Executable multivariate polynomials

Christian Sternagel and René Thiemann and Fabian Immler and Alexander Maletzky*

April 20, 2020

Abstract

We define multivariate polynomials over arbitrary (ordered) semirings in combination with (executable) operations like addition, multiplication, and substitution. We also define (weak) monotonicity of polynomials and comparison of polynomials where we provide standard estimations like absolute positiveness or the more recent approach of [3]. Moreover, it is proven that strongly normalizing (monotone) orders can be lifted to strongly normalizing (monotone) orders over polynomials.

Our formalization was performed as part of the IsaFoR/CeTA-system [5][1] which contains several termination techniques. The provided theories have been essential to formalize polynomial-interpretations [1, 2].

This formalization also contains an abstract representation as coefficient functions with finite support and a type of power-products. If this type is ordered by a linear (term) ordering, various additional notions, such as leading power-product, leading coefficient etc., are introduced as well. Furthermore, a lot of generic properties of, and functions on, multivariate polynomials are formalized, including the substitution and evaluation homomorphisms, embeddings of polynomial rings into larger rings (i.e. with one additional indeterminate), homogenization and dehomogenization of polynomials, and the canonical isomorphism between $R[X, Y]$ and $R[X][Y]$.

Contents

*Supported by the Austrian Science Fund (FWF): grant no. W1214-N15, project DK1

[1] http://cl-informatik.uibk.ac.at/software/ceta

1 Utilities

theory *Utils*
imports *Main Well-Quasi-Orders.Almost-Full-Relations*
begin

lemma *subset-imageE-inj*:
assumes $B \subseteq f$ ‘ A
obtains C **where** $C \subseteq A$ **and** $B = f$ ‘ C **and** *inj-on* f C
⟨*proof*⟩

lemma *wfP-chain*:
assumes $\neg(\exists f.\ \forall i.\ r\ (f\ (Suc\ i))\ (f\ i))$
shows *wfP* r
⟨*proof*⟩

lemma *transp-sequence*:
assumes *transp* r **and** $\bigwedge i.\ r\ (seq\ (Suc\ i))\ (seq\ i)$ **and** $i < j$
shows $r\ (seq\ j)\ (seq\ i)$
⟨*proof*⟩

lemma *almost-full-on-finite-subsetE*:
assumes *reflp* P **and** *almost-full-on* P S
obtains T **where** *finite* T **and** $T \subseteq S$ **and** $\bigwedge s.\ s \in S \Longrightarrow (\exists\, t \in T.\ P\ t\ s)$
⟨*proof*⟩

1.1 Lists

lemma *map-upt*: *map* (λ*i*. *f* (*xs* ! *i*)) [*0*..<*length xs*] = *map f xs*
⟨*proof*⟩

lemma *map-upt-zip*:
assumes *length xs* = *length ys*
shows *map* (λ*i*. *f* (*xs* ! *i*) (*ys* ! *i*)) [*0*..<*length ys*] = *map* (λ(*x*, *y*). *f x y*) (*zip xs ys*) (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *distinct-sorted-wrt-irrefl*:
assumes *irreflp rel* **and** *transp rel* **and** *sorted-wrt rel xs*
shows *distinct xs*
⟨*proof*⟩

lemma *distinct-sorted-wrt-imp-sorted-wrt-strict*:
assumes *distinct xs* **and** *sorted-wrt rel xs*
shows *sorted-wrt* (λ*x y*. *rel x y* ∧ ¬ *x* = *y*) *xs*
⟨*proof*⟩

lemma *sorted-wrt-distinct-set-unique*:
assumes *antisymp rel*
assumes *sorted-wrt rel xs distinct xs sorted-wrt rel ys distinct ys set xs* = *set ys*
shows *xs* = *ys*
⟨*proof*⟩

lemma *sorted-wrt-refl-nth-mono*:
assumes *reflp P* **and** *sorted-wrt P xs* **and** *i* ≤ *j* **and** *j* < *length xs*
shows *P* (*xs* ! *i*) (*xs* ! *j*)
⟨*proof*⟩

fun *merge-wrt* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a list* ⇒ '*a list* **where**
merge-wrt - xs [] = *xs*|
merge-wrt rel [] *ys* = *ys*|
merge-wrt rel (*x* # *xs*) (*y* # *ys*) =
(*if x* = *y then*
y # (*merge-wrt rel xs ys*)
else if rel x y then
x # (*merge-wrt rel xs* (*y* # *ys*))
else
y # (*merge-wrt rel* (*x* # *xs*) *ys*)
)

lemma *set-merge-wrt*: *set* (*merge-wrt rel xs ys*) = *set xs* ∪ *set ys*
⟨*proof*⟩

lemma *sorted-merge-wrt*:
assumes *transp rel* **and** ⋀*x y*. *x* ≠ *y* ⟹ *rel x y* ∨ *rel y x*
and *sorted-wrt rel xs* **and** *sorted-wrt rel ys*

shows *sorted-wrt rel* (*merge-wrt rel xs ys*)
⟨*proof*⟩

lemma *set-fold*:
assumes $\bigwedge x\ ys.\ set\ (f\ (g\ x)\ ys) = set\ (g\ x) \cup set\ ys$
shows $set\ (fold\ (\lambda x.\ f\ (g\ x))\ xs\ ys) = (\bigcup x{\in}set\ xs.\ set\ (g\ x)) \cup set\ ys$
⟨*proof*⟩

1.2 Sums and Products

lemma *additive-implies-homogenous*:
assumes $\bigwedge x\ y.\ f\ (x + y) = f\ x + ((f\ (y::'a::monoid\text{-}add))::'b::cancel\text{-}comm\text{-}monoid\text{-}add)$
shows $f\ 0 = 0$
⟨*proof*⟩

lemma *fun-sum-commute*:
assumes $f\ 0 = 0$ **and** $\bigwedge x\ y.\ f\ (x + y) = f\ x + f\ y$
shows $f\ (sum\ g\ A) = (\sum a{\in}A.\ f\ (g\ a))$
⟨*proof*⟩

lemma *fun-sum-commute-canc*:
assumes $\bigwedge x\ y.\ f\ (x + y) = f\ x + ((f\ y)::'a::cancel\text{-}comm\text{-}monoid\text{-}add)$
shows $f\ (sum\ g\ A) = (\sum a{\in}A.\ f\ (g\ a))$
⟨*proof*⟩

lemma *fun-sum-list-commute*:
assumes $f\ 0 = 0$ **and** $\bigwedge x\ y.\ f\ (x + y) = f\ x + f\ y$
shows $f\ (sum\text{-}list\ xs) = sum\text{-}list\ (map\ f\ xs)$
⟨*proof*⟩

lemma *fun-sum-list-commute-canc*:
assumes $\bigwedge x\ y.\ f\ (x + y) = f\ x + ((f\ y)::'a::cancel\text{-}comm\text{-}monoid\text{-}add)$
shows $f\ (sum\text{-}list\ xs) = sum\text{-}list\ (map\ f\ xs)$
⟨*proof*⟩

lemma *sum-set-upt-eq-sum-list*: $(\sum i = m..{<}n.\ f\ i) = (\sum i{\leftarrow}[m..{<}n].\ f\ i)$
⟨*proof*⟩

lemma *sum-list-upt*: $(\sum i{\leftarrow}[0..{<}(length\ xs)].\ f\ (xs\ !\ i)) = (\sum x{\leftarrow}xs.\ f\ x)$
⟨*proof*⟩

lemma *sum-list-upt-zip*:
assumes $length\ xs = length\ ys$
shows $(\sum i{\leftarrow}[0..{<}(length\ ys)].\ f\ (xs\ !\ i)\ (ys\ !\ i)) = (\sum (x,\ y){\leftarrow}(zip\ xs\ ys).\ f\ x\ y)$
⟨*proof*⟩

lemma *sum-list-zeroI*:
assumes $set\ xs \subseteq \{0\}$

shows *sum-list xs = 0*
⟨*proof*⟩

lemma *fun-prod-commute*:
assumes *f 1 = 1* **and** $\bigwedge x\ y.\ f\ (x * y) = f\ x * f\ y$
shows *f (prod g A) =* $(\prod a{\in}A.\ f\ (g\ a))$
⟨*proof*⟩

end

2 An abstract type for multivariate polynomials

theory *MPoly-Type*
imports *HOL−Library.Poly-Mapping*
begin

2.1 Abstract type definition

typedef (**overloaded**) *'a mpoly =*
UNIV :: $((nat \Rightarrow_0 nat) \Rightarrow_0 'a{::}zero)\ set$
morphisms *mapping-of MPoly*
⟨*proof*⟩

setup-lifting *type-definition-mpoly*

thm *mapping-of-inverse* **thm** *MPoly-inverse*
thm *mapping-of-inject* **thm** *MPoly-inject*
thm *mapping-of-induct* **thm** *MPoly-induct*
thm *mapping-of-cases* **thm** *MPoly-cases*

2.2 Additive structure

instantiation *mpoly* :: (*zero*) *zero*
begin

lift-definition *zero-mpoly* :: *'a mpoly*
is *0* :: $(nat \Rightarrow_0 nat) \Rightarrow_0 'a$ ⟨*proof*⟩

instance ⟨*proof*⟩

end

instantiation *mpoly* :: (*monoid-add*) *monoid-add*
begin

lift-definition *plus-mpoly* :: *'a mpoly ⇒ 'a mpoly ⇒ 'a mpoly*
is *Groups.plus* :: $((nat \Rightarrow_0 nat) \Rightarrow_0 'a) \Rightarrow$ - ⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *mpoly* :: (*comm-monoid-add*) *comm-monoid-add*
⟨*proof*⟩

instantiation *mpoly* :: (*cancel-comm-monoid-add*) *cancel-comm-monoid-add*
begin

lift-definition *minus-mpoly* :: *'a mpoly* ⇒ *'a mpoly* ⇒ *'a mpoly*
is *Groups.minus* :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ - ⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *mpoly* :: (*ab-group-add*) *ab-group-add*
begin

lift-definition *uminus-mpoly* :: *'a mpoly* ⇒ *'a mpoly*
is *Groups.uminus* :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ - ⟨*proof*⟩

instance
⟨*proof*⟩

end

2.3 Multiplication by a coefficient

lift-definition *smult* :: *'a*::{*times*,*zero*} ⇒ *'a mpoly* ⇒ *'a mpoly*
is λ*a*. *Poly-Mapping.map* (*Groups.times a*) :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ - ⟨*proof*⟩

2.4 Multiplicative structure

instantiation *mpoly* :: (*zero-neq-one*) *zero-neq-one*
begin

lift-definition *one-mpoly* :: *'a mpoly*
is *1* :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *mpoly* :: (*semiring-0*) *semiring-0*

begin

lift-definition *times-mpoly* :: *'a mpoly* ⇒ *'a mpoly* ⇒ *'a mpoly*
is *Groups.times* :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ - ⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *mpoly* :: (*comm-semiring-0*) *comm-semiring-0*
⟨*proof*⟩

instance *mpoly* :: (*semiring-0-cancel*) *semiring-0-cancel*
⟨*proof*⟩

instance *mpoly* :: (*comm-semiring-0-cancel*) *comm-semiring-0-cancel*
⟨*proof*⟩

instance *mpoly* :: (*semiring-1*) *semiring-1*
⟨*proof*⟩

instance *mpoly* :: (*comm-semiring-1*) *comm-semiring-1*
⟨*proof*⟩

instance *mpoly* :: (*semiring-1-cancel*) *semiring-1-cancel*
⟨*proof*⟩

instance *mpoly* :: (*ring*) *ring*
⟨*proof*⟩

instance *mpoly* :: (*comm-ring*) *comm-ring*
⟨*proof*⟩

instance *mpoly* :: (*ring-1*) *ring-1*
⟨*proof*⟩

instance *mpoly* :: (*comm-ring-1*) *comm-ring-1*
⟨*proof*⟩

2.5 Monomials

Terminology is not unique here, so we use the notions as follows: A "monomial" and a "coefficient" together give a "term". These notions are significant in connection with "leading", "leading term", "leading coefficient" and "leading monomial", which all rely on a monomial order.

lift-definition *monom* :: (*nat* ⇒$_0$ *nat*) ⇒ *'a*::*zero* ⇒ *'a mpoly*

is *Poly-Mapping.single* :: $(nat \Rightarrow_0 nat) \Rightarrow$ - ⟨*proof*⟩

lemma *mapping-of-monom* [*simp*]:
mapping-of (*monom m a*) = *Poly-Mapping.single m a*
⟨*proof*⟩

lemma *monom-zero* [*simp*]:
monom 0 0 = *0*
⟨*proof*⟩

lemma *monom-one* [*simp*]:
monom 0 1 = *1*
⟨*proof*⟩

lemma *monom-add*:
monom m (*a* + *b*) = *monom m a* + *monom m b*
⟨*proof*⟩

lemma *monom-uminus*:
monom m (− *a*) = − *monom m a*
⟨*proof*⟩

lemma *monom-diff*:
monom m (*a* − *b*) = *monom m a* − *monom m b*
⟨*proof*⟩

lemma *monom-numeral* [*simp*]:
monom 0 (*numeral n*) = *numeral n*
⟨*proof*⟩

lemma *monom-of-nat* [*simp*]:
monom 0 (*of-nat n*) = *of-nat n*
⟨*proof*⟩

lemma *of-nat-monom*:
of-nat = *monom 0* ∘ *of-nat*
⟨*proof*⟩

lemma *inj-monom* [*iff*]:
inj (*monom m*)
⟨*proof*⟩

lemma *mult-monom*: *monom x a* ∗ *monom y b* = *monom* (*x* + *y*) (*a* ∗ *b*)
⟨*proof*⟩

instance *mpoly* :: (*semiring-char-0*) *semiring-char-0*
⟨*proof*⟩

instance *mpoly* :: (*ring-char-0*) *ring-char-0*

⟨*proof*⟩

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

2.6 Constants and Indeterminates

Embedding of indeterminates and constants in type-class polynomials, can be used as constructors.

definition Var_0 :: *'a* ⇒ (*'a* ⇒$_0$ *nat*) ⇒$_0$ *'b*::{*one*,*zero*} **where**
Var_0 *n* ≡ *Poly-Mapping.single* (*Poly-Mapping.single n 1*) *1*
definition $Const_0$ :: *'b* ⇒ (*'a* ⇒$_0$ *nat*) ⇒$_0$ *'b*::*zero* **where** $Const_0$ *c* ≡ *Poly-Mapping.single 0 c*

lemma $Const_0$-*one*: $Const_0$ *1* = *1*
⟨*proof*⟩

lemma $Const_0$-*numeral*: $Const_0$ (*numeral x*) = *numeral x*
⟨*proof*⟩

lemma $Const_0$-*minus*: $Const_0$ (− *x*) = − $Const_0$ *x*
⟨*proof*⟩

lemma $Const_0$-*zero*: $Const_0$ *0* = *0*
⟨*proof*⟩

lemma Var_0-*power*: Var_0 *v* ^ *n* = *Poly-Mapping.single* (*Poly-Mapping.single v n*) *1*
⟨*proof*⟩

lift-definition *Var*::*nat* ⇒ *'b*::{*one*,*zero*} *mpoly* **is** Var_0 ⟨*proof*⟩
lift-definition *Const*::*'b*::*zero* ⇒ *'b mpoly* **is** $Const_0$ ⟨*proof*⟩

2.7 Integral domains

instance *mpoly* :: (*ring-no-zero-divisors*) *ring-no-zero-divisors*
⟨*proof*⟩

instance *mpoly* :: (*ring-1-no-zero-divisors*) *ring-1-no-zero-divisors*
⟨*proof*⟩

instance *mpoly* :: (*idom*) *idom*
⟨*proof*⟩

2.8 Monom coefficient lookup

definition *coeff* :: *'a*::*zero mpoly* ⇒ (*nat* ⇒$_0$ *nat*) ⇒ *'a*
where

coeff p = Poly-Mapping.lookup (mapping-of p)

2.9 Insertion morphism

definition *insertion-fun-natural* :: *(nat ⇒ 'a) ⇒ ((nat ⇒ nat) ⇒ 'a) ⇒ 'a::comm-semiring-1*
where
*insertion-fun-natural f p = (∑ m. p m * (∏ v. f v ^ m v))*

definition *insertion-fun* :: *(nat ⇒ 'a) ⇒ ((nat ⇒₀ nat) ⇒ 'a) ⇒ 'a::comm-semiring-1*
where
*insertion-fun f p = (∑ m. p m * (∏ v. f v ^ Poly-Mapping.lookup m v))*

N.b. have been unable to relate this to *insertion-fun-natural* using lifting!

lift-definition *insertion-aux* :: *(nat ⇒ 'a) ⇒ ((nat ⇒₀ nat) ⇒₀ 'a) ⇒ 'a::comm-semiring-1*
is *insertion-fun ⟨proof⟩*

lift-definition *insertion* :: *(nat ⇒ 'a) ⇒ 'a mpoly ⇒ 'a::comm-semiring-1*
is *insertion-aux ⟨proof⟩*

lemma *aux*:
Poly-Mapping.lookup f = (λ-. 0) ⟷ f = 0
⟨proof⟩

lemma *insertion-trivial* [*simp*]:
insertion (λ-. 0) p = coeff p 0
⟨proof⟩

lemma *insertion-zero* [*simp*]:
insertion f 0 = 0
⟨proof⟩

lemma *insertion-fun-add*:
fixes *f p q*
shows *insertion-fun f (Poly-Mapping.lookup (p + q)) =*
insertion-fun f (Poly-Mapping.lookup p) +
insertion-fun f (Poly-Mapping.lookup q)
⟨proof⟩

lemma *insertion-add*:
insertion f (p + q) = insertion f p + insertion f q
⟨proof⟩

lemma *insertion-one* [*simp*]:
insertion f 1 = 1
⟨proof⟩

lemma *insertion-fun-mult*:
fixes *f p q*
shows *insertion-fun f (Poly-Mapping.lookup (p * q)) =*
*insertion-fun f (Poly-Mapping.lookup p) **

insertion-fun f (Poly-Mapping.lookup q)
⟨*proof*⟩

lemma *insertion-mult*:
*insertion f (p * q) = insertion f p * insertion f q*
⟨*proof*⟩

2.10 Degree

lift-definition *degree* :: *'a::zero mpoly ⇒ nat ⇒ nat*
is *λp v. Max (insert 0 ((λm. Poly-Mapping.lookup m v) ‘ Poly-Mapping.keys p))*
⟨*proof*⟩

lift-definition *total-degree* :: *'a::zero mpoly ⇒ nat*
is *λp. Max (insert 0 ((λm. sum (Poly-Mapping.lookup m) (Poly-Mapping.keys m))*
‘ Poly-Mapping.keys p)) ⟨*proof*⟩

lemma *degree-zero* [*simp*]:
degree 0 v = 0
⟨*proof*⟩

lemma *total-degree-zero* [*simp*]:
total-degree 0 = 0
⟨*proof*⟩

lemma *degree-one* [*simp*]:
degree 1 v = 0
⟨*proof*⟩

lemma *total-degree-one* [*simp*]:
total-degree 1 = 0
⟨*proof*⟩

2.11 Pseudo-division of polynomials

lemma *smult-conv-mult*: *smult s p = monom 0 s * p*
⟨*proof*⟩

lemma *smult-monom* [*simp*]:
fixes *c* :: - :: *mult-zero*
shows *smult c (monom x c′) = monom x (c * c′)*
⟨*proof*⟩

lemma *smult-0* [*simp*]:
fixes *p* :: - :: *mult-zero mpoly*
shows *smult 0 p = 0*
⟨*proof*⟩

lemma *mult-smult-left*: *smult s p * q = smult s (p * q)*
⟨*proof*⟩

lift-definition *sdiv* :: *'a::euclidean-ring ⇒ 'a mpoly ⇒ 'a mpoly*
is *λa. Poly-Mapping.map (λb. b div a) :: ((nat ⇒₀ nat) ⇒₀ 'a) ⇒ -*
⟨*proof*⟩

'Polynomial division' is only possible on univariate polynomials $K[x]$ over a field K, all other kinds of polynomials only allow pseudo-division [1]p.40/41":

*∀ x y :: 'a mpoly. y ≠ 0 ⇒ ∃ a q r. smult a x = q * y + r*

The introduction of pseudo-division below generalises `~~/src/HOL/Computational_Algebra/Polynomial.thy`. [1] Winkler, Polynomial Algorithms, 1996. The generalisation raises issues addressed by Wenda Li and commented below. Florian replied to the issues conjecturing, that the abstract mpoly needs not be aware of the issues, in case these are only concerned with executability.

definition *pseudo-divmod-rel*
:: *'a::euclidean-ring => 'a mpoly => 'a mpoly => 'a mpoly => 'a mpoly => bool*
where
pseudo-divmod-rel a x y q r ⟷
*smult a x = q * y + r ∧ (if y = 0 then q = 0 else r = 0 ∨ degree r < degree y)*

definition *pdiv* :: *'a::euclidean-ring mpoly ⇒ 'a mpoly ⇒ ('a × 'a mpoly)* (**infixl** *pdiv 70*)
where
x pdiv y = (THE (a, q). ∃ r. pseudo-divmod-rel a x y q r)

definition *pmod* :: *'a::euclidean-ring mpoly ⇒ 'a mpoly ⇒ 'a mpoly* (**infixl** *pmod 70*)
where
x pmod y = (THE r. ∃ a q. pseudo-divmod-rel a x y q r)

definition *pdivmod* :: *'a::euclidean-ring mpoly ⇒ 'a mpoly ⇒ ('a × 'a mpoly) × 'a mpoly*
where
pdivmod p q = (p pdiv q, p pmod q)

lemma *pdiv-code*:
p pdiv q = fst (pdivmod p q)
⟨*proof*⟩

lemma *pmod-code*:
p pmod q = snd (pdivmod p q)
⟨*proof*⟩

definition *div* :: *'a*::{*euclidean-ring*,*field*} *mpoly* ⇒ *'a mpoly* ⇒ *'a mpoly* (**infixl** *div 70*)
where
x div y = (*THE q'*. ∃ *a q r*. (*pseudo-divmod-rel a x y q r*) ∧ (*q'* = *smult* (*inverse a*) *q*))

definition *mod* :: *'a*::{*euclidean-ring*,*field*} *mpoly* ⇒ *'a mpoly* ⇒ *'a mpoly* (**infixl** *mod 70*)
where
x mod y = (*THE r'*. ∃ *a q r*. (*pseudo-divmod-rel a x y q r*) ∧ (*r'* = *smult* (*inverse a*) *r*))

definition *divmod* :: *'a*::{*euclidean-ring*,*field*} *mpoly* ⇒ *'a mpoly* ⇒ *'a mpoly* × *'a mpoly*
where
divmod p q = (*p div q*, *p mod q*)

lemma *div-poly-code*:
p div q = *fst* (*divmod p q*)
⟨*proof*⟩

lemma *mod-poly-code*:
p mod q = *snd* (*divmod p q*)
⟨*proof*⟩

2.12 Primitive poly, etc

lift-definition *coeffs* :: *'a* :: *zero mpoly* ⇒ *'a set*
is *Poly-Mapping.range* :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ - ⟨*proof*⟩

lemma *finite-coeffs* [*simp*]: *finite* (*coeffs p*)
⟨*proof*⟩

[1]p.82 A "primitive"' polynomial has coefficients with GCD equal to 1. A polynomial is factored into "content" and "primitive part" for many different purposes.

definition *primitive* :: *'a*::{*euclidean-ring*,*semiring-Gcd*} *mpoly* ⇒ *bool*
where
primitive p ⟷ *Gcd* (*coeffs p*) = *1*

definition *content-primitive* :: *'a*::{*euclidean-ring*,*GCD.Gcd*} *mpoly* ⇒ *'a* × *'a mpoly*
where
content-primitive p = (
let d = *Gcd* (*coeffs p*)
in (*d*, *sdiv d p*))

value *let p = M [1,2,3] (4::int) + M [2,0,4] 6 + M [2,0,5] 8*
in content-primitive p

end

theory *More-MPoly-Type*
imports *MPoly-Type*
begin

abbreviation *lookup == Poly-Mapping.lookup*
abbreviation *keys == Poly-Mapping.keys*

3 MPpoly Mapping extenion

lemma *lookup-Abs-poly-mapping-when-finite*:
assumes *finite S*
shows *lookup (Abs-poly-mapping (λx. f x when x∈S)) = (λx. f x when x∈S)*
⟨*proof*⟩

definition *remove-key*::*'a ⇒ ('a ⇒₀ 'b*::*monoid-add) ⇒ ('a ⇒₀ 'b) **where***
remove-key k0 f = Abs-poly-mapping (λk. lookup f k when k ≠ k0)

lemma *remove-key-lookup*:
lookup (remove-key k0 f) k = (lookup f k when k ≠ k0)
⟨*proof*⟩

lemma *remove-key-keys*: *keys f − {k} = keys (remove-key k f)* (**is** *?A = ?B*)
⟨*proof*⟩

lemma *remove-key-sum*: *remove-key k f + Poly-Mapping.single k (lookup f k) = f*
⟨*proof*⟩

lemma *remove-key-single*[*simp*]: *remove-key v (Poly-Mapping.single v n) = 0*
⟨*proof*⟩

lemma *remove-key-add*: *remove-key v m + remove-key v m′ = remove-key v (m + m′)*
⟨*proof*⟩

lemma *poly-mapping-induct* [*case-names single sum*]:
fixes *P*::(*'a*, *'b*::*monoid-add*) *poly-mapping ⇒ bool*
assumes *single*:⋀*k v. P (Poly-Mapping.single k v)*
and *sum*:(⋀*f g k v. P f ⟹ P g ⟹ g = (Poly-Mapping.single k v) ⟹ k ∉ keys f ⟹ P (f+g)*)
shows *P f* ⟨*proof*⟩

lemma *map-lookup*:
assumes *g 0 = 0*
shows *lookup (Poly-Mapping.map g f) x = g ((lookup f) x)*
⟨*proof*⟩

lemma *keys-add*:
assumes *keys f ∩ keys g = {}*
shows *keys f ∪ keys g = keys (f+g)*
⟨*proof*⟩

lemma *fun-when*:
f 0 = 0 ⟹ f (a when P) = (f a when P) ⟨*proof*⟩

4 MPoly extension

lemma *coeff-all-0*:*(⋀m. coeff p m = 0) ⟹ p=0*
⟨*proof*⟩

definition *vars*::*'a*::*zero mpoly ⇒ nat set* **where**
vars p = ⋃ (keys ‘ keys (mapping-of p))

lemma *vars-finite*: *finite (vars p)* ⟨*proof*⟩

lemma *vars-monom-single*: *vars (monom (Poly-Mapping.single v k) a) ⊆ {v}*
⟨*proof*⟩

lemma *vars-monom-keys*:
assumes *a≠0*
shows *vars (monom m a) = keys m*
⟨*proof*⟩

lemma *vars-monom-subset*:
shows *vars (monom m a) ⊆ keys m*
⟨*proof*⟩

lemma *vars-monom-single-cases*: *vars (monom (Poly-Mapping.single v k) a) =*
(if k=0 ∨ a=0 then {} else {v})
⟨*proof*⟩

lemma *vars-monom*:
assumes *a≠0*
shows *vars (monom m (1*::*'a*::*zero-neq-one)) = vars (monom m (a*::*'a))*
⟨*proof*⟩

lemma *vars-add*: *vars (p1 + p2) ⊆ vars p1 ∪ vars p2*
⟨*proof*⟩

lemma *vars-mult*: *vars* $(p*q) \subseteq$ *vars* $p \cup$ *vars* q
⟨*proof*⟩

lemma *vars-add-monom*:
assumes $p2 =$ *monom* $m\ a\ m \notin$ *keys* (*mapping-of* $p1$)
shows *vars* $(p1 + p2) =$ *vars* $p1 \cup$ *vars* $p2$
⟨*proof*⟩

lemma *vars-setsum*: *finite* $S \Longrightarrow$ *vars* $(\sum m{\in}S.\ f\ m) \subseteq (\bigcup m{\in}S.\ vars\ (f\ m))$
⟨*proof*⟩

lemma *coeff-monom*: *coeff* (*monom* $m\ a$) $m' = (a$ *when* $m'{=}m)$
⟨*proof*⟩

lemma *coeff-add*: *coeff* $p\ m +$ *coeff* $q\ m =$ *coeff* $(p{+}q)\ m$
⟨*proof*⟩

lemma *coeff-eq*: *coeff* $p =$ *coeff* $q \longleftrightarrow p{=}q$ ⟨*proof*⟩

lemma *coeff-monom-mult*: *coeff* $((monom\ m'\ a)\ *\ q)\ (m' + m) = a *$ *coeff* $q\ m$
⟨*proof*⟩

lemma *one-term-is-monomial*:
assumes *card* (*keys* (*mapping-of* p)) ≤ 1
obtains m **where** $p =$ *monom* m (*coeff* $p\ m$)
⟨*proof*⟩

definition *remove-term*::$(nat \Rightarrow_0 nat) \Rightarrow 'a$::*zero mpoly* $\Rightarrow$ $'a$ *mpoly* **where**
remove-term m0 p = *MPoly* (*Abs-poly-mapping* ($\lambda m.$ *coeff p m when m* $\neq$ *m0*))

lemma *remove-term-coeff*: *coeff* (*remove-term m0 p*) *m* = (*coeff p m when m* $\neq$ *m0*)
⟨*proof*⟩

lemma *coeff-keys*: $m \in$ *keys* (*mapping-of* p) $\longleftrightarrow$ *coeff* $p\ m \neq 0$
⟨*proof*⟩

lemma *remove-term-keys*:
shows *keys* (*mapping-of p*) − {*m*} = *keys* (*mapping-of* (*remove-term m p*)) (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *remove-term-sum*: *remove-term* $m\ p +$ *monom* m (*coeff* $p\ m$) $= p$
⟨*proof*⟩

lemma *mpoly-induct* [*case-names monom sum*]:
assumes *monom*:$\bigwedge m\ a.\ P$ (*monom* $m\ a$)

and *sum*:($\bigwedge$*p1 p2 m a*. *P p1* $\Longrightarrow$ *P p2* $\Longrightarrow$ *p2* = (*monom m a*) $\Longrightarrow$ *m* $\notin$ *keys* (*mapping-of p1*) $\Longrightarrow$ *P* (*p1*+*p2*))
shows *P p* ⟨*proof*⟩

lemma *monom-pow*:*monom* (*Poly-Mapping.single v n0*) *a* ^ *n* = *monom* (*Poly-Mapping.single v* (*n0*∗*n*)) (*a* ^ *n*)
⟨*proof*⟩

lemma *insertion-fun-single*: *insertion-fun f* (λ*m*. (*a when* (*Poly-Mapping.single* (*v*::*nat*) (*n*::*nat*)) = *m*)) = *a* ∗ *f v* ^ *n* (**is** *?i* = -)
⟨*proof*⟩

lemma *insertion-single*[*simp*]: *insertion f* (*monom* (*Poly-Mapping.single* (*v*::*nat*) (*n*::*nat*)) *a*) = *a* ∗ *f v* ^ *n*
⟨*proof*⟩

lemma *insertion-fun-irrelevant-vars*:
fixes *p*::((*nat* ⇒$_0$ *nat*) ⇒ *'a*::*comm-ring-1*)
assumes $\bigwedge$*m v*. *p m* ≠ *0* $\Longrightarrow$ *lookup m v* ≠ *0* $\Longrightarrow$ *f v* = *g v*
shows *insertion-fun f p* = *insertion-fun g p*
⟨*proof*⟩

lemma *insertion-aux-irrelevant-vars*:
fixes *p*::((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*::*comm-ring-1*)
assumes $\bigwedge$*m v*. *lookup p m* ≠ *0* $\Longrightarrow$ *lookup m v* ≠ *0* $\Longrightarrow$ *f v* = *g v*
shows *insertion-aux f p* = *insertion-aux g p*
⟨*proof*⟩

lemma *insertion-irrelevant-vars*:
fixes *p*::*'a*::*comm-ring-1 mpoly*
assumes $\bigwedge$*v*. *v*∈*vars p* $\Longrightarrow$ *f v* = *g v*
shows *insertion f p* = *insertion g p*
⟨*proof*⟩

5 Nested MPoly

definition *reduce-nested-mpoly*::*'a*::*comm-ring-1 mpoly mpoly* ⇒ *'a mpoly* **where**
reduce-nested-mpoly pp = *insertion* (λ*v*. *monom* (*Poly-Mapping.single v 1*) *1*) *pp*

lemma *reduce-nested-mpoly-sum*:
fixes *p1*::*'a*::*comm-ring-1 mpoly mpoly*
shows *reduce-nested-mpoly* (*p1* + *p2*) = *reduce-nested-mpoly p1* + *reduce-nested-mpoly p2*
⟨*proof*⟩

lemma *reduce-nested-mpoly-prod*:
fixes *p1*::*'a*::*comm-ring-1 mpoly mpoly*
shows *reduce-nested-mpoly* (*p1* ∗ *p2*) = *reduce-nested-mpoly p1* ∗ *reduce-nested-mpoly*

p2
⟨*proof*⟩

lemma *reduce-nested-mpoly-0*:
shows *reduce-nested-mpoly 0 = 0* ⟨*proof*⟩

lemma *insertion-nested-poly*:
fixes *pp*::*'a*::*comm-ring-1 mpoly mpoly*
shows *insertion f* (*insertion* (λ*v*. *monom 0* (*f v*)) *pp*) = *insertion f* (*reduce-nested-mpoly pp*)
⟨*proof*⟩

definition *extract-var*::*'a*::*comm-ring-1 mpoly* ⇒ *nat* ⇒ *'a*::*comm-ring-1 mpoly mpoly* **where**
extract-var p v = (∑ *m*. *monom* (*remove-key v m*) (*monom* (*Poly-Mapping.single v* (*lookup m v*)) (*coeff p m*)))

lemma *extract-var-finite-set*:
assumes {*m'*. *coeff p m'* ≠ *0*} ⊆ *S*
assumes *finite S*
shows *extract-var p v* = (∑ *m*∈*S*. *monom* (*remove-key v m*) (*monom* (*Poly-Mapping.single v* (*lookup m v*)) (*coeff p m*)))
⟨*proof*⟩

lemma *extract-var-non-zero-coeff*: *extract-var p v* = (∑ *m*∈{*m'*. *coeff p m'* ≠ *0*}. *monom* (*remove-key v m*) (*monom* (*Poly-Mapping.single v* (*lookup m v*)) (*coeff p m*)))
⟨*proof*⟩

lemma *extract-var-sum*: *extract-var* (*p*+*p'*) *v* = *extract-var p v* + *extract-var p' v*
⟨*proof*⟩

lemma *extract-var-monom*:
shows *extract-var* (*monom m a*) *v* = *monom* (*remove-key v m*) (*monom* (*Poly-Mapping.single v* (*lookup m v*)) *a*)
⟨*proof*⟩

lemma *extract-var-monom-mult*:
shows *extract-var* (*monom* (*m*+*m'*) (*a*∗*b*)) *v* = *extract-var* (*monom m a*) *v* ∗ *extract-var* (*monom m' b*) *v*
⟨*proof*⟩

lemma *extract-var-single*: *extract-var* (*monom* (*Poly-Mapping.single v n*) *a*) *v* = *monom 0* (*monom* (*Poly-Mapping.single v n*) *a*)
⟨*proof*⟩

lemma *extract-var-single'*:
assumes $v \neq v'$
shows *extract-var* (*monom* (*Poly-Mapping.single v n*) *a*) v' = *monom* (*Poly-Mapping.single v n*) (*monom 0 a*)
⟨*proof*⟩

lemma *reduce-nested-mpoly-extract-var*:
fixes *p*::'*a*::*comm-ring-1 mpoly*
shows *reduce-nested-mpoly* (*extract-var p v*) = *p*
⟨*proof*⟩

lemma *vars-extract-var-subset*: *vars* (*extract-var p v*) ⊆ *vars p*
⟨*proof*⟩

lemma *v-not-in-vars-extract-var*: *v* ∉ *vars* (*extract-var p v*)
⟨*proof*⟩

lemma *vars-coeff-extract-var*: *vars* (*coeff* (*extract-var p v*) *j*) ⊆ {*v*}
⟨*proof*⟩

definition *replace-coeff*
where *replace-coeff f p* = *MPoly* (*Abs-poly-mapping* (λ*m*. *f* (*lookup* (*mapping-of p*) *m*)))

lemma *coeff-replace-coeff*:
assumes *f 0* = *0*
shows *coeff* (*replace-coeff f p*) *m* = *f* (*coeff p m*)
⟨*proof*⟩

lemma *replace-coeff-monom*:
assumes *f 0* = *0*
shows *replace-coeff f* (*monom m a*) = *monom m* (*f a*)
⟨*proof*⟩

lemma *replace-coeff-add*:
assumes *f 0* = *0*
assumes ⋀*a b*. *f* (*a*+*b*) = *f a* + *f b*
shows *replace-coeff f* (*p1* + *p2*) = *replace-coeff f p1* + *replace-coeff f p2*
⟨*proof*⟩

lemma *insertion-replace-coeff*:
fixes *pp*::'*a*::*comm-ring-1 mpoly mpoly*
shows *insertion f* (*replace-coeff* (*insertion f*) *pp*) = *insertion f* (*reduce-nested-mpoly pp*)
⟨*proof*⟩

lemma *replace-coeff-extract-var-cong*:
assumes *f v* = *g v*

shows *replace-coeff* (*insertion f*) (*extract-var p v*) = *replace-coeff* (*insertion g*) (*extract-var p v*)
⟨*proof*⟩

lemma *vars-replace-coeff*:
assumes *f 0* = *0*
shows *vars* (*replace-coeff f p*) ⊆ *vars p*
⟨*proof*⟩

definition *polyfun* :: *nat set* ⇒ ((*nat* ⇒ *'a*::*comm-semiring-1*) ⇒ *'a*) ⇒ *bool*
where *polyfun N f* = (∃ *p*. *vars p* ⊆ *N* ∧ (∀ *x*. *insertion x p* = *f x*))

lemma *polyfunI*: (⋀*P*. (⋀*p*. *vars p* ⊆ *N* ⟹ (⋀*x*. *insertion x p* = *f x*) ⟹ *P*) ⟹ *P*) ⟹ *polyfun N f*
⟨*proof*⟩

lemma *polyfun-subset*: *N*⊆*N'* ⟹ *polyfun N f* ⟹ *polyfun N' f*
⟨*proof*⟩

lemma *polyfun-const*: *polyfun N* (λ-. *c*)
⟨*proof*⟩

lemma *polyfun-add*:
assumes *polyfun N f polyfun N g*
shows *polyfun N* (λ*x*. *f x* + *g x*)
⟨*proof*⟩

lemma *polyfun-mult*:
assumes *polyfun N f polyfun N g*
shows *polyfun N* (λ*x*. *f x* ∗ *g x*)
⟨*proof*⟩

lemma *polyfun-Sum*:
assumes *finite I*
assumes ⋀*i*. *i*∈*I* ⟹ *polyfun N* (*f i*)
shows *polyfun N* (λ*x*. ∑ *i*∈*I*. *f i x*)
⟨*proof*⟩

lemma *polyfun-Prod*:
assumes *finite I*
assumes ⋀*i*. *i*∈*I* ⟹ *polyfun N* (*f i*)
shows *polyfun N* (λ*x*. ∏ *i*∈*I*. *f i x*)
⟨*proof*⟩

lemma *polyfun-single*:
assumes *i*∈*N*
shows *polyfun N* (λ*x*. *x i*)

⟨*proof*⟩

end

6 Abstract Power-Products

theory *Power-Products*
imports *Complex-Main*
HOL−Library.Function-Algebras
HOL−Library.Countable
More-MPoly-Type
Utils
Well-Quasi-Orders.Well-Quasi-Orders
begin

This theory formalizes the concept of "power-products". A power-product can be thought of as the product of some indeterminates, such as x, $x^2\,y$, $x\,y^3\,z^7$, etc., without any scalar coefficient.

The approach in this theory is to capture the notion of "power-product" (also called "monomial") as type class. A canonical instance for power-product is the type $'var \Rightarrow_0 nat$, which is interpreted as mapping from variables in the power-product to exponents.

A slightly unintuitive (but fitting better with the standard type class instantiations of $'a \Rightarrow_0 'b$) approach is to write addition to denote "multiplication" of power products. For example, x^2y would be represented as a function $p = (X \mapsto 2,\ Y \mapsto 1)$, xz as a function $q = (X \mapsto 1,\ Z \mapsto 1)$. With the (pointwise) instantiation of addition of $'a \Rightarrow_0 'b$, we will write $p + q = (X \mapsto 3,\ Y \mapsto 1,\ Z \mapsto 1)$ for the product $x^2y \cdot xz = x^3yz$

6.1 Constant *Keys*

Legacy:

lemmas *keys-eq-empty-iff* = *keys-eq-empty*

definition *Keys* :: $('a \Rightarrow_0 'b{::}zero)\ set \Rightarrow 'a\ set$
where *Keys F* = $\bigcup$(*keys* ‘ *F*)

lemma *in-Keys*: $s \in Keys\ F \longleftrightarrow (\exists f{\in}F.\ s \in keys\ f)$
⟨*proof*⟩

lemma *in-KeysI*:
assumes $s \in keys\ f$ **and** $f \in F$
shows $s \in Keys\ F$
⟨*proof*⟩

lemma *in-KeysE*:
assumes $s \in Keys\ F$

obtains f **where** $s \in keys\ f$ **and** $f \in F$
⟨*proof*⟩

lemma *Keys-mono*:
assumes $A \subseteq B$
shows $Keys\ A \subseteq Keys\ B$
⟨*proof*⟩

lemma *Keys-insert*: $Keys\ (insert\ a\ A) = keys\ a \cup Keys\ A$
⟨*proof*⟩

lemma *Keys-Un*: $Keys\ (A \cup B) = Keys\ A \cup Keys\ B$
⟨*proof*⟩

lemma *finite-Keys*:
assumes *finite* A
shows *finite* $(Keys\ A)$
⟨*proof*⟩

lemma *Keys-not-empty*:
assumes $a \in A$ **and** $a \neq 0$
shows $Keys\ A \neq \{\}$
⟨*proof*⟩

lemma *Keys-empty* [*simp*]: $Keys\ \{\} = \{\}$
⟨*proof*⟩

lemma *Keys-zero* [*simp*]: $Keys\ \{0\} = \{\}$
⟨*proof*⟩

lemma *keys-subset-Keys*:
assumes $f \in F$
shows $keys\ f \subseteq Keys\ F$
⟨*proof*⟩

lemma *Keys-minus*: $Keys\ (A - B) \subseteq Keys\ A$
⟨*proof*⟩

lemma *Keys-minus-zero*: $Keys\ (A - \{0\}) = Keys\ A$
⟨*proof*⟩

6.2 Constant *except*

definition *except-fun* :: $('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow ('a \Rightarrow 'b::zero)$
where $except\text{-}fun\ f\ S = (\lambda x.\ (f\ x\ when\ x \notin S))$

lift-definition *except* :: $('a \Rightarrow_0 'b) \Rightarrow 'a\ set \Rightarrow ('a \Rightarrow_0 'b::zero)$ **is** *except-fun*
⟨*proof*⟩

lemma *lookup-except-when*: *lookup* (*except p S*) = (λ*t*. *lookup p t when t* ∉ *S*)
⟨*proof*⟩

lemma *lookup-except*: *lookup* (*except p S*) = (λ*t*. *if t* ∈ *S then 0 else lookup p t*)
⟨*proof*⟩

lemma *lookup-except-singleton*: *lookup* (*except p* {*t*}) *t* = *0*
⟨*proof*⟩

lemma *except-zero* [*simp*]: *except 0 S* = *0*
⟨*proof*⟩

lemma *lookup-except-eq-idI*:
assumes *t* ∉ *S*
shows *lookup* (*except p S*) *t* = *lookup p t*
⟨*proof*⟩

lemma *lookup-except-eq-zeroI*:
assumes *t* ∈ *S*
shows *lookup* (*except p S*) *t* = *0*
⟨*proof*⟩

lemma *except-empty* [*simp*]: *except p* {} = *p*
⟨*proof*⟩

lemma *except-eq-zeroI*:
assumes *keys p* ⊆ *S*
shows *except p S* = *0*
⟨*proof*⟩

lemma *except-eq-zeroE*:
assumes *except p S* = *0*
shows *keys p* ⊆ *S*
⟨*proof*⟩

lemma *except-eq-zero-iff*: *except p S* = *0* ⟷ *keys p* ⊆ *S*
⟨*proof*⟩

lemma *except-keys* [*simp*]: *except p* (*keys p*) = *0*
⟨*proof*⟩

lemma *plus-except*: *p* = *Poly-Mapping.single t* (*lookup p t*) + *except p* {*t*}
⟨*proof*⟩

lemma *keys-except*: *keys* (*except p S*) = *keys p* − *S*
⟨*proof*⟩

lemma *except-single*: *except* (*Poly-Mapping.single u c*) *S* = (*Poly-Mapping.single u c when u* ∉ *S*)

⟨*proof*⟩

lemma *except-plus*: *except* (*p* + *q*) *S* = *except p S* + *except q S*
⟨*proof*⟩

lemma *except-minus*: *except* (*p* − *q*) *S* = *except p S* − *except q S*
⟨*proof*⟩

lemma *except-uminus*: *except* (− *p*) *S* = − *except p S*
⟨*proof*⟩

lemma *except-except*: *except* (*except p S*) *T* = *except p* (*S* ∪ *T*)
⟨*proof*⟩

lemma *poly-mapping-keys-eqI*:
assumes *a1*: *keys p* = *keys q* **and** *a2*: ⋀*t*. *t* ∈ *keys p* ⟹ *lookup p t* = *lookup q t*
shows *p* = *q*
⟨*proof*⟩

lemma *except-id-iff*: *except p S* = *p* ⟷ *keys p* ∩ *S* = {}
⟨*proof*⟩

lemma *keys-subset-wf*:
wfP (λ*p q*::('*a*, '*b*::*zero*) *poly-mapping*. *keys p* ⊂ *keys q*)
⟨*proof*⟩

lemma *poly-mapping-except-induct*:
assumes *base*: *P 0* **and** *ind*: ⋀*p t*. *p* ≠ *0* ⟹ *t* ∈ *keys p* ⟹ *P* (*except p* {*t*}) ⟹ *P p*
shows *P p*
⟨*proof*⟩

lemma *poly-mapping-except-induct'*:
assumes ⋀*p*. (⋀*t*. *t* ∈ *keys p* ⟹ *P* (*except p* {*t*})) ⟹ *P p*
shows *P p*
⟨*proof*⟩

lemma *poly-mapping-plus-induct*:
assumes *P 0* **and** ⋀*p c t*. *c* ≠ *0* ⟹ *t* ∉ *keys p* ⟹ *P p* ⟹ *P* (*Poly-Mapping.single t c* + *p*)
shows *P p*
⟨*proof*⟩

lemma *except-Diff-singleton*: *except p* (*keys p* − {*t*}) = *Poly-Mapping.single t* (*lookup p t*)
⟨*proof*⟩

lemma *except-Un-plus-Int*: *except p* (*U* ∪ *V*) + *except p* (*U* ∩ *V*) = *except p U*

$+$ *except p V*
⟨*proof*⟩

corollary *except-Int*:
assumes *keys p* ⊆ *U* ∪ *V*
shows *except p* (*U* ∩ *V*) = *except p U* + *except p V*
⟨*proof*⟩

lemma *except-keys-Int* [*simp*]: *except p* (*keys p* ∩ *U*) = *except p U*
⟨*proof*⟩

lemma *except-Int-keys* [*simp*]: *except p* (*U* ∩ *keys p*) = *except p U*
⟨*proof*⟩

lemma *except-keys-Diff*: *except p* (*keys p* − *U*) = *except p* (− *U*)
⟨*proof*⟩

lemma *except-decomp*: *p* = *except p U* + *except p* (− *U*)
⟨*proof*⟩

corollary *except-Compl*: *except p* (− *U*) = *p* − *except p U*
⟨*proof*⟩

6.3 'Divisibility' on Additive Structures

context *plus* **begin**

definition *adds* :: *'a* ⇒ *'a* ⇒ *bool* (**infix** *adds 50*)
where *b adds a* ⟷ (∃ *k*. *a* = *b* + *k*)

lemma *addsI* [*intro?*]: *a* = *b* + *k* ⟹ *b adds a*
⟨*proof*⟩

lemma *addsE* [*elim?*]: *b adds a* ⟹ (⋀*k*. *a* = *b* + *k* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

end

context *comm-monoid-add*
begin

lemma *adds-refl* [*simp*]: *a adds a*
⟨*proof*⟩

lemma *adds-trans* [*trans*]:
assumes *a adds b* **and** *b adds c*
shows *a adds c*
⟨*proof*⟩

lemma *subset-divisors-adds*: $\{c.\ c\ adds\ a\} \subseteq \{c.\ c\ adds\ b\} \longleftrightarrow a\ adds\ b$
⟨*proof*⟩

lemma *strict-subset-divisors-adds*: $\{c.\ c\ adds\ a\} \subset \{c.\ c\ adds\ b\} \longleftrightarrow a\ adds\ b \wedge \neg\ b\ adds\ a$
⟨*proof*⟩

lemma *zero-adds* [*simp*]: *0 adds a*
⟨*proof*⟩

lemma *adds-plus-right* [*simp*]: $a\ adds\ c \Longrightarrow a\ adds\ (b + c)$
⟨*proof*⟩

lemma *adds-plus-left* [*simp*]: $a\ adds\ b \Longrightarrow a\ adds\ (b + c)$
⟨*proof*⟩

lemma *adds-triv-right* [*simp*]: $a\ adds\ b + a$
⟨*proof*⟩

lemma *adds-triv-left* [*simp*]: $a\ adds\ a + b$
⟨*proof*⟩

lemma *plus-adds-mono*:
assumes $a\ adds\ b$
and $c\ adds\ d$
shows $a + c\ adds\ b + d$
⟨*proof*⟩

lemma *plus-adds-left*: $a + b\ adds\ c \Longrightarrow a\ adds\ c$
⟨*proof*⟩

lemma *plus-adds-right*: $a + b\ adds\ c \Longrightarrow b\ adds\ c$
⟨*proof*⟩

end

class *ninv-comm-monoid-add* = *comm-monoid-add* +
assumes *plus-eq-zero*: $s + t = 0 \Longrightarrow s = 0$
begin

lemma *plus-eq-zero-2*: $t = 0$ **if** $s + t = 0$
⟨*proof*⟩

lemma *adds-zero*: $s\ adds\ 0 \longleftrightarrow (s = 0)$
⟨*proof*⟩

end

context *canonically-ordered-monoid-add*

begin
subclass *ninv-comm-monoid-add* ⟨*proof*⟩
end

class *comm-powerprod* = *cancel-comm-monoid-add*
begin

lemma *adds-canc*: $s + u$ *adds* $t + u \longleftrightarrow s$ *adds* t **for** $s\ t\ u::'a$
⟨*proof*⟩

lemma *adds-canc-2*: $u + s$ *adds* $u + t \longleftrightarrow s$ *adds* t
⟨*proof*⟩

lemma *add-minus-2*: $(s + t) - s = t$
⟨*proof*⟩

lemma *adds-minus*:
assumes s *adds* t
shows $(t - s) + s = t$
⟨*proof*⟩

lemma *plus-adds-0*:
assumes $(s + t)$ *adds* u
shows s *adds* $(u - t)$
⟨*proof*⟩

lemma *plus-adds-2*:
assumes t *adds* u **and** s *adds* $(u - t)$
shows $(s + t)$ *adds* u
⟨*proof*⟩

lemma *plus-adds*:
shows $(s + t)$ *adds* $u \longleftrightarrow (t$ *adds* $u \wedge s$ *adds* $(u - t))$
⟨*proof*⟩

lemma *minus-plus*:
assumes s *adds* t
shows $(t - s) + u = (t + u) - s$
⟨*proof*⟩

lemma *minus-plus-minus*:
assumes s *adds* t **and** u *adds* v
shows $(t - s) + (v - u) = (t + v) - (s + u)$
⟨*proof*⟩

lemma *minus-plus-minus-cancel*:
assumes u *adds* t **and** s *adds* u
shows $(t - u) + (u - s) = t - s$
⟨*proof*⟩

end

Instances of class *lcs-powerprod* are types of commutative power-products admitting (not necessarily unique) least common sums (inspired from least common multiplies). Note that if the components of indeterminates are arbitrary integers (as for instance in Laurent polynomials), then no unique lcss exist.

class *lcs-powerprod* = *comm-powerprod* +
  fixes *lcs*::'*a* ⇒ '*a* ⇒ '*a*
  assumes *adds-lcs*: *s adds* (*lcs s t*)
  assumes *lcs-adds*: *s adds u* ⟹ *t adds u* ⟹ (*lcs s t*) *adds u*
  assumes *lcs-comm*: *lcs s t* = *lcs t s*
begin

lemma *adds-lcs-2*: *t adds* (*lcs s t*)
  ⟨*proof*⟩

lemma *lcs-adds-plus*: *lcs s t adds s* + *t* ⟨*proof*⟩

"gcs" stands for "greatest common summand".

definition *gcs* :: '*a* ⇒ '*a* ⇒ '*a* **where** *gcs s t* = (*s* + *t*) − (*lcs s t*)

lemma *gcs-plus-lcs*: (*gcs s t*) + (*lcs s t*) = *s* + *t*
  ⟨*proof*⟩

lemma *gcs-adds*: (*gcs s t*) *adds s*
⟨*proof*⟩

lemma *gcs-comm*: *gcs s t* = *gcs t s* ⟨*proof*⟩

lemma *gcs-adds-2*: (*gcs s t*) *adds t*
  ⟨*proof*⟩

end

class *ulcs-powerprod* = *lcs-powerprod* + *ninv-comm-monoid-add*
begin

lemma *adds-antisym*:
  assumes *s adds t t adds s*
  shows *s* = *t*
⟨*proof*⟩

lemma *lcs-unique*:
  assumes *s adds l* **and** *t adds l* **and** ∗: ⋀*u*. *s adds u* ⟹ *t adds u* ⟹ *l adds u*
  shows *l* = *lcs s t*
  ⟨*proof*⟩

lemma *lcs-zero*: *lcs 0 t* = *t*
⟨*proof*⟩

lemma *lcs-plus-left*: *lcs* (*u* + *s*) (*u* + *t*) = *u* + *lcs s t*
⟨*proof*⟩

lemma *lcs-plus-right*: *lcs* (*s* + *u*) (*t* + *u*) = (*lcs s t*) + *u*
⟨*proof*⟩

lemma *adds-gcs*:
assumes *u adds s* **and** *u adds t*
shows *u adds* (*gcs s t*)
⟨*proof*⟩

lemma *gcs-unique*:
assumes *g adds s* **and** *g adds t* **and** ∗: ⋀*u*. *u adds s* ⟹ *u adds t* ⟹ *u adds g*
shows *g* = *gcs s t*
⟨*proof*⟩

lemma *gcs-plus-left*: *gcs* (*u* + *s*) (*u* + *t*) = *u* + *gcs s t*
⟨*proof*⟩

lemma *gcs-plus-right*: *gcs* (*s* + *u*) (*t* + *u*) = (*gcs s t*) + *u*
⟨*proof*⟩

lemma *lcs-same* [*simp*]: *lcs s s* = *s*
⟨*proof*⟩

lemma *gcs-same* [*simp*]: *gcs s s* = *s*
⟨*proof*⟩

end

6.4 Dickson Classes

definition (**in** *plus*) *dickson-grading* :: ('*a* ⇒ *nat*) ⇒ *bool*
where *dickson-grading d* ⟷
((∀ *s t*. *d* (*s* + *t*) = *max* (*d s*) (*d t*)) ∧ (∀ *n*::*nat*. *almost-full-on* (*adds*) {*x*. *d x* ≤ *n*}))

definition *dgrad-set* :: ('*a* ⇒ *nat*) ⇒ *nat* ⇒ '*a set*
where *dgrad-set d m* = {*t*. *d t* ≤ *m*}

definition *dgrad-set-le* :: ('*a* ⇒ *nat*) ⇒ ('*a set*) ⇒ ('*a set*) ⇒ *bool*
where *dgrad-set-le d S T* ⟷ (∀ *s*∈*S*. ∃ *t*∈*T*. *d s* ≤ *d t*)

lemma *dickson-gradingI*:
assumes ⋀*s t*. *d* (*s* + *t*) = *max* (*d s*) (*d t*)
assumes ⋀*n*::*nat*. *almost-full-on* (*adds*) {*x*. *d x* ≤ *n*}

shows *dickson-grading d*
⟨*proof*⟩

lemma *dickson-gradingD1*: *dickson-grading d* ⟹ *d* (*s* + *t*) = *max* (*d s*) (*d t*)
⟨*proof*⟩

lemma *dickson-gradingD2*: *dickson-grading d* ⟹ *almost-full-on* (*adds*) {*x*. *d x* ≤ *n*}
⟨*proof*⟩

lemma *dickson-gradingD2'*:
assumes *dickson-grading* (*d*::'*a*::*comm-monoid-add* ⇒ *nat*)
shows *wqo-on* (*adds*) {*x*. *d x* ≤ *n*}
⟨*proof*⟩

lemma *dickson-gradingE*:
assumes *dickson-grading d* **and** ⋀*i*::*nat*. *d* ((*seq*::*nat* ⇒ '*a*::*plus*) *i*) ≤ *n*
obtains *i j* **where** *i* < *j* **and** *seq i adds seq j*
⟨*proof*⟩

lemma *dickson-grading-adds-imp-le*:
assumes *dickson-grading d* **and** *s adds t*
shows *d s* ≤ *d t*
⟨*proof*⟩

lemma *dickson-grading-minus*:
assumes *dickson-grading d* **and** *s adds* (*t*::'*a*::*cancel-ab-semigroup-add*)
shows *d* (*t* − *s*) ≤ *d t*
⟨*proof*⟩

lemma *dickson-grading-lcs*:
assumes *dickson-grading d*
shows *d* (*lcs s t*) ≤ *max* (*d s*) (*d t*)
⟨*proof*⟩

lemma *dickson-grading-lcs-minus*:
assumes *dickson-grading d*
shows *d* (*lcs s t* − *s*) ≤ *max* (*d s*) (*d t*)
⟨*proof*⟩

lemma *dgrad-set-leI*:
assumes ⋀*s*. *s* ∈ *S* ⟹ ∃ *t*∈*T*. *d s* ≤ *d t*
shows *dgrad-set-le d S T*
⟨*proof*⟩

lemma *dgrad-set-leE*:
assumes *dgrad-set-le d S T* **and** *s* ∈ *S*
obtains *t* **where** *t* ∈ *T* **and** *d s* ≤ *d t*
⟨*proof*⟩

lemma *dgrad-set-exhaust-expl*:
assumes *finite F*
shows $F \subseteq$ *dgrad-set d* (*Max* (*d* ‘ *F*))
⟨*proof*⟩

lemma *dgrad-set-exhaust*:
assumes *finite F*
obtains *m* **where** $F \subseteq$ *dgrad-set d m*
⟨*proof*⟩

lemma *dgrad-set-le-trans* [*trans*]:
assumes *dgrad-set-le d S T* **and** *dgrad-set-le d T U*
shows *dgrad-set-le d S U*
⟨*proof*⟩

lemma *dgrad-set-le-Un*: *dgrad-set-le d* ($S \cup T$) *U* $\longleftrightarrow$ (*dgrad-set-le d S U* $\wedge$ *dgrad-set-le d T U*)
⟨*proof*⟩

lemma *dgrad-set-le-subset*:
assumes $S \subseteq T$
shows *dgrad-set-le d S T*
⟨*proof*⟩

lemma *dgrad-set-le-refl*: *dgrad-set-le d S S*
⟨*proof*⟩

lemma *dgrad-set-le-dgrad-set*:
assumes *dgrad-set-le d F G* **and** $G \subseteq$ *dgrad-set d m*
shows $F \subseteq$ *dgrad-set d m*
⟨*proof*⟩

lemma *dgrad-set-dgrad*: $p \in$ *dgrad-set d* (*d p*)
⟨*proof*⟩

lemma *dgrad-setI* [*intro*]:
assumes *d t* $\leq m$
shows $t \in$ *dgrad-set d m*
⟨*proof*⟩

lemma *dgrad-setD*:
assumes $t \in$ *dgrad-set d m*
shows *d t* $\leq m$
⟨*proof*⟩

lemma *dgrad-set-zero* [*simp*]: *dgrad-set* (λ-. *0*) *m* = *UNIV*
⟨*proof*⟩

lemma *subset-dgrad-set-zero*: $F \subseteq$ *dgrad-set* (λ-. *0*) *m*
⟨*proof*⟩

lemma *dgrad-set-subset*:
assumes $m \leq n$
shows *dgrad-set d m* $\subseteq$ *dgrad-set d n*
⟨*proof*⟩

lemma *dgrad-set-closed-plus*:
assumes *dickson-grading d* **and** $s \in$ *dgrad-set d m* **and** $t \in$ *dgrad-set d m*
shows $s + t \in$ *dgrad-set d m*
⟨*proof*⟩

lemma *dgrad-set-closed-minus*:
assumes *dickson-grading d* **and** $s \in$ *dgrad-set d m* **and** *t adds* (s::$'a$::*cancel-ab-semigroup-add*)
shows $s - t \in$ *dgrad-set d m*
⟨*proof*⟩

lemma *dgrad-set-closed-lcs*:
assumes *dickson-grading d* **and** $s \in$ *dgrad-set d m* **and** $t \in$ *dgrad-set d m*
shows *lcs s t* $\in$ *dgrad-set d m*
⟨*proof*⟩

lemma *dickson-gradingD-dgrad-set*: *dickson-grading d* $\Longrightarrow$ *almost-full-on* (*adds*) (*dgrad-set d m*)
⟨*proof*⟩

lemma *ex-finite-adds*:
assumes *dickson-grading d* **and** $S \subseteq$ *dgrad-set d m*
obtains *T* **where** *finite T* **and** $T \subseteq S$ **and** $\bigwedge s.\ s \in S \Longrightarrow (\exists\, t \in T.\ t$ *adds* (s::$'a$::*cancel-comm-monoid-add*))
⟨*proof*⟩

class *graded-dickson-powerprod* = *ulcs-powerprod* +
assumes *ex-dgrad*: $\exists\, d$::$'a \Rightarrow$ *nat. dickson-grading d*
begin

definition *dgrad-dummy* **where** *dgrad-dummy* = (*SOME d. dickson-grading d*)

lemma *dickson-grading-dgrad-dummy*: *dickson-grading dgrad-dummy*
⟨*proof*⟩

end

class *dickson-powerprod* = *ulcs-powerprod* +
assumes *dickson*: *almost-full-on* (*adds*) *UNIV*
begin

lemma *dickson-grading-zero*: *dickson-grading* (λ-::$'a$. *0*)

⟨*proof*⟩

subclass *graded-dickson-powerprod* ⟨*proof*⟩

end

Class *graded-dickson-powerprod* is a slightly artificial construction. It is needed, because type $nat \Rightarrow_0 nat$ does not satisfy the usual conditions of a "Dickson domain" (as formulated in class *dickson-powerprod*), but we still want to use that type as the type of power-products in the computation of Gröbner bases. So, we exploit the fact that in a finite set of polynomials (which is the input of Buchberger's algorithm) there is always some "highest" indeterminate that occurs with non-zero exponent, and no "higher" indeterminates are generated during the execution of the algorithm. This allows us to prove that the algorithm terminates, even though there are in principle infinitely many indeterminates.

6.5 Additive Linear Orderings

lemma *group-eq-aux*: $a + (b - a) = (b::'a::ab\text{-}group\text{-}add)$
⟨*proof*⟩

class *semi-canonically-ordered-monoid-add* = *ordered-comm-monoid-add* +
assumes *le-imp-add*: $a \leq b \Longrightarrow (\exists\, c.\ b = a + c)$

context *canonically-ordered-monoid-add*
begin
subclass *semi-canonically-ordered-monoid-add*
⟨*proof*⟩
end

class *add-linorder-group* = *ordered-ab-semigroup-add-imp-le* + *ab-group-add* + *linorder*

class *add-linorder* = *ordered-ab-semigroup-add-imp-le* + *cancel-comm-monoid-add*
+ *semi-canonically-ordered-monoid-add* + *linorder*
begin

subclass *ordered-comm-monoid-add* ⟨*proof*⟩

subclass *ordered-cancel-comm-monoid-add* ⟨*proof*⟩

lemma *le-imp-inv*:
assumes $a \leq b$
shows $b = a + (b - a)$
⟨*proof*⟩

lemma *max-eq-sum*:
obtains y **where** $max\ a\ b = a + y$

⟨*proof*⟩

lemma *min-plus-max*:
shows $(min\ a\ b) + (max\ a\ b) = a + b$
⟨*proof*⟩

end

class *add-linorder-min* = *add-linorder* +
assumes *zero-min*: $0 \leq x$
begin

subclass *ninv-comm-monoid-add*
⟨*proof*⟩

lemma *leq-add-right*:
shows $x \leq x + y$
⟨*proof*⟩

lemma *leq-add-left*:
shows $x \leq y + x$
⟨*proof*⟩

subclass *canonically-ordered-monoid-add*
⟨*proof*⟩

end

class *add-wellorder* = *add-linorder-min* + *wellorder*

instantiation *nat* :: *add-linorder*
begin

instance ⟨*proof*⟩

end

instantiation *nat* :: *add-linorder-min*
begin
instance ⟨*proof*⟩
end

instantiation *nat* :: *add-wellorder*
begin
instance ⟨*proof*⟩
end

context *add-linorder-group*
begin

subclass *add-linorder*
⟨*proof*⟩

end

instantiation *int* :: *add-linorder-group*
begin
instance ⟨*proof*⟩
end

instantiation *rat* :: *add-linorder-group*
begin
instance ⟨*proof*⟩
end

instantiation *real* :: *add-linorder-group*
begin
instance ⟨*proof*⟩
end

6.6 Ordered Power-Products

locale *ordered-powerprod* =
  ordered-powerprod-lin: *linorder ord ord-strict*
  for *ord*::'*a* ⇒ '*a*::*comm-powerprod* ⇒ *bool* (**infixl** ⪯ *50*)
  and *ord-strict*::'*a* ⇒ '*a*::*comm-powerprod* ⇒ *bool* (**infixl** ≺ *50*) +
  assumes *zero-min*: *0* ⪯ *t*
  assumes *plus-monotone*: *s* ⪯ *t* ⟹ *s* + *u* ⪯ *t* + *u*
begin

abbreviation *ord-conv* (**infixl** ⪰ *50*) **where** *ord-conv* ≡ (⪯)$^{-1-1}$
abbreviation *ord-strict-conv* (**infixl** ≻ *50*) **where** *ord-strict-conv* ≡ (≺)$^{-1-1}$

lemma *ord-canc*:
  assumes *s* + *u* ⪯ *t* + *u*
  shows *s* ⪯ *t*
⟨*proof*⟩

lemma *ord-adds*:
  assumes *s adds t*
  shows *s* ⪯ *t*
⟨*proof*⟩

lemma *ord-canc-left*:
  assumes *u* + *s* ⪯ *u* + *t*
  shows *s* ⪯ *t*
  ⟨*proof*⟩

lemma *ord-strict-canc*:
assumes $s + u \prec t + u$
shows $s \prec t$
⟨*proof*⟩

lemma *ord-strict-canc-left*:
assumes $u + s \prec u + t$
shows $s \prec t$
⟨*proof*⟩

lemma *plus-monotone-left*:
assumes $s \preceq t$
shows $u + s \preceq u + t$
⟨*proof*⟩

lemma *plus-monotone-strict*:
assumes $s \prec t$
shows $s + u \prec t + u$
⟨*proof*⟩

lemma *plus-monotone-strict-left*:
assumes $s \prec t$
shows $u + s \prec u + t$
⟨*proof*⟩

end

locale *gd-powerprod* =
ordered-powerprod ord ord-strict
for *ord*::'*a* ⇒ '*a*::*graded-dickson-powerprod* ⇒ *bool* (**infixl** $\preceq$ *50*)
and *ord-strict* (**infixl** $\prec$ *50*)
begin

definition *dickson-le* :: ('*a* ⇒ *nat*) ⇒ *nat* ⇒ '*a* ⇒ '*a* ⇒ *bool*
where *dickson-le d m s t* ⟷ (*d s* ≤ *m* ∧ *d t* ≤ *m* ∧ *s* $\preceq$ *t*)

definition *dickson-less* :: ('*a* ⇒ *nat*) ⇒ *nat* ⇒ '*a* ⇒ '*a* ⇒ *bool*
where *dickson-less d m s t* ⟷ (*d s* ≤ *m* ∧ *d t* ≤ *m* ∧ *s* $\prec$ *t*)

lemma *dickson-leI*:
assumes *d s* ≤ *m* **and** *d t* ≤ *m* **and** *s* $\preceq$ *t*
shows *dickson-le d m s t*
⟨*proof*⟩

lemma *dickson-leD1*:
assumes *dickson-le d m s t*
shows *d s* ≤ *m*
⟨*proof*⟩

lemma *dickson-leD2*:
assumes *dickson-le d m s t*
shows $d\ t \leq m$
⟨*proof*⟩

lemma *dickson-leD3*:
assumes *dickson-le d m s t*
shows $s \preceq t$
⟨*proof*⟩

lemma *dickson-le-trans*:
assumes *dickson-le d m s t* **and** *dickson-le d m t u*
shows *dickson-le d m s u*
⟨*proof*⟩

lemma *dickson-lessI*:
assumes $d\ s \leq m$ **and** $d\ t \leq m$ **and** $s \prec t$
shows *dickson-less d m s t*
⟨*proof*⟩

lemma *dickson-lessD1*:
assumes *dickson-less d m s t*
shows $d\ s \leq m$
⟨*proof*⟩

lemma *dickson-lessD2*:
assumes *dickson-less d m s t*
shows $d\ t \leq m$
⟨*proof*⟩

lemma *dickson-lessD3*:
assumes *dickson-less d m s t*
shows $s \prec t$
⟨*proof*⟩

lemma *dickson-less-irrefl*: $\neg$ *dickson-less d m t t*
⟨*proof*⟩

lemma *dickson-less-trans*:
assumes *dickson-less d m s t* **and** *dickson-less d m t u*
shows *dickson-less d m s u*
⟨*proof*⟩

lemma *transp-dickson-less*: *transp* (*dickson-less d m*)
⟨*proof*⟩

lemma *wfp-on-ord-strict*:
assumes *dickson-grading d*
shows *wfp-on* $(\prec)$ $\{x.\ d\ x \leq n\}$

⟨*proof*⟩

lemma *wf-dickson-less*:
  assumes *dickson-grading d*
  shows *wfP* (*dickson-less d m*)
⟨*proof*⟩

end

gd-powerprod stands for *graded ordered Dickson power-products*.

locale *od-powerprod* =
  ordered-powerprod ord ord-strict
  for *ord*::′*a* ⇒ ′*a*::*dickson-powerprod* ⇒ *bool* (**infixl** ⪯ *50*)
  and *ord-strict* (**infixl** ≺ *50*)
begin

sublocale *gd-powerprod* ⟨*proof*⟩

lemma *wf-ord-strict*: *wfP* (≺)
⟨*proof*⟩

end

od-powerprod stands for *ordered Dickson power-products*.

6.7 Functions as Power-Products

lemma *finite-neq-0*:
  assumes *fin-A*: *finite* {*x*. *f x* ≠ *0*} **and** *fin-B*: *finite* {*x*. *g x* ≠ *0*} **and** ⋀*x*. *h x 0 0* = *0*
  shows *finite* {*x*. *h x* (*f x*) (*g x*) ≠ *0*}
⟨*proof*⟩

lemma *finite-neq-0′*:
  assumes *finite* {*x*. *f x* ≠ *0*} **and** *finite* {*x*. *g x* ≠ *0*} **and** *h 0 0* = *0*
  shows *finite* {*x*. *h* (*f x*) (*g x*) ≠ *0*}
  ⟨*proof*⟩

lemma *finite-neq-0-inv*:
  assumes *fin-A*: *finite* {*x*. *h x* (*f x*) (*g x*) ≠ *0*} **and** *fin-B*: *finite* {*x*. *f x* ≠ *0*} **and** ⋀*x y*. *h x 0 y* = *y*
  shows *finite* {*x*. *g x* ≠ *0*}
⟨*proof*⟩

lemma *finite-neq-0-inv′*:
  assumes *inf-A*: *finite* {*x*. *h* (*f x*) (*g x*) ≠ *0*} **and** *fin-B*: *finite* {*x*. *f x* ≠ *0*} **and** ⋀*x*. *h 0 x* = *x*
  shows *finite* {*x*. *g x* ≠ *0*}
  ⟨*proof*⟩

6.7.1 $'a \Rightarrow 'b$ belongs to class *comm-powerprod*

instance *fun* :: (*type*, *cancel-comm-monoid-add*) *comm-powerprod*
⟨*proof*⟩

6.7.2 $'a \Rightarrow 'b$ belongs to class *ninv-comm-monoid-add*

instance *fun* :: (*type*, *ninv-comm-monoid-add*) *ninv-comm-monoid-add*
⟨*proof*⟩

6.7.3 $'a \Rightarrow 'b$ belongs to class *lcs-powerprod*

instantiation *fun* :: (*type*, *add-linorder*) *lcs-powerprod*
begin

definition *lcs-fun*::$('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$ **where** *lcs f g* = $(\lambda x.$ *max* (*f x*) (*g x*))

lemma *adds-funI*:
assumes $s \leq t$
shows *s adds* (t::$'a \Rightarrow 'b$)
⟨*proof*⟩

lemma *adds-fun-iff*: *f adds* (g::$'a \Rightarrow 'b$) $\longleftrightarrow$ $(\forall x.$ *f x adds g x*)
⟨*proof*⟩

lemma *adds-fun-iff'*: *f adds* (g::$'a \Rightarrow 'b$) $\longleftrightarrow$ $(\forall x.\ \exists y.$ *g x* = *f x* + *y*)
⟨*proof*⟩

lemma *adds-lcs-fun*:
shows *s adds* (*lcs s* (t::$'a \Rightarrow 'b$))
⟨*proof*⟩

lemma *lcs-comm-fun*: *lcs s t* = *lcs t* (s::$'a \Rightarrow 'b$)
⟨*proof*⟩

lemma *lcs-adds-fun*:
assumes *s adds u* **and** *t adds* (u::$'a \Rightarrow 'b$)
shows (*lcs s t*) *adds u*
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *leq-lcs-fun-1*: $s \leq$ (*lcs s* (t::$'a \Rightarrow 'b$::*add-linorder*))
⟨*proof*⟩

lemma *leq-lcs-fun-2*: $t \leq$ (*lcs s* (t::$'a \Rightarrow 'b$::*add-linorder*))

⟨*proof*⟩

lemma *lcs-leq-fun*:
assumes $s \leq u$ **and** $t \leq (u::'a \Rightarrow 'b::add\text{-}linorder)$
shows $(lcs\ s\ t) \leq u$
⟨*proof*⟩

lemma *adds-fun*: $s\ adds\ t \longleftrightarrow s \leq t$
for $s\ t::'a \Rightarrow 'b::add\text{-}linorder\text{-}min$
⟨*proof*⟩

lemma *gcs-fun*: $gcs\ s\ (t::'a \Rightarrow ('b::add\text{-}linorder)) = (\lambda x.\ min\ (s\ x)\ (t\ x))$
⟨*proof*⟩

lemma *gcs-leq-fun-1*: $(gcs\ s\ (t::'a \Rightarrow 'b::add\text{-}linorder)) \leq s$
⟨*proof*⟩

lemma *gcs-leq-fun-2*: $(gcs\ s\ (t::'a \Rightarrow 'b::add\text{-}linorder)) \leq t$
⟨*proof*⟩

lemma *leq-gcs-fun*:
assumes $u \leq s$ **and** $u \leq (t::'a \Rightarrow 'b::add\text{-}linorder)$
shows $u \leq (gcs\ s\ t)$
⟨*proof*⟩

6.7.4 $'a \Rightarrow 'b$ belongs to class *ulcs-powerprod*

instance *fun* :: (*type*, *add-linorder-min*) *ulcs-powerprod* ⟨*proof*⟩

6.7.5 Power-products in a given set of indeterminates

definition $supp\text{-}fun::('a \Rightarrow 'b::zero) \Rightarrow 'a\ set$ **where** $supp\text{-}fun\ f = \{x.\ f\ x \neq 0\}$

supp-fun for general functions is like *keys* for *poly-mapping*, but does not need to be finite.

lemma *keys-eq-supp*: $keys\ s = supp\text{-}fun\ (lookup\ s)$
⟨*proof*⟩

lemma *supp-fun-zero* [*simp*]: $supp\text{-}fun\ 0 = \{\}$
⟨*proof*⟩

lemma *supp-fun-eq-zero-iff*: $supp\text{-}fun\ f = \{\} \longleftrightarrow f = 0$
⟨*proof*⟩

lemma *sub-supp-empty*: $supp\text{-}fun\ s \subseteq \{\} \longleftrightarrow (s = 0)$
⟨*proof*⟩

lemma *except-fun-idI*: $supp\text{-}fun\ f \cap V = \{\} \Longrightarrow except\text{-}fun\ f\ V = f$
⟨*proof*⟩

lemma *supp-except-fun*: *supp-fun* (*except-fun s V*) = *supp-fun s* − *V*
⟨*proof*⟩

lemma *supp-fun-plus-subset*: *supp-fun* (*s* + *t*) ⊆ *supp-fun s* ∪ *supp-fun* (*t*::′*a* ⇒ ′*b*::*monoid-add*)
⟨*proof*⟩

lemma *fun-eq-zeroI*:
assumes ⋀*x*. *x* ∈ *supp-fun f* ⟹ *f x* = *0*
shows *f* = *0*
⟨*proof*⟩

lemma *except-fun-cong1*:
supp-fun s ∩ ((*V* − *U*) ∪ (*U* − *V*)) ⊆ {} ⟹ *except-fun s V* = *except-fun s U*
⟨*proof*⟩

lemma *adds-except-fun*:
s adds t = (*except-fun s V adds except-fun t V* ∧ *except-fun s* (− *V*) *adds except-fun t* (− *V*))
for *s t* :: ′*a* ⇒ ′*b*::*add-linorder*
⟨*proof*⟩

lemma *adds-except-fun-singleton*: *s adds t* = (*except-fun s* {*v*} *adds except-fun t* {*v*} ∧ *s v adds t v*)
for *s t* :: ′*a* ⇒ ′*b*::*add-linorder*
⟨*proof*⟩

6.7.6 Dickson's lemma for power-products in finitely many indeterminates

lemma *Dickson-fun*:
assumes *finite V*
shows *almost-full-on* (*adds*) {*x*::′*a* ⇒ ′*b*::*add-wellorder*. *supp-fun x* ⊆ *V*}
⟨*proof*⟩

instance *fun* :: (*finite*, *add-wellorder*) *dickson-powerprod*
⟨*proof*⟩

6.7.7 Lexicographic Term Order

Term orders are certain linear orders on power-products, satisfying additional requirements. Further information on term orders can be found, e. g., in [4].

context *wellorder*
begin

lemma *neq-fun-alt*:
assumes *s* ≠ (*t*::′*a* ⇒ ′*b*)
obtains *x* **where** *s x* ≠ *t x* **and** ⋀*y*. *s y* ≠ *t y* ⟹ *x* ≤ *y*

⟨*proof*⟩

definition *lex-fun*::('*a* ⇒ '*b*) ⇒ ('*a* ⇒ '*b*::*order*) ⇒ *bool* **where**
lex-fun s t ≡ (∀ *x*. *s x* ≤ *t x* ∨ (∃ *y*<*x*. *s y* ≠ *t y*))

definition *lex-fun-strict s t* ⟷ *lex-fun s t* ∧ ¬ *lex-fun t s*

Attention! *lex-fun* reverses the order of the indeterminates: if x is smaller than y w.r.t. the order on '*a*, then the *power-product* x is *greater* than the *power-product* y.

lemma *lex-fun-alt*:
shows *lex-fun s t* = (*s* = *t* ∨ (∃ *x*. *s x* < *t x* ∧ (∀ *y*<*x*. *s y* = *t y*))) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *lex-fun-refl*: *lex-fun s s*
⟨*proof*⟩

lemma *lex-fun-antisym*:
assumes *lex-fun s t* **and** *lex-fun t s*
shows *s* = *t*
⟨*proof*⟩

lemma *lex-fun-trans*:
assumes *lex-fun s t* **and** *lex-fun t u*
shows *lex-fun s u*
⟨*proof*⟩

lemma *lex-fun-lin*: *lex-fun s t* ∨ *lex-fun t s* **for** *s t*::'*a* ⇒ '*b*::{*ordered-comm-monoid-add*, *linorder*}
⟨*proof*⟩

corollary *lex-fun-strict-alt* [*code*]:
lex-fun-strict s t = (¬ *lex-fun t s*) **for** *s t*::'*a* ⇒ '*b*::{*ordered-comm-monoid-add*, *linorder*}
⟨*proof*⟩

lemma *lex-fun-zero-min*: *lex-fun 0 s* **for** *s*::'*a* ⇒ '*b*::*add-linorder-min*
⟨*proof*⟩

lemma *lex-fun-plus-monotone*:
lex-fun (*s* + *u*) (*t* + *u*) **if** *lex-fun s t*
for *s t*::'*a* ⇒ '*b*::*ordered-cancel-comm-monoid-add*
⟨*proof*⟩

end

6.7.8 Degree

definition *deg-fun*::('*a* ⇒ '*b*::*comm-monoid-add*) ⇒ '*b* **where** *deg-fun s* ≡ ∑ *x*∈(*supp-fun s*). *s x*

lemma *deg-fun-zero*[*simp*]: *deg-fun 0 = 0*
⟨*proof*⟩

lemma *deg-fun-eq-0-iff*:
assumes *finite* (*supp-fun* (*s*::'*a* ⇒ '*b*::*add-linorder-min*))
shows *deg-fun s = 0* ⟷ *s = 0*
⟨*proof*⟩

lemma *deg-fun-superset*:
fixes *A*::'*a set*
assumes *supp-fun s* ⊆ *A* **and** *finite A*
shows *deg-fun s* = (∑ *x*∈*A*. *s x*)
⟨*proof*⟩

lemma *deg-fun-plus*:
assumes *finite* (*supp-fun s*) **and** *finite* (*supp-fun t*)
shows *deg-fun* (*s + t*) = *deg-fun s* + *deg-fun* (*t*::'*a* ⇒ '*b*::*comm-monoid-add*)
⟨*proof*⟩

lemma *deg-fun-leq*:
assumes *finite* (*supp-fun s*) **and** *finite* (*supp-fun t*) **and** *s* ≤ (*t*::'*a* ⇒ '*b*::*ordered-comm-monoid-add*)
shows *deg-fun s* ≤ *deg-fun t*
⟨*proof*⟩

6.7.9 General Degree-Orders

context *linorder*
begin

lemma *ex-min*:
assumes *finite* (*A*::'*a set*) **and** *A* ≠ {}
shows ∃ *y*∈*A*. (∀ *z*∈*A*. *y* ≤ *z*)
⟨*proof*⟩

definition *dord-fun*::((*'a* ⇒ *'b*::*ordered-comm-monoid-add*) ⇒ (*'a* ⇒ *'b*) ⇒ *bool*)
⇒ (*'a* ⇒ *'b*) ⇒ (*'a* ⇒ *'b*) ⇒ *bool*
where *dord-fun ord s t* ≡ (*let d1 = deg-fun s*; *d2 = deg-fun t in* (*d1 < d2* ∨
(*d1 = d2* ∧ *ord s t*)))

lemma *dord-fun-degD*:
assumes *dord-fun ord s t*
shows *deg-fun s* ≤ *deg-fun t*
⟨*proof*⟩

lemma *dord-fun-refl*:

assumes *ord s s*
shows *dord-fun ord s s*
⟨*proof*⟩

lemma *dord-fun-antisym*:
assumes *ord-antisym*: *ord s t* ⟹ *ord t s* ⟹ *s* = *t* **and** *dord-fun ord s t* **and** *dord-fun ord t s*
shows *s* = *t*
⟨*proof*⟩

lemma *dord-fun-trans*:
assumes *ord-trans*: *ord s t* ⟹ *ord t u* ⟹ *ord s u* **and** *dord-fun ord s t* **and** *dord-fun ord t u*
shows *dord-fun ord s u*
⟨*proof*⟩

lemma *dord-fun-lin*:
dord-fun ord s t ∨ *dord-fun ord t s*
if *ord s t* ∨ *ord t s*
for *s t*::*'a* ⇒ *'b*::{*ordered-comm-monoid-add*, *linorder*}
⟨*proof*⟩

lemma *dord-fun-zero-min*:
fixes *s t*::*'a* ⇒ *'b*::*add-linorder-min*
assumes *ord-refl*: ⋀*t*. *ord t t* **and** *finite* (*supp-fun s*)
shows *dord-fun ord 0 s*
⟨*proof*⟩

lemma *dord-fun-plus-monotone*:
fixes *s t u* ::*'a* ⇒ *'b*::{*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*}
assumes *ord-monotone*: *ord s t* ⟹ *ord* (*s* + *u*) (*t* + *u*) **and** *finite* (*supp-fun s*)
and *finite* (*supp-fun t*) **and** *finite* (*supp-fun u*) **and** *dord-fun ord s t*
shows *dord-fun ord* (*s* + *u*) (*t* + *u*)
⟨*proof*⟩

end

context *wellorder*
begin

6.7.10 Degree-Lexicographic Term Order

definition *dlex-fun*::(*'a* ⇒ *'b*::*ordered-comm-monoid-add*) ⇒ (*'a* ⇒ *'b*) ⇒ *bool*
where *dlex-fun* ≡ *dord-fun lex-fun*

definition *dlex-fun-strict s t* ⟷ *dlex-fun s t* ∧ ¬ *dlex-fun t s*

lemma *dlex-fun-refl*:
shows *dlex-fun s s*

⟨*proof*⟩

lemma *dlex-fun-antisym*:
assumes *dlex-fun s t* **and** *dlex-fun t s*
shows $s = t$
⟨*proof*⟩

lemma *dlex-fun-trans*:
assumes *dlex-fun s t* **and** *dlex-fun t u*
shows *dlex-fun s u*
⟨*proof*⟩

lemma *dlex-fun-lin*: *dlex-fun s t* ∨ *dlex-fun t s*
for *s t*::('*a* ⇒ '*b*::{*ordered-comm-monoid-add*, *linorder*})
⟨*proof*⟩

corollary *dlex-fun-strict-alt* [*code*]:
dlex-fun-strict s t = (¬ *dlex-fun t s*) **for** *s t*::'*a* ⇒ '*b*::{*ordered-comm-monoid-add*, *linorder*}
⟨*proof*⟩

lemma *dlex-fun-zero-min*:
fixes *s t*::('*a* ⇒ '*b*::*add-linorder-min*)
assumes *finite* (*supp-fun s*)
shows *dlex-fun 0 s*
⟨*proof*⟩

lemma *dlex-fun-plus-monotone*:
fixes *s t u*::'*a* ⇒ '*b*::{*ordered-cancel-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*}
assumes *finite* (*supp-fun s*) **and** *finite* (*supp-fun t*) **and** *finite* (*supp-fun u*) **and** *dlex-fun s t*
shows *dlex-fun* (*s* + *u*) (*t* + *u*)
⟨*proof*⟩

6.7.11 Degree-Reverse-Lexicographic Term Order

abbreviation *rlex-fun*::('*a* ⇒ '*b*) ⇒ ('*a* ⇒ '*b*::*order*) ⇒ *bool* **where**
rlex-fun s t ≡ *lex-fun t s*

Note that *rlex-fun* is not precisely the reverse-lexicographic order relation on power-products. Normally, the *last* (i. e. highest) indeterminate whose exponent differs in the two power-products to be compared is taken, but since we do not require the domain to be finite, there might not be such a last indeterminate. Therefore, we simply take the converse of *lex-fun*.

definition *drlex-fun*::('*a* ⇒ '*b*::*ordered-comm-monoid-add*) ⇒ ('*a* ⇒ '*b*) ⇒ *bool*
where *drlex-fun* ≡ *dord-fun rlex-fun*

definition *drlex-fun-strict s t* ⟷ *drlex-fun s t* ∧ ¬ *drlex-fun t s*

lemma *drlex-fun-refl*:
shows *drlex-fun s s*
⟨*proof*⟩

lemma *drlex-fun-antisym*:
assumes *drlex-fun s t* **and** *drlex-fun t s*
shows $s = t$
⟨*proof*⟩

lemma *drlex-fun-trans*:
assumes *drlex-fun s t* **and** *drlex-fun t u*
shows *drlex-fun s u*
⟨*proof*⟩

lemma *drlex-fun-lin*: *drlex-fun s t* ∨ *drlex-fun t s*
for *s t*::('*a* ⇒ '*b*::{*ordered-comm-monoid-add*, *linorder*})
⟨*proof*⟩

corollary *drlex-fun-strict-alt* [*code*]:
drlex-fun-strict s t = (¬ *drlex-fun t s*) **for** *s t*::'*a* ⇒ '*b*::{*ordered-comm-monoid-add*, *linorder*}
⟨*proof*⟩

lemma *drlex-fun-zero-min*:
fixes *s t*::('*a* ⇒ '*b*::*add-linorder-min*)
assumes *finite* (*supp-fun s*)
shows *drlex-fun 0 s*
⟨*proof*⟩

lemma *drlex-fun-plus-monotone*:
fixes *s t u*::'*a* ⇒ '*b*::{*ordered-cancel-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*}
assumes *finite* (*supp-fun s*) **and** *finite* (*supp-fun t*) **and** *finite* (*supp-fun u*) **and** *drlex-fun s t*
shows *drlex-fun* (*s* + *u*) (*t* + *u*)
⟨*proof*⟩

end

Every finite linear ordering is also a well-ordering. This fact is particularly useful when working with fixed finite sets of indeterminates.

class *finite-linorder* = *finite* + *linorder*
begin

subclass *wellorder*
⟨*proof*⟩

end

6.8 Type *poly-mapping*

lemma *poly-mapping-eq-zeroI*:
assumes *keys s* = {}
shows *s* = (*0*::('*a*, '*b*::*zero*) *poly-mapping*)
⟨*proof*⟩

lemma *keys-plus-ninv-comm-monoid-add*: *keys* (*s* + *t*) = *keys s* ∪ *keys* (*t*::'*a* ⇒$_0$ '*b*::*ninv-comm-monoid-add*)
⟨*proof*⟩

lemma *lookup-zero-fun*: *lookup 0* = *0*
⟨*proof*⟩

lemma *lookup-plus-fun*: *lookup* (*s* + *t*) = *lookup s* + *lookup t*
⟨*proof*⟩

lemma *lookup-uminus-fun*: *lookup* (− *s*) = − *lookup s*
⟨*proof*⟩

lemma *lookup-minus-fun*: *lookup* (*s* − *t*) = *lookup s* − *lookup t*
⟨*proof*⟩

lemma *poly-mapping-adds-iff*: *s adds t* ⟷ *lookup s adds lookup t*
⟨*proof*⟩

6.8.1 '*a* ⇒$_0$ '*b* belongs to class *comm-powerprod*

instance *poly-mapping* :: (*type*, *cancel-comm-monoid-add*) *comm-powerprod*
⟨*proof*⟩

6.8.2 '*a* ⇒$_0$ '*b* belongs to class *ninv-comm-monoid-add*

instance *poly-mapping* :: (*type*, *ninv-comm-monoid-add*) *ninv-comm-monoid-add*
⟨*proof*⟩

6.8.3 '*a* ⇒$_0$ '*b* belongs to class *lcs-powerprod*

instantiation *poly-mapping* :: (*type*, *add-linorder*) *lcs-powerprod*
begin

lift-definition *lcs-poly-mapping*::('*a* ⇒$_0$ '*b*) ⇒ ('*a* ⇒$_0$ '*b*) ⇒ ('*a* ⇒$_0$ '*b*) **is** λ*s t*. λ*x*. *max* (*s x*) (*t x*)
⟨*proof*⟩

lemma *adds-poly-mappingI*:
assumes *lookup s* ≤ *lookup* (*t*::'*a* ⇒$_0$ '*b*)
shows *s adds t*
⟨*proof*⟩

lemma *lookup-lcs-fun*: *lookup* (*lcs s t*) = *lcs* (*lookup s*) (*lookup* (*t*:: *'a* ⇒$_0$ *'b*))
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *adds-poly-mapping*: *s adds t* ⟷ *lookup s* ≤ *lookup t*
for *s t*::*'a* ⇒$_0$ *'b*::*add-linorder-min*
⟨*proof*⟩

lemma *lookup-gcs-fun*: *lookup* (*gcs s* (*t*::*'a* ⇒$_0$ (*'b*::*add-linorder*))) = *gcs* (*lookup s*) (*lookup t*)
⟨*proof*⟩

6.8.4 *'a* ⇒$_0$ *'b* belongs to class *ulcs-powerprod*

instance *poly-mapping* :: (*type*, *add-linorder-min*) *ulcs-powerprod* ⟨*proof*⟩

6.8.5 Power-products in a given set of indeterminates.

lemma *adds-except*:
s adds t = (*except s V adds except t V* ∧ *except s* (− *V*) *adds except t* (− *V*))
for *s t* :: *'a* ⇒$_0$ *'b*::*add-linorder*
⟨*proof*⟩

lemma *adds-except-singleton*:
s adds t ⟷ (*except s* {*v*} *adds except t* {*v*} ∧ *lookup s v adds lookup t v*)
for *s t* :: *'a* ⇒$_0$ *'b*::*add-linorder*
⟨*proof*⟩

6.8.6 Dickson's lemma for power-products in finitely many indeterminates

context *countable*
begin

definition *elem-index* :: *'a* ⇒ *nat* **where** *elem-index* = (*SOME f*. *inj f*)

lemma *inj-elem-index*: *inj elem-index*
⟨*proof*⟩

lemma *elem-index-inj*:
assumes *elem-index x* = *elem-index y*
shows *x* = *y*
⟨*proof*⟩

lemma *finite-nat-seg*: *finite* {*x*. *elem-index x* < *n*}
⟨*proof*⟩

end

lemma *Dickson-poly-mapping*:
assumes *finite V*
shows *almost-full-on* (*adds*) {*x*::'*a* ⇒$_0$ '*b*::*add-wellorder*. *keys x* ⊆ *V*}
⟨*proof*⟩

definition *varnum* :: '*x set* ⇒ ('*x*::*countable* ⇒$_0$ '*b*::*zero*) ⇒ *nat*
where *varnum X t* = (*if keys t* − *X* = {} *then 0 else Suc* (*Max* (*elem-index* ‘ (*keys t* − *X*))))

lemma *elem-index-less-varnum*:
assumes *x* ∈ *keys t*
obtains *x* ∈ *X* | *elem-index x* < *varnum X t*
⟨*proof*⟩

lemma *varnum-plus*:
varnum X (*s* + *t*) = *max* (*varnum X s*) (*varnum X* (*t*::'*x*::*countable* ⇒$_0$ '*b*::*ninv-comm-monoid-add*))
⟨*proof*⟩

lemma *dickson-grading-varnum*:
assumes *finite X*
shows *dickson-grading* ((*varnum X*)::('*x*::*countable* ⇒$_0$ '*b*::*add-wellorder*) ⇒ *nat*)
⟨*proof*⟩

corollary *dickson-grading-varnum-empty*:
dickson-grading ((*varnum* {})::(- ⇒$_0$ -::*add-wellorder*) ⇒ *nat*)
⟨*proof*⟩

lemma *varnum-le-iff*: *varnum X t* ≤ *n* ⟷ *keys t* ⊆ *X* ∪ {*x*. *elem-index x* < *n*}
⟨*proof*⟩

lemma *varnum-zero* [*simp*]: *varnum X 0* = *0*
⟨*proof*⟩

lemma *varnum-empty-eq-zero-iff*: *varnum* {} *t* = *0* ⟷ *t* = *0*
⟨*proof*⟩

instance *poly-mapping* :: (*countable*, *add-wellorder*) *graded-dickson-powerprod*
⟨*proof*⟩

instance *poly-mapping* :: (*finite*, *add-wellorder*) *dickson-powerprod*
⟨*proof*⟩

6.8.7 Lexicographic Term Order

definition *lex-pm* :: ('*a* ⇒$_0$ '*b*) ⇒ ('*a*::*linorder* ⇒$_0$ '*b*::{*zero*,*linorder*}) ⇒ *bool*
where *lex-pm* = (≤)

definition *lex-pm-strict* :: ('*a* ⇒$_0$ '*b*) ⇒ ('*a*::*linorder* ⇒$_0$ '*b*::{*zero*,*linorder*}) ⇒ *bool*
  where *lex-pm-strict* = (<)

lemma *lex-pm-alt*: *lex-pm s t* = (*s* = *t* ∨ (∃ *x*. *lookup s x* < *lookup t x* ∧ (∀ *y*<*x*. *lookup s y* = *lookup t y*)))
  ⟨*proof*⟩

lemma *lex-pm-refl*: *lex-pm s s*
  ⟨*proof*⟩

lemma *lex-pm-antisym*: *lex-pm s t* ⟹ *lex-pm t s* ⟹ *s* = *t*
  ⟨*proof*⟩

lemma *lex-pm-trans*: *lex-pm s t* ⟹ *lex-pm t u* ⟹ *lex-pm s u*
  ⟨*proof*⟩

lemma *lex-pm-lin*: *lex-pm s t* ∨ *lex-pm t s*
  ⟨*proof*⟩

corollary *lex-pm-strict-alt* [*code*]: *lex-pm-strict s t* = (¬ *lex-pm t s*)
  ⟨*proof*⟩

lemma *lex-pm-zero-min*: *lex-pm 0 s* **for** *s*::- ⇒$_0$ -::*add-linorder-min*
⟨*proof*⟩

lemma *lex-pm-plus-monotone*: *lex-pm s t* ⟹ *lex-pm* (*s* + *u*) (*t* + *u*)
  for *s t*::- ⇒$_0$ -::{*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*}
  ⟨*proof*⟩

6.8.8 Degree

lift-definition *deg-pm*::('*a* ⇒$_0$ '*b*::*comm-monoid-add*) ⇒ '*b* **is** *deg-fun* ⟨*proof*⟩

lemma *deg-pm-zero*[*simp*]: *deg-pm 0* = *0*
  ⟨*proof*⟩

lemma *deg-pm-eq-0-iff*[*simp*]: *deg-pm s* = *0* ⟷ *s* = *0* **for** *s*::'*a* ⇒$_0$ '*b*::*add-linorder-min*
  ⟨*proof*⟩

lemma *deg-pm-superset*:
  assumes *keys s* ⊆ *A* **and** *finite A*
  shows *deg-pm s* = (∑ *x*∈*A*. *lookup s x*)
  ⟨*proof*⟩

lemma *deg-pm-plus*: *deg-pm* (*s* + *t*) = *deg-pm s* + *deg-pm* (*t*::'*a* ⇒$_0$ '*b*::*comm-monoid-add*)
  ⟨*proof*⟩

lemma *deg-pm-single*: *deg-pm* (*Poly-Mapping.single x k*) = *k*
⟨*proof*⟩

6.8.9 General Degree-Orders

context *linorder*
begin

lift-definition *dord-pm*::((*'a* ⇒$_0$ *'b*::*ordered-comm-monoid-add*) ⇒ (*'a* ⇒$_0$ *'b*) ⇒ *bool*) ⇒ (*'a* ⇒$_0$ *'b*) ⇒ (*'a* ⇒$_0$ *'b*) ⇒ *bool*
is *dord-fun* ⟨*proof*⟩

lemma *dord-pm-alt*: *dord-pm ord* = (λ*x y*. *deg-pm x* < *deg-pm y* ∨ (*deg-pm x* = *deg-pm y* ∧ *ord x y*))
⟨*proof*⟩

lemma *dord-pm-degD*:
assumes *dord-pm ord s t*
shows *deg-pm s* ≤ *deg-pm t*
⟨*proof*⟩

lemma *dord-pm-refl*:
assumes *ord s s*
shows *dord-pm ord s s*
⟨*proof*⟩

lemma *dord-pm-antisym*:
assumes *ord s t* ⟹ *ord t s* ⟹ *s* = *t* **and** *dord-pm ord s t* **and** *dord-pm ord t s*
shows *s* = *t*
⟨*proof*⟩

lemma *dord-pm-trans*:
assumes *ord s t* ⟹ *ord t u* ⟹ *ord s u* **and** *dord-pm ord s t* **and** *dord-pm ord t u*
shows *dord-pm ord s u*
⟨*proof*⟩

lemma *dord-pm-lin*:
dord-pm ord s t ∨ *dord-pm ord t s*
if *ord s t* ∨ *ord t s*
for *s t*::*'a* ⇒$_0$ *'b*::{*ordered-comm-monoid-add*, *linorder*}
⟨*proof*⟩

lemma *dord-pm-zero-min*: *dord-pm ord 0 s*
if *ord-refl*: ⋀*t*. *ord t t*
for *s t*::*'a* ⇒$_0$ *'b*::*add-linorder-min*
⟨*proof*⟩

lemma *dord-pm-plus-monotone*:
fixes *s t u* ::*'a* ⇒$_0$ *'b*::{*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*}
assumes *ord s t* ⟹ *ord* (*s* + *u*) (*t* + *u*) **and** *dord-pm ord s t*
shows *dord-pm ord* (*s* + *u*) (*t* + *u*)
⟨*proof*⟩

end

6.8.10 Degree-Lexicographic Term Order

definition *dlex-pm*::(*'a*::*linorder* ⇒$_0$ *'b*::{*ordered-comm-monoid-add*,*linorder*}) ⇒ (*'a* ⇒$_0$ *'b*) ⇒ *bool*
where *dlex-pm* ≡ *dord-pm lex-pm*

definition *dlex-pm-strict s t* ⟷ *dlex-pm s t* ∧ ¬ *dlex-pm t s*

lemma *dlex-pm-refl*: *dlex-pm s s*
⟨*proof*⟩

lemma *dlex-pm-antisym*: *dlex-pm s t* ⟹ *dlex-pm t s* ⟹ *s* = *t*
⟨*proof*⟩

lemma *dlex-pm-trans*: *dlex-pm s t* ⟹ *dlex-pm t u* ⟹ *dlex-pm s u*
⟨*proof*⟩

lemma *dlex-pm-lin*: *dlex-pm s t* ∨ *dlex-pm t s*
⟨*proof*⟩

corollary *dlex-pm-strict-alt* [*code*]: *dlex-pm-strict s t* = (¬ *dlex-pm t s*)
⟨*proof*⟩

lemma *dlex-pm-zero-min*: *dlex-pm 0 s*
for *s t*::(- ⇒$_0$ -::*add-linorder-min*)
⟨*proof*⟩

lemma *dlex-pm-plus-monotone*: *dlex-pm s t* ⟹ *dlex-pm* (*s* + *u*) (*t* + *u*)
for *s t*::- ⇒$_0$ -::{*ordered-ab-semigroup-add-imp-le*, *ordered-cancel-comm-monoid-add*}
⟨*proof*⟩

6.8.11 Degree-Reverse-Lexicographic Term Order

definition *drlex-pm*::(*'a*::*linorder* ⇒$_0$ *'b*::{*ordered-comm-monoid-add*,*linorder*}) ⇒ (*'a* ⇒$_0$ *'b*) ⇒ *bool*
where *drlex-pm* ≡ *dord-pm* (λ*s t*. *lex-pm t s*)

definition *drlex-pm-strict s t* ⟷ *drlex-pm s t* ∧ ¬ *drlex-pm t s*

lemma *drlex-pm-refl*: *drlex-pm s s*
⟨*proof*⟩

lemma *drlex-pm-antisym*: *drlex-pm s t* $\Longrightarrow$ *drlex-pm t s* $\Longrightarrow$ $s = t$
⟨*proof*⟩

lemma *drlex-pm-trans*: *drlex-pm s t* $\Longrightarrow$ *drlex-pm t u* $\Longrightarrow$ *drlex-pm s u*
⟨*proof*⟩

lemma *drlex-pm-lin*: *drlex-pm s t* $\vee$ *drlex-pm t s*
⟨*proof*⟩

corollary *drlex-pm-strict-alt* [*code*]: *drlex-pm-strict s t* = ($\neg$ *drlex-pm t s*)
⟨*proof*⟩

lemma *drlex-pm-zero-min*: *drlex-pm 0 s*
for *s t*::(- $\Rightarrow_0$ -::*add-linorder-min*)
⟨*proof*⟩

lemma *drlex-pm-plus-monotone*: *drlex-pm s t* $\Longrightarrow$ *drlex-pm* (s + u) (t + u)
for *s t*::- $\Rightarrow_0$ -::{*ordered-ab-semigroup-add-imp-le*, *ordered-cancel-comm-monoid-add*}
⟨*proof*⟩

end

theory *More-Modules*
imports *HOL.Modules*
begin

More facts about modules.

7 Modules over Commutative Rings

context *module*
begin

lemma *scale-minus-both* [*simp*]: ($-$ a) $*s$ ($-$ x) = a $*s$ x
⟨*proof*⟩

7.1 Submodules Spanned by Sets of Module-Elements

lemma *span-insertI*:
assumes $p \in$ *span B*
shows $p \in$ *span* (*insert r B*)
⟨*proof*⟩

lemma *span-insertD*:
assumes $p \in$ *span* (*insert r B*) **and** $r \in$ *span B*
shows $p \in$ *span B*
⟨*proof*⟩

lemma *span-insert-idI*:
assumes $r \in$ *span* B
shows *span* (*insert* r B) = *span* B
⟨*proof*⟩

lemma *span-insert-zero*: *span* (*insert* 0 B) = *span* B
⟨*proof*⟩

lemma *span-Diff-zero*: *span* ($B - \{0\}$) = *span* B
⟨*proof*⟩

lemma *span-insert-subset*:
assumes *span* $A \subseteq$ *span* B **and** $r \in$ *span* B
shows *span* (*insert* r A) $\subseteq$ *span* B
⟨*proof*⟩

lemma *replace-span*:
assumes $q \in$ *span* B
shows *span* (*insert* q ($B - \{p\}$)) $\subseteq$ *span* B
⟨*proof*⟩

lemma *sum-in-spanI*: $(\sum b{\in}B.\ q\ b\ *s\ b) \in$ *span* B
⟨*proof*⟩

lemma *span-closed-sum-list*: $(\bigwedge x.\ x \in$ *set* $xs \Longrightarrow x \in$ *span* $B) \Longrightarrow$ *sum-list* $xs \in$ *span* B
⟨*proof*⟩

lemma *spanE*:
assumes $p \in$ *span* B
obtains A q **where** *finite* A **and** $A \subseteq B$ **and** $p = (\sum b{\in}A.\ (q\ b)\ *s\ b)$
⟨*proof*⟩

lemma *span-finite-subset*:
assumes $p \in$ *span* B
obtains A **where** *finite* A **and** $A \subseteq B$ **and** $p \in$ *span* A
⟨*proof*⟩

lemma *span-finiteE*:
assumes *finite* B **and** $p \in$ *span* B
obtains q **where** $p = (\sum b{\in}B.\ (q\ b)\ *s\ b)$
⟨*proof*⟩

lemma *span-subset-spanI*:
assumes $A \subseteq$ *span* B
shows *span* $A \subseteq$ *span* B
⟨*proof*⟩

lemma *span-insert-cong*:

assumes *span A = span B*
shows *span (insert p A) = span (insert p B)* (**is** *?l = ?r*)
⟨*proof*⟩

lemma *span-induct'* [*consumes 1*, *case-names base step*]:
assumes $p \in$ *span B* **and** *P 0*
and $\bigwedge a\ q\ p.\ a \in$ *span B* $\Longrightarrow$ *P a* $\Longrightarrow$ $p \in B \Longrightarrow q \neq 0 \Longrightarrow$ *P* $(a + q *s p)$
shows *P p*
⟨*proof*⟩

lemma *span-INT-subset*: *span* $(\bigcap a{\in}A.\ f\ a) \subseteq (\bigcap a{\in}A.$ *span* $(f\ a))$ (**is** *?l* $\subseteq$ *?r*)
⟨*proof*⟩

lemma *span-INT*: *span* $(\bigcap a{\in}A.$ *span* $(f\ a)) = (\bigcap a{\in}A.$ *span* $(f\ a))$ (**is** *?l = ?r*)
⟨*proof*⟩

lemma *span-Int-subset*: *span* $(A \cap B) \subseteq$ *span A* $\cap$ *span B*
⟨*proof*⟩

lemma *span-Int*: *span* (*span A* $\cap$ *span B*) = *span A* $\cap$ *span B*
⟨*proof*⟩

lemma *span-image-scale-eq-image-scale*: *span* $((*s)\ q\ `\ F) = (*s)\ q\ `$ *span F* (**is** *?A = ?B*)
⟨*proof*⟩

end

8 Ideals over Commutative Rings

lemma *module-times*: *module* $(*)$
⟨*proof*⟩

interpretation *ideal*: *module times*
⟨*proof*⟩

declare *ideal.scale-scale*[*simp del*]

abbreviation *ideal* $\equiv$ *ideal.span*

lemma *ideal-eq-UNIV-iff-contains-one*: *ideal B = UNIV* $\longleftrightarrow$ *1* $\in$ *ideal B*
⟨*proof*⟩

lemma *ideal-eq-zero-iff* [*iff*]: *ideal F* $= \{0\} \longleftrightarrow F \subseteq \{0\}$
⟨*proof*⟩

lemma *ideal-field-cases*:
obtains *ideal B* $= \{0\}$ | *ideal* $(B{::}'a{::}$*field set*$) =$ *UNIV*
⟨*proof*⟩

corollary *ideal-field-disj*: *ideal* $B = \{0\} \vee$ *ideal* $(B::'a::field\ set) = UNIV$
⟨*proof*⟩

lemma *image-ideal-subset*:
assumes $\bigwedge x\ y.\ h\ (x + y) = h\ x + h\ y$ **and** $\bigwedge x\ y.\ h\ (x * y) = h\ x * h\ y$
shows $h\ `\ ideal\ F \subseteq ideal\ (h\ `\ F)$
⟨*proof*⟩

lemma *image-ideal-eq-surj*:
assumes $\bigwedge x\ y.\ h\ (x + y) = h\ x + h\ y$ **and** $\bigwedge x\ y.\ h\ (x * y) = h\ x * h\ y$ **and** *surj h*
shows $h\ `\ ideal\ B = ideal\ (h\ `\ B)$
⟨*proof*⟩

context
fixes $h :: 'a \Rightarrow 'a::comm\text{-}ring\text{-}1$
assumes *h-plus*: $h\ (x + y) = h\ x + h\ y$
assumes *h-times*: $h\ (x * y) = h\ x * h\ y$
assumes *h-idem*: $h\ (h\ x) = h\ x$
begin

lemma *in-idealE-homomorphism-finite*:
assumes *finite B* **and** $B \subseteq range\ h$ **and** $p \in range\ h$ **and** $p \in ideal\ B$
obtains q **where** $\bigwedge b.\ q\ b \in range\ h$ **and** $p = (\sum b{\in}B.\ q\ b * b)$
⟨*proof*⟩

corollary *in-idealE-homomorphism*:
assumes $B \subseteq range\ h$ **and** $p \in range\ h$ **and** $p \in ideal\ B$
obtains $A\ q$ **where** *finite A* **and** $A \subseteq B$ **and** $\bigwedge b.\ q\ b \in range\ h$ **and** $p = (\sum b{\in}A.\ q\ b * b)$
⟨*proof*⟩

lemma *ideal-induct-homomorphism* [*consumes 3*, *case-names 0 plus*]:
assumes $B \subseteq range\ h$ **and** $p \in range\ h$ **and** $p \in ideal\ B$
assumes $P\ 0$ **and** $\bigwedge c\ b\ a.\ c \in range\ h \Longrightarrow b \in B \Longrightarrow P\ a \Longrightarrow a \in range\ h \Longrightarrow P\ (c * b + a)$
shows $P\ p$
⟨*proof*⟩

lemma *image-ideal-eq-Int*: $h\ `\ ideal\ B = ideal\ (h\ `\ B) \cap range\ h$
⟨*proof*⟩

end

end

9 Type-Class-Multivariate Polynomials

theory *MPoly-Type-Class*
imports
Utils
Power-Products
More-Modules
begin

This theory views $'a \Rightarrow_0 'b$ as multivariate polynomials, where type class constraints on $'a$ ensure that $'a$ represents something like monomials.

lemma *when-distrib*: $f\ (a\ when\ b) = (f\ a\ when\ b)$ **if** $\neg\ b \Longrightarrow f\ 0 = 0$
⟨*proof*⟩

definition *mapp-2* :: $('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd) \Rightarrow ('a \Rightarrow_0 'b{::}zero) \Rightarrow ('a \Rightarrow_0 'c{::}zero) \Rightarrow ('a \Rightarrow_0 'd{::}zero)$
where *mapp-2 f p q* = *Abs-poly-mapping* $(\lambda k.\ f\ k\ (lookup\ p\ k)\ (lookup\ q\ k)\ when\ k \in keys\ p \cup keys\ q)$

lemma *lookup-mapp-2*:
lookup (*mapp-2 f p q*) *k* = (*f k* (*lookup p k*) (*lookup q k*) *when k* ∈ *keys p* ∪ *keys q*)
⟨*proof*⟩

lemma *lookup-mapp-2-homogenous*:
assumes *f k 0 0* = *0*
shows *lookup* (*mapp-2 f p q*) *k* = *f k* (*lookup p k*) (*lookup q k*)
⟨*proof*⟩

lemma *mapp-2-cong* [*fundef-cong*]:
assumes *p* = *p'* **and** *q* = *q'*
assumes $\bigwedge k.\ k \in keys\ p' \cup keys\ q' \Longrightarrow f\ k\ (lookup\ p'\ k)\ (lookup\ q'\ k) = f'\ k\ (lookup\ p'\ k)\ (lookup\ q'\ k)$
shows *mapp-2 f p q* = *mapp-2 f' p' q'*
⟨*proof*⟩

lemma *keys-mapp-subset*: *keys* (*mapp-2 f p q*) ⊆ *keys p* ∪ *keys q*
⟨*proof*⟩

lemma *mapp-2-mapp*: *mapp-2* $(\lambda t\ a.\ f\ t)$ *0 p* = *Poly-Mapping.mapp f p*
⟨*proof*⟩

9.1 *keys*

lemma *in-keys-plusI1*:
assumes *t* ∈ *keys p* **and** *t* ∉ *keys q*
shows *t* ∈ *keys* (*p* + *q*)
⟨*proof*⟩

lemma *in-keys-plusI2*:

assumes $t \in keys\ q$ **and** $t \notin keys\ p$
shows $t \in keys\ (p + q)$
⟨*proof*⟩

lemma *keys-plus-eqI*:
assumes $keys\ p \cap keys\ q = \{\}$
shows $keys\ (p + q) = (keys\ p \cup keys\ q)$
⟨*proof*⟩

lemma *keys-uminus*: $keys\ (-\ p) = keys\ p$
⟨*proof*⟩

lemma *keys-minus*: $keys\ (p - q) \subseteq (keys\ p \cup keys\ q)$
⟨*proof*⟩

9.2 Monomials

abbreviation *monomial* $\equiv (\lambda c\ t.\ Poly\text{-}Mapping.single\ t\ c)$

lemma *keys-of-monomial*:
assumes $c \neq 0$
shows $keys\ (monomial\ c\ t) = \{t\}$
⟨*proof*⟩

lemma *monomial-uminus*:
shows $-\ monomial\ c\ s = monomial\ (-\ c)\ s$
⟨*proof*⟩

lemma *monomial-inj*:
assumes $monomial\ c\ s = monomial\ (d::'b::zero\text{-}neq\text{-}one)\ t$
shows $(c = 0 \wedge d = 0) \vee (c = d \wedge s = t)$
⟨*proof*⟩

definition *is-monomial* :: $('a \Rightarrow_0 'b::zero) \Rightarrow bool$
where $is\text{-}monomial\ p \longleftrightarrow card\ (keys\ p) = 1$

lemma *monomial-is-monomial*:
assumes $c \neq 0$
shows $is\text{-}monomial\ (monomial\ c\ t)$
⟨*proof*⟩

lemma *is-monomial-monomial*:
assumes *is-monomial p*
obtains $c\ t$ **where** $c \neq 0$ **and** $p = monomial\ c\ t$
⟨*proof*⟩

lemma *is-monomial-uminus*: $is\text{-}monomial\ (-p) \longleftrightarrow is\text{-}monomial\ p$
⟨*proof*⟩

lemma *monomial-not-0*:
assumes *is-monomial p*
shows $p \neq 0$
⟨*proof*⟩

lemma *keys-subset-singleton-imp-monomial*:
assumes *keys p* $\subseteq \{t\}$
shows *monomial* (*lookup p t*) *t* = *p*
⟨*proof*⟩

lemma *monomial-0I*:
assumes *c* = *0*
shows *monomial c t* = *0*
⟨*proof*⟩

lemma *monomial-0D*:
assumes *monomial c t* = *0*
shows *c* = *0*
⟨*proof*⟩

corollary *monomial-0-iff*: *monomial c t* = *0* ⟷ *c* = *0*
⟨*proof*⟩

lemma *lookup-times-monomial-left*: *lookup* (*monomial c t* ∗ *p*) *s* = (*c* ∗ *lookup p* (*s* − *t*) *when t adds s*)
for *c*::′*b*::*semiring-0* **and** *t*::′*a*::*comm-powerprod*
⟨*proof*⟩

lemma *lookup-times-monomial-right*: *lookup* (*p* ∗ *monomial c t*) *s* = (*lookup p* (*s* − *t*) ∗ *c when t adds s*)
for *c*::′*b*::*semiring-0* **and** *t*::′*a*::*comm-powerprod*
⟨*proof*⟩

9.3 Vector-Polynomials

From now on we consider multivariate vector-polynomials, i. e. vectors of scalar polynomials. We do this by adding a *component* to each power-product, yielding *terms*. Vector-polynomials are then again just linear combinations of terms. Note that a term is *not* the same as a vector of power-products!

We use define terms in a locale, such that later on we can interpret the locale also by ordinary power-products (without components), exploiting the canonical isomorphism between ′*a* and ′*a* × *unit*.

named-theorems *term-simps simplification rules for terms*

locale *term-powerprod* =
fixes *pair-of-term*::′*t* ⇒ (′*a*::*comm-powerprod* × ′*k*::*linorder*)
fixes *term-of-pair*::(′*a* × ′*k*) ⇒ ′*t*

assumes *term-pair* [*term-simps*]: *term-of-pair* (*pair-of-term* v) = v
assumes *pair-term* [*term-simps*]: *pair-of-term* (*term-of-pair* p) = p
begin

lemma *pair-of-term-injective*:
assumes *pair-of-term* u = *pair-of-term* v
shows $u = v$
⟨*proof*⟩

corollary *pair-of-term-inj*: *inj pair-of-term*
⟨*proof*⟩

lemma *term-of-pair-injective*:
assumes *term-of-pair* p = *term-of-pair* q
shows $p = q$
⟨*proof*⟩

corollary *term-of-pair-inj*: *inj term-of-pair*
⟨*proof*⟩

definition *pp-of-term* :: $'t \Rightarrow 'a$
where *pp-of-term* v = *fst* (*pair-of-term* v)

definition *component-of-term* :: $'t \Rightarrow 'k$
where *component-of-term* v = *snd* (*pair-of-term* v)

lemma *term-of-pair-pair* [*term-simps*]: *term-of-pair* (*pp-of-term* v, *component-of-term* v) = v
⟨*proof*⟩

lemma *pp-of-term-of-pair* [*term-simps*]: *pp-of-term* (*term-of-pair* (t, k)) = t
⟨*proof*⟩

lemma *component-of-term-of-pair* [*term-simps*]: *component-of-term* (*term-of-pair* (t, k)) = k
⟨*proof*⟩

9.3.1 Additive Structure of Terms

definition *splus* :: $'a \Rightarrow 't \Rightarrow 't$ (**infixl** $\oplus$ 75)
where *splus* t v = *term-of-pair* (t + *pp-of-term* v, *component-of-term* v)

definition *sminus* :: $'t \Rightarrow 'a \Rightarrow 't$ (**infixl** $\ominus$ 75)
where *sminus* v t = *term-of-pair* (*pp-of-term* v − t, *component-of-term* v)

Note that the argument order in ($\ominus$) is reversed compared to the order in ($\oplus$).

definition *adds-pp* :: $'a \Rightarrow 't \Rightarrow$ *bool* (**infix** $adds_p$ 50)
where *adds-pp* t $v \longleftrightarrow$ t *adds pp-of-term* v

definition *adds-term* :: $'t \Rightarrow 't \Rightarrow$ *bool* (**infix** $adds_t$ *50*)
where *adds-term u v* $\longleftrightarrow$ *component-of-term u* = *component-of-term v* $\wedge$ *pp-of-term u adds pp-of-term v*

lemma *pp-of-term-splus* [*term-simps*]: *pp-of-term* $(t \oplus v)$ = t + *pp-of-term v*
⟨*proof*⟩

lemma *component-of-term-splus* [*term-simps*]: *component-of-term* $(t \oplus v)$ = *component-of-term v*
⟨*proof*⟩

lemma *pp-of-term-sminus* [*term-simps*]: *pp-of-term* $(v \ominus t)$ = *pp-of-term v* − t
⟨*proof*⟩

lemma *component-of-term-sminus* [*term-simps*]: *component-of-term* $(v \ominus t)$ = *component-of-term v*
⟨*proof*⟩

lemma *splus-sminus* [*term-simps*]: $(t \oplus v) \ominus t = v$
⟨*proof*⟩

lemma *splus-zero* [*term-simps*]: *0* $\oplus$ $v = v$
⟨*proof*⟩

lemma *sminus-zero* [*term-simps*]: $v \ominus$ *0* = v
⟨*proof*⟩

lemma *splus-assoc* [*ac-simps*]: $(s + t) \oplus v = s \oplus (t \oplus v)$
⟨*proof*⟩

lemma *splus-left-commute* [*ac-simps*]: $s \oplus (t \oplus v) = t \oplus (s \oplus v)$
⟨*proof*⟩

lemma *splus-right-canc* [*term-simps*]: $t \oplus v = s \oplus v \longleftrightarrow t = s$
⟨*proof*⟩

lemma *splus-left-canc* [*term-simps*]: $t \oplus v = t \oplus u \longleftrightarrow v = u$
⟨*proof*⟩

lemma *adds-ppI* [*intro?*]:
assumes $v = t \oplus u$
shows t $adds_p$ v
⟨*proof*⟩

lemma *adds-ppE* [*elim?*]:
assumes t $adds_p$ v
obtains u **where** $v = t \oplus u$
⟨*proof*⟩

lemma *adds-pp-alt*: $t\ adds_p\ v \longleftrightarrow (\exists\, u.\ v = t \oplus u)$
⟨*proof*⟩

lemma *adds-pp-refl* [*term-simps*]: $(pp\text{-}of\text{-}term\ v)\ adds_p\ v$
⟨*proof*⟩

lemma *adds-pp-trans* [*trans*]:
assumes $s\ adds\ t$ **and** $t\ adds_p\ v$
shows $s\ adds_p\ v$
⟨*proof*⟩

lemma *zero-adds-pp* [*term-simps*]: $0\ adds_p\ v$
⟨*proof*⟩

lemma *adds-pp-splus*:
assumes $t\ adds_p\ v$
shows $t\ adds_p\ s \oplus v$
⟨*proof*⟩

lemma *adds-pp-triv* [*term-simps*]: $t\ adds_p\ t \oplus v$
⟨*proof*⟩

lemma *plus-adds-pp-mono*:
assumes $s\ adds\ t$
and $u\ adds_p\ v$
shows $s + u\ adds_p\ t \oplus v$
⟨*proof*⟩

lemma *plus-adds-pp-left*:
assumes $s + t\ adds_p\ v$
shows $s\ adds_p\ v$
⟨*proof*⟩

lemma *plus-adds-pp-right*:
assumes $s + t\ adds_p\ v$
shows $t\ adds_p\ v$
⟨*proof*⟩

lemma *adds-pp-sminus*:
assumes $t\ adds_p\ v$
shows $t \oplus (v \ominus t) = v$
⟨*proof*⟩

lemma *adds-pp-canc*: $t + s\ adds_p\ (t \oplus v) \longleftrightarrow s\ adds_p\ v$
⟨*proof*⟩

lemma *adds-pp-canc-2*: $s + t\ adds_p\ (t \oplus v) \longleftrightarrow s\ adds_p\ v$
⟨*proof*⟩

lemma *plus-adds-pp-0*:
assumes $(s + t)\ adds_p\ v$
shows $s\ adds_p\ (v \ominus t)$
⟨*proof*⟩

lemma *plus-adds-ppI-1*:
assumes $t\ adds_p\ v$ **and** $s\ adds_p\ (v \ominus t)$
shows $(s + t)\ adds_p\ v$
⟨*proof*⟩

lemma *plus-adds-ppI-2*:
assumes $t\ adds_p\ v$ **and** $s\ adds_p\ (v \ominus t)$
shows $(t + s)\ adds_p\ v$
⟨*proof*⟩

lemma *plus-adds-pp*: $(s + t)\ adds_p\ v \longleftrightarrow (t\ adds_p\ v \wedge s\ adds_p\ (v \ominus t))$
⟨*proof*⟩

lemma *minus-splus*:
assumes $s\ adds\ t$
shows $(t - s) \oplus v = (t \oplus v) \ominus s$
⟨*proof*⟩

lemma *minus-splus-sminus*:
assumes $s\ adds\ t$ **and** $u\ adds_p\ v$
shows $(t - s) \oplus (v \ominus u) = (t \oplus v) \ominus (s + u)$
⟨*proof*⟩

lemma *minus-splus-sminus-cancel*:
assumes $s\ adds\ t$ **and** $t\ adds_p\ v$
shows $(t - s) \oplus (v \ominus t) = v \ominus s$
⟨*proof*⟩

lemma *sminus-plus*:
assumes $s\ adds_p\ v$ **and** $t\ adds_p\ (v \ominus s)$
shows $v \ominus (s + t) = (v \ominus s) \ominus t$
⟨*proof*⟩

lemma *adds-termI* [*intro?*]:
assumes $v = t \oplus u$
shows $u\ adds_t\ v$
⟨*proof*⟩

lemma *adds-termE* [*elim?*]:
assumes $u\ adds_t\ v$
obtains t **where** $v = t \oplus u$
⟨*proof*⟩

lemma *adds-term-alt*: $u\ adds_t\ v \longleftrightarrow (\exists\, t.\ v = t \oplus u)$
⟨*proof*⟩

lemma *adds-term-refl* [*term-simps*]: $v\ adds_t\ v$
⟨*proof*⟩

lemma *adds-term-trans* [*trans*]:
assumes $u\ adds_t\ v$ **and** $v\ adds_t\ w$
shows $u\ adds_t\ w$
⟨*proof*⟩

lemma *adds-term-splus*:
assumes $u\ adds_t\ v$
shows $u\ adds_t\ s \oplus v$
⟨*proof*⟩

lemma *adds-term-triv* [*term-simps*]: $v\ adds_t\ t \oplus v$
⟨*proof*⟩

lemma *splus-adds-term-mono*:
assumes $s\ adds\ t$
and $u\ adds_t\ v$
shows $s \oplus u\ adds_t\ t \oplus v$
⟨*proof*⟩

lemma *splus-adds-term*:
assumes $t \oplus u\ adds_t\ v$
shows $u\ adds_t\ v$
⟨*proof*⟩

lemma *adds-term-adds-pp*:
$u\ adds_t\ v \longleftrightarrow (component\text{-}of\text{-}term\ u = component\text{-}of\text{-}term\ v \wedge pp\text{-}of\text{-}term\ u\ adds_p\ v)$
⟨*proof*⟩

lemma *adds-term-canc*: $t \oplus u\ adds_t\ t \oplus v \longleftrightarrow u\ adds_t\ v$
⟨*proof*⟩

lemma *adds-term-canc-2*: $s \oplus v\ adds_t\ t \oplus v \longleftrightarrow s\ adds\ t$
⟨*proof*⟩

lemma *splus-adds-term-0*:
assumes $t \oplus u\ adds_t\ v$
shows $u\ adds_t\ (v \ominus t)$
⟨*proof*⟩

lemma *splus-adds-termI-1*:
assumes $t\ adds_p\ v$ **and** $u\ adds_t\ (v \ominus t)$
shows $t \oplus u\ adds_t\ v$

⟨*proof*⟩

lemma *splus-adds-term-iff*: $t \oplus u$ *adds*$_t$ $v \longleftrightarrow (t$ *adds*$_p$ $v \wedge u$ *adds*$_t$ $(v \ominus t))$
⟨*proof*⟩

lemma *adds-minus-splus*:
assumes *pp-of-term u adds t*
shows $(t -$ *pp-of-term u*$) \oplus u =$ *term-of-pair* $(t,$ *component-of-term u*$)$
⟨*proof*⟩

9.3.2 Projections and Conversions

lift-definition *proj-poly* :: $'k \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'b$::*zero*$)$
is $\lambda k\ p\ t.\ p$ (*term-of-pair* $(t,\ k)$)
⟨*proof*⟩

definition *vectorize-poly* :: $('t \Rightarrow_0 'b) \Rightarrow ('k \Rightarrow_0 ('a \Rightarrow_0 'b$::*zero*$))$
where *vectorize-poly p* = *Abs-poly-mapping* $(\lambda k.$ *proj-poly k p*$)$

definition *atomize-poly* :: $('k \Rightarrow_0 ('a \Rightarrow_0 'b)) \Rightarrow ('t \Rightarrow_0 'b$::*zero*$)$
where *atomize-poly p* = *Abs-poly-mapping* $(\lambda v.$ *lookup* (*lookup p* (*component-of-term v*)) (*pp-of-term v*))

lemma *lookup-proj-poly*: *lookup* (*proj-poly k p*) *t* = *lookup p* (*term-of-pair* $(t,\ k)$)
⟨*proof*⟩

lemma *lookup-vectorize-poly*: *lookup* (*vectorize-poly p*) *k* = *proj-poly k p*
⟨*proof*⟩

lemma *lookup-atomize-poly*:
lookup (*atomize-poly p*) *v* = *lookup* (*lookup p* (*component-of-term v*)) (*pp-of-term v*)
⟨*proof*⟩

lemma *keys-proj-poly*: *keys* (*proj-poly k p*) = *pp-of-term* ‘ $\{x \in$ *keys p*. *component-of-term* $x = k\}$
⟨*proof*⟩

lemma *keys-vectorize-poly*: *keys* (*vectorize-poly p*) = *component-of-term* ‘ *keys p*
⟨*proof*⟩

lemma *keys-atomize-poly*:
keys (*atomize-poly p*) = $(\bigcup k \in$ *keys p*. $(\lambda t.$ *term-of-pair* $(t,\ k))$ ‘ *keys* (*lookup p k*)) (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *proj-atomize-poly* [*term-simps*]: *proj-poly k* (*atomize-poly p*) = *lookup p k*
⟨*proof*⟩

lemma *vectorize-atomize-poly* [*term-simps*]: *vectorize-poly* (*atomize-poly p*) = *p*
⟨*proof*⟩

lemma *atomize-vectorize-poly* [*term-simps*]: *atomize-poly* (*vectorize-poly p*) = *p*
⟨*proof*⟩

lemma *proj-zero* [*term-simps*]: *proj-poly k 0* = *0*
⟨*proof*⟩

lemma *proj-plus*: *proj-poly k* (*p* + *q*) = *proj-poly k p* + *proj-poly k q*
⟨*proof*⟩

lemma *proj-uminus* [*term-simps*]: *proj-poly k* (− *p*) = − *proj-poly k p*
⟨*proof*⟩

lemma *proj-minus*: *proj-poly k* (*p* − *q*) = *proj-poly k p* − *proj-poly k q*
⟨*proof*⟩

lemma *vectorize-zero* [*term-simps*]: *vectorize-poly 0* = *0*
⟨*proof*⟩

lemma *vectorize-plus*: *vectorize-poly* (*p* + *q*) = *vectorize-poly p* + *vectorize-poly q*
⟨*proof*⟩

lemma *vectorize-uminus* [*term-simps*]: *vectorize-poly* (− *p*) = − *vectorize-poly p*
⟨*proof*⟩

lemma *vectorize-minus*: *vectorize-poly* (*p* − *q*) = *vectorize-poly p* − *vectorize-poly q*
⟨*proof*⟩

lemma *atomize-zero* [*term-simps*]: *atomize-poly 0* = *0*
⟨*proof*⟩

lemma *atomize-plus*: *atomize-poly* (*p* + *q*) = *atomize-poly p* + *atomize-poly q*
⟨*proof*⟩

lemma *atomize-uminus* [*term-simps*]: *atomize-poly* (− *p*) = − *atomize-poly p*
⟨*proof*⟩

lemma *atomize-minus*: *atomize-poly* (*p* − *q*) = *atomize-poly p* − *atomize-poly q*
⟨*proof*⟩

lemma *proj-monomial*:
proj-poly k (*monomial c v*) = (*monomial c* (*pp-of-term v*) *when component-of-term v* = *k*)
⟨*proof*⟩

lemma *vectorize-monomial*:

vectorize-poly (*monomial c v*) = *monomial* (*monomial c* (*pp-of-term v*)) (*component-of-term v*)
⟨*proof*⟩

lemma *atomize-monomial-monomial*:
atomize-poly (*monomial* (*monomial c t*) *k*) = *monomial c* (*term-of-pair* (*t*, *k*))
⟨*proof*⟩

lemma *poly-mapping-eqI-proj*:
assumes ⋀*k*. *proj-poly k p* = *proj-poly k q*
shows *p* = *q*
⟨*proof*⟩

9.4 Scalar Multiplication by Monomials

definition *monom-mult* :: *'b*::*semiring-0* ⇒ *'a*::*comm-powerprod* ⇒ (*'t* ⇒$_0$ *'b*) ⇒ (*'t* ⇒$_0$ *'b*)
where *monom-mult c t p* = *Abs-poly-mapping* (λ*v*. *if t adds*$_p$ *v then c* ∗ (*lookup p* (*v* ⊖ *t*)) *else 0*)

lemma *keys-monom-mult-aux*:
{*v*. (*if t adds*$_p$ *v then c* ∗ *lookup p* (*v* ⊖ *t*) *else 0*) ≠ *0*} ⊆ (⊕) *t* ' *keys p* (**is** *?l* ⊆ *?r*)
for *c*::*'b*::*semiring-0*
⟨*proof*⟩

lemma *lookup-monom-mult*:
lookup (*monom-mult c t p*) *v* = (*if t adds*$_p$ *v then c* ∗ *lookup p* (*v* ⊖ *t*) *else 0*)
⟨*proof*⟩

lemma *lookup-monom-mult-plus*:
lookup (*monom-mult c t p*) (*t* ⊕ *v*) = (*c*::*'b*::*semiring-0*) ∗ *lookup p v*
⟨*proof*⟩

lemma *monom-mult-assoc*: *monom-mult c s* (*monom-mult d t p*) = *monom-mult* (*c* ∗ *d*) (*s* + *t*) *p*
⟨*proof*⟩

lemma *monom-mult-uminus-left*: *monom-mult* (− *c*) *t p* = − *monom-mult* (*c*::*'b*::*ring*) *t p*
⟨*proof*⟩

lemma *monom-mult-uminus-right*: *monom-mult c t* (− *p*) = − *monom-mult* (*c*::*'b*::*ring*) *t p*
⟨*proof*⟩

lemma *uminus-monom-mult*: − *p* = *monom-mult* (−*1*::*'b*::*comm-ring-1*) *0 p*
⟨*proof*⟩

lemma *monom-mult-dist-left*: *monom-mult* $(c + d)$ t p = (*monom-mult* c t p) + (*monom-mult* d t p)
⟨*proof*⟩

lemma *monom-mult-dist-left-minus*:
monom-mult $(c - d)$ t p = (*monom-mult* c t p) − (*monom-mult* $(d::'b::ring)$ t p)
⟨*proof*⟩

lemma *monom-mult-dist-right*:
monom-mult c t $(p + q)$ = (*monom-mult* c t p) + (*monom-mult* c t q)
⟨*proof*⟩

lemma *monom-mult-dist-right-minus*:
monom-mult c t $(p - q)$ = (*monom-mult* c t p) − (*monom-mult* $(c::'b::ring)$ t q)
⟨*proof*⟩

lemma *monom-mult-zero-left* [*simp*]: *monom-mult* 0 t p = 0
⟨*proof*⟩

lemma *monom-mult-zero-right* [*simp*]: *monom-mult* c t 0 = 0
⟨*proof*⟩

lemma *monom-mult-one-left* [*simp*]: (*monom-mult* $(1::'b::semiring-1)$ 0 p) = p
⟨*proof*⟩

lemma *monom-mult-monomial*:
monom-mult c s (*monomial* d v) = *monomial* $(c * (d::'b::semiring-0))$ $(s \oplus v)$
⟨*proof*⟩

lemma *monom-mult-eq-zero-iff*: (*monom-mult* c t $p = 0$) ⟷ $((c::'b::semiring-no-zero-divisors) = 0 \vee p = 0)$
⟨*proof*⟩

lemma *lookup-monom-mult-zero*: *lookup* (*monom-mult* c 0 p) t = c * *lookup* p t
⟨*proof*⟩

lemma *monom-mult-inj-1*:
assumes *monom-mult c1 t p* = *monom-mult c2 t p*
and $(p::(- \Rightarrow_0 'b::semiring-no-zero-divisors-cancel)) \neq 0$
shows *c1* = *c2*
⟨*proof*⟩

Multiplication by a monomial is injective in the second argument (the power-product) only in context *ordered-powerprod*; see lemma *monom-mult-inj-2* below.

lemma *monom-mult-inj-3*:
assumes *monom-mult c t p1* = *monom-mult c t* $(p2::(- \Rightarrow_0 'b::semiring-no-zero-divisors-cancel))$

and $c \neq 0$
shows $p1 = p2$
⟨*proof*⟩

lemma *keys-monom-multI*:
assumes $v \in keys\ p$ **and** $c \neq (0::'b::semiring\text{-}no\text{-}zero\text{-}divisors)$
shows $t \oplus v \in keys\ (monom\text{-}mult\ c\ t\ p)$
⟨*proof*⟩

lemma *keys-monom-mult-subset*: $keys\ (monom\text{-}mult\ c\ t\ p) \subseteq ((\oplus)\ t)\ `\ (keys\ p)$
⟨*proof*⟩

lemma *keys-monom-multE*:
assumes $v \in keys\ (monom\text{-}mult\ c\ t\ p)$
obtains u **where** $u \in keys\ p$ **and** $v = t \oplus u$
⟨*proof*⟩

lemma *keys-monom-mult*:
assumes $c \neq (0::'b::semiring\text{-}no\text{-}zero\text{-}divisors)$
shows $keys\ (monom\text{-}mult\ c\ t\ p) = ((\oplus)\ t)\ `\ (keys\ p)$
⟨*proof*⟩

lemma *monom-mult-when*: $monom\text{-}mult\ c\ t\ (p\ when\ P) = ((monom\text{-}mult\ c\ t\ p)\ when\ P)$
⟨*proof*⟩

lemma *when-monom-mult*: $monom\text{-}mult\ (c\ when\ P)\ t\ p = ((monom\text{-}mult\ c\ t\ p)\ when\ P)$
⟨*proof*⟩

lemma *monomial-power*: $(monomial\ c\ t)\ \hat{}\ n = monomial\ (c\ \hat{}\ n)\ (\sum i{=}0..{<}n.\ t)$
⟨*proof*⟩

9.5 Component-wise Lifting

Component-wise lifting of functions on $'a \Rightarrow_0 'b$ to functions on $'t \Rightarrow_0 'b$.

definition *lift-poly-fun-2* :: $(('a \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'b)) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b::zero)$
where $lift\text{-}poly\text{-}fun\text{-}2\ f\ p\ q = atomize\text{-}poly\ (mapp\text{-}2\ (\lambda\text{-}.\ f)\ (vectorize\text{-}poly\ p)\ (vectorize\text{-}poly\ q))$

definition *lift-poly-fun* :: $(('a \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'b)) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b::zero)$
where $lift\text{-}poly\text{-}fun\ f\ p = lift\text{-}poly\text{-}fun\text{-}2\ (\lambda\text{-}.\ f)\ 0\ p$

lemma *lookup-lift-poly-fun-2*:
$lookup\ (lift\text{-}poly\text{-}fun\text{-}2\ f\ p\ q)\ v =$
$(lookup\ (f\ (proj\text{-}poly\ (component\text{-}of\text{-}term\ v)\ p)\ (proj\text{-}poly\ (component\text{-}of\text{-}term$

$v)\ q))\ (pp\text{-}of\text{-}term\ v)$
$\quad when\ component\text{-}of\text{-}term\ v \in keys\ (vectorize\text{-}poly\ p) \cup keys\ (vectorize\text{-}poly\ q))$
⟨*proof*⟩

lemma *lookup-lift-poly-fun*:
$lookup\ (lift\text{-}poly\text{-}fun\ f\ p)\ v =$
$(lookup\ (f\ (proj\text{-}poly\ (component\text{-}of\text{-}term\ v)\ p))\ (pp\text{-}of\text{-}term\ v)\ when\ component\text{-}of\text{-}term\ v \in keys\ (vectorize\text{-}poly\ p))$
⟨*proof*⟩

lemma *lookup-lift-poly-fun-2-homogenous*:
assumes $f\ 0\ 0 = 0$
shows $lookup\ (lift\text{-}poly\text{-}fun\text{-}2\ f\ p\ q)\ v =$
$\quad lookup\ (f\ (proj\text{-}poly\ (component\text{-}of\text{-}term\ v)\ p)\ (proj\text{-}poly\ (component\text{-}of\text{-}term\ v)\ q))\ (pp\text{-}of\text{-}term\ v)$
⟨*proof*⟩

lemma *proj-lift-poly-fun-2-homogenous*:
assumes $f\ 0\ 0 = 0$
shows $proj\text{-}poly\ k\ (lift\text{-}poly\text{-}fun\text{-}2\ f\ p\ q) = f\ (proj\text{-}poly\ k\ p)\ (proj\text{-}poly\ k\ q)$
⟨*proof*⟩

lemma *lookup-lift-poly-fun-homogenous*:
assumes $f\ 0 = 0$
shows $lookup\ (lift\text{-}poly\text{-}fun\ f\ p)\ v = lookup\ (f\ (proj\text{-}poly\ (component\text{-}of\text{-}term\ v)\ p))\ (pp\text{-}of\text{-}term\ v)$
⟨*proof*⟩

lemma *proj-lift-poly-fun-homogenous*:
assumes $f\ 0 = 0$
shows $proj\text{-}poly\ k\ (lift\text{-}poly\text{-}fun\ f\ p) = f\ (proj\text{-}poly\ k\ p)$
⟨*proof*⟩

9.6 Component-wise Multiplication

definition $mult\text{-}vec :: ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b{::}semiring\text{-}0)$ (**infixl** $**\ 75$)
where $mult\text{-}vec = lift\text{-}poly\text{-}fun\text{-}2\ (*)$

lemma *lookup-mult-vec*:
$lookup\ (p\ **\ q)\ v = lookup\ ((proj\text{-}poly\ (component\text{-}of\text{-}term\ v)\ p) * (proj\text{-}poly\ (component\text{-}of\text{-}term\ v)\ q))\ (pp\text{-}of\text{-}term\ v)$
⟨*proof*⟩

lemma *proj-mult-vec* [*term-simps*]: $proj\text{-}poly\ k\ (p\ **\ q) = (proj\text{-}poly\ k\ p) * (proj\text{-}poly\ k\ q)$
⟨*proof*⟩

lemma *mult-vec-zero-left*: $0 ** p = 0$
⟨*proof*⟩

lemma *mult-vec-zero-right*: $p ** 0 = 0$
⟨*proof*⟩

lemma *mult-vec-assoc*: $(p ** q) ** r = p ** (q ** r)$
⟨*proof*⟩

lemma *mult-vec-distrib-right*: $(p + q) ** r = p ** r + q ** r$
⟨*proof*⟩

lemma *mult-vec-distrib-left*: $r ** (p + q) = r ** p + r ** q$
⟨*proof*⟩

lemma *mult-vec-minus-mult-left*: $(- p) ** q = - (p ** q)$
⟨*proof*⟩

lemma *mult-vec-minus-mult-right*: $p ** (- q) = - (p ** q)$
⟨*proof*⟩

lemma *minus-mult-vec-minus*: $(- p) ** (- q) = p ** q$
⟨*proof*⟩

lemma *minus-mult-vec-commute*: $(- p) ** q = p ** (- q)$
⟨*proof*⟩

lemma *mult-vec-right-diff-distrib*: $r ** (p - q) = r ** p - r ** q$
for r::- $\Rightarrow_0$ $'b$::*ring*
⟨*proof*⟩

lemma *mult-vec-left-diff-distrib*: $(p - q) ** r = p ** r - q ** r$
for p::- $\Rightarrow_0$ $'b$::*ring*
⟨*proof*⟩

lemma *mult-vec-commute*: $p ** q = q ** p$ **for** p::- $\Rightarrow_0$ $'b$::*comm-semiring-0*
⟨*proof*⟩

lemma *mult-vec-left-commute*: $p ** (q ** r) = q ** (p ** r)$
for p::- $\Rightarrow_0$ $'b$::*comm-semiring-0*
⟨*proof*⟩

lemma *mult-vec-monomial-monomial*:
$(monomial\ c\ u) ** (monomial\ d\ v) =$
$(monomial\ (c * d)\ (term\text{-}of\text{-}pair\ (pp\text{-}of\text{-}term\ u + pp\text{-}of\text{-}term\ v, component\text{-}of\text{-}term\ u))\ when$
$component\text{-}of\text{-}term\ u = component\text{-}of\text{-}term\ v)$
⟨*proof*⟩

lemma *mult-vec-rec-left*: p ** q = *monomial* (*lookup* p v) v ** q + (*except* p {v}) ** q
⟨*proof*⟩

lemma *mult-vec-rec-right*: p ** q = p ** *monomial* (*lookup* q v) v + p ** *except* q {v}
⟨*proof*⟩

lemma *in-keys-mult-vecE*:
assumes $w \in$ *keys* (p ** q)
obtains u v **where** $u \in$ *keys* p **and** $v \in$ *keys* q **and** *component-of-term* u = *component-of-term* v
and w = *term-of-pair* (*pp-of-term* u + *pp-of-term* v, *component-of-term* u)
⟨*proof*⟩

lemma *lookup-mult-vec-monomial-left*:
lookup (*monomial* c v ** p) u =
(c * *lookup* p (*term-of-pair* (*pp-of-term* u − *pp-of-term* v, *component-of-term* u)) *when* v *adds*$_t$ u)
⟨*proof*⟩

lemma *lookup-mult-vec-monomial-right*:
lookup (p ** *monomial* c v) u =
(*lookup* p (*term-of-pair* (*pp-of-term* u − *pp-of-term* v, *component-of-term* u)) * c *when* v *adds*$_t$ u)
⟨*proof*⟩

9.7 Scalar Multiplication

definition *mult-scalar* :: ('a ⇒$_0$ 'b) ⇒ ('t ⇒$_0$ 'b) ⇒ ('t ⇒$_0$ 'b::*semiring-0*) (**infixl** ⊙ 75)
where *mult-scalar* p = *lift-poly-fun* ((*) p)

lemma *lookup-mult-scalar*:
lookup (p ⊙ q) v = *lookup* (p * (*proj-poly* (*component-of-term* v) q)) (*pp-of-term* v)
⟨*proof*⟩

lemma *lookup-mult-scalar-explicit*:
lookup (p ⊙ q) u = (∑ t∈*keys* p. *lookup* p t * (∑ v∈*keys* q. *lookup* q v *when* u = t ⊕ v))
⟨*proof*⟩

lemma *proj-mult-scalar* [*term-simps*]: *proj-poly* k (p ⊙ q) = p * (*proj-poly* k q)
⟨*proof*⟩

lemma *mult-scalar-zero-left* [*simp*]: 0 ⊙ p = 0
⟨*proof*⟩

lemma *mult-scalar-zero-right* [*simp*]: $p \odot 0 = 0$
⟨*proof*⟩

lemma *mult-scalar-one* [*simp*]: $(1::- \Rightarrow_0 'b::semiring\text{-}1) \odot p = p$
⟨*proof*⟩

lemma *mult-scalar-assoc* [*ac-simps*]: $(p * q) \odot r = p \odot (q \odot r)$
⟨*proof*⟩

lemma *mult-scalar-distrib-right* [*algebra-simps*]: $(p + q) \odot r = p \odot r + q \odot r$
⟨*proof*⟩

lemma *mult-scalar-distrib-left* [*algebra-simps*]: $r \odot (p + q) = r \odot p + r \odot q$
⟨*proof*⟩

lemma *mult-scalar-minus-mult-left* [*simp*]: $(- p) \odot q = - (p \odot q)$
⟨*proof*⟩

lemma *mult-scalar-minus-mult-right* [*simp*]: $p \odot (- q) = - (p \odot q)$
⟨*proof*⟩

lemma *minus-mult-scalar-minus* [*simp*]: $(- p) \odot (- q) = p \odot q$
⟨*proof*⟩

lemma *minus-mult-scalar-commute*: $(- p) \odot q = p \odot (- q)$
⟨*proof*⟩

lemma *mult-scalar-right-diff-distrib* [*algebra-simps*]: $r \odot (p - q) = r \odot p - r \odot q$
for $r::- \Rightarrow_0 'b::ring$
⟨*proof*⟩

lemma *mult-scalar-left-diff-distrib* [*algebra-simps*]: $(p - q) \odot r = p \odot r - q \odot r$
for $p::- \Rightarrow_0 'b::ring$
⟨*proof*⟩

lemma *sum-mult-scalar-distrib-left*: $r \odot (sum\ f\ A) = (\sum a{\in}A.\ r \odot f\ a)$
⟨*proof*⟩

lemma *sum-mult-scalar-distrib-right*: $(sum\ f\ A) \odot v = (\sum a{\in}A.\ f\ a \odot v)$
⟨*proof*⟩

lemma *mult-scalar-monomial-monomial*: $(monomial\ c\ t) \odot (monomial\ d\ v) = monomial\ (c * d)\ (t \oplus v)$
⟨*proof*⟩

lemma *mult-scalar-monomial*: $(monomial\ c\ t) \odot p = monom\text{-}mult\ c\ t\ p$
⟨*proof*⟩

lemma *mult-scalar-rec-left*: $p \odot q = monom\text{-}mult\ (lookup\ p\ t)\ t\ q + (except\ p\ \{t\}) \odot q$
⟨*proof*⟩

lemma *mult-scalar-rec-right*: $p \odot q = p \odot monomial\ (lookup\ q\ v)\ v + p \odot except\ q\ \{v\}$
⟨*proof*⟩

lemma *in-keys-mult-scalarE*:
assumes $v \in keys\ (p \odot q)$
obtains $t\ u$ **where** $t \in keys\ p$ **and** $u \in keys\ q$ **and** $v = t \oplus u$
⟨*proof*⟩

lemma *lookup-mult-scalar-monomial-right*:
$lookup\ (p \odot monomial\ c\ v)\ u = (lookup\ p\ (pp\text{-}of\text{-}term\ u - pp\text{-}of\text{-}term\ v) * c\ when\ v\ adds_t\ u)$
⟨*proof*⟩

lemma *lookup-mult-scalar-monomial-right-plus*: $lookup\ (p \odot monomial\ c\ v)\ (t \oplus v) = lookup\ p\ t * c$
⟨*proof*⟩

lemma *keys-mult-scalar-monomial-right-subset*: $keys\ (p \odot monomial\ c\ v) \subseteq (\lambda t.\ t \oplus v)\ `\ keys\ p$
⟨*proof*⟩

lemma *keys-mult-scalar-monomial-right*:
assumes $c \neq (0::'b::semiring\text{-}no\text{-}zero\text{-}divisors)$
shows $keys\ (p \odot monomial\ c\ v) = (\lambda t.\ t \oplus v)\ `\ keys\ p$
⟨*proof*⟩

end

9.8 Sums and Products

lemma *sum-poly-mapping-eq-zeroI*:
assumes $p\ `\ A \subseteq \{0\}$
shows $sum\ p\ A = (0::(- \Rightarrow_0 'b::comm\text{-}monoid\text{-}add))$
⟨*proof*⟩

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

Legacy:

lemmas *keys-sum-subset* = *Poly-Mapping.keys-sum*

lemma *keys-sum-list-subset*: $keys\ (sum\text{-}list\ ps) \subseteq Keys\ (set\ ps)$
⟨*proof*⟩

lemma *keys-sum*:
assumes *finite A* **and** $\bigwedge a1\ a2.\ a1 \in A \Longrightarrow a2 \in A \Longrightarrow a1 \neq a2 \Longrightarrow keys\ (f\ a1) \cap keys\ (f\ a2) = \{\}$
shows $keys\ (sum\ f\ A) = (\bigcup a{\in}A.\ keys\ (f\ a))$
⟨*proof*⟩

lemma *poly-mapping-sum-monomials*: $(\sum a{\in}keys\ p.\ monomial\ (lookup\ p\ a)\ a) = p$
⟨*proof*⟩

lemma *monomial-sum*: $monomial\ (sum\ f\ C)\ a = (\sum c{\in}C.\ monomial\ (f\ c)\ a)$
⟨*proof*⟩

lemma *monomial-Sum-any*:
assumes *finite* $\{c.\ f\ c \neq 0\}$
shows $monomial\ (Sum\text{-}any\ f)\ a = (\sum c.\ monomial\ (f\ c)\ a)$
⟨*proof*⟩

context *term-powerprod*
begin

lemma *proj-sum*: $proj\text{-}poly\ k\ (sum\ f\ A) = (\sum a{\in}A.\ proj\text{-}poly\ k\ (f\ a))$
⟨*proof*⟩

lemma *proj-sum-list*: $proj\text{-}poly\ k\ (sum\text{-}list\ xs) = sum\text{-}list\ (map\ (proj\text{-}poly\ k)\ xs)$
⟨*proof*⟩

lemma *mult-scalar-sum-monomials*: $q \odot p = (\sum t{\in}keys\ q.\ monom\text{-}mult\ (lookup\ q\ t)\ t\ p)$
⟨*proof*⟩

lemma *fun-mult-scalar-commute*:
assumes $f\ 0 = 0$ **and** $\bigwedge x\ y.\ f\ (x + y) = f\ x + f\ y$
and $\bigwedge c\ t.\ f\ (monom\text{-}mult\ c\ t\ p) = monom\text{-}mult\ c\ t\ (f\ p)$
shows $f\ (q \odot p) = q \odot (f\ p)$
⟨*proof*⟩

lemma *fun-mult-scalar-commute-canc*:
assumes $\bigwedge x\ y.\ f\ (x + y) = f\ x + f\ y$ **and** $\bigwedge c\ t.\ f\ (monom\text{-}mult\ c\ t\ p) = monom\text{-}mult\ c\ t\ (f\ p)$
shows $f\ (q \odot p) = q \odot (f\ (p{::}'t \Rightarrow_0 'b{::}\{semiring\text{-}0, cancel\text{-}comm\text{-}monoid\text{-}add\}))$
⟨*proof*⟩

lemma *monom-mult-sum-left*: $monom\text{-}mult\ (sum\ f\ C)\ t\ p = (\sum c{\in}C.\ monom\text{-}mult\ (f\ c)\ t\ p)$
⟨*proof*⟩

lemma *monom-mult-sum-right*: $monom\text{-}mult\ c\ t\ (sum\ f\ P) = (\sum p{\in}P.\ monom\text{-}mult\ c\ t\ (f\ p))$

⟨*proof*⟩

lemma *monom-mult-Sum-any-left*:
assumes *finite* {*c*. *f c* ≠ *0*}
shows *monom-mult* (*Sum-any f*) *t p* = (∑ *c*. *monom-mult* (*f c*) *t p*)
⟨*proof*⟩

lemma *monom-mult-Sum-any-right*:
assumes *finite* {*p*. *f p* ≠ *0*}
shows *monom-mult c t* (*Sum-any f*) = (∑ *p*. *monom-mult c t* (*f p*))
⟨*proof*⟩

lemma *monomial-prod-sum*: *monomial* (*prod c I*) (*sum a I*) = (∏ *i*∈*I*. *monomial* (*c i*) (*a i*))
⟨*proof*⟩

9.9 Submodules

sublocale *pmdl*: *module mult-scalar*
⟨*proof*⟩

lemmas [*simp del*] = *pmdl.scale-one pmdl.scale-zero-left pmdl.scale-zero-right pmdl.scale-scale pmdl.scale-minus-left pmdl.scale-minus-right pmdl.span-eq-iff*

lemmas [*algebra-simps del*] = *pmdl.scale-left-distrib pmdl.scale-right-distrib pmdl.scale-left-diff-distrib pmdl.scale-right-diff-distrib*

abbreviation *pmdl* ≡ *pmdl.span*

lemma *pmdl-closed-monom-mult*:
assumes *p* ∈ *pmdl B*
shows *monom-mult c t p* ∈ *pmdl B*
⟨*proof*⟩

lemma *monom-mult-in-pmdl*: *b* ∈ *B* ⟹ *monom-mult c t b* ∈ *pmdl B*
⟨*proof*⟩

lemma *pmdl-induct* [*consumes 1*, *case-names module-0 module-plus*]:
assumes *p* ∈ *pmdl B* **and** *P 0*
and ⋀*a p c t*. *a* ∈ *pmdl B* ⟹ *P a* ⟹ *p* ∈ *B* ⟹ *c* ≠ *0* ⟹ *P* (*a* + *monom-mult c t p*)
shows *P p*
⟨*proof*⟩

lemma *components-pmdl*: *component-of-term* ‘ *Keys* (*pmdl B*) = *component-of-term* ‘ *Keys B*
⟨*proof*⟩

lemma *pmdl-idI*:

assumes $0 \in B$ **and** $\bigwedge b1\ b2.\ b1 \in B \Longrightarrow b2 \in B \Longrightarrow b1 + b2 \in B$
and $\bigwedge c\ t\ b.\ b \in B \Longrightarrow monom\text{-}mult\ c\ t\ b \in B$
shows $pmdl\ B = B$
⟨*proof*⟩

definition $full\text{-}pmdl :: 'k\ set \Rightarrow ('t \Rightarrow_0 'b{::}zero)\ set$
where $full\text{-}pmdl\ K = \{p.\ component\text{-}of\text{-}term\ `\ keys\ p \subseteq K\}$

definition $is\text{-}full\text{-}pmdl :: ('t \Rightarrow_0 'b{::}comm\text{-}ring\text{-}1)\ set \Rightarrow bool$
where $is\text{-}full\text{-}pmdl\ B \longleftrightarrow (\forall p.\ component\text{-}of\text{-}term\ `\ keys\ p \subseteq component\text{-}of\text{-}term\ `\ Keys\ B \longrightarrow p \in pmdl\ B)$

lemma *full-pmdl-iff*: $p \in full\text{-}pmdl\ K \longleftrightarrow component\text{-}of\text{-}term\ `\ keys\ p \subseteq K$
⟨*proof*⟩

lemma *full-pmdlI*:
assumes $\bigwedge v.\ v \in keys\ p \Longrightarrow component\text{-}of\text{-}term\ v \in K$
shows $p \in full\text{-}pmdl\ K$
⟨*proof*⟩

lemma *full-pmdlD*:
assumes $p \in full\text{-}pmdl\ K$ **and** $v \in keys\ p$
shows $component\text{-}of\text{-}term\ v \in K$
⟨*proof*⟩

lemma *full-pmdl-empty*: $full\text{-}pmdl\ \{\} = \{0\}$
⟨*proof*⟩

lemma *full-pmdl-UNIV*: $full\text{-}pmdl\ UNIV = UNIV$
⟨*proof*⟩

lemma *zero-in-full-pmdl*: $0 \in full\text{-}pmdl\ K$
⟨*proof*⟩

lemma *full-pmdl-closed-plus*:
assumes $p \in full\text{-}pmdl\ K$ **and** $q \in full\text{-}pmdl\ K$
shows $p + q \in full\text{-}pmdl\ K$
⟨*proof*⟩

lemma *full-pmdl-closed-monom-mult*:
assumes $p \in full\text{-}pmdl\ K$
shows $monom\text{-}mult\ c\ t\ p \in full\text{-}pmdl\ K$
⟨*proof*⟩

lemma *pmdl-full-pmdl*: $pmdl\ (full\text{-}pmdl\ K) = full\text{-}pmdl\ K$
⟨*proof*⟩

lemma *components-full-pmdl-subset*:
$component\text{-}of\text{-}term\ `\ Keys\ ((full\text{-}pmdl\ K){::}('t \Rightarrow_0 'b{::}zero)\ set) \subseteq K$ (**is** $?l \subseteq$ -)

⟨*proof*⟩

lemma *components-full-pmdl*:
component-of-term ‘ Keys ((*full-pmdl K*)::(*'t* ⇒$_0$ *'b*::*zero-neq-one*) *set*) = *K* (**is** *?l* = -)
⟨*proof*⟩

lemma *is-full-pmdlI*:
assumes ⋀*p*. *component-of-term ‘ keys p* ⊆ *component-of-term ‘ Keys B* ⟹ *p* ∈ *pmdl B*
shows *is-full-pmdl B*
⟨*proof*⟩

lemma *is-full-pmdlD*:
assumes *is-full-pmdl B* **and** *component-of-term ‘ keys p* ⊆ *component-of-term ‘ Keys B*
shows *p* ∈ *pmdl B*
⟨*proof*⟩

lemma *is-full-pmdl-alt*: *is-full-pmdl B* ⟷ *pmdl B* = *full-pmdl* (*component-of-term ‘ Keys B*)
⟨*proof*⟩

lemma *is-full-pmdl-pmdl*: *is-full-pmdl* (*pmdl B*) ⟷ *is-full-pmdl B*
⟨*proof*⟩

lemma *is-full-pmdl-subset*:
assumes *is-full-pmdl B1* **and** *is-full-pmdl B2*
and *component-of-term ‘ Keys B1* ⊆ *component-of-term ‘ Keys B2*
shows *pmdl B1* ⊆ *pmdl B2*
⟨*proof*⟩

lemma *is-full-pmdl-eq*:
assumes *is-full-pmdl B1* **and** *is-full-pmdl B2*
and *component-of-term ‘ Keys B1* = *component-of-term ‘ Keys B2*
shows *pmdl B1* = *pmdl B2*
⟨*proof*⟩

end

definition *map-scale* :: *'b* ⇒ (*'a* ⇒$_0$ *'b*) ⇒ (*'a* ⇒$_0$ *'b*::*mult-zero*) (**infixr** · *71*)
where *map-scale c* = *Poly-Mapping.map* ((∗) *c*)

If the polynomial mapping p is interpreted as a power-product, then $c \cdot p$ corresponds to exponentiation; if it is interpreted as a (vector-) polynomial, then $c \cdot p$ corresponds to multiplication by scalar from the coefficient type.

lemma *lookup-map-scale* [*simp*]: *lookup* (*c* · *p*) = (λ*x*. *c* ∗ *lookup p x*)
⟨*proof*⟩

lemma *map-scale-single* [*simp*]: $k \cdot$ *Poly-Mapping.single* $x\ l =$ *Poly-Mapping.single* $x\ (k * l)$
⟨*proof*⟩

lemma *map-scale-zero-left* [*simp*]: $0 \cdot t = 0$
⟨*proof*⟩

lemma *map-scale-zero-right* [*simp*]: $k \cdot 0 = 0$
⟨*proof*⟩

lemma *map-scale-eq-0-iff*: $c \cdot t = 0 \longleftrightarrow ((c::\text{-}::\textit{semiring-no-zero-divisors}) = 0 \vee t = 0)$
⟨*proof*⟩

lemma *keys-map-scale-subset*: *keys* $(k \cdot t) \subseteq$ *keys* t
⟨*proof*⟩

lemma *keys-map-scale*: *keys* $((k::'b::\textit{semiring-no-zero-divisors}) \cdot t) = (\textit{if } k = 0 \textit{ then } \{\} \textit{ else keys } t)$
⟨*proof*⟩

lemma *map-scale-one-left* [*simp*]: $(1::'b::\{\textit{mult-zero},\textit{monoid-mult}\}) \cdot t = t$
⟨*proof*⟩

lemma *map-scale-assoc* [*ac-simps*]: $c \cdot d \cdot t = (c * d) \cdot (t::\text{-} \Rightarrow_0 \text{-}::\{\textit{semigroup-mult},\textit{zero}\})$
⟨*proof*⟩

lemma *map-scale-distrib-left* [*algebra-simps*]: $(k::'b::\textit{semiring-0}) \cdot (s + t) = k \cdot s + k \cdot t$
⟨*proof*⟩

lemma *map-scale-distrib-right* [*algebra-simps*]: $(k + (l::'b::\textit{semiring-0})) \cdot t = k \cdot t + l \cdot t$
⟨*proof*⟩

lemma *map-scale-Suc*: $(\textit{Suc } k) \cdot t = k \cdot t + t$
⟨*proof*⟩

lemma *map-scale-uminus-left*: $(-\ k::'b::\textit{ring}) \cdot p = -\ (k \cdot p)$
⟨*proof*⟩

lemma *map-scale-uminus-right*: $(k::'b::\textit{ring}) \cdot (-\ p) = -\ (k \cdot p)$
⟨*proof*⟩

lemma *map-scale-uminus-uminus* [*simp*]: $(-\ k::'b::\textit{ring}) \cdot (-\ p) = k \cdot p$
⟨*proof*⟩

lemma *map-scale-minus-distrib-left* [*algebra-simps*]:
$(k::'b::\textit{comm-semiring-1-cancel}) \cdot (p - q) = k \cdot p - k \cdot q$

⟨*proof*⟩

lemma *map-scale-minus-distrib-right* [*algebra-simps*]:
$(k - (l::'b::comm\text{-}semiring\text{-}1\text{-}cancel)) \cdot f = k \cdot f - l \cdot f$
⟨*proof*⟩

lemma *map-scale-sum-distrib-left*: $(k::'b::semiring\text{-}0) \cdot (sum\ f\ A) = (\sum a{\in}A.\ k \cdot f\ a)$
⟨*proof*⟩

lemma *map-scale-sum-distrib-right*: $(sum\ (f::- \Rightarrow 'b::semiring\text{-}0)\ A) \cdot p = (\sum a{\in}A.\ f\ a \cdot p)$
⟨*proof*⟩

lemma *deg-pm-map-scale*: $deg\text{-}pm\ (k \cdot t) = (k::'b::semiring\text{-}0) * deg\text{-}pm\ t$
⟨*proof*⟩

interpretation *phull*: *module map-scale*
⟨*proof*⟩

Since the following lemmas are proved for more general ring-types above, we do not need to have them in the simpset.

lemmas [*simp del*] = *phull.scale-one phull.scale-zero-left phull.scale-zero-right phull.scale-scale phull.scale-minus-left phull.scale-minus-right phull.span-eq-iff*

lemmas [*algebra-simps del*] = *phull.scale-left-distrib phull.scale-right-distrib phull.scale-left-diff-distrib phull.scale-right-diff-distrib*

abbreviation *phull* ≡ *phull.span*

phull B is a module over the coefficient ring $'b$, whereas λ*term-of-pair*. *module.span* (*term-powerprod.mult-scalar B term-of-pair*) is a module over the (scalar) polynomial ring $'a \Rightarrow_0 'b$. Nevertheless, both modules can be sets of *vector-polynomials* of type $'t \Rightarrow_0 'b$.

context *term-powerprod*
begin

lemma *map-scale-eq-monom-mult*: $c \cdot p = monom\text{-}mult\ c\ 0\ p$
⟨*proof*⟩

lemma *map-scale-eq-mult-scalar*: $c \cdot p = monomial\ c\ 0 \odot p$
⟨*proof*⟩

lemma *phull-closed-mult-scalar*: $p \in phull\ B \Longrightarrow monomial\ c\ 0 \odot p \in phull\ B$
⟨*proof*⟩

lemma *mult-scalar-in-phull*: $b \in B \Longrightarrow monomial\ c\ 0 \odot b \in phull\ B$
⟨*proof*⟩

lemma *phull-subset-module*: *phull B* $\subseteq$ *pmdl B*
⟨*proof*⟩

lemma *components-phull*: *component-of-term* ‘ *Keys* (*phull B*) = *component-of-term* ‘ *Keys B*
⟨*proof*⟩

end

9.10 Interpretations

9.10.1 Isomorphism between $'a$ and $'a \times unit$

definition *to-pair-unit* :: $'a \Rightarrow ('a \times unit)$
  where *to-pair-unit x* = (*x*, ())

lemma *fst-to-pair-unit*: *fst* (*to-pair-unit x*) = *x*
  ⟨*proof*⟩

lemma *to-pair-unit-fst*: *to-pair-unit* (*fst x*) = (*x*::- × *unit*)
  ⟨*proof*⟩

interpretation *punit*: *term-powerprod to-pair-unit fst*
  ⟨*proof*⟩

For technical reasons it seems to be better not to put the following lemmas as rewrite-rules of interpretation *punit*.

lemma *punit-pp-of-term* [*simp*]: *punit.pp-of-term* = (λ*x*. *x*)
  ⟨*proof*⟩

lemma *punit-component-of-term* [*simp*]: *punit.component-of-term* = (λ-. ())
  ⟨*proof*⟩

lemma *punit-splus* [*simp*]: *punit.splus* = (+)
  ⟨*proof*⟩

lemma *punit-sminus* [*simp*]: *punit.sminus* = (−)
  ⟨*proof*⟩

lemma *punit-adds-pp* [*simp*]: *punit.adds-pp* = (*adds*)
  ⟨*proof*⟩

lemma *punit-adds-term* [*simp*]: *punit.adds-term* = (*adds*)
  ⟨*proof*⟩

lemma *punit-proj-poly* [*simp*]: *punit.proj-poly* = (λ-. *id*)
  ⟨*proof*⟩

lemma *punit-mult-vec* [*simp*]: *punit.mult-vec* = (∗)
  ⟨*proof*⟩

lemma *punit-mult-scalar* [*simp*]: *punit.mult-scalar* = (∗)
⟨*proof*⟩

context *term-powerprod*
begin

lemma *proj-monom-mult*: *proj-poly k* (*monom-mult c t p*) = *punit.monom-mult c t* (*proj-poly k p*)
⟨*proof*⟩

lemma *mult-scalar-monom-mult*: (*punit.monom-mult c t p*) ⊙ *q* = *monom-mult c t* (*p* ⊙ *q*)
⟨*proof*⟩

end

9.10.2 Interpretation of *term-powerprod* by $'a \times 'k$

interpretation *pprod*: *term-powerprod* (λ*x*::*'a*::*comm-powerprod* × *'k*::*linorder*. *x*) λ*x*. *x*
⟨*proof*⟩

lemma *pprod-pp-of-term* [*simp*]: *pprod.pp-of-term* = *fst*
⟨*proof*⟩

lemma *pprod-component-of-term* [*simp*]: *pprod.component-of-term* = *snd*
⟨*proof*⟩

9.10.3 Simplifier Setup

There is no reason to keep the interpreted theorems as simplification rules.

lemmas [*term-simps del*] = *term-simps*

lemmas *times-monomial-monomial* = *punit.mult-scalar-monomial-monomial*[*simplified*]
lemmas *times-monomial-left* = *punit.mult-scalar-monomial*[*simplified*]
lemmas *times-rec-left* = *punit.mult-scalar-rec-left*[*simplified*]
lemmas *times-rec-right* = *punit.mult-scalar-rec-right*[*simplified*]
lemmas *in-keys-timesE* = *punit.in-keys-mult-scalarE*[*simplified*]
lemmas *punit-monom-mult-monomial* = *punit.monom-mult-monomial*[*simplified*]
lemmas *lookup-times* = *punit.lookup-mult-scalar-explicit*[*simplified*]
lemmas *map-scale-eq-times* = *punit.map-scale-eq-mult-scalar*[*simplified*]

end

10 Type-Class-Multivariate Polynomials in Ordered Terms

theory *MPoly-Type-Class-Ordered*
imports *MPoly-Type-Class*
begin

class *the-min* = *linorder* +
fixes *the-min*::'*a*
assumes *the-min-min*: *the-min* ≤ *x*

Type class *the-min* guarantees that a least element exists. Instances of *the-min* should provide *computable* definitions of that element.

instantiation *nat* :: *the-min*
begin
definition *the-min-nat* = (*0*::*nat*)
instance ⟨*proof*⟩
end

instantiation *unit* :: *the-min*
begin
definition *the-min-unit* = ()
instance ⟨*proof*⟩
end

locale *ordered-term* =
term-powerprod pair-of-term term-of-pair +
ordered-powerprod ord ord-strict +
ord-term-lin: *linorder ord-term ord-term-strict*
for *pair-of-term*::'*t* ⇒ ('*a*::*comm-powerprod* × '*k*::{*the-min*,*wellorder*})
and *term-of-pair*::('*a* × '*k*) ⇒ '*t*
and *ord*::'*a* ⇒ '*a* ⇒ *bool* (**infixl** ⪯ *50*)
and *ord-strict* (**infixl** ≺ *50*)
and *ord-term*::'*t* ⇒ '*t* ⇒ *bool* (**infixl** ⪯$_t$ *50*)
and *ord-term-strict*::'*t* ⇒ '*t* ⇒ *bool* (**infixl** ≺$_t$ *50*) +
assumes *splus-mono*: *v* ⪯$_t$ *w* ⟹ *t* ⊕ *v* ⪯$_t$ *t* ⊕ *w*
assumes *ord-termI*: *pp-of-term v* ⪯ *pp-of-term w* ⟹ *component-of-term v* ≤ *component-of-term w* ⟹ *v* ⪯$_t$ *w*
begin

abbreviation *ord-term-conv* (**infixl** ⪰$_t$ *50*) **where** *ord-term-conv* ≡ (⪯$_t$)$^{-1-1}$
abbreviation *ord-term-strict-conv* (**infixl** ≻$_t$ *50*) **where** *ord-term-strict-conv* ≡ (≺$_t$)$^{-1-1}$

The definition of *ordered-term* only covers TOP and POT orderings. These two types of orderings are the only interesting ones.

definition *min-term* ≡ *term-of-pair* (*0*, *the-min*)

lemma *min-term-min*: *min-term* ⪯$_t$ *v*

⟨*proof*⟩

lemma *splus-mono-strict*:
assumes $v \prec_t w$
shows $t \oplus v \prec_t t \oplus w$
⟨*proof*⟩

lemma *splus-mono-left*:
assumes $s \preceq t$
shows $s \oplus v \preceq_t t \oplus v$
⟨*proof*⟩

lemma *splus-mono-strict-left*:
assumes $s \prec t$
shows $s \oplus v \prec_t t \oplus v$
⟨*proof*⟩

lemma *ord-term-canc*:
assumes $t \oplus v \preceq_t t \oplus w$
shows $v \preceq_t w$
⟨*proof*⟩

lemma *ord-term-strict-canc*:
assumes $t \oplus v \prec_t t \oplus w$
shows $v \prec_t w$
⟨*proof*⟩

lemma *ord-term-canc-left*:
assumes $t \oplus v \preceq_t s \oplus v$
shows $t \preceq s$
⟨*proof*⟩

lemma *ord-term-strict-canc-left*:
assumes $t \oplus v \prec_t s \oplus v$
shows $t \prec s$
⟨*proof*⟩

lemma *ord-adds-term*:
assumes $u\ adds_t\ v$
shows $u \preceq_t v$
⟨*proof*⟩

end

10.1 Interpretations

context *ordered-powerprod*
begin

10.1.1 Unit

sublocale *punit*: *ordered-term to-pair-unit fst* $(\preceq)$ $(\prec)$ $(\preceq)$ $(\prec)$
⟨*proof*⟩

lemma *punit-min-term* [*simp*]: *punit.min-term = 0*
⟨*proof*⟩

end

10.2 Definitions

context *ordered-term*
begin

definition *higher* :: $('t \Rightarrow_0 'b) \Rightarrow 't \Rightarrow ('t \Rightarrow_0 'b::zero)$
where *higher p t = except p* $\{s.\ s \preceq_t t\}$

definition *lower* :: $('t \Rightarrow_0 'b) \Rightarrow 't \Rightarrow ('t \Rightarrow_0 'b::zero)$
where *lower p t = except p* $\{s.\ t \preceq_t s\}$

definition *lt* :: $('t \Rightarrow_0 'b::zero) \Rightarrow 't$
where *lt p = (if p = 0 then min-term else ord-term-lin.Max (keys p))*

abbreviation *lp p* ≡ *pp-of-term* (*lt p*)

definition *lc* :: $('t \Rightarrow_0 'b::zero) \Rightarrow 'b$
where *lc p = lookup p* (*lt p*)

definition *tt* :: $('t \Rightarrow_0 'b::zero) \Rightarrow 't$
where *tt p = (if p = 0 then min-term else ord-term-lin.Min (keys p))*

abbreviation *tp p* ≡ *pp-of-term* (*tt p*)

definition *tc* :: $('t \Rightarrow_0 'b::zero) \Rightarrow 'b$
where *tc p* ≡ *lookup p* (*tt p*)

definition *tail* :: $('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b::zero)$
where *tail p* ≡ *lower p* (*lt p*)

10.3 Leading Term and Leading Coefficient: *lt* and *lc*

lemma *lt-zero* [*simp*]: *lt 0 = min-term*
⟨*proof*⟩

lemma *lc-zero* [*simp*]: *lc 0 = 0*
⟨*proof*⟩

lemma *lt-uminus* [*simp*]: *lt* (− *p*) = *lt p*
⟨*proof*⟩

lemma *lc-uminus* [*simp*]: $lc\ (-\ p) = -\ lc\ p$
⟨*proof*⟩

lemma *lt-alt*:
assumes $p \neq 0$
shows $lt\ p = ord\text{-}term\text{-}lin.Max\ (keys\ p)$
⟨*proof*⟩

lemma *lt-max*:
assumes $lookup\ p\ v \neq 0$
shows $v \preceq_t lt\ p$
⟨*proof*⟩

lemma *lt-eqI*:
assumes $lookup\ p\ v \neq 0$ **and** $\bigwedge u.\ lookup\ p\ u \neq 0 \Longrightarrow u \preceq_t v$
shows $lt\ p = v$
⟨*proof*⟩

lemma *lt-less*:
assumes $p \neq 0$ **and** $\bigwedge u.\ v \preceq_t u \Longrightarrow lookup\ p\ u = 0$
shows $lt\ p \prec_t v$
⟨*proof*⟩

lemma *lt-le*:
assumes $\bigwedge u.\ v \prec_t u \Longrightarrow lookup\ p\ u = 0$
shows $lt\ p \preceq_t v$
⟨*proof*⟩

lemma *lt-gr*:
assumes $lookup\ p\ s \neq 0$ **and** $t \prec_t s$
shows $t \prec_t lt\ p$
⟨*proof*⟩

lemma *lc-not-0*:
assumes $p \neq 0$
shows $lc\ p \neq 0$
⟨*proof*⟩

lemma *lc-eq-zero-iff*: $lc\ p = 0 \longleftrightarrow p = 0$
⟨*proof*⟩

lemma *lt-in-keys*:
assumes $p \neq 0$
shows $lt\ p \in (keys\ p)$
⟨*proof*⟩

lemma *lt-monomial*:
assumes $c \neq 0$

shows *lt* (*monomial c t*) = *t*
⟨*proof*⟩

lemma *lc-monomial* [*simp*]: *lc* (*monomial c t*) = *c*
⟨*proof*⟩

lemma *lt-le-iff*: *lt p* ⪯$_t$ *v* ⟷ (∀ *u*. *v* ≺$_t$ *u* ⟶ *lookup p u* = *0*) (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *lt-plus-eqI*:
assumes *lt p* ≺$_t$ *lt q*
shows *lt* (*p* + *q*) = *lt q*
⟨*proof*⟩

lemma *lt-plus-eqI-2*:
assumes *lt q* ≺$_t$ *lt p*
shows *lt* (*p* + *q*) = *lt p*
⟨*proof*⟩

lemma *lt-plus-eqI-3*:
assumes *lt q* = *lt p* **and** *lc p* + *lc q* ≠ *0*
shows *lt* (*p* + *q*) = *lt* (*p*::*'t* ⇒$_0$ *'b*::*monoid-add*)
⟨*proof*⟩

lemma *lt-plus-lessE*:
assumes *lt p* ≺$_t$ *lt* (*p* + *q*)
shows *lt p* ≺$_t$ *lt q*
⟨*proof*⟩

lemma *lt-plus-lessE-2*:
assumes *lt q* ≺$_t$ *lt* (*p* + *q*)
shows *lt q* ≺$_t$ *lt p*
⟨*proof*⟩

lemma *lt-plus-lessI'*:
fixes *p q* :: *'t* ⇒$_0$ *'b*::*monoid-add*
assumes *p* + *q* ≠ *0* **and** *lt-eq*: *lt q* = *lt p* **and** *lc-eq*: *lc p* + *lc q* = *0*
shows *lt* (*p* + *q*) ≺$_t$ *lt p*
⟨*proof*⟩

corollary *lt-plus-lessI*:
fixes *p q* :: *'t* ⇒$_0$ *'b*::*group-add*
assumes *p* + *q* ≠ *0* **and** *lt q* = *lt p* **and** *lc q* = − *lc p*
shows *lt* (*p* + *q*) ≺$_t$ *lt p*
⟨*proof*⟩

lemma *lt-plus-distinct-eq-max*:
assumes *lt p* ≠ *lt q*
shows *lt* (*p* + *q*) = *ord-term-lin.max* (*lt p*) (*lt q*)

⟨*proof*⟩

lemma *lt-plus-le-max*: *lt* (*p* + *q*) ⪯$_t$ *ord-term-lin.max* (*lt p*) (*lt q*)
⟨*proof*⟩

lemma *lt-minus-eqI*: *lt p* ≺$_t$ *lt q* ⟹ *lt* (*p* − *q*) = *lt q* **for** *p q* :: *'t* ⇒$_0$ *'b*::*ab-group-add*
⟨*proof*⟩

lemma *lt-minus-eqI-2*: *lt q* ≺$_t$ *lt p* ⟹ *lt* (*p* − *q*) = *lt p* **for** *p q* :: *'t* ⇒$_0$ *'b*::*ab-group-add*
⟨*proof*⟩

lemma *lt-minus-eqI-3*:
assumes *lt q* = *lt p* **and** *lc q* ≠ *lc p*
shows *lt* (*p* − *q*) = *lt* (*p*::*'t* ⇒$_0$ *'b*::*ab-group-add*)
⟨*proof*⟩

lemma *lt-minus-distinct-eq-max*:
assumes *lt p* ≠ *lt* (*q*::*'t* ⇒$_0$ *'b*::*ab-group-add*)
shows *lt* (*p* − *q*) = *ord-term-lin.max* (*lt p*) (*lt q*)
⟨*proof*⟩

lemma *lt-minus-lessE*: *lt p* ≺$_t$ *lt* (*p* − *q*) ⟹ *lt p* ≺$_t$ *lt q* **for** *p q* :: *'t* ⇒$_0$ *'b*::*ab-group-add*
⟨*proof*⟩

lemma *lt-minus-lessE-2*: *lt q* ≺$_t$ *lt* (*p* − *q*) ⟹ *lt q* ≺$_t$ *lt p* **for** *p q* :: *'t* ⇒$_0$ *'b*::*ab-group-add*
⟨*proof*⟩

lemma *lt-minus-lessI*: *p* − *q* ≠ *0* ⟹ *lt q* = *lt p* ⟹ *lc q* = *lc p* ⟹ *lt* (*p* − *q*) ≺$_t$ *lt p*
for *p q* :: *'t* ⇒$_0$ *'b*::*ab-group-add*
⟨*proof*⟩

lemma *lt-max-keys*:
assumes *v* ∈ *keys p*
shows *v* ⪯$_t$ *lt p*
⟨*proof*⟩

lemma *lt-eqI-keys*:
assumes *v* ∈ *keys p* **and** *a2*: ⋀*u*. *u* ∈ *keys p* ⟹ *u* ⪯$_t$ *v*
shows *lt p* = *v*
⟨*proof*⟩

lemma *lt-gr-keys*:
assumes *u* ∈ *keys p* **and** *v* ≺$_t$ *u*
shows *v* ≺$_t$ *lt p*
⟨*proof*⟩

lemma *lt-plus-eq-maxI*:
assumes $lt\ p = lt\ q \Longrightarrow lc\ p + lc\ q \neq 0$
shows $lt\ (p + q) = ord\text{-}term\text{-}lin.max\ (lt\ p)\ (lt\ q)$
⟨*proof*⟩

lemma *lt-monom-mult*:
assumes $c \neq (0::'b::semiring\text{-}no\text{-}zero\text{-}divisors)$ **and** $p \neq 0$
shows $lt\ (monom\text{-}mult\ c\ t\ p) = t \oplus lt\ p$
⟨*proof*⟩

lemma *lt-monom-mult-zero*:
assumes $c \neq (0::'b::semiring\text{-}no\text{-}zero\text{-}divisors)$
shows $lt\ (monom\text{-}mult\ c\ 0\ p) = lt\ p$
⟨*proof*⟩

corollary *lt-map-scale*: $c \neq (0::'b::semiring\text{-}no\text{-}zero\text{-}divisors) \Longrightarrow lt\ (c \cdot p) = lt\ p$
⟨*proof*⟩

lemma *lc-monom-mult* [*simp*]: $lc\ (monom\text{-}mult\ c\ t\ p) = (c::'b::semiring\text{-}no\text{-}zero\text{-}divisors) * lc\ p$
⟨*proof*⟩

corollary *lc-map-scale* [*simp*]: $lc\ (c \cdot p) = (c::'b::semiring\text{-}no\text{-}zero\text{-}divisors) * lc\ p$
⟨*proof*⟩

lemma (**in** *ordered-term*) *lt-mult-scalar-monomial-right*:
assumes $c \neq (0::'b::semiring\text{-}no\text{-}zero\text{-}divisors)$ **and** $p \neq 0$
shows $lt\ (p \odot monomial\ c\ v) = punit.lt\ p \oplus v$
⟨*proof*⟩

lemma *lc-mult-scalar-monomial-right*:
$lc\ (p \odot monomial\ c\ v) = punit.lc\ p * (c::'b::semiring\text{-}no\text{-}zero\text{-}divisors)$
⟨*proof*⟩

lemma *lookup-monom-mult-eq-zero*:
assumes $s \oplus lt\ p \prec_t v$
shows $lookup\ (monom\text{-}mult\ (c::'b::semiring\text{-}no\text{-}zero\text{-}divisors)\ s\ p)\ v = 0$
⟨*proof*⟩

lemma *in-keys-monom-mult-le*:
assumes $v \in keys\ (monom\text{-}mult\ c\ t\ p)$
shows $v \preceq_t t \oplus lt\ p$
⟨*proof*⟩

lemma *lt-monom-mult-le*: $lt\ (monom\text{-}mult\ c\ t\ p) \preceq_t t \oplus lt\ p$
⟨*proof*⟩

lemma *monom-mult-inj-2*:

assumes *monom-mult c t1 p* = *monom-mult c t2 p*
and $c \neq 0$ **and** $(p::'t \Rightarrow_0 'b::semiring\text{-}no\text{-}zero\text{-}divisors) \neq 0$
shows *t1* = *t2*
⟨*proof*⟩

10.4 Trailing Term and Trailing Coefficient: *tt* and *tc*

lemma *tt-zero* [*simp*]: *tt 0* = *min-term*
⟨*proof*⟩

lemma *tc-zero* [*simp*]: *tc 0* = *0*
⟨*proof*⟩

lemma *tt-alt*:
assumes $p \neq 0$
shows *tt p* = *ord-term-lin.Min* (*keys p*)
⟨*proof*⟩

lemma *tt-min-keys*:
assumes $v \in keys\ p$
shows $tt\ p \preceq_t v$
⟨*proof*⟩

lemma *tt-min*:
assumes $lookup\ p\ v \neq 0$
shows $tt\ p \preceq_t v$
⟨*proof*⟩

lemma *tt-in-keys*:
assumes $p \neq 0$
shows $tt\ p \in keys\ p$
⟨*proof*⟩

lemma *tt-eqI*:
assumes $v \in keys\ p$ **and** $\bigwedge u.\ u \in keys\ p \Longrightarrow v \preceq_t u$
shows *tt p* = *v*
⟨*proof*⟩

lemma *tt-gr*:
assumes $\bigwedge u.\ u \in keys\ p \Longrightarrow v \prec_t u$ **and** $p \neq 0$
shows $v \prec_t tt\ p$
⟨*proof*⟩

lemma *tt-less*:
assumes $u \in keys\ p$ **and** $u \prec_t v$
shows $tt\ p \prec_t v$
⟨*proof*⟩

lemma *tt-ge*:

assumes $\bigwedge u.\ u \prec_t v \Longrightarrow lookup\ p\ u = 0$ **and** $p \neq 0$
shows $v \preceq_t tt\ p$
⟨*proof*⟩

lemma *tt-ge-keys*:
assumes $\bigwedge u.\ u \in keys\ p \Longrightarrow v \preceq_t u$ **and** $p \neq 0$
shows $v \preceq_t tt\ p$
⟨*proof*⟩

lemma *tt-ge-iff*: $v \preceq_t tt\ p \longleftrightarrow ((p \neq 0 \vee v = min\text{-}term) \wedge (\forall u.\ u \prec_t v \longrightarrow lookup\ p\ u = 0))$
(**is** $?L \longleftrightarrow (?A \wedge ?B)$)
⟨*proof*⟩

lemma *tc-not-0*:
assumes $p \neq 0$
shows $tc\ p \neq 0$
⟨*proof*⟩

lemma *tt-monomial*:
assumes $c \neq 0$
shows $tt\ (monomial\ c\ v) = v$
⟨*proof*⟩

lemma *tc-monomial* [*simp*]: $tc\ (monomial\ c\ t) = c$
⟨*proof*⟩

lemma *tt-plus-eqI*:
assumes $p \neq 0$ **and** $tt\ p \prec_t tt\ q$
shows $tt\ (p + q) = tt\ p$
⟨*proof*⟩

lemma *tt-plus-lessE*:
fixes $p\ q$
assumes $p + q \neq 0$ **and** *tt*: $tt\ (p + q) \prec_t tt\ p$
shows $tt\ q \prec_t tt\ p$
⟨*proof*⟩

lemma *tt-plus-lessI*:
fixes $p\ q :: - \Rightarrow_0 'b{::}ring$
assumes $p + q \neq 0$ **and** *tt-eq*: $tt\ q = tt\ p$ **and** *tc-eq*: $tc\ q = -\ tc\ p$
shows $tt\ p \prec_t tt\ (p + q)$
⟨*proof*⟩

lemma *tt-uminus* [*simp*]: $tt\ (-\ p) = tt\ p$
⟨*proof*⟩

lemma *tc-uminus* [*simp*]: $tc\ (-\ p) = -\ tc\ p$
⟨*proof*⟩

lemma *tt-monom-mult*:
assumes $c \neq (0::'b::semiring\text{-}no\text{-}zero\text{-}divisors)$ **and** $p \neq 0$
shows $tt\ (monom\text{-}mult\ c\ t\ p) = t \oplus tt\ p$
⟨*proof*⟩

lemma *tt-map-scale*: $c \neq (0::'b::semiring\text{-}no\text{-}zero\text{-}divisors) \Longrightarrow tt\ (c \cdot p) = tt\ p$
⟨*proof*⟩

lemma *tc-monom-mult* [*simp*]: $tc\ (monom\text{-}mult\ c\ t\ p) = (c::'b::semiring\text{-}no\text{-}zero\text{-}divisors) * tc\ p$
⟨*proof*⟩

corollary *tc-map-scale* [*simp*]: $tc\ (c \cdot p) = (c::'b::semiring\text{-}no\text{-}zero\text{-}divisors) * tc\ p$
⟨*proof*⟩

lemma *in-keys-monom-mult-ge*:
assumes $v \in keys\ (monom\text{-}mult\ c\ t\ p)$
shows $t \oplus tt\ p \preceq_t v$
⟨*proof*⟩

lemma *lt-ge-tt*: $tt\ p \preceq_t lt\ p$
⟨*proof*⟩

lemma *lt-eq-tt-monomial*:
assumes *is-monomial p*
shows $lt\ p = tt\ p$
⟨*proof*⟩

10.5 *higher* and *lower*

lemma *lookup-higher*: $lookup\ (higher\ p\ u)\ v = (if\ u \prec_t v\ then\ lookup\ p\ v\ else\ 0)$
⟨*proof*⟩

lemma *lookup-higher-when*: $lookup\ (higher\ p\ u)\ v = (lookup\ p\ v\ when\ u \prec_t v)$
⟨*proof*⟩

lemma *higher-plus*: $higher\ (p + q)\ v = higher\ p\ v + higher\ q\ v$
⟨*proof*⟩

lemma *higher-uminus* [*simp*]: $higher\ (-\ p)\ v = -(higher\ p\ v)$
⟨*proof*⟩

lemma *higher-minus*: $higher\ (p - q)\ v = higher\ p\ v - higher\ q\ v$
⟨*proof*⟩

lemma *higher-zero* [*simp*]: $higher\ 0\ t = 0$
⟨*proof*⟩

lemma *higher-eq-iff*: *higher p v = higher q v* $\longleftrightarrow$ $(\forall u.\ v \prec_t u \longrightarrow$ *lookup p u = lookup q u*$)$ (**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

lemma *higher-eq-zero-iff*: *higher p v = 0* $\longleftrightarrow$ $(\forall u.\ v \prec_t u \longrightarrow$ *lookup p u = 0*$)$
⟨*proof*⟩

lemma *keys-higher*: *keys* (*higher p v*) = $\{u \in$*keys p*$.\ v \prec_t u\}$
⟨*proof*⟩

lemma *higher-higher*: *higher* (*higher p u*) *v* = *higher p* (*ord-term-lin.max u v*)
⟨*proof*⟩

lemma *lookup-lower*: *lookup* (*lower p u*) *v* = (*if v* $\prec_t$ *u then lookup p v else 0*)
⟨*proof*⟩

lemma *lookup-lower-when*: *lookup* (*lower p u*) *v* = (*lookup p v when v* $\prec_t$ *u*)
⟨*proof*⟩

lemma *lower-plus*: *lower* (*p* + *q*) *v* = *lower p v* + *lower q v*
⟨*proof*⟩

lemma *lower-uminus* [*simp*]: *lower* (− *p*) *v* = − *lower p v*
⟨*proof*⟩

lemma *lower-minus*: *lower* (*p* − (*q*::- $\Rightarrow_0$ *'b*::*ab-group-add*)) *v* = *lower p v* − *lower q v*
⟨*proof*⟩

lemma *lower-zero* [*simp*]: *lower 0 v = 0*
⟨*proof*⟩

lemma *lower-eq-iff*: *lower p v = lower q v* $\longleftrightarrow$ $(\forall u.\ u \prec_t v \longrightarrow$ *lookup p u = lookup q u*$)$ (**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

lemma *lower-eq-zero-iff*: *lower p v = 0* $\longleftrightarrow$ $(\forall u.\ u \prec_t v \longrightarrow$ *lookup p u = 0*$)$
⟨*proof*⟩

lemma *keys-lower*: *keys* (*lower p v*) = $\{u \in$*keys p*$.\ u \prec_t v\}$
⟨*proof*⟩

lemma *lower-lower*: *lower* (*lower p u*) *v* = *lower p* (*ord-term-lin.min u v*)
⟨*proof*⟩

lemma *lt-higher*:
assumes *v* $\prec_t$ *lt p*
shows *lt* (*higher p v*) = *lt p*

⟨*proof*⟩

lemma *lc-higher*:
  assumes $v \prec_t lt\ p$
  shows $lc\ (higher\ p\ v) = lc\ p$
  ⟨*proof*⟩

lemma *higher-eq-zero-iff'*: $higher\ p\ v = 0 \longleftrightarrow lt\ p \preceq_t v$
  ⟨*proof*⟩

lemma *higher-id-iff*: $higher\ p\ v = p \longleftrightarrow (p = 0 \vee v \prec_t tt\ p)$ (**is** $?L \longleftrightarrow ?R$)
⟨*proof*⟩

lemma *tt-lower*:
  assumes $tt\ p \prec_t v$
  shows $tt\ (lower\ p\ v) = tt\ p$
⟨*proof*⟩

lemma *tc-lower*:
  assumes $tt\ p \prec_t v$
  shows $tc\ (lower\ p\ v) = tc\ p$
  ⟨*proof*⟩

lemma *lt-lower*: $lt\ (lower\ p\ v) \preceq_t lt\ p$
⟨*proof*⟩

lemma *lt-lower-less*:
  assumes $lower\ p\ v \neq 0$
  shows $lt\ (lower\ p\ v) \prec_t v$
  ⟨*proof*⟩

lemma *lt-lower-eq-iff*: $lt\ (lower\ p\ v) = lt\ p \longleftrightarrow (lt\ p = min\text{-}term \vee lt\ p \prec_t v)$ (**is** $?L \longleftrightarrow ?R$)
⟨*proof*⟩

lemma *tt-higher*:
  assumes $v \prec_t lt\ p$
  shows $tt\ p \preceq_t tt\ (higher\ p\ v)$
⟨*proof*⟩

lemma *tt-higher-eq-iff*:
  $tt\ (higher\ p\ v) = tt\ p \longleftrightarrow ((lt\ p \preceq_t v \wedge tt\ p = min\text{-}term) \vee v \prec_t tt\ p)$ (**is** $?L \longleftrightarrow ?R$)
⟨*proof*⟩

lemma *lower-eq-zero-iff'*: $lower\ p\ v = 0 \longleftrightarrow (p = 0 \vee v \preceq_t tt\ p)$
  ⟨*proof*⟩

lemma *lower-id-iff*: $lower\ p\ v = p \longleftrightarrow (p = 0 \vee lt\ p \prec_t v)$ (**is** $?L \longleftrightarrow ?R$)

⟨*proof*⟩

lemma *lower-higher-commute*: *higher* (*lower p s*) *t* = *lower* (*higher p t*) *s*
⟨*proof*⟩

lemma *lt-lower-higher*:
assumes *v* ≺$_t$ *lt* (*lower p u*)
shows *lt* (*lower* (*higher p v*) *u*) = *lt* (*lower p u*)
⟨*proof*⟩

lemma *lc-lower-higher*:
assumes *v* ≺$_t$ *lt* (*lower p u*)
shows *lc* (*lower* (*higher p v*) *u*) = *lc* (*lower p u*)
⟨*proof*⟩

lemma *trailing-monomial-higher*:
assumes *p* ≠ *0*
shows *p* = (*higher p* (*tt p*)) + *monomial* (*tc p*) (*tt p*)
⟨*proof*⟩

lemma *higher-lower-decomp*: *higher p v* + *monomial* (*lookup p v*) *v* + *lower p v* = *p*
⟨*proof*⟩

10.6 *tail*

lemma *lookup-tail*: *lookup* (*tail p*) *v* = (*if v* ≺$_t$ *lt p then lookup p v else 0*)
⟨*proof*⟩

lemma *lookup-tail-when*: *lookup* (*tail p*) *v* = (*lookup p v when v* ≺$_t$ *lt p*)
⟨*proof*⟩

lemma *lookup-tail-2*: *lookup* (*tail p*) *v* = (*if v* = *lt p then 0 else lookup p v*)
⟨*proof*⟩

lemma *leading-monomial-tail*: *p* = *monomial* (*lc p*) (*lt p*) + *tail p* **for** *p*::- ⇒$_0$ *'b*::*comm-monoid-add*
⟨*proof*⟩

lemma *tail-alt*: *tail p* = *except p* {*lt p*}
⟨*proof*⟩

corollary *tail-alt-2*: *tail p* = *p* − *monomial* (*lc p*) (*lt p*)
⟨*proof*⟩

lemma *tail-zero* [*simp*]: *tail 0* = *0*
⟨*proof*⟩

lemma *lt-tail*:

assumes *tail p* $\neq$ *0*
shows *lt* (*tail p*) $\prec_t$ *lt p*
⟨*proof*⟩

lemma *keys-tail*: *keys* (*tail p*) = *keys p* − {*lt p*}
⟨*proof*⟩

lemma *tail-monomial*: *tail* (*monomial c v*) = *0*
⟨*proof*⟩

lemma (**in** *ordered-term*) *mult-scalar-tail-rec-left*:
p ⊙ *q* = *monom-mult* (*punit.lc p*) (*punit.lt p*) *q* + (*punit.tail p*) ⊙ *q*
⟨*proof*⟩

lemma *mult-scalar-tail-rec-right*: *p* ⊙ *q* = *p* ⊙ *monomial* (*lc q*) (*lt q*) + *p* ⊙ *tail q*
⟨*proof*⟩

lemma *lt-tail-max*:
assumes *tail p* $\neq$ *0* **and** *v* $\in$ *keys p* **and** *v* $\prec_t$ *lt p*
shows *v* $\preceq_t$ *lt* (*tail p*)
⟨*proof*⟩

lemma *keys-tail-less-lt*:
assumes *v* $\in$ *keys* (*tail p*)
shows *v* $\prec_t$ *lt p*
⟨*proof*⟩

lemma *tt-tail*:
assumes *tail p* $\neq$ *0*
shows *tt* (*tail p*) = *tt p*
⟨*proof*⟩

lemma *tc-tail*:
assumes *tail p* $\neq$ *0*
shows *tc* (*tail p*) = *tc p*
⟨*proof*⟩

lemma *tt-tail-min*:
assumes *s* $\in$ *keys p*
shows *tt* (*tail p*) $\preceq_t$ *s*
⟨*proof*⟩

lemma *tail-monom-mult*:
tail (*monom-mult c t p*) = *monom-mult* (*c*::*'b*::*semiring-no-zero-divisors*) *t* (*tail p*)
⟨*proof*⟩

lemma *keys-plus-eq-lt-tt-D*:

assumes *keys* $(p + q) = \{lt\ p, tt\ q\}$ **and** $lt\ q \prec_t lt\ p$ **and** $tt\ q \prec_t tt$ $(p::\text{-} \Rightarrow_0 'b::comm\text{-}monoid\text{-}add)$
shows $tail\ p + higher\ q\ (tt\ q) = 0$
⟨*proof*⟩

10.7 Order Relation on Polynomials

definition *ord-strict-p* :: $('t \Rightarrow_0 'b::zero) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow bool$ (**infixl** $\prec_p$ 50) **where**
$p \prec_p q \longleftrightarrow (\exists v.\ lookup\ p\ v = 0 \wedge lookup\ q\ v \neq 0 \wedge (\forall u.\ v \prec_t u \longrightarrow lookup\ p\ u = lookup\ q\ u))$

definition *ord-p* :: $('t \Rightarrow_0 'b::zero) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow bool$ (**infixl** $\preceq_p$ 50) **where**
$ord\text{-}p\ p\ q \equiv (p \prec_p q \vee p = q)$

lemma *ord-strict-pI*:
assumes $lookup\ p\ v = 0$ **and** $lookup\ q\ v \neq 0$ **and** $\bigwedge u.\ v \prec_t u \Longrightarrow lookup\ p\ u = lookup\ q\ u$
shows $p \prec_p q$
⟨*proof*⟩

lemma *ord-strict-pE*:
assumes $p \prec_p q$
obtains v **where** $lookup\ p\ v = 0$ **and** $lookup\ q\ v \neq 0$ **and** $\bigwedge u.\ v \prec_t u \Longrightarrow lookup\ p\ u = lookup\ q\ u$
⟨*proof*⟩

lemma *not-ord-pI*:
assumes $lookup\ p\ v \neq lookup\ q\ v$ **and** $lookup\ p\ v \neq 0$ **and** $\bigwedge u.\ v \prec_t u \Longrightarrow lookup\ p\ u = lookup\ q\ u$
shows $\neg\ p \preceq_p q$
⟨*proof*⟩

corollary *not-ord-strict-pI*:
assumes $lookup\ p\ v \neq lookup\ q\ v$ **and** $lookup\ p\ v \neq 0$ **and** $\bigwedge u.\ v \prec_t u \Longrightarrow lookup\ p\ u = lookup\ q\ u$
shows $\neg\ p \prec_p q$
⟨*proof*⟩

lemma *ord-strict-higher*: $p \prec_p q \longleftrightarrow (\exists v.\ lookup\ p\ v = 0 \wedge lookup\ q\ v \neq 0 \wedge higher\ p\ v = higher\ q\ v)$
⟨*proof*⟩

lemma *ord-strict-p-asymmetric*:
assumes $p \prec_p q$
shows $\neg\ q \prec_p p$
⟨*proof*⟩

lemma *ord-strict-p-irreflexive*: $\neg\ p \prec_p p$

⟨*proof*⟩

lemma *ord-strict-p-transitive*:
assumes $a \prec_p b$ **and** $b \prec_p c$
shows $a \prec_p c$
⟨*proof*⟩

sublocale *order ord-p ord-strict-p*
⟨*proof*⟩

lemma *ord-p-zero-min*: $0 \preceq_p p$
⟨*proof*⟩

lemma *lt-ord-p*:
assumes $lt\ p \prec_t lt\ q$
shows $p \prec_p q$
⟨*proof*⟩

lemma *ord-p-lt*:
assumes $p \preceq_p q$
shows $lt\ p \preceq_t lt\ q$
⟨*proof*⟩

lemma *ord-p-tail*:
assumes $p \neq 0$ **and** $lt\ p = lt\ q$ **and** $p \prec_p q$
shows $tail\ p \prec_p tail\ q$
⟨*proof*⟩

lemma *tail-ord-p*:
assumes $p \neq 0$
shows $tail\ p \prec_p p$
⟨*proof*⟩

lemma *higher-lookup-eq-zero*:
assumes *pt*: $lookup\ p\ v = 0$ **and** *hp*: $higher\ p\ v = 0$ **and** *le*: $q \preceq_p p$
shows $(lookup\ q\ v = 0) \wedge (higher\ q\ v) = 0$
⟨*proof*⟩

lemma *ord-strict-p-recI*:
assumes $lt\ p = lt\ q$ **and** $lc\ p = lc\ q$ **and** *tail*: $tail\ p \prec_p tail\ q$
shows $p \prec_p q$
⟨*proof*⟩

lemma *ord-strict-p-recE1*:
assumes $p \prec_p q$
shows $q \neq 0$
⟨*proof*⟩

lemma *ord-strict-p-recE2*:

assumes $p \neq 0$ **and** $p \prec_p q$ **and** $lt\ p = lt\ q$
shows $lc\ p = lc\ q$
⟨*proof*⟩

lemma *ord-strict-p-rec* [*code*]:
$p \prec_p q =$
$(q \neq 0 \wedge$
$(p = 0 \vee$
$(let\ v1 = lt\ p;\ v2 = lt\ q\ in$
$(v1 \prec_t v2 \vee (v1 = v2 \wedge lookup\ p\ v1 = lookup\ q\ v2 \wedge lower\ p\ v1 \prec_p lower\ q\ v2))$
$)$
$)$
$)$
(**is** $?L = ?R$)
⟨*proof*⟩

lemma *ord-strict-p-monomial-iff*: $p \prec_p monomial\ c\ v \longleftrightarrow (c \neq 0 \wedge (p = 0 \vee lt\ p \prec_t v))$
⟨*proof*⟩

corollary *ord-strict-p-monomial-plus*:
assumes $p \prec_p monomial\ c\ v$ **and** $q \prec_p monomial\ c\ v$
shows $p + q \prec_p monomial\ c\ v$
⟨*proof*⟩

lemma *ord-strict-p-monom-mult*:
assumes $p \prec_p q$ **and** $c \neq (0::'b::semiring\text{-}no\text{-}zero\text{-}divisors)$
shows $monom\text{-}mult\ c\ t\ p \prec_p monom\text{-}mult\ c\ t\ q$
⟨*proof*⟩

lemma *ord-strict-p-plus*:
assumes $p \prec_p q$ **and** $keys\ r \cap keys\ q = \{\}$
shows $p + r \prec_p q + r$
⟨*proof*⟩

lemma *poly-mapping-tail-induct* [*case-names 0 tail*]:
assumes $P\ 0$ **and** $\bigwedge p.\ p \neq 0 \Longrightarrow P\ (tail\ p) \Longrightarrow P\ p$
shows $P\ p$
⟨*proof*⟩

lemma *poly-mapping-neqE*:
assumes $p \neq q$
obtains v **where** $v \in keys\ p \cup keys\ q$ **and** $lookup\ p\ v \neq lookup\ q\ v$
and $\bigwedge u.\ v \prec_t u \Longrightarrow lookup\ p\ u = lookup\ q\ u$
⟨*proof*⟩

10.8 Monomials

lemma *keys-monomial*:
assumes *is-monomial p*
shows *keys p* = {*lt p*}
⟨*proof*⟩

lemma *monomial-eq-itself*:
assumes *is-monomial p*
shows *monomial* (*lc p*) (*lt p*) = *p*
⟨*proof*⟩

lemma *lt-eq-min-term-monomial*:
assumes *lt p* = *min-term*
shows *monomial* (*lc p*) *min-term* = *p*
⟨*proof*⟩

lemma *is-monomial-monomial-ordered*:
assumes *is-monomial p*
obtains *c v* **where** *c* ≠ *0* **and** *lc p* = *c* **and** *lt p* = *v* **and** *p* = *monomial c v*
⟨*proof*⟩

lemma *monomial-plus-not-0*:
assumes *c* ≠ *0* **and** *lt p* ≺$_t$ *v*
shows *monomial c v* + *p* ≠ *0*
⟨*proof*⟩

lemma *lt-monomial-plus*:
assumes *c* ≠ (*0*::*'b*::*comm-monoid-add*) **and** *lt p* ≺$_t$ *v*
shows *lt* (*monomial c v* + *p*) = *v*
⟨*proof*⟩

lemma *lc-monomial-plus*:
assumes *c* ≠ (*0*::*'b*::*comm-monoid-add*) **and** *lt p* ≺$_t$ *v*
shows *lc* (*monomial c v* + *p*) = *c*
⟨*proof*⟩

lemma *tt-monomial-plus*:
assumes *p* ≠ (*0*::- ⇒$_0$ *'b*::*comm-monoid-add*) **and** *lt p* ≺$_t$ *v*
shows *tt* (*monomial c v* + *p*) = *tt p*
⟨*proof*⟩

lemma *tc-monomial-plus*:
assumes *p* ≠ (*0*::- ⇒$_0$ *'b*::*comm-monoid-add*) **and** *lt p* ≺$_t$ *v*
shows *tc* (*monomial c v* + *p*) = *tc p*
⟨*proof*⟩

lemma *tail-monomial-plus*:
assumes *c* ≠ (*0*::*'b*::*comm-monoid-add*) **and** *lt p* ≺$_t$ *v*
shows *tail* (*monomial c v* + *p*) = *p* (**is** *tail ?q* = -)

⟨*proof*⟩

10.9 Lists of Keys

In algorithms one very often needs to compute the sorted list of all terms appearing in a list of polynomials.

definition *pps-to-list* :: *'t set* ⇒ *'t list* **where**
pps-to-list S = *rev* (*ord-term-lin.sorted-list-of-set S*)

definition *keys-to-list* :: (*'t* ⇒$_0$ *'b*::*zero*) ⇒ *'t list*
where *keys-to-list p* = *pps-to-list* (*keys p*)

definition *Keys-to-list* :: (*'t* ⇒$_0$ *'b*::*zero*) *list* ⇒ *'t list*
where *Keys-to-list ps* = *fold* (λ*p ts*. *merge-wrt* (≻$_t$) (*keys-to-list p*) *ts*) *ps* []

Function *pps-to-list* turns finite sets of terms into sorted lists, where the lists are sorted descending (i. e. greater elements come before smaller ones).

lemma *distinct-pps-to-list*: *distinct* (*pps-to-list S*)
⟨*proof*⟩

lemma *set-pps-to-list*:
assumes *finite S*
shows *set* (*pps-to-list S*) = *S*
⟨*proof*⟩

lemma *length-pps-to-list*: *length* (*pps-to-list S*) = *card S*
⟨*proof*⟩

lemma *pps-to-list-sorted-wrt*: *sorted-wrt* (≻$_t$) (*pps-to-list S*)
⟨*proof*⟩

lemma *pps-to-list-nth-leI*:
assumes *j* ≤ *i* **and** *i* < *card S*
shows (*pps-to-list S*) ! *i* ⪯$_t$ (*pps-to-list S*) ! *j*
⟨*proof*⟩

lemma *pps-to-list-nth-lessI*:
assumes *j* < *i* **and** *i* < *card S*
shows (*pps-to-list S*) ! *i* ≺$_t$ (*pps-to-list S*) ! *j*
⟨*proof*⟩

lemma *pps-to-list-nth-leD*:
assumes (*pps-to-list S*) ! *i* ⪯$_t$ (*pps-to-list S*) ! *j* **and** *j* < *card S*
shows *j* ≤ *i*
⟨*proof*⟩

lemma *pps-to-list-nth-lessD*:
assumes (*pps-to-list S*) ! *i* ≺$_t$ (*pps-to-list S*) ! *j* **and** *j* < *card S*
shows *j* < *i*

⟨*proof*⟩

lemma *set-keys-to-list*: *set* (*keys-to-list p*) = *keys p*
⟨*proof*⟩

lemma *length-keys-to-list*: *length* (*keys-to-list p*) = *card* (*keys p*)
⟨*proof*⟩

lemma *keys-to-list-zero* [*simp*]: *keys-to-list 0* = []
⟨*proof*⟩

lemma *Keys-to-list-Nil* [*simp*]: *Keys-to-list* [] = []
⟨*proof*⟩

lemma *set-Keys-to-list*: *set* (*Keys-to-list ps*) = *Keys* (*set ps*)
⟨*proof*⟩

lemma *Keys-to-list-sorted-wrt-aux*:
assumes *sorted-wrt* (≻$_t$) *ts*
shows *sorted-wrt* (≻$_t$) (*fold* (λ*p ts*. *merge-wrt* (≻$_t$) (*keys-to-list p*) *ts*) *ps ts*)
⟨*proof*⟩

corollary *Keys-to-list-sorted-wrt*: *sorted-wrt* (≻$_t$) (*Keys-to-list ps*)
⟨*proof*⟩

corollary *distinct-Keys-to-list*: *distinct* (*Keys-to-list ps*)
⟨*proof*⟩

lemma *length-Keys-to-list*: *length* (*Keys-to-list ps*) = *card* (*Keys* (*set ps*))
⟨*proof*⟩

lemma *Keys-to-list-eq-pps-to-list*: *Keys-to-list ps* = *pps-to-list* (*Keys* (*set ps*))
⟨*proof*⟩

10.10 Multiplication

lemma *in-keys-mult-scalar-le*:
assumes *v* ∈ *keys* (*p* ⊙ *q*)
shows *v* ⪯$_t$ *punit.lt p* ⊕ *lt q*
⟨*proof*⟩

lemma *in-keys-mult-scalar-ge*:
assumes *v* ∈ *keys* (*p* ⊙ *q*)
shows *punit.tt p* ⊕ *tt q* ⪯$_t$ *v*
⟨*proof*⟩

lemma (**in** *ordered-term*) *lookup-mult-scalar-lt-lt*:
lookup (*p* ⊙ *q*) (*punit.lt p* ⊕ *lt q*) = *punit.lc p* ∗ *lc q*
⟨*proof*⟩

lemma *lookup-mult-scalar-tt-tt*: *lookup* $(p \odot q)$ $(punit.tt\ p \oplus tt\ q) = punit.tc\ p *$
$tc\ q$
⟨*proof*⟩

lemma *lt-mult-scalar*:
assumes $p \neq 0$ **and** $q \neq (0::'t \Rightarrow_0 'b::semiring\text{-}no\text{-}zero\text{-}divisors)$
shows $lt\ (p \odot q) = punit.lt\ p \oplus lt\ q$
⟨*proof*⟩

lemma *tt-mult-scalar*:
assumes $p \neq 0$ **and** $q \neq (0::'t \Rightarrow_0 'b::semiring\text{-}no\text{-}zero\text{-}divisors)$
shows $tt\ (p \odot q) = punit.tt\ p \oplus tt\ q$
⟨*proof*⟩

lemma *lc-mult-scalar*: $lc\ (p \odot q) = punit.lc\ p * lc\ (q::'t \Rightarrow_0 'b::semiring\text{-}no\text{-}zero\text{-}divisors)$
⟨*proof*⟩

lemma *tc-mult-scalar*: $tc\ (p \odot q) = punit.tc\ p * tc\ (q::'t \Rightarrow_0 'b::semiring\text{-}no\text{-}zero\text{-}divisors)$
⟨*proof*⟩

lemma *mult-scalar-not-zero*:
assumes $p \neq 0$ **and** $q \neq (0::'t \Rightarrow_0 'b::semiring\text{-}no\text{-}zero\text{-}divisors)$
shows $p \odot q \neq 0$
⟨*proof*⟩

end

context *ordered-powerprod*
begin

lemmas *in-keys-times-le* = *punit.in-keys-mult-scalar-le*[*simplified*]
lemmas *in-keys-times-ge* = *punit.in-keys-mult-scalar-ge*[*simplified*]
lemmas *lookup-times-lp-lp* = *punit.lookup-mult-scalar-lt-lt*[*simplified*]
lemmas *lookup-times-tp-tp* = *punit.lookup-mult-scalar-tt-tt*[*simplified*]
lemmas *lookup-times-monomial-right-plus* = *punit.lookup-mult-scalar-monomial-right-plus*[*simplified*]
lemmas *lookup-times-monomial-right* = *punit.lookup-mult-scalar-monomial-right*[*simplified*]
lemmas *lp-times* = *punit.lt-mult-scalar*[*simplified*]
lemmas *tp-times* = *punit.tt-mult-scalar*[*simplified*]
lemmas *lc-times* = *punit.lc-mult-scalar*[*simplified*]
lemmas *tc-times* = *punit.tc-mult-scalar*[*simplified*]
lemmas *times-not-zero* = *punit.mult-scalar-not-zero*[*simplified*]
lemmas *times-tail-rec-left* = *punit.mult-scalar-tail-rec-left*[*simplified*]
lemmas *times-tail-rec-right* = *punit.mult-scalar-tail-rec-right*[*simplified*]
lemmas *punit-in-keys-monom-mult-le* = *punit.in-keys-monom-mult-le*[*simplified*]
lemmas *punit-in-keys-monom-mult-ge* = *punit.in-keys-monom-mult-ge*[*simplified*]
lemmas *lp-monom-mult* = *punit.lt-monom-mult*[*simplified*]
lemmas *tp-monom-mult* = *punit.tt-monom-mult*[*simplified*]

end

10.11 *dgrad-p-set* and *dgrad-p-set-le*

locale *gd-term* =
  ordered-term pair-of-term term-of-pair ord ord-strict ord-term ord-term-strict
  for *pair-of-term*::*'t* ⇒ (*'a*::*graded-dickson-powerprod* × *'k*::{*the-min*,*wellorder*})
    and *term-of-pair*::(*'a* × *'k*) ⇒ *'t*
    and *ord*::*'a* ⇒ *'a* ⇒ *bool* (**infixl** ⪯ *50*)
    and *ord-strict* (**infixl** ≺ *50*)
    and *ord-term*::*'t* ⇒ *'t* ⇒ *bool* (**infixl** ⪯$_t$ *50*)
    and *ord-term-strict*::*'t* ⇒ *'t* ⇒ *bool* (**infixl** ≺$_t$ *50*)
begin

sublocale *gd-powerprod* ⟨*proof*⟩

lemma *adds-term-antisym*:
  assumes *u adds*$_t$ *v* **and** *v adds*$_t$ *u*
  shows *u* = *v*
  ⟨*proof*⟩

definition *dgrad-p-set* :: (*'a* ⇒ *nat*) ⇒ *nat* ⇒ (*'t* ⇒$_0$ *'b*::*zero*) *set*
  where *dgrad-p-set d m* = {*p*. *pp-of-term* ‘ *keys p* ⊆ *dgrad-set d m*}

definition *dgrad-p-set-le* :: (*'a* ⇒ *nat*) ⇒ ((*'t* ⇒$_0$ *'b*) *set*) ⇒ ((*'t* ⇒$_0$ *'b*::*zero*) *set*) ⇒ *bool*
  where *dgrad-p-set-le d F G* ⟷ (*dgrad-set-le d* (*pp-of-term* ‘ *Keys F*) (*pp-of-term* ‘ *Keys G*))

lemma *in-dgrad-p-set-iff*: *p* ∈ *dgrad-p-set d m* ⟷ (∀ *v*∈*keys p*. *d* (*pp-of-term v*) ≤ *m*)
  ⟨*proof*⟩

lemma *dgrad-p-setI* [*intro*]:
  assumes ⋀*v*. *v* ∈ *keys p* ⟹ *d* (*pp-of-term v*) ≤ *m*
  shows *p* ∈ *dgrad-p-set d m*
  ⟨*proof*⟩

lemma *dgrad-p-setD*:
  assumes *p* ∈ *dgrad-p-set d m* **and** *v* ∈ *keys p*
  shows *d* (*pp-of-term v*) ≤ *m*
  ⟨*proof*⟩

lemma *zero-in-dgrad-p-set*: *0* ∈ *dgrad-p-set d m*
  ⟨*proof*⟩

lemma *dgrad-p-set-zero* [*simp*]: *dgrad-p-set* (λ-. *0*) *m* = *UNIV*
  ⟨*proof*⟩

lemma *subset-dgrad-p-set-zero*: $F \subseteq$ *dgrad-p-set* (λ-. *0*) *m*
⟨*proof*⟩

lemma *dgrad-p-set-subset*:
assumes $m \leq n$
shows *dgrad-p-set d m* $\subseteq$ *dgrad-p-set d n*
⟨*proof*⟩

lemma *dgrad-p-setD-lp*:
assumes $p \in$ *dgrad-p-set d m* **and** $p \neq 0$
shows *d* (*lp p*) $\leq m$
⟨*proof*⟩

lemma *dgrad-p-set-exhaust-expl*:
assumes *finite F*
shows $F \subseteq$ *dgrad-p-set d* (*Max* (*d* ‘ *pp-of-term* ‘ *Keys F*))
⟨*proof*⟩

lemma *dgrad-p-set-exhaust*:
assumes *finite F*
obtains *m* **where** $F \subseteq$ *dgrad-p-set d m*
⟨*proof*⟩

lemma *dgrad-p-set-insert*:
assumes $F \subseteq$ *dgrad-p-set d m*
obtains *n* **where** $m \leq n$ **and** $f \in$ *dgrad-p-set d n* **and** $F \subseteq$ *dgrad-p-set d n*
⟨*proof*⟩

lemma *dgrad-p-set-leI*:
assumes $\bigwedge f.\ f \in F \Longrightarrow$ *dgrad-p-set-le d* $\{f\}$ *G*
shows *dgrad-p-set-le d F G*
⟨*proof*⟩

lemma *dgrad-p-set-le-trans* [*trans*]:
assumes *dgrad-p-set-le d F G* **and** *dgrad-p-set-le d G H*
shows *dgrad-p-set-le d F H*
⟨*proof*⟩

lemma *dgrad-p-set-le-subset*:
assumes $F \subseteq G$
shows *dgrad-p-set-le d F G*
⟨*proof*⟩

lemma *dgrad-p-set-leI-insert-keys*:
assumes *dgrad-p-set-le d F G* **and** *dgrad-set-le d* (*pp-of-term* ‘ *keys f*) (*pp-of-term* ‘ *Keys G*)
shows *dgrad-p-set-le d* (*insert f F*) *G*
⟨*proof*⟩

lemma *dgrad-p-set-leI-insert*:
assumes *dgrad-p-set-le d F G* **and** *dgrad-p-set-le d* {*f*} *G*
shows *dgrad-p-set-le d* (*insert f F*) *G*
⟨*proof*⟩

lemma *dgrad-p-set-leI-Un*:
assumes *dgrad-p-set-le d F1 G* **and** *dgrad-p-set-le d F2 G*
shows *dgrad-p-set-le d* (*F1* ∪ *F2*) *G*
⟨*proof*⟩

lemma *dgrad-p-set-le-dgrad-p-set*:
assumes *dgrad-p-set-le d F G* **and** *G* ⊆ *dgrad-p-set d m*
shows *F* ⊆ *dgrad-p-set d m*
⟨*proof*⟩

lemma *dgrad-p-set-le-except*: *dgrad-p-set-le d* {*except p S*} {*p*}
⟨*proof*⟩

lemma *dgrad-p-set-le-tail*: *dgrad-p-set-le d* {*tail p*} {*p*}
⟨*proof*⟩

lemma *dgrad-p-set-le-plus*: *dgrad-p-set-le d* {*p* + *q*} {*p*, *q*}
⟨*proof*⟩

lemma *dgrad-p-set-le-uminus*: *dgrad-p-set-le d* {−*p*} {*p*}
⟨*proof*⟩

lemma *dgrad-p-set-le-minus*: *dgrad-p-set-le d* {*p* − *q*} {*p*, *q*}
⟨*proof*⟩

lemma *dgrad-set-le-monom-mult*:
assumes *dickson-grading d*
shows *dgrad-set-le d* (*pp-of-term* ‘ *keys* (*monom-mult c t p*)) (*insert t* (*pp-of-term* ‘ *keys p*))
⟨*proof*⟩

lemma *dgrad-p-set-closed-plus*:
assumes *p* ∈ *dgrad-p-set d m* **and** *q* ∈ *dgrad-p-set d m*
shows *p* + *q* ∈ *dgrad-p-set d m*
⟨*proof*⟩

lemma *dgrad-p-set-closed-uminus*:
assumes *p* ∈ *dgrad-p-set d m*
shows −*p* ∈ *dgrad-p-set d m*
⟨*proof*⟩

lemma *dgrad-p-set-closed-minus*:
assumes *p* ∈ *dgrad-p-set d m* **and** *q* ∈ *dgrad-p-set d m*
shows *p* − *q* ∈ *dgrad-p-set d m*

⟨*proof*⟩

lemma *dgrad-p-set-closed-monom-mult*:
assumes *dickson-grading d* **and** *d t* ≤ *m* **and** *p* ∈ *dgrad-p-set d m*
shows *monom-mult c t p* ∈ *dgrad-p-set d m*
⟨*proof*⟩

lemma *dgrad-p-set-closed-monom-mult-zero*:
assumes *p* ∈ *dgrad-p-set d m*
shows *monom-mult c 0 p* ∈ *dgrad-p-set d m*
⟨*proof*⟩

lemma *dgrad-p-set-closed-except*:
assumes *p* ∈ *dgrad-p-set d m*
shows *except p S* ∈ *dgrad-p-set d m*
⟨*proof*⟩

lemma *dgrad-p-set-closed-tail*:
assumes *p* ∈ *dgrad-p-set d m*
shows *tail p* ∈ *dgrad-p-set d m*
⟨*proof*⟩

10.12 Dickson's Lemma for Sequences of Terms

lemma *Dickson-term*:
assumes *dickson-grading d* **and** *finite K*
shows *almost-full-on* (*adds*$_t$) {*t*. *pp-of-term t* ∈ *dgrad-set d m* ∧ *component-of-term t* ∈ *K*}
(**is** *almost-full-on - ?A*)
⟨*proof*⟩

corollary *Dickson-termE*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ‘ *range* (*f*::*nat* ⇒ *'t*))
and *pp-of-term* ‘ *range f* ⊆ *dgrad-set d m*
obtains *i j* **where** *i* < *j* **and** *f i adds*$_t$ *f j*
⟨*proof*⟩

lemma *ex-finite-adds-term*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ‘ *S*) **and** *pp-of-term* ‘ *S* ⊆ *dgrad-set d m*
obtains *T* **where** *finite T* **and** *T* ⊆ *S* **and** ⋀*s*. *s* ∈ *S* ⟹ (∃ *t*∈*T*. *t adds*$_t$ *s*)
⟨*proof*⟩

10.13 Well-foundedness

definition *dickson-less-v* :: (*'a* ⇒ *nat*) ⇒ *nat* ⇒ *'t* ⇒ *'t* ⇒ *bool*
where *dickson-less-v d m v u* ⟷ (*d* (*pp-of-term v*) ≤ *m* ∧ *d* (*pp-of-term u*) ≤ *m* ∧ *v* ≺$_t$ *u*)

definition *dickson-less-p* :: $('a \Rightarrow nat) \Rightarrow nat \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b::zero) \Rightarrow bool$
where *dickson-less-p d m p q* $\longleftrightarrow$ $(\{p, q\} \subseteq$ *dgrad-p-set d m* $\wedge$ $p \prec_p q)$

lemma *dickson-less-vI*:
assumes *d* (*pp-of-term v*) $\leq$ *m* **and** *d* (*pp-of-term u*) $\leq$ *m* **and** $v \prec_t u$
shows *dickson-less-v d m v u*
⟨*proof*⟩

lemma *dickson-less-vD1*:
assumes *dickson-less-v d m v u*
shows *d* (*pp-of-term v*) $\leq$ *m*
⟨*proof*⟩

lemma *dickson-less-vD2*:
assumes *dickson-less-v d m v u*
shows *d* (*pp-of-term u*) $\leq$ *m*
⟨*proof*⟩

lemma *dickson-less-vD3*:
assumes *dickson-less-v d m v u*
shows $v \prec_t u$
⟨*proof*⟩

lemma *dickson-less-v-irrefl*: $\neg$ *dickson-less-v d m v v*
⟨*proof*⟩

lemma *dickson-less-v-trans*:
assumes *dickson-less-v d m v u* **and** *dickson-less-v d m u w*
shows *dickson-less-v d m v w*
⟨*proof*⟩

lemma *wf-dickson-less-v-aux1*:
assumes *dickson-grading d* **and** $\bigwedge i$::*nat*. *dickson-less-v d m* (*seq* (*Suc i*)) (*seq i*)
obtains *i* **where** $\bigwedge j.\ j > i \Longrightarrow$ *component-of-term* (*seq j*) $<$ *component-of-term* (*seq i*)
⟨*proof*⟩

lemma *wf-dickson-less-v-aux2*:
assumes *dickson-grading d* **and** $\bigwedge i$::*nat*. *dickson-less-v d m* (*seq* (*Suc i*)) (*seq i*)
and $\bigwedge i$::*nat*. *component-of-term* (*seq i*) $< k$
shows *thesis*
⟨*proof*⟩

lemma *wf-dickson-less-v*:
assumes *dickson-grading d*
shows *wfP* (*dickson-less-v d m*)
⟨*proof*⟩

lemma *dickson-less-v-zero*: *dickson-less-v* $(\lambda\text{-}.\ 0)\ m = (\prec_t)$
⟨*proof*⟩

lemma *dickson-less-pI*:
assumes $p \in$ *dgrad-p-set* $d\ m$ **and** $q \in$ *dgrad-p-set* $d\ m$ **and** $p \prec_p q$
shows *dickson-less-p* $d\ m\ p\ q$
⟨*proof*⟩

lemma *dickson-less-pD1*:
assumes *dickson-less-p* $d\ m\ p\ q$
shows $p \in$ *dgrad-p-set* $d\ m$
⟨*proof*⟩

lemma *dickson-less-pD2*:
assumes *dickson-less-p* $d\ m\ p\ q$
shows $q \in$ *dgrad-p-set* $d\ m$
⟨*proof*⟩

lemma *dickson-less-pD3*:
assumes *dickson-less-p* $d\ m\ p\ q$
shows $p \prec_p q$
⟨*proof*⟩

lemma *dickson-less-p-irrefl*: $\neg$ *dickson-less-p* $d\ m\ p\ p$
⟨*proof*⟩

lemma *dickson-less-p-trans*:
assumes *dickson-less-p* $d\ m\ p\ q$ **and** *dickson-less-p* $d\ m\ q\ r$
shows *dickson-less-p* $d\ m\ p\ r$
⟨*proof*⟩

lemma *dickson-less-p-mono*:
assumes *dickson-less-p* $d\ m\ p\ q$ **and** $m \leq n$
shows *dickson-less-p* $d\ n\ p\ q$
⟨*proof*⟩

lemma *dickson-less-p-zero*: *dickson-less-p* $(\lambda\text{-}.\ 0)\ m = (\prec_p)$
⟨*proof*⟩

lemma *wf-dickson-less-p-aux*:
assumes *dickson-grading* d
assumes $x \in Q$ **and** $\forall\, y{\in}Q.\ y \neq 0 \longrightarrow (y \in$ *dgrad-p-set* $d\ m \wedge$ *dickson-less-v* $d\ m\ (lt\ y)\ u)$
shows $\exists\, p{\in}Q.\ (\forall\, q{\in}Q.\ \neg$ *dickson-less-p* $d\ m\ q\ p)$
⟨*proof*⟩

theorem *wf-dickson-less-p*:
assumes *dickson-grading* d
shows *wfP* (*dickson-less-p* $d\ m$)

⟨*proof*⟩

corollary *ord-p-minimum-dgrad-p-set*:
assumes *dickson-grading d* **and** $x \in Q$ **and** $Q \subseteq$ *dgrad-p-set d m*
obtains q **where** $q \in Q$ **and** $\bigwedge y.\ y \prec_p q \Longrightarrow y \notin Q$
⟨*proof*⟩

lemma *ord-term-minimum-dgrad-set*:
assumes *dickson-grading d* **and** $v \in V$ **and** *pp-of-term* ' $V \subseteq$ *dgrad-set d m*
obtains u **where** $u \in V$ **and** $\bigwedge w.\ w \prec_t u \Longrightarrow w \notin V$
⟨*proof*⟩

end

10.14 More Interpretations

context *gd-powerprod*
begin

sublocale *punit*: *gd-term to-pair-unit fst* $(\preceq)$ $(\prec)$ $(\preceq)$ $(\prec)$ ⟨*proof*⟩

end

locale *od-term* =
ordered-term pair-of-term term-of-pair ord ord-strict ord-term ord-term-strict
for *pair-of-term*::$'t \Rightarrow$ ($'a$::*dickson-powerprod* $\times$ $'k$::{*the-min*,*wellorder*})
and *term-of-pair*::$('a \times 'k) \Rightarrow 't$
and *ord*::$'a \Rightarrow 'a \Rightarrow$ *bool* (**infixl** $\preceq$ *50*)
and *ord-strict* (**infixl** $\prec$ *50*)
and *ord-term*::$'t \Rightarrow 't \Rightarrow$ *bool* (**infixl** $\preceq_t$ *50*)
and *ord-term-strict*::$'t \Rightarrow 't \Rightarrow$ *bool* (**infixl** $\prec_t$ *50*)
begin

sublocale *gd-term* ⟨*proof*⟩

lemma *ord-p-wf*: *wfP* $(\prec_p)$
⟨*proof*⟩

end

end

theory *Poly-Mapping-Finite-Map*
imports
More-MPoly-Type
HOL−Library.Finite-Map
begin

10.15 TODO: move!

lemma *fmdom'-fmap-of-list*: *fmdom'* (*fmap-of-list xs*) = *set* (*map fst xs*)
⟨*proof*⟩

In this theory, type $'a \Rightarrow_0 'b$ is represented as association lists. Code equations are proved in order actually perform computations (addition, multiplication, etc.).

10.16 Utilities

instantiation *poly-mapping* :: (*type*, {*equal*, *zero*}) *equal*
begin
definition *equal-poly-mapping*::('*a*, '*b*) *poly-mapping* ⇒ ('*a*, '*b*) *poly-mapping* ⇒ *bool* **where**
equal-poly-mapping p q ≡ (∀ *t*. *lookup p t* = *lookup q t*)

instance ⟨*proof*⟩
end

definition *clearjunk0 m* = *fmfilter* (λ*k*. *fmlookup m k* ≠ *Some 0*) *m*

definition *fmlookup-default d m x* = (*case fmlookup m x of Some v* ⇒ *v* | *None* ⇒ *d*)
abbreviation *lookup0* ≡ *fmlookup-default 0*

lemma *fmlookup-default-fmmap*:
fmlookup-default d (*fmmap f M*) *x* = (*if x* ∈ *fmdom' M then f* (*fmlookup-default d M x*) *else d*)
⟨*proof*⟩

lemma *fmlookup-default-fmmap-keys*: *fmlookup-default d* (*fmmap-keys f M*) *x* =
(*if x* ∈ *fmdom' M then f x* (*fmlookup-default d M x*) *else d*)
⟨*proof*⟩

lemma *fmlookup-default-add*[*simp*]:
fmlookup-default d (*m* +++$_f$ *n*) *x* =
(*if x* |∈| *fmdom n then the* (*fmlookup n x*)
else fmlookup-default d m x)
⟨*proof*⟩

lemma *fmlookup-default-if*[*simp*]:
fmlookup ys a = *Some r* ⟹ *fmlookup-default d ys a* = *r*
fmlookup ys a = *None* ⟹ *fmlookup-default d ys a* = *d*
⟨*proof*⟩

lemma *finite-lookup-default*:
finite {*x*. *fmlookup-default d xs x* ≠ *d*}
⟨*proof*⟩

lemma *lookup0-clearjunk0*: *lookup0 xs s* = *lookup0* (*clearjunk0 xs*) *s*
⟨*proof*⟩

lemma *clearjunk0-nonzero*:
assumes *t* ∈ *fmdom′* (*clearjunk0 xs*)
shows *fmlookup xs t* ≠ *Some 0*
⟨*proof*⟩

lemma *clearjunk0-map-of-SomeD*:
assumes *a1*: *fmlookup xs t* = *Some c* **and** *c* ≠ *0*
shows *t* ∈ *fmdom′* (*clearjunk0 xs*)
⟨*proof*⟩

10.17 Implementation of Polynomial Mappings as Association Lists

lift-definition *Pm-fmap*::('*a*, '*b*::*zero*) *fmap* ⇒ '*a* ⇒$_0$ '*b* **is** *lookup0*
⟨*proof*⟩

lemmas [*simp*] = *Pm-fmap.rep-eq*

code-datatype *Pm-fmap*

lemma *PM-clearjunk0-cong*:
Pm-fmap (*clearjunk0 xs*) = *Pm-fmap xs*
⟨*proof*⟩

lemma *PM-all-2*:
assumes *P 0 0*
shows (∀ *x*. *P* (*lookup* (*Pm-fmap xs*) *x*) (*lookup* (*Pm-fmap ys*) *x*)) =
fmpred (λ*k v*. *P* (*lookup0 xs k*) (*lookup0 ys k*)) (*xs* ++$_f$ *ys*)
⟨*proof*⟩

lemma *compute-keys-pp*[*code*]: *keys* (*Pm-fmap xs*) = *fmdom′* (*clearjunk0 xs*)
⟨*proof*⟩

lemma *compute-zero-pp*[*code*]: *0* = *Pm-fmap fmempty*
⟨*proof*⟩

lemma *compute-plus-pp* [*code*]:
Pm-fmap xs + *Pm-fmap ys* = *Pm-fmap* (*clearjunk0* (*fmmap-keys* (λ*k v*. *lookup0 xs k* + *lookup0 ys k*) (*xs* ++$_f$ *ys*)))
⟨*proof*⟩

lemma *compute-lookup-pp*[*code*]:
lookup (*Pm-fmap xs*) *x* = *lookup0 xs x*
⟨*proof*⟩

lemma *compute-minus-pp* [*code*]:

Pm-fmap xs − Pm-fmap ys = Pm-fmap (clearjunk0 (fmmap-keys (λk v. lookup0 xs k − lookup0 ys k) (xs ++$_f$ ys)))
⟨*proof*⟩

lemma *compute-uminus-pp*[*code*]:
− Pm-fmap ys = Pm-fmap (fmmap-keys (λk v. − lookup0 ys k) ys)
⟨*proof*⟩

lemma *compute-equal-pp*[*code*]:
equal-class.equal (Pm-fmap xs) (Pm-fmap ys) = fmpred (λk v. lookup0 xs k = lookup0 ys k) (xs ++$_f$ ys)
⟨*proof*⟩

lemma *compute-map-pp*[*code*]:
Poly-Mapping.map f (Pm-fmap xs) = Pm-fmap (fmmap (λx. f x when x ≠ 0) xs)
⟨*proof*⟩

lemma *fmran'-fmfilter-eq*: *fmran' (fmfilter p fm) = {y | y. ∃ x ∈ fmdom' fm. p x ∧ fmlookup fm x = Some y}*
⟨*proof*⟩

lemma *compute-range-pp*[*code*]:
Poly-Mapping.range (Pm-fmap xs) = fmran' (clearjunk0 xs)
⟨*proof*⟩

10.17.1 Constructors

definition *sparse*$_0$ *xs = Pm-fmap (fmap-of-list xs)* — sparse representation
definition *dense*$_0$ *xs = Pm-fmap (fmap-of-list (zip [0..<length xs] xs))* — dense representation

lemma *compute-single*[*code*]: *Poly-Mapping.single k v = sparse*$_0$ *[(k, v)]*
⟨*proof*⟩

end

11 Executable Representation of Polynomial Mappings as Association Lists

theory *MPoly-Type-Class-FMap*
imports
MPoly-Type-Class-Ordered
Poly-Mapping-Finite-Map
begin

In this theory, (type class) multivariate polynomials of type $'a \Rightarrow_0 'b$ are represented as association lists.

It is important to note that theory *MPoly-Type-Class-OAlist*, which represents polynomials as *ordered* associative lists, is much better suited for doing actual computations. This theory is only included for being able to compare the two representations in terms of efficiency.

11.1 Power Products

lemma *compute-lcs-pp*[*code*]:
lcs (*Pm-fmap xs*) (*Pm-fmap ys*) =
Pm-fmap (*fmmap-keys* (λ*k v*. *Orderings.max* (*lookup0 xs k*) (*lookup0 ys k*)) (*xs* ++$_f$ *ys*))
⟨*proof*⟩

lemma *compute-deg-pp*[*code*]:
deg-pm (*Pm-fmap xs*) = *sum* (*the o fmlookup xs*) (*fmdom′ xs*)
⟨*proof*⟩

definition *adds-pp-add-linorder* :: ('*b* ⇒$_0$ '*a*::*add-linorder*) ⇒ - ⇒ *bool*
where [*code-abbrev*]: *adds-pp-add-linorder* = (*adds*)

lemma *compute-adds-pp*[*code*]:
adds-pp-add-linorder (*Pm-fmap xs*) (*Pm-fmap ys*) =
(*fmpred* (λ*k v*. *lookup0 xs k* ≤ *lookup0 ys k*) (*xs* ++$_f$ *ys*))
for *xs ys*::('*a*, '*b*::*add-linorder-min*) *fmap*
⟨*proof*⟩

Computing *lex* as below is certainly not the most efficient way, but it works.

lemma *lex-pm-iff*: *lex-pm s t* = (∀ *x*. *lookup s x* ≤ *lookup t x* ∨ (∃ *y*<*x*. *lookup s y* ≠ *lookup t y*))
⟨*proof*⟩

lemma *compute-lex-pp*[*code*]:
(*lex-pm* (*Pm-fmap xs*) (*Pm-fmap* (*ys*::(-, -::*ordered-comm-monoid-add*) *fmap*))) =
(*let zs* = *xs* ++$_f$ *ys in*
fmpred (λ*x v*.
lookup0 xs x ≤ *lookup0 ys x* ∨
¬ *fmpred* (λ*y w*. *y* ≥ *x* ∨ *lookup0 xs y* = *lookup0 ys y*) *zs*) *zs*
)
⟨*proof*⟩

lemma *compute-dord-pp*[*code*]:
(*dord-pm ord* (*Pm-fmap xs*) (*Pm-fmap* (*ys*::('*a*::*wellorder* , '*b*::*ordered-comm-monoid-add*) *fmap*))) =
(*let dx* = *deg-pm* (*Pm-fmap xs*) *in let dy* = *deg-pm* (*Pm-fmap ys*) *in*
dx < *dy* ∨ (*dx* = *dy* ∧ *ord* (*Pm-fmap xs*) (*Pm-fmap ys*))
)
⟨*proof*⟩

11.1.1 Computations

experiment begin

abbreviation $X \equiv 0::nat$
abbreviation $Y \equiv 1::nat$
abbreviation $Z \equiv 2::nat$

lemma
$sparse_0\ [(X,\ 2::nat),\ (Z,\ 7)] + sparse_0\ [(Y,\ 3),\ (Z,\ 2)] = sparse_0\ [(X,\ 2),\ (Z,\ 9),\ (Y,\ 3)]$
$dense_0\ [2,\ 0,\ 7::nat] + dense_0\ [0,\ 3,\ 2] = dense_0\ [2,\ 3,\ 9]$
⟨*proof*⟩

lemma
$sparse_0\ [(X,\ 2::nat),\ (Z,\ 7)] - sparse_0\ [(X,\ 2),\ (Z,\ 2)] = sparse_0\ [(Z,\ 5)]$
⟨*proof*⟩

lemma
$lcs\ (sparse_0\ [(X,\ 2::nat),\ (Y,\ 1),\ (Z,\ 7)])\ (sparse_0\ [(Y,\ 3),\ (Z,\ 2)]) = sparse_0\ [(X,\ 2),\ (Y,\ 3),\ (Z,\ 7)]$
⟨*proof*⟩

lemma
$(sparse_0\ [(X,\ 2::nat),\ (Z,\ 1)])\ adds\ (sparse_0\ [(X,\ 3),\ (Y,\ 2),\ (Z,\ 1)])$
⟨*proof*⟩

lemma
$lookup\ (sparse_0\ [(X,\ 2::nat),\ (Z,\ 3)])\ X = 2$
⟨*proof*⟩

lemma
$deg\text{-}pm\ (sparse_0\ [(X,\ 2::nat),\ (Y,\ 1),\ (Z,\ 3),\ (X,\ 1)]) = 6$
⟨*proof*⟩

lemma
$lex\text{-}pm\ (sparse_0\ [(X,\ 2::nat),\ (Y,\ 1),\ (Z,\ 3)])\ (sparse_0\ [(X,\ 4)])$
⟨*proof*⟩

lemma
$lex\text{-}pm\ (sparse_0\ [(X,\ 2::nat),\ (Y,\ 1),\ (Z,\ 3)])\ (sparse_0\ [(X,\ 4)])$
⟨*proof*⟩

lemma
$\neg\ (dlex\text{-}pm\ (sparse_0\ [(X,\ 2::nat),\ (Y,\ 1),\ (Z,\ 3)])\ (sparse_0\ [(X,\ 4)]))$
⟨*proof*⟩

lemma
$dlex\text{-}pm\ (sparse_0\ [(X,\ 2::nat),\ (Y,\ 1),\ (Z,\ 2)])\ (sparse_0\ [(X,\ 5)])$
⟨*proof*⟩

lemma
¬ (*drlex-pm* (*sparse*$_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *2*)]) (*sparse*$_0$ [(*X*, *5*)]))
⟨*proof*⟩

end

11.2 Implementation of Multivariate Polynomials as Association Lists

11.2.1 Unordered Power-Products

lemma *compute-monomial* [*code*]:
monomial c t = (*if c* = *0 then 0 else sparse*$_0$ [(*t*, *c*)])
⟨*proof*⟩

lemma *compute-one-poly-mapping* [*code*]: *1* = *sparse*$_0$ [(*0*, *1*)]
⟨*proof*⟩

lemma *compute-except-poly-mapping* [*code*]:
except (*Pm-fmap xs*) *S* = *Pm-fmap* (*fmfilter* (λ*k*. *k* ∉ *S*) *xs*)
⟨*proof*⟩

lemma *lookup0-fmap-of-list-simps*:
lookup0 (*fmap-of-list* ((*x*, *y*)#*xs*)) *i* = (*if x* = *i then y else lookup0* (*fmap-of-list xs*) *i*)
lookup0 (*fmap-of-list* []) *i* = *0*
⟨*proof*⟩

lemma *if-poly-mapping-eq-iff*:
(*if x* = *y then a else b*) =
(*if* (∀ *i*∈*keys x* ∪ *keys y*. *lookup x i* = *lookup y i*) *then a else b*)
⟨*proof*⟩

lemma *keys-add-eq*: *keys* (*a* + *b*) = *keys a* ∪ *keys b* − {*x* ∈ *keys a* ∩ *keys b*. *lookup a x* + *lookup b x* = *0*}
⟨*proof*⟩

context *term-powerprod*
begin

context includes *fmap.lifting* **begin**

lift-definition *shift-keys*::'*a* ⇒ ('*t*, '*b*) *fmap* ⇒ ('*t*, '*b*) *fmap*
is λ*t m x*. *if t adds*$_p$ *x then m* (*x* ⊖ *t*) *else None*
⟨*proof*⟩

definition *shift-map-keys t f m* = *fmmap f* (*shift-keys t m*)

lemma *compute-shift-map-keys*[*code*]:

shift-map-keys t f (fmap-of-list xs) = fmap-of-list (map (λ(k, v). (t ⊕ k, f v)) xs)
⟨*proof*⟩

end

lemmas [*simp*] = *compute-zero-pp*[*symmetric*]

lemma *compute-monom-mult-poly-mapping* [*code*]:
monom-mult c t (Pm-fmap xs) = Pm-fmap (if c = 0 then fmempty else shift-map-keys t ((∗) c) xs)
⟨*proof*⟩

lemma *compute-mult-scalar-poly-mapping* [*code*]:
Pm-fmap (fmap-of-list xs) ⊙ q = (case xs of ((t, c) # ys) ⇒
(monom-mult c t q + except (Pm-fmap (fmap-of-list ys)) {t} ⊙ q) | - ⇒
Pm-fmap fmempty)
⟨*proof*⟩

end

11.2.2 restore constructor view

named-theorems *mpoly-simps*

definition *monomial1 pp = monomial 1 pp*

lemma *monomial1-Nil*[*mpoly-simps*]: *monomial1 0 = 1*
⟨*proof*⟩

lemma *monomial-mp*: *monomial c (pp::'a⇒₀nat) = Const₀ c ∗ monomial1 pp*
for *c::'b::comm-semiring-1*
⟨*proof*⟩

lemma *monomial1-add*: *(monomial1 (a + b)::('a::monoid-add⇒₀'b::comm-semiring-1))*
= monomial1 a ∗ monomial1 b
⟨*proof*⟩

lemma *monomial1-monomial*: *monomial1 (monomial n v) = (Var₀ v::-⇒₀('b::comm-semiring-1)) ^n*
⟨*proof*⟩

lemma *Ball-True*: *(∀ x∈X. True) ⟷ True* ⟨*proof*⟩
lemma *Collect-False*: *{x. False} = {}* ⟨*proof*⟩

lemma *Pm-fmap-sum*: *Pm-fmap f = (∑ x ∈ fmdom' f. monomial (lookup0 f x) x)*
including *fmap.lifting*
⟨*proof*⟩

lemma *MPoly-numeral*: *MPoly* (*numeral x*) = *numeral x*
⟨*proof*⟩

lemma *MPoly-power*: *MPoly* (*x* ^ *n*) = *MPoly x* ^ *n*
⟨*proof*⟩

lemmas [*mpoly-simps*] = *Pm-fmap-sum*
add.assoc[*symmetric*] *mult.assoc*[*symmetric*]
add-0 add-0-right mult-1 mult-1-right mult-zero-left mult-zero-right power-0 power-one-right
fmdom'-fmap-of-list
list.map fst-conv
sum.insert-remove finite-insert finite.emptyI
lookup0-fmap-of-list-simps
num.simps rel-simps
if-True if-False
insert-Diff-if insert-iff empty-Diff empty-iff
simp-thms
sum.empty
if-poly-mapping-eq-iff
keys-zero keys-one
keys-add-eq
keys-single
Un-insert-left Un-empty-left
Int-insert-left Int-empty-left
Collect-False
lookup-add lookup-single lookup-zero lookup-one
Set.ball-simps
when-simps
monomial-mp
monomial1-add
monomial1-monomial
Const$_0$*-one Const*$_0$*-zero Const*$_0$*-numeral Const*$_0$*-minus*
set-simps

A simproc for postprocessing with *mpoly-simps* and not polluting [*code-post*]:

⟨*ML*⟩

11.2.3 Ordered Power-Products

lemma *foldl-assoc*:
assumes $\bigwedge x\ y\ z.\ f\ (f\ x\ y)\ z = f\ x\ (f\ y\ z)$
shows *foldl f* (*f a b*) *xs* = *f a* (*foldl f b xs*)
⟨*proof*⟩

context *ordered-term*
begin

definition *list-max*::*'t list* ⇒ *'t* **where**
list-max xs ≡ *foldl ord-term-lin.max min-term xs*

lemma *list-max-Cons*: *list-max* (*x* # *xs*) = *ord-term-lin.max x* (*list-max xs*)
⟨*proof*⟩

lemma *list-max-empty*: *list-max* [] = *min-term*
⟨*proof*⟩

lemma *list-max-in-list*:
assumes *xs* ≠ []
shows *list-max xs* ∈ *set xs*
⟨*proof*⟩

lemma *list-max-maximum*:
assumes *a* ∈ *set xs*
shows *a* ⪯$_t$ (*list-max xs*)
⟨*proof*⟩

lemma *list-max-nonempty*:
assumes *xs* ≠ []
shows *list-max xs* = *ord-term-lin.Max* (*set xs*)
⟨*proof*⟩

lemma *in-set-clearjunk-iff-map-of-eq-Some*:
(*a*, *b*) ∈ *set* (*AList.clearjunk xs*) ⟷ *map-of xs a* = *Some b*
⟨*proof*⟩

lemma *Pm-fmap-of-list-eq-zero-iff*:
Pm-fmap (*fmap-of-list xs*) = *0* ⟷ [(*k*, *v*)←*AList.clearjunk xs* . *v* ≠ *0*] = []
⟨*proof*⟩

lemma *fmdom'-clearjunk0*: *fmdom'* (*clearjunk0 xs*) = *fmdom' xs* − {*x*. *fmlookup xs x* = *Some 0*}
⟨*proof*⟩

lemma *compute-lt-poly-mapping*[*code*]:
lt (*Pm-fmap* (*fmap-of-list xs*)) = *list-max* (*map fst* [(*k*, *v*) ← *AList.clearjunk xs*. *v* ≠ *0*])
⟨*proof*⟩

lemma *compute-higher-poly-mapping* [*code*]:
higher (*Pm-fmap xs*) *t* = *Pm-fmap* (*fmfilter* (λ*k*. *t* ≺$_t$ *k*) *xs*)
⟨*proof*⟩

lemma *compute-lower-poly-mapping* [*code*]:
lower (*Pm-fmap xs*) *t* = *Pm-fmap* (*fmfilter* (λ*k*. *k* ≺$_t$ *t*) *xs*)
⟨*proof*⟩

end

lifting-update *poly-mapping.lifting*

lifting-forget *poly-mapping.lifting*

11.3 Computations

11.3.1 Scalar Polynomials

type-synonym *'a mpoly-tc* = (*nat* ⇒$_0$ *nat*)⇒$_0$ *'a*

definition *shift-map-keys-punit* = *term-powerprod.shift-map-keys to-pair-unit fst*

lemma *compute-shift-map-keys-punit* [*code*]:
shift-map-keys-punit t f (*fmap-of-list xs*) = *fmap-of-list* (*map* (λ(*k*, *v*). (*t* + *k*, *f v*)) *xs*)
⟨*proof*⟩

global-interpretation *punit*: *term-powerprod to-pair-unit fst*
rewrites *punit.adds-term* = (*adds*)
and *punit.pp-of-term* = (λ*x*. *x*)
and *punit.component-of-term* = (λ-. ())
defines *monom-mult-punit* = *punit.monom-mult*
and *mult-scalar-punit* = *punit.mult-scalar*
⟨*proof*⟩

lemma *compute-monom-mult-punit* [*code*]:
monom-mult-punit c t (*Pm-fmap xs*) = *Pm-fmap* (*if c* = *0 then fmempty else shift-map-keys-punit t* ((∗) *c*) *xs*)
⟨*proof*⟩

lemma *compute-mult-scalar-punit* [*code*]:
Pm-fmap (*fmap-of-list xs*) ∗ *q* = (*case xs of* ((*t*, *c*) # *ys*) ⇒
(*monom-mult-punit c t q* + *except* (*Pm-fmap* (*fmap-of-list ys*)) {*t*} ∗ *q*) | - ⇒
Pm-fmap fmempty)
⟨*proof*⟩

locale *trivariate*$_0$-*rat*
begin

abbreviation *X*::*rat mpoly-tc* **where** *X* ≡ *Var*$_0$ (*0*::*nat*)
abbreviation *Y*::*rat mpoly-tc* **where** *Y* ≡ *Var*$_0$ (*1*::*nat*)
abbreviation *Z*::*rat mpoly-tc* **where** *Z* ≡ *Var*$_0$ (*2*::*nat*)

end

locale *trivariate*
begin

abbreviation *X* ≡ *Var 0*
abbreviation *Y* ≡ *Var 1*
abbreviation *Z* ≡ *Var 2*

end

experiment begin interpretation *trivariate$_0$-rat* ⟨*proof*⟩

lemma
keys $(X^2 * Z \hat{} 3 + 2 * Y \hat{} 3 * Z^2) =$
$\{$*monomial 2 0* + *monomial 3 2*, *monomial 3 1* + *monomial 2 2*$\}$
⟨*proof*⟩

lemma
keys $(X^2 * Z \hat{} 3 + 2 * Y \hat{} 3 * Z^2) =$
$\{$*monomial 2 0* + *monomial 3 2*, *monomial 3 1* + *monomial 2 2*$\}$
⟨*proof*⟩

lemma
$-1 * X^2 * Z \hat{} 7 + -2 * Y \hat{} 3 * Z^2 = -X^2 * Z \hat{} 7 + -2 * Y \hat{} 3 * Z^2$
⟨*proof*⟩

lemma
$X^2 * Z \hat{} 7 + 2 * Y \hat{} 3 * Z^2 + X^2 * Z \hat{} 4 + -2 * Y \hat{} 3 * Z^2 = X^2 * Z \hat{} 7 + X^2 * Z \hat{} 4$
⟨*proof*⟩

lemma
$X^2 * Z \hat{} 7 + 2 * Y \hat{} 3 * Z^2 - X^2 * Z \hat{} 4 + -2 * Y \hat{} 3 * Z^2 =$
$X^2 * Z \hat{} 7 - X^2 * Z \hat{} 4$
⟨*proof*⟩

lemma
lookup $(X^2 * Z \hat{} 7 + 2 * Y \hat{} 3 * Z^2 + 2)$ $(sparse_0\ [(0, 2), (2, 7)]) = 1$
⟨*proof*⟩

lemma
$X^2 * Z \hat{} 7 + 2 * Y \hat{} 3 * Z^2 \neq$
$X^2 * Z \hat{} 4 + -2 * Y \hat{} 3 * Z^2$
⟨*proof*⟩

lemma
$0 * X\hat{}2 * Z\hat{}7 + 0 * Y\hat{}3 * Z^2 = 0$
⟨*proof*⟩

lemma
monom-mult-punit 3 $(sparse_0\ [(1, 2::nat)])$ $(X^2 * Z + 2 * Y \hat{} 3 * Z^2) =$
$3 * Y^2 * Z * X^2 + 6 * Y \hat{} 5 * Z^2$
⟨*proof*⟩

lemma
monomial (-4) $(sparse_0\ [(0, 2::nat)]) = -4 * X^2$

⟨*proof*⟩

lemma *monomial* (*0*::*rat*) (*sparse*$_0$ [(*0*::*nat*, *2*::*nat*)]) = *0*
⟨*proof*⟩

lemma
(*X*^{2} * *Z* + *2* * *Y* ^ *3* * *Z*^{2}) * (*X*^{2} * *Z* ^ *3* + − *2* * *Y* ^ *3* * *Z*^{2}) =
X ^ *4* * *Z* ^ *4* + − *2* * *X*^{2} * *Z* ^ *3* * *Y* ^ *3* +
− *4* * *Y* ^ *6* * *Z* ^ *4* + *2* * *Y* ^ *3* * *Z* ^ *5* * *X*^{2}
⟨*proof*⟩

end

11.3.2 Vector-Polynomials

type-synonym *'a vmpoly-tc* = ((*nat* ⇒$_0$ *nat*) × *nat*) ⇒$_0$ *'a*

definition *shift-map-keys-pprod* = *pprod.shift-map-keys*

global-interpretation *pprod*: *term-powerprod* λ*x*. *x* λ*x*. *x*
rewrites *pprod.pp-of-term* = *fst*
and *pprod.component-of-term* = *snd*
defines *splus-pprod* = *pprod.splus*
and *monom-mult-pprod* = *pprod.monom-mult*
and *mult-scalar-pprod* = *pprod.mult-scalar*
and *adds-term-pprod* = *pprod.adds-term*
⟨*proof*⟩

lemma *compute-adds-term-pprod* [*code-unfold*]:
adds-term-pprod u v = (*snd u* = *snd v* ∧ *adds-pp-add-linorder* (*fst u*) (*fst v*))
⟨*proof*⟩

lemma *compute-splus-pprod* [*code*]: *splus-pprod t* (*s*, *i*) = (*t* + *s*, *i*)
⟨*proof*⟩

lemma *compute-shift-map-keys-pprod* [*code*]:
shift-map-keys-pprod t f (*fmap-of-list xs*) = *fmap-of-list* (*map* (λ(*k*, *v*). (*splus-pprod t k*, *f v*)) *xs*)
⟨*proof*⟩

lemma *compute-monom-mult-pprod* [*code*]:
monom-mult-pprod c t (*Pm-fmap xs*) = *Pm-fmap* (*if c* = *0 then fmempty else shift-map-keys-pprod t* ((∗) *c*) *xs*)
⟨*proof*⟩

lemma *compute-mult-scalar-pprod* [*code*]:
mult-scalar-pprod (*Pm-fmap* (*fmap-of-list xs*)) *q* = (*case xs of* ((*t*, *c*) # *ys*) ⇒
(*monom-mult-pprod c t q* + *mult-scalar-pprod* (*except* (*Pm-fmap* (*fmap-of-list ys*)) {*t*}) *q*) | - ⇒

Pm-fmap fmempty)
⟨*proof*⟩

definition $Vec_0 :: nat \Rightarrow ((('a \Rightarrow_0 nat) \Rightarrow_0 'b) \Rightarrow ((('a \Rightarrow_0 nat) \times nat) \Rightarrow_0 'b$::*semiring-1* **where**
$Vec_0\ i\ p = mult\text{-}scalar\text{-}pprod\ p\ (Poly\text{-}Mapping.single\ (0,\ i)\ 1)$

experiment begin interpretation $trivariate_0$-*rat* ⟨*proof*⟩

lemma
$keys\ (Vec_0\ 0\ (X^2 * Z\ \hat{}\ 3) + Vec_0\ 1\ (2 * Y\ \hat{}\ 3 * Z^2)) =$
$\{(sparse_0\ [(0,\ 2),\ (2,\ 3)],\ 0),\ (sparse_0\ [(1,\ 3),\ (2,\ 2)],\ 1)\}$
⟨*proof*⟩

lemma
$keys\ (Vec_0\ 0\ (X^2 * Z\ \hat{}\ 3) + Vec_0\ 2\ (2 * Y\ \hat{}\ 3 * Z^2)) =$
$\{(sparse_0\ [(0,\ 2),\ (2,\ 3)],\ 0),\ (sparse_0\ [(1,\ 3),\ (2,\ 2)],\ 2)\}$
⟨*proof*⟩

lemma
$Vec_0\ 1\ (X^2 * Z\ \hat{}\ 7 + 2 * Y\ \hat{}\ 3 * Z^2) + Vec_0\ 3\ (X^2 * Z\ \hat{}\ 4) + Vec_0\ 1\ (-\ 2 * Y\ \hat{}\ 3 * Z^2) =$
$Vec_0\ 1\ (X^2 * Z\ \hat{}\ 7) + Vec_0\ 3\ (X^2 * Z\ \hat{}\ 4)$
⟨*proof*⟩

lemma
$lookup\ (Vec_0\ 0\ (X^2 * Z\ \hat{}\ 7) + Vec_0\ 1\ (2 * Y\ \hat{}\ 3 * Z^2 + 2))\ (sparse_0\ [(0,\ 2),\ (2,\ 7)],\ 0) = 1$
⟨*proof*⟩

lemma
$lookup\ (Vec_0\ 0\ (X^2 * Z\ \hat{}\ 7) + Vec_0\ 1\ (2 * Y\ \hat{}\ 3 * Z^2 + 2))\ (sparse_0\ [(0,\ 2),\ (2,\ 7)],\ 1) = 0$
⟨*proof*⟩

lemma
$Vec_0\ 0\ (0 * X\hat{}2 * Z\hat{}7) + Vec_0\ 1\ (0 * Y\hat{}3 * Z^2) = 0$
⟨*proof*⟩

lemma
$monom\text{-}mult\text{-}pprod\ 3\ (sparse_0\ [(1,\ 2::nat)])\ (Vec_0\ 0\ (X^2 * Z) + Vec_0\ 1\ (2 * Y\ \hat{}\ 3 * Z^2)) =$
$Vec_0\ 0\ (3 * Y^2 * Z * X^2) + Vec_0\ 1\ (6 * Y\ \hat{}\ 5 * Z^2)$
⟨*proof*⟩

end

11.4 Code setup for type MPoly

postprocessing from Var_0, $Const_0$ to *Var*, *Const*.

lemmas [*code-post*] =
plus-mpoly.abs-eq[*symmetric*]
times-mpoly.abs-eq[*symmetric*]
MPoly-numeral
MPoly-power
one-mpoly-def[*symmetric*]
Var.abs-eq[*symmetric*]
Const.abs-eq[*symmetric*]

instantiation *mpoly*::({*equal*, *zero*})*equal* **begin**

lift-definition *equal-mpoly*:: *'a mpoly* ⇒ *'a mpoly* ⇒ *bool* **is** *HOL.equal* ⟨*proof*⟩

instance ⟨*proof*⟩

end

experiment begin interpretation *trivariate* ⟨*proof*⟩

lemmas [*mpoly-simps*] = *plus-mpoly.abs-eq*

lemma *content-primitive* (*4 * X * Y^2 * Z^3 + 6 * X^2 * Y^4 + 8 * X^2 * Y^5*)
=
(*2*::*int*, *2 * X * Y^2 * Z ^ 3 + 3 * X^2 * Y ^ 4 + 4 * X^2 * Y ^ 5*)
⟨*proof*⟩

end

end

theory *PP-Type*
imports *Power-Products*
begin

For code generation, we must introduce a copy of type $'a \Rightarrow_0 'b$ for power-products.

typedef (**overloaded**) (*'a*, *'b*) *pp* = *UNIV*::(*'a* ⇒$_0$ *'b*) *set*
morphisms *mapping-of PP* ⟨*proof*⟩

setup-lifting *type-definition-pp*

lift-definition *pp-of-fun* :: (*'a* ⇒ *'b*) ⇒ (*'a*, *'b*::*zero*) *pp*
is *Abs-poly-mapping* ⟨*proof*⟩

11.5 *lookup-pp*, *keys-pp* and *single-pp*

lift-definition *lookup-pp* :: $('a, 'b{::}zero)\ pp \Rightarrow 'a \Rightarrow 'b$ **is** *lookup* ⟨*proof*⟩

lift-definition *keys-pp* :: $('a, 'b{::}zero)\ pp \Rightarrow 'a\ set$ **is** *keys* ⟨*proof*⟩

lift-definition *single-pp* :: $'a \Rightarrow 'b \Rightarrow ('a, 'b{::}zero)\ pp$ **is** *Poly-Mapping.single* ⟨*proof*⟩

lemma *lookup-pp-of-fun*: *finite* $\{x.\ f\ x \neq 0\} \Longrightarrow$ *lookup-pp* (*pp-of-fun f*) = *f*
⟨*proof*⟩

lemma *pp-of-lookup*: *pp-of-fun* (*lookup-pp t*) = *t*
⟨*proof*⟩

lemma *pp-eqI*: $(\bigwedge u.$ *lookup-pp s u* = *lookup-pp t u*$) \Longrightarrow$ *s* = *t*
⟨*proof*⟩

lemma *pp-eq-iff*: (*s* = *t*) $\longleftrightarrow$ (*lookup-pp s* = *lookup-pp t*)
⟨*proof*⟩

lemma *keys-pp-iff*: *x* ∈ *keys-pp t* $\longleftrightarrow$ (*lookup-pp t x* ≠ *0*)
⟨*proof*⟩

lemma *pp-eqI'*:
assumes $\bigwedge u.$ *u* ∈ *keys-pp s* ∪ *keys-pp t* $\Longrightarrow$ *lookup-pp s u* = *lookup-pp t u*
shows *s* = *t*
⟨*proof*⟩

lemma *lookup-single-pp*: *lookup-pp* (*single-pp x e*) *y* = (*e when x* = *y*)
⟨*proof*⟩

11.6 Additive Structure

instantiation *pp* :: (*type*, *zero*) *zero*
begin

lift-definition *zero-pp* :: $('a, 'b)\ pp$ **is** $0{::}'a \Rightarrow_0 'b$ ⟨*proof*⟩

lemma *lookup-zero-pp* [*simp*]: *lookup-pp 0* = *0*
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *single-pp-zero* [*simp*]: *single-pp x 0* = *0*
⟨*proof*⟩

instantiation *pp* :: (*type*, *monoid-add*) *monoid-add*

begin

lift-definition *plus-pp* :: ('*a*, '*b*) *pp* ⇒ ('*a*, '*b*) *pp* ⇒ ('*a*, '*b*) *pp* **is** (+)::('*a* ⇒$_0$ '*b*) ⇒ - ⟨*proof*⟩

lemma *lookup-plus-pp*: *lookup-pp* (*s* + *t*) = *lookup-pp s* + *lookup-pp t*
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *single-pp-plus*: *single-pp x a* + *single-pp x b* = *single-pp x* (*a* + *b*)
⟨*proof*⟩

instance *pp* :: (*type*, *comm-monoid-add*) *comm-monoid-add*
⟨*proof*⟩

instantiation *pp* :: (*type*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add*
begin

lift-definition *minus-pp* :: ('*a*, '*b*) *pp* ⇒ ('*a*, '*b*) *pp* ⇒ ('*a*, '*b*) *pp* **is** (−)::('*a* ⇒$_0$ '*b*) ⇒ - ⟨*proof*⟩

lemma *lookup-minus-pp*: *lookup-pp* (*s* − *t*) = *lookup-pp s* − *lookup-pp t*
⟨*proof*⟩

instance ⟨*proof*⟩

end

11.7 '*a* ⇒$_0$ '*b* belongs to class *comm-powerprod*

instance *poly-mapping* :: (*type*, *cancel-comm-monoid-add*) *comm-powerprod*
⟨*proof*⟩

11.8 '*a* ⇒$_0$ '*b* belongs to class *ninv-comm-monoid-add*

instance *poly-mapping* :: (*type*, *ninv-comm-monoid-add*) *ninv-comm-monoid-add*
⟨*proof*⟩

11.9 ('*a*, '*b*) *pp* belongs to class *lcs-powerprod*

lemma *adds-pp-iff*: (*s adds t*) ⟷ (*mapping-of s adds mapping-of t*)
⟨*proof*⟩

instantiation *pp* :: (*type*, *add-linorder*) *lcs-powerprod*
begin

lift-definition *lcs-pp* :: (*'a*, *'b*) *pp* ⇒ (*'a*, *'b*) *pp* ⇒ (*'a*, *'b*) *pp* **is** *lcs-powerprod-class.lcs*
⟨*proof*⟩

lemma *lookup-lcs-pp*: *lookup-pp* (*lcs s t*) *x* = *max* (*lookup-pp s x*) (*lookup-pp t x*)
⟨*proof*⟩

instance
⟨*proof*⟩

end

11.10 (*'a*, *'b*) *pp* belongs to class *ulcs-powerprod*

instance *pp* :: (*type*, *add-linorder-min*) *ulcs-powerprod* ⟨*proof*⟩

11.11 Dickson's lemma for power-products in finitely many indeterminates

lemma *almost-full-on-pp-iff*:
almost-full-on (*adds*) *A* ⟷ *almost-full-on* (*adds*) (*mapping-of* ‘ *A*) (**is** *?l* ⟷ *?r*)
⟨*proof*⟩

lift-definition *varnum-pp* :: (*'a*::*countable*, *'b*::*zero*) *pp* ⇒ *nat* **is** *varnum* {} ⟨*proof*⟩

lemma *dickson-grading-varnum-pp*:
dickson-grading (*varnum-pp*::(*'a*::*countable*, *'b*::*add-wellorder*) *pp* ⇒ *nat*)
⟨*proof*⟩

instance *pp* :: (*countable*, *add-wellorder*) *graded-dickson-powerprod*
⟨*proof*⟩

instance *pp* :: (*finite*, *add-wellorder*) *dickson-powerprod*
⟨*proof*⟩

11.12 Lexicographic Term Order

lift-definition *lex-pp* :: (*'a*, *'b*) *pp* ⇒ (*'a*::*linorder*, *'b*::{*zero*,*linorder*}) *pp* ⇒ *bool*
is *lex-pm* ⟨*proof*⟩

lift-definition *lex-pp-strict* :: (*'a*, *'b*) *pp* ⇒ (*'a*::*linorder*, *'b*::{*zero*,*linorder*}) *pp*
⇒ *bool* **is** *lex-pm-strict* ⟨*proof*⟩

lemma *lex-pp-alt*: *lex-pp s t* = (*s* = *t* ∨ (∃ *x*. *lookup-pp s x* < *lookup-pp t x* ∧
(∀ *y*<*x*. *lookup-pp s y* = *lookup-pp t y*)))
⟨*proof*⟩

lemma *lex-pp-refl*: *lex-pp s s*
⟨*proof*⟩

lemma *lex-pp-antisym*: *lex-pp s t* $\Longrightarrow$ *lex-pp t s* $\Longrightarrow$ *s* = *t*
⟨*proof*⟩

lemma *lex-pp-trans*: *lex-pp s t* $\Longrightarrow$ *lex-pp t u* $\Longrightarrow$ *lex-pp s u*
⟨*proof*⟩

lemma *lex-pp-lin*: *lex-pp s t* ∨ *lex-pp t s*
⟨*proof*⟩

lemma *lex-pp-lin'*: ¬ *lex-pp t s* $\Longrightarrow$ *lex-pp s t*
⟨*proof*⟩

corollary *lex-pp-strict-alt* [*code*]:
lex-pp-strict s t = (¬ *lex-pp t s*) **for** *s t*::(-, -::*ordered-comm-monoid-add*) *pp*
⟨*proof*⟩

lemma *lex-pp-zero-min*: *lex-pp 0 s* **for** *s*::(-, -::*add-linorder-min*) *pp*
⟨*proof*⟩

lemma *lex-pp-plus-monotone*: *lex-pp s t* $\Longrightarrow$ *lex-pp* (*s* + *u*) (*t* + *u*)
for *s t*::(-, -::{*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*}) *pp*
⟨*proof*⟩

lemma *lex-pp-plus-monotone'*: *lex-pp s t* $\Longrightarrow$ *lex-pp* (*u* + *s*) (*u* + *t*)
for *s t*::(-, -::{*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*}) *pp*
⟨*proof*⟩

instantiation *pp* :: (*linorder*, {*ordered-comm-monoid-add*, *linorder*}) *linorder*
begin

definition *less-eq-pp* :: ('*a*, '*b*) *pp* ⇒ ('*a*, '*b*) *pp* ⇒ *bool*
where *less-eq-pp* = *lex-pp*

definition *less-pp* :: ('*a*, '*b*) *pp* ⇒ ('*a*, '*b*) *pp* ⇒ *bool*
where *less-pp* = *lex-pp-strict*

instance ⟨*proof*⟩

end

11.13 Degree

lift-definition *deg-pp* :: ('*a*, '*b*::*comm-monoid-add*) *pp* ⇒ '*b* **is** *deg-pm* ⟨*proof*⟩

lemma *deg-pp-alt*: *deg-pp s* = *sum* (*lookup-pp s*) (*keys-pp s*)
⟨*proof*⟩

lemma *deg-pp-zero* [*simp*]: *deg-pp 0* = *0*
⟨*proof*⟩

lemma *deg-pp-eq-0-iff* [*simp*]: *deg-pp s = 0* ⟷ *s = 0* **for** *s*::('*a*, '*b*::*add-linorder-min*) *pp*
⟨*proof*⟩

lemma *deg-pp-plus*: *deg-pp* (*s* + *t*) = *deg-pp s* + *deg-pp* (*t*::('*a*, '*b*::*comm-monoid-add*) *pp*)
⟨*proof*⟩

lemma *deg-pp-single*: *deg-pp* (*single-pp x k*) = *k*
⟨*proof*⟩

11.14 Degree-Lexicographic Term Order

lift-definition *dlex-pp* :: ('*a*::*linorder*, '*b*::{*ordered-comm-monoid-add*,*linorder*}) *pp* ⇒ ('*a*, '*b*) *pp* ⇒ *bool*
is *dlex-pm* ⟨*proof*⟩

lift-definition *dlex-pp-strict* :: ('*a*::*linorder*, '*b*::{*ordered-comm-monoid-add*,*linorder*}) *pp* ⇒ ('*a*, '*b*) *pp* ⇒ *bool*
is *dlex-pm-strict* ⟨*proof*⟩

lemma *dlex-pp-alt*: *dlex-pp s t* ⟷ (*deg-pp s* < *deg-pp t* ∨ (*deg-pp s* = *deg-pp t* ∧ *lex-pp s t*))
⟨*proof*⟩

lemma *dlex-pp-refl*: *dlex-pp s s*
⟨*proof*⟩

lemma *dlex-pp-antisym*: *dlex-pp s t* ⟹ *dlex-pp t s* ⟹ *s* = *t*
⟨*proof*⟩

lemma *dlex-pp-trans*: *dlex-pp s t* ⟹ *dlex-pp t u* ⟹ *dlex-pp s u*
⟨*proof*⟩

lemma *dlex-pp-lin*: *dlex-pp s t* ∨ *dlex-pp t s*
⟨*proof*⟩

corollary *dlex-pp-strict-alt* [*code*]: *dlex-pp-strict s t* = (¬ *dlex-pp t s*)
⟨*proof*⟩

lemma *dlex-pp-zero-min*: *dlex-pp 0 s*
for *s t*::(-, -::*add-linorder-min*) *pp*
⟨*proof*⟩

lemma *dlex-pp-plus-monotone*: *dlex-pp s t* ⟹ *dlex-pp* (*s* + *u*) (*t* + *u*)
for *s t*::(-, -::{*ordered-ab-semigroup-add-imp-le*, *ordered-cancel-comm-monoid-add*}) *pp*
⟨*proof*⟩

11.15 Degree-Reverse-Lexicographic Term Order

lift-definition *drlex-pp* :: ('*a*::*linorder*, '*b*::{*ordered-comm-monoid-add*,*linorder*}) *pp* ⇒ ('*a*, '*b*) *pp* ⇒ *bool*
  is *drlex-pm* ⟨*proof*⟩

lift-definition *drlex-pp-strict* :: ('*a*::*linorder*, '*b*::{*ordered-comm-monoid-add*,*linorder*}) *pp* ⇒ ('*a*, '*b*) *pp* ⇒ *bool*
  is *drlex-pm-strict* ⟨*proof*⟩

lemma *drlex-pp-alt*: *drlex-pp s t* ⟷ (*deg-pp s* < *deg-pp t* ∨ (*deg-pp s* = *deg-pp t* ∧ *lex-pp t s*))
  ⟨*proof*⟩

lemma *drlex-pp-refl*: *drlex-pp s s*
  ⟨*proof*⟩

lemma *drlex-pp-antisym*: *drlex-pp s t* ⟹ *drlex-pp t s* ⟹ *s* = *t*
  ⟨*proof*⟩

lemma *drlex-pp-trans*: *drlex-pp s t* ⟹ *drlex-pp t u* ⟹ *drlex-pp s u*
  ⟨*proof*⟩

lemma *drlex-pp-lin*: *drlex-pp s t* ∨ *drlex-pp t s*
  ⟨*proof*⟩

corollary *drlex-pp-strict-alt* [*code*]: *drlex-pp-strict s t* = (¬ *drlex-pp t s*)
  ⟨*proof*⟩

lemma *drlex-pp-zero-min*: *drlex-pp 0 s*
  for *s t*::(-, -::*add-linorder-min*) *pp*
  ⟨*proof*⟩

lemma *drlex-pp-plus-monotone*: *drlex-pp s t* ⟹ *drlex-pp* (*s* + *u*) (*t* + *u*)
  for *s t*::(-, -::{*ordered-ab-semigroup-add-imp-le*, *ordered-cancel-comm-monoid-add*}) *pp*
  ⟨*proof*⟩

end

12 Associative Lists with Sorted Keys

theory *OAlist*
  imports *Deriving.Comparator*
begin

We define the type of *ordered associative lists* (oalist). An oalist is an associative list (i. e. a list of pairs) such that the keys are distinct and sorted wrt. some linear order relation, and no key is mapped to *0*::'*a*. The latter

invariant allows to implement various functions operating on oalists more efficiently.

The ordering of the keys in an oalist *xs* is encoded as an additional parameter of *xs*. This means that oalists may be ordered wrt. different orderings, even if they are of the same type. Operations operating on more than one oalists, like *map2-val*, typically ensure that the orderings of their arguments are identical by re-ordering one argument wrt. the order relation of the other. This, however, implies that equality of order relations must be effectively decidable if executable code is to be generated.

12.1 Preliminaries

fun *min-list-param* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a$ **where**
min-list-param rel $(x \# xs) =$ (*case xs of* $[] \Rightarrow x \mid \text{-} \Rightarrow$ (*let m = min-list-param rel xs in if rel x m then x else m*))

lemma *min-list-param-in*:
assumes $xs \neq []$
shows *min-list-param rel xs* $\in$ *set xs*
⟨*proof*⟩

lemma *min-list-param-minimal*:
assumes *transp rel* **and** $\bigwedge x\ y.\ x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow rel\ x\ y \vee rel\ y\ x$
and $z \in set\ xs$
shows *rel* (*min-list-param rel xs*) *z*
⟨*proof*⟩

definition *comp-of-ord* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ comparator$ **where**
comp-of-ord le x y = (*if le x y then if x = y then Eq else Lt else Gt*)

lemma *comp-of-ord-eq-comp-of-ords*:
assumes *antisymp le*
shows *comp-of-ord le* = *comp-of-ords le* $(\lambda x\ y.\ le\ x\ y \wedge \neg\ le\ y\ x)$
⟨*proof*⟩

lemma *comparator-converse*:
assumes *comparator cmp*
shows *comparator* $(\lambda x\ y.\ cmp\ y\ x)$
⟨*proof*⟩

lemma *comparator-composition*:
assumes *comparator cmp* **and** *inj f*
shows *comparator* $(\lambda x\ y.\ cmp\ (f\ x)\ (f\ y))$
⟨*proof*⟩

12.2 Type *key-order*

typedef $'a$ *key-order* = $\{compare :: 'a\ comparator.\ comparator\ compare\}$

morphisms *key-compare Abs-key-order*
⟨*proof*⟩

lemma *comparator-key-compare* [*simp*, *intro*!]: *comparator* (*key-compare ko*)
⟨*proof*⟩

instantiation *key-order* :: (*type*) *equal*
begin

definition *equal-key-order* :: *'a key-order* ⇒ *'a key-order* ⇒ *bool* **where** *equal-key-order* = (=)

instance ⟨*proof*⟩

end

setup-lifting *type-definition-key-order*

instantiation *key-order* :: (*type*) *uminus*
begin

lift-definition *uminus-key-order* :: *'a key-order* ⇒ *'a key-order* **is** λ*c x y*. *c y x*
⟨*proof*⟩

instance ⟨*proof*⟩

end

lift-definition *le-of-key-order* :: *'a key-order* ⇒ *'a* ⇒ *'a* ⇒ *bool* **is** λ*cmp*. *le-of-comp cmp* ⟨*proof*⟩

lift-definition *lt-of-key-order* :: *'a key-order* ⇒ *'a* ⇒ *'a* ⇒ *bool* **is** λ*cmp*. *lt-of-comp cmp* ⟨*proof*⟩

definition *key-order-of-ord* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a key-order*
where *key-order-of-ord ord* = *Abs-key-order* (*comp-of-ord ord*)

lift-definition *key-order-of-le* :: *'a*::*linorder key-order* **is** *comparator-of*
⟨*proof*⟩

interpretation *key-order-lin*: *linorder le-of-key-order ko lt-of-key-order ko*
⟨*proof*⟩

lemma *le-of-key-order-alt*: *le-of-key-order ko x y* = (*key-compare ko x y* ≠ *Gt*)
⟨*proof*⟩

lemma *lt-of-key-order-alt*: *lt-of-key-order ko x y* = (*key-compare ko x y* = *Lt*)
⟨*proof*⟩

lemma *key-compare-Gt*: *key-compare ko x y = Gt ⟷ key-compare ko y x = Lt*
⟨*proof*⟩

lemma *key-compare-Eq*: *key-compare ko x y = Eq ⟷ x = y*
⟨*proof*⟩

lemma *key-compare-same* [*simp*]: *key-compare ko x x = Eq*
⟨*proof*⟩

lemma *uminus-key-compare* [*simp*]: *invert-order* (*key-compare ko x y*) = *key-compare ko y x*
⟨*proof*⟩

lemma *key-compare-uminus* [*simp*]: *key-compare* (− *ko*) *x y* = *key-compare ko y x*
⟨*proof*⟩

lemma *uminus-key-order-sameD*:
assumes − *ko* = (*ko*::′*a key-order*)
shows *x* = (*y*::′*a*)
⟨*proof*⟩

lemma *key-compare-key-order-of-ord*:
assumes *antisymp ord* **and** *transp ord* **and** ⋀*x y*. *ord x y* ∨ *ord y x*
shows *key-compare* (*key-order-of-ord ord*) = (λ*x y*. *if ord x y then if x* = *y then Eq else Lt else Gt*)
⟨*proof*⟩

lemma *key-compare-key-order-of-le*:
key-compare key-order-of-le = (λ*x y*. *if x* < *y then Lt else if x* = *y then Eq else Gt*)
⟨*proof*⟩

12.3 Invariant in Context *comparator*

context *comparator*
begin

definition *oalist-inv-raw* :: (′*a* × ′*b*::*zero*) *list* ⇒ *bool*
where *oalist-inv-raw xs* ⟷ (*0* ∉ *snd* ‘ *set xs* ∧ *sorted-wrt lt* (*map fst xs*))

lemma *oalist-inv-rawI*:
assumes *0* ∉ *snd* ‘ *set xs* **and** *sorted-wrt lt* (*map fst xs*)
shows *oalist-inv-raw xs*
⟨*proof*⟩

lemma *oalist-inv-rawD1*:
assumes *oalist-inv-raw xs*
shows *0* ∉ *snd* ‘ *set xs*

⟨*proof*⟩

lemma *oalist-inv-rawD2*:
assumes *oalist-inv-raw xs*
shows *sorted-wrt lt* (*map fst xs*)
⟨*proof*⟩

lemma *oalist-inv-raw-Nil*: *oalist-inv-raw* []
⟨*proof*⟩

lemma *oalist-inv-raw-singleton*: *oalist-inv-raw* [(*k*, *v*)] ⟷ (*v* ≠ *0*)
⟨*proof*⟩

lemma *oalist-inv-raw-ConsI*:
assumes *oalist-inv-raw xs* **and** *v* ≠ *0* **and** *xs* ≠ [] ⟹ *lt k* (*fst* (*hd xs*))
shows *oalist-inv-raw* ((*k*, *v*) # *xs*)
⟨*proof*⟩

lemma *oalist-inv-raw-ConsD1*:
assumes *oalist-inv-raw* (*x* # *xs*)
shows *oalist-inv-raw xs*
⟨*proof*⟩

lemma *oalist-inv-raw-ConsD2*:
assumes *oalist-inv-raw* ((*k*, *v*) # *xs*)
shows *v* ≠ *0*
⟨*proof*⟩

lemma *oalist-inv-raw-ConsD3*:
assumes *oalist-inv-raw* ((*k*, *v*) # *xs*) **and** *k′* ∈ *fst* ‘ *set xs*
shows *lt k k′*
⟨*proof*⟩

lemma *oalist-inv-raw-tl*:
assumes *oalist-inv-raw xs*
shows *oalist-inv-raw* (*tl xs*)
⟨*proof*⟩

lemma *oalist-inv-raw-filter*:
assumes *oalist-inv-raw xs*
shows *oalist-inv-raw* (*filter P xs*)
⟨*proof*⟩

lemma *oalist-inv-raw-map*:
assumes *oalist-inv-raw xs*
and ⋀*a*. *snd* (*f a*) = *0* ⟹ *snd a* = *0*
and ⋀*a b*. *comp* (*fst* (*f a*)) (*fst* (*f b*)) = *comp* (*fst a*) (*fst b*)
shows *oalist-inv-raw* (*map f xs*)
⟨*proof*⟩

lemma *oalist-inv-raw-induct* [*consumes 1*, *case-names Nil Cons*]:
assumes *oalist-inv-raw xs*
assumes *P* []
assumes $\bigwedge k\ v\ xs.$ *oalist-inv-raw* $((k, v) \# xs) \Longrightarrow$ *oalist-inv-raw xs* $\Longrightarrow v \neq 0 \Longrightarrow$
$(\bigwedge k'.\ k' \in$ *fst* ‘ *set xs* $\Longrightarrow$ *lt k k'*$) \Longrightarrow P\ xs \Longrightarrow P\ ((k, v) \# xs)$
shows *P xs*
⟨*proof*⟩

12.4 Operations on Lists of Pairs in Context *comparator*

type-synonym (**in** −) $('a, 'b)$ *comp-opt* $= 'a \Rightarrow 'b \Rightarrow$ (*order option*)

definition (**in** −) *lookup-dflt* :: $('a \times 'b)$ *list* $\Rightarrow 'a \Rightarrow 'b$::*zero*
where *lookup-dflt xs k* = (*case map-of xs k of Some v* $\Rightarrow v \mid$ *None* $\Rightarrow 0$)

lookup-dflt is only an auxiliary function needed for proving some lemmas.

fun *lookup-pair* :: $('a \times 'b)$ *list* $\Rightarrow 'a \Rightarrow 'b$::*zero*
where
lookup-pair [] *x* = *0*|
lookup-pair $((k, v) \# xs)\ x =$
(*case comp x k of*
Lt $\Rightarrow$ *0*
| *Eq* $\Rightarrow$ *v*
| *Gt* $\Rightarrow$ *lookup-pair xs x*)

fun *update-by-pair* :: $('a \times 'b) \Rightarrow ('a \times 'b)$ *list* $\Rightarrow ('a \times 'b$::*zero*$)$ *list*
where
update-by-pair (k, v) [] = (*if v* = *0 then* [] *else* $[(k, v)]$)
| *update-by-pair* $(k, v)\ ((k', v') \# xs) =$
(*case comp k k' of Lt* $\Rightarrow$ (*if v* = *0 then* $(k', v') \# xs$ *else* $(k, v) \# (k', v') \# xs$)
| *Eq* $\Rightarrow$ (*if v* = *0 then xs else* $(k, v) \# xs$)
| *Gt* $\Rightarrow (k', v') \#$ *update-by-pair* (k, v) *xs*)

definition *sort-oalist* :: $('a \times 'b)$ *list* $\Rightarrow ('a \times 'b$::*zero*$)$ *list*
where *sort-oalist xs* = *foldr update-by-pair xs* []

fun *update-by-fun-pair* :: $'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a \times 'b)$ *list* $\Rightarrow ('a \times 'b$::*zero*$)$ *list*
where
update-by-fun-pair k f [] = (*let v* = *f 0 in if v* = *0 then* [] *else* $[(k, v)]$)
| *update-by-fun-pair k f* $((k', v') \# xs) =$
(*case comp k k' of Lt* $\Rightarrow$ (*let v* = *f 0 in if v* = *0 then* $(k', v') \#$ *xs else* $(k, v) \# (k', v') \#$ *xs*)
| *Eq* $\Rightarrow$ (*let v* = *f v' in if v* = *0 then xs else* $(k, v) \# xs$)
| *Gt* $\Rightarrow (k', v') \#$ *update-by-fun-pair k f xs*)

definition *update-by-fun-gr-pair* :: *'a* ⇒ (*'b* ⇒ *'b*) ⇒ (*'a* × *'b*) *list* ⇒ (*'a* × *'b*::*zero*) *list*
where *update-by-fun-gr-pair k f xs* =
(*if xs* = [] *then*
(*let v* = *f* 0 *in if v* = 0 *then* [] *else* [(*k*, *v*)])
else if comp k (*fst* (*last xs*)) = *Gt then*
(*let v* = *f* 0 *in if v* = 0 *then xs else xs* @ [(*k*, *v*)])
else
update-by-fun-pair k f xs
)

fun (**in** −) *map-pair* :: ((*'a* × *'b*) ⇒ (*'a* × *'c*)) ⇒ (*'a* × *'b*::*zero*) *list* ⇒ (*'a* × *'c*::*zero*) *list*
where
map-pair f [] = []
| *map-pair f* (*kv* # *xs*) =
(*let* (*k*, *v*) = *f kv*; *aux* = *map-pair f xs in if v* = 0 *then aux else* (*k*, *v*) # *aux*)

The difference between *map* and *map-pair* is that the latter removes *0*::*'b* values, whereas the former does not.

abbreviation (**in** −) *map-val-pair* :: (*'a* ⇒ *'b* ⇒ *'c*) ⇒ (*'a* × *'b*::*zero*) *list* ⇒ (*'a* × *'c*::*zero*) *list*
where *map-val-pair f* ≡ *map-pair* (λ(*k*, *v*). (*k*, *f k v*))

fun *map2-val-pair* :: (*'a* ⇒ *'b* ⇒ *'c* ⇒ *'d*) ⇒ ((*'a* × *'b*) *list* ⇒ (*'a* × *'d*) *list*) ⇒
((*'a* × *'c*) *list* ⇒ (*'a* × *'d*) *list*) ⇒
(*'a* × *'b*::*zero*) *list* ⇒ (*'a* × *'c*::*zero*) *list* ⇒ (*'a* × *'d*::*zero*) *list*
where
map2-val-pair f g h xs [] = *g xs*
| *map2-val-pair f g h* [] *ys* = *h ys*
| *map2-val-pair f g h* ((*kx*, *vx*) # *xs*) ((*ky*, *vy*) # *ys*) =
(*case comp kx ky of*
Lt ⇒ (*let v* = *f kx vx* 0; *aux* = *map2-val-pair f g h xs* ((*ky*, *vy*) # *ys*) *in if v* = 0 *then aux else* (*kx*, *v*) # *aux*)
| *Eq* ⇒ (*let v* = *f kx vx vy*; *aux* = *map2-val-pair f g h xs ys in if v* = 0 *then aux else* (*kx*, *v*) # *aux*)
| *Gt* ⇒ (*let v* = *f ky* 0 *vy*; *aux* = *map2-val-pair f g h* ((*kx*, *vx*) # *xs*) *ys in if v* = 0 *then aux else* (*ky*, *v*) # *aux*))

fun *lex-ord-pair* :: (*'a* ⇒ ((*'b*, *'c*) *comp-opt*)) ⇒ ((*'a* × *'b*::*zero*) *list*, (*'a* × *'c*::*zero*) *list*) *comp-opt*
where
lex-ord-pair f [] [] = *Some Eq*|
lex-ord-pair f [] ((*ky*, *vy*) # *ys*) =
(*let aux* = *f ky* 0 *vy in if aux* = *Some Eq then lex-ord-pair f* [] *ys else aux*)|
lex-ord-pair f ((*kx*, *vx*) # *xs*) [] =
(*let aux* = *f kx vx* 0 *in if aux* = *Some Eq then lex-ord-pair f xs* [] *else aux*)|
lex-ord-pair f ((*kx*, *vx*) # *xs*) ((*ky*, *vy*) # *ys*) =
(*case comp kx ky of*

Lt ⇒ (*let aux* = *f kx vx 0 in if aux* = *Some Eq then lex-ord-pair f xs* ((*ky*, *vy*) # *ys*) *else aux*)
| *Eq* ⇒ (*let aux* = *f kx vx vy in if aux* = *Some Eq then lex-ord-pair f xs ys else aux*)
| *Gt* ⇒ (*let aux* = *f ky 0 vy in if aux* = *Some Eq then lex-ord-pair f* ((*kx*, *vx*) # *xs*) *ys else aux*))

fun *prod-ord-pair* :: ('*a* ⇒ '*b* ⇒ '*c* ⇒ *bool*) ⇒ ('*a* × '*b*::*zero*) *list* ⇒ ('*a* × '*c*::*zero*) *list* ⇒ *bool*
where
prod-ord-pair f [] [] = *True*|
prod-ord-pair f [] ((*ky*, *vy*) # *ys*) = (*f ky 0 vy* ∧ *prod-ord-pair f* [] *ys*)|
prod-ord-pair f ((*kx*, *vx*) # *xs*) [] = (*f kx vx 0* ∧ *prod-ord-pair f xs* [])|
prod-ord-pair f ((*kx*, *vx*) # *xs*) ((*ky*, *vy*) # *ys*) =
(*case comp kx ky of*
Lt ⇒ (*f kx vx 0* ∧ *prod-ord-pair f xs* ((*ky*, *vy*) # *ys*))
| *Eq* ⇒ (*f kx vx vy* ∧ *prod-ord-pair f xs ys*)
| *Gt* ⇒ (*f ky 0 vy* ∧ *prod-ord-pair f* ((*kx*, *vx*) # *xs*) *ys*))

prod-ord-pair is actually just a special case of *lex-ord-pair*, as proved below in lemma *prod-ord-pair-eq-lex-ord-pair*.

12.4.1 *lookup-pair*

lemma *lookup-pair-eq-0*:
assumes *oalist-inv-raw xs*
shows *lookup-pair xs k* = *0* ⟷ (*k* ∉ *fst* ‘ *set xs*)
⟨*proof*⟩

lemma *lookup-pair-eq-value*:
assumes *oalist-inv-raw xs* **and** *v* ≠ *0*
shows *lookup-pair xs k* = *v* ⟷ ((*k*, *v*) ∈ *set xs*)
⟨*proof*⟩

lemma *lookup-pair-eq-valueI*:
assumes *oalist-inv-raw xs* **and** (*k*, *v*) ∈ *set xs*
shows *lookup-pair xs k* = *v*
⟨*proof*⟩

lemma *lookup-dflt-eq-lookup-pair*:
assumes *oalist-inv-raw xs*
shows *lookup-dflt xs* = *lookup-pair xs*
⟨*proof*⟩

lemma *lookup-pair-inj*:
assumes *oalist-inv-raw xs* **and** *oalist-inv-raw ys* **and** *lookup-pair xs* = *lookup-pair ys*
shows *xs* = *ys*
⟨*proof*⟩

lemma *lookup-pair-tl*:
assumes *oalist-inv-raw xs*
shows *lookup-pair (tl xs) k = (if (∀ k'∈fst ‘ set xs. le k k') then 0 else lookup-pair xs k)*
⟨*proof*⟩

lemma *lookup-pair-tl'*:
assumes *oalist-inv-raw xs*
shows *lookup-pair (tl xs) k = (if k = fst (hd xs) then 0 else lookup-pair xs k)*
⟨*proof*⟩

lemma *lookup-pair-filter*:
assumes *oalist-inv-raw xs*
shows *lookup-pair (filter P xs) k = (let v = lookup-pair xs k in if P (k, v) then v else 0)*
⟨*proof*⟩

lemma *lookup-pair-map*:
assumes *oalist-inv-raw xs*
and *⋀k'. snd (f (k', 0)) = 0*
and *⋀a b. comp (fst (f a)) (fst (f b)) = comp (fst a) (fst b)*
shows *lookup-pair (map f xs) (fst (f (k, v))) = snd (f (k, lookup-pair xs k))*
⟨*proof*⟩

lemma *lookup-pair-Cons*:
assumes *oalist-inv-raw ((k, v) # xs)*
shows *lookup-pair ((k, v) # xs) k0 = (if k = k0 then v else lookup-pair xs k0)*
⟨*proof*⟩

lemma *lookup-pair-single*: *lookup-pair [(k, v)] k0 = (if k = k0 then v else 0)*
⟨*proof*⟩

12.4.2 *update-by-pair*

lemma *set-update-by-pair-subset*: *set (update-by-pair kv xs) ⊆ insert kv (set xs)*
⟨*proof*⟩

lemma *update-by-pair-sorted*:
assumes *sorted-wrt lt (map fst xs)*
shows *sorted-wrt lt (map fst (update-by-pair kv xs))*
⟨*proof*⟩

lemma *update-by-pair-not-0*:
assumes *0 ∉ snd ‘ set xs*
shows *0 ∉ snd ‘ set (update-by-pair kv xs)*
⟨*proof*⟩

corollary *oalist-inv-raw-update-by-pair*:

assumes *oalist-inv-raw xs*
shows *oalist-inv-raw* (*update-by-pair kv xs*)
⟨*proof*⟩

lemma *update-by-pair-less*:
assumes $v \neq 0$ **and** $xs = [] \vee comp\ k\ (fst\ (hd\ xs)) = Lt$
shows *update-by-pair* (k, v) *xs* = (k, v) # *xs*
⟨*proof*⟩

lemma *lookup-pair-update-by-pair*:
assumes *oalist-inv-raw xs*
shows *lookup-pair* (*update-by-pair* (*k1*, *v*) *xs*) *k2* = (*if k1* = *k2 then v else lookup-pair xs k2*)
⟨*proof*⟩

corollary *update-by-pair-id*:
assumes *oalist-inv-raw xs* **and** *lookup-pair xs k* = *v*
shows *update-by-pair* (k, v) *xs* = *xs*
⟨*proof*⟩

lemma *set-update-by-pair*:
assumes *oalist-inv-raw xs* **and** $v \neq 0$
shows *set* (*update-by-pair* (k, v) *xs*) = *insert* (k, v) (*set xs* − *range* (*Pair k*)) (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *set-update-by-pair-zero*:
assumes *oalist-inv-raw xs*
shows *set* (*update-by-pair* $(k, 0)$ *xs*) = *set xs* − *range* (*Pair k*) (**is** *?A* = *?B*)
⟨*proof*⟩

12.4.3 *update-by-fun-pair* and *update-by-fun-gr-pair*

lemma *update-by-fun-pair-eq-update-by-pair*:
assumes *oalist-inv-raw xs*
shows *update-by-fun-pair k f xs* = *update-by-pair* $(k, f\ (lookup\text{-}pair\ xs\ k))$ *xs*
⟨*proof*⟩

corollary *oalist-inv-raw-update-by-fun-pair*:
assumes *oalist-inv-raw xs*
shows *oalist-inv-raw* (*update-by-fun-pair k f xs*)
⟨*proof*⟩

corollary *lookup-pair-update-by-fun-pair*:
assumes *oalist-inv-raw xs*
shows *lookup-pair* (*update-by-fun-pair k1 f xs*) *k2* = (*if k1* = *k2 then f else id*) (*lookup-pair xs k2*)
⟨*proof*⟩

lemma *update-by-fun-pair-gr*:
assumes *oalist-inv-raw xs* **and** *xs* = [] ∨ *comp k* (*fst* (*last xs*)) = *Gt*
shows *update-by-fun-pair k f xs* = *xs* @ (*if f 0* = *0 then* [] *else* [(*k*, *f 0*)])
⟨*proof*⟩

corollary *update-by-fun-gr-pair-eq-update-by-fun-pair*:
assumes *oalist-inv-raw xs*
shows *update-by-fun-gr-pair k f xs* = *update-by-fun-pair k f xs*
⟨*proof*⟩

corollary *oalist-inv-raw-update-by-fun-gr-pair*:
assumes *oalist-inv-raw xs*
shows *oalist-inv-raw* (*update-by-fun-gr-pair k f xs*)
⟨*proof*⟩

corollary *lookup-pair-update-by-fun-gr-pair*:
assumes *oalist-inv-raw xs*
shows *lookup-pair* (*update-by-fun-gr-pair k1 f xs*) *k2* = (*if k1* = *k2 then f else id*) (*lookup-pair xs k2*)
⟨*proof*⟩

12.4.4 *map-pair*

lemma *map-pair-cong*:
assumes ⋀*kv*. *kv* ∈ *set xs* ⟹ *f kv* = *g kv*
shows *map-pair f xs* = *map-pair g xs*
⟨*proof*⟩

lemma *map-pair-subset*: *set* (*map-pair f xs*) ⊆ *f* ‘ *set xs*
⟨*proof*⟩

lemma *oalist-inv-raw-map-pair*:
assumes *oalist-inv-raw xs*
and ⋀*a b*. *comp* (*fst* (*f a*)) (*fst* (*f b*)) = *comp* (*fst a*) (*fst b*)
shows *oalist-inv-raw* (*map-pair f xs*)
⟨*proof*⟩

lemma *lookup-pair-map-pair*:
assumes *oalist-inv-raw xs* **and** *snd* (*f* (*k*, *0*)) = *0*
and ⋀*a b*. *comp* (*fst* (*f a*)) (*fst* (*f b*)) = *comp* (*fst a*) (*fst b*)
shows *lookup-pair* (*map-pair f xs*) (*fst* (*f* (*k*, *v*))) = *snd* (*f* (*k*, *lookup-pair xs k*))
⟨*proof*⟩

lemma *lookup-dflt-map-pair*:
assumes *distinct* (*map fst xs*) **and** *snd* (*f* (*k*, *0*)) = *0*
and ⋀*a b*. (*fst* (*f a*) = *fst* (*f b*)) ⟷ (*fst a* = *fst b*)
shows *lookup-dflt* (*map-pair f xs*) (*fst* (*f* (*k*, *v*))) = *snd* (*f* (*k*, *lookup-dflt xs k*))
⟨*proof*⟩

lemma *distinct-map-pair*:
assumes *distinct* (*map fst xs*) **and** $\bigwedge a\ b.$ *fst* (*f a*) = *fst* (*f b*) $\Longrightarrow$ *fst a* = *fst b*
shows *distinct* (*map fst* (*map-pair f xs*))
⟨*proof*⟩

lemma *map-val-pair-cong*:
assumes $\bigwedge k\ v.$ (*k*, *v*) ∈ *set xs* $\Longrightarrow$ *f k v* = *g k v*
shows *map-val-pair f xs* = *map-val-pair g xs*
⟨*proof*⟩

lemma *oalist-inv-raw-map-val-pair*:
assumes *oalist-inv-raw xs*
shows *oalist-inv-raw* (*map-val-pair f xs*)
⟨*proof*⟩

lemma *lookup-pair-map-val-pair*:
assumes *oalist-inv-raw xs* **and** *f k 0* = *0*
shows *lookup-pair* (*map-val-pair f xs*) *k* = *f k* (*lookup-pair xs k*)
⟨*proof*⟩

lemma *map-pair-id*:
assumes *oalist-inv-raw xs*
shows *map-pair id xs* = *xs*
⟨*proof*⟩

12.4.5 *map2-val-pair*

definition *map2-val-compat* :: (('*a* × '*b*::*zero*) *list* ⇒ ('*a* × '*c*::*zero*) *list*) ⇒ *bool*
where *map2-val-compat f* ⟷ (∀ *zs*. (*oalist-inv-raw zs* ⟶
(*oalist-inv-raw* (*f zs*) ∧ *fst* ‘ *set* (*f zs*) ⊆ *fst* ‘ *set zs*)))

lemma *map2-val-compatI*:
assumes $\bigwedge zs.$ *oalist-inv-raw zs* $\Longrightarrow$ *oalist-inv-raw* (*f zs*)
and $\bigwedge zs.$ *oalist-inv-raw zs* $\Longrightarrow$ *fst* ‘ *set* (*f zs*) ⊆ *fst* ‘ *set zs*
shows *map2-val-compat f*
⟨*proof*⟩

lemma *map2-val-compatD1*:
assumes *map2-val-compat f* **and** *oalist-inv-raw zs*
shows *oalist-inv-raw* (*f zs*)
⟨*proof*⟩

lemma *map2-val-compatD2*:
assumes *map2-val-compat f* **and** *oalist-inv-raw zs*
shows *fst* ‘ *set* (*f zs*) ⊆ *fst* ‘ *set zs*
⟨*proof*⟩

lemma *map2-val-compat-Nil*:
assumes *map2-val-compat* (*f*::('*a* × '*b*::*zero*) *list* ⇒ ('*a* × '*c*::*zero*) *list*)

shows $f\ [] = []$
⟨*proof*⟩

lemma *map2-val-compat-id*: *map2-val-compat id*
⟨*proof*⟩

lemma *map2-val-compat-map-val-pair*: *map2-val-compat* (*map-val-pair f*)
⟨*proof*⟩

lemma *fst-map2-val-pair-subset*:
assumes *oalist-inv-raw xs* **and** *oalist-inv-raw ys*
assumes *map2-val-compat g* **and** *map2-val-compat h*
shows *fst ‘ set* (*map2-val-pair f g h xs ys*) ⊆ *fst ‘ set xs* ∪ *fst ‘ set ys*
⟨*proof*⟩

lemma *oalist-inv-raw-map2-val-pair*:
assumes *oalist-inv-raw xs* **and** *oalist-inv-raw ys*
assumes *map2-val-compat g* **and** *map2-val-compat h*
shows *oalist-inv-raw* (*map2-val-pair f g h xs ys*)
⟨*proof*⟩

lemma *lookup-pair-map2-val-pair*:
assumes *oalist-inv-raw xs* **and** *oalist-inv-raw ys*
assumes *map2-val-compat g* **and** *map2-val-compat h*
assumes ⋀*zs*. *oalist-inv-raw zs* ⟹ *g zs* = *map-val-pair* (λ*k v*. *f k v 0*) *zs*
and ⋀*zs*. *oalist-inv-raw zs* ⟹ *h zs* = *map-val-pair* (λ*k*. *f k 0*) *zs*
and ⋀*k*. *f k 0 0* = *0*
shows *lookup-pair* (*map2-val-pair f g h xs ys*) *k0* = *f k0* (*lookup-pair xs k0*) (*lookup-pair ys k0*)
⟨*proof*⟩

lemma *map2-val-pair-singleton-eq-update-by-fun-pair*:
assumes *oalist-inv-raw xs*
assumes ⋀*k x*. *f k x 0* = *x* **and** ⋀*zs*. *oalist-inv-raw zs* ⟹ *g zs* = *zs*
and *h* [(*k*, *v*)] = *map-val-pair* (λ*k*. *f k 0*) [(*k*, *v*)]
shows *map2-val-pair f g h xs* [(*k*, *v*)] = *update-by-fun-pair k* (λ*x*. *f k x v*) *xs*
⟨*proof*⟩

12.4.6 *lex-ord-pair*

lemma *lex-ord-pair-EqI*:
assumes *oalist-inv-raw xs* **and** *oalist-inv-raw ys*
and ⋀*k*. *k* ∈ *fst ‘ set xs* ∪ *fst ‘ set ys* ⟹ *f k* (*lookup-pair xs k*) (*lookup-pair ys k*) = *Some Eq*
shows *lex-ord-pair f xs ys* = *Some Eq*
⟨*proof*⟩

lemma *lex-ord-pair-valI*:
assumes *oalist-inv-raw xs* **and** *oalist-inv-raw ys* **and** *aux* ≠ *Some Eq*

assumes $k \in fst\ `\ set\ xs \cup fst\ `\ set\ ys$ **and** $aux = f\ k\ (lookup\text{-}pair\ xs\ k)\ (lookup\text{-}pair\ ys\ k)$
and $\bigwedge k'.\ k' \in fst\ `\ set\ xs \cup fst\ `\ set\ ys \Longrightarrow lt\ k'\ k \Longrightarrow f\ k'\ (lookup\text{-}pair\ xs\ k')\ (lookup\text{-}pair\ ys\ k') = Some\ Eq$
shows $lex\text{-}ord\text{-}pair\ f\ xs\ ys = aux$
⟨*proof*⟩

lemma *lex-ord-pair-EqD*:
assumes *oalist-inv-raw xs* **and** *oalist-inv-raw ys* **and** $lex\text{-}ord\text{-}pair\ f\ xs\ ys = Some\ Eq$
and $k \in fst\ `\ set\ xs \cup fst\ `\ set\ ys$
shows $f\ k\ (lookup\text{-}pair\ xs\ k)\ (lookup\text{-}pair\ ys\ k) = Some\ Eq$
⟨*proof*⟩

lemma *lex-ord-pair-valE*:
assumes *oalist-inv-raw xs* **and** *oalist-inv-raw ys* **and** $lex\text{-}ord\text{-}pair\ f\ xs\ ys = aux$
and $aux \neq Some\ Eq$
obtains k **where** $k \in fst\ `\ set\ xs \cup fst\ `\ set\ ys$ **and** $aux = f\ k\ (lookup\text{-}pair\ xs\ k)\ (lookup\text{-}pair\ ys\ k)$
and $\bigwedge k'.\ k' \in fst\ `\ set\ xs \cup fst\ `\ set\ ys \Longrightarrow lt\ k'\ k \Longrightarrow f\ k'\ (lookup\text{-}pair\ xs\ k')\ (lookup\text{-}pair\ ys\ k') = Some\ Eq$
⟨*proof*⟩

12.4.7 *prod-ord-pair*

lemma *prod-ord-pair-eq-lex-ord-pair*:
$prod\text{-}ord\text{-}pair\ P\ xs\ ys = (lex\text{-}ord\text{-}pair\ (\lambda k\ x\ y.\ if\ P\ k\ x\ y\ then\ Some\ Eq\ else\ None)\ xs\ ys = Some\ Eq)$
⟨*proof*⟩

lemma *prod-ord-pairI*:
assumes *oalist-inv-raw xs* **and** *oalist-inv-raw ys*
and $\bigwedge k.\ k \in fst\ `\ set\ xs \cup fst\ `\ set\ ys \Longrightarrow P\ k\ (lookup\text{-}pair\ xs\ k)\ (lookup\text{-}pair\ ys\ k)$
shows $prod\text{-}ord\text{-}pair\ P\ xs\ ys$
⟨*proof*⟩

lemma *prod-ord-pairD*:
assumes *oalist-inv-raw xs* **and** *oalist-inv-raw ys* **and** $prod\text{-}ord\text{-}pair\ P\ xs\ ys$
and $k \in fst\ `\ set\ xs \cup fst\ `\ set\ ys$
shows $P\ k\ (lookup\text{-}pair\ xs\ k)\ (lookup\text{-}pair\ ys\ k)$
⟨*proof*⟩

corollary *prod-ord-pair-alt*:
assumes *oalist-inv-raw xs* **and** *oalist-inv-raw ys*
shows $(prod\text{-}ord\text{-}pair\ P\ xs\ ys) \longleftrightarrow (\forall k \in fst\ `\ set\ xs \cup fst\ `\ set\ ys.\ P\ k\ (lookup\text{-}pair\ xs\ k)\ (lookup\text{-}pair\ ys\ k))$
⟨*proof*⟩

12.4.8 *sort-oalist*

lemma *oalist-inv-raw-foldr-update-by-pair*:
assumes *oalist-inv-raw ys*
shows *oalist-inv-raw* (*foldr update-by-pair xs ys*)
⟨*proof*⟩

corollary *oalist-inv-raw-sort-oalist*: *oalist-inv-raw* (*sort-oalist xs*)
⟨*proof*⟩

lemma *sort-oalist-id*:
assumes *oalist-inv-raw xs*
shows *sort-oalist xs* = *xs*
⟨*proof*⟩

lemma *set-sort-oalist*:
assumes *distinct* (*map fst xs*)
shows *set* (*sort-oalist xs*) = {*kv*. *kv* ∈ *set xs* ∧ *snd kv* ≠ *0*}
⟨*proof*⟩

lemma *lookup-pair-sort-oalist′*:
assumes *distinct* (*map fst xs*)
shows *lookup-pair* (*sort-oalist xs*) = *lookup-dflt xs*
⟨*proof*⟩

end

locale *comparator2* = *comparator comp1* + *cmp2*: *comparator comp2* **for** *comp1 comp2* :: *′a comparator*
begin

lemma *set-sort-oalist*:
assumes *cmp2.oalist-inv-raw xs*
shows *set* (*sort-oalist xs*) = *set xs*
⟨*proof*⟩

lemma *lookup-pair-eqI*:
assumes *oalist-inv-raw xs* **and** *cmp2.oalist-inv-raw ys* **and** *set xs* = *set ys*
shows *lookup-pair xs* = *cmp2.lookup-pair ys*
⟨*proof*⟩

corollary *lookup-pair-sort-oalist*:
assumes *cmp2.oalist-inv-raw xs*
shows *lookup-pair* (*sort-oalist xs*) = *cmp2.lookup-pair xs*
⟨*proof*⟩

end

12.5 Invariant on Pairs

type-synonym $('a, 'b, 'c)$ *oalist-raw* $= ('a \times 'b)$ *list* $\times 'c$

locale *oalist-raw* = **fixes** *rep-key-order*::$'o \Rightarrow 'a$ *key-order*
begin

sublocale *comparator key-compare* (*rep-key-order x*)
⟨*proof*⟩

definition *oalist-inv* :: $('a, 'b$::*zero*, $'o)$ *oalist-raw* $\Rightarrow$ *bool*
where *oalist-inv xs* $\longleftrightarrow$ *oalist-inv-raw* (*snd xs*) (*fst xs*)

lemma *oalist-inv-alt*: *oalist-inv* $(xs, ko) \longleftrightarrow$ *oalist-inv-raw ko xs*
⟨*proof*⟩

12.6 Operations on Raw Ordered Associative Lists

fun *sort-oalist-aux* :: $'o \Rightarrow ('a, 'b, 'o)$ *oalist-raw* $\Rightarrow ('a \times 'b$::*zero*$)$ *list*
where *sort-oalist-aux ko* $(xs, ox) =$ (*if ko* $=$ *ox then xs else sort-oalist ko xs*)

fun *lookup-raw* :: $('a, 'b, 'o)$ *oalist-raw* $\Rightarrow 'a \Rightarrow 'b$::*zero*
where *lookup-raw* $(xs, ko) =$ *lookup-pair ko xs*

definition *sorted-domain-raw* :: $'o \Rightarrow ('a, 'b$::*zero*, $'o)$ *oalist-raw* $\Rightarrow 'a$ *list*
where *sorted-domain-raw ko xs* $=$ *map fst* (*sort-oalist-aux ko xs*)

fun *tl-raw* :: $('a, 'b, 'o)$ *oalist-raw* $\Rightarrow ('a, 'b$::*zero*, $'o)$ *oalist-raw*
where *tl-raw* $(xs, ko) =$ (*List.tl xs*, *ko*)

fun *min-key-val-raw* :: $'o \Rightarrow ('a, 'b, 'o)$ *oalist-raw* $\Rightarrow ('a \times 'b$::*zero*$)$
where *min-key-val-raw ko* $(xs, ox) =$
(*if ko* $=$ *ox then List.hd else min-list-param* $(\lambda x\ y.$ *le ko* (*fst x*) (*fst y*))) *xs*

fun *update-by-raw* :: $('a \times 'b) \Rightarrow ('a, 'b, 'o)$ *oalist-raw* $\Rightarrow ('a, 'b$::*zero*, $'o)$ *oalist-raw*
where *update-by-raw kv* $(xs, ko) =$ (*update-by-pair ko kv xs*, *ko*)

fun *update-by-fun-raw* :: $'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b, 'o)$ *oalist-raw* $\Rightarrow ('a, 'b$::*zero*, $'o)$ *oalist-raw*
where *update-by-fun-raw k f* $(xs, ko) =$ (*update-by-fun-pair ko k f xs*, *ko*)

fun *update-by-fun-gr-raw* :: $'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b, 'o)$ *oalist-raw* $\Rightarrow ('a, 'b$::*zero*, $'o)$ *oalist-raw*
where *update-by-fun-gr-raw k f* $(xs, ko) =$ (*update-by-fun-gr-pair ko k f xs*, *ko*)

fun (**in** −) *filter-raw* :: $('a \Rightarrow$ *bool*$) \Rightarrow ('a$ *list* $\times 'b) \Rightarrow ('a$ *list* $\times 'b)$
where *filter-raw P* $(xs, ko) =$ (*filter P xs*, *ko*)

fun (**in** −) *map-raw* :: $((('a \times 'b) \Rightarrow ('a \times 'c)) \Rightarrow (('a \times 'b$::*zero*$)$ *list* $\times 'd) \Rightarrow ('a$ $\times 'c$::*zero*$)$ *list* $\times 'd$

where *map-raw f* (*xs*, *ko*) = (*map-pair f xs*, *ko*)

abbreviation (**in** −) *map-val-raw f* ≡ *map-raw* (λ(*k*, *v*). (*k*, *f k v*))

fun *map2-val-raw* :: ('*a* ⇒ '*b* ⇒ '*c* ⇒ '*d*) ⇒ (('*a*, '*b*, '*o*) *oalist-raw* ⇒ ('*a*, '*d*, '*o*) *oalist-raw*) ⇒
('*a*, '*c*, '*o*) *oalist-raw* ⇒ ('*a*, '*d*, '*o*) *oalist-raw*) ⇒
('*a*, '*b*::*zero*, '*o*) *oalist-raw* ⇒ ('*a*, '*c*::*zero*, '*o*) *oalist-raw* ⇒
('*a*, '*d*::*zero*, '*o*) *oalist-raw*
where *map2-val-raw f g h* (*xs*, *ox*) *ys* =
(*map2-val-pair ox f* (λ*zs*. *fst* (*g* (*zs*, *ox*))) (λ*zs*. *fst* (*h* (*zs*, *ox*)))
xs (*sort-oalist-aux ox ys*), *ox*)

definition *lex-ord-raw* :: '*o* ⇒ ('*a* ⇒ (('*b*, '*c*) *comp-opt*)) ⇒
(('*a*, '*b*::*zero*, '*o*) *oalist-raw*, ('*a*, '*c*::*zero*, '*o*) *oalist-raw*) *comp-opt*
where *lex-ord-raw ko f xs ys* = *lex-ord-pair ko f* (*sort-oalist-aux ko xs*) (*sort-oalist-aux ko ys*)

fun *prod-ord-raw* :: ('*a* ⇒ '*b* ⇒ '*c* ⇒ *bool*) ⇒ ('*a*, '*b*::*zero*, '*o*) *oalist-raw* ⇒
('*a*, '*c*::*zero*, '*o*) *oalist-raw* ⇒ *bool*
where *prod-ord-raw f* (*xs*, *ox*) *ys* = *prod-ord-pair ox f xs* (*sort-oalist-aux ox ys*)

fun *oalist-eq-raw* :: ('*a*, '*b*, '*o*) *oalist-raw* ⇒ ('*a*, '*b*::*zero*, '*o*) *oalist-raw* ⇒ *bool*
where *oalist-eq-raw* (*xs*, *ox*) *ys* = (*xs* = (*sort-oalist-aux ox ys*))

fun *sort-oalist-raw* :: ('*a*, '*b*, '*o*) *oalist-raw* ⇒ ('*a*, '*b*::*zero*, '*o*) *oalist-raw*
where *sort-oalist-raw* (*xs*, *ko*) = (*sort-oalist ko xs*, *ko*)

12.6.1 *sort-oalist-aux*

lemma *set-sort-oalist-aux*:
assumes *oalist-inv xs*
shows *set* (*sort-oalist-aux ko xs*) = *set* (*fst xs*)
⟨*proof*⟩

lemma *oalist-inv-raw-sort-oalist-aux*:
assumes *oalist-inv xs*
shows *oalist-inv-raw ko* (*sort-oalist-aux ko xs*)
⟨*proof*⟩

lemma *oalist-inv-sort-oalist-aux*:
assumes *oalist-inv xs*
shows *oalist-inv* (*sort-oalist-aux ko xs*, *ko*)
⟨*proof*⟩

lemma *lookup-pair-sort-oalist-aux*:
assumes *oalist-inv xs*
shows *lookup-pair ko* (*sort-oalist-aux ko xs*) = *lookup-raw xs*
⟨*proof*⟩

12.6.2 *lookup-raw*

lemma *lookup-raw-eq-value*:
assumes *oalist-inv xs* **and** $v \neq 0$
shows *lookup-raw xs k* $= v \longleftrightarrow ((k, v) \in set\ (fst\ xs))$
⟨*proof*⟩

lemma *lookup-raw-eq-valueI*:
assumes *oalist-inv xs* **and** $(k, v) \in set\ (fst\ xs)$
shows *lookup-raw xs k* $= v$
⟨*proof*⟩

lemma *lookup-raw-inj*:
assumes *oalist-inv* (xs, ko) **and** *oalist-inv* (ys, ko) **and** *lookup-raw* $(xs, ko) =$ *lookup-raw* (ys, ko)
shows $xs = ys$
⟨*proof*⟩

12.6.3 *sorted-domain-raw*

lemma *set-sorted-domain-raw*:
assumes *oalist-inv xs*
shows *set* (*sorted-domain-raw ko xs*) = *fst* ‘ *set* (*fst xs*)
⟨*proof*⟩

corollary *in-sorted-domain-raw-iff-lookup-raw*:
assumes *oalist-inv xs*
shows $k \in set\ (sorted\text{-}domain\text{-}raw\ ko\ xs) \longleftrightarrow (lookup\text{-}raw\ xs\ k \neq 0)$
⟨*proof*⟩

lemma *sorted-sorted-domain-raw*:
assumes *oalist-inv xs*
shows *sorted-wrt* (*lt-of-key-order* (*rep-key-order ko*)) (*sorted-domain-raw ko xs*)
⟨*proof*⟩

12.6.4 *tl-raw*

lemma *oalist-inv-tl-raw*:
assumes *oalist-inv xs*
shows *oalist-inv* (*tl-raw xs*)
⟨*proof*⟩

lemma *lookup-raw-tl-raw*:
assumes *oalist-inv xs*
shows *lookup-raw* (*tl-raw xs*) *k* =
(*if* $(\forall k' \in fst\ ‘\ set\ (fst\ xs).\ le\ (snd\ xs)\ k\ k')$ *then 0 else lookup-raw xs k*)
⟨*proof*⟩

lemma *lookup-raw-tl-raw′*:
assumes *oalist-inv xs*

shows *lookup-raw* (*tl-raw xs*) *k* = (*if k* = *fst* (*List.hd* (*fst xs*)) *then 0 else lookup-raw xs k*)
⟨*proof*⟩

12.6.5 *min-key-val-raw*

lemma *min-key-val-raw-alt*:
assumes *oalist-inv xs* **and** *fst xs* ≠ []
shows *min-key-val-raw ko xs* = *List.hd* (*sort-oalist-aux ko xs*)
⟨*proof*⟩

lemma *min-key-val-raw-in*:
assumes *fst xs* ≠ []
shows *min-key-val-raw ko xs* ∈ *set* (*fst xs*)
⟨*proof*⟩

lemma *snd-min-key-val-raw*:
assumes *oalist-inv xs* **and** *fst xs* ≠ []
shows *snd* (*min-key-val-raw ko xs*) = *lookup-raw xs* (*fst* (*min-key-val-raw ko xs*))
⟨*proof*⟩

lemma *min-key-val-raw-minimal*:
assumes *oalist-inv xs* **and** *z* ∈ *set* (*fst xs*)
shows *le ko* (*fst* (*min-key-val-raw ko xs*)) (*fst z*)
⟨*proof*⟩

12.6.6 *filter-raw*

lemma *oalist-inv-filter-raw*:
assumes *oalist-inv xs*
shows *oalist-inv* (*filter-raw P xs*)
⟨*proof*⟩

lemma *lookup-raw-filter-raw*:
assumes *oalist-inv xs*
shows *lookup-raw* (*filter-raw P xs*) *k* = (*let v* = *lookup-raw xs k in if P* (*k*, *v*) *then v else 0*)
⟨*proof*⟩

12.6.7 *update-by-raw*

lemma *oalist-inv-update-by-raw*:
assumes *oalist-inv xs*
shows *oalist-inv* (*update-by-raw kv xs*)
⟨*proof*⟩

lemma *lookup-raw-update-by-raw*:
assumes *oalist-inv xs*
shows *lookup-raw* (*update-by-raw* (*k1*, *v*) *xs*) *k2* = (*if k1* = *k2 then v else lookup-raw xs k2*)

⟨*proof*⟩

12.6.8 *update-by-fun-raw* and *update-by-fun-gr-raw*

lemma *update-by-fun-raw-eq-update-by-raw*:
assumes *oalist-inv xs*
shows *update-by-fun-raw k f xs* = *update-by-raw* (*k*, *f* (*lookup-raw xs k*)) *xs*
⟨*proof*⟩

corollary *oalist-inv-update-by-fun-raw*:
assumes *oalist-inv xs*
shows *oalist-inv* (*update-by-fun-raw k f xs*)
⟨*proof*⟩

corollary *lookup-raw-update-by-fun-raw*:
assumes *oalist-inv xs*
shows *lookup-raw* (*update-by-fun-raw k1 f xs*) *k2* = (*if k1* = *k2 then f else id*) (*lookup-raw xs k2*)
⟨*proof*⟩

lemma *update-by-fun-gr-raw-eq-update-by-fun-raw*:
assumes *oalist-inv xs*
shows *update-by-fun-gr-raw k f xs* = *update-by-fun-raw k f xs*
⟨*proof*⟩

corollary *oalist-inv-update-by-fun-gr-raw*:
assumes *oalist-inv xs*
shows *oalist-inv* (*update-by-fun-gr-raw k f xs*)
⟨*proof*⟩

corollary *lookup-raw-update-by-fun-gr-raw*:
assumes *oalist-inv xs*
shows *lookup-raw* (*update-by-fun-gr-raw k1 f xs*) *k2* = (*if k1* = *k2 then f else id*) (*lookup-raw xs k2*)
⟨*proof*⟩

12.6.9 *map-raw* and *map-val-raw*

lemma *map-raw-cong*:
assumes ⋀*kv*. *kv* ∈ *set* (*fst xs*) ⟹ *f kv* = *g kv*
shows *map-raw f xs* = *map-raw g xs*
⟨*proof*⟩

lemma *map-raw-subset*: *set* (*fst* (*map-raw f xs*)) ⊆ *f* ‘ *set* (*fst xs*)
⟨*proof*⟩

lemma *oalist-inv-map-raw*:
assumes *oalist-inv xs*
and ⋀*a b*. *key-compare* (*rep-key-order* (*snd xs*)) (*fst* (*f a*)) (*fst* (*f b*)) = *key-compare* (*rep-key-order* (*snd xs*)) (*fst a*) (*fst b*)

shows *oalist-inv* (*map-raw f xs*)
⟨*proof*⟩

lemma *lookup-raw-map-raw*:
assumes *oalist-inv xs* **and** *snd* (*f* (*k*, *0*)) = *0*
and ⋀*a b*. *key-compare* (*rep-key-order* (*snd xs*)) (*fst* (*f a*)) (*fst* (*f b*)) = *key-compare* (*rep-key-order* (*snd xs*)) (*fst a*) (*fst b*)
shows *lookup-raw* (*map-raw f xs*) (*fst* (*f* (*k*, *v*))) = *snd* (*f* (*k*, *lookup-raw xs k*))
⟨*proof*⟩

lemma *map-raw-id*:
assumes *oalist-inv xs*
shows *map-raw id xs* = *xs*
⟨*proof*⟩

lemma *map-val-raw-cong*:
assumes ⋀*k v*. (*k*, *v*) ∈ *set* (*fst xs*) ⟹ *f k v* = *g k v*
shows *map-val-raw f xs* = *map-val-raw g xs*
⟨*proof*⟩

lemma *oalist-inv-map-val-raw*:
assumes *oalist-inv xs*
shows *oalist-inv* (*map-val-raw f xs*)
⟨*proof*⟩

lemma *lookup-raw-map-val-raw*:
assumes *oalist-inv xs* **and** *f k 0* = *0*
shows *lookup-raw* (*map-val-raw f xs*) *k* = *f k* (*lookup-raw xs k*)
⟨*proof*⟩

12.6.10 *map2-val-raw*

definition *map2-val-compat′* :: ((*'a*, *'b*::*zero*, *'o*) *oalist-raw* ⇒ (*'a*, *'c*::*zero*, *'o*) *oalist-raw*) ⇒ *bool*
where *map2-val-compat′ f* ⟷
(∀ *zs*. (*oalist-inv zs* ⟶ (*oalist-inv* (*f zs*) ∧ *snd* (*f zs*) = *snd zs* ∧ *fst* ‘ *set* (*fst* (*f zs*)) ⊆ *fst* ‘ *set* (*fst zs*))))

lemma *map2-val-compat′I*:
assumes ⋀*zs*. *oalist-inv zs* ⟹ *oalist-inv* (*f zs*)
and ⋀*zs*. *oalist-inv zs* ⟹ *snd* (*f zs*) = *snd zs*
and ⋀*zs*. *oalist-inv zs* ⟹ *fst* ‘ *set* (*fst* (*f zs*)) ⊆ *fst* ‘ *set* (*fst zs*)
shows *map2-val-compat′ f*
⟨*proof*⟩

lemma *map2-val-compat′D1*:
assumes *map2-val-compat′ f* **and** *oalist-inv zs*
shows *oalist-inv* (*f zs*)
⟨*proof*⟩

lemma *map2-val-compat'D2*:
assumes *map2-val-compat' f* **and** *oalist-inv zs*
shows *snd* (*f zs*) = *snd zs*
⟨*proof*⟩

lemma *map2-val-compat'D3*:
assumes *map2-val-compat' f* **and** *oalist-inv zs*
shows *fst* ‘ *set* (*fst* (*f zs*)) ⊆ *fst* ‘ *set* (*fst zs*)
⟨*proof*⟩

lemma *map2-val-compat'-map-val-raw*: *map2-val-compat'* (*map-val-raw f*)
⟨*proof*⟩

lemma *map2-val-compat'-id*: *map2-val-compat' id*
⟨*proof*⟩

lemma *map2-val-compat'-imp-map2-val-compat*:
assumes *map2-val-compat' g*
shows *map2-val-compat ko* (λ*zs*. *fst* (*g* (*zs*, *ko*)))
⟨*proof*⟩

lemma *oalist-inv-map2-val-raw*:
assumes *oalist-inv xs* **and** *oalist-inv ys*
assumes *map2-val-compat' g* **and** *map2-val-compat' h*
shows *oalist-inv* (*map2-val-raw f g h xs ys*)
⟨*proof*⟩

lemma *lookup-raw-map2-val-raw*:
assumes *oalist-inv xs* **and** *oalist-inv ys*
assumes *map2-val-compat' g* **and** *map2-val-compat' h*
assumes ⋀*zs*. *oalist-inv zs* ⟹ *g zs* = *map-val-raw* (λ*k v*. *f k v 0*) *zs*
and ⋀*zs*. *oalist-inv zs* ⟹ *h zs* = *map-val-raw* (λ*k*. *f k 0*) *zs*
and ⋀*k*. *f k 0 0* = *0*
shows *lookup-raw* (*map2-val-raw f g h xs ys*) *k0* = *f k0* (*lookup-raw xs k0*)
(*lookup-raw ys k0*)
⟨*proof*⟩

lemma *map2-val-raw-singleton-eq-update-by-fun-raw*:
assumes *oalist-inv xs*
assumes ⋀*k x*. *f k x 0* = *x* **and** ⋀*zs*. *oalist-inv zs* ⟹ *g zs* = *zs*
and ⋀*ko*. *h* ([(*k*, *v*)], *ko*) = *map-val-raw* (λ*k*. *f k 0*) ([(*k*, *v*)], *ko*)
shows *map2-val-raw f g h xs* ([(*k*, *v*)], *ko*) = *update-by-fun-raw k* (λ*x*. *f k x v*) *xs*
⟨*proof*⟩

12.6.11 *lex-ord-raw*

lemma *lex-ord-raw-EqI*:
assumes *oalist-inv xs* **and** *oalist-inv ys*

and $\bigwedge k.\ k \in fst\ `\ set\ (fst\ xs) \cup fst\ `\ set\ (fst\ ys) \Longrightarrow f\ k\ (lookup\text{-}raw\ xs\ k)\ (lookup\text{-}raw\ ys\ k) = Some\ Eq$
shows *lex-ord-raw ko f xs ys = Some Eq*
⟨*proof*⟩

lemma *lex-ord-raw-valI*:
assumes *oalist-inv xs* **and** *oalist-inv ys* **and** $aux \neq Some\ Eq$
assumes $k \in fst\ `\ set\ (fst\ xs) \cup fst\ `\ set\ (fst\ ys)$ **and** $aux = f\ k\ (lookup\text{-}raw\ xs\ k)\ (lookup\text{-}raw\ ys\ k)$
and $\bigwedge k'.\ k' \in fst\ `\ set\ (fst\ xs) \cup fst\ `\ set\ (fst\ ys) \Longrightarrow lt\ ko\ k'\ k \Longrightarrow f\ k'\ (lookup\text{-}raw\ xs\ k')\ (lookup\text{-}raw\ ys\ k') = Some\ Eq$
shows *lex-ord-raw ko f xs ys = aux*
⟨*proof*⟩

lemma *lex-ord-raw-EqD*:
assumes *oalist-inv xs* **and** *oalist-inv ys* **and** *lex-ord-raw ko f xs ys = Some Eq*
and $k \in fst\ `\ set\ (fst\ xs) \cup fst\ `\ set\ (fst\ ys)$
shows $f\ k\ (lookup\text{-}raw\ xs\ k)\ (lookup\text{-}raw\ ys\ k) = Some\ Eq$
⟨*proof*⟩

lemma *lex-ord-raw-valE*:
assumes *oalist-inv xs* **and** *oalist-inv ys* **and** *lex-ord-raw ko f xs ys = aux*
and $aux \neq Some\ Eq$
obtains k **where** $k \in fst\ `\ set\ (fst\ xs) \cup fst\ `\ set\ (fst\ ys)$
and $aux = f\ k\ (lookup\text{-}raw\ xs\ k)\ (lookup\text{-}raw\ ys\ k)$
and $\bigwedge k'.\ k' \in fst\ `\ set\ (fst\ xs) \cup fst\ `\ set\ (fst\ ys) \Longrightarrow lt\ ko\ k'\ k \Longrightarrow f\ k'\ (lookup\text{-}raw\ xs\ k')\ (lookup\text{-}raw\ ys\ k') = Some\ Eq$
⟨*proof*⟩

12.6.12 *prod-ord-raw*

lemma *prod-ord-rawI*:
assumes *oalist-inv xs* **and** *oalist-inv ys*
and $\bigwedge k.\ k \in fst\ `\ set\ (fst\ xs) \cup fst\ `\ set\ (fst\ ys) \Longrightarrow P\ k\ (lookup\text{-}raw\ xs\ k)\ (lookup\text{-}raw\ ys\ k)$
shows *prod-ord-raw P xs ys*
⟨*proof*⟩

lemma *prod-ord-rawD*:
assumes *oalist-inv xs* **and** *oalist-inv ys* **and** *prod-ord-raw P xs ys*
and $k \in fst\ `\ set\ (fst\ xs) \cup fst\ `\ set\ (fst\ ys)$
shows $P\ k\ (lookup\text{-}raw\ xs\ k)\ (lookup\text{-}raw\ ys\ k)$
⟨*proof*⟩

corollary *prod-ord-raw-alt*:
assumes *oalist-inv xs* **and** *oalist-inv ys*
shows *prod-ord-raw P xs ys* $\longleftrightarrow$
$(\forall k \in fst\ `\ set\ (fst\ xs) \cup fst\ `\ set\ (fst\ ys).\ P\ k\ (lookup\text{-}raw\ xs\ k)\ (lookup\text{-}raw\ ys\ k))$

⟨*proof*⟩

12.6.13 *oalist-eq-raw*

lemma *oalist-eq-rawI*:
assumes *oalist-inv xs* **and** *oalist-inv ys*
and $\bigwedge k.\ k \in fst\ `\ set\ (fst\ xs) \cup fst\ `\ set\ (fst\ ys) \Longrightarrow lookup\text{-}raw\ xs\ k = lookup\text{-}raw\ ys\ k$
shows *oalist-eq-raw xs ys*
⟨*proof*⟩

lemma *oalist-eq-rawD*:
assumes *oalist-inv ys* **and** *oalist-eq-raw xs ys*
shows *lookup-raw xs* = *lookup-raw ys*
⟨*proof*⟩

lemma *oalist-eq-raw-alt*:
assumes *oalist-inv xs* **and** *oalist-inv ys*
shows *oalist-eq-raw xs ys* $\longleftrightarrow$ (*lookup-raw xs* = *lookup-raw ys*)
⟨*proof*⟩

12.6.14 *sort-oalist-raw*

lemma *oalist-inv-sort-oalist-raw*: *oalist-inv* (*sort-oalist-raw xs*)
⟨*proof*⟩

lemma *sort-oalist-raw-id*:
assumes *oalist-inv xs*
shows *sort-oalist-raw xs* = *xs*
⟨*proof*⟩

lemma *set-sort-oalist-raw*:
assumes *distinct* (*map fst* (*fst xs*))
shows *set* (*fst* (*sort-oalist-raw xs*)) = $\{kv.\ kv \in set\ (fst\ xs) \wedge snd\ kv \neq 0\}$
⟨*proof*⟩

end

12.7 Fundamental Operations on One List

locale *oalist-abstract* = *oalist-raw rep-key-order* **for** *rep-key-order*::$'o \Rightarrow 'a$ *key-order*
+
fixes *list-of-oalist* :: $'x \Rightarrow ('a, 'b{::}zero, 'o)$ *oalist-raw*
fixes *oalist-of-list* :: $('a, 'b, 'o)$ *oalist-raw* $\Rightarrow 'x$
assumes *oalist-inv-list-of-oalist*: *oalist-inv* (*list-of-oalist x*)
and *list-of-oalist-of-list*: *list-of-oalist* (*oalist-of-list xs*) = *sort-oalist-raw xs*
and *oalist-of-list-of-oalist*: *oalist-of-list* (*list-of-oalist x*) = *x*
begin

lemma *list-of-oalist-of-list-id*:

assumes *oalist-inv xs*
shows *list-of-oalist* (*oalist-of-list xs*) = *xs*
⟨*proof*⟩

definition *lookup* :: *'x* ⇒ *'a* ⇒ *'b*
where *lookup xs* = *lookup-raw* (*list-of-oalist xs*)

definition *sorted-domain* :: *'o* ⇒ *'x* ⇒ *'a list*
where *sorted-domain ko xs* = *sorted-domain-raw ko* (*list-of-oalist xs*)

definition *empty* :: *'o* ⇒ *'x*
where *empty ko* = *oalist-of-list* ([], *ko*)

definition *reorder* :: *'o* ⇒ *'x* ⇒ *'x*
where *reorder ko xs* = *oalist-of-list* (*sort-oalist-aux ko* (*list-of-oalist xs*), *ko*)

definition *tl* :: *'x* ⇒ *'x*
where *tl xs* = *oalist-of-list* (*tl-raw* (*list-of-oalist xs*))

definition *hd* :: *'x* ⇒ (*'a* × *'b*)
where *hd xs* = *List.hd* (*fst* (*list-of-oalist xs*))

definition *except-min* :: *'o* ⇒ *'x* ⇒ *'x*
where *except-min ko xs* = *tl* (*reorder ko xs*)

definition *min-key-val* :: *'o* ⇒ *'x* ⇒ (*'a* × *'b*)
where *min-key-val ko xs* = *min-key-val-raw ko* (*list-of-oalist xs*)

definition *insert* :: (*'a* × *'b*) ⇒ *'x* ⇒ *'x*
where *insert x xs* = *oalist-of-list* (*update-by-raw x* (*list-of-oalist xs*))

definition *update-by-fun* :: *'a* ⇒ (*'b* ⇒ *'b*) ⇒ *'x* ⇒ *'x*
where *update-by-fun k f xs* = *oalist-of-list* (*update-by-fun-raw k f* (*list-of-oalist xs*))

definition *update-by-fun-gr* :: *'a* ⇒ (*'b* ⇒ *'b*) ⇒ *'x* ⇒ *'x*
where *update-by-fun-gr k f xs* = *oalist-of-list* (*update-by-fun-gr-raw k f* (*list-of-oalist xs*))

definition *filter* :: ((*'a* × *'b*) ⇒ *bool*) ⇒ *'x* ⇒ *'x*
where *filter P xs* = *oalist-of-list* (*filter-raw P* (*list-of-oalist xs*))

definition *map2-val-neutr* :: (*'a* ⇒ *'b* ⇒ *'b* ⇒ *'b*) ⇒ *'x* ⇒ *'x* ⇒ *'x*
where *map2-val-neutr f xs ys* = *oalist-of-list* (*map2-val-raw f id id* (*list-of-oalist xs*) (*list-of-oalist ys*))

definition *oalist-eq* :: *'x* ⇒ *'x* ⇒ *bool*
where *oalist-eq xs ys* = *oalist-eq-raw* (*list-of-oalist xs*) (*list-of-oalist ys*)

12.7.1 Invariant

lemma *zero-notin-list-of-oalist*: $0 \notin snd\ `\ set\ (fst\ (list\text{-}of\text{-}oalist\ xs))$
⟨*proof*⟩

lemma *list-of-oalist-sorted*: $sorted\text{-}wrt\ (lt\ (snd\ (list\text{-}of\text{-}oalist\ xs)))\ (map\ fst\ (fst\ (list\text{-}of\text{-}oalist\ xs)))$
⟨*proof*⟩

12.7.2 *lookup*

lemma *lookup-eq-value*: $v \neq 0 \Longrightarrow lookup\ xs\ k = v \longleftrightarrow ((k, v) \in set\ (fst\ (list\text{-}of\text{-}oalist\ xs)))$
⟨*proof*⟩

lemma *lookup-eq-valueI*: $(k, v) \in set\ (fst\ (list\text{-}of\text{-}oalist\ xs)) \Longrightarrow lookup\ xs\ k = v$
⟨*proof*⟩

lemma *lookup-oalist-of-list*:
$distinct\ (map\ fst\ xs) \Longrightarrow lookup\ (oalist\text{-}of\text{-}list\ (xs, ko)) = lookup\text{-}dflt\ xs$
⟨*proof*⟩

12.7.3 *sorted-domain*

lemma *set-sorted-domain*: $set\ (sorted\text{-}domain\ ko\ xs) = fst\ `\ set\ (fst\ (list\text{-}of\text{-}oalist\ xs))$
⟨*proof*⟩

lemma *in-sorted-domain-iff-lookup*: $k \in set\ (sorted\text{-}domain\ ko\ xs) \longleftrightarrow (lookup\ xs\ k \neq 0)$
⟨*proof*⟩

lemma *sorted-sorted-domain*: $sorted\text{-}wrt\ (lt\ ko)\ (sorted\text{-}domain\ ko\ xs)$
⟨*proof*⟩

12.7.4 *local.empty* and Singletons

lemma *list-of-oalist-empty* [*simp*, *code abstract*]: $list\text{-}of\text{-}oalist\ (empty\ ko) = ([], ko)$
⟨*proof*⟩

lemma *lookup-empty*: $lookup\ (empty\ ko)\ k = 0$
⟨*proof*⟩

lemma *lookup-oalist-of-list-single*:
$lookup\ (oalist\text{-}of\text{-}list\ ([(k, v)], ko))\ k' = (if\ k = k'\ then\ v\ else\ 0)$
⟨*proof*⟩

12.7.5 *reorder*

lemma *list-of-oalist-reorder* [*simp*, *code abstract*]:

list-of-oalist (*reorder ko xs*) = (*sort-oalist-aux ko* (*list-of-oalist xs*), *ko*)
⟨*proof*⟩

lemma *lookup-reorder*: *lookup* (*reorder ko xs*) *k* = *lookup xs k*
⟨*proof*⟩

12.7.6 *local.hd* and *local.tl*

lemma *list-of-oalist-tl* [*simp*, *code abstract*]: *list-of-oalist* (*tl xs*) = *tl-raw* (*list-of-oalist xs*)
⟨*proof*⟩

lemma *lookup-tl*:
lookup (*tl xs*) *k* =
(*if* (∀ *k'*∈*fst* ‘ *set* (*fst* (*list-of-oalist xs*)). *le* (*snd* (*list-of-oalist xs*)) *k k'*) *then* *0 else lookup xs k*)
⟨*proof*⟩

lemma *hd-in*:
assumes *fst* (*list-of-oalist xs*) ≠ []
shows *hd xs* ∈ *set* (*fst* (*list-of-oalist xs*))
⟨*proof*⟩

lemma *snd-hd*:
assumes *fst* (*list-of-oalist xs*) ≠ []
shows *snd* (*hd xs*) = *lookup xs* (*fst* (*hd xs*))
⟨*proof*⟩

lemma *lookup-tl'*: *lookup* (*tl xs*) *k* = (*if k* = *fst* (*hd xs*) *then 0 else lookup xs k*)
⟨*proof*⟩

lemma *hd-tl*:
assumes *fst* (*list-of-oalist xs*) ≠ []
shows *list-of-oalist xs* = ((*hd xs*) # (*fst* (*list-of-oalist* (*tl xs*))), *snd* (*list-of-oalist* (*tl xs*)))
⟨*proof*⟩

12.7.7 *min-key-val*

lemma *min-key-val-alt*:
assumes *fst* (*list-of-oalist xs*) ≠ []
shows *min-key-val ko xs* = *hd* (*reorder ko xs*)
⟨*proof*⟩

lemma *min-key-val-in*:
assumes *fst* (*list-of-oalist xs*) ≠ []
shows *min-key-val ko xs* ∈ *set* (*fst* (*list-of-oalist xs*))
⟨*proof*⟩

lemma *snd-min-key-val*:

assumes *fst* (*list-of-oalist xs*) ≠ []
shows *snd* (*min-key-val ko xs*) = *lookup xs* (*fst* (*min-key-val ko xs*))
⟨*proof*⟩

lemma *min-key-val-minimal*:
assumes *z* ∈ *set* (*fst* (*list-of-oalist xs*))
shows *le ko* (*fst* (*min-key-val ko xs*)) (*fst z*)
⟨*proof*⟩

12.7.8 *except-min*

lemma *list-of-oalist-except-min* [*simp*, *code abstract*]:
list-of-oalist (*except-min ko xs*) = (*List.tl* (*sort-oalist-aux ko* (*list-of-oalist xs*)), *ko*)
⟨*proof*⟩

lemma *except-min-Nil*:
assumes *fst* (*list-of-oalist xs*) = []
shows *fst* (*list-of-oalist* (*except-min ko xs*)) = []
⟨*proof*⟩

lemma *lookup-except-min*:
lookup (*except-min ko xs*) *k* =
(*if* (∀ *k′*∈*fst* ‘ *set* (*fst* (*list-of-oalist xs*)). *le ko k k′*) *then 0 else lookup xs k*)
⟨*proof*⟩

lemma *lookup-except-min′*:
lookup (*except-min ko xs*) *k* = (*if k* = *fst* (*min-key-val ko xs*) *then 0 else lookup xs k*)
⟨*proof*⟩

12.7.9 *local.insert*

lemma *list-of-oalist-insert* [*simp*, *code abstract*]:
list-of-oalist (*insert x xs*) = *update-by-raw x* (*list-of-oalist xs*)
⟨*proof*⟩

lemma *lookup-insert*: *lookup* (*insert* (*k*, *v*) *xs*) *k′* = (*if k* = *k′ then v else lookup xs k′*)
⟨*proof*⟩

12.7.10 *update-by-fun* **and** *update-by-fun-gr*

lemma *list-of-oalist-update-by-fun* [*simp*, *code abstract*]:
list-of-oalist (*update-by-fun k f xs*) = *update-by-fun-raw k f* (*list-of-oalist xs*)
⟨*proof*⟩

lemma *lookup-update-by-fun*:
lookup (*update-by-fun k f xs*) *k′* = (*if k* = *k′ then f else id*) (*lookup xs k′*)
⟨*proof*⟩

lemma *list-of-oalist-update-by-fun-gr* [*simp*, *code abstract*]:
list-of-oalist (*update-by-fun-gr k f xs*) = *update-by-fun-gr-raw k f* (*list-of-oalist xs*)
⟨*proof*⟩

lemma *update-by-fun-gr-eq-update-by-fun*: *update-by-fun-gr* = *update-by-fun*
⟨*proof*⟩

12.7.11 *local.filter*

lemma *list-of-oalist-filter* [*simp*, *code abstract*]:
list-of-oalist (*filter P xs*) = *filter-raw P* (*list-of-oalist xs*)
⟨*proof*⟩

lemma *lookup-filter*: *lookup* (*filter P xs*) *k* = (*let v* = *lookup xs k in if P* (*k*, *v*) *then v else 0*)
⟨*proof*⟩

12.7.12 *map2-val-neutr*

lemma *list-of-oalist-map2-val-neutr* [*simp*, *code abstract*]:
list-of-oalist (*map2-val-neutr f xs ys*) = *map2-val-raw f id id* (*list-of-oalist xs*) (*list-of-oalist ys*)
⟨*proof*⟩

lemma *lookup-map2-val-neutr*:
assumes ⋀*k x*. *f k x 0* = *x* **and** ⋀*k x*. *f k 0 x* = *x*
shows *lookup* (*map2-val-neutr f xs ys*) *k* = *f k* (*lookup xs k*) (*lookup ys k*)
⟨*proof*⟩

12.7.13 *oalist-eq*

lemma *oalist-eq-alt*: *oalist-eq xs ys* ⟷ (*lookup xs* = *lookup ys*)
⟨*proof*⟩

end

12.8 Fundamental Operations on Three Lists

locale *oalist-abstract3* =
oalist-abstract rep-key-order list-of-oalistx oalist-of-listx +
oay: *oalist-abstract rep-key-order list-of-oalisty oalist-of-listy* +
oaz: *oalist-abstract rep-key-order list-of-oalistz oalist-of-listz*
for *rep-key-order* :: *'o* ⇒ *'a key-order*
and *list-of-oalistx* :: *'x* ⇒ (*'a*, *'b*::*zero*, *'o*) *oalist-raw*
and *oalist-of-listx* :: (*'a*, *'b*, *'o*) *oalist-raw* ⇒ *'x*
and *list-of-oalisty* :: *'y* ⇒ (*'a*, *'c*::*zero*, *'o*) *oalist-raw*
and *oalist-of-listy* :: (*'a*, *'c*, *'o*) *oalist-raw* ⇒ *'y*
and *list-of-oalistz* :: *'z* ⇒ (*'a*, *'d*::*zero*, *'o*) *oalist-raw*

and *oalist-of-listz* :: ('*a*, '*d*, '*o*) *oalist-raw* ⇒ '*z*
begin

definition *map-val* :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ '*x* ⇒ '*y*
where *map-val f xs* = *oalist-of-listy* (*map-val-raw f* (*list-of-oalistx xs*))

definition *map2-val* :: ('*a* ⇒ '*b* ⇒ '*c* ⇒ '*d*) ⇒ '*x* ⇒ '*y* ⇒ '*z*
where *map2-val f xs ys* =
oalist-of-listz (*map2-val-raw f* (*map-val-raw* (λ*k b*. *f k b 0*)) (*map-val-raw* (λ*k*. *f k 0*))
(*list-of-oalistx xs*) (*list-of-oalisty ys*))

definition *map2-val-rneutr* :: ('*a* ⇒ '*b* ⇒ '*c* ⇒ '*b*) ⇒ '*x* ⇒ '*y* ⇒ '*x*
where *map2-val-rneutr f xs ys* =
oalist-of-listx (*map2-val-raw f id* (*map-val-raw* (λ*k*. *f k 0*)) (*list-of-oalistx xs*) (*list-of-oalisty ys*))

definition *lex-ord* :: '*o* ⇒ ('*a* ⇒ ('*b*, '*c*) *comp-opt*) ⇒ ('*x*, '*y*) *comp-opt*
where *lex-ord ko f xs ys* = *lex-ord-raw ko f* (*list-of-oalistx xs*) (*list-of-oalisty ys*)

definition *prod-ord* :: ('*a* ⇒ '*b* ⇒ '*c* ⇒ *bool*) ⇒ '*x* ⇒ '*y* ⇒ *bool*
where *prod-ord f xs ys* = *prod-ord-raw f* (*list-of-oalistx xs*) (*list-of-oalisty ys*)

12.8.1 *map-val*

lemma *map-val-cong*:
assumes ⋀*k v*. (*k*, *v*) ∈ *set* (*fst* (*list-of-oalistx xs*)) ⟹ *f k v* = *g k v*
shows *map-val f xs* = *map-val g xs*
⟨*proof*⟩

lemma *list-of-oalist-map-val* [*simp*, *code abstract*]:
list-of-oalisty (*map-val f xs*) = *map-val-raw f* (*list-of-oalistx xs*)
⟨*proof*⟩

lemma *lookup-map-val*: *f k 0* = *0* ⟹ *oay.lookup* (*map-val f xs*) *k* = *f k* (*lookup xs k*)
⟨*proof*⟩

12.8.2 *map2-val* **and** *map2-val-rneutr*

lemma *list-of-oalist-map2-val* [*simp*, *code abstract*]:
list-of-oalistz (*map2-val f xs ys*) =
map2-val-raw f (*map-val-raw* (λ*k b*. *f k b 0*)) (*map-val-raw* (λ*k*. *f k 0*)) (*list-of-oalistx xs*) (*list-of-oalisty ys*)
⟨*proof*⟩

lemma *list-of-oalist-map2-val-rneutr* [*simp*, *code abstract*]:
list-of-oalistx (*map2-val-rneutr f xs ys*) =
map2-val-raw f id (*map-val-raw* (λ*k c*. *f k 0 c*)) (*list-of-oalistx xs*) (*list-of-oalisty ys*)

⟨*proof*⟩

lemma *lookup-map2-val*:
assumes $\bigwedge k.\ f\ k\ 0\ 0 = 0$
shows *oaz.lookup* (*map2-val f xs ys*) $k = f\ k$ (*lookup xs k*) (*oay.lookup ys k*)
⟨*proof*⟩

lemma *lookup-map2-val-rneutr*:
assumes $\bigwedge k\ x.\ f\ k\ x\ 0 = x$
shows *lookup* (*map2-val-rneutr f xs ys*) $k = f\ k$ (*lookup xs k*) (*oay.lookup ys k*)
⟨*proof*⟩

lemma *map2-val-rneutr-singleton-eq-update-by-fun*:
assumes $\bigwedge a\ x.\ f\ a\ x\ 0 = x$ **and** *list-of-oalisty ys* = ([(*k*, *v*)], *oy*)
shows *map2-val-rneutr f xs ys* = *update-by-fun k* ($\lambda x.\ f\ k\ x\ v$) *xs*
⟨*proof*⟩

12.8.3 *lex-ord* and *prod-ord*

lemma *lex-ord-EqI*:
($\bigwedge k.\ k \in$ *fst* ‘ *set* (*fst* (*list-of-oalistx xs*)) ∪ *fst* ‘ *set* (*fst* (*list-of-oalisty ys*)) ⟹
f k (*lookup xs k*) (*oay.lookup ys k*) = *Some Eq*) ⟹
lex-ord ko f xs ys = *Some Eq*
⟨*proof*⟩

lemma *lex-ord-valI*:
assumes *aux* ≠ *Some Eq* **and** $k \in$ *fst* ‘ *set* (*fst* (*list-of-oalistx xs*)) ∪ *fst* ‘ *set* (*fst* (*list-of-oalisty ys*))
shows *aux* = *f k* (*lookup xs k*) (*oay.lookup ys k*) ⟹
($\bigwedge k'.\ k' \in$ *fst* ‘ *set* (*fst* (*list-of-oalistx xs*)) ∪ *fst* ‘ *set* (*fst* (*list-of-oalisty ys*)) ⟹
lt ko k′ k ⟹ *f k′* (*lookup xs k′*) (*oay.lookup ys k′*) = *Some Eq*) ⟹
lex-ord ko f xs ys = *aux*
⟨*proof*⟩

lemma *lex-ord-EqD*:
lex-ord ko f xs ys = *Some Eq* ⟹
$k \in$ *fst* ‘ *set* (*fst* (*list-of-oalistx xs*)) ∪ *fst* ‘ *set* (*fst* (*list-of-oalisty ys*)) ⟹
f k (*lookup xs k*) (*oay.lookup ys k*) = *Some Eq*
⟨*proof*⟩

lemma *lex-ord-valE*:
assumes *lex-ord ko f xs ys* = *aux* **and** *aux* ≠ *Some Eq*
obtains *k* **where** $k \in$ *fst* ‘ *set* (*fst* (*list-of-oalistx xs*)) ∪ *fst* ‘ *set* (*fst* (*list-of-oalisty ys*))
and *aux* = *f k* (*lookup xs k*) (*oay.lookup ys k*)
and $\bigwedge k'.\ k' \in$ *fst* ‘ *set* (*fst* (*list-of-oalistx xs*)) ∪ *fst* ‘ *set* (*fst* (*list-of-oalisty ys*)) ⟹
lt ko k′ k ⟹ *f k′* (*lookup xs k′*) (*oay.lookup ys k′*) = *Some Eq*

⟨*proof*⟩

lemma *prod-ord-alt*:
  prod-ord P xs ys ⟷
    (∀ *k*∈*fst* ‘ *set* (*fst* (*list-of-oalistx xs*)) ∪ *fst* ‘ *set* (*fst* (*list-of-oalisty ys*)).
      P k (*lookup xs k*) (*oay.lookup ys k*))
  ⟨*proof*⟩

end

12.9 Type *oalist*

global-interpretation *ko*: *comparator key-compare ko*
  defines *lookup-pair-ko* = *ko.lookup-pair*
  and *update-by-pair-ko* = *ko.update-by-pair*
  and *update-by-fun-pair-ko* = *ko.update-by-fun-pair*
  and *update-by-fun-gr-pair-ko* = *ko.update-by-fun-gr-pair*
  and *map2-val-pair-ko* = *ko.map2-val-pair*
  and *lex-ord-pair-ko* = *ko.lex-ord-pair*
  and *prod-ord-pair-ko* = *ko.prod-ord-pair*
  and *sort-oalist-ko′* = *ko.sort-oalist*
  ⟨*proof*⟩

lemma *ko-le*: *ko.le* = *le-of-key-order*
  ⟨*proof*⟩

global-interpretation *ko*: *oalist-raw* λ*x*. *x*
  rewrites *comparator.lookup-pair* (*key-compare ko*) = *lookup-pair-ko ko*
  and *comparator.update-by-pair* (*key-compare ko*) = *update-by-pair-ko ko*
  and *comparator.update-by-fun-pair* (*key-compare ko*) = *update-by-fun-pair-ko ko*
  and *comparator.update-by-fun-gr-pair* (*key-compare ko*) = *update-by-fun-gr-pair-ko ko*
  and *comparator.map2-val-pair* (*key-compare ko*) = *map2-val-pair-ko ko*
  and *comparator.lex-ord-pair* (*key-compare ko*) = *lex-ord-pair-ko ko*
  and *comparator.prod-ord-pair* (*key-compare ko*) = *prod-ord-pair-ko ko*
  and *comparator.sort-oalist* (*key-compare ko*) = *sort-oalist-ko′ ko*
  defines *sort-oalist-aux-ko* = *ko.sort-oalist-aux*
  and *lookup-ko* = *ko.lookup-raw*
  and *sorted-domain-ko* = *ko.sorted-domain-raw*
  and *tl-ko* = *ko.tl-raw*
  and *min-key-val-ko* = *ko.min-key-val-raw*
  and *update-by-ko* = *ko.update-by-raw*
  and *update-by-fun-ko* = *ko.update-by-fun-raw*
  and *update-by-fun-gr-ko* = *ko.update-by-fun-gr-raw*
  and *map2-val-ko* = *ko.map2-val-raw*
  and *lex-ord-ko* = *ko.lex-ord-raw*
  and *prod-ord-ko* = *ko.prod-ord-raw*
  and *oalist-eq-ko* = *ko.oalist-eq-raw*

and *sort-oalist-ko* = *ko.sort-oalist-raw*
⟨*proof*⟩

typedef (**overloaded**) ('*a*, '*b*) *oalist* = {*xs*::('*a*, '*b*::*zero*, '*a key-order*) *oalist-raw*. *ko.oalist-inv xs*}
morphisms *list-of-oalist Abs-oalist*
⟨*proof*⟩

lemma *oalist-eq-iff*: *xs* = *ys* ⟷ *list-of-oalist xs* = *list-of-oalist ys*
⟨*proof*⟩

lemma *oalist-eqI*: *list-of-oalist xs* = *list-of-oalist ys* ⟹ *xs* = *ys*
⟨*proof*⟩

Formal, totalized constructor for ('*a*, '*b*) *oalist*:

definition *OAlist* :: ('*a* × '*b*) *list* × '*a key-order* ⇒ ('*a*, '*b*::*zero*) *oalist* **where**
OAlist xs = *Abs-oalist* (*sort-oalist-ko xs*)

definition *oalist-of-list* = *OAlist*

lemma *oalist-inv-list-of-oalist*: *ko.oalist-inv* (*list-of-oalist xs*)
⟨*proof*⟩

lemma *list-of-oalist-OAlist*: *list-of-oalist* (*OAlist xs*) = *sort-oalist-ko xs*
⟨*proof*⟩

lemma *OAlist-list-of-oalist* [*code abstype*]: *OAlist* (*list-of-oalist xs*) = *xs*
⟨*proof*⟩

lemma [*code abstract*]: *list-of-oalist* (*oalist-of-list xs*) = *sort-oalist-ko xs*
⟨*proof*⟩

global-interpretation *oa*: *oalist-abstract* λ*x*. *x list-of-oalist OAlist*
defines *OAlist-lookup* = *oa.lookup*
and *OAlist-sorted-domain* = *oa.sorted-domain*
and *OAlist-empty* = *oa.empty*
and *OAlist-reorder* = *oa.reorder*
and *OAlist-tl* = *oa.tl*
and *OAlist-hd* = *oa.hd*
and *OAlist-except-min* = *oa.except-min*
and *OAlist-min-key-val* = *oa.min-key-val*
and *OAlist-insert* = *oa.insert*
and *OAlist-update-by-fun* = *oa.update-by-fun*
and *OAlist-update-by-fun-gr* = *oa.update-by-fun-gr*
and *OAlist-filter* = *oa.filter*
and *OAlist-map2-val-neutr* = *oa.map2-val-neutr*
and *OAlist-eq* = *oa.oalist-eq*
⟨*proof*⟩

global-interpretation *oa*: *oalist-abstract3* $\lambda x.\ x$
  list-of-oalist::('*a*, '*b*) *oalist* ⇒ ('*a*, '*b*::*zero*, '*a key-order*) *oalist-raw OAlist*
  list-of-oalist::('*a*, '*c*) *oalist* ⇒ ('*a*, '*c*::*zero*, '*a key-order*) *oalist-raw OAlist*
  list-of-oalist::('*a*, '*d*) *oalist* ⇒ ('*a*, '*d*::*zero*, '*a key-order*) *oalist-raw OAlist*
 defines *OAlist-map-val* = *oa.map-val*
 and *OAlist-map2-val* = *oa.map2-val*
 and *OAlist-map2-val-rneutr* = *oa.map2-val-rneutr*
 and *OAlist-lex-ord* = *oa.lex-ord*
 and *OAlist-prod-ord* = *oa.prod-ord* ⟨*proof*⟩

lemmas *OAlist-lookup-single* = *oa.lookup-oalist-of-list-single*[*folded oalist-of-list-def*]

12.10 Type *oalist-tc*

"tc" stands for "type class".

global-interpretation *tc*: *comparator comparator-of*
 defines *lookup-pair-tc* = *tc.lookup-pair*
 and *update-by-pair-tc* = *tc.update-by-pair*
 and *update-by-fun-pair-tc* = *tc.update-by-fun-pair*
 and *update-by-fun-gr-pair-tc* = *tc.update-by-fun-gr-pair*
 and *map2-val-pair-tc* = *tc.map2-val-pair*
 and *lex-ord-pair-tc* = *tc.lex-ord-pair*
 and *prod-ord-pair-tc* = *tc.prod-ord-pair*
 and *sort-oalist-tc* = *tc.sort-oalist*
 ⟨*proof*⟩

lemma *tc-le-lt* [*simp*]: *tc.le* = (≤) *tc.lt* = (<)
 ⟨*proof*⟩

typedef (**overloaded**) ('*a*, '*b*) *oalist-tc* = {*xs*::('*a*::*linorder* × '*b*::*zero*) *list*. *tc.oalist-inv-raw xs*}
 morphisms *list-of-oalist-tc Abs-oalist-tc*
 ⟨*proof*⟩

lemma *oalist-tc-eq-iff*: *xs* = *ys* ⟷ *list-of-oalist-tc xs* = *list-of-oalist-tc ys*
 ⟨*proof*⟩

lemma *oalist-tc-eqI*: *list-of-oalist-tc xs* = *list-of-oalist-tc ys* ⟹ *xs* = *ys*
 ⟨*proof*⟩

Formal, totalized constructor for ('*a*, '*b*) *oalist-tc*:

definition *OAlist-tc* :: ('*a* × '*b*) *list* ⇒ ('*a*::*linorder*, '*b*::*zero*) *oalist-tc* **where**
 OAlist-tc xs = *Abs-oalist-tc* (*sort-oalist-tc xs*)

definition *oalist-tc-of-list* = *OAlist-tc*

lemma *oalist-inv-list-of-oalist-tc*: *tc.oalist-inv-raw* (*list-of-oalist-tc xs*)
 ⟨*proof*⟩

lemma *list-of-oalist-OAlist-tc*: *list-of-oalist-tc* (*OAlist-tc xs*) = *sort-oalist-tc xs*
⟨*proof*⟩

lemma *OAlist-list-of-oalist-tc* [*code abstype*]: *OAlist-tc* (*list-of-oalist-tc xs*) = *xs*
⟨*proof*⟩

lemma *list-of-oalist-tc-of-list* [*code abstract*]: *list-of-oalist-tc* (*oalist-tc-of-list xs*) = *sort-oalist-tc xs*
⟨*proof*⟩

lemma *list-of-oalist-tc-of-list-id*:
assumes *tc.oalist-inv-raw xs*
shows *list-of-oalist-tc* (*OAlist-tc xs*) = *xs*
⟨*proof*⟩

It is better to define the following operations directly instead of interpreting *oalist-abstract*, because *oalist-abstract* defines the operations via their *-raw* analogues, whereas in this case we can define them directly via their *-pair* analogues.

definition *OAlist-tc-lookup* :: ('*a*::*linorder*, '*b*::*zero*) *oalist-tc* ⇒ '*a* ⇒ '*b*
where *OAlist-tc-lookup xs* = *lookup-pair-tc* (*list-of-oalist-tc xs*)

definition *OAlist-tc-sorted-domain* :: ('*a*::*linorder*, '*b*::*zero*) *oalist-tc* ⇒ '*a list*
where *OAlist-tc-sorted-domain xs* = *map fst* (*list-of-oalist-tc xs*)

definition *OAlist-tc-empty* :: ('*a*::*linorder*, '*b*::*zero*) *oalist-tc*
where *OAlist-tc-empty* = *OAlist-tc* []

definition *OAlist-tc-except-min* :: ('*a*, '*b*) *oalist-tc* ⇒ ('*a*::*linorder*, '*b*::*zero*) *oalist-tc*
where *OAlist-tc-except-min xs* = *OAlist-tc* (*tl* (*list-of-oalist-tc xs*))

definition *OAlist-tc-min-key-val* :: ('*a*::*linorder*, '*b*::*zero*) *oalist-tc* ⇒ ('*a* × '*b*)
where *OAlist-tc-min-key-val xs* = *hd* (*list-of-oalist-tc xs*)

definition *OAlist-tc-insert* :: ('*a* × '*b*) ⇒ ('*a*, '*b*) *oalist-tc* ⇒ ('*a*::*linorder*, '*b*::*zero*) *oalist-tc*
where *OAlist-tc-insert x xs* = *OAlist-tc* (*update-by-pair-tc x* (*list-of-oalist-tc xs*))

definition *OAlist-tc-update-by-fun* :: '*a* ⇒ ('*b* ⇒ '*b*) ⇒ ('*a*, '*b*) *oalist-tc* ⇒ ('*a*::*linorder*, '*b*::*zero*) *oalist-tc*
where *OAlist-tc-update-by-fun k f xs* = *OAlist-tc* (*update-by-fun-pair-tc k f* (*list-of-oalist-tc xs*))

definition *OAlist-tc-update-by-fun-gr* :: '*a* ⇒ ('*b* ⇒ '*b*) ⇒ ('*a*, '*b*) *oalist-tc* ⇒ ('*a*::*linorder*, '*b*::*zero*) *oalist-tc*
where *OAlist-tc-update-by-fun-gr k f xs* = *OAlist-tc* (*update-by-fun-gr-pair-tc k f* (*list-of-oalist-tc xs*))

definition *OAlist-tc-filter* :: (('*a* × '*b*) ⇒ *bool*) ⇒ ('*a*, '*b*) *oalist-tc* ⇒ ('*a*::*linorder*,

'b::zero) oalist-tc
  where *OAlist-tc-filter P xs = OAlist-tc (filter P (list-of-oalist-tc xs))*

definition *OAlist-tc-map-val* :: ('a ⇒ 'b ⇒ 'c) ⇒ ('a, 'b::zero) oalist-tc ⇒ ('a::linorder, 'c::zero) oalist-tc
  where *OAlist-tc-map-val f xs = OAlist-tc (map-val-pair f (list-of-oalist-tc xs))*

definition *OAlist-tc-map2-val* :: ('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ ('a, 'b::zero) oalist-tc ⇒ ('a, 'c::zero) oalist-tc ⇒
('a::linorder, 'd::zero) oalist-tc
  where *OAlist-tc-map2-val f xs ys =*
OAlist-tc (map2-val-pair-tc f (map-val-pair (λk b. f k b 0)) (map-val-pair (λk. f k 0))
(list-of-oalist-tc xs) (list-of-oalist-tc ys))

definition *OAlist-tc-map2-val-rneutr* :: ('a ⇒ 'b ⇒ 'c ⇒ 'b) ⇒ ('a, 'b) oalist-tc ⇒ ('a, 'c::zero) oalist-tc ⇒
('a::linorder, 'b::zero) oalist-tc
  where *OAlist-tc-map2-val-rneutr f xs ys =*
OAlist-tc (map2-val-pair-tc f id (map-val-pair (λk. f k 0)) (list-of-oalist-tc xs) (list-of-oalist-tc ys))

definition *OAlist-tc-map2-val-neutr* :: ('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a, 'b) oalist-tc ⇒
('a, 'b) oalist-tc ⇒ ('a::linorder, 'b::zero) oalist-tc
  where *OAlist-tc-map2-val-neutr f xs ys = OAlist-tc (map2-val-pair-tc f id id (list-of-oalist-tc xs) (list-of-oalist-tc ys))*

definition *OAlist-tc-lex-ord* :: ('a ⇒ ('b, 'c) comp-opt) ⇒ (('a, 'b::zero) oalist-tc, ('a::linorder, 'c::zero) oalist-tc) comp-opt
  where *OAlist-tc-lex-ord f xs ys = lex-ord-pair-tc f (list-of-oalist-tc xs) (list-of-oalist-tc ys)*

definition *OAlist-tc-prod-ord* :: ('a ⇒ 'b ⇒ 'c ⇒ bool) ⇒ ('a, 'b::zero) oalist-tc ⇒ ('a::linorder, 'c::zero) oalist-tc ⇒ bool
  where *OAlist-tc-prod-ord f xs ys = prod-ord-pair-tc f (list-of-oalist-tc xs) (list-of-oalist-tc ys)*

12.10.1 *OAlist-tc-lookup*

lemma *OAlist-tc-lookup-eq-valueI*: (k, v) ∈ *set* (*list-of-oalist-tc xs*) ⟹ *OAlist-tc-lookup xs k = v*
  ⟨*proof*⟩

lemma *OAlist-tc-lookup-inj*: *OAlist-tc-lookup xs = OAlist-tc-lookup ys* ⟹ *xs = ys*
  ⟨*proof*⟩

lemma *OAlist-tc-lookup-oalist-of-list*:
  distinct (*map fst xs*) ⟹ *OAlist-tc-lookup* (*oalist-tc-of-list xs*) = *lookup-dflt xs*

⟨*proof*⟩

12.10.2 *OAlist-tc-sorted-domain*

lemma *set-OAlist-tc-sorted-domain*: *set* (*OAlist-tc-sorted-domain xs*) = *fst* ‘ *set* (*list-of-oalist-tc xs*)
⟨*proof*⟩

lemma *in-OAlist-tc-sorted-domain-iff-lookup*: $k \in$ *set* (*OAlist-tc-sorted-domain xs*) $\longleftrightarrow$ (*OAlist-tc-lookup xs k* $\neq$ *0*)
⟨*proof*⟩

lemma *sorted-OAlist-tc-sorted-domain*: *sorted-wrt* (<) (*OAlist-tc-sorted-domain xs*)
⟨*proof*⟩

12.10.3 *OAlist-tc-empty* and Singletons

lemma *list-of-oalist-OAlist-tc-empty* [*simp*, *code abstract*]: *list-of-oalist-tc OAlist-tc-empty* = []
⟨*proof*⟩

lemma *lookup-OAlist-tc-empty*: *OAlist-tc-lookup OAlist-tc-empty k* = *0*
⟨*proof*⟩

lemma *OAlist-tc-lookup-single*:
OAlist-tc-lookup (*oalist-tc-of-list* [(*k*, *v*)]) *k′* = (*if k* = *k′ then v else 0*)
⟨*proof*⟩

12.10.4 *OAlist-tc-except-min*

lemma *list-of-oalist-OAlist-tc-except-min* [*simp*, *code abstract*]:
list-of-oalist-tc (*OAlist-tc-except-min xs*) = *tl* (*list-of-oalist-tc xs*)
⟨*proof*⟩

lemma *lookup-OAlist-tc-except-min*:
OAlist-tc-lookup (*OAlist-tc-except-min xs*) *k* =
(*if* (∀ *k′*∈*fst* ‘ *set* (*list-of-oalist-tc xs*). *k* ≤ *k′*) *then 0 else OAlist-tc-lookup xs k*)
⟨*proof*⟩

12.10.5 *OAlist-tc-min-key-val*

lemma *OAlist-tc-min-key-val-in*:
assumes *list-of-oalist-tc xs* ≠ []
shows *OAlist-tc-min-key-val xs* ∈ *set* (*list-of-oalist-tc xs*)
⟨*proof*⟩

lemma *snd-OAlist-tc-min-key-val*:
assumes *list-of-oalist-tc xs* ≠ []

shows *snd* (*OAlist-tc-min-key-val xs*) = *OAlist-tc-lookup xs* (*fst* (*OAlist-tc-min-key-val xs*))
⟨*proof*⟩

lemma *OAlist-tc-min-key-val-minimal*:
assumes $z \in$ *set* (*list-of-oalist-tc xs*)
shows *fst* (*OAlist-tc-min-key-val xs*) ≤ *fst z*
⟨*proof*⟩

12.10.6 *OAlist-tc-insert*

lemma *list-of-oalist-OAlist-tc-insert* [*simp*, *code abstract*]:
list-of-oalist-tc (*OAlist-tc-insert x xs*) = *update-by-pair-tc x* (*list-of-oalist-tc xs*)
⟨*proof*⟩

lemma *lookup-OAlist-tc-insert*: *OAlist-tc-lookup* (*OAlist-tc-insert* (*k*, *v*) *xs*) *k'* = (*if k* = *k' then v else OAlist-tc-lookup xs k'*)
⟨*proof*⟩

12.10.7 *OAlist-tc-update-by-fun* and *OAlist-tc-update-by-fun-gr*

lemma *list-of-oalist-OAlist-tc-update-by-fun* [*simp*, *code abstract*]:
list-of-oalist-tc (*OAlist-tc-update-by-fun k f xs*) = *update-by-fun-pair-tc k f* (*list-of-oalist-tc xs*)
⟨*proof*⟩

lemma *lookup-OAlist-tc-update-by-fun*:
OAlist-tc-lookup (*OAlist-tc-update-by-fun k f xs*) *k'* = (*if k* = *k' then f else id*) (*OAlist-tc-lookup xs k'*)
⟨*proof*⟩

lemma *list-of-oalist-OAlist-tc-update-by-fun-gr* [*simp*, *code abstract*]:
list-of-oalist-tc (*OAlist-tc-update-by-fun-gr k f xs*) = *update-by-fun-gr-pair-tc k f* (*list-of-oalist-tc xs*)
⟨*proof*⟩

lemma *OAlist-tc-update-by-fun-gr-eq-OAlist-tc-update-by-fun*: *OAlist-tc-update-by-fun-gr* = *OAlist-tc-update-by-fun*
⟨*proof*⟩

12.10.8 *OAlist-tc-filter*

lemma *list-of-oalist-OAlist-tc-filter* [*simp*, *code abstract*]:
list-of-oalist-tc (*OAlist-tc-filter P xs*) = *filter P* (*list-of-oalist-tc xs*)
⟨*proof*⟩

lemma *lookup-OAlist-tc-filter*: *OAlist-tc-lookup* (*OAlist-tc-filter P xs*) *k* = (*let v* = *OAlist-tc-lookup xs k in if P* (*k*, *v*) *then v else 0*)
⟨*proof*⟩

12.10.9 *OAlist-tc-map-val*

lemma *list-of-oalist-OAlist-tc-map-val* [*simp*, *code abstract*]:
list-of-oalist-tc (*OAlist-tc-map-val f xs*) = *map-val-pair f* (*list-of-oalist-tc xs*)
⟨*proof*⟩

lemma *OAlist-tc-map-val-cong*:
assumes ⋀*k v*. (*k*, *v*) ∈ *set* (*list-of-oalist-tc xs*) ⟹ *f k v* = *g k v*
shows *OAlist-tc-map-val f xs* = *OAlist-tc-map-val g xs*
⟨*proof*⟩

lemma *lookup-OAlist-tc-map-val*: *f k 0* = *0* ⟹ *OAlist-tc-lookup* (*OAlist-tc-map-val f xs*) *k* = *f k* (*OAlist-tc-lookup xs k*)
⟨*proof*⟩

12.10.10 *OAlist-tc-map2-val OAlist-tc-map2-val-rneutr* **and** *OAlist-tc-map2-val-neutr*

lemma *list-of-oalist-map2-val* [*simp*, *code abstract*]:
list-of-oalist-tc (*OAlist-tc-map2-val f xs ys*) =
map2-val-pair-tc f (*map-val-pair* (λ*k b*. *f k b 0*)) (*map-val-pair* (λ*k*. *f k 0*)) (*list-of-oalist-tc xs*) (*list-of-oalist-tc ys*)
⟨*proof*⟩

lemma *list-of-oalist-OAlist-tc-map2-val-rneutr* [*simp*, *code abstract*]:
list-of-oalist-tc (*OAlist-tc-map2-val-rneutr f xs ys*) =
map2-val-pair-tc f id (*map-val-pair* (λ*k c*. *f k 0 c*)) (*list-of-oalist-tc xs*) (*list-of-oalist-tc ys*)
⟨*proof*⟩

lemma *list-of-oalist-OAlist-tc-map2-val-neutr* [*simp*, *code abstract*]:
list-of-oalist-tc (*OAlist-tc-map2-val-neutr f xs ys*) = *map2-val-pair-tc f id id* (*list-of-oalist-tc xs*) (*list-of-oalist-tc ys*)
⟨*proof*⟩

lemma *lookup-OAlist-tc-map2-val*:
assumes ⋀*k*. *f k 0 0* = *0*
shows *OAlist-tc-lookup* (*OAlist-tc-map2-val f xs ys*) *k* = *f k* (*OAlist-tc-lookup xs k*) (*OAlist-tc-lookup ys k*)
⟨*proof*⟩

lemma *lookup-OAlist-tc-map2-val-rneutr*:
assumes ⋀*k x*. *f k x 0* = *x*
shows *OAlist-tc-lookup* (*OAlist-tc-map2-val-rneutr f xs ys*) *k* = *f k* (*OAlist-tc-lookup xs k*) (*OAlist-tc-lookup ys k*)
⟨*proof*⟩

lemma *lookup-OAlist-tc-map2-val-neutr*:
assumes ⋀*k x*. *f k x 0* = *x* **and** ⋀*k x*. *f k 0 x* = *x*
shows *OAlist-tc-lookup* (*OAlist-tc-map2-val-neutr f xs ys*) *k* = *f k* (*OAlist-tc-lookup xs k*) (*OAlist-tc-lookup ys k*)

⟨*proof*⟩

lemma *OAlist-tc-map2-val-rneutr-singleton-eq-OAlist-tc-update-by-fun*:
assumes $\bigwedge a\ x.\ f\ a\ x\ 0 = x$ **and** *list-of-oalist-tc ys* = [(*k*, *v*)]
shows *OAlist-tc-map2-val-rneutr f xs ys* = *OAlist-tc-update-by-fun k* (λ*x*. *f k x v*) *xs*
⟨*proof*⟩

12.10.11 *OAlist-tc-lex-ord* and *OAlist-tc-prod-ord*

lemma *OAlist-tc-lex-ord-EqI*:
$(\bigwedge k.\ k \in fst\ `\ set\ (list\text{-}of\text{-}oalist\text{-}tc\ xs) \cup fst\ `\ set\ (list\text{-}of\text{-}oalist\text{-}tc\ ys) \Longrightarrow$
f k (*OAlist-tc-lookup xs k*) (*OAlist-tc-lookup ys k*) = *Some Eq*) ⟹
OAlist-tc-lex-ord f xs ys = *Some Eq*
⟨*proof*⟩

lemma *OAlist-tc-lex-ord-valI*:
assumes *aux* ≠ *Some Eq* **and** *k* ∈ *fst* ` *set* (*list-of-oalist-tc xs*) ∪ *fst* ` *set* (*list-of-oalist-tc ys*)
shows *aux* = *f k* (*OAlist-tc-lookup xs k*) (*OAlist-tc-lookup ys k*) ⟹
$(\bigwedge k'.\ k' \in fst\ `\ set\ (list\text{-}of\text{-}oalist\text{-}tc\ xs) \cup fst\ `\ set\ (list\text{-}of\text{-}oalist\text{-}tc\ ys) \Longrightarrow$
$k' < k \Longrightarrow f\ k'\ (OAlist\text{-}tc\text{-}lookup\ xs\ k')\ (OAlist\text{-}tc\text{-}lookup\ ys\ k') = Some\ Eq) \Longrightarrow$
OAlist-tc-lex-ord f xs ys = *aux*
⟨*proof*⟩

lemma *OAlist-tc-lex-ord-EqD*:
OAlist-tc-lex-ord f xs ys = *Some Eq* ⟹
k ∈ *fst* ` *set* (*list-of-oalist-tc xs*) ∪ *fst* ` *set* (*list-of-oalist-tc ys*) ⟹
f k (*OAlist-tc-lookup xs k*) (*OAlist-tc-lookup ys k*) = *Some Eq*
⟨*proof*⟩

lemma *OAlist-tc-lex-ord-valE*:
assumes *OAlist-tc-lex-ord f xs ys* = *aux* **and** *aux* ≠ *Some Eq*
obtains *k* **where** *k* ∈ *fst* ` *set* (*list-of-oalist-tc xs*) ∪ *fst* ` *set* (*list-of-oalist-tc ys*)
and *aux* = *f k* (*OAlist-tc-lookup xs k*) (*OAlist-tc-lookup ys k*)
and $\bigwedge k'.\ k' \in fst\ `\ set\ (list\text{-}of\text{-}oalist\text{-}tc\ xs) \cup fst\ `\ set\ (list\text{-}of\text{-}oalist\text{-}tc\ ys) \Longrightarrow$
$k' < k \Longrightarrow f\ k'\ (OAlist\text{-}tc\text{-}lookup\ xs\ k')\ (OAlist\text{-}tc\text{-}lookup\ ys\ k') = Some\ Eq$
⟨*proof*⟩

lemma *OAlist-tc-prod-ord-alt*:
OAlist-tc-prod-ord P xs ys ⟷
(∀ *k*∈*fst* ` *set* (*list-of-oalist-tc xs*) ∪ *fst* ` *set* (*list-of-oalist-tc ys*).
P k (*OAlist-tc-lookup xs k*) (*OAlist-tc-lookup ys k*))
⟨*proof*⟩

12.10.12 Instance of *equal*

instantiation *oalist-tc* :: (*linorder*, *zero*) *equal*

begin

definition *equal-oalist-tc* :: ('*a*, '*b*) *oalist-tc* ⇒ ('*a*, '*b*) *oalist-tc* ⇒ *bool*
  where *equal-oalist-tc xs ys* = (*list-of-oalist-tc xs* = *list-of-oalist-tc ys*)

instance ⟨*proof*⟩

end

12.11 Experiment

lemma *oalist-tc-of-list* [(*0*::*nat*, *4*::*nat*), (*1*, *3*), (*0*, *2*), (*1*, *1*)] = *oalist-tc-of-list* [(*0*, *4*), (*1*, *3*)]
  ⟨*proof*⟩

lemma *OAlist-tc-except-min* (*oalist-tc-of-list* ([(*1*, *3*), (*0*::*nat*, *4*::*nat*), (*0*, *2*), (*1*, *1*)])) = *oalist-tc-of-list* [(*1*, *3*)]
  ⟨*proof*⟩

lemma *OAlist-tc-min-key-val* (*oalist-tc-of-list* [(*1*, *3*), (*0*::*nat*, *4*::*nat*), (*0*, *2*), (*1*, *1*)]) = (*0*, *4*)
  ⟨*proof*⟩

lemma *OAlist-tc-lookup* (*oalist-tc-of-list* [(*0*::*nat*, *4*::*nat*), (*1*, *3*), (*0*, *2*), (*1*, *1*)]) *1* = *3*
  ⟨*proof*⟩

lemma *OAlist-tc-prod-ord* (λ-. *greater-eq*)
    (*oalist-tc-of-list* [(*1*, *4*), (*0*::*nat*, *4*::*nat*), (*1*, *3*), (*0*, *2*), (*3*, *1*)])
    (*oalist-tc-of-list* [(*0*, *4*), (*1*, *3*), (*2*, *2*), (*1*, *1*)]) = *False*
  ⟨*proof*⟩

lemma *OAlist-tc-map2-val-rneutr* (λ-. *minus*)
    (*oalist-tc-of-list* [(*1*, *4*), (*0*::*nat*, *4*::*int*), (*1*, *3*), (*0*, *2*), (*3*, *1*)])
    (*oalist-tc-of-list* [(*0*, *4*), (*1*, *3*), (*2*, *2*), (*1*, *1*)]) =
  oalist-tc-of-list [(*1*, *1*), (*2*, − *2*), (*3*, *1*)]
  ⟨*proof*⟩

end

13 Ordered Associative Lists for Polynomials

theory *OAlist-Poly-Mapping*
  imports *PP-Type MPoly-Type-Class-Ordered OAlist*
begin

We introduce a dedicated type for ordered associative lists (oalists) representing polynomials. To that end, we require the order relation the oalists are sorted wrt. to be admissible term orders, and furthermore sort the lists

descending rather than *ascending*, because this allows to implement various operations more efficiently. For technical reasons, we must restrict the type of terms to types embeddable into $(nat,\ nat)\ pp \times nat$, though. All types we are interested in meet this requirement.

lemma *comparator-lexicographic*:
fixes $f::'a \Rightarrow 'b$ **and** $g::'a \Rightarrow 'c$
assumes *comparator c1* **and** *comparator c2* **and** $\bigwedge x\ y.\ f\ x = f\ y \Longrightarrow g\ x = g\ y \Longrightarrow x = y$
shows *comparator* $(\lambda x\ y.$ *case c1* $(f\ x)\ (f\ y)$ *of Eq* $\Rightarrow$ *c2* $(g\ x)\ (g\ y) \mid val \Rightarrow val)$ (**is** *comparator ?c3*)
⟨*proof*⟩

class *nat-term* =
fixes *rep-nat-term* :: $'a \Rightarrow ((nat,\ nat)\ pp \times nat)$
and *splus* :: $'a \Rightarrow 'a \Rightarrow 'a$
assumes *rep-nat-term-inj*: *rep-nat-term x* = *rep-nat-term y* $\Longrightarrow x = y$
and *full-component*: *snd* (*rep-nat-term x*) = *i* $\Longrightarrow (\exists\, y.$ *rep-nat-term y* = $(t,\ i))$
and *splus-term*: *rep-nat-term* (*splus x y*) = *pprod.splus* (*fst* (*rep-nat-term x*)) (*rep-nat-term y*)
begin

definition *lex-comp-aux* = $(\lambda x\ y.$ *case comp-of-ord lex-pp* (*fst* (*rep-nat-term x*)) (*fst* (*rep-nat-term y*)) *of*
Eq $\Rightarrow$ *comparator-of* (*snd* (*rep-nat-term x*)) (*snd* (*rep-nat-term y*)) | *val* $\Rightarrow$ *val*)

lemma *full-componentE*:
assumes *snd* (*rep-nat-term x*) = *i*
obtains *y* **where** *rep-nat-term y* = $(t,\ i)$
⟨*proof*⟩

end

class *nat-pp-term* = *nat-term* + *zero* + *plus* +
assumes *rep-nat-term-zero*: *rep-nat-term 0* = $(0,\ 0)$
and *splus-pp-term*: *splus* = $(+)$

definition *nat-term-comp* :: $'a$::*nat-term comparator* $\Rightarrow$ *bool*
where *nat-term-comp cmp* $\longleftrightarrow$
$(\forall\, u\ v.$ *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*) $\longrightarrow$ *fst* (*rep-nat-term u*) = *0* $\longrightarrow$ *cmp u v* $\neq$ *Gt*) $\wedge$
$(\forall\, u\ v.$ *fst* (*rep-nat-term u*) = *fst* (*rep-nat-term v*) $\longrightarrow$ *snd* (*rep-nat-term u*) $<$ *snd* (*rep-nat-term v*) $\longrightarrow$ *cmp u v* = *Lt*) $\wedge$
$(\forall\, t\ u\ v.$ *cmp u v* = *Lt* $\longrightarrow$ *cmp* (*splus t u*) (*splus t v*) = *Lt*) $\wedge$
$(\forall\, u\ v\ a\ b.$ *fst* (*rep-nat-term u*) = *fst* (*rep-nat-term a*) $\longrightarrow$ *fst* (*rep-nat-term v*) = *fst* (*rep-nat-term b*) $\longrightarrow$
snd (*rep-nat-term u*) = *snd* (*rep-nat-term v*) $\longrightarrow$ *snd* (*rep-nat-term a*) = *snd* (*rep-nat-term b*) $\longrightarrow$

$cmp\ a\ b = Lt \longrightarrow cmp\ u\ v = Lt)$

lemma *nat-term-compI*:
assumes $\bigwedge u\ v.\ snd\ (rep\text{-}nat\text{-}term\ u) = snd\ (rep\text{-}nat\text{-}term\ v) \Longrightarrow fst\ (rep\text{-}nat\text{-}term\ u) = 0 \Longrightarrow cmp\ u\ v \neq Gt$
and $\bigwedge u\ v.\ fst\ (rep\text{-}nat\text{-}term\ u) = fst\ (rep\text{-}nat\text{-}term\ v) \Longrightarrow snd\ (rep\text{-}nat\text{-}term\ u) < snd\ (rep\text{-}nat\text{-}term\ v) \Longrightarrow cmp\ u\ v = Lt$
and $\bigwedge t\ u\ v.\ cmp\ u\ v = Lt \Longrightarrow cmp\ (splus\ t\ u)\ (splus\ t\ v) = Lt$
and $\bigwedge u\ v\ a\ b.\ fst\ (rep\text{-}nat\text{-}term\ u) = fst\ (rep\text{-}nat\text{-}term\ a) \Longrightarrow fst\ (rep\text{-}nat\text{-}term\ v) = fst\ (rep\text{-}nat\text{-}term\ b) \Longrightarrow$
$snd\ (rep\text{-}nat\text{-}term\ u) = snd\ (rep\text{-}nat\text{-}term\ v) \Longrightarrow snd\ (rep\text{-}nat\text{-}term\ a) = snd\ (rep\text{-}nat\text{-}term\ b) \Longrightarrow$
$cmp\ a\ b = Lt \Longrightarrow cmp\ u\ v = Lt$
shows *nat-term-comp cmp*
⟨*proof*⟩

lemma *nat-term-compD1*:
assumes *nat-term-comp cmp* **and** $snd\ (rep\text{-}nat\text{-}term\ u) = snd\ (rep\text{-}nat\text{-}term\ v)$ **and** $fst\ (rep\text{-}nat\text{-}term\ u) = 0$
shows $cmp\ u\ v \neq Gt$
⟨*proof*⟩

lemma *nat-term-compD2*:
assumes *nat-term-comp cmp* **and** $fst\ (rep\text{-}nat\text{-}term\ u) = fst\ (rep\text{-}nat\text{-}term\ v)$ **and** $snd\ (rep\text{-}nat\text{-}term\ u) < snd\ (rep\text{-}nat\text{-}term\ v)$
shows $cmp\ u\ v = Lt$
⟨*proof*⟩

lemma *nat-term-compD3*:
assumes *nat-term-comp cmp* **and** $cmp\ u\ v = Lt$
shows $cmp\ (splus\ t\ u)\ (splus\ t\ v) = Lt$
⟨*proof*⟩

lemma *nat-term-compD4*:
assumes *nat-term-comp cmp* **and** $fst\ (rep\text{-}nat\text{-}term\ u) = fst\ (rep\text{-}nat\text{-}term\ a)$
and $fst\ (rep\text{-}nat\text{-}term\ v) = fst\ (rep\text{-}nat\text{-}term\ b)$ **and** $snd\ (rep\text{-}nat\text{-}term\ u) = snd\ (rep\text{-}nat\text{-}term\ v)$
and $snd\ (rep\text{-}nat\text{-}term\ a) = snd\ (rep\text{-}nat\text{-}term\ b)$ **and** $cmp\ a\ b = Lt$
shows $cmp\ u\ v = Lt$
⟨*proof*⟩

lemma *nat-term-compD1'*:
assumes *comparator cmp* **and** *nat-term-comp cmp* **and** $snd\ (rep\text{-}nat\text{-}term\ u) \leq snd\ (rep\text{-}nat\text{-}term\ v)$
and $fst\ (rep\text{-}nat\text{-}term\ u) = 0$
shows $cmp\ u\ v \neq Gt$
⟨*proof*⟩

lemma *nat-term-compD4'*:

assumes *comparator cmp* **and** *nat-term-comp cmp* **and** *fst* (*rep-nat-term u*) = *fst* (*rep-nat-term a*)
and *fst* (*rep-nat-term v*) = *fst* (*rep-nat-term b*) **and** *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*)
and *snd* (*rep-nat-term a*) = *snd* (*rep-nat-term b*)
shows *cmp u v* = *cmp a b*
⟨*proof*⟩

lemma *nat-term-compD4''*:
assumes *comparator cmp* **and** *nat-term-comp cmp* **and** *fst* (*rep-nat-term u*) = *fst* (*rep-nat-term a*)
and *fst* (*rep-nat-term v*) = *fst* (*rep-nat-term b*) **and** *snd* (*rep-nat-term u*) ≤ *snd* (*rep-nat-term v*)
and *snd* (*rep-nat-term a*) = *snd* (*rep-nat-term b*) **and** *cmp a b* ≠ *Gt*
shows *cmp u v* ≠ *Gt*
⟨*proof*⟩

lemma *comparator-lex-comp-aux*: *comparator* (*lex-comp-aux*::'*a*::*nat-term comparator*)
⟨*proof*⟩

lemma *nat-term-comp-lex-comp-aux*: *nat-term-comp* (*lex-comp-aux*::'*a*::*nat-term comparator*)
⟨*proof*⟩

typedef (**overloaded**) '*a nat-term-order* =
{*cmp*::'*a*::*nat-term comparator*. *comparator cmp* ∧ *nat-term-comp cmp*}
morphisms *nat-term-compare Abs-nat-term-order*
⟨*proof*⟩

lemma *nat-term-compare-Abs-nat-term-order-id*:
assumes *comparator cmp* **and** *nat-term-comp cmp*
shows *nat-term-compare* (*Abs-nat-term-order cmp*) = *cmp*
⟨*proof*⟩

instantiation *nat-term-order* :: (*type*) *equal*
begin

definition *equal-nat-term-order* :: '*a nat-term-order* ⇒ '*a nat-term-order* ⇒ *bool*
where *equal-nat-term-order* = (=)

instance ⟨*proof*⟩

end

definition *nat-term-compare-inv* :: '*a nat-term-order* ⇒ '*a*::*nat-term comparator*
where *nat-term-compare-inv to* = (λ*x y*. *nat-term-compare to y x*)

definition *key-order-of-nat-term-order* :: '*a nat-term-order* ⇒ '*a*::*nat-term key-order*

where *key-order-of-nat-term-order-def* [*code del*]:
key-order-of-nat-term-order to = *Abs-key-order* (*nat-term-compare to*)

definition *key-order-of-nat-term-order-inv* :: *'a nat-term-order* ⇒ *'a*::*nat-term key-order*
where *key-order-of-nat-term-order-inv-def* [*code del*]:
key-order-of-nat-term-order-inv to = *Abs-key-order* (*nat-term-compare-inv to*)

definition *le-of-nat-term-order* :: *'a nat-term-order* ⇒ *'a* ⇒ *'a*::*nat-term* ⇒ *bool*
where *le-of-nat-term-order to* = *le-of-key-order* (*key-order-of-nat-term-order to*)

definition *lt-of-nat-term-order* :: *'a nat-term-order* ⇒ *'a* ⇒ *'a*::*nat-term* ⇒ *bool*
where *lt-of-nat-term-order to* = *lt-of-key-order* (*key-order-of-nat-term-order to*)

definition *nat-term-order-of-le* :: *'a*::{*linorder*,*nat-term*} *nat-term-order*
where *nat-term-order-of-le* = *Abs-nat-term-order* (*comparator-of*)

lemma *comparator-nat-term-compare*: *comparator* (*nat-term-compare to*)
⟨*proof*⟩

lemma *nat-term-comp-nat-term-compare*: *nat-term-comp* (*nat-term-compare to*)
⟨*proof*⟩

lemma *nat-term-compare-splus*: *nat-term-compare to* (*splus t u*) (*splus t v*) =
nat-term-compare to u v
⟨*proof*⟩

lemma *nat-term-compare-conv*: *nat-term-compare to* = *key-compare* (*key-order-of-nat-term-order to*)
⟨*proof*⟩

lemma *comparator-nat-term-compare-inv*: *comparator* (*nat-term-compare-inv to*)
⟨*proof*⟩

lemma *nat-term-compare-inv-conv*: *nat-term-compare-inv to* = *key-compare* (*key-order-of-nat-term-order-inv to*)
⟨*proof*⟩

lemma *nat-term-compare-inv-alt* [*code-unfold*]: *nat-term-compare-inv to x y* = *nat-term-compare to y x*
⟨*proof*⟩

lemma *le-of-nat-term-order* [*code*]: *le-of-nat-term-order to x y* = (*nat-term-compare to x y* ≠ *Gt*)
⟨*proof*⟩

lemma *lt-of-nat-term-order* [*code*]: *lt-of-nat-term-order to x y* = (*nat-term-compare to x y* = *Lt*)
⟨*proof*⟩

lemma *le-of-nat-term-order-alt*:
le-of-nat-term-order to = (*λu v*. *ko.le* (*key-order-of-nat-term-order-inv to*) *v u*)
⟨*proof*⟩

lemma *lt-of-nat-term-order-alt*:
lt-of-nat-term-order to = (*λu v*. *ko.lt* (*key-order-of-nat-term-order-inv to*) *v u*)
⟨*proof*⟩

lemma *linorder-le-of-nat-term-order*: *class.linorder* (*le-of-nat-term-order to*) (*lt-of-nat-term-order to*)
⟨*proof*⟩

lemma *le-of-nat-term-order-zero-min*: *le-of-nat-term-order to 0* (*t*::*'a*::*nat-pp-term*)
⟨*proof*⟩

lemma *le-of-nat-term-order-plus-monotone*:
assumes *le-of-nat-term-order to s* (*t*::*'a*::*nat-pp-term*)
shows *le-of-nat-term-order to* (*u* + *s*) (*u* + *t*)
⟨*proof*⟩

global-interpretation *ko-ntm*: *comparator nat-term-compare-inv ko*
defines *lookup-pair-ko-ntm* = *ko-ntm.lookup-pair*
and *update-by-pair-ko-ntm* = *ko-ntm.update-by-pair*
and *update-by-fun-pair-ko-ntm* = *ko-ntm.update-by-fun-pair*
and *update-by-fun-gr-pair-ko-ntm* = *ko-ntm.update-by-fun-gr-pair*
and *map2-val-pair-ko-ntm* = *ko-ntm.map2-val-pair*
and *lex-ord-pair-ko-ntm* = *ko-ntm.lex-ord-pair*
and *prod-ord-pair-ko-ntm* = *ko-ntm.prod-ord-pair*
and *sort-oalist-ko-ntm′* = *ko-ntm.sort-oalist*
⟨*proof*⟩

lemma *ko-ntm-le*: *ko-ntm.le to* = (*λx y*. *le-of-nat-term-order to y x*)
⟨*proof*⟩

global-interpretation *ko-ntm*: *oalist-raw key-order-of-nat-term-order-inv*
rewrites *comparator.lookup-pair* (*key-compare* (*key-order-of-nat-term-order-inv ko*)) = *lookup-pair-ko-ntm ko*
and *comparator.update-by-pair* (*key-compare* (*key-order-of-nat-term-order-inv ko*)) = *update-by-pair-ko-ntm ko*
and *comparator.update-by-fun-pair* (*key-compare* (*key-order-of-nat-term-order-inv ko*)) = *update-by-fun-pair-ko-ntm ko*
and *comparator.update-by-fun-gr-pair* (*key-compare* (*key-order-of-nat-term-order-inv ko*)) = *update-by-fun-gr-pair-ko-ntm ko*
and *comparator.map2-val-pair* (*key-compare* (*key-order-of-nat-term-order-inv ko*)) = *map2-val-pair-ko-ntm ko*
and *comparator.lex-ord-pair* (*key-compare* (*key-order-of-nat-term-order-inv ko*)) = *lex-ord-pair-ko-ntm ko*
and *comparator.prod-ord-pair* (*key-compare* (*key-order-of-nat-term-order-inv ko*)) = *prod-ord-pair-ko-ntm ko*

and *comparator.sort-oalist* (*key-compare* (*key-order-of-nat-term-order-inv ko*)) = *sort-oalist-ko-ntm′ ko*
defines *sort-oalist-aux-ko-ntm* = *ko-ntm.sort-oalist-aux*
and *lookup-ko-ntm* = *ko-ntm.lookup-raw*
and *sorted-domain-ko-ntm* = *ko-ntm.sorted-domain-raw*
and *tl-ko-ntm* = *ko-ntm.tl-raw*
and *min-key-val-ko-ntm* = *ko-ntm.min-key-val-raw*
and *update-by-ko-ntm* = *ko-ntm.update-by-raw*
and *update-by-fun-ko-ntm* = *ko-ntm.update-by-fun-raw*
and *update-by-fun-gr-ko-ntm* = *ko-ntm.update-by-fun-gr-raw*
and *map2-val-ko-ntm* = *ko-ntm.map2-val-raw*
and *lex-ord-ko-ntm* = *ko-ntm.lex-ord-raw*
and *prod-ord-ko-ntm* = *ko-ntm.prod-ord-raw*
and *oalist-eq-ko-ntm* = *ko-ntm.oalist-eq-raw*
and *sort-oalist-ko-ntm* = *ko-ntm.sort-oalist-raw*
⟨*proof*⟩

lemma *compute-min-key-val-ko-ntm* [*code*]:
min-key-val-ko-ntm ko (*xs*, *ox*) =
(*if ko* = *ox then hd else min-list-param* (λ*x y*. (*le-of-nat-term-order ko*) (*fst y*) (*fst x*))) *xs*
⟨*proof*⟩

typedef (**overloaded**) (′*a*, ′*b*) *oalist-ntm* =
{*xs*::(′*a*, ′*b*::*zero*, ′*a*::*nat-term nat-term-order*) *oalist-raw*. *ko-ntm.oalist-inv xs*}
morphisms *list-of-oalist-ntm Abs-oalist-ntm*
⟨*proof*⟩

lemma *oalist-ntm-eq-iff*: *xs* = *ys* ⟷ *list-of-oalist-ntm xs* = *list-of-oalist-ntm ys*
⟨*proof*⟩

lemma *oalist-ntm-eqI*: *list-of-oalist-ntm xs* = *list-of-oalist-ntm ys* ⟹ *xs* = *ys*
⟨*proof*⟩

Formal, totalized constructor for (′*a*, ′*b*) *oalist-ntm*:

definition *OAlist-ntm* :: (′*a* × ′*b*) *list* × ′*a nat-term-order* ⇒ (′*a*::*nat-term*, ′*b*::*zero*) *oalist-ntm*
where *OAlist-ntm xs* = *Abs-oalist-ntm* (*sort-oalist-ko-ntm xs*)

definition *oalist-of-list-ntm* = *OAlist-ntm*

lemma *oalist-inv-list-of-oalist-ntm*: *ko-ntm.oalist-inv* (*list-of-oalist-ntm xs*)
⟨*proof*⟩

lemma *list-of-oalist-OAlist-ntm*: *list-of-oalist-ntm* (*OAlist-ntm xs*) = *sort-oalist-ko-ntm xs*
⟨*proof*⟩

lemma *OAlist-list-of-oalist-ntm* [*simp*, *code abstype*]: *OAlist-ntm* (*list-of-oalist-ntm*

xs) = *xs*
⟨*proof*⟩

lemma [*code abstract*]: *list-of-oalist-ntm* (*oalist-of-list-ntm xs*) = *sort-oalist-ko-ntm xs*
⟨*proof*⟩

global-interpretation *oa-ntm*: *oalist-abstract key-order-of-nat-term-order-inv list-of-oalist-ntm OAlist-ntm*
defines *OAlist-lookup-ntm* = *oa-ntm.lookup*
and *OAlist-sorted-domain-ntm* = *oa-ntm.sorted-domain*
and *OAlist-empty-ntm* = *oa-ntm.empty*
and *OAlist-reorder-ntm* = *oa-ntm.reorder*
and *OAlist-tl-ntm* = *oa-ntm.tl*
and *OAlist-hd-ntm* = *oa-ntm.hd*
and *OAlist-except-min-ntm* = *oa-ntm.except-min*
and *OAlist-min-key-val-ntm* = *oa-ntm.min-key-val*
and *OAlist-insert-ntm* = *oa-ntm.insert*
and *OAlist-update-by-fun-ntm* = *oa-ntm.update-by-fun*
and *OAlist-update-by-fun-gr-ntm* = *oa-ntm.update-by-fun-gr*
and *OAlist-filter-ntm* = *oa-ntm.filter*
and *OAlist-map2-val-neutr-ntm* = *oa-ntm.map2-val-neutr*
and *OAlist-eq-ntm* = *oa-ntm.oalist-eq*
⟨*proof*⟩

global-interpretation *oa-ntm*: *oalist-abstract3 key-order-of-nat-term-order-inv*
list-of-oalist-ntm::('*a*, '*b*) *oalist-ntm* ⇒ ('*a*, '*b*::*zero*, '*a*::*nat-term nat-term-order*) *oalist-raw OAlist-ntm*
list-of-oalist-ntm::('*a*, '*c*) *oalist-ntm* ⇒ ('*a*, '*c*::*zero*, '*a nat-term-order*) *oalist-raw OAlist-ntm*
list-of-oalist-ntm::('*a*, '*d*) *oalist-ntm* ⇒ ('*a*, '*d*::*zero*, '*a nat-term-order*) *oalist-raw OAlist-ntm*
defines *OAlist-map-val-ntm* = *oa-ntm.map-val*
and *OAlist-map2-val-ntm* = *oa-ntm.map2-val*
and *OAlist-map2-val-rneutr-ntm* = *oa-ntm.map2-val-rneutr*
and *OAlist-lex-ord-ntm* = *oa-ntm.lex-ord*
and *OAlist-prod-ord-ntm* = *oa-ntm.prod-ord* ⟨*proof*⟩

lemmas *OAlist-lookup-ntm-single* = *oa-ntm.lookup-oalist-of-list-single*[*folded oalist-of-list-ntm-def*]

end

14 Computable Term Orders

theory *Term-Order*
imports *OAlist-Poly-Mapping HOL−Library.Product-Lexorder*
begin

14.1 Type Class *nat*

class *nat* = *zero* + *plus* + *minus* + *order* + *equal* +
  fixes *rep-nat* :: *'a* ⇒ *nat*
    and *abs-nat* :: *nat* ⇒ *'a*
  assumes *rep-inverse* [*simp*]: *abs-nat* (*rep-nat x*) = *x*
    and *abs-inverse* [*simp*]: *rep-nat* (*abs-nat n*) = *n*
    and *abs-zero* [*simp*]: *abs-nat 0* = *0*
    and *abs-plus*: *abs-nat m* + *abs-nat n* = *abs-nat* (*m* + *n*)
    and *abs-minus*: *abs-nat m* − *abs-nat n* = *abs-nat* (*m* − *n*)
    and *abs-ord*: *m* ≤ *n* ⟹ *abs-nat m* ≤ *abs-nat n*
begin

lemma *rep-inj*:
  assumes *rep-nat x* = *rep-nat y*
  shows *x* = *y*
⟨*proof*⟩

corollary *rep-eq-iff*: (*rep-nat x* = *rep-nat y*) ⟷ (*x* = *y*)
  ⟨*proof*⟩

lemma *abs-inj*:
  assumes *abs-nat m* = *abs-nat n*
  shows *m* = *n*
⟨*proof*⟩

corollary *abs-eq-iff*: (*abs-nat m* = *abs-nat n*) ⟷ (*m* = *n*)
  ⟨*proof*⟩

lemma *rep-zero* [*simp*]: *rep-nat 0* = *0*
  ⟨*proof*⟩

lemma *rep-zero-iff*: (*rep-nat x* = *0*) ⟷ (*x* = *0*)
  ⟨*proof*⟩

lemma *plus-eq*: *x* + *y* = *abs-nat* (*rep-nat x* + *rep-nat y*)
  ⟨*proof*⟩

lemma *rep-plus*: *rep-nat* (*x* + *y*) = *rep-nat x* + *rep-nat y*
  ⟨*proof*⟩

lemma *minus-eq*: *x* − *y* = *abs-nat* (*rep-nat x* − *rep-nat y*)
  ⟨*proof*⟩

lemma *rep-minus*: *rep-nat* (*x* − *y*) = *rep-nat x* − *rep-nat y*
  ⟨*proof*⟩

lemma *ord-iff*:
  x ≤ *y* ⟷ *rep-nat x* ≤ *rep-nat y* (**is** *?thesis1*)
  x < *y* ⟷ *rep-nat x* < *rep-nat y* (**is** *?thesis2*)

⟨*proof*⟩

lemma *ex-iff-abs*: $(\exists\, x{::}'a.\ P\ x) \longleftrightarrow (\exists\, n{::}nat.\ P\ (abs\text{-}nat\ n))$
⟨*proof*⟩

lemma *ex-iff-abs'*: $(\exists\, x{<}abs\text{-}nat\ m.\ P\ x) \longleftrightarrow (\exists\, n{::}nat{<}m.\ P\ (abs\text{-}nat\ n))$
⟨*proof*⟩

lemma *all-iff-abs*: $(\forall\, x{::}'a.\ P\ x) \longleftrightarrow (\forall\, n{::}nat.\ P\ (abs\text{-}nat\ n))$
⟨*proof*⟩

lemma *all-iff-abs'*: $(\forall\, x{<}abs\text{-}nat\ m.\ P\ x) \longleftrightarrow (\forall\, n{::}nat{<}m.\ P\ (abs\text{-}nat\ n))$
⟨*proof*⟩

subclass *linorder* ⟨*proof*⟩

lemma *comparator-of-rep* [*simp*]: *comparator-of* (*rep-nat x*) (*rep-nat y*) = *comparator-of x y*
⟨*proof*⟩

subclass *wellorder*
⟨*proof*⟩

subclass *comm-monoid-add* ⟨*proof*⟩

lemma *sum-rep*: *sum* (*rep-nat* ∘ *f*) *A* = *rep-nat* (*sum f A*) **for** *f* :: *'b* ⇒ *'a* **and** *A* :: *'b set*
⟨*proof*⟩

subclass *ordered-comm-monoid-add* ⟨*proof*⟩

subclass *countable* ⟨*proof*⟩

subclass *cancel-comm-monoid-add*
⟨*proof*⟩

subclass *add-wellorder*
⟨*proof*⟩

end

lemma *the-min-eq-zero*: *the-min* = (*0*::*'a*::{*the-min*,*nat*})
⟨*proof*⟩

instantiation *nat* :: *nat*
begin

definition *rep-nat-nat* :: *nat* ⇒ *nat* **where** *rep-nat-nat-def* [*code-unfold*]: *rep-nat-nat* = (λ*x*. *x*)

definition *abs-nat-nat* :: *nat* ⇒ *nat* **where** *abs-nat-nat-def* [*code-unfold*]: *abs-nat-nat* = (λ*x*. *x*)

instance ⟨*proof*⟩

end

instantiation *natural* :: *nat*
begin

definition *rep-nat-natural* :: *natural* ⇒ *nat*
  where *rep-nat-natural-def* [*code-unfold*]: *rep-nat-natural* = *nat-of-natural*
definition *abs-nat-natural* :: *nat* ⇒ *natural*
  where *abs-nat-natural-def* [*code-unfold*]: *abs-nat-natural* = *natural-of-nat*

instance ⟨*proof*⟩

end

14.2 Term Orders

14.2.1 Type Classes

class *nat-pp-compare* = *linorder* + *zero* + *plus* +
  fixes *rep-nat-pp* :: *'a* ⇒ (*nat*, *nat*) *pp*
    and *abs-nat-pp* :: (*nat*, *nat*) *pp* ⇒ *'a*
    and *lex-comp'* :: *'a comparator*
    and *deg'* :: *'a* ⇒ *nat*
  assumes *rep-nat-pp-inverse* [*simp*]: *abs-nat-pp* (*rep-nat-pp x*) = *x*
    and *abs-nat-pp-inverse* [*simp*]: *rep-nat-pp* (*abs-nat-pp t*) = *t*
    and *lex-comp'*: *lex-comp' x y* = *comp-of-ord lex-pp* (*rep-nat-pp x*) (*rep-nat-pp y*)
    and *deg'*: *deg' x* = *deg-pp* (*rep-nat-pp x*)
    and *le-pp*: *rep-nat-pp x* ≤ *rep-nat-pp y* ⟹ *x* ≤ *y*
    and *zero-pp*: *rep-nat-pp 0* = *0*
    and *plus-pp*: *rep-nat-pp* (*x* + *y*) = *rep-nat-pp x* + *rep-nat-pp y*
begin

lemma *less-pp*:
  assumes *rep-nat-pp x* < *rep-nat-pp y*
  shows *x* < *y*
⟨*proof*⟩

lemma *rep-nat-pp-inj*:
  assumes *rep-nat-pp x* = *rep-nat-pp y*
  shows *x* = *y*
⟨*proof*⟩

lemma *lex-comp'-EqD*:
  assumes *lex-comp' x y* = *Eq*

shows $x = y$
⟨*proof*⟩

lemma *lex-comp'-valE*:
assumes *lex-comp' s t* ≠ *Eq*
obtains *x* **where** *x* ∈ *keys-pp* (*rep-nat-pp s*) ∪ *keys-pp* (*rep-nat-pp t*)
and *comparator-of* (*lookup-pp* (*rep-nat-pp s*) *x*) (*lookup-pp* (*rep-nat-pp t*) *x*) = *lex-comp' s t*
and ⋀*y*. *y* < *x* ⟹ *lookup-pp* (*rep-nat-pp s*) *y* = *lookup-pp* (*rep-nat-pp t*) *y*
⟨*proof*⟩

end

class *nat-term-compare* = *linorder* + *nat-term* +
fixes *is-scalar* :: *'a itself* ⇒ *bool*
and *lex-comp* :: *'a comparator*
and *deg-comp* :: *'a comparator* ⇒ *'a comparator*
and *pot-comp* :: *'a comparator* ⇒ *'a comparator*
assumes *zero-component*: ∃ *x*. *snd* (*rep-nat-term x*) = *0*
and *is-scalar*: *is-scalar* = (λ-. ∀ *x*. *snd* (*rep-nat-term x*) = *0*)
and *lex-comp*: *lex-comp* = *lex-comp-aux* — For being able to implement *lex-comp* efficiently.
and *deg-comp*: *deg-comp cmp* = (λ*x y*. *case comparator-of* (*deg-pp* (*fst* (*rep-nat-term x*))) (*deg-pp* (*fst* (*rep-nat-term y*))) *of Eq* ⇒ *cmp x y* | *val* ⇒ *val*)
and *pot-comp*: *pot-comp cmp* = (λ*x y*. *case comparator-of* (*snd* (*rep-nat-term x*)) (*snd* (*rep-nat-term y*)) *of Eq* ⇒ *cmp x y* | *val* ⇒ *val*)
and *le-term*: *rep-nat-term x* ≤ *rep-nat-term y* ⟹ *x* ≤ *y*
begin

There is no need to add something like *top-comp* for TOP orders to class *nat-term-compare*, because by default all comparators should *first* compare power-products and *then* positions. *lex-comp* obviously does.

lemma *less-term*:
assumes *rep-nat-term x* < *rep-nat-term y*
shows *x* < *y*
⟨*proof*⟩

lemma *lex-comp-alt*: *lex-comp* = (*comparator-of*::*'a comparator*)
⟨*proof*⟩

lemma *full-component-zeroE*: **obtains** *x* **where** *rep-nat-term x* = (*t*, *0*)
⟨*proof*⟩

end

lemma *comparator-lex-comp*: *comparator lex-comp*
⟨*proof*⟩

lemma *nat-term-comp-lex-comp*: *nat-term-comp lex-comp*
⟨*proof*⟩

lemma *comparator-deg-comp*:
assumes *comparator cmp*
shows *comparator* (*deg-comp cmp*)
⟨*proof*⟩

lemma *comparator-pot-comp*:
assumes *comparator cmp*
shows *comparator* (*pot-comp cmp*)
⟨*proof*⟩

lemma *deg-comp-zero-min*:
assumes *comparator cmp* **and** *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*) **and** *fst* (*rep-nat-term u*) = *0*
shows *deg-comp cmp u v* ≠ *Gt*
⟨*proof*⟩

lemma *deg-comp-pos*:
assumes *cmp u v* = *Lt* **and** *fst* (*rep-nat-term u*) = *fst* (*rep-nat-term v*)
shows *deg-comp cmp u v* = *Lt*
⟨*proof*⟩

lemma *deg-comp-monotone*:
assumes *cmp u v* = *Lt* ⟹ *cmp* (*splus t u*) (*splus t v*) = *Lt* **and** *deg-comp cmp u v* = *Lt*
shows *deg-comp cmp* (*splus t u*) (*splus t v*) = *Lt*
⟨*proof*⟩

lemma *pot-comp-zero-min*:
assumes *cmp u v* ≠ *Gt* **and** *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*)
shows *pot-comp cmp u v* ≠ *Gt*
⟨*proof*⟩

lemma *pot-comp-pos*:
assumes *snd* (*rep-nat-term u*) < *snd* (*rep-nat-term v*)
shows *pot-comp cmp u v* = *Lt*
⟨*proof*⟩

lemma *pot-comp-monotone*:
assumes *cmp u v* = *Lt* ⟹ *cmp* (*splus t u*) (*splus t v*) = *Lt* **and** *pot-comp cmp u v* = *Lt*
shows *pot-comp cmp* (*splus t u*) (*splus t v*) = *Lt*
⟨*proof*⟩

lemma *deg-comp-cong*:
assumes *deg-pp* (*fst* (*rep-nat-term u*)) = *deg-pp* (*fst* (*rep-nat-term v*)) ⟹ *to1*

u v = to2 u v
shows *deg-comp to1 u v = deg-comp to2 u v*
⟨*proof*⟩

lemma *pot-comp-cong*:
assumes *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*) ⟹ *to1 u v = to2 u v*
shows *pot-comp to1 u v = pot-comp to2 u v*
⟨*proof*⟩

instantiation *pp* :: (*nat*, *nat*) *nat-pp-compare*
begin

definition *rep-nat-pp-pp* :: ('*a*, '*b*) *pp* ⇒ (*nat*, *nat*) *pp*
where *rep-nat-pp-pp-def* [*code del*]: *rep-nat-pp-pp x = pp-of-fun* (λ*n*::*nat*. *rep-nat* (*lookup-pp x* (*abs-nat n*)))

definition *abs-nat-pp-pp* :: (*nat*, *nat*) *pp* ⇒ ('*a*, '*b*) *pp*
where *abs-nat-pp-pp-def* [*code del*]: *abs-nat-pp-pp t = pp-of-fun* (λ*n*::'*a*. *abs-nat* (*lookup-pp t* (*rep-nat n*)))

definition *lex-comp'-pp* :: ('*a*, '*b*) *pp comparator*
where *lex-comp'-pp-def* [*code del*]: *lex-comp'-pp = comp-of-ord lex-pp*

definition *deg'-pp* :: ('*a*, '*b*) *pp* ⇒ *nat*
where *deg'-pp x = rep-nat* (*deg-pp x*)

lemma *lookup-rep-nat-pp-pp*:
lookup-pp (*rep-nat-pp t*) = (λ*n*::*nat*. *rep-nat* (*lookup-pp t* (*abs-nat n*)))
⟨*proof*⟩

lemma *lookup-abs-nat-pp-pp*:
lookup-pp (*abs-nat-pp t*) = (λ*n*::'*a*. *abs-nat* (*lookup-pp t* (*rep-nat n*)))
⟨*proof*⟩

lemma *keys-rep-nat-pp-pp*: *keys-pp* (*rep-nat-pp t*) = *rep-nat* ‘ *keys-pp t*
⟨*proof*⟩

lemma *rep-nat-pp-pp-inverse*: *abs-nat-pp* (*rep-nat-pp x*) = *x* **for** *x*::('*a*, '*b*) *pp*
⟨*proof*⟩

lemma *abs-nat-pp-pp-inverse*: *rep-nat-pp* ((*abs-nat-pp t*)::('*a*, '*b*) *pp*) = *t*
⟨*proof*⟩

corollary *rep-nat-pp-pp-inj*:
fixes *x y* :: ('*a*, '*b*) *pp*
assumes *rep-nat-pp x = rep-nat-pp y*
shows *x = y*
⟨*proof*⟩

corollary *rep-nat-pp-pp-eq-iff*: $(\mathit{rep\text{-}nat\text{-}pp}\ x = \mathit{rep\text{-}nat\text{-}pp}\ y) \longleftrightarrow (x = y)$ **for** x $y :: ('a, 'b)\ pp$
⟨*proof*⟩

lemma *lex-rep-nat-pp*: $\mathit{lex\text{-}pp}\ (\mathit{rep\text{-}nat\text{-}pp}\ x)\ (\mathit{rep\text{-}nat\text{-}pp}\ y) \longleftrightarrow \mathit{lex\text{-}pp}\ x\ y$
⟨*proof*⟩

corollary *lex-comp'-pp*: $\mathit{lex\text{-}comp'}\ x\ y = \mathit{comp\text{-}of\text{-}ord}\ \mathit{lex\text{-}pp}\ (\mathit{rep\text{-}nat\text{-}pp}\ x)\ (\mathit{rep\text{-}nat\text{-}pp}\ y)$ **for** $x\ y :: ('a, 'b)\ pp$
⟨*proof*⟩

corollary *le-pp-pp*: $\mathit{rep\text{-}nat\text{-}pp}\ x \leq \mathit{rep\text{-}nat\text{-}pp}\ y \Longrightarrow x \leq y$ **for** $x\ y :: ('a, 'b)\ pp$
⟨*proof*⟩

lemma *deg-rep-nat-pp*: $\mathit{deg\text{-}pp}\ (\mathit{rep\text{-}nat\text{-}pp}\ t) = \mathit{rep\text{-}nat}\ (\mathit{deg\text{-}pp}\ t)$ **for** $t :: ('a, 'b)\ pp$
⟨*proof*⟩

corollary *deg'-pp*: $\mathit{deg'}\ t = \mathit{deg\text{-}pp}\ (\mathit{rep\text{-}nat\text{-}pp}\ t)$ **for** $t :: ('a, 'b)\ pp$
⟨*proof*⟩

lemma *zero-pp-pp*: $\mathit{rep\text{-}nat\text{-}pp}\ (0::('a, 'b)\ pp) = 0$
⟨*proof*⟩

lemma *plus-pp-pp*: $\mathit{rep\text{-}nat\text{-}pp}\ (x + y) = \mathit{rep\text{-}nat\text{-}pp}\ x + \mathit{rep\text{-}nat\text{-}pp}\ y$
for $x\ y :: ('a, 'b)\ pp$
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation $pp :: (nat, nat)\ nat\text{-}term$
begin

definition *rep-nat-term-pp* :: $('a, 'b)\ pp \Rightarrow (nat, nat)\ pp \times nat$
where *rep-nat-term-pp-def* [*code del*]: $\mathit{rep\text{-}nat\text{-}term\text{-}pp}\ t = (\mathit{rep\text{-}nat\text{-}pp}\ t, 0)$

definition *splus-pp* :: $('a, 'b)\ pp \Rightarrow ('a, 'b)\ pp \Rightarrow ('a, 'b)\ pp$
where *splus-pp-def* [*code del*]: $\mathit{splus\text{-}pp} = (+)$

instance ⟨*proof*⟩

end

instantiation $pp :: (nat, nat)\ nat\text{-}term\text{-}compare$
begin

definition *is-scalar-pp* :: ('*a*, '*b*) *pp itself* ⇒ *bool*
where *is-scalar-pp-def* [*code-unfold*]: *is-scalar-pp* = (λ-. *True*)

definition *lex-comp-pp* :: ('*a*, '*b*) *pp comparator*
where *lex-comp-pp-def* [*code-unfold*]: *lex-comp-pp* = *lex-comp'*

definition *deg-comp-pp* :: ('*a*, '*b*) *pp comparator* ⇒ ('*a*, '*b*) *pp comparator*
where *deg-comp-pp-def*: *deg-comp-pp cmp* = (λ*x y*. *case comparator-of* (*deg-pp x*) (*deg-pp y*) *of Eq* ⇒ *cmp x y* | *val* ⇒ *val*)

definition *pot-comp-pp* :: ('*a*, '*b*) *pp comparator* ⇒ ('*a*, '*b*) *pp comparator*
where *pot-comp-pp-def* [*code-unfold*]: *pot-comp-pp* = (λ*cmp*. *cmp*)

instance ⟨*proof*⟩

end

instance *pp* :: (*nat*, *nat*) *nat-pp-term*
⟨*proof*⟩

instantiation *prod* :: ({*nat-pp-compare*, *comm-powerprod*}, *nat*) *nat-term*
begin

definition *rep-nat-term-prod* :: ('*a* × '*b*) ⇒ ((*nat*, *nat*) *pp* × *nat*)
where *rep-nat-term-prod-def* [*code del*]: *rep-nat-term-prod u* = (*rep-nat-pp* (*fst u*), *rep-nat* (*snd u*))

definition *splus-prod* :: ('*a* × '*b*) ⇒ ('*a* × '*b*) ⇒ ('*a* × '*b*)
where *splus-prod-def* [*code del*]: *splus-prod t u* = *pprod.splus* (*fst t*) *u*

instance ⟨*proof*⟩

end

instantiation *prod* :: ({*nat-pp-compare*, *comm-powerprod*}, *nat*) *nat-term-compare*
begin

definition *is-scalar-prod* :: ('*a* × '*b*) *itself* ⇒ *bool*
where *is-scalar-prod-def* [*code-unfold*]: *is-scalar-prod* = (λ-. *False*)

definition *lex-comp-prod* :: ('*a* × '*b*) *comparator*
where *lex-comp-prod* = (λ*u v*. *case lex-comp'* (*fst u*) (*fst v*) *of Eq* ⇒ *comparator-of* (*snd u*) (*snd v*) | *val* ⇒ *val*)

definition *deg-comp-prod* :: ('*a* × '*b*) *comparator* ⇒ ('*a* × '*b*) *comparator*
where *deg-comp-prod-def*: *deg-comp-prod cmp* = (λ*x y*. *case comparator-of* (*deg'* (*fst x*)) (*deg'* (*fst y*)) *of Eq* ⇒ *cmp x y* | *val* ⇒ *val*)

definition *pot-comp-prod* :: ('*a* × '*b*) *comparator* ⇒ ('*a* × '*b*) *comparator*

where *pot-comp-prod cmp* = (λ*u v*. *case comparator-of* (*snd u*) (*snd v*) *of Eq* ⇒ *cmp u v* | *val* ⇒ *val*)

instance ⟨*proof*⟩

end

lemmas [*code del*] = *deg-pp.rep-eq plus-pp.abs-eq minus-pp.abs-eq*

lemma *rep-nat-pp-nat* [*code-unfold*]: (*rep-nat-pp*::(*nat*, *nat*) *pp* ⇒ (*nat*, *nat*) *pp*) = (λ*x*. *x*)
⟨*proof*⟩

14.2.2 *LEX*, *DRLEX*, *DEG* and *POT*

definition *LEX* :: 'a::*nat-term-compare nat-term-order* **where** *LEX* = *Abs-nat-term-order lex-comp*

definition *DRLEX* :: 'a::*nat-term-compare nat-term-order*
where *DRLEX* = *Abs-nat-term-order* (*deg-comp* (*pot-comp* (λ*x y*. *lex-comp y x*)))

definition *DEG* :: 'a::*nat-term-compare nat-term-order* ⇒ 'a *nat-term-order*
where *DEG to* = *Abs-nat-term-order* (*deg-comp* (*nat-term-compare to*))

definition *POT* :: 'a::*nat-term-compare nat-term-order* ⇒ 'a *nat-term-order*
where *POT to* = *Abs-nat-term-order* (*pot-comp* (*nat-term-compare to*))

DRLEX must apply *pot-comp*, for otherwise it does not satisfy the second condition of *nat-term-comp*.

Instead of *DRLEX* one could also introduce another unary constructor *DEGREV*, analogous to *DEG* and *POT*. Then, however, proving (in)equalities of the term orders gets really messy (think of *DEG* (*POT to*) = *DEGREV* (*DEGREV to*), for instance). So, we restrict the formalization to *DRLEX* only.

abbreviation *DLEX* ≡ *DEG LEX*

code-datatype *LEX DRLEX DEG POT*

lemma *nat-term-compare-LEX* [*code*]: *nat-term-compare LEX* = *lex-comp*
⟨*proof*⟩

lemma *nat-term-compare-DRLEX* [*code*]: *nat-term-compare DRLEX* = *deg-comp* (*pot-comp* (λ*x y*. *lex-comp y x*))
⟨*proof*⟩

lemma *nat-term-compare-DEG* [*code*]: *nat-term-compare* (*DEG to*) = *deg-comp* (*nat-term-compare to*)

⟨*proof*⟩

lemma *nat-term-compare-POT* [*code*]: *nat-term-compare* (*POT to*) = *pot-comp* (*nat-term-compare to*)
⟨*proof*⟩

lemma *nat-term-compare-POT-DRLEX* [*code*]:
nat-term-compare (*POT DRLEX*) = *pot-comp* (*deg-comp* (λ*x y*. *lex-comp y x*))
⟨*proof*⟩

lemma *compute-lex-pp* [*code*]: *lex-pp p q* = (*lex-comp*′ *p q* ≠ *Gt*)
⟨*proof*⟩

lemma *compute-dlex-pp* [*code*]: *dlex-pp p q* = (*deg-comp lex-comp*′ *p q* ≠ *Gt*)
⟨*proof*⟩

lemma *compute-drlex-pp* [*code*]: *drlex-pp p q* = (*deg-comp* (λ*x y*. *lex-comp*′ *y x*) *p q* ≠ *Gt*)
⟨*proof*⟩

lemma *nat-pp-order-of-le-nat-pp* [*code*]: *nat-term-order-of-le* = *LEX*
⟨*proof*⟩

14.2.3 Equality of Term Orders

definition *nat-term-order-eq* :: ′*a nat-term-order* ⇒ ′*a*::*nat-term-compare nat-term-order* ⇒ *bool* ⇒ *bool* ⇒ *bool*
where *nat-term-order-eq-def* [*code del*]:
nat-term-order-eq to1 to2 dg ps =
(∀ *u v*. (*dg* ⟶ *deg-pp* (*fst* (*rep-nat-term u*)) = *deg-pp* (*fst* (*rep-nat-term v*))) ⟶
(*ps* ⟶ *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*)) ⟶
nat-term-compare to1 u v = *nat-term-compare to2 u v*)

lemma *nat-term-order-eqI*:
assumes ⋀*u v*. (*dg* ⟹ *deg-pp* (*fst* (*rep-nat-term u*)) = *deg-pp* (*fst* (*rep-nat-term v*))) ⟹
(*ps* ⟹ *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*)) ⟹
nat-term-compare to1 u v = *nat-term-compare to2 u v*
shows *nat-term-order-eq to1 to2 dg ps*
⟨*proof*⟩

lemma *nat-term-order-eqD*:
assumes *nat-term-order-eq to1 to2 dg ps*
and *dg* ⟹ *deg-pp* (*fst* (*rep-nat-term u*)) = *deg-pp* (*fst* (*rep-nat-term v*))
and *ps* ⟹ *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*)
shows *nat-term-compare to1 u v* = *nat-term-compare to2 u v*
⟨*proof*⟩

lemma *nat-term-order-eq-sym*: *nat-term-order-eq to1 to2 dg ps* $\longleftrightarrow$ *nat-term-order-eq to2 to1 dg ps*
⟨*proof*⟩

lemma *nat-term-order-eq-DEG-dg*:
nat-term-order-eq (*DEG to1*) *to2 True ps* $\longleftrightarrow$ *nat-term-order-eq to1 to2 True ps*
⟨*proof*⟩

lemma *nat-term-order-eq-DEG-dg'*:
nat-term-order-eq to1 (*DEG to2*) *True ps* $\longleftrightarrow$ *nat-term-order-eq to1 to2 True ps*
⟨*proof*⟩

lemma *nat-term-order-eq-POT-ps*:
assumes *ps* $\vee$ *is-scalar TYPE*('*a*::*nat-term-compare*)
shows *nat-term-order-eq* (*POT* (*to1*::'*a nat-term-order*)) *to2 dg ps* $\longleftrightarrow$ *nat-term-order-eq to1 to2 dg ps*
⟨*proof*⟩

lemma *nat-term-order-eq-POT-ps'*:
assumes *ps* $\vee$ *is-scalar TYPE*('*a*::*nat-term-compare*)
shows *nat-term-order-eq to1* (*POT* (*to2*::'*a nat-term-order*)) *dg ps* $\longleftrightarrow$ *nat-term-order-eq to1 to2 dg ps*
⟨*proof*⟩

lemma *snd-rep-nat-term-eqI*:
assumes *ps* $\vee$ *is-scalar TYPE*('*a*::*nat-term-compare*) **and** *ps* $\Longrightarrow$ *snd* (*rep-nat-term* (*u*::'*a*)) = *snd* (*rep-nat-term* (*v*::'*a*))
shows *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*)
⟨*proof*⟩

definition *of-exps* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ '*a*::*nat-term-compare*
where *of-exps a b i* =
(*THE u. rep-nat-term u* = (*pp-of-fun* (λx. *if x* = *0 then a else if x* = *1 then b else 0*),
if ($\exists$ *v*::'*a. snd* (*rep-nat-term v*) = *i*) *then i else 0*))

of-exps is an auxiliary function needed for proving the equalities of the various term orders.

lemma *rep-nat-term-of-exps*:
rep-nat-term ((*of-exps a b i*)::'*a*::*nat-term-compare*) =
(*pp-of-fun* (λx::*nat. if x* = *0 then a else if x* = *1 then b else 0*), *if* ($\exists$ *y*::'*a. snd* (*rep-nat-term y*) = *i*) *then i else 0*)
⟨*proof*⟩

lemma *lookup-pp-of-exps*:
lookup-pp (*fst* (*rep-nat-term* (*of-exps a b i*))) = (λx. *if x* = *0 then a else if x* = *1 then b else 0*)
⟨*proof*⟩

lemma *keys-pp-of-exps*: *keys-pp* (*fst* (*rep-nat-term* (*of-exps a b i*))) ⊆ {*0*, *1*}
⟨*proof*⟩

lemma *deg-pp-of-exps* [*simp*]: *deg-pp* (*fst* (*rep-nat-term* ((*of-exps a b i*)::'*a*::*nat-term-compare*)))
= *a* + *b*
⟨*proof*⟩

lemma *snd-of-exps*:
assumes *snd* (*rep-nat-term* (*x*::'*a*)) = *i*
shows *snd* (*rep-nat-term* ((*of-exps a b i*)::'*a*::*nat-term-compare*)) = *i*
⟨*proof*⟩

lemma *snd-of-exps-zero* [*simp*]: *snd* (*rep-nat-term* ((*of-exps a b 0*)::'*a*::*nat-term-compare*))
= *0*
⟨*proof*⟩

lemma *eq-of-exps*:
(*fst* (*rep-nat-term* (*of-exps a1 b1 i*)) = *fst* (*rep-nat-term* (*of-exps a2 b2 j*))) ⟷
(*a1* = *a2* ∧ *b1* = *b2*)
⟨*proof*⟩

lemma *lex-pp-of-exps*:
lex-pp (*fst* (*rep-nat-term* ((*of-exps a1 b1 i*)::'*a*))) (*fst* (*rep-nat-term* ((*of-exps a2
b2 j*)::'*a*::*nat-term-compare*))) ⟷
(*a1* < *a2* ∨ (*a1* = *a2* ∧ *b1* ≤ *b2*)) (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *LEX-eq* [*code*]:
nat-term-order-eq LEX (*LEX*::'*a nat-term-order*) *dg ps* = *True* (**is** *?thesis1*)
nat-term-order-eq LEX (*DRLEX*::'*a nat-term-order*) *dg ps* = *False* (**is** *?thesis2*)
nat-term-order-eq LEX (*DEG* (*to*::'*a nat-term-order*)) *dg ps* =
(*dg* ∧ *nat-term-order-eq LEX to dg ps*) (**is** *?thesis3*)
nat-term-order-eq LEX (*POT* (*to*::'*a nat-term-order*)) *dg ps* =
((*ps* ∨ *is-scalar TYPE*('*a*::*nat-term-compare*)) ∧ *nat-term-order-eq LEX to dg
ps*) (**is** *?thesis4*)
⟨*proof*⟩

lemma *DRLEX-eq* [*code*]:
nat-term-order-eq DRLEX (*LEX*::'*a nat-term-order*) *dg ps* = *False* (**is** *?thesis1*)
nat-term-order-eq DRLEX DRLEX dg ps = *True* (**is** *?thesis2*)
nat-term-order-eq DRLEX (*DEG* (*to*::'*a nat-term-order*)) *dg ps* =
nat-term-order-eq DRLEX to True ps (**is** *?thesis3*)
nat-term-order-eq DRLEX (*POT* (*to*::'*a nat-term-order*)) *dg ps* =
((*dg* ∨ *ps* ∨ *is-scalar TYPE*('*a*::*nat-term-compare*)) ∧ *nat-term-order-eq DRLEX
to dg True*) (**is** *?thesis4*)
⟨*proof*⟩

lemma *DEG-eq* [*code*]:
nat-term-order-eq (*DEG to*) (*LEX*::'*a nat-term-order*) *dg ps* = *nat-term-order-eq*

LEX (*DEG to*) *dg ps*
nat-term-order-eq (*DEG to*) (*DRLEX*::*'a nat-term-order*) *dg ps* = *nat-term-order-eq DRLEX* (*DEG to*) *dg ps*
nat-term-order-eq (*DEG to1*) (*DEG* (*to2*::*'a nat-term-order*)) *dg ps* =
nat-term-order-eq to1 to2 True ps (**is** *?thesis3*)
nat-term-order-eq (*DEG to1*) (*POT* (*to2*::*'a nat-term-order*)) *dg ps* =
(*if dg then nat-term-order-eq to1* (*POT to2*) *dg ps*
else ((*ps* ∨ *is-scalar TYPE*(*'a*::*nat-term-compare*)) ∧ *nat-term-order-eq* (*DEG to1*) *to2 dg ps*)) (**is** *?thesis4*)
⟨*proof*⟩

lemma *POT-eq* [*code*]:
nat-term-order-eq (*POT to*) *LEX dg ps* = *nat-term-order-eq LEX* (*POT to*) *dg ps*
nat-term-order-eq (*POT to1*) (*DEG to2*) *dg ps* = *nat-term-order-eq* (*DEG to2*) (*POT to1*) *dg ps*
nat-term-order-eq (*POT to1*) *DRLEX dg ps* = *nat-term-order-eq DRLEX* (*POT to1*) *dg ps*
nat-term-order-eq (*POT to1*) (*POT* (*to2*::*'a*::*nat-term-compare nat-term-order*)) *dg ps* =
nat-term-order-eq to1 to2 dg True (**is** *?thesis4*)
⟨*proof*⟩

lemma *nat-term-order-equal* [*code*]: *HOL.equal to1 to2* = *nat-term-order-eq to1 to2 False False*
⟨*proof*⟩

hide-const (**open**) *of-exps*

value [*code*] *DEG* (*POT DRLEX*) = (*DRLEX*::((*nat*, *nat*) *pp* × *nat*) *nat-term-order*)

value [*code*] *POT LEX* = (*LEX*::((*nat*, *nat*) *pp* × *nat*) *nat-term-order*)

value [*code*] *POT LEX* = (*LEX*::(*nat*, *nat*) *pp nat-term-order*)

end

15 Executable Representation of Polynomial Mappings as Association Lists

theory *MPoly-Type-Class-OAlist*
imports *Term-Order*
begin

instantiation *pp* :: (*type*, {*equal*, *zero*}) *equal*
begin

definition *equal-pp* :: (*'a*, *'b*) *pp* ⇒ (*'a*, *'b*) *pp* ⇒ *bool* **where**

equal-pp p q ≡ (∀ *t*. *lookup-pp p t* = *lookup-pp q t*)

instance ⟨*proof*⟩

end

instantiation *poly-mapping* :: (*type*, {*equal*, *zero*}) *equal*
begin

definition *equal-poly-mapping* :: ('*a*, '*b*) *poly-mapping* ⇒ ('*a*, '*b*) *poly-mapping* ⇒ *bool* **where**
equal-poly-mapping-def [*code del*]: *equal-poly-mapping p q* ≡ (∀ *t*. *lookup p t* = *lookup q t*)

instance ⟨*proof*⟩

end

15.1 Power-Products Represented by *oalist-tc*

definition *PP-oalist* :: ('*a*::*linorder*, '*b*::*zero*) *oalist-tc* ⇒ ('*a*, '*b*) *pp*
where *PP-oalist xs* = *pp-of-fun* (*OAlist-tc-lookup xs*)

code-datatype *PP-oalist*

lemma *lookup-PP-oalist* [*simp*, *code*]: *lookup-pp* (*PP-oalist xs*) = *OAlist-tc-lookup xs*
⟨*proof*⟩

lemma *keys-PP-oalist* [*code*]: *keys-pp* (*PP-oalist xs*) = *set* (*OAlist-tc-sorted-domain xs*)
⟨*proof*⟩

lemma *lex-comp-PP-oalist* [*code*]:
lex-comp' (*PP-oalist xs*) (*PP-oalist ys*) =
the (*OAlist-tc-lex-ord* (λ- *x y*. *Some* (*comparator-of x y*)) *xs ys*)
for *xs ys*::('*a*::*nat*, '*b*::*nat*) *oalist-tc*
⟨*proof*⟩

lemma *zero-PP-oalist* [*code*]: (*0*::('*a*::*linorder*, '*b*::*zero*) *pp*) = *PP-oalist OAlist-tc-empty*
⟨*proof*⟩

lemma *plus-PP-oalist* [*code*]:
PP-oalist xs + *PP-oalist ys* = *PP-oalist* (*OAlist-tc-map2-val-neutr* (λ-. (+)) *xs ys*)
⟨*proof*⟩

lemma *minus-PP-oalist* [*code*]:
PP-oalist xs − *PP-oalist ys* = *PP-oalist* (*OAlist-tc-map2-val-rneutr* (λ-. (−)) *xs*

ys)
⟨*proof*⟩

lemma *equal-PP-oalist* [*code*]: *equal-class.equal* (*PP-oalist xs*) (*PP-oalist ys*) = (*xs* = *ys*)
⟨*proof*⟩

lemma *lcs-PP-oalist* [*code*]:
lcs (*PP-oalist xs*) (*PP-oalist ys*) = *PP-oalist* (*OAlist-tc-map2-val-neutr* (λ-. *max*) *xs ys*)
for *xs ys* :: ('*a*::*linorder*, '*b*::*add-linorder-min*) *oalist-tc*
⟨*proof*⟩

lemma *deg-pp-PP-oalist* [*code*]: *deg-pp* (*PP-oalist xs*) = *sum-list* (*map snd* (*list-of-oalist-tc xs*))
⟨*proof*⟩

lemma *single-PP-oalist* [*code*]: *single-pp x e* = *PP-oalist* (*oalist-tc-of-list* [(*x*, *e*)])
⟨*proof*⟩

definition *adds-pp-add-linorder* :: ('*b*, '*a*::*add-linorder*) *pp* ⇒ - ⇒ *bool*
where [*code-abbrev*]: *adds-pp-add-linorder* = (*adds*)

lemma *adds-pp-PP-oalist* [*code*]:
adds-pp-add-linorder (*PP-oalist xs*) (*PP-oalist ys*) = *OAlist-tc-prod-ord* (λ-. *less-eq*) *xs ys*
for *xs ys*::('*a*::*linorder*, '*b*::*add-linorder-min*) *oalist-tc*
⟨*proof*⟩

15.1.1 Constructor

definition $sparse_0$ *xs* = *PP-oalist* (*oalist-tc-of-list xs*) — sparse representation

15.1.2 Computations

experiment begin

abbreviation *X* ≡ *0*::*nat*
abbreviation *Y* ≡ *1*::*nat*
abbreviation *Z* ≡ *2*::*nat*

value [*code*] $sparse_0$ [(*X*, *2*::*nat*), (*Z*, *7*)]

lemma
$sparse_0$ [(*X*, *2*::*nat*), (*Z*, *7*)] − $sparse_0$ [(*X*, *2*), (*Z*, *2*)] = $sparse_0$ [(*Z*, *5*)]
⟨*proof*⟩

lemma
lcs ($sparse_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *7*)]) ($sparse_0$ [(*Y*, *3*), (*Z*, *2*)]) = $sparse_0$ [(*X*, *2*), (*Y*, *3*), (*Z*, *7*)]

⟨*proof*⟩

lemma
(*sparse*$_0$ [(*X*, *2*::*nat*), (*Z*, *1*)]) *adds* (*sparse*$_0$ [(*X*, *3*), (*Y*, *2*), (*Z*, *1*)])
⟨*proof*⟩

lemma
lookup-pp (*sparse*$_0$ [(*X*, *2*::*nat*), (*Z*, *3*)]) *X* = *2*
⟨*proof*⟩

lemma
deg-pp (*sparse*$_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *3*), (*X*, *1*)]) = *6*
⟨*proof*⟩

lemma
lex-comp (*sparse*$_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *3*)]) (*sparse*$_0$ [(*X*, *4*)]) = *Lt*
⟨*proof*⟩

lemma
lex-comp (*sparse*$_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *3*)], *3*::*nat*) (*sparse*$_0$ [(*X*, *4*)], *2*) = *Lt*
⟨*proof*⟩

lemma
lex-pp (*sparse*$_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *3*)]) (*sparse*$_0$ [(*X*, *4*)])
⟨*proof*⟩

lemma
lex-pp (*sparse*$_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *3*)]) (*sparse*$_0$ [(*X*, *4*)])
⟨*proof*⟩

lemma
¬ *dlex-pp* (*sparse*$_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *3*)]) (*sparse*$_0$ [(*X*, *4*)])
⟨*proof*⟩

lemma
dlex-pp (*sparse*$_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *2*)]) (*sparse*$_0$ [(*X*, *5*)])
⟨*proof*⟩

lemma
¬ *drlex-pp* (*sparse*$_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *2*)]) (*sparse*$_0$ [(*X*, *5*)])
⟨*proof*⟩

end

15.2 *MP-oalist*

lift-definition *MP-oalist* :: ('*a*::*nat-term*, '*b*::*zero*) *oalist-ntm* ⇒ '*a* ⇒$_0$ '*b*
is *OAlist-lookup-ntm*

⟨*proof*⟩

lemmas [*simp*, *code*] = *MP-oalist.rep-eq*

code-datatype *MP-oalist*

lemma *keys-MP-oalist* [*code*]: *keys* (*MP-oalist xs*) = *set* (*map fst* (*fst* (*list-of-oalist-ntm xs*)))
⟨*proof*⟩

lemma *MP-oalist-empty* [*simp*]: *MP-oalist* (*OAlist-empty-ntm ko*) = *0*
⟨*proof*⟩

lemma *zero-MP-oalist* [*code*]: (*0*::('*a*::{*linorder*,*nat-term*} ⇒$_0$ '*b*::*zero*)) = *MP-oalist* (*OAlist-empty-ntm nat-term-order-of-le*)
⟨*proof*⟩

definition *is-zero* :: ('*a* ⇒$_0$ '*b*::*zero*) ⇒ *bool*
where [*code-abbrev*]: *is-zero p* ⟷ (*p* = *0*)

lemma *is-zero-MP-oalist* [*code*]: *is-zero* (*MP-oalist xs*) = *List.null* (*fst* (*list-of-oalist-ntm xs*))
⟨*proof*⟩

lemma *plus-MP-oalist* [*code*]: *MP-oalist xs* + *MP-oalist ys* = *MP-oalist* (*OAlist-map2-val-neutr-ntm* (λ-. (+)) *xs ys*)
⟨*proof*⟩

lemma *minus-MP-oalist* [*code*]: *MP-oalist xs* − *MP-oalist ys* = *MP-oalist* (*OAlist-map2-val-rneutr-ntm* (λ-. (−)) *xs ys*)
⟨*proof*⟩

lemma *uminus-MP-oalist* [*code*]: − *MP-oalist xs* = *MP-oalist* (*OAlist-map-val-ntm* (λ-. *uminus*) *xs*)
⟨*proof*⟩

lemma *equal-MP-oalist* [*code*]: *equal-class.equal* (*MP-oalist xs*) (*MP-oalist ys*) = (*OAlist-eq-ntm xs ys*)
⟨*proof*⟩

lemma *map-MP-oalist* [*code*]: *Poly-Mapping.map f* (*MP-oalist xs*) = *MP-oalist* (*OAlist-map-val-ntm* (λ-. *f*) *xs*)
⟨*proof*⟩

lemma *range-MP-oalist* [*code*]: *Poly-Mapping.range* (*MP-oalist xs*) = *set* (*map snd* (*fst* (*list-of-oalist-ntm xs*)))
⟨*proof*⟩

lemma *if-poly-mapping-eq-iff*:

(*if x = y then a else b*) = (*if* (∀ *i*∈*keys x* ∪ *keys y*. *lookup x i* = *lookup y i*) *then a else b*)
⟨*proof*⟩

lemma *keys-add-eq*: *keys* (*a* + *b*) = *keys a* ∪ *keys b* − {*x* ∈ *keys a* ∩ *keys b*. *lookup a x* + *lookup b x* = *0*}
⟨*proof*⟩

locale *gd-nat-term* =
gd-term pair-of-term term-of-pair
λ*s t*. *le-of-nat-term-order cmp-term* (*term-of-pair* (*s*, *the-min*)) (*term-of-pair* (*t*, *the-min*))
λ*s t*. *lt-of-nat-term-order cmp-term* (*term-of-pair* (*s*, *the-min*)) (*term-of-pair* (*t*, *the-min*))
le-of-nat-term-order cmp-term
lt-of-nat-term-order cmp-term
for *pair-of-term*::'*t*::*nat-term* ⇒ ('*a*::{*nat-term*,*graded-dickson-powerprod*} × '*k*::{*countable*,*the-min*,*wellorder*})
and *term-of-pair*::('*a* × '*k*) ⇒ '*t*
and *cmp-term* +
assumes *splus-eq-splus*: *t* ⊕ *u* = *nat-term-class.splus* (*term-of-pair* (*t*, *the-min*)) *u*
begin

definition *shift-map-keys* :: '*a* ⇒ ('*b* ⇒ '*b*) ⇒ ('*t*, '*b*) *oalist-ntm* ⇒ ('*t*, '*b*::*semiring-0*) *oalist-ntm*
where *shift-map-keys t f xs* = *OAlist-ntm* (*map-raw* (λ*kv*. (*t* ⊕ *fst kv*, *f* (*snd kv*))) (*list-of-oalist-ntm xs*))

lemma *list-of-oalist-shift-keys*:
list-of-oalist-ntm (*shift-map-keys t f xs*) = (*map-raw* (λ*kv*. (*t* ⊕ *fst kv*, *f* (*snd kv*))) (*list-of-oalist-ntm xs*))
⟨*proof*⟩

lemma *lookup-shift-map-keys-plus*:
lookup (*MP-oalist* (*shift-map-keys t* ((∗) *c*) *xs*)) (*t* ⊕ *u*) = *c* ∗ *lookup* (*MP-oalist xs*) *u* (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *keys-shift-map-keys-subset*:
keys (*MP-oalist* (*shift-map-keys t* ((∗) *c*) *xs*)) ⊆ ((⊕) *t*) ' *keys* (*MP-oalist xs*) (**is** *?l* ⊆ *?r*)
⟨*proof*⟩

lemma *monom-mult-MP-oalist* [*code*]:
monom-mult c t (*MP-oalist xs*) =
MP-oalist (*if c* = *0 then OAlist-empty-ntm* (*snd* (*list-of-oalist-ntm xs*)) *else shift-map-keys t* ((∗) *c*) *xs*)
⟨*proof*⟩

lemma *mult-scalar-MP-oalist* [*code*]:
(*MP-oalist xs*) ⊙ (*MP-oalist ys*) =
(*if is-zero* (*MP-oalist xs*) *then*
MP-oalist (*OAlist-empty-ntm* (*snd* (*list-of-oalist-ntm ys*)))
else
let ct = *OAlist-hd-ntm xs in*
monom-mult (*snd ct*) (*fst ct*) (*MP-oalist ys*) + (*MP-oalist* (*OAlist-tl-ntm xs*)) ⊙ (*MP-oalist ys*))
⟨*proof*⟩

end

15.2.1 Special case of addition: adding monomials

definition *plus-monomial-less* :: ('*a* ⇒$_0$ '*b*) ⇒ '*b* ⇒ '*a* ⇒ ('*a* ⇒$_0$ '*b*::*monoid-add*)
where *plus-monomial-less p c u* = *p* + *monomial c u*

plus-monomial-less is useful when adding a monomial to a polynomial, where the term of the monomial is known to be smaller than all terms in the polynomial, because it can be implemented more efficiently than general addition.

lemma *plus-monomial-less-MP-oalist* [*code*]:
plus-monomial-less (*MP-oalist xs*) *c u* = *MP-oalist* (*OAlist-update-by-fun-gr-ntm u* (λ*c0*. *c0* + *c*) *xs*)
⟨*proof*⟩

plus-monomial-less is computed by *OAlist-update-by-fun-gr-ntm*, because greater terms come *before* smaller ones in *oalist-ntm*.

15.2.2 Constructors

definition *distr*$_0$ *ko xs* = *MP-oalist* (*oalist-of-list-ntm* (*xs*, *ko*)) — sparse representation

definition V_0 :: '*a* ⇒ ('*a*, *nat*) *pp* ⇒$_0$ '*b*::{*one*,*zero*} **where**
V_0 *n* ≡ *monomial 1* (*single-pp n 1*)

definition C_0 :: '*b* ⇒ ('*a*, *nat*) *pp* ⇒$_0$ '*b*::*zero* **where** C_0 *c* ≡ *monomial c 0*

lemma C_0-*one*: C_0 *1* = *1*
⟨*proof*⟩

lemma C_0-*numeral*: C_0 (*numeral x*) = *numeral x*
⟨*proof*⟩

lemma C_0-*minus*: C_0 (− *x*) = − C_0 *x*
⟨*proof*⟩

lemma C_0-*zero*: C_0 *0* = *0*

⟨*proof*⟩

lemma V_0*-power*: $V_0\ v\ \hat{}\ n = monomial\ 1\ (single\text{-}pp\ v\ n)$
⟨*proof*⟩

lemma *single-MP-oalist* [*code*]: *Poly-Mapping.single k v* = $distr_0$ *nat-term-order-of-le* [(*k*, *v*)]
⟨*proof*⟩

lemma *one-MP-oalist* [*code*]: *1* = $distr_0$ *nat-term-order-of-le* [(*0*, *1*)]
⟨*proof*⟩

lemma *except-MP-oalist* [*code*]: *except* (*MP-oalist xs*) *S* = *MP-oalist* (*OAlist-filter-ntm* (λ*kv*. *fst kv* ∉ *S*) *xs*)
⟨*proof*⟩

15.2.3 Changing the Internal Order

definition *change-ord* :: *'a*::*nat-term-compare nat-term-order* ⇒ (*'a* ⇒$_0$ *'b*) ⇒ (*'a* ⇒$_0$ *'b*)
where *change-ord to* = (λ*x*. *x*)

lemma *change-ord-MP-oalist* [*code*]: *change-ord to* (*MP-oalist xs*) = *MP-oalist* (*OAlist-reorder-ntm to xs*)
⟨*proof*⟩

15.2.4 Ordered Power-Products

lemma *foldl-assoc*:
assumes ⋀*x y z*. *f* (*f x y*) *z* = *f x* (*f y z*)
shows *foldl f* (*f a b*) *xs* = *f a* (*foldl f b xs*)
⟨*proof*⟩

context *gd-nat-term*
begin

definition *ord-pp* :: *'a* ⇒ *'a* ⇒ *bool*
where *ord-pp s t* = *le-of-nat-term-order cmp-term* (*term-of-pair* (*s*, *the-min*)) (*term-of-pair* (*t*, *the-min*))

definition *ord-pp-strict* :: *'a* ⇒ *'a* ⇒ *bool*
where *ord-pp-strict s t* = *lt-of-nat-term-order cmp-term* (*term-of-pair* (*s*, *the-min*)) (*term-of-pair* (*t*, *the-min*))

lemma *lt-MP-oalist* [*code*]:
lt (*MP-oalist xs*) = (*if is-zero* (*MP-oalist xs*) *then min-term else fst* (*OAlist-min-key-val-ntm cmp-term xs*))
⟨*proof*⟩

lemma *lc-MP-oalist* [*code*]:

lc (*MP-oalist xs*) = (*if is-zero* (*MP-oalist xs*) *then 0 else snd* (*OAlist-min-key-val-ntm cmp-term xs*))
⟨*proof*⟩

lemma *tail-MP-oalist* [*code*]: *tail* (*MP-oalist xs*) = *MP-oalist* (*OAlist-except-min-ntm cmp-term xs*)
⟨*proof*⟩

definition *comp-opt-p* :: ('*t* ⇒$_0$ '*c*::*zero*, '*t* ⇒$_0$ '*c*) *comp-opt*
where *comp-opt-p p q* =
(*if p* = *q then Some Eq else if ord-strict-p p q then Some Lt else if ord-strict-p q p then Some Gt else None*)

lemma *comp-opt-p-MP-oalist* [*code*]:
comp-opt-p (*MP-oalist xs*) (*MP-oalist ys*) =
OAlist-lex-ord-ntm cmp-term (λ- *x y*. *if x* = *y then Some Eq else if x* = *0 then Some Lt else if y* = *0 then Some Gt else None*) *xs ys*
⟨*proof*⟩

lemma *compute-ord-p* [*code*]: *ord-p p q* = (*let aux* = *comp-opt-p p q in aux* = *Some Lt* ∨ *aux* = *Some Eq*)
⟨*proof*⟩

lemma *compute-ord-p-strict* [*code*]: *ord-strict-p p q* = (*comp-opt-p p q* = *Some Lt*)
⟨*proof*⟩

lemma *keys-to-list-MP-oalist* [*code*]: *keys-to-list* (*MP-oalist xs*) = *OAlist-sorted-domain-ntm cmp-term xs*
⟨*proof*⟩

end

lifting-update *poly-mapping.lifting*
lifting-forget *poly-mapping.lifting*

15.3 Interpretations

lemma *term-powerprod-gd-term*:
fixes *pair-of-term* :: '*t*::*nat-term* ⇒ ('*a*::{*graded-dickson-powerprod*,*nat-pp-compare*} × '*k*::{*the-min*,*wellorder*})
assumes *term-powerprod pair-of-term term-of-pair*
and ⋀*v*. *fst* (*rep-nat-term v*) = *rep-nat-pp* (*fst* (*pair-of-term v*))
and ⋀*t*. *snd* (*rep-nat-term* (*term-of-pair* (*t*, *the-min*))) = *0*
and ⋀*v w*. *snd* (*pair-of-term v*) ≤ *snd* (*pair-of-term w*) ⟹ *snd* (*rep-nat-term v*) ≤ *snd* (*rep-nat-term w*)
and ⋀*s t k*. *term-of-pair* (*s* + *t*, *k*) = *splus* (*term-of-pair* (*s*, *k*)) (*term-of-pair* (*t*, *k*))
and ⋀*t v*. *term-powerprod.splus pair-of-term term-of-pair t v* = *splus* (*term-of-pair*

(*t*, *the-min*)) *v*
shows *gd-term pair-of-term term-of-pair*
(λ*s t*. *le-of-nat-term-order cmp-term* (*term-of-pair* (*s*, *the-min*)) (*term-of-pair* (*t*, *the-min*)))
(λ*s t*. *lt-of-nat-term-order cmp-term* (*term-of-pair* (*s*, *the-min*)) (*term-of-pair* (*t*, *the-min*)))
(*le-of-nat-term-order cmp-term*)
(*lt-of-nat-term-order cmp-term*)
⟨*proof*⟩

lemma *gd-term-to-pair-unit*:
gd-term (*to-pair-unit*::*'a*::{*nat-term-compare*,*nat-pp-term*,*graded-dickson-powerprod*} ⇒ -) *fst*
(λ*s t*. *le-of-nat-term-order cmp-term* (*fst* (*s*, *the-min*)) (*fst* (*t*, *the-min*)))
(λ*s t*. *lt-of-nat-term-order cmp-term* (*fst* (*s*, *the-min*)) (*fst* (*t*, *the-min*)))
(*le-of-nat-term-order cmp-term*)
(*lt-of-nat-term-order cmp-term*)
⟨*proof*⟩

corollary *gd-nat-term-to-pair-unit*:
gd-nat-term (*to-pair-unit*::*'a*::{*nat-term-compare*,*nat-pp-term*,*graded-dickson-powerprod*} ⇒ -) *fst cmp-term*
⟨*proof*⟩

lemma *gd-term-id*:
gd-term (λ*x*::(*'a*::{*nat-term-compare*,*nat-pp-compare*,*nat-pp-term*,*graded-dickson-powerprod*} × *'b*::{*nat*,*the-min*}). *x*) (λ*x*. *x*)
(λ*s t*. *le-of-nat-term-order cmp-term* (*s*, *the-min*) (*t*, *the-min*))
(λ*s t*. *lt-of-nat-term-order cmp-term* (*s*, *the-min*) (*t*, *the-min*))
(*le-of-nat-term-order cmp-term*)
(*lt-of-nat-term-order cmp-term*)
⟨*proof*⟩

corollary *gd-nat-term-id*: *gd-nat-term* (λ*x*. *x*) (λ*x*. *x*) *cmp-term*
for *cmp-term* :: (*'a*::{*nat-term-compare*,*nat-pp-compare*,*nat-pp-term*,*graded-dickson-powerprod*} × *'c*::{*nat*,*the-min*}) *nat-term-order*
⟨*proof*⟩

15.4 Computations

type-synonym *'a mpoly-tc* = (*nat*, *nat*) *pp* ⇒$_0$ *'a*

global-interpretation *punit0*: *gd-nat-term to-pair-unit*::*'a*::{*nat-term-compare*,*nat-pp-term*,*graded-dickson-p*
⇒ - *fst cmp-term*
rewrites *punit.adds-term* = (*adds*)
and *punit.pp-of-term* = (λ*x*. *x*)
and *punit.component-of-term* = (λ-. ())
for *cmp-term*
defines *monom-mult-punit* = *punit.monom-mult*

and *mult-scalar-punit* = *punit.mult-scalar*
and *shift-map-keys-punit* = *punit0.shift-map-keys*
and *ord-pp-punit* = *punit0.ord-pp*
and *ord-pp-strict-punit* = *punit0.ord-pp-strict*
and *min-term-punit* = *punit0.min-term*
and *lt-punit* = *punit0.lt*
and *lc-punit* = *punit0.lc*
and *tail-punit* = *punit0.tail*
and *comp-opt-p-punit* = *punit0.comp-opt-p*
and *ord-p-punit* = *punit0.ord-p*
and *ord-strict-p-punit* = *punit0.ord-strict-p*
and *keys-to-list-punit* = *punit0.keys-to-list*
⟨*proof*⟩

lemma *shift-map-keys-punit-MP-oalist* [*code abstract*]:
list-of-oalist-ntm (*shift-map-keys-punit t f xs*) = *map-raw* (λ(*k*, *v*). (*t* + *k*, *f v*))
(*list-of-oalist-ntm xs*)
⟨*proof*⟩

lemmas [*code*] = *punit0.mult-scalar-MP-oalist*[*unfolded mult-scalar-punit-def punit-mult-scalar*]
punit0.punit-min-term

lemma *ord-pp-punit-alt* [*code-unfold*]: *ord-pp-punit* = *le-of-nat-term-order*
⟨*proof*⟩

lemma *ord-pp-strict-punit-alt* [*code-unfold*]: *ord-pp-strict-punit* = *lt-of-nat-term-order*
⟨*proof*⟩

lemma *gd-powerprod-ord-pp-punit*: *gd-powerprod* (*ord-pp-punit cmp-term*) (*ord-pp-strict-punit*
cmp-term)
⟨*proof*⟩

locale *trivariate*$_0$*-rat*
begin

abbreviation *X*::*rat mpoly-tc* **where** *X* ≡ V_0 (*0*::*nat*)
abbreviation *Y*::*rat mpoly-tc* **where** *Y* ≡ V_0 (*1*::*nat*)
abbreviation *Z*::*rat mpoly-tc* **where** *Z* ≡ V_0 (*2*::*nat*)

end

experiment begin interpretation *trivariate*$_0$*-rat* ⟨*proof*⟩

value [*code*] *X* ^ *2*

value [*code*] $X^2 * Z + 2 * Y \hat{} 3 * Z^2$

value [*code*] *distr*$_0$ *DRLEX* [(*sparse*$_0$ [(*0*::*nat*, *3*::*nat*)], *1*::*rat*)] = *distr*$_0$ *DRLEX*
[(*sparse*$_0$ [(*0*, *3*)], *1*)]

lemma
ord-strict-p-punit DRLEX $(X^2 * Z + 2 * Y \hat{} 3 * Z^2)$ $(X^2 * Z^2 + 2 * Y \hat{} 3 * Z^2)$
⟨*proof*⟩

lemma
tail-punit DLEX $(X^2 * Z + 2 * Y \hat{} 3 * Z^2) = X^2 * Z$
⟨*proof*⟩

value [*code*] *min-term-punit*::(*nat*, *nat*) *pp*

value [*code*] *is-zero* (*distr*$_0$ *DRLEX* [(*sparse*$_0$ [(*0*::*nat*, *3*::*nat*)], *1*::*rat*)])

value [*code*] *lt-punit DRLEX* (*distr*$_0$ *DRLEX* [(*sparse*$_0$ [(*0*::*nat*, *3*::*nat*)], *1*::*rat*)])

lemma
lt-punit DRLEX $(X^2 * Z + 2 * Y \hat{} 3 * Z^2) =$ *sparse*$_0$ $[(1, 3), (2, 2)]$
⟨*proof*⟩

lemma
lt-punit DRLEX $(X + Y + Z) =$ *sparse*$_0$ $[(2, 1)]$
⟨*proof*⟩

lemma
keys $(X^2 * Z \hat{} 3 + 2 * Y \hat{} 3 * Z^2) =$
$\{$*sparse*$_0$ $[(0, 2), (2, 3)]$, *sparse*$_0$ $[(1, 3), (2, 2)]\}$
⟨*proof*⟩

lemma
$- 1 * X^2 * Z \hat{} 7 + - 2 * Y \hat{} 3 * Z^2 = - X^2 * Z \hat{} 7 + - 2 * Y \hat{} 3 * Z^2$
⟨*proof*⟩

lemma
$X^2 * Z \hat{} 7 + 2 * Y \hat{} 3 * Z^2 + X^2 * Z \hat{} 4 + - 2 * Y \hat{} 3 * Z^2 = X^2 * Z \hat{} 7 + X^2 * Z \hat{} 4$
⟨*proof*⟩

lemma
$X^2 * Z \hat{} 7 + 2 * Y \hat{} 3 * Z^2 - X^2 * Z \hat{} 4 + - 2 * Y \hat{} 3 * Z^2 =$
$X^2 * Z \hat{} 7 - X^2 * Z \hat{} 4$
⟨*proof*⟩

lemma
lookup $(X^2 * Z \hat{} 7 + 2 * Y \hat{} 3 * Z^2 + 2)$ (*sparse*$_0$ $[(0, 2), (2, 7)]) = 1$
⟨*proof*⟩

lemma
$X^2 * Z \hat{} 7 + 2 * Y \hat{} 3 * Z^2 \neq$

$X^2 * Z \hat{} 4 + - 2 * Y \hat{} 3 * Z^2$
⟨*proof*⟩

lemma
$0 * X\hat{}2 * Z\hat{}7 + 0 * Y\hat{}3 * Z^2 = 0$
⟨*proof*⟩

lemma
monom-mult-punit 3 (*sparse*$_0$ [(*1*, *2*::*nat*)]) ($X^2 * Z + 2 * Y \hat{} 3 * Z^2$) =
$3 * Y^2 * Z * X^2 + 6 * Y \hat{} 5 * Z^2$
⟨*proof*⟩

lemma
monomial (*−4*) (*sparse*$_0$ [(*0*, *2*::*nat*)]) = $- 4 * X^2$
⟨*proof*⟩

lemma *monomial* (*0*::*rat*) (*sparse*$_0$ [(*0*::*nat*, *2*::*nat*)]) = *0*
⟨*proof*⟩

lemma
($X^2 * Z + 2 * Y \hat{} 3 * Z^2$) * ($X^2 * Z \hat{} 3 + - 2 * Y \hat{} 3 * Z^2$) =
$X \hat{} 4 * Z \hat{} 4 + - 2 * X^2 * Z \hat{} 3 * Y \hat{} 3 +$
$- 4 * Y \hat{} 6 * Z \hat{} 4 + 2 * Y \hat{} 3 * Z \hat{} 5 * X^2$
⟨*proof*⟩

end

15.5 Code setup for type MPoly

postprocessing from Var_0, $Const_0$ to *Var*, *Const*.

lemmas [*code-post*] =
plus-mpoly.abs-eq[*symmetric*]
times-mpoly.abs-eq[*symmetric*]
one-mpoly-def[*symmetric*]
Var.abs-eq[*symmetric*]
Const.abs-eq[*symmetric*]

instantiation *mpoly*::({*equal*, *zero*})*equal* **begin**

lift-definition *equal-mpoly*:: *'a mpoly* ⇒ *'a mpoly* ⇒ *bool* **is** *HOL.equal* ⟨*proof*⟩

instance ⟨*proof*⟩

end

end

16 Quasi-Poly-Mapping Power-Products

theory *Quasi-PM-Power-Products*
imports *MPoly-Type-Class-Ordered*
begin

In this theory we introduce a subclass of *graded-dickson-powerprod* that approximates polynomial mappings even closer. We need this class for signature-based Gröbner basis algorithms.

definition (**in** *monoid-add*) *hom-grading-fun* :: $('a \Rightarrow nat) \Rightarrow (nat \Rightarrow 'a \Rightarrow 'a) \Rightarrow bool$
where $hom\text{-}grading\text{-}fun\ d\ f \longleftrightarrow (\forall n.\ (\forall s\ t.\ f\ n\ (s + t) = f\ n\ s + f\ n\ t) \wedge (\forall t.\ d\ (f\ n\ t) \leq n \wedge (d\ t \leq n \longrightarrow f\ n\ t = t)))$

definition (**in** *monoid-add*) *hom-grading* :: $('a \Rightarrow nat) \Rightarrow bool$
where $hom\text{-}grading\ d \longleftrightarrow (\exists f.\ hom\text{-}grading\text{-}fun\ d\ f)$

definition (**in** *monoid-add*) *decr-grading* :: $('a \Rightarrow nat) \Rightarrow nat \Rightarrow 'a \Rightarrow 'a$
where $decr\text{-}grading\ d = (SOME\ f.\ hom\text{-}grading\text{-}fun\ d\ f)$

lemma *decr-grading*:
assumes *hom-grading d*
shows *hom-grading-fun d* (*decr-grading d*)
⟨*proof*⟩

lemma *decr-grading-plus*:
$hom\text{-}grading\ d \Longrightarrow decr\text{-}grading\ d\ n\ (s + t) = decr\text{-}grading\ d\ n\ s + decr\text{-}grading\ d\ n\ t$
⟨*proof*⟩

lemma *decr-grading-zero*:
assumes *hom-grading d*
shows $decr\text{-}grading\ d\ n\ 0 = (0::'a::cancel\text{-}comm\text{-}monoid\text{-}add)$
⟨*proof*⟩

lemma *decr-grading-le*: $hom\text{-}grading\ d \Longrightarrow d\ (decr\text{-}grading\ d\ n\ t) \leq n$
⟨*proof*⟩

lemma *decr-grading-idI*: $hom\text{-}grading\ d \Longrightarrow d\ t \leq n \Longrightarrow decr\text{-}grading\ d\ n\ t = t$
⟨*proof*⟩

class *quasi-pm-powerprod* = *ulcs-powerprod* +
assumes *ex-hgrad*: $\exists d::'a \Rightarrow nat.\ dickson\text{-}grading\ d \wedge hom\text{-}grading\ d$
begin

subclass *graded-dickson-powerprod*
⟨*proof*⟩

end

lemma *hom-grading-varnum*:
hom-grading ((*varnum X*)::('*x*::*countable* ⇒$_0$ '*b*::*add-wellorder*) ⇒ *nat*)
⟨*proof*⟩

instance *poly-mapping* :: (*countable*, *add-wellorder*) *quasi-pm-powerprod*
⟨*proof*⟩

context *term-powerprod*
begin

definition *decr-grading-term* :: ('*a* ⇒ *nat*) ⇒ *nat* ⇒ '*t* ⇒ '*t*
where *decr-grading-term d n v* = *term-of-pair* (*decr-grading d n* (*pp-of-term v*), *component-of-term v*)

definition *decr-grading-p* :: ('*a* ⇒ *nat*) ⇒ *nat* ⇒ ('*t* ⇒$_0$ '*b*) ⇒ ('*t* ⇒$_0$ '*b*::*comm-monoid-add*)
where *decr-grading-p d n p* = (∑ *v*∈*keys p*. *monomial* (*lookup p v*) (*decr-grading-term d n v*))

lemma *decr-grading-term-splus*:
hom-grading d ⟹ *decr-grading-term d n* (*t* ⊕ *v*) = *decr-grading d n t* ⊕ *decr-grading-term d n v*
⟨*proof*⟩

lemma *decr-grading-term-le*: *hom-grading d* ⟹ *d* (*pp-of-term* (*decr-grading-term d n v*)) ≤ *n*
⟨*proof*⟩

lemma *decr-grading-term-idI*: *hom-grading d* ⟹ *d* (*pp-of-term v*) ≤ *n* ⟹ *decr-grading-term d n v* = *v*
⟨*proof*⟩

lemma *punit-decr-grading-term*: *punit.decr-grading-term* = *decr-grading*
⟨*proof*⟩

lemma *decr-grading-p-zero*: *decr-grading-p d n 0* = *0*
⟨*proof*⟩

lemma *decr-grading-p-monomial*: *decr-grading-p d n* (*monomial c v*) = *monomial c* (*decr-grading-term d n v*)
⟨*proof*⟩

lemma *decr-grading-p-plus*:
decr-grading-p d n (*p* + *q*) = (*decr-grading-p d n p*) + (*decr-grading-p d n q*)
⟨*proof*⟩

corollary *decr-grading-p-sum*: *decr-grading-p d n* (*sum f A*) = (∑ *a*∈*A*. *decr-grading-p d n* (*f a*))
⟨*proof*⟩

lemma *decr-grading-p-monom-mult*:
assumes *hom-grading d*
shows *decr-grading-p d n* (*monom-mult c t p*) = *monom-mult c* (*decr-grading d n t*) (*decr-grading-p d n p*)
⟨*proof*⟩

lemma *decr-grading-p-mult-scalar*:
assumes *hom-grading d*
shows *decr-grading-p d n* (*p* ⊙ *q*) = *punit.decr-grading-p d n p* ⊙ *decr-grading-p d n q*
⟨*proof*⟩

lemma *decr-grading-p-keys-subset*: *keys* (*decr-grading-p d n p*) ⊆ *decr-grading-term d n* ‘ *keys p*
⟨*proof*⟩

lemma *decr-grading-p-idI'*:
assumes *hom-grading d* **and** ⋀*v*. *v* ∈ *keys p* ⟹ *d* (*pp-of-term v*) ≤ *n*
shows *decr-grading-p d n p* = *p*
⟨*proof*⟩

end

context *gd-term*
begin

lemma *decr-grading-p-idI*:
assumes *hom-grading d* **and** *p* ∈ *dgrad-p-set d m*
shows *decr-grading-p d m p* = *p*
⟨*proof*⟩

lemma *decr-grading-p-dgrad-p-setI*:
assumes *hom-grading d*
shows *decr-grading-p d m p* ∈ *dgrad-p-set d m*
⟨*proof*⟩

lemma (**in** *gd-term*) *in-pmdlE-dgrad-p-set*:
assumes *hom-grading d* **and** *B* ⊆ *dgrad-p-set d m* **and** *p* ∈ *dgrad-p-set d m* **and** *p* ∈ *pmdl B*
obtains *A q* **where** *finite A* **and** *A* ⊆ *B* **and** ⋀*b*. *q b* ∈ *punit.dgrad-p-set d m* **and** *p* = (∑ *b*∈*A*. *q b* ⊙ *b*)
⟨*proof*⟩

end

end

17 Multivariate Polynomials with Power-Products Represented by Polynomial Mappings

theory *MPoly-PM*
imports *Quasi-PM-Power-Products*
begin

Many notions introduced in this theory for type $('x \Rightarrow_0 'a) \Rightarrow_0 'b$ closely resemble those introduced in *Polynomials.MPoly-Type* for type $'a$ *mpoly*.

lemma *monomial-single-power*:
$(monomial\ c\ (Poly\text{-}Mapping.single\ x\ k))\ \hat{}\ n = monomial\ (c\ \hat{}\ n)\ (Poly\text{-}Mapping.single\ x\ (k * n))$
⟨*proof*⟩

lemma *monomial-power-map-scale*: $(monomial\ c\ t)\ \hat{}\ n = monomial\ (c\ \hat{}\ n)\ (n \cdot t)$
⟨*proof*⟩

lemma *times-canc-left*:
assumes $h * p = h * q$ **and** $h \neq (0::('x::linorder \Rightarrow_0 nat) \Rightarrow_0 'a::ring\text{-}no\text{-}zero\text{-}divisors)$
shows $p = q$
⟨*proof*⟩

lemma *times-canc-right*:
assumes $p * h = q * h$ **and** $h \neq (0::('x::linorder \Rightarrow_0 nat) \Rightarrow_0 'a::ring\text{-}no\text{-}zero\text{-}divisors)$
shows $p = q$
⟨*proof*⟩

17.1 Degree

lemma *plus-minus-assoc-pm-nat-1*: $s + t - u = (s - (u - t)) + (t - (u::- \Rightarrow_0 nat))$
⟨*proof*⟩

lemma *plus-minus-assoc-pm-nat-2*:
$s + (t - u) = (s + (except\ (u - t)\ (-\ keys\ s))) + t - (u::- \Rightarrow_0 nat)$
⟨*proof*⟩

lemma *deg-pm-sum*: $deg\text{-}pm\ (sum\ t\ A) = (\sum a{\in}A.\ deg\text{-}pm\ (t\ a))$
⟨*proof*⟩

lemma *deg-pm-mono*: $s\ adds\ t \Longrightarrow deg\text{-}pm\ s \leq deg\text{-}pm\ (t::- \Rightarrow_0 -::add\text{-}linorder\text{-}min)$
⟨*proof*⟩

lemma *adds-deg-pm-antisym*: $s\ adds\ t \Longrightarrow deg\text{-}pm\ t \leq deg\text{-}pm\ (s::- \Rightarrow_0 -::add\text{-}linorder\text{-}min) \Longrightarrow s = t$
⟨*proof*⟩

lemma *deg-pm-minus*:

assumes *s adds* (*t*::- ⇒$_0$ -::*comm-monoid-add*)
shows *deg-pm* (*t* − *s*) = *deg-pm t* − *deg-pm s*
⟨*proof*⟩

lemma *adds-group* [*simp*]: *s adds* (*t*::'*a* ⇒$_0$ '*b*::*ab-group-add*)
⟨*proof*⟩

lemmas *deg-pm-minus-group* = *deg-pm-minus*[*OF adds-group*]

lemma *deg-pm-minus-le*: *deg-pm* (*t* − *s*) ≤ *deg-pm* (*t*::- ⇒$_0$ *nat*)
⟨*proof*⟩

lemma *minus-id-iff*: *t* − *s* = *t* ⟷ *keys t* ∩ *keys* (*s*::- ⇒$_0$ *nat*) = {}
⟨*proof*⟩

lemma *deg-pm-minus-id-iff*: *deg-pm* (*t* − *s*) = *deg-pm t* ⟷ *keys t* ∩ *keys* (*s*::- ⇒$_0$ *nat*) = {}
⟨*proof*⟩

definition *poly-deg* :: (('*x* ⇒$_0$ '*a*::*add-linorder*) ⇒$_0$ '*b*::*zero*) ⇒ '*a* **where**
poly-deg p = (*if keys p* = {} *then 0 else Max* (*deg-pm* ' *keys p*))

definition *maxdeg* :: (('*x* ⇒$_0$ '*a*::*add-linorder*) ⇒$_0$ '*b*::*zero*) *set* ⇒ '*a* **where**
maxdeg A = *Max* (*poly-deg* ' *A*)

definition *mindeg* :: (('*x* ⇒$_0$ '*a*::*add-linorder*) ⇒$_0$ '*b*::*zero*) *set* ⇒ '*a* **where**
mindeg A = *Min* (*poly-deg* ' *A*)

lemma *poly-deg-monomial*: *poly-deg* (*monomial c t*) = (*if c* = *0 then 0 else deg-pm t*)
⟨*proof*⟩

lemma *poly-deg-monomial-zero* [*simp*]: *poly-deg* (*monomial c 0*) = *0*
⟨*proof*⟩

lemma *poly-deg-zero* [*simp*]: *poly-deg 0* = *0*
⟨*proof*⟩

lemma *poly-deg-one* [*simp*]: *poly-deg 1* = *0*
⟨*proof*⟩

lemma *poly-degE*:
assumes *p* ≠ *0*
obtains *t* **where** *t* ∈ *keys p* **and** *poly-deg p* = *deg-pm t*
⟨*proof*⟩

lemma *poly-deg-max-keys*: *t* ∈ *keys p* ⟹ *deg-pm t* ≤ *poly-deg p*
⟨*proof*⟩

lemma *poly-deg-leI*: $(\bigwedge t.\ t \in keys\ p \Longrightarrow deg\text{-}pm\ t \leq (d::'a::add\text{-}linorder\text{-}min)) \Longrightarrow poly\text{-}deg\ p \leq d$
⟨*proof*⟩

lemma *poly-deg-lessI*:
$p \neq 0 \Longrightarrow (\bigwedge t.\ t \in keys\ p \Longrightarrow deg\text{-}pm\ t < (d::'a::add\text{-}linorder\text{-}min)) \Longrightarrow poly\text{-}deg\ p < d$
⟨*proof*⟩

lemma *poly-deg-zero-imp-monomial*:
assumes $poly\text{-}deg\ p = (0::'a::add\text{-}linorder\text{-}min)$
shows $monomial\ (lookup\ p\ 0)\ 0 = p$
⟨*proof*⟩

lemma *poly-deg-plus-le*:
$poly\text{-}deg\ (p + q) \leq max\ (poly\text{-}deg\ p)\ (poly\text{-}deg\ (q::(- \Rightarrow_0 'a::add\text{-}linorder\text{-}min) \Rightarrow_0 -))$
⟨*proof*⟩

lemma *poly-deg-uminus* [*simp*]: $poly\text{-}deg\ (-p) = poly\text{-}deg\ p$
⟨*proof*⟩

lemma *poly-deg-minus-le*:
$poly\text{-}deg\ (p - q) \leq max\ (poly\text{-}deg\ p)\ (poly\text{-}deg\ (q::(- \Rightarrow_0 'a::add\text{-}linorder\text{-}min) \Rightarrow_0 -))$
⟨*proof*⟩

lemma *poly-deg-times-le*:
$poly\text{-}deg\ (p * q) \leq poly\text{-}deg\ p + poly\text{-}deg\ (q::(- \Rightarrow_0 'a::add\text{-}linorder\text{-}min) \Rightarrow_0 -)$
⟨*proof*⟩

lemma *poly-deg-times*:
assumes $p \neq 0$ **and** $q \neq (0::('x::linorder \Rightarrow_0 'a::add\text{-}linorder\text{-}min) \Rightarrow_0 'b::semiring\text{-}no\text{-}zero\text{-}divisors)$
shows $poly\text{-}deg\ (p * q) = poly\text{-}deg\ p + poly\text{-}deg\ q$
⟨*proof*⟩

corollary *poly-deg-monom-mult-le*:
$poly\text{-}deg\ (punit.monom\text{-}mult\ c\ (t::- \Rightarrow_0 'a::add\text{-}linorder\text{-}min)\ p) \leq deg\text{-}pm\ t + poly\text{-}deg\ p$
⟨*proof*⟩

lemma *poly-deg-monom-mult*:
assumes $c \neq 0$ **and** $p \neq (0::(- \Rightarrow_0 'a::add\text{-}linorder\text{-}min) \Rightarrow_0 'b::semiring\text{-}no\text{-}zero\text{-}divisors)$
shows $poly\text{-}deg\ (punit.monom\text{-}mult\ c\ t\ p) = deg\text{-}pm\ t + poly\text{-}deg\ p$
⟨*proof*⟩

lemma *poly-deg-map-scale*:
$poly\text{-}deg\ (c \cdot p) = (if\ c = (0::-::semiring\text{-}no\text{-}zero\text{-}divisors)\ then\ 0\ else\ poly\text{-}deg\ p)$
⟨*proof*⟩

lemma *poly-deg-sum-le*: $((poly\text{-}deg\ (sum\ f\ A))::'a::add\text{-}linorder\text{-}min) \leq Max\ (poly\text{-}deg\ `\ f\ `\ A)$
⟨*proof*⟩

lemma *poly-deg-prod-le*: $((poly\text{-}deg\ (prod\ f\ A))::'a::add\text{-}linorder\text{-}min) \leq (\sum a{\in}A.\ poly\text{-}deg\ (f\ a))$
⟨*proof*⟩

lemma *maxdeg-max*:
assumes *finite A* **and** $p \in A$
shows *poly-deg p* $\leq$ *maxdeg A*
⟨*proof*⟩

lemma *mindeg-min*:
assumes *finite A* **and** $p \in A$
shows *mindeg A* $\leq$ *poly-deg p*
⟨*proof*⟩

17.2 Indeterminates

definition *indets* :: $(('x \Rightarrow_0 nat) \Rightarrow_0 'b::zero) \Rightarrow 'x\ set$
where *indets p* = $\bigcup$ (*keys* ` *keys p*)

definition *PPs* :: $'x\ set \Rightarrow ('x \Rightarrow_0 nat)\ set$ (.[(-)])
where *PPs X* = $\{t.\ keys\ t \subseteq X\}$

definition *Polys* :: $'x\ set \Rightarrow (('x \Rightarrow_0 nat) \Rightarrow_0 'b::zero)\ set$ (*P*[(-)])
where *Polys X* = $\{p.\ keys\ p \subseteq .[X]\}$

17.2.1 *indets*

lemma *in-indetsI*:
assumes $x \in keys\ t$ **and** $t \in keys\ p$
shows $x \in indets\ p$
⟨*proof*⟩

lemma *in-indetsE*:
assumes $x \in indets\ p$
obtains t **where** $t \in keys\ p$ **and** $x \in keys\ t$
⟨*proof*⟩

lemma *keys-subset-indets*: $t \in keys\ p \Longrightarrow keys\ t \subseteq indets\ p$
⟨*proof*⟩

lemma *indets-empty-imp-monomial*:
assumes *indets p* = {}
shows *monomial* (*lookup p 0*) *0* = *p*
⟨*proof*⟩

lemma *finite-indets*: *finite* (*indets p*)
⟨*proof*⟩

lemma *indets-zero* [*simp*]: *indets 0* = {}
⟨*proof*⟩

lemma *indets-one* [*simp*]: *indets 1* = {}
⟨*proof*⟩

lemma *indets-monomial-single-subset*: *indets* (*monomial c* (*Poly-Mapping.single v k*)) ⊆ {*v*}
⟨*proof*⟩

lemma *indets-monomial-single*:
assumes *c* ≠ *0* **and** *k* ≠ *0*
shows *indets* (*monomial c* (*Poly-Mapping.single v k*)) = {*v*}
⟨*proof*⟩

lemma *indets-monomial*:
assumes *c* ≠ *0*
shows *indets* (*monomial c t*) = *keys t*
⟨*proof*⟩

lemma *indets-monomial-subset*: *indets* (*monomial c t*) ⊆ *keys t*
⟨*proof*⟩

lemma *indets-monomial-zero* [*simp*]: *indets* (*monomial c 0*) = {}
⟨*proof*⟩

lemma *indets-plus-subset*: *indets* (*p* + *q*) ⊆ *indets p* ∪ *indets q*
⟨*proof*⟩

lemma *indets-uminus* [*simp*]: *indets* (−*p*) = *indets p*
⟨*proof*⟩

lemma *indets-minus-subset*: *indets* (*p* − *q*) ⊆ *indets p* ∪ *indets q*
⟨*proof*⟩

lemma *indets-times-subset*: *indets* (*p* ∗ *q*) ⊆ *indets p* ∪ *indets* (*q*::(- ⇒$_0$ -::*cancel-comm-monoid-add*) ⇒$_0$ -)
⟨*proof*⟩

corollary *indets-monom-mult-subset*: *indets* (*punit.monom-mult c t p*) ⊆ *keys t* ∪ *indets p*
⟨*proof*⟩

lemma *indets-monom-mult*:
assumes *c* ≠ *0* **and** *p* ≠ (*0*::('*x* ⇒$_0$ *nat*) ⇒$_0$ '*b*::*semiring-no-zero-divisors*)
shows *indets* (*punit.monom-mult c t p*) = *keys t* ∪ *indets p*

⟨*proof*⟩

lemma *indets-sum-subset*: *indets* (*sum f A*) ⊆ (⋃ *a*∈*A*. *indets* (*f a*))
⟨*proof*⟩

lemma *indets-prod-subset*:
indets (*prod* (*f*::- ⇒ ((- ⇒$_0$ -::*cancel-comm-monoid-add*) ⇒$_0$ -)) *A*) ⊆ (⋃ *a*∈*A*.
indets (*f a*))
⟨*proof*⟩

lemma *indets-power-subset*: *indets* (*p* ^ *n*) ⊆ *indets* (*p*::('*x* ⇒$_0$ *nat*) ⇒$_0$ '*b*::*comm-semiring-1*)
⟨*proof*⟩

lemma *indets-empty-iff-poly-deg-zero*: *indets p* = {} ⟷ *poly-deg p* = *0*
⟨*proof*⟩

17.2.2 *PPs*

lemma *PPsI*: *keys t* ⊆ *X* ⟹ *t* ∈ .[*X*]
⟨*proof*⟩

lemma *PPsD*: *t* ∈ .[*X*] ⟹ *keys t* ⊆ *X*
⟨*proof*⟩

lemma *PPs-empty* [*simp*]: .[{}] = {*0*}
⟨*proof*⟩

lemma *PPs-UNIV* [*simp*]: .[*UNIV*] = *UNIV*
⟨*proof*⟩

lemma *PPs-singleton*: .[{*x*}] = *range* (*Poly-Mapping.single x*)
⟨*proof*⟩

lemma *zero-in-PPs*: *0* ∈ .[*X*]
⟨*proof*⟩

lemma *PPs-mono*: *X* ⊆ *Y* ⟹ .[*X*] ⊆ .[*Y*]
⟨*proof*⟩

lemma *PPs-closed-single*:
assumes *x* ∈ *X*
shows *Poly-Mapping.single x e* ∈ .[*X*]
⟨*proof*⟩

lemma *PPs-closed-plus*:
assumes *s* ∈ .[*X*] **and** *t* ∈ .[*X*]
shows *s* + *t* ∈ .[*X*]
⟨*proof*⟩

lemma *PPs-closed-minus*:
assumes $s \in .[X]$
shows $s - t \in .[X]$
⟨*proof*⟩

lemma *PPs-closed-adds*:
assumes $s \in .[X]$ **and** t *adds* s
shows $t \in .[X]$
⟨*proof*⟩

lemma *PPs-closed-gcs*:
assumes $s \in .[X]$
shows *gcs* s $t \in .[X]$
⟨*proof*⟩

lemma *PPs-closed-lcs*:
assumes $s \in .[X]$ **and** $t \in .[X]$
shows *lcs* s $t \in .[X]$
⟨*proof*⟩

lemma *PPs-closed-except'*: $t \in .[X] \Longrightarrow$ *except* t $Y \in .[X - Y]$
⟨*proof*⟩

lemma *PPs-closed-except*: $t \in .[X] \Longrightarrow$ *except* t $Y \in .[X]$
⟨*proof*⟩

lemma *PPs-UnI*:
assumes $tx \in .[X]$ **and** $ty \in .[Y]$ **and** $t = tx + ty$
shows $t \in .[X \cup Y]$
⟨*proof*⟩

lemma *PPs-UnE*:
assumes $t \in .[X \cup Y]$
obtains tx ty **where** $tx \in .[X]$ **and** $ty \in .[Y]$ **and** $t = tx + ty$
⟨*proof*⟩

lemma *PPs-Un*: $.[X \cup Y] = (\bigcup t{\in}.[X].\ (+)\ t\ `\ .[Y])$ (**is** *?A* = *?B*)
⟨*proof*⟩

corollary *PPs-insert*: $.[insert\ x\ X] = (\bigcup e.\ (+)\ (Poly\text{-}Mapping.single\ x\ e)\ `\ .[X])$
⟨*proof*⟩

corollary *PPs-insertI*:
assumes $tx \in .[X]$ **and** $t = Poly\text{-}Mapping.single\ x\ e + tx$
shows $t \in .[insert\ x\ X]$
⟨*proof*⟩

corollary *PPs-insertE*:
assumes $t \in .[insert\ x\ X]$

obtains *e tx* **where** $tx \in .[X]$ **and** $t = $ *Poly-Mapping.single x e* $+$ *tx*
⟨*proof*⟩

lemma *PPs-Int*: $.[X \cap Y] = .[X] \cap .[Y]$
⟨*proof*⟩

lemma *PPs-INT*: $.[\bigcap X] = \bigcap (PPs\ `\ X)$
⟨*proof*⟩

17.2.3 *Polys*

lemma *Polys-alt*: $P[X] = \{p.\ indets\ p \subseteq X\}$
⟨*proof*⟩

lemma *PolysI*: $keys\ p \subseteq .[X] \Longrightarrow p \in P[X]$
⟨*proof*⟩

lemma *PolysI-alt*: $indets\ p \subseteq X \Longrightarrow p \in P[X]$
⟨*proof*⟩

lemma *PolysD*:
assumes $p \in P[X]$
shows $keys\ p \subseteq .[X]$ **and** $indets\ p \subseteq X$
⟨*proof*⟩

lemma *Polys-empty*: $P[\{\}] = ((range\ (Poly\text{-}Mapping.single\ 0))::(('x \Rightarrow_0 nat) \Rightarrow_0 'b::zero)\ set)$
⟨*proof*⟩

lemma *Polys-UNIV* [*simp*]: $P[UNIV] = UNIV$
⟨*proof*⟩

lemma *zero-in-Polys*: $0 \in P[X]$
⟨*proof*⟩

lemma *one-in-Polys*: $1 \in P[X]$
⟨*proof*⟩

lemma *Polys-mono*: $X \subseteq Y \Longrightarrow P[X] \subseteq P[Y]$
⟨*proof*⟩

lemma *Polys-closed-monomial*: $t \in .[X] \Longrightarrow monomial\ c\ t \in P[X]$
⟨*proof*⟩

lemma *Polys-closed-plus*: $p \in P[X] \Longrightarrow q \in P[X] \Longrightarrow p + q \in P[X]$
⟨*proof*⟩

lemma *Polys-closed-uminus*: $p \in P[X] \Longrightarrow -p \in P[X]$
⟨*proof*⟩

lemma *Polys-closed-minus*: $p \in P[X] \Longrightarrow q \in P[X] \Longrightarrow p - q \in P[X]$
⟨*proof*⟩

lemma *Polys-closed-monom-mult*: $t \in .[X] \Longrightarrow p \in P[X] \Longrightarrow$ *punit.monom-mult* $c\ t\ p \in P[X]$
⟨*proof*⟩

corollary *Polys-closed-map-scale*: $p \in P[X] \Longrightarrow (c::-::semiring\text{-}0) \cdot p \in P[X]$
⟨*proof*⟩

lemma *Polys-closed-times*: $p \in P[X] \Longrightarrow q \in P[X] \Longrightarrow p * q \in P[X]$
⟨*proof*⟩

lemma *Polys-closed-power*: $p \in P[X] \Longrightarrow p\ \hat{}\ m \in P[X]$
⟨*proof*⟩

lemma *Polys-closed-sum*: $(\bigwedge a.\ a \in A \Longrightarrow f\ a \in P[X]) \Longrightarrow sum\ f\ A \in P[X]$
⟨*proof*⟩

lemma *Polys-closed-prod*: $(\bigwedge a.\ a \in A \Longrightarrow f\ a \in P[X]) \Longrightarrow prod\ f\ A \in P[X]$
⟨*proof*⟩

lemma *Polys-closed-sum-list*: $(\bigwedge x.\ x \in set\ xs \Longrightarrow x \in P[X]) \Longrightarrow sum\text{-}list\ xs \in P[X]$
⟨*proof*⟩

lemma *Polys-closed-except*: $p \in P[X] \Longrightarrow except\ p\ T \in P[X]$
⟨*proof*⟩

lemma *times-in-PolysD*:
assumes $p * q \in P[X]$ **and** $p \in P[X]$ **and** $p \neq (0::('x::linorder \Rightarrow_0 nat) \Rightarrow_0 'a::semiring\text{-}no\text{-}zero\text{-}divisors)$
shows $q \in P[X]$
⟨*proof*⟩

lemma *poly-mapping-plus-induct-Polys* [*consumes 1*, *case-names 0 plus*]:
assumes $p \in P[X]$ **and** $P\ 0$
and $\bigwedge p\ c\ t.\ t \in .[X] \Longrightarrow p \in P[X] \Longrightarrow c \neq 0 \Longrightarrow t \notin keys\ p \Longrightarrow P\ p \Longrightarrow P\ (monomial\ c\ t + p)$
shows $P\ p$
⟨*proof*⟩

lemma *Polys-Int*: $P[X \cap Y] = P[X] \cap P[Y]$
⟨*proof*⟩

lemma *Polys-INT*: $P[\bigcap X] = \bigcap (Polys\ `\ X)$
⟨*proof*⟩

17.3 Substitution Homomorphism

The substitution homomorphism defined here is more general than *insertion*, since it replaces indeterminates by *polynomials* rather than coefficients, and therefore constructs new polynomials.

definition *subst-pp* :: ('*x* ⇒ (('*y* ⇒$_0$ *nat*) ⇒$_0$ '*a*)) ⇒ ('*x* ⇒$_0$ *nat*) ⇒ (('*y* ⇒$_0$ *nat*) ⇒$_0$ '*a*::*comm-semiring-1*)
  where *subst-pp f t* = (∏ *x*∈*keys t*. (*f x*) ^ (*lookup t x*))

definition *poly-subst* :: ('*x* ⇒ (('*y* ⇒$_0$ *nat*) ⇒$_0$ '*a*)) ⇒ (('*x* ⇒$_0$ *nat*) ⇒$_0$ '*a*) ⇒ (('*y* ⇒$_0$ *nat*) ⇒$_0$ '*a*::*comm-semiring-1*)
  where *poly-subst f p* = (∑ *t*∈*keys p*. *punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*))

lemma *subst-pp-alt*: *subst-pp f t* = (∏ *x*. (*f x*) ^ (*lookup t x*))
⟨*proof*⟩

lemma *subst-pp-zero* [*simp*]: *subst-pp f 0* = *1*
  ⟨*proof*⟩

lemma *subst-pp-trivial-not-zero*:
  assumes *t* ≠ *0*
  shows *subst-pp* (λ-. *0*) *t* = (*0*::(- ⇒$_0$ '*b*::*comm-semiring-1*))
  ⟨*proof*⟩

lemma *subst-pp-single*: *subst-pp f* (*Poly-Mapping.single x e*) = (*f x*) ^ *e*
  ⟨*proof*⟩

corollary *subst-pp-trivial*: *subst-pp* (λ-. *0*) *t* = (*if t* = *0 then 1 else 0*)
  ⟨*proof*⟩

lemma *power-lookup-not-one-subset-keys*: {*x*. *f x* ^ (*lookup t x*) ≠ *1*} ⊆ *keys t*
⟨*proof*⟩

corollary *finite-power-lookup-not-one*: *finite* {*x*. *f x* ^ (*lookup t x*) ≠ *1*}
  ⟨*proof*⟩

lemma *subst-pp-plus*: *subst-pp f* (*s* + *t*) = *subst-pp f s* ∗ *subst-pp f t*
  ⟨*proof*⟩

lemma *subst-pp-id*:
  assumes ⋀*x*. *x* ∈ *keys t* ⟹ *f x* = *monomial 1* (*Poly-Mapping.single x 1*)
  shows *subst-pp f t* = *monomial 1 t*
⟨*proof*⟩

lemma *in-indets-subst-ppE*:
  assumes *x* ∈ *indets* (*subst-pp f t*)
  obtains *y* **where** *y* ∈ *keys t* **and** *x* ∈ *indets* (*f y*)
⟨*proof*⟩

lemma *subst-pp-by-monomials*:
assumes $\bigwedge y.\ y \in keys\ t \Longrightarrow f\ y = monomial\ (c\ y)\ (s\ y)$
shows *subst-pp f t* = *monomial* $(\prod y{\in}keys\ t.\ (c\ y)\ \hat{}\ lookup\ t\ y)$ $(\sum y{\in}keys\ t.$ *lookup t y* · *s y*)
⟨*proof*⟩

lemma *poly-deg-subst-pp-eq-zeroI*:
assumes $\bigwedge x.\ x \in keys\ t \Longrightarrow poly\text{-}deg\ (f\ x) = 0$
shows *poly-deg* (*subst-pp f t*) = *0*
⟨*proof*⟩

lemma *poly-deg-subst-pp-le*:
assumes $\bigwedge x.\ x \in keys\ t \Longrightarrow poly\text{-}deg\ (f\ x) \leq 1$
shows *poly-deg* (*subst-pp f t*) ≤ *deg-pm t*
⟨*proof*⟩

lemma *poly-subst-alt*: *poly-subst f p* = $(\sum t.$ *punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*))
⟨*proof*⟩

lemma *poly-subst-trivial* [*simp*]: *poly-subst* (λ-. *0*) *p* = *monomial* (*lookup p 0*) *0*
⟨*proof*⟩

lemma *poly-subst-zero* [*simp*]: *poly-subst f 0* = *0*
⟨*proof*⟩

lemma *monom-mult-lookup-not-zero-subset-keys*:
$\{t.$ *punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*) ≠ *0*$\} \subseteq$ *keys p*
⟨*proof*⟩

corollary *finite-monom-mult-lookup-not-zero*:
finite $\{t.$ *punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*) ≠ *0*$\}$
⟨*proof*⟩

lemma *poly-subst-plus*: *poly-subst f* (*p* + *q*) = *poly-subst f p* + *poly-subst f q*
⟨*proof*⟩

lemma *poly-subst-uminus*: *poly-subst f* (−*p*) = − *poly-subst f* (*p*::(*'x* ⇒$_0$ *nat*) ⇒$_0$ *'b*::*comm-ring-1*)
⟨*proof*⟩

lemma *poly-subst-minus*:
poly-subst f (*p* − *q*) = *poly-subst f p* − *poly-subst f* (*q*::(*'x* ⇒$_0$ *nat*) ⇒$_0$ *'b*::*comm-ring-1*)
⟨*proof*⟩

lemma *poly-subst-monomial*: *poly-subst f* (*monomial c t*) = *punit.monom-mult c 0* (*subst-pp f t*)
⟨*proof*⟩

corollary *poly-subst-one* [*simp*]: *poly-subst f 1 = 1*
⟨*proof*⟩

lemma *poly-subst-times*: *poly-subst f (p * q) = poly-subst f p * poly-subst f q*
⟨*proof*⟩

corollary *poly-subst-monom-mult*:
*poly-subst f (punit.monom-mult c t p) = punit.monom-mult c 0 (subst-pp f t * poly-subst f p)*
⟨*proof*⟩

corollary *poly-subst-monom-mult'*:
*poly-subst f (punit.monom-mult c t p) = (punit.monom-mult c 0 (subst-pp f t)) * poly-subst f p*
⟨*proof*⟩

lemma *poly-subst-sum*: *poly-subst f (sum p A) = (∑ a∈A. poly-subst f (p a))*
⟨*proof*⟩

lemma *poly-subst-prod*: *poly-subst f (prod p A) = (∏ a∈A. poly-subst f (p a))*
⟨*proof*⟩

lemma *poly-subst-power*: *poly-subst f (p ^ n) = (poly-subst f p) ^ n*
⟨*proof*⟩

lemma *poly-subst-subst-pp*: *poly-subst f (subst-pp g t) = subst-pp (λx. poly-subst f (g x)) t*
⟨*proof*⟩

lemma *poly-subst-poly-subst*: *poly-subst f (poly-subst g p) = poly-subst (λx. poly-subst f (g x)) p*
⟨*proof*⟩

lemma *poly-subst-id*:
assumes *⋀x. x ∈ indets p ⟹ f x = monomial 1 (Poly-Mapping.single x 1)*
shows *poly-subst f p = p*
⟨*proof*⟩

lemma *in-keys-poly-substE*:
assumes *t ∈ keys (poly-subst f p)*
obtains *s* **where** *s ∈ keys p* **and** *t ∈ keys (subst-pp f s)*
⟨*proof*⟩

lemma *in-indets-poly-substE*:
assumes *x ∈ indets (poly-subst f p)*
obtains *y* **where** *y ∈ indets p* **and** *x ∈ indets (f y)*
⟨*proof*⟩

lemma *poly-deg-poly-subst-eq-zeroI*:
assumes $\bigwedge x.\ x \in \mathit{indets}\ p \Longrightarrow \mathit{poly\text{-}deg}\ (f\ x) = 0$
shows *poly-deg* (*poly-subst* (f::- ⇒ (('y ⇒$_0$ -) ⇒$_0$ -)) (p::('x ⇒$_0$ -) ⇒$_0$ 'b::*comm-semiring-1*)) = *0*
⟨*proof*⟩

lemma *poly-deg-poly-subst-le*:
assumes $\bigwedge x.\ x \in \mathit{indets}\ p \Longrightarrow \mathit{poly\text{-}deg}\ (f\ x) \leq 1$
shows *poly-deg* (*poly-subst* (f::- ⇒ (('y ⇒$_0$ -) ⇒$_0$ -)) (p::('x ⇒$_0$ *nat*) ⇒$_0$ 'b::*comm-semiring-1*)) ≤ *poly-deg p*
⟨*proof*⟩

lemma *subst-pp-cong*: $s = t \Longrightarrow (\bigwedge x.\ x \in \mathit{keys}\ t \Longrightarrow f\ x = g\ x) \Longrightarrow \mathit{subst\text{-}pp}\ f\ s = \mathit{subst\text{-}pp}\ g\ t$
⟨*proof*⟩

lemma *poly-subst-cong*:
assumes $p = q$ **and** $\bigwedge x.\ x \in \mathit{indets}\ q \Longrightarrow f\ x = g\ x$
shows *poly-subst f p* = *poly-subst g q*
⟨*proof*⟩

lemma *Polys-homomorphismE*:
obtains h **where** $\bigwedge p\ q.\ h\ (p + q) = h\ p + h\ q$ **and** $\bigwedge p\ q.\ h\ (p * q) = h\ p * h\ q$
and $\bigwedge p$::('x ⇒$_0$ *nat*) ⇒$_0$ 'a::*comm-ring-1*. $h\ (h\ p) = h\ p$ **and** *range h* = $P[X]$
⟨*proof*⟩

lemma *in-idealE-Polys-finite*:
assumes *finite B* **and** $B \subseteq P[X]$ **and** $p \in P[X]$ **and** (p::('x ⇒$_0$ *nat*) ⇒$_0$ 'a::*comm-ring-1*) ∈ *ideal B*
obtains q **where** $\bigwedge b.\ q\ b \in P[X]$ **and** $p = (\sum b{\in}B.\ q\ b * b)$
⟨*proof*⟩

corollary *in-idealE-Polys*:
assumes $B \subseteq P[X]$ **and** $p \in P[X]$ **and** $p \in \mathit{ideal}\ B$
obtains $A\ q$ **where** *finite A* **and** $A \subseteq B$ **and** $\bigwedge b.\ q\ b \in P[X]$ **and** $p = (\sum b{\in}A.\ q\ b * b)$
⟨*proof*⟩

lemma *ideal-induct-Polys* [*consumes 3*, *case-names 0 plus*]:
assumes $F \subseteq P[X]$ **and** $p \in P[X]$ **and** $p \in \mathit{ideal}\ F$
assumes $P\ 0$ **and** $\bigwedge c\ q\ h.\ c \in P[X] \Longrightarrow q \in F \Longrightarrow P\ h \Longrightarrow h \in P[X] \Longrightarrow P\ (c * q + h)$
shows P (p::('x ⇒$_0$ *nat*) ⇒$_0$ 'a::*comm-ring-1*)
⟨*proof*⟩

lemma *image-poly-subst-ideal-subset*: *poly-subst g* ‘ *ideal F* ⊆ *ideal* (*poly-subst g* ‘ *F*)
⟨*proof*⟩

17.4 Evaluating Polynomials

lemma *lookup-times-zero*:
lookup (*p* ∗ *q*) *0* = *lookup p 0* ∗ *lookup q* (*0*::*'a*::{*comm-powerprod*,*ninv-comm-monoid-add*})
⟨*proof*⟩

corollary *lookup-prod-zero*:
lookup (*prod f I*) *0* = (∏ *i*∈*I*. *lookup* (*f i*) (*0*::-::{*comm-powerprod*,*ninv-comm-monoid-add*}))
⟨*proof*⟩

corollary *lookup-power-zero*:
lookup (*p* ^ *k*) *0* = *lookup p* (*0*::-::{*comm-powerprod*,*ninv-comm-monoid-add*}) ^
k
⟨*proof*⟩

definition *poly-eval* :: (*'x* ⇒ *'a*) ⇒ ((*'x* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ *'a*::*comm-semiring-1*
where *poly-eval a p* = *lookup* (*poly-subst* (λ*y*. *monomial* (*a y*) (*0*::*'x* ⇒$_0$ *nat*))
p) *0*

lemma *poly-eval-alt*: *poly-eval a p* = (∑ *t*∈*keys p*. *lookup p t* ∗ (∏ *x*∈*keys t*. *a x*
^ *lookup t x*))
⟨*proof*⟩

lemma *poly-eval-monomial*: *poly-eval a* (*monomial c t*) = *c* ∗ (∏ *x*∈*keys t*. *a x* ^
lookup t x)
⟨*proof*⟩

lemma *poly-eval-zero* [*simp*]: *poly-eval a 0* = *0*
⟨*proof*⟩

lemma *poly-eval-zero-left* [*simp*]: *poly-eval 0 p* = *lookup p 0*
⟨*proof*⟩

lemma *poly-eval-plus*: *poly-eval a* (*p* + *q*) = *poly-eval a p* + *poly-eval a q*
⟨*proof*⟩

lemma *poly-eval-uminus* [*simp*]: *poly-eval a* (− *p*) = − *poly-eval* (*a*::-::*comm-ring-1*)
p
⟨*proof*⟩

lemma *poly-eval-minus*: *poly-eval a* (*p* − *q*) = *poly-eval a p* − *poly-eval* (*a*::-::*comm-ring-1*)
q
⟨*proof*⟩

lemma *poly-eval-one* [*simp*]: *poly-eval a 1* = *1*
⟨*proof*⟩

lemma *poly-eval-times*: *poly-eval a* (*p* ∗ *q*) = *poly-eval a p* ∗ *poly-eval a q*
⟨*proof*⟩

lemma *poly-eval-power*: *poly-eval a* (*p* ^ *m*) = *poly-eval a p* ^ *m*
⟨*proof*⟩

lemma *poly-eval-sum*: *poly-eval a* (*sum f I*) = (∑ *i*∈*I*. *poly-eval a* (*f i*))
⟨*proof*⟩

lemma *poly-eval-prod*: *poly-eval a* (*prod f I*) = (∏ *i*∈*I*. *poly-eval a* (*f i*))
⟨*proof*⟩

lemma *poly-eval-cong*: *p* = *q* ⟹ (⋀*x*. *x* ∈ *indets q* ⟹ *a x* = *b x*) ⟹ *poly-eval a p* = *poly-eval b q*
⟨*proof*⟩

lemma *indets-poly-eval-subset*:
indets (*poly-eval a p*) ⊆ ⋃ (*indets* ‘ *a* ‘ *indets p*) ∪ ⋃ (*indets* ‘ *lookup p* ‘ *keys p*)
⟨*proof*⟩

lemma *image-poly-eval-ideal*: *poly-eval a* ‘ *ideal F* = *ideal* (*poly-eval a* ‘ *F*)
⟨*proof*⟩

17.5 Replacing Indeterminates

definition *map-indets* **where** *map-indets f* = *poly-subst* (λ*x*. *monomial 1* (*Poly-Mapping.single* (*f x*) *1*))

lemma
shows *map-indets-zero* [*simp*]: *map-indets f 0* = *0*
and *map-indets-one* [*simp*]: *map-indets f 1* = *1*
and *map-indets-uminus* [*simp*]: *map-indets f* (− *r*) = − *map-indets f* (*r*::- ⇒$_0$ -::*comm-ring-1*)
and *map-indets-plus*: *map-indets f* (*p* + *q*) = *map-indets f p* + *map-indets f q*
and *map-indets-minus*: *map-indets f* (*r* − *s*) = *map-indets f r* − *map-indets f s*
and *map-indets-times*: *map-indets f* (*p* ∗ *q*) = *map-indets f p* ∗ *map-indets f q*
and *map-indets-power* [*simp*]: *map-indets f* (*p* ^ *m*) = *map-indets f p* ^ *m*
and *map-indets-sum*: *map-indets f* (*sum g A*) = (∑ *a*∈*A*. *map-indets f* (*g a*))
and *map-indets-prod*: *map-indets f* (*prod g A*) = (∏ *a*∈*A*. *map-indets f* (*g a*))
⟨*proof*⟩

lemma *map-indets-monomial*:
map-indets f (*monomial c t*) = *monomial c* (∑ *x*∈*keys t*. *Poly-Mapping.single* (*f x*) (*lookup t x*))
⟨*proof*⟩

lemma *map-indets-id*: (⋀*x*. *x* ∈ *indets p* ⟹ *f x* = *x*) ⟹ *map-indets f p* = *p*
⟨*proof*⟩

lemma *map-indets-map-indets*: *map-indets f* (*map-indets g p*) = *map-indets* (*f* ∘

$g)$ p
⟨*proof*⟩

lemma *map-indets-cong*: $p = q \Longrightarrow (\bigwedge x.\ x \in indets\ q \Longrightarrow f\ x = g\ x) \Longrightarrow$ *map-indets* $f\ p =$ *map-indets* $g\ q$
⟨*proof*⟩

lemma *poly-subst-map-indets*: *poly-subst* f (*map-indets* $g\ p$) = *poly-subst* $(f \circ g)$ p
⟨*proof*⟩

lemma *poly-eval-map-indets*: *poly-eval* a (*map-indets* $g\ p$) = *poly-eval* $(a \circ g)\ p$
⟨*proof*⟩

lemma *map-indets-inverseE-Polys*:
assumes *inj-on* $f\ X$ **and** $p \in P[X]$
shows *map-indets* (*the-inv-into* $X\ f$) (*map-indets* $f\ p$) = p
⟨*proof*⟩

lemma *map-indets-inverseE*:
assumes *inj* f
obtains g **where** $g =$ *the-inv* f **and** $g \circ f = id$ **and** *map-indets* $g \circ$ *map-indets* $f = id$
⟨*proof*⟩

lemma *indets-map-indets-subset*: *indets* (*map-indets* f (p::- ⇒$_0$ 'a::*comm-semiring-1*)) ⊆ f ‘ *indets* p
⟨*proof*⟩

corollary *map-indets-in-Polys*: *map-indets* $f\ p \in P[f$ ‘ *indets* $p]$
⟨*proof*⟩

lemma *indets-map-indets*:
assumes *inj-on* f (*indets* p)
shows *indets* (*map-indets* $f\ p$) = f ‘ *indets* p
⟨*proof*⟩

lemma *image-map-indets-Polys*: *map-indets* f ‘ $P[X] = (P[f$ ‘ $X]$::(- ⇒$_0$ 'a::*comm-semiring-1*) *set*)
⟨*proof*⟩

corollary *range-map-indets*: *range* (*map-indets* f) = $P[$*range* $f]$
⟨*proof*⟩

lemma *in-keys-map-indetsE*:
assumes $t \in keys$ (*map-indets* f (p::- ⇒$_0$ 'a::*comm-semiring-1*))
obtains s **where** $s \in keys\ p$ **and** $t = (\sum x \in keys\ s.$ *Poly-Mapping.single* $(f\ x)$ (*lookup* $s\ x$))
⟨*proof*⟩

lemma *keys-map-indets-subset*:
keys (*map-indets f p*) ⊆ (λ*t*. ∑ *x*∈*keys t*. *Poly-Mapping.single* (*f x*) (*lookup t x*)) ‘ *keys p*
⟨*proof*⟩

lemma *keys-map-indets*:
assumes *inj-on f* (*indets p*)
shows *keys* (*map-indets f p*) = (λ*t*. ∑ *x*∈*keys t*. *Poly-Mapping.single* (*f x*) (*lookup t x*)) ‘ *keys p*
⟨*proof*⟩

lemma *poly-deg-map-indets-le*: *poly-deg* (*map-indets f p*) ≤ *poly-deg p*
⟨*proof*⟩

lemma *poly-deg-map-indets*:
assumes *inj-on f* (*indets p*)
shows *poly-deg* (*map-indets f p*) = *poly-deg p*
⟨*proof*⟩

lemma *map-indets-inj-on-PolysI*:
assumes *inj-on* (*f*::′*x* ⇒ ′*y*) *X*
shows *inj-on* ((*map-indets f*)::- ⇒ - ⇒$_0$ ′*a*::*comm-semiring-1*) *P*[*X*]
⟨*proof*⟩

lemma *map-indets-injI*:
assumes *inj f*
shows *inj* (*map-indets f*)
⟨*proof*⟩

lemma *image-map-indets-ideal*:
assumes *inj f*
shows *map-indets f* ‘ *ideal F* = *ideal* (*map-indets f* ‘ (*F*::(- ⇒$_0$ ′*a*::*comm-ring-1*) *set*)) ∩ *P*[*range f*]
⟨*proof*⟩

17.6 Homogeneity

definition *homogeneous* :: ((′*x* ⇒$_0$ *nat*) ⇒$_0$ ′*a*::*zero*) ⇒ *bool*
where *homogeneous p* ⟷ (∀ *s*∈*keys p*. ∀ *t*∈*keys p*. *deg-pm s* = *deg-pm t*)

definition *hom-component* :: ((′*x* ⇒$_0$ *nat*) ⇒$_0$ ′*a*) ⇒ *nat* ⇒ ((′*x* ⇒$_0$ *nat*) ⇒$_0$ ′*a*::*zero*)
where *hom-component p n* = *except p* {*t*. *deg-pm t* ≠ *n*}

definition *hom-components* :: ((′*x* ⇒$_0$ *nat*) ⇒$_0$ ′*a*) ⇒ ((′*x* ⇒$_0$ *nat*) ⇒$_0$ ′*a*::*zero*) *set*
where *hom-components p* = *hom-component p* ‘ *deg-pm* ‘ *keys p*

definition *homogeneous-set* :: $((('x \Rightarrow_0 nat) \Rightarrow_0 'a::zero)\ set \Rightarrow bool$
  where *homogeneous-set* $A \longleftrightarrow (\forall\, a{\in}A.\ \forall\, n.\ hom\text{-}component\ a\ n \in A)$

lemma *homogeneousI*: $(\bigwedge s\ t.\ s \in keys\ p \Longrightarrow t \in keys\ p \Longrightarrow deg\text{-}pm\ s = deg\text{-}pm\ t) \Longrightarrow homogeneous\ p$
  ⟨*proof*⟩

lemma *homogeneousD*: $homogeneous\ p \Longrightarrow s \in keys\ p \Longrightarrow t \in keys\ p \Longrightarrow deg\text{-}pm\ s = deg\text{-}pm\ t$
  ⟨*proof*⟩

lemma *homogeneousD-poly-deg*:
  assumes *homogeneous* p **and** $t \in keys\ p$
  shows $deg\text{-}pm\ t = poly\text{-}deg\ p$
⟨*proof*⟩

lemma *homogeneous-monomial* [*simp*]: $homogeneous\ (monomial\ c\ t)$
  ⟨*proof*⟩

corollary *homogeneous-zero* [*simp*]: *homogeneous 0* **and** *homogeneous-one* [*simp*]: *homogeneous 1*
  ⟨*proof*⟩

lemma *homogeneous-uminus-iff* [*simp*]: $homogeneous\ (-\ p) \longleftrightarrow homogeneous\ p$
  ⟨*proof*⟩

lemma *homogeneous-monom-mult*: $homogeneous\ p \Longrightarrow homogeneous\ (punit.monom\text{-}mult\ c\ t\ p)$
  ⟨*proof*⟩

lemma *homogeneous-monom-mult-rev*:
  assumes $c \neq (0::'a::semiring\text{-}no\text{-}zero\text{-}divisors)$ **and** $homogeneous\ (punit.monom\text{-}mult\ c\ t\ p)$
  shows *homogeneous* p
⟨*proof*⟩

lemma *homogeneous-times*:
  assumes *homogeneous* p **and** *homogeneous* q
  shows $homogeneous\ (p * q)$
⟨*proof*⟩

lemma *lookup-hom-component*: $lookup\ (hom\text{-}component\ p\ n) = (\lambda t.\ lookup\ p\ t\ when\ deg\text{-}pm\ t = n)$
  ⟨*proof*⟩

lemma *keys-hom-component*: $keys\ (hom\text{-}component\ p\ n) = \{t.\ t \in keys\ p \wedge deg\text{-}pm\ t = n\}$
  ⟨*proof*⟩

lemma *keys-hom-componentD*:
assumes $t \in$ *keys* (*hom-component* p n)
shows $t \in$ *keys* p **and** *deg-pm* $t = n$
⟨*proof*⟩

lemma *homogeneous-hom-component*: *homogeneous* (*hom-component* p n)
⟨*proof*⟩

lemma *hom-component-zero* [*simp*]: *hom-component* $0 = 0$
⟨*proof*⟩

lemma *hom-component-zero-iff*: *hom-component* p $n = 0 \longleftrightarrow (\forall t{\in}keys\ p.\ deg\text{-}pm\ t \neq n)$
⟨*proof*⟩

lemma *hom-component-uminus* [*simp*]: *hom-component* $(-p) = -$ *hom-component* p
⟨*proof*⟩

lemma *hom-component-plus*: *hom-component* $(p + q)$ $n =$ *hom-component* p $n +$ *hom-component* q n
⟨*proof*⟩

lemma *hom-component-minus*: *hom-component* $(p - q)$ $n =$ *hom-component* p n $-$ *hom-component* q n
⟨*proof*⟩

lemma *hom-component-monom-mult*:
punit.monom-mult c t (*hom-component* p n) = *hom-component* (*punit.monom-mult* c t p) (*deg-pm* $t + n$)
⟨*proof*⟩

lemma *hom-component-inject*:
assumes $t \in$ *keys* p **and** *hom-component* p (*deg-pm* t) = *hom-component* p n
shows *deg-pm* $t = n$
⟨*proof*⟩

lemma *hom-component-of-homogeneous*:
assumes *homogeneous* p
shows *hom-component* p n = (p *when* n = *poly-deg* p)
⟨*proof*⟩

lemma *hom-components-zero* [*simp*]: *hom-components* $0 = \{\}$
⟨*proof*⟩

lemma *hom-components-zero-iff* [*simp*]: *hom-components* $p = \{\} \longleftrightarrow p = 0$
⟨*proof*⟩

lemma *hom-components-uminus*: *hom-components* $(-p)$ = *uminus* ‘ *hom-components*

p
⟨*proof*⟩

lemma *hom-components-monom-mult*:
hom-components (*punit.monom-mult c t p*) = (*if c* = *0 then* {} *else punit.monom-mult c t* ‘ *hom-components p*)
for *c*::′*a*::*semiring-no-zero-divisors*
⟨*proof*⟩

lemma *hom-componentsI*: *q* = *hom-component p* (*deg-pm t*) ⟹ *t* ∈ *keys p* ⟹ *q* ∈ *hom-components p*
⟨*proof*⟩

lemma *hom-componentsE*:
assumes *q* ∈ *hom-components p*
obtains *t* **where** *t* ∈ *keys p* **and** *q* = *hom-component p* (*deg-pm t*)
⟨*proof*⟩

lemma *hom-components-of-homogeneous*:
assumes *homogeneous p*
shows *hom-components p* = (*if p* = *0 then* {} *else* {*p*})
⟨*proof*⟩

lemma *finite-hom-components*: *finite* (*hom-components p*)
⟨*proof*⟩

lemma *hom-components-homogeneous*: *q* ∈ *hom-components p* ⟹ *homogeneous q*
⟨*proof*⟩

lemma *hom-components-nonzero*: *q* ∈ *hom-components p* ⟹ *q* ≠ *0*
⟨*proof*⟩

lemma *deg-pm-hom-components*:
assumes *q1* ∈ *hom-components p* **and** *q2* ∈ *hom-components p* **and** *t1* ∈ *keys q1* **and** *t2* ∈ *keys q2*
shows *deg-pm t1* = *deg-pm t2* ⟷ *q1* = *q2*
⟨*proof*⟩

lemma *poly-deg-hom-components*:
assumes *q1* ∈ *hom-components p* **and** *q2* ∈ *hom-components p*
shows *poly-deg q1* = *poly-deg q2* ⟷ *q1* = *q2*
⟨*proof*⟩

lemma *hom-components-keys-disjoint*:
assumes *q1* ∈ *hom-components p* **and** *q2* ∈ *hom-components p* **and** *q1* ≠ *q2*
shows *keys q1* ∩ *keys q2* = {}
⟨*proof*⟩

lemma *Keys-hom-components*: *Keys* (*hom-components p*) = *keys p*
⟨*proof*⟩

lemma *lookup-hom-components*: *q* ∈ *hom-components p* ⟹ *t* ∈ *keys q* ⟹ *lookup q t* = *lookup p t*
⟨*proof*⟩

lemma *poly-deg-hom-components-le*:
assumes *q* ∈ *hom-components p*
shows *poly-deg q* ≤ *poly-deg p*
⟨*proof*⟩

lemma *sum-hom-components*: ∑ (*hom-components p*) = *p*
⟨*proof*⟩

lemma *homogeneous-setI*: (⋀*a n*. *a* ∈ *A* ⟹ *hom-component a n* ∈ *A*) ⟹ *homogeneous-set A*
⟨*proof*⟩

lemma *homogeneous-setD*: *homogeneous-set A* ⟹ *a* ∈ *A* ⟹ *hom-component a n* ∈ *A*
⟨*proof*⟩

lemma *homogeneous-set-Polys*: *homogeneous-set* (*P*[*X*]::(- ⇒$_0$ 'a::*zero*) *set*)
⟨*proof*⟩

lemma *homogeneous-set-IntI*: *homogeneous-set A* ⟹ *homogeneous-set B* ⟹ *homogeneous-set* (*A* ∩ *B*)
⟨*proof*⟩

lemma *homogeneous-setD-hom-components*:
assumes *homogeneous-set A* **and** *a* ∈ *A* **and** *b* ∈ *hom-components a*
shows *b* ∈ *A*
⟨*proof*⟩

lemma *zero-in-homogeneous-set*:
assumes *homogeneous-set A* **and** *A* ≠ {}
shows *0* ∈ *A*
⟨*proof*⟩

lemma *homogeneous-ideal*:
assumes ⋀*f*. *f* ∈ *F* ⟹ *homogeneous f* **and** *p* ∈ *ideal F*
shows *hom-component p n* ∈ *ideal F*
⟨*proof*⟩

corollary *homogeneous-set-homogeneous-ideal*:
(⋀*f*. *f* ∈ *F* ⟹ *homogeneous f*) ⟹ *homogeneous-set* (*ideal F*)
⟨*proof*⟩

corollary *homogeneous-ideal'*:
 assumes $\bigwedge f.\ f \in F \Longrightarrow$ *homogeneous f* **and** $p \in$ *ideal F* **and** $q \in$ *hom-components p*
 shows $q \in$ *ideal F*
 ⟨*proof*⟩

lemma *homogeneous-idealE-homogeneous*:
 assumes $\bigwedge f.\ f \in F \Longrightarrow$ *homogeneous f* **and** $p \in$ *ideal F* **and** *homogeneous p*
 obtains $F'\ q$ **where** *finite* F' **and** $F' \subseteq F$ **and** $p = (\sum f \in F'.\ q\ f * f)$ **and** $\bigwedge f.$ *homogeneous* $(q\ f)$
 and $\bigwedge f.\ f \in F' \Longrightarrow$ *poly-deg* $(q\ f * f) =$ *poly-deg p* **and** $\bigwedge f.\ f \notin F' \Longrightarrow q\ f = 0$
⟨*proof*⟩

corollary *homogeneous-idealE*:
 assumes $\bigwedge f.\ f \in F \Longrightarrow$ *homogeneous f* **and** $p \in$ *ideal F*
 obtains $F'\ q$ **where** *finite* F' **and** $F' \subseteq F$ **and** $p = (\sum f \in F'.\ q\ f * f)$
 and $\bigwedge f.$ *poly-deg* $(q\ f * f) \leq$ *poly-deg p* **and** $\bigwedge f.\ f \notin F' \Longrightarrow q\ f = 0$
⟨*proof*⟩

corollary *homogeneous-idealE-finite*:
 assumes *finite F* **and** $\bigwedge f.\ f \in F \Longrightarrow$ *homogeneous f* **and** $p \in$ *ideal F*
 obtains q **where** $p = (\sum f \in F.\ q\ f * f)$ **and** $\bigwedge f.$ *poly-deg* $(q\ f * f) \leq$ *poly-deg p*
 and $\bigwedge f.\ f \notin F \Longrightarrow q\ f = 0$
⟨*proof*⟩

17.6.1 Homogenization and Dehomogenization

definition *homogenize* :: $'x \Rightarrow (('x \Rightarrow_0 nat) \Rightarrow_0 'a) \Rightarrow (('x \Rightarrow_0 nat) \Rightarrow_0 'a$::*semiring-1*)
 where *homogenize x p* = $(\sum t \in$ *keys p*. *monomial* (*lookup p t*) (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm t*) + *t*))

definition *dehomo-subst* :: $'x \Rightarrow 'x \Rightarrow (('x \Rightarrow_0 nat) \Rightarrow_0 'a$::*zero-neq-one*)
 where *dehomo-subst x* = $(\lambda y.$ *if y* = *x then 1 else monomial 1* (*Poly-Mapping.single y 1*))

definition *dehomogenize* :: $'x \Rightarrow (('x \Rightarrow_0 nat) \Rightarrow_0 'a) \Rightarrow (('x \Rightarrow_0 nat) \Rightarrow_0 'a$::*comm-semiring-1*)
 where *dehomogenize x* = *poly-subst* (*dehomo-subst x*)

lemma *homogenize-zero* [*simp*]: *homogenize x 0* = *0*
 ⟨*proof*⟩

lemma *homogenize-uminus* [*simp*]: *homogenize x* (− *p*) = − *homogenize x* (*p*::-$\Rightarrow_0 'a$::*ring-1*)
 ⟨*proof*⟩

lemma *homogenize-monom-mult* [*simp*]:

homogenize x (punit.monom-mult c t p) = punit.monom-mult c t (homogenize x p)
for *c*::*'a*::{*semiring-1*,*semiring-no-zero-divisors-cancel*}
⟨*proof*⟩

lemma *homogenize-alt*:
homogenize x p = (∑ *q*∈*hom-components p*. *punit.monom-mult 1* (*Poly-Mapping.single x* (*poly-deg p* − *poly-deg q*)) *q*)
⟨*proof*⟩

lemma *keys-homogenizeE*:
assumes *t* ∈ *keys* (*homogenize x p*)
obtains *t'* **where** *t'* ∈ *keys p* **and** *t* = *Poly-Mapping.single x* (*poly-deg p* − *deg-pm t'*) + *t'*
⟨*proof*⟩

lemma *keys-homogenizeE-alt*:
assumes *t* ∈ *keys* (*homogenize x p*)
obtains *q t'* **where** *q* ∈ *hom-components p* **and** *t'* ∈ *keys q*
and *t* = *Poly-Mapping.single x* (*poly-deg p* − *poly-deg q*) + *t'*
⟨*proof*⟩

lemma *deg-pm-homogenize*:
assumes *t* ∈ *keys* (*homogenize x p*)
shows *deg-pm t* = *poly-deg p*
⟨*proof*⟩

corollary *homogeneous-homogenize*: *homogeneous* (*homogenize x p*)
⟨*proof*⟩

corollary *poly-deg-homogenize-le*: *poly-deg* (*homogenize x p*) ≤ *poly-deg p*
⟨*proof*⟩

lemma *homogenize-id-iff* [*simp*]: *homogenize x p* = *p* ⟷ *homogeneous p*
⟨*proof*⟩

lemma *homogenize-homogenize* [*simp*]: *homogenize x* (*homogenize x p*) = *homogenize x p*
⟨*proof*⟩

lemma *homogenize-monomial*: *homogenize x* (*monomial c t*) = *monomial c t*
⟨*proof*⟩

lemma *indets-homogenize-subset*: *indets* (*homogenize x p*) ⊆ *insert x* (*indets p*)
⟨*proof*⟩

lemma *homogenize-in-Polys*: *p* ∈ *P*[*X*] ⟹ *homogenize x p* ∈ *P*[*insert x X*]
⟨*proof*⟩

lemma *lookup-homogenize*:
assumes $x \notin$ *indets* p **and** $x \notin$ *keys* t
shows *lookup* (*homogenize* x p) (*Poly-Mapping.single* x (*poly-deg* p − *deg-pm* t) + t) = *lookup* p t
⟨*proof*⟩

lemma *keys-homogenizeI*:
assumes $x \notin$ *indets* p **and** $t \in$ *keys* p
shows *Poly-Mapping.single* x (*poly-deg* p − *deg-pm* t) + $t \in$ *keys* (*homogenize* x p) (**is** *?t* ∈ *keys ?p*)
⟨*proof*⟩

lemma *keys-homogenize*:
$x \notin$ *indets* $p \Longrightarrow$ *keys* (*homogenize* x p) = (λt. *Poly-Mapping.single* x (*poly-deg* p − *deg-pm* t) + t) ‘ *keys* p
⟨*proof*⟩

lemma *card-keys-homogenize*:
assumes $x \notin$ *indets* p
shows *card* (*keys* (*homogenize* x p)) = *card* (*keys* p)
⟨*proof*⟩

lemma *poly-deg-homogenize*:
assumes $x \notin$ *indets* p
shows *poly-deg* (*homogenize* x p) = *poly-deg* p
⟨*proof*⟩

lemma *maxdeg-homogenize*:
assumes $x \notin \bigcup$ (*indets* ‘ F)
shows *maxdeg* (*homogenize* x ‘ F) = *maxdeg* F
⟨*proof*⟩

lemma *homogeneous-ideal-homogenize*:
assumes $\bigwedge f$. $f \in F \Longrightarrow$ *homogeneous* f **and** $p \in$ *ideal* F
shows *homogenize* x $p \in$ *ideal* F
⟨*proof*⟩

lemma *subst-pp-dehomo-subst* [*simp*]:
subst-pp (*dehomo-subst* x) t = *monomial* (*1*::*'b*::*comm-semiring-1*) (*except* t {x})
⟨*proof*⟩

lemma
shows *dehomogenize-zero* [*simp*]: *dehomogenize* x *0* = *0*
and *dehomogenize-one* [*simp*]: *dehomogenize* x *1* = *1*
and *dehomogenize-monomial*: *dehomogenize* x (*monomial* c t) = *monomial* c (*except* t {x})
and *dehomogenize-plus*: *dehomogenize* x (p + q) = *dehomogenize* x p + *dehomogenize* x q
and *dehomogenize-uminus*: *dehomogenize* x (− r) = − *dehomogenize* x (r::-

$\Rightarrow_0$ -::*comm-ring-1*)
and *dehomogenize-minus*: *dehomogenize x* (*r* − *r'*) = *dehomogenize x r* − *dehomogenize x r'*
and *dehomogenize-times*: *dehomogenize x* (*p* ∗ *q*) = *dehomogenize x p* ∗ *dehomogenize x q*
and *dehomogenize-power*: *dehomogenize x* (*p* ^ *n*) = *dehomogenize x p* ^ *n*
and *dehomogenize-sum*: *dehomogenize x* (*sum f A*) = (∑ *a*∈*A*. *dehomogenize x* (*f a*))
and *dehomogenize-prod*: *dehomogenize x* (*prod f A*) = (∏ *a*∈*A*. *dehomogenize x* (*f a*))
⟨*proof*⟩

corollary *dehomogenize-monom-mult*:
dehomogenize x (*punit.monom-mult c t p*) = *punit.monom-mult c* (*except t* {*x*}) (*dehomogenize x p*)
⟨*proof*⟩

lemma *poly-deg-dehomogenize-le*: *poly-deg* (*dehomogenize x p*) ≤ *poly-deg p*
⟨*proof*⟩

lemma *indets-dehomogenize*: *indets* (*dehomogenize x p*) ⊆ *indets p* − {*x*}
for *p*::('*x* ⇒$_0$ *nat*) ⇒$_0$ '*a*::*comm-semiring-1*
⟨*proof*⟩

lemma *dehomogenize-id-iff* [*simp*]: *dehomogenize x p* = *p* ⟷ *x* ∉ *indets p*
⟨*proof*⟩

lemma *dehomogenize-dehomogenize* [*simp*]: *dehomogenize x* (*dehomogenize x p*) = *dehomogenize x p*
⟨*proof*⟩

lemma *dehomogenize-homogenize* [*simp*]: *dehomogenize x* (*homogenize x p*) = *dehomogenize x p*
⟨*proof*⟩

corollary *dehomogenize-homogenize-id*: *x* ∉ *indets p* ⟹ *dehomogenize x* (*homogenize x p*) = *p*
⟨*proof*⟩

lemma *range-dehomogenize*: *range* (*dehomogenize x*) = (*P*[− {*x*}] :: (- ⇒$_0$ '*a*::*comm-semiring-1*) *set*)
⟨*proof*⟩

lemma *dehomogenize-alt*: *dehomogenize x p* = (∑ *t*∈*keys p*. *monomial* (*lookup p t*) (*except t* {*x*}))
⟨*proof*⟩

lemma *keys-dehomogenizeE*:
assumes *t* ∈ *keys* (*dehomogenize x p*)

obtains s **where** $s \in keys\ p$ **and** $t = except\ s\ \{x\}$
⟨*proof*⟩

lemma *except-inj-on-keys-homogeneous*:
assumes *homogeneous p*
shows *inj-on* (λ*t*. *except t* {*x*}) (*keys p*)
⟨*proof*⟩

lemma *lookup-dehomogenize*:
assumes *homogeneous p* **and** $t \in keys\ p$
shows *lookup* (*dehomogenize x p*) (*except t* {*x*}) = *lookup p t*
⟨*proof*⟩

lemma *keys-dehomogenizeI*:
assumes *homogeneous p* **and** $t \in keys\ p$
shows *except t* {*x*} ∈ *keys* (*dehomogenize x p*)
⟨*proof*⟩

lemma *homogeneous-homogenize-dehomogenize*:
assumes *homogeneous p*
obtains *d* **where** *d* = *poly-deg p* − *poly-deg* (*homogenize x* (*dehomogenize x p*))
and *punit.monom-mult 1* (*Poly-Mapping.single x d*) (*homogenize x* (*dehomogenize x p*)) = *p*
⟨*proof*⟩

lemma *dehomogenize-zeroD*:
assumes *dehomogenize x p* = *0* **and** *homogeneous p*
shows *p* = *0*
⟨*proof*⟩

lemma *dehomogenize-ideal*: *dehomogenize x ‘ ideal F* = *ideal* (*dehomogenize x ‘ F*) ∩ *P*[− {*x*}]
⟨*proof*⟩

corollary *dehomogenize-ideal-subset*: *dehomogenize x ‘ ideal F* ⊆ *ideal* (*dehomogenize x ‘ F*)
⟨*proof*⟩

lemma *ideal-dehomogenize*:
assumes *ideal G* = *ideal* (*homogenize x ‘ F*) **and** *F* ⊆ *P*[*UNIV* − {*x*}]
shows *ideal* (*dehomogenize x ‘ G*) = *ideal F*
⟨*proof*⟩

17.7 Embedding Polynomial Rings in Larger Polynomial Rings (With One Additional Indeterminate)

We define a homomorphism for embedding a polynomial ring in a larger polynomial ring, and its inverse. This is mainly needed for homogenizing wrt. a fresh indeterminate.

definition *extend-indets-subst* :: *'x* ⇒ (*'x option* ⇒$_0$ *nat*) ⇒$_0$ *'a*::*comm-semiring-1*
where *extend-indets-subst x* = *monomial 1* (*Poly-Mapping.single* (*Some x*) *1*)

definition *extend-indets* :: ((*'x* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ (*'x option* ⇒$_0$ *nat*) ⇒$_0$ *'a*::*comm-semiring-1*
where *extend-indets* = *poly-subst extend-indets-subst*

definition *restrict-indets-subst* :: *'x option* ⇒ *'x* ⇒$_0$ *nat*
where *restrict-indets-subst x* = (*case x of Some y* ⇒ *Poly-Mapping.single y 1* |
- ⇒ *0*)

definition *restrict-indets* :: ((*'x option* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ (*'x* ⇒$_0$ *nat*) ⇒$_0$ *'a*::*comm-semiring-1*
where *restrict-indets* = *poly-subst* (λ*x*. *monomial 1* (*restrict-indets-subst x*))

definition *restrict-indets-pp* :: (*'x option* ⇒$_0$ *nat*) ⇒ (*'x* ⇒$_0$ *nat*)
where *restrict-indets-pp t* = (∑ *x*∈*keys t*. *lookup t x* · *restrict-indets-subst x*)

lemma *lookup-extend-indets-subst-aux*:
lookup (∑ *y*∈*keys t*. *Poly-Mapping.single* (*Some y*) (*lookup t y*)) = (λ*x*. *case x*
of Some y ⇒ *lookup t y* | *-* ⇒ *0*)
⟨*proof*⟩

lemma *keys-extend-indets-subst-aux*:
keys (∑ *y*∈*keys t*. *Poly-Mapping.single* (*Some y*) (*lookup t y*)) = *Some* ‘ *keys t*
⟨*proof*⟩

lemma *subst-pp-extend-indets-subst*:
subst-pp extend-indets-subst t = *monomial 1* (∑ *y*∈*keys t*. *Poly-Mapping.single*
(*Some y*) (*lookup t y*))
⟨*proof*⟩

lemma *keys-extend-indets*:
keys (*extend-indets p*) = (λ*t*. ∑ *y*∈*keys t*. *Poly-Mapping.single* (*Some y*) (*lookup*
t y)) ‘ *keys p*
⟨*proof*⟩

lemma *indets-extend-indets*: *indets* (*extend-indets p*) = *Some* ‘ *indets* (*p*::*-* ⇒$_0$
'a::*comm-semiring-1*)
⟨*proof*⟩

lemma *poly-deg-extend-indets* [*simp*]: *poly-deg* (*extend-indets p*) = *poly-deg p*
⟨*proof*⟩

lemma
shows *extend-indets-zero* [*simp*]: *extend-indets 0* = *0*
and *extend-indets-one* [*simp*]: *extend-indets 1* = *1*
and *extend-indets-monomial*: *extend-indets* (*monomial c t*) = *punit.monom-mult*
c 0 (*subst-pp extend-indets-subst t*)
and *extend-indets-plus*: *extend-indets* (*p* + *q*) = *extend-indets p* + *extend-indets*
q

and *extend-indets-uminus*: *extend-indets* $(-\ r) = -$ *extend-indets* (r::- $\Rightarrow_0$ -::*comm-ring-1*)
and *extend-indets-minus*: *extend-indets* $(r - r') =$ *extend-indets* $r -$ *extend-indets* r'
and *extend-indets-times*: *extend-indets* $(p * q) =$ *extend-indets* $p *$ *extend-indets* q
and *extend-indets-power*: *extend-indets* $(p\ \hat{}\ n) =$ *extend-indets* $p\ \hat{}\ n$
and *extend-indets-sum*: *extend-indets* $(sum\ f\ A) = (\sum a{\in}A.$ *extend-indets* $(f\ a))$
and *extend-indets-prod*: *extend-indets* $(prod\ f\ A) = (\prod a{\in}A.$ *extend-indets* $(f\ a))$
⟨*proof*⟩

lemma *extend-indets-zero-iff* [*simp*]: *extend-indets* $p = 0 \longleftrightarrow p = 0$
⟨*proof*⟩

lemma *extend-indets-inject*:
assumes *extend-indets* $p =$ *extend-indets* (q::- $\Rightarrow_0$ -::*comm-ring-1*)
shows $p = q$
⟨*proof*⟩

corollary *inj-extend-indets*: *inj* (*extend-indets*::- $\Rightarrow$ - $\Rightarrow_0$ -::*comm-ring-1*)
⟨*proof*⟩

lemma *poly-subst-extend-indets*: *poly-subst* f (*extend-indets* p) = *poly-subst* $(f \circ$ *Some*$)$ p
⟨*proof*⟩

lemma *poly-eval-extend-indets*: *poly-eval* a (*extend-indets* p) = *poly-eval* $(a \circ$ *Some*$)$ p
⟨*proof*⟩

lemma *lookup-restrict-indets-pp*: *lookup* (*restrict-indets-pp* t) = $(\lambda x.$ *lookup* t (*Some* x))
⟨*proof*⟩

lemma *keys-restrict-indets-pp*: *keys* (*restrict-indets-pp* t) = *the* ‘ (*keys* $t - \{None\}$)
⟨*proof*⟩

lemma *subst-pp-restrict-indets-subst*:
subst-pp $(\lambda x.$ *monomial 1* (*restrict-indets-subst* x)) $t =$ *monomial 1* (*restrict-indets-pp* t)
⟨*proof*⟩

lemma *restrict-indets-pp-zero* [*simp*]: *restrict-indets-pp 0 = 0*
⟨*proof*⟩

lemma *restrict-indets-pp-plus*: *restrict-indets-pp* $(s + t) =$ *restrict-indets-pp* $s +$ *restrict-indets-pp* t

⟨*proof*⟩

lemma *restrict-indets-pp-except-None* [*simp*]:
restrict-indets-pp (*except t* {*None*}) = *restrict-indets-pp t*
⟨*proof*⟩

lemma *deg-pm-restrict-indets-pp*: *deg-pm* (*restrict-indets-pp t*) + *lookup t None* = *deg-pm t*
⟨*proof*⟩

lemma *keys-restrict-indets-subset*: *keys* (*restrict-indets p*) ⊆ *restrict-indets-pp* ‘ *keys p*
⟨*proof*⟩

lemma *keys-restrict-indets*:
assumes *None* ∉ *indets p*
shows *keys* (*restrict-indets p*) = *restrict-indets-pp* ‘ *keys p*
⟨*proof*⟩

lemma *indets-restrict-indets-subset*: *indets* (*restrict-indets p*) ⊆ *the* ‘ (*indets p* − {*None*})
⟨*proof*⟩

lemma *poly-deg-restrict-indets-le*: *poly-deg* (*restrict-indets p*) ≤ *poly-deg p*
⟨*proof*⟩

lemma
shows *restrict-indets-zero* [*simp*]: *restrict-indets 0* = *0*
and *restrict-indets-one* [*simp*]: *restrict-indets 1* = *1*
and *restrict-indets-monomial*: *restrict-indets* (*monomial c t*) = *monomial c* (*restrict-indets-pp t*)
and *restrict-indets-plus*: *restrict-indets* (*p* + *q*) = *restrict-indets p* + *restrict-indets q*
and *restrict-indets-uminus*: *restrict-indets* (− *r*) = − *restrict-indets* (*r*::- ⇒$_0$ -::*comm-ring-1*)
and *restrict-indets-minus*: *restrict-indets* (*r* − *r*′) = *restrict-indets r* − *restrict-indets r*′
and *restrict-indets-times*: *restrict-indets* (*p* ∗ *q*) = *restrict-indets p* ∗ *restrict-indets q*
and *restrict-indets-power*: *restrict-indets* (*p* ^ *n*) = *restrict-indets p* ^ *n*
and *restrict-indets-sum*: *restrict-indets* (*sum f A*) = (∑ *a*∈*A*. *restrict-indets* (*f a*))
and *restrict-indets-prod*: *restrict-indets* (*prod f A*) = (∏ *a*∈*A*. *restrict-indets* (*f a*))
⟨*proof*⟩

lemma *restrict-extend-indets* [*simp*]: *restrict-indets* (*extend-indets p*) = *p*
⟨*proof*⟩

lemma *extend-restrict-indets*:
assumes *None* $\notin$ *indets p*
shows *extend-indets* (*restrict-indets p*) = *p*
⟨*proof*⟩

lemma *restrict-indets-dehomogenize* [*simp*]: *restrict-indets* (*dehomogenize None p*) = *restrict-indets p*
⟨*proof*⟩

corollary *restrict-indets-comp-dehomogenize*: *restrict-indets* ∘ *dehomogenize None* = *restrict-indets*
⟨*proof*⟩

corollary *extend-restrict-indets-eq-dehomogenize*:
extend-indets (*restrict-indets p*) = *dehomogenize None p*
⟨*proof*⟩

corollary *extend-indets-comp-restrict-indets*: *extend-indets* ∘ *restrict-indets* = *dehomogenize None*
⟨*proof*⟩

lemma *restrict-homogenize-extend-indets* [*simp*]:
restrict-indets (*homogenize None* (*extend-indets p*)) = *p*
⟨*proof*⟩

lemma *dehomogenize-extend-indets* [*simp*]: *dehomogenize None* (*extend-indets p*) = *extend-indets p*
⟨*proof*⟩

lemma *restrict-indets-ideal*: *restrict-indets* ‘ *ideal F* = *ideal* (*restrict-indets* ‘ *F*)
⟨*proof*⟩

lemma *ideal-restrict-indets*:
ideal G = *ideal* (*homogenize None* ‘ *extend-indets* ‘ *F*) ⟹ *ideal* (*restrict-indets* ‘ *G*) = *ideal F*
⟨*proof*⟩

lemma *extend-indets-ideal*: *extend-indets* ‘ *ideal F* = *ideal* (*extend-indets* ‘ *F*) ∩ *P*[− {*None*}]
⟨*proof*⟩

corollary *extend-indets-ideal-subset*: *extend-indets* ‘ *ideal F* ⊆ *ideal* (*extend-indets* ‘ *F*)
⟨*proof*⟩

17.8 Canonical Isomorphisms between $P[X, Y]$ and $P[X][Y]$: *focus* and *flatten*

definition *focus* :: *'x set* ⇒ ((*'x* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ ((*'x* ⇒$_0$ *nat*) ⇒$_0$ (*'x* ⇒$_0$ *nat*) ⇒$_0$ *'a*::*comm-monoid-add*)
where *focus X p* = (∑ *t*∈*keys p*. *monomial* (*monomial* (*lookup p t*) (*except t X*)) (*except t* (− *X*)))

definition *flatten* :: (*'a* ⇒$_0$ *'a* ⇒$_0$ *'b*) ⇒ (*'a*::*comm-powerprod* ⇒$_0$ *'b*::*semiring-1*)
where *flatten p* = (∑ *t*∈*keys p*. *punit.monom-mult 1 t* (*lookup p t*))

lemma *focus-superset*:
assumes *finite A* **and** *keys p* ⊆ *A*
shows *focus X p* = (∑ *t*∈*A*. *monomial* (*monomial* (*lookup p t*) (*except t X*)) (*except t* (− *X*)))
⟨*proof*⟩

lemma *keys-focus*: *keys* (*focus X p*) = (λ*t*. *except t* (− *X*)) ‘ *keys p*
⟨*proof*⟩

lemma *keys-coeffs-focus-subset*:
assumes *c* ∈ *range* (*lookup* (*focus X p*))
shows *keys c* ⊆ (λ*t*. *except t X*) ‘ *keys p*
⟨*proof*⟩

lemma *focus-in-Polys'*:
assumes *p* ∈ *P*[*Y*]
shows *focus X p* ∈ *P*[*Y* ∩ *X*]
⟨*proof*⟩

corollary *focus-in-Polys*: *focus X p* ∈ *P*[*X*]
⟨*proof*⟩

lemma *focus-coeffs-subset-Polys'*:
assumes *p* ∈ *P*[*Y*]
shows *range* (*lookup* (*focus X p*)) ⊆ *P*[*Y* − *X*]
⟨*proof*⟩

corollary *focus-coeffs-subset-Polys*: *range* (*lookup* (*focus X p*)) ⊆ *P*[− *X*]
⟨*proof*⟩

corollary *lookup-focus-in-Polys*: *lookup* (*focus X p*) *t* ∈ *P*[− *X*]
⟨*proof*⟩

lemma *focus-zero* [*simp*]: *focus X 0* = *0*
⟨*proof*⟩

lemma *focus-eq-zero-iff* [*iff*]: *focus X p* = *0* ⟷ *p* = *0*
⟨*proof*⟩

lemma *focus-one* [*simp*]: *focus X 1 = 1*
⟨*proof*⟩

lemma *focus-monomial*: *focus X* (*monomial c t*) = *monomial* (*monomial c* (*except t X*)) (*except t* (− *X*))
⟨*proof*⟩

lemma *focus-uminus* [*simp*]: *focus X* (− *p*) = − *focus X p*
⟨*proof*⟩

lemma *focus-plus*: *focus X* (*p* + *q*) = *focus X p* + *focus X q*
⟨*proof*⟩

lemma *focus-minus*: *focus X* (*p* − *q*) = *focus X p* − *focus X* (*q*::- ⇒$_0$ -::*ab-group-add*)
⟨*proof*⟩

lemma *focus-times*: *focus X* (*p* ∗ *q*) = *focus X p* ∗ *focus X q*
⟨*proof*⟩

lemma *focus-sum*: *focus X* (*sum f I*) = (∑ *i*∈*I*. *focus X* (*f i*))
⟨*proof*⟩

lemma *focus-prod*: *focus X* (*prod f I*) = (∏ *i*∈*I*. *focus X* (*f i*))
⟨*proof*⟩

lemma *focus-power* [*simp*]: *focus X* (*f* ^ *m*) = *focus X f* ^ *m*
⟨*proof*⟩

lemma *focus-Polys*:
assumes *p* ∈ *P*[*X*]
shows *focus X p* = (∑ *t*∈*keys p*. *monomial* (*monomial* (*lookup p t*) *0*) *t*)
⟨*proof*⟩

corollary *lookup-focus-Polys*: *p* ∈ *P*[*X*] ⟹ *lookup* (*focus X p*) *t* = *monomial* (*lookup p t*) *0*
⟨*proof*⟩

lemma *focus-Polys-Compl*:
assumes *p* ∈ *P*[− *X*]
shows *focus X p* = *monomial p 0*
⟨*proof*⟩

corollary *focus-empty* [*simp*]: *focus* {} *p* = *monomial p 0*
⟨*proof*⟩

lemma *focus-Int*:
assumes *p* ∈ *P*[*Y*]
shows *focus* (*X* ∩ *Y*) *p* = *focus X p*
⟨*proof*⟩

lemma *range-focusD*:
assumes $p \in$ *range* (*focus* X)
shows $p \in P[X]$ **and** *range* (*lookup* p) $\subseteq P[-\ X]$ **and** *lookup* $p\ t \in P[-\ X]$
⟨*proof*⟩

lemma *range-focusI*:
assumes $p \in P[X]$ **and** *lookup* p ‘ *keys* (p::- $\Rightarrow_0$ - $\Rightarrow_0$ -::*semiring-1*) $\subseteq P[-\ X]$
shows $p \in$ *range* (*focus* X)
⟨*proof*⟩

lemma *inj-focus*: *inj* ((*focus* X) :: (('x $\Rightarrow_0$ *nat*) $\Rightarrow_0$ 'a::*ab-group-add*) $\Rightarrow$ -)
⟨*proof*⟩

lemma *flatten-superset*:
assumes *finite* A **and** *keys* $p \subseteq A$
shows *flatten* p = ($\sum t{\in}A$. *punit.monom-mult 1 t* (*lookup* $p\ t$))
⟨*proof*⟩

lemma *keys-flatten-subset*: *keys* (*flatten* p) $\subseteq$ ($\bigcup t{\in}$*keys* p. (+) t ‘ *keys* (*lookup* p t))
⟨*proof*⟩

lemma *flatten-in-Polys*:
assumes $p \in P[X]$ **and** *lookup* p ‘ *keys* $p \subseteq P[Y]$
shows *flatten* $p \in P[X \cup Y]$
⟨*proof*⟩

lemma *flatten-zero* [*simp*]: *flatten 0 = 0*
⟨*proof*⟩

lemma *flatten-one* [*simp*]: *flatten 1 = 1*
⟨*proof*⟩

lemma *flatten-monomial*: *flatten* (*monomial* $c\ t$) = *punit.monom-mult 1 t c*
⟨*proof*⟩

lemma *flatten-uminus* [*simp*]: *flatten* (− p) = − *flatten* (p::- $\Rightarrow_0$ - $\Rightarrow_0$ -::*ring*)
⟨*proof*⟩

lemma *flatten-plus*: *flatten* (p + q) = *flatten* p + *flatten* q
⟨*proof*⟩

lemma *flatten-minus*: *flatten* (p − q) = *flatten* p − *flatten* (q::- $\Rightarrow_0$ - $\Rightarrow_0$ -::*ring*)
⟨*proof*⟩

lemma *flatten-times*: *flatten* (p * q) = *flatten* p * *flatten* (q::- $\Rightarrow_0$ - $\Rightarrow_0$ 'b::*comm-semiring-1*)
⟨*proof*⟩

lemma *flatten-monom-mult*:
flatten (*punit.monom-mult c t p*) = *punit.monom-mult 1 t* (*c* ∗ *flatten* (*p*::- ⇒$_0$
- ⇒$_0$ *'b*::*comm-semiring-1*))
⟨*proof*⟩

lemma *flatten-sum*: *flatten* (*sum f I*) = (∑ *i*∈*I*. *flatten* (*f i*))
⟨*proof*⟩

lemma *flatten-prod*: *flatten* (*prod f I*) = (∏ *i*∈*I*. *flatten* (*f i* :: - ⇒$_0$ -::*comm-semiring-1*))
⟨*proof*⟩

lemma *flatten-power* [*simp*]: *flatten* (*f* ^ *m*) = *flatten* (*f*:: - ⇒$_0$ -::*comm-semiring-1*)
^ *m*
⟨*proof*⟩

lemma *surj-flatten*: *surj flatten*
⟨*proof*⟩

lemma *flatten-focus* [*simp*]: *flatten* (*focus X p*) = *p*
⟨*proof*⟩

lemma *focus-flatten*:
assumes *p* ∈ *P*[*X*] **and** *lookup p* ‘ *keys p* ⊆ *P*[− *X*]
shows *focus X* (*flatten p*) = *p*
⟨*proof*⟩

lemma *image-focus-ideal*: *focus X* ‘ *ideal F* = *ideal* (*focus X* ‘ *F*) ∩ *range* (*focus*
X)
⟨*proof*⟩

lemma *image-flatten-ideal*: *flatten* ‘ *ideal F* = *ideal* (*flatten* ‘ *F*)
⟨*proof*⟩

lemma *poly-eval-focus*:
poly-eval a (*focus X p*) = *poly-subst* (λ*x*. *if x* ∈ *X then a x else monomial 1*
(*Poly-Mapping.single x 1*)) *p*
⟨*proof*⟩

corollary *poly-eval-poly-eval-focus*:
poly-eval a (*poly-eval b* (*focus X p*)) = *poly-eval* (λ*x*::*'x*. *if x* ∈ *X then poly-eval*
a (*b x*) *else a x*) *p*
⟨*proof*⟩

lemma *indets-poly-eval-focus-subset*:
indets (*poly-eval a* (*focus X p*)) ⊆ ⋃ (*indets* ‘ *a* ‘ *X*) ∪ (*indets p* − *X*)
⟨*proof*⟩

lemma *lookup-poly-eval-focus*:
lookup (*poly-eval* (λ*x*. *monomial* (*a x*) *0*) (*focus X p*)) *t* = *poly-eval a* (*lookup*

(*focus* (− *X*) *p*) *t*)
⟨*proof*⟩

lemma *keys-poly-eval-focus-subset*:
keys (*poly-eval* (λ*x*. *monomial* (*a x*) *0*) (*focus X p*)) ⊆ (λ*t*. *except t X*) ‘ *keys p*
⟨*proof*⟩

lemma *poly-eval-focus-in-Polys*:
assumes *p* ∈ *P*[*X*]
shows *poly-eval* (λ*x*. *monomial* (*a x*) *0*) (*focus Y p*) ∈ *P*[*X* − *Y*]
⟨*proof*⟩

lemma *image-poly-eval-focus-ideal*:
poly-eval (λ*x*. *monomial* (*a x*) *0*) ‘ *focus X* ‘ *ideal F* =
ideal (*poly-eval* (λ*x*. *monomial* (*a x*) *0*) ‘ *focus X* ‘ *F*) ∩
(*P*[− *X*]::((′*x* ⇒$_0$ *nat*) ⇒$_0$ ′*a*::*comm-ring-1*) *set*)
⟨*proof*⟩

17.9 Locale *pm-powerprod*

lemma *varnum-eq-zero-iff*: *varnum X t* = *0* ⟷ *t* ∈ .[*X*]
⟨*proof*⟩

lemma *dgrad-set-varnum*: *dgrad-set* (*varnum X*) *0* = .[*X*]
⟨*proof*⟩

context *ordered-powerprod*
begin

abbreviation *lcf* ≡ *punit.lc*
abbreviation *tcf* ≡ *punit.tc*
abbreviation *lpp* ≡ *punit.lt*
abbreviation *tpp* ≡ *punit.tt*

end

locale *pm-powerprod* =
ordered-powerprod ord ord-strict
for *ord*::(′*x*::{*countable*,*linorder*} ⇒$_0$ *nat*) ⇒ (′*x* ⇒$_0$ *nat*) ⇒ *bool* (**infixl** ⪯ *50*)
and *ord-strict* (**infixl** ≺ *50*)
begin

sublocale *gd-powerprod* ⟨*proof*⟩

lemma *PPs-closed-lpp*:
assumes *p* ∈ *P*[*X*]
shows *lpp p* ∈ .[*X*]
⟨*proof*⟩

lemma *PPs-closed-tpp*:
assumes $p \in P[X]$
shows *tpp* $p \in .[X]$
⟨*proof*⟩

corollary *PPs-closed-image-lpp*: $F \subseteq P[X] \Longrightarrow$ *lpp* ‘ $F \subseteq .[X]$
⟨*proof*⟩

corollary *PPs-closed-image-tpp*: $F \subseteq P[X] \Longrightarrow$ *tpp* ‘ $F \subseteq .[X]$
⟨*proof*⟩

lemma *hom-component-lpp*:
assumes $p \neq 0$
shows *hom-component* p (*deg-pm* (*lpp* p)) $\neq 0$ (**is** $?p \neq 0$)
and *lpp* (*hom-component* p (*deg-pm* (*lpp* p))) = *lpp* p
⟨*proof*⟩

definition *is-hom-ord* :: $'x \Rightarrow$ *bool*
where *is-hom-ord* $x \longleftrightarrow (\forall s\ t.$ *deg-pm* $s =$ *deg-pm* $t \longrightarrow (s \preceq t \longleftrightarrow$ *except* s $\{x\} \preceq$ *except* t $\{x\}))$

lemma *is-hom-ordD*: *is-hom-ord* $x \Longrightarrow$ *deg-pm* $s =$ *deg-pm* $t \Longrightarrow s \preceq t \longleftrightarrow$ *except* s $\{x\} \preceq$ *except* t $\{x\}$
⟨*proof*⟩

lemma *dgrad-p-set-varnum*: *punit.dgrad-p-set* (*varnum* X) $0 = P[X]$
⟨*proof*⟩

end

We must create a copy of *pm-powerprod* to avoid infinite chains of interpretations.

instantiation *option* :: (*linorder*) *linorder*
begin

fun *less-eq-option* :: $'a$ *option* $\Rightarrow$ $'a$ *option* $\Rightarrow$ *bool* **where**
less-eq-option None - = *True* |
less-eq-option (*Some* x) *None* = *False* |
less-eq-option (*Some* x) (*Some* y) = $(x \leq y)$

definition *less-option* :: $'a$ *option* $\Rightarrow$ $'a$ *option* $\Rightarrow$ *bool*
where *less-option* $x\ y \longleftrightarrow x \leq y \wedge \neg\ y \leq x$

instance ⟨*proof*⟩

end

locale *extended-ord-pm-powerprod* = *pm-powerprod*
begin

definition *extended-ord* :: (*'a option* $\Rightarrow_0$ *nat*) $\Rightarrow$ (*'a option* $\Rightarrow_0$ *nat*) $\Rightarrow$ *bool*
where *extended-ord s t* $\longleftrightarrow$ (*restrict-indets-pp s* $\prec$ *restrict-indets-pp t* $\vee$
(*restrict-indets-pp s* = *restrict-indets-pp t* $\wedge$ *lookup s None* $\leq$ *lookup t None*))

definition *extended-ord-strict* :: (*'a option* $\Rightarrow_0$ *nat*) $\Rightarrow$ (*'a option* $\Rightarrow_0$ *nat*) $\Rightarrow$ *bool*
where *extended-ord-strict s t* $\longleftrightarrow$ (*restrict-indets-pp s* $\prec$ *restrict-indets-pp t* $\vee$
(*restrict-indets-pp s* = *restrict-indets-pp t* $\wedge$ *lookup s None* $<$ *lookup t None*))

sublocale *extended-ord*: *pm-powerprod extended-ord extended-ord-strict*
⟨*proof*⟩

lemma *extended-ord-is-hom-ord*: *extended-ord.is-hom-ord None*
⟨*proof*⟩

end

end

theory *MPoly-Type-Univariate*
imports
More-MPoly-Type
HOL−Computational-Algebra.Polynomial
begin

This file connects univariate MPolys to the theory of univariate polynomials from *HOL−Computational-Algebra.Polynomial*.

definition *poly-to-mpoly*::*nat* $\Rightarrow$ *'a*::*comm-monoid-add poly* $\Rightarrow$ *'a mpoly*
where *poly-to-mpoly v p* = *MPoly* (*Abs-poly-mapping* (λ*m*. (*coeff p* (*Poly-Mapping.lookup m v*)) *when Poly-Mapping.keys m* $\subseteq$ {*v*}))

lemma *poly-to-mpoly-finite*: *finite* {*m*::*nat* $\Rightarrow_0$ *nat*. (*coeff p* (*Poly-Mapping.lookup m v*) *when Poly-Mapping.keys m* $\subseteq$ {*v*}) $\neq$ *0*} (**is** *finite ?M*)
⟨*proof*⟩

lemma *coeff-poly-to-mpoly*: *MPoly-Type.coeff* (*poly-to-mpoly v p*) (*Poly-Mapping.single v k*) = *Polynomial.coeff p k*
⟨*proof*⟩

definition *mpoly-to-poly*::*nat* $\Rightarrow$ *'a*::*comm-monoid-add mpoly* $\Rightarrow$ *'a poly*
where *mpoly-to-poly v p* = *Abs-poly* (λ*k*. *MPoly-Type.coeff p* (*Poly-Mapping.single v k*))

lemma *coeff-mpoly-to-poly*[*simp*]: *Polynomial.coeff* (*mpoly-to-poly v p*) *k* = *MPoly-Type.coeff p* (*Poly-Mapping.single v k*)
⟨*proof*⟩

lemma *mpoly-to-poly-inverse*:
assumes *vars p* ⊆ {*v*}
shows *poly-to-mpoly v* (*mpoly-to-poly v p*) = *p*
⟨*proof*⟩

lemma *poly-to-mpoly-inverse*: *mpoly-to-poly v* (*poly-to-mpoly v p*) = *p*
⟨*proof*⟩

lemma *poly-to-mpoly0*: *poly-to-mpoly v 0* = *0*
⟨*proof*⟩

lemma *mpoly-to-poly-add*: *mpoly-to-poly v* (*p1* + *p2*) = *mpoly-to-poly v p1* + *mpoly-to-poly v p2*
⟨*proof*⟩

lemma *poly-eq-insertion*:
assumes *vars p* ⊆ {*v*}
shows *poly* (*mpoly-to-poly v p*) *x* = *insertion* (λ*v*. *x*) *p*
⟨*proof*⟩

Using the new connection between MPoly and univariate polynomials, we can transfer:

lemma *univariate-mpoly-roots-finite*:
fixes *p*::'*a*::*idom mpoly*
assumes *vars p* ⊆ {*v*} *p* ≠ *0*
shows *finite* {*x*. *insertion* (λ*v*. *x*) *p* = *0*}
⟨*proof*⟩

end

18 Polynomials

theory *Polynomials*
imports
Abstract−Rewriting.SN-Orders
Matrix.Utility
begin

18.1 Polynomials represented as trees

datatype (*vars-tpoly*: '*v*, *nums-tpoly*: '*a*)*tpoly* = *PVar* '*v* | *PNum* '*a* | *PSum* ('*v*,'*a*)*tpoly list* | *PMult* ('*v*,'*a*)*tpoly list*

type-synonym ('*v*,'*a*)*assign* = '*v* ⇒ '*a*

primrec *eval-tpoly* :: ('*v*,'*a*::{*monoid-add*,*monoid-mult*})*assign* ⇒ ('*v*,'*a*)*tpoly* ⇒ '*a*
where *eval-tpoly* α (*PVar x*) = α *x*
| *eval-tpoly* α (*PNum a*) = *a*

| *eval-tpoly α (PSum ps) = sum-list (map (eval-tpoly α) ps)*
| *eval-tpoly α (PMult ps) = prod-list (map (eval-tpoly α) ps)*

18.2 Polynomials represented in normal form as lists of monomials

The internal representation of polynomials is a sum of products of monomials with coefficients where all coefficients are non-zero, and all monomials are different

Definition of type *monom*

type-synonym *'v monom-list = ('v × nat)list*

- $[(x, n), (y, m)]$ represent $x^n \cdot y^m$
- invariants: all powers are ≥ 1 and each variable occurs at most once hence: $[(x, 1), (y, 2), (x, 2)]$ will not occur, but $[(x, 3), (y, 2)]$; $[(x, 1), (y, 0)]$ will not occur, but $[(x, 1)]$

context *linorder*
begin
definition *monom-inv :: 'a monom-list ⇒ bool* **where**
monom-inv m ≡ (∀ (x,n) ∈ set m. 1 ≤ n) ∧ distinct (map fst m) ∧ sorted (map fst m)

fun *eval-monom-list :: ('a,'b :: comm-semiring-1)assign ⇒ ('a monom-list) ⇒ 'b* **where**
eval-monom-list α [] = 1
| *eval-monom-list α ((x,p) # m) = eval-monom-list α m ∗ (α x) ^p*

lemma *eval-monom-list[simp]: eval-monom-list α (m @ n) = eval-monom-list α m ∗ eval-monom-list α n*
⟨*proof*⟩

definition *sum-var-list :: 'a monom-list ⇒ 'a ⇒ nat* **where**
sum-var-list m x ≡ sum-list (map (λ (y,c). if x = y then c else 0) m)

lemma *sum-var-list-not: x ∉ fst ' set m ⟹ sum-var-list m x = 0*
⟨*proof*⟩

show that equality of monomials is equivalent to statement that all variables occur with the same (accumulated) power; afterwards properties like transitivity, etc. are easy to prove

lemma *monom-inv-Cons*: **assumes** *monom-inv ((x,p) # m)*
and *y ≤ x* **shows** *y ∉ fst ' set m*
⟨*proof*⟩

lemma *eq-monom-sum-var-list*: **assumes** *monom-inv m* **and** *monom-inv n*

shows $(m = n) = (\forall\ x.\ sum\text{-}var\text{-}list\ m\ x = sum\text{-}var\text{-}list\ n\ x)$ (**is** *?l* = *?r*)
⟨*proof*⟩

equality of monomials is also a complete for several carriers, e.g. the naturals, integers, where $x^p = x^q$ implies $p = q$. note that it is not complete for carriers like the Booleans where e.g. $x^{Suc(m)} = x^{Suc(n)}$ for all n, m.

abbreviation (*input*) *monom-list-vars* :: *'a monom-list* ⇒ *'a set*
where *monom-list-vars m* ≡ *fst* ' *set m*

fun *monom-mult-list* :: *'a monom-list* ⇒ *'a monom-list* ⇒ *'a monom-list* **where**
monom-mult-list [] *n* = *n*
| *monom-mult-list* ((*x*,*p*) # *m*) *n* = (*case n of*
Nil ⇒ (*x*,*p*) # *m*
| (*y*,*q*) # *n'* ⇒ *if x* = *y then* (*x*,*p* + *q*) # *monom-mult-list m n' else*
if x < *y then* (*x*,*p*) # *monom-mult-list m n else* (*y*,*q*) # *monom-mult-list*
((*x*,*p*) # *m*) *n'*)

lemma *monom-list-mult-list-vars*: *monom-list-vars* (*monom-mult-list m1 m2*) = *monom-list-vars m1* ∪ *monom-list-vars m2*
⟨*proof*⟩

lemma *monom-mult-list-inv*: *monom-inv m1* ⟹ *monom-inv m2* ⟹ *monom-inv* (*monom-mult-list m1 m2*)
⟨*proof*⟩

lemma *monom-inv-ConsD*: *monom-inv* (*x* # *xs*) ⟹ *monom-inv xs*
⟨*proof*⟩

lemma *sum-var-list-monom-mult-list*: *sum-var-list* (*monom-mult-list m n*) *x* = *sum-var-list m x* + *sum-var-list n x*
⟨*proof*⟩

lemma *monom-mult-list-inj*: **assumes** *m*: *monom-inv m* **and** *m1*: *monom-inv m1* **and** *m2*: *monom-inv m2*
and *eq*: *monom-mult-list m m1* = *monom-mult-list m m2*
shows *m1* = *m2*
⟨*proof*⟩

lemma *monom-mult-list*[*simp*]: *eval-monom-list* α (*monom-mult-list m n*) = *eval-monom-list* α *m* ∗ *eval-monom-list* α *n*
⟨*proof*⟩
end

declare *monom-mult-list.simps*[*simp del*]

typedef (**overloaded**) *'v monom* = *Collect* (*monom-inv* :: *'v* :: *linorder monom-list* ⇒ *bool*)
⟨*proof*⟩

setup-lifting *type-definition-monom*

lift-definition *eval-monom* :: ('*v* :: *linorder*,'*a* :: *comm-semiring-1*)*assign* ⇒ '*v monom* ⇒ '*a*
is *eval-monom-list* ⟨*proof*⟩

lift-definition *sum-var* :: '*v* :: *linorder monom* ⇒ '*v* ⇒ *nat* **is** *sum-var-list* ⟨*proof*⟩

instantiation *monom* :: (*linorder*) *comm-monoid-mult*
begin

lift-definition *times-monom* :: '*a monom* ⇒ '*a monom* ⇒ '*a monom* **is** *monom-mult-list*
⟨*proof*⟩

lift-definition *one-monom* :: '*a monom* **is** *Nil*
⟨*proof*⟩

instance
⟨*proof*⟩
end

lemma *eq-monom-sum-var*: *m* = *n* ⟷ (∀ *x*. *sum-var m x* = *sum-var n x*)
⟨*proof*⟩

lemma *eval-monom-mult*[*simp*]: *eval-monom* α (*m* ∗ *n*) = *eval-monom* α *m* ∗ *eval-monom* α *n*
⟨*proof*⟩

lemma *sum-var-monom-mult*: *sum-var* (*m* ∗ *n*) *x* = *sum-var m x* + *sum-var n x*
⟨*proof*⟩

lemma *monom-mult-inj*: **fixes** *m1* :: - *monom*
shows *m* ∗ *m1* = *m* ∗ *m2* ⟹ *m1* = *m2*
⟨*proof*⟩

lemma *one-monom-inv-sum-var-inv*[*simp*]: *sum-var 1 x* = *0*
⟨*proof*⟩

lemma *eval-monom-1*[*simp*]: *eval-monom* α *1* = *1*
⟨*proof*⟩

lift-definition *var-monom* :: '*v* :: *linorder* ⇒ '*v monom* **is** λ *x*. [(*x*,*1*)]
⟨*proof*⟩

lemma *var-monom-1*[*simp*]: *var-monom x* ≠ *1*
⟨*proof*⟩

lemma *eval-var-monom*[*simp*]: *eval-monom* α (*var-monom x*) = α *x*
⟨*proof*⟩

lemma *sum-var-monom-var*: *sum-var* (*var-monom x*) *y* = (*if x* = *y then 1 else 0*)
⟨*proof*⟩

instantiation *monom* :: ({*equal*,*linorder*})*equal*
begin

lift-definition *equal-monom* :: 'a *monom* ⇒ 'a *monom* ⇒ *bool* **is** (=) ⟨*proof*⟩

instance ⟨*proof*⟩
end

Polynomials are represented with as sum of monomials multiplied by some coefficient

type-synonym ('*v*,'*a*)*poly* = ('*v monom* × '*a*)*list*

The polynomials we construct satisfy the following invariants:

- all coefficients are non-zero
- the monomial list is distinct

definition *poly-inv* :: ('*v*,'*a* :: *zero*)*poly* ⇒ *bool*
where *poly-inv p* ≡ (∀ *c* ∈ *snd* ' *set p*. *c* ≠ *0*) ∧ *distinct* (*map fst p*)

abbreviation *eval-monomc* **where** *eval-monomc* α *mc* ≡ *eval-monom* α (*fst mc*) ∗ (*snd mc*)

primrec *eval-poly* :: ('*v* :: *linorder*, '*a* :: *comm-semiring-1*)*assign* ⇒ ('*v*,'*a*)*poly* ⇒ '*a* **where**
eval-poly α [] = *0*
| *eval-poly* α (*mc* # *p*) = *eval-monomc* α *mc* + *eval-poly* α *p*

definition *poly-const* :: '*a* :: *zero* ⇒ ('*v* :: *linorder*,'*a*)*poly* **where**
poly-const a = (*if a* = *0 then* [] *else* [(*1*,*a*)])

lemma *poly-const*[*simp*]: *eval-poly* α (*poly-const a*) = *a*
⟨*proof*⟩

lemma *poly-const-inv*: *poly-inv* (*poly-const a*)
⟨*proof*⟩

fun *poly-add* :: ('*v*,'*a*)*poly* ⇒ ('*v*,'*a* :: *semiring-0*)*poly* ⇒ ('*v*,'*a*)*poly* **where**
poly-add [] *q* = *q*
| *poly-add* ((*m*,*c*) # *p*) *q* = (*case List.extract* (λ *mc*. *fst mc* = *m*) *q of*
None ⇒ (*m*,*c*) # *poly-add p q*

| *Some* (*q1*,(-,*d*),*q2*) ⇒ *if* (*c*+*d* = *0*) *then poly-add p* (*q1* @ *q2*) *else* (*m*,*c*+*d*) # *poly-add p* (*q1* @ *q2*))

lemma *eval-poly-append*[*simp*]: *eval-poly* α (*mc1* @ *mc2*) = *eval-poly* α *mc1* + *eval-poly* α *mc2*
⟨*proof*⟩

abbreviation *poly-monoms* :: ('*v*,'*a*)*poly* ⇒ '*v monom set*
where *poly-monoms p* ≡ *fst* ' *set p*

lemma *poly-add-monoms*: *poly-monoms* (*poly-add p1 p2*) ⊆ *poly-monoms p1* ∪ *poly-monoms p2*
⟨*proof*⟩

lemma *poly-add-inv*: *poly-inv p* ⟹ *poly-inv q* ⟹ *poly-inv* (*poly-add p q*)
⟨*proof*⟩

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

declare *poly-add.simps*[*simp del*]

fun *monom-mult-poly* :: ('*v* :: *linorder monom* × '*a*) ⇒ ('*v*,'*a* :: *semiring-0*)*poly* ⇒ ('*v*,'*a*)*poly* **where**
monom-mult-poly - [] = []
| *monom-mult-poly* (*m*,*c*) ((*m*',*d*) # *p*) = (*if c* ∗ *d* = *0 then monom-mult-poly* (*m*,*c*) *p else* (*m* ∗ *m*', *c* ∗ *d*) # *monom-mult-poly* (*m*,*c*) *p*)

lemma *monom-mult-poly-inv*: *poly-inv p* ⟹ *poly-inv* (*monom-mult-poly* (*m*,*c*) *p*)
⟨*proof*⟩

lemma *monom-mult-poly*[*simp*]: *eval-poly* α (*monom-mult-poly mc p*) = *eval-monomc* α *mc* ∗ *eval-poly* α *p*
⟨*proof*⟩

declare *monom-mult-poly.simps*[*simp del*]

definition *poly-minus* :: ('*v* :: *linorder*,'*a* :: *ring-1*)*poly* ⇒ ('*v*,'*a*)*poly* ⇒ ('*v*,'*a*)*poly* **where**
poly-minus f g = *poly-add f* (*monom-mult-poly* (*1*,−*1*) *g*)

lemma *poly-minus*[*simp*]: *eval-poly* α (*poly-minus f g*) = *eval-poly* α *f* − *eval-poly* α *g*
⟨*proof*⟩

lemma *poly-minus-inv*: *poly-inv f* ⟹ *poly-inv g* ⟹ *poly-inv* (*poly-minus f g*)
⟨*proof*⟩

fun *poly-mult* :: (*'v* :: *linorder*, *'a* :: *semiring-0*)*poly* ⇒ (*'v*,*'a*)*poly* ⇒ (*'v*,*'a*)*poly*
where
  poly-mult [] *q* = []
| *poly-mult* (*mc* # *p*) *q* = *poly-add* (*monom-mult-poly mc q*) (*poly-mult p q*)

lemma *poly-mult-inv*: **assumes** *p*: *poly-inv p* **and** *q*: *poly-inv q*
  shows *poly-inv* (*poly-mult p q*)
⟨*proof*⟩

lemma *poly-mult*[*simp*]: *eval-poly* α (*poly-mult p q*) = *eval-poly* α *p* ∗ *eval-poly* α *q*
  ⟨*proof*⟩

declare *poly-mult.simps*[*simp del*]

definition *zero-poly* :: (*'v*,*'a*)*poly*
where *zero-poly* ≡ []

lemma *zero-poly-inv*: *poly-inv zero-poly* ⟨*proof*⟩

definition *one-poly* :: (*'v* :: *linorder*,*'a* :: *semiring-1*)*poly* **where**
  one-poly ≡ [(*1*,*1*)]

lemma *one-poly-inv*: *poly-inv one-poly* ⟨*proof*⟩

lemma *poly-one*[*simp*]: *eval-poly* α *one-poly* = *1*
  ⟨*proof*⟩

lemma *poly-zero-add*: *poly-add zero-poly p* = *p* ⟨*proof*⟩

lemma *poly-zero-mult*: *poly-mult zero-poly p* = *zero-poly* ⟨*proof*⟩

equality of polynomials

definition *eq-poly* :: (*'v* :: *linorder*, *'a* :: *comm-semiring-1*)*poly* ⇒ (*'v*,*'a*)*poly* ⇒ *bool* (**infix** =*p* *51*)
where *p* =*p q* ≡ ∀ α. *eval-poly* α *p* = *eval-poly* α *q*

lemma *poly-one-mult*: *poly-mult one-poly p* =*p p*
  ⟨*proof*⟩

lemma *eq-poly-refl*[*simp*]: *p* =*p p* ⟨*proof*⟩

lemma *eq-poly-trans*[*trans*]: ⟦*p1* =*p p2*; *p2* =*p p3*⟧ ⟹ *p1* =*p p3*
⟨*proof*⟩

lemma *poly-add-comm*: *poly-add p q* =*p poly-add q p* ⟨*proof*⟩

lemma *poly-add-assoc*: *poly-add p1* (*poly-add p2 p3*) =*p poly-add* (*poly-add p1 p2*) *p3* ⟨*proof*⟩

lemma *poly-mult-comm*: *poly-mult p q =p poly-mult q p* ⟨*proof*⟩

lemma *poly-mult-assoc*: *poly-mult p1* (*poly-mult p2 p3*) =*p poly-mult* (*poly-mult p1 p2*) *p3* ⟨*proof*⟩

lemma *poly-distrib*: *poly-mult p* (*poly-add q1 q2*) =*p poly-add* (*poly-mult p q1*) (*poly-mult p q2*) ⟨*proof*⟩

18.3 Computing normal forms of polynomials

fun
poly-of :: ('*v* :: *linorder*,'*a* :: *comm-semiring-1*)*tpoly* ⇒ ('*v*,'*a*)*poly*
where *poly-of* (*PNum i*) = (*if i* = *0 then* [] *else* [(*1*,*i*)])
| *poly-of* (*PVar x*) = [(*var-monom x*,*1*)]
| *poly-of* (*PSum* []) = *zero-poly*
| *poly-of* (*PSum* (*p* # *ps*)) = (*poly-add* (*poly-of p*) (*poly-of* (*PSum ps*)))
| *poly-of* (*PMult* []) = *one-poly*
| *poly-of* (*PMult* (*p* # *ps*)) = (*poly-mult* (*poly-of p*) (*poly-of* (*PMult ps*)))

evaluation is preserved by poly_of

lemma *poly-of*: *eval-poly α* (*poly-of p*) = *eval-tpoly α p*
⟨*proof*⟩

poly_of only generates polynomials that satisfy the invariant

lemma *poly-of-inv*: *poly-inv* (*poly-of p*)
⟨*proof*⟩

18.4 Powers and substitutions of polynomials

fun *poly-power* :: ('*v* :: *linorder*, '*a* :: *comm-semiring-1*)*poly* ⇒ *nat* ⇒ ('*v*,'*a*)*poly*
where
poly-power - 0 = *one-poly*
| *poly-power p* (*Suc n*) = *poly-mult p* (*poly-power p n*)

lemma *poly-power*[*simp*]: *eval-poly α* (*poly-power p n*) = (*eval-poly α p*) ^ *n*
⟨*proof*⟩

lemma *poly-power-inv*: **assumes** *p*: *poly-inv p*
shows *poly-inv* (*poly-power p n*)
⟨*proof*⟩

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

fun *monom-list-subst* :: ('*v* ⇒ ('*w* :: *linorder*,'*a* :: *comm-semiring-1*)*poly*) ⇒ '*v*
monom-list ⇒ ('*w*,'*a*)*poly* **where**
monom-list-subst σ [] = *one-poly*
| *monom-list-subst σ* ((*x*,*p*) # *m*) = *poly-mult* (*poly-power* (*σ x*) *p*) (*monom-list-subst σ m*)

lift-definition *monom-list* :: *'v* :: *linorder monom* ⇒ *'v monom-list* **is** λ *x*. *x*
⟨*proof*⟩

definition *monom-subst* :: (*'v* :: *linorder* ⇒ (*'w* :: *linorder*,*'a* :: *comm-semiring-1*)*poly*)
⇒ *'v monom* ⇒ (*'w*,*'a*)*poly* **where**
monom-subst σ *m* = *monom-list-subst* σ (*monom-list m*)

lemma *monom-list-subst-inv*: **assumes** *sub*: ⋀ *x*. *poly-inv* (σ *x*)
shows *poly-inv* (*monom-list-subst* σ *m*)
⟨*proof*⟩

lemma *monom-subst-inv*: **assumes** *sub*: ⋀ *x*. *poly-inv* (σ *x*)
shows *poly-inv* (*monom-subst* σ *m*)
⟨*proof*⟩

lemma *monom-subst*[*simp*]: *eval-poly* α (*monom-subst* σ *m*) = *eval-monom* (λ *v*.
eval-poly α (σ *v*)) *m*
⟨*proof*⟩

fun *poly-subst* :: (*'v* :: *linorder* ⇒ (*'w* :: *linorder*,*'a* :: *comm-semiring-1*)*poly*) ⇒
(*'v*,*'a*)*poly* ⇒ (*'w*,*'a*)*poly* **where**
poly-subst σ [] = *zero-poly*
| *poly-subst* σ ((*m*,*c*) # *p*) = *poly-add* (*poly-mult* [(*1*,*c*)] (*monom-subst* σ *m*))
(*poly-subst* σ *p*)

lemma *poly-subst-inv*: **assumes** *sub*: ⋀ *x*. *poly-inv* (σ *x*) **and** *p*: *poly-inv p*
shows *poly-inv* (*poly-subst* σ *p*)
⟨*proof*⟩

lemma *poly-subst*: *eval-poly* α (*poly-subst* σ *p*) = *eval-poly* (λ *v*. *eval-poly* α (σ
v)) *p*
⟨*proof*⟩

lemma *eval-poly-subst*:
assumes *eq*: ⋀ *w*. *f w* = *eval-poly g* (*q w*)
shows *eval-poly f p* = *eval-poly g* (*poly-subst q p*)
⟨*proof*⟩

lift-definition *monom-vars-list* :: *'v* :: *linorder monom* ⇒ *'v list* **is** *map fst* ⟨*proof*⟩

lemma *monom-vars-list-subst*: **assumes** ⋀ *w*. *w* ∈ *set* (*monom-vars-list m*) ⟹
f w = *g w*
shows *monom-subst f m* = *monom-subst g m*
⟨*proof*⟩

lemma *eval-monom-vars-list*: **assumes** ⋀ *x*. *x* ∈ *set* (*monom-vars-list xs*) ⟹ α
x = β *x*
shows *eval-monom* α *xs* = *eval-monom* β *xs* ⟨*proof*⟩

definition *monom-vars* **where** *monom-vars m = set (monom-vars-list m)*

lemma *monom-vars-list-1*[*simp*]: *monom-vars-list 1 = []*
⟨*proof*⟩

lemma *monom-vars-list-var-monom*[*simp*]: *monom-vars-list (var-monom x) = [x]*

⟨*proof*⟩

lemma *monom-vars-eval-monom*:
(⋀ x. x ∈ monom-vars m ⟹ f x = g x) ⟹ eval-monom f m = eval-monom g m
⟨*proof*⟩

definition *poly-vars-list* :: *('v :: linorder,'a)poly ⇒ 'v list* **where**
poly-vars-list p = remdups (concat (map (monom-vars-list o fst) p))

definition *poly-vars* :: *('v :: linorder,'a)poly ⇒ 'v set* **where**
poly-vars p = set (concat (map (monom-vars-list o fst) p))

lemma *poly-vars-list*[*simp*]: *set (poly-vars-list p) = poly-vars p*
⟨*proof*⟩

lemma *poly-vars*: **assumes** *eq*: *⋀ w. w ∈ poly-vars p ⟹ f w = g w*
shows *poly-subst f p = poly-subst g p*
⟨*proof*⟩

lemma *poly-var*: **assumes** *pv*: *v ∉ poly-vars p* **and** *diff*: *⋀ w. v ≠ w ⟹ f w = g w*
shows *poly-subst f p = poly-subst g p*
⟨*proof*⟩

lemma *eval-poly-vars*: **assumes** *⋀ x. x ∈ poly-vars p ⟹ α x = β x*
shows *eval-poly α p = eval-poly β p*
⟨*proof*⟩

declare *poly-subst.simps*[*simp del*]

18.5 Polynomial orders

definition *pos-assign* :: *('v,'a :: ordered-semiring-0)assign ⇒ bool*
where *pos-assign α = (∀ x. α x ≥ 0)*

definition *poly-ge* :: *('v :: linorder,'a :: poly-carrier)poly ⇒ ('v,'a)poly ⇒ bool* (**infix** *≥p 51*)

where $p \geq_p q = (\forall\ \alpha.\ \textit{pos-assign}\ \alpha \longrightarrow \textit{eval-poly}\ \alpha\ p \geq \textit{eval-poly}\ \alpha\ q)$

lemma *poly-ge-refl*[*simp*]: $p \geq_p p$
⟨*proof*⟩

lemma *poly-ge-trans*[*trans*]: $[\![p1 \geq_p p2;\ p2 \geq_p p3]\!] \Longrightarrow p1 \geq_p p3$
⟨*proof*⟩

lemma *pos-assign-monom-list*: **fixes** $\alpha :: ('v :: \textit{linorder}, 'a :: \textit{poly-carrier})\textit{assign}$
assumes *pos*: *pos-assign* α
shows *eval-monom-list* $\alpha\ m \geq 0$
⟨*proof*⟩

lemma *pos-assign-monom*: **fixes** $\alpha :: ('v :: \textit{linorder}, 'a :: \textit{poly-carrier})\textit{assign}$
assumes *pos*: *pos-assign* α
shows *eval-monom* $\alpha\ m \geq 0$
⟨*proof*⟩

lemma *pos-assign-poly*: **assumes** *pos*: *pos-assign* α
and *p*: $p \geq_p \textit{zero-poly}$
shows *eval-poly* $\alpha\ p \geq 0$
⟨*proof*⟩

lemma *poly-add-ge-mono*: **assumes** $p1 \geq_p p2$ **shows** *poly-add p1 q* $\geq_p$ *poly-add p2 q*
⟨*proof*⟩

lemma *poly-mult-ge-mono*: **assumes** $p1 \geq_p p2$ **and** $q \geq_p \textit{zero-poly}$
shows *poly-mult p1 q* $\geq_p$ *poly-mult p2 q*
⟨*proof*⟩

context *poly-order-carrier*
begin

definition *poly-gt* :: $('v :: \textit{linorder}, 'a)\textit{poly} \Rightarrow ('v, 'a)\textit{poly} \Rightarrow \textit{bool}$ (**infix** $>_p$ *51*)
where $p >_p q = (\forall\ \alpha.\ \textit{pos-assign}\ \alpha \longrightarrow \textit{eval-poly}\ \alpha\ p \succ \textit{eval-poly}\ \alpha\ q)$

lemma *poly-gt-imp-poly-ge*: $p >_p q \Longrightarrow p \geq_p q$ ⟨*proof*⟩

abbreviation *poly-GT* :: $('v :: \textit{linorder}, 'a)\textit{poly rel}$
where $\textit{poly-GT} \equiv \{(p,q) \mid p\ q.\ p >_p q \wedge q \geq_p \textit{zero-poly}\}$

lemma *poly-compat*: $[\![p1 \geq_p p2;\ p2 >_p p3]\!] \Longrightarrow p1 >_p p3$
⟨*proof*⟩

lemma *poly-compat2*: $[\![p1 >_p p2;\ p2 \geq_p p3]\!] \Longrightarrow p1 >_p p3$

⟨*proof*⟩

lemma *poly-gt-trans*[*trans*]: ⟦*p1* >*p* *p2*; *p2* >*p* *p3*⟧ ⟹ *p1* >*p* *p3*
⟨*proof*⟩

lemma *poly-GT-SN*: *SN poly-GT*
⟨*proof*⟩
end

monotonicity of polynomials

lemma *eval-monom-list-mono*: **assumes** *fg*: ⋀ *x*. (*f* :: (*'v* :: *linorder*,*'a* :: *poly-carrier*)*assign*) *x* ≥ *g x*
and *g*: ⋀ *x*. *g x* ≥ *0*
shows *eval-monom-list f m* ≥ *eval-monom-list g m eval-monom-list g m* ≥ *0*
⟨*proof*⟩

lemma *eval-monom-mono*: **assumes** *fg*: ⋀ *x*. (*f* :: (*'v* :: *linorder*,*'a* :: *poly-carrier*)*assign*) *x* ≥ *g x*
and *g*: ⋀ *x*. *g x* ≥ *0*
shows *eval-monom f m* ≥ *eval-monom g m eval-monom g m* ≥ *0*
⟨*proof*⟩

definition *poly-weak-mono-all* :: (*'v* :: *linorder*,*'a* :: *poly-carrier*)*poly* ⇒ *bool* **where**

poly-weak-mono-all p ≡ ∀ (*α* :: (*'v*,*'a*)*assign*) *β*. (∀ *x*. *α x* ≥ *β x*)
⟶ *pos-assign β* ⟶ *eval-poly α p* ≥ *eval-poly β p*

lemma *poly-weak-mono-all-E*: **assumes** *p*: *poly-weak-mono-all p* **and**
ge: ⋀ *x*. *f x* ≥*p* *g x* ∧ *g x* ≥*p* *zero-poly*
shows *poly-subst f p* ≥*p* *poly-subst g p*
⟨*proof*⟩

definition *poly-weak-mono* :: (*'v* :: *linorder*,*'a* :: *poly-carrier*)*poly* ⇒ *'v* ⇒ *bool* **where**
poly-weak-mono p v ≡ ∀ (*α* :: (*'v*,*'a*)*assign*) *β*. (∀ *x*. *v* ≠ *x* ⟶ *α x* = *β x*) ⟶ *pos-assign β* ⟶ *α v* ≥ *β v* ⟶ *eval-poly α p* ≥ *eval-poly β p*

lemma *poly-weak-mono-E*: **assumes** *p*: *poly-weak-mono p v*
and *fgw*: ⋀ *w*. *v* ≠ *w* ⟹ *f w* = *g w*
and *g*: ⋀ *w*. *g w* ≥*p* *zero-poly*
and *fgv*: *f v* ≥*p* *g v*
shows *poly-subst f p* ≥*p* *poly-subst g p*
⟨*proof*⟩

definition *poly-weak-anti-mono* :: (*'v* :: *linorder*,*'a* :: *poly-carrier*)*poly* ⇒ *'v* ⇒ *bool* **where**
poly-weak-anti-mono p v ≡ ∀ (*α* :: (*'v*,*'a*)*assign*) *β*. (∀ *x*. *v* ≠ *x* ⟶ *α x* = *β x*) ⟶ *pos-assign β* ⟶ *α v* ≥ *β v* ⟶ *eval-poly β p* ≥ *eval-poly α p*

lemma *poly-weak-anti-mono-E*: **assumes** *p*: *poly-weak-anti-mono p v*
and *fgw*: $\bigwedge w.\ v \neq w \Longrightarrow f\ w = g\ w$
and *g*: $\bigwedge w.\ g\ w \geq_p zero\text{-}poly$
and *fgv*: $f\ v \geq_p g\ v$
shows *poly-subst g p* $\geq_p$ *poly-subst f p*
⟨*proof*⟩

lemma *poly-weak-mono*: **fixes** *p* :: (*'v* :: *linorder*,*'a* :: *poly-carrier*)*poly*
assumes *mono*: $\bigwedge v.\ v \in poly\text{-}vars\ p \Longrightarrow poly\text{-}weak\text{-}mono\ p\ v$
shows *poly-weak-mono-all p*
⟨*proof*⟩

lemma *poly-weak-mono-all*: **fixes** *p* :: (*'v* :: *linorder*,*'a* :: *poly-carrier*)*poly*
assumes *p*: *poly-weak-mono-all p*
shows *poly-weak-mono p v*
⟨*proof*⟩

lemma *poly-weak-mono-all-pos*:
fixes *p* :: (*'v* :: *linorder*,*'a* :: *poly-carrier*)*poly*
assumes *pos-at-zero*: *eval-poly* (λ *w*. *0*) *p* $\geq$ *0*
and *mono*: *poly-weak-mono-all p*
shows *p* $\geq_p$ *zero-poly*
⟨*proof*⟩

context *poly-order-carrier*
begin

definition *poly-strict-mono* :: (*'v* :: *linorder*,*'a*)*poly* $\Rightarrow$ *'v* $\Rightarrow$ *bool* **where**
poly-strict-mono p v $\equiv \forall$ (α :: (*'v*,*'a*)*assign*) β. ($\forall$ *x*. (*v* $\neq$ *x* $\longrightarrow$ α *x* = β *x*)) $\longrightarrow$ *pos-assign* β $\longrightarrow$ α *v* $\succ$ β *v* $\longrightarrow$ *eval-poly* α *p* $\succ$ *eval-poly* β *p*

lemma *poly-strict-mono-E*: **assumes** *p*: *poly-strict-mono p v*
and *fgw*: $\bigwedge w.\ v \neq w \Longrightarrow f\ w = g\ w$
and *g*: $\bigwedge w.\ g\ w \geq_p zero\text{-}poly$
and *fgv*: $f\ v >_p g\ v$
shows *poly-subst f p* $>_p$ *poly-subst g p*
⟨*proof*⟩

lemma *poly-add-gt-mono*: **assumes** *p1* $>_p$ *p2* **shows** *poly-add p1 q* $>_p$ *poly-add p2 q*
⟨*proof*⟩

lemma *poly-mult-gt-mono*:
fixes *q* :: (*'v* :: *linorder*,*'a*)*poly*
assumes *gt*: *p1* $>_p$ *p2* **and** *mono*: *q* $\geq_p$ *one-poly*
shows *poly-mult p1 q* $>_p$ *poly-mult p2 q*
⟨*proof*⟩
end

18.6 Degree of polynomials

definition *monom-list-degree* :: *'v monom-list* ⇒ *nat* **where**
monom-list-degree xps ≡ *sum-list* (*map snd xps*)

lift-definition *monom-degree* :: *'v* :: *linorder monom* ⇒ *nat* **is** *monom-list-degree*
⟨*proof*⟩

definition *poly-degree* :: (-,*'a*) *poly* ⇒ *nat* **where**
poly-degree p ≡ *max-list* (*map* (λ (*m*,*c*). *monom-degree m*) *p*)

definition *poly-coeff-sum* :: (*'v*,*'a* :: *ordered-ab-semigroup*) *poly* ⇒ *'a* **where**
poly-coeff-sum p ≡ *sum-list* (*map* (λ *mc*. *max 0* (*snd mc*)) *p*)

lemma *monom-list-degree*: *eval-monom-list* (λ -. *x*) *m* = *x* ^ *monom-list-degree m*
⟨*proof*⟩

lemma *monom-list-var-monom*[*simp*]: *monom-list* (*var-monom x*) = [(*x*,*1*)]
⟨*proof*⟩

lemma *monom-list-1*[*simp*]: *monom-list 1* = []
⟨*proof*⟩

lemma *monom-degree*: *eval-monom* (λ -. *x*) *m* = *x* ^ *monom-degree m*
⟨*proof*⟩

lemma *poly-coeff-sum*: *poly-coeff-sum p* ≥ *0*
⟨*proof*⟩

lemma *poly-degree*: **assumes** *x*: *x* ≥ (*1* :: *'a* :: *poly-carrier*)
shows *poly-coeff-sum p* ∗ (*x* ^ *poly-degree p*) ≥ *eval-poly* (λ -. *x*) *p*
⟨*proof*⟩

lemma *poly-degree-bound*: **assumes** *x*: *x* ≥ (*1* :: *'a* :: *poly-carrier*)
and *c*: *c* ≥ *poly-coeff-sum p*
and *d*: *d* ≥ *poly-degree p*
shows *c* ∗ (*x* ^ *d*) ≥ *eval-poly* (λ -. *x*) *p*
⟨*proof*⟩

18.7 Executable and sufficient criteria to compare polynomials and ensure monotonicity

poly_split extracts the coefficient for a given monomial and returns additionally the remaining polynomial

definition *poly-split* :: (*'v monom*) ⇒ (*'v*,*'a* :: *zero*)*poly* ⇒ *'a* × (*'v*,*'a*)*poly*
where *poly-split m p* ≡ *case List.extract* (λ (*n*,-). *m* = *n*) *p of None* ⇒ (*0*,*p*) |
Some (*p1*,(-,*c*),*p2*) ⇒ (*c*, *p1* @ *p2*)

lemma *poly-split*: **assumes** *poly-split m p* = (*c*,*q*)

shows $p =p\ (m,c)\ \#\ q$
⟨*proof*⟩

lemma *poly-split-eval*: **assumes** *poly-split m p* = (*c*,*q*)
shows *eval-poly α p* = (*eval-monom α m* ∗ *c*) + *eval-poly α q*
⟨*proof*⟩

fun *check-poly-eq* :: (′*v*,′*a* :: *semiring-0*)*poly* ⇒ (′*v*,′*a*)*poly* ⇒ *bool* **where**
check-poly-eq [] *q* = (*q* = [])
| *check-poly-eq* ((*m*,*c*) # *p*) *q* = (*case List.extract* (λ *nd*. *fst nd* = *m*) *q of*
None ⇒ *False*
| *Some* (*q1*,(-,*d*),*q2*) ⇒ *c* = *d* ∧ *check-poly-eq p* (*q1* @ *q2*))

lemma *check-poly-eq*: **fixes** *p* :: (′*v* :: *linorder*,′*a* :: *poly-carrier*)*poly*
assumes *chk*: *check-poly-eq p q*
shows *p* =*p q* ⟨*proof*⟩

declare *check-poly-eq.simps*[*simp del*]

fun *check-poly-ge* :: (′*v*,′*a* :: *ordered-semiring-0*)*poly* ⇒ (′*v*,′*a*)*poly* ⇒ *bool* **where**

check-poly-ge [] *q* = *list-all* (λ (-,*d*). *0* ≥ *d*) *q*
| *check-poly-ge* ((*m*,*c*) # *p*) *q* = (*case List.extract* (λ *nd*. *fst nd* = *m*) *q of*
None ⇒ *c* ≥ *0* ∧ *check-poly-ge p q*
| *Some* (*q1*,(-,*d*),*q2*) ⇒ *c* ≥ *d* ∧ *check-poly-ge p* (*q1* @ *q2*))

lemma *check-poly-ge*: **fixes** *p* :: (′*v* :: *linorder*,′*a* :: *poly-carrier*)*poly*
shows *check-poly-ge p q* ⟹ *p* ≥*p q*
⟨*proof*⟩

declare *check-poly-ge.simps*[*simp del*]

definition *check-poly-weak-mono-all* :: (′*v*,′*a* :: *ordered-semiring-0*)*poly* ⇒ *bool*
where *check-poly-weak-mono-all p* ≡ *list-all* (λ (*m*,*c*). *c* ≥ *0*) *p*

lemma *check-poly-weak-mono-all*: **fixes** *p* :: (′*v* :: *linorder*,′*a* :: *poly-carrier*)*poly*
assumes *check-poly-weak-mono-all p* **shows** *poly-weak-mono-all p*
⟨*proof*⟩

lemma *check-poly-weak-mono-all-pos*:
assumes *check-poly-weak-mono-all p* **shows** *p* ≥*p zero-poly*
⟨*proof*⟩

better check for weak monotonicity for discrete carriers: p is monotone in v if $p(\ldots v+1\ldots) \geq p(\ldots v\ldots)$

definition *check-poly-weak-mono-discrete* :: (′*v* :: *linorder*,′*a* :: *poly-carrier*)*poly*
⇒ ′*v* ⇒ *bool*

where *check-poly-weak-mono-discrete p v ≡ check-poly-ge (poly-subst (λ w. poly-of (if w = v then PSum [PNum 1, PVar v] else PVar w)) p) p*

definition *check-poly-weak-mono-and-pos* :: *bool* ⇒ (*'v* :: *linorder*,*'a* :: *poly-carrier*)*poly* ⇒ *bool*
where *check-poly-weak-mono-and-pos discrete p ≡*
if discrete then list-all (λ v. check-poly-weak-mono-discrete p v) (poly-vars-list p) ∧ eval-poly (λ w. 0) p ≥ 0
else check-poly-weak-mono-all p

definition *check-poly-weak-anti-mono-discrete* :: (*'v* :: *linorder*,*'a* :: *poly-carrier*)*poly* ⇒ *'v* ⇒ *bool*
where *check-poly-weak-anti-mono-discrete p v ≡ check-poly-ge p (poly-subst (λ w. poly-of (if w = v then PSum [PNum 1, PVar v] else PVar w)) p)*

context *poly-order-carrier*
begin

lemma *check-poly-weak-mono-discrete*:
fixes *v* :: *'v* :: *linorder* **and** *p* :: (*'v*,*'a*)*poly*
assumes *discrete* **and** *check*: *check-poly-weak-mono-discrete p v*
shows *poly-weak-mono p v*
⟨*proof*⟩

lemma *check-poly-weak-anti-mono-discrete*:
fixes *v* :: *'v* :: *linorder* **and** *p* :: (*'v*,*'a*)*poly*
assumes *discrete* **and** *check*: *check-poly-weak-anti-mono-discrete p v*
shows *poly-weak-anti-mono p v*
⟨*proof*⟩

lemma *check-poly-weak-mono-and-pos*:
fixes *p* :: (*'v* :: *linorder*,*'a*)*poly*
assumes *check-poly-weak-mono-and-pos discrete p*
shows *poly-weak-mono-all p ∧ (p ≥p zero-poly)*
⟨*proof*⟩

end

definition *check-poly-weak-mono* :: (*'v* :: *linorder*,*'a* :: *ordered-semiring-0*)*poly* ⇒ *'v* ⇒ *bool*
where *check-poly-weak-mono p v ≡ list-all (λ (m,c). c ≥ 0 ∨ v ∉ monom-vars m) p*

lemma *check-poly-weak-mono*: **fixes** *p* :: (*'v* :: *linorder*,*'a* :: *poly-carrier*)*poly*
assumes *check-poly-weak-mono p v* **shows** *poly-weak-mono p v*
⟨*proof*⟩

definition *check-poly-weak-mono-smart* :: *bool* ⇒ (*'v* :: *linorder*,*'a* :: *poly-carrier*)*poly* ⇒ *'v* ⇒ *bool*

where *check-poly-weak-mono-smart discrete* ≡ *if discrete then check-poly-weak-mono-discrete else check-poly-weak-mono*

lemma (**in** *poly-order-carrier*) *check-poly-weak-mono-smart*: **fixes** *p* :: (*'v* :: *linorder*,*'a* :: *poly-carrier*)*poly*
shows *check-poly-weak-mono-smart discrete p v* ⟹ *poly-weak-mono p v*
⟨*proof*⟩

definition *check-poly-weak-anti-mono* :: (*'v* :: *linorder*,*'a* :: *ordered-semiring-0*)*poly* ⇒ *'v* ⇒ *bool*
where *check-poly-weak-anti-mono p v* ≡ *list-all* (λ (*m*,*c*). *0* ≥ *c* ∨ *v* ∉ *monom-vars m*) *p*

lemma *check-poly-weak-anti-mono*: **fixes** *p* :: (*'v* :: *linorder*,*'a* :: *poly-carrier*)*poly*
assumes *check-poly-weak-anti-mono p v* **shows** *poly-weak-anti-mono p v*
⟨*proof*⟩

definition *check-poly-weak-anti-mono-smart* :: *bool* ⇒ (*'v* :: *linorder*,*'a* :: *poly-carrier*)*poly* ⇒ *'v* ⇒ *bool*
where *check-poly-weak-anti-mono-smart discrete* ≡ *if discrete then check-poly-weak-anti-mono-discrete else check-poly-weak-anti-mono*

lemma (**in** *poly-order-carrier*) *check-poly-weak-anti-mono-smart*: **fixes** *p* :: (*'v* :: *linorder*,*'a* :: *poly-carrier*)*poly*
shows *check-poly-weak-anti-mono-smart discrete p v* ⟹ *poly-weak-anti-mono p v*
⟨*proof*⟩

definition *check-poly-gt* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'v* :: *linorder*,*'a* :: *ordered-semiring-0*)*poly* ⇒ (*'v*,*'a*)*poly* ⇒ *bool*
where *check-poly-gt gt p q* ≡ *let* (*a1*,*p1*) = *poly-split 1 p*; (*b1*,*q1*) = *poly-split 1 q in gt a1 b1* ∧ *check-poly-ge p1 q1*

fun *univariate-power-list* :: *'v* ⇒ *'v monom-list* ⇒ *nat option* **where**
univariate-power-list x [(*y*,*n*)] = (*if x* = *y then Some n else None*)
| *univariate-power-list - - = None*

lemma *univariate-power-list*: **assumes** *monom-inv m univariate-power-list x m* = *Some n*
shows *sum-var-list m* = (λ *y*. *if x* = *y then n else 0*)
eval-monom-list α m = ((*α x*) ^*n*)
n ≥ *1*
⟨*proof*⟩

lift-definition *univariate-power* :: *'v* :: *linorder* ⇒ *'v monom* ⇒ *nat option*
is *univariate-power-list* ⟨*proof*⟩

lemma *univariate-power*: **assumes** *univariate-power x m* = *Some n*
shows *sum-var m* = (λ *y*. *if x* = *y then n else 0*)

eval-monom α *m* = ((α *x*) ^*n*)
n ≥ *1*
⟨*proof*⟩

lemma *univariate-power-var-monom*: *univariate-power y* (*var-monom x*) = (*if x* = *y then Some 1 else None*)
⟨*proof*⟩

definition *check-monom-strict-mono* :: *bool* ⇒ *'v* :: *linorder monom* ⇒ *'v* ⇒ *bool*
where
check-monom-strict-mono pm m v ≡ *case univariate-power v m of*
Some p ⇒ *pm* ∨ *p* = *1*
| *None* ⇒ *False*

definition *check-poly-strict-mono* :: *bool* ⇒ (*'v* :: *linorder*, *'a* :: *poly-carrier*)*poly* ⇒ *'v* ⇒ *bool*
where *check-poly-strict-mono pm p v* ≡ *list-ex* (λ (*m*,*c*). (*c* ≥ *1*) ∧ *check-monom-strict-mono pm m v*) *p*

definition *check-poly-strict-mono-discrete* :: (*'a* :: *poly-carrier* ⇒ *'a* ⇒ *bool*) ⇒ (*'v* :: *linorder*,*'a*)*poly* ⇒ *'v* ⇒ *bool*
where *check-poly-strict-mono-discrete gt p v* ≡ *check-poly-gt gt* (*poly-subst* (λ *w*. *poly-of* (*if w* = *v then PSum* [*PNum 1*, *PVar v*] *else PVar w*)) *p*) *p*

definition *check-poly-strict-mono-smart* :: *bool* ⇒ *bool* ⇒ (*'a* :: *poly-carrier* ⇒ *'a* ⇒ *bool*) ⇒ (*'v* :: *linorder*,*'a*)*poly* ⇒ *'v* ⇒ *bool*
where *check-poly-strict-mono-smart discrete pm gt p v* ≡
if discrete then check-poly-strict-mono-discrete gt p v else check-poly-strict-mono pm p v

context *poly-order-carrier*
begin
lemma *check-monom-strict-mono*: **fixes** α β :: (*'v* :: *linorder*,*'a*)*assign* **and** *v* :: *'v* **and** *m* :: *'v monom*
assumes *check*: *check-monom-strict-mono power-mono m v*
and *gt*: α *v* ≻ β *v*
and *ge*: β *v* ≥ *0*
shows *eval-monom* α *m* ≻ *eval-monom* β *m*
⟨*proof*⟩

lemma *check-poly-strict-mono*:
assumes *check1*: *check-poly-strict-mono power-mono p v*
and *check2*: *check-poly-weak-mono-all p*
shows *poly-strict-mono p v*
⟨*proof*⟩

lemma *check-poly-gt*:
fixes *p* :: (*'v* :: *linorder*,*'a*)*poly*

assumes *check-poly-gt gt p q* **shows** *p >p q*
⟨*proof*⟩

lemma *check-poly-strict-mono-discrete*:
fixes *v* :: *'v* :: *linorder* **and** *p* :: (*'v*,*'a*)*poly*
assumes *discrete* **and** *check*: *check-poly-strict-mono-discrete gt p v*
shows *poly-strict-mono p v*
⟨*proof*⟩

lemma *check-poly-strict-mono-smart*:
assumes *check1*: *check-poly-strict-mono-smart discrete power-mono gt p v*
and *check2*: *check-poly-weak-mono-and-pos discrete p*
shows *poly-strict-mono p v*
⟨*proof*⟩

end

end

19 Displaying Polynomials

theory *Show-Polynomials*
imports
Polynomials
Show.Show-Instances
begin

fun *shows-monom-list* :: (*'v* :: {*linorder*,*show*})*monom-list* ⇒ *string* ⇒ *string*
where
shows-monom-list [(*x*,*p*)] = (*if p = 1 then shows x else shows x* +@+ *shows-string ''^''* +@+ *shows p*)
| *shows-monom-list* ((*x*,*p*) # *m*) = ((*if p = 1 then shows x else shows x* +@+ *shows-string ''^''* +@+ *shows p*) +@+ *shows-string ''*''* +@+ *shows-monom-list m*)
| *shows-monom-list* [] = *shows-string ''1''*

instantiation *monom* :: ({*linorder*,*show*}) *show*
begin

lift-definition *shows-prec-monom* :: *nat* ⇒ *'a monom* ⇒ *shows* **is** λ *n*. *shows-monom-list*
⟨*proof*⟩

lemma *shows-prec-monom-append* [*show-law-simps*]:
shows-prec d (*m* :: *'a monom*) (*r* @ *s*) = *shows-prec d m r* @ *s*
⟨*proof*⟩

definition *shows-list* (*ts* :: *'a monom list*) = *showsp-list shows-prec 0 ts*

instance ⟨*proof*⟩

end

fun *shows-poly* :: ('*v* :: {*show*,*linorder*},'*a* :: {*one*,*show*})*poly* ⇒ *string* ⇒ *string*
where
shows-poly [] = *shows-string* ''0''
| *shows-poly* ((*m*,*c*) # *p*) = ((*if c* = *1 then shows m else if m* = *1 then shows c else shows c* +@+
shows-string ''∗'' +@+ *shows m*) +@+ (*if p* = [] *then shows-string* [] *else shows-string* '' + '' +@+ *shows-poly p*))
end

20 Monotonicity criteria of Neurauter, Zankl, and Middeldorp

theory *NZM*
imports *Abstract−Rewriting.SN-Order-Carrier Polynomials*
begin

We show that our check on monotonicity is strong enough to capture the exact criterion for polynomials of degree 2 that is presented in [3]:

- $ax^2 + bx + c$ is monotone if $b + a > 0$ and $a \geq 0$
- $ax^2 + bx + c$ is weakly monotone if $b + a \geq 0$ and $a \geq 0$

lemma *var-monom-x-x* [*simp*]: *var-monom x* ∗ *var-monom x* ≠ *1*
⟨*proof*⟩

lemma *monom-list-x-x*[*simp*]: *monom-list* (*var-monom x* ∗ *var-monom x*) = [(*x*,*2*)]
⟨*proof*⟩

lemma assumes *b*: *b* + *a* > *0* **and** *a*: (*a* :: *int*) ≥ *0*
shows *check-poly-strict-mono-discrete* (>) (*poly-of* (*PSum* [*PNum c*, *PMult* [*PNum b*, *PVar x*], *PMult* [*PNum a*, *PVar x*, *PVar x*]])) *x*
⟨*proof*⟩

lemma assumes *b*: *b* + *a* ≥ *0* **and** *a*: (*a* :: *int*) ≥ *0*
shows *check-poly-weak-mono-discrete* (*poly-of* (*PSum* [*PNum c*, *PMult* [*PNum b*, *PVar x*], *PMult* [*PNum a*, *PVar x*, *PVar x*]])) *x*
⟨*proof*⟩

end

References

[1] D. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA, 1979.

[2] S. Lucas. Polynomials over the reals in proofs of termination: From theory to practice. *RAIRO Theoretical Informatics and Applications*, 39(3):547–586, 2005.

[3] F. Neurauter, H. Zankl, and A. Middeldorp. Monotonicity criteria for polynomial interpretations over the naturals. In *Proceedings of the 5th International Joint Conference on Automated Reasoning*, LNAI 6173, pages 502–517, 2010.

[4] L. Robbiano. On the Theory of Graded Structures. *Journal of Symbolic Computation*, 2:138–170, 1985.

[5] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proc. TPHOLs'09*, LNCS 5674, pages 452–468, 2009.