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 [4] which contains several termination techniques. The provided theories have been essential to formalize polynomial-interpretations [1, 2].
1 Utility Functions and Lemmas

theory Utility
imports Main
begin

1.1 Miscellaneous

lemma infinite-imp-elem: ¬ finite A ⇒ ∃ x. x ∈ A
  by (cases A = {}, auto)

lemma inf-pigeonhole-principle:
  assumes ∀ k::nat. ∃ i<n::nat. f k i
  shows ∃ i<n. ∀ k. ∃ k′≥k. f k′ i
proof −
  have nfin: ¬ finite (UNIV :: nat set) by auto
  have fin: finite ({i. i < n}) by auto
  from pigeonhole-infinite-rel[OF nfin fin] assms
  obtain i where: i < n and nfin: ¬ finite {a. f a i} by auto
  show ?thesis
proof (intro exI conjI, rule i, intro allI)
  fix k
  have finite {a. f a i ∧ a < k} by auto
  with nfin have ¬ finite ({a. f a i} − {a. f a i ∧ a < k}) by auto
  from infinite-imp-elem[OF this] obtain a where: f a i and a ≥ k by auto
  thus ∃ k′ ≥ k. f k′ i by force
qed

lemma map-upt-Suc: map f [0 ..< Suc n] = f 0 # map (λ i. f (Suc i)) [0 ..< n]
  by (induct n arbitrary: f, auto)

lemma map-upt-add: map f [0 ..< n + m] = map f [0 ..< n] @ map (λ i. f (i + n)) [0 ..< m]
proof (induct n arbitrary: f)
  case (Suc n f)
  have map f [0 ..< Suc n + m] = map f [0 ..< Suc (n+m)] by simp
  also have ... = f 0 # map (λ i. f (Suc i)) [0 ..< n + m] unfolding map-upt-Suc
  ..
  finally show ?case unfolding Suc map-upt-Suc by simp
qed simp

lemma map-upt-split: assumes i: i < n
  shows map f [0 ..< n] = map f [0 ..< i] @ f i # map (λ j. f (j + Suc i)) [0 ..< n − Suc i]
proof −
  from i have n = i + Suc 0 + (n − Suc i) by arith
  hence id: [0 ..< n] = [0 ..< i + Suc 0 + (n − Suc i)] by simp
  show ?thesis unfolding id
qed
unfolding map-upt-add by auto

qed

lemma all-Suc-conv:
(∀ i< Suc n. P i) ←→ P 0 ∧ (∃ i<n. P (Suc i)) (is ?l = ?r)

proof
  assume ?l thus ?r by auto
next
  assume ?r show ?l
  proof (intro allI impI)
    fix i
    assume i < Suc n
    with ⟨?r⟩ show P i by (cases i, auto)
  qed
qed

lemma ex-Suc-conv:
(∃ i< Suc n. P i) ←→ P 0 ∨ (∃ i<n. P (Suc i)) (is ?l = ?r)

using all-Suc-conv[of n λi. ¬ P i] by blast

fun sorted-list-subset :: 'a :: linorder list ⇒ 'a list ⇒ 'a option where
  sorted-list-subset (a # as) (b # bs) =
  (if a = b then sorted-list-subset as bs
   else if a > b then sorted-list-subset (a # as) bs
   else Some a)
  | sorted-list-subset [] - = None
  | sorted-list-subset (a # -) [] = Some a

lemma sorted-list-subset:
  assumes sorted as and sorted bs
  shows (sorted-list-subset as bs = None) = (set as ⊆ set bs)

using assms

proof (induct rule: sorted-list-subset.induct)
  case (2 bs)
  thus ?case by auto
next
  case (3 a as)
  thus ?case by auto
next
  case (1 a as b bs)
  from 1(3) have sas; sorted as and a: \( a' \in \text{set as} \implies a \leq a' \)
  unfolding linorder-class.sorted.simps[of a # as] by auto
  from 1(4) have sbs; sorted bs and b: \( b' \in \text{set bs} \implies b \leq b' \)
  unfolding linorder-class.sorted.simps[of b # bs] by auto
  show ?case
  proof (cases a = b)
    case True
    from 1(1)[OF this sas 1(4)] True show ?thesis by auto
  next

3
case False note oFalse = this
show ?thesis 
proof (cases a > b)
case True
with a b have b \notin set as by force
with 1(2)[OF False True 1(3) sbs] False True show ?thesis by auto
next
case False
with oFalse have a < b by auto
with a b have a \notin set bs by force
with oFalse False show ?thesis by auto
qed
qed


lemma zip-nth-conv: length xs = length ys \implies zip xs ys = map (\lambda i. (xs ! i, ys ! i)) [0 ..< length ys]
proof (induct xs arbitrary: ys, simp)
case (Cons x xs)
then obtain y yys where ys: ys = y # yys by (cases ys, auto)
with Cons have len: length xs = length yys by simp
show ?case unfolding ys 
  by (simp del: upt-Suc add: map-upt-Suc, unfold Cons(1)[OF len], simp)
qed


lemma nth-map-conv:
  assumes length xs = length ys
  and \forall i<length xs. f (xs ! i) = g (ys ! i)
  shows map f xs = map g ys
using assms
proof (induct xs arbitrary: ys)
case (Cons x xs) thus ?case
proof (induct ys)
case (Cons y ys)
have \forall i<length xs. f (xs ! i) = g (ys ! i)
proof (intro allI impI)
  fix i assume i < length xs thus f (xs ! i) = g (ys ! i) using Cons(4) by force
qed
with Cons show ?case by auto
qed simp


lemma listsum-0: \[x. x \in \text{set} xs \implies x = 0\] \implies listsum xs = 0
by (induct xs, auto)


lemma foldr-foldr-concat: foldr (foldr f) m a = foldr f (concat m) a
proof (induct m arbitrary: a)
case Nil show ?case by simp
lemma listsum-double-concat:
  fixes f :: 'b ⇒ 'c ⇒ 'a :: comm-monoid-add and g as bs
  shows listsum (concat (map (λ i. map (λ j. f i j + g i j) as) bs))
    = listsum (concat (map (λ i. map (λ j. f i j) as) bs)) +
      listsum (concat (map (λ i. map (λ j. g i j) as) bs))
proof (induct bs)
  case Nil thus ?case by simp
next
  case (Cons b bs)
  have id: (∑ j← as. f b j + g b j) = listsum (map (f b) as) + listsum (map (g b) as)
    by (induct as, auto simp: ac-simps)
  show ?case unfolding list.map concat.simps listsum-append
    unfolding Cons
    unfolding id
    by (simp add: ac-simps)
qed

fun max-list :: nat list ⇒ nat where
  max-list [] = 0
| max-list (x # xs) = max x (max-list xs)

lemma max-list: x ∈ set xs ⇒ x ≤ max-list xs
  by (induct xs) auto
lemma max-list-mem: xs ≠ [] ⇒ max-list xs ∈ set xs
proof (induct xs)
  case (Cons x xs)
  show ?case
    proof (cases x ≥ max-list xs)
      case True
      thus ?thesis by auto
    next
      case False
      hence max: max-list xs > x by auto
      hence nil: xs ≠ [] by (cases xs, auto)
      from max have max: max x (max-list xs) = max-list xs by auto
      from Cons(1)[OF nil] max show ?thesis by auto
    qed
    qed simp

lemma max-list-set: max-list xs = (if set xs = {} then 0 else (THE x. x ∈ set xs)
∀ (y ∈ set xs. y ≤ x))
proof (cases xs = [])
  case True thus ?thesis by simp
next
case False
note p = max-list-mem[OF this] max-list[of - xs]
from False have id: (set xs = {}) = False by simp
show ?thesis unfolding id if-False
proof (rule the-equality[symmetric], intro conjI ballI, rule p, rule p)
  fix x
  assume x ∈ set xs ∧ (∀ y ∈ set xs. y ≤ x)
  hence mem: x ∈ set xs and le: ∀ y. y ∈ set xs ⇒ y ≤ x by auto
  from max-list[of mem] le[of max-list-mem[OF False]]
  show x = max-list xs by simp
qed
qed

lemma max-list-eq-set: set xs = set ys ⇒ max-list xs = max-list ys
unfolding max-list-set by simp

end

2 Polynomials

theory Polynomial
imports
../Abstract−Rewriting/SN-Orders
../Matrix/Utility
begin

2.1 Polynomials represented as trees

datatype ('v,'a)tpoly = PVar 'v | PNum 'a | PSum ('v,'a)tpoly list | PMult ('v,'a)tpoly list

type-synonym ('v,'a)assign = 'v ⇒ 'a

fun eval-tpoly :: ('v,'a :: semiring-1)assign ⇒ ('v,'a)tpoly ⇒ 'a
where eval-tpoly α (PVar x) = α x
  | eval-tpoly α (PNum a) = a
  | eval-tpoly α (PSum []) = 0
  | eval-tpoly α (PSum (p # ps)) = eval-tpoly α p + eval-tpoly α (PSum ps)
  | eval-tpoly α (PMult []) = 1
  | eval-tpoly α (PMult (p # ps)) = eval-tpoly α p * eval-tpoly α (PMult ps)
2.2 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 = ('v × nat)list

- [(x, n), (y, m)] represent \( x^n \cdot y^m \)
- invariants: all powers are \( \geq 1 \) 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)]

definition monom-inv :: 'v monom ⇒ bool
where monom-inv m ≡ (∀ (x,n) ∈ set m. 1 ≤ n) ∧ distinct (map fst m)

fun eval-monom :: ('v,'a :: comm-semiring-1)assign ⇒ ('v monom) ⇒ 'a
where eval-monom α [] = 1
    | eval-monom α ((x,p) # m) = eval-monom α m * (α x)^p

lemma eval-monom-list[simp]: eval-monom α (m @ n) = eval-monom α m * eval-monom α n
  by (induct m, auto simp: field-simps)

equality of monomials should be able to identify \( x^2 \cdot y \) with \( y \cdot x^2 \), essentially, it checks for permutations. Checking of permutations suffices, since in \( x^2 \cdot x^1 = x^3 \), the left-hand side should not be constructed due to the invariant, that every variable occurs at most once in a monomial.

fun eq-monom :: ('v monom ⇒ 'v monom ⇒ bool (infix =m 51)) where
  [] =m n = (n = [])
  | (x,p) # m =m n =
      (case List.extract (λyq. fst yq = x) n of
          None ⇒ False
        | Some (n1,(-,q),n2) ⇒ p = q ∧ m =m (n1 @ n2))

lemma eq-monom-refl: m =m m
proof (induct m)
  case (Cons xp m)
  show ?case
  proof (cases xp)
    case (Pair x p)
    show ?thesis
      by (simp add: Pair Cons extract-Cons-code)
  qed
qed simp

definition sum-var-list :: 'v monom ⇒ 'v ⇒ nat
where sum-var-list m x ≡ listsum (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
using assms
proof (induct m arbitrary: n)
case Nil
show ?thesis by simp
next
case (Cons xp m)
obtain x p where xp: xp = (x,p) by (cases xp, auto)
with Cons(2) have p: 0 < p and x: x ∉ fst ' set m and m: monom-inv m
unfolding monom-inv-def by auto
show ?thesis by (simp add: Cons, rule exI[of - y], simp add: sum-var-list-def yp p)
qed simp

next
case (Cons xp m)
obtain x p where xp: xp = (x,p) by (cases xp, auto)
with Cons(2) have p: 0 < p and x: x ∉ fst ' set m and m: monom-inv m
unfolding monom-inv-def by auto
show ?thesis by (cases xp, auto)
proof
(cases List.extract (λ yq. fst yq = x) n)
case None
hence not1: x ∉ fst ' set n by (auto simp: extract-None-iff)
show ?thesis by (simp add: xp Cons None, rule exI[of - x], simp add: sum-var-list-not[OF not1], simp add: sum-var-list-def p)
next
case (Some res)
obtain n1 yq n2 where res = (n1,yq,n2) by (cases res, auto)
then obtain y q where res = (n1,(y,q),n2) by (cases yq, auto)
with extract-SomeE[OF Some[simplified this]] have n: n = n1 @ (x,q) # n2
and res: res = (n1,(x,q),n2) by auto
from Cons(3)[unfolded n] have q: 1 ≤ q and n1n2: x ∉ fst ' (set (n1 @ n2))
and n1n2m: monom-inv (n1 @ n2) unfolding monom-inv-def by auto
show ?thesis
proof (cases p = q)
case False
have sum-var-list ((x,p) # m) x = p + sum-var-list m x unfolding

sum-var-list-def by auto
also have ... = p using sum-var-list-not[OF x] by simp
also have ... ≠ q using False.
also have q = q + sum-var-list (n1 @ n2) x using sum-var-list-not[OF n1n2] by simp
also have ... = sum-var-list n x unfolding n sum-var-list-def by auto
finally have not2: sum-var-list ((x,p) # m) x ≠ sum-var-list n x.
show ?thesis
by (simp add: xp Some res False, rule exI[of - x], rule not2)
next
case True
{
fix y
have (sum-var-list m y = sum-var-list (n1 @ n2) y) =
(sum-var-list ((x,q) # m) y = sum-var-list (n1 @ (x,q) # n2) y)
by (unfold sum-var-list-def, cases x = y, auto)
}
hence id: (∀ y. sum-var-list m y = sum-var-list (n1 @ n2) y) =
(∀ y. sum-var-list ((x,q) # m) y = sum-var-list (n1 @ (x,q) # n2) y)
by blast
show ?thesis
by (simp add: xp Some res True Cons(1)[OF m n1n2m], simp only: n, rule id)
qed
qed
done

lemma eq-monom-sym: assumes m: monom-inv m and n: monom-inv n shows
(m =m n) = (n =m m)
by (simp add: eq-monom-sum-var-list[OF m n] eq-monom-sum-var-list[OF n m]
sum-var-list-def, auto)

lemma eq-monom-trans: assumes m1: monom-inv m1 and m2: monom-inv m2
and m3: monom-inv m3 shows
m1 =m m2 ⇒ m2 =m m3 ⇒ m1 =m m3
by (simp add: eq-monom-sum-var-list[OF m1 m2] eq-monom-sum-var-list[OF m2 m3]
 eq-monom-sum-var-list[OF m1 m3] sum-var-list-def)

show that equality of monomials implies equal evaluations

lemma eq-monom: m =m n ⇒ eval-monom α m = eval-monom α n
proof (induct m arbitrary: n)
case (Cons xp m) note mCons = this
show ?case
proof (cases xp)
case (Pair x p)
show ?thesis
proof (cases List.extract (λ yq. fst yq = x) n)
case None
with Cons Pair show ?thesis by auto

next
case (Some res)
obtain n1 yq n2 where res = (n1,yq,n2) by (cases res, auto)
then obtain y q where res : res = (n1,(y,q),n2) by (cases yq, auto)
from extract-SomeE[OF Some[simplified res]] mCons(2) Some Pair res have n : n = n1 @ (x,p) # n2 and rec : m =m (n1 @ n2) by auto
show ?thesis by (simp add: Pair mCons(1)[OF rec] n field-simps)
qed
qed
qed simp

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. $x \text{Suc}(m) = x \text{Suc}(n)$ for all $n, m$.

lemma eq-monom-inv:
  fixes m :: 'v monom
  assumes exp-inject : $\forall p q :: \text{nat}. \exists \text{base :: 'a :: poly-carrier. base}^p = \text{base}^q$ $\Rightarrow p = q$ and m : monom-inv m and n : monom-inv n shows (m =m n) = ($\forall \alpha :: (\text{'v,a :: poly-carrier}) \text{assign. eval-monom } \alpha \text{ m} = \text{eval-monom } \alpha \text{ n}$)
proof(intro iffI allI, rule eq-monom)
  assume $\forall \alpha :: (\text{'v,a :: poly-carrier}) \text{assign. eval-monom } \alpha \text{ m} = \text{eval-monom } \alpha \text{ n}$
  with m n show m =m n
  proof (induct m arbitrary: n)
    case Nil
    show ?case
    proof (cases n)
      case (Cons yq nn)
      with Nil obtain y q where yq : yq = (y,q) and I $\leq$ q by (cases yq, auto simp: monom-inv-def)
      then obtain qq where q : q = Suc (qq) by (cases q, auto)
      from Nil(3) have I = eval-monom (\lam x. \theta :: 'a) n (is ?one = -) by simp
      also have ... = 0 by (simp add: Cons yq q)
      finally show ?thesis by simp
    qed simp
  next
case (Cons xp m) note mCons = this
  show ?case
  proof (cases xp)
    case (Pair x p)
    let ?ass = (\lam v y. if x = y then v else 1 :: 'a)
    fix v :: 'a and m :: 'v monom
    assume x $\notin$ fst '(set m)
    hence eval-monom (?ass v) m = 1
    proof (induct m)
      case (Cons yq m)
      thus ?case
  \end{proof}

10
by (cases yp, cases fst yp = x, auto)
qed simp

} note ass = this

from Cons(2)[unfolded Pair] obtain pp where p: p = Suc pp and xm: x ≠ fst ' (set m) unfolding monom-inv-def by (cases p, auto)

from ass[OF xm] have \( \forall v. eval\text{-}monom (?ass v) \) \( n \in v \ast v'pp \) by (simp add: Pair p)

with Cons(4) have eval: \( \forall v. eval\text{-}monom (?ass v) \) \( n = v \ast v'pp \) by auto

show \(?thesis\)

proof (cases List.extract (λ yq. fst yq = x) n)

<table>
<thead>
<tr>
<th>case None</th>
</tr>
</thead>
</table>
| have \( \forall v. \text{eval\text{-}monom } (\?ass v) \) \( n = 1 \) by (auto simp: extract-None-iff)
| from this[of 0] eval[of 0] show \(?thesis\) by simp |

next

case (Some res)

obtain n1 yq n2 where res = (n1,yq,n2) by (cases res, auto)

then obtain y q where res = (n1,(y,q),n2) by (cases yq, auto)

from extract-SomeE[OF Some[simplified this]] mCons(2) Some Pair this

have n: \( n = n1 @ \langle x,y \rangle \# n2 \) and res: res = (n1,(x,y),n2) by auto

from mCons(3)[unfolded n] have xn: \( x \notin \text{fst } (\text{set } (n1 \# n2)) \) unfolding monom-inv-def by auto

have \( \forall v. eval\text{-}monom (?ass v) \) \( n = v'q \ast eval\text{-}monom (?ass v) \) \( n1 \# n2 \) unfolding n by (auto simp: field-simps)

from eval[unfolded this ass[OF xm]] have id: \( \forall v :: 'a. \text{v'}p = v'q \) using p by auto

from \( \text{eval\text{-}inject[of p q]} \) id have pq: \( p = q \) by auto

have rec: \( ((xp \# m) = m n) = (m = m (n1 \# n2)) \) by (simp add: Pair Some res pq)

have ass: \( \forall \alpha :: ('v,'a)\text{assign. }eval\text{-}monom \alpha m = eval\text{-}monom \alpha (n1 @ n2) \)

proof

fix \( \alpha :: ('v,'a)\text{assign} \\

show eval\text{-}monom \( \alpha m = eval\text{-}monom \alpha (n1 @ n2) \) by (rule ccontr)

assume neq: \( \neg \?thesis \)

let \( ?ass = \lambda y :: 'v. \text{if } x = y \text{ then } 1 \text{ else } \alpha y \)

\{ 

fix \( m :: 'v \text{ monom} \\

assume \( x \notin \text{fst } \text{set } m \) \\

hence eval\text{-}monom \( \alpha m = eval\text{-}monom ?ass m \) by (induct m, auto)
\}

} note ass = this

have \( \forall \alpha :: (n1 @ n2) = eval\text{-}monom ?ass (n1 @ n2) \) using ass[OF xm] .

also have \( \ldots = eval\text{-}monom ?ass n \) unfolding n by auto

also have \( \ldots = eval\text{-}monom ?ass \) \( (xp \# m) \) using mCons(4) by auto

also have \( \ldots = eval\text{-}monom ?ass m \) unfolding Pair by simp

also have \( \ldots = eval\text{-}monom \alpha m \) using ass[OF xm] by simp

also have \( \ldots \neq eval\text{-}monom \alpha (n1 @ n2) \) by (rule neq)
finally show False by simp
  qed
  qed
  from mCons(2) mCons(3) have monom-inv m and monom-inv (n1 @ n2)
  unfolding monom-inv-def n by auto
  from mCons(1)[OF this ass] rec show ?thesis by simp
  qed
  qed
  qed
  qed simp

declare eq-monom.simps[simp del]

abbreviation monom-vars :: 'v monom ⇒ 'v set
  where monom-vars m ≡ fst ' set m

fun monom-mult :: 'v monom ⇒ 'v monom ⇒ 'v monom
  where monom-mult [] n = n
  | monom-mult ((x,p) #) m n = (case List.extract (λ yq. fst yq = x) n of
    None ⇒ (x,p) # monom-mult m n
    | Some (n1,(q),n2) ⇒ (x,p+q) # monom-mult m (n1 @ n2))

lemma monom-mult-vars: monom-vars (monom-mult m1 m2) = monom-vars m1
  ∪ monom-vars m2
  proof (induct m1 arbitrary: m2)
    case (Cons xp m) note mCons = this
    show ?case
    proof (cases xp)
      case (Pair x p)
      show ?thesis
      proof (cases List.extract (λ yq. fst yq = x) m2)
        case None
        with Cons Pair show ?thesis by auto
      next
        case (Some res)
        obtain n1 yq n2 where res = (n1,yq,n2) by (cases res, auto)
        then obtain y q where res: res = (n1,(y,q),n2) by (cases yq, auto)
        from extract-SomeE[OF Some[simplified res]] Some Pair res have m2: m2 = n1 @ (x,q) # n2
        and rec: monom-mult (xp #) m2 = (x, p+q) # monom-mult m (n1 @ n2) by auto
        show ?thesis by (simp only: rec, simp add: mCons Pair m2)
      qed
    qed
    qed simp

lemma monom-mult-inv: monom-inv m1 ⇒ monom-inv m2 ⇒ monom-inv (monom-mult m1 m2)
proof (induct m1 arbitrary: m2)
  case Nil thus ?case by (simp add: monom-inv-def)
next
  case (Cons xp m1)
  obtain x p where xp: xp = (x,p) by (cases xp) auto
  from xp Cons(2) have m1: monom-inv m1 and x: x \notin monom-vars m1 and p: 1 \leq p by (auto simp: monom-inv-def)
  show ?case
    proof (cases List.extract (\lambda yq. fst yq = x) m2)
      case None
      hence x \notin monom-vars m2 by (auto simp: extract-None-iff)
      from x this have x: x \notin monom-vars (monom-mult m1 m2) by (simp add: monom-mult-vars)
      with None Cons(1)[OF m1 Cons(3)] x p xp show ?thesis by (auto simp: monom-inv-def)
    next
      case (Some res)
      obtain n1 yq n2 where res = (n1,yq,n2) by (cases res, auto)
      then obtain q where q: q \in monom-vars m2 by (auto simp: monom-inv-def)
      from extract-SomeE[OF Some[simplified res]] Some xp res have m2: m2 = n1 @ (x,q) # n2
      and rec: monom-mult (xp # m1) m2 = (x, p+q) # monom-mult m1 (n1 @ n2) by auto
      from Cons(3) m2 have xn1: x \notin monom-vars (n1 @ n2)
      and n1n2: monom-inv (n1 @ n2) and q: 1 \leq q by (auto simp: monom-inv-def)
      from x xn1 have x: x \notin monom-vars (monom-mult m1 (n1 @ n2)) by (simp add: monom-mult-vars)
      from p q have pq: 1 \leq p + q by auto
      from Cons(1)[OF m1 n1n2] pq xn1 show ?thesis by (simp only: rec, auto simp: monom-inv-def x)
    qed
  qed

lemma monom-mult-inj: assumes m: monom-inv m and m1: monom-inv m1 and m2: monom-inv m2 and eq: monom-mult m m1 =m monom-mult m m2
shows m1 =m m2
proof -
  { fix x n
    have sum-var-list (monom-mult m n) x = sum-var-list m x + sum-var-list n x
      proof (induct m arbitrary: n, simp add: sum-var-list-def)
        case (Cons yp m)
        obtain y p where yp: yp = (y,p) by (cases yp, auto)
        hence id: sum-var-list (yp # m) x = (if x = y then p else 0) + sum-var-list m x unfolding sum-var-list-def by auto
        show ?case
          proof (cases List.extract (\lambda zq. fst zq = y) n)
            ...
case None
  have sum-var-list (monom-mult (yp # m) n) x = (if x = y then p else 0)
  thus ?thesis
  by (simp add: yp None sum-var-list-def)
next
  case (Some res)
  obtain n1 z q n2 where res = (n1,zq,n2) by (cases res, auto)
  then obtain z q where res = (n1,(z,q),n2) by (cases zq, auto)
  from extract-SomeE[OF Some[simplified res]] Some res yp have n: n = n1 @ (y,q) # n2
  and rec: sum-var-list (monom-mult (yp # m) n) x = (if x = y then p+q
  else 0) + sum-var-list (monom-mult m (n1 @ n2)) x
  unfolding sum-var-list-def by auto
  show ?thesis
  by (simp only: rec Cons id, simp add: n sum-var-list-def)
  qed
qed

lemma monom-mult[simp]: eval-monom α (monom-mult m n) = eval-monom α m * eval-monom α n
proof (induct m arbitrary: n)
case (Cons xp m) note mCons = this
show ?case
proof (cases xp)
case (Pair x p)
  show ?thesis
  proof (cases List.extract (λ yq. fst yq = x) n)
    case None
    with Cons Pair show ?thesis by (auto simp: field-simps)
  next
    case (Some res)
    obtain n1 yq n2 where res = (n1,yq,n2) by (cases res, auto)
    then obtain y q where res = (n1,(y,q),n2) by (cases yq, auto)
    from extract-SomeE[OF Some[simplified res]] Some Pair res have n: n = n1 @ (x,q) # n2
    and rec: monom-mult (xp # m) n = (x, p+q) # monom-mult m (n1 @ n2) by auto
    show ?thesis by (simp only: rec, simp add: Pair mCons[of n1 @ n2] n
    field-simps power-add)
  qed
  qed

qed
Polynomials are represented as a sum of monomials multiplied by some coefficient.

**type-synonym** \((\text{\texttt{v}}, \text{\texttt{a}})\text{poly} = (\text{\texttt{v monom}} \times \text{\texttt{a}})\text{list} \)**

The polynomials we construct satisfy the following invariants:

- All monomials satisfy their invariant.
- All coefficients are non-zero.
- No two equivalent monomials occur in the list.

```plaintext
fun distinct-eq :: \((\text{\texttt{a}} \Rightarrow \text{\texttt{a}} \Rightarrow \text{bool}) \Rightarrow \text{\texttt{a list}} \Rightarrow \text{bool} \) where
  distinct-eq - [] = True
  | distinct-eq eq (x # xs) = ((\forall y \in \text{set} \text{\texttt{xs}}. \neg (\text{eq} \ y \ x)) \land \text{distinct-eq} \ eq \ \text{\texttt{xs}})

lemma distinct-eq-append: distinct-eq eq (xs @ ys) = (distinct-eq eq \text{\texttt{xs}} \land \text{distinct-eq eq \text{\texttt{ys}}} \land (\forall x \in \text{set} \text{\texttt{xs}}. \forall y \in \text{set} \text{\texttt{ys}}. \neg (\text{eq} \ y \ x)))
  by (induct \text{\texttt{xs}}, \text{auto})

definition poly-inv :: \((\text{\texttt{v}}, \text{\texttt{a}} :: \text{\texttt{zero}})\text{poly} \Rightarrow \text{bool} \) where
  poly-inv p \equiv (\forall (\text{\texttt{m}}, \text{\texttt{c}}) \in \text{\texttt{set}} \text{\texttt{p}}. \text{\texttt{monom-inv}} \text{\texttt{m}} \land \text{\texttt{c}} \neq 0) \land \text{\text{distinct-eq}} (\lambda (\text{\texttt{m}1,-}) (\text{\texttt{m}2,-}). \text{\texttt{m}1} = \text{\texttt{m}2}) \text{\texttt{p}}

abbreviation eval-monomc where eval-monomc \(\text{\texttt{mc}} \equiv \text{eval-monom} \ (\text{\texttt{fst}} \ \text{\texttt{mc}}) \ast (\text{\texttt{snd}} \ \text{\texttt{mc}})

fun eval-poly :: \((\text{\texttt{v}}, \text{\texttt{a}} :: \text{\texttt{comm-semiring-0}})\text{assign} \Rightarrow \text{\texttt{v}}\text{poly} \Rightarrow \text{\texttt{a}}\text{poly} \Rightarrow \text{\texttt{a}}\text{poly} \) where
  eval-poly \(\text{\texttt{\[]}} = \text{\texttt{0}}\)
  | eval-poly \((\text{\texttt{mc}} \# \text{\texttt{p}})\) = eval-monomc \(\text{\texttt{mc}}\) + eval-poly \(\text{\texttt{p}}\)

fun poly-add :: \((\text{\texttt{v}}, \text{\texttt{a}})\text{poly} \Rightarrow \text{\texttt{v}}\text{poly} \Rightarrow \text{\texttt{a}}\text{poly} \Rightarrow \text{\texttt{a}}\text{poly} \) where
  poly-add \(\text{\texttt{[]}} q = q\)
  | poly-add \((\text{\texttt{mc}}, \text{\texttt{c}}) \# p\) q = (case List.extract (\lambda \text{\texttt{mc}}. \text{\texttt{fst}} \text{\texttt{mc}} = \text{\texttt{m}} \text{\texttt{m}}) \text{\texttt{q}} \) of
    None \Rightarrow (\text{\texttt{mc}}, \text{\texttt{c}}) \# \text{\texttt{poly-add}} p \text{\texttt{q}}
    | Some (\text{\texttt{q}1,(-,-)},\text{\texttt{q}2}) \Rightarrow if \ (\text{\texttt{c}} + \text{\texttt{d}} = 0) \ then \text{\texttt{poly-add}} p \text{\texttt{(q1 \ @ \ q2)}}
      else (\text{\texttt{m}},\text{\texttt{c+d}}) \# \text{\texttt{poly-add}} p \text{\texttt{(q1 \ @ \ q2)}}

lemma eval-poly-append[simp]: eval-poly \(\text{\texttt{\alpha}} \ (\text{\texttt{mc}1} \ @ \ \text{\texttt{mc}2}) = \text{eval-poly} \ (\text{\texttt{mc}1} + \text{\texttt{poly}} \ (\text{\texttt{mc}2})\)\)
  by (induct \text{\texttt{mc}1}, \text{auto} simp: field-simps)

abbreviation poly-monoms :: \((\text{\texttt{v}}, \text{\texttt{a}})\text{poly} \Rightarrow \text{\texttt{v monom set}} \)
where poly-monom p ≡ fst ' set p

lemma poly-add-monom: poly-monom (poly-add p1 p2) ⊆ poly-monom p1 ∪ poly-monom p2
proof (induct p1 arbitrary: p2)
case (Cons mc p)
obtain m c where mc: mc = (m,c) by (cases mc, auto)
hence m: m ∈ poly-monom (mc # p1) by auto
show ?case
proof (cases List.extract (λ nd. fst nd =m m) p2)
case None
  with Cons m show ?thesis by (auto simp: mc)
next
case (Some res)
obtain q1 md q2 where res: res = (q1,md,q2) by (cases res, auto)
from extract- SomeE[OF Some[simplified res]] res obtain m' d where q: p2 = q1 @ (m',d) ≠ q2 and res: res = (q1,(m',d),q2) and mm': m' =m m by (cases md, auto)
  show ?thesis
    by (simp add: mc Some res, rule subset-trans[OF Cons[of q1 @ q2]], auto simp: q)
qed
qed simp

lemma poly-add-inv: poly-inv p → poly-inv q → poly-inv (poly-add p q)
proof (induct p arbitrary: q)
case (Cons mc p)
obtain m c where mc: mc = (m,c) by (cases mc, auto)
with Cons(2) have p: poly-inv p and m: monom-inv m and c: c ≠ 0 and mp:
∀ (mm,dd) ∈ set p. (¬ mm =m m) unfolding poly-inv-def by auto
show ?case
proof (cases List.extract (λ mc. fst mc =m m) q)
case None
  hence mq: ∀ (mm,dd) ∈ set q. ¬ mm =m m by (auto simp: extract-None-iff)
  { fix mm dd
    assume (mm,dd) ∈ set (poly-add p q)
    with poly-add-monom have mm ∈ poly-monom p ∨ mm ∈ poly-monom q
      by force
    hence ¬ mm =m m
    proof
      assume mm ∈ poly-monom p
      thus ?thesis using mp by auto
    next
      assume mm ∈ poly-monom q
      thus ?thesis using mq by auto
    qed
  } note main = this

show ?thesis using \textit{Cons}(1)[OF p Cons(3)] unfolding poly-inv-def by (auto simp add: None mc m c main)

next

case (Some res)

obtain q1 md q2 where res: res = (q1,md,q2) by (cases res, auto)

from extract-SomeE[OF Some[simplified res]] res obtain m' d where q: q = q1 @ (m',d) ≠ q2 and res: res = (q1,(m',d),q2) and mm': m' =m m by (cases md, auto)

from q Cons(3) have q1q2: poly-inv (q1 @ q2) and m': monom-inv m' unfolding poly-inv-def by (auto simp: distinct-eq-append)

from Cons(1)[OF p q1q2] have main1: poly-inv (poly-add p (q1 @ q2)) .

{ fix mm dd
  assume (mm,dd) ∈ set (poly-add p (q1 @ q2))
  with poly-add-monomms have mm ∈ poly-monomms p ∨ mm ∈ poly-monomms (q1 @ q2) by force
  hence ¬ mm =m m
  proof
    assume mm ∈ poly-monomms p
    thus ?thesis using mp by auto
  next
    assume member: mm ∈ poly-monomms (q1 @ q2)
    with q1q2 have mm: monom-inv mm unfolding poly-inv-def by auto
    from member have mm ∈ poly-monomms q1 ∨ mm ∈ poly-monomms q2 by auto
    hence mmm': ¬ mm =m m'
    proof
      assume mm ∈ poly-monomms q2
      with Cons(3)[simplified q]
      show ?thesis unfolding poly-inv-def by (auto simp: distinct-eq-append)
    next
      assume mm ∈ poly-monomms q1
      with Cons(3)[simplified q]
      have ¬ m' =m mm unfolding poly-inv-def by (auto simp: distinct-eq-append)
      thus ?thesis using eq-monom-sym[OF m' mm] by blast
    qed
    show ?thesis
    proof
      assume mm =m m
      from this mm[simplified eq-monom-sym[OF m' m]]
      have mm =m m' using eq-monom-trans[OF mm m m'] by blast
      with mmm' show False by simp
    qed
    qed
  }

note main2 = this
show ?thesis
  by (simp add: mc Some res main1, simp add: poly-inv-def m, auto simp: main1[unfolded poly-inv-def] main2)
qed
lemma poly-add[simp]: eval-poly α (poly-add p q) = eval-poly α p + eval-poly α q
proof (induct p arbitrary: q)
  case (Cons mc p)
  obtain m c where mc: mc = (m,c) by (cases mc, auto)
  show ?thesis by (simp add: Cons[of q] mc None field-simps)
next
  case (Some res)
  obtain q1 md q2 where res: res = (q1,md,q2) by (cases res, auto)
  from extract-SomeE[OF Some[simplified res]] res obtain m’ d where q: q = q1 @ (m’,d) # q2 and m’ =m m and res: res = (q1,(m’,d),q2) by (cases md, auto)
  { fix x
    assume c: c + d = 0
    have c * x + d * x = (c + d) * x by (auto simp: field-simps)
    also have ... = 0 * x by (simp only: c)
    finally have c * x + d * x = 0 by simp
  } note id = this
  show ?thesis by (simp add: Cons[of q1 @ q2] mc Some res, simp only: q, simp add: eq-monom[OF m'] field-simps, auto simp: field-simps id)
qed
qed simp

declare poly-add.simps[simp del]

fun monom-mult-poly :: ('v 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 (monom-mult m m’, c * d) # monom-mult-poly (m,c) p)

lemma monom-mult-poly-inv: assumes m: monom-inv m shows poly-inv p ⇒ poly-inv (monom-mult-poly (m,c) p)
proof (induct p)
  case Nil thus ?case by (simp add: poly-inv-def)
next
  case (Cons md p)
  obtain m’ d where md: md = (m’,d) by (cases md, auto)
  with Cons(2) have m’: monom-inv m’ and p: poly-inv p unfolding poly-inv-def
  by auto
  from Cons(1)[OF p] have prod: poly-inv (monom-mult-poly (m,c) p) .
  {
\textbf{fix \( mm \), \( dd \)}

\textbf{assume one:} \((mm,dd) \in \text{set \( \{\text{monom-mult-poly} \ (m,c) \ p\} \)}\)

\textbf{and two:} \( mm = m \ \text{monom-mult} \ m' \)

\textbf{have poly-monom} \((\text{monom-mult-poly} \ (m,c) \ p) \subseteq \text{monom-mult} \ m \ \text{monom-mult-poly} \ p\)

\textbf{proof (induct \( p \), simp)}

\textbf{case (Cons \( md \ p \))}

\textbf{thus \( ?\text{case} \)}

\textbf{by (cases \( md \), auto)}

\textbf{qed}

\textbf{with one have} \( mm \in \text{monom-mult} \ m \ \text{monom-mult-poly} \ p \) \textbf{by force}

\textbf{then obtain mmm where} \( mmm \in \text{poly-monom} \ p \) \textbf{and} \( mm = \text{monom-mult} \ m \ mmm \) \textbf{by blast}

\textbf{from Cons(2) \( [\text{simplified \( md \)] \ mmm \ have not1: } \neg \ mmm = m \ m' \ \text{and} \ mmm: \text{monom-inv} \ m \ mmm \ \text{unfolding poly-inv-def by (auto simp: distinct-eq-append)} \)\n
\textbf{from mmm two have} \( \text{monom-mult} \ m \ mmm = m \ \text{monom-mult} \ m \ m' \) \textbf{by simp}

\textbf{from monom-mult-inv[OF \( m \ mmm \ m' \ this\] not1 \}

\textbf{have False by simp}

\} \textbf{thus \( ?\text{case} \) by (simp add: md prod, intro impI, simp add: poly-inv-def, simp add: monom-mult-inv[OF \( m \ m' \), auto simp: prod[simplified poly-inv-def])}

\textbf{qed}

\textbf{lemma} \( \text{monom-mult-poly[simp]:} \ \text{eval-poly} \alpha \ (\text{monom-mult-poly} \ mc \ p) = \text{eval-monom} \alpha \ mc \ast \text{eval-poly} \alpha \ p \)

\textbf{proof (cases \( mc \))}

\textbf{case (Pair \( m \ c \))}

\textbf{show \( ?\text{thesis} \) proof (simp add: Pair, induct \( p \))}

\textbf{case (Cons \( nd \ q \)}

\textbf{obtain} \( n \ d \) \textbf{where} \( nd: \ nd = (n,d) \) \textbf{by (cases \( nd \), auto)}

\textbf{show \( ?\text{case} \) proof (cases \( c \ast d = 0 \))}

\textbf{case False}

\textbf{thus \( ?\text{thesis} \) by (simp add: nd Cons field-simps)}

\textbf{finally have} \( l: \ l = 0 \) \textbf{by (simp only: True, simp add: field-simps)}

\textbf{show \( ?\text{thesis} \) by (simp add: nd Cons True, simp add: field-simps \( l \))}

\textbf{qed}

\textbf{qed simp}

\textbf{qed}
declare monom-mult-poly.simps [simp del]

fun poly-mult :: ('v,'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)
using p
proof (induct p)
case Nil thus _case by (simp add: poly-inv-def)
next
case (Cons mc p)
obtain m c where mc: mc = (m,c) by (cases mc, auto)
with Cons(2) have m: monom-inv m and p: poly-inv p unfolding poly-inv-def
by auto
  show ?case
    by (simp add: mc, rule poly-add-inv[OF monom-mult-poly-inv[OF m q] Cons(1)[OF p]])
qed

lemma poly-mult[simp]: eval-poly α (poly-mult p q) = eval-poly α p * eval-poly α q
proof (induct p)
case (Cons mc p)
  thus ?case
    by (simp add: field-simps)
qed simp

declare poly-mult.simsps [simp del]

definition zero-poly :: ('v,'a :: semiring-1)poly
where zero-poly ≡ []

lemma zero-poly-inv: poly-inv zero-poly unfolding zero-poly-def poly-inv-def by auto

definition one-poly :: ('v,'a :: semiring-1)poly
where one-poly ≡ ([],[1])

lemma one-poly-inv: poly-inv one-poly unfolding one-poly-def poly-inv-def monom-inv-def
by auto


lemma poly-zero-mult: poly-mult zero-poly zero-poly p = zero-poly unfolding zero-poly-def
using poly-mult.simsps by auto

  equality of polynomials
definition eq-poly :: ('v,'a :: comm-semiring-1)poly ⇒ ('v,'a)poly ⇒ bool (infix \(=_{p}51\))
where \( p =_{p} q \equiv \forall \alpha. \text{eval-poly} \alpha p = \text{eval-poly} \alpha q \)

lemma poly-one-mult: poly-mult one-poly p =_{p} p

unfolding eq-poly-def one-poly-def
by (simp)

lemma eq-poly-refl[simp]: \( p =_{p} p \) unfolding eq-poly-def by auto

lemma eq-poly-trans[trans]: \([p1 =_{p} p2; p2 =_{p} p3] \implies p1 =_{p} p3\)
unfolding eq-poly-def by (auto simp: field-simps)

lemma poly-mult-comm: poly-mult p q =_{p} poly-mult q p

unfolding eq-poly-def by (auto simp: field-simps)

lemma poly-mult-assoc: poly-mult p1 (poly-mult p2 p3) =_{p} poly-mult (poly-mult p1 p2) p3
unfolding eq-poly-def by (auto simp: field-simps)

lemma poly-distrib: poly-mult p (poly-add q1 q2) =_{p} poly-add (poly-mult p q1)
(poly-mult p q2) unfolding eq-poly-def by (auto simp: field-simps)

2.3 Computing normal forms of polynomials

fun poly-of :: ('v,'a :: comm-semiring-1)tpoly ⇒ ('v,'a)poly
where poly-of (PNum i) = (if i = \(0\) then [] else [(\([\([x,1]\)],1\)])]
| poly-of (PVar x) = [(\([\([x,1]\)],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-inv: eval-poly \(\alpha\) (poly-of p) = eval-tpoly \(\alpha\) p
by (induct p rule: poly-of.induct, (simp add: zero-poly-def one-poly-def)+)

poly_of only generates polynomials that satisfy the invariant

lemma poly-of-inv: poly-inv (poly-of p)
2.4 Powers and substitutions of polynomials

fun poly-power :: ('v, '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
by (induct n, auto simp: one-poly-def)

lemma poly-power-inv: assumes p: poly-inv p
shows poly-inv (poly-power p n)
by (induct n, simp add: one-poly-inv, simp add: poly-mult-inv[OF p])

declare poly-power.simps[simp del]

fun monom-subst :: ('v ⇒ (w, 'a :: comm-semiring-1)poly) ⇒ 'v monom ⇒ (w, 'a)poly
where monom-subst σ [] = one-poly
| monom-subst σ ((x,p) ≠ m) = poly-mult (poly-power (σ x) p) (monom-subst σ m)

lemma monom-subst-inv: assumes sub: \( \forall x. \text{poly-inv}(\sigma x) \)
shows poly-inv (monom-subst σ m)
proof (induct m)
  case Nil thus ?case by (simp add: one-poly-inv)
next
  case (Cons xp m)
  obtain x p where xp: xp = (x,p) by (cases xp, auto)
  show ?case by (simp add: xp, rule poly-mult-inv[OF poly-power-inv[OF sub] Cons])
qed

declare monom-subst.simps[simp del]

fun poly-subst :: ('v ⇒ ('w,'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 [[],c]) (monom-subst σ m)
(poly-subst σ p)

lemma poly-subst-inv: assumes sub: \( \forall x. \text{poly-inv}(\sigma x) \) and p: poly-inv p
shows poly-inv (poly-subst σ p)
using p
proof (induct p)
  case Nil thus ?case by (simp add: zero-poly-inv)
next
  case (Cons mc p)
  obtain m c where mc: mc = (m,c) by (cases mc, auto)
  with Cons(2) have c: c ≠ 0 and p: poly-inv p unfolding poly-inv-def by auto
  from c have c: poly-inv [([],c)] unfolding poly-inv-def monom-inv-def by auto
  show ?case
    by (simp add: mc, rule poly-add-inv [OF poly-mult-inv [OF c monom-subst-inv[OF sub]] Cons(1)[OF p]])
qed

lemma poly-subst: eval-poly α (poly-subst σ p) = eval-poly (λ v. eval-poly α (σ v)) p
  by (induct p, simp add: zero-poly-def, auto simp: field-simps)

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 (induct p)
  case Nil thus ?case by (simp add: zero-poly-def)
next
  case (Cons mc p)
  obtain m c where mc: mc = (m,c) by (cases mc, auto)
  have id: eval-monom f m = eval-monom (λ v. eval-poly g (q v)) m
  proof (induct m)
    case (Cons wp m)
    obtain w p where wp: wp = (w,p) by (cases wp, auto)
    show ?case
      by (simp add: wp Cons eq)
  qed simp
  show ?case
    by (simp add: mc Cons id, simp add: field-simps)
qed

definition poly-vars-list :: ('v,'a)poly ⇒ 'v list
where poly-vars-list p = remdups (concat (map (map fst o fst) p))

definition poly-vars :: ('v,'a)poly ⇒ 'v set
where poly-vars p = set (concat (map (map fst o fst) p))

lemma poly-vars-list[simp]: set (poly-vars-list p) = poly-vars p
  unfolding poly-vars-list-def poly-vars-def by auto

lemma poly-vars: assumes eq: ∀ w. w ∈ poly-vars p ⇒ f w = g w
  shows poly-subst f p = poly-subst g p
  using eq
  proof (induct p)
case (Cons mc p) 
  hence rec: poly-subst f p = poly-subst g p unfolding poly-vars-def by auto 
  show ?case 
  proof (cases mc) 
    case (Pair m c) 
      with Cons(2) have \( \wedge \ w. \ w \in \text{set} (\text{map} \ \text{fst} \ m) \implies f \ w = g \ w \) unfolding poly-vars-def by auto 
    hence monom-subst f m = monom-subst g m 
    proof (induct m) 
      case Nil thus \( \text{thesis} \) by (simp add: monom-subst.simps) 
      next 
      case (Cons wn m) 
      hence rec: monom-subst f m = monom-subst g m and eq: \( f (\text{fst} \ wn) = g (\text{fst} \ wn) \) by auto 
      show ?case 
      proof (cases wn) 
        case (Pair w n) 
        hence rec: monom-subst f m = monom-subst g m and eq: \( f (\text{fst} \ wn) = g (\text{fst} \ wn) \) by auto 
        show ?thesis by (auto simp: monom-subst.simps) 
      qed 
      qed 
      with rec Pair show ?thesis by auto 
    qed 
    qed 
    simp 
  qed 

lemma poly-var: assumes pv: \( v \notin \text{poly-vars} \ p \) and diff: \( \wedge \ w. \ v \neq w \implies f \ w = g \ w \) 
  shows poly-subst f p = poly-subst g p 
proof (rule poly-vars) 
  fix w 
  assume w \in poly-vars p 
  thus \( f \ w = g \ w \) using pv diff by (cases v = w, auto) 
qed 

lemma eval-poly-vars: assumes \( \wedge \ x. \ x \in \text{poly-vars} \ p \implies \alpha \ x = \beta \ x \) 
  shows eval-poly \( \alpha \ p = \text{eval-poly} \ \beta \ p \) 
using assms 
proof (induct p) 
  case Nil thus \( \text{thesis} \) by simp 
next 
  case (Cons m p) 
  from Cons(2) have \( \wedge \ x. \ x \in \text{poly-vars} \ p \implies \alpha \ x = \beta \ x \) unfolding poly-vars-def by auto 
  from Cons(1)[OF this] have IH: eval-poly \( \alpha \ p = \text{eval-poly} \ \beta \ p \) . 
  obtain xs c where m = (xs,c) by force 
  from Cons(2) have \( \wedge \ x. \ x \in \text{set} (\text{map} \ \text{fst} \ xs) \implies \alpha \ x = \beta \ x \) unfolding poly-vars-def \( m \) by auto 
  hence eval-monom \( \alpha \ xs = \text{eval-monom} \ \beta \ xs \) 
  proof (induct xs) 
    case Nil thus \( \text{thesis} \) by simp 
  qed
next
  case (Cons xi xs)
  hence IH: eval-monom α xs = eval-monom β xs by auto
obtain x i where xi: xi = (x,i) by force
from Cons(2) xi have α x = β x by auto
with IH show ?case unfolding xi by auto
qed
thus ?case unfolding eval-poly.simps IH m by auto
qed

declare poly-subst.simps[simp del]

2.5 Polynomial orders

definition pos-assign :: ('v,'a :: ordered-semiring-0)assign ⇒ bool
where pos-assign α = (∀ x. α x ≥ 0)

definition poly-ge :: ('v,'a :: poly-carrier)poly ⇒ ('v,'a)poly ⇒ bool (infix ≥p 51)
where p ≥p q = (∀ α. pos-assign α −→ eval-poly α p ≥ eval-poly α q)

lemma poly-ge-refl[simp]: p ≥p p
unfolding poly-ge-def using ge-refl by auto

lemma poly-ge-trans[trans]: [p1 ≥p p2; p2 ≥p p3] ⇒ p1 ≥p p3
using assms unfolding poly-ge-def using ge-trans by blast

lemma pos-assign-monom: fixes α :: ('v,'a :: poly-carrier)assign
  assumes pos: pos-assign α
  shows eval-monom α m ≥ 0
proof (induct m)
case Nil thus ?case by (simp add: one-ge-zero)
next
case (Cons xp m)
show ?case
proof (cases xp)
case (Pair x p)
  from pos[unfolded pos-assign-def] have ge: α x ≥ 0 by simp
  have ge: α x ∧ p ≥ 0
  proof (induct p)
  case 0 thus ?case by (simp add: one-ge-zero)
  next
case (Suc p)
  from ge-trans[OF times-left-mono[OF ge Suc] times-right-mono[OF ge-refl ge]]
  show ?case by (simp add: field-simps)
qed
from ge-trans[OF times-right-mono[OF Cons ge] times-left-mono[OF ge-refl ge]]
Cons]

show ?thesis
  by (simp add: Pair)
qed

lemma pos-assign-poly: assumes pos: pos-assign α
  and p: p ≥ p zero-poly
  shows eval-poly α p ≥ 0
proof –
  from p[unfolded poly-ge-def zero-poly-def] pos
  show ?thesis by auto
qed

lemma poly-add-ge-mono: assumes p1 ≥ p p2 shows poly-add p1 q ≥ p poly-add p2 q
using assms unfolding poly-ge-def by (auto simp: field-simps plus-left-mono)

lemma poly-mult-ge-mono: assumes p1 ≥ p p2 and q ≥ p zero-poly
  shows poly-mult p1 q ≥ p poly-mult p2 q
using assms unfolding poly-ge-def zero-poly-def by (auto simp: times-left-mono)

context poly-order-carrier
begin

definition poly-gt :: ('a,poly)⇒ ('a,poly)⇒ bool
where p > p q = (∀ α. pos-assign α → eval-poly α p ≻ eval-poly α q)

lemma poly-gt-imp-poly-ge: p > p q =⇒ p ≥ p q unfolding poly-ge-def poly-gt-def
using gt-imp-ge by blast

abbreviation poly-GT :: ('a,poly)rel
where poly-GT ≡ {((p,q) | p q. p >p q ∧ q ≥ p zero-poly}

lemma poly-compat: [p1 ≥ p p2; p2 ≥ p p3] =⇒ p1 > p p3
using assms unfolding poly-ge-def poly-gt-def using compat by blast

lemma poly-compat2: [p1 > p p2; p2 ≥ p p3] =⇒ p1 > p p3
using assms unfolding poly-ge-def poly-gt-def using compat2 by blast

lemma poly-gt-trans: trans: [p1 > p p2; p2 > p p3] =⇒ p1 > p p3
using assms unfolding poly-gt-def using gt-trans by blast

lemma poly-GT-SN: SN poly-GT

proof
  fix f :: nat ⇒ ('c,'a)poly
  assume f: ∀ i. (f i, f (Suc i)) ∈ poly-GT
  have pos: pos-assign ((λ x. 0) :: ('c,'a)assign) (is pos-assign ?ass) unfolding

pos-assign-def using ge-refl by auto
obtain g where g: \( \land \ i \cdot g \ i = \text{eval-poly } ?\text{ass} \ (f \ i) \) by auto
from f pos have \( \forall \ i \cdot g \ (\text{Suc } i) \geq 0 \land g \ i \succ g \ (\text{Suc } i) \) unfolding poly-gt-def g
using pos-assign-poly by auto
with SN show False unfolding SN-defs by blast
qed
end

monotonicity of polynomials

lemma eval-monom-mono: assumes fg: \( \land \ x \cdot (f :: (\'v,\'a :: \text{poly-carrier})\text{assign}) \ x \geq g \ x \)
and g: \( \land \ x \cdot g \ x \geq 0 \)
shows eval-monom f m \geq eval-monom g m eval-monom g m \geq 0
proof –
have eval-monom f m \geq eval-monom g m \land eval-monom g m \geq 0
proof (induct m)
case Nil show \( ?\)case using one-ge-zero by (auto simp: ge-refl)
next
case (Cons xd m)
hence IH1: eval-monom f m \geq eval-monom g m and IH2: eval-monom g m \geq 0 by auto
obtain x d where xd: xd = (x,d) by force
from pow-mono[OF fg g, OF x d] have fgd: \( f \ x \ ^d \geq g \ x \ ^d \) and gd: \( g \ x \ ^d \geq 0 \) by auto
show \( ?\)case unfolding xd eval-monom.simps
proof (rule conjI, rule ge-trans[OF times-left-mono[OF times-right-mono[OF IH2 fgd]]])
show \( f \ x \geq 0 \) by (rule ge-trans[OF fg g])
show eval-monom g m \ast g \ x \ ^d \geq 0
by (rule mult-ge-zero[OF IH2 gd])
qed
qed
thus eval-monom f m \geq eval-monom g m eval-monom g m \geq 0 by auto
qed

definition poly-weak-mono-all :: (\'v,\'a :: \text{poly-carrier})\text{poly} \rightarrow \text{bool} where
poly-weak-mono-all p \equiv \forall \ (\alpha :: (\'v,\'a)\text{assign}) \beta. (\forall \ x \cdot \alpha \ x \geq \beta \ x) 
\rightarrow pos-assign \ \beta \rightarrow \text{eval-poly } \alpha \ p \geq \text{eval-poly } \beta \ p

lemma poly-weak-mono-all-E: assumes p: poly-weak-mono-all p and
\( ge: \land \ x \cdot f \ x \geq_p g \ x \land g \ x \geq_p \text{zero-poly} \)
shows poly-subst f p \geq_p poly-subst g p
unfolding poly-ge-def poly-subst
proof (intro allI impI, rule p[unfolded poly-weak-mono-all-def, rule-format!])
fix \alpha :: (\'c,\'b)\text{assign} and x
show pos-assign \ \alpha \Rightarrow \text{eval-poly } \alpha \ (f \ x) \geq \text{eval-poly } \alpha \ (g \ x) \) using ge[of x]
unfolding poly-ge-def by auto
next
fix \alpha :: (\'c,\'b)\text{assign}
assume \( \text{alpha}: \text{pos-assign} \alpha \)
show \( \text{pos-assign} (\lambda v. \text{eval-poly} \alpha (g v)) \)
unfolding \text{pos-assign-def}

proof
  fix \( x \)
  show \( \text{eval-poly} \alpha (g x) \geq 0 \)
  using \( \text{ge}[\text{of} x] \) unfolding \text{poly-ge-def} \text{zero-poly-def} using \( \text{alpha} \) by \text{auto}
qed


definition \text{poly-weak-mono} :: \( 'v , 'a :: \text{poly-carrier} \) \text{poly} \Rightarrow 'v \Rightarrow \text{bool} \) where
  \( \text{poly-weak-mono} p v \equiv \forall (\alpha :: (\mathcal{V}' , 'a) \text{assign}) \beta. (\forall x. v \neq x \rightarrow \alpha x = \beta x) \rightarrow \text{pos-assign} \beta \rightarrow \alpha v \geq \beta v \rightarrow \text{eval-poly} \alpha p \geq \text{eval-poly} \beta p \)

lemma \text{poly-weak-mono-E}: assumes \( p : \text{poly-weak-mono} p v \)
and \( \text{fgw} : \bigwedge w. v \neq w \rightarrow f w = g w \)
and \( g : \bigwedge w. g w \geq p \text{ zero-poly} \)
and \( \text{fgv} : f v \geq p g v \)
shows \( \text{poly-subst} f p \geq p \text{ poly-subst} g p \)
unfolding \text{poly-ge-def} \text{poly-subst}

proof
  intro allI impI, rule \( p \) [unfolded \text{poly-weak-mono-def}, \text{rule-format}] 
  fix \( \alpha :: (\mathcal{C}' , 'b) \text{assign} \)
  show \( \text{pos-assign} \alpha \Rightarrow \text{eval-poly} \alpha (f v) \geq \text{eval-poly} \alpha (g v) \) using \( \text{fgv} \) unfolding \text{poly-ge-def} by \text{auto}
qed

next
  fix \( \alpha :: (\mathcal{C}' , 'b) \text{assign} \)
  assume \( v : v \neq x \)
  show \( \text{pos-assign} \alpha \Rightarrow \text{eval-poly} \alpha (f x) = \text{eval-poly} \alpha (g x) \) using \( \text{fgv}[\text{OF} v] \)
unfolding \text{poly-ge-def} by \text{auto}
qed


definition \text{poly-weak-anti-mono} :: \( 'v , 'a :: \text{poly-carrier} \) \text{poly} \Rightarrow 'v \Rightarrow \text{bool} \) where
  \( \text{poly-weak-anti-mono} p v \equiv \forall (\alpha :: (\mathcal{V}' , 'a) \text{assign}) \beta. (\forall x. v \neq x \rightarrow \alpha x = \beta x) \rightarrow \text{pos-assign} \beta \rightarrow \alpha v \geq \beta v \rightarrow \text{eval-poly} \beta p \geq \text{eval-poly} \alpha p \)

lemma \text{poly-weak-anti-mono-E}: assumes \( p : \text{poly-weak-anti-mono} p v \)
and \( \text{fgw} : \bigwedge w. v \neq w \rightarrow f w = g w \)
and \( g : \bigwedge w. g w \geq p \text{ zero-poly} \)
and \( \text{fgv} : f v \geq p g v \)
shows poly-subst g p ≥ p poly-subst f p
unfolding poly-ge-def poly-subst
proof (intro allI impI, rule p[unfolded poly-weak-anti-mono-def, rule-format])
  fix α :: ('c,'b)assign
  show pos-assign α → eval-poly α (f v) ≥ eval-poly α (g v) using fgv unfolding
  poly-ge-def by auto
next
  fix α :: ('c,'b)assign
  assume alpha: pos-assign α
  show pos-assign (λv. eval-poly α (g v))
    unfolding pos-assign-def
proof
  fix x
  show eval-poly α (g x) ≥ 0
    using g[of x] unfolding poly-ge-def zero-poly-def using alpha by auto
qed

lemma poly-weak-mono: fixes p :: ('v,'a :: poly-carrier)poly
  assumes mono: ⋀ v. v ∈ poly-vars p → poly-weak-mono p v
  shows poly-weak-mono-all p
unfolding poly-weak-mono-all-def
proof (intro allI impI)
  fix α β :: ('v,'a)assign
  assume all: ∀ x. α x ≥ β x
  assume pos: pos-assign β
  let ?ab = λ vs v. if (v ∈ set vs) then α v else β v

  { fix vs :: 'v list
    assume set vs ⊆ poly-vars p
    hence eval-poly (?ab vs) p ≥ eval-poly β p
    proof (induct vs)
      case Nil show ?case by (simp add: ge-refl)
    next
      case (Cons v vs)
      hence subset: set vs ⊆ poly-vars p and v: v ∈ poly-vars p by auto
      show ?case
      proof (rule ge-trans[OF mono[OF v, unfolded poly-weak-mono-def, rule-format]
      Cons(1)(OF subset)])
        show pos-assign (?ab vs) unfolding pos-assign-def
        proof
          fix x
          from pos[unfolded pos-assign-def] have beta: β x ≥ 0 by simp
          from ge-trans[OF all[rule-format] this] have alpha: α x ≥ 0
      qed

29
from alpha beta show ?ab vs x ≥ 0 by auto
qed

show (?ab (v # vs) v) ≥ (?ab vs v) using all ge-refl by auto
next
fix x
assume v ≠ x
thus (?ab (v # vs) x) = (?ab vs x) by simp
qed
qed

]\}

}\)

from this[of poly-vars-list p, unfolded poly-vars-list]

have eval-poly (λv. if v ∈ poly-vars p then α v else β v) p ≥ eval-poly β p by auto

also have eval-poly (λv. if v ∈ poly-vars p then α v else β v) p = eval-poly α p
by (rule eval-poly-vars, auto)

finally
show eval-poly α p ≥ eval-poly β p .

qed

lemma poly-weak-mono-all: fixes p :: ('a, 'v :: poly-carrier)poly

assumes p: poly-weak-mono-all p

shows poly-weak-mono p v

unfolding poly-weak-mono-def

proof (intro allI impI)

fix α β :: ('a, 'v)assign

assume all: ∀x. v ≠ x −→ α x = β x

assume pos: pos-assign β

assume v: α v ≥ β v

show eval-poly α p ≥ eval-poly β p

proof (rule p[unfolded poly-weak-mono-all-def, rule-format, OF - pos])

fix x

show α x ≥ β x

using v all ge-refl[of β x] by auto

qed

qed

lemma poly-weak-mono-all-pos:

fixes p :: ('v, 'a :: poly-carrier)poly

assumes pos-at-zero: eval-poly (λ w. 0) p ≥ 0

and mono: poly-weak-mono-all p

shows p ≥ p zero-poly

unfolding poly-ge-def zero-poly-def

proof (intro allI impI, simp)

fix α :: ('v, 'a)assign

assume pos: pos-assign α

show eval-poly α p ≥ 0

proof −

let ?id = λ w. poly-of (PVar w)

let ?z = λ w. zero-poly

30
have \( \text{poly-subst } ?\text{id } p \geq_p \text{poly-subst } ?z p \)
  by (rule \text{poly-weak-mono-all-E[OF mono]},
    \text{simp, simp add: poly-ge-def zero-poly-def pos-assign-def})

hence \( \text{eval-poly } \alpha (\text{poly-subst } ?\text{id } p) \geq \text{eval-poly } \alpha (\text{poly-subst } ?z p) \) (is - \( \geq \) \( ?\text{res} \))

  unfolding \text{poly-ge-def} using \text{pos by simp}

also have \( \text{?res } = \text{eval-poly } (\lambda w. \ 0) p \) by \( (\text{simp add: poly-subst zero-poly-def}) \)
also have \( \dots \geq 0 \) by \( (\text{rule pos-at-zero}) \)
finally show \( \text{thesis by } (\text{simp add: poly-subst}) \)
qed

context \text{poly-order-carrier}
begin


definition \text{poly-strict-mono } :: \('v', 'a\)\text{poly } \Rightarrow 'v \Rightarrow \text{bool}
  where
\text{poly-strict-mono } p \ v \equiv \forall \ (\alpha :: \('v', 'a\)\text{assign}) \beta. \ (\forall x. (v \neq x \rightarrow \alpha x = \beta x))
\rightarrow \text{pos-assign } \beta \rightarrow \alpha v > \beta v \rightarrow \text{eval-poly } \alpha p > \text{eval-poly } \beta p

lemma \text{poly-strict-mono-E: assumes } p \text{: poly-strict-mono } p \ v
  and \( \text{fgw} : \wedge w. \ v \neq w \Rightarrow f w = g w \)
  and \( g : \wedge w. \ g w \geq_p \text{zero-poly} \)
  and \( \text{fgv} : f v > p g v \)
  shows \( \text{poly-subst } f p >p \text{poly-subst } g p \)
  unfolding \text{poly-gt-def poly-subst}

proof \( (\text{intro allI impI, rule p[unfolded poly-strict-mono-def, rule-format]})) \)
  fix \( \alpha :: \('c', 'a\)\text{assign} \)
  show \( \text{pos-assign } \alpha \Rightarrow \text{eval-poly } \alpha (f v) > \text{eval-poly } \alpha (g v) \) using \text{fgv unfolding poly-gt-def by auto}

next
  fix \( \alpha :: \('c', 'a\)\text{assign} \)
  assume \( \alpha : \text{pos-assign } \alpha \)
  show \( \text{pos-assign } (\lambda v. \text{eval-poly } \alpha (g v)) \)
    unfolding \text{pos-assign-def}

proof
  fix \( x \)
  show \( \text{eval-poly } \alpha (g x) \geq 0 \)
    using \( g[\text{of } x] \) unfolding \text{poly-ge-def zero-poly-def using alpha by auto}

next
  fix \( \alpha :: \('c', 'a\)\text{assign} \) and \( x \)
  assume \( v : v \neq x \)
  show \( \text{pos-assign } \alpha \Rightarrow \text{eval-poly } \alpha (f x) = \text{eval-poly } \alpha (g x) \) using \text{fgw[OF v] unfolding poly-gt-def by auto}

lemma \text{poly-add-gt-mono: assumes } p1 >p p2 \ shows \text{poly-add } p1 q >p \text{poly-add } p2 q
  using \text{assms unfolding poly-gt-def by } (\text{auto simp: field-simps plus-gt-left-mono})
lemma poly-mult-gt-mono:
frees q :: ('v, 'a)poly
assumes gt: p1 > p p2 and mono: q ≥ p one-poly
shows poly-mult p1 q > p poly-mult p2 q
proof (unfold poly-gt-def, intro impI allI)
fix α :: ('v, 'a)assign
assume p: pos-assign α
with gt have gt: eval-poly α p1 > eval-poly α p2 unfolding poly-gt-def by simp
from mono p have one: eval-poly α q ≥ 1 unfolding poly-ge-def one-poly-def by auto
show eval-poly α (poly-mult p1 q) > eval-poly α (poly-mult p2 q)
using times-gt-mono[OF gt one] by simp
qed
end

2.6 Degree of polynomials

definition monom-degree :: 'v monom ⇒ nat where
monom-degree xps ≡ listsum (map snd xps)
definition poly-degree :: ('v, '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 ≡ listsum (map (λ mc. max 0 (snd mc)) p)

lemma monom-degree: eval-monom (λ -. x) m = x ^ monom-degree m
unfolding monom-degree-def
proof (induct m)
case Nil show ?case by simp
next
case (Cons mc m)
thus ?case by (cases mc, auto simp: power-add field-simps)
qed

lemma poly-coeff-sum: poly-coeff-sum p ≥ 0
unfolding poly-coeff-sum-def
proof (induct p)
case Nil show ?case by (simp add: ge-refl)
next
case (Cons mc p)
have (∑ mc←mc # p. max 0 (snd mc)) = max 0 (snd mc) + (∑ mc←p. max 0 (snd mc)) by auto
also have ... ≥ 0 + 0
by (rule ge-trans[OF plus-left-mono plus-right-mono[OF Cons]], auto)
finally show ?case by simp
qed
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 (induct p)
case Nil show ?case by (simp add: ge-refl poly-degree-def poly-coeff-sum-def)
next
case (Cons mc p)
  obtain m c where mc = (m,c) by force
  from gc-trans[OF x one-ge-zero] have x0: x ≥ 0 .
  have id1: eval-poly (λ-. x) (mc ≠ p) = x ^ monom-degree m * c + eval-poly (λ-. x) p unfolding mc by (simp add: monom-degree)
  have id2: poly-coeff-sum (mc ≠ p) * x ^ poly-degree (mc ≠ p) = x ^ max (monom-degree m) (poly-degree p) * (max 0 c) + poly-coeff-sum p * x ^ max (monom-degree m) (poly-degree p)
    unfolding poly-coeff-sum-def poly-degree-def by (simp add: field-simps)
  show poly-coeff-sum (mc ≠ p) * x ^ poly-degree (mc ≠ p) ≥ eval-poly (λ-. x) (mc ≠ p)
    unfolding id1 id2
    proof (rule ge-trans[OF plus-left-mono plus-right-mono])
      show x ^ max (monom-degree m) (poly-degree p) * max 0 c ≥ x ^ monom-degree m * c
        by (rule ge-trans[OF times-left-mono[OF pow-mono-exp] times-right-mono[OF pow-ge-zero]], insert x x0, auto)
      show poly-coeff-sum p * x ^ max (monom-degree m) (poly-degree p) ≥ eval-poly (λ-. x) p
        by (rule ge-trans[OF times-right-mono[OF poly-coeff-sum pow-mono-exp[OF x]] Cons], auto)
    qed
qed

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

by (rule ge-trans[OF gc-trans[OF times-left-mono[OF pow-ge-zero] gc-trans[OF x one-ge-zero]] c] times-right-mono[OF poly-coeff-sum pow-mono-exp[OF x d]] poly-degree[OF x]])

2.7 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 = 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

33
proof (cases List.extract (λ (n,-). m =m n) p)
  case None
  with assms have (c,q) = (0,p) unfolding poly-split-def by auto
  thus ?thesis unfolding eq-poly-def by auto
next
  case (Some res)
  obtain p1 mc p2 where res = (p1,mc,p2) by (cases res, auto)
  with extract- SomeE[OF Some[simplified this]] obtain a m' where p: p = p1
  @ (m',a) # p2 and m': m =m m' and res: res = (p1,(m',a),p2) by (cases mc, auto)
from Some res assms have c: c = a and q: q = p1 @ p2 unfolding poly-split-def
by auto
  show ?thesis unfolding eq-poly-def by (simp add: p c q eq-monom[OF m']
field-simps)
qed

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
using poly-split[OF assms] unfolding eq-poly-def by auto

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 m) q of
    None ⇒ False
    | Some (q1,(d),q2) ⇒ c = d ∧ check-poly-eq p
      (q1 @ q2))

lemma check-poly-eq: fixes p :: ('v,'a :: poly-carrier)poly
  assumes chk: check-poly-eq p q
  shows p =p q unfolding eq-poly-def
proof
  fix α
  from chk show eval-poly α p = eval-poly α q
proof (induct p arbitrary: q)
  case Nil
  thus ?case by auto
next
  case (Cons mc p)
  obtain m c where mc: mc = (m,c) by (cases mc, auto)
  show ?case
proof (cases List.extract (λ mc. fst mc =m m) q)
  case None
  with Cons(2) show ?thesis unfolding mc by simp
next
  case (Some res)
  obtain q1 md q2 where res = (q1,md,q2) by (cases res, auto)
  with extract- SomeE[OF Some[simplified this]] obtain m' d where q: q = q1
  @ (m',d) # q2 and m': m' =m m and res: res = (q1,(m',d),q2)

  34
by (cases md, auto)

from Cons(2) Some mc res have rec: check-poly-eq p (q1 @ q2) and c: c = d by auto

from Cons(1)[OF rec] have p: eval-poly α p = eval-poly α (q1 @ q2).

show ?thesis unfolding mc eval-poly_simps c p q using eq-monom[OF m’,
of α] by (simp add: ac-simps)

qed

qed

qed

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 m) q of
        None ⇒ c ≥ 0 ∧ check-poly-ge p q
    | Some (q1,(-d),q2) ⇒ c ≥ d ∧ check-poly-ge p

lemma check-poly-ge: fixes p :: (’v, ’a :: poly-carrier)poly

shows check-poly-ge p q ⇒ p ≥p q

proof (induct p arbitrary: q)

case Nil

hence ∀ (n,d) ∈ set q. 0 ≥ d using list-all-iff[of - q] by auto

hence [] ≥p q

proof (induct q)

case Nil thus ?case by (simp)

next

case (Cons nd q)

hence rec: [] ≥p q by simp

show ?case

proof (cases nd)

case (Pair n d)

with Cons have ge: 0 ≥ d by auto

show ?thesis

proof (simp only: Pair, unfold poly-ge-def, intro allI impI)

fix α :: (’v, ’a)assign

assume pos: pos-assign α

have ge: 0 ≥ eval-monom α n ∗ d

using times-right-mono[OF pos-assign-monom[OF pos, of n] ge] by simp

from rec[unfolded poly-ge-def] pos have ge2: 0 ≥ eval-poly α q by auto

show eval-poly α [] ≥ eval-poly α ((n,d) # q) using ge-trans[OF plus-left-mono[OF ge]
    plus-right-mono[OF ge2]]

    by simp

qed

qed

qed

thus ?case by simp
next

\begin{itemize}
  \item \textbf{case (Cons mc p)}
  \item \textbf{obtain }m \ c \ \textbf{where }mc = (m,c) \ \textbf{by (cases mc, auto)}
  \item \textbf{show }?\textbf{case (cases List.extract (λ mc. fst mc = m) q)}
  \item \textbf{proof (simp only: mc, unfold poly-ge-def, intro allI impI)}
  \item \textbf{fix }α :: ('v,'a)\textbf{assign}
  \item \textbf{assume }pos: pos-assign α
  \item \textbf{have }ge: eval-monom α m * c ≥ 0
  \item \textbf{using times-right-mono[OF pos-assign-monom[OF pos, of m] c] by simp}
  \item \textbf{from rec have pq: eval-poly α p ≥ eval-poly α q unfolding poly-ge-def using}
  \item \textbf{pos by auto}
  \item \textbf{show eval-poly α ((m,c) # p) ≥ eval-poly α q}
  \item \textbf{using ge-trans[OF plus-left-mono[OF ge] plus-right-mono[OF pq]] by simp}
\end{itemize}

\textbf{qed}

\textbf{next}

\begin{itemize}
  \item \textbf{case (Some res)}
  \item \textbf{obtain }q1 \ md \ q2 \ \textbf{where }res = (q1,md,q2) \ \textbf{by (cases res, auto)}
  \item \textbf{with extract-SomeE[OF Some[simplified this]] obtain }m' \ d \ \textbf{where }q: q = q1 \ @ (m',d) \ # q2 \ \textbf{and }m': m' = m \ m \ \textbf{and res: res = (q1,(m',d),q2)}
  \item \textbf{by (cases md, auto)}
  \item \textbf{from Cons(2) Some mc res have rec: check-poly-ge p (q1 @ q2) and c: c ≥ d by auto}
  \item \textbf{from Cons(1)[OF rec] have }p: p ≥ p q1 \ @ q2 .
  \item \textbf{show }?\textbf{thesis}
  \item \textbf{proof (simp only: mc, unfold poly-ge-def, intro allI impI)}
  \item \textbf{fix }α :: ('v,'a)\textbf{assign}
  \item \textbf{assume }pos: pos-assign α
  \item \textbf{have }ge: eval-monom α m * c ≥ 0
  \item \textbf{using times-right-mono[OF pos-assign-monom[OF pos, of m] c] by simp}
  \item \textbf{from p have ge2: eval-poly α p ≥ eval-poly α (q1 @ q2) unfolding poly-ge-def using}
  \item \textbf{pos by auto}
  \item \textbf{show eval-poly α ((m,c) # p) ≥ eval-poly α q using ge-trans[OF plus-left-mono[OF ge] plus-right-mono[OF ge2]] by simp add: q field-simps}
\end{itemize}

\textbf{qed}

\textbf{qed}

\textbf{declare check-poly-ge.simps[simp del]}

\textbf{definition check-poly-weak-mono-all :: ('v,'a :: ordered-semiring-0)poly ⇒ bool}
\textbf{where check-poly-weak-mono-all p ≡ list-all (λ (m,c). c ≥ 0) p}
lemma check-poly-weak-mono-all: fixes p :: ('v,'a :: poly-carrier)poly
  assumes check-poly-weak-mono-all p shows poly-weak-mono-all p
unfolding poly-weak-mono-all-def
proof (intro allI impI)
  fix f g :: ('v,'a)assign
  assume fg: \forall x. f x \geq g x
  and pos: pos-assign g
  hence fg: \forall x. f x \geq g x by auto
  from pos[unfolded pos-assign-def] have g: \forall x. g x \geq 0 ..
from assms have \\\exists m c. (m,c) \in set p \Longrightarrow c \geq 0 unfolding check-poly-weak-mono-all-def
by (auto simp: list-all-iff)
  thus eval-poly f p \geq eval-poly g p
proof (induct p)
  case Nil thus \thesis by (simp add: ge-refl)
next
  case (Cons mc p)
  hence IH: eval-poly f p \geq eval-poly g p by auto
  show \thesis
  proof (cases mc)
  case (Pair m c)
  with Cons have c: c \geq 0 by auto
  show \thesis unfolding eval-poly.simps fst-conv snd-conv
  proof (rule ge-trans [OF plus-left-mono [OF times-left-mono [OF c] ] ] plus-right-mono[OF IH]])
  show eval-monom f m \geq eval-monom g m
  by (rule eval-monom-mono[1] [OF fg g])
  qed
  qed
  qed
qed

lemma check-poly-weak-mono-all-pos:
  assumes check-poly-weak-mono-all p shows p \geq p zero-poly
unfolding zero-poly-def
proof (rule check-poly-ge)
from assms have \\\exists m c. (m,c) \in set p \Longrightarrow c \geq 0 unfolding check-poly-weak-mono-all-def
by (auto simp: list-all-iff)
thus check-poly-ge p []
  by (induct p, simp add: check-poly-ge.simps, clarify, auto simp: check-poly-ge.simps extract-Nil-code)
qed

better check for weak monotonicity for discrete carriers: p is monotone in v if p(...v + 1...) \geq p(...v...)

definition check-poly-weak-mono-discrete :: ('v,'a :: poly-carrier)poly \Rightarrow 'v \Rightarrow bool
  where check-poly-weak-mono-discrete p v \equiv check-poly-ge (poly-subst (\lambda 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 \Rightarrow ('v,'a :: poly-carrier)poly \Rightarrow
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,'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)

lemma check-poly-weak-mono-discrete:
  fixes v :: 'v and p :: ('v,'a)poly
  assumes discrete and check: check-poly-weak-mono-discrete p v
  shows poly-weak-mono p v
unfolding poly-weak-mono-def
proof (intro allI impI)
  fix f g :: ('v,'a)assign
  assume fgw: ∀ w. (v ≠ w −→ f w = g w)
  and gass: pos-assign g
  and v: f v ≥ g v
  from fgw have w: ∀ w. v ≠ w −→ f w = g w by auto
  from assms check-poly-ge have ge: poly-ge (poly-subst (λ w. poly-of (if w = v then PSum [PNum 1, PVar v] else PVar w)) p) p (is poly-ge ?p1 p) unfolding
  check-poly-weak-mono-discrete-def by blast
  from discrete[OF (discrete) v] obtain k' where id: f v = ((op + 1) ^^ k') (g v)
  by auto
  show eval-poly f p ≥ eval-poly g p
  proof (cases k')
  case 0
    { fix x
      have f x = g x using id 0 w by (cases x = v, auto)
    }
    hence f = g ..
    thus ?thesis using ge-refl by simp
  next
  case (Suc k)
  with id have f v = ((op + 1) ^^ (Suc k)) (g v) by simp
  with w gass show eval-poly f p ≥ eval-poly g p
  proof (induct k arbitrary: f g rule: less-induct)
  case (less k)
  show ?case
  proof (cases k)
  case 0
    with less have id0: f v = 1 + g v by simp

38
have $id1$: eval-poly $f$ ?$p1$ = eval-poly $g$ ?$p1$

proof (rule eval-poly-subst)
fix $w$
show $f$ $w$ = eval-poly $g$ (poly-of (if $w$ = $v$ then $P$Sum $t$ $v$ $v$ $v$ $v$ else $P$Var $w$))
proof (cases $w$ = $v$)
  case True
  show ?thesis by (simp add: True id0 zero-poly-def)
next
  case False
  with less have $f$ $w$ = $g$ $w$ by simp
  thus ?thesis by (simp add: False)
qed
qed

have eval-poly $g$ ?$p1$ ≥ eval-poly $g$ $p$ using ge less unfolding poly-ge-def by simp
with $id1$ show ?thesis by simp
next
  case (Suc $kk$)
  obtain $g'$ where $g'$: $g'$ = ($\lambda$ $w$. if ($w$ = $v$) then $1$ + $g$ $w$ else $g$ $w$) by auto
  have $1$ ($\cdot$ $'a$) + $g$ $v$ ≥ $1$ + $0$
    by (rule plus-right-mono, simp add: less(3)[unfolded pos-assign-def])
  also have $1$ + ($0$ ($\cdot$ $'a$) = $1$ by simp
  also have ... ≥ $0$ by (rule one-ge-zero)
  finally have $g'$pos: pos-assign $g'$ using less(3) unfolding pos-assign-def by (simp add: $g'$)
  {
    fix $w$
    assume $v$ ≠ $w$
    hence $f$ $w$ = $g'$ $w$
      unfolding $g'$ by (simp add: less)
  }
  note $w$ = this
  have eq: $f$ $v$ = ($o$ $p$ + ($1$ ($\cdot$ $'a$) ^^ Suc $kk$) ($('a$) $v$))
    by (simp add: less(4) $g'$ Suc, rule arg-cong[where $f$ = $o$ $p$ + $1$], induct
kk, auto)
  from Suc have $kk$: $kk$ < $k$ by simp
  from less(1)[OF $kk$ $w$ $g'$pos] eq
  have rec1: eval-poly $f$ $p$ ≥ eval-poly $g'$ $p$ by simp
  {
    fix $w$
    assume $v$ ≠ $w$
    hence $g'$ $w$ = $g$ $w$
      unfolding $g'$ by simp
  }
  note $w$ = this
  from Suc have $z$: $0$ < $k$ by simp
  from less(1)[OF $z$ $w$ less(3)] $g'$
  have rec2: eval-poly $g'$ $p$ ≥ eval-poly $g$ $p$ by simp
  show ?thesis by (rule ge-trans[OF rec1 rec2])
qed
lemma check-poly-weak-anti-mono-discrete:

fixes v :: 'v and p :: ('v, 'a)poly

assumes discrete and check: check-poly-weak-anti-mono-discrete p v

shows poly-weak-anti-mono p v

unfolding poly-weak-anti-mono-def

proof (intro allI impI)

fix f g :: ('v, 'a)assign

assume fgw: ∀ w. (v ≠ w → f w = g w)

and gass: pos-assign g

and v: f v ≥ g v

from fgw have w: ∀ w. v ≠ w → f w = g w by auto

from assms check-poly-ge have ge: poly-ge p (poly-subst (λ w. poly-of (if w = v then PSum [PNum 1, PVar v] else PVar w)) p) (is poly-ge p ?p1) unfolding check-poly-weak-anti-mono-discrete-def by blast

from discrete[OF ⟨discrete v⟩] obtain k' where id: f v = ((op + 1) ^ k') (g v)

by auto

show eval-poly g p ≥ eval-poly f p

proof (cases k')

case 0

{ fix x
  have f x = g x using id 0 w by (cases x = v, auto)
}

hence f = g ..

thus ?thesis using ge-refl by simp

next

case (Suc k)

with id have f v = ((op + 1) ^ Suc k) (g v) by simp

with w gass show eval-poly g p ≥ eval-poly f p

proof (induct k arbitrary: f g rule: less-induct)

case (less k)

show ?case

proof (cases k)

case 0

with less have id0: f v = 1 + g v by simp

have id1: eval-poly f p = eval-poly g ?p1

proof (rule eval-poly-subst)

fix w

show f w = eval-poly g (poly-of (if w = v then PSum [PNum 1, PVar v] else PVar w))

proof (cases w = v)

case True

show ?thesis by (simp add: True id0 zero-poly-def)

next

case False

qed

qed

qed
with less have \( f \cdot w = g \cdot w \) by simp
thus \( \text{thesis} \) by (simp add: False)
qed
qed

have eval-poly \( g \cdot p \geq \) eval-poly \( g \cdot \text{?p1} \) using ge less unfolding poly-ge-def by simp
with id1 show \( \text{thesis} \) by simp

next

case (Suc \( kk \))

obtain \( g' \) where \( g' = (\lambda w. \text{if } (w = v) \text{ then } 1 + g \cdot w \text{ else } g \cdot w) \) by auto
have \( (1 :: 'a) + g \cdot v \geq 1 + 0 \)
  by (rule plus-right-mono, simp add: less(3)[unfolded pos-assign-def])
also have \( (1 :: 'a) + 0 = 1 \) by simp
also have \( \ldots \geq 0 \) by (rule one-ge-zero)
finally have \( g'\text{pos} : \text{pos-assign} \) \( g' \) using less(3) unfolding pos-assign-def
  by (simp add: \( g' \))

\{
  fix \( w \)
  assume \( v \neq w \)
  hence \( f \cdot w = g' \cdot w \)
  unfolding \( g' \) by (simp add: less)
\}

note \( w = \text{this} \)

have eq: \( f \cdot v = (\text{op } + (1 :: 'a) \text{ } ^\wedge \text{Suc} \cdot kk) \cdot ((g' \cdot v)) \)
  by (simp add: less(4) \( g' \cdot \text{Suc} \), rule arg-cong[where \( f = \text{op } + 1 \]), induct \( kk, \text{auto} \))

from Suc have \( kk \cdot kk < k \) by simp
from less(1)[OF \( kk \cdot w \cdot g'\text{pos} \)] eq
have rec1: \( \text{eval-poly} \) \( g' \cdot p \geq \text{eval-poly} \) \( f \cdot p \) by simp

\{
  fix \( w \)
  assume \( v \neq w \)
  hence \( g' \cdot w = g \cdot w \)
  unfolding \( g' \) by simp
\}

note \( w = \text{this} \)

from Suc have \( z : 0 < k \) by simp
from less(1)[OF \( \text{z} \cdot w \cdot \text{less}(3) \)] \( g' \)
have rec2: \( \text{eval-poly} \) \( g \cdot p \geq \text{eval-poly} \) \( g' \cdot p \) by simp
show \( \text{thesis} \) by (rule ge-trans[OF rec2 rec1])

qed
qed
qed

lemma \text{check-poly-weak-mono-and-pos}:
  \( \text{fixes } \cdot p :: ('v', 'a)\text{poly} \)
  \( \text{assumes } \text{check-poly-weak-mono-and-pos } \cdot p \) discrete \( p \)
  \( \text{shows } \text{poly-weak-mono-all } \cdot p \land (\cdot p \geq \cdot p \text{ } \text{zero-poly}) \)
proof (cases discrete)

  case False

  lemma \text{check-poly-weak-mono-and-pos}:
with assms have c: check-poly-weak-mono-all p unfolding check-poly-weak-mono-and-pos-def 
by auto
from check-poly-weak-mono-all[OF c] check-poly-weak-mono-all-pos[OF c] show 
thesis by auto
next
  case True
  with assms have c: list-all (λ v. check-poly-weak-mono-discrete p v) (poly-vars-list 
p) and g: eval-poly (λ w. 0) p ≥ 0
  unfolding check-poly-weak-mono-and-pos-def by auto
  have m: poly-weak-mono-all p
  proof (rule poly-weak-mono)
    fix v :: 'v
    assume v: v ∈ poly-vars p
    show poly-weak-mono p v
    by (rule check-poly-weak-mono-discrete[OF True], insert c[unfolded list-all-iff] 
v, auto)
  qed
  have m': poly-weak-mono-all p
  proof (rule poly-weak-mono)
    fix v :: 'v
    assume v: v ∈ poly-vars p
    show poly-weak-mono p v
    by (rule check-poly-weak-mono-discrete[OF True], insert c[unfolded list-all-iff] 
v, auto)
  qed
  from poly-weak-mono-all-pos[OF g m'] m show thesis by auto
  qed
end

lemma monom-vars-eval-monom: [∀ x. x ∈ monom-vars m ⇒ f x = g x] ⇒ 
eval-monom f m = eval-monom g m
by (induct m, auto)
definition check-poly-weak-mono :: (′v,′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,′a :: poly-carrier)poly
  assumes check-poly-weak-mono p v shows poly-weak-mono p v
unfolding poly-weak-mono-def
proof (intro allI impI)
  fix f g :: (′v,′a)assign
  assume ∀ x. v ≠ x ⇒ f x = g x
  and pos: pos-assign g
  and ge: f v ≥ g v
  hence fg: ∀ x. v ≠ x ⇒ f x = g x by auto
from pos[unfolded pos-assign-def] have g: [∀ x. g x ≥ 0 .. 
from assms have ∃ m c. (m,c) ∈ set p ⇒ c ≥ 0 ∨ v ∉ monom-vars m

42
unfolding check-poly-weak-mono-def by (auto simp: list-all-iff)
thus eval-poly f p ≥ eval-poly g p
proof (induct p)
case Nil thus ?case by (simp add: ge-refl)
next
case (Cons mc p)
hence IH: eval-poly f p ≥ eval-poly g p by auto
obtain m c where mc: mc = (m,c) by force
with Cons have c: c ≥ 0 ∨ v /∈ monom-vars m by auto
show ?case unfolding mc eval-poly.simps fst-conv snd-conv
proof (rule ge-trans[OF plus-left-mono plus-right-mono[OF IH]])
from c show eval-monom f m * c ≥ eval-monom g m * c
proof
  assume c: c ≥ 0
  show ?thesis
  proof (rule times-left-mono[OF c], rule eval-monom-mono(1)[OF - g])
    fix x
    show f x ≥ g x using ge fg[of x] by (cases x = v, auto simp: ge-refl)
  qed
next
  assume v: v ∈ monom-vars m
  have eval-monom f m = eval-monom g m
    by (rule monom-vars-eval-monom, insert fg v, auto)
  thus ?thesis by (simp add: ge-refl)
  qed
 qed
qed

definition check-poly-weak-mono-smart :: bool ⇒ (′v,′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,′a :: poly-carrier)poly
shows check-poly-weak-mono-smart discrete p v ⇒ poly-weak-mono p v
unfolding check-poly-weak-mono-smart-def
using check-poly-weak-mono check-poly-weak-mono-discrete by (cases discrete, auto)

definition check-poly-weak-anti-mono :: (′v,′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,′a :: poly-carrier)poly
assumes check-poly-weak-anti-mono p v shows poly-weak-anti-mono p v
unfolding poly-weak-anti-mono-def
proof (intro allI impI)
fix f g :: ('v, 'a)assign
assume \( \forall \ x. \ v \neq x \rightarrow f \ x = g \ x \)
and pos: pos-assign g
and ge: \( f \ v \geq g \ v \)
hence \( f g: \text{\( \forall \ x. \ v \neq x \rightarrow f \ x = g \ x \) by auto} \)
from pos[unfolded pos-assign-def] have g: \( \text{\( \forall \ x. \ g \ x \geq 0 \) ..} \)
from assms have \( \text{\( \forall m. \ c. \ (m, c) \in \text{set} \ p \rightarrow 0 \geq c \lor v \notin \text{monom-vars} \ m \) }
unfolding check-poly-weak-anti-mono-def by (auto simp: list-all-iff)
thus \( \text{\( \text{eval-poly} \ g \ p \geq \text{eval-poly} \ f \ p \) }
proof (induct p)
case Nil thus \(?case \ by \ (\text{simp add: ge-refl}) \)
next
case (Cons mc p)
hence IH: \( \text{\( \text{eval-poly} \ g \ p \geq \text{eval-poly} \ f \ p \) by auto} \)
obtain m c where mc: \( \text{\( m = (m, c) \) by force} \)
with Cons have c: \( \text{\( 0 \geq c \lor v \notin \text{monom-vars} \ m \) by auto} \)
show ?case unfolding mc eval-poly.simps fst-conv snd-conv
proof (rule ge-trans[OF plus-left-mono plus-right-mono[OF IH]])
from c show \( \text{\( \text{eval-monom} \ g \ m \ast c \geq \text{eval-monom} \ f \ m \ast c \) }
proof
  assume c: \( \text{\( 0 \geq c \) }
  show \(?thesis \ proof (rule times-left-anti-mono[OF eval-monom-mono(1)][OF - g c])
  fix x
  show \( f x \geq g x \) using ge fg[of x] by (cases x = v, auto simp: ge-refl)
qed
next
assume v: \( v \notin \text{monom-vars} \ m \)
have \( \text{\( \text{eval-monom} \ f \ m = \text{eval-monom} \ g \ m \) }
by (rule monom-vars-eval-monom, insert fg v, auto)
thus \(?thesis \ by \ (\text{simp add: ge-refl}) \)
qed
qed
qed

definition check-poly-weak-anti-mono-smart :: bool \( \rightarrow \ ('v, 'a :: \text{poly-carrier})\text{poly} \)
\( \Rightarrow \ 'v \Rightarrow \ bool \)
where check-poly-weak-anti-mono-smart discrete \( \equiv \) 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: \( \text{fixes} \ p :: ('v, 'a :: \text{poly-carrier})\text{poly} \)
shows \( \text{check-poly-weak-anti-mono-smart discrete p v \( \Rightarrow \) \text{poly-weak-anti-mono} p v} \)
unfolding check-poly-weak-anti-mono-smart-def
by (cases discrete, auto)

**definition** check-poly-gt :: ('a ⇒ 'a ⇒ bool) ⇒ ('v, 'a :: ordered-semiring-0)poly ⇒ ('v, 'a)poly ⇒ bool

**where** check-poly-gt gt p q ≡ let (a1, p1) = poly-split [] p; (b1, q1) = poly-split [] q in gt a1 b1 ∧ check-poly-ge p1 q1

**definition** check-monom-strict-mono :: bool ⇒ 'v monom ⇒ 'v ⇒ bool

**where** check-monom-strict-mono pm m v ≡ pm ∧ tl m = [] ∧ fst (hd m) = v ∧ (λ p. if pm then 1 ≤ p else p = 1) (snd (hd m))

**definition** check-poly-strict-mono :: bool ⇒ ('v, '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-smart :: bool ⇒ bool ⇒ ('a :: poly-carrier ⇒ 'a ⇒ bool) ⇒ ('v, '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, '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** (cases power-mono)

**case** False

**with** check obtain n where m = [(v,1)] unfolding check-monom-strict-mono-def by (cases m, cases tl m, cases hd m, auto)

**with** gt show ?thesis by (auto)

**next**

**case** True

**with** check obtain n where m = [(v,n)] and n: 1 ≤ n unfolding check-monom-strict-mono-def by (cases m, cases tl m, cases hd m, auto)

**from** power-mono[OF True gt ge n] m show ?thesis by (auto)

**qed**

**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

unfolding poly-strict-mono-def

proof (intro allI impI)
  fix f g :: ('b,'a)assign
  assume fgv: ∀ w. (v ≠ w → f w = g w)
  and pos: pos-assign g
  and fgv: f v ≥ g v
  from pos[unfolded pos-assign-def] have g: ∀ x. g x ≥ 0 ..
  { fix w
    have f w ≥ g w
    proof (cases v = w)
      case False
      with fgw ge-refl show ?thesis by auto
    next
      case True
      from fgv[unfolded True] show ?thesis by (rule gt-imp-ge)
    qed
  } note fgw2 = this
  let ?e = eval-poly
  show ?e f p ≻ ?e g p
  using check1[unfolded check-poly-strict-mono-def, simplified list-ex-iff]
  check2[unfolded check-poly-weak-mono-all-def, simplified list-all-iff, THEN
  bspec]
  proof (induct p)
    case Nil thus ?case by simp
  next
    case (Cons mc p)
    obtain m c where mc: mc = (m,c) by (cases mc, auto)
    show ?case
    proof (cases c ≥ 1 ∧ check-monom-strict-mono power-mono m v)
      case True
      hence c: c ≥ 1 and m: check-monom-strict-mono power-mono m v by blast+
      from times-gt-mono[OF check-monom-strict-mono[OF m, of f g, OF fgv g] c]
      show gt: eval-monom f m * c ≻ eval-monom g m * c .
      from Cons(3) have check-poly-weak-mono-all p unfolding check-poly-weak-mono-all-def
      list-all-iff by auto
      from check-poly-weak-mono-all[OF this, unfolded poly-weak-mono-all-def, rule-format, OF fgw2 pos]
      have ge: ?e f p ≥ ?e g p .
      from compat2[OF plus-gt-left-mono[OF gt] plus-right-mono[OF ge]]
      show ?thesis unfolding mc by simp
    next
    case False
    with Cons(2) mc have ∃ mc ∈ set p. (λ (m,c). c ≥ 1 ∧ check-monom-strict-mono
    power-mono m v) mc by auto
    from Cons(1)[OF this] Cons(3) have rec: ?e f p ≻ ?e g p by simp
    from Cons(3) mc have c: c ≥ 0 by auto

  }
from times-left-mono[OF c eval-monom-mono(1)[OF fgw2 g]]
have ge: eval-monom f m * c ≥ eval-monom g m * c .
from compat2[OF plus-gt-left-mono[OF rec] plus-right-mono[OF ge]]
show ?thesis by (simp add: mc field-simps)
qed
qed
qed

lemma check-poly-gt:
fixes p :: ('v', 'a)poly
assumes check-poly-gt gt p q shows p >p q
proof –
  obtain a1 p1 where p: poly-split [] p = (a1, p1) by (cases poly-split [] p, auto)
  obtain b1 q1 where q: poly-split [] q = (b1, q1) by (cases poly-split [] q, auto)
from p q assms have gt: a1 > b1 and ge: p1 ≥ p q1 unfolding check-poly-gt-def
using check-poly-ge[of p1 q1] by auto
show ?thesis proof (simp add: poly-split[OF p1 q1], unfold poly-gt-def)
fix α :: ('v', 'a)assign
  assume pos-assign α
  with ge have ge: eval-poly α p1 ≥ eval-poly α q1 unfolding poly-ge-def by simp
  from plus-gt-left-mono[OF gt] compat[OF plus-left-mono[OF ge]] have gt: a1 + eval-poly α p1 > b1 + eval-poly α q1 by (force simp: field-simps)
  show eval-poly α p > eval-poly α q
qed
qed

lemma check-poly-strict-mono-discrete:
fixes v :: 'v and p :: ('v', 'a)poly
assumes discrete and check: check-poly-strict-mono-discrete gt p v
shows poly-strict-mono p v
unfolding poly-strict-mono-def
proof (intro allI impI)
  fix f g :: ('v', 'a)assign
  assume fgw: ∀ w. (v ≠ w → f w = g w)
  and gass: pos-assign g
  and w: f v > g v
  from gass have g: ∀ x. g x ≥ 0 unfolding pos-assign-def ..
  from fgw have w: ∀ w. v ≠ w → f w = g w by auto
  from assms check-poly-gt have gt: poly-gt (poly-subst (λ w. poly-of (if w = v
then PSum [PNum 1, PVar v] else PVar w)) p) p (is poly-gt ?p1 p) unfolding
check-poly-strict-mono-discrete-def by blast
  from discrete[OF (discrete) gt-imp-ge[OF v]] obtain k' where id: f v = ((op +
1) k') (g v) by auto
  {
assume \( k' = 0 \)

from \( \text{v[unfolded id this]} \) have \( g \; v \preceq \; g \; v \) by simp

hence \( \text{False using SN \; g[of \; v]} \) unfolding SN-defs by auto

\}

with \( \text{id \; obtain \; k \; where \; id: \; f \; v = ((op + 1) \; ^{\prime} \; \text{Suc} \; k) \; (g \; v) \; by \; (cases \; k', \; auto)} \)

with \( \text{w \; gass} \)

show \( \text{eval-poly} \; f \; p \; \succ \; \text{eval-poly} \; g \; p \)

proof (induct \( k \) arbitrary: \( f \; g \) rule: less-induct)

\text{case \( (\text{less} \; k) \)

show \( ?\text{case} \)

proof (cases \( k \))

\text{case \( 0 \)

with \( \text{less}(4) \) have \( \text{id0}: \; f \; v = 1 + g \; v \) by simp

have \( \text{id1: \; eval-poly} \; f \; p = \text{eval-poly} \; g \; ?p1 \)

proof (rule eval-poly-subst)

fix \( w \)

show \( f \; w = \text{eval-poly} \; g \; (\text{poly-of} \; (\text{if} \; w = v \; \text{then} \; \text{PSum} \; [\text{PNum} \; 1, \; \text{PVar} \; v] \)

\text{else} \; \text{PVar} \; w)) \)

proof (cases \( w = v \))

\text{case \( \text{True} \)

\text{show} \( ?\text{thesis} \) by (simp add: True \text{id0} \text{zero-poly-def})

next

\text{case \( \text{False} \)

with \( \text{less} \) have \( f \; w = g \; w \) by simp

thus \( ?\text{thesis} \) by (simp add: False)

qed

qed

have \( \text{eval-poly} \; g \; ?p1 \; \succ \; \text{eval-poly} \; g \; p \) using \text{gt \; less \; unfolding} \; \text{poly-gt-def \; by \; simp}

with \( \text{id1 \; show} \; ?\text{thesis \; by \; simp} \)

next

\text{case \( (\text{Suc} \; kk) \)

obtain \( g' \) where \( g': \; g' = (\lambda \; w. \; \text{if} \; (w = v) \; \text{then} \; 1 + g \; w \) \) else \( g \; w \) \) by auto

have \( (1 :: \; 'a) + g \; v \geq 1 + 0 \)

by (rule plus-right-mono, simp add: less(3)[unfolded pos-assign-def])

also have \( (1 :: \; 'a) + 0 = 1 \) by simp

also have \( \ldots \geq 0 \) by (rule one-ge-zero)

finally have \( g' \text{pos: \; pos-assign} \; g' \) using \text{less(3) \; unfolding} \; \text{pos-assign-def}

by (simp add: \( g' \))

\}

fix \( w \)

assume \( v \neq w \)

hence \( f \; w = g' \; w \)

unfolding \( g' \) by (simp add: less)

\}

note \( w = \text{this} \)

have \( eq: \; f \; v = (op + (1 :: \; 'a) \; ^{\prime} \; \text{Suc} \; kk) \; ((g' \; v)) \)

by (simp add: less(4) \; g' \; \text{Suc, \; rule \; arg-cong[where \; f = \; op + 1, \; induct \; kk, \; auto]} \)

from \( \text{Suc \; have \; kk: \; kk < k \; by \; simp} \)

48
from less(1)(OF kk w g'pos] eq
have rec1: eval-poly f p ≻ eval-poly g' p by simp
{
  fix w
  assume v ≠ w
  hence g' w = g w
  unfolding g' by simp
} note w = this
from Suc have z: 0 < k by simp
from less(1)(OF z w less(3)] g'
have rec2: eval-poly g' p ≻ eval-poly g p by simp
show ?thesis by (rule gt-trans[OF rec1 rec2])
qed
qed

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 (cases discrete)
case True
  with check1[unfolded check-poly-strict-mono-smart-def]
  check-poly-strict-mono-discrete[OF True]
  show ?thesis by auto
next
case False
  from check-poly-strict-mono[OF check1[unfolded check-poly-strict-mono-smart-def, simplified False, simplified]]
  check2[unfolded check-poly-weak-mono-and-pos-def, simplified False, simplified]
  show ?thesis by auto
qed

end

3 Monotonicity criteria of Neurauter, Zankl, and
Middeldorp

theory NZM
imports ../Abstract−Rewriting/SN-Order-Carrier Polynomial
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** assumes $b \colon b + a > \theta$ and $a \colon (a :: int) \geq \theta$
**shows** check-poly-strict-mono-discrete (op $>$) (poly-of (PSum [PNum c, PMult [PNum b, PVar x]], PMult [PNum a, PVar x, PVar x]])) $x$
**proof** (cases $a = 0$)
  **case** True
  **with** $b$ have $b \colon b > \theta \land b \neq 0$ by auto
  **show** ?thesis using $b$ True
  **next**
  **case** False
  **show** ?thesis using False $a$ $b$
  qed

**lemma** assumes $b \colon b + a \geq \theta$ and $a \colon (a :: int) \geq \theta$
**shows** check-poly-weak-mono-discrete (op $\geq$) (poly-of (PSum [PNum c, PMult [PNum b, PVar x]], PMult [PNum a, PVar x, PVar x]])) $x$
**proof** (cases $a = 0$)
  **case** True
  **with** $b$ have $b \colon 0 \leq b$ by auto
  **show** ?thesis using $b$ True
  **next**
  **case** False
  **show** ?thesis using False $a$ $b$
  qed

end

References

[1] D. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA,
1979.

