Executable multivariate polynomials

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

March 17, 2025

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 ⊆ f ‘ A
  obtains C where C ⊆ A and B = f ‘ C and inj-on f C
proof –
  define g where g = (λx. SOME a. a ∈ A ∧ f a = x)
  have g b ∈ A ∧ f (g b) = b if b ∈ B for b
  proof –
    from that assms have b ∈ f ‘ A ..
    then obtain a where a ∈ A and b = f a ..
    hence a ∈ A ∧ f a = b by simp
    thus ?thesis unfolding g-def by (rule someI)
  qed
  hence 1: ⋀b. b ∈ B ⟹ g b ∈ A and 2: ⋀b. b ∈ B ⟹ f (g b) = b by simp-all
  let ?C = g ‘ B
  show ?thesis
  proof
    show ?C ⊆ A by (auto intro: 1)
  next
    show B = f ‘ ?C
```

proof (*rule set-eqI*)
fix *b*
show $b \in B \longleftrightarrow b \in f$ ‘ *?C*
proof
assume $b \in B$
moreover from *this* **have** $f\ (g\ b) = b$ **by** (*rule 2*)
ultimately show $b \in f$ ‘ *?C* **by** *force*
next
assume $b \in f$ ‘ *?C*
then obtain b' **where** $b' \in B$ **and** $b = f\ (g\ b')$ **unfolding** *image-image* **..**
moreover from *this*(*1*) **have** $f\ (g\ b') = b'$ **by** (*rule 2*)
ultimately show $b \in B$ **by** *simp*
qed
qed
next
show *inj-on f ?C*
proof
fix *x y*
assume $x \in$ *?C*
then obtain *bx* **where** $bx \in B$ **and** *x*: $x = g\ bx$ **..**
moreover from *this*(*1*) **have** $f\ (g\ bx) = bx$ **by** (*rule 2*)
ultimately have $*$: $f\ x = bx$ **by** *simp*
assume $y \in$ *?C*
then obtain *by* **where** $by \in B$ **and** *y*: $y = g\ by$ **..**
moreover from *this*(*1*) **have** $f\ (g\ by) = by$ **by** (*rule 2*)
ultimately have $f\ y = by$ **by** *simp*
moreover assume $f\ x = f\ y$
ultimately have $bx = by$ **using** $*$ **by** *simp*
thus $x = y$ **by** (*simp only*: *x y*)
qed
qed
qed

lemma *wfP-chain*:
assumes $\neg(\exists f.\ \forall i.\ r\ (f\ (Suc\ i))\ (f\ i))$
shows *wfP r*
proof −
from *assms wf-iff-no-infinite-down-chain*[*of* $\{(x, y).\ r\ x\ y\}$] **have** *wf* $\{(x, y).\ r\ x\ y\}$ **by** *auto*
thus *wfP r* **unfolding** *wfp-def* **.**
qed

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 −
have $\bigwedge k.\ r\ (seq\ (i + Suc\ k))\ (seq\ i)$
proof −
fix *k*::*nat*

```
    show r (seq (i + Suc k)) (seq i)
    proof (induct k)
      case 0
      from assms(2) have r (seq (Suc i)) (seq i) .
      thus ?case by simp
    next
      case (Suc k)
      note assms(1)
      moreover from assms(2) have r (seq (Suc (Suc i + k))) (seq (Suc (i + k)))
by simp
      moreover have r (seq (Suc (i + k))) (seq i) using Suc.hyps by simp
      ultimately have r (seq (Suc (Suc i + k))) (seq i) by (rule transpD)
      thus ?case by simp
    qed
  qed
  hence r (seq (i + Suc(j − i − 1))) (seq i) .
  thus r (seq j) (seq i) using ‹i < j› by simp
qed

lemma almost-full-on-finite-subsetE:
  assumes reflp P and almost-full-on P S
  obtains T where finite T and T ⊆ S and ⋀s. s ∈ S ⟹ (∃t∈T. P t s)
proof −
  define crit where crit = (λU s. s ∈ S ∧ (∀u∈U. ¬ P u s))
  have critD: s ∉ U if crit U s for U s
  proof
    assume s ∈ U
    from ‹crit U s› have ∀u∈U. ¬ P u s unfolding crit-def ..
    from this ‹s ∈ U› have ¬ P s s ..
    moreover from assms(1) have P s s by (rule reflpD)
    ultimately show False ..
  qed
  define fun
    where fun = (λU. (if (∃s. crit U s) then
                    insert (SOME s. crit U s) U
                  else
                    U
                  ))
  define seq where seq = rec-nat {} (λ-. fun)
  have seq-Suc: seq (Suc i) = fun (seq i) for i by (simp add: seq-def)

  have seq-incr-Suc: seq i ⊆ seq (Suc i) for i by (auto simp add: seq-Suc fun-def)
  have seq-incr: i ≤ j ⟹ seq i ⊆ seq j for i j
  proof −
    assume i ≤ j
    hence i = j ∨ i < j by auto
    thus seq i ⊆ seq j
    proof
      assume i = j
```

thus *?thesis* **by** *simp*
next
assume $i < j$
with - *seq-incr-Suc* **show** *?thesis* **by** (*rule transp-sequence*, *simp add*: *transp-def*)
qed
qed
have *sub*: $seq\ i \subseteq S$ **for** i
proof (*induct i*, *simp add*: *seq-def*, *simp add*: *seq-Suc fun-def*, *rule*)
fix i
assume $Ex\ (crit\ (seq\ i))$
hence $crit\ (seq\ i)\ (Eps\ (crit\ (seq\ i)))$ **by** (*rule someI-ex*)
thus $Eps\ (crit\ (seq\ i)) \in S$ **by** (*simp add*: *crit-def*)
qed
have $\exists\, i.\ seq\ (Suc\ i) = seq\ i$
proof (*rule ccontr*, *simp*)
assume $\forall\, i.\ seq\ (Suc\ i) \neq seq\ i$
with *seq-incr-Suc* **have** $seq\ i \subset seq\ (Suc\ i)$ **for** i **by** *blast*
define *seq1* **where** $seq1 = (\lambda n.\ (SOME\ s.\ s \in seq\ (Suc\ n) \wedge s \notin seq\ n))$
have *seq1*: $seq1\ n \in seq\ (Suc\ n) \wedge seq1\ n \notin seq\ n$ **for** n **unfolding** *seq1-def*
proof (*rule someI-ex*)
from ‹$seq\ n \subset seq\ (Suc\ n)$› **show** $\exists\, x.\ x \in seq\ (Suc\ n) \wedge x \notin seq\ n$ **by** *blast*
qed
have $seq1\ i \in S$ **for** i
proof
from *seq1*[*of i*] **show** $seq1\ i \in seq\ (Suc\ i)$ **..**
qed (*fact sub*)
with *assms*(*2*) **obtain** $a\ b$ **where** $a < b$ **and** $P\ (seq1\ a)\ (seq1\ b)$ **by** (*rule almost-full-onD*)
from ‹$a < b$› **have** $Suc\ a \leq b$ **by** *simp*
from *seq1* **have** $seq1\ a \in seq\ (Suc\ a)$ **..**
also from ‹$Suc\ a \leq b$› **have** $\ldots \subseteq seq\ b$ **by** (*rule seq-incr*)
finally have $seq1\ a \in seq\ b$ **.**
from *seq1* **have** $seq1\ b \in seq\ (Suc\ b)$ **and** $seq1\ b \notin seq\ b$ **by** *blast+*
hence $crit\ (seq\ b)\ (seq1\ b)$ **by** (*simp add*: *seq-Suc fun-def someI split*: *if-splits*)
hence $\forall\, u{\in}seq\ b.\ \neg\ P\ u\ (seq1\ b)$ **by** (*simp add*: *crit-def*)
from *this* ‹$seq1\ a \in seq\ b$› **have** $\neg\ P\ (seq1\ a)\ (seq1\ b)$ **..**
from *this* ‹$P\ (seq1\ a)\ (seq1\ b)$› **show** *False* **..**
qed
then obtain i **where** $seq\ (Suc\ i) = seq\ i$ **..**
show *?thesis*
proof
show $finite\ (seq\ i)$ **by** (*induct i*, *simp-all add*: *seq-def fun-def*)
next
fix s
assume $s \in S$
let $?s = Eps\ (crit\ (seq\ i))$
show $\exists\, t{\in}seq\ i.\ P\ t\ s$
proof (*rule ccontr*, *simp*)
assume $\forall\, t{\in}seq\ i.\ \neg\ P\ t\ s$

with ‹*s* ∈ *S*› **have** *crit* (*seq i*) *s* **by** (*simp only*: *crit-def*)
hence *crit* (*seq i*) *?s* **and** *eq*: *seq* (*Suc i*) = *insert ?s* (*seq i*)
by (*auto simp add*: *seq-Suc fun-def intro*: *someI*)
from *this*(*1*) **have** *?s* ∉ *seq i* **by** (*rule critD*)
hence *seq* (*Suc i*) ≠ *seq i* **unfolding** *eq* **by** *blast*
from *this* ‹*seq* (*Suc i*) = *seq i*› **show** *False* **..**
qed
qed (*fact sub*)
qed

1.1 Lists

lemma *map-upt*: *map* (λ*i*. *f* (*xs* ! *i*)) [*0*..<*length xs*] = *map f xs*
by (*auto intro*: *nth-equalityI*)

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 −
have *len-l*: *length ?l* = *length ys* **by** *simp*
from *assms* **have** *len-r*: *length ?r* = *length ys* **by** *simp*
show *?thesis*
proof (*simp only*: *list-eq-iff-nth-eq len-l len-r*, *rule*, *rule*, *intro allI impI*)
fix *i*
assume *i* < *length ys*
hence *i* < *length ?l* **and** *i* < *length ?r* **by** (*simp-all only*: *len-l len-r*)
thus *map* (λ*i*. *f* (*xs* ! *i*) (*ys* ! *i*)) [*0*..<*length ys*] ! *i* = *map* (λ(*x*, *y*). *f x y*) (*zip xs ys*) ! *i*
by *simp*
qed
qed

lemma *distinct-sorted-wrt-irrefl*:
assumes *irreflp rel* **and** *transp rel* **and** *sorted-wrt rel xs*
shows *distinct xs*
using *assms*(*3*)
proof (*induct xs*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons x xs*)
from *Cons*(*2*) **have** *sorted-wrt rel xs* **and** ∗: ∀ *y*∈*set xs*. *rel x y*
by (*simp-all*)
from *this*(*1*) **have** *distinct xs* **by** (*rule Cons*(*1*))
show *?case*
proof (*simp add*: ‹*distinct xs*›, *rule*)
assume *x* ∈ *set xs*
with ∗ **have** *rel x x* **..**

```
    with assms(1) show False by (simp add: irreflp-def)
  qed
qed

lemma distinct-sorted-wrt-imp-sorted-wrt-strict:
  assumes distinct xs and sorted-wrt rel xs
  shows sorted-wrt (λx y. rel x y ∧ ¬ x = y) xs
  using assms
proof (induct xs)
  case Nil
  show ?case by simp
next
  case step: (Cons x xs)
  show ?case
  proof (cases xs)
    case Nil
    thus ?thesis by simp
  next
    case (Cons y zs)
    from step(2) have x ≠ y and 1: distinct (y # zs) by (simp-all add: Cons)
     from step(3) have rel x y and 2: sorted-wrt rel (y # zs) by (simp-all add:
Cons)
   from 1 2 have sorted-wrt (λx y. rel x y ∧ x ≠ y) (y # zs) by (rule step(1)[simplified
Cons])
    with ‹x ≠ y› ‹rel x y› show ?thesis using step.prems by (auto simp: Cons)
  qed
qed

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 −
  from assms have 1: length xs = length ys by (auto dest!: distinct-card)
  from assms(2−6) show ?thesis
  proof(induct rule:list-induct2[OF 1])
    case 1
    show ?case by simp
  next
    case (2 x xs y ys)
    from 2(4) have x ∉ set xs and distinct xs by simp-all
    from 2(6) have y ∉ set ys and distinct ys by simp-all
    have x = y
    proof (rule ccontr)
      assume x ≠ y
      from 2(3) have ∀ z∈set xs. rel x z by (simp)
      moreover from ‹x ≠ y› have y ∈ set xs using 2(7) by auto
      ultimately have ∗: rel x y ..
      from 2(5) have ∀ z∈set ys. rel y z by (simp)
```

moreover from ‹$x \neq y$› **have** $x \in set\ ys$ **using** *2(7)* **by** *auto*
ultimately have *rel y x* **..**
with *assms(1)* $*$ **have** $x = y$ **by** (*rule antisympD*)
with ‹$x \neq y$› **show** *False* **..**
qed
from *2(3)* **have** *sorted-wrt rel xs* **by** (*simp*)
moreover note ‹*distinct xs*›
moreover from *2(5)* **have** *sorted-wrt rel ys* **by** (*simp*)
moreover note ‹*distinct ys*›
moreover from *2(7)* ‹$x \notin set\ xs$› ‹$y \notin set\ ys$› **have** $set\ xs = set\ ys$ **by** (*auto simp add*: ‹$x = y$›)
ultimately have $xs = ys$ **by** (*rule 2(2)*)
with ‹$x = y$› **show** *?case* **by** *simp*
qed
qed

lemma *sorted-wrt-refl-nth-mono*:
assumes *reflp P* **and** *sorted-wrt P xs* **and** $i \leq j$ **and** $j < length\ xs$
shows *P (xs ! i) (xs ! j)*
proof (*cases* $i < j$)
case *True*
from *assms(2) this assms(4)* **show** *?thesis* **by** (*rule sorted-wrt-nth-less*)
next
case *False*
with *assms(3)* **have** $i = j$ **by** *simp*
from *assms(1)* **show** *?thesis* **unfolding** ‹$i = j$› **by** (*rule reflpD*)
qed

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)* = $set\ xs \cup set\ ys$
proof (*induct rel xs ys rule*: *merge-wrt.induct*)
case (*1 rel xs*)
show *?case* **by** *simp*
next
case (*2 rel y ys*)
show *?case* **by** *simp*
next
case (*3 rel x xs y ys*)

```
  show ?case
  proof (cases x = y)
    case True
    thus ?thesis by (simp add: 3(1))
  next
    case False
    show ?thesis
    proof (cases rel x y)
      case True
      with ‹x ≠ y› show ?thesis by (simp add: 3(2) insert-commute)
    next
      case False
      with ‹x ≠ y› show ?thesis by (simp add: 3(3))
    qed
  qed
qed

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)
  using assms
proof (induct rel xs ys rule: merge-wrt.induct)
  case (1 rel xs)
  from 1(3) show ?case by simp
next
  case (2 rel y ys)
  from 2(4) show ?case by simp
next
  case (3 rel x xs y ys)
  show ?case
  proof (cases x = y)
    case True
    show ?thesis
    proof (auto simp add: True)
      fix z
      assume z ∈ set (merge-wrt rel xs ys)
      hence z ∈ set xs ∪ set ys by (simp only: set-merge-wrt)
      thus rel y z
      proof
        assume z ∈ set xs
        with 3(6) show ?thesis by (simp add: True)
      next
        assume z ∈ set ys
        with 3(7) show ?thesis by (simp)
      qed
    next
      note True 3(4, 5)
      moreover from 3(6) have sorted-wrt rel xs by (simp)
```

moreover from *3(7)* **have** *sorted-wrt rel ys* **by** (*simp*)
ultimately show *sorted-wrt rel* (*merge-wrt rel xs ys*) **by** (*rule 3(1)*)
qed
next
case *False*
show *?thesis*
proof (*cases rel x y*)
case *True*
show *?thesis*
proof (*auto simp add*: *False True*)
fix *z*
assume $z \in$ *set* (*merge-wrt rel xs* (*y* # *ys*))
hence $z \in$ *insert y* (*set xs* ∪ *set ys*) **by** (*simp add*: *set-merge-wrt*)
thus *rel x z*
proof
assume $z = y$
with *True* **show** *?thesis* **by** *simp*
next
assume $z \in$ *set xs* ∪ *set ys*
thus *?thesis*
proof
assume $z \in$ *set xs*
with *3(6)* **show** *?thesis* **by** (*simp*)
next
assume $z \in$ *set ys*
with *3(7)* **have** *rel y z* **by** (*simp*)
with *3(4) True* **show** *?thesis* **by** (*rule transpD*)
qed
qed
next
note *False True 3(4, 5)*
moreover from *3(6)* **have** *sorted-wrt rel xs* **by** (*simp*)
ultimately show *sorted-wrt rel* (*merge-wrt rel xs* (*y* # *ys*)) **using** *3(7)* **by**
(*rule 3(2)*)
qed
next
assume ¬ *rel x y*
from ‹$x \neq y$› **have** *rel x y* ∨ *rel y x* **by** (*rule 3(5)*)
with ‹¬ *rel x y*› **have** ∗: *rel y x* **by** *simp*
show *?thesis*
proof (*auto simp add*: *False* ‹¬ *rel x y*›)
fix *z*
assume $z \in$ *set* (*merge-wrt rel* (*x* # *xs*) *ys*)
hence $z \in$ *insert x* (*set xs* ∪ *set ys*) **by** (*simp add*: *set-merge-wrt*)
thus *rel y z*
proof
assume $z = x$
with ∗ **show** *?thesis* **by** *simp*
next

assume $z \in$ *set xs* $\cup$ *set ys*
thus *?thesis*
proof
assume $z \in$ *set xs*
with *3*(*6*) **have** *rel x z* **by** (*simp*)
with *3*(*4*) ∗ **show** *?thesis* **by** (*rule transpD*)
next
assume $z \in$ *set ys*
with *3*(*7*) **show** *?thesis* **by** (*simp*)
qed
qed
next
note *False* ‹¬ *rel x y*› *3*(*4*, *5*, *6*)
moreover from *3*(*7*) **have** *sorted-wrt rel ys* **by** (*simp*)
ultimately show *sorted-wrt rel* (*merge-wrt rel* (*x* # *xs*) *ys*) **by** (*rule 3*(*3*))
qed
qed
qed
qed

lemma *set-fold*:
assumes $\bigwedge$*x ys*. *set* (*f* (*g x*) *ys*) = *set* (*g x*) $\cup$ *set ys*
shows *set* (*fold* (λ*x*. *f* (*g x*)) *xs ys*) = ($\bigcup$*x*$\in$*set xs*. *set* (*g x*)) $\cup$ *set ys*
proof (*induct xs arbitrary*: *ys*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons x xs*)
have *eq*: *set* (*fold* (λ*x*. *f* (*g x*)) *xs* (*f* (*g x*) *ys*)) = ($\bigcup$*x*$\in$*set xs*. *set* (*g x*)) $\cup$ *set* (*f* (*g x*) *ys*)
by (*rule Cons*)
show *?case* **by** (*simp add*: *o-def assms set-merge-wrt eq ac-simps*)
qed

1.2 Sums and Products

lemma *additive-implies-homogenous*:
assumes $\bigwedge$*x y*. *f* (*x* + *y*) = *f x* + ((*f* (*y*::'*a*::*monoid-add*))::'*b*::*cancel-comm-monoid-add*)
shows *f 0* = *0*
proof −
have *f* (*0* + *0*) = *f 0* + *f 0* **by** (*rule assms*)
hence *f 0* = *f 0* + *f 0* **by** *simp*
thus *f 0* = *0* **by** *simp*
qed

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 (*cases finite A*)

case *True*
thus *?thesis*
proof (*induct A*)
case *empty*
thus *?case* **by** (*simp add*: *assms*(*1*))
next
case *step*: (*insert a A*)
show *?case* **by** (*simp add*: *sum.insert*[*OF step*(*1*) *step*(*2*)] *assms*(*2*) *step*(*3*))
qed
next
case *False*
thus *?thesis* **by** (*simp add*: *assms*(*1*))
qed

lemma *fun-sum-commute-canc*:
assumes ⋀*x y*. *f* (*x* + *y*) = *f x* + ((*f y*)::'*a*::*cancel-comm-monoid-add*)
shows *f* (*sum g A*) = (∑ *a*∈*A*. *f* (*g a*))
by (*rule fun-sum-commute*, *rule additive-implies-homogenous*, *fact*+)

lemma *fun-sum-list-commute*:
assumes *f 0* = *0* **and** ⋀*x y*. *f* (*x* + *y*) = *f x* + *f y*
shows *f* (*sum-list xs*) = *sum-list* (*map f xs*)
proof (*induct xs*)
case *Nil*
thus *?case* **by** (*simp add*: *assms*(*1*))
next
case (*Cons x xs*)
thus *?case* **by** (*simp add*: *assms*(*2*))
qed

lemma *fun-sum-list-commute-canc*:
assumes ⋀*x y*. *f* (*x* + *y*) = *f x* + ((*f y*)::'*a*::*cancel-comm-monoid-add*)
shows *f* (*sum-list xs*) = *sum-list* (*map f xs*)
by (*rule fun-sum-list-commute*, *rule additive-implies-homogenous*, *fact*+)

lemma *sum-set-upt-eq-sum-list*: (∑ *i* = *m*..<*n*. *f i*) = (∑ *i*←[*m*..<*n*]. *f i*)
using *sum-set-upt-conv-sum-list-nat* **by** *auto*

lemma *sum-list-upt*: (∑ *i*←[*0*..<(*length xs*)]. *f* (*xs* ! *i*)) = (∑ *x*←*xs*. *f x*)
by (*simp only*: *map-upt*)

lemma *sum-list-upt-zip*:
assumes *length xs* = *length ys*
shows (∑ *i*←[*0*..<(*length ys*)]. *f* (*xs* ! *i*) (*ys* ! *i*)) = (∑ (*x*, *y*)←(*zip xs ys*). *f x y*)
by (*simp only*: *map-upt-zip*[*OF assms*])

lemma *sum-list-zeroI*:
assumes *set xs* ⊆ {*0*}

```
  shows sum-list xs = 0
  using assms by (induct xs, auto)

lemma fun-prod-commute:
  assumes f 1 = 1 and ⋀x y. f (x * y) = f x * f y
  shows f (prod g A) = (∏ a∈A. f (g a))
proof (cases finite A)
  case True
  thus ?thesis
  proof (induct A)
    case empty
    thus ?case by (simp add: assms(1))
  next
    case step: (insert a A)
    show ?case by (simp add: prod.insert[OF step(1) step(2)] assms(2) step(3))
  qed
next
  case False
  thus ?thesis by (simp add: assms(1))
qed

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 ⇒₀ nat) ⇒₀ 'a::zero) set
  morphisms mapping-of MPoly
 ..

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* :: $(nat \Rightarrow_0 nat) \Rightarrow_0 'a$ **.**

instance ..

end

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

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

instance
by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff add.assoc*)+

end

instance *mpoly* :: (*comm-monoid-add*) *comm-monoid-add*
by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff ac-simps*)+

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

lift-definition *minus-mpoly* :: *'a mpoly* ⇒ *'a mpoly* ⇒ *'a mpoly*
is *Groups.minus* :: $((nat \Rightarrow_0 nat) \Rightarrow_0 'a) \Rightarrow$ - **.**

instance
by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff diff-diff-add*)+

end

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

lift-definition *uminus-mpoly* :: *'a mpoly* ⇒ *'a mpoly*
is *Groups.uminus* :: $((nat \Rightarrow_0 nat) \Rightarrow_0 'a) \Rightarrow$ - **.**

instance
by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff add-uminus-conv-diff*)+

end

2.3 Multiplication by a coefficient

lift-definition *smult* :: *'a*::{*times*,*zero*} ⇒ *'a mpoly* ⇒ *'a mpoly*
is *λa. Poly-Mapping.map* (*Groups.times a*) :: $((nat \Rightarrow_0 nat) \Rightarrow_0 'a) \Rightarrow$ - **.**

2.4 Multiplicative structure

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

lift-definition *one-mpoly* :: *'a mpoly*
  is *1* :: ((*nat* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'a*) **.**

instance
  by *intro-classes* (*transfer*, *simp*)

end

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

lift-definition *times-mpoly* :: *'a mpoly* ⇒ *'a mpoly* ⇒ *'a mpoly*
  is *Groups.times* :: ((*nat* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'a*) ⇒ - **.**

instance
  by *intro-classes* (*transfer*, *simp add*: *algebra-simps*)+

end

instance *mpoly* :: (*comm-semiring-0*) *comm-semiring-0*
  by *intro-classes* (*transfer*, *simp add*: *algebra-simps*)+

instance *mpoly* :: (*semiring-0-cancel*) *semiring-0-cancel*
  ..

instance *mpoly* :: (*comm-semiring-0-cancel*) *comm-semiring-0-cancel*
  ..

instance *mpoly* :: (*semiring-1*) *semiring-1*
  by *intro-classes* (*transfer*, *simp*)+

instance *mpoly* :: (*comm-semiring-1*) *comm-semiring-1*
  by *intro-classes* (*transfer*, *simp*)+

instance *mpoly* :: (*semiring-1-cancel*) *semiring-1-cancel*
  ..

instance *mpoly* :: (*ring*) *ring*
  ..

instance *mpoly* :: (*comm-ring*) *comm-ring*
  ..

instance *mpoly* :: (*ring-1*) *ring-1*
..

instance *mpoly* :: (*comm-ring-1*) *comm-ring-1*
..

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* :: $(nat \Rightarrow_0 nat) \Rightarrow$ *'a::zero* $\Rightarrow$ *'a mpoly*
is *Poly-Mapping.single* :: $(nat \Rightarrow_0 nat) \Rightarrow$ - .

lemma *mapping-of-monom* [*simp*]:
mapping-of (*monom m a*) = *Poly-Mapping.single m a*
by(*fact monom.rep-eq*)

lemma *monom-zero* [*simp*]:
monom 0 0 = *0*
by *transfer simp*

lemma *monom-one* [*simp*]:
monom 0 1 = *1*
by *transfer simp*

lemma *monom-add*:
monom m (*a* + *b*) = *monom m a* + *monom m b*
by *transfer* (*simp add*: *single-add*)

lemma *monom-uminus*:
monom m (− *a*) = − *monom m a*
by *transfer* (*simp add*: *single-uminus*)

lemma *monom-diff*:
monom m (*a* − *b*) = *monom m a* − *monom m b*
by *transfer* (*simp add*: *single-diff*)

lemma *monom-numeral* [*simp*]:
monom 0 (*numeral n*) = *numeral n*
by (*induct n*) (*simp-all only*: *numeral.simps numeral-add monom-zero monom-one monom-add*)

lemma *monom-of-nat* [*simp*]:
monom 0 (*of-nat n*) = *of-nat n*
by (*induct n*) (*simp-all add*: *monom-add*)

lemma *of-nat-monom*:

of-nat = *monom 0* ∘ *of-nat*
by (*simp add*: *fun-eq-iff*)

lemma *inj-monom* [*iff*]:
inj (*monom m*)
proof (*rule injI*, *transfer*)
fix *a b* :: *'a* **and** *m* :: *nat* $\Rightarrow_0$ *nat*
assume *Poly-Mapping.single m a* = *Poly-Mapping.single m b*
with *injD* [*of Poly-Mapping.single m a b*]
show *a* = *b* **by** *simp*
qed

lemma *mult-monom*: *monom x a* ∗ *monom y b* = *monom* (*x* + *y*) (*a* ∗ *b*)
by *transfer'* (*simp add*: *Poly-Mapping.mult-single*)

instance *mpoly* :: (*semiring-char-0*) *semiring-char-0*
by *intro-classes* (*auto simp add*: *of-nat-monom inj-of-nat intro*: *inj-compose*)

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

lemma *monom-of-int* [*simp*]:
monom 0 (*of-int k*) = *of-int k*
apply (*cases k*)
apply *simp-all*
unfolding *monom-diff monom-uminus*
apply *simp*
done

2.6 Constants and Indeterminates

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

definition Var_0 :: *'a* ⇒ (*'a* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'b*::{*one*,*zero*} **where**
Var_0 *n* ≡ *Poly-Mapping.single* (*Poly-Mapping.single n 1*) *1*
definition $Const_0$ :: *'b* ⇒ (*'a* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'b*::*zero* **where** $Const_0$ *c* ≡ *Poly-Mapping.single 0 c*

lemma $Const_0$*-one*: $Const_0$ *1* = *1*
by (*simp add*: $Const_0$*-def*)

lemma $Const_0$*-numeral*: $Const_0$ (*numeral x*) = *numeral x*
by (*auto intro*!: *poly-mapping-eqI simp*: $Const_0$*-def lookup-numeral*)

lemma $Const_0$*-minus*: $Const_0$ (− *x*) = − $Const_0$ *x*
by (*simp add*: $Const_0$*-def single-uminus*)

lemma $Const_0$*-zero*: $Const_0$ *0* = *0*
by (*auto intro*!: *poly-mapping-eqI simp*: $Const_0$*-def*)

lemma *Var*$_0$*-power*: *Var*$_0$ *v* ^ *n* = *Poly-Mapping.single* (*Poly-Mapping.single v n*) *1*
by (*induction n*) (*auto simp*: *Var*$_0$*-def mult-single single-add*[*symmetric*])

lift-definition *Var*::*nat* ⇒ *'b*::{*one*,*zero*} *mpoly* **is** *Var*$_0$ **.**
lift-definition *Const*::*'b*::*zero* ⇒ *'b mpoly* **is** *Const*$_0$ **.**

2.7 Integral domains

instance *mpoly* :: (*ring-no-zero-divisors*) *ring-no-zero-divisors*
by *intro-classes* (*transfer*, *simp*)

instance *mpoly* :: (*ring-1-no-zero-divisors*) *ring-1-no-zero-divisors*
..

instance *mpoly* :: (*idom*) *idom*
..

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* **.**

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

lemma *aux*:
Poly-Mapping.lookup f = (λ-. *0*) ⟷ *f* = *0*
apply *transfer* **apply** *simp* **done**

lemma *insertion-trivial* [*simp*]:
insertion (λ-. *0*) *p* = *coeff p 0*
proof −
{ **fix** *f* :: (*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*

have *insertion-aux* (λ-. *0*) *f* = *Poly-Mapping.lookup f 0*
apply (*simp add*: *insertion-aux-def insertion-fun-def power-Sum-any* [*symmetric*])
apply (*simp add*: *zero-power-eq mult-when aux*)
done
}
then show *?thesis* **by** (*simp add*: *coeff-def insertion-def*)
qed

lemma *insertion-zero* [*simp*]:
insertion f 0 = *0*
by *transfer* (*simp add*: *insertion-aux-def insertion-fun-def*)

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*)
unfolding *insertion-fun-def*
apply (*subst Sum-any.distrib* [*symmetric*])
apply (*simp-all add*: *plus-poly-mapping.rep-eq algebra-simps*)
apply (*rule finite-mult-not-eq-zero-rightI*)
apply *simp*
apply (*rule finite-mult-not-eq-zero-rightI*)
apply *simp*
done

lemma *insertion-add*:
insertion f (*p* + *q*) = *insertion f p* + *insertion f q*
by *transfer* (*simp add*: *insertion-aux-def insertion-fun-add*)

lemma *insertion-one* [*simp*]:
insertion f 1 = *1*
by *transfer* (*simp add*: *insertion-aux-def insertion-fun-def one-poly-mapping.rep-eq when-mult*)

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 −
{ fix *m* :: *nat* $\Rightarrow_0$ *nat*
have *finite* {*v*. *Poly-Mapping.lookup m v* ≠ *0*}
by *simp*
then have *finite* {*v*. *f v* ^ *Poly-Mapping.lookup m v* ≠ *1*}
by (*rule rev-finite-subset*) (*auto intro*: *ccontr*)
}
moreover define *g* **where** *g m* = (∏ *v*. *f v* ^ *Poly-Mapping.lookup m v*) **for** *m*
ultimately have ∗: ⋀*a b*. *g* (*a* + *b*) = *g a* ∗ *g b*

by (*simp add*: *plus-poly-mapping.rep-eq power-add Prod-any.distrib*)
have *bij*: *bij* (λ(*l*, *n*, *m*). (*m*, *l*, *n*))
by (*auto intro*!: *bijI injI simp add*: *image-def*)
let *?P* = {*l*. *Poly-Mapping.lookup p l* ≠ *0*}
let *?Q* = {*n*. *Poly-Mapping.lookup q n* ≠ *0*}
let *?PQ* = {*l* + *n* | *l n*. *l* ∈ *Poly-Mapping.keys p* ∧ *n* ∈ *Poly-Mapping.keys q*}
have *finite* {*l* + *n* | *l n*. *Poly-Mapping.lookup p l* ≠ *0* ∧ *Poly-Mapping.lookup q n* ≠ *0*}
by (*rule finite-not-eq-zero-sumI*) *simp-all*
then have *fin-PQ*: *finite ?PQ*
by (*simp add*: *in-keys-iff*)
have (∑ *m*. *Poly-Mapping.lookup* (*p* ∗ *q*) *m* ∗ *g m*) =
(∑ *m*. (∑ *l*. *Poly-Mapping.lookup p l* ∗ (∑ *n*. *Poly-Mapping.lookup q n when m* = *l* + *n*)) ∗ *g m*)
by (*simp add*: *times-poly-mapping.rep-eq prod-fun-def*)
also have . . . = (∑ *m*. (∑ *l*. (∑ *n*. *g m* ∗ (*Poly-Mapping.lookup p l* ∗ *Poly-Mapping.lookup q n*) *when m* = *l* + *n*)))
apply (*subst Sum-any-left-distrib*)
apply (*auto intro*: *finite-mult-not-eq-zero-rightI*)
apply (*subst Sum-any-right-distrib*)
apply (*auto intro*: *finite-mult-not-eq-zero-rightI*)
apply (*subst Sum-any-left-distrib*)
apply (*auto intro*: *finite-mult-not-eq-zero-leftI*)
apply (*simp add*: *ac-simps mult-when*)
done
also have . . . = (∑ *m*. (∑ (*l*, *n*). *g m* ∗ (*Poly-Mapping.lookup p l* ∗ *Poly-Mapping.lookup q n*) *when m* = *l* + *n*))
apply (*subst* (*2*) *Sum-any.cartesian-product* [*of ?P* × *?Q*])
apply (*auto dest*!: *mult-not-zero*)
done
also have . . . = (∑ (*m*, *l*, *n*). *g m* ∗ (*Poly-Mapping.lookup p l* ∗ *Poly-Mapping.lookup q n*) *when m* = *l* + *n*)
apply (*subst Sum-any.cartesian-product* [*of ?PQ* × (*?P* × *?Q*)])
apply (*auto dest*!: *mult-not-zero simp add*: *fin-PQ*)
apply (*auto simp*: *in-keys-iff*)
done
also have . . . = (∑ (*l*, *n*, *m*). *g m* ∗ (*Poly-Mapping.lookup p l* ∗ *Poly-Mapping.lookup q n*) *when m* = *l* + *n*)
using *bij* **by** (*rule Sum-any.reindex-cong* [*of* λ(*l*, *n*, *m*). (*m*, *l*, *n*)]) (*simp add*: *fun-eq-iff*)
also have . . . = (∑ (*l*, *n*). ∑ *m*. *g m* ∗ (*Poly-Mapping.lookup p l* ∗ *Poly-Mapping.lookup q n*) *when m* = *l* + *n*)
apply (*subst Sum-any.cartesian-product2* [*of* (*?P* × *?Q*) × *?PQ*])
apply (*auto dest*!: *mult-not-zero simp add*: *fin-PQ*)
apply (*auto simp*: *in-keys-iff*)
done
also have . . . = (∑ (*l*, *n*). (*g l* ∗ *g n*) ∗ (*Poly-Mapping.lookup p l* ∗ *Poly-Mapping.lookup q n*))
by (*simp add*: ∗)

also have ... = ($\sum$ *l*. $\sum$ *n*. (*g l* ∗ *g n*) ∗ (*Poly-Mapping.lookup p l* ∗ *Poly-Mapping.lookup q n*))
  apply (*subst Sum-any.cartesian-product* [*of ?P* × *?Q*])
  apply (*auto dest*!: *mult-not-zero*)
  done
also have ... = ($\sum$ *l*. $\sum$ *n*. (*Poly-Mapping.lookup p l* ∗ *g l*) ∗ (*Poly-Mapping.lookup q n* ∗ *g n*))
  by (*simp add*: *ac-simps*)
also have ... =
  ($\sum$ *m*. *Poly-Mapping.lookup p m* ∗ *g m*) ∗
  ($\sum$ *m*. *Poly-Mapping.lookup q m* ∗ *g m*)
 by (*rule Sum-any-product* [*symmetric*]) (*auto intro*: *finite-mult-not-eq-zero-rightI*)
finally show *?thesis* **by** (*simp add*: *insertion-fun-def g-def*)
qed

lemma *insertion-mult*:
  insertion f (*p* ∗ *q*) = *insertion f p* ∗ *insertion f q*
  by *transfer* (*simp add*: *insertion-aux-def insertion-fun-mult*)

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*)) **.**

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*)) **.**

lemma *degree-zero* [*simp*]:
  degree 0 v = *0*
  by *transfer simp*

lemma *total-degree-zero* [*simp*]:
  total-degree 0 = *0*
  by *transfer simp*

lemma *degree-one* [*simp*]:
  degree 1 v = *0*
  by *transfer simp*

lemma *total-degree-one* [*simp*]:
  total-degree 1 = *0*
  by *transfer simp*

2.11 Pseudo-division of polynomials

lemma *smult-conv-mult*: *smult s p* = *monom 0 s* ∗ *p*
by *transfer* (*simp add*: *mult-map-scale-conv-mult*)

lemma *smult-monom* [*simp*]:
  fixes *c* :: - :: *mult-zero*
  shows *smult c* (*monom x c′*) = *monom x* (*c* ∗ *c′*)
by *transfer simp*

lemma *smult-0* [*simp*]:
  fixes *p* :: - :: *mult-zero mpoly*
  shows *smult 0 p* = *0*
by *transfer*(*simp add*: *map-eq-zero-iff*)

lemma *mult-smult-left*: *smult s p* ∗ *q* = *smult s* (*p* ∗ *q*)
by(*simp add*: *smult-conv-mult mult.assoc*)

lift-definition *sdiv* :: *′a*::*euclidean-ring* ⇒ *′a mpoly* ⇒ *′a mpoly*
  is *λa. Poly-Mapping.map* (*λb. b div a*) :: ((*nat* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *′a*) ⇒ -
.

‘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*)
by (*simp add*: *pdivmod-def*)

lemma *pmod-code*:
p pmod q = *snd* (*pdivmod p q*)
by (*simp add*: *pdivmod-def*)

2.12 Primitive poly, etc

lift-definition *coeffs* :: *'a* :: *zero mpoly* ⇒ *'a set*
is *Poly-Mapping.range* :: ((*nat* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'a*) ⇒ - **.**

lemma *finite-coeffs* [*simp*]: *finite* (*coeffs p*)
by *transfer simp*

[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* ($\lambda x.$ *f x when x*∈*S*)) = ($\lambda x.$ *f x when x*∈*S*)
proof −
  have *finite* {*x*. (*f x when x*∈*S*) ≠ *0*} **using** *assms* **by** *auto*
  then show *?thesis* **using** *lookup-Abs-poly-mapping* **by** *fast*
qed

definition *remove-key*::$'a \Rightarrow ('a \Rightarrow_0 'b$::*monoid-add*$) \Rightarrow ('a \Rightarrow_0 'b)$ **where**
  remove-key k0 f = *Abs-poly-mapping* ($\lambda k.$ *lookup f k when k* ≠ *k0*)

lemma *remove-key-lookup*:
  lookup (*remove-key k0 f*) *k* = (*lookup f k when k* ≠ *k0*)
unfolding *remove-key-def* **using** *finite-subset* **by** (*simp add*: *lookup-Abs-poly-mapping*)

lemma *remove-key-keys*: *keys f* − {*k*} = *keys* (*remove-key k f*) (**is** *?A* = *?B*)
proof (*rule antisym*; *rule subsetI*)
  fix *x* **assume** *x* ∈ *?A*
  then show *x* ∈ *?B* **using** *remove-key-lookup lookup-not-eq-zero-eq-in-keys DiffD1 DiffD2 insertCI*
    by (*metis* (*mono-tags*, *lifting*) *when-def*)
next
  fix *x* **assume** *x* ∈ *?B*
  then have *lookup* (*remove-key k f*) *x* ≠ *0* **by** *blast*
  then show *x* ∈ *?A*
    by (*simp add*: *lookup-not-eq-zero-eq-in-keys remove-key-lookup*)
qed

lemma *remove-key-sum*: *remove-key k f* + *Poly-Mapping.single k* (*lookup f k*) = *f*
proof −
  {
  fix *k'*
  have *rem*:(*lookup f k' when k'* ≠ *k*) = *lookup* (*remove-key k f*) *k'*
    using *when-def* **by** (*simp add*: *remove-key-lookup*)
  have *sin*:(*lookup f k when k'*=*k*) = *lookup* (*Poly-Mapping.single k* (*lookup f k*)) *k'*
    by (*simp add*: *lookup-single-not-eq when-def*)
  have *lookup f k'* = (*lookup f k' when k'* ≠ *k*) + ((*lookup f k*) *when k'*=*k*)
    unfolding *when-def* **by** *fastforce*
  with *rem sin* **have** *lookup f k'* = *lookup* ((*remove-key k f*) + *Poly-Mapping.single k* (*lookup f k*)) *k'*
    using *lookup-add* **by** *metis*
  }
  then show *?thesis* **by** (*metis poly-mapping-eqI*)
qed

lemma *remove-key-single*[*simp*]: *remove-key v* (*Poly-Mapping.single v n*) = *0*
proof −
have *0*:⋀*k*. (*lookup* (*Poly-Mapping.single v n*) *k when k* ≠ *v*) = *0* **by** (*simp add*: *lookup-single-not-eq when-def*)
show *?thesis* **unfolding** *remove-key-def 0*
by *auto*
qed

lemma *remove-key-add*: *remove-key v m* + *remove-key v m′* = *remove-key v* (*m* + *m′*)
by (*rule poly-mapping-eqI*; *simp add*: *lookup-add remove-key-lookup when-add-distrib*)

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* **using** *finite-keys*[*of f*]
proof (*induction keys f arbitrary*: *f rule*: *finite-induct*)
case (*empty*)
then show *?case* **using** *single*[*of - 0*] **by** (*metis* (*full-types*) *aux empty-iff not-in-keys-iff-lookup-eq-zero single-zero*)
next
case (*insert k K f*)
obtain *f1 f2* **where** *f12-def*: *f1* = *remove-key k f f2* = *Poly-Mapping.single k* (*lookup f k*) **by** *blast*
have *P f1*
proof −
have *Suc* (*card* (*keys f1*)) = *card* (*keys f*)
using *remove-key-keys finite-keys f12-def*(*1*) **by** (*metis* (*no-types*) *Diff-insert-absorb card-insert-disjoint insert.hyps*(*2*) *insert.hyps*(*4*))
then show *?thesis* **using** *insert lessI* **by** (*metis Diff-insert-absorb f12-def*(*1*) *remove-key-keys*)
qed
have *P f2* **by** (*simp add*: *single f12-def*(*2*))
have *f1* + *f2* = *f* **using** *remove-key-sum f12-def* **by** *auto*
have *k* ∉ *keys f1* **using** *remove-key-keys f12-def* **by** *fast*
then show *?case* **using** ‹*P f1*› ‹*P f2*› *sum*[*of f1 f2 k lookup f k*] ‹*f1* + *f2* = *f*› *f12-def* **by** *auto*
qed

lemma *map-lookup*:
assumes *g 0* = *0*
shows *lookup* (*Poly-Mapping.map g f*) *x* = *g* ((*lookup f*) *x*)
proof −
have (*g* (*lookup f x*) *when lookup f x* ≠ *0*) = *g* (*lookup f x*)
by (*metis* (*mono-tags*, *lifting*) *assms when-def*)
then have (*g* (*lookup f x*) *when x* ∈ *keys f*) = *g* (*lookup f x*)

using *lookup-not-eq-zero-eq-in-keys* [*of f*] **by** *simp*
then show *?thesis*
by (*simp add*: *Poly-Mapping.map-def map-fun-def in-keys-iff*)
qed

lemma *keys-add*:
assumes *keys f* ∩ *keys g* = {}
shows *keys f* ∪ *keys g* = *keys* (*f*+*g*)
proof
have *keys f* ⊆ *keys* (*f*+*g*)
proof
fix *x* **assume** *x*∈*keys f*
then have *lookup* (*f*+*g*) *x* = *lookup f x* **by** (*metis add.right-neutral assms disjoint-iff-not-equal not-in-keys-iff-lookup-eq-zero plus-poly-mapping.rep-eq*)
then show *x*∈*keys* (*f*+*g*) **using** ‹*x*∈*keys f*› **by** (*metis not-in-keys-iff-lookup-eq-zero*)
qed
moreover have *keys g* ⊆ *keys* (*f*+*g*)
proof
fix *x* **assume** *x*∈*keys g*
then have *lookup* (*f*+*g*) *x* = *lookup g x* **by** (*metis IntI add.left-neutral assms empty-iff not-in-keys-iff-lookup-eq-zero plus-poly-mapping.rep-eq*)
then show *x*∈*keys* (*f*+*g*) **using** ‹*x*∈*keys g*› **by** (*metis not-in-keys-iff-lookup-eq-zero*)
qed
ultimately show *keys f* ∪ *keys g* ⊆ *keys* (*f*+*g*) **by** *simp*
next
show *keys* (*f* + *g*) ⊆ *keys f* ∪ *keys g* **by** (*simp add*: *keys-add*)
qed

lemma *fun-when*:
f 0 = *0* ⟹ *f* (*a when P*) = (*f a when P*) **by** (*simp add*: *when-def*)

4 MPoly extension

lemma *coeff-all-0*:(⋀*m*. *coeff p m* = *0*) ⟹ *p*=*0*
by (*metis aux coeff-def mapping-of-inject zero-mpoly.rep-eq*)

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

lemma *vars-finite*: *finite* (*vars p*) **unfolding** *vars-def* **by** *auto*

lemma *vars-monom-single*: *vars* (*monom* (*Poly-Mapping.single v k*) *a*) ⊆ {*v*}
proof
fix *w* **assume** *w* ∈ *vars* (*monom* (*Poly-Mapping.single v k*) *a*)
then have *w* = *v* **using** *vars-def* **by** (*metis UN-E lookup-eq-zero-in-keys-contradict lookup-single-not-eq monom.rep-eq*)
then show *w* ∈ {*v*} **by** *auto*
qed

lemma *vars-monom-keys*:
assumes *a≠0*
shows *vars* (*monom m a*) = *keys m*
proof (*rule antisym*; *rule subsetI*)
  fix *w* **assume** *w* ∈ *vars* (*monom m a*)
  then have *lookup m w* ≠ *0* **using** *vars-def* **by** (*metis UN-E lookup-eq-zero-in-keys-contradict lookup-single-not-eq monom.rep-eq*)
  then show *w* ∈ *keys m* **by** (*meson lookup-not-eq-zero-eq-in-keys*)
next
  fix *w* **assume** *w* ∈ *keys m*
  then have *lookup m w* ≠ *0* **by** (*meson lookup-not-eq-zero-eq-in-keys*)
  then show *w* ∈ *vars* (*monom m a*) **unfolding** *vars-def* **using** *assms* **by** (*metis UN-iff lookup-not-eq-zero-eq-in-keys lookup-single-eq monom.rep-eq*)
qed

lemma *vars-monom-subset*:
shows *vars* (*monom m a*) ⊆ *keys m*
  by (*cases a=0*; *simp add*: *vars-def vars-monom-keys*)

lemma *vars-monom-single-cases*: *vars* (*monom* (*Poly-Mapping.single v k*) *a*) = (*if k=0* ∨ *a=0 then* {} *else* {*v*})
proof(*cases k=0*)
  assume *k=0*
  then have (*Poly-Mapping.single v k*) = *0* **by** *simp*
  then have *vars* (*monom* (*Poly-Mapping.single v k*) *a*) = {}
    by (*metis* (*mono-tags*, *lifting*) *single-zero singleton-inject subset-singletonD vars-monom-single zero-neq-one*)
  then show *?thesis* **using** ‹*k=0*› **by** *auto*
next
  assume *k≠0*
  then show *?thesis*
  proof (*cases a=0*)
    assume *a=0*
    then have *monom* (*Poly-Mapping.single v k*) *a* = *0* **by** (*metis monom.abs-eq monom-zero single-zero*)
    then show *?thesis* **by** (*metis* (*mono-tags*, *opaque-lifting*) ‹*k* ≠ *0*› ‹*a=0*› *monom.abs-eq single-zero singleton-inject subset-singletonD vars-monom-single*)
  next
    assume *a≠0*
    then have *v* ∈ *vars* (*monom* (*Poly-Mapping.single v k*) *a*) **by** (*simp add*: ‹*k* ≠ *0*› *vars-def*)
    then show *?thesis* **using** ‹*a≠0*› ‹*k* ≠ *0*› *vars-monom-single* **by** *fastforce*
  qed
qed

lemma *vars-monom*:
assumes *a≠0*
shows *vars* (*monom m* (*1*::*'a*::*zero-neq-one*)) = *vars* (*monom m* (*a*::*'a*))
  unfolding *vars-monom-keys*[*OF assms*] **using** *vars-monom-keys*[*of 1*] *one-neq-zero*

by *blast*

lemma *vars-add*: *vars* (*p1* + *p2*) ⊆ *vars p1* ∪ *vars p2*
proof
 fix *w* **assume** *w* ∈ *vars* (*p1* + *p2*)
 then obtain *m* **where** *w* ∈ *keys m m* ∈ *keys* (*mapping-of* (*p1* + *p2*)) **by** (*metis UN-E vars-def*)
 then have *m* ∈ *keys* (*mapping-of* (*p1*)) ∪ *keys* (*mapping-of* (*p2*))
  by (*metis Poly-Mapping.keys-add plus-mpoly.rep-eq subset-iff*)
 then show *w* ∈ *vars p1* ∪ *vars p2* **using** *vars-def* ‹*w* ∈ *keys m*› **by** *fastforce*
qed

lemma *vars-mult*: *vars* (*p*∗*q*) ⊆ *vars p* ∪ *vars q*
proof
 fix *x* **assume** *x*∈*vars* (*p*∗*q*)
 then obtain *m* **where** *m*∈*keys* (*mapping-of* (*p*∗*q*)) *x*∈*keys m*
  using *vars-def* **by** *blast*
 then have *m*∈*keys* (*mapping-of p* ∗ *mapping-of q*)
  by (*simp add*: *times-mpoly.rep-eq*)
 then obtain *a b* **where** *m*=*a* + *b a* ∈ *keys* (*mapping-of p*) *b* ∈ *keys* (*mapping-of q*)
  using *keys-mult* **by** *blast*
 then have *x* ∈ *keys a* ∪ *keys b*
  using *Poly-Mapping.keys-add* ‹*x* ∈ *keys m*› **by** *force*
 then show *x* ∈ *vars p* ∪ *vars q* **unfolding** *vars-def*
  using ‹*a* ∈ *keys* (*mapping-of p*)› ‹*b* ∈ *keys* (*mapping-of q*)› **by** *blast*
qed

lemma *vars-add-monom*:
assumes *p2* = *monom m a m* ∉ *keys* (*mapping-of p1*)
shows *vars* (*p1* + *p2*) = *vars p1* ∪ *vars p2*
proof −
 have *keys* (*mapping-of p2*) ⊆ {*m*} **using** *monom-def keys-single assms* **by** *auto*
 have *keys* (*mapping-of* (*p1*+*p2*)) = *keys* (*mapping-of p1*) ∪ *keys* (*mapping-of p2*)
  using *keys-add* **by** (*metis Int-insert-right-if0* ‹*keys* (*mapping-of p2*) ⊆ {*m*}› *assms*(*2*) *inf-bot-right plus-mpoly.rep-eq subset-singletonD*)
 then show *?thesis* **unfolding** *vars-def* **by** *simp*
qed

lemma *vars-setsum*: *finite S* ⟹ *vars* ($\sum$ *m*∈*S*. *f m*) ⊆ ($\bigcup$ *m*∈*S*. *vars* (*f m*))
proof (*induction S rule*:*finite-induct*)
 case *empty*
 then show *?case* **by** (*metis UN-empty eq-iff monom-zero sum.empty single-zero vars-monom-single-cases*)
next
 case (*insert s S*)
 then have *vars* (*sum f* (*insert s S*)) = *vars* (*f s* + *sum f S*) **by** (*metis sum.insert*)
 also have ... ⊆ *vars* (*f s*) ∪ *vars* (*sum f S*) **by** (*simp add*: *vars-add*)

also have ... ⊆ ($\bigcup$ *m*∈*insert s S*. *vars* (*f m*)) **using** *insert.IH* **by** *auto*
finally show *?case* **by** *metis*
qed

lemma *coeff-monom*: *coeff* (*monom m a*) *m′* = (*a when m′*=*m*)
by (*simp add*: *coeff-def lookup-single-not-eq when-def*)

lemma *coeff-add*: *coeff p m* + *coeff q m* = *coeff* (*p*+*q*) *m*
by (*simp add*: *coeff-def lookup-add plus-mpoly.rep-eq*)

lemma *coeff-eq*: *coeff p* = *coeff q* ⟷ *p*=*q* **by** (*simp add*: *coeff-def lookup-inject mapping-of-inject*)

lemma *coeff-monom-mult*: *coeff* ((*monom m′ a*) ∗ *q*) (*m′* + *m*) = *a* ∗ *coeff q m*
unfolding *coeff-def times-mpoly.rep-eq lookup-mult mapping-of-monom lookup-single when-mult*
Sum-any-when-equal′ Groups.cancel-semigroup-add-class.add-left-cancel **by** *metis*

lemma *one-term-is-monomial*:
assumes *card* (*keys* (*mapping-of p*)) ≤ *1*
obtains *m* **where** *p* = *monom m* (*coeff p m*)
proof (*cases keys* (*mapping-of p*) = {})
case *True*
then show *?thesis* **using** *aux coeff-def empty-iff mapping-of-inject mapping-of-monom not-in-keys-iff-lookup-eq-zero single-zero* **by** (*metis* (*no-types*) *that*)
next
case *False*
then obtain *m* **where** *keys* (*mapping-of p*) = {*m*} **using** *assms* **by** (*metis One-nat-def Suc-leI antisym card-0-eq card-eq-SucD finite-keys neq0-conv*)
have *p* = *monom m* (*coeff p m*)
unfolding *mapping-of-inject*[*symmetric*]
by (*rule poly-mapping-eqI*, *metis* (*no-types*, *lifting*) ‹*keys* (*mapping-of p*) = {*m*}›
coeff-def keys-single lookup-single-eq mapping-of-monom not-in-keys-iff-lookup-eq-zero singletonD)
then show *?thesis* **..**
qed

definition *remove-term*::(*nat* $\Rightarrow_0$ *nat*) ⇒ *′a*::*zero mpoly* ⇒ *′a mpoly* **where**
remove-term m0 p = *MPoly* (*Abs-poly-mapping* (λ*m*. *coeff p m when m* ≠ *m0*))

lemma *remove-term-coeff*: *coeff* (*remove-term m0 p*) *m* = (*coeff p m when m* ≠ *m0*)
proof −
have {*m*. (*coeff p m when m* ≠ *m0*) ≠ *0*} ⊆ {*m*. *coeff p m* ≠ *0*} **by** *auto*
then have *finite* {*m*. (*coeff p m when m* ≠ *m0*) ≠ *0*} **unfolding** *coeff-def* **using** *finite-subset* **by** *auto*
then have *lookup* (*Abs-poly-mapping* (λ*m*. *coeff p m when m* ≠ *m0*)) *m* = (*coeff*

p m when m ≠ m0) **using** *lookup-Abs-poly-mapping* **by** *fastforce*
then show *?thesis* **unfolding** *remove-term-def* **using** *coeff-def* **by** (*metis* (*mono-tags*, *lifting*) *Quotient-mpoly Quotient-rep-abs-fold-unmap*)
qed

lemma *coeff-keys*: *m* ∈ *keys* (*mapping-of p*) ⟷ *coeff p m* ≠ *0*
by (*simp add*: *coeff-def in-keys-iff*)

lemma *remove-term-keys*:
shows *keys* (*mapping-of p*) − {*m*} = *keys* (*mapping-of* (*remove-term m p*)) (**is** *?A* = *?B*)
proof
show *?A* ⊆ *?B*
proof
fix *m′* **assume** *m′*∈*?A*
then show *m′* ∈ *?B* **by** (*simp add*: *coeff-keys remove-term-coeff*)
qed
show *?B* ⊆ *?A*
proof
fix *m′* **assume** *m′*∈ *?B*
then show *m′* ∈ *?A* **by** (*simp add*: *coeff-keys remove-term-coeff*)
qed
qed

lemma *remove-term-sum*: *remove-term m p* + *monom m* (*coeff p m*) = *p*
proof −
have *coeff p* = (λ*m′*. (*coeff p m′ when m′* ≠ *m*) + ((*coeff p m*) *when m′*=*m*))
unfolding *when-def* **by** *fastforce*
moreover have *coeff* (*remove-term m p* + *monom m* (*coeff p m*)) = ...
using *remove-term-coeff coeff-monom coeff-add* **by** (*metis* (*no-types*))
ultimately show *?thesis* **using** *coeff-eq* **by** *auto*
qed

lemma *mpoly-induct* [*case-names monom sum*]:
assumes *monom*:⋀*m a*. *P* (*monom m a*)
and *sum*:(⋀*p1 p2 m a*. *P p1* ⟹ *P p2* ⟹ *p2* = (*monom m a*) ⟹ *m* ∉ *keys* (*mapping-of p1*) ⟹ *P* (*p1*+*p2*))
shows *P p* **using** *assms*
using *poly-mapping-induct*[*of* λ*p* :: (*nat* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *′a*. *P* (*MPoly p*)] *MPoly-induct monom.abs-eq plus-mpoly.abs-eq*
by (*metis* (*no-types*) *MPoly-inverse UNIV-I*)

lemma *monom-pow*:*monom* (*Poly-Mapping.single v n0*) *a* ^ *n* = *monom* (*Poly-Mapping.single v* (*n0*∗*n*)) (*a* ^ *n*)
apply (*induction n*)
apply *auto*
by (*metis* (*no-types*, *lifting*) *mult-monom single-add*)

lemma *insertion-fun-single*: *insertion-fun f* (*λm.* (*a when* (*Poly-Mapping.single* (*v*::*nat*) (*n*::*nat*)) = *m*)) = *a* ∗ *f v* ^ *n* (**is** *?i* = -)
proof –
have *setsum-single*:⋀ *a f.* (∑ *m*∈{*a*}. *f m*) = *f a*
by (*metis add.right-neutral empty-Diff finite.emptyI sum.empty sum.insert-remove*)

have *1*:*?i* = (∑ *m.* (*a when Poly-Mapping.single v n* = *m*) ∗ (∏ *v. f v* ^ *lookup m v*))
unfolding *insertion-fun-def* **by** *metis*
have ∀ *m. m* ≠ *Poly-Mapping.single v n* ⟶ (*a when Poly-Mapping.single v n* = *m*) = *0* **by** *simp*

have (∑ *m*∈{*Poly-Mapping.single v n*}. (*a when Poly-Mapping.single v n* = *m*) ∗ (∏ *v. f v* ^ *lookup m v*)) = *?i*
unfolding *1 when-mult* **unfolding** *when-def* **by** *auto*
then have *2*:*?i* = *a* ∗ (∏ *va. f va* ^ *lookup* (*Poly-Mapping.single v n*) *va*)
unfolding *setsum-single*[*of λm.* (*a when Poly-Mapping.single v n* = *m*) ∗ (∏ *v. f v* ^ *lookup m v*) *Poly-Mapping.single k v*]
by *auto*
have ∀ *v0. v0*≠*v* ⟶ *lookup* (*Poly-Mapping.single v n*) *v0* = *0* **by** (*simp add*: *lookup-single-not-eq*)
then have ∀ *va. va*≠*v* ⟶ *f va* ^ *lookup* (*Poly-Mapping.single v n*) *va* = *1* **by** *simp*
then have *a* ∗ (∏ *va*∈{*v*}. *f va* ^ *lookup* (*Poly-Mapping.single v n*) *va*) = *?i* **unfolding** *2*
using *Prod-any.expand-superset*[*of* {*v*} *λva. f va* ^ *lookup* (*Poly-Mapping.single v n*) *va*, *simplified*]
by *fastforce*
then show *?thesis* **by** *simp*
qed

lemma *insertion-single*[*simp*]: *insertion f* (*monom* (*Poly-Mapping.single* (*v*::*nat*) (*n*::*nat*)) *a*) = *a* ∗ *f v* ^ *n*
using *insertion-fun-single Sum-any.cong insertion.rep-eq insertion-aux.rep-eq insertion-fun-def*
mapping-of-monom single.rep-eq **by** (*metis* (*no-types*, *lifting*))

lemma *insertion-fun-irrelevant-vars*:
fixes *p*::((*nat* $\Rightarrow_0$ *nat*) ⇒ *'a*::*comm-ring-1*)
assumes ⋀*m v. p m* ≠ *0* ⟹ *lookup m v* ≠ *0* ⟹ *f v* = *g v*
shows *insertion-fun f p* = *insertion-fun g p*
proof –
{
fix *m*::*nat*$\Rightarrow_0$*nat*
assume *p m* ≠ *0*
then have (∏ *v. f v* ^ *lookup m v*) = (∏ *v. g v* ^ *lookup m v*)
using *assms* **by** (*metis power-0*)
}
then show *?thesis* **unfolding** *insertion-fun-def* **by** (*metis* (*no-types*, *lifting*)

mult-not-zero)
qed

lemma *insertion-aux-irrelevant-vars*:
fixes *p*::((*nat* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'a*::*comm-ring-1*)
assumes $\bigwedge$*m v. lookup p m* $\neq$ *0* $\Longrightarrow$ *lookup m v* $\neq$ *0* $\Longrightarrow$ *f v* = *g v*
shows *insertion-aux f p* = *insertion-aux g p*
  using *insertion-fun-irrelevant-vars*[*of lookup p f g*] *assms*
  by (*metis insertion-aux.rep-eq*)

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 −
  {
    fix *m v* **assume** *lookup* (*mapping-of p*) *m* $\neq$ *0 lookup m v* $\neq$ *0*
    then have *v* ∈ *vars p* **unfolding** *vars-def* **by** (*meson UN-I lookup-not-eq-zero-eq-in-keys*)
    then have *f v* = *g v* **using** *assms* **by** *auto*
  }
  then show *?thesis*
    unfolding *insertion-def* **using** *insertion-aux-irrelevant-vars*[*of mapping-of p*]
    by (*metis insertion.rep-eq insertion-def*)
qed

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*
  by (*simp add*: *insertion-add reduce-nested-mpoly-def*)

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*
  by (*simp add*: *insertion-mult reduce-nested-mpoly-def*)

lemma *reduce-nested-mpoly-0*:
shows *reduce-nested-mpoly 0* = *0* **by** (*simp add*: *reduce-nested-mpoly-def*)

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 (*induction pp rule*:*mpoly-induct*)
case (*monom m a*)
then show *?case*
proof (*induction m arbitrary*:*a rule*:*poly-mapping-induct*)
case (*single v n*)
show *?case* **unfolding** *reduce-nested-mpoly-def*
apply (*simp add*: *insertion-mult monom-pow*)
using *monom-pow*[*of 0 0 f v n*] **apply** *simp*
using *insertion-single*[*of f 0 0*] **by** *auto*
next
case (*sum m1 m2 k v*)
then have *insertion f* (*insertion* (*λv. monom 0* (*f v*)) (*monom m1 a* ∗ *monom m2 1*))
= *insertion f* (*reduce-nested-mpoly* (*monom m1 a* ∗ *monom m2 1*))
unfolding *reduce-nested-mpoly-prod insertion-mult* **by** *metis*
then show *?case* **using** *mult-monom*[*of m1 a m2 1*] **by** *auto*
qed
next
case (*sum p1 p2 m a*)
then show *?case* **by** (*simp add*: *reduce-nested-mpoly-sum insertion-add*)
qed

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−
{
fix *m′* **assume** *coeff p m′* = *0*
then have *monom* (*remove-key v m′*) (*monom* (*Poly-Mapping.single v* (*lookup m′ v*)) (*coeff p m′*)) = *0*
using *monom.abs-eq monom-zero single-zero* **by** *metis*
}
then have *0*:{*a. monom* (*remove-key v a*) (*monom* (*Poly-Mapping.single v* (*lookup a v*)) (*coeff p a*)) ≠ *0*} ⊆ *S*
using ‹{*m′. coeff p m′* ≠ *0*} ⊆ *S*› **by** *fastforce*
then show *?thesis*
unfolding *extract-var-def* **using** *Sum-any.expand-superset* [*OF* ‹*finite S*› *0*] **by** *metis*
qed

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)))
  using *extract-var-finite-set coeff-def finite-lookup order-refl* **by** (*metis* (*no-types*, *lifting*) *Collect-cong sum.cong*)

lemma *extract-var-sum*: *extract-var* (*p*+*p'*) *v* = *extract-var p v* + *extract-var p' v*
proof −
  define *S* **where** *S* = {*m*. *coeff p m* ≠ *0*} ∪ {*m*. *coeff p' m* ≠ *0*} ∪ {*m*. *coeff* (*p*+*p'*) *m* ≠ *0*}
  have *subsets*:{*m*. *coeff p m* ≠ *0*} ⊆ *S* {*m*. *coeff p' m* ≠ *0*} ⊆ *S* {*m*. *coeff* (*p*+*p'*) *m* ≠ *0*} ⊆ *S*
    unfolding *S-def* **by** *auto*
  have *finite S* **unfolding** *S-def* **using** *coeff-def finite-lookup*
    by (*metis* (*mono-tags*) *Collect-disj-eq finite-Collect-disjI*)
  then show *?thesis* **unfolding**
    extract-var-finite-set[*OF subsets*(*1*) ‹*finite S*›]
    extract-var-finite-set[*OF subsets*(*2*) ‹*finite S*›]
    extract-var-finite-set[*OF subsets*(*3*) ‹*finite S*›]
    coeff-add[*symmetric*] *monom-add sum.distrib*
    by *metis*
qed

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 (*cases a* = *0*)
  assume *a* ≠ *0*
  have *0*:{*m'*. *coeff* (*monom m a*) *m'* ≠ *0*} = {*m*}
    unfolding *coeff-monom* **using** ‹*a* ≠ *0*› **by** *auto*
  show *?thesis*
    unfolding *extract-var-non-zero-coeff* **unfolding** *0* **unfolding** *coeff-monom*
    using *sum.insert*[*OF finite.emptyI*, *unfolded sum.empty add.right-neutral*] *when-def*
    by *auto*
next
  assume *a* = *0*
  have *0*:{*m'*. *coeff* (*monom m a*) *m'* ≠ *0*} = {}
    unfolding *coeff-monom* **using** ‹*a* = *0*› **by** *auto*
  show *?thesis* **unfolding** *extract-var-non-zero-coeff 0*
    using ‹*a* = *0*› *monom.abs-eq monom-zero sum.empty single-zero* **by** (*metis* (*no-types*, *lifting*))
qed

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*
unfolding *extract-var-monom remove-key-add lookup-add single-add mult-monom*
by *auto*

lemma *extract-var-single*: *extract-var* (*monom* (*Poly-Mapping.single v n*) *a*) *v* = *monom 0* (*monom* (*Poly-Mapping.single v n*) *a*)
unfolding *extract-var-monom* **by** *simp*

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*)
unfolding *extract-var-monom* **using** *assms* **by** (*metis add.right-neutral lookup-single-not-eq remove-key-sum single-zero*)

lemma *reduce-nested-mpoly-extract-var*:
fixes *p*::*'a*::*comm-ring-1 mpoly*
shows *reduce-nested-mpoly* (*extract-var p v*) = *p*
proof (*induction p rule*:*mpoly-induct*)
  case (*monom m a*)
  then show *?case*
  proof (*induction m arbitrary*:*a rule*:*poly-mapping-induct*)
    case (*single v' n*)
    show *?case*
    proof (*cases v' = v*)
      case *True*
      then show *?thesis*
        by (*metis* (*no-types*, *lifting*) *insertion-single mult.right-neutral power-0 reduce-nested-mpoly-def single-zero extract-var-single*)
    next
      case *False*
    then show *?thesis* **unfolding** *extract-var-single'*[*OF False*] *reduce-nested-mpoly-def insertion-single*
        by (*simp add*: *monom-pow mult-monom*)
    qed
  next
    case (*sum m m' v n a*)
    then show *?case*
        using *extract-var-monom-mult*[*of m m' a 1*] *reduce-nested-mpoly-prod* **by** (*metis mult.right-neutral mult-monom*)
  qed
next
  case (*sum p1 p2 m a*)
  then show *?case* **unfolding** *extract-var-sum reduce-nested-mpoly-sum* **by** *auto*
qed

lemma *vars-extract-var-subset*: *vars* (*extract-var p v*) ⊆ *vars p*
proof
  have *finite* {*m'*. *coeff p m'* ≠ *0*} **by** (*simp add*: *coeff-def*)
  fix *x* **assume** *x* ∈ *vars* (*extract-var p v*)
  then have *x* ∈ *vars* (∑ *m*∈{*m'*. *coeff p m'* ≠ *0*}. *monom* (*remove-key v m*)

(*monom* (*Poly-Mapping.single v* (*lookup m v*)) (*coeff p m*)))
unfolding *extract-var-non-zero-coeff* **by** *metis*
then have $x \in (\bigcup m{\in}\{m'.\ coeff\ p\ m' \neq 0\}.\ vars\ (monom\ (remove\text{-}key\ v\ m)\ (monom\ (Poly\text{-}Mapping.single\ v\ (lookup\ m\ v))\ (coeff\ p\ m))))$
using *vars-setsum*[*OF* ‹*finite* {*m'*. *coeff p m'* ≠ *0*}›] **by** *auto*
then obtain *m* **where** $m{\in}\{m'.\ coeff\ p\ m' \neq 0\}$ $x \in vars\ (monom\ (remove\text{-}key\ v\ m)\ (monom\ (Poly\text{-}Mapping.single\ v\ (lookup\ m\ v))\ (coeff\ p\ m)))$
by *blast*
show $x \in vars\ p$ **by** (*metis* (*mono-tags*, *lifting*) *DiffD1 UN-I* ‹$m \in \{m'.\ coeff\ p\ m' \neq 0\}$›
‹$x \in vars\ (monom\ (remove\text{-}key\ v\ m)\ (monom\ (Poly\text{-}Mapping.single\ v\ (lookup\ m\ v))\ (coeff\ p\ m)))$›
coeff-keys mem-Collect-eq remove-key-keys subsetCE vars-def vars-monom-subset)
qed

lemma *v-not-in-vars-extract-var*: $v \notin vars\ (extract\text{-}var\ p\ v)$
proof −
have *finite* {*m'*. *coeff p m'* ≠ *0*} **by** (*simp add*: *coeff-def*)
have $\bigwedge m.\ m{\in}\{m'.\ coeff\ p\ m' \neq 0\} \Longrightarrow v \notin vars\ (monom\ (remove\text{-}key\ v\ m)\ (monom\ (Poly\text{-}Mapping.single\ v\ (lookup\ m\ v))\ (coeff\ p\ m)))$
by (*metis Diff-iff remove-key-keys singletonI subsetCE vars-monom-subset*)
then have $v \notin (\bigcup m{\in}\{m'.\ coeff\ p\ m' \neq 0\}.\ vars\ (monom\ (remove\text{-}key\ v\ m)\ (monom\ (Poly\text{-}Mapping.single\ v\ (lookup\ m\ v))\ (coeff\ p\ m))))$
by *simp*
then show *?thesis*
unfolding *extract-var-non-zero-coeff* **using** *vars-setsum*[*OF* ‹*finite* {*m'*. *coeff p m'* ≠ *0*}›] **by** *blast*
qed

lemma *vars-coeff-extract-var*: $vars\ (coeff\ (extract\text{-}var\ p\ v)\ j) \subseteq \{v\}$
proof (*induction p rule*:*mpoly-induct*)
case (*monom m a*)
then show *?case* **unfolding** *extract-var-monom coeff-monom vars-monom-single-cases*
by (*metis monom-zero single-zero vars-monom-single when-def*)
next
case (*sum p1 p2 m a*)
then show *?case* **unfolding** *extract-var-sum coeff-add*[*symmetric*]
by (*metis* (*no-types*, *lifting*) *Un-insert-right insert-absorb2 subset-insertI2 subset-singletonD sup-bot.right-neutral vars-add*)
qed

definition *replace-coeff*
where $replace\text{-}coeff\ f\ p = MPoly\ (Abs\text{-}poly\text{-}mapping\ (\lambda m.\ f\ (lookup\ (mapping\text{-}of\ p)\ m)))$

lemma *coeff-replace-coeff*:
assumes $f\ 0 = 0$
shows $coeff\ (replace\text{-}coeff\ f\ p)\ m = f\ (coeff\ p\ m)$
proof −

have *0*:*finite* {*m*. *f* (*lookup* (*mapping-of p*) *m*) ≠ *0*}
unfolding *coeff-def*[*symmetric*] **by** (*metis* (*mono-tags*, *lifting*) *Collect-mono assms*(*1*) *coeff-def finite-lookup finite-subset*)+
then show *?thesis* **unfolding** *replace-coeff-def coeff-def* **using** *lookup-Abs-poly-mapping*[*OF 0*]
by (*metis* (*mono-tags*, *lifting*) *Quotient-mpoly Quotient-rep-abs-fold-unmap*)
qed

lemma *replace-coeff-monom*:
assumes *f 0* = *0*
shows *replace-coeff f* (*monom m a*) = *monom m* (*f a*)
unfolding *replace-coeff-def*
unfolding *mapping-of-inject*[*symmetric*] *lookup-inject*[*symmetric*] **apply** (*rule HOL.ext*)
unfolding *lookup-single mapping-of-monom fun-when*[*of f*, *OF* ‹*f 0* = *0*›]
by (*metis coeff-def coeff-monom lookup-single lookup-single-not-eq monom.abs-eq single.abs-eq*)

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 −
have *finite* {*m*. *f* (*lookup* (*mapping-of p1*) *m*) ≠ *0*}
finite {*m*. *f* (*lookup* (*mapping-of p2*) *m*) ≠ *0*}
unfolding *coeff-def*[*symmetric*] **by** (*metis* (*mono-tags*, *lifting*) *Collect-mono assms*(*1*) *coeff-def finite-lookup finite-subset*)+
then show *?thesis*
unfolding *replace-coeff-def plus-mpoly.rep-eq* **unfolding** *Poly-Mapping.plus-poly-mapping.rep-eq*
unfolding *assms*(*2*) *plus-mpoly.abs-eq* **using** *Poly-Mapping.plus-poly-mapping.abs-eq*[*unfolded eq-onp-def*] **by** *fastforce*
qed

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 (*induction pp rule*:*mpoly-induct*)
case (*monom m a*)
then show *?case*
proof (*induction m arbitrary*:*a rule*:*poly-mapping-induct*)
case (*single v n*)
show *?case* **unfolding** *reduce-nested-mpoly-def* **unfolding** *replace-coeff-monom*[*of insertion f*, *OF insertion-zero*]
insertion-single insertion-mult **using** *insertion-single* **by** (*simp add*: *monom-pow*)
next
case (*sum m1 m2 k v*)
have *replace-coeff* (*insertion f*) (*monom m1 a* ∗ *monom m2 1*) = *replace-coeff* (*insertion f*) (*monom m1 a*) ∗ *replace-coeff* (*insertion f*) (*monom m2 1*)

by (*simp add*: *mult-monom replace-coeff-monom*)
then have *insertion f* (*replace-coeff* (*insertion f*) (*monom m1 a* ∗ *monom m2 1*)) = *insertion f* (*reduce-nested-mpoly* (*monom m1 a* ∗ *monom m2 1*))
unfolding *reduce-nested-mpoly-prod insertion-mult*
by (*simp add*: *insertion-mult sum.IH*(*1*) *sum.IH*(*2*))
then show *?case* **using** *mult-monom*[*of m1 a m2 1*] **by** *auto*
qed
next
case (*sum p1 p2 m a*)
then show *?case* **using** *reduce-nested-mpoly-sum insertion-add*
replace-coeff-add[*of insertion f*, *OF insertion-zero insertion-add*] **by** *metis*
qed

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*)
by (*induction p rule*:*mpoly-induct*;*simp add*: *assms extract-var-monom replace-coeff-monom extract-var-sum insertion-add replace-coeff-add*)

lemma *vars-replace-coeff*:
assumes *f 0* = *0*
shows *vars* (*replace-coeff f p*) ⊆ *vars p*
unfolding *vars-def* **apply** (*rule subsetI*) **unfolding** *mem-simps*(*8*) *coeff-keys*
using *assms coeff-replace-coeff* **by** (*metis coeff-keys*)

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*
unfolding *polyfun-def* **by** *metis*

lemma *polyfun-subset*: *N*⊆*N'* ⟹ *polyfun N f* ⟹ *polyfun N' f*
unfolding *polyfun-def* **by** *blast*

lemma *polyfun-const*: *polyfun N* (λ-. *c*)
proof −
have ⋀*x*. *insertion x* (*monom 0 c*) = *c* **using** *insertion-single* **by** (*metis insertion-one monom-one mult.commute mult.right-neutral single-zero*)
then show *?thesis* **unfolding** *polyfun-def* **by** (*metis* (*full-types*) *empty-iff keys-single single-zero subsetI subset-antisym vars-monom-subset*)
qed

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

proof −
  obtain *p1 p2* **where** *vars p1* ⊆ *N* ∀*x*. *insertion x p1* = *f x*
                    vars p2 ⊆ *N* ∀*x*. *insertion x p2* = *g x*
    using *polyfun-def assms* **by** *metis*
  then have *vars* (*p1* + *p2*) ⊆ *N* ∀*x*. *insertion x* (*p1* + *p2*) = *f x* + *g x*
    using *vars-add* **using** *Un-iff subsetCE subsetI* **apply** *blast*
    by (*simp add*: ‹∀*x*. *insertion x p1* = *f x*› ‹∀*x*. *insertion x p2* = *g x*› *insertion-add*)
  then show *?thesis* **using** *polyfun-def* **by** *blast*
qed

lemma *polyfun-mult*:
assumes *polyfun N f polyfun N g*
shows *polyfun N* (*λx*. *f x* ∗ *g x*)
proof −
  obtain *p1 p2* **where** *vars p1* ⊆ *N* ∀*x*. *insertion x p1* = *f x*
                    vars p2 ⊆ *N* ∀*x*. *insertion x p2* = *g x*
    using *polyfun-def assms* **by** *metis*
  then have *vars* (*p1* ∗ *p2*) ⊆ *N* ∀*x*. *insertion x* (*p1* ∗ *p2*) = *f x* ∗ *g x*
    using *vars-mult* **using** *Un-iff subsetCE subsetI* **apply** *blast*
    by (*simp add*: ‹∀*x*. *insertion x p1* = *f x*› ‹∀*x*. *insertion x p2* = *g x*› *insertion-mult*)
  then show *?thesis* **using** *polyfun-def* **by** *blast*
qed

lemma *polyfun-Sum*:
assumes *finite I*
assumes ⋀*i*. *i*∈*I* ⟹ *polyfun N* (*f i*)
shows *polyfun N* (*λx*. ∑ *i*∈*I*. *f i x*)
  using *assms*
  apply (*induction I rule*:*finite-induct*)
  apply (*simp add*: *polyfun-const*)
  using *comm-monoid-add-class.sum.insert polyfun-add* **by** *fastforce*

lemma *polyfun-Prod*:
assumes *finite I*
assumes ⋀*i*. *i*∈*I* ⟹ *polyfun N* (*f i*)
shows *polyfun N* (*λx*. ∏ *i*∈*I*. *f i x*)
  using *assms*
  apply (*induction I rule*:*finite-induct*)
  apply (*simp add*: *polyfun-const*)
  using *comm-monoid-add-class.sum.insert polyfun-mult* **by** *fastforce*

lemma *polyfun-single*:
assumes *i*∈*N*
shows *polyfun N* (*λx*. *x i*)
proof −
  have ∀*f*. *insertion f* (*monom* (*Poly-Mapping.single i 1*) *1*) = *f i* **using** *insertion-single* **by** *simp*

then show *?thesis* **unfolding** *polyfun-def*
using *vars-monom-single[of i 1 1] One-nat-def assms singletonD subset-eq*
by *blast*
qed

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*: $s \in Keys\ F \longleftrightarrow (\exists f{\in}F.\ s \in keys\ f)$
unfolding *Keys-def* **by** *simp*

lemma *in-KeysI*:
assumes $s \in keys\ f$ **and** $f \in F$
shows $s \in Keys\ F$
unfolding *in-Keys* **using** *assms* **..**

lemma *in-KeysE*:
  assumes $s \in$ *Keys F*
  obtains *f* **where** $s \in$ *keys f* **and** $f \in F$
  using *assms* **unfolding** *in-Keys* **..**

lemma *Keys-mono*:
  assumes $A \subseteq B$
  shows *Keys A* $\subseteq$ *Keys B*
  using *assms* **by** (*auto simp add*: *Keys-def*)

lemma *Keys-insert*: *Keys* (*insert a A*) = *keys a* $\cup$ *Keys A*
  by (*simp add*: *Keys-def*)

lemma *Keys-Un*: *Keys* ($A \cup B$) = *Keys A* $\cup$ *Keys B*
  by (*simp add*: *Keys-def*)

lemma *finite-Keys*:
  assumes *finite A*
  shows *finite* (*Keys A*)
  unfolding *Keys-def* **by** (*rule*, *fact assms*, *rule finite-keys*)

lemma *Keys-not-empty*:
  assumes $a \in A$ **and** $a \neq 0$
  shows *Keys A* $\neq \{\}$
proof
  assume *Keys A* $= \{\}$
  from ‹$a \neq 0$› **have** *keys a* $\neq \{\}$ **using** *aux* **by** *fastforce*
  then obtain *s* **where** $s \in$ *keys a* **by** *blast*
  from *this assms*(*1*) **have** $s \in$ *Keys A* **by** (*rule in-KeysI*)
  with ‹*Keys A* $= \{\}$› **show** *False* **by** *simp*
qed

lemma *Keys-empty* [*simp*]: *Keys* $\{\} = \{\}$
  by (*simp add*: *Keys-def*)

lemma *Keys-zero* [*simp*]: *Keys* $\{0\} = \{\}$
  by (*simp add*: *Keys-def*)

lemma *keys-subset-Keys*:
  assumes $f \in F$
  shows *keys f* $\subseteq$ *Keys F*
  using *in-KeysI*[*OF - assms*] **by** *auto*

lemma *Keys-minus*: *Keys* ($A - B$) $\subseteq$ *Keys A*
  by (*auto simp add*: *Keys-def*)

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

case *True*
hence $(A - \{0\}) \cup \{0\} = A$ **by** *auto*
hence *Keys* $A =$ *Keys* $((A - \{0\}) \cup \{0\})$ **by** *simp*
also have ... = *Keys* $(A - \{0\}) \cup$ *Keys* $\{0::('a \Rightarrow_0 'b)\}$ **by** (*fact Keys-Un*)
also have ... = *Keys* $(A - \{0\})$ **by** *simp*
finally show *?thesis* **by** *simp*
next
case *False*
hence $A - \{0\} = A$ **by** *simp*
thus *?thesis* **by** *simp*
qed

6.2 Constant *except*

definition *except-fun* :: $('a \Rightarrow 'b) \Rightarrow 'a$ *set* $\Rightarrow ('a \Rightarrow 'b::zero)$
where *except-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 −
fix $p::'a \Rightarrow 'b$ **and** $S::'a$ *set*
assume *finite* $\{t.\ p\ t \neq 0\}$
show *finite* $\{t.$ *except-fun* $p\ S\ t \neq 0\}$
proof (*rule finite-subset*[*of* - $\{t.\ p\ t \neq 0\}$], *rule*)
fix *u*
assume $u \in \{t.$ *except-fun* $p\ S\ t \neq 0\}$
hence $p\ u \neq 0$ **by** (*simp add*: *except-fun-def*)
thus $u \in \{t.\ p\ t \neq 0\}$ **by** *simp*
qed *fact*
qed

lemma *lookup-except-when*: *lookup* (*except p S*) = $(\lambda t.$ *lookup p t when* $t \notin S)$
by (*auto simp*: *except.rep-eq except-fun-def*)

lemma *lookup-except*: *lookup* (*except p S*) = $(\lambda t.$ *if* $t \in S$ *then 0 else lookup p t*)
by (*rule ext*) (*simp add*: *lookup-except-when*)

lemma *lookup-except-singleton*: *lookup* (*except p* $\{t\}$) $t = 0$
by (*simp add*: *lookup-except*)

lemma *except-zero* [*simp*]: *except* $0\ S = 0$
by (*rule poly-mapping-eqI*) (*simp add*: *lookup-except*)

lemma *lookup-except-eq-idI*:
assumes $t \notin S$
shows *lookup* (*except p S*) t = *lookup p t*
using *assms* **by** (*simp add*: *lookup-except*)

lemma *lookup-except-eq-zeroI*:
assumes $t \in S$

shows *lookup* (*except p S*) *t* = *0*
using *assms* **by** (*simp add*: *lookup-except*)

lemma *except-empty* [*simp*]: *except p* {} = *p*
by (*rule poly-mapping-eqI*) (*simp add*: *lookup-except*)

lemma *except-eq-zeroI*:
assumes *keys p* ⊆ *S*
shows *except p S* = *0*
proof (*rule poly-mapping-eqI*, *simp*)
fix *t*
show *lookup* (*except p S*) *t* = *0*
proof (*cases t* ∈ *S*)
case *True*
thus *?thesis* **by** (*rule lookup-except-eq-zeroI*)
next
case *False* **then show** *?thesis*
by (*metis assms in-keys-iff lookup-except-eq-idI subset-eq*)
qed
qed

lemma *except-eq-zeroE*:
assumes *except p S* = *0*
shows *keys p* ⊆ *S*
by (*metis assms aux in-keys-iff lookup-except-eq-idI subset-iff*)

lemma *except-eq-zero-iff*: *except p S* = *0* ⟷ *keys p* ⊆ *S*
by (*rule*, *elim except-eq-zeroE*, *elim except-eq-zeroI*)

lemma *except-keys* [*simp*]: *except p* (*keys p*) = *0*
by (*rule except-eq-zeroI*, *rule subset-refl*)

lemma *plus-except*: *p* = *Poly-Mapping.single t* (*lookup p t*) + *except p* {*t*}
by (*rule poly-mapping-eqI*, *simp add*: *lookup-add lookup-single lookup-except when-def split*: *if-split*)

lemma *keys-except*: *keys* (*except p S*) = *keys p* − *S*
by (*transfer*, *auto simp*: *except-fun-def*)

lemma *except-single*: *except* (*Poly-Mapping.single u c*) *S* = (*Poly-Mapping.single u c when u* ∉ *S*)
by (*rule poly-mapping-eqI*) (*simp add*: *lookup-except lookup-single when-def*)

lemma *except-plus*: *except* (*p* + *q*) *S* = *except p S* + *except q S*
by (*rule poly-mapping-eqI*) (*simp add*: *lookup-except lookup-add*)

lemma *except-minus*: *except* (*p* − *q*) *S* = *except p S* − *except q S*
by (*rule poly-mapping-eqI*) (*simp add*: *lookup-except lookup-minus*)

lemma *except-uminus*: *except* (− *p*) *S* = − *except p S*
by (*rule poly-mapping-eqI*) (*simp add*: *lookup-except*)

lemma *except-except*: *except* (*except p S*) *T* = *except p* (*S* ∪ *T*)
by (*rule poly-mapping-eqI*) (*simp add*: *lookup-except*)

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 (*rule poly-mapping-eqI*)
fix *t*
show *lookup p t* = *lookup q t*
proof (*cases t* ∈ *keys p*)
case *True*
thus *?thesis* **by** (*rule a2*)
next
case *False*
moreover from *this* **have** *t* ∉ *keys q* **unfolding** *a1* **.**
ultimately have *lookup p t* = *0* **and** *lookup q t* = *0* **unfolding** *in-keys-iff* **by** *simp-all*
thus *?thesis* **by** *simp*
qed
qed

lemma *except-id-iff*: *except p S* = *p* ⟷ *keys p* ∩ *S* = {}
by (*metis Diff-Diff-Int Diff-eq-empty-iff Diff-triv inf-le2 keys-except lookup-except-eq-idI lookup-except-eq-zeroI not-in-keys-iff-lookup-eq-zero poly-mapping-keys-eqI*)

lemma *keys-subset-wf*:
wfP (λ*p q*::(′*a*, ′*b*::*zero*) *poly-mapping*. *keys p* ⊂ *keys q*)
unfolding *wfp-def*
proof (*intro wfI-min*)
fix *x*::(′*a*, ′*b*) *poly-mapping* **and** *Q*
assume *x-in*: *x* ∈ *Q*
let *?Q0* = *card* ‘ *keys* ‘ *Q*
from *x-in* **have** *card* (*keys x*) ∈ *?Q0* **by** *simp*
from *wfE-min*[*OF wf this*] **obtain** *z0*
where *z0-in*: *z0* ∈ *?Q0* **and** *z0-min*: ⋀*y*. (*y*, *z0*) ∈ {(*x*, *y*). *x* < *y*} ⟹ *y* ∉ *?Q0* **by** *auto*
from *z0-in* **obtain** *z* **where** *z0-def*: *z0* = *card* (*keys z*) **and** *z* ∈ *Q* **by** *auto*
show ∃ *z*∈*Q*. ∀ *y*. (*y*, *z*) ∈ {(*p*, *q*). *keys p* ⊂ *keys q*} ⟶ *y* ∉ *Q*
proof (*intro bexI*[*of - z*], *rule*, *rule*)
fix *y*::(′*a*, ′*b*) *poly-mapping*
let *?y0* = *card* (*keys y*)
assume (*y*, *z*) ∈ {(*p*, *q*). *keys p* ⊂ *keys q*}
hence *keys y* ⊂ *keys z* **by** *simp*
hence *?y0* < *z0* **unfolding** *z0-def* **by** (*simp add*: *psubset-card-mono*)
hence (*?y0*, *z0*) ∈ {(*x*, *y*). *x* < *y*} **by** *simp*

```
    from z0-min[OF this] show y ∉ Q by auto
  qed (fact)
qed

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 (induct rule: wfp-induct[OF keys-subset-wf])
  fix p::('a, 'b) poly-mapping
  assume ∀ q. keys q ⊂ keys p ⟶ P q
  hence IH: ⋀q. keys q ⊂ keys p ⟹ P q by simp
  show P p
  proof (cases p = 0)
    case True
    thus ?thesis using base by simp
  next
    case False
    hence keys p ≠ {} by simp
    then obtain t where t ∈ keys p by blast
    show ?thesis
     proof (rule ind, fact, fact, rule IH, simp only: keys-except, rule, rule Diff-subset,
rule)
      assume keys p − {t} = keys p
      hence t ∉ keys p by blast
      from this ‹t ∈ keys p› show False ..
    qed
  qed
qed

lemma poly-mapping-except-induct':
  assumes ⋀p. (⋀t. t ∈ keys p ⟹ P (except p {t})) ⟹ P p
  shows P p
proof (induct card (keys p) arbitrary: p)
  case 0
  with finite-keys[of p] have keys p = {} by simp
  show ?case by (rule assms, simp add: ‹keys p = {}›)
next
  case step: (Suc n)
  show ?case
  proof (rule assms)
    fix t
    assume t ∈ keys p
    show P (except p {t})
    proof (rule step(1), simp add: keys-except)
      from step(2) ‹t ∈ keys p› finite-keys[of p] show n = card (keys p − {t}) by
simp
    qed
  qed
```

qed

lemma *poly-mapping-plus-induct*:
 assumes *P 0* **and** $\bigwedge p\ c\ t.\ c \neq 0 \Longrightarrow t \notin keys\ p \Longrightarrow P\ p \Longrightarrow P\ (Poly\text{-}Mapping.single\ t\ c + p)$
 shows *P p*
proof (*induct card* (*keys p*) *arbitrary*: *p*)
 case *0*
 with *finite-keys*[*of p*] **have** *keys p* = {} **by** *simp*
 hence *p* = *0* **by** *simp*
 with *assms*(*1*) **show** *?case* **by** *simp*
next
 case *step*: (*Suc n*)
 from *step*(*2*) **obtain** *t* **where** *t*: $t \in keys\ p$ **by** (*metis card-eq-SucD insert-iff*)
 define *c* **where** *c* = *lookup p t*
 define *q* **where** *q* = *except p* {*t*}
 have ∗: *p* = *Poly-Mapping.single t c* + *q*
  by (*rule poly-mapping-eqI*, *simp add*: *lookup-add lookup-single Poly-Mapping.when-def*, *intro conjI impI*,
   simp add: *q-def lookup-except c-def*, *simp add*: *q-def lookup-except-eq-idI*)
 show *?case*
 proof (*simp only*: ∗, *rule assms*(*2*))
  from *t* **show** $c \neq 0$
   using *c-def* **by** *auto*
 next
  show $t \notin keys\ q$ **by** (*simp add*: *q-def keys-except*)
 next
  show *P q*
  proof (*rule step*(*1*))
   from *step*(*2*) ‹$t \in keys\ p$› **show** *n* = *card* (*keys q*) **unfolding** *q-def keys-except*
    by (*metis Suc-inject card.remove finite-keys*)
  qed
 qed
qed

lemma *except-Diff-singleton*: *except p* (*keys p* − {*t*}) = *Poly-Mapping.single t* (*lookup p t*)
 by (*rule poly-mapping-eqI*) (*simp add*: *lookup-single in-keys-iff lookup-except when-def*)

lemma *except-Un-plus-Int*: *except p* ($U \cup V$) + *except p* ($U \cap V$) = *except p U* + *except p V*
 by (*rule poly-mapping-eqI*) (*simp add*: *lookup-except lookup-add*)

corollary *except-Int*:
 assumes $keys\ p \subseteq U \cup V$
 shows *except p* ($U \cap V$) = *except p U* + *except p V*
proof −
 from *assms* **have** *except p* ($U \cup V$) = *0* **by** (*rule except-eq-zeroI*)

hence *except p* $(U \cap V)$ = *except p* $(U \cup V)$ + *except p* $(U \cap V)$ **by** *simp*
also have ... = *except p U* + *except p V* **by** (*fact except-Un-plus-Int*)
finally show *?thesis* **.**
qed

lemma *except-keys-Int* [*simp*]: *except p* (*keys p* $\cap$ *U*) = *except p U*
by (*rule poly-mapping-eqI*) (*simp add*: *in-keys-iff lookup-except*)

lemma *except-Int-keys* [*simp*]: *except p* (*U* $\cap$ *keys p*) = *except p U*
by (*simp only*: *Int-commute*[*of U*] *except-keys-Int*)

lemma *except-keys-Diff*: *except p* (*keys p* − *U*) = *except p* (− *U*)
proof −
have *except p* (*keys p* − *U*) = *except p* (*keys p* $\cap$ (− *U*)) **by** (*simp only*: *Diff-eq*)
also have ... = *except p* (− *U*) **by** *simp*
finally show *?thesis* **.**
qed

lemma *except-decomp*: *p* = *except p U* + *except p* (− *U*)
by (*rule poly-mapping-eqI*) (*simp add*: *lookup-except lookup-add*)

corollary *except-Compl*: *except p* (− *U*) = *p* − *except p U*
by (*metis add-diff-cancel-left′ except-decomp*)

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*
unfolding *adds-def* **..**

lemma *addsE* [*elim?*]: *b adds a* ⟹ (⋀*k*. *a* = *b* + *k* ⟹ *P*) ⟹ *P*
unfolding *adds-def* **by** *blast*

end

context *comm-monoid-add*
begin

lemma *adds-refl* [*simp*]: *a adds a*
proof
show *a* = *a* + *0* **by** *simp*
qed

lemma *adds-trans* [*trans*]:
assumes *a adds b* **and** *b adds c*

shows *a adds c*
proof −
from *assms* **obtain** *v* **where** *b* = *a* + *v*
by (*auto elim*!: *addsE*)
moreover from *assms* **obtain** *w* **where** *c* = *b* + *w*
by (*auto elim*!: *addsE*)
ultimately have *c* = *a* + (*v* + *w*)
by (*simp add*: *add.assoc*)
then show *?thesis* **..**
qed

lemma *subset-divisors-adds*: {*c*. *c adds a*} ⊆ {*c*. *c adds b*} ⟷ *a adds b*
by (*auto simp add*: *subset-iff intro*: *adds-trans*)

lemma *strict-subset-divisors-adds*: {*c*. *c adds a*} ⊂ {*c*. *c adds b*} ⟷ *a adds b* ∧ ¬ *b adds a*
by (*auto simp add*: *subset-iff intro*: *adds-trans*)

lemma *zero-adds* [*simp*]: *0 adds a*
by (*auto intro*!: *addsI*)

lemma *adds-plus-right* [*simp*]: *a adds c* ⟹ *a adds* (*b* + *c*)
by (*auto intro*!: *add.left-commute addsI elim*!: *addsE*)

lemma *adds-plus-left* [*simp*]: *a adds b* ⟹ *a adds* (*b* + *c*)
using *adds-plus-right* [*of a b c*] **by** (*simp add*: *ac-simps*)

lemma *adds-triv-right* [*simp*]: *a adds b* + *a*
by (*rule adds-plus-right*) (*rule adds-refl*)

lemma *adds-triv-left* [*simp*]: *a adds a* + *b*
by (*rule adds-plus-left*) (*rule adds-refl*)

lemma *plus-adds-mono*:
assumes *a adds b*
and *c adds d*
shows *a* + *c adds b* + *d*
proof −
from ‹*a adds b*› **obtain** *b′* **where** *b* = *a* + *b′* **..**
moreover from ‹*c adds d*› **obtain** *d′* **where** *d* = *c* + *d′* **..**
ultimately have *b* + *d* = (*a* + *c*) + (*b′* + *d′*)
by (*simp add*: *ac-simps*)
then show *?thesis* **..**
qed

lemma *plus-adds-left*: *a* + *b adds c* ⟹ *a adds c*
by (*simp add*: *adds-def add.assoc*) *blast*

lemma *plus-adds-right*: *a* + *b adds c* ⟹ *b adds c*

```
  using plus-adds-left [of b a c] by (simp add: ac-simps)

end

class ninv-comm-monoid-add = comm-monoid-add +
  assumes plus-eq-zero: s + t = 0 ⟹ s = 0
begin

lemma plus-eq-zero-2: t = 0 if s + t = 0
  using that
  by (simp only: add-commute[of s t] plus-eq-zero)

lemma adds-zero: s adds 0 ⟷ (s = 0)
proof
  assume s adds 0
  from this obtain k where 0 = s + k unfolding adds-def ..
  from this plus-eq-zero[of s k] show s = 0
    by blast
next
  assume s = 0
  thus s adds 0 by simp
qed

end

context canonically-ordered-monoid-add
begin
subclass ninv-comm-monoid-add by (standard, simp)
end

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

lemma adds-canc: s + u adds t + u ⟷ s adds t for s t u::'a
  unfolding adds-def
  apply auto
  apply (metis local.add.left-commute local.add-diff-cancel-left' local.add-diff-cancel-right')
  using add-assoc add-commute by auto

lemma adds-canc-2: u + s adds u + t ⟷ s adds t
  by (simp add: adds-canc ac-simps)

lemma add-minus-2: (s + t) − s = t
  by simp

lemma adds-minus:
  assumes s adds t
  shows (t − s) + s = t
proof −
```

from *assms adds-def*[*of s t*] **obtain** *u* **where** *u*: *t* = *u* + *s* **by** (*auto simp*: *ac-simps*)
then have *t* − *s* = *u*
by *simp*
thus *?thesis* **using** *u* **by** *simp*
qed

lemma *plus-adds-0*:
assumes (*s* + *t*) *adds u*
shows *s adds* (*u* − *t*)
proof −
from *assms* **have** (*s* + *t*) *adds* ((*u* − *t*) + *t*) **using** *adds-minus local.plus-adds-right* **by** *presburger*
thus *?thesis* **using** *adds-canc*[*of s t u* − *t*] **by** *simp*
qed

lemma *plus-adds-2*:
assumes *t adds u* **and** *s adds* (*u* − *t*)
shows (*s* + *t*) *adds u*
by (*metis adds-canc adds-minus assms*)

lemma *plus-adds*:
shows (*s* + *t*) *adds u* ⟷ (*t adds u* ∧ *s adds* (*u* − *t*))
proof
assume *a1*: (*s* + *t*) *adds u*
show *t adds u* ∧ *s adds* (*u* − *t*)
proof
from *plus-adds-right*[*OF a1*] **show** *t adds u* **.**
next
from *plus-adds-0*[*OF a1*] **show** *s adds* (*u* − *t*) **.**
qed
next
assume *t adds u* ∧ *s adds* (*u* − *t*)
hence *t adds u* **and** *s adds* (*u* − *t*) **by** *auto*
from *plus-adds-2*[*OF* ‹*t adds u*› ‹*s adds* (*u* − *t*)›] **show** (*s* + *t*) *adds u* **.**
qed

lemma *minus-plus*:
assumes *s adds t*
shows (*t* − *s*) + *u* = (*t* + *u*) − *s*
proof −
from *assms* **obtain** *k* **where** *k*: *t* = *s* + *k* **unfolding** *adds-def* **..**
hence *t* − *s* = *k* **by** *simp*
also from *k* **have** (*t* + *u*) − *s* = *k* + *u*
by (*simp add*: *add-assoc*)
finally show *?thesis* **by** *simp*
qed

lemma *minus-plus-minus*:

assumes *s adds t* **and** *u adds v*
shows $(t - s) + (v - u) = (t + v) - (s + u)$
using *add-commute assms(1) assms(2) diff-diff-add minus-plus* **by** *auto*

lemma *minus-plus-minus-cancel*:
assumes *u adds t* **and** *s adds u*
shows $(t - u) + (u - s) = t - s$
by (*metis assms(1) assms(2) local.add-diff-cancel-left′ local.add-diff-cancel-right local.addsE minus-plus*)

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*)
by (*simp only*: *lcs-comm*[*of s t*], *rule adds-lcs*)

lemma *lcs-adds-plus*: *lcs s t adds s* + *t* **by** (*simp add*: *lcs-adds*)

"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*
unfolding *gcs-def* **by** (*rule adds-minus*, *fact lcs-adds-plus*)

lemma *gcs-adds*: (*gcs s t*) *adds s*
proof −
have *t adds* (*lcs s t*) (**is** *t adds ?l*) **unfolding** *lcs-comm*[*of s t*] **by** (*fact adds-lcs*)
then obtain *u* **where** *eq1*: *?l* = *t* + *u* **unfolding** *adds-def* **..**
from *lcs-adds-plus*[*of s t*] **obtain** *v* **where** *eq2*: *s* + *t* = *?l* + *v* **unfolding** *adds-def* **..**
hence *t* + *s* = *t* + (*u* + *v*) **unfolding** *eq1* **by** (*simp add*: *ac-simps*)
hence *s*: *s* = *u* + *v* **unfolding** *add-left-cancel* **.**
show *?thesis* **unfolding** *eq2 gcs-def* **unfolding** *s* **by** *simp*
qed

lemma *gcs-comm*: *gcs s t* = *gcs t s* **unfolding** *gcs-def* **by** (*simp add*: *lcs-comm ac-simps*)

lemma *gcs-adds-2*: (*gcs s t*) *adds t*
  by (*simp only*: *gcs-comm*[*of s t*], *rule gcs-adds*)

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 −
  from ‹*s adds t*› **obtain** *u* **where** *u-def*: *t* = *s* + *u* **unfolding** *adds-def* **..**
  from ‹*t adds s*› **obtain** *v* **where** *v-def*: *s* = *t* + *v* **unfolding** *adds-def* **..**
  from *u-def v-def* **have** *s* = (*s* + *u*) + *v* **by** (*simp add*: *ac-simps*)
  hence *s* + *0* = *s* + (*u* + *v*) **by** (*simp add*: *ac-simps*)
  hence *u* + *v* = *0* **by** *simp*
  hence *u* = *0* **using** *plus-eq-zero*[*of u v*] **by** *simp*
  thus *?thesis* **using** *u-def* **by** *simp*
qed

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*
  by (*rule adds-antisym*, *rule* ∗, *fact adds-lcs*, *fact adds-lcs-2*, *rule lcs-adds*, *fact*+)

lemma *lcs-zero*: *lcs 0 t* = *t*
  by (*rule lcs-unique*[*symmetric*], *fact zero-adds*, *fact adds-refl*)

lemma *lcs-plus-left*: *lcs* (*u* + *s*) (*u* + *t*) = *u* + *lcs s t*
proof (*rule lcs-unique*[*symmetric*], *simp-all only*: *adds-canc-2*, *fact adds-lcs*, *fact adds-lcs-2*,
    simp add: *add.commute*[*of u*] *plus-adds*)
  fix *v*
  assume *u adds v* ∧ *s adds v* − *u*
  hence *s adds v* − *u* **..**
  assume *t adds v* − *u*
  with ‹*s adds v* − *u*› **show** *lcs s t adds v* − *u* **by** (*rule lcs-adds*)
qed

lemma *lcs-plus-right*: *lcs* (*s* + *u*) (*t* + *u*) = (*lcs s t*) + *u*
  using *lcs-plus-left*[*of u s t*] **by** (*simp add*: *ac-simps*)

lemma *adds-gcs*:
  assumes *u adds s* **and** *u adds t*
  shows *u adds* (*gcs s t*)
proof −
  from *assms* **have** *s* + *u adds s* + *t* **and** *t* + *u adds t* + *s*
    by (*simp-all add*: *plus-adds-mono*)

hence *lcs* (*s* + *u*) (*t* + *u*) *adds s* + *t*
by (*auto intro*: *lcs-adds simp add*: *ac-simps*)
hence *u* + (*lcs s t*) *adds s* + *t* **unfolding** *lcs-plus-right* **by** (*simp add*: *ac-simps*)
hence *u adds* (*s* + *t*) − (*lcs s t*) **unfolding** *plus-adds* **..**
thus *?thesis* **unfolding** *gcs-def* **.**
qed

lemma *gcs-unique*:
assumes *g adds s* **and** *g adds t* **and** ∗: $\bigwedge$*u*. *u adds s* ⟹ *u adds t* ⟹ *u adds g*
shows *g* = *gcs s t*
by (*rule adds-antisym*, *rule adds-gcs*, *fact*, *fact*, *rule* ∗, *fact gcs-adds*, *fact gcs-adds-2*)

lemma *gcs-plus-left*: *gcs* (*u* + *s*) (*u* + *t*) = *u* + *gcs s t*
proof −
have *u* + *s* + (*u* + *t*) − (*u* + *lcs s t*) = *u* + *s* + (*u* + *t*) − *u* − *lcs s t* **by** (*simp only*: *diff-diff-add*)
also have ... = *u* + *s* + *t* + (*u* − *u*) − *lcs s t* **by** (*simp add*: *add.left-commute*)
also have ... = *u* + *s* + *t* − *lcs s t* **by** *simp*
also have ... = *u* + (*s* + *t* − *lcs s t*)
using *add-assoc add-commute local.lcs-adds-plus local.minus-plus* **by** *auto*
finally show *?thesis* **unfolding** *gcs-def lcs-plus-left* **.**
qed

lemma *gcs-plus-right*: *gcs* (*s* + *u*) (*t* + *u*) = (*gcs s t*) + *u*
using *gcs-plus-left*[*of u s t*] **by** (*simp add*: *ac-simps*)

lemma *lcs-same* [*simp*]: *lcs s s* = *s*
proof −
have *lcs s s adds s* **by** (*rule lcs-adds*, *simp-all*)
moreover have *s adds lcs s s* **by** (*rule adds-lcs*)
ultimately show *?thesis* **by** (*rule adds-antisym*)
qed

lemma *gcs-same* [*simp*]: *gcs s s* = *s*
proof −
have *gcs s s adds s* **by** (*rule gcs-adds*)
moreover have *s adds gcs s s* **by** (*rule adds-gcs*, *simp-all*)
ultimately show *?thesis* **by** (*rule adds-antisym*)
qed

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*
  unfolding *dickson-grading-def* **using** *assms* **by** *blast*

lemma *dickson-gradingD1*: *dickson-grading d* ⟹ *d* (*s* + *t*) = *max* (*d s*) (*d t*)
  by (*auto simp add*: *dickson-grading-def*)

lemma *dickson-gradingD2*: *dickson-grading d* ⟹ *almost-full-on* (*adds*) {*x*. *d x* ≤ *n*}
  by (*auto simp add*: *dickson-grading-def*)

lemma *dickson-gradingD2'*:
  assumes *dickson-grading* (*d*::'*a*::*comm-monoid-add* ⇒ *nat*)
  shows *wqo-on* (*adds*) {*x*. *d x* ≤ *n*}
proof (*intro wqo-onI transp-onI*)
  fix *x y z* :: '*a*
  assume *x adds y* **and** *y adds z*
  thus *x adds z* **by** (*rule adds-trans*)
next
  from *assms* **show** *almost-full-on* (*adds*) {*x*. *d x* ≤ *n*} **by** (*rule dickson-gradingD2*)
qed

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 −
  from *assms*(*1*) **have** *almost-full-on* (*adds*) {*x*. *d x* ≤ *n*} **by** (*rule dickson-gradingD2*)
  moreover from *assms*(*2*) **have** ⋀*i*. *seq i* ∈ {*x*. *d x* ≤ *n*} **by** *simp*
  ultimately obtain *i j* **where** *i* < *j* **and** *seq i adds seq j* **by** (*rule almost-full-onD*)
  thus *?thesis* **..**
qed

lemma *dickson-grading-adds-imp-le*:
  assumes *dickson-grading d* **and** *s adds t*
  shows *d s* ≤ *d t*
proof −
  from *assms*(*2*) **obtain** *u* **where** *t* = *s* + *u* **..**
  hence *d t* = *max* (*d s*) (*d u*) **by** (*simp only*: *dickson-gradingD1*[*OF assms*(*1*)])
  thus *?thesis* **by** *simp*
qed

lemma *dickson-grading-minus*:
assumes *dickson-grading d* **and** *s adds* (*t*::′*a*::*cancel-ab-semigroup-add*)
shows *d* (*t* − *s*) ≤ *d t*
proof −
from *assms*(*2*) **obtain** *u* **where** *t* = *s* + *u* **..**
hence *t* − *s* = *u* **by** *simp*
from *assms*(*1*) **have** *d t* = *ord-class.max* (*d s*) (*d u*) **unfolding** ‹*t* = *s* + *u*› **by** (*rule dickson-gradingD1*)
thus *?thesis* **by** (*simp add*: ‹*t* − *s* = *u*›)
qed

lemma *dickson-grading-lcs*:
assumes *dickson-grading d*
shows *d* (*lcs s t*) ≤ *max* (*d s*) (*d t*)
proof −
from *assms* **have** *d* (*lcs s t*) ≤ *d* (*s* + *t*) **by** (*rule dickson-grading-adds-imp-le*, *intro lcs-adds-plus*)
thus *?thesis* **by** (*simp only*: *dickson-gradingD1*[*OF assms*])
qed

lemma *dickson-grading-lcs-minus*:
assumes *dickson-grading d*
shows *d* (*lcs s t* − *s*) ≤ *max* (*d s*) (*d t*)
proof −
from *assms* **have** *d* (*lcs s t* − *s*) ≤ *d* (*lcs s t*) **by** (*rule dickson-grading-minus*, *intro adds-lcs*)
also from *assms* **have** ... ≤ *max* (*d s*) (*d t*) **by** (*rule dickson-grading-lcs*)
finally show *?thesis* **.**
qed

lemma *dgrad-set-leI*:
assumes $\bigwedge$*s*. *s* ∈ *S* ⟹ ∃ *t*∈*T*. *d s* ≤ *d t*
shows *dgrad-set-le d S T*
using *assms* **by** (*auto simp*: *dgrad-set-le-def*)

lemma *dgrad-set-leE*:
assumes *dgrad-set-le d S T* **and** *s* ∈ *S*
obtains *t* **where** *t* ∈ *T* **and** *d s* ≤ *d t*
using *assms* **by** (*auto simp*: *dgrad-set-le-def*)

lemma *dgrad-set-exhaust-expl*:
assumes *finite F*
shows *F* ⊆ *dgrad-set d* (*Max* (*d* ‘ *F*))
proof
fix *f*
assume *f* ∈ *F*
hence *d f* ∈ *d* ‘ *F* **by** *simp*
with - **have** *d f* ≤ *Max* (*d* ‘ *F*)
proof (*rule Max-ge*)

from *assms* **show** *finite* (*d* ' *F*) **by** *auto*
qed
hence *dgrad-set d* (*d f*) ⊆ *dgrad-set d* (*Max* (*d* ' *F*)) **by** (*auto simp*: *dgrad-set-def*)
moreover have *f* ∈ *dgrad-set d* (*d f*) **by** (*simp add*: *dgrad-set-def*)
ultimately show *f* ∈ *dgrad-set d* (*Max* (*d* ' *F*)) **..**
qed

lemma *dgrad-set-exhaust*:
assumes *finite F*
obtains *m* **where** *F* ⊆ *dgrad-set d m*
proof
from *assms* **show** *F* ⊆ *dgrad-set d* (*Max* (*d* ' *F*)) **by** (*rule dgrad-set-exhaust-expl*)
qed

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*
unfolding *dgrad-set-le-def*
proof
fix *s*
assume *s* ∈ *S*
with *assms*(*1*) **obtain** *t* **where** *t* ∈ *T* **and** *1*: *d s* ≤ *d t* **by** (*auto simp add*: *dgrad-set-le-def*)
from *assms*(*2*) *this*(*1*) **obtain** *u* **where** *u* ∈ *U* **and** *2*: *d t* ≤ *d u* **by** (*auto simp add*: *dgrad-set-le-def*)
from *this*(*1*) **show** ∃ *u*∈*U*. *d s* ≤ *d u*
proof
from *1 2* **show** *d s* ≤ *d u* **by** (*rule le-trans*)
qed
qed

lemma *dgrad-set-le-Un*: *dgrad-set-le d* (*S* ∪ *T*) *U* ⟷ (*dgrad-set-le d S U* ∧ *dgrad-set-le d T U*)
by (*auto simp add*: *dgrad-set-le-def*)

lemma *dgrad-set-le-subset*:
assumes *S* ⊆ *T*
shows *dgrad-set-le d S T*
unfolding *dgrad-set-le-def* **using** *assms* **by** *blast*

lemma *dgrad-set-le-refl*: *dgrad-set-le d S S*
by (*rule dgrad-set-le-subset*, *fact subset-refl*)

lemma *dgrad-set-le-dgrad-set*:
assumes *dgrad-set-le d F G* **and** *G* ⊆ *dgrad-set d m*
shows *F* ⊆ *dgrad-set d m*
proof
fix *f*
assume *f* ∈ *F*

with *assms*(*1*) **obtain** *g* **where** *g* ∈ *G* **and** ∗: *d f* ≤ *d g* **by** (*auto simp add*: *dgrad-set-le-def*)
from *assms*(*2*) *this*(*1*) **have** *g* ∈ *dgrad-set d m* **..**
hence *d g* ≤ *m* **by** (*simp add*: *dgrad-set-def*)
with ∗ **have** *d f* ≤ *m* **by** (*rule le-trans*)
thus *f* ∈ *dgrad-set d m* **by** (*simp add*: *dgrad-set-def*)
qed

lemma *dgrad-set-dgrad*: *p* ∈ *dgrad-set d* (*d p*)
by (*simp add*: *dgrad-set-def*)

lemma *dgrad-setI* [*intro*]:
assumes *d t* ≤ *m*
shows *t* ∈ *dgrad-set d m*
using *assms* **by** (*auto simp*: *dgrad-set-def*)

lemma *dgrad-setD*:
assumes *t* ∈ *dgrad-set d m*
shows *d t* ≤ *m*
using *assms* **by** (*simp add*: *dgrad-set-def*)

lemma *dgrad-set-zero* [*simp*]: *dgrad-set* (λ-. *0*) *m* = *UNIV*
by *auto*

lemma *subset-dgrad-set-zero*: *F* ⊆ *dgrad-set* (λ-. *0*) *m*
by *simp*

lemma *dgrad-set-subset*:
assumes *m* ≤ *n*
shows *dgrad-set d m* ⊆ *dgrad-set d n*
using *assms* **by** (*auto simp*: *dgrad-set-def*)

lemma *dgrad-set-closed-plus*:
assumes *dickson-grading d* **and** *s* ∈ *dgrad-set d m* **and** *t* ∈ *dgrad-set d m*
shows *s* + *t* ∈ *dgrad-set d m*
proof −
from *assms*(*1*) **have** *d* (*s* + *t*) = *ord-class.max* (*d s*) (*d t*) **by** (*rule dickson-gradingD1*)
also from *assms*(*2*, *3*) **have** ... ≤ *m* **by** (*simp add*: *dgrad-set-def*)
finally show *?thesis* **by** (*simp add*: *dgrad-set-def*)
qed

lemma *dgrad-set-closed-minus*:
assumes *dickson-grading d* **and** *s* ∈ *dgrad-set d m* **and** *t adds* (*s*::*'a*::*cancel-ab-semigroup-add*)
shows *s* − *t* ∈ *dgrad-set d m*
proof −
from *assms*(*1*, *3*) **have** *d* (*s* − *t*) ≤ *d s* **by** (*rule dickson-grading-minus*)
also from *assms*(*2*) **have** ... ≤ *m* **by** (*simp add*: *dgrad-set-def*)
finally show *?thesis* **by** (*simp add*: *dgrad-set-def*)

qed

lemma *dgrad-set-closed-lcs*:
  assumes *dickson-grading d* **and** *s* ∈ *dgrad-set d m* **and** *t* ∈ *dgrad-set d m*
  shows *lcs s t* ∈ *dgrad-set d m*
proof −
  from *assms*(*1*) **have** *d* (*lcs s t*) ≤ *ord-class.max* (*d s*) (*d t*) **by** (*rule dickson-grading-lcs*)
  also from *assms*(*2*, *3*) **have** ... ≤ *m* **by** (*simp add*: *dgrad-set-def*)
  finally show *?thesis* **by** (*simp add*: *dgrad-set-def*)
qed

lemma *dickson-gradingD-dgrad-set*: *dickson-grading d* ⟹ *almost-full-on* (*adds*) (*dgrad-set d m*)
  by (*auto dest*: *dickson-gradingD2 simp*: *dgrad-set-def*)

lemma *ex-finite-adds*:
  assumes *dickson-grading d* **and** *S* ⊆ *dgrad-set d m*
  obtains *T* **where** *finite T* **and** *T* ⊆ *S* **and** ⋀*s*. *s* ∈ *S* ⟹ (∃ *t*∈*T*. *t adds* (*s*::*'a*::*cancel-comm-monoid-add*))
proof −
  have *reflp* ((*adds*)::*'a* ⇒ -) **by** (*simp add*: *reflp-def*)
  moreover from *assms*(*2*) **have** *almost-full-on* (*adds*) *S*
  proof (*rule almost-full-on-subset*)
    from *assms*(*1*) **show** *almost-full-on* (*adds*) (*dgrad-set d m*) **by** (*rule dickson-gradingD-dgrad-set*)
  qed
  ultimately obtain *T* **where** *finite T* **and** *T* ⊆ *S* **and** ⋀*s*. *s* ∈ *S* ⟹ (∃ *t*∈*T*. *t adds s*)
    by (*rule almost-full-on-finite-subsetE*, *blast*)
  thus *?thesis* **..**
qed

class *graded-dickson-powerprod* = *ulcs-powerprod* +
  assumes *ex-dgrad*: ∃ *d*::*'a* ⇒ *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*
  unfolding *dgrad-dummy-def* **using** *ex-dgrad* **by** (*rule someI-ex*)

end

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

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

by (*simp add*: *dickson-grading-def dickson*)

subclass *graded-dickson-powerprod* **by** (*standard*, *rule*, *fact dickson-grading-zero*)

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 −
  have $a + (b - a) = b - a + a$ **by** *simp*
  also have ... $= b$ **by** *simp*
  finally show *?thesis* **.**
qed

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*
  by (*standard*, *simp only*: *le-iff-add*)
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* **..**

subclass *ordered-cancel-comm-monoid-add* **..**

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

using *le-imp-add*[*OF assms*] **by** *auto*

lemma *max-eq-sum*:
obtains *y* **where** *max a b* = *a* + *y*
unfolding *max-def*
proof (*cases a* ≤ *b*)
case *True*
hence *b* = *a* + (*b* − *a*) **by** (*rule le-imp-inv*)
then obtain *c* **where** *eq*: *b* = *a* + *c* **..**
show *?thesis*
proof
from *True* **show** *max a b* = *a* + *c* **unfolding** *max-def eq* **by** *simp*
qed
next
case *False*
show *?thesis*
proof
from *False* **show** *max a b* = *a* + *0* **unfolding** *max-def* **by** *simp*
qed
qed

lemma *min-plus-max*:
shows (*min a b*) + (*max a b*) = *a* + *b*
proof (*cases a* ≤ *b*)
case *True*
thus *?thesis* **unfolding** *min-def max-def* **by** *simp*
next
case *False*
thus *?thesis* **unfolding** *min-def max-def* **by** (*simp add*: *ac-simps*)
qed

end

class *add-linorder-min* = *add-linorder* +
assumes *zero-min*: *0* ≤ *x*
begin

subclass *ninv-comm-monoid-add*
proof
fix *x y*
assume ∗: *x* + *y* = *0*
show *x* = *0*
proof −
from *zero-min*[*of x*] **have** *0* = *x* ∨ *x* > *0* **by** *auto*
thus *?thesis*
proof
assume *x* > *0*
have *0* ≤ *y* **by** (*fact zero-min*)
also have ... = *0* + *y* **by** *simp*

also from ‹$x > 0$› **have** ... $< x + y$ **by** (*rule add-strict-right-mono*)
finally have $0 < x + y$ **.**
hence $x + y \neq 0$ **by** *simp*
from *this* ∗ **show** *?thesis* **..**
qed *simp*
qed
qed

lemma *leq-add-right*:
shows $x \leq x + y$
using *add-left-mono*[*OF zero-min*[*of y*], *of x*] **by** *simp*

lemma *leq-add-left*:
shows $x \leq y + x$
using *add-right-mono*[*OF zero-min*[*of y*], *of x*] **by** *simp*

subclass *canonically-ordered-monoid-add*
by (*standard*, *rule*, *elim le-imp-add*, *elim exE*, *simp add*: *leq-add-right*)

end

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

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

instance by (*standard*, *simp*)

end

instantiation *nat* :: *add-linorder-min*
begin
instance by (*standard*, *simp*)
end

instantiation *nat* :: *add-wellorder*
begin
instance ..
end

context *add-linorder-group*
begin

subclass *add-linorder*
proof (*standard*, *intro exI*)
fix *a b*
show $b = a + (b - a)$ **using** *add-commute local.diff-add-cancel* **by** *auto*
qed

end

instantiation *int* :: *add-linorder-group*
begin
instance ..
end

instantiation *rat* :: *add-linorder-group*
begin
instance ..
end

instantiation *real* :: *add-linorder-group*
begin
instance ..
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*: $0 \preceq t$
  assumes *plus-monotone*: $s \preceq t \Longrightarrow s + u \preceq t + u$
begin

Conceal these relations defined in Equipollence

no-notation *lesspoll* (**infixl** ‹≺› *50*)
no-notation *lepoll* (**infixl** ‹≲› *50*)

abbreviation *ord-conv* (**infixl** ‹⪰› *50*) **where** *ord-conv* ≡ $(\preceq)^{-1-1}$
abbreviation *ord-strict-conv* (**infixl** ‹≻› *50*) **where** *ord-strict-conv* ≡ $(\prec)^{-1-1}$

lemma *ord-canc*:
  assumes $s + u \preceq t + u$
  shows $s \preceq t$
proof (*rule ordered-powerprod-lin.le-cases*[*of s t*], *simp*)
  assume $t \preceq s$
  from *assms plus-monotone*[*OF this*, *of u*] **have** $t + u = s + u$
    using *ordered-powerprod-lin.order.eq-iff* **by** *simp*
  hence $t = s$ **by** *simp*
  thus $s \preceq t$ **by** *simp*
qed

lemma *ord-adds*:
  assumes *s adds t*
  shows $s \preceq t$
proof −
  from *assms* **have** $\exists\, u.\ t = s + u$ **unfolding** *adds-def* **by** *simp*

then obtain *k* **where** $t = s + k$ **..**
thus *?thesis* **using** *plus-monotone*[*OF zero-min*[*of k*], *of s*] **by** (*simp add*: *ac-simps*)
qed

lemma *ord-canc-left*:
assumes $u + s \preceq u + t$
shows $s \preceq t$
using *assms* **unfolding** *add.commute*[*of u*] **by** (*rule ord-canc*)

lemma *ord-strict-canc*:
assumes $s + u \prec t + u$
shows $s \prec t$
using *assms ord-canc*[*of s u t*] *add-right-cancel*[*of s u t*]
ordered-powerprod-lin.le-imp-less-or-eq ordered-powerprod-lin.order.strict-implies-order
by *blast*

lemma *ord-strict-canc-left*:
assumes $u + s \prec u + t$
shows $s \prec t$
using *assms* **unfolding** *add.commute*[*of u*] **by** (*rule ord-strict-canc*)

lemma *plus-monotone-left*:
assumes $s \preceq t$
shows $u + s \preceq u + t$
using *assms*
by (*simp add*: *add.commute*, *rule plus-monotone*)

lemma *plus-monotone-strict*:
assumes $s \prec t$
shows $s + u \prec t + u$
using *assms*
by (*simp add*: *ordered-powerprod-lin.order.strict-iff-order plus-monotone*)

lemma *plus-monotone-strict-left*:
assumes $s \prec t$
shows $u + s \prec u + t$
using *assms*
by (*simp add*: *ordered-powerprod-lin.order.strict-iff-order plus-monotone-left*)

end

locale *gd-powerprod* =
ordered-powerprod ord ord-strict
for *ord*::*'a* ⇒ *'a*::*graded-dickson-powerprod* ⇒ *bool* (**infixl** ‹⪯› *50*)
and *ord-strict* (**infixl** ‹≺› *50*)
begin

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

definition *dickson-less* :: $('a \Rightarrow nat) \Rightarrow nat \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
  where *dickson-less d m s t* $\longleftrightarrow (d\ s \leq m \wedge d\ t \leq m \wedge s \prec t)$

lemma *dickson-leI*:
  assumes $d\ s \leq m$ **and** $d\ t \leq m$ **and** $s \preceq t$
  shows *dickson-le d m s t*
  using *assms* **by** (*simp add*: *dickson-le-def*)

lemma *dickson-leD1*:
  assumes *dickson-le d m s t*
  shows $d\ s \leq m$
  using *assms* **by** (*simp add*: *dickson-le-def*)

lemma *dickson-leD2*:
  assumes *dickson-le d m s t*
  shows $d\ t \leq m$
  using *assms* **by** (*simp add*: *dickson-le-def*)

lemma *dickson-leD3*:
  assumes *dickson-le d m s t*
  shows $s \preceq t$
  using *assms* **by** (*simp add*: *dickson-le-def*)

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*
  using *assms* **by** (*auto simp add*: *dickson-le-def*)

lemma *dickson-lessI*:
  assumes $d\ s \leq m$ **and** $d\ t \leq m$ **and** $s \prec t$
  shows *dickson-less d m s t*
  using *assms* **by** (*simp add*: *dickson-less-def*)

lemma *dickson-lessD1*:
  assumes *dickson-less d m s t*
  shows $d\ s \leq m$
  using *assms* **by** (*simp add*: *dickson-less-def*)

lemma *dickson-lessD2*:
  assumes *dickson-less d m s t*
  shows $d\ t \leq m$
  using *assms* **by** (*simp add*: *dickson-less-def*)

lemma *dickson-lessD3*:
  assumes *dickson-less d m s t*
  shows $s \prec t$
  using *assms* **by** (*simp add*: *dickson-less-def*)

lemma *dickson-less-irrefl*: ¬ *dickson-less d m t t*
by (*simp add*: *dickson-less-def*)

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*
using *assms* **by** (*auto simp add*: *dickson-less-def*)

lemma *transp-dickson-less*: *transp* (*dickson-less d m*)
by (*rule transpI*, *fact dickson-less-trans*)

lemma *wfp-on-ord-strict*:
assumes *dickson-grading d*
shows *wfp-on* (≺) {*x*. *d x* ≤ *n*}
proof −
let *?A* = {*x*. *d x* ≤ *n*}
have *strict* (⪯) = (≺) **by** (*intro ext*, *simp only*: *ordered-powerprod-lin.less-le-not-le*)
have *qo-on* (*adds*) *?A* **by** (*auto simp*: *qo-on-def reflp-on-def transp-on-def dest*: *adds-trans*)
moreover from *assms* **have** *wqo-on* (*adds*) *?A* **by** (*rule dickson-gradingD2′*)
ultimately have (∀ *Q*. (∀ *x*∈*?A*. ∀ *y*∈*?A*. *x adds y* ⟶ *Q x y*) ∧ *qo-on Q ?A* ⟶ *wfp-on* (*strict Q*) *?A*)
by (*simp only*: *wqo-extensions-wf-conv*)
hence (∀ *x*∈*?A*. ∀ *y*∈*?A*. *x adds y* ⟶ *x* ⪯ *y*) ∧ *qo-on* (⪯) *?A* ⟶ *wfp-on* (*strict* (⪯)) *?A* **..**
thus *?thesis* **unfolding** ‹*strict* (⪯) = (≺)›
proof
show (∀ *x*∈*?A*. ∀ *y*∈*?A*. *x adds y* ⟶ *x* ⪯ *y*) ∧ *qo-on* (⪯) *?A*
proof (*intro conjI ballI impI ord-adds*)
show *qo-on* (⪯) *?A* **by** (*auto simp*: *qo-on-def reflp-on-def transp-on-def*)
qed
qed
qed

lemma *wf-dickson-less*:
assumes *dickson-grading d*
shows *wfP* (*dickson-less d m*)
proof (*rule wfP-chain*)
show ¬ (∃ *seq*. ∀ *i*. *dickson-less d m* (*seq* (*Suc i*)) (*seq i*))
proof
assume ∃ *seq*. ∀ *i*. *dickson-less d m* (*seq* (*Suc i*)) (*seq i*)
then obtain *seq*::*nat* ⇒ *′a* **where** ∀ *i*. *dickson-less d m* (*seq* (*Suc i*)) (*seq i*) **..**
hence ∗: ⋀*i*. *dickson-less d m* (*seq* (*Suc i*)) (*seq i*) **..**
with *transp-dickson-less* **have** *seq-decr*: ⋀*i j*. *i* < *j* ⟹ *dickson-less d m* (*seq j*) (*seq i*)
by (*rule transp-sequence*)

from *assms* **obtain** *i j* **where** *i* < *j* **and** *i-adds-j*: *seq i adds seq j*
proof (*rule dickson-gradingE*)

fix *i*
from ∗ **show** *d* (*seq i*) ≤ *m* **by** (*rule dickson-lessD2*)
qed
from ‹*i* < *j*› **have** *dickson-less d m* (*seq j*) (*seq i*) **by** (*rule seq-decr*)
hence *seq j* ≺ *seq i* **by** (*rule dickson-lessD3*)
moreover from *i-adds-j* **have** *seq i* ⪯ *seq j* **by** (*rule ord-adds*)
ultimately show *False* **by** *simp*
qed
qed

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* **by** *standard*

lemma *wf-ord-strict*: *wfP* (≺)
proof (*rule wfP-chain*)
show ¬ (∃ *seq*. ∀ *i*. *seq* (*Suc i*) ≺ *seq i*)
proof
assume ∃ *seq*. ∀ *i*. *seq* (*Suc i*) ≺ *seq i*
then obtain *seq*::*nat* ⇒ '*a* **where** ∀ *i*. *seq* (*Suc i*) ≺ *seq i* **..**
hence ⋀*i*. *seq* (*Suc i*) ≺ *seq i* **..**
with *ordered-powerprod-lin.transp-on-less* **have** *seq-decr*: ⋀*i j*. *i* < *j* ⟹ (*seq j*) ≺ (*seq i*)
by (*rule transp-sequence*)

from *dickson* **obtain** *i j*::*nat* **where** *i* < *j* **and** *i-adds-j*: *seq i adds seq j*
by (*auto elim*!: *almost-full-onD*)
from *seq-decr*[*OF* ‹*i* < *j*›] **have** *seq j* ⪯ *seq i* ∧ *seq j* ≠ *seq i* **by** *auto*
hence *seq j* ⪯ *seq i* **and** *seq j* ≠ *seq i* **by** *simp-all*
from ‹*seq j* ≠ *seq i*› ‹*seq j* ⪯ *seq i*› *ord-adds*[*OF i-adds-j*]
ordered-powerprod-lin.order.eq-iff [*of seq j seq i*]
show *False* **by** *simp*
qed
qed

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 −
from *fin-A fin-B* **have** *finite* ({*x*. *f x* ≠ *0*} ∪ {*x*. *g x* ≠ *0*}) **by** (*intro finite-UnI*)
hence *finite-union*: *finite* {*x*. (*f x* ≠ *0*) ∨ (*g x* ≠ *0*)} **by** (*simp only*: *Collect-disj-eq*)
have {*x*. *h x* (*f x*) (*g x*) ≠ *0*} ⊆ {*x*. (*f x* ≠ *0*) ∨ (*g x* ≠ *0*)}
proof (*intro Collect-mono*, *rule*)
fix *x*::′*a*
assume *h-not-zero*: *h x* (*f x*) (*g x*) ≠ *0*
have *f x* = *0* ⟹ *g x* ≠ *0*
proof
assume *f x* = *0 g x* = *0*
thus *False* **using** *h-not-zero* ‹*h x 0 0* = *0*› **by** *simp*
qed
thus *f x* ≠ *0* ∨ *g x* ≠ *0* **by** *auto*
qed
from *finite-subset*[*OF this*] *finite-union* **show** *finite* {*x*. *h x* (*f x*) (*g x*) ≠ *0*} **.**
qed

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*}
using *assms* **by** (*rule finite-neq-0*)

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 −
from *fin-A* **and** *fin-B* **have** *finite* ({*x*. *h x* (*f x*) (*g x*) ≠ *0*} ∪ {*x*. *f x* ≠ *0*}) **by** (*intro finite-UnI*)
hence *finite-union*: *finite* {*x*. (*h x* (*f x*) (*g x*) ≠ *0*) ∨ *f x* ≠ *0*} **by** (*simp only*: *Collect-disj-eq*)
have {*x*. *g x* ≠ *0*} ⊆ {*x*. (*h x* (*f x*) (*g x*) ≠ *0*) ∨ *f x* ≠ *0*}
by (*intro Collect-mono*, *rule*, *rule disjCI*, *simp add*: *assms*(*3*))
from *this finite-union* **show** *finite* {*x*. *g x* ≠ *0*} **by** (*rule finite-subset*)
qed

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*}
using *assms* **by** (*rule finite-neq-0-inv*)

6.7.1 ′*a* ⇒ ′*b* belongs to class *comm-powerprod*

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

by *standard*

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

instance *fun* :: (*type*, *ninv-comm-monoid-add*) *ninv-comm-monoid-add*
by (*standard*, *simp only*: *plus-fun-def zero-fun-def fun-eq-iff*, *intro allI*, *rule plus-eq-zero*, *auto*)

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 $s \le t$
shows *s adds* (t::$'a \Rightarrow 'b$)
proof (*rule addsI*, *rule*)
fix *x*
from *assms* **have** $s\ x \le t\ x$ **unfolding** *le-fun-def* **..**
hence $t\ x = s\ x + (t\ x - s\ x)$ **by** (*rule le-imp-inv*)
thus $t\ x = (s + (t - s))\ x$ **by** *simp*
qed

lemma *adds-fun-iff*: *f adds* (g::$'a \Rightarrow 'b$) $\longleftrightarrow$ ($\forall x.$ *f x adds g x*)
unfolding *adds-def plus-fun-def* **by** *metis*

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

lemma *adds-lcs-fun*:
shows *s adds* (*lcs s* (t::$'a \Rightarrow 'b$))
by (*rule adds-funI*, *simp only*: *le-fun-def lcs-fun-def*, *auto simp*: *max-def*)

lemma *lcs-comm-fun*: *lcs s t* = *lcs t* (s::$'a \Rightarrow 'b$)
unfolding *lcs-fun-def*
by (*auto simp*: *max-def intro*!: *ext*)

lemma *lcs-adds-fun*:
assumes *s adds u* **and** *t adds* (u::$'a \Rightarrow 'b$)
shows (*lcs s t*) *adds u*
using *assms* **unfolding** *lcs-fun-def adds-fun-iff'*
proof −
assume *a1*: $\forall x.\ \exists y.\ u\ x = s\ x + y$ **and** *a2*: $\forall x.\ \exists y.\ u\ x = t\ x + y$
show $\forall x.\ \exists y.\ u\ x = max\ (s\ x)\ (t\ x) + y$
proof
fix *x*
from *a1* **have** *b1*: $\exists y.\ u\ x = s\ x + y$ **..**

from *a2* **have** *b2*: ∃ *y*. *u x* = *t x* + *y* **..**
show ∃ *y*. *u x* = *max* (*s x*) (*t x*) + *y* **unfolding** *max-def*
by (*split if-split*, *intro conjI impI*, *rule b2*, *rule b1*)
qed
qed

instance
apply *standard*
subgoal by (*rule adds-lcs-fun*)
subgoal by (*rule lcs-adds-fun*)
subgoal by (*rule lcs-comm-fun*)
done

end

lemma *leq-lcs-fun-1*: *s* ≤ (*lcs s* (*t*::′*a* ⇒ ′*b*::*add-linorder*))
by (*simp add*: *lcs-fun-def le-fun-def*)

lemma *leq-lcs-fun-2*: *t* ≤ (*lcs s* (*t*::′*a* ⇒ ′*b*::*add-linorder*))
by (*simp add*: *lcs-fun-def le-fun-def*)

lemma *lcs-leq-fun*:
assumes *s* ≤ *u* **and** *t* ≤ (*u*::′*a* ⇒ ′*b*::*add-linorder*)
shows (*lcs s t*) ≤ *u*
using *assms* **by** (*simp add*: *lcs-fun-def le-fun-def*)

lemma *adds-fun*: *s adds t* ⟷ *s* ≤ *t*
for *s t*::′*a* ⇒ ′*b*::*add-linorder-min*
proof
assume *s adds t*
from *this* **obtain** *k* **where** *t* = *s* + *k* **..**
show *s* ≤ *t* **unfolding** ‹*t* = *s* + *k*› *le-fun-def plus-fun-def le-iff-add* **by** (*simp add*: *leq-add-right*)
qed (*rule adds-funI*)

lemma *gcs-fun*: *gcs s* (*t*::′*a* ⇒ (′*b*::*add-linorder*)) = (λ*x*. *min* (*s x*) (*t x*))
proof −
show *?thesis* **unfolding** *gcs-def lcs-fun-def fun-diff-def*
proof (*simp*, *rule*)
fix *x*
have *eq*: *s x* + *t x* = *max* (*s x*) (*t x*) + *min* (*s x*) (*t x*) **by** (*metis add.commute min-def max-def*)
thus *s x* + *t x* − *max* (*s x*) (*t x*) = *min* (*s x*) (*t x*) **by** *simp*
qed
qed

lemma *gcs-leq-fun-1*: (*gcs s* (*t*::′*a* ⇒ ′*b*::*add-linorder*)) ≤ *s*
by (*simp add*: *gcs-fun le-fun-def*)

lemma *gcs-leq-fun-2*: (*gcs s* (*t*::'*a* ⇒ '*b*::*add-linorder*)) ≤ *t*
by (*simp add*: *gcs-fun le-fun-def*)

lemma *leq-gcs-fun*:
assumes *u* ≤ *s* **and** *u* ≤ (*t*::'*a* ⇒ '*b*::*add-linorder*)
shows *u* ≤ (*gcs s t*)
using *assms* **by** (*simp add*: *gcs-fun le-fun-def*)

6.7.4 '*a* ⇒ '*b* belongs to class *ulcs-powerprod*

instance *fun* :: (*type*, *add-linorder-min*) *ulcs-powerprod* **..**

6.7.5 Power-products in a given set of indeterminates

definition *supp-fun*::('*a* ⇒ '*b*::*zero*) ⇒ '*a set* **where** *supp-fun f* = {*x*. *f x* ≠ *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-fun* (*lookup s*)
unfolding *supp-fun-def* **by** (*transfer*, *rule*)

lemma *supp-fun-zero* [*simp*]: *supp-fun 0* = {}
by (*auto simp*: *supp-fun-def*)

lemma *supp-fun-eq-zero-iff*: *supp-fun f* = {} ⟷ *f* = *0*
by (*auto simp*: *supp-fun-def*)

lemma *sub-supp-empty*: *supp-fun s* ⊆ {} ⟷ (*s* = *0*)
by (*auto simp*: *supp-fun-def*)

lemma *except-fun-idI*: *supp-fun f* ∩ *V* = {} ⟹ *except-fun f V* = *f*
by (*auto simp*: *except-fun-def supp-fun-def when-def intro*!: *ext*)

lemma *supp-except-fun*: *supp-fun* (*except-fun s V*) = *supp-fun s* − *V*
by (*auto simp*: *except-fun-def supp-fun-def*)

lemma *supp-fun-plus-subset*: *supp-fun* (*s* + *t*) ⊆ *supp-fun s* ∪ *supp-fun* (*t*::'*a* ⇒ '*b*::*monoid-add*)
unfolding *supp-fun-def* **by** *force*

lemma *fun-eq-zeroI*:
assumes ⋀*x*. *x* ∈ *supp-fun f* ⟹ *f x* = *0*
shows *f* = *0*
proof (*rule*, *simp*)
fix *x*
show *f x* = *0*
proof (*cases x* ∈ *supp-fun f*)
case *True*
then show *?thesis* **by** (*rule assms*)
next

case *False*
then show *?thesis* **by** (*simp add*: *supp-fun-def*)
qed
qed

lemma *except-fun-cong1*:
supp-fun s ∩ ((*V* − *U*) ∪ (*U* − *V*)) ⊆ {} ⟹ *except-fun s V* = *except-fun s U*
by (*auto simp*: *except-fun-def when-def supp-fun-def intro*!: *ext*)

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*
by (*auto simp*: *supp-fun-def except-fun-def adds-fun-iff when-def*)

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*
by (*auto simp*: *supp-fun-def except-fun-def adds-fun-iff when-def*)

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*}
using *assms*
proof (*induct V*)
case *empty*
have *finite* {*0*} **by** *simp*
moreover have *reflp-on* {*0*::′*a* ⇒ ′*b*} (*adds*) **by** (*simp add*: *reflp-on-def*)
ultimately have *almost-full-on* (*adds*) {*0*::′*a* ⇒ ′*b*} **by** (*rule finite-almost-full-on*)
thus *?case* **by** (*simp add*: *supp-fun-eq-zero-iff*)
next
case (*insert v V*)
show *?case*
proof (*rule almost-full-onI*)
fix *seq*::*nat* ⇒ ′*a* ⇒ ′*b*
assume ∀ *i*. *seq i* ∈ {*x*. *supp-fun x* ⊆ *insert v V*}
hence *a*: *supp-fun* (*seq i*) ⊆ *insert v V* **for** *i* **by** *simp*
define *seq*′ **where** *seq*′ = (λ*i*. (*except-fun* (*seq i*) {*v*}, *except-fun* (*seq i*) *V*))
have *almost-full-on* (*adds*) {*x*::′*a* ⇒ ′*b*. *supp-fun x* ⊆ {*v*}}
proof (*rule almost-full-onI*)
fix *f*::*nat* ⇒ ′*a* ⇒ ′*b*
assume ∀ *i*. *f i* ∈ {*x*. *supp-fun x* ⊆ {*v*}}
hence *b*: *supp-fun* (*f i*) ⊆ {*v*} **for** *i* **by** *simp*
let *?f* = λ*i*. *f i v*
have *wfP* ((<)::′*b* ⇒ -) **by** (*simp add*: *wf wfp-def*)
hence ∄*f* :: - ⇒ ′*b*. ∀ *i*. *f* (*Suc i*) < *f i*

unfolding *wf-iff-no-infinite-down-chain*[*to-pred*] **.**
hence ∀ *f*::*nat* ⇒ *'b*. ∃ *i*. *f i* ≤ *f* (*Suc i*)
by (*simp add*: *not-less*)
hence ∃ *i*. *?f i* ≤ *?f* (*Suc i*) **..**
then obtain *i* **where** *?f i* ≤ *?f* (*Suc i*) **..**
have *i* < *Suc i* **by** *simp*
moreover have *f i adds f* (*Suc i*) **unfolding** *adds-fun-iff*
proof
fix *x*
show *f i x adds f* (*Suc i*) *x*
proof (*cases x* = *v*)
case *True*
with ‹*?f i* ≤ *?f* (*Suc i*)› **show** *?thesis* **by** (*simp add*: *adds-def le-iff-add*)
next
case *False*
with *b* **have** *x* ∉ *supp-fun* (*f i*) **and** *x* ∉ *supp-fun* (*f* (*Suc i*)) **by** *blast+*
thus *?thesis* **by** (*simp add*: *supp-fun-def*)
qed
qed
ultimately show *good* (*adds*) *f* **by** (*meson goodI*)
qed
with *insert*(*3*) **have**
almost-full-on (*prod-le* (*adds*) (*adds*)) ({*x*::*'a* ⇒ *'b*. *supp-fun x* ⊆ *V*} × {*x*::*'a* ⇒ *'b*. *supp-fun x* ⊆ {*v*}})
(**is** *almost-full-on ?P ?A*) **by** (*rule almost-full-on-Sigma*)
moreover from *a* **have** *seq' i* ∈ *?A* **for** *i* **by** (*auto simp add*: *seq'-def supp-except-fun*)
ultimately obtain *i j* **where** *i* < *j* **and** *?P* (*seq' i*) (*seq' j*) **by** (*rule almost-full-onD*)
have *seq i adds seq j* **unfolding** *adds-except-fun*[**where** *s*=*seq i* **and** *V*=*V*]
proof
from ‹*?P* (*seq' i*) (*seq' j*)› **show** *except-fun* (*seq i*) *V adds except-fun* (*seq j*) *V*
by (*simp add*: *prod-le-def seq'-def*)
next
from ‹*?P* (*seq' i*) (*seq' j*)› **have** *except-fun* (*seq i*) {*v*} *adds except-fun* (*seq j*) {*v*}
by (*simp add*: *prod-le-def seq'-def*)
moreover have *except-fun* (*seq i*) (− *V*) = *except-fun* (*seq i*) {*v*}
by (*rule except-fun-cong1*; *use a*[*of i*] *insert.hyps*(*2*) **in** *blast*)
moreover have *except-fun* (*seq j*) (− *V*) = *except-fun* (*seq j*) {*v*}
by (*rule except-fun-cong1*; *use a*[*of j*] *insert.hyps*(*2*) **in** *blast*)
ultimately show *except-fun* (*seq i*) (− *V*) *adds except-fun* (*seq j*) (− *V*) **by** *simp*
qed
with ‹*i* < *j*› **show** *good* (*adds*) *seq* **by** (*meson goodI*)
qed
qed

instance *fun* :: (*finite*, *add-wellorder*) *dickson-powerprod*
proof
  have *finite* (*UNIV*::′*a set*) **by** *simp*
  hence *almost-full-on* (*adds*) {*x*::′*a* ⇒ ′*b*. *supp-fun x* ⊆ *UNIV*} **by** (*rule Dickson-fun*)
  thus *almost-full-on* (*adds*) (*UNIV*::(′*a* ⇒ ′*b*) *set*) **by** *simp*
qed

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 $s \neq (t::'a \Rightarrow 'b)$
  obtains *x* **where** $s\ x \neq t\ x$ **and** $\bigwedge y.\ s\ y \neq t\ y \Longrightarrow x \leq y$
proof −
  from *assms ext*[*of s t*] **have** $\exists x.\ s\ x \neq t\ x$ **by** *auto*
  with *exists-least-iff* [*of* $\lambda x.\ s\ x \neq t\ x$]
  obtain *x* **where** *x1*: $s\ x \neq t\ x$ **and** *x2*: $\bigwedge y.\ y < x \Longrightarrow s\ y = t\ y$
    by *auto*
  show *?thesis*
  proof
    from *x1* **show** $s\ x \neq t\ x$ **.**
  next
    fix *y*
    assume $s\ y \neq t\ y$
    with *x2*[*of y*] **have** $\neg\ y < x$ **by** *auto*
    thus $x \leq y$ **by** *simp*
  qed
qed

definition *lex-fun*::(′*a* ⇒ ′*b*) ⇒ (′*a* ⇒ ′*b*::*order*) ⇒ *bool* **where**
  $lex\text{-}fun\ s\ t \equiv (\forall x.\ s\ x \leq t\ x \vee (\exists y{<}x.\ s\ y \neq t\ y))$

definition $lex\text{-}fun\text{-}strict\ s\ t \longleftrightarrow lex\text{-}fun\ s\ t \wedge \neg\ lex\text{-}fun\ t\ s$

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 $lex\text{-}fun\ s\ t = (s = t \vee (\exists x.\ s\ x < t\ x \wedge (\forall y{<}x.\ s\ y = t\ y)))$ (**is** *?L* = *?R*)
proof
  assume *?L*
  show *?R*

```
  proof (cases s = t)
    assume s = t
    thus ?R by simp
  next
    assume s ≠ t
    from neq-fun-alt[OF this] obtain x0
      where x0-neq: s x0 ≠ t x0 and x0-min: ⋀z. s z ≠ t z ⟹ x0 ≤ z by auto
    show ?R
    proof (intro disjI2, rule exI[of - x0], intro conjI)
     from ‹?L› have s x0 ≤ t x0 ∨ (∃ y. y < x0 ∧ s y ≠ t y) unfolding lex-fun-def
..
      thus s x0 < t x0
      proof
        assume s x0 ≤ t x0
        from this x0-neq show ?thesis by simp
      next
        assume ∃ y. y < x0 ∧ s y ≠ t y
        then obtain y where y < x0 and y-neq: s y ≠ t y by auto
        from ‹y < x0› x0-min[OF y-neq] show ?thesis by simp
      qed
    next
      show ∀ y<x0. s y = t y
      proof (rule, rule)
        fix y
        assume y < x0
        hence ¬ x0 ≤ y by simp
        from this x0-min[of y] show s y = t y by auto
      qed
    qed
  qed
next
  assume ?R
  thus ?L
  proof
    assume s = t
    thus ?thesis by (simp add: lex-fun-def)
  next
    assume ∃ x. s x < t x ∧ (∀ y<x. s y = t y)
    then obtain y where y: s y < t y and y-min: ∀ z<y. s z = t z by auto
    show ?thesis unfolding lex-fun-def
    proof
      fix x
      show s x ≤ t x ∨ (∃ y<x. s y ≠ t y)
      proof (cases s x ≤ t x)
        assume s x ≤ t x
        thus ?thesis by simp
      next
        assume x: ¬ s x ≤ t x
        show ?thesis
```

```
        proof (intro disjI2, rule exI[of - y], intro conjI)
          have ¬ x ≤ y
          proof
            assume x ≤ y
            hence x < y ∨ y = x by auto
            thus False
            proof
              assume x < y
              from x y-min[rule-format, OF this] show ?thesis by simp
            next
              assume y = x
              from this x y show ?thesis
                by (auto simp: preorder-class.less-le-not-le)
            qed
          qed
          thus y < x by simp
        next
          from y show s y ≠ t y by simp
        qed
      qed
    qed
  qed
qed

lemma lex-fun-refl: lex-fun s s
unfolding lex-fun-alt by simp

lemma lex-fun-antisym:
  assumes lex-fun s t and lex-fun t s
  shows s = t
proof
  fix x
  from assms(1) have s = t ∨ (∃x. s x < t x ∧ (∀y<x. s y = t y))
    unfolding lex-fun-alt .
  thus s x = t x
  proof
    assume s = t
    thus ?thesis by simp
  next
    assume ∃x. s x < t x ∧ (∀y<x. s y = t y)
     then obtain x0 where x0: s x0 < t x0 and x0-min: ∀y<x0. s y = t y by
auto
    from assms(2) have t = s ∨ (∃x. t x < s x ∧ (∀y<x. t y = s y)) unfolding
lex-fun-alt .
    thus ?thesis
    proof
      assume t = s
      thus ?thesis by simp
    next
```

assume ∃ *x*. *t x* < *s x* ∧ (∀ *y*<*x*. *t y* = *s y*)
then obtain *x1* **where** *x1*: *t x1* < *s x1* **and** *x1-min*: ∀ *y*<*x1*. *t y* = *s y* **by** *auto*
have *x0* < *x1* ∨ *x1* < *x0* ∨ *x1* = *x0* **using** *local.antisym-conv3* **by** *auto*
show *?thesis*
proof (*rule linorder-cases*[*of x0 x1*])
assume *x1* < *x0*
from *x0-min*[*rule-format*, *OF this*] *x1* **show** *?thesis* **by** *simp*
next
assume *x0* = *x1*
from *this x0 x1* **show** *?thesis* **by** *simp*
next
assume *x0* < *x1*
from *x1-min*[*rule-format*, *OF this*] *x0* **show** *?thesis* **by** *simp*
qed
qed
qed
qed

lemma *lex-fun-trans*:
assumes *lex-fun s t* **and** *lex-fun t u*
shows *lex-fun s u*
proof −
from *assms*(*1*) **have** *s* = *t* ∨ (∃ *x*. *s x* < *t x* ∧ (∀ *y*<*x*. *s y* = *t y*)) **unfolding** *lex-fun-alt* **.**
thus *?thesis*
proof
assume *s* = *t*
from *this assms*(*2*) **show** *?thesis* **by** *simp*
next
assume ∃ *x*. *s x* < *t x* ∧ (∀ *y*<*x*. *s y* = *t y*)
then obtain *x0* **where** *x0*: *s x0* < *t x0* **and** *x0-min*: ∀ *y*<*x0*. *s y* = *t y*
by *auto*
from *assms*(*2*) **have** *t* = *u* ∨ (∃ *x*. *t x* < *u x* ∧ (∀ *y*<*x*. *t y* = *u y*)) **unfolding** *lex-fun-alt* **.**
thus *?thesis*
proof
assume *t* = *u*
from *this assms*(*1*) **show** *?thesis* **by** *simp*
next
assume ∃ *x*. *t x* < *u x* ∧ (∀ *y*<*x*. *t y* = *u y*)
then obtain *x1* **where** *x1*: *t x1* < *u x1* **and** *x1-min*: ∀ *y*<*x1*. *t y* = *u y* **by** *auto*
show *?thesis* **unfolding** *lex-fun-alt*
proof (*intro disjI2*)
show ∃ *x*. *s x* < *u x* ∧ (∀ *y*<*x*. *s y* = *u y*)
proof (*rule linorder-cases*[*of x0 x1*])
assume *x1* < *x0*
show *?thesis*

```
          proof (rule exI[of - x1], intro conjI)
           from x0-min[rule-format, OF ‹x1 < x0›] x1 show s x1 < u x1 by simp
          next
            show ∀ y<x1. s y = u y
            proof (intro allI, intro impI)
              fix y
              assume y < x1
              from this ‹x1 < x0› have y < x0 by simp
              from x0-min[rule-format, OF this] x1-min[rule-format, OF ‹y < x1›]
                show s y = u y by simp
            qed
          qed
        next
          assume x0 < x1
          show ?thesis
          proof (rule exI[of - x0], intro conjI)
           from x1-min[rule-format, OF ‹x0 < x1›] x0 show s x0 < u x0 by simp
          next
            show ∀ y<x0. s y = u y
            proof (intro allI, intro impI)
              fix y
              assume y < x0
              from this ‹x0 < x1› have y < x1 by simp
              from x0-min[rule-format, OF ‹y < x0›] x1-min[rule-format, OF this]
                show s y = u y by simp
            qed
          qed
        next
          assume x0 = x1
          show ?thesis
          proof (rule exI[of - x1], intro conjI)
            from ‹x0 = x1› x0 x1 show s x1 < u x1 by simp
          next
            show ∀ y<x1. s y = u y
            proof (intro allI, intro impI)
              fix y
              assume y < x1
              hence y < x0 using ‹x0 = x1› by simp
              from x0-min[rule-format, OF this] x1-min[rule-format, OF ‹y < x1›]
                show s y = u y by simp
            qed
          qed
        qed
      qed
    qed
  qed
qed
```

lemma *lex-fun-lin*: *lex-fun s t* ∨ *lex-fun t s* **for** *s t*::′*a* ⇒ ′*b*::{*ordered-comm-monoid-add*,

linorder}
proof (*intro disjCI*)
  assume ¬ *lex-fun t s*
  hence *a*: ∀ *x*. ¬ (*t x* < *s x*) ∨ (∃ *y*<*x*. *t y* ≠ *s y*) **unfolding** *lex-fun-alt* **by** *auto*
  show *lex-fun s t* **unfolding** *lex-fun-def*
  proof
    fix *x*
    from *a* **have** ¬ (*t x* < *s x*) ∨ (∃ *y*<*x*. *t y* ≠ *s y*) **..**
    thus *s x* ≤ *t x* ∨ (∃ *y*<*x*. *s y* ≠ *t y*) **by** *auto*
  qed
qed

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}
  unfolding *lex-fun-strict-def* **using** *lex-fun-lin*[*of s t*] **by** *auto*

lemma *lex-fun-zero-min*: *lex-fun 0 s* **for** *s*::'*a* ⇒ '*b*::*add-linorder-min*
  by (*simp add*: *lex-fun-def zero-min*)

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*
unfolding *lex-fun-def*
proof
  fix *x*
  from *that* **have** *s x* ≤ *t x* ∨ (∃ *y*<*x*. *s y* ≠ *t y*) **unfolding** *lex-fun-def* **..**
  thus (*s* + *u*) *x* ≤ (*t* + *u*) *x* ∨ (∃ *y*<*x*. (*s* + *u*) *y* ≠ (*t* + *u*) *y*)
  proof
    assume *a1*: *s x* ≤ *t x*
    show *?thesis*
    proof (*intro disjI1*)
      from *a1* **show** (*s* + *u*) *x* ≤ (*t* + *u*) *x* **by** (*auto simp*: *add-right-mono*)
    qed
  next
    assume ∃ *y*<*x*. *s y* ≠ *t y*
    then obtain *y* **where** *y* < *x* **and** *a2*: *s y* ≠ *t y* **by** *auto*
    show *?thesis*
    proof (*intro disjI2*, *rule exI*[*of - y*], *intro conjI*, *fact*)
      from *a2* **show** (*s* + *u*) *y* ≠ (*t* + *u*) *y* **by** (*auto simp*: *add-right-mono*)
    qed
  qed
qed

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*
  by (*auto simp*: *deg-fun-def*)

lemma *deg-fun-eq-0-iff*:
  assumes *finite* (*supp-fun* (*s*::'*a* ⇒ '*b*::*add-linorder-min*))
  shows *deg-fun s* = *0* ⟷ *s* = *0*
proof
  assume *deg-fun s* = *0*
  hence ∗: (∑ *x*∈(*supp-fun s*). *s x*) = *0* **by** (*simp only*: *deg-fun-def*)
  have ∗∗: ⋀*x*. *0* ≤ *s x* **by** (*rule zero-min*)
  from ∗ **have** ⋀*x*. *x* ∈ *supp-fun s* ⟹ *s x* = *0* **by** (*simp only*: *sum-nonneg-eq-0-iff*[*OF assms* ∗∗])
  thus *s* = *0* **by** (*rule fun-eq-zeroI*)
qed *simp*

lemma *deg-fun-superset*:
  fixes *A*::'*a set*
  assumes *supp-fun s* ⊆ *A* **and** *finite A*
  shows *deg-fun s* = (∑ *x*∈*A*. *s x*)
  unfolding *deg-fun-def*
proof (*rule sum.mono-neutral-cong-left*, *fact*, *fact*, *rule*)
  fix *x*
  assume *x* ∈ *A* − *supp-fun s*
  hence *x* ∉ *supp-fun s* **by** *simp*
  thus *s x* = *0* **by** (*simp add*: *supp-fun-def*)
qed *rule*

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 −
  from *assms* **have** *fin*: *finite* (*supp-fun s* ∪ *supp-fun t*) **by** *simp*
  have *deg-fun* (*s* + *t*) = (∑ *x*∈(*supp-fun* (*s* + *t*)). *s x* + *t x*) **by** (*simp add*: *deg-fun-def*)
  also from *fin* **have** ... = (∑ *x*∈(*supp-fun s* ∪ *supp-fun t*). *s x* + *t x*)
  proof (*rule sum.mono-neutral-cong-left*)
    show ∀ *x*∈*supp-fun s* ∪ *supp-fun t* − *supp-fun* (*s* + *t*). *s x* + *t x* = *0*
    proof
      fix *x*
      assume *x* ∈ *supp-fun s* ∪ *supp-fun t* − *supp-fun* (*s* + *t*)
      hence *x* ∉ *supp-fun* (*s* + *t*) **by** *simp*
      thus *s x* + *t x* = *0* **by** (*simp add*: *supp-fun-def*)
    qed
  qed (*rule supp-fun-plus-subset*, *rule*)
  also have ... = (∑ *x*∈(*supp-fun s* ∪ *supp-fun t*). *s x*) + (∑ *x*∈(*supp-fun s* ∪

supp-fun t). *t x*)
  by (*rule sum.distrib*)
 also from *fin* **have** ($\sum$ *x*∈(*supp-fun s* ∪ *supp-fun t*). *s x*) = *deg-fun s* **unfolding** *deg-fun-def*
 proof (*rule sum.mono-neutral-cong-right*)
  show ∀ *x*∈*supp-fun s* ∪ *supp-fun t* − *supp-fun s*. *s x* = *0*
  proof
   fix *x*
   assume *x* ∈ *supp-fun s* ∪ *supp-fun t* − *supp-fun s*
   hence *x* ∉ *supp-fun s* **by** *simp*
   thus *s x* = *0* **by** (*simp add*: *supp-fun-def*)
  qed
 qed *simp-all*
 also from *fin* **have** ($\sum$ *x*∈(*supp-fun s* ∪ *supp-fun t*). *t x*) = *deg-fun t* **unfolding** *deg-fun-def*
 proof (*rule sum.mono-neutral-cong-right*)
 show ∀ *x*∈*supp-fun s* ∪ *supp-fun t* − *supp-fun t*. *t x* = *0*
  proof
   fix *x*
   assume *x* ∈ *supp-fun s* ∪ *supp-fun t* − *supp-fun t*
   hence *x* ∉ *supp-fun t* **by** *simp*
   thus *t x* = *0* **by** (*simp add*: *supp-fun-def*)
  qed
 qed *simp-all*
 finally show *?thesis* **.**
qed

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 −
 let *?A* = *supp-fun s* ∪ *supp-fun t*
 from *assms*(*1*) *assms*(*2*) **have** *1*: *finite ?A* **by** *simp*
 have *s*: *supp-fun s* ⊆ *?A* **and** *t*: *supp-fun t* ⊆ *?A* **by** *simp-all*
 show *?thesis* **unfolding** *deg-fun-superset*[*OF s 1*] *deg-fun-superset*[*OF t 1*]
 proof (*rule sum-mono*)
  fix *i*
  from *assms*(*3*) **show** *s i* ≤ *t i* **unfolding** *le-fun-def* **..**
 qed
qed

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*)

```
using assms
proof (induct rule: finite-induct)
  assume {} ≠ {}
  thus ∃ y∈{}. ∀ z∈{}. y ≤ z by simp
next
  fix a::'a and A::'a set
  assume a ∉ A and IH: A ≠ {} ⟹ ∃ y∈A. (∀ z∈A. y ≤ z)
  show ∃ y∈insert a A. (∀ z∈insert a A. y ≤ z)
  proof (cases A = {})
    case True
    show ?thesis
    proof (rule bexI[of - a], intro ballI)
      fix z
      assume z ∈ insert a A
      from this True have z = a by simp
      thus a ≤ z by simp
    qed (simp)
  next
    case False
    from IH[OF False] obtain y where y ∈ A and y-min: ∀ z∈A. y ≤ z by auto
    from linear[of a y] show ?thesis
    proof
      assume y ≤ a
      show ?thesis
      proof (rule bexI[of - y], intro ballI)
        fix z
        assume z ∈ insert a A
        hence z = a ∨ z ∈ A by simp
        thus y ≤ z
        proof
          assume z = a
          from this ‹y ≤ a› show y ≤ z by simp
        next
          assume z ∈ A
          from y-min[rule-format, OF this] show y ≤ z .
        qed
      next
        from ‹y ∈ A› show y ∈ insert a A by simp
      qed
    next
      assume a ≤ y
      show ?thesis
      proof (rule bexI[of - a], intro ballI)
        fix z
        assume z ∈ insert a A
        hence z = a ∨ z ∈ A by simp
        thus a ≤ z
        proof
          assume z = a
```

        from *this* **show** $a \leq z$ **by** *simp*
      next
        assume $z \in A$
        from *y-min*[*rule-format*, *OF this*] ‹$a \leq y$› **show** $a \leq z$ **by** *simp*
      qed
    qed (*simp*)
  qed
 qed
qed

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*
 using *assms* **unfolding** *dord-fun-def Let-def* **by** *auto*

lemma *dord-fun-refl*:
 assumes *ord s s*
 shows *dord-fun ord s s*
 using *assms* **unfolding** *dord-fun-def* **by** *simp*

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 −
 from *assms*(*3*) **have** *ts*: *deg-fun t* < *deg-fun s* ∨ (*deg-fun t* = *deg-fun s* ∧ *ord t s*)
  unfolding *dord-fun-def Let-def* **.**
 from *assms*(*2*) **have** *st*: *deg-fun s* < *deg-fun t* ∨ (*deg-fun s* = *deg-fun t* ∧ *ord s t*)
  unfolding *dord-fun-def Let-def* **.**
 thus *?thesis*
 proof
  assume *deg-fun s* < *deg-fun t*
  thus *?thesis* **using** *ts* **by** *auto*
 next
  assume *deg-fun s* = *deg-fun t* ∧ *ord s t*
  hence *deg-fun s* = *deg-fun t* **and** *ord s t* **by** *simp-all*
  from ‹*deg-fun s* = *deg-fun t*› *ts* **have** *ord t s* **by** *simp*
  with ‹*ord s t*› **show** *?thesis* **by** (*rule ord-antisym*)
 qed
qed

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 −
from *assms*(*3*) **have** *ts*: *deg-fun t* < *deg-fun u* ∨ (*deg-fun t* = *deg-fun u* ∧ *ord t u*)
unfolding *dord-fun-def Let-def* **.**
from *assms*(*2*) **have** *st*: *deg-fun s* < *deg-fun t* ∨ (*deg-fun s* = *deg-fun t* ∧ *ord s t*)
unfolding *dord-fun-def Let-def* **.**
thus *?thesis*
proof
assume *deg-fun s* < *deg-fun t*
from *this dord-fun-degD*[*OF assms*(*3*)] **have** *deg-fun s* < *deg-fun u* **by** *simp*
thus *?thesis* **by** (*simp add*: *dord-fun-def Let-def*)
next
assume *deg-fun s* = *deg-fun t* ∧ *ord s t*
hence *deg-fun s* = *deg-fun t* **and** *ord s t* **by** *simp-all*
from *ts* **show** *?thesis*
proof
assume *deg-fun t* < *deg-fun u*
hence *deg-fun s* < *deg-fun u* **using** ‹*deg-fun s* = *deg-fun t*› **by** *simp*
thus *?thesis* **by** (*simp add*: *dord-fun-def Let-def*)
next
assume *deg-fun t* = *deg-fun u* ∧ *ord t u*
hence *deg-fun t* = *deg-fun u* **and** *ord t u* **by** *simp-all*
from *ord-trans*[*OF* ‹*ord s t*› ‹*ord t u*›] ‹*deg-fun s* = *deg-fun t*› ‹*deg-fun t* = *deg-fun u*› **show** *?thesis*
by (*simp add*: *dord-fun-def Let-def*)
qed
qed
qed

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 (*intro disjCI*)
assume ¬ *dord-fun ord t s*
hence *deg-fun s* ≤ *deg-fun t* ∧ (*deg-fun t* ≠ *deg-fun s* ∨ ¬ *ord t s*)
unfolding *dord-fun-def Let-def* **by** *auto*
hence *deg-fun s* ≤ *deg-fun t* **and** *dis1*: *deg-fun t* ≠ *deg-fun s* ∨ ¬ *ord t s* **by** *simp-all*
show *dord-fun ord s t* **unfolding** *dord-fun-def Let-def*
proof (*intro disjCI*)
assume ¬ (*deg-fun s* = *deg-fun t* ∧ *ord s t*)
hence *dis2*: *deg-fun s* ≠ *deg-fun t* ∨ ¬ *ord s t* **by** *simp*
show *deg-fun s* < *deg-fun t*
proof (*cases deg-fun s* = *deg-fun t*)

```
      case True
      from True dis1 have ¬ ord t s by simp
      from True dis2 have ¬ ord s t by simp
      from ‹¬ ord s t› ‹¬ ord t s› that show ?thesis by simp
    next
      case False
      from this ‹deg-fun s ≤ deg-fun t› show ?thesis by simp
    qed
  qed
qed

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
  unfolding dord-fun-def Let-def deg-fun-zero
proof (rule disjCI)
  assume ¬ (0 = deg-fun s ∧ ord 0 s)
  hence dis: deg-fun s ≠ 0 ∨ ¬ ord 0 s by simp
  show 0 < deg-fun s
  proof (cases deg-fun s = 0)
    case True
    hence s = 0 using deg-fun-eq-0-iff[OF assms(2)] by auto
    hence ord 0 s using ord-refl by simp
    with True dis show ?thesis by simp
  next
    case False
    thus ?thesis by (auto simp: zero-less-iff-neq-zero)
  qed
qed

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 -
  from assms(5) have deg-fun s < deg-fun t ∨ (deg-fun s = deg-fun t ∧ ord s t)
    unfolding dord-fun-def Let-def .
  thus ?thesis
  proof
    assume deg-fun s < deg-fun t
     hence deg-fun (s + u) < deg-fun (t + u) by (auto simp: deg-fun-plus[OF -
assms(4)] assms(2) assms(3))
    thus ?thesis unfolding dord-fun-def Let-def by simp
  next
    assume deg-fun s = deg-fun t ∧ ord s t
    hence deg-fun s = deg-fun t and ord s t by simp-all
    from ‹deg-fun s = deg-fun t› have deg-fun (s + u) = deg-fun (t + u)
```

by (*auto simp*: *deg-fun-plus*[*OF* - *assms*(*4*)] *assms*(*2*) *assms*(*3*))
from *this ord-monotone*[*OF* ‹*ord s t*›] **show** *?thesis* **unfolding** *dord-fun-def Let-def* **by** *simp*
qed
qed

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*
unfolding *dlex-fun-def* **by** (*rule dord-fun-refl*, *rule lex-fun-refl*)

lemma *dlex-fun-antisym*:
assumes *dlex-fun s t* **and** *dlex-fun t s*
shows *s* = *t*
by (*rule dord-fun-antisym*, *erule lex-fun-antisym*, *assumption*, *simp-all only*: *dlex-fun-def*[*symmetric*], *fact*+)

lemma *dlex-fun-trans*:
assumes *dlex-fun s t* **and** *dlex-fun t u*
shows *dlex-fun s u*
by (*simp only*: *dlex-fun-def*, *rule dord-fun-trans*, *erule lex-fun-trans*, *assumption*, *simp-all only*: *dlex-fun-def*[*symmetric*], *fact*+)

lemma *dlex-fun-lin*: *dlex-fun s t* ∨ *dlex-fun t s*
for *s t*::(′*a* ⇒ ′*b*::{*ordered-comm-monoid-add*, *linorder*})
unfolding *dlex-fun-def* **by** (*rule dord-fun-lin*, *rule lex-fun-lin*)

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*}
unfolding *dlex-fun-strict-def* **using** *dlex-fun-lin* **by** *auto*

lemma *dlex-fun-zero-min*:
fixes *s t*::(′*a* ⇒ ′*b*::*add-linorder-min*)
assumes *finite* (*supp-fun s*)
shows *dlex-fun 0 s*
unfolding *dlex-fun-def* **by** (*rule dord-fun-zero-min*, *rule lex-fun-refl*, *fact*)

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*)
using *lex-fun-plus-monotone*[*of s t u*] *assms* **unfolding** *dlex-fun-def*
by (*rule dord-fun-plus-monotone*)

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*
unfolding *drlex-fun-def* **by** (*rule dord-fun-refl*, *fact lex-fun-refl*)

lemma *drlex-fun-antisym*:
assumes *drlex-fun s t* **and** *drlex-fun t s*
shows *s* = *t*
by (*rule dord-fun-antisym*, *erule lex-fun-antisym*, *assumption*,
simp-all only: *drlex-fun-def*[*symmetric*], *fact*+)

lemma *drlex-fun-trans*:
assumes *drlex-fun s t* **and** *drlex-fun t u*
shows *drlex-fun s u*
by (*simp only*: *drlex-fun-def*, *rule dord-fun-trans*, *erule lex-fun-trans*, *assumption*,
simp-all only: *drlex-fun-def*[*symmetric*], *fact*+)

lemma *drlex-fun-lin*: *drlex-fun s t* ∨ *drlex-fun t s*
for *s t*::(*'a* ⇒ *'b*::{*ordered-comm-monoid-add*, *linorder*})
unfolding *drlex-fun-def* **by** (*rule dord-fun-lin*, *rule lex-fun-lin*)

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*}
unfolding *drlex-fun-strict-def* **using** *drlex-fun-lin* **by** *auto*

lemma *drlex-fun-zero-min*:

fixes *s t*::(*'a* ⇒ *'b*::*add-linorder-min*)
assumes *finite* (*supp-fun s*)
shows *drlex-fun 0 s*
unfolding *drlex-fun-def* **by** (*rule dord-fun-zero-min*, *rule lex-fun-refl*, *fact*)

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*)
using *lex-fun-plus-monotone*[*of t s u*] *assms* **unfolding** *drlex-fun-def*
by (*rule dord-fun-plus-monotone*)

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
fix *P*::*'a* ⇒ *bool* **and** *a*
assume *hyp*: $\bigwedge x.\ (\bigwedge y.\ (y < x) \Longrightarrow P\ y) \Longrightarrow P\ x$
show *P a*
proof (*rule ccontr*)
assume ¬ *P a*
have *finite* {*x*. ¬ *P x*} (**is** *finite ?A*) **by** *simp*
from ‹¬ *P a*› **have** *a* ∈ *?A* **by** *simp*
hence *?A* ≠ {} **by** *auto*
from *ex-min*[*OF* ‹*finite ?A*› *this*] **obtain** *b* **where** *b* ∈ *?A* **and** *b-min*: ∀ *y*∈*?A*. *b* ≤ *y* **by** *auto*
from ‹*b* ∈ *?A*› **have** ¬ *P b* **by** *simp*
with *hyp*[*of b*] **obtain** *y* **where** *y* < *b* **and** ¬ *P y* **by** *auto*
from ‹¬ *P y*› **have** *y* ∈ *?A* **by** *simp*
with *b-min* **have** *b* ≤ *y* **by** *simp*
with ‹*y* < *b*› **show** *False* **by** *simp*
qed
qed

end

6.8 Type *poly-mapping*

lemma *poly-mapping-eq-zeroI*:
assumes *keys s* = {}
shows *s* = (*0*::(*'a*, *'b*::*zero*) *poly-mapping*)
proof (*rule poly-mapping-eqI*, *simp*)
fix *x*

from *assms* **show** *lookup s x = 0* **by** *auto*
qed

lemma *keys-plus-ninv-comm-monoid-add*: *keys* (*s* + *t*) = *keys s* ∪ *keys* (*t*::'*a* $\Rightarrow_0$ '*b*::*ninv-comm-monoid-add*)
proof (*rule*, *fact Poly-Mapping.keys-add*, *rule*)
  fix *x*
  assume *x* ∈ *keys s* ∪ *keys t*
  thus *x* ∈ *keys* (*s* + *t*)
  proof
    assume *x* ∈ *keys s*
    thus *?thesis*
      by (*metis in-keys-iff lookup-add plus-eq-zero*)
  next
    assume *x* ∈ *keys t*
    thus *?thesis*
      by (*metis in-keys-iff lookup-add plus-eq-zero-2*)
  qed
qed

lemma *lookup-zero-fun*: *lookup 0 = 0*
  by (*simp only*: *zero-poly-mapping.rep-eq zero-fun-def*)

lemma *lookup-plus-fun*: *lookup* (*s* + *t*) = *lookup s* + *lookup t*
  by (*simp only*: *plus-poly-mapping.rep-eq plus-fun-def*)

lemma *lookup-uminus-fun*: *lookup* (− *s*) = − *lookup s*
  by (*fact uminus-poly-mapping.rep-eq*)

lemma *lookup-minus-fun*: *lookup* (*s* − *t*) = *lookup s* − *lookup t*
  by (*simp only*: *minus-poly-mapping.rep-eq*, *rule*, *simp only*: *minus-apply*)

lemma *poly-mapping-adds-iff*: *s adds t* ⟷ *lookup s adds lookup t*
  unfolding *adds-def*
proof
  assume ∃ *k*. *t* = *s* + *k*
  then obtain *k* **where** ∗: *t* = *s* + *k* **..**
  show ∃ *k*. *lookup t* = *lookup s* + *k*
  proof
    from ∗ **show** *lookup t* = *lookup s* + *lookup k* **by** (*simp only*: *lookup-plus-fun*)
  qed
next
  assume ∃ *k*. *lookup t* = *lookup s* + *k*
  then obtain *k* **where** ∗: *lookup t* = *lookup s* + *k* **..**
  have ∗∗: *k* ∈ {*f*. *finite* {*x*. *f x* ≠ *0*}}
  proof
    have *finite* {*x*. *lookup t x* ≠ *0*} **by** *transfer*
    hence *finite* {*x*. *lookup s x* + *k x* ≠ *0*} **by** (*simp only*: ∗ *plus-fun-def*)
    moreover have *finite* {*x*. *lookup s x* ≠ *0*} **by** *transfer*

ultimately show *finite* {*x*. *k x* ≠ *0*} **by** (*rule finite-neq-0-inv'*, *simp*)
qed
show ∃ *k*. *t* = *s* + *k*
proof
show *t* = *s* + *Abs-poly-mapping k*
by (*rule poly-mapping-eqI*, *simp add*: ∗ *lookup-add Abs-poly-mapping-inverse*[*OF* ∗∗])
qed
qed

6.8.1 *'a* ⇒₀ *'b* belongs to class *comm-powerprod*

instance *poly-mapping* :: (*type*, *cancel-comm-monoid-add*) *comm-powerprod*
by *standard*

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 (*standard*, *transfer*)
fix *s t*::*'a* ⇒ *'b*
assume (λ*k*. *s k* + *t k*) = (λ-. *0*)
hence *s* + *t* = *0* **by** (*simp only*: *plus-fun-def zero-fun-def*)
hence *s* = *0* **by** (*rule plus-eq-zero*)
thus *s* = (λ-. *0*) **by** (*simp only*: *zero-fun-def*)
qed

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 −
fix *fun1 fun2*::*'a* ⇒ *'b*
assume *finite* {*t*. *fun1 t* ≠ *0*} **and** *finite* {*t*. *fun2 t* ≠ *0*}
from *finite-neq-0'*[*OF this*, *of max*] **show** *finite* {*t*. *max* (*fun1 t*) (*fun2 t*) ≠ *0*}
by (*auto simp*: *max-def*)
qed

lemma *adds-poly-mappingI*:
assumes *lookup s* ≤ *lookup* (*t*::*'a* ⇒₀ *'b*)
shows *s adds t*
unfolding *poly-mapping-adds-iff* **using** *assms* **by** (*rule adds-funI*)

lemma *lookup-lcs-fun*: *lookup* (*lcs s t*) = *lcs* (*lookup s*) (*lookup* (*t*:: *'a* ⇒₀ *'b*))
by (*simp only*: *lcs-poly-mapping.rep-eq lcs-fun-def*)

instance

by (*standard*, *simp-all only*: *poly-mapping-adds-iff lookup-lcs-fun*, *rule adds-lcs*, *elim lcs-adds*,
assumption, *rule poly-mapping-eqI*, *simp only*: *lookup-lcs-fun lcs-comm*)

end

lemma *adds-poly-mapping*: s *adds* $t \longleftrightarrow$ *lookup* $s \leq$ *lookup* t
for s t::$'a \Rightarrow_0 'b$::*add-linorder-min*
by (*simp only*: *poly-mapping-adds-iff adds-fun*)

lemma *lookup-gcs-fun*: *lookup* (*gcs* s (t::$'a \Rightarrow_0$ ($'b$::*add-linorder*))) = *gcs* (*lookup* s) (*lookup* t)
proof
fix x
show *lookup* (*gcs* s t) x = *gcs* (*lookup* s) (*lookup* t) x
by (*simp add*: *gcs-def lookup-minus lookup-add lookup-lcs-fun*)
qed

6.8.4 $'a \Rightarrow_0 'b$ belongs to class *ulcs-powerprod*

instance *poly-mapping* :: (*type*, *add-linorder-min*) *ulcs-powerprod* **..**

6.8.5 Power-products in a given set of indeterminates.

lemma *adds-except*:
s *adds* t = (*except* s V *adds except* t V $\wedge$ *except* s ($-$ V) *adds except* t ($-$ V))
for s t :: $'a \Rightarrow_0 'b$::*add-linorder*
by (*simp add*: *poly-mapping-adds-iff adds-except-fun*[*of lookup s*, **where** V=V] *except.rep-eq*)

lemma *adds-except-singleton*:
s *adds* $t \longleftrightarrow$ (*except* s $\{v\}$ *adds except* t $\{v\}$ $\wedge$ *lookup* s v *adds lookup* t v)
for s t :: $'a \Rightarrow_0 'b$::*add-linorder*
by (*simp add*: *poly-mapping-adds-iff adds-except-fun-singleton*[*of lookup s*, **where** v=v] *except.rep-eq*)

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

context *countable*
begin

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

lemma *inj-elem-index*: *inj elem-index*
unfolding *elem-index-def* **using** *ex-inj* **by** (*rule someI-ex*)

lemma *elem-index-inj*:
assumes *elem-index* x = *elem-index* y
shows $x = y$

using *inj-elem-index assms* **by** (*rule injD*)

lemma *finite-nat-seg*: *finite* {*x*. *elem-index x* < *n*}
proof (*rule finite-imageD*)
have *elem-index* ' {*x*. *elem-index x* < *n*} ⊆ {*0*..<*n*} **by** *auto*
moreover have *finite* ... **..**
ultimately show *finite* (*elem-index* ' {*x*. *elem-index x* < *n*}) **by** (*rule finite-subset*)
next
from *inj-elem-index* **show** *inj-on elem-index* {*x*. *elem-index x* < *n*} **using** *inj-on-subset* **by** *blast*
qed

end

lemma *Dickson-poly-mapping*:
assumes *finite V*
shows *almost-full-on* (*adds*) {*x*::*'a* $\Rightarrow_0$ *'b*::*add-wellorder*. *keys x* ⊆ *V*}
proof (*rule almost-full-onI*)
fix *seq*::*nat* ⇒ *'a* $\Rightarrow_0$ *'b*
assume *a*: ∀ *i*. *seq i* ∈ {*x*::*'a* $\Rightarrow_0$ *'b*. *keys x* ⊆ *V*}
define *seq'* **where** *seq'* = (λ*i*. *lookup* (*seq i*))
from *assms* **have** *almost-full-on* (*adds*) {*x*::*'a* ⇒ *'b*. *supp-fun x* ⊆ *V*} **by** (*rule Dickson-fun*)
moreover from *a* **have** ⋀*i*. *seq' i* ∈ {*x*::*'a* ⇒ *'b*. *supp-fun x* ⊆ *V*}
by (*auto simp*: *seq'-def keys-eq-supp*)
ultimately obtain *i j* **where** *i* < *j* **and** *seq' i adds seq' j* **by** (*rule almost-full-onD*)
from *this*(*2*) **have** *seq i adds seq j* **by** (*simp add*: *seq'-def poly-mapping-adds-iff*)
with ‹*i* < *j*› **show** *good* (*adds*) *seq* **by** (*rule goodI*)
qed

definition *varnum* :: *'x set* ⇒ (*'x*::*countable* $\Rightarrow_0$ *'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 (*cases x* ∈ *X*)
case *True*
thus *?thesis* **..**
next
case *False*
with *assms* **have** *1*: *x* ∈ *keys t* − *X* **by** *simp*
hence *keys t* − *X* ≠ {} **by** *blast*
hence *eq*: *varnum X t* = *Suc* (*Max* (*elem-index* ' (*keys t* − *X*))) **by** (*simp add*: *varnum-def*)
hence *elem-index x* < *varnum X t* **using** *1* **by** (*simp add*: *less-Suc-eq-le*)
thus *?thesis* **..**
qed

lemma *varnum-plus*:
varnum X (*s* + *t*) = *max* (*varnum X s*) (*varnum X* (*t*::′*x*::*countable* $\Rightarrow_0$ ′*b*::*ninv-comm-monoid-add*))
proof (*simp add*: *varnum-def keys-plus-ninv-comm-monoid-add image-Un Un-Diff*
del: *diff-shunt-var*, *intro impI*)
assume *1*: *keys s* − *X* ≠ {} **and** *2*: *keys t* − *X* ≠ {}
have *finite* (*elem-index* ‘ (*keys s* − *X*)) **by** *simp*
moreover from *1* **have** *elem-index* ‘ (*keys s* − *X*) ≠ {} **by** *simp*
moreover have *finite* (*elem-index* ‘ (*keys t* − *X*)) **by** *simp*
moreover from *2* **have** *elem-index* ‘ (*keys t* − *X*) ≠ {} **by** *simp*
ultimately show *Max* (*elem-index* ‘ (*keys s* − *X*) ∪ *elem-index* ‘ (*keys t* − *X*))
=
max (*Max* (*elem-index* ‘ (*keys s* − *X*))) (*Max* (*elem-index* ‘ (*keys*
t − *X*)))
by (*rule Max-Un*)
qed

lemma *dickson-grading-varnum*:
assumes *finite X*
shows *dickson-grading* ((*varnum X*)::(′*x*::*countable* $\Rightarrow_0$ ′*b*::*add-wellorder*) ⇒ *nat*)
using *varnum-plus*
proof (*rule dickson-gradingI*)
fix *m*::*nat*
let *?V* = *X* ∪ {*x*. *elem-index x* < *m*}
have {*t*::′*x* $\Rightarrow_0$ ′*b*. *varnum X t* ≤ *m*} ⊆ {*t*. *keys t* ⊆ *?V*}
proof (*rule*, *simp*, *intro subsetI*, *simp*)
fix *t*::′*x* $\Rightarrow_0$ ′*b* **and** *x*::′*x*
assume *varnum X t* ≤ *m*
assume *x* ∈ *keys t*
thus *x* ∈ *X* ∨ *elem-index x* < *m*
proof (*rule elem-index-less-varnum*)
assume *x* ∈ *X*
thus *?thesis* **..**
next
assume *elem-index x* < *varnum X t*
hence *elem-index x* < *m* **using** ‹*varnum X t* ≤ *m*› **by** (*rule less-le-trans*)
thus *?thesis* **..**
qed
qed
thus *almost-full-on* (*adds*) {*t*::′*x* $\Rightarrow_0$ ′*b*. *varnum X t* ≤ *m*}
proof (*rule almost-full-on-subset*)
from *assms finite-nat-seg* **have** *finite ?V* **by** (*rule finite-UnI*)
thus *almost-full-on* (*adds*) {*t*::′*x* $\Rightarrow_0$ ′*b*. *keys t* ⊆ *?V*} **by** (*rule Dickson-poly-mapping*)
qed
qed

corollary *dickson-grading-varnum-empty*:
dickson-grading ((*varnum* {})::(- $\Rightarrow_0$ -::*add-wellorder*) ⇒ *nat*)
using *finite.emptyI* **by** (*rule dickson-grading-varnum*)

lemma *varnum-le-iff*: *varnum X t* ≤ *n* ⟷ *keys t* ⊆ *X* ∪ {*x*. *elem-index x* < *n*}
  by (*auto simp*: *varnum-def Suc-le-eq*)

lemma *varnum-zero* [*simp*]: *varnum X 0* = *0*
  by (*simp add*: *varnum-def*)

lemma *varnum-empty-eq-zero-iff*: *varnum* {} *t* = *0* ⟷ *t* = *0*
proof
  assume *varnum* {} *t* = *0*
  hence *keys t* = {} **by** (*simp add*: *varnum-def split*: *if-splits*)
  thus *t* = *0* **by** (*rule poly-mapping-eq-zeroI*)
qed *simp*

instance *poly-mapping* :: (*countable*, *add-wellorder*) *graded-dickson-powerprod*
  by *standard* (*rule*, *fact dickson-grading-varnum-empty*)

instance *poly-mapping* :: (*finite*, *add-wellorder*) *dickson-powerprod*
proof
  have *finite* (*UNIV*::′*a set*) **by** *simp*
  hence *almost-full-on* (*adds*) {*x*::′*a* ⇒₀ ′*b*. *keys x* ⊆ *UNIV*} **by** (*rule Dickson-poly-mapping*)
  thus *almost-full-on* (*adds*) (*UNIV*::(′*a* ⇒₀ ′*b*) *set*) **by** *simp*
qed

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*)))
  unfolding *lex-pm-def* **by** (*metis less-eq-poly-mapping.rep-eq less-funE less-funI poly-mapping-eq-iff*)

lemma *lex-pm-refl*: *lex-pm s s*
  by (*simp add*: *lex-pm-def*)

lemma *lex-pm-antisym*: *lex-pm s t* ⟹ *lex-pm t s* ⟹ *s* = *t*
  by (*simp add*: *lex-pm-def*)

lemma *lex-pm-trans*: *lex-pm s t* ⟹ *lex-pm t u* ⟹ *lex-pm s u*
  by (*simp add*: *lex-pm-def*)

lemma *lex-pm-lin*: *lex-pm s t* ∨ *lex-pm t s*

by (*simp add*: *lex-pm-def linear*)

corollary *lex-pm-strict-alt* [*code*]: *lex-pm-strict s t* = (¬ *lex-pm t s*)
by (*auto simp*: *lex-pm-strict-def lex-pm-def*)

lemma *lex-pm-zero-min*: *lex-pm 0 s* **for** *s*::- $\Rightarrow_0$ -::*add-linorder-min*
proof (*rule ccontr*)
assume ¬ *lex-pm 0 s*
hence *lex-pm-strict s 0* **by** (*simp add*: *lex-pm-strict-alt*)
thus *False* **by** (*simp add*: *lex-pm-strict-def less-poly-mapping.rep-eq less-fun-def*)
qed

lemma *lex-pm-plus-monotone*: *lex-pm s t* ⟹ *lex-pm* (*s* + *u*) (*t* + *u*)
for *s t*::- $\Rightarrow_0$ -::{*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*}
by (*simp add*: *lex-pm-def add-right-mono*)

6.8.8 Degree

lift-definition *deg-pm*::(*'a* $\Rightarrow_0$ *'b*::*comm-monoid-add*) ⇒ *'b* **is** *deg-fun* **.**

lemma *deg-pm-zero*[*simp*]: *deg-pm 0* = *0*
by (*simp add*: *deg-pm.rep-eq lookup-zero-fun*)

lemma *deg-pm-eq-0-iff* [*simp*]: *deg-pm s* = *0* ⟷ *s* = *0* **for** *s*::*'a* $\Rightarrow_0$ *'b*::*add-linorder-min*
by (*simp only*: *deg-pm.rep-eq poly-mapping-eq-iff lookup-zero-fun*, *rule deg-fun-eq-0-iff*,
simp add: *keys-eq-supp*[*symmetric*])

lemma *deg-pm-superset*:
assumes *keys s* ⊆ *A* **and** *finite A*
shows *deg-pm s* = (∑ *x*∈*A*. *lookup s x*)
using *assms* **by** (*simp only*: *deg-pm.rep-eq keys-eq-supp*, *elim deg-fun-superset*)

lemma *deg-pm-plus*: *deg-pm* (*s* + *t*) = *deg-pm s* + *deg-pm* (*t*::*'a* $\Rightarrow_0$ *'b*::*comm-monoid-add*)
by (*simp only*: *deg-pm.rep-eq lookup-plus-fun*, *rule deg-fun-plus*, *simp-all add*:
keys-eq-supp[*symmetric*])

lemma *deg-pm-single*: *deg-pm* (*Poly-Mapping.single x k*) = *k*
proof −
have *keys* (*Poly-Mapping.single x k*) ⊆ {*x*} **by** *simp*
moreover have *finite* {*x*} **by** *simp*
ultimately have *deg-pm* (*Poly-Mapping.single x k*) = (∑ *y*∈{*x*}. *lookup* (*Poly-Mapping.single x k*) *y*)
by (*rule deg-pm-superset*)
also have ... = *k* **by** *simp*
finally show *?thesis* **.**
qed

6.8.9 General Degree-Orders

context *linorder*

begin

lift-definition *dord-pm*::(($'a \Rightarrow_0 'b$::*ordered-comm-monoid-add*) ⇒ ($'a \Rightarrow_0 'b$) ⇒ *bool*) ⇒ ($'a \Rightarrow_0 'b$) ⇒ ($'a \Rightarrow_0 'b$) ⇒ *bool*
is *dord-fun* **by** (*metis local.dord-fun-def*)

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*))
by (*intro ext*) (*transfer*, *simp add*: *dord-fun-def Let-def*)

lemma *dord-pm-degD*:
assumes *dord-pm ord s t*
shows *deg-pm s* ≤ *deg-pm t*
using *assms* **by** (*simp only*: *dord-pm.rep-eq deg-pm.rep-eq*, *elim dord-fun-degD*)

lemma *dord-pm-refl*:
assumes *ord s s*
shows *dord-pm ord s s*
using *assms* **by** (*simp only*: *dord-pm.rep-eq*, *intro dord-fun-refl*, *simp add*: *lookup-inverse*)

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*
using *assms*
proof (*simp only*: *dord-pm.rep-eq poly-mapping-eq-iff*)
assume *1*: (*ord s t* ⟹ *ord t s* ⟹ *lookup s* = *lookup t*)
assume *2*: *dord-fun* (*map-fun Abs-poly-mapping id* ∘ *ord* ∘ *Abs-poly-mapping*) (*lookup s*) (*lookup t*)
assume *3*: *dord-fun* (*map-fun Abs-poly-mapping id* ∘ *ord* ∘ *Abs-poly-mapping*) (*lookup t*) (*lookup s*)
from - *2 3* **show** *lookup s* = *lookup t* **by** (*rule dord-fun-antisym*, *simp add*: *lookup-inverse 1*)
qed

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*
using *assms*
proof (*simp only*: *dord-pm.rep-eq poly-mapping-eq-iff*)
assume *1*: (*ord s t* ⟹ *ord t u* ⟹ *ord s u*)
assume *2*: *dord-fun* (*map-fun Abs-poly-mapping id* ∘ *ord* ∘ *Abs-poly-mapping*) (*lookup s*) (*lookup t*)
assume *3*: *dord-fun* (*map-fun Abs-poly-mapping id* ∘ *ord* ∘ *Abs-poly-mapping*) (*lookup t*) (*lookup u*)
from - *2 3* **show** *dord-fun* (*map-fun Abs-poly-mapping id* ∘ *ord* ∘ *Abs-poly-mapping*) (*lookup s*) (*lookup u*)
by (*rule dord-fun-trans*, *simp add*: *lookup-inverse 1*)
qed

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* $\Rightarrow_0$ ′*b*::{*ordered-comm-monoid-add*, *linorder*}
using *that* **by** (*simp only*: *dord-pm.rep-eq*, *intro dord-fun-lin*, *simp add*: *lookup-inverse*)

lemma *dord-pm-zero-min*: *dord-pm ord 0 s*
if *ord-refl*: ⋀*t*. *ord t t*
for *s t*::′*a* $\Rightarrow_0$ ′*b*::*add-linorder-min*
using *that*
by (*simp only*: *dord-pm.rep-eq lookup-zero-fun*, *intro dord-fun-zero-min*,
simp add: *lookup-inverse*, *simp add*: *keys-eq-supp*[*symmetric*])

lemma *dord-pm-plus-monotone*:
fixes *s t u* ::′*a* $\Rightarrow_0$ ′*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*)
using *assms*
by (*simp only*: *dord-pm.rep-eq lookup-plus-fun*, *intro dord-fun-plus-monotone*,
simp add: *lookup-inverse lookup-plus-fun*[*symmetric*],
simp add: *keys-eq-supp*[*symmetric*],
simp add: *keys-eq-supp*[*symmetric*],
simp add: *keys-eq-supp*[*symmetric*],
simp add: *lookup-inverse*)

end

6.8.10 Degree-Lexicographic Term Order

definition *dlex-pm*::(′*a*::*linorder* $\Rightarrow_0$ ′*b*::{*ordered-comm-monoid-add*,*linorder*}) ⇒
(′*a* $\Rightarrow_0$ ′*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*
unfolding *dlex-pm-def* **using** *lex-pm-refl* **by** (*rule dord-pm-refl*)

lemma *dlex-pm-antisym*: *dlex-pm s t* ⟹ *dlex-pm t s* ⟹ *s* = *t*
unfolding *dlex-pm-def* **using** *lex-pm-antisym* **by** (*rule dord-pm-antisym*)

lemma *dlex-pm-trans*: *dlex-pm s t* ⟹ *dlex-pm t u* ⟹ *dlex-pm s u*
unfolding *dlex-pm-def* **using** *lex-pm-trans* **by** (*rule dord-pm-trans*)

lemma *dlex-pm-lin*: *dlex-pm s t* ∨ *dlex-pm t s*
unfolding *dlex-pm-def* **using** *lex-pm-lin* **by** (*rule dord-pm-lin*)

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

unfolding *dlex-pm-strict-def* **using** *dlex-pm-lin* **by** *auto*

lemma *dlex-pm-zero-min*: *dlex-pm 0 s*
for *s t*::(- $\Rightarrow_0$ -::*add-linorder-min*)
unfolding *dlex-pm-def* **using** *lex-pm-refl* **by** (*rule dord-pm-zero-min*)

lemma *dlex-pm-plus-monotone*: *dlex-pm s t* $\Longrightarrow$ *dlex-pm* (*s* + *u*) (*t* + *u*)
for *s t*::- $\Rightarrow_0$ -::{*ordered-ab-semigroup-add-imp-le*, *ordered-cancel-comm-monoid-add*}
unfolding *dlex-pm-def* **using** *lex-pm-plus-monotone* **by** (*rule dord-pm-plus-monotone*)

6.8.11 Degree-Reverse-Lexicographic Term Order

definition *drlex-pm*::(*'a*::*linorder* $\Rightarrow_0$ *'b*::{*ordered-comm-monoid-add*,*linorder*}) $\Rightarrow$ (*'a* $\Rightarrow_0$ *'b*) $\Rightarrow$ *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*
unfolding *drlex-pm-def* **using** *lex-pm-refl* **by** (*rule dord-pm-refl*)

lemma *drlex-pm-antisym*: *drlex-pm s t* $\Longrightarrow$ *drlex-pm t s* $\Longrightarrow$ *s* = *t*
unfolding *drlex-pm-def* **using** *lex-pm-antisym* **by** (*rule dord-pm-antisym*)

lemma *drlex-pm-trans*: *drlex-pm s t* $\Longrightarrow$ *drlex-pm t u* $\Longrightarrow$ *drlex-pm s u*
unfolding *drlex-pm-def* **using** *lex-pm-trans* **by** (*rule dord-pm-trans*)

lemma *drlex-pm-lin*: *drlex-pm s t* ∨ *drlex-pm t s*
unfolding *drlex-pm-def* **using** *lex-pm-lin* **by** (*rule dord-pm-lin*)

corollary *drlex-pm-strict-alt* [*code*]: *drlex-pm-strict s t* = (¬ *drlex-pm t s*)
unfolding *drlex-pm-strict-def* **using** *drlex-pm-lin* **by** *auto*

lemma *drlex-pm-zero-min*: *drlex-pm 0 s*
for *s t*::(- $\Rightarrow_0$ -::*add-linorder-min*)
unfolding *drlex-pm-def* **using** *lex-pm-refl* **by** (*rule dord-pm-zero-min*)

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*}
unfolding *drlex-pm-def* **using** *lex-pm-plus-monotone* **by** (*rule dord-pm-plus-monotone*)

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*
  by *simp*

7.1 Submodules Spanned by Sets of Module-Elements

lemma *span-insertI*:
  assumes *p* ∈ *span B*
  shows *p* ∈ *span* (*insert r B*)
proof −
  have *B* ⊆ *insert r B* **by** *blast*
  hence *span B* ⊆ *span* (*insert r B*) **by** (*rule span-mono*)
  with *assms* **show** *?thesis* **..**
qed

lemma *span-insertD*:
  assumes *p* ∈ *span* (*insert r B*) **and** *r* ∈ *span B*
  shows *p* ∈ *span B*
  using *assms*(*1*)
proof (*induct p rule*: *span-induct-alt*)
  case *base*
  show *0* ∈ *span B* **by** (*fact span-zero*)
next
  case *step*: (*step q b a*)
  from *step*(*1*) **have** *b* = *r* ∨ *b* ∈ *B* **by** *simp*
  thus *q* ∗*s* *b* + *a* ∈ *span B*
  proof
    assume *eq*: *b* = *r*
   from *step*(*2*) *assms*(*2*) **show** *?thesis* **unfolding** *eq* **by** (*intro span-add span-scale*)
  next
    assume *b* ∈ *B*
    hence *b* ∈ *span B* **using** *span-superset* **..**
    with *step*(*2*) **show** *?thesis* **by** (*intro span-add span-scale*)
  qed
qed

lemma *span-insert-idI*:
  assumes *r* ∈ *span B*
  shows *span* (*insert r B*) = *span B*
proof (*intro subset-antisym subsetI*)
  fix *p*
  assume *p* ∈ *span* (*insert r B*)
  from *this assms* **show** *p* ∈ *span B* **by** (*rule span-insertD*)
next
  fix *p*
  assume *p* ∈ *span B*

thus *p* ∈ *span* (*insert r B*) **by** (*rule span-insertI*)
qed

lemma *span-insert-zero*: *span* (*insert 0 B*) = *span B*
using *span-zero* **by** (*rule span-insert-idI*)

lemma *span-Diff-zero*: *span* (*B* − {*0*}) = *span B*
by (*metis span-insert-zero insert-Diff-single*)

lemma *span-insert-subset*:
assumes *span A* ⊆ *span B* **and** *r* ∈ *span B*
shows *span* (*insert r A*) ⊆ *span B*
proof
fix *p*
assume *p* ∈ *span* (*insert r A*)
thus *p* ∈ *span B*
proof (*induct p rule*: *span-induct-alt*)
case *base*
show *?case* **by** (*fact span-zero*)
next
case *step*: (*step q b a*)
show *?case*
proof (*intro span-add span-scale*)
from ‹*b* ∈ *insert r A*› **show** *b* ∈ *span B*
proof
assume *b* = *r*
thus *b* ∈ *span B* **using** *assms*(*2*) **by** *simp*
next
assume *b* ∈ *A*
hence *b* ∈ *span A* **using** *span-superset* **..**
thus *b* ∈ *span B* **using** *assms*(*1*) **..**
qed
qed *fact*
qed
qed

lemma *replace-span*:
assumes *q* ∈ *span B*
shows *span* (*insert q* (*B* − {*p*})) ⊆ *span B*
by (*rule span-insert-subset*, *rule span-mono*, *fact Diff-subset*, *fact*)

lemma *sum-in-spanI*: ($\sum$ *b*∈*B*. *q b* ∗*s b*) ∈ *span B*
by (*auto simp*: *intro*: *span-sum span-scale dest*: *span-base*)

lemma *span-closed-sum-list*: ($\bigwedge$*x*. *x* ∈ *set xs* ⟹ *x* ∈ *span B*) ⟹ *sum-list xs* ∈ *span B*
by (*induct xs*) (*auto intro*: *span-zero span-add*)

lemma *spanE*:

assumes *p* ∈ *span B*
obtains *A q* **where** *finite A* **and** *A* ⊆ *B* **and** *p* = (∑ *b*∈*A*. (*q b*) ∗*s b*)
using *assms* **by** (*auto simp*: *span-explicit*)

lemma *span-finite-subset*:
assumes *p* ∈ *span B*
obtains *A* **where** *finite A* **and** *A* ⊆ *B* **and** *p* ∈ *span A*
proof −
from *assms* **obtain** *A q* **where** *finite A* **and** *A* ⊆ *B* **and** *p*: *p* = (∑ *a*∈*A*. *q a* ∗*s a*)
by (*rule spanE*)
note *this*(*1*, *2*)
moreover have *p* ∈ *span A* **unfolding** *p* **by** (*rule sum-in-spanI*)
ultimately show *?thesis* **..**
qed

lemma *span-finiteE*:
assumes *finite B* **and** *p* ∈ *span B*
obtains *q* **where** *p* = (∑ *b*∈*B*. (*q b*) ∗*s b*)
using *assms* **by** (*auto simp*: *span-finite*)

lemma *span-subset-spanI*:
assumes *A* ⊆ *span B*
shows *span A* ⊆ *span B*
using *assms subspace-span* **by** (*rule span-minimal*)

lemma *span-insert-cong*:
assumes *span A* = *span B*
shows *span* (*insert p A*) = *span* (*insert p B*) (**is** *?l* = *?r*)
proof
have *1*: *span* (*insert p C1*) ⊆ *span* (*insert p C2*) **if** *span C1* = *span C2* **for** *C1 C2*
proof (*rule span-subset-spanI*)
show *insert p C1* ⊆ *span* (*insert p C2*)
proof (*rule insert-subsetI*)
show *p* ∈ *span* (*insert p C2*) **by** (*rule span-base*) *simp*
next
have *C1* ⊆ *span C1* **by** (*rule span-superset*)
also from *that* **have** . . . = *span C2* **.**
also have . . . ⊆ *span* (*insert p C2*) **by** (*rule span-mono*) *blast*
finally show *C1* ⊆ *span* (*insert p C2*) **.**
qed
qed
from *assms* **show** *?l* ⊆ *?r* **by** (*rule 1*)
from *assms*[*symmetric*] **show** *?r* ⊆ *?l* **by** (*rule 1*)
qed

lemma *span-induct′* [*consumes 1*, *case-names base step*]:
assumes *p* ∈ *span B* **and** *P 0*

```
    and ⋀a q p. a ∈ span B ⟹ P a ⟹ p ∈ B ⟹ q ≠ 0 ⟹ P (a + q *s p)
  shows P p
  using assms(1, 1)
proof (induct p rule: span-induct-alt)
  case base
  from assms(2) show ?case .
next
  case (step q b a)
  from step.hyps(1) have b ∈ span B by (rule span-base)
  hence q *s b ∈ span B by (rule span-scale)
  with step.prems have a ∈ span B by (simp only: span-add-eq)
  hence P a by (rule step.hyps)
  show ?case
  proof (cases q = 0)
    case True
    from ‹P a› show ?thesis by (simp add: True)
  next
    case False
    with ‹a ∈ span B› ‹P a› step.hyps(1) have P (a + q *s b) by (rule assms(3))
    thus ?thesis by (simp only: add.commute)
  qed
qed

lemma span-INT-subset: span (⋂ a∈A. f a) ⊆ (⋂ a∈A. span (f a)) (is ?l ⊆ ?r)
proof
  fix p
  assume p ∈ ?l
  show p ∈ ?r
  proof
    fix a
    assume a ∈ A
    from ‹p ∈ ?l› show p ∈ span (f a)
    proof (induct p rule: span-induct')
      case base
      show ?case by (fact span-zero)
    next
      case (step p q b)
      from step(3) ‹a ∈ A› have b ∈ f a ..
      hence b ∈ span (f a) by (rule span-base)
      with step(2) show ?case by (intro span-add span-scale)
    qed
  qed
qed

lemma span-INT: span (⋂ a∈A. span (f a)) = (⋂ a∈A. span (f a)) (is ?l = ?r)
proof
  have ?l ⊆ (⋂ a∈A. span (span (f a))) by (rule span-INT-subset)
  also have ... = ?r by (simp add: span-span)
  finally show ?l ⊆ ?r .
```

qed (*fact span-superset*)

lemma *span-Int-subset*: *span* (*A* ∩ *B*) ⊆ *span A* ∩ *span B*
proof −
  have *span* (*A* ∩ *B*) = *span* (⋂*x*∈{*A*, *B*}. *x*) **by** *simp*
  also have . . . ⊆ (⋂*x*∈{*A*, *B*}. *span x*) **by** (*fact span-INT-subset*)
  also have . . . = *span A* ∩ *span B* **by** *simp*
  finally show *?thesis* **.**
qed

lemma *span-Int*: *span* (*span A* ∩ *span B*) = *span A* ∩ *span B*
proof −
  have *span* (*span A* ∩ *span B*) = *span* (⋂*x*∈{*A*, *B*}. *span x*) **by** *simp*
  also have . . . = (⋂*x*∈{*A*, *B*}. *span x*) **by** (*fact span-INT*)
  also have . . . = *span A* ∩ *span B* **by** *simp*
  finally show *?thesis* **.**
qed

lemma *span-image-scale-eq-image-scale*: *span* ((∗*s*) *q* ‘ *F*) = (∗*s*) *q* ‘ *span F* (**is** *?A* = *?B*)
proof (*intro subset-antisym subsetI*)
  fix *p*
  assume *p* ∈ *?A*
  thus *p* ∈ *?B*
  proof (*induct p rule*: *span-induct′*)
    case *base*
    from *span-zero* **show** *?case* **by** (*rule rev-image-eqI*) *simp*
  next
    case (*step p r a*)
    from *step.hyps*(*2*) **obtain** *p′* **where** *p′* ∈ *span F* **and** *p*: *p* = *q* ∗*s* *p′* **..**
    from *step.hyps*(*3*) **obtain** *a′* **where** *a′* ∈ *F* **and** *a*: *a* = *q* ∗*s* *a′* **..**
    from *this*(*1*) **have** *a′* ∈ *span F* **by** (*rule span-base*)
    hence *r* ∗*s* *a′* ∈ *span F* **by** (*rule span-scale*)
    with ‹*p′* ∈ *span F*› **have** *p′* + *r* ∗*s* *a′* ∈ *span F* **by** (*rule span-add*)
    hence *q* ∗*s* (*p′* + *r* ∗*s* *a′*) ∈ *?B* **by** (*rule imageI*)
    also have *q* ∗*s* (*p′* + *r* ∗*s* *a′*) = *p* + *r* ∗*s* *a* **by** (*simp add*: *a p algebra-simps*)
    finally show *?case* **.**
  qed
next
  fix *p*
  assume *p* ∈ *?B*
  then obtain *p′* **where** *p′* ∈ *span F* **and** *p* = *q* ∗*s* *p′* **..**
  from *this*(*1*) **show** *p* ∈ *?A* **unfolding** ‹*p* = *q* ∗*s* *p′*›
  proof (*induct p′ rule*: *span-induct′*)
    case *base*
    show *?case* **by** (*simp add*: *span-zero*)
  next
    case (*step p r a*)
    from *step.hyps*(*3*) **have** *q* ∗*s* *a* ∈ (∗*s*) *q* ‘ *F* **by** (*rule imageI*)

hence *q ∗s a ∈ ?A* **by** (*rule span-base*)
hence *r ∗s* (*q ∗s a*) *∈ ?A* **by** (*rule span-scale*)
with *step.hyps*(*2*) **have** *q ∗s p + r ∗s* (*q ∗s a*) *∈ ?A* **by** (*rule span-add*)
also have *q ∗s p + r ∗s* (*q ∗s a*) = *q ∗s* (*p + r ∗s a*) **by** (*simp add*: *algebra-simps*)
finally show *?case* **.**
qed
qed

end

8 Ideals over Commutative Rings

lemma *module-times*: *module* (∗)
by (*standard*, *simp-all add*: *algebra-simps*)

interpretation *ideal*: *module times*
by (*fact module-times*)

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

abbreviation *ideal* ≡ *ideal.span*

lemma *ideal-eq-UNIV-iff-contains-one*: *ideal B = UNIV ⟷ 1 ∈ ideal B*
proof
assume ∗: *1 ∈ ideal B*
show *ideal B = UNIV*
proof
show *UNIV ⊆ ideal B*
proof
fix *x*
from ∗ **have** *x ∗ 1 ∈ ideal B* **by** (*rule ideal.span-scale*)
thus *x ∈ ideal B* **by** *simp*
qed
qed *simp*
qed *simp*

lemma *ideal-eq-zero-iff* [*iff*]: *ideal F* = {*0*} ⟷ *F* ⊆ {*0*}
by (*metis empty-subsetI ideal.span-empty ideal.span-eq*)

lemma *ideal-field-cases*:
obtains *ideal B* = {*0*} | *ideal* (*B*::*'a*::*field set*) = *UNIV*
proof (*cases ideal B* = {*0*})
case *True*
thus *?thesis* **..**
next
case *False*
hence ¬ *B* ⊆ {*0*} **by** *simp*
then obtain *b* **where** *b ∈ B* **and** *b ≠ 0* **by** *blast*
from *this*(*1*) **have** *b ∈ ideal B* **by** (*rule ideal.span-base*)

hence *inverse b* ∗ *b* ∈ *ideal B* **by** (*rule ideal.span-scale*)
with ‹*b* ≠ *0*› **have** *ideal B* = *UNIV* **by** (*simp add*: *ideal-eq-UNIV-iff-contains-one*)
thus *?thesis* **..**
qed

corollary *ideal-field-disj*: *ideal B* = {*0*} ∨ *ideal* (*B*::′*a*::*field set*) = *UNIV*
by (*rule ideal-field-cases*) *blast*+

lemma *image-ideal-subset*:
assumes ⋀*x y*. *h* (*x* + *y*) = *h x* + *h y* **and** ⋀*x y*. *h* (*x* ∗ *y*) = *h x* ∗ *h y*
shows *h* ‘ *ideal F* ⊆ *ideal* (*h* ‘ *F*)
proof (*intro subsetI*, *elim imageE*)
fix *g f*
assume *g*: *g* = *h f*
assume *f* ∈ *ideal F*
thus *g* ∈ *ideal* (*h* ‘ *F*) **unfolding** *g*
proof (*induct f rule*: *ideal.span-induct-alt*)
case *base*
have *h 0* = *h* (*0* + *0*) **by** *simp*
also have ... = *h 0* + *h 0* **by** (*simp only*: *assms*(*1*))
finally show *?case* **by** (*simp add*: *ideal.span-zero*)
next
case (*step c f g*)
from *step.hyps*(*1*) **have** *h f* ∈ *ideal* (*h* ‘ *F*)
by (*intro ideal.span-base imageI*)
hence *h c* ∗ *h f* ∈ *ideal* (*h* ‘ *F*) **by** (*rule ideal.span-scale*)
hence *h c* ∗ *h f* + *h g* ∈ *ideal* (*h* ‘ *F*)
using *step.hyps*(*2*) **by** (*rule ideal.span-add*)
thus *?case* **by** (*simp only*: *assms*)
qed
qed

lemma *image-ideal-eq-surj*:
assumes ⋀*x y*. *h* (*x* + *y*) = *h x* + *h y* **and** ⋀*x y*. *h* (*x* ∗ *y*) = *h x* ∗ *h y* **and** *surj h*
shows *h* ‘ *ideal B* = *ideal* (*h* ‘ *B*)
proof
from *assms*(*1*, *2*) **show** *h* ‘ *ideal B* ⊆ *ideal* (*h* ‘ *B*) **by** (*rule image-ideal-subset*)
next
show *ideal* (*h* ‘ *B*) ⊆ *h* ‘ *ideal B*
proof
fix *b*
assume *b* ∈ *ideal* (*h* ‘ *B*)
thus *b* ∈ *h* ‘ *ideal B*
proof (*induct b rule*: *ideal.span-induct-alt*)
case *base*
have *h 0* = *h* (*0* + *0*) **by** *simp*
also have ... = *h 0* + *h 0* **by** (*simp only*: *assms*(*1*))
finally have *0* = *h 0* **by** *simp*

```
      with ideal.span-zero show ?case by (rule rev-image-eqI)
    next
      case (step c b a)
      from assms(3) obtain c' where c: c = h c' by (rule surjE)
      from step.hyps(2) obtain a' where a' ∈ ideal B and a: a = h a' ..
      from step.hyps(1) obtain b' where b' ∈ B and b: b = h b' ..
      from this(1) have b' ∈ ideal B by (rule ideal.span-base)
      hence c' * b' ∈ ideal B by (rule ideal.span-scale)
      hence c' * b' + a' ∈ ideal B using ‹a' ∈ -› by (rule ideal.span-add)
      moreover have c * b + a = h (c' * b' + a')
        by (simp add: c b a assms(1, 2))
      ultimately show ?case by (rule rev-image-eqI)
    qed
  qed
qed

context
  fixes h :: 'a ⇒ 'a::comm-ring-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 ⊆ range h and p ∈ range h and p ∈ ideal B
  obtains q where ⋀b. q b ∈ range h and p = (∑ b∈B. q b * b)
proof −
 from assms(1, 4) obtain q0 where p: p = (∑ b∈B. q0 b * b) by (rule ideal.span-finiteE)
  define q where q = (λb. h (q0 b))
  show ?thesis
  proof
    fix b
    show q b ∈ range h unfolding q-def by (rule rangeI)
  next
    from assms(3) obtain p' where p = h p' ..
    hence p = h p by (simp only: h-idem)
    also from ‹finite B› have ... = (∑ b∈B. q b * h b) unfolding p
    proof (induct B)
      case empty
      have h 0 = h (0 + 0) by simp
      also have ... = h 0 + h 0 by (simp only: h-plus)
      finally show ?case by simp
    next
      case (insert b B)
      thus ?case by (simp add: h-plus h-times q-def)
    qed
    also from refl have ... = (∑ b∈B. q b * b)
    proof (rule sum.cong)
      fix b
```

assume $b \in B$
hence $b \in$ *range* h **using** *assms*(*2*) **..**
then obtain b' **where** $b = h\ b'$ **..**
thus $q\ b * h\ b = q\ b * b$ **by** (*simp only*: *h-idem*)
qed
finally show $p = (\sum b{\in}B.\ q\ b * b)$ **.**
qed
qed

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 −
from *assms*(*3*) **obtain** A **where** *finite* A **and** $A \subseteq B$ **and** $p \in$ *ideal* A
by (*rule ideal.span-finite-subset*)
from *this*(*2*) *assms*(*1*) **have** $A \subseteq$ *range* h **by** (*rule subset-trans*)
with ‹*finite* A› **obtain** q **where** $\bigwedge b.\ q\ b \in$ *range* h **and** $p = (\sum b{\in}A.\ q\ b * b)$
using *assms*(*2*) ‹$p \in$ *ideal* A› **by** (*rule in-idealE-homomorphism-finite*) *blast*
with ‹*finite* A› ‹$A \subseteq B$› **show** *?thesis* **..**
qed

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 −
from *assms*(*1*−*3*) **obtain** $A\ q$ **where** *finite* A **and** $A \subseteq B$ **and** *rl*: $\bigwedge f.\ q\ f \in$ *range* h
and *p*: $p = (\sum f{\in}A.\ q\ f * f)$ **by** (*rule in-idealE-homomorphism*) *blast*
show *?thesis* **unfolding** *p* **using** ‹*finite* A› ‹$A \subseteq B$›
proof (*induct* A)
case *empty*
from *assms*(*4*) **show** *?case* **by** *simp*
next
case (*insert* $a\ A$)
from *insert.hyps*(*1*, *2*) **have** $(\sum f{\in}insert\ a\ A.\ q\ f * f) = q\ a * a + (\sum f{\in}A.\ q\ f * f)$ **by** *simp*
also from *rl* **have** P ...
proof (*rule assms*(*5*))
have $a \in$ *insert* $a\ A$ **by** *simp*
thus $a \in B$ **using** *insert.prems* **..**
next
from *insert.prems* **have** $A \subseteq B$ **by** *simp*
thus $P\ (\sum f{\in}A.\ q\ f * f)$ **by** (*rule insert.hyps*)
next
from *insert.prems* **have** $A \subseteq B$ **by** *simp*
hence $A \subseteq$ *range* h **using** *assms*(*1*) **by** (*rule subset-trans*)

with ‹*finite A*› **show** $(\sum f\in A.\ q\ f * f) \in$ *range h*
proof (*induct A*)
case *empty*
have $h\ 0 = h\ (0 + 0)$ **by** *simp*
also have $\ldots = h\ 0 + h\ 0$ **by** (*simp only*: *h-plus*)
finally have $(\sum f\in\{\}.\ q\ f * f) = h\ 0$ **by** *simp*
thus *?case* **by** (*rule image-eqI*) *simp*
next
case (*insert a A*)
from *insert.prems* **have** $a \in$ *range h* **and** $A \subseteq$ *range h* **by** *simp-all*
from *this*(*1*) **obtain** a' **where** *a*: $a = h\ a'$ **..**
from ‹$q\ a \in$ *range h*› **obtain** q' **where** *q*: $q\ a = h\ q'$ **..**
from ‹$A \subseteq$ -› **have** $(\sum f\in A.\ q\ f * f) \in$ *range h* **by** (*rule insert.hyps*)
then obtain *m* **where** *eq*: $(\sum f\in A.\ q\ f * f) = h\ m$ **..**
from *insert.hyps*(*1*, *2*) **have** $(\sum f\in insert\ a\ A.\ q\ f * f) = q\ a * a + (\sum f\in A.\ q\ f * f)$ **by** *simp*
also have $\ldots = h\ (q' * a' + m)$ **unfolding** *q* **by** (*simp add*: *a eq h-plus h-times*)
also have $\ldots \in$ *range h* **by** (*rule rangeI*)
finally show *?case* **.**
qed
qed
finally show *?case* **.**
qed
qed

lemma *image-ideal-eq-Int*: *h ‘ ideal B* = *ideal* (*h ‘ B*) ∩ *range h*
proof
from *h-plus h-times* **have** *h ‘ ideal B* ⊆ *ideal* (*h ‘ B*) **by** (*rule image-ideal-subset*)
thus *h ‘ ideal B* ⊆ *ideal* (*h ‘ B*) ∩ *range h* **by** *blast*
next
show *ideal* (*h ‘ B*) ∩ *range h* ⊆ *h ‘ ideal B*
proof
fix *b*
assume *b* ∈ *ideal* (*h ‘ B*) ∩ *range h*
hence *b* ∈ *ideal* (*h ‘ B*) **and** *b* ∈ *range h* **by** *simp-all*
have *h ‘ B* ⊆ *range h* **by** *blast*
thus *b* ∈ *h ‘ ideal B* **using** ‹*b* ∈ *range h*› ‹*b* ∈ *ideal* (*h ‘ B*)›
proof (*induct b rule*: *ideal-induct-homomorphism*)
case *0*
have $h\ 0 = h\ (0 + 0)$ **by** *simp*
also have $\ldots = h\ 0 + h\ 0$ **by** (*simp only*: *h-plus*)
finally have $0 = h\ 0$ **by** *simp*
with *ideal.span-zero* **show** *?case* **by** (*rule rev-image-eqI*)
next
case (*plus c b a*)
from *plus.hyps*(*1*) **obtain** c' **where** *c*: $c = h\ c'$ **..**
from *plus.hyps*(*3*) **obtain** a' **where** $a' \in$ *ideal B* **and** *a*: $a = h\ a'$ **..**
from *plus.hyps*(*2*) **obtain** b' **where** $b' \in B$ **and** *b*: $b = h\ b'$ **..**

from *this*(*1*) **have** $b' \in$ *ideal B* **by** (*rule ideal.span-base*)
hence $c' * b' \in$ *ideal B* **by** (*rule ideal.span-scale*)
hence $c' * b' + a' \in$ *ideal B* **using** ‹$a' \in$ -› **by** (*rule ideal.span-add*)
moreover have $c * b + a = h\ (c' * b' + a')$ **by** (*simp add*: *a b c h-plus h-times*)
ultimately show *?case* **by** (*rule rev-image-eqI*)
qed
qed
qed

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** $\neg\ b \Longrightarrow f\ 0 = 0$
using *that* **by** (*auto simp*: *when-def*)

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* (λ*k*. *f k* (*lookup p k*) (*lookup q k*) *when* $k \in$ *keys p* ∪ *keys q*)

lemma *lookup-mapp-2*:
lookup (*mapp-2 f p q*) *k* = (*f k* (*lookup p k*) (*lookup q k*) *when* $k \in$ *keys p* ∪ *keys q*)
proof −
have *lookup* (*Abs-poly-mapping* (λ*k*. *f k* (*lookup p k*) (*lookup q k*) *when* $k \in$ *keys p* ∪ *keys q*)) =
(λ*k*. *f k* (*lookup p k*) (*lookup q k*) *when* $k \in$ *keys p* ∪ *keys q*)
by (*rule Abs-poly-mapping-inverse*, *simp*)
thus *?thesis* **by** (*simp add*: *mapp-2-def*)
qed

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*)
by (*simp add*: *lookup-mapp-2 when-def in-keys-iff assms*)

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 $mapp\text{-}2\ f\ p\ q = mapp\text{-}2\ f'\ p'\ q'$
by (*rule poly-mapping-eqI*, *simp add*: *assms*(*1*, *2*) *lookup-mapp-2*, *rule when-cong*, *fact refl*, *rule assms*(*3*), *blast*)

lemma *keys-mapp-subset*: $keys\ (mapp\text{-}2\ f\ p\ q) \subseteq keys\ p \cup keys\ q$
proof
fix *t*
assume $t \in keys\ (mapp\text{-}2\ f\ p\ q)$
hence $lookup\ (mapp\text{-}2\ f\ p\ q)\ t \neq 0$ **by** (*simp add*: *in-keys-iff*)
thus $t \in keys\ p \cup keys\ q$ **by** (*simp add*: *lookup-mapp-2 when-def split*: *if-split-asm*)
qed

lemma *mapp-2-mapp*: $mapp\text{-}2\ (\lambda t\ a.\ f\ t)\ 0\ p = Poly\text{-}Mapping.mapp\ f\ p$
by (*rule poly-mapping-eqI*, *simp add*: *lookup-mapp lookup-mapp-2*)

9.1 *keys*

lemma *in-keys-plusI1*:
assumes $t \in keys\ p$ **and** $t \notin keys\ q$
shows $t \in keys\ (p + q)$
using *assms* **unfolding** *in-keys-iff lookup-add* **by** *simp*

lemma *in-keys-plusI2*:
assumes $t \in keys\ q$ **and** $t \notin keys\ p$
shows $t \in keys\ (p + q)$
using *assms* **unfolding** *in-keys-iff lookup-add* **by** *simp*

lemma *keys-plus-eqI*:
assumes $keys\ p \cap keys\ q = \{\}$
shows $keys\ (p + q) = (keys\ p \cup keys\ q)$
proof
show $keys\ (p + q) \subseteq keys\ p \cup keys\ q$
by (*simp add*: *Poly-Mapping.keys-add*)
show $keys\ p \cup keys\ q \subseteq keys\ (p + q)$
by (*simp add*: *More-MPoly-Type.keys-add assms*)
qed

lemma *keys-uminus*: $keys\ (-\ p) = keys\ p$
by (*transfer*, *auto*)

lemma *keys-minus*: $keys\ (p - q) \subseteq (keys\ p \cup keys\ q)$
by (*transfer*, *auto*)

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*}
  using *assms* **by** *simp*

lemma *monomial-uminus*:
  shows − *monomial c s* = *monomial* (− *c*) *s*
  by (*transfer*, *rule ext*, *simp add*: *Poly-Mapping.when-def*)

lemma *monomial-inj*:
  assumes *monomial c s* = *monomial* (*d*::′*b*::*zero-neq-one*) *t*
  shows (*c* = *0* ∧ *d* = *0*) ∨ (*c* = *d* ∧ *s* = *t*)
  using *assms* **unfolding** *poly-mapping-eq-iff*
  by (*metis* (*mono-tags*, *opaque-lifting*) *lookup-single-eq lookup-single-not-eq*)

definition *is-monomial* :: (′*a* $\Rightarrow_0$ ′*b*::*zero*) ⇒ *bool*
  where *is-monomial p* ⟷ *card* (*keys p*) = *1*

lemma *monomial-is-monomial*:
  assumes $c \neq 0$
  shows *is-monomial* (*monomial c t*)
  using *keys-single*[*of t c*] *assms* **by** (*simp add*: *is-monomial-def*)

lemma *is-monomial-monomial*:
  assumes *is-monomial p*
  obtains *c t* **where** $c \neq 0$ **and** *p* = *monomial c t*
proof −
  from *assms* **have** *card* (*keys p*) = *1* **unfolding** *is-monomial-def* **.**
  then obtain *t* **where** *sp*: *keys p* = {*t*} **by** (*rule card-1-singletonE*)
  let *?c* = *lookup p t*
  from *sp* **have** $?c \neq 0$ **by** *fastforce*
  show *?thesis*
  proof
    show *p* = *monomial ?c t*
    proof (*intro poly-mapping-keys-eqI*)
      from *sp* **show** *keys p* = *keys* (*monomial ?c t*) **using** ‹$?c \neq 0$› **by** *simp*
    next
      fix *s*
      assume *s* ∈ *keys p*
      with *sp* **have** *s* = *t* **by** *simp*
      show *lookup p s* = *lookup* (*monomial ?c t*) *s* **by** (*simp add*: ‹*s* = *t*›)
    qed
  qed *fact*
qed

lemma *is-monomial-uminus*: *is-monomial* (−*p*) ⟷ *is-monomial p*
  unfolding *is-monomial-def keys-uminus* **..**

lemma *monomial-not-0*:
  assumes *is-monomial p*
  shows $p \neq 0$
  using *assms* **unfolding** *is-monomial-def* **by** *auto*

lemma *keys-subset-singleton-imp-monomial*:
  assumes *keys* $p \subseteq \{t\}$
  shows *monomial* (*lookup p t*) $t = p$
proof (*rule poly-mapping-eqI*, *simp add*: *lookup-single when-def*, *rule*)
  fix *s*
  assume $t \neq s$
  hence $s \notin$ *keys p* **using** *assms* **by** *blast*
  thus *lookup p s* = *0* **by** (*simp add*: *in-keys-iff*)
qed

lemma *monomial-0I*:
  assumes *c* = *0*
  shows *monomial c t* = *0*
  using *assms* **by** *transfer* (*auto*)

lemma *monomial-0D*:
  assumes *monomial c t* = *0*
  shows *c* = *0*
  using *assms* **by** *transfer* (*auto simp*: *fun-eq-iff when-def*; *meson*)

corollary *monomial-0-iff*: *monomial c t* = *0* $\longleftrightarrow$ *c* = *0*
  by (*rule*, *erule monomial-0D*, *erule monomial-0I*)

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 (*induct p rule*: *poly-mapping-except-induct*, *simp*)
  fix *p*::′*a* $\Rightarrow_0$ ′*b* **and** *w*
  assume $p \neq 0$ **and** $w \in$ *keys p*
    and *IH*: *lookup* (*monomial c t* ∗ *except p* {*w*}) *s* =
        (*c* ∗ *lookup* (*except p* {*w*}) (*s* − *t*) *when t adds s*) (**is** - = *?x*)
  have *monomial c t* ∗ *p* = *monomial c t* ∗ (*monomial* (*lookup p w*) *w* + *except p* {*w*})
    by (*simp only*: *plus-except*[*symmetric*])
  also have ... = *monomial c t* ∗ *monomial* (*lookup p w*) *w* + *monomial c t* ∗ *except p* {*w*}
    by (*simp add*: *algebra-simps*)
  also have ... = *monomial* (*c* ∗ *lookup p w*) (*t* + *w*) + *monomial c t* ∗ *except p* {*w*}
    by (*simp only*: *mult-single*)
  finally have *lookup* (*monomial c t* ∗ *p*) *s* = *lookup* (*monomial* (*c* ∗ *lookup p w*) (*t* + *w*)) *s* + *?x*
    by (*simp only*: *lookup-add IH*)
  also have ... = (*lookup* (*monomial* (*c* ∗ *lookup p w*) (*t* + *w*)) *s* +

c ∗ *lookup* (*except p* {*w*}) (*s* − *t*) *when t adds s*)
  by (*rule when-distrib*, *auto simp add*: *lookup-single when-def*)
 also from *refl* **have** ... = (*c* ∗ *lookup p* (*s* − *t*) *when t adds s*)
 proof (*rule when-cong*)
  assume *t adds s*
  then obtain *u* **where** *u*: *s* = *t* + *u* **..**
  show *lookup* (*monomial* (*c* ∗ *lookup p w*) (*t* + *w*)) *s* + *c* ∗ *lookup* (*except p* {*w*}) (*s* − *t*) =
    c ∗ *lookup p* (*s* − *t*)
   by (*simp add*: *u*, *cases u* = *w*, *simp-all add*: *lookup-except lookup-single add.commute*)
 qed
 finally show *lookup* (*monomial c t* ∗ *p*) *s* = (*c* ∗ *lookup p* (*s* − *t*) *when t adds s*) **.**
qed

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 (*induct p rule*: *poly-mapping-except-induct*, *simp*)
 fix *p*::′*a* $\Rightarrow_0$ ′*b* **and** *w*
 assume *p* ≠ *0* **and** *w* ∈ *keys p*
  and *IH*: *lookup* (*except p* {*w*} ∗ *monomial c t*) *s* =
    ((*lookup* (*except p* {*w*}) (*s* − *t*)) ∗ *c when t adds s*)
   (**is** - = *?x*)
 have *p* ∗ *monomial c t* = (*monomial* (*lookup p w*) *w* + *except p* {*w*}) ∗ *monomial c t*
  by (*simp only*: *plus-except*[*symmetric*])
 also have ... = *monomial* (*lookup p w*) *w* ∗ *monomial c t* + *except p* {*w*} ∗ *monomial c t*
  by (*simp add*: *algebra-simps*)
 also have ... = *monomial* (*lookup p w* ∗ *c*) (*w* + *t*) + *except p* {*w*} ∗ *monomial c t*
  by (*simp only*: *mult-single*)
 finally have *lookup* (*p* ∗ *monomial c t*) *s* = *lookup* (*monomial* (*lookup p w* ∗ *c*) (*w* + *t*)) *s* + *?x*
  by (*simp only*: *lookup-add IH*)
 also have ... = (*lookup* (*monomial* (*lookup p w* ∗ *c*) (*w* + *t*)) *s* +
    lookup (*except p* {*w*}) (*s* − *t*) ∗ *c when t adds s*)
  by (*rule when-distrib*, *auto simp add*: *lookup-single when-def*)
 also from *refl* **have** ... = (*lookup p* (*s* − *t*) ∗ *c when t adds s*)
 proof (*rule when-cong*)
  assume *t adds s*
  then obtain *u* **where** *u*: *s* = *t* + *u* **..**
  show *lookup* (*monomial* (*lookup p w* ∗ *c*) (*w* + *t*)) *s* + *lookup* (*except p* {*w*}) (*s* − *t*) ∗ *c* =
    lookup p (*s* − *t*) ∗ *c*
   by (*simp add*: *u*, *cases u* = *w*, *simp-all add*: *lookup-except lookup-single add.commute*)

qed
finally show *lookup* (*p* ∗ *monomial c t*) *s* = (*lookup p* (*s* − *t*) ∗ *c when t adds s*) **.**
qed

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 −
from *assms* **have** *term-of-pair* (*pair-of-term u*) = *term-of-pair* (*pair-of-term v*) **by** (*simp only*:)
thus *?thesis* **by** (*simp add*: *term-simps*)
qed

corollary *pair-of-term-inj*: *inj pair-of-term*
using *pair-of-term-injective* **by** (*rule injI*)

lemma *term-of-pair-injective*:
assumes *term-of-pair p* = *term-of-pair q*
shows *p* = *q*
proof −
from *assms* **have** *pair-of-term* (*term-of-pair p*) = *pair-of-term* (*term-of-pair q*) **by** (*simp only*:)
thus *?thesis* **by** (*simp add*: *term-simps*)
qed

corollary *term-of-pair-inj*: *inj term-of-pair*
using *term-of-pair-injective* **by** (*rule injI*)

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*
by (*simp add*: *pp-of-term-def component-of-term-def term-pair*)

lemma *pp-of-term-of-pair* [*term-simps*]: *pp-of-term* (*term-of-pair* (*t*, *k*)) = *t*
by (*simp add*: *pp-of-term-def pair-term*)

lemma *component-of-term-of-pair* [*term-simps*]: *component-of-term* (*term-of-pair* (*t*, *k*)) = *k*
by (*simp add*: *component-of-term-def pair-term*)

9.3.1 Additive Structure of Terms

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

Note that the argument order in (⊖) is reversed compared to the order in (⊕).

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

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

lemma *pp-of-term-splus* [*term-simps*]: *pp-of-term* (*t* ⊕ *v*) = *t* + *pp-of-term v*
by (*simp add*: *splus-def term-simps*)

lemma *component-of-term-splus* [*term-simps*]: *component-of-term* (*t* ⊕ *v*) = *component-of-term v*
by (*simp add*: *splus-def term-simps*)

lemma *pp-of-term-sminus* [*term-simps*]: *pp-of-term* (*v* ⊖ *t*) = *pp-of-term v* − *t*
by (*simp add*: *sminus-def term-simps*)

lemma *component-of-term-sminus* [*term-simps*]: *component-of-term* (*v* ⊖ *t*) = *component-of-term v*
by (*simp add*: *sminus-def term-simps*)

lemma *splus-sminus* [*term-simps*]: (*t* ⊕ *v*) ⊖ *t* = *v*

by (*simp add*: *sminus-def term-simps*)

lemma *splus-zero* [*term-simps*]: *0* ⊕ *v* = *v*
  by (*simp add*: *splus-def term-simps*)

lemma *sminus-zero* [*term-simps*]: *v* ⊖ *0* = *v*
  by (*simp add*: *sminus-def term-simps*)

lemma *splus-assoc* [*ac-simps*]: (*s* + *t*) ⊕ *v* = *s* ⊕ (*t* ⊕ *v*)
  by (*simp add*: *splus-def ac-simps term-simps*)

lemma *splus-left-commute* [*ac-simps*]: *s* ⊕ (*t* ⊕ *v*) = *t* ⊕ (*s* ⊕ *v*)
  by (*simp add*: *splus-def ac-simps term-simps*)

lemma *splus-right-canc* [*term-simps*]: *t* ⊕ *v* = *s* ⊕ *v* ⟷ *t* = *s*
  by (*metis add-right-cancel pp-of-term-splus*)

lemma *splus-left-canc* [*term-simps*]: *t* ⊕ *v* = *t* ⊕ *u* ⟷ *v* = *u*
  by (*metis splus-sminus*)

lemma *adds-ppI* [*intro?*]:
  assumes *v* = *t* ⊕ *u*
  shows *t* $adds_p$ *v*
  by (*simp add*: *adds-pp-def assms splus-def term-simps*)

lemma *adds-ppE* [*elim?*]:
  assumes *t* $adds_p$ *v*
  obtains *u* **where** *v* = *t* ⊕ *u*
proof −
  from *assms* **obtain** *s* **where** ∗: *pp-of-term v* = *t* + *s* **unfolding** *adds-pp-def* **..**
  have *v* = *t* ⊕ (*term-of-pair* (*s*, *component-of-term v*))
    by (*simp add*: *splus-def term-simps*, *metis* ∗ *add.commute term-of-pair-pair*)
  thus *?thesis* **..**
qed

lemma *adds-pp-alt*: *t* $adds_p$ *v* ⟷ (∃ *u*. *v* = *t* ⊕ *u*)
  by (*meson adds-ppE adds-ppI*)

lemma *adds-pp-refl* [*term-simps*]: (*pp-of-term v*) $adds_p$ *v*
  by (*simp add*: *adds-pp-def*)

lemma *adds-pp-trans* [*trans*]:
  assumes *s adds t* **and** *t* $adds_p$ *v*
  shows *s* $adds_p$ *v*
proof −
  note *assms*(*1*)
  also from *assms*(*2*) **have** *t adds pp-of-term v* **by** (*simp only*: *adds-pp-def*)
  finally show *?thesis* **by** (*simp only*: *adds-pp-def*)
qed

lemma *zero-adds-pp* [*term-simps*]: *0* $adds_p$ *v*
  by (*simp add*: *adds-pp-def*)

lemma *adds-pp-splus*:
  assumes *t* $adds_p$ *v*
  shows *t* $adds_p$ *s* ⊕ *v*
  using *assms* **by** (*simp add*: *adds-pp-def term-simps*)

lemma *adds-pp-triv* [*term-simps*]: *t* $adds_p$ *t* ⊕ *v*
  by (*simp add*: *adds-pp-def term-simps*)

lemma *plus-adds-pp-mono*:
  assumes *s adds t*
    and *u* $adds_p$ *v*
  shows *s* + *u* $adds_p$ *t* ⊕ *v*
  using *assms* **by** (*simp add*: *adds-pp-def term-simps*) (*rule plus-adds-mono*)

lemma *plus-adds-pp-left*:
  assumes *s* + *t* $adds_p$ *v*
  shows *s* $adds_p$ *v*
  using *assms* **by** (*simp add*: *adds-pp-def plus-adds-left*)

lemma *plus-adds-pp-right*:
  assumes *s* + *t* $adds_p$ *v*
  shows *t* $adds_p$ *v*
  using *assms* **by** (*simp add*: *adds-pp-def plus-adds-right*)

lemma *adds-pp-sminus*:
  assumes *t* $adds_p$ *v*
  shows *t* ⊕ (*v* ⊖ *t*) = *v*
proof −
  from *assms adds-pp-alt*[*of t v*] **obtain** *u* **where** *u*: *v* = *t* ⊕ *u* **by** (*auto simp*: *ac-simps*)
  hence *v* ⊖ *t* = *u* **by** (*simp add*: *term-simps*)
  thus *?thesis* **using** *u* **by** *simp*
qed

lemma *adds-pp-canc*: *t* + *s* $adds_p$ (*t* ⊕ *v*) ⟷ *s* $adds_p$ *v*
  by (*simp add*: *adds-pp-def adds-canc-2 term-simps*)

lemma *adds-pp-canc-2*: *s* + *t* $adds_p$ (*t* ⊕ *v*) ⟷ *s* $adds_p$ *v*
  by (*simp add*: *adds-pp-canc add.commute*[*of s t*])

lemma *plus-adds-pp-0*:
  assumes (*s* + *t*) $adds_p$ *v*
  shows *s* $adds_p$ (*v* ⊖ *t*)
  using *assms* **by** (*simp add*: *adds-pp-def term-simps*) (*rule plus-adds-0*)

lemma *plus-adds-ppI-1*:
assumes *t* $adds_p$ *v* **and** *s* $adds_p$ $(v \ominus t)$
shows $(s + t)$ $adds_p$ *v*
using *assms* **by** (*simp add*: *adds-pp-def term-simps*) (*rule plus-adds-2*)

lemma *plus-adds-ppI-2*:
assumes *t* $adds_p$ *v* **and** *s* $adds_p$ $(v \ominus t)$
shows $(t + s)$ $adds_p$ *v*
unfolding *add.commute*[*of t s*] **using** *assms* **by** (*rule plus-adds-ppI-1*)

lemma *plus-adds-pp*: $(s + t)$ $adds_p$ $v \longleftrightarrow (t$ $adds_p$ $v \wedge s$ $adds_p$ $(v \ominus t))$
by (*simp add*: *adds-pp-def plus-adds term-simps*)

lemma *minus-splus*:
assumes *s adds t*
shows $(t - s) \oplus v = (t \oplus v) \ominus s$
by (*simp add*: *assms minus-plus sminus-def splus-def term-simps*)

lemma *minus-splus-sminus*:
assumes *s adds t* **and** *u* $adds_p$ *v*
shows $(t - s) \oplus (v \ominus u) = (t \oplus v) \ominus (s + u)$
using *assms minus-plus-minus term-powerprod.adds-pp-def term-powerprod-axioms sminus-def*
splus-def term-simps **by** *fastforce*

lemma *minus-splus-sminus-cancel*:
assumes *s adds t* **and** *t* $adds_p$ *v*
shows $(t - s) \oplus (v \ominus t) = v \ominus s$
by (*simp add*: *adds-pp-sminus assms minus-splus*)

lemma *sminus-plus*:
assumes *s* $adds_p$ *v* **and** *t* $adds_p$ $(v \ominus s)$
shows $v \ominus (s + t) = (v \ominus s) \ominus t$
by (*simp add*: *diff-diff-add sminus-def term-simps*)

lemma *adds-termI* [*intro?*]:
assumes $v = t \oplus u$
shows *u* $adds_t$ *v*
by (*simp add*: *adds-term-def assms splus-def term-simps*)

lemma *adds-termE* [*elim?*]:
assumes *u* $adds_t$ *v*
obtains *t* **where** $v = t \oplus u$
proof −
from *assms* **have** *eq*: *component-of-term u* = *component-of-term v* **and** *pp-of-term u adds pp-of-term v*
by (*simp-all add*: *adds-term-def*)
from *this*(*2*) **obtain** *s* **where** ∗: *s* + *pp-of-term u* = *pp-of-term v* **unfolding** *adds-term-def*

using *adds-minus* **by** *blast*
have $v = s \oplus u$ **by** (*simp add*: *splus-def eq* $*$ *term-simps*)
thus *?thesis* **..**
qed

lemma *adds-term-alt*: u $adds_t$ $v \longleftrightarrow (\exists\, t.\ v = t \oplus u)$
by (*meson adds-termE adds-termI*)

lemma *adds-term-refl* [*term-simps*]: v $adds_t$ v
by (*simp add*: *adds-term-def*)

lemma *adds-term-trans* [*trans*]:
assumes u $adds_t$ v **and** v $adds_t$ w
shows u $adds_t$ w
using *assms* **unfolding** *adds-term-def* **using** *adds-trans* **by** *auto*

lemma *adds-term-splus*:
assumes u $adds_t$ v
shows u $adds_t$ $s \oplus v$
using *assms* **by** (*simp add*: *adds-term-def term-simps*)

lemma *adds-term-triv* [*term-simps*]: v $adds_t$ $t \oplus v$
by (*simp add*: *adds-term-def term-simps*)

lemma *splus-adds-term-mono*:
assumes s *adds* t
and u $adds_t$ v
shows $s \oplus u$ $adds_t$ $t \oplus v$
using *assms* **by** (*auto simp*: *adds-term-def term-simps intro*: *plus-adds-mono*)

lemma *splus-adds-term*:
assumes $t \oplus u$ $adds_t$ v
shows u $adds_t$ v
using *assms* **by** (*auto simp add*: *adds-term-def term-simps elim*: *plus-adds-right*)

lemma *adds-term-adds-pp*:
u $adds_t$ $v \longleftrightarrow$ (*component-of-term* u = *component-of-term* v $\wedge$ *pp-of-term* u $adds_p$ v)
by (*simp add*: *adds-term-def adds-pp-def*)

lemma *adds-term-canc*: $t \oplus u$ $adds_t$ $t \oplus v \longleftrightarrow u$ $adds_t$ v
by (*simp add*: *adds-term-def adds-canc-2 term-simps*)

lemma *adds-term-canc-2*: $s \oplus v$ $adds_t$ $t \oplus v \longleftrightarrow s$ *adds* t
by (*simp add*: *adds-term-def adds-canc term-simps*)

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

using *assms* **by** (*simp add*: *adds-term-def add.commute*[*of t*] *term-simps*) (*auto intro*: *plus-adds-0*)

lemma *splus-adds-termI-1*:
assumes $t\ adds_p\ v$ **and** $u\ adds_t\ (v \ominus t)$
shows $t \oplus u\ adds_t\ v$
using *assms* **apply** (*simp add*: *adds-term-def term-simps*) **by** (*metis add.commute adds-pp-def plus-adds-2*)

lemma *splus-adds-term-iff*: $t \oplus u\ adds_t\ v \longleftrightarrow (t\ adds_p\ v \wedge u\ adds_t\ (v \ominus t))$
by (*metis adds-ppI adds-pp-splus adds-termE splus-adds-termI-1 splus-adds-term-0*)

lemma *adds-minus-splus*:
assumes *pp-of-term u adds t*
shows (*t* − *pp-of-term u*) ⊕ *u* = *term-of-pair* (*t*, *component-of-term u*)
by (*simp add*: *splus-def adds-minus*[*OF assms*])

9.3.2 Projections and Conversions

lift-definition *proj-poly* :: $'k \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'b::zero)$
is λ*k p t*. *p* (*term-of-pair* (*t*, *k*))
proof −
fix *k*::'*k* **and** *p*::'*t* ⇒ '*b*
assume *fin*: *finite* {*v*. *p v* ≠ *0*}
have {*t*. *p* (*term-of-pair* (*t*, *k*)) ≠ *0*} ⊆ *pp-of-term* ‘ {*v*. *p v* ≠ *0*}
proof (*rule*, *simp*)
fix *t*
assume *p* (*term-of-pair* (*t*, *k*)) ≠ *0*
hence ∗: *term-of-pair* (*t*, *k*) ∈ {*v*. *p v* ≠ *0*} **by** *simp*
have *t* = *pp-of-term* (*term-of-pair* (*t*, *k*)) **by** (*simp add*: *pp-of-term-def pair-term*)
from *this* ∗ **show** *t* ∈ *pp-of-term* ‘ {*v*. *p v* ≠ *0*} **..**
qed
moreover from *fin* **have** *finite* (*pp-of-term* ‘ {*v*. *p v* ≠ *0*}) **by** (*rule finite-imageI*)
ultimately show *finite* {*t*. *p* (*term-of-pair* (*t*, *k*)) ≠ *0*} **by** (*rule finite-subset*)
qed

definition *vectorize-poly* :: $('t \Rightarrow_0 'b) \Rightarrow ('k \Rightarrow_0 ('a \Rightarrow_0 'b::zero))$
where *vectorize-poly p* = *Abs-poly-mapping* (λ*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* (λ*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*))
by (*transfer*, *simp*)

lemma *lookup-vectorize-poly*: *lookup* (*vectorize-poly p*) *k* = *proj-poly k p*
proof −
have *lookup* (*Abs-poly-mapping* (λ*k*. *proj-poly k p*)) = (λ*k*. *proj-poly k p*)

proof (*rule Abs-poly-mapping-inverse*, *simp*)
have {*k*. *proj-poly k p* ≠ *0*} ⊆ *component-of-term* ' *keys p*
proof (*rule*, *simp*)
fix *k*
assume *proj-poly k p* ≠ *0*
hence *keys* (*proj-poly k p*) ≠ {} **using** *poly-mapping-eq-zeroI* **by** *blast*
then obtain *t* **where** *lookup* (*proj-poly k p*) *t* ≠ *0* **by** *blast*
hence *term-of-pair* (*t*, *k*) ∈ *keys p* **by** (*simp add*: *lookup-proj-poly in-keys-iff*)
hence *component-of-term* (*term-of-pair* (*t*, *k*)) ∈ *component-of-term* ' *keys p* **by** *fastforce*
thus *k* ∈ *component-of-term* ' *keys p* **by** (*simp add*: *term-simps*)
qed
moreover from *finite-keys* **have** *finite* (*component-of-term* ' *keys p*) **by** (*rule finite-imageI*)
ultimately show *finite* {*k*. *proj-poly k p* ≠ *0*} **by** (*rule finite-subset*)
qed
thus *?thesis* **by** (*simp add*: *vectorize-poly-def*)
qed

lemma *lookup-atomize-poly*:
lookup (*atomize-poly p*) *v* = *lookup* (*lookup p* (*component-of-term v*)) (*pp-of-term v*)
proof −
have *lookup* (*Abs-poly-mapping* (λ*v*. *lookup* (*lookup p* (*component-of-term v*)) (*pp-of-term v*))) =
(λ*v*. *lookup* (*lookup p* (*component-of-term v*)) (*pp-of-term v*))
proof (*rule Abs-poly-mapping-inverse*, *simp*)
have {*v*. *pp-of-term v* ∈ *keys* (*lookup p* (*component-of-term v*))} ⊆
(⋃*k*∈*keys p*. (λ*t*. *term-of-pair* (*t*, *k*)) ' *keys* (*lookup p k*)) (**is** - ⊆ *?A*)
proof (*rule*, *simp*)
fix *v*
assume ∗: *pp-of-term v* ∈ *keys* (*lookup p* (*component-of-term v*))
hence *keys* (*lookup p* (*component-of-term v*)) ≠ {} **by** *blast*
hence *lookup p* (*component-of-term v*) ≠ *0* **by** *auto*
hence *component-of-term v* ∈ *keys p* (**is** *?k* ∈ -)
by (*simp add*: *in-keys-iff*)
thus ∃*k*∈*keys p*. *v* ∈ (λ*t*. *term-of-pair* (*t*, *k*)) ' *keys* (*lookup p k*)
proof
have *v* = *term-of-pair* (*pp-of-term v*, *component-of-term v*) **by** (*simp add*: *term-simps*)
from *this* ∗ **show** *v* ∈ (λ*t*. *term-of-pair* (*t*, *?k*)) ' *keys* (*lookup p ?k*) **..**
qed
qed
moreover have *finite ?A* **by** (*rule*, *fact finite-keys*, *rule finite-imageI*, *rule finite-keys*)
ultimately show *finite* {*x*. *lookup* (*lookup p* (*component-of-term x*)) (*pp-of-term x*) ≠ *0*}
by (*simp add*: *finite-subset in-keys-iff*)
qed

thus *?thesis* **by** (*simp add*: *atomize-poly-def*)
qed

lemma *keys-proj-poly*: *keys* (*proj-poly k p*) = *pp-of-term* ' {*x*∈*keys p*. *component-of-term x* = *k*}
proof
show *keys* (*proj-poly k p*) ⊆ *pp-of-term* ' {*x*∈*keys p*. *component-of-term x* = *k*}
proof
fix *t*
assume *t* ∈ *keys* (*proj-poly k p*)
hence *lookup* (*proj-poly k p*) *t* ≠ *0* **by** (*simp add*: *in-keys-iff*)
hence *term-of-pair* (*t*, *k*) ∈ *keys p* **by** (*simp add*: *in-keys-iff lookup-proj-poly*)
hence *term-of-pair* (*t*, *k*) ∈ {*x*∈*keys p*. *component-of-term x* = *k*} **by** (*simp add*: *term-simps*)
hence *pp-of-term* (*term-of-pair* (*t*, *k*)) ∈ *pp-of-term* ' {*x*∈*keys p*. *component-of-term x* = *k*} **by** (*rule imageI*)
thus *t* ∈ *pp-of-term* ' {*x*∈*keys p*. *component-of-term x* = *k*} **by** (*simp only*: *pp-of-term-of-pair*)
qed
next
show *pp-of-term* ' {*x*∈*keys p*. *component-of-term x* = *k*} ⊆ *keys* (*proj-poly k p*)
proof
fix *t*
assume *t* ∈ *pp-of-term* ' {*x*∈*keys p*. *component-of-term x* = *k*}
then obtain *x* **where** *x* ∈ {*x*∈*keys p*. *component-of-term x* = *k*} **and** *t* = *pp-of-term x* **..**
from *this*(*1*) **have** *x* ∈ *keys p* **and** *k* = *component-of-term x* **by** *simp-all*
from *this*(*2*) **have** *x* = *term-of-pair* (*t*, *k*) **by** (*simp add*: *term-of-pair-pair* ‹*t* = *pp-of-term x*›)
with ‹*x* ∈ *keys p*› **have** *lookup p* (*term-of-pair* (*t*, *k*)) ≠ *0* **by** (*simp add*: *in-keys-iff*)
hence *lookup* (*proj-poly k p*) *t* ≠ *0* **by** (*simp add*: *lookup-proj-poly*)
thus *t* ∈ *keys* (*proj-poly k p*) **by** (*simp add*: *in-keys-iff*)
qed
qed

lemma *keys-vectorize-poly*: *keys* (*vectorize-poly p*) = *component-of-term* ' *keys p*
proof
show *keys* (*vectorize-poly p*) ⊆ *component-of-term* ' *keys p*
proof
fix *k*
assume *k* ∈ *keys* (*vectorize-poly p*)
hence *lookup* (*vectorize-poly p*) *k* ≠ *0* **by** (*simp add*: *in-keys-iff*)
hence *proj-poly k p* ≠ *0* **by** (*simp add*: *lookup-vectorize-poly*)
then obtain *t* **where** *lookup* (*proj-poly k p*) *t* ≠ *0* **using** *aux* **by** *blast*
hence *term-of-pair* (*t*, *k*) ∈ *keys p* **by** (*simp add*: *lookup-proj-poly in-keys-iff*)
hence *component-of-term* (*term-of-pair* (*t*, *k*)) ∈ *component-of-term* ' *keys p* **by** (*rule imageI*)
thus *k* ∈ *component-of-term* ' *keys p* **by** (*simp only*: *component-of-term-of-pair*)

qed
next
show *component-of-term ` keys p* ⊆ *keys* (*vectorize-poly p*)
proof
fix *k*
assume *k* ∈ *component-of-term ` keys p*
then obtain *x* **where** *x* ∈ *keys p* **and** *k* = *component-of-term x* **..**
from *this*(*2*) **have** *term-of-pair* (*pp-of-term x*, *k*) = *x* **by** (*simp add*: *term-of-pair-pair*)
with ‹*x* ∈ *keys p*› **have** *lookup p* (*term-of-pair* (*pp-of-term x*, *k*)) ≠ *0* **by** (*simp add*: *in-keys-iff*)
hence *lookup* (*proj-poly k p*) (*pp-of-term x*) ≠ *0* **by** (*simp add*: *lookup-proj-poly*)
hence *proj-poly k p* ≠ *0* **by** *auto*
hence *lookup* (*vectorize-poly p*) *k* ≠ *0* **by** (*simp add*: *lookup-vectorize-poly*)
thus *k* ∈ *keys* (*vectorize-poly p*) **by** (*simp add*: *in-keys-iff*)
qed
qed

lemma *keys-atomize-poly*:
keys (*atomize-poly p*) = (⋃*k*∈*keys p*. (λ*t*. *term-of-pair* (*t*, *k*)) *` keys* (*lookup p k*)) (**is** *?l* = *?r*)
proof
show *?l* ⊆ *?r*
proof
fix *v*
assume *v* ∈ *?l*
hence *lookup* (*atomize-poly p*) *v* ≠ *0* **by** (*simp add*: *in-keys-iff*)
hence ∗: *pp-of-term v* ∈ *keys* (*lookup p* (*component-of-term v*)) **by** (*simp add*: *in-keys-iff lookup-atomize-poly*)
hence *lookup p* (*component-of-term v*) ≠ *0* **by** *fastforce*
hence *component-of-term v* ∈ *keys p* **by** (*simp add*: *in-keys-iff*)
thus *v* ∈ *?r*
proof
from ∗ **have** *term-of-pair* (*pp-of-term v*, *component-of-term v*) ∈
(λ*t*. *term-of-pair* (*t*, *component-of-term v*)) *` keys* (*lookup p* (*component-of-term v*))
by (*rule imageI*)
thus *v* ∈ (λ*t*. *term-of-pair* (*t*, *component-of-term v*)) *` keys* (*lookup p* (*component-of-term v*))
by (*simp only*: *term-of-pair-pair*)
qed
qed
next
show *?r* ⊆ *?l*
proof
fix *v*
assume *v* ∈ *?r*
then obtain *k* **where** *k* ∈ *keys p* **and** *v* ∈ (λ*t*. *term-of-pair* (*t*, *k*)) *` keys* (*lookup p k*) **..**
from *this*(*2*) **obtain** *t* **where** *t* ∈ *keys* (*lookup p k*) **and** *v*: *v* = *term-of-pair*

(*t*, *k*) **..**
from *this*(*1*) **have** *lookup* (*atomize-poly p*) $v \neq 0$ **by** (*simp add*: *v lookup-atomize-poly in-keys-iff term-simps*)
thus $v \in$ *?l* **by** (*simp add*: *in-keys-iff*)
qed
qed

lemma *proj-atomize-poly* [*term-simps*]: *proj-poly k* (*atomize-poly p*) = *lookup p k*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-proj-poly lookup-atomize-poly term-simps*)

lemma *vectorize-atomize-poly* [*term-simps*]: *vectorize-poly* (*atomize-poly p*) = *p*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-vectorize-poly term-simps*)

lemma *atomize-vectorize-poly* [*term-simps*]: *atomize-poly* (*vectorize-poly p*) = *p*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-atomize-poly lookup-vectorize-poly lookup-proj-poly term-simps*)

lemma *proj-zero* [*term-simps*]: *proj-poly k 0* = *0*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-proj-poly*)

lemma *proj-plus*: *proj-poly k* (*p* + *q*) = *proj-poly k p* + *proj-poly k q*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-proj-poly lookup-add*)

lemma *proj-uminus* [*term-simps*]: *proj-poly k* (− *p*) = − *proj-poly k p*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-proj-poly*)

lemma *proj-minus*: *proj-poly k* (*p* − *q*) = *proj-poly k p* − *proj-poly k q*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-proj-poly lookup-minus*)

lemma *vectorize-zero* [*term-simps*]: *vectorize-poly 0* = *0*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-vectorize-poly term-simps*)

lemma *vectorize-plus*: *vectorize-poly* (*p* + *q*) = *vectorize-poly p* + *vectorize-poly q*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-vectorize-poly lookup-add proj-plus*)

lemma *vectorize-uminus* [*term-simps*]: *vectorize-poly* (− *p*) = − *vectorize-poly p*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-vectorize-poly term-simps*)

lemma *vectorize-minus*: *vectorize-poly* (*p* − *q*) = *vectorize-poly p* − *vectorize-poly q*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-vectorize-poly lookup-minus proj-minus*)

lemma *atomize-zero* [*term-simps*]: *atomize-poly 0* = *0*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-atomize-poly*)

lemma *atomize-plus*: *atomize-poly* (*p* + *q*) = *atomize-poly p* + *atomize-poly q*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-atomize-poly lookup-add*)

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

by (*rule poly-mapping-eqI*, *simp add*: *lookup-atomize-poly*)

lemma *atomize-minus*: *atomize-poly* (*p* − *q*) = *atomize-poly p* − *atomize-poly q*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-atomize-poly lookup-minus*)

lemma *proj-monomial*:
proj-poly k (*monomial c v*) = (*monomial c* (*pp-of-term v*) *when component-of-term v* = *k*)
proof (*rule poly-mapping-eqI*, *simp add*: *lookup-proj-poly lookup-single when-def term-simps*, *intro impI*)
fix *t*
assume *1*: *pp-of-term v* = *t* **and** *2*: *component-of-term v* = *k*
assume *v* ≠ *term-of-pair* (*t*, *k*)
moreover have *v* = *term-of-pair* (*t*, *k*) **by** (*simp add*: *1*[*symmetric*] *2*[*symmetric*] *term-simps*)
ultimately show *c* = *0* **..**
qed

lemma *vectorize-monomial*:
vectorize-poly (*monomial c v*) = *monomial* (*monomial c* (*pp-of-term v*)) (*component-of-term v*)
by (*rule poly-mapping-eqI*, *simp add*: *lookup-vectorize-poly proj-monomial lookup-single*)

lemma *atomize-monomial-monomial*:
atomize-poly (*monomial* (*monomial c t*) *k*) = *monomial c* (*term-of-pair* (*t*, *k*))
proof −
define *v* **where** *v* = *term-of-pair* (*t*, *k*)
have *t*: *t* = *pp-of-term v* **and** *k*: *k* = *component-of-term v* **by** (*simp-all add*: *v-def term-simps*)
show *?thesis* **by** (*simp add*: *t k vectorize-monomial*[*symmetric*] *term-simps*)
qed

lemma *poly-mapping-eqI-proj*:
assumes $\bigwedge$*k*. *proj-poly k p* = *proj-poly k q*
shows *p* = *q*
proof (*rule poly-mapping-eqI*)
fix *v*::′*t*
have *proj-poly* (*component-of-term v*) *p* = *proj-poly* (*component-of-term v*) *q* **by** (*rule assms*)
hence *lookup* (*proj-poly* (*component-of-term v*) *p*) (*pp-of-term v*) =
lookup (*proj-poly* (*component-of-term v*) *q*) (*pp-of-term v*) **by** *simp*
thus *lookup p v* = *lookup q v* **by** (*simp add*: *lookup-proj-poly term-simps*)
qed

9.4 Scalar Multiplication by Monomials

definition *monom-mult* :: ′*b*::*semiring-0* ⇒ ′*a*::*comm-powerprod* ⇒ (′*t* $\Rightarrow_0$ ′*b*) ⇒ (′*t* $\Rightarrow_0$ ′*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
fix *v*::'*t*
assume *v* ∈ *?l*
hence (*if t adds*$_p$ *v then c* ∗ *lookup p* (*v* ⊖ *t*) *else 0*) ≠ *0* **by** *simp*
hence *t adds*$_p$ *v* **and** *cp-not-zero*: *c* ∗ *lookup p* (*v* ⊖ *t*) ≠ *0* **by** (*simp-all split*: *if-split-asm*)
show *v* ∈ *?r*
proof
from *adds-pp-sminus*[*OF* ‹*t adds*$_p$ *v*›] **show** *v* = *t* ⊕ (*v* ⊖ *t*) **by** *simp*
next
from *mult-not-zero*[*OF cp-not-zero*] **show** *v* ⊖ *t* ∈ *keys p*
by (*simp add*: *in-keys-iff*)
qed
qed

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 −
have *lookup* (*monom-mult c t p*) = (λ*v*. *if t adds*$_p$ *v then c* ∗ *lookup p* (*v* ⊖ *t*) *else 0*)
unfolding *monom-mult-def*
proof (*rule Abs-poly-mapping-inverse*)
from *finite-keys* **have** *finite* ((⊕) *t* ' *keys p*) **..**
with *keys-monom-mult-aux* **have** *finite* {*v*. (*if t adds*$_p$ *v then c* ∗ *lookup p* (*v* ⊖ *t*) *else 0*) ≠ *0*}
by (*rule finite-subset*)
thus (λ*v*. *if t adds*$_p$ *v then c* ∗ *lookup p* (*v* ⊖ *t*) *else 0*) ∈ {*f*. *finite* {*x*. *f x* ≠ *0*}} **by** *simp*
qed
thus *?thesis* **by** *simp*
qed

lemma *lookup-monom-mult-plus*:
lookup (*monom-mult c t p*) (*t* ⊕ *v*) = (*c*::'*b*::*semiring-0*) ∗ *lookup p v*
by (*simp add*: *lookup-monom-mult term-simps*)

lemma *monom-mult-assoc*: *monom-mult c s* (*monom-mult d t p*) = *monom-mult* (*c* ∗ *d*) (*s* + *t*) *p*
proof (*rule poly-mapping-eqI*, *simp add*: *lookup-monom-mult sminus-plus ac-simps*, *intro conjI impI*)
fix *v*
assume *s adds*$_p$ *v* **and** *t adds*$_p$ *v* ⊖ *s*
hence *s* + *t adds*$_p$ *v* **by** (*rule plus-adds-ppI-2*)

moreover assume ¬ *s* + *t adds*$_p$ *v*
ultimately show *c* ∗ (*d* ∗ *lookup p* (*v* ⊖ *s* ⊖ *t*)) = *0* **by** *simp*
next
fix *v*
assume *s* + *t adds*$_p$ *v*
hence *s adds*$_p$ *v* **by** (*rule plus-adds-pp-left*)
moreover assume ¬ *s adds*$_p$ *v*
ultimately show *c* ∗ (*d* ∗ *lookup p* (*v* ⊖ (*s* + *t*))) = *0* **by** *simp*
next
fix *v*
assume *s* + *t adds*$_p$ *v*
hence *t adds*$_p$ *v* ⊖ *s* **by** (*simp add*: *add.commute plus-adds-pp-0*)
moreover assume ¬ *t adds*$_p$ *v* ⊖ *s*
ultimately show *c* ∗ (*d* ∗ *lookup p* (*v* ⊖ (*s* + *t*))) = *0* **by** *simp*
qed

lemma *monom-mult-uminus-left*: *monom-mult* (− *c*) *t p* = − *monom-mult* (*c*::′*b*::*ring*) *t p*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-monom-mult*)

lemma *monom-mult-uminus-right*: *monom-mult c t* (− *p*) = − *monom-mult* (*c*::′*b*::*ring*) *t p*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-monom-mult*)

lemma *uminus-monom-mult*: − *p* = *monom-mult* (−*1*::′*b*::*comm-ring-1*) *0 p*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-monom-mult term-simps*)

lemma *monom-mult-dist-left*: *monom-mult* (*c* + *d*) *t p* = (*monom-mult c t p*) + (*monom-mult d t p*)
by (*rule poly-mapping-eqI*, *simp add*: *lookup-monom-mult lookup-add algebra-simps*)

lemma *monom-mult-dist-left-minus*:
monom-mult (*c* − *d*) *t p* = (*monom-mult c t p*) − (*monom-mult* (*d*::′*b*::*ring*) *t p*)
using *monom-mult-dist-left*[*of c* −*d t p*] *monom-mult-uminus-left*[*of d t p*] **by** *simp*

lemma *monom-mult-dist-right*:
monom-mult c t (*p* + *q*) = (*monom-mult c t p*) + (*monom-mult c t q*)
by (*rule poly-mapping-eqI*, *simp add*: *lookup-monom-mult lookup-add algebra-simps*)

lemma *monom-mult-dist-right-minus*:
monom-mult c t (*p* − *q*) = (*monom-mult c t p*) − (*monom-mult* (*c*::′*b*::*ring*) *t q*)
using *monom-mult-dist-right*[*of c t p* −*q*] *monom-mult-uminus-right*[*of c t q*] **by** *simp*

lemma *monom-mult-zero-left* [*simp*]: *monom-mult 0 t p* = *0*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-monom-mult*)

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

by (*rule poly-mapping-eqI*, *simp add*: *lookup-monom-mult*)

lemma *monom-mult-one-left* [*simp*]: (*monom-mult* (*1*::′*b*::*semiring-1*) *0 p*) = *p*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-monom-mult term-simps*)

lemma *monom-mult-monomial*:
monom-mult c s (*monomial d v*) = *monomial* (*c* ∗ (*d*::′*b*::*semiring-0*)) (*s* ⊕ *v*)
by (*rule poly-mapping-eqI*, *auto simp add*: *lookup-monom-mult lookup-single adds-pp-alt when-def term-simps*, *metis*)

lemma *monom-mult-eq-zero-iff*: (*monom-mult c t p* = *0*) ⟷ ((*c*::′*b*::*semiring-no-zero-divisors*) = *0* ∨ *p* = *0*)
proof
assume *eq*: *monom-mult c t p* = *0*
show *c* = *0* ∨ *p* = *0*
proof (*rule ccontr*, *simp*)
assume *c* ≠ *0* ∧ *p* ≠ *0*
hence *c* ≠ *0* **and** *p* ≠ *0* **by** *simp-all*
from *lookup-zero poly-mapping-eq-iff*[*of p 0*] ‹*p* ≠ *0*› **obtain** *v* **where** *lookup p v* ≠ *0* **by** *fastforce*
from *eq lookup-zero* **have** *lookup* (*monom-mult c t p*) (*t* ⊕ *v*) = *0* **by** *simp*
hence *c* ∗ *lookup p v* = *0* **by** (*simp only*: *lookup-monom-mult-plus*)
with ‹*c* ≠ *0*› ‹*lookup p v* ≠ *0*› **show** *False* **by** *auto*
qed
next
assume *c* = *0* ∨ *p* = *0*
with *monom-mult-zero-left*[*of t p*] *monom-mult-zero-right*[*of c t*] **show** *monom-mult c t p* = *0* **by** *auto*
qed

lemma *lookup-monom-mult-zero*: *lookup* (*monom-mult c 0 p*) *t* = *c* ∗ *lookup p t*
proof −
have *lookup* (*monom-mult c 0 p*) *t* = *lookup* (*monom-mult c 0 p*) (*0* ⊕ *t*) **by** (*simp add*: *term-simps*)
also have ... = *c* ∗ *lookup p t* **by** (*rule lookup-monom-mult-plus*)
finally show *?thesis* **.**
qed

lemma *monom-mult-inj-1*:
assumes *monom-mult c1 t p* = *monom-mult c2 t p*
and (*p*::(- ⇒$_0$ ′*b*::*semiring-no-zero-divisors-cancel*)) ≠ *0*
shows *c1* = *c2*
proof −
from *assms*(*2*) **have** *keys p* ≠ {} **using** *poly-mapping-eq-zeroI* **by** *blast*
then obtain *v* **where** *v* ∈ *keys p* **by** *blast*
hence ∗: *lookup p v* ≠ *0* **by** (*simp add*: *in-keys-iff*)
from *assms*(*1*) **have** *lookup* (*monom-mult c1 t p*) (*t* ⊕ *v*) = *lookup* (*monom-mult c2 t p*) (*t* ⊕ *v*)
by *simp*

hence *c1* ∗ *lookup p v* = *c2* ∗ *lookup p v* **by** (*simp only*: *lookup-monom-mult-plus*)
with ∗ **show** *?thesis* **by** *auto*
qed

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* ≠ *0*
shows *p1* = *p2*
proof (*rule poly-mapping-eqI*)
fix *v*
from *assms*(*1*) **have** *lookup* (*monom-mult c t p1*) (*t* ⊕ *v*) = *lookup* (*monom-mult c t p2*) (*t* ⊕ *v*)
by *simp*
hence *c* ∗ *lookup p1 v* = *c* ∗ *lookup p2 v* **by** (*simp only*: *lookup-monom-mult-plus*)
with *assms*(*2*) **show** *lookup p1 v* = *lookup p2 v* **by** *simp*
qed

lemma *keys-monom-multI*:
assumes *v* ∈ *keys p* **and** *c* ≠ (*0*::*'b*::*semiring-no-zero-divisors*)
shows *t* ⊕ *v* ∈ *keys* (*monom-mult c t p*)
using *assms* **unfolding** *in-keys-iff lookup-monom-mult-plus* **by** *simp*

lemma *keys-monom-mult-subset*: *keys* (*monom-mult c t p*) ⊆ ((⊕) *t*) ‘ (*keys p*)
proof −
have *keys* (*monom-mult c t p*) ⊆ {*v*. (*if t* $adds_p$ *v then c* ∗ *lookup p* (*v* ⊖ *t*) *else* *0*) ≠ *0*} (**is** - ⊆ *?A*)
proof
fix *v*
assume *v* ∈ *keys* (*monom-mult c t p*)
hence *lookup* (*monom-mult c t p*) *v* ≠ *0* **by** (*simp add*: *in-keys-iff*)
thus *v* ∈ *?A* **unfolding** *lookup-monom-mult* **by** *simp*
qed
also note *keys-monom-mult-aux*
finally show *?thesis* **.**
qed

lemma *keys-monom-multE*:
assumes *v* ∈ *keys* (*monom-mult c t p*)
obtains *u* **where** *u* ∈ *keys p* **and** *v* = *t* ⊕ *u*
proof −
note *assms*
also have *keys* (*monom-mult c t p*) ⊆ ((⊕) *t*) ‘ (*keys p*) **by** (*fact keys-monom-mult-subset*)
finally have *v* ∈ ((⊕) *t*) ‘ (*keys p*) **.**
then obtain *u* **where** *u* ∈ *keys p* **and** *v* = *t* ⊕ *u* **..**
thus *?thesis* **..**
qed

lemma *keys-monom-mult*:
assumes *c* ≠ (*0*::′*b*::*semiring-no-zero-divisors*)
shows *keys* (*monom-mult c t p*) = ((⊕) *t*) ‘ (*keys p*)
proof (*rule*, *fact keys-monom-mult-subset*, *rule*)
fix *v*
assume *v* ∈ (⊕) *t* ‘ *keys p*
then obtain *u* **where** *u* ∈ *keys p* **and** *v*: *v* = *t* ⊕ *u* **..**
from ‹*u* ∈ *keys p*› *assms* **show** *v* ∈ *keys* (*monom-mult c t p*) **unfolding** *v* **by** (*rule keys-monom-multI*)
qed

lemma *monom-mult-when*: *monom-mult c t* (*p when P*) = ((*monom-mult c t p*) *when P*)
by (*cases P*, *simp-all*)

lemma *when-monom-mult*: *monom-mult* (*c when P*) *t p* = ((*monom-mult c t p*) *when P*)
by (*cases P*, *simp-all*)

lemma *monomial-power*: (*monomial c t*) ^ *n* = *monomial* (*c* ^ *n*) (∑ *i*=*0*..<*n*. *t*)
by (*induct n*, *simp-all add*: *mult-single monom-mult-monomial add.commute*)

9.5 Component-wise Lifting

Component-wise lifting of functions on ′*a* ⇒$_0$ ′*b* to functions on ′*t* ⇒$_0$ ′*b*.

definition *lift-poly-fun-2* :: ((′*a* ⇒$_0$ ′*b*) ⇒ (′*a* ⇒$_0$ ′*b*) ⇒ (′*a* ⇒$_0$ ′*b*)) ⇒ (′*t* ⇒$_0$ ′*b*) ⇒ (′*t* ⇒$_0$ ′*b*) ⇒ (′*t* ⇒$_0$ ′*b*::*zero*)
where *lift-poly-fun-2 f p q* = *atomize-poly* (*mapp-2* (λ-. *f*) (*vectorize-poly p*) (*vectorize-poly q*))

definition *lift-poly-fun* :: ((′*a* ⇒$_0$ ′*b*) ⇒ (′*a* ⇒$_0$ ′*b*)) ⇒ (′*t* ⇒$_0$ ′*b*) ⇒ (′*t* ⇒$_0$ ′*b*::*zero*)
where *lift-poly-fun f p* = *lift-poly-fun-2* (λ-. *f*) *0 p*

lemma *lookup-lift-poly-fun-2*:
lookup (*lift-poly-fun-2 f p q*) *v* =
(*lookup* (*f* (*proj-poly* (*component-of-term v*) *p*) (*proj-poly* (*component-of-term v*) *q*)) (*pp-of-term v*)
when component-of-term v ∈ *keys* (*vectorize-poly p*) ∪ *keys* (*vectorize-poly q*))
by (*simp add*: *lift-poly-fun-2-def lookup-atomize-poly lookup-mapp-2 lookup-vectorize-poly when-distrib*[*of* - λ*q*. *lookup q* (*pp-of-term v*), *OF lookup-zero*])

lemma *lookup-lift-poly-fun*:
lookup (*lift-poly-fun f p*) *v* =
(*lookup* (*f* (*proj-poly* (*component-of-term v*) *p*)) (*pp-of-term v*) *when component-of-term v* ∈ *keys* (*vectorize-poly p*))
by (*simp add*: *lift-poly-fun-def lookup-lift-poly-fun-2 term-simps*)

lemma *lookup-lift-poly-fun-2-homogenous*:
assumes *f 0 0 = 0*
shows *lookup* (*lift-poly-fun-2 f p q*) *v* =
lookup (*f* (*proj-poly* (*component-of-term v*) *p*) (*proj-poly* (*component-of-term v*) *q*)) (*pp-of-term v*)
by (*simp add*: *lookup-lift-poly-fun-2 when-def in-keys-iff lookup-vectorize-poly assms*)

lemma *proj-lift-poly-fun-2-homogenous*:
assumes *f 0 0 = 0*
shows *proj-poly k* (*lift-poly-fun-2 f p q*) = *f* (*proj-poly k p*) (*proj-poly k q*)
by (*rule poly-mapping-eqI*,
simp add: *lookup-proj-poly lookup-lift-poly-fun-2-homogenous*[*of f*, *OF assms*] *term-simps*)

lemma *lookup-lift-poly-fun-homogenous*:
assumes *f 0 = 0*
shows *lookup* (*lift-poly-fun f p*) *v* = *lookup* (*f* (*proj-poly* (*component-of-term v*) *p*)) (*pp-of-term v*)
by (*simp add*: *lookup-lift-poly-fun when-def in-keys-iff lookup-vectorize-poly assms*)

lemma *proj-lift-poly-fun-homogenous*:
assumes *f 0 = 0*
shows *proj-poly k* (*lift-poly-fun f p*) = *f* (*proj-poly k p*)
by (*rule poly-mapping-eqI*,
simp add: *lookup-proj-poly lookup-lift-poly-fun-homogenous*[*of f*, *OF assms*] *term-simps*)

9.6 Component-wise Multiplication

definition *mult-vec* :: ($'t \Rightarrow_0 'b$) ⇒ ($'t \Rightarrow_0 'b$) ⇒ ($'t \Rightarrow_0 'b$::*semiring-0*) (**infixl** ‹**› *75*)
where *mult-vec* = *lift-poly-fun-2* (∗)

lemma *lookup-mult-vec*:
lookup (*p* ∗∗ *q*) *v* = *lookup* ((*proj-poly* (*component-of-term v*) *p*) ∗ (*proj-poly* (*component-of-term v*) *q*)) (*pp-of-term v*)
unfolding *mult-vec-def* **by** (*rule lookup-lift-poly-fun-2-homogenous*, *simp*)

lemma *proj-mult-vec* [*term-simps*]: *proj-poly k* (*p* ∗∗ *q*) = (*proj-poly k p*) ∗ (*proj-poly k q*)
unfolding *mult-vec-def* **by** (*rule proj-lift-poly-fun-2-homogenous*, *simp*)

lemma *mult-vec-zero-left*: *0* ∗∗ *p* = *0*
by (*rule poly-mapping-eqI-proj*, *simp add*: *term-simps*)

lemma *mult-vec-zero-right*: *p* ∗∗ *0* = *0*
by (*rule poly-mapping-eqI-proj*, *simp add*: *term-simps*)

lemma *mult-vec-assoc*: (*p* ∗∗ *q*) ∗∗ *r* = *p* ∗∗ (*q* ∗∗ *r*)
by (*rule poly-mapping-eqI-proj*, *simp add*: *ac-simps term-simps*)

lemma *mult-vec-distrib-right*: (*p* + *q*) ∗∗ *r* = *p* ∗∗ *r* + *q* ∗∗ *r*
by (*rule poly-mapping-eqI-proj*, *simp add*: *algebra-simps proj-plus term-simps*)

lemma *mult-vec-distrib-left*: *r* ∗∗ (*p* + *q*) = *r* ∗∗ *p* + *r* ∗∗ *q*
by (*rule poly-mapping-eqI-proj*, *simp add*: *algebra-simps proj-plus term-simps*)

lemma *mult-vec-minus-mult-left*: (− *p*) ∗∗ *q* = − (*p* ∗∗ *q*)
by (*rule sym*, *rule minus-unique*, *simp add*: *mult-vec-distrib-right*[*symmetric*] *mult-vec-zero-left*)

lemma *mult-vec-minus-mult-right*: *p* ∗∗ (− *q*) = − (*p* ∗∗ *q*)
by (*rule sym*, *rule minus-unique*, *simp add*: *mult-vec-distrib-left* [*symmetric*] *mult-vec-zero-right*)

lemma *minus-mult-vec-minus*: (− *p*) ∗∗ (− *q*) = *p* ∗∗ *q*
by (*simp add*: *mult-vec-minus-mult-left mult-vec-minus-mult-right*)

lemma *minus-mult-vec-commute*: (− *p*) ∗∗ *q* = *p* ∗∗ (− *q*)
by (*simp add*: *mult-vec-minus-mult-left mult-vec-minus-mult-right*)

lemma *mult-vec-right-diff-distrib*: *r* ∗∗ (*p* − *q*) = *r* ∗∗ *p* − *r* ∗∗ *q*
for *r*::- $\Rightarrow_0$ *'b*::*ring*
using *mult-vec-distrib-left* [*of r p* − *q*] **by** (*simp add*: *mult-vec-minus-mult-right*)

lemma *mult-vec-left-diff-distrib*: (*p* − *q*) ∗∗ *r* = *p* ∗∗ *r* − *q* ∗∗ *r*
for *p*::- $\Rightarrow_0$ *'b*::*ring*
using *mult-vec-distrib-right* [*of p* − *q r*] **by** (*simp add*: *mult-vec-minus-mult-left*)

lemma *mult-vec-commute*: *p* ∗∗ *q* = *q* ∗∗ *p* **for** *p*::- $\Rightarrow_0$ *'b*::*comm-semiring-0*
by (*rule poly-mapping-eqI-proj*, *simp add*: *term-simps ac-simps*)

lemma *mult-vec-left-commute*: *p* ∗∗ (*q* ∗∗ *r*) = *q* ∗∗ (*p* ∗∗ *r*)
for *p*::- $\Rightarrow_0$ *'b*::*comm-semiring-0*
by (*rule poly-mapping-eqI-proj*, *simp add*: *term-simps ac-simps*)

lemma *mult-vec-monomial-monomial*:
(*monomial c u*) ∗∗ (*monomial d v*) =
(*monomial* (*c* ∗ *d*) (*term-of-pair* (*pp-of-term u* + *pp-of-term v*, *component-of-term u*)) *when*
component-of-term u = *component-of-term v*)
by (*rule poly-mapping-eqI-proj*, *simp add*: *proj-monomial mult-single when-def term-simps*)

lemma *mult-vec-rec-left*: *p* ∗∗ *q* = *monomial* (*lookup p v*) *v* ∗∗ *q* + (*except p* {*v*}) ∗∗ *q*

proof −
from *plus-except*[*of p v*] **have** *p* ∗∗ *q* = (*monomial* (*lookup p v*) *v* + *except p* {*v*}) ∗∗ *q* **by** *simp*
also have ... = *monomial* (*lookup p v*) *v* ∗∗ *q* + *except p* {*v*} ∗∗ *q*
by (*simp only*: *mult-vec-distrib-right*)
finally show *?thesis* **.**
qed

lemma *mult-vec-rec-right*: *p* ∗∗ *q* = *p* ∗∗ *monomial* (*lookup q v*) *v* + *p* ∗∗ *except q* {*v*}
proof −
have *p* ∗∗ *monomial* (*lookup q v*) *v* + *p* ∗∗ *except q* {*v*} = *p* ∗∗ (*monomial* (*lookup q v*) *v* + *except q* {*v*})
by (*simp only*: *mult-vec-distrib-left*)
also have ... = *p* ∗∗ *q* **by** (*simp only*: *plus-except*[*of q v*, *symmetric*])
finally show *?thesis* **by** *simp*
qed

lemma *in-keys-mult-vecE*:
assumes *w* ∈ *keys* (*p* ∗∗ *q*)
obtains *u v* **where** *u* ∈ *keys p* **and** *v* ∈ *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 −
from *assms* **have** *0* ≠ *lookup* (*p* ∗∗ *q*) *w* **by** (*simp add*: *in-keys-iff*)
also have *lookup* (*p* ∗∗ *q*) *w* =
lookup ((*proj-poly* (*component-of-term w*) *p*) ∗ (*proj-poly* (*component-of-term w*) *q*)) (*pp-of-term w*)
by (*fact lookup-mult-vec*)
finally have *pp-of-term w* ∈ *keys* ((*proj-poly* (*component-of-term w*) *p*) ∗ (*proj-poly* (*component-of-term w*) *q*))
by (*simp add*: *in-keys-iff*)
from *this keys-mult*
have *pp-of-term w* ∈ {*t* + *s* |*t s*. *t* ∈ *keys* (*proj-poly* (*component-of-term w*) *p*) ∧
s ∈ *keys* (*proj-poly* (*component-of-term w*) *q*)} **..**
then obtain *t s* **where** *1*: *t* ∈ *keys* (*proj-poly* (*component-of-term w*) *p*)
and *2*: *s* ∈ *keys* (*proj-poly* (*component-of-term w*) *q*)
and *eq*: *pp-of-term w* = *t* + *s* **by** *fastforce*
let *?u* = *term-of-pair* (*t*, *component-of-term w*)
let *?v* = *term-of-pair* (*s*, *component-of-term w*)
from *1* **have** *?u* ∈ *keys p* **by** (*simp only*: *in-keys-iff lookup-proj-poly not-False-eq-True*)
moreover from *2* **have** *?v* ∈ *keys q* **by** (*simp only*: *in-keys-iff lookup-proj-poly not-False-eq-True*)
moreover have *component-of-term ?u* = *component-of-term ?v* **by** (*simp add*: *term-simps*)
moreover have *w* = *term-of-pair* (*pp-of-term ?u* + *pp-of-term ?v*, *component-of-term ?u*)
by (*simp add*: *eq*[*symmetric*] *term-simps*)
ultimately show *?thesis* **..**

qed

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 −
have *eq1*: *lookup* ((*monomial c* (*pp-of-term v*) *when component-of-term v* = *component-of-term u*) ∗ *proj-poly* (*component-of-term u*) *p*)
(*pp-of-term u*) =
(*lookup* ((*monomial c* (*pp-of-term v*)) ∗ *proj-poly* (*component-of-term u*) *p*) (*pp-of-term u*) *when*
component-of-term v = *component-of-term u*)
by (*rule when-distrib*, *simp*)
show *?thesis*
by (*simp add*: *lookup-mult-vec proj-monomial eq1 lookup-times-monomial-left when-when*
adds-term-def lookup-proj-poly conj-commute)
qed

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 −
have *eq1*: *lookup* (*proj-poly* (*component-of-term u*) *p* ∗ (*monomial c* (*pp-of-term v*) *when component-of-term v* = *component-of-term u*))
(*pp-of-term u*) =
(*lookup* (*proj-poly* (*component-of-term u*) *p* ∗ (*monomial c* (*pp-of-term v*))) (*pp-of-term u*) *when*
component-of-term v = *component-of-term u*)
by (*rule when-distrib*, *simp*)
show *?thesis*
by (*simp add*: *lookup-mult-vec proj-monomial eq1 lookup-times-monomial-right when-when*
adds-term-def lookup-proj-poly conj-commute)
qed

9.7 Scalar Multiplication

definition *mult-scalar* :: ('*a* $\Rightarrow_0$ '*b*) ⇒ ('*t* $\Rightarrow_0$ '*b*) ⇒ ('*t* $\Rightarrow_0$ '*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*)
unfolding *mult-scalar-def* **by** (*rule lookup-lift-poly-fun-homogenous*, *simp*)

lemma *lookup-mult-scalar-explicit*:
lookup ($p \odot q$) u = ($\sum$ $t \in$*keys* p. *lookup* p t $*$ ($\sum$ $v \in$*keys* q. *lookup* q v *when* u = $t \oplus v$))
proof −
let *?f* = λt s. *lookup* (*proj-poly* (*component-of-term* u) q) s *when* *pp-of-term* u = $t + s$
note *lookup-mult-scalar*
also have *lookup* (p $*$ *proj-poly* (*component-of-term* u) q) (*pp-of-term* u) =
($\sum$ t. *lookup* p t $*$ (*Sum-any* (*?f* t)))
by (*fact lookup-mult*)
also from *finite-keys* **have** ... = ($\sum$ $t \in$*keys* p. *lookup* p t $*$ (*Sum-any* (*?f* t)))
by (*rule Sum-any.expand-superset*) (*auto simp*: *in-keys-iff dest*: *mult-not-zero*)
also from *refl* **have** ... = ($\sum$ $t \in$*keys* p. *lookup* p t $*$ ($\sum$ $v \in$*keys* q. *lookup* q v *when* u = $t \oplus v$))
proof (*rule sum.cong*)
fix t
assume $t \in$ *keys* p
from *finite-keys* **have** *Sum-any* (*?f* t) = ($\sum$ $s \in$*keys* (*proj-poly* (*component-of-term* u) q). *?f* t s)
by (*rule Sum-any.expand-superset*) (*auto simp*: *in-keys-iff*)
also have ... = ($\sum$ $v \in \{x \in$ *keys* q. *component-of-term* x = *component-of-term* $u\}$. *?f* t (*pp-of-term* v))
unfolding *keys-proj-poly*
proof (*intro sum.reindex*[*simplified o-def*] *inj-onI*)
fix *v1 v2*
assume *v1* $\in \{x \in$ *keys* q. *component-of-term* x = *component-of-term* $u\}$
and *v2* $\in \{x \in$ *keys* q. *component-of-term* x = *component-of-term* $u\}$
hence *component-of-term v1* = *component-of-term v2* **by** *simp*
moreover assume *pp-of-term v1* = *pp-of-term v2*
ultimately show *v1* = *v2* **by** (*metis term-of-pair-pair*)
qed
also from *finite-keys* **have** ... = ($\sum$ $v \in$*keys* q. *lookup* q v *when* u = $t \oplus v$)
proof (*intro sum.mono-neutral-cong-left ballI*)
fix v
assume $v \in$ *keys* q − $\{x \in$ *keys* q. *component-of-term* x = *component-of-term* $u\}$
hence $u \neq t \oplus v$ **by** (*auto simp*: *component-of-term-splus*)
thus (*lookup* q v *when* u = $t \oplus v$) = *0* **by** *simp*
next
fix v
assume $v \in \{x \in$ *keys* q. *component-of-term* x = *component-of-term* $u\}$
hence *eq*[*symmetric*]: *component-of-term* v = *component-of-term* u **by** *simp*
have $u = t \oplus v \longleftrightarrow$ *pp-of-term* u = t + *pp-of-term* v
proof
assume *pp-of-term* u = t + *pp-of-term* v
hence *pp-of-term* u = *pp-of-term* ($t \oplus v$) **by** (*simp only*: *pp-of-term-splus*)
moreover have *component-of-term* u = *component-of-term* ($t \oplus v$)
by (*simp only*: *eq component-of-term-splus*)
ultimately show $u = t \oplus v$ **by** (*metis term-of-pair-pair*)

qed (*simp add*: *pp-of-term-splus*)
thus *?f t* (*pp-of-term v*) = (*lookup q v when u* = *t* ⊕ *v*)
by (*simp add*: *lookup-proj-poly eq term-of-pair-pair*)
qed *auto*
finally show *lookup p t* ∗ (*Sum-any* (*?f t*)) = *lookup p t* ∗ (∑ *v*∈*keys q*. *lookup q v when u* = *t* ⊕ *v*)
by (*simp only*:)
qed
finally show *?thesis* **.**
qed

lemma *proj-mult-scalar* [*term-simps*]: *proj-poly k* (*p* ⊙ *q*) = *p* ∗ (*proj-poly k q*)
unfolding *mult-scalar-def* **by** (*rule proj-lift-poly-fun-homogenous*, *simp*)

lemma *mult-scalar-zero-left* [*simp*]: *0* ⊙ *p* = *0*
by (*rule poly-mapping-eqI-proj*, *simp add*: *term-simps*)

lemma *mult-scalar-zero-right* [*simp*]: *p* ⊙ *0* = *0*
by (*rule poly-mapping-eqI-proj*, *simp add*: *term-simps*)

lemma *mult-scalar-one* [*simp*]: (*1*::- $\Rightarrow_0$ *'b*::*semiring-1*) ⊙ *p* = *p*
by (*rule poly-mapping-eqI-proj*, *simp add*: *term-simps*)

lemma *mult-scalar-assoc* [*ac-simps*]: (*p* ∗ *q*) ⊙ *r* = *p* ⊙ (*q* ⊙ *r*)
by (*rule poly-mapping-eqI-proj*, *simp add*: *ac-simps term-simps*)

lemma *mult-scalar-distrib-right* [*algebra-simps*]: (*p* + *q*) ⊙ *r* = *p* ⊙ *r* + *q* ⊙ *r*
by (*rule poly-mapping-eqI-proj*, *simp add*: *algebra-simps proj-plus term-simps*)

lemma *mult-scalar-distrib-left* [*algebra-simps*]: *r* ⊙ (*p* + *q*) = *r* ⊙ *p* + *r* ⊙ *q*
by (*rule poly-mapping-eqI-proj*, *simp add*: *algebra-simps proj-plus term-simps*)

lemma *mult-scalar-minus-mult-left* [*simp*]: (− *p*) ⊙ *q* = − (*p* ⊙ *q*)
by (*rule sym*, *rule minus-unique*, *simp add*: *mult-scalar-distrib-right*[*symmetric*])

lemma *mult-scalar-minus-mult-right* [*simp*]: *p* ⊙ (− *q*) = − (*p* ⊙ *q*)
by (*rule sym*, *rule minus-unique*, *simp add*: *mult-scalar-distrib-left* [*symmetric*])

lemma *minus-mult-scalar-minus* [*simp*]: (− *p*) ⊙ (− *q*) = *p* ⊙ *q*
by *simp*

lemma *minus-mult-scalar-commute*: (− *p*) ⊙ *q* = *p* ⊙ (− *q*)
by *simp*

lemma *mult-scalar-right-diff-distrib* [*algebra-simps*]: *r* ⊙ (*p* − *q*) = *r* ⊙ *p* − *r* ⊙ *q*
for *r*::- $\Rightarrow_0$ *'b*::*ring*
using *mult-scalar-distrib-left* [*of r p* − *q*] **by** *simp*

lemma *mult-scalar-left-diff-distrib* [*algebra-simps*]: $(p - q) \odot r = p \odot r - q \odot r$
  for p::- $\Rightarrow_0$ $'b$::*ring*
  using *mult-scalar-distrib-right* [*of* $p - q$ r] **by** *simp*

lemma *sum-mult-scalar-distrib-left*: $r \odot$ (*sum* f A) = ($\sum a \in A$. $r \odot f$ a)
  by (*induct* A *rule*: *infinite-finite-induct*, *simp-all add*: *algebra-simps*)

lemma *sum-mult-scalar-distrib-right*: (*sum* f A) $\odot$ v = ($\sum a \in A$. f a $\odot$ v)
  by (*induct* A *rule*: *infinite-finite-induct*, *simp-all add*: *algebra-simps*)

lemma *mult-scalar-monomial-monomial*: (*monomial* c t) $\odot$ (*monomial* d v) = *monomial* ($c * d$) ($t \oplus v$)
  by (*rule poly-mapping-eqI-proj*, *simp add*: *proj-monomial mult-single when-def term-simps*)

lemma *mult-scalar-monomial*: (*monomial* c t) $\odot$ p = *monom-mult* c t p
  by (*rule poly-mapping-eqI-proj*, *rule poly-mapping-eqI*,
    auto simp add: *lookup-times-monomial-left lookup-proj-poly lookup-monom-mult when-def*
      adds-pp-def sminus-def term-simps)

lemma *mult-scalar-rec-left*: $p \odot q$ = *monom-mult* (*lookup* p t) t q + (*except* p $\{t\}$) $\odot$ q
proof −
  from *plus-except*[*of* p t] **have** $p \odot q$ = (*monomial* (*lookup* p t) t + *except* p $\{t\}$) $\odot$ q **by** *simp*
  also have ... = *monomial* (*lookup* p t) $t \odot q$ + *except* p $\{t\}$ $\odot$ q **by** (*simp only*: *algebra-simps*)
  finally show *?thesis* **by** (*simp only*: *mult-scalar-monomial*)
qed

lemma *mult-scalar-rec-right*: $p \odot q = p \odot$ *monomial* (*lookup* q v) v + $p \odot$ *except* q $\{v\}$
proof −
  have $p \odot$ *monomial* (*lookup* q v) v + $p \odot$ *except* q $\{v\}$ = $p \odot$ (*monomial* (*lookup* q v) v + *except* q $\{v\}$)
    by (*simp only*: *mult-scalar-distrib-left*)
  also have ... = $p \odot q$ **by** (*simp only*: *plus-except*[*of* q v, *symmetric*])
  finally show *?thesis* **by** *simp*
qed

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 −
  from *assms* **have** $0 \neq$ *lookup* ($p \odot q$) v **by** (*simp add*: *in-keys-iff*)
  also have *lookup* ($p \odot q$) v = *lookup* (p * (*proj-poly* (*component-of-term* v) q)) (*pp-of-term* v)
    by (*fact lookup-mult-scalar*)

finally have *pp-of-term v* ∈ *keys* (*p* ∗ *proj-poly* (*component-of-term v*) *q*) **by** (*simp add*: *in-keys-iff*)
from *this keys-mult* **have** *pp-of-term v* ∈ {*t* + *s* |*t s*. *t* ∈ *keys p* ∧ *s* ∈ *keys* (*proj-poly* (*component-of-term v*) *q*)} **..**
then obtain *t s* **where** *t* ∈ *keys p* **and** ∗: *s* ∈ *keys* (*proj-poly* (*component-of-term v*) *q*)
and *eq*: *pp-of-term v* = *t* + *s* **by** *fastforce*
note *this*(*1*)
moreover from ∗ **have** *term-of-pair* (*s*, *component-of-term v*) ∈ *keys q*
by (*simp only*: *in-keys-iff lookup-proj-poly not-False-eq-True*)
moreover have *v* = *t* ⊕ *term-of-pair* (*s*, *component-of-term v*)
by (*simp add*: *splus-def eq*[*symmetric*] *term-simps*)
ultimately show *?thesis* **..**
qed

lemma *lookup-mult-scalar-monomial-right*:
lookup (*p* ⊙ *monomial c v*) *u* = (*lookup p* (*pp-of-term u* − *pp-of-term v*) ∗ *c when* *v adds*$_t$ *u*)
proof −
have *eq1*: *lookup* (*p* ∗ (*monomial c* (*pp-of-term v*) *when component-of-term v* = *component-of-term u*)) (*pp-of-term u*) =
(*lookup* (*p* ∗ (*monomial c* (*pp-of-term v*))) (*pp-of-term u*) *when component-of-term v* = *component-of-term u*)
by (*rule when-distrib*, *simp*)
show *?thesis*
by (*simp add*: *lookup-mult-scalar eq1 proj-monomial lookup-times-monomial-right when-when*
adds-term-def lookup-proj-poly conj-commute)
qed

lemma *lookup-mult-scalar-monomial-right-plus*: *lookup* (*p* ⊙ *monomial c v*) (*t* ⊕ *v*) = *lookup p t* ∗ *c*
by (*simp add*: *lookup-mult-scalar-monomial-right term-simps*)

lemma *keys-mult-scalar-monomial-right-subset*: *keys* (*p* ⊙ *monomial c v*) ⊆ (*λt*. *t* ⊕ *v*) ‘ *keys p*
proof
fix *u*
assume *u* ∈ *keys* (*p* ⊙ *monomial c v*)
then obtain *t w* **where** *t* ∈ *keys p* **and** *w* ∈ *keys* (*monomial c v*) **and** *u* = *t* ⊕ *w*
by (*rule in-keys-mult-scalarE*)
from *this*(*2*) **have** *w* = *v* **by** (*metis empty-iff insert-iff keys-single*)
from ‹*t* ∈ *keys p*› **show** *u* ∈ (*λt*. *t* ⊕ *v*) ‘ *keys p* **unfolding** ‹*u* = *t* ⊕ *w*› ‹*w* = *v*› **by** *fastforce*
qed

lemma *keys-mult-scalar-monomial-right*:
assumes *c* ≠ (*0*::′*b*::*semiring-no-zero-divisors*)

shows *keys* (*p* ⊙ *monomial c v*) = (λ*t*. *t* ⊕ *v*) ' *keys p*
proof
show (λ*t*. *t* ⊕ *v*) ' *keys p* ⊆ *keys* (*p* ⊙ *monomial c v*)
proof
fix *u*
assume *u* ∈ (λ*t*. *t* ⊕ *v*) ' *keys p*
then obtain *t* **where** *t* ∈ *keys p* **and** *u* = *t* ⊕ *v* **..**
have *lookup* (*p* ⊙ *monomial c v*) (*t* ⊕ *v*) = *lookup p t* ∗ *c*
by (*fact lookup-mult-scalar-monomial-right-plus*)
also from ‹*t* ∈ *keys p*› *assms* **have** ... ≠ *0* **by** (*simp add*: *in-keys-iff*)
finally show *u* ∈ *keys* (*p* ⊙ *monomial c v*) **by** (*simp add*: *in-keys-iff* ‹*u* = *t* ⊕ *v*›)
qed
qed (*fact keys-mult-scalar-monomial-right-subset*)

end

9.8 Sums and Products

lemma *sum-poly-mapping-eq-zeroI*:
assumes *p* ' *A* ⊆ {*0*}
shows *sum p A* = (*0*::(- ⇒$_0$ '*b*::*comm-monoid-add*))
proof (*rule ccontr*)
assume *sum p A* ≠ *0*
then obtain *a* **where** *a* ∈ *A* **and** *p a* ≠ *0*
by (*rule comm-monoid-add-class.sum.not-neutral-contains-not-neutral*)
with *assms* **show** *False* **by** *auto*
qed

lemma *lookup-sum-list*: *lookup* (*sum-list ps*) *a* = *sum-list* (*map* (λ*p*. *lookup p a*) *ps*)
proof (*induct ps*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons p ps*)
thus *?case* **by** (*simp add*: *lookup-add*)
qed

Legacy:

lemmas *keys-sum-subset* = *Poly-Mapping.keys-sum*

lemma *keys-sum-list-subset*: *keys* (*sum-list ps*) ⊆ *Keys* (*set ps*)
proof (*induct ps*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons p ps*)
have *keys* (*sum-list* (*p* # *ps*)) = *keys* (*p* + *sum-list ps*) **by** *simp*
also have ... ⊆ *keys p* ∪ *keys* (*sum-list ps*) **by** (*fact Poly-Mapping.keys-add*)

also from *Cons* **have** ... ⊆ *keys p* ∪ *Keys* (*set ps*) **by** *blast*
also have ... = *Keys* (*set* (*p* # *ps*)) **by** (*simp add*: *Keys-insert*)
finally show *?case* **.**
qed

lemma *keys-sum*:
assumes *finite A* **and** ⋀*a1 a2*. *a1* ∈ *A* ⟹ *a2* ∈ *A* ⟹ *a1* ≠ *a2* ⟹ *keys* (*f a1*) ∩ *keys* (*f a2*) = {}
shows *keys* (*sum f A*) = (⋃*a*∈*A*. *keys* (*f a*))
using *assms*
proof (*induct A*)
case *empty*
show *?case* **by** *simp*
next
case (*insert a A*)
have *IH*: *keys* (*sum f A*) = (⋃*i*∈*A*. *keys* (*f i*)) **by** (*rule insert*(*3*), *rule insert.prems*, *simp-all*)
have *keys* (*sum f* (*insert a A*)) = *keys* (*f a*) ∪ *keys* (*sum f A*)
proof (*simp only*: *comm-monoid-add-class.sum.insert*[*OF insert*(*1*) *insert*(*2*)], *rule keys-add*[*symmetric*])
have *keys* (*f a*) ∩ *keys* (*sum f A*) = (⋃*i*∈*A*. *keys* (*f a*) ∩ *keys* (*f i*))
by (*simp only*: *IH Int-UN-distrib*)
also have ... = {}
proof −
have *i* ∈ *A* ⟹ *keys* (*f a*) ∩ *keys* (*f i*) = {} **for** *i*
proof (*rule insert.prems*)
assume *i* ∈ *A*
with *insert*(*2*) **show** *a* ≠ *i* **by** *blast*
qed *simp-all*
thus *?thesis* **by** *simp*
qed
finally show *keys* (*f a*) ∩ *keys* (*sum f A*) = {} **.**
qed
also have ... = (⋃*a*∈*insert a A*. *keys* (*f a*)) **by** (*simp add*: *IH*)
finally show *?case* **.**
qed

lemma *poly-mapping-sum-monomials*: (∑*a*∈*keys p*. *monomial* (*lookup p a*) *a*) = *p*
proof (*induct p rule*: *poly-mapping-plus-induct*)
case *1*
show *?case* **by** *simp*
next
case *step*: (*2 p c t*)
from *step*(*2*) **have** *lookup p t* = *0* **by** (*simp add*: *in-keys-iff*)
have ∗: *keys* (*monomial c t* + *p*) = *insert t* (*keys p*)
proof −
from *step*(*1*) **have** *a*: *keys* (*monomial c t*) = {*t*} **by** *simp*
with *step*(*2*) **have** *keys* (*monomial c t*) ∩ *keys p* = {} **by** *simp*

hence *keys* (*monomial c t* + *p*) = {*t*} ∪ *keys p* **by** (*simp only*: *a keys-plus-eqI*)
thus *?thesis* **by** *simp*
qed
have ∗∗: (∑ *ta*∈*keys p*. *monomial* ((*c when t* = *ta*) + *lookup p ta*) *ta*) = (∑ *ta*∈*keys p*. *monomial* (*lookup p ta*) *ta*)
proof (*rule comm-monoid-add-class.sum.cong*, *rule refl*)
fix *s*
assume *s* ∈ *keys p*
with *step*(*2*) **have** *t* ≠ *s* **by** *auto*
thus *monomial* ((*c when t* = *s*) + *lookup p s*) *s* = *monomial* (*lookup p s*) *s* **by** *simp*
qed
show *?case* **by** (*simp only*: ∗ *comm-monoid-add-class.sum.insert*[*OF finite-keys step*(*2*)],
simp add: *lookup-add lookup-single* ‹*lookup p t* = *0*› ∗∗ *step*(*3*))
qed

lemma *monomial-sum*: *monomial* (*sum f C*) *a* = (∑ *c*∈*C*. *monomial* (*f c*) *a*)
by (*rule fun-sum-commute*, *simp-all add*: *single-add*)

lemma *monomial-Sum-any*:
assumes *finite* {*c*. *f c* ≠ *0*}
shows *monomial* (*Sum-any f*) *a* = (∑ *c*. *monomial* (*f c*) *a*)
proof −
have {*c*. *monomial* (*f c*) *a* ≠ *0*} ⊆ {*c*. *f c* ≠ *0*} **by** (*rule*, *auto*)
with *assms* **show** *?thesis*
by (*simp add*: *Groups-Big-Fun.comm-monoid-add-class.Sum-any.expand-superset monomial-sum*)
qed

context *term-powerprod*
begin

lemma *proj-sum*: *proj-poly k* (*sum f A*) = (∑ *a*∈*A*. *proj-poly k* (*f a*))
using *proj-zero proj-plus* **by** (*rule fun-sum-commute*)

lemma *proj-sum-list*: *proj-poly k* (*sum-list xs*) = *sum-list* (*map* (*proj-poly k*) *xs*)
using *proj-zero proj-plus* **by** (*rule fun-sum-list-commute*)

lemma *mult-scalar-sum-monomials*: *q* ⊙ *p* = (∑ *t*∈*keys q*. *monom-mult* (*lookup q t*) *t p*)
by (*rule poly-mapping-eqI-proj*, *simp add*: *proj-sum mult-scalar-monomial*[*symmetric*]
sum-distrib-right[*symmetric*] *poly-mapping-sum-monomials term-simps*)

lemma *fun-mult-scalar-commute*:
assumes *f 0* = *0* **and** ⋀*x y*. *f* (*x* + *y*) = *f x* + *f y*
and ⋀*c t*. *f* (*monom-mult c t p*) = *monom-mult c t* (*f p*)
shows *f* (*q* ⊙ *p*) = *q* ⊙ (*f p*)
by (*simp add*: *mult-scalar-sum-monomials assms*(*3*)[*symmetric*], *rule fun-sum-commute*,

fact+)

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 $f\ (q \odot p) = q \odot (f\ (p::'t \Rightarrow_0 'b::\{semiring\text{-}0,cancel\text{-}comm\text{-}monoid\text{-}add\}))$
by (*simp add*: *mult-scalar-sum-monomials assms*(*2*)[*symmetric*], *rule fun-sum-commute-canc*, *fact*)

lemma *monom-mult-sum-left*: $monom\text{-}mult\ (sum\ f\ C)\ t\ p = (\sum c \in C.\ monom\text{-}mult\ (f\ c)\ t\ p)$
by (*rule fun-sum-commute*, *simp-all add*: *monom-mult-dist-left*)

lemma *monom-mult-sum-right*: $monom\text{-}mult\ c\ t\ (sum\ f\ P) = (\sum p \in P.\ monom\text{-}mult\ c\ t\ (f\ p))$
by (*rule fun-sum-commute*, *simp-all add*: *monom-mult-dist-right*)

lemma *monom-mult-Sum-any-left*:
assumes $finite\ \{c.\ f\ c \neq 0\}$
shows $monom\text{-}mult\ (Sum\text{-}any\ f)\ t\ p = (\sum c.\ monom\text{-}mult\ (f\ c)\ t\ p)$
proof −
have $\{c.\ monom\text{-}mult\ (f\ c)\ t\ p \neq 0\} \subseteq \{c.\ f\ c \neq 0\}$ **by** (*rule*, *auto*)
with *assms* **show** *?thesis*
by (*simp add*: *Groups-Big-Fun.comm-monoid-add-class.Sum-any.expand-superset monom-mult-sum-left*)
qed

lemma *monom-mult-Sum-any-right*:
assumes $finite\ \{p.\ f\ p \neq 0\}$
shows $monom\text{-}mult\ c\ t\ (Sum\text{-}any\ f) = (\sum p.\ monom\text{-}mult\ c\ t\ (f\ p))$
proof −
have $\{p.\ monom\text{-}mult\ c\ t\ (f\ p) \neq 0\} \subseteq \{p.\ f\ p \neq 0\}$ **by** (*rule*, *auto*)
with *assms* **show** *?thesis*
by (*simp add*: *Groups-Big-Fun.comm-monoid-add-class.Sum-any.expand-superset monom-mult-sum-right*)
qed

lemma *monomial-prod-sum*: $monomial\ (prod\ c\ I)\ (sum\ a\ I) = (\prod i \in I.\ monomial\ (c\ i)\ (a\ i))$
proof (*cases finite I*)
case *True*
thus *?thesis*
proof (*induct I*)
case *empty*
show *?case* **by** *simp*
next
case (*insert i I*)
show *?case*
by (*simp only*: *comm-monoid-add-class.sum.insert*[*OF insert*(*1*) *insert*(*2*)]

comm-monoid-mult-class.prod.insert[*OF insert*(*1*) *insert*(*2*)] *insert*(*3*)
mult-single[*symmetric*])
qed
next
case *False*
thus *?thesis* **by** *simp*
qed

9.9 Submodules

sublocale *pmdl*: *module mult-scalar*
apply *standard*
subgoal by (*rule poly-mapping-eqI-proj*, *simp add*: *algebra-simps proj-plus*)
subgoal by (*rule poly-mapping-eqI-proj*, *simp add*: *algebra-simps proj-plus*)
subgoal by (*rule poly-mapping-eqI-proj*, *simp add*: *ac-simps*)
subgoal by (*rule poly-mapping-eqI-proj*, *simp*)
done

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 $p \in pmdl\ B$
shows *monom-mult c t p* $\in$ *pmdl B*
unfolding *mult-scalar-monomial*[*symmetric*] **using** *assms* **by** (*rule pmdl.span-scale*)

lemma *monom-mult-in-pmdl*: $b \in B \Longrightarrow$ *monom-mult c t b* $\in$ *pmdl B*
by (*intro pmdl-closed-monom-mult pmdl.span-base*)

lemma *pmdl-induct* [*consumes 1*, *case-names module-0 module-plus*]:
assumes $p \in pmdl\ B$ **and** *P 0*
and $\bigwedge a\ p\ c\ t.\ a \in pmdl\ B \Longrightarrow P\ a \Longrightarrow p \in B \Longrightarrow c \neq 0 \Longrightarrow P\ (a +$
monom-mult c t p)
shows *P p*
using *assms*(*1*)
proof (*induct p rule*: *pmdl.span-induct′*)
case *base*
from *assms*(*2*) **show** *?case* **.**
next
case (*step a q b*)
from *this*(*1*) *this*(*2*) **show** *?case*
proof (*induct q arbitrary*: *a rule*: *poly-mapping-except-induct*)
case *1*
thus *?case* **by** *simp*

next
case *step*: (*2 q0 t*)
from *this*(*4*) *step*(*5*) ‹*b* ∈ *B*› **have** *P* (*a* + *monomial* (*lookup q0 t*) *t* ⊙ *b*)
unfolding *mult-scalar-monomial*
proof (*rule assms*(*3*))
from *step*(*2*) **show** *lookup q0 t* ≠ *0* **by** (*simp add*: *in-keys-iff*)
qed
with - **have** *P* ((*a* + *monomial* (*lookup q0 t*) *t* ⊙ *b*) + *except q0* {*t*} ⊙ *b*)
proof (*rule step*(*3*))
from ‹*b* ∈ *B*› **have** *b* ∈ *pmdl B* **by** (*rule pmdl.span-base*)
hence *monomial* (*lookup q0 t*) *t* ⊙ *b* ∈ *pmdl B* **by** (*rule pmdl.span-scale*)
with *step*(*4*) **show** *a* + *monomial* (*lookup q0 t*) *t* ⊙ *b* ∈ *pmdl B* **by** (*rule pmdl.span-add*)
qed
hence *P* (*a* + (*monomial* (*lookup q0 t*) *t* + *except q0* {*t*}) ⊙ *b*) **by** (*simp add*: *algebra-simps*)
thus *?case* **by** (*simp only*: *plus-except*[*of q0 t*, *symmetric*])
qed
qed

lemma *components-pmdl*: *component-of-term* ‘ *Keys* (*pmdl B*) = *component-of-term* ‘ *Keys B*
proof
show *component-of-term* ‘ *Keys* (*pmdl B*) ⊆ *component-of-term* ‘ *Keys B*
proof
fix *k*
assume *k* ∈ *component-of-term* ‘ *Keys* (*pmdl B*)
then obtain *v* **where** *v* ∈ *Keys* (*pmdl B*) **and** *k* = *component-of-term v* **..**
from *this*(*1*) **obtain** *b* **where** *b* ∈ *pmdl B* **and** *v* ∈ *keys b* **by** (*rule in-KeysE*)
thus *k* ∈ *component-of-term* ‘ *Keys B*
proof (*induct b rule*: *pmdl-induct*)
case *module-0*
thus *?case* **by** *simp*
next
case *ind*: (*module-plus a p c t*)
from *ind.prems Poly-Mapping.keys-add* **have** *v* ∈ *keys a* ∪ *keys* (*monom-mult c t p*) **..**
thus *?case*
proof
assume *v* ∈ *keys a*
thus *?thesis* **by** (*rule ind.hyps*(*2*))
next
assume *v* ∈ *keys* (*monom-mult c t p*)
from *this keys-monom-mult-subset* **have** *v* ∈ (⊕) *t* ‘ *keys p* **..**
then obtain *u* **where** *u* ∈ *keys p* **and** *v* = *t* ⊕ *u* **..**
have *k* = *component-of-term u* **by** (*simp add*: ‹*k* = *component-of-term v*› ‹*v* = *t* ⊕ *u*› *term-simps*)
moreover from ‹*u* ∈ *keys p*› *ind.hyps*(*3*) **have** *u* ∈ *Keys B* **by** (*rule in-KeysI*)

ultimately show *?thesis* **..**
qed
qed
qed
next
show *component-of-term ‘ Keys B* ⊆ *component-of-term ‘ Keys* (*pmdl B*)
by (*rule image-mono*, *rule Keys-mono*, *fact pmdl.span-superset*)
qed

lemma *pmdl-idI*:
assumes *0* ∈ *B* **and** ⋀*b1 b2*. *b1* ∈ *B* ⟹ *b2* ∈ *B* ⟹ *b1* + *b2* ∈ *B*
and ⋀*c t b*. *b* ∈ *B* ⟹ *monom-mult c t b* ∈ *B*
shows *pmdl B* = *B*
proof
show *pmdl B* ⊆ *B*
proof
fix *p*
assume *p* ∈ *pmdl B*
thus *p* ∈ *B*
proof (*induct p rule*: *pmdl-induct*)
case *module-0*
show *?case* **by** (*fact assms*(*1*))
next
case *step*: (*module-plus a b c t*)
from *step*(*2*) **show** *?case*
proof (*rule assms*(*2*))
from *step*(*3*) **show** *monom-mult c t b* ∈ *B* **by** (*rule assms*(*3*))
qed
qed
qed
qed (*fact pmdl.span-superset*)

definition *full-pmdl* :: *'k set* ⇒ (*'t* $\Rightarrow_0$ *'b*::*zero*) *set*
where *full-pmdl K* = {*p*. *component-of-term ‘ keys p* ⊆ *K*}

definition *is-full-pmdl* :: (*'t* $\Rightarrow_0$ *'b*::*comm-ring-1*) *set* ⇒ *bool*
where *is-full-pmdl B* ⟷ (∀ *p*. *component-of-term ‘ keys p* ⊆ *component-of-term ‘ Keys B* ⟶ *p* ∈ *pmdl B*)

lemma *full-pmdl-iff*: *p* ∈ *full-pmdl K* ⟷ *component-of-term ‘ keys p* ⊆ *K*
by (*simp add*: *full-pmdl-def*)

lemma *full-pmdlI*:
assumes ⋀*v*. *v* ∈ *keys p* ⟹ *component-of-term v* ∈ *K*
shows *p* ∈ *full-pmdl K*
using *assms* **by** (*auto simp add*: *full-pmdl-iff*)

lemma *full-pmdlD*:
assumes *p* ∈ *full-pmdl K* **and** *v* ∈ *keys p*

shows *component-of-term v* ∈ *K*
using *assms* **by** (*auto simp add*: *full-pmdl-iff*)

lemma *full-pmdl-empty*: *full-pmdl* {} = {*0*}
by (*simp add*: *full-pmdl-def*)

lemma *full-pmdl-UNIV*: *full-pmdl UNIV* = *UNIV*
by (*simp add*: *full-pmdl-def*)

lemma *zero-in-full-pmdl*: *0* ∈ *full-pmdl K*
by (*simp add*: *full-pmdl-iff*)

lemma *full-pmdl-closed-plus*:
assumes *p* ∈ *full-pmdl K* **and** *q* ∈ *full-pmdl K*
shows *p* + *q* ∈ *full-pmdl K*
proof (*rule full-pmdlI*)
fix *v*
assume *v* ∈ *keys* (*p* + *q*)
also have ... ⊆ *keys p* ∪ *keys q* **by** (*fact Poly-Mapping.keys-add*)
finally show *component-of-term v* ∈ *K*
proof
assume *v* ∈ *keys p*
with *assms*(*1*) **show** *?thesis* **by** (*rule full-pmdlD*)
next
assume *v* ∈ *keys q*
with *assms*(*2*) **show** *?thesis* **by** (*rule full-pmdlD*)
qed
qed

lemma *full-pmdl-closed-monom-mult*:
assumes *p* ∈ *full-pmdl K*
shows *monom-mult c t p* ∈ *full-pmdl K*
proof (*rule full-pmdlI*)
fix *v*
assume *v* ∈ *keys* (*monom-mult c t p*)
also have ... ⊆ (⊕) *t* ' *keys p* **by** (*fact keys-monom-mult-subset*)
finally obtain *u* **where** *u* ∈ *keys p* **and** *v*: *v* = *t* ⊕ *u* **..**
have *component-of-term v* = *component-of-term u* **by** (*simp add*: *v term-simps*)
also from *assms* ‹*u* ∈ *keys p*› **have** ... ∈ *K* **by** (*rule full-pmdlD*)
finally show *component-of-term v* ∈ *K* **.**
qed

lemma *pmdl-full-pmdl*: *pmdl* (*full-pmdl K*) = *full-pmdl K*
using *zero-in-full-pmdl full-pmdl-closed-plus full-pmdl-closed-monom-mult* **by** (*rule pmdl-idI*)

lemma *components-full-pmdl-subset*:
component-of-term ' *Keys* ((*full-pmdl K*)::(*'t* $\Rightarrow_0$ *'b*::*zero*) *set*) ⊆ *K* (**is** *?l* ⊆ -)
proof

let *?M* = (*full-pmdl K*)::($'t \Rightarrow_0 'b$) *set*
fix *k*
assume *k* ∈ *?l*
then obtain *v* **where** *v* ∈ *Keys ?M* **and** *k*: *k* = *component-of-term v* **..**
from *this*(*1*) **obtain** *p* **where** *p* ∈ *?M* **and** *v* ∈ *keys p* **by** (*rule in-KeysE*)
thus *k* ∈ *K* **unfolding** *k* **by** (*rule full-pmdlD*)
qed

lemma *components-full-pmdl*:
component-of-term ‘ *Keys* ((*full-pmdl K*)::($'t \Rightarrow_0 'b$::*zero-neq-one*) *set*) = *K* (**is** *?l* = -)
proof
let *?M* = (*full-pmdl K*)::($'t \Rightarrow_0 'b$) *set*
show *K* ⊆ *?l*
proof
fix *k*
assume *k* ∈ *K*
hence *monomial 1* (*term-of-pair* (*0*, *k*)) ∈ *?M* **by** (*simp add*: *full-pmdl-iff term-simps*)
hence *keys* (*monomial* (*1*::*'b*) (*term-of-pair* (*0*, *k*))) ⊆ *Keys ?M* **by** (*rule keys-subset-Keys*)
hence *term-of-pair* (*0*, *k*) ∈ *Keys ?M* **by** *simp*
hence *component-of-term* (*term-of-pair* (*0*, *k*)) ∈ *component-of-term* ‘ *Keys ?M* **by** (*rule imageI*)
thus *k* ∈ *?l* **by** (*simp only*: *component-of-term-of-pair*)
qed
qed (*fact components-full-pmdl-subset*)

lemma *is-full-pmdlI*:
assumes ⋀*p*. *component-of-term* ‘ *keys p* ⊆ *component-of-term* ‘ *Keys B* ⟹ *p* ∈ *pmdl B*
shows *is-full-pmdl B*
unfolding *is-full-pmdl-def* **using** *assms* **by** *blast*

lemma *is-full-pmdlD*:
assumes *is-full-pmdl B* **and** *component-of-term* ‘ *keys p* ⊆ *component-of-term* ‘ *Keys B*
shows *p* ∈ *pmdl B*
using *assms* **unfolding** *is-full-pmdl-def* **by** *blast*

lemma *is-full-pmdl-alt*: *is-full-pmdl B* ⟷ *pmdl B* = *full-pmdl* (*component-of-term* ‘ *Keys B*)
proof −
have *b* ∈ *pmdl B* ⟹ *v* ∈ *keys b* ⟹ *component-of-term v* ∈ *component-of-term* ‘ *Keys B* **for** *b v*
by (*metis components-pmdl image-eqI in-KeysI*)
thus *?thesis* **by** (*auto simp add*: *is-full-pmdl-def full-pmdl-def*)
qed

lemma *is-full-pmdl-pmdl*: *is-full-pmdl* (*pmdl B*) ⟷ *is-full-pmdl B*
by (*simp only*: *is-full-pmdl-def pmdl.span-span components-pmdl*)

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
fix *p*
assume *p* ∈ *pmdl B1*
from *assms*(*2*) **show** *p* ∈ *pmdl B2*
proof (*rule is-full-pmdlD*)
have *component-of-term* ' *keys p* ⊆ *component-of-term* ' *Keys* (*pmdl B1*)
by (*rule image-mono*, *rule keys-subset-Keys*, *fact*)
also have ... = *component-of-term* ' *Keys B1* **by** (*fact components-pmdl*)
finally show *component-of-term* ' *keys p* ⊆ *component-of-term* ' *Keys B2* **using** *assms*(*3*)
by (*rule subset-trans*)
qed
qed

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
have *component-of-term* ' *Keys B1* ⊆ *component-of-term* ' *Keys B2* **by** (*simp add*: *assms*(*3*))
with *assms*(*1*, *2*) **show** *pmdl B1* ⊆ *pmdl B2* **by** (*rule is-full-pmdl-subset*)
next
have *component-of-term* ' *Keys B2* ⊆ *component-of-term* ' *Keys B1* **by** (*simp add*: *assms*(*3*))
with *assms*(*2*, *1*) **show** *pmdl B2* ⊆ *pmdl B1* **by** (*rule is-full-pmdl-subset*)
qed

end

definition *map-scale* :: *'b* ⇒ (*'a* $\Rightarrow_0$ *'b*) ⇒ (*'a* $\Rightarrow_0$ *'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* · *p* corresponds to exponentiation; if it is interpreted as a (vector-) polynomial, then *c* · *p* corresponds to multiplication by scalar from the coefficient type.

lemma *lookup-map-scale* [*simp*]: *lookup* (*c* · *p*) = (λ*x*. *c* ∗ *lookup p x*)
by (*auto simp*: *map-scale-def map.rep-eq when-def*)

lemma *map-scale-single* [*simp*]: *k* · *Poly-Mapping.single x l* = *Poly-Mapping.single x* (*k* ∗ *l*)
by (*simp add*: *map-scale-def*)

lemma *map-scale-zero-left* [*simp*]: *0* · *t* = *0*
  by (*rule poly-mapping-eqI*) *simp*

lemma *map-scale-zero-right* [*simp*]: *k* · *0* = *0*
  by (*rule poly-mapping-eqI*) *simp*

lemma *map-scale-eq-0-iff*: *c* · *t* = *0* ⟷ ((*c*::-::*semiring-no-zero-divisors*) = *0* ∨ *t* = *0*)
  by (*metis aux lookup-map-scale mult-eq-0-iff*)

lemma *keys-map-scale-subset*: *keys* (*k* · *t*) ⊆ *keys t*
  by (*metis in-keys-iff lookup-map-scale mult-zero-right subsetI*)

lemma *keys-map-scale*: *keys* ((*k*::′*b*::*semiring-no-zero-divisors*) · *t*) = (*if k* = *0* *then* {} *else keys t*)
proof (*split if-split*, *intro conjI impI*)
  assume *k* = *0*
  thus *keys* (*k* · *t*) = {} **by** *simp*
next
  assume *k* ≠ *0*
  show *keys* (*k* · *t*) = *keys t*
  proof
   show *keys t* ⊆ *keys* (*k* · *t*) **by** *rule* (*simp add*: ‹*k* ≠ *0*› *flip*: *lookup-not-eq-zero-eq-in-keys*)
  qed (*fact keys-map-scale-subset*)
qed

lemma *map-scale-one-left* [*simp*]: (*1*::′*b*::{*mult-zero*,*monoid-mult*}) · *t* = *t*
  by (*rule poly-mapping-eqI*) *simp*

lemma *map-scale-assoc* [*ac-simps*]: *c* · *d* · *t* = (*c* ∗ *d*) · (*t*::- $\Rightarrow_0$ -::{*semigroup-mult*,*zero*})
  by (*rule poly-mapping-eqI*) (*simp add*: *ac-simps*)

lemma *map-scale-distrib-left* [*algebra-simps*]: (*k*::′*b*::*semiring-0*) · (*s* + *t*) = *k* · *s* + *k* · *t*
  by (*rule poly-mapping-eqI*) (*simp add*: *lookup-add distrib-left*)

lemma *map-scale-distrib-right* [*algebra-simps*]: (*k* + (*l*::′*b*::*semiring-0*)) · *t* = *k* · *t* + *l* · *t*
  by (*rule poly-mapping-eqI*) (*simp add*: *lookup-add distrib-right*)

lemma *map-scale-Suc*: (*Suc k*) · *t* = *k* · *t* + *t*
  by (*rule poly-mapping-eqI*) (*simp add*: *lookup-add distrib-right*)

lemma *map-scale-uminus-left*: (− *k*::′*b*::*ring*) · *p* = − (*k* · *p*)
  by (*rule poly-mapping-eqI*) *auto*

lemma *map-scale-uminus-right*: (*k*::′*b*::*ring*) · (− *p*) = − (*k* · *p*)
  by (*rule poly-mapping-eqI*) *auto*

lemma *map-scale-uminus-uminus* [*simp*]: (− *k*::′*b*::*ring*) · (− *p*) = *k* · *p*
by (*simp add*: *map-scale-uminus-left map-scale-uminus-right*)

lemma *map-scale-minus-distrib-left* [*algebra-simps*]:
(*k*::′*b*::*comm-semiring-1-cancel*) · (*p* − *q*) = *k* · *p* − *k* · *q*
by (*rule poly-mapping-eqI*) (*auto simp add*: *lookup-minus right-diff-distrib′*)

lemma *map-scale-minus-distrib-right* [*algebra-simps*]:
(*k* − (*l*::′*b*::*comm-semiring-1-cancel*)) · *f* = *k* · *f* − *l* · *f*
by (*rule poly-mapping-eqI*) (*auto simp add*: *lookup-minus left-diff-distrib′*)

lemma *map-scale-sum-distrib-left*: (*k*::′*b*::*semiring-0*) · (*sum f A*) = (∑ *a*∈*A*. *k* · *f a*)
by (*induct A rule*: *infinite-finite-induct*) (*simp-all add*: *map-scale-distrib-left*)

lemma *map-scale-sum-distrib-right*: (*sum* (*f*::- ⇒ ′*b*::*semiring-0*) *A*) · *p* = (∑ *a*∈*A*. *f a* · *p*)
by (*induct A rule*: *infinite-finite-induct*) (*simp-all add*: *map-scale-distrib-right*)

lemma *deg-pm-map-scale*: *deg-pm* (*k* · *t*) = (*k*::′*b*::*semiring-0*) ∗ *deg-pm t*
proof −
from *keys-map-scale-subset finite-keys* **have** *deg-pm* (*k* · *t*) = *sum* (*lookup* (*k* · *t*)) (*keys t*)
by (*rule deg-pm-superset*)
also have ... = *k* ∗ *sum* (*lookup t*) (*keys t*) **by** (*simp add*: *sum-distrib-left*)
also from *subset-refl finite-keys* **have** *sum* (*lookup t*) (*keys t*) = *deg-pm t*
by (*rule deg-pm-superset*[*symmetric*])
finally show *?thesis* **.**
qed

interpretation *phull*: *module map-scale*
apply *standard*
subgoal by (*fact map-scale-distrib-left*)
subgoal by (*fact map-scale-distrib-right*)
subgoal by (*fact map-scale-assoc*)
subgoal by (*fact map-scale-one-left*)
done

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 *λterm-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 · p = monom-mult c 0 p
 by (rule poly-mapping-eqI) (simp only: lookup-map-scale lookup-monom-mult-zero)

lemma map-scale-eq-mult-scalar: c · p = monomial c 0 ⊙ p
  by (simp only: map-scale-eq-monom-mult mult-scalar-monomial)

lemma phull-closed-mult-scalar: p ∈ phull B ⟹ monomial c 0 ⊙ p ∈ phull B
  unfolding map-scale-eq-mult-scalar[symmetric] by (rule phull.span-scale)

lemma mult-scalar-in-phull: b ∈ B ⟹ monomial c 0 ⊙ b ∈ phull B
  by (intro phull-closed-mult-scalar phull.span-base)

lemma phull-subset-module: phull B ⊆ pmdl B
proof
  fix p
  assume p ∈ phull B
  thus p ∈ pmdl B
  proof (induct p rule: phull.span-induct')
    case base
    show ?case by (fact pmdl.span-zero)
  next
    case (step a c p)
    from step(3) have p ∈ pmdl B by (rule pmdl.span-base)
   hence c · p ∈ pmdl B unfolding map-scale-eq-monom-mult by (rule pmdl-closed-monom-mult)
    with step(2) show ?case by (rule pmdl.span-add)
  qed
qed

lemma components-phull: component-of-term ‘ Keys (phull B) = component-of-term
‘ Keys B
proof
  have component-of-term ‘ Keys (phull B) ⊆ component-of-term ‘ Keys (pmdl B)
    by (rule image-mono, rule Keys-mono, fact phull-subset-module)
  also have ... = component-of-term ‘ Keys B by (fact components-pmdl)
  finally show component-of-term ‘ Keys (phull B) ⊆ component-of-term ‘ Keys
B .
next
  show component-of-term ‘ Keys B ⊆ component-of-term ‘ Keys (phull B)
    by (rule image-mono, rule Keys-mono, fact phull.span-superset)
qed

end
```

9.10 Interpretations

9.10.1 Isomorphism between 'a and 'a × unit

definition *to-pair-unit* :: *'a* ⇒ (*'a* × *unit*)
  where *to-pair-unit x* = (*x*, ())

lemma *fst-to-pair-unit*: *fst* (*to-pair-unit x*) = *x*
  by (*simp add*: *to-pair-unit-def*)

lemma *to-pair-unit-fst*: *to-pair-unit* (*fst x*) = (*x*::- × *unit*)
  by (*metis* (*full-types*) *old.unit.exhaust prod.collapse to-pair-unit-def*)

interpretation *punit*: *term-powerprod to-pair-unit fst*
  apply *standard*
  subgoal by (*fact fst-to-pair-unit*)
  subgoal by (*fact to-pair-unit-fst*)
  done

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*)
  by (*rule*, *simp add*: *punit.pp-of-term-def punit.term-pair*)

lemma *punit-component-of-term* [*simp*]: *punit.component-of-term* = (λ-. ())
  by (*rule*, *simp add*: *punit.component-of-term-def*)

lemma *punit-splus* [*simp*]: *punit.splus* = (+)
  by (*rule*, *rule*, *simp add*: *punit.splus-def*)

lemma *punit-sminus* [*simp*]: *punit.sminus* = (−)
  by (*rule*, *rule*, *simp add*: *punit.sminus-def*)

lemma *punit-adds-pp* [*simp*]: *punit.adds-pp* = (*adds*)
  by (*rule*, *rule*, *simp add*: *punit.adds-pp-def*)

lemma *punit-adds-term* [*simp*]: *punit.adds-term* = (*adds*)
  by (*rule*, *rule*, *simp add*: *punit.adds-term-def*)

lemma *punit-proj-poly* [*simp*]: *punit.proj-poly* = (λ-. *id*)
  by (*rule*, *rule*, *rule poly-mapping-eqI*, *simp add*: *punit.lookup-proj-poly*)

lemma *punit-mult-vec* [*simp*]: *punit.mult-vec* = (∗)
  by (*rule*, *rule*, *rule poly-mapping-eqI*, *simp add*: *punit.lookup-mult-vec*)

lemma *punit-mult-scalar* [*simp*]: *punit.mult-scalar* = (∗)
  by (*rule*, *rule*, *rule poly-mapping-eqI*, *simp add*: *punit.lookup-mult-scalar*)

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*)
by (*metis mult-scalar-monomial proj-mult-scalar punit.mult-scalar-monomial punit-mult-scalar*)

lemma *mult-scalar-monom-mult*: (*punit.monom-mult c t p*) ⊙ *q* = *monom-mult c t* (*p* ⊙ *q*)
by (*simp add*: *punit.mult-scalar-monomial*[*symmetric*] *mult-scalar-assoc mult-scalar-monomial*)

end

9.10.2 Interpretation of *term-powerprod* by *'a* × *'k*

interpretation *pprod*: *term-powerprod* (λ*x*::*'a*::*comm-powerprod* × *'k*::*linorder*. *x*) λ*x*. *x*
by (*standard*, *simp*)

lemma *pprod-pp-of-term* [*simp*]: *pprod.pp-of-term* = *fst*
by (*rule*, *simp add*: *pprod.pp-of-term-def*)

lemma *pprod-component-of-term* [*simp*]: *pprod.component-of-term* = *snd*
by (*rule*, *simp add*: *pprod.component-of-term-def*)

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* $\leq$ *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 by (*standard*, *simp add*: *the-min-nat-def*)
end

instantiation *unit* :: *the-min*
begin
definition *the-min-unit* = ()
instance by (*standard*, *simp add*: *the-min-unit-def*)
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* $\Rightarrow$ ('*a*::*comm-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*) +
assumes *splus-mono*: $v \preceq_t w \Longrightarrow t \oplus v \preceq_t t \oplus w$
assumes *ord-termI*: *pp-of-term v* $\preceq$ *pp-of-term w* $\Longrightarrow$ *component-of-term v* $\leq$ *component-of-term w* $\Longrightarrow$ $v \preceq_t w$
begin

abbreviation *ord-term-conv* (**infixl** ‹$\succeq_t$› *50*) **where** *ord-term-conv* $\equiv (\preceq_t)^{-1-1}$
abbreviation *ord-term-strict-conv* (**infixl** ‹$\succ_t$› *50*) **where** *ord-term-strict-conv* $\equiv (\prec_t)^{-1-1}$

The definition of *ordered-term* only covers TOP and POT orderings. These two types of orderings are the only interesting ones.

definition *min-term* $\equiv$ *term-of-pair* (*0*, *the-min*)

lemma *min-term-min*: *min-term* $\preceq_t$ *v*
proof (*rule ord-termI*)
show *pp-of-term min-term* $\preceq$ *pp-of-term v* **by** (*simp add*: *min-term-def zero-min term-simps*)
next
show *component-of-term min-term* $\leq$ *component-of-term v* **by** (*simp add*: *min-term-def the-min-min term-simps*)
qed

lemma *splus-mono-strict*:

assumes $v \prec_t w$
shows $t \oplus v \prec_t t \oplus w$
proof −
from *assms* **have** $v \preceq_t w$ **and** $v \neq w$ **by** *simp-all*
from *this*(*1*) **have** $t \oplus v \preceq_t t \oplus w$ **by** (*rule splus-mono*)
moreover from ‹$v \neq w$› **have** $t \oplus v \neq t \oplus w$ **by** (*simp add*: *term-simps*)
ultimately show *?thesis* **using** *ord-term-lin.antisym-conv1* **by** *blast*
qed

lemma *splus-mono-left*:
assumes $s \preceq t$
shows $s \oplus v \preceq_t t \oplus v$
proof (*rule ord-termI*, *simp-all add*: *term-simps*)
from *assms* **show** s + *pp-of-term* $v \preceq t$ + *pp-of-term* v **by** (*rule plus-monotone*)
qed

lemma *splus-mono-strict-left*:
assumes $s \prec t$
shows $s \oplus v \prec_t t \oplus v$
proof −
from *assms* **have** $s \preceq t$ **and** $s \neq t$ **by** *simp-all*
from *this*(*1*) **have** $s \oplus v \preceq_t t \oplus v$ **by** (*rule splus-mono-left*)
moreover from ‹$s \neq t$› **have** $s \oplus v \neq t \oplus v$ **by** (*simp add*: *term-simps*)
ultimately show *?thesis* **using** *ord-term-lin.antisym-conv1* **by** *blast*
qed

lemma *ord-term-canc*:
assumes $t \oplus v \preceq_t t \oplus w$
shows $v \preceq_t w$
proof (*rule ccontr*)
assume $\neg\ v \preceq_t w$
hence $w \prec_t v$ **by** *simp*
hence $t \oplus w \prec_t t \oplus v$ **by** (*rule splus-mono-strict*)
with *assms* **show** *False* **by** *simp*
qed

lemma *ord-term-strict-canc*:
assumes $t \oplus v \prec_t t \oplus w$
shows $v \prec_t w$
proof (*rule ccontr*)
assume $\neg\ v \prec_t w$
hence $w \preceq_t v$ **by** *simp*
hence $t \oplus w \preceq_t t \oplus v$ **by** (*rule splus-mono*)
with *assms* **show** *False* **by** *simp*
qed

lemma *ord-term-canc-left*:
assumes $t \oplus v \preceq_t s \oplus v$
shows $t \preceq s$

proof (*rule ccontr*)
  assume $\neg\ t \preceq s$
  hence $s \prec t$ **by** *simp*
  hence $s \oplus v \prec_t t \oplus v$ **by** (*rule splus-mono-strict-left*)
  with *assms* **show** *False* **by** *simp*
qed

lemma *ord-term-strict-canc-left*:
  assumes $t \oplus v \prec_t s \oplus v$
  shows $t \prec s$
proof (*rule ccontr*)
  assume $\neg\ t \prec s$
  hence $s \preceq t$ **by** *simp*
  hence $s \oplus v \preceq_t t \oplus v$ **by** (*rule splus-mono-left*)
  with *assms* **show** *False* **by** *simp*
qed

lemma *ord-adds-term*:
  assumes *u* $adds_t$ *v*
  shows $u \preceq_t v$
proof −
  from *assms* **have** ∗: *component-of-term u* ≤ *component-of-term v* **and** *pp-of-term u adds pp-of-term v*
    by (*simp-all add*: *adds-term-def*)
  from *this*(*2*) **have** *pp-of-term u* ⪯ *pp-of-term v* **by** (*rule ord-adds*)
  from *this* ∗ **show** *?thesis* **by** (*rule ord-termI*)
qed

end

10.1 Interpretations

context *ordered-powerprod*
begin

10.1.1 Unit

sublocale *punit*: *ordered-term to-pair-unit fst* (⪯) (≺) (⪯) (≺)
  apply *standard*
  subgoal by (*simp*, *fact plus-monotone-left*)
  subgoal by (*simp only*: *punit-pp-of-term punit-component-of-term*)
  done

lemma *punit-min-term* [*simp*]: *punit.min-term* = *0*
  by (*simp add*: *punit.min-term-def*)

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*
  by (*simp add*: *lt-def*)

lemma *lc-zero* [*simp*]: *lc 0* = *0*
  by (*simp add*: *lc-def*)

lemma *lt-uminus* [*simp*]: *lt* (− *p*) = *lt p*
  by (*simp add*: *lt-def keys-uminus*)

lemma *lc-uminus* [*simp*]: *lc* (− *p*) = − *lc p*
  by (*simp add*: *lc-def*)

lemma *lt-alt*:
  assumes *p* ≠ *0*
  shows *lt p* = *ord-term-lin.Max* (*keys p*)
  using *assms* **unfolding** *lt-def* **by** *simp*

lemma *lt-max*:

assumes *lookup p v* $\neq$ *0*
shows *v* $\preceq_t$ *lt p*
proof –
from *assms* **have** *t-in*: *v* $\in$ *keys p* **by** (*simp add*: *in-keys-iff*)
hence *keys p* $\neq$ {} **by** *auto*
hence *p* $\neq$ *0* **using** *keys-zero* **by** *blast*
from *lt-alt*[*OF this*] *ord-term-lin.Max-ge*[*OF finite-keys t-in*] **show** *?thesis* **by** *simp*
qed

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 –
from *assms*(*1*) **have** *v* $\in$ *keys p* **by** (*simp add*: *in-keys-iff*)
hence *keys p* $\neq$ {} **by** *auto*
hence *p* $\neq$ *0*
using *keys-zero* **by** *blast*
have *u* $\preceq_t$ *v* **if** *u* $\in$ *keys p* **for** *u*
proof –
from *that* **have** *lookup p u* $\neq$ *0* **by** (*simp add*: *in-keys-iff*)
thus *u* $\preceq_t$ *v* **by** (*rule assms*(*2*))
qed
from *lt-alt*[*OF* ‹*p* $\neq$ *0*›] *ord-term-lin.Max-eqI*[*OF finite-keys this* ‹*v* $\in$ *keys p*›] **show** *?thesis* **by** *simp*
qed

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 –
from ‹*p* $\neq$ *0*› **have** *keys p* $\neq$ {}
by *simp*
have $\forall$ *u*$\in$*keys p. u* $\prec_t$ *v*
proof
fix *u*::*'t*
assume *u* $\in$ *keys p*
hence *lookup p u* $\neq$ *0* **by** (*simp add*: *in-keys-iff*)
hence ¬ *v* $\preceq_t$ *u* **using** *assms*(*2*)[*of u*] **by** *auto*
thus *u* $\prec_t$ *v* **by** *simp*
qed
with *lt-alt*[*OF assms*(*1*)] *ord-term-lin.Max-less-iff*[*OF finite-keys* ‹*keys p* $\neq$ {}›] **show** *?thesis* **by** *simp*
qed

lemma *lt-le*:
assumes $\bigwedge$*u. v* $\prec_t$ *u* $\Longrightarrow$ *lookup p u* = *0*
shows *lt p* $\preceq_t$ *v*
proof (*cases p* = *0*)

case *True*
show *?thesis* **by** (*simp add*: *True min-term-min*)
next
case *False*
hence *keys p* $\neq$ {} **by** *simp*
have $\forall$ *u*$\in$*keys p*. *u* $\preceq_t$ *v*
proof
fix *u*::'*t*
assume *u* $\in$ *keys p*
hence *lookup p u* $\neq$ *0* **unfolding** *keys-def* **by** *simp*
hence $\neg$ *v* $\prec_t$ *u* **using** *assms*[*of u*] **by** *auto*
thus *u* $\preceq_t$ *v* **by** *simp*
qed
with *lt-alt*[*OF False*] *ord-term-lin.Max-le-iff*[*OF finite-keys*[*of p*] ‹*keys p* $\neq$ {}›]
show *?thesis* **by** *simp*
qed

lemma *lt-gr*:
assumes *lookup p s* $\neq$ *0* **and** *t* $\prec_t$ *s*
shows *t* $\prec_t$ *lt p*
using *assms lt-max ord-term-lin.order.strict-trans2* **by** *blast*

lemma *lc-not-0*:
assumes *p* $\neq$ *0*
shows *lc p* $\neq$ *0*
proof −
from *keys-zero assms* **have** *keys p* $\neq$ {} **by** *auto*
from *lt-alt*[*OF assms*] *ord-term-lin.Max-in*[*OF finite-keys this*] **show** *?thesis* **by** (*simp add*: *in-keys-iff lc-def*)
qed

lemma *lc-eq-zero-iff*: *lc p* = *0* $\longleftrightarrow$ *p* = *0*
using *lc-not-0 lc-zero* **by** *blast*

lemma *lt-in-keys*:
assumes *p* $\neq$ *0*
shows *lt p* $\in$ (*keys p*)
by (*metis assms in-keys-iff lc-def lc-not-0*)

lemma *lt-monomial*:
lt (*monomial c t*) = *t* **if** *c* $\neq$ *0*
using *that* **by** (*auto simp add*: *lt-def dest*: *monomial-0D*)

lemma *lc-monomial* [*simp*]: *lc* (*monomial c t*) = *c*
proof (*cases c* = *0*)
case *True*
thus *?thesis* **by** *simp*
next
case *False*

thus *?thesis* **by** (*simp add*: *lc-def lt-monomial*)
qed

lemma *lt-le-iff*: *lt p* $\preceq_t$ *v* $\longleftrightarrow$ ($\forall$ *u*. *v* $\prec_t$ *u* $\longrightarrow$ *lookup p u* = *0*) (**is** *?L* $\longleftrightarrow$ *?R*)
proof
assume *?L*
show *?R*
proof (*intro allI impI*)
fix *u*
note ‹*lt p* $\preceq_t$ *v*›
also assume *v* $\prec_t$ *u*
finally have *lt p* $\prec_t$ *u* **.**
hence ¬ *u* $\preceq_t$ *lt p* **by** *simp*
with *lt-max*[*of p u*] **show** *lookup p u* = *0* **by** *blast*
qed
next
assume *?R*
thus *?L* **using** *lt-le* **by** *auto*
qed

lemma *lt-plus-eqI*:
assumes *lt p* $\prec_t$ *lt q*
shows *lt* (*p* + *q*) = *lt q*
proof (*cases q* = *0*)
case *True*
with *assms* **have** *lt p* $\prec_t$ *min-term* **by** (*simp add*: *lt-def*)
with *min-term-min*[*of lt p*] **show** *?thesis* **by** *simp*
next
case *False*
show *?thesis*
proof (*intro lt-eqI*)
from *lt-gr*[*of p lt q lt p*] *assms* **have** *lookup p* (*lt q*) = *0* **by** *blast*
with *lookup-add*[*of p q lt q*] *lc-not-0*[*OF False*] **show** *lookup* (*p* + *q*) (*lt q*) ≠ *0*
unfolding *lc-def* **by** *simp*
next
fix *u*
assume *lookup* (*p* + *q*) *u* ≠ *0*
show *u* $\preceq_t$ *lt q*
proof (*rule ccontr*)
assume ¬ *u* $\preceq_t$ *lt q*
hence *qs*: *lt q* $\prec_t$ *u* **by** *simp*
with *assms* **have** *lt p* $\prec_t$ *u* **by** *simp*
with *lt-gr*[*of p u lt p*] **have** *lookup p u* = *0* **by** *blast*
moreover from *qs lt-gr*[*of q u lt q*] **have** *lookup q u* = *0* **by** *blast*
ultimately show *False* **using** ‹*lookup* (*p* + *q*) *u* ≠ *0*› *lookup-add*[*of p q u*]
by *auto*
qed
qed
qed

lemma *lt-plus-eqI-2*:
assumes *lt q* $\prec_t$ *lt p*
shows *lt* (*p* + *q*) = *lt p*
proof (*cases p* = *0*)
case *True*
with *assms* **have** *lt q* $\prec_t$ *min-term* **by** (*simp add*: *lt-def*)
with *min-term-min*[*of lt q*] **show** *?thesis* **by** *simp*
next
case *False*
show *?thesis*
proof (*intro lt-eqI*)
from *lt-gr*[*of q lt p lt q*] *assms* **have** *lookup q* (*lt p*) = *0* **by** *blast*
with *lookup-add*[*of p q lt p*] *lc-not-0*[*OF False*] **show** *lookup* (*p* + *q*) (*lt p*) $\neq$ *0*
unfolding *lc-def* **by** *simp*
next
fix *u*
assume *lookup* (*p* + *q*) *u* $\neq$ *0*
show *u* $\preceq_t$ *lt p*
proof (*rule ccontr*)
assume $\neg$ *u* $\preceq_t$ *lt p*
hence *ps*: *lt p* $\prec_t$ *u* **by** *simp*
with *assms* **have** *lt q* $\prec_t$ *u* **by** *simp*
with *lt-gr*[*of q u lt q*] **have** *lookup q u* = *0* **by** *blast*
also from *ps lt-gr*[*of p u lt p*] **have** *lookup p u* = *0* **by** *blast*
ultimately show *False* **using** ‹*lookup* (*p* + *q*) *u* $\neq$ *0*› *lookup-add*[*of p q u*]
by *auto*
qed
qed
qed

lemma *lt-plus-eqI-3*:
assumes *lt q* = *lt p* **and** *lc p* + *lc q* $\neq$ *0*
shows *lt* (*p* + *q*) = *lt* (*p*::'*t* $\Rightarrow_0$ '*b*::*monoid-add*)
proof (*rule lt-eqI*)
from *assms*(*2*) **show** *lookup* (*p* + *q*) (*lt p*) $\neq$ *0* **by** (*simp add*: *lookup-add lc-def*
assms(*1*))
next
fix *u*
assume *lookup* (*p* + *q*) *u* $\neq$ *0*
hence *lookup p u* + *lookup q u* $\neq$ *0* **by** (*simp add*: *lookup-add*)
hence *lookup p u* $\neq$ *0* $\vee$ *lookup q u* $\neq$ *0* **by** *auto*
thus *u* $\preceq_t$ *lt p*
proof
assume *lookup p u* $\neq$ *0*
thus *?thesis* **by** (*rule lt-max*)
next
assume *lookup q u* $\neq$ *0*
hence *u* $\preceq_t$ *lt q* **by** (*rule lt-max*)

```
    thus ?thesis by (simp only: assms(1))
  qed
qed

lemma lt-plus-lessE:
  assumes lt p ≺t lt (p + q)
  shows lt p ≺t lt q
proof (rule ccontr)
  assume ¬ lt p ≺t lt q
  hence lt p = lt q ∨ lt q ≺t lt p by auto
  thus False
  proof
    assume lt-eq: lt p = lt q
    have lt (p + q) ⪯t lt p
    proof (rule lt-le)
      fix u
      assume lt p ≺t u
      with lt-gr[of p u lt p] have lookup p u = 0 by blast
      from ‹lt p ≺t u› have lt q ≺t u using lt-eq by simp
      with lt-gr[of q u lt q] have lookup q u = 0 by blast
     with ‹lookup p u = 0› show lookup (p + q) u = 0 by (simp add: lookup-add)
    qed
    with assms show False by simp
  next
    assume lt q ≺t lt p
    from lt-plus-eqI-2[OF this] assms show False by simp
  qed
qed

lemma lt-plus-lessE-2:
  assumes lt q ≺t lt (p + q)
  shows lt q ≺t lt p
proof (rule ccontr)
  assume ¬ lt q ≺t lt p
  hence lt q = lt p ∨ lt p ≺t lt q by auto
  thus False
  proof
    assume lt-eq: lt q = lt p
    have lt (p + q) ⪯t lt q
    proof (rule lt-le)
      fix u
      assume lt q ≺t u
      with lt-gr[of q u lt q] have lookup q u = 0 by blast
      from ‹lt q ≺t u› have lt p ≺t u using lt-eq by simp
      with lt-gr[of p u lt p] have lookup p u = 0 by blast
     with ‹lookup q u = 0› show lookup (p + q) u = 0 by (simp add: lookup-add)
    qed
    with assms show False by simp
  next
```

assume *lt p* $\prec_t$ *lt q*
from *lt-plus-eqI*[*OF this*] *assms* **show** *False* **by** *simp*
qed
qed

lemma *lt-plus-lessI'*:
fixes *p q* :: *'t* $\Rightarrow_0$ *'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*) $\prec_t$ *lt p*
proof (*rule ccontr*)
assume ¬ *lt* (*p* + *q*) $\prec_t$ *lt p*
hence *lt* (*p* + *q*) = *lt p* ∨ *lt p* $\prec_t$ *lt* (*p* + *q*) **by** *auto*
thus *False*
proof
assume *lt* (*p* + *q*) = *lt p*
have *lookup* (*p* + *q*) (*lt p*) = (*lookup p* (*lt p*)) + (*lookup q* (*lt q*)) **unfolding** *lt-eq lookup-add* **..**
also have ... = *lc p* + *lc q* **unfolding** *lc-def* **..**
also have ... = *0* **unfolding** *lc-eq* **by** *simp*
finally have *lookup* (*p* + *q*) (*lt p*) = *0* **.**
hence *lt* (*p* + *q*) ≠ *lt p* **using** *lc-not-0*[*OF* ‹*p* + *q* ≠ *0*›] **unfolding** *lc-def* **by** *auto*
with ‹*lt* (*p* + *q*) = *lt p*› **show** *False* **by** *simp*
next
assume *lt p* $\prec_t$ *lt* (*p* + *q*)
have *lt p* $\prec_t$ *lt q* **by** (*rule lt-plus-lessE*, *fact*+)
hence *lt p* ≠ *lt q* **by** *simp*
with *lt-eq* **show** *False* **by** *simp*
qed
qed

corollary *lt-plus-lessI*:
fixes *p q* :: *'t* $\Rightarrow_0$ *'b*::*group-add*
assumes *p* + *q* ≠ *0* **and** *lt q* = *lt p* **and** *lc q* = − *lc p*
shows *lt* (*p* + *q*) $\prec_t$ *lt p*
using *assms*(*1*, *2*)
proof (*rule lt-plus-lessI'*)
from *assms*(*3*) **show** *lc p* + *lc q* = *0* **by** *simp*
qed

lemma *lt-plus-distinct-eq-max*:
assumes *lt p* ≠ *lt q*
shows *lt* (*p* + *q*) = *ord-term-lin.max* (*lt p*) (*lt q*)
proof (*rule ord-term-lin.linorder-cases*)
assume *a*: *lt p* $\prec_t$ *lt q*
hence *lt* (*p* + *q*) = *lt q* **by** (*rule lt-plus-eqI*)
also from *a* **have** ... = *ord-term-lin.max* (*lt p*) (*lt q*)
by (*simp add*: *ord-term-lin.max.absorb2*)
finally show *?thesis* **.**

next
  assume *a*: *lt q* $\prec_t$ *lt p*
  hence *lt* (*p* + *q*) = *lt p* **by** (*rule lt-plus-eqI-2*)
  also from *a* **have** ... = *ord-term-lin.max* (*lt p*) (*lt q*)
    by (*simp add*: *ord-term-lin.max.absorb1*)
  finally show *?thesis* **.**
next
  assume *lt p* = *lt q*
  with *assms* **show** *?thesis* **..**
qed

lemma *lt-plus-le-max*: *lt* (*p* + *q*) $\preceq_t$ *ord-term-lin.max* (*lt p*) (*lt q*)
proof (*cases lt p* = *lt q*)
  case *True*
  show *?thesis*
  proof (*rule lt-le*)
    fix *u*
    assume *ord-term-lin.max* (*lt p*) (*lt q*) $\prec_t$ *u*
    hence *lt p* $\prec_t$ *u* **and** *lt q* $\prec_t$ *u* **by** *simp-all*
    hence *lookup p u* = *0* **and** *lookup q u* = *0* **using** *lt-max ord-term-lin.leD* **by** *blast+*
    thus *lookup* (*p* + *q*) *u* = *0* **by** (*simp add*: *lookup-add*)
  qed
next
  case *False*
  hence *lt* (*p* + *q*) = *ord-term-lin.max* (*lt p*) (*lt q*) **by** (*rule lt-plus-distinct-eq-max*)
  thus *?thesis* **by** *simp*
qed

lemma *lt-minus-eqI*: *lt p* $\prec_t$ *lt q* $\Longrightarrow$ *lt* (*p* − *q*) = *lt q* **for** *p q* :: *'t* $\Rightarrow_0$ *'b*::*ab-group-add*
  by (*metis lt-plus-eqI-2 lt-uminus uminus-add-conv-diff*)

lemma *lt-minus-eqI-2*: *lt q* $\prec_t$ *lt p* $\Longrightarrow$ *lt* (*p* − *q*) = *lt p* **for** *p q* :: *'t* $\Rightarrow_0$ *'b*::*ab-group-add*
  by (*metis lt-minus-eqI lt-uminus minus-diff-eq*)

lemma *lt-minus-eqI-3*:
  assumes *lt q* = *lt p* **and** *lc q* ≠ *lc p*
  shows *lt* (*p* − *q*) = *lt* (*p*::*'t* $\Rightarrow_0$ *'b*::*ab-group-add*)
proof (*rule lt-eqI*)
  from *assms*(*2*) **show** *lookup* (*p* − *q*) (*lt p*) ≠ *0* **by** (*simp add*: *lookup-minus lc-def assms*(*1*))
next
  fix *u*
  assume *lookup* (*p* − *q*) *u* ≠ *0*
  hence *lookup p u* ≠ *lookup q u* **by** (*simp add*: *lookup-minus*)
  hence *lookup p u* ≠ *0* ∨ *lookup q u* ≠ *0* **by** *auto*
  thus *u* $\preceq_t$ *lt p*
  proof

assume *lookup p u* $\neq$ *0*
thus *?thesis* **by** (*rule lt-max*)
next
assume *lookup q u* $\neq$ *0*
hence $u \preceq_t$ *lt q* **by** (*rule lt-max*)
thus *?thesis* **by** (*simp only*: *assms*(*1*))
qed
qed

lemma *lt-minus-distinct-eq-max*:
assumes *lt p* $\neq$ *lt* (*q*::*'t* $\Rightarrow_0$ *'b*::*ab-group-add*)
shows *lt* (*p* − *q*) = *ord-term-lin.max* (*lt p*) (*lt q*)
proof (*rule ord-term-lin.linorder-cases*)
assume *a*: *lt p* $\prec_t$ *lt q*
hence *lt* (*p* − *q*) = *lt q* **by** (*rule lt-minus-eqI*)
also from *a* **have** ... = *ord-term-lin.max* (*lt p*) (*lt q*)
by (*simp add*: *ord-term-lin.max.absorb2*)
finally show *?thesis* **.**
next
assume *a*: *lt q* $\prec_t$ *lt p*
hence *lt* (*p* − *q*) = *lt p* **by** (*rule lt-minus-eqI-2*)
also from *a* **have** ... = *ord-term-lin.max* (*lt p*) (*lt q*)
by (*simp add*: *ord-term-lin.max.absorb1*)
finally show *?thesis* **.**
next
assume *lt p* = *lt q*
with *assms* **show** *?thesis* **..**
qed

lemma *lt-minus-lessE*: *lt p* $\prec_t$ *lt* (*p* − *q*) $\Longrightarrow$ *lt p* $\prec_t$ *lt q* **for** *p q* :: *'t* $\Rightarrow_0$ *'b*::*ab-group-add*
using *lt-minus-eqI-2* **by** *fastforce*

lemma *lt-minus-lessE-2*: *lt q* $\prec_t$ *lt* (*p* − *q*) $\Longrightarrow$ *lt q* $\prec_t$ *lt p* **for** *p q* :: *'t* $\Rightarrow_0$ *'b*::*ab-group-add*
using *lt-plus-eqI-2* **by** *fastforce*

lemma *lt-minus-lessI*: *p* − *q* $\neq$ *0* $\Longrightarrow$ *lt q* = *lt p* $\Longrightarrow$ *lc q* = *lc p* $\Longrightarrow$ *lt* (*p* − *q*) $\prec_t$ *lt p*
for *p q* :: *'t* $\Rightarrow_0$ *'b*::*ab-group-add*
by (*metis* (*no-types*, *opaque-lifting*) *diff-diff-eq2 diff-self group-eq-aux lc-def lc-not-0 lookup-minus*
lt-minus-eqI ord-term-lin.antisym-conv3)

lemma *lt-max-keys*:
assumes *v* $\in$ *keys p*
shows *v* $\preceq_t$ *lt p*
proof (*rule lt-max*)
from *assms* **show** *lookup p v* $\neq$ *0* **by** (*simp add*: *in-keys-iff*)

```
qed

lemma lt-eqI-keys:
  assumes v ∈ keys p and a2: ⋀u. u ∈ keys p ⟹ u ≼t v
  shows lt p = v
  by (rule lt-eqI, simp-all only: in-keys-iff[symmetric], fact+)

lemma lt-gr-keys:
  assumes u ∈ keys p and v ≺t u
  shows v ≺t lt p
proof (rule lt-gr)
  from assms(1) show lookup p u ≠ 0 by (simp add: in-keys-iff)
qed fact

lemma lt-plus-eq-maxI:
  assumes lt p = lt q ⟹ lc p + lc q ≠ 0
  shows lt (p + q) = ord-term-lin.max (lt p) (lt q)
proof (cases lt p = lt q)
  case True
  show ?thesis
  proof (rule lt-eqI-keys)
    from True have lc p + lc q ≠ 0 by (rule assms)
    thus ord-term-lin.max (lt p) (lt q) ∈ keys (p + q)
      by (simp add: in-keys-iff lc-def lookup-add True)
  next
    fix u
    assume u ∈ keys (p + q)
    hence u ≼t lt (p + q) by (rule lt-max-keys)
    also have ... ≼t ord-term-lin.max (lt p) (lt q) by (fact lt-plus-le-max)
    finally show u ≼t ord-term-lin.max (lt p) (lt q) .
  qed
next
  case False
  thus ?thesis by (rule lt-plus-distinct-eq-max)
qed

lemma lt-monom-mult:
  assumes c ≠ (0::'b::semiring-no-zero-divisors) and p ≠ 0
  shows lt (monom-mult c t p) = t ⊕ lt p
proof (intro lt-eqI)
  from assms(1) show lookup (monom-mult c t p) (t ⊕ lt p) ≠ 0
  proof (simp add: lookup-monom-mult-plus)
    show lookup p (lt p) ≠ 0
      using assms(2) lt-in-keys by auto
  qed
next
  fix u::'t
  assume lookup (monom-mult c t p) u ≠ 0
  hence u ∈ keys (monom-mult c t p) by (simp add: in-keys-iff)
```

also have ... ⊆ (⊕) *t* ‘ *keys p* **by** (*fact keys-monom-mult-subset*)
finally obtain *v* **where** *v* ∈ *keys p* **and** *u* = *t* ⊕ *v* **..**
show $u \preceq_t$ *t* ⊕ *lt p* **unfolding** ‹*u* = *t* ⊕ *v*›
proof (*rule splus-mono*)
from ‹*v* ∈ *keys p*› **show** $v \preceq_t$ *lt p* **by** (*rule lt-max-keys*)
qed
qed

lemma *lt-monom-mult-zero*:
assumes *c* ≠ (*0*::′*b*::*semiring-no-zero-divisors*)
shows *lt* (*monom-mult c 0 p*) = *lt p*
proof (*cases p* = *0*)
case *True*
show *?thesis* **by** (*simp add*: *True*)
next
case *False*
with *assms* **show** *?thesis* **by** (*simp add*: *lt-monom-mult term-simps*)
qed

corollary *lt-map-scale*: *c* ≠ (*0*::′*b*::*semiring-no-zero-divisors*) ⟹ *lt* (*c* · *p*) = *lt p*
by (*simp add*: *map-scale-eq-monom-mult lt-monom-mult-zero*)

lemma *lc-monom-mult* [*simp*]: *lc* (*monom-mult c t p*) = (*c*::′*b*::*semiring-no-zero-divisors*) ∗ *lc p*
proof (*cases c* = *0*)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
show *?thesis*
proof (*cases p* = *0*)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
with ‹*c* ≠ *0*› **show** *?thesis* **by** (*simp add*: *lc-def lt-monom-mult lookup-monom-mult-plus*)
qed
qed

corollary *lc-map-scale* [*simp*]: *lc* (*c* · *p*) = (*c*::′*b*::*semiring-no-zero-divisors*) ∗ *lc p*
by (*simp add*: *map-scale-eq-monom-mult*)

lemma (**in** *ordered-term*) *lt-mult-scalar-monomial-right*:
assumes *c* ≠ (*0*::′*b*::*semiring-no-zero-divisors*) **and** *p* ≠ *0*
shows *lt* (*p* ⊙ *monomial c v*) = *punit.lt p* ⊕ *v*
proof (*intro lt-eqI*)
from *assms*(*1*) **show** *lookup* (*p* ⊙ *monomial c v*) (*punit.lt p* ⊕ *v*) ≠ *0*
proof (*simp add*: *lookup-mult-scalar-monomial-right-plus*)
from *assms*(*2*) **show** *lookup p* (*punit.lt p*) ≠ *0*

    using *in-keys-iff punit.lt-in-keys* **by** *fastforce*
  qed
next
  fix $u::'t$
  assume *lookup* $(p \odot$ *monomial* $c\ v)\ u \neq 0$
  hence $u \in$ *keys* $(p \odot$ *monomial* $c\ v)$ **by** (*simp add*: *in-keys-iff*)
  also have ... $\subseteq (\lambda t.\ t \oplus v)$ ‘ *keys p* **by** (*fact keys-mult-scalar-monomial-right-subset*)
  finally obtain t **where** $t \in$ *keys p* **and** $u = t \oplus v$ **..**
  show $u \preceq_t$ *punit.lt* $p \oplus v$ **unfolding** ‹$u = t \oplus v$›
  proof (*rule splus-mono-left*)
    from ‹$t \in$ *keys p*› **show** $t \preceq$ *punit.lt p* **by** (*rule punit.lt-max-keys*)
  qed
qed

lemma *lc-mult-scalar-monomial-right*:
  lc $(p \odot$ *monomial* $c\ v) =$ *punit.lc* $p * (c::'b::$*semiring-no-zero-divisors*$)$
proof (*cases* $c = 0$)
  case *True*
  thus *?thesis* **by** *simp*
next
  case *False*
  show *?thesis*
  proof (*cases* $p = 0$)
    case *True*
    thus *?thesis* **by** *simp*
  next
    case *False*
    with ‹$c \neq 0$› **show** *?thesis*
     by (*simp add*: *punit.lc-def lc-def lt-mult-scalar-monomial-right lookup-mult-scalar-monomial-right-plus*)
  qed
qed

lemma *lookup-monom-mult-eq-zero*:
  assumes $s \oplus$ *lt* $p \prec_t v$
  shows *lookup* (*monom-mult* $(c::'b::$*semiring-no-zero-divisors*$)\ s\ p)\ v = 0$
 by (*metis assms aux lt-gr lt-monom-mult monom-mult-zero-left monom-mult-zero-right*
    ord-term-lin.order.strict-implies-not-eq)

lemma *in-keys-monom-mult-le*:
  assumes $v \in$ *keys* (*monom-mult* $c\ t\ p$)
  shows $v \preceq_t t \oplus$ *lt* p
proof −
  from *keys-monom-mult-subset assms* **have** $v \in (\oplus)\ t$ ‘ (*keys p*) **..**
  then obtain u **where** $u \in$ *keys p* **and** $v = t \oplus u$ **..**
  from ‹$u \in$ *keys p*› **have** $u \preceq_t$ *lt p* **by** (*rule lt-max-keys*)
  thus $v \preceq_t t \oplus$ *lt p* **unfolding** ‹$v = t \oplus u$› **by** (*rule splus-mono*)
qed

lemma *lt-monom-mult-le*: *lt* (*monom-mult* $c\ t\ p$) $\preceq_t t \oplus$ *lt p*

by (*metis aux in-keys-monom-mult-le lt-in-keys lt-le-iff*)

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 −
from *assms*(*1*) **have** *lt* (*monom-mult c t1 p*) = *lt* (*monom-mult c t2 p*) **by** *simp*
with ‹$c \neq 0$› ‹$p \neq 0$› **have** $t1 \oplus lt\ p = t2 \oplus lt\ p$ **by** (*simp add*: *lt-monom-mult*)
thus *?thesis* **by** (*simp add*: *term-simps*)
qed

10.4 Trailing Term and Trailing Coefficient: *tt* and *tc*

lemma *tt-zero* [*simp*]: *tt 0 = min-term*
by (*simp add*: *tt-def*)

lemma *tc-zero* [*simp*]: *tc 0 = 0*
by (*simp add*: *tc-def*)

lemma *tt-alt*:
assumes $p \neq 0$
shows *tt p = ord-term-lin.Min* (*keys p*)
using *assms* **unfolding** *tt-def* **by** *simp*

lemma *tt-min-keys*:
assumes $v \in keys\ p$
shows $tt\ p \preceq_t v$
proof −
from *assms* **have** $keys\ p \neq \{\}$ **by** *auto*
hence $p \neq 0$ **by** *simp*
from *tt-alt*[*OF this*] *ord-term-lin.Min-le*[*OF finite-keys assms*] **show** *?thesis* **by** *simp*
qed

lemma *tt-min*:
assumes $lookup\ p\ v \neq 0$
shows $tt\ p \preceq_t v$
proof −
from *assms* **have** $v \in keys\ p$ **unfolding** *keys-def* **by** *simp*
thus *?thesis* **by** (*rule tt-min-keys*)
qed

lemma *tt-in-keys*:
assumes $p \neq 0$
shows $tt\ p \in keys\ p$
unfolding *tt-alt*[*OF assms*]
by (*rule ord-term-lin.Min-in*, *fact finite-keys*, *simp add*: *assms*)

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 −
  from *assms*(*1*) **have** $keys\ p \neq \{\}$ **by** *auto*
  hence $p \neq 0$ **by** *simp*
  from *assms*(*1*) **have** $tt\ p \preceq_t v$ **by** (*rule tt-min-keys*)
  moreover have $v \preceq_t tt\ p$ **by** (*rule assms*(*2*), *rule tt-in-keys*, *fact* ‹$p \neq 0$›)
  ultimately show *?thesis* **by** *simp*
qed

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 −
  from ‹$p \neq 0$› **have** $keys\ p \neq \{\}$ **by** *simp*
  show *?thesis* **by** (*rule assms*(*1*), *rule tt-in-keys*, *fact* ‹$p \neq 0$›)
qed

lemma *tt-less*:
  assumes $u \in keys\ p$ **and** $u \prec_t v$
  shows $tt\ p \prec_t v$
proof −
  from ‹$u \in keys\ p$› **have** $tt\ p \preceq_t u$ **by** (*rule tt-min-keys*)
  also have ... $\prec_t v$ **by** *fact*
  finally show $tt\ p \prec_t v$ **.**
qed

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 −
  from ‹$p \neq 0$› **have** $keys\ p \neq \{\}$ **by** *simp*
  have $\forall u \in keys\ p.\ v \preceq_t u$
  proof
    fix $u::'t$
    assume $u \in keys\ p$
    hence $lookup\ p\ u \neq 0$ **unfolding** *keys-def* **by** *simp*
    hence $\neg\ u \prec_t v$ **using** *assms*(*1*)[*of u*] **by** *auto*
    thus $v \preceq_t u$ **by** *simp*
  qed
  with *tt-alt*[*OF* ‹$p \neq 0$›] *ord-term-lin.Min-ge-iff*[*OF finite-keys*[*of p*] ‹$keys\ p \neq \{\}$›]
    show *?thesis* **by** *simp*
qed

lemma *tt-ge-keys*:
  assumes $\bigwedge u.\ u \in keys\ p \Longrightarrow v \preceq_t u$ **and** $p \neq 0$
  shows $v \preceq_t tt\ p$

by (*rule assms*(*1*), *rule tt-in-keys*, *fact*)

lemma *tt-ge-iff*: $v \preceq_t$ *tt p* $\longleftrightarrow$ (($p \neq 0 \lor v =$ *min-term*) $\land$ ($\forall u.\ u \prec_t v \longrightarrow$ *lookup p u* $= 0$))
(**is** *?L* $\longleftrightarrow$ (*?A* $\land$ *?B*))
proof
assume *?L*
show *?A* $\land$ *?B*
proof (*intro conjI allI impI*)
show $p \neq 0 \lor v =$ *min-term*
proof (*cases* $p = 0$)
case *True*
show *?thesis*
proof (*rule disjI2*)
from ‹*?L*› *True* **have** $v \preceq_t$ *min-term* **by** (*simp add*: *tt-def*)
with *min-term-min*[*of v*] **show** $v =$ *min-term* **by** *simp*
qed
next
case *False*
thus *?thesis* **..**
qed
next
fix *u*
assume $u \prec_t v$
also note ‹$v \preceq_t$ *tt p*›
finally have $u \prec_t$ *tt p* **.**
hence $\neg$ *tt p* $\preceq_t u$ **by** *simp*
with *tt-min*[*of p u*] **show** *lookup p u* $= 0$ **by** *blast*
qed
next
assume *?A* $\land$ *?B*
hence *?A* **and** *?B* **by** *simp-all*
show *?L*
proof (*cases* $p = 0$)
case *True*
with ‹*?A*› **have** $v =$ *min-term* **by** *simp*
with *True* **show** *?thesis* **by** (*simp add*: *tt-def*)
next
case *False*
from ‹*?B*› **show** *?thesis* **using** *tt-ge*[*OF - False*] **by** *auto*
qed
qed

lemma *tc-not-0*:
assumes $p \neq 0$
shows *tc p* $\neq 0$
unfolding *tc-def in-keys-iff*[*symmetric*] **using** *assms* **by** (*rule tt-in-keys*)

lemma *tt-monomial*:

assumes $c \neq 0$
shows *tt* (*monomial c v*) = *v*
proof (*rule tt-eqI*)
from *keys-of-monomial*[*OF assms*, *of v*] **show** $v \in$ *keys* (*monomial c v*) **by** *simp*
next
fix *u*
assume $u \in$ *keys* (*monomial c v*)
with *keys-of-monomial*[*OF assms*, *of v*] **have** $u = v$ **by** *simp*
thus $v \preceq_t u$ **by** *simp*
qed

lemma *tc-monomial* [*simp*]: *tc* (*monomial c t*) = *c*
proof (*cases* $c = 0$)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
thus *?thesis* **by** (*simp add*: *tc-def tt-monomial*)
qed

lemma *tt-plus-eqI*:
assumes $p \neq 0$ **and** *tt p* $\prec_t$ *tt q*
shows *tt* $(p + q)$ = *tt p*
proof (*intro tt-eqI*)
from *tt-less*[*of tt p q tt q*] ‹*tt p* $\prec_t$ *tt q*› **have** *tt p* $\notin$ *keys q* **by** *blast*
with *lookup-add*[*of p q tt p*] *tc-not-0*[*OF* ‹$p \neq 0$›] **show** *tt p* $\in$ *keys* $(p + q)$
unfolding *in-keys-iff tc-def* **by** *simp*
next
fix *u*
assume $u \in$ *keys* $(p + q)$
show *tt p* $\preceq_t u$
proof (*rule ccontr*)
assume ¬ *tt p* $\preceq_t u$
hence *sp*: $u \prec_t$ *tt p* **by** *simp*
hence $u \prec_t$ *tt q* **using** ‹*tt p* $\prec_t$ *tt q*› **by** *simp*
with *tt-less*[*of u q tt q*] **have** $u \notin$ *keys q* **by** *blast*
moreover from *sp tt-less*[*of u p tt p*] **have** $u \notin$ *keys p* **by** *blast*
ultimately show *False* **using** ‹$u \in$ *keys* $(p + q)$› *Poly-Mapping.keys-add*[*of p q*] **by** *auto*
qed
qed

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 (*cases* $p = 0$)
case *True*
with *tt* **show** *?thesis* **by** *simp*

next
  case *False*
  show *?thesis*
  proof (*rule ccontr*)
    assume ¬ *tt q* $\prec_t$ *tt p*
    hence *tt p* = *tt q* ∨ *tt p* $\prec_t$ *tt q* **by** *auto*
    thus *False*
    proof
      assume *tt-eq*: *tt p* = *tt q*
      have *tt p* $\preceq_t$ *tt* (*p* + *q*)
      proof (*rule tt-ge-keys*)
        fix *u*
        assume *u* ∈ *keys* (*p* + *q*)
        hence *u* ∈ *keys p* ∪ *keys q*
        proof
          show *keys* (*p* + *q*) ⊆ *keys p* ∪ *keys q* **by** (*fact Poly-Mapping.keys-add*)
        qed
        thus *tt p* $\preceq_t$ *u*
        proof
          assume *u* ∈ *keys p*
          thus *?thesis* **by** (*rule tt-min-keys*)
        next
          assume *u* ∈ *keys q*
          thus *?thesis* **unfolding** *tt-eq* **by** (*rule tt-min-keys*)
        qed
      qed (*fact* ‹*p* + *q* ≠ *0*›)
      with *tt* **show** *False* **by** *simp*
    next
      assume *tt p* $\prec_t$ *tt q*
      from *tt-plus-eqI*[*OF False this*] *tt* **show** *False* **by** (*simp add*: *ac-simps*)
    qed
  qed
qed

lemma *tt-plus-lessI*:
  fixes *p q* :: - ⇒$_0$ ′*b*::*ring*
  assumes *p* + *q* ≠ *0* **and** *tt-eq*: *tt q* = *tt p* **and** *tc-eq*: *tc q* = − *tc p*
  shows *tt p* $\prec_t$ *tt* (*p* + *q*)
proof (*rule ccontr*)
  assume ¬ *tt p* $\prec_t$ *tt* (*p* + *q*)
  hence *tt p* = *tt* (*p* + *q*) ∨ *tt* (*p* + *q*) $\prec_t$ *tt p* **by** *auto*
  thus *False*
  proof
    assume *tt p* = *tt* (*p* + *q*)
    have *lookup* (*p* + *q*) (*tt p*) = (*lookup p* (*tt p*)) + (*lookup q* (*tt q*)) **unfolding** *tt-eq lookup-add* **..**
    also have ... = *tc p* + *tc q* **unfolding** *tc-def* **..**
    also have ... = *0* **unfolding** *tc-eq* **by** *simp*
    finally have *lookup* (*p* + *q*) (*tt p*) = *0* **.**

hence *tt* (*p* + *q*) ≠ *tt p* **using** *tc-not-0*[*OF* ‹*p* + *q* ≠ *0*›] **unfolding** *tc-def* **by** *auto*
with ‹*tt p* = *tt* (*p* + *q*)› **show** *False* **by** *simp*
next
assume *tt* (*p* + *q*) $\prec_t$ *tt p*
have *tt q* $\prec_t$ *tt p* **by** (*rule tt-plus-lessE*, *fact*+)
hence *tt q* ≠ *tt p* **by** *simp*
with *tt-eq* **show** *False* **by** *simp*
qed
qed

lemma *tt-uminus* [*simp*]: *tt* (− *p*) = *tt p*
by (*simp add*: *tt-def keys-uminus*)

lemma *tc-uminus* [*simp*]: *tc* (− *p*) = − *tc p*
by (*simp add*: *tc-def*)

lemma *tt-monom-mult*:
assumes *c* ≠ (*0*::′*b*::*semiring-no-zero-divisors*) **and** *p* ≠ *0*
shows *tt* (*monom-mult c t p*) = *t* ⊕ *tt p*
proof (*intro tt-eqI*, *rule keys-monom-multI*, *rule tt-in-keys*, *fact*, *fact*)
fix *u*
assume *u* ∈ *keys* (*monom-mult c t p*)
then obtain *v* **where** *v* ∈ *keys p* **and** *u*: *u* = *t* ⊕ *v* **by** (*rule keys-monom-multE*)
show *t* ⊕ *tt p* $\preceq_t$ *u* **unfolding** *u add.commute*[*of t*] **by** (*rule splus-mono*, *rule tt-min-keys*, *fact*)
qed

lemma *tt-map-scale*: *c* ≠ (*0*::′*b*::*semiring-no-zero-divisors*) ⟹ *tt* (*c* · *p*) = *tt p*
by (*cases p* = *0*) (*simp-all add*: *map-scale-eq-monom-mult tt-monom-mult term-simps*)

lemma *tc-monom-mult* [*simp*]: *tc* (*monom-mult c t p*) = (*c*::′*b*::*semiring-no-zero-divisors*) ∗ *tc p*
proof (*cases c* = *0*)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
show *?thesis*
proof (*cases p* = *0*)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
with ‹*c* ≠ *0*› **show** *?thesis* **by** (*simp add*: *tc-def tt-monom-mult lookup-monom-mult-plus*)
qed
qed

corollary *tc-map-scale* [*simp*]: *tc* (*c* · *p*) = (*c*::′*b*::*semiring-no-zero-divisors*) ∗ *tc p*

by (*simp add*: *map-scale-eq-monom-mult*)

lemma *in-keys-monom-mult-ge*:
assumes $v \in$ *keys* (*monom-mult c t p*)
shows $t \oplus tt\ p \preceq_t v$
proof −
from *keys-monom-mult-subset assms* **have** $v \in (\oplus)\ t$ ‘ (*keys p*) **..**
then obtain *u* **where** $u \in$ *keys p* **and** $v = t \oplus u$ **..**
from ‹$u \in$ *keys p*› **have** $tt\ p \preceq_t u$ **by** (*rule tt-min-keys*)
thus $t \oplus tt\ p \preceq_t v$ **unfolding** ‹$v = t \oplus u$› **by** (*rule splus-mono*)
qed

lemma *lt-ge-tt*: $tt\ p \preceq_t lt\ p$
proof (*cases p = 0*)
case *True*
show *?thesis* **unfolding** *True lt-def tt-def* **by** *simp*
next
case *False*
show *?thesis* **by** (*rule lt-max-keys*, *rule tt-in-keys*, *fact False*)
qed

lemma *lt-eq-tt-monomial*:
assumes *is-monomial p*
shows *lt p = tt p*
proof −
from *assms* **obtain** *c v* **where** $c \neq 0$ **and** *p*: *p = monomial c v* **by** (*rule is-monomial-monomial*)
from ‹$c \neq 0$› **have** *lt p = v* **and** *tt p = v* **unfolding** *p* **by** (*rule lt-monomial*, *rule tt-monomial*)
thus *?thesis* **by** *simp*
qed

10.5 *higher* and *lower*

lemma *lookup-higher*: *lookup* (*higher p u*) *v* = (*if* $u \prec_t v$ *then lookup p v else 0*)
by (*auto simp add*: *higher-def lookup-except*)

lemma *lookup-higher-when*: *lookup* (*higher p u*) *v* = (*lookup p v when* $u \prec_t v$)
by (*auto simp add*: *lookup-higher when-def*)

lemma *higher-plus*: *higher* (*p* + *q*) *v* = *higher p v* + *higher q v*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-add lookup-higher*)

lemma *higher-uminus* [*simp*]: *higher* (− *p*) *v* = −(*higher p v*)
by (*rule poly-mapping-eqI*, *simp add*: *lookup-higher*)

lemma *higher-minus*: *higher* (*p* − *q*) *v* = *higher p v* − *higher q v*
by (*auto intro*!: *poly-mapping-eqI simp*: *lookup-minus lookup-higher*)

lemma *higher-zero* [*simp*]: *higher 0 t* = *0*
  by (*rule poly-mapping-eqI*, *simp add*: *lookup-higher*)

lemma *higher-eq-iff*: *higher p v* = *higher q v* ⟷ (∀ *u*. *v* $\prec_t$ *u* ⟶ *lookup p u* = *lookup q u*) (**is** *?L* ⟷ *?R*)
proof
  assume *?L*
  show *?R*
  proof (*intro allI impI*)
    fix *u*
    assume *v* $\prec_t$ *u*
    moreover from ‹*?L*› **have** *lookup* (*higher p v*) *u* = *lookup* (*higher q v*) *u* **by** *simp*
    ultimately show *lookup p u* = *lookup q u* **by** (*simp add*: *lookup-higher*)
  qed
next
  assume *?R*
  show *?L*
  proof (*rule poly-mapping-eqI*, *simp add*: *lookup-higher*, *rule*)
    fix *u*
    assume *v* $\prec_t$ *u*
    with ‹*?R*› **show** *lookup p u* = *lookup q u* **by** *simp*
  qed
qed

lemma *higher-eq-zero-iff*: *higher p v* = *0* ⟷ (∀ *u*. *v* $\prec_t$ *u* ⟶ *lookup p u* = *0*)
proof −
  have *higher p v* = *higher 0 v* ⟷ (∀ *u*. *v* $\prec_t$ *u* ⟶ *lookup p u* = *lookup 0 u*) **by** (*rule higher-eq-iff*)
  thus *?thesis* **by** *simp*
qed

lemma *keys-higher*: *keys* (*higher p v*) = {*u*∈*keys p*. *v* $\prec_t$ *u*}
  by (*rule set-eqI*, *simp only*: *in-keys-iff*, *simp add*: *lookup-higher*)

lemma *higher-higher*: *higher* (*higher p u*) *v* = *higher p* (*ord-term-lin.max u v*)
  by (*rule poly-mapping-eqI*, *simp add*: *lookup-higher*)

lemma *lookup-lower*: *lookup* (*lower p u*) *v* = (*if v* $\prec_t$ *u then lookup p v else 0*)
  by (*auto simp add*: *lower-def lookup-except*)

lemma *lookup-lower-when*: *lookup* (*lower p u*) *v* = (*lookup p v when v* $\prec_t$ *u*)
  by (*auto simp add*: *lookup-lower when-def*)

lemma *lower-plus*: *lower* (*p* + *q*) *v* = *lower p v* + *lower q v*
  by (*rule poly-mapping-eqI*, *simp add*: *lookup-add lookup-lower*)

lemma *lower-uminus* [*simp*]: *lower* (− *p*) *v* = − *lower p v*
  by (*rule poly-mapping-eqI*, *simp add*: *lookup-lower*)

lemma *lower-minus*: *lower* (*p* − (*q*::- ⇒$_0$ '*b*::*ab-group-add*)) *v* = *lower p v* − *lower q v*
  by (*auto intro*!: *poly-mapping-eqI simp*: *lookup-minus lookup-lower*)

lemma *lower-zero* [*simp*]: *lower 0 v* = *0*
  by (*rule poly-mapping-eqI*, *simp add*: *lookup-lower*)

lemma *lower-eq-iff*: *lower p v* = *lower q v* ⟷ (∀ *u*. *u* ≺$_t$ *v* ⟶ *lookup p u* = *lookup q u*) (**is** *?L* ⟷ *?R*)
proof
  assume *?L*
  show *?R*
  proof (*intro allI impI*)
    fix *u*
    assume *u* ≺$_t$ *v*
    moreover from ‹*?L*› **have** *lookup* (*lower p v*) *u* = *lookup* (*lower q v*) *u* **by** *simp*
    ultimately show *lookup p u* = *lookup q u* **by** (*simp add*: *lookup-lower*)
  qed
next
  assume *?R*
  show *?L*
  proof (*rule poly-mapping-eqI*, *simp add*: *lookup-lower*, *rule*)
    fix *u*
    assume *u* ≺$_t$ *v*
    with ‹*?R*› **show** *lookup p u* = *lookup q u* **by** *simp*
  qed
qed

lemma *lower-eq-zero-iff*: *lower p v* = *0* ⟷ (∀ *u*. *u* ≺$_t$ *v* ⟶ *lookup p u* = *0*)
proof −
  have *lower p v* = *lower 0 v* ⟷ (∀ *u*. *u* ≺$_t$ *v* ⟶ *lookup p u* = *lookup 0 u*) **by** (*rule lower-eq-iff*)
  thus *?thesis* **by** *simp*
qed

lemma *keys-lower*: *keys* (*lower p v*) = {*u*∈*keys p*. *u* ≺$_t$ *v*}
  by (*rule set-eqI*, *simp only*: *in-keys-iff*, *simp add*: *lookup-lower*)

lemma *lower-lower*: *lower* (*lower p u*) *v* = *lower p* (*ord-term-lin.min u v*)
  by (*rule poly-mapping-eqI*, *simp add*: *lookup-lower*)

lemma *lt-higher*:
  assumes *v* ≺$_t$ *lt p*
  shows *lt* (*higher p v*) = *lt p*
proof (*rule lt-eqI-keys*, *simp-all add*: *keys-higher*, *rule conjI*, *rule lt-in-keys*, *rule*)
  assume *p* = *0*
  hence *lt p* = *min-term* **by** (*simp add*: *lt-def*)

with *min-term-min*[*of v*] *assms* **show** *False* **by** *simp*
next
fix *u*
assume $u \in keys\ p \wedge v \prec_t u$
hence $u \in keys\ p$ **..**
thus $u \preceq_t lt\ p$ **by** (*rule lt-max-keys*)
qed *fact*

lemma *lc-higher*:
assumes $v \prec_t lt\ p$
shows *lc* (*higher p v*) = *lc p*
by (*simp add*: *lc-def lt-higher assms lookup-higher*)

lemma *higher-eq-zero-iff'*: $higher\ p\ v = 0 \longleftrightarrow lt\ p \preceq_t v$
by (*simp add*: *higher-eq-zero-iff lt-le-iff*)

lemma *higher-id-iff*: $higher\ p\ v = p \longleftrightarrow (p = 0 \vee v \prec_t tt\ p)$ (**is** $?L \longleftrightarrow ?R$)
proof
assume *?L*
show *?R*
proof (*cases p = 0*)
case *True*
thus *?thesis* **..**
next
case *False*
show *?thesis*
proof (*rule disjI2*, *rule tt-gr*)
fix *u*
assume $u \in keys\ p$
hence $lookup\ p\ u \neq 0$ **by** (*simp add*: *in-keys-iff*)
from ‹*?L*› **have** *lookup* (*higher p v*) *u* = *lookup p u* **by** *simp*
hence $lookup\ p\ u = (if\ v \prec_t u\ then\ lookup\ p\ u\ else\ 0)$ **by** (*simp only*: *lookup-higher*)
hence $\neg v \prec_t u \Longrightarrow lookup\ p\ u = 0$ **by** *simp*
with ‹$lookup\ p\ u \neq 0$› **show** $v \prec_t u$ **by** *auto*
qed *fact*
qed
next
assume *?R*
show *?L*
proof (*cases p = 0*)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
with ‹*?R*› **have** $v \prec_t tt\ p$ **by** *simp*
show *?thesis*
proof (*rule poly-mapping-eqI*, *simp add*: *lookup-higher*, *intro impI*)
fix *u*

assume ¬ $v \prec_t u$
hence $u \preceq_t v$ **by** *simp*
from *this* ‹$v \prec_t$ *tt p*› **have** $u \prec_t$ *tt p* **by** *simp*
hence ¬ *tt p* $\preceq_t u$ **by** *simp*
with *tt-min*[*of p u*] **show** *lookup p u* = *0* **by** *blast*
qed
qed
qed

lemma *tt-lower*:
assumes *tt p* $\prec_t v$
shows *tt* (*lower p v*) = *tt p*
proof (*cases p* = *0*)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
show *?thesis*
proof (*rule tt-eqI*, *simp-all add*: *keys-lower*, *rule*, *rule tt-in-keys*)
fix *u*
assume $u \in$ *keys p* ∧ $u \prec_t v$
hence $u \in$ *keys p* **..**
thus *tt p* $\preceq_t u$ **by** (*rule tt-min-keys*)
qed *fact+*
qed

lemma *tc-lower*:
assumes *tt p* $\prec_t v$
shows *tc* (*lower p v*) = *tc p*
by (*simp add*: *tc-def tt-lower assms lookup-lower*)

lemma *lt-lower*: *lt* (*lower p v*) $\preceq_t$ *lt p*
proof (*cases lower p v* = *0*)
case *True*
thus *?thesis* **by** (*simp add*: *lt-def min-term-min*)
next
case *False*
show *?thesis*
proof (*rule lt-le*, *simp add*: *lookup-lower*, *rule impI*, *rule ccontr*)
fix *u*
assume *lookup p u* ≠ *0*
hence $u \preceq_t$ *lt p* **by** (*rule lt-max*)
moreover assume *lt p* $\prec_t u$
ultimately show *False* **by** *simp*
qed
qed

lemma *lt-lower-less*:
assumes *lower p v* ≠ *0*

shows *lt* (*lower p v*) $\prec_t$ *v*
using *assms*
proof (*rule lt-less*)
fix *u*
assume *v* $\preceq_t$ *u*
thus *lookup* (*lower p v*) *u* = *0* **by** (*simp add*: *lookup-lower-when*)
qed

lemma *lt-lower-eq-iff*: *lt* (*lower p v*) = *lt p* ⟷ (*lt p* = *min-term* ∨ *lt p* $\prec_t$ *v*) (**is** *?L* ⟷ *?R*)
proof
assume *?L*
show *?R*
proof (*rule ccontr*, *simp*, *elim conjE*)
assume *lt p* ≠ *min-term*
hence *min-term* $\prec_t$ *lt p* **using** *min-term-min ord-term-lin.dual-order.not-eq-order-implies-strict* **by** *blast*
assume ¬ *lt p* $\prec_t$ *v*
hence *v* $\preceq_t$ *lt p* **by** *simp*
have *lt* (*lower p v*) $\prec_t$ *lt p*
proof (*cases lower p v* = *0*)
case *True*
thus *?thesis* **using** ‹*min-term* $\prec_t$ *lt p*› **by** (*simp add*: *lt-def*)
next
case *False*
show *?thesis*
proof (*rule lt-less*)
fix *u*
assume *lt p* $\preceq_t$ *u*
with ‹*v* $\preceq_t$ *lt p*› **have** ¬ *u* $\prec_t$ *v* **by** *simp*
thus *lookup* (*lower p v*) *u* = *0* **by** (*simp add*: *lookup-lower*)
qed *fact*
qed
with ‹*?L*› **show** *False* **by** *simp*
qed
next
assume *?R*
show *?L*
proof (*cases lt p* = *min-term*)
case *True*
hence *lt p* $\preceq_t$ *lt* (*lower p v*) **by** (*simp add*: *min-term-min*)
with *lt-lower*[*of p v*] **show** *?thesis* **by** *simp*
next
case *False*
with ‹*?R*› **have** *lt p* $\prec_t$ *v* **by** *simp*
show *?thesis*
proof (*rule lt-eqI-keys*, *simp-all add*: *keys-lower*, *rule conjI*, *rule lt-in-keys*, *rule*)
assume *p* = *0*

hence *lt p* = *min-term* **by** (*simp add*: *lt-def*)
with *False* **show** *False* **..**
next
fix *u*
assume $u \in keys\ p \wedge u \prec_t v$
hence $u \in keys\ p$ **..**
thus $u \preceq_t lt\ p$ **by** (*rule lt-max-keys*)
qed *fact*
qed
qed

lemma *tt-higher*:
assumes $v \prec_t lt\ p$
shows $tt\ p \preceq_t tt\ (higher\ p\ v)$
proof (*rule tt-ge-keys*, *simp add*: *keys-higher*)
fix *u*
assume $u \in keys\ p \wedge v \prec_t u$
hence $u \in keys\ p$ **..**
thus $tt\ p \preceq_t u$ **by** (*rule tt-min-keys*)
next
show $higher\ p\ v \neq 0$
proof (*simp add*: *higher-eq-zero-iff*, *intro exI conjI*)
have $p \neq 0$
proof
assume $p = 0$
hence $lt\ p \preceq_t v$ **by** (*simp add*: *lt-def min-term-min*)
with *assms* **show** *False* **by** *simp*
qed
thus $lookup\ p\ (lt\ p) \neq 0$
using *lt-in-keys* **by** *auto*
qed *fact*
qed

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
assume *?L*
show *?R*
proof (*rule ccontr*, *simp*, *elim conjE*)
assume *a*: $lt\ p \preceq_t v \longrightarrow tt\ p \neq min\text{-}term$
assume $\neg\ v \prec_t tt\ p$
hence $tt\ p \preceq_t v$ **by** *simp*
have $tt\ p \prec_t tt\ (higher\ p\ v)$
proof (*cases higher p v* = *0*)
case *True*
with ‹*?L*› **have** *tt p* = *min-term* **by** (*simp add*: *tt-def*)
with *a* **have** $v \prec_t lt\ p$ **by** *auto*
have $lt\ p \neq min\text{-}term$

```
    proof
      assume lt p = min-term
      with ‹v ≺t lt p› show False using min-term-min[of v] by auto
    qed
    hence p ≠ 0 by (auto simp add: lt-def)
    from ‹v ≺t lt p› have higher p v ≠ 0 by (simp add: higher-eq-zero-iff')
    from this True show ?thesis ..
  next
    case False
    show ?thesis
    proof (rule tt-gr)
      fix u
      assume u ∈ keys (higher p v)
      hence v ≺t u by (simp add: keys-higher)
      with ‹tt p ⪯t v› show tt p ≺t u by simp
    qed fact
  qed
  with ‹?L› show False by simp
 qed
next
 assume ?R
 show ?L
 proof (cases lt p ⪯t v ∧ tt p = min-term)
  case True
  hence lt p ⪯t v and tt p = min-term by simp-all
  from ‹lt p ⪯t v› have higher p v = 0 by (simp add: higher-eq-zero-iff')
  with ‹tt p = min-term› show ?thesis by (simp add: tt-def)
 next
  case False
  with ‹?R› have v ≺t tt p by simp
  show ?thesis
  proof (rule tt-eqI, simp-all add: keys-higher, rule conjI, rule tt-in-keys, rule)
    assume p = 0
    hence tt p = min-term by (simp add: tt-def)
    with ‹v ≺t tt p› min-term-min[of v] show False by simp
  next
    fix u
    assume u ∈ keys p ∧ v ≺t u
    hence u ∈ keys p ..
    thus tt p ⪯t u by (rule tt-min-keys)
  qed fact
 qed
qed

lemma lower-eq-zero-iff': lower p v = 0 ⟷ (p = 0 ∨ v ⪯t tt p)
 by (auto simp add: lower-eq-zero-iff tt-ge-iff)

lemma lower-id-iff: lower p v = p ⟷ (p = 0 ∨ lt p ≺t v) (is ?L ⟷ ?R)
proof
```

assume *?L*
show *?R*
proof (*cases p = 0*)
case *True*
thus *?thesis* **..**
next
case *False*
show *?thesis*
proof (*rule disjI2*, *rule lt-less*)
fix *u*
assume $v \preceq_t u$
from ‹*?L*› **have** *lookup* (*lower p v*) *u* = *lookup p u* **by** *simp*
hence *lookup p u* = (*if* $u \prec_t v$ *then lookup p u else 0*) **by** (*simp only*: *lookup-lower*)
hence $v \preceq_t u \Longrightarrow$ *lookup p u* = *0* **by** (*meson ord-term-lin.leD*)
with ‹$v \preceq_t u$› **show** *lookup p u* = *0* **by** *simp*
qed *fact*
qed
next
assume *?R*
show *?L*
proof (*cases p = 0*, *simp*)
case *False*
with ‹*?R*› **have** *lt p* $\prec_t$ *v* **by** *simp*
show *?thesis*
proof (*rule poly-mapping-eqI*, *simp add*: *lookup-lower*, *intro impI*)
fix *u*
assume ¬ $u \prec_t v$
hence $v \preceq_t u$ **by** *simp*
with ‹*lt p* $\prec_t$ *v*› **have** *lt p* $\prec_t$ *u* **by** *simp*
hence ¬ $u \preceq_t$ *lt p* **by** *simp*
with *lt-max*[*of p u*] **show** *lookup p u* = *0* **by** *blast*
qed
qed
qed

lemma *lower-higher-commute*: *higher* (*lower p s*) *t* = *lower* (*higher p t*) *s*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-higher lookup-lower*)

lemma *lt-lower-higher*:
assumes *v* $\prec_t$ *lt* (*lower p u*)
shows *lt* (*lower* (*higher p v*) *u*) = *lt* (*lower p u*)
by (*simp add*: *lower-higher-commute*[*symmetric*] *lt-higher*[*OF assms*])

lemma *lc-lower-higher*:
assumes *v* $\prec_t$ *lt* (*lower p u*)
shows *lc* (*lower* (*higher p v*) *u*) = *lc* (*lower p u*)
using *assms* **by** (*simp add*: *lc-def lt-lower-higher lookup-lower lookup-higher*)

lemma *trailing-monomial-higher*:
  assumes $p \neq 0$
  shows $p = (higher\ p\ (tt\ p)) + monomial\ (tc\ p)\ (tt\ p)$
proof (*rule poly-mapping-eqI*, *simp only*: *lookup-add*)
  fix *v*
  show $lookup\ p\ v = lookup\ (higher\ p\ (tt\ p))\ v + lookup\ (monomial\ (tc\ p)\ (tt\ p))\ v$
  proof (*cases* $tt\ p \preceq_t v$)
    case *True*
    show *?thesis*
    proof (*cases* $v = tt\ p$)
      assume $v = tt\ p$
      hence $\neg\ tt\ p \prec_t v$ **by** *simp*
      hence $lookup\ (higher\ p\ (tt\ p))\ v = 0$ **by** (*simp add*: *lookup-higher*)
      moreover from ‹$v = tt\ p$› **have** $lookup\ (monomial\ (tc\ p)\ (tt\ p))\ v = tc\ p$ **by** (*simp add*: *lookup-single*)
      moreover from ‹$v = tt\ p$› **have** $lookup\ p\ v = tc\ p$ **by** (*simp add*: *tc-def*)
      ultimately show *?thesis* **by** *simp*
    next
      assume $v \neq tt\ p$
      from *this True* **have** $tt\ p \prec_t v$ **by** *simp*
      hence $lookup\ (higher\ p\ (tt\ p))\ v = lookup\ p\ v$ **by** (*simp add*: *lookup-higher*)
      moreover from ‹$v \neq tt\ p$› **have** $lookup\ (monomial\ (tc\ p)\ (tt\ p))\ v = 0$ **by** (*simp add*: *lookup-single*)
      ultimately show *?thesis* **by** *simp*
    qed
  next
    case *False*
    hence $v \prec_t tt\ p$ **by** *simp*
    hence $tt\ p \neq v$ **by** *simp*
    from *False* **have** $\neg\ tt\ p \prec_t v$ **by** *simp*
    have $lookup\ p\ v = 0$
    proof (*rule ccontr*)
      assume $lookup\ p\ v \neq 0$
      from *tt-min*[*OF this*] *False* **show** *False* **by** *simp*
    qed
    moreover from ‹$tt\ p \neq v$› **have** $lookup\ (monomial\ (tc\ p)\ (tt\ p))\ v = 0$ **by** (*simp add*: *lookup-single*)
    moreover from ‹$\neg\ tt\ p \prec_t v$› **have** $lookup\ (higher\ p\ (tt\ p))\ v = 0$ **by** (*simp add*: *lookup-higher*)
    ultimately show *?thesis* **by** *simp*
  qed
qed

lemma *higher-lower-decomp*: $higher\ p\ v + monomial\ (lookup\ p\ v)\ v + lower\ p\ v = p$
proof (*rule poly-mapping-eqI*)
  fix *u*
  show $lookup\ (higher\ p\ v + monomial\ (lookup\ p\ v)\ v + lower\ p\ v)\ u = lookup\ p\ u$
  proof (*rule ord-term-lin.linorder-cases*)

assume $u \prec_t v$
thus *?thesis* **by** (*simp add*: *lookup-add lookup-higher-when lookup-single lookup-lower-when*)
next
assume $u = v$
thus *?thesis* **by** (*simp add*: *lookup-add lookup-higher-when lookup-single lookup-lower-when*)
next
assume $v \prec_t u$
thus *?thesis* **by** (*simp add*: *lookup-add lookup-higher-when lookup-single lookup-lower-when*)
qed
qed

10.6 *tail*

lemma *lookup-tail*: *lookup* (*tail p*) v = (*if* $v \prec_t$ *lt p then lookup p v else 0*)
by (*simp add*: *lookup-lower tail-def*)

lemma *lookup-tail-when*: *lookup* (*tail p*) v = (*lookup p v when* $v \prec_t$ *lt p*)
by (*simp add*: *lookup-lower-when tail-def*)

lemma *lookup-tail-2*: *lookup* (*tail p*) v = (*if v = lt p then 0 else lookup p v*)
proof (*rule ord-term-lin.linorder-cases*[*of v lt p*])
assume $v \prec_t$ *lt p*
hence $v \neq$ *lt p* **by** *simp*
from *this* ‹$v \prec_t$ *lt p*› *lookup-tail*[*of p v*] **show** *?thesis* **by** *simp*
next
assume *v = lt p*
hence $\neg\ v \prec_t$ *lt p* **by** *simp*
from ‹*v = lt p*› *this lookup-tail*[*of p v*] **show** *?thesis* **by** *simp*
next
assume *lt p* $\prec_t v$
hence $\neg\ v \preceq_t$ *lt p* **by** *simp*
hence *cp*: *lookup p v = 0*
using *lt-max* **by** *blast*
from ‹$\neg\ v \preceq_t$ *lt p*› **have** ¬ *v = lt p* **and** $\neg\ v \prec_t$ *lt p* **by** *simp-all*
thus *?thesis* **using** *cp lookup-tail*[*of p v*] **by** *simp*
qed

lemma *leading-monomial-tail*: *p = monomial* (*lc p*) (*lt p*) + *tail p* **for** *p*::- $\Rightarrow_0$
'b::*comm-monoid-add*
proof (*rule poly-mapping-eqI*)
fix *v*
have *lookup p v = lookup* (*monomial* (*lc p*) (*lt p*)) *v* + *lookup* (*tail p*) *v*
proof (*cases* $v \preceq_t$ *lt p*)
case *True*
show *?thesis*
proof (*cases v = lt p*)
assume *v = lt p*
hence $\neg\ v \prec_t$ *lt p* **by** *simp*
hence *c3*: *lookup* (*tail p*) *v = 0* **unfolding** *lookup-tail*[*of p v*] **by** *simp*

from ‹$v = lt\ p$› **have** *c2*: *lookup* (*monomial* (*lc p*) (*lt p*)) v = *lc p* **by** *simp*
from ‹$v = lt\ p$› **have** *c1*: *lookup p v* = *lc p* **by** (*simp add*: *lc-def*)
from *c1 c2 c3* **show** *?thesis* **by** *simp*
next
assume $v \neq lt\ p$
from *this True* **have** $v \prec_t lt\ p$ **by** *simp*
hence *c2*: *lookup* (*tail p*) v = *lookup p v* **unfolding** *lookup-tail*[*of p v*] **by** *simp*
from ‹$v \neq lt\ p$› **have** *c1*: *lookup* (*monomial* (*lc p*) (*lt p*)) $v = 0$ **by** (*simp add*: *lookup-single*)
from *c1 c2* **show** *?thesis* **by** *simp*
qed
next
case *False*
hence $lt\ p \prec_t v$ **by** *simp*
hence $lt\ p \neq v$ **by** *simp*
from *False* **have** $\neg\ v \prec_t lt\ p$ **by** *simp*
have *c1*: *lookup p v* = *0*
proof (*rule ccontr*)
assume *lookup p v* $\neq$ *0*
from *lt-max*[*OF this*] *False* **show** *False* **by** *simp*
qed
from ‹$lt\ p \neq v$› **have** *c2*: *lookup* (*monomial* (*lc p*) (*lt p*)) $v = 0$ **by** (*simp add*: *lookup-single*)
from ‹$\neg\ v \prec_t lt\ p$› *lookup-tail*[*of p v*] **have** *c3*: *lookup* (*tail p*) $v = 0$ **by** *simp*
from *c1 c2 c3* **show** *?thesis* **by** *simp*
qed
thus *lookup p v* = *lookup* (*monomial* (*lc p*) (*lt p*) + *tail p*) v **by** (*simp add*: *lookup-add*)
qed

lemma *tail-alt*: *tail p* = *except p* {*lt p*}
by (*rule poly-mapping-eqI*, *simp add*: *lookup-tail-2 lookup-except*)

corollary *tail-alt-2*: *tail p* = *p* − *monomial* (*lc p*) (*lt p*)
proof −
have *p* = *monomial* (*lc p*) (*lt p*) + *tail p* **by** (*fact leading-monomial-tail*)
also have ... = *tail p* + *monomial* (*lc p*) (*lt p*) **by** (*simp only*: *add.commute*)
finally have *p* − *monomial* (*lc p*) (*lt p*) = (*tail p* + *monomial* (*lc p*) (*lt p*)) − *monomial* (*lc p*) (*lt p*) **by** *simp*
thus *?thesis* **by** *simp*
qed

lemma *tail-zero* [*simp*]: *tail 0* = *0*
by (*simp only*: *tail-alt except-zero*)

lemma *lt-tail*:
assumes *tail p* $\neq$ *0*
shows *lt* (*tail p*) $\prec_t$ *lt p*

proof (*intro lt-less*)
  fix *u*
  assume *lt p* $\preceq_t$ *u*
  hence ¬ *u* $\prec_t$ *lt p* **by** *simp*
  thus *lookup* (*tail p*) *u* = *0* **unfolding** *lookup-tail*[*of p u*] **by** *simp*
qed *fact*

lemma *keys-tail*: *keys* (*tail p*) = *keys p* − {*lt p*}
  by (*simp add*: *tail-alt keys-except*)

lemma *tail-monomial*: *tail* (*monomial c v*) = *0*
  by (*metis* (*no-types*, *lifting*) *lookup-tail-2 lookup-single-not-eq lt-less lt-monomial*
    ord-term-lin.dual-order.strict-implies-not-eq single-zero tail-zero)

lemma (**in** *ordered-term*) *mult-scalar-tail-rec-left*:
  p ⊙ *q* = *monom-mult* (*punit.lc p*) (*punit.lt p*) *q* + (*punit.tail p*) ⊙ *q*
  unfolding *punit.lc-def punit.tail-alt* **by** (*fact mult-scalar-rec-left*)

lemma *mult-scalar-tail-rec-right*: *p* ⊙ *q* = *p* ⊙ *monomial* (*lc q*) (*lt q*) + *p* ⊙ *tail q*
  unfolding *tail-alt lc-def* **by** (*rule mult-scalar-rec-right*)

lemma *lt-tail-max*:
  assumes *tail p* ≠ *0* **and** *v* ∈ *keys p* **and** *v* $\prec_t$ *lt p*
  shows *v* $\preceq_t$ *lt* (*tail p*)
proof (*rule lt-max-keys*, *simp add*: *keys-tail assms*(*2*))
  from *assms*(*3*) **show** *v* ≠ *lt p* **by** *auto*
qed

lemma *keys-tail-less-lt*:
  assumes *v* ∈ *keys* (*tail p*)
  shows *v* $\prec_t$ *lt p*
  using *assms* **by** (*meson in-keys-iff lookup-tail*)

lemma *tt-tail*:
  assumes *tail p* ≠ *0*
  shows *tt* (*tail p*) = *tt p*
proof (*rule tt-eqI*, *simp-all add*: *keys-tail*)
  from *assms* **have** *p* ≠ *0* **using** *tail-zero* **by** *auto*
  show *tt p* ∈ *keys p* ∧ *tt p* ≠ *lt p*
  proof (*rule conjI*, *rule tt-in-keys*, *fact*)
    have *tt p* $\prec_t$ *lt p*
      by (*metis assms lower-eq-zero-iff′ tail-def ord-term-lin.le-less-linear*)
    thus *tt p* ≠ *lt p* **by** *simp*
  qed
next
  fix *u*
  assume *u* ∈ *keys p* ∧ *u* ≠ *lt p*
  hence *u* ∈ *keys p* **..**
  thus *tt p* $\preceq_t$ *u* **by** (*rule tt-min-keys*)

qed

lemma *tc-tail*:
  assumes *tail p* ≠ *0*
  shows *tc* (*tail p*) = *tc p*
proof (*simp add*: *tc-def tt-tail*[*OF assms*] *lookup-tail-2*, *rule*)
  assume *tt p* = *lt p*
  moreover have *tt p* $\prec_t$ *lt p*
    by (*metis assms lower-eq-zero-iff' tail-def ord-term-lin.le-less-linear*)
  ultimately show *lookup p* (*lt p*) = *0* **by** *simp*
qed

lemma *tt-tail-min*:
  assumes *s* ∈ *keys p*
  shows *tt* (*tail p*) $\preceq_t$ *s*
proof (*cases tail p* = *0*)
  case *True*
  hence *tt* (*tail p*) = *min-term* **by** (*simp add*: *tt-def*)
  thus *?thesis* **by** (*simp add*: *min-term-min*)
next
  case *False*
  from *assms* **show** *?thesis* **by** (*simp add*: *tt-tail*[*OF False*], *rule tt-min-keys*)
qed

lemma *tail-monom-mult*:
  tail (*monom-mult c t p*) = *monom-mult* (*c*::'*b*::*semiring-no-zero-divisors*) *t* (*tail p*)
proof (*cases p* = *0*)
  case *True*
  hence *tail p* = *0* **and** *monom-mult c t p* = *0* **by** *simp-all*
  thus *?thesis* **by** *simp*
next
  case *False*
  show *?thesis*
  proof (*cases c* = *0*)
    case *True*
    hence *monom-mult c t p* = *0* **and** *monom-mult c t* (*tail p*) = *0* **by** *simp-all*
    thus *?thesis* **by** *simp*
  next
    case *False*
    let *?a* = *monom-mult c t p*
    let *?b* = *monom-mult c t* (*tail p*)
    from ‹*p* ≠ *0*› *False* **have** *?a* ≠ *0* **by** (*simp add*: *monom-mult-eq-zero-iff*)
    from *False* ‹*p* ≠ *0*› **have** *lt-a*: *lt ?a* = *t* ⊕ *lt p* **by** (*rule lt-monom-mult*)
    show *?thesis*
    proof (*rule poly-mapping-eqI*, *simp add*: *lookup-tail lt-a*, *intro conjI impI*)
      fix *u*
      assume *u* $\prec_t$ *t* ⊕ *lt p*
      show *lookup* (*monom-mult c t p*) *u* = *lookup* (*monom-mult c t* (*tail p*)) *u*

proof (*cases t adds*$_p$ *u*)
case *True*
then obtain *v* **where** $u = t \oplus v$ **by** (*rule adds-ppE*)
from ‹$u \prec_t t \oplus lt\ p$› **have** $v \prec_t lt\ p$ **unfolding** ‹$u = t \oplus v$› **by** (*rule ord-term-strict-canc*)
hence *lookup p v = lookup* (*tail p*) *v* **by** (*simp add*: *lookup-tail*)
thus *?thesis* **by** (*simp add*: ‹$u = t \oplus v$› *lookup-monom-mult-plus*)
next
case *False*
hence *lookup ?a u = 0* **by** (*simp add*: *lookup-monom-mult*)
moreover have *lookup ?b u = 0*
proof (*rule ccontr*, *simp only*: *in-keys-iff* [*symmetric*] *keys-monom-mult*[*OF* ‹$c \neq 0$›])
assume $u \in (\oplus)\ t$ ‘ *keys* (*tail p*)
then obtain *v* **where** $u = t \oplus v$ **by** *auto*
hence *t adds*$_p$ *u* **by** (*simp add*: *term-simps*)
with *False* **show** *False* **..**
qed
ultimately show *?thesis* **by** *simp*
qed
next
fix *u*
assume $\neg\ u \prec_t t \oplus lt\ p$
hence $t \oplus lt\ p \preceq_t u$ **by** *simp*
show *lookup* (*monom-mult c t* (*tail p*)) *u = 0*
proof (*rule ccontr*, *simp only*: *in-keys-iff* [*symmetric*] *keys-monom-mult*[*OF False*])
assume $u \in (\oplus)\ t$ ‘ *keys* (*tail p*)
then obtain *v* **where** $v \in$ *keys* (*tail p*) **and** $u = t \oplus v$ **by** *auto*
from ‹$t \oplus lt\ p \preceq_t u$› **have** $lt\ p \preceq_t v$ **unfolding** ‹$u = t \oplus v$› **by** (*rule ord-term-canc*)
from ‹$v \in$ *keys* (*tail p*)› **have** $v \in$ *keys p* **and** $v \neq lt\ p$ **by** (*simp-all add*: *keys-tail*)
from ‹$v \in$ *keys p*› **have** $v \preceq_t lt\ p$ **by** (*rule lt-max-keys*)
with ‹$lt\ p \preceq_t v$› **have** $v = lt\ p$ **by** *simp*
with ‹$v \neq lt\ p$› **show** *False* **..**
qed
qed
qed
qed

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*::- $\Rightarrow_0$ '*b*::*comm-monoid-add*)
shows *tail p* + *higher q* (*tt q*) = *0*
proof −
note *assms*(*3*)
also have ... $\preceq_t lt\ p$ **by** (*rule lt-ge-tt*)
finally have $tt\ q \prec_t lt\ p$ **.**

hence *lt p* ≠ *tt q* **by** *simp*
have *q* ≠ *0*
proof
  assume *q* = *0*
  hence *tt q* = *min-term* **by** (*simp add*: *tt-def*)
  with ‹*q* = *0*› *assms*(*1*) **have** *keys p* = {*lt p*, *min-term*} **by** *simp*
  hence *min-term* ∈ *keys p* **by** *simp*
  hence *tt p* $\preceq_t$ *tt q* **unfolding** ‹*tt q* = *min-term*› **by** (*rule tt-min-keys*)
  with *assms*(*3*) **show** *False* **by** *simp*
qed
hence *tc q* ≠ *0* **by** (*rule tc-not-0*)
have *p* = *monomial* (*lc p*) (*lt p*) + *tail p* **by** (*rule leading-monomial-tail*)
moreover from ‹*q* ≠ *0*› **have** *q* = *higher q* (*tt q*) + *monomial* (*tc q*) (*tt q*) **by** (*rule trailing-monomial-higher*)
ultimately have *pq*: *p* + *q* = (*monomial* (*lc p*) (*lt p*) + *monomial* (*tc q*) (*tt q*)) + (*tail p* + *higher q* (*tt q*))
  (**is** - = (*?m1* + *?m2*) + *?b*) **by** (*simp add*: *algebra-simps*)
have *keys-m1*: *keys ?m1* = {*lt p*}
proof (*rule keys-of-monomial*, *rule lc-not-0*, *rule*)
  assume *p* = *0*
  with *assms*(*2*) **have** *lt q* $\prec_t$ *min-term* **by** (*simp add*: *lt-def*)
  with *min-term-min*[*of lt q*] **show** *False* **by** *simp*
qed
moreover from ‹*tc q* ≠ *0*› **have** *keys-m2*: *keys ?m2* = {*tt q*} **by** (*rule keys-of-monomial*)
ultimately have *keys-m1-m2*: *keys* (*?m1* + *?m2*) = {*lt p*, *tt q*}
  using ‹*lt p* ≠ *tt q*› *keys-plus-eqI*[*of ?m1 ?m2*] **by** *auto*
show *?thesis*
proof (*rule ccontr*)
  assume *?b* ≠ *0*
  hence *keys ?b* ≠ {} **by** *simp*
  then obtain *t* **where** *t* ∈ *keys ?b* **by** *blast*
  hence *t-in*: *t* ∈ *keys* (*tail p*) ∪ *keys* (*higher q* (*tt q*))
    using *Poly-Mapping.keys-add*[*of tail p higher q* (*tt q*)] **by** *blast*
  hence *t* ≠ *lt p*
  proof (*rule*, *simp add*: *keys-tail*, *simp add*: *keys-higher*, *elim conjE*)
    assume *t* ∈ *keys q*
    hence *t* $\preceq_t$ *lt q* **by** (*rule lt-max-keys*)
    from *this assms*(*2*) **show** *?thesis* **by** *simp*
  qed
  moreover from *t-in* **have** *t* ≠ *tt q*
  proof (*rule*, *simp add*: *keys-tail*, *elim conjE*)
    assume *t* ∈ *keys p*
    hence *tt p* $\preceq_t$ *t* **by** (*rule tt-min-keys*)
    with *assms*(*3*) **show** *?thesis* **by** *simp*
  next
    assume *t* ∈ *keys* (*higher q* (*tt q*))
    thus *?thesis* **by** (*auto simp only*: *keys-higher*)
  qed
  ultimately have *t* ∉ *keys* (*?m1* + *?m2*) **by** (*simp add*: *keys-m1-m2*)

moreover from *in-keys-plusI2*[*OF* ‹$t \in$ *keys ?b*› *this*] **have** $t \in$ *keys* (*?m1* + *?m2*)
  by (*simp only*: *keys-m1-m2 pq*[*symmetric*] *assms*(*1*))
 ultimately show *False* **..**
 qed
qed

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-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 $p \prec_p q$
 unfolding *ord-strict-p-def* **using** *assms* **by** *blast*

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* $q\ u$
 using *assms* **unfolding** *ord-strict-p-def* **by** *blast*

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
 assume $p \preceq_p q$
 hence $p \prec_p q \vee p = q$ **by** (*simp only*: *ord-p-def*)
 thus *False*
 proof
  assume $p \prec_p q$
  then obtain v' **where** *1*: *lookup* $p\ v' = 0$ **and** *2*: *lookup* $q\ v' \neq 0$
   and *3*: $\bigwedge u.\ v' \prec_t u \Longrightarrow$ *lookup* $p\ u =$ *lookup* $q\ u$ **by** (*rule ord-strict-pE*, *blast*)
  from *1 2* **have** *lookup* $p\ v' \neq$ *lookup* $q\ v'$ **by** *simp*
  hence $\neg\ v \prec_t v'$ **using** *assms*(*3*) **by** *blast*
  hence $v' \prec_t v \vee v' = v$ **by** *auto*
  thus *?thesis*
  proof
   assume $v' \prec_t v$
   hence *lookup* $p\ v =$ *lookup* $q\ v$ **by** (*rule 3*)
   with *assms*(*1*) **show** *?thesis* **..**

next
assume $v' = v$
with *assms*(2) *1* **show** *?thesis* **by** *auto*
qed
next
assume $p = q$
hence *lookup p v = lookup q v* **by** *simp*
with *assms*(*1*) **show** *?thesis* **..**
qed
qed

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 –
from *assms* **have** $\neg\ p \preceq_p q$ **by** (*rule not-ord-pI*)
thus *?thesis* **by** (*simp add*: *ord-p-def*)
qed

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*)
unfolding *ord-strict-p-def higher-eq-iff* **..**

lemma *ord-strict-p-asymmetric*:
assumes $p \prec_p q$
shows $\neg\ q \prec_p p$
using *assms* **unfolding** *ord-strict-p-def*
proof
fix *v1*::'*t*
assume *lookup p v1* $= 0 \wedge$ *lookup q v1* $\neq 0 \wedge (\forall\, u.\ v1 \prec_t u \longrightarrow$ *lookup p u = lookup q u*)
hence *lookup p v1* $= 0$ **and** *lookup q v1* $\neq 0$ **and** *v1*: $\forall\, u.\ v1 \prec_t u \longrightarrow$ *lookup p u = lookup q u*
by *auto*
show $\neg\ (\exists\, v.$ *lookup q v* $= 0 \wedge$ *lookup p v* $\neq 0 \wedge (\forall\, u.\ v \prec_t u \longrightarrow$ *lookup q u = lookup p u*))
proof (*intro notI*, *erule exE*)
fix *v2*::'*t*
assume *lookup q v2* $= 0 \wedge$ *lookup p v2* $\neq 0 \wedge (\forall\, u.\ v2 \prec_t u \longrightarrow$ *lookup q u = lookup p u*)
hence *lookup q v2* $= 0$ **and** *lookup p v2* $\neq 0$ **and** *v2*: $\forall\, u.\ v2 \prec_t u \longrightarrow$ *lookup q u = lookup p u*
by *auto*
show *False*
proof (*rule ord-term-lin.linorder-cases*)
assume $v1 \prec_t v2$
from *v1*[*rule-format*, *OF this*] ‹*lookup q v2* $= 0$› ‹*lookup p v2* $\neq 0$› **show** *?thesis* **by** *simp*

next
assume *v1* = *v2*
thus *?thesis* **using** ‹*lookup p v1* = *0*› ‹*lookup p v2* ≠ *0*› **by** *simp*
next
assume *v2* $\prec_t$ *v1*
from *v2*[*rule-format*, *OF this*] ‹*lookup p v1* = *0*› ‹*lookup q v1* ≠ *0*› **show** *?thesis* **by** *simp*
qed
qed
qed

lemma *ord-strict-p-irreflexive*: ¬ *p* $\prec_p$ *p*
unfolding *ord-strict-p-def*
proof (*intro notI*, *erule exE*)
fix *v*::′*t*
assume *lookup p v* = *0* ∧ *lookup p v* ≠ *0* ∧ (∀ *u*. *v* $\prec_t$ *u* ⟶ *lookup p u* = *lookup p u*)
hence *lookup p v* = *0* **and** *lookup p v* ≠ *0* **by** *auto*
thus *False* **by** *simp*
qed

lemma *ord-strict-p-transitive*:
assumes *a* $\prec_p$ *b* **and** *b* $\prec_p$ *c*
shows *a* $\prec_p$ *c*
proof −
from ‹*a* $\prec_p$ *b*› **obtain** *v1* **where** *lookup a v1* = *0*
and *lookup b v1* ≠ *0*
and *v1*[*rule-format*]: (∀ *u*. *v1* $\prec_t$ *u* ⟶ *lookup a u* = *lookup b u*)
unfolding *ord-strict-p-def* **by** *auto*
from ‹*b* $\prec_p$ *c*› **obtain** *v2* **where** *lookup b v2* = *0*
and *lookup c v2* ≠ *0*
and *v2*[*rule-format*]: (∀ *u*. *v2* $\prec_t$ *u* ⟶ *lookup b u* = *lookup c u*)
unfolding *ord-strict-p-def* **by** *auto*
show *a* $\prec_p$ *c*
proof (*rule ord-term-lin.linorder-cases*)
assume *v1* $\prec_t$ *v2*
show *?thesis* **unfolding** *ord-strict-p-def*
proof
show *lookup a v2* = *0* ∧ *lookup c v2* ≠ *0* ∧ (∀ *u*. *v2* $\prec_t$ *u* ⟶ *lookup a u* = *lookup c u*)
proof (*intro conjI allI impI*)
from ‹*lookup b v2* = *0*› *v1*[*OF* ‹*v1* $\prec_t$ *v2*›] **show** *lookup a v2* = *0* **by** *simp*
next
from ‹*lookup c v2* ≠ *0*› **show** *lookup c v2* ≠ *0* **.**
next
fix *u*
assume *v2* $\prec_t$ *u*

from *ord-term-lin.less-trans*[*OF* ‹*v1* $\prec_t$ *v2*› *this*] **have** *v1* $\prec_t$ *u* **.**
from *v2*[*OF* ‹*v2* $\prec_t$ *u*›] *v1*[*OF this*] **show** *lookup a u* = *lookup c u* **by** *simp*
qed
qed
next
assume *v2* $\prec_t$ *v1*
show *?thesis* **unfolding** *ord-strict-p-def*
proof
show *lookup a v1* = *0* ∧ *lookup c v1* ≠ *0* ∧ (∀ *u*. *v1* $\prec_t$ *u* ⟶ *lookup a u* = *lookup c u*)
proof (*intro conjI allI impI*)
from ‹*lookup a v1* = *0*› **show** *lookup a v1* = *0* **.**
next
from ‹*lookup b v1* ≠ *0*› *v2*[*OF* ‹*v2* $\prec_t$ *v1*›] **show** *lookup c v1* ≠ *0* **by** *simp*
next
fix *u*
assume *v1* $\prec_t$ *u*
from *ord-term-lin.less-trans*[*OF* ‹*v2* $\prec_t$ *v1*› *this*] **have** *v2* $\prec_t$ *u* **.**
from *v1*[*OF* ‹*v1* $\prec_t$ *u*›] *v2*[*OF this*] **show** *lookup a u* = *lookup c u* **by** *simp*
qed
qed
next
assume *v1* = *v2*
thus *?thesis* **using** ‹*lookup b v1* ≠ *0*› ‹*lookup b v2* = *0*› **by** *simp*
qed
qed

sublocale *order ord-p ord-strict-p*
proof (*intro order-strictI*)
fix *p q* :: *'t* $\Rightarrow_0$ *'b*
show (*p* $\preceq_p$ *q*) = (*p* $\prec_p$ *q* ∨ *p* = *q*) **unfolding** *ord-p-def* **..**
next
fix *p q* :: *'t* $\Rightarrow_0$ *'b*
assume *p* $\prec_p$ *q*
thus ¬ *q* $\prec_p$ *p* **by** (*rule ord-strict-p-asymmetric*)
next
fix *p*::*'t* $\Rightarrow_0$ *'b*
show ¬ *p* $\prec_p$ *p* **by** (*fact ord-strict-p-irreflexive*)
next
fix *a b c* :: *'t* $\Rightarrow_0$ *'b*
assume *a* $\prec_p$ *b* **and** *b* $\prec_p$ *c*
thus *a* $\prec_p$ *c* **by** (*rule ord-strict-p-transitive*)
qed

lemma *ord-p-zero-min*: *0* $\preceq_p$ *p*
unfolding *ord-p-def ord-strict-p-def*
proof (*cases p* = *0*)
case *True*
thus (∃ *v*. *lookup 0 v* = *0* ∧ *lookup p v* ≠ *0* ∧ (∀ *u*. *v* $\prec_t$ *u* ⟶ *lookup 0 u* =

lookup p u)) ∨ *0* = *p*
  by *auto*
next
  case *False*
  show (∃ *v*. *lookup 0 v* = *0* ∧ *lookup p v* ≠ *0* ∧ (∀ *u*. *v* $\prec_t$ *u* ⟶ *lookup 0 u* = *lookup p u*)) ∨ *0* = *p*
  proof
    show (∃ *v*. *lookup 0 v* = *0* ∧ *lookup p v* ≠ *0* ∧ (∀ *u*. *v* $\prec_t$ *u* ⟶ *lookup 0 u* = *lookup p u*))
    proof
      show *lookup 0* (*lt p*) = *0* ∧ *lookup p* (*lt p*) ≠ *0* ∧ (∀ *u*. (*lt p*) $\prec_t$ *u* ⟶ *lookup 0 u* = *lookup p u*)
      proof (*intro conjI allI impI*)
        show *lookup 0* (*lt p*) = *0* **by** (*transfer*, *simp*)
      next
        from *lc-not-0*[*OF False*] **show** *lookup p* (*lt p*) ≠ *0* **unfolding** *lc-def* **.**
      next
        fix *u*
        assume *lt p* $\prec_t$ *u*
        hence ¬ *u* $\preceq_t$ *lt p* **by** *simp*
        hence *lookup p u* = *0* **using** *lt-max*[*of p u*] **by** *metis*
        thus *lookup 0 u* = *lookup p u* **by** *simp*
      qed
    qed
  qed
qed

lemma *lt-ord-p*:
  assumes *lt p* $\prec_t$ *lt q*
  shows *p* $\prec_p$ *q*
proof −
  have *q* ≠ *0*
  proof
    assume *q* = *0*
    with *assms* **have** *lt p* $\prec_t$ *min-term* **by** (*simp add*: *lt-def*)
    with *min-term-min*[*of lt p*] **show** *False* **by** *simp*
  qed
  show *?thesis* **unfolding** *ord-strict-p-def*
  proof (*intro exI conjI allI impI*)
    show *lookup p* (*lt q*) = *0*
    proof (*rule ccontr*)
      assume *lookup p* (*lt q*) ≠ *0*
      from *lt-max*[*OF this*] ‹*lt p* $\prec_t$ *lt q*› **show** *False* **by** *simp*
    qed
  next
    from *lc-not-0*[*OF* ‹*q* ≠ *0*›] **show** *lookup q* (*lt q*) ≠ *0* **unfolding** *lc-def* **.**
  next
    fix *u*
    assume *lt q* $\prec_t$ *u*

hence *lt p* $\prec_t$ *u* **using** ‹*lt p* $\prec_t$ *lt q*› **by** *simp*
have *c1*: *lookup q u* = *0*
proof (*rule ccontr*)
assume *lookup q u* ≠ *0*
from *lt-max*[*OF this*] ‹*lt q* $\prec_t$ *u*› **show** *False* **by** *simp*
qed
have *c2*: *lookup p u* = *0*
proof (*rule ccontr*)
assume *lookup p u* ≠ *0*
from *lt-max*[*OF this*] ‹*lt p* $\prec_t$ *u*› **show** *False* **by** *simp*
qed
from *c1 c2* **show** *lookup p u* = *lookup q u* **by** *simp*
qed
qed

lemma *ord-p-lt*:
assumes *p* $\preceq_p$ *q*
shows *lt p* $\preceq_t$ *lt q*
proof (*rule ccontr*)
assume ¬ *lt p* $\preceq_t$ *lt q*
hence *lt q* $\prec_t$ *lt p* **by** *simp*
from *lt-ord-p*[*OF this*] ‹*p* $\preceq_p$ *q*› **show** *False* **by** *simp*
qed

lemma *ord-p-tail*:
assumes *p* ≠ *0* **and** *lt p* = *lt q* **and** *p* $\prec_p$ *q*
shows *tail p* $\prec_p$ *tail q*
using *assms* **unfolding** *ord-strict-p-def*
proof −
assume *p* ≠ *0* **and** *lt p* = *lt q*
and ∃ *v*. *lookup p v* = *0* ∧ *lookup q v* ≠ *0* ∧ (∀ *u*. *v* $\prec_t$ *u* ⟶ *lookup p u* = *lookup q u*)
then obtain *v* **where** *lookup p v* = *0*
and *lookup q v* ≠ *0*
and *a*: ∀ *u*. *v* $\prec_t$ *u* ⟶ *lookup p u* = *lookup q u* **by** *auto*
from *lt-max*[*OF* ‹*lookup q v* ≠ *0*›] ‹*lt p* = *lt q*› **have** *v* $\prec_t$ *lt p* ∨ *v* = *lt p* **by** *auto*
hence *v* $\prec_t$ *lt p*
proof
assume *v* $\prec_t$ *lt p*
thus *?thesis* **.**
next
assume *v* = *lt p*
thus *?thesis* **using** *lc-not-0*[*OF* ‹*p* ≠ *0*›] ‹*lookup p v* = *0*› **unfolding** *lc-def* **by** *auto*
qed
have *pt*: *lookup* (*tail p*) *v* = *lookup p v* **using** *lookup-tail*[*of p v*] ‹*v* $\prec_t$ *lt p*› **by** *simp*
have *q* ≠ *0*

proof
assume $q = 0$
hence $p \prec_p 0$ **using** ‹$p \prec_p q$› **by** *simp*
hence $\neg\ 0 \preceq_p p$ **by** *auto*
thus *False* **using** *ord-p-zero-min*[*of p*] **by** *simp*
qed
have *qt*: *lookup* (*tail q*) *v* = *lookup q v*
using *lookup-tail*[*of q v*] ‹$v \prec_t$ *lt p*› ‹*lt p* = *lt q*› **by** *simp*
show ∃ *w*. *lookup* (*tail p*) *w* = *0* ∧ *lookup* (*tail q*) *w* ≠ *0* ∧
(∀ *u*. $w \prec_t u$ ⟶ *lookup* (*tail p*) *u* = *lookup* (*tail q*) *u*)
proof (*intro exI conjI allI impI*)
from *pt* ‹*lookup p v* = *0*› **show** *lookup* (*tail p*) *v* = *0* **by** *simp*
next
from *qt* ‹*lookup q v* ≠ *0*› **show** *lookup* (*tail q*) *v* ≠ *0* **by** *simp*
next
fix *u*
assume $v \prec_t u$
from *a*[*rule-format*, *OF* ‹$v \prec_t u$›] *lookup-tail*[*of p u*] *lookup-tail*[*of q u*]
‹*lt p* = *lt q*› **show** *lookup* (*tail p*) *u* = *lookup* (*tail q*) *u* **by** *simp*
qed
qed

lemma *tail-ord-p*:
assumes *p* ≠ *0*
shows *tail p* $\prec_p$ *p*
proof (*cases tail p* = *0*)
case *True*
with *ord-p-zero-min*[*of p*] ‹*p* ≠ *0*› **show** *?thesis* **by** *simp*
next
case *False*
from *lt-tail*[*OF False*] **show** *?thesis* **by** (*rule lt-ord-p*)
qed

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*) ∧ (*higher q v*) = *0*
using *le* **unfolding** *ord-p-def*
proof
assume $q \prec_p p$
thus *?thesis* **unfolding** *ord-strict-p-def*
proof
fix *w*
assume *lookup q w* = *0* ∧ *lookup p w* ≠ *0* ∧ (∀ *u*. $w \prec_t u$ ⟶ *lookup q u* = *lookup p u*)
hence *qs*: *lookup q w* = *0* **and** *ps*: *lookup p w* ≠ *0* **and** *u*: ∀ *u*. $w \prec_t u$ ⟶ *lookup q u* = *lookup p u*
by *auto*
from *hp* **have** *pu*: ∀ *u*. $v \prec_t u$ ⟶ *lookup p u* = *0* **by** (*simp only*: *higher-eq-zero-iff*)
from *pu*[*rule-format*, *of w*] *ps* **have** ¬ $v \prec_t w$ **by** *auto*

hence $w \preceq_t v$ **by** *simp*
hence $w \prec_t v \lor w = v$ **by** *auto*
hence *st*: $w \prec_t v$
proof (*rule disjE*, *simp-all*)
assume $w = v$
from *this pt ps* **show** *False* **by** *simp*
qed
show *?thesis*
proof
from *u*[*rule-format*, *OF st*] *pt* **show** *lookup q v = 0* **by** *simp*
next
have $\forall u.\ v \prec_t u \longrightarrow$ *lookup q u = 0*
proof (*intro allI*, *intro impI*)
fix *u*
assume $v \prec_t u$
from *this st* **have** $w \prec_t u$ **by** *simp*
from *u*[*rule-format*, *OF this*] *pu*[*rule-format*, *OF* ‹$v \prec_t u$›] **show** *lookup q u = 0* **by** *simp*
qed
thus *higher q v = 0* **by** (*simp only*: *higher-eq-zero-iff*)
qed
qed
next
assume *q = p*
thus *?thesis* **using** *assms* **by** *simp*
qed

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 −
from *tail* **obtain** *v* **where** *pt*: *lookup* (*tail p*) *v = 0*
and *qt*: *lookup* (*tail q*) $v \neq 0$
and *a*: $\forall u.\ v \prec_t u \longrightarrow$ *lookup* (*tail p*) *u = lookup* (*tail q*) *u*
unfolding *ord-strict-p-def* **by** *auto*
from *qt lookup-zero*[*of v*] **have** *tail q* $\neq 0$ **by** *auto*
from *lt-max*[*OF qt*] *lt-tail*[*OF this*] **have** $v \prec_t$ *lt q* **by** *simp*
hence $v \prec_t$ *lt p* **using** ‹*lt p = lt q*› **by** *simp*
show *?thesis* **unfolding** *ord-strict-p-def*
proof (*rule exI*[*of - v*], *intro conjI allI impI*)
from *lookup-tail*[*of p v*] ‹$v \prec_t$ *lt p*› *pt* **show** *lookup p v = 0* **by** *simp*
next
from *lookup-tail*[*of q v*] ‹$v \prec_t$ *lt q*› *qt* **show** *lookup q v* $\neq 0$ **by** *simp*
next
fix *u*
assume $v \prec_t u$
from *this a* **have** *s*: *lookup* (*tail p*) *u = lookup* (*tail q*) *u* **by** *simp*
show *lookup p u = lookup q u*
proof (*cases u = lt p*)

```
      case True
      from True ‹lc p = lc q› ‹lt p = lt q› show ?thesis unfolding lc-def by simp
    next
      case False
      from False s lookup-tail-2[of p u] lookup-tail-2[of q u] ‹lt p = lt q› show
?thesis by simp
    qed
  qed
qed

lemma ord-strict-p-recE1:
  assumes p ≺p q
  shows q ≠ 0
proof
  assume q = 0
  from this assms ord-p-zero-min[of p] show False by simp
qed

lemma ord-strict-p-recE2:
  assumes p ≠ 0 and p ≺p q and lt p = lt q
  shows lc p = lc q
proof –
  from ‹p ≺p q› obtain v where pt: lookup p v = 0
                         and qt: lookup q v ≠ 0
                         and a: ∀ u. v ≺t u ⟶ lookup p u = lookup q u
    unfolding ord-strict-p-def by auto
  show ?thesis
  proof (cases v ≺t lt p)
    case True
    from this a have lookup p (lt p) = lookup q (lt p) by simp
    thus ?thesis using ‹lt p = lt q› unfolding lc-def by simp
  next
    case False
    from this lt-max[OF qt] ‹lt p = lt q› have v = lt p by simp
    from this lc-not-0[OF ‹p ≠ 0›] pt show ?thesis unfolding lc-def by auto
  qed
qed

lemma ord-strict-p-rec [code]:
  p ≺p q =
  (q ≠ 0 ∧
    (p = 0 ∨
      (let v1 = lt p; v2 = lt q in
        (v1 ≺t v2 ∨ (v1 = v2 ∧ lookup p v1 = lookup q v2 ∧ lower p v1 ≺p lower
q v2))
      )
    )
  )
  (is ?L = ?R)
```

proof
  assume *?L*
  show *?R*
  proof (*intro conjI*, *rule ord-strict-p-recE1*, *fact*)
    have ((*lt p* = *lt q* ∧ *lc p* = *lc q* ∧ *tail p* $\prec_p$ *tail q*) ∨ *lt p* $\prec_t$ *lt q*) ∨ *p* = *0*
    proof (*intro disjCI*)
      assume *p* ≠ *0* **and** *nl*: ¬ *lt p* $\prec_t$ *lt q*
      from ‹*?L*› **have** *p* $\preceq_p$ *q* **by** *simp*
      from *ord-p-lt*[*OF this*] *nl* **have** *lt p* = *lt q* **by** *simp*
      show *lt p* = *lt q* ∧ *lc p* = *lc q* ∧ *tail p* $\prec_p$ *tail q*
        by (*intro conjI*, *fact*, *rule ord-strict-p-recE2*, *fact+*, *rule ord-p-tail*, *fact+*)
    qed
    thus *p* = *0* ∨
          (*let v1* = *lt p*; *v2* = *lt q in*
            (*v1* $\prec_t$ *v2* ∨ *v1* = *v2* ∧ *lookup p v1* = *lookup q v2* ∧ *lower p v1* $\prec_p$
lower q v2)
)
      unfolding *lc-def tail-def* **by** *auto*
  qed
next
  assume *?R*
  hence *q* ≠ *0*
    and *dis*: *p* = *0* ∨
              (*let v1* = *lt p*; *v2* = *lt q in*
               (*v1* $\prec_t$ *v2* ∨ *v1* = *v2* ∧ *lookup p v1* = *lookup q v2* ∧ *lower p v1* $\prec_p$
lower q v2)
)
    by *simp-all*
  show *?L*
  proof (*cases p* = *0*)
    assume *p* = *0*
    hence *p* $\preceq_p$ *q* **using** *ord-p-zero-min*[*of q*] **by** *simp*
    thus *?thesis* **using** ‹*p* = *0*› ‹*q* ≠ *0*› **by** *simp*
  next
    assume *p* ≠ *0*
    hence *let v1* = *lt p*; *v2* = *lt q in*
        (*v1* $\prec_t$ *v2* ∨ *v1* = *v2* ∧ *lookup p v1* = *lookup q v2* ∧ *lower p v1* $\prec_p$ *lower*
q v2)
      using *dis* **by** *simp*
    hence *lt p* $\prec_t$ *lt q* ∨ (*lt p* = *lt q* ∧ *lc p* = *lc q* ∧ *tail p* $\prec_p$ *tail q*)
      unfolding *lc-def tail-def* **by** (*simp add*: *Let-def*)
    thus *?thesis*
    proof
      assume *lt p* $\prec_t$ *lt q*
      from *lt-ord-p*[*OF this*] **show** *?thesis* **.**
    next
      assume *lt p* = *lt q* ∧ *lc p* = *lc q* ∧ *tail p* $\prec_p$ *tail q*
      hence *lt p* = *lt q* **and** *lc p* = *lc q* **and** *tail p* $\prec_p$ *tail q* **by** *simp-all*
      thus *?thesis* **by** (*rule ord-strict-p-recI*)

qed
qed
qed

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 −
from *ord-p-zero-min*[*of tail p*] **have** ∗: ¬ *tail p* $\prec_p$ *0* **by** *auto*
show *?thesis*
by (*simp add*: *ord-strict-p-rec*[*of p*] *Let-def tail-def*[*symmetric*] *lc-def*[*symmetric*] *monomial-0-iff tail-monomial* ∗, *simp add*: *lt-monomial cong*: *conj-cong*)
qed

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 −
from *assms*(*1*) **have** $c \neq 0$ **and** $p = 0 \vee$ *lt* $p \prec_t v$ **by** (*simp-all add*: *ord-strict-p-monomial-iff*)
from *this*(*2*) **show** *?thesis*
proof
assume $p = 0$
with *assms*(*2*) **show** *?thesis* **by** *simp*
next
assume *lt* $p \prec_t v$
from *assms*(*2*) **have** $q = 0 \vee$ *lt* $q \prec_t v$ **by** (*simp add*: *ord-strict-p-monomial-iff*)
thus *?thesis*
proof
assume $q = 0$
with *assms*(*1*) **show** *?thesis* **by** *simp*
next
assume *lt* $q \prec_t v$
with ‹*lt* $p \prec_t v$› **have** *lt* $(p + q) \prec_t v$
using *lt-plus-le-max ord-term-lin.dual-order.strict-trans2 ord-term-lin.max-less-iff-conj*
by *blast*
with ‹$c \neq 0$› **show** *?thesis* **by** (*simp add*: *ord-strict-p-monomial-iff*)
qed
qed
qed

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-mult c t p* $\prec_p$ *monom-mult c t q*
proof −
from *assms*(*1*) **obtain** *v* **where** *1*: *lookup p v* $= 0$ **and** *2*: *lookup q v* $\neq 0$
and *3*: $\bigwedge u.\ v \prec_t u \Longrightarrow$ *lookup p u* = *lookup q u* **unfolding** *ord-strict-p-def* **by** *auto*
show *?thesis* **unfolding** *ord-strict-p-def*
proof (*intro exI conjI allI impI*)
from *1* **show** *lookup* (*monom-mult c t p*) $(t \oplus v) = 0$ **by** (*simp add*: *lookup-monom-mult-plus*)

next
from *2 assms*(*2*) **show** *lookup* (*monom-mult c t q*) (*t* ⊕ *v*) ≠ *0* **by** (*simp add*: *lookup-monom-mult-plus*)
next
fix *u*
assume $t \oplus v \prec_t u$
show *lookup* (*monom-mult c t p*) *u* = *lookup* (*monom-mult c t q*) *u*
proof (*cases t* $adds_p$ *u*)
case *True*
then obtain *w* **where** *u*: *u* = *t* ⊕ *w* **..**
from ‹$t \oplus v \prec_t u$› **have** $v \prec_t w$ **unfolding** *u* **by** (*rule ord-term-strict-canc*)
hence *lookup p w* = *lookup q w* **by** (*rule 3*)
thus *?thesis* **by** (*simp add*: *u lookup-monom-mult-plus*)
next
case *False*
thus *?thesis* **by** (*simp add*: *lookup-monom-mult*)
qed
qed
qed

lemma *ord-strict-p-plus*:
assumes $p \prec_p q$ **and** *keys r* ∩ *keys q* = {}
shows $p + r \prec_p q + r$
proof −
from *assms*(*1*) **obtain** *v* **where** *1*: *lookup p v* = *0* **and** *2*: *lookup q v* ≠ *0*
and *3*: ⋀*u*. $v \prec_t u$ ⟹ *lookup p u* = *lookup q u* **unfolding** *ord-strict-p-def* **by** *auto*
have *eq*: *lookup r v* = *0*
by (*meson 2 assms*(*2*) *disjoint-iff-not-equal in-keys-iff*)
show *?thesis* **unfolding** *ord-strict-p-def*
proof (*intro exI conjI allI impI*, *simp-all add*: *lookup-add*)
from *1* **show** *lookup p v* + *lookup r v* = *0* **by** (*simp add*: *eq*)
next
from *2* **show** *lookup q v* + *lookup r v* ≠ *0* **by** (*simp add*: *eq*)
next
fix *u*
assume $v \prec_t u$
hence *lookup p u* = *lookup q u* **by** (*rule 3*)
thus *lookup p u* + *lookup r u* = *lookup q u* + *lookup r u* **by** *simp*
qed
qed

lemma *poly-mapping-tail-induct* [*case-names 0 tail*]:
assumes *P 0* **and** ⋀*p*. *p* ≠ *0* ⟹ *P* (*tail p*) ⟹ *P p*
shows *P p*
proof (*induct card* (*keys p*) *arbitrary*: *p*)
case *0*
with *finite-keys*[*of p*] **have** *keys p* = {} **by** *simp*
hence *p* = *0* **by** *simp*

from ‹*P 0*› **show** *?case* **unfolding** ‹*p* = *0*› **.**
next
case *ind*: (*Suc n*)
from *ind*(*2*) **have** *keys p* ≠ {} **by** *auto*
hence *p* ≠ *0* **by** *simp*
thus *?case*
proof (*rule assms*(*2*))
show *P* (*tail p*)
proof (*rule ind*(*1*))
from ‹*p* ≠ *0*› **have** *lt p* ∈ *keys p* **by** (*rule lt-in-keys*)
hence *card* (*keys* (*tail p*)) = *card* (*keys p*) − *1* **by** (*simp add*: *keys-tail*)
also have ... = *n* **unfolding** *ind*(*2*)[*symmetric*] **by** *simp*
finally show *n* = *card* (*keys* (*tail p*)) **by** *simp*
qed
qed
qed

lemma *poly-mapping-neqE*:
assumes *p* ≠ *q*
obtains *v* **where** *v* ∈ *keys p* ∪ *keys q* **and** *lookup p v* ≠ *lookup q v*
and ⋀*u*. $v \prec_t u$ ⟹ *lookup p u* = *lookup q u*
proof −
let *?A* = {*v*. *lookup p v* ≠ *lookup q v*}
define *v* **where** *v* = *ord-term-lin.Max ?A*
have *?A* ⊆ *keys p* ∪ *keys q*
using *UnI2 in-keys-iff* **by** *fastforce*
also have *finite* ... **by** (*rule finite-UnI*) (*fact finite-keys*)+
finally(*finite-subset*) **have** *fin*: *finite ?A* **.**
moreover have *?A* ≠ {}
proof
assume *?A* = {}
hence *p* = *q*
using *poly-mapping-eqI* **by** *fastforce*
with *assms* **show** *False* **..**
qed
ultimately have *v* ∈ *?A* **unfolding** *v-def* **by** (*rule ord-term-lin.Max-in*)
show *?thesis*
proof
from ‹*?A* ⊆ *keys p* ∪ *keys q*› ‹*v* ∈ *?A*› **show** *v* ∈ *keys p* ∪ *keys q* **..**
next
from ‹*v* ∈ *?A*› **show** *lookup p v* ≠ *lookup q v* **by** *simp*
next
fix *u*
assume $v \prec_t u$
show *lookup p u* = *lookup q u*
proof (*rule ccontr*)
assume *lookup p u* ≠ *lookup q u*
hence *u* ∈ *?A* **by** *simp*
with *fin* **have** $u \preceq_t v$ **unfolding** *v-def* **by** (*rule ord-term-lin.Max-ge*)

with ‹$v \prec_t u$› **show** *False* **by** *simp*
qed
qed
qed

10.8 Monomials

lemma *keys-monomial*:
assumes *is-monomial p*
shows *keys p* = {*lt p*}
using *assms* **by** (*metis is-monomial-monomial lt-monomial keys-of-monomial*)

lemma *monomial-eq-itself*:
assumes *is-monomial p*
shows *monomial* (*lc p*) (*lt p*) = *p*
proof −
from *assms* **have** $p \neq 0$ **by** (*rule monomial-not-0*)
hence *lc p* ≠ *0* **by** (*rule lc-not-0*)
hence *keys1*: *keys* (*monomial* (*lc p*) (*lt p*)) = {*lt p*} **by** (*rule keys-of-monomial*)
show *?thesis*
by (*rule poly-mapping-keys-eqI*, *simp only*: *keys-monomial*[*OF assms*] *keys1*,
simp only: *keys1 lookup-single Poly-Mapping.when-def*, *auto simp add*: *lc-def*)
qed

lemma *lt-eq-min-term-monomial*:
assumes *lt p* = *min-term*
shows *monomial* (*lc p*) *min-term* = *p*
proof (*rule poly-mapping-eqI*)
fix *v*
from *min-term-min*[*of v*] **have** *v* = *min-term* ∨ *min-term* $\prec_t$ *v* **by** *auto*
thus *lookup* (*monomial* (*lc p*) *min-term*) *v* = *lookup p v*
proof
assume *v* = *min-term*
thus *?thesis* **by** (*simp add*: *lookup-single lc-def assms*)
next
assume *min-term* $\prec_t$ *v*
moreover have *v* ∉ *keys p*
proof
assume *v* ∈ *keys p*
hence *v* $\preceq_t$ *lt p* **by** (*rule lt-max-keys*)
with ‹*min-term* $\prec_t$ *v*› **show** *False* **by** (*simp add*: *assms*)
qed
ultimately show *?thesis* **by** (*simp add*: *lookup-single in-keys-iff*)
qed
qed

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 −
 from *assms* **obtain** *c v* **where** $c \neq 0$ **and** *p-eq*: $p = monomial\ c\ v$ **by** (*rule is-monomial-monomial*)
 note *this*(*1*)
 moreover have $lc\ p = c$ **unfolding** *p-eq* **by** (*rule lc-monomial*)
 moreover from ‹$c \neq 0$› **have** $lt\ p = v$ **unfolding** *p-eq* **by** (*rule lt-monomial*)
 ultimately show *?thesis* **using** *p-eq* **..**
qed

lemma *monomial-plus-not-0*:
 assumes $c \neq 0$ **and** $lt\ p \prec_t v$
 shows $monomial\ c\ v + p \neq 0$
proof
 assume $monomial\ c\ v + p = 0$
 hence $0 = lookup\ (monomial\ c\ v + p)\ v$ **by** *simp*
 also have ... $= c + lookup\ p\ v$ **by** (*simp add*: *lookup-add*)
 also have ... $= c$
 proof −
  from *assms*(*2*) **have** $\neg\ v \preceq_t lt\ p$ **by** *simp*
  with *lt-max*[*of p v*] **have** $lookup\ p\ v = 0$ **by** *blast*
  thus *?thesis* **by** *simp*
 qed
 finally show *False* **using** ‹$c \neq 0$› **by** *simp*
qed

lemma *lt-monomial-plus*:
 assumes $c \neq (0::'b::comm\text{-}monoid\text{-}add)$ **and** $lt\ p \prec_t v$
 shows $lt\ (monomial\ c\ v + p) = v$
proof −
 have *eq*: $lt\ (monomial\ c\ v) = v$ **by** (*simp only*: *lt-monomial*[*OF* ‹$c \neq 0$›])
 moreover have $lt\ (p + monomial\ c\ v) = lt\ (monomial\ c\ v)$ **by** (*rule lt-plus-eqI*, *simp only*: *eq*, *fact*)
 ultimately show *?thesis* **by** (*simp add*: *add.commute*)
qed

lemma *lc-monomial-plus*:
 assumes $c \neq (0::'b::comm\text{-}monoid\text{-}add)$ **and** $lt\ p \prec_t v$
 shows $lc\ (monomial\ c\ v + p) = c$
proof −
 from *assms*(*2*) **have** $\neg\ v \preceq_t lt\ p$ **by** *simp*
 with *lt-max*[*of p v*] **have** $lookup\ p\ v = 0$ **by** *blast*
 thus *?thesis* **by** (*simp add*: *lc-def lt-monomial-plus*[*OF assms*] *lookup-add*)
qed

lemma *tt-monomial-plus*:
 assumes $p \neq (0::\text{-} \Rightarrow_0 'b::comm\text{-}monoid\text{-}add)$ **and** $lt\ p \prec_t v$
 shows $tt\ (monomial\ c\ v + p) = tt\ p$
proof (*cases* $c = 0$)
 case *True*

thus *?thesis* **by** (*simp add*: *monomial-0I*)
next
case *False*
have *eq*: *tt* (*monomial c v*) = *v* **by** (*simp only*: *tt-monomial*[*OF* ‹*c* ≠ *0*›])
moreover have *tt* (*p* + *monomial c v*) = *tt p*
proof (*rule tt-plus-eqI*, *fact*, *simp only*: *eq*)
from *lt-ge-tt*[*of p*] *assms*(*2*) **show** *tt p* $\prec_t$ *v* **by** *simp*
qed
ultimately show *?thesis* **by** (*simp add*: *ac-simps*)
qed

lemma *tc-monomial-plus*:
assumes *p* ≠ (*0*::- $\Rightarrow_0$ *'b*::*comm-monoid-add*) **and** *lt p* $\prec_t$ *v*
shows *tc* (*monomial c v* + *p*) = *tc p*
proof (*simp add*: *tc-def tt-monomial-plus*[*OF assms*] *lookup-add lookup-single Poly-Mapping.when-def*, *rule impI*)
assume *v* = *tt p*
with *assms*(*2*) **have** *lt p* $\prec_t$ *tt p* **by** *simp*
with *lt-ge-tt*[*of p*] **show** *c* + *lookup p* (*tt p*) = *lookup p* (*tt p*) **by** *simp*
qed

lemma *tail-monomial-plus*:
assumes *c* ≠ (*0*::*'b*::*comm-monoid-add*) **and** *lt p* $\prec_t$ *v*
shows *tail* (*monomial c v* + *p*) = *p* (**is** *tail ?q* = -)
proof −
from *assms* **have** *lt ?q* = *v* **by** (*rule lt-monomial-plus*)
moreover have *lower* (*monomial c v*) *v* = *0*
by (*simp add*: *lower-eq-zero-iff'*, *rule disjI2*, *simp add*: *tt-monomial*[*OF* ‹*c* ≠ *0*›])
ultimately show *?thesis* **by** (*simp add*: *tail-def lower-plus lower-id-iff*, *intro disjI2 assms*(*2*))
qed

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* $\Rightarrow_0$ *'b*::*zero*) ⇒ *'t list*
where *keys-to-list p* = *pps-to-list* (*keys p*)

definition *Keys-to-list* :: (*'t* $\Rightarrow_0$ *'b*::*zero*) *list* ⇒ *'t list*
where *Keys-to-list ps* = *fold* (λ*p ts*. *merge-wrt* ($\succ_t$) (*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*)

unfolding *pps-to-list-def distinct-rev* **by** (*rule ord-term-lin.distinct-sorted-list-of-set*)

lemma *set-pps-to-list*:
assumes *finite S*
shows *set* (*pps-to-list S*) = *S*
unfolding *pps-to-list-def set-rev* **using** *assms* **by** *simp*

lemma *length-pps-to-list*: *length* (*pps-to-list S*) = *card S*
proof (*cases finite S*)
case *True*
from *distinct-card*[*OF distinct-pps-to-list*] **have** *length* (*pps-to-list S*) = *card* (*set* (*pps-to-list S*))
by *simp*
also from *True* **have** ... = *card S* **by** (*simp only*: *set-pps-to-list*)
finally show *?thesis* **.**
next
case *False*
thus *?thesis* **by** (*simp add*: *pps-to-list-def*)
qed

lemma *pps-to-list-sorted-wrt*: *sorted-wrt* ($\succ_t$) (*pps-to-list S*)
proof −
have *sorted-wrt* ($\succeq_t$) (*pps-to-list S*)
proof −
have *tr*: *transp* ($\preceq_t$) **using** *transp-def* **by** *fastforce*
have ∗: ($\lambda x\ y.\ y \succeq_t x$) = ($\preceq_t$) **by** *simp*
show *?thesis*
by (*simp only*: ∗ *pps-to-list-def sorted-wrt-rev*,
rule ord-term-lin.sorted-sorted-list-of-set)
qed
with *distinct-pps-to-list* **have** *sorted-wrt* ($\lambda x\ y.\ x \succeq_t y \wedge x \neq y$) (*pps-to-list S*)
by (*rule distinct-sorted-wrt-imp-sorted-wrt-strict*)
moreover have ($\succ_t$) = ($\lambda x\ y.\ x \succeq_t y \wedge x \neq y$)
using *ord-term-lin.dual-order.order-iff-strict* **by** *auto*
ultimately show *?thesis* **by** *simp*
qed

lemma *pps-to-list-nth-leI*:
assumes $j \leq i$ **and** $i < card\ S$
shows (*pps-to-list S*) ! $i \preceq_t$ (*pps-to-list S*) ! j
proof (*cases* $j = i$)
case *True*
show *?thesis* **by** (*simp add*: *True*)
next
case *False*
with *assms*(*1*) **have** $j < i$ **by** *simp*
let *?ts* = *pps-to-list S*
from *pps-to-list-sorted-wrt* ‹$j < i$› **have** $(\prec_t)^{-1-1}$ (*?ts* ! j) (*?ts* ! i)
proof (*rule sorted-wrt-nth-less*)

from *assms*(*2*) **show** $i <$ *length ?ts* **by** (*simp only*: *length-pps-to-list*)
qed
thus *?thesis* **by** *simp*
qed

lemma *pps-to-list-nth-lessI*:
assumes $j < i$ **and** $i <$ *card S*
shows (*pps-to-list S*) ! $i \prec_t$ (*pps-to-list S*) ! j
proof −
let *?ts* = *pps-to-list S*
from *assms*(*1*) **have** $j \le i$ **and** $i \ne j$ **by** *simp-all*
with *assms*(*2*) **have** $i <$ *length ?ts* **and** $j <$ *length ?ts* **by** (*simp-all only*: *length-pps-to-list*)
show *?thesis*
proof (*rule ord-term-lin.neq-le-trans*)
from ‹$i \ne j$› **show** *?ts* ! $i \ne$ *?ts* ! j
by (*simp add*: *nth-eq-iff-index-eq*[*OF distinct-pps-to-list* ‹$i <$ *length ?ts*› ‹$j <$ *length ?ts*›])
next
from ‹$j \le i$› *assms*(*2*) **show** *?ts* ! $i \preceq_t$ *?ts* ! j **by** (*rule pps-to-list-nth-leI*)
qed
qed

lemma *pps-to-list-nth-leD*:
assumes (*pps-to-list S*) ! $i \preceq_t$ (*pps-to-list S*) ! j **and** $j <$ *card S*
shows $j \le i$
proof (*rule ccontr*)
assume $\neg\ j \le i$
hence $i < j$ **by** *simp*
from *this* ‹$j <$ *card S*› **have** (*pps-to-list S*) ! $j \prec_t$ (*pps-to-list S*) ! i **by** (*rule pps-to-list-nth-lessI*)
with *assms*(*1*) **show** *False* **by** *simp*
qed

lemma *pps-to-list-nth-lessD*:
assumes (*pps-to-list S*) ! $i \prec_t$ (*pps-to-list S*) ! j **and** $j <$ *card S*
shows $j < i$
proof (*rule ccontr*)
assume $\neg\ j < i$
hence $i \le j$ **by** *simp*
from *this* ‹$j <$ *card S*› **have** (*pps-to-list S*) ! $j \preceq_t$ (*pps-to-list S*) ! i **by** (*rule pps-to-list-nth-leI*)
with *assms*(*1*) **show** *False* **by** *simp*
qed

lemma *set-keys-to-list*: *set* (*keys-to-list p*) = *keys p*
by (*simp add*: *keys-to-list-def set-pps-to-list*)

lemma *length-keys-to-list*: *length* (*keys-to-list p*) = *card* (*keys p*)

by (*simp only*: *keys-to-list-def length-pps-to-list*)

lemma *keys-to-list-zero* [*simp*]: *keys-to-list 0* = []
  by (*simp add*: *keys-to-list-def pps-to-list-def*)

lemma *Keys-to-list-Nil* [*simp*]: *Keys-to-list* [] = []
  by (*simp add*: *Keys-to-list-def*)

lemma *set-Keys-to-list*: *set* (*Keys-to-list ps*) = *Keys* (*set ps*)
proof −
  have *set* (*Keys-to-list ps*) = ($\bigcup$*p*∈*set ps*. *set* (*keys-to-list p*)) ∪ *set* []
    unfolding *Keys-to-list-def* **by** (*rule set-fold*, *simp only*: *set-merge-wrt*)
  also have ... = *Keys* (*set ps*) **by** (*simp add*: *Keys-def set-keys-to-list*)
  finally show *?thesis* **.**
qed

lemma *Keys-to-list-sorted-wrt-aux*:
  assumes *sorted-wrt* ($\succ_t$) *ts*
  shows *sorted-wrt* ($\succ_t$) (*fold* (λ*p ts*. *merge-wrt* ($\succ_t$) (*keys-to-list p*) *ts*) *ps ts*)
  using *assms*
proof (*induct ps arbitrary*: *ts*)
  case *Nil*
  thus *?case* **by** *simp*
next
  case (*Cons p ps*)
  show *?case*
  proof (*simp only*: *fold.simps o-def*, *rule Cons*(*1*), *rule sorted-merge-wrt*)
    show *transp* ($\succ_t$) **unfolding** *transp-def* **by** *fastforce*
  next
    fix *x y* :: *'t*
    assume $x \neq y$
    thus $x \succ_t y \lor y \succ_t x$ **by** *auto*
  next
    show *sorted-wrt* ($\succ_t$) (*keys-to-list p*) **unfolding** *keys-to-list-def*
      by (*fact pps-to-list-sorted-wrt*)
  qed *fact*
qed

corollary *Keys-to-list-sorted-wrt*: *sorted-wrt* ($\succ_t$) (*Keys-to-list ps*)
  unfolding *Keys-to-list-def*
proof (*rule Keys-to-list-sorted-wrt-aux*)
  show *sorted-wrt* ($\succ_t$) [] **by** *simp*
qed

corollary *distinct-Keys-to-list*: *distinct* (*Keys-to-list ps*)
proof (*rule distinct-sorted-wrt-irrefl*)
  show *irreflp* ($\succ_t$) **by** (*simp add*: *irreflp-def*)
next
  show *transp* ($\succ_t$) **unfolding** *transp-def* **by** *fastforce*

next
  show *sorted-wrt* $(\succ_t)$ (*Keys-to-list ps*) **by** (*fact Keys-to-list-sorted-wrt*)
qed

lemma *length-Keys-to-list*: *length* (*Keys-to-list ps*) = *card* (*Keys* (*set ps*))
proof –
  from *distinct-Keys-to-list* **have** *card* (*set* (*Keys-to-list ps*)) = *length* (*Keys-to-list ps*)
    by (*rule distinct-card*)
  thus *?thesis* **by** (*simp only*: *set-Keys-to-list*)
qed

lemma *Keys-to-list-eq-pps-to-list*: *Keys-to-list ps* = *pps-to-list* (*Keys* (*set ps*))
  using - *Keys-to-list-sorted-wrt distinct-Keys-to-list pps-to-list-sorted-wrt distinct-pps-to-list*
proof (*rule sorted-wrt-distinct-set-unique*)
  show *antisymp* $(\succ_t)$ **unfolding** *antisymp-def* **by** *fastforce*
next
  from *finite-set* **have** *fin*: *finite* (*Keys* (*set ps*)) **by** (*rule finite-Keys*)
  show *set* (*Keys-to-list ps*) = *set* (*pps-to-list* (*Keys* (*set ps*)))
    by (*simp add*: *set-Keys-to-list set-pps-to-list*[*OF fin*])
qed

10.10 Multiplication

lemma *in-keys-mult-scalar-le*:
  assumes $v \in$ *keys* $(p \odot q)$
  shows $v \preceq_t$ *punit.lt* $p \oplus$ *lt* q
proof –
  from *assms* **obtain** t u **where** $t \in$ *keys* p **and** $u \in$ *keys* q **and** $v = t \oplus u$
    by (*rule in-keys-mult-scalarE*)
  from ‹$t \in$ *keys* p› **have** $t \preceq$ *punit.lt* p **by** (*rule punit.lt-max-keys*)
  from ‹$u \in$ *keys* q› **have** $u \preceq_t$ *lt* q **by** (*rule lt-max-keys*)
  hence $v \preceq_t t \oplus$ *lt* q **unfolding** ‹$v = t \oplus u$› **by** (*rule splus-mono*)
  also from ‹$t \preceq$ *punit.lt* p› **have** ... $\preceq_t$ *punit.lt* $p \oplus$ *lt* q **by** (*rule splus-mono-left*)
  finally show *?thesis* **.**
qed

lemma *in-keys-mult-scalar-ge*:
  assumes $v \in$ *keys* $(p \odot q)$
  shows *punit.tt* $p \oplus$ *tt* $q \preceq_t v$
proof –
  from *assms* **obtain** t u **where** $t \in$ *keys* p **and** $u \in$ *keys* q **and** $v = t \oplus u$
    by (*rule in-keys-mult-scalarE*)
  from ‹$t \in$ *keys* p› **have** *punit.tt* $p \preceq t$ **by** (*rule punit.tt-min-keys*)
  from ‹$u \in$ *keys* q› **have** *tt* $q \preceq_t u$ **by** (*rule tt-min-keys*)
  hence *punit.tt* $p \oplus$ *tt* $q \preceq_t$ *punit.tt* $p \oplus u$ **by** (*rule splus-mono*)
  also from ‹*punit.tt* $p \preceq t$› **have** ... $\preceq_t v$ **unfolding** ‹$v = t \oplus u$› **by** (*rule splus-mono-left*)
  finally show *?thesis* **.**

qed

lemma (**in** *ordered-term*) *lookup-mult-scalar-lt-lt*:
lookup ($p \odot q$) (*punit.lt* $p \oplus$ *lt* q) = *punit.lc* $p *$ *lc* q
proof (*induct p rule*: *punit.poly-mapping-tail-induct*)
case *0*
show *?case* **by** *simp*
next
case *step*: (*tail p*)
from *step*(*1*) **have** *punit.lc* $p \neq 0$ **by** (*rule punit.lc-not-0*)
let *?t* = *punit.lt* $p \oplus$ *lt* q
show *?case*
proof (*cases is-monomial p*)
case *True*
then obtain *c t* **where** $c \neq 0$ **and** *punit.lt* $p = t$ **and** *punit.lc* $p = c$ **and** *p-eq*: $p =$ *monomial c t*
by (*rule punit.is-monomial-monomial-ordered*)
hence $p \odot q$ = *monom-mult* (*punit.lc p*) (*punit.lt p*) *q* **by** (*simp add*: *mult-scalar-monomial*)
thus *?thesis* **by** (*simp add*: *lookup-monom-mult-plus lc-def*)
next
case *False*
have *punit.lt* (*punit.tail p*) $\neq$ *punit.lt p*
proof (*simp add*: *punit.tail-def punit.lt-lower-eq-iff*, *rule*)
assume *punit.lt* $p = 0$
have *keys* $p \subseteq \{$*punit.lt p*$\}$
proof (*rule*, *simp*)
fix *s*
assume $s \in$ *keys p*
hence $s \preceq$ *punit.lt p* **by** (*rule punit.lt-max-keys*)
moreover have *punit.lt* $p \preceq s$ **unfolding** ‹*punit.lt* $p = 0$› **by** (*rule zero-min*)
ultimately show $s =$ *punit.lt p* **by** *simp*
qed
hence *card* (*keys p*) $= 0 \vee$ *card* (*keys p*) $= 1$ **using** *subset-singletonD* **by** *fastforce*
thus *False*
proof
assume *card* (*keys p*) = *0*
hence $p = 0$ **by** (*meson card-0-eq keys-eq-empty finite-keys*)
with *step*(*1*) **show** *False* **..**
next
assume *card* (*keys p*) = *1*
with *False* **show** *False* **unfolding** *is-monomial-def* **..**
qed
qed
with *punit.lt-lower*[*of p punit.lt p*] **have** *punit.lt* (*punit.tail p*) $\prec$ *punit.lt p*
by (*simp add*: *punit.tail-def*)
have *eq*: *lookup* ((*punit.tail p*) $\odot$ *q*) *?t* = *0*
proof (*rule ccontr*)
assume *lookup* ((*punit.tail p*) $\odot$ *q*) *?t* $\neq 0$

hence *?t* $\preceq_t$ *punit.lt* (*punit.tail p*) ⊕ *lt q*
by (*meson in-keys-mult-scalar-le lookup-not-eq-zero-eq-in-keys*)
hence *punit.lt p* ⪯ *punit.lt* (*punit.tail p*) **by** (*rule ord-term-canc-left*)
also have ... ≺ *punit.lt p* **by** *fact*
finally show *False* **..**
qed
from *step*(*2*) **have** *lookup* (*monom-mult* (*punit.lc p*) (*punit.lt p*) *q*) *?t* = *punit.lc p* ∗ *lc q*
by (*simp only*: *lookup-monom-mult-plus lc-def*)
thus *?thesis* **by** (*simp add*: *mult-scalar-tail-rec-left*[*of p q*] *lookup-add eq*)
qed
qed

lemma *lookup-mult-scalar-tt-tt*: *lookup* (*p* ⊙ *q*) (*punit.tt p* ⊕ *tt q*) = *punit.tc p* ∗ *tc q*
proof (*induct p rule*: *punit.poly-mapping-tail-induct*)
case *0*
show *?case* **by** *simp*
next
case *step*: (*tail p*)
from *step*(*1*) **have** *punit.lc p* ≠ *0* **by** (*rule punit.lc-not-0*)
let *?t* = *punit.tt p* ⊕ *tt q*
show *?case*
proof (*cases is-monomial p*)
case *True*
then obtain *c t* **where** *c* ≠ *0* **and** *punit.lt p* = *t* **and** *punit.lc p* = *c* **and** *p-eq*: *p* = *monomial c t*
by (*rule punit.is-monomial-monomial-ordered*)
from ‹*c* ≠ *0*› **have** *punit.tt p* = *t* **and** *punit.tc p* = *c* **by** (*simp-all add*: *p-eq punit.tt-monomial*)
with *p-eq* **have** *p* ⊙ *q* = *monom-mult* (*punit.tc p*) (*punit.tt p*) *q* **by** (*simp add*: *mult-scalar-monomial*)
thus *?thesis* **by** (*simp add*: *lookup-monom-mult-plus tc-def*)
next
case *False*
from *step*(*1*) **have** *keys p* ≠ {} **by** *simp*
with *finite-keys* **have** *card* (*keys p*) ≠ *0* **by** *auto*
with *False* **have** *2* ≤ *card* (*keys p*) **unfolding** *is-monomial-def* **by** *linarith*
then obtain *s t* **where** *s* ∈ *keys p* **and** *t* ∈ *keys p* **and** *s* ≺ *t*
by (*metis* (*mono-tags*, *lifting*) *card.empty card.infinite card-insert-disjoint card-mono empty-iff*
finite.emptyI insertCI lessI not-less numeral-2-eq-2 ordered-powerprod-lin.infinite-growing
ordered-powerprod-lin.neqE preorder-class.less-le-trans subsetI)
from *this*(*1*) *this*(*3*) **have** *punit.tt p* ≺ *t* **by** (*rule punit.tt-less*)
also from ‹*t* ∈ *keys p*› **have** *t* ⪯ *punit.lt p* **by** (*rule punit.lt-max-keys*)
finally have *punit.tt p* ≺ *punit.lt p* **.**
hence *tt-tail*: *punit.tt* (*punit.tail p*) = *punit.tt p* **and** *tc-tail*: *punit.tc* (*punit.tail p*) = *punit.tc p*
unfolding *punit.tail-def* **by** (*rule punit.tt-lower*, *rule punit.tc-lower*)

have *eq*: *lookup* (*monom-mult* (*punit.lc p*) (*punit.lt p*) *q*) *?t* = *0*
proof (*rule ccontr*)
assume *lookup* (*monom-mult* (*punit.lc p*) (*punit.lt p*) *q*) *?t* ≠ *0*
hence *punit.lt p* ⊕ *tt q* $\preceq_t$ *?t*
by (*meson in-keys-iff in-keys-monom-mult-ge*)
hence *punit.lt p* ⪯ *punit.tt p* **by** (*rule ord-term-canc-left*)
also have ... ≺ *punit.lt p* **by** *fact*
finally show *False* **..**
qed
from *step*(*2*) **have** *lookup* (*punit.tail p* ⊙ *q*) *?t* = *punit.tc p* ∗ *tc q* **by** (*simp only*: *tt-tail tc-tail*)
thus *?thesis* **by** (*simp add*: *mult-scalar-tail-rec-left*[*of p q*] *lookup-add eq*)
qed
qed

lemma *lt-mult-scalar*:
assumes *p* ≠ *0* **and** *q* ≠ (*0*::*'t* $\Rightarrow_0$ *'b*::*semiring-no-zero-divisors*)
shows *lt* (*p* ⊙ *q*) = *punit.lt p* ⊕ *lt q*
proof (*rule lt-eqI-keys*, *simp only*: *in-keys-iff lookup-mult-scalar-lt-lt*)
from *assms*(*1*) **have** *punit.lc p* ≠ *0* **by** (*rule punit.lc-not-0*)
moreover from *assms*(*2*) **have** *lc q* ≠ *0* **by** (*rule lc-not-0*)
ultimately show *punit.lc p* ∗ *lc q* ≠ *0* **by** *simp*
qed (*rule in-keys-mult-scalar-le*)

lemma *tt-mult-scalar*:
assumes *p* ≠ *0* **and** *q* ≠ (*0*::*'t* $\Rightarrow_0$ *'b*::*semiring-no-zero-divisors*)
shows *tt* (*p* ⊙ *q*) = *punit.tt p* ⊕ *tt q*
proof (*rule tt-eqI*, *simp only*: *in-keys-iff lookup-mult-scalar-tt-tt*)
from *assms*(*1*) **have** *punit.tc p* ≠ *0* **by** (*rule punit.tc-not-0*)
moreover from *assms*(*2*) **have** *tc q* ≠ *0* **by** (*rule tc-not-0*)
ultimately show *punit.tc p* ∗ *tc q* ≠ *0* **by** *simp*
qed (*rule in-keys-mult-scalar-ge*)

lemma *lc-mult-scalar*: *lc* (*p* ⊙ *q*) = *punit.lc p* ∗ *lc* (*q*::*'t* $\Rightarrow_0$ *'b*::*semiring-no-zero-divisors*)
proof (*cases p* = *0*)
case *True*
thus *?thesis* **by** (*simp add*: *lc-def*)
next
case *False*
show *?thesis*
proof (*cases q* = *0*)
case *True*
thus *?thesis* **by** (*simp add*: *lc-def*)
next
case *False*
with ‹*p* ≠ *0*› **show** *?thesis* **by** (*simp add*: *lc-def lt-mult-scalar lookup-mult-scalar-lt-lt*)
qed
qed

lemma *tc-mult-scalar*: *tc* (*p* ⊙ *q*) = *punit.tc p* ∗ *tc* (*q*::′*t* ⇒₀ ′*b*::*semiring-no-zero-divisors*)
proof (*cases p* = *0*)
  case *True*
  thus *?thesis* **by** (*simp add*: *tc-def*)
next
  case *False*
  show *?thesis*
  proof (*cases q* = *0*)
    case *True*
    thus *?thesis* **by** (*simp add*: *tc-def*)
  next
    case *False*
   with ‹*p* ≠ *0*› **show** *?thesis* **by** (*simp add*: *tc-def tt-mult-scalar lookup-mult-scalar-tt-tt*)
  qed
qed

lemma *mult-scalar-not-zero*:
  assumes *p* ≠ *0* **and** *q* ≠ (*0*::′*t* ⇒₀ ′*b*::*semiring-no-zero-divisors*)
  shows *p* ⊙ *q* ≠ *0*
proof
  assume *p* ⊙ *q* = *0*
  hence *0* = *lc* (*p* ⊙ *q*) **by** (*simp add*: *lc-def*)
  also have ... = *punit.lc p* ∗ *lc q* **by** (*rule lc-mult-scalar*)
  finally have *punit.lc p* ∗ *lc q* = *0* **by** *simp*
  moreover from *assms*(*1*) **have** *punit.lc p* ≠ *0* **by** (*rule punit.lc-not-0*)
  moreover from *assms*(*2*) **have** *lc q* ≠ *0* **by** (*rule lc-not-0*)
  ultimately show *False* **by** *simp*
qed

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** ‹$\preceq_t$› *50*)
and *ord-term-strict*::′*t* ⇒ ′*t* ⇒ *bool* (**infixl** ‹$\prec_t$› *50*)
begin

sublocale *gd-powerprod* **..**

lemma *adds-term-antisym*:
assumes *u* $adds_t$ *v* **and** *v* $adds_t$ *u*
shows *u* = *v*
using *assms* **unfolding** *adds-term-def* **using** *adds-antisym* **by** (*metis term-of-pair-pair*)

definition *dgrad-p-set* :: (′*a* ⇒ *nat*) ⇒ *nat* ⇒ (′*t* $\Rightarrow_0$ ′*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* $\Rightarrow_0$ ′*b*) *set*) ⇒ ((′*t* $\Rightarrow_0$ ′*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*)
by (*auto simp add*: *dgrad-p-set-def dgrad-set-def*)

lemma *dgrad-p-setI* [*intro*]:
assumes ⋀*v*. *v* ∈ *keys p* ⟹ *d* (*pp-of-term v*) ≤ *m*
shows *p* ∈ *dgrad-p-set d m*
using *assms* **by** (*auto simp*: *in-dgrad-p-set-iff*)

lemma *dgrad-p-setD*:
assumes *p* ∈ *dgrad-p-set d m* **and** *v* ∈ *keys p*
shows *d* (*pp-of-term v*) ≤ *m*
using *assms* **by** (*simp only*: *in-dgrad-p-set-iff*)

lemma *zero-in-dgrad-p-set*: *0* ∈ *dgrad-p-set d m*
by (*rule*, *simp*)

lemma *dgrad-p-set-zero* [*simp*]: *dgrad-p-set* (λ-. *0*) *m* = *UNIV*
  by *auto*

lemma *subset-dgrad-p-set-zero*: *F* ⊆ *dgrad-p-set* (λ-. *0*) *m*
  by *simp*

lemma *dgrad-p-set-subset*:
  assumes *m* ≤ *n*
  shows *dgrad-p-set d m* ⊆ *dgrad-p-set d n*
  using *assms* **by** (*auto simp*: *dgrad-p-set-def dgrad-set-def*)

lemma *dgrad-p-setD-lp*:
  assumes *p* ∈ *dgrad-p-set d m* **and** *p* ≠ *0*
  shows *d* (*lp p*) ≤ *m*
  by (*rule dgrad-p-setD*, *fact*, *rule lt-in-keys*, *fact*)

lemma *dgrad-p-set-exhaust-expl*:
  assumes *finite F*
  shows *F* ⊆ *dgrad-p-set d* (*Max* (*d* ' *pp-of-term* ' *Keys F*))
proof
  fix *f*
  assume *f* ∈ *F*
  from *assms* **have** *finite* (*Keys F*) **by** (*rule finite-Keys*)
  have *fin*: *finite* (*d* ' *pp-of-term* ' *Keys F*) **by** (*intro finite-imageI*, *fact*)
  show *f* ∈ *dgrad-p-set d* (*Max* (*d* ' *pp-of-term* ' *Keys F*))
  proof (*rule dgrad-p-setI*)
    fix *v*
    assume *v* ∈ *keys f*
    from *this* ‹*f* ∈ *F*› **have** *v* ∈ *Keys F* **by** (*rule in-KeysI*)
    hence *d* (*pp-of-term v*) ∈ *d* ' *pp-of-term* ' *Keys F* **by** *simp*
     with *fin* **show** *d* (*pp-of-term v*) ≤ *Max* (*d* ' *pp-of-term* ' *Keys F*) **by** (*rule Max-ge*)
  qed
qed

lemma *dgrad-p-set-exhaust*:
  assumes *finite F*
  obtains *m* **where** *F* ⊆ *dgrad-p-set d m*
proof
  from *assms* **show** *F* ⊆ *dgrad-p-set d* (*Max* (*d* ' *pp-of-term* ' *Keys F*)) **by** (*rule dgrad-p-set-exhaust-expl*)
qed

lemma *dgrad-p-set-insert*:
  assumes *F* ⊆ *dgrad-p-set d m*
  obtains *n* **where** *m* ≤ *n* **and** *f* ∈ *dgrad-p-set d n* **and** *F* ⊆ *dgrad-p-set d n*
proof −
  have *finite* {*f*} **by** *simp*
  then obtain *m1* **where** {*f*} ⊆ *dgrad-p-set d m1* **by** (*rule dgrad-p-set-exhaust*)

hence *f* ∈ *dgrad-p-set d m1* **by** *simp*
define *n* **where** *n* = *ord-class.max m m1*
have *m* ≤ *n* **and** *m1* ≤ *n* **by** (*simp-all add*: *n-def*)
from *this*(*1*) **show** *?thesis*
proof
from ‹*m1* ≤ *n*› **have** *dgrad-p-set d m1* ⊆ *dgrad-p-set d n* **by** (*rule dgrad-p-set-subset*)
with ‹*f* ∈ *dgrad-p-set d m1*› **show** *f* ∈ *dgrad-p-set d n* **..**
next
from ‹*m* ≤ *n*› **have** *dgrad-p-set d m* ⊆ *dgrad-p-set d n* **by** (*rule dgrad-p-set-subset*)
with *assms* **show** *F* ⊆ *dgrad-p-set d n* **by** (*rule subset-trans*)
qed
qed

lemma *dgrad-p-set-leI*:
assumes ⋀*f*. *f* ∈ *F* ⟹ *dgrad-p-set-le d* {*f*} *G*
shows *dgrad-p-set-le d F G*
unfolding *dgrad-p-set-le-def dgrad-set-le-def*
proof
fix *s*
assume *s* ∈ *pp-of-term* ' *Keys F*
then obtain *v* **where** *v* ∈ *Keys F* **and** *s* = *pp-of-term v* **..**
from *this*(*1*) **obtain** *f* **where** *f* ∈ *F* **and** *v* ∈ *keys f* **by** (*rule in-KeysE*)
from *this*(*2*) **have** *s* ∈ *pp-of-term* ' *Keys* {*f*} **by** (*simp add*: ‹*s* = *pp-of-term v*› *Keys-insert*)
from ‹*f* ∈ *F*› **have** *dgrad-p-set-le d* {*f*} *G* **by** (*rule assms*)
from *this* ‹*s* ∈ *pp-of-term* ' *Keys* {*f*}› **show** ∃ *t*∈*pp-of-term* ' *Keys G*. *d s* ≤ *d t*
unfolding *dgrad-p-set-le-def dgrad-set-le-def* **..**
qed

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*
using *assms* **unfolding** *dgrad-p-set-le-def* **by** (*rule dgrad-set-le-trans*)

lemma *dgrad-p-set-le-subset*:
assumes *F* ⊆ *G*
shows *dgrad-p-set-le d F G*
unfolding *dgrad-p-set-le-def* **by** (*rule dgrad-set-le-subset*, *rule image-mono*, *rule Keys-mono*, *fact*)

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*
using *assms* **by** (*simp add*: *dgrad-p-set-le-def Keys-insert dgrad-set-le-Un image-Un*)

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*
using *assms* **by** (*simp add*: *dgrad-p-set-le-def Keys-insert dgrad-set-le-Un image-Un*)

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*
using *assms* **by** (*auto simp*: *dgrad-p-set-le-def dgrad-set-le-def Keys-Un*)

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
fix *f*
assume *f* ∈ *F*
show *f* ∈ *dgrad-p-set d m*
proof (*rule dgrad-p-setI*)
fix *v*
assume *v* ∈ *keys f*
from *this* ‹*f* ∈ *F*› **have** *v* ∈ *Keys F* **by** (*rule in-KeysI*)
hence *pp-of-term v* ∈ *pp-of-term* ‘ *Keys F* **by** *simp*
with *assms*(*1*) **obtain** *s* **where** *s* ∈ *pp-of-term* ‘ *Keys G* **and** *d* (*pp-of-term v*) ≤ *d s*
unfolding *dgrad-p-set-le-def* **by** (*rule dgrad-set-leE*)
from *this*(*1*) **obtain** *u* **where** *u* ∈ *Keys G* **and** *s*: *s* = *pp-of-term u* **..**
from *this*(*1*) **obtain** *g* **where** *g* ∈ *G* **and** *u* ∈ *keys g* **by** (*rule in-KeysE*)
from *this*(*1*) *assms*(*2*) **have** *g* ∈ *dgrad-p-set d m* **..**
from *this* ‹*u* ∈ *keys g*› **have** *d s* ≤ *m* **unfolding** *s* **by** (*rule dgrad-p-setD*)
with ‹*d* (*pp-of-term v*) ≤ *d s*› **show** *d* (*pp-of-term v*) ≤ *m* **by** (*rule le-trans*)
qed
qed

lemma *dgrad-p-set-le-except*: *dgrad-p-set-le d* {*except p S*} {*p*}
by (*auto simp add*: *dgrad-p-set-le-def Keys-insert keys-except intro*: *dgrad-set-le-subset*)

lemma *dgrad-p-set-le-tail*: *dgrad-p-set-le d* {*tail p*} {*p*}
by (*simp only*: *tail-def lower-def*, *fact dgrad-p-set-le-except*)

lemma *dgrad-p-set-le-plus*: *dgrad-p-set-le d* {*p* + *q*} {*p*, *q*}
by (*simp add*: *dgrad-p-set-le-def Keys-insert*, *rule dgrad-set-le-subset*, *rule image-mono*, *fact Poly-Mapping.keys-add*)

lemma *dgrad-p-set-le-uminus*: *dgrad-p-set-le d* {−*p*} {*p*}
by (*simp add*: *dgrad-p-set-le-def Keys-insert keys-uminus*, *fact dgrad-set-le-refl*)

lemma *dgrad-p-set-le-minus*: *dgrad-p-set-le d* {*p* − *q*} {*p*, *q*}
by (*simp add*: *dgrad-p-set-le-def Keys-insert*, *rule dgrad-set-le-subset*, *rule image-mono*, *fact keys-minus*)

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 (*rule dgrad-set-leI*)
  fix *s*
  assume *s* ∈ *pp-of-term ‘ keys* (*monom-mult c t p*)
  with *keys-monom-mult-subset* **have** *s* ∈ *pp-of-term ‘* ((⊕) *t ‘ keys p*) **by** *fastforce*
  then obtain *v* **where** *v* ∈ *keys p* **and** *s*: *s* = *pp-of-term* (*t* ⊕ *v*) **by** *fastforce*
  have *d s* = *ord-class.max* (*d t*) (*d* (*pp-of-term v*))
    by (*simp only*: *s pp-of-term-splus dickson-gradingD1*[*OF assms*(*1*)])
  hence *d s* = *d t* ∨ *d s* = *d* (*pp-of-term v*) **by** *auto*
  thus ∃ *t*∈*insert t* (*pp-of-term ‘ keys p*). *d s* ≤ *d t*
  proof
    assume *d s* = *d t*
    thus *?thesis* **by** *simp*
  next
    assume *d s* = *d* (*pp-of-term v*)
    show *?thesis*
    proof
      from ‹*d s* = *d* (*pp-of-term v*)› **show** *d s* ≤ *d* (*pp-of-term v*) **by** *simp*
    next
      from ‹*v* ∈ *keys p*› **show** *pp-of-term v* ∈ *insert t* (*pp-of-term ‘ keys p*) **by** *simp*
    qed
  qed
qed

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 −
  from *dgrad-p-set-le-plus* **have** {*p* + *q*} ⊆ *dgrad-p-set d m*
  proof (*rule dgrad-p-set-le-dgrad-p-set*)
    from *assms* **show** {*p*, *q*} ⊆ *dgrad-p-set d m* **by** *simp*
  qed
  thus *?thesis* **by** *simp*
qed

lemma *dgrad-p-set-closed-uminus*:
  assumes *p* ∈ *dgrad-p-set d m*
  shows −*p* ∈ *dgrad-p-set d m*
proof −
  from *dgrad-p-set-le-uminus* **have** {−*p*} ⊆ *dgrad-p-set d m*
  proof (*rule dgrad-p-set-le-dgrad-p-set*)
    from *assms* **show** {*p*} ⊆ *dgrad-p-set d m* **by** *simp*
  qed
  thus *?thesis* **by** *simp*
qed

lemma *dgrad-p-set-closed-minus*:
  assumes $p \in$ *dgrad-p-set d m* **and** $q \in$ *dgrad-p-set d m*
  shows $p - q \in$ *dgrad-p-set d m*
proof −
  from *dgrad-p-set-le-minus* **have** $\{p - q\} \subseteq$ *dgrad-p-set d m*
  proof (*rule dgrad-p-set-le-dgrad-p-set*)
    from *assms* **show** $\{p, q\} \subseteq$ *dgrad-p-set d m* **by** *simp*
  qed
  thus *?thesis* **by** *simp*
qed

lemma *dgrad-p-set-closed-monom-mult*:
  assumes *dickson-grading d* **and** $d\ t \leq m$ **and** $p \in$ *dgrad-p-set d m*
  shows *monom-mult c t p* $\in$ *dgrad-p-set d m*
proof (*rule dgrad-p-setI*)
  fix *v*
  assume $v \in$ *keys* (*monom-mult c t p*)
  hence *pp-of-term v* $\in$ *pp-of-term* ‘ *keys* (*monom-mult c t p*) **by** *simp*
  with *dgrad-set-le-monom-mult*[*OF assms*(*1*)] **obtain** *s* **where** $s \in$ *insert t* (*pp-of-term* ‘ *keys p*)
    and *d* (*pp-of-term v*) $\leq$ *d s* **by** (*rule dgrad-set-leE*)
  from *this*(*1*) **have** $s = t \vee s \in$ *pp-of-term* ‘ *keys p* **by** *simp*
  thus *d* (*pp-of-term v*) $\leq m$
  proof
    assume $s = t$
    with ‹*d* (*pp-of-term v*) $\leq$ *d s*› *assms*(*2*) **show** *?thesis* **by** *simp*
  next
    assume $s \in$ *pp-of-term* ‘ *keys p*
    then obtain *u* **where** $u \in$ *keys p* **and** $s =$ *pp-of-term u* **..**
    from *assms*(*3*) *this*(*1*) **have** $d\ s \leq m$ **unfolding** ‹$s =$ *pp-of-term u*› **by** (*rule dgrad-p-setD*)
    with ‹*d* (*pp-of-term v*) $\leq$ *d s*› **show** *?thesis* **by** (*rule le-trans*)
  qed
qed

lemma *dgrad-p-set-closed-monom-mult-zero*:
  assumes $p \in$ *dgrad-p-set d m*
  shows *monom-mult c 0 p* $\in$ *dgrad-p-set d m*
proof (*rule dgrad-p-setI*)
  fix *v*
  assume $v \in$ *keys* (*monom-mult c 0 p*)
  hence *pp-of-term v* $\in$ *pp-of-term* ‘ *keys* (*monom-mult c 0 p*) **by** *simp*
  then obtain *u* **where** $u \in$ *keys* (*monom-mult c 0 p*) **and** *eq*: *pp-of-term v* = *pp-of-term u* **..**
  from *this*(*1*) **have** $u \in$ *keys p* **by** (*metis keys-monom-multE splus-zero*)
  with *assms* **have** *d* (*pp-of-term u*) $\leq m$ **by** (*rule dgrad-p-setD*)
  thus *d* (*pp-of-term v*) $\leq m$ **by** (*simp only*: *eq*)
qed

lemma *dgrad-p-set-closed-except*:
assumes $p \in$ *dgrad-p-set d m*
shows *except p S* $\in$ *dgrad-p-set d m*
by (*rule dgrad-p-setI*, *rule dgrad-p-setD*, *rule assms*, *simp add*: *keys-except*)

lemma *dgrad-p-set-closed-tail*:
assumes $p \in$ *dgrad-p-set d m*
shows *tail p* $\in$ *dgrad-p-set d m*
unfolding *tail-def lower-def* **using** *assms* **by** (*rule dgrad-p-set-closed-except*)

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* $\in$ *dgrad-set d m* $\wedge$ *component-of-term t* $\in$ *K*}
(**is** *almost-full-on - ?A*)
proof (*rule almost-full-onI*)
fix *seq* :: *nat* $\Rightarrow$ *'t*
assume $*$: $\forall$ *i. seq i* $\in$ *?A*
define *seq'* **where** *seq'* = (λ*i.* (*pp-of-term* (*seq i*), *component-of-term* (*seq i*)))
have *pp-of-term* ‘ *?A* $\subseteq$ {*x. d x* $\leq$ *m*} **by** (*auto dest*: *dgrad-setD*)
moreover from *assms*(*1*) **have** *almost-full-on* (*adds*) {*x. d x* $\leq$ *m*} **by** (*rule dickson-gradingD2*)
ultimately have *almost-full-on* (*adds*) (*pp-of-term* ‘ *?A*) **by** (*rule almost-full-on-subset*)
moreover have *almost-full-on* (=) (*component-of-term* ‘ *?A*)
proof (*rule eq-almost-full-on-finite-set*)
have *component-of-term* ‘ *?A* $\subseteq$ *K* **by** *blast*
thus *finite* (*component-of-term* ‘ *?A*) **using** *assms*(*2*) **by** (*rule finite-subset*)
qed
ultimately have *almost-full-on* (*prod-le* (*adds*) (=)) (*pp-of-term* ‘ *?A* $\times$ *component-of-term* ‘ *?A*)
by (*rule almost-full-on-Sigma*)
moreover from $*$ **have** $\bigwedge$*i. seq' i* $\in$ *pp-of-term* ‘ *?A* $\times$ *component-of-term* ‘ *?A* **by** (*simp add*: *seq'-def*)
ultimately obtain *i j* **where** *i* < *j* **and** *prod-le* (*adds*) (=) (*seq' i*) (*seq' j*)
by (*rule almost-full-onD*)
from *this*(*2*) **have** *seq i* $adds_t$ *seq j* **by** (*simp add*: *seq'-def prod-le-def adds-term-def*)
with ‹*i* < *j*› **show** *good* ($adds_t$) *seq* **by** (*rule goodI*)
qed

corollary *Dickson-termE*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ‘ *range* (*f*::*nat* $\Rightarrow$ *'t*))
and *pp-of-term* ‘ *range f* $\subseteq$ *dgrad-set d m*
obtains *i j* **where** *i* < *j* **and** *f i* $adds_t$ *f j*
proof −
let *?A* = {*t. pp-of-term t* $\in$ *dgrad-set d m* $\wedge$ *component-of-term t* $\in$ *component-of-term* ‘ *range f*}
from *assms*(*1*, *2*) **have** *almost-full-on* ($adds_t$) *?A* **by** (*rule Dickson-term*)

moreover from *assms*(*3*) **have** $\bigwedge i.\ f\ i \in\ ?A$ **by** *blast*
ultimately obtain *i j* **where** $i < j$ **and** $f\ i\ adds_t\ f\ j$ **by** (*rule almost-full-onD*)
thus *?thesis* **..**
qed

lemma *ex-finite-adds-term*:
assumes *dickson-grading d* **and** *finite* (*component-of-term* ' *S*) **and** *pp-of-term* ' $S \subseteq$ *dgrad-set d m*
obtains *T* **where** *finite T* **and** $T \subseteq S$ **and** $\bigwedge s.\ s \in S \Longrightarrow (\exists t \in T.\ t\ adds_t\ s)$
proof −
let $?A = \{t.$ *pp-of-term* $t \in$ *dgrad-set d m* $\wedge$ *component-of-term* $t \in$ *component-of-term* ' $S\}$
have *reflp* ((*adds*$_t$)::'*t* ⇒ -) **by** (*simp add*: *reflp-def adds-term-refl*)
moreover have *almost-full-on* (*adds*$_t$) *S*
proof (*rule almost-full-on-subset*)
from *assms*(*3*) **show** $S \subseteq\ ?A$ **by** *blast*
next
from *assms*(*1*, *2*) **show** *almost-full-on* (*adds*$_t$) *?A* **by** (*rule Dickson-term*)
qed
ultimately obtain *T* **where** *finite T* **and** $T \subseteq S$ **and** $\bigwedge s.\ s \in S \Longrightarrow (\exists t \in T.\ t\ adds_t\ s)$
by (*rule almost-full-on-finite-subsetE*, *blast*)
thus *?thesis* **..**
qed

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* ∧ $v \prec_t u$)

definition *dickson-less-p* :: ('*a* ⇒ *nat*) ⇒ *nat* ⇒ ('*t* $\Rightarrow_0$ '*b*) ⇒ ('*t* $\Rightarrow_0$ '*b*::*zero*) ⇒ *bool*
where *dickson-less-p d m p q* ⟷ ({*p*, *q*} ⊆ *dgrad-p-set d m* ∧ $p \prec_p q$)

lemma *dickson-less-vI*:
assumes *d* (*pp-of-term v*) ≤ *m* **and** *d* (*pp-of-term u*) ≤ *m* **and** $v \prec_t u$
shows *dickson-less-v d m v u*
using *assms* **by** (*simp add*: *dickson-less-v-def*)

lemma *dickson-less-vD1*:
assumes *dickson-less-v d m v u*
shows *d* (*pp-of-term v*) ≤ *m*
using *assms* **by** (*simp add*: *dickson-less-v-def*)

lemma *dickson-less-vD2*:
assumes *dickson-less-v d m v u*
shows *d* (*pp-of-term u*) ≤ *m*
using *assms* **by** (*simp add*: *dickson-less-v-def*)

lemma *dickson-less-vD3*:
  assumes *dickson-less-v d m v u*
  shows $v \prec_t u$
  using *assms* **by** (*simp add*: *dickson-less-v-def*)

lemma *dickson-less-v-irrefl*: ¬ *dickson-less-v d m v v*
  by (*simp add*: *dickson-less-v-def*)

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*
  using *assms* **by** (*auto simp add*: *dickson-less-v-def*)

lemma *wf-dickson-less-v-aux1*:
  assumes *dickson-grading d* **and** ⋀*i*::*nat*. *dickson-less-v d m* (*seq* (*Suc i*)) (*seq i*)
  obtains *i* **where** ⋀*j*. *j* > *i* ⟹ *component-of-term* (*seq j*) < *component-of-term* (*seq i*)
proof −
  let *?Q* = *pp-of-term* ‘ *range seq*
  have *pp-of-term* (*seq 0*) ∈ *?Q* **by** *simp*
  with *wf-dickson-less*[*OF assms*(*1*)] **obtain** *t* **where** *t* ∈ *?Q* **and** ∗: ⋀*s*. *dickson-less d m s t* ⟹ *s* ∉ *?Q*
    by (*rule wfE-min*[*to-pred*], *blast*)
  from *this*(*1*) **obtain** *i* **where** *t*: *t* = *pp-of-term* (*seq i*) **by** *fastforce*
  show *?thesis*
  proof
    fix *j*
    assume *i* < *j*
    with - *assms*(*2*) **have** *dlv*: *dickson-less-v d m* (*seq j*) (*seq i*)
    proof (*rule transp-sequence*)
      from *dickson-less-v-trans* **show** *transp* (*dickson-less-v d m*) **by** (*rule transpI*)
    qed
    hence $seq\ j \prec_t seq\ i$ **by** (*rule dickson-less-vD3*)
    define *s* **where** *s* = *pp-of-term* (*seq j*)
    have *pp-of-term* (*seq j*) ∈ *?Q* **by** *simp*
    hence ¬ *dickson-less d m s t* **unfolding** *s-def* **using** ∗ **by** *blast*
    moreover from *dlv* **have** *d s* ≤ *m* **and** *d t* ≤ *m* **unfolding** *s-def t*
      by (*rule dickson-less-vD1*, *rule dickson-less-vD2*)
    ultimately have *t* ⪯ *s* **by** (*simp add*: *dickson-less-def*)
    show *component-of-term* (*seq j*) < *component-of-term* (*seq i*)
    proof (*rule ccontr*, *simp only*: *not-less*)
      assume *component-of-term* (*seq i*) ≤ *component-of-term* (*seq j*)
      with ‹*t* ⪯ *s*› **have** $seq\ i \preceq_t seq\ j$ **unfolding** *s-def t* **by** (*rule ord-termI*)
      moreover from *dlv* **have** $seq\ j \prec_t seq\ i$ **by** (*rule dickson-less-vD3*)
      ultimately show *False* **by** *simp*
    qed
  qed
qed

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*
using *assms*(*2*, *3*)
proof (*induct k arbitrary*: *seq thesis rule*: *less-induct*)
case (*less k*)
from *assms*(*1*) *less*(*2*) **obtain** *i* **where** *: $\bigwedge$*j*. *j* > *i* $\Longrightarrow$ *component-of-term* (*seq j*) < *component-of-term* (*seq i*)
by (*rule wf-dickson-less-v-aux1*, *blast*)
define *seq1* **where** *seq1* = (λ*j*. *seq* (*Suc* (*i* + *j*)))
from *less*(*3*) **show** *?case*
proof (*rule less*(*1*))
fix *j*
show *dickson-less-v d m* (*seq1* (*Suc j*)) (*seq1 j*) **by** (*simp add*: *seq1-def*, *fact less*(*2*))
next
fix *j*
show *component-of-term* (*seq1 j*) < *component-of-term* (*seq i*) **by** (*simp add*: *seq1-def* *)
qed
qed

lemma *wf-dickson-less-v*:
assumes *dickson-grading d*
shows *wfP* (*dickson-less-v d m*)
proof (*rule wfP-chain*, *rule*, *elim exE*)
fix *seq*::*nat* ⇒ *'t*
assume ∀ *i*. *dickson-less-v d m* (*seq* (*Suc i*)) (*seq i*)
hence *: $\bigwedge$*i*. *dickson-less-v d m* (*seq* (*Suc i*)) (*seq i*) **..**
with *assms* **obtain** *i* **where** **: $\bigwedge$*j*. *j* > *i* $\Longrightarrow$ *component-of-term* (*seq j*) < *component-of-term* (*seq i*)
by (*rule wf-dickson-less-v-aux1*, *blast*)
define *seq1* **where** *seq1* = (λ*j*. *seq* (*Suc* (*i* + *j*)))
from *assms* **show** *False*
proof (*rule wf-dickson-less-v-aux2*)
fix *j*
show *dickson-less-v d m* (*seq1* (*Suc j*)) (*seq1 j*) **by** (*simp add*: *seq1-def*, *fact* *)
next
fix *j*
show *component-of-term* (*seq1 j*) < *component-of-term* (*seq i*) **by** (*simp add*: *seq1-def* **)
qed
qed

lemma *dickson-less-v-zero*: *dickson-less-v* (λ-. *0*) *m* = ($\prec_t$)
by (*rule*, *rule*, *simp add*: *dickson-less-v-def*)

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*
  using *assms* **by** (*simp add*: *dickson-less-p-def*)

lemma *dickson-less-pD1*:
  assumes *dickson-less-p d m p q*
  shows $p \in$ *dgrad-p-set d m*
  using *assms* **by** (*simp add*: *dickson-less-p-def*)

lemma *dickson-less-pD2*:
  assumes *dickson-less-p d m p q*
  shows $q \in$ *dgrad-p-set d m*
  using *assms* **by** (*simp add*: *dickson-less-p-def*)

lemma *dickson-less-pD3*:
  assumes *dickson-less-p d m p q*
  shows $p \prec_p q$
  using *assms* **by** (*simp add*: *dickson-less-p-def*)

lemma *dickson-less-p-irrefl*: ¬ *dickson-less-p d m p p*
  by (*simp add*: *dickson-less-p-def*)

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*
  using *assms* **by** (*auto simp add*: *dickson-less-p-def*)

lemma *dickson-less-p-mono*:
  assumes *dickson-less-p d m p q* **and** $m \le n$
  shows *dickson-less-p d n p q*
proof –
  from *assms*(*2*) **have** *dgrad-p-set d m* ⊆ *dgrad-p-set d n* **by** (*rule dgrad-p-set-subset*)
  moreover from *assms*(*1*) **have** $p \in$ *dgrad-p-set d m* **and** $q \in$ *dgrad-p-set d m* **and** $p \prec_p q$
    by (*rule dickson-less-pD1*, *rule dickson-less-pD2*, *rule dickson-less-pD3*)
  ultimately have $p \in$ *dgrad-p-set d n* **and** $q \in$ *dgrad-p-set d n* **by** *auto*
  from *this* ‹$p \prec_p q$› **show** *?thesis* **by** (*rule dickson-less-pI*)
qed

lemma *dickson-less-p-zero*: *dickson-less-p* (λ-. *0*) $m = (\prec_p)$
  by (*rule*, *rule*, *simp add*: *dickson-less-p-def*)

lemma *wf-dickson-less-p-aux*:
  assumes *dickson-grading d*
  assumes $x \in Q$ **and** ∀ *y*∈*Q*. *y* ≠ *0* ⟶ (*y* ∈ *dgrad-p-set d m* ∧ *dickson-less-v d m* (*lt y*) *u*)
  shows ∃ *p*∈*Q*. (∀ *q*∈*Q*. ¬ *dickson-less-p d m q p*)
  using *assms*(*2*) *assms*(*3*)

proof (*induct u arbitrary*: *x Q rule*: *wfp-induct*[*OF wf-dickson-less-v*, *OF assms*(*1*)])
fix *u*::′*t* **and** *x*::′*t* ⇒₀ ′*b* **and** *Q*::(′*t* ⇒₀ ′*b*) *set*
assume *hyp*: ∀ *u0*. *dickson-less-v d m u0 u* ⟶ (∀ *x0 Q0*::(′*t* ⇒₀ ′*b*) *set*. *x0* ∈ *Q0* ⟶
(∀ *y*∈*Q0*. *y* ≠ *0* ⟶ (*y* ∈ *dgrad-p-set d m* ∧ *dickson-less-v d m* (*lt y*) *u0*)) ⟶
(∃ *p*∈*Q0*. ∀ *q*∈*Q0*. ¬ *dickson-less-p d m q p*))
assume *x* ∈ *Q*
assume ∀ *y*∈*Q*. *y* ≠ *0* ⟶ (*y* ∈ *dgrad-p-set d m* ∧ *dickson-less-v d m* (*lt y*) *u*)
hence *bounded*: ⋀*y*. *y* ∈ *Q* ⟹ *y* ≠ *0* ⟹ (*y* ∈ *dgrad-p-set d m* ∧ *dickson-less-v d m* (*lt y*) *u*) **by** *auto*
show ∃ *p*∈*Q*. ∀ *q*∈*Q*. ¬ *dickson-less-p d m q p*
proof (*cases 0* ∈ *Q*)
case *True*
show *?thesis*
proof (*rule*, *rule*, *rule*)
fix *q*::′*t* ⇒₀ ′*b*
assume *dickson-less-p d m q 0*
hence *q* ≺ₚ *0* **by** (*rule dickson-less-pD3*)
thus *False* **using** *ord-p-zero-min*[*of q*] **by** *simp*
next
from *True* **show** *0* ∈ *Q* **.**
qed
next
case *False*
define *Q1* **where** *Q1* = {*lt p* | *p*. *p* ∈ *Q*}
from ‹*x* ∈ *Q*› **have** *lt x* ∈ *Q1* **unfolding** *Q1-def* **by** *auto*
with *wf-dickson-less-v*[*OF assms*(*1*)] **obtain** *v*
where *v* ∈ *Q1* **and** *v-min-1*: ⋀*q*. *dickson-less-v d m q v* ⟹ *q* ∉ *Q1*
by (*rule wfE-min*[*to-pred*], *auto*)
have *v-min*: ⋀*q*. *q* ∈ *Q* ⟹ ¬ *dickson-less-v d m* (*lt q*) *v*
proof −
fix *q*
assume *q* ∈ *Q*
hence *lt q* ∈ *Q1* **unfolding** *Q1-def* **by** *auto*
thus ¬ *dickson-less-v d m* (*lt q*) *v* **using** *v-min-1* **by** *auto*
qed
from ‹*v* ∈ *Q1*› **obtain** *p* **where** *lt p* = *v* **and** *p* ∈ *Q* **unfolding** *Q1-def* **by** *auto*
hence *p* ≠ *0* **using** *False* **by** *auto*
with ‹*p* ∈ *Q*› **have** *p* ∈ *dgrad-p-set d m* ∧ *dickson-less-v d m* (*lt p*) *u* **by** (*rule bounded*)
hence *p* ∈ *dgrad-p-set d m* **and** *dickson-less-v d m* (*lt p*) *u* **by** *simp-all*
moreover from *this*(*1*) ‹*p* ≠ *0*› **have** *d* (*pp-of-term* (*lt p*)) ≤ *m* **by** (*rule dgrad-p-setD-lp*)
ultimately have *d* (*pp-of-term v*) ≤ *m* **by** (*simp only*: ‹*lt p* = *v*›)
define *Q2* **where** *Q2* = {*tail p* | *p*. *p* ∈ *Q* ∧ *lt p* = *v*}
from ‹*p* ∈ *Q*› ‹*lt p* = *v*› **have** *tail p* ∈ *Q2* **unfolding** *Q2-def* **by** *auto*
have ∀ *q*∈*Q2*. *q* ≠ *0* ⟶ (*q* ∈ *dgrad-p-set d m* ∧ *dickson-less-v d m* (*lt q*) (*lt*

p))
proof (*intro ballI impI*)
fix *q*
assume *q* ∈ *Q2*
then obtain *q0* **where** *q*: *q* = *tail q0* **and** *lt q0* = *lt p* **and** *q0* ∈ *Q*
using ‹*lt p* = *v*› **by** (*auto simp add*: *Q2-def*)
assume *q* ≠ *0*
hence *tail q0* ≠ *0* **using** ‹*q* = *tail q0*› **by** *simp*
hence *q0* ≠ *0* **by** *auto*
with ‹*q0* ∈ *Q*› **have** *q0* ∈ *dgrad-p-set d m* ∧ *dickson-less-v d m* (*lt q0*) *u* **by** (*rule bounded*)
hence *q0* ∈ *dgrad-p-set d m* **and** *dickson-less-v d m* (*lt q0*) *u* **by** *simp-all*
from *this*(*1*) **have** *q* ∈ *dgrad-p-set d m* **unfolding** *q* **by** (*rule dgrad-p-set-closed-tail*)
show *q* ∈ *dgrad-p-set d m* ∧ *dickson-less-v d m* (*lt q*) (*lt p*)
proof
show *dickson-less-v d m* (*lt q*) (*lt p*)
proof (*rule dickson-less-vI*)
from ‹*q* ∈ *dgrad-p-set d m*› ‹*q* ≠ *0*› **show** *d* (*pp-of-term* (*lt q*)) ≤ *m* **by** (*rule dgrad-p-setD-lp*)
next
from ‹*dickson-less-v d m* (*lt p*) *u*› **show** *d* (*pp-of-term* (*lt p*)) ≤ *m* **by** (*rule dickson-less-vD1*)
next
from *lt-tail*[*OF* ‹*tail q0* ≠ *0*›] ‹*q* = *tail q0*› ‹*lt q0* = *lt p*› **show** *lt q* $\prec_t$ *lt p* **by** *simp*
qed
qed *fact*
qed
with *hyp* ‹*dickson-less-v d m* (*lt p*) *u*› ‹*tail p* ∈ *Q2*› **have** ∃ *p*∈*Q2*. ∀ *q*∈*Q2*. ¬ *dickson-less-p d m q p*
by *blast*
then obtain *q* **where** *q* ∈ *Q2* **and** *q-min*: ∀ *r*∈*Q2*. ¬ *dickson-less-p d m r q* **..**
from ‹*q* ∈ *Q2*› **obtain** *q0* **where** *q* = *tail q0* **and** *q0* ∈ *Q* **and** *lt q0* = *v* **unfolding** *Q2-def* **by** *auto*
from *q-min* ‹*q* = *tail q0*› **have** *q0-tail-min*: ⋀*r*. *r* ∈ *Q2* ⟹ ¬ *dickson-less-p d m r* (*tail q0*) **by** *simp*
from ‹*q0* ∈ *Q*› **show** *?thesis*
proof
show ∀ *r*∈*Q*. ¬ *dickson-less-p d m r q0*
proof (*intro ballI notI*)
fix *r*
assume *dickson-less-p d m r q0*
hence *r* ∈ *dgrad-p-set d m* **and** *q0* ∈ *dgrad-p-set d m* **and** *r* $\prec_p$ *q0*
by (*rule dickson-less-pD1*, *rule dickson-less-pD2*, *rule dickson-less-pD3*)
from *this*(*3*) **have** *lt r* $\preceq_t$ *lt q0* **by** (*simp add*: *ord-p-lt*)
with ‹*lt q0* = *v*› **have** *lt r* $\preceq_t$ *v* **by** *simp*
assume *r* ∈ *Q*
hence ¬ *dickson-less-v d m* (*lt r*) *v* **by** (*rule v-min*)
from *False* ‹*r* ∈ *Q*› **have** *r* ≠ *0* **using** *False* **by** *blast*

```
        with ‹r ∈ dgrad-p-set d m› have d (pp-of-term (lt r)) ≤ m by (rule
dgrad-p-setD-lp)
      have ¬ lt r ≺t v
      proof
        assume lt r ≺t v
      with ‹d (pp-of-term (lt r)) ≤ m› ‹d (pp-of-term v) ≤ m› have dickson-less-v
d m (lt r) v
          by (rule dickson-less-vI)
        with ‹¬ dickson-less-v d m (lt r) v› show False ..
      qed
      with ‹lt r ⪯t v› have lt r = v by simp
      with ‹r ∈ Q› have tail r ∈ Q2 by (auto simp add: Q2-def)
      have dickson-less-p d m (tail r) (tail q0)
      proof (rule dickson-less-pI)
        show tail r ∈ dgrad-p-set d m by (rule dgrad-p-set-closed-tail, fact)
      next
        show tail q0 ∈ dgrad-p-set d m by (rule dgrad-p-set-closed-tail, fact)
      next
        have lt r = lt q0 by (simp only: ‹lt r = v› ‹lt q0 = v›)
        from ‹r ≠ 0› this ‹r ≺p q0› show tail r ≺p tail q0 by (rule ord-p-tail)
      qed
      with q0-tail-min[OF ‹tail r ∈ Q2›] show False ..
    qed
  qed
 qed
qed

theorem wf-dickson-less-p:
 assumes dickson-grading d
 shows wfP (dickson-less-p d m)
proof (rule wfI-min[to-pred])
 fix Q::('t ⇒0 'b) set and x
 assume x ∈ Q
 show ∃ z∈Q. ∀ y. dickson-less-p d m y z ⟶ y ∉ Q
 proof (cases 0 ∈ Q)
  case True
  show ?thesis
  proof (rule, rule, rule)
   from True show 0 ∈ Q .
  next
   fix q::'t ⇒0 'b
   assume dickson-less-p d m q 0
   hence q ≺p 0 by (rule dickson-less-pD3)
   thus q ∉ Q using ord-p-zero-min[of q] by simp
  qed
 next
  case False
  show ?thesis
  proof (cases Q ⊆ dgrad-p-set d m)
```

case *True*
let *?L* = *lt* ‘ *Q*
from ‹*x* ∈ *Q*› **have** *lt x* ∈ *?L* **by** *simp*
with *wf-dickson-less-v*[*OF assms*] **obtain** *v* **where** *v* ∈ *?L*
and *v-min*: ⋀*u*. *dickson-less-v d m u v* ⟹ *u* ∉ *?L* **by** (*rule wfE-min*[*to-pred*], *blast*)
from *this*(*1*) **obtain** *x1* **where** *x1* ∈ *Q* **and** *v* = *lt x1* **..**
from *this*(*1*) *True False* **have** *x1* ∈ *dgrad-p-set d m* **and** *x1* ≠ *0* **by** *auto*
hence *d* (*pp-of-term v*) ≤ *m* **unfolding** ‹*v* = *lt x1*› **by** (*rule dgrad-p-setD-lp*)
define *Q1* **where** *Q1* = {*tail p* | *p*. *p* ∈ *Q* ∧ *lt p* = *v*}
from ‹*x1* ∈ *Q*› **have** *tail x1* ∈ *Q1* **by** (*auto simp add*: *Q1-def* ‹*v* = *lt x1*›)
with *assms* **have** ∃ *p*∈*Q1*. ∀ *q*∈*Q1*. ¬ *dickson-less-p d m q p*
proof (*rule wf-dickson-less-p-aux*)
show ∀ *y*∈*Q1*. *y* ≠ *0* ⟶ *y* ∈ *dgrad-p-set d m* ∧ *dickson-less-v d m* (*lt y*) *v*
proof (*intro ballI impI*)
fix *y*
assume *y* ∈ *Q1* **and** *y* ≠ *0*
from *this*(*1*) **obtain** *y1* **where** *y1* ∈ *Q* **and** *v* = *lt y1* **and** *y* = *tail y1* **unfolding** *Q1-def*
by *blast*
from *this*(*1*) *True* **have** *y1* ∈ *dgrad-p-set d m* **..**
hence *y* ∈ *dgrad-p-set d m* **unfolding** ‹*y* = *tail y1*› **by** (*rule dgrad-p-set-closed-tail*)
thus *y* ∈ *dgrad-p-set d m* ∧ *dickson-less-v d m* (*lt y*) *v*
proof
show *dickson-less-v d m* (*lt y*) *v*
proof (*rule dickson-less-vI*)
from ‹*y* ∈ *dgrad-p-set d m*› ‹*y* ≠ *0*› **show** *d* (*pp-of-term* (*lt y*)) ≤ *m*
by (*rule dgrad-p-setD-lp*)
next
from ‹*y* ≠ *0*› **show** *lt y* $\prec_t$ *v* **unfolding** ‹*v* = *lt y1*› ‹*y* = *tail y1*› **by** (*rule lt-tail*)
qed *fact*
qed
qed
qed
then obtain *p0* **where** *p0* ∈ *Q1* **and** *p0-min*: ⋀*q*. *q* ∈ *Q1* ⟹ ¬ *dickson-less-p d m q p0* **by** *blast*
from *this*(*1*) **obtain** *p* **where** *p* ∈ *Q* **and** *v* = *lt p* **and** *p0* = *tail p* **unfolding** *Q1-def*
by *blast*
from *this*(*1*) *False* **have** *p* ≠ *0* **by** *blast*
show *?thesis*
proof (*intro bexI allI impI notI*)
fix *y*
assume *y* ∈ *Q*
hence *y* ≠ *0* **using** *False* **by** *blast*
assume *dickson-less-p d m y p*
hence *y* ∈ *dgrad-p-set d m* **and** *p* ∈ *dgrad-p-set d m* **and** *y* $\prec_p$ *p*
by (*rule dickson-less-pD1*, *rule dickson-less-pD2*, *rule dickson-less-pD3*)

from *this*(*3*) **have** $y \preceq_p p$ **by** *simp*
hence *lt y* $\preceq_t$ *lt p* **by** (*rule ord-p-lt*)
moreover have ¬ *lt y* $\prec_t$ *lt p*
proof
assume *lt y* $\prec_t$ *lt p*
have *dickson-less-v d m* (*lt y*) *v* **unfolding** ‹*v* = *lt p*›
by (*rule dickson-less-vI*, *rule dgrad-p-setD-lp*, *fact+*, *rule dgrad-p-setD-lp*, *fact+*)
hence *lt y* ∉ *?L* **by** (*rule v-min*)
hence *y* ∉ *Q* **by** *fastforce*
from *this* ‹*y* ∈ *Q*› **show** *False* **..**
qed
ultimately have *lt y* = *lt p* **by** *simp*
from ‹*y* ≠ *0*› *this* ‹$y \prec_p p$› **have** *tail y* $\prec_p$ *tail p* **by** (*rule ord-p-tail*)
from ‹*y* ∈ *Q*› **have** *tail y* ∈ *Q1* **by** (*auto simp add*: *Q1-def* ‹*v* = *lt p*› ‹*lt y* = *lt p*›[*symmetric*])
hence ¬ *dickson-less-p d m* (*tail y*) *p0* **by** (*rule p0-min*)
moreover have *dickson-less-p d m* (*tail y*) *p0* **unfolding** ‹*p0* = *tail p*›
by (*rule dickson-less-pI*, *rule dgrad-p-set-closed-tail*, *fact*, *rule dgrad-p-set-closed-tail*, *fact+*)
ultimately show *False* **..**
qed *fact*
next
case *False*
then obtain *q* **where** *q* ∈ *Q* **and** *q* ∉ *dgrad-p-set d m* **by** *blast*
from *this*(*1*) **show** *?thesis*
proof
show ∀ *y*. *dickson-less-p d m y q* ⟶ *y* ∉ *Q*
proof (*intro allI impI*)
fix *y*
assume *dickson-less-p d m y q*
hence *q* ∈ *dgrad-p-set d m* **by** (*rule dickson-less-pD2*)
with ‹*q* ∉ *dgrad-p-set d m*› **show** *y* ∉ *Q* **..**
qed
qed
qed
qed
qed

corollary *ord-p-minimum-dgrad-p-set*:
assumes *dickson-grading d* **and** *x* ∈ *Q* **and** *Q* ⊆ *dgrad-p-set d m*
obtains *q* **where** *q* ∈ *Q* **and** ⋀*y*. $y \prec_p q$ ⟹ *y* ∉ *Q*
proof −
from *assms*(*1*) **have** *wfP* (*dickson-less-p d m*) **by** (*rule wf-dickson-less-p*)
from *this assms*(*2*) **obtain** *q* **where** *q* ∈ *Q* **and** ∗: ⋀*y*. *dickson-less-p d m y q* ⟹ *y* ∉ *Q*
by (*rule wfE-min*[*to-pred*], *auto*)
from *assms*(*3*) ‹*q* ∈ *Q*› **have** *q* ∈ *dgrad-p-set d m* **..**
from ‹*q* ∈ *Q*› **show** *?thesis*

proof
fix *y*
assume $y \prec_p q$
show $y \notin Q$
proof
assume $y \in Q$
with *assms(3)* **have** $y \in$ *dgrad-p-set d m* **..**
from *this* ‹$q \in$ *dgrad-p-set d m*› ‹$y \prec_p q$› **have** *dickson-less-p d m y q*
by (*rule dickson-less-pI*)
hence $y \notin Q$ **by** (*rule* $*$)
from *this* ‹$y \in Q$› **show** *False* **..**
qed
qed
qed

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 −
from *assms(1)* **have** *wfP* (*dickson-less-v d m*) **by** (*rule wf-dickson-less-v*)
then obtain *u* **where** $u \in V$ **and** $*$: $\bigwedge w.$ *dickson-less-v d m w u* $\Longrightarrow w \notin V$
using *assms(2)*
by (*rule wfE-min*[*to-pred*]) *blast*
from *this(1)* **have** *pp-of-term u* $\in$ *pp-of-term* ‘ *V* **by** (*rule imageI*)
with *assms(3)* **have** *pp-of-term u* $\in$ *dgrad-set d m* **..**
hence *d* (*pp-of-term u*) $\leq m$ **by** (*rule dgrad-setD*)
from ‹$u \in V$› **show** *?thesis*
proof
fix *w*
assume $w \prec_t u$
show $w \notin V$
proof
assume $w \in V$
hence *pp-of-term w* $\in$ *pp-of-term* ‘ *V* **by** (*rule imageI*)
with *assms(3)* **have** *pp-of-term w* $\in$ *dgrad-set d m* **..**
hence *d* (*pp-of-term w*) $\leq m$ **by** (*rule dgrad-setD*)
from *this* ‹*d* (*pp-of-term u*) $\leq m$› ‹$w \prec_t u$› **have** *dickson-less-v d m w u*
by (*rule dickson-less-vI*)
hence $w \notin V$ **by** (*rule* $*$)
from *this* ‹$w \in V$› **show** *False* **..**
qed
qed
qed

end

10.14 More Interpretations

context *gd-powerprod*

begin

sublocale *punit*: *gd-term to-pair-unit fst* (⪯) (≺) (⪯) (≺) **..**

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* ⇒ (′*a*::*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** ‹$\preceq_t$› *50*)
and *ord-term-strict*::′*t* ⇒ ′*t* ⇒ *bool* (**infixl** ‹$\prec_t$› *50*)
begin

sublocale *gd-term* **..**

lemma *ord-p-wf*: *wfP* ($\prec_p$)
proof −
from *dickson-grading-zero* **have** *wfP* (*dickson-less-p* (λ-. *0*) *0*) **by** (*rule wf-dickson-less-p*)
thus *?thesis* **by** (*simp only*: *dickson-less-p-zero*)
qed

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*)
by (*auto simp*: *fmdom′-def fmdom′I fmap-of-list.rep-eq weak-map-of-SomeI*)
(*metis map-of-eq-None-iff option.distinct*(*1*))

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 by *standard* (*auto simp add*: *equal-poly-mapping-def poly-mapping-eqI*)
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*)
by (*auto simp*: *fmlookup-default-def fmdom′-notI split*: *option.splits*)

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*)
by (*auto simp*: *fmlookup-default-def fmdom′-notI split*: *option.splits*)

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)
by (*auto simp*: *fmlookup-default-def*)

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*
by (*auto simp*: *fmlookup-default-def*)

lemma *finite-lookup-default*:
finite {*x*. *fmlookup-default d xs x* ≠ *d*}
proof −
have {*x*. *fmlookup-default d xs x* ≠ *d*} ⊆ *fmdom′ xs*
by (*auto simp*: *fmlookup-default-def fmdom′I split*: *option.splits*)
also have *finite* ...
by *simp*
finally (*finite-subset*) **show** *?thesis* **.**
qed

lemma *lookup0-clearjunk0*: *lookup0 xs s* = *lookup0* (*clearjunk0 xs*) *s*
unfolding *clearjunk0-def fmlookup-default-def*
by *auto*

lemma *clearjunk0-nonzero*:
assumes *t* ∈ *fmdom′* (*clearjunk0 xs*)
shows *fmlookup xs t* ≠ *Some 0*
using *assms* **unfolding** *clearjunk0-def* **by** *simp*

lemma *clearjunk0-map-of-SomeD*:
  assumes *a1*: *fmlookup xs t* = *Some c* **and** *c* ≠ *0*
  shows *t* ∈ *fmdom′* (*clearjunk0 xs*)
  using *assms*
  by (*auto simp*: *clearjunk0-def fmdom′I*)

10.17 Implementation of Polynomial Mappings as Association Lists

lift-definition *Pm-fmap*::(′*a*, ′*b*::*zero*) *fmap* ⇒ ′*a* $\Rightarrow_0$ ′*b* **is** *lookup0*
  by (*rule finite-lookup-default*)

lemmas [*simp*] = *Pm-fmap.rep-eq*

code-datatype *Pm-fmap*

lemma *PM-clearjunk0-cong*:
  Pm-fmap (*clearjunk0 xs*) = *Pm-fmap xs*
  by (*metis Pm-fmap.rep-eq lookup0-clearjunk0 poly-mapping-eqI*)

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*)
  using *assms* **unfolding** *list-all-def*
  by (*force simp*: *fmlookup-default-def fmlookup-dom-iff*
      split: *option.splits if-splits*)

lemma *compute-keys-pp*[*code*]: *keys* (*Pm-fmap xs*) = *fmdom′* (*clearjunk0 xs*)
  by *transfer*
    (*auto simp*: *fmlookup-dom′-iff clearjunk0-def fmlookup-default-def fmdom′I split*: *option.splits*)

lemma *compute-zero-pp*[*code*]: *0* = *Pm-fmap fmempty*
  by (*auto intro*!: *poly-mapping-eqI simp*: *fmlookup-default-def*)

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*)))
  by (*auto intro*!: *poly-mapping-eqI*
      simp: *fmlookup-default-def lookup-add fmlookup-dom-iff PM-clearjunk0-cong*
      split: *option.splits*)

lemma *compute-lookup-pp*[*code*]:
  lookup (*Pm-fmap xs*) *x* = *lookup0 xs x*
  by (*transfer*, *simp*)

lemma *compute-minus-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*)))
by (*auto intro*!: *poly-mapping-eqI*
simp: *fmlookup-default-def lookup-minus fmlookup-dom-iff PM-clearjunk0-cong*
split: *option.splits*)

lemma *compute-uminus-pp*[*code*]:
− *Pm-fmap ys* = *Pm-fmap* (*fmmap-keys* (λ*k v*. − *lookup0 ys k*) *ys*)
by (*auto intro*!: *poly-mapping-eqI*
simp: *fmlookup-default-def*
split: *option.splits*)

lemma *compute-equal-pp*[*code*]:
equal-class.equal (*Pm-fmap xs*) (*Pm-fmap ys*) = *fmpred* (λ*k v*. *lookup0 xs k* = *lookup0 ys k*) (*xs* $++_f$ *ys*)
unfolding *equal-poly-mapping-def* **by** (*simp only*: *PM-all-2*)

lemma *compute-map-pp*[*code*]:
Poly-Mapping.map f (*Pm-fmap xs*) = *Pm-fmap* (*fmmap* (λ*x*. *f x when x* ≠ *0*) *xs*)
by (*auto intro*!: *poly-mapping-eqI*
simp: *fmlookup-default-def map.rep-eq*
split: *option.splits*)

lemma *fmran'-fmfilter-eq*: *fmran'* (*fmfilter p fm*) = {*y* | *y*. ∃ *x* ∈ *fmdom' fm*. *p x* ∧ *fmlookup fm x* = *Some y*}
by (*force simp*: *fmlookup-ran'-iff fmdom'I split*: *if-splits*)

lemma *compute-range-pp*[*code*]:
Poly-Mapping.range (*Pm-fmap xs*) = *fmran'* (*clearjunk0 xs*)
by (*force simp*: *range.rep-eq clearjunk0-def fmran'-fmfilter-eq fmdom'I*
fmlookup-default-def split: *option.splits*)

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*)]
by (*auto simp*: $sparse_0$*-def fmlookup-default-def lookup-single intro*!: *poly-mapping-eqI*)

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*))
by (*rule poly-mapping-eqI*)
(*auto simp add*: *fmlookup-default-fmmap-keys fmlookup-dom-iff fmdom'-notI lcs-poly-mapping.rep-eq fmdom'-notD*)

lemma *compute-deg-pp*[*code*]:
deg-pm (*Pm-fmap xs*) = *sum* (*the o fmlookup xs*) (*fmdom' xs*)
proof −
have *deg-pm* (*Pm-fmap xs*) = *sum* (*lookup* (*Pm-fmap xs*)) (*keys* (*Pm-fmap xs*))
by (*rule deg-pm-superset*) *auto*
also have . . . = *sum* (*the o fmlookup xs*) (*fmdom' xs*)
by (*rule sum.mono-neutral-cong-left*)
(*auto simp*: *fmlookup-dom'-iff fmdom'I in-keys-iff fmlookup-default-def split*: *option.splits*)
finally show *?thesis* **.**
qed

definition *adds-pp-add-linorder* :: ($'b \Rightarrow_0 'a$::*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*
unfolding *adds-pp-add-linorder-def*
unfolding *adds-poly-mapping*
using *fmdom-notI*
by (*force simp*: *fmlookup-dom-iff le-fun-def split*: *option.splits if-splits*)

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 −
have *lex-pm s t* = (¬ *lex-pm-strict t s*) **by** (*simp add*: *lex-pm-strict-alt*)
also have ... = (∀ *x*. *lookup s x* ≤ *lookup t x* ∨ (∃ *y*<*x*. *lookup s y* ≠ *lookup t y*))
by (*simp add*: *lex-pm-strict-def less-poly-mapping-def less-fun-def*) (*metis leD leI*)
finally show *?thesis* **.**
qed

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*
)
unfolding *Let-def lex-pm-iff fmpred-iff Pm-fmap.rep-eq fmlookup-add fmlookup-dom-iff*
apply (*intro iffI*)
apply (*metis fmdom′-notD fmlookup-default-if*(*2*) *fmlookup-dom′-iff leD*)
apply (*metis eq-iff not-le fmdom′-notD fmlookup-default-if*(*2*) *fmlookup-dom′-iff*)
done

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*))
)
by (*auto simp*: *Let-def deg-pm.rep-eq dord-fun-def dord-pm.rep-eq*)
(*simp-all add*: *Pm-fmap.abs-eq*)

11.1.1 Computations

experiment begin

abbreviation *X* ≡ *0*::*nat*
abbreviation *Y* ≡ *1*::*nat*
abbreviation *Z* ≡ *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*]
by *eval*+

lemma
$sparse_0$ [(*X*, *2*::*nat*), (*Z*, *7*)] − $sparse_0$ [(*X*, *2*), (*Z*, *2*)] = $sparse_0$ [(*Z*, *5*)]

by *eval*

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*)]
by *eval*

lemma
(*sparse*$_0$ [(*X*, *2*::*nat*), (*Z*, *1*)]) *adds* (*sparse*$_0$ [(*X*, *3*), (*Y*, *2*), (*Z*, *1*)])
by *eval*

lemma
lookup (*sparse*$_0$ [(*X*, *2*::*nat*), (*Z*, *3*)]) *X* = *2*
by *eval*

lemma
deg-pm (*sparse*$_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *3*), (*X*, *1*)]) = *6*
by *eval*

lemma
lex-pm (*sparse*$_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *3*)]) (*sparse*$_0$ [(*X*, *4*)])
by *eval*

lemma
lex-pm (*sparse*$_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *3*)]) (*sparse*$_0$ [(*X*, *4*)])
by *eval*

lemma
¬ (*dlex-pm* (*sparse*$_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *3*)]) (*sparse*$_0$ [(*X*, *4*)]))
by *eval*

lemma
dlex-pm (*sparse*$_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *2*)]) (*sparse*$_0$ [(*X*, *5*)])
by *eval*

lemma
¬ (*drlex-pm* (*sparse*$_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *2*)]) (*sparse*$_0$ [(*X*, *5*)]))
by *eval*

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*)])
by (*auto intro*!: *poly-mapping-eqI simp*: *sparse*$_0$*-def fmlookup-default-def lookup-single*)

lemma *compute-one-poly-mapping* [*code*]: *1* = *sparse*$_0$ [(*0*, *1*)]
  by (*metis compute-monomial single-one zero-neq-one*)

lemma *compute-except-poly-mapping* [*code*]:
  except (*Pm-fmap xs*) *S* = *Pm-fmap* (*fmfilter* (λ*k*. *k* ∉ *S*) *xs*)
  by (*auto simp*: *fmlookup-default-def lookup-except split*: *option.splits intro*!: *poly-mapping-eqI*)

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*
  by (*auto simp*: *fmlookup-default-def fmlookup-of-list split*: *if-splits option.splits*)

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*)
  by *simp* (*metis UnI1 UnI2 in-keys-iff poly-mapping-eqI*)

lemma *keys-add-eq*: *keys* (*a* + *b*) = *keys a* ∪ *keys b* − {*x* ∈ *keys a* ∩ *keys b*. *lookup a x* + *lookup b x* = *0*}
  by (*auto simp*: *in-keys-iff lookup-add add-eq-0-iff*)

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 −
  fix *t* **and** *f*::′*t* ⇒ ′*b option*
  assume *finite* (*dom f*)
  have *dom* (λ*x*. *if t adds*$_p$ *x then f* (*x* ⊖ *t*) *else None*) ⊆ (⊕) *t* ‘ *dom f*
    by (*auto simp*: *adds-pp-alt domI term-simps split*: *if-splits*)
  also have *finite* . . .
    using ‹*finite* (*dom f*)› **by** *simp*
  finally (*finite-subset*) **show** *finite* (*dom* (λ*x*. *if t adds*$_p$ *x then f* (*x* ⊖ *t*) *else None*)) **.**
qed

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*)
  unfolding *shift-map-keys-def*
  apply *transfer*
  subgoal for *f t xs*
  proof −
    show *?thesis*

apply (*rule ext*)
subgoal for *x*
apply (*cases t* $adds_p$ *x*)
subgoal by (*induction xs*) (*auto simp*: *adds-pp-alt term-simps*)
subgoal by (*induction xs*) (*auto simp*: *adds-pp-alt term-simps*)
done
done
qed
done

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 (*cases c* = *0*)
case *True*
hence *monom-mult c t* (*Pm-fmap xs*) = *0* **using** *monom-mult-zero-left* **by** *simp*
thus *?thesis* **using** *True*
by *simp*
next
case *False*
thus *?thesis*
by (*auto simp*: *simp*: *fmlookup-default-def shift-map-keys-def lookup-monom-mult adds-def group-eq-aux shift-keys.rep-eq*
intro!: *poly-mapping-eqI split*: *option.splits*)
qed

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 (*split list.splits*, *simp*, *intro conjI impI allI*, *goal-cases*)
case (*1 t c ys*)
have *Pm-fmap* (*fmupd t c* (*fmap-of-list ys*)) = $sparse_0$ [(*t*, *c*)] + *except* ($sparse_0$ *ys*) {*t*}
by (*auto simp*: $sparse_0$*-def fmlookup-default-def lookup-add lookup-except split*: *option.splits intro*!: *poly-mapping-eqI*)
also have $sparse_0$ [(*t*, *c*)] = *monomial c t*
by (*auto simp*: $sparse_0$*-def lookup-single fmlookup-default-def intro*!: *poly-mapping-eqI*)
finally show *?case*
by (*simp add*: *algebra-simps mult-scalar-monomial* $sparse_0$*-def*)
qed

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*
by (*simp add*: *monomial1-def*)

lemma *monomial-mp*: *monomial c* (*pp*::$'a \Rightarrow_0 nat$) = $Const_0$ *c* ∗ *monomial1 pp*
for *c*::$'b$::*comm-semiring-1*
by (*auto intro*!: *poly-mapping-eqI simp*: *monomial1-def* $Const_0$*-def mult-single*)

lemma *monomial1-add*: (*monomial1* (*a* + *b*)::($'a$::*monoid-add*$\Rightarrow_0$ $'b$::*comm-semiring-1*))
= *monomial1 a* ∗ *monomial1 b*
by (*auto simp*: *monomial1-def mult-single*)

lemma *monomial1-monomial*: *monomial1* (*monomial n v*) = (Var_0 *v*::-$\Rightarrow_0$($'b$::*comm-semiring-1*))^*n*
by (*auto intro*!: *poly-mapping-eqI simp*: *monomial1-def* Var_0*-power lookup-single*
when-def)

lemma *Ball-True*: (∀ *x*∈*X*. *True*) ⟷ *True* **by** *auto*
lemma *Collect-False*: {*x*. *False*} = {} **by** *simp*

lemma *Pm-fmap-sum*: *Pm-fmap f* = ($\sum$ *x* ∈ *fmdom' f*. *monomial* (*lookup0 f x*)
x)
including *fmap.lifting*
by (*auto intro*!: *poly-mapping-eqI sum.neutral*
simp: *fmlookup-default-def lookup-sum lookup-single when-def fmdom'I*
split: *option.splits*)

lemma *MPoly-numeral*: *MPoly* (*numeral x*) = *numeral x*
by (*metis monom.abs-eq monom-numeral single-numeral*)

lemma *MPoly-power*: *MPoly* (*x* ^ *n*) = *MPoly x* ^ *n*
by (*induction n*) (*auto simp*: *one-mpoly-def times-mpoly.abs-eq*[*symmetric*])

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*]:

simproc-setup passive *mpoly* (*Pm-fmap mpp*::(- $\Rightarrow_0$ *nat*) $\Rightarrow_0$ -) =
‹*K* (*fn ctxt* => *fn ct* =>
SOME (*Simplifier.rewrite* (*put-simpset HOL-basic-ss ctxt addsimps*
(*Named-Theorems.get ctxt* (***named-theorems*** ‹*mpoly-simps*›))) *ct*))›

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 (*induct xs arbitrary*: *a b*)
fix *a b*
show *foldl f* (*f a b*) [] = *f a* (*foldl f b* []) **by** *simp*
next
fix *a b x xs*
assume $\bigwedge a\ b.$ *foldl f* (*f a b*) *xs* = *f a* (*foldl f b xs*)
from *assms*[*of a b x*] *this*[*of a f b x*]
show *foldl f* (*f a b*) (*x* # *xs*) = *f a* (*foldl f b* (*x* # *xs*)) **unfolding** *foldl-Cons*
by *simp*
qed

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*)
unfolding *list-max-def foldl-Cons*
proof −
have *foldl ord-term-lin.max* (*ord-term-lin.max x min-term*) *xs* =
ord-term-lin.max x (*foldl ord-term-lin.max min-term xs*)
by (*rule foldl-assoc*, *rule ord-term-lin.max.assoc*)
from *this ord-term-lin.max.commute*[*of min-term x*]

show *foldl ord-term-lin.max* (*ord-term-lin.max min-term x*) *xs* =
ord-term-lin.max x (*foldl ord-term-lin.max min-term xs*) **by** *simp*
qed

lemma *list-max-empty*: *list-max* [] = *min-term*
unfolding *list-max-def* **by** *simp*

lemma *list-max-in-list*:
assumes *xs* ≠ []
shows *list-max xs* ∈ *set xs*
using *assms*
proof (*induct xs*, *simp*)
fix *x xs*
assume *IH*: *xs* ≠ [] ⟹ *list-max xs* ∈ *set xs*
show *list-max* (*x* # *xs*) ∈ *set* (*x* # *xs*)
proof (*cases xs* = [])
case *True*
hence *list-max* (*x* # *xs*) = *ord-term-lin.max min-term x* **unfolding** *list-max-def*
by *simp*
also have ... = *x* **unfolding** *ord-term-lin.max-def* **by** (*simp add*: *min-term-min*)
finally show *?thesis* **by** *simp*
next
assume *xs* ≠ []
show *?thesis*
proof (*cases x* $\preceq_t$ *list-max xs*)
case *True*
hence *list-max* (*x* # *xs*) = *list-max xs*
unfolding *list-max-Cons ord-term-lin.max-def* **by** *simp*
thus *?thesis* **using** *IH*[*OF* ‹*xs* ≠ []›] **by** *simp*
next
case *False*
hence *list-max* (*x* # *xs*) = *x* **unfolding** *list-max-Cons ord-term-lin.max-def*
by *simp*
thus *?thesis* **by** *simp*
qed
qed
qed

lemma *list-max-maximum*:
assumes *a* ∈ *set xs*
shows *a* $\preceq_t$ (*list-max xs*)
using *assms*
proof (*induct xs*)
assume *a* ∈ *set* []
thus *a* $\preceq_t$ *list-max* [] **by** *simp*
next
fix *x xs*
assume *IH*: *a* ∈ *set xs* ⟹ *a* $\preceq_t$ *list-max xs* **and** *a-in*: *a* ∈ *set* (*x* # *xs*)
from *a-in* **have** *a* = *x* ∨ *a* ∈ *set xs* **by** *simp*

thus *a* $\preceq_t$ *list-max* (*x* # *xs*) **unfolding** *list-max-Cons*
proof
assume *a* = *x*
thus *a* $\preceq_t$ *ord-term-lin.max x* (*list-max xs*) **by** *simp*
next
assume *a* ∈ *set xs*
from *IH*[*OF this*] **show** *a* $\preceq_t$ *ord-term-lin.max x* (*list-max xs*)
by (*simp add*: *ord-term-lin.le-max-iff-disj*)
qed
qed

lemma *list-max-nonempty*:
assumes *xs* ≠ []
shows *list-max xs* = *ord-term-lin.Max* (*set xs*)
proof −
have *fin*: *finite* (*set xs*) **by** *simp*
have *ord-term-lin.Max* (*set xs*) = *list-max xs*
proof (*rule ord-term-lin.Max-eqI*[*OF fin*, *of list-max xs*])
fix *y*
assume *y* ∈ *set xs*
from *list-max-maximum*[*OF this*] **show** *y* $\preceq_t$ *list-max xs* **.**
next
from *list-max-in-list*[*OF assms*] **show** *list-max xs* ∈ *set xs* **.**
qed
thus *?thesis* **by** *simp*
qed

lemma *in-set-clearjunk-iff-map-of-eq-Some*:
(*a*, *b*) ∈ *set* (*AList.clearjunk xs*) ⟷ *map-of xs a* = *Some b*
by (*metis Some-eq-map-of-iff distinct-clearjunk map-of-clearjunk*)

lemma *Pm-fmap-of-list-eq-zero-iff*:
Pm-fmap (*fmap-of-list xs*) = *0* ⟷ [(*k*, *v*)←*AList.clearjunk xs* . *v* ≠ *0*] = []
by (*auto simp*: *poly-mapping-eq-iff fmlookup-default-def fun-eq-iff in-set-clearjunk-iff-map-of-eq-Some filter-empty-conv fmlookup-of-list split*: *option.splits*)

lemma *fmdom′-clearjunk0*: *fmdom′* (*clearjunk0 xs*) = *fmdom′ xs* − {*x*. *fmlookup xs x* = *Some 0*}
by (*metis* (*no-types*, *lifting*) *clearjunk0-def fmdom′-drop-set fmfilter-alt-defs*(*2*) *fmfilter-cong′ mem-Collect-eq*)

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 −
have *keys* (*Pm-fmap* (*fmap-of-list xs*)) = *fst* ‘ {*x* ∈ *set* (*AList.clearjunk xs*). *case x of* (*k*, *v*) ⇒ *v* ≠ *0*}
by (*auto simp*: *compute-keys-pp fmdom′-clearjunk0 fmap-of-list.rep-eq*

in-set-clearjunk-iff-map-of-eq-Some fmdom'I image-iff fmlookup-dom'-iff)
then show *?thesis*
unfolding *lt-def*
by (*auto simp*: *Pm-fmap-of-list-eq-zero-iff list-max-empty list-max-nonempty*)
qed

lemma *compute-higher-poly-mapping* [*code*]:
higher (*Pm-fmap xs*) *t* = *Pm-fmap* (*fmfilter* (λ*k*. $t \prec_t k$) *xs*)
unfolding *higher-def compute-except-poly-mapping*
by (*metis mem-Collect-eq ord-term-lin.leD ord-term-lin.leI*)

lemma *compute-lower-poly-mapping* [*code*]:
lower (*Pm-fmap xs*) *t* = *Pm-fmap* (*fmfilter* (λ*k*. $k \prec_t t$) *xs*)
unfolding *lower-def compute-except-poly-mapping*
by (*metis mem-Collect-eq ord-term-lin.leD ord-term-lin.leI*)

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* $\Rightarrow_0$ *nat*)$\Rightarrow_0$*'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*)
by (*simp add*: *punit.compute-shift-map-keys shift-map-keys-punit-def*)

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*
apply (*fact MPoly-Type-Class.punit.term-powerprod-axioms*)
apply (*fact MPoly-Type-Class.punit-adds-term*)
apply (*fact MPoly-Type-Class.punit-pp-of-term*)
apply (*fact MPoly-Type-Class.punit-component-of-term*)
done

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*)
by (*simp add*: *monom-mult-punit-def punit.compute-monom-mult-poly-mapping*

shift-map-keys-punit-def)

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)
  by (*simp only*: *punit-mult-scalar*[*symmetric*] *punit.compute-mult-scalar-poly-mapping* *monom-mult-punit-def*)

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* **.**

lemma
  keys (X^2 ∗ *Z* ^ *3* + *2* ∗ *Y* ^ *3* ∗ Z^2) =
    {*monomial 2 0* + *monomial 3 2*, *monomial 3 1* + *monomial 2 2*}
  by *eval*

lemma
  keys (X^2 ∗ *Z* ^ *3* + *2* ∗ *Y* ^ *3* ∗ Z^2) =
    {*monomial 2 0* + *monomial 3 2*, *monomial 3 1* + *monomial 2 2*}
  by *eval*

lemma
  − *1* ∗ X^2 ∗ *Z* ^ *7* + − *2* ∗ *Y* ^ *3* ∗ Z^2 = − X^2 ∗ *Z* ^ *7* + − *2* ∗ *Y* ^ *3* ∗ Z^2
  by *eval*

lemma
  X^2 ∗ *Z* ^ *7* + *2* ∗ *Y* ^ *3* ∗ Z^2 + X^2 ∗ *Z* ^ *4* + − *2* ∗ *Y* ^ *3* ∗ Z^2 = X^2 ∗ *Z* ^ *7* + X^2 ∗ *Z* ^ *4*
  by *eval*

lemma

$X^2 * Z \,\hat{}\, 7 + 2 * Y \,\hat{}\, 3 * Z^2 - X^2 * Z \,\hat{}\, 4 + - 2 * Y \,\hat{}\, 3 * Z^2 =$
$X^2 * Z \,\hat{}\, 7 - X^2 * Z \,\hat{}\, 4$
by *eval*

lemma
lookup $(X^2 * Z \,\hat{}\, 7 + 2 * Y \,\hat{}\, 3 * Z^2 + 2)$ $(sparse_0\ [(0, 2), (2, 7)]) = 1$
by *eval*

lemma
$X^2 * Z \,\hat{}\, 7 + 2 * Y \,\hat{}\, 3 * Z^2 \neq$
$X^2 * Z \,\hat{}\, 4 + - 2 * Y \,\hat{}\, 3 * Z^2$
by *eval*

lemma
$0 * X\hat{}2 * Z\hat{}7 + 0 * Y\hat{}3{*}Z^2 = 0$
by *eval*

lemma
monom-mult-punit 3 $(sparse_0\ [(1, 2{::}nat)])$ $(X^2 * Z + 2 * Y \,\hat{}\, 3 * Z^2) =$
$3 * Y^2 * Z * X^2 + 6 * Y \,\hat{}\, 5 * Z^2$
by *eval*

lemma
monomial (-4) $(sparse_0\ [(0, 2{::}nat)]) = - 4 * X^2$
by *eval*

lemma *monomial* $(0{::}rat)$ $(sparse_0\ [(0{::}nat, 2{::}nat)]) = 0$
by *eval*

lemma
$(X^2 * Z + 2 * Y \,\hat{}\, 3 * Z^2) * (X^2 * Z \,\hat{}\, 3 + - 2 * Y \,\hat{}\, 3 * Z^2) =$
$X \,\hat{}\, 4 * Z \,\hat{}\, 4 + - 2 * X^2 * Z \,\hat{}\, 3 * Y \,\hat{}\, 3 +$
$- 4 * Y \,\hat{}\, 6 * Z \,\hat{}\, 4 + 2 * Y \,\hat{}\, 3 * Z \,\hat{}\, 5 * X^2$
by *eval*

end

11.3.2 Vector-Polynomials

type-synonym $'a$ *vmpoly-tc* $= ((nat \Rightarrow_0 nat) \times nat) \Rightarrow_0 {'a}$

definition *shift-map-keys-pprod* = *pprod.shift-map-keys*

global-interpretation *pprod*: *term-powerprod* $\lambda x.\ x\ \lambda x.\ x$
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*
apply (*fact MPoly-Type-Class.pprod.term-powerprod-axioms*)
apply (*fact MPoly-Type-Class.pprod-pp-of-term*)
apply (*fact MPoly-Type-Class.pprod-component-of-term*)
done

lemma *compute-adds-term-pprod* [*code-unfold*]:
adds-term-pprod u v = (*snd u* = *snd v* ∧ *adds-pp-add-linorder* (*fst u*) (*fst v*))
by (*simp add*: *adds-term-pprod-def pprod.adds-term-def adds-pp-add-linorder-def*)

lemma *compute-splus-pprod* [*code*]: *splus-pprod t* (*s*, *i*) = (*t* + *s*, *i*)
by (*simp add*: *splus-pprod-def pprod.splus-def*)

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*)
by (*simp add*: *pprod.compute-shift-map-keys shift-map-keys-pprod-def splus-pprod-def*)

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*)
by (*simp add*: *monom-mult-pprod-def pprod.compute-monom-mult-poly-mapping shift-map-keys-pprod-def*)

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)
by (*simp only*: *mult-scalar-pprod-def pprod.compute-mult-scalar-poly-mapping monom-mult-pprod-def*)

definition Vec_0 :: *nat* ⇒ ((*'a* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'b*) ⇒ ((*'a* $\Rightarrow_0$ *nat*) × *nat*) $\Rightarrow_0$ *'b*::*semiring-1* **where**
Vec_0 *i p* = *mult-scalar-pprod p* (*Poly-Mapping.single* (*0*, *i*) *1*)

experiment begin interpretation $trivariate_0$*-rat* **.**

lemma
keys (Vec_0 *0* (X^2 ∗ *Z* ^ *3*) + Vec_0 *1* (*2* ∗ *Y* ^ *3* ∗ Z^2)) =
{($sparse_0$ [(*0*, *2*), (*2*, *3*)], *0*), ($sparse_0$ [(*1*, *3*), (*2*, *2*)], *1*)}
by *eval*

lemma
keys (Vec_0 *0* (X^2 ∗ *Z* ^ *3*) + Vec_0 *2* (*2* ∗ *Y* ^ *3* ∗ Z^2)) =
{($sparse_0$ [(*0*, *2*), (*2*, *3*)], *0*), ($sparse_0$ [(*1*, *3*), (*2*, *2*)], *2*)}
by *eval*

lemma

Vec_0 *1* $(X^2 * Z$ ^ *7* + *2* $* Y$ ^ *3* $* Z^2)$ + Vec_0 *3* $(X^2 * Z$ ^ *4*) + Vec_0 *1* (− *2* $* Y$ ^ *3* $* Z^2)$ =
Vec_0 *1* $(X^2 * Z$ ^ *7*) + Vec_0 *3* $(X^2 * Z$ ^ *4*)
by *eval*

lemma
lookup (Vec_0 *0* $(X^2 * Z$ ^ *7*) + Vec_0 *1* (*2* $* Y$ ^ *3* $* Z^2$ + *2*)) ($sparse_0$ [(*0*, *2*), (*2*, *7*)], *0*) = *1*
by *eval*

lemma
lookup (Vec_0 *0* $(X^2 * Z$ ^ *7*) + Vec_0 *1* (*2* $* Y$ ^ *3* $* Z^2$ + *2*)) ($sparse_0$ [(*0*, *2*), (*2*, *7*)], *1*) = *0*
by *eval*

lemma
Vec_0 *0* (*0* $* X$^*2* $* Z$^*7*) + Vec_0 *1* (*0* $* Y^{*3*}*Z^2$) = *0*
by *eval*

lemma
monom-mult-pprod 3 ($sparse_0$ [(*1*, *2*::*nat*)]) (Vec_0 *0* $(X^2 * Z)$ + Vec_0 *1* (*2* $* Y$ ^ *3* $* Z^2$)) =
Vec_0 *0* (*3* $* Y^2 * Z * X^2$) + Vec_0 *1* (*6* $* Y$ ^ *5* $* Z^2$)
by *eval*

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* **.**

instance proof *standard* **qed** (*transfer*, *rule equal-eq*)

end

experiment begin interpretation *trivariate* **.**

lemmas [*mpoly-simps*] = *plus-mpoly.abs-eq*

lemma *content-primitive* ($4 * X * Y\hat{}2 * Z\hat{}3 + 6 * X^2 * Y\hat{}4 + 8 * X^2 * Y\hat{}5$)
=
(2::*int*, $2 * X * Y^2 * Z \hat{} 3 + 3 * X^2 * Y \hat{} 4 + 4 * X^2 * Y \hat{} 5$)
by *eval*

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*::($'a \Rightarrow_0 'b$) *set*
morphisms *mapping-of PP* **..**

setup-lifting *type-definition-pp*

lift-definition *pp-of-fun* :: ($'a \Rightarrow 'b$) $\Rightarrow$ ($'a$, $'b$::*zero*) *pp*
is *Abs-poly-mapping* **.**

11.5 *lookup-pp*, *keys-pp* and *single-pp*

lift-definition *lookup-pp* :: ($'a$, $'b$::*zero*) *pp* $\Rightarrow 'a \Rightarrow 'b$ **is** *lookup* **.**

lift-definition *keys-pp* :: ($'a$, $'b$::*zero*) *pp* $\Rightarrow 'a$ *set* **is** *keys* **.**

lift-definition *single-pp* :: $'a \Rightarrow 'b \Rightarrow$ ($'a$, $'b$::*zero*) *pp* **is** *Poly-Mapping.single* **.**

lemma *lookup-pp-of-fun*: *finite* $\{x.\ f\ x \neq 0\} \Longrightarrow$ *lookup-pp* (*pp-of-fun* f) = f
by (*transfer*, *rule Abs-poly-mapping-inverse*, *simp*)

lemma *pp-of-lookup*: *pp-of-fun* (*lookup-pp* t) = t
by (*transfer*, *fact lookup-inverse*)

lemma *pp-eqI*: ($\bigwedge u$. *lookup-pp* $s\ u$ = *lookup-pp* $t\ u$) $\Longrightarrow s = t$
by (*transfer*, *rule poly-mapping-eqI*)

lemma *pp-eq-iff*: ($s = t$) $\longleftrightarrow$ (*lookup-pp* s = *lookup-pp* t)
by (*auto intro*: *pp-eqI*)

lemma *keys-pp-iff*: $x \in$ *keys-pp* $t \longleftrightarrow$ (*lookup-pp* $t\ x \neq 0$)
by (*simp add*: *in-keys-iff keys-pp.rep-eq lookup-pp.rep-eq*)

lemma *pp-eqI'*:
  assumes $\bigwedge u.\ u \in$ *keys-pp* $s \cup$ *keys-pp* $t \Longrightarrow$ *lookup-pp* $s\ u =$ *lookup-pp* $t\ u$
  shows $s = t$
proof (*rule pp-eqI*)
  fix *u*
  show *lookup-pp* $s\ u =$ *lookup-pp* $t\ u$
  proof (*cases* $u \in$ *keys-pp* $s \cup$ *keys-pp* t)
    case *True*
    thus *?thesis* **by** (*rule assms*)
  next
    case *False*
    thus *?thesis* **by** (*simp add*: *keys-pp-iff*)
  qed
qed

lemma *lookup-single-pp*: *lookup-pp* (*single-pp* $x\ e$) $y = (e$ *when* $x = y)$
  by (*transfer*, *simp only*: *lookup-single*)

11.6 Additive Structure

instantiation *pp* :: (*type*, *zero*) *zero*
begin

lift-definition *zero-pp* :: ('*a*, '*b*) *pp* **is** $0::'a \Rightarrow_0 'b$ **.**

lemma *lookup-zero-pp* [*simp*]: *lookup-pp* $0 = 0$
  by (*transfer*, *simp add*: *lookup-zero-fun*)

instance ..

end

lemma *single-pp-zero* [*simp*]: *single-pp* $x\ 0 = 0$
  by (*rule pp-eqI*, *simp add*: *lookup-single-pp*)

instantiation *pp* :: (*type*, *monoid-add*) *monoid-add*
begin

lift-definition *plus-pp* :: ('*a*, '*b*) *pp* $\Rightarrow$ ('*a*, '*b*) *pp* $\Rightarrow$ ('*a*, '*b*) *pp* **is** $(+)::('a \Rightarrow_0 'b) \Rightarrow$ - **.**

lemma *lookup-plus-pp*: *lookup-pp* $(s + t) =$ *lookup-pp* $s +$ *lookup-pp* t
  by (*transfer*, *simp add*: *lookup-plus-fun*)

instance by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff add.assoc*)+

end

lemma *single-pp-plus*: *single-pp* $x\ a +$ *single-pp* $x\ b =$ *single-pp* $x\ (a + b)$

by (*rule pp-eqI*, *simp add*: *lookup-single-pp lookup-plus-pp when-def*)

instance *pp* :: (*type*, *comm-monoid-add*) *comm-monoid-add*
by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff ac-simps*)+

instantiation *pp* :: (*type*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add*
begin

lift-definition *minus-pp* :: ($'a$, $'b$) *pp* $\Rightarrow$ ($'a$, $'b$) *pp* $\Rightarrow$ ($'a$, $'b$) *pp* **is** $(-)$::($'a \Rightarrow_0$ $'b$) $\Rightarrow$ - **.**

lemma *lookup-minus-pp*: *lookup-pp* $(s - t)$ = *lookup-pp* s $-$ *lookup-pp* t
by (*transfer*, *simp only*: *lookup-minus-fun*)

instance by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff diff-diff-add*)+

end

11.7 $'a \Rightarrow_0 'b$ belongs to class *comm-powerprod*

instance *poly-mapping* :: (*type*, *cancel-comm-monoid-add*) *comm-powerprod*
by *standard*

11.8 $'a \Rightarrow_0 'b$ belongs to class *ninv-comm-monoid-add*

instance *poly-mapping* :: (*type*, *ninv-comm-monoid-add*) *ninv-comm-monoid-add*
proof (*standard*, *transfer*)
fix s t::$'a \Rightarrow 'b$
assume $(\lambda k.\ s\ k + t\ k) = (\lambda\text{-}.\ 0)$
hence $s + t = 0$ **by** (*simp only*: *plus-fun-def zero-fun-def*)
hence $s = 0$ **by** (*rule plus-eq-zero*)
thus $s = (\lambda\text{-}.\ 0)$ **by** (*simp only*: *zero-fun-def*)
qed

11.9 ($'a$, $'b$) *pp* belongs to class *lcs-powerprod*

lemma *adds-pp-iff*: (s *adds* t) $\longleftrightarrow$ (*mapping-of* s *adds* *mapping-of* t)
unfolding *adds-def* **by** (*transfer*, *fact refl*)

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

lift-definition *lcs-pp* :: ($'a$, $'b$) *pp* $\Rightarrow$ ($'a$, $'b$) *pp* $\Rightarrow$ ($'a$, $'b$) *pp* **is** *lcs-powerprod-class.lcs*
.

lemma *lookup-lcs-pp*: *lookup-pp* (*lcs* s t) x = *max* (*lookup-pp* s x) (*lookup-pp* t x)
by (*transfer*, *simp add*: *lookup-lcs-fun lcs-fun-def*)

instance
apply (*intro-classes*, *simp-all only*: *adds-pp-iff*)

subgoal by (*transfer*, *rule adds-lcs*)
subgoal by (*transfer*, *elim lcs-adds*)
subgoal by (*transfer*, *rule lcs-comm*)
done

end

11.10 (*'a*, *'b*) *pp* belongs to class *ulcs-powerprod*

instance *pp* :: (*type*, *add-linorder-min*) *ulcs-powerprod* **by** *intro-classes* (*transfer*, *elim plus-eq-zero*)

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
assume *?l*
with - **show** *?r*
proof (*rule almost-full-on-hom*)
fix *x y* :: (*'a*, *'b*) *pp*
assume *x adds y*
thus *mapping-of x adds mapping-of y* **by** (*simp only*: *adds-pp-iff*)
qed
next
assume *?r*
hence *almost-full-on* (λ*x y*. *mapping-of x adds mapping-of y*) *A*
using *subset-refl* **by** (*rule almost-full-on-map*)
thus *?l* **by** (*simp only*: *adds-pp-iff*[*symmetric*])
qed

lift-definition *varnum-pp* :: (*'a*::*countable*, *'b*::*zero*) *pp* ⇒ *nat* **is** *varnum* {} **.**

lemma *dickson-grading-varnum-pp*:
dickson-grading (*varnum-pp*::(*'a*::*countable*, *'b*::*add-wellorder*) *pp* ⇒ *nat*)
proof (*rule dickson-gradingI*)
fix *s t* :: (*'a*, *'b*) *pp*
show *varnum-pp* (*s* + *t*) = *max* (*varnum-pp s*) (*varnum-pp t*) **by** (*transfer*, *rule varnum-plus*)
next
fix *m*::*nat*
show *almost-full-on* (*adds*) {*x*::(*'a*, *'b*) *pp*. *varnum-pp x* ≤ *m*} **unfolding** *almost-full-on-pp-iff*
proof (*transfer*, *simp*)
fix *m*::*nat*
from *dickson-grading-varnum-empty* **show** *almost-full-on* (*adds*) {*x*::*'a* $\Rightarrow_0$ *'b*. *varnum* {} *x* ≤ *m*}

by (*rule dickson-gradingD2*)
qed
qed

instance *pp* :: (*countable*, *add-wellorder*) *graded-dickson-powerprod*
by (*standard*, *rule*, *fact dickson-grading-varnum-pp*)

instance *pp* :: (*finite*, *add-wellorder*) *dickson-powerprod*
proof
have *eq*: *range mapping-of* = *UNIV* **by** (*rule surjI*, *rule PP-inverse*, *rule UNIV-I*)
show *almost-full-on* (*adds*) (*UNIV*::('*a*, '*b*) *pp set*) **by** (*simp add*: *almost-full-on-pp-iff eq dickson*)
qed

11.12 Lexicographic Term Order

lift-definition *lex-pp* :: ('*a*, '*b*) *pp* ⇒ ('*a*::*linorder*, '*b*::{*zero*,*linorder*}) *pp* ⇒ *bool* **is** *lex-pm* **.**

lift-definition *lex-pp-strict* :: ('*a*, '*b*) *pp* ⇒ ('*a*::*linorder*, '*b*::{*zero*,*linorder*}) *pp* ⇒ *bool* **is** *lex-pm-strict* **.**

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*)))
by (*transfer*, *fact lex-pm-alt*)

lemma *lex-pp-refl*: *lex-pp s s*
by (*transfer*, *fact lex-pm-refl*)

lemma *lex-pp-antisym*: *lex-pp s t* ⟹ *lex-pp t s* ⟹ *s* = *t*
by (*transfer*, *intro lex-pm-antisym*)

lemma *lex-pp-trans*: *lex-pp s t* ⟹ *lex-pp t u* ⟹ *lex-pp s u*
by (*transfer*, *rule lex-pm-trans*)

lemma *lex-pp-lin*: *lex-pp s t* ∨ *lex-pp t s*
by (*transfer*, *fact lex-pm-lin*)

lemma *lex-pp-lin'*: ¬ *lex-pp t s* ⟹ *lex-pp s t*
using *lex-pp-lin* **by** *blast* — Better suited for *auto*.

corollary *lex-pp-strict-alt* [*code*]:
lex-pp-strict s t = (¬ *lex-pp t s*) **for** *s t*::(-, -::*ordered-comm-monoid-add*) *pp*
by (*transfer*, *fact lex-pm-strict-alt*)

lemma *lex-pp-zero-min*: *lex-pp 0 s* **for** *s*::(-, -::*add-linorder-min*) *pp*
by (*transfer*, *fact lex-pm-zero-min*)

lemma *lex-pp-plus-monotone*: *lex-pp s t* ⟹ *lex-pp* (*s* + *u*) (*t* + *u*)

for *s t*::(-, -::{*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*}) *pp*
by (*transfer*, *intro lex-pm-plus-monotone*)

lemma *lex-pp-plus-monotone'*: *lex-pp s t* ⟹ *lex-pp* (*u* + *s*) (*u* + *t*)
for *s t*::(-, -::{*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*}) *pp*
unfolding *add.commute*[*of u*] **by** (*rule lex-pp-plus-monotone*)

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 by *intro-classes* (*auto simp*: *less-eq-pp-def less-pp-def lex-pp-refl lex-pp-strict-alt intro*: *lex-pp-antisym lex-pp-lin' elim*: *lex-pp-trans*)

end

11.13 Degree

lift-definition *deg-pp* :: (*'a*, *'b*::*comm-monoid-add*) *pp* ⇒ *'b* **is** *deg-pm* **.**

lemma *deg-pp-alt*: *deg-pp s* = *sum* (*lookup-pp s*) (*keys-pp s*)
by (*transfer*, *transfer*, *simp add*: *deg-fun-def supp-fun-def*)

lemma *deg-pp-zero* [*simp*]: *deg-pp 0* = *0*
by (*transfer*, *fact deg-pm-zero*)

lemma *deg-pp-eq-0-iff* [*simp*]: *deg-pp s* = *0* ⟷ *s* = *0* **for** *s*::(*'a*, *'b*::*add-linorder-min*) *pp*
by (*transfer*, *fact deg-pm-eq-0-iff*)

lemma *deg-pp-plus*: *deg-pp* (*s* + *t*) = *deg-pp s* + *deg-pp* (*t*::(*'a*, *'b*::*comm-monoid-add*) *pp*)
by (*transfer*, *fact deg-pm-plus*)

lemma *deg-pp-single*: *deg-pp* (*single-pp x k*) = *k*
by (*transfer*, *fact deg-pm-single*)

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* **.**

lift-definition *dlex-pp-strict* :: (*'a*::*linorder*, *'b*::{*ordered-comm-monoid-add*,*linorder*}) *pp* ⇒ (*'a*, *'b*) *pp* ⇒ *bool*

is *dlex-pm-strict* **.**

lemma *dlex-pp-alt*: *dlex-pp s t* ⟷ (*deg-pp s* < *deg-pp t* ∨ (*deg-pp s* = *deg-pp t* ∧ *lex-pp s t*))
by *transfer* (*simp only*: *dlex-pm-def dord-pm-alt*)

lemma *dlex-pp-refl*: *dlex-pp s s*
by (*transfer*) (*fact dlex-pm-refl*)

lemma *dlex-pp-antisym*: *dlex-pp s t* ⟹ *dlex-pp t s* ⟹ *s* = *t*
by (*transfer*, *elim dlex-pm-antisym*)

lemma *dlex-pp-trans*: *dlex-pp s t* ⟹ *dlex-pp t u* ⟹ *dlex-pp s u*
by (*transfer*, *rule dlex-pm-trans*)

lemma *dlex-pp-lin*: *dlex-pp s t* ∨ *dlex-pp t s*
by (*transfer*, *fact dlex-pm-lin*)

corollary *dlex-pp-strict-alt* [*code*]: *dlex-pp-strict s t* = (¬ *dlex-pp t s*)
by (*transfer*, *fact dlex-pm-strict-alt*)

lemma *dlex-pp-zero-min*: *dlex-pp 0 s*
for *s t*::(-, -::*add-linorder-min*) *pp*
by (*transfer*, *fact dlex-pm-zero-min*)

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*
by (*transfer*, *rule dlex-pm-plus-monotone*)

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* **.**

lift-definition *drlex-pp-strict* :: (*'a*::*linorder*, *'b*::{*ordered-comm-monoid-add*,*linorder*}) *pp* ⇒ (*'a*, *'b*) *pp* ⇒ *bool*
is *drlex-pm-strict* **.**

lemma *drlex-pp-alt*: *drlex-pp s t* ⟷ (*deg-pp s* < *deg-pp t* ∨ (*deg-pp s* = *deg-pp t* ∧ *lex-pp t s*))
by *transfer* (*simp only*: *drlex-pm-def dord-pm-alt*)

lemma *drlex-pp-refl*: *drlex-pp s s*
by (*transfer*, *fact drlex-pm-refl*)

lemma *drlex-pp-antisym*: *drlex-pp s t* ⟹ *drlex-pp t s* ⟹ *s* = *t*
by (*transfer*, *rule drlex-pm-antisym*)

lemma *drlex-pp-trans*: *drlex-pp s t* ⟹ *drlex-pp t u* ⟹ *drlex-pp s u*
  by (*transfer*, *rule drlex-pm-trans*)

lemma *drlex-pp-lin*: *drlex-pp s t* ∨ *drlex-pp t s*
  by (*transfer*, *fact drlex-pm-lin*)

corollary *drlex-pp-strict-alt* [*code*]: *drlex-pp-strict s t* = (¬ *drlex-pp t s*)
  by (*transfer*, *fact drlex-pm-strict-alt*)

lemma *drlex-pp-zero-min*: *drlex-pp 0 s*
  for *s t*::(-, -::*add-linorder-min*) *pp*
  by (*transfer*, *fact drlex-pm-zero-min*)

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*
  by (*transfer*, *rule drlex-pm-plus-monotone*)

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*. 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* ⇒ *'a* ⇒ *bool*) ⇒ *'a list* ⇒ *'a* **where**
  min-list-param rel (*x* # *xs*) = (*case xs of* [] ⇒ *x* | - ⇒ (*let m* = *min-list-param rel xs in if rel x m then x else m*))

lemma *min-list-param-in*:

assumes *xs* ≠ []
shows *min-list-param rel xs* ∈ *set xs*
using *assms*
proof (*induct xs*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons x xs*)
show *?case*
proof (*simp add*: *min-list-param.simps*[*of rel x xs*] *Let-def del*: *min-list-param.simps set-simps*(*2*) *split*: *list.split*,
intro conjI impI allI, *simp*, *simp*)
fix *y ys*
assume *xs*: *xs* = *y* # *ys*
have *min-list-param rel* (*y* # *ys*) = *min-list-param rel xs* **by** (*simp only*: *xs*)
also have ... ∈ *set xs* **by** (*rule Cons*(*1*), *simp add*: *xs*)
also have ... ⊆ *set* (*x* # *y* # *ys*) **by** (*auto simp*: *xs*)
finally show *min-list-param rel* (*y* # *ys*) ∈ *set* (*x* # *y* # *ys*) **.**
qed
qed

lemma *min-list-param-minimal*:
assumes *transp rel* **and** ⋀*x y*. *x* ∈ *set xs* ⟹ *y* ∈ *set xs* ⟹ *rel x y* ∨ *rel y x*
and *z* ∈ *set xs*
shows *rel* (*min-list-param rel xs*) *z*
using *assms*(*2*, *3*)
proof (*induct xs*)
case *Nil*
from *Nil*(*2*) **show** *?case* **by** *simp*
next
case (*Cons x xs*)
from *Cons*(*3*) **have** *disj1*: *z* = *x* ∨ *z* ∈ *set xs* **by** *simp*
have *x* ∈ *set* (*x* # *xs*) **by** *simp*
hence *disj2*: *rel x z* ∨ *rel z x* **using** *Cons*(*3*) **by** (*rule Cons*(*2*))
have ∗: *rel* (*min-list-param rel xs*) *z* **if** *z* ∈ *set xs* **using** - *that*
proof (*rule Cons*(*1*))
fix *a b*
assume *a* ∈ *set xs* **and** *b* ∈ *set xs*
hence *a* ∈ *set* (*x* # *xs*) **and** *b* ∈ *set* (*x* # *xs*) **by** *simp-all*
thus *rel a b* ∨ *rel b a* **by** (*rule Cons*(*2*))
qed
show *?case*
proof (*simp add*: *min-list-param.simps*[*of rel x xs*] *Let-def del*: *min-list-param.simps set-simps*(*2*) *split*: *list.split*,
intro conjI impI allI)
assume *xs* = []
with *disj1 disj2* **show** *rel x z* **by** *simp*
next
fix *y ys*

```
    assume xs = y # ys and rel x (min-list-param rel (y # ys))
    hence rel x (min-list-param rel xs) by simp
    from disj1 show rel x z
    proof
      assume z = x
      thus ?thesis using disj2 by simp
    next
      assume z ∈ set xs
      hence rel (min-list-param rel xs) z by (rule *)
      with assms(1) ‹rel x (min-list-param rel xs)› show ?thesis by (rule transpD)
    qed
  next
    fix y ys
    assume xs: xs = y # ys and ¬ rel x (min-list-param rel (y # ys))
    from disj1 show rel (min-list-param rel (y # ys)) z
    proof
      assume z = x
      have min-list-param rel (y # ys) ∈ set (y # ys) by (rule min-list-param-in,
simp)
      hence min-list-param rel (y # ys) ∈ set (x # xs) by (simp add: xs)
        with ‹x ∈ set (x # xs)› have rel x (min-list-param rel (y # ys)) ∨ rel
(min-list-param rel (y # ys)) x
        by (rule Cons(2))
      with ‹¬ rel x (min-list-param rel (y # ys))› have rel (min-list-param rel (y
# ys)) x by simp
      thus ?thesis by (simp only: ‹z = x›)
    next
      assume z ∈ set xs
      hence rel (min-list-param rel xs) z by (rule *)
      thus ?thesis by (simp only: xs)
    qed
  qed
qed

definition comp-of-ord :: ('a ⇒ 'a ⇒ bool) ⇒ '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 (λx y. le x y ∧ ¬ le y x)
   by (intro ext, auto simp: comp-of-ord-def comp-of-ords-def intro: assms anti-
sympD)

lemma comparator-converse:
  assumes comparator cmp
  shows comparator (λx y. cmp y x)
proof −
  from assms interpret comp?: comparator cmp .
  show ?thesis by (unfold-locales, auto simp: comp.eq comp.sym intro: comp-trans)
```

qed

lemma *comparator-composition*:
  assumes *comparator cmp* **and** *inj f*
  shows *comparator* ($\lambda x\ y.\ cmp\ (f\ x)\ (f\ y)$)
proof −
  from *assms*(*1*) **interpret** *comp?*: *comparator cmp* **.**
  from *assms*(*2*) **have** ∗: $x = y$ **if** $f\ x = f\ y$ **for** *x y* **using** *that* **by** (*rule injD*)
  show *?thesis* **by** (*unfold-locales*, *auto simp*: *comp.sym comp.eq* ∗ *intro*: *comp-trans*)
qed

12.2 Type *key-order*

typedef *'a key-order* = {*compare* :: *'a comparator*. *comparator compare*}
  morphisms *key-compare Abs-key-order*
proof −
  from *well-order-on* **obtain** *r* **where** *well-order-on* (*UNIV*::*'a set*) *r* **..**
  hence *linear-order r* **by** (*simp only*: *well-order-on-def*)
  hence *lin*: $(x, y) \in r \lor (y, x) \in r$ **for** *x y*
   by (*metis Diff-iff Linear-order-in-diff-Id UNIV-I* ‹*well-order r*› *well-order-on-Field*)
  have *antisym*: $(x, y) \in r \Longrightarrow (y, x) \in r \Longrightarrow x = y$ **for** *x y*
    by (*meson* ‹*linear-order r*› *antisymD linear-order-on-def partial-order-on-def*)
  have *trans*: $(x, y) \in r \Longrightarrow (y, z) \in r \Longrightarrow (x, z) \in r$ **for** *x y z*
   by (*meson* ‹*linear-order r*› *linear-order-on-def order-on-defs*(*1*) *partial-order-on-def trans-def*)
  define *comp* **where** *comp* = ($\lambda x\ y.$ *if* $(x, y) \in r$ *then if* $(y, x) \in r$ *then Eq else Lt else Gt*)
  show *?thesis*
  proof (*rule*, *simp*)
    show *comparator comp*
    proof (*standard*, *simp-all add*: *comp-def split*: *if-splits*, *intro impI*)
      fix *x y*
      assume $(x, y) \in r$ **and** $(y, x) \in r$
      thus $x = y$ **by** (*rule antisym*)
    next
      fix *x y*
      assume $(x, y) \notin r$
      with *lin* **show** $(y, x) \in r$ **by** *blast*
    next
      fix *x y z*
      assume $(y, x) \notin r$ **and** $(z, y) \notin r$
      assume $(x, y) \in r$ **and** $(y, z) \in r$
      hence $(x, z) \in r$ **by** (*rule trans*)
      moreover have $(z, x) \notin r$
      proof
        assume $(z, x) \in r$
        with ‹$(x, z) \in r$› **have** $x = z$ **by** (*rule antisym*)
        from ‹$(z, y) \notin r$› ‹$(x, y) \in r$› **show** *False* **unfolding** ‹$x = z$› **..**
      qed

ultimately show $(z, x) \notin r \wedge ((z, x) \notin r \longrightarrow (x, z) \in r)$ **by** *simp*
qed
qed
qed

lemma *comparator-key-compare* [*simp*, *intro*!]: *comparator* (*key-compare ko*)
using *key-compare*[*of ko*] **by** *simp*

instantiation *key-order* :: (*type*) *equal*
begin

definition *equal-key-order* :: *'a key-order* ⇒ *'a key-order* ⇒ *bool* **where** *equal-key-order* = (=)

instance by (*standard*, *simp add*: *equal-key-order-def*)

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** *λc x y. c y x*
by (*fact comparator-converse*)

instance ..

end

lift-definition *le-of-key-order* :: *'a key-order* ⇒ *'a* ⇒ *'a* ⇒ *bool* **is** *λcmp. le-of-comp cmp* **.**

lift-definition *lt-of-key-order* :: *'a key-order* ⇒ *'a* ⇒ *'a* ⇒ *bool* **is** *λcmp. lt-of-comp cmp* **.**

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*
by (*fact comparator-of*)

interpretation *key-order-lin*: *linorder le-of-key-order ko lt-of-key-order ko*
proof *transfer*
fix *comp*::*'a comparator*
assume *comparator comp*
then interpret *comp*: *comparator comp* **.**
show *class.linorder comp.le comp.lt* **by** (*fact comp.linorder*)
qed

lemma *le-of-key-order-alt*: *le-of-key-order ko x y* = (*key-compare ko x y* ≠ *Gt*)
  by (*transfer*, *simp add*: *comparator.nGt-le-conv*)

lemma *lt-of-key-order-alt*: *lt-of-key-order ko x y* = (*key-compare ko x y* = *Lt*)
  by (*transfer*, *meson comparator.Lt-lt-conv*)

lemma *key-compare-Gt*: *key-compare ko x y* = *Gt* ⟷ *key-compare ko y x* = *Lt*
  by (*transfer*, *meson comparator.nGt-le-conv comparator.nLt-le-conv*)

lemma *key-compare-Eq*: *key-compare ko x y* = *Eq* ⟷ *x* = *y*
  by (*transfer*, *simp add*: *comparator.eq*)

lemma *key-compare-same* [*simp*]: *key-compare ko x x* = *Eq*
  by (*simp add*: *key-compare-Eq*)

lemma *uminus-key-compare* [*simp*]: *invert-order* (*key-compare ko x y*) = *key-compare ko y x*
  by (*transfer*, *simp add*: *comparator.sym*)

lemma *key-compare-uminus* [*simp*]: *key-compare* (− *ko*) *x y* = *key-compare ko y x*
  by (*transfer*, *rule refl*)

lemma *uminus-key-order-sameD*:
  assumes − *ko* = (*ko*::′*a key-order*)
  shows *x* = (*y*::′*a*)
proof (*rule ccontr*)
  assume *x* ≠ *y*
  hence *key-compare ko x y* ≠ *Eq* **by** (*simp add*: *key-compare-Eq*)
  hence *key-compare ko x y* ≠ *invert-order* (*key-compare ko x y*)
    by (*metis invert-order.elims order.distinct*(*5*))
  also have *invert-order* (*key-compare ko x y*) = *key-compare* (− *ko*) *x y* **by** *simp*
  finally have − *ko* ≠ *ko* **by** (*auto simp del*: *key-compare-uminus*)
  thus *False* **using** *assms* **..**
qed

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 −
  have *eq*: *key-compare* (*key-order-of-ord ord*) = *comp-of-ord ord*
    unfolding *key-order-of-ord-def comp-of-ord-eq-comp-of-ords*[*OF assms*(*1*)]
  proof (*rule Abs-key-order-inverse*, *simp*, *rule comp-of-ords*, *unfold-locales*)
    fix *x*
    from *assms*(*3*) **show** *ord x x* **by** *blast*
  next
    fix *x y z*
    assume *ord x y* **and** *ord y z*

with *assms*(*2*) **show** *ord x z* **by** (*rule transpD*)
next
fix *x y*
assume *ord x y* **and** *ord y x*
with *assms*(*1*) **show** *x* = *y* **by** (*rule antisympD*)
qed (*rule refl*, *rule assms*(*3*))
have ∗: *x* = *y* **if** *ord x y* **and** *ord y x* **for** *x y* **using** *assms*(*1*) *that* **by** (*rule antisympD*)
show *?thesis* **by** (*rule*, *rule*, *auto simp*: *eq comp-of-ord-def intro*: ∗)
qed

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*)
by (*transfer*, *intro ext*, *fact comparator-of-def*)

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*
unfolding *oalist-inv-raw-def* **using** *assms* **unfolding** *fst-conv snd-conv* **by** *blast*

lemma *oalist-inv-rawD1*:
assumes *oalist-inv-raw xs*
shows *0* ∉ *snd* ‘ *set xs*
using *assms* **unfolding** *oalist-inv-raw-def fst-conv* **by** *blast*

lemma *oalist-inv-rawD2*:
assumes *oalist-inv-raw xs*
shows *sorted-wrt lt* (*map fst xs*)
using *assms* **unfolding** *oalist-inv-raw-def fst-conv snd-conv* **by** *blast*

lemma *oalist-inv-raw-Nil*: *oalist-inv-raw* []
by (*simp add*: *oalist-inv-raw-def*)

lemma *oalist-inv-raw-singleton*: *oalist-inv-raw* [(*k*, *v*)] ←→ (*v* ≠ *0*)
by (*auto simp*: *oalist-inv-raw-def*)

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 (*rule oalist-inv-rawI*)

from *assms*(*1*) **have** *0* ∉ *snd* ‘ *set xs* **by** (*rule oalist-inv-rawD1*)
with *assms*(*2*) **show** *0* ∉ *snd* ‘ *set* ((*k*, *v*) # *xs*) **by** *simp*
next
show *sorted-wrt lt* (*map fst* ((*k*, *v*) # *xs*))
proof (*cases xs* = [])
case *True*
thus *?thesis* **by** *simp*
next
case *False*
then obtain *k′ v′ xs′* **where** *xs*: *xs* = (*k′*, *v′*) # *xs′* **by** (*metis list.exhaust prod.exhaust*)
from *assms*(*3*)[*OF False*] **have** *lt k k′* **by** (*simp add*: *xs*)
moreover from *assms*(*1*) **have** *sorted-wrt lt* (*map fst xs*) **by** (*rule oalist-inv-rawD2*)
ultimately show *sorted-wrt lt* (*map fst* ((*k*, *v*) # *xs*))
by (*simp add*: *xs sorted-wrt2*[*OF transp-on-less*] *del*: *sorted-wrt.simps*)
qed
qed

lemma *oalist-inv-raw-ConsD1*:
assumes *oalist-inv-raw* (*x* # *xs*)
shows *oalist-inv-raw xs*
proof (*rule oalist-inv-rawI*)
from *assms* **have** *0* ∉ *snd* ‘ *set* (*x* # *xs*) **by** (*rule oalist-inv-rawD1*)
thus *0* ∉ *snd* ‘ *set xs* **by** *simp*
next
from *assms* **have** *sorted-wrt lt* (*map fst* (*x* # *xs*)) **by** (*rule oalist-inv-rawD2*)
thus *sorted-wrt lt* (*map fst xs*) **by** *simp*
qed

lemma *oalist-inv-raw-ConsD2*:
assumes *oalist-inv-raw* ((*k*, *v*) # *xs*)
shows *v* ≠ *0*
proof −
from *assms* **have** *0* ∉ *snd* ‘ *set* ((*k*, *v*) # *xs*) **by** (*rule oalist-inv-rawD1*)
thus *?thesis* **by** *auto*
qed

lemma *oalist-inv-raw-ConsD3*:
assumes *oalist-inv-raw* ((*k*, *v*) # *xs*) **and** *k′* ∈ *fst* ‘ *set xs*
shows *lt k k′*
proof −
from *assms*(*2*) **obtain** *x* **where** *x* ∈ *set xs* **and** *k′* = *fst x* **by** *fastforce*
from *assms*(*1*) **have** *sorted-wrt lt* (*map fst* ((*k*, *v*) # *xs*)) **by** (*rule oalist-inv-rawD2*)
hence ∀ *x*∈*set xs. lt k* (*fst x*) **by** *simp*
hence *lt k* (*fst x*) **using** ‹*x* ∈ *set xs*› **..**
thus *?thesis* **by** (*simp only*: ‹*k′* = *fst x*›)
qed

lemma *oalist-inv-raw-tl*:
  assumes *oalist-inv-raw xs*
  shows *oalist-inv-raw* (*tl xs*)
proof (*rule oalist-inv-rawI*)
  from *assms* **have** *0* ∉ *snd* ‘ *set xs* **by** (*rule oalist-inv-rawD1*)
  thus *0* ∉ *snd* ‘ *set* (*tl xs*) **by** (*metis* (*no-types*, *lifting*) *image-iff list.set-sel*(*2*) *tl-Nil*)
next
  show *sorted-wrt lt* (*map fst* (*tl xs*))
    by (*metis hd-Cons-tl oalist-inv-rawD2 oalist-inv-raw-ConsD1 assms tl-Nil*)
qed

lemma *oalist-inv-raw-filter*:
  assumes *oalist-inv-raw xs*
  shows *oalist-inv-raw* (*filter P xs*)
proof (*rule oalist-inv-rawI*)
  from *assms* **have** *0* ∉ *snd* ‘ *set xs* **by** (*rule oalist-inv-rawD1*)
  thus *0* ∉ *snd* ‘ *set* (*filter P xs*) **by** *auto*
next
  from *assms* **have** *sorted-wrt lt* (*map fst xs*) **by** (*rule oalist-inv-rawD2*)
  thus *sorted-wrt lt* (*map fst* (*filter P xs*)) **by** (*induct xs*, *simp*, *simp*)
qed

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 (*rule oalist-inv-rawI*)
  show *0* ∉ *snd* ‘ *set* (*map f xs*)
  proof (*simp*, *rule*)
    assume *0* ∈ *snd* ‘ *f* ‘ *set xs*
    then obtain *a* **where** *a* ∈ *set xs* **and** *snd* (*f a*) = *0* **by** *fastforce*
    from *this*(*2*) **have** *snd a* = *0* **by** (*rule assms*(*2*))
    from *assms*(*1*) **have** *0* ∉ *snd* ‘ *set xs* **by** (*rule oalist-inv-rawD1*)
    moreover from ‹*a* ∈ *set xs*› **have** *0* ∈ *snd* ‘ *set xs* **by** (*simp add*: ‹*snd a* = *0*›[*symmetric*])
    ultimately show *False* **..**
  qed
next
  from *assms*(*1*) **have** *sorted-wrt lt* (*map fst xs*) **by** (*rule oalist-inv-rawD2*)
  hence *sorted-wrt* (λ*x y*. *comp* (*fst x*) (*fst y*) = *Lt*) *xs*
    by (*simp only*: *sorted-wrt-map Lt-lt-conv*)
  thus *sorted-wrt lt* (*map fst* (*map f xs*))
    by (*simp add*: *sorted-wrt-map Lt-lt-conv*[*symmetric*] *assms*(*3*))
qed

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*
using *assms*(*1*)
proof (*induct xs*)
case *Nil*
from *assms*(*2*) **show** *?case* **.**
next
case (*Cons x xs*)
obtain *k v* **where** *x*: *x* = (*k*, *v*) **by** *fastforce*
from *Cons*(*2*) **have** *oalist-inv-raw* ((*k*, *v*) # *xs*) **and** *oalist-inv-raw xs* **and** *v* $\neq$ *0* **unfolding** *x*
by (*this*, *rule oalist-inv-raw-ConsD1*, *rule oalist-inv-raw-ConsD2*)
moreover from *Cons*(*2*) **have** *lt k k'* **if** *k'* $\in$ *fst* ' *set xs* **for** *k'* **using** *that*
unfolding *x* **by** (*rule oalist-inv-raw-ConsD3*)
moreover from ‹*oalist-inv-raw xs*› **have** *P xs* **by** (*rule Cons*(*1*))
ultimately show *?case* **unfolding** *x* **by** (*rule assms*(*3*))
qed

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* × '*b*) *list* $\Rightarrow$ '*a* $\Rightarrow$ '*b*::*zero*
where *lookup-dflt xs k* = (*case map-of xs k of Some v* $\Rightarrow$ *v* | *None* $\Rightarrow$ *0*)

lookup-dflt is only an auxiliary function needed for proving some lemmas.

fun *lookup-pair* :: ('*a* × '*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* × '*b*) $\Rightarrow$ ('*a* × '*b*) *list* $\Rightarrow$ ('*a* × '*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* × '*b*) *list* $\Rightarrow$ ('*a* × '*b*::*zero*) *list*
where *sort-oalist xs* = *foldr update-by-pair xs* []

fun *update-by-fun-pair* :: *'a* ⇒ (*'b* ⇒ *'b*) ⇒ (*'a* × *'b*) *list* ⇒ (*'a* × *'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* ⇒ (*let v* = *f 0 in if v* = *0 then* (*k'*, *v'*) # *xs else* (*k*, *v*) # (*k'*, *v'*) # *xs*)
    | *Eq* ⇒ (*let v* = *f v' in if v* = *0 then xs else* (*k*, *v*) # *xs*)
    | *Gt* ⇒ (*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* 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* :: $('a \Rightarrow (('b, 'c)\ comp\text{-}opt)) \Rightarrow (('a \times 'b::zero)\ list, ('a \times 'c::zero)\ list)\ comp\text{-}opt$
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* :: $('a \Rightarrow 'b \Rightarrow 'c \Rightarrow bool) \Rightarrow ('a \times 'b::zero)\ list \Rightarrow ('a \times 'c::zero)\ list \Rightarrow bool$
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*)
using *assms*
proof (*induct xs rule*: *oalist-inv-raw-induct*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons k′ v′ xs*)
show *?case*
proof (*simp add*: *Cons*(*3*) *eq split*: *order.splits*, *rule*, *simp-all only*: *atomize-imp*[*symmetric*])
assume *comp k k′* = *Lt*
hence *k* ≠ *k′* **by** *auto*
moreover have *k* ∉ *fst* ‘ *set xs*
proof

assume $k \in fst$ ‘ $set\ xs$
hence $lt\ k'\ k$ **by** ($rule\ Cons(4)$)
with ‹$comp\ k\ k' = Lt$› **show** $False$ **by** ($simp\ add$: Lt-lt-$conv$)
qed
ultimately show $k \neq k' \wedge k \notin fst$ ‘ $set\ xs$ **..**
next
assume $comp\ k\ k' = Gt$
hence $k \neq k'$ **by** $auto$
thus ($lookup$-$pair\ xs\ k = 0$) = ($k \neq k' \wedge k \notin fst$ ‘ $set\ xs$) **by** ($simp\ add$: $Cons(5)$)
qed
qed

lemma *lookup-pair-eq-value*:
assumes *oalist-inv-raw xs* **and** $v \neq 0$
shows *lookup-pair xs k* $= v \longleftrightarrow ((k, v) \in set\ xs)$
using *assms*(*1*)
proof (*induct xs rule*: *oalist-inv-raw-induct*)
case *Nil*
from *assms*(*2*) **show** *?case* **by** *simp*
next
case (*Cons k' v' xs*)
have ∗: $(k', u) \notin set\ xs$ **for** u
proof
assume $(k', u) \in set\ xs$
hence $fst\ (k', u) \in fst$ ‘ $set\ xs$ **by** *fastforce*
hence $k' \in fst$ ‘ $set\ xs$ **by** *simp*
hence $lt\ k'\ k'$ **by** (*rule Cons*(*4*))
thus *False* **by** (*simp add*: *lt-of-key-order-alt*[*symmetric*])
qed
show *?case*
proof (*simp add*: *assms*(*2*) *Cons*(*5*) *eq split*: *order.split*, *intro conjI impI*)
assume $comp\ k\ k' = Lt$
show $(k, v) \notin set\ xs$
proof
assume $(k, v) \in set\ xs$
hence $fst\ (k, v) \in fst$ ‘ $set\ xs$ **by** *fastforce*
hence $k \in fst$ ‘ $set\ xs$ **by** *simp*
hence $lt\ k'\ k$ **by** (*rule Cons*(*4*))
with ‹$comp\ k\ k' = Lt$› **show** *False* **by** (*simp add*: *Lt-lt-conv*)
qed
qed (*auto simp*: ∗)
qed

lemma *lookup-pair-eq-valueI*:
assumes *oalist-inv-raw xs* **and** $(k, v) \in set\ xs$
shows *lookup-pair xs k* $= v$
proof −
from *assms*(*2*) **have** $v \in snd$ ‘ $set\ xs$ **by** *force*
moreover from *assms*(*1*) **have** $0 \notin snd$ ‘ $set\ xs$ **by** (*rule oalist-inv-rawD1*)

ultimately have *v* ≠ *0* **by** *blast*
with *assms* **show** *?thesis* **by** (*simp add*: *lookup-pair-eq-value*)
qed

lemma *lookup-dflt-eq-lookup-pair*:
assumes *oalist-inv-raw xs*
shows *lookup-dflt xs* = *lookup-pair xs*
proof (*rule*, *simp add*: *lookup-dflt-def split*: *option.split*, *intro conjI impI allI*)
fix *k*
assume *map-of xs k* = *None*
with *assms* **show** *lookup-pair xs k* = *0* **by** (*simp add*: *lookup-pair-eq-0 map-of-eq-None-iff*)
next
fix *k v*
assume *map-of xs k* = *Some v*
hence (*k*, *v*) ∈ *set xs* **by** (*rule map-of-SomeD*)
with *assms* **have** *lookup-pair xs k* = *v* **by** (*rule lookup-pair-eq-valueI*)
thus *v* = *lookup-pair xs k* **by** (*rule HOL.sym*)
qed

lemma *lookup-pair-inj*:
assumes *oalist-inv-raw xs* **and** *oalist-inv-raw ys* **and** *lookup-pair xs* = *lookup-pair ys*
shows *xs* = *ys*
using *assms*
proof (*induct xs arbitrary*: *ys rule*: *oalist-inv-raw-induct*)
case *Nil*
thus *?case*
proof (*induct ys rule*: *oalist-inv-raw-induct*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons k′ v′ ys*)
have *v′* = *lookup-pair* ((*k′*, *v′*) # *ys*) *k′* **by** *simp*
also have ... = *lookup-pair* [] *k′* **by** (*simp only*: *Cons*(*6*))
also have ... = *0* **by** *simp*
finally have *v′* = *0* **.**
with *Cons*(*3*) **show** *?case* **..**
qed
next
case ∗: (*Cons k v xs*)
from ∗(*6*, *7*) **show** *?case*
proof (*induct ys rule*: *oalist-inv-raw-induct*)
case *Nil*
have *v* = *lookup-pair* ((*k*, *v*) # *xs*) *k* **by** *simp*
also have ... = *lookup-pair* [] *k* **by** (*simp only*: *Nil*)
also have ... = *0* **by** *simp*
finally have *v* = *0* **.**
with ∗(*3*) **show** *?case* **..**
next

```
case (Cons k' v' ys)
show ?case
proof (cases comp k k')
  case Lt
  hence ¬ lt k' k by (simp add: Lt-lt-conv)
  with Cons(4) have k ∉ fst ` set ys by blast
  moreover from Lt have k ≠ k' by auto
  ultimately have k ∉ fst ` set ((k', v') # ys) by simp
  hence 0 = lookup-pair ((k', v') # ys) k
    by (simp add: lookup-pair-eq-0[OF Cons(1)] del: lookup-pair.simps)
  also have ... = lookup-pair ((k, v) # xs) k by (simp only: Cons(6))
  also have ... = v by simp
  finally have v = 0 by simp
  with *(3) show ?thesis ..
next
  case Eq
  hence k' = k by (simp only: eq)
  have v' = lookup-pair ((k', v') # ys) k' by simp
  also have ... = lookup-pair ((k, v) # xs) k by (simp only: Cons(6) ‹k' = k›)
  also have ... = v by simp
  finally have v' = v .
  moreover note ‹k' = k›
  moreover from Cons(2) have xs = ys
  proof (rule *(5))
    show lookup-pair xs = lookup-pair ys
    proof
      fix k0
      show lookup-pair xs k0 = lookup-pair ys k0
      proof (cases lt k k0)
        case True
        hence eq: comp k0 k = Gt
          by (simp add: Gt-lt-conv)
        have lookup-pair xs k0 = lookup-pair ((k, v) # xs) k0 by (simp add: eq)
         also have ... = lookup-pair ((k, v') # ys) k0 by (simp only: Cons(6)
‹k' = k›)
        also have ... = lookup-pair ys k0 by (simp add: eq)
        finally show ?thesis .
      next
        case False
        with *(4) have k0 ∉ fst ` set xs by blast
       with *(2) have eq: lookup-pair xs k0 = 0 by (simp add: lookup-pair-eq-0)
        from False Cons(4) have k0 ∉ fst ` set ys unfolding ‹k' = k› by blast
       with Cons(2) have lookup-pair ys k0 = 0 by (simp add: lookup-pair-eq-0)
        with eq show ?thesis by simp
      qed
    qed
  qed
  ultimately show ?thesis by simp
next
```

case *Gt*
hence ¬ *lt k k′* **by** (*simp add*: *Gt-lt-conv*)
with ∗(*4*) **have** *k′* ∉ *fst* ‘ *set xs* **by** *blast*
moreover from *Gt* **have** *k′* ≠ *k* **by** *auto*
ultimately have *k′* ∉ *fst* ‘ *set* ((*k*, *v*) # *xs*) **by** *simp*
hence *0* = *lookup-pair* ((*k*, *v*) # *xs*) *k′*
by (*simp add*: *lookup-pair-eq-0*[*OF* ∗(*1*)] *del*: *lookup-pair.simps*)
also have ... = *lookup-pair* ((*k′*, *v′*) # *ys*) *k′* **by** (*simp only*: *Cons*(*6*))
also have ... = *v′* **by** *simp*
finally have *v′* = *0* **by** *simp*
with *Cons*(*3*) **show** *?thesis* **..**
qed
qed
qed

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 −
from *assms* **have** *1*: *oalist-inv-raw* (*tl xs*) **by** (*rule oalist-inv-raw-tl*)
show *?thesis*
proof (*split if-split*, *intro conjI impI*)
assume ∗: ∀ *x*∈*fst* ‘ *set xs*. *le k x*
show *lookup-pair* (*tl xs*) *k* = *0*
proof (*simp add*: *lookup-pair-eq-0*[*OF 1*], *rule*)
assume *k-in*: *k* ∈ *fst* ‘ *set* (*tl xs*)
hence *xs* ≠ [] **by** *auto*
then obtain *k′ v′ ys* **where** *xs*: *xs* = (*k′*, *v′*) # *ys* **using** *prod.exhaust list.exhaust* **by** *metis*
have *k′* ∈ *fst* ‘ *set xs* **unfolding** *xs* **by** *fastforce*
with ∗ **have** *le k k′* **..**
from *assms* **have** *oalist-inv-raw* ((*k′*, *v′*) # *ys*) **by** (*simp only*: *xs*)
moreover from *k-in* **have** *k* ∈ *fst* ‘ *set ys* **by** (*simp add*: *xs*)
ultimately have *lt k′ k* **by** (*rule oalist-inv-raw-ConsD3*)
with ‹*le k k′*› **show** *False* **by** *simp*
qed
next
assume ¬ (∀ *k′*∈*fst* ‘ *set xs*. *le k k′*)
hence ∃ *x*∈*fst* ‘ *set xs*. ¬ *le k x* **by** *simp*
then obtain *k″* **where** *k″-in*: *k″* ∈ *fst* ‘ *set xs* **and** ¬ *le k k″* **..**
from *this*(*2*) **have** *lt k″ k* **by** *simp*
from *k″-in* **have** *xs* ≠ [] **by** *auto*
then obtain *k′ v′ ys* **where** *xs*: *xs* = (*k′*, *v′*) # *ys* **using** *prod.exhaust list.exhaust* **by** *metis*
from *k″-in* **have** *k″* = *k′* ∨ *k″* ∈ *fst* ‘ *set ys* **by** (*simp add*: *xs*)
hence *lt k′ k*
proof
assume *k″* = *k′*

with ‹*lt k'' k*› **show** *?thesis* **by** *simp*
next
from *assms* **have** *oalist-inv-raw* ((*k'*, *v'*) # *ys*) **by** (*simp only*: *xs*)
moreover assume $k'' \in$ *fst* ‘ *set ys*
ultimately have *lt k' k''* **by** (*rule oalist-inv-raw-ConsD3*)
thus *?thesis* **using** ‹*lt k'' k*› **by** (*rule less-trans*)
qed
hence *comp k k'* = *Gt* **by** (*simp add*: *Gt-lt-conv*)
thus *lookup-pair* (*tl xs*) *k* = *lookup-pair xs k* **by** (*simp add*: *xs lt-of-key-order-alt*)
qed
qed

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 −
from *assms* **have** *1*: *oalist-inv-raw* (*tl xs*) **by** (*rule oalist-inv-raw-tl*)
show *?thesis*
proof (*split if-split*, *intro conjI impI*)
assume *k*: *k* = *fst* (*hd xs*)
show *lookup-pair* (*tl xs*) *k* = *0*
proof (*simp add*: *lookup-pair-eq-0*[*OF 1*], *rule*)
assume *k-in*: $k \in$ *fst* ‘ *set* (*tl xs*)
hence $xs \neq []$ **by** *auto*
then obtain *k' v' ys* **where** *xs*: *xs* = (*k'*, *v'*) # *ys* **using** *prod.exhaust list.exhaust* **by** *metis*
from *assms* **have** *oalist-inv-raw* ((*k'*, *v'*) # *ys*) **by** (*simp only*: *xs*)
moreover from *k-in* **have** $k' \in$ *fst* ‘ *set ys* **by** (*simp add*: *k xs*)
ultimately have *lt k' k'* **by** (*rule oalist-inv-raw-ConsD3*)
thus *False* **by** *simp*
qed
next
assume $k \neq$ *fst* (*hd xs*)
show *lookup-pair* (*tl xs*) *k* = *lookup-pair xs k*
proof (*cases xs* = [])
case *True*
show *?thesis* **by** (*simp add*: *True*)
next
case *False*
then obtain *k' v' ys* **where** *xs*: *xs* = (*k'*, *v'*) # *ys* **using** *prod.exhaust list.exhaust* **by** *metis*
show *?thesis*
proof (*simp add*: *xs eq Lt-lt-conv split*: *order.split*, *intro conjI impI*)
from ‹$k \neq$ *fst* (*hd xs*)› **have** $k \neq k'$ **by** (*simp add*: *xs*)
moreover assume *k* = *k'*
ultimately show *lookup-pair ys k'* = *v'* **..**
next
assume *lt k k'*
from *assms* **have** *oalist-inv-raw ys* **unfolding** *xs* **by** (*rule oalist-inv-raw-ConsD1*)

moreover have *k* ∉ *fst* ' *set ys*
proof
assume *k* ∈ *fst* ' *set ys*
with *assms* **have** *lt k' k* **unfolding** *xs* **by** (*rule oalist-inv-raw-ConsD3*)
with ‹*lt k k'*› **show** *False* **by** *simp*
qed
ultimately show *lookup-pair ys k* = *0* **by** (*simp add*: *lookup-pair-eq-0*)
qed
qed
qed
qed

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*)
using *assms*
proof (*induct xs rule*: *oalist-inv-raw-induct*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons k' v' xs*)
show *?case*
proof (*simp add*: *Cons*(*5*) *Let-def eq split*: *order.split*, *intro conjI impI*)
show *lookup-pair xs k'* = *0*
proof (*simp add*: *lookup-pair-eq-0 Cons*(*2*), *rule*)
assume *k'* ∈ *fst* ' *set xs*
hence *lt k' k'* **by** (*rule Cons*(*4*))
thus *False* **by** *simp*
qed
next
assume *comp k k'* = *Lt*
hence *lt k k'* **by** (*simp only*: *Lt-lt-conv*)
show *lookup-pair xs k* = *0*
proof (*simp add*: *lookup-pair-eq-0 Cons*(*2*), *rule*)
assume *k* ∈ *fst* ' *set xs*
hence *lt k' k* **by** (*rule Cons*(*4*))
with ‹*lt k k'*› **show** *False* **by** *simp*
qed
qed
qed

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*))
using *assms*(*1*)
proof (*induct xs rule*: *oalist-inv-raw-induct*)

case *Nil*
show *?case* **by** (*simp add*: *assms*(*2*))
next
case (*Cons k′ v′ xs*)
obtain *k′′ v′′* **where** *f*: *f* (*k′*, *v′*) = (*k′′*, *v′′*) **by** *fastforce*
have *comp k k′* = *comp* (*fst* (*f* (*k*, *v*))) (*fst* (*f* (*k′*, *v′*)))
by (*simp add*: *assms*(*3*))
also have ... = *comp* (*fst* (*f* (*k*, *v*))) *k′′* **by** (*simp add*: *f*)
finally have *eq0*: *comp k k′* = *comp* (*fst* (*f* (*k*, *v*))) *k′′* **.**
show *?case*
proof (*simp add*: *assms*(*2*) *split*: *order.split*, *intro conjI impI*, *simp add*: *eq*)
assume *k* = *k′*
hence *lookup-pair* (*f* (*k′*, *v′*) # *map f xs*) (*fst* (*f* (*k′*, *v*))) =
lookup-pair (*f* (*k′*, *v′*) # *map f xs*) (*fst* (*f* (*k*, *v*))) **by** *simp*
also have ... = *snd* (*f* (*k′*, *v′*)) **by** (*simp add*: *f eq0*[*symmetric*], *simp add*: ‹*k*
= *k′*›)
finally show *lookup-pair* (*f* (*k′*, *v′*) # *map f xs*) (*fst* (*f* (*k′*, *v*))) = *snd* (*f* (*k′*,
v′)) **.**
qed (*simp-all add*: *f eq0 Cons*(*5*))
qed

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 (*simp add*: *eq split*: *order.split*, *intro impI*)
assume *comp k0 k* = *Lt*
from *assms* **have** *inv*: *oalist-inv-raw xs* **by** (*rule oalist-inv-raw-ConsD1*)
show *lookup-pair xs k0* = *0*
proof (*simp only*: *lookup-pair-eq-0*[*OF inv*], *rule*)
assume *k0* ∈ *fst* ‘ *set xs*
with *assms* **have** *lt k k0* **by** (*rule oalist-inv-raw-ConsD3*)
with ‹*comp k0 k* = *Lt*› **show** *False* **by** (*simp add*: *Lt-lt-conv*)
qed
qed

lemma *lookup-pair-single*: *lookup-pair* [(*k*, *v*)] *k0* = (*if k* = *k0 then v else 0*)
by (*simp add*: *eq split*: *order.split*)

12.4.2 *update-by-pair*

lemma *set-update-by-pair-subset*: *set* (*update-by-pair kv xs*) ⊆ *insert kv* (*set xs*)
proof (*induct xs arbitrary*: *kv*)
case *Nil*
obtain *k v* **where** *kv*: *kv* = (*k*, *v*) **by** *fastforce*
thus *?case* **by** *simp*
next
case (*Cons x xs*)
obtain *k′ v′* **where** *x*: *x* = (*k′*, *v′*) **by** *fastforce*
obtain *k v* **where** *kv*: *kv* = (*k*, *v*) **by** *fastforce*

have *1*: *set xs* ⊆ *insert a* (*insert b* (*set xs*)) **for** *a b* **by** *auto*
have *2*: *set* (*update-by-pair kv xs*) ⊆ *insert kv* (*insert* (*k'*, *v'*) (*set xs*)) **for** *kv*
using *Cons* **by** *blast*
show *?case* **by** (*simp add*: *x kv 1 2 split*: *order.split*)
qed

lemma *update-by-pair-sorted*:
assumes *sorted-wrt lt* (*map fst xs*)
shows *sorted-wrt lt* (*map fst* (*update-by-pair kv xs*))
using *assms*
proof (*induct xs arbitrary*: *kv*)
case *Nil*
obtain *k v* **where** *kv*: *kv* = (*k*, *v*) **by** *fastforce*
thus *?case* **by** *simp*
next
case (*Cons x xs*)
obtain *k' v'* **where** *x*: *x* = (*k'*, *v'*) **by** *fastforce*
obtain *k v* **where** *kv*: *kv* = (*k*, *v*) **by** *fastforce*
from *Cons*(*2*) **have** *1*: *sorted-wrt lt* (*k'* # (*map fst xs*)) **by** (*simp add*: *x*)
hence *2*: *sorted-wrt lt* (*map fst xs*) **using** *sorted-wrt.elims*(*3*) **by** *fastforce*
hence *3*: *sorted-wrt lt* (*map fst* (*update-by-pair* (*k*, *u*) *xs*)) **for** *u* **by** (*rule Cons*(*1*))
have *4*: *sorted-wrt lt* (*k'* # *map fst* (*update-by-pair* (*k*, *u*) *xs*))
if ∗: *comp k k'* = *Gt* **for** *u*
proof (*simp*, *intro conjI ballI*)
fix *y*
assume *y* ∈ *set* (*update-by-pair* (*k*, *u*) *xs*)
also from *set-update-by-pair-subset* **have** ... ⊆ *insert* (*k*, *u*) (*set xs*) **.**
finally have *y* = (*k*, *u*) ∨ *y* ∈ *set xs* **by** *simp*
thus *lt k'* (*fst y*)
proof
assume *y* = (*k*, *u*)
hence *fst y* = *k* **by** *simp*
with ∗ **show** *?thesis* **by** (*simp only*: *Gt-lt-conv*)
next
from *1* **have** *5*: ∀ *y* ∈ *fst* ‘ *set xs*. *lt k' y* **by** *simp*
assume *y* ∈ *set xs*
hence *fst y* ∈ *fst* ‘ *set xs* **by** *simp*
with *5* **show** *?thesis* **..**
qed
qed (*fact 3*)
show *?case*
by (*simp add*: *kv x 1 2 4 sorted-wrt2 split*: *order.split del*: *sorted-wrt.simps*,
intro conjI impI, *simp add*: *1 eq del*: *sorted-wrt.simps*, *simp add*: *Lt-lt-conv*)
qed

lemma *update-by-pair-not-0*:
assumes *0* ∉ *snd* ‘ *set xs*
shows *0* ∉ *snd* ‘ *set* (*update-by-pair kv xs*)

using *assms*
proof (*induct xs arbitrary*: *kv*)
case *Nil*
obtain *k v* **where** *kv*: *kv* = (*k*, *v*) **by** *fastforce*
thus *?case* **by** *simp*
next
case (*Cons x xs*)
obtain *k′ v′* **where** *x*: *x* = (*k′*, *v′*) **by** *fastforce*
obtain *k v* **where** *kv*: *kv* = (*k*, *v*) **by** *fastforce*
from *Cons*(*2*) **have** *1*: *v′* ≠ *0* **and** *2*: *0* ∉ *snd* ‘ *set xs* **by** (*auto simp*: *x*)
from *2* **have** *3*: *0* ∉ *snd* ‘ *set* (*update-by-pair* (*k*, *u*) *xs*) **for** *u* **by** (*rule Cons*(*1*))
show *?case* **by** (*auto simp*: *kv x 1 2 3 split*: *order.split*)
qed

corollary *oalist-inv-raw-update-by-pair*:
assumes *oalist-inv-raw xs*
shows *oalist-inv-raw* (*update-by-pair kv xs*)
proof (*rule oalist-inv-rawI*)
from *assms* **have** *0* ∉ *snd* ‘ *set xs* **by** (*rule oalist-inv-rawD1*)
thus *0* ∉ *snd* ‘ *set* (*update-by-pair kv xs*) **by** (*rule update-by-pair-not-0*)
next
from *assms* **have** *sorted-wrt lt* (*map fst xs*) **by** (*rule oalist-inv-rawD2*)
thus *sorted-wrt lt* (*map fst* (*update-by-pair kv xs*)) **by** (*rule update-by-pair-sorted*)
qed

lemma *update-by-pair-less*:
assumes *v* ≠ *0* **and** *xs* = [] ∨ *comp k* (*fst* (*hd xs*)) = *Lt*
shows *update-by-pair* (*k*, *v*) *xs* = (*k*, *v*) # *xs*
using *assms*(*2*)
proof (*induct xs*)
case *Nil*
from *assms*(*1*) **show** *?case* **by** *simp*
next
case (*Cons x xs*)
obtain *k′ v′* **where** *x*: *x* = (*k′*, *v′*) **by** *fastforce*
from *Cons*(*2*) **have** *comp k k′* = *Lt* **by** (*simp add*: *x*)
with *assms*(*1*) **show** *?case* **by** (*simp add*: *x*)
qed

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*)
using *assms*
proof (*induct xs arbitrary*: *v rule*: *oalist-inv-raw-induct*)
case *Nil*
show *?case* **by** (*simp split*: *order.split*, *simp add*: *eq*)
next
case (*Cons k′ v′ xs*)

show *?case*
proof (*split if-split*, *intro conjI impI*)
assume *k1* = *k2*
with *Cons*(*5*) **have** *eq0*: *lookup-pair* (*update-by-pair* (*k2*, *u*) *xs*) *k2* = *u* **for** *u*
by (*simp del*: *update-by-pair.simps*)
show *lookup-pair* (*update-by-pair* (*k1*, *v*) ((*k'*, *v'*) # *xs*)) *k2* = *v*
proof (*simp add*: ‹*k1* = *k2*› *eq0 split*: *order.split*, *intro conjI impI*)
assume *comp k2 k'* = *Eq*
hence ¬ *lt k' k2* **by** (*simp add*: *eq*)
with *Cons*(*4*) **have** *k2* ∉ *fst* ‘ *set xs* **by** *auto*
thus *lookup-pair xs k2* = *0* **using** *Cons*(*2*) **by** (*simp add*: *lookup-pair-eq-0*)
qed
next
assume *k1* ≠ *k2*
with *Cons*(*5*) **have** *eq0*: *lookup-pair* (*update-by-pair* (*k1*, *u*) *xs*) *k2* = *lookup-pair xs k2* **for** *u*
by (*simp del*: *update-by-pair.simps*)
have ∗: *lookup-pair xs k2* = *0* **if** *lt k2 k'*
proof −
from ‹*lt k2 k'*› **have** ¬ *lt k' k2* **by** *auto*
with *Cons*(*4*) **have** *k2* ∉ *fst* ‘ *set xs* **by** *auto*
thus *lookup-pair xs k2* = *0* **using** *Cons*(*2*) **by** (*simp add*: *lookup-pair-eq-0*)
qed
show *lookup-pair* (*update-by-pair* (*k1*, *v*) ((*k'*, *v'*) # *xs*)) *k2* = *lookup-pair* ((*k'*, *v'*) # *xs*) *k2*
by (*simp add*: ‹*k1* ≠ *k2*› *eq0 split*: *order.split*,
auto intro: ∗ *simp*: ‹*k1* ≠ *k2*›[*symmetric*] *eq Gt-lt-conv Lt-lt-conv*)
qed
qed

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 (*rule lookup-pair-inj*, *rule oalist-inv-raw-update-by-pair*)
show *lookup-pair* (*update-by-pair* (*k*, *v*) *xs*) = *lookup-pair xs*
proof
fix *k0*
from *assms*(*2*) **show** *lookup-pair* (*update-by-pair* (*k*, *v*) *xs*) *k0* = *lookup-pair xs k0*
by (*auto simp*: *lookup-pair-update-by-pair*[*OF assms*(*1*)])
qed
qed *fact*+

lemma *set-update-by-pair*:
assumes *oalist-inv-raw xs* **and** *v* ≠ *0*
shows *set* (*update-by-pair* (*k*, *v*) *xs*) = *insert* (*k*, *v*) (*set xs* − *range* (*Pair k*)) (**is** *?A* = *?B*)
proof (*rule set-eqI*)
fix *x*::*'a* × *'b*

obtain *k′ v′* **where** *x*: *x* = (*k′*, *v′*) **by** *fastforce*
from *assms*(*1*) **have** *inv*: *oalist-inv-raw* (*update-by-pair* (*k*, *v*) *xs*)
by (*rule oalist-inv-raw-update-by-pair*)
show (*x* ∈ *?A*) ⟷ (*x* ∈ *?B*)
proof (*cases v′* = *0*)
case *True*
have *0* ∉ *snd* ‘ *set* (*update-by-pair* (*k*, *v*) *xs*) **and** *0* ∉ *snd* ‘ *set xs*
by (*rule oalist-inv-rawD1*, *fact*)+
hence (*k′*, *0*) ∉ *set* (*update-by-pair* (*k*, *v*) *xs*) **and** (*k′*, *0*) ∉ *set xs*
using *image-iff* **by** *fastforce*+
thus *?thesis* **by** (*simp add*: *x True assms*(*2*))
next
case *False*
show *?thesis*
by (*auto simp*: *x lookup-pair-eq-value*[*OF inv False*, *symmetric*] *lookup-pair-eq-value*[*OF assms*(*1*) *False*]
lookup-pair-update-by-pair[*OF assms*(*1*)])
qed
qed

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 (*rule set-eqI*)
fix *x*::*′a* × *′b*
obtain *k′ v′* **where** *x*: *x* = (*k′*, *v′*) **by** *fastforce*
from *assms*(*1*) **have** *inv*: *oalist-inv-raw* (*update-by-pair* (*k*, *0*) *xs*)
by (*rule oalist-inv-raw-update-by-pair*)
show (*x* ∈ *?A*) ⟷ (*x* ∈ *?B*)
proof (*cases v′* = *0*)
case *True*
have *0* ∉ *snd* ‘ *set* (*update-by-pair* (*k*, *0*) *xs*) **and** *0* ∉ *snd* ‘ *set xs*
by (*rule oalist-inv-rawD1*, *fact*)+
hence (*k′*, *0*) ∉ *set* (*update-by-pair* (*k*, *0*) *xs*) **and** (*k′*, *0*) ∉ *set xs*
using *image-iff* **by** *fastforce*+
thus *?thesis* **by** (*simp add*: *x True*)
next
case *False*
show *?thesis*
by (*auto simp*: *x lookup-pair-eq-value*[*OF inv False*, *symmetric*] *lookup-pair-eq-value*[*OF assms False*]
lookup-pair-update-by-pair[*OF assms*] *False*)
qed
qed

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-pair xs k*)) *xs*
using *assms* **by** (*induct xs rule*: *oalist-inv-raw-induct*, *simp*, *simp split*: *order.split*)

corollary *oalist-inv-raw-update-by-fun-pair*:
assumes *oalist-inv-raw xs*
shows *oalist-inv-raw* (*update-by-fun-pair k f xs*)
unfolding *update-by-fun-pair-eq-update-by-pair*[*OF assms*] **using** *assms* **by** (*rule oalist-inv-raw-update-by-pair*)

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*)
by (*simp add*: *update-by-fun-pair-eq-update-by-pair*[*OF assms*] *lookup-pair-update-by-pair*[*OF assms*])

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*)])
using *assms*
proof (*induct xs rule*: *oalist-inv-raw-induct*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons k′ v′ xs*)
from *Cons*(*6*) **have** *1*: *comp k* (*fst* (*last* ((*k′*, *v′*) # *xs*))) = *Gt* **by** *simp*
have *eq1*: *comp k k′* = *Gt*
proof (*cases xs* = [])
case *True*
with *1* **show** *?thesis* **by** *simp*
next
case *False*
have *lt k′* (*fst* (*last xs*)) **by** (*rule Cons*(*4*), *simp add*: *False*)
from *False 1* **have** *comp k* (*fst* (*last xs*)) = *Gt* **by** *simp*
moreover from ‹*lt k′* (*fst* (*last xs*))› **have** *comp* (*fst* (*last xs*)) *k′* = *Gt*
by (*simp add*: *Gt-lt-conv*)
ultimately show *?thesis*
by (*meson Gt-lt-conv less-trans Lt-lt-conv*[*symmetric*])
qed
have *eq2*: *update-by-fun-pair k f xs* = *xs* @ (*if f 0* = *0 then* [] *else* [(*k*, *f 0*)])
proof (*rule Cons*(*5*), *simp only*: *disj-commute*[*of xs* = []], *rule disjCI*)
assume *xs* ≠ []
with *1* **show** *comp k* (*fst* (*last xs*)) = *Gt* **by** *simp*
qed
show *?case* **by** (*simp split*: *order.split add*: *Let-def eq1 eq2*)
qed

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*
by (*simp add*: *update-by-fun-gr-pair-def Let-def update-by-fun-pair-gr*[*OF assms*] *split*: *order.split*)

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*)
unfolding *update-by-fun-pair-eq-update-by-pair*[*OF assms*] *update-by-fun-gr-pair-eq-update-by-fun-pair*[*OF assms*]
using *assms* **by** (*rule oalist-inv-raw-update-by-pair*)

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*)
by (*simp add*: *update-by-fun-pair-eq-update-by-pair*[*OF assms*]
update-by-fun-gr-pair-eq-update-by-fun-pair[*OF assms*] *lookup-pair-update-by-pair*[*OF assms*])

12.4.4 *map-pair*

lemma *map-pair-cong*:
assumes $\bigwedge kv.\ kv \in set\ xs \Longrightarrow f\ kv = g\ kv$
shows *map-pair f xs* = *map-pair g xs*
using *assms*
proof (*induct xs*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons x xs*)
have *f x* = *g x* **by** (*rule Cons*(*2*), *simp*)
moreover have *map-pair f xs* = *map-pair g xs* **by** (*rule Cons*(*1*), *rule Cons*(*2*), *simp*)
ultimately show *?case* **by** *simp*
qed

lemma *map-pair-subset*: *set* (*map-pair f xs*) ⊆ *f* ‘ *set xs*
proof (*induct xs rule*: *map-pair.induct*)
case (*1 f*)
show *?case* **by** *simp*
next
case (*2 f kv xs*)
obtain *k v* **where** *f*: *f kv* = (*k*, *v*) **by** *fastforce*
from *f*[*symmetric*] *HOL.refl* **have** ∗: *set* (*map-pair f xs*) ⊆ *f* ‘ *set xs*
by (*rule 2*)
show *?case* **by** (*simp add*: *f Let-def*, *intro conjI impI subset-insertI2* ∗)
qed

lemma *oalist-inv-raw-map-pair*:
  assumes *oalist-inv-raw xs*
    and $\bigwedge$*a b. comp* (*fst* (*f a*)) (*fst* (*f b*)) = *comp* (*fst a*) (*fst b*)
  shows *oalist-inv-raw* (*map-pair f xs*)
  using *assms*(*1*)
proof (*induct xs rule*: *oalist-inv-raw-induct*)
  case *Nil*
  from *oalist-inv-raw-Nil* **show** *?case* **by** *simp*
next
  case (*Cons k v xs*)
  obtain *k′ v′* **where** *f*: *f* (*k*, *v*) = (*k′*, *v′*) **by** *fastforce*
  show *?case*
  proof (*simp add*: *f Let-def Cons*(*5*), *rule*)
    assume *v′* ≠ *0*
    with *Cons*(*5*) **show** *oalist-inv-raw* ((*k′*, *v′*) # *map-pair f xs*)
    proof (*rule oalist-inv-raw-ConsI*)
      assume *map-pair f xs* ≠ []
      hence *hd* (*map-pair f xs*) ∈ *set* (*map-pair f xs*) **by** *simp*
      also have ... ⊆ *f* ‘ *set xs* **by** (*fact map-pair-subset*)
      finally obtain *x* **where** *x* ∈ *set xs* **and** *eq*: *hd* (*map-pair f xs*) = *f x* **..**
      from *this*(*1*) **have** *fst x* ∈ *fst* ‘ *set xs* **by** *fastforce*
      hence *lt k* (*fst x*) **by** (*rule Cons*(*4*))
      hence *lt* (*fst* (*f* (*k*, *v*))) (*fst* (*f x*))
        by (*simp add*: *Lt-lt-conv*[*symmetric*] *assms*(*2*))
      thus *lt k′* (*fst* (*hd* (*map-pair f xs*))) **by** (*simp add*: *f eq*)
    qed
  qed
qed

lemma *lookup-pair-map-pair*:
  assumes *oalist-inv-raw xs* **and** *snd* (*f* (*k*, *0*)) = *0*
    and $\bigwedge$*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*))
  using *assms*(*1*)
proof (*induct xs rule*: *oalist-inv-raw-induct*)
  case *Nil*
  show *?case* **by** (*simp add*: *assms*(*2*))
next
  case (*Cons k′ v′ xs*)
  obtain *k″ v″* **where** *f*: *f* (*k′*, *v′*) = (*k″*, *v″*) **by** *fastforce*
  have *comp* (*fst* (*f* (*k*, *v*))) *k″* = *comp* (*fst* (*f* (*k*, *v*))) (*fst* (*f* (*k′*, *v′*)))
    by (*simp add*: *f*)
  also have ... = *comp k k′*
    by (*simp add*: *assms*(*3*))
  finally have *eq0*: *comp* (*fst* (*f* (*k*, *v*))) *k″* = *comp k k′* **.**
  have ∗: *lookup-pair xs k* = *0* **if** *comp k k′* ≠ *Gt*
  proof (*simp add*: *lookup-pair-eq-0*[*OF Cons*(*2*)], *rule*)
    assume *k* ∈ *fst* ‘ *set xs*
    hence *lt k′ k* **by** (*rule Cons*(*4*))

hence *comp k k′ = Gt* **by** (*simp add*: *Gt-lt-conv*)
with ‹*comp k k′ ≠ Gt*› **show** *False* **..**
qed
show *?case*
proof (*simp add*: *assms*(*2*) *f Let-def eq0 Cons*(*5*) *split*: *order.split*, *intro conjI impI*)
assume *comp k k′ = Lt*
hence *comp k k′ ≠ Gt* **by** *simp*
hence *lookup-pair xs k = 0* **by** (*rule* ∗)
thus *snd* (*f* (*k*, *lookup-pair xs k*)) = *0* **by** (*simp add*: *assms*(*2*))
next
assume *v′′ = 0*
assume *comp k k′ = Eq*
hence *k = k′* **and** *comp k k′ ≠ Gt* **by** (*simp only*: *eq*, *simp*)
from *this*(*2*) **have** *lookup-pair xs k = 0* **by** (*rule* ∗)
hence *snd* (*f* (*k*, *lookup-pair xs k*)) = *0* **by** (*simp add*: *assms*(*2*))
also have ... = *snd* (*f* (*k*, *v′*)) **by** (*simp add*: ‹*k = k′*› *f* ‹*v′′ = 0*›)
finally show *snd* (*f* (*k*, *lookup-pair xs k*)) = *snd* (*f* (*k*, *v′*)) **.**
qed (*simp add*: *f eq*)
qed

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*))
using *assms*(*1*)
proof (*induct xs*)
case *Nil*
show *?case* **by** (*simp add*: *lookup-dflt-def assms*(*2*))
next
case (*Cons x xs*)
obtain *k′ v′* **where** *x*: *x* = (*k′*, *v′*) **by** *fastforce*
obtain *k′′ v′′* **where** *f*: *f* (*k′*, *v′*) = (*k′′*, *v′′*) **by** *fastforce*
from *Cons*(*2*) **have** *distinct* (*map fst xs*) **and** *k′ ∉ fst ‘ set xs* **by** (*simp-all add*: *x*)
from *this*(*1*) **have** *eq1*: *lookup-dflt* (*map-pair f xs*) (*fst* (*f* (*k*, *v*))) = *snd* (*f* (*k*, *lookup-dflt xs k*))
by (*rule Cons*(*1*))
have *eq2*: *lookup-dflt* ((*a*, *b*) # *ys*) *c* = (*if c = a then b else lookup-dflt ys c*)
for *a b c* **and** *ys*::(′*b* × ′*e*::*zero*) *list* **by** (*simp add*: *lookup-dflt-def map-of-Cons-code*)
from ‹*k′ ∉ fst ‘ set xs*› **have** *map-of xs k′ = None* **by** (*simp add*: *map-of-eq-None-iff*)
hence *eq3*: *lookup-dflt xs k′ = 0* **by** (*simp add*: *lookup-dflt-def*)
show *?case*
proof (*simp add*: *f Let-def x eq1 eq2 eq3*, *intro conjI impI*)
assume *k = k′*
hence *snd* (*f* (*k′*, *0*)) = *snd* (*f* (*k*, *0*)) **by** *simp*
also have ... = *0* **by** (*fact assms*(*2*))
finally show *snd* (*f* (*k′*, *0*)) = *0* **.**
next

    assume *fst* (*f* (k', *v*)) $\neq$ k''
    hence *fst* (*f* (k', *v*)) $\neq$ *fst* (*f* (k', v')) **by** (*simp add*: *f*)
    thus *snd* (*f* (k', *0*)) = v'' **by** (*simp add*: *assms*(*3*))
  next
    assume $k \neq k'$
    assume *fst* (*f* (*k*, *v*)) = k''
    also have ... = *fst* (*f* (k', v')) **by** (*simp add*: *f*)
    finally have $k = k'$ **by** (*simp add*: *assms*(*3*))
    with ‹$k \neq k'$› **show** v'' = *snd* (*f* (*k*, *lookup-dflt xs k*)) **..**
  qed
qed

lemma *distinct-map-pair*:
  assumes *distinct* (*map fst xs*) **and** ⋀*a b*. *fst* (*f a*) = *fst* (*f b*) ⟹ *fst a* = *fst b*
  shows *distinct* (*map fst* (*map-pair f xs*))
  using *assms*(*1*)
proof (*induct xs*)
  case *Nil*
  show *?case* **by** *simp*
next
  case (*Cons x xs*)
  obtain *k v* **where** *x*: *x* = (*k*, *v*) **by** *fastforce*
  obtain k' v' **where** *f*: *f* (*k*, *v*) = (k', v') **by** *fastforce*
  from *Cons*(*2*) **have** *distinct* (*map fst xs*) **and** *k* ∉ *fst* ‘ *set xs* **by** (*simp-all add*:
x)
  from *this*(*1*) **have** *1*: *distinct* (*map fst* (*map-pair f xs*)) **by** (*rule Cons*(*1*))
  show *?case*
  proof (*simp add*: *x f Let-def 1*, *intro impI notI*)
    assume $v' \neq 0$
    assume k' ∈ *fst* ‘ *set* (*map-pair f xs*)
    then obtain *y* **where** *y* ∈ *set* (*map-pair f xs*) **and** k' = *fst y* **..**
    from *this*(*1*) *map-pair-subset* **have** *y* ∈ *f* ‘ *set xs* **..**
    then obtain *z* **where** *z* ∈ *set xs* **and** *y* = *f z* **..**
    from *this*(*2*) **have** *fst* (*f z*) = k' **by** (*simp add*: ‹k' = *fst y*›)
    also have ... = *fst* (*f* (*k*, *v*)) **by** (*simp add*: *f*)
    finally have *fst z* = *fst* (*k*, *v*) **by** (*rule assms*(*2*))
    also have ... = *k* **by** *simp*
    finally have *k* ∈ *fst* ‘ *set xs* **using** ‹*z* ∈ *set xs*› **by** *blast*
    with ‹*k* ∉ *fst* ‘ *set xs*› **show** *False* **..**
  qed
qed

lemma *map-val-pair-cong*:
  assumes ⋀*k v*. (*k*, *v*) ∈ *set xs* ⟹ *f k v* = *g k v*
  shows *map-val-pair f xs* = *map-val-pair g xs*
proof (*rule map-pair-cong*)
  fix *kv*
  assume *kv* ∈ *set xs*
  moreover obtain *k v* **where** *kv* = (*k*, *v*) **by** *fastforce*

ultimately show (*case kv of* (*k*, *v*) ⇒ (*k*, *f k v*)) = (*case kv of* (*k*, *v*) ⇒ (*k*, *g k v*))
  by (*simp add*: *assms*)
qed

lemma *oalist-inv-raw-map-val-pair*:
  assumes *oalist-inv-raw xs*
  shows *oalist-inv-raw* (*map-val-pair f xs*)
  by (*rule oalist-inv-raw-map-pair*, *fact assms*, *auto*)

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 −
  let *?f* = λ(*k′*, *v′*). (*k′*, *f k′ v′*)
  have *lookup-pair* (*map-val-pair f xs*) *k* = *lookup-pair* (*map-val-pair f xs*) (*fst* (*?f* (*k*, *0*)))
    by *simp*
  also have ... = *snd* (*?f* (*k*, *local.lookup-pair xs k*))
    by (*rule lookup-pair-map-pair*, *fact assms*(*1*), *auto simp*: *assms*(*2*))
  also have ... = *f k* (*lookup-pair xs k*) **by** *simp*
  finally show *?thesis* **.**
qed

lemma *map-pair-id*:
  assumes *oalist-inv-raw xs*
  shows *map-pair id xs* = *xs*
  using *assms*
proof (*induct xs rule*: *oalist-inv-raw-induct*)
  case *Nil*
  show *?case* **by** *simp*
next
  case (*Cons k v xs′*)
  show *?case* **by** (*simp add*: *Let-def Cons*(*3*, *5*) *id-def*[*symmetric*])
qed

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 ⋀*zs*. *oalist-inv-raw zs* ⟹ *oalist-inv-raw* (*f zs*)
    and ⋀*zs*. *oalist-inv-raw zs* ⟹ *fst* ‘ *set* (*f zs*) ⊆ *fst* ‘ *set zs*
  shows *map2-val-compat f*
  unfolding *map2-val-compat-def* **using** *assms* **by** *blast*

lemma *map2-val-compatD1*:

assumes *map2-val-compat f* **and** *oalist-inv-raw zs*
shows *oalist-inv-raw* (*f zs*)
using *assms* **unfolding** *map2-val-compat-def* **by** *blast*

lemma *map2-val-compatD2*:
assumes *map2-val-compat f* **and** *oalist-inv-raw zs*
shows *fst ‘ set* (*f zs*) ⊆ *fst ‘ set zs*
using *assms* **unfolding** *map2-val-compat-def* **by** *blast*

lemma *map2-val-compat-Nil*:
assumes *map2-val-compat* (*f*::(′*a* × ′*b*::*zero*) *list* ⇒ (′*a* × ′*c*::*zero*) *list*)
shows *f* [] = []
proof −
from *assms oalist-inv-raw-Nil* **have** *fst ‘ set* (*f* []) ⊆ *fst ‘ set* ([]::(′*a* × ′*b*) *list*)
by (*rule map2-val-compatD2*)
thus *?thesis* **by** *simp*
qed

lemma *map2-val-compat-id*: *map2-val-compat id*
by (*rule map2-val-compatI*, *auto*)

lemma *map2-val-compat-map-val-pair*: *map2-val-compat* (*map-val-pair f*)
proof (*rule map2-val-compatI*, *erule oalist-inv-raw-map-val-pair*)
fix *zs*
from *map-pair-subset image-iff* **show** *fst ‘ set* (*map-val-pair f zs*) ⊆ *fst ‘ set zs*
by *fastforce*
qed

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*
using *assms*
proof (*induct f g h xs ys rule*: *map2-val-pair.induct*)
case (*1 f g h xs*)
show *?case* **by** (*simp*, *rule map2-val-compatD2*, *fact*+)
next
case (*2 f g h v va*)
show *?case* **by** (*simp del*: *set-simps*(*2*), *rule map2-val-compatD2*, *fact*+)
next
case (*3 f g h kx vx xs ky vy ys*)
from *3*(*4*) **have** *oalist-inv-raw xs* **by** (*rule oalist-inv-raw-ConsD1*)
from *3*(*5*) **have** *oalist-inv-raw ys* **by** (*rule oalist-inv-raw-ConsD1*)
show *?case*
proof (*simp split*: *order.split*, *intro conjI impI*)
assume *comp kx ky* = *Lt*
hence *fst ‘ set* (*map2-val-pair f g h xs* ((*ky*, *vy*) # *ys*)) ⊆ *fst ‘ set xs* ∪ *fst ‘ set* ((*ky*, *vy*) # *ys*)
using *HOL.refl* ‹*oalist-inv-raw xs*› *3*(*5*, *6*, *7*) **by** (*rule 3*(*2*))

thus *fst ‘ set* (*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*)
⊆ *insert ky* (*insert kx* (*fst ‘ set xs* ∪ *fst ‘ set ys*)) **by** (*auto simp*: *Let-def*)
next
assume *comp kx ky* = *Eq*
hence *fst ‘ set* (*map2-val-pair f g h xs ys*) ⊆ *fst ‘ set xs* ∪ *fst ‘ set ys*
using *HOL.refl* ‹*oalist-inv-raw xs*› ‹*oalist-inv-raw ys*› *3*(*6*, *7*) **by** (*rule 3*(*1*))
thus *fst ‘ set* (*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*)
⊆ *insert ky* (*insert kx* (*fst ‘ set xs* ∪ *fst ‘ set ys*)) **by** (*auto simp*: *Let-def*)
next
assume *comp kx ky* = *Gt*
hence *fst ‘ set* (*map2-val-pair f g h* ((*kx*, *vx*) # *xs*) *ys*) ⊆ *fst ‘ set* ((*kx*, *vx*) # *xs*) ∪ *fst ‘ set ys*
using *HOL.refl 3*(*4*) ‹*oalist-inv-raw ys*› *3*(*6*, *7*) **by** (*rule 3*(*3*))
thus *fst ‘ set* (*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*)
⊆ *insert ky* (*insert kx* (*fst ‘ set xs* ∪ *fst ‘ set ys*)) **by** (*auto simp*: *Let-def*)
qed
qed

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*)
using *assms*(*1*, *2*)
proof (*induct xs arbitrary*: *ys rule*: *oalist-inv-raw-induct*)
case *Nil*
show *?case*
proof (*cases ys*)
case *Nil*
show *?thesis* **by** (*simp add*: *Nil*, *rule map2-val-compatD1*, *fact assms*(*3*), *fact oalist-inv-raw-Nil*)
next
case (*Cons y ys′*)
show *?thesis* **by** (*simp add*: *Cons*, *rule map2-val-compatD1*, *fact assms*(*4*), *simp only*: *Cons*[*symmetric*], *fact Nil*)
qed
next
case ∗: (*Cons k v xs*)
from ∗(*6*) **show** *?case*
proof (*induct ys rule*: *oalist-inv-raw-induct*)
case *Nil*
show *?case* **by** (*simp*, *rule map2-val-compatD1*, *fact assms*(*3*), *fact* ∗(*1*))
next
case (*Cons k′ v′ ys*)
show *?case*
proof (*simp split*: *order.split*, *intro conjI impI*)
assume *comp k k′* = *Lt*

hence *0*: *lt k k′* **by** (*simp only*: *Lt-lt-conv*)
from *Cons*(*1*) **have** *1*: *oalist-inv-raw* (*map2-val-pair f g h xs* ((*k′*, *v′*) # *ys*)) **by** (*rule* ∗(*5*))
show *oalist-inv-raw* (*let v* = *f k v 0*; *aux* = *map2-val-pair f g h xs* ((*k′*, *v′*) # *ys*)
in if v = *0 then aux else* (*k*, *v*) # *aux*)
proof (*simp add*: *Let-def*, *intro conjI impI*)
assume *f k v 0* ≠ *0*
with *1* **show** *oalist-inv-raw* ((*k*, *f k v 0*) # *map2-val-pair f g h xs* ((*k′*, *v′*) # *ys*))
proof (*rule oalist-inv-raw-ConsI*)
define *k0* **where** *k0* = *fst* (*hd* (*local.map2-val-pair f g h xs* ((*k′*, *v′*) # *ys*)))
assume *map2-val-pair f g h xs* ((*k′*, *v′*) # *ys*) ≠ []
hence *k0* ∈ *fst* ‘ *set* (*map2-val-pair f g h xs* ((*k′*, *v′*) # *ys*)) **by** (*simp add*: *k0-def*)
also from ∗(*2*) *Cons*(*1*) *assms*(*3*, *4*) **have** ... ⊆ *fst* ‘ *set xs* ∪ *fst* ‘ *set* ((*k′*, *v′*) # *ys*)
by (*rule fst-map2-val-pair-subset*)
finally have *k0* ∈ *fst* ‘ *set xs* ∨ *k0* = *k′* ∨ *k0* ∈ *fst* ‘ *set ys* **by** *auto*
thus *lt k k0*
proof (*elim disjE*)
assume *k0* = *k′*
with *0* **show** *?thesis* **by** *simp*
next
assume *k0* ∈ *fst* ‘ *set ys*
hence *lt k′ k0* **by** (*rule Cons*(*4*))
with *0* **show** *?thesis* **by** (*rule less-trans*)
qed (*rule* ∗(*4*))
qed
qed (*rule 1*)
next
assume *comp k k′* = *Eq*
hence *k* = *k′* **by** (*simp only*: *eq*)
from *Cons*(*2*) **have** *1*: *oalist-inv-raw* (*map2-val-pair f g h xs ys*) **by** (*rule* ∗(*5*))
show *oalist-inv-raw* (*let v* = *f k v v′*; *aux* = *map2-val-pair f g h xs ys in if v* = *0 then aux else* (*k*, *v*) # *aux*)
proof (*simp add*: *Let-def*, *intro conjI impI*)
assume *f k v v′* ≠ *0*
with *1* **show** *oalist-inv-raw* ((*k*, *f k v v′*) # *map2-val-pair f g h xs ys*)
proof (*rule oalist-inv-raw-ConsI*)
define *k0* **where** *k0* = *fst* (*hd* (*map2-val-pair f g h xs ys*))
assume *map2-val-pair f g h xs ys* ≠ []
hence *k0* ∈ *fst* ‘ *set* (*map2-val-pair f g h xs ys*) **by** (*simp add*: *k0-def*)
also from ∗(*2*) *Cons*(*2*) *assms*(*3*, *4*) **have** ... ⊆ *fst* ‘ *set xs* ∪ *fst* ‘ *set ys*
by (*rule fst-map2-val-pair-subset*)
finally show *lt k k0*
proof

assume *k0* ∈ *fst* ' *set ys*
hence *lt k′ k0* **by** (*rule Cons*(*4*))
thus *?thesis* **by** (*simp only*: ‹*k* = *k′*›)
qed (*rule* ∗(*4*))
qed
qed (*rule 1*)
next
assume *comp k k′* = *Gt*
hence *0*: *lt k′ k* **by** (*simp only*: *Gt-lt-conv*)
show *oalist-inv-raw* (*let va* = *f k′ 0 v′*; *aux* = *map2-val-pair f g h* ((*k*, *v*) # *xs*) *ys*
in if va = *0 then aux else* (*k′*, *va*) # *aux*)
proof (*simp add*: *Let-def*, *intro conjI impI*)
assume *f k′ 0 v′* ≠ *0*
with *Cons*(*5*) **show** *oalist-inv-raw* ((*k′*, *f k′ 0 v′*) # *map2-val-pair f g h* ((*k*, *v*) # *xs*) *ys*)
proof (*rule oalist-inv-raw-ConsI*)
define *k0* **where** *k0* = *fst* (*hd* (*map2-val-pair f g h* ((*k*, *v*) # *xs*) *ys*))
assume *map2-val-pair f g h* ((*k*, *v*) # *xs*) *ys* ≠ []
hence *k0* ∈ *fst* ' *set* (*map2-val-pair f g h* ((*k*, *v*) # *xs*) *ys*) **by** (*simp add*: *k0-def*)
also from ∗(*1*) *Cons*(*2*) *assms*(*3*, *4*) **have** ... ⊆ *fst* ' *set* ((*k*, *v*) # *xs*) ∪ *fst* ' *set ys*
by (*rule fst-map2-val-pair-subset*)
finally have *k0* = *k* ∨ *k0* ∈ *fst* ' *set xs* ∨ *k0* ∈ *fst* ' *set ys* **by** *auto*
thus *lt k′ k0*
proof (*elim disjE*)
assume *k0* = *k*
with *0* **show** *?thesis* **by** *simp*
next
assume *k0* ∈ *fst* ' *set xs*
hence *lt k k0* **by** (*rule* ∗(*4*))
with *0* **show** *?thesis* **by** (*rule less-trans*)
qed (*rule Cons*(*4*))
qed
qed (*rule Cons*(*5*))
qed
qed
qed

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*)
using *assms*(*1*, *2*)

proof (*induct xs arbitrary*: *ys rule*: *oalist-inv-raw-induct*)
  case *Nil*
  show *?case*
  proof (*cases ys*)
    case *Nil*
    show *?thesis* **by** (*simp add*: *Nil map2-val-compat-Nil*[*OF assms*(*3*)] *assms*(*7*))
  next
    case (*Cons y ys'*)
    then obtain *k v ys'* **where** *ys*: *ys* = (*k*, *v*) # *ys'* **by** *fastforce*
    from *Nil* **have** *lookup-pair* (*h ys*) *k0* = *lookup-pair* (*map-val-pair* (λ*k*. *f k 0*) *ys*) *k0*
      by (*simp only*: *assms*(*6*))
    also have ... = *f k0 0* (*lookup-pair ys k0*) **by** (*rule lookup-pair-map-val-pair*, *fact Nil*, *fact assms*(*7*))
    finally have *lookup-pair* (*h* ((*k*, *v*) # *ys'*)) *k0* = *f k0 0* (*lookup-pair* ((*k*, *v*) # *ys'*) *k0*)
      by (*simp only*: *ys*)
    thus *?thesis* **by** (*simp add*: *ys*)
  qed
next
  case ∗: (*Cons k v xs*)
  from ∗(*6*) **show** *?case*
  proof (*induct ys rule*: *oalist-inv-raw-induct*)
    case *Nil*
    from ∗(*1*) **have** *lookup-pair* (*g* ((*k*, *v*) # *xs*)) *k0* = *lookup-pair* (*map-val-pair* (λ*k v*. *f k v 0*) ((*k*, *v*) # *xs*)) *k0*
      by (*simp only*: *assms*(*5*))
    also have ... = *f k0* (*lookup-pair* ((*k*, *v*) # *xs*) *k0*) *0*
      by (*rule lookup-pair-map-val-pair*, *fact* ∗(*1*), *fact assms*(*7*))
    finally show *?case* **by** *simp*
  next
    case (*Cons k' v' ys*)
    show *?case*
    proof (*cases comp k0 k* = *Lt* ∧ *comp k0 k'* = *Lt*)
      case *True*
      hence *1*: *comp k0 k* = *Lt* **and** *2*: *comp k0 k'* = *Lt* **by** *simp-all*
      hence *eq*: *f k0* (*lookup-pair* ((*k*, *v*) # *xs*) *k0*) (*lookup-pair* ((*k'*, *v'*) # *ys*) *k0*) = *0*
        by (*simp add*: *assms*(*7*))
      from ∗(*1*) *Cons*(*1*) *assms*(*3*, *4*) **have** *inv*: *oalist-inv-raw* (*map2-val-pair f g h* ((*k*, *v*) # *xs*) ((*k'*, *v'*) # *ys*))
        by (*rule oalist-inv-raw-map2-val-pair*)
      show *?thesis*
      proof (*simp only*: *eq lookup-pair-eq-0*[*OF inv*], *rule*)
        assume *k0* ∈ *fst* ‘ *set* (*local.map2-val-pair f g h* ((*k*, *v*) # *xs*) ((*k'*, *v'*) # *ys*))
        also from ∗(*1*) *Cons*(*1*) *assms*(*3*, *4*) **have** ... ⊆ *fst* ‘ *set* ((*k*, *v*) # *xs*) ∪ *fst* ‘ *set* ((*k'*, *v'*) # *ys*)
          by (*rule fst-map2-val-pair-subset*)

finally have *k0* ∈ *fst* ' *set xs* ∨ *k0* ∈ *fst* ' *set ys* **using** *1 2* **by** *auto*
thus *False*
proof
assume *k0* ∈ *fst* ' *set xs*
hence *lt k k0* **by** (*rule* ∗(*4*))
with *1* **show** *?thesis* **by** (*simp add*: *Lt-lt-conv*)
next
assume *k0* ∈ *fst* ' *set ys*
hence *lt k′ k0* **by** (*rule Cons*(*4*))
with *2* **show** *?thesis* **by** (*simp add*: *Lt-lt-conv*)
qed
qed
next
case *False*
show *?thesis*
proof (*simp split*: *order.split del*: *lookup-pair.simps*, *intro conjI impI*)
assume *comp k k′* = *Lt*
with *False* **have** *comp k0 k* ≠ *Lt* **by** (*auto simp*: *Lt-lt-conv*)
show *lookup-pair* (*let v* = *f k v 0*; *aux* = *map2-val-pair f g h xs* ((*k′*, *v′*) # *ys*)
in if v = *0 then aux else* (*k*, *v*) # *aux*) *k0* =
f k0 (*lookup-pair* ((*k*, *v*) # *xs*) *k0*) (*lookup-pair* ((*k′*, *v′*) # *ys*) *k0*)
proof (*cases comp k0 k*)
case *Lt*
with ‹*comp k0 k* ≠ *Lt*› **show** *?thesis* **..**
next
case *Eq*
hence *k0* = *k* **by** (*simp only*: *eq*)
with ‹*comp k k′* = *Lt*› **have** *comp k0 k′* = *Lt* **by** *simp*
hence *eq1*: *lookup-pair* ((*k′*, *v′*) # *ys*) *k* = *0* **by** (*simp add*: ‹*k0* = *k*›)
have *eq2*: *lookup-pair* ((*k*, *v*) # *xs*) *k* = *v* **by** *simp*
show *?thesis*
proof (*simp add*: *Let-def eq1 eq2* ‹*k0* = *k*› *del*: *lookup-pair.simps*, *intro conjI impI*)
from ∗(*2*) *Cons*(*1*) *assms*(*3*, *4*) **have** *inv*: *oalist-inv-raw* (*map2-val-pair f g h xs* ((*k′*, *v′*) # *ys*))
by (*rule oalist-inv-raw-map2-val-pair*)
show *lookup-pair* (*map2-val-pair f g h xs* ((*k′*, *v′*) # *ys*)) *k* = *0*
proof (*simp only*: *lookup-pair-eq-0*[*OF inv*], *rule*)
assume *k* ∈ *fst* ' *set* (*local.map2-val-pair f g h xs* ((*k′*, *v′*) # *ys*))
also from ∗(*2*) *Cons*(*1*) *assms*(*3*, *4*) **have** ... ⊆ *fst* ' *set xs* ∪ *fst* ' *set* ((*k′*, *v′*) # *ys*)
by (*rule fst-map2-val-pair-subset*)
finally have *k* ∈ *fst* ' *set xs* ∨ *k* ∈ *fst* ' *set ys* **using** ‹*comp k k′* = *Lt*›
by *auto*
thus *False*
proof
assume *k* ∈ *fst* ' *set xs*
hence *lt k k* **by** (*rule* ∗(*4*))

thus *?thesis* **by** *simp*
next
assume $k \in fst$ ' *set ys*
hence *lt k' k* **by** (*rule Cons*(*4*))
with ‹*comp k k' = Lt*› **show** *?thesis* **by** (*simp add*: *Lt-lt-conv*)
qed
qed
qed *simp*
next
case *Gt*
hence *eq1*: *lookup-pair* ((*k*, *v*) # *xs*) *k0* = *lookup-pair xs k0*
and *eq2*: *lookup-pair* ((*k*, *f k v 0*) # *map2-val-pair f g h xs* ((*k'*, *v'*) # *ys*)) *k0* =
lookup-pair (*map2-val-pair f g h xs* ((*k'*, *v'*) # *ys*)) *k0* **by** *simp-all*
show *?thesis*
by (*simp add*: *Let-def eq1 eq2 del*: *lookup-pair.simps*, *rule* ∗(*5*), *fact Cons*(*1*))
qed
next
assume *comp k k' = Eq*
hence *k = k'* **by** (*simp only*: *eq*)
with *False* **have** *comp k0 k'* ≠ *Lt* **by** (*auto simp*: *Lt-lt-conv*)
show *lookup-pair* (*let v = f k v v'*; *aux = map2-val-pair f g h xs ys in*
if v = 0 then aux else (*k*, *v*) # *aux*) *k0* =
f k0 (*lookup-pair* ((*k*, *v*) # *xs*) *k0*) (*lookup-pair* ((*k'*, *v'*) # *ys*) *k0*)
proof (*cases comp k0 k'*)
case *Lt*
with ‹*comp k0 k'* ≠ *Lt*› **show** *?thesis* **..**
next
case *Eq*
hence *k0 = k'* **by** (*simp only*: *eq*)
show *?thesis*
proof (*simp add*: *Let-def* ‹*k = k'*› ‹*k0 = k'*›, *intro impI*)
from ∗(*2*) *Cons*(*2*) *assms*(*3*, *4*) **have** *inv*: *oalist-inv-raw* (*map2-val-pair f g h xs ys*)
by (*rule oalist-inv-raw-map2-val-pair*)
show *lookup-pair* (*map2-val-pair f g h xs ys*) *k'* = *0*
proof (*simp only*: *lookup-pair-eq-0*[*OF inv*], *rule*)
assume $k' \in fst$ ' *set* (*map2-val-pair f g h xs ys*)
also from ∗(*2*) *Cons*(*2*) *assms*(*3*, *4*) **have** ... ⊆ *fst* ' *set xs* ∪ *fst* ' *set ys*
by (*rule fst-map2-val-pair-subset*)
finally show *False*
proof
assume $k' \in fst$ ' *set ys*
hence *lt k' k'* **by** (*rule Cons*(*4*))
thus *?thesis* **by** *simp*
next
assume $k' \in fst$ ' *set xs*

hence *lt k k'* **by** (*rule* ∗(*4*))
thus *?thesis* **by** (*simp add*: ‹*k* = *k'*›)
qed
qed
qed
next
case *Gt*
hence *eq1*: *lookup-pair* ((*k*, *v*) # *xs*) *k0* = *lookup-pair xs k0*
and *eq2*: *lookup-pair* ((*k'*, *v'*) # *ys*) *k0* = *lookup-pair ys k0*
and *eq3*: *lookup-pair* ((*k*, *f k v v'*) # *map2-val-pair f g h xs ys*) *k0* = *lookup-pair* (*map2-val-pair f g h xs ys*) *k0* **by** (*simp-all add*: ‹*k* = *k'*›)
show *?thesis* **by** (*simp add*: *Let-def eq1 eq2 eq3 del*: *lookup-pair.simps*, *rule* ∗(*5*), *fact Cons*(*2*))
qed
next
assume *comp k k'* = *Gt*
hence *comp k' k* = *Lt* **by** (*simp only*: *Gt-lt-conv Lt-lt-conv*)
with *False* **have** *comp k0 k'* ≠ *Lt* **by** (*auto simp*: *Lt-lt-conv*)
show *lookup-pair* (*let va* = *f k' 0 v'*; *aux* = *map2-val-pair f g h* ((*k*, *v*) # *xs*) *ys*
in if va = *0 then aux else* (*k'*, *va*) # *aux*) *k0* =
f k0 (*lookup-pair* ((*k*, *v*) # *xs*) *k0*) (*lookup-pair* ((*k'*, *v'*) # *ys*) *k0*)
proof (*cases comp k0 k'*)
case *Lt*
with ‹*comp k0 k'* ≠ *Lt*› **show** *?thesis* **..**
next
case *Eq*
hence *k0* = *k'* **by** (*simp only*: *eq*)
with ‹*comp k' k* = *Lt*› **have** *comp k0 k* = *Lt* **by** *simp*
hence *eq1*: *lookup-pair* ((*k*, *v*) # *xs*) *k'* = *0* **by** (*simp add*: ‹*k0* = *k'*›)
have *eq2*: *lookup-pair* ((*k'*, *v'*) # *ys*) *k'* = *v'* **by** *simp*
show *?thesis*
proof (*simp add*: *Let-def eq1 eq2* ‹*k0* = *k'*› *del*: *lookup-pair.simps*, *intro conjI impI*)
from ∗(*1*) *Cons*(*2*) *assms*(*3*, *4*) **have** *inv*: *oalist-inv-raw* (*map2-val-pair f g h* ((*k*, *v*) # *xs*) *ys*)
by (*rule oalist-inv-raw-map2-val-pair*)
show *lookup-pair* (*map2-val-pair f g h* ((*k*, *v*) # *xs*) *ys*) *k'* = *0*
proof (*simp only*: *lookup-pair-eq-0*[*OF inv*], *rule*)
assume *k'* ∈ *fst* ‘ *set* (*map2-val-pair f g h* ((*k*, *v*) # *xs*) *ys*)
also from ∗(*1*) *Cons*(*2*) *assms*(*3*, *4*) **have** ... ⊆ *fst* ‘ *set* ((*k*, *v*) # *xs*) ∪ *fst* ‘ *set ys*
by (*rule fst-map2-val-pair-subset*)
finally have *k'* ∈ *fst* ‘ *set xs* ∨ *k'* ∈ *fst* ‘ *set ys* **using** ‹*comp k' k* = *Lt*›
by *auto*
thus *False*
proof
assume *k'* ∈ *fst* ‘ *set ys*

hence *lt k′ k′* **by** (*rule Cons*(*4*))
thus *?thesis* **by** *simp*
next
assume *k′* ∈ *fst ‘ set xs*
hence *lt k k′* **by** (*rule* ∗(*4*))
with ‹*comp k′ k* = *Lt*› **show** *?thesis* **by** (*simp add*: *Lt-lt-conv*)
qed
qed
qed *simp*
next
case *Gt*
hence *eq1*: *lookup-pair* ((*k′*, *v′*) # *ys*) *k0* = *lookup-pair ys k0*
and *eq2*: *lookup-pair* ((*k′*, *f k′ 0 v′*) # *map2-val-pair f g h* ((*k*, *v*) # *xs*) *ys*) *k0* =
lookup-pair (*map2-val-pair f g h* ((*k*, *v*) # *xs*) *ys*) *k0* **by** *simp-all*
show *?thesis* **by** (*simp add*: *Let-def eq1 eq2 del*: *lookup-pair.simps*, *rule Cons*(*5*))
qed
qed
qed
qed
qed

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*
using *assms*(*1*)
proof (*induct xs rule*: *oalist-inv-raw-induct*)
case *Nil*
show *?case* **by** (*simp add*: *Let-def assms*(*4*))
next
case (*Cons k′ v′ xs*)
show *?case*
proof (*cases comp k′ k*)
case *Lt*
hence *gr*: *comp k k′* = *Gt* **by** (*simp only*: *Gt-lt-conv Lt-lt-conv*)
show *?thesis* **by** (*simp add*: *Lt gr Let-def assms*(*2*) *Cons*(*3*, *5*))
next
case *Eq*
hence *eq1*: *comp k k′* = *Eq* **and** *eq2*: *k* = *k′* **by** (*simp-all only*: *eq*)
show *?thesis* **by** (*simp add*: *Eq eq1 eq2 Let-def assms*(*3*)[*OF Cons*(*2*)])
next
case *Gt*
hence *less*: *comp k k′* = *Lt* **by** (*simp only*: *Gt-lt-conv Lt-lt-conv*)
show *?thesis* **by** (*simp add*: *Gt less Let-def assms*(*3*)[*OF Cons*(*1*)])
qed
qed

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*
using *assms*
proof (*induct xs arbitrary*: *ys rule*: *oalist-inv-raw-induct*)
case *Nil*
thus *?case*
proof (*induct ys rule*: *oalist-inv-raw-induct*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons k v ys*)
show *?case*
proof (*simp add*: *Let-def*, *intro conjI impI*, *rule Cons*(*5*))
fix *k0*
assume *k0* ∈ *fst* ‘ *set* [] ∪ *fst* ‘ *set ys*
hence *k0* ∈ *fst* ‘ *set ys* **by** *simp*
hence *lt k k0* **by** (*rule Cons*(*4*))
hence *f k0* (*lookup-pair* [] *k0*) (*lookup-pair ys k0*) = *f k0* (*lookup-pair* [] *k0*) (*lookup-pair* ((*k*, *v*) # *ys*) *k0*)
by (*auto simp add*: *lookup-pair-Cons*[*OF Cons*(*1*)] *simp del*: *lookup-pair.simps*)
also have ... = *Some Eq* **by** (*rule Cons*(*6*), *simp add*: ‹*k0* ∈ *fst* ‘ *set ys*›)
finally show *f k0* (*lookup-pair* [] *k0*) (*lookup-pair ys k0*) = *Some Eq* **.**
next
have *f k 0 v* = *f k* (*lookup-pair* [] *k*) (*lookup-pair* ((*k*, *v*) # *ys*) *k*) **by** *simp*
also have ... = *Some Eq* **by** (*rule Cons*(*6*), *simp*)
finally show *f k 0 v* = *Some Eq* **.**
qed
qed
next
case ∗: (*Cons k v xs*)
from ∗(*6*, *7*) **show** *?case*
proof (*induct ys rule*: *oalist-inv-raw-induct*)
case *Nil*
show *?case*
proof (*simp add*: *Let-def*, *intro conjI impI*, *rule* ∗(*5*), *rule oalist-inv-raw-Nil*)
fix *k0*
assume *k0* ∈ *fst* ‘ *set xs* ∪ *fst* ‘ *set* []
hence *k0* ∈ *fst* ‘ *set xs* **by** *simp*
hence *lt k k0* **by** (*rule* ∗(*4*))
hence *f k0* (*lookup-pair xs k0*) (*lookup-pair* [] *k0*) = *f k0* (*lookup-pair* ((*k*, *v*) # *xs*) *k0*) (*lookup-pair* [] *k0*)
by (*auto simp add*: *lookup-pair-Cons*[*OF* ∗(*1*)] *simp del*: *lookup-pair.simps*)
also have ... = *Some Eq* **by** (*rule Nil*, *simp add*: ‹*k0* ∈ *fst* ‘ *set xs*›)
finally show *f k0* (*lookup-pair xs k0*) (*lookup-pair* [] *k0*) = *Some Eq* **.**
next

have *f k v 0 = f k (lookup-pair ((k, v) # xs) k) (lookup-pair [] k)* **by** *simp*
also have ... = *Some Eq* **by** (*rule Nil*, *simp*)
finally show *f k v 0 = Some Eq* **.**
qed
next
case (*Cons k′ v′ ys*)
show *?case*
proof (*simp split*: *order.split*, *intro conjI impI*)
assume *comp k k′ = Lt*
show (*let aux = f k v 0 in if aux = Some Eq then lex-ord-pair f xs ((k′, v′) # ys) else aux*) = *Some Eq*
proof (*simp add*: *Let-def*, *intro conjI impI*, *rule* ∗(*5*), *rule Cons*(*1*))
fix *k0*
assume *k0-in*: *k0 ∈ fst ‘ set xs ∪ fst ‘ set ((k′, v′) # ys)*
hence *k0 ∈ fst ‘ set xs ∨ k0 = k′ ∨ k0 ∈ fst ‘ set ys* **by** *auto*
hence *k0 ≠ k*
proof (*elim disjE*)
assume *k0 ∈ fst ‘ set xs*
hence *lt k k0* **by** (*rule* ∗(*4*))
thus *?thesis* **by** *simp*
next
assume *k0 = k′*
with ‹*comp k k′ = Lt*› **show** *?thesis* **by** *auto*
next
assume *k0 ∈ fst ‘ set ys*
hence *lt k′ k0* **by** (*rule Cons*(*4*))
with ‹*comp k k′ = Lt*› **show** *?thesis* **by** (*simp add*: *Lt-lt-conv*)
qed
hence *f k0 (lookup-pair xs k0) (lookup-pair ((k′, v′) # ys) k0) =*
f k0 (lookup-pair ((k, v) # xs) k0) (lookup-pair ((k′, v′) # ys) k0)
by (*auto simp add*: *lookup-pair-Cons*[*OF* ∗(*1*)] *simp del*: *lookup-pair.simps*)
also have ... = *Some Eq* **by** (*rule Cons*(*6*), *rule rev-subsetD*, *fact k0-in*, *auto*)
finally show *f k0 (lookup-pair xs k0) (lookup-pair ((k′, v′) # ys) k0) = Some Eq* **.**
next
have *f k v 0 = f k (lookup-pair ((k, v) # xs) k) (lookup-pair ((k′, v′) # ys) k)*
by (*simp add*: ‹*comp k k′ = Lt*›)
also have ... = *Some Eq* **by** (*rule Cons*(*6*), *simp*)
finally show *f k v 0 = Some Eq* **.**
qed
next
assume *comp k k′ = Eq*
hence *k = k′* **by** (*simp only*: *eq*)
show (*let aux = f k v v′ in if aux = Some Eq then lex-ord-pair f xs ys else aux*) = *Some Eq*
proof (*simp add*: *Let-def*, *intro conjI impI*, *rule* ∗(*5*), *rule Cons*(*2*))
fix *k0*

assume *k0-in*: *k0* ∈ *fst* ' *set xs* ∪ *fst* ' *set ys*
hence *k0* ≠ *k'*
proof
assume *k0* ∈ *fst* ' *set xs*
hence *lt k k0* **by** (*rule* ∗(*4*))
thus *?thesis* **by** (*simp add*: ‹*k* = *k'*›)
next
assume *k0* ∈ *fst* ' *set ys*
hence *lt k' k0* **by** (*rule Cons*(*4*))
thus *?thesis* **by** *simp*
qed
hence *f k0* (*lookup-pair xs k0*) (*lookup-pair ys k0*) =
f k0 (*lookup-pair* ((*k*, *v*) # *xs*) *k0*) (*lookup-pair* ((*k'*, *v'*) # *ys*) *k0*)
by (*simp add*: *lookup-pair-Cons*[*OF* ∗(*1*)] *lookup-pair-Cons*[*OF Cons*(*1*)] *del*: *lookup-pair.simps*,
auto simp: ‹*k* = *k'*›)
also have ... = *Some Eq* **by** (*rule Cons*(*6*), *rule rev-subsetD*, *fact k0-in*, *auto*)
finally show *f k0* (*lookup-pair xs k0*) (*lookup-pair ys k0*) = *Some Eq* **.**
next
have *f k v v'* = *f k* (*lookup-pair* ((*k*, *v*) # *xs*) *k*) (*lookup-pair* ((*k'*, *v'*) # *ys*) *k*)
by (*simp add*: ‹*k* = *k'*›)
also have ... = *Some Eq* **by** (*rule Cons*(*6*), *simp*)
finally show *f k v v'* = *Some Eq* **.**
qed
next
assume *comp k k'* = *Gt*
hence *comp k' k* = *Lt* **by** (*simp only*: *Gt-lt-conv Lt-lt-conv*)
show (*let aux* = *f k' 0 v' in if aux* = *Some Eq then lex-ord-pair f* ((*k*, *v*) # *xs*) *ys else aux*) = *Some Eq*
proof (*simp add*: *Let-def*, *intro conjI impI*, *rule Cons*(*5*))
fix *k0*
assume *k0-in*: *k0* ∈ *fst* ' *set* ((*k*, *v*) # *xs*) ∪ *fst* ' *set ys*
hence *k0* ∈ *fst* ' *set xs* ∨ *k0* = *k* ∨ *k0* ∈ *fst* ' *set ys* **by** *auto*
hence *k0* ≠ *k'*
proof (*elim disjE*)
assume *k0* ∈ *fst* ' *set xs*
hence *lt k k0* **by** (*rule* ∗(*4*))
with ‹*comp k' k* = *Lt*› **show** *?thesis* **by** (*simp add*: *Lt-lt-conv*)
next
assume *k0* = *k*
with ‹*comp k' k* = *Lt*› **show** *?thesis* **by** *auto*
next
assume *k0* ∈ *fst* ' *set ys*
hence *lt k' k0* **by** (*rule Cons*(*4*))
thus *?thesis* **by** *simp*
qed
hence *f k0* (*lookup-pair* ((*k*, *v*) # *xs*) *k0*) (*lookup-pair ys k0*) =

f k0 (*lookup-pair* ((*k*, *v*) # *xs*) *k0*) (*lookup-pair* ((*k'*, *v'*) # *ys*) *k0*)
by (*auto simp add*: *lookup-pair-Cons*[*OF Cons*(*1*)] *simp del*: *lookup-pair.simps*)
also have ... = *Some Eq* **by** (*rule Cons*(*6*), *rule rev-subsetD*, *fact k0-in*, *auto*)
finally show *f k0* (*lookup-pair* ((*k*, *v*) # *xs*) *k0*) (*lookup-pair ys k0*) = *Some Eq* **.**
next
have *f k' 0 v'* = *f k'* (*lookup-pair* ((*k*, *v*) # *xs*) *k'*) (*lookup-pair* ((*k'*, *v'*) # *ys*) *k'*)
by (*simp add*: ‹*comp k' k* = *Lt*›)
also have ... = *Some Eq* **by** (*rule Cons*(*6*), *simp*)
finally show *f k' 0 v'* = *Some Eq* **.**
qed
qed
qed
qed

lemma *lex-ord-pair-valI*:
assumes *oalist-inv-raw xs* **and** *oalist-inv-raw ys* **and** *aux* ≠ *Some Eq*
assumes *k* ∈ *fst* ‘ *set xs* ∪ *fst* ‘ *set ys* **and** *aux* = *f k* (*lookup-pair xs k*) (*lookup-pair ys k*)
and ⋀*k'*. *k'* ∈ *fst* ‘ *set xs* ∪ *fst* ‘ *set ys* ⟹ *lt k' k* ⟹
f k' (*lookup-pair xs k'*) (*lookup-pair ys k'*) = *Some Eq*
shows *lex-ord-pair f xs ys* = *aux*
using *assms*(*1*, *2*, *4*, *5*, *6*)
proof (*induct xs arbitrary*: *ys rule*: *oalist-inv-raw-induct*)
case *Nil*
thus *?case*
proof (*induct ys rule*: *oalist-inv-raw-induct*)
case *Nil*
from *Nil*(*1*) **show** *?case* **by** *simp*
next
case (*Cons k' v' ys*)
from *Cons*(*6*) **have** *k* = *k'* ∨ *k* ∈ *fst* ‘ *set ys* **by** *simp*
thus *?case*
proof
assume *k* = *k'*
with *Cons*(*7*) **have** *f k' 0 v'* = *aux* **by** *simp*
thus *?thesis* **by** (*simp add*: *Let-def* ‹*k* = *k'*› *assms*(*3*))
next
assume *k* ∈ *fst* ‘*set ys*
hence *lt k' k* **by** (*rule Cons*(*4*))
hence *comp k k'* = *Gt* **by** (*simp add*: *Gt-lt-conv*)
hence *eq1*: *lookup-pair* ((*k'*, *v'*) # *ys*) *k* = *lookup-pair ys k* **by** *simp*
have *f k'* (*lookup-pair* [] *k'*) (*lookup-pair* ((*k'*, *v'*) # *ys*) *k'*) = *Some Eq*
by (*rule Cons*(*8*), *simp*, *fact*)
hence *eq2*: *f k' 0 v'* = *Some Eq* **by** *simp*
show *?thesis*
proof (*simp add*: *Let-def eq2*, *rule Cons*(*5*))

```
        from ‹k ∈ fst ‘set ys› show k ∈ fst ‘ set [] ∪ fst ‘ set ys by simp
      next
       show aux = f k (lookup-pair [] k) (lookup-pair ys k) by (simp only: Cons(7)
eq1)
      next
        fix k0
        assume lt k0 k
        assume k0 ∈ fst ‘ set [] ∪ fst ‘ set ys
        hence k0-in: k0 ∈ fst ‘ set ys by simp
        hence lt k' k0 by (rule Cons(4))
        hence comp k0 k' = Gt by (simp add: Gt-lt-conv)
        hence f k0 (lookup-pair [] k0) (lookup-pair ys k0) =
                f k0 (lookup-pair [] k0) (lookup-pair ((k', v') # ys) k0) by simp
        also have ... = Some Eq by (rule Cons(8), simp add: k0-in, fact)
        finally show f k0 (lookup-pair [] k0) (lookup-pair ys k0) = Some Eq .
      qed
    qed
  qed
next
  case *: (Cons k' v' xs)
  from *(6, 7, 8, 9) show ?case
  proof (induct ys rule: oalist-inv-raw-induct)
    case Nil
    from Nil(1) have k = k' ∨ k ∈ fst ‘ set xs by simp
    thus ?case
    proof
      assume k = k'
      with Nil(2) have f k' v' 0 = aux by simp
      thus ?thesis by (simp add: Let-def ‹k = k'› assms(3))
    next
      assume k ∈ fst ‘ set xs
      hence lt k' k by (rule *(4))
      hence comp k k' = Gt by (simp add: Gt-lt-conv)
      hence eq1: lookup-pair ((k', v') # xs) k = lookup-pair xs k by simp
      have f k' (lookup-pair ((k', v') # xs) k') (lookup-pair [] k') = Some Eq
        by (rule Nil(3), simp, fact)
      hence eq2: f k' v' 0 = Some Eq by simp
      show ?thesis
      proof (simp add: Let-def eq2, rule *(5), fact oalist-inv-raw-Nil)
        from ‹k ∈ fst ‘set xs› show k ∈ fst ‘ set xs ∪ fst ‘ set [] by simp
      next
        show aux = f k (lookup-pair xs k) (lookup-pair [] k) by (simp only: Nil(2)
eq1)
      next
        fix k0
        assume lt k0 k
        assume k0 ∈ fst ‘ set xs ∪ fst ‘ set []
        hence k0-in: k0 ∈ fst ‘ set xs by simp
        hence lt k' k0 by (rule *(4))
```

hence *comp k0 k′ = Gt* **by** (*simp add*: *Gt-lt-conv*)
hence *f k0* (*lookup-pair xs k0*) (*lookup-pair* [] *k0*) =
f k0 (*lookup-pair* ((*k′*, *v′*) # *xs*) *k0*) (*lookup-pair* [] *k0*) **by** *simp*
also have ... = *Some Eq* **by** (*rule Nil*(*3*), *simp add*: *k0-in*, *fact*)
finally show *f k0* (*lookup-pair xs k0*) (*lookup-pair* [] *k0*) = *Some Eq* **.**
qed
qed
next
case (*Cons k′′ v′′ ys*)

have *0*: *thesis* **if** *1*: *lt k k′* **and** *2*: *lt k k′′* **for** *thesis*
proof −
from *1* **have** *k* ≠ *k′* **by** *simp*
moreover from *2* **have** *k* ≠ *k′′* **by** *simp*
ultimately have *k* ∈ *fst ‘ set xs* ∨ *k* ∈ *fst ‘ set ys* **using** *Cons*(*6*) **by** *simp*
thus *?thesis*
proof
assume *k* ∈ *fst ‘ set xs*
hence *lt k′ k* **by** (*rule* ∗(*4*))
with *1* **show** *?thesis* **by** *simp*
next
assume *k* ∈ *fst ‘ set ys*
hence *lt k′′ k* **by** (*rule Cons*(*4*))
with *2* **show** *?thesis* **by** *simp*
qed
qed

show *?case*
proof (*simp split*: *order.split*, *intro conjI impI*)
assume *Lt*: *comp k′ k′′ = Lt*
show (*let aux = f k′ v′ 0 in if aux = Some Eq then lex-ord-pair f xs* ((*k′′*, *v′′*) # *ys*) *else aux*) = *aux*
proof (*simp add*: *Let-def split*: *order.split*, *intro conjI impI*)
assume *f k′ v′ 0 = Some Eq*
have *k* ≠ *k′*
proof
assume *k = k′*
have *aux = f k v′ 0* **by** (*simp add*: *Cons*(*7*) ‹*k = k′*› *Lt*)
with ‹*f k′ v′ 0 = Some Eq*› *assms*(*3*) **show** *False* **by** (*simp add*: ‹*k = k′*›)
qed
from *Cons*(*1*) **show** *lex-ord-pair f xs* ((*k′′*, *v′′*) # *ys*) = *aux*
proof (*rule* ∗(*5*))
from *Cons*(*6*) ‹*k* ≠ *k′*› **show** *k* ∈ *fst ‘ set xs* ∪ *fst ‘ set* ((*k′′*, *v′′*) # *ys*) **by** *simp*
next
show *aux = f k* (*lookup-pair xs k*) (*lookup-pair* ((*k′′*, *v′′*) # *ys*) *k*)
by (*simp add*: *Cons*(*7*) *lookup-pair-Cons*[*OF* ∗(*1*)] ‹*k* ≠ *k′*›[*symmetric*] *del*: *lookup-pair.simps*)
next

fix *k0*
assume *lt k0 k*
assume *k0-in*: $k0 \in fst$ ‘ *set xs* $\cup$ *fst* ‘ *set* $((k'', v'') \# ys)$
also have ... $\subseteq$ *fst* ‘ *set* $((k', v') \# xs) \cup$ *fst* ‘ *set* $((k'', v'') \# ys)$ **by** *fastforce*
finally have *k0-in'*: $k0 \in$ *fst* ‘ *set* $((k', v') \# xs) \cup$ *fst* ‘ *set* $((k'', v'') \# ys)$ **.**
have $k' \neq k0$
proof
assume $k' = k0$
with *k0-in* **have** $k' \in$ *fst* ‘ *set xs* $\cup$ *fst* ‘ *set* $((k'', v'') \# ys)$ **by** *simp*
with *Lt* **have** $k' \in$ *fst* ‘ *set xs* $\vee k' \in$ *fst* ‘ *set ys* **by** *auto*
thus *False*
proof
assume $k' \in$ *fst* ‘ *set xs*
hence *lt k' k'* **by** (*rule* $*(4)$)
thus *?thesis* **by** *simp*
next
assume $k' \in$ *fst* ‘ *set ys*
hence *lt k'' k'* **by** (*rule Cons*(*4*))
with *Lt* **show** *?thesis* **by** (*simp add*: *Lt-lt-conv*)
qed
qed
hence *f k0* (*lookup-pair xs k0*) (*lookup-pair* $((k'', v'') \# ys)$ *k0*) =
f k0 (*lookup-pair* $((k', v') \# xs)$ *k0*) (*lookup-pair* $((k'', v'') \# ys)$ *k0*)
by (*simp add*: *lookup-pair-Cons*[*OF* $*(1)$] *del*: *lookup-pair.simps*)
also from *k0-in'* ‹*lt k0 k*› **have** ... = *Some Eq* **by** (*rule Cons*(*8*))
finally show *f k0* (*lookup-pair xs k0*) (*lookup-pair* $((k'', v'') \# ys)$ *k0*) = *Some Eq* **.**
qed
next
assume *f k' v' 0* $\neq$ *Some Eq*
have $\neg$ *lt k' k*
proof
have $k' \in$ *fst* ‘ *set* $((k', v') \# xs) \cup$ *fst* ‘ *set* $((k'', v'') \# ys)$ **by** *simp*
moreover assume *lt k' k*
ultimately have *f k'* (*lookup-pair* $((k', v') \# xs)$ *k'*) (*lookup-pair* $((k'', v'') \# ys)$ *k'*) = *Some Eq*
by (*rule Cons*(*8*))
hence *f k' v' 0* = *Some Eq* **by** (*simp add*: *Lt*)
with ‹*f k' v' 0* $\neq$ *Some Eq*› **show** *False* **..**
qed
moreover have $\neg$ *lt k k'*
proof
assume *lt k k'*
moreover from *this Lt* **have** *lt k k''* **by** (*simp add*: *Lt-lt-conv*)
ultimately show *False* **by** (*rule 0*)
qed
ultimately have $k = k'$ **by** *simp*

show *f k′ v′ 0* = *aux* **by** (*simp add*: *Cons*(*7*) ‹*k* = *k′*› *Lt*)
qed
next
assume *comp k′ k″* = *Eq*
hence *k′* = *k″* **by** (*simp only*: *eq*)
show (*let aux* = *f k′ v′ v″ in if aux* = *Some Eq then lex-ord-pair f xs ys else aux*) = *aux*
proof (*simp add*: *Let-def* ‹*k′* = *k″*› *split*: *order.split*, *intro conjI impI*)
assume *f k″ v′ v″* = *Some Eq*
have *k* ≠ *k″*
proof
assume *k* = *k″*
have *aux* = *f k v′ v″* **by** (*simp add*: *Cons*(*7*) ‹*k* = *k″*› ‹*k′* = *k″*›)
with ‹*f k″ v′ v″* = *Some Eq*› *assms*(*3*) **show** *False* **by** (*simp add*: ‹*k* = *k″*›)
qed
from *Cons*(*2*) **show** *lex-ord-pair f xs ys* = *aux*
proof (*rule* ∗(*5*))
from *Cons*(*6*) ‹*k* ≠ *k″*› **show** *k* ∈ *fst* ‘ *set xs* ∪ *fst* ‘ *set ys* **by** (*simp add*: ‹*k′* = *k″*›)
next
show *aux* = *f k* (*lookup-pair xs k*) (*lookup-pair ys k*)
by (*simp add*: *Cons*(*7*) *lookup-pair-Cons*[*OF* ∗(*1*)] *lookup-pair-Cons*[*OF Cons*(*1*)] *del*: *lookup-pair.simps*,
simp add: ‹*k′* = *k″*› ‹*k* ≠ *k″*›[*symmetric*])
next
fix *k0*
assume *lt k0 k*
assume *k0-in*: *k0* ∈ *fst* ‘ *set xs* ∪ *fst* ‘ *set ys*
also have ... ⊆ *fst* ‘ *set* ((*k′*, *v′*) # *xs*) ∪ *fst* ‘ *set* ((*k″*, *v″*) # *ys*) **by** *fastforce*
finally have *k0-in′*: *k0* ∈ *fst* ‘ *set* ((*k′*, *v′*) # *xs*) ∪ *fst* ‘ *set* ((*k″*, *v″*) # *ys*) **.**
have *k″* ≠ *k0*
proof
assume *k″* = *k0*
with *k0-in* **have** *k″* ∈ *fst* ‘ *set xs* ∪ *fst* ‘ *set ys* **by** *simp*
thus *False*
proof
assume *k″* ∈ *fst* ‘ *set xs*
hence *lt k′ k″* **by** (*rule* ∗(*4*))
thus *?thesis* **by** (*simp add*: ‹*k′* = *k″*›)
next
assume *k″* ∈ *fst* ‘ *set ys*
hence *lt k″ k″* **by** (*rule Cons*(*4*))
thus *?thesis* **by** *simp*
qed
qed
hence *f k0* (*lookup-pair xs k0*) (*lookup-pair ys k0*) =

f k0 (*lookup-pair* ((*k'*, *v'*) # *xs*) *k0*) (*lookup-pair* ((*k''*, *v''*) # *ys*) *k0*)
by (*simp add*: *lookup-pair-Cons*[*OF* ∗(*1*)] *lookup-pair-Cons*[*OF Cons*(*1*)]
del: *lookup-pair.simps*,
simp add: ‹*k' = k''*›)
also from *k0-in'* ‹*lt k0 k*› **have** ... = *Some Eq* **by** (*rule Cons*(*8*))
finally show *f k0* (*lookup-pair xs k0*) (*lookup-pair ys k0*) = *Some Eq* **.**
qed
next
assume *f k'' v' v''* ≠ *Some Eq*
have ¬ *lt k'' k*
proof
have *k''* ∈ *fst* ‘ *set* ((*k'*, *v'*) # *xs*) ∪ *fst* ‘ *set* ((*k''*, *v''*) # *ys*) **by** *simp*
moreover assume *lt k'' k*
ultimately have *f k''* (*lookup-pair* ((*k'*, *v'*) # *xs*) *k''*) (*lookup-pair* ((*k''*,
v'') # *ys*) *k''*) = *Some Eq*
by (*rule Cons*(*8*))
hence *f k'' v' v''* = *Some Eq* **by** (*simp add*: ‹*k' = k''*›)
with ‹*f k'' v' v''* ≠ *Some Eq*› **show** *False* **..**
qed
moreover have ¬ *lt k k''*
proof
assume *lt k k''*
hence *lt k k'* **by** (*simp only*: ‹*k' = k''*›)
thus *False* **using** ‹*lt k k''*› **by** (*rule 0*)
qed
ultimately have *k = k''* **by** *simp*
show *f k'' v' v''* = *aux* **by** (*simp add*: *Cons*(*7*) ‹*k = k''*› ‹*k' = k''*›)
qed
next
assume *Gt*: *comp k' k''* = *Gt*
hence *Lt*: *comp k'' k'* = *Lt* **by** (*simp only*: *Gt-lt-conv Lt-lt-conv*)
show (*let aux* = *f k'' 0 v'' in if aux* = *Some Eq then lex-ord-pair f* ((*k'*, *v'*)
xs) *ys else aux*) = *aux*
proof (*simp add*: *Let-def split*: *order.split*, *intro conjI impI*)
assume *f k'' 0 v''* = *Some Eq*
have *k* ≠ *k''*
proof
assume *k = k''*
have *aux* = *f k 0 v''* **by** (*simp add*: *Cons*(*7*) ‹*k = k''*› *Lt*)
with ‹*f k'' 0 v''* = *Some Eq*› *assms*(*3*) **show** *False* **by** (*simp add*: ‹*k* =
k''›)
qed
show *lex-ord-pair f* ((*k'*, *v'*) # *xs*) *ys* = *aux*
proof (*rule Cons*(*5*))
from *Cons*(*6*) ‹*k* ≠ *k''*› **show** *k* ∈ *fst* ‘ *set* ((*k'*, *v'*) # *xs*) ∪ *fst* ‘ *set ys*
by *simp*
next
show *aux* = *f k* (*lookup-pair* ((*k'*, *v'*) # *xs*) *k*) (*lookup-pair ys k*)
by (*simp add*: *Cons*(*7*) *lookup-pair-Cons*[*OF Cons*(*1*)] ‹*k* ≠ *k''*›[*symmetric*]

del: *lookup-pair.simps*)
next
fix *k0*
assume *lt k0 k*
assume *k0-in*: *k0* ∈ *fst* ‘ *set* ((*k*′, *v*′) # *xs*) ∪ *fst* ‘ *set ys*
also have ... ⊆ *fst* ‘ *set* ((*k*′, *v*′) # *xs*) ∪ *fst* ‘ *set* ((*k*′′, *v*′′) # *ys*) **by** *fastforce*
finally have *k0-in*′: *k0* ∈ *fst* ‘ *set* ((*k*′, *v*′) # *xs*) ∪ *fst* ‘ *set* ((*k*′′, *v*′′) # *ys*) **.**
have *k*′′ ≠ *k0*
proof
assume *k*′′ = *k0*
with *k0-in* **have** *k*′′ ∈ *fst* ‘ *set* ((*k*′, *v*′) # *xs*) ∪ *fst* ‘ *set ys* **by** *simp*
with *Lt* **have** *k*′′ ∈ *fst* ‘ *set xs* ∨ *k*′′ ∈ *fst* ‘ *set ys* **by** *auto*
thus *False*
proof
assume *k*′′ ∈ *fst* ‘ *set xs*
hence *lt k*′ *k*′′ **by** (*rule* ∗(*4*))
with *Lt* **show** *?thesis* **by** (*simp add*: *Lt-lt-conv*)
next
assume *k*′′ ∈ *fst* ‘ *set ys*
hence *lt k*′′ *k*′′ **by** (*rule Cons*(*4*))
thus *?thesis* **by** *simp*
qed
qed
hence *f k0* (*lookup-pair* ((*k*′, *v*′) # *xs*) *k0*) (*lookup-pair ys k0*) =
f k0 (*lookup-pair* ((*k*′, *v*′) # *xs*) *k0*) (*lookup-pair* ((*k*′′, *v*′′) # *ys*) *k0*)
by (*simp add*: *lookup-pair-Cons*[*OF Cons*(*1*)] *del*: *lookup-pair.simps*)
also from *k0-in*′ ‹*lt k0 k*› **have** ... = *Some Eq* **by** (*rule Cons*(*8*))
finally show *f k0* (*lookup-pair* ((*k*′, *v*′) # *xs*) *k0*) (*lookup-pair ys k0*) = *Some Eq* **.**
qed
next
assume *f k*′′ *0 v*′′ ≠ *Some Eq*
have ¬ *lt k*′′ *k*
proof
have *k*′′ ∈ *fst* ‘ *set* ((*k*′, *v*′) # *xs*) ∪ *fst* ‘ *set* ((*k*′′, *v*′′) # *ys*) **by** *simp*
moreover assume *lt k*′′ *k*
ultimately have *f k*′′ (*lookup-pair* ((*k*′, *v*′) # *xs*) *k*′′) (*lookup-pair* ((*k*′′, *v*′′) # *ys*) *k*′′) = *Some Eq*
by (*rule Cons*(*8*))
hence *f k*′′ *0 v*′′ = *Some Eq* **by** (*simp add*: *Lt*)
with ‹*f k*′′ *0 v*′′ ≠ *Some Eq*› **show** *False* **..**
qed
moreover have ¬ *lt k k*′′
proof
assume *lt k k*′′
with *Lt* **have** *lt k k*′ **by** (*simp add*: *Lt-lt-conv*)
thus *False* **using** ‹*lt k k*′′› **by** (*rule 0*)

qed
ultimately have $k = k''$ **by** *simp*
show $f\ k''\ 0\ v'' = aux$ **by** (*simp add*: *Cons*(*7*) ‹$k = k''$› *Lt*)
qed
qed
qed
qed

lemma *lex-ord-pair-EqD*:
assumes *oalist-inv-raw xs* **and** *oalist-inv-raw ys* **and** *lex-ord-pair f xs ys* = *Some Eq*
and $k \in fst$ ‘ *set xs* ∪ *fst* ‘ *set ys*
shows *f k* (*lookup-pair xs k*) (*lookup-pair ys k*) = *Some Eq*
proof (*rule ccontr*)
let *?A* = (*fst* ‘ *set xs* ∪ *fst* ‘ *set ys*) ∩ {*k*. *f k* (*lookup-pair xs k*) (*lookup-pair ys k*) ≠ *Some Eq*}
define *k0* **where** *k0* = *Min ?A*
have *finite ?A* **by** *auto*
assume *f k* (*lookup-pair xs k*) (*lookup-pair ys k*) ≠ *Some Eq*
with *assms*(*4*) **have** *k* ∈ *?A* **by** *simp*
hence *?A* ≠ {} **by** *blast*
with ‹*finite ?A*› **have** *k0* ∈ *?A* **unfolding** *k0-def* **by** (*rule Min-in*)
hence *k0-in*: *k0* ∈ *fst* ‘ *set xs* ∪ *fst* ‘ *set ys*
and *neq*: *f k0* (*lookup-pair xs k0*) (*lookup-pair ys k0*) ≠ *Some Eq* **by** *simp-all*
have *le k0 k′* **if** *k′* ∈ *?A* **for** *k′* **unfolding** *k0-def* **using** ‹*finite ?A*› *that*
by (*rule Min-le*)
hence *f k′* (*lookup-pair xs k′*) (*lookup-pair ys k′*) = *Some Eq*
if *k′* ∈ *fst* ‘ *set xs* ∪ *fst* ‘ *set ys* **and** *lt k′ k0* **for** *k′* **using** *that* **by** *fastforce*
with *assms*(*1*, *2*) *neq k0-in HOL.refl* **have** *lex-ord-pair f xs ys* = *f k0* (*lookup-pair xs k0*) (*lookup-pair ys k0*)
by (*rule lex-ord-pair-valI*)
with *assms*(*3*) *neq* **show** *False* **by** *simp*
qed

lemma *lex-ord-pair-valE*:
assumes *oalist-inv-raw xs* **and** *oalist-inv-raw ys* **and** *lex-ord-pair f xs ys* = *aux*
and *aux* ≠ *Some Eq*
obtains *k* **where** *k* ∈ *fst* ‘ *set xs* ∪ *fst* ‘ *set ys* **and** *aux* = *f k* (*lookup-pair xs k*) (*lookup-pair ys k*)
and ⋀*k′*. *k′* ∈ *fst* ‘ *set xs* ∪ *fst* ‘ *set ys* ⟹ *lt k′ k* ⟹
f k′ (*lookup-pair xs k′*) (*lookup-pair ys k′*) = *Some Eq*
proof −
let *?A* = (*fst* ‘ *set xs* ∪ *fst* ‘ *set ys*) ∩ {*k*. *f k* (*lookup-pair xs k*) (*lookup-pair ys k*) ≠ *Some Eq*}
define *k* **where** *k* = *Min ?A*
have *finite ?A* **by** *auto*
have ∃ *k* ∈ *fst* ‘ *set xs* ∪ *fst* ‘ *set ys*. *f k* (*lookup-pair xs k*) (*lookup-pair ys k*) ≠ *Some Eq* (**is** *?prop*)
proof (*rule ccontr*)

```
    assume ¬ ?prop
    hence f k (lookup-pair xs k) (lookup-pair ys k) = Some Eq
      if k ∈ fst ‘ set xs ∪ fst ‘ set ys for k using that by auto
   with assms(1, 2) have lex-ord-pair f xs ys = Some Eq by (rule lex-ord-pair-EqI)
    with assms(3, 4) show False by simp
  qed
  then obtain k0 where k0 ∈ fst ‘ set xs ∪ fst ‘ set ys
    and f k0 (lookup-pair xs k0) (lookup-pair ys k0) ≠ Some Eq ..
  hence k0 ∈ ?A by simp
  hence ?A ≠ {} by blast
  with ‹finite ?A› have k ∈ ?A unfolding k-def by (rule Min-in)
  hence k-in: k ∈ fst ‘ set xs ∪ fst ‘ set ys
    and neq: f k (lookup-pair xs k) (lookup-pair ys k) ≠ Some Eq by simp-all
  have le k k' if k' ∈ ?A for k' unfolding k-def using ‹finite ?A› that
    by (rule Min-le)
  hence *: ⋀k'. k' ∈ fst ‘ set xs ∪ fst ‘ set ys ⟹ lt k' k ⟹
          f k' (lookup-pair xs k') (lookup-pair ys k') = Some Eq by fastforce
  with assms(1, 2) neq k-in HOL.refl have lex-ord-pair f xs ys = f k (lookup-pair
xs k) (lookup-pair ys k)
    by (rule lex-ord-pair-valI)
  hence aux = f k (lookup-pair xs k) (lookup-pair ys k) by (simp only: assms(3))
  with k-in show ?thesis using * ..
qed
```

12.4.7 *prod-ord-pair*

```
lemma prod-ord-pair-eq-lex-ord-pair:
  prod-ord-pair P xs ys = (lex-ord-pair (λk x y. if P k x y then Some Eq else None)
xs ys = Some Eq)
proof (induct P xs ys rule: prod-ord-pair.induct)
  case (1 P)
  show ?case by simp
next
  case (2 P ky vy ys)
  thus ?case by simp
next
  case (3 P kx vx xs)
  thus ?case by simp
next
  case (4 P kx vx xs ky vy ys)
  show ?case
  proof (cases comp kx ky)
    case Lt
    thus ?thesis by (simp add: 4(2)[OF Lt])
  next
    case Eq
    thus ?thesis by (simp add: 4(1)[OF Eq])
  next
    case Gt
```

thus *?thesis* **by** (*simp add*: *4*(*3*)[*OF Gt*])
qed
qed

lemma *prod-ord-pairI*:
assumes *oalist-inv-raw xs* **and** *oalist-inv-raw ys*
and $\bigwedge k.\ k \in fst$ *‘ set xs* ∪ *fst ‘ set ys* ⟹ *P k* (*lookup-pair xs k*) (*lookup-pair ys k*)
shows *prod-ord-pair P xs ys*
unfolding *prod-ord-pair-eq-lex-ord-pair* **by** (*rule lex-ord-pair-EqI*, *fact*, *fact*, *simp add*: *assms*(*3*))

lemma *prod-ord-pairD*:
assumes *oalist-inv-raw xs* **and** *oalist-inv-raw ys* **and** *prod-ord-pair P xs ys*
and *k* ∈ *fst ‘ set xs* ∪ *fst ‘ set ys*
shows *P k* (*lookup-pair xs k*) (*lookup-pair ys k*)
proof −
from *assms* **have** (*if P k* (*lookup-pair xs k*) (*lookup-pair ys k*) *then Some Eq else None*) = *Some Eq*
unfolding *prod-ord-pair-eq-lex-ord-pair* **by** (*rule lex-ord-pair-EqD*)
thus *?thesis* **by** (*simp split*: *if-splits*)
qed

corollary *prod-ord-pair-alt*:
assumes *oalist-inv-raw xs* **and** *oalist-inv-raw ys*
shows (*prod-ord-pair P xs ys*) ⟷ (∀ *k*∈*fst ‘ set xs* ∪ *fst ‘ set ys*. *P k* (*lookup-pair xs k*) (*lookup-pair ys k*))
using *prod-ord-pairI*[*OF assms*] *prod-ord-pairD*[*OF assms*] **by** *meson*

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 (*induct xs*)
case *Nil*
from *assms* **show** *?case* **by** *simp*
next
case (*Cons x xs*)
hence *oalist-inv-raw* (*update-by-pair x* (*foldr update-by-pair xs ys*))
by (*rule oalist-inv-raw-update-by-pair*)
thus *?case* **by** *simp*
qed

corollary *oalist-inv-raw-sort-oalist*: *oalist-inv-raw* (*sort-oalist xs*)
proof −
from *oalist-inv-raw-Nil* **have** *oalist-inv-raw* (*foldr local.update-by-pair xs* [])
by (*rule oalist-inv-raw-foldr-update-by-pair*)
thus *oalist-inv-raw* (*sort-oalist xs*) **by** (*simp only*: *sort-oalist-def*)

qed

lemma *sort-oalist-id*:
  assumes *oalist-inv-raw xs*
  shows *sort-oalist xs* = *xs*
proof −
  have *foldr update-by-pair xs ys* = *xs* @ *ys* **if** *oalist-inv-raw* (*xs* @ *ys*) **for** *ys* **using** *assms that*
  proof (*induct xs rule*: *oalist-inv-raw-induct*)
    case *Nil*
    show *?case* **by** *simp*
  next
    case (*Cons k v xs*)
    from *Cons*(*6*) **have** ∗: *oalist-inv-raw* ((*k*, *v*) # (*xs* @ *ys*)) **by** *simp*
    hence *1*: *oalist-inv-raw* (*xs* @ *ys*) **by** (*rule oalist-inv-raw-ConsD1*)
    hence *2*: *foldr update-by-pair xs ys* = *xs* @ *ys* **by** (*rule Cons*(*5*))
    show *?case*
    proof (*simp add*: *2*, *rule update-by-pair-less*)
      from ∗ **show** *v* ≠ *0* **by** (*auto simp*: *oalist-inv-raw-def*)
    next
      have *comp k* (*fst* (*hd* (*xs* @ *ys*))) = *Lt* ∨ *xs* @ *ys* = []
      proof (*rule disjCI*)
        assume *xs* @ *ys* ≠ []
        then obtain *k′′ v′′ zs* **where** *eq0*: *xs* @ *ys* = (*k′′*, *v′′*) # *zs*
          using *list.exhaust prod.exhaust* **by** *metis*
        from ∗ **have** *lt k k′′* **by** (*simp add*: *eq0 oalist-inv-raw-def*)
        thus *comp k* (*fst* (*hd* (*xs* @ *ys*))) = *Lt* **by** (*simp add*: *eq0 Lt-lt-conv*)
      qed
      thus *xs* @ *ys* = [] ∨ *comp k* (*fst* (*hd* (*xs* @ *ys*))) = *Lt* **by** *auto*
    qed
  qed
  with *assms* **show** *?thesis* **by** (*simp add*: *sort-oalist-def*)
qed

lemma *set-sort-oalist*:
  assumes *distinct* (*map fst xs*)
  shows *set* (*sort-oalist xs*) = {*kv*. *kv* ∈ *set xs* ∧ *snd kv* ≠ *0*}
  using *assms*
proof (*induct xs*)
  case *Nil*
  show *?case* **by** (*simp add*: *sort-oalist-def*)
next
  case (*Cons x xs*)
  obtain *k v* **where** *x*: *x* = (*k*, *v*) **by** *fastforce*
  from *Cons*(*2*) **have** *distinct* (*map fst xs*) **and** *k* ∉ *fst* ‘ *set xs* **by** (*simp-all add*: *x*)
  from *this*(*1*) **have** *set* (*sort-oalist xs*) = {*kv* ∈ *set xs*. *snd kv* ≠ *0*} **by** (*rule Cons*(*1*))
  with ‹*k* ∉ *fst* ‘ *set xs*› **have** *eq*: *set* (*sort-oalist xs*) − *range* (*Pair k*) = {*kv* ∈ *set*

xs. *snd kv* ≠ *0*}
    by (*auto simp*: *image-iff*)
  have *set* (*sort-oalist* (*x* # *xs*)) = *set* (*update-by-pair* (*k*, *v*) (*sort-oalist xs*))
    by (*simp add*: *sort-oalist-def x*)
  also have ... = {*kv* ∈ *set* (*x* # *xs*). *snd kv* ≠ *0*}
  proof (*cases v* = *0*)
    case *True*
    have *set* (*update-by-pair* (*k*, *v*) (*sort-oalist xs*)) = *set* (*sort-oalist xs*) − *range* (*Pair k*)
    unfolding *True* **using** *oalist-inv-raw-sort-oalist* **by** (*rule set-update-by-pair-zero*)
    also have ... = {*kv* ∈ *set* (*x* # *xs*). *snd kv* ≠ *0*} **by** (*auto simp*: *eq x True*)
    finally show *?thesis* **.**
  next
    case *False*
    with *oalist-inv-raw-sort-oalist*
    have *set* (*update-by-pair* (*k*, *v*) (*sort-oalist xs*)) = *insert* (*k*, *v*) (*set* (*sort-oalist xs*) − *range* (*Pair k*))
      by (*rule set-update-by-pair*)
    also have ... = {*kv* ∈ *set* (*x* # *xs*). *snd kv* ≠ *0*} **by** (*auto simp*: *eq x False*)
    finally show *?thesis* **.**
  qed
  finally show *?case* **.**
qed

lemma *lookup-pair-sort-oalist′*:
  assumes *distinct* (*map fst xs*)
  shows *lookup-pair* (*sort-oalist xs*) = *lookup-dflt xs*
  using *assms*
proof (*induct xs*)
  case *Nil*
  show *?case* **by** (*simp add*: *sort-oalist-def lookup-dflt-def*)
next
  case (*Cons x xs*)
  obtain *k v* **where** *x*: *x* = (*k*, *v*) **by** *fastforce*
  from *Cons*(*2*) **have** *distinct* (*map fst xs*) **and** *k* ∉ *fst* ‘ *set xs* **by** (*simp-all add*: *x*)
  from *this*(*1*) **have** *eq1*: *lookup-pair* (*sort-oalist xs*) = *lookup-dflt xs* **by** (*rule Cons*(*1*))
  have *eq2*: *sort-oalist* (*x* # *xs*) = *update-by-pair* (*k*, *v*) (*sort-oalist xs*) **by** (*simp add*: *x sort-oalist-def*)
  show *?case*
  proof
    fix *k′*
    have *lookup-pair* (*sort-oalist* (*x* # *xs*)) *k′* = (*if k* = *k′ then v else lookup-dflt xs k′*)
     by (*simp add*: *eq1 eq2 lookup-pair-update-by-pair*[*OF oalist-inv-raw-sort-oalist*])
    also have ... = *lookup-dflt* (*x* # *xs*) *k′* **by** (*simp add*: *x lookup-dflt-def*)
    finally show *lookup-pair* (*sort-oalist* (*x* # *xs*)) *k′* = *lookup-dflt* (*x* # *xs*) *k′* **.**
  qed

```
qed

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 –
  have rl: set (foldr update-by-pair xs ys) = set xs ∪ set ys
    if oalist-inv-raw ys and fst ‘ set xs ∩ fst ‘ set ys = {} for ys
    using assms that(2)
  proof (induct xs rule: cmp2.oalist-inv-raw-induct)
    case Nil
    show ?case by simp
  next
    case (Cons k v xs)
     from Cons(6) have k ∉ fst ‘ set ys and fst ‘ set xs ∩ fst ‘ set ys = {} by
simp-all
     from this(2) have eq1: set (foldr update-by-pair xs ys) = set xs ∪ set ys by
(rule Cons(5))
    have ¬ cmp2.lt k k by auto
    with Cons(4) have k ∉ fst ‘ set xs by blast
    with ‹k ∉ fst ‘ set ys› have k ∉ fst ‘ (set xs ∪ set ys) by (simp add: image-Un)
    hence (set xs ∪ set ys) ∩ range (Pair k) = {} by (smt (verit) Int-emptyI fstI
image-iff)
     hence eq2: (set xs ∪ set ys) − range (Pair k) = set xs ∪ set ys by (rule
Diff-triv)
    from ‹oalist-inv-raw ys› have oalist-inv-raw (foldr update-by-pair xs ys)
      by (rule oalist-inv-raw-foldr-update-by-pair)
    hence set (update-by-pair (k, v) (foldr update-by-pair xs ys)) =
            insert (k, v) (set (foldr update-by-pair xs ys) − range (Pair k))
      using Cons(3) by (rule set-update-by-pair)
    also have ... = insert (k, v) (set xs ∪ set ys) by (simp only: eq1 eq2)
    finally show ?case by simp
  qed
  have set (foldr update-by-pair xs []) = set xs ∪ set []
    by (rule rl, fact oalist-inv-raw-Nil, simp)
  thus ?thesis by (simp add: sort-oalist-def)
qed

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
  fix k
```

show *lookup-pair xs k = cmp2.lookup-pair ys k*
proof (*cases cmp2.lookup-pair ys k = 0*)
case *True*
with *assms*(*2*) **have** $k \notin$ *fst ' set ys* **by** (*simp add*: *cmp2.lookup-pair-eq-0*)
with *assms*(*1*) **show** *?thesis* **by** (*simp add*: *True assms*(*3*)[*symmetric*] *lookup-pair-eq-0*)
next
case *False*
define *v* **where** *v = cmp2.lookup-pair ys k*
from *False* **have** $v \neq 0$ **by** (*simp add*: *v-def*)
with *assms*(*2*) *v-def*[*symmetric*] **have** $(k, v) \in$ *set ys* **by** (*simp add*: *cmp2.lookup-pair-eq-value*)
with *assms*(*1*) ‹$v \neq 0$› **have** *lookup-pair xs k = v*
by (*simp add*: *assms*(*3*)[*symmetric*] *lookup-pair-eq-value*)
thus *?thesis* **by** (*simp only*: *v-def*)
qed
qed

corollary *lookup-pair-sort-oalist*:
assumes *cmp2.oalist-inv-raw xs*
shows *lookup-pair* (*sort-oalist xs*) = *cmp2.lookup-pair xs*
by (*rule lookup-pair-eqI*, *rule oalist-inv-raw-sort-oalist*, *fact*, *rule set-sort-oalist*, *fact*)

end

12.5 Invariant on Pairs

type-synonym (*'a*, *'b*, *'c*) *oalist-raw* = (*'a* × *'b*) *list* × *'c*

locale *oalist-raw* = **fixes** *rep-key-order*::*'o* ⇒ *'a key-order*
begin

sublocale *comparator key-compare* (*rep-key-order x*)
by (*fact comparator-key-compare*)

definition *oalist-inv* :: (*'a*, *'b*::*zero*, *'o*) *oalist-raw* ⇒ *bool*
where *oalist-inv xs* ⟷ *oalist-inv-raw* (*snd xs*) (*fst xs*)

lemma *oalist-inv-alt*: *oalist-inv* (*xs*, *ko*) ⟷ *oalist-inv-raw ko xs*
by (*simp add*: *oalist-inv-def*)

12.6 Operations on Raw Ordered Associative Lists

fun *sort-oalist-aux* :: *'o* ⇒ (*'a*, *'b*, *'o*) *oalist-raw* ⇒ (*'a* × *'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* ⇒ *'a* ⇒ *'b*::*zero*
where *lookup-raw* (*xs*, *ko*) = *lookup-pair ko xs*

definition *sorted-domain-raw* :: *'o* ⇒ (*'a*, *'b*::*zero*, *'o*) *oalist-raw* ⇒ *'a list*
where *sorted-domain-raw ko xs* = *map fst* (*sort-oalist-aux ko xs*)

fun *tl-raw* :: ('*a*, '*b*, '*o*) *oalist-raw* ⇒ ('*a*, '*b*::*zero*, '*o*) *oalist-raw*
where *tl-raw* (*xs*, *ko*) = (*List.tl xs*, *ko*)

fun *min-key-val-raw* :: '*o* ⇒ ('*a*, '*b*, '*o*) *oalist-raw* ⇒ ('*a* × '*b*::*zero*)
where *min-key-val-raw ko* (*xs*, *ox*) =
(*if ko* = *ox then List.hd else min-list-param* (λ*x y*. *le ko* (*fst x*) (*fst y*))) *xs*

fun *update-by-raw* :: ('*a* × '*b*) ⇒ ('*a*, '*b*, '*o*) *oalist-raw* ⇒ ('*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* ⇒ ('*b* ⇒ '*b*) ⇒ ('*a*, '*b*, '*o*) *oalist-raw* ⇒ ('*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* ⇒ ('*b* ⇒ '*b*) ⇒ ('*a*, '*b*, '*o*) *oalist-raw* ⇒ ('*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* ⇒ *bool*) ⇒ ('*a list* × '*b*) ⇒ ('*a list* × '*b*)
where *filter-raw P* (*xs*, *ko*) = (*filter P xs*, *ko*)

fun (**in** −) *map-raw* :: (('*a* × '*b*) ⇒ ('*a* × '*c*)) ⇒ (('*a* × '*b*::*zero*) *list* × '*d*) ⇒ ('*a* × '*c*::*zero*) *list* × '*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 −
obtain *xs' ko'* **where** *xs*: *xs* = (*xs'*, *ko'*) **by** *fastforce*
interpret *ko2*: *comparator2 key-compare* (*rep-key-order ko*) *key-compare* (*rep-key-order ko'*) **..**
from *assms* **show** *?thesis* **by** (*simp add*: *xs oalist-inv-alt ko2.set-sort-oalist*)
qed

lemma *oalist-inv-raw-sort-oalist-aux*:
assumes *oalist-inv xs*
shows *oalist-inv-raw ko* (*sort-oalist-aux ko xs*)
proof −
obtain *xs' ko'* **where** *xs*: *xs* = (*xs'*, *ko'*) **by** *fastforce*
from *assms* **show** *?thesis* **by** (*simp add*: *xs oalist-inv-alt oalist-inv-raw-sort-oalist*)
qed

lemma *oalist-inv-sort-oalist-aux*:
assumes *oalist-inv xs*
shows *oalist-inv* (*sort-oalist-aux ko xs*, *ko*)
unfolding *oalist-inv-alt* **using** *assms* **by** (*rule oalist-inv-raw-sort-oalist-aux*)

lemma *lookup-pair-sort-oalist-aux*:
assumes *oalist-inv xs*
shows *lookup-pair ko* (*sort-oalist-aux ko xs*) = *lookup-raw xs*
proof −
obtain *xs' ko'* **where** *xs*: *xs* = (*xs'*, *ko'*) **by** *fastforce*
interpret *ko2*: *comparator2 key-compare* (*rep-key-order ko*) *key-compare* (*rep-key-order ko'*) **..**
from *assms* **show** *?thesis* **by** (*simp add*: *xs oalist-inv-alt ko2.lookup-pair-sort-oalist*)
qed

12.6.2 *lookup-raw*

lemma *lookup-raw-eq-value*:
assumes *oalist-inv xs* **and** $v \neq 0$
shows *lookup-raw xs k* = *v* ⟷ ((*k*, *v*) ∈ *set* (*fst xs*))
proof −
obtain *xs' ox* **where** *xs*: *xs* = (*xs'*, *ox*) **by** *fastforce*
from *assms*(*1*) **have** *oalist-inv-raw ox xs'* **by** (*simp add*: *xs oalist-inv-def*)
show *?thesis* **by** (*simp add*: *xs*, *rule lookup-pair-eq-value*, *fact+*)
qed

lemma *lookup-raw-eq-valueI*:

assumes *oalist-inv xs* **and** $(k, v) \in set\ (fst\ xs)$
shows *lookup-raw xs k* = *v*
proof −
obtain *xs′ ox* **where** *xs*: *xs* = (*xs′*, *ox*) **by** *fastforce*
from *assms*(*1*) **have** *oalist-inv-raw ox xs′* **by** (*simp add*: *xs oalist-inv-def*)
from *assms*(*2*) **have** $(k, v) \in set\ xs'$ **by** (*simp add*: *xs*)
show *?thesis* **by** (*simp add*: *xs*, *rule lookup-pair-eq-valueI*, *fact*+)
qed

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*
using *assms* **unfolding** *lookup-raw.simps oalist-inv-alt* **by** (*rule lookup-pair-inj*)

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*)
using *assms* **by** (*simp add*: *sorted-domain-raw-def set-sort-oalist-aux*)

corollary *in-sorted-domain-raw-iff-lookup-raw*:
assumes *oalist-inv xs*
shows $k \in set$ (*sorted-domain-raw ko xs*) $\longleftrightarrow$ (*lookup-raw xs k* $\neq$ *0*)
unfolding *set-sorted-domain-raw*[*OF assms*]
proof −
obtain *xs′ ko′* **where** *xs*: *xs* = (*xs′*, *ko′*) **by** *fastforce*
from *assms* **show** $k \in fst$ ‘ *set* (*fst xs*) $\longleftrightarrow$ (*lookup-raw xs k* $\neq$ *0*)
by (*simp add*: *xs oalist-inv-alt lookup-pair-eq-0*)
qed

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*)
unfolding *sorted-domain-raw-def oalist-inv-alt lt-of-key-order.rep-eq*
by (*rule oalist-inv-rawD2*, *rule oalist-inv-raw-sort-oalist-aux*, *fact*)

12.6.4 *tl-raw*

lemma *oalist-inv-tl-raw*:
assumes *oalist-inv xs*
shows *oalist-inv* (*tl-raw xs*)
proof −
obtain *xs′ ko* **where** *xs*: *xs* = (*xs′*, *ko*) **by** *fastforce*
from *assms* **show** *?thesis* **unfolding** *xs tl-raw.simps oalist-inv-alt* **by** (*rule oalist-inv-raw-tl*)
qed

lemma *lookup-raw-tl-raw*:

assumes *oalist-inv xs*
shows *lookup-raw* (*tl-raw xs*) *k* =
(*if* (∀ *k'*∈*fst* ' *set* (*fst xs*). *le* (*snd xs*) *k k'*) *then 0 else lookup-raw xs k*)
proof −
obtain *xs' ko* **where** *xs*: *xs* = (*xs'*, *ko*) **by** *fastforce*
from *assms* **show** *?thesis* **by** (*simp add*: *xs lookup-pair-tl oalist-inv-alt split del*: *if-split cong*: *if-cong*)
qed

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 −
obtain *xs' ko* **where** *xs*: *xs* = (*xs'*, *ko*) **by** *fastforce*
from *assms* **show** *?thesis* **by** (*simp add*: *xs lookup-pair-tl' oalist-inv-alt*)
qed

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 −
obtain *xs' ox* **where** *xs*: *xs* = (*xs'*, *ox*) **by** *fastforce*
from *assms*(*2*) **have** *xs'* ≠ [] **by** (*simp add*: *xs*)
interpret *ko2*: *comparator2 key-compare* (*rep-key-order ko*) *key-compare* (*rep-key-order ox*) ..
from *assms*(*1*) **have** *oalist-inv-raw ox xs'* **by** (*simp only*: *xs oalist-inv-alt*)
hence *set-sort*: *set* (*sort-oalist ko xs'*) = *set xs'* **by** (*rule ko2.set-sort-oalist*)
also from ‹*xs'* ≠ []› **have** ... ≠ {} **by** *simp*
finally have *sort-oalist ko xs'* ≠ [] **by** *simp*
then obtain *k v xs''* **where** *eq*: *sort-oalist ko xs'* = (*k*, *v*) # *xs''*
using *prod.exhaust list.exhaust* **by** *metis*
hence (*k*, *v*) ∈ *set xs'* **by** (*simp add*: *set-sort*[*symmetric*])
have ∗: *le ko k k'* **if** *k'* ∈ *fst* ' *set xs'* **for** *k'*
proof −
from *that* **have** *k'* = *k* ∨ *k'* ∈ *fst* ' *set xs''* **by** (*simp add*: *set-sort*[*symmetric*] *eq*)
thus *?thesis*
proof
assume *k'* = *k*
thus *?thesis* **by** *simp*
next
have *oalist-inv-raw ko* ((*k*, *v*) # *xs''*) **unfolding** *eq*[*symmetric*] **by** (*fact oalist-inv-raw-sort-oalist*)
moreover assume *k'* ∈ *fst* ' *set xs''*
ultimately have *lt ko k k'* **by** (*rule oalist-inv-raw-ConsD3*)
thus *?thesis* **by** *simp*

qed
qed
from ‹*xs'* ≠ []› **have** *min-list-param* (λ*x y*. *le ko* (*fst x*) (*fst y*)) *xs'* ∈ *set xs'* **by** (*rule min-list-param-in*)
with ‹*oalist-inv-raw ox xs'*› **have** *lookup-pair ox xs'* (*fst* (*min-list-param* (λ*x y*. *le ko* (*fst x*) (*fst y*)) *xs'*)) =
snd (*min-list-param* (λ*x y*. *le ko* (*fst x*) (*fst y*)) *xs'*) **by** (*auto intro*: *lookup-pair-eq-valueI*)
moreover have *1*: *fst* (*min-list-param* (λ*x y*. *le ko* (*fst x*) (*fst y*)) *xs'*) = *k*
proof (*rule antisym*)
from *order-trans*
have *transp* (λ*x y*. *le ko* (*fst x*) (*fst y*)) **by** (*rule transpI*)
hence *le ko* (*fst* (*min-list-param* (λ*x y*. *le ko* (*fst x*) (*fst y*)) *xs'*)) (*fst* (*k*, *v*))
using *linear* ‹(*k*, *v*) ∈ *set xs'*› **by** (*rule min-list-param-minimal*)
thus *le ko* (*fst* (*min-list-param* (λ*x y*. *le ko* (*fst x*) (*fst y*)) *xs'*)) *k* **by** *simp*
next
show *le ko k* (*fst* (*min-list-param* (λ*x y*. *le ko* (*fst x*) (*fst y*)) *xs'*)) **by** (*rule* ∗, *rule imageI*, *fact*)
qed
ultimately have *snd* (*min-list-param* (λ*x y*. *le ko* (*fst x*) (*fst y*)) *xs'*) = *lookup-pair ox xs' k*
by *simp*
also from ‹*oalist-inv-raw ox xs'*› ‹(*k*, *v*) ∈ *set xs'*› **have** ... = *v* **by** (*rule lookup-pair-eq-valueI*)
finally have *snd* (*min-list-param* (λ*x y*. *le ko* (*fst x*) (*fst y*)) *xs'*) = *v* **.**
with *1* **have** *min-list-param* (λ*x y*. *le ko* (*fst x*) (*fst y*)) *xs'* = (*k*, *v*) **by** *auto*
thus *?thesis* **by** (*simp add*: *xs eq*)
qed

lemma *min-key-val-raw-in*:
assumes *fst xs* ≠ []
shows *min-key-val-raw ko xs* ∈ *set* (*fst xs*)
proof −
obtain *xs' ox* **where** *xs*: *xs* = (*xs'*, *ox*) **by** *fastforce*
from *assms* **have** *xs'* ≠ [] **by** (*simp add*: *xs*)
show *?thesis* **unfolding** *xs*
proof (*simp*, *intro conjI impI*)
from ‹*xs'* ≠ []› **show** *hd xs'* ∈ *set xs'* **by** *simp*
next
from ‹*xs'* ≠ []› **show** *min-list-param* (λ*x y*. *le ko* (*fst x*) (*fst y*)) *xs'* ∈ *set xs'*
by (*rule min-list-param-in*)
qed
qed

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 −
obtain *xs' ox* **where** *xs*: *xs* = (*xs'*, *ox*) **by** *fastforce*
from *assms*(*1*) **have** *oalist-inv-raw ox xs'* **by** (*simp only*: *xs oalist-inv-alt*)

from *assms*(*2*) **have** *min-key-val-raw ko xs* ∈ *set* (*fst xs*) **by** (*rule min-key-val-raw-in*)
hence ∗: *min-key-val-raw ko* (*xs*′, *ox*) ∈ *set xs*′ **by** (*simp add*: *xs*)
show *?thesis* **unfolding** *xs lookup-raw.simps*
by (*rule HOL.sym*, *rule lookup-pair-eq-valueI*, *fact*, *simp add*: ∗ *del*: *min-key-val-raw.simps*)
qed

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 −
obtain *xs*′ *ox* **where** *xs*: *xs* = (*xs*′, *ox*) **by** *fastforce*
from *assms* **have** *oalist-inv* (*xs*′, *ox*) **and** *z* ∈ *set xs*′ **by** (*simp-all add*: *xs*)
show *?thesis* **unfolding** *xs*
proof (*simp*, *intro conjI impI*)
from ‹*z* ∈ *set xs*′› **have** *xs*′ ≠ [] **by** *auto*
then obtain *k v ys* **where** *xs*′: *xs*′ = (*k*, *v*) # *ys* **using** *prod.exhaust list.exhaust*
by *metis*
from ‹*z* ∈ *set xs*′› **have** *z* = (*k*, *v*) ∨ *z* ∈ *set ys* **by** (*simp add*: *xs*′)
thus *le ox* (*fst* (*hd xs*′)) (*fst z*)
proof
assume *z* = (*k*, *v*)
show *?thesis* **by** (*simp add*: *xs*′ ‹*z* = (*k*, *v*)›)
next
assume *z* ∈ *set ys*
hence *fst z* ∈ *fst* ‘ *set ys* **by** *fastforce*
with ‹*oalist-inv* (*xs*′, *ox*)› **have** *lt ox k* (*fst z*)
unfolding *xs*′ *oalist-inv-alt lt-of-key-order.rep-eq* **by** (*rule oalist-inv-raw-ConsD3*)
thus *?thesis* **by** (*simp add*: *xs*′)
qed
next
show *le ko* (*fst* (*min-list-param* (λ*x y*. *le ko* (*fst x*) (*fst y*)) *xs*′)) (*fst z*)
proof (*rule min-list-param-minimal*[*of* λ*x y*. *le ko* (*fst x*) (*fst y*)])
thm *trans local.trans order.trans local.order-trans*
print-context
show *transp* (λ*x y*. *le ko* (*fst x*) (*fst y*)) **by** (*metis* (*no-types*, *lifting*) *order-trans*
transpI)
qed (*auto intro*: ‹*z* ∈ *set xs*′›)
qed
qed

12.6.6 *filter-raw*

lemma *oalist-inv-filter-raw*:
assumes *oalist-inv xs*
shows *oalist-inv* (*filter-raw P xs*)
proof −
obtain *xs*′ *ko* **where** *xs*: *xs* = (*xs*′, *ko*) **by** *fastforce*
from *assms* **show** *?thesis* **unfolding** *xs filter-raw.simps oalist-inv-alt*
by (*rule oalist-inv-raw-filter*)

qed

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 −
obtain *xs′ ko* **where** *xs*: *xs* = (*xs′*, *ko*) **by** *fastforce*
from *assms* **show** *?thesis* **unfolding** *xs lookup-raw.simps filter-raw.simps oalist-inv-alt*
by (*rule lookup-pair-filter*)
qed

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 −
obtain *xs′ ko* **where** *xs*: *xs* = (*xs′*, *ko*) **by** *fastforce*
from *assms* **show** *?thesis* **unfolding** *xs update-by-raw.simps oalist-inv-alt*
by (*rule oalist-inv-raw-update-by-pair*)
qed

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 −
obtain *xs′ ko* **where** *xs*: *xs* = (*xs′*, *ko*) **by** *fastforce*
from *assms* **show** *?thesis* **unfolding** *xs lookup-raw.simps update-by-raw.simps oalist-inv-alt*
by (*rule lookup-pair-update-by-pair*)
qed

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 −
obtain *xs′ ko* **where** *xs*: *xs* = (*xs′*, *ko*) **by** *fastforce*
from *assms* **have** *oalist-inv-raw ko xs′* **by** (*simp only*: *xs oalist-inv-alt*)
show *?thesis* **unfolding** *xs update-by-fun-raw.simps lookup-raw.simps update-by-raw.simps*
by (*rule*, *rule conjI*, *rule update-by-fun-pair-eq-update-by-pair*, *fact*, *fact HOL.refl*)
qed

corollary *oalist-inv-update-by-fun-raw*:
assumes *oalist-inv xs*
shows *oalist-inv* (*update-by-fun-raw k f xs*)

unfolding *update-by-fun-raw-eq-update-by-raw*[*OF assms*] **using** *assms* **by** (*rule oalist-inv-update-by-raw*)

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*)
using *assms* **by** (*simp add*: *update-by-fun-raw-eq-update-by-raw lookup-raw-update-by-raw*)

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 −
obtain *xs′ ko* **where** *xs*: *xs* = (*xs′*, *ko*) **by** *fastforce*
from *assms* **have** *oalist-inv-raw ko xs′* **by** (*simp only*: *xs oalist-inv-alt*)
show *?thesis* **unfolding** *xs update-by-fun-raw.simps lookup-raw.simps update-by-fun-gr-raw.simps*
by (*rule*, *rule conjI*, *rule update-by-fun-gr-pair-eq-update-by-fun-pair*, *fact*, *fact HOL.refl*)
qed

corollary *oalist-inv-update-by-fun-gr-raw*:
assumes *oalist-inv xs*
shows *oalist-inv* (*update-by-fun-gr-raw k f xs*)
unfolding *update-by-fun-gr-raw-eq-update-by-fun-raw*[*OF assms*] **using** *assms* **by** (*rule oalist-inv-update-by-fun-raw*)

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*)
using *assms* **by** (*simp add*: *update-by-fun-gr-raw-eq-update-by-fun-raw lookup-raw-update-by-fun-raw*)

12.6.9 *map-raw* and *map-val-raw*

lemma *map-raw-cong*:
assumes $\bigwedge$*kv*. *kv* ∈ *set* (*fst xs*) ⟹ *f kv* = *g kv*
shows *map-raw f xs* = *map-raw g xs*
proof −
obtain *xs′ ko* **where** *xs*: *xs* = (*xs′*, *ko*) **by** *fastforce*
from *assms* **have** *f kv* = *g kv* **if** *kv* ∈ *set xs′* **for** *kv* **using** *that* **by** (*simp add*: *xs*)
thus *?thesis* **by** (*simp add*: *xs*, *rule map-pair-cong*)
qed

lemma *map-raw-subset*: *set* (*fst* (*map-raw f xs*)) ⊆ *f* ‘ *set* (*fst xs*)
proof −
obtain *xs′ ko* **where** *xs*: *xs* = (*xs′*, *ko*) **by** *fastforce*
show *?thesis* **by** (*simp add*: *xs map-pair-subset*)
qed

lemma *oalist-inv-map-raw*:
assumes *oalist-inv xs*
and $\bigwedge$*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 −
obtain *xs′ ko* **where** *xs*: *xs* = (*xs′*, *ko*) **by** *fastforce*
from *assms*(*1*) **have** *oalist-inv* (*xs′*, *ko*) **by** (*simp only*: *xs*)
moreover from *assms*(*2*)
have $\bigwedge$*a b. key-compare* (*rep-key-order ko*) (*fst* (*f a*)) (*fst* (*f b*)) = *key-compare* (*rep-key-order ko*) (*fst a*) (*fst b*)
by (*simp add*: *xs*)
ultimately show *?thesis* **unfolding** *xs map-raw.simps oalist-inv-alt* **by** (*rule oalist-inv-raw-map-pair*)
qed

lemma *lookup-raw-map-raw*:
assumes *oalist-inv xs* **and** *snd* (*f* (*k*, *0*)) = *0*
and $\bigwedge$*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 −
obtain *xs′ ko* **where** *xs*: *xs* = (*xs′*, *ko*) **by** *fastforce*
from *assms*(*1*) **have** *oalist-inv* (*xs′*, *ko*) **by** (*simp only*: *xs*)
moreover note *assms*(*2*)
moreover from *assms*(*3*)
have $\bigwedge$*a b. key-compare* (*rep-key-order ko*) (*fst* (*f a*)) (*fst* (*f b*)) = *key-compare* (*rep-key-order ko*) (*fst a*) (*fst b*)
by (*simp add*: *xs*)
ultimately show *?thesis* **unfolding** *xs lookup-raw.simps map-raw.simps oalist-inv-alt*
by (*rule lookup-pair-map-pair*)
qed

lemma *map-raw-id*:
assumes *oalist-inv xs*
shows *map-raw id xs* = *xs*
proof −
obtain *xs′ ko* **where** *xs*: *xs* = (*xs′*, *ko*) **by** *fastforce*
from *assms* **have** *oalist-inv-raw ko xs′* **by** (*simp only*: *xs oalist-inv-alt*)
hence *map-pair id xs′* = *xs′*
proof (*induct xs′ rule*: *oalist-inv-raw-induct*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons k v xs′*)
show *?case* **by** (*simp add*: *Let-def Cons*(*3*, *5*) *id-def*[*symmetric*])
qed

thus *?thesis* **by** (*simp add*: *xs*)
qed

lemma *map-val-raw-cong*:
assumes $\bigwedge k\ v.\ (k,\ v) \in set\ (fst\ xs) \Longrightarrow f\ k\ v = g\ k\ v$
shows *map-val-raw f xs* = *map-val-raw g xs*
proof (*rule map-raw-cong*)
fix *kv*
assume $kv \in set\ (fst\ xs)$
moreover obtain *k v* **where** $kv = (k,\ v)$ **by** *fastforce*
ultimately show (*case kv of* $(k,\ v) \Rightarrow (k,\ f\ k\ v)) = ($*case kv of* $(k,\ v) \Rightarrow (k,\ g\ k\ v))$
by (*simp add*: *assms*)
qed

lemma *oalist-inv-map-val-raw*:
assumes *oalist-inv xs*
shows *oalist-inv* (*map-val-raw f xs*)
proof −
obtain *xs' ko* **where** *xs*: $xs = (xs',\ ko)$ **by** *fastforce*
from *assms* **show** *?thesis* **unfolding** *xs map-raw.simps oalist-inv-alt* **by** (*rule oalist-inv-raw-map-val-pair*)
qed

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 −
obtain *xs' ko* **where** *xs*: $xs = (xs',\ ko)$ **by** *fastforce*
from *assms* **show** *?thesis* **unfolding** *xs map-raw.simps lookup-raw.simps oalist-inv-alt*
by (*rule lookup-pair-map-val-pair*)
qed

12.6.10 *map2-val-raw*

definition *map2-val-compat'* :: $(('a,\ 'b::zero,\ 'o)\ oalist\text{-}raw \Rightarrow ('a,\ 'c::zero,\ 'o)\ oalist\text{-}raw) \Rightarrow bool$
where *map2-val-compat' f* $\longleftrightarrow$
$(\forall zs.\ (oalist\text{-}inv\ zs \longrightarrow (oalist\text{-}inv\ (f\ zs) \wedge snd\ (f\ zs) = snd\ zs \wedge fst\ `\ set\ (fst\ (f\ zs)) \subseteq fst\ `\ set\ (fst\ zs))))$

lemma *map2-val-compat'I*:
assumes $\bigwedge zs.\ oalist\text{-}inv\ zs \Longrightarrow oalist\text{-}inv\ (f\ zs)$
and $\bigwedge zs.\ oalist\text{-}inv\ zs \Longrightarrow snd\ (f\ zs) = snd\ zs$
and $\bigwedge zs.\ oalist\text{-}inv\ zs \Longrightarrow fst\ `\ set\ (fst\ (f\ zs)) \subseteq fst\ `\ set\ (fst\ zs)$
shows *map2-val-compat' f*
unfolding *map2-val-compat'-def* **using** *assms* **by** *blast*

lemma *map2-val-compat′D1*:
assumes *map2-val-compat′ f* **and** *oalist-inv zs*
shows *oalist-inv* (*f zs*)
using *assms* **unfolding** *map2-val-compat′-def* **by** *blast*

lemma *map2-val-compat′D2*:
assumes *map2-val-compat′ f* **and** *oalist-inv zs*
shows *snd* (*f zs*) = *snd zs*
using *assms* **unfolding** *map2-val-compat′-def* **by** *blast*

lemma *map2-val-compat′D3*:
assumes *map2-val-compat′ f* **and** *oalist-inv zs*
shows *fst ‘ set* (*fst* (*f zs*)) ⊆ *fst ‘ set* (*fst zs*)
using *assms* **unfolding** *map2-val-compat′-def* **by** *blast*

lemma *map2-val-compat′-map-val-raw*: *map2-val-compat′* (*map-val-raw f*)
proof (*rule map2-val-compat′I*, *erule oalist-inv-map-val-raw*)
fix *zs*::(′*a*, ′*b*, ′*o*) *oalist-raw*
obtain *zs′ ko* **where** *zs* = (*zs′*, *ko*) **by** *fastforce*
thus *snd* (*map-val-raw f zs*) = *snd zs* **by** *simp*
next
fix *zs*::(′*a*, ′*b*, ′*o*) *oalist-raw*
obtain *zs′ ko* **where** *zs*: *zs* = (*zs′*, *ko*) **by** *fastforce*
show *fst ‘ set* (*fst* (*map-val-raw f zs*)) ⊆ *fst ‘ set* (*fst zs*)
proof (*simp add*: *zs*)
from *map-pair-subset* **have** *fst ‘ set* (*map-val-pair f zs′*) ⊆ *fst ‘* (λ(*k*, *v*). (*k*, *f k v*)) *‘ set zs′*
by (*rule image-mono*)
also have ... = *fst ‘ set zs′* **by** *force*
finally show *fst ‘ set* (*map-val-pair f zs′*) ⊆ *fst ‘ set zs′* **.**
qed
qed

lemma *map2-val-compat′-id*: *map2-val-compat′ id*
by (*rule map2-val-compat′I*, *auto*)

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 (*rule map2-val-compatI*)
fix *zs*::(′*a* × ′*b*) *list*
assume *a*: *oalist-inv-raw ko zs*
hence *oalist-inv* (*zs*, *ko*) **by** (*simp only*: *oalist-inv-alt*)
with *assms* **have** *oalist-inv* (*g* (*zs*, *ko*)) **by** (*rule map2-val-compat′D1*)
hence *oalist-inv* (*fst* (*g* (*zs*, *ko*)), *snd* (*g* (*zs*, *ko*))) **by** *simp*
thus *oalist-inv-raw ko* (*fst* (*g* (*zs*, *ko*))) **using** *assms a* **by** (*simp add*: *oalist-inv-alt map2-val-compat′D2*)
next
fix *zs*::(′*a* × ′*b*) *list*

assume *a*: *oalist-inv-raw ko zs*
hence *oalist-inv* (*zs*, *ko*) **by** (*simp only*: *oalist-inv-alt*)
with *assms* **have** *fst ‘ set* (*fst* (*g* (*zs*, *ko*))) ⊆ *fst ‘ set* (*fst* (*zs*, *ko*)) **by** (*rule map2-val-compat′D3*)
thus *fst ‘ set* (*fst* (*g* (*zs*, *ko*))) ⊆ *fst ‘ set zs* **by** *simp*
qed

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 −
obtain *xs′ ox* **where** *xs*: *xs* = (*xs′*, *ox*) **by** *fastforce*
let *?ys* = *sort-oalist-aux ox ys*
from *assms*(*1*) **have** *oalist-inv-raw ox xs′* **by** (*simp add*: *xs oalist-inv-alt*)
moreover from *assms*(*2*) **have** *oalist-inv-raw ox* (*sort-oalist-aux ox ys*)
by (*rule oalist-inv-raw-sort-oalist-aux*)
moreover from *assms*(*3*) **have** *map2-val-compat ko* (*λzs. fst* (*g* (*zs*, *ko*))) **for** *ko*
by (*rule map2-val-compat′-imp-map2-val-compat*)
moreover from *assms*(*4*) **have** *map2-val-compat ko* (*λzs. fst* (*h* (*zs*, *ko*))) **for** *ko*
by (*rule map2-val-compat′-imp-map2-val-compat*)
ultimately have *oalist-inv-raw ox* (*map2-val-pair ox f* (*λzs. fst* (*g* (*zs*, *ox*))) (*λzs. fst* (*h* (*zs*, *ox*))) *xs′ ?ys*)
by (*rule oalist-inv-raw-map2-val-pair*)
thus *?thesis* **by** (*simp add*: *xs oalist-inv-alt*)
qed

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* (*λk v. f k v 0*) *zs*
and ⋀*zs. oalist-inv zs* ⟹ *h zs* = *map-val-raw* (*λk. 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 −
obtain *xs′ ox* **where** *xs*: *xs* = (*xs′*, *ox*) **by** *fastforce*
let *?ys* = *sort-oalist-aux ox ys*
from *assms*(*1*) **have** *oalist-inv-raw ox xs′* **by** (*simp add*: *xs oalist-inv-alt*)
moreover from *assms*(*2*) **have** *oalist-inv-raw ox* (*sort-oalist-aux ox ys*) **by** (*rule oalist-inv-raw-sort-oalist-aux*)
moreover from *assms*(*3*) **have** *map2-val-compat ko* (*λzs. fst* (*g* (*zs*, *ko*))) **for** *ko*
by (*rule map2-val-compat′-imp-map2-val-compat*)
moreover from *assms*(*4*) **have** *map2-val-compat ko* (*λzs. fst* (*h* (*zs*, *ko*))) **for** *ko*
by (*rule map2-val-compat′-imp-map2-val-compat*)

ultimately have *lookup-pair ox* (*map2-val-pair ox f* (*λzs. fst* (*g* (*zs*, *ox*))) (*λzs. fst* (*h* (*zs*, *ox*))) *xs′ ?ys*) *k0* =
f k0 (*lookup-pair ox xs′ k0*) (*lookup-pair ox ?ys k0*) **using** - - *assms*(*7*)
proof (*rule lookup-pair-map2-val-pair*)
fix *zs*::(*′a* × *′b*) *list*
assume *oalist-inv-raw ox zs*
hence *oalist-inv* (*zs*, *ox*) **by** (*simp only*: *oalist-inv-alt*)
hence *g* (*zs*, *ox*) = *map-val-raw* (*λk v. f k v 0*) (*zs*, *ox*) **by** (*rule assms*(*5*))
thus *fst* (*g* (*zs*, *ox*)) = *map-val-pair* (*λk v. f k v 0*) *zs* **by** *simp*
next
fix *zs*::(*′a* × *′c*) *list*
assume *oalist-inv-raw ox zs*
hence *oalist-inv* (*zs*, *ox*) **by** (*simp only*: *oalist-inv-alt*)
hence *h* (*zs*, *ox*) = *map-val-raw* (*λk. f k 0*) (*zs*, *ox*) **by** (*rule assms*(*6*))
thus *fst* (*h* (*zs*, *ox*)) = *map-val-pair* (*λk. f k 0*) *zs* **by** *simp*
qed
also from *assms*(*2*) **have** ... = *f k0* (*lookup-pair ox xs′ k0*) (*lookup-raw ys k0*)
by (*simp only*: *lookup-pair-sort-oalist-aux*)
finally have ∗: *lookup-pair ox* (*map2-val-pair ox f* (*λzs. fst* (*g* (*zs*, *ox*))) (*λzs. fst* (*h* (*zs*, *ox*))) *xs′ ?ys*) *k0* =
f k0 (*lookup-pair ox xs′ k0*) (*lookup-raw ys k0*) **.**
thus *?thesis* **by** (*simp add*: *xs*)
qed

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* (*λk. f k 0*) ([(*k*, *v*)], *ko*)
shows *map2-val-raw f g h xs* ([(*k*, *v*)], *ko*) = *update-by-fun-raw k* (*λx. f k x v*) *xs*
proof −
obtain *xs′ ox* **where** *xs*: *xs* = (*xs′*, *ox*) **by** *fastforce*
let *?ys* = *sort-oalist ox* [(*k*, *v*)]
from *assms*(*1*) **have** *inv*: *oalist-inv* (*xs′*, *ox*) **by** (*simp only*: *xs*)
hence *inv-raw*: *oalist-inv-raw ox xs′* **by** (*simp only*: *oalist-inv-alt*)
show *?thesis*
proof (*simp add*: *xs*, *intro conjI impI*)
assume *ox* = *ko*
from *inv-raw* **have** *oalist-inv-raw ko xs′* **by** (*simp only*: ‹*ox* = *ko*›)
thus *map2-val-pair ko f* (*λzs. fst* (*g* (*zs*, *ko*))) (*λzs. fst* (*h* (*zs*, *ko*))) *xs′* [(*k*, *v*)] =
update-by-fun-pair ko k (*λx. f k x v*) *xs′*
using *assms*(*2*)
proof (*rule map2-val-pair-singleton-eq-update-by-fun-pair*)
fix *zs*::(*′a* × *′b*) *list*
assume *oalist-inv-raw ko zs*
hence *oalist-inv* (*zs*, *ko*) **by** (*simp only*: *oalist-inv-alt*)
hence *g* (*zs*, *ko*) = (*zs*, *ko*) **by** (*rule assms*(*3*))
thus *fst* (*g* (*zs*, *ko*)) = *zs* **by** *simp*

next
  show *fst* (*h* ([(*k*, *v*)], *ko*)) = *map-val-pair* (λ*k*. *f k 0*) [(*k*, *v*)] **by** (*simp add*: *assms*(*4*))
qed
next
  show *map2-val-pair ox f* (λ*zs*. *fst* (*g* (*zs*, *ox*))) (λ*zs*. *fst* (*h* (*zs*, *ox*))) *xs'* (*sort-oalist ox* [(*k*, *v*)]) =
    update-by-fun-pair ox k (λ*x*. *f k x v*) *xs'*
proof (*cases v* = *0*)
  case *True*
  have *eq1*: *sort-oalist ox* [(*k*, *0*)] = [] **by** (*simp add*: *sort-oalist-def*)
  from *inv* **have** *eq2*: *g* (*xs'*, *ox*) = (*xs'*, *ox*) **by** (*rule assms*(*3*))
  show *?thesis*
  by (*simp add*: *True eq1 eq2 assms*(*2*) *update-by-fun-pair-eq-update-by-pair*[*OF inv-raw*],
    rule HOL.sym, *rule update-by-pair-id*, *fact inv-raw*, *fact HOL.refl*)
next
  case *False*
  hence *oalist-inv-raw ox* [(*k*, *v*)] **by** (*simp add*: *oalist-inv-raw-singleton*)
  hence *eq*: *sort-oalist ox* [(*k*, *v*)] = [(*k*, *v*)] **by** (*rule sort-oalist-id*)
  show *?thesis* **unfolding** *eq* **using** *inv-raw assms*(*2*)
  proof (*rule map2-val-pair-singleton-eq-update-by-fun-pair*)
    fix *zs*::(*'a* × *'b*) *list*
    assume *oalist-inv-raw ox zs*
    hence *oalist-inv* (*zs*, *ox*) **by** (*simp only*: *oalist-inv-alt*)
    hence *g* (*zs*, *ox*) = (*zs*, *ox*) **by** (*rule assms*(*3*))
    thus *fst* (*g* (*zs*, *ox*)) = *zs* **by** *simp*
  next
    show *fst* (*h* ([(*k*, *v*)], *ox*)) = *map-val-pair* (λ*k*. *f k 0*) [(*k*, *v*)] **by** (*simp add*: *assms*(*4*))
  qed
qed
qed
qed

12.6.11 *lex-ord-raw*

lemma *lex-ord-raw-EqI*:
  assumes *oalist-inv xs* **and** *oalist-inv ys*
  and ⋀*k*. *k* ∈ *fst* ' *set* (*fst xs*) ∪ *fst* ' *set* (*fst ys*) ⟹ *f k* (*lookup-raw xs k*) (*lookup-raw ys k*) = *Some Eq*
  shows *lex-ord-raw ko f xs ys* = *Some Eq*
  unfolding *lex-ord-raw-def*
  by (*rule lex-ord-pair-EqI*, *simp-all add*: *assms oalist-inv-raw-sort-oalist-aux lookup-pair-sort-oalist-aux set-sort-oalist-aux*)

lemma *lex-ord-raw-valI*:
  assumes *oalist-inv xs* **and** *oalist-inv ys* **and** *aux* ≠ *Some Eq*
  assumes *k* ∈ *fst* ' *set* (*fst xs*) ∪ *fst* ' *set* (*fst ys*) **and** *aux* = *f k* (*lookup-raw xs*

k) (*lookup-raw ys k*)
  and $\bigwedge k'$. $k' \in$ *fst* ' *set* (*fst xs*) ∪ *fst* ' *set* (*fst ys*) ⟹ *lt ko* k' *k* ⟹
    f k' (*lookup-raw xs* k') (*lookup-raw ys* k') = *Some Eq*
 shows *lex-ord-raw ko f xs ys* = *aux*
 unfolding *lex-ord-raw-def*
 using *oalist-inv-sort-oalist-aux*[*OF assms*(*1*)] *oalist-inv-raw-sort-oalist-aux*[*OF assms*(*2*)] *assms*(*3*)
 unfolding *oalist-inv-alt*
proof (*rule lex-ord-pair-valI*)
 from *assms*(*1*, *2*, *4*) **show** *k* ∈ *fst* ' *set* (*sort-oalist-aux ko xs*) ∪ *fst* ' *set* (*sort-oalist-aux ko ys*)
  by (*simp add*: *set-sort-oalist-aux*)
next
 from *assms*(*1*, *2*, *5*) **show** *aux* = *f k* (*lookup-pair ko* (*sort-oalist-aux ko xs*) *k*) (*lookup-pair ko* (*sort-oalist-aux ko ys*) *k*)
  by (*simp add*: *lookup-pair-sort-oalist-aux*)
next
 fix k'
 assume $k' \in$ *fst* ' *set* (*sort-oalist-aux ko xs*) ∪ *fst* ' *set* (*sort-oalist-aux ko ys*)
 with *assms*(*1*, *2*) **have** $k' \in$ *fst* ' *set* (*fst xs*) ∪ *fst* ' *set* (*fst ys*) **by** (*simp add*: *set-sort-oalist-aux*)
 moreover assume *lt ko* k' *k*
 ultimately have *f* k' (*lookup-raw xs* k') (*lookup-raw ys* k') = *Some Eq* **by** (*rule assms*(*6*))
 with *assms*(*1*, *2*) **show** *f* k' (*lookup-pair ko* (*sort-oalist-aux ko xs*) k') (*lookup-pair ko* (*sort-oalist-aux ko ys*) k') = *Some Eq*
  by (*simp add*: *lookup-pair-sort-oalist-aux*)
qed

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* ∈ *fst* ' *set* (*fst xs*) ∪ *fst* ' *set* (*fst ys*)
 shows *f k* (*lookup-raw xs k*) (*lookup-raw ys k*) = *Some Eq*
proof −
 have *f k* (*lookup-pair ko* (*sort-oalist-aux ko xs*) *k*) (*lookup-pair ko* (*sort-oalist-aux ko ys*) *k*) = *Some Eq*
  by (*rule lex-ord-pair-EqD*[**where** *f*=*f*],
   simp-all add: *oalist-inv-raw-sort-oalist-aux assms lex-ord-raw-def*[*symmetric*] *set-sort-oalist-aux del*: *Un-iff*)
 with *assms*(*1*, *2*) **show** *?thesis* **by** (*simp add*: *lookup-pair-sort-oalist-aux*)
qed

lemma *lex-ord-raw-valE*:
 assumes *oalist-inv xs* **and** *oalist-inv ys* **and** *lex-ord-raw ko f xs ys* = *aux*
  and *aux* ≠ *Some Eq*
 obtains *k* **where** *k* ∈ *fst* ' *set* (*fst xs*) ∪ *fst* ' *set* (*fst ys*)
  and *aux* = *f k* (*lookup-raw xs k*) (*lookup-raw ys k*)
  and $\bigwedge k'$. $k' \in$ *fst* ' *set* (*fst xs*) ∪ *fst* ' *set* (*fst ys*) ⟹ *lt ko* k' *k* ⟹
    f k' (*lookup-raw xs* k') (*lookup-raw ys* k') = *Some Eq*

proof −
  let *?xs* = *sort-oalist-aux ko xs*
  let *?ys* = *sort-oalist-aux ko ys*
  from *assms*(*3*) **have** *lex-ord-pair ko f ?xs ?ys* = *aux* **by** (*simp only*: *lex-ord-raw-def*)
  with *oalist-inv-sort-oalist-aux*[*OF assms*(*1*)] *oalist-inv-sort-oalist-aux*[*OF assms*(*2*)]
  obtain *k* **where** *a*: $k \in fst$ ‘ *set ?xs* ∪ *fst* ‘ *set ?ys*
    and *b*: *aux* = *f k* (*lookup-pair ko ?xs k*) (*lookup-pair ko ?ys k*)
    and *c*: $\bigwedge k'$. $k' \in fst$ ‘ *set ?xs* ∪ *fst* ‘ *set ?ys* ⟹ *lt ko k' k* ⟹
        f k' (*lookup-pair ko ?xs k'*) (*lookup-pair ko ?ys k'*) = *Some Eq*
    using *assms*(*4*) **unfolding** *oalist-inv-alt* **by** (*rule lex-ord-pair-valE*, *blast*)
  from *a* **have** $k \in fst$ ‘ *set* (*fst xs*) ∪ *fst* ‘ *set* (*fst ys*)
    by (*simp add*: *set-sort-oalist-aux assms*(*1*, *2*))
  moreover from *b* **have** *aux* = *f k* (*lookup-raw xs k*) (*lookup-raw ys k*)
    by (*simp add*: *lookup-pair-sort-oalist-aux assms*(*1*, *2*))
  moreover have *f k'* (*lookup-raw xs k'*) (*lookup-raw ys k'*) = *Some Eq*
    if *k'-in*: $k' \in fst$ ‘ *set* (*fst xs*) ∪ *fst* ‘ *set* (*fst ys*) **and** *k'-less*: *lt ko k' k* **for** *k'*
  proof −
    have *f k'* (*lookup-raw xs k'*) (*lookup-raw ys k'*) = *f k'* (*lookup-pair ko ?xs k'*) (*lookup-pair ko ?ys k'*)
      by (*simp add*: *lookup-pair-sort-oalist-aux assms*(*1*, *2*))
    also have ... = *Some Eq*
    proof (*rule c*)
      from *k'-in* **show** $k' \in fst$ ‘ *set ?xs* ∪ *fst* ‘ *set ?ys*
        by (*simp add*: *set-sort-oalist-aux assms*(*1*, *2*))
    next
      from *k'-less* **show** *lt ko k' k* **by** (*simp only*: *lt-of-key-order.rep-eq*)
    qed
    finally show *?thesis* **.**
  qed
  ultimately show *?thesis* **..**
qed

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*) ∪ *fst* ‘ *set* (*fst ys*) ⟹ *P k* (*lookup-raw xs k*) (*lookup-raw ys k*)
  shows *prod-ord-raw P xs ys*
proof −
  obtain *xs' ox* **where** *xs*: *xs* = (*xs'*, *ox*) **by** *fastforce*
  from *assms*(*1*) **have** *oalist-inv-raw ox xs'* **by** (*simp only*: *xs oalist-inv-alt*)
  hence ∗: *prod-ord-pair ox P xs'* (*sort-oalist-aux ox ys*) **using** *oalist-inv-raw-sort-oalist-aux*
  proof (*rule prod-ord-pairI*)
    fix *k*
    assume $k \in fst$ ‘ *set xs'* ∪ *fst* ‘ *set* (*sort-oalist-aux ox ys*)
    hence $k \in fst$ ‘ *set* (*fst xs*) ∪ *fst* ‘ *set* (*fst ys*) **by** (*simp add*: *xs set-sort-oalist-aux assms*(*2*))
    hence *P k* (*lookup-raw xs k*) (*lookup-raw ys k*) **by** (*rule assms*(*3*))

thus *P k* (*lookup-pair ox xs′ k*) (*lookup-pair ox* (*sort-oalist-aux ox ys*) *k*)
by (*simp add*: *xs lookup-pair-sort-oalist-aux assms*(*2*))
qed *fact*
thus *?thesis* **by** (*simp add*: *xs*)
qed

lemma *prod-ord-rawD*:
assumes *oalist-inv xs* **and** *oalist-inv ys* **and** *prod-ord-raw P xs ys*
and *k* ∈ *fst* ‘ *set* (*fst xs*) ∪ *fst* ‘ *set* (*fst ys*)
shows *P k* (*lookup-raw xs k*) (*lookup-raw ys k*)
proof −
obtain *xs′ ox* **where** *xs*: *xs* = (*xs′*, *ox*) **by** *fastforce*
from *assms*(*1*) **have** *oalist-inv-raw ox xs′* **by** (*simp only*: *xs oalist-inv-alt*)
moreover note *oalist-inv-raw-sort-oalist-aux*[*OF assms*(*2*)]
moreover from *assms*(*3*) **have** *prod-ord-pair ox P xs′* (*sort-oalist-aux ox ys*) **by** (*simp add*: *xs*)
moreover from *assms*(*4*) **have** *k* ∈ *fst* ‘ *set xs′* ∪ *fst* ‘ *set* (*sort-oalist-aux ox ys*)
by (*simp add*: *xs set-sort-oalist-aux assms*(*2*))
ultimately have ∗: *P k* (*lookup-pair ox xs′ k*) (*lookup-pair ox* (*sort-oalist-aux ox ys*) *k*)
by (*rule prod-ord-pairD*)
thus *?thesis* **by** (*simp add*: *xs lookup-pair-sort-oalist-aux assms*(*2*))
qed

corollary *prod-ord-raw-alt*:
assumes *oalist-inv xs* **and** *oalist-inv ys*
shows *prod-ord-raw P xs ys* ⟷
(∀ *k*∈*fst* ‘ *set* (*fst xs*) ∪ *fst* ‘ *set* (*fst ys*). *P k* (*lookup-raw xs k*) (*lookup-raw ys k*))
using *prod-ord-rawI*[*OF assms*] *prod-ord-rawD*[*OF assms*] **by** *meson*

12.6.13 *oalist-eq-raw*

lemma *oalist-eq-rawI*:
assumes *oalist-inv xs* **and** *oalist-inv ys*
and ⋀*k*. *k* ∈ *fst* ‘ *set* (*fst xs*) ∪ *fst* ‘ *set* (*fst ys*) ⟹ *lookup-raw xs k* = *lookup-raw ys k*
shows *oalist-eq-raw xs ys*
proof −
obtain *xs′ ox* **where** *xs*: *xs* = (*xs′*, *ox*) **by** *fastforce*
from *assms*(*1*) **have** *oalist-inv-raw ox xs′* **by** (*simp only*: *xs oalist-inv-alt*)
hence ∗: *xs′* = *sort-oalist-aux ox ys* **using** *oalist-inv-raw-sort-oalist-aux*[*OF assms*(*2*)]
proof (*rule lookup-pair-inj*)
show *lookup-pair ox xs′* = *lookup-pair ox* (*sort-oalist-aux ox ys*)
proof
fix *k*
show *lookup-pair ox xs′ k* = *lookup-pair ox* (*sort-oalist-aux ox ys*) *k*
proof (*cases k* ∈ *fst* ‘ *set xs′* ∪ *fst* ‘ *set* (*sort-oalist-aux ox ys*))

case *True*
hence $k \in$ *fst* ' *set* (*fst xs*) $\cup$ *fst* ' *set* (*fst ys*) **by** (*simp add*: *xs set-sort-oalist-aux assms*(*2*))
hence *lookup-raw xs k* = *lookup-raw ys k* **by** (*rule assms*(*3*))
thus *?thesis* **by** (*simp add*: *xs lookup-pair-sort-oalist-aux assms*(*2*))
next
case *False*
hence $k \notin$ *fst* ' *set xs'* **and** $k \notin$ *fst* ' *set* (*sort-oalist-aux ox ys*) **by** *simp-all*
with ‹*oalist-inv-raw ox xs'*› *oalist-inv-raw-sort-oalist-aux*[*OF assms*(*2*)]
have *lookup-pair ox xs' k* = *0* **and** *lookup-pair ox* (*sort-oalist-aux ox ys*) *k* = *0*
by (*simp-all add*: *lookup-pair-eq-0*)
thus *?thesis* **by** *simp*
qed
qed
qed
thus *?thesis* **by** (*simp add*: *xs*)
qed

lemma *oalist-eq-rawD*:
assumes *oalist-inv ys* **and** *oalist-eq-raw xs ys*
shows *lookup-raw xs* = *lookup-raw ys*
proof –
obtain *xs' ox* **where** *xs*: *xs* = (*xs'*, *ox*) **by** *fastforce*
from *assms*(*2*) **have** *xs'* = *sort-oalist-aux ox ys* **by** (*simp add*: *xs*)
hence *lookup-pair ox xs'* = *lookup-pair ox* (*sort-oalist-aux ox ys*) **by** *simp*
thus *?thesis* **by** (*simp add*: *xs lookup-pair-sort-oalist-aux assms*(*1*))
qed

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*)
using *oalist-eq-rawI*[*OF assms*] *oalist-eq-rawD*[*OF assms*(*2*)] **by** *metis*

12.6.14 *sort-oalist-raw*

lemma *oalist-inv-sort-oalist-raw*: *oalist-inv* (*sort-oalist-raw xs*)
proof –
obtain *xs' ko* **where** *xs*: *xs* = (*xs'*, *ko*) **by** *fastforce*
show *?thesis* **by** (*simp add*: *xs oalist-inv-raw-sort-oalist oalist-inv-alt*)
qed

lemma *sort-oalist-raw-id*:
assumes *oalist-inv xs*
shows *sort-oalist-raw xs* = *xs*
proof –
obtain *xs' ko* **where** *xs*: *xs* = (*xs'*, *ko*) **by** *fastforce*
from *assms* **have** *oalist-inv-raw ko xs'* **by** (*simp only*: *xs oalist-inv-alt*)
hence *sort-oalist ko xs'* = *xs'* **by** (*rule sort-oalist-id*)

thus *?thesis* **by** (*simp add*: *xs*)
qed

lemma *set-sort-oalist-raw*:
assumes *distinct* (*map fst* (*fst xs*))
shows *set* (*fst* (*sort-oalist-raw xs*)) = {*kv*. *kv* ∈ *set* (*fst xs*) ∧ *snd kv* ≠ *0*}
proof −
obtain *xs′ ko* **where** *xs*: *xs* = (*xs′*, *ko*) **by** *fastforce*
from *assms* **have** *distinct* (*map fst xs′*) **by** (*simp add*: *xs*)
hence *set* (*sort-oalist ko xs′*) = {*kv* ∈ *set xs′*. *snd kv* ≠ *0*} **by** (*rule set-sort-oalist*)
thus *?thesis* **by** (*simp add*: *xs*)
qed

end

12.7 Fundamental Operations on One List

locale *oalist-abstract* = *oalist-raw rep-key-order* **for** *rep-key-order*::*′o* ⇒ *′a key-order*
+
fixes *list-of-oalist* :: *′x* ⇒ (*′a*, *′b*::*zero*, *′o*) *oalist-raw*
fixes *oalist-of-list* :: (*′a*, *′b*, *′o*) *oalist-raw* ⇒ *′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 −
obtain *xs′ ox* **where** *xs*: *xs* = (*xs′*, *ox*) **by** *fastforce*
from *assms* **show** *?thesis* **by** (*simp add*: *xs list-of-oalist-of-list sort-oalist-id oal-ist-inv-alt*)
qed

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* ∉ *snd* ‘ *set* (*fst* (*list-of-oalist xs*))
proof −
from *oalist-inv-list-of-oalist* **have** *oalist-inv-raw* (*snd* (*list-of-oalist xs*)) (*fst* (*list-of-oalist xs*))
by (*simp only*: *oalist-inv-def*)
thus *?thesis* **by** (*rule oalist-inv-rawD1*)
qed

lemma *list-of-oalist-sorted*: *sorted-wrt* (*lt* (*snd* (*list-of-oalist xs*))) (*map fst* (*fst* (*list-of-oalist xs*)))
proof −
from *oalist-inv-list-of-oalist* **have** *oalist-inv-raw* (*snd* (*list-of-oalist xs*)) (*fst* (*list-of-oalist xs*))
by (*simp only*: *oalist-inv-def*)
thus *?thesis* **by** (*rule oalist-inv-rawD2*)
qed

12.7.2 *lookup*

lemma *lookup-eq-value*: $v \neq 0 \Longrightarrow$ *lookup xs k* $= v \longleftrightarrow ((k, v) \in$ *set* (*fst* (*list-of-oalist xs*)))
unfolding *lookup-def* **using** *oalist-inv-list-of-oalist* **by** (*rule lookup-raw-eq-value*)

lemma *lookup-eq-valueI*: $(k, v) \in$ *set* (*fst* (*list-of-oalist xs*)) $\Longrightarrow$ *lookup xs k* $= v$
unfolding *lookup-def* **using** *oalist-inv-list-of-oalist* **by** (*rule lookup-raw-eq-valueI*)

lemma *lookup-oalist-of-list*:
distinct (*map fst xs*) $\Longrightarrow$ *lookup* (*oalist-of-list* (*xs*, *ko*)) = *lookup-dflt xs*
by (*simp add*: *list-of-oalist-of-list lookup-def lookup-pair-sort-oalist′*)

12.7.3 *sorted-domain*

lemma *set-sorted-domain*: *set* (*sorted-domain ko xs*) = *fst* ‘ *set* (*fst* (*list-of-oalist xs*))
unfolding *sorted-domain-def* **using** *oalist-inv-list-of-oalist* **by** (*rule set-sorted-domain-raw*)

lemma *in-sorted-domain-iff-lookup*: $k \in$ *set* (*sorted-domain ko xs*) $\longleftrightarrow$ (*lookup xs k* $\neq 0$)
unfolding *sorted-domain-def lookup-def* **using** *oalist-inv-list-of-oalist*
by (*rule in-sorted-domain-raw-iff-lookup-raw*)

lemma *sorted-sorted-domain*: *sorted-wrt* (*lt ko*) (*sorted-domain ko xs*)
unfolding *sorted-domain-def lt-of-key-order.rep-eq*[*symmetric*]
using *oalist-inv-list-of-oalist* **by** (*rule sorted-sorted-domain-raw*)

12.7.4 *local.empty* and Singletons

lemma *list-of-oalist-empty* [*simp*, *code abstract*]: *list-of-oalist* (*empty ko*) = ([], *ko*)
by (*simp add*: *empty-def sort-oalist-def list-of-oalist-of-list*)

lemma *lookup-empty*: *lookup* (*empty ko*) $k = 0$
by (*simp add*: *lookup-def*)

lemma *lookup-oalist-of-list-single*:
lookup (*oalist-of-list* ([(*k*, *v*)], *ko*)) k' = (*if* $k = k'$ *then v else 0*)
by (*simp add*: *lookup-def list-of-oalist-of-list sort-oalist-def key-compare-Eq split*: *order.split*)

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*)
unfolding *reorder-def*
by (*rule list-of-oalist-of-list-id*, *simp add*: *oalist-inv-def*, *rule oalist-inv-raw-sort-oalist-aux*, *fact oalist-inv-list-of-oalist*)

lemma *lookup-reorder*: *lookup* (*reorder ko xs*) *k* = *lookup xs k*

by (*simp add*: *lookup-def lookup-pair-sort-oalist-aux oalist-inv-list-of-oalist*)

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*)
unfolding *tl-def*
by (*rule list-of-oalist-of-list-id*, *rule oalist-inv-tl-raw*, *fact oalist-inv-list-of-oalist*)

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*)
by (*simp add*: *lookup-def lookup-raw-tl-raw oalist-inv-list-of-oalist*)

lemma *hd-in*:
assumes *fst* (*list-of-oalist xs*) ≠ []
shows *hd xs* ∈ *set* (*fst* (*list-of-oalist xs*))
unfolding *hd-def* **using** *assms* **by** (*rule hd-in-set*)

lemma *snd-hd*:
assumes *fst* (*list-of-oalist xs*) ≠ []
shows *snd* (*hd xs*) = *lookup xs* (*fst* (*hd xs*))
proof −
from *assms* **have** ∗: *hd xs* ∈ *set* (*fst* (*list-of-oalist xs*)) **by** (*rule hd-in*)
show *?thesis* **by** (*rule HOL.sym*, *rule lookup-eq-valueI*, *simp add*: ∗)
qed

lemma *lookup-tl'*: *lookup* (*tl xs*) *k* = (*if k* = *fst* (*hd xs*) *then 0 else lookup xs k*)
by (*simp add*: *lookup-def lookup-raw-tl-raw' oalist-inv-list-of-oalist hd-def*)

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 −
obtain *xs' ko* **where** *xs*: *list-of-oalist xs* = (*xs'*, *ko*) **by** *fastforce*
from *assms* **obtain** *x xs''* **where** *xs'*: *xs'* = *x* # *xs''* **unfolding** *xs fst-conv* **using** *list.exhaust* **by** *blast*
show *?thesis* **by** (*simp add*: *xs xs' hd-def*)
qed

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*)
using *assms oalist-inv-list-of-oalist* **by** (*simp add*: *min-key-val-def hd-def min-key-val-raw-alt*)

lemma *min-key-val-in*:

assumes *fst* (*list-of-oalist xs*) ≠ []
shows *min-key-val ko xs* ∈ *set* (*fst* (*list-of-oalist xs*))
unfolding *min-key-val-def* **using** *assms* **by** (*rule min-key-val-raw-in*)

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*))
unfolding *lookup-def min-key-val-def* **using** *oalist-inv-list-of-oalist assms* **by** (*rule snd-min-key-val-raw*)

lemma *min-key-val-minimal*:
assumes *z* ∈ *set* (*fst* (*list-of-oalist xs*))
shows *le ko* (*fst* (*min-key-val ko xs*)) (*fst z*)
unfolding *min-key-val-def*
by (*rule min-key-val-raw-minimal*, *fact oalist-inv-list-of-oalist*, *fact*)

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*)
by (*simp add*: *except-min-def*)

lemma *except-min-Nil*:
assumes *fst* (*list-of-oalist xs*) = []
shows *fst* (*list-of-oalist* (*except-min ko xs*)) = []
proof −
obtain *xs′ ox* **where** *eq*: *list-of-oalist xs* = (*xs′*, *ox*) **by** *fastforce*
from *assms* **have** *xs′* = [] **by** (*simp add*: *eq*)
show *?thesis* **by** (*simp add*: *eq* ‹*xs′* = []› *sort-oalist-def*)
qed

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*)
by (*simp add*: *except-min-def lookup-tl set-sort-oalist-aux oalist-inv-list-of-oalist lookup-reorder*)

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 (*cases fst* (*list-of-oalist xs*) = [])
case *True*
hence *lookup xs k* = *0* **by** (*metis empty-def lookup-empty oalist-of-list-of-oalist prod.collapse*)
thus *?thesis* **by** (*simp add*: *lookup-except-min True*)
next
case *False*
thus *?thesis* **by** (*simp add*: *except-min-def lookup-tl′ min-key-val-alt lookup-reorder*)

qed

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*)
unfolding *insert-def*
by (*rule list-of-oalist-of-list-id*, *rule oalist-inv-update-by-raw*, *fact oalist-inv-list-of-oalist*)

lemma *lookup-insert*: *lookup* (*insert* (*k*, *v*) *xs*) *k'* = (*if k* = *k' then v else lookup xs k'*)
by (*simp add*: *lookup-def lookup-raw-update-by-raw oalist-inv-list-of-oalist split del*: *if-split cong*: *if-cong*)

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*)
unfolding *update-by-fun-def*
by (*rule list-of-oalist-of-list-id*, *rule oalist-inv-update-by-fun-raw*, *fact oalist-inv-list-of-oalist*)

lemma *lookup-update-by-fun*:
lookup (*update-by-fun k f xs*) *k'* = (*if k* = *k' then f else id*) (*lookup xs k'*)
by (*simp add*: *lookup-def lookup-raw-update-by-fun-raw oalist-inv-list-of-oalist split del*: *if-split cong*: *if-cong*)

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*)
unfolding *update-by-fun-gr-def*
by (*rule list-of-oalist-of-list-id*, *rule oalist-inv-update-by-fun-gr-raw*, *fact oalist-inv-list-of-oalist*)

lemma *update-by-fun-gr-eq-update-by-fun*: *update-by-fun-gr* = *update-by-fun*
by (*rule*, *rule*, *rule*,
simp add: *update-by-fun-gr-def update-by-fun-def update-by-fun-gr-raw-eq-update-by-fun-raw oalist-inv-list-of-oalist*)

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*)
unfolding *filter-def*
by (*rule list-of-oalist-of-list-id*, *rule oalist-inv-filter-raw*, *fact oalist-inv-list-of-oalist*)

lemma *lookup-filter*: *lookup* (*filter P xs*) *k* = (*let v* = *lookup xs k in if P* (*k*, *v*) *then v else 0*)
by (*simp add*: *lookup-def lookup-raw-filter-raw oalist-inv-list-of-oalist*)

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*)
unfolding *map2-val-neutr-def*
by (*rule list-of-oalist-of-list-id*, *rule oalist-inv-map2-val-raw*,
fact oalist-inv-list-of-oalist, *fact oalist-inv-list-of-oalist*,
fact map2-val-compat′-id, *fact map2-val-compat′-id*)

lemma *lookup-map2-val-neutr*:
assumes $\bigwedge k\ x.\ f\ k\ x\ 0 = x$ **and** $\bigwedge k\ x.\ f\ k\ 0\ x = x$
shows *lookup* (*map2-val-neutr f xs ys*) *k* = *f k* (*lookup xs k*) (*lookup ys k*)
proof (*simp add*: *lookup-def*, *rule lookup-raw-map2-val-raw*)
fix *zs*::(′*a*, ′*b*, ′*o*) *oalist-raw*
assume *oalist-inv zs*
thus *id zs* = *map-val-raw* (λ*k v*. *f k v 0*) *zs* **by** (*simp add*: *assms*(*1*) *map-raw-id*)
next
fix *zs*::(′*a*, ′*b*, ′*o*) *oalist-raw*
assume *oalist-inv zs*
thus *id zs* = *map-val-raw* (λ*k*. *f k 0*) *zs* **by** (*simp add*: *assms*(*2*) *map-raw-id*)
qed (*fact oalist-inv-list-of-oalist*, *fact oalist-inv-list-of-oalist*,
fact map2-val-compat′-id, *fact map2-val-compat′-id*, *simp only*: *assms*(*1*))

12.7.13 *oalist-eq*

lemma *oalist-eq-alt*: *oalist-eq xs ys* ⟷ (*lookup xs* = *lookup ys*)
by (*simp add*: *oalist-eq-def lookup-def oalist-eq-raw-alt oalist-inv-list-of-oalist*)

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* (*λk. 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*
unfolding *map-val-def* **by** (*rule arg-cong*[**where** *f*=*oalist-of-listy*], *rule map-val-raw-cong*, *elim assms*)

lemma *list-of-oalist-map-val* [*simp*, *code abstract*]:
list-of-oalisty (*map-val f xs*) = *map-val-raw f* (*list-of-oalistx xs*)
unfolding *map-val-def*
by (*rule oay.list-of-oalist-of-list-id*, *rule oalist-inv-map-val-raw*, *fact oalist-inv-list-of-oalist*)

lemma *lookup-map-val*: *f k 0* = *0* ⟹ *oay.lookup* (*map-val f xs*) *k* = *f k* (*lookup xs k*)
by (*simp add*: *oay.lookup-def lookup-def lookup-raw-map-val-raw oalist-inv-list-of-oalist*)

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* (*λk b. f k b 0*)) (*map-val-raw* (*λk. f k 0*)) (*list-of-oalistx xs*) (*list-of-oalisty ys*)
unfolding *map2-val-def*
by (*rule oaz.list-of-oalist-of-list-id*, *rule oalist-inv-map2-val-raw*,
fact oalist-inv-list-of-oalist, *fact oay.oalist-inv-list-of-oalist*,
fact map2-val-compat'-map-val-raw, *fact map2-val-compat'-map-val-raw*)

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* (*λk c. f k 0 c*)) (*list-of-oalistx xs*) (*list-of-oalisty ys*)
unfolding *map2-val-rneutr-def*
by (*rule list-of-oalist-of-list-id*, *rule oalist-inv-map2-val-raw*,
fact oalist-inv-list-of-oalist, *fact oay.oalist-inv-list-of-oalist*,
fact map2-val-compat'-id, *fact map2-val-compat'-map-val-raw*)

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*)
by (*simp add*: *oaz.lookup-def oay.lookup-def lookup-def lookup-raw-map2-val-raw map2-val-compat′-map-val-raw assms oalist-inv-list-of-oalist oay.oalist-inv-list-of-oalist*)

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 (*simp add*: *lookup-def oay.lookup-def*, *rule lookup-raw-map2-val-raw*)
fix *zs*::(′*a*, ′*b*, ′*o*) *oalist-raw*
assume *oalist-inv zs*
thus *id zs* = *map-val-raw* (λ*k v*. *f k v 0*) *zs* **by** (*simp add*: *assms map-raw-id*)
qed (*fact oalist-inv-list-of-oalist*, *fact oay.oalist-inv-list-of-oalist*,
fact map2-val-compat′-id, *fact map2-val-compat′-map-val-raw*, *rule HOL.refl*,
simp only: *assms*)

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* (λ*x*. *f k x v*) *xs*
by (*simp add*: *map2-val-rneutr-def update-by-fun-def assms map2-val-raw-singleton-eq-update-by-fun-raw oalist-inv-list-of-oalist*)

12.8.3 *lex-ord* and *prod-ord*

lemma *lex-ord-EqI*:
($\bigwedge k.$ *k* ∈ *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*
by (*simp add*: *lex-ord-def lookup-def oay.lookup-def*, *rule lex-ord-raw-EqI*,
rule oalist-inv-list-of-oalist, *rule oay.oalist-inv-list-of-oalist*, *blast*)

lemma *lex-ord-valI*:
assumes *aux* ≠ *Some Eq* **and** *k* ∈ *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*′ ∈ *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*
by (*simp* (*no-asm-use*) *add*: *lex-ord-def lookup-def oay.lookup-def*, *rule lex-ord-raw-valI*,
rule oalist-inv-list-of-oalist, *rule oay.oalist-inv-list-of-oalist*, *rule assms*(*1*), *rule assms*(*2*), *blast*+)

lemma *lex-ord-EqD*:
lex-ord ko f xs ys = *Some Eq* ⟹
k ∈ *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*
by (*simp add*: *lex-ord-def lookup-def oay.lookup-def*, *rule lex-ord-raw-EqD*[**where**

f=*f*],
  rule oalist-inv-list-of-oalist, *rule oay.oalist-inv-list-of-oalist*, *assumption*, *simp*)

lemma *lex-ord-valE*:
  assumes *lex-ord ko f xs ys* = *aux* **and** *aux* ≠ *Some Eq*
  obtains *k* **where** *k* ∈ *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 ⋀*k'*. *k'* ∈ *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 −
  note *oalist-inv-list-of-oalist oay.oalist-inv-list-of-oalist*
  moreover from *assms*(*1*) **have** *lex-ord-raw ko f* (*list-of-oalistx xs*) (*list-of-oalisty ys*) = *aux*
    by (*simp only*: *lex-ord-def*)
  ultimately obtain *k* **where** *1*: *k* ∈ *fst* ‘ *set* (*fst* (*list-of-oalistx xs*)) ∪ *fst* ‘ *set* (*fst* (*list-of-oalisty ys*))
    and *aux* = *f k* (*lookup-raw* (*list-of-oalistx xs*) *k*) (*lookup-raw* (*list-of-oalisty ys*) *k*)
    and ⋀*k'*. *k'* ∈ *fst* ‘ *set* (*fst* (*list-of-oalistx xs*)) ∪ *fst* ‘ *set* (*fst* (*list-of-oalisty ys*)) ⟹
      lt ko k' k ⟹
      f k' (*lookup-raw* (*list-of-oalistx xs*) *k'*) (*lookup-raw* (*list-of-oalisty ys*) *k'*) = *Some Eq*
    using *assms*(*2*) **by** (*rule lex-ord-raw-valE*, *blast*)
  from *this*(*2*, *3*) **have** *aux* = *f k* (*lookup xs k*) (*oay.lookup ys k*)
    and ⋀*k'*. *k'* ∈ *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*
    by (*simp-all only*: *lookup-def oay.lookup-def*)
  with *1* **show** *?thesis* **..**
qed

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*))
  by (*simp add*: *prod-ord-def lookup-def oay.lookup-def prod-ord-raw-alt oalist-inv-list-of-oalist oay.oalist-inv-list-of-oalist*)

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*
by (*fact comparator-key-compare*)

lemma *ko-le*: *ko.le* = *le-of-key-order*
by (*intro ext*, *simp add*: *le-of-comp-def le-of-key-order-alt split*: *order.split*)

global-interpretation *ko*: *oalist-raw* $\lambda x.\ x$
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*
subgoal by (*simp only*: *lookup-pair-ko-def*)
subgoal by (*simp only*: *update-by-pair-ko-def*)
subgoal by (*simp only*: *update-by-fun-pair-ko-def*)
subgoal by (*simp only*: *update-by-fun-gr-pair-ko-def*)
subgoal by (*simp only*: *map2-val-pair-ko-def*)
subgoal by (*simp only*: *lex-ord-pair-ko-def*)
subgoal by (*simp only*: *prod-ord-pair-ko-def*)
subgoal by (*simp only*: *sort-oalist-ko′-def*)
done

typedef (**overloaded**) (′*a*, ′*b*) *oalist* = {*xs*::(′*a*, ′*b*::*zero*, ′*a key-order*) *oalist-raw*. *ko.oalist-inv xs*}
morphisms *list-of-oalist Abs-oalist*
by (*auto simp*: *ko.oalist-inv-def intro*: *ko.oalist-inv-raw-Nil*)

lemma *oalist-eq-iff*: *xs* = *ys* ⟷ *list-of-oalist xs* = *list-of-oalist ys*
  by (*simp add*: *list-of-oalist-inject*)

lemma *oalist-eqI*: *list-of-oalist xs* = *list-of-oalist ys* ⟹ *xs* = *ys*
  by (*simp add*: *oalist-eq-iff*)

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*)
  using *list-of-oalist* [*of xs*] **by** *simp*

lemma *list-of-oalist-OAlist*: *list-of-oalist* (*OAlist xs*) = *sort-oalist-ko xs*
proof −
  obtain *xs' ox* **where** *xs*: *xs* = (*xs'*, *ox*) **by** *fastforce*
  show *?thesis* **by** (*simp add*: *xs OAlist-def Abs-oalist-inverse ko.oalist-inv-raw-sort-oalist ko.oalist-inv-alt*)
qed

lemma *OAlist-list-of-oalist* [*code abstype*]: *OAlist* (*list-of-oalist xs*) = *xs*
proof −
  obtain *xs' ox* **where** *xs*: *list-of-oalist xs* = (*xs'*, *ox*) **by** *fastforce*
  have *ko.oalist-inv-raw ox xs'* **by** (*simp add*: *xs*[*symmetric*] *ko.oalist-inv-alt*[*symmetric*] *oalist-inv-list-of-oalist*)
  thus *?thesis* **by** (*simp add*: *xs OAlist-def ko.sort-oalist-id*, *simp add*: *list-of-oalist-inverse xs*[*symmetric*])
qed

lemma [*code abstract*]: *list-of-oalist* (*oalist-of-list xs*) = *sort-oalist-ko xs*
  by (*simp add*: *list-of-oalist-OAlist oalist-of-list-def*)

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*

apply *standard*
subgoal by (*fact oalist-inv-list-of-oalist*)
subgoal by (*simp only*: *list-of-oalist-OAlist sort-oalist-ko-def*)
subgoal by (*fact OAlist-list-of-oalist*)
done

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* **..**

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*
by (*fact comparator-of*)

lemma *tc-le-lt* [*simp*]: *tc.le* = (≤) *tc.lt* = (<)
by (*auto simp*: *le-of-comp-def lt-of-comp-def comparator-of-def intro*!: *ext split*: *order.split-asm if-split-asm*)

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*
by (*auto intro*: *tc.oalist-inv-raw-Nil*)

lemma *oalist-tc-eq-iff*: *xs* = *ys* ⟷ *list-of-oalist-tc xs* = *list-of-oalist-tc ys*
by (*simp add*: *list-of-oalist-tc-inject*)

lemma *oalist-tc-eqI*: *list-of-oalist-tc xs* = *list-of-oalist-tc ys* ⟹ *xs* = *ys*
by (*simp add*: *oalist-tc-eq-iff*)

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*)
using *list-of-oalist-tc*[*of xs*] **by** *simp*

lemma *list-of-oalist-OAlist-tc*: *list-of-oalist-tc* (*OAlist-tc xs*) = *sort-oalist-tc xs*
by (*simp add*: *OAlist-tc-def Abs-oalist-tc-inverse tc.oalist-inv-raw-sort-oalist*)

lemma *OAlist-list-of-oalist-tc* [*code abstype*]: *OAlist-tc* (*list-of-oalist-tc xs*) = *xs*
by (*simp add*: *OAlist-tc-def tc.sort-oalist-id list-of-oalist-tc-inverse oalist-inv-list-of-oalist-tc*)

lemma *list-of-oalist-tc-of-list* [*code abstract*]: *list-of-oalist-tc* (*oalist-tc-of-list xs*) = *sort-oalist-tc xs*
by (*simp add*: *list-of-oalist-OAlist-tc oalist-tc-of-list-def*)

lemma *list-of-oalist-tc-of-list-id*:
assumes *tc.oalist-inv-raw xs*
shows *list-of-oalist-tc* (*OAlist-tc xs*) = *xs*
using *assms* **by** (*simp add*: *list-of-oalist-OAlist-tc tc.sort-oalist-id*)

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* (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*))

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* (λ*k*. *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*

unfolding *OAlist-tc-lookup-def* **using** *oalist-inv-list-of-oalist-tc* **by** (*rule tc.lookup-pair-eq-valueI*)

lemma *OAlist-tc-lookup-inj*: *OAlist-tc-lookup xs* = *OAlist-tc-lookup ys* $\Longrightarrow$ *xs* = *ys*
by (*simp add*: *OAlist-tc-lookup-def oalist-inv-list-of-oalist-tc oalist-tc-eqI tc.lookup-pair-inj*)

lemma *OAlist-tc-lookup-oalist-of-list*:
distinct (*map fst xs*) $\Longrightarrow$ *OAlist-tc-lookup* (*oalist-tc-of-list xs*) = *lookup-dflt xs*
by (*simp add*: *OAlist-tc-lookup-def list-of-oalist-tc-of-list tc.lookup-pair-sort-oalist'*)

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*)
unfolding *OAlist-tc-sorted-domain-def* **by** *simp*

lemma *in-OAlist-tc-sorted-domain-iff-lookup*: *k* ∈ *set* (*OAlist-tc-sorted-domain xs*) $\longleftrightarrow$ (*OAlist-tc-lookup xs k* ≠ *0*)
unfolding *OAlist-tc-sorted-domain-def OAlist-tc-lookup-def* **using** *oalist-inv-list-of-oalist-tc tc.lookup-pair-eq-0*
by *fastforce*

lemma *sorted-OAlist-tc-sorted-domain*: *sorted-wrt* (<) (*OAlist-tc-sorted-domain xs*)
unfolding *OAlist-tc-sorted-domain-def tc-le-lt*[*symmetric*] **using** *oalist-inv-list-of-oalist-tc*
by (*rule tc.oalist-inv-rawD2*)

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* = []
unfolding *OAlist-tc-empty-def* **using** *tc.oalist-inv-raw-Nil* **by** (*rule list-of-oalist-tc-of-list-id*)

lemma *lookup-OAlist-tc-empty*: *OAlist-tc-lookup OAlist-tc-empty k* = *0*
by (*simp add*: *OAlist-tc-lookup-def*)

lemma *OAlist-tc-lookup-single*:
OAlist-tc-lookup (*oalist-tc-of-list* [(*k*, *v*)]) *k'* = (*if k* = *k' then v else 0*)
by (*simp add*: *OAlist-tc-lookup-def list-of-oalist-tc-of-list tc.sort-oalist-def comparator-of-def split*: *order.split*)

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*)
unfolding *OAlist-tc-except-min-def*
by (*rule list-of-oalist-tc-of-list-id*, *rule tc.oalist-inv-raw-tl*, *fact oalist-inv-list-of-oalist-tc*)

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*)
by (*simp add*: *OAlist-tc-lookup-def tc.lookup-pair-tl oalist-inv-list-of-oalist-tc split del*: *if-split cong*: *if-cong*)

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*)
unfolding *OAlist-tc-min-key-val-def* **using** *assms* **by** *simp*

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 −
let *?xs* = *list-of-oalist-tc xs*
from *assms* **have** ∗: *OAlist-tc-min-key-val xs* ∈ *set ?xs* **by** (*rule OAlist-tc-min-key-val-in*)
show *?thesis* **unfolding** *OAlist-tc-lookup-def*
by (*rule HOL.sym*, *rule tc.lookup-pair-eq-valueI*, *fact oalist-inv-list-of-oalist-tc*, *simp add*: ∗)
qed

lemma *OAlist-tc-min-key-val-minimal*:
assumes *z* ∈ *set* (*list-of-oalist-tc xs*)
shows *fst* (*OAlist-tc-min-key-val xs*) ≤ *fst z*
proof −
let *?xs* = *list-of-oalist-tc xs*
from *assms* **have** *?xs* ≠ [] **by** *auto*
hence *OAlist-tc-sorted-domain xs* ≠ [] **by** (*simp add*: *OAlist-tc-sorted-domain-def*)
then obtain *h xs′* **where** *eq*: *OAlist-tc-sorted-domain xs* = *h* # *xs′* **using** *list.exhaust* **by** *blast*
with *sorted-OAlist-tc-sorted-domain*[*of xs*] **have** ∗: ∀ *y*∈*set xs′*. *h* < *y* **by** *simp*
from *assms* **have** *fst z* ∈ *set* (*OAlist-tc-sorted-domain xs*) **by** (*simp add*: *OAlist-tc-sorted-domain-def*)
hence *disj*: *fst z* = *h* ∨ *fst z* ∈ *set xs′* **by** (*simp add*: *eq*)
from ‹*?xs* ≠ []› **have** *fst* (*OAlist-tc-min-key-val xs*) = *hd* (*OAlist-tc-sorted-domain xs*)
by (*simp add*: *OAlist-tc-min-key-val-def OAlist-tc-sorted-domain-def hd-map*)
also have ... = *h* **by** (*simp add*: *eq*)
also from *disj* **have** ... ≤ *fst z*
proof
assume *fst z* = *h*
thus *?thesis* **by** *simp*
next
assume *fst z* ∈ *set xs′*
with ∗ **have** *h* < *fst z* **..**
thus *?thesis* **by** *simp*

qed
finally show *?thesis* **.**
qed

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*)
unfolding *OAlist-tc-insert-def*
by (*rule list-of-oalist-tc-of-list-id*, *rule tc.oalist-inv-raw-update-by-pair*, *fact oalist-inv-list-of-oalist-tc*)

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′*)
by (*simp add*: *OAlist-tc-lookup-def tc.lookup-pair-update-by-pair oalist-inv-list-of-oalist-tc split del*: *if-split cong*: *if-cong*)

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*)
unfolding *OAlist-tc-update-by-fun-def*
by (*rule list-of-oalist-tc-of-list-id*, *rule tc.oalist-inv-raw-update-by-fun-pair*, *fact oalist-inv-list-of-oalist-tc*)

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′*)
by (*simp add*: *OAlist-tc-lookup-def tc.lookup-pair-update-by-fun-pair oalist-inv-list-of-oalist-tc split del*: *if-split cong*: *if-cong*)

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*)
unfolding *OAlist-tc-update-by-fun-gr-def*
by (*rule list-of-oalist-tc-of-list-id*, *rule tc.oalist-inv-raw-update-by-fun-gr-pair*, *fact oalist-inv-list-of-oalist-tc*)

lemma *OAlist-tc-update-by-fun-gr-eq-OAlist-tc-update-by-fun*: *OAlist-tc-update-by-fun-gr* = *OAlist-tc-update-by-fun*
by (*rule*, *rule*, *rule*,
simp add: *OAlist-tc-update-by-fun-gr-def OAlist-tc-update-by-fun-def*
tc.update-by-fun-gr-pair-eq-update-by-fun-pair oalist-inv-list-of-oalist-tc)

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*)
unfolding *OAlist-tc-filter-def*

by (*rule list-of-oalist-tc-of-list-id*, *rule tc.oalist-inv-raw-filter*, *fact oalist-inv-list-of-oalist-tc*)

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*)
by (*simp add*: *OAlist-tc-lookup-def tc.lookup-pair-filter oalist-inv-list-of-oalist-tc*)

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*)
unfolding *OAlist-tc-map-val-def*
by (*rule list-of-oalist-tc-of-list-id*, *rule tc.oalist-inv-raw-map-val-pair*, *fact oalist-inv-list-of-oalist-tc*)

lemma *OAlist-tc-map-val-cong*:
assumes $\bigwedge k\ v.\ (k,\ v) \in set\ (list\text{-}of\text{-}oalist\text{-}tc\ xs) \Longrightarrow f\ k\ v = g\ k\ v$
shows *OAlist-tc-map-val f xs* = *OAlist-tc-map-val g xs*
unfolding *OAlist-tc-map-val-def* **by** (*rule arg-cong*[**where** *f*=*OAlist-tc*], *rule tc.map-val-pair-cong*, *elim assms*)

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*)
by (*simp add*: *OAlist-tc-lookup-def tc.lookup-pair-map-val-pair oalist-inv-list-of-oalist-tc*)

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*)
unfolding *OAlist-tc-map2-val-def*
by (*rule list-of-oalist-tc-of-list-id*, *rule tc.oalist-inv-raw-map2-val-pair*,
fact oalist-inv-list-of-oalist-tc, *fact oalist-inv-list-of-oalist-tc*,
fact tc.map2-val-compat-map-val-pair, *fact tc.map2-val-compat-map-val-pair*)

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*)
unfolding *OAlist-tc-map2-val-rneutr-def*
by (*rule list-of-oalist-tc-of-list-id*, *rule tc.oalist-inv-raw-map2-val-pair*,
fact oalist-inv-list-of-oalist-tc, *fact oalist-inv-list-of-oalist-tc*,
fact tc.map2-val-compat-id, *fact tc.map2-val-compat-map-val-pair*)

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*)
unfolding *OAlist-tc-map2-val-neutr-def*
by (*rule list-of-oalist-tc-of-list-id*, *rule tc.oalist-inv-raw-map2-val-pair*,
fact oalist-inv-list-of-oalist-tc, *fact oalist-inv-list-of-oalist-tc*,

fact tc.map2-val-compat-id, *fact tc.map2-val-compat-id*)

lemma *lookup-OAlist-tc-map2-val*:
assumes $\bigwedge k$. *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*)
by (*simp add*: *OAlist-tc-lookup-def tc.lookup-pair-map2-val-pair tc.map2-val-compat-map-val-pair assms oalist-inv-list-of-oalist-tc*)

lemma *lookup-OAlist-tc-map2-val-rneutr*:
assumes $\bigwedge k$ *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 (*simp add*: *OAlist-tc-lookup-def*, *rule tc.lookup-pair-map2-val-pair*)
fix *zs*::(*'a* × *'b*) *list*
assume *tc.oalist-inv-raw zs*
thus *id zs* = *map-val-pair* (λ*k v*. *f k v 0*) *zs* **by** (*simp add*: *assms tc.map-pair-id*)
qed (*fact oalist-inv-list-of-oalist-tc*, *fact oalist-inv-list-of-oalist-tc*, *fact tc.map2-val-compat-id*, *fact tc.map2-val-compat-map-val-pair*, *rule refl*, *simp only*: *assms*)

lemma *lookup-OAlist-tc-map2-val-neutr*:
assumes $\bigwedge k$ *x*. *f k x 0* = *x* **and** $\bigwedge k$ *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 (*simp add*: *OAlist-tc-lookup-def*, *rule tc.lookup-pair-map2-val-pair*)
fix *zs*::(*'a* × *'b*) *list*
assume *tc.oalist-inv-raw zs*
thus *id zs* = *map-val-pair* (λ*k v*. *f k v 0*) *zs* **by** (*simp add*: *assms*(*1*) *tc.map-pair-id*)
next
fix *zs*::(*'a* × *'b*) *list*
assume *tc.oalist-inv-raw zs*
thus *id zs* = *map-val-pair* (λ*k*. *f k 0*) *zs* **by** (*simp add*: *assms*(*2*) *tc.map-pair-id*)
qed (*fact oalist-inv-list-of-oalist-tc*, *fact oalist-inv-list-of-oalist-tc*, *fact tc.map2-val-compat-id*, *fact tc.map2-val-compat-id*, *simp only*: *assms*(*1*))

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*
by (*simp add*: *OAlist-tc-map2-val-rneutr-def OAlist-tc-update-by-fun-def assms tc.map2-val-pair-singleton-eq-update-by-fun-pair oalist-inv-list-of-oalist-tc*)

12.10.11 *OAlist-tc-lex-ord* and *OAlist-tc-prod-ord*

lemma *OAlist-tc-lex-ord-EqI*:
($\bigwedge k$. *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*) ⟹
OAlist-tc-lex-ord f xs ys = *Some Eq*

by (*simp add*: *OAlist-tc-lex-ord-def OAlist-tc-lookup-def*, *rule tc.lex-ord-pair-EqI*, *rule oalist-inv-list-of-oalist-tc*, *rule oalist-inv-list-of-oalist-tc*, *blast*)

lemma *OAlist-tc-lex-ord-valI*:
assumes $aux \neq$ *Some Eq* **and** $k \in$ *fst ‘ set* (*list-of-oalist-tc xs*) $\cup$ *fst ‘ set* (*list-of-oalist-tc ys*)
shows $aux = f\ k$ (*OAlist-tc-lookup xs k*) (*OAlist-tc-lookup ys k*) $\Longrightarrow$
($\bigwedge k'$. $k' \in$ *fst ‘ set* (*list-of-oalist-tc xs*) $\cup$ *fst ‘ set* (*list-of-oalist-tc ys*) $\Longrightarrow$
$k' < k \Longrightarrow f\ k'$ (*OAlist-tc-lookup xs k'*) (*OAlist-tc-lookup ys k'*) = *Some Eq*) $\Longrightarrow$
OAlist-tc-lex-ord f xs ys = *aux*
by (*simp* (*no-asm-use*) *add*: *OAlist-tc-lex-ord-def OAlist-tc-lookup-def*, *rule tc.lex-ord-pair-valI*, *rule oalist-inv-list-of-oalist-tc*, *rule oalist-inv-list-of-oalist-tc*, *rule assms*(*1*), *rule assms*(*2*), *simp-all*)

lemma *OAlist-tc-lex-ord-EqD*:
OAlist-tc-lex-ord f xs ys = *Some Eq* $\Longrightarrow$
$k \in$ *fst ‘ set* (*list-of-oalist-tc xs*) $\cup$ *fst ‘ set* (*list-of-oalist-tc ys*) $\Longrightarrow$
f k (*OAlist-tc-lookup xs k*) (*OAlist-tc-lookup ys k*) = *Some Eq*
by (*simp add*: *OAlist-tc-lex-ord-def OAlist-tc-lookup-def*, *rule tc.lex-ord-pair-EqD*[**where** *f=f*], *rule oalist-inv-list-of-oalist-tc*, *rule oalist-inv-list-of-oalist-tc*, *assumption*, *simp*)

lemma *OAlist-tc-lex-ord-valE*:
assumes *OAlist-tc-lex-ord f xs ys* = *aux* **and** $aux \neq$ *Some Eq*
obtains *k* **where** $k \in$ *fst ‘ set* (*list-of-oalist-tc xs*) $\cup$ *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-of-oalist-tc xs*) $\cup$ *fst ‘ set* (*list-of-oalist-tc ys*) $\Longrightarrow$
$k' < k \Longrightarrow f\ k'$ (*OAlist-tc-lookup xs k'*) (*OAlist-tc-lookup ys k'*) = *Some Eq*
proof –
note *oalist-inv-list-of-oalist-tc oalist-inv-list-of-oalist-tc*
moreover from *assms*(*1*) **have** *lex-ord-pair-tc f* (*list-of-oalist-tc xs*) (*list-of-oalist-tc ys*) = *aux*
by (*simp only*: *OAlist-tc-lex-ord-def*)
ultimately obtain *k* **where** *1*: $k \in$ *fst ‘ set* (*list-of-oalist-tc xs*) $\cup$ *fst ‘ set* (*list-of-oalist-tc ys*)
and $aux = f\ k$ (*lookup-pair-tc* (*list-of-oalist-tc xs*) *k*) (*lookup-pair-tc* (*list-of-oalist-tc ys*) *k*)
and $\bigwedge k'$. $k' \in$ *fst ‘ set* (*list-of-oalist-tc xs*) $\cup$ *fst ‘ set* (*list-of-oalist-tc ys*) $\Longrightarrow$
$k' < k \Longrightarrow$
$f\ k'$ (*lookup-pair-tc* (*list-of-oalist-tc xs*) *k'*) (*lookup-pair-tc* (*list-of-oalist-tc ys*) *k'*) = *Some Eq*
using *assms*(*2*) **unfolding** *tc-le-lt*[*symmetric*] **by** (*rule tc.lex-ord-pair-valE*, *blast*)
from *this*(*2*, *3*) **have** $aux = f\ k$ (*OAlist-tc-lookup xs k*) (*OAlist-tc-lookup ys k*)
and $\bigwedge k'$. $k' \in$ *fst ‘ set* (*list-of-oalist-tc xs*) $\cup$ *fst ‘ set* (*list-of-oalist-tc ys*) $\Longrightarrow$
$k' < k \Longrightarrow f\ k'$ (*OAlist-tc-lookup xs k'*) (*OAlist-tc-lookup ys k'*) = *Some Eq*

by (*simp-all only*: *OAlist-tc-lookup-def*)
with *1* **show** *?thesis* **..**
qed

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*))
by (*simp add*: *OAlist-tc-prod-ord-def OAlist-tc-lookup-def tc.prod-ord-pair-alt oalist-inv-list-of-oalist-tc*)

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 by (*intro-classes*, *simp add*: *equal-oalist-tc-def list-of-oalist-tc-inject*)

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*)]
by *eval*

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*)]
by *eval*

lemma *OAlist-tc-min-key-val* (*oalist-tc-of-list* [(*1*, *3*), (*0*::*nat*, *4*::*nat*), (*0*, *2*), (*1*, *1*)]) = (*0*, *4*)
by *eval*

lemma *OAlist-tc-lookup* (*oalist-tc-of-list* [(*0*::*nat*, *4*::*nat*), (*1*, *3*), (*0*, *2*), (*1*, *1*)]) *1* = *3*
by *eval*

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*
by *eval*

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*)]

by *eval*

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* × *nat*, though. All types we are interested in meet this requirement.

lemma *comparator-lexicographic*:
  fixes *f*::*'a* ⇒ *'b* **and** *g*::*'a* ⇒ *'c*
  assumes *comparator c1* **and** *comparator c2* **and** ⋀*x y*. *f x* = *f y* ⟹ *g x* = *g y* ⟹ *x* = *y*
  shows *comparator* (λ*x y*. *case c1* (*f x*) (*f y*) *of Eq* ⇒ *c2* (*g x*) (*g y*) | *val* ⇒ *val*)
      (**is** *comparator ?c3*)
proof −
  from *assms*(*1*) **interpret** *c1*: *comparator c1* **.**
  from *assms*(*2*) **interpret** *c2*: *comparator c2* **.**
  show *?thesis*
  proof
    fix *x y* :: *'a*
    show *invert-order* (*?c3 x y*) = *?c3 y x*
      by (*simp add*: *c1.eq c2.eq split*: *order.split*,
          metis invert-order.simps(*1*) *invert-order.simps*(*2*) *c1.sym c2.sym order.distinct*(*5*))
  next
    fix *x y* :: *'a*
    assume *?c3 x y* = *Eq*
    hence *f x* = *f y* **and** *g x* = *g y* **by** (*simp-all add*: *c1.eq c2.eq split*: *order.splits if-split-asm*)
    thus *x* = *y* **by** (*rule assms*(*3*))
  next
    fix *x y z* :: *'a*
    assume *?c3 x y* = *Lt*
    hence *d1*: *c1* (*f x*) (*f y*) = *Lt* ∨ (*c1* (*f x*) (*f y*) = *Eq* ∧ *c2* (*g x*) (*g y*) = *Lt*)
      by (*simp split*: *order.splits*)
    assume *?c3 y z* = *Lt*
    hence *d2*: *c1* (*f y*) (*f z*) = *Lt* ∨ (*c1* (*f y*) (*f z*) = *Eq* ∧ *c2* (*g y*) (*g z*) = *Lt*)
      by (*simp split*: *order.splits*)
    from *d1* **show** *?c3 x z* = *Lt*

proof
assume *1*: *c1* (*f x*) (*f y*) = *Lt*
from *d2* **show** *?thesis*
proof
assume *c1* (*f y*) (*f z*) = *Lt*
with *1* **have** *c1* (*f x*) (*f z*) = *Lt* **by** (*rule c1.comp-trans*)
thus *?thesis* **by** *simp*
next
assume *c1* (*f y*) (*f z*) = *Eq* ∧ *c2* (*g y*) (*g z*) = *Lt*
hence *f z* = *f y* **and** *c2* (*g y*) (*g z*) = *Lt* **by** (*simp-all add*: *c1.eq*)
with *1* **show** *?thesis* **by** *simp*
qed
next
assume *c1* (*f x*) (*f y*) = *Eq* ∧ *c2* (*g x*) (*g y*) = *Lt*
hence *1*: *f x* = *f y* **and** *2*: *c2* (*g x*) (*g y*) = *Lt* **by** (*simp-all add*: *c1.eq*)
from *d2* **show** *?thesis*
proof
assume *c1* (*f y*) (*f z*) = *Lt*
thus *?thesis* **by** (*simp add*: *1*)
next
assume *c1* (*f y*) (*f z*) = *Eq* ∧ *c2* (*g y*) (*g z*) = *Lt*
hence *3*: *f y* = *f z* **and** *c2* (*g y*) (*g z*) = *Lt* **by** (*simp-all add*: *c1.eq*)
from *2 this*(*2*) **have** *c2* (*g x*) (*g z*) = *Lt* **by** (*rule c2.comp-trans*)
thus *?thesis* **by** (*simp add*: *1 3*)
qed
qed
qed
qed

class *nat-term* =
fixes *rep-nat-term* :: *'a* ⇒ ((*nat*, *nat*) *pp* × *nat*)
and *splus* :: *'a* ⇒ *'a* ⇒ *'a*
assumes *rep-nat-term-inj*: *rep-nat-term x* = *rep-nat-term y* ⟹ *x* = *y*
and *full-component*: *snd* (*rep-nat-term x*) = *i* ⟹ (∃ *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* = (*λx y*. *case comp-of-ord lex-pp* (*fst* (*rep-nat-term x*)) (*fst* (*rep-nat-term y*)) *of*
Eq ⇒ *comparator-of* (*snd* (*rep-nat-term x*)) (*snd* (*rep-nat-term y*)) | *val* ⇒ *val*)

lemma *full-componentE*:
assumes *snd* (*rep-nat-term x*) = *i*
obtains *y* **where** *rep-nat-term y* = (*t*, *i*)
proof −
from *assms* **have** ∃ *y*. *rep-nat-term y* = (*t*, *i*) **by** (*rule full-component*)

then obtain *y* **where** *rep-nat-term y* = (*t*, *i*) **..**
thus *?thesis* **..**
qed

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* ⇒ *bool*
where *nat-term-comp cmp* ⟷
(∀ *u v*. *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*) ⟶ *fst* (*rep-nat-term u*) = *0* ⟶ *cmp u v* ≠ *Gt*) ∧
(∀ *u v*. *fst* (*rep-nat-term u*) = *fst* (*rep-nat-term v*) ⟶ *snd* (*rep-nat-term u*) < *snd* (*rep-nat-term v*) ⟶ *cmp u v* = *Lt*) ∧
(∀ *t u v*. *cmp u v* = *Lt* ⟶ *cmp* (*splus t u*) (*splus t v*) = *Lt*) ∧
(∀ *u v a b*. *fst* (*rep-nat-term u*) = *fst* (*rep-nat-term a*) ⟶ *fst* (*rep-nat-term v*) = *fst* (*rep-nat-term b*) ⟶
snd (*rep-nat-term u*) = *snd* (*rep-nat-term v*) ⟶ *snd* (*rep-nat-term a*) = *snd* (*rep-nat-term b*) ⟶
cmp a b = *Lt* ⟶ *cmp u v* = *Lt*)

lemma *nat-term-compI*:
assumes ⋀*u v*. *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*) ⟹ *fst* (*rep-nat-term u*) = *0* ⟹ *cmp u v* ≠ *Gt*
and ⋀*u v*. *fst* (*rep-nat-term u*) = *fst* (*rep-nat-term v*) ⟹ *snd* (*rep-nat-term u*) < *snd* (*rep-nat-term v*) ⟹ *cmp u v* = *Lt*
and ⋀*t u v*. *cmp u v* = *Lt* ⟹ *cmp* (*splus t u*) (*splus t v*) = *Lt*
and ⋀*u v a b*. *fst* (*rep-nat-term u*) = *fst* (*rep-nat-term a*) ⟹ *fst* (*rep-nat-term v*) = *fst* (*rep-nat-term b*) ⟹
snd (*rep-nat-term u*) = *snd* (*rep-nat-term v*) ⟹ *snd* (*rep-nat-term a*) = *snd* (*rep-nat-term b*) ⟹
cmp a b = *Lt* ⟹ *cmp u v* = *Lt*
shows *nat-term-comp cmp*
unfolding *nat-term-comp-def fst-conv snd-conv* **using** *assms* **by** *blast*

lemma *nat-term-compD1*:
assumes *nat-term-comp cmp* **and** *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*) **and** *fst* (*rep-nat-term u*) = *0*
shows *cmp u v* ≠ *Gt*
using *assms* **unfolding** *nat-term-comp-def fst-conv* **by** *blast*

lemma *nat-term-compD2*:
assumes *nat-term-comp cmp* **and** *fst* (*rep-nat-term u*) = *fst* (*rep-nat-term v*) **and** *snd* (*rep-nat-term u*) < *snd* (*rep-nat-term v*)
shows *cmp u v* = *Lt*
using *assms* **unfolding** *nat-term-comp-def fst-conv snd-conv* **by** *blast*

lemma *nat-term-compD3*:
assumes *nat-term-comp cmp* **and** *cmp u v = Lt*
shows *cmp* (*splus t u*) (*splus t v*) = *Lt*
using *assms* **unfolding** *nat-term-comp-def snd-conv* **by** *blast*

lemma *nat-term-compD4*:
assumes *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 = Lt*
shows *cmp u v = Lt*
using *assms* **unfolding** *nat-term-comp-def snd-conv* **by** *blast*

lemma *nat-term-compD1'*:
assumes *comparator cmp* **and** *nat-term-comp cmp* **and** *snd* (*rep-nat-term u*) ≤ *snd* (*rep-nat-term v*)
and *fst* (*rep-nat-term u*) = *0*
shows *cmp u v* ≠ *Gt*
proof (*cases snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*))
case *True*
with *assms*(*2*) **show** *?thesis* **using** *assms*(*4*) **by** (*rule nat-term-compD1*)
next
from *assms*(*1*) **interpret** *cmp*: *comparator cmp* **.**
case *False*
with *assms*(*3*) **have** *a*: *snd* (*rep-nat-term u*) < *snd* (*rep-nat-term v*) **by** *simp*
from *refl* **obtain** *w*::*'a* **where** *eq*: *rep-nat-term w* = (*0*, *snd* (*rep-nat-term v*)) **by** (*rule full-componentE*)
have *cmp u w = Lt* **by** (*rule nat-term-compD2*, *fact assms*(*2*), *simp-all add*: *eq assms*(*4*) *a*)
moreover have *cmp w v* ≠ *Gt* **by** (*rule nat-term-compD1*, *fact assms*(*2*), *simp-all add*: *eq*)
ultimately show *cmp u v* ≠ *Gt* **by** (*simp add*: *cmp.nGt-le-conv cmp.Lt-lt-conv*)
qed

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 −
from *assms*(*1*) **interpret** *cmp*: *comparator cmp* **.**
show *?thesis*
proof (*cases cmp a b*)
case *Eq*
hence *fst* (*rep-nat-term u*) = *fst* (*rep-nat-term v*) **by** (*simp add*: *cmp.eq assms*(*3*, *4*))
hence *rep-nat-term u = rep-nat-term v* **using** *assms*(*5*) **by** (*rule prod-eqI*)

hence *u* = *v* **by** (*rule rep-nat-term-inj*)
thus *?thesis* **by** (*simp add*: *Eq*)
next
case *Lt*
with *assms*(*2*, *3*, *4*, *5*, *6*) **have** *cmp u v* = *Lt* **by** (*rule nat-term-compD4*)
thus *?thesis* **by** (*simp add*: *Lt*)
next
case *Gt*
hence *cmp b a* = *Lt* **by** (*simp only*: *cmp.Gt-lt-conv cmp.Lt-lt-conv*)
with *assms*(*2*, *4*, *3*) *assms*(*5*, *6*)[*symmetric*] **have** *cmp v u* = *Lt* **by** (*rule nat-term-compD4*)
hence *cmp u v* = *Gt* **by** (*simp only*: *cmp.Gt-lt-conv cmp.Lt-lt-conv*)
thus *?thesis* **by** (*simp add*: *Gt*)
qed
qed

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 (*cases snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*))
case *True*
with *assms*(*1*, *2*, *3*, *4*) **have** *cmp u v* = *cmp a b* **using** *assms*(*6*) **by** (*rule nat-term-compD4'*)
thus *?thesis* **using** *assms*(*7*) **by** *simp*
next
case *False*
from *assms*(*1*) **interpret** *cmp*: *comparator cmp* **.**
from *refl* **obtain** *w*::*'a* **where** *w*: *rep-nat-term w* = (*fst* (*rep-nat-term u*), *snd* (*rep-nat-term v*))
by (*rule full-componentE*)
have *1*: *fst* (*rep-nat-term w*) = *fst* (*rep-nat-term a*) **and** *2*: *snd* (*rep-nat-term w*) = *snd* (*rep-nat-term v*)
by (*simp-all add*: *w assms*(*3*))
from *False assms*(*5*) **have** ∗: *snd* (*rep-nat-term u*) < *snd* (*rep-nat-term v*) **by** *simp*
have *cmp u w* = *Lt* **by** (*rule nat-term-compD2*, *fact assms*(*2*), *simp-all add*: ∗ *w*)
moreover from *assms*(*1*, *2*) *1 assms*(*4*) *2 assms*(*6*) **have** *cmp w v* = *cmp a b* **by** (*rule nat-term-compD4'*)
ultimately show *?thesis* **using** *assms*(*7*) **by** (*metis cmp.nGt-le-conv cmp.nLt-le-conv cmp.comp-trans*)
qed

lemma *comparator-lex-comp-aux*: *comparator* (*lex-comp-aux*::*'a*::*nat-term comparator*)

unfolding *lex-comp-aux-def*
proof (*rule comparator-composition*)
from *lex-pp-antisym* **have** *as*: *antisymp lex-pp* **by** (*rule antisympI*)
have *comparator* (*comp-of-ord* (*lex-pp*::(*nat*, *nat*) *pp* ⇒ -))
unfolding *comp-of-ord-eq-comp-of-ords*[*OF as*]
by (*rule comp-of-ords*, *unfold-locales*,
auto simp: *lex-pp-refl intro*: *lex-pp-trans lex-pp-lin′ elim*!: *lex-pp-antisym*)
thus *comparator* (λ*x y*::((*nat*, *nat*) *pp* × *nat*). *case comp-of-ord lex-pp* (*fst x*) (*fst y*) *of*
Eq ⇒ *comparator-of* (*snd x*) (*snd y*) | *val* ⇒ *val*)
using *comparator-of prod-eqI* **by** (*rule comparator-lexicographic*)
next
from *rep-nat-term-inj* **show** *inj rep-nat-term* **by** (*rule injI*)
qed

lemma *nat-term-comp-lex-comp-aux*: *nat-term-comp* (*lex-comp-aux*::′*a*::*nat-term comparator*)
proof −
from *lex-pp-antisym* **have** *as*: *antisymp lex-pp* **by** (*rule antisympI*)
interpret *lex*: *comparator comp-of-ord* (*lex-pp*::(*nat*, *nat*) *pp* ⇒ -)
unfolding *comp-of-ord-eq-comp-of-ords*[*OF as*]
by (*rule comp-of-ords*, *unfold-locales*,
auto simp: *lex-pp-refl intro*: *lex-pp-trans lex-pp-lin′ elim*!: *lex-pp-antisym*)
show *?thesis*
proof (*rule nat-term-compI*)
fix *u v* :: ′*a*
assume *1*: *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*) **and** *2*: *fst* (*rep-nat-term u*) = *0*
show *lex-comp-aux u v* ≠ *Gt*
by (*simp add*: *lex-comp-aux-def 1 2 split*: *order.split*, *simp add*: *comp-of-ord-def lex-pp-zero-min*)
next
fix *u v* :: ′*a*
assume *1*: *fst* (*rep-nat-term u*) = *fst* (*rep-nat-term v*) **and** *2*: *snd* (*rep-nat-term u*) < *snd* (*rep-nat-term v*)
show *lex-comp-aux u v* = *Lt*
by (*simp add*: *lex-comp-aux-def 1 split*: *order.split*, *simp add*: *comparator-of-def 2*)
next
fix *t u v* :: ′*a*
show *lex-comp-aux u v* = *Lt* ⟹ *lex-comp-aux* (*splus t u*) (*splus t v*) = *Lt*
by (*auto simp*: *lex-comp-aux-def splus-term pprod.splus-def comp-of-ord-def lex-pp-refl*
split: *order.splits if-splits intro*: *lex-pp-plus-monotone′*)
next
fix *u v a b* :: ′*a*
assume *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 a*) = *snd* (*rep-nat-term b*) **and** *lex-comp-aux a b* = *Lt*

  thus *lex-comp-aux u v* = *Lt* **by** (*simp add*: *lex-comp-aux-def split*: *order.splits*)
 qed
qed

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 (*rule*, *simp*)
 from *comparator-lex-comp-aux nat-term-comp-lex-comp-aux*
 show *comparator lex-comp-aux* ∧ *nat-term-comp lex-comp-aux* **..**
qed

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*
 by (*rule Abs-nat-term-order-inverse*, *simp add*: *assms*)

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 by (*standard*, *simp add*: *equal-nat-term-order-def*)

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*)
 using *nat-term-compare* **by** *blast*

lemma *nat-term-comp-nat-term-compare*: *nat-term-comp* (*nat-term-compare to*)
using *nat-term-compare* **by** *blast*

lemma *nat-term-compare-splus*: *nat-term-compare to* (*splus t u*) (*splus t v*) = *nat-term-compare to u v*
proof −
from *comparator-nat-term-compare* **interpret** *cmp*: *comparator nat-term-compare to* **.**
show *?thesis*
proof (*cases nat-term-compare to u v*)
case *Eq*
hence *splus t u* = *splus t v* **by** (*simp add*: *cmp.eq*)
thus *?thesis* **by** (*simp add*: *cmp.eq Eq*)
next
case *Lt*
moreover from *nat-term-comp-nat-term-compare this* **have** *nat-term-compare to* (*splus t u*) (*splus t v*) = *Lt*
by (*rule nat-term-compD3*)
ultimately show *?thesis* **by** *simp*
next
case *Gt*
hence *nat-term-compare to v u* = *Lt* **using** *cmp.Gt-lt-conv cmp.Lt-lt-conv* **by** *auto*
with *nat-term-comp-nat-term-compare* **have** *nat-term-compare to* (*splus t v*) (*splus t u*) = *Lt*
by (*rule nat-term-compD3*)
hence *nat-term-compare to* (*splus t u*) (*splus t v*) = *Gt* **using** *cmp.Gt-lt-conv cmp.Lt-lt-conv* **by** *auto*
with *Gt* **show** *?thesis* **by** *simp*
qed
qed

lemma *nat-term-compare-conv*: *nat-term-compare to* = *key-compare* (*key-order-of-nat-term-order to*)
unfolding *key-order-of-nat-term-order-def*
by (*rule sym*, *rule Abs-key-order-inverse*, *simp add*: *comparator-nat-term-compare*)

lemma *comparator-nat-term-compare-inv*: *comparator* (*nat-term-compare-inv to*)
unfolding *nat-term-compare-inv-def* **using** *comparator-nat-term-compare* **by** (*rule comparator-converse*)

lemma *nat-term-compare-inv-conv*: *nat-term-compare-inv to* = *key-compare* (*key-order-of-nat-term-order-inv to*)
unfolding *key-order-of-nat-term-order-inv-def*
by (*rule sym*, *rule Abs-key-order-inverse*, *simp add*: *comparator-nat-term-compare-inv*)

lemma *nat-term-compare-inv-alt* [*code-unfold*]: *nat-term-compare-inv to x y* = *nat-term-compare to y x*

by (*simp only*: *nat-term-compare-inv-def*)

lemma *le-of-nat-term-order* [*code*]: *le-of-nat-term-order to x y* = (*nat-term-compare to x y* ≠ *Gt*)
by (*simp add*: *le-of-key-order-alt le-of-nat-term-order-def nat-term-compare-conv*)

lemma *lt-of-nat-term-order* [*code*]: *lt-of-nat-term-order to x y* = (*nat-term-compare to x y* = *Lt*)
by (*simp add*: *lt-of-key-order-alt lt-of-nat-term-order-def nat-term-compare-conv*)

lemma *le-of-nat-term-order-alt*:
le-of-nat-term-order to = (λ*u v*. *ko.le* (*key-order-of-nat-term-order-inv to*) *v u*)
by (*intro ext*, *simp add*: *le-of-comp-def nat-term-compare-inv-conv*[*symmetric*] *le-of-nat-term-order-def*
le-of-key-order-def nat-term-compare-conv[*symmetric*] *nat-term-compare-inv-alt*)

lemma *lt-of-nat-term-order-alt*:
lt-of-nat-term-order to = (λ*u v*. *ko.lt* (*key-order-of-nat-term-order-inv to*) *v u*)
by (*intro ext*, *simp add*: *lt-of-comp-def nat-term-compare-inv-conv*[*symmetric*] *lt-of-nat-term-order-def*
lt-of-key-order-def nat-term-compare-conv[*symmetric*] *nat-term-compare-inv-alt*)

lemma *linorder-le-of-nat-term-order*: *class.linorder* (*le-of-nat-term-order to*) (*lt-of-nat-term-order to*)
unfolding *le-of-nat-term-order-alt lt-of-nat-term-order-alt* **using** *ko.linorder*
by (*rule linorder.dual-linorder*)

lemma *le-of-nat-term-order-zero-min*: *le-of-nat-term-order to 0* (*t*::′*a*::*nat-pp-term*)
unfolding *le-of-nat-term-order*
by (*rule nat-term-compD1*′, *fact comparator-nat-term-compare*, *fact nat-term-comp-nat-term-compare*, *simp-all add*: *rep-nat-term-zero*)

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*)
using *assms* **by** (*simp add*: *le-of-nat-term-order splus-pp-term*[*symmetric*] *nat-term-compare-splus*)

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*
by (*fact comparator-nat-term-compare-inv*)

lemma *ko-ntm-le*: *ko-ntm.le to* = (λ*x y*. *le-of-nat-term-order to y x*)

by (*intro ext*, *simp add*: *le-of-comp-def le-of-nat-term-order nat-term-compare-inv-def split*: *order.split*)

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*
subgoal by (*simp only*: *lookup-pair-ko-ntm-def nat-term-compare-inv-conv*)
subgoal by (*simp only*: *update-by-pair-ko-ntm-def nat-term-compare-inv-conv*)
subgoal by (*simp only*: *update-by-fun-pair-ko-ntm-def nat-term-compare-inv-conv*)
subgoal by (*simp only*: *update-by-fun-gr-pair-ko-ntm-def nat-term-compare-inv-conv*)
subgoal by (*simp only*: *map2-val-pair-ko-ntm-def nat-term-compare-inv-conv*)
subgoal by (*simp only*: *lex-ord-pair-ko-ntm-def nat-term-compare-inv-conv*)
subgoal by (*simp only*: *prod-ord-pair-ko-ntm-def nat-term-compare-inv-conv*)
subgoal by (*simp only*: *sort-oalist-ko-ntm′-def nat-term-compare-inv-conv*)
done

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 −
have *ko.le* (*key-order-of-nat-term-order-inv ko*) = (λ*x y*. *le-of-nat-term-order ko*

y x)
by (*metis ko.nGt-le-conv le-of-nat-term-order nat-term-compare-inv-conv nat-term-compare-inv-def*)
thus *?thesis* **by** (*simp only*: *min-key-val-ko-ntm-def oalist-raw.min-key-val-raw.simps*)
qed

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*
by (*auto simp*: *ko-ntm.oalist-inv-def intro*: *ko.oalist-inv-raw-Nil*)

lemma *oalist-ntm-eq-iff*: *xs* = *ys* ⟷ *list-of-oalist-ntm xs* = *list-of-oalist-ntm ys*
by (*simp add*: *list-of-oalist-ntm-inject*)

lemma *oalist-ntm-eqI*: *list-of-oalist-ntm xs* = *list-of-oalist-ntm ys* ⟹ *xs* = *ys*
by (*simp add*: *oalist-ntm-eq-iff*)

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*)
using *list-of-oalist-ntm*[*of xs*] **by** *simp*

lemma *list-of-oalist-OAlist-ntm*: *list-of-oalist-ntm* (*OAlist-ntm xs*) = *sort-oalist-ko-ntm xs*
proof −
obtain *xs' ox* **where** *xs*: *xs* = (*xs'*, *ox*) **by** *fastforce*
have *ko-ntm.oalist-inv* (*sort-oalist-ko-ntm' ox xs'*, *ox*)
using *ko-ntm.oalist-inv-sort-oalist-raw* **by** *fastforce*
thus *?thesis* **by** (*simp add*: *xs OAlist-ntm-def Abs-oalist-ntm-inverse*)
qed

lemma *OAlist-list-of-oalist-ntm* [*simp*, *code abstype*]: *OAlist-ntm* (*list-of-oalist-ntm xs*) = *xs*
proof −
obtain *xs' ox* **where** *xs*: *list-of-oalist-ntm xs* = (*xs'*, *ox*) **by** *fastforce*
have *ko-ntm.oalist-inv-raw ox xs'*
by (*simp add*: *xs*[*symmetric*] *ko-ntm.oalist-inv-alt*[*symmetric*] *nat-term-compare-inv-conv oalist-inv-list-of-oalist-ntm*)
thus *?thesis* **by** (*simp add*: *xs OAlist-ntm-def ko-ntm.sort-oalist-id*, *simp add*: *list-of-oalist-ntm-inverse xs*[*symmetric*])
qed

lemma [*code abstract*]: *list-of-oalist-ntm* (*oalist-of-list-ntm xs*) = *sort-oalist-ko-ntm xs*
by (*simp add*: *list-of-oalist-OAlist-ntm oalist-of-list-ntm-def*)

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*
  apply *unfold-locales*
  subgoal by (*fact oalist-inv-list-of-oalist-ntm*)
  subgoal by (*simp only*: *list-of-oalist-OAlist-ntm sort-oalist-ko-ntm-def*)
  subgoal by (*fact OAlist-list-of-oalist-ntm*)
  done

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* **..**

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 −
  have *abs-nat* (*rep-nat x*) = *abs-nat* (*rep-nat y*) **by** (*simp only*: *assms*)
  thus *?thesis* **by** (*simp only*: *rep-inverse*)
qed

corollary *rep-eq-iff*: (*rep-nat x* = *rep-nat y*) ⟷ (*x* = *y*)
  by (*auto elim*: *rep-inj*)

lemma *abs-inj*:
  assumes *abs-nat m* = *abs-nat n*
  shows *m* = *n*
proof −
  have *rep-nat* (*abs-nat m*) = *rep-nat* (*abs-nat n*) **by** (*simp only*: *assms*)
  thus *?thesis* **by** (*simp only*: *abs-inverse*)
qed

corollary *abs-eq-iff*: (*abs-nat m* = *abs-nat n*) ⟷ (*m* = *n*)
  by (*auto elim*: *abs-inj*)

lemma *rep-zero* [*simp*]: *rep-nat 0* = *0*
  using *abs-inverse abs-zero* **by** *fastforce*

lemma *rep-zero-iff*: (*rep-nat x* = *0*) ⟷ (*x* = *0*)
  using *rep-eq-iff* **by** *fastforce*

lemma *plus-eq*: *x* + *y* = *abs-nat* (*rep-nat x* + *rep-nat y*)
  by (*metis abs-plus rep-inverse*)

lemma *rep-plus*: *rep-nat* (*x* + *y*) = *rep-nat x* + *rep-nat y*
  by (*simp add*: *plus-eq*)

lemma *minus-eq*: *x* − *y* = *abs-nat* (*rep-nat x* − *rep-nat y*)
  by (*metis abs-minus rep-inverse*)

lemma *rep-minus*: *rep-nat* (*x* − *y*) = *rep-nat x* − *rep-nat y*
  by (*simp add*: *minus-eq*)

lemma *ord-iff*:
  x ≤ *y* ⟷ *rep-nat x* ≤ *rep-nat y* (**is** *?thesis1*)
  x < *y* ⟷ *rep-nat x* < *rep-nat y* (**is** *?thesis2*)
proof −
  show *?thesis1*
  proof
    assume *x* ≤ *y*
    show *rep-nat x* ≤ *rep-nat y*
    proof (*rule ccontr*)
      assume ¬ *rep-nat x* ≤ *rep-nat y*
      hence *rep-nat y* ≤ *rep-nat x* **and** *rep-nat x* ≠ *rep-nat y* **by** *simp-all*
     from *this*(*1*) **have** *abs-nat* (*rep-nat y*) ≤ *abs-nat* (*rep-nat x*) **by** (*rule abs-ord*)
      hence *y* ≤ *x* **by** (*simp only*: *rep-inverse*)
      moreover from ‹*rep-nat x* ≠ *rep-nat y*› **have** *y* ≠ *x* **using** *rep-inj* **by** *auto*
      ultimately have *y* < *x* **by** *simp*
      with ‹*x* ≤ *y*› **show** *False* **by** *simp*
    qed
  next
    assume *rep-nat x* ≤ *rep-nat y*
    hence *abs-nat* (*rep-nat x*) ≤ *abs-nat* (*rep-nat y*) **by** (*rule abs-ord*)
    thus *x* ≤ *y* **by** (*simp only*: *rep-inverse*)
  qed
  thus *?thesis2* **using** *rep-inj*[*of x y*] **by** (*auto simp*: *less-le Nat.nat-less-le*)
qed

lemma *ex-iff-abs*: (∃ *x*::′*a*. *P x*) ⟷ (∃ *n*::*nat*. *P* (*abs-nat n*))
  by (*metis rep-inverse*)

lemma *ex-iff-abs′*: (∃ *x*<*abs-nat m*. *P x*) ⟷ (∃ *n*::*nat*<*m*. *P* (*abs-nat n*))
  by (*metis abs-inverse rep-inverse ord-iff*(*2*))

lemma *all-iff-abs*: (∀ *x*::′*a*. *P x*) ⟷ (∀ *n*::*nat*. *P* (*abs-nat n*))
  by (*metis rep-inverse*)

lemma *all-iff-abs′*: (∀ *x*<*abs-nat m*. *P x*) ⟷ (∀ *n*::*nat*<*m*. *P* (*abs-nat n*))
  by (*metis abs-inverse rep-inverse ord-iff*(*2*))

subclass *linorder* **by** (*standard*, *auto simp*: *ord-iff rep-inj*)

lemma *comparator-of-rep* [*simp*]: *comparator-of* (*rep-nat x*) (*rep-nat y*) = *comparator-of x y*
  by (*simp add*: *comparator-of-def linorder-class.comparator-of-def ord-iff rep-inj*)

subclass *wellorder*
proof
  fix *P*::′*a* ⇒ *bool* **and** *a*::′*a*

let *?P* = λn::*nat*. *P* (*abs-nat n*)
assume *a*: $\bigwedge x$. ($\bigwedge y$. $y < x \Longrightarrow P\ y$) $\Longrightarrow P\ x$
have *P* (*abs-nat* (*rep-nat a*))
proof (*rule less-induct*[*of* - *rep-nat a*])
fix *n*::*nat*
assume *b*: $\bigwedge m$. $m < n \Longrightarrow$ *?P m*
show *?P n*
proof (*rule a*)
fix *y*
assume *y* < *abs-nat n*
hence *rep-nat y* < *n* **by** (*simp only*: *ord-iff abs-inverse*)
hence *?P* (*rep-nat y*) **by** (*rule b*)
thus *P y* **by** (*simp only*: *rep-inverse*)
qed
qed
thus *P a* **by** (*simp only*: *rep-inverse*)
qed

subclass *comm-monoid-add* **by** (*standard*, *auto simp*: *plus-eq intro*: *arg-cong*)

lemma *sum-rep*: *sum* (*rep-nat* ∘ *f*) *A* = *rep-nat* (*sum f A*) **for** *f* :: *'b* ⇒ *'a* **and** *A* :: *'b set*
proof (*induct A rule*: *infinite-finite-induct*)
case (*infinite A*)
thus *?case* **by** *simp*
next
case *empty*
show *?case* **by** *simp*
next
case (*insert a A*)
from *insert*(*1*, *2*) **show** *?case* **by** (*simp del*: *comp-apply add*: *insert*(*3*) *rep-plus*, *simp*)
qed

subclass *ordered-comm-monoid-add* **by** (*standard*, *simp add*: *ord-iff plus-eq*)

subclass *countable* **by** *intro-classes* (*intro exI*[*of* - *rep-nat*] *injI*, *elim rep-inj*)

subclass *cancel-comm-monoid-add*
apply *standard*
subgoal by (*simp add*: *minus-eq rep-plus*)
subgoal by (*simp add*: *minus-eq rep-plus*)
done

subclass *add-wellorder*
apply *standard*
subgoal by (*simp add*: *ord-iff rep-plus*)
subgoal unfolding *ord-iff* **by** (*drule le-imp-add*, *metis abs-plus rep-inverse*)
subgoal by (*simp add*: *ord-iff*)

done

end

lemma *the-min-eq-zero*: *the-min* = (*0*::*'a*::{*the-min*,*nat*})
proof −
have *the-min* ≤ (*0*::*'a*) **by** (*fact the-min-min*)
hence *rep-nat* (*the-min*::*'a*) ≤ *rep-nat* (*0*::*'a*) **by** (*simp only*: *ord-iff*)
also have ... = *0* **by** *simp*
finally have *rep-nat* (*the-min*::*'a*) = *0* **by** *simp*
thus *?thesis* **by** (*simp only*: *rep-zero-iff*)
qed

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 by (*standard*, *simp-all add*: *rep-nat-nat-def abs-nat-nat-def*)

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 by (*standard*, *simp-all add*: *rep-nat-natural-def abs-nat-natural-def*, *metis minus-natural.rep-eq nat-of-natural-of-nat of-nat-of-natural*)

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 −
 from *assms* **have** *1*: *rep-nat-pp* *x* ≤ *rep-nat-pp* *y* **and** *2*: *rep-nat-pp* *x* ≠ *rep-nat-pp* *y* **by** *simp-all*
 from *1* **have** *x* ≤ *y* **by** (*rule* *le-pp*)
 moreover from *2* **have** *x* ≠ *y* **by** *auto*
 ultimately show *?thesis* **by** *simp*
qed

lemma *rep-nat-pp-inj*:
 assumes *rep-nat-pp* *x* = *rep-nat-pp* *y*
 shows *x* = *y*
proof −
 have *abs-nat-pp* (*rep-nat-pp* *x*) = *abs-nat-pp* (*rep-nat-pp* *y*) **by** (*simp* *only*: *assms*)
 thus *?thesis* **by** *simp*
qed

lemma *lex-comp'-EqD*:
 assumes *lex-comp'* *x* *y* = *Eq*
 shows *x* = *y*
proof (*rule* *rep-nat-pp-inj*)
 from *assms* **show** *rep-nat-pp* *x* = *rep-nat-pp* *y* **by** (*simp* *add*: *lex-comp'* *comp-of-ord-def* *split*: *if-split-asm*)
qed

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 (*cases* *lex-comp'* *s* *t*)
 case *Eq*
 with *assms* **show** *?thesis* **..**
next
 case *Lt*
 hence *rep-nat-pp* *s* ≠ *rep-nat-pp* *t* **and** *lex-pp* (*rep-nat-pp* *s*) (*rep-nat-pp* *t*)
  by (*auto* *simp*: *lex-comp'* *comp-of-ord-def* *split*: *if-split-asm*)
 hence ∃ *x*. *lookup-pp* (*rep-nat-pp* *s*) *x* < *lookup-pp* (*rep-nat-pp* *t*) *x* ∧
      (∀ *y*<*x*. *lookup-pp* (*rep-nat-pp* *s*) *y* = *lookup-pp* (*rep-nat-pp* *t*) *y*)

by (*simp add*: *lex-pp-alt*)
then obtain *x* **where** *1*: *lookup-pp* (*rep-nat-pp s*) *x* < *lookup-pp* (*rep-nat-pp t*) *x*
and *2*: $\bigwedge y.\ y < x \Longrightarrow$ *lookup-pp* (*rep-nat-pp s*) *y* = *lookup-pp* (*rep-nat-pp t*) *y* **by** *blast*
show *?thesis*
proof
show *x* ∈ *keys-pp* (*rep-nat-pp s*) ∪ *keys-pp* (*rep-nat-pp t*)
proof (*rule ccontr*)
assume *x* ∉ *keys-pp* (*rep-nat-pp s*) ∪ *keys-pp* (*rep-nat-pp t*)
with *1* **show** *False* **by** (*simp add*: *keys-pp-iff*)
qed
next
show *comparator-of* (*lookup-pp* (*rep-nat-pp s*) *x*) (*lookup-pp* (*rep-nat-pp t*) *x*) = *lex-comp′ s t*
by (*simp add*: *linorder-class.comparator-of-def 1 Lt*)
qed (*fact 2*)
next
case *Gt*
hence ¬ *lex-pp* (*rep-nat-pp s*) (*rep-nat-pp t*)
by (*auto simp*: *lex-comp′ comp-of-ord-def split*: *if-split-asm*)
hence *lex-pp* (*rep-nat-pp t*) (*rep-nat-pp s*) **by** (*rule lex-pp-lin′*)
moreover have *rep-nat-pp t* ≠ *rep-nat-pp s*
proof
assume *rep-nat-pp t* = *rep-nat-pp s*
moreover from *this* **have** *lex-pp* (*rep-nat-pp s*) (*rep-nat-pp t*) **by** (*simp add*: *lex-pp-refl*)
ultimately have *lex-comp′ s t* = *Eq* **by** (*simp add*: *lex-comp′ comp-of-ord-def*)
with *Gt* **show** *False* **by** *simp*
qed
ultimately have ∃ *x*. *lookup-pp* (*rep-nat-pp t*) *x* < *lookup-pp* (*rep-nat-pp s*) *x* ∧
(∀ *y*<*x*. *lookup-pp* (*rep-nat-pp t*) *y* = *lookup-pp* (*rep-nat-pp s*) *y*)
by (*simp add*: *lex-pp-alt*)
then obtain *x* **where** *1*: *lookup-pp* (*rep-nat-pp t*) *x* < *lookup-pp* (*rep-nat-pp s*) *x*
and *2*: $\bigwedge y.\ y < x \Longrightarrow$ *lookup-pp* (*rep-nat-pp t*) *y* = *lookup-pp* (*rep-nat-pp s*) *y* **by** *blast*
show *?thesis*
proof
show *x* ∈ *keys-pp* (*rep-nat-pp s*) ∪ *keys-pp* (*rep-nat-pp t*)
proof (*rule ccontr*)
assume *x* ∉ *keys-pp* (*rep-nat-pp s*) ∪ *keys-pp* (*rep-nat-pp t*)
with *1* **show** *False* **by** (*simp add*: *keys-pp-iff*)
qed
next
from *1* **have** ¬ *lookup-pp* (*rep-nat-pp s*) *x* < *lookup-pp* (*rep-nat-pp t*) *x*
and *lookup-pp* (*rep-nat-pp s*) *x* ≠ *lookup-pp* (*rep-nat-pp t*) *x* **by** *simp-all*
thus *comparator-of* (*lookup-pp* (*rep-nat-pp s*) *x*) (*lookup-pp* (*rep-nat-pp t*) *x*) = *lex-comp′ s t*

by (*simp add*: *linorder-class.comparator-of-def Gt*)
qed (*simp add*: *2*)
qed

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 −
from *assms* **have** *1*: *rep-nat-term x* ≤ *rep-nat-term y* **and** *2*: *rep-nat-term x* ≠ *rep-nat-term y* **by** *simp-all*
from *1* **have** *x* ≤ *y* **by** (*rule le-term*)
moreover from *2* **have** *x* ≠ *y* **by** *auto*
ultimately show *?thesis* **by** *simp*
qed

lemma *lex-comp-alt*: *lex-comp* = (*comparator-of*::*'a comparator*)
proof −
from *lex-pp-antisym* **have** *as*: *antisymp lex-pp* **by** (*rule antisympI*)
interpret *lex*: *comparator comp-of-ord* (*lex-pp*::(*nat*, *nat*) *pp* ⇒ -)
unfolding *comp-of-ord-eq-comp-of-ords*[*OF as*]
by (*rule comp-of-ords*, *unfold-locales*,
auto simp: *lex-pp-refl intro*: *lex-pp-trans lex-pp-lin' elim*!: *lex-pp-antisym*)

have *1*: *x* = *y* **if** *fst* (*rep-nat-term x*) = *fst* (*rep-nat-term y*)
and *snd* (*rep-nat-term x*) = *snd* (*rep-nat-term y*) **for** *x y*
by (*rule rep-nat-term-inj*, *rule prod-eqI*, *fact*+)
have *2*: *x* < *y* **if** *fst* (*rep-nat-term x*) = *fst* (*rep-nat-term y*)
and *snd* (*rep-nat-term x*) < *snd* (*rep-nat-term y*) **for** *x y*

by (*rule less-term*, *simp add*: *less-prod-def that*)
have *3*: *False* **if** *fst* (*rep-nat-term x*) = *fst* (*rep-nat-term y*)
and ¬ *snd* (*rep-nat-term x*) < *snd* (*rep-nat-term y*) **and** *x* < *y* **for** *x* *y*
proof −
from *that*(*2*) **have** *a*: *snd* (*rep-nat-term y*) ≤ *snd* (*rep-nat-term x*) **by** *simp*
have *y* ≤ *x* **by** (*rule le-term*, *simp add*: *less-eq-prod-def that*(*1*) *a*)
also have ... < *y* **by** *fact*
finally show *False* **..**
qed
have *4*: *x* < *y* **if** *fst* (*rep-nat-term x*) ≠ *fst* (*rep-nat-term y*)
and *lex-pp* (*fst* (*rep-nat-term x*)) (*fst* (*rep-nat-term y*)) **for** *x y*
proof −
from *that*(*2*) **have** *fst* (*rep-nat-term x*) ≤ *fst* (*rep-nat-term y*) **by** (*simp only*: *less-eq-pp-def*)
with *that*(*1*) **have** *fst* (*rep-nat-term x*) < *fst* (*rep-nat-term y*) **by** *simp*
hence *rep-nat-term x* < *rep-nat-term y* **by** (*simp add*: *less-prod-def*)
thus *?thesis* **by** (*rule less-term*)
qed
have *5*: *False* **if** *fst* (*rep-nat-term x*) ≠ *fst* (*rep-nat-term y*)
and ¬ *lex-pp* (*fst* (*rep-nat-term x*)) (*fst* (*rep-nat-term y*)) **and** *x* < *y* **for** *x y*
proof −
from *that*(*2*) **have** *a*: *lex-pp* (*fst* (*rep-nat-term y*)) (*fst* (*rep-nat-term x*)) **by** (*rule lex-pp-lin′*)
with *that*(*1*)[*symmetric*] **have** *y* < *x* **by** (*rule 4*)
also have ... < *y* **by** *fact*
finally show *False* **..**
qed

show *?thesis*
by (*intro ext*, *simp add*: *lex-comp lex-comp-aux-def comparator-of-def linorder-class.comparator-of-def lex.eq split*: *order.splits*,
auto simp: *lex-pp-refl comp-of-ord-def elim*: *1 2 3 4 5*)
qed

lemma *full-component-zeroE*: **obtains** *x* **where** *rep-nat-term x* = (*t*, *0*)
proof −
from *zero-component* **obtain** *x′* **where** *snd* (*rep-nat-term x′*) = *0* **..**
then obtain *x* **where** *rep-nat-term x* = (*t*, *0*) **by** (*rule full-componentE*)
thus *?thesis* **..**
qed

end

lemma *comparator-lex-comp*: *comparator lex-comp*
unfolding *lex-comp* **by** (*fact comparator-lex-comp-aux*)

lemma *nat-term-comp-lex-comp*: *nat-term-comp lex-comp*
unfolding *lex-comp* **by** (*fact nat-term-comp-lex-comp-aux*)

lemma *comparator-deg-comp*:
assumes *comparator cmp*
shows *comparator* (*deg-comp cmp*)
unfolding *deg-comp* **using** *comparator-of assms* **by** (*rule comparator-lexicographic*)

lemma *comparator-pot-comp*:
assumes *comparator cmp*
shows *comparator* (*pot-comp cmp*)
unfolding *pot-comp* **using** *comparator-of assms* **by** (*rule comparator-lexicographic*)

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 (*simp add*: *deg-comp assms*(*3*) *comparator-of-def split*: *order.split*, *intro impI*)
assume *fst* (*rep-nat-term v*) = *0*
with *assms*(*3*) **have** *fst* (*rep-nat-term u*) = *fst* (*rep-nat-term v*) **by** *simp*
hence *rep-nat-term u* = *rep-nat-term v* **using** *assms*(*2*) **by** (*rule prod-eqI*)
hence *u* = *v* **by** (*rule rep-nat-term-inj*)
from *assms*(*1*) **interpret** *c*: *comparator cmp* **.**
show *cmp u v* ≠ *Gt* **by** (*simp add*: ‹*u* = *v*›)
qed

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*
by (*simp add*: *deg-comp assms split*: *order.split*)

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*
using *assms*(*2*) **by** (*auto simp*: *deg-comp splus-term pprod.splus-def comparator-of-def deg-pp-plus*
split: *order.splits if-splits intro*: *assms*(*1*))

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*
by (*simp add*: *pot-comp comparator-of-def assms split*: *order.split*)

lemma *pot-comp-pos*:
assumes *snd* (*rep-nat-term u*) < *snd* (*rep-nat-term v*)
shows *pot-comp cmp u v* = *Lt*

by (*simp add*: *pot-comp comparator-of-def assms split*: *order.split*)

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*
using *assms*(*2*) **by** (*auto simp*: *pot-comp splus-term pprod.splus-def comparator-of-def deg-pp-plus*
split: *order.splits if-splits intro*: *assms*(*1*))

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*
using *assms* **by** (*simp add*: *deg-comp comparator-of-def split*: *order.split*)

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*
using *assms* **by** (*simp add*: *pot-comp comparator-of-def split*: *order.split*)

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*)))
unfolding *rep-nat-pp-pp-def*
proof (*rule lookup-pp-of-fun*)
have {*n*. *lookup-pp t* (*abs-nat n*) ≠ *0*} ⊆ *rep-nat* ‘ {*x*. *lookup-pp t x* ≠ *0*}
proof
fix *n*
have *n* = *rep-nat* (*abs-nat n*) **by** (*simp only*: *nat-class.abs-inverse*)
assume *n* ∈ {*n*. *lookup-pp t* (*abs-nat n*) ≠ *0*}
hence *abs-nat n* ∈ {*x*. *lookup-pp t x* ≠ *0*} **by** *simp*
with ‹*n* = *rep-nat* (*abs-nat n*)› **show** *n* ∈ *rep-nat* ‘ {*x*. *lookup-pp t x* ≠ *0*} **..**

qed
also have *finite* ... **by** (*rule finite-imageI*, *transfer*, *simp*)
also (*finite-subset*) **have** {*n. lookup-pp t* (*abs-nat n*) ≠ *0*} = {*n. rep-nat* (*lookup-pp t* (*abs-nat n*)) ≠ *0*}
by (*metis rep-inj rep-zero*)
finally show *finite* {*x. rep-nat* (*lookup-pp t* (*abs-nat x*)) ≠ *0*} **.**
qed

lemma *lookup-abs-nat-pp-pp*:
lookup-pp (*abs-nat-pp t*) = (λ*n*::′*a. abs-nat* (*lookup-pp t* (*rep-nat n*)))
unfolding *abs-nat-pp-pp-def*
proof (*rule lookup-pp-of-fun*)
have {*n*::′*a. lookup-pp t* (*rep-nat n*) ≠ *0*} ⊆ *abs-nat* ‘ {*x. lookup-pp t x* ≠ *0*}
proof
fix *n* :: ′*a*
have *n* = *abs-nat* (*rep-nat n*) **by** (*simp only*: *nat-class.rep-inverse*)
assume *n* ∈ {*n. lookup-pp t* (*rep-nat n*) ≠ *0*}
hence *rep-nat n* ∈ {*x. lookup-pp t x* ≠ *0*} **by** *simp*
with ‹*n* = *abs-nat* (*rep-nat n*)› **show** *n* ∈ *abs-nat* ‘ {*x. lookup-pp t x* ≠ *0*} **..**
qed
also have *finite* ... **by** (*rule finite-imageI*, *transfer*, *simp*)
also (*finite-subset*) **have** {*n*::′*a. lookup-pp t* (*rep-nat n*) ≠ *0*} = {*n. abs-nat* (*lookup-pp t* (*rep-nat n*)) ≠ *0*}
by (*metis abs-inverse abs-zero*)
finally show *finite* {*n*::′*a. abs-nat* (*lookup-pp t* (*rep-nat n*)) ≠ *0*} **.**
qed

lemma *keys-rep-nat-pp-pp*: *keys-pp* (*rep-nat-pp t*) = *rep-nat* ‘ *keys-pp t*
by (*rule set-eqI*,
simp add: *keys-pp-iff lookup-rep-nat-pp-pp image-iff Bex-def ex-iff-abs*[**where** ′*a*=′*a*] *rep-zero-iff del*: *neq0-conv*)

lemma *rep-nat-pp-pp-inverse*: *abs-nat-pp* (*rep-nat-pp x*) = *x* **for** *x*::(′*a*, ′*b*) *pp*
by (*rule pp-eqI*, *simp add*: *lookup-abs-nat-pp-pp lookup-rep-nat-pp-pp*)

lemma *abs-nat-pp-pp-inverse*: *rep-nat-pp* ((*abs-nat-pp t*)::(′*a*, ′*b*) *pp*) = *t*
by (*rule pp-eqI*, *simp add*: *lookup-abs-nat-pp-pp lookup-rep-nat-pp-pp*)

corollary *rep-nat-pp-pp-inj*:
fixes *x y* :: (′*a*, ′*b*) *pp*
assumes *rep-nat-pp x* = *rep-nat-pp y*
shows *x* = *y*
by (*metis* (*no-types*) *rep-nat-pp-pp-inverse assms*)

corollary *rep-nat-pp-pp-eq-iff*: (*rep-nat-pp x* = *rep-nat-pp y*) ⟷ (*x* = *y*) **for** *x y* :: (′*a*, ′*b*) *pp*
by (*auto elim*: *rep-nat-pp-pp-inj*)

lemma *lex-rep-nat-pp*: *lex-pp* (*rep-nat-pp x*) (*rep-nat-pp y*) ⟷ *lex-pp x y*

by (*simp add*: *lex-pp-alt rep-nat-pp-pp-eq-iff lookup-rep-nat-pp-pp rep-eq-iff ord-iff*[*symmetric*] *ex-iff-abs*[**where** *'a='a*] *all-iff-abs'*)

corollary *lex-comp'-pp*: *lex-comp' x y = comp-of-ord lex-pp* (*rep-nat-pp x*) (*rep-nat-pp y*) **for** *x y* :: (*'a*, *'b*) *pp*
by (*simp add*: *lex-comp'-pp-def comp-of-ord-def rep-nat-pp-pp-eq-iff lex-rep-nat-pp*)

corollary *le-pp-pp*: *rep-nat-pp x* ≤ *rep-nat-pp y* ⟹ *x* ≤ *y* **for** *x y* :: (*'a*, *'b*) *pp*
by (*simp only*: *less-eq-pp-def lex-rep-nat-pp*)

lemma *deg-rep-nat-pp*: *deg-pp* (*rep-nat-pp t*) = *rep-nat* (*deg-pp t*) **for** *t* :: (*'a*, *'b*) *pp*
proof −
have *keys-pp* (*rep-nat-pp t*) = *rep-nat* ‘ *keys-pp t*
by (*rule set-eqI*, *simp add*: *keys-pp-iff image-iff lookup-rep-nat-pp-pp Bex-def ex-iff-abs*[**where** *'a='a*] *rep-zero-iff del*: *neq0-conv*)
hence *deg-pp* (*rep-nat-pp t*) = *sum* (*lookup-pp* (*rep-nat-pp t*)) (*rep-nat* ‘ *keys-pp t*)
by (*simp add*: *deg-pp-alt*)
also have ... = *sum* (*lookup-pp* (*rep-nat-pp t*) ∘ *rep-nat*) (*keys-pp t*)
by (*rule sum.reindex*, *rule inj-onI*, *elim rep-inj*)
also have ... = *sum* (*rep-nat* ∘ (*lookup-pp t*)) (*keys-pp t*)
by (*simp add*: *lookup-rep-nat-pp-pp*)
also have ... = *rep-nat* (*deg-pp t*) **by** (*simp only*: *deg-pp-alt sum-rep*)
finally show *?thesis* **.**
qed

corollary *deg'-pp*: *deg' t* = *deg-pp* (*rep-nat-pp t*) **for** *t* :: (*'a*, *'b*) *pp*
by (*simp add*: *deg'-pp-def deg-rep-nat-pp*)

lemma *zero-pp-pp*: *rep-nat-pp* (*0*::(*'a*, *'b*) *pp*) = *0*
by (*rule pp-eqI*, *simp add*: *lookup-rep-nat-pp-pp*)

lemma *plus-pp-pp*: *rep-nat-pp* (*x* + *y*) = *rep-nat-pp x* + *rep-nat-pp y*
for *x y* :: (*'a*, *'b*) *pp*
by (*rule pp-eqI*, *simp add*: *lookup-rep-nat-pp-pp lookup-plus-pp rep-plus*)

instance
apply *intro-classes*
subgoal by (*fact rep-nat-pp-pp-inverse*)
subgoal by (*fact abs-nat-pp-pp-inverse*)
subgoal by (*fact lex-comp'-pp*)
subgoal by (*fact deg'-pp*)
subgoal by (*rule le-pp-pp*)
subgoal by (*fact zero-pp-pp*)
subgoal by (*fact plus-pp-pp*)
done

end

instantiation *pp* :: (*nat*, *nat*) *nat-term*
begin

definition *rep-nat-term-pp* :: (′*a*, ′*b*) *pp* ⇒ (*nat*, *nat*) *pp* × *nat*
  where *rep-nat-term-pp-def* [*code del*]: *rep-nat-term-pp t* = (*rep-nat-pp t*, *0*)

definition *splus-pp* :: (′*a*, ′*b*) *pp* ⇒ (′*a*, ′*b*) *pp* ⇒ (′*a*, ′*b*) *pp*
  where *splus-pp-def* [*code del*]: *splus-pp* = (+)

instance proof
  fix *x y* :: (′*a*, ′*b*) *pp*
  assume *rep-nat-term x* = *rep-nat-term y*
  hence *rep-nat-pp x* = *rep-nat-pp y* **by** (*simp add*: *rep-nat-term-pp-def*)
  thus *x* = *y* **by** (*rule rep-nat-pp-pp-inj*)
next
  fix *x*::(′*a*, ′*b*) *pp* **and** *i t*
  assume *snd* (*rep-nat-term x*) = *i*
  hence *i* = *0* **by** (*simp add*: *rep-nat-term-pp-def*)
  show ∃ *y*::(′*a*, ′*b*) *pp*. *rep-nat-term y* = (*t*, *i*) **unfolding** ‹*i* = *0*›
  proof
   show *rep-nat-term* ((*abs-nat-pp t*)::(′*a*, ′*b*) *pp*) = (*t*, *0*) **by** (*simp add*: *rep-nat-term-pp-def*)
  qed
next
  fix *x y* :: (′*a*, ′*b*) *pp*
  show *rep-nat-term* (*splus x y*) = *pprod.splus* (*fst* (*rep-nat-term x*)) (*rep-nat-term y*)
    by (*simp add*: *splus-pp-def rep-nat-term-pp-def pprod.splus-def plus-pp-pp*)
qed

end

instantiation *pp* :: (*nat*, *nat*) *nat-term-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

show ∃ *x*::(′*a*, ′*b*) *pp*. *snd* (*rep-nat-term x*) = *0*
proof
show *snd* (*rep-nat-term* (*0*::(′*a*, ′*b*) *pp*)) = *0* **by** (*simp add*: *rep-nat-term-pp-def*)
qed
next
show *is-scalar* = (λ-::(′*a*, ′*b*) *pp itself*. ∀ *x*::(′*a*, ′*b*) *pp*. *snd* (*rep-nat-term x*) = *0*)
by (*simp add*: *is-scalar-pp-def rep-nat-term-pp-def*)
next
show *lex-comp* = (*lex-comp-aux*::(′*a*, ′*b*) *pp comparator*)
by (*auto simp*: *lex-comp-pp-def lex-comp-aux-def rep-nat-term-pp-def lex-comp*′*-pp split*: *order.split intro*!: *ext*)
next
fix *cmp* :: (′*a*, ′*b*) *pp comparator*
show *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*
| *Lt* ⇒ *Lt* | *Gt* ⇒ *Gt*)
by (*simp add*: *rep-nat-term-pp-def deg-comp-pp-def deg-rep-nat-pp comparator-of-rep*)
next
fix *cmp* :: (′*a*, ′*b*) *pp comparator*
show *pot-comp cmp* =
(λ*x y*. *case comparator-of* (*snd* (*rep-nat-term x*)) (*snd* (*rep-nat-term y*)) *of Eq* ⇒ *cmp x y* | *Lt* ⇒ *Lt* | *Gt* ⇒ *Gt*)
by (*simp add*: *rep-nat-term-pp-def pot-comp-pp-def*)
next
fix *x y* :: (′*a*, ′*b*) *pp*
assume *rep-nat-term x* ≤ *rep-nat-term y*
hence *rep-nat-pp x* ≤ *rep-nat-pp y* **by** (*auto simp*: *rep-nat-term-pp-def*)
thus *x* ≤ *y* **by** (*rule le-pp-pp*)
qed

end

instance *pp* :: (*nat*, *nat*) *nat-pp-term*
proof
show *rep-nat-term* (*0*::(′*a*, ′*b*) *pp*) = (*0*, *0*)
by (*simp add*: *rep-nat-term-pp-def*) (*metis deg-pp-eq-0-iff deg-rep-nat-pp rep-zero*)
next
show *splus* = ((+)::(′*a*, ′*b*) *pp* ⇒ -) **by** (*simp add*: *splus-pp-def*)
qed

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
fix *x y* :: '*a* × '*b*
assume *rep-nat-term x* = *rep-nat-term y*
hence *1*: *rep-nat-pp* (*fst x*) = *rep-nat-pp* (*fst y*) **and** *2*: *rep-nat* (*snd x*) = *rep-nat* (*snd y*)
by (*simp-all add*: *rep-nat-term-prod-def*)
from *1* **have** *fst x* = *fst y* **by** (*rule rep-nat-pp-inj*)
moreover from *2* **have** *snd x* = *snd y* **by** (*rule rep-inj*)
ultimately show *x* = *y* **by** (*rule prod-eqI*)
next
fix *i t*
show ∃ *y*::'*a* × '*b*. *rep-nat-term y* = (*t*, *i*)
proof
show *rep-nat-term* (*abs-nat-pp t*, *abs-nat i*) = (*t*, *i*) **by** (*simp add*: *rep-nat-term-prod-def*)
qed
next
fix *x y* :: '*a* × '*b*
show *rep-nat-term* (*splus x y*) = *pprod.splus* (*fst* (*rep-nat-term x*)) (*rep-nat-term y*)
by (*simp add*: *splus-prod-def rep-nat-term-prod-def pprod.splus-def plus-pp*)
qed

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
show ∃ *x*::'*a* × '*b*. *snd* (*rep-nat-term x*) = *0*
proof

show *snd* (*rep-nat-term* (*abs-nat-pp 0*, *0*)) = *0* **by** (*simp add*: *rep-nat-term-prod-def*)
qed
next
have ¬ (∀ *a*. *rep-nat* (*a*::′*b*) = *0*)
proof
assume ∀ *a*. *rep-nat* (*a*::′*b*) = *0*
hence *rep-nat* ((*abs-nat 1*)::′*b*) = *0* **by** *blast*
hence ((*abs-nat 1*)::′*b*) = *0* **by** (*simp only*: *rep-zero-iff*)
hence (*1*::*nat*) = *0* **by** (*metis abs-inj abs-zero*)
thus *False* **by** *simp*
qed
thus *is-scalar* = (λ-::(′*a* × ′*b*) *itself*. ∀ *x*. *snd* (*rep-nat-term* (*x*::′*a* × ′*b*)) = *0*)
by (*auto simp add*: *is-scalar-prod-def rep-nat-term-prod-def intro*!: *ext*)
next
show *lex-comp* = (*lex-comp-aux*::(′*a* × ′*b*) *comparator*)
by (*auto simp*: *lex-comp-prod-def lex-comp-aux-def rep-nat-term-prod-def lex-comp′ comparator-of-rep split*: *order.split intro*!: *ext*)
next
fix *cmp* :: (′*a* × ′*b*) *comparator*
show *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*
| *Lt* ⇒ *Lt* | *Gt* ⇒ *Gt*)
by (*simp add*: *rep-nat-term-prod-def deg-comp-prod-def deg′*)
next
fix *cmp* :: (′*a* × ′*b*) *comparator*
show *pot-comp cmp* =
(λ*x y*. *case comparator-of* (*snd* (*rep-nat-term x*)) (*snd* (*rep-nat-term y*)) *of Eq* ⇒ *cmp x y* | *Lt* ⇒ *Lt* | *Gt* ⇒ *Gt*)
by (*simp add*: *rep-nat-term-prod-def pot-comp-prod-def comparator-of-rep*)
next
fix *x y* :: ′*a* × ′*b*
assume *rep-nat-term x* ≤ *rep-nat-term y*
hence *rep-nat-pp* (*fst x*) < *rep-nat-pp* (*fst y*) ∨ (*rep-nat-pp* (*fst x*) ≤ *rep-nat-pp* (*fst y*) ∧ *rep-nat* (*snd x*) ≤ *rep-nat* (*snd y*))
by (*simp add*: *rep-nat-term-prod-def*)
thus *x* ≤ *y* **by** (*auto simp*: *less-eq-prod-def ord-iff*[*symmetric*] *intro*: *le-pp less-pp*)
qed

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*)
by (*intro ext pp-eqI*, *simp add*: *lookup-rep-nat-pp-pp abs-nat-nat-def rep-nat-nat-def*)

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*
unfolding *LEX-def* **using** *comparator-lex-comp nat-term-comp-lex-comp*
by (*rule nat-term-compare-Abs-nat-term-order-id*)

lemma *nat-term-compare-DRLEX* [*code*]: *nat-term-compare DRLEX* = *deg-comp* (*pot-comp* (λ*x y*. *lex-comp y x*))
proof −
have *cmp*: *comparator* (*pot-comp* (λ*x y*. *lex-comp y x*))
by (*rule comparator-pot-comp*, *rule comparator-converse*, *fact comparator-lex-comp*)
show *?thesis* **unfolding** *DRLEX-def*
proof (*rule nat-term-compare-Abs-nat-term-order-id*)
from *cmp* **show** *comparator* (*deg-comp* (*pot-comp* (λ*x y*::*'a*. *lex-comp y x*)))
by (*rule comparator-deg-comp*)
next
show *nat-term-comp* (*deg-comp* (*pot-comp* (λ*x y*::*'a*. *lex-comp y x*)))
proof (*rule nat-term-compI*)
fix *u v* :: *'a*
assume *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*) **and** *fst* (*rep-nat-term u*) = *0*
with *cmp* **show** *deg-comp* (*pot-comp* (λ*x y*::*'a*. *lex-comp y x*)) *u v* ≠ *Gt*
by (*rule deg-comp-zero-min*)
next
fix *u v* :: *'a*

assume *snd* (*rep-nat-term u*) < *snd* (*rep-nat-term v*)
hence *pot-comp* (*λx y. lex-comp y x*) *u v* = *Lt* **by** (*rule pot-comp-pos*)
moreover assume *fst* (*rep-nat-term u*) = *fst* (*rep-nat-term v*)
ultimately show *deg-comp* (*pot-comp* (*λx y. lex-comp y x*)) *u v* = *Lt* **by** (*rule deg-comp-pos*)
next
fix *t u v* :: *'a*
have *pot-comp* (*λx y. lex-comp y x*) (*splus t u*) (*splus t v*) = *Lt*
if *pot-comp* (*λx y. lex-comp y x*) *u v* = *Lt* **using** - *that*
proof (*rule pot-comp-monotone*)
assume *lex-comp v u* = *Lt*
with *nat-term-comp-lex-comp* **show** *lex-comp* (*splus t v*) (*splus t u*) = *Lt*
by (*rule nat-term-compD3*)
qed
moreover assume *deg-comp* (*pot-comp* (*λx y. lex-comp y x*)) *u v* = *Lt*
ultimately show *deg-comp* (*pot-comp* (*λx y. lex-comp y x*)) (*splus t u*) (*splus t v*) = *Lt*
by (*rule deg-comp-monotone*)
next
fix *u v a b* :: *'a*
assume *fst* (*rep-nat-term v*) = *fst* (*rep-nat-term b*) **and** *fst* (*rep-nat-term u*) = *fst* (*rep-nat-term a*)
and *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*) **and** *snd* (*rep-nat-term a*) = *snd* (*rep-nat-term b*)
moreover from *comparator-lex-comp nat-term-comp-lex-comp this*(*1*, *2*) *this*(*3*, *4*)[*symmetric*]
have *lex-comp v u* = *lex-comp b a* **by** (*rule nat-term-compD4'*)
moreover assume *deg-comp* (*pot-comp* (*λx y. lex-comp y x*)) *a b* = *Lt*
ultimately show *deg-comp* (*pot-comp* (*λx y. lex-comp y x*)) *u v* = *Lt*
by (*simp add*: *deg-comp pot-comp split*: *order.splits*)
qed
qed
qed

lemma *nat-term-compare-DEG* [*code*]: *nat-term-compare* (*DEG to*) = *deg-comp* (*nat-term-compare to*)
unfolding *DEG-def*
proof (*rule nat-term-compare-Abs-nat-term-order-id*)
from *comparator-nat-term-compare* **show** *comparator* (*deg-comp* (*nat-term-compare to*))
by (*rule comparator-deg-comp*)
next
show *nat-term-comp* (*deg-comp* (*nat-term-compare to*))
proof (*rule nat-term-compI*)
fix *u v* :: *'a*
assume *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*) **and** *fst* (*rep-nat-term u*) = *0*
with *comparator-nat-term-compare* **show** *deg-comp* (*nat-term-compare to*) *u v* ≠ *Gt*

by (*rule deg-comp-zero-min*)
next
fix *u v* :: *'a*
assume *a*: *fst* (*rep-nat-term u*) = *fst* (*rep-nat-term v*) **and** *snd* (*rep-nat-term u*) < *snd* (*rep-nat-term v*)
with *nat-term-comp-nat-term-compare* **have** *nat-term-compare to u v* = *Lt* **by** (*rule nat-term-compD2*)
thus *deg-comp* (*nat-term-compare to*) *u v* = *Lt* **using** *a* **by** (*rule deg-comp-pos*)
next
fix *t u v* :: *'a*
from *nat-term-comp-nat-term-compare*
have *nat-term-compare to u v* = *Lt* ⟹ *nat-term-compare to* (*splus t u*) (*splus t v*) = *Lt*
by (*rule nat-term-compD3*)
moreover assume *deg-comp* (*nat-term-compare to*) *u v* = *Lt*
ultimately show *deg-comp* (*nat-term-compare to*) (*splus t u*) (*splus t v*) = *Lt* **by** (*rule deg-comp-monotone*)
next
fix *u v a b* :: *'a*
assume *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*)
moreover from *comparator-nat-term-compare nat-term-comp-nat-term-compare this*
have *nat-term-compare to u v* = *nat-term-compare to a b*
by (*rule nat-term-compD4'*)
moreover assume *deg-comp* (*nat-term-compare to*) *a b* = *Lt*
ultimately show *deg-comp* (*nat-term-compare to*) *u v* = *Lt*
by (*simp add*: *deg-comp split*: *order.splits*)
qed
qed

lemma *nat-term-compare-POT* [*code*]: *nat-term-compare* (*POT to*) = *pot-comp* (*nat-term-compare to*)
unfolding *POT-def*
proof (*rule nat-term-compare-Abs-nat-term-order-id*)
from *comparator-nat-term-compare* **show** *comparator* (*pot-comp* (*nat-term-compare to*))
by (*rule comparator-pot-comp*)
next
show *nat-term-comp* (*pot-comp* (*nat-term-compare to*))
proof (*rule nat-term-compI*)
fix *u v* :: *'a*
assume *a*: *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*) **and** *fst* (*rep-nat-term u*) = *0*
with *nat-term-comp-nat-term-compare* **have** *nat-term-compare to u v* ≠ *Gt* **by** (*rule nat-term-compD1*)
thus *pot-comp* (*nat-term-compare to*) *u v* ≠ *Gt* **using** *a* **by** (*rule pot-comp-zero-min*)

next
fix *u v* :: *'a*
assume *snd* (*rep-nat-term u*) < *snd* (*rep-nat-term v*)
thus *pot-comp* (*nat-term-compare to*) *u v* = *Lt* **by** (*rule pot-comp-pos*)
next
fix *t u v* :: *'a*
from *nat-term-comp-nat-term-compare*
have *nat-term-compare to u v* = *Lt* ⟹ *nat-term-compare to* (*splus t u*) (*splus t v*) = *Lt*
by (*rule nat-term-compD3*)
moreover assume *pot-comp* (*nat-term-compare to*) *u v* = *Lt*
ultimately show *pot-comp* (*nat-term-compare to*) (*splus t u*) (*splus t v*) = *Lt* **by** (*rule pot-comp-monotone*)
next
fix *u v a b* :: *'a*
assume *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*)
moreover from *comparator-nat-term-compare nat-term-comp-nat-term-compare this*
have *nat-term-compare to u v* = *nat-term-compare to a b*
by (*rule nat-term-compD4'*)
moreover assume *pot-comp* (*nat-term-compare to*) *a b* = *Lt*
ultimately show *pot-comp* (*nat-term-compare to*) *u v* = *Lt*
by (*simp add*: *pot-comp split*: *order.splits*)
qed
qed

lemma *nat-term-compare-POT-DRLEX* [*code*]:
nat-term-compare (*POT DRLEX*) = *pot-comp* (*deg-comp* (λ*x y*. *lex-comp y x*))
unfolding *nat-term-compare-POT nat-term-compare-DRLEX*
by (*intro ext pot-comp-cong deg-comp-cong*, *simp add*: *pot-comp*)

lemma *compute-lex-pp* [*code*]: *lex-pp p q* = (*lex-comp' p q* ≠ *Gt*)
by (*simp add*: *lex-comp'-pp-def comp-of-ord-def*)

lemma *compute-dlex-pp* [*code*]: *dlex-pp p q* = (*deg-comp lex-comp' p q* ≠ *Gt*)
by (*simp add*: *deg-comp-pp-def dlex-pp-alt compute-lex-pp comparator-of-def*)

lemma *compute-drlex-pp* [*code*]: *drlex-pp p q* = (*deg-comp* (λ*x y*. *lex-comp' y x*) *p q* ≠ *Gt*)
by (*simp add*: *deg-comp-pp-def drlex-pp-alt compute-lex-pp comparator-of-def*)

lemma *nat-pp-order-of-le-nat-pp* [*code*]: *nat-term-order-of-le* = *LEX*
by (*simp add*: *nat-term-order-of-le-def LEX-def lex-comp-alt*)

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*
unfolding *nat-term-order-eq-def* **using** *assms* **by** *blast*

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*
using *assms* **unfolding** *nat-term-order-eq-def* **by** *blast*

lemma *nat-term-order-eq-sym*: *nat-term-order-eq to1 to2 dg ps* ⟷ *nat-term-order-eq to2 to1 dg ps*
by (*auto simp*: *nat-term-order-eq-def*)

lemma *nat-term-order-eq-DEG-dg*:
nat-term-order-eq (*DEG to1*) *to2 True ps* ⟷ *nat-term-order-eq to1 to2 True ps*
by (*auto simp*: *nat-term-order-eq-def nat-term-compare-DEG deg-comp*)

lemma *nat-term-order-eq-DEG-dg'*:
nat-term-order-eq to1 (*DEG to2*) *True ps* ⟷ *nat-term-order-eq to1 to2 True ps*
by (*simp add*: *nat-term-order-eq-sym*[*of to1*] *nat-term-order-eq-DEG-dg*)

lemma *nat-term-order-eq-POT-ps*:
assumes *ps* ∨ *is-scalar TYPE*(*'a::nat-term-compare*)
shows *nat-term-order-eq* (*POT* (*to1*::*'a nat-term-order*)) *to2 dg ps* ⟷ *nat-term-order-eq to1 to2 dg ps*
using *assms*
proof
assume *ps*
thus *?thesis* **by** (*auto simp*: *nat-term-order-eq-def nat-term-compare-POT pot-comp*)
next
assume *is-scalar TYPE*(*'a*)
hence *snd* (*rep-nat-term x*) = *0* **for** *x*::*'a* **by** (*simp add*: *is-scalar*)
thus *?thesis* **by** (*auto simp*: *nat-term-order-eq-def nat-term-compare-POT pot-comp*)

qed

lemma *nat-term-order-eq-POT-ps′*:
  assumes *ps* ∨ *is-scalar TYPE(′a::nat-term-compare)*
  shows *nat-term-order-eq to1 (POT (to2::′a nat-term-order)) dg ps* ⟷ *nat-term-order-eq to1 to2 dg ps*
  using *assms* **by** (*simp add*: *nat-term-order-eq-sym*[*of to1*] *nat-term-order-eq-POT-ps*)

lemma *snd-rep-nat-term-eqI*:
  assumes *ps* ∨ *is-scalar TYPE(′a::nat-term-compare)* **and** *ps* ⟹ *snd (rep-nat-term (u::′a)) = snd (rep-nat-term (v::′a))*
  shows *snd (rep-nat-term u) = snd (rep-nat-term v)*
  using *assms(1)*
proof
  assume *is-scalar TYPE(′a)*
  thus *?thesis* **by** (*simp add*: *is-scalar*)
qed (*fact assms(2)*)

definition *of-exps* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *′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 (∃ 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 (∃ y::′a. snd (rep-nat-term y) = i) then i else 0*)
proof (*cases* ∃ *y::′a. snd (rep-nat-term y) = i*)
  case *True*
  then obtain *y::′a* **where** *snd (rep-nat-term y) = i* **..**
  then obtain *u::′a* **where** *u*: *rep-nat-term u = (pp-of-fun (λx::nat. if x = 0 then a else if x = 1 then b else 0), i)*
    by (*rule full-componentE*)
  from *True* **have** *eq*: (*if ∃ y::′a. snd (rep-nat-term y) = i then i else 0*) = *i* **by** *simp*
  show *?thesis* **unfolding** *of-exps-def eq*
  proof (*rule theI*)
    fix *v* :: *′a*
    assume *rep-nat-term v = (pp-of-fun (λx::nat. if x = 0 then a else if x = 1 then b else 0), i)*
    thus *v = u* **unfolding** *u*[*symmetric*] **by** (*rule rep-nat-term-inj*)
  qed (*fact u*)
next
  case *False*
  hence *eq*: (*if ∃ y::′a. snd (rep-nat-term y) = i then i else 0*) = *0* **by** *simp*
  obtain *u::′a* **where** *u*: *rep-nat-term u = (pp-of-fun (λx::nat. if x = 0 then a else*

if x = 1 then b else 0), *0*)
    by (*rule full-component-zeroE*)
  show *?thesis* **unfolding** *of-exps-def eq*
  proof (*rule theI*)
    fix *v* :: *'a*
    assume *rep-nat-term v* = (*pp-of-fun* (λ*x*::*nat*. *if x = 0 then a else if x = 1 then b else 0*), *0*)
    thus *v* = *u* **unfolding** *u*[*symmetric*] **by** (*rule rep-nat-term-inj*)
  qed (*fact u*)
qed

lemma *lookup-pp-of-exps*:
  lookup-pp (*fst* (*rep-nat-term* (*of-exps a b i*))) = (λ*x*. *if x = 0 then a else if x = 1 then b else 0*)
  unfolding *rep-nat-term-of-exps fst-conv*
proof (*rule lookup-pp-of-fun*)
  have {*x*. (*if x = 0 then a else if x = 1 then b else 0*) ≠ *0*} ⊆ {*0*, *1*}
    by (*rule*, *simp split*: *if-split-asm*)
  also have *finite* ... **by** *simp*
  finally(*finite-subset*) **show** *finite* {*x*. (*if x = 0 then a else if x = 1 then b else 0*) ≠ *0*} **.**
qed

lemma *keys-pp-of-exps*: *keys-pp* (*fst* (*rep-nat-term* (*of-exps a b i*))) ⊆ {*0*, *1*}
  by (*rule*, *simp add*: *keys-pp-iff lookup-pp-of-exps split*: *if-split-asm*)

lemma *deg-pp-of-exps* [*simp*]: *deg-pp* (*fst* (*rep-nat-term* ((*of-exps a b i*)::*'a*::*nat-term-compare*))) = *a* + *b*
proof −
  let *?u* = (*of-exps a b i*)::*'a*
  have *sum* (*lookup-pp* (*fst* (*rep-nat-term ?u*))) (*keys-pp* (*fst* (*rep-nat-term ?u*))) =
      sum (*lookup-pp* (*fst* (*rep-nat-term ?u*))) {*0*, *1*}
  proof (*rule sum.mono-neutral-left*, *simp*, *fact keys-pp-of-exps*, *intro ballI*)
    fix *x*
    assume *x* ∈ {*0*, *1*} − *keys-pp* (*fst* (*rep-nat-term ?u*))
    thus *lookup-pp* (*fst* (*rep-nat-term ?u*)) *x* = *0* **by** (*simp add*: *keys-pp-iff*)
  qed
  also have ... = *a* + *b* **by** (*simp add*: *lookup-pp-of-exps*)
  finally show *?thesis* **by** (*simp only*: *deg-pp-alt*)
qed

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 −
  from *assms* **have** ∃ *x*::*'a*. *snd* (*rep-nat-term* (*x*::*'a*)) = *i* **..**
  thus *?thesis* **by** (*simp add*: *rep-nat-term-of-exps*)
qed

lemma *snd-of-exps-zero* [*simp*]: *snd* (*rep-nat-term* ((*of-exps a b 0*)::'*a*::*nat-term-compare*)) = *0*
proof −
  from *zero-component* **obtain** *x*::'*a* **where** *snd* (*rep-nat-term* (*x*::'*a*)) = *0* **..**
  thus *?thesis* **by** (*rule snd-of-exps*)
qed

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 −
  have *a1* = *a2* ∧ *b1* = *b2*
    if (λ*x*::*nat*. *if x* = *0 then a1 else if x* = *1 then b1 else 0*) = (λ*x*. *if x* = *0 then a2 else if x* = *1 then b2 else 0*)
  proof
    from *fun-cong*[*OF that*, *of 0*] **show** *a1* = *a2* **by** *simp*
  next
    from *fun-cong*[*OF that*, *of 1*] **show** *b1* = *b2* **by** *simp*
  qed
  thus *?thesis* **by** (*auto simp*: *pp-eq-iff lookup-pp-of-exps*)
qed

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 −
  let *?u* = *fst* (*rep-nat-term* ((*of-exps a1 b1 i*)::'*a*))
  let *?v* = *fst* (*rep-nat-term* ((*of-exps a2 b2 j*)::'*a*))
  show *?thesis*
  proof
    assume *?L*
    hence *?u* = *?v* ∨ (∃ *x*. *lookup-pp ?u x* < *lookup-pp ?v x* ∧ (∀ *y*<*x*. *lookup-pp ?u y* = *lookup-pp ?v y*))
      by (*simp only*: *lex-pp-alt*)
    thus *?R*
    proof
      assume *?u* = *?v*
      thus *?thesis* **by** (*simp add*: *eq-of-exps*)
    next
      assume ∃ *x*. *lookup-pp ?u x* < *lookup-pp ?v x* ∧ (∀ *y*<*x*. *lookup-pp ?u y* = *lookup-pp ?v y*)
      then obtain *x* **where** *1*: *lookup-pp ?u x* < *lookup-pp ?v x* **and** *2*: ⋀*y*. *y* < *x* ⟹ *lookup-pp ?u y* = *lookup-pp ?v y*
        by *auto*
      from *1* **have** *lookup-pp ?v x* ≠ *0* **by** *simp*
      hence *x* ∈ *keys-pp ?v* **by** (*simp add*: *keys-pp-iff*)
      also have ... ⊆ {*0*, *1*} **by** (*fact keys-pp-of-exps*)

finally have *x* = *0* ∨ *x* = *1* **by** *simp*
thus *?thesis*
proof
assume *x* = *0*
from *1* **show** *?thesis* **by** (*simp add*: *lookup-pp-of-exps* ‹*x* = *0*›)
next
assume *x* = *1*
hence *0* < *x* **by** *simp*
hence *lookup-pp ?u 0* = *lookup-pp ?v 0* **by** (*rule 2*)
hence *a1* = *a2* **by** (*simp add*: *lookup-pp-of-exps*)
from *1* **show** *?thesis* **by** (*simp add*: *lookup-pp-of-exps* ‹*x* = *1*› ‹*a1* = *a2*›)
qed
qed
next
assume *?R*
thus *?L*
proof
assume *a1* < *a2*
show *?thesis* **unfolding** *lex-pp-alt*
proof (*intro disjI2 exI conjI allI impI*)
from ‹*a1* < *a2*› **show** *lookup-pp ?u 0* < *lookup-pp ?v 0* **by** (*simp add*: *lookup-pp-of-exps*)
next
fix *y*::*nat*
assume *y* < *0*
thus *lookup-pp ?u y* = *lookup-pp ?v y* **by** *simp*
qed
next
assume *a1* = *a2* ∧ *b1* ≤ *b2*
hence *a1* = *a2* **and** *b1* ≤ *b2* **by** *simp-all*
from *this*(*2*) **have** *b1* < *b2* ∨ *b1* = *b2* **by** *auto*
thus *?thesis*
proof
assume *b1* < *b2*
show *?thesis* **unfolding** *lex-pp-alt*
proof (*intro disjI2 exI conjI allI impI*)
from ‹*b1* < *b2*› **show** *lookup-pp ?u 1* < *lookup-pp ?v 1* **by** (*simp add*: *lookup-pp-of-exps*)
next
fix *y*::*nat*
assume *y* < *1*
hence *y* = *0* **by** *simp*
show *lookup-pp ?u y* = *lookup-pp ?v y* **by** (*simp add*: *lookup-pp-of-exps* ‹*y* = *0*› ‹*a1* = *a2*›)
qed
next
assume *b1* = *b2*
show *?thesis* **by** (*simp add*: *lex-pp-alt eq-of-exps* ‹*a1* = *a2*› ‹*b1* = *b2*›)
qed

    qed
  qed
qed

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 −
  show *?thesis1* **by** (*simp add*: *nat-term-order-eq-def*)
next
  show *?thesis2*
  proof (*intro iffI*)
    assume *a*: *nat-term-order-eq LEX* (*DRLEX*::′*a nat-term-order*) *dg ps*
    let *?u* = (*of-exps 0 1 0*)::′*a*
    let *?v* = (*of-exps 1 0 0*)::′*a*
    have *nat-term-compare LEX ?u ?v* = *nat-term-compare DRLEX ?u ?v*
      by (*rule nat-term-order-eqD*, *fact a*, *simp-all*)
    thus *False*
     by (*simp add*: *nat-term-compare-LEX lex-comp lex-comp-aux-def nat-term-compare-DRLEX deg-comp*
          pot-comp comparator-of-def comp-of-ord-def lex-pp-of-exps eq-of-exps)
  qed (*rule FalseE*)
next
  show *?thesis3*
  proof (*intro iffI*)
    assume *a*: *nat-term-order-eq LEX* (*DEG to*) *dg ps*
    have *dg*
    proof (*rule ccontr*)
      assume ¬ *dg*
      let *?u* = (*of-exps 0 2 0*)::′*a*
      let *?v* = (*of-exps 1 0 0*)::′*a*
      have *nat-term-compare LEX ?u ?v* = *nat-term-compare* (*DEG to*) *?u ?v*
        by (*rule nat-term-order-eqD*, *fact a*, *simp-all add*: ‹¬ *dg*›)
      thus *False*
      by (*simp add*: *nat-term-compare-LEX lex-comp lex-comp-aux-def nat-term-compare-DEG deg-comp*
          comparator-of-def comp-of-ord-def lex-pp-of-exps eq-of-exps)
    qed
    show *dg* ∧ *nat-term-order-eq LEX to dg ps*
    proof (*intro conjI* ‹*dg*› *nat-term-order-eqI*)
      fix *u v* :: ′*a*
      assume *1*: *dg* ⟹ *deg-pp* (*fst* (*rep-nat-term u*)) = *deg-pp* (*fst* (*rep-nat-term v*))
      from ‹*dg*› **have** *eq*: *deg-pp* (*fst* (*rep-nat-term u*)) = *deg-pp* (*fst* (*rep-nat-term*

v)) **by** (*rule 1*)
assume *ps* ⟹ *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*)
with *a 1* **have** *nat-term-compare LEX u v* = *nat-term-compare* (*DEG to*) *u v*
by (*rule nat-term-order-eqD*)
also have ... = *nat-term-compare to u v* **by** (*simp add*: *nat-term-compare-DEG deg-comp eq*)
finally show *nat-term-compare LEX u v* = *nat-term-compare to u v* **.**
qed
next
assume *dg* ∧ *nat-term-order-eq LEX to dg ps*
hence *dg* **and** *a*: *nat-term-order-eq LEX to dg ps* **by** *auto*
show *nat-term-order-eq LEX* (*DEG to*) *dg ps*
proof (*rule nat-term-order-eqI*)
fix *u v* :: *'a*
assume *1*: *dg* ⟹ *deg-pp* (*fst* (*rep-nat-term u*)) = *deg-pp* (*fst* (*rep-nat-term v*))
from ‹*dg*› **have** *eq*: *deg-pp* (*fst* (*rep-nat-term u*)) = *deg-pp* (*fst* (*rep-nat-term v*)) **by** (*rule 1*)
assume *ps* ⟹ *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*)
with *a 1* **have** *nat-term-compare LEX u v* = *nat-term-compare to u v* **by** (*rule nat-term-order-eqD*)
also have ... = *nat-term-compare* (*DEG to*) *u v* **by** (*simp add*: *nat-term-compare-DEG deg-comp eq*)
finally show *nat-term-compare LEX u v* = *nat-term-compare* (*DEG to*) *u v* **.**
qed
qed
next
show *?thesis4*
proof (*intro iffI*)
assume *a*: *nat-term-order-eq LEX* (*POT to*) *dg ps*
have ∗: *ps* ∨ *is-scalar TYPE*(*'a*)
proof (*rule ccontr*)
assume ¬ (*ps* ∨ *is-scalar TYPE*(*'a*))
hence ¬ *ps* **and** ¬ *is-scalar TYPE*(*'a*) **by** *simp-all*
from *this*(*2*) **obtain** *x*::*'a* **where** *snd* (*rep-nat-term x*) ≠ *0* **unfolding** *is-scalar* **by** *auto*
moreover define *i*::*nat* **where** *i* = *snd* (*rep-nat-term x*)
ultimately have *i* ≠ *0* **by** *simp*
let *?u* = (*of-exps 0 1 i*)::*'a*
let *?v* = (*of-exps 1 0 0*)::*'a*
from *i-def*[*symmetric*] **have** *eq*: *snd* (*rep-nat-term ?u*) = *i* **by** (*rule snd-of-exps*)
have *nat-term-compare LEX ?u ?v* = *nat-term-compare* (*POT to*) *?u ?v*
by (*rule nat-term-order-eqD*, *fact a*, *simp-all add*: ‹¬ *ps*›)
thus *False*
by (*simp add*: *nat-term-compare-LEX lex-comp lex-comp-aux-def pot-comp nat-term-compare-POT*
comparator-of-def comp-of-ord-def lex-pp-of-exps eq-of-exps eq ‹*i* ≠ *0*› *del*: *One-nat-def*)
qed

show (*ps* ∨ *is-scalar TYPE*(′*a*)) ∧ *nat-term-order-eq LEX to dg ps*
proof (*intro conjI* ∗ *nat-term-order-eqI*)
fix *u v* :: ′*a*
assume *1*: *dg* ⟹ *deg-pp* (*fst* (*rep-nat-term u*)) = *deg-pp* (*fst* (*rep-nat-term v*))
assume *2*: *ps* ⟹ *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*)
with ∗ **have** *eq*: *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*) **by** (*rule snd-rep-nat-term-eqI*)
from *a 1 2* **have** *nat-term-compare LEX u v* = *nat-term-compare* (*POT to*) *u v*
by (*rule nat-term-order-eqD*)
also have ... = *nat-term-compare to u v* **by** (*simp add*: *nat-term-compare-POT eq pot-comp*)
finally show *nat-term-compare LEX u v* = *nat-term-compare to u v* **.**
qed
next
assume (*ps* ∨ *is-scalar TYPE*(′*a*)) ∧ *nat-term-order-eq LEX to dg ps*
hence ∗: *ps* ∨ *is-scalar TYPE*(′*a*) **and** *a*: *nat-term-order-eq LEX to dg ps* **by** *auto*
show *nat-term-order-eq LEX* (*POT to*) *dg ps*
proof (*rule nat-term-order-eqI*)
fix *u v* :: ′*a*
assume *1*: *dg* ⟹ *deg-pp* (*fst* (*rep-nat-term u*)) = *deg-pp* (*fst* (*rep-nat-term v*))
assume *2*: *ps* ⟹ *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*)
with ∗ **have** *eq*: *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*) **by** (*rule snd-rep-nat-term-eqI*)
from *a 1 2* **have** *nat-term-compare LEX u v* = *nat-term-compare to u v* **by** (*rule nat-term-order-eqD*)
also have ... = *nat-term-compare* (*POT to*) *u v* **by** (*simp add*: *nat-term-compare-POT eq pot-comp*)
finally show *nat-term-compare LEX u v* = *nat-term-compare* (*POT to*) *u v* **.**
qed
qed
qed

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 −
from *nat-term-order-eq-sym*[*of DRLEX*::′*a nat-term-order*] **show** *?thesis1* **by** (*simp only*: *LEX-eq*)
next
show *?thesis2* **by** (*simp add*: *nat-term-order-eq-def*)

next
  show *?thesis3*
  proof (*intro iffI*)
    assume *a*: *nat-term-order-eq DRLEX* (*DEG to*) *dg ps*
    show *nat-term-order-eq DRLEX to True ps*
    proof (*rule nat-term-order-eqI*)
      fix *u v* :: *'a*
      assume *1*: *True* $\Longrightarrow$ *deg-pp* (*fst* (*rep-nat-term u*)) = *deg-pp* (*fst* (*rep-nat-term v*))
        and *ps* $\Longrightarrow$ *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*)
      with *a* **have** *nat-term-compare DRLEX u v* = *nat-term-compare* (*DEG to*) *u v*
        by (*rule nat-term-order-eqD*, *blast+*)
      also have ... = *nat-term-compare to u v* **by** (*simp add*: *nat-term-compare-DEG deg-comp 1*)
      finally show *nat-term-compare DRLEX u v* = *nat-term-compare to u v* **.**
    qed
  next
    assume *a*: *nat-term-order-eq DRLEX to True ps*
    show *nat-term-order-eq DRLEX* (*DEG to*) *dg ps*
    proof (*rule nat-term-order-eqI*)
      fix *u v* :: *'a*
      assume *1*: *ps* $\Longrightarrow$ *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*)
      show *nat-term-compare DRLEX u v* = *nat-term-compare* (*DEG to*) *u v*
      proof (*simp add*: *nat-term-compare-DRLEX nat-term-compare-DEG deg-comp comparator-of-def split*: *order.split*, *rule*)
        assume *2*: *deg-pp* (*fst* (*rep-nat-term u*)) = *deg-pp* (*fst* (*rep-nat-term v*))
        with *a* **have** *nat-term-compare DRLEX u v* = *nat-term-compare to u v*
          using *1* **by** (*rule nat-term-order-eqD*)
        thus *pot-comp* (λ*x y*. *lex-comp y x*) *u v* = *nat-term-compare to u v*
          by (*simp add*: *nat-term-compare-DRLEX deg-comp 2*)
      qed
    qed
  qed
next
  show *?thesis4*
  proof (*intro iffI*)
    assume *a*: *nat-term-order-eq DRLEX* (*POT to*) *dg ps*
    have $*$: *dg* $\vee$ *ps* $\vee$ *is-scalar TYPE*(*'a*)
    proof (*rule ccontr*)
      assume $\neg$ (*dg* $\vee$ *ps* $\vee$ *is-scalar TYPE*(*'a*))
      hence $\neg$ *dg* **and** $\neg$ *ps* **and** $\neg$ *is-scalar TYPE*(*'a*) **by** *simp-all*
      from *this*(*3*) **obtain** *x*::*'a* **where** *snd* (*rep-nat-term x*) $\neq$ *0* **unfolding** *is-scalar* **by** *auto*
      moreover define *i*::*nat* **where** *i* = *snd* (*rep-nat-term x*)
      ultimately have *i* $\neq$ *0* **by** *simp*
      let *?u* = (*of-exps 1 0 i*)::*'a*
      let *?v* = (*of-exps 2 0 0*)::*'a*
    from *i-def*[*symmetric*] **have** *eq*: *snd* (*rep-nat-term ?u*) = *i* **by** (*rule snd-of-exps*)

have *nat-term-compare DRLEX ?u ?v* = *nat-term-compare* (*POT to*) *?u ?v*
by (*rule nat-term-order-eqD*, *fact a*, *simp-all add*: ‹¬ *ps*› ‹¬ *dg*›)
thus *False*
by (*simp add*: *nat-term-compare-DRLEX deg-comp pot-comp nat-term-compare-POT comparator-of-def eq* ‹*i* ≠ *0*› *del*: *One-nat-def*)
qed
show (*dg* ∨ *ps* ∨ *is-scalar TYPE*(′*a*)) ∧ *nat-term-order-eq DRLEX to dg True*
proof (*intro conjI* ∗ *nat-term-order-eqI*)
fix *u v* :: ′*a*
assume *1*: *dg* ⟹ *deg-pp* (*fst* (*rep-nat-term u*)) = *deg-pp* (*fst* (*rep-nat-term v*))
assume *2*: *True* ⟹ *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*)
from *a 1 2* **have** *nat-term-compare DRLEX u v* = *nat-term-compare* (*POT to*) *u v*
by (*rule nat-term-order-eqD*, *blast*+)
also have ... = *nat-term-compare to u v* **by** (*simp add*: *nat-term-compare-POT 2 pot-comp*)
finally show *nat-term-compare DRLEX u v* = *nat-term-compare to u v* **.**
qed
next
assume (*dg* ∨ *ps* ∨ *is-scalar TYPE*(′*a*)) ∧ *nat-term-order-eq DRLEX to dg True*
hence *disj*: *dg* ∨ *ps* ∨ *is-scalar TYPE*(′*a*) **and** *a*: *nat-term-order-eq DRLEX to dg True* **by** *auto*
show *nat-term-order-eq DRLEX* (*POT to*) *dg ps*
proof (*rule nat-term-order-eqI*)
fix *u v* :: ′*a*
assume *1*: *dg* ⟹ *deg-pp* (*fst* (*rep-nat-term u*)) = *deg-pp* (*fst* (*rep-nat-term v*))
assume *2*: *ps* ⟹ *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*)
from *disj* **show** *nat-term-compare DRLEX u v* = *nat-term-compare* (*POT to*) *u v*
proof
assume *dg*
hence *eq1*: *deg-pp* (*fst* (*rep-nat-term u*)) = *deg-pp* (*fst* (*rep-nat-term v*)) **by** (*rule 1*)
show *?thesis*
proof (*simp add*: *nat-term-compare-DRLEX deg-comp eq1 nat-term-compare-POT pot-comp comparator-of-def split*: *order.split*, *rule*)
assume *eq2*: *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*)
with *a 1* **have** *nat-term-compare DRLEX u v* = *nat-term-compare to u v* **by** (*rule nat-term-order-eqD*)
thus *lex-comp v u* = *nat-term-compare to u v*
by (*simp add*: *nat-term-compare-DRLEX deg-comp eq1 pot-comp eq2*)
qed
next
assume *ps* ∨ *is-scalar TYPE*(′*a*)
hence *eq*: *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*) **using** *2* **by** (*rule snd-rep-nat-term-eqI*)

with *a 1* **have** *nat-term-compare DRLEX u v = nat-term-compare to u v* **by** (*rule nat-term-order-eqD*)
also have ... = *nat-term-compare* (*POT to*) *u v* **by** (*simp add*: *nat-term-compare-POT pot-comp eq*)
finally show *?thesis* **.**
qed
qed
qed
qed

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 −
show *?thesis3*
proof (*rule iffI*)
assume *a*: *nat-term-order-eq* (*DEG to1*) (*DEG to2*) *dg ps*
show *nat-term-order-eq to1 to2 True ps*
proof (*rule nat-term-order-eqI*)
fix *u v* :: '*a*
assume *b*: *True* ⟹ *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*)
with *a* **have** *nat-term-compare* (*DEG to1*) *u v* = *nat-term-compare* (*DEG to2*) *u v*
by (*rule nat-term-order-eqD*, *blast*+)
thus *nat-term-compare to1 u v* = *nat-term-compare to2 u v*
by (*simp add*: *nat-term-compare-DEG deg-comp comparator-of-def b*)
qed
next
assume *a*: *nat-term-order-eq to1 to2 True ps*
show *nat-term-order-eq* (*DEG to1*) (*DEG to2*) *dg ps*
proof (*rule nat-term-order-eqI*)
fix *u v* :: '*a*
assume *b*: *ps* ⟹ *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*)
show *nat-term-compare* (*DEG to1*) *u v* = *nat-term-compare* (*DEG to2*) *u v*
proof (*simp add*: *nat-term-compare-DEG deg-comp comparator-of-def split*: *order.split*, *rule impI*)
assume *deg-pp* (*fst* (*rep-nat-term u*)) = *deg-pp* (*fst* (*rep-nat-term v*))
with *a* **show** *nat-term-compare to1 u v* = *nat-term-compare to2 u v* **using** *b* **by** (*rule nat-term-order-eqD*)

qed
qed
qed
next
show *?thesis4*
proof (*simp add*: *nat-term-order-eq-DEG-dg split*: *if-split*, *intro impI*)
show *nat-term-order-eq* (*DEG to1*) (*POT to2*) *False ps* =
((*ps* ∨ *is-scalar TYPE*(′*a*)) ∧ *nat-term-order-eq* (*DEG to1*) *to2 False ps*)
proof (*intro iffI*)
assume *a*: *nat-term-order-eq* (*DEG to1*) (*POT to2*) *False ps*
have ∗: *ps* ∨ *is-scalar TYPE*(′*a*)
proof (*rule ccontr*)
assume ¬ (*ps* ∨ *is-scalar TYPE*(′*a*))
hence ¬ *ps* **and** ¬ *is-scalar TYPE*(′*a*) **by** *simp-all*
from *this*(*2*) **obtain** *x*::′*a* **where** *snd* (*rep-nat-term x*) ≠ *0* **unfolding** *is-scalar* **by** *auto*
moreover define *i*::*nat* **where** *i* = *snd* (*rep-nat-term x*)
ultimately have *i* ≠ *0* **by** *simp*
let *?u* = (*of-exps 1 0 i*)::′*a*
let *?v* = (*of-exps 2 0 0*)::′*a*
from *i-def*[*symmetric*] **have** *eq*: *snd* (*rep-nat-term ?u*) = *i* **by** (*rule snd-of-exps*)
have *nat-term-compare* (*DEG to1*) *?u ?v* = *nat-term-compare* (*POT to2*) *?u ?v*
by (*rule nat-term-order-eqD*, *fact a*, *simp-all add*: ‹¬ *ps*›)
thus *False*
by (*simp add*: *nat-term-compare-DEG deg-comp pot-comp nat-term-compare-POT comparator-of-def comp-of-ord-def lex-pp-of-exps eq-of-exps eq* ‹*i* ≠ *0*› *del*: *One-nat-def*)
qed
moreover from *this a* **have** *nat-term-order-eq* (*DEG to1*) *to2 False ps* **by** (*simp add*: *nat-term-order-eq-POT-ps*′)
ultimately show (*ps* ∨ *is-scalar TYPE*(′*a*)) ∧ *nat-term-order-eq* (*DEG to1*) *to2 False ps* **..**
qed (*simp add*: *nat-term-order-eq-POT-ps*′)
qed
qed (*fact nat-term-order-eq-sym*)+

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 −

show *?thesis4*
proof (*rule iffI*)
assume *a*: *nat-term-order-eq* (*POT to1*) (*POT to2*) *dg ps*
show *nat-term-order-eq to1 to2 dg True*
proof (*rule nat-term-order-eqI*)
fix *u v* :: *'a*
assume *dg* ⟹ *deg-pp* (*fst* (*rep-nat-term u*)) = *deg-pp* (*fst* (*rep-nat-term v*))
and *b*: *True* ⟹ *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*)
with *a* **have** *nat-term-compare* (*POT to1*) *u v* = *nat-term-compare* (*POT to2*) *u v*
by (*rule nat-term-order-eqD*, *blast+*)
thus *nat-term-compare to1 u v* = *nat-term-compare to2 u v*
by (*simp add*: *nat-term-compare-POT pot-comp comparator-of-def b*)
qed
next
assume *a*: *nat-term-order-eq to1 to2 dg True*
show *nat-term-order-eq* (*POT to1*) (*POT to2*) *dg ps*
proof (*rule nat-term-order-eqI*)
fix *u v* :: *'a*
assume *b*: *dg* ⟹ *deg-pp* (*fst* (*rep-nat-term u*)) = *deg-pp* (*fst* (*rep-nat-term v*))
show *nat-term-compare* (*POT to1*) *u v* = *nat-term-compare* (*POT to2*) *u v*
proof (*simp add*: *nat-term-compare-POT pot-comp comparator-of-def split*: *order.split*, *rule impI*)
assume *snd* (*rep-nat-term u*) = *snd* (*rep-nat-term v*)
with *a b* **show** *nat-term-compare to1 u v* = *nat-term-compare to2 u v* **by** (*rule nat-term-order-eqD*)
qed
qed
qed
qed (*fact nat-term-order-eq-sym*)+

lemma *nat-term-order-equal* [*code*]: *HOL.equal to1 to2* = *nat-term-order-eq to1 to2 False False*
by (*auto simp*: *nat-term-order-eq-def equal-eq nat-term-compare-inject*[*symmetric*])

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 by *standard* (*auto simp*: *equal-pp-def intro*: *pp-eqI*)

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 by *standard* (*auto simp*: *equal-poly-mapping-def intro*: *poly-mapping-eqI*)

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*
  unfolding *PP-oalist-def*
proof (*rule lookup-pp-of-fun*)
  have {*x*. *OAlist-tc-lookup xs x* ≠ *0*} ⊆ *fst* ‘ *set* (*list-of-oalist-tc xs*)
  proof (*rule*, *simp*)
    fix *x*
    assume *OAlist-tc-lookup xs x* ≠ *0*
    thus *x* ∈ *fst* ‘ *set* (*list-of-oalist-tc xs*)
      using *in-OAlist-tc-sorted-domain-iff-lookup set-OAlist-tc-sorted-domain* **by** *blast*
  qed
  also have *finite* ... **by** *simp*

finally (*finite-subset*) **show** *finite* {*x. OAlist-tc-lookup xs x* ≠ *0*} **.**
qed

lemma *keys-PP-oalist* [*code*]: *keys-pp* (*PP-oalist xs*) = *set* (*OAlist-tc-sorted-domain xs*)
by (*rule set-eqI*, *simp add*: *keys-pp-iff in-OAlist-tc-sorted-domain-iff-lookup*)

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 (*cases lex-comp′* (*PP-oalist xs*) (*PP-oalist ys*) = *Eq*)
case *True*
hence *PP-oalist xs* = *PP-oalist ys* **by** (*rule lex-comp′-EqD*)
hence *eq*: *OAlist-tc-lookup xs* = *OAlist-tc-lookup ys* **by** (*simp add*: *pp-eq-iff*)
have *OAlist-tc-lex-ord* (λ- *x y. Some* (*comparator-of x y*)) *xs ys* = *Some Eq*
by (*rule OAlist-tc-lex-ord-EqI*, *simp add*: *eq*)
thus *?thesis* **by** (*simp add*: *True*)
next
case *False*
then obtain *x* **where** *1*: *x* ∈ *keys-pp* (*rep-nat-pp* (*PP-oalist xs*)) ∪ *keys-pp* (*rep-nat-pp* (*PP-oalist ys*))
and *2*: *comparator-of* (*lookup-pp* (*rep-nat-pp* (*PP-oalist xs*)) *x*) (*lookup-pp* (*rep-nat-pp* (*PP-oalist ys*)) *x*) =
lex-comp′ (*PP-oalist xs*) (*PP-oalist ys*)
and *3*: ⋀*y. y* < *x* ⟹ *lookup-pp* (*rep-nat-pp* (*PP-oalist xs*)) *y* = *lookup-pp* (*rep-nat-pp* (*PP-oalist ys*)) *y*
by (*rule lex-comp′-valE*, *blast*)
have *OAlist-tc-lex-ord* (λ- *x y. Some* (*comparator-of x y*)) *xs ys* = *Some* (*lex-comp′* (*PP-oalist xs*) (*PP-oalist ys*))
proof (*rule OAlist-tc-lex-ord-valI*)
from *False* **show** *Some* (*lex-comp′* (*PP-oalist xs*) (*PP-oalist ys*)) ≠ *Some Eq* **by** *simp*
next
from *1* **have** *abs-nat x* ∈ *abs-nat* ‘ (*keys-pp* (*rep-nat-pp* (*PP-oalist xs*)) ∪ *keys-pp* (*rep-nat-pp* (*PP-oalist ys*)))
by (*rule imageI*)
also have ... = *fst* ‘ *set* (*list-of-oalist-tc xs*) ∪ *fst* ‘ *set* (*list-of-oalist-tc ys*)
by (*simp add*: *keys-rep-nat-pp-pp keys-PP-oalist OAlist-tc-sorted-domain-def image-Un image-image*)
finally show *abs-nat x* ∈ *fst* ‘ *set* (*list-of-oalist-tc xs*) ∪ *fst* ‘ *set* (*list-of-oalist-tc ys*) **.**
next
show *Some* (*lex-comp′* (*PP-oalist xs*) (*PP-oalist ys*)) =
Some (*comparator-of* (*OAlist-tc-lookup xs* (*abs-nat x*)) (*OAlist-tc-lookup ys* (*abs-nat x*)))
by (*simp add*: *2*[*symmetric*] *lookup-rep-nat-pp-pp*)
next
fix *y*::′*a*

assume *y* < *abs-nat x*
hence *rep-nat y* < *x* **by** (*metis abs-inverse ord-iff*(*2*))
hence *lookup-pp* (*rep-nat-pp* (*PP-oalist xs*)) (*rep-nat y*) = *lookup-pp* (*rep-nat-pp* (*PP-oalist ys*)) (*rep-nat y*)
by (*rule 3*)
hence *OAlist-tc-lookup xs y* = *OAlist-tc-lookup ys y* **by** (*auto simp*: *lookup-rep-nat-pp-pp elim*: *rep-inj*)
thus *Some* (*comparator-of* (*OAlist-tc-lookup xs y*) (*OAlist-tc-lookup ys y*)) = *Some Eq* **by** *simp*
qed
thus *?thesis* **by** *simp*
qed

lemma *zero-PP-oalist* [*code*]: (*0*::(*'a*::*linorder*, *'b*::*zero*) *pp*) = *PP-oalist OAlist-tc-empty*
by (*rule pp-eqI*, *simp add*: *lookup-OAlist-tc-empty*)

lemma *plus-PP-oalist* [*code*]:
PP-oalist xs + *PP-oalist ys* = *PP-oalist* (*OAlist-tc-map2-val-neutr* (λ-. (+)) *xs ys*)
by (*rule pp-eqI*, *simp add*: *lookup-plus-pp*, *rule lookup-OAlist-tc-map2-val-neutr*[*symmetric*], *simp-all*)

lemma *minus-PP-oalist* [*code*]:
PP-oalist xs − *PP-oalist ys* = *PP-oalist* (*OAlist-tc-map2-val-rneutr* (λ-. (−)) *xs ys*)
by (*rule pp-eqI*, *simp add*: *lookup-minus-pp*, *rule lookup-OAlist-tc-map2-val-rneutr*[*symmetric*], *simp*)

lemma *equal-PP-oalist* [*code*]: *equal-class.equal* (*PP-oalist xs*) (*PP-oalist ys*) = (*xs* = *ys*)
by (*simp add*: *equal-eq pp-eq-iff*, *auto elim*: *OAlist-tc-lookup-inj*)

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*
by (*rule pp-eqI*, *simp add*: *lookup-lcs-pp*, *rule lookup-OAlist-tc-map2-val-neutr*[*symmetric*], *simp-all add*: *max-def*)

lemma *deg-pp-PP-oalist* [*code*]: *deg-pp* (*PP-oalist xs*) = *sum-list* (*map snd* (*list-of-oalist-tc xs*))
proof −
have *irreflp* ((<)::-::*linorder* ⇒ -) **by** (*rule irreflpI*, *simp*)
have *deg-pp* (*PP-oalist xs*) = *sum* (*OAlist-tc-lookup xs*) (*set* (*OAlist-tc-sorted-domain xs*))
by (*simp add*: *deg-pp-alt keys-PP-oalist*)
also have ... = *sum-list* (*map* (*OAlist-tc-lookup xs*) (*OAlist-tc-sorted-domain xs*))
by (*rule sum.distinct-set-conv-list*, *rule distinct-sorted-wrt-irrefl*,
fact, *fact transp-on-less*, *fact sorted-OAlist-tc-sorted-domain*)

also have ... = *sum-list* (*map snd* (*list-of-oalist-tc xs*))
by (*rule arg-cong*[**where** *f*=*sum-list*], *simp add*: *OAlist-tc-sorted-domain-def OAlist-tc-lookup-eq-valueI*)
finally show *?thesis* **.**
qed

lemma *single-PP-oalist* [*code*]: *single-pp x e* = *PP-oalist* (*oalist-tc-of-list* [(*x*, *e*)])
by (*rule pp-eqI*, *simp add*: *lookup-single-pp OAlist-tc-lookup-single*)

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 (*simp add*: *adds-pp-add-linorder-def adds-pp-iff adds-poly-mapping lookup-pp.rep-eq*[*symmetric*] *OAlist-tc-prod-ord-alt le-fun-def*,
intro iffI allI ballI)
fix *k*
assume ∀ *x*. *OAlist-tc-lookup xs x* ≤ *OAlist-tc-lookup ys x*
thus *OAlist-tc-lookup xs k* ≤ *OAlist-tc-lookup ys k* **by** *blast*
next
fix *x*
assume ∗: ∀ *k*∈*fst* ‘ *set* (*list-of-oalist-tc xs*) ∪ *fst* ‘ *set* (*list-of-oalist-tc ys*).
OAlist-tc-lookup xs k ≤ *OAlist-tc-lookup ys k*
show *OAlist-tc-lookup xs x* ≤ *OAlist-tc-lookup ys x*
proof (*cases x* ∈ *fst* ‘ *set* (*list-of-oalist-tc xs*) ∪ *fst* ‘ *set* (*list-of-oalist-tc ys*))
case *True*
with ∗ **show** *?thesis* **..**
next
case *False*
hence *x* ∉ *set* (*OAlist-tc-sorted-domain xs*) **and** *x* ∉ *set* (*OAlist-tc-sorted-domain ys*)
by (*simp-all add*: *set-OAlist-tc-sorted-domain*)
thus *?thesis* **by** (*simp add*: *in-OAlist-tc-sorted-domain-iff-lookup*)
qed
qed

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*)]
by *eval*

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*)]
by *eval*

lemma
($sparse_0$ [(*X*, *2*::*nat*), (*Z*, *1*)]) *adds* ($sparse_0$ [(*X*, *3*), (*Y*, *2*), (*Z*, *1*)])
by *eval*

lemma
lookup-pp ($sparse_0$ [(*X*, *2*::*nat*), (*Z*, *3*)]) *X* = *2*
by *eval*

lemma
deg-pp ($sparse_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *3*), (*X*, *1*)]) = *6*
by *eval*

lemma
lex-comp ($sparse_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *3*)]) ($sparse_0$ [(*X*, *4*)]) = *Lt*
by *eval*

lemma
lex-comp ($sparse_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *3*)], *3*::*nat*) ($sparse_0$ [(*X*, *4*)], *2*) = *Lt*
by *eval*

lemma
lex-pp ($sparse_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *3*)]) ($sparse_0$ [(*X*, *4*)])
by *eval*

lemma
lex-pp ($sparse_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *3*)]) ($sparse_0$ [(*X*, *4*)])
by *eval*

lemma
¬ *dlex-pp* ($sparse_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *3*)]) ($sparse_0$ [(*X*, *4*)])
by *eval*

lemma
dlex-pp ($sparse_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *2*)]) ($sparse_0$ [(*X*, *5*)])
by *eval*

lemma
  ¬ *drlex-pp* (*sparse*$_0$ [(*X*, *2*::*nat*), (*Y*, *1*), (*Z*, *2*)]) (*sparse*$_0$ [(*X*, *5*)])
  by *eval*

end

15.2 *MP-oalist*

lift-definition *MP-oalist* :: ('*a*::*nat-term*, '*b*::*zero*) *oalist-ntm* ⇒ '*a* ⇒$_0$ '*b*
  is *OAlist-lookup-ntm*
proof −
  fix *xs* :: ('*a*, '*b*) *oalist-ntm*
  have {*x*. *OAlist-lookup-ntm xs x* ≠ *0*} ⊆ *fst* ' *set* (*fst* (*list-of-oalist-ntm xs*))
  proof (*rule*, *simp*)
    fix *x*
    assume *OAlist-lookup-ntm xs x* ≠ *0*
    thus *x* ∈ *fst* ' *set* (*fst* (*list-of-oalist-ntm xs*))
      using *oa-ntm.in-sorted-domain-iff-lookup oa-ntm.set-sorted-domain* **by** *blast*
  qed
  also have *finite ...* **by** *simp*
  finally (*finite-subset*) **show** *finite* {*x*. *OAlist-lookup-ntm xs x* ≠ *0*} **.**
qed

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*)))
  by (*rule set-eqI*, *simp add*: *in-keys-iff oa-ntm.in-sorted-domain-iff-lookup*[*simplified oa-ntm.set-sorted-domain*])

lemma *MP-oalist-empty* [*simp*]: *MP-oalist* (*OAlist-empty-ntm ko*) = *0*
  by (*rule poly-mapping-eqI*, *simp add*: *oa-ntm.lookup-empty*)

lemma *zero-MP-oalist* [*code*]: (*0*::('*a*::{*linorder*,*nat-term*} ⇒$_0$ '*b*::*zero*)) = *MP-oalist* (*OAlist-empty-ntm nat-term-order-of-le*)
  by *simp*

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*))
  unfolding *is-zero-def List.null-def*
proof
  assume *MP-oalist xs* = *0*
  hence *OAlist-lookup-ntm xs k* = *0* **for** *k* **by** (*simp add*: *poly-mapping-eq-iff*)
  thus *fst* (*list-of-oalist-ntm xs*) = []

by (*metis image-eqI ko-ntm.min-key-val-raw-in oa-ntm.in-sorted-domain-iff-lookup oa-ntm.set-sorted-domain*)
next
assume *fst* (*list-of-oalist-ntm xs*) = []
hence *OAlist-lookup-ntm xs k* = *0* **for** *k*
by (*metis oa-ntm.list-of-oalist-empty oa-ntm.lookup-empty oalist-ntm-eqI surjective-pairing*)
thus *MP-oalist xs* = *0* **by** (*simp add*: *poly-mapping-eq-iff ext*)
qed

lemma *plus-MP-oalist* [*code*]: *MP-oalist xs* + *MP-oalist ys* = *MP-oalist* (*OAlist-map2-val-neutr-ntm* (λ-. (+)) *xs ys*)
by (*rule poly-mapping-eqI*, *simp add*: *lookup-plus-fun*, *rule oa-ntm.lookup-map2-val-neutr*[*symmetric*], *simp-all*)

lemma *minus-MP-oalist* [*code*]: *MP-oalist xs* − *MP-oalist ys* = *MP-oalist* (*OAlist-map2-val-rneutr-ntm* (λ-. (−)) *xs ys*)
by (*rule poly-mapping-eqI*, *simp add*: *lookup-minus-fun*, *rule oa-ntm.lookup-map2-val-rneutr*[*symmetric*], *simp*)

lemma *uminus-MP-oalist* [*code*]: − *MP-oalist xs* = *MP-oalist* (*OAlist-map-val-ntm* (λ-. *uminus*) *xs*)
by (*rule poly-mapping-eqI*, *simp*, *rule oa-ntm.lookup-map-val*[*symmetric*], *simp*)

lemma *equal-MP-oalist* [*code*]: *equal-class.equal* (*MP-oalist xs*) (*MP-oalist ys*) = (*OAlist-eq-ntm xs ys*)
by (*simp add*: *oa-ntm.oalist-eq-alt equal-eq poly-mapping-eq-iff*)

lemma *map-MP-oalist* [*code*]: *Poly-Mapping.map f* (*MP-oalist xs*) = *MP-oalist* (*OAlist-map-val-ntm* (λ-. *f*) *xs*)
proof −
have *eq*: *OAlist-map-val-ntm* (λ-. *f*) *xs* = *OAlist-map-val-ntm* (λ- *c*. *f c when c* ≠ *0*) *xs*
proof (*rule oa-ntm.map-val-cong*)
fix *t c*
assume ∗: (*t*, *c*) ∈ *set* (*fst* (*list-of-oalist-ntm xs*))
hence *fst* (*t*, *c*) ∈ *fst* ‘ *set* (*fst* (*list-of-oalist-ntm xs*)) **by** (*rule imageI*)
hence *OAlist-lookup-ntm xs t* ≠ *0*
by (*simp add*: *oa-ntm.in-sorted-domain-iff-lookup*[*simplified oa-ntm.set-sorted-domain*])
moreover from ∗ **have** *OAlist-lookup-ntm xs t* = *c* **by** (*rule oa-ntm.lookup-eq-valueI*)
ultimately have *c* ≠ *0* **by** *simp*
thus *f c* = (*f c when c* ≠ *0*) **by** *simp*
qed
show *?thesis*
by (*rule poly-mapping-eqI*, *simp add*: *Poly-Mapping.map.rep-eq eq*, *rule oa-ntm.lookup-map-val*[*symmetric*], *simp*)
qed

lemma *range-MP-oalist* [*code*]: *Poly-Mapping.range* (*MP-oalist xs*) = *set* (*map snd*

(*fst* (*list-of-oalist-ntm xs*)))
proof (*simp add*: *Poly-Mapping.range.rep-eq*, *intro set-eqI iffI*)
  fix *c*
  assume *c* ∈ *range* (*OAlist-lookup-ntm xs*) − {*0*}
  hence *c* ∈ *range* (*OAlist-lookup-ntm xs*) **and** *c* ≠ *0* **by** *simp-all*
  from *this*(*1*) **obtain** *t* **where** *OAlist-lookup-ntm xs t* = *c* **by** *fastforce*
   with ‹*c* ≠ *0*› **have** (*t*, *c*) ∈ *set* (*fst* (*list-of-oalist-ntm xs*)) **by** (*simp add*: *oa-ntm.lookup-eq-value*)
  hence *snd* (*t*, *c*) ∈ *snd* ‘ *set* (*fst* (*list-of-oalist-ntm xs*)) **by** (*rule imageI*)
  thus *c* ∈ *snd* ‘ *set* (*fst* (*list-of-oalist-ntm xs*)) **by** *simp*
next
  fix *c*
  assume *c* ∈ *snd* ‘ *set* (*fst* (*list-of-oalist-ntm xs*))
  then obtain *t* **where** ∗: (*t*, *c*) ∈ *set* (*fst* (*list-of-oalist-ntm xs*)) **by** *fastforce*
  hence *fst* (*t*, *c*) ∈ *fst* ‘ *set* (*fst* (*list-of-oalist-ntm xs*)) **by** (*rule imageI*)
  hence *OAlist-lookup-ntm xs t* ≠ *0*
   by (*simp add*: *oa-ntm.in-sorted-domain-iff-lookup*[*simplified oa-ntm.set-sorted-domain*])
  moreover from ∗ **have** *OAlist-lookup-ntm xs t* = *c* **by** (*rule oa-ntm.lookup-eq-valueI*)
  ultimately show *c* ∈ *range* (*OAlist-lookup-ntm xs*) − {*0*} **by** *fastforce*
qed

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*)
  by *simp* (*metis UnI1 UnI2 in-keys-iff poly-mapping-eqI*)

lemma *keys-add-eq*: *keys* (*a* + *b*) = *keys a* ∪ *keys b* − {*x* ∈ *keys a* ∩ *keys b*. *lookup a x* + *lookup b x* = *0*}
  by (*auto simp*: *in-keys-iff lookup-add add-eq-0-iff*
      simp del: *lookup-not-eq-zero-eq-in-keys*)

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*))
unfolding *shift-map-keys-def*
by (*rule oa-ntm.list-of-oalist-of-list-id*, *rule ko-ntm.oalist-inv-map-raw*, *fact oalist-inv-list-of-oalist-ntm*,
simp add: *nat-term-compare-inv-conv*[*symmetric*] *nat-term-compare-inv-def splus-eq-splus nat-term-compare-splus*)

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 −
let *?f* = λ*kv*. (*t* ⊕ *fst kv*, *c* ∗ *snd kv*)
have *?l* = *lookup-ko-ntm* (*map-raw ?f* (*list-of-oalist-ntm xs*)) (*fst* (*?f* (*u*, *c*)))
by (*simp add*: *oa-ntm.lookup-def list-of-oalist-shift-keys*)
also have ... = *snd* (*?f* (*u*, *lookup-ko-ntm* (*list-of-oalist-ntm xs*) *u*))
by (*rule ko-ntm.lookup-raw-map-raw*, *fact oalist-inv-list-of-oalist-ntm*, *simp*,
simp add: *nat-term-compare-inv-conv*[*symmetric*] *nat-term-compare-inv-def splus-eq-splus nat-term-compare-splus*)
also have ... = *?r* **by** (*simp add*: *oa-ntm.lookup-def*)
finally show *?thesis* **.**
qed

lemma *keys-shift-map-keys-subset*:
keys (*MP-oalist* (*shift-map-keys t* ((∗) *c*) *xs*)) ⊆ ((⊕) *t*) ' *keys* (*MP-oalist xs*) (**is** *?l* ⊆ *?r*)
proof −
let *?f* = λ*kv*. (*t* ⊕ *fst kv*, *c* ∗ *snd kv*)
have *?l* = *fst* ' *set* (*fst* (*map-raw ?f* (*list-of-oalist-ntm xs*)))
by (*simp add*: *keys-MP-oalist list-of-oalist-shift-keys*)
also from *ko-ntm.map-raw-subset* **have** ... ⊆ *fst* ' *?f* ' *set* (*fst* (*list-of-oalist-ntm xs*))
by (*rule image-mono*)
also have ... ⊆ *?r* **by** (*simp add*: *keys-MP-oalist image-image*)
finally show *?thesis* **.**
qed

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 (*cases c* = *0*)
case *True*
hence *monom-mult c t* (*MP-oalist xs*) = *0* **using** *monom-mult-zero-left* **by** *simp*
thus *?thesis* **using** *True* **by** *simp*

next
  case *False*
  have *monom-mult c t* (*MP-oalist xs*) = *MP-oalist* (*shift-map-keys t* ((∗) *c*) *xs*)
  proof (*rule poly-mapping-eqI*, *simp add*: *lookup-monom-mult del*: *MP-oalist.rep-eq*, *intro conjI impI*)
    fix *u*
    assume *t* $adds_p$ *u*
    then obtain *v* **where** *u* = *t* ⊕ *v* **by** (*rule adds-ppE*)
    thus *c* ∗ *lookup* (*MP-oalist xs*) (*u* ⊖ *t*) = *lookup* (*MP-oalist* (*shift-map-keys t* ((∗) *c*) *xs*)) *u*
      by (*simp add*: *splus-sminus lookup-shift-map-keys-plus del*: *MP-oalist.rep-eq*)
  next
    fix *u*
    assume ¬ *t* $adds_p$ *u*
    have *u* ∉ *keys* (*MP-oalist* (*shift-map-keys t* ((∗) *c*) *xs*))
    proof
      assume *u* ∈ *keys* (*MP-oalist* (*shift-map-keys t* ((∗) *c*) *xs*))
    also have ... ⊆ ((⊕) *t*) ' *keys* (*MP-oalist xs*) **by** (*fact keys-shift-map-keys-subset*)
      finally obtain *v* **where** *u* = *t* ⊕ *v* **..**
      hence *t* $adds_p$ *u* **by** (*rule adds-ppI*)
      with ‹¬ *t* $adds_p$ *u*› **show** *False* **..**
    qed
    thus *lookup* (*MP-oalist* (*shift-map-keys t* ((∗) *c*) *xs*)) *u* = *0* **by** (*simp add*: *in-keys-iff*)
  qed
  thus *?thesis* **by** (*simp add*: *False*)
qed

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 (*split if-split*, *intro conjI impI*)
  assume *is-zero* (*MP-oalist xs*)
  thus *MP-oalist xs* ⊙ *MP-oalist ys* = *MP-oalist* (*OAlist-empty-ntm* (*snd* (*list-of-oalist-ntm ys*)))
    by (*simp add*: *is-zero-def*)
next
  assume ¬ *is-zero* (*MP-oalist xs*)
  hence ∗: *fst* (*list-of-oalist-ntm xs*) ≠ [] **by** (*simp add*: *is-zero-MP-oalist List.null-def*)
  define *ct* **where** *ct* = *OAlist-hd-ntm xs*
  have *eq*: *except* (*MP-oalist xs*) {*fst ct*} = *MP-oalist* (*OAlist-tl-ntm xs*)
    by (*rule poly-mapping-eqI*, *simp add*: *lookup-except ct-def oa-ntm.lookup-tl'*)
  have *MP-oalist xs* ⊙ *MP-oalist ys* =
        monom-mult (*lookup* (*MP-oalist xs*) (*fst ct*)) (*fst ct*) (*MP-oalist ys*) +

except (*MP-oalist xs*) {*fst ct*} ⊙ *MP-oalist ys* **by** (*fact mult-scalar-rec-left*)
also have ... = *monom-mult* (*snd ct*) (*fst ct*) (*MP-oalist ys*) + *except* (*MP-oalist xs*) {*fst ct*} ⊙ *MP-oalist ys*
using ∗ **by** (*simp add*: *ct-def oa-ntm.snd-hd*)
also have ... = *monom-mult* (*snd ct*) (*fst ct*) (*MP-oalist ys*) + *MP-oalist* (*OAlist-tl-ntm xs*) ⊙ *MP-oalist ys*
by (*simp only*: *eq*)
finally show *MP-oalist xs* ⊙ *MP-oalist ys* =
(*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*)
by (*simp add*: *ct-def Let-def*)
qed

end

15.2.1 Special case of addition: adding monomials

definition *plus-monomial-less* :: ($'a \Rightarrow_0 'b$) ⇒ $'b$ ⇒ $'a$ ⇒ ($'a \Rightarrow_0 'b$::*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*)
unfolding *plus-monomial-less-def oa-ntm.update-by-fun-gr-eq-update-by-fun*
by (*rule poly-mapping-eqI*, *simp add*: *lookup-plus-fun oa-ntm.lookup-update-by-fun lookup-single*)

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* $\Rightarrow_0$ $'b$::{*one*,*zero*} **where**
V_0 *n* ≡ *monomial 1* (*single-pp n 1*)

definition C_0 :: $'b$ ⇒ ($'a$, *nat*) *pp* $\Rightarrow_0$ $'b$::*zero* **where** C_0 *c* ≡ *monomial c 0*

lemma C_0-*one*: C_0 *1* = *1*
by (*simp add*: C_0-*def*)

lemma C_0-*numeral*: C_0 (*numeral x*) = *numeral x*

by (*auto intro*!: *poly-mapping-eqI simp*: C_0*-def lookup-numeral*)

lemma C_0*-minus*: C_0 (− *x*) = − C_0 *x*
by (*simp add*: C_0*-def single-uminus*)

lemma C_0*-zero*: C_0 *0* = *0*
by (*auto intro*!: *poly-mapping-eqI simp*: C_0*-def*)

lemma V_0*-power*: V_0 *v* ^ *n* = *monomial 1* (*single-pp v n*)
by (*induction n*) (*auto simp*: V_0*-def mult-single single-pp-plus*)

lemma *single-MP-oalist* [*code*]: *Poly-Mapping.single k v* = $distr_0$ *nat-term-order-of-le* [(*k*, *v*)]
unfolding $distr_0$*-def* **by** (*rule poly-mapping-eqI*, *simp add*: *lookup-single OAlist-lookup-ntm-single*)

lemma *one-MP-oalist* [*code*]: *1* = $distr_0$ *nat-term-order-of-le* [(*0*, *1*)]
by (*metis single-MP-oalist single-one*)

lemma *except-MP-oalist* [*code*]: *except* (*MP-oalist xs*) *S* = *MP-oalist* (*OAlist-filter-ntm* (λ*kv*. *fst kv* ∉ *S*) *xs*)
by (*rule poly-mapping-eqI*, *simp add*: *lookup-except oa-ntm.lookup-filter*)

15.2.3 Changing the Internal Order

definition *change-ord* :: *'a*::*nat-term-compare nat-term-order* ⇒ (*'a* $\Rightarrow_0$ *'b*) ⇒ (*'a* $\Rightarrow_0$ *'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*)
by (*rule poly-mapping-eqI*, *simp add*: *change-ord-def oa-ntm.lookup-reorder*)

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 (*induct xs arbitrary*: *a b*)
fix *a b*
show *foldl f* (*f a b*) [] = *f a* (*foldl f b* []) **by** *simp*
next
fix *a b x xs*
assume ⋀*a b*. *foldl f* (*f a b*) *xs* = *f a* (*foldl f b xs*)
from *assms*[*of a b x*] *this*[*of a f b x*]
show *foldl f* (*f a b*) (*x* # *xs*) = *f a* (*foldl f b* (*x* # *xs*)) **unfolding** *foldl-Cons* **by** *simp*
qed

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 (*split if-split*, *intro conjI impI*)
assume *is-zero* (*MP-oalist xs*)
thus *lt* (*MP-oalist xs*) = *min-term* **by** (*simp add*: *is-zero-def*)
next
assume ¬ *is-zero* (*MP-oalist xs*)
hence *fst* (*list-of-oalist-ntm xs*) ≠ [] **by** (*simp add*: *is-zero-MP-oalist List.null-def*)
show *lt* (*MP-oalist xs*) = *fst* (*OAlist-min-key-val-ntm cmp-term xs*)
proof (*rule lt-eqI-keys*)
show *fst* (*OAlist-min-key-val-ntm cmp-term xs*) ∈ *keys* (*MP-oalist xs*)
by (*simp add*: *keys-MP-oalist*, *rule imageI*, *rule oa-ntm.min-key-val-in*, *fact*)
next
fix *u*
assume *u* ∈ *keys* (*MP-oalist xs*)
also have ... = *fst* ' *set* (*fst* (*list-of-oalist-ntm xs*)) **by** (*simp add*: *keys-MP-oalist*)
finally obtain *z* **where** *z* ∈ *set* (*fst* (*list-of-oalist-ntm xs*)) **and** *u* = *fst z* **..**
from *this*(*1*) **have** *ko.le* (*key-order-of-nat-term-order-inv cmp-term*) (*fst* (*OAlist-min-key-val-ntm cmp-term xs*)) *u*
unfolding ‹*u* = *fst z*› **by** (*rule oa-ntm.min-key-val-minimal*)
thus *le-of-nat-term-order cmp-term u* (*fst* (*OAlist-min-key-val-ntm cmp-term xs*))
by (*simp add*: *le-of-nat-term-order-alt*)
qed
qed

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 (*split if-split*, *intro conjI impI*)
assume *is-zero* (*MP-oalist xs*)
thus *lc* (*MP-oalist xs*) = *0* **by** (*simp add*: *is-zero-def*)
next
assume ¬ *is-zero* (*MP-oalist xs*)
moreover from *this* **have** *fst* (*list-of-oalist-ntm xs*) ≠ [] **by** (*simp add*: *is-zero-MP-oalist List.null-def*)
ultimately show *lc* (*MP-oalist xs*) = *snd* (*OAlist-min-key-val-ntm cmp-term xs*)
by (*simp add*: *lc-def lt-MP-oalist oa-ntm.snd-min-key-val*)

qed

lemma *tail-MP-oalist* [*code*]: *tail* (*MP-oalist xs*) = *MP-oalist* (*OAlist-except-min-ntm cmp-term xs*)
proof (*cases is-zero* (*MP-oalist xs*))
  case *True*
  hence *fst* (*list-of-oalist-ntm xs*) = [] **by** (*simp add*: *is-zero-MP-oalist List.null-def*)
  hence *fst* (*list-of-oalist-ntm* (*OAlist-except-min-ntm cmp-term xs*)) = []
    by (*rule oa-ntm.except-min-Nil*)
  hence *is-zero* (*MP-oalist* (*OAlist-except-min-ntm cmp-term xs*))
    by (*simp add*: *is-zero-MP-oalist List.null-def*)
  with *True* **show** *?thesis* **by** (*simp add*: *is-zero-def*)
next
  case *False*
  show *?thesis* **by** (*rule poly-mapping-eqI*, *simp add*: *lookup-tail-2 oa-ntm.lookup-except-min′ lt-MP-oalist False*)
qed

definition *comp-opt-p* :: ($'t \Rightarrow_0 'c$::*zero*, $'t \Rightarrow_0 'c$) *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 −
  let *?f* = 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*
  show *?thesis*
  proof (*cases comp-opt-p* (*MP-oalist xs*) (*MP-oalist ys*) = *Some Eq*)
    case *True*
    hence *MP-oalist xs* = *MP-oalist ys* **by** (*simp add*: *comp-opt-p-def split*: *if-splits*)
    hence *lookup* (*MP-oalist xs*) = *lookup* (*MP-oalist ys*) **by** (*rule arg-cong*)
    hence *eq*: *OAlist-lookup-ntm xs* = *OAlist-lookup-ntm ys* **by** *simp*
    have *OAlist-lex-ord-ntm cmp-term ?f xs ys* = *Some Eq*
      by (*rule oa-ntm.lex-ord-EqI*, *simp add*: *eq*)
    with *True* **show** *?thesis* **by** *simp*
  next
    case *False*
    hence *neq*: *MP-oalist xs* ≠ *MP-oalist ys* **by** (*simp add*: *comp-opt-p-def split*: *if-splits*)
    then obtain *v* **where** *1*: *v* ∈ *keys* (*MP-oalist xs*) ∪ *keys* (*MP-oalist ys*)
      and *2*: *lookup* (*MP-oalist xs*) *v* ≠ *lookup* (*MP-oalist ys*) *v*
      and *3*: ⋀*u. lt-of-nat-term-order cmp-term v u* ⟹ *lookup* (*MP-oalist xs*) *u* = *lookup* (*MP-oalist ys*) *u*
      by (*rule poly-mapping-neqE*, *blast*)
    show *?thesis*

proof (*rule HOL.sym*, *rule oa-ntm.lex-ord-valI*)
from *1* **show** *v* ∈ *fst* ‘ *set* (*fst* (*list-of-oalist-ntm xs*)) ∪ *fst* ‘ *set* (*fst* (*list-of-oalist-ntm ys*))
by (*simp add*: *keys-MP-oalist*)
next
from *2* **have** *4*: *OAlist-lookup-ntm xs v* ≠ *OAlist-lookup-ntm ys v* **by** *simp*
show *comp-opt-p* (*MP-oalist xs*) (*MP-oalist ys*) =
(*if OAlist-lookup-ntm xs v* = *OAlist-lookup-ntm ys v then Some Eq*
else if OAlist-lookup-ntm xs v = *0 then Some Lt*
else if OAlist-lookup-ntm ys v = *0 then Some Gt else None*)
proof (*simp add*: *4*, *intro conjI impI*)
assume *OAlist-lookup-ntm ys v* = *0* **and** *OAlist-lookup-ntm xs v* = *0*
with *4* **show** *comp-opt-p* (*MP-oalist xs*) (*MP-oalist ys*) = *Some Lt* **by** *simp*
next
assume *OAlist-lookup-ntm xs v* ≠ *0* **and** *OAlist-lookup-ntm ys v* = *0*
hence *lookup* (*MP-oalist ys*) *v* = *0* **and** *lookup* (*MP-oalist xs*) *v* ≠ *0* **by** *simp-all*
hence *ord-strict-p* (*MP-oalist ys*) (*MP-oalist xs*) **using** *3*[*symmetric*]
by (*rule ord-strict-pI*)
with *neq* **show** *comp-opt-p* (*MP-oalist xs*) (*MP-oalist ys*) = *Some Gt* **by** (*auto simp*: *comp-opt-p-def*)
next
assume *OAlist-lookup-ntm ys v* ≠ *0* **and** *OAlist-lookup-ntm xs v* = *0*
hence *lookup* (*MP-oalist xs*) *v* = *0* **and** *lookup* (*MP-oalist ys*) *v* ≠ *0* **by** *simp-all*
hence *ord-strict-p* (*MP-oalist xs*) (*MP-oalist ys*) **using** *3* **by** (*rule ord-strict-pI*)
with *neq* **show** *comp-opt-p* (*MP-oalist xs*) (*MP-oalist ys*) = *Some Lt* **by** (*auto simp*: *comp-opt-p-def*)
next
assume *OAlist-lookup-ntm xs v* ≠ *0*
hence *lookup* (*MP-oalist xs*) *v* ≠ *0* **by** *simp*
with *2* **have** *a*: ¬ *ord-strict-p* (*MP-oalist xs*) (*MP-oalist ys*) **using** *3* **by** (*rule not-ord-strict-pI*)
assume *OAlist-lookup-ntm ys v* ≠ *0*
hence *lookup* (*MP-oalist ys*) *v* ≠ *0* **by** *simp*
with *2*[*symmetric*] **have** ¬ *ord-strict-p* (*MP-oalist ys*) (*MP-oalist xs*)
using *3*[*symmetric*] **by** (*rule not-ord-strict-pI*)
with *neq a* **show** *comp-opt-p* (*MP-oalist xs*) (*MP-oalist ys*) = *None* **by** (*auto simp*: *comp-opt-p-def*)
qed
next
fix *u*
assume *ko.lt* (*key-order-of-nat-term-order-inv cmp-term*) *u v*
hence *lt-of-nat-term-order cmp-term v u* **by** (*simp only*: *lt-of-nat-term-order-alt*)
hence *lookup* (*MP-oalist xs*) *u* = *lookup* (*MP-oalist ys*) *u* **by** (*rule 3*)
thus (*if OAlist-lookup-ntm xs u* = *OAlist-lookup-ntm ys u then Some Eq*
else if OAlist-lookup-ntm xs u = *0 then Some Lt*
else if OAlist-lookup-ntm ys u = *0 then Some Gt else None*) = *Some Eq* **by** *simp*

qed *fact*
qed
qed

lemma *compute-ord-p* [*code*]: *ord-p p q* = (*let aux* = *comp-opt-p p q in aux* = *Some Lt* ∨ *aux* = *Some Eq*)
by (*auto simp*: *ord-p-def comp-opt-p-def*)

lemma *compute-ord-p-strict* [*code*]: *ord-strict-p p q* = (*comp-opt-p p q* = *Some Lt*)
by (*auto simp*: *comp-opt-p-def*)

lemma *keys-to-list-MP-oalist* [*code*]: *keys-to-list* (*MP-oalist xs*) = *OAlist-sorted-domain-ntm cmp-term xs*
proof −
have *eq*: *ko.lt* (*key-order-of-nat-term-order-inv cmp-term*) = *ord-term-strict-conv*
by (*intro ext*, *simp add*: *lt-of-nat-term-order-alt*)
have *1*: *irreflp ord-term-strict-conv* **by** (*rule irreflpI*, *simp*)
have *2*: *transp ord-term-strict-conv* **by** (*rule transpI*, *simp*)
have *antisymp ord-term-strict-conv* **by** (*rule antisympI*, *simp*)
moreover have *3*: *sorted-wrt ord-term-strict-conv* (*keys-to-list* (*MP-oalist xs*))
unfolding *keys-to-list-def* **by** (*fact pps-to-list-sorted-wrt*)
moreover note -
moreover have *4*: *sorted-wrt ord-term-strict-conv* (*OAlist-sorted-domain-ntm cmp-term xs*)
unfolding *eq*[*symmetric*] **by** (*fact oa-ntm.sorted-sorted-domain*)
ultimately show *?thesis*
proof (*rule sorted-wrt-distinct-set-unique*)
from *1 2 3* **show** *distinct* (*keys-to-list* (*MP-oalist xs*)) **by** (*rule distinct-sorted-wrt-irrefl*)
next
from *1 2 4* **show** *distinct* (*OAlist-sorted-domain-ntm cmp-term xs*) **by** (*rule distinct-sorted-wrt-irrefl*)
next
show *set* (*keys-to-list* (*MP-oalist xs*)) = *set* (*OAlist-sorted-domain-ntm cmp-term xs*)
by (*simp add*: *set-keys-to-list keys-MP-oalist oa-ntm.set-sorted-domain*)
qed
qed

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 $\bigwedge v$. *fst* (*rep-nat-term v*) = *rep-nat-pp* (*fst* (*pair-of-term v*))
and $\bigwedge t$. *snd* (*rep-nat-term* (*term-of-pair* (*t*, *the-min*))) = *0*
and $\bigwedge v\ w$. *snd* (*pair-of-term v*) ≤ *snd* (*pair-of-term w*) ⟹ *snd* (*rep-nat-term v*) ≤ *snd* (*rep-nat-term w*)
and $\bigwedge s\ t\ k$. *term-of-pair* (*s* + *t*, *k*) = *splus* (*term-of-pair* (*s*, *k*)) (*term-of-pair* (*t*, *k*))
and $\bigwedge t\ v$. *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 −
from *assms*(*1*) **interpret** *tp*: *term-powerprod pair-of-term term-of-pair* **.**
let *?f* = λ*x*. *term-of-pair* (*x*, *the-min*)
show *?thesis*
proof (*intro gd-term.intro ordered-term.intro*)
from *assms*(*1*) **show** *term-powerprod pair-of-term term-of-pair* **.**
next
show *ordered-powerprod* (λ*s t*. *le-of-nat-term-order cmp-term* (*?f s*) (*?f t*))
(λ*s t*. *lt-of-nat-term-order cmp-term* (*?f s*) (*?f t*))
proof (*intro ordered-powerprod.intro ordered-powerprod-axioms.intro*)
show *class.linorder* (λ*s t*. *le-of-nat-term-order cmp-term* (*?f s*) (*?f t*))
(λ*s t*. *lt-of-nat-term-order cmp-term* (*?f s*) (*?f t*))
proof (*unfold-locales*, *simp-all add*: *lt-of-nat-term-order-alt le-of-nat-term-order-alt ko.linear ko.less-le-not-le*)
fix *x y*
assume *ko.le* (*key-order-of-nat-term-order-inv cmp-term*) (*term-of-pair* (*x*, *the-min*)) (*term-of-pair* (*y*, *the-min*))
and *ko.le* (*key-order-of-nat-term-order-inv cmp-term*) (*term-of-pair* (*y*, *the-min*)) (*term-of-pair* (*x*, *the-min*))
hence *term-of-pair* (*x*, *the-min*) = *term-of-pair* (*y*, *the-min*)
by (*rule ko.antisym*)
hence (*x*, *the-min*) = (*y*, *the-min*::*'k*) **by** (*rule tp.term-of-pair-injective*)
thus *x* = *y* **by** *simp*
qed
next
fix *t*
show *le-of-nat-term-order cmp-term* (*?f 0*) (*?f t*)
unfolding *le-of-nat-term-order*
by (*rule nat-term-compD1'*, *fact comparator-nat-term-compare*, *fact nat-term-comp-nat-term-compare*, *simp add*: *assms*(*3*), *simp add*: *assms*(*2*) *zero-pp tp.pair-term*)
next
fix *s t u*
assume *le-of-nat-term-order cmp-term* (*?f s*) (*?f t*)
hence *le-of-nat-term-order cmp-term* (*?f* (*u* + *s*)) (*?f* (*u* + *t*))

by (*simp add*: *le-of-nat-term-order assms*(*5*) *nat-term-compare-splus*)
thus *le-of-nat-term-order cmp-term* (*?f* (*s* + *u*)) (*?f* (*t* + *u*)) **by** (*simp only*: *ac-simps*)
qed
next
show *class.linorder* (*le-of-nat-term-order cmp-term*) (*lt-of-nat-term-order cmp-term*)
by (*fact linorder-le-of-nat-term-order*)
next
show *ordered-term-axioms pair-of-term term-of-pair* (*λs t*. *le-of-nat-term-order cmp-term* (*?f s*) (*?f t*))
(*le-of-nat-term-order cmp-term*)
proof
fix *v w t*
assume *le-of-nat-term-order cmp-term v w*
thus *le-of-nat-term-order cmp-term* (*t* ⊕ *v*) (*t* ⊕ *w*)
by (*simp add*: *le-of-nat-term-order assms*(*6*) *nat-term-compare-splus*)
next
fix *v w*
assume *le-of-nat-term-order cmp-term* (*?f* (*tp.pp-of-term v*)) (*?f* (*tp.pp-of-term w*))
hence *3*: *nat-term-compare cmp-term* (*?f* (*tp.pp-of-term v*)) (*?f* (*tp.pp-of-term w*)) ≠ *Gt*
by (*simp add*: *le-of-nat-term-order*)
assume *tp.component-of-term v* ≤ *tp.component-of-term w*
hence *4*: *snd* (*rep-nat-term v*) ≤ *snd* (*rep-nat-term w*)
by (*simp add*: *tp.component-of-term-def assms*(*4*))
note *comparator-nat-term-compare nat-term-comp-nat-term-compare*
moreover have *fst* (*rep-nat-term v*) = *fst* (*rep-nat-term* (*?f* (*tp.pp-of-term v*)))
by (*simp add*: *assms*(*2*) *tp.pp-of-term-def tp.pair-term*)
moreover have *fst* (*rep-nat-term w*) = *fst* (*rep-nat-term* (*?f* (*tp.pp-of-term w*)))
by (*simp add*: *assms*(*2*) *tp.pp-of-term-def tp.pair-term*)
moreover note *4*
moreover have *snd* (*rep-nat-term* (*?f* (*tp.pp-of-term v*))) = *snd* (*rep-nat-term* (*?f* (*tp.pp-of-term w*)))
by (*simp add*: *assms*(*3*))
ultimately show *le-of-nat-term-order cmp-term v w* **unfolding** *le-of-nat-term-order* **using** *3*
by (*rule nat-term-compD4''*)
qed
qed
qed

lemma *gd-term-to-pair-unit*:
gd-term (*to-pair-unit*::*'a*::{*nat-term-compare*,*nat-pp-term*,*graded-dickson-powerprod*} ⇒ -) *fst*
(*λs t*. *le-of-nat-term-order cmp-term* (*fst* (*s*, *the-min*)) (*fst* (*t*, *the-min*)))
(*λs 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 (*intro gd-term.intro ordered-term.intro*)
show *term-powerprod to-pair-unit fst* **by** *unfold-locales*
next
show *ordered-powerprod* (*λs t. le-of-nat-term-order cmp-term* (*fst* (*s*, *the-min*)) (*fst* (*t*, *the-min*)))
(*λs t. lt-of-nat-term-order cmp-term* (*fst* (*s*, *the-min*)) (*fst* (*t*, *the-min*)))
unfolding *fst-conv* **using** *linorder-le-of-nat-term-order*
proof (*intro ordered-powerprod.intro*)
from *le-of-nat-term-order-zero-min* **show** *ordered-powerprod-axioms* (*le-of-nat-term-order cmp-term*)
proof (*unfold-locales*)
fix *s t u*
assume *le-of-nat-term-order cmp-term s t*
hence *le-of-nat-term-order cmp-term* (*u* + *s*) (*u* + *t*) **by** (*rule le-of-nat-term-order-plus-monotone*)
thus *le-of-nat-term-order cmp-term* (*s* + *u*) (*t* + *u*) **by** (*simp only*: *ac-simps*)
qed
qed
next
show *class.linorder* (*le-of-nat-term-order cmp-term*) (*lt-of-nat-term-order cmp-term*)
by (*fact linorder-le-of-nat-term-order*)
next
show *ordered-term-axioms to-pair-unit fst* (*λs t. le-of-nat-term-order cmp-term* (*fst* (*s*, *the-min*)) (*fst* (*t*, *the-min*)))
(*le-of-nat-term-order cmp-term*) **by** (*unfold-locales*, *auto intro*: *le-of-nat-term-order-plus-monotone*)
qed

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*
by (*rule gd-nat-term.intro*, *fact gd-term-to-pair-unit*, *rule gd-nat-term-axioms.intro*, *simp add*: *splus-pp-term*)

lemma *gd-term-id*:
gd-term (*λx*::(*'a*::{*nat-term-compare*,*nat-pp-compare*,*nat-pp-term*,*graded-dickson-powerprod*} × *'b*::{*nat*,*the-min*}). *x*) (*λx*. *x*)
(*λs t. le-of-nat-term-order cmp-term* (*s*, *the-min*) (*t*, *the-min*))
(*λs 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*)
apply (*rule term-powerprod-gd-term*)
subgoal by *unfold-locales*
subgoal by (*simp add*: *rep-nat-term-prod-def*)
subgoal by (*simp add*: *rep-nat-term-prod-def the-min-eq-zero*)
subgoal by (*simp add*: *rep-nat-term-prod-def ord-iff*[*symmetric*])
subgoal by (*simp add*: *splus-prod-def pprod.splus-def*)
subgoal by (*simp add*: *splus-prod-def*)

done

corollary *gd-nat-term-id*: *gd-nat-term* (λx. x) (λx. 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*
by (*rule gd-nat-term.intro*, *fact gd-term-id*, *rule gd-nat-term-axioms.intro*, *simp add*: *splus-prod-def*)

15.4 Computations

type-synonym 'a *mpoly-tc* = (*nat*, *nat*) *pp* $\Rightarrow_0$ 'a

global-interpretation *punit0*: *gd-nat-term to-pair-unit*::'a::{nat-term-compare,nat-pp-term,graded-dickson-po
⇒ - *fst cmp-term*
rewrites *punit.adds-term* = (*adds*)
and *punit.pp-of-term* = (λx. 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*
subgoal by (*fact gd-nat-term-to-pair-unit*)
subgoal by (*fact punit-adds-term*)
subgoal by (*fact punit-pp-of-term*)
subgoal by (*fact punit-component-of-term*)
done

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*)
by (*simp add*: *punit0.list-of-oalist-shift-keys case-prod-beta'*)

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*
by (*intro ext*, *simp add*: *punit0.ord-pp-def*)

lemma *ord-pp-strict-punit-alt* [*code-unfold*]: *ord-pp-strict-punit* = *lt-of-nat-term-order*

by (*intro ext*, *simp add*: *punit0.ord-pp-strict-def*)

lemma *gd-powerprod-ord-pp-punit*: *gd-powerprod* (*ord-pp-punit cmp-term*) (*ord-pp-strict-punit cmp-term*)
unfolding *punit0.ord-pp-def punit0.ord-pp-strict-def* **..**

locale *trivariate*$_0$*-rat*
begin

abbreviation *X*::*rat mpoly-tc* **where** $X \equiv V_0$ (*0*::*nat*)
abbreviation *Y*::*rat mpoly-tc* **where** $Y \equiv V_0$ (*1*::*nat*)
abbreviation *Z*::*rat mpoly-tc* **where** $Z \equiv V_0$ (*2*::*nat*)

end

experiment begin interpretation *trivariate*$_0$*-rat* **.**

value [*code*] *X* ^ *2*

value [*code*] $X^2 * Z + 2 * Y$ ^ $3 * Z^2$

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 ($X^2 * Z + 2 * Y$ ^ $3 * Z^2$) ($X^2 * Z^2 + 2 * Y$ ^ $3 * Z^2$)
by *eval*

lemma
tail-punit DLEX ($X^2 * Z + 2 * Y$ ^ $3 * Z^2$) = $X^2 * Z$
by *eval*

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 ($X^2 * Z + 2 * Y$ ^ $3 * Z^2$) = *sparse*$_0$ [(*1*, *3*), (*2*, *2*)]
by *eval*

lemma
lt-punit DRLEX (*X* + *Y* + *Z*) = *sparse*$_0$ [(*2*, *1*)]
by *eval*

lemma
keys ($X^2 * Z$ ^ $3 + 2 * Y$ ^ $3 * Z^2$) =

$\{sparse_0\ [(0,\ 2),\ (2,\ 3)],\ sparse_0\ [(1,\ 3),\ (2,\ 2)]\}$
by *eval*

lemma
$-\ 1 * X^2 * Z\ \widehat{}\ 7 + -\ 2 * Y\ \widehat{}\ 3 * Z^2 = -\ X^2 * Z\ \widehat{}\ 7 + -\ 2 * Y\ \widehat{}\ 3 * Z^2$
by *eval*

lemma
$X^2 * Z\ \widehat{}\ 7 + 2 * Y\ \widehat{}\ 3 * Z^2 + X^2 * Z\ \widehat{}\ 4 + -\ 2 * Y\ \widehat{}\ 3 * Z^2 = X^2 * Z\ \widehat{}$
$7 + X^2 * Z\ \widehat{}\ 4$
by *eval*

lemma
$X^2 * Z\ \widehat{}\ 7 + 2 * Y\ \widehat{}\ 3 * Z^2 - X^2 * Z\ \widehat{}\ 4 + -\ 2 * Y\ \widehat{}\ 3 * Z^2 =$
$X^2 * Z\ \widehat{}\ 7 - X^2 * Z\ \widehat{}\ 4$
by *eval*

lemma
$lookup\ (X^2 * Z\ \widehat{}\ 7 + 2 * Y\ \widehat{}\ 3 * Z^2 + 2)\ (sparse_0\ [(0,\ 2),\ (2,\ 7)]) = 1$
by *eval*

lemma
$X^2 * Z\ \widehat{}\ 7 + 2 * Y\ \widehat{}\ 3 * Z^2 \neq$
$X^2 * Z\ \widehat{}\ 4 + -\ 2 * Y\ \widehat{}\ 3 * Z^2$
by *eval*

lemma
$0 * X\widehat{}2 * Z\widehat{}7 + 0 * Y\widehat{}3{*}Z^2 = 0$
by *eval*

lemma
$monom\text{-}mult\text{-}punit\ 3\ (sparse_0\ [(1,\ 2{::}nat)])\ (X^2 * Z + 2 * Y\ \widehat{}\ 3 * Z^2) =$
$3 * Y^2 * Z * X^2 + 6 * Y\ \widehat{}\ 5 * Z^2$
by *eval*

lemma
$monomial\ (-4)\ (sparse_0\ [(0,\ 2{::}nat)]) = -\ 4 * X^2$
by *eval*

lemma $monomial\ (0{::}rat)\ (sparse_0\ [(0{::}nat,\ 2{::}nat)]) = 0$
by *eval*

lemma
$(X^2 * Z + 2 * Y\ \widehat{}\ 3 * Z^2) * (X^2 * Z\ \widehat{}\ 3 + -\ 2 * Y\ \widehat{}\ 3 * Z^2) =$
$X\ \widehat{}\ 4 * Z\ \widehat{}\ 4 + -\ 2 * X^2 * Z\ \widehat{}\ 3 * Y\ \widehat{}\ 3 +$
$-\ 4 * Y\ \widehat{}\ 6 * Z\ \widehat{}\ 4 + 2 * Y\ \widehat{}\ 3 * Z\ \widehat{}\ 5 * X^2$
by *eval*

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* **.**

instance proof *standard* **qed** (*transfer*, *rule equal-eq*)

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* ⇒ *nat*) ⇒ (*nat* ⇒ *'a* ⇒ *'a*) ⇒ *bool*
  where *hom-grading-fun d f* ⟷ (∀ *n*. (∀ *s t*. *f n* (*s* + *t*) = *f n s* + *f n t*) ∧
    (∀ *t*. *d* (*f n t*) ≤ *n* ∧ (*d t* ≤ *n* ⟶ *f n t* = *t*)))

definition (**in** *monoid-add*) *hom-grading* :: (*'a* ⇒ *nat*) ⇒ *bool*
  where *hom-grading d* ⟷ (∃ *f*. *hom-grading-fun d f*)

definition (**in** *monoid-add*) *decr-grading* :: (*'a* ⇒ *nat*) ⇒ *nat* ⇒ *'a* ⇒ *'a*
  where *decr-grading d* = (*SOME f*. *hom-grading-fun d f*)

lemma *decr-grading*:
  assumes *hom-grading d*
  shows *hom-grading-fun d* (*decr-grading d*)
proof −
  from *assms* **obtain** *f* **where** *hom-grading-fun d f* **unfolding** *hom-grading-def* **..**
  thus *?thesis* **unfolding** *decr-grading-def* **by** (*metis someI*)
qed

lemma *decr-grading-plus*:
hom-grading d $\Longrightarrow$ *decr-grading d n* (*s* + *t*) = *decr-grading d n s* + *decr-grading d n t*
using *decr-grading* **unfolding** *hom-grading-fun-def* **by** *blast*

lemma *decr-grading-zero*:
assumes *hom-grading d*
shows *decr-grading d n 0* = (*0*::'*a*::*cancel-comm-monoid-add*)
proof −
have *decr-grading d n 0* = *decr-grading d n* (*0* + *0*) **by** *simp*
also from *assms* **have** ... = *decr-grading d n 0* + *decr-grading d n 0* **by** (*rule decr-grading-plus*)
finally show *?thesis* **by** *simp*
qed

lemma *decr-grading-le*: *hom-grading d* $\Longrightarrow$ *d* (*decr-grading d n t*) $\le$ *n*
using *decr-grading* **unfolding** *hom-grading-fun-def* **by** *blast*

lemma *decr-grading-idI*: *hom-grading d* $\Longrightarrow$ *d t* $\le$ *n* $\Longrightarrow$ *decr-grading d n t* = *t*
using *decr-grading* **unfolding** *hom-grading-fun-def* **by** *blast*

class *quasi-pm-powerprod* = *ulcs-powerprod* +
assumes *ex-hgrad*: ∃ *d*::'*a* ⇒ *nat*. *dickson-grading d* ∧ *hom-grading d*
begin

subclass *graded-dickson-powerprod*
proof
from *ex-hgrad* **show** ∃ *d*. *dickson-grading d* **by** *blast*
qed

end

lemma *hom-grading-varnum*:
hom-grading ((*varnum X*)::('*x*::*countable* $\Rightarrow_0$ '*b*::*add-wellorder*) ⇒ *nat*)
proof −
define *f* **where** *f* = (λ*n t*. (*except t* (− (*X* ∪ {*x*. *elem-index x* < *n*})))::'*x* $\Rightarrow_0$ '*b*)
show *?thesis* **unfolding** *hom-grading-def hom-grading-fun-def*
proof (*intro exI allI conjI impI*)
fix *n s t*
show *f n* (*s* + *t*) = *f n s* + *f n t* **by** (*simp only*: *f-def except-plus*)
next
fix *n t*
show *varnum X* (*f n t*) $\le$ *n* **by** (*auto simp*: *varnum-le-iff keys-except f-def*)
next
fix *n t*
show *varnum X t* $\le$ *n* $\Longrightarrow$ *f n t* = *t* **by** (*auto simp*: *f-def except-id-iff varnum-le-iff*)

qed
qed

instance *poly-mapping* :: (*countable*, *add-wellorder*) *quasi-pm-powerprod*
by (*standard*, *intro exI conjI*, *fact dickson-grading-varnum-empty*, *fact hom-grading-varnum*)

context *term-powerprod*
begin

definition *decr-grading-term* :: ($'a \Rightarrow nat$) $\Rightarrow$ *nat* $\Rightarrow$ $'t \Rightarrow 't$
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* :: ($'a \Rightarrow nat$) $\Rightarrow$ *nat* $\Rightarrow$ ($'t \Rightarrow_0 'b$) $\Rightarrow$ ($'t \Rightarrow_0 'b$::*comm-monoid-add*)
where *decr-grading-p d n p* = ($\sum$ *v*∈*keys p*. *monomial* (*lookup p v*) (*decr-grading-term d n v*))

lemma *decr-grading-term-splus*:
hom-grading d $\Longrightarrow$ *decr-grading-term d n* ($t \oplus v$) = *decr-grading d n t* $\oplus$ *decr-grading-term d n v*
by (*simp add*: *decr-grading-term-def term-simps decr-grading-plus splus-def*)

lemma *decr-grading-term-le*: *hom-grading d* $\Longrightarrow$ *d* (*pp-of-term* (*decr-grading-term d n v*)) $\le$ *n*
by (*simp add*: *decr-grading-term-def term-simps decr-grading-le*)

lemma *decr-grading-term-idI*: *hom-grading d* $\Longrightarrow$ *d* (*pp-of-term v*) $\le$ *n* $\Longrightarrow$ *decr-grading-term d n v* = *v*
by (*simp add*: *decr-grading-term-def term-simps decr-grading-idI*)

lemma *punit-decr-grading-term*: *punit.decr-grading-term* = *decr-grading*
by (*intro ext*, *simp add*: *punit.decr-grading-term-def*)

lemma *decr-grading-p-zero*: *decr-grading-p d n 0* = *0*
by (*simp add*: *decr-grading-p-def*)

lemma *decr-grading-p-monomial*: *decr-grading-p d n* (*monomial c v*) = *monomial c* (*decr-grading-term d n v*)
by (*simp add*: *decr-grading-p-def*)

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 −
from *finite-keys finite-keys* **have** *fin*: *finite* (*keys p* ∪ *keys q*) **by** (*rule finite-UnI*)
hence *eq1*: ($\sum$ *v*∈*keys p* ∪ *keys q*. *monomial* (*lookup p v*) (*decr-grading-term d n v*)) =
($\sum$ *v*∈*keys p*. *monomial* (*lookup p v*) (*decr-grading-term d n v*))
proof (*rule sum.mono-neutral-right*)
show ∀ *v*∈*keys p* ∪ *keys q* − *keys p*. *monomial* (*lookup p v*) (*decr-grading-term*

d n v) = *0*
    by (*simp add*: *in-keys-iff*)
  qed *simp*
  from *fin* **have** *eq2*: (∑ *v*∈*keys p* ∪ *keys q*. *monomial* (*lookup q v*) (*decr-grading-term d n v*)) =
      (∑ *v*∈*keys q*. *monomial* (*lookup q v*) (*decr-grading-term d n v*))
  proof (*rule sum.mono-neutral-right*)
    show ∀ *v*∈*keys p* ∪ *keys q* − *keys q*. *monomial* (*lookup q v*) (*decr-grading-term d n v*) = *0*
      by (*simp add*: *in-keys-iff*)
  qed *simp*
  from *fin Poly-Mapping.keys-add*
  have *decr-grading-p d n* (*p* + *q*) =
      (∑ *v*∈*keys p* ∪ *keys q*. *monomial* (*lookup* (*p* + *q*) *v*) (*decr-grading-term d n v*))
    unfolding *decr-grading-p-def*
  proof (*rule sum.mono-neutral-left*)
    show ∀ *v*∈*keys p* ∪ *keys q* − *keys* (*p* + *q*). *monomial* (*lookup* (*p* + *q*) *v*) (*decr-grading-term d n v*) = *0*
      by (*simp add*: *in-keys-iff*)
  qed
  also have ... = (∑ *v*∈*keys p* ∪ *keys q*. *monomial* (*lookup p v*) (*decr-grading-term d n v*)) +
      (∑ *v*∈*keys p* ∪ *keys q*. *monomial* (*lookup q v*) (*decr-grading-term d n v*))
    by (*simp only*: *lookup-add single-add sum.distrib*)
  also have ... = (*decr-grading-p d n p*) + (*decr-grading-p d n q*)
    by (*simp only*: *eq1 eq2 decr-grading-p-def*)
  finally show *?thesis* **.**
qed

corollary *decr-grading-p-sum*: *decr-grading-p d n* (*sum f A*) = (∑ *a*∈*A*. *decr-grading-p d n* (*f a*))
  using *decr-grading-p-zero decr-grading-p-plus* **by** (*rule fun-sum-commute*)

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 (*induct p rule*: *poly-mapping-plus-induct*)
  case *1*
  show *?case* **by** (*simp add*: *decr-grading-p-zero*)
next
  case (*2 p a s*)
  from *assms* **show** *?case*
   by (*simp add*: *monom-mult-dist-right decr-grading-p-plus 2*(*3*) *monom-mult-monomial decr-grading-p-monomial decr-grading-term-splus*)
qed

lemma *decr-grading-p-mult-scalar*:
assumes *hom-grading d*
shows *decr-grading-p d n* ($p \odot q$) = *punit.decr-grading-p d n p* $\odot$ *decr-grading-p d n q*
proof (*induct p rule*: *poly-mapping-plus-induct*)
case *1*
show *?case* **by** (*simp add*: *punit.decr-grading-p-zero decr-grading-p-zero*)
next
case (*2 p a s*)
from *assms* **show** *?case*
by (*simp add*: *mult-scalar-distrib-right decr-grading-p-plus punit.decr-grading-p-plus 2*(*3*)
punit.decr-grading-p-monomial mult-scalar-monomial decr-grading-p-monom-mult punit-decr-grading-term)
qed

lemma *decr-grading-p-keys-subset*: *keys* (*decr-grading-p d n p*) $\subseteq$ *decr-grading-term d n ' keys p*
proof
fix *v*
assume $v \in$ *keys* (*decr-grading-p d n p*)
also have ... $\subseteq$ ($\bigcup u \in$*keys p*. *keys* (*monomial* (*lookup p u*) (*decr-grading-term d n u*)))
unfolding *decr-grading-p-def* **by** (*fact keys-sum-subset*)
finally obtain *u* **where** $u \in$ *keys p* **and** $v \in$ *keys* (*monomial* (*lookup p u*) (*decr-grading-term d n u*)) **..**
from *this*(*2*) **have** *eq*: *v* = *decr-grading-term d n u* **by** (*simp split*: *if-split-asm*)
show $v \in$ *decr-grading-term d n ' keys p* **unfolding** *eq* **using** ‹$u \in$ *keys p*› **by** (*rule imageI*)
qed

lemma *decr-grading-p-idI'*:
assumes *hom-grading d* **and** $\bigwedge v.\ v \in$ *keys p* $\Longrightarrow$ *d* (*pp-of-term v*) $\leq n$
shows *decr-grading-p d n p* = *p*
proof −
have *decr-grading-p d n p* = ($\sum v \in$ *keys p*. *monomial* (*lookup p v*) *v*) **unfolding** *decr-grading-p-def*
using *refl*
proof (*rule sum.cong*)
fix *v*
assume $v \in$ *keys p*
hence *d* (*pp-of-term v*) $\leq n$ **by** (*rule assms*(*2*))
with *assms*(*1*) **have** *decr-grading-term d n v* = *v* **by** (*rule decr-grading-term-idI*)
thus *monomial* (*lookup p v*) (*decr-grading-term d n v*) = *monomial* (*lookup p v*) *v* **by** *simp*
qed
also have ... = *p* **by** (*fact poly-mapping-sum-monomials*)
finally show *?thesis* **.**
qed

end

context *gd-term*
begin

lemma *decr-grading-p-idI*:
  assumes *hom-grading d* **and** $p \in$ *dgrad-p-set d m*
  shows *decr-grading-p d m p* = *p*
proof −
  from *assms*(*2*) **have** $\bigwedge v.\ v \in$ *keys p* $\Longrightarrow$ *d* (*pp-of-term v*) $\leq$ *m*
    by (*auto simp*: *dgrad-p-set-def dgrad-set-def*)
  with *assms*(*1*) **show** *?thesis* **by** (*rule decr-grading-p-idI'*)
qed

lemma *decr-grading-p-dgrad-p-setI*:
  assumes *hom-grading d*
  shows *decr-grading-p d m p* $\in$ *dgrad-p-set d m*
proof (*rule dgrad-p-setI*)
  fix *v*
  assume $v \in$ *keys* (*decr-grading-p d m p*)
  hence $v \in$ *decr-grading-term d m* ‘ *keys p* **using** *decr-grading-p-keys-subset* **..**
  then obtain *u* **where** *v* = *decr-grading-term d m u* **..**
  with *assms* **show** *d* (*pp-of-term v*) $\leq$ *m* **by** (*simp add*: *decr-grading-term-le*)
qed

lemma (**in** *gd-term*) *in-pmdlE-dgrad-p-set*:
  assumes *hom-grading d* **and** $B \subseteq$ *dgrad-p-set d m* **and** $p \in$ *dgrad-p-set d m* **and** $p \in$ *pmdl B*
  obtains *A q* **where** *finite A* **and** $A \subseteq B$ **and** $\bigwedge b.\ q\ b \in$ *punit.dgrad-p-set d m*
    and $p = (\sum b{\in}A.\ q\ b \odot b)$
proof −
  from *assms*(*4*) **obtain** *A q0* **where** *finite A* **and** $A \subseteq B$ **and** *p*: $p = (\sum b{\in}A.\ q0\ b \odot b)$
    by (*rule pmdl.spanE*)
  define *q* **where** $q = (\lambda b.$ *punit.decr-grading-p d m* (*q0 b*))
  from ‹*finite A*› ‹$A \subseteq B$› **show** *?thesis*
  proof
    fix *b*
    show *q b* $\in$ *punit.dgrad-p-set d m* **unfolding** *q-def* **using** *assms*(*1*) **by** (*rule punit.decr-grading-p-dgrad-p-setI*)
  next
    from *assms*(*1*, *3*) **have** *p* = *decr-grading-p d m p* **by** (*simp only*: *decr-grading-p-idI*)
    also from *assms*(*1*) **have** ... = $(\sum b{\in}A.\ q\ b \odot$ (*decr-grading-p d m b*))
      by (*simp add*: *p q-def decr-grading-p-sum decr-grading-p-mult-scalar*)
    also from *refl* **have** ... = $(\sum b{\in}A.\ q\ b \odot b)$
    proof (*rule sum.cong*)
      fix *b*
      assume $b \in A$

hence $b \in B$ **using** ‹$A \subseteq B$› **..**
hence $b \in$ *dgrad-p-set d m* **using** *assms*(*2*) **..**
with *assms*(*1*) **have** *decr-grading-p d m b* = *b* **by** (*rule decr-grading-p-idI*)
thus *q b* ⊙ *decr-grading-p d m b* = *q b* ⊙ *b* **by** *simp*
qed
finally show $p = (\sum b{\in}A.\ q\ b \odot b)$ **.**
qed
qed

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-Mapping.single x k*)) ^ *n* = *monomial* (*c* ^ *n*) (*Poly-Mapping.single x* (*k* ∗ *n*))
proof −
have *eq*: ($\sum$ *i* = *0*..<*n*. *Poly-Mapping.single x k*) = *Poly-Mapping.single x* (*k* ∗ *n*)
by (*induct n*, *simp-all add*: *add.commute single-add*)
show *?thesis* **by** (*simp add*: *punit.monomial-power eq*)
qed

lemma *monomial-power-map-scale*: (*monomial c t*) ^ *n* = *monomial* (*c* ^ *n*) (*n* · *t*)
proof −
have ($\sum$ *i* = *0*..<*n*. *t*) = ($\sum$ *i* = *0*..<*n*. *1*) · *t*
by (*simp only*: *map-scale-sum-distrib-right map-scale-one-left*)
thus *?thesis* **by** (*simp add*: *punit.monomial-power*)
qed

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 (*rule ccontr*)
assume *p* ≠ *q*
hence *p* − *q* ≠ *0* **by** *simp*
with *assms*(*2*) **have** *h* ∗ (*p* − *q*) ≠ *0* **by** *simp*
hence *h* ∗ *p* ≠ *h* ∗ *q* **by** (*simp add*: *algebra-simps*)

thus *False* **using** *assms*(*1*) **..**
qed

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 (*rule ccontr*)
assume $p \neq q$
hence $p - q \neq 0$ **by** *simp*
hence $(p - q) * h \neq 0$ **using** *assms*(*2*) **by** *simp*
hence $p * h \neq q * h$ **by** (*simp add*: *algebra-simps*)
thus *False* **using** *assms*(*1*) **..**
qed

17.1 Degree

lemma *plus-minus-assoc-pm-nat-1*: $s + t - u = (s - (u - t)) + (t - (u::- \Rightarrow_0 nat))$
by (*rule poly-mapping-eqI*, *simp add*: *lookup-add lookup-minus*)

lemma *plus-minus-assoc-pm-nat-2*:
$s + (t - u) = (s + (except\ (u - t)\ (- keys\ s))) + t - (u::- \Rightarrow_0 nat)$
proof (*rule poly-mapping-eqI*)
fix *x*
show $lookup\ (s + (t - u))\ x = lookup\ (s + except\ (u - t)\ (- keys\ s) + t - u)\ x$
proof (*cases* $x \in keys\ s$)
case *True*
thus *?thesis*
by (*simp add*: *plus-minus-assoc-pm-nat-1 lookup-add lookup-minus lookup-except*)
next
case *False*
hence $lookup\ s\ x = 0$ **by** (*simp add*: *in-keys-iff*)
with *False* **show** *?thesis*
by (*simp add*: *lookup-add lookup-minus lookup-except*)
qed
qed

lemma *deg-pm-sum*: $deg\text{-}pm\ (sum\ t\ A) = (\sum a \in A.\ deg\text{-}pm\ (t\ a))$
by (*induct A rule*: *infinite-finite-induct*) (*auto simp*: *deg-pm-plus*)

lemma *deg-pm-mono*: $s\ adds\ t \Longrightarrow deg\text{-}pm\ s \leq deg\text{-}pm\ (t::- \Rightarrow_0 -::add\text{-}linorder\text{-}min)$
by (*metis addsE deg-pm-plus le-iff-add*)

lemma *adds-deg-pm-antisym*: $s\ adds\ t \Longrightarrow deg\text{-}pm\ t \leq deg\text{-}pm\ (s::- \Rightarrow_0 -::add\text{-}linorder\text{-}min) \Longrightarrow s = t$
by (*metis* (*no-types*, *lifting*) *add.right-neutral add.right-neutral add-left-cancel addsE*
deg-pm-eq-0-iff deg-pm-mono deg-pm-plus dual-order.antisym)

lemma *deg-pm-minus*:
  assumes *s adds* (*t*::- $\Rightarrow_0$ -::*comm-monoid-add*)
  shows *deg-pm* (*t* − *s*) = *deg-pm t* − *deg-pm s*
proof −
  from *assms* **have** (*t* − *s*) + *s* = *t* **by** (*rule adds-minus*)
  hence *deg-pm t* = *deg-pm* ((*t* − *s*) + *s*) **by** *simp*
  also have ... = *deg-pm* (*t* − *s*) + *deg-pm s* **by** (*simp only*: *deg-pm-plus*)
  finally show *?thesis* **by** *simp*
qed

lemma *adds-group* [*simp*]: *s adds* (*t*::'*a* $\Rightarrow_0$ '*b*::*ab-group-add*)
proof (*rule addsI*)
  show *t* = *s* + (*t* − *s*) **by** *simp*
qed

lemmas *deg-pm-minus-group* = *deg-pm-minus*[*OF adds-group*]

lemma *deg-pm-minus-le*: *deg-pm* (*t* − *s*) ≤ *deg-pm* (*t*::- $\Rightarrow_0$ *nat*)
proof −
  have *keys* (*t* − *s*) ⊆ *keys t* **by** (*rule*, *simp add*: *lookup-minus in-keys-iff*)
  hence *deg-pm* (*t* − *s*) = (∑*x*∈*keys t*. *lookup* (*t* − *s*) *x*) **using** *finite-keys* **by** (*rule deg-pm-superset*)
  also have ... ≤ (∑*x*∈*keys t*. *lookup t x*) **by** (*rule sum-mono*) (*simp add*: *lookup-minus*)
  also have ... = *deg-pm t* **by** (*rule sym*, *rule deg-pm-superset*, *fact subset-refl*, *fact finite-keys*)
  finally show *?thesis* **.**
qed

lemma *minus-id-iff*: *t* − *s* = *t* ⟷ *keys t* ∩ *keys* (*s*::- $\Rightarrow_0$ *nat*) = {}
proof
  assume *t* − *s* = *t*
  {
    fix *x*
    assume *x* ∈ *keys t* **and** *x* ∈ *keys s*
    hence *0* < *lookup t x* **and** *0* < *lookup s x* **by** (*simp-all add*: *in-keys-iff*)
    hence *lookup* (*t* − *s*) *x* ≠ *lookup t x* **by** (*simp add*: *lookup-minus*)
    with ‹*t* − *s* = *t*› **have** *False* **by** *simp*
  }
  thus *keys t* ∩ *keys s* = {} **by** *blast*
next
  assume ∗: *keys t* ∩ *keys s* = {}
  show *t* − *s* = *t*
  proof (*rule poly-mapping-eqI*)
    fix *x*
    have *lookup t x* − *lookup s x* = *lookup t x*
    proof (*cases x* ∈ *keys t*)
      case *True*
      with ∗ **have** *x* ∉ *keys s* **by** *blast*

thus *?thesis* **by** (*simp add*: *in-keys-iff*)
next
case *False*
thus *?thesis* **by** (*simp add*: *in-keys-iff*)
qed
thus *lookup* (*t* − *s*) *x* = *lookup t x* **by** (*simp only*: *lookup-minus*)
qed
qed

lemma *deg-pm-minus-id-iff*: *deg-pm* (*t* − *s*) = *deg-pm t* ⟷ *keys t* ∩ *keys* (*s*::- ⇒$_0$ *nat*) = {}
proof
assume *eq*: *deg-pm* (*t* − *s*) = *deg-pm t*
{
fix *x*
assume *x* ∈ *keys t* **and** *x* ∈ *keys s*
hence *0* < *lookup t x* **and** *0* < *lookup s x* **by** (*simp-all add*: *in-keys-iff*)
hence ∗: *lookup* (*t* − *s*) *x* < *lookup t x* **by** (*simp add*: *lookup-minus*)
have *keys* (*t* − *s*) ⊆ *keys t* **by** (*rule*, *simp add*: *lookup-minus in-keys-iff*)
hence *deg-pm* (*t* − *s*) = (∑ *x*∈*keys t*. *lookup* (*t* − *s*) *x*) **using** *finite-keys* **by** (*rule deg-pm-superset*)
also from *finite-keys* **have** ... < (∑ *x*∈*keys t*. *lookup t x*)
proof (*rule sum-strict-mono-ex1*)
show ∀ *x*∈*keys t*. *lookup* (*t* − *s*) *x* ≤ *lookup t x* **by** (*simp add*: *lookup-minus*)
next
from ‹*x* ∈ *keys t*› ∗ **show** ∃ *x*∈*keys t*. *lookup* (*t* − *s*) *x* < *lookup t x* **..**
qed
also have ... = *deg-pm t* **by** (*rule sym*, *rule deg-pm-superset*, *fact subset-refl*, *fact finite-keys*)
finally have *False* **by** (*simp add*: *eq*)
}
thus *keys t* ∩ *keys s* = {} **by** *blast*
next
assume *keys t* ∩ *keys s* = {}
hence *t* − *s* = *t* **by** (*simp only*: *minus-id-iff*)
thus *deg-pm* (*t* − *s*) = *deg-pm t* **by** (*simp only*:)
qed

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*)

by (*simp add*: *poly-deg-def*)

lemma *poly-deg-monomial-zero* [*simp*]: *poly-deg* (*monomial c 0*) = *0*
by (*simp add*: *poly-deg-monomial*)

lemma *poly-deg-zero* [*simp*]: *poly-deg 0* = *0*
by (*simp only*: *single-zero*[*of 0*, *symmetric*] *poly-deg-monomial-zero*)

lemma *poly-deg-one* [*simp*]: *poly-deg 1* = *0*
by (*simp only*: *single-one*[*symmetric*] *poly-deg-monomial-zero*)

lemma *poly-degE*:
assumes $p \neq 0$
obtains *t* **where** $t \in$ *keys p* **and** *poly-deg p* = *deg-pm t*
proof −
from *assms* **have** *poly-deg p* = *Max* (*deg-pm* ‘ *keys p*) **by** (*simp add*: *poly-deg-def*)
also have ... ∈ *deg-pm* ‘ *keys p*
proof (*rule Max-in*)
from *assms* **show** *deg-pm* ‘ *keys p* ≠ {} **by** *simp*
qed *simp*
finally obtain *t* **where** $t \in$ *keys p* **and** *poly-deg p* = *deg-pm t* **..**
thus *?thesis* **..**
qed

lemma *poly-deg-max-keys*: $t \in$ *keys p* ⟹ *deg-pm t* ≤ *poly-deg p*
using *finite-keys* **by** (*auto simp*: *poly-deg-def*)

lemma *poly-deg-leI*: (⋀*t*. $t \in$ *keys p* ⟹ *deg-pm t* ≤ (*d*::′*a*::*add-linorder-min*)) ⟹ *poly-deg p* ≤ *d*
using *finite-keys* **by** (*auto simp*: *poly-deg-def*)

lemma *poly-deg-lessI*:
$p \neq 0$ ⟹ (⋀*t*. $t \in$ *keys p* ⟹ *deg-pm t* < (*d*::′*a*::*add-linorder-min*)) ⟹ *poly-deg p* < *d*
using *finite-keys* **by** (*auto simp*: *poly-deg-def*)

lemma *poly-deg-zero-imp-monomial*:
assumes *poly-deg p* = (*0*::′*a*::*add-linorder-min*)
shows *monomial* (*lookup p 0*) *0* = *p*
proof (*rule keys-subset-singleton-imp-monomial*, *rule*)
fix *t*
assume $t \in$ *keys p*
have *t* = *0*
proof (*rule ccontr*)
assume $t \neq 0$
hence *deg-pm t* ≠ *0* **by** *simp*
hence *0* < *deg-pm t* **using** *not-gr-zero* **by** *blast*
also from ‹$t \in$ *keys p*› **have** ... ≤ *poly-deg p* **by** (*rule poly-deg-max-keys*)
finally have *poly-deg p* ≠ *0* **by** *simp*

from *this assms* **show** *False* **..**
qed
thus $t \in \{0\}$ **by** *simp*
qed

lemma *poly-deg-plus-le*:
poly-deg (*p* + *q*) ≤ *max* (*poly-deg p*) (*poly-deg* (*q*::(- ⇒$_0$ ′*a*::*add-linorder-min*) ⇒$_0$ -))
proof (*rule poly-deg-leI*)
fix *t*
assume *t* ∈ *keys* (*p* + *q*)
also have ... ⊆ *keys p* ∪ *keys q* **by** (*fact Poly-Mapping.keys-add*)
finally show *deg-pm t* ≤ *max* (*poly-deg p*) (*poly-deg q*)
proof
assume *t* ∈ *keys p*
hence *deg-pm t* ≤ *poly-deg p* **by** (*rule poly-deg-max-keys*)
thus *?thesis* **by** (*simp add*: *le-max-iff-disj*)
next
assume *t* ∈ *keys q*
hence *deg-pm t* ≤ *poly-deg q* **by** (*rule poly-deg-max-keys*)
thus *?thesis* **by** (*simp add*: *le-max-iff-disj*)
qed
qed

lemma *poly-deg-uminus* [*simp*]: *poly-deg* (−*p*) = *poly-deg p*
by (*simp add*: *poly-deg-def keys-uminus*)

lemma *poly-deg-minus-le*:
poly-deg (*p* − *q*) ≤ *max* (*poly-deg p*) (*poly-deg* (*q*::(- ⇒$_0$ ′*a*::*add-linorder-min*) ⇒$_0$ -))
proof (*rule poly-deg-leI*)
fix *t*
assume *t* ∈ *keys* (*p* − *q*)
also have ... ⊆ *keys p* ∪ *keys q* **by** (*fact keys-minus*)
finally show *deg-pm t* ≤ *max* (*poly-deg p*) (*poly-deg q*)
proof
assume *t* ∈ *keys p*
hence *deg-pm t* ≤ *poly-deg p* **by** (*rule poly-deg-max-keys*)
thus *?thesis* **by** (*simp add*: *le-max-iff-disj*)
next
assume *t* ∈ *keys q*
hence *deg-pm t* ≤ *poly-deg q* **by** (*rule poly-deg-max-keys*)
thus *?thesis* **by** (*simp add*: *le-max-iff-disj*)
qed
qed

lemma *poly-deg-times-le*:
poly-deg (*p* ∗ *q*) ≤ *poly-deg p* + *poly-deg* (*q*::(- ⇒$_0$ ′*a*::*add-linorder-min*) ⇒$_0$ -)
proof (*rule poly-deg-leI*)

fix *t*
assume *t* ∈ *keys* (*p* ∗ *q*)
then obtain *u v* **where** *u* ∈ *keys p* **and** *v* ∈ *keys q* **and** *t* = *u* + *v* **by** (*rule in-keys-timesE*)
from ‹*u* ∈ *keys p*› **have** *deg-pm u* ≤ *poly-deg p* **by** (*rule poly-deg-max-keys*)
moreover from ‹*v* ∈ *keys q*› **have** *deg-pm v* ≤ *poly-deg q* **by** (*rule poly-deg-max-keys*)
ultimately show *deg-pm t* ≤ *poly-deg p* + *poly-deg q* **by** (*simp add*: ‹*t* = *u* + *v*› *deg-pm-plus add-mono*)
qed

lemma *poly-deg-times*:
assumes *p* ≠ *0* **and** *q* ≠ (*0*::(*'x*::*linorder* ⇒₀ *'a*::*add-linorder-min*) ⇒₀ *'b*::*semiring-no-zero-divisors*)
shows *poly-deg* (*p* ∗ *q*) = *poly-deg p* + *poly-deg q*
using *poly-deg-times-le*
proof (*rule antisym*)
let *?A* = λ*f*. {*u*. *deg-pm u* < *poly-deg f*}
define *p1* **where** *p1* = *except p* (*?A p*)
define *p2* **where** *p2* = *except p* (− *?A p*)
define *q1* **where** *q1* = *except q* (*?A q*)
define *q2* **where** *q2* = *except q* (− *?A q*)
have *deg-p1*: *deg-pm t* = *poly-deg p* **if** *t* ∈ *keys p1* **for** *t*
proof −
from *that* **have** *t* ∈ *keys p* **and** *poly-deg p* ≤ *deg-pm t*
by (*simp-all add*: *p1-def keys-except not-less*)
from *this*(*1*) **have** *deg-pm t* ≤ *poly-deg p* **by** (*rule poly-deg-max-keys*)
thus *?thesis* **using** ‹*poly-deg p* ≤ *deg-pm t*› **by** (*rule antisym*)
qed
have *deg-p2*: *t* ∈ *keys p2* ⟹ *deg-pm t* < *poly-deg p* **for** *t* **by** (*simp add*: *p2-def keys-except*)
have *deg-q1*: *deg-pm t* = *poly-deg q* **if** *t* ∈ *keys q1* **for** *t*
proof −
from *that* **have** *t* ∈ *keys q* **and** *poly-deg q* ≤ *deg-pm t*
by (*simp-all add*: *q1-def keys-except not-less*)
from *this*(*1*) **have** *deg-pm t* ≤ *poly-deg q* **by** (*rule poly-deg-max-keys*)
thus *?thesis* **using** ‹*poly-deg q* ≤ *deg-pm t*› **by** (*rule antisym*)
qed
have *deg-q2*: *t* ∈ *keys q2* ⟹ *deg-pm t* < *poly-deg q* **for** *t* **by** (*simp add*: *q2-def keys-except*)
have *p*: *p* = *p1* + *p2* **unfolding** *p1-def p2-def* **by** (*fact except-decomp*)
have *p1* ≠ *0*
proof −
from *assms*(*1*) **obtain** *t* **where** *t* ∈ *keys p* **and** *poly-deg p* = *deg-pm t* **by** (*rule poly-degE*)
hence *t* ∈ *keys p1* **by** (*simp add*: *p1-def keys-except*)
thus *?thesis* **by** *auto*
qed
have *q*: *q* = *q1* + *q2* **unfolding** *q1-def q2-def* **by** (*fact except-decomp*)
have *q1* ≠ *0*
proof −

from *assms*(*2*) **obtain** *t* **where** *t* ∈ *keys q* **and** *poly-deg q* = *deg-pm t* **by** (*rule poly-degE*)
  hence *t* ∈ *keys q1* **by** (*simp add*: *q1-def keys-except*)
  thus *?thesis* **by** *auto*
qed
with ‹*p1* ≠ *0*› **have** *p1* ∗ *q1* ≠ *0* **by** *simp*
hence *keys* (*p1* ∗ *q1*) ≠ {} **by** *simp*
then obtain *u* **where** *u* ∈ *keys* (*p1* ∗ *q1*) **by** *blast*
then obtain *s t* **where** *s* ∈ *keys p1* **and** *t* ∈ *keys q1* **and** *u*: *u* = *s* + *t* **by** (*rule in-keys-timesE*)
from ‹*s* ∈ *keys p1*› **have** *deg-pm s* = *poly-deg p* **by** (*rule deg-p1*)
moreover from ‹*t* ∈ *keys q1*› **have** *deg-pm t* = *poly-deg q* **by** (*rule deg-q1*)
ultimately have *eq*: *poly-deg p* + *poly-deg q* = *deg-pm u* **by** (*simp only*: *u deg-pm-plus*)
also have . . . ≤ *poly-deg* (*p* ∗ *q*)
proof (*rule poly-deg-max-keys*)
  have *u* ∉ *keys* (*p1* ∗ *q2* + *p2* ∗ *q*)
  proof
    assume *u* ∈ *keys* (*p1* ∗ *q2* + *p2* ∗ *q*)
    also have . . . ⊆ *keys* (*p1* ∗ *q2*) ∪ *keys* (*p2* ∗ *q*) **by** (*rule Poly-Mapping.keys-add*)
    finally have *deg-pm u* < *poly-deg p* + *poly-deg q*
    proof
      assume *u* ∈ *keys* (*p1* ∗ *q2*)
      then obtain *s′ t′* **where** *s′* ∈ *keys p1* **and** *t′* ∈ *keys q2* **and** *u*: *u* = *s′* + *t′*
        by (*rule in-keys-timesE*)
      from ‹*s′* ∈ *keys p1*› **have** *deg-pm s′* = *poly-deg p* **by** (*rule deg-p1*)
      moreover from ‹*t′* ∈ *keys q2*› **have** *deg-pm t′* < *poly-deg q* **by** (*rule deg-q2*)
      ultimately show *?thesis* **by** (*simp add*: *u deg-pm-plus*)
    next
      assume *u* ∈ *keys* (*p2* ∗ *q*)
      then obtain *s′ t′* **where** *s′* ∈ *keys p2* **and** *t′* ∈ *keys q* **and** *u*: *u* = *s′* + *t′*
        by (*rule in-keys-timesE*)
      from ‹*s′* ∈ *keys p2*› **have** *deg-pm s′* < *poly-deg p* **by** (*rule deg-p2*)
      moreover from ‹*t′* ∈ *keys q*› **have** *deg-pm t′* ≤ *poly-deg q* **by** (*rule poly-deg-max-keys*)
      ultimately show *?thesis* **by** (*simp add*: *u deg-pm-plus add-less-le-mono*)
    qed
    thus *False* **by** (*simp only*: *eq*)
  qed
  with ‹*u* ∈ *keys* (*p1* ∗ *q1*)› **have** *u* ∈ *keys* (*p1* ∗ *q1* + (*p1* ∗ *q2* + *p2* ∗ *q*)) **by** (*rule in-keys-plusI1*)
  thus *u* ∈ *keys* (*p* ∗ *q*) **by** (*simp only*: *p q algebra-simps*)
qed
finally show *poly-deg p* + *poly-deg q* ≤ *poly-deg* (*p* ∗ *q*) **.**
qed

corollary *poly-deg-monom-mult-le*:
poly-deg (*punit.monom-mult c* (*t*::- $\Rightarrow_0$ *′a*::*add-linorder-min*) *p*) ≤ *deg-pm t* + *poly-deg p*

proof −
have *poly-deg* (*punit.monom-mult c t p*) ≤ *poly-deg* (*monomial c t*) + *poly-deg p*
by (*simp only*: *times-monomial-left*[*symmetric*] *poly-deg-times-le*)
also have ... ≤ *deg-pm t* + *poly-deg p* **by** (*simp add*: *poly-deg-monomial*)
finally show *?thesis* **.**
qed

lemma *poly-deg-monom-mult*:
assumes *c* ≠ *0* **and** *p* ≠ (*0*::(- ⇒$_0$ ′*a*::*add-linorder-min*) ⇒$_0$ ′*b*::*semiring-no-zero-divisors*)
shows *poly-deg* (*punit.monom-mult c t p*) = *deg-pm t* + *poly-deg p*
proof (*rule order.antisym*, *fact poly-deg-monom-mult-le*)
from *assms*(*2*) **obtain** *s* **where** *s* ∈ *keys p* **and** *poly-deg p* = *deg-pm s* **by** (*rule poly-degE*)
have *deg-pm t* + *poly-deg p* = *deg-pm* (*t* + *s*) **by** (*simp add*: ‹*poly-deg p* = *deg-pm s*› *deg-pm-plus*)
also have ... ≤ *poly-deg* (*punit.monom-mult c t p*)
proof (*rule poly-deg-max-keys*)
from ‹*s* ∈ *keys p*› **show** *t* + *s* ∈ *keys* (*punit.monom-mult c t p*)
unfolding *punit.keys-monom-mult*[*OF assms*(*1*)] **by** *fastforce*
qed
finally show *deg-pm t* + *poly-deg p* ≤ *poly-deg* (*punit.monom-mult c t p*) **.**
qed

lemma *poly-deg-map-scale*:
poly-deg (*c* · *p*) = (*if c* = (*0*::-::*semiring-no-zero-divisors*) *then 0 else poly-deg p*)
by (*simp add*: *poly-deg-def keys-map-scale*)

lemma *poly-deg-sum-le*: ((*poly-deg* (*sum f A*))::′*a*::*add-linorder-min*) ≤ *Max* (*poly-deg* ‘ *f* ‘ *A*)
proof (*cases finite A*)
case *True*
thus *?thesis*
proof (*induct A*)
case *empty*
show *?case* **by** *simp*
next
case (*insert a A*)
show *?case*
proof (*cases A* = {})
case *True*
thus *?thesis* **by** *simp*
next
case *False*
have *poly-deg* (*sum f* (*insert a A*)) ≤ *max* (*poly-deg* (*f a*)) (*poly-deg* (*sum f A*))
by (*simp only*: *comm-monoid-add-class.sum.insert*[*OF insert*(*1*) *insert*(*2*)] *poly-deg-plus-le*)
also have ... ≤ *max* (*poly-deg* (*f a*)) (*Max* (*poly-deg* ‘ *f* ‘ *A*))
using *insert*(*3*) *max.mono* **by** *blast*

also have ... = (*Max* (*poly-deg* ‘ *f* ‘ (*insert a A*))) **using** *False* **by** (*simp add*: *insert*(*1*))
finally show *?thesis* **.**
qed
qed
next
case *False*
thus *?thesis* **by** *simp*
qed

lemma *poly-deg-prod-le*: ((*poly-deg* (*prod f A*))::′*a*::*add-linorder-min*) ≤ (∑ *a*∈*A*. *poly-deg* (*f a*))
proof (*cases finite A*)
case *True*
thus *?thesis*
proof (*induct A*)
case *empty*
show *?case* **by** *simp*
next
case (*insert a A*)
have *poly-deg* (*prod f* (*insert a A*)) ≤ (*poly-deg* (*f a*)) + (*poly-deg* (*prod f A*))
by (*simp only*: *comm-monoid-mult-class.prod.insert*[*OF insert*(*1*) *insert*(*2*)] *poly-deg-times-le*)
also have ... ≤ (*poly-deg* (*f a*)) + (∑ *a*∈*A*. *poly-deg* (*f a*))
using *insert*(*3*) *add-le-cancel-left* **by** *blast*
also have ... = (∑ *a*∈*insert a A*. *poly-deg* (*f a*)) **by** (*simp add*: *insert*(*1*) *insert*(*2*))
finally show *?case* **.**
qed
next
case *False*
thus *?thesis* **by** *simp*
qed

lemma *maxdeg-max*:
assumes *finite A* **and** *p* ∈ *A*
shows *poly-deg p* ≤ *maxdeg A*
unfolding *maxdeg-def* **using** *assms* **by** *auto*

lemma *mindeg-min*:
assumes *finite A* **and** *p* ∈ *A*
shows *mindeg A* ≤ *poly-deg p*
unfolding *mindeg-def* **using** *assms* **by** *auto*

17.2 Indeterminates

definition *indets* :: ((′*x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ ′*b*::*zero*) ⇒ ′*x set*
where *indets p* = ⋃ (*keys* ‘ *keys p*)

definition *PPs* :: *'x set* ⇒ (*'x* $\Rightarrow_0$ *nat*) *set* (‹.[(-)]›)
  where *PPs X* = {*t*. *keys t* ⊆ *X*}

definition *Polys* :: *'x set* ⇒ ((*'x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'b*::*zero*) *set* (‹*P*[(-)]›)
  where *Polys X* = {*p*. *keys p* ⊆ .[*X*]}

17.2.1 *indets*

lemma *in-indetsI*:
  assumes *x* ∈ *keys t* **and** *t* ∈ *keys p*
  shows *x* ∈ *indets p*
  using *assms* **by** (*auto simp add*: *indets-def*)

lemma *in-indetsE*:
  assumes *x* ∈ *indets p*
  obtains *t* **where** *t* ∈ *keys p* **and** *x* ∈ *keys t*
  using *assms* **by** (*auto simp add*: *indets-def*)

lemma *keys-subset-indets*: *t* ∈ *keys p* ⟹ *keys t* ⊆ *indets p*
  by (*auto dest*: *in-indetsI*)

lemma *indets-empty-imp-monomial*:
  assumes *indets p* = {}
  shows *monomial* (*lookup p 0*) *0* = *p*
proof (*rule keys-subset-singleton-imp-monomial*, *rule*)
  fix *t*
  assume *t* ∈ *keys p*
  have *t* = *0*
  proof (*rule ccontr*)
    assume *t* ≠ *0*
    hence *keys t* ≠ {} **by** *simp*
    then obtain *x* **where** *x* ∈ *keys t* **by** *blast*
    from *this* ‹*t* ∈ *keys p*› **have** *x* ∈ *indets p* **by** (*rule in-indetsI*)
    with *assms* **show** *False* **by** *simp*
  qed
  thus *t* ∈ {*0*} **by** *simp*
qed

lemma *finite-indets*: *finite* (*indets p*)
  by (*simp only*: *indets-def*, *rule finite-UN-I*, (*rule finite-keys*)+)

lemma *indets-zero* [*simp*]: *indets 0* = {}
  by (*simp add*: *indets-def*)

lemma *indets-one* [*simp*]: *indets 1* = {}
  by (*simp add*: *indets-def*)

lemma *indets-monomial-single-subset*: *indets* (*monomial c* (*Poly-Mapping.single v k*)) ⊆ {*v*}

proof
  fix *x* **assume** $x \in$ *indets* (*monomial c* (*Poly-Mapping.single v k*))
  then have $x = v$ **unfolding** *indets-def*
    by (*metis UN-E lookup-eq-zero-in-keys-contradict lookup-single-not-eq*)
  thus $x \in \{v\}$ **by** *simp*
qed

lemma *indets-monomial-single*:
  assumes $c \neq 0$ **and** $k \neq 0$
  shows *indets* (*monomial c* (*Poly-Mapping.single v k*)) = $\{v\}$
proof (*rule*, *fact indets-monomial-single-subset*, *simp*)
  from *assms* **show** $v \in$ *indets* (*monomial c* (*monomial k v*)) **by** (*simp add*: *indets-def*)
qed

lemma *indets-monomial*:
  assumes $c \neq 0$
  shows *indets* (*monomial c t*) = *keys t*
proof (*rule antisym*; *rule subsetI*)
  fix *x*
  assume $x \in$ *indets* (*monomial c t*)
  then have *lookup t x* $\neq 0$ **unfolding** *indets-def*
    by (*metis UN-E lookup-eq-zero-in-keys-contradict lookup-single-not-eq*)
  thus $x \in$ *keys t* **by** (*meson lookup-not-eq-zero-eq-in-keys*)
next
  fix *x*
  assume $x \in$ *keys t*
  then have *lookup t x* $\neq 0$ **by** (*meson lookup-not-eq-zero-eq-in-keys*)
  thus $x \in$ *indets* (*monomial c t*) **unfolding** *indets-def* **using** *assms*
    by (*metis UN-iff lookup-not-eq-zero-eq-in-keys lookup-single-eq*)
qed

lemma *indets-monomial-subset*: *indets* (*monomial c t*) $\subseteq$ *keys t*
  by (*cases* $c = 0$, *simp-all add*: *indets-def*)

lemma *indets-monomial-zero* [*simp*]: *indets* (*monomial c 0*) = {}
  by (*simp add*: *indets-def*)

lemma *indets-plus-subset*: *indets* $(p + q) \subseteq$ *indets p* $\cup$ *indets q*
proof
  fix *x*
  assume $x \in$ *indets* $(p + q)$
  then obtain *t* **where** $x \in$ *keys t* **and** $t \in$ *keys* $(p + q)$ **by** (*metis UN-E indets-def*)
  hence $t \in$ *keys p* $\cup$ *keys q* **by** (*metis Poly-Mapping.keys-add subsetCE*)
  thus $x \in$ *indets p* $\cup$ *indets q* **using** *indets-def* ‹$x \in$ *keys t*› **by** *fastforce*
qed

lemma *indets-uminus* [*simp*]: *indets* $(-p)$ = *indets p*
  by (*simp add*: *indets-def keys-uminus*)

lemma *indets-minus-subset*: *indets* (*p* − *q*) ⊆ *indets p* ∪ *indets q*
proof
fix *x*
assume *x* ∈ *indets* (*p* − *q*)
then obtain *t* **where** *x* ∈ *keys t* **and** *t* ∈ *keys* (*p* − *q*) **by** (*metis UN-E indets-def*)
hence *t* ∈ *keys p* ∪ *keys q* **by** (*metis keys-minus subsetCE*)
thus *x* ∈ *indets p* ∪ *indets q* **using** *indets-def* ‹*x* ∈ *keys t*› **by** *fastforce*
qed

lemma *indets-times-subset*: *indets* (*p* ∗ *q*) ⊆ *indets p* ∪ *indets* (*q*::(- ⇒$_0$ -::*cancel-comm-monoid-add*) ⇒$_0$ -)
proof
fix *x*
assume *x* ∈ *indets* (*p* ∗ *q*)
then obtain *t* **where** *t* ∈ *keys* (*p* ∗ *q*) **and** *x* ∈ *keys t* **unfolding** *indets-def* **by** *blast*
from *this*(*1*) **obtain** *u v* **where** *u* ∈ *keys p v* ∈ *keys q* **and** *t* = *u* + *v* **by** (*rule in-keys-timesE*)
hence *x* ∈ *keys u* ∪ *keys v* **by** (*metis* ‹*x* ∈ *keys t*› *Poly-Mapping.keys-add subsetCE*)
thus *x* ∈ *indets p* ∪ *indets q* **unfolding** *indets-def* **using** ‹*u* ∈ *keys p*› ‹*v* ∈ *keys q*› **by** *blast*
qed

corollary *indets-monom-mult-subset*: *indets* (*punit.monom-mult c t p*) ⊆ *keys t* ∪ *indets p*
proof −
have *indets* (*punit.monom-mult c t p*) ⊆ *indets* (*monomial c t*) ∪ *indets p*
by (*simp only*: *times-monomial-left*[*symmetric*] *indets-times-subset*)
also have ... ⊆ *keys t* ∪ *indets p* **using** *indets-monomial-subset*[*of t c*] **by** *blast*
finally show *?thesis* **.**
qed

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 (*rule*, *fact indets-monom-mult-subset*, *rule*)
fix *x*
assume *x* ∈ *keys t* ∪ *indets p*
thus *x* ∈ *indets* (*punit.monom-mult c t p*)
proof
assume *x* ∈ *keys t*
from *assms*(*2*) **have** *keys p* ≠ {} **by** *simp*
then obtain *s* **where** *s* ∈ *keys p* **by** *blast*
hence *t* + *s* ∈ (+) *t* ‘ *keys p* **by** *fastforce*
also from *assms*(*1*) **have** ... = *keys* (*punit.monom-mult c t p*) **by** (*simp add*: *punit.keys-monom-mult*)
finally have *t* + *s* ∈ *keys* (*punit.monom-mult c t p*) **.**

show *?thesis*
proof (*rule in-indetsI*)
from ‹$x \in$ *keys t*› **show** $x \in$ *keys* $(t + s)$ **by** (*simp add*: *keys-plus-ninv-comm-monoid-add*)
qed *fact*
next
assume $x \in$ *indets p*
then obtain *s* **where** $s \in$ *keys p* **and** $x \in$ *keys s* **by** (*rule in-indetsE*)
from *this*(*1*) **have** $t + s \in (+)$ *t* ‘ *keys p* **by** *fastforce*
also from *assms*(*1*) **have** ... = *keys* (*punit.monom-mult c t p*) **by** (*simp add*: *punit.keys-monom-mult*)
finally have $t + s \in$ *keys* (*punit.monom-mult c t p*) **.**
show *?thesis*
proof (*rule in-indetsI*)
from ‹$x \in$ *keys s*› **show** $x \in$ *keys* $(t + s)$ **by** (*simp add*: *keys-plus-ninv-comm-monoid-add*)
qed *fact*
qed
qed

lemma *indets-sum-subset*: *indets* (*sum f A*) $\subseteq$ ($\bigcup a \in A$. *indets* (*f a*))
proof (*cases finite A*)
case *True*
thus *?thesis*
proof (*induct A*)
case *empty*
show *?case* **by** *simp*
next
case (*insert a A*)
have *indets* (*sum f* (*insert a A*)) $\subseteq$ *indets* (*f a*) $\cup$ *indets* (*sum f A*)
by (*simp only*: *comm-monoid-add-class.sum.insert*[*OF insert*(*1*) *insert*(*2*)] *indets-plus-subset*)
also have ... $\subseteq$ *indets* (*f a*) $\cup$ ($\bigcup a \in A$. *indets* (*f a*)) **using** *insert*(*3*) **by** *blast*
also have ... = ($\bigcup a \in$ *insert a A*. *indets* (*f a*)) **by** *simp*
finally show *?case* **.**
qed
next
case *False*
thus *?thesis* **by** *simp*
qed

lemma *indets-prod-subset*:
indets (*prod* (*f*::- ⇒ ((- $\Rightarrow_0$ -::*cancel-comm-monoid-add*) $\Rightarrow_0$ -)) *A*) $\subseteq$ ($\bigcup a \in A$. *indets* (*f a*))
proof (*cases finite A*)
case *True*
thus *?thesis*
proof (*induct A*)
case *empty*
show *?case* **by** *simp*
next

case (*insert a A*)
have *indets* (*prod f* (*insert a A*)) ⊆ *indets* (*f a*) ∪ *indets* (*prod f A*)
by (*simp only*: *comm-monoid-mult-class.prod.insert*[*OF insert*(*1*) *insert*(*2*)] *indets-times-subset*)
also have ... ⊆ *indets* (*f a*) ∪ (⋃ *a*∈*A*. *indets* (*f a*)) **using** *insert*(*3*) **by** *blast*
also have ... = (⋃ *a*∈*insert a A*. *indets* (*f a*)) **by** *simp*
finally show *?case* **.**
qed
next
case *False*
thus *?thesis* **by** *simp*
qed

lemma *indets-power-subset*: *indets* (*p* ^ *n*) ⊆ *indets* (*p*::(*'x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'b*::*comm-semiring-1*)
proof −
have *p* ^ *n* = (∏ *i*=*0*..<*n*. *p*) **by** *simp*
also have *indets* ... ⊆ (⋃ *i*∈{*0*..<*n*}. *indets p*) **by** (*fact indets-prod-subset*)
also have ... ⊆ *indets p* **by** *simp*
finally show *?thesis* **.**
qed

lemma *indets-empty-iff-poly-deg-zero*: *indets p* = {} ⟷ *poly-deg p* = *0*
proof
assume *indets p* = {}
hence *monomial* (*lookup p 0*) *0* = *p* **by** (*rule indets-empty-imp-monomial*)
moreover have *poly-deg* (*monomial* (*lookup p 0*) *0*) = *0* **by** *simp*
ultimately show *poly-deg p* = *0* **by** *metis*
next
assume *poly-deg p* = *0*
hence *monomial* (*lookup p 0*) *0* = *p* **by** (*rule poly-deg-zero-imp-monomial*)
moreover have *indets* (*monomial* (*lookup p 0*) *0*) = {} **by** *simp*
ultimately show *indets p* = {} **by** *metis*
qed

17.2.2 *PPs*

lemma *PPsI*: *keys t* ⊆ *X* ⟹ *t* ∈ .[*X*]
by (*simp add*: *PPs-def*)

lemma *PPsD*: *t* ∈ .[*X*] ⟹ *keys t* ⊆ *X*
by (*simp add*: *PPs-def*)

lemma *PPs-empty* [*simp*]: .[{}] = {*0*}
by (*simp add*: *PPs-def*)

lemma *PPs-UNIV* [*simp*]: .[*UNIV*] = *UNIV*
by (*simp add*: *PPs-def*)

lemma *PPs-singleton*: .[{*x*}] = *range* (*Poly-Mapping.single x*)

proof (*rule set-eqI*)
  fix *t*
  show $t \in .[\{x\}] \longleftrightarrow t \in$ *range* (*Poly-Mapping.single x*)
  proof
    assume $t \in .[\{x\}]$
    hence *keys* $t \subseteq \{x\}$ **by** (*rule PPsD*)
    hence *Poly-Mapping.single x* (*lookup t x*) = *t* **by** (*rule keys-subset-singleton-imp-monomial*)
    from *this*[*symmetric*] *UNIV-I* **show** $t \in$ *range* (*Poly-Mapping.single x*) **..**
  next
    assume $t \in$ *range* (*Poly-Mapping.single x*)
    then obtain *e* **where** *t* = *Poly-Mapping.single x e* **..**
    thus $t \in .[\{x\}]$ **by** (*simp add*: *PPs-def*)
  qed
qed

lemma *zero-in-PPs*: $0 \in .[X]$
  by (*simp add*: *PPs-def*)

lemma *PPs-mono*: $X \subseteq Y \Longrightarrow .[X] \subseteq .[Y]$
  by (*auto simp*: *PPs-def*)

lemma *PPs-closed-single*:
  assumes $x \in X$
  shows *Poly-Mapping.single x e* $\in .[X]$
proof (*rule PPsI*)
  have *keys* (*Poly-Mapping.single x e*) $\subseteq \{x\}$ **by** *simp*
  also from *assms* **have** ... $\subseteq X$ **by** *simp*
  finally show *keys* (*Poly-Mapping.single x e*) $\subseteq X$ **.**
qed

lemma *PPs-closed-plus*:
  assumes $s \in .[X]$ **and** $t \in .[X]$
  shows $s + t \in .[X]$
proof −
  have *keys* $(s + t) \subseteq$ *keys* $s \cup$ *keys* t **by** (*fact Poly-Mapping.keys-add*)
  also from *assms* **have** ... $\subseteq X$ **by** (*simp add*: *PPs-def*)
  finally show *?thesis* **by** (*rule PPsI*)
qed

lemma *PPs-closed-minus*:
  assumes $s \in .[X]$
  shows $s - t \in .[X]$
proof −
  have *keys* $(s - t) \subseteq$ *keys* s **by** (*metis lookup-minus lookup-not-eq-zero-eq-in-keys subsetI zero-diff*)
  also from *assms* **have** ... $\subseteq X$ **by** (*rule PPsD*)
  finally show *?thesis* **by** (*rule PPsI*)
qed

lemma *PPs-closed-adds*:
  assumes $s \in .[X]$ **and** *t adds s*
  shows $t \in .[X]$
proof −
  from *assms*(*2*) **have** $s - (s - t) = t$ **by** (*metis add-minus-2 adds-minus*)
  moreover from *assms*(*1*) **have** $s - (s - t) \in .[X]$ **by** (*rule PPs-closed-minus*)
  ultimately show *?thesis* **by** *simp*
qed

lemma *PPs-closed-gcs*:
  assumes $s \in .[X]$
  shows *gcs s t* $\in .[X]$
  using *assms gcs-adds* **by** (*rule PPs-closed-adds*)

lemma *PPs-closed-lcs*:
  assumes $s \in .[X]$ **and** $t \in .[X]$
  shows *lcs s t* $\in .[X]$
proof −
  from *assms* **have** $s + t \in .[X]$ **by** (*rule PPs-closed-plus*)
  hence $(s + t) -$ *gcs s t* $\in .[X]$ **by** (*rule PPs-closed-minus*)
  thus *?thesis* **by** (*simp add*: *gcs-plus-lcs*[*of s t*, *symmetric*])
qed

lemma *PPs-closed-except'*: $t \in .[X] \Longrightarrow$ *except t Y* $\in .[X - Y]$
  by (*auto simp*: *keys-except PPs-def*)

lemma *PPs-closed-except*: $t \in .[X] \Longrightarrow$ *except t Y* $\in .[X]$
  by (*auto simp*: *keys-except PPs-def*)

lemma *PPs-UnI*:
  assumes $tx \in .[X]$ **and** $ty \in .[Y]$ **and** $t = tx + ty$
  shows $t \in .[X \cup Y]$
proof −
  from *assms*(*1*) **have** $tx \in .[X \cup Y]$ **by** *rule* (*simp add*: *PPs-mono*)
  moreover from *assms*(*2*) **have** $ty \in .[X \cup Y]$ **by** *rule* (*simp add*: *PPs-mono*)
  ultimately show *?thesis* **unfolding** *assms*(*3*) **by** (*rule PPs-closed-plus*)
qed

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 −
  from *assms* **have** *keys* $t \subseteq X \cup Y$ **by** (*rule PPsD*)
  define *tx* **where** *tx* = *except t* $(- X)$
  have *keys* $tx \subseteq X$ **by** (*simp add*: *tx-def keys-except*)
  hence $tx \in .[X]$ **by** (*simp add*: *PPs-def*)
  have *tx adds t* **by** (*simp add*: *tx-def adds-poly-mappingI le-fun-def lookup-except*)
  from *adds-minus*[*OF this*] **have** $t = tx + (t - tx)$ **by** (*simp only*: *ac-simps*)
  have $t - tx \in .[Y]$

proof (*rule PPsI*, *rule*)
fix *x*
assume $x \in$ *keys* $(t - tx)$
also have ... $\subseteq$ *keys* $t \cup$ *keys* *tx* **by** (*rule keys-minus*)
also from ‹*keys* $t \subseteq X \cup Y$› ‹*keys* $tx \subseteq X$› **have** ... $\subseteq X \cup Y$ **by** *blast*
finally show $x \in Y$
proof
assume $x \in X$
hence $x \notin$ *keys* $(t - tx)$ **by** (*simp add*: *tx-def lookup-except lookup-minus in-keys-iff*)
thus *?thesis* **using** ‹$x \in$ *keys* $(t - tx)$› **..**
qed
qed
with ‹$tx \in .[X]$› **show** *?thesis* **using** ‹$t = tx + (t - tx)$› **..**
qed

lemma *PPs-Un*: $.[X \cup Y] = (\bigcup t{\in}.[X].\ (+)\ t\ `\ .[Y])$ (**is** $?A = ?B$)
proof (*rule set-eqI*)
fix *t*
show $t \in ?A \longleftrightarrow t \in ?B$
proof
assume $t \in ?A$
then obtain *tx ty* **where** $tx \in .[X]$ **and** $ty \in .[Y]$ **and** $t = tx + ty$ **by** (*rule PPs-UnE*)
from *this*(*2*) **have** $t \in (+)\ tx\ `\ .[Y]$ **unfolding** ‹$t = tx + ty$› **by** (*rule imageI*)
with ‹$tx \in .[X]$› **show** $t \in ?B$ **..**
next
assume $t \in ?B$
then obtain *tx* **where** $tx \in .[X]$ **and** $t \in (+)\ tx\ `\ .[Y]$ **..**
from *this*(*2*) **obtain** *ty* **where** $ty \in .[Y]$ **and** $t = tx + ty$ **..**
with ‹$tx \in .[X]$› **show** $t \in ?A$ **by** (*rule PPs-UnI*)
qed
qed

corollary *PPs-insert*: .[*insert x X*] $= (\bigcup e.\ (+)$ (*Poly-Mapping.single x e*) $`\ .[X])$
proof −
have .[*insert x X*] $= .[\{x\} \cup X]$ **by** *simp*
also have ... $= (\bigcup t{\in}.[\{x\}].\ (+)\ t\ `\ .[X])$ **by** (*fact PPs-Un*)
also have ... $= (\bigcup e.\ (+)$ (*Poly-Mapping.single x e*) $`\ .[X])$ **by** (*simp add*: *PPs-singleton*)
finally show *?thesis* **.**
qed

corollary *PPs-insertI*:
assumes $tx \in .[X]$ **and** $t =$ *Poly-Mapping.single x e* $+ tx$
shows $t \in$.[*insert x X*]
proof −
from *assms*(*1*) **have** $t \in (+)$ (*Poly-Mapping.single x e*) $`\ .[X]$ **unfolding** *assms*(*2*) **by** (*rule imageI*)

with *UNIV-I* **show** *?thesis* **unfolding** *PPs-insert* **by** (*rule UN-I*)
qed

corollary *PPs-insertE*:
assumes $t \in .[insert\ x\ X]$
obtains *e tx* **where** $tx \in .[X]$ **and** $t = Poly\text{-}Mapping.single\ x\ e + tx$
proof −
from *assms* **obtain** *e* **where** $t \in (+)\ (Poly\text{-}Mapping.single\ x\ e)$ ` $.[X]$ **unfolding** *PPs-insert* **..**
then obtain *tx* **where** $tx \in .[X]$ **and** $t = Poly\text{-}Mapping.single\ x\ e + tx$ **..**
thus *?thesis* **..**
qed

lemma *PPs-Int*: $.[X \cap Y] = .[X] \cap .[Y]$
by (*auto simp*: *PPs-def*)

lemma *PPs-INT*: $.[\bigcap X] = \bigcap (PPs$ ` $X)$
by (*auto simp*: *PPs-def*)

17.2.3 *Polys*

lemma *Polys-alt*: $P[X] = \{p.\ indets\ p \subseteq X\}$
by (*auto simp*: *Polys-def PPs-def indets-def*)

lemma *PolysI*: $keys\ p \subseteq .[X] \Longrightarrow p \in P[X]$
by (*simp add*: *Polys-def*)

lemma *PolysI-alt*: $indets\ p \subseteq X \Longrightarrow p \in P[X]$
by (*simp add*: *Polys-alt*)

lemma *PolysD*:
assumes $p \in P[X]$
shows $keys\ p \subseteq .[X]$ **and** $indets\ p \subseteq X$
using *assms* **by** (*simp add*: *Polys-def*, *simp add*: *Polys-alt*)

lemma *Polys-empty*: $P[\{\}] = ((range\ (Poly\text{-}Mapping.single\ 0))::(('x \Rightarrow_0 nat) \Rightarrow_0 'b::zero)\ set)$
proof (*rule set-eqI*)
fix $p :: ('x \Rightarrow_0 nat) \Rightarrow_0 'b::zero$
show $p \in P[\{\}] \longleftrightarrow p \in range\ (Poly\text{-}Mapping.single\ 0)$
proof
assume $p \in P[\{\}]$
hence $keys\ p \subseteq .[\{\}]$ **by** (*rule PolysD*)
also have $... = \{0\}$ **by** *simp*
finally have $keys\ p \subseteq \{0\}$ **.**
hence *Poly-Mapping.single 0* (*lookup p 0*) = *p* **by** (*rule keys-subset-singleton-imp-monomial*)
from *this*[*symmetric*] *UNIV-I* **show** $p \in range\ (Poly\text{-}Mapping.single\ 0)$ **..**
next
assume $p \in range\ (Poly\text{-}Mapping.single\ 0)$

then obtain *c* **where** *p* = *monomial c 0* **..**
thus *p* ∈ *P*[{}] **by** (*simp add*: *Polys-def*)
qed
qed

lemma *Polys-UNIV* [*simp*]: *P*[*UNIV*] = *UNIV*
by (*simp add*: *Polys-def*)

lemma *zero-in-Polys*: *0* ∈ *P*[*X*]
by (*simp add*: *Polys-def*)

lemma *one-in-Polys*: *1* ∈ *P*[*X*]
by (*simp add*: *Polys-def zero-in-PPs*)

lemma *Polys-mono*: *X* ⊆ *Y* ⟹ *P*[*X*] ⊆ *P*[*Y*]
by (*auto simp*: *Polys-alt*)

lemma *Polys-closed-monomial*: *t* ∈ .[*X*] ⟹ *monomial c t* ∈ *P*[*X*]
using *indets-monomial-subset*[**where** *c*=*c* **and** *t*=*t*] **by** (*auto simp*: *Polys-alt PPs-def*)

lemma *Polys-closed-plus*: *p* ∈ *P*[*X*] ⟹ *q* ∈ *P*[*X*] ⟹ *p* + *q* ∈ *P*[*X*]
using *indets-plus-subset*[*of p q*] **by** (*auto simp*: *Polys-alt PPs-def*)

lemma *Polys-closed-uminus*: *p* ∈ *P*[*X*] ⟹ −*p* ∈ *P*[*X*]
by (*simp add*: *Polys-def keys-uminus*)

lemma *Polys-closed-minus*: *p* ∈ *P*[*X*] ⟹ *q* ∈ *P*[*X*] ⟹ *p* − *q* ∈ *P*[*X*]
using *indets-minus-subset*[*of p q*] **by** (*auto simp*: *Polys-alt PPs-def*)

lemma *Polys-closed-monom-mult*: *t* ∈ .[*X*] ⟹ *p* ∈ *P*[*X*] ⟹ *punit.monom-mult c t p* ∈ *P*[*X*]
using *indets-monom-mult-subset*[*of c t p*] **by** (*auto simp*: *Polys-alt PPs-def*)

corollary *Polys-closed-map-scale*: *p* ∈ *P*[*X*] ⟹ (*c*::-::*semiring-0*) · *p* ∈ *P*[*X*]
unfolding *punit.map-scale-eq-monom-mult* **using** *zero-in-PPs* **by** (*rule Polys-closed-monom-mult*)

lemma *Polys-closed-times*: *p* ∈ *P*[*X*] ⟹ *q* ∈ *P*[*X*] ⟹ *p* ∗ *q* ∈ *P*[*X*]
using *indets-times-subset*[*of p q*] **by** (*auto simp*: *Polys-alt PPs-def*)

lemma *Polys-closed-power*: *p* ∈ *P*[*X*] ⟹ *p* ^ *m* ∈ *P*[*X*]
by (*induct m*) (*auto intro*: *one-in-Polys Polys-closed-times*)

lemma *Polys-closed-sum*: (⋀*a*. *a* ∈ *A* ⟹ *f a* ∈ *P*[*X*]) ⟹ *sum f A* ∈ *P*[*X*]
by (*induct A rule*: *infinite-finite-induct*) (*auto intro*: *zero-in-Polys Polys-closed-plus*)

lemma *Polys-closed-prod*: (⋀*a*. *a* ∈ *A* ⟹ *f a* ∈ *P*[*X*]) ⟹ *prod f A* ∈ *P*[*X*]
by (*induct A rule*: *infinite-finite-induct*) (*auto intro*: *one-in-Polys Polys-closed-times*)

lemma *Polys-closed-sum-list*: $(\bigwedge x.\ x \in set\ xs \Longrightarrow x \in P[X]) \Longrightarrow sum\text{-}list\ xs \in P[X]$
by (*induct xs*) (*auto intro*: *zero-in-Polys Polys-closed-plus*)

lemma *Polys-closed-except*: $p \in P[X] \Longrightarrow except\ p\ T \in P[X]$
by (*auto intro*!: *PolysI simp*: *keys-except dest*!: *PolysD*(*1*))

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 −
define *qX* **where** *qX* = *except q* (− .[*X*])
define *qY* **where** *qY* = *except q* .[*X*]
have *q*: *q* = *qX* + *qY* **by** (*simp only*: *qX-def qY-def add.commute flip*: *except-decomp*)
have *qX* ∈ *P*[*X*] **by** (*rule PolysI*) (*simp add*: *qX-def keys-except*)
with *assms*(*2*) **have** *p* ∗ *qX* ∈ *P*[*X*] **by** (*rule Polys-closed-times*)
show *?thesis*
proof (*cases qY* = *0*)
case *True*
with ‹*qX* ∈ *P*[*X*]› **show** *?thesis* **by** (*simp add*: *q*)
next
case *False*
with *assms*(*3*) **have** *p* ∗ *qY* ≠ *0* **by** *simp*
hence *keys* (*p* ∗ *qY*) ≠ {} **by** *simp*
then obtain *t* **where** *t* ∈ *keys* (*p* ∗ *qY*) **by** *blast*
then obtain *t1 t2* **where** *t2* ∈ *keys qY* **and** *t*: *t* = *t1* + *t2* **by** (*rule in-keys-timesE*)
have *t* ∉ .[*X*] **unfolding** *t*
proof
assume *t1* + *t2* ∈ .[*X*]
hence *t1* + *t2* − *t1* ∈ .[*X*] **by** (*rule PPs-closed-minus*)
hence *t2* ∈ .[*X*] **by** *simp*
with ‹*t2* ∈ *keys qY*› **show** *False* **by** (*simp add*: *qY-def keys-except*)
qed
have *t* ∉ *keys* (*p* ∗ *qX*)
proof
assume *t* ∈ *keys* (*p* ∗ *qX*)
also from ‹*p* ∗ *qX* ∈ *P*[*X*]› **have** . . . ⊆ .[*X*] **by** (*rule PolysD*)
finally have *t* ∈ .[*X*] **.**
with ‹*t* ∉ .[*X*]› **show** *False* **..**
qed
with ‹*t* ∈ *keys* (*p* ∗ *qY*)› **have** *t* ∈ *keys* (*p* ∗ *qX* + *p* ∗ *qY*) **by** (*rule in-keys-plusI2*)
also have . . . = *keys* (*p* ∗ *q*) **by** (*simp only*: *q algebra-simps*)
finally have *p* ∗ *q* ∉ *P*[*X*] **using** ‹*t* ∉ .[*X*]› **by** (*auto simp*: *Polys-def*)
thus *?thesis* **using** *assms*(*1*) **..**
qed

qed

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 $P\ p$
  using *assms*(*1*)
proof (*induct p rule*: *poly-mapping-plus-induct*)
  case *1*
  show *?case* **by** (*fact assms*(*2*))
next
  case *step*: (*2 p c t*)
  from *step.hyps*(*1*) **have** *1*: $keys\ (monomial\ c\ t) = \{t\}$ **by** *simp*
  also from *step.hyps*(*2*) **have** $\ldots \cap keys\ p = \{\}$ **by** *simp*
  finally have $keys\ (monomial\ c\ t + p) = keys\ (monomial\ c\ t) \cup keys\ p$ **by** (*rule keys-add*[*symmetric*])
  hence $keys\ (monomial\ c\ t + p) = insert\ t\ (keys\ p)$ **by** (*simp only*: *1 flip*: *insert-is-Un*)
  moreover from *step.prems*(*1*) **have** $keys\ (monomial\ c\ t + p) \subseteq .[X]$ **by** (*rule PolysD*)
  ultimately have $t \in .[X]$ **and** $keys\ p \subseteq .[X]$ **by** *blast+*
  from *this*(*2*) **have** $p \in P[X]$ **by** (*rule PolysI*)
  hence $P\ p$ **by** (*rule step.hyps*)
  with ‹$t \in .[X]$› ‹$p \in P[X]$› *step.hyps*(*1*, *2*) **show** *?case* **by** (*rule assms*(*3*))
qed

lemma *Polys-Int*: $P[X \cap Y] = P[X] \cap P[Y]$
  by (*auto simp*: *Polys-def PPs-Int*)

lemma *Polys-INT*: $P[\bigcap X] = \bigcap (Polys\ `\ X)$
  by (*auto simp*: *Polys-def PPs-INT*)

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* :: $('x \Rightarrow (('y \Rightarrow_0 nat) \Rightarrow_0 'a)) \Rightarrow ('x \Rightarrow_0 nat) \Rightarrow (('y \Rightarrow_0 nat) \Rightarrow_0 'a::comm\text{-}semiring\text{-}1)$
  where $subst\text{-}pp\ f\ t = (\prod x{\in}keys\ t.\ (f\ x)\ \hat{}\ (lookup\ t\ x))$

definition *poly-subst* :: $('x \Rightarrow (('y \Rightarrow_0 nat) \Rightarrow_0 'a)) \Rightarrow (('x \Rightarrow_0 nat) \Rightarrow_0 'a) \Rightarrow (('y \Rightarrow_0 nat) \Rightarrow_0 'a::comm\text{-}semiring\text{-}1)$
  where $poly\text{-}subst\ f\ p = (\sum t{\in}keys\ p.\ punit.monom\text{-}mult\ (lookup\ p\ t)\ 0\ (subst\text{-}pp\ f\ t))$

lemma *subst-pp-alt*: $subst\text{-}pp\ f\ t = (\prod x.\ (f\ x)\ \hat{}\ (lookup\ t\ x))$
proof −

from *finite-keys* **have** *subst-pp f t* = ($\prod$ *x. if x* ∈ *keys t then* (*f x*) ^ (*lookup t x*) *else 1*)
unfolding *subst-pp-def* **by** (*rule Prod-any.conditionalize*)
also have ... = ($\prod$ *x.* (*f x*) ^ (*lookup t x*)) **by** (*rule Prod-any.cong*) (*simp add*: *in-keys-iff*)
finally show *?thesis* **.**
qed

lemma *subst-pp-zero* [*simp*]: *subst-pp f 0* = *1*
by (*simp add*: *subst-pp-def*)

lemma *subst-pp-trivial-not-zero*:
assumes *t* ≠ *0*
shows *subst-pp* (λ-. *0*) *t* = (*0*::(- ⇒$_0$ ′*b*::*comm-semiring-1*))
unfolding *subst-pp-def* **using** *finite-keys*
proof (*rule prod-zero*)
from *assms* **have** *keys t* ≠ {} **by** *simp*
then obtain *x* **where** *x* ∈ *keys t* **by** *blast*
thus ∃ *x*∈*keys t. 0* ^ *lookup t x* = (*0*::(- ⇒$_0$ ′*b*))
proof
from ‹*x* ∈ *keys t*› **have** *0* < *lookup t x* **by** (*simp add*: *in-keys-iff*)
thus *0* ^ *lookup t x* = (*0*::(- ⇒$_0$ ′*b*)) **by** (*rule Power.semiring-1-class.zero-power*)
qed
qed

lemma *subst-pp-single*: *subst-pp f* (*Poly-Mapping.single x e*) = (*f x*) ^ *e*
by (*simp add*: *subst-pp-def*)

corollary *subst-pp-trivial*: *subst-pp* (λ-. *0*) *t* = (*if t* = *0 then 1 else 0*)
by (*simp split*: *if-split add*: *subst-pp-trivial-not-zero*)

lemma *power-lookup-not-one-subset-keys*: {*x. f x* ^ (*lookup t x*) ≠ *1*} ⊆ *keys t*
proof (*rule*, *simp*)
fix *x*
assume *f x* ^ (*lookup t x*) ≠ *1*
thus *x* ∈ *keys t* **unfolding** *in-keys-iff* **by** (*metis power-0*)
qed

corollary *finite-power-lookup-not-one*: *finite* {*x. f x* ^ (*lookup t x*) ≠ *1*}
by (*rule finite-subset*, *fact power-lookup-not-one-subset-keys*, *fact finite-keys*)

lemma *subst-pp-plus*: *subst-pp f* (*s* + *t*) = *subst-pp f s* ∗ *subst-pp f t*
by (*simp add*: *subst-pp-alt lookup-add power-add*, *rule Prod-any.distrib*, (*fact finite-power-lookup-not-one*)+)

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 −

have *subst-pp f t* = ($\prod$ *x*∈*keys t*. *monomial 1* (*Poly-Mapping.single x* (*lookup t x*)))
proof (*simp only*: *subst-pp-def*, *rule prod.cong*, *fact refl*)
fix *x*
assume *x* ∈ *keys t*
thus *f x* ^ *lookup t x* = *monomial 1* (*Poly-Mapping.single x* (*lookup t x*))
by (*simp add*: *assms monomial-single-power*)
qed
also have ... = *monomial 1 t*
by (*simp add*: *punit.monomial-prod-sum*[*symmetric*] *poly-mapping-sum-monomials*)
finally show *?thesis* **.**
qed

lemma *in-indets-subst-ppE*:
assumes *x* ∈ *indets* (*subst-pp f t*)
obtains *y* **where** *y* ∈ *keys t* **and** *x* ∈ *indets* (*f y*)
proof −
note *assms*
also have *indets* (*subst-pp f t*) ⊆ ($\bigcup$ *y*∈*keys t*. *indets* ((*f y*) ^ (*lookup t y*)))
unfolding *subst-pp-def*
by (*rule indets-prod-subset*)
finally obtain *y* **where** *y* ∈ *keys t* **and** *x* ∈ *indets* ((*f y*) ^ (*lookup t y*)) **..**
note *this*(*2*)
also have *indets* ((*f y*) ^ (*lookup t y*)) ⊆ *indets* (*f y*) **by** (*rule indets-power-subset*)
finally have *x* ∈ *indets* (*f y*) **.**
with ‹*y* ∈ *keys t*› **show** *?thesis* **..**
qed

lemma *subst-pp-by-monomials*:
assumes $\bigwedge$*y*. *y* ∈ *keys t* ⟹ *f y* = *monomial* (*c y*) (*s y*)
shows *subst-pp f t* = *monomial* ($\prod$ *y*∈*keys t*. (*c y*) ^ *lookup t y*) ($\sum$ *y*∈*keys t*. *lookup t y* · *s y*)
by (*simp add*: *subst-pp-def assms monomial-power-map-scale punit.monomial-prod-sum*)

lemma *poly-deg-subst-pp-eq-zeroI*:
assumes $\bigwedge$*x*. *x* ∈ *keys t* ⟹ *poly-deg* (*f x*) = *0*
shows *poly-deg* (*subst-pp f t*) = *0*
proof −
have *poly-deg* (*subst-pp f t*) ≤ ($\sum$ *x*∈*keys t*. *poly-deg* ((*f x*) ^ (*lookup t x*)))
unfolding *subst-pp-def* **by** (*fact poly-deg-prod-le*)
also have ... = *0*
proof (*rule sum.neutral*, *rule*)
fix *x*
assume *x* ∈ *keys t*
hence *poly-deg* (*f x*) = *0* **by** (*rule assms*)
have *f x* ^ *lookup t x* = ($\prod$ *i*=*0*..<*lookup t x*. *f x*) **by** *simp*
also have *poly-deg* ... ≤ ($\sum$ *i*=*0*..<*lookup t x*. *poly-deg* (*f x*)) **by** (*rule poly-deg-prod-le*)
also have ... = *0* **by** (*simp add*: ‹*poly-deg* (*f x*) = *0*›)
finally show *poly-deg* (*f x* ^ *lookup t x*) = *0* **by** *simp*

qed
finally show *?thesis* **by** *simp*
qed

lemma *poly-deg-subst-pp-le*:
assumes $\bigwedge x.\ x \in keys\ t \Longrightarrow poly\text{-}deg\ (f\ x) \le 1$
shows $poly\text{-}deg\ (subst\text{-}pp\ f\ t) \le deg\text{-}pm\ t$
proof −
have $poly\text{-}deg\ (subst\text{-}pp\ f\ t) \le (\sum x \in keys\ t.\ poly\text{-}deg\ ((f\ x)\ \widehat{}\ (lookup\ t\ x)))$
unfolding *subst-pp-def* **by** (*fact poly-deg-prod-le*)
also have ... $\le (\sum x \in keys\ t.\ lookup\ t\ x)$
proof (*rule sum-mono*)
fix *x*
assume $x \in keys\ t$
hence $poly\text{-}deg\ (f\ x) \le 1$ **by** (*rule assms*)
have $f\ x\ \widehat{}\ lookup\ t\ x = (\prod i=0..<lookup\ t\ x.\ f\ x)$ **by** *simp*
also have $poly\text{-}deg$ ... $\le (\sum i=0..<lookup\ t\ x.\ poly\text{-}deg\ (f\ x))$ **by** (*rule poly-deg-prod-le*)
also from ‹$poly\text{-}deg\ (f\ x) \le 1$› **have** ... $\le (\sum i=0..<lookup\ t\ x.\ 1)$ **by** (*rule sum-mono*)
finally show $poly\text{-}deg\ (f\ x\ \widehat{}\ lookup\ t\ x) \le lookup\ t\ x$ **by** *simp*
qed
also have ... $= deg\text{-}pm\ t$ **by** (*rule deg-pm-superset*[*symmetric*], *fact subset-refl*, *fact finite-keys*)
finally show *?thesis* **by** *simp*
qed

lemma *poly-subst-alt*: $poly\text{-}subst\ f\ p = (\sum t.\ punit.monom\text{-}mult\ (lookup\ p\ t)\ 0\ (subst\text{-}pp\ f\ t))$
proof −
from *finite-keys* **have** $poly\text{-}subst\ f\ p = (\sum t.\ if\ t \in keys\ p\ then\ punit.monom\text{-}mult\ (lookup\ p\ t)\ 0\ (subst\text{-}pp\ f\ t)\ else\ 0)$
unfolding *poly-subst-def* **by** (*rule Sum-any.conditionalize*)
also have $\ldots = (\sum t.\ punit.monom\text{-}mult\ (lookup\ p\ t)\ 0\ (subst\text{-}pp\ f\ t))$
by (*rule Sum-any.cong*) (*simp add*: *in-keys-iff*)
finally show *?thesis* **.**
qed

lemma *poly-subst-trivial* [*simp*]: $poly\text{-}subst\ (\lambda\text{-}.\ 0)\ p = monomial\ (lookup\ p\ 0)\ 0$
by (*simp add*: *poly-subst-def subst-pp-trivial if-distrib in-keys-iff cong*: *if-cong*)
(*metis mult.right-neutral times-monomial-left*)

lemma *poly-subst-zero* [*simp*]: $poly\text{-}subst\ f\ 0 = 0$
by (*simp add*: *poly-subst-def*)

lemma *monom-mult-lookup-not-zero-subset-keys*:
$\{t.\ punit.monom\text{-}mult\ (lookup\ p\ t)\ 0\ (subst\text{-}pp\ f\ t) \neq 0\} \subseteq keys\ p$
proof (*rule*, *simp*)
fix *t*
assume $punit.monom\text{-}mult\ (lookup\ p\ t)\ 0\ (subst\text{-}pp\ f\ t) \neq 0$

thus *t* ∈ *keys p* **unfolding** *in-keys-iff* **by** (*metis punit.monom-mult-zero-left*)
qed

corollary *finite-monom-mult-lookup-not-zero*:
finite {*t. punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*) ≠ *0*}
by (*rule finite-subset*, *fact monom-mult-lookup-not-zero-subset-keys*, *fact finite-keys*)

lemma *poly-subst-plus*: *poly-subst f* (*p* + *q*) = *poly-subst f p* + *poly-subst f q*
by (*simp add*: *poly-subst-alt lookup-add punit.monom-mult-dist-left*, *rule Sum-any.distrib*,
(*fact finite-monom-mult-lookup-not-zero*)+)

lemma *poly-subst-uminus*: *poly-subst f* (−*p*) = − *poly-subst f* (*p*::(*'x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'b*::*comm-ring-1*)
by (*simp add*: *poly-subst-def keys-uminus punit.monom-mult-uminus-left sum-negf*)

lemma *poly-subst-minus*:
poly-subst f (*p* − *q*) = *poly-subst f p* − *poly-subst f* (*q*::(*'x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'b*::*comm-ring-1*)
proof −
have *poly-subst f* (*p* + (−*q*)) = *poly-subst f p* + *poly-subst f* (−*q*) **by** (*fact poly-subst-plus*)
thus *?thesis* **by** (*simp add*: *poly-subst-uminus*)
qed

lemma *poly-subst-monomial*: *poly-subst f* (*monomial c t*) = *punit.monom-mult c 0* (*subst-pp f t*)
by (*simp add*: *poly-subst-def lookup-single*)

corollary *poly-subst-one* [*simp*]: *poly-subst f 1* = *1*
by (*simp add*: *single-one*[*symmetric*] *poly-subst-monomial punit.monom-mult-monomial del*: *single-one*)

lemma *poly-subst-times*: *poly-subst f* (*p* ∗ *q*) = *poly-subst f p* ∗ *poly-subst f q*
proof −
have *bij*: *bij* (λ(*l*, *n*, *m*). (*m*, *l*, *n*))
by (*auto intro*!: *bijI injI simp add*: *image-def*)
let *?P* = *keys p*
let *?Q* = *keys q*
let *?PQ* = {*s* + *t* | *s t. lookup p s* ≠ *0* ∧ *lookup q t* ≠ *0*}
have *fin-PQ*: *finite ?PQ*
by (*rule finite-not-eq-zero-sumI*, *simp-all*)
have *fin-1*: *finite* {*l. lookup p l* ∗ (∑ *qa. lookup q qa when t* = *l* + *qa*) ≠ *0*} **for** *t*
proof (*rule finite-subset*)
show {*l. lookup p l* ∗ (∑ *qa. lookup q qa when t* = *l* + *qa*) ≠ *0*} ⊆ *keys p*
by (*rule*, *auto simp*: *in-keys-iff*)
qed (*fact finite-keys*)
have *fin-2*: *finite* {*v.* (*lookup q v when t* = *u* + *v*) ≠ *0*} **for** *t u*
proof (*rule finite-subset*)
show {*v.* (*lookup q v when t* = *u* + *v*) ≠ *0*} ⊆ *keys q*

by (*rule*, *auto simp*: *in-keys-iff*)
qed (*fact finite-keys*)
have *fin-3*: *finite* {*v*. (*lookup p u* ∗ *lookup q v when t* = *u* + *v*) ≠ *0*} **for** *t u*
proof (*rule finite-subset*)
show {*v*. (*lookup p u* ∗ *lookup q v when t* = *u* + *v*) ≠ *0*} ⊆ *keys q*
by (*rule*, *auto simp add*: *in-keys-iff simp del*: *lookup-not-eq-zero-eq-in-keys*)
qed (*fact finite-keys*)
have (∑ *t*. *punit.monom-mult* (*lookup* (*p* ∗ *q*) *t*) *0* (*subst-pp f t*)) =
(∑ *t*. ∑ *u*. *punit.monom-mult* (*lookup p u* ∗ (∑ *v*. *lookup q v when t* = *u* + *v*)) *0* (*subst-pp f t*))
by (*simp add*: *times-poly-mapping.rep-eq prod-fun-def punit.monom-mult-Sum-any-left*[*OF fin-1*])
also have ... = (∑ *t*. ∑ *u*. ∑ *v*. (*punit.monom-mult* (*lookup p u* ∗ *lookup q v*) *0* (*subst-pp f t*)) *when t* = *u* + *v*)
by (*simp add*: *Sum-any-right-distrib*[*OF fin-2*] *punit.monom-mult-Sum-any-left*[*OF fin-3*] *mult-when punit.when-monom-mult*)
also have ... = (∑ *t*. (∑ (*u*, *v*). (*punit.monom-mult* (*lookup p u* ∗ *lookup q v*) *0* (*subst-pp f t*)) *when t* = *u* + *v*))
by (*subst* (*2*) *Sum-any.cartesian-product* [*of ?P* × *?Q*]) (*auto simp*: *in-keys-iff*)
also have ... = (∑ (*t*, *u*, *v*). *punit.monom-mult* (*lookup p u* ∗ *lookup q v*) *0* (*subst-pp f t*) *when t* = *u* + *v*)
apply (*subst Sum-any.cartesian-product* [*of ?PQ* × (*?P* × *?Q*)])
apply (*auto simp*: *fin-PQ in-keys-iff*)
apply (*metis monomial-0I mult-not-zero times-monomial-left*)
done
also have ... = (∑ (*u*, *v*, *t*). *punit.monom-mult* (*lookup p u* ∗ *lookup q v*) *0* (*subst-pp f t*) *when t* = *u* + *v*)
using *bij* **by** (*rule Sum-any.reindex-cong* [*of* λ(*u*, *v*, *t*). (*t*, *u*, *v*)]) (*simp add*: *fun-eq-iff*)
also have ... = (∑ (*u*, *v*). ∑ *t*. *punit.monom-mult* (*lookup p u* ∗ *lookup q v*) *0* (*subst-pp f t*) *when t* = *u* + *v*)
apply (*subst Sum-any.cartesian-product2* [*of* (*?P* × *?Q*) × *?PQ*])
apply (*auto simp*: *fin-PQ in-keys-iff*)
apply (*metis monomial-0I mult-not-zero times-monomial-left*)
done
also have ... = (∑ (*u*, *v*). *punit.monom-mult* (*lookup p u* ∗ *lookup q v*) *0* (*subst-pp f u* ∗ *subst-pp f v*))
by (*simp add*: *subst-pp-plus*)
also have ... = (∑ *u*. ∑ *v*. *punit.monom-mult* (*lookup p u* ∗ *lookup q v*) *0* (*subst-pp f u* ∗ *subst-pp f v*))
by (*subst Sum-any.cartesian-product* [*of ?P* × *?Q*]) (*auto simp*: *in-keys-iff*)
also have ... = (∑ *u*. ∑ *v*. (*punit.monom-mult* (*lookup p u*) *0* (*subst-pp f u*)) ∗ (*punit.monom-mult* (*lookup q v*) *0* (*subst-pp f v*)))
by (*simp add*: *times-monomial-left*[*symmetric*] *ac-simps mult-single*)
also have ... = (∑ *t*. *punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*)) ∗
(∑ *t*. *punit.monom-mult* (*lookup q t*) *0* (*subst-pp f t*))
by (*rule Sum-any-product* [*symmetric*], (*fact finite-monom-mult-lookup-not-zero*)+)
finally show *?thesis* **by** (*simp add*: *poly-subst-alt*)
qed

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*)
by (*simp only*: *times-monomial-left*[*symmetric*] *poly-subst-times poly-subst-monomial mult.assoc*)

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*
by (*simp only*: *times-monomial-left*[*symmetric*] *poly-subst-times poly-subst-monomial*)

lemma *poly-subst-sum*: *poly-subst f* (*sum p A*) = (∑ *a*∈*A*. *poly-subst f* (*p a*))
by (*rule fun-sum-commute*, *simp-all add*: *poly-subst-plus*)

lemma *poly-subst-prod*: *poly-subst f* (*prod p A*) = (∏ *a*∈*A*. *poly-subst f* (*p a*))
by (*rule fun-prod-commute*, *simp-all add*: *poly-subst-times*)

lemma *poly-subst-power*: *poly-subst f* (*p* ^ *n*) = (*poly-subst f p*) ^ *n*
by (*induct n*, *simp-all add*: *poly-subst-times*)

lemma *poly-subst-subst-pp*: *poly-subst f* (*subst-pp g t*) = *subst-pp* (*λx*. *poly-subst f* (*g x*)) *t*
by (*simp only*: *subst-pp-def poly-subst-prod poly-subst-power*)

lemma *poly-subst-poly-subst*: *poly-subst f* (*poly-subst g p*) = *poly-subst* (*λx*. *poly-subst f* (*g x*)) *p*
proof −
have *poly-subst f* (*poly-subst g p*) =
poly-subst f (∑ *t*∈*keys p*. *punit.monom-mult* (*lookup p t*) *0* (*subst-pp g t*))
by (*simp only*: *poly-subst-def*)
also have ... = (∑ *t*∈*keys p*. *punit.monom-mult* (*lookup p t*) *0* (*subst-pp* (*λx*. *poly-subst f* (*g x*)) *t*))
by (*simp add*: *poly-subst-sum poly-subst-monom-mult poly-subst-subst-pp*)
also have ... = *poly-subst* (*λx*. *poly-subst f* (*g x*)) *p* **by** (*simp only*: *poly-subst-def*)
finally show *?thesis* **.**
qed

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 −
have *poly-subst f p* = (∑ *t*∈*keys p*. *monomial* (*lookup p t*) *t*)
proof (*simp only*: *poly-subst-def*, *rule sum.cong*, *fact refl*)
fix *t*
assume *t* ∈ *keys p*
have *eq*: *subst-pp f t* = *monomial 1 t*
by (*rule subst-pp-id*, *rule assms*, *erule in-indetsI*, *fact* ‹*t* ∈ *keys p*›)
show *punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*) = *monomial* (*lookup p t*)

t
 by (*simp add*: *eq punit.monom-mult-monomial*)
 qed
 also have ... = *p* **by** (*simp only*: *poly-mapping-sum-monomials*)
 finally show *?thesis* **.**
qed

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 −
 note *assms*
 also have *keys* (*poly-subst f p*) ⊆ (⋃*t*∈*keys p*. *keys* (*punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*)))
 unfolding *poly-subst-def* **by** (*rule keys-sum-subset*)
 finally obtain *s* **where** *s* ∈ *keys p* **and** *t* ∈ *keys* (*punit.monom-mult* (*lookup p s*) *0* (*subst-pp f s*)) **..**
 note *this*(*2*)
 also have . . . ⊆ (+) *0* ' *keys* (*subst-pp f s*) **by** (*rule punit.keys-monom-mult-subset*[*simplified*])
 also have . . . = *keys* (*subst-pp f s*) **by** *simp*
 finally have *t* ∈ *keys* (*subst-pp f s*) **.**
 with ‹*s* ∈ *keys p*› **show** *?thesis* **..**
qed

lemma *in-indets-poly-substE*:
 assumes *x* ∈ *indets* (*poly-subst f p*)
 obtains *y* **where** *y* ∈ *indets p* **and** *x* ∈ *indets* (*f y*)
proof −
 note *assms*
 also have *indets* (*poly-subst f p*) ⊆ (⋃*t*∈*keys p*. *indets* (*punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*)))
 unfolding *poly-subst-def* **by** (*rule indets-sum-subset*)
 finally obtain *t* **where** *t* ∈ *keys p* **and** *x* ∈ *indets* (*punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*)) **..**
 note *this*(*2*)
 also have *indets* (*punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*)) ⊆ *keys* (*0*::('*a* $\Rightarrow_0$ *nat*)) ∪ *indets* (*subst-pp f t*)
 by (*rule indets-monom-mult-subset*)
 also have ... = *indets* (*subst-pp f t*) **by** *simp*
 finally obtain *y* **where** *y* ∈ *keys t* **and** *x* ∈ *indets* (*f y*) **by** (*rule in-indets-subst-ppE*)
 from *this*(*1*) ‹*t* ∈ *keys p*› **have** *y* ∈ *indets p* **by** (*rule in-indetsI*)
 from *this* ‹*x* ∈ *indets* (*f y*)› **show** *?thesis* **..**
qed

lemma *poly-deg-poly-subst-eq-zeroI*:
 assumes ⋀*x*. *x* ∈ *indets p* ⟹ *poly-deg* (*f x*) = *0*
 shows *poly-deg* (*poly-subst* (*f*::- ⇒ (('*y* $\Rightarrow_0$ -) $\Rightarrow_0$ -)) (*p*::('*x* $\Rightarrow_0$ -) $\Rightarrow_0$ '*b*::*comm-semiring-1*)) = *0*
proof (*cases p* = *0*)

case *True*
thus *?thesis* **by** *simp*
next
case *False*
have *poly-deg* (*poly-subst f p*) ≤ *Max* (*poly-deg* ‘ (λ*t*. *punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*)) ‘ *keys p*)
unfolding *poly-subst-def* **by** (*fact poly-deg-sum-le*)
also have ... ≤ *0*
proof (*rule Max.boundedI*)
show *finite* (*poly-deg* ‘ (λ*t*. *punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*)) ‘ *keys p*)
by (*simp add*: *finite-image-iff*)
next
from *False* **show** *poly-deg* ‘ (λ*t*. *punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*)) ‘ *keys p* ≠ {} **by** *simp*
next
fix *d*
assume *d* ∈ *poly-deg* ‘ (λ*t*. *punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*)) ‘ *keys p*
then obtain *t* **where** *t* ∈ *keys p* **and** *d*: *d* = *poly-deg* (*punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*))
by *fastforce*
have *d* ≤ *deg-pm* (*0*::′*y* $\Rightarrow_0$ *nat*) + *poly-deg* (*subst-pp f t*)
unfolding *d* **by** (*fact poly-deg-monom-mult-le*)
also have ... = *poly-deg* (*subst-pp f t*) **by** *simp*
also have ... = *0* **by** (*rule poly-deg-subst-pp-eq-zeroI*, *rule assms*, *erule in-indetsI*, *fact*)
finally show *d* ≤ *0* **.**
qed
finally show *?thesis* **by** *simp*
qed

lemma *poly-deg-poly-subst-le*:
assumes ⋀*x*. *x* ∈ *indets p* ⟹ *poly-deg* (*f x*) ≤ *1*
shows *poly-deg* (*poly-subst* (*f*::- ⇒ ((′*y* $\Rightarrow_0$ -) $\Rightarrow_0$ -)) (*p*::(′*x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ ′*b*::*comm-semiring-1*)) ≤ *poly-deg p*
proof (*cases p* = *0*)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
have *poly-deg* (*poly-subst f p*) ≤ *Max* (*poly-deg* ‘ (λ*t*. *punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*)) ‘ *keys p*)
unfolding *poly-subst-def* **by** (*fact poly-deg-sum-le*)
also have ... ≤ *poly-deg p*
proof (*rule Max.boundedI*)
show *finite* (*poly-deg* ‘ (λ*t*. *punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*)) ‘ *keys p*)
by (*simp add*: *finite-image-iff*)

next
from *False* **show** *poly-deg* ‘ (λ*t*. *punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*)) ‘ *keys p* ≠ {} **by** *simp*
next
fix *d*
assume *d* ∈ *poly-deg* ‘ (λ*t*. *punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*)) ‘ *keys p*
then obtain *t* **where** *t* ∈ *keys p* **and** *d*: *d* = *poly-deg* (*punit.monom-mult* (*lookup p t*) *0* (*subst-pp f t*))
by *fastforce*
have $d \leq$ *deg-pm* (*0*::′*y* $\Rightarrow_0$ *nat*) + *poly-deg* (*subst-pp f t*)
unfolding *d* **by** (*fact poly-deg-monom-mult-le*)
also have ... = *poly-deg* (*subst-pp f t*) **by** *simp*
also have ... ≤ *deg-pm t* **by** (*rule poly-deg-subst-pp-le*, *rule assms*, *erule in-indetsI*, *fact*)
also from ‹*t* ∈ *keys p*› **have** ... ≤ *poly-deg p* **by** (*rule poly-deg-max-keys*)
finally show *d* ≤ *poly-deg p* **.**
qed
finally show *?thesis* **by** *simp*
qed

lemma *subst-pp-cong*: *s* = *t* ⟹ (⋀*x*. *x* ∈ *keys t* ⟹ *f x* = *g x*) ⟹ *subst-pp f s* = *subst-pp g t*
by (*simp add*: *subst-pp-def*)

lemma *poly-subst-cong*:
assumes *p* = *q* **and** ⋀*x*. *x* ∈ *indets q* ⟹ *f x* = *g x*
shows *poly-subst f p* = *poly-subst g q*
proof (*simp add*: *poly-subst-def assms*(*1*), *rule sum.cong*)
fix *t*
assume *t* ∈ *keys q*
{
fix *x*
assume *x* ∈ *keys t*
with ‹*t* ∈ *keys q*› **have** *x* ∈ *indets q* **by** (*auto simp*: *indets-def*)
hence *f x* = *g x* **by** (*rule assms*(*2*))
}
thus *punit.monom-mult* (*lookup q t*) *0* (*subst-pp f t*) = *punit.monom-mult* (*lookup q t*) *0* (*subst-pp g t*)
by (*simp cong*: *subst-pp-cong*)
qed (*fact refl*)

lemma *Polys-homomorphismE*:
obtains *h* **where** ⋀*p q*. *h* (*p* + *q*) = *h p* + *h q* **and** ⋀*p q*. *h* (*p* ∗ *q*) = *h p* ∗ *h q*
and ⋀*p*::(′*x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ ′*a*::*comm-ring-1*. *h* (*h p*) = *h p* **and** *range h* = *P*[*X*]
proof −
let *?f* = λ*x*. *if x* ∈ *X then monomial* (*1*::′*a*) (*Poly-Mapping.single x 1*) *else 1*

have *1*: *poly-subst ?f p* = *p* **if** *p* ∈ *P*[*X*] **for** *p*

proof (*rule poly-subst-id*)
  fix *x*
  assume $x \in$ *indets p*
  also from *that* **have** $\ldots \subseteq X$ **by** (*rule PolysD*)
  finally show *?f x* = *monomial 1* (*Poly-Mapping.single x 1*) **by** *simp*
qed

have *2*: *poly-subst ?f p* $\in P[X]$ **for** *p*
proof (*intro PolysI-alt subsetI*)
  fix *x*
  assume $x \in$ *indets* (*poly-subst ?f p*)
  then obtain *y* **where** $x \in$ *indets* (*?f y*) **by** (*rule in-indets-poly-substE*)
  thus $x \in X$ **by** (*simp add*: *indets-monomial split*: *if-split-asm*)
qed

from *poly-subst-plus poly-subst-times* **show** *?thesis*
proof
  fix *p*
  from *2* **show** *poly-subst ?f* (*poly-subst ?f p*) = *poly-subst ?f p* **by** (*rule 1*)
next
  show *range* (*poly-subst ?f*) = $P[X]$
  proof (*intro set-eqI iffI*)
    fix *p* :: - $\Rightarrow_0$ *'a*
    assume $p \in P[X]$
    hence *p* = *poly-subst ?f p* **by** (*simp only*: *1*)
    thus $p \in$ *range* (*poly-subst ?f*) **by** (*rule image-eqI*) *simp*
  qed (*auto intro*: *2*)
qed
qed

lemma *in-idealE-Polys-finite*:
  assumes *finite B* **and** $B \subseteq P[X]$ **and** $p \in P[X]$ **and** (*p*::(*'x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'a*::*comm-ring-1*) $\in$ *ideal B*
  obtains *q* **where** $\bigwedge b.\ q\ b \in P[X]$ **and** $p = (\sum b{\in}B.\ q\ b * b)$
proof −
  obtain *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* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'a*. *h* (*h p*) = *h p* **and** *rng*[*symmetric*]: *range h* = $P[X]$
    by (*rule Polys-homomorphismE*) *blast*
  from *this*(*1*−*3*) *assms* **obtain** *q* **where** $\bigwedge b.\ q\ b \in P[X]$ **and** $p = (\sum b{\in}B.\ q\ b * b)$
    unfolding *rng* **by** (*rule in-idealE-homomorphism-finite*) *blast*
  thus *?thesis* **..**
qed

corollary *in-idealE-Polys*:
  assumes $B \subseteq P[X]$ **and** $p \in P[X]$ **and** $p \in$ *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 −
  from *assms*(*3*) **obtain** *A* **where** *finite A* **and** $A \subseteq B$ **and** $p \in$ *ideal A*
    by (*rule ideal.span-finite-subset*)
  from *this*(*2*) *assms*(*1*) **have** $A \subseteq P[X]$ **by** (*rule subset-trans*)
  with ‹*finite A*› **obtain** *q* **where** $\bigwedge b.\ q\ b \in P[X]$ **and** $p = (\sum b{\in}A.\ q\ b * b)$
    using *assms*(*2*) ‹$p \in$ *ideal A*› **by** (*rule in-idealE-Polys-finite*) *blast*
  with ‹*finite A*› ‹$A \subseteq B$› **show** *?thesis* **..**
qed

lemma *ideal-induct-Polys* [*consumes 3*, *case-names 0 plus*]:
  assumes $F \subseteq P[X]$ **and** $p \in P[X]$ **and** $p \in$ *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* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'a*::*comm-ring-1*)
proof −
  obtain *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* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'a*. *h* (*h p*) = *h p* **and** *rng*[*symmetric*]: *range h* =
$P[X]$
    by (*rule Polys-homomorphismE*) *blast*
  from *this*(*1*−*3*) *assms* **show** *?thesis*
    unfolding *rng* **by** (*rule ideal-induct-homomorphism*) *blast*
qed

lemma *image-poly-subst-ideal-subset*: *poly-subst g* ‘ *ideal F* ⊆ *ideal* (*poly-subst g* ‘
F)
proof (*intro subsetI*, *elim imageE*)
  fix *h f*
  assume *h*: *h* = *poly-subst g f*
  assume *f* ∈ *ideal F*
  thus *h* ∈ *ideal* (*poly-subst g* ‘ *F*) **unfolding** *h*
  proof (*induct f rule*: *ideal.span-induct-alt*)
    case *base*
    show *?case* **by** (*simp add*: *ideal.span-zero*)
  next
    case (*step c f h*)
    from *step.hyps*(*1*) **have** *poly-subst g f* ∈ *ideal* (*poly-subst g* ‘ *F*)
      by (*intro ideal.span-base imageI*)
    hence *poly-subst g c* ∗ *poly-subst g f* ∈ *ideal* (*poly-subst g* ‘ *F*) **by** (*rule*
ideal.span-scale)
    hence *poly-subst g c* ∗ *poly-subst g f* + *poly-subst g h* ∈ *ideal* (*poly-subst g* ‘ *F*)
      using *step.hyps*(*2*) **by** (*rule ideal.span-add*)
    thus *?case* **by** (*simp only*: *poly-subst-plus poly-subst-times*)
  qed
qed

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 −
have *eq*: ($\sum$ *v*∈*keys q*. *lookup q v when t* + *v* = *0*) = (*lookup q 0 when t* = *0*) **for** *t*
proof −
have ($\sum$ *v*∈*keys q*. *lookup q v when t* + *v* = *0*) = ($\sum$ *v*∈*keys q* ∩ {*0*}. *lookup q v when t* + *v* = *0*)
proof (*intro sum.mono-neutral-right ballI*)
fix *v*
assume *v* ∈ *keys q* − *keys q* ∩ {*0*}
hence *v* ≠ *0* **by** *blast*
hence *t* + *v* ≠ *0* **using** *plus-eq-zero-2* **by** *blast*
thus (*lookup q v when t* + *v* = *0*) = *0* **by** *simp*
qed *simp-all*
also have ... = (*lookup q 0 when t* = *0*) **by** (*cases 0* ∈ *keys q*) (*simp-all add*: *in-keys-iff*)
finally show *?thesis* **.**
qed
have ($\sum$ *t*∈*keys p*. *lookup p t* ∗ *lookup q 0 when t* = *0*) =
($\sum$ *t*∈*keys p* ∩ {*0*}. *lookup p t* ∗ *lookup q 0 when t* = *0*)
proof (*intro sum.mono-neutral-right ballI*)
fix *t*
assume *t* ∈ *keys p* − *keys p* ∩ {*0*}
hence *t* ≠ *0* **by** *blast*
thus (*lookup p t* ∗ *lookup q 0 when t* = *0*) = *0* **by** *simp*
qed *simp-all*
also have ... = *lookup p 0* ∗ *lookup q 0* **by** (*cases 0* ∈ *keys p*) (*simp-all add*: *in-keys-iff*)
finally show *?thesis* **by** (*simp add*: *lookup-times eq when-distrib*)
qed

corollary *lookup-prod-zero*:
lookup (*prod f I*) *0* = ($\prod$ *i*∈*I*. *lookup* (*f i*) (*0*::-::{*comm-powerprod*,*ninv-comm-monoid-add*}))
by (*induct I rule*: *infinite-finite-induct*) (*simp-all add*: *lookup-times-zero*)

corollary *lookup-power-zero*:
lookup (*p* ^ *k*) *0* = *lookup p* (*0*::-::{*comm-powerprod*,*ninv-comm-monoid-add*}) ^ *k*
by (*induct k*) (*simp-all add*: *lookup-times-zero*)

definition *poly-eval* :: (*'x* ⇒ *'a*) ⇒ ((*'x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'a*) ⇒ *'a*::*comm-semiring-1*
where *poly-eval a p* = *lookup* (*poly-subst* (λ*y*. *monomial* (*a y*) (*0*::*'x* $\Rightarrow_0$ *nat*)) *p*) *0*

lemma *poly-eval-alt*: *poly-eval a p* = ($\sum$ *t*∈*keys p*. *lookup p t* ∗ ($\prod$ *x*∈*keys t*. *a x* ^ *lookup t x*))
by (*simp add*: *poly-eval-def poly-subst-def lookup-sum lookup-times-zero subst-pp-def lookup-prod-zero lookup-power-zero flip*: *times-monomial-left*)

lemma *poly-eval-monomial*: *poly-eval a* (*monomial c t*) = *c* ∗ ($\prod$ *x*∈*keys t*. *a x* ^

lookup t x)
by (*simp add*: *poly-eval-def poly-subst-monomial subst-pp-def punit.lookup-monom-mult lookup-prod-zero lookup-power-zero*)

lemma *poly-eval-zero* [*simp*]: *poly-eval a 0* = *0*
by (*simp only*: *poly-eval-def poly-subst-zero lookup-zero*)

lemma *poly-eval-zero-left* [*simp*]: *poly-eval 0 p* = *lookup p 0*
by (*simp add*: *poly-eval-def*)

lemma *poly-eval-plus*: *poly-eval a* (*p* + *q*) = *poly-eval a p* + *poly-eval a q*
by (*simp only*: *poly-eval-def poly-subst-plus lookup-add*)

lemma *poly-eval-uminus* [*simp*]: *poly-eval a* (− *p*) = − *poly-eval* (*a*::-::*comm-ring-1*) *p*
by (*simp only*: *poly-eval-def poly-subst-uminus lookup-uminus*)

lemma *poly-eval-minus*: *poly-eval a* (*p* − *q*) = *poly-eval a p* − *poly-eval* (*a*::-::*comm-ring-1*) *q*
by (*simp only*: *poly-eval-def poly-subst-minus lookup-minus*)

lemma *poly-eval-one* [*simp*]: *poly-eval a 1* = *1*
by (*simp add*: *poly-eval-def lookup-one*)

lemma *poly-eval-times*: *poly-eval a* (*p* ∗ *q*) = *poly-eval a p* ∗ *poly-eval a q*
by (*simp only*: *poly-eval-def poly-subst-times lookup-times-zero*)

lemma *poly-eval-power*: *poly-eval a* (*p* ^ *m*) = *poly-eval a p* ^ *m*
by (*induct m*) (*simp-all add*: *poly-eval-times*)

lemma *poly-eval-sum*: *poly-eval a* (*sum f I*) = (∑ *i*∈*I*. *poly-eval a* (*f i*))
by (*induct I rule*: *infinite-finite-induct*) (*simp-all add*: *poly-eval-plus*)

lemma *poly-eval-prod*: *poly-eval a* (*prod f I*) = (∏ *i*∈*I*. *poly-eval a* (*f i*))
by (*induct I rule*: *infinite-finite-induct*) (*simp-all add*: *poly-eval-times*)

lemma *poly-eval-cong*: *p* = *q* ⟹ (⋀*x*. *x* ∈ *indets q* ⟹ *a x* = *b x*) ⟹ *poly-eval a p* = *poly-eval b q*
by (*simp add*: *poly-eval-def cong*: *poly-subst-cong*)

lemma *indets-poly-eval-subset*:
indets (*poly-eval a p*) ⊆ ⋃ (*indets* ‘ *a* ‘ *indets p*) ∪ ⋃ (*indets* ‘ *lookup p* ‘ *keys p*)
proof (*induct p rule*: *poly-mapping-plus-induct*)
case *1*
show *?case* **by** *simp*
next
case (*2 p c t*)
have *keys* (*monomial c t* + *p*) = *keys* (*monomial c t*) ∪ *keys p*
by (*rule keys-plus-eqI*) (*simp add*: *2*(*2*))

with *2*(*1*) **have** *eq1*: *keys* (*monomial c t* + *p*) = *insert t* (*keys p*) **by** *simp*
hence *eq2*: *indets* (*monomial c t* + *p*) = *keys t* ∪ *indets p* **by** (*simp add*: *indets-def*)
from *2*(*2*) **have** *eq3*: *lookup* (*monomial c t* + *p*) *t* = *c* **by** (*simp add*: *lookup-add in-keys-iff*)
have *eq4*: *lookup* (*monomial c t* + *p*) *s* = *lookup p s* **if** *s* ∈ *keys p* **for** *s*
using *that 2*(*2*) **by** (*auto simp*: *lookup-add lookup-single when-def*)
have *indets* (*poly-eval a* (*monomial c t* + *p*)) =
indets (*c* ∗ (∏ *x*∈*keys t*. *a x* ^ *lookup t x*) + *poly-eval a p*)
by (*simp only*: *poly-eval-plus poly-eval-monomial*)
also have ... ⊆ *indets* (*c* ∗ (∏ *x*∈*keys t*. *a x* ^ *lookup t x*)) ∪ *indets* (*poly-eval a p*)
by (*fact indets-plus-subset*)
also have ... ⊆ *indets c* ∪ (⋃ (*indets* ‘ *a* ‘ *keys t*)) ∪
(⋃ (*indets* ‘ *a* ‘ *indets p*) ∪ ⋃ (*indets* ‘ *lookup p* ‘ *keys p*))
proof (*intro Un-mono 2*(*3*))
have *indets* (*c* ∗ (∏ *x*∈*keys t*. *a x* ^ *lookup t x*)) ⊆ *indets c* ∪ *indets* (∏ *x*∈*keys t*. *a x* ^ *lookup t x*)
by (*fact indets-times-subset*)
also have *indets* (∏ *x*∈*keys t*. *a x* ^ *lookup t x*) ⊆ (⋃ *x*∈*keys t*. *indets* (*a x* ^ *lookup t x*))
by (*fact indets-prod-subset*)
also have ... ⊆ (⋃ *x*∈*keys t*. *indets* (*a x*)) **by** (*intro UN-mono subset-refl indets-power-subset*)
also have ... = ⋃ (*indets* ‘ *a* ‘ *keys t*) **by** *simp*
finally show *indets* (*c* ∗ (∏ *x*∈*keys t*. *a x* ^ *lookup t x*)) ⊆ *indets c* ∪ ⋃ (*indets* ‘ *a* ‘ *keys t*)
by *blast*
qed
also have ... = ⋃ (*indets* ‘ *a* ‘ *indets* (*monomial c t* + *p*)) ∪
⋃ (*indets* ‘ *lookup* (*monomial c t* + *p*) ‘ *keys* (*monomial c t* + *p*))
by (*simp add*: *eq1 eq2 eq3 eq4 Un-commute Un-assoc Un-left-commute*)
finally show *?case* **.**
qed

lemma *image-poly-eval-ideal*: *poly-eval a* ‘ *ideal F* = *ideal* (*poly-eval a* ‘ *F*)
proof (*intro image-ideal-eq-surj poly-eval-plus poly-eval-times surjI*)
fix *x*
show *poly-eval a* (*monomial x 0*) = *x* **by** (*simp add*: *poly-eval-monomial*)
qed

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*))
by (*simp-all add*: *map-indets-def poly-subst-uminus poly-subst-plus poly-subst-minus poly-subst-times*
poly-subst-power poly-subst-sum poly-subst-prod)

lemma *map-indets-monomial*:
map-indets f (*monomial c t*) = *monomial c* (∑ *x*∈*keys t*. *Poly-Mapping.single* (*f x*) (*lookup t x*))
by (*simp add*: *map-indets-def poly-subst-monomial subst-pp-def monomial-power-map-scale*
punit.monom-mult-monomial flip: *punit.monomial-prod-sum*)

lemma *map-indets-id*: (⋀*x*. *x* ∈ *indets p* ⟹ *f x* = *x*) ⟹ *map-indets f p* = *p*
by (*simp add*: *map-indets-def poly-subst-id*)

lemma *map-indets-map-indets*: *map-indets f* (*map-indets g p*) = *map-indets* (*f* ∘ *g*) *p*
by (*simp add*: *map-indets-def poly-subst-poly-subst poly-subst-monomial subst-pp-single*)

lemma *map-indets-cong*: *p* = *q* ⟹ (⋀*x*. *x* ∈ *indets q* ⟹ *f x* = *g x*) ⟹ *map-indets f p* = *map-indets g q*
unfolding *map-indets-def* **by** (*simp cong*: *poly-subst-cong*)

lemma *poly-subst-map-indets*: *poly-subst f* (*map-indets g p*) = *poly-subst* (*f* ∘ *g*) *p*
by (*simp add*: *map-indets-def poly-subst-poly-subst poly-subst-monomial subst-pp-single comp-def*)

lemma *poly-eval-map-indets*: *poly-eval a* (*map-indets g p*) = *poly-eval* (*a* ∘ *g*) *p*
by (*simp add*: *poly-eval-def poly-subst-map-indets comp-def*)
(*simp add*: *poly-subst-def lookup-sum lookup-times-zero subst-pp-def lookup-prod-zero*
lookup-power-zero flip: *times-monomial-left*)

lemma *map-indets-inverseE-Polys*:
assumes *inj-on f X* **and** *p* ∈ *P*[*X*]
shows *map-indets* (*the-inv-into X f*) (*map-indets f p*) = *p*
unfolding *map-indets-map-indets*
proof (*rule map-indets-id*)
fix *x*
assume *x* ∈ *indets p*
also from *assms*(*2*) **have** . . . ⊆ *X* **by** (*rule PolysD*)
finally show (*the-inv-into X f* ∘ *f*) *x* = *x* **using** *assms*(*1*) **by** (*auto intro*: *the-inv-into-f-f*)

qed

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 −
  define *g* **where** $g = $ *the-inv f*
  moreover from *assms* **have** *eq*: $g \circ f = id$ **by** (*auto intro*!: *ext the-inv-f-f simp*:
g-def)
  moreover have *map-indets g* $\circ$ *map-indets f* $= id$
    by (*rule ext*) (*simp add*: *map-indets-map-indets eq map-indets-id*)
  ultimately show *?thesis* **..**
qed

lemma *indets-map-indets-subset*: *indets* (*map-indets f* (*p*::- $\Rightarrow_0$ *'a*::*comm-semiring-1*))
$\subseteq$ *f* ‘ *indets p*
proof
  fix *x*
  assume $x \in$ *indets* (*map-indets f p*)
  then obtain *y* **where** $y \in$ *indets p* **and** $x \in$ *indets* (*monomial* (*1*::*'a*) (*Poly-Mapping.single*
(*f y*) *1*))
    unfolding *map-indets-def* **by** (*rule in-indets-poly-substE*)
  from *this*(*2*) **have** *x*: $x = f\ y$ **by** (*simp add*: *indets-monomial*)
  from ‹$y \in$ *indets p*› **show** $x \in$ *f* ‘ *indets p* **unfolding** *x* **by** (*rule imageI*)
qed

corollary *map-indets-in-Polys*: *map-indets f p* $\in$ *P*[*f* ‘ *indets p*]
  using *indets-map-indets-subset* **by** (*rule PolysI-alt*)

lemma *indets-map-indets*:
  assumes *inj-on f* (*indets p*)
  shows *indets* (*map-indets f p*) = *f* ‘ *indets p*
  using *indets-map-indets-subset*
proof (*rule subset-antisym*)
  let *?g* = *the-inv-into* (*indets p*) *f*
  have *p* = *map-indets ?g* (*map-indets f p*) **unfolding** *map-indets-map-indets*
    by (*rule sym*, *rule map-indets-id*) (*simp add*: *assms the-inv-into-f-f*)
  also have *indets* ... $\subseteq$ *?g* ‘ *indets* (*map-indets f p*) **by** (*fact indets-map-indets-subset*)
  finally have *f* ‘ *indets p* $\subseteq$ *f* ‘ *?g* ‘ *indets* (*map-indets f p*) **by** (*rule image-mono*)
  also have ... = ($\lambda x.\ x$) ‘ *indets* (*map-indets f p*) **unfolding** *image-image* **using**
refl
  proof (*rule image-cong*)
    fix *x*
    assume $x \in$ *indets* (*map-indets f p*)
    with *indets-map-indets-subset* **have** $x \in$ *f* ‘ *indets p* **..**
    with *assms* **show** *f* (*?g x*) = *x* **by** (*rule f-the-inv-into-f*)
  qed
  finally show *f* ‘ *indets p* $\subseteq$ *indets* (*map-indets f p*) **by** *simp*

qed

lemma *image-map-indets-Polys*: *map-indets f ‘ P*[*X*] = (*P*[*f ‘ X*]::(- $\Rightarrow_0$ ′*a*::*comm-semiring-1*) *set*)
proof (*intro set-eqI iffI*)
  fix *p* :: - $\Rightarrow_0$ ′*a*
  assume *p* ∈ *map-indets f ‘ P*[*X*]
  then obtain *q* **where** *q* ∈ *P*[*X*] **and** *p* = *map-indets f q* **..**
  note *this*(*2*)
  also have *map-indets f q* ∈ *P*[*f ‘ indets q*] **by** (*fact map-indets-in-Polys*)
  also from ‹*q* ∈ -› **have** ... ⊆ *P*[*f ‘ X*] **by** (*auto intro*!: *Polys-mono imageI dest*: *PolysD*)
  finally show *p* ∈ *P*[*f ‘ X*] **.**
next
  fix *p* :: - $\Rightarrow_0$ ′*a*
  assume *p* ∈ *P*[*f ‘ X*]
  define *g* **where** *g* = (λ*y*. *SOME x*. *x* ∈ *X* ∧ *f x* = *y*)
  have *g y* ∈ *X* ∧ *f* (*g y*) = *y* **if** *y* ∈ *indets p* **for** *y*
  proof −
    note *that*
    also from ‹*p* ∈ -› **have** *indets p* ⊆ *f ‘ X* **by** (*rule PolysD*)
    finally obtain *x* **where** *x* ∈ *X* **and** *y* = *f x* **..**
    hence *x* ∈ *X* ∧ *f x* = *y* **by** *simp*
    thus *?thesis* **unfolding** *g-def* **by** (*rule someI*)
  qed
  hence *1*: *g y* ∈ *X* **and** *2*: *f* (*g y*) = *y* **if** *y* ∈ *indets p* **for** *y* **using** *that* **by** *simp-all*
  show *p* ∈ *map-indets f ‘ P*[*X*]
  proof
    show *p* = *map-indets f* (*map-indets g p*)
      by (*rule sym*) (*simp add*: *map-indets-map-indets map-indets-id 2*)
  next
    have *map-indets g p* ∈ *P*[*g ‘ indets p*] **by** (*fact map-indets-in-Polys*)
    also have ... ⊆ *P*[*X*] **by** (*auto intro*!: *Polys-mono 1*)
    finally show *map-indets g p* ∈ *P*[*X*] **.**
  qed
qed

corollary *range-map-indets*: *range* (*map-indets f*) = *P*[*range f*]
proof −
  have *range* (*map-indets f*) = *map-indets f ‘ P*[*UNIV*] **by** *simp*
  also have ... = *P*[*range f*] **by** (*simp only*: *image-map-indets-Polys*)
  finally show *?thesis* **.**
qed

lemma *in-keys-map-indetsE*:
  assumes *t* ∈ *keys* (*map-indets f* (*p*::- $\Rightarrow_0$ ′*a*::*comm-semiring-1*))
  obtains *s* **where** *s* ∈ *keys p* **and** *t* = (∑*x*∈*keys s*. *Poly-Mapping.single* (*f x*) (*lookup s x*))

proof −
  let *?f* = (λ*x*. *monomial* (*1*::′*a*) (*Poly-Mapping.single* (*f x*) *1*))
  from *assms* **obtain** *s* **where** *s* ∈ *keys p* **and** *t* ∈ *keys* (*subst-pp ?f s*) **unfolding** *map-indets-def*
    by (*rule in-keys-poly-substE*)
  note *this*(*2*)
  also have ... ⊆ {∑ *x*∈*keys s*. *Poly-Mapping.single* (*f x*) (*lookup s x*)}
   by (*simp add*: *subst-pp-def monomial-power-map-scale flip*: *punit.monomial-prod-sum*)
  finally have *t* = (∑ *x*∈*keys s*. *Poly-Mapping.single* (*f x*) (*lookup s x*)) **by** *simp*
  with ‹*s* ∈ *keys p*› **show** *?thesis* **..**
qed

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*
  by (*auto elim*: *in-keys-map-indetsE*)

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*
  using *keys-map-indets-subset*
proof (*rule subset-antisym*)
  let *?g* = *the-inv-into* (*indets p*) *f*
  have *p* = *map-indets ?g* (*map-indets f p*) **unfolding** *map-indets-map-indets*
    by (*rule sym*, *rule map-indets-id*) (*simp add*: *assms the-inv-into-f-f*)
   also have *keys* ... ⊆ (λ*t*. ∑ *x*∈*keys t*. *monomial* (*lookup t x*) (*?g x*)) ‘ *keys* (*map-indets f p*)
    by (*rule keys-map-indets-subset*)
  finally have (λ*t*. ∑ *x*∈*keys t*. *Poly-Mapping.single* (*f x*) (*lookup t x*)) ‘ *keys p* ⊆
              (λ*t*. ∑ *x*∈*keys t*. *Poly-Mapping.single* (*f x*) (*lookup t x*)) ‘
                (λ*t*. ∑ *x*∈*keys t*. *Poly-Mapping.single* (*?g x*) (*lookup t x*)) ‘ *keys* (*map-indets f p*)
    by (*rule image-mono*)
  also from *refl* **have** ... = (λ*t*. ∑ *x*. *Poly-Mapping.single* (*f x*) (*lookup t x*)) ‘
                  (λ*t*. ∑ *x*∈*keys t*. *Poly-Mapping.single* (*?g x*) (*lookup t x*)) ‘ *keys* (*map-indets f p*)
    by (*rule image-cong*)
      (*smt* (*verit*) *Sum-any.conditionalize Sum-any.cong finite-keys not-in-keys-iff-lookup-eq-zero single-zero*)
  also have ... = (λ*t*. *t*) ‘ *keys* (*map-indets f p*) **unfolding** *image-image* **using** *refl*
  proof (*rule image-cong*)
    fix *t*
    assume *t* ∈ *keys* (*map-indets f p*)
   have (∑ *x*. *monomial* (*lookup* (∑ *y*∈*keys t*. *Poly-Mapping.single* (*?g y*) (*lookup t y*)) *x*) (*f x*)) =
          (∑ *x*. ∑ *y*∈*keys t*. *monomial* (*lookup t y when ?g y* = *x*) (*f x*))
      by (*simp add*: *lookup-sum lookup-single monomial-sum*)

also have ... = ($\sum x \in indets\ p.\ \sum y \in keys\ t.$ *Poly-Mapping.single* (*f x*) (*lookup t y when ?g y = x*))
proof (*intro Sum-any.expand-superset finite-indets subsetI*)
fix *x*
assume $x \in \{a.$ ($\sum y \in keys\ t.$ *Poly-Mapping.single* (*f a*) (*lookup t y when ?g y = a*)) $\neq 0\}$
hence ($\sum y \in keys\ t.$ *Poly-Mapping.single* (*f x*) (*lookup t y when ?g y = x*)) $\neq$ *0* **by** *simp*
then obtain *y* **where** $y \in keys\ t$ **and** $*$: *Poly-Mapping.single* (*f x*) (*lookup t y when ?g y = x*) $\neq$ *0*
by (*rule sum.not-neutral-contains-not-neutral*)
from *this*(*1*) **have** $y \in$ *indets* (*map-indets f p*) **using** ‹$t \in$ -› **by** (*rule in-indetsI*)
with *indets-map-indets-subset* **have** $y \in f$ ‘ *indets p* **..**
from $*$ **have** *x = ?g y* **by** (*simp add*: *when-def split*: *if-split-asm*)
also from *assms* ‹$y \in f$ ‘ *indets p*› *subset-refl* **have** ... $\in$ *indets p* **by** (*rule the-inv-into-into*)
finally show $x \in$ *indets p* **.**
qed
also have ... = ($\sum y \in keys\ t.\ \sum x \in indets\ p.$ *Poly-Mapping.single* (*f x*) (*lookup t y when ?g y = x*))
by (*fact sum.swap*)
also from *refl* **have** ... = ($\sum y \in keys\ t.$ *Poly-Mapping.single y* (*lookup t y*))
proof (*rule sum.cong*)
fix *x*
assume $x \in keys\ t$
hence $x \in$ *indets* (*map-indets f p*) **using** ‹$t \in$ -› **by** (*rule in-indetsI*)
with *indets-map-indets-subset* **have** $x \in f$ ‘ *indets p* **..**
with *assms* **have** *?g x* $\in$ *indets p* **using** *subset-refl* **by** (*rule the-inv-into-into*)
hence {*?g x*} $\subseteq$ *indets p* **by** *simp*
with *finite-indets* **have** ($\sum y \in indets\ p.$ *Poly-Mapping.single* (*f y*) (*lookup t x when ?g x = y*)) =
($\sum y \in \{?g\ x\}.$ *Poly-Mapping.single* (*f y*) (*lookup t x when ?g x = y*))
by (*rule sum.mono-neutral-right*) (*simp add*: *monomial-0-iff when-def*)
also from *assms* ‹$x \in f$ ‘ *indets p*› **have** ... = *Poly-Mapping.single x* (*lookup t x*)
by (*simp add*: *f-the-inv-into-f*)
finally show ($\sum y \in indets\ p.$ *Poly-Mapping.single* (*f y*) (*lookup t x when ?g x = y*)) =
Poly-Mapping.single x (*lookup t x*) **.**
qed
also have ... = *t* **by** (*fact poly-mapping-sum-monomials*)
finally show ($\sum x.$ *monomial* (*lookup* ($\sum y \in keys\ t.$ *Poly-Mapping.single* (*?g y*) (*lookup t y*)) *x*) (*f x*)) = *t* **.**
qed
also have ... = *keys* (*map-indets f p*) **by** *simp*
finally show ($\lambda t.\ \sum x \in keys\ t.$ *Poly-Mapping.single* (*f x*) (*lookup t x*)) ‘ *keys p* $\subseteq$ *keys* (*map-indets f p*) **.**

qed

lemma *poly-deg-map-indets-le*: *poly-deg* (*map-indets f p*) ≤ *poly-deg p*
proof (*rule poly-deg-leI*)
  fix *t*
  assume *t* ∈ *keys* (*map-indets f p*)
  then obtain *s* **where** *s* ∈ *keys p* **and** *t*: *t* = (∑ *x*∈*keys s*. *Poly-Mapping.single* (*f x*) (*lookup s x*))
    by (*rule in-keys-map-indetsE*)
  from *this*(*1*) **have** *deg-pm s* ≤ *poly-deg p* **by** (*rule poly-deg-max-keys*)
  thus *deg-pm t* ≤ *poly-deg p*
    by (*simp add*: *t deg-pm-sum deg-pm-single deg-pm-superset*[*OF subset-refl*])
qed

lemma *poly-deg-map-indets*:
  assumes *inj-on f* (*indets p*)
  shows *poly-deg* (*map-indets f p*) = *poly-deg p*
proof −
  from *assms* **have** *deg-pm* ‘ *keys* (*map-indets f p*) = *deg-pm* ‘ *keys p*
    by (*simp add*: *keys-map-indets image-image deg-pm-sum deg-pm-single*
        flip: *deg-pm-superset*[*OF subset-refl*])
  thus *?thesis* **by** (*auto simp*: *poly-deg-def*)
qed

lemma *map-indets-inj-on-PolysI*:
  assumes *inj-on* (*f*::'*x* ⇒ '*y*) *X*
  shows *inj-on* ((*map-indets f*)::- ⇒ - $\Rightarrow_0$ '*a*::*comm-semiring-1*) *P*[*X*]
proof (*rule inj-onI*)
  fix *p q* :: - $\Rightarrow_0$ '*a*
  assume *p* ∈ *P*[*X*]
  with *assms* **have** *1*: *map-indets* (*the-inv-into X f*) (*map-indets f p*) = *p* (**is** *map-indets ?g* - = -)
    by (*rule map-indets-inverseE-Polys*)
  assume *q* ∈ *P*[*X*]
  with *assms* **have** *map-indets ?g* (*map-indets f q*) = *q* **by** (*rule map-indets-inverseE-Polys*)
  moreover assume *map-indets f p* = *map-indets f q*
  ultimately show *p* = *q* **using** *1* **by** (*simp add*: *map-indets-map-indets*)
qed

lemma *map-indets-injI*:
  assumes *inj f*
  shows *inj* (*map-indets f*)
proof −
  from *assms* **have** *inj-on* (*map-indets f*) *P*[*UNIV*] **by** (*rule map-indets-inj-on-PolysI*)
  thus *?thesis* **by** *simp*
qed

lemma *image-map-indets-ideal*:
  assumes *inj f*

shows *map-indets f ‘ ideal F* = *ideal* (*map-indets f ‘* (*F*::(- $\Rightarrow_0$ *'a*::*comm-ring-1*) *set*)) ∩ *P*[*range f*]
proof
from *map-indets-plus map-indets-times* **have** *map-indets f ‘ ideal F* ⊆ *ideal* (*map-indets f ‘ F*)
by (*rule image-ideal-subset*)
moreover from *subset-UNIV* **have** *map-indets f ‘ ideal F* ⊆ *range* (*map-indets f*) **by** (*rule image-mono*)
ultimately show *map-indets f ‘ ideal F* ⊆ *ideal* (*map-indets f ‘ F*) ∩ *P*[*range f*]
unfolding *range-map-indets* **by** *blast*
next
show *ideal* (*map-indets f ‘ F*) ∩ *P*[*range f*] ⊆ *map-indets f ‘ ideal F*
proof
fix *p*
assume *p* ∈ *ideal* (*map-indets f ‘ F*) ∩ *P*[*range f*]
hence *p* ∈ *ideal* (*map-indets f ‘ F*) **and** *p* ∈ *range* (*map-indets f*)
by (*simp-all add*: *range-map-indets*)
from *this*(*1*) **obtain** *F0 q* **where** *F0* ⊆ *map-indets f ‘ F* **and** *p*: *p* = ($\sum$ *f'*∈*F0*. *q f'* ∗ *f'*)
by (*rule ideal.spanE*)
from *this*(*1*) **obtain** *F'* **where** *F'* ⊆ *F* **and** *F0*: *F0* = *map-indets f ‘ F'* **by** (*rule subset-imageE*)
from *assms* **obtain** *g* **where** *map-indets g* ∘ *map-indets f* = (*id*::- ⇒ - $\Rightarrow_0$ *'a*)
by (*rule map-indets-inverseE*)
hence *eq*: *map-indets g* (*map-indets f p'*) = *p'* **for** *p'*::- $\Rightarrow_0$ *'a*
by (*simp add*: *pointfree-idE*)
from *assms* **have** *inj* (*map-indets f*) **by** (*rule map-indets-injI*)
from *this subset-UNIV* **have** *inj-on* (*map-indets f*) *F'* **by** (*rule inj-on-subset*)
from ‹*p* ∈ *range* -› **obtain** *p'* **where** *p* = *map-indets f p'* **..**
hence *p* = *map-indets f* (*map-indets g p*) **by** (*simp add*: *eq*)
also from ‹*inj-on* - *F'*› **have** ... = *map-indets f* ($\sum$ *f'*∈*F'*. *map-indets g* (*q* (*map-indets f f'*)) ∗ *f'*)
by (*simp add*: *p F0 sum.reindex map-indets-sum map-indets-times eq*)
finally have *p* = *map-indets f* ($\sum$ *f'*∈*F'*. *map-indets g* (*q* (*map-indets f f'*)) ∗ *f'*) **.**
moreover have ($\sum$ *f'*∈*F'*. *map-indets g* (*q* (*map-indets f f'*)) ∗ *f'*) ∈ *ideal F*
proof
show ($\sum$ *f'*∈*F'*. *map-indets g* (*q* (*map-indets f f'*)) ∗ *f'*) ∈ *ideal F'* **by** (*rule ideal.sum-in-spanI*)
next
from ‹*F'* ⊆ *F*› **show** *ideal F'* ⊆ *ideal F* **by** (*rule ideal.span-mono*)
qed
ultimately show *p* ∈ *map-indets f ‘ ideal F* **by** (*rule image-eqI*)
qed
qed

17.6 Homogeneity

definition *homogeneous* :: ((*'x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'a*::*zero*) ⇒ *bool*

where *homogeneous p* ⟷ (∀ *s*∈*keys p*. ∀ *t*∈*keys p*. *deg-pm s* = *deg-pm t*)

definition *hom-component* :: ((′*x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ ′*a*) ⇒ *nat* ⇒ ((′*x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ ′*a*::*zero*)
where *hom-component p n* = *except p* {*t*. *deg-pm t* ≠ *n*}

definition *hom-components* :: ((′*x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ ′*a*) ⇒ ((′*x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ ′*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* ⇒ *bool*
where *homogeneous-set A* ⟷ (∀ *a*∈*A*. ∀ *n*. *hom-component a n* ∈ *A*)

lemma *homogeneousI*: (⋀*s t*. *s* ∈ *keys p* ⟹ *t* ∈ *keys p* ⟹ *deg-pm s* = *deg-pm t*) ⟹ *homogeneous p*
unfolding *homogeneous-def* **by** *blast*

lemma *homogeneousD*: *homogeneous p* ⟹ *s* ∈ *keys p* ⟹ *t* ∈ *keys p* ⟹ *deg-pm s* = *deg-pm t*
unfolding *homogeneous-def* **by** *blast*

lemma *homogeneousD-poly-deg*:
assumes *homogeneous p* **and** *t* ∈ *keys p*
shows *deg-pm t* = *poly-deg p*
proof (*rule antisym*)
from *assms*(*2*) **show** *deg-pm t* ≤ *poly-deg p* **by** (*rule poly-deg-max-keys*)
next
show *poly-deg p* ≤ *deg-pm t*
proof (*rule poly-deg-leI*)
fix *s*
assume *s* ∈ *keys p*
with *assms*(*1*) **have** *deg-pm s* = *deg-pm t* **using** *assms*(*2*) **by** (*rule homogeneousD*)
thus *deg-pm s* ≤ *deg-pm t* **by** *simp*
qed
qed

lemma *homogeneous-monomial* [*simp*]: *homogeneous* (*monomial c t*)
by (*auto split*: *if-split-asm intro*: *homogeneousI*)

corollary *homogeneous-zero* [*simp*]: *homogeneous 0* **and** *homogeneous-one* [*simp*]: *homogeneous 1*
by (*simp-all only*: *homogeneous-monomial flip*: *single-zero*[*of 0*] *single-one*)

lemma *homogeneous-uminus-iff* [*simp*]: *homogeneous* (− *p*) ⟷ *homogeneous p*
by (*auto intro*!: *homogeneousI dest*: *homogeneousD simp*: *keys-uminus*)

lemma *homogeneous-monom-mult*: *homogeneous p* ⟹ *homogeneous* (*punit.monom-mult c t p*)

by (*auto intro*!: *homogeneousI elim*!: *punit.keys-monom-multE simp*: *deg-pm-plus dest*: *homogeneousD*)

lemma *homogeneous-monom-mult-rev*:
assumes $c \neq (0::'a::semiring\text{-}no\text{-}zero\text{-}divisors)$ **and** *homogeneous* (*punit.monom-mult c t p*)
shows *homogeneous p*
proof (*rule homogeneousI*)
fix *s s'*
assume $s \in keys\ p$
hence *1*: $t + s \in keys$ (*punit.monom-mult c t p*)
using *assms*(*1*) **by** (*rule punit.keys-monom-multI*[*simplified*])
assume $s' \in keys\ p$
hence $t + s' \in keys$ (*punit.monom-mult c t p*)
using *assms*(*1*) **by** (*rule punit.keys-monom-multI*[*simplified*])
with *assms*(*2*) *1* **have** *deg-pm* $(t + s) =$ *deg-pm* $(t + s')$ **by** (*rule homogeneousD*)
thus *deg-pm* $s =$ *deg-pm* s' **by** (*simp add*: *deg-pm-plus*)
qed

lemma *homogeneous-times*:
assumes *homogeneous p* **and** *homogeneous q*
shows *homogeneous* $(p * q)$
proof (*rule homogeneousI*)
fix *s t*
assume $s \in keys\ (p * q)$
then obtain *sp sq* **where** *sp*: $sp \in keys\ p$ **and** *sq*: $sq \in keys\ q$ **and** *s*: $s = sp + sq$
by (*rule in-keys-timesE*)
assume $t \in keys\ (p * q)$
then obtain *tp tq* **where** *tp*: $tp \in keys\ p$ **and** *tq*: $tq \in keys\ q$ **and** *t*: $t = tp + tq$
by (*rule in-keys-timesE*)
from *assms*(*1*) *sp tp* **have** *deg-pm sp* = *deg-pm tp* **by** (*rule homogeneousD*)
moreover from *assms*(*2*) *sq tq* **have** *deg-pm sq* = *deg-pm tq* **by** (*rule homogeneousD*)
ultimately show *deg-pm s* = *deg-pm t* **by** (*simp only*: *s t deg-pm-plus*)
qed

lemma *lookup-hom-component*: *lookup* (*hom-component p n*) = (λt. *lookup p t when deg-pm* $t = n$)
by (*rule ext*) (*simp add*: *hom-component-def lookup-except*)

lemma *keys-hom-component*: *keys* (*hom-component p n*) = $\{t.\ t \in keys\ p \wedge$ *deg-pm* $t = n\}$
by (*auto simp*: *hom-component-def keys-except*)

lemma *keys-hom-componentD*:
assumes $t \in keys$ (*hom-component p n*)
shows $t \in keys\ p$ **and** *deg-pm* $t = n$
using *assms* **by** (*simp-all add*: *keys-hom-component*)

lemma *homogeneous-hom-component*: *homogeneous* (*hom-component p n*)
by (*auto dest*: *keys-hom-componentD intro*: *homogeneousI*)

lemma *hom-component-zero* [*simp*]: *hom-component 0* = *0*
by (*rule ext*) (*simp add*: *hom-component-def*)

lemma *hom-component-zero-iff*: *hom-component p n* = *0* ⟷ (∀ *t*∈*keys p*. *deg-pm t* ≠ *n*)
by (*metis* (*mono-tags*, *lifting*) *empty-iff keys-eq-empty-iff keys-hom-component mem-Collect-eq subsetI subset-antisym*)

lemma *hom-component-uminus* [*simp*]: *hom-component* (− *p*) = − *hom-component p*
by (*intro ext poly-mapping-eqI*) (*simp add*: *hom-component-def lookup-except*)

lemma *hom-component-plus*: *hom-component* (*p* + *q*) *n* = *hom-component p n* + *hom-component q n*
by (*rule poly-mapping-eqI*) (*simp add*: *hom-component-def lookup-except lookup-add*)

lemma *hom-component-minus*: *hom-component* (*p* − *q*) *n* = *hom-component p n* − *hom-component q n*
by (*rule poly-mapping-eqI*) (*simp add*: *hom-component-def lookup-except lookup-minus*)

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*)
by (*auto simp*: *hom-component-def lookup-except punit.lookup-monom-mult deg-pm-minus deg-pm-mono intro*!: *poly-mapping-eqI*)

lemma *hom-component-inject*:
assumes *t* ∈ *keys p* **and** *hom-component p* (*deg-pm t*) = *hom-component p n*
shows *deg-pm t* = *n*
proof −
from *assms*(*1*) **have** *t* ∈ *keys* (*hom-component p* (*deg-pm t*)) **by** (*simp add*: *keys-hom-component*)
hence *0* ≠ *lookup* (*hom-component p* (*deg-pm t*)) *t* **by** (*simp add*: *in-keys-iff*)
also have *lookup* (*hom-component p* (*deg-pm t*)) *t* = *lookup* (*hom-component p n*) *t*
by (*simp only*: *assms*(*2*))
also have . . . = (*lookup p t when deg-pm t* = *n*) **by** (*simp only*: *lookup-hom-component*)
finally show *?thesis* **by** *simp*
qed

lemma *hom-component-of-homogeneous*:
assumes *homogeneous p*
shows *hom-component p n* = (*p when n* = *poly-deg p*)
proof (*cases n* = *poly-deg p*)
case *True*

have *hom-component p n = p*
proof (*rule poly-mapping-eqI*)
fix *t*
show *lookup* (*hom-component p n*) *t = lookup p t*
proof (*cases t ∈ keys p*)
case *True*
with *assms* **have** *deg-pm t = n* **unfolding** ‹*n = poly-deg p*› **by** (*rule homogeneousD-poly-deg*)
thus *?thesis* **by** (*simp add*: *lookup-hom-component*)
next
case *False*
moreover from *this* **have** *t ∉ keys* (*hom-component p n*) **by** (*simp add*: *keys-hom-component*)
ultimately show *?thesis* **by** (*simp add*: *in-keys-iff*)
qed
qed
with *True* **show** *?thesis* **by** *simp*
next
case *False*
have *hom-component p n = 0* **unfolding** *hom-component-zero-iff*
proof (*intro ballI notI*)
fix *t*
assume *t ∈ keys p*
with *assms* **have** *deg-pm t = poly-deg p* **by** (*rule homogeneousD-poly-deg*)
moreover assume *deg-pm t = n*
ultimately show *False* **using** *False* **by** *simp*
qed
with *False* **show** *?thesis* **by** *simp*
qed

lemma *hom-components-zero* [*simp*]: *hom-components 0* = {}
by (*simp add*: *hom-components-def*)

lemma *hom-components-zero-iff* [*simp*]: *hom-components p* = {} ⟷ *p = 0*
by (*simp add*: *hom-components-def*)

lemma *hom-components-uminus*: *hom-components* (− *p*) = *uminus* ‘ *hom-components p*
by (*simp add*: *hom-components-def keys-uminus image-image*)

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*
by (*simp add*: *hom-components-def punit.keys-monom-mult image-image deg-pm-plus hom-component-monom-mult*)

lemma *hom-componentsI*: *q = hom-component p* (*deg-pm t*) ⟹ *t ∈ keys p* ⟹ *q ∈ hom-components p*

unfolding *hom-components-def* **by** *blast*

lemma *hom-componentsE*:
assumes $q \in$ *hom-components p*
obtains *t* **where** $t \in$ *keys p* **and** $q =$ *hom-component p* (*deg-pm t*)
using *assms* **unfolding** *hom-components-def* **by** *blast*

lemma *hom-components-of-homogeneous*:
assumes *homogeneous p*
shows *hom-components p* = (*if p* = *0 then* {} *else* {*p*})
proof (*split if-split*, *intro conjI impI*)
assume $p \neq 0$
have *deg-pm* ' *keys p* = {*poly-deg p*}
proof (*rule set-eqI*)
fix *n*
have $n \in$ *deg-pm* ' *keys p* $\longleftrightarrow$ *n* = *poly-deg p*
proof
assume $n \in$ *deg-pm* ' *keys p*
then obtain *t* **where** $t \in$ *keys p* **and** *n* = *deg-pm t* **..**
from *assms this*(*1*) **have** *deg-pm t* = *poly-deg p* **by** (*rule homogeneousD-poly-deg*)
thus *n* = *poly-deg p* **by** (*simp only*: ‹*n* = *deg-pm t*›)
next
assume *n* = *poly-deg p*
from ‹$p \neq 0$› **have** *keys p* ≠ {} **by** *simp*
then obtain *t* **where** $t \in$ *keys p* **by** *blast*
with *assms* **have** *deg-pm t* = *poly-deg p* **by** (*rule homogeneousD-poly-deg*)
hence *n* = *deg-pm t* **by** (*simp only*: ‹*n* = *poly-deg p*›)
with ‹$t \in$ *keys p*› **show** $n \in$ *deg-pm* ' *keys p* **by** (*rule rev-image-eqI*)
qed
thus $n \in$ *deg-pm* ' *keys p* $\longleftrightarrow$ $n \in$ {*poly-deg p*} **by** *simp*
qed
with *assms* **show** *hom-components p* = {*p*}
by (*simp add*: *hom-components-def hom-component-of-homogeneous*)
qed *simp*

lemma *finite-hom-components*: *finite* (*hom-components p*)
unfolding *hom-components-def* **using** *finite-keys* **by** (*intro finite-imageI*)

lemma *hom-components-homogeneous*: $q \in$ *hom-components p* $\Longrightarrow$ *homogeneous q*
by (*elim hom-componentsE*) (*simp only*: *homogeneous-hom-component*)

lemma *hom-components-nonzero*: $q \in$ *hom-components p* $\Longrightarrow$ $q \neq 0$
by (*auto elim*!: *hom-componentsE simp*: *hom-component-zero-iff*)

lemma *deg-pm-hom-components*:
assumes $q1 \in$ *hom-components p* **and** $q2 \in$ *hom-components p* **and** $t1 \in$ *keys q1* **and** $t2 \in$ *keys q2*
shows *deg-pm t1* = *deg-pm t2* $\longleftrightarrow$ *q1* = *q2*

proof −
from *assms*(*1*) **obtain** *s1* **where** *s1* ∈ *keys p* **and** *q1*: *q1* = *hom-component p* (*deg-pm s1*)
by (*rule hom-componentsE*)
from *assms*(*3*) **have** *t1*: *deg-pm t1* = *deg-pm s1* **unfolding** *q1* **by** (*rule keys-hom-componentD*)
from *assms*(*2*) **obtain** *s2* **where** *s2* ∈ *keys p* **and** *q2*: *q2* = *hom-component p* (*deg-pm s2*)
by (*rule hom-componentsE*)
from *assms*(*4*) **have** *t2*: *deg-pm t2* = *deg-pm s2* **unfolding** *q2* **by** (*rule keys-hom-componentD*)
from ‹*s1* ∈ *keys p*› **show** *?thesis* **by** (*auto simp*: *q1 q2 t1 t2 dest*: *hom-component-inject*)
qed

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 −
from *assms*(*1*) **have** *homogeneous q1* **and** *q1* ≠ *0*
by (*rule hom-components-homogeneous*, *rule hom-components-nonzero*)
from *this*(*2*) **have** *keys q1* ≠ {} **by** *simp*
then obtain *t1* **where** *t1* ∈ *keys q1* **by** *blast*
with ‹*homogeneous q1*› **have** *t1*: *deg-pm t1* = *poly-deg q1* **by** (*rule homogeneousD-poly-deg*)
from *assms*(*2*) **have** *homogeneous q2* **and** *q2* ≠ *0*
by (*rule hom-components-homogeneous*, *rule hom-components-nonzero*)
from *this*(*2*) **have** *keys q2* ≠ {} **by** *simp*
then obtain *t2* **where** *t2* ∈ *keys q2* **by** *blast*
with ‹*homogeneous q2*› **have** *t2*: *deg-pm t2* = *poly-deg q2* **by** (*rule homogeneousD-poly-deg*)
from *assms* ‹*t1* ∈ *keys q1*› ‹*t2* ∈ *keys q2*› **have** *deg-pm t1* = *deg-pm t2* ⟷ *q1* = *q2*
by (*rule deg-pm-hom-components*)
thus *?thesis* **by** (*simp only*: *t1 t2*)
qed

lemma *hom-components-keys-disjoint*:
assumes *q1* ∈ *hom-components p* **and** *q2* ∈ *hom-components p* **and** *q1* ≠ *q2*
shows *keys q1* ∩ *keys q2* = {}
proof (*rule ccontr*)
assume *keys q1* ∩ *keys q2* ≠ {}
then obtain *t* **where** *t* ∈ *keys q1* **and** *t* ∈ *keys q2* **by** *blast*
with *assms*(*1*, *2*) **have** *deg-pm t* = *deg-pm t* ⟷ *q1* = *q2* **by** (*rule deg-pm-hom-components*)
with *assms*(*3*) **show** *False* **by** *simp*
qed

lemma *Keys-hom-components*: *Keys* (*hom-components p*) = *keys p*
by (*auto simp*: *Keys-def hom-components-def keys-hom-component*)

lemma *lookup-hom-components*: *q* ∈ *hom-components p* ⟹ *t* ∈ *keys q* ⟹ *lookup q t* = *lookup p t*

by (*auto elim*!: *hom-componentsE simp*: *keys-hom-component lookup-hom-component*)

lemma *poly-deg-hom-components-le*:
  assumes $q \in$ *hom-components p*
  shows *poly-deg q* $\leq$ *poly-deg p*
proof (*rule poly-deg-leI*)
  fix *t*
  assume $t \in$ *keys q*
  also from *assms* **have** ... $\subseteq$ *Keys* (*hom-components p*) **by** (*rule keys-subset-Keys*)
  also have ... = *keys p* **by** (*fact Keys-hom-components*)
  finally show *deg-pm t* $\leq$ *poly-deg p* **by** (*rule poly-deg-max-keys*)
qed

lemma *sum-hom-components*: $\sum$(*hom-components p*) = *p*
proof (*rule poly-mapping-eqI*)
  fix *t*
  show *lookup* ($\sum$(*hom-components p*)) *t* = *lookup p t* **unfolding** *lookup-sum*
  proof (*cases* $t \in$ *keys p*)
    case *True*
    also have *keys p* = *Keys* (*hom-components p*) **by** (*simp only*: *Keys-hom-components*)
    finally obtain *q* **where** *q*: $q \in$ *hom-components p* **and** *t*: $t \in$ *keys q* **by** (*rule in-KeysE*)
    from *this*(*1*) **have** ($\sum$ *q0*$\in$*hom-components p*. *lookup q0 t*) =
                    ($\sum$ *q0*$\in$*insert q* (*hom-components p*). *lookup q0 t*)
      by (*simp only*: *insert-absorb*)
    also from *finite-hom-components* **have** ... = *lookup q t* + ($\sum$ *q0*$\in$*hom-components p* $-$ {*q*}. *lookup q0 t*)
      by (*rule sum.insert-remove*)
    also from *q t* **have** ... = *lookup p t* + ($\sum$ *q0*$\in$*hom-components p* $-$ {*q*}. *lookup q0 t*)
      by (*simp only*: *lookup-hom-components*)
    also have ($\sum$ *q0*$\in$*hom-components p* $-$ {*q*}. *lookup q0 t*) = *0*
    proof (*intro sum.neutral ballI*)
      fix *q0*
      assume *q0* $\in$ *hom-components p* $-$ {*q*}
      hence *q0* $\in$ *hom-components p* **and** $q \neq$ *q0* **by** *blast*+
      with *q* **have** *keys q* $\cap$ *keys q0* = {} **by** (*rule hom-components-keys-disjoint*)
      with *t* **have** $t \notin$ *keys q0* **by** *blast*
      thus *lookup q0 t* = *0* **by** (*simp add*: *in-keys-iff*)
    qed
    finally show ($\sum$ *q*$\in$*hom-components p*. *lookup q t*) = *lookup p t* **by** *simp*
  next
    case *False*
    hence $t \notin$ *Keys* (*hom-components p*) **by** (*simp add*: *Keys-hom-components*)
    hence $\forall$ *q*$\in$*hom-components p*. *lookup q t* = *0* **by** (*simp add*: *Keys-def in-keys-iff*)
    hence ($\sum$ *q*$\in$*hom-components p*. *lookup q t*) = *0* **by** (*rule sum.neutral*)
    also from *False* **have** ... = *lookup p t* **by** (*simp add*: *in-keys-iff*)
    finally show ($\sum$ *q*$\in$*hom-components p*. *lookup q t*) = *lookup p t* **.**
  qed

qed

lemma *homogeneous-setI*: $(\bigwedge a\ n.\ a \in A \Longrightarrow hom\text{-}component\ a\ n \in A) \Longrightarrow$ *homogeneous-set A*
  by (*simp add*: *homogeneous-set-def*)

lemma *homogeneous-setD*: *homogeneous-set A* $\Longrightarrow a \in A \Longrightarrow$ *hom-component a n* $\in A$
  by (*simp add*: *homogeneous-set-def*)

lemma *homogeneous-set-Polys*: *homogeneous-set* $(P[X]::(\text{-} \Rightarrow_0 {'a}::zero)\ set)$
proof (*intro homogeneous-setI PolysI subsetI*)
  fix $p::\text{-} \Rightarrow_0 {'a}$ **and** *n t*
  assume $p \in P[X]$
  assume $t \in keys\ (hom\text{-}component\ p\ n)$
  hence $t \in keys\ p$ **by** (*rule keys-hom-componentD*)
  also from ‹$p \in P[X]$› **have** $\ldots \subseteq .[X]$ **by** (*rule PolysD*)
  finally show $t \in .[X]$ **.**
qed

lemma *homogeneous-set-IntI*: *homogeneous-set A* $\Longrightarrow$ *homogeneous-set B* $\Longrightarrow$ *homogeneous-set* $(A \cap B)$
  by (*simp add*: *homogeneous-set-def*)

lemma *homogeneous-setD-hom-components*:
  assumes *homogeneous-set A* **and** $a \in A$ **and** $b \in hom\text{-}components\ a$
  shows $b \in A$
proof −
  from *assms*(*3*) **obtain** $t::{'a} \Rightarrow_0 nat$ **where** $b = hom\text{-}component\ a\ (deg\text{-}pm\ t)$
    by (*rule hom-componentsE*)
  also from *assms*(*1*, *2*) **have** $\ldots \in A$ **by** (*rule homogeneous-setD*)
  finally show *?thesis* **.**
qed

lemma *zero-in-homogeneous-set*:
  assumes *homogeneous-set A* **and** $A \neq \{\}$
  shows $0 \in A$
proof −
  from *assms*(*2*) **obtain** *a* **where** $a \in A$ **by** *blast*
  have *lookup a t* $= 0$ **if** *deg-pm t* $=$ *Suc* (*poly-deg a*) **for** *t*
  proof (*rule ccontr*)
    assume *lookup a t* $\neq 0$
    hence $t \in keys\ a$ **by** (*simp add*: *in-keys-iff*)
    hence *deg-pm t* $\leq$ *poly-deg a* **by** (*rule poly-deg-max-keys*)
    thus *False* **by** (*simp add*: *that*)
  qed
  hence $0 =$ *hom-component a* (*Suc* (*poly-deg a*))
    by (*intro poly-mapping-eqI*) (*simp add*: *lookup-hom-component when-def*)
  also from *assms*(*1*) ‹$a \in A$› **have** $\ldots \in A$ **by** (*rule homogeneous-setD*)

finally show *?thesis* **.**
qed

lemma *homogeneous-ideal*:
assumes $\bigwedge f.\ f \in F \Longrightarrow$ *homogeneous f* **and** $p \in$ *ideal F*
shows *hom-component p n* $\in$ *ideal F*
proof −
from *assms*(*2*) **have** $p \in$ *punit.pmdl F* **by** *simp*
thus *?thesis*
proof (*induct p rule*: *punit.pmdl-induct*)
case *module-0*
show *?case* **by** (*simp add*: *ideal.span-zero*)
next
case (*module-plus a f c t*)
let *?f* = *punit.monom-mult c t f*
from *module-plus.hyps*(*3*) **have** $f \in$ *punit.pmdl F* **by** (*simp add*: *ideal.span-base*)
hence ∗: *?f* $\in$ *punit.pmdl F* **by** (*rule punit.pmdl-closed-monom-mult*)
from *module-plus.hyps*(*3*) **have** *homogeneous f* **by** (*rule assms*(*1*))
hence *homogeneous ?f* **by** (*rule homogeneous-monom-mult*)
hence *hom-component ?f n* = (*?f when n* = *poly-deg ?f*) **by** (*rule hom-component-of-homogeneous*)
also from ∗ **have** ... $\in$ *ideal F* **by** (*simp add*: *when-def ideal.span-zero*)
finally have *hom-component ?f n* $\in$ *ideal F* **.**
with *module-plus.hyps*(*2*) **show** *?case* **unfolding** *hom-component-plus* **by** (*rule ideal.span-add*)
qed
qed

corollary *homogeneous-set-homogeneous-ideal*:
($\bigwedge f.\ f \in F \Longrightarrow$ *homogeneous f*) $\Longrightarrow$ *homogeneous-set* (*ideal F*)
by (*auto intro*: *homogeneous-setI homogeneous-ideal*)

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*
using - *assms*(*2*, *3*)
proof (*rule homogeneous-setD-hom-components*)
from *assms*(*1*) **show** *homogeneous-set* (*ideal F*) **by** (*rule homogeneous-set-homogeneous-ideal*)
qed

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 −
from *assms*(*2*) **obtain** F'' q' **where** *finite* F'' **and** $F'' \subseteq F$ **and** *p*: $p = (\sum f \in F''.\ q'\ f * f)$

by (*rule ideal.spanE*)
let *?A* = λf. {*h* ∈ *hom-components* (*q′ f*). *poly-deg h* + *poly-deg f* = *poly-deg p*}
let *?B* = λf. {*h* ∈ *hom-components* (*q′ f*). *poly-deg h* + *poly-deg f* ≠ *poly-deg p*}
define *F′* **where** *F′* = {*f* ∈ *F″*. (∑ (*?A f*)) ∗ *f* ≠ *0*}
define *q* **where** *q* = (λf. (∑ (*?A f*)) *when f* ∈ *F′*)
have *F′* ⊆ *F″* **by** (*simp add*: *F′-def*)
hence *F′* ⊆ *F* **using** ‹*F″* ⊆ *F*› **by** (*rule subset-trans*)
have *1*: *deg-pm t* + *poly-deg f* = *poly-deg p* **if** *f* ∈ *F′* **and** *t* ∈ *keys* (*q f*) **for** *f t*
proof −
from *that* **have** *t* ∈ *keys* (∑ (*?A f*)) **by** (*simp add*: *q-def*)
also have ... ⊆ (⋃*h*∈*?A f*. *keys h*) **by** (*fact keys-sum-subset*)
finally obtain *h* **where** *h* ∈ *?A f* **and** *t* ∈ *keys h* **..**
from *this*(*1*) **have** *h* ∈ *hom-components* (*q′ f*) **and** *eq*: *poly-deg h* + *poly-deg f* = *poly-deg p*
by *simp-all*
from *this*(*1*) **have** *homogeneous h* **by** (*rule hom-components-homogeneous*)
hence *deg-pm t* = *poly-deg h* **using** ‹*t* ∈ *keys h*› **by** (*rule homogeneousD-poly-deg*)
thus *?thesis* **by** (*simp only*: *eq*)
qed
have *2*: *deg-pm t* = *poly-deg p* **if** *f* ∈ *F′* **and** *t* ∈ *keys* (*q f* ∗ *f*) **for** *f t*
proof −
from *that*(*1*) ‹*F′* ⊆ *F*› **have** *f* ∈ *F* **..**
hence *homogeneous f* **by** (*rule assms*(*1*))
from *that*(*2*) **obtain** *s1 s2* **where** *s1* ∈ *keys* (*q f*) **and** *s2* ∈ *keys f* **and** *t*: *t* = *s1* + *s2*
by (*rule in-keys-timesE*)
from *that*(*1*) *this*(*1*) **have** *deg-pm s1* + *poly-deg f* = *poly-deg p* **by** (*rule 1*)
moreover from ‹*homogeneous f*› ‹*s2* ∈ *keys f*› **have** *deg-pm s2* = *poly-deg f*
by (*rule homogeneousD-poly-deg*)
ultimately show *?thesis* **by** (*simp add*: *t deg-pm-plus*)
qed
from ‹*F′* ⊆ *F″*› ‹*finite F″*› **have** *finite F′* **by** (*rule finite-subset*)
thus *?thesis* **using** ‹*F′* ⊆ *F*›
proof
note *p*
also from *refl* **have** (∑*f*∈*F″*. *q′ f* ∗ *f*) = (∑*f*∈*F″*. (∑ (*?A f*) ∗ *f*) + (∑ (*?B f*) ∗ *f*))
proof (*rule sum.cong*)
fix *f*
assume *f* ∈ *F″*
from *sum-hom-components* **have** *q′ f* = (∑ (*hom-components* (*q′ f*))) **by** (*rule sym*)
also have ... = (∑ (*?A f* ∪ *?B f*)) **by** (*rule arg-cong*[**where** *f*=*sum* (λx. *x*)]) *blast*
also have ... = ∑ (*?A f*) + ∑ (*?B f*)
proof (*rule sum.union-disjoint*)
have *?A f* ⊆ *hom-components* (*q′ f*) **by** *blast*
thus *finite* (*?A f*) **using** *finite-hom-components* **by** (*rule finite-subset*)
next

have *?B f* ⊆ *hom-components* (*q′ f*) **by** *blast*
thus *finite* (*?B f*) **using** *finite-hom-components* **by** (*rule finite-subset*)
qed *blast*
finally show $q'\ f * f = (\sum (?A\ f) * f) + (\sum (?B\ f) * f)$
by (*metis* (*no-types*, *lifting*) *distrib-right*)
qed
also have $\ldots = (\sum f{\in}F''. \sum (?A\ f) * f) + (\sum f{\in}F''. \sum (?B\ f) * f)$ **by** (*rule sum.distrib*)
also from ‹*finite F″*› ‹*F′* ⊆ *F″*› **have** $(\sum f{\in}F''. \sum (?A\ f) * f) = (\sum f{\in}F'. q\ f * f)$
proof (*intro sum.mono-neutral-cong-right ballI*)
fix *f*
assume $f \in F'' - F'$
thus $\sum (?A\ f) * f = 0$ **by** (*simp add*: *F′-def*)
next
fix *f*
assume $f \in F'$
thus $\sum (?A\ f) * f = q\ f * f$ **by** (*simp add*: *q-def*)
qed
finally have *p*[*symmetric*]: $p = (\sum f{\in}F'. q\ f * f) + (\sum f{\in}F''. \sum (?B\ f) * f)$ **.**
moreover have $keys\ (\sum f{\in}F''. \sum (?B\ f) * f) = \{\}$
proof (*rule*, *rule*)
fix *t*
assume *t-in*: $t \in keys\ (\sum f{\in}F''. \sum (?B\ f) * f)$
also have $\ldots \subseteq (\bigcup f{\in}F''. keys\ (\sum (?B\ f) * f))$ **by** (*fact keys-sum-subset*)
finally obtain *f* **where** $f \in F''$ **and** $t \in keys\ (\sum (?B\ f) * f)$ **..**
from *this*(*2*) **obtain** *s1 s2* **where** $s1 \in keys\ (\sum (?B\ f))$ **and** $s2 \in keys\ f$ **and** *t*: $t = s1 + s2$
by (*rule in-keys-timesE*)
from ‹$f \in F''$› ‹$F'' \subseteq F$› **have** $f \in F$ **..**
hence *homogeneous f* **by** (*rule assms*(*1*))
note ‹$s1 \in keys\ (\sum (?B\ f))$›
also have $keys\ (\sum (?B\ f)) \subseteq (\bigcup h{\in}?B\ f.\ keys\ h)$ **by** (*fact keys-sum-subset*)
finally obtain *h* **where** $h \in ?B\ f$ **and** $s1 \in keys\ h$ **..**
from *this*(*1*) **have** $h \in$ *hom-components* (*q′ f*) **and** *neq*: *poly-deg h* + *poly-deg f* ≠ *poly-deg p*
by *simp-all*
from *this*(*1*) **have** *homogeneous h* **by** (*rule hom-components-homogeneous*)
hence *deg-pm s1* = *poly-deg h* **using** ‹$s1 \in keys\ h$› **by** (*rule homogeneousD-poly-deg*)
moreover from ‹*homogeneous f*› ‹$s2 \in keys\ f$› **have** *deg-pm s2* = *poly-deg f*
by (*rule homogeneousD-poly-deg*)
ultimately have *deg-pm t* ≠ *poly-deg p* **using** *neq* **by** (*simp add*: *t deg-pm-plus*)
have $t \notin keys\ (\sum f{\in}F'. q\ f * f)$
proof
assume $t \in keys\ (\sum f{\in}F'. q\ f * f)$
also have $\ldots \subseteq (\bigcup f{\in}F'. keys\ (q\ f * f))$ **by** (*fact keys-sum-subset*)
finally obtain *f* **where** $f \in F'$ **and** $t \in keys\ (q\ f * f)$ **..**
hence *deg-pm t* = *poly-deg p* **by** (*rule 2*)

with ‹*deg-pm t* ≠ *poly-deg p*› **show** *False* **..**
qed
with *t-in* **have** $t \in keys\ ((\sum f{\in}F'.\ q\ f * f) + (\sum f{\in}F''.\ \sum (?B\ f) * f))$
by (*rule in-keys-plusI2*)
hence $t \in keys\ p$ **by** (*simp only*: *p*)
with *assms*(*3*) **have** *deg-pm t* = *poly-deg p* **by** (*rule homogeneousD-poly-deg*)
with ‹*deg-pm t* ≠ *poly-deg p*› **show** $t \in \{\}$ **..**
qed (*fact empty-subsetI*)
ultimately show $p = (\sum f{\in}F'.\ q\ f * f)$ **by** *simp*
next
fix *f*
show *homogeneous* (*q f*)
proof (*cases* $f \in F'$)
case *True*
show *?thesis*
proof (*rule homogeneousI*)
fix *s t*
assume $s \in keys\ (q\ f)$
with *True* **have** ∗: *deg-pm s* + *poly-deg f* = *poly-deg p* **by** (*rule 1*)
assume $t \in keys\ (q\ f)$
with *True* **have** *deg-pm t* + *poly-deg f* = *poly-deg p* **by** (*rule 1*)
with ∗ **show** *deg-pm s* = *deg-pm t* **by** *simp*
qed
next
case *False*
thus *?thesis* **by** (*simp add*: *q-def*)
qed

assume $f \in F'$
show *poly-deg* (*q f* ∗ *f*) = *poly-deg p*
proof (*intro antisym*)
show *poly-deg* (*q f* ∗ *f*) ≤ *poly-deg p*
proof (*rule poly-deg-leI*)
fix *t*
assume $t \in keys\ (q\ f * f)$
with ‹$f \in F'$› **have** *deg-pm t* = *poly-deg p* **by** (*rule 2*)
thus *deg-pm t* ≤ *poly-deg p* **by** *simp*
qed
next
from ‹$f \in F'$› **have** *q f* ∗ *f* ≠ *0* **by** (*simp add*: *q-def F'-def*)
hence *keys* (*q f* ∗ *f*) ≠ {} **by** *simp*
then obtain *t* **where** $t \in keys\ (q\ f * f)$ **by** *blast*
with ‹$f \in F'$› **have** *deg-pm t* = *poly-deg p* **by** (*rule 2*)
moreover from ‹$t \in keys\ (q\ f * f)$› **have** *deg-pm t* ≤ *poly-deg* (*q f* ∗ *f*) **by**
(*rule poly-deg-max-keys*)
ultimately show *poly-deg p* ≤ *poly-deg* (*q f* ∗ *f*) **by** *simp*
qed
qed (*simp add*: *q-def*)
qed

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 (*cases* $p = 0$)
  case *True*
  show *?thesis*
  proof
    show $p = (\sum f{\in}\{\}.\ (\lambda\text{-}.\ 0)\ f * f)$ **by** (*simp add*: *True*)
  qed *simp-all*
next
  case *False*
  define *P* **where** $P = (\lambda h\ qf.$ *finite* (*fst qf*) $\wedge$ *fst qf* $\subseteq F \wedge h = (\sum f{\in}$*fst qf*. *snd qf f* $* f) \wedge$
          $(\forall f{\in}$*fst qf*. *poly-deg* (*snd qf f* $* f$) = *poly-deg h*) $\wedge$ ($\forall f.\ f \notin$ *fst qf* $\longrightarrow$ *snd qf f* $= 0$))
  define *q0* **where** *q0* = (λh. *SOME qf*. *P h qf*)
  have *1*: *P h* (*q0 h*) **if** $h \in$ *hom-components p* **for** *h*
  proof −
    note *assms*(*1*)
    moreover from *assms that* **have** $h \in$ *ideal F* **by** (*rule homogeneous-ideal′*)
    moreover from *that* **have** *homogeneous h* **by** (*rule hom-components-homogeneous*)
    ultimately obtain F' *q* **where** *finite* F' **and** $F' \subseteq F$ **and** $h = (\sum f{\in}F'.\ q\ f * f)$
      and $\bigwedge f.\ f \in F' \Longrightarrow$ *poly-deg* $(q\ f * f) =$ *poly-deg h* **and** $\bigwedge f.\ f \notin F' \Longrightarrow q\ f = 0$
      by (*rule homogeneous-idealE-homogeneous*) *blast+*
    hence *P h* (F', *q*) **by** (*simp add*: *P-def*)
    thus *?thesis* **unfolding** *q0-def* **by** (*rule someI*)
  qed
  define F' **where** $F' = (\bigcup h{\in}$*hom-components p*. *fst* (*q0 h*))
  define *q* **where** $q = (\lambda f.\ \sum h{\in}$*hom-components p*. *snd* (*q0 h*) *f*)
  show *?thesis*
  proof
    have *finite* $F' \wedge F' \subseteq F$ **unfolding** F'*-def UN-subset-iff finite-UN*[*OF finite-hom-components*]
    proof (*intro conjI ballI*)
      fix *h*
      assume $h \in$ *hom-components p*
      hence *P h* (*q0 h*) **by** (*rule 1*)
      thus *finite* (*fst* (*q0 h*)) **and** *fst* (*q0 h*) $\subseteq F$ **by** (*simp-all only*: *P-def*)
    qed
    thus *finite* F' **and** $F' \subseteq F$ **by** *simp-all*

    from *sum-hom-components* **have** $p = (\sum$ (*hom-components p*)) **by** (*rule sym*)
    also from *refl* **have** $\ldots = (\sum h{\in}$*hom-components p*. $\sum f{\in}F'.$ *snd* (*q0 h*) $f * f$)
    proof (*rule sum.cong*)

fix h
assume $h \in$ *hom-components p*
hence $P\ h\ (q0\ h)$ **by** (*rule 1*)
hence $h = (\sum f \in fst\ (q0\ h).\ snd\ (q0\ h)\ f * f)$ **and** *2*: $\bigwedge f.\ f \notin fst\ (q0\ h) \Longrightarrow snd\ (q0\ h)\ f = 0$
by (*simp-all add*: *P-def*)
note *this*(*1*)
also from ‹*finite F*′› **have** $(\sum f \in fst\ (q0\ h).\ (snd\ (q0\ h))\ f * f) = (\sum f \in F'.\ snd\ (q0\ h)\ f * f)$
proof (*intro sum.mono-neutral-left ballI*)
show $fst\ (q0\ h) \subseteq F'$ **unfolding** *F*′*-def* **using** ‹$h \in$ *hom-components p*› **by** *blast*
next
fix f
assume $f \in F' - fst\ (q0\ h)$
hence $f \notin fst\ (q0\ h)$ **by** *simp*
hence $snd\ (q0\ h)\ f = 0$ **by** (*rule 2*)
thus $snd\ (q0\ h)\ f * f = 0$ **by** *simp*
qed
finally show $h = (\sum f \in F'.\ snd\ (q0\ h)\ f * f)$ **.**
qed
also have $\ldots = (\sum f \in F'.\ \sum h \in hom\text{-}components\ p.\ snd\ (q0\ h)\ f * f)$ **by** (*rule sum.swap*)
also have $\ldots = (\sum f \in F'.\ q\ f * f)$ **by** (*simp only*: *q-def sum-distrib-right*)
finally show $p = (\sum f \in F'.\ q\ f * f)$ **.**

fix f
have $poly\text{-}deg\ (q\ f * f) = poly\text{-}deg\ (\sum h \in hom\text{-}components\ p.\ snd\ (q0\ h)\ f * f)$
by (*simp only*: *q-def sum-distrib-right*)
also have $\ldots \le Max\ (poly\text{-}deg\ `\ (\lambda h.\ snd\ (q0\ h)\ f * f)\ `\ hom\text{-}components\ p)$
by (*rule poly-deg-sum-le*)
also have $\ldots = Max\ ((\lambda h.\ poly\text{-}deg\ (snd\ (q0\ h)\ f * f))\ `\ hom\text{-}components\ p)$
(**is** - = *Max* (*?f* ‘ -)) **by** (*simp only*: *image-image*)
also have $\ldots \le poly\text{-}deg\ p$
proof (*rule Max.boundedI*)
from *finite-hom-components* **show** *finite* (*?f* ‘ *hom-components p*) **by** (*rule finite-imageI*)
next
from *False* **show** *?f* ‘ *hom-components p* ≠ {} **by** *simp*
next
fix d
assume $d \in$ *?f* ‘ *hom-components p*
then obtain h **where** $h \in$ *hom-components p* **and** *d*: $d =$ *?f h* **..**
from *this*(*1*) **have** $P\ h\ (q0\ h)$ **by** (*rule 1*)
hence *2*: $\bigwedge f.\ f \in fst\ (q0\ h) \Longrightarrow poly\text{-}deg\ (snd\ (q0\ h)\ f * f) = poly\text{-}deg\ h$
and *3*: $\bigwedge f.\ f \notin fst\ (q0\ h) \Longrightarrow snd\ (q0\ h)\ f = 0$ **by** (*simp-all add*: *P-def*)
show $d \le poly\text{-}deg\ p$
proof (*cases* $f \in fst\ (q0\ h)$)
case *True*

hence *poly-deg* (*snd* (*q0 h*) *f* $*$ *f*) = *poly-deg h* **by** (*rule 2*)
hence *d* = *poly-deg h* **by** (*simp only*: *d*)
also from ‹*h* ∈ *hom-components p*› **have** ... ≤ *poly-deg p* **by** (*rule poly-deg-hom-components-le*)
finally show *?thesis* **.**
next
case *False*
hence *snd* (*q0 h*) *f* = *0* **by** (*rule 3*)
thus *?thesis* **by** (*simp add*: *d*)
qed
qed
finally show *poly-deg* (*q f* $*$ *f*) ≤ *poly-deg p* **.**

assume *f* ∉ *F'*
show *q f* = *0* **unfolding** *q-def*
proof (*intro sum.neutral ballI*)
fix *h*
assume *h* ∈ *hom-components p*
hence *P h* (*q0 h*) **by** (*rule 1*)
hence *2*: ⋀*f*. *f* ∉ *fst* (*q0 h*) ⟹ *snd* (*q0 h*) *f* = *0* **by** (*simp add*: *P-def*)
show *snd* (*q0 h*) *f* = *0*
proof (*intro 2 notI*)
assume *f* ∈ *fst* (*q0 h*)
hence *f* ∈ *F'* **unfolding** *F'-def* **using** ‹*h* ∈ *hom-components p*› **by** *blast*
with ‹*f* ∉ *F'*› **show** *False* **..**
qed
qed
qed
qed

corollary *homogeneous-idealE-finite*:
assumes *finite F* **and** ⋀*f*. *f* ∈ *F* ⟹ *homogeneous f* **and** *p* ∈ *ideal F*
obtains *q* **where** *p* = (∑*f*∈*F*. *q f* $*$ *f*) **and** ⋀*f*. *poly-deg* (*q f* $*$ *f*) ≤ *poly-deg p*
and ⋀*f*. *f* ∉ *F* ⟹ *q f* = *0*
proof −
from *assms*(*2*, *3*) **obtain** *F' q* **where** *F'* ⊆ *F* **and** *p*: *p* = (∑*f*∈*F'*. *q f* $*$ *f*)
and ⋀*f*. *poly-deg* (*q f* $*$ *f*) ≤ *poly-deg p* **and** *1*: ⋀*f*. *f* ∉ *F'* ⟹ *q f* = *0*
by (*rule homogeneous-idealE*) *blast+*
show *?thesis*
proof
from *assms*(*1*) ‹*F'* ⊆ *F*› **have** (∑*f*∈*F'*. *q f* $*$ *f*) = (∑*f*∈*F*. *q f* $*$ *f*)
proof (*intro sum.mono-neutral-left ballI*)
fix *f*
assume *f* ∈ *F* − *F'*
hence *f* ∉ *F'* **by** *simp*
hence *q f* = *0* **by** (*rule 1*)
thus *q f* $*$ *f* = *0* **by** *simp*
qed
thus *p* = (∑*f*∈*F*. *q f* $*$ *f*) **by** (*simp only*: *p*)

next
fix *f*
show *poly-deg* (*q f* ∗ *f*) ≤ *poly-deg p* **by** *fact*

assume *f* ∉ *F*
with ‹*F*′ ⊆ *F*› **have** *f* ∉ *F*′ **by** *blast*
thus *q f* = *0* **by** (*rule 1*)
qed
qed

17.6.1 Homogenization and Dehomogenization

definition *homogenize* :: ′*x* ⇒ ((′*x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ ′*a*) ⇒ ((′*x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ ′*a*::*semiring-1*)
where *homogenize x p* = (∑ *t*∈*keys p*. *monomial* (*lookup p t*) (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm t*) + *t*))

definition *dehomo-subst* :: ′*x* ⇒ ′*x* ⇒ ((′*x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ ′*a*::*zero-neq-one*)
where *dehomo-subst x* = (λ*y*. *if y* = *x then 1 else monomial 1* (*Poly-Mapping.single y 1*))

definition *dehomogenize* :: ′*x* ⇒ ((′*x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ ′*a*) ⇒ ((′*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*
by (*simp add*: *homogenize-def*)

lemma *homogenize-uminus* [*simp*]: *homogenize x* (− *p*) = − *homogenize x* (*p*::- $\Rightarrow_0$ ′*a*::*ring-1*)
by (*simp add*: *homogenize-def keys-uminus sum.reindex inj-on-def single-uminus sum-negf*)

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 (*cases p* = *0*)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
show *?thesis*
proof (*cases c* = *0*)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
show *?thesis*
by (*simp add*: *homogenize-def punit.keys-monom-mult* ‹*p* ≠ *0*› *False sum.reindex punit.lookup-monom-mult punit.monom-mult-sum-right poly-deg-monom-mult*

punit.monom-mult-monomial ac-simps deg-pm-plus)
qed
qed

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 −
have *homogenize x p* = (∑ *t*∈*Keys* (*hom-components p*). *monomial* (*lookup p t*) (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm t*) + *t*))
by (*simp only*: *homogenize-def Keys-hom-components*)
also have ... = (∑ *t*∈(⋃ (*keys* ' *hom-components p*)). *monomial* (*lookup p t*) (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm t*) + *t*))
by (*simp only*: *Keys-def*)
also have ... = (∑ *q*∈*hom-components p*. (∑ *t*∈*keys q*. *monomial* (*lookup p t*) (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm t*) + *t*)))
by (*auto intro*!: *sum.UNION-disjoint finite-hom-components finite-keys dest*: *hom-components-keys-disjoint*)
also have ... = (∑ *q*∈*hom-components p*. *punit.monom-mult 1* (*Poly-Mapping.single x* (*poly-deg p* − *poly-deg q*)) *q*)
using *refl*
proof (*rule sum.cong*)
fix *q*
assume *q*: *q* ∈ *hom-components p*
hence *homogeneous q* **by** (*rule hom-components-homogeneous*)
have (∑ *t*∈*keys q*. *monomial* (*lookup p t*) (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm t*) + *t*)) =
(∑ *t*∈*keys q*. *punit.monom-mult 1* (*Poly-Mapping.single x* (*poly-deg p* − *poly-deg q*)) (*monomial* (*lookup q t*) *t*))
using *refl*
proof (*rule sum.cong*)
fix *t*
assume *t* ∈ *keys q*
with ‹*homogeneous q*› **have** *deg-pm t* = *poly-deg q* **by** (*rule homogeneousD-poly-deg*)
moreover from *q* ‹*t* ∈ *keys q*› **have** *lookup q t* = *lookup p t* **by** (*rule lookup-hom-components*)
ultimately show *monomial* (*lookup p t*) (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm t*) + *t*) =
punit.monom-mult 1 (*Poly-Mapping.single x* (*poly-deg p* − *poly-deg q*)) (*monomial* (*lookup q t*) *t*)
by (*simp add*: *punit.monom-mult-monomial*)
qed
also have ... = *punit.monom-mult 1* (*Poly-Mapping.single x* (*poly-deg p* − *poly-deg q*)) *q*
by (*simp only*: *poly-mapping-sum-monomials flip*: *punit.monom-mult-sum-right*)
finally show (∑ *t*∈*keys q*. *monomial* (*lookup p t*) (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm t*) + *t*)) =
punit.monom-mult 1 (*Poly-Mapping.single x* (*poly-deg p* − *poly-deg q*)) *q* .

qed
finally show *?thesis* **.**
qed

lemma *keys-homogenizeE*:
assumes $t \in$ *keys* (*homogenize x p*)
obtains t' **where** $t' \in$ *keys p* **and** $t =$ *Poly-Mapping.single x* (*poly-deg p* − *deg-pm* t') + t'
proof −
note *assms*
also have *keys* (*homogenize x p*) ⊆
(⋃ *t*∈*keys p*. *keys* (*monomial* (*lookup p t*) (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm t*) + *t*)))
unfolding *homogenize-def* **by** (*rule keys-sum-subset*)
finally obtain t' **where** $t' \in$ *keys p*
and $t \in$ *keys* (*monomial* (*lookup p* t') (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm* t') + t')) **..**
from *this*(*2*) **have** $t =$ *Poly-Mapping.single x* (*poly-deg p* − *deg-pm* t') + t'
by (*simp split*: *if-split-asm*)
with ‹$t' \in$ *keys p*› **show** *?thesis* **..**
qed

lemma *keys-homogenizeE-alt*:
assumes $t \in$ *keys* (*homogenize x p*)
obtains *q* t' **where** $q \in$ *hom-components p* **and** $t' \in$ *keys q*
and $t =$ *Poly-Mapping.single x* (*poly-deg p* − *poly-deg q*) + t'
proof −
note *assms*
also have *keys* (*homogenize x p*) ⊆
(⋃ *q*∈*hom-components p*. *keys* (*punit.monom-mult 1* (*Poly-Mapping.single x* (*poly-deg p* − *poly-deg q*)) *q*))
unfolding *homogenize-alt* **by** (*rule keys-sum-subset*)
finally obtain *q* **where** *q*: $q \in$ *hom-components p*
and $t \in$ *keys* (*punit.monom-mult 1* (*Poly-Mapping.single x* (*poly-deg p* − *poly-deg q*)) *q*) **..**
note *this*(*2*)
also have ... ⊆ (+) (*Poly-Mapping.single x* (*poly-deg p* − *poly-deg q*)) ‘ *keys q*
by (*rule punit.keys-monom-mult-subset*[*simplified*])
finally obtain t' **where** $t' \in$ *keys q* **and** $t =$ *Poly-Mapping.single x* (*poly-deg p* − *poly-deg q*) + t' **..**
with *q* **show** *?thesis* **..**
qed

lemma *deg-pm-homogenize*:
assumes $t \in$ *keys* (*homogenize x p*)
shows *deg-pm t* = *poly-deg p*
proof −
from *assms* **obtain** *q* t' **where** *q*: $q \in$ *hom-components p* **and** $t' \in$ *keys q*
and *t*: $t =$ *Poly-Mapping.single x* (*poly-deg p* − *poly-deg q*) + t' **by** (*rule*

keys-homogenizeE-alt)
from *q* **have** *homogeneous q* **by** (*rule hom-components-homogeneous*)
hence *deg-pm t′ = poly-deg q* **using** ‹*t′ ∈ keys q*› **by** (*rule homogeneousD-poly-deg*)
moreover from *q* **have** *poly-deg q ≤ poly-deg p* **by** (*rule poly-deg-hom-components-le*)
ultimately show *?thesis* **by** (*simp add*: *t deg-pm-plus deg-pm-single*)
qed

corollary *homogeneous-homogenize*: *homogeneous* (*homogenize x p*)
proof (*rule homogeneousI*)
fix *s t*
assume *s ∈ keys* (*homogenize x p*)
hence ∗: *deg-pm s = poly-deg p* **by** (*rule deg-pm-homogenize*)
assume *t ∈ keys* (*homogenize x p*)
hence *deg-pm t = poly-deg p* **by** (*rule deg-pm-homogenize*)
with ∗ **show** *deg-pm s = deg-pm t* **by** *simp*
qed

corollary *poly-deg-homogenize-le*: *poly-deg* (*homogenize x p*) *≤ poly-deg p*
proof (*rule poly-deg-leI*)
fix *t*
assume *t ∈ keys* (*homogenize x p*)
hence *deg-pm t = poly-deg p* **by** (*rule deg-pm-homogenize*)
thus *deg-pm t ≤ poly-deg p* **by** *simp*
qed

lemma *homogenize-id-iff* [*simp*]: *homogenize x p = p ⟷ homogeneous p*
proof
assume *homogenize x p = p*
moreover have *homogeneous* (*homogenize x p*) **by** (*fact homogeneous-homogenize*)
ultimately show *homogeneous p* **by** *simp*
next
assume *homogeneous p*
hence *hom-components p* = (*if p = 0 then* {} *else* {*p*}) **by** (*rule hom-components-of-homogeneous*)
thus *homogenize x p = p* **by** (*simp add*: *homogenize-alt split*: *if-split-asm*)
qed

lemma *homogenize-homogenize* [*simp*]: *homogenize x* (*homogenize x p*) = *homogenize x p*
by (*simp add*: *homogeneous-homogenize*)

lemma *homogenize-monomial*: *homogenize x* (*monomial c t*) = *monomial c t*
by (*simp only*: *homogenize-id-iff homogeneous-monomial*)

lemma *indets-homogenize-subset*: *indets* (*homogenize x p*) *⊆ insert x* (*indets p*)
proof
fix *y*
assume *y ∈ indets* (*homogenize x p*)
then obtain *t* **where** *t ∈ keys* (*homogenize x p*) **and** *y ∈ keys t* **by** (*rule in-indetsE*)

from *this*(*1*) **obtain** *t′* **where** *t′* ∈ *keys p*
and *t*: *t* = *Poly-Mapping.single x* (*poly-deg p* − *deg-pm t′*) + *t′* **by** (*rule keys-homogenizeE*)
note ‹*y* ∈ *keys t*›
also have *keys t* ⊆ *keys* (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm t′*)) ∪ *keys t′*
unfolding *t* **by** (*rule Poly-Mapping.keys-add*)
finally show *y* ∈ *insert x* (*indets p*)
proof
assume *y* ∈ *keys* (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm t′*))
thus *?thesis* **by** (*simp split*: *if-split-asm*)
next
assume *y* ∈ *keys t′*
hence *y* ∈ *indets p* **using** ‹*t′* ∈ *keys p*› **by** (*rule in-indetsI*)
thus *?thesis* **by** *simp*
qed
qed

lemma *homogenize-in-Polys*: *p* ∈ *P*[*X*] =⇒ *homogenize x p* ∈ *P*[*insert x X*]
using *indets-homogenize-subset*[*of x p*] **by** (*auto simp*: *Polys-alt*)

lemma *lookup-homogenize*:
assumes *x* ∉ *indets p* **and** *x* ∉ *keys t*
shows *lookup* (*homogenize x p*) (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm t*) + *t*) = *lookup p t*
proof −
let *?p* = *homogenize x p*
let *?t* = *Poly-Mapping.single x* (*poly-deg p* − *deg-pm t*) + *t*
have *eq*: (∑ *s*∈*keys p* − {*t*}. *lookup* (*monomial* (*lookup p s*) (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm s*) + *s*)) *?t*) = *0*
proof (*intro sum.neutral ballI*)
fix *s*
assume *s* ∈ *keys p* − {*t*}
hence *s* ∈ *keys p* **and** *s* ≠ *t* **by** *simp-all*
from *this*(*1*) **have** *keys s* ⊆ *indets p* **by** (*simp add*: *in-indetsI subsetI*)
with *assms*(*1*) **have** *x* ∉ *keys s* **by** *blast*
have *?t* ≠ *Poly-Mapping.single x* (*poly-deg p* − *deg-pm s*) + *s*
proof
assume *a*: *?t* = *Poly-Mapping.single x* (*poly-deg p* − *deg-pm s*) + *s*
hence *lookup ?t x* = *lookup* (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm s*) + *s*) *x*
by *simp*
moreover from *assms*(*2*) **have** *lookup t x* = *0* **by** (*simp add*: *in-keys-iff*)
moreover from ‹*x* ∉ *keys s*› **have** *lookup s x* = *0* **by** (*simp add*: *in-keys-iff*)
ultimately have *poly-deg p* − *deg-pm t* = *poly-deg p* − *deg-pm s* **by** (*simp add*: *lookup-add*)
with *a* **have** *s* = *t* **by** *simp*
with ‹*s* ≠ *t*› **show** *False* **..**
qed

thus *lookup* (*monomial* (*lookup p s*) (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm s*) + *s*)) *?t* = *0*
by (*simp add*: *lookup-single*)
qed
show *?thesis*
proof (*cases t* ∈ *keys p*)
case *True*
have *lookup ?p ?t* = (∑ *s*∈*keys p*. *lookup* (*monomial* (*lookup p s*) (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm s*) + *s*)) *?t*)
by (*simp add*: *homogenize-def lookup-sum*)
also have . . . = *lookup* (*monomial* (*lookup p t*) *?t*) *?t* +
(∑ *s*∈*keys p* − {*t*}. *lookup* (*monomial* (*lookup p s*) (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm s*) + *s*)) *?t*)
using *finite-keys True* **by** (*rule sum.remove*)
also have . . . = *lookup p t* **by** (*simp add*: *eq*)
finally show *?thesis* **.**
next
case *False*
hence *1*: *keys p* − {*t*} = *keys p* **by** *simp*
have *lookup ?p ?t* = (∑ *s*∈*keys p* − {*t*}. *lookup* (*monomial* (*lookup p s*) (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm s*) + *s*)) *?t*)
by (*simp add*: *homogenize-def lookup-sum 1*)
also have . . . = *0* **by** (*simp only*: *eq*)
also from *False* **have** . . . = *lookup p t* **by** (*simp add*: *in-keys-iff*)
finally show *?thesis* **.**
qed
qed

lemma *keys-homogenizeI*:
assumes *x* ∉ *indets p* **and** *t* ∈ *keys p*
shows *Poly-Mapping.single x* (*poly-deg p* − *deg-pm t*) + *t* ∈ *keys* (*homogenize x p*) (**is** *?t* ∈ *keys ?p*)
proof −
from *assms*(*2*) **have** *keys t* ⊆ *indets p* **by** (*simp add*: *in-indetsI subsetI*)
with *assms*(*1*) **have** *x* ∉ *keys t* **by** *blast*
with *assms*(*1*) **have** *lookup ?p ?t* = *lookup p t* **by** (*rule lookup-homogenize*)
also from *assms*(*2*) **have** . . . ≠ *0* **by** (*simp add*: *in-keys-iff*)
finally show *?thesis* **by** (*simp add*: *in-keys-iff*)
qed

lemma *keys-homogenize*:
x ∉ *indets p* =⇒ *keys* (*homogenize x p*) = (λ*t*. *Poly-Mapping.single x* (*poly-deg p* − *deg-pm t*) + *t*) ‘ *keys p*
by (*auto intro*: *keys-homogenizeI elim*: *keys-homogenizeE*)

lemma *card-keys-homogenize*:
assumes *x* ∉ *indets p*
shows *card* (*keys* (*homogenize x p*)) = *card* (*keys p*)
unfolding *keys-homogenize*[*OF assms*]

```
proof (intro card-image inj-onI)
  fix s t
  assume s ∈ keys p and t ∈ keys p
  with assms have x ∉ keys s and x ∉ keys t by (auto dest: in-indetsI simp only:)
  let ?s = Poly-Mapping.single x (poly-deg p − deg-pm s)
  let ?t = Poly-Mapping.single x (poly-deg p − deg-pm t)
  assume ?s + s = ?t + t
  hence lookup (?s + s) x = lookup (?t + t) x by simp
  with ‹x ∉ keys s› ‹x ∉ keys t› have ?s = ?t by (simp add: lookup-add in-keys-iff)
  with ‹?s + s = ?t + t› show s = t by simp
qed

lemma poly-deg-homogenize:
  assumes x ∉ indets p
  shows poly-deg (homogenize x p) = poly-deg p
proof (cases p = 0)
  case True
  thus ?thesis by simp
next
  case False
  then obtain t where t ∈ keys p and 1: poly-deg p = deg-pm t by (rule poly-degE)
  from assms this(1) have Poly-Mapping.single x (poly-deg p − deg-pm t) + t ∈
keys (homogenize x p)
    by (rule keys-homogenizeI)
  hence t ∈ keys (homogenize x p) by (simp add: 1)
  hence poly-deg p ≤ poly-deg (homogenize x p) unfolding 1 by (rule poly-deg-max-keys)
  with poly-deg-homogenize-le show ?thesis by (rule antisym)
qed

lemma maxdeg-homogenize:
  assumes x ∉ ⋃ (indets ‘ F)
  shows maxdeg (homogenize x ‘ F) = maxdeg F
  unfolding maxdeg-def image-image
proof (rule arg-cong[where f=Max], rule set-eqI)
  fix d
  show d ∈ (λf. poly-deg (homogenize x f)) ‘ F ⟷ d ∈ poly-deg ‘ F
  proof
    assume d ∈ (λf. poly-deg (homogenize x f)) ‘ F
    then obtain f where f ∈ F and d: d = poly-deg (homogenize x f) ..
    from assms this(1) have x ∉ indets f by blast
    hence d = poly-deg f by (simp add: d poly-deg-homogenize)
    with ‹f ∈ F› show d ∈ poly-deg ‘ F by (rule rev-image-eqI)
  next
    assume d ∈ poly-deg ‘ F
    then obtain f where f ∈ F and d: d = poly-deg f ..
    from assms this(1) have x ∉ indets f by blast
    hence d = poly-deg (homogenize x f) by (simp add: d poly-deg-homogenize)
    with ‹f ∈ F› show d ∈ (λf. poly-deg (homogenize x f)) ‘ F by (rule rev-image-eqI)
  qed
```

qed

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 –
  have *homogenize x p* = ($\sum$ *q*∈*hom-components p. punit.monom-mult 1* (*Poly-Mapping.single x* (*poly-deg p* − *poly-deg q*)) *q*)
    by (*fact homogenize-alt*)
  also have ... ∈ *ideal F*
  proof (*rule ideal.span-sum*)
    fix *q*
    assume *q* ∈ *hom-components p*
    with *assms* **have** *q* ∈ *ideal F* **by** (*rule homogeneous-ideal′*)
    thus *punit.monom-mult 1* (*Poly-Mapping.single x* (*poly-deg p* − *poly-deg q*)) *q* ∈ *ideal F*
      by (*rule punit.pmdl-closed-monom-mult*[*simplified*])
  qed
  finally show *?thesis* **.**
qed

lemma *subst-pp-dehomo-subst* [*simp*]:
  subst-pp (*dehomo-subst x*) *t* = *monomial* (*1*::′*b*::*comm-semiring-1*) (*except t* {*x*})
proof –
  have *subst-pp* (*dehomo-subst x*) *t* = (($\prod$ *y*∈*keys t. dehomo-subst x y* ^ *lookup t y*)::- $\Rightarrow_0$ ′*b*)
    by (*fact subst-pp-def*)
  also have ... = ($\prod$ *y*∈*keys t* − {*y0. dehomo-subst x y0* ^ *lookup t y0* = (*1*::- $\Rightarrow_0$ ′*b*)}. *dehomo-subst x y* ^ *lookup t y*)
    by (*rule sym*, *rule prod.setdiff-irrelevant*, *fact finite-keys*)
  also have ... = ($\prod$ *y*∈*keys t* − {*x*}. *monomial 1* (*Poly-Mapping.single y 1*) ^ *lookup t y*)
  proof (*rule prod.cong*)
    have *dehomo-subst x x* ^ *lookup t x* = *1* **by** (*simp add*: *dehomo-subst-def*)
    moreover {
      fix *y*
      assume *y* ≠ *x*
      hence *dehomo-subst x y* ^ *lookup t y* = *monomial 1* (*Poly-Mapping.single y* (*lookup t y*))
        by (*simp add*: *dehomo-subst-def monomial-single-power*)
      moreover assume *dehomo-subst x y* ^ *lookup t y* = *1*
      ultimately have *Poly-Mapping.single y* (*lookup t y*) = *0*
        by (*smt* (*verit*) *single-one monomial-inj zero-neq-one*)
      hence *lookup t y* = *0* **by** (*rule monomial-0D*)
      moreover assume *y* ∈ *keys t*
      ultimately have *False* **by** (*simp add*: *in-keys-iff*)
    }
    ultimately show *keys t* − {*y0. dehomo-subst x y0* ^ *lookup t y0* = *1*} = *keys t* − {*x*} **by** *auto*

qed (*simp add*: *dehomo-subst-def*)
also have ... = ($\prod$ *y*∈*keys t* − {*x*}. *monomial 1* (*Poly-Mapping.single y* (*lookup t y*)))
by (*simp add*: *monomial-single-power*)
also have ... = *monomial 1* ($\sum$ *y*∈*keys t* − {*x*}. *Poly-Mapping.single y* (*lookup t y*))
by (*simp flip*: *punit.monomial-prod-sum*)
also have ($\sum$ *y*∈*keys t* − {*x*}. *Poly-Mapping.single y* (*lookup t y*)) = *except t* {*x*}
proof (*rule poly-mapping-eqI*, *simp add*: *lookup-sum lookup-except lookup-single*, *rule*)
fix *y*
assume *y* ≠ *x*
show ($\sum$ *z*∈*keys t* − {*x*}. *lookup t z when z* = *y*) = *lookup t y*
proof (*cases y* ∈ *keys t*)
case *True*
have *finite* (*keys t* − {*x*}) **by** *simp*
moreover from *True* ‹*y* ≠ *x*› **have** *y* ∈ *keys t* − {*x*} **by** *simp*
ultimately have ($\sum$ *z*∈*keys t* − {*x*}. *lookup t z when z* = *y*) =
(*lookup t y when y* = *y*) + ($\sum$ *z*∈*keys t* − {*x*} − {*y*}. *lookup t z when z* = *y*)
by (*rule sum.remove*)
also have ($\sum$ *z*∈*keys t* − {*x*} − {*y*}. *lookup t z when z* = *y*) = *0* **by** *auto*
finally show *?thesis* **by** *simp*
next
case *False*
hence ($\sum$ *z*∈*keys t* − {*x*}. *lookup t z when z* = *y*) = *0* **by** (*auto simp*: *when-def*)
also from *False* **have** ... = *lookup t y* **by** (*simp add*: *in-keys-iff*)
finally show *?thesis* **.**
qed
qed
finally show *?thesis* **.**
qed

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*::-⇒$_0$ -::*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) = ($\sum a{\in}A$. *dehomogenize* x (f a))
and *dehomogenize-prod*: *dehomogenize* x (*prod* f A) = ($\prod a{\in}A$. *dehomogenize* x (f a))
by (*simp-all add*: *dehomogenize-def poly-subst-monomial poly-subst-plus poly-subst-uminus poly-subst-minus poly-subst-times poly-subst-power poly-subst-sum poly-subst-prod punit.monom-mult-monomial*)

corollary *dehomogenize-monom-mult*:
dehomogenize x (*punit.monom-mult* c t p) = *punit.monom-mult* c (*except* t $\{x\}$) (*dehomogenize* x p)
by (*simp only*: *times-monomial-left*[*symmetric*] *dehomogenize-times dehomogenize-monomial*)

lemma *poly-deg-dehomogenize-le*: *poly-deg* (*dehomogenize* x p) $\leq$ *poly-deg* p
unfolding *dehomogenize-def dehomo-subst-def*
by (*rule poly-deg-poly-subst-le*) (*simp add*: *poly-deg-monomial deg-pm-single*)

lemma *indets-dehomogenize*: *indets* (*dehomogenize* x p) $\subseteq$ *indets* p $-$ $\{x\}$
for p::($'x \Rightarrow_0$ *nat*) $\Rightarrow_0$ $'a$::*comm-semiring-1*
proof
fix y::$'x$
assume $y \in$ *indets* (*dehomogenize* x p)
then obtain y' **where** $y' \in$ *indets* p **and** $y \in$ *indets* ((*dehomo-subst* x y')::- $\Rightarrow_0$ $'a$)
unfolding *dehomogenize-def* **by** (*rule in-indets-poly-substE*)
from *this*(*2*) **have** $y = y'$ **and** $y' \neq x$
by (*simp-all add*: *dehomo-subst-def indets-monomial split*: *if-split-asm*)
with ‹$y' \in$ *indets* p› **show** $y \in$ *indets* p $-$ $\{x\}$ **by** *simp*
qed

lemma *dehomogenize-id-iff* [*simp*]: *dehomogenize* x p = p $\longleftrightarrow$ $x \notin$ *indets* p
proof
assume *eq*: *dehomogenize* x p = p
from *indets-dehomogenize*[*of* x p] **show** $x \notin$ *indets* p **by** (*auto simp*: *eq*)
next
assume *a*: $x \notin$ *indets* p
show *dehomogenize* x p = p **unfolding** *dehomogenize-def*
proof (*rule poly-subst-id*)
fix y
assume $y \in$ *indets* p
with *a* **have** $y \neq x$ **by** *blast*
thus *dehomo-subst* x y = *monomial 1* (*Poly-Mapping.single* y *1*) **by** (*simp add*: *dehomo-subst-def*)
qed
qed

lemma *dehomogenize-dehomogenize* [*simp*]: *dehomogenize* x (*dehomogenize* x p) = *dehomogenize* x p

proof –
  from *indets-dehomogenize*[*of x p*] **have** $x \notin$ *indets* (*dehomogenize x p*) **by** *blast*
  thus *?thesis* **by** *simp*
qed

lemma *dehomogenize-homogenize* [*simp*]: *dehomogenize x* (*homogenize x p*) = *dehomogenize x p*
proof –
  have *dehomogenize x* (*homogenize x p*) = *sum* (*dehomogenize x*) (*hom-components p*)
    by (*simp add*: *homogenize-alt dehomogenize-sum dehomogenize-monom-mult except-single*)
  also have ... = *dehomogenize x p* **by** (*simp only*: *sum-hom-components flip*: *dehomogenize-sum*)
  finally show *?thesis* **.**
qed

corollary *dehomogenize-homogenize-id*: $x \notin$ *indets p* $\Longrightarrow$ *dehomogenize x* (*homogenize x p*) = *p*
  by *simp*

lemma *range-dehomogenize*: *range* (*dehomogenize x*) = (P[$-$ {x}] :: (- $\Rightarrow_0$ $'a$::*comm-semiring-1*) *set*)
proof (*intro subset-antisym subsetI PolysI-alt range-eqI*)
  fix *p*::- $\Rightarrow_0$ $'a$ **and** *y*
  assume $p \in$ *range* (*dehomogenize x*)
  then obtain *q* **where** *p*: *p* = *dehomogenize x q* **..**
  assume $y \in$ *indets p*
  hence $y \in$ *indets* (*dehomogenize x q*) **by** (*simp only*: *p*)
  with *indets-dehomogenize* **have** $y \in$ *indets q* $-$ {x} **..**
  thus $y \in -$ {x} **by** *simp*
next
  fix *p*::- $\Rightarrow_0$ $'a$
  assume $p \in P[-\ \{x\}]$
  hence $x \notin$ *indets p* **by** (*auto dest*: *PolysD*)
  thus *p* = *dehomogenize x* (*homogenize x p*) **by** (*rule dehomogenize-homogenize-id*[*symmetric*])
qed

lemma *dehomogenize-alt*: *dehomogenize x p* = ($\sum$ *t*∈*keys p*. *monomial* (*lookup p t*) (*except t* {x}))
proof –
  have *dehomogenize x p* = *dehomogenize x* ($\sum$ *t*∈*keys p*. *monomial* (*lookup p t*) *t*)
    by (*simp only*: *poly-mapping-sum-monomials*)
  also have ... = ($\sum$ *t*∈*keys p*. *monomial* (*lookup p t*) (*except t* {x}))
    by (*simp only*: *dehomogenize-sum dehomogenize-monomial*)
  finally show *?thesis* **.**
qed

lemma *keys-dehomogenizeE*:
  assumes $t \in keys\ (dehomogenize\ x\ p)$
  obtains s **where** $s \in keys\ p$ **and** $t = except\ s\ \{x\}$
proof −
  note *assms*
  also have $keys\ (dehomogenize\ x\ p) \subseteq (\bigcup s \in keys\ p.\ keys\ (monomial\ (lookup\ p\ s)\ (except\ s\ \{x\})))$
    unfolding *dehomogenize-alt* **by** (*rule keys-sum-subset*)
  finally obtain s **where** $s \in keys\ p$ **and** $t \in keys\ (monomial\ (lookup\ p\ s)\ (except\ s\ \{x\}))$ **..**
  from *this*(*2*) **have** $t = except\ s\ \{x\}$ **by** (*simp split*: *if-split-asm*)
  with ‹$s \in keys\ p$› **show** *?thesis* **..**
qed

lemma *except-inj-on-keys-homogeneous*:
  assumes *homogeneous p*
  shows $inj\text{-}on\ (\lambda t.\ except\ t\ \{x\})\ (keys\ p)$
proof
  fix $s\ t$
  assume $s \in keys\ p$ **and** $t \in keys\ p$
  from *assms this*(*1*) **have** $deg\text{-}pm\ s = poly\text{-}deg\ p$ **by** (*rule homogeneousD-poly-deg*)
  moreover from *assms* ‹$t \in keys\ p$› **have** $deg\text{-}pm\ t = poly\text{-}deg\ p$ **by** (*rule homogeneousD-poly-deg*)
  ultimately have $deg\text{-}pm\ (Poly\text{-}Mapping.single\ x\ (lookup\ s\ x) + except\ s\ \{x\}) =$
        $deg\text{-}pm\ (Poly\text{-}Mapping.single\ x\ (lookup\ t\ x) + except\ t\ \{x\})$
    by (*simp only*: *flip*: *plus-except*)
  moreover assume *1*: $except\ s\ \{x\} = except\ t\ \{x\}$
  ultimately have *2*: $lookup\ s\ x = lookup\ t\ x$
    by (*simp only*: *deg-pm-plus deg-pm-single*)
  show $s = t$
  proof (*rule poly-mapping-eqI*)
    fix y
    show $lookup\ s\ y = lookup\ t\ y$
    proof (*cases* $y = x$)
      case *True*
      with *2* **show** *?thesis* **by** *simp*
    next
      case *False*
      hence $lookup\ s\ y = lookup\ (except\ s\ \{x\})\ y$ **and** $lookup\ t\ y = lookup\ (except\ t\ \{x\})\ y$
        by (*simp-all add*: *lookup-except*)
      with *1* **show** *?thesis* **by** *simp*
    qed
  qed
qed

lemma *lookup-dehomogenize*:
  assumes *homogeneous p* **and** $t \in keys\ p$
  shows $lookup\ (dehomogenize\ x\ p)\ (except\ t\ \{x\}) = lookup\ p\ t$

proof −
let *?t* = *except t* {*x*}
have *eq*: (∑ *s*∈*keys p* − {*t*}. *lookup* (*monomial* (*lookup p s*) (*except s* {*x*})) *?t*) = *0*
proof (*intro sum.neutral ballI*)
fix *s*
assume *s* ∈ *keys p* − {*t*}
hence *s* ∈ *keys p* **and** *s* ≠ *t* **by** *simp-all*
have *?t* ≠ *except s* {*x*}
proof
from *assms*(*1*) **have** *inj-on* (λ*t*. *except t* {*x*}) (*keys p*) **by** (*rule except-inj-on-keys-homogeneous*)
moreover assume *?t* = *except s* {*x*}
ultimately have *t* = *s* **using** *assms*(*2*) ‹*s* ∈ *keys p*› **by** (*rule inj-onD*)
with ‹*s* ≠ *t*› **show** *False* **by** *simp*
qed
thus *lookup* (*monomial* (*lookup p s*) (*except s* {*x*})) *?t* = *0* **by** (*simp add*: *lookup-single*)
qed
have *lookup* (*dehomogenize x p*) *?t* = (∑ *s*∈*keys p*. *lookup* (*monomial* (*lookup p s*) (*except s* {*x*})) *?t*)
by (*simp only*: *dehomogenize-alt lookup-sum*)
also have ... = *lookup* (*monomial* (*lookup p t*) *?t*) *?t* +
(∑ *s*∈*keys p* − {*t*}. *lookup* (*monomial* (*lookup p s*) (*except s* {*x*})) *?t*)
using *finite-keys assms*(*2*) **by** (*rule sum.remove*)
also have ... = *lookup p t* **by** (*simp add*: *eq*)
finally show *?thesis* **.**
qed

lemma *keys-dehomogenizeI*:
assumes *homogeneous p* **and** *t* ∈ *keys p*
shows *except t* {*x*} ∈ *keys* (*dehomogenize x p*)
proof −
from *assms* **have** *lookup* (*dehomogenize x p*) (*except t* {*x*}) = *lookup p t* **by** (*rule lookup-dehomogenize*)
also from *assms*(*2*) **have** ... ≠ *0* **by** (*simp add*: *in-keys-iff*)
finally show *?thesis* **by** (*simp add*: *in-keys-iff*)
qed

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 (*cases p* = *0*)
case *True*
hence *0* = *poly-deg p* − *poly-deg* (*homogenize x* (*dehomogenize x p*))
and *punit.monom-mult 1* (*Poly-Mapping.single x 0*) (*homogenize x* (*dehomogenize x p*)) = *p*

by *simp-all*
thus *?thesis* **..**
next
case *False*
let *?q* = *dehomogenize x p*
let *?p* = *homogenize x ?q*
define *d* **where** *d* = *poly-deg p* − *poly-deg ?p*
show *?thesis*
proof
have *punit.monom-mult 1* (*Poly-Mapping.single x d*) *?p* =
(∑ *t*∈*keys ?q*. *monomial* (*lookup ?q t*) (*Poly-Mapping.single x* (*d* + (*poly-deg ?q* − *deg-pm t*)) + *t*))
by (*simp add*: *homogenize-def punit.monom-mult-sum-right punit.monom-mult-monomial flip*: *add.assoc single-add*)
also have ... = (∑ *t*∈*keys ?q*. *monomial* (*lookup ?q t*) (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm t*) + *t*))
using *refl*
proof (*rule sum.cong*)
fix *t*
assume *t* ∈ *keys ?q*
have *poly-deg ?p* = *poly-deg ?q*
proof (*rule poly-deg-homogenize*)
from *indets-dehomogenize* **show** *x* ∉ *indets ?q* **by** *fastforce*
qed
hence *d*: *d* = *poly-deg p* − *poly-deg ?q* **by** (*simp only*: *d-def*)
thm *poly-deg-dehomogenize-le*
from ‹*t* ∈ *keys ?q*› **have** *d* + (*poly-deg ?q* − *deg-pm t*) = (*d* + *poly-deg ?q*) − *deg-pm t*
by (*intro add-diff-assoc poly-deg-max-keys*)
also have *d* + *poly-deg ?q* = *poly-deg p* **by** (*simp add*: *d poly-deg-dehomogenize-le*)
finally show *monomial* (*lookup ?q t*) (*Poly-Mapping.single x* (*d* + (*poly-deg ?q* − *deg-pm t*)) + *t*) =
monomial (*lookup ?q t*) (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm t*) + *t*)
by (*simp only*:)
qed
also have ... = (∑ *t*∈(λ*s*. *except s* {*x*}) ‘ *keys p*.
monomial (*lookup ?q t*) (*Poly-Mapping.single x* (*poly-deg p* − *deg-pm t*) + *t*))
proof (*rule sum.mono-neutral-left*)
show *keys* (*dehomogenize x p*) ⊆ (λ*s*. *except s* {*x*}) ‘ *keys p*
proof
fix *t*
assume *t* ∈ *keys* (*dehomogenize x p*)
then obtain *s* **where** *s* ∈ *keys p* **and** *t* = *except s* {*x*} **by** (*rule keys-dehomogenizeE*)
thus *t* ∈ (λ*s*. *except s* {*x*}) ‘ *keys p* **by** (*rule rev-image-eqI*)
qed
qed (*simp-all add*: *in-keys-iff*)
also from *assms* **have** ... = (∑ *t*∈*keys p*. *monomial* (*lookup ?q* (*except t* {*x*}))

(*Poly-Mapping.single x* (*poly-deg p* − *deg-pm* (*except t* {*x*})) + *except t* {*x*}))
by (*intro sum.reindex*[*unfolded comp-def*] *except-inj-on-keys-homogeneous*)
also from *refl* **have** ... = (∑ *t*∈*keys p*. *monomial* (*lookup p t*) *t*)
proof (*rule sum.cong*)
fix *t*
assume *t* ∈ *keys p*
with *assms* **have** *lookup ?q* (*except t* {*x*}) = *lookup p t* **by** (*rule lookup-dehomogenize*)
moreover have *Poly-Mapping.single x* (*poly-deg p* − *deg-pm* (*except t* {*x*})) + *except t* {*x*} = *t*
(**is** *?l* = -)
proof (*rule poly-mapping-eqI*)
fix *y*
show *lookup ?l y* = *lookup t y*
proof (*cases y* = *x*)
case *True*
from *assms* ‹*t* ∈ *keys p*› **have** *deg-pm t* = *poly-deg p* **by** (*rule homogeneousD-poly-deg*)
also have *deg-pm t* = *deg-pm* (*Poly-Mapping.single x* (*lookup t x*) + *except t* {*x*})
by (*simp flip*: *plus-except*)
also have ... = *lookup t x* + *deg-pm* (*except t* {*x*}) **by** (*simp only*: *deg-pm-plus deg-pm-single*)
finally have *poly-deg p* − *deg-pm* (*except t* {*x*}) = *lookup t x* **by** *simp*
thus *?thesis* **by** (*simp add*: *True lookup-add lookup-except lookup-single*)
next
case *False*
thus *?thesis* **by** (*simp add*: *lookup-add lookup-except lookup-single*)
qed
qed
ultimately show *monomial* (*lookup ?q* (*except t* {*x*}))
(*Poly-Mapping.single x* (*poly-deg p* − *deg-pm* (*except t* {*x*})) + *except t* {*x*}) =
monomial (*lookup p t*) *t* **by** (*simp only*:)
qed
also have ... = *p* **by** (*fact poly-mapping-sum-monomials*)
finally show *punit.monom-mult 1* (*Poly-Mapping.single x d*) *?p* = *p* **.**
qed (*simp only*: *d-def*)
qed

lemma *dehomogenize-zeroD*:
assumes *dehomogenize x p* = *0* **and** *homogeneous p*
shows *p* = *0*
proof −
from *assms*(*2*) **obtain** *d*
where *punit.monom-mult 1* (*Poly-Mapping.single x d*) (*homogenize x* (*dehomogenize x p*)) = *p*
by (*rule homogeneous-homogenize-dehomogenize*)
thus *?thesis* **by** (*simp add*: *assms*(*1*))

qed

lemma *dehomogenize-ideal*: *dehomogenize x ‘ ideal F* = *ideal* (*dehomogenize x ‘ F*) ∩ *P*[− {*x*}]
unfolding *range-dehomogenize*[*symmetric*]
using *dehomogenize-plus dehomogenize-times dehomogenize-dehomogenize* **by** (*rule image-ideal-eq-Int*)

corollary *dehomogenize-ideal-subset*: *dehomogenize x ‘ ideal F* ⊆ *ideal* (*dehomogenize x ‘ F*)
by (*simp add*: *dehomogenize-ideal*)

lemma *ideal-dehomogenize*:
assumes *ideal G* = *ideal* (*homogenize x ‘ F*) **and** *F* ⊆ *P*[*UNIV* − {*x*}]
shows *ideal* (*dehomogenize x ‘ G*) = *ideal F*
proof −
have *eq*: *dehomogenize x* (*homogenize x f*) = *f* **if** *f* ∈ *F* **for** *f*
proof (*rule dehomogenize-homogenize-id*)
from *that assms*(*2*) **have** *f* ∈ *P*[*UNIV* − {*x*}] **..**
thus *x* ∉ *indets f* **by** (*auto simp*: *Polys-alt*)
qed
show *?thesis*
proof (*intro Set.equalityI ideal.span-subset-spanI*)
show *dehomogenize x ‘ G* ⊆ *ideal F*
proof
fix *q*
assume *q* ∈ *dehomogenize x ‘ G*
then obtain *g* **where** *g* ∈ *G* **and** *q*: *q* = *dehomogenize x g* **..**
from *this*(*1*) **have** *g* ∈ *ideal G* **by** (*rule ideal.span-base*)
also have . . . = *ideal* (*homogenize x ‘ F*) **by** *fact*
finally have *q* ∈ *dehomogenize x ‘ ideal* (*homogenize x ‘ F*) **using** *q* **by** (*rule rev-image-eqI*)
also have . . . ⊆ *ideal* (*dehomogenize x ‘ homogenize x ‘ F*) **by** (*rule dehomogenize-ideal-subset*)
also have *dehomogenize x ‘ homogenize x ‘ F* = *F*
by (*auto simp*: *eq image-image simp del*: *dehomogenize-homogenize intro*!: *image-eqI*)
finally show *q* ∈ *ideal F* **.**
qed
next
show *F* ⊆ *ideal* (*dehomogenize x ‘ G*)
proof
fix *f*
assume *f* ∈ *F*
hence *homogenize x f* ∈ *homogenize x ‘ F* **by** (*rule imageI*)
also have . . . ⊆ *ideal* (*homogenize x ‘ F*) **by** (*rule ideal.span-superset*)
also from *assms*(*1*) **have** . . . = *ideal G* **by** (*rule sym*)
finally have *dehomogenize x* (*homogenize x f*) ∈ *dehomogenize x ‘ ideal G* **by** (*rule imageI*)

with ‹$f \in F$› **have** $f \in$ *dehomogenize x ‘ ideal G* **by** (*simp only*: *eq*)
also have ... $\subseteq$ *ideal* (*dehomogenize x ‘ G*) **by** (*rule dehomogenize-ideal-subset*)
finally show $f \in$ *ideal* (*dehomogenize x ‘ G*) **.**
qed
qed
qed

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* :: $'x \Rightarrow ('x$ *option* $\Rightarrow_0$ *nat*$) \Rightarrow_0$ *'a::comm-semiring-1*
where *extend-indets-subst x = monomial 1 (Poly-Mapping.single (Some x) 1)*

definition *extend-indets* :: $(('x \Rightarrow_0$ *nat*$) \Rightarrow_0$ $'a) \Rightarrow ('x$ *option* $\Rightarrow_0$ *nat*$) \Rightarrow_0$ *'a::comm-semiring-1*
where *extend-indets = poly-subst extend-indets-subst*

definition *restrict-indets-subst* :: $'x$ *option* $\Rightarrow$ $'x \Rightarrow_0$ *nat*
where *restrict-indets-subst x = (case x of Some y ⇒ Poly-Mapping.single y 1 |
- ⇒ 0)*

definition *restrict-indets* :: $(('x$ *option* $\Rightarrow_0$ *nat*$) \Rightarrow_0$ $'a) \Rightarrow ('x \Rightarrow_0$ *nat*$) \Rightarrow_0$ *'a::comm-semiring-1*
where *restrict-indets = poly-subst (λx. monomial 1 (restrict-indets-subst x))*

definition *restrict-indets-pp* :: $('x$ *option* $\Rightarrow_0$ *nat*$) \Rightarrow ('x \Rightarrow_0$ *nat*$)$
where *restrict-indets-pp t* = ($\sum$ *x∈keys t. lookup t x · restrict-indets-subst x*)

lemma *lookup-extend-indets-subst-aux*:
lookup ($\sum$ *y∈keys t. Poly-Mapping.single (Some y) (lookup t y)) = (λx. case x of
Some y ⇒ lookup t y | - ⇒ 0)*
proof −
have ($\sum$ *x∈keys t. lookup t x when x = y) = lookup t y* **for** *y*
proof (*cases y ∈ keys t*)
case *True*
hence ($\sum$ *x∈keys t. lookup t x when x = y*) = ($\sum$ *x∈insert y (keys t). lookup t
x when x = y*)
by (*simp only*: *insert-absorb*)
also have ... = *lookup t y* + ($\sum$ *x∈keys t − {y}. lookup t x when x = y*)
by (*simp add*: *sum.insert-remove*)
also have ($\sum$ *x∈keys t − {y}. lookup t x when x = y) = 0*
by (*auto simp*: *when-def intro*: *sum.neutral*)
finally show *?thesis* **by** *simp*
next
case *False*
hence ($\sum$ *x∈keys t. lookup t x when x = y) = 0* **by** (*auto simp*: *when-def intro*:
sum.neutral)
with *False* **show** *?thesis* **by** (*simp add*: *in-keys-iff*)

qed
thus *?thesis* **by** (*auto simp*: *lookup-sum lookup-single split*: *option.split*)
qed

lemma *keys-extend-indets-subst-aux*:
keys ($\sum$ *y*∈*keys t. Poly-Mapping.single* (*Some y*) (*lookup t y*)) = *Some* ‘ *keys t*
by (*auto simp*: *lookup-extend-indets-subst-aux simp flip*: *lookup-not-eq-zero-eq-in-keys split*: *option.splits*)

lemma *subst-pp-extend-indets-subst*:
subst-pp extend-indets-subst t = *monomial 1* ($\sum$ *y*∈*keys t. Poly-Mapping.single* (*Some y*) (*lookup t y*))
proof −
have *subst-pp extend-indets-subst t* =
monomial ($\prod$ *y*∈*keys t. 1* ^ *lookup t y*) ($\sum$ *y*∈*keys t. lookup t y* · *Poly-Mapping.single* (*Some y*) *1*)
by (*rule subst-pp-by-monomials*) (*simp only*: *extend-indets-subst-def*)
also have . . . = *monomial 1* ($\sum$ *y*∈*keys t. Poly-Mapping.single* (*Some y*) (*lookup t y*))
by *simp*
finally show *?thesis* **.**
qed

lemma *keys-extend-indets*:
keys (*extend-indets p*) = (λ*t*. $\sum$ *y*∈*keys t. Poly-Mapping.single* (*Some y*) (*lookup t y*)) ‘ *keys p*
proof −
have *keys* (*extend-indets p*) = ($\bigcup$ *t*∈*keys p. keys* (*punit.monom-mult* (*lookup p t*) *0* (*subst-pp extend-indets-subst t*)))
unfolding *extend-indets-def poly-subst-def* **using** *finite-keys*
proof (*rule keys-sum*)
fix *s t* :: *'a* $\Rightarrow_0$ *nat*
assume *s* ≠ *t*
then obtain *x* **where** *lookup s x* ≠ *lookup t x* **by** (*meson poly-mapping-eqI*)
have ($\sum$ *y*∈*keys t. monomial* (*lookup t y*) (*Some y*)) ≠ ($\sum$ *y*∈*keys s. monomial* (*lookup s y*) (*Some y*))
(**is** *?l* ≠ *?r*)
proof
assume *?l* = *?r*
hence *lookup ?l* (*Some x*) = *lookup ?r* (*Some x*) **by** (*simp only*:)
hence *lookup s x* = *lookup t x* **by** (*simp add*: *lookup-extend-indets-subst-aux*)
with ‹*lookup s x* ≠ *lookup t x*› **show** *False* **..**
qed
thus *keys* (*punit.monom-mult* (*lookup p s*) *0* (*subst-pp extend-indets-subst s*)) ∩
keys (*punit.monom-mult* (*lookup p t*) *0* (*subst-pp extend-indets-subst t*)) = {}
by (*simp add*: *subst-pp-extend-indets-subst punit.monom-mult-monomial*)
qed

also have . . . = (λt. ∑ y∈keys t. monomial (lookup t y) (Some y)) ' keys p
by (auto simp: subst-pp-extend-indets-subst punit.monom-mult-monomial split: if-split-asm)
finally show ?thesis **.**
qed

lemma indets-extend-indets: indets (extend-indets p) = Some ' indets (p::- ⇒$_0$ 'a::comm-semiring-1)
proof (rule set-eqI)
fix x
show x ∈ indets (extend-indets p) ⟷ x ∈ Some ' indets p
proof
assume x ∈ indets (extend-indets p)
then obtain y **where** y ∈ indets p **and** x ∈ indets (monomial (1::'a) (Poly-Mapping.single (Some y) 1))
unfolding extend-indets-def extend-indets-subst-def **by** (rule in-indets-poly-substE)
from this(2) indets-monomial-single-subset **have** x ∈ {Some y} **..**
hence x = Some y **by** simp
with ‹y ∈ indets p› **show** x ∈ Some ' indets p **by** (rule rev-image-eqI)
next
assume x ∈ Some ' indets p
then obtain y **where** y ∈ indets p **and** x: x = Some y **..**
from this(1) **obtain** t **where** t ∈ keys p **and** y ∈ keys t **by** (rule in-indetsE)
from this(2) **have** Some y ∈ keys (∑ y∈keys t. Poly-Mapping.single (Some y) (lookup t y))
unfolding keys-extend-indets-subst-aux **by** (rule imageI)
moreover have (∑ y∈keys t. Poly-Mapping.single (Some y) (lookup t y)) ∈ keys (extend-indets p)
unfolding keys-extend-indets **using** ‹t ∈ keys p› **by** (rule imageI)
ultimately show x ∈ indets (extend-indets p) **unfolding** x **by** (rule in-indetsI)
qed
qed

lemma poly-deg-extend-indets [simp]: poly-deg (extend-indets p) = poly-deg p
proof −
have eq: deg-pm ((∑ y∈keys t. Poly-Mapping.single (Some y) (lookup t y))) = deg-pm t
for t::'a ⇒$_0$ nat
proof −
have deg-pm ((∑ y∈keys t. Poly-Mapping.single (Some y) (lookup t y))) = (∑ y∈keys t. lookup t y)
by (simp add: deg-pm-sum deg-pm-single)
also from subset-refl finite-keys **have** . . . = deg-pm t **by** (rule deg-pm-superset[symmetric])
finally show ?thesis **.**
qed
show ?thesis
proof (rule antisym)
show poly-deg (extend-indets p) ≤ poly-deg p
proof (rule poly-deg-leI)

fix *t*
assume *t* ∈ *keys* (*extend-indets p*)
then obtain *s* **where** *s* ∈ *keys p* **and** *t* = (∑ *y*∈*keys s*. *Poly-Mapping.single* (*Some y*) (*lookup s y*))
unfolding *keys-extend-indets* **..**
from *this*(*2*) **have** *deg-pm t* = *deg-pm s* **by** (*simp only*: *eq*)
also from ‹*s* ∈ *keys p*› **have** ... ≤ *poly-deg p* **by** (*rule poly-deg-max-keys*)
finally show *deg-pm t* ≤ *poly-deg p* **.**
qed
next
show *poly-deg p* ≤ *poly-deg* (*extend-indets p*)
proof (*rule poly-deg-leI*)
fix *t*
assume *t* ∈ *keys p*
hence ∗: (∑ *y*∈*keys t*. *Poly-Mapping.single* (*Some y*) (*lookup t y*)) ∈ *keys* (*extend-indets p*)
unfolding *keys-extend-indets* **by** (*rule imageI*)
have *deg-pm t* = *deg-pm* (∑ *y*∈*keys t*. *Poly-Mapping.single* (*Some y*) (*lookup t y*))
by (*simp only*: *eq*)
also from ∗ **have** ... ≤ *poly-deg* (*extend-indets p*) **by** (*rule poly-deg-max-keys*)
finally show *deg-pm t* ≤ *poly-deg* (*extend-indets p*) **.**
qed
qed
qed

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*::- ⇒$_0$ -::*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* ^ *n*) = *extend-indets p* ^ *n*
and *extend-indets-sum*: *extend-indets* (*sum f A*) = (∑ *a*∈*A*. *extend-indets* (*f a*))
and *extend-indets-prod*: *extend-indets* (*prod f A*) = (∏ *a*∈*A*. *extend-indets* (*f a*))
by (*simp-all add*: *extend-indets-def poly-subst-monomial poly-subst-plus poly-subst-uminus poly-subst-minus poly-subst-times poly-subst-power poly-subst-sum poly-subst-prod*)

lemma *extend-indets-zero-iff* [*simp*]: *extend-indets p* = *0* ⟷ *p* = *0*
by (*metis* (*no-types*, *lifting*) *imageE imageI keys-extend-indets lookup-zero*

not-in-keys-iff-lookup-eq-zero poly-deg-extend-indets poly-deg-zero poly-deg-zero-imp-monomial)

lemma *extend-indets-inject*:
assumes *extend-indets p* = *extend-indets* (*q*::- ⇒$_0$ -::*comm-ring-1*)
shows *p* = *q*
proof −
from *assms* **have** *extend-indets* (*p* − *q*) = *0* **by** (*simp add*: *extend-indets-minus*)
thus *?thesis* **by** *simp*
qed

corollary *inj-extend-indets*: *inj* (*extend-indets*::- ⇒ - ⇒$_0$ -::*comm-ring-1*)
using *extend-indets-inject* **by** (*intro injI*)

lemma *poly-subst-extend-indets*: *poly-subst f* (*extend-indets p*) = *poly-subst* (*f* ∘ *Some*) *p*
by (*simp add*: *extend-indets-def poly-subst-poly-subst extend-indets-subst-def poly-subst-monomial subst-pp-single o-def*)

lemma *poly-eval-extend-indets*: *poly-eval a* (*extend-indets p*) = *poly-eval* (*a* ∘ *Some*) *p*
proof −
have *eq*: *poly-eval a* (*extend-indets* (*monomial c t*)) = *poly-eval* (λ*x*. *a* (*Some x*)) (*monomial c t*)
for *c t*
by (*simp add*: *extend-indets-monomial poly-eval-times poly-eval-monomial poly-eval-prod poly-eval-power*
subst-pp-def extend-indets-subst-def flip: *times-monomial-left*)
show *?thesis*
by (*induct p rule*: *poly-mapping-plus-induct*) (*simp-all add*: *extend-indets-plus poly-eval-plus eq*)
qed

lemma *lookup-restrict-indets-pp*: *lookup* (*restrict-indets-pp t*) = (λ*x*. *lookup t* (*Some x*))
proof −
let *?f* = λ*z x*. *lookup t x* ∗ *lookup* (*case x of None* ⇒ *0* | *Some y* ⇒ *Poly-Mapping.single y 1*) *z*
have *sum* (*?f z*) (*keys t*) = *lookup t* (*Some z*) **for** *z*
proof (*cases Some z* ∈ *keys t*)
case *True*
hence *sum* (*?f z*) (*keys t*) = *sum* (*?f z*) (*insert* (*Some z*) (*keys t*))
by (*simp only*: *insert-absorb*)
also have . . . = *lookup t* (*Some z*) + *sum* (*?f z*) (*keys t* − {*Some z*})
by (*simp add*: *sum.insert-remove*)
also have *sum* (*?f z*) (*keys t* − {*Some z*}) = *0*
by (*auto simp*: *when-def lookup-single intro*: *sum.neutral split*: *option.splits*)
finally show *?thesis* **by** *simp*
next
case *False*

hence *sum* (*?f z*) (*keys t*) = *0*
by (*auto simp*: *when-def lookup-single intro*: *sum.neutral split*: *option.splits*)
with *False* **show** *?thesis* **by** (*simp add*: *in-keys-iff*)
qed
thus *?thesis* **by** (*auto simp*: *restrict-indets-pp-def restrict-indets-subst-def lookup-sum*)
qed

lemma *keys-restrict-indets-pp*: *keys* (*restrict-indets-pp t*) = *the* ‘ (*keys t* − {*None*})
proof (*rule set-eqI*)
fix *x*
show *x* ∈ *keys* (*restrict-indets-pp t*) ⟷ *x* ∈ *the* ‘ (*keys t* − {*None*})
proof
assume *x* ∈ *keys* (*restrict-indets-pp t*)
hence *Some x* ∈ *keys t* **by** (*simp add*: *lookup-restrict-indets-pp flip*: *lookup-not-eq-zero-eq-in-keys*)
hence *Some x* ∈ *keys t* − {*None*} **by** *blast*
moreover have *x* = *the* (*Some x*) **by** *simp*
ultimately show *x* ∈ *the* ‘ (*keys t* − {*None*}) **by** (*rule rev-image-eqI*)
next
assume *x* ∈ *the* ‘ (*keys t* − {*None*})
then obtain *y* **where** *y* ∈ *keys t* − {*None*} **and** *x* = *the y* **..**
hence *Some x* ∈ *keys t* **by** *auto*
thus *x* ∈ *keys* (*restrict-indets-pp t*)
by (*simp add*: *lookup-restrict-indets-pp flip*: *lookup-not-eq-zero-eq-in-keys*)
qed
qed

lemma *subst-pp-restrict-indets-subst*:
subst-pp (λ*x*. *monomial 1* (*restrict-indets-subst x*)) *t* = *monomial 1* (*restrict-indets-pp t*)
by (*simp add*: *subst-pp-def monomial-power-map-scale restrict-indets-pp-def flip*: *punit.monomial-prod-sum*)

lemma *restrict-indets-pp-zero* [*simp*]: *restrict-indets-pp 0* = *0*
by (*simp add*: *restrict-indets-pp-def*)

lemma *restrict-indets-pp-plus*: *restrict-indets-pp* (*s* + *t*) = *restrict-indets-pp s* + *restrict-indets-pp t*
by (*rule poly-mapping-eqI*) (*simp add*: *lookup-add lookup-restrict-indets-pp*)

lemma *restrict-indets-pp-except-None* [*simp*]:
restrict-indets-pp (*except t* {*None*}) = *restrict-indets-pp t*
by (*rule poly-mapping-eqI*) (*simp add*: *lookup-add lookup-restrict-indets-pp lookup-except*)

lemma *deg-pm-restrict-indets-pp*: *deg-pm* (*restrict-indets-pp t*) + *lookup t None* = *deg-pm t*
proof −
have *deg-pm t* = *sum* (*lookup t*) (*insert None* (*keys t*)) **by** (*rule deg-pm-superset*) *auto*
also from *finite-keys* **have** ... = *lookup t None* + *sum* (*lookup t*) (*keys t* −

{*None*})
by (*rule sum.insert-remove*)
also have *sum* (*lookup t*) (*keys t* − {*None*}) = ($\sum$ *x*∈*keys t*. *lookup t x* ∗ *deg-pm* (*restrict-indets-subst x*))
by (*intro sum.mono-neutral-cong-left*) (*auto simp*: *restrict-indets-subst-def deg-pm-single*)
also have ... = *deg-pm* (*restrict-indets-pp t*)
by (*simp only*: *restrict-indets-pp-def deg-pm-sum deg-pm-map-scale*)
finally show *?thesis* **by** *simp*
qed

lemma *keys-restrict-indets-subset*: *keys* (*restrict-indets p*) ⊆ *restrict-indets-pp* ' *keys p*
proof
fix *t*
assume *t* ∈ *keys* (*restrict-indets p*)
also have ... = *keys* ($\sum$ *t*∈*keys p*. *monomial* (*lookup p t*) (*restrict-indets-pp t*))
by (*simp add*: *restrict-indets-def poly-subst-def subst-pp-restrict-indets-subst punit.monom-mult-monomial*)
also have ... ⊆ ($\bigcup$ *t*∈*keys p*. *keys* (*monomial* (*lookup p t*) (*restrict-indets-pp t*)))
by (*rule keys-sum-subset*)
also have ... = *restrict-indets-pp* ' *keys p* **by** (*auto split*: *if-split-asm*)
finally show *t* ∈ *restrict-indets-pp* ' *keys p* **.**
qed

lemma *keys-restrict-indets*:
assumes *None* ∉ *indets p*
shows *keys* (*restrict-indets p*) = *restrict-indets-pp* ' *keys p*
proof −
have *keys* (*restrict-indets p*) = *keys* ($\sum$ *t*∈*keys p*. *monomial* (*lookup p t*) (*restrict-indets-pp t*))
by (*simp add*: *restrict-indets-def poly-subst-def subst-pp-restrict-indets-subst punit.monom-mult-monomial*)
also from *finite-keys* **have** ... = ($\bigcup$ *t*∈*keys p*. *keys* (*monomial* (*lookup p t*) (*restrict-indets-pp t*)))
proof (*rule keys-sum*)
fix *s t*
assume *s* ∈ *keys p*
hence *keys s* ⊆ *indets p* **by** (*rule keys-subset-indets*)
with *assms* **have** *None* ∉ *keys s* **by** *blast*
assume *t* ∈ *keys p*
hence *keys t* ⊆ *indets p* **by** (*rule keys-subset-indets*)
with *assms* **have** *None* ∉ *keys t* **by** *blast*
assume *s* ≠ *t*
then obtain *x* **where** *neq*: *lookup s x* ≠ *lookup t x* **by** (*meson poly-mapping-eqI*)
have *x* ≠ *None*
proof
assume *x* = *None*
with ‹*None* ∉ *keys s*› **and** ‹*None* ∉ *keys t*› **have** *x* ∉ *keys s* **and** *x* ∉ *keys t* **by** *blast+*

with *neq* **show** *False* **by** (*simp add*: *in-keys-iff*)
qed
then obtain *y* **where** *x*: *x* = *Some y* **by** *blast*
have *restrict-indets-pp t* ≠ *restrict-indets-pp s*
proof
assume *restrict-indets-pp t* = *restrict-indets-pp s*
hence *lookup* (*restrict-indets-pp t*) *y* = *lookup* (*restrict-indets-pp s*) *y* **by** (*simp only*:)
hence *lookup s x* = *lookup t x* **by** (*simp add*: *x lookup-restrict-indets-pp*)
with *neq* **show** *False* **..**
qed
thus *keys* (*monomial* (*lookup p s*) (*restrict-indets-pp s*)) ∩
keys (*monomial* (*lookup p t*) (*restrict-indets-pp t*)) = {}
by (*simp add*: *subst-pp-extend-indets-subst*)
qed
also have ... = *restrict-indets-pp* ' *keys p* **by** (*auto split*: *if-split-asm*)
finally show *?thesis* **.**
qed

lemma *indets-restrict-indets-subset*: *indets* (*restrict-indets p*) ⊆ *the* ' (*indets p* − {*None*})
proof
fix *x*
assume *x* ∈ *indets* (*restrict-indets p*)
then obtain *t* **where** *t* ∈ *keys* (*restrict-indets p*) **and** *x* ∈ *keys t* **by** (*rule in-indetsE*)
from *this*(*1*) *keys-restrict-indets-subset* **have** *t* ∈ *restrict-indets-pp* ' *keys p* **..**
then obtain *s* **where** *s* ∈ *keys p* **and** *t* = *restrict-indets-pp s* **..**
from ‹*x* ∈ *keys t*› *this*(*2*) **have** *x* ∈ *the* ' (*keys s* − {*None*}) **by** (*simp only*: *keys-restrict-indets-pp*)
also from ‹*s* ∈ *keys p*› **have** ... ⊆ *the* ' (*indets p* − {*None*})
by (*intro image-mono Diff-mono keys-subset-indets subset-refl*)
finally show *x* ∈ *the* ' (*indets p* − {*None*}) **.**
qed

lemma *poly-deg-restrict-indets-le*: *poly-deg* (*restrict-indets p*) ≤ *poly-deg p*
proof (*rule poly-deg-leI*)
fix *t*
assume *t* ∈ *keys* (*restrict-indets p*)
hence *t* ∈ *restrict-indets-pp* ' *keys p* **using** *keys-restrict-indets-subset* **..**
then obtain *s* **where** *s* ∈ *keys p* **and** *t* = *restrict-indets-pp s* **..**
from *this*(*2*) **have** *deg-pm t* + *lookup s None* = *deg-pm s*
by (*simp only*: *deg-pm-restrict-indets-pp*)
also from ‹*s* ∈ *keys p*› **have** ... ≤ *poly-deg p* **by** (*rule poly-deg-max-keys*)
finally show *deg-pm t* ≤ *poly-deg p* **by** *simp*
qed

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*::- $\Rightarrow_0$ -::*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*))
by (*simp-all add*: *restrict-indets-def poly-subst-monomial poly-subst-plus poly-subst-uminus poly-subst-minus poly-subst-times poly-subst-power poly-subst-sum poly-subst-prod subst-pp-restrict-indets-subst punit.monom-mult-monomial*)

lemma *restrict-extend-indets* [*simp*]: *restrict-indets* (*extend-indets p*) = *p*
unfolding *extend-indets-def restrict-indets-def poly-subst-poly-subst*
by (*rule poly-subst-id*)
(*simp add*: *extend-indets-subst-def restrict-indets-subst-def poly-subst-monomial subst-pp-single*)

lemma *extend-restrict-indets*:
assumes *None* ∉ *indets p*
shows *extend-indets* (*restrict-indets p*) = *p*
unfolding *extend-indets-def restrict-indets-def poly-subst-poly-subst*
proof (*rule poly-subst-id*)
fix *x*
assume *x* ∈ *indets p*
with *assms* **have** *x* ≠ *None* **by** *meson*
then obtain *y* **where** *x*: *x* = *Some y* **by** *blast*
thus *poly-subst extend-indets-subst* (*monomial 1* (*restrict-indets-subst x*)) = *monomial 1* (*Poly-Mapping.single x 1*)
by (*simp add*: *extend-indets-subst-def restrict-indets-subst-def poly-subst-monomial subst-pp-single*)
qed

lemma *restrict-indets-dehomogenize* [*simp*]: *restrict-indets* (*dehomogenize None p*) = *restrict-indets p*
proof −
have *eq*: *poly-subst* (λ*x*. (*monomial 1* (*restrict-indets-subst x*))) (*dehomo-subst None y*) = *monomial 1* (*restrict-indets-subst y*) **for** *y*::′*x option*
by (*auto simp*: *restrict-indets-subst-def dehomo-subst-def poly-subst-monomial subst-pp-single*)

show *?thesis* **by** (*simp only*: *dehomogenize-def restrict-indets-def poly-subst-poly-subst eq*)
qed

corollary *restrict-indets-comp-dehomogenize*: *restrict-indets* ∘ *dehomogenize None* = *restrict-indets*
by (*rule ext*) (*simp only*: *o-def restrict-indets-dehomogenize*)

corollary *extend-restrict-indets-eq-dehomogenize*:
extend-indets (*restrict-indets p*) = *dehomogenize None p*
proof −
have *extend-indets* (*restrict-indets p*) = *extend-indets* (*restrict-indets* (*dehomogenize None p*))
by *simp*
also have . . . = *dehomogenize None p*
proof (*intro extend-restrict-indets notI*)
assume *None* ∈ *indets* (*dehomogenize None p*)
hence *None* ∈ *indets p* − {*None*} **using** *indets-dehomogenize* **..**
thus *False* **by** *simp*
qed
finally show *?thesis* **.**
qed

corollary *extend-indets-comp-restrict-indets*: *extend-indets* ∘ *restrict-indets* = *dehomogenize None*
by (*rule ext*) (*simp only*: *o-def extend-restrict-indets-eq-dehomogenize*)

lemma *restrict-homogenize-extend-indets* [*simp*]:
restrict-indets (*homogenize None* (*extend-indets p*)) = *p*
proof −
have *restrict-indets* (*homogenize None* (*extend-indets p*)) =
restrict-indets (*dehomogenize None* (*homogenize None* (*extend-indets p*)))
by (*simp only*: *restrict-indets-dehomogenize*)
also have . . . = *restrict-indets* (*dehomogenize None* (*extend-indets p*))
by (*simp only*: *dehomogenize-homogenize*)
also have . . . = *p* **by** *simp*
finally show *?thesis* **.**
qed

lemma *dehomogenize-extend-indets* [*simp*]: *dehomogenize None* (*extend-indets p*) = *extend-indets p*
by (*simp add*: *indets-extend-indets*)

lemma *restrict-indets-ideal*: *restrict-indets* ‘ *ideal F* = *ideal* (*restrict-indets* ‘ *F*)
using *restrict-indets-plus restrict-indets-times*
proof (*rule image-ideal-eq-surj*)
from *restrict-extend-indets* **show** *surj restrict-indets* **by** (*rule surjI*)
qed

lemma *ideal-restrict-indets*:
ideal G = *ideal* (*homogenize None* ‘ *extend-indets* ‘ *F*) ⟹ *ideal* (*restrict-indets* ‘ *G*) = *ideal F*
by (*simp flip*: *restrict-indets-ideal*) (*simp add*: *restrict-indets-ideal image-image*)

lemma *extend-indets-ideal*: *extend-indets* ‘ *ideal F* = *ideal* (*extend-indets* ‘ *F*) ∩ *P*[− {*None*}]
proof −
have *extend-indets* ‘ *ideal F* = *extend-indets* ‘ *restrict-indets* ‘ *ideal* (*extend-indets* ‘ *F*)
by (*simp add*: *restrict-indets-ideal image-image*)
also have ... = *ideal* (*extend-indets* ‘ *F*) ∩ *P*[− {*None*}]
by (*simp add*: *extend-restrict-indets-eq-dehomogenize dehomogenize-ideal image-image*)
finally show *?thesis* **.**
qed

corollary *extend-indets-ideal-subset*: *extend-indets* ‘ *ideal F* ⊆ *ideal* (*extend-indets* ‘ *F*)
by (*simp add*: *extend-indets-ideal*)

17.8 Canonical Isomorphisms between $P[X,Y]$ and $P[X][Y]$: *focus* and *flatten*

definition *focus* :: *'x set* ⇒ ((*'x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'a*) ⇒ ((*'x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ (*'x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'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* $\Rightarrow_0$ *'a* $\Rightarrow_0$ *'b*) ⇒ (*'a*::*comm-powerprod* $\Rightarrow_0$ *'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*)))
unfolding *focus-def* **using** *assms* **by** (*rule sum.mono-neutral-left*) (*simp add*: *in-keys-iff*)

lemma *keys-focus*: *keys* (*focus X p*) = (λ*t*. *except t* (− *X*)) ‘ *keys p*
proof
have *keys* (*focus X p*) ⊆ (⋃ *t*∈*keys p*. *keys* (*monomial* (*monomial* (*lookup p t*) (*except t X*)) (*except t* (− *X*))))
unfolding *focus-def* **by** (*rule keys-sum-subset*)
also have ... ⊆ (⋃ *t*∈*keys p*. {*except t* (− *X*)}) **by** (*intro UN-mono subset-refl*) *simp*
also have ... = (λ*t*. *except t* (− *X*)) ‘ *keys p* **by** (*rule UNION-singleton-eq-range*)
finally show *keys* (*focus X p*) ⊆ (λ*t*. *except t* (− *X*)) ‘ *keys p* **.**
next

{
fix *s*
assume *s* ∈ *keys p*
have *lookup* (*focus X p*) (*except s* (− *X*)) =
(∑ *t*∈*keys p*. *monomial* (*lookup p t*) (*except t X*) *when except t* (− *X*) = *except s* (− *X*))
(is - = *?p*)
by (*simp only*: *focus-def lookup-sum lookup-single*)
also have ... ≠ *0*
proof
have *lookup ?p* (*except s X*) =
(∑ *t*∈*keys p*. *lookup p t when except t X* = *except s X* ∧ *except t* (− *X*) = *except s* (− *X*))
by (*simp add*: *lookup-sum lookup-single when-def if-distrib if-distribR*)
(*metis* (*no-types*, *opaque-lifting*) *lookup-single-eq lookup-single-not-eq lookup-zero*)
also have ... = (∑ *t*∈{*s*}. *lookup p t*)
proof (*intro sum.mono-neutral-cong-right ballI*)
fix *t*
assume *t* ∈ *keys p* − {*s*}
hence *t* ≠ *s* by *simp*
hence *except t X* + *except t* (− *X*) ≠ *except s X* + *except s* (− *X*)
by (*simp flip*: *except-decomp*)
thus (*lookup p t when except t X* = *except s X* ∧ *except t* (− *X*) = *except s* (− *X*)) = *0*
by (*auto simp*: *when-def*)
next
from ‹*s* ∈ *keys p*› show {*s*} ⊆ *keys p* by *simp*
qed *simp-all*
also from ‹*s* ∈ *keys p*› have ... ≠ *0* by (*simp add*: *in-keys-iff*)
finally have *except s X* ∈ *keys ?p* by (*simp add*: *in-keys-iff*)
moreover assume *?p* = *0*
ultimately show *False* by *simp*
qed
finally have *except s* (− *X*) ∈ *keys* (*focus X p*) by (*simp add*: *in-keys-iff*)
}
thus (λ*t*. *except t* (− *X*)) ‘ *keys p* ⊆ *keys* (*focus X p*) by *blast*
qed

lemma *keys-coeffs-focus-subset*:
assumes *c* ∈ *range* (*lookup* (*focus X p*))
shows *keys c* ⊆ (λ*t*. *except t X*) ‘ *keys p*
proof −
from *assms* obtain *s* where *c* = *lookup* (*focus X p*) *s* ..
hence *keys c* = *keys* (*lookup* (*focus X p*) *s*) by (*simp only*:)
also have ... ⊆ (⋃ *t*∈*keys p*. *keys* (*lookup* (*monomial* (*monomial* (*lookup p t*) (*except t X*)) (*except t* (− *X*))) *s*))
unfolding *focus-def lookup-sum* by (*rule keys-sum-subset*)
also from *subset-refl* have ... ⊆ (⋃ *t*∈*keys p*. {*except t X*})

by (*rule UN-mono*) (*simp add*: *lookup-single when-def*)
also have ... = (λt. *except t X*) ‘ *keys p* **by** (*rule UNION-singleton-eq-range*)
finally show *?thesis* **.**
qed

lemma *focus-in-Polys'*:
assumes $p \in P[Y]$
shows *focus X p* $\in P[Y \cap X]$
proof (*intro PolysI subsetI*)
fix *t*
assume $t \in$ *keys* (*focus X p*)
then obtain *s* **where** $s \in$ *keys p* **and** *t*: $t =$ *except s* ($- X$) **unfolding** *keys-focus* **..**
note *this*(*1*)
also from *assms* **have** *keys p* $\subseteq .[Y]$ **by** (*rule PolysD*)
finally have *keys s* $\subseteq Y$ **by** (*rule PPsD*)
hence *keys t* $\subseteq Y \cap X$ **by** (*simp add*: *t keys-except le-infI1*)
thus $t \in .[Y \cap X]$ **by** (*rule PPsI*)
qed

corollary *focus-in-Polys*: *focus X p* $\in P[X]$
proof $-$
have $p \in P[UNIV]$ **by** *simp*
hence *focus X p* $\in P[UNIV \cap X]$ **by** (*rule focus-in-Polys'*)
thus *?thesis* **by** *simp*
qed

lemma *focus-coeffs-subset-Polys'*:
assumes $p \in P[Y]$
shows *range* (*lookup* (*focus X p*)) $\subseteq P[Y - X]$
proof (*intro subsetI PolysI*)
fix *c t*
assume $c \in$ *range* (*lookup* (*focus X p*))
hence *keys c* $\subseteq$ (λt. *except t X*) ‘ *keys p* **by** (*rule keys-coeffs-focus-subset*)
moreover assume $t \in$ *keys c*
ultimately have $t \in$ (λt. *except t X*) ‘ *keys p* **..**
then obtain *s* **where** $s \in$ *keys p* **and** *t*: $t =$ *except s X* **..**
note *this*(*1*)
also from *assms* **have** *keys p* $\subseteq .[Y]$ **by** (*rule PolysD*)
finally have *keys s* $\subseteq Y$ **by** (*rule PPsD*)
hence *keys t* $\subseteq Y - X$ **by** (*simp add*: *t keys-except Diff-mono*)
thus $t \in .[Y - X]$ **by** (*rule PPsI*)
qed

corollary *focus-coeffs-subset-Polys*: *range* (*lookup* (*focus X p*)) $\subseteq P[- X]$
proof $-$
have $p \in P[UNIV]$ **by** *simp*
hence *range* (*lookup* (*focus X p*)) $\subseteq P[UNIV - X]$ **by** (*rule focus-coeffs-subset-Polys'*)
thus *?thesis* **by** (*simp only*: *Compl-eq-Diff-UNIV*)

qed

corollary *lookup-focus-in-Polys*: *lookup* (*focus X p*) *t* ∈ *P*[− *X*]
  using *focus-coeffs-subset-Polys* **by** *blast*

lemma *focus-zero* [*simp*]: *focus X 0* = *0*
  by (*simp add*: *focus-def*)

lemma *focus-eq-zero-iff* [*iff*]: *focus X p* = *0* ⟷ *p* = *0*
  by (*simp only*: *keys-focus flip*: *keys-eq-empty-iff*) *simp*

lemma *focus-one* [*simp*]: *focus X 1* = *1*
  by (*simp add*: *focus-def*)

lemma *focus-monomial*: *focus X* (*monomial c t*) = *monomial* (*monomial c* (*except t X*)) (*except t* (− *X*))
  by (*simp add*: *focus-def*)

lemma *focus-uminus* [*simp*]: *focus X* (− *p*) = − *focus X p*
  by (*simp add*: *focus-def keys-uminus single-uminus sum-negf*)

lemma *focus-plus*: *focus X* (*p* + *q*) = *focus X p* + *focus X q*
proof −
  have *finite* (*keys p* ∪ *keys q*) **by** *simp*
  moreover have *keys* (*p* + *q*) ⊆ *keys p* ∪ *keys q* **by** (*rule Poly-Mapping.keys-add*)
  ultimately show *?thesis*
    by (*simp add*: *focus-superset*[**where** *A*=*keys p* ∪ *keys q*] *lookup-add single-add sum.distrib*)
qed

lemma *focus-minus*: *focus X* (*p* − *q*) = *focus X p* − *focus X* (*q*::- ⇒$_0$ -::*ab-group-add*)
  by (*simp only*: *diff-conv-add-uminus focus-plus focus-uminus*)

lemma *focus-times*: *focus X* (*p* ∗ *q*) = *focus X p* ∗ *focus X q*
proof −
  have *eq*: *focus X* (*monomial c s* ∗ *q*) = *focus X* (*monomial c s*) ∗ *focus X q* **for** *c s*
  proof −
    have *focus X* (*monomial c s* ∗ *q*) = *focus X* (*punit.monom-mult c s q*)
      by (*simp only*: *times-monomial-left*)
    also have . . . = (∑ *t*∈(+) *s* ‘ *keys q*. *monomial* (*monomial* (*lookup* (*punit.monom-mult c s q*) *t*)
                    (*except t X*)) (*except t* (− *X*)))
     by (*rule focus-superset*) (*simp-all add*: *punit.keys-monom-mult-subset*[*simplified*])
    also have . . . = (∑ *t*∈*keys q*. ((λ*t*. *monomial* (*monomial* (*lookup* (*punit.monom-mult c s q*) *t*)
                    (*except t X*)) (*except t* (− *X*))) ∘ ((+) *s*)) *t*)
      by (*rule sum.reindex*) *simp*
    also have . . . = *monomial* (*monomial c* (*except s X*)) (*except s* (− *X*)) ∗

($\sum$ *t*∈*keys q*. *monomial* (*monomial* (*lookup q t*) (*except t X*)) (*except t* (− *X*)))
by (*simp add*: *o-def punit.lookup-monom-mult except-plus times-monomial-monomial sum-distrib-left*)
also have ... = *focus X* (*monomial c s*) ∗ *focus X q*
by (*simp only*: *focus-monomial focus-def*[**where** *p*=*q*])
finally show *?thesis* **.**
qed
show *?thesis* **by** (*induct p rule*: *poly-mapping-plus-induct*) (*simp-all add*: *ring-distribs focus-plus eq*)
qed

lemma *focus-sum*: *focus X* (*sum f I*) = ($\sum$ *i*∈*I*. *focus X* (*f i*))
by (*induct I rule*: *infinite-finite-induct*) (*simp-all add*: *focus-plus*)

lemma *focus-prod*: *focus X* (*prod f I*) = ($\prod$ *i*∈*I*. *focus X* (*f i*))
by (*induct I rule*: *infinite-finite-induct*) (*simp-all add*: *focus-times*)

lemma *focus-power* [*simp*]: *focus X* (*f* ^ *m*) = *focus X f* ^ *m*
by (*induct m*) (*simp-all add*: *focus-times*)

lemma *focus-Polys*:
assumes *p* ∈ *P*[*X*]
shows *focus X p* = ($\sum$ *t*∈*keys p*. *monomial* (*monomial* (*lookup p t*) *0*) *t*)
unfolding *focus-def*
proof (*rule sum.cong*)
fix *t*
assume *t* ∈ *keys p*
also from *assms* **have** ... ⊆ .[*X*] **by** (*rule PolysD*)
finally have *keys t* ⊆ *X* **by** (*rule PPsD*)
hence *except t X* = *0* **and** *except t* (− *X*) = *t* **by** (*rule except-eq-zeroI*, *auto simp*: *except-id-iff*)
thus *monomial* (*monomial* (*lookup p t*) (*except t X*)) (*except t* (− *X*)) =
monomial (*monomial* (*lookup p t*) *0*) *t* **by** *simp*
qed (*fact refl*)

corollary *lookup-focus-Polys*: *p* ∈ *P*[*X*] ⟹ *lookup* (*focus X p*) *t* = *monomial* (*lookup p t*) *0*
by (*simp add*: *focus-Polys lookup-sum lookup-single when-def in-keys-iff*)

lemma *focus-Polys-Compl*:
assumes *p* ∈ *P*[− *X*]
shows *focus X p* = *monomial p 0*
proof −
have *focus X p* = ($\sum$ *t*∈*keys p*. *monomial* (*monomial* (*lookup p t*) *t*) *0*) **unfolding** *focus-def*
proof (*rule sum.cong*)
fix *t*
assume *t* ∈ *keys p*

also from *assms* **have** ... ⊆ .[− X] **by** (*rule PolysD*)
finally have *keys t* ⊆ − X **by** (*rule PPsD*)
hence *except t* (− X) = *0* **and** *except t X* = *t* **by** (*rule except-eq-zeroI*, *auto simp*: *except-id-iff*)
thus *monomial* (*monomial* (*lookup p t*) (*except t X*)) (*except t* (− X)) = *monomial* (*monomial* (*lookup p t*) *t*) *0* **by** *simp*
qed (*fact refl*)
also have ... = *monomial* (∑ *t*∈*keys p*. *monomial* (*lookup p t*) *t*) *0* **by** (*simp only*: *monomial-sum*)
also have ... = *monomial p 0* **by** (*simp only*: *poly-mapping-sum-monomials*)
finally show *?thesis* **.**
qed

corollary *focus-empty* [*simp*]: *focus* {} *p* = *monomial p 0*
by (*rule focus-Polys-Compl*) *simp*

lemma *focus-Int*:
assumes *p* ∈ *P*[*Y*]
shows *focus* (*X* ∩ *Y*) *p* = *focus X p*
unfolding *focus-def* **using** *refl*
proof (*rule sum.cong*)
fix *t*
assume *t* ∈ *keys p*
also from *assms* **have** ... ⊆ .[*Y*] **by** (*rule PolysD*)
finally have *keys t* ⊆ *Y* **by** (*rule PPsD*)
hence *keys t* ⊆ *X* ∪ *Y* **by** *blast*
hence *except t* (*X* ∩ *Y*) = *except t X* + *except t Y* **by** (*rule except-Int*)
also from ‹*keys t* ⊆ *Y*› **have** *except t Y* = *0* **by** (*rule except-eq-zeroI*)
finally have *eq*: *except t* (*X* ∩ *Y*) = *except t X* **by** *simp*
have *except t* (− (*X* ∩ *Y*)) = *except* (*except t* (− *Y*)) (− *X*) **by** (*simp add*: *except-except Un-commute*)
also from ‹*keys t* ⊆ *Y*› **have** *except t* (− *Y*) = *t* **by** (*auto simp*: *except-id-iff*)
finally show *monomial* (*monomial* (*lookup p t*) (*except t* (*X* ∩ *Y*))) (*except t* (− (*X* ∩ *Y*))) = *monomial* (*monomial* (*lookup p t*) (*except t X*)) (*except t* (− *X*)) **by** (*simp only*: *eq*)
qed

lemma *range-focusD*:
assumes *p* ∈ *range* (*focus X*)
shows *p* ∈ *P*[*X*] **and** *range* (*lookup p*) ⊆ *P*[− *X*] **and** *lookup p t* ∈ *P*[− *X*]
using *assms* **by** (*auto intro*: *focus-in-Polys lookup-focus-in-Polys*)

lemma *range-focusI*:
assumes *p* ∈ *P*[*X*] **and** *lookup p* ‘ *keys* (*p*::- $\Rightarrow_0$ - $\Rightarrow_0$ -::*semiring-1*) ⊆ *P*[− *X*]
shows *p* ∈ *range* (*focus X*)
using *assms*
proof (*induct p rule*: *poly-mapping-plus-induct-Polys*)
case *0*

show *?case* **by** *simp*
next
case (*plus p c t*)
from *plus.hyps*(*3*) **have** *1*: *keys* (*monomial c t*) = {*t*} **by** *simp*
also from *plus.hyps*(*4*) **have** ... ∩ *keys p* = {} **by** *simp*
finally have *keys* (*monomial c t* + *p*) = *keys* (*monomial c t*) ∪ *keys p* **by** (*rule keys-add*[*symmetric*])
hence *2*: *keys* (*monomial c t* + *p*) = *insert t* (*keys p*) **by** (*simp only*: *1 flip*: *insert-is-Un*)
from ‹*t* ∈ .[*X*]› **have** *keys t* ⊆ *X* **by** (*rule PPsD*)
hence *eq1*: *except t X* = *0* **and** *eq2*: *except t* (− *X*) = *t*
by (*rule except-eq-zeroI*, *auto simp*: *except-id-iff*)
from *plus.hyps*(*3*, *4*) *plus.prems* **have** *c* ∈ *P*[− *X*] **and** *lookup p* ‘ *keys p* ⊆ *P*[− *X*]
by (*simp-all add*: *2 lookup-add lookup-single in-keys-iff*)
(*smt* (*verit*) *add.commute add.right-neutral image-cong plus.hyps*(*4*) *when-simps*(*2*))
from *this*(*2*) **have** *p* ∈ *range* (*focus X*) **by** (*rule plus.hyps*)
then obtain *q* **where** *p*: *p* = *focus X q* **..**
moreover from ‹*c* ∈ *P*[− *X*]› **have** *monomial c t* = *focus X* (*monomial 1 t* ∗ *c*)
by (*simp add*: *focus-times focus-monomial eq1 eq2 focus-Polys-Compl times-monomial-monomial*)
ultimately have *monomial c t* + *p* = *focus X* (*monomial 1 t* ∗ *c* + *q*) **by** (*simp only*: *focus-plus*)
thus *?case* **by** (*rule range-eqI*)
qed

lemma *inj-focus*: *inj* ((*focus X*) :: ((*'x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'a*::*ab-group-add*) ⇒ -)
proof (*rule injI*)
fix *p q* :: (*'x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'a*
assume *focus X p* = *focus X q*
hence *focus X* (*p* − *q*) = *0* **by** (*simp add*: *focus-minus*)
thus *p* = *q* **by** *simp*
qed

lemma *flatten-superset*:
assumes *finite A* **and** *keys p* ⊆ *A*
shows *flatten p* = (∑ *t*∈*A*. *punit.monom-mult 1 t* (*lookup p t*))
unfolding *flatten-def* **using** *assms* **by** (*rule sum.mono-neutral-left*) (*simp add*: *in-keys-iff*)

lemma *keys-flatten-subset*: *keys* (*flatten p*) ⊆ (⋃ *t*∈*keys p*. (+) *t* ‘ *keys* (*lookup p t*))
proof −
have *keys* (*flatten p*) ⊆ (⋃ *t*∈*keys p*. *keys* (*punit.monom-mult 1 t* (*lookup p t*)))
unfolding *flatten-def* **by** (*rule keys-sum-subset*)
also from *subset-refl* **have** ... ⊆ (⋃ *t*∈*keys p*. (+) *t* ‘ *keys* (*lookup p t*))
by (*rule UN-mono*) (*rule punit.keys-monom-mult-subset*[*simplified*])
finally show *?thesis* **.**
qed

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 (*intro PolysI subsetI*)
  fix *t*
  assume $t \in$ *keys* (*flatten p*)
  also have . . . $\subseteq$ ($\bigcup$ *t*∈*keys p*. (+) *t ‘ keys* (*lookup p t*)) **by** (*rule keys-flatten-subset*)
  finally obtain *s* **where** $s \in$ *keys p* **and** $t \in (+)$ *s ‘ keys* (*lookup p s*) **..**
  from *this*(*2*) **obtain** *s′* **where** $s' \in$ *keys* (*lookup p s*) **and** *t*: $t = s + s'$ **..**
  from *assms*(*1*) **have** *keys p* $\subseteq$.[*X*] **by** (*rule PolysD*)
  with ‹*s* ∈ *keys p*› **have** $s \in$.[*X*] **..**
  also have . . . $\subseteq$.[*X* ∪ *Y*] **by** (*rule PPs-mono*) *simp*
  finally have *1*: $s \in$.[*X* ∪ *Y*] **.**
  from ‹*s* ∈ *keys p*› **have** *lookup p s* ∈ *lookup p ‘ keys p* **by** (*rule imageI*)
  also have . . . $\subseteq P[Y]$ **by** *fact*
  finally have *keys* (*lookup p s*) $\subseteq$.[*Y*] **by** (*rule PolysD*)
  with ‹*s′* ∈ -› **have** $s' \in$.[*Y*] **..**
  also have . . . $\subseteq$.[*X* ∪ *Y*] **by** (*rule PPs-mono*) *simp*
  finally have $s' \in$.[*X* ∪ *Y*] **.**
  with *1* **show** $t \in$.[*X* ∪ *Y*] **unfolding** *t* **by** (*rule PPs-closed-plus*)
qed

lemma *flatten-zero* [*simp*]: *flatten 0 = 0*
  by (*simp add*: *flatten-def*)

lemma *flatten-one* [*simp*]: *flatten 1 = 1*
  by (*simp add*: *flatten-def*)

lemma *flatten-monomial*: *flatten* (*monomial c t*) = *punit.monom-mult 1 t c*
  by (*simp add*: *flatten-def*)

lemma *flatten-uminus* [*simp*]: *flatten* (− *p*) = − *flatten* (*p*::- $\Rightarrow_0$ - $\Rightarrow_0$ -::*ring*)
  by (*simp add*: *flatten-def keys-uminus punit.monom-mult-uminus-right sum-negf*)

lemma *flatten-plus*: *flatten* (*p* + *q*) = *flatten p* + *flatten q*
proof −
  have *finite* (*keys p* ∪ *keys q*) **by** *simp*
  moreover have *keys* (*p* + *q*) ⊆ *keys p* ∪ *keys q* **by** (*rule Poly-Mapping.keys-add*)
  ultimately show *?thesis*
    by (*simp add*: *flatten-superset*[**where** *A*=*keys p* ∪ *keys q*] *punit.monom-mult-dist-right*
lookup-add
          sum.distrib)
qed

lemma *flatten-minus*: *flatten* (*p* − *q*) = *flatten p* − *flatten* (*q*::- $\Rightarrow_0$ - $\Rightarrow_0$ -::*ring*)
  by (*simp only*: *diff-conv-add-uminus flatten-plus flatten-uminus*)

lemma *flatten-times*: *flatten* (*p* ∗ *q*) = *flatten p* ∗ *flatten* (*q*::- $\Rightarrow_0$ - $\Rightarrow_0$ ′*b*::*comm-semiring-1*)

proof −
have *eq*: *flatten* (*monomial c s* ∗ *q*) = *flatten* (*monomial c s*) ∗ *flatten q* **for** *c s*
proof −
have *eq*: *monomial 1* (*t* + *s*) = *monomial 1 s* ∗ *monomial* (*1*::′*b*) *t* **for** *t*
by (*simp add*: *times-monomial-monomial add.commute*)
have *flatten* (*monomial c s* ∗ *q*) = *flatten* (*punit.monom-mult c s q*)
by (*simp only*: *times-monomial-left*)
also have . . . = (∑ *t*∈(+) *s* ' *keys q*. *punit.monom-mult 1 t* (*lookup* (*punit.monom-mult c s q*) *t*))
by (*rule flatten-superset*) (*simp-all add*: *punit.keys-monom-mult-subset*[*simplified*])
also have . . . = (∑ *t*∈*keys q*. ((λ*t*. *punit.monom-mult 1 t* (*lookup* (*punit.monom-mult c s q*) *t*)) ∘ (+) *s*) *t*)
by (*rule sum.reindex*) *simp*
thm *times-monomial-left*
also have . . . = *punit.monom-mult 1 s c* ∗
(∑ *t*∈*keys q*. *punit.monom-mult 1 t* (*lookup q t*))
by (*simp add*: *o-def punit.lookup-monom-mult sum-distrib-left*)
(*simp add*: *algebra-simps eq flip*: *times-monomial-left*)
also have . . . = *flatten* (*monomial c s*) ∗ *flatten q*
by (*simp only*: *flatten-monomial flatten-def*[**where** *p*=*q*])
finally show *?thesis* **.**
qed
show *?thesis* **by** (*induct p rule*: *poly-mapping-plus-induct*) (*simp-all add*: *ring-distribs flatten-plus eq*)
qed

lemma *flatten-monom-mult*:
flatten (*punit.monom-mult c t p*) = *punit.monom-mult 1 t* (*c* ∗ *flatten* (*p*::- $\Rightarrow_0$ - $\Rightarrow_0$ ′*b*::*comm-semiring-1*))
by (*simp only*: *flatten-times flatten-monomial mult.assoc flip*: *times-monomial-left*)

lemma *flatten-sum*: *flatten* (*sum f I*) = (∑ *i*∈*I*. *flatten* (*f i*))
by (*induct I rule*: *infinite-finite-induct*) (*simp-all add*: *flatten-plus*)

lemma *flatten-prod*: *flatten* (*prod f I*) = (∏ *i*∈*I*. *flatten* (*f i* :: - $\Rightarrow_0$ -::*comm-semiring-1*))
by (*induct I rule*: *infinite-finite-induct*) (*simp-all add*: *flatten-times*)

lemma *flatten-power* [*simp*]: *flatten* (*f* ^ *m*) = *flatten* (*f*:: - $\Rightarrow_0$ -::*comm-semiring-1*) ^ *m*
by (*induct m*) (*simp-all add*: *flatten-times*)

lemma *surj-flatten*: *surj flatten*
proof (*rule surjI*)
fix *p*
show *flatten* (*monomial p 0*) = *p* **by** (*simp add*: *flatten-monomial*)
qed

lemma *flatten-focus* [*simp*]: *flatten* (*focus X p*) = *p*
by (*induct p rule*: *poly-mapping-plus-induct*)

(*simp-all add*: *focus-plus flatten-plus focus-monomial flatten-monomial punit.monom-mult-monomial add.commute flip*: *except-decomp*)

lemma *focus-flatten*:
assumes $p \in P[X]$ **and** *lookup* p ' *keys* $p \subseteq P[- X]$
shows *focus* X (*flatten* p) = p
proof −
from *assms* **have** $p \in$ *range* (*focus* X) **by** (*rule range-focusI*)
then obtain q **where** $p =$ *focus* X q **..**
thus *?thesis* **by** *simp*
qed

lemma *image-focus-ideal*: *focus* X ' *ideal* F = *ideal* (*focus* X ' F) ∩ *range* (*focus* X)
proof
from *focus-plus focus-times* **have** *focus* X ' *ideal* $F \subseteq$ *ideal* (*focus* X ' F)
by (*rule image-ideal-subset*)
moreover from *subset-UNIV* **have** *focus* X ' *ideal* $F \subseteq$ *range* (*focus* X) **by** (*rule image-mono*)
ultimately show *focus* X ' *ideal* $F \subseteq$ *ideal* (*focus* X ' F) ∩ *range* (*focus* X) **by** *blast*
next
show *ideal* (*focus* X ' F) ∩ *range* (*focus* X) $\subseteq$ *focus* X ' *ideal* F
proof
fix p
assume $p \in$ *ideal* (*focus* X ' F) ∩ *range* (*focus* X)
hence $p \in$ *ideal* (*focus* X ' F) **and** $p \in$ *range* (*focus* X) **by** *simp-all*
from *this*(*1*) **obtain** *F0* q **where** *F0* $\subseteq$ *focus* X ' F **and** p: $p = (\sum f' \in F0.\ q\ f' * f')$
by (*rule ideal.spanE*)
from *this*(*1*) **obtain** F' **where** $F' \subseteq F$ **and** *F0*: *F0* = *focus* X ' F' **by** (*rule subset-imageE*)
from *inj-focus subset-UNIV* **have** *inj-on* (*focus* X) F' **by** (*rule inj-on-subset*)
from ‹$p \in$ *range* -› **obtain** p' **where** $p =$ *focus* X p' **..**
hence $p =$ *focus* X (*flatten* p) **by** *simp*
also from ‹*inj-on* - F'› **have** ... = *focus* X ($\sum f' \in F'$. *flatten* (q (*focus* X f')) $* f'$)
by (*simp add*: *p F0 sum.reindex flatten-sum flatten-times*)
finally have $p =$ *focus* X ($\sum f' \in F'$. *flatten* (q (*focus* X f')) $* f'$) **.**
moreover have ($\sum f' \in F'$. *flatten* (q (*focus* X f')) $* f'$) $\in$ *ideal* F
proof
show ($\sum f' \in F'$. *flatten* (q (*focus* X f')) $* f'$) $\in$ *ideal* F' **by** (*rule ideal.sum-in-spanI*)
next
from ‹$F' \subseteq F$› **show** *ideal* $F' \subseteq$ *ideal* F **by** (*rule ideal.span-mono*)
qed
ultimately show $p \in$ *focus* X ' *ideal* F **by** (*rule image-eqI*)
qed
qed

lemma *image-flatten-ideal*: *flatten ‘ ideal F* = *ideal* (*flatten ‘ F*)
using *flatten-plus flatten-times surj-flatten* **by** (*rule image-ideal-eq-surj*)

lemma *poly-eval-focus*:
poly-eval a (*focus X p*) = *poly-subst* (*λx. if x ∈ X then a x else monomial 1* (*Poly-Mapping.single x 1*)) *p*
proof −
let *?b* = *λx. if x ∈ X then a x else monomial 1* (*Poly-Mapping.single x 1*)
have ∗: *lookup* (*punit.monom-mult* (*monomial* (*lookup p t*) (*except t X*)) *0*
(*subst-pp* (*λx. monomial* (*a x*) *0*) (*except t* (− *X*)))) *0* =
punit.monom-mult (*lookup p t*) *0* (*subst-pp ?b t*) **for** *t*
proof −
have *1*: *subst-pp ?b* (*except t X*) = *monomial 1* (*except t X*)
by (*rule subst-pp-id*) (*simp add*: *keys-except*)
from *refl* **have** *2*: *subst-pp ?b* (*except t* (− *X*)) = *subst-pp a* (*except t* (−*X*))
by (*rule subst-pp-cong*) (*simp add*: *keys-except*)
have *lookup* (*punit.monom-mult* (*monomial* (*lookup p t*) (*except t X*)) *0*
(*subst-pp* (*λx. monomial* (*a x*) *0*) (*except t* (− *X*)))) *0* =
punit.monom-mult (*lookup p t*) (*except t X*) (*subst-pp a* (*except t* (− *X*)))
by (*simp add*: *lookup-times-zero subst-pp-def lookup-prod-zero lookup-power-zero flip*: *times-monomial-left*)
also have . . . = *punit.monom-mult* (*lookup p t*) *0* (*monomial 1* (*except t X*) ∗ *subst-pp a* (*except t* (− *X*)))
by (*simp add*: *times-monomial-monomial flip*: *times-monomial-left mult.assoc*)
also have . . . = *punit.monom-mult* (*lookup p t*) *0* (*subst-pp ?b* (*except t X* + *except t* (− *X*)))
by (*simp only*: *subst-pp-plus 1 2*)
also have . . . = *punit.monom-mult* (*lookup p t*) *0* (*subst-pp ?b t*) **by** (*simp flip*: *except-decomp*)
finally show *?thesis* **.**
qed
show *?thesis* **by** (*simp add*: *poly-eval-def focus-def poly-subst-sum lookup-sum poly-subst-monomial* ∗
flip: *poly-subst-def*)
qed

corollary *poly-eval-poly-eval-focus*:
poly-eval a (*poly-eval b* (*focus X p*)) = *poly-eval* (*λx::'x. if x ∈ X then poly-eval a* (*b x*) *else a x*) *p*
proof −
have *eq*: *monomial* (*lookup* (*poly-subst* (*λy. monomial* (*a y*) (*0::'x* $\Rightarrow_0$ *nat*)) *q*) *0*) *0* =
poly-subst (*λy. monomial* (*a y*) (*0::'x* $\Rightarrow_0$ *nat*)) *q* **for** *q*
by (*intro poly-deg-zero-imp-monomial poly-deg-poly-subst-eq-zeroI*) *simp*
show *?thesis* **unfolding** *poly-eval-focus*
by (*simp add*: *poly-eval-def poly-subst-poly-subst if-distrib poly-subst-monomial subst-pp-single eq*
cong: *if-cong*)
qed

lemma *indets-poly-eval-focus-subset*:
indets (*poly-eval a* (*focus X p*)) $\subseteq \bigcup$ (*indets* ' *a* ' *X*) $\cup$ (*indets p* $-$ *X*)
proof
fix *x*
assume $x \in$ *indets* (*poly-eval a* (*focus X p*))
also have ... = *indets* (*poly-subst* (λx. *if* $x \in X$ *then a x else monomial 1* (*Poly-Mapping.single x 1*)) *p*)
(**is** - = *indets* (*poly-subst ?f* -)) **by** (*simp only*: *poly-eval-focus*)
finally obtain *y* **where** $y \in$ *indets p* **and** $x \in$ *indets* (*?f y*) **by** (*rule in-indets-poly-substE*)
from *this*(*2*) **have** ($x \notin X \wedge x = y$) $\vee$ ($y \in X \wedge x \in$ *indets* (*a y*))
by (*simp add*: *indets-monomial split*: *if-split-asm*)
thus $x \in \bigcup$ (*indets* ' *a* ' *X*) $\cup$ (*indets p* $-$ *X*)
proof (*elim disjE conjE*)
assume $x \notin X$ **and** $x = y$
with ‹$y \in$ *indets p*› **have** $x \in$ *indets p* $-$ *X* **by** *simp*
thus *?thesis* **..**
next
assume $y \in X$ **and** $x \in$ *indets* (*a y*)
hence $x \in \bigcup$ (*indets* ' *a* ' *X*) **by** *blast*
thus *?thesis* **..**
qed
qed

lemma *lookup-poly-eval-focus*:
lookup (*poly-eval* (λx. *monomial* (*a x*) *0*) (*focus X p*)) *t* = *poly-eval a* (*lookup* (*focus* ($-$ *X*) *p*) *t*)
proof $-$
let *?f* = λx. *if* $x \in X$ *then monomial* (*a x*) *0 else monomial 1* (*Poly-Mapping.single x 1*)
have *eq*: *subst-pp ?f s* = *monomial* ($\prod x \in$*keys s* $\cap$ *X*. *a x* ^ *lookup s x*) (*except s X*) **for** *s*
proof $-$
have *subst-pp ?f s* = ($\prod x \in$(*keys s* $\cap$ *X*) $\cup$ (*keys s* $-$ *X*). *?f x* ^ *lookup s x*)
unfolding *subst-pp-def* **by** (*rule prod.cong*) *blast+*
also have ... = ($\prod x \in$*keys s* $\cap$ *X*. *?f x* ^ *lookup s x*) $*$ ($\prod x \in$*keys s* $-$ *X*. *?f x* ^ *lookup s x*)
by (*rule prod.union-disjoint*) *auto*
also have ... = *monomial* ($\prod x \in$*keys s* $\cap$ *X*. *a x* ^ *lookup s x*)
($\sum x \in$*keys s* $-$ *X*. *Poly-Mapping.single x* (*lookup s x*))
by (*simp add*: *monomial-power-map-scale times-monomial-monomial flip*: *punit.monomial-prod-sum*)
also have ($\sum x \in$*keys s* $-$ *X*. *Poly-Mapping.single x* (*lookup s x*)) = *except s X*
by (*metis* (*mono-tags*, *lifting*) *DiffD2 keys-except lookup-except-eq-idI*
poly-mapping-sum-monomials sum.cong)
finally show *?thesis* **.**
qed
show *?thesis*
by (*simp add*: *poly-eval-focus poly-subst-def lookup-sum eq flip*: *punit.map-scale-eq-monom-mult*)

(*simp add*: *focus-def lookup-sum poly-eval-sum lookup-single when-distrib poly-eval-monomial*
keys-except lookup-except-when)
qed

lemma *keys-poly-eval-focus-subset*:
keys (*poly-eval* (λx. *monomial* (*a x*) *0*) (*focus X p*)) $\subseteq$ (λt. *except t X*) ‘ *keys p*
proof
fix *t*
assume $t \in$ *keys* (*poly-eval* (λx. *monomial* (*a x*) *0*) (*focus X p*))
hence *lookup* (*poly-eval* (λx. *monomial* (*a x*) *0*) (*focus X p*)) *t* $\neq$ *0* **by** (*simp add*: *in-keys-iff*)
hence *poly-eval a* (*lookup* (*focus* ($-$ *X*) *p*) *t*) $\neq$ *0* **by** (*simp add*: *lookup-poly-eval-focus*)
hence $t \in$ *keys* (*focus* ($-$ *X*) *p*) **by** (*auto simp flip*: *lookup-not-eq-zero-eq-in-keys*)
thus $t \in$ (λt. *except t X*) ‘ *keys p* **by** (*simp add*: *keys-focus*)
qed

lemma *poly-eval-focus-in-Polys*:
assumes $p \in P[X]$
shows *poly-eval* (λx. *monomial* (*a x*) *0*) (*focus Y p*) $\in P[X - Y]$
proof (*rule PolysI-alt*)
have *indets* (*poly-eval* (λx. *monomial* (*a x*) *0*) (*focus Y p*)) $\subseteq$
$\bigcup$ (*indets* ‘ (λx. *monomial* (*a x*) *0*) ‘ *Y*) $\cup$ (*indets p* $-$ *Y*)
by (*fact indets-poly-eval-focus-subset*)
also have ... = *indets p* $-$ *Y* **by** *simp*
also from *assms* **have** ... $\subseteq X - Y$ **by** (*auto dest*: *PolysD*)
finally show *indets* (*poly-eval* (λx. *monomial* (*a x*) *0*) (*focus Y p*)) $\subseteq X - Y$ **.**
qed

lemma *image-poly-eval-focus-ideal*:
poly-eval (λx. *monomial* (*a x*) *0*) ‘ *focus X* ‘ *ideal F* =
ideal (*poly-eval* (λx. *monomial* (*a x*) *0*) ‘ *focus X* ‘ *F*) $\cap$
($P[- X]$::(('*x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ '*a*::*comm-ring-1*) *set*)
proof $-$
let *?h* = λf. *poly-eval* (λx. *monomial* (*a x*) *0*) (*focus X f*)
have *h-id*: *?h p* = *p* **if** $p \in P[- X]$ **for** *p*
proof $-$
from *that* **have** *focus X p* $\in P[- X \cap X]$ **by** (*rule focus-in-Polys'*)
also have ... = $P[\{\}]$ **by** *simp*
finally obtain *c* **where** *eq*: *focus X p* = *monomial c 0* **unfolding** *Polys-empty* **..**
hence *flatten* (*focus X p*) = *flatten* (*monomial c 0*) **by** (*rule arg-cong*)
hence *c* = *p* **by** (*simp add*: *flatten-monomial*)
thus *?h p* = *p* **by** (*simp add*: *eq poly-eval-monomial*)
qed
have *rng*: *range ?h* = $P[- X]$
proof (*intro subset-antisym subsetI*, *elim rangeE*)
fix *b f*
assume *b*: *b* = *?h f*

have *?h f* ∈ *P*[*UNIV* − *X*] **by** (*rule poly-eval-focus-in-Polys*) *simp*
thus *b* ∈ *P*[− *X*] **by** (*simp add*: *b Compl-eq-Diff-UNIV*)
next
fix *p* :: (′*x* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ ′*a*
assume *p* ∈ *P*[− *X*]
hence *?h p* = *p* **by** (*rule h-id*)
hence *p* = *?h p* **by** (*rule sym*)
thus *p* ∈ *range ?h* **by** (*rule range-eqI*)
qed
have *poly-eval* (λ*x*. *monomial* (*a x*) *0*) ' *focus X* ' *ideal F* = *?h* ' *ideal F* **by** (*fact image-image*)
also have . . . = *ideal* (*?h* ' *F*) ∩ *range ?h*
proof (*rule image-ideal-eq-Int*)
fix *p*
have *?h p* ∈ *range ?h* **by** (*fact rangeI*)
also have . . . = *P*[− *X*] **by** *fact*
finally show *?h* (*?h p*) = *?h p* **by** (*rule h-id*)
qed (*simp-all only*: *focus-plus poly-eval-plus focus-times poly-eval-times*)
also have . . . = *ideal* (*poly-eval* (λ*x*. *monomial* (*a x*) *0*) ' *focus X* ' *F*) ∩ *P*[− *X*]
by (*simp only*: *image-image rng*)
finally show *?thesis* **.**
qed

17.9 Locale *pm-powerprod*

lemma *varnum-eq-zero-iff*: *varnum X t* = *0* ⟷ *t* ∈ .[*X*]
by (*auto simp*: *varnum-def PPs-def*)

lemma *dgrad-set-varnum*: *dgrad-set* (*varnum X*) *0* = .[*X*]
by (*simp add*: *dgrad-set-def PPs-def varnum-eq-zero-iff*)

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*} $\Rightarrow_0$ *nat*) ⇒ (′*x* $\Rightarrow_0$ *nat*) ⇒ *bool* (**infixl** ‹⪯› *50*)
and *ord-strict* (**infixl** ‹≺› *50*)
begin

sublocale *gd-powerprod* **..**

lemma *PPs-closed-lpp*:
  assumes $p \in P[X]$
  shows *lpp* $p \in .[X]$
proof (*cases* $p = 0$)
  case *True*
  thus *?thesis* **by** (*simp add*: *zero-in-PPs*)
next
  case *False*
  hence *lpp p* $\in$ *keys p* **by** (*rule punit.lt-in-keys*)
  also from *assms* **have** $\ldots \subseteq .[X]$ **by** (*rule PolysD*)
  finally show *?thesis* **.**
qed

lemma *PPs-closed-tpp*:
  assumes $p \in P[X]$
  shows *tpp* $p \in .[X]$
proof (*cases* $p = 0$)
  case *True*
  thus *?thesis* **by** (*simp add*: *zero-in-PPs*)
next
  case *False*
  hence *tpp p* $\in$ *keys p* **by** (*rule punit.tt-in-keys*)
  also from *assms* **have** $\ldots \subseteq .[X]$ **by** (*rule PolysD*)
  finally show *?thesis* **.**
qed

corollary *PPs-closed-image-lpp*: $F \subseteq P[X] \Longrightarrow$ *lpp* ‘ $F \subseteq .[X]$
  by (*auto intro*: *PPs-closed-lpp*)

corollary *PPs-closed-image-tpp*: $F \subseteq P[X] \Longrightarrow$ *tpp* ‘ $F \subseteq .[X]$
  by (*auto intro*: *PPs-closed-tpp*)

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 −
  from *assms* **have** *lpp p* $\in$ *keys p* **by** (*rule punit.lt-in-keys*)
  hence $*$: *lpp p* $\in$ *keys ?p* **by** (*simp add*: *keys-hom-component*)
  thus *?p* $\neq 0$ **by** *auto*

  from $*$ **show** *lpp ?p* = *lpp p*
  proof (*rule punit.lt-eqI-keys*)
    fix *t*
    assume $t \in$ *keys ?p*
    hence $t \in$ *keys p* **by** (*simp add*: *keys-hom-component*)
    thus $t \preceq$ *lpp p* **by** (*rule punit.lt-max-keys*)
  qed

qed

definition *is-hom-ord* :: *'x* ⇒ *bool*
  where *is-hom-ord x* ⟷ (∀ *s t*. *deg-pm s* = *deg-pm t* ⟶ (*s* ⪯ *t* ⟷ *except s* {*x*} ⪯ *except t* {*x*}))

lemma *is-hom-ordD*: *is-hom-ord x* ⟹ *deg-pm s* = *deg-pm t* ⟹ *s* ⪯ *t* ⟷ *except s* {*x*} ⪯ *except t* {*x*}
  by (*simp add*: *is-hom-ord-def*)

lemma *dgrad-p-set-varnum*: *punit.dgrad-p-set* (*varnum X*) *0* = *P*[*X*]
  by (*simp add*: *punit.dgrad-p-set-def dgrad-set-varnum Polys-def*)

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* ⇒ *'a option* ⇒ *bool* **where**
  less-eq-option None - = True |
  less-eq-option (*Some x*) *None* = *False* |
  less-eq-option (*Some x*) (*Some y*) = (*x* ≤ *y*)

definition *less-option* :: *'a option* ⇒ *'a option* ⇒ *bool*
  where *less-option x y* ⟷ *x* ≤ *y* ∧ ¬ *y* ≤ *x*

instance proof
  fix *x* :: *'a option*
  show *x* ≤ *x* **using** *less-eq-option.elims*(*3*) **by** *fastforce*
qed (*auto simp*: *less-option-def elim*!: *less-eq-option.elims*)

end

locale *extended-ord-pm-powerprod* = *pm-powerprod*
begin

definition *extended-ord* :: (*'a option* $\Rightarrow_0$ *nat*) ⇒ (*'a option* $\Rightarrow_0$ *nat*) ⇒ *bool*
  where *extended-ord s t* ⟷ (*restrict-indets-pp s* ≺ *restrict-indets-pp t* ∨
      (*restrict-indets-pp s* = *restrict-indets-pp t* ∧ *lookup s None* ≤ *lookup t None*))

definition *extended-ord-strict* :: (*'a option* $\Rightarrow_0$ *nat*) ⇒ (*'a option* $\Rightarrow_0$ *nat*) ⇒ *bool*
  where *extended-ord-strict s t* ⟷ (*restrict-indets-pp s* ≺ *restrict-indets-pp t* ∨
      (*restrict-indets-pp s* = *restrict-indets-pp t* ∧ *lookup s None* < *lookup t None*))

sublocale *extended-ord*: *pm-powerprod extended-ord extended-ord-strict*

proof −
  have *1*: *s* = *t* **if** *lookup s None* = *lookup t None* **and** *restrict-indets-pp s* = *restrict-indets-pp t*
    for *s t* :: *'a option* $\Rightarrow_0$ *nat*
  proof (*rule poly-mapping-eqI*)
    fix *y*
    show *lookup s y* = *lookup t y*
    proof (*cases y*)
      case *None*
      with *that*(*1*) **show** *?thesis* **by** *simp*
    next
      case *y*: (*Some z*)
    have *lookup s y* = *lookup* (*restrict-indets-pp s*) *z* **by** (*simp only*: *lookup-restrict-indets-pp y*)
      also have . . . = *lookup* (*restrict-indets-pp t*) *z* **by** (*simp only*: *that*(*2*))
      also have . . . = *lookup t y* **by** (*simp only*: *lookup-restrict-indets-pp y*)
      finally show *?thesis* **.**
    qed
  qed
  have *2*: *0* ≺ *t* **if** *t* ≠ *0* **for** *t*::*'a* $\Rightarrow_0$ *nat*
   using *that zero-min* **by** (*rule ordered-powerprod-lin.dual-order.not-eq-order-implies-strict*)
  show *pm-powerprod extended-ord extended-ord-strict*
   by *standard* (*auto simp*: *extended-ord-def extended-ord-strict-def restrict-indets-pp-plus lookup-add 1*
                dest: *plus-monotone-strict 2*)
qed

lemma *extended-ord-is-hom-ord*: *extended-ord.is-hom-ord None*
  by (*auto simp add*: *extended-ord-def lookup-restrict-indets-pp lookup-except extended-ord.is-hom-ord-def*
          simp flip: *deg-pm-restrict-indets-pp*)

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* ⇒ *'a*::*comm-monoid-add poly* ⇒ *'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* ⊆ {*v*}))

lemma *poly-to-mpoly-finite*: *finite* {*m*::*nat* $\Rightarrow_0$ *nat*. (*coeff p* (*Poly-Mapping.lookup*

m v) *when Poly-Mapping.keys m* ⊆ {*v*}) ≠ *0*} (**is** *finite ?M*)
proof −
have *?M* ⊆ *Poly-Mapping.single v* ' {*x. Polynomial.coeff p x* ≠ *0*}
proof
fix *m* **assume** *m* ∈ *?M*
then have ⋀*v'. v'*≠*v* ⟹ *Poly-Mapping.lookup m v'* = *0* **by** (*fastforce simp add*: *in-keys-iff*)
then have *m* = *Poly-Mapping.single v* (*Poly-Mapping.lookup m v*)
using *Poly-Mapping.poly-mapping-eqI* **by** (*metis* (*full-types*) *lookup-single-eq lookup-single-not-eq*)
then show *m* ∈ (*Poly-Mapping.single v*) ' {*x. Polynomial.coeff p x* ≠ *0*} **using** ‹*m* ∈ *?M*› **by** *auto*
qed
then show *?thesis* **using** *finite-surj*[*OF MOST-coeff-eq-0*[*unfolded eventually-cofinite*]] **by** *blast*
qed

lemma *coeff-poly-to-mpoly*: *MPoly-Type.coeff* (*poly-to-mpoly v p*) (*Poly-Mapping.single v k*) = *Polynomial.coeff p k*
unfolding *poly-to-mpoly-def coeff-def MPoly-inverse*[*OF Set.UNIV-I*] *lookup-Abs-poly-mapping*[*OF poly-to-mpoly-finite*]
using *empty-subsetI keys-single lookup-single order-refl when-simps*(*1*) **by** *simp*

definition *mpoly-to-poly*::*nat* ⇒ *'a*::*comm-monoid-add mpoly* ⇒ *'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 −
have *0*:*Poly-Mapping.single v* ' {*x. Poly-Mapping.lookup* (*mapping-of p*) (*Poly-Mapping.single v x*) ≠ *0*}
⊆ {*k. Poly-Mapping.lookup* (*mapping-of p*) *k* ≠ *0*}
by *auto*
have ∀$_\infty$ *k. MPoly-Type.coeff p* (*Poly-Mapping.single v k*) = *0* **unfolding** *coeff-def eventually-cofinite*
using *finite-imageD*[*OF finite-subset*[*OF 0 Poly-Mapping.finite-lookup*]] *inj-single* **by** (*metis inj-eq inj-onI*)
then show *?thesis*
unfolding *mpoly-to-poly-def* **by** (*simp add*: *Abs-poly-inverse*)
qed

lemma *mpoly-to-poly-inverse*:
assumes *vars p* ⊆ {*v*}
shows *poly-to-mpoly v* (*mpoly-to-poly v p*) = *p*
proof −
define *f* **where** *f* = (λ*m. Polynomial.coeff* (*mpoly-to-poly v p*) (*Poly-Mapping.lookup m v*) *when Poly-Mapping.keys m* ⊆ {*v*})
have *finite* {*m. f m* ≠ *0*} **unfolding** *f-def* **using** *poly-to-mpoly-finite* **by** *blast*

have *Abs-poly-mapping f* = *mapping-of p*
proof (*rule Poly-Mapping.poly-mapping-eqI*)
fix *m*
show *Poly-Mapping.lookup* (*Abs-poly-mapping f*) *m* = *Poly-Mapping.lookup* (*mapping-of p*) *m*
proof (*cases Poly-Mapping.keys m* ⊆ {*v*})
assume *Poly-Mapping.keys m* ⊆ {*v*}
then show *?thesis* **unfolding** *Poly-Mapping.lookup-Abs-poly-mapping*[*OF* ‹*finite* {*m. f m* ≠ *0*}›] **unfolding** *f-def*
unfolding *coeff-mpoly-to-poly coeff-def* **using** *when-simps*(*1*) **apply** *simp*
using *keys-single lookup-not-eq-zero-eq-in-keys lookup-single-eq*
lookup-single-not-eq poly-mapping-eqI subset-singletonD
by (*metis* (*no-types*, *lifting*) *aux lookup-eq-zero-in-keys-contradict*)
next
assume ¬*Poly-Mapping.keys m* ⊆ {*v*}
then show *?thesis* **unfolding** *Poly-Mapping.lookup-Abs-poly-mapping*[*OF* ‹*finite* {*m. f m* ≠ *0*}›] **unfolding** *f-def*
using ‹*vars p* ⊆ {*v*}› **unfolding** *vars-def* **by** (*metis* (*no-types*, *lifting*) *UN-I lookup-not-eq-zero-eq-in-keys subsetCE subsetI when-def*)
qed
qed
then show *?thesis*
unfolding *poly-to-mpoly-def f-def* **by** (*simp add*: *mapping-of-inverse*)
qed

lemma *poly-to-mpoly-inverse*: *mpoly-to-poly v* (*poly-to-mpoly v p*) = *p*
unfolding *mpoly-to-poly-def coeff-poly-to-mpoly* **by** (*simp add*: *coeff-inverse*)

lemma *poly-to-mpoly0*: *poly-to-mpoly v 0* = *0*
proof −
have ⋀*m.* (*Polynomial.coeff 0* (*Poly-Mapping.lookup m v*) *when Poly-Mapping.keys m* ⊆ {*v*}) = *0* **by** *simp*
have *Abs-poly-mapping* (λ*m. Polynomial.coeff 0* (*Poly-Mapping.lookup m v*) *when Poly-Mapping.keys m* ⊆ {*v*}) = *0*
apply (*rule Poly-Mapping.poly-mapping-eqI*) **unfolding** *lookup-Abs-poly-mapping*[*OF poly-to-mpoly-finite*] **by** *auto*
then show *?thesis* **using** *poly-to-mpoly-def zero-mpoly.abs-eq* **by** (*metis* (*no-types*))
qed

lemma *mpoly-to-poly-add*: *mpoly-to-poly v* (*p1* + *p2*) = *mpoly-to-poly v p1* + *mpoly-to-poly v p2*
unfolding *Polynomial.plus-poly.abs-eq More-MPoly-Type.coeff-add coeff-mpoly-to-poly*
using *mpoly-to-poly-def* **by** *auto*

lemma *poly-eq-insertion*:
assumes *vars p* ⊆ {*v*}
shows *poly* (*mpoly-to-poly v p*) *x* = *insertion* (λ*v. x*) *p*
using *assms* **proof** (*induction p rule*:*mpoly-induct*)
case (*monom m a*)

then show *?case*
proof (*cases a=0*)
case *True*
then show *?thesis*
by (*metis MPoly-Type.monom.abs-eq insertion-zero monom-zero poly-0 poly-to-mpoly0 poly-to-mpoly-inverse single-zero*)
next
case *False*
then have *Poly-Mapping.keys m* ⊆ {*v*} **using** *monom* **unfolding** *vars-def MPoly-Type.mapping-of-monom keys-single* **by** *simp*
then have ⋀*v'*. *v'*≠*v* ⟹ *Poly-Mapping.lookup m v'* = *0* **unfolding** *vars-def* **by** (*auto simp*: *in-keys-iff*)
then have *m* = *Poly-Mapping.single v* (*Poly-Mapping.lookup m v*)
by (*metis lookup-single-eq lookup-single-not-eq poly-mapping-eqI*)
then have *0*:*insertion* (λ*v*. *x*) (*MPoly-Type.monom m a*) = *a* ∗ *x* ^(*Poly-Mapping.lookup m v*)
using *insertion-single* **by** *metis*
have ⋀*k*. *Poly-Mapping.single v k* = *m* ⟷ *Poly-Mapping.lookup m v* = *k*
using ‹*m* = *Poly-Mapping.single v* (*Poly-Mapping.lookup m v*)› **by** *auto*
then have *monom a* (*Poly-Mapping.lookup m v*) = (*Abs-poly* (λ*k*. *if Poly-Mapping.single v k* = *m then a else 0*))
by (*simp add*: *Polynomial.monom.abs-eq*)
then show *?thesis* **unfolding** *mpoly-to-poly-def More-MPoly-Type.coeff-monom 0 when-def* **by** (*metis poly-monom*)
qed
next
case (*sum p1 p2 m a*)
then have *poly* (*mpoly-to-poly v p1*) *x* = *insertion* (λ*v*. *x*) *p1*
poly (*mpoly-to-poly v p2*) *x* = *insertion* (λ*v*. *x*) *p2*
by (*simp-all add*: *vars-add-monom*)
then show *?case* **unfolding** *insertion-add mpoly-to-poly-add* **by** *simp*
qed

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*}
using *poly-roots-finite*[*of mpoly-to-poly v p*, *unfolded poly-eq-insertion*[*OF* ‹*vars p* ⊆ {*v*}›]]
using *assms*(*1*) *assms*(*2*) *mpoly-to-poly-inverse poly-to-mpoly0* **by** *fastforce*

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*
by (*induct m*, *auto simp*: *field-simps*)

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*
unfolding *sum-var-list-def* **by** (*induct m*, *auto*)

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 −
define *M* **where** *M* = *map fst m*
from *assms*[*unfolded monom-inv-def*]
have *distinct* (*x* # *map fst m*) *sorted* (*x* # *map fst m*) **by** *auto*
with *assms*(*2*) **have** *y* ∉ *set* (*map fst m*) **unfolding** *M-def*[*symmetric*]
by (*induct M*, *auto*)
thus *?thesis* **by** *auto*
qed

lemma *eq-monom-sum-var-list*: **assumes** *monom-inv m* **and** *monom-inv n*
shows (*m* = *n*) = (∀ *x*. *sum-var-list m x* = *sum-var-list n x*) (**is** *?l* = *?r*)
using *assms*
proof (*induct m arbitrary*: *n*)
case *Nil*
show *?case*
proof (*cases n*)
case (*Cons yp nn*)
obtain *y p* **where** *yp*: *yp* = (*y*,*p*) **by** (*cases yp*, *auto*)
with *Cons Nil*(*2*)[*unfolded monom-inv-def*] **have** *p*: *0* < *p* **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 *?case*
proof (*cases n*)
case *Nil*
thus *?thesis* **by** (*auto simp*: *xp sum-var-list-def p intro*!: *exI*[*of - x*])
next
case *n*: (*Cons yq n'*)
from *Cons*(*3*)[*unfolded n*] **have** *n'*: *monom-inv n'* **by** (*auto simp*: *monom-inv-def*)
show *?thesis*
proof (*cases yq* = *xp*)

case *True*
show *?thesis* **unfolding** *n True* **using** *Cons*(*1*)[*OF m n'*] **by** (*auto simp*: *xp sum-var-list-def*)
next
case *False*
obtain *y q* **where** *yq*: *yq* = (*y*,*q*) **by** *force*
from *Cons*(*3*)[*unfolded n yq monom-inv-def*] **have** *q*: *q* > *0* **by** *auto*
define *z* **where** *z* = *min x y*
have *zm*: *z* ∉ *fst* ‘ *set m* **using** *Cons*(*2*) **unfolding** *xp z-def*
by (*rule monom-inv-Cons*, *simp*)
have *zn'*: *z* ∉ *fst* ‘ *set n'* **using** *Cons*(*3*) **unfolding** *n yq z-def*
by (*rule monom-inv-Cons*, *simp*)
have *smz*: *sum-var-list* (*xp* # *m*) *z* = *sum-var-list* [(*x*,*p*)] *z*
using *sum-var-list-not*[*OF zm*] **by** (*simp add*: *sum-var-list-def xp*)
also have ... ≠ *sum-var-list* [(*y*,*q*)] *z* **using** *False* **unfolding** *xp yq*
by (*auto simp*: *sum-var-list-def z-def p q min-def*)
also have *sum-var-list* [(*y*,*q*)] *z* = *sum-var-list n z*
using *sum-var-list-not*[*OF zn'*] **by** (*simp add*: *sum-var-list-def n yq*)
finally show *?thesis* **using** *False* **unfolding** *n* **by** *auto*
qed
qed
qed

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*
by (*induct m1 m2 rule*: *monom-mult-list.induct*, *auto split*: *list.splits*)

lemma *monom-mult-list-inv*: *monom-inv m1* ⟹ *monom-inv m2* ⟹ *monom-inv* (*monom-mult-list m1 m2*)
proof (*induct m1 m2 rule*: *monom-mult-list.induct*)
case (*2 x p m n'*)
note *IH* = *2*(*1*−*3*)
note *xpm* = *2*(*4*)
note *n'* = *2*(*5*)

show *?case*
proof (*cases n'*)
case *Nil*
with *xpm* **show** *?thesis* **by** *auto*
next
case (*Cons yq n*)
then obtain *y q* **where** *id*: $n' = ((y,q) \# n)$ **by** (*cases yq*, *auto*)
from *xpm* **have** *m*: *monom-inv m* **and** *p*: $p > 0$ **and** *x*: $x \notin fst$ ‘ *set m*
and *xm*: $\bigwedge z.\ z \in fst$ ‘ *set m* $\Longrightarrow x \leq z$
unfolding *monom-inv-def* **by** (*auto*)
from *n'*[*unfolded id*] **have** *n*: *monom-inv n* **and** *q*: $q > 0$ **and** *y*: $y \notin fst$ ‘ *set n*
and *yn*: $\bigwedge z.\ z \in fst$ ‘ *set n* $\Longrightarrow y \leq z$
unfolding *monom-inv-def* **by** (*auto*)
show *?thesis*
proof (*cases* $x = y$)
case *True*
hence *res*: *monom-mult-list* $((x, p) \# m)\ n' = (x, p + q) \#$ *monom-mult-list m n*
by (*simp add*: *id*)
from *IH*(*1*)[*OF id refl True m n*] **have** *inv*: *monom-inv* (*monom-mult-list m n*) **by** *simp*
show *?thesis* **unfolding** *res* **using** *inv p x y True xm yn*
by (*fastforce simp add*: *monom-inv-def monom-list-mult-list-vars*)
next
case *False*
show *?thesis*
proof (*cases* $x < y$)
case *True*
hence *res*: *monom-mult-list* $((x, p) \# m)\ n' = (x,p) \#$ *monom-mult-list m n'*
by (*auto simp add*: *id*)
from *IH*(*2*)[*OF id refl False True m n'*] **have** *inv*: *monom-inv* (*monom-mult-list m n'*) **.**
show *?thesis* **unfolding** *res* **using** *inv p x y True xm yn* **unfolding** *id*
by (*fastforce simp add*: *monom-inv-def monom-list-mult-list-vars*)
next
case *gt*: *False*
with *False* **have** *lt*: $y < x$ **by** *auto*
hence *res*: *monom-mult-list* $((x, p) \# m)\ n' = (y,q) \#$ *monom-mult-list* $((x, p) \# m)\ n$
using *False* **by** (*auto simp add*: *id*)
from *lt* **have** *zm*: $z \leq x \Longrightarrow (z,b) \notin$ *set m* **for** *z b* **using** *xm*[*of z*] *x* **by** *force*
from *zm*[*of y*] *lt* **have** *ym*: $(y,b) \notin$ *set m* **for** *b* **by** *auto*
from *yn* **have** *yn'*: $(a, b) \in$ *set n* $\Longrightarrow y \leq a$ **for** *a b* **by** *force*
from *IH*(*3*)[*OF id refl False gt xpm n*] **have** *inv*: *monom-inv* (*monom-mult-list* $((x, p) \# m)\ n$) **.**
define *xpm* **where** $xpm = ((x,p) \# m)$
have *xpm'*: *fst* ‘ *set xpm* = *insert x* (*fst* ‘ *set m*) **unfolding** *xpm-def* **by**

auto
  show *?thesis* **unfolding** *res* **using** *inv p q x y False gt ym lt xm yn′ zm xpm′* **unfolding** *id xpm-def*[*symmetric*]
   by (*auto simp add*: *monom-inv-def monom-list-mult-list-vars*)
  qed
 qed
 qed
qed *auto*

lemma *monom-inv-ConsD*: *monom-inv* (*x* # *xs*) $\Longrightarrow$ *monom-inv xs*
 by (*auto simp*: *monom-inv-def*)

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 (*induct m n rule*: *monom-mult-list.induct*)
 case (*2 x p m n*)
 thus *?case* **by** (*cases n*; *cases hd n*, *auto split*: *if-splits simp*: *sum-var-list-def*)
qed (*auto simp*: *sum-var-list-def*)

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 −
 from *eq sum-var-list-monom-mult-list*[*of m*] **show** *?thesis*
  by (*auto simp*: *eq-monom-sum-var-list*[*OF m1 m2*] *eq-monom-sum-var-list*[*OF monom-mult-list-inv*[*OF m m1*] *monom-mult-list-inv*[*OF m m2*]])
qed

lemma *monom-mult-list*[*simp*]: *eval-monom-list* α (*monom-mult-list m n*) = *eval-monom-list* α *m* ∗ *eval-monom-list* α *n*
 by (*induct m n rule*: *monom-mult-list.induct*, *auto split*: *list.splits prod.splits simp*: *field-simps power-add*)
end

declare *monom-mult-list.simps*[*simp del*]

typedef (**overloaded**) ′*v monom* = *Collect* (*monom-inv* :: ′*v* :: *linorder monom-list* ⇒ *bool*)
 by (*rule exI*[*of - Nil*], *auto simp*: *monom-inv-def*)

setup-lifting *type-definition-monom*

lift-definition *eval-monom* :: (′*v* :: *linorder*,′*a* :: *comm-semiring-1*)*assign* ⇒ ′*v monom* ⇒ ′*a*
 is *eval-monom-list* .

lift-definition *sum-var* :: ′*v* :: *linorder monom* ⇒ ′*v* ⇒ *nat* **is** *sum-var-list* .

instantiation *monom* :: (*linorder*) *comm-monoid-mult*
begin

lift-definition *times-monom* :: *'a monom* ⇒ *'a monom* ⇒ *'a monom* **is** *monom-mult-list*
  using *monom-mult-list-inv* **by** *auto*

lift-definition *one-monom* :: *'a monom* **is** *Nil*
  by (*auto simp*: *monom-inv-def*)

instance
proof
  fix *a b c* :: *'a monom*
  show $a * b * c = a * (b * c)$
   by (*transfer*, *auto simp*: *eq-monom-sum-var-list monom-mult-list-inv sum-var-list-monom-mult-list*)
  show $a * b = b * a$
   by (*transfer*, *auto simp*: *eq-monom-sum-var-list monom-mult-list-inv sum-var-list-monom-mult-list*)
  show $1 * a = a$
   by (*transfer*, *auto simp*: *eq-monom-sum-var-list monom-mult-list-inv sum-var-list-monom-mult-list monom-mult-list.simps*)
qed
end

lemma *eq-monom-sum-var*: *m* = *n* ⟷ (∀ *x*. *sum-var m x* = *sum-var n x*)
  by (*transfer*, *auto simp*: *eq-monom-sum-var-list*)

lemma *eval-monom-mult*[*simp*]: *eval-monom* α (*m* ∗ *n*) = *eval-monom* α *m* ∗ *eval-monom* α *n*
  by (*transfer*, *rule monom-mult-list*)

lemma *sum-var-monom-mult*: *sum-var* (*m* ∗ *n*) *x* = *sum-var m x* + *sum-var n x*
  by (*transfer*, *rule sum-var-list-monom-mult-list*)

lemma *monom-mult-inj*: **fixes** *m1* :: *- monom*
  shows *m* ∗ *m1* = *m* ∗ *m2* ⟹ *m1* = *m2*
  by (*transfer*, *rule monom-mult-list-inj*, *auto*)

lemma *one-monom-inv-sum-var-inv*[*simp*]: *sum-var 1 x* = *0*
  by (*transfer*, *auto simp*: *sum-var-list-def*)

lemma *eval-monom-1*[*simp*]: *eval-monom* α *1* = *1*
  by (*transfer*, *auto*)

lift-definition *var-monom* :: *'v* :: *linorder* ⇒ *'v monom* **is** λ *x*. [(*x*,*1*)]
  by (*auto simp*: *monom-inv-def*)

lemma *var-monom-1*[*simp*]: *var-monom x* ≠ *1*
  by (*transfer*, *auto*)

lemma *eval-var-monom*[*simp*]: *eval-monom* α (*var-monom* x) = α x
  by (*transfer*, *auto*)

lemma *sum-var-monom-var*: *sum-var* (*var-monom* x) y = (*if* $x = y$ *then 1 else 0*)
  by (*transfer*, *auto simp*: *sum-var-list-def*)

instantiation *monom* :: ({*equal*,*linorder*})*equal*
begin

lift-definition *equal-monom* :: *'a monom* ⇒ *'a monom* ⇒ *bool* **is** (=) **.**

instance by (*standard*, *transfer*, *auto*)
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. $c \neq 0$) ∧ *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
  unfolding *poly-const-def* **by** *auto*

lemma *poly-const-inv*: *poly-inv* (*poly-const* a)
  unfolding *poly-const-def poly-inv-def* **by** *auto*

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*
by (*induct mc1*, *auto simp*: *field-simps*)

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 (*induct p1 arbitrary*: *p2*)
case (*Cons mc p*)
obtain *m c* **where** *mc*: *mc* = (*m*,*c*) **by** (*cases mc*, *auto*)
hence *m*: *m* ∈ *poly-monoms* (*mc* # *p1*) **by** *auto*
show *?case*
proof (*cases List.extract* (λ *nd*. *fst nd* = *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** *d* **where** *q*: *p2* = *q1* @ (*m*,*d*) # *q2* **and** *res*: *res* = (*q1*,(*m*,*d*),*q2*) **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** *c*: *c* ≠ *0* **and** *mp*: ∀ *mm* ∈ *fst* ' *set p*. (¬ *mm* = *m*) **unfolding** *poly-inv-def* **by** *auto*
show *?case*
proof (*cases List.extract* (λ *mc*. *fst mc* = *m*) *q*)
case *None*
hence *mq*: ∀ *mm* ∈ *fst* ' *set q*. ¬ *mm* = *m* **by** (*auto simp*: *extract-None-iff*)
{
fix *mm*
assume *mm* ∈ *fst* ' *set* (*poly-add p q*)
then obtain *dd* **where** (*mm*,*dd*) ∈ *set* (*poly-add p q*) **by** *auto*
with *poly-add-monoms* **have** *mm* ∈ *poly-monoms p* ∨ *mm* ∈ *poly-monoms q* **by** *force*
hence ¬ *mm* = *m* **using** *mp mq* **by** *auto*

} **note** *main* = *this*
show *?thesis* **using** *Cons*(*1*)[*OF p Cons*(*3*)] **unfolding** *poly-inv-def* **using** *main* **by** (*auto simp add*: *None mc c*)
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** *d* **where** *q*: *q* = *q1* @ (*m*,*d*) # *q2* **and** *res*: *res* = (*q1*,(*m*,*d*),*q2*) **by** (*cases md*, *auto*)
from *q Cons*(*3*) **have** *q1q2*: *poly-inv* (*q1* @ *q2*) **unfolding** *poly-inv-def* **by** *auto*
from *Cons*(*1*)[*OF p q1q2*] **have** *main1*: *poly-inv* (*poly-add p* (*q1* @ *q2*)) **.**
{
fix *mm*
assume *mm* ∈ *fst* ‘ *set* (*poly-add p* (*q1* @ *q2*))
then obtain *dd* **where** (*mm*,*dd*) ∈ *set* (*poly-add p* (*q1* @ *q2*)) **by** *auto*
with *poly-add-monoms* **have** *mm* ∈ *poly-monoms p* ∨ *mm* ∈ *poly-monoms* (*q1* @ *q2*) **by** *force*
hence *mm* ≠ *m*
proof
assume *mm* ∈ *poly-monoms p*
thus *?thesis* **using** *mp* **by** *auto*
next
assume *member*: *mm* ∈ *poly-monoms* (*q1* @ *q2*)
from *member* **have** *mm* ∈ *poly-monoms q1* ∨ *mm* ∈ *poly-monoms q2* **by** *auto*
thus *mm* ≠ *m*
proof
assume *mm* ∈ *poly-monoms q2*
with *Cons*(*3*)[*simplified q*]
show *?thesis* **unfolding** *poly-inv-def* **by** *auto*
next
assume *mm* ∈ *poly-monoms q1*
with *Cons*(*3*)[*simplified q*]
show *?thesis* **unfolding** *poly-inv-def* **by** *auto*
qed
qed
} **note** *main2* = *this*
show *?thesis* **using** *main1*[*unfolded poly-inv-def*] *main2*
by (*auto simp*: *poly-inv-def mc Some res*)
qed
qed *simp*

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 *?case*
proof (*cases List.extract* (λ *mc*. *fst mc* = *m*) *q*)
case *None*

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** *d* **where** *q*: *q* = *q1* @ (*m*,*d*) # *q2* **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*: *field-simps*, *auto simp*: *field-simps id*)
qed
qed *simp*

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 (*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** *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*) **.**
{
fix *mm*
assume *mm* ∈ *fst* ‘ *set* (*monom-mult-poly* (*m*,*c*) *p*)
and *two*: *mm* = *m* ∗ *m'*
then obtain *dd* **where** *one*: (*mm*,*dd*) ∈ *set* (*monom-mult-poly* (*m*,*c*) *p*) **by** *auto*
have *poly-monoms* (*monom-mult-poly* (*m*,*c*) *p*) ⊆ (∗) *m* ‘ *poly-monoms p*
proof (*induct p*, *simp*)
case (*Cons md p*)
thus *?case*
by (*cases md*, *auto*)
qed
with *one* **have** *mm* ∈ (∗) *m* ‘ *poly-monoms p* **by** *force*
then obtain *mmm* **where** *mmm*: *mmm* ∈ *poly-monoms p* **and** *mm*: *mm* = *m*

∗ *mmm* **by** *blast*
from *Cons*(*2*)[*simplified md*] *mmm* **have** *not1*: ¬ *mmm* = *m′* **unfolding** *poly-inv-def* **by** *auto*
from *mm two* **have** *m* ∗ *mmm* = *m* ∗ *m′* **by** *simp*
from *monom-mult-inj*[*OF this*] *not1*
have *False* **by** *simp*
}
thus *?case*
by (*simp add*: *md prod*, *intro impI*, *auto simp*: *poly-inv-def prod*[*simplified poly-inv-def*])
qed

lemma *monom-mult-poly*[*simp*]: *eval-poly* α (*monom-mult-poly mc p*) = *eval-monomc* α *mc* ∗ *eval-poly* α *p*
proof (*cases mc*)
case (*Pair m c*)
show *?thesis*
proof (*simp add*: *Pair*, *induct p*)
case (*Cons nd q*)
obtain *n d* **where** *nd*: *nd* = (*n*,*d*) **by** (*cases nd*, *auto*)
show *?case*
proof (*cases c* ∗ *d* = *0*)
case *False*
thus *?thesis* **by** (*simp add*: *nd Cons field-simps*)
next
case *True*
let *?l* = *c* ∗ (*d* ∗ (*eval-monom* α *m* ∗ *eval-monom* α *n*))
have *?l* = (*c* ∗ *d*) ∗ (*eval-monom* α *m* ∗ *eval-monom* α *n*)
by (*simp only*: *field-simps*)
also have . . . = *0* **by** (*simp only*: *True*, *simp add*: *field-simps*)
finally have *l*: *?l* = *0* **.**
show *?thesis*
by (*simp add*: *nd Cons True*, *simp add*: *field-simps l*)
qed
qed *simp*
qed

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*
unfolding *poly-minus-def* **by** *simp*

lemma *poly-minus-inv*: *poly-inv f* =⇒ *poly-inv g* =⇒ *poly-inv* (*poly-minus f g*)
unfolding *poly-minus-def* **by** (*intro poly-add-inv monom-mult-poly-inv*)

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*)
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** *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 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*
by (*induct p*, *auto simp*: *field-simps*)

declare *poly-mult.simps*[*simp del*]

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

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

lemma *poly-one*[*simp*]: *eval-poly* α *one-poly* = *1*
unfolding *one-poly-def* **by** *simp*

lemma *poly-zero-add*: *poly-add zero-poly p* = *p* **unfolding** *zero-poly-def* **using** *poly-add.simps* **by** *auto*

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

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*
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*⟧ ⟹ *p1 =p p3*
unfolding *eq-poly-def* **by** *auto*

lemma *poly-add-comm*: *poly-add p q =p poly-add q p* **unfolding** *eq-poly-def* **by** (*auto simp*: *field-simps*)

lemma *poly-add-assoc*: *poly-add p1* (*poly-add p2 p3*) *=p poly-add* (*poly-add p1 p2*) *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*)

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*
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*)
by (*induct p rule*: *poly-of.induct*,
simp add: *poly-inv-def monom-inv-def*,
simp add: *poly-inv-def monom-inv-def*,
simp add: *zero-poly-inv*,
simp add: *poly-add-inv*,
simp add: *one-poly-inv*,
simp add: *poly-mult-inv*)

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*
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-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* **.**

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 (*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

lemma *monom-subst-inv*: **assumes** *sub*: ⋀ *x*. *poly-inv* (σ *x*)
shows *poly-inv* (*monom-subst* σ *m*)
unfolding *monom-subst-def* **by** (*rule monom-list-subst-inv*[*OF sub*])

lemma *monom-subst*[*simp*]: *eval-poly* α (*monom-subst* σ *m*) = *eval-monom* (λ *v*. *eval-poly* α (σ *v*)) *m*
unfolding *monom-subst-def*
proof (*transfer fixing*: α σ, *clarsimp*)
fix *m*
show *monom-inv m* ⟹ *eval-poly* α (*monom-list-subst* σ *m*) = *eval-monom-list* (λ*v*. *eval-poly* α (σ *v*)) *m*

by (*induct m*, *simp add*: *one-poly-def*, *auto simp*: *field-simps monom-inv-ConsD*)
qed

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*)
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* [(*1*,*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 (*transfer fixing*: *f g q*, *clarsimp*)
fix *m*
show *eval-monom-list f m* = *eval-monom-list* (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*
qed
show *?case*
by (*simp add*: *mc Cons id*, *simp add*: *field-simps*)

qed

lift-definition *monom-vars-list* :: *'v* :: *linorder monom* ⇒ *'v list* **is** *map fst* **.**

lemma *monom-vars-list-subst*: **assumes** $\bigwedge$ *w*. *w* ∈ *set* (*monom-vars-list m*) ⟹ *f w* = *g w*
  shows *monom-subst f m* = *monom-subst g m*
  unfolding *monom-subst-def* **using** *assms*
proof (*transfer fixing*: *f g*)
  fix *m* :: *'a monom-list*
  assume *eq*: $\bigwedge$*w*. *w* ∈ *set* (*map fst m*) ⟹ *f w* = *g w*
  thus *monom-list-subst f m* = *monom-list-subst g m*
  proof (*induct m*)
    case (*Cons wn m*)
    hence *rec*: *monom-list-subst f m* = *monom-list-subst g m* **and** *eq*: *f* (*fst wn*) = *g* (*fst wn*) **by** *auto*
    show *?case*
    proof (*cases wn*)
      case (*Pair w n*)
      with *eq rec* **show** *?thesis* **by** *auto*
    qed
  qed *simp*
qed

lemma *eval-monom-vars-list*: **assumes** $\bigwedge$ *x*. *x* ∈ *set* (*monom-vars-list xs*) ⟹ α *x* = β *x*
  shows *eval-monom* α *xs* = *eval-monom* β *xs* **using** *assms*
proof (*transfer fixing*: α β)
  fix *xs* :: *'a monom-list*
  assume *eq*: $\bigwedge$*w*. *w* ∈ *set* (*map fst xs*) ⟹ α *w* = β *w*
  thus *eval-monom-list* α *xs* = *eval-monom-list* β *xs*
  proof (*induct xs*)
    case (*Cons xi xs*)
    hence *IH*: *eval-monom-list* α *xs* = *eval-monom-list* 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 *simp*
qed

definition *monom-vars* **where** *monom-vars m* = *set* (*monom-vars-list m*)

lemma *monom-vars-list-1*[*simp*]: *monom-vars-list 1* = []
  by *transfer auto*

lemma *monom-vars-list-var-monom*[*simp*]: *monom-vars-list* (*var-monom x*) = [*x*]

  by *transfer auto*

lemma *monom-vars-eval-monom*:
($\bigwedge$ *x*. *x* ∈ *monom-vars m* ⟹ *f x* = *g x*) ⟹ *eval-monom f m* = *eval-monom g m*
by (*rule eval-monom-vars-list*, *auto simp*: *monom-vars-def*)

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*
unfolding *poly-vars-list-def poly-vars-def* **by** *auto*

lemma *poly-vars*: **assumes** *eq*: $\bigwedge$ *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** $\bigwedge$ *w*. *w* ∈ *set* (*monom-vars-list m*) ⟹ *f w* = *g w* **unfolding** *poly-vars-def* **by** *auto*
hence *monom-subst f m* = *monom-subst g m*
by (*rule monom-vars-list-subst*)
with *rec Pair* **show** *?thesis* **by** *auto*
qed
qed *simp*

lemma *poly-var*: **assumes** *pv*: *v* ∉ *poly-vars p* **and** *diff*: $\bigwedge$ *w*. *v* ≠ *w* ⟹ *f w* = *g w*
shows *poly-subst f p* = *poly-subst g p*
proof (*rule poly-vars*)
fix *w*
assume *w* ∈ *poly-vars p*
thus *f w* = *g w* **using** *pv diff* **by** (*cases v* = *w*, *auto*)
qed

lemma *eval-poly-vars*: **assumes** $\bigwedge$ *x*. *x* ∈ *poly-vars p* ⟹ α *x* = β *x*
shows *eval-poly* α *p* = *eval-poly* β *p*
using *assms*
proof (*induct p*)
case *Nil* **thus** *?case* **by** *simp*
next

case (*Cons m p*)
from *Cons*(*2*) **have** $\bigwedge$ *x*. *x* ∈ *poly-vars p* ⟹ α *x* = β *x* **unfolding** *poly-vars-def*
by *auto*
from *Cons*(*1*)[*OF this*] **have** *IH*: *eval-poly* α *p* = *eval-poly* β *p* **.**
obtain *xs c* **where** *m*: *m* = (*xs*,*c*) **by** *force*
from *Cons*(*2*) **have** $\bigwedge$ *x*. *x* ∈ *set* (*monom-vars-list xs*) ⟹ α *x* = β *x* **unfolding**
poly-vars-def m **by** *auto*
hence *eval-monom* α *xs* = *eval-monom* β *xs*
by (*rule eval-monom-vars-list*)
thus *?case* **unfolding** *eval-poly.simps IH m* **by** *auto*
qed

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* ≥*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*
unfolding *poly-ge-def* **using** *ge-trans* **by** *blast*

lemma *pos-assign-monom-list*: **fixes** α :: (′*v* :: *linorder*, ′*a* :: *poly-carrier*)*assign*
assumes *pos*: *pos-assign* α
shows *eval-monom-list* 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 Cons*]]
show *?thesis*
by (*simp add*: *Pair*)
qed
qed

lemma *pos-assign-monom*: **fixes** α :: (*'v* :: *linorder*, *'a* :: *poly-carrier*)*assign*
assumes *pos*: *pos-assign* α
shows *eval-monom* α *m* $\geq$ *0*
by (*transfer fixing*: α, *rule pos-assign-monom-list*[*OF pos*])

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

lemma *poly-add-ge-mono*: **assumes** *p1* $\geq$*p p2* **shows** *poly-add p1 q* $\geq$*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* $\geq$*p p2* **and** *q* $\geq$*p zero-poly*
shows *poly-mult p1 q* $\geq$*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* :: (*'v* :: *linorder*,*'a*)*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-gt-imp-poly-ge*: *p* >*p q* ⟹ *p* $\geq$*p q* **unfolding** *poly-ge-def poly-gt-def*
using *gt-imp-ge* **by** *blast*

abbreviation *poly-GT* :: (*'v* :: *linorder*,*'a*)*poly rel*
where *poly-GT* ≡ {(*p*,*q*) | *p q*. *p* >*p q* ∧ *q* $\geq$*p zero-poly*}

lemma *poly-compat*: ⟦*p1* $\geq$*p p2*; *p2* >*p p3*⟧ ⟹ *p1* >*p p3*
unfolding *poly-ge-def poly-gt-def* **using** *compat* **by** *blast*

lemma *poly-compat2*: ⟦*p1* >*p p2*; *p2* $\geq$*p p3*⟧ ⟹ *p1* >*p p3*
unfolding *poly-ge-def poly-gt-def* **using** *compat2* **by** *blast*

lemma *poly-gt-trans*[*trans*]: $[\![p1 >p\ p2;\ p2 >p\ p3]\!] \Longrightarrow p1 >p\ p3$
unfolding *poly-gt-def* **using** *gt-trans* **by** *blast*

lemma *poly-GT-SN*: *SN poly-GT*
proof
fix *f* :: *nat* ⇒ (′*c* :: *linorder*,′*a*)*poly*
assume *f*: ∀ *i*. (*f i*, *f* (*Suc i*)) ∈ *poly-GT*
have *pos*: *pos-assign* ((λ *x*. *0*) :: (′*v*,′*a*)*assign*) (**is** *pos-assign ?ass*) **unfolding** *pos-assign-def* **using** *ge-refl* **by** *auto*
obtain *g* **where** *g*: ⋀ *i*. *g i* = *eval-poly ?ass* (*f i*) **by** *auto*
from *f pos* **have** ∀ *i*. *g* (*Suc i*) ≥ *0* ∧ *g i* ≻ *g* (*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-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 (*atomize*(*full*), *induct m*)
case *Nil* **show** *?case* **using** *one-ge-zero* **by** (*auto simp*: *ge-refl*)
next
case (*Cons xd m*)
hence *IH1*: *eval-monom-list f m* ≥ *eval-monom-list g m* **and** *IH2*: *eval-monom-list g m* ≥ *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* ≥ *g x* ^ *d* **and** *gd*: *g x* ^ *d* ≥ *0* **by** *auto*
show *?case* **unfolding** *xd eval-monom-list.simps*
proof (*rule conjI*, *rule ge-trans*[*OF times-left-mono*[*OF pow-ge-zero IH1*] *times-right-mono*[*OF IH2 fgd*]])
show *f x* ≥ *0* **by** (*rule ge-trans*[*OF fg g*])
show *eval-monom-list g m* ∗ *g x* ^ *d* ≥ *0*
by (*rule mult-ge-zero*[*OF IH2 gd*])
qed
qed

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*
by (*atomize*(*full*), *transfer fixing*: *f g*, *insert eval-monom-list-mono*[*of g f*, *OF fg g*], *auto*)

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*
unfolding *poly-ge-def poly-subst*
proof (*intro allI impI*, *rule p*[*unfolded poly-weak-mono-all-def*, *rule-format*])
fix α :: (′*c*,′*b*)*assign* **and** *x*
show *pos-assign* α ⟹ *eval-poly* α (*f x*) ≥ *eval-poly* α (*g x*) **using** *ge*[*of x*] **unfolding** *poly-ge-def* **by** *auto*
next
fix α :: (′*c*,′*b*)*assign*
assume *alpha*: *pos-assign* α
show *pos-assign* (λ*v*. *eval-poly* α (*g v*))
unfolding *pos-assign-def*
proof
fix *x*
show *eval-poly* α (*g x*) ≥ *0*
using *ge*[*of x*] **unfolding** *poly-ge-def zero-poly-def* **using** *alpha* **by** *auto*
qed
qed

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*
unfolding *poly-ge-def poly-subst*
proof (*intro allI impI*, *rule p*[*unfolded poly-weak-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* (λ*v*. *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

next
  fix *α* :: (*'c*,*'b*)*assign* **and** *x*
  assume *v*: *v* ≠ *x*
  show *pos-assign α* ⟹ *eval-poly α* (*f x*) = *eval-poly α* (*g x*) **using** *fgw*[*OF v*]
unfolding *poly-ge-def* **by** *auto*
qed

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*: ⋀ *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 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
next
  fix *α* :: (*'c*,*'b*)*assign* **and** *x*
  assume *v*: *v* ≠ *x*
  show *pos-assign α* ⟹ *eval-poly α* (*f x*) = *eval-poly α* (*g x*) **using** *fgw*[*OF v*]
unfolding *poly-ge-def* **by** *auto*
qed

lemma *poly-weak-mono*: **fixes** *p* :: (*'v* :: *linorder*,*'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* **.**
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* :: (*'v* :: *linorder*,*'a* :: *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* :: (*'v*,*'a*)*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* :: *linorder*,*'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*
    have *poly-subst ?id p* ≥*p poly-subst ?z p*
      by (*rule poly-weak-mono-all-E*[*OF mono*],
        simp, *simp add*: *poly-ge-def zero-poly-def pos-assign-def*)
     hence *eval-poly* α (*poly-subst ?id p*) ≥ *eval-poly* α (*poly-subst ?z p*) (**is** - ≥ *?res*)
      unfolding *poly-ge-def* **using** *pos* **by** *simp*
    also have *?res* = *eval-poly* (λ *w*. *0*) *p* **by** (*simp add*: *poly-subst zero-poly-def*)
    also have ... ≥ *0* **by** (*rule pos-at-zero*)
    finally show *?thesis* **by** (*simp add*: *poly-subst*)
  qed
qed

context *poly-order-carrier*
begin

definition *poly-strict-mono* :: (*'v* :: *linorder*,*'a*)*poly* ⇒ *'v* ⇒ *bool* **where**
  poly-strict-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-strict-mono-E*: **assumes** *p*: *poly-strict-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*
  unfolding *poly-gt-def poly-subst*
proof (*intro allI impI*, *rule p*[*unfolded poly-strict-mono-def*, *rule-format*])
  fix α :: (*'c*,*'a*)*assign*
  show *pos-assign* α ⟹ *eval-poly* α (*f v*) ≻ *eval-poly* α (*g v*) **using** *fgv* **unfolding** *poly-gt-def* **by** *auto*
next
  fix α :: (*'c*,*'a*)*assign*
  assume *alpha*: *pos-assign* α
  show *pos-assign* (λ*v*. *eval-poly* α (*g v*))

unfolding *pos-assign-def*
proof
fix *x*
show *eval-poly* α (*g x*) $\geq$ *0*
using *g*[*of x*] **unfolding** *poly-ge-def zero-poly-def* **using** *alpha* **by** *auto*
qed
next
fix α :: ('*c*,'*a*)*assign* **and** *x*
assume *v*: $v \neq x$
show *pos-assign* $\alpha \Longrightarrow$ *eval-poly* α (*f x*) = *eval-poly* α (*g x*) **using** *fgw*[*OF v*]
unfolding *poly-ge-def* **by** *auto*
qed

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

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 (*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* $\succ$ *eval-poly* α *p2* **unfolding** *poly-gt-def* **by** *simp*
from *mono p* **have** *one*: *eval-poly* α *q* $\geq$ *1* **unfolding** *poly-ge-def one-poly-def*
by *auto*
show *eval-poly* α (*poly-mult p1 q*) $\succ$ *eval-poly* α (*poly-mult p2 q*)
using *times-gt-mono*[*OF gt one*] **by** *simp*
qed
end

18.6 Degree of polynomials

definition *monom-list-degree* :: '*v monom-list* $\Rightarrow$ *nat* **where**
monom-list-degree xps $\equiv$ *sum-list* (*map snd xps*)

lift-definition *monom-degree* :: '*v* :: *linorder monom* $\Rightarrow$ *nat* **is** *monom-list-degree*
.

definition *poly-degree* :: (-,'*a*) *poly* $\Rightarrow$ *nat* **where**
poly-degree p $\equiv$ *max-list* (*map* (λ (*m*,*c*). *monom-degree m*) *p*)

definition *poly-coeff-sum* :: ('*v*,'*a* :: *ordered-ab-semigroup*) *poly* $\Rightarrow$ '*a* **where**
poly-coeff-sum p $\equiv$ *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*
unfolding *monom-list-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 *monom-list-var-monom*[*simp*]: *monom-list* (*var-monom x*) = [(*x*,*1*)]
by (*transfer*, *simp*)

lemma *monom-list-1*[*simp*]: *monom-list 1* = []
by (*transfer*, *simp*)

lemma *monom-degree*: *eval-monom* (λ -. *x*) *m* = *x* ^ *monom-degree m*
by (*transfer*, *rule monom-list-degree*)

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*: *mc* = (*m*,*c*) **by** *force*
from *ge-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*: *mc 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 ge-trans*[*OF*
times-left-mono[*OF pow-ge-zero*[*OF ge-trans*[*OF x one-ge-zero*]] *c*]
times-right-mono[*OF poly-coeff-sum pow-mono-exp*[*OF x d*]]] *poly-degree*[*OF x*]])

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 (*cases List.extract* (λ (*n*,-). *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* **where** *p*: *p* = *p1* @ (*m*,*a*) # *p2* **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 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*) *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* **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*) *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** *d* **where** *q*: *q* = *q1* @ (*m*,*d*) # *q2* **and** *res*: *res* = (*q1*,(*m*,*d*),*q2*)
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* **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*) *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 (*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
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*) *q*)
case *None*
with *Cons*(*2*) **have** *rec*: *check-poly-ge p q* **and** *c*: *c* ≥ *0* **using** *mc* **by** *auto*
from *Cons*(*1*)[*OF rec*] **have** *rec*: *p* ≥*p q* **.**
show *?thesis*
proof (*simp only*: *mc*, *unfold poly-ge-def*, *intro allI impI*)
fix α :: ('*v*,'*a*)*assign*
assume *pos*: *pos-assign* α
have *ge*: *eval-monom* α *m* ∗ *c* ≥ *0*
using *times-right-mono*[*OF pos-assign-monom*[*OF pos*, *of m*] *c*] **by** *simp*
from *rec* **have** *pq*: *eval-poly* α *p* ≥ *eval-poly* α *q* **unfolding** *poly-ge-def* **using** *pos* **by** *auto*
show *eval-poly* α ((*m*,*c*) # *p*) ≥ *eval-poly* α *q*
using *ge-trans*[*OF plus-left-mono*[*OF ge*] *plus-right-mono*[*OF pq*]] **by** *simp*
qed
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** *d* **where** *q*: *q* = *q1* @ (*m*,*d*) # *q2* **and** *res*: *res* = (*q1*,(*m*,*d*),*q2*)
by (*cases md*, *auto*)
from *Cons*(*2*) *Some mc res* **have** *rec*: *check-poly-ge p* (*q1* @ *q2*) **and** *c*: *c* ≥ *d* **by** *auto*

from *Cons*(*1*)[*OF rec*] **have** *p*: *p* ≥*p q1* @ *q2* **.**
show *?thesis*
proof (*simp only*: *mc*, *unfold poly-ge-def*, *intro allI impI*)
fix α :: (′*v*,′*a*)*assign*
assume *pos*: *pos-assign* α
have *ge*: *eval-monom* α *m* ∗ *c* ≥ *eval-monom* α *m* ∗ *d*
using *times-right-mono*[*OF pos-assign-monom*[*OF pos*, *of m*] *c*]
by *simp*
from *p* **have** *ge2*: *eval-poly* α *p* ≥ *eval-poly* α (*q1* @ *q2*) **unfolding** *poly-ge-def*
using *pos* **by** *auto*
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*)
qed
qed
qed

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*
unfolding *poly-weak-mono-all-def*
proof (*intro allI impI*)
fix *f g* :: (′*v*,′*a*)*assign*
assume *fg*: ∀ *x*. *f x* ≥ *g x*
and *pos*: *pos-assign g*
hence *fg*: ⋀ *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* **unfolding** *check-poly-weak-mono-all-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*
show *?case*
proof (*cases mc*)
case (*Pair m c*)
with *Cons* **have** *c*: *c* ≥ *0* **by** *auto*
show *?thesis* **unfolding** *Pair 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* ≥ *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** $\bigwedge$ *m c*. (*m*,*c*) ∈ *set p* ⟹ *c* ≥ *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 $p(\ldots v+1\ldots) \geq p(\ldots v\ldots)$

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*
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*: $\bigwedge$ *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* = (((+) *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* = (((+) *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*
        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*
            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′* = (λ *w*. *if* (*w* = *v*) *then 1* + *g w else g w*) **by** *auto*
        have (*1* :: *′a*) + *g v* ≥ *1* + *0*
          by (*rule plus-right-mono*, *simp add*: *less*(*3*)[*unfolded pos-assign-def*])
        also have *1* + (*0* :: *′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 $v \neq w$
hence $f\ w = g'\ w$
unfolding g' by (*simp add*: *less*)
} note $w = this$
have *eq*: $f\ v = ((+)\ (1 :: 'a)\ \widehat{}\widehat{}\ Suc\ kk)\ ((g'\ v))$
by (*simp add*: *less*(4) g' *Suc*, *rule arg-cong*[**where** $f = (+)\ 1$], *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* $\geq$ *eval-poly g' p* by *simp*
{
fix w
assume $v \neq w$
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* $\geq$ *eval-poly g p* by *simp*
show *?thesis* by (*rule ge-trans*[*OF rec1 rec2*])
qed
qed
qed
qed

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*
unfolding *poly-weak-anti-mono-def*
proof (*intro allI impI*)
fix $f\ g :: ('v,'a)assign$
assume *fgw*: $\forall\ w.\ (v \neq w \longrightarrow f\ w = g\ w)$
and *gass*: *pos-assign g*
and v: $f\ v \geq g\ v$
from *fgw* **have** w: $\bigwedge w.\ v \neq w \Longrightarrow f\ w = g\ w$ **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*: $f\ v = (((+)\ 1)\widehat{}\widehat{}k')\ (g\ v)$ **by** *auto*
show *eval-poly g p* $\geq$ *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* = (((+) *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*
with *less* **have** *f w* = *g w* **by** *simp*
thus *?thesis* **by** (*simp add*: *False*)
qed
qed
have *eval-poly g p* ≥ *eval-poly g ?p1* **using** *ge less* **unfolding** *poly-ge-def* **by** *simp*
with *id1* **show** *?thesis* **by** *simp*
next
case (*Suc kk*)
obtain *g′* **where** *g′*: *g′* = (λ *w*. *if* (*w* = *v*) *then 1* + *g w else g w*) **by** *auto*
have (*1* :: ′*a*) + *g v* ≥ *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 ... ≥ *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* = ((+) (*1* :: ′*a*) ^^ *Suc kk*) ((*g′ v*))
by (*simp add*: *less*(*4*) *g′ Suc*, *rule arg-cong*[**where** *f* = (+) *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 g′ p* ≥ *eval-poly f 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 rec2 rec1*])
qed
qed
qed
qed

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 (*cases discrete*)
case *False*
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

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
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
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 (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, fast)
        thus ?thesis by (simp add: ge-refl)
      qed
    qed
  qed (simp add: ge-refl)
qed

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*
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* :: *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*
unfolding *poly-weak-anti-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* ⟹ *0* ≥ *c* ∨ *v* ∉ *monom-vars m* **unfolding** *check-poly-weak-anti-mono-def* **by** (*auto simp*: *list-all-iff*)
thus *eval-poly g p* ≥ *eval-poly f p*
proof (*induct p*)
case *Nil* **thus** *?case* **by** (*simp add*: *ge-refl*)
next
case (*Cons mc p*)
hence *IH*: *eval-poly g p* ≥ *eval-poly f p* **by** *auto*
obtain *m c* **where** *mc*: *mc* = (*m*,*c*) **by** *force*
with *Cons* **have** *c*: *0* ≥ *c* ∨ *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 g m* ∗ *c* ≥ *eval-monom f m* ∗ *c*
proof
assume *c*: *0* ≥ *c*
show *?thesis*
proof (*rule times-left-anti-mono*[*OF eval-monom-mono*(*1*)[*OF - g*] *c*])
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*, *fast*)

thus *?thesis* **by** (*simp add*: *ge-refl*)
qed
qed
qed
qed

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*
unfolding *check-poly-weak-anti-mono-smart-def*
using *check-poly-weak-anti-mono*[*of p v*] *check-poly-weak-anti-mono-discrete*[*of p v*]
by (*cases discrete*, *auto*)

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 −
have *m*: *m* = [(*x*,*n*)] **using** *assms*
by (*induct x m rule*: *univariate-power-list.induct*, *auto split*: *if-splits*)
show *eval-monom-list* α *m* = ((α *x*)^*n*) *sum-var-list m* = (λ *y*. *if x* = *y then n else 0*)
n ≥ *1* **using** *assms*(*1*)
unfolding *m monom-inv-def* **by** (*auto simp*: *sum-var-list-def*)
qed

lift-definition *univariate-power* :: ′*v* :: *linorder* ⇒ ′*v monom* ⇒ *nat option*
is *univariate-power-list* **.**

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*
by (*atomize*(*full*), *insert assms*, *transfer*, *auto dest*: *univariate-power-list*)

lemma *univariate-power-var-monom*: *univariate-power y* (*var-monom x*) = (*if x* = *y then Some 1 else None*)
by (*transfer*, *auto*)

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 −
from *check*[*unfolded check-monom-strict-mono-def*] **obtain** *n* **where**
uni: *univariate-power v m* = *Some n* **and** *1*: ¬ *power-mono* ⟹ *n* = *1*
by (*auto split*: *option.splits*)
from *univariate-power*[*OF uni*]
have *n1*: *n* ≥ *1* **and** *eval*: *eval-monom a m* = *a v* ^ *n* **for** *a* :: (*'v*,*'a*)*assign*
by *auto*
show *?thesis*
proof (*cases power-mono*)
case *False*
with *gt 1*[*OF this*] **show** *?thesis* **unfolding** *eval* **by** *auto*
next

case *True*
from *power-mono*[*OF True gt ge n1*] **show** *?thesis* **unfolding** *eval* **.**
qed
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 *fgw*: $\forall$ *w*. ($v \neq w \longrightarrow f\ w = g\ w$)
and *pos*: *pos-assign g*
and *fgv*: $f\ v \succ g\ v$
from *pos*[*unfolded pos-assign-def*] **have** *g*: $\bigwedge x.\ g\ x \geq 0$ **..**
{
fix *w*
have $f\ w \geq g\ w$
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* $\succ$ *?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* $c \geq 1$ $\wedge$ *check-monom-strict-mono power-mono m v*)
case *True*
hence *c*: $c \geq 1$ **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*]
have *gt*: *eval-monom f m* $*$ *c* $\succ$ *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* $\geq$ *?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* :: *linorder*,*'a*)*poly*
assumes *check-poly-gt gt p q* **shows** *p* >*p q*
proof −
obtain *a1 p1* **where** *p*: *poly-split 1 p* = (*a1*,*p1*) **by** *force*
obtain *b1 q1* **where** *q*: *poly-split 1 q* = (*b1*,*q1*) **by** *force*
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 (*unfold poly-gt-def*, *intro impI allI*)
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*
by (*simp add*: *poly-split*[*OF p*, *unfolded eq-poly-def*] *poly-split*[*OF q*, *unfolded eq-poly-def*] *gt*)
qed
qed

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*
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 *v*: *f v* ≻ *g v*
from *gass* **have** *g*: ⋀ *x*. *g x* ≥ *0* **unfolding** *pos-assign-def* **..**

from *fgw* **have** *w*: $\bigwedge$ *w*. $v \neq w \Longrightarrow f\ w = g\ w$ **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*: $f\ v = (((+)\ 1)$^^$k')\ (g\ v)$ **by** *auto*
{
  assume $k' = 0$
  from *v*[*unfolded id this*] **have** $g\ v \succ g\ v$ **by** *simp*
  hence *False* **using** *SN g*[*of v*] **unfolding** *SN-defs* **by** *auto*
}
with *id* **obtain** *k* **where** *id*: $f\ v = (((+)\ 1)$^^$(Suc\ k))\ (g\ v)$ **by** (*cases k'*, *auto*)
with *w gass*
show *eval-poly f p* $\succ$ *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*(*4*) **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*
        with *less* **have** $f\ w = g\ w$ **by** *simp*
        thus *?thesis* **by** (*simp add*: *False*)
      qed
    qed
    have *eval-poly g ?p1* $\succ$ *eval-poly g p* **using** *gt less* **unfolding** *poly-gt-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 $(1 :: 'a) + g\ v \geq 1 + 0$
      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'pos*: *pos-assign g'* **using** *less*(*3*) **unfolding** *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* = *this*
have *eq*: *f v* = ((+) (*1* :: *′a*) ^^ *Suc kk*) ((*g′ v*))
by (*simp add*: *less*(*4*) *g′ Suc*, *rule arg-cong*[**where** *f* = (+) *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 gt-trans*[*OF rec1 rec2*])
qed
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

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*
.

lemma *shows-prec-monom-append* [*show-law-simps*]:
shows-prec d (*m* :: *'a monom*) (*r* @ *s*) = *shows-prec d m r* @ *s*
proof (*transfer fixing*: *d r s*)
fix *m* :: *'a monom-list*
show *shows-monom-list m* (*r* @ *s*) = *shows-monom-list m r* @ *s*
by (*induct m arbitrary*: *r s rule*: *shows-monom-list.induct*, *auto simp*: *show-law-simps*)
qed

definition *shows-list* (*ts* :: *'a monom list*) = *showsp-list shows-prec 0 ts*

instance by (*standard*, *auto simp*: *show-law-simps shows-list-monom-def*)
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*
by (*unfold eq-monom-sum-var*, *auto simp*: *sum-var-monom-mult sum-var-monom-var*)

lemma *monom-list-x-x*[*simp*]: *monom-list* (*var-monom x* ∗ *var-monom x*) = [(*x*,*2*)]
by (*transfer*, *auto simp*: *monom-mult-list.simps*)

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 −
note [*simp*] = *poly-add.simps poly-mult.simps monom-mult-poly.simps zero-poly-def one-poly-def*
extract-def check-poly-strict-mono-discrete-def poly-subst.simps monom-subst-def poly-power.simps
check-poly-gt-def poly-split-def check-poly-ge.simps
show *?thesis*
proof (*cases a* = *0*)
case *True*
with *b* **have** *b*: *b* > *0* ∧ *b* ≠ *0* **by** *auto*
show *?thesis* **using** *b True* **by** *simp*
next
case *False*
have [*simp*]: *2* = *Suc* (*Suc 0*) **by** *simp*
show *?thesis* **using** *False a b* **by** *simp*
qed
qed

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 −
note [*simp*] = *poly-add.simps poly-mult.simps monom-mult-poly.simps zero-poly-def one-poly-def*
extract-def check-poly-weak-mono-discrete-def poly-subst.simps monom-subst-def poly-power.simps
check-poly-gt-def poly-split-def check-poly-ge.simps
show *?thesis*
proof (*cases a* = *0*)
case *True*
with *b* **have** *b*: *0* ≤ *b* **by** *auto*
show *?thesis* **using** *b True* **by** *simp*
next
case *False*
have [*simp*]: *2* = *Suc* (*Suc 0*) **by** *simp*
show *?thesis* **using** *False a b* **by** *simp*
qed

qed

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.