divmod-abs x y

lemma *divmod-integer-code''*[*code*]: *divmod-integer k l =*
(if k = 0 then (0, 0)
else if l > 0 then
(if k > 0 then divmod-abs' k l
else case divmod-abs' (− k) l of (r, s) ⇒
if s = 0 then (− r, 0) else (− r − 1, l − s))
else if l = 0 then (0, k)
else apsnd uminus
(if k < 0 then divmod-abs' (−k) (−l)
else case divmod-abs' k (−l) of (r, s) ⇒
if s = 0 then (− r, 0) else (− r − 1, − l − s)))
⟨*proof*⟩

code-printing — FIXME illusion of partiality
constant *divmod-abs'* ⇀
(*SML*) *IntInf.divMod/ (-,/ -)*
and (*Eval*) *Integer.div'-mod/ (-)/ (-)*
and (*OCaml*) *Z.div'-rem*
and (*Haskell*) *divMod/ (-)/ (-)*
and (*Scala*) !((*k*: *BigInt*) => (*l*: *BigInt*) =>/ *if* (*l* == *0*)/ (*BigInt*(*0*), *k*) *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) ⇒ if s = 0 then (− r, 0) else (− r − 1, ?l − s) else if ?l = 0 then (0, ?k) else apsnd uminus (if ?k < 0 then divmod-abs' (− ?k) (− ?l) else case divmod-abs' ?k (− ?l) of (r, s) ⇒ if s = 0 then (− r, 0) else (− r − 1, − ?l − s)))*.

lemma *divmod-nat-code'*[*code*]: *Divides.divmod-nat m n =* (
let k = integer-of-nat m; l = integer-of-nat n
in map-prod nat-of-integer nat-of-integer
(if k = 0 then (0, 0)
else if l = 0 then (0,k) else

divmod-abs′ k l))
⟨*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-distribs*

5.1 Basic Homomorphism Locales

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

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

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

locale *plus-hom* =
fixes *hom* :: ′*a* :: *plus* ⇒ ′*b* :: *plus*

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

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

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

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

Homomorphism distributes over product:

lemma *hom-prod-list*: *hom* (*prod-list xs*) = *prod-list* (*map hom xs*)
⟨*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-distribs*]: *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-distribs*]: *hom* (−*x*) = − *hom x*
⟨*proof*⟩
lemma *hom-minus* [*hom-distribs*]: *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* ⇒ *'b* :: *comm-monoid-mult*
begin
  sublocale *ab-semigroup-mult-hom*⟨*proof*⟩
  lemma *hom-prod*[*hom-distribs*]: *hom* (*prod f X*) = ($\prod$ *x* ∈ *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* ⟹ *hom x dvd 1* ⟨*proof*⟩
end

locale *comm-monoid-add-hom* = *monoid-add-hom hom*
  for *hom* :: *'a* :: *comm-monoid-add* ⇒ *'b* :: *comm-monoid-add*
begin
  sublocale *ab-semigroup-add-hom*⟨*proof*⟩
  lemma *hom-sum*[*hom-distribs*]: *hom* (*sum f X*) = ($\sum$ *x* ∈ *X*. *hom* (*f x*))
    ⟨*proof*⟩
 lemma *hom-sum-mset*[*hom-distribs*,*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* ⇒ *'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* ⇒ *'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* ⇒ *'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* ⇒ *'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* ⇒ *'b* :: *comm-ring-1*
begin
  sublocale *comm-semiring-hom*⟨*proof*⟩
end

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

5.3 Division

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

end

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

  lemma *hom-inverse*[*hom-distribs*]: *hom* (*inverse x*) = *inverse* (*hom x*)
    ⟨*proof*⟩

  sublocale *idom-divide-hom hom*
  ⟨*proof*⟩

end

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

5.4 (Partial) Injectivitiy

locale *zero-hom-0* = *zero-hom* +
  assumes *hom-0*: $\bigwedge x.$ *hom x* = *0* ⟹ *x* = *0*
begin
  lemma *hom-0-iff*[*iff*]: *hom x* = *0* ⟷ *x* = *0* ⟨*proof*⟩
end

locale *one-hom-1* = *one-hom* +
  assumes *hom-1*: $\bigwedge x.$ *hom x* = *1* ⟹ *x* = *1*
begin
  lemma *hom-1-iff*[*iff*]: *hom x* = *1* ⟷ *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* ⇒ *'b* :: *comm-monoid-mult*

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

locale *injective* =
  fixes *f* :: *'a* ⇒ *'b* **assumes** *injectivity*: $\bigwedge x\ y.\ f\ x = f\ y \Longrightarrow x = y$
begin
  lemma *eq-iff* [*simp*]: $f\ x = f\ y \longleftrightarrow x = y$ ⟨*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* ⟷ *x dvd y*
⟨*proof*⟩

lemma *hom-dvd-simp*[*simp*]:
shows *hom x dvd y'* ⟷ *x dvd inv hom y'*
⟨*proof*⟩

end

locale *comm-monoid-add-isom* = *monoid-add-isom hom*
for *hom* :: *'a* :: *comm-monoid-add* ⇒ *'b* :: *comm-monoid-add*
begin
sublocale *inv*: *monoid-add-isom inv hom* ⟨*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* ⇒ *'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* ⇒ *'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* $\lambda x.\ c * x$ **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* ≤ (*x* :: *'a* :: *linordered-field*)
shows *1* + *of-nat n* ∗ *x* ≤ (*1* + *x*) ˆ *n*
⟨*proof*⟩

context
fixes *b* :: *'a* :: *archimedean-field*
assumes *b*: *0* < *b b* < *1*
begin
private lemma *pow-one*: *b* ˆ *x* ≤ *1* ⟨*proof*⟩ **lemma** *pow-zero*: *0* < *b* ˆ *x* ⟨*proof*⟩

lemma *exp-tends-to-zero*: **assumes** *c*: *c* > *0*
shows ∃ *x*. *b* ˆ *x* ≤ *c*
⟨*proof*⟩

lemma *linear-exp-bound*: ∃ *p*. ∀ *x*. *b* ˆ *x* ∗ *of-nat x* ≤ *p*
⟨*proof*⟩

lemma *poly-exp-bound*: ∃ *p*. ∀ *x*. *b* ˆ *x* ∗ *of-nat x* ˆ *deg* ≤ *p*
⟨*proof*⟩
end

lemma *prod-list-replicate*[*simp*]: *prod-list* (*replicate n a*) = *a* ˆ *n*
⟨*proof*⟩

lemma *prod-list-power*: **fixes** *xs* :: *'a* :: *comm-monoid-mult list*
shows *prod-list xs* ˆ *n* = (∏ *x*←*xs*. *x* ˆ *n*)
⟨*proof*⟩

lemma *set-upt-Suc*: {*0* ..< *Suc i*} = *insert i* {*0* ..< *i*}
⟨*proof*⟩

lemma *prod-pow*[*simp*]: (∏ *i* = *0*..<*n*. *p*) = (*p* :: *'a* :: *comm-monoid-mult*) ˆ *n*
⟨*proof*⟩

lemma *dvd-abs-mult-left-int* [*simp*]:
$|a| * y$ *dvd* $x \longleftrightarrow a * y$ *dvd* x **for** $x\ y\ a$ :: *int*
⟨*proof*⟩

lemma *gcd-abs-mult-right-int* [*simp*]:
gcd x $(|a| * y)$ = *gcd* x $(a * y)$ **for** $x\ y\ a$ :: *int*
⟨*proof*⟩

lemma *lcm-abs-mult-right-int* [*simp*]:
lcm x $(|a| * y)$ = *lcm* x $(a * y)$ **for** $x\ y\ a$ :: *int*
⟨*proof*⟩

lemma *gcd-abs-mult-left-int* [*simp*]:
gcd x $(a * |y|)$ = *gcd* x $(a * y)$ **for** $x\ y\ a$ :: *int*
⟨*proof*⟩

lemma *lcm-abs-mult-left-int* [*simp*]:
lcm x $(a * |y|)$ = *lcm* x $(a * y)$ **for** $x\ y\ a$ :: *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*: $x \in$ *set xs* $\Longrightarrow$ *list-gcd xs dvd* x
⟨*proof*⟩

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

lemma *list-gcd-mult-int* [*simp*]:
fixes *xs* :: *int list*
shows *list-gcd* (*map* (*times* a) *xs*) = $|a| *$ *list-gcd xs*
⟨*proof*⟩

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

lemma *list-lcm*: $x \in$ *set xs* $\Longrightarrow$ x *dvd list-lcm xs*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

lemma *sgn-real-mono*: $x \leq y \Longrightarrow sgn\ x \leq sgn\ (y :: real)$
⟨*proof*⟩

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

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

lemma *inverse-le-iff-sgn*: **assumes** *sgn*: $sgn\ x = sgn\ y$
shows $(inverse\ (x :: real) \leq inverse\ y) = (y \leq x)$

⟨*proof*⟩

lemma *inverse-le-sgn*: **assumes** *sgn*: *sgn x* = *sgn y* **and** *xy*: *x* ≤ (*y* :: *real*)
  shows *inverse y* ≤ *inverse x*
  ⟨*proof*⟩

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

lemma *prod-list-dvd*: **assumes** (*x* :: ′*a* :: *comm-monoid-mult*) ∈ *set xs*
  shows *x dvd prod-list xs*
⟨*proof*⟩

lemma *dvd-prod*:
fixes *A*::′*b set*
assumes ∃ *b*∈*A*. *a dvd f b finite A*
shows *a dvd prod f A*
⟨*proof*⟩

context
  fixes *xs* :: ′*a* :: *comm-monoid-mult list*
begin
lemma *prod-list-filter*: *prod-list* (*filter f xs*) ∗ *prod-list* (*filter* (λ *x*. ¬ *f x*) *xs*) = *prod-list xs*
  ⟨*proof*⟩

lemma *prod-list-partition*: **assumes** *partition f xs* = (*ys*, *zs*)
  shows *prod-list xs* = *prod-list ys* ∗ *prod-list zs*
  ⟨*proof*⟩
end

lemma *dvd-imp-mult-div-cancel-left*[*simp*]:
  assumes (*a* :: ′*a* :: *semidom-divide*) *dvd b*
  shows *a* ∗ (*b div a*) = *b*
⟨*proof*⟩

lemma (**in** *semidom*) *prod-list-zero-iff*[*simp*]:
  prod-list xs = *0* ⟷ *0* ∈ *set xs* ⟨*proof*⟩

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

lemma *unit-prod* [*intro*]:
  shows *a dvd 1* ⟹ *b dvd 1* ⟹ (*a* ∗ *b*) *dvd 1*
  ⟨*proof*⟩

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

⟨*proof*⟩

end

context *comm-semiring-1*
begin
lemma *irreducibleE*[*elim*]:
assumes *irreducible p*
and $p \neq 0 \Longrightarrow \neg\ p\ dvd\ 1 \Longrightarrow (\bigwedge a\ b.\ p = a * b \Longrightarrow a\ dvd\ 1 \vee b\ dvd\ 1) \Longrightarrow$ *thesis*
shows *thesis* ⟨*proof*⟩

lemma *not-irreducibleE*:
assumes $\neg$ *irreducible x*
and $x = 0 \Longrightarrow thesis$
and $x\ dvd\ 1 \Longrightarrow thesis$
and $\bigwedge a\ b.\ x = a * b \Longrightarrow \neg\ a\ dvd\ 1 \Longrightarrow \neg\ b\ dvd\ 1 \Longrightarrow thesis$
shows *thesis* ⟨*proof*⟩

lemma *prime-elem-dvd-prod-list*:
assumes *p*: *prime-elem p* **and** *pA*: *p dvd prod-list A* **shows** $\exists\, a \in set\ A.\ p\ dvd\ a$
⟨*proof*⟩

lemma *prime-elem-dvd-prod-mset*:
assumes *p*: *prime-elem p* **and** *pA*: *p dvd prod-mset A* **shows** $\exists\, a \in\#\ A.\ p\ dvd\ a$
⟨*proof*⟩

lemma *mult-unit-dvd-iff*[*simp*]:
assumes *b dvd 1*
shows $a * b\ dvd\ c \longleftrightarrow a\ dvd\ c$
⟨*proof*⟩

lemma *mult-unit-dvd-iff'*[*simp*]: $a\ dvd\ 1 \Longrightarrow (a * b)\ dvd\ c \longleftrightarrow b\ dvd\ c$
⟨*proof*⟩

lemma *irreducibleD'*:
assumes *irreducible a b dvd a*
shows $a\ dvd\ b \vee b\ dvd\ 1$
⟨*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 $a * b$ *dvd* $a * c \longleftrightarrow b$ *dvd* c
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *dvd-times-right-cancel-iff* [*simp*]:
assumes $a \neq 0$
shows $b * a$ *dvd* $c * a \longleftrightarrow b$ *dvd* c
⟨*proof*⟩

lemma *irreducibleI'*:
assumes $a \neq 0$ $\neg$ *a dvd 1* $\bigwedge b.$ *b dvd a* $\Longrightarrow$ *a dvd b* $\vee$ *b dvd 1*
shows *irreducible a*
⟨*proof*⟩

lemma *irreducible-altdef*:
shows *irreducible x* $\longleftrightarrow x \neq 0 \wedge \neg$ *x dvd 1* $\wedge$ ($\forall b.$ *b dvd x* $\longrightarrow$ *x dvd b* $\vee$ *b dvd 1*)
⟨*proof*⟩

lemma *dvd-mult-unit-iff*:
assumes *b*: *b dvd 1*
shows *a dvd* $c * b \longleftrightarrow$ *a dvd c*
⟨*proof*⟩

lemma *dvd-mult-unit-iff'*: *b dvd 1* $\Longrightarrow$ *a dvd* $b * c \longleftrightarrow$ *a dvd c*
⟨*proof*⟩

lemma *irreducible-mult-unit-left*:
shows *a dvd 1* $\Longrightarrow$ *irreducible* $(a * p) \longleftrightarrow$ *irreducible p*
⟨*proof*⟩

lemma *irreducible-mult-unit-right*:
shows *a dvd 1* $\Longrightarrow$ *irreducible* $(p * a) \longleftrightarrow$ *irreducible p*
⟨*proof*⟩

lemma *prime-elem-imp-irreducible*:
assumes *prime-elem p*
shows *irreducible p*
⟨*proof*⟩

lemma *unit-imp-dvd* [*dest*]: *b dvd 1* $\Longrightarrow$ *b dvd a*
⟨*proof*⟩

lemma *unit-mult-left-cancel*: *a dvd 1* $\Longrightarrow a * b = a * c \longleftrightarrow b = c$
⟨*proof*⟩

lemma *unit-mult-right-cancel*: *a dvd 1* $\Longrightarrow$ *b* $*$ *a* = *c* $*$ *a* $\longleftrightarrow$ *b* = *c*
⟨*proof*⟩

New parts from here

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

end

lemma (**in** *field*) *irreducible-field*[*simp*]:
irreducible x $\longleftrightarrow$ *False* ⟨*proof*⟩

lemma (**in** *idom*) *irreducible-mult*:
shows *irreducible* (*a*$*$*b*) $\longleftrightarrow$ *a dvd 1* $\wedge$ *irreducible b* $\vee$ *b dvd 1* $\wedge$ *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** *degree p* = *0*
shows [: *coeff p 0* :] = *p*
⟨*proof*⟩

lemma *degree0-coeffs*: *degree p* = *0* $\Longrightarrow$
$\exists$ *a*. *p* = [: *a* :]
⟨*proof*⟩

lemma *degree1-coeffs*: *degree p* = *1* $\Longrightarrow$
$\exists$ *a b*. *p* = [: *b*, *a* :] $\wedge$ *a* $\neq$ *0*
⟨*proof*⟩

lemma *degree2-coeffs*: *degree p* = *2* $\Longrightarrow$
$\exists$ *a b c*. *p* = [: *c*, *b*, *a* :] $\wedge$ *a* $\neq$ *0*
⟨*proof*⟩

lemma *poly-zero*:
fixes $p :: 'a :: comm\text{-}ring\text{-}1\ poly$
assumes x: $poly\ p\ x = 0$ **shows** $p = 0 \longleftrightarrow degree\ p = 0$
⟨*proof*⟩

lemma *coeff-monom-Suc*: $coeff\ (monom\ a\ (Suc\ d) * p)\ (Suc\ i) = coeff\ (monom\ a\ d * p)\ i$
⟨*proof*⟩

lemma *coeff-sum-monom*:
assumes n: $n \leq d$
shows $coeff\ (\sum i{\leq}d.\ monom\ (f\ i)\ i)\ n = f\ n$ (**is** $?l = \text{-}$)
⟨*proof*⟩

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

lemma *degree-lcoeff-sum*: **assumes** *deg*: $degree\ (f\ q) = n$
and *fin*: $finite\ S$ **and** q: $q \in S$ **and** *degle*: $\bigwedge\ p\ .\ p \in S - \{q\} \Longrightarrow degree\ (f\ p) < n$
and *cong*: $coeff\ (f\ q)\ n = c$
shows $degree\ (sum\ f\ S) = n \wedge coeff\ (sum\ f\ S)\ n = c$
⟨*proof*⟩

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

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

lemma *smult-sum*: $smult\ (\sum i \in S.\ f\ i)\ p = (\sum i \in S.\ smult\ (f\ i)\ p)$
⟨*proof*⟩

lemma *range-coeff*: $range\ (coeff\ p) = insert\ 0\ (set\ (coeffs\ p))$
⟨*proof*⟩

lemma *smult-power*: $(smult\ a\ p)\ \hat{}\ n = smult\ (a\ \hat{}\ n)\ (p\ \hat{}\ n)$
⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma *poly-eqI2*:
assumes $degree\ p = degree\ q$ **and** $\bigwedge i.\ i \le degree\ p \Longrightarrow coeff\ p\ i = coeff\ q\ i$
shows $p = q$
⟨*proof*⟩

A nice extension rule for polynomials.

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

Copied from non-negative variants.

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

lemma *degree-linear-power-neg*[*simp*]:
fixes $a :: 'a::\{idom,comm\text{-}ring\text{-}1\}$
shows $degree\ ([:a,\ -1:]\ \hat{}\ n) = n$
⟨*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* $\longleftrightarrow$ *p* = *0*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *normalize-monic*: *monic p* $\Longrightarrow$ *normalize p* = *p*
⟨*proof*⟩

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

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*
⟨*proof*⟩

lemma *degree-prod-sum-monic*: **assumes**
S: *finite S*
and *nzd*: *0* $\notin$ (*degree o f*) ‘ *S*
and *monic*: ($\bigwedge$ *a* . *a* $\in$ *S* $\Longrightarrow$ *monic* (*f a*))
shows *degree* (*prod f S*) = (*sum* (*degree o f*) *S*) $\wedge$ *coeff* (*prod f S*) (*sum* (*degree o f*) *S*) = *1*
⟨*proof*⟩

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

lemma *degree-prod-sum-lt-n*: **assumes** $\bigwedge$ *i*. *i* < *n* $\Longrightarrow$ *degree* (*f i* :: *'a* :: *comm-semiring-1 poly*) ≤ *1*
and *i*: *i* < *n* **and** *fi*: *degree* (*f i*) = *0*
shows *degree* (*prod f* {*0* ..< *n*}) < *n*
⟨*proof*⟩

lemma *degree-linear-factors*: *degree* ($\prod$ *a* ← *as*. [: *f a*, *1*:]) = *length as*
⟨*proof*⟩

lemma *monic-mult*:
fixes *p q* :: *'a* :: *idom poly*
assumes *monic p monic q*
shows *monic* (*p* ∗ *q*)
⟨*proof*⟩

lemma *monic-factor*:
fixes *p q* :: *'a* :: *idom poly*
assumes *monic* (*p* ∗ *q*) *monic p*
shows *monic q*
⟨*proof*⟩

lemma *monic-prod*:
fixes *f* :: *'a* ⇒ *'b* :: *idom poly*
assumes $\bigwedge$ *a*. *a* ∈ *as* $\Longrightarrow$ *monic* (*f a*)
shows *monic* (*prod f as*) ⟨*proof*⟩

lemma *monic-prod-list*:
fixes *as* :: *'a* :: *idom poly list*
assumes $\bigwedge$ *a*. *a* ∈ *set as* $\Longrightarrow$ *monic a*
shows *monic* (*prod-list as*) ⟨*proof*⟩

lemma *monic-power*:
assumes *monic* (*p* :: *'a* :: *idom poly*)
shows *monic* (*p* ˆ *n*)
⟨*proof*⟩

lemma *monic-prod-list-pow*: *monic* ($\prod$(*x*::*'a*::*idom*, *i*)←*xis*. [:− *x*, *1*:] ˆ *Suc i*)
⟨*proof*⟩

lemma *monic-degree-0*: *monic p* $\Longrightarrow$ (*degree p* = *0*) = (*p* = *1*)
⟨*proof*⟩

7.4 Roots

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

lemma *poly-roots-degree*:
fixes *p* :: *'a*::*idom poly*
shows *p* ≠ *0* $\Longrightarrow$ *card* {*x*. *poly p x* = *0*} ≤ *degree p*

⟨*proof*⟩

lemma *poly-root-factor*: (*poly* ([: *r*, *1*:] ∗ *q*) (*k* :: *'a* :: *idom*) = *0*) = (*k* = −*r* ∨ *poly q k* = *0*) (**is** *?one*)
(*poly* (*q* ∗ [: *r*, *1*:]) *k* = *0*) = (*k* = −*r* ∨ *poly q k* = *0*) (**is** *?two*)
(*poly* [: *r*, *1* :] *k* = *0*) = (*k* = −*r*) (**is** *?three*)
⟨*proof*⟩

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

lemma *poly-linear-exp-linear-factors-rev*:
([:*b*,*1*:])ˆ(*length* (*filter* ((=) *b*) *as*)) *dvd* (∏ (*a* :: *'a* :: *comm-ring-1*) ← *as*. [: *a*, *1*:])
⟨*proof*⟩

lemma *order-max*: **assumes** *dvd*: [: −*a*, *1* :] ˆ *k dvd p* **and** *p*: *p* ≠ *0*
shows *k* ≤ *order a p*
⟨*proof*⟩

7.5 Divisibility

context
assumes *SORT-CONSTRAINT*(*'a* :: *idom*)
begin
lemma *poly-linear-linear-factor*: **assumes**
dvd: [:*b*,*1*:] *dvd* (∏ (*a* :: *'a*) ← *as*. [: *a*, *1*:])
shows *b* ∈ *set as*
⟨*proof*⟩

lemma *poly-linear-exp-linear-factors*:
assumes *dvd*: ([:*b*,*1*:])ˆ*n dvd* (∏ (*a* :: *'a*) ← *as*. [: *a*, *1*:])
shows *length* (*filter* ((=) *b*) *as*) ≥ *n*
⟨*proof*⟩
end

lemma *const-poly-dvd*: ([:*a*:] *dvd* [:*b*:]) = (*a dvd b*)
⟨*proof*⟩

lemma *const-poly-dvd-1* [*simp*]:
[:*a*:] *dvd 1* ⟷ *a dvd 1*
⟨*proof*⟩

lemma *poly-dvd-1*:
fixes *p* :: *'a* :: {*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
shows *p dvd 1* ⟷ *degree p* = *0* ∧ *coeff p 0 dvd 1*
⟨*proof*⟩

Degree based version of irreducibility.

definition *irreducible$_d$* :: *'a* :: *comm-semiring-1 poly* ⇒ *bool* **where**
irreducible$_d$ p = (*degree p* > *0* ∧ (∀ *q r*. *degree q* < *degree p* ⟶ *degree r* < *degree p* ⟶ *p* ≠ *q* ∗ *r*))

lemma *irreducible$_d$I* [*intro*]:
assumes *1*: *degree p* > *0*
and *2*: ⋀*q r*. *degree q* > *0* ⟹ *degree q* < *degree p* ⟹ *degree r* > *0* ⟹ *degree r* < *degree p* ⟹ *p* = *q* ∗ *r* ⟹ *False*
shows *irreducible$_d$ p*
⟨*proof*⟩

lemma *irreducible$_d$I2*:
fixes *p* :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
assumes *deg*: *degree p* > *0* **and** *ndvd*: ⋀ *q*. *degree q* > *0* ⟹ *degree q* ≤ *degree p div 2* ⟹ ¬ *q dvd p*
shows *irreducible$_d$ p*
⟨*proof*⟩

lemma *reducible$_d$I*:
assumes *degree p* > *0* ⟹ ∃ *q r*. *degree q* < *degree p* ∧ *degree r* < *degree p* ∧ *p* = *q* ∗ *r*
shows ¬ *irreducible$_d$ p*
⟨*proof*⟩

lemma *irreducible$_d$E* [*elim*]:
assumes *irreducible$_d$ p*
and *degree p* > *0* ⟹ (⋀*q r*. *degree q* < *degree p* ⟹ *degree r* < *degree p* ⟹ *p* ≠ *q* ∗ *r*) ⟹ *thesis*
shows *thesis*
⟨*proof*⟩

lemma *reducible$_d$E* [*elim*]:
assumes *red*: ¬ *irreducible$_d$ p*
and *1*: *degree p* = *0* ⟹ *thesis*
and *2*: ⋀*q r*. *degree q* > *0* ⟹ *degree q* < *degree p* ⟹ *degree r* > *0* ⟹ *degree r* < *degree p* ⟹ *p* = *q* ∗ *r* ⟹ *thesis*
shows *thesis*
⟨*proof*⟩

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

theorem *irreducible$_d$-factorization-exists*:
assumes *degree p* > *0*
shows ∃*fs*. *fs* ≠ [] ∧ (∀*f* ∈ *set fs*. *irreducible$_d$ f* ∧ *degree f* ≤ *degree p*) ∧ *p* =

prod-list fs
  and $\neg$*irreducible*$_d$ *p* $\Longrightarrow$ $\exists$ *fs*. *length fs* $>$ *1* $\wedge$ ($\forall$ *f* $\in$ *set fs*. *irreducible*$_d$ *f* $\wedge$ *degree f* $<$ *degree p*) $\wedge$ *p* $=$ *prod-list fs*
⟨*proof*⟩

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

end

lemma *zero-divisor-0*[*simp*]:
  zero-divisor (*0*::*'a*::{*mult-zero*,*zero-neq-one*})
  ⟨*proof*⟩

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

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

lemma *mult-eq-0-not-zero-divisor-left*[*simp*]:
  fixes *a b* :: *'a* :: *mult-zero*
  assumes $\neg$ *zero-divisor a*
  shows *a* $*$ *b* $=$ *0* $\longleftrightarrow$ *b* $=$ *0*
  ⟨*proof*⟩

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

lemma *degree-smult-not-zero-divisor-left*[*simp*]:
  assumes ¬ *zero-divisor c*
  shows *degree* (*smult c p*) = *degree p*
⟨*proof*⟩

lemma *degree-smult-not-zero-divisor-right*[*simp*]:
  assumes ¬ *zero-divisor* (*lead-coeff p*)
  shows *degree* (*smult c p*) = (*if c* = *0 then 0 else degree p*)
⟨*proof*⟩

lemma $irreducible_d$*-smult-not-zero-divisor-left*:
  assumes *c0*: ¬ *zero-divisor c*
  assumes *L*: $irreducible_d$ (*smult c p*)
  shows $irreducible_d$ *p*
⟨*proof*⟩

lemmas $irreducible_d$*-smultI* =
  $irreducible_d$*-smult-not-zero-divisor-left*
  [**where** ′*a* = ′*a* :: {*comm-semiring-1*,*semiring-no-zero-divisors*}, *simplified*]

lemma $irreducible_d$*-smult-not-zero-divisor-right*:
  assumes *p0*: ¬ *zero-divisor* (*lead-coeff p*) **and** *L*: $irreducible_d$ (*smult c p*)
  shows $irreducible_d$ *p*
⟨*proof*⟩

lemma *zero-divisor-mult-left*:
  fixes *a b* :: ′*a* :: {*ab-semigroup-mult*, *mult-zero*}
  assumes *zero-divisor a*
  shows *zero-divisor* (*a* ∗ *b*)
⟨*proof*⟩

lemma *zero-divisor-mult-right*:
  fixes *a b* :: ′*a* :: {*semigroup-mult*, *mult-zero*}
  assumes *zero-divisor b*
  shows *zero-divisor* (*a* ∗ *b*)
⟨*proof*⟩

lemma *not-zero-divisor-mult*:
  fixes *a b* :: ′*a* :: {*ab-semigroup-mult*, *mult-zero*}
  assumes ¬ *zero-divisor* (*a* ∗ *b*)
  shows ¬ *zero-divisor a* **and** ¬ *zero-divisor b*
  ⟨*proof*⟩

lemma *zero-divisor-smult-left*:
  assumes *zero-divisor a*
  shows *zero-divisor* (*smult a f*)
⟨*proof*⟩

lemma *unit-not-zero-divisor*:
fixes a :: *'a* :: {*comm-monoid-mult*, *mult-zero*}
assumes *a dvd 1*
shows ¬*zero-divisor a*
⟨*proof*⟩

lemma *linear-irreducible*$_d$: **assumes** *degree p = 1*
shows *irreducible*$_d$ *p*
⟨*proof*⟩

lemma *irreducible*$_d$*-dvd-smult*:
fixes *p* :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
assumes *degree p > 0 irreducible*$_d$ *q p dvd q*
shows ∃ *c*. *c* ≠ *0* ∧ *q* = *smult c p*
⟨*proof*⟩

7.6 Map over Polynomial Coefficients

lemma *map-poly-simps*:
shows *map-poly f* (*pCons c p*) =
(*if c* = *0* ∧ *p* = *0 then 0 else pCons* (*f c*) (*map-poly f p*))
⟨*proof*⟩

lemma *map-poly-pCons*[*simp*]:
assumes *c* ≠ *0* ∨ *p* ≠ *0*
shows *map-poly f* (*pCons c p*) = *pCons* (*f c*) (*map-poly f p*)
⟨*proof*⟩

lemma *map-poly-map-poly*:
assumes *f0*: *f 0* = *0*
shows *map-poly f* (*map-poly g p*) = *map-poly* (*f* ∘ *g*) *p*
⟨*proof*⟩

lemma *map-poly-zero*:
assumes *f*: ∀ *c*. *f c* = *0* ⟶ *c* = *0*
shows [*simp*]: *map-poly f p* = *0* ⟷ *p* = *0*
⟨*proof*⟩

lemma *map-poly-add*:
assumes *h0*: *h 0* = *0*
and *h-add*: ∀ *p q*. *h* (*p* + *q*) = *h p* + *h q*
shows *map-poly h* (*p* + *q*) = *map-poly h p* + *map-poly h q*
⟨*proof*⟩

7.7 Morphismic properties of *pCons* (*0*::*'a*)

lemma *monom-pCons-0-monom*:

monom (*pCons 0* (*monom a n*)) *d* = *map-poly* (*pCons 0*) (*monom* (*monom a n*) *d*)
⟨*proof*⟩

lemma *pCons-0-add*: *pCons 0* (*p* + *q*) = *pCons 0 p* + *pCons 0 q* ⟨*proof*⟩

lemma *sum-pCons-0-commute*:
sum (λ*i*. *pCons 0* (*f i*)) *S* = *pCons 0* (*sum f S*)
⟨*proof*⟩

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

7.8 Misc

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

lemma *expand-powers*: **fixes** *f* :: *'a* ⇒ *'b* :: *comm-ring-1*
shows (∏ (*n*,*a*) ← *n-as*. *f a* ^ *n*) = (∏ *a* ← *expand-powers n-as*. *f a*)
⟨*proof*⟩

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

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

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

lemma *poly-power-zero-iff*: **fixes** *x* :: *'a* :: *idom*
shows (*poly* (*p*^*n*) *x* = *0*) = (*n* ≠ *0* ∧ *poly p x* = *0*)
⟨*proof*⟩

lemma *sum-monom-0-iff*: **assumes** *fin*: *finite S*
and *g*: ⋀ *i j*. *g i* = *g j* ⟹ *i* = *j*
shows *sum* (λ *i*. *monom* (*f i*) (*g i*)) *S* = *0* ⟷ (∀ *i* ∈ *S*. *f i* = *0*) (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *degree-prod-list-eq*: **assumes** ⋀ *p*. *p* ∈ *set ps* ⟹ (*p* :: *'a* :: *idom poly*) ≠ *0*

shows *degree* (*prod-list ps*) = *sum-list* (*map degree ps*) ⟨*proof*⟩

lemma *degree-power-eq*: **assumes** *p*: $p \neq 0$
shows *degree* (p ^ n) = *degree* (p :: $'a$ :: *idom poly*) ∗ n
⟨*proof*⟩

lemma *coeff-Poly*: *coeff* (*Poly xs*) *i* = (*nth-default 0 xs i*)
⟨*proof*⟩

lemma *rsquarefree-def'*: *rsquarefree p* = ($p \neq 0$ ∧ (∀ *a*. *order a p* ≤ *1*))
⟨*proof*⟩

lemma *order-prod-list*: (⋀ *p*. *p* ∈ *set ps* ⟹ $p \neq 0$) ⟹ *order x* (*prod-list ps*) = *sum-list* (*map* (*order x*) *ps*)
⟨*proof*⟩

lemma $irreducible_d$*-dvd-eq*:
fixes *a b* :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
assumes $irreducible_d$ *a* **and** $irreducible_d$ *b*
and *a dvd b*
and *monic a* **and** *monic b*
shows *a* = *b*
⟨*proof*⟩

lemma *monic-gcd-dvd*:
assumes *fg*: *f dvd g* **and** *mon*: *monic f* **and** *gcd*: *gcd g h* ∈ {*1*, *g*}
shows *gcd f h* ∈ {*1*, *f*}
⟨*proof*⟩

lemma *monom-power*: (*monom a b*)^*n* = *monom* (*a*^*n*) (*b*∗*n*)
⟨*proof*⟩

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

lemma *degree-pderiv-le*: *degree* (*pderiv f*) ≤ *degree f* − *1*
⟨*proof*⟩

lemma *map-div-is-smult-inverse*: *map-poly* (λ*x*. *x* / (*a* :: *'a* :: *field*)) *p* = *smult* (*inverse a*) *p*
⟨*proof*⟩

lemma *normalize-poly-old-def*:
normalize (*f* :: *'a* :: {*normalization-semidom*,*field*} *poly*) = *smult* (*inverse* (*unit-factor* (*lead-coeff f*))) *f*
⟨*proof*⟩

lemma *poly-dvd-antisym*:

fixes *p q* :: *'b::idom poly*
assumes *coeff*: *coeff p* (*degree p*) = *coeff q* (*degree q*)
assumes *dvd1*: *p dvd q* **and** *dvd2*: *q dvd p* **shows** *p* = *q*
⟨*proof*⟩

lemma *coeff-f-0-code*[*code-unfold*]: *coeff f 0* = (*case coeffs f of* [] ⇒ *0* | *x* # - ⇒ *x*)
⟨*proof*⟩

lemma *poly-compare-0-code*[*code-unfold*]: (*f* = *0*) = (*case coeffs f of* [] ⇒ *True* | - ⇒ *False*)
⟨*proof*⟩

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

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

lemma *leading-coeff-code* [*code*]:
leading-coeff f = (*let xs* = *coeffs f in if xs* = [] *then 0 else last xs*)
⟨*proof*⟩

lemma *nth-coeffs-coeff*: *i* < *length* (*coeffs f*) ⟹ *coeffs f* ! *i* = *coeff f i*
⟨*proof*⟩

lemma *degree-prod-eq-sum-degree*:
fixes *A* :: *'a set*
and *f* :: *'a* ⇒ *'b::field poly*
assumes *f0*: ∀ *i*∈*A*. *f i* ≠ *0*
shows *degree* (∏ *i*∈*A*. (*f i*)) = (∑ *i*∈*A*. *degree* (*f i*))
⟨*proof*⟩

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

lemma *monom-mult-unfold* [*code-unfold*]:
monom 1 n ∗ *f* = *monom-mult n f*
f ∗ *monom 1 n* = *monom-mult n f*
⟨*proof*⟩

lemma *monom-mult-code* [*code abstract*]:
coeffs (*monom-mult n f*) = (*let xs* = *coeffs f in*
if xs = [] *then xs else replicate n 0* @ *xs*)
⟨*proof*⟩

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

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

lemma *dvd-dvd-smult*: *a dvd b* ⟹ *f dvd g* ⟹ *smult a f dvd smult b g*
⟨*proof*⟩

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

lemma *smult-map-poly*: *smult a* = *map-poly* ((∗) *a*)
⟨*proof*⟩

lemma *smult-exact-sdiv-poly*: **assumes** ⋀ *c. c* ∈ *set* (*coeffs p*) ⟹ *a dvd c*
shows *smult a* (*sdiv-poly p a*) = *p*
⟨*proof*⟩

lemma *coeff-sdiv-poly*: *coeff* (*sdiv-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* ≠ *0*
shows ∃ *n*. ∀ *x* ≥ *n*. *poly p x* ≥ *b*
⟨*proof*⟩

lemma *pderiv-sum*: *pderiv* (*sum f I*) = *sum* (λ *i*. (*pderiv* (*f i*))) *I*
⟨*proof*⟩

lemma *smult-sum2*: *smult m* ($\sum$ *i* ∈ *S*. *f i*) = ($\sum$ *i* ∈ *S*. *smult m* (*f i*))
⟨*proof*⟩

lemma *degree-mult-not-eq*:
degree (*f* ∗ *g*) ≠ *degree f* + *degree g* ⟹ *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* ∧ *a* ≠ *0* ∧ $irreducible_d$ *b* ∨ *degree b* = *0* ∧ *b* ≠ *0* ∧ $irreducible_d$ *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* $p \longleftrightarrow p \neq 0 \wedge$ *degree* $p = 0$
⟨*proof*⟩

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

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

lemma *monic-irreducible-factorization*: **fixes** p :: *'a* :: *field poly*
shows *monic* $p \Longrightarrow$
$\exists$ *as* f. *finite as* $\wedge$ p = *prod* (λ a. a ^ *Suc* (f a)) *as* $\wedge$ *as* $\subseteq$ {q. *irreducible* q $\wedge$ *monic* q}
⟨*proof*⟩

lemma *monic-irreducible-gcd*:
monic (*f*::*'a*::{*field*,*euclidean-ring-gcd*,*semiring-gcd-mult-normalize*,
normalization-euclidean-semiring-multiplicative} *poly*) $\Longrightarrow$
irreducible f $\Longrightarrow$ *gcd f u* $\in$ {*1*,*f*}
⟨*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* λp. *poly* p a ⟨*proof*⟩

interpretation *poly-hom*: *comm-ring-hom* λp. *poly* p a⟨*proof*⟩

interpretation *poly-hom*: *idom-hom* λp. *poly* p a⟨*proof*⟩

($\circ_p$) as a homomorphism.

interpretation *pcompose-hom*: *comm-semiring-hom* λq. $q \circ_p p$
⟨*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* :: ($'a \Rightarrow 'b$ :: *comm-semiring-1*) $\Rightarrow$ $'a$ :: *zero poly* $\Rightarrow$ $'b \Rightarrow 'b$
where
[*code del*]: *eval-poly h p* = *poly* (*map-poly h p*)

lemma *eval-poly-code*[*code*]: *eval-poly h p x* = *fold-coeffs* (λ *a b*. *h a* + *x* ∗ *b*) *p 0*
⟨*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*≤*degree p*. *x*ˆ*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* ˆ *n* = *monom 1 n*
⟨*proof*⟩

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

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-distribs*]:
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* ⟨*proof*⟩
end

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

context *comm-semiring-hom* **begin**
lemma *map-poly-pCons-hom*[*hom-distribs*]: *map-poly hom* (*pCons a p*) = *pCons* (*hom a*) (*map-poly hom p*)
⟨*proof*⟩
lemma *map-poly-hom-smult*[*hom-distribs*]:
map-poly hom (*smult c p*) = *smult* (*hom c*) (*map-poly hom p*)
⟨*proof*⟩
lemma *poly-map-poly*[*simp*]: *poly* (*map-poly hom p*) (*hom x*) = *hom* (*poly p x*)
⟨*proof*⟩
end

locale *map-poly-comm-semiring-hom* = *base*: *comm-semiring-hom*
begin
sublocale *map-poly-comm-monoid-add-hom*⟨*proof*⟩
sublocale *comm-semiring-hom map-poly hom*
⟨*proof*⟩
end

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

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

8.1.1 Injectivity

locale *map-poly-inj-zero-hom* = *base*: *inj-zero-hom*
begin
  sublocale *inj-zero-hom map-poly hom*
  ⟨*proof*⟩
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*≤*degree p*. *monom* (*hom* (*coeff p j*)) *j*) = *map-poly hom p*
⟨*proof*⟩

lemma *map-poly-pcompose*[*hom-distribs*]:
map-poly hom (*f* $\circ_p$ *g*) = *map-poly hom f* $\circ_p$ *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* ˆ *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*≤*degree p*. *x* ˆ *i* ∗ *hom* (*coeff p i*))
= ($\sum$*i*≤*n*. *x* ˆ *i* ∗ *hom* (*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-poly hom p* (*hom x*) = *hom* (*poly p x*)
⟨*proof*⟩

end

context *comm-ring-hom* **begin**
interpretation *map-poly-hom*: *map-poly-comm-ring-hom*⟨*proof*⟩

lemma *pseudo-divmod-main-hom*:
pseudo-divmod-main (*hom lc*) (*map-poly hom q*) (*map-poly hom r*) (*map-poly hom d*) *dr i* =
map-prod (*map-poly hom*) (*map-poly hom*) (*pseudo-divmod-main lc q r d dr i*)
⟨*proof*⟩
end

lemma(**in** *inj-comm-ring-hom*) *pseudo-divmod-hom*:
pseudo-divmod (*map-poly hom p*) (*map-poly hom q*) =
map-prod (*map-poly hom*) (*map-poly hom*) (*pseudo-divmod p q*)
⟨*proof*⟩

lemma(**in** *inj-idom-hom*) *pseudo-mod-hom*:
pseudo-mod (*map-poly hom p*) (*map-poly hom q*) = *map-poly hom* (*pseudo-mod p q*)
⟨*proof*⟩

lemma(**in** *idom-hom*) *map-poly-pderiv*[*hom-distribs*]:
map-poly hom (*pderiv p*) = *pderiv* (*map-poly hom p*)
⟨*proof*⟩

context *field-hom*
begin

lemma *map-poly-pdivmod*[*hom-distribs*]:
map-prod (*map-poly hom*) (*map-poly hom*) (*p div q*, *p mod q*) =
(*map-poly hom p div map-poly hom q*, *map-poly hom p mod map-poly hom q*)
(**is** *?l* = *?r*)
⟨*proof*⟩

lemma *map-poly-div*[*hom-distribs*]: *map-poly hom* (*p div q*) = *map-poly hom p div map-poly hom q*

⟨*proof*⟩

lemma *map-poly-mod*[*hom-distribs*]: *map-poly hom* (*p mod q*) = *map-poly hom p mod map-poly hom q*
⟨*proof*⟩

end

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

lemma *map-poly-normalize*[*hom-distribs*]: *map-poly hom* (*normalize p*) = *normalize* (*map-poly hom p*)
⟨*proof*⟩

lemma *map-poly-gcd*[*hom-distribs*]: *map-poly hom* (*gcd p q*) = *gcd* (*map-poly hom p*) (*map-poly hom q*)
⟨*proof*⟩

end

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

lemma *smult-div-poly*: **assumes** ⋀ *c. c* ∈ *set* (*coeffs p*) ⟹ *a dvd c*
shows *smult a* (*div-poly a p*) = *p*
⟨*proof*⟩

lemma *coeff-div-poly*: *coeff* (*div-poly a f*) *n* = *coeff f n div a*
⟨*proof*⟩

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

sublocale *inj-idom-divide-hom map-poly hom*
⟨*proof*⟩

lemma *order-hom*: *order* (*hom x*) (*map-poly hom f*) = *order x f*
⟨*proof*⟩
end

8.2 Example Interpretations

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

interpretation *of-int-poly-hom*: *map-poly-comm-semiring-hom of-int*⟨*proof*⟩
interpretation *of-int-poly-hom*: *map-poly-comm-ring-hom of-int*⟨*proof*⟩
interpretation *of-int-poly-hom*: *map-poly-idom-hom of-int*⟨*proof*⟩
interpretation *of-int-poly-hom*:
map-poly-inj-comm-ring-hom of-int :: *int* ⇒ *'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* λ*x*. *monom x d* ⟨*proof*⟩
interpretation *monom-hom*: *inj-monoid-add-hom* λ*x*. *monom x d* ⟨*proof*⟩
interpretation *monom-hom*: *inj-comm-monoid-add-hom* λ*x*. *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 $(x_1, p(x_1)), (x_2, p(x_2)), \ldots$. In experiments it performs much faster than the Lagrange interpolation.

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

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

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

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

| *horner-composition - - = 0*

lemma (**in** *map-poly-comm-ring-hom*) *horner-composition-hom*:
horner-composition (*map hom cs*) (*map hom xs*) = *map-poly hom* (*horner-composition cs xs*)
⟨*proof*⟩

lemma *horner-coeffs-ints*: **assumes** *len*: *length cs* ≤ *Suc* (*length ys*)
shows (*set* (*coeffs* (*horner-composition cs* (*map rat-of-int ys*))) ⊆ ℤ) = (*set cs* ⊆ ℤ)
⟨*proof*⟩

context
fixes
ty :: ′*a* :: *field itself*
and *xs* :: ′*a list*
and *fs* :: ′*a list*
begin

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

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

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

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

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

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

lemma [*simp*]: *xd i i* = *0 xd i j* + *xd j k* = *xd i k xd i j* + *xd k i* = *xd k j*
⟨*proof*⟩ **function** *xij-f* :: *nat* ⇒ *nat* ⇒ ′*a* **where**
xij-f i j = (*if i* < *j then* (*xij-f* (*i* + *1*) *j* − *xij-f i* (*j* − *1*)) / *xd j i else f i*)

⟨*proof*⟩

termination ⟨*proof*⟩ **definition** $c :: nat \Rightarrow 'a$ **where**
$c\ i = xij\text{-}f\ 0\ i$

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

private function $b :: nat \Rightarrow nat \Rightarrow 'a\ poly$ **where**
$b\ i\ n = (if\ i \geq n\ then\ [:c\ n:]\ else\ b\ (Suc\ i)\ n * X\ i + [:c\ i:])$
⟨*proof*⟩

termination ⟨*proof*⟩

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

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

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

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

lemma *Xii-1*[*simp*]: $Xij\ i\ i = 1$ ⟨*proof*⟩
lemma *smult-1*[*simp*]: $smult\ d\ 1 = [:d:]$
⟨*proof*⟩ **lemma** *newton-poly-sum*:
$newton\text{-}poly\ n = sum\text{-}list\ (map\ (\lambda\ i.\ smult\ (c\ i)\ (N\ i))\ [0\ ..<\ Suc\ n])$
⟨*proof*⟩ **lemma** *poly-newton-poly*: $poly\ (newton\text{-}poly\ n)\ y = sum\text{-}list\ (map\ (\lambda\ i.\ c\ i * poly\ (N\ i)\ y)\ [0\ ..<\ Suc\ n])$
⟨*proof*⟩ **definition** $pprod\ k\ i\ j = (\prod l \leftarrow [i..<j].\ xd\ k\ l)$

private lemma *poly-N-xi*: $poly\ (N\ i)\ (x\ j) = pprod\ j\ 0\ i$
⟨*proof*⟩ **lemma** *poly-N-xi-cond*: $poly\ (N\ i)\ (x\ j) = (if\ j < i\ then\ 0\ else\ pprod\ j\ 0\ i)$
⟨*proof*⟩ **lemma** *poly-newton-poly-xj*: **assumes** $j \leq n$
shows $poly\ (newton\text{-}poly\ n)\ (x\ j) = sum\text{-}list\ (map\ (\lambda\ i.\ c\ i * poly\ (N\ i)\ (x\ j))\ [0\ ..<\ Suc\ j])$
⟨*proof*⟩

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

context
fixes n
assumes *dist*: $\bigwedge\ i\ j.\ i < j \Longrightarrow j \leq n \Longrightarrow x\ i \neq x\ j$
begin
private lemma *xd-diff*: $i < j \Longrightarrow j \leq n \Longrightarrow xd\ i\ j \neq 0$
$i < j \Longrightarrow j \leq n \Longrightarrow xd\ j\ i \neq 0$ ⟨*proof*⟩

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

private lemma *divided-differences-main*: **assumes** $k \leq n\ i < k$
shows $sum\text{-}list\ (map\ (\lambda\ j.\ xij\text{-}f\ i\ (i + j) * pprod\ k\ i\ (i + j))\ [0..<Suc\ k - i])$

=
  sum-list (*map* (λ *j*. *xij-f* (*Suc i*) (*Suc i* + *j*) ∗ *pprod k* (*Suc i*) (*Suc i* + *j*)) [*0*..<*Suc k* − *Suc i*])
⟨*proof*⟩ **lemma** *divided-differences*: **assumes** *kn*: *k* ≤ *n* **and** *ik*: *i* ≤ *k*
  shows *sum-list* (*map* (λ *j*. *xij-f i* (*i* + *j*) ∗ *pprod k i* (*i* + *j*)) [*0*..<*Suc k* − *i*]) = *f k*
⟨*proof*⟩

lemma *newton-poly-sound*: **assumes** *k* ≤ *n*
  shows *poly* (*newton-poly n*) (*x k*) = *f k*
⟨*proof*⟩
end

lemma *newton-poly-degree*: *degree* (*newton-poly n*) ≤ *n*
⟨*proof*⟩

context
  fixes *n*
  assumes *xs*: *length xs* = *n*
    and *fs*: *length fs* = *n*
begin
lemma *newton-coefficients-main*:
  k < *n* ⟹ *newton-coefficients-main* (*rev* (*map f* [*0*..<*Suc k*])) (*rev* (*map x* [*0*..<*Suc k*]))
    = *rev* (*map* (λ *i*. *map* (λ *j*. *xij-f j i*) [*0*..<*Suc i*]) [*0*..<*Suc k*])
⟨*proof*⟩

lemma *newton-coefficients*: *newton-coefficients* = *rev* (*map c* [*0* ..< *n*])
⟨*proof*⟩

lemma *newton-poly-impl*: **assumes** *n* = *Suc nn*
  shows *newton-poly-impl* = *newton-poly nn*
⟨*proof*⟩
end
end

context
  fixes *xs fs* :: *int list*
begin

fun *divided-differences-impl-int* :: *int list* ⇒ *int* ⇒ *int* ⇒ *int list* ⇒ *int list option*
where
  divided-differences-impl-int (*xi-j1* # *x-j1s*) *fj xj* (*xi* # *xis*) = (
    case divided-differences-impl-int x-j1s fj xj xis of None ⇒ *None*
  | *Some x-js* ⇒ *let* (*new*,*m*) = *divmod-int* (*hd x-js* − *xi-j1*) (*xj* − *xi*)
    in if m = *0 then Some* (*new* # *x-js*) *else None*)
| *divided-differences-impl-int* [] *fj xj xis* = *Some* [*fj*]

fun *newton-coefficients-main-int* :: *int list* ⇒ *int list* ⇒ *int list list option* **where**

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

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

lemma *divided-differences-impl-int-Some*:
length gs ≤ *length ys*
⟹ *divided-differences-impl-int gs g x ys* = *Some res*
⟹ *divided-differences-impl* (*map rat-of-int gs*) (*rat-of-int g*) (*rat-of-int x*) (*map rat-of-int ys*) = *map rat-of-int res*
∧ *length res* = *Suc* (*length gs*)
⟨*proof*⟩

lemma *div-Ints-mod-0*: **assumes** *rat-of-int a* / *rat-of-int b* ∈ ℤ *b* ≠ *0*
shows *a mod b* = *0*
⟨*proof*⟩

lemma *divided-differences-impl-int-None*:
length gs ≤ *length ys*
⟹ *divided-differences-impl-int gs g x ys* = *None*
⟹ *x* ∉ *set* (*take* (*length gs*) *ys*)
⟹ *hd* (*divided-differences-impl* (*map rat-of-int gs*) (*rat-of-int g*) (*rat-of-int x*) (*map rat-of-int ys*)) ∉ ℤ
⟨*proof*⟩

lemma *newton-coefficients-main-int-Some*:
length gs = *length ys* ⟹ *length ys* ≤ *length xs*
⟹ *newton-coefficients-main-int gs ys* = *Some res*
⟹ *newton-coefficients-main* (*map rat-of-int xs*) (*map rat-of-int gs*) (*map rat-of-int ys*) = *map* (*map rat-of-int*) *res*
∧ (∀ *x* ∈ *set res*. *x* ≠ [] ∧ *length x* ≤ *length ys*) ∧ *length res* = *length gs*
⟨*proof*⟩

lemma *newton-coefficients-main-int-None*: **assumes** *dist*: *distinct xs*
shows *length gs* = *length ys* ⟹ *length ys* ≤ *length xs*
⟹ *newton-coefficients-main-int gs ys* = *None*
⟹ *ys* = *drop* (*length xs* − *length ys*) (*rev xs*)
⟹ ∃ *row* ∈ *set* (*newton-coefficients-main* (*map rat-of-int xs*) (*map rat-of-int gs*) (*map rat-of-int ys*)). *hd row* ∉ ℤ
⟨*proof*⟩

lemma *newton-coefficients-int*: **assumes** *dist*: *distinct xs*
and *len*: *length xs = length fs*
shows *newton-coefficients-int = (let cs = newton-coefficients (map rat-of-int xs) (map of-int fs)*
in if set cs ⊆ ℤ then Some (map int-of-rat cs) else None)
⟨*proof*⟩

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

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

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

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

lemma *newton-interpolation-poly*: **assumes** *dist*: *distinct (map fst xs-ys)*
and *p*: *p = newton-interpolation-poly xs-ys*
and *xy*: *(x,y) ∈ set xs-ys*
shows *poly p x = y*
⟨*proof*⟩

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

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

lemma *newton-interpolation-poly-int*: **assumes** *dist*: *distinct (map fst xs-ys)*
shows *newton-interpolation-poly-int xs-ys = (let*
rxs-ys = map (λ (x,y). (rat-of-int x, rat-of-int y)) xs-ys;
rp = newton-interpolation-poly rxs-ys

in if (∀ *x* ∈ *set* (*coeffs rp*). *is-int-rat x*) *then*
Some (*map-poly int-of-rat rp*) *else None*)
⟨*proof*⟩

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

10 Lagrange Interpolation

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

theory *Lagrange-Interpolation*
imports
Missing-Polynomial
begin

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

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

lemma [*code*]:
lagrange-basis-poly xs xj = (*let ys* = *filter* (λ *x*. *x* ≠ *xj*) *xs*
in prod-list (*map* (λ *xi*. *let ii* = *inverse* (*xj* − *xi*) *in* [: − *ii* ∗ *xi*, *ii* :]) *ys*))
⟨*proof*⟩

lemma *degree-lagrange-basis-poly*: *degree* (*lagrange-basis-poly xs xj*) ≤ *length* (*filter* (λ *x*. *x* ≠ *xj*) *xs*)
⟨*proof*⟩

lemma *degree-lagrange-interpolation-poly*:
shows *degree* (*lagrange-interpolation-poly xs-ys*) ≤ *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'* ∈ *set* (*map fst xs-ys*) **and** *x'* ≠ *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 ⋀ *x y*. (*x*,*y*) ∈ *set xs-ys* ⟹ *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* :: *nat* ⇒ ′*a* :: *field*
  and *f* :: *nat* ⇒ ′*a*
begin

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

function *neville-aitken-main* :: *nat* ⇒ *nat* ⇒ ′*a poly* **where**
  neville-aitken-main i j = (*if i* < *j then*
      (*smult* (*inverse* (*x j* − *x i*)) (*X i* ∗ *neville-aitken-main* (*i* + *1*) *j* −
      X j ∗ *neville-aitken-main i* (*j* − *1*)))
    else [:*f i*:])
  ⟨*proof*⟩

termination ⟨*proof*⟩

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

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

lemma *neville-aitken-main*: **assumes** *dist*: ⋀ *i j*. *i* < *j* ⟹ *j* ≤ *n* ⟹ *x i* ≠ *x j*
  shows *i* ≤ *k* ⟹ *k* ≤ *j* ⟹ *j* ≤ *n* ⟹ *poly* (*neville-aitken-main i j*) (*x k*) = (*f k*)
⟨*proof*⟩

lemma *degree-neville-aitken-main*: *degree* (*neville-aitken-main i j*) ≤ *j* − *i*
⟨*proof*⟩

lemma *degree-neville-aitken*: *degree* (*neville-aitken n*) ≤ *n*
⟨*proof*⟩

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

lemma *length-neville-aitken-merge*[*termination-simp*]: *length* (*neville-aitken-merge xs*) = *length xs* − *1*
⟨*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 = *map* (λ *i*. (*x i*, *x* (*i* + *j*), *neville-aitken-main i* (*i* + *j*))) [*l* ..< *Suc* (*l* + *k*)]
⟹ *neville-aitken-merge xs*
= (*map* (λ *i*. (*x i*, *x* (*i* + *Suc j*), *neville-aitken-main i* (*i* + *Suc j*))) [*l* ..< *l* + *k*])
⟨*proof*⟩

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

lemma *neville-aitken-impl*:
xs = *map* (λ *i*. (*x i*, *x i*, [:*f i*:])) [*0* ..< *Suc k*]
⟹ *neville-aitken-impl-main xs* = *neville-aitken k*
⟨*proof*⟩
end

lemma *neville-aitken*: **assumes** ⋀ *i j*. *i* < *j* ⟹ *j* ≤ *n* ⟹ *x i* ≠ *x j*
shows *j* ≤ *n* ⟹ *poly* (*neville-aitken x f n*) (*x j*) = (*f j*)
⟨*proof*⟩

definition *neville-aitken-interpolation-poly* :: ('*a* :: *field* × '*a*)*list* ⇒ '*a poly* **where**
neville-aitken-interpolation-poly x-fs = (*let*
start = *map* (λ (*xi*,*fi*). (*xi*,*xi*,[:*fi*:])) *x-fs in*
neville-aitken-impl-main start)

lemma *neville-aitken-interpolation-impl*: **assumes** *x-fs* ≠ []
shows *neville-aitken-interpolation-poly x-fs* =
neville-aitken (λ *i*. *fst* (*x-fs* ! *i*)) (λ *i*. *snd* (*x-fs* ! *i*)) (*length x-fs* − *1*)
⟨*proof*⟩

lemma *neville-aitken-interpolation-poly*: **assumes** *dist*: *distinct* (*map fst xs-ys*)
and *p*: *p* = *neville-aitken-interpolation-poly xs-ys*
and *xy*: (*x*,*y*) ∈ *set xs-ys*
shows *poly p x* = *y*
⟨*proof*⟩

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 ($\forall$ *x* $\in$ *set* (*coeffs rp*). *is-int-rat x*) *then*
Some (*map-poly int-of-rat rp*) *else None*)

lemma *interpolation-poly-int-def*: *distinct* (*map fst xs-ys*) $\Longrightarrow$
interpolation-poly-int alg xs-ys = (*let*
rxs-ys = *map* (λ (*x*,*y*). (*of-int x*, *of-int y*)) *xs-ys*;
rp = *interpolation-poly alg 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*⟩

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

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

lemma *uniqueness-of-interpolation*: **fixes** *p* :: *'a* :: *idom poly*
assumes *cS*: *card S* = *Suc n*
and *degree p* $\leq$ *n* **and** *degree q* $\leq$ *n* **and**
id: $\bigwedge$ *x*. *x* $\in$ *S* $\Longrightarrow$ *poly p x* = *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*) $\in$ *set xs-ys* $\Longrightarrow$ *poly p x* = *y degree p* < *length xs-ys*
and *q*: $\bigwedge$ *x y*. (*x*,*y*) $\in$ *set xs-ys* $\Longrightarrow$ *poly q x* = *y degree q* < *length xs-ys*
shows *p* = *q*
⟨*proof*⟩

lemma *exactly-one-poly-interpolation*: **assumes** *xs*: *xs-ys* $\neq$ [] **and** *dist*: *distinct* (*map fst xs-ys*)
shows $\exists$! *p*. *degree p* < *length xs-ys* $\wedge$ ($\forall$ *x y*. (*x*,*y*) $\in$ *set xs-ys* $\longrightarrow$ *poly p x* = (*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*) $\in$ *set xs-ys* $\Longrightarrow$ *poly p x* = *y degree p* $\leq$ *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*) ∈ *set xs-ys* ⟹ *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 point fails, and so adding further points (which corresponds to increasing the degree) will also fail. Of course, this can be generalized, showing that whenever you cannot interpolate a set of n points with an integer polynomial of degree $n - 1$, then you cannot interpolate this set of points with any integer polynomial. However, we did not formally prove this more general fact.

lemma *impossible-p-0-is-0-and-p-2-is-1*: ¬ (∃ *p*. *poly p 0* = *0* ∧ *poly p 2* = (*1* :: *int*))
⟨*proof*⟩

end

References

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