Executable multivariate polynomials

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

April 20, 2020

Abstract

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

Our formalization was performed as part of the IsaFoR/CeTA-system [5] 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

1 Utilities 7
  1.1 Lists ......................................................... 11
  1.2 Sums and Products ........................................ 16

2 An abstract type for multivariate polynomials 18
  2.1 Abstract type definition .................................. 18
  2.2 Additive structure ........................................ 18

*Supported by the Austrian Science Fund (FWF): grant no. W1214-N15, project DK1
1http://cl-informatik.uibk.ac.at/software/ceta
2.3 Multiplication by a coefficient ........................................ 19
2.4 Multiplicative structure .................................................. 20
2.5 Monomials ........................................................................ 21
2.6 Constants and Indeterminates ............................................. 22
2.7 Integral domains .................................................................. 23
2.8 Monom coefficient lookup .................................................. 23
2.9 Insertion morphism ............................................................. 23
2.10 Degree ............................................................................. 26
2.11 Pseudo-division of polynomials ......................................... 26
2.12 Primitive poly, etc .............................................................. 28

3 MPpoly Mapping extension .................................................. 29

4 MPoly extension ................................................................... 32

5 Nested MPoly ...................................................................... 38

6 Abstract Power-Products ..................................................... 45
  6.1 Constant Keys .................................................................. 46
  6.2 Constant except ............................................................... 47
  6.3 'Divisibility' on Additive Structures ................................ 53
  6.4 Dickson Classes ............................................................... 59
  6.5 Additive Linear Orderings ............................................... 64
  6.6 Ordered Power-Products .................................................. 67
  6.7 Functions as Power-Products ............................................ 72
    6.7.1 'a ⇒ 'b belongs to class comm-powerprod ............... 73
    6.7.2 'a ⇒ 'b belongs to class ninv-comm-monoid-add ....... 73
    6.7.3 'a ⇒ 'b belongs to class lcs-powerprod ................. 73
    6.7.4 'a ⇒ 'b belongs to class ulcs-powerprod ............... 75
    6.7.5 Power-products in a given set of indeterminates ....... 75
    6.7.6 Dickson's lemma for power-products in finitely many indeterminates ........................................... 76
    6.7.7 Lexicographic Term Order ......................................... 78
    6.7.8 Degree ....................................................................... 84
    6.7.9 General Degree-Orders ............................................. 86
    6.7.10 Degree-Lexicographic Term Order .......................... 90
    6.7.11 Degree-Reverse-Lexicographic Term Order .......... 91
  6.8 Type poly-mapping ............................................................ 93
    6.8.1 'a ⇒_0 'b belongs to class comm-powerprod .......... 94
    6.8.2 'a ⇒_0 'b belongs to class ninv-comm-monoid-add .... 94
    6.8.3 'a ⇒_0 'b belongs to class lcs-powerprod ............... 94
    6.8.4 'a ⇒_0 'b belongs to class ulcs-powerprod ............. 95
    6.8.5 Power-products in a given set of indeterminates ....... 95
6.8.6 Dickson’s lemma for power-products in finitely many indeterminates . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
6.8.7 Lexicographic Term Order . . . . . . . . . . . . . . . . 98
6.8.8 Degree . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
6.8.9 General Degree-Orders . . . . . . . . . . . . . . . . . . 100
6.8.10 Degree-Lexicographic Term Order . . . . . . . . . . 101
6.8.11 Degree-Reverse-Lexicographic Term Order . . . . . . 102

7 Modules over Commutative Rings 103
7.1 Submodules Spanned by Sets of Module-Elements . . . . . 103

8 Ideals over Commutative Rings 108

9 Type-Class-Multivariate Polynomials 113
9.1 keys . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
9.2 Monomials . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
9.3 Vector-Polynomials . . . . . . . . . . . . . . . . . . . . . . . . 118
9.3.1 Additive Structure of Terms . . . . . . . . . . . . . . . . . 119
9.3.2 Projections and Conversions . . . . . . . . . . . . . . . . 124
9.4 Scalar Multiplication by Monomials . . . . . . . . . . . . . . 130
9.5 Component-wise Lifting . . . . . . . . . . . . . . . . . . . . . 134
9.6 Component-wise Multiplication . . . . . . . . . . . . . . . . . 136
9.7 Scalar Multiplication . . . . . . . . . . . . . . . . . . . . . . . 139
9.8 Sums and Products . . . . . . . . . . . . . . . . . . . . . . . . 143
9.9 Submodules . . . . . . . . . . . . . . . . . . . . . . . . . . . . 147
9.10 Interpretations . . . . . . . . . . . . . . . . . . . . . . . . . . 156
9.10.1 Isomorphism between 'a and 'a × unit . . . . . . . . . 156
9.10.2 Interpretation of term-powerprod by 'a × 'k . . . . . . 157
9.10.3 Simplifier Setup . . . . . . . . . . . . . . . . . . . . . . 157

10 Type-Class-Multivariate Polynomials in Ordered Terms 158
10.1 Interpretations . . . . . . . . . . . . . . . . . . . . . . . . . . . 161
10.1.1 Unit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 161
10.2 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 161
10.3 Leading Term and Leading Coefficient: lt and lc . . . . . . 162
10.4 Trailing Term and Trailing Coefficient: tt and tc . . . . . . 173
10.5 higher and lower . . . . . . . . . . . . . . . . . . . . . . . . . 180
10.6 tail . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 189
10.7 Order Relation on Polynomials . . . . . . . . . . . . . . . . . 195
10.8 Monomials . . . . . . . . . . . . . . . . . . . . . . . . . . . . 208
10.9 Lists of Keys . . . . . . . . . . . . . . . . . . . . . . . . . . . 211
10.10 Multiplication . . . . . . . . . . . . . . . . . . . . . . . . . . 214
10.11 dgrad-p-set and dgrad-p-set-le . . . . . . . . . . . . . . . 219
10.12 Dickson’s Lemma for Sequences of Terms . . . . . . . . . . 225
11 Executable Representation of Polynomial Mappings as Association Lists 240
11.1 Power Products 240
11.1.1 Computations 242
11.2 Implementation of Multivariate Polynomials as Association Lists 243
11.2.1 Unordered Power-Products 243
11.2.2 restore constructor view 245
11.2.3 Ordered Power-Products 247
11.3 Computations 250
11.3.1 Scalar Polynomials 250
11.3.2 Vector-Polynomials 252
11.4 Code setup for type MPoly 254
11.5 lookup-pp, keys-pp and single-pp 255
11.6 Additive Structure 256
11.7 'a ⇒0 'b belongs to class comm-powerprod 257
11.8 '(a ⇒0 'b belongs to class ninv-comm-monoid-add 257
11.9 ('a, 'b) pp belongs to class lcs-powerprod 257
11.10('a, 'b) pp belongs to class ulcs-powerprod 257
11.11 Dickson’s lemma for power-products in finitely many indeterminates 257
11.12 Lexicographic Term Order 258
11.13 Degree 260
11.14 Degree-Lexicographic Term Order 260
11.15 Degree-Reverse-Lexicographic Term Order 261

12 Associative Lists with Sorted Keys 262
12.1 Preliminaries 262
12.2 Type key-order 264
12.3 Invariant in Context comparator 268
12.4 Operations on Lists of Pairs in Context comparator 271
12.4.1 lookup-pair 273
12.4.2 update-by-pair 280
12.4.3 update-by-fun-pair and update-by-fun-gr-pair 284
12.4.4 map-pair 286
12.4.5 map2-val-pair 290
12.4.6 lex-ord-pair 299
12.10.7 OAlist-tc-update-by-fun and OAlist-tc-update-by-fun-gr 351
12.10.8 OAlist-tc-filter 352
12.10.9 OAlist-tc-map-val 352
12.10.10 OAlist-tc-map2-val OAlist-tc-map2-val-neutr and OAlist-tc-map2-val-neutr 353
12.10.11 OAlist-tc-lex-ord and OAlist-tc-prod-ord 354
12.10.12 Instance of equal 355
12.11 Experiment 356

13 Ordered Associative Lists for Polynomials 356

14 Computable Term Orders 368
  14.1 Type Class nat 368
  14.2 Term Orders 372
    14.2.1 Type Classes 372
    14.2.2 LEX, DRLEX, DEG and POT 384
    14.2.3 Equality of Term Orders 388

15 Executable Representation of Polynomial Mappings as Association Lists 401
  15.1 Power-Products Represented by oalist-tc 402
    15.1.1 Constructor 405
    15.1.2 Computations 405
  15.2 MP-oalist 406
    15.2.1 Special case of addition: adding monomials 411
    15.2.2 Constructors 412
    15.2.3 Changing the Internal Order 412
    15.2.4 Ordered Power-Products 413
  15.3 Interpretations 417
  15.4 Computations 420
  15.5 Code setup for type MPoly 423

16 Quasi-Poly-Mapping Power-Products 423

17 Multivariate Polynomials with Power-Products Represented by Polynomial Mappings 429
  17.1 Degree 430
  17.2 Indeterminates 439
    17.2.1 indets 439
    17.2.2 PPs 444
    17.2.3 Polys 447
  17.3 Substitution Homomorphism 451
  17.4 Evaluating Polynomials 462
  17.5 Replacing Indeterminates 465
  17.6 Homogeneity 472
17.6.1 Homogenization and Dehomogenization .......................... 487
17.7 Embedding Polynomial Rings in Larger Polynomial Rings
(With One Additional Indeterminate) .......................... 503
17.8 Canonical Isomorphisms between \( P[X,Y] \) and \( P[X][Y] \):
focus and flatten ............................................. 514
17.9 Locale pm-powerprod ......................................... 527

18 Polynomials ........................................... 533
18.1 Polynomials represented as trees ............................. 533
18.2 Polynomials represented in normal form as lists of monomials 533
18.3 Computing normal forms of polynomials .......................... 546
18.4 Powers and substitutions of polynomials .......................... 546
18.5 Polynomial orders ........................................... 550
18.6 Degree of polynomials ........................................ 558
18.7 Executable and sufficient criteria to compare polynomials and
ensure monotonicity ........................................... 559

19 Displaying Polynomials ........................................ 576
20 Monotonicity criteria of Neurauter, Zankl, and Middeldorp 577

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 \( b \in B \leftrightarrow b \in f' \ ?C \)
proof
  assume \( b \in B \)
  moreover from \text{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 \( \text{image-image} \) ..
  moreover from \text{this}(1) have \( f \ (g \ b') = b' \) by (rule 2)
  ultimately show \( b \in B \) by simp
qed
qed
next
show \( \text{inj-on} \ f \ ?C \)
proof
  fix \( x \ y \)
  assume \( x \in \ ?C \)
  then obtain \( bx \) where \( bx \in B \) and \( x = g \ bx \) ..
  moreover from \text{this}(1) have \( f \ (g \ bx) = bx \) by (rule 2)
  ultimately have \( \ast : f \ x = bx \) by simp
  assume \( y \in \ ?C \)
  then obtain \( by \) where \( by \in B \) and \( y = g \ by \) ..
  moreover from \text{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 \( bx = by \) unfolding \( \ast \) by simp
  thus \( x = y \) by (simp only: \( x \ y \))
  qed
  qed
qed

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

lemma \( \text{transp-sequence} : \)
  assumes \( \text{transp} \ r \ \text{and} \ \bigwedge i. \ r \ (\text{seq} \ (\text{Suc} \ i)) \ (\text{seq} \ i) \ \text{and} \ i < j \)
  shows \( r \ (\text{seq} \ j) \ (\text{seq} \ i) \)
proof −
  have \( \bigwedge k. \ r \ (\text{seq} \ (i + \text{Suc} \ k)) \ (\text{seq} \ i) \)
  proof −
    fix \( k :: \text{nat} \)
    show \( r \ (\text{seq} \ (i + \text{Suc} \ k)) \ (\text{seq} \ i) \)

8
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 (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

hence r (seq (Suc (Suc (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 \( \forall s \in S \Rightarrow (\exists t \in T. P t s) \)
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 \( i < j \)

(*seq-incr-Suc*) show ?thesis by (rule transp-sequence, simp add: transp-def)

qed

qed

have \( \text{sub: seq } i \subseteq S \text{ for } i \)

proof (induct \( i \), simp add: seq-def, simp add: seq-Suc fun-def, rule)

fix \( i \)

assume \( \text{Ex (crit (seq } i)) \)

hence \( \text{crit (seq } i) (Eps (\text{crit (seq } i))) \) by (rule someI-ex)

thus \( Eps (\text{crit (seq } i)) \in S \) by (simp add: crit-def)

qed

have \( \exists i. \text{seq (Suc } i) = \text{seq } i \)

proof (rule ccontr, simp)

assume \( \forall i. \text{seq (Suc } i) \neq \text{seq } i \)

with \( \text{seq-incr-Suc} \) have \( \text{seq } i \subseteq \text{seq (Suc } i) \) for \( i \) by blast

define \( \text{seq1} \) where \( \text{seq1} = (\lambda n. (\text{SOME } s. s \in \text{seq (Suc } n) \land s \notin \text{seq } n)) \)

have \( \text{seq1} n \in \text{seq (Suc } n) \land \text{seq1 } n \notin \text{seq } n \) for \( n \) unfolding \( \text{seq1-def} \)

proof (rule someI-ex)

from \( \text{seq } n \subseteq \text{seq (Suc } n) \) show \( \exists x. x \in \text{seq (Suc } n) \land x \notin \text{seq } n \) by blast

qed

have \( \text{seq1 } i \in S \) for \( i \)

proof (rule fact sub)

with \( \text{assms(2)} \) obtain \( a \) \( b \) where \( a < b \) and \( \text{P (seq1 } a) (\text{seq1 } b) \) by (rule almost-full-onD)

from \( a < b \) have \( \text{Suc } a \leq b \) by simp

from \( \text{seq1} \) have \( \text{seq1 } a \in \text{seq (Suc } a) \) ..

also from \( \text{Suc } a \leq b \) have \( \ldots \subseteq \text{seq } b \) by (rule seq-incr)

finally have \( \text{seq1 } a \in \text{seq } b \).

from \( \text{seq1} \) have \( \text{seq1 } b \in \text{seq (Suc } b) \) and \( \text{seq1 } b \notin \text{seq } b \) by blast+

hence \( \text{crit (seq } b) (\text{seq1 } b) \) by (simp add: seq-Suc fun-def someI split: if-splits)

hence \( \forall u \in \text{seq } b. \neg \text{P } u (\text{seq1 } b) \) by (simp add: crit-def)

from this \( \text{seq1 } a \in \text{seq } b \) have \( \neg \text{P (seq1 } a) (\text{seq1 } b) \) ..

from this \( \text{P (seq1 } a) (\text{seq1 } b) \) show \( \text{False} \) ..

qed

then obtain \( i \) where \( \text{seq (Suc } i) = \text{seq } i \) ..

show ?thesis

proof

show \( \text{finite (seq } i) \) by (induct \( i \), simp-all add: seq-def fun-def)

next

fix \( s \)

assume \( s \in S \)

let \( ?s = Eps (\text{crit (seq } i)) \)

show \( \exists t \in \text{seq } i. \text{P } t \)

proof (rule ccontr, simp)

assume \( \forall t \in \text{seq } i. \neg \text{P } t \)

qed
with \( s \in S \) have \( \text{crit} \ (\text{seq} \ i) \ s \) by \( \text{simp only: crit-def} \)

hence \( \text{crit} \ (\text{seq} \ i) \ ?s \) and \( \text{eq: seq} \ (\text{Suc} \ i) = \text{insert} \ ?s \ (\text{seq} \ i) \)

by \( \text{auto simp add: seq-Suc fun-def intro: someI} \)

from this \( (1) \) have \( ?s \notin \text{seq} \ i \) by \( \text{rule critD} \)

hence \( \text{seq} \ (\text{Suc} \ i) \neq \text{seq} \ i \) unfolding \( \text{eq} \) by blast

from this \( (\text{seq} \ (\text{Suc} \ i) = \text{seq} \ i) \) show False ..

qed

qed \( \text{(fact sub)} \)

qed

1.1 Lists

lemma map-upt: \( \text{map} \ (\lambda i. \ f \ (\text{xs} ! i)) \ [0..<\text{length} \ \text{xs}] = \text{map} \ f \ \text{xs} \)

by \( \text{(auto intro: nth-equalityI)} \)

lemma map-upt-zip:

assumes \( \text{length} \ \text{xs} = \text{length} \ \text{ys} \)

shows \( \text{map} \ (\lambda i. \ f \ (\text{xs} ! i) \ (\text{ys} ! i)) \ [0..<\text{length} \ \text{ys}] = \text{map} \ (\lambda (x, y). \ f \ x \ y) \ (\text{zip} \ \text{xs} \ \text{ys}) \) \( (\text{is} \ ?l = ?r) \)

proof –

have len-l: \( \text{length} \ ?l = \text{length} \ \text{ys} \) by simp

from assms have len-r: \( \text{length} \ ?r = \text{length} \ \text{ys} \) by simp

show ?thesis

proof (simp only: list-eq-iff-nth-eq len-l len-r, rule, rule, intro allI impI)

fix \( i \)

assume \( i < \text{length} \ \text{ys} \) and \( i < \text{length} \ ?r \) by (simp-all only: len-l len-r)

thus \( \text{map} \ (\lambda i. \ f \ (\text{xs} ! i) \ (\text{ys} ! i)) \ [0..<\text{length} \ \text{ys}] ! i = \text{map} \ (\lambda (x, y). \ f \ x \ y) \ (\text{zip} \ \text{xs} \ \text{ys}) ! i \)

by simp

qed

qed

lemma distinct-sorted-wrt-irrefl:

assumes \( \text{irreflp} \ \text{rel} \) and \( \text{transp} \ \text{rel} \) and \( \text{sorted-wrt} \ \text{rel} \ \text{xs} \)

shows \( \text{distinct} \ \text{xs} \)

using assms(3)

proof (induct \( \text{xs} \))

case Nil

show ?case by simp

next

case (Cons \( x \ \text{xs} \))

from Cons(2) have \( \text{sorted-wrt} \ \text{rel} \ \text{xs} \) and \( *: \forall y \in \text{set} \ \text{xs}. \ \text{rel} \ x \ y \)

by (simp-all)

from this(1) have \( \text{distinct} \ \text{xs} \) by (rule Cons(1))

show ?case

proof (simp add: (distinct \( \text{xs} \)), rule)

assume \( x \in \text{set} \ \text{xs} \)

with \( * \) have \( \text{rel} \ x \ x \ .. \)

11
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 antisym 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)
    with * have x = y 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 antisymD)
with \( x \neq y \) show False ..
qed
from 2(3) have sorted-wrt rel xs by (simp)
moreover note \( \langle \text{distinct } xs \rangle \)
moreover from 2(5) have sorted-wrt rel ys by (simp)
moreover note \( \langle \text{distinct } ys \rangle \)
moreover from 2(7) \( \langle x \notin set \ xs \rangle \langle y \notin set \ ys \rangle \) 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

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

fun merge-wrt :: \( (a \\Rightarrow a \Rightarrow bool) \\Rightarrow a list \\Rightarrow a list \\Rightarrow a list \) 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
do 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 \( x \neq y \) show ?thesis by (simp add: 3(2) insert-commute)
    next
case False
      with \( x \neq y \) show ?thesis by (simp add: 3(3))
qeda
lemma sorted-merge-wrt:
  assumes transp rel and \( \forall x, y. x \neq y \implies \text{rel } x y \lor \text{rel } y x \)
  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 \( z \in \text{set } (\text{merge-wrt rel } xs \ ys) \)
hence \( z \in \text{set } xs \cup \text{set } ys \) by (simp only: set-merge-wrt)
thus \( \text{rel } y z \)
proof
  assume \( z \in \text{set } xs \)
  with 3(6) show ?thesis by (simp add: True)
next
  assume \( z \in \text{set } ys \)
  with 3(7) show ?thesis by (simp)
qeda
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 ∈ set (merge-wrt rel xs (y # ys))
hence z ∈ 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 ∈ set xs ∪ set ys
thus ?thesis
proof
assume z ∈ set xs
with 3(6) show ?thesis by (simp)
next
assume z ∈ 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 ≠ 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 ∈ set (merge-wrt rel (x # xs) ys)
hence z ∈ 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 $z \in \text{set } xs \cup \text{set } ys$

thus ?thesis

proof

assume $z \in \text{set } xs$

with \text{3(6)} have $\text{rel } x \ z$ by (simp)

with \text{3(4)} * show ?thesis by (rule transpD)

next

assume $z \in \text{set } ys$

with \text{3(7)} show ?thesis by (simp)

qed

qed

note False $\neg \text{rel } x \ y$; \text{3(4), 5, 6}

moreover from \text{3(7)} have $\text{sorted-wrt rel } ys$ by (simp)

ultimately show $\text{sorted-wrt rel } (\text{merge-wrt rel } (x \#\hspace{1em} xs) \ ys)$ by (rule \text{3(3)})

qed

qed

qed

lemma \text{set-fold}:

assumes $\forall x \ y s. \text{set } (f \ (g x) \ ys) = \text{set } (g x) \cup \text{set } ys$

shows $\text{set } (\text{fold } (\lambda x. f \ (g x)) \ xs \ ys) = (\bigcup x \in \text{set } xs. \text{set } (g x)) \cup \text{set } ys$

proof (induct \text{xs arbitrary: } ys)

case Nil

show ?case by simp

next

case (\text{Cons } x \ \text{xs})

have eq: $\text{set } (\text{fold } (\lambda x. f \ (g x)) \ xs \ (f \ (g x) \ ys)) = (\bigcup x \in \text{set } xs. \text{set } (g x)) \cup \text{set } (f \ (g x) \ ys)$

by (rule Cons)

show ?case by (simp add: o-def assms set-merge-wrt eq ac-simps)

qed

1.2 Sums and Products

lemma \text{additive-implies-homogenous}:

assumes $\forall x \ y f. \ (x + y) = f x + (f \ (y::\hspace{1em} 'a::monoid-add))::('b::cancel-comm-monoid-add)$

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 \text{fun-sum-commute}:

assumes $f \ 0 = 0$ and $\forall x \ y f. \ (x + y) = f x + f y$

shows $f \ (\text{sum } g \ A) = (\sum a \in A. f \ (g a))$

proof (cases \text{finite } A)

qed
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 \( \forall x y. f (x + y) = f x + ((f y)::'a::cancel-comm-monoid-add) \)
shows \( f (\sum A) = (\sum a\in A. f (g a)) \)
by (rule fun-sum-commute, rule additive-implies-homogenous, fact+)

lemma fun-sum-list-commute:
assumes \( f 0 = 0 \) and \( \forall x y. f (x + y) = f x + f y \)
shows \( f (\sum xs) = \sum (map f xs) \)
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 \( \forall x y. f (x + y) = f x + ((f y)::'a::cancel-comm-monoid-add) \)
shows \( f (\sum xs) = \sum (map f xs) \)
by (rule fun-sum-list-commute, rule additive-implies-homogenous, fact+)

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

lemma sum-list-upt: \( \sum i\in[0..<n]. (xs ! i) \) = \( \sum x\in xs. f x \)
by (simp only: map-upt)

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

lemma sum-list-zeroI:
assumes \( \text{set } xs \subseteq \{0\} \)
shows $\sum\text{list } xs = 0$
using $\text{assms by (induct xs, auto)}$

lemma fun-prod-commute:
  assumes $f 1 = 1$ and $\forall x y. f(x*y) = f(x)*f(y)$
  shows $f(\prod_{a\in A} g(a)) = \prod_{a\in A} f(g(a))$
proof (cases finite $A$)
  case True 
  thus $?\text{thesis}$
  proof (induct $A$)
    case empty
    thus $?\text{case}$ by (simp add: $\text{assms(1)}$)
  next
    case step: $(\text{insert } a\ A)$
    show $?\text{case}$ by (simp add: prod.insert[OF step(1) step(2)] $\text{assms(2) step(3)}$)
  qed
next
  case False 
  thus $?\text{thesis}$ by (simp add: $\text{assms(1)}$)
  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 $\Rightarrow$ 0 nat) $\Rightarrow$ ‘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 θ :: (nat ⇒₀ nat) ⇒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 ⇒₀ nat) ⇒₀ 'a) ⇒ .
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 ⇒₀ nat) ⇒₀ 'a) ⇒ .
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 ⇒₀ nat) ⇒₀ 'a) ⇒ .
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 ⇒₀ nat) ⇒₀ 'a) ⇒ .
2.4 Multiplicative structure

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

lift-definition one-mpoly :: 'a mpoly
is 1 :: ((nat ⇒₀ nat) ⇒₀ 'a).

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 ⇒₀ nat) ⇒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 ⇒_0 nat) ⇒ 'a::zero ⇒ 'a mpoly
  is Poly-Mapping.single :: (nat ⇒_0 nat) ⇒ -.

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 ⇒ 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 mpol :: (semiring-char-0) semiring-char-0
by intro_classes (auto simp add: of-nat-monom inj-of-nat intro: inj-compose)

instance mpol :: (ring-char-0)
by intro_classes (auto simp add: of-nat-monom inj-of-nat intro: inj-compose)

2.6 Constants and Indeterminates

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

definition Var0 :: 'a ⇒ ('a ⇒ 0 nat) ⇒ 0 'b::{one,zero} where
  Var0 n ≡ Poly-Mapping.single (Poly-Mapping.single n 1) 1

definition Const0 :: 'b ⇒ ('a ⇒ 0 nat) ⇒ 0 'b::zero where Const0 c ≡ Poly-Mapping.single 0 c

lemma Const0-one: Const0 1 = 1
by (simp add: Const0-def)

lemma Const0-numeral: Const0 (numeral x) = numeral x
by (auto intro!: poly-mapping-eqI simp: Const0-def lookup-numeral)

lemma Const0-minus: Const0 (− x) = − Const0 x
by (simp add: Const0-def single-uminus)

lemma Const0-zero: Const0 0 = 0
by (auto intro!: poly-mapping-eqI simp: Const0-def)
lemma Var\textsuperscript{\textdagger}: Var cool to\textsuperscript{\textdagger} = Poly-Mapping.single (Poly-Mapping.single cool to\textsuperscript{\textdagger})

by (induction n) (auto simp: Var-def mult-single single-add[symmetric])

lift-definition Var::nat ⇒ 'b::{one,zero} mpoly is Var\textsubscript{0}.

lift-definition Const::'b::zero ⇒ 'b mpoly is Const\textsubscript{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 ⇒ 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 * (Πv. f v cool to m v))


definition insertion-fun :: (nat ⇒ 'a) ⇒ ((nat ⇒ nat) ⇒ 'a) ⇒ 'a::comm-semiring-1

where
insertion-fun f p = (∑m. p m * (Πv. f v cool to 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 ⇒ 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 ⇒ nat) ⇒ 0 'a

23
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 ⇒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 = (Π v. f v ´ Poly-Mapping.lookup m v) for m
ultimately have *: Π a b. g (a + b) = g a * g b

24
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 (subt Sum-any-left-distrib)
  apply (auto intro: finite-mult-not-eq-zero-rightI)
  apply (subt Sum-any-right-distrib)
  apply (auto intro: finite-mult-not-eq-zero-rightI)
  apply (subt 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 (subt (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 (subt 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 (subt 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: *)

25
also have \( \ldots = (\sum_l. \sum_n. (g_l * g_n) * (\text{Poly-Mapping.lookup } p_l * \text{Poly-Mapping.lookup } q_n)) \)
apply (subst \text{Sum-any.cartesian-product} \[\text{of } ?P \times ?Q\])
apply (auto dest!: \text{mult-not-zero})
done
also have \( \ldots = (\sum_l. \sum_n. (\text{Poly-Mapping.lookup } p_l * g_l) * (\text{Poly-Mapping.lookup } q_n * g_n)) \)
by (simp add: \text{ac-simps})
also have \( \ldots = (\sum_m. \text{Poly-Mapping.lookup } p_m * g_m) * (\sum_m. \text{Poly-Mapping.lookup } q_m * g_m) \)
by (rule \text{Sum-any-product [symmetric]}) (auto intro: \text{finite-mult-not-eq-zero-rightI})
finally show \( ?\text{thesis} \) by (simp add: \text{insertion-fun-def g-def})
qed

lemma insertion-mult:
insertion \( f \) \((p * q)\) = insertion \( f \) \( p \) * insertion \( f \) \( q \)
by transfer (simp add: \text{insertion-aux-def insertion-fun-mult})

2.10 Degree

lift-definition degree :: \('a::zero \text{mpoly} \Rightarrow \text{nat} \Rightarrow \text{nat}'
is \( \lambda p \ v. \ \text{Max} \ (\text{insert } 0 \ ((\lambda m. \text{Poly-Mapping.lookup } m \ v) \ ' \text{Poly-Mapping.keys } p)) \).

lift-definition total-degree :: \('a::zero \text{mpoly} \Rightarrow \text{nat}'
is \( \lambda p. \ \text{Max} \ (\text{insert } 0 \ ((\lambda m. \text{sum} \ (\text{Poly-Mapping.lookup } m) \ (\text{Poly-Mapping.keys } m)) ' \text{Poly-Mapping.keys } p)) \).

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: \text{smult } s \ p = \text{monom } 0 \ s * p

26
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 ⇒ 0 nat) ⇒ 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”:

\[
\forall x y :: 'a mpoly. y \neq 0 \Rightarrow \exists a q r. \text{smult } a x = q \ast y + r
\]

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 ⇒ 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)

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

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

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

lemma div-poly-code:
  p div q = fst (divmod p q)
  by (simp add: divmod-def)

lemma mod-poly-code:
  p mod q = snd (divmod p q)
  by (simp add: divmod-def)

2.12 Primitive poly, etc

lift-definition coeffs :: 'a :: zero mpoly ⇒ 'a set
  is Poly-Mapping.range :: ((nat ⇒0 nat) ⇒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 (λx. f x when x∈S)) = (λx. 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 ⇒ ('a ⇒₀ 'b::monoid-add) ⇒ ('a ⇒₀ 'b) where
  remove-key k0 f = Abs-poly-mapping (λk. lookup f k when k ≠ k0)

lemma remove-key-lookup:
  lookup (remove-key k0 f) k = (lookup f k when k ≠ k0)
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 \( x \in \mathcal{B} \)
then have \( \text{lookup} (\text{remove-key} \, k \, f) \, x \neq 0 \) by blast
then show \( x \in \mathcal{A} \)
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 \( k' \neq k \)) = lookup (remove-key \( f \, k \)) \( k' \)
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 \( f \, k \, k' \) = (lookup \( f \, k' \) when \( k' \neq k \)) + (lookup \( f \, k \) when \( k'=k \))
unfolding when-def by fastforce
with rem sin have lookup \( f \, k' \, k \) = lookup ((remove-key \( f \, k \)) + Poly-Mapping.single \( k \) (lookup \( f \, k \))) \( k' \)
using lookup-add by metis
\}
then show \( ?\text{thesis} \) by (metis poly-mapping-eqI)
qed

**lemma** remove-key-single[simp]: remove-key \( v \) (Poly-Mapping.single \( v \, n \)) = 0
proof −
have 0:\( \forall k. \) (lookup (Poly-Mapping.single \( v \, n \)) \( k \)) when \( k \neq v \) = 0 by (simp add: lookup-single-not-eq when-def)
show \( ?\text{thesis} \) 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 \( P::(\alpha, \, \beta::\text{monoid-add}) \Rightarrow \text{poly-mapping} \Rightarrow \text{bool} \)
assumes single:\( \forall k \, v \. \) P (Poly-Mapping.single \( k \) \( v \))
and sum:\( \forall f \, g \, k \, v \. \) P \( f \) \( \Longrightarrow \) P \( g \) \( \Longrightarrow \) g = (Poly-Mapping.single \( k \) \( v \)) \( \Longrightarrow \) k \notin \text{keys}
\( f \) \( \Longrightarrow \) P \( f + g \))
shows P \( f \) using finite-keys[of \( f \)]
proof (induction keys \( f \) arbitrary; \( f \) rule: finite-induct)
case (empty)
then show \( ?\text{case} \) 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 f1-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 f1-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 f1-def(1) remove-key-keys)
qed
have P f2 by (simp add: single f1-def(2))
have f1 + f2 = f using remove-key-sum f1-def by auto
have k /∈ keys f1 using remove-key-keys f1-def by fast
then show ?case using ⟨P f1⟩ ⟨P f2⟩ sum[of f1 f2 k lookup f k] (f1 + f2 = f) f1-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
  (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
  (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
  assume k = 0
then have \((\text{Poly-Mapping.single } v k) = 0\) by simp
then have \(\text{vars } (\text{monom } (\text{Poly-Mapping.single } v k) a) = \{\}\)
by (metis (mono-tags, lifting) single-zero singleton-inject subset-singletonD vars-monom-single zero-neq-one)
then show \(\text{thesis using } (k=0)\) by auto
next
assume \(k \neq 0\)
then show \(\text{thesis}\)
proof (cases \(a = 0\))
assume \(a = 0\)
then have \(\text{monom } (\text{Poly-Mapping.single } v k) a = 0\)
by (metis monom.abs-eq monom-zero single-zero)
then show \(\text{thesis}\) by (metis (mono-tags, hide-lams) ⟨\(k \neq 0\)⟩ ⟨\(a = 0\)⟩ monom.abs-eq)
next
assume \(a \neq 0\)
then have \(v \in \text{vars } (\text{monom } (\text{Poly-Mapping.single } v k) a)\)
by (simp add: ⟨\(k \neq 0\)⟩ vars-def)
then show \(\text{thesis using } (a \neq 0; k \neq 0)\) vars-monom-single by fastforce
qed

lemma vars-monom:
assumes \(a \neq 0\)
shows \(\text{vars } (\text{monom } m \ (1::'a::zero-neq-one)) = \text{vars } (\text{monom } m \ (a::'a))\)

lemma vars-add: \(\text{vars } (p1 + p2) \subseteq \text{vars } p1 \cup \text{vars } p2\)
proof
fix \(w\) assume \(w \in \text{vars } (p1 + p2)\)
then obtain \(m\) where \(w \in \text{keys } m \ m \in \text{keys } (\text{mapping-of } (p1 + p2))\)
by (metis UN-E vars-def)
then have \(m \in \text{keys } (\text{mapping-of } (p1)) \cup \text{keys } (\text{mapping-of } (p2))\)
by (metis Poly-Mapping.keys-add plus-mpoly.rep-eq subset-iff)
then show \(w \in \text{vars } p1 \cup \text{vars } p2\) using vars-def ⟨\(w \in \text{keys } m\)⟩ by fastforce
qed

lemma vars-mult: \(\text{vars } (p*q) \subseteq \text{vars } p \cup \text{vars } q\)
proof
fix \(x\) assume \(x \in \text{vars } (p*q)\)
then obtain \(m\) where \(m \in \text{keys } (\text{mapping-of } (p*q)) x \in \text{keys } m\)
using vars-def by blast
then have \(m \in \text{keys } (\text{mapping-of } p * \text{mapping-of } q)\)
by (simp add: times-mpoly.rep-eq)
then obtain \(a b\) where \(m = a + b = a \in \text{keys } (\text{mapping-of } p) b \in \text{keys } (\text{mapping-of } q)\)
using keys-mult by blast
then have \(x \in \text{keys } a \cup \text{keys } b\)
using Poly-Mapping.keys-add (∃ x ∈ keys m) by force
then show \( x \in \text{vars} \ p \cup \text{vars} \ q \) unfolding vars-def
using \( a \in \text{keys} \ (\text{mapping-of} \ p) ; \ b \in \text{keys} \ (\text{mapping-of} \ q) \) by blast
qed

lemma vars-add-monom:
assumes \( p^2 = \text{monom} \ m \ a \ m \not\in \text{keys} \ (\text{mapping-of} \ p) \)
shows \( \text{vars} \ (p1 + p2) = \text{vars} \ p1 \cup \text{vars} \ p2 \)
proof
have \( \text{keys} \ (\text{mapping-of} \ p2) \subseteq \{m\} \) using monom-def keys-single assms by auto

have \( \text{keys} \ (\text{mapping-of} \ (p1+p2)) = \text{keys} \ (\text{mapping-of} \ p1) \cup \text{keys} \ (\text{mapping-of} \ p2) \)
using keys-add by (metis Int-insert-right-if0 \( \text{keys} \ (\text{mapping-of} \ p2) \subseteq \{m\} \) assms inf-bot-right plus-mpoly.rep-eq subset-singletonD)

then show \( \text{thesis} \) unfolding vars-def by simp
qed

lemma vars-setsum:
finite \( S \Rightarrow \text{vars} \ (\sum m \in S. f m) \subseteq (\bigcup m \in S. \text{vars} \ (f m)) \)
proof (induction \( S \) rule:finite-induct)
case empty
then show \( \text{case} \) by (metis UN-empty eq-iff monom-zero sum.empty single-zero vars-monom-single-cases)
next
case \( \text{insert} \ s \ S \)
then have \( \text{vars} \ (\sum f \ (\text{insert} \ s \ S)) = \text{vars} \ (f \ s + \sum f \ S) \) by (metis sum.insert)
also have \( \ldots \subseteq \text{vars} \ (f \ s) \cup \text{vars} \ (\sum f \ S) \) by (simp add: vars-add)
also have \( \ldots \subseteq (\bigcup m \in \text{insert} \ s \ S. \text{vars} \ (f \ m)) \) using insert.IH by auto
finally show \( \text{case} \) bymetis
qed

lemma coeff-monom:
\( \text{coeff} \ (\text{monom} \ m \ a) \ m' = (a \text{ when } m' = m) \)
by (simp add: coeff-def lookup-single-not-eq when-def)

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

lemma coeff-eq:
\( \text{coeff} \ p = \text{coeff} \ q \leftrightarrow p = q \) by (simp add: coeff-def lookup-inject mapping-of-inject)

lemma coeff-monom-mult:
\( \text{coeff} \ ((\text{monom} \ m' \ a) \ast q) \ (m' + m) = a \ast \text{coeff} \ q \ m \)
unfolding coeff-def times-mpoly.rep-eq lookup-mult mapping-of-monom lookup-single when-mult
\( \text{Sum-\text{any-when-equal'} Groups.cancel-semi\text{group-add-class}.add-left-cancel} \) by metis

lemma one-term-is-monomial:
assumes \( \text{card} \ (\text{keys} \ (\text{mapping-of} \ p)) \leq 1 \)
obtains \( m \) where \( p = \text{monom} \ m \ (\text{coeff} \ p \ m) \)
proof (cases keys \( (\text{mapping-of} \ p) = \{\} \))
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) \{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 ⇒_0 nat) ⇒ 'a::zero mpoly ⇒ 'a mpoly where remove-term m0 p = MPoly (Abs-poly-mapping (λm. 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 (λm. 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 = (λm'. (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 λp :: (nat ⇒ 0 nat) ⇒ 0 a. P (MPoly p)] MPoly-induct
monom.

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) (w::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.emptyI 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

36
then have \( a \ast (\prod_{v \in \{v\}}. f va \ast \text{lookup} (\text{Poly-Mapping}.\text{single} v n) va) = {?i}\)
unfolding 2
  using Prod-any.expand-superset[of \{v\} \lambda va. f va \ast \text{lookup} (\text{Poly-Mapping}.\text{single} v n) va, simplified]
  by fastforce
then show \(?thesis by simp\)
qed

lemma insertion-single[simp]: insertion \( f \) (\text{monom} (\text{Poly-Mapping}.\text{single} (v::nat) (n::nat)) a) = a \ast f v \ast 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 \{nat\}) \Rightarrow 'a::comm-ring-1)
assumes \( \forall m v. p m \neq 0 \Longrightarrow \text{lookup} m v \neq 0 \Longrightarrow f v = g v \)
shows insertion-fun \( f p = \text{insertion-fun} g p \)
proof –
  { fix m::nat\Rightarrow\{nat\}
    assume p m \neq 0
    then have \( (\prod_{v}. f v \ast \text{lookup} m v) = (\prod_{v}. g v \ast \text{lookup} m v) \)
    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 \{nat\}) \Rightarrow 'a::comm-ring-1)
assumes \( \forall m v. \text{lookup} p m \neq 0 \Longrightarrow \text{lookup} m v \neq 0 \Longrightarrow f v = g v \)
shows insertion-aux \( f p = \text{insertion-aux} g p \)
  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 \( \forall v. v \in \text{vars} p \Longrightarrow f v = g v \)
shows insertion \( f p = \text{insertion} g p \)
proof –
  { fix m v assume \text{lookup} (\text{mapping-of} p) m \neq 0 \text{lookup} m v \neq 0
    then have \( v \in \text{vars} p \) 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 \(?thesis unfolding insertion-def using insertion-aux-irrelevant-vars[of mapping-of p] by (metis insertion.rep-eq insertion-def)\)

37
qed

5 Nested MPoly

definition reduce-nested-mpoly:: 'a::comm-ring-1 mpoly mpoly ⇒ 'a mpoly where
  reduce-nested-mpoly pp = insertion (λv. 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 (λv. 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 ≠ 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 ∈ (⋃m∈{m'. coeff p m' ≠ 0}. vars (monom (remove-key v m)
    (monom (Poly-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∈{m'. coeff p m' ≠ 0} x ∈ vars (monom (remove-key v m)
    (monom (Poly-Mapping.single v (lookup m v)) (coeff p m)))
    by blast
  show x ∈ vars p by (metis (mono-tags, lifting) DiffD1 UN-I 'm ∈ {m'. coeff p m' ≠ 0})
  (x ∈ vars (monom (remove-key v m) (monom (Poly-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 /∈ vars (extract-var p v)
proof
  have finite {m'. coeff p m' ≠ 0} by (simp add: coeff-def)
  have ∀m. m∈{m'. coeff p m' ≠ 0} ⇒ v /∈ vars (monom (remove-key v m)
    (monom (Poly-Mapping.single v (lookup m v)) (coeff p m)))
    by (metis Diff_iff remove-key-keys singletonI subsetCE vars-def vars-monom-subset)
then have \( v \notin \bigcup m \in \{m', \text{coeff } p \ m' \neq 0\}, \text{vars} \ (\text{monom} \ (\text{remove-key } v \ m) \ (\text{monom} \ (\text{Poly-Mapping.single } v \ (\text{lookup } m \ v)) \ (\text{coeff } p \ m))) \)
  by simp

then show \(?thesis
  unfolding \text{extract-var-non-zero-coeff using vars-setsum[of finite \{m', \text{coeff } p \ m' \neq 0\}] by blast
qed

lemma \text{vars-coeff-extract-var}:
\( \text{vars} \ (\text{coeff} \ (\text{extract-var } p \ v) \ j) \subseteq \{v\} \)
proof (induction p rule:mpoly-induct)
  case (monom m a)
  then show \(?case unfolding \text{extract-var-monom coef-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 \text{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
\( \text{replace-coeff } f \ p = \text{MPoly} \ (\lambda m. f \ (\text{lookup } (\text{mapping-of } p) \ m))) \)

lemma \text{coeff-replace-coeff}:
assumes \( f \ 0 = 0 \)
shows \( \text{coeff} \ (\text{replace-coeff } f \ p) \ m = f \ (\text{coeff } p \ m) \)
proof -
  have 0:finite \{m. f \ (\text{lookup } (\text{mapping-of } p) \ m) \neq 0\}
    unfolding coeff-def [symmetric] by (metis (mono-tags, lifting) Collect-mono assumptions(1) coeff-def finite-lookup finite-subset)+
  then show \(?thesis unfolding \text{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 \text{replace-coeff-monom}:
assumes \( f \ 0 = 0 \)
shows \( \text{replace-coeff } f \ (\text{monom } m \ a) = \text{monom } m \ (f \ a) \)
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 \text{replace-coeff-add}:
assumes \( f \ 0 = 0 \)
assumes \( \land a \ b. f \ (a+b) = f \ a + f \ b \)
shows \( \text{replace-coeff } f (p1 + p2) = \text{replace-coeff } f p1 + \text{replace-coeff } f p2 \)

proof
- have finite \( \{ m. \ f \ (\text{lookup} \ (\text{mapping-of } p1) \ m) \neq 0 \} \)
  finite \( \{ m. \ f \ (\text{lookup} \ (\text{mapping-of } p2) \ m) \neq 0 \} \)
  unfolding coeff-def [symmetric] by (metis (mono-tags, lifting) Collect-mono)
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-mp-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 \( \text{replace-coeff } (\text{insertion } f) (\text{extract-var } p v) = \text{replace-coeff } (\text{insertion } g) (\text{extract-var } p v) \)
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 \( \text{vars} \ (\text{replace-coeff} \ f \ p) \subseteq \text{vars} \ p \)
  unfolding \( \text{vars-def} \) apply \((\text{rule subsetI})\) unfolding mem-simps(8) coeff-keys
  using \( \text{assms} \ \text{coeff-replace-coeff} \) by \((\text{metis coeff-keys})\)

definition \( \text{polyfun} : \text{nat set} \Rightarrow ((\text{nat} \Rightarrow 'a::\text{comm-semiring-1}) \Rightarrow 'a) \Rightarrow \text{bool} \)
where \( \text{polyfun} \ N \ f = (\exists \ p. \ \text{vars} \ p \subseteq N \land (\forall x. \text{insertion} \ x \ p = f \ x)) \)

lemma \( \text{polyfunI} : ((\forall x. \text{insertion} \ x \ p = f \ x) \Rightarrow P) \Rightarrow \text{polyfun} \ N \ f \)
  unfolding \( \text{polyfun-def} \) by \(\text{metis}\)

lemma \( \text{polyfun-subset} : N \subseteq N' \Rightarrow \text{polyfun} \ N \ f \Rightarrow \text{polyfun} \ N' \ f \)
  unfolding \( \text{polyfun-def} \) by \(\text{blast}\)

lemma \( \text{polyfun-const} : \text{polyfun} \ N \ (\lambda - \ c) \)
proof –
  have \( (\forall x. \text{insertion} \ x \ (\text{monom} \ 0 \ c) = c) \) using \(\text{insertion-single} \) by \(\text{metis} \ \text{insertion-one monom-one mult.commute mult.right-neutral single-zero} \)
  then show ?thesis unfolding \( \text{polyfun-def} \) by \(\text{metis} \ (\text{full-types} \ \text{empty-iff} \ \text{keys-single} \ \text{single-zero subsetI subset-antisym vars-monom-subset})\)
qed

lemma \( \text{polyfun-add} : \)
assumes \( \text{polyfun} \ N \ f \ \text{polyfun} \ N \ g \)
shows \( \text{polyfun} \ N \ (\lambda x. \ f \ x + g \ x) \)
proof –
  obtain \( p1 \ p2 \) where \( \text{vars} \ p1 \subseteq N \land (\forall x. \text{insertion} \ x \ p1 = f \ x) \)
    \( \text{vars} \ p2 \subseteq N \land (\forall x. \text{insertion} \ x \ p2 = g \ x) \)
    using \( \text{polyfun-def} \ \text{assms} \) by \(\text{metis}\)
  then have \( \text{vars} \ (p1 + p2) \subseteq N \land (\forall x. \text{insertion} \ x \ (p1 + p2) = f \ x + g \ x) \)
    using \( \text{vars-add} \) using \(\text{Un-iff subsetCE subsetI apply blast} \)
    by \(\text{simp add:} (\forall x. \text{insertion} \ x \ p1 = f \ x) \land (\forall x. \text{insertion} \ x \ p2 = g \ x) \ \text{insertion-add}\)
  then show ?thesis using \(\text{polyfun-def} \) by \(\text{blast}\)
qed

lemma \( \text{polyfun-mult} : \)
assumes \( \text{polyfun} \ N \ f \ \text{polyfun} \ N \ g \)
shows \( \text{polyfun} \ N \ (\lambda x. \ f \ x \ast g \ x) \)
proof –
  obtain \( p1 \ p2 \) where \( \text{vars} \ p1 \subseteq N \land (\forall x. \text{insertion} \ x \ p1 = f \ x) \)
    \( \text{vars} \ p2 \subseteq N \land (\forall x. \text{insertion} \ x \ p2 = g \ x) \)
    using \( \text{polyfun-def} \ \text{assms} \) by \(\text{metis}\)
  then have \( \text{vars} \ (p1 \ast p2) \subseteq N \land (\forall x. \text{insertion} \ x \ (p1 \ast p2) = f \ x \ast g \ x) \)
    using \( \text{vars-mult} \) using \(\text{Un-iff subsetCE subsetI apply blast} \)
    by \(\text{simp add:} (\forall x. \text{insertion} \ x \ p1 = f \ x) \land (\forall x. \text{insertion} \ x \ p2 = g \ x) \ \text{insertion-mult}\)
  then show ?thesis using \(\text{polyfun-def} \) by \(\text{blast}\)
lemma polyfun-Sum:
assumes finite I
assumes $\forall i. i \in I \Rightarrow \text{polyfun } N (f_i)$
shows $\text{polyfun } N (\lambda x. \sum i \in I. f_i x)$
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 $\forall i. i \in I \Rightarrow \text{polyfun } N (f_i)$
shows $\text{polyfun } N (\lambda x. \prod i \in I. f_i x)$
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 $i \in N$
shows $\text{polyfun } N (\lambda x. x i)$
proof
  have $\forall f. \text{insertion } f (\text{monom } (\text{Poly-Mapping}. \text{single } i 1) 1) = f_i$ 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

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 '\(a \Rightarrow_0 \text{nat}\), 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, \(x^2 y\) 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 \(x^2 y \cdot xz = x^3 y z\)

\[6.1\) Constant Keys

Legacy:

\textbf{lemmas} keys-eq-empty-iff = keys-eq-empty

\textbf{definition} Keys :: \(\langle a \Rightarrow_0 b::\text{zero} \rangle \) \text{ set} \Rightarrow \langle a \rangle \text{ set}

where Keys F = \(\bigcup (\text{keys } \cdot F)\)

\textbf{lemma in-Keys}: \(s \in \text{Keys } F \leftrightarrow (\exists f \in F. s \in \text{keys } f)\)

\textit{unfolding} Keys-def by simp

\textbf{lemma in-KeysI}:

assumes \(s \in \text{keys } f \text{ and } f \in F\)

shows \(s \in \text{Keys } F\)

\textit{unfolding} in-Keys using assms ..

\textbf{lemma in-KeysE}:

assumes \(s \in \text{Keys } F\)

obtains \(f \text{ where } s \in \text{keys } f \text{ and } f \in F\)

using assms unfolding in-Keys ..

\textbf{lemma Keys-mono}:

assumes \(A \subseteq B\)

shows \(\text{Keys } A \subseteq \text{Keys } B\)

using assms by (auto simp add: Keys-def)

\textbf{lemma Keys-insert}: \(\text{Keys } (\text{insert } a A) = \text{keys } a \cup \text{Keys } A\)

by (simp add: Keys-def)

\textbf{lemma Keys-Un}: \(\text{Keys } (A \cup B) = \text{Keys } A \cup \text{Keys } B\)

by (simp add: Keys-def)

\textbf{lemma finite-Keys}:

assumes \(\text{finite } A\)

shows \(\text{finite } (\text{Keys } A)\)

\textit{unfolding} Keys-def by (rule, fact assms, rule finite-keys)

\textbf{lemma Keys-not-empty}:

assumes \(a \in A\) \text{ and } \(a \neq 0\)
shows \( \text{Keys } A \neq \{ \} \)
proof
 assume \( \text{Keys } A = \{ \} \)
 from \( \{ a \neq 0 \} \) have \( \text{keys } a \neq \{ \} \) using aux by fastforce
 then obtain \( s \) where \( s \in \text{keys } a \) by blast
 from this assms(1) have \( s \in \text{Keys } A \) by (rule in-KeysI)
 with \( \{ \text{Keys } A = \{ \} \} \) show False by simp
qed

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

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

lemma \( \text{keys-subset-Keys} \):
 assumes \( f \in F \)
 shows \( \text{keys } f \subseteq \text{Keys } F \)
using in-KeysI [OF - assms] by auto

lemma \( \text{Keys-minus} \):
\( \text{Keys } (A - B) \subseteq \text{Keys } A \)
by (auto simp add: Keys-def)

lemma \( \text{Keys-minus-zero} \):
\( \text{Keys } (A - \{ 0 \}) = \text{Keys } A \)
proof (cases \( 0 \in A \))
 case True
 hence \( (A - \{ 0 \}) \cup \{ 0 \} = A \) by auto
 hence \( \text{Keys } A = \text{Keys } ((A - \{ 0 \}) \cup \{ 0 \}) \) by simp
 also have \( \ldots = \text{Keys } (A - \{ 0 \}) \cup \text{Keys } \{ 0 :: (a \Rightarrow b) \} \) by (fact Keys-Un)
 also have \( \ldots = \text{Keys } (A - \{ 0 \}) \) 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 \( \text{except-fun} :: (a \Rightarrow b) \Rightarrow 'a set \Rightarrow (a \Rightarrow b::zero) \)
 where \( \text{except-fun } f \ S = (\lambda x. (f x \text{ when } x \notin S)) \)
lift-definition \( \text{except} :: (a \Rightarrow 0 'b) \Rightarrow 'a set \Rightarrow (a \Rightarrow 0 'b::zero) \) is \( \text{except-fun} \)
proof
 fix \( p :: a \Rightarrow 'b \) and \( S :: 'a \) set
 assume finite \( \{ t. p t \neq 0 \} \)
 show finite \( \{ t. \text{except-fun } p \ S \ t \neq 0 \} \)
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

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 $\text{lookup (except p S) t} = \text{lookup p t}$
using assms by (simp add: lookup-except)

lemma lookup-except-eq-zeroI:
assumes $t \in S$
shows $\text{lookup (except p S) t} = 0$
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 $\text{keys p} \subseteq S$
shows $\text{except p S} = 0$
proof (rule poly-mapping-eqI, simp)
fix $t$
show $\text{lookup (except p S) t} = 0$
proof (cases $t \in S$)
case True
thus $\text{thesis}$ by (rule lookup-except-eq-zeroI)
next
case False then show $\text{thesis}$
  by (metis assms in_keys_iff lookup-except-eq-idI subset_eq)
qed

lemma except-eq-zeroE:
assumes $\text{except p S} = 0$
shows $\text{keys p} \subseteq S$
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

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 (λp 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: \((x, y). x < y → y ∉ Q\)
  ?Q0 by auto
  from z0-in obtain z where z0-def: z0 = card (keys z) and z ∈ Q by auto
  show ∃y::('a, 'b) poly-mapping
  proof (intro bezI[of - z], rule, rule)
    fix y::('a, 'b) poly-mapping
    let ?y0 = card (keys y)
    assume (y, z) ∈ {(p, q). keys p ⊂ keys q} → y ∉ Q
    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
qed

50
lemma poly-mapping-except-induct':
assumes \( \bigwedge p. (\bigwedge t. t \in \text{keys } p \implies P (\text{except } p \{ t \})) \implies P \) p
shows P p
proof (induct card (\text{keys } p) arbitrary: p)
case 0
with \text{finite-keys[of } p\text{]} have \text{keys } p = {} by simp
show ?case by (rule assms, simp add: \langle \text{keys } p = {} \rangle)
next
case step: \((\text{Suc } n)\)
show ?case
proof (rule assms)
fix t
assume t \in \text{keys } p
show P (\text{except } p \{ t \})
proof (rule step(1), simp add: \text{keys-except})
from step(2) \(t \in \text{keys } p\): \text{finite-keys[of } p\text{]} show \(n = \text{card } (\text{keys } p - \{ t \})\) by simp
qed
qed
qed

lemma poly-mapping-plus-induct:
assumes P 0 and \(\bigwedge p c t. c \neq 0 \implies t \notin \text{keys } p \implies P p \implies P (\text{Poly-Mapping.single } t c + p)\)
shows P p
proof (induct card (\text{keys } p) arbitrary: p)
case 0
with \text{finite-keys[of } p\text{]} have \text{keys } p = {} by simp
hence p = 0 by simp
with \text{assms}(1) show ?case by simp
next
case step: \((\text{Suc } n)\)
from step(2) obtain t where t: t \in \text{keys } p by (metis \text{card-eq-SucD insert-iff})
define c where c = \text{lookup } p t
define q where q = \text{except } p \{ t \}
have \*: p = \text{Poly-Mapping.single } t c + q
by (rule poly-mapping-eqI, simp add: \text{lookup-when-def}\text{.lookup-single Poly-Mapping.when-def}, intro conjI impI,
    simp add: \text{q-def lookup-except c-def}, simp add: \text{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 \( t \notin \text{keys } q \) by (simp add: q-def keys-except)
next
  show \( P q \)
proof (rule step(1))
  from step(2) \( t \in \text{keys } p \) show \( n = \text{card } (\text{keys } q) \) unfolding q-def keys-except
    by (metis Suc-inject card. remove finite-keys)
qeda
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 \( \text{keys } p \subseteq U \cup V \)
  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 \( \ldots = \) except \( p \) \( U \) + except \( p \) \( V \) by (fact except-Un-plus-Int)
finally show \( \text{?thesis} \).
qeda

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 \( p \) (\( U \cap \text{keys } p \)) = 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 \( \ldots = \) except \( p \) (\( - U \)) by simp
  finally show \( \text{?thesis} \).
qeda

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)

52
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 

**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 \( c \) \( \Rightarrow \) \( a \) adds \( c \)
by (simp add: adds-def add.assoc) blast

**lemma** plus-adds-right: \( a + b \) adds \( c \) \( \Rightarrow \) \( b \) 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: \( s + t = 0 \) \( \Rightarrow \) \( s = 0 \)
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 \( 0 \) \( \iff \) \( (s = 0) \)
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 \( t \text{ adds } u \land s \text{ adds } (u - t) \)

proof
  from plus-adds-right[OF a1] show \( t \text{ adds } u \) .
next
  from plus-adds-0[OF a1] show \( s \text{ adds } (u - t) \) .
qed

lemma minus-plus:
  assumes \( s \text{ adds } t \)
  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 \( s \text{ adds } t \) and \( u \text{ adds } v \)
  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 \( u \text{ adds } t \) and \( s \text{ adds } u \)
  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 \( \text{lcs-powerprod} \) 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 \( \text{lcs-powerprod} \) = \( \text{comm-powerprod} \) +
  fixes \( \text{lcs}::'a \Rightarrow 'a \Rightarrow 'a \)
  assumes \( \text{adds-lcs}: s \text{ adds } (lcs \ s \ t) \)
  assumes \( \text{lcs-adds}: s \text{ adds } u \Longrightarrow t \text{ adds } u \Longrightarrow (lcs \ s \ t) \text{ adds } u \)
  assumes \( \text{lcs-comm}: lcs \ s \ t = lcs \ t \ s \)
begin

lemma \( \text{adds-lcs-2}: t \text{ adds } (lcs \ s \ t) \)
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
done
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 ∗: ∀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 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 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 ⇒ 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
  59
thus $x$ adds $z$ by (rule adds-trans)

next
from assms show almost-full-on (adds) \{x. \ d \ x \leq \ n\} by (rule dickson-gradingD2)

qed

lemma dickson-gradingE:
assumes dickson-grading d and $\forall i\cdot (\text{seq} \cdot \text{nat} \Rightarrow 'a::plus) \ i \leq n$
obtains $i \ j$ where $i < j$ and seq i adds seq j

proof –
from assms(1) have almost-full-on (adds) \{x. \ d \ x \leq \ n\} by (rule dickson-gradingD2)
moreover from assms(2) have $\forall i. \ \text{seq} \ i \in \{x. \ d \ x \leq n\}$ 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 $d \ s \leq d \ t$

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 $d \ (t - s) \leq d \ t$

proof –
from assms(2) obtain $u$ where $t = s + u$ ..
hence $t - s = u$ by simp
from assms(1) have $d \ t = \text{ord-class} \cdot \max (d \ s) (d \ u)$ unfolding $:t = s + u$ by (rule dickson-gradingD1)
thus \thesis by (simp add: $\langle t - s = u \rangle$)

qed

lemma dickson-grading-lcs:
assumes dickson-grading d
shows $d \ (\text{lcs} \ s \ t) \leq \max (d \ s) (d \ t)$

proof –
from assms have $d \ (\text{lcs} \ s \ t) \leq d \ (s + t)$ 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 $d \ (\text{lcs} \ s \ t - s) \leq \max (d \ s) (d \ t)$

proof –
from assms have $d \ (\text{lcs} \ s \ t - s) \leq d \ (\text{lcs} \ s \ t)$ by (rule dickson-grading-minus,
intro adds-lcs
also from assms have \( \ldots \leq \max (d_s, d_t) \) by (rule dickson-grading-lcs)
finally show \(?thesis\).
qed

lemma dgrad-set-leI:
assumes \( \forall s. \ s \in S \implies \exists t \in T. \ d_s \leq d_t \)
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 \( s \in S \)
obtains \( t \) where \( t \in T \) and \( d_s \leq d_t \)
using assms by (auto simp: dgrad-set-le-def)

lemma dgrad-set-exhaust-expl:
assumes finite F
shows \( F \subseteq dgrad-set d (\Max (d^\prime F)) \)
proof
fix f
assume \( f \in F \)
hence \( d_f \in d^\prime F \) by simp
with - have \( d_f \leq \Max (d^\prime F) \)
proof (rule Max-ge)
  from assms show finite \( (d^\prime F) \) by auto
qed
hence dgrad-set d \( (d_f) \subseteq dgrad-set d (\Max (d^\prime F)) \) by (auto simp: dgrad-set-def)
moreover have \( f \in dgrad-set d (d_f) \) by (simp add: dgrad-set-def)
ultimately show \( f \in dgrad-set d (\Max (d^\prime F)) \).
qed

lemma dgrad-set-exhaust:
assumes finite F
obtains \( m \) where \( F \subseteq dgrad-set d m \)
proof
from assms show \( F \subseteq dgrad-set d (\Max (d^\prime F)) \) 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 \( s \in S \)
with assms(1) obtain \( t \) where \( t \in T \) and \( I: d_s \leq d_t \) by (auto simp add: dgrad-set-le-def)
from assms(2) this(1) obtain \( u \) where \( u \in U \) and \( 2: d_t \leq d_u \) by (auto simp add: dgrad-set-le-def)
from this(1) show $\exists u \in U. \ d s \leq d u$
proof
from 1 2 show $d s \leq d u$ by (rule le-trans)
qed
qed

lemma dgrad-set-le-Un: dgrad-set-le d $(S \cup T)$ U $\leftrightarrow$ (dgrad-set-le d S U $\land$ dgrad-set-le d T U)
by (auto simp add: dgrad-set-le-def)

lemma dgrad-set-le-subset:
assumes $S \subseteq T$
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 $G \subseteq dgrad-set d m$
shows $F \subseteq dgrad-set d m$
proof
fix f
assume $f \in F$
with assms(1) obtain $g$ where $g \in G$ and $*: d f \leq d g$ by (auto simp add: dgrad-set-le-def)
from assms(2) this(1) have $g \in dgrad-set d m$ ..
hence $d g \leq m$ by (simp add: dgrad-set-def)
with $*$ have $d f \leq m$ by (rule le-trans)
thus $f \in dgrad-set d m$ by (simp add: dgrad-set-def)
qed

lemma dgrad-set-dgrad: $p \in dgrad-set d (d p)$
by (simp add: dgrad-set-def)

lemma dgrad-setI [intro]:
assumes $d t \leq m$
shows $t \in dgrad-set d m$
using assms by (auto simp: dgrad-set-def)

lemma dgrad-setD:
assumes $t \in dgrad-set d m$
shows $d t \leq m$
using assms by (simp add: dgrad-set-def)

lemma dgrad-set-zero [simp]: dgrad-set $(\lambda-. 0)$ m = UNIV
by auto

lemma subset-dgrad-set-zero: $F \subseteq dgrad-set (\lambda-. 0)$ m
by simp

lemma dgrad-set-subset:
  assumes \( m \leq n \)
  shows \( \text{dgrad-set } d \ m \subseteq \text{dgrad-set } d \ n \)
  using assms by (auto simp: dgrad-set-def)

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

lemma dgrad-set-closed-minus:
  assumes \( \text{dickson-grading } d \) and \( s \in \text{dgrad-set } d \ m \) and \( \text{t adds} \ (s::\text{a::cancel-ab-semigroup-add}) \)
  shows \( s - t \in \text{dgrad-set } d \ m \)
  proof
  from assms\( (1, 3) \) have \( d \ (s - t) \leq d \ s \) by (rule dickson-grading-minus)
  also from assms\( (2) \) have \( \ldots \leq m \) by (simp add: dgrad-set-def)
  finally show ?thesis by (simp add: dgrad-set-def)
  qed

lemma dgrad-set-closed-lcs:
  assumes \( \text{dickson-grading } d \) and \( s \in \text{dgrad-set } d \ m \) and \( t \in \text{dgrad-set } d \ m \)
  shows \( \text{lcs } s \ t \in \text{dgrad-set } d \ m \)
  proof
  from assms\( (1) \) have \( d \ (\text{lcs } s \ t) \leq \text{ord-class}.\text{max} \ (d \ s) \ (d \ t) \) by (rule dickson-grading-lcs)
  also from assms\( (2, 3) \) have \( \ldots \leq m \) by (simp add: dgrad-set-def)
  finally show ?thesis by (simp add: dgrad-set-def)
  qed

lemma dickson-gradingD-dgrad-set: \( \text{dickson-grading } d \Rightarrow \text{almost-full-on} \ (\text{adds}) \ (\text{dgrad-set } d \ m) \)
  by (auto dest: dickson-gradingD2 simp: dgrad-set-def)

lemma ex-finite-adds:
  assumes \( \text{dickson-grading } d \) and \( S \subseteq \text{dgrad-set } d \ m \)
  obtains \( T \) where \( \text{finite } T \) and \( T \subseteq S \) and \( \forall s. \ s \in S \Rightarrow (\exists t \in T. \ \text{t adds} \ (s::\text{a::cancel-comm-monoid-add})) \)
  proof
  have reflp \((\text{adds})::\text{a} \Rightarrow -\) by (simp add: reflp-def)
  moreover from assms\( (2) \) have \( \text{almost-full-on} \ (\text{adds}) \ S \)
      proof (rule almost-full-on-subset)
        from assms\( (1) \) show \( \text{almost-full-on} \ (\text{adds}) \ (\text{dgrad-set } d \ m) \) by (rule dickson-gradingD-dgrad-set)
      qed
      ultimately obtain \( T \) where \( \text{finite } T \) and \( T \subseteq S \) and \( \forall s. \ s \in S \Rightarrow (\exists t \in T. \ \text{t adds} \ (s::\text{a::cancel-comm-monoid-add})) \)
      qed

63
Class graded-dickson-powerprod is a slightly artificial construction. It is needed, because type nat ⇒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-group-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 \( \text{le-imp-add: } a \leq b \Rightarrow (\exists c. b = a + c) \)

context canonically-ordered-monoid-add

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 \( \text{max } a b = a + y \)
  unfolding max-def
  proof (cases \( a \leq b \))
    case True
    hence \( b = a + (b - a) \) by (rule le-imp-inv)
    then obtain \( c \) where \( \text{eq: } b = a + c \) ..
    show \(?thesis\)
      proof
        from True show \( \text{max } a b = a + c \) unfolding max-def eq by simp
      qed
    next
    case False
    show \(?thesis\)
      proof
        from False show \( \text{max } a b = a + 0 \) unfolding max-def by simp
      qed
    qed

lemma min-plus-max:
  shows \( (\text{min } a b) + (\text{max } a b) = a + b \)
  proof (cases \( a \leq b \))
    case True
    thus \(?thesis\) unfolding min-def max-def by simp
  next
    case False
thus \textit{thesis} unfolding \textit{min-def max-def} by (simp add: ac-simps)

qed

eend

class add-linorder-min = add-linorder +
  assumes zero-min: \(0 \leq x\)
begin

subclass ninv-comm-monoid-add
proof
  fix \(x \ y\)
  assume \(*\): \(x + y = 0\)
  show \(x = 0\)
  proof
    from zero-min[of \(x\)] have \(0 = x \lor x > 0\) by auto
    thus \textit{thesis}
  proof
    assume \(x > 0\)
    have \(0 \leq y\) by (fact zero-min)
    also have \(\ldots = 0 + y\) by simp
    also from \(\langle x > 0 \rangle\) have \(\ldots < x + y\) by (rule add-strict-right-mono)
    finally have \(0 < x + y\).
    hence \(x + y \neq 0\) by simp
    from this * show \textit{thesis} ..
  qed simp
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 \textit{nat} :: add-linorder
begin

instance by (standard, simp)
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 ≤ t
  assumes plus-monotone: \( s ≤ t \implies s + u ≤ t + u \)
begin

abbreviation ord-conv (infixl ≥ 50) where ord-conv ≡ \((≤)^{-1}\)
abbreviation ord-strict-conv (infixl \(\succ\)) where ord-strict-conv \(\equiv (\prec)^{-1^{-1}}\)

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.eq-iff by simp
  hence \(t = s\) by simp
  thus \(s \preceq t\) by simp
qed

lemma ord-adds:
  assumes \(s \text{ adds } t\)
  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 \(s + u < t + u\)
  shows \(s < t\)
using assms ord-canc[of s 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 \(u + s < u + t\)
  shows \(s < t\)
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 \(s < t\)
  shows \(s + u < t + u\)
using assms
by \((simp\ add:\ ordered-powerprod-lin.order.strict-iff-order\ plus-monotone)\)

**Lemma** \(\text{plus-monotone-strict-left}\):

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

end

**Locale** \(\text{gd-powerprod}\) =

- ordered-powerprod ord ord-strict
- for \(\text{ord} : 'a \Rightarrow 'a\Rightarrow\ True\ (\text{infixl} \leq 50)\)
- and \(\text{ord-strict} : (\text{infixl} < 50)\)
- begin

**Definition** \(\text{dickson-le} :: ('a \Rightarrow \text{nat}) \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}\)

**Where** \(\text{dickson-le} d m s t \iff (d s \leq m \land d t \leq m \land s \succeq t)\)

**Definition** \(\text{dickson-less} :: ('a \Rightarrow \text{nat}) \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}\)

**Where** \(\text{dickson-less} d m s t \iff (d s \leq m \land d t \leq m \land s < t)\)

**Lemma** \(\text{dickson-leI}\):

- assumes \(d s \leq m\) and \(d t \leq m\) and \(s \succeq t\)
- shows \(\text{dickson-le } d m s t\)
- using \(\text{assms}\) by \((simp\ add: \text{dickson-le-def})\)

**Lemma** \(\text{dickson-leD1}\):

- assumes \(\text{dickson-le } d m s t\)
- shows \(d s \leq m\)
- using \(\text{assms}\) by \((simp\ add: \text{dickson-le-def})\)

**Lemma** \(\text{dickson-leD2}\):

- assumes \(\text{dickson-le } d m s t\)
- shows \(d t \leq m\)
- using \(\text{assms}\) by \((simp\ add: \text{dickson-le-def})\)

**Lemma** \(\text{dickson-leD3}\):

- assumes \(\text{dickson-le } d m s t\)
- shows \(s \succeq t\)
- using \(\text{assms}\) by \((simp\ add: \text{dickson-le-def})\)

**Lemma** \(\text{dickson-le-trans}\):

- assumes \(\text{dickson-le } d m s t\) and \(\text{dickson-le } d m t u\)
- shows \(\text{dickson-le } d m s u\)
- using \(\text{assms}\) by \((auto\ simp\ add: \text{dickson-le-def})\)

**Lemma** \(\text{dickson-lessI}\):

- assumes \(d s \leq m\) and \(d t \leq m\) and \(s < t\)
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 ≤ m
  using assms by (simp add: dickson-less-def)

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

lemma dickson-lessD3:
  assumes dickson-less d m s t
  shows s ≺ 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
qed

**lemma** \textit{wf-dickson-less}:

**assumes** dickson-grading \(d\)

**shows** \(\text{wfP} (\text{dickson-less} \ d \ m)\)

**proof** (rule \textit{wfP-chain})

\(\text{show } \neg (\exists \ seq. \ \forall \ i. \ \text{dickson-less} \ d \ m \ (\text{seq} (\text{Suc} \ i)) \ (\text{seq} \ i))\)

**proof**

\(\text{assume } \exists \ seq. \ \forall \ i. \ \text{dickson-less} \ d \ m \ (\text{seq} (\text{Suc} \ i)) \ (\text{seq} \ i)\)

\(\text{then obtain } \text{seq} :: \text{nat} \Rightarrow \ 'a \ \text{where } \forall \ i. \ \text{dickson-less} \ d \ m \ (\text{seq} (\text{Suc} \ i)) \ (\text{seq} \ i) \ ..\)

\(\text{hence } \forall i. \ \text{dickson-less} \ d \ m \ (\text{seq} (\text{Suc} \ i)) \ (\text{seq} \ i) \ ..\)

\(\text{with transp-dickson-less have seq-decr: } \forall i. j < j \implies \text{dickson-less} \ d \ m \ (\text{seq} j) \ (\text{seq} i)\)

\(\text{by (rule transp-sequence)}\)

\(\text{from assms obtain } i j \text{ where } i < j \text{ and } i\text{-adds-j: seq i adds seq j}\)

**proof** (rule dickson-gradingE)

\(\text{fix } i\)

\(\text{from } \ast \text{ show } d \ (\text{seq} \ i) \leq m \text{ by (rule dickson-lessD2)}\)

**qed**

\(\text{from } \ast \text{ show } \text{seq i adds seq j by (rule ord-adds)}\)

\(\text{ultimately show False by simp}\)

**qed**

**end**

\(\text{gd-powerprod stands for graded ordered Dickson power-products.}\)

**locale** \textit{od-powerprod} =

\(\text{ordered-powerprod ord ord-strict}\)

\(\text{for ord: } 'a \Rightarrow 'a::dickson-powerprod \Rightarrow bool (\text{infixl} \leq 50)\)

\(\text{and ord-strict (infixl} < 50)\)

**begin**

**sublocale** \textit{gd-powerprod} by standard

**lemma** \textit{wf-ord-strict}: \(\text{wfP} (~)\)

**proof** (rule \textit{wfP-chain})

\(\text{show } \neg (\exists \ seq. \ \forall i. \ \text{seq} (\text{Suc} \ i) \prec \text{seq} i)\)

**proof**

\(\text{assume } \exists \ seq. \ \forall i. \ \text{seq} (\text{Suc} \ i) \prec \text{seq} i\)

\(\text{then obtain } \text{seq} :: \text{nat} \Rightarrow 'a \ \text{where } \forall i. \ \text{seq} (\text{Suc} \ i) \prec \text{seq} i \ ..\)

\(\text{hence } \forall i. \ \text{seq} (\text{Suc} \ i) \prec \text{seq} i \ ..\)

\(\text{with ordered-powerprod-lin.transp-less have seq-decr: } \forall i. j < j \implies (\text{seq} j) \prec (\text{seq} i)\)

\(\text{by (rule transp-sequence)}\)

71
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
hence seq j ≤ seq i and seq j ≠ seq i by auto
from seq j ≠ seq i ∨ seq j ≤ seq i ord-adds[OF i-adds-j]
  ordered-powerprod-lin.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 \A 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 \A 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 $\{ x. (h x (f x) (g x) \neq 0) \lor f x \neq 0 \}$ by (simp only: Collect-disj-eq)

have $\{ x. g x \neq 0 \} \subseteq \{ x. (h x (f x) (g x) \neq 0) \lor f x \neq 0 \}$

by (intro Collect-mono, rule, rule disjCI, simp add: asms(3))

from this finite-union show finite $\{ x. g x \neq 0 \}$ by (rule finite-subset)

qed

lemma finite-neq-0-inv':
  assumes inf-A: finite $\{ x. h (f x) (g x) \neq 0 \}$ and fin-B: finite $\{ x. f x \neq 0 \}$ and
  $\forall x. h 0 x = x$
  shows finite $\{ x. g x \neq 0 \}$
  using asms 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 ⇒ '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 ⇒ 'b belongs to class lcs-powerprod

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

definition lcs-fun::('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'b) where lcs f g = (λx. max (f x) (g x))

lemma adds-funI:
  assumes s ≤ t
  shows s adds (t::'a ⇒ 'b)
proof (rule addsI, rule)
  fix x
  from asms have s x ≤ 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 ⇒ 'b) ⇔ (∀ x. f x adds g x)
  unfolding adds-def plus-fun-def by metis

lemma adds-fun-iff': f adds (g::'a ⇒ 'b) ⇔ (∀ x. ∃ 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 ⇒ '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 ⇒ '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 ⇒ 'b)
  shows (lcs s t) adds u
  using assms unfolding lcs-fun-def adds-fun-iff'
proof –
  assume a1: ∀x. ∃y. u x = s x + y and a2: ∀x. ∃y. u x = t x + y
  show ∀x. ∃y. u x = max (s x) (t x) + y
  proof
    fix x
    from a1 have b1: ∃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 implI, 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

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)

74
qed (rule adds-funI)

lemma gcs-fun: gcs s (t::'a ⇒ ('b::add-linorder)) = (λx. 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-supf: 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-if: supp-fun f = {} ↔ f = 0
  by (auto simp: supp-fun-def)

lemma sub-supf-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-zero1:
    assumes \( \forall x. x \in \text{supp-fun } f \implies f x = 0 \)
    shows \( f = 0 \)
proof (rule, simp)
  fix \( x \)
  show \( f x = 0 \)
  proof (cases \( x \in \text{supp-fun } f \))
    case True
    then show \( ?\text{thesis} \) by (rule assms)
  next
    case False
    then show \( ?\text{thesis} \) by (simp add: supp-fun-def)
  qed
qed

lemma except-fun-cong1:
    \( \text{supp-fun } s \cap ((V - U) \cup (U - V)) \subseteq {} \implies \text{except-fun } s V = \text{except-fun } s U \)
by (auto simp: except-fun-def when-def supp-fun-def intro! ext)

lemma adds-except-fun:
    \( s \text{ adds } t = (\text{except-fun } s V \text{ adds } \text{except-fun } t V \land \text{except-fun } s (- V) \text{ adds } \text{except-fun } t (- V)) \)
    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:
    \( s \text{ adds } t = (\text{except-fun } s \{ v \} \text{ adds } \text{except-fun } t \{ v \} \land s v \text{ adds } t v) \)
    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) \( \{ x::'a ⇒ 'b::add-wellorder. \text{supp-fun } x \subseteq V \} \)
    using assms
proof (induct \( V \))
  case empty
  have finite \( \{ \emptyset \} \) by simp
moreover have refl-on (adds) \( \{ \emptyset::'a ⇒ 'b \} \) by (simp add: refl-on-def)
ultimately have almost-full-on (adds) \( \{ \emptyset::'a ⇒ 'b \} \) by (rule finite-almost-full-on)
thus \( ?\text{case} \) by (simp add: supp-fun-eq-zero-iff)

76
next

\[\text{case } (\text{insert } v \ V)\]
\[\text{show } ?\text{case}\]
\[\text{proof (rule almost-full-onI)}\]
\[\text{fix } \text{seq} :: \text{nat} \Rightarrow 'a \Rightarrow 'b\]
\[\text{assume } \forall \ i. \ \text{seq} \ i \in \{ x. \ \text{supp-fun} \ x \subseteq \text{insert } v \ V \}\]
\[\text{hence } a : \text{supp-fun} \ (\text{seq} \ i) \subseteq \text{insert } v \ V \text{ for } i \text{ by simp}\]
\[\text{define } \text{seq}' : \text{where } \text{seq}' = (\lambda i. \ (\text{except-fun} \ (\text{seq} \ i) \ {v}) \ (\text{except-fun} \ (\text{seq} \ i) \ V))\]
\[\text{have } \text{almost-full-on} \text{ (adds) } \{ x::'a \Rightarrow 'b. \ \text{supp-fun} \ x \subseteq \{v\}\}\]
\[\text{proof (rule almost-full-onI)}\]
\[\text{fix } f :: \text{nat} \Rightarrow 'a \Rightarrow 'b\]
\[\text{assume } \forall i. \ f \ i \in \{ x. \ \text{supp-fun} \ x \subseteq \{v\}\}\]
\[\text{hence } b : \text{supp-fun} \ (f \ i) \subseteq \{v\} \text{ for } i \text{ by simp}\]
\[\text{let } ?f = (\lambda i. \ f \ i \ v)\]
\[\text{have wfP } ((<::'b \Rightarrow -) \text{ by (simp add: wfP-def)}\]
\[\text{hence } \forall f :: \text{nat} \Rightarrow 'b. \exists i. \ f \ i \leq f \ (\text{Suc} \ i)\]
\[\text{by (simp add: wf-iff-no-infinite-down-chain[to-pred] not-less)}\]
\[\text{then obtain } i \text{ where } ?f \ i \leq ?f \ (\text{Suc} \ i)\]
\[\text{have } i < \text{Suc} \ i \text{ by simp}\]
\[\text{moreover have } f \ i \text{ adds } f \ (\text{Suc} \ i) \text{ unfolding adds-fun-iff}\]
\[\text{proof}\]
\[\text{fix } x\]
\[\text{show } f \ i \ x \text{ adds } f \ (\text{Suc} \ i) \ x\]
\[\text{proof (cases } x = v\)
\[\text{case True}\]
\[\text{with } \langle ?f \ i \leq ?f \ (\text{Suc} \ i) \rangle \text{ show } \text{thesis by (simp add: adds-def le-iff-add)}\]
\[\text{next}\]
\[\text{case False}\]
\[\text{with } b \text{ have } x \notin \text{supp-fun} \ (f \ i) \text{ and } x \notin \text{supp-fun} \ (f \ (\text{Suc} \ i)) \text{ by blast+}\]
\[\text{thus } \text{thesis by (simp add: supp-fun-def)}\]
\[\text{qed}\]
\[\text{qed}\]
\[\text{ultimately show } \text{good (adds) } f \text{ by (meson goodI)}\]
\[\text{qed}\]
\[\text{with } \text{insert}(\beta) \text{ have}\]
\[\text{almost-full-on} \text{ (prod-le (adds) (adds)) } \{(x::'a \Rightarrow 'b. \ \text{supp-fun} \ x \subseteq V) \times \{x::'a \Rightarrow 'b. \ \text{supp-fun} \ x \subseteq \{v\}\}\}\]
\[\text{(is almost-full-on } ?P \ ?A) \text{ by (rule almost-full-on-Sigma)}\]
\[\text{moreover from } a \text{ have } \text{seq}' \ i \in ?A \text{ for } i \text{ by (auto simp add: seq'-def supp-except-fun)}\]
\[\text{ultimately obtain } i \ j \text{ where } i < j \text{ and } ?P \ (\text{seq}' \ i) \ (\text{seq'} \ j) \text{ by (rule almost-full-onD)}\]
\[\text{have } \text{seq} \ i \text{ adds } \text{seq} \ j \text{ unfolding adds-except-fun[where } s=\text{seq} \ i \text{ and } V=V]\]
\[\text{proof}\]
\[\text{from } (?P \ (\text{seq}' \ i) \ (\text{seq'} \ j)) \text{ show } \text{except-fun} \ (\text{seq} \ i) \ V \text{ adds } \text{except-fun} \ (\text{seq} \ j) \ V\]
\[\text{by (simp add: prod-le-def seq'-def)}\]
\[\text{next}\]

77
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) moreove 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

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 ≠ (t::'a ⇒ 'b)
  obtains x where s x ≠ t x and ∃y. s y ≠ t y =⇒ x ≤ y
proof –
  from assms ext[of s t] have ∃x. s x ≠ t x by auto
  with exists-least-iff[of λx. s x ≠ t x]
  obtain x where x1: s x ≠ t x and x2: s y < x =⇒ s y = t y
  by auto
  show ?thesis
proof
  from x1 show s x ≠ t x .
next
  fix y
  assume s y ≠ t y
  with x2[of y] have ¬ y < x by auto
  thus x ≤ y by simp
qed

definition lex-fun::('a ⇒ 'b) ⇒ ('a ⇒ 'b::order) ⇒ bool where
\[ \text{lex-fun } s \ t \equiv (\forall x. s \ x \leq t \ x \lor (\exists y. s \ y \neq t \ y)) \]

**Definition** \( \text{lex-fun-strict } s \ t \iff \text{lex-fun } s \ t \land \neg \text{lex-fun } t \ s \)

Attention! \( \text{lex-fun} \) 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** \( \text{lex-fun-alt} \):

shows \( \text{lex-fun } s \ t = (s = t \lor (\exists x. s \ x < t \ x \land (\forall y. s \ y = t \ y))) \) (is \( ?L = \ ?R \))

**Proof**

assume \( ?L \)

show \( ?R \)

proof

(cases \( s = t \))

assume \( s = t \)

thus \( ?R \) by simp

next

assume \( s \neq t \)

from \( \text{neq-fun-alt}[\text{OF this}] \) obtain \( x0 \)

where \( x0-neq \): \( s \ x0 \neq t \ x0 \) and \( x0-min \): \( \forall z. s \ z \neq t \ z \implies x0 \leq z \) by auto

show \( ?R \)

proof

(intro disjI2, rule exI[of - \( x0 \)], intro conjI)

from \( ?L \) have \( s \ x0 \leq t \ x0 \lor (\exists y. y < x0 \land s \ y \neq t \ y) \) unfolding \( \text{lex-fun-def} \)

thus \( s \ x0 < t \ x0 \)

proof

assume \( s \ x0 \leq t \ x0 \)

from this \( x0-neq \) show \( ?\text{thesis} \) by simp

next

assume \( \exists y. y < x0 \land s \ y \neq t \ y \)

then obtain \( y \) where \( y < x0 \) and \( y-neq \): \( s \ y \neq t \ y \) by auto

from \( y < x0 \) \( x0-min[\text{OF } y-neq] \) show \( ?\text{thesis} \) by simp

qed

next

show \( \forall y < x0. s \ y = t \ y \)

proof

(rule, rule)

fix \( y \)

assume \( y < x0 \)

hence \( \neg x0 \leq y \) by simp

from this \( x0-min[\text{of } y] \) show \( s \ y = t \ y \) by auto

qed

next

assume \( ?R \)

thus \( ?L \)

proof

assume \( s = t \)

thus \( ?\text{thesis} \) by \( (\text{simp add: lex-fun-def}) \)
next
assume \( \exists x. \ s x < t x \land (\forall y < x. \ s y = t y) \)
then obtain \( y \) where \( y: s y < t y \) and \( s \)-min: \( \forall z < y. \ s z = t z \) by auto
show \( \text{thesis} \) unfolding \( \text{lex-fun-def} \)
proof
fix \( x \)
show \( s x \leq t x \lor (\exists y < x. \ s y \neq t y) \)
proof (cases \( s x \leq t x \))
  assume \( s x \leq t x \)
  thus \( \text{thesis} \) by simp
next
assume \( x: \neg s x \leq t x \)
show \( \text{thesis} \)
proof (intro disjI2, rule exI[of - y], intro conjI)
  have \( \neg x \leq y \)
  proof
    assume \( x \leq y \)
    hence \( x < y \lor y = x \) by auto
    thus \( \text{False} \) by (auto simp: preorder-class.less-le-not-le)
  qed
  qed
  thus \( y < x \) by simp
next
from \( y \) show \( s y \neq t y \) by simp
qed
qed
qed

lemma \( \text{lex-fun-refl} \): \( \text{lex-fun} \ s \ s \)
unfolding \( \text{lex-fun-alt} \) by simp

lemma \( \text{lex-fun-antisym} \):
  assumes \( \text{lex-fun} \ s \ t \) and \( \text{lex-fun} \ t \ s \)
  shows \( s = t \)
proof
fix \( x \)
from \text{assms}(1) have \( s = t \lor (\exists x. \ s x < t x \land (\forall y < x. \ s y = t y)) \)
  unfolding \( \text{lex-fun-alt} \).
thus \( s \ x = t \ x \)
proof
assume \( s = t \)
thus \(?thesis\) by simp
next
assume \( \exists x. \ s \ x < t \ x \ \land (\forall y < x. \ s \ y = t \ y) \)
then obtain \( x \) where \( x : s \ x < t x \) and \( x \)-min: \( \forall y < x. \ s \ y = t \ y \) by auto
from \( \text{assms}(2) \) have \( t = s \ \lor (\exists x. \ t \ x < s \ x \ \land (\forall y < x. \ t \ y = s \ y)) \) unfolding lex-fun-alt .

thus \(?thesis\) proof
assume \( t = s \)
thus \(?thesis\) by simp
next
assume \( \exists x. \ s \ x < t \ x \ \land (\forall y < x. \ s \ y = t \ y) \)
then obtain \( x \) where \( x: t x < s x \) and \( x\)-min: \( \forall y < x. \ t \ y = s \ y \) by auto
have \( x0 < x1 \ \lor \ x1 < x0 \ \lor \ x1 = x0 \) using local.antisymconv3 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

lemma lex-fun-trans:
assumes \( \text{lex-fun} \ s \ t \) and \( \text{lex-fun} \ t \ u \)
shows \( \text{lex-fun} \ s \ u \)
proof –
from \( \text{assms}(1) \) have \( s = t \ \lor (\exists x. \ s \ x < t \ x \ \land (\forall y < x. \ s \ y = t \ y)) \) unfolding lex-fun-alt .
thus \(?thesis\)
proof
assume \( s = t \)
from this \( \text{assms}(2) \) show \(?thesis\) by simp
next
assume \( \exists x. \ s \ x < t \ x \ \land (\forall y < x. \ s \ y = t \ y) \)
then obtain \( x0 \) where \( x0: s x < t x0 \) and \( x0\)-min: \( \forall y < x0. \ s \ y = t \ y \)
by auto
from \( \text{assms}(2) \) have \( t = u \ \lor (\exists x. \ t \ x < u x \ \land (\forall y < x. \ t \ y = u \ y)) \) unfolding lex-fun-alt .
thus \(?thesis\)
proof
  assume \( t = u \)
  from this assms(1) show \(?thesis\) by simp
next
  assume \( \exists x . \ t x < u x \land (\forall y < x . \ t y = u y) \)
then obtain \( x1 \) where \( x1 : t x1 < u x1 \) and \( x1\text{-min} : \forall y < x1 . \ t y = u y \) by auto
show \(?thesis\) unfolding lex-fun-alt
  proof (intro disjI2)
  show \( \exists x . \ s x < u x \land (\forall y < x . \ s y = u y) \)
  proof (rule linorder-cases[of \( x0 \) \( x1 \)])
    assume \( x1 < x0 \)
    show \(?thesis\)
    proof (rule exI[of - \( x1 \)], intro conjI)
      fix \( y \)
      assume \( y < x1 \)
      from this \( x1 < x0 \) have \( y < x0 \) by simp
      from \( x0\text{-min} \)\[rule-format\], OF this \( x1\text{-min} \)\[rule-format\], 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 \( x0 < x1 \)\[rule-format\], OF \( x0 < x1 \)\]
    show \( s x1 < u x1 \) by simp
next
  show \( \forall y < x0 . \ s y = u y \)
  proof (intro allI, intro impI)
    fix \( y \)
    assume \( y < x0 \)
    from this \( x0 < x1 \) have \( y < x1 \) by simp
    from \( x0\text{-min} \)\[rule-format\], OF this \( x0\text{-min} \)\[rule-format\], OF \( y < x1 \)\]
    show \( s y = u y \) by simp
  qed
  qed
next
  assume \( x0 = x1 \)
  show \(?thesis\)
  proof (rule exI[of - \( x1 \)], intro conjI)
    from \( x0 = x1 \)\[rule-format\], OF \( x0 = x1 \)\]
    show \( s x1 < u x1 \) by simp
next
  show \( \forall y < x1 . \ s y = u y \)
  proof (intro allI, intro impI)
    fix \( y \)
    from this \( x0 < x1 \) have \( y < x1 \) by simp
    from \( x0\text{-min} \)\[rule-format\], OF this \( x0\text{-min} \)\[rule-format\], OF \( y < x1 \)\]
    show \( s y = u y \) by simp
  qed
assume $y < x_1$

hence $y < x_0$ using $(x_0 = x_1)$ by simp

from $x_0 \text{-} \text{min}[\text{rule-format, OF this}]$ $x_1 \text{-} \text{min}[\text{rule-format, OF } y < x_1]$

show $s \ y = u \ y$ by simp

qed

qed

lemma lex-fun-lin: $\text{lex-fun } s \ t \lor \text{lex-fun } t \ s$ for $s \ t :: \ 'a \Rightarrow 'b :: \ {\text{ordered-comm-monoid-add}, \ \text{linorder}}$

proof (intro disjCI)

assume $\neg \text{lex-fun } t \ s$

hence $a: \forall \ x. \ \neg (t \ x < s \ x) \lor (\exists \ y < x. \ t \ y \neq s \ y)$ unfolding lex-fun-alt by auto

show lex-fun s t unfolding lex-fun-def

proof

fix $x$

from $a$ have $\neg (t \ x < s \ x) \lor (\exists \ y < x. \ t \ y \neq s \ y)$ ..

thus $s \ x \leq t \ x \lor (\exists \ y < x. \ s \ y \neq t \ y)$ by auto

qed

qed

corollary lex-fun-strict-alt [code]:

lex-fun-strict $s \ t = (\neg \text{lex-fun } t \ s)$ for $s \ t :: \ 'a \Rightarrow 'b :: \ {\text{ordered-comm-monoid-add}, \ \text{linorder}}$

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

lemma lex-fun-zero-min: $\text{lex-fun } 0 \ s$ for $s :: 'a \Rightarrow 'b :: \ {\text{add-linorder-min}}$

by (simp add: lex-fun-def zero-min)

lemma lex-fun-plus-monotone:

lex-fun $(s + u) \ (t + u)$ if $\text{lex-fun } s \ t$

for $s \ t :: 'a \Rightarrow 'b :: \ {\text{ordered-cancel-comm-monoid-add}}$

unfolding lex-fun-def

proof

fix $x$

from that have $s \ x \leq t \ x \lor (\exists \ y < x. \ s \ y \neq t \ y)$ unfolding lex-fun-def ..

thus $(s + u) \ x \leq (t + u) \ x \lor (\exists \ y < x. \ (s + u) \ y \neq (t + u) \ y)$

proof

assume $a1: s \ x \leq t \ x$

show $?thesis$

proof (intro disjI1)

from $a1$ show $(s + u) \ x \leq (t + u) \ x$ by (auto simp: add-right-mono)

qed

next

assume $\exists y < x. \ s \ y \neq t \ y$
then obtain $y$ where $y < x$ and $a2: s y \neq t y$ by auto

proof (intro disjI2, rule exI[of - $y$], intro conjI, fact)

  from $a2$ show $(s + u) y \neq (t + u) y$ by (auto simp: add-right-mono)

qed

qed

end

6.7.8 Degree

definition $\deg$-fun::"(′a ⇒ ′b::comm-monoid-add) ⇒ ′b where $\deg$-fun $s ≡ \sum x∈(\operatorname{supp-fun} s). s x$"

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 $*$: $(\sum x∈(\operatorname{supp-fun} s). s x) = 0$ by (simp only: $\deg$-fun-def)
  have $**$: $\forall x. 0 \leq s x$ by (rule zero-min)
  from $*$ have $\forall x. x ∈ \operatorname{supp-fun} s \Longrightarrow s x = 0$ 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 $s \subseteq A$ and finite $A$
  shows $\deg$-fun $s = (\sum x\in A. s x)$

unfoldng $\deg$-fun-def

proof (rule sum.monono-neutral-cong-left, fact, fact, rule)
  fix $x$
  assume $x \in A - \operatorname{supp-fun} s$
  hence $x \notin \operatorname{supp-fun} s$ 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 $(t::′a ⇒ ′b::\text{comm-monoid-add})$

proof
  from assms have fin: finite (supp-fun $s \cup$ supp-fun $t$) by simp
  have $\deg$-fun $(s + t) = (\sum x\in(supp-fun (s + t)). s x + t x)$ 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 (∑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 (∑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 1] deg-fun-superset[OF 1 t]
  proof (rule sum-mono)
    fix i
    from assms(3) show s i ≤ t i unfolding le-fun-def ..

85
6.7.9 General Degree-Orders

context linorder

begin

lemma ex-min:
assumes finite (\(A::\text{\text{\`a set}}\) and \(A \neq \{\}\)
shows \(\exists y \in A. (\forall z \in A. y \leq z)\)
using assms
proof (induct rule: finite-induct)
assume \(\{\} \neq \{\}\)
thus \(\exists y \in \{\}. \forall z \in \{\}. y \leq z\) by simp
next
fix \(a::\text{\text{\`a set}}\) and \(A::\text{\text{\`a set}}\)
assume \(a \notin A\) and \(IH: A \neq \{\} \Rightarrow \exists y \in A. (\forall z \in A. y \leq z)\)
show \(\exists y \in \text{insert} a A. (\forall z \in \text{insert} a A. y \leq z)\)
proof (cases \(A = \{\}\))
  case True
  show \(?\text{thesis}\)
  proof (rule bexI[of - a], intro ballI)
    fix z
    assume z \(\in\) insert a A
    from this True have z = a by simp
    thus a \(\leq\) z by simp
  qed (simp)
next
  case False
  from \(IH[\text{OF False}]\) obtain y where y \(\in\) A and \(y-min: \forall z \in A. y \leq z\) by auto
  from \(linear[\text{of y a}]\) show \(?\text{thesis}\)
  proof
    assume y \(\leq\) a
    show \(?\text{thesis}\)
    proof (rule bexI[of - y], intro ballI)
      fix z
      assume z \(\in\) insert a A
      hence z = a \(\lor\) z \(\in\) A by simp
      thus y \(\leq\) z
    proof
      assume z = a
      from this \((y \leq a)\) show y \(\leq\) z by simp
    next
      assume z \(\in\) A
      from \(y-min[\text{rule-format, OF this}]\) show y \(\leq\) z .
    qed
  next
    from \((y \in A)\) show y \(\in\) insert a A by simp

qed

qed
qed

next
assume $a \leq y$
show ?thesis
proof (rule bexI[of - a], intro ballI)
  fix $z$
  assume $z \in \text{insert} \ a \ A$
  hence $z = a \lor z \in A$ by simp
  thus $a \leq z$
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

definition dord-fun::(('a => 'b::ordered-comm-monoid-add) => ('a => 'b) => bool)
  => ('a => 'b) => ('a => 'b) => bool
where dord-fun ord s t ≡ (let $d_1 = \text{deg-fun} \ s$; $d_2 = \text{deg-fun} \ t$ in ($d_1 < d_2 \lor (d_1 = d_2 \land \text{ord} \ s \ t)$))

lemma dord-fun-degD:
  assumes dord-fun ord s t
  shows deg-fun s $\leq$ 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 $\implies$ ord t s $\implies$ 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 $\lor$ (deg-fun t = deg-fun s $\land$ ord t s)
    unfolding dord-fun-def Let-def .
  from assms(2) have st: deg-fun s $<$ deg-fun t $\lor$ (deg-fun s = deg-fun t $\land$ 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 \( \deg-fun s = \deg-fun t \land \ord s t \)
 hence \( \deg-fun s = \deg-fun t \) and \( \ord s t \) by simp-all
from \( \deg-fun s = \deg-fun t \land \ord s t \) have \( \ord t s \) by simp
with \( \ord s t \) show \(?thesis by (rule ord-antisym)\)
qed
qed

lemma \( \text{dord-fun-trans} \):
assumes \( \text{ord-trans}: \ord s t \Longrightarrow \ord t u \Longrightarrow \ord s u \) and \( \text{dord-fun ord s t} \) and \( \text{dord-fun ord t u} \)
shows \( \text{dord-fun ord s u} \)
proof –
from assms\((3)\) have \( ts: \deg-fun t < \deg-fun u \lor (\deg-fun t = \deg-fun u \land \ord t u) \)
unfolding \( \text{dord-fun-def Let-def} \).
from assms\((2)\) have \( st: \deg-fun s < \deg-fun t \lor (\deg-fun s = \deg-fun t \land \ord s t) \)
unfolding \( \text{dord-fun-def Let-def} \).
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 \( \deg-fun s = \deg-fun t \land \ord s t \)
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 \( \deg-fun t = \deg-fun u \land \ord t u \)
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

lemma \( \text{dord-fun-lin} \):
\( \text{dord-fun ord s t} \lor \text{dord-fun ord t s} \)
if \( \ord s t \lor \ord t s \)
for \( s:t:'a \Rightarrow 'b::\{\text{ordered-comm-monoid-add}, \text{linorder}\} \)
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 s t 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

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::order) ⇒ 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 \texttt{drlex-fun-lin}: \texttt{drlex-fun} \ s \ t \lor \texttt{drlex-fun} \ t \ s  \\
\texttt{for} s \ t::(\'a \Rightarrow \'b::\{ordered-comm-monoid-add, linorder\})  \\
\texttt{unfolding} \ \texttt{drlex-fun-def} \ \texttt{by} (\texttt{rule} \texttt{dord-fun-lin}, \texttt{rule} \texttt{lex-fun-lin})

corollary \texttt{drlex-fun-strict-alt} \ \texttt{[code]}:  \\
\texttt{drlex-fun-strict} \ s \ t = (\neg \texttt{drlex-fun} \ t \ s)  \\
\texttt{for} s \ t::(\'a \Rightarrow \'b::\{ordered-comm-monoid-add, linorder\})  \\
\texttt{unfolding} \ \texttt{drlex-fun-strict-def} \ \texttt{using} \ \texttt{drlex-fun-lin} \ \texttt{by} \ \texttt{auto}

lemma \texttt{drlex-fun-zero-min}:  \\
\texttt{fixes} s \ t::(\'a \Rightarrow \'b::\{add-linorder-min\})  \\
\texttt{assumes} finite (\texttt{supp-fun} \ s)  \\
\texttt{shows} \texttt{drlex-fun} \ 0 \ s  \\
\texttt{unfolding} \ \texttt{drlex-fun-def} \ \texttt{by} (\texttt{rule} \texttt{dord-fun-zero-min}, \texttt{rule} \texttt{lex-fun-refl}, \texttt{fact})

lemma \texttt{drlex-fun-plus-monotone}:  \\
\texttt{fixes} s \ t \ u::(\'a \Rightarrow \'b::\{ordered-cancel-comm-monoid-add, ordered-ab-semigroup-add-imp-le\})  \\
\texttt{assumes} finite (\texttt{supp-fun} \ s) \ \texttt{and} \ finite (\texttt{supp-fun} \ t) \ \texttt{and} \ finite (\texttt{supp-fun} \ u) \ \texttt{and}  \\
\texttt{drlex-fun} \ s \ t  \\
\texttt{shows} \texttt{drlex-fun} \ (s + u) \ (t + u)  \\
\texttt{using} \texttt{lex-fun-plus-monotone[of t s u]} \ \texttt{assms} \ \texttt{unfolding} \ \texttt{drlex-fun-def}  \\
\texttt{by} (\texttt{rule} \texttt{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 \texttt{finite-linorder} = \texttt{finite} + \texttt{linorder}

begin

subclass \texttt{wellorder}

proof

fix \texttt{P}::\'a \Rightarrow bool \ \texttt{and} \ a

assume \texttt{hyp}: \land x. (\land y. (y < x) \Longrightarrow P y) \Longrightarrow P x

show \texttt{P} \ a

proof (\texttt{rule} econtr)

assume \neg \texttt{P} \ a

have finite \ \{x. \neg \texttt{P} \ x\} \ \texttt{(is finite} \ ?A) \ \texttt{by} \ \texttt{simpl}

from \neg \texttt{P} \ a \ \texttt{have} \ a \ \in \ ?A \ \texttt{by} \ \texttt{simpl}

hence \ ?A \ \neq \ \{\} \ \texttt{by} \ \texttt{auto}

from \texttt{ex-min[OF \texttt{finite} ?A \ this]} \ \texttt{obtain} \ b \ \texttt{where} \ b \ \in \ ?A \ \texttt{and \ b-min}: \forall y\in？A. \ b \leq y \ \texttt{by} \ \texttt{auto}

b \leq y \ \texttt{by} \ \texttt{auto}

from \ ?A \ \texttt{have} \ \neg \texttt{P} \ b \ \texttt{by} \ \texttt{simpl}

with \texttt{hyp[of b]} \ \texttt{obtain} \ y \ \texttt{where} \ y < b \ \texttt{and} \ \neg \texttt{P} \ y \ \texttt{by} \ \texttt{auto}

from \neg \texttt{P} \ y \ \texttt{have} \ y \ \in \ ?A \ \texttt{by} \ \texttt{simpl}

with \texttt{b-min} \ \texttt{have} \ b \leq y \ \texttt{by} \ \texttt{simpl}

with \texttt{y < b} \ \texttt{show} \ False \ \texttt{by} \ \texttt{simpl}

qed
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 ⇒_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"
qed
proof
  from * show lookup t = lookup s + lookup k by (simp only: lookup-plus-fun)
qed

next
assume \exists k. lookup t = lookup s + k
then obtain k where *: lookup t = lookup s + k ..
have **: k \in \{f. finite \{x. f x \neq 0\}\}
  proof
    have finite \{x. lookup t x \neq 0\} by transfer
    hence finite \{x. lookup s x + k x \neq 0\} by (simp only: * plus-fun-def)
    moreover have finite \{x. lookup s x \neq 0\} by transfer
    ultimately show finite \{x. k x \neq 0\} by (rule finite-neq-0-inv', simp)
  qed

show \exists 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 ⇒ 'b belongs to class comm-powerprod
instance poly-mapping :: (type, cancel-comm-monoid-add) comm-powerprod
  by standard

6.8.2 'a ⇒ '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 (λk. 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 ⇒ 'b belongs to class lcs-powerprod
instantiation poly-mapping :: (type, add-linorder) lcs-powerprod
  begin
    lift-definition lcs-poly-mapping::('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'b) is λs t.
      λx. max (s x) (t x)
    proof
      fix fun1 fun2 :: 'a ⇒ 'b
      assume finite \{t. fun1 t \neq 0\} and finite \{t. fun2 t \neq 0\}
      from finite-neq-0[OF this, of max] show finite \{t. max (fun1 t) (fun2 t) \neq 0\}
        by (auto simp: max-def)
    qed

94
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 ←→ lookup s ≤ lookup t
  for s t::'a ⇒_0 'b::add-linorder-min
  by (simp only: poly-mapping-adds-iff adds-fun)

lemma lookup-gcs-fun: lookup (gcs s (t::'a ⇒_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 ⇒_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 ∧ except s (- V) adds except t (- V))
  for s t :: 'a ⇒_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 ←→ (except s {v} adds except t {v} ∧ lookup s v adds lookup t v)
  for s t :: 'a ⇒_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 ⇒ 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 ⇒ 0::add-wellorder. keys x ⊆ V}
proof (rule almost-full-onI)
  fix seq::nat ⇒ 'a ⇒ 0
  assume a: ∀ i. seq i ∈ {x::'a ⇒ 0. keys x ⊆ V}
  define seq' where seq' = (λi. lookup (seq i))
  from assms have almost-full-on (adds) {x::'a ⇒ 0. supp-fun x ⊆ V} by (rule Dickson-fan)
  moreover from a have ∀ i. seq' i ∈ {x::'a ⇒ 0. 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 ⇒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

96
thus \( \text{thesis} \).

next

\begin{itemize}
\item case \texttt{False}
\item with \texttt{assms have 1: x \in keys t - X by simp}
\item hence \texttt{keys t - X \neq \{\} by blast}
\item hence eq: varnum X t = Suc (Max (elem-index' (keys t - X))) by \texttt{(simp add: varnum-def)}
\item hence elem-index x < varnum X t using 1 by \texttt{(simp add: less-Suc-eq-le)}
\item thus \text{thesis} ..
\end{itemize}

qed

\begin{itemize}
\item lemma \texttt{varnum-plus:}
\item varnum X (s + t) = max (varnum X s) (varnum X (t::'x::countable \Rightarrow \'b::ninv-comm-monoid-add))
\item proof \texttt{(simp add: varnum-def keys-plus-ninv-comm-monoid-add image-Un Un-Diff del: Diff-eq-empty-iff, intro impI)}
\item assume 1: \texttt{keys s - X \neq \{\} and 2: keys t - X \neq \{\}}
\item have \texttt{finite (elem-index' (keys s - X)) by simp}
\item moreover from 1 have \texttt{elem-index' (keys s - X) \neq \{\} by simp}
\item moreover have \texttt{finite (elem-index' (keys t - X)) by simp}
\item moreover from 2 have \texttt{elem-index' (keys t - X) \neq \{\} by simp}
\item ultimately show \texttt{Max (elem-index' (keys s - X) \cup elem-index' (keys t - X)) = max (Max (elem-index' (keys s - X))) (Max (elem-index' (keys t - X))) by \texttt{(rule Max-Un)}}
\item qed
\end{itemize}

\begin{itemize}
\item lemma \texttt{dickson-grading-varnum:}
\item assumes \texttt{finite X}
\item shows \texttt{dickson-grading ((varnum X)::('x::countable \Rightarrow _b::add-wellorder) \Rightarrow nat)}
\item using \texttt{varnum-plus}
\item proof \texttt{(rule dickson-gradingI)}
\item fix \texttt{m::nat}
\item let \texttt{?V = X \cup \{x. elem-index x < m\}}
\item have \texttt{\{t::'x \Rightarrow _b. varnum X t \leq m\} \subseteq \{t. keys t \subseteq ?V\}}
\item proof \texttt{(rule, simp, intro subsetI, simp)}
\item fix \texttt{t::'x \Rightarrow _b and x::'x}
\item assume \texttt{varnum X t \leq m}
\item assume \texttt{x \in keys t}
\item thus \texttt{x \in X \lor elem-index x < m}
\item proof \texttt{(rule elem-index-less-varnum)}
\item assume \texttt{x \in X}
\item thus \text{thesis} ..
\item next
\item assume \texttt{elem-index x < varnum X t}
\item hence \texttt{elem-index x < m using varnum X t \leq m by \texttt{(rule less-le-trans)}}
\item thus \text{thesis} ..
\item qed
\item qed
\end{itemize}

97
thus almost-full-on (adds) \{ t::'x \Rightarrow_0 'b. varnum X t \leq 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 \subseteq ?V \} by (rule Dickson-poly-mapping)
qed

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

lemma varnum-le-iff: varnum X t \leq n \iff keys t \subseteq X \cup \{ x. \text{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 \iff 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 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 \Rightarrow_0 'b. keys x \subseteq UNIV \} by (rule Dickson-poly-mapping)
  thus almost-full-on (adds) (UNIV::('a \Rightarrow_0 'b) set) by simp
qed

6.8.7 Lexicographic Term Order

definition lex-pm :: ('a \Rightarrow_0 'b) \Rightarrow ('a::linorder \Rightarrow_0 'b::\{zero,linorder\}) \Rightarrow bool
  where lex-pm = (\leq)

definition lex-pm-strict :: ('a \Rightarrow_0 'b) \Rightarrow ('a::linorder \Rightarrow_0 'b::\{zero,linorder\}) \Rightarrow bool
  where lex-pm-strict = (<)

lemma lex-pm-alt: lex-pm s t = (s = t \vee (\exists x. lookup s x < lookup t x \wedge (\forall 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 :: _
  (simp add: lex-pm-strict-def less-poly-mapping.rep-eq less-fun-def)

proof (rule ccontr)
  assume ¬ lex-pm 0 s
  hence lex-pm-strict s 0
  thus False
qed

lemma lex-pm-plus-monotone: lex-pm s t ⟹ lex-pm (s + u) (t + u)
  for s t :: _
  by (simp add: lex-pm-def add-right-mono)

6.8.8 Degree

lift-definition deg-pm::('a ⇒ '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 :: _
  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:: _
  by (simp only: deg-pm.rep-eq lookup-plus-fun, rule deg-fun-plus, simp-all add: keys-eq-sup[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
  qed

99
ultimately have \( \text{deg-pm} (\text{Poly-Mapping} \cdot \text{single} \ x \ k) = (\sum_{y \in \{x\}} \text{lookup} (\text{Poly-Mapping} \cdot \text{single} \ x \ k) \ y) \)
by \( \text{rule deg-pm-superset} \)
also have \(... = k \) by simp
finally show \( \\text{thesis} \).
qed

6.8.9 General Degree-Orders

context linorder begin

lift-definition \( \text{dord-pm} :: (('a \Rightarrow 'b::ordered-comm-monoid-add) \Rightarrow ('a \Rightarrow 'b) \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'b \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool} \)
is \( \text{dord-fun} \) by \( \text{metis local.dord-fun-def} \)

lemma \( \text{dord-pm-alt} : \text{dord-pm} \ 	ext{ord} = (\lambda x \ y. \ \text{deg-pm} \ x < \text{deg-pm} \ y \lor (\text{deg-pm} \ x = \text{deg-pm} \ y \land \text{ord} \ x \ y)) \)
by \( \text{intro ext} \) \( \text{(transfer, simp add: dord-fun-def Let-def)} \)

lemma \( \text{dord-pm-degD} : \)
assumes \( \text{dord-pm ord} \ s \ t \)
shows \( \text{deg-pm} \ s \leq \text{deg-pm} \ t \)
using \( \text{assms} \) by \( \text{simp only: dord-pm.rep-eq deg-pm.rep-eq, elim dord-fun-degD} \)

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

lemma \( \text{dord-pm-antisym} : \)
assumes \( \text{ord} \ s \ t \Longrightarrow \text{ord} \ t \ s \Longrightarrow \ s = t \) and \( \text{dord-pm ord} \ s \ t \) and \( \text{dord-pm ord} \ t \ s \)
shows \( s = t \)
using \( \text{assms} \)
proof \( \text{(simp only: dord-pm.rep-eq poly-mapping-eq-iff)} \)
assume \( \text{1: (ord} \ s \ t \Longrightarrow \text{ord} \ t \ s \Longrightarrow \text{lookup} \ s = \text{lookup} \ t) \)
assume \( \text{2: dord-fun (map-fun Abs-poly-mapping id} \circ \text{ord} \circ \text{Abs-poly-mapping) (lookup} \ s) \ (\text{lookup} \ t) \)
assume \( \text{3: dord-fun (map-fun Abs-poly-mapping id} \circ \text{ord} \circ \text{Abs-poly-mapping) (lookup} \ t) \ (\text{lookup} \ s) \)
from \( \text{- 2 3 show lookup} \ s = \text{lookup} \ t \) by \( \text{rule dord-fun-antisym, simp add: lookup-inverse 1} \)
qed

lemma \( \text{dord-pm-trans} : \)
assumes \( \text{ord} \ s \ t \Longrightarrow \text{ord} \ t \ u \Longrightarrow \text{ord} \ s \ u \) and \( \text{dord-pm ord} \ s \ t \) and \( \text{dord-pm ord} \ t \ u \)
shows \( \text{dord-pm ord} \ s \ u \)
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 ⇒ '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 ⇒ 'b::add-linorder-min
  using that by (simp only: dord-pm.rep-eq lookup-zero-fun, intro dord-fun-zero-min, simp add: keys-eq-supp)

lemma dord-pm-plus-monotone:
  fixes s t u ::'a ⇒ '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: keys-eq-supp[symmetric], keys-eq-supp[symmetric], keys-eq-supp[symmetric], keys-eq-supp[symmetric], lookup-inverse)

end

6.8.10 Degree-Lexicographic Term Order

definition dlex-pm::('a::linorder ⇒ 'b::{ordered-comm-monoid-add,linorder}) ⇒ ('a ⇒ '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 \textit{dlex-pm-antisym}: \textit{dlex-pm} \ s \ t \iff \textit{dlex-pm} \ t \ s \iff \ s = t
unfolding \textit{dlex-pm-def} using \textit{lex-pm-antisym} by (rule \textit{dord-pm-antisym})

lemma \textit{dlex-pm-trans}: \textit{dlex-pm} \ s \ t \iff \textit{dlex-pm} \ t \ u \iff \textit{dlex-pm} \ s \ u
unfolding \textit{dlex-pm-def} using \textit{lex-pm-trans} by (rule \textit{dord-pm-trans})

lemma \textit{dlex-pm-lin}: \textit{dlex-pm} \ s \ t \lor \textit{dlex-pm} \ t \ s
unfolding \textit{dlex-pm-def} using \textit{lex-pm-lin} by (rule \textit{dord-pm-lin})

corollary \textit{dlex-pm-strict-alt} [code]: \textit{dlex-pm-strict} \  s \ t = (\neg \textit{dlex-pm} \ t \ s)
unfolding \textit{dlex-pm-strict-def} using \textit{dlex-pm-lin} by auto

lemma \textit{dlex-pm-zero-min}: \textit{dlex-pm} \ 0 \ s
for \ s \ t :: (\neg \Rightarrow \textit{0} \neg :: \textit{add-linorder-min})
unfolding \textit{dlex-pm-def} using \textit{lex-pm-refl} by (rule \textit{dord-pm-zero-min})

lemma \textit{dlex-pm-plus-monotone}: \textit{dlex-pm} \ s \ t \Rightarrow \textit{dlex-pm} \ (s + u) \ (t + u)
for \ s \ t :: (- \Rightarrow \textit{0} :: \{\textit{ordered-ab-semigroup-add-imp-le}, \textit{ordered-cancel-comm-monoid-add}\})
unfolding \textit{dlex-pm-def} using \textit{lex-pm-plus-monotone} by (rule \textit{dord-pm-plus-monotone})

\subsection*{6.8.11 Degree-Reverse-Lexicographic Term Order}

definition \textit{drlex-pm} :: (\textit{a} :: \textit{linorder} \Rightarrow \textit{0} \textit{b} :: \{\textit{ordered-comm-monoid-add}, \textit{linorder}\}) \Rightarrow (\textit{a} \Rightarrow \textit{b}) \Rightarrow \textit{bool}
where \textit{drlex-pm} \equiv \textit{dord-pm} (\lambda \ s. \textit{lex-pm} \ t \ s)

definition \textit{drlex-pm-strict} \ s \ t \leftrightarrow \textit{drlex-pm} \ s \ t \land \neg \textit{drlex-pm} \ t \ s

lemma \textit{drlex-pm-refl}: \textit{drlex-pm} \ s \ s
unfolding \textit{drlex-pm-def} using \textit{lex-pm-refl} by (rule \textit{dord-pm-refl})

lemma \textit{drlex-pm-antisym}: \textit{drlex-pm} \ s \ t \iff \textit{drlex-pm} \ t \ s \iff \ s = t
unfolding \textit{drlex-pm-def} using \textit{lex-pm-antisym} by (rule \textit{dord-pm-antisym})

lemma \textit{drlex-pm-trans}: \textit{drlex-pm} \ s \ t \iff \textit{drlex-pm} \ t \ u \iff \textit{drlex-pm} \ s \ u
unfolding \textit{drlex-pm-def} using \textit{lex-pm-trans} by (rule \textit{dord-pm-trans})

lemma \textit{drlex-pm-lin}: \textit{drlex-pm} \ s \ t \lor \textit{drlex-pm} \ t \ s
unfolding \textit{drlex-pm-def} using \textit{lex-pm-lin} by (rule \textit{dord-pm-lin})

corollary \textit{drlex-pm-strict-alt} [code]: \textit{drlex-pm-strict} \ s \ t = (\neg \textit{drlex-pm} \ t \ s)
unfolding \textit{drlex-pm-strict-def} using \textit{drlex-pm-lin} by auto

lemma \textit{drlex-pm-zero-min}: \textit{drlex-pm} \ 0 \ s
for \ s \ t :: (\neg \Rightarrow \textit{0} :: \textit{add-linorder-min})
unfolding \textit{drlex-pm-def} using \textit{lex-pm-refl} by (rule \textit{dord-pm-zero-min})

lemma \textit{drlex-pm-plus-monotone}: \textit{drlex-pm} \ s \ t \Rightarrow \textit{drlex-pm} \ (s + u) \ (t + u)
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 \(p \in \text{span } B\)
  shows \(p \in \text{span } (\text{insert } r B)\)
proof
  have \(B \subseteq \text{insert } r B\) by blast
  hence \(\text{span } B \subseteq \text{span } (\text{insert } r B)\) by (rule span-mono)
with assms show \(?thesis \).
qed

lemma span-insertD:
  assumes \(p \in \text{span } (\text{insert } r B)\) and \(r \in \text{span } B\)
  shows \(p \in \text{span } B\)
using assms(1)
proof (induct p rule: span-induct-alt)
case base
  show \(0 \in \text{span } B\) by (fact span-zero)
next
  case step: (step q b a)
  from step(1) have \(b = r \lor b \in B\) by simp
  thus \(q * s b + a \in \text{span } B\)
proof
    assume eq: \(b = r\)
    from step(2) assms(2) show \(?thesis unfolding eq\) by (intro span-add span-scale)
next
    assume \(b \in B\)
    hence \(b \in \text{span } B\) using span-superset ..
    with step(2) show \(?thesis by (intro span-add span-scale)\)
lemma \textit{span-insert-idI}:  
\hspace{1em} \text{assumes } r \in \text{span } B  
\hspace{1em} \text{shows } \text{span } (\text{insert } r \ B) = \text{span } B  
\hspace{1em} \text{proof } (\text{intro subset-antisym subsetI})  
\hspace{2em} \text{fix } p  
\hspace{3em} \text{assume } p \in \text{span } (\text{insert } r \ B)  
\hspace{3em} \text{from this assms show } p \in \text{span } B \ \text{by } (\text{rule span-insertD})  
\hspace{1em} \text{next}  
\hspace{2em} \text{fix } p  
\hspace{3em} \text{assume } p \in \text{span } B  
\hspace{3em} \text{thus } p \in \text{span } (\text{insert } r \ B) \ \text{by } (\text{rule span-insertI})  
\text{qed}  

lemma \textit{span-insert-zero}:  
\text{span } (\text{insert } 0 \ B) = \text{span } B  
\text{using span-zero by } (\text{rule span-insert-idI})  

lemma \textit{span-Diff-zero}:  
\text{span } (B - \{0\}) = \text{span } B  
\text{by } (\text{metis span-insert-zero insert-Diff-single})  

lemma \textit{span-insert-subset}:  
\hspace{1em} \text{assumes } \text{span } A \subseteq \text{span } B \ \text{and } r \in \text{span } B  
\hspace{1em} \text{shows } \text{span } (\text{insert } r \ A) \subseteq \text{span } B  
\hspace{1em} \text{proof}  
\hspace{2em} \text{fix } p  
\hspace{3em} \text{assume } p \in \text{span } (\text{insert } r \ A)  
\hspace{3em} \text{thus } p \in \text{span } B \ \text{proof } (\text{induct } p \ \text{rule: span-induct-alt})  
\hspace{4em} \text{case base}  
\hspace{5em} \text{show } ?\text{case by } (\text{fact span-zero})  
\hspace{3em} \text{next}  
\hspace{4em} \text{case step: } (\text{step } q \ b \ a)  
\hspace{5em} \text{show } ?\text{case}  
\hspace{4em} \text{proof } (\text{intro span-add span-scale})  
\hspace{5em} \text{from } \langle b \in \text{insert } r \ A \rangle \ \text{show } b \in \text{span } B \ \text{proof}  
\hspace{6em} \text{assume } b = r  
\hspace{7em} \text{thus } b \in \text{span } B \ \text{using } \text{assms}(2) \ \text{by simp}  
\hspace{5em} \text{next}  
\hspace{6em} \text{assume } b \in A  
\hspace{7em} \text{hence } b \in \text{span } A \ \text{using } \text{span-superset } ..  
\hspace{7em} \text{thus } b \in \text{span } B \ \text{using } \text{assms}(1) \ ..  
\hspace{6em} \text{qed}  
\hspace{5em} \text{qed fact}  
\hspace{4em} \text{qed}  
\hspace{3em} \text{qed}  
\hspace{2em} \text{qed}  
\hspace{1em} \text{qed}
lemma replace-span:
  assumes \( q \in \text{span} B \)
  shows \( \text{span} (\text{insert} q (B - \{p\})) \subseteq \text{span} B \)
  by (rule span-insert-subset, rule span-mono, fact Diff-subset, fact)

lemma sum-in-spanI: \( (\sum b \in B. \ q b * s b) \in \text{span} B \)
  by (auto simp: intro: span-sum span-scale dest: span-base)

lemma span-closed-sum-list: \( (\forall x. x \in \text{set} \ \text{xs} \implies x \in \text{span} B) \implies \text{sum-list} \ \text{xs} \in \text{span} B \)
  by (induct \text{xs}) (auto intro: span-zero span-add)

lemma span-finiteE:
  assumes \( p \in \text{span} B \)
  obtains \( A \ q \) where finite \( A \) and \( A \subseteq B \) and \( p = (\sum a \in A. \ (q a) * s a) \)
  using assms by (auto simp: span-finite)

lemma span-finite-subset:
  assumes \( p \in \text{span} B \)
  obtains \( A \) where finite \( A \) and \( A \subseteq B \) and \( p \in \text{span} A \)
  proof
    from assms obtain \( A \ q \) where finite \( A \) and \( A \subseteq B \) and \( p = (\sum a \in A. \ q a * s a) \)
      by (rule spanE)
    note this[1, 2]
    moreover have \( p \in \text{span} A \) unfolding \text{p} by (rule sum-in-spanI)
    ultimately show ?thesis ..
  qed

lemma span-subset-spanI:
  assumes \( A \subseteq \text{span} B \)
  shows \( \text{span} A \subseteq \text{span} B \)
  using assms subspace-span by (rule span-minimal)

lemma span-insert-cong:
  assumes \( \text{span} A = \text{span} B \)
  shows \( \text{span} (\text{insert} \ p \ A) = \text{span} (\text{insert} \ p \ B) \) (is \( \?l = \?r \))
  proof
    have 1: \( \text{span} (\text{insert} \ p \ C1) \subseteq \text{span} (\text{insert} \ p \ C2) \) if \( \text{span} \ C1 = \text{span} \ C2 \) for \( C1 \) \( C2 \)
      proof (rule span-subset-spanI)
        show \( \text{insert} \ p \ C1 \subseteq \text{span} (\text{insert} \ p \ C2) \)
        proof (rule insert-subsetI)
          show \( p \in \text{span} (\text{insert} \ p \ C2) \) by (rule span-base) simp
        qed
      qed
  qed
have $C_1 \subseteq \text{span } C_1$ by (rule span-superset)
also from that have ... = \text{span } C_2 .
also have ... \subseteq \text{span } (\text{insert } p \text{ } C_2) \text{ by (rule span-mono) blast}
finally show $C_1 \subseteq \text{span } (\text{insert } p \text{ } C_2) .

qed

from assms show $?l \subseteq ?r$ by (rule 1)
from assms[symmetric] show $?r \subseteq ?l$ by (rule 1)

qed

lemma span-induct' [consumes 1, case-names base step]:
assumes $p \in \text{span } B \text{ and } P 0$
and $\bigwedge a \ q \ p. \ a \in \text{span } B \Rightarrow P a \Rightarrow p \in B \Rightarrow q \neq 0 \Rightarrow P (a + q \cdot s \ p)$
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 $b \in \text{span } B$ by (rule span-base)
hence $q \cdot s \ b \in \text{span } B$ by (rule span-scale)
with step.prems have $a \in \text{span } B$ 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 $a \in \text{span } B$ $P a$ step.hyps(1) have $P (a + q \cdot s \ b)$ by (rule assms(3))
  thus ?thesis by (simp only: add.commute)
qed

qed

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

proof
  fix $p$
  assume $p \in ?l$
  show $p \in ?r$
  proof
    fix $a$
    assume $a \in A$
    from $p \in ?l$ show $p \in \text{span } (f a)$
  proof (induct p rule: span-induct')
    case base
    show ?case by (fact span-zero)
  next

qed
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}. span 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-eql) 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 \$\textit{case} \$.

qed

next

fix \(p\)

assume \(p \in \mathcal{B}\)

then obtain \(p'\) where \(p' \in \text{span } F\) and \(p = q * s\ p'\).

from this(1) show \(p \in \mathcal{A}\)

unfolding \(p = q * s\ p'\)

proof (induct \(p'\) rule: span-induct')

  case base

  show \$\textit{case} by (simp add: span-zero)

next

  case (step \(p\ r\ a\))

  from step.hyps(3) have \(q * s\ a \in (s)\) \(q * F\) by (rule imageI)

  hence \(q * s\ a \in \mathcal{A}\) by (rule span-base)

  hence \(r * s\ (q * s\ a) \in \mathcal{A}\) by (rule span-scale)

  with step.hyps(2) have \(q * s\ p + r * s\ (q * s\ a) \in \mathcal{A}\)

  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 \$\textit{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 \(B\) = \(\text{UNIV}\) \(\iff\) \(1 \in \text{ideal } B\)

proof

  assume \(\ast\): \(1 \in \text{ideal } B\)

  show \(\text{ideal } B = \text{UNIV}\)

  proof

    fix \(x\)

    from \(\ast\) have \(x * 1 \in \text{ideal } B\)

     by (rule ideal.span-scale)

    thus \(x \in \text{ideal } B\)

     by simp

  qed

  qed simp

  qed simp
lemma ideal-eq-zero-iff \[\text{iff} \]: \(\text{ideal} \ F = \{0\} \leftrightarrow F \subseteq \{0\}\)
by (metis empty-subsetI ideal.span-empty ideal.span-eq)

lemma ideal-field-cases:
  obtains \(\text{ideal} \ B = \{0\} \mid \text{ideal} \ (B :: \text{a::field set}) = \text{UNIV}\)
proof (cases ideal B = \{0\})
  case True
  thus \(?thesis \).
next
  case False
  hence \(\neg B \subseteq \{0\}\) by simp
  then obtain \(b\) where \(b \in B\) and \(b \neq 0\) by blast
  from this(1) have \(b \in \text{ideal} \ B\) by (rule ideal.span-base)
  hence \(\text{inverse} \ b \ast b \in \text{ideal} \ B\) by (rule ideal.span-scale)
  with \(b \neq 0\) have \(\text{ideal} \ B = \text{UNIV}\) by (simp add: ideal-eq-UNIV-iff-contains-one)
  thus \(?thesis \).
qed

corollary ideal-field-disj: \(\text{ideal} \ B = \{0\} \lor \text{ideal} \ (B :: \text{a::field set}) = \text{UNIV}\)
by (rule ideal-field-cases) blast+

lemma image-ideal-subset:
assumes \(\forall x \ y. \ h \ (x + y) = h \ x + h \ y\) and \(\forall x \ y. \ h \ (x \ast y) = h \ x \ast h \ y\)
shows \(h \cdot \text{ideal} \ F \subseteq \text{ideal} \ (h \cdot F)\)
proof (intro subsetI, elim imageE)
fix \(g f\)
assume \(g: g = h f\)
assume \(f \in \text{ideal} \ F\)
thus \(g \in \text{ideal} \ (h \cdot F)\) unfolding g
proof (induct f rule: ideal.span-induct-alt)
  case base
  have \(h \ 0 = h \ (0 + 0)\) by simp
  also have \(\ldots = h \ 0 + h \ 0\) 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 \(h \ f \in \text{ideal} \ (h \cdot F)\)
  by (intro ideal.span-base imageI)
  hence \(h \ c \ast h \ f \in \text{ideal} \ (h \cdot F)\) by (rule ideal.span-scale)
  hence \(h \ c \ast h \ f + h \ g \in \text{ideal} \ (h \cdot F)\)
  using step.hyps(2) by (rule ideal.span-add)
  thus \(?case\) by (simp only: assms)
qed
qed

lemma image-ideal-eq-surj:
assumes \(\forall x \ y. \ h \ (x + y) = h \ x + h \ y\) and \(\forall x \ y. \ h \ (x \ast y) = h \ x \ast h \ y\) and
\(\text{surj} \ h\)
shows $h' \text{id} \ B = \text{id} (h' B)$

proof
from assms(1, 2) show $h' \text{id} B \subseteq \text{id} (h' B)$ by (rule image-ideal-subset)
next
show $\text{id} (h' B) \subseteq h' \text{id} B$
proof
fix $b$
assume $b \in \text{id} (h' B)$
thus $b \in h' \text{id} B$
proof (induct $b$ rule: ideal.span-induct-alt)
case base
have $0 = h (0 + 0)$ by (simp only: assms(1))
finally have $0 = h 0$ by (rule rev-image-eqI)
next
case (step $c \ a b$)
from assms(3) obtain $c'$ where $c = h c'$ by (rule surjE)
from step.hyps(2) obtain $a'$ where $a' \in \text{id} B$ and $a = h a'$. ..
from step.hyps(1) obtain $b'$ where $b' \in B$ and $b = h b'$. ..
from this(1) have $b' \in \text{id} B$ by (rule ideal.span-base)
hence $c' * b' \in \text{id} B$ by (rule ideal.span-scale)
hence $c' * b' + a' \in \text{id} B$ using ($a' \in -$) 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

context
fixes $h : 'a \Rightarrow 'a::comm-ring-1$
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 $B \subseteq \text{range} \ h$ and $p \in \text{range} \ h$ and $p \in \text{id} B$
obtains $q$ where $\bigwedge b. q b \in \text{range} \ h$ and $p = (\sum b \in B. q b \ast b)$
proof
from assms(1, 4) obtain $q0$ where $p = (\sum b \in B. q0 b \ast b)$ by (rule ideal.span-finiteE)
define $q$ where $q = (\lambda b. h (q0 b))$
show ?thesis
proof
fix $b$
show $q b \in \text{range} \ h$ unfolding $q$-def by (rule rangeI)
next

110
from assms(3) obtain \( p' \) where \( p = h \cdot p' \).

hence \( p = h \cdot p \) by (simp only: h-idem)
also from \( \text{finite } B \) have \( \ldots = (\sum b \in B. \ q \cdot h \cdot b) \) unfolding \( p \)

proof (induct \( B \))
  case empty
  have \( h \cdot 0 = h \cdot (0 + 0) \) by simp
  also have \( \ldots = h \cdot 0 + h \cdot 0 \) by (simp only: h-plus)
  finally show \( ?\text{case} \) by simp

next
  case (insert \( b \) \( B \))
  thus \( ?\text{case} \) by (simp add: h-plus h-times q-def)

qed

also from \( \text{refl} \) have \( \ldots = (\sum b \in B. \ q \cdot b \cdot b) \)

proof (rule sum.cong)
fix \( b \)
assume \( b \in B \)
  hence \( b \in \text{range } h \) using assms(2) ..
  then obtain \( b' \) where \( b = h \cdot b' \) ..
  thus \( q \cdot b \cdot h = q \cdot b \cdot b \) by (simp only: h-idem)

qed

finally show \( p = (\sum b \in B. \ q \cdot b \cdot b) \).

qed

corollary \text{in-idealE-homomorphism}:
  assumes \( B \subseteq \text{range } h \) and \( p \in \text{range } h \) and \( p \in \text{ideal } B \)
  obtains \( A \) \( q \) where \( \text{finite } A \) and \( A \subseteq B \) and \( \forall b. \ q \cdot b \in \text{range } h \) and \( p = (\sum b \in A. \ q \cdot b \cdot b) \)

proof –
  from assms(3) obtain \( A \) \( q \) where \( \text{finite } A \) and \( A \subseteq B \) and \( p = (\sum b \in A. \ q \cdot b \cdot b) \)
  using assms(2) \( \forall p \in \text{ideal } A \) by (rule in-idealE-homomorphism-finite) blast

qed

lemma \text{ideal-induct-homomorphism} [consumes 3, case-names 0 plus]:
  assumes \( B \subseteq \text{range } h \) and \( p \in \text{range } h \) and \( p \in \text{ideal } B \)
  assumes \( P \ 0 \) and \( \forall c \ a. \ c \in \text{range } h \Rightarrow b \in B \Rightarrow P \ a \Rightarrow a \in \text{range } h \Rightarrow P \ (c \cdot b + a) \)
  shows \( P \ p \)

proof –
  from assms(1–3) obtain \( A \) \( q \) where \( \text{finite } A \) and \( A \subseteq B \) and \( rl: \forall f. \ q \cdot f \in \text{range } h \)
  and \( p = (\sum f \in A. \ q \cdot f \cdot f) \) by (rule in-idealE-homomorphism) blast
  show \( ?\text{thesis} \) unfolding \( p \) using \( \text{finite } A \) \( \forall A \subseteq B \)
  proof (induct \( A \))
    case empty
    
    111
from assms(4) show ?case by simp
next
case (insert a A)
from insert.hyps(1, 2) have \((\sum f \in \text{insert } a \ A \cdot q f \cdot f) = q a \cdot a + (\sum f \in A \cdot q f \cdot f)\) by simp
also from rl have P . .
proof (rule assms(5))
have a \in \text{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 \cdot q f \cdot f) by (rule insert.hyps)
next
from insert.prems have A \subseteq B by simp
hence A \subseteq \text{range } h using assms(1) by (rule subset-trans)
with (finite A) show (\sum f \in A \cdot q f \cdot f) \in \text{range } h
proof (induct A)
case empty
have h 0 = h (0 \cdot 0) by simp
also have . . = h 0 + h 0 by (simp only: h-plus)
finally have (\sum f \in \emptyset \cdot q f \cdot f) = h 0 by simp
thus ?case by (rule rangeI) simp
next
case (insert a A)
from insert.prems have a \in \text{range } h and A \subseteq \text{range } h by simp-all
from this(1) obtain a' where a: a = h a' ..
from (q a \in \text{range } h) obtain q' where q: q a = h q' ..
from (A \subseteq -) have (\sum f \in A \cdot q f \cdot f) \in \text{range } h by (rule insert.hyps)
then obtain m where eq: (\sum f \in A \cdot q f \cdot f) = h m ..
from insert.hyps(1, 2) have (\sum f \in \text{insert } a \ A \cdot q f \cdot f) = q a \cdot a + (\sum f \in A \cdot q f \cdot f) by simp
also have . . = h (q' \cdot a' + m) unfolding q by (simp add: a eq h-plus h-times)
also have . . \in \text{range } h by (rule rangeI)
finally show ?case .
qed
qed

lemma image-ideal-eq-Int: h' \text{ ideal } B = \text{ ideal } (h' \cdot B) \cap \text{ range } h
proof
from h-plus h-times have h' \text{ ideal } B \subseteq \text{ ideal } (h' \cdot B) by (rule image-ideal-subset)
thus h' \text{ ideal } B \subseteq \text{ ideal } (h' \cdot B) \cap \text{ range } h by blast
next
show ideal (h' \cdot B) \cap \text{ range } h \subseteq h' \text{ ideal } B
proof
fix b

112
assume $b \in \text{ideal}(h \cdot B) \cap \text{range } h$

hence $b \in \text{ideal}(h \cdot B)$ and $b \in \text{range } h$ by simp-all

have $h \cdot B \subseteq \text{range } h$ by blast

thus $b \in h \cdot \text{ideal } B$ using $(b \in \text{range } h) \cdot (b \in \text{ideal}(h \cdot B))$

proof (induct $b$ rule: ideal-induct-homomorphism)

  case 0
  have $h \cdot 0 = h \cdot (0 + 0)$ by simp
  also have $\ldots = h \cdot 0 + h \cdot 0$ by (simp only: h-plus)
  finally have $0 = h \cdot 0$ by simp

with ideal.span-zero show ?case by (rule rev-image-eqI)

next

  case (plus $c \cdot b \cdot a$)
  from plus.hyps(1) obtain $c'$ where $c = h \cdot c'$ ..
  from plus.hyps(3) obtain $a'$ where $a' \in \text{ideal } B$ and $a = h \cdot a'$ ..
  from plus.hyps(2) obtain $b'$ where $b' \in B$ and $b = h \cdot b'$ ..
  from this(1) have $b' \in \text{ideal } B$ by (rule ideal.span-base)
  hence $c' \cdot b' \in \text{ideal } B$ by (rule ideal.span-scale)
  hence $c' \cdot b' + a' \in \text{ideal } B$ using $(a' \in -)$ by (rule ideal.span-add)

  moreover have $c \cdot b + a = h \cdot (c' \cdot b' + a')$ by (simp add: $a \cdot b \cdot c$ h-plus h-times)
  ultimately show ?case by (rule rev-image-eqI)

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: $f \cdot (a \text{ when } b) = (f \cdot a \text{ when } b)$ if $\neg b \Rightarrow f \cdot 0 = 0$

using that by (auto simp: when-def)

definition mapp-2 :: $('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd) \Rightarrow ('a \Rightarrow_0 'b::zero) \Rightarrow ('a \Rightarrow_0 'c::zero)$

where mapp-2 $f \cdot p \cdot q = \text{Abs-poly-mapping } (\lambda k. f \cdot k \cdot (\text{lookup } p \cdot k) \cdot (\text{lookup } q \cdot k))$ when $k \in \text{keys } p \cup \text{keys } q$

lemma lookup-mapp-2:
lookup \((\text{mapp-2}\ f\ p\ q)\) \(k\) \(=\) \((f\ k\ (\text{lookup}\ p\ k)\ (\text{lookup}\ q\ k)\)\) when \(k\) \(\in\) \(\text{keys}\ p\ \cup\ \text{keys}\ q\)

**proof**

- **have** lookup \((\text{Abs-poly-mapping}\ (\lambda k.\ f k\ (\text{lookup}\ p\ k)\ (\text{lookup}\ q\ k))\)\) when \(k\) \(\in\) \(\text{keys}\ p\ \cup\ \text{keys}\ q\) \(=\)
  \((\lambda k.\ f k\ (\text{lookup}\ p\ k)\ (\text{lookup}\ q\ k))\) when \(k\) \(\in\) \(\text{keys}\ p\ \cup\ \text{keys}\ q\)
  - **by** \((\text{rule}\ \text{Abs-poly-mapping-inverse},\ \text{simp})\)
- **thus** ?thesis **by** \((\text{simp}\ \text{add}:\ \text{mapp-2-def})\)

**qed**

**lemma** lookup-mapp-2-homogenous:

- **assumes** \(f\ k\ 0\ 0\ =\ 0\)
- **shows** lookup \((\text{mapp-2}\ f\ p\ q)\) \(k\) \(=\) \(f\ k\ (\text{lookup}\ p\ k)\ (\text{lookup}\ q\ k)\)
  - **by** \((\text{simp}\ \text{add}:\ \text{lookup-mapp-2}\ \text{when-def}\ \text{in-keys-iff}\ \text{assms})\)

**lemma** mapp-2-cong \([\text{fundef-cong}]\):

- **assumes** \(p\ \equiv\ p'\)\ and \(q\ \equiv\ q'\)
- **assumes** \(\forall k.\ k\ \in\ \text{keys}\ p'\ \cup\ \text{keys}\ q'\ \Rightarrow\ f k\ (\text{lookup}\ p' k)\ (\text{lookup}\ q' k)\ =\ f' k\ (\text{lookup}\ p' k)\ (\text{lookup}\ q' k)\)
- **shows** \(\text{mapp-2}\ f\ p\ q\ =\ \text{mapp-2}\ f'\ p'\ q'\)
  - **by** \((\text{rule}\ \text{poly-mapping-eqI},\ \text{simp}\ \text{add}:\ \text{assms}(1,\ 2)\ \text{lookup-mapp-2},\ \text{rule}\ \text{when-cong},\ \text{fact}\ \text{refl},\ \text{rule}\ \text{assms}(3),\ \text{blast})\)

**lemma** keys-mapp-subset:

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

**lemma** mapp-2-mapp:

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

### 9.1 keys

**lemma** in-keys-plusI1:

- **assumes** \(t\ \in\ \text{keys}\ p\)\ and \(t\ \notin\ \text{keys}\ q\)
- **shows** \(t\ \in\ \text{keys}\ (p\ +\ q)\)
  - **using** \text{assms} \text{unfolding} \text{in-keys-iff} \text{lookup-add} **by** \text{simp}

**lemma** in-keys-plusI2:

- **assumes** \(t\ \in\ \text{keys}\ q\)\ and \(t\ \notin\ \text{keys}\ p\)
- **shows** \(t\ \in\ \text{keys}\ (p\ +\ q)\)
  - **using** \text{assms} \text{unfolding} \text{in-keys-iff} \text{lookup-add} **by** \text{simp}

**lemma** keys-plus-eqI:

- **assumes** \(\text{keys}\ p\ \cap\ \text{keys}\ q\ =\ \{\}\)
- **shows** \(\text{keys}\ (p\ +\ q)\ =\ (\text{keys}\ p\ \cup\ \text{keys}\ q)\)
proof
  
  show \( (p + q) \subseteq keys p \cup keys q \)
  by (simp add: Poly-Mapping.keys-add)

  show \( keys p \cup keys q \subseteq (p + q) \)
  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 \( \equiv (\lambda c \cdot Poly-Mapping.single t c) \)

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 \( (c = 0 \land d = 0) \lor (c = d \land s = t) \)
  using assms unfolding poly-mapping-eq-iff
  by (metis mono-tags, hide-lams) lookup-single-eq lookup-single-not-eq

definition is-monomial :: ('a::zero) \( \Rightarrow \) bool
  where \( is-monomial p \iff \text{card} (keys p) = 1 \)

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 \( \text{card} (keys p) = 1 \) unfolding is-monomial-def
    then obtain \( t \) where \( sp: keys p = \{ t \} \) by (rule card-1-singletonE)
    let \( \?c = \text{lookup} p t \)
    from \( sp \) have \( \?c \neq 0 \) by fastforce
    show \( ?thesis \)
proof
  show \( p = \text{monomial} \ ?c \ t \)
proof (intro poly-mapping-keys-eqI)
  from sp show \( \text{keys} \ p = \text{keys} \ (\text{monomial} \ ?c \ t) \) using \(?c \neq 0\) by simp
next
  fix \( s \)
  assume \( s \in \text{keys} \ p \)
  with sp have \( s = t \) by simp
  show \( \text{lookup} \ p \ s = \text{lookup} \ (\text{monomial} \ ?c \ t) \ s \) by (simp add: \(?s = t\))
  qed
  qed fact
qed

lemma \( \text{is-monomial-uminus}: \text{is-monomial} \ (-p) \longleftrightarrow \text{is-monomial} \ p \)
unfolding \( \text{is-monomial-def \ keys-uminus} \).

lemma \( \text{monomial-not-0}: \)
  assumes \( \text{is-monomial} \ p \)
  shows \( p \neq 0 \)
using assms unfolding \( \text{is-monomial-def} \) by auto

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

lemma \( \text{monomial-0I}: \)
  assumes \( c = 0 \)
  shows \( \text{monomial} \ c \ t = 0 \)
using assms by transfer (auto)

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

corollary \( \text{monomial-0-iff}: \text{monomial} \ c \ t = 0 \longleftrightarrow c = 0 \)
bv (rule, erule monomial-0D, erule monomial-0I)

lemma \( \text{lookup-times-monomial-left}: \text{lookup} \ (\text{monomial} \ c \ t * p) \ s = (c * \text{lookup} \ p \ (s - t)) \) 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 \ ) 'b and \( w \)
assume \( p \neq 0 \) and \( w \in \text{keys } p \)
and \( \text{IH}: \text{lookup} (\text{monomial } c \times t \times \text{except } p \{w\}) s =
\{(c \times \text{lookup} (\text{except } p \{w\}) (s - t)) \text{ when } t \text{ adds } s\} (\text{is } - = ?x)\)

have \( \text{monomial } c \times t \times p = \text{monomial } c \times t \times \{\text{monomial } (\text{lookup } p \ w \times \text{except } p \{w\})\}\)
by \((\text{simp only: plus-except\{symmetric\}})\)
also have \( ... = \text{monomial } c \times t \times \text{monomial } (\text{lookup } p \ w) \times \text{monomial } c \times t \times \text{except } p \{w\}\)
by \((\text{simp add: algebra-simps})\)
also have \( ... = \text{monomial } (c \times \text{lookup } p \ w) \times (t + w) \times \text{monomial } c \times t \times \text{except } p \{w\}\)
by \((\text{simp only: mult-single})\)
finally have \( \text{lookup} (\text{monomial } c \times t \times p) s = \text{lookup} (\text{monomial } (c \times \text{lookup } p \ w) \times (t + w)) s + ?x\)
by \((\text{simp only: lookup-add IH})\)
also have \( ... = (\text{lookup} (\text{monomial } (c \times \text{lookup } p \ w) \times (t + w)) s + \text{c \times lookup } \text{except } p \{w\}) (s - t) \text{ when } t \text{ adds } s\)\)
by \((\text{rule when-distrib, auto simp add: lookup-single when-def})\)
also from \( \text{refl} \) have \( ... = (c \times \text{lookup } p \ (s - t) \text{ when } t \text{ adds } s)\)
proof \((\text{rule when-cong})\)
assume \( t \text{ adds } s \)
then obtain \( u \) where \( u; s = t + u \) ...
show \( \text{lookup} (\text{monomial } (c \times \text{lookup } p \ w) \times (t + w)) s + c \times \text{lookup} (\text{except } p \{w\}) (s - t) =
\text{c \times lookup } p \ (s - t)\)
by \((\text{simp add: u, cases u = w, simp-all add: lookup-except lookup-single add.commute})\)
qed
finally show \( \text{lookup} (\text{monomial } c \times t \times p) s = (c \times \text{lookup } p \ (s - t) \text{ when } t \text{ adds } s)\) .
qed

lemma \( \text{lookup-times-monomial-right}: \text{lookup} (p \times \text{monomial } c \times t) s = (\text{lookup } p \ (s - t) \times c \text{ when } t \text{ adds } s)\)
for \( c::\text{semiring-0} \) and \( t::\text{comm-powerprod}\)
proof \((\text{induct } p \text{ rule: poly-mapping-except-induct, simp})\)
fix \( p::'a \Rightarrow _0 'b \) and \( w \)
assume \( p \neq 0 \) and \( w \in \text{keys } p \)
and \( \text{IH}: \text{lookup} (\text{except } p \{w\} \times \text{monomial } c \times t) s =
(((\text{lookup } \text{except } p \{w\}) (s - t)) \times c \text{ when } t \text{ adds } s)\)
by \((\text{simp only: plus-except\{symmetric\}})\)
also have \( p \times \text{monomial } c \times t \times (\text{monomial } (\text{lookup } p \ w \times \text{except } p \{w\}) \times \text{monomial } c \times t)\)
by \((\text{simp add: algebra-simps})\)
also have \( ... = \text{monomial } (\text{lookup } p \ w \times c) \times (w + t) \times \text{except } p \{w\} \times \text{monomial } c \times t\)
by (simp only: mult-single)

finally have \( \text{lookup} \ (p \ast \text{monomial} \ c \ t) \ s = \text{lookup} \ (\text{monomial} \ \text{lookup} \ p \ w \ c) \) 
\((w + t)) \ s + ?x \)
by (simp only: lookup-add IH)
also have \( ... = (\text{lookup} \ (\text{monomial} \ \text{lookup} \ p \ w \ c) \ (w + t)) \ s + \)
\(\text{lookup} \ (\text{except} \ p \ \{w\}) \ (s - t) \ast c \ \text{when} \ t \ \text{adds} \ s)\)
by (rule when-distrib, auto simp add: lookup-single when-def)
also from refl have \( ... = (\text{lookup} \ p \ (s - t) \ast c \ \text{when} \ t \ \text{adds} \ s)\)
proof (rule when-cong)
assume \( t \ \text{adds} \ s \)
then obtain \( u \) where \( u: s = t + u \) ..
show \( \text{lookup} \ (\text{monomial} \ \text{lookup} \ p \ w \ c) \ (w + t)) \ s + \text{lookup} \ (\text{except} \ p \ \{w\}) \)
\((s - t) \ast c = \)
\(\text{lookup} \ p \ (s - t) \ast c \)
by (simp add: \( u \), cases \( u = w \), simp-all add: lookup-except lookup-single add.commute)
qed
finally show \( \text{lookup} \ (p \ast \text{monomial} \ c \ t) \ s = (\text{lookup} \ p \ (s - t) \ast c \ \text{when} \ t \ \text{adds} \ s) \).
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 \( 'a \times \text{unit} \).

named-theorems term-simps simplification rules for terms

locale term-powerprod =
fixes pair-of-term::\( 't \Rightarrow ('a::\text{comm-powerprod} \times 'k::\text{linorder}) \)
fixes term-of-pair::\(('a \times 'k) \Rightarrow 't \)
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 \( u = \text{pair-of-term} \ v \)
shows \( u = v \)
proof -
from \( \text{assms} \) have term-of-pair (pair-of-term \( u \)) = term-of-pair (pair-of-term \( v \))
by (simp only:)
thus \( \text{thesis} \) 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 :: ′a ⇒ ′t
where pp-of-term v = fst (pair-of-term v)
definition component-of-term :: ′a ⇒ ′t
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 ⇒ ′t ⇒ ′t (infixl ⊕ 75)
where splus t v = term-of-pair (t + pp-of-term v, component-of-term v)
definition sminus :: ′a ⇒ ′t ⇒ ′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 ⇒ ′t ⇒ bool (infix adds_p 50)
where adds-pp t v ←→ t adds pp-of-term v
definition adds-term :: ′t ⇒ ′t ⇒ 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 \((t \oplus v) = t + pp-of-term\ v\)
  by (simp add: splus-def term-simps)

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

lemma pp-of-term-sminus [term-simps]: pp-of-term \((v \ominus t) = pp-of-term\ v - t\)
  by (simp add: sminus-def term-simps)

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

lemma splus-sminus [term-simps]: \((t \oplus v) \ominus t = v\)
  by (simp add: sminus-def term-simps)

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

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

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

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

lemma splus-right-canc [term-simps]: \(t \oplus v = s \oplus v \iff t = s\)
  by (metis add-right-cancel pp-of-term-splus)

lemma splus-left-canc [term-simps]: \(t \oplus v = t \oplus u \iff v = u\)
  by (metis splus-sminus)

lemma adds-ppI [intro?]:
  assumes \(v = t \oplus u\)
  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 \(v = t \oplus u\)
  proof
    from assms obtain \(s\) where \(*: pp-of-term\ v = t + s\ unfolding\ adds-pp-def..\)
    have \(v = t \oplus (\text{term-of-pair}\ (s, \text{component-of-term}\ v))\)
      by (simp add: splus-def term-simps, metis * add.commute term-of-pair-pair)
    thus \(?thesis..\)
qed

lemma adds-pp-alt: \( t \text{ adds}_p v \iff (\exists u. v = t \oplus u) \)
by (meson adds-ppE adds-ppI)

lemma adds-pp-refl [term-simps]: \( \text{pp-of-term v} \text{ adds}_p v \)
by (simp add: adds-pp-def)

lemma adds-pp-trans [trans]:
assumes \( s \text{ adds}_p t \) and \( t \text{ adds}_p v \)
shows \( s \text{ adds}_p v \)
proof –
  note assms(1)
  also from assms(2) have \( t \text{ adds pp-of-term v} \) by (simp only: adds-pp-def)
  finally show ?thesis by (simp only: adds-pp-def)
qed

lemma zero-adds-pp [term-simps]: \( 0 \text{ adds}_p v \)
by (simp add: adds-pp-def)

lemma adds-pp-splus:
assumes \( t \text{ adds}_p v \)
shows \( t \text{ adds}_p s \oplus v \)
using assms by (simp add: adds-pp-def term-simps)

lemma adds-pp-triv [term-simps]: \( t \text{ adds}_p t \oplus v \)
by (simp add: adds-pp-def term-simps)

lemma plus-adds-pp-mono:
assumes \( s + t \text{ adds}_p v \)
and \( u \text{ adds}_p v \)
shows \( s + u \text{ adds}_p t \oplus v \)
using assms by (simp add: adds-pp-def term-simps) (rule plus-adds-mono)

lemma plus-adds-pp-left:
assumes \( s + t \text{ adds}_p v \)
shows \( s \text{ adds}_p v \)
using assms by (simp add: adds-pp-def plus-adds-left)

lemma plus-adds-pp-right:
assumes \( s + t \text{ adds}_p v \)
shows \( t \text{ adds}_p v \)
using assms by (simp add: adds-pp-def plus-adds-right)

lemma adds-pp-sminus:
assumes \( t \text{ adds}_p v \)
shows \( t \oplus (v \ominus t) = v \)
proof –
  from assms adds-pp-alt[of v t] obtain \( u \) where \( u: v = t \oplus u \) by (auto simp:
hence $v \ominus t = u$ by (simp add: term-simps)
thus ?thesis using $u$ by simp
qed

lemma adds-pp-canc: $t + s$ adds$_p$ $(t \oplus v) \iff s$ adds$_p$ $v$
  by (simp add: adds-pp-def adds-canc-2 term-simps)

lemma adds-pp-canc-2: $s + t$ adds$_p$ $(t \oplus v) \iff s$ 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$_p$ $(v \odot t)$
  using assms by (simp add: adds-pp-def term-simps) (rule plus-adds-0)

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

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

lemma plus-adds-pp: $(s + t)$ adds$_p$ $v \iff (t$ adds$_p$ $v$ $\land s$ adds$_p$ $(v \odot t))$
  by (simp add: adds-pp-def plus-adds term-simps)

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

lemma minus-splus-sminus:
  assumes $s$ adds $t$ and $u$ adds$_p$ $v$
  shows $(t - s)$ $\ominus$ $(v \ominus u) = (t \oplus v) \ominus (s + u)$
  using assms minus-plus-minus term-powerprod.adds-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 $(t - s)$ $\ominus$ $(v \odot t) = v \ominus s$
  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 $v$ $\ominus$ $(s + t) = (v \ominus s) \odot t$
by (simp add: diff-diff-add sminus-def term-simps)

lemma adds-termI [intro?]:
assumes \( v = t \oplus u \)
shows \( u \text{ adds}_t v \)
by (simp add: adds-term-def assms splus-def term-simps)

lemma adds-termE [elim?]:
assumes \( u \text{ adds}_t v \)
obtains \( t \) where \( v = t \oplus u \)
proof -
  from assms have eq: \( \text{component-of-term } u = \text{component-of-term } v \) and \( \text{pp-of-term } u \text{ adds } \text{pp-of-term } v \)
    by (simp-all add: adds-term-def)
  from this(2) obtain \( s \) where \( *: s + \text{pp-of-term } u = \text{pp-of-term } v \) unfolding adds-term-def
    using adds-minus by blast
  have \( v = s \oplus v \) by (simp add: splus-def eq * term-simps)
  thus ?thesis ..
qed

lemma adds-term-alt: \( u \text{ adds}_t v \) \( \iff \) \( \exists t \). \( v = t \oplus u \)
by (meson adds-termE adds-termI)

lemma adds-term-refl [term-simps]: \( v \text{ adds}_t v \)
by (simp add: adds-term-def)

lemma adds-term-trans [trans]:
assumes \( u \text{ adds}_t v \) and \( v \text{ adds}_t w \)
shows \( u \text{ adds}_t w \)
using assms unfolding adds-term-def using adds-trans by auto

lemma adds-term-splus:
assumes \( u \text{ adds}_t v \)
shows \( u \text{ adds}_t s \oplus v \)
using assms by (simp add: adds-term-def term-simps)

lemma adds-term-triv [term-simps]: \( v \text{ adds}_t t \oplus v \)
by (simp add: adds-term-def term-simps)

lemma splus-adds-term-mono:
assumes \( s \text{ adds } t \)
  and \( u \text{ adds}_t v \)
shows \( s \oplus u \text{ adds}_t t \oplus v \)
using assms by (auto simp: adds-term-def term-simps intro: plus-adds-mono)

lemma splus-adds-term:
assumes \( t \oplus u \text{ adds}_t v \)
shows \( u \text{ adds}_t v \)
using assms by (auto simp add: adds-term-def term-simps elim: plus-adds-right)

lemma adds-term-adds-pp:
u adds v ←→ (component-of-term u = component-of-term v ∨ pp-of-term u adds v)
  by (simp add: adds-term-def adds-pp-def)

lemma adds-term-canc: t ⊕ u adds v ←→ u adds t
  by (simp add: adds-term-def adds-canc-2 term-simps)

lemma adds-term-canc-2: s ⊕ v adds t ⊕ v ←→ s adds t
  by (simp add: adds-term-def adds-canc term-simps)

lemma splus-adds-term-0:
  assumes t ⊕ u adds t and u adds t (v ⊖ t)
  shows u adds t (v ⊖ t)
  using assms by (simp add: adds-term-def adds-pp-def plus-adds-2)

lemma splus-adds-termI-1:
  assumes t adds p v and u adds t (v ⊖ t)
  shows t ⊕ u adds t v

lemma splus-adds-term-iff:
  t ⊕ u adds t v ←→ (t adds p v ∨ u adds t (v ⊖ 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 ⇒ (′t ⇒ 0 ′b) ⇒ (′a ⇒ 0 ′b::zero)
is λk 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)

124
ultimately show finite \{ t. p \text{ (term-of-pair} (t, k)) \neq 0 \} by (rule finite-subset)

qed

definition vectorize-poly :: \( t \Rightarrow 0 \Rightarrow (\lambda t. k \Rightarrow 0 (\lambda h. b::zero)) \)

where vectorize-poly p = Abs-poly-mapping (\lambda k. proj-poly k p)

definition atomize-poly :: \( k \Rightarrow 0 \Rightarrow (\lambda a \Rightarrow 0 \Rightarrow b::zero) \)

where atomize-poly p = Abs-poly-mapping (\lambda v. lookup (lookup p (component-of-term v)) (pp-of-term v))

lemma lookup-proj-poly: lookup (proj-poly k p) t = lookup p (term-of-pair (t, k))

by (transfer, simp)

lemma lookup-vectorize-poly: lookup (vectorize-poly p) k = proj-poly k p

proof
  -
  have lookup (Abs-poly-mapping (\lambda k. proj-poly k p)) = (\lambda k. proj-poly k p)
  proof (rule Abs-poly-mapping-inverse, simp)
  have \{ k. proj-poly k p \neq 0 \} \subseteq component-of-term ' keys p
  proof (rule, simp)
  fix k
  assume proj-poly k p \neq 0
  hence keys (proj-poly k p) \neq \{ \} using poly-mapping-eq-zeroI by blast
  then obtain t where lookup (proj-poly k p) t \neq 0 by blast
  hence term-of-pair (t, k) \in keys p by (simp add: lookup-proj-poly in-keys-iff)
  hence component-of-term (term-of-pair (t, k)) \in component-of-term ' keys p
  by fastforce
  thus k \in 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 \neq 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 (\lambda v. lookup (lookup p (component-of-term v))) (pp-of-term v)) =
    (\lambda 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 \in keys (lookup p (component-of-term v)) \} \subseteq \bigcup k \in keys p. (\lambda t. term-of-pair (t, k)) ' keys (lookup p k) (is - \subseteq \{A\})
  proof (rule, simp)
  fix v
  assume \*: pp-of-term v \in keys (lookup p (component-of-term v))
  hence keys (lookup p (component-of-term v)) \neq \{ \} by blast

125
hence \text{lookup} p \ (\text{component-of-term} \ v) \neq 0 \ \text{by auto}

hence \text{component-of-term} \ v \in \text{keys} \ p \ (\text{is} \ k \in -)

by \ (\text{simp add: in-keys-iff})

thus \exists k \in \text{keys} \ p. \ v \in (\lambda t. \text{term-of-pair} \ (t, k)) \ \text{by} \ (\text{simp add: term-simps})

proof

have \ v = \text{term-of-pair} \ (\text{pp-of-term} \ v, \text{component-of-term} \ v) \ \text{by} \ (\text{simp add: term-simps})

from this \ast \ show \ v \in (\lambda t. \text{term-of-pair} \ (t, k)) \ \text{by} \ (\text{simp add: in-keys-iff})

qed

moreover have \text{finite} \ ?A \ \text{by} \ (\text{rule, fact finite-keys, rule finite-imageI, rule finite-keys})

ultimately show \text{finite} \ \{x. \text{lookup} \ (\text{lookup} p \ (\text{component-of-term} \ x)) \ (\text{pp-of-term} \ x) \neq 0\}

by \ (\text{simp add: finite-subset in-keys-iff})

qed

thus \thesis \ \text{by} \ (\text{simp add: atomize-poly-def})

qed

lemma \text{keys-proj-poly}: \text{keys} \ (\text{proj-poly} \ k \ p) = \text{pp-of-term} \ \{x \in \text{keys} \ p. \ \text{component-of-term} x = k\}

proof

show \text{keys} \ (\text{proj-poly} \ k \ p) \subseteq \text{pp-of-term} \ \{x \in \text{keys} \ p. \ \text{component-of-term} x = k\}

proof

fix \ t

assume \ t \in \text{keys} \ (\text{proj-poly} \ k \ p)

hence \text{lookup} \ (\text{proj-poly} \ k \ p) \ t \neq 0 \ \text{by} \ (\text{simp add: in-keys-iff})

hence \text{term-of-pair} \ (t, k) \in \text{keys} \ p \ \text{by} \ (\text{simp add: in-keys-iff lookup-proj-poly})

hence \text{term-of-pair} \ (t, k) \in \{x \in \text{keys} \ p. \ \text{component-of-term} x = k\} \ \text{by} \ (\text{simp add: term-simps})

hence \text{pp-of-term} \ (\text{term-of-pair} (t, k)) \in \text{pp-of-term} \ \{x \in \text{keys} \ p. \ \text{component-of-term} x = k\} \ \text{by} \ (\text{rule imageI})

thus \ t \in \text{pp-of-term} \ \{x \in \text{keys} \ p. \ \text{component-of-term} x = k\} \ \text{by} \ (\text{simp only: pp-of-term-of-pair})

qed

next

show \text{pp-of-term} \ \{x \in \text{keys} \ p. \ \text{component-of-term} x = k\} \subseteq \text{keys} \ (\text{proj-poly} \ k \ p)

proof

fix \ t

assume \ t \in \text{pp-of-term} \ \{x \in \text{keys} \ p. \ \text{component-of-term} x = k\}

then obtain \ x \ where \ x \in \{x \in \text{keys} \ p. \ \text{component-of-term} x = k\} \ \text{and} \ \ t = \text{pp-of-term} \ x \ \text{..}

from this(1) have \ x \in \text{keys} \ p \ \text{and} \ k = \text{component-of-term} \ x \ \text{by} \ \text{simp-all}

from this(2) have \ x = \text{term-of-pair} \ (t, k) \ \text{by} \ (\text{simp add: term-of-pair-pair} \ t = \text{pp-of-term} \ x)

with \ z \in \text{keys} \ p \ \text{have} \ \text{lookup} \ (\text{term-of-pair} \ (t, k)) \neq 0 \ \text{by} \ (\text{simp add: in-keys-iff})

hence \text{lookup} \ (\text{proj-poly} \ k \ p) \ t \neq 0 \ \text{by} \ (\text{simp add: lookup-proj-poly})

thus \ t \in \text{keys} \ (\text{proj-poly} \ k \ p) \ \text{by} \ (\text{simp add: in-keys-iff})

126
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

lemma keys-atomize-poly:
  keys (atomize-poly p) = (⨆ k∈keys p. (λt. 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) ∈
(λt. term-of-pair (t, component-of-term v)) ' keys (lookup p (component-of-term v))
by (rule imageI)
thus v ∈ (λt. term-of-pair (t, component-of-term v)) ' keys (lookup p (component-of-term v))
by (simp only: term-of-pair)
qed
qed
next
show ?r ⊆ ?l
proof
fix v
assume v ∈ ?r
then obtain k where k ∈ keys p and v ∈ (λt. term-of-pair (t, k)) ' keys (lookup p k) ..
from this(2) obtain t where t ∈ keys (lookup p k) and v = term-of-pair (t, k) ..
from this(1) have lookup (atomize-poly p) v ≠ 0 by (simp add: v lookup-atomize-poly in-keys-iff term-simps)
thus v ∈ ?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-eql, simp add: lookup-vectorize-poly lookup-add proj-plus)

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

lemma vectorize-minus: vectorize-poly \((p - q)\) = vectorize-poly \(p -\) vectorize-poly \(q\)
by (rule poly-mapping-eqlI, simp add: lookup-vectorize-poly lookup-minus proj-minus)

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

lemma atomize-plus [term-simps]: atomize-poly \((p + q)\) = atomize-poly \(p +\) atomize-poly \(q\)
by (rule poly-mapping-eqlI, simp add: lookup-atomize-poly)

lemma atomize-uminus [term-simps]: atomize-poly \((- p)\) = \(-\) atomize-poly \(p\)
by (rule poly-mapping-eqlI, simp add: lookup-atomize-poly)

lemma atomize-minus [term-simps]: atomize-poly \((p - q)\) = atomize-poly \(p -\) atomize-poly \(q\)
by (rule poly-mapping-eqlI, simp add: lookup-atomize-poly)

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-eqlI, 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 \(v\) \(\neq\) 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-eqlI, 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 \(\\vdash\)thesis by (simp add: \(t\) \(k\) vectorize-monomial[symmetric] term-simps)
qed

lemma poly-mapping-eql-proj:
assumes \( k \cdot \text{proj-poly} k p = \text{proj-poly} k q \)
shows \( p = q \)
proof (rule poly-mapping-eqI)
fix \( v :: 't \)
have \( \text{proj-poly} (\text{component-of-term} v) p = \text{proj-poly} (\text{component-of-term} v) q \) by (rule assms)
hence \( \text{lookup} (\text{proj-poly} (\text{component-of-term} v) p) (\text{pp-of-term} v) = \text{lookup} (\text{proj-poly} (\text{component-of-term} v) q) (\text{pp-of-term} v) \) by simp
thus \( \text{lookup} p v = \text{lookup} q v \) 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 ⇒ 0' b) ⇒ ('t ⇒ 0' b)
where monom-mult \( c \cdot t \cdot p = \text{Abs-poly-mapping} (\lambda v. \text{if} \ (t \cdot \text{adds}_p v \ \text{then} \ c \cdot \text{lookup} p \ (v \ominus t) \ \text{else} \ 0) \)

lemma keys-monom-mult-aux:
\( \{ v. (\text{if} \ t \cdot \text{adds}_p v \ \text{then} \ c \cdot \text{lookup} p \ (v \ominus t) \ \text{else} \ 0) \neq 0 \} \subseteq (\oplus) \ t \cdot \text{keys} p \) (is \ ?l \ \subseteq \ ?r)
for \( c :: 'b::semiring-0 \)
proof
fix \( v :: 't \)
assume \( v \in ?l \)
hence \( (\text{if} \ t \cdot \text{adds}_p v \ \text{then} \ c \cdot \text{lookup} p \ (v \ominus t) \ \text{else} \ 0) \neq 0 \) by simp
hence \( t \cdot \text{adds}_p v \ \text{and} \ \text{cp-not-zero} : c \cdot \text{lookup} p \ (v \ominus t) \neq 0 \) by (simp-all split: if-split-asm)
show \( v \in ?r \)
proof
from \( \text{adds-pp-sminus}[OF t \cdot \text{adds}_p v] \) show \( v = t \oplus (v \ominus t) \) by simp
next
from \( \text{mult-not-zero}[OF \text{cp-not-zero}] \) show \( v \ominus t \in \text{keys} p \)
by (simp add: in-keys-iff)
qed
qed

lemma lookup-monom-mult:
\( \text{lookup} (\text{monom-mult} c \cdot t \cdot p) v = (\text{if} \ t \cdot \text{adds}_p v \ \text{then} \ c \cdot \text{lookup} p \ (v \ominus t) \ \text{else} \ 0) \)
proof
have \( \text{lookup} (\text{monom-mult} c \cdot t \cdot p) = (\lambda v. \text{if} \ t \cdot \text{adds}_p v \ \text{then} \ c \cdot \text{lookup} p \ (v \ominus t) \ \text{else} \ 0) \)
unfolding monom-mult-def
proof (rule Abs-poly-mapping-inverse)
from \( \text{finite-keys} \) have \( \text{finite} ((\oplus) t \cdot \text{keys} p) \) ..
with \( \text{keys-monom-mult-aux} \) have \( \text{finite} \ \{ v. (\text{if} \ t \cdot \text{adds}_p v \ \text{then} \ c \cdot \text{lookup} p \ (v \ominus t) \ \text{else} \ 0) \neq 0 \} \)
by (rule finite-subset)
thus \( (\lambda v. \text{if} \ t \cdot \text{adds}_p v \ \text{then} \ c \cdot \text{lookup} p \ (v \ominus t) \ \text{else} \ 0) \in \{ f. \text{finite} \ \{ x. f \cdot x \neq 0 \} \} \)
qed
lemma lookup-monom-mult-plus:
  \text{lookup} \ (\text{monom-mult} \ c \ t \ p) \ (t \oplus v) = (c::'b::semiring-0) \ast \text{lookup} \ p \ v
by (simp add: lookup-monom-mult term-simps)

lemma monom-mult-assoc: monom-mult \ c \ s \ (\text{monom-mult} \ d \ t \ p) = \text{monom-mult} \ (c \ast 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 \oplus s
  hence \ s + t \ adds_p \ v \ by \ (rule \ plus-adds-ppI-2)
  moreover assume \ -s + t \ adds_p \ v
  ultimately show \ c \ast (d \ast \text{lookup} \ p \ (v \ominus (s \ominus 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 \ -t \ adds_p \ v \ominus s
  ultimately show \ c \ast (d \ast \text{lookup} \ p \ (v \ominus (s + t))) = 0 \ by \ simp
qed

lemma monom-mult-uminus-left: monom-mult \ (-c) \ t \ p = -\text{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) = -\text{monom-mult} \ (c::'b::ring) \ t \ p
by (rule poly-mapping-eqI, simp add: lookup-monom-mult)

lemma uminus-monom-mult: \ -p = \text{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 = (\text{monom-mult} \ c \ t \ p) + \ (\text{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 = (\text{monom-mult} \ c \ t \ p) - (\text{monom-mult} \ (d::'b::ring) \ t \ p)
lemmas monom-mul-dist-right:
  monom-mul c t (p + q) = (monom-mul c t p) + (monom-mul c t q)
  by (rule poly-mapping-eqI, simp add: lookup-monom-mul lookup-add algebra-simps)

lemmas monom-mul-dist-right-minus:
  monom-mul c t (p - q) = (monom-mul c t p) - (monom-mul c::'b::ring t q)
  using monom-mul-dist-right[of c t p - q] monom-mul-uminus-right[of c t q] by simp

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

lemmas monom-mul-zero-right [simp]: monom-mul c t 0 = 0
  by (rule poly-mapping-eqI, simp add: lookup-monom-mul)

lemmas monom-mul-one-left [simp]: (monom-mul (1::'b::semiring-1) 0 p) = p
  by (rule poly-mapping-eqI, auto simp add: lookup-monom-mul term-simps)

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

lemmas monom-mul-eq-zero-iff: (monom-mul c t p = 0) ↔ ((c::'b::semiring-no-zero-divisors) = 0 ∨ p = 0)
  proof
    assume eq: monom-mul 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
      hence c * lookup p v = 0 by simp
      with c ≠ 0 lookup p v ≠ 0 show False by auto
    qed
  qed

next
  assume c = 0 ∨ p = 0
  with monom-mul-zero-left[of t p] monom-mul-zero-right[of c t] show monom-mul c t p = 0 by auto
  qed

lemmas lookup-monom-mul-zero: lookup (monom-mul c 0 p) t = c * lookup p t

132
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::(- ⇒ '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 c1 = 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
  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 c1 = lookup p1 v = c2 * lookup p2 v by (simp only: lookup-monom-mult-plus)
  with * show ?thesis by auto
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

133
\begin{verbatim}
assume \( v \in \text{keys} (\text{monom-mult} \; c \; t \; p) \)
hence \( \text{lookup} (\text{monom-mult} \; c \; t \; p) \; v \neq 0 \) by (simp add: in-keys-iff)
thus \( v \in ?A \) unfolding \( \text{lookup-monom-mult} \) by simp
qed
also note \( \text{keys-monom-mult-aux} \)
finally show \( \text{thesis} \).
qed
lemma \( \text{keys-monom-multE} \):
assumes \( v \in \text{keys} (\text{monom-mult} \; c \; t \; p) \)
obtains \( u \) where \( u \in \text{keys} \; p \) and \( v = t \oplus u \)
proof

note \( \text{assms} \)
also have \( \text{keys} (\text{monom-mult} \; c \; t \; p) \subseteq ((\oplus) \; t) \; '^\ast \; (\text{keys} \; p) \).
finally have \( v \in ((\oplus) \; t) \; '^\ast \; (\text{keys} \; p) \).
then obtain \( u \) where \( u \in \text{keys} \; p \) and \( v = t \oplus u \).
thus \( \text{thesis} \).
qed
lemma \( \text{keys-monom-mult} \):
assumes \( c \neq (0 :: 'b :: \text{semiring-no-zero-divisors}) \)
shows \( \text{keys} \; (\text{monom-mult} \; c \; t \; p) = ((\oplus) \; t) \; '^\ast \; (\text{keys} \; p) \)
proof (rule, fact \( \text{keys-monom-mult-subset} \), rule)
fix \( v \)
assume \( v \in ((\oplus) \; t) \; '^\ast \; \text{keys} \; p \)
then obtain \( u \) where \( u \in \text{keys} \; p \) and \( v = t \oplus u \).
from \( \langle u \in \text{keys} \; p \rangle \) \( \text{assms} \) show \( v \in \text{keys} \; (\text{monom-mult} \; c \; t \; p) \) unfolding \( v \) by (rule \( \text{keys-monom-multI} \))
qed
lemma \( \text{monomial-power} \): \( (\text{monomial} \; c \; t) \; '^\ast \; n = (\text{monomial} \; (c \; '^\ast \; n) \; (\sum i=0..<n. \; t)) \)
by (induct \( n \), simp-all add: \( \text{mult-single} \; \text{monom-mult-monomial} \; \text{add.commute} \))
\end{verbatim
definition \( \text{lift-poly-fun} :: (('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)) \Rightarrow ('t \Rightarrow 'b) \Rightarrow ('t \Rightarrow 'b) :: \text{zero}) \)

where \( \text{lift-poly-fun} f p = \text{lift-poly-fun-2} (\lambda f \cdot 0) p \)

lemma lookup-lift-poly-fun-2:
\[
\text{lookup} (\text{lift-poly-fun-2} f p q) v = \\
(\text{lookup} (f (\text{proj-poly} (\text{component-of-term} v) p) (\text{proj-poly} (\text{component-of-term} v) q)) (\text{pp-of-term} v))
\]
when component-of-term \( v \in \text{keys} \ (\text{vectorize-poly} p) \cup \text{keys} \ (\text{vectorize-poly} q) \)
by (simp add: lift-poly-fun-2-def lookup-atomize-poly lookup-mapp-2 lookup-vectorize-poly when-distrib [of \(-\lambda q. \text{lookup} q (\text{pp-of-term} v), \text{OF lookup-zero}])

lemma lookup-lift-poly-fun:
\[
\text{lookup} (\text{lift-poly-fun} f p v) = \\
(\text{lookup} (f (\text{proj-poly} (\text{component-of-term} v) p) (\text{pp-of-term} v)) \text{ when component-of-term } v \in \text{keys} \ (\text{vectorize-poly} p))
\]
by (simp add: lift-poly-fun-def lookup-lift-poly-fun-2 term-simps)

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

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

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

lemma proj-lift-poly-fun-homogenous:
\[
\text{assumes } f \ 0 = 0 \\
\text{shows } \text{proj-poly} k (\text{lift-poly-fun} f p) = f (\text{proj-poly} k p)
\]
by (rule poly-mapping-eqI,
simp add: lookup-proj-poly lookup-lift-poly-fun-homogenous[of \(f\), \(\text{OF assms}\) term-simps]
9.6 Component-wise Multiplication

definition mult-vec :: ('t ⇒ 'b) ⇒ ('t ⇒ 'b) ⇒ ('t ⇒ '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::-'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::-'b::ring
using mult-vec-distrib-right \[ of \text{p} - \text{q} \] by (simp add: mult-vec-minus-mult-left)

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

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

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

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

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

lemma in-keys-mult-vecE:
assumes \text{w} \in keys (\text{p} ** \text{q})
obtains \text{u} \text{v} where \text{u} \in keys \text{p} and \text{v} \in keys \text{q} and component-of-term \text{u} = component-of-term \text{v}
and \text{w} = term-of-pair (pp-of-term \text{u} + pp-of-term \text{v}, component-of-term \text{u})
proof –
from assms have 0 \ne lookup (\text{p} ** \text{q}) \text{w} by (simp add: in-keys-iff)
also have lookup (\text{p} ** \text{q}) \text{w} =
lookup ((proj-poly (component-of-term \text{w}) \text{p}) * (proj-poly (component-of-term \text{w}) \text{q})) (pp-of-term \text{w})
by (fact lookup-mult-vec)
finally have pp-of-term \text{w} \in keys ((proj-poly (component-of-term \text{w}) \text{p}) * (proj-poly

137
(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, 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, 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)))

when component-of-term \( v = \text{component-of-term} \ u \) when when

\[ \text{by (rule when-distrib, simp)} \]

\[ \text{show \ ?thesis} \]

\[ \text{by (simp add: lookup-mult-vec proj-monomial eq1 lookup-times-monomial-right when-when \ adds-term-def lookup-proj-poly conj-commute)} \]

qed

9.7 Scalar Multiplication

definition \( \text{mult-scalar} :: (\'a \Rightarrow \text{0} ^{\prime} b) \Rightarrow (\'t \Rightarrow \text{0} ^{\prime} b) \Rightarrow (\text{0} ^{\prime} b :: \text{semiring-0}) \) (infixl \( \odot \) 75)

where \( \text{mult-scalar} \ p = \text{lift-poly-fun} ((\ast) \ p) \)

lemma lookup-mult-scalar:
lookup \( (p \odot q) \ v = \text{lookup} \ (p * (\text{proj-poly} \ (\text{component-of-term} \ u) \ v)) \) \( \text{pp-of-term} \ v \)

unfolding mult-scalar-def by (rule lookup-lift-poly-fun-homogenous, simp)

lemma lookup-mult-scalar-explicit:
lookup \( (p \odot q) \ u = (\sum t \in \text{keys} \ p. \ \text{lookup} \ p \ t * (\sum v \in \text{keys} \ q. \ \text{lookup} \ q \ v \ \text{when} \ u = t \oplus v)) \)

proof –

let \( \?f = \lambda t \ s. \ \text{lookup} \ (\text{proj-poly} \ (\text{component-of-term} \ u) \ v) \ s \) \( \text{when pp-of-term} \ u = t + s \)

note lookup-mult-scalar

also have \( \text{lookup} \ (p * \text{proj-poly} \ (\text{component-of-term} \ u) \ q) \) \( \text{pp-of-term} \ u \) = \( (\sum t. \ \text{lookup} \ p \ t * (\text{Sum-any} \ (\?f \ t))) \)

by (fact lookup-mult)

also from finite-keys have \( \ldots = (\sum t \in \text{keys} \ p. \ \text{lookup} \ p \ t * (\text{Sum-any} \ (\?f \ t))) \)

by (rule Sum-any.expand-superset) (auto simp: in-keys-iff dest: mult-not-zero)

also from refl have \( \ldots = (\sum t \in \text{keys} \ p. \ \text{lookup} \ p \ t * (\sum v \in \text{keys} \ q. \ \text{lookup} \ q \ v \ \text{when} \ u = t \oplus v)) \)

proof (rule sum.cong)

fix \( t \)

assume \( t \in \text{keys} \ p \)

from finite-keys have \( \text{Sum-any} \ (\?f \ t) = (\sum s \in \text{keys} \ (\text{proj-poly} \ (\text{component-of-term} \ u) \ q)). \?f \ t \)

by (rule Sum-any.expand-superset) (auto simp: in-keys-iff)

also have \( \ldots = (\sum v \in \{x \in \text{keys} \ q. \ \text{component-of-term} \ x = \text{component-of-term} \ u\} \?f t \ (\text{pp-of-term} \ v)) \)

unfolding keys-proj-poly

proof (intro sum.reindex\[simplified o-def] inj-an1)

fix \( v1 \ v2 \)

assume \( v1 \in \{x \in \text{keys} \ q. \ \text{component-of-term} \ x = \text{component-of-term} \ u\} \)

and \( v2 \in \{x \in \text{keys} \ q. \ \text{component-of-term} \ x = \text{component-of-term} \ u\} \)

hence \( \text{component-of-term} \ v1 = \text{component-of-term} \ v2 \) by simp

moreover assume \( \text{pp-of-term} \ v1 = \text{pp-of-term} \ v2 \)
ultimately show \( v_1 = v_2 \) by (metis term-of-pair-pair)
qede
also from finite-keys have \( \ldots = (\sum v \in \text{keys } q. \, \text{lookup } q \, v \text{ when } u = t \oplus v) \)
proof (intro sum.mono-neutral-cong-left ball)
fix \( v \)
assume \( v \in \text{keys } q - \{ x \in \text{keys } q. \, \text{component-of-term } x = \text{component-of-term } u \} \)

hence \( u \neq t \oplus v \) by (auto simp; component-of-term-splus)
thus \( \text{lookup } q \, v \text{ when } u = t \oplus v = 0 \) by simp
next
fix \( v \)
assume \( v \in \{ x \in \text{keys } q. \, \text{component-of-term } x = \text{component-of-term } u \} \)

hence \( \text{component-of-term } v = \text{component-of-term } u \) by simp
have \( u = t \oplus v \longleftrightarrow \text{pp-of-term } u = t + \text{pp-of-term } v \)
proof
assume \( \text{pp-of-term } u = t + \text{pp-of-term } v \)

hence \( \text{pp-of-term } u = \text{pp-of-term } (t \oplus v) \) by (simp only: pp-of-term-splus)
moreover have \( \text{component-of-term } u = \text{component-of-term } (t \oplus v) \)
ultimately show \( u = t \oplus v \) by (metis term-of-pair-pair)
qed (simp add: pp-of-term-splus)
thus \( ?f \, t \) \[ \text{lookup } p \, t \ast (\text{Sum-any } (\text{?f } t)) = \text{lookup } p \, t \ast (\sum v \in \text{keys } q. \, \text{lookup } q \, v \text{ when } u = t \oplus v) \]
by (simp only:)
qed
finally show \( \text{thesis} \).
qede

lemma proj-mult-scalar [term-simps]: \( \text{proj-poly } k \, (p \odot q) = p \ast (\text{proj-poly } k \, q) \)
unfolding mult-scalar-def by (rule proj-lift-poly-fun-homogenous, simp)

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

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

lemma mult-scalar-one [simp]: \( 1::\Rightarrow 0 \, 'b::semiring-1 \odot p = p \)
by (rule poly-mapping-eqI-proj, simp add: term-simps)

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

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

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

lemma mult-scalar-minus-mult-right [simp]: \(p \odot (- q) = - (p \odot q)\)
by simp

lemma minus-mult-scalar-minus [simp]: \((- p) \odot (- q) = p \odot q\)
by simp

lemma minus-mult-scalar-commute: \((- p) \odot q = p \odot (- q)\)
by simp

lemma mult-scalar-right-diff-distrib [algebra-simps]: \(r \odot (p - q) = r \odot p - r \odot q\)
for \(r::\mathbb{R}\rightarrow\mathbb{R}\) by simp

lemma mult-scalar-left-diff-distrib [algebra-simps]: \((p - q) \odot r = p \odot r - q \odot r\)
for \(p::\mathbb{R}\rightarrow\mathbb{R}\) by simp

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

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

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

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

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

lemma in-keys-mult-scalarE:
  assumes \( v \in \text{keys} \ (p \odot q) \)
  obtains \( t \ u \) where \( t \in \text{keys} \ p \) and \( u \in \text{keys} \ q \) and \( v = t \oplus u \)
proof
  from \( \text{assms} \) have \( 0 \neq \text{lookup} \ (p \odot q) \ v \) by (simp add: in-keys-iff)
also have \( \text{lookup} \ (p \odot q) \ v = \text{lookup} \ (p \ast (\text{proj-poly} \ \text{component-of-term} v) \ q) \)
  (\( \text{pp-of-term} v \))
  
  by (fact \( \text{lookup-mult-scalar} \))
finally have \( \text{pp-of-term} v \in \text{keys} \ (p \ast \text{proj-poly} \ \text{component-of-term} v) \ q) \) by
(simp add: in-keys-iff)
from this \( \text{keys-mult} \) have \( \text{pp-of-term} v \in \{ t + s \mid t \ s. \ t \in \text{keys} \ p \ \land \ s \in \text{keys} \ (\text{proj-poly} \ \text{component-of-term} v) \ q \} \ldots \)
then obtain \( t \ s \) where \( t \in \text{keys} \ p \) and \( s \in \text{keys} \ (\text{proj-poly} \ \text{component-of-term} v) \ q) \)

and eq: \( \text{pp-of-term} v = t + s \) by fastforce

note \( \text{this}(1) \)
moreover from \( \ast \) have \( \text{term-of-pair} \ (s, \text{component-of-term} v) \in \text{keys} \ q \)
  by (simp only: in-keys-iff \( \text{lookup-proj-poly} \ \text{not-False-eq-True} \))
moreover have \( v = t \oplus \text{term-of-pair} \ (s, \text{component-of-term} v) \)
  by (simp add: \( \text{splas-def eq}[\text{symmetric}] \text{ term-simps} \))
ultimately show \(?\text{thesis}\) ..
qed

lemma lookup-mult-scalar-monomial-right:
  \( \text{lookup} \ (p \odot \text{monomial} \ c \ v) \ u = (\text{lookup} \ p \ (\text{pp-of-term} \ u - \text{pp-of-term} \ v) \ast c \) when \( v \ \text{adds}_v \ u) \)
proof
  have \( \text{eq1}: \text{lookup} \ (p \ast (\text{monomial} \ c \ \text{pp-of-term} v) \ when \ \text{component-of-term} \ v = \ \text{component-of-term} \ u) \) (\( \text{pp-of-term} u \)) =
  (\( \text{lookup} \ (p \ast (\text{monomial} \ c \ \text{pp-of-term} v)) \) (\( \text{pp-of-term} u \)) \ when
\( \text{component-of-term} \ v = \ \text{component-of-term} \ u) \)
  
  by (rule \( \text{when-distrib}, \text{simp} \))

show \(?\text{thesis}\)
  by (simp add: \( \text{lookup-mult-scalar} \ \text{eq1} \) \( \text{proj-monomial} \text{ lookup-times-monomial-right when-when} \)
  \( \text{adds-term-def lookup-proj-poly conj-commute} \))
qed
lemma lookup-mult-scalar-monomial-right-plus: lookup \(p \odot \text{monomial } c \cdot v\) \((t \oplus v)\) = lookup \(p \odot t \odot c\)
by (simp add: lookup-mult-scalar-monomial-right term-simps)

lemma keys-mult-scalar-monomial-right-subset: keys \((p \odot \text{monomial } c \cdot v)\) \(\subseteq (\lambda t. t \oplus v) \cdot \text{keys } p\)
proof
  fix \(u\)
  assume \(u \in \text{keys } (p \odot \text{monomial } c \cdot v)\)
  then obtain \(t \cdot w\) where \(t \in \text{keys } p\) and \(w \in \text{keys } (\text{monomial } c \cdot v)\) and \(u = t \oplus w\)
  by (rule in-keys-mult-scalarE)
  from this(2) have \(w = v\) by (metis empty-iff insert-iff keys-single)
  from \((t \in \text{keys } p)\) show \(u \in (\lambda t. t \oplus v) \cdot \text{keys } p\) unfolding \((u = t \oplus w) \cdot (w = v)\)
  by fastforce
qed

lemma keys-mult-scalar-monomial-right-
assumes \(c \neq (0::'b::semiring-no-zero-divisors)\)
shows keys \((p \odot \text{monomial } c \cdot v)\) = \((\lambda t. t \oplus v) \cdot \text{keys } p\)
proof
  show \((\lambda t. t \oplus v) \cdot \text{keys } p \subseteq \text{keys } (p \odot \text{monomial } c \cdot v)\)
  proof
    fix \(u\)
    assume \(u \in (\lambda t. t \oplus v) \cdot \text{keys } p\)
    then obtain \(t \cdot w\) where \(t \in \text{keys } p\) and \(u = t \oplus w\)
    have lookup \((p \odot \text{monomial } c \cdot v)\) \((t \oplus v)\) = lookup \(p \odot t \odot c\)
    by (fact lookup-mult-scalar-monomial-right-plus)
    also from \((t \in \text{keys } p)\) assms have \(... \neq 0\) by (simp add: in-keys-iff)
    finally show \(u \in \text{keys } (p \odot \text{monomial } c \cdot v)\)
    by (simp add: in-keys-iff \((u = t \oplus w) \cdot (w = v)\))
  qed
  qed (fact keys-mult-scalar-monomial-right-subset)
end

9.8 Sums and Products

lemma sum-poly-mapping-eq-zeroI:
assumes \(p \cdot A \subseteq \{0\}\)
shows \(\text{sum } p \cdot A = (0::(- :=_0 'b::comm-monoid-add))\)
proof (rule ccontr)
  assume \(\text{sum } p \cdot A \neq 0\)
  then obtain \(a\) where \(a \in A\) and \(p \cdot a \neq 0\)
  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 (λp. 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 (set ps) by blast
  also have ... = Keys (set (p # ps)) by blast
  finally show ?case.
qed

lemma keys-sum:
  assumes finite A and \( a1 a2. a1 \in A \land a2 \in A \Rightarrow a1 \neq a2 \Rightarrow \text{keys } f a1 \cap \text{keys } f a2 = \{\}\)
  shows keys (sum f A) = (\( \bigcup \{\text{keys } f i \mid i \in A\}\))
  using assms
proof (induct A)
case empty
  show ?case by simp
next
case (insert a A)
  have IH: keys (sum f A) = (\( \bigcup \text{keys } f i \mid i \in A\)) 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) = (\( \bigcup \text{keys } f i \mid i \in A\))
      proof (simp only: IH Int-UN-distrib)
    also have ... = {}
      proof
        have i ∈ A \Rightarrow keys (f a) ∩ keys (f i) = {} for i
        proof (rule insert.prems)
          assume i ∈ A
          with insert(2) show a ≠ i by blast
      qed
    qed
  qed
  also have keys (f a) = (\( \bigcup \{\text{keys } f i \mid i \in A\}\))
    proof (rule insert.prems)
      assume i ∈ A
      with insert(2) show a ≠ i by blast
    qed
  finally show ?case.
qed
qed simp-all
thus thesis by simp
qed
finally show \( \text{keys (f a)} \cap \text{keys (sum f A)} = \{\} \).
qed
also have \( \ldots = (\bigcup a \in \text{insert a A. keys (f a)}) \) by (simp add: IH)
finally show thesis case.
qed

lemma poly-mapping-sum-monomials: \( \left( \sum a \in \text{keys p. monomial (lookup p a)} a \right) = p \)
proof (induct p rule: poly-mapping-plus-induct)
case 1
show thesis 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 \(*: \text{keys (monomial c t + p)} = \text{insert t (keys p)} \)
proof
  from step(1) have a: \( \text{keys (monomial c t)} = \{t\} \) by simp
  with step(2) have \( \text{keys (monomial c t)} \cap \text{keys p} = \{\} \) by simp
  hence \( \text{keys (monomial c t + p)} = \{t\} \cup \text{keys p} \) by (simp only: a keys-plus-eqI)
  thus thesis by simp
qed
have \(*\): \( \left( \sum ta \in \text{keys p. monomial ((c when t = ta) + lookup p ta)} ta \right) = \)
\( \left( \sum ta \in \text{keys p. monomial (lookup p ta)} ta \right) \)
proof (rule comm-monoid-add-class.sum.cong, rule refl)
  fix s
  assume s \( \in \text{keys p} \)
  with step(2) have t \( \neq s \) by auto
  thus \( \text{monomial ((c when t = s) + lookup p s)} s = \text{monomial (lookup p s)} s \)
  by simp
qed
show thesis 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: \( \text{monomial (sum f C)} a = \left( \sum c \in C. \text{monomial (f c)} a \right) \)
by (rule fun-sum-commute, simp-all add: single-add)

lemma monomial-Sum-any:
  assumes finite \( \{c. f c \neq 0\} \)
  shows \( \text{monomial (Sum-any f)} a = \left( \sum c. \text{monomial (f c)} a \right) \)
proof
  have \( \{c. \text{monomial (f c) a} \neq 0\} \subseteq \{c. f c \neq 0\} \) by (rule, auto)
  with assms show thesis
  by (simp add: Groups-Big-Fun.comm-monoid-add-class.Sum-any.expand-superset monomial-sum)

145
qed

context term-powerprod
begin

lemma proj-sum: proj-poly k (\sum f A) = (\sum a \in 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 \odot\ p = (\sum t \in \text{keys } q. \text{monom-mult} (\text{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 \forall x y. f (x + y) = f x + f y
  and \forall c t. f (\text{monom-mult } c t p) = \text{monom-mult } c t (f p)
  shows f (q \odot p) = q \odot (f p)
  by (simp add: mult-scalar-sum-monomials assms [symmetric], rule fun-sum-commute, fact+)

lemma fun-mult-scalar-commute-canc:
  assumes \forall x y. f (x + y) = f x + f y and \forall c t. f (\text{monom-mult } c t p) = \text{monom-mult } c t (f p)
  shows f (q \odot p) = q \odot (f (p :: t \Rightarrow 'b::{semiring_0,cancel-comm-monoid-add}))
  by (simp add: mult-scalar-sum-monomials assms [symmetric], rule fun-sum-commute-canc, fact)

lemma monom-mult-sum-left: \text{monom-mult} (\sum f C) t p = (\sum c \in C. \text{monom-mult } (f c) t p)
  by (rule fun-sum-commute, simp-all add: monom-mult-dist-left)

lemma monom-mult-sum-right: \text{monom-mult } c t (\sum f P) = (\sum p \in P. \text{monom-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 \text{monom-mult } (\text{Sum-any } f) t p = (\sum c \in C. \text{monom-mult } (f c) t p)
  proof
  have \{c. \text{monom-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 \( \{p \cdot f \neq 0\} \)
shows monom-mult \( \cdot t \)(Sum-any \( f \)) = (\( \sum \) p. monom-mult \( \cdot t \)(f p))
proof –
  have \( \{p \cdot monom-mult \cdot t \}(f p) \neq 0\} \subseteq \{p \cdot f \neq 0\} \) by (rule, auto)
  with assms show \( \text{thesis} \)
    by (simp add: Groups-Big-Fun.comm-monoid-add-class.Sum-any.expand-superset monom-mult-sum-right)
qed

lemma monomial-prod-sum: monomial \( \cdot (\prod c I \cdot a I) \) = (\( \prod \) i\( \in \)I. monomial \( \cdot (c i \cdot a i) \))
proof (cases finite I)
  case True
  thus \( \text{thesis} \)
  proof (induct I)
    case empty
    show \( \text{case by simp} \)
    next
    case (insert i I)
    show \( \text{case} \)
      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 \( \text{thesis by simp} \)
  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 ∈ pmdl B
shows \( \text{monom-mult } c \ t \ p \in \text{pmdl } B \)

unfolding \( \text{mult-scalar-monomial}[\text{symmetric}] \) using assms by (rule pmdl.span-scale)

lemma monom-mult-in-pmdl : \( b \in B \Rightarrow \text{monom-mult } c \ t \ b \in \text{pmdl } B \)

by (intro pmdl-closed-monom-mult pmdl-span-base)

lemma pmdl-induct [consumes 1, case-names module-0 module-plus]:

assumes \( p \in \text{pmdl } B \) and \( P 0 \)

and \( \forall a \ p \ c \ t. \ a \in \text{pmdl } B \Rightarrow P a \Rightarrow p \in B \Rightarrow c \neq 0 \Rightarrow P (a + \text{monom-mult } c \ t \ p) \)

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) \( b \in B \) have \( P (a + \text{monomial } (\text{lookup } q0 \ t) \ t \odot b) \)

unfolding \( \text{mult-scalar-monomial} \)

proof (rule assms(3))

from step(2) show \( \text{lookup } q0 \ t \neq 0 \) by (simp add: in-keys-iff)

qed

with - have \( P ((a + \text{monomial } (\text{lookup } q0 \ t) \ t \odot b) + \text{except } q0 \ \{t\} \odot b) \)

proof (rule step(3))

from \( b \in B \) have \( b \in \text{pmdl } B \) by (rule pmdl.span-base)

hence \( \text{monomial } (\text{lookup } q0 \ t) \ t \odot b \in \text{pmdl } B \) by (rule pmdl.span-scale)

with step(4) show \( a + \text{monomial } (\text{lookup } q0 \ t) \ t \odot b \in \text{pmdl } B \) by (rule pmdl.span-add)

qed

hence \( P (a + (\text{monomial } (\text{lookup } q0 \ t) \ t + \text{except } q0 \ \{t\}) \odot b) \) by (simp add: algebra-simps)

thus \( ?case \) by (simp only: plus-except[of q0 t, symmetric])

qed

qed

lemma components-pmdl: \( \text{component-of-term }^{\ '} \text{Keys } (\text{pmdl } B) = \text{component-of-term }^{\ '} \text{Keys } B \)

proof

show \( \text{component-of-term }^{\ '} \text{Keys } (\text{pmdl } B) \subseteq \text{component-of-term }^{\ '} \text{Keys } B \)

proof

fix \( k \)

assume \( k \in \text{component-of-term }^{\ '} \text{Keys } (\text{pmdl } B) \)

then obtain \( v \) where \( v \in \text{Keys } (\text{pmdl } B) \) and \( k = \text{component-of-term } v \).

148
\begin{verbatim}
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 ⊕ w⟩ term-simps)
  moreover from ⟨u ∈ keys p⟩ ind.hyps(3) have u ∈ Keys B by (rule
  in-KeysI)
  ultimately show ?thesis ..
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 \( \land b1 b2. b1 ∈ B \implies b2 ∈ B \implies b1 + b2 ∈ B \)
  and \( \land c t b. b ∈ B \implies monom-mult c t b ∈ B \)
  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

149
\end{verbatim}
definition full-pmdl :: \k \rightarrow (\{ k \} \Rightarrow \{ 0 \}) set
where full-pmdl K = \{ p. \text{component-of-term} \ ' \text{keys} p \subseteq K \}

definition is-full-pmdl :: \{} \Rightarrow \{ \} \Rightarrow \text{bool}
where is-full-pmdl B \longleftrightarrow (\forall p. \text{component-of-term} \ ' \text{keys} p \subseteq \text{component-of-term} \ ' \text{Keys} B \Rightarrow p \in \text{pmdl} B)\)

lemma full-pmdl-iff: p \in full-pmdl K \longleftrightarrow \text{component-of-term} \ ' \text{keys} p \subseteq K
by (simp add: full-pmdl-def)

lemma full-pmdlI:
assumes \( \forall v. v \in \text{keys} p \Rightarrow \text{component-of-term} v \in K \)
shows p \in full-pmdl K
using assms by (auto simp add: full-pmdl-iff)

lemma full-pmdlD:
assumes p \in full-pmdl K and v \in \text{keys} p
shows \text{component-of-term} v \in 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 \in full-pmdl K
by (simp add: full-pmdl-iff)

lemma full-pmdl-closed-plus:
assumes p \in full-pmdl K and q \in full-pmdl K
shows p + q \in full-pmdl K
proof (rule full-pmdlI)
fix v
assume v \in \text{keys} (p + q)
also have ... \subseteq \text{keys} p \cup \text{keys} q by (fact Poly-Mapping.keys-add)
finally show \text{component-of-term} v \in K
proof
assume v \in \text{keys} p
with assms(1) show ?thesis by (rule full-pmdlD)
next
assume v \in \text{keys} q
with assms(2) show ?thesis by (rule full-pmdlD)
qed
qed
lemma full-pmdl-closed-monom-mult:
  assumes \( p \in \text{full-pmdl } K \)
  shows monom-mult \( c \cdot t \cdot p \in \text{full-pmdl } K \)
proof (rule full-pmdlI)
  fix \( v \)
  assume \( v \in \text{keys } (\text{monom-mult } c \cdot t \cdot p) \)
  also have \( \ldots \subseteq (v \oplus) \) \( t \cdot \text{keys } p \) by (fact keys-monom-mult-subset)
  finally obtain \( u \) where \( u \in \text{keys } p \) and \( v = t \oplus u \) ..
  have component-of-term \( v = \text{component-of-term } u \) by (simp add: v term-simps)
  also from \( \text{assms } (u \in \text{keys } p) \) have \( \ldots \in K \) by (rule full-pmdlD)
  finally show \( \text{component-of-term } v \in K \).
qed

lemma pmdl-full-pmdl: \( \text{pmdl } (\text{full-pmdl } K) = \text{full-pmdl } K \)
  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 \((\text{full-pmdl } K)::('t \Rightarrow 'b::zero) \) set \( \subseteq K \) (is \(?l \subseteq -\))
proof
  let \(?M = (\text{full-pmdl } K)::('t \Rightarrow 'b) \) set
  fix \( k \)
  assume \( k \in ?l \)
  then obtain \( v \) where \( v \in \text{Keys } ?M \) and \( k = \text{component-of-term } v \) ..
  from this(1) obtain \( p \) where \( p \in ?M \) and \( v \in \text{keys } p \) by (rule in-KeysE)
  thus \( k \in K \) unfolding \( k \) by (rule full-pmdlD)
qed

lemma components-full-pmdl:
  component-of-term \( ' \) Keys \((\text{full-pmdl } K)::('t \Rightarrow 'b::zero-neq-one) \) set \( = K \) (is \(?l = -\))
proof
  let \(?M = (\text{full-pmdl } K)::('t \Rightarrow 'b) \) set
  show \( K \subseteq ?l \)
  proof
    fix \( k \)
    assume \( k \in K \)
    hence monomial 1 \((\text{term-of-pair } (0, k)) \) \in \(?M \) by (simp add: full-pmdl-iff term-simps)
    hence keys \((\text{monomial } (1::'b) (\text{term-of-pair } (0, k))) \) \subseteq Keys \(?M \) by (rule keys-subset-Keys)
    hence term-of-pair \((0, k) \) \in Keys \(?M \) by simp
    hence component-of-term \((\text{term-of-pair } (0, k)) \) \in component-of-term \( ' \) Keys \(?M \) by (rule imageI)
    thus \( k \in ?l \) by (simp only: component-of-term-of-pair)
  qed
  qed (fact components-full-pmdl-subset)
lemma is-full-pmdlI:
  assumes \( \land p. \text{component-of-term } \Downarrow \text{keys } p \subseteq \text{component-of-term } \Downarrow \text{Keys } B \Rightarrow p \in \text{pmdl } B \)
  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 \( \Downarrow \text{keys } p \subseteq \text{component-of-term } \Downarrow \text{Keys } B \)
  shows \( p \in \text{pmdl } B \)
  using assms unfolding is-full-pmdl-def by blast

lemma is-full-pmdl-alt: is-full-pmdl B \iff pmdl B = full-pmdl (component-of-term \( \Downarrow \text{Keys } B \))
proof
  have \( b \in \text{pmdl } B \Rightarrow v \in \text{keys } b \Rightarrow \text{component-of-term } v \in \text{component-of-term } \Downarrow \text{Keys } B \)
    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-subset:
  assumes is-full-pmdl B1 and is-full-pmdl B2
    and component-of-term \( \Downarrow \text{Keys } B1 \subseteq \text{component-of-term } \Downarrow \text{Keys } B2 \)
  shows \( \text{pmdl } B1 \subseteq \text{pmdl } B2 \)
proof
  fix \( p \)
  assume \( p \in \text{pmdl } B1 \)
  from assms(2) show \( p \in \text{pmdl } B2 \)
  proof (rule is-full-pmdlD)
    have \( \text{component-of-term } \Downarrow \text{keys } p \subseteq \text{component-of-term } \Downarrow \text{Keys } (\text{pmdl } B1) \)
      by (rule image-mono, rule keys-subset-Keys, fact)
    also have \( \ldots = \text{component-of-term } \Downarrow \text{Keys } B1 \)
      by (fact components-pmdl)
    finally show \( \text{component-of-term } \Downarrow \text{keys } p \subseteq \text{component-of-term } \Downarrow \text{Keys } B2 \)
      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 \( \Downarrow \text{Keys } B1 = \text{component-of-term } \Downarrow \text{Keys } B2 \)
  shows \( \text{pmdl } B1 = \text{pmdl } B2 \)
proof
  have \( \text{component-of-term } \Downarrow \text{Keys } B1 \subseteq \text{component-of-term } \Downarrow \text{Keys } B2 \)
    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 ⇒ 0' b) ⇒ ('a ⇒ 0' b :: mult-zero) (infixr · 71)
where map-scale c = Poly-Mapping.map ((•) c)

If the polynomial mapping $p$ is interpreted as a power-product, then $c \cdot p$ corresponds to exponentiation; if it is interpreted as a (vector-) polynomial, then $c \cdot p$ corresponds to multiplication by scalar from the coefficient type.

lemma lookup-map-scale [simp]: lookup (c \cdot p) = (λx. c \cdot lookup p x)
by (auto simp: map-scale-def map_rep-eq when-def)

lemma map-scale-single [simp]: $k \cdot Poly-Mapping.single x l = Poly-Mapping.single x (k \cdot l)$
by (simp add: map-scale-def)

lemma map-scale-zero-left [simp]: $0 \cdot t = 0$
by (rule poly-mapping-eqI simp)

lemma map-scale-zero-right [simp]: $k \cdot 0 = 0$
by (rule poly-mapping-eqI simp)

lemma map-scale-eq-0-iff: $c \cdot t = 0 \iff \left((c::semiring-no-zero-divisors) = 0 \lor t = 0\right)$
by (metis aux lookup-map-scale mult-zero-right subsetI)

lemma keys-map-scale-subset: keys (k \cdot t) ⊆ keys t
by (metis in-keys_iff lookup-map-scale mult-eq-0_iff)

lemma keys-map-scale: keys ((k::'b::semiring-no-zero-divisors) \cdot t) = (if k = 0 then {} else keys t)
proof (split if-split, intro conjI implI)
  assume k = 0
  thus keys (k \cdot t) = {} by simp
next
  assume k ≠ 0
  show keys (k \cdot t) = keys t
proof
  show keys t ⊆ keys (k \cdot 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}) \cdot t = t

153
by (rule poly-mapping-eqI) simp

lemma map-scale-assoc [ac-simps]: \( c \cdot d \cdot t = (c \cdot d) \cdot (t :-> 0 :\{\text{semigroup-mult,zero}\}) \)
   by (rule poly-mapping-eqI) (simp add: ac-simps)

lemma map-scale-distrib-left [algebra-simps]: \( (k :-> t) \cdot (s + t) = k \cdot s + k \cdot t \)
   by (rule poly-mapping-eqI) (simp add: lookup-add distrib-left)

lemma map-scale-distrib-right [algebra-simps]: \( (k + (l :-> t)) \cdot t = k \cdot t + l \cdot t \)
   by (rule poly-mapping-eqI) (simp add: lookup-add distrib-right)

lemma map-scale-Suc: \( (\text{Suc } k) \cdot t = k \cdot t + t \)
   by (rule poly-mapping-eqI) (simp add: lookup-add distrib-right)

lemma map-scale-uminus-left: \( - (k :-> p) = -(k \cdot p) \)
   by (rule poly-mapping-eqI) auto

lemma map-scale-uminus-right: \( (k :-> p) \cdot (- p) = -(k \cdot p) \)
   by (rule poly-mapping-eqI) auto

lemma map-scale-uminus-uminus [simp]: \( - (k :-> p) = -(k \cdot p) \)
   by (simp add: map-scale-uminus-left map-scale-uminus-right)

lemma map-scale-minus-distrib-left [algebra-simps]:
   \( (k :-> t) \cdot (p - q) = k \cdot p - k \cdot q \)
   by (rule poly-mapping-eqI) (auto simp add: lookup-minus right-diff-distrib)

lemma map-scale-minus-distrib-right [algebra-simps]:
   \( (k - (l :-> t)) \cdot f = k \cdot f - l \cdot f \)
   by (rule poly-mapping-eqI) (auto simp add: lookup-minus left-diff-distrib)

lemma map-scale-sum-distrib-left: \( (k :-> f) \cdot (\text{sum } f A) = (\sum_{a \in A} k \cdot f a) \)
   by (induct A rule: infinite-finite-induct) (simp-all add: map-scale-distrib-left)

lemma map-scale-sum-distrib-right: \( (\text{sum } f :-> f) \cdot A = (\sum_{a \in A} k \cdot f a \cdot p) \)
   by (induct A rule: infinite-finite-induct) (simp-all add: map-scale-distrib-right)

lemma deg-pm-map-scale: \( \text{deg-pm } (k \cdot t) = (k :-> \text{deg-pm } t) \cdot (k \cdot t) \)

proof
  from keys-map-scale-subset finite-keys have deg-pm \((k \cdot t) = \text{sum } (\text{lookup } (k \cdot t) (\text{keys } t))\)
    by (rule deg-pm-superset)
  also have \( \ldots = k \cdot \text{sum } (\text{lookup } t) (\text{keys } t) \) by (simp add: sum-distrib-left)
  also from subset-refl finite-keys have \( \text{sum } (\text{lookup } t) (\text{keys } t) = \text{deg-pm } t \)
    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 ⇒0 'b. Nevertheless, both modules can be sets of vector-polynomials of type 't ⇒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

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 = (λx. 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)

class 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.monom-mult-scalar 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.monom-mult-scalar[ symmetric] mult-scalar-assoc mult-scalar-monomial)

end

9.10.2 Interpretation of term-powerprod by 'a × 'k

interpretation pprod: term-powerprod (λx::'a::comm-powerprod × 'k::linorder. x) λx. 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-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.mult-scalar-rec-right[simplified]
lemmas lookup-times = punit.lookup-mult-scalar-explicit[simplified]
lemmas map-scale-eq-times = punit.map-scale-eq-mult-scalar[simplified]

end

10 Type-Class-Multivariate Polynomials in Ordered Terms

theory MPoly-Type-Class-Ordered
  imports MPoly-Type-Class
begin

class the-min = linorder +
  fixes the-min::'a
  assumes the-min-min: the-min ≤ x

  Type class the-min guarantees that a least element exists. Instances of the-min should provide computable definitions of that element.

instantiation nat :: the-min
begin
  definition the-min-nat = (0::nat)
  instance 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 +
  for pair-of-term::'t ⇒ ('a::comm-powerprod × 'k::{the-min,wellorder})
  and term-of-pair::('a × 'k) ⇒ 't
  and ord::'a ⇒ 'a ⇒ bool (infixl ≤ 50)
  and ord-strict (infixl < 50)
  and ord-term::'t ⇒ 't ⇒ bool (infixl ≤₄ 50)
  and ord-term-strict::'t ⇒ 't ⇒ bool (infixl <₄ 50) +
  assumes splus-mono: v ≤₄ w ⇒ t ⊕ v ≤₄ t ⊕ w
assumes ord-term1: pp-of-term v ≤ pp-of-term w ⇒ component-of-term v ≤ component-of-term w ⇒ v ⪯ t

begin

abbreviation ord-term-conv (infixl ≥t 50) where ord-term-conv ≡ (≤t)^−1
abbreviation ord-term-strict-conv (infixl >t 50) where ord-term-strict-conv ≡ (<t)^−1

The definition of ordered-term only covers TOP and POT orderings. These two types of orderings are the only interesting ones.

definition min-term ≡ term-of-pair (0, the-min)

lemma min-term-min: min-term ⪯ t v
proof (rule ord-termI)
  show pp-of-term min-term ≤ pp-of-term v by (simp add: min-term-def zero-min term-simps)
next
  show component-of-term min-term ≤ component-of-term v by (simp add: min-term-def the-min-min term-simps)
qed

lemma splus-mono-strict:
  assumes v ≺ t w
  shows t ⊕ v ≺ t ⊕ w
proof
  from assms have v ≤ t w and v ≠ w by simp
  from this(1) have t ⊕ v ≤ t ⊕ w by (rule splus-mono)
  moreover from (v ≠ w) have t ⊕ v ≠ t ⊕ w by (simp add: term-simps)
  ultimately show ?thesis using ord-term-lin.antisym-conv1 by blast
qed

lemma splus-mono-left:
  assumes s ≤ t
  shows s ⊕ v ≤ t ⊕ v
proof (rule ord-termI, simp-all add: term-simps)
  from assms show s + pp-of-term v ≤ t + pp-of-term v by (rule plus-monotone)
qed

lemma splus-mono-strict-left:
  assumes s ≺ t
  shows s ⊕ v ≺ t ⊕ v
proof
  from assms have s ≤ t and s ≠ t by simp
  from this(1) have s ⊕ v ≤ t ⊕ v by (rule splus-mono-left)
  moreover from (s ≠ t) have s ⊕ v ≠ t ⊕ 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 $t \oplus w \prec t \oplus v$ 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 $t \oplus w \preceq t \oplus v$ by (rule splus-mono)
  with assms show False by simp
qed

lemma ord-term-canc-left:
  assumes $t \oplus v \preceq_s s \oplus v$
  shows $t \preceq s$
proof (rule ccontr)
  assume $\neg t \preceq s$
  hence $s \prec t$ by simp
  hence $s \oplus v \prec t \oplus v$ by (rule splus-mono-strict-left)
  with assms show False by simp
qed

lemma ord-term-strict-canc-left:
  assumes $t \oplus v \prec_s s \oplus v$
  shows $t \prec s$
proof (rule ccontr)
  assume $\neg t \prec s$
  hence $s \preceq t$ by simp
  hence $s \oplus v \preceq t \oplus v$ by (rule splus-mono-left)
  with assms show False by simp
qed

lemma ord-adds-term:
  assumes $u \text{ adds}_t v$
  shows $u \preceq_t v$
proof —
from assms have $*: \text{ component-of-term } u \preceq \text{ component-of-term } v$ and $pp\text{-of-term } u \text{ adds } pp\text{-of-term } v$
  by (simp-all add; adds-term-def)
from this(2) have $pp\text{-of-term } u \preceq pp\text{-of-term } v$ by (rule ord-adds)
from this * show $?thesis$ by (rule ord-termI)
qed
10.1 Interpretations

context ordered-powerprod
begin

10.1.1 Unit

sublocale punit: ordered-term to-pair-unit fst (\leq) (\prec) (\leq) (\prec)
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 ⇒ 'b :: zero) ⇒ 't ⇒ ('t ⇒ 'b :: zero)
where higher p t = except p {s. s \leq t}

definition lower :: ('t ⇒ 'b :: zero) ⇒ 't ⇒ ('t ⇒ 'b :: zero)
where lower p t = except p {s. t \leq s}

definition lt :: ('t ⇒ 'b :: zero) ⇒ 't
where lt p = (if p = 0 then min-term else ord-term-lin.Min (keys p))

abbreviation lp p ≡ pp-of-term (lt p)

definition lc :: ('t ⇒ 'b :: zero) ⇒ 'b
where lc p = lookup p (lt p)

definition tt :: ('t ⇒ 'b :: zero) ⇒ '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 ⇒ 'b :: zero) ⇒ 'b
where tc p ≡ lookup p (tt p)

definition tail :: ('t ⇒ 'b :: zero) ⇒ ('t ⇒ 'b :: zero)
where tail p ≡ lower p (lt p)

end
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 ≠ 0
  shows v ≼ₜ lt p
  proof –
  from assms have t-in: v ∈ keys p by (simp add: in-keys-iff)
  hence keys p ≠ {} by auto
  hence p ≠ 0 using keys-zero by blast
  qed

**lemma** lt-eqI:
  assumes lookup p v ≠ 0 and ∃u. lookup p u ≠ 0 ⇒ u ≼ₜ v
  shows lt p = v
  proof –
  from assms(1) have v ∈ keys p by (simp add: in-keys-iff)
  hence keys p ≠ {} by auto
  hence p ≠ 0 using keys-zero by blast
  have u ≼ₜ v if u ∈ keys p for u
  proof –
  from that have lookup p u ≠ 0 by (simp add: in-keys-iff)
  thus u ≼ₜ v by (rule assms(2))
  qed
  from lt-alt[OF ⟨p ≠ 0⟩] ord-term-lin.Max-eqI[OF finite-keys this ⟨v ∈ keys p⟩] show ?thesis by simp
  qed

**lemma** lt-less:
  assumes p ≠ 0 and ∃u. v ≼ₜ u ⇒ lookup p u = 0
  shows lt p ≺ₜ v
proof
from \( p \neq 0 \) have keys \( p \neq \{ \} \)
by simp
have \( \forall u \in \text{keys} \ p. \ u \prec_t v \)
proof
fix \( u :: t \)
assume \( u \in \text{keys} \ p \)
hence lookup \( p u \neq 0 \) by (simp add: in-keys-iff)
hence \( \neg v \preceq_t u \) using assms(2)[of \( u \)] by auto
thus \( u \preceq_t v \) by simp
qed
with lt-alt[OF assms(1)] ord-term-lin.Max-less-iff[OF finite-keys \( \{\} \)]
show \( \text{thesis} \) by simp
qed

lemma lt-le:
assumes \( \forall u. \ v \prec u \Longrightarrow \text{lookup} \ p u = 0 \)
shows \( \text{lt} \ p \preceq_t v \)
proof (cases \( p = 0 \))
  case True
  show \( \text{thesis} \) by (simp add: True min-term-min)
next
  case False
  hence \( \text{keys} \ p \neq \{ \} \) by simp
  have \( \forall u \in \text{keys} \ p. \ u \preceq_t v \)
  proof
    fix \( u :: t \)
    assume \( u \in \text{keys} \ p \)
    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
  show \( \text{thesis} \) by simp
qed

lemma lt-gr:
assumes \( \text{lookup} \ p s \neq 0 \) and \( t \prec s \)
shows \( t \prec_t \text{lt} \ p \)
using assms lt-max ord-term-lin.order.strict-trans2 by blast

lemma lc-not-0:
assumes \( p \neq 0 \)
shows \( \text{lc} \ p \neq 0 \)
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 \( \text{thesis} \) by (simp add: in-keys-iff lc-def)
qed
lemma lc-eq-zero-iff: lc p = 0 ↔ p = 0
using lc-not-0 lc-zero by blast

lemma lt-in-keys:
  assumes p ≠ 0
  shows lt p ∈ (keys p)
  by (metis assms in-keys-iff lc-def lc-not-0)

lemma lt-monomial:
  assumes c ≠ 0
  shows lt (monomial c t) = t
  by (metis assms lookup-single-eq lookup-single-not-eq lt-eqI ord-term-lin.eq-iff)

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 ⪯ t v ←→ (∀ u. v ≺₄ u → lookup p u = 0) (is ?L ←→ ?R)
proof
  assume ?L
  show ?R
  proof (intro allI impI)
    fix u
    note (lt p ⪯₄ v)
    also assume v ≺₄ u
    finally have lt p ≺₄ u .
    hence ¬ u ⪯₄ 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 ≺₄ lt q
  shows lt (p + q) = lt q
proof (cases q = 0)
  case True
  with assms have lt p ≺₄ 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 ≤ₜ lt q
    proof (rule ccontr)
      assume ¬ u ≤ₜ lt q
      hence qs: lt q ∼ₜ u by simp
      with assms have lt p ∼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 ∼ₜ lt p
  shows lt (p + q) = lt p
proof (cases p = 0)
  case True
  with assms have lt q ∼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) ≠ 0
  unfolding lc-def by simp
next
  fix u
  assume lookup (p + q) u ≠ 0
  show u ≤ₜ lt p
    proof (rule ccontr)
      assume ¬ u ≤ₜ lt p
      hence ps: lt p ∼ₜ u by simp
      with assms have lt q ∼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 ≠ 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 \( lt \ (p + q) = lt \ (p::'t \Rightarrow 'b::monoid-add) \)
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 \( p + lookups q \) \( u \neq 0 \) by (simp add: lookup-add)
  hence lookup \( p \) \( u \neq 0 \) \( \lor \) 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 \( lt \ p \prec_t lt \ (p + q) \)
  shows \( lt \ p \prec_t lt \ q \)
proof (rule ccontr)
  assume \( \neg \lt \ p \prec_t lt \ q \)
  hence \( lt \ p = lt \ q \lor lt \ q \prec_t lt \ p \) by auto
  thus False
  proof
    assume lt-eq: \( lt \ p = lt \ q \)
    have \( lt \ (p + q) \preceq_t lt \ p \)
    proof (rule lt-le)
      fix \( u \)
      assume \( lt \ p \prec_t u \)
      with \( \lt-gr[of p u lt p] \) have lookup \( p \) \( u = 0 \) by blast
      from \( \lt p \prec_t u \) have \( lt \ q \prec_t u \) \( \text{using lt-eq by simp} \)
      with \( \lt-gr[of q u lt q] \) have lookup \( q \) \( u = 0 \) by blast
      with \( \lt p = lt \ q \) \( u = 0 \) show lookup \( p + q \) \( u = 0 \) by (simp add: lookup-add)
    qed
    with assms show False by simp
  next
    assume \( lt \ q \prec_t lt \ p \)
    from lt-plus-eqI-2[OF this] assms show False by simp
  qed
qed

qed
lemma \textit{lt-plus-lessE-2}:  
assumes \( \text{lt } q \prec_t \text{lt } (p + q) \)  
shows \( \text{lt } q \prec_t \text{lt } p \)  
proof (rule ccontr)  
assume \( \neg \text{lt } q \prec_t \text{lt } p \)  
hence \( \text{lt } q = \text{lt } p \lor \text{lt } p \prec_t \text{lt } q \) by auto  
thus False  
proof  
assume \( \text{lt-eq: } \text{lt } q = \text{lt } p \)  
have \( \text{lt } (p + q) \leq_t \text{lt } q \)  
proof (rule lt-le)  
fix \( u \)  
assume \( \text{lt } q \prec_t u \)  
with \( \text{lt-gr}[\{q \ u \ | \text{lt } q\}] \) have \( \text{lookup } q \ u = 0 \) by blast  
from \( \langle \text{lt } q \prec_t u \rangle \) have \( \text{lt } p \prec_t u \) using \( \text{lt-eq by simp} \)  
with \( \text{lt-gr}[\{p \ u \ | \text{lt } p\}] \) have \( \text{lookup } p \ u = 0 \) by blast  
with \( \langle \text{lookup } q \ u = 0 \rangle \) show \( \text{lookup } (p + q) \ u = 0 \) by (simp add: lookup-add)  
qed  
with \( \text{assms show } False \) by simp  
next  
assume \( \text{lt } p \prec_t \text{lt } q \)  
from \( \text{lt-plus-eqI[OF this]} \) assms show \( False \) by simp  
qed  

lemma \textit{lt-plus-lessI\':}  
fixes \( p \ q :: 't \Rightarrow 'b::monoid-add \)  
assumes \( p + q \neq 0 \) and \( \text{lt-eq: } \text{lt } q = \text{lt } p \) and \( \text{lc-eq: } \text{lc } p + \text{lc } q = 0 \)  
shows \( \text{lt } (p + q) \prec_t \text{lt } p \)  
proof (rule ccontr)  
assume \( \neg \text{lt } (p + q) \prec_t \text{lt } p \)  
hence \( \text{lt } (p + q) = \text{lt } p \lor \text{lt } p \prec_t \text{lt } (p + q) \) by auto  
thus False  
proof  
assume \( \text{lt } (p + q) = \text{lt } p \)  
have \( \text{lookup } (p + q) \ (\text{lt } p) = (\text{lookup } p \ (\text{lt } p)) + (\text{lookup } q \ (\text{lt } q)) \) unfolding \( \text{lt-eq lookup-add} \)  
also have \( ... = \text{lc } p + \text{lc } q \) unfolding \( \text{lc-def} \)  
also have \( ... = 0 \) unfolding \( \text{lc-eq by simp} \)  
finally have \( \text{lookup } (p + q) \ (\text{lt } p) = 0 \) .  
hence \( \text{lt } (p + q) \neq \text{lt } p \) using \( \text{lc-not-0[OF \langle p + q \neq 0 \rangle]} \) unfolding \( \text{lc-def by auto} \)  
with \( \langle \text{lt } (p + q) = \text{lt } p \rangle \) show \( False \) by simp  
next  
assume \( \text{lt } p \prec_t \text{lt } (p + q) \)  
have \( \text{lt } p \prec_t \text{lt } q \) by (rule \text{lt-plus-lessE}, \text{fact+})  
hence \( \text{lt } p \neq \text{lt } q \) by simp  
with \( \text{lt-eq show } False \) by simp  
qed
corollary \ltplus-lessI:
fixes p q :: 't \Rightarrow 'b::group_add
assumes p + q \neq 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 ltplus-lessI')
from assms(3) show lc p + lc q = 0 by simp
qed

lemma \ltplus-distinct-eq-max:
assumes lt p \neq 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_\tau 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.absorb2)
finally show \?thesis.
next
assume a: lt q \prec_\tau 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.absorb1)
finally show \?thesis.
next
assume lt p = lt q
with assms show \?thesis ..
qed

lemma \ltplus-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_\tau u
hence lt p \prec_\tau u and lt q \prec_\tau 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

qed
lemma `lt-minus-eqI`: \(\mathsf{lt} \ p \prec \ t \Rightarrow (\mathsf{lt} \ (p - q) = \mathsf{lt} \ q)\) for \(p, q :: 't \Rightarrow 'b::ab-group-add\)
by (metis `lt-plus-eqI-2` `lt-uminus uminus-add-conv-diff`)

lemma `lt-minus-eqI-2`: \(\mathsf{lt} \ q \prec \ t \Rightarrow (\mathsf{lt} \ (p - q) = \mathsf{lt} \ p)\) for \(p, q :: 't \Rightarrow 'b::ab-group-add\)
by (metis `lt-minus-eqI` `lt-uminus minus-diff-eq`)

lemma `lt-minus-eqI-3`:
assumes \(\mathsf{lt} \ q = \mathsf{lt} \ p\) and \(\mathsf{lc} \ q \neq \mathsf{lc} \ p\)
shows \(\mathsf{lt} \ (p - q) = \mathsf{ord-term-lin}.\max (\mathsf{lt} \ p) (\mathsf{lt} \ q)\)
proof (rule `ord-term-lin`.linorder-cases)
  assume \(a :: \mathsf{lt} \ p \prec \ t \Rightarrow \mathsf{lt} \ q\)
  hence \(\mathsf{lt} \ (p - q) = \mathsf{lt} \ q\) by (rule `lt-minus-eqI`)
  also from \(a\) have \(\ldots = \mathsf{ord-term-lin}.\max (\mathsf{lt} \ p) (\mathsf{lt} \ q)\)
    by (simp add: `ord-term-lin`.max.absorb2)
  finally show \(?thesis\).
next
  assume \(a :: \mathsf{lt} \ q \prec \ t \Rightarrow \mathsf{lt} \ p\)
  hence \(\mathsf{lt} \ (p - q) = \mathsf{lt} \ p\) by (rule `lt-minus-eqI-2`)
  also from \(a\) have \(\ldots = \mathsf{ord-term-lin}.\max (\mathsf{lt} \ p) (\mathsf{lt} \ q)\)
    by (simp add: `ord-term-lin`.max.absorb1)
  finally show \(?thesis\).
next
assume \(\mathsf{lt} \ p = \mathsf{lt} \ q\)
with \(\text{assms}\) show \(?thesis\).
qed
lemma \( \text{lt-minus-lessE} \): \( \text{lt} p \prec_t \text{lt} (p - q) \implies \text{lt} p \prec_t \text{lt} q \) for \( p \neq q \)

using \( \text{lt-minus-eqI-2} \) by fastforce

lemma \( \text{lt-minus-lessE-2} \): \( \text{lt} q \prec_t \text{lt} (p - q) \implies \text{lt} q \prec_t \text{lt} p \) for \( p \neq q \)

using \( \text{lt-plus-eqI-2} \) by fastforce

lemma \( \text{lt-minus-lessI} \): \( p - q \neq 0 \implies \text{lt} q \prec_t \text{lt} p \)

proof (rule \( \text{lt-max} \))

from assms show \( \text{lookup} p v \neq 0 \) by (simp add: in-keys-iff)

qed

lemma \( \text{lt-eqI-keys} \):

assumes \( v \in \text{keys} p \)

shows \( v \preceq_t \text{lt} p \)

proof (rule \( \text{lt-eqI} \), simp-all only: in-keys-iff[symmetric], fact+)

lemma \( \text{lt-gr-keys} \):

assumes \( u \in \text{keys} p \) and \( v \prec_t u \)

shows \( v \prec_t \text{lt} p \)

proof (rule \( \text{lt-gr} \))

from assms(1) show \( \text{lookup} p u \neq 0 \) by (simp add: in-keys-iff)

qed fact

lemma \( \text{lt-plus-eq-maxI} \):

assumes \( \text{lt} p = \text{lt} q = \text{lc} p = \text{lc} q = 0 \implies \text{lt} (p + q) \)

proof (cases \( \text{lt} p = \text{lt} q \))

case True

show \( \text{thesis} \)

proof (rule \( \text{lt-eqI-keys} \))

from True have \( \text{lc} p = \text{lc} q = 0 \) by (rule assms)

thus \( \text{ord-term-lin.max} (\text{lt} p) (\text{lt} q) \) by (simp add: in-keys-iff lc-def lookup-add True)

next

fix \( u \)

assume \( u \in \text{keys} (p + q) \)

hence \( u \preceq_t \text{lt} (p + q) \) by (rule \( \text{lt-max-keys} \))

also have \( \ldots \preceq_t \text{ord-term-lin.max} (\text{lt} p) (\text{lt} q) \) by (fact \( \text{lt-plus-le-max} \))
finally show \( u \preceq \text{ord-term-lin.max} \ (lt \ p) \ (lt \ q) \).

qed

next
case False
thus ?thesis by (rule lt-plus-distinct-eq-max)

qed

lemma lt-monom-mult:
assumes \( c \neq (0 :: 'b::{semiring-no-zero-divisors}) \) and \( p \neq 0 \)
shows \( lt \ (\text{monom-mult} \ c \ t \ p) = t \oplus \lt \ p \)

proof (intro lt-eqI)
from assms(1) show \( \text{lookup} \ (\text{monom-mult} \ c \ t \ p) \ (t \oplus \lt \ p) \neq 0 \)

proof (simp add: lookup-monom-mult-plus)
  show \( \text{lookup} \ p \ (\lt \ p) = 0 \)

using assms(2) lt-in-keys by auto

qed

next
fix \( u :: 't \)
assume \( \text{lookup} \ (\text{monom-mult} \ c \ t \ p) \ u = 0 \)
hence \( u \in \text{keys} \ (\text{monom-mult} \ c \ t \ p) \)
  by (simp add: in-keys-iff)
also have \( u \subseteq (\oplus) \ t \cdot \text{keys} \ p \)
finally obtain \( v \)
where \( v \in \text{keys} \ p \)
and \( u = t \oplus \lt p \)

proof (rule splus-mono)
  from \( v \in \text{keys} \ p \)
  show \( v \preceq \lt p \)

by (rule lt-max-keys)

qed

qed

lemma lt-monom-mult-zero:
assumes \( c \neq (0 :: 'b::{semiring-no-zero-divisors}) \)
shows \( lt \ (\text{monom-mult} \ c \ 0 \ p) = \lt \ p \)

proof (cases \( p = 0 \))
case True
show ?thesis by simp

next
case False
with assms show ?thesis by (simp add: lt-monom-mult term-simps)

qed

corollary lt-map-scale: \( c \neq (0 :: 'b::{semiring-no-zero-divisors}) \implies lt \ (c \cdot \lt p) = \lt \ p \)
by (simp add: map-scale-eq-monom-mult lt-monom-mult-zero)

lemma lc-monom-mult [simp]: \( lc \ (\text{monom-mult} \ c \ t \ p) = (c :: 'b::{semiring-no-zero-divisors}) \)
\* \( lc \ p \)

proof (cases \( c = 0 \))
case True
thus ?thesis by simp

next
case False

171
show \( ?\text{thesis} \)
proof (cases \( p = 0 \))
  case True
  thus \( ?\text{thesis} \) by simp
next
case False
  with \( \langle \ c \neq 0 \ \rangle \) show \( ?\text{thesis} \) by (simp add: lc-def lt-monom-mult lookup-monom-mult-plus)
qed

corollary \( \text{lc-map-scale [simp]} : \text{lc (c \cdot p)} = (c::'b::semiring-no-zero-divisors) \ast \text{lc p} \)
by (simp add: map-scale-eq-monom-mult)

lemma (in ordered-term) \( \text{lt-mult-scalar-monomial-right} \):
  assumes \( c \neq (0::'b::semiring-no-zero-divisors) \) and \( p \neq 0 \)
  shows \( \text{lt (p \circ monomial c v)} = \text{punit.lt p \oplus v} \)
proof (intro lt-eqI)
  from assms(1) show \( \text{lookup (p \circ monomial c v)} (\text{punit.lt p \oplus v}) \neq 0 \)
  proof (simp add: lookup-mult-scalar-monomial-right-plus)
    from assms(2) show \( \text{lookup p (punit.lt p)} \neq 0 \)
    using in-keys-iff punit.lt-in-keys by fastforce
  qed
next
fix \( u::'t \)
  assume \( \text{lookup (p \circ monomial c v)} u \neq 0 \)
  hence \( u \in \text{keys (p \circ monomial c v)} \) by (simp add: in-keys-iff)
  also have \( \ldots \subseteq (\lambda t. t \oplus v) \cdot \text{keys p} \) by (fact keys-mult-scalar-monomial-right-subset)
  finally obtain \( t \) where \( t \in \text{keys p} \) and \( u = t \oplus v \).
  show \( u \preceq_1 \text{punit.lt p \oplus v} \) unfolding (\( u = t \oplus v \))
  proof (rule splus-mono-left)
    from \( t \in \text{keys p} \) show \( t \preceq \text{punit.lt p} \) by (rule punit.lt-max-keys)
  qed
qed

lemma \( \text{lc-mult-scalar-monomial-right} \):
  \( \text{lc (p \circ monomial c v)} = \text{punit.lc p \ast (c::'b::semiring-no-zero-divisors)} \)
proof (cases \( c = 0 \))
  case True
  thus \( ?\text{thesis} \) by simp
next
case False
  show \( ?\text{thesis} \)
  proof (cases \( p = 0 \))
    case True
    thus \( ?\text{thesis} \) by simp
next
case False
  with \( \langle \ c \neq 0 \ \rangle \) show \( ?\text{thesis} \)
  by (simp add: punit.lc-def lc-def lt-mult-scalar-monomial-right lookup-mult-scalar-monomial-right-plus)

172
lemma lookup-monom-mult-eq-zero:
assumes s ⊕ lt p ≺ 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 ∈ keys (monom-mult c t p)
show v ⪯ t t ⊕ lt p
proof −
from keys-monom-mult-subset assms have v ∈ (⊔) t ' (keys p) ..
then obtain u where u ∈ keys p and v = t ⊕ u ..
from (u ∈ keys p) have u ⪯ t lt p by (rule lt-max-keys)
thus v ⪯ t t ⊕ lt p unfolding ⟨v = t ⊕ u⟩ by (rule splus-mono)
qed

lemma lt-monom-mult-le:
l t (monom-mult c t p) ⪯ t t ⊕ 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 ≠ 0 and (p::'t ⇒ 'b::semiring-no-zero-divisors) ≠ 0
shows t1 = t2
proof −
from asms(1) have lt (monom-mult c t1 p) = lt (monom-mult c t2 p) by simp
with (c ≠ 0) (p ≠ 0) have t1 ⊕ lt p = t2 ⊕ 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 ≠ 0
shows tt p = ord-term-lin.Min (keys p)
using assms unfolding tt-def by simp

lemma tt-min-keys:
assumes v ∈ keys p
shows tt p ⪯ t v
proof −
from assms have keys p ≠ {} by auto
hence p ≠ 0 by simp
qed

lemma tt-min:
  assumes lookup p v ≠ 0
  shows tt p ≤ₜ v
proof –
  from assms have v ∈ keys p unfolding keys-def by simp
  thus ?thesis by (rule tt-min-keys)
qed

lemma tt-in-keys:
  assumes p ≠ 0
  shows tt p ∈ 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 ∈ keys p and u ∈ keys p =⇒ v ≤ₜ u
  shows tt p = v
proof –
  from assms(1) have keys p ≠ {} by auto
  hence p ≠ 0 by simp
  from assms(1) have tt p ≤ₜ v by (rule tt-min-keys)
  moreover have v ≤ₜ tt p by (rule assms(2), rule tt-in-keys, fact ⟨p ≠ 0⟩)
  ultimately show ?thesis by simp
qed

lemma tt-gr:
  assumes u ∈ keys p and u ≺ₜ v
  shows tt p ≺ₜ v
proof –
  from ⟨u ∈ keys p⟩ have tt p ≺ₜ v by fact
data show ?thesis by (rule assms(1), rule tt-in-keys, fact ⟨p ≠ 0⟩)
qed

lemma tt-less:
  assumes u ∈ keys p and u ⇐ₜ v
  shows tt p ⇐ₜ v
proof –
  from ⟨u ∈ keys p⟩ have tt p ≦ₜ u by (rule tt-min-keys)
  also have ... ≻ₜ v by fact
  finally show tt p ≺ₜ v .
qed

lemma tt-ge:
assumes $\bigwedge u. u \prec_t v \implies \text{lookup p u} = 0$ and $p \neq 0$
shows $v \preceq_t t \top p$
proof
from $\langle p \neq 0 \rangle$ have $\text{keys p} \neq \{\}$ by simp
have $\forall u \in \text{keys p}. \ v \preceq_t u$
proof
fix $u :: t$
assume $u \in \text{keys p}$
hence $\text{lookup p u} \neq 0$ unfolding keys-def by simp
hence $\neg u \prec_t v$ using assms $(1)[u]$ by auto
thus $v \preceq_t u$ by simp
qed
with tt-alt $\langle \text{OF} \langle p \neq 0 \rangle \rangle$ ord-term-lin. Min-ge-iff $\langle \text{OF finite-keys}[\text{OF p} \langle \text{keys p} \neq \{\} \rangle] \rangle$
show ?thesis by simp
qed

lemma tt-ge-keys:
assumes $\bigwedge u. u \in \text{keys p} \implies v \preceq_t u$ and $p \neq 0$
shows $v \preceq_t t \top p$
by (rule assms $(1)$, rule tt-in-keys, fact)

lemma tt-ge-iff: $v \preceq_t t \top p \iff ((p \neq 0 \lor v = \text{min-term}) \land (\forall u. u \prec_t v \implies \text{lookup p u} = 0))$
(is $?L \iff (\text{?A} \land \text{?B})$)
proof
assume $?L$
show $?A \land ?B$
proof (intro conjI allI impI)
show $p \neq 0 \lor v = \text{min-term}$
proof (cases $p = 0$
  case True
  show ?thesis
  proof (rule disjI2)
    from $?L$ True have $v \preceq_t \text{min-term}$ by (simp add: tt-def)
    with min-term-min[of v] show $v = \text{min-term}$ by simp
  qed
next
  case False
  thus ?thesis ..
  qed
next
  fix $u$
  assume $u \prec_t v$
  also note $\langle v \preceq_t t \top p \rangle$
  finally have $u \prec_t t \top p$.
  hence $\neg t \top p \preceq_t u$ by simp
  with tt-min[of p u] show lookup p u = 0 by blast
  qed
next
  assume ?A ∧ ?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 ≠ 0
  shows tc p ≠ 0
unfolding tc-def in-keys-iff[symmetric] using assms by (rule tt-in-keys)

lemma tt-monomial:
  assumes c ≠ 0
  shows tt (monomial c v) = v
proof (rule tt-eqI)
  from keys-of-monomial[OF assms, of v] show v ∈ keys (monomial c v) by simp
next
  fix u
  assume u ∈ keys (monomial c v)
  with keys-of-monomial[OF assms, of v] have u = v by simp
  thus v ⪯ₕ 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 ≠ 0 and tt p ≺ₕ tt q
  shows tt (p + q) = tt p
proof (intro tt-eqI)
  from tt-less[of tt p q tt q] ⟨tt p ≺ₕ tt q⟩ have tt p ≠ keys q by blast
  with lookup-add[of p q tt p] tc-not-0[OF p ≠ 0] show tt p ∈ keys (p + q)
    unfolding in-keys-iff tc-def by simp
next
  fix u

176
assume \( u \in \text{keys}(p + q) \)
show \( tt\ p \preceq_t u \)
proof (rule ccontr)
  assume \( \neg tt\ p \preceq_t u \)
  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 \( u \notin\ \text{keys} q \) by blast
  moreover from sp \( tt\)-less[of \( u\ p\ tt\ p \)] have \( u \notin\ \text{keys} p \) by blast
  ultimately show False using \( u \in \text{keys}(p + q)\) Poly-Mapping.keys-add[of \( p\ q \)] by auto
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 \( \neg tt\ q \prec_t tt\ p \)
    hence \( tt\ p = tt\ q \lor tt\ p \prec_t tt\ q \) by auto
    thus False
  proof
    assume \( tt\)-eq: \( tt\ p = tt\ q \)
    have \( tt\ p \succeq_t tt\ (p + q) \)
    proof (rule \( tt\)-ge-keys)
      fix \( u \)
      assume \( u \in \text{keys}(p + q) \)
      hence \( u \in \text{keys} p \cup \text{keys} q \)
      proof
        show \( \text{keys}(p + q) \subseteq\ \text{keys} p \cup \text{keys} q \) by (fact Poly-Mapping.keys-add)
        qed
        thus \( tt\ p \preceq_t u \)
        proof
          assume \( u \in \text{keys} p \)
          thus \( ?thesis \) by (rule \( tt\)-min-keys)
        next
          assume \( u \in \text{keys} q \)
          thus \( ?thesis \) unfolding \( tt\)-eq by (rule \( tt\)-min-keys)
          qed
        qed (fact \( p + q \neq 0 \))
        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

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 ≺ tt (p + q)
proof (rule ccontr)
  assume ¬ tt p ≺ tt (p + q)
  hence tt p = tt (p + q) ∨ tt (p + q) ≺ 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) ≺ tt p
  have tt q ≺ tt p by (rule tt-plus-lessE, fact+)
  hence tt q ≠ tt p by simp
    with tt-eq show False by simp
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 ≺ tt 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 ∈ keys (monom-mult c t p)
  shows t ⊕ tt p ≥ₜ v
proof –
  from keys-monom-mult-subset assms have v ∈ (⊕) t · (keys p) ..
  then obtain u where u ∈ keys p and v = t ⊕ u ..
  from ⟨u ∈ keys p⟩ have tt p ≥ₜ u by (rule tt-min-keys)
  thus t ⊕ tt p ≥ₜ v unfolding ⟨v = t ⊕ u⟩ by (rule splus-mono)
qed

lemma lt-ge-tt:
  tt p ≥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 ≠ 0 and p: p = monomial c v by (rule is-monomial-monomial)
  from ⟨c ≠ 0⟩ have lt p = v and tt p = v unfolding p by (rule lt-monomial,
10.5 higher and lower

lemma lookup-higher: lookup (higher p u) v = (if u ≺ 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 ≺ 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 ≺ t u → lookup p u = lookup q u) (is ?L ←→ ?R)
proof
  assume ?L
  show ?R
  proof (intro allI impI)
    fix u
    assume v ≺ 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 ≺ 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 ≺ t u → lookup p u = 0)
proof
  have higher p v = higher 0 v ←→ (∀ u. v ≺ 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 \in keys p. v \prec 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-when: lookup (lower p u) v = (lookup p v when v \prec u)
  by (auto simp add: lookup-minus when-def)

lemma lower-eq-iff: lower p v = lower q v ←→ (∀ u. u \prec v → lookup p u = lookup q u) (is ?L ↔ ?R)
proof
  assume ?L
  show ?R
  proof (intro allI impl)
    fix u
    assume u \prec v
    moreover from ⟨?L⟩ have lookup (lower p u) = lookup (lower q u) 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 \prec 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 ∈ keys p ∧ v ≺_t u
  hence u ∈ keys p ..
  thus u ≼_t lt p by (rule lt-max-keys)
qed fact

lemma lc-higher:
  assumes v ≺_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 ←→ lt p ≼_t v
  by (simp add: higher-eq-zero-iff lt-le-iff)

lemma higher-id-iff: higher p v = p ←→ (p = 0 ∨ v ≺_t tt p) (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 tt-gr)
      fix u
assume \( u \in \text{keys } p \)

hence lookup \( p u \neq 0 \) by (simp add: in-keys-iff)

from \(?L\) have lookup (higher \( p v \)) \( u = \text{lookup } p u \) by simp

hence lookup \( p u = (\text{if } v \prec_t u \text{ then lookup } p u \text{ else } 0) \) by (simp only: lookup-higher)

hence \( \neg v \prec_t u \implies \text{lookup } p u = 0 \) by simp

with (lookup \( p u \neq 0 \)) show \( v \prec_t u \) by auto

qed

qed

lemma tt-lower:

assumes \( \text{tt } p \prec_t t \ v \)

shows \( \text{tt (lower } p v) = \text{tt } p \)

proof (cases \( p = 0 \))

case True

thus \(?thesis\) by simp

next

case False

with \(?R\): have \( v \prec_t \text{tt } p \) by simp

show \(?thesis\)

proof (rule tt-eqI, simp-all add: keys-lower, intro impI)

fix \( u \)

assume \( v \prec_t u \)

hence \( u \preceq_t v \) by simp

from this (\( v \prec_t \text{tt } p \)) have \( u \prec_t \text{tt } p \) by simp

hence \( \neg \text{tt } p \preceq_t u \) by simp

with \( \text{tt-min}[\text{of } p u] \) show \( \text{lookup } p u = 0 \) by blast

qed

qed

lemma tc-lower:

assumes \( \text{tt } p \prec_t t \ v \)

proof (rule tt-eqI, simp-all add: keys-lower, rule, rule tt-in-keys)

fix \( u \)

assume \( u \in \text{keys } p \land u \prec_t v \)

hence \( u \in \text{keys } p \) ..

thus \( \text{tt } p \preceq_t u \) by (rule tt-min-keys)

qed

lemma tc-lower:

assumes \( \text{tt } p \prec_t t \ v \)
shows \( tc (\text{lower} \ p \ v) = tc \ p \)
by \((\text{simp add: tc-def}\ tt\-lower\ \text{assms lookup-lower})\)

\textbf{lemma} \( lt\-lower\): \( lt (\text{lower} \ p \ v) \preceq_t lt \ p \)
\textbf{proof} \((\text{cases} \ \text{lower} \ p \ v = 0)\)
  \textbf{case} True
  \textbf{thus} \(?\text{thesis}\) by \((\text{simp add: lt-def min-term-min})\)
\textbf{next}
  \textbf{case} False
  \textbf{show} \(?\text{thesis}\)
  \textbf{proof} \((\text{rule lt-le},\ \text{simp add: lookup-lower},\ \text{rule impI},\ \text{rule ccontr})\)
    \textbf{fix} \(u\)
    \textbf{assume} lookup \(p\ u \neq 0\)
    \textbf{hence} \(u \preceq_t lt \ p\) by \((\text{rule lt-max})\)
    moreover \textbf{assume} \(lt \ p \prec_t u\)
    \textbf{ultimately show} \(\text{False}\) by \(\text{simp}\)
  \textbf{qed}
\textbf{qed}

\textbf{lemma} \( lt\-lower\-less\):
\textbf{assumes} \(\text{lower} \ p \ v \neq 0\)
\textbf{shows} \( lt (\text{lower} \ p \ v) \prec_t v\)
\textbf{using} \(\text{assms}\)
\textbf{proof} \((\text{rule lt-less})\)
  \textbf{fix} \(u\)
  \textbf{assume} \(v \preceq_t u\)
  \textbf{thus} \(\text{lookup} (\text{lower} \ p \ v) \ u = 0\) by \((\text{simp add: lookup-lower-when})\)
\textbf{qed}

\textbf{lemma} \( lt\-lower\-eq\-iff\): \( lt (\text{lower} \ p \ v) = lt \ p \iff (lt \ p = \text{min-term} \lor lt \ p \prec_t v)\) \((\text{is} \ \ ?L \iff ?R)\)
\textbf{proof}
  \textbf{assume} \(\neg ?L\)
  \textbf{show} \(\neg ?R\)
  \textbf{proof} \((\text{rule ccontr},\ \text{simp},\ \text{elim conjE})\)
    \textbf{assume} \(lt \ p \neq \text{min-term}\)
    \textbf{hence} \(\text{min-term} \prec_t lt \ p\) \textbf{using} \text{min-term-min ord-term-lin.dual-order.not-eq-order-implies-strict}
    by \(\text{blast}\)
    \textbf{assume} \(\neg \neg \text{lt} \ p \prec_t v\)
    \textbf{hence} \(v \preceq_t \text{lt} \ p\) by \(\text{simp}\)
    \textbf{have} \(lt (\text{lower} \ p \ v) \prec_t \text{lt} \ p\)
    \textbf{proof} \((\text{cases} \ \text{lower} \ p \ v = 0)\)
      \textbf{case} True
      \textbf{thus} \(?\text{thesis}\) \textbf{using} \(\text{min-term} \prec_t \text{lt} \ p\) \textbf{by} \((\text{simp add: lt-def})\)
    \textbf{next}
      \textbf{case} False
      \textbf{show} \(?\text{thesis}\)
      \textbf{proof} \((\text{rule lt-less})\)
      \textbf{fix} \(u\)

184
assume \( \preceq_p \ u \)

with \( \langle v \preceq_p \ l t \ p \rangle \) have \( \neg \ u \prec t \ v \) by simp

gd lookup (lower p v) \( u = 0 \) by (simp add: lookup-lower)

qed

fact

qed

next

assume \( ?R \)

show \( ?L \)

proof
  (cases \( l t \ p = \min \text{-term} \))
  case True
  hence \( l t \ p \preceq t \ (l t \ (l t \ p \ v)) \) by (simp add: \( \min \text{-term-min} \))
  with \( l t \)-lower[\( o f \ p \ v \)] show \( ?\text{thesis} \) by simp

next
  case False
  with \( \langle ?R \rangle \) have \( l t \ p \prec t \ v \) by simp
  show \( ?\text{thesis} \)
    proof
      (rule \( l t \)-eqI-keys, simp-all add: keys-lower, rule \( \text{conjI} \), rule \( l t \)-in-keys, rule)
      assume \( p = 0 \)
      hence \( l t \ p = \min \text{-term} \) by (simp add: \( l t \)-def)
      with False show False ..

next
  fix \( u \)
  assume \( u \in \text{keys} p \land u \prec t \ v \)
  hence \( u \in \text{keys} p \) ..
  thus \( u \preceq t l t \ p \) by (rule \( l t \)-max-keys)

qed

fact

qed

lemma \( \text{tt-higher} \)

assumes \( v \prec t \ l t \ p \)

shows \( \text{tt} p \preceq t \ (h \text{igher} p v) \)

proof
  (rule \( \text{tt-ge-keys} \), simp add: \( \text{keys-higher} \))
  fix \( u \)
  assume \( u \in \text{keys} p \land v \prec t u \)
  hence \( u \in \text{keys} p \) ..
  thus \( \text{tt} p \preceq t u \) by (rule \( \text{tt-max-keys} \))

next
  show \( \text{higher} p v \neq 0 \)
  proof
    (simp add: \( \text{higher-eq-zero-iff} \), intro \( \text{exI} \ \text{conjI} \))
    have \( p \neq 0 \)
    proof
      assume \( p = 0 \)
      hence \( l t \ p \preceq t v \) by (simp add: \( l t \)-def min \( \text{-term-min} \))
      with assms show False by simp
    qed
thus lookup p (lt p) ≠ 0 using lt-in-keys by auto

qed fact

lemma tt-higher-eq-iff:

tt (higher p v) = tt p<-> ((lt p ≤₄ v ∧ tt p = min-term) ∨ v ≺₄ tt p) (is ?L <- ?R)

proof
assume ?L
show ?R
proof (rule ccontr, simp, elim conjE)
assume a: lt p ≤₄ v ----> tt p ≠ min-term
assume ~ v ≺₄ tt p
hence tt p ≤₄ v by simp
have tt p ≺₄ 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 ≺₄ lt p by auto
    have lt p ≠ min-term
      proof
        assume lt p = min-term
        with ⟨v ≺₄ lt p⟩ show False using min-term-min[of v] by auto
      qed
    hence p ≠ 0 by (auto simp add: lt-def)
    from ⟨v ≺₄ 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 ≺₄ u by (simp add: keys-higher)
      with ⟨tt p ≤₄ v⟩ show tt p ≺₄ u by simp
    qed
    qed
  qed
with ⟨?L⟩ show False by simp

next
assume ?R
show ?L
proof (cases lt p ≤₄ v ∧ tt p = min-term)
  case True
  hence lt p ≤₄ v and tt p = min-term by simp-all
  from ⟨lt p ≤₄ v⟩ have higher p v ≠ 0 by (simp add: higher-eq-zero-iff)
  with ⟨tt p = min-term⟩ show ?thesis by (simp add: lt-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

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 ≼_t u
      from ⟨?L⟩ have lookup (lower p v) u = lookup p u by simp
      hence lookup p u = (if u ≺_t v then lookup p u else 0) by (simp only: lookup-lower)
      hence v ≼_t u ⟹ lookup p u = 0 by (meson ord-term-lin.leD)
      with ⟨v ≼_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 ≺_t v by simp
    show ?thesis
    proof (rule poly-mapping-eqI, simp add: lookup-lower, intro impI)
      fix u
      assume ¬ u ≺_t v

  187
hence \( v \preceq_t u \) by simp
with \( \text{lt } p \prec_t v \) have \( \text{lt } p \prec_t u \) by simp
hence \( \neg u \preceq_t \text{lt } p \) by simp
with \( \text{lt-max[of } p u \) show \( \text{lookup } p u = 0 \) by blast
qed
qed
qed

lemma lower-higher-commute: higher \((\text{lower } p s) t = \text{lower } (\text{higher } p t) s \)
by \((\text{rule poly-mapping-eqI, } \text{simp add: lookup-higher lookup-lower})\)

lemma lt-lower-higher:
assumes \( v \prec_t \text{lt } (\text{lower } p u) \)
shows \( \text{lt } (\text{lower } (\text{higher } p v) u) = \text{lt } (\text{lower } p u) \)
by \((\text{simp add: lower-higher-commute[symmetric] lt-higher[OF assms]}\)

lemma lc-lower-higher:
assumes \( v \prec_t \text{lt } (\text{lower } p u) \)
shows \( \text{lc } (\text{lower } (\text{higher } p v) u) = \text{lc } (\text{lower } p u) \)
using assms by \((\text{simp add: lc-def lt-lower-higher lookup-lower lookup-higher})\)

lemma trailing-monomial-higher:
assumes \( p \neq 0 \)
shows \( p = (\text{higher } p (\text{tt } p)) + \text{monomial } (\text{tc } p) (\text{tt } p) \)
proof \((\text{rule poly-mapping-eqI, } \text{simp only: lookup-add})\)
fix \( v \)
show \( \text{lookup } p v = \text{lookup } (\text{higher } p (\text{tt } p)) v + \text{lookup } (\text{monomial } (\text{tc } p) (\text{tt } p)) v \)
proof \((\text{cases tt } p \preceq_t v)\)
  case True
  show \(?thesis\)
  proof \((\text{cases } v = \text{tt } p)\)
    assume \( v = \text{tt } p \)
    hence \( \neg \text{tt } p \prec_t v \) by simp
    hence \( \text{lookup } (\text{higher } p (\text{tt } p)) v = 0 \) by \((\text{simp add: lookup-higher})\)
    moreover from \( (v = \text{tt } p) \) have \( \text{lookup } (\text{monomial } (\text{tc } p) (\text{tt } p)) v = \text{tc } p \)
by \((\text{simp add: lookup-single})\)
    moreover from \( (v = \text{tt } p) \) have \( \text{lookup } p v = \text{tc } p \) by \((\text{simp add: tc-def})\)
    ultimately show \(?thesis\) by simp
next
  assume \( v \neq \text{tt } p \)
  from this True have \( \text{tt } p \prec_t v \) by simp
  hence \( \text{lookup } (\text{higher } p (\text{tt } p)) v = \text{lookup } p v \) by \((\text{simp add: lookup-higher})\)
  moreover from \( (v \neq \text{tt } p) \) have \( \text{lookup } (\text{monomial } (\text{tc } p) (\text{tt } p)) v = 0 \) by \((\text{simp add: lookup-single})\)
  ultimately show \(?thesis\) by simp
qed
next
  case False
hence \( v \prec_t t \) by simp
hence \( tt \neq v \) by simp
from False have \( \neg tt \prec v \) 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 \( \langle\langle tt p \neq v \rangle\rangle \) have lookup \( \operatorname{monomial}(tc p)(tt p) v = 0 \) by (simp add: lookup-single)
moreover from \( \langle\langle \neg tt p \prec t v \rangle\rangle \) have lookup \( \operatorname{higher}(p(tt p)) v = 0 \) by (simp add: lookup-higher)
ultimately show ?thesis by simp
qed

lemma higher-lower-decomp: \( \operatorname{higher}(p v) + \operatorname{monomial}(\operatorname{lookup}(p v)) v + \operatorname{lower}(p v) = p \)
proof (rule poly-mapping-eqI)
  fix \( u \)
  show \( \operatorname{lookup}(\operatorname{higher}(p v) + \operatorname{monomial}(\operatorname{lookup}(p v)) v + \operatorname{lower}(p v)) u = \operatorname{lookup}(p u) \)
  proof (rule ord-term-lin.l inorder-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: \( \operatorname{lookup}(\operatorname{tail}(p)) v = (\text{if } v \prec_t lt \text{ then } \operatorname{lookup}(p v) \text{ else } 0) \) by (simp add: lookup-lower tail-def)

lemma lookup-tail-when: \( \operatorname{lookup}(\operatorname{tail}(p)) v = (\operatorname{lookup}(p v \text{ when } v \prec_t lt \text{ p}) \text{ when } v \prec_t lt \text{ p}) \) by (simp add: lookup-lower-when tail-def)

lemma lookup-tail-2: \( \operatorname{lookup}(\operatorname{tail}(p)) v = (\text{if } v = lt \text{ p then } 0 \text{ else } \operatorname{lookup}(p v)) \)
proof (rule ord-term-lin.l inorder-cases[of v lt p])
  assume \( v \prec_t lt \text{ p} \)
  hence \( v \neq lt \text{ p} \) by simp
  from this \( v \prec_t lt \text{ p} \) lookup-tail[of p v] show ?thesis by simp
next
  assume \( v = lt \text{ p} \)
hence \( \neg v \prec_t \text{lt } p \) by simp

from \( v = \text{lt } p \) this lookup-tail[of p v] show \(?thesis\) by simp

next
assume \( \text{lt } p \prec_t v \)
hence \( \neg v \preceq_t \text{lt } p \) by simp
hence \( \text{cp}: \text{lookup } p \ v = 0 \)
  using \( \text{lt-max} \) by blast
from \( (\neg v \preceq_t \text{lt } p) \) have \( \neg v = \text{lt } p \) and \( \neg v \prec_t \text{lt } p \) by simp-all
thus \(?thesis\) using \( \text{cp} \) lookup-tail[of p v] by simp

qed

lemma leading-monomial-tail: \( p = \text{monomial } (\text{lc } p) (\text{lt } p) + \text{tail } p \) for \( p ::\Rightarrow_0 \)

proof (rule poly-mapping-eqI)

fix \( v \)

have \( \text{lookup } p \ v = \text{lookup } (\text{monomial } (\text{lc } p) (\text{lt } p)) \ v + \text{lookup } (\text{tail } p) \ v \)

proof (cases \( v \preceq_t \text{lt } p \))
case True
show \(?thesis\)
proof (cases \( v = \text{lt } p \))
assume \( v = \text{lt } p \)
hence \( \neg v \prec_t \text{lt } p \) by simp
hence \( \text{c3: lookup } (\text{tail } p) \ v = 0 \) unfolding lookup-tail[of p v] by simp
from \( (v = \text{lt } p) \) have \( \text{c2: lookup } (\text{monomial } (\text{lc } p) (\text{lt } p)) \ v = \text{lc } p \) by simp
from \( (v = \text{lt } p) \) have \( \text{c1: lookup } p \ v = \text{lc } p \) (simp add: lc-def)
from \( \text{c1 c2 c3} \) show \(?thesis\) by simp
next
assume \( v \neq \text{lt } p \)
from this True have \( v \prec_t \text{lt } p \) by simp
hence \( \text{c2: lookup } (\text{tail } p) \ v = \text{lookup } p \ v \) unfolding lookup-tail[of p v] by simp
from \( (v \neq \text{lt } p) \) have \( \text{c1: lookup } (\text{monomial } (\text{lc } p) (\text{lt } p)) \ v = 0 \) (simp add: lookup-single)
from \( \text{c1 c2} \) show \(?thesis\) by simp

qed

next

case False
hence \( \text{lt } p \prec_t v \) by simp
hence \( \text{lt } p \neq v \) by simp
from False have \( \neg v \prec_t \text{lt } p \) by simp
have \( \text{c1: lookup } p \ v = 0 \)
proof (rule ccontr)
assume \( \text{lookup } p \ v \neq 0 \)
from \( \text{lt-max}[OF this False] \) show \( \text{False} \) by simp
qed
from \( \text{lt } p \neq v \) have \( \text{c2: lookup } (\text{monomial } (\text{lc } p) (\text{lt } p)) \ v = 0 \) (simp add: lookup-single)
from \( (\neg v \prec_t \text{lt } p) \) lookup-tail[of p v] have \( \text{c3: lookup } (\text{tail } p) \ v = 0 \) by simp
from \( \text{c1 c2 c3} \) show \(?thesis\) by simp

190
thus \( \text{lookup } p \ v = \text{lookup } (\text{nominal } (lc \ p) \ (lt \ p) \ + \ \text{tail } p) \ v \) \ by \ (\text{simp add: lookup-add})

qed

lemma tail-alt: \( \text{tail } p = \text{except } p \ \{lt \ p\} \)
  \ by \ (\text{rule poly-mapping-eqI, simp add: lookup-tail-2 lookup-except})

corollary tail-alt-2: \( \text{tail } p = \text{p - nominal } (lc \ p) \ (lt \ p) \)
proof
  have \( p = \text{nominal } (lc \ p) \ (lt \ p) \ + \ \text{tail } p \) \ by \ (\text{fact leading-nominal-tail})
  also have \( \ldots = \text{tail } p + \text{nominal } (lc \ p) \ (lt \ p) \) \ by \ (\text{simp only: add.commute})
  finally have \( p - \text{nominal } (lc \ p) \ (lt \ p) = (\text{tail } p + \text{nominal } (lc \ p) \ (lt \ p)) - \text{nominal } (lc \ p) \ (lt \ p) \) \ by \ simp
  thus \(?thesis \) \ by \ simp
  qed

lemma tail-zero [simp]: \( \text{tail } 0 = 0 \)
  \ by \ (\text{simp only: tail-alt except-zero})

lemma lt-tail:
  assumes \( \text{tail } p \neq 0 \)
  shows \( \text{lt } (\text{tail } p) \prec_{\ell} \text{lt } p \)
proof \( (\text{intro lt-less}) \)
  fix \( u \)
  assume \( \text{lt } p \prec_{\ell} u \)
  hence \( u \prec_{\ell} \text{lt } p \) \ by \ simp
  thus \( \text{lookup } (\text{tail } p) \ u = 0 \) \ unfolding \( \text{lookup-tail[of } p \ u} \) \ by \ simp
  qed \ fact

lemma keys-tail: \( \text{keys } (\text{tail } p) = \text{keys } p - \{lt \ p\} \)
  \ by \ (\text{simp add: tail-alt keys-except})

lemma tail-monomial: \( \text{tail } (\text{nominal } c \ v) = 0 \)
  \ by \ (\text{metis (no-types, lifting) lookup-tail-2 lookup-single-not-eq lt-less lt-nominal ord-term-lin.dual-order.strict-implies-not-eq single-zero tail-zero})

lemma (in ordered-term) mult-scalar-tail-rec-left:
  \( p \odot q = \text{monom-mult } (\text{unit} \ p) \ (\text{unit} \ p) \ q + (\text{unit} \ p) \odot q \)
  unfolding \( \text{unit} \ p \text{-def} \text{unit} \ p \text{-alt} \) \ by \ (\text{fact mult-scalar-rec-left})

lemma mult-scalar-tail-rec-right: \( p \odot q = p \odot \text{nominal } (lc \ q) \ (lt \ q) + p \odot \text{tail } q \)
  unfolding \( \text{tail-alt lc-def} \) \ by \ (\text{rule mult-scalar-rec-right})

lemma lt-tail-max:
  assumes \( \text{tail } p \neq 0 \) \ and \( v \in \text{keys } p \) \ and \( v \prec_{\ell} \text{lt } p \)
  shows \( v \preceq_{\ell} \text{lt } (\text{tail } p) \)
proof \( (\text{rule lt-max-keys, simp add: keys-tail assms(2)}) \)

191
lemma keys-tail-less-lt:
  assumes v ∈ keys (tail p)
  shows v ≺ 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 ≺ 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 ≤ 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 ≺ 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) ≤ 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:
\[ \text{tail} \ (\text{monom-mult} \ c \ t \ p) = \text{monom-mult} \ (c::'b::semiring-no-zero-divisors) \ t \ (\text{tail} \ p) \]

\textbf{proof (cases } p = 0 \text{)}

\textbf{case } True

\textbf{hence} tail p = 0 \text{ and } monom-mult c t p = 0 \text{ by simp-all}

\textbf{thus } \text{?thesis by simp}

\textbf{next}

\textbf{case } False

\textbf{show } \text{?thesis}

\textbf{proof (cases } c = 0 \text{)}

\textbf{case } True

\textbf{hence} monom-mult c t p = 0 \text{ and } monom-mult c t (tail p) = 0 \text{ by simp-all}

\textbf{thus } \text{?thesis by simp}

\textbf{next}

\textbf{case } False

\textbf{let } ?a = monom-mult c t p

\textbf{let } ?b = monom-mult c t (tail p)

\textbf{from } (p \neq 0) \text{ False have } ?a \neq 0 \text{ by (simp add: monom-mult-eq-zero-iff)}

\textbf{from False } (p \neq 0) \text{ have } lt-a: \text{ lt } ?a = t \oplus lt p \text{ by (rule lt-monom-mult)}

\textbf{show } \text{?thesis}

\textbf{proof (rule poly-mapping-eqI, simp add: lookup-tail lt-a, intro conjI impI)}

\textbf{fix } u

\textbf{assume } u \prec t \ t \oplus lt p

\textbf{show } \text{lookup} \ (\text{monom-mult} \ c \ t \ p) \ u = \text{lookup} \ (\text{monom-mult} \ c \ t \ (\text{tail} \ p)) \ u

\textbf{proof (cases } t \text{ adds } p \text{ u)}

\textbf{case } True

\textbf{then obtain } v \text{ where } u = t \oplus v \text{ by (rule adds-ppE)}

\textbf{from } (u \prec t \ t \oplus lt p) \text{ have } v \prec t \ lt p \text{ unfolding } (u = t \oplus v) \text{ by (rule ord-term-strict-canc)}

\textbf{hence } \text{lookup} p \ v = \text{lookup} \ (\text{tail} \ p) \ v \text{ by (simp add: lookup-tail)}

\textbf{thus } \text{?thesis by (simp add: } (u = t \oplus v) \text{ lookup-monom-mult-plus) }

\textbf{next}

\textbf{case } False

\textbf{hence } \text{lookup } ?a \ u = 0 \text{ by (simp add: lookup-monom-mult)}

\textbf{moreover have } \text{lookup } ?b \ u = 0

\textbf{proof (rule ccontr, simp only: in-keys-iff[symmetric] keys-monom-mult[OF } (c \neq 0)])

\textbf{assume } u \in (\oplus) \ t \ t \text{ keys } (\text{tail} \ p)

\textbf{then obtain } v \text{ where } u = t \oplus v \text{ by auto}

\textbf{hence } t \text{ adds}_{p} u \text{ by (simp add: term-simps)}

\textbf{with False show False }..\textbf{ }

\textbf{qed}

\textbf{ultimately show } \text{?thesis by simp}

\textbf{qed}

\textbf{next}

\textbf{fix } u

\textbf{assume } \neg u \prec t \ t \oplus lt p

\textbf{hence } t \oplus lt p \preceq t, u \text{ by simp}

\textbf{show } \text{lookup} \ (\text{monom-mult} \ c \ t \ (\text{tail} \ p)) \ u = 0

193
proof (rule ccontr, simp only: in-keys-iff[symmetric] keys-monom-mult[OF False])
  assume \( u \in (\oplus) t \cdot \text{keys (tail p)} \)
  then obtain \( v \) where \( v \in \text{keys (tail p)} \) and \( u = t \oplus v \) by auto
  from \( t \oplus \text{lt p} \preceq v \) unfolding \((a = t \oplus v)\) by (rule ord-term-canc)
  from \( v \in \text{keys (tail p)} \) have \( v \in \text{keys p} \) and \( v \neq \text{lt p} \) by (simp-all add: keys-tail)
  with \( v = \text{lt p} \)
  show False ..

qed

lemma keys-plus-eq-lt-tt-D:
  assumes \( \text{keys (p + q)} = \{\text{lt p}, \text{tt q}\} \) and \( \text{lt q} \prec \text{lt p} \) and \( \text{tt q} \prec \text{tt} (p::\Rightarrow 0) \)
  shows tail p + higher q \((tt q)\) = 0
proof –
  note assms(3)
  also have ... \( \preceq_\tau \) lt p by (rule lt-ge-tt)
  finally have \( tt q \prec \text{lt p} \)
  hence \( q = 0 \) by simp
  have \( q = 0 \) by simp
  hence \( tc q \neq 0 \) by (rule tc-not-0)
  have \( p = \text{monomial (lc p)} \) (lt p) + tail p by (rule leading-monomial-tail)
  moreover from \( q \neq 0 \) have \( q = \text{higher q} (tt q) + \text{monomial (tc q)} (tt q) \) by (rule trailing-monomial-higher)
  ultimately have \( pq: p + q = (\text{monomial (lc p)} (lt p) + \text{monomial (tc q)} (tt q)) \)
  \((\text{is -} = (?m1 + ?m2) + ?b) \) by (simp add: algebra-simps)
  have \( \text{keys-m1: keys} \ ?m1 = \{\text{lt p}\} \)
  proof (rule keys-of-monomial, rule lc-not-0, rule)
    assume \( p = 0 \)
    with assms(2) have \( \text{lt q} \prec_\tau \text{min-term} \) by (simp add: lt-def)
    with \( \text{min-term-min [of lt q]} \) show False by simp
  qed
  moreover from \( tc q \neq 0 \) have \( \text{keys-m2: keys} \ ?m2 = \{\text{tt q}\} \) by (rule keys-of-monomial)
  ultimately have \( \text{keys-m1-m2: keys} (\text{?m1 + ?m2}) = \{\text{lt p, tt q}\} \)
using \( \langle \text{lt } p \neq \text{tt } q \rangle \) \text{keys-plus-eqI(of } ?m1 \ ?m2 \text{) by auto}
show \(?\text{thesis}\)
proof (rule ccontr)
  assume \(?b \neq 0\) by simp
  hence \(\text{keys } ?b \neq \{\}\) by simp
  then obtain \(t\) where \(t \in \text{keys } ?b\) by blast
  hence \(\text{t-in: } t \in \text{keys (tail } p \rangle \cup \text{keys (higher } q \langle \text{tt } q \rangle\rangle\) by blast
  hence \(t \neq \text{lt } p\)
proof (rule, simp add: \text{keys-tail, simp add: keys-higher, elim conjE})
  assume \(t \in \text{keys } q\)
  hence \(tt \text{p } \preceq t\) \text{t by (rule } \text{tt-max-keys)\}
  from this assms(2) show \(?\text{thesis}\) by simp
qed
moreover from \text{t-in have} \(t \neq \text{tt } q\)
proof (rule, simp add: \text{keys-tail, elim conjE})
  assume \(t \in \text{keys } p\)
  hence \(tt \text{p } \preceq t\) \text{t by (rule } \text{tt-min-keys)\}
  with assms(3) show \(?\text{thesis}\) by simp
next
  assume \(t \in \text{keys (higher } q \langle \text{tt } q \rangle\rangle\)
  thus \(?\text{thesis}\) by (auto simp only: \text{keys-higher})
qed
ultimately have \(t \notin \text{keys (} ?m1 \ + \ ?m2 \text{)}\) by (simp add: \text{keys-m1-m2})
moreover from \text{in-keys-plusI2[OF } ?t \in \text{keys } ?b \text{ this]} have \(t \in \text{keys (} ?m1 \ + \ ?m2 \text{)}\)
  by (simp only: \text{keys-m1-m2 pq[symmetric] assms(1)})
  ultimately show False ..
qed

10.7 Order Relation on Polynomials

definition \text{ord-strict-p} :: \(\langle t \Rightarrow_0 \ 'b::'a\rangle \Rightarrow \langle t \Rightarrow_0 \ 'b\rangle \Rightarrow \text{bool (infixl } \prec_p 50)\)\ where
  \(p \prec_p q \leftarrow (\exists u. \text{lookup } p = 0 \land \text{lookup } q = 0 \land (\forall u. \text{v } \prec_t u \rightarrow \text{lookup } p = \text{lookup } q u))\)

definition \text{ord-p} :: \(\langle t \Rightarrow_0 \ 'b::'a\rangle \Rightarrow \langle t \Rightarrow_0 \ 'b\rangle \Rightarrow \text{bool (infixl } \preceq_p 50)\)\ where
  \(\text{ord-p } p \ q \equiv \langle p \prec_p q \lor p = q \rangle\)

lemma \text{ord-strict-pI}\:
  assumes \text{lookup } p v = 0 \text{ and lookup } q v \neq 0 \text{ and } (\forall u. \text{v } \prec_t u \rightarrow \text{lookup } p u = \text{lookup } q u)
  shows \(p \prec_p q\)
  unfolding \text{ord-strict-p-def} using assms by blast

lemma \text{ord-strict-pE}\:
  assumes \(p \prec_p q\)
obtains $v$ where $\text{lookup } p \ v = 0$ and $\text{lookup } q \ v \neq 0$ and $\bigwedge u. \ v \prec_t u \Rightarrow 
\text{lookup } p \ u = \text{lookup } q \ u$
using assms unfolding ord-strict-p-def by blast

lemma not-ord-pI:
‘\text{assumes } \text{lookup } p \ v \neq \text{lookup } q \ v \text{ and } \text{lookup } p \ v \neq 0 \text{ and } \bigwedge u. \ v \prec_t u \Rightarrow 
\text{lookup } p \ u = \text{lookup } q \ u
\text{shows } \neg \ p \preceq_p q$
proof
assume $p \preceq_p q$
hence $p \prec_p q \lor p = q$ by (simp only: ord-p-def)
thus $\text{False}$
proof
assume $v' \prec_t v$
hence $\text{lookup } p \ v = \text{lookup } q \ v$ by (rule 3)
with assms(1) show $\text{thesis}$ ..
next
assume $v' = v$
with assms(2) 1 show $\text{thesis}$ by auto
qed
next
assume $p = q$
hence $\text{lookup } p \ v = \text{lookup } q \ v$ by simp
with assms(1) show $\text{thesis}$ ..
qed

qed

corollary not-ord-strict-pI:
\text{assumes } \text{lookup } p \ v \neq \text{lookup } q \ v \text{ and } \text{lookup } p \ v \neq 0 \text{ and } \bigwedge u. \ v \prec_t u \Rightarrow 
\text{lookup } p \ u = \text{lookup } q \ u
\text{shows } \neg \ p \prec_p q$
proof
from assms have $\neg \ p \preceq_p q$ by (rule not-ord-pI)
thus $\text{thesis}$ by (simp add: ord-p-def)
qed

lemma ord-strict-higher: $p \prec_p q \iff (\exists v. \text{lookup } p \ v = 0 \land \text{lookup } q \ v \neq 0 \land 
\text{higher } p \ v = \text{higher } q \ v)$
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 \land lookup q v1 \neq 0 \land (\forall u. v1 \prec_t u \rightarrow 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 \rightarrow lookup p u = lookup q u
  by auto
  show $\neg (\exists v. lookup q v = 0 \land lookup p v \neq 0 \land (\forall u. v \prec_t u \rightarrow lookup q u = lookup p u))$
  proof (intro notI, erule exE)
  fix v2 :: 't
  assume lookup q v2 = 0 \land lookup p v2 \neq 0 \land (\forall u. v2 \prec_t u \rightarrow 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 \rightarrow 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 \neq 0⟩ by simp
  next
  assume v2 \prec_t v1
  from v2[rule-format, OF this] ⟨lookup p v1 = 0⟩ ⟨lookup q v1 \neq 0⟩ show ?thesis by simp
  qed
  qed

lemma ord-strict-p-irreflexive: $\neg p \prec_p p$
unfolding ord-strict-p-def
proof (intro notI, erule exE)
  fix v :: 't
  assume lookup p v = 0 \land lookup q v = 0 \land (\forall u. v \prec_t u \rightarrow lookup p u = lookup q u)
  hence lookup p v = 0 and lookup p v \neq 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 ≺ₚ b) obtain v₁ where lookup a v₁ = 0
  and lookup b v₁ ≠ 0
  and v₁[rule-format]: (∀ u. v₁ ≺ᵣ u → lookup a u = lookup b u)

unfolding ord-strict-p-def by auto
from (b ≺ₚ c) obtain v₂ where lookup b v₂ = 0
  and lookup c v₂ ≠ 0
  and v₂[rule-format]: (∀ u. v₂ ≺ᵣ u → lookup b u = lookup c u)

unfolding ord-strict-p-def by auto
show a ≺ₚ c
proof (rule ord-term-lin.linorder-cases)
  assume v₁ ≺ᵣ v₂
  show ?thesis unfolding ord-strict-p-def
  proof
    show lookup a v₂ = 0 ∧ lookup c v₂ ≠ 0 ∧ (∀ u. v₂ ≺ᵣ u → lookup a u = lookup c u)
      proof (intro conjI allI impI)
        from ⟨lookup b v₂ = 0⟩ v₁[of v₁ ≺ᵣ v₂] show lookup a v₂ = 0 by simp
      next
      from ⟨lookup c v₂ ≠ 0⟩ show lookup c v₂ ≠ 0 .
    next
    fix u
    assume v₂ ≺ᵣ u
    from ord-term-lin.less-trans[OF ⟨v₂ ≺ᵣ v₁⟩ this] have v₁ ≺ᵣ u .
    from v₂[of v₂ ≺ᵣ u] v₁[of this] show lookup a u = lookup c u by simp
    qed
  qed
next
  assume v₂ ≺ᵣ v₁
  show ?thesis unfolding ord-strict-p-def
  proof
    show lookup a v₁ = 0 ∧ lookup c v₁ ≠ 0 ∧ (∀ u. v₁ ≺ᵣ 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 ≺ᵣ v₁] show lookup c v₁ ≠ 0 by simp
      next
      fix u
      assume v₁ ≺ᵣ u
      from ord-term-lin.less-trans[OF ⟨v₂ ≺ᵣ v₁⟩ this] have v₂ ≺ᵣ u .
      from v₁[of v₁ ≺ᵣ 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

198
Qed

Sublocale order ord-p ord-strict-p
Proof (intro order-strictI)
  Fix p q :: 't ⇒ 'b
  Show (p ≤_p q) = (p ~_p q ∨ p = q) unfolding ord-p-def ..
Next
  Fix p q :: 't ⇒ 'b
  Assume p ~_p q
  Thus ¬ q ~_p p by (rule ord-strict-p-asymmetric)
Next
  Fix a b c :: 't ⇒ 'b
  Assume a ~_p b and b ~_p c
  Thus a ~_p c by (rule ord-strict-p-transitive)
Qed

Lemma ord-p-zero-min: 0 ≤_p p
Proof (cases p = 0)
  Case True
  Thus (∃ v. lookup 0 v = 0 ∧ lookup p v ≠ 0 ∧ (∀ u. v ~_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 ~_t u → lookup 0 u = lookup p u)) ∨ 0 = p
  Proof
  Show (∃ v. lookup 0 v = 0 ∧ lookup p v ≠ 0 ∧ (∀ u. v ~_t u → lookup 0 u = lookup p u))
  Proof
  Show lookup 0 (lt p) = 0 ∧ lookup p (lt p) ≠ 0 ∧ (∀ u. (lt p) ~_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 ~_t u
  Hence u ~_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 ≺ t lt q
  shows p ≺ p q
proof
  have q ≠ 0
  proof
    assume q = 0
    with assms have lt p ≺ 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
    (rule ccontr)
    assume ¬ lt p ≺ t lt q
    hence lt q ≺ t lt p by simp
    from lt-ord-p[OF this] ⟨lt p ≺ 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 ≺ t u
  hence lt p ≺ t u using ⟨lt p ≺ 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 ≺ 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 ≺ 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 ≤ₚ q
  shows lt p ≤ₚ lt q
proof (rule ccontr)
  assume ¬ lt p ≤ₚ lt q
  hence lt q ≺ₚ lt p by simp
  from lt-ord-p[OF this] ⟨p ≤ₚ q⟩ show False by simp
  qed

lemma ord-p-tail:
assumes $p \neq 0$ and $lt\ p = lt\ q$ and $p \prec_p q$
shows $tail\ p \prec_p tail\ q$
using assms unfolding ord-strict-p-def
proof -
  assume $p \neq 0$ and $lt\ p = lt\ q$
  and $\exists v.\ lookup\ p\ v = 0 \land lookup\ q\ v \neq 0 \land (\forall u.\ v \prec_t u \longrightarrow lookup\ p\ u = lookup\ q\ u)$
  then obtain $v$ where $lookup\ p\ v = 0$
    and $lookup\ q\ v \neq 0$
    and $\exists v.\ v \prec_t t\ p \lor v = lt\ p$ by auto
  hence $v \prec_t lt\ p$
proof
  assume $v \prec_t lt\ p$
  thus $\neg thesis$ .
  next
  assume $v = lt\ p$
  thus $\neg thesis$ using lc-not-0 [OF $p \neq 0$] [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 $q \neq 0$
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] ($v \prec_t lt\ q$) [lookup p p = qt] by simp
show $\exists w.\ lookup\ (tail\ p)\ w = 0 \land lookup\ (tail\ q)\ w \neq 0 \land$
  $(\forall u.\ w \prec_t u \longrightarrow lookup\ (tail\ p)\ u = lookup\ (tail\ q)\ u)$
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 ($v \prec_t w$) 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 $p \neq 0$
  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 ⊆ₚ p
  shows (lookup q v = 0) ∧ (higher q v) = 0
using le unfolding ord-p-def
proof
  assume q ≺ₚ p
  thus ?thesis unfolding ord-strict-p-def
  proof
    fix w
    assume lookup q w = 0 ∧ lookup p w ≠ 0 ∧ (∀ u. w ∪ₚ u −→ lookup q u = lookup p u)
    hence qs: lookup q w = 0 and ps: lookup p w ≠ 0 and u: ∀ u. w ∪ₚ u −→ lookup q u = lookup p u
      by auto
    from hp have pu: ∀ u. v ∪ₚ u −→ lookup p u = 0 by (simp only: higher-eq-zero-iff)
    from pu[rule-format, of w] ps have ¬ v ∪ₚ w by auto
    hence w ≤ₚ v by simp
    hence w ∪ₚ v ∨ w = v by auto
    hence st: w ∪ₚ v by simp
    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 ∀ u. v ∪ₚ u −→ lookup q u = 0
    proof (intro allI, intro impl)
      fix u
      assume v ∪ₚ u
      from this st have w ∪ₚ u by simp
      from u[rule-format, OF this] pu[rule-format, OF ∪ₚ v ∪ₚ u] show lookup q u = 0 by simp
    qed
    qed
  qed
next
  assume q = p
  thus ?thesis using assms by simp

202
lemma ord-strict-p-recI:
  assumes lt p = lt q and lc p = lc q and tail: tail p ≺ p tail q
  shows p ≺ p q
proof -
  from tail obtain v where pt: lookup (tail p) v = 0
    and qt: lookup (tail q) v ≠ 0
    and a: ∀ u. v ≺ u u → lookup (tail p) u = lookup (tail q) u
    unfolding ord-strict-p-def by auto
  from qt lookup-zero[of v] have tail q ≠ 0 by auto
  from lt-max[OF qt] lt-tail[OF this] have v ≺ lt q by simp
  hence v ≺ 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 ≺ lt p⟩ pt show lookup p v = 0 by simp
  next
    from lookup-tail[of q v] ⟨v ≺ lt q⟩ qt show lookup q v ≠ 0 by simp
  next
    fix u
    assume v ≺ 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 ≺ u u → lookup p u = lookup q u
    unfolding ord-strict-p-def by auto
  from qt lookup-zero[of v] have tail q ≠ 0 by auto
  from lt-max[OF qt] lt-tail[OF this] have v ≺ lt q by simp
  hence v ≺ 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 ≺ lt p⟩ pt show lookup p v = 0 by simp
  next
    from lookup-tail[of q v] ⟨v ≺ lt q⟩ qt show lookup q v ≠ 0 by simp
  next
    fix u
    assume v ≺ 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
unfolding ord-strict-p-def by auto

show ?thesis

proof (cases v \prec_t lt p)
  case True
  from this a have lookup p (lt p) = lookup q (lt p) by simp
  thus ?thesis using \langle lt p = lt q \rangle unfolding lc-def by simp

next
  case False
  from this lt-max[OF qt] have v = lt p by simp
  from this lc-not-0[OF ⟨p \neq 0⟩] pt show ?thesis unfolding lc-def by auto

qed

lemma ord-strict-p-rec [code]:
  p \prec_p q =
  (q \neq 0 \land
   (p = 0 \lor
    (let v1 = lt p; v2 = lt q in
     (v1 \prec_t v2 \lor (v1 = v2 \land lookup p v1 = lookup q v2 \land lower p v1 \prec_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 \land lc p = lc q \land tail p \prec_p tail q) \lor lt p \prec_t lt q) \lor p = 0
  proof (intro disjCI)
    assume p \neq 0 and nl: \neg lt p \prec_t lt q
    from ⟨?L⟩ have p \leq_p q by simp
    from ord-p-lt[OF this] nl have lt p = lt q by simp
    show lt p = lt q \land lc p = lc q \land 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 \lor
  (let v1 = lt p; v2 = lt q in
   (v1 \prec_t v2 \lor v1 = v2 \land lookup p v1 = lookup q v2 \land lower p v1 \prec_p lower q v2)
  )

  unfolding lc-def tail-def by auto

qed

next
  assume ?R

  hence q \neq 0
  and dis: p = 0 \lor
    (let v1 = lt p; v2 = lt q in
     (v1 \prec_t v2 \lor v1 = v2 \land lookup p v1 = lookup q v2 \land 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 \( \langle p = 0 \rangle \) \( \langle q \neq 0 \rangle \) by simp

next

assume \( p \neq 0 \)

hence let \( v1 = \) \( lt p \); \( v2 = \) \( lt q \) in

\( \langle v1 \prec v1 \lor v1 = v2 \land \) lookup \( p v1 = \) lookup \( q v2 \land \) lower \( p v1 \prec \) lower \( q v2 \rangle \)

using dis by simp

hence \( lt p \prec \) \( lt q \lor \) \( lt p = \) \( lt q \land \) lc \( p = \) lc \( q \land \) tail \( p \prec_p \) tail \( q \)

unfolding lc-def tail-def by (simp add: Let-def)

thus \( ?\)thesis

proof

assume \( lt p \prec \) \( lt q \)

from \( \langle 1 \rangle \) of this show \( ?\)thesis .

next

assume \( lt p = \) \( lt q \land \) lc \( p = \) lc \( q \land \) 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

lemma ord-strict-p-monomial-iff: \( p \prec_p \) monomial \( c \) \( v \) \( \iff \) \( (c \neq 0 \land (p = 0 \lor \) \( lt p \prec \) \( v \rangle) \)

proof

from ord-p-zero-min[of tail \( p \)] have \( \star \): \( \neg \) tail \( p \prec_p \) \( 0 \) by auto

show \( ?\)thesis


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 \( p = 0 \lor \) \( lt p \prec \) \( v \) 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 \( lt p \prec \) \( v \)

from assms(2) have \( q = 0 \lor \) \( lt q \prec \) \( v \) by (simp add: ord-strict-p-monomial-iff)
thus \(?\text{thesis}\)

proof

assume \( q = 0 \)

with \( \text{assms}(1) \) show \(?\text{thesis}\) by simp

next

assume \( l t \ q \prec_{\iota} v \)

with \( l t \ (p + q) \prec_{\iota} v \)

using \( \text{lt-plus-le-max ord-term-lin.dual-order.strict-trans2 ord-term-lin.max-less-iiff-conj} \)

by blast

with \( c \neq 0 \) show \(?\text{thesis}\) by \( \text{(simp add: ord-strict-p-monomial-iiff)} \)

qed

lemma \text{ord-strict-p-monom-mult}:

assumes \( p \prec_{p} q \text{ and } c \neq (0::'b::semiring-no-zero-divisors) \)

shows \( \text{monom-mult } c \ t \ p \prec_{p} \text{ monom-mult } c \ t \ q \)

proof –

from \( \text{assms}(1) \text{ obtain } v \text{ where } 1: \text{lookup } p \ v = 0 \text{ and } 2: \text{lookup } q \ v \neq 0 \)

and \( 3: \bigwedge u. \ v \prec_{\iota} u \Longrightarrow \text{lookup } p \ u = \text{lookup } q \ u \) unfolding \( \text{ord-strict-p-def} \) by auto

show \(?\text{thesis}\) unfolding \( \text{ord-strict-p-def} \)

proof \( \text{(intro exI conjI allI impI)} \)

from \( 1 \) show \( \text{lookup } (\text{monom-mult } c \ t \ p) (t \oplus v) = 0 \) by \( \text{(simp add: lookup-monom-mult-plus)} \)

next

from \( 2 \) \( \text{assms}(2) \) show \( \text{lookup } (\text{monom-mult } c \ t \ q) (t \oplus v) \neq 0 \) by \( \text{(simp add: lookup-monom-mult-plus)} \)

next

fix \( u \)

assume \( t \oplus v \prec_{\iota} u \)

show \( \text{lookup } (\text{monom-mult } c \ t \ p) \ u = \text{lookup } (\text{monom-mult } c \ t \ q) \ u \)

proof \( \text{(cases } t \text{ adds}_{p} u) \)

case True

then obtain \( w \text{ where } w: u = t \oplus w \).

from \( t \oplus v \prec_{\iota} w \) have \( v \prec_{\iota} w \) unfolding \( u \) by \( \text{(rule ord-term-strict-canc)} \)

hence \( \text{lookup } p \ w = \text{lookup } q \ w \) by \( \text{(rule } 3) \)

thus \(?\text{thesis}\) by \( \text{(simp add: } u \text{ lookup-monom-mult-plus)} \)

next

case False

thus \(?\text{thesis}\) by \( \text{(simp add: lookup-monom-mult)} \)

qed

qed

lemma \text{ord-strict-p-plus}:

assumes \( p \prec_{p} q \text{ and } \text{keys } r \cap \text{keys } q = \{ \} \)

shows \( p + r \prec_{p} q + r \)

proof –

from \( \text{assms}(1) \text{ obtain } v \text{ where } 1: \text{lookup } p \ v = 0 \text{ and } 2: \text{lookup } q \ v \neq 0 \)
and 3: \( \forall u. v \prec_t u \implies \text{lookup } p u = \text{lookup } q u \) 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 \( \text{thesis} \) 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 \( r \) \( v \) \( \neq \) 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

lemma poly-mapping-tail-induct [case-names 0 tail]:
assumes \( P \) \( 0 \) and \( \forall p. p \neq 0 \implies P (\text{tail } p) \implies P p \)
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 \( ?\text{case} \) unfolding \( \langle p = 0 \rangle \).
next
case ind: \( \text{Suc } n \)
from ind(2) have keys \( p \neq \) {} by auto
hence \( p \neq 0 \) by simp
thus \( ?\text{case} \)
proof (rule assms(2))
  show \( P (\text{tail } p) \)
  proof (rule ind(1))
    from \( p \neq 0 \) have lt \( p \) \( \in \) keys \( p \) by (rule lt-in-keys)
    hence card (keys (\text{tail } \( p \) )) = card (keys \( p \)) - 1 by (simp add: keys-tail)
    also have \( \ldots = n \) unfolding \( \langle \text{ind } (\text{Suc } n) \rangle \) by simp
    finally show \( n = \text{card } (\text{keys } (\text{tail } p)) \) by simp
  qed
qed

lemma poly-mapping-neqE:
assumes \( p \neq q \)
obtains \( v \) where \( v \in \text{keys } p \cup \text{keys } q \) and lookup \( p \) \( v \) \( \neq \) lookup \( q \) \( v \)
  and \( \forall u. v \prec_t u \implies \text{lookup } p u = \text{lookup } q u \)
proof
  let \( ?A = \{ v. \text{lookup } p v \neq \text{lookup } q v \} \)
define \( v \) where \( v = \text{ord-term-lin}\text{-Max } ?A \)
have \( ?A \subseteq \text{keys } p \cup \text{keys } q \)
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 ≺ 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 ≲ t v` unfolding `v-def` by (rule `ord-term-lin.Max-ge`)
with ⟨`v ≺ 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 ≠ 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 \( \text{lt } p = \text{min-term} \)
shows \( \text{monomial } (\text{lc } p) \text{ min-term } = p \)
proof (rule poly-mapping-eqI)
  fix \( v \)
  from \( \text{min-term-min[of } v \text{]} \) have \( v = \text{min-term } \lor \text{min-term } \prec_t v \) by auto
  thus \( \text{lookup } (\text{monomial } (\text{lc } p) \text{ min-term}) v = \text{lookup } p \text{ v} \)
proof
  assume \( v = \text{min-term} \)
  thus \( \text{thesis by } (\text{simp add: lookup-single lc-def assms}) \)
next
  assume \( \text{min-term } \prec_t v \)
  moreover have \( v \notin \text{keys } p \)
  proof
    assume \( v \in \text{keys } p \)
    hence \( v \leq_t \text{lt } p \)
    by (rule lt-max-keys)
    with \( \text{min-term } \prec_t v \)
    show False
    by (simp add: assms)
  qed
  ultimately show \( \text{thesis by } (\text{simp add: lookup-single in-keys-iff}) \)
qed

lemma is-monomial-monomial-ordered:
assumes \( \text{is-monomial } p \)
obtains \( c v \) where \( c 
eq 0 \) and \( \text{lc } p = c \) and \( \text{lt } p = v \) and \( p = \text{monomial } c v \)
proof
  from \( \text{assms} \) obtain \( c v \) where \( c 
eq 0 \) and \( p-eq: p = \text{monomial } c v \) by (rule is-monomial-monomial)
  note this(1)
  moreover have \( \text{lc } p = c \)
  unfolding \( p-eq \)
  by (rule lc-monomial)
  moreover from \( c 
eq 0 \)
  have \( \text{lt } p = v \)
  unfolding \( p-eq \)
  by (rule lt-monomial)
  ultimately show \( \text{thesis using } p-eq .. \)
qed

lemma monomial-plus-not-0:
assumes \( c 
eq 0 \) and \( \text{lt } p \prec_t v \)
shows \( \text{monomial } c v + p 
eq 0 \)
proof
  assume \( \text{monomial } c v + p = 0 \)
  hence \( 0 = \text{lookup } (\text{monomial } c v + p) v \) by simp
  also have \( .. = c + \text{lookup } p \text{ v} \)
  by (simp add: lookup-add)
  also have \( .. = c \)
  proof
    from \( \text{assms}(2) \)
    have \( \neg v \prec_t \text{lt } p \)
    by simp
    with \( \text{lt-max[of } p v \text{]} \)
    have \( \text{lookup } p \text{ v } = 0 \)
    by blast
    thus \( \text{thesis by simp} \)
  qed
  finally show \( \text{False using } (c 
eq 0) \) by simp
qed

209
lemma lt-monomial-plus:
  assumes \( c \neq (0 :: 'b :: \text{comm-monoid-add}) \) and \( \text{lt} \ p \prec_\text{lt} \ v \)
  shows \( \text{lt} \ (\text{monomial} \ c \ v + p) = v \)
proof
  have eq: \( \text{lt} \ (\text{monomial} \ c \ v) = v \) by (simp only: \text{lt-monomial}[OF \( c \neq 0 \)])
  moreover have \( \text{lt} \ (p + \text{monomial} \ c \ v) = \text{lt} \ (\text{monomial} \ c \ v) \) by (rule \text{lt-plus-eqI}, simp only: eq, fact)
  ultimately show ?thesis by (simp add: \text{add}.commute)
qed

lemma lc-monomial-plus:
  assumes \( c \neq (0 :: 'b :: \text{comm-monoid-add}) \) and \( \text{lt} \ p \prec_\text{lt} \ v \)
  shows \( \text{lc} \ (\text{monomial} \ c \ v + p) = c \)
proof
  from assms (2) have \( \neg \ v \preceq_\text{lt} \ p \) by simp
  with \text{lt-max}[of p v] have \( \text{lookup} \ p \ v = 0 \) by blast
  thus ?thesis by (simp add: \text{lc-def lt-monomial-plus}[OF assms] lookup-add)
qed

lemma tt-monomial-plus:
  assumes \( p \neq (0 :: \Rightarrow 0' \ b :: \text{comm-monoid-add}) \) and \( \text{lt} \ p \prec_\text{lt} \ v \)
  shows \( \text{tt} \ (\text{monomial} \ c \ v + p) = \text{tt} \ p \)
proof (cases \( c = 0 \))
  case True
  thus ?thesis by (simp add: \text{monomial-0I})
next
  case False
  have eq: \( \text{tt} \ (\text{monomial} \ c \ v) = v \) by (simp only: \text{tt-monomial}[OF \( c \neq 0 \)])
  moreover have \( \text{tt} \ (p + \text{monomial} \ c \ v) = \text{tt} \ p \) by (rule \text{tt-plus-eqI}, fact, simp only: eq)
  from \text{lt-ge-tt}[of p v] assms (2) show \( \text{tt} \ p \prec_\text{tt} \ v \) by simp
  qed
ultimately show ?thesis by (simp add: \text{ac-simps})
qed

lemma tc-monomial-plus:
  assumes \( p \neq (0 :: \Rightarrow 0' \ b :: \text{comm-monoid-add}) \) and \( \text{lt} \ p \prec_\text{lt} \ v \)
  shows \( \text{tc} \ (\text{monomial} \ c \ v + p) = \text{tc} \ p \)
proof (simp add: \text{tc-def tt-monomial-plus}[OF assms] lookup-add lookup-single \text{Poly-Mapping}.\text{when-def}, rule \text{implI})
  assume v = \( tt \ p \)
  with assms (2) have \( \text{lt} \ p \prec_\text{tt} \ tt \ p \) by simp
  with \text{lt-ge-tt}[of p] show \( c + \text{lookup} \ p \ (tt \ p) = \text{lookup} \ p \ (tt \ p) \) by simp
  qed

lemma tail-monomial-plus:
  assumes \( c \neq (0 :: 'b :: \text{comm-monoid-add}) \) and \( \text{lt} \ p \prec_\text{lt} \ v \)
  shows \( \text{tail} \ (\text{monomial} \ c \ v + p) = p \) (is \( \text{tail} \ ?q = - \))
proof
  

210
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 \( c \neq 0 \)])
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 \( \Rightarrow \) 't list
where pps-to-list S = rev (ord-term-lin.sorted-list-of-set S)
definition keys-to-list :: ('t \Rightarrow 'b::zero) \( \Rightarrow \) 't list
where keys-to-list p = pps-to-list (keys p)
definition Keys-to-list :: ('t \Rightarrow 'b::zero) list \( \Rightarrow \) 't list
where Keys-to-list ps = fold (\( \lambda p ts \). merge-wrt (≻ t) (keys-to-list p) ts) ps []

Function \( pps-to-list \) turns finite sets of terms into sorted lists, where the lists are sorted descending (i.e. greater elements come before smaller ones).

lemma distinct-pps-to-list: distinct (pps-to-list S)
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 (≻ t) (pps-to-list S)
proof -
have sorted-wrt (\( \geq t \)) (pps-to-list S)
proof -
have tr: transp (\( \geq t \)) using transp-def by fastforce
have \( \lambda x y. y \geq_t x \) = \( \preceq_t \) by simp

show \( \text{thesis} \)

by \( \text{(simp only: \* \pps-to-list-def sorted-wrt-rev ord-term-lin.sorted-sorted-wrt \symmetric, rule ord-term-lin.sorted-sorted-list-of-set)} \)

qed

with \( \text{distinct-pps-to-list have sorted-wrt} (\lambda x y. x \geq_t y \land x \neq y) \) (\ppss-to-list \ S)

by \( \text{(rule distinct-sorted-wrt-imp-sorted-wrt-strict)} \)

moreover have \( (\succ_t) = (\lambda x y. x \geq_t y \land x \neq y) \)

using \( \text{ord-term-lin.dual-order.order-iff-strict by auto} \)

ultimately show \( \text{thesis} \) by simp

qed

lemma \( \text{pps-to-list-nth-leI} \):

assumes \( j \leq i \) and \( i < \text{card } S \)

shows \( \text{(pps-to-list } \ S \text{)! } i \preceq_t \text{(pps-to-list } \ S \text{)! } j \)

proof \( \text{(cases } j = i \) )

\( \text{case True} \)

show \( \text{thesis} \) by \( \text{(simp add: True)} \)

next

\( \text{case False} \)

with \( \text{assms(1) have } j < i \) by simp

let \( ?ts = \text{pps-to-list } \ S \)

from \( \text{pps-to-list-sorted-wrt } (j < i) \) have \( (\prec_t)^{-1-1} \) \( (?ts ! j) (?ts ! i) \)

proof \( \text{(rule sorted-wrt-nth-less)} \)

from \( \text{assms(2) show } i < \text{length } ?ts \) by \( \text{(simp only: length-pps-to-list)} \)

qed

thus \( \text{thesis} \) by simp

qed

lemma \( \text{pps-to-list-nth-lessI} \):

assumes \( j < i \) and \( i < \text{card } S \)

shows \( \text{(pps-to-list } \ S \text{)! } i \prec_t \text{(pps-to-list } \ S \text{)! } j \)

proof

let \( ?ts = \text{pps-to-list } \ S \)

from \( \text{assms(1) have } j \leq i \) and \( i \neq j \) by simp-all

with \( \text{assms(2) have } i < \text{length } ?ts \text{ and } j < \text{length } ?ts \) by \( \text{(simp-all only: length-pps-to-list)} \)

show \( \text{thesis} \)

proof \( \text{(rule ord-term-lin.neq-le-trans)} \)

from \( i \neq j \) show \( ?ts ! i \neq ?ts ! j \)

by \( \text{(simp add: nth-eq-iff-index-eq[OF distinct-pps-to-list \( i < \text{length } ?ts \text{) \( j < \text{length } ?ts \text{]} \)}} \)

next

from \( j \leq i \) \( \text{assms(2) show } ?ts ! i \preceq_t ?ts ! j \) by \( \text{(rule pps-to-list-nth-leI)} \)

qed

qed

lemma \( \text{pps-to-list-nth-leD} \):

assumes \( \text{(pps-to-list } \ S \text{)! } i \preceq_t \text{(pps-to-list } \ S \text{)! } j \) and \( j < \text{card } S \)

\begin{verbatim}

212

\end{verbatim}
shows \( j \leq i \)
proof (rule ccontr)
assume \( \neg j \leq i \)
hence \( i < j \) by simp
from this \( i < j \ < \) \( \text{card} \ S \) have \( \text{pps-to-list} \ S \ ! j \prec t \ (\text{pps-to-list} \ S \ ! i) \) by (rule pps-to-list-nth-lessI)
with assms(1) show False by simp
qed

lemma pps-to-list-nth-lessD:
assumes \( \text{pps-to-list} \ S \ ! i \prec t \ (\text{pps-to-list} \ S \ ! j) \) and \( j < \text{card} \ S \)
shows \( j < i \)
proof (rule ccontr)
assume \( \neg j \prec i \)
hence \( i \leq j \) by simp
from this \( j \ < \) \( \text{card} \ S \) have \( \text{pps-to-list} \ S \ ! j \preceq t \ (\text{pps-to-list} \ S \ ! i) \) by (rule pps-to-list-nth-leI)
with assms(1) show False by simp
qed

lemma set-keys-to-list: \( \text{set} \ (\text{keys-to-list} \ p) = \text{keys} \ p \)
by (simp add: keys-to-list-def set-pps-to-list)

lemma length-keys-to-list: \( \text{length} \ (\text{keys-to-list} \ p) = \text{card} \ (\text{keys} \ p) \)
by (simp only: keys-to-list-def length-pps-to-list)

lemma keys-to-list-zero [simp]: \( \text{keys-to-list} \ \emptyset = \emptyset \)
by (simp add: keys-to-list-def)

lemma Keys-to-list-Nil [simp]: \( \text{Keys-to-list} \ [\ ] = [\ ] \)
by (simp add: Keys-to-list-def)

lemma set-Keys-to-list: \( \text{set} \ (\text{Keys-to-list} \ ps) = \text{Keys} \ (\text{set} \ ps) \)
proof
have \( \text{set} \ (\text{Keys-to-list} \ ps) = (\bigcup p \in \text{set} \ ps. \ \text{set} \ (\text{keys-to-list} \ p)) \cup [\ ] \)
unfolding Keys-to-list-def by (rule set-fold, simp only: set-merge-wrt)
also have ... = \( \text{Keys} \ (\text{set} \ ps) \) by (simp add: Keys-def set-keys-to-list)
finally show \( ?\text{thesis} \).
qed

lemma Keys-to-list-sorted-wrt-aux:
assumes sorted-wrt \( (>_{\text{t}}) \) ts
shows sorted-wrt \( (>_{\text{t}}) \) \( \{\text{fold} \ (\lambda p \ ts. \ \text{merge-wrt} \ (>_t) \ (\text{keys-to-list} \ p) \ ts) \ ps \ ts\} \)
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 (∫₁) unfolding transp-def by fastforce
next
  fix x y :: 't
  assume x ≠ y
  thus x ∫₁ y ∨ y ∫₁ x by auto
next
  show sorted-wrt (∫₁) (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 (∫₁) (Keys-to-list ps)
 unfolding Keys-to-list-def
proof (rule Keys-to-list-sorted-wrt-aux)
  show sorted-wrt (∫₁) [] by simp
  qed

 corollary distinct-Keys-to-list: distinct (Keys-to-list ps)
proof (rule distinct-sorted-wrt-irrefl)
  show irreflp (∫₁) by (simp add: irreflp-def)
next
  show transp (∫₁) unfolding transp-def by fastforce
next
  show sorted-wrt (∫₁) (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 (∫₁) 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 $v \in \text{keys} (p \odot q)$
shows $v \preceq_t \text{punit.lt} p \oplus \text{lt} q$

proof –
from assms obtain $t$ $u$ where $t \in \text{keys} p$ and $u \in \text{keys} q$ and $v = t \oplus u$
  by (rule in-keys-mult-scalarE)
from ($t \in \text{keys} p$) have $t \preceq \text{punit.lt} p$ by (rule punit.lt-max-keys)
from ($u \in \text{keys} q$) have $u \preceq \text{lt} q$ by (rule lt-max-keys)
  hence $v \preceq_t t \oplus u$ unfolding ($v = t \oplus u$) by (rule plus mono)
also from ($t \preceq \text{punit.lt} p$) have ... $\preceq_t \text{punit.lt} p \oplus \text{lt} q$ by (rule plus mono-left)
finally show \text{thesis}.
qed

lemma in-keys-mult-scalar-ge:
assumes $v \in \text{keys} (p \odot q)$
shows $\text{punit.tt} p \oplus \text{lt} q \preceq_t v$

proof –
from assms obtain $t$ $u$ where $t \in \text{keys} p$ and $u \in \text{keys} q$ and $v = t \oplus u$
  by (rule in-keys-mult-scalarE)
from ($t \in \text{keys} p$) have \text{punit.tt} $p \preceq t$ by (rule punit.tt-min-keys)
from ($u \in \text{keys} q$) have \text{lt} $q \preceq_t u$ by (rule lt-min-keys)
  hence \text{punit.tt} $p \oplus \text{lt} q \preceq_t \text{punit.tt} p \oplus u$ by (rule plus mono)
  also from ($\text{punit.tt} p \preceq_t t$) have ... $\preceq_t v$ unfolding ($v = t \oplus u$) by (rule plus mono-left)
finally show \text{thesis}.
qed

lemma (in ordered-term) lookup-mult-scalar-lt-lt:
$\text{lookup} (p \odot q) \ (\text{punit.lt} p \oplus \text{lt} q) = \text{punit.lc} p \ast \text{lc} q$

proof (induct $p$ rule: \text{punit.poly-mapping-tail-induct})
case \texttt{0}
show \texttt{?case} by simp
next
case step: (tail $p$)
from step(1) have \text{punit.lc} $p \neq 0$ by (rule punit.lc-not-0)
let \texttt{?t} = \text{punit.lt} $p \oplus \text{lt} q$
show \texttt{?case}
proof (cases is-monomial $p$)
case True
then obtain $c$ $t$ where $c \neq 0$ and \text{punit.lt} $p = t$ and \text{punit.lc} $p = c$ and
  $p$-eq: $p = \text{monomial} c t$
  by (rule punit.is-monomial-monomial-ordered)
  hence $p \odot q = \text{monom-mult} \ (\text{punit.lc} p) \ (\text{punit.lt} p) q$ by (simp add: mult-scalar-monomial)
  thus \texttt{thesis} by (simp add: lookup-monom-mult-plus lc-def)
next
case False
have \text{punit.lt} (\text{punit.tail} $p$) $\neq$ \text{punit.lt} $p$
proof (simp add: \text{punit.tail-def} \text{punit.lt-lower-eq-iff}, rule)
  assume \text{punit.lt} $p = 0$
  have \text{keys} $p \subseteq \{\text{punit.lt} p\}$
proof (rule, simp)
  fix s
  assume s ∈ keys p
  hence s ⪯ punit.lt p by (rule punit.lt-max-keys)
moreover have punit.lt p ⪯ s unfolding punit.lt p = 0) by (rule zero-min)
ultimately show s = punit.lt p by simp 
qed
hence card (keys p) = 0 ∨ 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
theorem
  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 \( c \neq 0 \) and \( \text{punit}.\lt \ p = t \) and \( \text{punit}.\leq \ p = c \) and

\[ \text{p-eq}: \ p = \text{monomial} \ c \ t \]

by (rule \text{punit}.\text{is-monomial}-\text{monomial-ordered})

from \( c \neq 0 \) have \( \text{punit}.\lt \ p = t \) and \( \text{punit}.\leq \ p = c \) by (simp-all add: \text{p-eq} \text{punit}.\text{tt-monomial})

with \( \text{p-eq} \) have \( p \odot q = \text{monom-mul} (\text{punit}.\leq \ p) (\text{punit}.\lt \ p) q \) by (simp add: \text{mult-scalar-monomial})

thus \(?thesis\) by (simp add: \text{lookup-monom-mul-plus tc-def})

next

case \text{False}

from \text{step(1)} have \( \text{keys} \ p \neq {} \) by simp

with \text{finite-keys} have \( \text{card} \ (\text{keys} \ p) \neq 0 \) by auto

with \text{False} have \( 2 \leq \text{card} \ (\text{keys} \ p) \) unfolding \text{is-monomial-def} by linarith

then obtain \( s \) \( t \) where \( s \in \text{keys} \ p \) and \( t \in \text{keys} \ p \) and \( s \prec t \)

by (metis (mono-tags, lifting) \text{card-empty} \text{card-infinite} \text{card-insert-disjoint} \text{card-mono empty-iff})

finite.emptyI insertCI lessI not-less numeral-2-eq-2 ordered-powerprod-lin.infinite-growing

ordered-powerprod-lin.preorder-class.less-le-trans subsetI)

from \text{this(1)} this(2) have \( \text{punit}.\lt \ p \prec t \) by (rule \text{punit}.\lt-less)

also from \( t \in \text{keys} \ p \) have \( t \leq \text{punit}.\lt \ p \) by (rule \text{punit}.\lt-max-keys)

finally have \( \text{punit}.\lt \ p \prec \text{punit}.\lt \ p \).

\text{hence} \( \text{tt-tail}; \text{punit}.\lt \ (\text{punit}.\tail \ p) = \text{punit}.\lt \ p \) \text{ and tc-tail; \text{punit}.\leq \ p (\text{punit}.\tail \ p) = \text{punit}.\leq \ p \}

\text{unfolding} \text{punit}.\tail-def by (rule \text{punit}.\lt-lower, rule \text{punit}.\leq-lower)

have eq: \( \text{lookup} \ (\text{monom-mul} (\text{punit}.\leq \ p) (\text{punit}.\lt \ p) q) \ ?t = 0 \)

proof (rule ccontr)

assume lookup \( (\text{monom-mul} (\text{punit}.\leq \ p) (\text{punit}.\lt \ p) q) \ ?t \neq 0 \)

hence \( \text{punit}.\lt \ p \odot tt q \preceq_t \ ?t \)

by (meson in-keys-iff in-keys-monom-mult-ge)

hence \( \text{punit}.\lt \ p \preceq \text{punit}.\lt \ p \) by (rule ord-term-canc-left)

also have \( ... \prec \text{punit}.\lt \ p \) by fact

finally show \text{False} ..

qed

from \text{step(2)} have \( \text{lookup} \ (\text{punit}.\tail \ p \odot q) \ ?t = \text{punit}.\leq \ p \circ tc \ q \) by (simp only: \text{tt-tail tc-tail})

thus \(?thesis\) by (simp add: \text{mult-scalar-tail-rec-left}[of p q] \text{lookup-add eq})

qed

qed

lemma \text{lt-mul-scalar}:  

assumes \( p \neq 0 \) and \( q \neq (0::'b::{semiring-no-zero-divisors}) \)

shows \( \lt (p \odot q) = \text{punit}.\lt \ p \odot \lt q \)

proof (rule \text{lt-eq-keys}, simp only: in-keys-iff lookup-mul-scalar-\text{lt-\text{lt}})

from \text{assms(1)} have \( \text{punit}.\leq \ p \neq 0 \) by (rule \text{punit}.\leq-not-0)

moreover from \text{assms(2)} have \( \leq c \ q \neq 0 \) by (rule \text{lc-not-0})

ultimately show \( \text{punit}.\leq \ p \circ \leq c \ q \neq 0 \) by simp

qed (rule in-keys-mul-scalar-\text{lc})

lemma \text{tt-mul-scalar}:
assumes $p \neq 0$ and $q \neq (0::'a \Rightarrow 'b::semiring-no-zero-divisors)$
sows $tt (p \odot q) = punit tt p \oplus tt q$
proof (rule tt-eqI, simp only: in_keys_iff lookup_mult_scalar_tt tt)
  from assms(1) have $punit tt p \neq 0$ by (rule punit tt not 0)
  moreover from assms(2) have $tc q \neq 0$ by (rule tc not 0)
  ultimately show $punit tt p * tc q \neq 0$ by simp
qed (rule in_keys_mult_scalar_ge)

lemma lc-mult-scalar: $lc (p \odot q) = punit lc p * lc (q::'a \Rightarrow 'b::semiring-no-zero-divisors)$
proof (cases $p = 0$)
  case True
  thus $?thesis$ by (simp add: lc_def)
next
  case False
  show $?thesis$ by (simp add: lc_def)
next
  case False
  with $p \neq 0$ show $?thesis$ by (simp add: lc_def lt_mult_scalar lookup_mult_scalar lt lt)
qed

lemma tc-mult-scalar: $tc (p \odot q) = punit tc p * tc (q::'a \Rightarrow 'b::semiring-no-zero-divisors)$
proof (cases $p = 0$)
  case True
  thus $?thesis$ by (simp add: tc_def)
next
  case False
  show $?thesis$ by (simp add: tc_def)
next
  case False
  with $p \neq 0$ show $?thesis$ by (simp add: tc_def tt_mult_scalar lookup_mult_scalar tt tt)
qed

lemma mult-scalar-not-zero:
  assumes $p \neq 0$ and $q \neq (0::'a \Rightarrow 'b::semiring-no-zero-divisors)$
  shows $p \odot q \neq 0$
proof
  assume $p \odot q = 0$
  hence $0 = lc (p \odot q)$ 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 $punit lc p \neq 0$ by (rule punit lc not 0)

end
moreover from \texttt{assms(2)} have \textit{lc} \textit{q} \neq 0 by (rule \textit{lc-not-0}) 
ultimately show False by simp 
qed 
end 

\textbf{context} ordered-powerprod 
begin 

\textbf{lemmas} in-keys-times-le = \textit{punit}.in-keys-mult-scalar-le[simplified] 
\textbf{lemmas} in-keys-times-ge = \textit{punit}.in-keys-mult-scalar-ge[simplified] 
\textbf{lemmas} lookup-times-lp-lp = \textit{punit}.lookup-mult-scalar-lt-lt[simplified] 
\textbf{lemmas} lookup-times-tp-tp = \textit{punit}.lookup-mult-scalar-lt-tt[simplified] 
\textbf{lemmas} lp-times = \textit{punit}.lt-mult-scalar[simplified] 
\textbf{lemmas} tp-times = \textit{punit}.tt-mult-scalar[simplified] 
\textbf{lemmas} times-not-zero = \textit{punit}.mult-scalar-not-zero[simplified] 
\textbf{lemmas} times-tail-rec-left = \textit{punit}.mult-scalar-tail-rec-left[simplified] 
\textbf{lemmas} times-tail-rec-right = \textit{punit}.mult-scalar-tail-rec-right[simplified] 
\textbf{lemmas} punit-in-keys-monom-mult-le = \textit{punit}.in-keys-monom-mult-le[simplified] 
\textbf{lemmas} lp-monom-mult = \textit{punit}.lt-monom-mult[simplified] 
\textbf{lemmas} tp-monom-mult = \textit{punit}.tt-monom-mult[simplified] 
end 

10.11 \textit{dgrad-p-set} and \textit{dgrad-p-set-le} 
\textbf{locale} \textit{gd-term} = 
\textit{ordered-term} \textit{pair-of-term} \textit{ord} \textit{ord-strict} \textit{ord-term} \textit{ord-term-strict} 
\textbf{for} \textit{pair-of-term}::\texttt{t} \Rightarrow \texttt{\textit{a}::graded-dickson-powerprod \times \textit{k}::\texttt{\{the-min, wellorder\}}} 
\textbf{and} \textit{term-of-pair}::\texttt{\textit{a} \times \textit{k}} \Rightarrow \texttt{t} 
\textbf{and} \textit{ord}::\texttt{\textit{a} \Rightarrow \textit{a} \Rightarrow bool} (\textbf{infixl} \leq 50) 
\textbf{and} \textit{ord-strict} (\textbf{infixl} \prec 50) 
\textbf{and} \textit{ord-term}::\texttt{\textit{t} \Rightarrow \textit{t} \Rightarrow bool} (\textbf{infixl} \preceq 50) 
\textbf{and} \textit{ord-term-strict}::\texttt{\textit{t} \Rightarrow \textit{t} \Rightarrow bool} (\textbf{infixl} \prec 50) 
begin 

\textbf{sublocale} \textit{gd-powerprod} .. 

\textbf{lemma} \textit{adds-term-antisym}: 
\textbf{assumes} \texttt{u adds\textit{1} v and v adds\textit{1} u} 
\textbf{shows} \texttt{u = v} 
\textbf{using} \texttt{assms} \textbf{unfolding} \textit{adds-term-def} \textbf{using} \textit{adds-antisym} \textbf{by} \texttt{(metis term-of-pair-pair)} 

\textbf{definition} \textit{dgrad-p-set} :: \texttt{\textit{a} \Rightarrow nat} \Rightarrow nat \Rightarrow \texttt{\textit{t} \Rightarrow \textit{b}::zero} set
where \(\text{dgrad-p-set } d \; m = \{ p. \ \text{pp-of-term } \; \text{keys } p \subseteq \text{dgrad-set } d \; m \}\)

definition \(\text{dgrad-p-set-le} : (\; \text{a } \Rightarrow \text{nat} \; \Rightarrow \; (\; \text{t } \Rightarrow_0 \; \text{b} \; \text{set} \; \Rightarrow \; ((\; \text{t } \Rightarrow_0 \; \text{b::zero} \; \text{set} \; \Rightarrow \; bool) \; \text{set}) \; \text{set}) \Rightarrow \text{set} \; \text{set}) \Rightarrow \text{bool}\)

where \(\text{dgrad-p-set-le } d \; F \; G \leftarrow (\text{dgrad-set-le } d \; (\; \text{pp-of-term } \; \text{Keys } F \; \text{set}) \; (\; \text{pp-of-term } \; \text{Keys } G \; \text{set}))\)

lemma \(\text{in-dgrad-p-set-iff}: p \in \text{dgrad-p-set } d \; m \leftrightarrow (\forall v \in \text{keys } p. \; d \; (\; \text{pp-of-term } v \; \text{set}) \leq m)\)

by (auto simp add: \(\text{dgrad-p-set-def} \; \text{dgrad-set-def}\))

lemma \(\text{dgrad-p-setI } \; \text{intro}:\)

assumes \(\forall v. \; v \in \text{keys } p \Rightarrow d \; (\; \text{pp-of-term } v \; \text{set}) \leq m\)

shows \(p \in \text{dgrad-p-set } d \; m\)

using assms by (auto simp: \(\text{in-dgrad-p-set-iff}\))

lemma \(\text{dgrad-p-setD}:\)

assumes \(p \in \text{dgrad-p-set } d \; m \; \text{and } v \in \text{keys } p\)

shows \(d \; (\; \text{pp-of-term } v \; \text{set}) \leq m\)

using assms by (simp only: \(\text{in-dgrad-p-set-iff}\))

lemma \(\text{zero-in-dgrad-p-set}: 0 \in \text{dgrad-p-set } d \; m\)

by (rule, simp)

lemma \(\text{dgrad-p-set-zero } \; \text{simp}: \; \text{dgrad-p-set } (\lambda. \; 0) \; m = \text{UNIV}\)

by auto

lemma \(\text{subset-dgrad-p-set-zero}: F \subseteq \text{dgrad-p-set } (\lambda. \; 0) \; m\)

by simp

lemma \(\text{dgrad-p-set-subset}:\)

assumes \(m \leq n\)

shows \(\text{dgrad-p-set } d \; m \subseteq \text{dgrad-p-set } d \; n\)

using assms by (auto simp: \(\text{dgrad-p-set-def} \; \text{dgrad-set-def}\))

lemma \(\text{dgrad-p-setD-lp}:\)

assumes \(p \in \text{dgrad-p-set } d \; m \; \text{and } p \neq 0\)

shows \(d \; (\; \text{lp } p \; \text{set}) \leq m\)

by (rule \(\text{dgrad-p-setD}\), fact, rule \(\text{lt-in-keys}\), fact)

lemma \(\text{dgrad-p-set-exhaust-expl}:\)

assumes \(\text{finite } F\)

shows \(F \subseteq \text{dgrad-p-set } d \; (\; \text{Max } (d \; \text{pp-of-term } \; \text{Keys } F)\; )\)

proof

fix \(f\)

assume \(f \in F\)

from assms have \(\text{finite } (\text{Keys } F)\) by (rule \(\text{finite-Keys}\))

have \(\text{fin: finite } (d \; \text{pp-of-term } \; \text{Keys } F)\) by (intro \(\text{finite-imageI}\), fact)

show \(f \in \text{dgrad-p-set } d \; (\; \text{Max } (d \; \text{pp-of-term } \; \text{Keys } F)\) )

220
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

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

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 \( s \in \text{pp-of-term}' \\text{Keys} \{f\} \) show \( \exists t \in \text{pp-of-term}' \\text{Keys} G. \ d \ s \leq d \ t \)

unfolding \( \text{dgrad-p-set-le-def} \) \( \text{dgrad-set-le-def} \) ..

qed

lemma \( \text{dgrad-p-set-le-trans} \) [trans]:
assumes \( \text{dgrad-p-set-le} \ d \ F \ G \text{ and } \text{dgrad-p-set-le} \ d \ G \ H \)
shows \( \text{dgrad-p-set-le} \ d \ F \ H \)
using assms unfolding \( \text{dgrad-p-set-le-def} \) by (rule \( \text{dgrad-set-le-trans} \))

lemma \( \text{dgrad-p-set-le-subset} \):
assumes \( F \subseteq G \)
shows \( \text{dgrad-p-set-le} \ d \ F \ G \)
unfolding \( \text{dgrad-p-set-le-def} \) by (rule \( \text{dgrad-set-le-subset} \), rule \( \text{image-mono} \), rule \( \text{Keys-mono} \), fact)

lemma \( \text{dgrad-p-set-leI-insert-keys} \):
assumes \( \text{dgrad-p-set-le} \ d \ F \ G \text{ and } \text{dgrad-set-le} \ d \ (\text{pp-of-term}' \text{keys} f) \ (\text{pp-of-term}' \text{Keys} G) \)
shows \( \text{dgrad-p-set-le} \ d \ (\text{insert} f F) \ G \)
using assms by (simp add: \( \text{dgrad-p-set-le-def} \) \( \text{Keys-insert} \) \( \text{dgrad-set-le-Un} \) \( \text{image-Un} \))

lemma \( \text{dgrad-p-set-leI-insert} \):
assumes \( \text{dgrad-p-set-le} \ d \ F \ G \text{ and } \text{dgrad-p-set-le} \ d \ \{f\} \ G \)
shows \( \text{dgrad-p-set-le} \ d \ (\text{insert} f F) \ G \)
using assms by (simp add: \( \text{dgrad-p-set-le-def} \) \( \text{Keys-insert} \) \( \text{dgrad-set-le-Un} \) \( \text{image-Un} \))

lemma \( \text{dgrad-p-set-leI-Un} \):
assumes \( \text{dgrad-p-set-le} \ d \ F1 \ G \text{ and } \text{dgrad-p-set-le} \ d \ F2 \ G \)
shows \( \text{dgrad-p-set-le} \ d \ (F1 \cup F2) \ G \)
using assms by (auto simp: \( \text{dgrad-p-set-le-def} \) \( \text{dgrad-set-le-def} \) \( \text{Keys-Un} \))

lemma \( \text{dgrad-p-set-le-dgrad-p-set} \):
assumes \( \text{dgrad-p-set-le} \ d \ F \ G \text{ and } G \subseteq \text{dgrad-p-set} \ d \ m \)
shows \( F \subseteq \text{dgrad-p-set} \ d \ m \)
proof
  fix \( f \)
  assume \( f \in F \)
  show \( f \in \text{dgrad-p-set} \ d \ m \)
  proof (rule \( \text{dgrad-p-setI} \))
    fix \( v \)
    assume \( v \in \text{keys} f \)
    from this \( \{f\} \) have \( v \in \text{Keys} F \) by (rule \( \text{in-KeysI} \))
    hence \( \text{pp-of-term} \ v \in \text{pp-of-term}' \text{Keys} F \) by simp
    with assms(1) obtain \( s \) where \( s \in \text{pp-of-term}' \text{Keys} G \text{ and } d \ (\text{pp-of-term} \ v) \leq d \ s \)
    unfolding \( \text{dgrad-p-set-le-def} \) by (rule \( \text{dgrad-set-leE} \))
    from this(1) obtain \( u \) where \( u \in \text{Keys} G \text{ and } s = \text{pp-of-term} \ u \).
    from this(1) obtain \( g \) where \( g \in G \text{ and } u \in \text{keys} g \) by (rule \( \text{in-KeysE} \))
    from this(1) assms(2) have \( g \in \text{dgrad-p-set} \ d \ m \).
  qed

222
from this \( a \in \text{keys } \gamma \) have \( d \ s \leq m \) unfolding \( s \) by (rule dgrad-p-setD)

with \( d \ (\text{pp-of-term } v) \leq d \ s \) show \( d \ (\text{pp-of-term } v) \leq m \) by (rule le-trans)

qed

qed

lemma dgrad-p-set-le-except: dgrad-p-set-le d \{\text{except } p S} \} \{p\}


lemma dgrad-p-set-le-tail: dgrad-p-set-le d \{\text{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\}


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 (\text{pp-of-term ' keys (monom-mult c t p)) (insert t (pp-of-term ' keys p))})

proof (rule dgrad-set-leI)

fix s

assume \( s \in \text{pp-of-term ' keys (monom-mult c t p)} \)

with keys-monom-mult-subset have \( s \in \text{pp-of-term ' ((\oplus) t ' keys p)} \) by fastforce

then obtain v where \( v \in \text{keys p and } s = \text{pp-of-term (t \oplus v)} \) by fastforce

have \( d \ s = \text{ord-class.max (d t) (d (pp-of-term v))} \)

by (simp only: s pp-of-term-splus dickson-gradingD1[OF assms(1)])

hence \( d \ s = d \ t \lor d \ s = d \ (\text{pp-of-term v}) \) by auto

thus \( \exists t \in \text{insert t (pp-of-term ' keys p). d s \leq d t} \)

proof

assume \( d \ s = d \ t \)

thus ?thesis by simp

next

assume \( d \ s = d \ (\text{pp-of-term v}) \)

show ?thesis

proof

from \( d \ s = d \ (\text{pp-of-term v}) \) show \( d \ s \leq d \ (\text{pp-of-term v}) \) by simp

next

from \( v \in \text{keys p} \) show \( \text{pp-of-term v } \in \text{insert t (pp-of-term ' keys p)} \) by simp

qed

qed

qed
lemma dgrad-p-set-closed-plus:
assumes \( p \in \text{dgrad-p-set} d m \) and \( q \in \text{dgrad-p-set} d m \)
shows \( p + q \in \text{dgrad-p-set} d m \)
proof
  from dgrad-p-set-le-plus have \( \{ p + q \} \subseteq \text{dgrad-p-set} d m \)
  proof (rule dgrad-p-set-le-dgrad-p-set)
    from assms show \( \{ p, q \} \subseteq \text{dgrad-p-set} d m \) by simp
  qed
thus ?thesis by simp
qed

lemma dgrad-p-set-closed-uminus:
assumes \( p \in \text{dgrad-p-set} d m \)
shows \( -p \in \text{dgrad-p-set} d m \)
proof
  from dgrad-p-set-le-uminus have \( \{-p\} \subseteq \text{dgrad-p-set} d m \)
  proof (rule dgrad-p-set-le-dgrad-p-set)
    from assms show \( \{ p \} \subseteq \text{dgrad-p-set} d m \) by simp
  qed
thus ?thesis by simp
qed

lemma dgrad-p-set-closed-minus:
assumes \( p \in \text{dgrad-p-set} d m \) and \( q \in \text{dgrad-p-set} d m \)
shows \( p - q \in \text{dgrad-p-set} d m \)
proof
  from dgrad-p-set-le-minus have \( \{ p - q \} \subseteq \text{dgrad-p-set} d m \)
  proof (rule dgrad-p-set-le-dgrad-p-set)
    from assms show \( \{ p, q \} \subseteq \text{dgrad-p-set} d m \) 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 \( p \in \text{dgrad-p-set} d m \)
shows monom-mult \( c t p \) \( \in \text{dgrad-p-set} d m \)
proof (rule dgrad-p-setI)
  fix \( v \)
  assume \( v \in \text{keys} \ (\text{monom-mult} c t p) \)
  hence pp-of-term \( v \in \text{pp-of-term} \ (\text{keys} \ (\text{monom-mult} c t p)) \) by simp
with dgrad-set-le-monom-mult[OF assms(1)] obtain \( s \) where \( s \in \text{insert} t \ (\text{pp-of-term} \ (\text{keys} p)) \)
  and \( d \ (\text{pp-of-term} v) \leq d s \) by (rule dgrad-set-leE)
from this(1) have \( s = t \lor s \in \text{pp-of-term} \ (\text{keys} p) \) by simp
thus \( d \ (\text{pp-of-term} v) \leq m \)
proof
  assume \( s = t \)
  with \( d \ (\text{pp-of-term} v) \leq d s \) assms(2) show ?thesis by simp
next
assume \( s \in \text{pp-of-term} \cdot \text{keys} \ p \)
then obtain \( u \) where \( u \in \text{keys} \ p \) and \( s = \text{pp-of-term} \ u \)
from \( \text{assms}(3) \) this(1) have \( d \ s \leq m \) unfolding \( s = \text{pp-of-term} \ u \) by (rule dgrad-p-setD)
with \( d \ (\text{pp-of-term} \ v) \leq d \ s \) show \( ?\text{thesis} \) by (rule le-trans)
qed

qed

lemma dgrad-p-set-closed-monom-mult-zero:
assumes \( p \in \text{dgrad-p-set} \ d \ m \)
shows \( \text{monom-mult} \ c \ 0 \ p \in \text{dgrad-p-set} \ d \ m \)
proof (rule dgrad-p-setI)
fix \( v \)
assume \( v \in \text{keys} \ (\text{monom-mult} \ c \ 0 \ p) \)
then obtain \( u \) where \( u \in \text{keys} \ (\text{monom-mult} \ c \ 0 \ p) \) and eq: \( \text{pp-of-term} \ v = \text{pp-of-term} \ u \)
from this(1) have \( u \in \text{keys} \ p \) by (metis keys-monom-multE splus-zero)
with \( \text{assms} \) have \( d \ (\text{pp-of-term} \ u) \leq m \) by (rule dgrad-p-setD)
thus \( d \ (\text{pp-of-term} \ v) \leq m \) by (simp only: eq)
qed

lemma dgrad-p-set-closed-except:
assumes \( p \in \text{dgrad-p-set} \ d \ m \)
shows \( \text{except} \ p \ S \in \text{dgrad-p-set} \ d \ m \)
by (rule dgrad-p-setI, rule dgrad-p-setD, rule \( \text{assms} \), simp add: keys-except)

lemma dgrad-p-set-closed-tail:
assumes \( p \in \text{dgrad-p-set} \ d \ m \)
shows \( \text{tail} \ p \in \text{dgrad-p-set} \ d \ m \)
unfolding tail-def lower-def using \( \text{assms} \) by (rule dgrad-p-set-closed-except)

10.12 Dickson’s Lemma for Sequences of Terms

lemma Dickson-term:
assumes \( \text{dickson-grading} \ d \) and \( \text{finite} \ K \)
shows almost-full-on (adds1) \( \{ t. \ \text{pp-of-term} \ t \in \text{dgrad-set} \ d \ m \land \text{component-of-term} \ t \in K \} \)
(is almost-full-on - ?A)
proof (rule almost-full-onI)
fix seq :: nat \( \Rightarrow \) 't
assume \( \ast \); \( \forall i. \ \text{seq} \ i \in ?A \)
define \( \text{seq}' \) where \( \text{seq}' = (\lambda i. (\text{pp-of-term} \ (\text{seq} \ i), \text{component-of-term} \ (\text{seq} \ i))) \)
have \( \text{pp-of-term} \ ?A \subseteq \{ x. \ d \ x \leq m \} \) by (auto dest: dgrad-setD)
moreover from \( \text{assms}(1) \) have almost-full-on (adds) \( \{ x. \ d \ x \leq m \} \) by (rule dickson-gradingD2)
ultimately have almost-full-on (adds) \( \text{(pp-of-term} \ ?A) \) by (rule almost-full-on-subset)
moreover have almost-full-on (=) \( \text{(component-of-term} \ ?A) \)
proof (rule eq-almost-full-on-finite-set)
have component-of-term ' ?A ⊆ 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 × component-of-term ' ?A)
by (rule almost-full-on-Sigma)
moreover from * have ∀ i. seq' i ∈ pp-of-term ' ?A × component-of-term ' ?A
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, seq j by (simp add: seq'-def prod-le-def adds-term-def)
with (i < j) show good (adds,seq) by (rule goodI)
qed

corollary Dickson-termE:
assumes dickson-grading d and finite (component-of-term ' range (f::nat ⇒ 't))
and pp-of-term ' range f ⊆ dgrad-set d m
obtains i j where i < j and f i adds, f j
proof –
let ?A = {t. pp-of-term t ∈ dgrad-set d m ∧ component-of-term t ∈ component-of-term ' range f}
from assms(1, 2) have almost-full-on (adds, ?A) by (rule Dickson-term)
moreover from assms(3) have ∀ i. f i ∈ ?A by blast
ultimately obtain i j where i < j and f i adds, 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 ⊆ dgrad-set d m
obtains T where finite T and T ⊆ S and ∀ s ∈ S ⇒ (∃ t ∈ T. t adds, t)
proof –
let ?A = {t. pp-of-term t ∈ dgrad-set d m ∧ component-of-term t ∈ 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 ⊆ ?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 ⊆ S and ∀ s ∈ S ⇒ (∃ t ∈ T. t adds, t)
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 ≺ₜ u)

**Definition** dickson-less-p :: ('a ⇒ nat) ⇒ nat ⇒ ('t ⇒ 0) ⇒ ('t ⇒ 0) :: zero ⇒ bool

where dickson-less-p d m p q ←→ (\{p, q\} ⊆ dgrad-p-set d m ∧ p ≺ₚ q)

**Lemma** dickson-less-vI:

assumes d (pp-of-term v) ≤ m and d (pp-of-term u) ≤ m and v ≺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 ≺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** wfdickson-less-v-aux1:

assumes dickson-grading d and \( \bigwedge i :: nat. \ dickson-less-v d m \ (seq (Suc i)) \ (seq i) \)

obtains i where \( \bigwedge j. j > i \implies \) component-of-term (seq j) < component-of-term (seq i)

(proof –

let \( ?Q = pp-of-term 'r range seq \)

have pp-of-term (seq 0) ∈ ?Q by simp

with wfdickson-less[OF assms(1)] obtain t where \( ?Q \ and \ \ast :: \ \bigwedge s. \ dickson-less d m t \implies s \notin \ ?Q \)

by (rule wfdickson-less[simp add: 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 \texttt{assms}(2) have \texttt{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 \preceq seq i by (rule dickson-less-vD3)
  define s where s = \texttt{pp-of-term} (seq j)
  have \texttt{pp-of-term} (seq j) \in \mathbb{Q} by simp
  hence \neg dickson-less d m s t unfolding s-def using * by blast
  moreover from \texttt{dlv} have d s \leq m and d t \leq m unfolding s-def t
    by (rule dickson-less-vD1, rule dickson-less-vD2)
  ultimately have t \preceq s by (simp add: dickson-less-vD3)
  qed
  qed

lemma \texttt{wf-dickson-less-v-aux2}:
  assumes dickson-grading d and \( \forall i :: \text{nat}. \, \text{dickson-less-v} d m \ (\text{seq} \ (\text{Suc} \ i)) \ (\text{seq} \ i) \)
      and \( \forall i :: \text{nat}. \, \text{component-of-term} \ (\text{seq} \ i) < \text{component-of-term} \ (\text{seq} \ i) \)
  shows thesis
  using \texttt{assms}(2, 3)
  proof (induct k arbitrary: seq thesis rule: \text{less-induct})
    case (less k)
    from \texttt{assms}(1) less(2) obtain i where *: \( \forall j. \, j > i \Rightarrow \text{component-of-term} \ (\text{seq} \ j) < \text{component-of-term} \ (\text{seq} \ i) \)
      by (rule \texttt{wf-dickson-less-v-aux1}, blast)
    define seq1 where seq1 = (\lambda j. \text{seq} (\text{Suc} \ (i + j)))
    from less(3) show \texttt{?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 \text{component-of-term} \ (seq1 \ j) < \text{component-of-term} \ (seq \ i) by (simp add: seq1-def *)
    qed
  qed

lemma \texttt{wf-dickson-less-v}:
  assumes \text{dickson-grading} d
shows \( \text{wfP} \) (\text{dickson-less-v} \ d \ m)  
proof (\text{rule \text{wfP-chain}}, \text{rule}, \text{elim \text{exE}})  
fix \ seq :: \text{nat} \Rightarrow \prime t  
assume \ \forall \ i. \ \text{dickson-less-v} \ d \ m \ (\text{seq} \ (\text{Suc} \ i)) \ (\text{seq} \ i)  
hence \ \ast: \ \exists i. \ \text{dickson-less-v} \ d \ m \ (\text{seq} \ (\text{Suc} \ i)) \ (\text{seq} \ i) .  
with \ \text{assms} \ \text{obtain} \ i \ \text{where} \ \ast\ast: \ \bigwedge j. \ j > i \implies \text{component-of-term} \ (\text{seq} \ j) < \text{component-of-term} \ (\text{seq} \ i)  
\text{by (\text{rule \text{wf-dickson-less-v-aux1}}, \text{blast})}  
define \ seq1 \ \text{where} \ \seq1 = (\lambda j. \ \text{seq} \ (\text{Suc} \ (i + j)))  
from \ \text{assms} \ \text{show} \ False  
proof (\text{rule \text{wf-dickson-less-v-aux2}})  
fix \ j  
show \ \text{dickson-less-v} \ d \ m \ (\text{seq1} \ (\text{Suc} \ j)) \ (\text{seq1} \ j) \ \text{by (simp add: seq1-def, fact \ast)}  
next  
fix \ j  
show \ \text{component-of-term} \ (\text{seq1} \ j) < \text{component-of-term} \ (\text{seq} \ i) \ \text{by (simp add: seq1-def \ast\ast)}  
qed  
qed

lemma \text{dickson-less-v-zero}: \ \text{dickson-less-v} \ (\lambda \cdot 0) \ m = (\prec t)  
by (\text{rule, rule}, \text{simp add: dickson-less-v-def})

lemma \text{dickson-less-pI}:  
assumes \ p \in \text{dgrad-p-set} \ d \ m \ \text{and} \ q \in \text{dgrad-p-set} \ d \ m \ \text{and} \ p \prec_p q  
shows \ \text{dickson-less-p} \ d \ m \ p \ q  
using \ \text{assms} \ \text{by (simp add: dickson-less-p-def)}

lemma \text{dickson-less-pD1}:  
assumes \ \text{dickson-less-p} \ d \ m \ p \ q  
shows \ \text{p} \in \text{dgrad-p-set} \ d \ m  
using \ \text{assms} \ \text{by (simp add: dickson-less-p-def)}

lemma \text{dickson-less-pD2}:  
assumes \ \text{dickson-less-p} \ d \ m \ p \ q  
shows \ \text{q} \in \text{dgrad-p-set} \ d \ m  
using \ \text{assms} \ \text{by (simp add: dickson-less-p-def)}

lemma \text{dickson-less-pD3}:  
assumes \ \text{dickson-less-p} \ d \ m \ p \ q  
shows \ \text{p} \prec_p q  
using \ \text{assms} \ \text{by (simp add: dickson-less-p-def)}

lemma \text{dickson-less-p-irrefl}: \ \neg \ \text{dickson-less-p} \ d \ m \ p \ p  
by (simp add: dickson-less-p-def)

lemma \text{dickson-less-p-trans}:  
assumes \ \text{dickson-less-p} \ d \ m \ p \ q \ \text{and} \ \text{dickson-less-p} \ d \ m \ q \ r
shows \( \text{dickson-less-p} \ d \ m \ p \ r \)
using \( \text{assms} \) by (\( \text{auto simp add: dickson-less-p-def} \))

lemma \( \text{dickson-less-p-mono} \):
\begin{align*}
\text{assumes} & \quad \text{dickson-less-p} \ d \ m \ p \ q \ \text{and} \ m \leq n \\
\text{shows} & \quad \text{dickson-less-p} \ d \ n \ p \ q
\end{align*}
\begin{proof}
\from \text{assms}(2) \ have \ \( \text{dgrad-p-set} \ d \ m \subseteq \text{dgrad-p-set} \ d \ n \) \text{by (rule dgrad-p-set-subset)}
\moreover \text{from assms}(1) \ have \ p \in \text{dgrad-p-set} \ d \ m \ \text{and} \ q \in \text{dgrad-p-set} \ d \ m \\
\text{and} \ p \prec_p q \ \text{by (rule dickson-less-pD1, rule dickson-less-pD2, rule dickson-less-pD3)}
\ultimately \have \ p \in \text{dgrad-p-set} \ d \ n \ \text{and} \ q \in \text{dgrad-p-set} \ d \ n \ \text{by auto}
\langle p \prec_p q \rangle \ show \ \text{thesis} \ by \ (\text{rule dickson-less-pI})
\end{proof}
qed

lemma \( \text{dickson-less-p-zero} \): \( \text{dickson-less-p} \ (\lambda \cdot \ 0) \ m = (\prec_p) \)
by (\text{rule, rule, simp add: dickson-less-p-def})

lemma \( \text{wf-dickson-less-p-aux} \):
\begin{align*}
\text{assumes} & \quad \text{dickson-grading} \ d \\
\text{assumes} & \quad x \in Q \ \text{and} \ \forall y \in Q. y \neq 0 \longrightarrow (y \in \text{dgrad-p-set} \ d \ m \ \text{and} \ \text{dickson-less-v} \\
\text{d m} \ (lt \ y) \ u) \\
\text{shows} & \quad \exists p \in Q. (\forall q \in Q. \neg \text{dickson-less-p} \ d \ m \ q \ p) \\
\text{using} & \quad \text{assms}(2) \ \text{assms}(3)
\end{align*}
\begin{proof}
\text{induct} \ a \ \text{arbitrary; x Q rule: \text{wfP-induct[OF \text{wf-dickson-less-v}, \text{OF assms}(1)]}}
\fix \ a::'t \ \text{and} \ x::'t \Rightarrow 0 'b \ \text{and} \ Q::('t \Rightarrow 0 'b) \ \text{set}
\text{assume} \ \text{hyp}: \forall u0. \ \text{dickson-less-v} \ d \ m \ u0 \ u \longrightarrow (\forall x0 Q0::('t \Rightarrow 0 'b) \ \text{set.} \ x0 \in Q0 \longrightarrow \ (\forall y \in Q0. y \neq 0 \longrightarrow (y \in \text{dgrad-p-set} \ d \ m \ \text{and} \ \text{dickson-less-v} \\
\text{d m} \ (lt \ y) \ u0)) \longrightarrow \ (\exists p \in Q0. \forall q \in Q0. \neg \text{dickson-less-p} \ d \ m \ q \ p))
\text{assume} \ x \in Q \\
\text{assume} \ \forall y \in Q. y \neq 0 \longrightarrow (y \in \text{dgrad-p-set} \ d \ m \ \text{and} \ \text{dickson-less-v} \ d \ m \ (lt \ y) \ u) \\
\text{hence} \ \text{bounded}: \ \forall y, y \in Q \Longrightarrow y \neq 0 \Longrightarrow (y \in \text{dgrad-p-set} \ d \ m \ \text{and} \ \text{dickson-less-v} \\
\text{d m} \ (lt \ y) \ u) \ \text{by auto}
\text{show} \ \exists p \in Q. \forall q \in Q. \neg \text{dickson-less-p} \ d \ m \ q \ p
\text{proof} \ (\text{cases} \ 0 \in Q)
\text{case} \ True \\
\text{show} \ \text{thesis}
\text{proof} \ (\text{rule, rule, rule})
\fix q::'t \Rightarrow 0 'b \\
\text{assume} \ \text{dickson-less-p} \ d \ m \ q \ 0 \\
\text{hence} \ q \prec_p 0 \ \text{by (rule dickson-less-pD3)} \\
\text{thus} \ False \ \text{using \text{ord-p-zero-min[of q] by simp}
\text{next}
\text{from} \ True \ \text{show} \ 0 \in Q .
\text{qed}
\text{next}
\text{case} \ False

230
define \( Q_1 \) where \( Q_1 = \{ \text{lt } p \mid p, p \in Q \} \)

from \( x \in Q \): have \( \text{lt } x \in Q_1 \) unfolding \( Q_1\text{-def} \) by auto

with \( \text{wf-dickson-less-v}[\text{OF assms(1)}] \) obtain \( v \)

where \( v \in Q_1 \) and \( v\text{-min-1}: \\forall q. \ \text{dickson-less-v } d \ m \ q \ v \implies q \notin Q_1 \)

by (rule \( \text{wfE-min[to-pred]} \), auto)

have \( v\text{-min}: \\forall q. q \in Q \implies \neg \text{dickson-less-v } d \ m \ (\text{lt } q) \ v \)

proof

- fix \( q \)
  - assume \( q \in Q \)
  - hence \( \text{lt } q \in Q_1 \) unfolding \( Q_1\text{-def} \) by auto
  - thus \( \neg \text{dickson-less-v } d \ m \ (\text{lt } q) \ v \) using \( \text{v-min-1} \) by auto

qed

from \( \langle v \in Q_1 \rangle \) obtain \( p \) where \( \text{lt } p = v \) and \( p \in Q \) unfolding \( Q_1\text{-def} \) by auto

hence \( p \neq 0 \) using \( \text{False} \) by auto

with \( p \in Q \): have \( p \in \text{dgrad-p-set } d \ m \wedge \text{dickson-less-v } d \ m \ (\text{lt } p) \ u \) by (rule bounded)

hence \( p \in \text{dgrad-p-set } d \ m \) and \( \text{dickson-less-v } d \ m \ (\text{lt } p) \ u \) by simp-all

moreover from \( \text{this}(1) \ (p \neq 0) \) have \( d \ (\text{pp-of-term } (\text{lt } p)) \leq m \) by (rule \( \text{dgrad-p-setD-lp} \))

ultimately have \( d \ (\text{pp-of-term } v) \leq m \) by (simp only: \( \text{lt } p = v \))

define \( Q_2 \) where \( Q_2 = \{ \text{tail } p \mid p, p \in Q \wedge \text{lt } p = v \} \)

from \( p \in Q \): \( \langle \text{lt } p = v \rangle \) have \( \text{tail } p \in Q_2 \) unfolding \( Q_2\text{-def} \) by auto

have \( \forall q \in Q_2, q \neq 0 \implies (q \in \text{dgrad-p-set } d \ m \wedge \text{dickson-less-v } d \ m \ (\text{lt } q) \ (\text{lt } p)) \)

proof (intro ballI impI)

fix \( q \)

- assume \( q \in Q_2 \)
  - then obtain \( q' \) where \( q: q = \text{tail } q' \) and \( \text{lt } q' = \text{lt } p \) and \( q' \in Q \)
    using \( \langle \text{lt } p = v \rangle \) by (auto simp add: \( Q_2\text{-def} \))
    assume \( q \neq 0 \)
    hence \( \text{tail } q' \neq 0 \) using \( \langle q = \text{tail } q' \rangle \) by simp
    hence \( q' \neq 0 \) by auto

with \( \langle q' \in Q \rangle \) have \( q' \in \text{dgrad-p-set } d \ m \wedge \text{dickson-less-v } d \ m \ (\text{lt } q') \ u \) by

(rule bounded)

hence \( q' \in \text{dgrad-p-set } d \ m \) and \( \text{dickson-less-v } d \ m \ (\text{lt } q') \ u \) by simp-all

from \( \text{this}(1) \) have \( q \in \text{dgrad-p-set } d \ m \) unfolding \( q \) by (rule \( \text{dgrad-p-set-closed-tail} \))

show \( q \in \text{dgrad-p-set } d \ m \wedge \text{dickson-less-v } d \ m \ (\text{lt } q) \ (\text{lt } p) \)

proof

- show \( \text{dickson-less-v } d \ m \ (\text{lt } q) \ (\text{lt } p) \)
  proof (rule \( \text{dickson-less-vI} \))
    from \( \langle q \in \text{dgrad-p-set } d \ m \rangle \ (q \neq 0) \) show \( d \ (\text{pp-of-term } (\text{lt } q)) \leq m \) by

    (rule \( \text{dgrad-p-setD-lp} \))

next

from \( \langle \text{dickson-less-v } d \ m \ (\text{lt } p) \ w \rangle \) show \( d \ (\text{pp-of-term } (\text{lt } p)) \leq m \) by

(rule \( \text{dickson-less-vD1} \))

next

from \( \text{lt-tail}[\text{OF } (\text{tail } q' \neq 0)] \ (\text{lt } q) = \text{tail } q'; \ (\text{lt } q') = \text{lt } p) \) show \( \text{lt } q \prec_t \text{lt } p \) by simp

231
qed

qed fact

qed

with hyp ⟨dickson-less-v d m (lt p) w⟩ ⟨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 ⊲ₚ q0
  by (rule dickson-less-pD1, rule dickson-less-pD2, rule dickson-less-pD3)
from this(3) have lt r ⪯ₜ lt q0 by (simp add: ord-p-lt)
with ⟨lt q0 = v⟩ have lt r ⪯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
theorem wf-dickson-less-p:
assumes dickson-grading d
shows wff (dickson-less-p d m)
proof (rule wfI-min[ta-pred])
fix Q::('t ⇒ '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 wfI-min[ta-pred])
from True show 0 ∈ Q.
next
fix q::'t ⇒ 'b
assume dickson-less-p d m q 0
hence q ≺ p 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[ta-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 = {x ∈ Q | lt x ∈ ?L}
from ⟨x1 ∈ Q⟩ have tail x1 ∈ Q1 by (auto simp add: Q1-def [OF v = lt x1])
with assms have ∃v∈Q1. ∀q∈Q1. ¬ dickson-less-p d m q p
by (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
qed
proof (rule dickson-less-vI)
  from \(\langle y \in dgrad-p-set \ d \ m \ \ y \neq 0 \rangle\) show \(d \ (pp-of-term \ (lt \ y)) \leq m\)
  by (rule dgrad-p-setD-lp)
next
  from \(\langle y \neq 0 \rangle\) show \(lt \ y \prec_t v\) unfolding \(v = \lt y1\) \(y = \tail y1\) by
    (rule lt-tail)
    qed
    qed
    qed
then obtain \(p0\) where \(p0 \in Q1\) and \(\forall q \in Q1. \neg \text{dickson-less-p} \ d \ m \ q \ p0\)
  by blast
  from this\((1)\) obtain \(p\) where \(p \in Q\) and \(v = \lt p\) and \(p0 = \tail p\) unfolding
  \(Q1-def\)
  by blast
  from this\((1)\) False have \(p \neq 0\) by blast
  show \(?thesis\)
    proof
      (intro bexI allI impI notI)
      fix \(y\)
      assume \(y \in Q\)
      hence \(y \neq 0\) using False by blast
      assume \(\text{dickson-less-p} \ d \ m \ y \ p\)
      hence \(y \in dgrad-p-set \ d \ m\) \(\text{and} \ p \in dgrad-p-set \ d \ m\) \(\text{and} \ y \prec_p \ p\)
        by (rule dickson-less-pD1, rule dickson-less-pD2, rule dickson-less-pD3)
      from this\((3)\) have \(y \preceq_p p\) by simp
      hence \(lt \ y \preceq_l \lt p\) by (rule ord-p-lt)
      moreover have \(\neg \ (lt \ y \prec_t \lt p)\)
      proof
        assume \(lt \ y \prec_t \lt p\)
        have \(\text{dickson-less-v} \ d \ m \ (lt \ y) \ v\) unfolding \(v = \lt p\)
          by (rule dickson-less-vI, rule dgrad-p-setD-lp, fact+, rule dgrad-p-setD-lp, fact+)
        hence \(lt \ y \notin \?L\) by (rule v-min)
        hence \(y \notin Q\) by fastforce
        from this \((y \in Q)\) show False ..
      qed
      ultimately have \(lt \ y = \lt p\) by simp
      from \((y \neq 0)\) this \((y \prec_p p)\) have \(\tail y \prec_p \tail p\) by (rule ord-p-tail)
      from \((y \in Q)\) have \(\tail y \in Q1\) by (auto simp add: \(Q1-def\) \(v = \lt p\) \(\lt y\)
        = \lt p[\text{symmetric}]\)
      hence \(\neg \text{dickson-less-p} \ d \ m \ (\tail y) \ p0\) by (rule p0-min)
      moreover have \(\text{dickson-less-p} \ d \ m \ (\tail y) \ p0\) 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
next
  case False
  then obtain \(q\) where \(q \in Q\) and \(q \notin dgrad-p-set \ d \ m\) by blast
from this\(^{(1)}\) show \(?\)thesis
proof
  show \(\forall y. \text{dickson-less-p } d m y q \rightarrow y \notin Q\)
  proof (intro allI impI)
    fix y
    assume \(\text{dickson-less-p } d m y q\)
    hence \(q \in \text{dgrad-p-set } d m\) by (rule \text{dickson-less-pD2})
    with \(\langle q \notin \text{dgrad-p-set } d m \rangle\) show \(y \notin Q\) ..
  qed
  qed
  qed
  qed

\textbf{corollary} \textit{ord-p-minimum-dgrad-p-set:}
  assumes \(\text{dickson-grading } d\) and \(x \in Q\) and \(Q \subseteq \text{dgrad-p-set } d\ m\)
  obtains \(q\) where \(q \in Q\) and \(\forall y. y \prec_P q \implies y \notin Q\)
proof --
  from \textit{assms(1)} have \(\text{wfP } (\text{dickson-less-p } d m)\) by (rule \text{wf-dickson-less-p})
  from this \textit{assms(2)} obtain \(q\) where \(q \in Q\) and \(\ast:\forall y. \text{dickson-less-p } d m y q \implies y \notin Q\)
  by (rule \text{wfE-min[to-pred], auto})
  from \textit{assms(3)} \(q \in Q\) have \(q \in \text{dgrad-p-set } d m\) ..
  from \(q \in Q\) show \(?\)thesis
proof
    fix y
    assume \(y \prec_P q\)
    show \(y \notin Q\)
    proof
      assume \(y \in Q\)
      with \textit{assms(3)} have \(y \in \text{dgrad-p-set } d m\) ..
      from this \(\langle q \in \text{dgrad-p-set } d m \rangle \langle y \prec_P q \rangle\) have \(\text{dickson-less-p } d m y q\)
      by (rule \text{dickson-less-pI})
      hence \(y \notin Q\) by (rule \ast)
      from this \(y \in Q\) show False ..
    qed
    qed
    qed

\textbf{lemma} \textit{ord-term-minimum-dgrad-set:}
  assumes \(\text{dickson-grading } d\) and \(v \in V\) and \(\text{pp-of-term} \ v \subseteq \text{dgrad-set } d\ m\)
  obtains \(u\) where \(u \in V\) and \(\forall w. w \prec_t u \implies w \notin V\)
proof --
  from \textit{assms(1)} have \(\text{wfP } (\text{dickson-less-v } d m)\) by (rule \text{wf-dickson-less-v})
  then obtain \(u\) where \(u \in V\) and \(\ast:\forall w. \text{dickson-less-v } d m w u \implies w \notin V\)
  using \textit{assms(2)}
  by (rule \text{wfE-min[to-pred]} blast
  from this\(^{(1)}\) have \(\text{pp-of-term } u \in \text{pp-of-term} \ v\) by (rule \text{imageI})
  with \textit{assms(3)} have \(\text{pp-of-term } u \in \text{dgrad-set } d m\) ..
hence \( d \ (\text{pp-of-term } u) \leq m \) by (rule \text{dgrad-setD})

from \( \langle u \in V \rangle \) show \( ?\text{thesis} \)

proof

\begin{align*}
\text{fix } w \\
\text{assume } w \prec_t u \\
\text{show } w \notin V
\end{align*}

proof

\begin{align*}
\text{assume } w \in V \\
\text{hence } \text{pp-of-term } w \in \text{pp-of-term ' } V \text{ by (rule imageI)} \\
\text{with } \text{assms}(3) \text{ have } \text{pp-of-term } w \in \text{dgrad-set } d \ m ..
\end{align*}

\begin{align*}
\text{hence } d \ (\text{pp-of-term } w) \leq m \text{ by (rule \text{dgrad-setD})} \\
\text{from this } \langle d \ (\text{pp-of-term } u) \leq m \rangle \ (w \prec_t u) \text{ have } \text{dickson-less-v } d \ m \ w \ a
\end{align*}

by (rule \text{dickson-less-vI})

\begin{align*}
\text{hence } w \notin V \text{ by (rule *)} \\
\text{from this } \langle w \in V \rangle \text{ show } \text{False} ..
\end{align*}

qed

qed

qed

end

10.14 More Interpretations

context \text{gd-powerprod}

begin

sublocale punit: \text{gd-term to-pair-unit} \text{fst} (\preceq) (\prec) (\preceq) (\prec) ..

end

locale \text{od-term} =

\begin{align*}
\text{ordered-term} \quad \text{pair-of-term} \quad \text{term-of-pair} \quad \text{ord} \quad \text{ord-strict} \quad \text{ord-term} \quad \text{ord-term-strict}
\end{align*}

\begin{align*}
\text{for } \text{pair-of-term}: \text{'} t \Rightarrow (\text{'a::dickson-powerprod} \times \text{'} k::\{\text{the-min,wellorder}\}) \\
\text{and } \text{term-of-pair}: (\text{'} a \times \text{'} k) \Rightarrow \text{'} t \\
\text{and } \text{ord}: \text{'a} \Rightarrow \text{'a} \Rightarrow \text{bool} \ (\text{infixl} \leq 50) \\
\text{and } \text{ord-strict} \ (\text{infixl} < 50) \\
\text{and } \text{ord-term}: \text{'} t \Rightarrow \text{'} t \Rightarrow \text{bool} \ (\text{infixl} \preceq 50) \\
\text{and } \text{ord-term-strict}: \text{'} t \Rightarrow \text{'} t \Rightarrow \text{bool} \ (\text{infixl} \prec 50)
\end{align*}

begin

sublocale \text{gd-term} ..

lemma \text{ord-p-wf}: \text{wfP} (\prec_p)

proof

\begin{align*}
\text{from } \text{dickson-grading-zero} \text{ have } \text{wfP} (\text{dickson-less-p } (\lambda\cdot. \ 0) \ 0) \text{ by (rule \text{wf-dickson-less-p})} \\
\text{thus } ?\text{thesis} \text{ by (simp only: dickson-less-p-zero)}
\end{align*}

qed

end

236
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(I))

In this theory, type 'a ⇒'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 (λk. 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 (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)

end
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-def 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 ⇒ '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 (λk v. P (lookup0 xs k) (lookup0 ys k)) (xs ++ 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 (λk v. lookup0 xs k + lookup0 ys k) (xs ++ 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 k
  by (transfer, simp)

lemma compute-minus-pp [code]:
  Pm-fmap xs − Pm-fmap ys = Pm-fmap (clearjunk0 (fmmap-keys (λk v. lookup0 xs k − lookup0 ys k) (xs ++ 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 (λk 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 (λk v. lookup0 xs k = lookup0 ys k) (xs ++ 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 (λx. f x when x ≠ 0) x)
  by (auto intro!: poly-mapping-eqI
    simp: fmlookup-default-def map.rep-eq
    split: option.splits)
lemma fmrang′-fmfilter-eq: fmrang′ (fmfilter p fm) = {y | \exists x \in fmdom′ fm, p x \land 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) = fmrang′ (clearjunk0 xs)
by (force simp: range.rep-eq clearjunk0-def fmrang′-fmfilter-eq fmdom′ I
fmlookup-default-def split: option.splits)

10.17.1 Constructors
definition sparse0 xs = Pm-fmap (fmap-of-list xs) — sparse representation
definition dense0 xs = Pm-fmap (fmap-of-list (zip [0..<\length xs] xs)) — dense representation

lemma compute-single[code]: Poly-Mapping.single k v = sparse0 [(k,v)]
by (auto simp: sparse0-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 (\lambda k v. Orderings.max (lookup0 xs k) (lookup0 ys k)) (xs ++ \ 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]:
\[ \text{deg-pm} (\text{Pm-fmap } \text{xs}) = \text{sum} (\text{the o fmlookup } \text{xs}) (\text{fdom}' \text{xs}) \]

proof

- have \( \text{deg-pm} (\text{Pm-fmap } \text{xs}) = \text{sum} (\text{lookup} (\text{Pm-fmap } \text{xs})) (\text{keys} (\text{Pm-fmap } \text{xs})) \)
  by (rule \text{deg-pm-superset}) auto

also have \( \ldots = \text{sum} (\text{the o fmlookup } \text{xs}) (\text{fdom}' \text{xs}) \)
  by (rule \text{sum.mono-neutral-cong-left})

(auto simp: \text{fmlookup-dom}'-iff \text{fmdom}'I \text{in-iff fmlookup-default-def} \\
  split: \text{option.splits})

finally show \( ?\text{thesis} \).

qed

definition adds-pp-add-linorder :: \( ('b \Rightarrow '0 \Rightarrow 'a) \Rightarrow - \Rightarrow \\
\Rightarrow \text{bool} \)
where [code-abbrev]: \text{adds-pp-add-linorder} = \text{adds}

lemma compute-adds-pp[code]:
\[ \text{adds-pp-add-linorder} (\text{Pm-fmap } \text{xs}) (\text{Pm-fmap } \text{ys}) = \text{fmpred} (\lambda k v. \text{lookup0 } \text{xs } k \leq \text{lookup0 } \text{ys } k) (\text{xs} ++ f \text{ys}) \]

for \text{xs \text{ys} :: ('a, \text{--} :: \text{ordered-comm-monoid-add}) \\
\Rightarrow \Rightarrow \text{fmap} \}

unfolding \text{adds-pp-add-linorder-def} 

unfolding \text{adds-poly-mapping} 

using \text{fmdom-notI} 

by (force simp: \text{fmlookup-dom-iff le-fun-def} \\
  split: \text{option.splits if-splits})

Computing \( \text{lex} \) as below is certainly not the most efficient way, but it works.

lemma lex-pm-iff: \text{lex-pm } s t = (\forall x. \text{lookup } s x \leq \text{lookup } t x \lor (\exists y < x. \text{lookup } s y \neq \text{lookup } t y))

proof

- have \( \text{lex-pm } s t = (\neg \text{lex-pm-strict } t s) \) by (simp add: \text{lex-pm-strict-alt})

also have \( \ldots = (\forall x. \text{lookup } s x \leq \text{lookup } t x \lor (\exists y < x. \text{lookup } s y \neq \text{lookup } t y)) \)

  by (simp add: \text{lex-pm-strict-def less-poly-mapping-def less-fun-def}) (metis leD leI)

finally show \( ?\text{thesis} \).

qed

lemma compute-lex-pp[code]:
\[ (\text{lex-pm} (\text{Pm-fmap } \text{xs}) (\text{Pm-fmap } (\text{ys} :: (\text{--} :: \text{ordered-comm-monoid-add}) \text{fmap}))) = \]

\[ \text{let } \text{zs} = \text{xs} ++ f \text{ys} \text{ in} \]
\[ \text{fmpred} (\lambda x v. \text{lookup0 } \text{zs } x \leq \text{lookup0 } \text{ys } x \lor \neg \text{fmpred} (\lambda y w, y \geq x \lor \text{lookup0 } \text{zs } y \leq \text{lookup0 } \text{ys } y) \text{zs}) \text{zs} \]

unfolding \text{Let-def lex-pm-iff fmpred-iff Pm-fmap.rep-eq fmlookup-add fmlookup-dom-iff} 

apply (intro iffl) 

apply (metis \text{fmdom}'-notD \text{fmlookup-default-if} (2) \text{fmlookup-dom-}'-iff leD)
apply (metis eq_iff not-le fnlookup-default-if (2) fnlookup-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
(auto simp: Let-def deg-pm.
rep-eq dord-fun-def dord-pm.
rep-eq
(auto simp: Pm-fmap.abs-eq)
11.1.1 Computations

experiment begin

abbreviation X ≡ 0::nat
abbreviation Y ≡ 1::nat
abbreviation Z ≡ 2::nat

lemma sparse0 [(X, 2::nat), (Z, 7)] + sparse0 [(Y, 3), (Z, 2)] = sparse0 [(X, 2), (Z, 9), (Y, 3)]
dense0 [2, 0, 7::nat] + dense0 [0, 3, 2] = dense0 [2, 3, 9]
by eval+

lemma sparse0 [(X, 2::nat), (Z, 7)] − sparse0 [(X, 2), (Z, 2)] = sparse0 [(Z, 5)]
by eval

lemma lcs (sparse0 [(X, 2::nat), (Y, 1), (Z, 7)]) (sparse0 [(Y, 3), (Z, 2)]) = sparse0
[(X, 2), (Y, 3), (Z, 7)]
by eval

lemma (sparse0 [(X, 2::nat), (Z, 1)]) adds (sparse0 [(X, 3), (Y, 2), (Z, 1)])
by eval

lemma lookup (sparse0 [(X, 2::nat), (Z, 3)]) X = 2
by eval

lemma deg-pm (sparse0 [(X, 2::nat), (Y, 1), (Z, 3), (X, 1)]) = 6
by eval

lemma leq-pm (sparse0 [(X, 2::nat), (Y, 1), (Z, 3)]) (sparse0 [(X, 4)])
lemmas

```
lemma by eval
lex-pm (sparse \((X, 2::nat), (Y, 1), (Z, 3)\)) (sparse \((X, 4)\))
by eval

lemma by eval
¬ (dlex-pm (sparse \((X, 2::nat), (Y, 1), (Z, 3)\)) (sparse \((X, 4)\)))
by eval

lemma by eval
dlex-pm (sparse \((X, 2::nat), (Y, 1), (Z, 2)\)) (sparse \((X, 5)\))
by eval

lemma by eval
¬ (drlex-pm (sparse \((X, 2::nat), (Y, 1), (Z, 2)\)) (sparse \((X, 5)\)))
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_def fmlookup-default_def lookup-single)

lemma compute-one-poly-mapping [code]: \(1 = \text{sparse} \((0, 1)\)\)
by (metis compute-monomial single-one zero-neq-one)

lemma compute-except-poly-mapping [code]:
except (Pm-fmap xs) S = Pm-fmap (fmfilter (\(\lambda k. k \notin S\)) xs)
by (auto simp: fmlookup-default-default fmlookup-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-default fmlookup-of-list-splits: if-splits option.splits)

lemma if-poly-mapping-eq-iff:
(if \(x = y\) then \(a\) else \(b\)) =
(if \(\forall i\in\text{keys} x \cup \text{keys} y. \text{lookup} x i = \text{lookup} y i\) then \(a\) else \(b\))
by simp (metis UnI1 UnI2 in-keys-iff poly-mapping-eqI)

lemma keys-add-eq: \(\text{keys} (a + b) = \text{keys} a \cup \text{keys} b - \{ x \in \text{keys} a \cap \text{keys} b. \text{lookup} a x + \text{lookup} b x = 0\}\)
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 λt 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 (λx. 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 (λx. 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

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: fnlookup-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 fnempty)
proof (split list.splits, simp, intro conjI impI allI, goal-cases)
case (1 t c ys)
have Pm-fmap (fnupd t c (fmap-of-list ys)) = sparse0 [(t, c)] + except (sparse0 ys) {t}
  by (auto simp add: sparse0-def fnlookup-default-def lookup-add lookup-except
    split: option.splits intro!: poly-mapping-eqI)
also have sparse0 [(t, c)] = monomial c t
  by (auto simp: sparse0-def lookup-single fnlookup-default-def intro!: poly-mapping-eqI)
finally show ?case
  by (simp add: algebra-simps mult-scalar-monomial sparse0-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⇒nat) = Const0 c * monomial1 pp
for c::'b::comm-semiring-1
by (auto intro!: poly-mapping-eqI simp: monomial1-def Const0-def mult-single)

lemma monomial1-add: (monomial1 (a + b)::('a::monoid-add⇒0,'b::comm-semiring-1))
  = monomial1 a * monomial1 b
by (auto simp: monomial1-def mult-single)

lemma monomial1-monomial: monomial1 (monomial n v) = (Var0 v::⇒0('b::comm-semiring-1)) ^n
by (auto intro!: poly-mapping-eqI simp: monomial1-def Var0-power lookup-single
  when-def)
**lemma** Ball-True: \( \forall x \in X. \text{True} \) \( \leftrightarrow \) True by auto

**lemma** Collect-False: \([x. \text{False}] = \{\}\ by \ simp

**lemma** Pm-fmap-sum: Pm-fmap \( f = (\sum x \in \text{fdom}' f. \text{monomial (lookup0 f x)} x) \)

including fmap.lifting
by (auto intro: poly-mapping-eqI sum.neutral
  simp: fnlookup-default-def lookup-sum lookup-single when-def fdom'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
fdom'-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
monomialI-add
monomialI-monomial
Const0-one Const0-zero Const0-numeral Const0-minus
set-simps

A simproc for postprocessing with mpoly-simps and not polluting [code-post]:

**ML** (val mpoly-simproc = Simplifier.make-simproc @{context} multivariate polynomials
  {lhsss = @{term Pm-fmap mpp::(- \( \Rightarrow \) 0 nat) \( \Rightarrow \) 0 -}],
  proc = (K (fn ctxt => fn ct =>
        SOME (Simplifier.rewrite (put-simpset HOL-basic-ss ctxt addsimps

246
11.2.3 Ordered Power-Products

lemma foldl-assoc:
assumes \( \forall x, y, z. f (f x y) z = f x (f y z) \)
shows \( \text{foldl } f (f a b) \text{ xs} = f a (\text{foldl } f b \text{ xs}) \)
proof (induct xs arbitrary: a b)
  fix a b
  show \( \text{foldl } f (f a b) [] = f a (\text{foldl } f b []) \) by simp
next
  fix a b x xs
  assume \( \forall a, b. \text{foldl } f (f a b) \text{ xs} = f a (\text{foldl } f b \text{ xs}) \)
  from assms[of a b x] this[of a f b x]
  show \( \text{foldl } f (f a b) (x \# xs) = f a (\text{foldl } f b (x \# xs)) \) unfolding foldl-Cons
  by simp
qed

context ordered-term
begin

definition list-max :: 'a list ⇒ 'a 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 \( \text{foldl } \text{ord-term-lin.max} (\text{ord-term-lin.max} x \text{ min-term}) \text{ xs} = \)
    \( \text{ord-term-lin.max} x (\text{foldl } \text{ord-term-lin.max} \text{ min-term} \text{ xs}) \)
  by (rule foldl-assoc, rule ord-term-lin.max.assoc)
  from this ord-term-lin.max.commute[of min-term x]
  show \( \text{foldl } \text{ord-term-lin.max} (\text{ord-term-lin.max} x \text{ min-term} \text{ x}) \text{ xs} = \)
    \( \text{ord-term-lin.max} x (\text{foldl } \text{ord-term-lin.max} \text{ min-term} \text{ xs}) \) by simp
qed

lemma list-max-empty: list-max [] = min-term
unfolding list-max-def by simp

lemma list-max-in-list:
assumes xs \( \neq [] \)
shows list-max xs \( \in \text{ set } \text{xs} \)
using assms
proof (induct xs, simp)
  fix x xs
  assume IH: xs \( \neq [] \) \( \rightarrow \) list-max xs \( \in \text{ set } \text{ xs} \)
  show list-max (x # xs) \( \in \text{ set } (x \# \text{ xs}) \)
  proof (cases xs = [])
    case True
    hence list-max (x # xs) = ord-term-lin.max min-term x unfolding list-max-def

247
by simp
also have \ldots = \mathit{x} unfolding \textit{ord-term-lin.max-def} by (simp add: \textit{min-term-min})
finally show \textit{thesis} by simp
next
assume \(xs \neq []\)
show \textit{thesis}
proof (cases \(x \sqsubseteq \mathit{list-max} \, xs\))
  case True
  hence \(\mathit{list-max} \, (x \# \, xs) = \mathit{list-max} \, xs\)
  unfolding \textit{list-max-Cons ord-term-lin.max-def} by simp
  thus \textit{thesis} using IH\[\text{OF } \langle \mathit{xs} \neq [] \rangle\] by simp
next
  case False
  hence \(\mathit{list-max} \, (x \# \, xs) = \mathit{x}\)
  unfolding \textit{list-max-Cons ord-term-lin.max-def}
  by simp
  thus \textit{thesis} by simp
qed
qed
lemma \textit{list-max-maximum}:
  assumes \(a \in \textit{set} \, xs\)
  shows \(a \sqsubseteq \mathit{list-max} \, \mathit{xs}\)
  using \textit{assms}
proof (induct \(xs\))
  assume \(a \in \textit{set} \, []\)
  thus \(a \sqsubseteq \mathit{list-max} \, []\) by simp
next
  fix \(x \, \mathit{xs}\)
  assume \(\mathit{IH}: a \in \textit{set} \, \mathit{xs} \Longrightarrow a \sqsubseteq \mathit{list-max} \, \mathit{xs}\) and \(\mathit{a-in}: a \in \textit{set} \, (x \neq \, \mathit{xs})\)
  from \textit{a-in} have \(a = x \lor a \in \textit{set} \, \mathit{xs}\) by simp
  thus \(a \sqsubseteq \mathit{list-max} \, (x \neq \, \mathit{xs})\) unfolding \textit{list-max-Cons}
  proof
    assume \(a = x\)
    thus \(a \sqsubseteq \mathit{ord-term-lin.max} \, x \, (\mathit{list-max} \, \mathit{xs})\) by simp
next
  assume \(a \in \textit{set} \, \mathit{xs}\)
  from \(\mathit{IH}[\text{OF this}]\) show \(a \sqsubseteq \mathit{ord-term-lin.max} \, x \, (\mathit{list-max} \, \mathit{xs})\)
  by (simp add: \textit{ord-term-lin.le-max-iff-disj})
qed
qed
lemma \textit{list-max-nonempty}:
  assumes \(xs \neq []\)
  shows \(\mathit{list-max} \, \mathit{xs} = \mathit{ord-term-lin.Max} \, \textit{(set} \, \mathit{xs})\)
proof
  have fin: \textit{finite} \,(\textit{set} \, \mathit{xs}) by simp
  have \textit{ord-term-lin.Max} \,(\textit{set} \, \mathit{xs}) = \mathit{list-max} \, \mathit{xs}\nproof (rule \textit{ord-term-lin.Max-eql}[OF \textit{fin}, of \textit{list-max} \, \mathit{xs}]])
fix y
assume y ∈ set xs
from list-max-maximum[OF this] show y ≤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 fnfilter-alt-defs(2)
fnfilter-cong' mem-Collect-eq)

lemma compute-lt-poly-mapping[code]:
l t (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 (fnfilter (λk. 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 (fnfilter (λk. k ≺t k) xs)
unfolding lower-def compute-except-poly-mapping
by (metis mem-Collect-eq ord-term-lin.leD ord-term-lin.leI)
end
11.3 Computations

11.3.1 Scalar Polynomials

type-synonym 'a mpoly-tc = (nat ⇒ 0 => nat) ⇒ 0 => 'a

definition shift-map-keys-punit = term-powerprod.shift-map-keys to-pair-unit fst

lemma compute-shift-map-keys-punit [code]:
  shift-map-keys-punit t f (fmap-of-list xs) = fmap-of-list (map (λ(k, v). (t + k, f v)) xs)
  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 = (λx. 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 trivariate0-rat
begin

abbreviation X::rat mpoly-tc where X ≡ Var₀ (0::nat)
abbreviation Y::rat mpoly-tc where Y ≡ Var₀ (1::nat)
abbreviation Z::rat mpoly-tc where Z ≡ Var₀ (2::nat)

end
locale trivariate
begin

abbreviation X ≡ Var 0
abbreviation Y ≡ Var 1
abbreviation Z ≡ Var 2

end

experiment begin interpretation trivariate0-rat .

lemma
keys \((X^2 \cdot Z^7 + 2 \cdot Y^3 \cdot Z^2)\) =
\{monomial 2 0 + monomial 3 2, monomial 3 1 + monomial 2 2\}
by eval

lemma
keys \((X^2 \cdot Z^7 + 2 \cdot Y^3 \cdot Z^2)\) =
\{monomial 2 0 + monomial 3 2, monomial 3 1 + monomial 2 2\}
by eval

lemma
\(-1 \cdot X^2 \cdot Z^7 + -2 \cdot Y^3 \cdot Z^2 = -X^2 \cdot Z^7 + -2 \cdot Y^3 \cdot Z^2\)
by eval

lemma
\(X^2 \cdot Z^7 + 2 \cdot Y^3 \cdot Z^2 + X^2 \cdot Z^4 + -2 \cdot Y^3 \cdot Z^2 = X^2 \cdot Z^7 + X^2 \cdot Z^4\)
by eval

lemma
\(X^2 \cdot Z^7 + 2 \cdot Y^3 \cdot Z^2 - X^2 \cdot Z^4 + -2 \cdot Y^3 \cdot Z^2 = X^2 \cdot Z^7 - X^2 \cdot Z^4\)
by eval

lemma
lookup \((X^2 \cdot Z^7 + 2 \cdot Y^3 \cdot Z^2 + 2)\) (sparse0 [(0, 2), (2, 7)]) = 1
by eval

lemma
\(X^2 \cdot Z^7 + 2 \cdot Y^3 \cdot Z^2 \neq\)
\(X^2 \cdot Z^4 + -2 \cdot Y^3 \cdot Z^2\)
by eval

lemma
\(0 \cdot X^2 \cdot Z^7 + 0 \cdot Y^3 \cdot Z^2 = 0\)
by eval
lemma

\textbf{monom-mul-punit 3} (\texttt{[sparse0 \[(1, 2::nat)\]]}) \( (X^2 \ast Z + 2 \ast Y \ast 3 \ast Z^2) = 3 \ast Y^2 \ast Z \ast X^2 + 6 \ast Y \ast 5 \ast Z^2 \)

\textbf{by eval}

lemma

\textbf{monomial} \((-4) \ (\texttt{sparse0 \[(0, 2::nat)\]}) = - 4 \ast X^2 \)

\textbf{by eval}

lemma \textbf{monomial} \((0::rat) \ (\texttt{sparse0 \[(0::nat, 2::nat)\]}) = 0 \)

\textbf{by eval}

lemma

\( (X^2 \ast Z + 2 \ast Y \ast 3 \ast Z^2) \ast (X^2 \ast Z \ast 3 - 2 \ast Y \ast 3 \ast Z^2) = X \ast 4 \ast Z \ast 4 + 2 \ast X^2 \ast Z \ast 3 \ast Y \ast 3 + \)

\(- 4 \ast Y \ast 6 \ast Z \ast 4 + 2 \ast Y \ast 3 \ast Z \ast 5 \ast X^2 \)

\textbf{by eval}

end

11.3.2 Vector-Polynomials

\textbf{type-synonym} \( 'a \ vmpoly-tc = (\texttt{(nat \Rightarrow 0 nat)} \times \texttt{nat}) \Rightarrow 0 'a \)

\textbf{definition} \texttt{shift-map-keys-pprod = pprod.shift-map-keys}

\textbf{global-interpretation} \( pprod: \texttt{term-powerprod \ \lambda x. x \ \lambda x. x} \)

\textbf{rewrites} \( \texttt{pprod.pp-of-term = fst} \)

\textbf{and} \( \texttt{pprod.component-of-term = snd} \)

\textbf{defines} \( \texttt{splus-pprod = pprod.splus} \)

\textbf{and} \( \texttt{monom-mul-pprod = pprod.monom-mul} \)

\textbf{and} \( \texttt{mult-scalar-pprod = pprod.mult-scalar} \)

\textbf{and} \( \texttt{adds-term-pprod = pprod.adds-term} \)

\textbf{apply} \( \texttt{(fact MPoly-Type-Class.pprod.term-powerprod-axioms)} \)

\textbf{apply} \( \texttt{(fact MPoly-Type-Class.pprod.pp-of-term)} \)

\textbf{apply} \( \texttt{(fact MPoly-Type-Class.pprod.component-of-term)} \)

\textbf{done}

lemma \textbf{compute-adds-term-pprod} [\texttt{code-unfold}]:

\( \texttt{adds-term-pprod u v = (snd u = snd v \land adds-pp-add-linorder (fst u) (fst v))} \)

\textbf{by} \( \texttt{(simp add: adds-term-pprod-def pprod.adds-term-def adds-pp-add-linorder-def)} \)

lemma \textbf{compute-splus-pprod} [\texttt{code}]: \( \texttt{splus-pprod t (s, i) = (t + s, i)} \)

\textbf{by} \( \texttt{(simp add: splus-pprod-def pprod.splus-def)} \)

lemma \textbf{compute-shift-map-keys-pprod} [\texttt{code}]:

\( \texttt{shift-map-keys-pprod t f (fmap-of-list xs) = fmap-of-list (map (\lambda(k, v). (splus-pprod t k, f v)) xs)} \)

\textbf{by} \( \texttt{(simp add: pprod.compute-shift-map-keys shift-map-keys-pprod-def splus-pprod-def)} \)

252
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.computer-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 :: nat ⇒ ((a ⇒ nat) × nat) ⇒ (a × nat) where
Vec 0 i p = mult-scalar-pprod p (Poly-Mapping.single (0, i) 1)
experiment begin interpretation trivariate0-rat .

lemma keys (Vec 0 0 (X^2 * Z^3) + Vec 0 1 (2 * Y^3 * Z^2)) =
{(sparse [(0, 2), (2, 3)], 0), (sparse [(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, 2), (2, 3)], 0), (sparse [(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, 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, 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

253
lemma
  monom-mult-pprod 3 (sparse0 [(1, 2::nat)]) (Vec0 0 (X^2 * Z)) + Vec0 1 (2 * Y ^ 3 * Z^2)) =
  Vec0 0 (3 * Y^2 * Z * X^2) + Vec0 1 (6 * Y ^ 5 * Z^2)
by eval

11.4 Code setup for type MPol

postprocessing from Var0, Const0 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^2 * Z^3 + 6 * X^2 * Y^4 + 8 * X^2 * Y^5)
= (2::int, 2 * X * Y^2 * Z ^ 3 + 3 * X^2 * Y ^ 4 + 4 * X^2 * Y ^ 5)
by eval

end

done

theory PP-Type
  imports Power-Products
begin

  For code generation, we must introduce a copy of type 'a ⇒0 'b for power-products.
typedef (overloaded) ('a, 'b) pp = UNIV::{'a ⇒ 'b} set
morphisms mapping-of PP ..

setup-lifting type-definition-pp

lift-definition pp-of-fun :: ('a ⇒ 'b) ⇒ ('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 ⇒ 'a ⇒ 'b is lookup .

lift-definition keys-pp :: ('a, 'b::zero) pp ⇒ 'a set is keys .

lift-definition single-pp :: ('a ⇒ 'b) ⇒ ('a, 'b::zero) pp is Poly-Mapping.single .

lemma lookup-pp-of-fun: finite {x. f x ≠ 0} ⇒ 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: (∏ u. lookup-pp s u = lookup-pp t u) ⇒ s = t
by (transfer, rule poly-mapping-eqI)

lemma pp-eq-iff: (s = t) ⇔ (lookup-pp s = lookup-pp t)
by (auto intro: pp-eqI)

lemma keys-pp-iff: x ∈ keys-pp t ⇔ (lookup-pp t x ≠ 0)

lemma pp-eqI!:
 assumes ∏ u. u ∈ keys-pp s ∪ keys-pp t ⇒ 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 ∈ keys-pp s ∪ 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 \( \text{pp} :: (\text{type}, \text{zero}) \text{zero} \)

begin

lift-definition \( \text{zero-pp} :: (\text{'}a, \text{'}b) \text{pp} \Rightarrow 0::(\text{'}a \Rightarrow_0 \text{'}b) \).

lemma lookup-zero-pp [simp]: \( \text{lookup-pp} \ 0 = 0 \)
by (transfer, simp add: lookup-zero-fun)

instance ..

end

lemma single-pp-zero [simp]: \( \text{single-pp} \ x \ 0 = 0 \)
by (rule \( \text{pp-eqI} \), simp add: lookup-single-pp)

instantiation \( \text{pp} :: (\text{type}, \text{monoid-add}) \text{monoid-add} \)

begin

lift-definition \( \text{plus-pp} :: (\text{'}a, \text{'}b) \text{pp} \Rightarrow (\text{'}a, \text{'}b) \text{pp} \Rightarrow (\text{'}a, \text{'}b) \text{pp} \Rightarrow (+)::(\text{'}a \Rightarrow_0 \text{'}b) \Rightarrow \ -. \)

lemma lookup-plus-pp: \( \text{lookup-pp} \ (s + t) = \text{lookup-pp} \ s + \text{lookup-pp} \ t \)
by (transfer, simp add: lookup-plus-fun)

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

end

lemma single-pp-plus: \( \text{single-pp} \ x \ a + \text{single-pp} \ x \ b = \text{single-pp} \ x \ (a + b) \)
by (rule \( \text{pp-eqI} \), simp add: lookup-single-pp lookup-plus-pp when-def)

instance \( \text{pp} :: (\text{type}, \text{comm-monoid-add}) \text{comm-monoid-add} \)
by intro-classes (transfer, simp add: fun-eq-iff ac-simps+)

instantiation \( \text{pp} :: (\text{type}, \text{cancel-comm-monoid-add}) \text{cancel-comm-monoid-add} \)

begin

lift-definition \( \text{minus-pp} :: (\text{'}a, \text{'}b) \text{pp} \Rightarrow (\text{'}a, \text{'}b) \text{pp} \Rightarrow (\text{'}a, \text{'}b) \text{pp} \ \text{is} \ (-)::(\text{'}a \Rightarrow_0 \text{'}b) \Rightarrow \ -. \)

lemma lookup-minus-pp: \( \text{lookup-pp} \ (s - t) = \text{lookup-pp} \ s - \text{lookup-pp} \ t \)
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 \textit{comm-powerprod}

instance \text{poly-mapping} :: \text{(type, cancel-comm-monoid-add)} \text{ comm-powerprod}
   by standard

11.8 \( 'a \Rightarrow_0 'b \) belongs to class \textit{ninv-comm-monoid-add}

instance \text{poly-mapping} :: \text{(type, ninv-comm-monoid-add)} \text{ ninv-comm-monoid-add}
proof (standard, transfer)
   fix \( s \; t :: 'a \Rightarrow_0 'b \)
   assume \( \lambda \_ k. \; s \; k \; + \; t \; k = (\_ \Rightarrow_0 0) \)
   hence \( s \; + \; t = 0 \) by (simp only: plus-fun-def zero-fun-def)
   hence \( s = 0 \) by (rule plus-eq-zero)
   thus \( s = (\_ \Rightarrow_0 0) \) by (simp only: zero-fun-def)
qed

11.9 \( ('a, \; 'b) \) pp belongs to class \textit{lcs-powerprod}

lemma \text{adds-pp-iff}: \( s \; \text{adds} \; t \) \( \leftrightarrow \) \( \text{mapping-of} \; s \; \text{adds} \; \text{mapping-of} \; t \)
unfolding \text{adds-def} by (transfer, fact refl)

instantiation pp :: \text{(type, add-linorder)} \text{ lcs-powerprod}
begin

lift-definition \text{lcs-pp} :: \( ('a, \; 'b) \) pp \Rightarrow \( ('a, \; 'b) \) pp is \text{lcs-powerprod-class.lcs}

lemma \text{lookup-lcs-pp}: \text{lookup-pp} \; (\text{lcs} \; s \; t) \; x = \text{max} \; (\text{lookup-pp} \; s \; x) \; (\text{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 \textit{ulcs-powerprod}

instance pp :: \text{(type, add-linorder-min)} \text{ ulcs-powerprod} by intro-classes (transfer, elim plus-eq-zero)

11.11 Dickson’s lemma for power-products in finitely many indeterminates

lemma \text{almost-full-on-pp-iff}:
   \text{almost-full-on} \; (\text{adds}) \; A \leftrightarrow \text{almost-full-on} \; (\text{adds}) \; (\text{mapping-of} \; 'A) \; (\text{is} \; ?l \leftrightarrow \; ?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 ($\lambda x y. mapping-of x$ 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 ⇒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 .

258
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::('a::ordered-comm-monoid-add) pp
  by (transfer, fact lex-pm-strict-alt)

lemma lex-pp-zero-min: lex-pp 0 s
  for s::('a::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::('a::{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::('a::{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)
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)
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 \implies 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 \Rightarrow ('a, 'b) pp \Rightarrow bool
is drlex-pm.

lift-definition drlex-pp-strict :: ('a::linorder, 'b::{ordered-comm-monoid-add, linorder})
pp \Rightarrow ('a, 'b) pp \Rightarrow bool
is drlex-pm-strict.

lemma drlex-pp-alt: drlex-pp s t \iff (deg-pp s < deg-pp t \lor (deg-pp s = deg-pp t \land 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 \implies drlex-pp t s \implies s = t
by (transfer, rule drlex-pm-antisym)

lemma drlex-pp-trans: drlex-pp s t \implies drlex-pp t u \implies drlex-pp s u
by (transfer, rule drlex-pm-trans)

lemma drlex-pp-lin: drlex-pp s t \lor drlex-pp t s
by (transfer, fact drlex-pm-lin)

corollary drlex-pp-strict-alt [code]: drlex-pp-strict s t = (\neg 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 \implies 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)
12 Associative Lists with Sorted Keys

theory OAlist
  imports Deriving.Comparator
begin

We define the type of ordered associative lists (oalist). An oalist is an associative list (i.e. a list of pairs) such that the keys are distinct and sorted wrt. some linear order relation, and no key is mapped to 0::'a. The latter invariant allows to implement various functions operating on oalists more efficiently.

The ordering of the keys in an oalist $xs$ is encoded as an additional parameter of $xs$. This means that oalists may be ordered wrt. different orderings, even if they are of the same type. Operations operating on more than one oalists, like $map2-val$, typically ensure that the orderings of their arguments are identical by re-ordering one argument wrt. the order relation of the other. This, however, implies that equality of order relations must be effectively decidable if executable code is to be generated.

12.1 Preliminaries

fun min-list-param :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a
where
min-list-param rel (x ≠ xs) = (case xs of [] ⇒ x | xs ⇒ (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
lemma min-list-param-minimal:
assumes transp rel and \( \forall x y. x \in \text{set} \; xs \implies y \in \text{set} \; xs \implies \text{rel} \; x \; y \lor \text{rel} \; y \; x \)
and \( z \in \text{set} \; xs \)
shows \( \text{rel} \; (\text{min-list-param rel} \; xs) \; z \)
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: \( z = x \lor z \in \text{set} \; xs \) by simp
have \( x \in \text{set} \; (x \# xs) \) by simp
hence disj2: \( \text{rel} \; x \; z \lor \text{rel} \; z \; x \) using Cons(3) by (rule Cons(2))
have \( \ast: \text{rel} \; (\text{min-list-param rel} \; xs) \; z \) if \( z \in \text{set} \; xs \) using - that
proof (rule Cons(1))
fix \( a \; b \)
assume \( a \in \text{set} \; xs \) and \( b \in \text{set} \; xs \)
hence \( a \in \text{set} \; (x \# xs) \) and \( b \in \text{set} \; (x \# xs) \) by simp-all
thus \( \text{rel} \; a \; b \lor \text{rel} \; b \; a \) 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 \( \text{rel} \; x \; z \) by simp
next
fix y ys
assume \( xs = y \# ys \) and \( \text{rel} \; x \; (\text{min-list-param rel} \; (y \# ys)) \)
hence \( \text{rel} \; x \; (\text{min-list-param rel} \; xs) \) by simp
from disj1 show \( \text{rel} \; x \; z \)
proof
  assume \( z = x \)
  thus \( \text{thesis} \) using disj2 by simp
next
  assume \( z \in \text{set} \; xs \)
  hence \( \text{rel} \; (\text{min-list-param rel} \; xs) \; z \) by (rule \( \ast \))
  with assms(1) \( \text{rel} \; x \; (\text{min-list-param rel} \; xs) \); show \( \text{thesis} \) by (rule transpD)
qed
next
fix y ys
assume \( xs: xs = y \# ys \) and \( \neg \text{rel} \; x \; (\text{min-list-param rel} \; (y \# ys)) \)
from disj1 show \( \text{rel} \; (\text{min-list-param rel} \; (y \# ys)) \; z \)
proof
  assume \( z = x \)
  have \( \text{min-list-param rel} \; (y \# ys) \in \text{set} \; (y \# ys) \) by (rule min-list-param-in,
  simp)

qed
hence min-list-param rel (y # ys) ∈ set (x # xs) by (simp add: xs)
with (x ∈ set (x # xs)) have rel z (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: x)
next
assume z ∈ set xs
hence rel (min-list-param rel xs) z by (rule *)
thus ?thesis by (simp only: xs)
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 antisym 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 antisymD)

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 (λx 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 bin: (x, y) ∈ r ∨ (y, x) ∈ 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) ∈ r → (y, x) ∈ r → x = y for x y
  by (meson (linear-order r) antisymD linear-order-on-def partial-order-on-def)
have trans: (x, y) ∈ r → (y, z) ∈ r → (x, z) ∈ 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 = (λx y. if (x, y) ∈ r then if (y, x) ∈ 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) ∈ r and (y, x) ∈ r
thus x = y by (rule antisym)
next
fix x y
assume (x, y) /∈ r
with lin show (y, x) ∈ r by blast
next
fix x y z
assume (y, z) /∈ r and (z, y) /∈ r
assume (x, y) ∈ r and (y, z) ∈ r
hence (x, z) ∈ r by (rule trans)
moreover have (z, x) /∈ r
proof
  assume (z, x) ∈ r
  with (x, z) ∈ r! have x = z by (rule antisym)
  from (z, y) /∈ r ⟨(x, y) ∈ r⟩ show False unfolding (x = z) ..
qed
ultimately show (z, x) /∈ r ∧ ((z, x) /∈ r → (x, z) ∈ r) by simp
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 antisym 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-def 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 antisymD)
  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 antisymD)
  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)

267
12.3 Invariant in Context comparator

countex comparator
begun

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-cone 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-cone 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-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 \( \forall a. \text{snd} \ (f \ a) = 0 \implies \text{snd} \ a = 0 \)
  and \( \forall a \ b. \text{comp} \ (\text{fst} \ (f \ a)) \ (\text{fst} \ (f \ b)) = \text{comp} \ (\text{fst} \ a) \ (\text{fst} \ b) \)
  shows oalist-inv-raw (map f xs)
proof (rule oalist-inv-rawI)
  show \( 0 \notin \text{snd} \ ' \text{set} \ (\text{map} \ f \ xs) \)
  proof (simp, rule)
    assume \( 0 \in \text{snd} \ ' \text{set} \ \text{xs} \)
    then obtain \( a \) where \( a \in \text{set} \ \text{xs} \) and \( \text{snd} \ (f \ a) = 0 \) by fastforce
    from this(2) have \( \text{snd} \ a = 0 \) by (rule assms(2))
    from assms(1) have \( 0 \notin \text{snd} \ ' \text{set} \ \text{xs} \) by (rule oalist-inv-rawD1)
    moreover from \( \langle a \in \text{set} \ \text{xs} \rangle \) have \( 0 \in \text{snd} \ ' \text{set} \ \text{xs} \) by (simp add: \( \langle \text{snd} \ a = 0 \rangle \) [symmetric])
    ultimately show False ..
  qed
next
  from assms(1) have sorted-wrt lt (map fst xs) by (rule oalist-inv-rawD2)
  hence sorted-wrt (\( \lambda x \ y. \text{comp} \ (\text{fst} \ x) \ (\text{fst} \ y) = \text{Lt} \) \ \text{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 \( \forall k \ v \ \text{xs}. \ \text{oalist-inv-raw} \ ((k, v) \ # \ \text{xs}) \implies \text{oalist-inv-raw} \ \text{xs} \implies v \neq 0 \)
  \implies
    \( \forall k', k' \in \text{fst} \ ' \text{set} \ \text{xs} \implies \text{lt} \ k \ k' \implies P \ \text{xs} \implies P \ ((k, v) \ # \ \text{xs}) \)
  shows \( P \ \text{xs} \)
  using assms(1)
proof (induct xs)
  case Nil
  from assms(2) show ?case .
next
  case (Cons x xs)
  obtain \( k \ v \) where \( x = (k, v) \) 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 \( \text{lt} \ k \ k' \text{ if} k' \in \text{fst} \ ' \text{set} \ \text{xs} \text{ for} k' \text{ using} \) that
    unfolding x by (rule oalist-inv-raw-ConsD3)
  moreover from oalist-inv-raw xs have \( P \ \text{xs} \) by (rule Cons(1))
ultimately show \( \text{case unfolding } x \) by (rule \text{assms(3)})

\[ \text{qed} \]

12.4 Operations on Lists of Pairs in Context \( \text{comparator} \)

- **type-synonym** \( \text{(in \(-\) \( \{a, \ b\} \ \text{comp-opt} \Rightarrow a \Rightarrow b \Rightarrow \text{(order option)} \))} \)

- **definition** \( \text{(in \(-\) \( \{a \times b\} \ \text{list} \Rightarrow a \Rightarrow b::\text{zero} \))} \)
  - \( \text{where} \) \( \text{lookup-dflt xs k} = \{(\text{case map-of xs k of Some v \Rightarrow v | None \Rightarrow 0})\} \)
  - \( \text{lookup-dflt} \) is only an auxiliary function needed for proving some lemmas.

- **fun** \( \text{lookup-pair :: } \{a \times b\} \ \text{list} \Rightarrow a \Rightarrow b::\text{zero} \)
  - \( \text{where} \)
    - \( \text{lookup-pair } [] x = 0 | \)
    - \( \text{lookup-pair } ((k, v) \ # xs) x = \)
      - \( \text{(case comp x k of} \)
      - \( \text{Lt} \Rightarrow 0 \)
      - \( \text{| Eq} \Rightarrow v \)
      - \( \text{| Gt} \Rightarrow \text{lookup-pair xs x} \)

- **fun** \( \text{update-by-pair :: } \{a \times b\} \Rightarrow \{a \times b::\text{zero}\} \ \text{list} \)
  - \( \text{where} \)
    - \( \text{update-by-pair } (k, v) [] = (\text{if v = 0 then [] else } [(k, v)]) \)
    - \( \text{| update-by-pair } (k, v) ((k', v') \ # xs) = \)
      - \( \text{(case comp k k' of} \text{Lt} \Rightarrow (\text{if v = 0 then } (k', v') \ # xs \text{ else } (k, v) \ # (k', v') \ # xs) }\)
      - \( \text{| Eq} \Rightarrow (\text{if v = 0 then xs else } (k, v) \ # xs) \)
      - \( \text{| Gt} \Rightarrow (k', v') \ # \text{update-by-pair } (k, v) xs \)

- **definition** \( \text{sort-oalist :: } \{a \times b\} \ \text{list} \Rightarrow \{a \times b::\text{zero}\} \ \text{list} \)
  - \( \text{where} \) \( \text{sort-oalist xs = foldr update-by-pair xs []} \)

- **fun** \( \text{update-by-fun-pair :: } a \Rightarrow (b \Rightarrow \text{b::zero}) \Rightarrow \{a \times b::\text{zero}\} \ \text{list} \)
  - \( \text{where} \)
    - \( \text{update-by-fun-pair } k f [] = (\text{let v = f 0 in if v = 0 then [] else } [(k, v)]) \)
    - \( \text{| update-by-fun-pair } k f ((k', v') \ # xs) = \)
      - \( \text{(case comp k k' of} \text{Lt} \Rightarrow (\text{let v = f 0 in if v = 0 then } (k', v') \ # xs \text{ else } (k, v) \ # (k', v') \ # xs) }\)
      - \( \text{| Eq} \Rightarrow (\text{let v = f v' in if v = 0 then xs else } (k, v) \ # xs) \)
      - \( \text{| Gt} \Rightarrow (k', v') \ # \text{update-by-fun-pair } k f xs \)

- **definition** \( \text{update-by-fun-gr-pair :: } a \Rightarrow (b \Rightarrow \text{b::zero}) \Rightarrow \{a \times b\} \ \text{list} \Rightarrow \{a \times b::\text{zero}\} \ \text{list} \)
  - \( \text{where} \)
    - \( \text{update-by-fun-gr-pair } k f xs = \)
      - \( \text{(if xs = [] then} \)
      - \( \text{(let v = f 0 in if v = 0 then [] else } [(k, v)]) \)
      - \( \text{else if comp k (fst (last xs)) = Gt then} \)
      - \( \text{(let v = f 0 in if v = 0 then xs else xs @ [(k, v)])} \)
      - \( \text{else} \)

271
\[\text{update-by-fun-pair } k f xs\]

\[\text{fun } \text{(in } \Rightarrow \text{) } \text{map-pair } :: (('a \times 'b) \Rightarrow ('a \times 'c)) \Rightarrow ('a \times 'b;zero) \text{ list } \Rightarrow ('a \times 'c;zero) \text{ list} \]

\text{where}

\[\text{map-pair } f \; [] = []\]

\[\text{where} \quad \text{map-pair } f \; (k v \# x s) = \]

\[(\text{let } (k, v) = f k v; aux = \text{map-pair } f x s \text{ in if } v = 0 \text{ then aux else } (k, v) \# aux)\]

The difference between \text{map} and \text{map-pair} is that the latter removes 0::'b values, whereas the former does not.

\text{abbreviation} \ (\text{in } \Rightarrow \text{) } \text{map-val-pair } :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a \times 'b;zero) \text{ list } \Rightarrow ('a \times 'c;zero) \text{ list} \]

\text{where} \quad \text{map-val-pair } f \equiv \text{map-pair } (\lambda (k, v). (k, f k v))

\[\text{fun } \text{map2-val-pair } :: ('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd) \Rightarrow (('a \times 'b) \text{ list } \Rightarrow ('a \times 'd) \text{ list}) \Rightarrow (('a \times 'c) \text{ list } \Rightarrow ('a \times 'd) \text{ list}) \Rightarrow
\]

\[('a \times 'b;zero) \text{ list } \Rightarrow ('a \times 'd;zero) \text{ list}\]

\text{where}

\[\text{map2-val-pair } f \; g \; h \; x s \; [] = g \; x s\]

\[\text{where} \quad \text{map2-val-pair } f \; g \; h \; [] \; y s = h \; y s\]

\[\text{map2-val-pair } f \; g \; h \; ((k x, v x) \# x s) \; ((k y, v y) \# y s) = \]

\[(\text{case comp } k x k y \text{ of}
\]

\[\text{Lt } \Rightarrow (\text{let } v = f k x v x 0; aux = \text{map2-val-pair } f \; g \; h \; x \; s \; ((k y, v y) \# y s) \text{ in if } v = 0 \text{ then aux else } (k x, v) \# aux)
\]

\[| \quad \text{Eq } \Rightarrow (\text{let } v = f k x v x v y; aux = \text{map2-val-pair } f \; g \; h \; x \; s \; y s \text{ in if } v = 0 \text{ then aux else } (k x, v) \# aux)
\]

\[| \quad \text{Gt } \Rightarrow (\text{let } v = f k y v x 0; aux = \text{map2-val-pair } f \; g \; h \; (k x, v x) \# x s) \; y s \text{ in if } v = 0 \text{ then aux else } (k y, v) \# aux)\]

\[\text{fun } \text{lex-ord-pair } :: ('a \Rightarrow (('b, 'c) \text{ comp-opt})) \Rightarrow (('a \times 'b;zero) \text{ list}, ('a \times 'c;zero) \text{ list}) \text{ comp-opt}\]

\text{where}

\[\text{lex-ord-pair } f \; [] \; [] = \text{Some Eq}\]

\[\text{lex-ord-pair } f \; [] \; ((k y, v y) \# y s) = \]

\[(\text{let aux } = f k y 0 v y \text{ in if aux } = \text{Some Eq then lex-ord-pair } f \; [] \; y s \text{ else aux})\]

\[\text{lex-ord-pair } f \; ((k x, v x) \# x s) \; [] = \]

\[(\text{let aux } = f k x v x 0 \text{ in if aux } = \text{Some Eq then lex-ord-pair } f \; x \; s \; [] \text{ else aux})\]

\[\text{lex-ord-pair } f \; ((k x, v x) \# x s) \; ((k y, v y) \# y s) = \]

\[(\text{case comp } k x k y \text{ of}
\]

\[\text{Lt } \Rightarrow (\text{let aux } = f k x v x 0 \text{ in if aux } = \text{Some Eq then lex-ord-pair } f \; x \; s \; ((k y, v y) \# y s) \text{ else aux})
\]

\[| \quad \text{Eq } \Rightarrow (\text{let aux } = f k x v x v y \text{ in if aux } = \text{Some Eq then lex-ord-pair } f \; x \; s \; y s \text{ else aux})
\]

\[| \quad \text{Gt } \Rightarrow (\text{let aux } = f k y v x 0 \text{ in if aux } = \text{Some Eq then lex-ord-pair } f \; ((k x, v x) \# x s) \; y s \text{ else aux})\]

\[\text{fun } \text{prod-ord-pair } :: ('a \Rightarrow 'b \Rightarrow 'c \Rightarrow \text{ bool}) \Rightarrow ('a \times 'b;zero) \text{ list } \Rightarrow ('a \times 'c;zero)\]
list ⇒ bool

where

prod-ord-pair f [] [] = True | prod-ord-pair f [] ((ky, vy) # xs) = (f ky 0 vy ∧ prod-ord-pair f xs [])
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 ∈ 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 ≠ k' ∧ k /∈ fst ' set xs ..
    next
    assume comp k k' = Gt
    hence k ≠ k' by auto
    thus (lookup-pair xs k = 0) = (k ≠ k' ∧ k /∈ fst ' set xs) by (simp add: Cons(5))
      qed
  qed

lemma lookup-pair-eq-value:
assumes oalist-inv-raw xs and v ≠ 0
shows lookup-pair xs k = v ←→ ((k, v) ∈ 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 \neq 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) \in 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
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 ..
next
case (Cons k' v' xs)
  have v' = lookup-pair ((k', v') # xs) 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 ..
have $v' = \text{lookup-pair} \ ((k', v') \neq ys) \ k' \ by \ \text{simp}$
also have $... = \text{lookup-pair} \ ((k, v) \neq xs) \ k \ by \ (\text{simp only}: \text{Cons}(6) \ k' = k)$
also have $... = v \ by \ \text{simp}$
finally have $v' = v$.
moreover note $(k' = k)$
moreover from Cons(2) have $xs = ys$
proof (rule *(5))
  show $\text{lookup-pair} \ xs = \text{lookup-pair} \ ys$
  proof
    fix $k0$
    show $\text{lookup-pair} \ xs \ k0 = \text{lookup-pair} \ ys \ k0$
    proof (cases lt $k \ k0$)
      case True
      hence eq: $\text{comp} \ k0 \ k = \text{Gt}$
      by (simp add: Gt-lt-conv)
      have $\text{lookup-pair} \ xs \ k0 = \text{lookup-pair} \ ((k, v) \neq xs) \ k0 \ by \ (\text{simp add: eq})$
      also have $... = \text{lookup-pair} \ ((k, v') \neq ys) \ k0 \ by \ (\text{simp only}: \text{Cons}(6) \ k' = k)$
      also have $... = \text{lookup-pair} \ ys \ k0 \ by \ (\text{simp add: eq})$
      finally show $?thesis$ .
    next
      case False
      with *(4) have $k0 \notin \text{fst } set xs$ by blast
      with *(2) have eq: $\text{lookup-pair} \ xs \ k0 = 0 \ by \ (\text{simp add: lookup-pair-eq-0})$
      from False Cons(4) have $k0 \notin \text{fst } set ys$ unfolding $(k' = k)$ by blast
      with Cons(2) have $\text{lookup-pair} \ ys \ k0 = 0 \ by \ (\text{simp add: lookup-pair-eq-0})$
      with eq show $?thesis \ by \ \text{simp}$
    qed
    qed
    qed
    ultimately show $?thesis \ by \ \text{simp}$
  next
    case Gt
    hence $\neg \text{lt} \ k \ k'$ by (simp add: Gt-lt-conv)
    with *(4) have $k' \notin \text{fst } set xs$ by blast
    moreover from Gt have $k' \neq k \ by \ \text{auto}$
    ultimately have $k' \notin \text{fst } set ((k, v) \neq xs) \ k'$
    hence $0 = \text{lookup-pair} \ ((k, v) \neq xs) \ k'$
    by (simp add: lookup-pair-eq-0[of *(1)] del: lookup-pair.simps)
    also have $... = \text{lookup-pair} \ ((k', v') \neq ys) \ k' \ by \ (\text{simp only}: \text{Cons}(6))$
    also have $... = v' \ by \ \text{simp}$
    finally have $v' = 0 \ by \ \text{simp}$
    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
    from assms have 1:
    oalist-inv-raw (tl xs) by (rule oalist-inv-raw-tl)
  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
  proof
    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 only: 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
    proof
      have k'' = k' ∨ k'' ∈ fst ' set xs by (simp only: 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'' ∈ 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 only: 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 \( ?\text{thesis} \)

proof (split if-split, intro conjI impI)
  assume \( k \): \( k = \text{fst} (\text{hd} \, \text{xs}) \)
  show \( \text{lookup-pair} \, (\text{tl} \, \text{xs}) \, k = 0 \)
  proof (simp add: \text{lookup-pair-eq-0}[OF 1], rule)
    assume \( k\text{-in}: k \in \text{fst} \, \text{set} (\text{tl} \, \text{xs}) \)
    hence \( \text{xs} \neq [] \) by auto
    then obtain \( k' \, v' \, \text{ys} \) where \( \text{xs} = (k', v') \# \, \text{ys} \) using prod.exhaust
    list.exhaust by metis
    from \( \text{assms} \) have \( \text{oalist-inv-raw} \, ((k', v') \# \, \text{ys}) \) by (simp only: \text{xs})
    moreover from \( k\text{-in} \) have \( k' \in \text{fst} \, \text{set} \, \text{ys} \) by (simp add: \text{ks} \, \text{xs})
    ultimately have \( \text{lt} \, k \, k' \) by (rule oalist-inv-raw-ConsD3)
    thus \( \text{False} \) by simp
  qed
next
  assume \( k \neq \text{fst} (\text{hd} \, \text{xs}) \)
  show \( \text{lookup-pair} \, (\text{tl} \, \text{xs}) \, k = \text{lookup-pair} \, \text{xs} \, k \)
  proof (cases \( \text{xs} = [] \))
    case True
    show \( ?\text{thesis} \) by (simp add: True)
  next
    case False
    then obtain \( k' \, v' \, \text{ys} \) where \( \text{xs} = (k', v') \# \, \text{ys} \) using prod.exhaust
    list.exhaust by metis
    show \( ?\text{thesis} \)
    proof (simp add: \text{xs} eq Lt-lt-conv split: order.split, intro conjI impI)
      from \( k \neq \text{fst} (\text{hd} \, \text{xs}) \) have \( k \neq k' \) by (simp add: \text{xs})
      moreover assume \( k = k' \)
      ultimately show \( \text{lookup-pair} \, \text{ys} \, k' = v' \)
    next
    assume \( \text{lt} \, k \, k' \)
    from \( \text{assms} \) have \( \text{oalist-inv-raw} \, \text{ys} \) unfolding \( \text{xs} \) by (rule oalist-inv-raw-ConsD1)
    moreover have \( k \notin \text{fst} \, \text{set} \, \text{ys} \)
    proof
      assume \( k \in \text{fst} \, \text{set} \, \text{ys} \)
      with \( \text{assms} \) have \( \text{lt} \, k' \, k \) unfolding \( \text{xs} \) by (rule oalist-inv-raw-ConsD3)
      with \( \text{lt} \, k \, \text{ks} \) show \( \text{False} \) by simp
    qed
    ultimately show \( \text{lookup-pair} \, \text{ys} \, k = 0 \) by (simp add: \text{lookup-pair-eq-0})
  qed
  qed
  qed
  qed
  qed

lemma \text{lookup-pair-filter}:
  assumes \( \text{oalist-inv-raw} \, \text{xs} \)
  shows \( \text{lookup-pair} \, (\text{filter} \, P \, \text{xs}) \, k = (\text{let} \, v = \text{lookup-pair} \, \text{xs} \, k \, \text{in} \, \text{if} \, P \, (k, v) \, \text{then} \, v \, \text{else} \, 0) \)
  using \( \text{assms} \)

278
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
  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))
\[
v')\).
\]
\textbf{qed} (simp-all add: f eq\(\theta\) Cons(5))
\textbf{qed}

\textbf{lemma} lookup-pair-Cons:
\textbf{assumes} oalist-inv-raw \(((k, v) \# xs)\)
\textbf{shows} lookup-pair \(((k, v) \# xs) k\(\theta\) = (if \(k = k\(\theta\) then \(v\) else lookup-pair \(xs\ k\(\theta\)\))
\textbf{proof} (simp add: eq split: order.split, intro impI)
\textbf{assume} \(\text{comp } k\(\theta\) k = Lt\)
from \textbf{assms} \textbf{have} \(\text{inv: oalist-inv-raw } xs\) by (rule oalist-inv-raw-ConsD1)
\textbf{show} \(\text{lookup-pair } xs\ k\(\theta\) = 0\)
\textbf{proof} (simp only: lookup-pair-eq-0 [OF \text{inv}])
\textbf{assume} \(k\(\theta\) \in \text{fst ' set } xs\)
with \textbf{assms} \textbf{have} \(\text{lt } k k\(\theta\)\)
\textbf{using} \textbf{comp } k\(\theta\) k = Lt \textbf{show} \(\text{False}\) by (simp add: Lt-lt-conv)
\textbf{qed}
\textbf{qed}

\textbf{lemma} lookup-pair-single: lookup-pair \[[(k, v)]\] k\(\theta\) = (if \(k = k\(\theta\) then \(v\) else 0)
\textbf{by} (simp add: eq split: order.split)

12.4.2 \textbf{update-by-pair}

\textbf{lemma} set-update-by-pair-subset: set (update-by-pair \(kv\) \(xs\)) \(\subseteq\) insert \(kv\) (set \(xs\))
\textbf{proof} (induct \(xs\) arbitrary: \(kv\))
\textbf{case} Nil
\textbf{obtain} \(k v\) \textbf{where} \(kv: kv = (k, v)\) by fastforce
\textbf{thus} \(?\text{case}\) by simp
\textbf{next}
\textbf{case} (Cons \(x\) \(xs\))
\textbf{obtain} \(k' v'\) \textbf{where} \(x: x = (k', v')\) by fastforce
\textbf{obtain} \(k v\) \textbf{where} \(kv: kv = (k, v)\) by fastforce
\textbf{have} 1: \(set\ \(xs\) \(\subseteq\) insert a (insert b (set \(xs\)))\)
\textbf{for} \(a\ b\) \textbf{by} auto
\textbf{have} 2: \(set\ (update-by-pair\ kv\ xs) \(\subseteq\) insert kv (insert \(k'\ v')\ (set\ xs))\)
\textbf{for} \(kv\)
\textbf{using} Cons \textbf{by} blast
\textbf{show} \(?\text{case}\) by (simp add: x kv 1 2 split: order.split)
\textbf{qed}

\textbf{lemma} update-by-pair-sorted:
\textbf{assumes} sorted-wrt lt \(\text{map } \text{fst } xs\)
\textbf{shows} sorted-wrt lt \(\text{map } \text{fst} (\text{update-by-pair } kv\ xs)\)
\textbf{using} \textbf{assms}
\textbf{proof} (induct \(xs\) arbitrary: \(kv\))
\textbf{case} Nil
\textbf{obtain} \(k v\) \textbf{where} \(kv: kv = (k, v)\) by fastforce
\textbf{thus} \(?\text{case}\) by simp
\textbf{next}
\textbf{case} (Cons \(x\) \(xs\))
\textbf{obtain} \(k' v'\) \textbf{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 $(k' \# (map \text{fst} \; xs))$ by (simp add: $x$)
hence 2: sorted-wrt lt $(map \text{fst} \; xs)$ using sorted-wrt.elims(3) by fastforce
hence 3: sorted-wrt lt $(map \text{fst} \; (update-by-pair \; (k, u) \; xs))$ for $u$ by (rule Cons(1))
  have 4: sorted-wrt lt $(k' \# \; map \text{fst} \; (update-by-pair \; (k, u) \; xs))$
  if $*$: comp $k k' = \text{Gt}$ for $u$
  proof (simp, intro conjI ballI)
    fix $y$
    assume $y \in \text{set} \; (update-by-pair \; (k, u) \; xs)$
    also from set-update-by-pair-subset have ... $\subseteq \text{insert} \; (k, u) \; \text{(set} \; xs)$ .
    finally have $y = (k, u) \lor y \in \text{set} \; xs$ by simp
    thus $\text{lt} \; k' \; (\text{fst} \; y)$
  proof
    assume $y = (k, u)$
    hence $\text{fst} \; y = k$ by simp
    with $*$ show ?thesis by (simp only: Gt-lt-conv)
  next
    from 1 have 5: $\forall y \in \text{fst} \; (\text{set} \; xs) \; \text{lt} \; k' \; y$ by simp
    assume $y \in \text{set} \; xs$
    hence $\text{fst} \; y \in \text{fst} \; (\text{set} \; xs)$ by simp
    with 5 show ?thesis ..
  qed
  qed (fact 3)
show ?case
  by (simp add: $kv \; x \; 1 \; 2 \; 4 \; \text{sorted-wrt2 split: order.split del: sorted-wrt.simps, intro conjI impI, simp add: 1 eq del: sorted-wrt.simps, simp add: Lt-lt-conv}$)
qed

lemma update-by-pair-not-0:
  assumes 0 $\notin \; \text{snd} \; \text{'} \; \text{set} \; xs$
  shows 0 $\notin \; \text{snd} \; \text{'} \; \text{set} \; (update-by-pair \; kv \; xs)$
  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: $v' \neq 0$ and 2: 0 $\notin \text{snd} \; \text{'} \; \text{set} \; xs$ by (auto simp: $x$)
  from 2 have 3: 0 $\notin \text{snd} \; \text{'} \; \text{set} \; (update-by-pair \; (k, u) \; xs)$ for $u$ by (rule Cons(1))
  show ?case by (auto simp: $kv \; x \; 1 \; 2 \; 3 \; \text{split: order.split}$)
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 = (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 implI)
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 implI)
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)

282
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

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 k
    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

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 = (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: lookup-pair-eq-value[OF inv False, symmetric]
        lookup-pair-eq-value[OF assms(1) False]
        lookup-pair-update-by-pair[OF assms(1)])

283
lemma set-update-by-pair-zero:
assumes oalist-inv-raw xs
shows \( \text{set} \left( \text{update-by-pair} \ (k, 0) \ xs \right) = \text{set} \ xs - \text{range} \ (\text{Pair} \ k) \ (\text{is} \ ?A = ?B) \)
proof (rule set-eqI)
fix \( x :: 'a \times 'b \)
obtain \( k' \ v' \) where \( x = (k', v') \) 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 \( (x \in ?A) \iff (x \in ?B) \)
proof (cases \( v' = 0 \))
next
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 \( \text{update-by-fun-pair} \ k \ f \ xs = \text{update-by-pair} \ (k, f (\text{lookup-pair} \ xs \ k)) \ xs \)
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)
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)
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)) k' = Gt
        by (simp add: Gt-lt-conv)
      ultimately show ?thesis
        by (meson Gt-lt-conv less-trans Lt-lt-conv [symmetric])
    qed
  qed
  show ?case by (simp split: order.split add: Let-def eq1 eq2)
next
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
    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)
  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-gr-pair-eq-update-by-pair [OF assms]
12.4.4 map-pair

lemma map-pair-cong:
  assumes \( \forall kv. \; kv \in \text{set} \; xs \implies f \; kv = g \; kv \)
  shows \( \text{map-pair} \; f \; xs = \text{map-pair} \; g \; xs \)
  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 \( \text{map-pair} \; f \; xs = \text{map-pair} \; g \; xs \)
  by (rule Cons(1), rule Cons(2), simp)
  ultimately show ?case by simp
qed

lemma map-pair-subset: set (\text{map-pair} \; f \; xs) \subseteq f ' \; \text{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 \( f \): \( f \; (k, \; v) = (k', \; v') \) by fastforce
  from \( f \) symmetric refl have \( \ast \): set (\text{map-pair} \; f \; xs) \subseteq f ' \; \text{set} \; xs \) by (rule 2)
  show ?case by (simp add: f Let-def, intro conjI impI subset-insertI2 \ast)
qed

lemma oalist-inv-raw-map-pair:
  assumes oalist-inv-raw xs
  and \( \forall a \; b. \; \text{comp} \; (\text{fst} \; (f \; a)) \; (\text{fst} \; (f \; b)) = \text{comp} \; (\text{fst} \; a) \; (\text{fst} \; b) \)
  shows oalist-inv-raw (\text{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 \( v' \neq 0 \)
    with Cons(5) show oalist-inv-raw ((k', v') \# \text{map-pair} \; f \; xs)
  proof (rule oalist-inv-raw-ConsI)
    assume \( \text{map-pair} \; f \; xs \neq [] \)
    hence \( \text{hd} \; (\text{map-pair} \; f \; xs) \in \text{set} \; (\text{map-pair} \; f \; xs) \) by simp
  qed
  qed
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 (comp k) k 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 !!a b. comp (fst (f a)) (fst (f b)) = comp (fst a) (fst b)
sshows 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
    impl)
    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')) .

287
\textbf{lemma} lookup-dflt-map-pair:
\begin{itemize}
\item \textbf{assumes} distinct \((\text{map fst} \; \text{xs})\) \textbf{and} \(\text{snd} \; (f \; (k, \; 0)) = 0\)
\item and \(\bigwedge a \; b. \; (\text{fst} \; (f \; a) = \text{fst} \; (f \; b)) \iff (\text{fst} \; a = \text{fst} \; b)\)
\item \textbf{shows} \(\text{lookup-dflt} \; (\text{map-pair} \; f \; \text{xs}) \; (\text{fst} \; (f \; (k, \; v))) = \text{snd} \; (f \; (k, \; \text{lookup-dflt} \; \text{xs} \; k))\)
\end{itemize}
\textbf{using} \(\text{assms}(1)\)
\textbf{proof} (\textit{induct} \textit{xs})
\begin{itemize}
\item case Nil
\item \textbf{show} ?case by (\textit{simp add: lookup-dflt-def assms}(2))
\end{itemize}
\textbf{next}
\begin{itemize}
\item case (\textit{Cons} \textit{x} \textit{xs})
\item \textbf{obtain} \(k' \; v'\) \textbf{where} \(x : x = (k', \; v')\) by \textit{fastforce}
\item \textbf{obtain} \(k'' \; v''\) \textbf{where} \(f : (f \; (k', \; v')) = (k'', \; v'')\) by \textit{fastforce}
\item \textbf{from} \(\text{Cons}(2)\) \textbf{have} distinct \((\text{map} \; \text{fst} \; \text{xs})\) \textbf{and} \(k' \notin \text{fst} \; \text{set} \; \text{xs}\) by (\textit{simp-all add: x})
\item \textbf{from} \(\text{this}(1)\) \textbf{have} \(\text{eq1:} \; \text{lookup-dflt} \; (\text{map-pair} \; f \; \text{xs}) \; (\text{fst} \; (f \; (k, \; v))) = \text{snd} \; (f \; (k, \; \text{lookup-dflt} \; \text{xs} \; k))\)
\item by (\textit{rule Cons}(1))
\item \textbf{have} \(\text{eq2:} \; \text{lookup-dflt} \; ((a, \; b) \neq \text{ys}) \; c = (\text{if} \; c = a \; \text{then} \; b \; \text{else} \; \text{lookup-dflt} \; \text{ys} \; c)\)
\item \textbf{for} \(a \; b \; c \; \textbf{and} \; \textbf{ys} : (b \times 'c::\text{zero})\) \textbf{list} by (\textit{simp add: lookup-dflt-def map-of-Cons-code})
\item \textbf{from} \(k' \notin \text{fst} \; \text{set} \; \text{xs}\) \textbf{have} \(\text{map-of} \; \text{xs} \; k' = \text{None}\) by (\textit{simp add: map-of-eq-None-iff})
\item hence \(\text{eq3:} \; \text{lookup-dflt} \; \text{xs} \; k' = 0\) by (\textit{simp add: lookup-dflt-def})
\item \textbf{show} ?case
\item \textbf{proof} (\textit{simp add: f Let-def x eq1 eq2 eq3, intro conjI impI})
\item \textbf{assume} \(k = k'\)
\item hence \(\text{snd} \; (f \; (k', \; 0)) = \text{snd} \; (f \; (k, \; 0))\) by \textit{simp}
\item also \textbf{have} \(\ldots = 0\) by (\textit{fact assms}(2))
\item finally \textbf{show} \(\text{snd} \; (f \; (k', \; 0)) = 0\)
\item \textbf{next}
\item \textbf{assume} \(\text{fst} \; (f \; (k', \; v)) \neq k''\)
\item hence \(\text{fst} \; (f \; (k', \; v)) \neq \text{fst} \; (f \; (k', \; v'))\) by (\textit{simp add: f})
\item thus \(\text{snd} \; (f \; (k', \; 0)) = v''\) by (\textit{simp add: assms}(3))
\item \textbf{next}
\item \textbf{assume} \(k \neq k'\)
\item \textbf{assume} \(\text{fst} \; (f \; (k, \; v)) = k''\)
\item also \textbf{have} \(\ldots = \text{fst} \; (f \; (k', \; v'))\) by (\textit{simp add: f})
\item finally \textbf{have} \(k = k'\) by (\textit{simp add: assms}(3))
\item with \(k \neq k'\) \textbf{show} \(v'' = \text{snd} \; (f \; (k, \; \text{lookup-dflt} \; \text{xs} \; k))\)
\item \textbf{qed}
\item \textbf{qed}
\end{itemize}
\textbf{lemma} distinct-map-pair:
\begin{itemize}
\item \textbf{assumes} distinct \((\text{map} \; \text{fst} \; \text{xs})\) \textbf{and} \(\bigwedge a \; b. \; f \; (f \; a) = f \; (f \; b) \implies f \; a = f \; b\)
\item \textbf{shows} distinct \((\text{map} \; \text{fst} \; (\text{map-pair} \; f \; \text{xs}))\)
\end{itemize}
\textbf{using} \(\text{assms}(1)\)
\textbf{proof} (\textit{induct} \textit{xs})
\begin{itemize}
\item case Nil
\end{itemize}
\textbf{qed}
show ?case by simp

next
case (Cons x xs)
obtain k v where x = (k, v) by fastforce

obtain k' v' where 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' \∉ 0
  assume k' \in fst ' set (map-pair f xs)
  then obtain y where y \in set (map-pair f xs) and k' = fst y..

  from this(1) map-pair-subset have y \in f ' set xs ..
  then obtain z where z \in set xs and y = f z ..

  from this(2) have fst (f z) = k' by (simp add: \langle k' = fst y \rangle)

  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 \in fst ' set xs using (z \in set xs) by blast
  with \langle k \notin fst ' set xs \rangle show False ..
qued

qed

lemma map-val-pair-cong:
  assumes \( \forall k \in \text{ set } xs \Rightarrow f k v = g k v \)
  shows map-val-pair f xs = map-val-pair g xs
proof (rule map-pair-cong)
  fix kv

  assume kv \in set 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)
qued

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 = \lambda (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 \( \cdots = \text{snd} (\exists f. (k, \text{lookup-pair} \, \text{xs} \, k)) \)
by (rule lookup-pair-map-pair, fact assms(1), auto simp: assms(2))
also have \( \cdots = f \, k \) (lookup-pair \, \text{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
qed

12.4.5 map2-val-pair

definition map2-val-compat ::
(\'a \times \'b::zero) list \Rightarrow (\'a \times \'c::zero) list \Rightarrow bool
where map2-val-compat f \iff
(oalist-inv-raw (f zs) \Rightarrow
(oalist-inv-raw (f zs) \Rightarrow
(fst set (f zs) \subseteq fst set zs)))

lemma map2-val-compatI:
assumes \(?\forall zs. oalist-inv-raw zs \Rightarrow oalist-inv-raw (f zs)
and \(?\forall zs. oalist-inv-raw zs \Rightarrow fst set (f zs) \subseteq 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 fst set (f zs) \subseteq fst set 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) \subseteq fst set zs
using assms unfolding map2-val-compat-def by blast

lemma map2-val-compat-Nil:
assumes map2-val-compat (f::(\'a \times \'b::zero) list \Rightarrow (\'a \times \'c::zero) list)
shows f [] = []
proof
from assms oalist-inv-raw-Nil have fst set (f []) \subseteq fst set ([]::(\'a \times \'b) list)
by (rule map2-val-compatD2)
thus \(?thesis by simp
qed

290
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 \( \text{fst} \circ \text{set} (\text{map-val-pair} f \ zs) \subseteq \text{fst} \circ \text{set} \ zs \) 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 \( \text{fst} \circ \text{set} (\text{map2-val-pair} g \ h \ xs \ ys) \subseteq \text{fst} \circ \text{set} \ xs \cup \text{fst} \circ \text{set} \ ys \) using assms proof (induct f g h xs ys rule: map2-val-pair.induct) case (1 f g h xs) show \( ?\text{case} \) by (simp, rule map2-val-compatD2, fact+)

next case (2 f g h v va) show \( ?\text{case} \) 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 \( ?\text{case} \) proof (simp split: order.split, intro conjI impI) assume comp kx ky = Lt hence \( \text{fst} \circ \text{set} (\text{map2-val-pair} f \ g \ h \ xs \ ((\text{ky}, \ \text{vy}) \ # \ ys)) \subseteq \text{fst} \circ \text{set} \ xs \cup \text{fst} \circ \text{set} \ ((\text{ky}, \ \text{vy}) \ # \ ys) \) using refl \( \langle \text{alist-inv-raw} \ xs \rangle \ 3(5, 6, 7) \) by (rule 3(2)) thus \( \text{fst} \circ \text{set} \ (\text{let} \ v = f \ kx \ v0; \text{aux} = \text{map2-val-pair} f \ g \ h \ xs \ ((\text{ky}, \ \text{vy}) \ # \ ys) \) in if \( v = 0 \) then aux else \((\text{kx}, \ v) \ # \ aux) \subseteq \text{insert} \ \text{ky} \ (\text{insert} \ \text{kx} \ (\text{fst} \circ \text{set} \ xs \cup \text{fst} \circ \text{set} \ ys)) \) by (auto simp: Let-def)

next assume comp kx ky = Eq hence \( \text{fst} \circ \text{set} (\text{map2-val-pair} f \ g \ h \ xs \ ys) \subseteq \text{fst} \circ \text{set} \ xs \cup \text{fst} \circ \text{set} \ ys \) using refl \( \langle \text{alist-inv-raw} \ xs \rangle \ 3(6, 7) \) by (rule 3(1)) thus \( \text{fst} \circ \text{set} \ (\text{let} \ v = f \ kx \ vx; \text{aux} = \text{map2-val-pair} f \ g \ h \ xs \ ys \in \text{in} \) if \( v = 0 \) then aux else \((\text{kx}, \ v) \ # \ aux) \subseteq \text{insert} \ \text{ky} \ (\text{insert} \ \text{kx} \ (\text{fst} \circ \text{set} \ xs \cup \text{fst} \circ \text{set} \ ys)) \) by (auto simp: Let-def)

next assume comp kx ky = Gt hence \( \text{fst} \circ \text{set} (\text{map2-val-pair} f \ g \ h \ ((\text{kx}, \ \text{vx}) \ # \ xs) \ ys) \subseteq \text{fst} \circ \text{set} \ ((\text{kx}, \ \text{vx}) \ # \ xs) \cup \text{fst} \circ \text{set} \ ys \) using refl 3(4) \( \langle \text{alist-inv-raw} \ ys \rangle \ 3(6, 7) \) by (rule 3(3)) thus \( \text{fst} \circ \text{set} \ (\text{let} \ v = f \ ky \ v0; \text{aux} = \text{map2-val-pair} f \ g \ h \ ((\text{kx}, \ \text{vx}) \ # \ xs) \ ys \in \text{in} \) if \( v = 0 \) then aux else \((\text{ky}, \ v) \ # \ aux) \subseteq \text{insert} \ \text{ky} \ (\text{insert} \ \text{kx} \ (\text{fst} \circ \text{set} \ xs \cup \text{fst} \circ \text{set} \ ys)) \) by (auto simp: Let-def)
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 \((\text{map2-val-pair } f \ h \ x s \ y s)\)
  using assms(1, 2)
proof (induct xs arbitrary: ys rule: oalist-inv-raw-induct)
case Nil
  show \(?\) by (simp add: Nil, rule map2-val-compatD1, fact assms(3),
    fact oalist-inv-raw-Nil)
next
case (Cons y ys)
  show \(?\) 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 \(?\)
  proof (induct ys rule: oalist-inv-raw-induct)
    case Nil
    show \(?\) by (simp add: Cons, rule map2-val-compatD1, fact assms(3), fact *1))
next
case (Cons k' v' ys)
  show \(?\) by (simp split: order.split, intro conjI impI)
    assume \(\text{comp } k \ k' = \text{Lt}\)
    hence 0: \(\text{lt } k \ k'\) by (simp only: Lt-lt-conv)
    from Cons(1) have 1: oalist-inv-raw \((\text{map2-val-pair } f \ h \ x s ((k', \ v') \ # \ y s))\)
      by (rule *5))
    show oalist-inv-raw \((\text{let } v = f \ k \ v \ 0; \ aux = \text{map2-val-pair } f \ h \ x s ((k', \ v') \ # \ y s))\)
      in if \(v = 0\) then aux else \((k, \ v) \ # \ aux\)
      proof (simp add: Let-def, intro conjI impI)
        assume \(f \ k \ v \ 0 \neq \ 0\)
        with 1 show oalist-inv-raw \(((k, f \ k \ v \ 0) \ # \ \text{map2-val-pair } f \ h \ x s ((k', \ v') \ # \ y s))\)
      proof (rule oalist-inv-raw-ConsI)
        define \(k0\) where \(k0 = \text{fst } (hd (\text{local.map2-val-pair } f \ h \ x s ((k', \ v') \ # \ y s))))\)
        assume \(\text{map2-val-pair } f \ g \ x s ((k', \ v') \ # \ y s)\) \([\not= \]\)
        hence \(k0 \in \text{fst } ' \ \text{set} (\text{map2-val-pair } f \ h \ x s ((k', \ v') \ # \ y s))\) by (simp add: k0-def)
        also from *(2) Cons(1) assms(3, 4) have ... \subseteq \text{fst } ' \ \text{set } xs \cup \text{fst } ' \ \text{set} ((k', \ v') \ # \ y s)\)

292
by (rule fst-map2-val-pair-subset)
finally have \( k0 \in \text{fst ' set xs} \lor k0 = k' \lor k0 \in \text{fst ' set ys} \) by auto
thus \( \text{lt k k0} \)
proof (elim disjE)
  assume \( k0 = k' \)
  with \( 0 \) show \(?thesis\) by simp
next
  assume \( k0 \in \text{fst ' set ys} \)
  hence \( \text{lt k' k0} \) by (rule Cons(4))
  with \( 0 \) show \(?thesis\) by (rule less-trans)
qed (rule *(4))
qed (rule 1)

next
assume \( \text{comp k k'} = \text{Eq} \)
  hence \( k = k' \) by (simp only; eq)
from Cons(2) have \( 1: \text{oalist-inv-raw (map2-val-pair f g h xs ys)} \) by (rule *
\( *(5) \))
show oalist-inv-raw (let \( v = f k v v' \); aux = map2-val-pair f g h xs ys \( \text{in if v = 0 \ then aux else (k, v) \# aux} \))
proof (simp add; Let-def, intro conjI impl)
  assume \( f k v v' \neq 0 \)
  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 \( k0 = \text{fst (hd (map2-val-pair f g h xs ys))} \)
    assume \( \text{map2-val-pair f g h xs ys \neq []} \)
    hence \( k0 \in \text{fst ' set (map2-val-pair f g h xs ys)} \) by (simp add; k0-def)
    also from \( *\( (2) \) Cons(2) \) assms\( (3, 4) \) have \( \ldots \subseteq \text{fst ' set xs} \cup \text{fst ' set ys} \)
    by (rule \( \text{fst-map2-val-pair-subset} \))
finally show \( \text{lt k k0} \)
proof
  assume \( k0 \in \text{fst ' set ys} \)
  hence \( \text{lt k' k0} \) by (rule Cons(4))
  thus \(?thesis\) by (simp only; \( \langle k = k' \rangle \))
qed (rule *(4))
qed
qed (rule 1)

next
assume \( \text{comp k k'} = \text{Gt} \)
  hence \( \text{lt k' k} \) 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 \( \text{in if va = 0 \ then aux else (k', va) \# aux} \))
proof (simp add; Let-def, intro conjI impl)
  assume \( f k' 0 v' \neq 0 \)
  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 = \text{fst (hd (map2-val-pair f g h ((\( k, v \)# xs) ys))} \)

293
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
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 (rule Cons(5))
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 (λk v. f k v 0) zs
  and ∀zs. oalist-inv-raw zs =⇒ h zs = map-val-pair (λk. f k 0) zs
  and λk. 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 (λk. f k 0) ys) k0
    by (simp only: assms(6))
  also have ... = f k0 0 (lookup-pair ys k0) by (rule lookup-pair-map2-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: y)
  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 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-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 \( v = f \ k \ v \ o \); \( \text{aux} = \text{map2-val-pair} \ f \ g \ h \ x s \ ((k', v') \# y s) \) in if \( v = 0 \) then \( \text{aux} \) else \( (k, v) \# \text{aux} \)) \( k0 = f k0 \) (lookup-pair ((k, v) \# x s) k0) (lookup-pair ((k', v') \# y s) k0)

proof (cases \( \text{comp} k0 \) \( k \))
  case \( \text{Lt} \)
  with (\( \langle \text{comp} k0 \ k \neq \text{Lt} \rangle \)) show \( ?\text{thesis} .. \)
  next
  case \( \text{Eq} \)
  hence \( k0 = k \) by (simp only: eq)
  with (\( \langle \text{comp} k \ k' = \text{Lt} \rangle \)) have \( \text{comp} k0 \ k' = \text{Lt} \) by simp
  hence eq1: lookup-pair ((k', v') \# y s) \( k = 0 \) by (simp add: \( \langle k0 = k \rangle \))
  have eq2: lookup-pair ((k, v) \# x s) \( k = v \) by simp
  show \( ?\text{thesis} \)
  proof (simp add: Let-def eq1 eq2 del: lookup-pair.simps, intro conjI impI)
    from \( *(2) \) \( \text{Cons}(1) \) assms(3, 4) have inv: oalist-inv-raw (map2-val-pair f g h x s ((k', v') \# y s))
    by (rule oalist-inv-rue-map2-val-pair)
    show lookup-pair (map2-val-pair f g h x s ((k', v') \# y s)) \( k = 0 \)
    proof (simp only: lookup-pair-eq-0[OF inv], rule)
      assume \( k \in \text{fst \ ' set} \ (\text{local.map2-val-pair} \ f \ g \ h \ x s ((k', v') \# y s))) \( \)
      also from \( *(2) \) \( \text{Cons}(1) \) assms(3, 4) have ... \( \subseteq \text{fst \ ' set} \ x s \cup \text{fst \ ' set} (\ (k', v') \# y s) \)
      by (rule fst-map2-val-pair-subset)
      finally have \( k \in \text{fst \ ' set} \ x s \cup \text{fst \ ' set} \ y s \) using (\( \langle \text{comp} k \ k' = \text{Lt} \rangle \))
      by auto
      thus False
    proof
      assume \( k \in \text{fst \ ' set} \ x s \)
      hence \( \text{lt} k \ k \) by (rule *(4))
      thus \( ?\text{thesis} \) by simp
    next
      assume \( k \in \text{fst \ ' set} \ y s \)
      hence \( \text{lt} k \ k' \) by (rule \( \text{Cons}(4) \))
      with (\( \langle \text{comp} k \ k' = \text{Lt} \rangle \)) show \( ?\text{thesis} \) by (simp add: Lt-lt-conv)
    qed
    qed
    qed simp
  next
    case \( \text{Gt} \)
    hence eq1: lookup-pair ((k, v) \# x s) k0 = lookup-pair x s k0
    and eq2: lookup-pair ((k, f k v \ o) \# map2-val-pair f g h x s ((k', v') \# y s)) k0 =
    lookup-pair (map2-val-pair f g h x s ((k', v') \# y s)) k0 by simp
    show \( ?\text{thesis} \)
    by (simp add: Let-def eq1 eq2 del: lookup-pair.simps, rule *(5), fact \( \text{Cons}(1) \))
    qed

296
assume \( \text{comp } k \, k' = \text{Eq} \)

hence \( k = k' \) by (simp only: eq)

with \( \text{False} \) have \( \text{comp } k0 \, k' \neq \text{Lt} \) by (auto simp: Lt-lt-conv)

show lookup-pair \( (\text{let } v = f \, k \, v' \mid \, \text{aux } = \text{map2-val-pair } f \, g \, h \, x \, s \, y \, s \) \) in

if \( v = 0 \) then \( \text{aux } \) else \( (k, v) \# \, \text{aux} \) \( k0 = \)

\( f \, k0 \) (lookup-pair \( ((k, v) \# \, x \, s) \, k0) \) (lookup-pair \( ((k', v') \# \, y \, s) \, k0) \)

proof (cases \( \text{comp } k0 \, k' \))

case \( \text{Lt} \)

with \( \langle \text{comp } k0 \, k' \neq \text{Lt} \rangle \)

show \( ?\text{thesis} \) ..

next

case \( \text{Eq} \)

hence \( k0 = k' \) by (simp only: eq)

show \( ?\text{thesis} \)

proof (simp add: Let-def \( \langle k = k' \mid k0 = k', \text{intro impl} \rangle \) \( \langle \text{assms } (3, 4) \rangle \) have inv: oalist-inv-raw (map2-val-pair f g h x s y s)

by (rule oalist-inv-raw-map2-val-pair)

show lookup-pair (map2-val-pair f g h x s y s) \( k' = 0 \)

proof (simp only: lookup-pair-eq-0[OF inv], rule)

assume \( k' \in \text{fst } \text{set} \) (map2-val-pair f g h x s y s)

also from \( \ast(2) \) \( \text{Cons}(2) \) \( \text{assms}(3, 4) \) have \( \cdots \subseteq \text{fst } \text{set} \, x \in \text{fst } \text{set} \)

ys

by (rule fst-map2-val-pair-subset)

finally show \( \text{False} \)

proof

assume \( k' \in \text{fst } \text{set} \)

hence \( \text{lt } k \, k' \) by (rule Cons(4))

thus \( ?\text{thesis} \) by simp

next

assume \( k' \in \text{fst } \text{set} \)

hence \( \text{lt } k \, k' \) by (rule \( \ast(4) \))

thus \( ?\text{thesis} \) by (simp add: \( \langle k = k' \rangle \))

qed

qed

qed

next

case \( \text{Gt} \)

hence eq1: lookup-pair \( ((k, v) \# \, x \, s) \, k0 = \text{lookup-pair } x \, s \, k0 \)

and eq2: lookup-pair \( ((k', v') \# \, y \, s) \, k0 = \text{lookup-pair } y \, s \, k0 \)

and eq3: lookup-pair \( ((k, f \, k \, v \, v') \# \, \text{map2-val-pair } f \, g \, h \, x \, y \, s \) \) \( k0 = \)

lookup-pair \( \text{map2-val-pair } f \, g \, h \, x \, y \, s \) \( k0 \) by (simp-all add: \( k = k' \))

show \( ?\text{thesis} \) by (simp add: Let-def eq1 eq2 eq3 del: lookup-pair.simps, rule \( \ast(5) \), fact Cons(2))

qed

next

assume \( \text{comp } k \, k' = \text{Gt} \)

hence \( \text{comp } k' \, k = \text{Lt} \) 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 impl)
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

298
lemma map2-val-pair-singleton-eq-update-by-fun-pair:
assumes oalist-inv-raw xs
assumes \( k \in \text{fst } \text{set } xs \cup \text{fst } \text{set } ys \implies f k (\text{lookup-pair } xs k) = \text{lookup-pair } ys k \) = Some Eq
shows map2-val-pair f g xs \( [(k, v)] = \text{update-by-fun-pair } k (\lambda x. f k x v) \) 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) OF 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

12.4.6 lex-ord-pair
lemma lex-ord-pair-EqI:
assumes oalist-inv-raw xs and oalist-inv-raw ys
and \( k \in \text{fst } \text{set } xs \cup \text{fst } \text{set } ys \implies f k (\text{lookup-pair } xs k) = \text{lookup-pair } ys k \) = 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
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 split: order.split, intro conjI impI)
  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 Cons(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 0 v = 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 0 v = 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

300
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: Ltl-tl-cone) 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 (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 k v 0 = 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 \( \ldots = \text{Some Eq} \) by (rule Cons(6), rule rev-subsetD, fact k0-in, auto)

finally show \( f \, k_0 \, (\text{lookup-pair} \, xs \, k_0) \, (\text{lookup-pair} \, ys \, k_0) = \text{Some Eq} \).

next

have \( f \, k \, v \, v' = f \, k \, (\text{lookup-pair} \, ((k, v) \# \, xs) \, k) \, (\text{lookup-pair} \, ((k', v') \# \, ys) \, k) \)

by (simp add: \( k = k' \))

also have \( \ldots = \text{Some Eq} \) by (rule Cons(6), simp)

finally show \( f \, k \, v \, v' = \text{Some Eq} \).

qed

next

assume \( \text{comp} \, k \, k' = \text{Gt} \)

hence \( \text{comp} \, k' \, k = \text{Lt} \) by (simp only: Gt-lt-conv Lt-lt-conv)

show \((\text{let} \, \text{aux} = f \, k' \, 0 \, v' \, \text{in} \, \text{if} \, \text{aux} = \text{Some Eq} \, \text{then} \, \text{lex-ord-pair} \, f \, ((k, v) \# \, xs) \, ys \, \text{else} \, \text{aux}) = \text{Some Eq}\)

proof (simp add: Let-def, intro conjI impI, rule Cons(5))

fix \( k_0 \)

assume \( k_0\text{-in}: k_0 \in \text{fst} \, \text{'} \, \text{set} \, ((k, v) \# \, xs) \cup \text{fst} \, \text{'} \, \text{set} \, ys \)

hence \( k_0 \in \text{fst} \, \text{'} \, \text{set} \, xs \, \land \, k_0 = k \, \lor \, k_0 \in \text{fst} \, \text{'} \, \text{set} \, ys \) by auto

hence \( k_0 \neq k' \)

proof (elim disjE)

assume \( k_0 \in \text{fst} \, \text{'} \, \text{set} \, xs \)

hence \( \text{lt} \, k \, k_0 \) by (rule * (4))

with \( \langle \text{comp} \, k' \, k = \text{Lt} \rangle \)

show \( ?\text{thesis} \) by (simp add: Lt-lt-conv)

next

assume \( k_0 = k \)

with \( \langle \text{comp} \, k' \, k = \text{Lt} \rangle \)

show \( ?\text{thesis} \) by auto

next

assume \( k_0 \in \text{fst} \, \text{'} \, \text{set} \, ys \)

hence \( \text{lt} \, k' \, k_0 \) by (rule Cons(4))

thus \( ?\text{thesis} \) by simp

qed

hence \( f \, k_0 \, (\text{lookup-pair} \, ((k, v) \# \, xs) \, k_0) \, (\text{lookup-pair} \, ys \, k_0) = \)

\( f \, k_0 \, (\text{lookup-pair} \, ((k, v) \# \, xs) \, k_0) \, (\text{lookup-pair} \, ((k', v') \# \, ys) \, k_0) \)

by (auto simp add: lookup-pair-Con[OF Cons(1)] simp del: lookup-pair.simps)

also have \( \ldots = \text{Some Eq} \) by (rule Cons(6), rule rev-subsetD, fact k0-in, auto)

finally show \( f \, k_0 \, (\text{lookup-pair} \, ((k, v) \# \, xs) \, k_0) \, (\text{lookup-pair} \, ys \, k_0) = \text{Some Eq} \).

next

have \( f \, k' \, 0 \, v' = f \, k' \, (\text{lookup-pair} \, ((k, v) \# \, xs) \, k') \, (\text{lookup-pair} \, ((k', v') \# \, ys) \, k') \)

by (simp add: \( \langle \text{comp} \, k' \, k = \text{Lt} \rangle \))

also have \( \ldots = \text{Some Eq} \) by (rule Cons(6), simp)

finally show \( f \, k' \, 0 \, v' = \text{Some Eq} \).

qed

qed

qed

302
**lemma** lex-ord-pair-valI:

assumes oalist-inv-raw xs and oalist-inv-raw ys and aux ≠ Some Eq

assumes \( k \in \mathit{fst} \setminus \mathit{set} \bigcup \mathit{fst} \setminus \mathit{set} \mathit{ys} \) and \( \mathit{aux} = \mathit{f} \, k \, (\mathit{lookup-pair} \, \mathit{xs} \, k) \, (\mathit{lookup-pair} \, \mathit{ys} \, k) \)

and \( \land k', k' \in \mathit{fst} \setminus \mathit{set} \bigcup \mathit{fst} \setminus \mathit{set} \mathit{ys} \implies \mathit{lt} \, k' \, k \implies \mathit{f} \, k' \, (\mathit{lookup-pair} \, \mathit{xs} \, k') \, (\mathit{lookup-pair} \, \mathit{ys} \, k') = \text{Some Eq} \)

shows lex-ord-pair \( \mathit{f} \, \mathit{xs} \, \mathit{ys} = \mathit{aux} \)

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** \( \text{Cons} \, k' \, v' \, \mathit{ys} \)

from Cons(6) have \( k = k' \lor k \in \mathit{fst} \setminus \mathit{set} \mathit{ys} \) by simp

thus ?case

**proof**

assume \( k = k' \)

with Cons(7) have \( \mathit{f} \, k' \, 0 \, v' = \mathit{aux} \) by simp

thus ?thesis by (simp add: Let-def \( k = k' \) assms(3))

next

assume \( k \in \mathit{fst} \setminus \mathit{set} \mathit{ys} \)

hence \( \mathit{lt} \, k' \, k \) by (rule Cons(4))

hence \( \mathit{comp} \, k \, k' = \mathit{Gt} \) by (simp add: Gt-lt-conv)

hence eq1: \( \mathit{lookup-pair} \, ((k', v') \# \mathit{ys}) \, k = \mathit{lookup-pair} \, \mathit{ys} \, k \) by simp

have \( \mathit{f} \, k' \, (\mathit{lookup-pair} \, [] \, k') \, (\mathit{lookup-pair} \, ((k', v') \# \mathit{ys}) \, k') = \text{Some Eq} \)

by (rule Cons(8), simp, fact)

hence eq2: \( \mathit{f} \, k' \, 0 \, v' = \text{Some Eq} \) by simp

show ?thesis

**proof** (simp add: Let-def eq2, rule Cons(5))

from \( \langle k \in \mathit{fst} \setminus \mathit{set} \mathit{ys} \rangle \) show \( k \in \mathit{fst} \setminus \mathit{set} \, [] \, \mathit{ys} \) by simp

next

show \( \mathit{aux} = \mathit{f} \, k \, (\mathit{lookup-pair} \, [] \, k) \, (\mathit{lookup-pair} \, \mathit{ys} \, k) \) by (simp only: Cons(7) eq1)

next

fix \( k0 \)

assume \( \mathit{lt} \, k0 \) \( k \)

assumes \( k0 \in \mathit{fst} \setminus \mathit{set} \, [] \, \mathit{ys} \)

hence \( \mathit{k0-in} \, k0 \in \mathit{fst} \setminus \mathit{set} \mathit{ys} \) by simp

hence \( \mathit{lt} \, k' \, k0 \) by (rule Cons(4))

hence \( \mathit{comp} \, k0 \, k' = \mathit{Gt} \) by (simp add: Gt-lt-conv)

hence \( \mathit{f} \, k0 \, (\mathit{lookup-pair} \, [] \, k0) \, (\mathit{lookup-pair} \, \mathit{ys} \, k0) = \mathit{f} \, k0 \, (\mathit{lookup-pair} \, [] \, k0) \, (\mathit{lookup-pair} \, ((k', v') \# \mathit{ys}) \, k0) \) by simp

also have ... = \text{Some Eq} by (rule Cons(8), simp add: k0-in, fact)

finally show \( \mathit{f} \, k0 \, (\mathit{lookup-pair} \, [] \, k0) \, (\mathit{lookup-pair} \, \mathit{ys} \, k0) = \text{Some Eq} \).

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))
      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
    qed
next

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
  qed

next
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

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' by simp
    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'\[symmetric\])
    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 ∈ fst ' set xs ∪ fst ' set ((k'', v'') ≠ 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 ((k'', v'') ≠ 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))
  qed
thus thesis by simp

next
  assume k' ∈ fst ' set ys
  hence lt k'' k' by (rule Cons(4))
  with Lt show thesis by (simp add: Lt-lt-conv)
qed

next
  assume f k' v' 0 ≠ 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))
    qed
  moreover have ¬ 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 assm(3) show False by (simp add: k = k'')
        qed
    qed

306
from Cons(2) show lex-ord-pair f xs ys = aux
proof (rule * (5))
from Cons(6) \( k \neq k'' \) show \( k \in \text{fst} \cdot \text{set} \cdot \text{fst} \cdot \text{set} \cdot \text{ys} \) by (simp add: \( \langle k' = k'' \rangle \))
next
  show aux = \( f \cdot k \) \( \langle \text{lookup-pair} \cdot \text{xs} \cdot k \rangle \) \( \langle \text{lookup-pair} \cdot \text{ys} \cdot k \rangle \)
  by (simp add: Cons(7) lookup-pair-Cons[OF \( \star \langle \text{1} \rangle \)] lookup-pair-Cons[OF Cons(1)])
  del: lookup-pair.simps,
  simp add: \( \langle k' = k'' \rangle \) \( \langle k \neq k'' \rangle \) [symmetric]
next
  fix \( k0 \)
  assume lt k0 k
  assume k0-in: \( k0 \in \text{fst} \cdot \text{set} \cdot \text{xs} \cup \text{fst} \cdot \text{set} \cdot \text{ys} \)
  also have \( \ldots \subseteq \text{fst} \cdot \text{set} \cdot ((k', v') \# \text{xs}) \cup \text{fst} \cdot \text{set} \cdot ((k'', v'') \# \text{ys}) \) by
fastforce
finally have k0-in': \( k0 \in \text{fst} \cdot \text{set} \cdot ((k', v') \# \text{xs}) \cup \text{fst} \cdot \text{set} \cdot ((k'', v'') \# \text{ys}) \).
  have k'' \neq k0
proof
  assume k'' = k0
  with k0-in have \( k'' \in \text{fst} \cdot \text{set} \cdot \text{xs} \cup \text{fst} \cdot \text{set} \cdot \text{ys} \) by simp
  thus False
proof
    assume k'' \in \text{fst} \cdot \text{set} \cdot \text{xs}
    hence lt k' k'' by (rule Cons(4))
    thus \text{thesis} by (simp add: \( \langle k' = k'' \rangle \))
next
  assume k'' \in \text{fst} \cdot \text{set} \cdot \text{ys}
  hence lt k'' k'' by (rule Cons(4))
  thus \text{thesis} by simp
qed

qed

have \( k'' \neq k0 \)
proof
  assume k'' = k0
  with k0-in have \( k'' \in \text{fst} \cdot \text{set} \cdot \text{xs} \cup \text{fst} \cdot \text{set} \cdot \text{ys} \) by simp
  thus False
proof
    assume k'' \in \text{fst} \cdot \text{set} \cdot \text{xs}
    hence lt k' k'' by (rule Cons(4))
    thus \text{thesis} by (simp add: \( \langle k' = k'' \rangle \))
next
  assume k'' \in \text{fst} \cdot \text{set} \cdot \text{ys}
  hence lt k'' k'' by (rule Cons(4))
  thus \text{thesis} by simp
qed

also from k0-in': \( \text{lt} \cdot k0 \cdot k \) have \( \ldots = \text{Some} \cdot \text{Eq} \) by (rule Cons(8))
finally show \( f \cdot k0 \) \( \langle \text{lookup-pair} \cdot \text{xs} \cdot k0 \rangle \) \( \langle \text{lookup-pair} \cdot \text{ys} \cdot k0 \rangle \) = \text{Some} \cdot \text{Eq}.

qed

next
  assume \( f \cdot k'' \cdot v' = v'' \neq \text{Some} \cdot \text{Eq} \)
  have \( \neg \text{lt} \cdot k'' \cdot k \)
  proof
    have k'' \in \text{fst} \cdot \text{set} \cdot ((k', v') \# \text{xs}) \cup \text{fst} \cdot \text{set} \cdot ((k'', v'') \# \text{ys})
    by simp
    moreover assume \( \text{lt} \cdot k'' \cdot k \)
    ultimately have \( f \cdot k'' \cdot \langle \text{lookup-pair} \cdot ((k', v') \# \text{xs}) \cdot k'' \rangle \cdot \langle \text{lookup-pair} \cdot ((k'', v'') \# \text{ys}) \rangle \cdot \text{Some} \cdot \text{Eq} \).

  qed

also from k0-in': \( \text{lt} \cdot k0 \cdot k \) have \( \ldots = \text{Some} \cdot \text{Eq} \) by (rule Cons(8))
finally show \( f \cdot k0 \) \( \langle \text{lookup-pair} \cdot \text{xs} \cdot k0 \rangle \) \( \langle \text{lookup-pair} \cdot \text{ys} \cdot k0 \rangle \) = \text{Some} \cdot \text{Eq}.

qed
\(v'' \# ys\) \(k''\) = Some Eq  
by (rule Cons(8))
  hence \(f k'' v' v''\) = Some Eq by (simp add: \(k' = k''\))
  with \(f k'' v' v''\) \(\neq\) Some Eq show False ..
qed
moreover have \(\neg lt k k''\)
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) \(\langle k = k''\rangle\) \(\langle k' = k''\rangle\))
qed
next
assume \(Gt: \text{comp} k' k'' = Gt\)
  hence \(Lt: \text{comp} k' k'' = Lt\) by (simp only: Gt-lt-conv Lt-lt-conv)
  show (let \(aux = f k'' 0 v''\) in if \(aux = \text{Some Eq}\) then lex-ord-pair \(f\) \((k', v')\)
  \# xs) \(ys\) else \(aux\) = \(aux\)
proof (simp add: Let-def split: order.split, intro conjI impI)
  assume \(f k'' 0 v'' = \text{Some Eq}\)
  have \(k \neq k''\)
  proof
    assume \(k = k''\)
    have \(aux = f k 0 v''\) by (simp add: Cons(7) \(\langle k = k''\rangle\) Lt)
    with \(f k'' 0 v'' = \text{Some Eq}\) assms(3) show False by (simp add: \(\langle k = k''\rangle\))
  qed
  show lex-ord-pair \(f\) \((k', v')\) \# xs\) \(ys\) = \(aux\)
  proof (rule Cons(5))
    from Cons(6) \(\langle k \neq k''\rangle\) show \(k \in \text{fst ' set} \langle (k', v') \# xs\rangle\) \(\cup\) \(\text{fst ' set} \langle ys\rangle\)
  by simp
  next
    show \(aux = f k \langle \text{lookup-pair} \langle (k', v') \# xs\rangle k\rangle \langle \text{lookup-pair} \langle ys\rangle k\rangle\)
  by (simp add: Cons(7) lookup-pair-Cons[OF Cons(1)] \(\langle k \neq k''\rangle\)[symmetric]
del: lookup-pair.simps)
next
  fix \(k0\)
  assume \(Lt k0 k\)
  assume k0-in: \(k0 \in \text{fst ' set} \langle (k', v') \# xs\rangle\) \(\cup\) \(\text{fst ' set} \langle ys\rangle\)
  also have \(\ldots \subseteq \text{fst ' set} \langle (k', v') \# xs\rangle\) \(\cup\) \(\text{fst ' set} \langle (k'', v'') \# ys\rangle\)
by fastforce
  finally have k0-in': \(k0 \in \text{fst ' set} \langle (k', v') \# xs\rangle\) \(\cup\) \(\text{fst ' set} \langle (k'', v'') \# ys\rangle\).
  have \(k'' \neq k0\)
  proof
    assume \(k'' = k0\)
    with k0-in have \(k'' \in \text{fst ' set} \langle (k', v') \# xs\rangle\) \(\cup\) \(\text{fst ' set} \langle ys\rangle\) by simp
    with \(Lt\) have \(k'' \in \text{fst ' set} xs\) \(\lor\) \(k'' \in \text{fst ' set} ys\) by auto

308
thus False

proof
  assume \( k'' \in \text{fst ' set xs} \)
  hence \( \text{lt } k' k'' \) by (rule \(*4\))
  with \( \text{Lt} \) show \( ?\text{thesis} \) by (simp add: \( \text{Lt-lt-conv} \))
next
  assume \( k'' \in \text{fst ' set ys} \)
  hence \( \text{lt } k'' k'' \) by (rule \( \text{Cons}(4) \))
  thus \( ?\text{thesis} \) by simp
qed

hence \( f k0 \ (\text{lookup-pair }((k', v') \# \text{xs}) k0) \ (\text{lookup-pair } \text{ys} k0) = f k0 \ (\text{lookup-pair } ((k', v') \# \text{xs}) k0) \ (\text{lookup-pair } ((k'', v'') \# \text{ys}) k0) \)
by (simp add: \( \text{lookup-pair-Cons} \) \( \text{OF} \) \( \text{Cons}(1) \)) del: \( \text{lookup-pair.simps} \)
also from \( k0\text{-in'} : \text{lt } k0 \ k \) have \( ... = \text{Some Eq} \) by (rule \( \text{Cons}(8) \))
finally show \( f k0 \ (\text{lookup-pair }((k', v') \# \text{xs}) k0) \ (\text{lookup-pair } \text{ys} k0) = \text{Some Eq} \).
  qed
next
  assume \( f k'' 0 v'' \neq \text{Some Eq} \)
  have \( \neg \text{lt } k'' k \)
proof
  have \( k'' \in \text{fst ' set } ((k', v') \# \text{xs}) \cup \text{fst ' set } ((k'', v'') \# \text{ys}) \) by simp
  moreover assume \( \text{lt } k'' k \)
  ultimately have \( f k'' \ (\text{lookup-pair } ((k', v') \# \text{xs}) k'') \ (\text{lookup-pair } ((k'', v'') \# \text{ys}) k'') = \text{Some Eq} \)
  by (rule \( \text{Cons}(8) \))
  hence \( f k'' 0 v'' = \text{Some Eq} \) by (simp add: \( \text{Lt} \))
  with \( f k'' 0 v'' \neq \text{Some Eq} \) show False ..
  qed
moreover have \( \neg \text{lt } k k'' \)
proof
  assume \( \text{lt } k k'' \)
  with \( \text{Lt} \) have \( \text{lt } k k' \) by (simp add: \( \text{Lt-lt-conv} \))
  thus False using \( \text{lt } k k'' \) by (rule \( \text{0} \))
  qed
ultimately have \( k = k'' \) by simp
show \( f k'' 0 v'' = \text{aux} \) by (simp add: \( \text{Cons}(7) \) \( \text{i} = k k'' \text{Lt} \))
  qed
  qed
  qed
  qed

lemma \text{lex-ord-pair-EqD}:
  assumes \( \text{oalist-inv-raw xs} \) \( \text{and} \) \( \text{oalist-inv-raw ys} \) \( \text{and} \) \( \text{lex-ord-pair } f \text{xs} \text{ ys} = \text{Some Eq} \)
  and \( k \in \text{fst ' set xs} \cup \text{fst ' set ys} \)
  shows \( f k \ (\text{lookup-pair } \text{xs} k) \ (\text{lookup-pair } \text{ys} k) = \text{Some Eq} \)
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 ys 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 refl have lex-ord-pair f xs ys = f k0 (lookup-pair xs k0) (lookup-pair ys k0)
by (rule lex-ord-pair-val)
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

310
have \( k' \in ?A \) for unfolding \( k \)-def using \( \text{finite } ?A \) that by (rule Min-le)
hence \( \ast: \forall k', k' \in \text{fst } \set xs \cup \text{fst } \set ys \implies \text{lt } k' \implies \)
\( f k' (\text{lookup-pair } xs \ k') (\text{lookup-pair } ys \ k') = \text{Some Eq} \) by fastforce
with \( \text{assms(1, 2)} \) neq k-in refl have \( \text{lex-ord-pair } f \ xs \ ys = f \ k \ (\text{lookup-pair } \ xs \ k) (\text{lookup-pair } \ ys \ k) \)
by (rule \( \text{lex-ord-pair-valI} \))
hence \( \text{aux} = f \ k (\text{lookup-pair } \ xs \ k) (\text{lookup-pair } \ ys \ k) \) by (simp only: \( \text{assms(3)} \))
with k-in show \( \text{thesis} \) using \( \ast \) ..
qed

12.4.7 prod-ord-pair

lemma prod-ord-pair-eq-lex-ord-pair:
\( \text{prod-ord-pair } P \ \xs \ ys = (\text{lex-ord-pair } (\lambda k \ x \ y. \text{if } P \ k \ x \ y \text{ then } \text{Some Eq} \text{ else None}) \ \xs \ ys = \text{Some Eq}) \)
proof (induct \( P \ \xs \ ys \) rule: prod-ord-pair.induct)
case \( (1 \ P) \)
show \( ?\text{case by simp} \)
next
case \( (2 \ P \ ky \ vy \ ys) \)
thus \( ?\text{case by simp} \)
next
case \( (3 \ P \ kx \ vx \ xs) \)
thus \( ?\text{case by simp} \)
next
case \( (4 \ P \ kx \ vx \ xs \ ky \ vy \ ys) \)
show \( ?\text{case} \)
proof (cases comp \( kx \ ky \))
case \( \text{Lt} \)
thus \( ?\text{thesis by (simp add: 4(2)[OFLt])} \)
next
case \( \text{Eq} \)
thus \( ?\text{thesis by (simp add: 4(1)[OFEq])} \)
next
case \( \text{Gt} \)
thus \( ?\text{thesis by (simp add: 4(3)[OFGt])} \)
qed
qed

lemma prod-ord-pairI:
assumes oalist-inv-raw \( \xs \) and oalist-inv-raw \( \ys \)
and \( \forall k. \ k \in \text{fst } \set xs \cup \text{fst } \set ys \implies P \ k \ (\text{lookup-pair } \ xs \ k) (\text{lookup-pair } \ ys \ k) \)
shows prod-ord-pair \( P \ \xs \ ys \)
unfolding prod-ord-pair-eq-lex-ord-pair by (rule \( \text{lex-ord-pair-EqI} \), \( \text{fact} \), \( \text{fact} \), \( \text{simp add: assms(3)} \))

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

313
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 \( \text{fst ' set xs \cap \text{fst ' set ys}} = \{\}\) 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 \( k \notin \text{fst ' set ys}\) and \( \text{fst ' set xs \cap \text{fst ' set ys}} = \{\}\) by simp-all
from this(2) have eq1: \( \text{set (foldr update-by-pair xs ys)} = \text{set xs \cup set ys}\) by (rule Cons(5))
have \( \neg \text{cmp2 lt k k}\) by auto
with Cons(4) have \( k \notin \text{fst ' set xs}\) by blast
with \( k \notin \text{fst ' set ys}\) have \( \text{set (set xs \cup set ys)} = \{\}\) by (smt Int-emptyI fstI image-iff)
hence eq2: \( \text{set (set xs \cup set ys)} - \text{range (Pair k)} = \text{set xs \cup set ys}\) 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 \cup set ys) by (simp only: eq1 eq2)
finally show ?case by simp
qed

have set (foldr update-by-pair xs []) = set xs \cup 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 \( \text{cmp2.oalist-inv-raw ys}\) and \( \text{set xs = set ys}\)
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 \( k \notin \text{fst ' set ys}\) 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 \( v = \text{cmp2.lookup-pair ys k}\)
from False have \( v \neq 0\) by (simp add: v-def)
with assms(2) v-def[symmetric] have \( k, v \in \text{set ys}\) 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
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)

end

12.6 Operations on Raw Ordered Associative Lists

fun sort-oalist-aux :: 'o ⇒ ('a, 'b::zero, '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-raw 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 (λx 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::zero) list × 'd
  where map-raw f (xs, ko) = (map-pair f xs, ko)

abbreviation (in −) map2-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

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)
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-aux)
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' ox where xs: xs = (xs’, ox) 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 ≠ 0
  shows lookup-raw xs k = v \leftrightarrow ((k, v) \in 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-valueI, 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

318
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) = \{ \text{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 \( k \in \text{set (sorted-domain-raw ko xs)} \) \iff \( \text{lookup-raw xs k \neq 0} \)
  unfolding set-sorted-domain-raw[OF assms]
  proof -
  obtain \( xs' \ ko' \) where \( xs = (xs', ko') \) by fastforce
  from assms show \( k \in \text{set (fst xs) \iff \text{lookup-raw xs k \neq 0}} \)
  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 \( xs = (xs', ko) \) 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 = \( \text{if } \forall k' \in \text{fst set (fst xs), le (snd xs) k k'} \text{ then 0 else lookup-raw xs k} \)
  proof -
  obtain \( xs' \ ko \) where \( xs = (xs', ko) \) 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 = \( \text{if } k = \text{fst (List.hd (fst xs)) then 0 else lookup-raw xs k} \)
proof –
  obtain \(xs'\) \(\text{ko where } xs: xs = (xs', \text{ko})\) by fastforce
from assms show ?thesis by (simp add: xs lookup-pair-tl' oalist-inv-alt)
qued

12.6.5 \(\text{min-key-val-raw}\)

lemma \(\text{min-key-val-raw-alt}\):
  assumes oalist-inv \(xs\) and \(\text{fst } xs \neq []\)
  shows \(\text{min-key-val-raw ko xs = List.hd (sort-oalist-aux ko xs)}\)
proof –
  obtain \(xs'\) \(\text{ox where } xs: xs = (xs', \text{ox})\) by fastforce
from assms(2) have \(xs' \neq []\) 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 \(xs' \neq []\) have \(... \neq \{\}\) by simp
finally have sort-oalist ko \(xs' \neq []\) by simp
then obtain \(k v xs''\) where eq: sort-oalist ko \(xs' = (k, v) \neq xs''\)
  using prod.exhaust list.exhaust by metis
hence \((k, v) \in \text{set } xs'\) by (simp add: set-sort[ symmetric ])
have *: le ko \(k k'\) if \(k' \in \text{fst ' set } xs'\) for \(k'\)
proof –
  from that have \(k' = k \lor k' \in \text{fst ' set } xs''\) by (simp add: set-sort[ symmetric ])
egq
  thus \(\text{?thesis}\)
proof
  assume \(k' = k\)
  thus \(\text{?thesis}\) by simp
next
  have oalist-inv-raw ko \((k, v) \neq xs''\) unfolding eq[symmetric] by (fact oalist-inv-raw-sort-oalist)
  moreover assume \(k' \in \text{fst ' set } xs''\)
  ultimately have lt ko \(k k'\) by (rule oalist-inv-raw-ConsD3)
  thus \(\text{?thesis}\) by simp
qed
qed
from \(xs' \neq []\) have min-list-param \((\lambda x y. \text{le ko (fst x) (fst y)})\) \(xs'\) \(\in \text{set } xs'\) by (rule min-list-param-in)
  with oalist-inv-raw ox \(xs'\) have lookup-pair ox \(xs'\) (fst (min-list-param \((\lambda x y. \text{le ko (fst x) (fst y)})\) \(xs'\))) =
  snd (min-list-param \((\lambda x y. \text{le ko (fst x) (fst y)})\) \(xs'\)) by (auto intro: lookup-pair-eq-valueI)
moreover have 1: \(\text{fst (min-list-param (\lambda x y. \text{le ko (fst x) (fst y)}) \(xs'\)) = k}\)
proof (rule antisym)
from order.trans
  have transp \((\lambda x y. \text{le ko (fst x) (fst y)})\) by (rule transpI)
hence le ko \((\text{fst (min-list-param (\lambda x y. \text{le ko (fst x) (fst y)}) \(xs'\))) (\text{fst (k, v)})\)
  using linear \((k, v) \in \text{set } xs'\) by (rule min-list-param-minimal)
thus \( \text{le } k \) \((\text{fst } (\text{min-list-param } (\lambda x. y. \text{le } k \text{ (fst } x \text{) (fst } y \text{)}) \text{ xs}'))\) \(k\) by \text{simpl}

next
show le ko k \((\text{fst } (\text{min-list-param } (\lambda x. y. \text{le } k \text{ (fst } x \text{) (fst } y \text{)}) \text{ xs}'))\) by \text{rule \(*\), rule imageI, fact}\\
\\
qed\\
ultimately have \(\text{snd } (\text{min-list-param } (\lambda x. y. \text{le } k \text{ (fst } x \text{) (fst } y \text{)}) \text{ xs}')) = \text{lookup-pair \ ox \ xs}' \ k\)
by \text{simpl}
also from oalist-inv-raw ox xs' \((k, v) \in \text{set } xs'\), have \(v = v\) by \text{rule lookup-pair-eq-valueI}\\
finally have \(\text{snd } (\text{min-list-param } (\lambda x. y. \text{le } k \text{ (fst } x \text{) (fst } y \text{)}) \text{ xs}')) = v .\\
with \I have \(\text{min-list-param } (\lambda x. y. \text{le } k \text{ (fst } x \text{) (fst } y \text{)}) \text{ xs}' = (k, v)\) by \text{auto}\\
thus ?thesis by \((\text{simp add: } xs \ eq)\)
\\
\\
lemma min-key-val-raw-in:\\
assumes \(\text{fst } xs \neq []\)
shows \(\text{min-key-val-raw } ko \text { xs} \in \text{set } (\text{fst } xs)\)
proof --
obtain \(\text{xs'} \ \text{ox} \ \text{where xs: xs} = (xs', \ \text{ox})\) by \text{fastforce}
from \text{assms have} \(\text{xs'} \neq []\) \text{by} \((\text{simp add: } xs)\)
show ?thesis unfolding \text{xs}
proof \((\text{simp, intro conjI impl})\)
from \(\text{xs'} \neq []\) show \(\text{hd } \text{xs}' \in \text{set } \text{xs'}\) by \text{simpl}
next
from \(\text{xs'} \neq []\) show \(\text{min-list-param } (\lambda x. y. \text{le } k \text{ (fst } x \text{) (fst } y \text{)}) \text{ xs}' \in \text{set } \text{xs'}\)
by \((\text{rule min-list-param-in})\)
\\
qed
\\
\\
lemma snd-min-key-val-raw:
assumes oalist-inv \text{xs} and \(\text{fst } \text{xs} \neq []\)
shows \(\text{snd } (\text{min-key-val-raw } ko \text { xs}) = \text{lookup-raw } xs \ (\text{fst } (\text{min-key-val-raw } ko \text { xs}))\)
proof --
obtain \(\text{xs'} \ \text{ox} \ \text{where xs: xs} = (xs', \ \text{ox})\) by \text{fastforce}
from \text{assms(1) have} oalist-inv-raw ox \text{xs}' by \((\text{simp only: } xs \ oalist-inv-alt)\)
from \text{assms(2) have} min-key-val-raw \text{ko} \text { xs} \in \text{set } (\text{fst } xs) \text{by} \((\text{rule min-key-val-raw-in})\)
hence \(*: \text{min-key-val-raw } ko \ (xs', \ ox) \in \text{set } xs'\) \text{by} \((\text{simp add: } xs)\)
show ?thesis unfolding \text{xs lookup-raw.simps}
by \((\text{rule HOL.sym, rule lookup-pair-eq-valueI, fact, simp add: * del: min-key-val-raw.simps})\)
\\
qed
\\
lemma min-key-val-raw-minimal:
assumes oalist-inv \text{xs} and \(z \in \text{set } (\text{fst } xs)\)
shows le ko \((\text{fst } (\text{min-key-val-raw } ko \text { xs})) \ (\text{fst } z)\)
proof --
obtain \(\text{xs'} \ \text{ox} \ \text{where xs: xs} = (xs', \ \text{ox})\) by \text{fastforce}
from \text{assms have} oalist-inv \text{ (xs', } ox) \text{ and} \(z \in \text{set } xs'\) \text{by} \((\text{simp-all add: } xs)\)
show ?thesis unfolding \text{xs}
proof \((\text{simp, intro conjI impl})\)
\\
321
from \( z \in \text{set } xs' \) have \( xs' \neq [] \) by auto
then obtain \( k \) \( v \) \( ys \) where \( xs' = (k, v) \neq ys \) using prod.exhaust list.exhaust
by metis
from \( z \in \text{set } xs' \) have \( z = (k, v) \) \( \lor \) \( z \in \text{set } ys \) by (simp add: \( xs' \))
thus \( \text{le } ox (\text{fst } (\text{hd } xs')) ) (\text{fst } z) \)
proof
assume \( z = (k, v) \)
show \( ?\text{thesis} \) by (simp add: \( xs' \) \( \langle z = (k, v) \rangle \))
next
assume \( z \in \text{set } ys \)
\text{hence} \( \text{fst } z \in \text{fst } ' \text{set } ys \) by fastforce
with \( \langle \text{alist-inv } (xs' \), ox) \rangle \) have \( \text{lt } ox k (\text{fst } z) \)
\text{unfolding } xs' \text{alist-inv-alt lt-of-key-order.rep-eq} by (rule oalist-inv-raw-ConsD3)
thus \( ?\text{thesis} \) by (simp add: \( xs' \))
qed
next
show \( \text{le } ko (\text{fst } (\text{min-list-param } (\lambda x y. \text{le } ko (\text{fst } x) (\text{fst } y))) xs')) ) (\text{fst } z) \)
proof (rule min-list-param-minimal[of \( \lambda x y. \text{le } ko (\text{fst } x) (\text{fst } y)\)])
show \( \text{transp } (\lambda x y. \text{le } ko (\text{fst } x) (\text{fst } y)) \) by (metis (no-types, lifting) order.trans)
transpI
qed (auto intro: \( \langle z \in \text{set } xs' \rangle \))
qed

12.6.6 filter-raw

\text{lemma \ oalist-inv-filter-raw:}
\text{assumes \ oalist-inv } xs
\text{shows \ oalist-inv } (\text{filter-raw } P \  xs)
\text{proof –}
obtain \( xs' \) \( ko \) where \( xs: \  xs = (xs', \ ko) \) by fastforce
from \( \text{assms} \) show \( ?\text{thesis} \) \text{unfolding } xs \text{filter-raw.simps} \text{ oalist-inv-alt}
\text{by (rule oalist-inv-raw-filter)}
\text{qed
lemma \ lookup-raw-filter-raw:}
\text{assumes \ oalist-inv } xs
\text{shows \ lookup-raw } (\text{filter-raw } P \  xs) \  k = (\text{let } v = \text{lookup-raw } xs \ k \ \text{in if } P \ (k, v) \ \text{then } v \ \text{else } 0)\)
\text{proof –}
obtain \( xs' \) \( ko \) where \( xs: \  xs = (xs', \ ko) \) by fastforce
from \( \text{assms} \) show \( ?\text{thesis} \) \text{unfolding } xs \text{lookup-raw.simps} \text{filter-raw.simps} \text{ oalist-inv-alt}
\text{by (rule lookup-pair-filter)}
\text{qed
12.6.7 update-by-raw

\text{lemma \ oalist-inv-update-by-raw:}
\text{assumes \ oalist-inv } xs
\text{shows \ oalist-inv } (\text{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-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
    oalist-inv-alt
    by (rule update-by-fun-pair-eq-update-by-pair, fact, fact refl)
qed

corollary oalist-inv-update-by-fun-raw:
  assumes oalist-inv \(xs\)
  shows oalist-inv \((update-by-fun-raw k f)\) \(xs\) using assms by (rule oalist-inv-update-by-raw)

corollary lookup-raw-update-by-fun-raw:
  assumes oalist-inv \(xs\)
  shows lookup-raw \((update-by-fun-raw k1 f)\) \(xs\) \(k2 = (if k1 = k2 then f else id)\) \((lookup-raw xs k2)\)

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-gr-raw.simps update-by-fun-raw.simps
    update-by-fun-gr-raw.simps
    by (rule, rule conjI, rule update-by-fun-gr-pair-eq-update-by-pair, fact, fact refl)
}

323
corollary oalist-inv-update-by-fun-gr-raw:
  assumes oalist-inv xs
  shows oalist-inv (update-by-fun-gr-raw k f xs)
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)

12.6.9 map-raw and map-val-raw

lemma map-raw-cong:
  assumes ⋀ kv. kv ∈ set (fst xs) ⇒ f kv = g kv
  shows map-raw f xs = map-raw g xs
proof –
  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
  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 ⋀ 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 ⋀ 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-rd-map-rd:
  assumes oalist-inv xs and snd (f (k, 0)) = 0
  and \( \forall a \ b. \text{key-compare} (\text{rep-key-order} (\text{snd} \ xs)) (\text{fst} \ (f \ a)) \ (\text{fst} \ (f \ b)) = \text{key-compare} (\text{rep-key-order} (\text{snd} \ xs)) (\text{fst} \ a) \ (\text{fst} \ b) \)
  shows lookup-rd (map-rd f xs) (\text{fst} \ (f \ (k, v))) = \text{snd} \ (f \ (k, \text{lookup-rd} \ 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 \( \forall a \ b. \text{key-compare} (\text{rep-key-order} \ ko) (\text{fst} \ (f \ a)) \ (\text{fst} \ (f \ b)) = \text{key-compare} \ (\text{rep-key-order} \ ko) \ (\text{fst} \ a) \ (\text{fst} \ b) \)
    by (simp add: xs)
  ultimately show \( \text{thesis unfolding} \ \text{xs lookup-rd} \simps \text{map-rd} \simps \text{map-rd} \simps \text{oaalist-inv-alt} \)
    by (rule lookup-pair-map-pair)
qed

lemma map-rd-id:
  assumes oalist-inv xs
  shows map-rd 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 \( \text{?case by simp} \)
next
  case (Cons k v xs')
  show \( \text{?case by (simp add: Let-def Cons, 5) id-def[symmetric]} \)
qed
  thus \( \text{thesis by (simp add: xs)} \)
qed

lemma map-val-rd-cong:
  assumes \( \forall k \ v. \ (k, v) \in \text{set} \ (\text{fst} \ xs) \implies f \ k \ v = g \ k \ v \)
  shows map-val-rd f xs = map-val-rd g xs
proof (rule map-rd-cong)
  fix kv
  assume kv \in \text{set} \ (\text{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-rd:
  assumes oalist-inv xs
  shows oalist-inv (map-val-rd 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 \(\text{lookup-raw} (\text{map-val-raw} f xs) k = f k (\text{lookup-raw} xs k)\)

proof

obtain \(xs'\) ko where \(xs\): \(xs = (xs', ko)\) by fastforce

from assms show \(?thesis unfolding xs map-raw.simps \text{lookup-raw.simps oalist-inv-alt}\)
by (rule lookup-pair-map-val-pair)

qed

12.6.10 map2-val-raw

definition \(\text{map2-val-compat'} :: (('a, 'b::zero, 'o) oalist-raw \Rightarrow ('a, 'c::zero, 'o) oalist-raw) \Rightarrow \text{bool}\)

where \(\text{map2-val-compat'} f \mapsto \)

\((\forall zs. \text{alist-inv} zs \Rightarrow \text{alist-inv} (f zs) \land \text{snd} (f zs) = \text{snd} zs \land \text{fst ' set} (\text{fst} (f zs)) \subseteq \text{fst ' set} (\text{fst} zs)))\)

lemma map2-val-compat'I:
assumes \(\forall zs. \text{alist-inv} zs \Rightarrow \text{alist-inv} (f zs)\)

and \(\forall zs. \text{alist-inv} zs \Rightarrow \text{snd} (f zs) = \text{snd} zs\)

and \(\forall zs. \text{alist-inv} zs \Rightarrow \text{fst ' set} (\text{fst} (f zs)) \subseteq \text{fst ' set} (\text{fst} zs)\)

shows \(\text{map2-val-compat'} f\)

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 \(\text{snd} (f zs) = \text{snd} zs\)

using assms unfolding map2-val-compat'-def by blast

lemma map2-val-compat'D3:
assumes map2-val-compat' \(f\) and oalist-inv zs
shows \(\text{fst ' set} (\text{fst} (f zs)) \subseteq \text{fst ' set} (\text{fst} zs)\)

using assms unfolding map2-val-compat'-def by blast

lemma map2-val-compat'-map-val-raw: \(\text{map2-val-compat'} (\text{map-val-raw} f)\)

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 \( \text{snd} (\text{map-val-raw} f \, zs) = \text{snd} \, zs \) by simp

next

fix \(\text{zs} \:: (a', b', o')\) oalist-raw

obtain \(\text{zs}'\, \text{ko}\) where \(\text{zs} = (\text{zs}', \text{ko})\) by fastforce

show \(\text{fst} \cdot \text{set} (\text{fst} (\text{map-val-raw} f \, \text{zs})) \subseteq \text{fst} \cdot \text{set} (\text{fst} \, \text{zs})\)

proof (simp add: zs)

from map-pair-subset have \(\text{fst} \cdot \text{set} (\text{map-val-pair} f \, \text{zs}'') \subseteq \text{fst} \cdot (\lambda (k, v). (k, f k v)) \cdot \text{set} \, \text{zs}'\)

by (rule image-mono)

also have \(\ldots = \text{fst} \cdot \text{set} \, \text{zs}'\) by force

finally show \(\text{fst} \cdot \text{set} (\text{map-val-pair} f \, \text{zs}'') \subseteq \text{fst} \cdot \text{set} \, \text{zs}'\).

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 (\(\lambda \text{zs}. \text{fst} (g \, (\text{zs}, \text{ko})))\)

proof (rule map2-val-compatI)

fix \(\text{zs} \:: (a \times b)\) list

assume a: oalist-inv-raw ko \(\text{zs}\)

hence oalist-inv (\(\text{zs}, \text{ko}\)) by (simp only: oalist-inv-alt)

with assms have oalist-inv (\(g \, (\text{zs}, \text{ko})))\) by (rule map2-val-compat'D1)

hence oalist-inv (\(\text{fst} (g \, (\text{zs}, \text{ko}))), \text{snd} (g \, (\text{zs}, \text{ko})))\) by simp

thus oalist-inv-raw ko (\(\text{fst} (g \, (\text{zs}, \text{ko})))\) using assms a by (simp add: oalist-inv-alt map2-val-compat'D2)

next

fix \(\text{zs} \:: (a \times b)\) list

assume a: oalist-inv-raw ko \(\text{zs}\)

hence oalist-inv (\(\text{zs}, \text{ko}\)) by (simp only: oalist-inv-alt)

with assms have \(\text{fst} \cdot \text{set} (\text{fst} (g \, (\text{zs}, \text{ko}))) \subseteq \text{fst} \cdot \text{set} (\text{fst} \, \text{zs}))\) by (rule map2-val-compat'D3)

thus \(\text{fst} \cdot \text{set} (\text{fst} (g \, (\text{zs}, \text{ko}))) \subseteq \text{fst} \cdot \text{set} \, \text{zs}\) by simp

qed

lemma oalist-inv-map2-val-raw:

assumes oalist-inv \(\text{xs}\) and oalist-inv \(\text{ys}\)

assumes map2-val-compat' \(g\) and map2-val-compat' \(h\)

shows oalist-inv (\(\text{map2-val-raw} f \, g \, h \, \text{xs} \, \text{ys}\))

proof -

obtain \(\text{xs}'\, \text{ox}\) where \(\text{xs} = (\text{xs}', \text{ox})\) by fastforce

let \(?\text{ys} = \text{sort-oalist-aux} \, \text{ox} \, \text{ys}\)

from assms(1) have oalist-inv-raw \(\text{ox} \, \text{xs}'\) by (simp add: \(\text{xs}\) oalist-inv-alt)

moreover from assms(2) have oalist-inv-raw \(\text{ox}\) (sort-oalist-aux \(\text{ox} \, \text{ys}\)) by (rule oalist-inv-raw-sort-oalist-aux)

moreover from assms(3) have map2-val-compat ko (\(\lambda \text{zs}. \text{fst} (g \, (\text{zs}, \text{ko}))\)) for \(\text{ko}\)

327
by (rule map2-val-compat’-imp-map2-val-compat)
moreover from assms(4) have map2-val-compat ko (\lambda zs. 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 (\lambda zs. fst (g (zs, ox)))) (\lambda zs. fst (h (zs, ox))) xs' ?ys
  by (rule oalist-inv-raw-map2-val-pair)
thus ?thesis by (simp add: zs 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 \wedge zs. oalist-inv zs \implies g zs = map-val-raw (\lambda k v f k v 0) zs
and \wedge zs. oalist-inv zs \implies h zs = map-val-raw (\lambda k f k 0) zs
and \wedge k. f k 0 0 = 0
shows lookup-raw (map2-val-raw f g h xs ys) k0 = f k0 (lookup-raw zs ks)
(lookup-raw ys k0)

proof
obtain zs' ox where xs: zs = (xs', ox) by fastforce
let ?ys = sort-oalist-aux ox ys
from assms(1) have oalist-inv-raw ox zs' by (simp add: zs 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 (\lambda zs. fst (g (zs, ko))) for ko
  by (rule map2-val-compat’-imp-map2-val-compat)
moreover from assms(4) have map2-val-compat ko (\lambda zs. 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 (\lambda zs. fst (g (zs, ox)))) (\lambdazs. 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 (\lambda k v f k v 0) (zs, ox) by (rule assms(5))
thus fst (g (zs, ox)) = map-val-pair (\lambda k 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 (\lambda k f k 0) (zs, ox) by (rule assms(6))
thus fst (h (zs, ox)) = map-val-pair (\lambda k 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)

328
finally have \( \ast \colon \text{lookup-pair } ox \ \text{(map2-val-pair } ox \ (\lambda zs. \ \text{fst} \ (g \ (zs, ox))) \ (\lambda zs. \ \text{fst} \ (h \ (zs, ox))) \xs') \) \( \ys \) \( k0 = f k0 \ (\text{lookup-pair } ox \ xs' \ k0) \ (\text{lookup-raw } ys \ k0) \).

thus \( \ast \text{thesis by (simp add: xs) } \)

\textbf{qed}

\textbf{lemma} \( \text{map2-val-raw-singleton-eq-update-by-fun-raw: } \)

\textbf{assumes} \( \text{oalist-inv } xs \)

\textbf{assumes} \( \bigwedge k x. \ f k x 0 = x \ \text{and} \ \bigwedge zs. \ \text{oalist-inv } zs \ \Longrightarrow \ g zs = zs \)

\text{and} \( \bigwedge ko. h \ (((k, v)], ko) = \text{map-raw } (\lambda k. f k 0) \ (([k, v]], ko) \)

\textbf{shows} \( \text{map2-val-raw } f h xs \ (((k, v]], ko) = \text{update-by-fun-raw } k \ (\lambda x. f k x v) \ xs \)

\textbf{proof} –

\textbf{obtain} \( xs' \ ox \ \text{where} \ xs = (xs', ox) \ \text{by fastforce} \)

\textbf{let} \( \ys = \text{sort-oalist } ox \ ((k, v]] \)

\textbf{from} \( \text{assms(1)} \ \text{have} \ inv\colon \text{oalist-inv } (xs', ox) \ \text{by (simp only: xs) } \)

\textbf{hence} \( \text{inv-raw: oalist-raw } ox \ xs' \ \text{by (simp only: oalist-inv-alt) } \)

\textbf{show} \( \ast \text{thesis } \)

\textbf{proof} \( \text{(simp add: xs, intro condl impI) } \)

\textbf{assume} \( ox = ko \)

\textbf{from} \( \text{inv-raw have oalist-raw-ko } xs' \ \text{by (simp only: } (ox = ko) \)

\textbf{thus} \( \text{map2-val-pair } ko \ f \ (\lambda zs. \ \text{fst} \ (g \ (zs, ko))) \ (\lambda zs. \ \text{fst} \ (h \ (zs, ko))) \ xs' \ [[k, v]] = \)

\( \text{update-by-fun-pair } ko \ k \ (\lambda x. f k x v) \ xs' \)

\textbf{using} \( \text{assms(2) } \)

\textbf{proof} \( \text{(rule map2-val-pair-singleton-eq-update-by-fun-pair) } \)

\textbf{fix} \( zs\colon (a \times b) \ \text{list} \)

\textbf{assume} \( \text{oalist-raw } ko \ zs \)

\textbf{hence} \( \text{oalist-inv } (zs, ko) \ \text{by (simp only: oalist-inv-alt) } \)

\textbf{hence} \( g \ (zs, ko) = (zs, ko) \ \text{by (rule } \text{assms(3)} \)

\textbf{thus} \( \text{fst} \ (g \ (zs, ko)) = zs \ \text{by simp } \)

\textbf{next} \)

\textbf{show} \( \text{fst} \ (h \ (((k, v]], ko)) = \text{map-raw } (\lambda k. f k 0) \ [[k, v]] \ \text{by (simp add: } \text{assms(4)} \)

\textbf{qed}

\textbf{next} \)

\textbf{show} \( \text{map2-val-pair } ox \ f \ (\lambda zs. \ \text{fst} \ (g \ (zs, ox))) \ (\lambda zs. \ \text{fst} \ (h \ (zs, ox))) \ xs' \)

\( \bigwedge ox \ ([(k, v]]) = \\

\text{update-by-fun-pair } ox \ k \ (\lambda x. f k x v) \ xs' \)

\textbf{proof} \( \text{(cases } v = 0) \)

case \( \text{True} \)

\textbf{have} \( eq1\colon \text{sort-oalist } ox \ [[k, 0]] = [] \ \text{by (simp add: sort-oalist-def) } \)

\textbf{from} \( \text{inv have} \ eq2\colon g \ (xs', ox) = (xs', ox) \ \text{by (rule } \text{assms(3)} \)

\textbf{show} \( \ast \text{thesis } \)

\textbf{by (simp add: True eq1 eq2 } \text{assms(2)} \ \text{update-by-fun-pair-eq-update-by-pair[OF } \text{inv-raw}], \\

\text{rule HOL.sym, rule update-by-pair-id, fact inv-raw, fact refl) } \)

\textbf{next} \)

case \( \text{False} \)

\textbf{hence} \( \text{oalist-raw } ox \ [[k, v]] \ \text{by (simp add: oalist-raw-singleton) } \)

329
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 \( \text{fst} (g (zs, ox)) = zs \) by simp
next
  show \( \text{fst} (h ([(k, v)], ox)) = \text{map-val-pair} (\lambda k \cdot f k 0) [(k, v)] \) by (simp add: assms(4))
  qed
  qed
  qed

12.6.11 lex-ord-raw

lemma lex-ord-raw-EqI:
  assumes oalist-inv xs and oalist-inv ys
  and \( \forall k \cdot \text{fst} \cdot \text{set} \cdot (\text{fst} \cdot \text{set} \cdot (\text{fst} \cdot \text{set} \cdot (\text{fst} \cdot \text{set} \cdot f k \cdot (\text{lookup-raw} \cdot x s k) \cdot (\text{lookup-raw} \cdot y s k) = \text{Some} \cdot Eq) \)
  shows lex-ord-raw ko f xs ys = Some Eq
  unfolding lex-ord-raw-def

lemma lex-ord-raw-valI:
  assumes oalist-inv xs and oalist-inv ys and aux \( \neq \text{Some} \cdot Eq \)
  assumes \( k \in \text{fst} \cdot \text{set} \cdot (\text{fst} \cdot \text{set} \cdot (\text{fst} \cdot \text{set} \cdot (\text{fst} \cdot \text{set} \cdot f k \cdot (\text{lookup-raw} \cdot x s k) \cdot (\text{lookup-raw} \cdot y s k) = \text{Some} \cdot Eq) \)
  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-sort-oalist-aux[of assms(2)] assms(3)
  unfolding oalist-inv-alt
  proof (rule lex-ord-pair-valI)
    from assms(1, 2, 4) show \( k \in \text{fst} \cdot \text{set} \cdot (\text{sort-oalist-aux} \cdot ko \cdot xs) \cup \text{fst} \cdot \text{set} \cdot (\text{sort-oalist-aux} \cdot ko \cdot ys) \)
    by (simp add: set-sort-oalist-aux)
  next
    from assms(1, 2, 5) show aux = \( f k \cdot (\text{lookup-pair} \cdot ko \cdot (\text{sort-oalist-aux} \cdot ko \cdot xs) \cdot k) \)
      (lookup-pair \cdot ko \cdot (\text{sort-oalist-aux} \cdot ko \cdot ys) \cdot k)
    by (simp add: lookup-pair-sort-oalist-aux)
  next
    fix \( k' \)
    assume \( k' \in \text{fst} \cdot \text{set} \cdot (\text{sort-oalist-aux} \cdot ko \cdot xs) \cup \text{fst} \cdot \text{set} \cdot (\text{sort-oalist-aux} \cdot ko \cdot ys) \)

with assms(1, 2) have \( k' \in \text{fst} \set (\text{fst} \, \text{xs}) \cup \text{fst} \set (\text{fst} \, \text{ys}) \) by (simp add: set-sort-oalist-aux)
moreover assume \( \text{lt} \, \text{ko} \, k' \, k \)
ultimately have \( f \, k' \left( \text{lookup-raw} \, \text{xs} \, k' \right) \left( \text{lookup-raw} \, \text{ys} \, k' \right) = \text{Some Eq} \) by (rule assms(6))
with assms(1, 2) show \( f \, k' \left( \text{lookup-pair} \, \text{ko} \left( \text{sort-oalist-aux} \, \text{ko} \, \text{xs} \right) \, k' \right) \left( \text{lookup-pair} \, \text{ko} \left( \text{sort-oalist-aux} \, \text{ko} \, \text{ys} \right) \, k' \right) = \text{Some Eq} \)
by (simp add: lookup-pair-sort-oalist-aux)
qed

lemma \text{lex-ord-raw-EqD}:
assumes \( \text{oalist-inv} \, \text{xs} \) and \( \text{oalist-inv} \, \text{ys} \) and \( \text{lex-ord-raw} \, \text{ko} \, f \, \text{xs} \, \text{ys} = \text{Some Eq} \)
and \( k \in \text{fst} \set (\text{fst} \, \text{xs}) \cup \text{fst} \set (\text{fst} \, \text{ys}) \)
shows \( f \, k \left( \text{lookup-pair} \, \text{ko} \left( \text{sort-oalist-aux} \, \text{ko} \, \text{xs} \right) \, k \right) \left( \text{lookup-pair} \, \text{ko} \left( \text{sort-oalist-aux} \, \text{ko} \, \text{ys} \right) \, k \right) = \text{Some Eq} \)
proof –
have \( f \, k \left( \text{lookup-pair} \, \text{ko} \left( \text{sort-oalist-aux} \, \text{ko} \, \text{xs} \right) \, k \right) \left( \text{lookup-pair} \, \text{ko} \left( \text{sort-oalist-aux} \, \text{ko} \, \text{ys} \right) \, k \right) = \text{Some Eq} \)
with assms(1, 2) show \( ?\text{thesis} \) by (simp add: lookup-pair-sort-oalist-aux)
qed

lemma \text{lex-ord-raw-valE}:
assumes \( \text{oalist-inv} \, \text{xs} \) and \( \text{oalist-inv} \, \text{ys} \) and \( \text{lex-ord-raw} \, \text{ko} \, f \, \text{xs} \, \text{ys} = \text{aux} \)
and \( \text{aux} \neq \text{Some Eq} \)
obtains \( k \) where \( k \in \text{fst} \set (\text{fst} \, \text{xs}) \cup \text{fst} \set (\text{fst} \, \text{ys}) \)
and \( \text{aux} = f \, k \left( \text{lookup-raw} \, \text{xs} \, k \right) \left( \text{lookup-raw} \, \text{ys} \, k \right) \)
and \( \forall k', k' \in \text{fst} \set (\text{fst} \, \text{xs}) \cup \text{fst} \set (\text{fst} \, \text{ys}) \Rightarrow \text{lt} \, \text{ko} \, k' \, k \Rightarrow \)
\( f \, k' \left( \text{lookup-pair} \, \text{ko} \left( \text{sort-oalist-aux} \, \text{ko} \, \text{xs} \right) \, k' \right) \left( \text{lookup-pair} \, \text{ko} \left( \text{sort-oalist-aux} \, \text{ko} \, \text{ys} \right) \, k' \right) = \text{Some Eq} \)
proof –
let \( ?\text{x}s = \text{sort-oalist-aux} \, \text{ko} \, \text{xs} \)
let \( ?\text{y}s = \text{sort-oalist-aux} \, \text{ko} \, \text{ys} \)
from assms(3) have \( \text{lex-ord-pair} \, \text{ko} \, f \, ?\text{x}s \, ?\text{y}s = \text{aux} \) 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 \( a: k \in \text{fst} \set ?\text{x}s \cup \text{fst} \set ?\text{y}s \)
and \( b: \text{aux} = f \, k \left( \text{lookup-pair} \, \text{ko} \, ?\text{x}s \, k \right) \left( \text{lookup-pair} \, \text{ko} \, ?\text{y}s \, k \right) \)
and \( c: \forall k', k' \in \text{fst} \set ?\text{x}s \cup \text{fst} \set ?\text{y}s \Rightarrow \text{lt} \, \text{ko} \, k' \, k \Rightarrow \)
\( f \, k' \left( \text{lookup-pair} \, \text{ko} \left( \text{sort-oalist-aux} \, \text{ko} \, \text{xs} \right) \, k' \right) \left( \text{lookup-pair} \, \text{ko} \left( \text{sort-oalist-aux} \, \text{ko} \, \text{ys} \right) \, k' \right) = \text{Some Eq} \)
using assms(4) unfolding oalist-inv-alt by (rule lex-ord-pair-valE, blast)
from \( a \) have \( k \in \text{fst} \set (\text{fst} \, \text{xs}) \cup \text{fst} \set (\text{fst} \, \text{ys}) \)
by (simp add: set-sort-oalist-aux assms(1, 2))
moreover from \( b \) have \( \text{aux} = f \, k \left( \text{lookup-raw} \, \text{xs} \, k \right) \left( \text{lookup-raw} \, \text{ys} \, k \right) \)
by (simp add: lookup-pair-sort-oalist-aux assms(1, 2))
moreover have \( f \, k' \left( \text{lookup-raw} \, \text{xs} \, k' \right) \left( \text{lookup-raw} \, \text{ys} \, k' \right) = \text{Some Eq} \)
if \( k'\text{-in}: k' \in \text{fst} \set (\text{fst} \, \text{xs}) \cup \text{fst} \set (\text{fst} \, \text{ys}) \) and \( k'\text{-less}: \text{lt} \, \text{ko} \, k' \, k \) for \( k' \)
proof –
have \( f \, k' \left( \text{lookup-raw} \, \text{xs} \, k' \right) \left( \text{lookup-raw} \, \text{ys} \, k' \right) = f \, k' \left( \text{lookup-pair} \, \text{ko} \, ?\text{x}s \, k' \right) \left( \text{lookup-pair} \, \text{ko} \, ?\text{y}s \, k' \right) \)
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 \cup 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 \( \forall k. k \in \text{fst } ' \text{ set } (\text{fst } xs) \cup \text{fst } ' \text{ set } (\text{fst } ys) \implies P k (\text{lookup-raw } xs \ k) \) (\text{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' (\text{sort-oalist-aux } ox \ ys) using oalist-inv-raw-sort-oalist-aux
  proof (rule prod-ord-pairI)
    fix k
    assume k \in \text{fst } ' \text{ set } xs' \cup \text{fst } ' \text{ set } (\text{sort-oalist-aux } ox \ ys)
    hence k \in \text{fst } ' \text{ set } (\text{fst } xs) \cup \text{fst } ' \text{ set } (\text{fst } ys) by (simp add: xs set-sort-oalist-aux assms(2))
    hence P k (\text{lookup-raw } xs \ k) (\text{lookup-raw } ys \ k) by (rule assms(3))
    thus P k (\text{lookup-pair ox xs' k}) (\text{lookup-pair ox } (\text{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 \in \text{fst } ' \text{ set } (\text{fst } xs) \cup \text{fst } ' \text{ set } (\text{fst } ys)
  shows P k (\text{lookup-raw } xs \ k) (\text{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' (\text{sort-oalist-aux } ox \ ys) by (simp add: xs)
  moreover from assms(4) have k \in \text{fst } ' \text{ set } xs' \cup \text{fst } ' \text{ set } (\text{sort-oalist-aux } ox \ ys)
    by (simp add: xs set-sort-oalist-aux assms(2))
ultimately have \(*\): \( P \ k \ (\text{lookup-pair} \ ox \ x's \ k) \ (\text{lookup-pair} \ ox \ (\text{sort-oalist-aux} \ ox \ y's) \ k) \)
by (rule prod-ord-pairD)
thus \(?\)thesis by (simp add: x's lookup-pair-sort-oalist-aux assms(2))
qed

corollary prod-ord-raw-alt:
assumes oalist-inv x's and oalist-inv y's
shows prod-ord-raw \( P \ x's \ y's \longleftrightarrow \ (\forall k \in \text{fst} \ ' \ \text{set} \ (\text{fst} \ x's) \cup \text{fst} \ ' \ \text{set} \ (\text{fst} \ y's). \ P \ k \ (\text{lookup-raw} \ x's \ k) \ (\text{lookup-raw} \ y's \ k)) \)

12.6.13 oalist-eq-raw

lemma oalist-eq-rawI:
assumes oalist-inv x's and oalist-inv y's
and \( \forall k. k \in \text{fst} \ ' \ \text{set} \ (\text{fst} \ x's) \cup \text{fst} \ ' \ \text{set} \ (\text{fst} \ y's) \implies \text{lookup-raw} \ x's \ k = \text{lookup-raw} \ y's \ k \)
shows oalist-eq-raw x's y's
proof
obtain x's' ox where x's = (x's', ox) by fastforce
from assms(1) have oalist-inv-raw x's' ox' by (simp only: x's oalist-inv-alt)
hence \(*\): x's' = sort-oalist-aux ox y's using oalist-inv-raw-sort-oalist-aux[OF assms(2)]
proof (rule lookup-pair-inj)
show lookup-pair ox x's' = lookup-pair ox (sort-oalist-aux ox y's)
proof
fix k
show lookup-pair ox x's' k = lookup-pair ox (sort-oalist-aux ox y's) k
proof (cases k \in \text{fst} \ ' \ \text{set} x's' \cup \text{fst} \ ' \ \text{set} \ (\text{sort-oalist-aux} \ ox y's))
case True
hence k \in \text{fst} \ ' \ \text{set} \ (\text{fst} \ x's) \cup \text{fst} \ ' \ \text{set} \ (\text{fst} \ y's) by (simp add: x's set-sort-oalist-aux assms(2))
hence lookup-raw x's k = lookup-raw y's k by (rule assms(3))
thus \(?\)thesis by (simp add: x's lookup-pair-sort-oalist-aux assms(2))
next
case False
hence k \notin \text{fst} \ ' \ \text{set} x's' and k \notin \text{fst} \ ' \ \text{set} \ (\text{sort-oalist-aux} \ ox y's) by simp-all
with oalist-inv-raw x's' ox x's' oalist-inv-raw-sort-oalist-aux[OF assms(2)]
have lookup-pair ox x's' k = 0 and lookup-pair ox (sort-oalist-aux ox y's) k = 0
by (simp-all add: lookup-pair-eq-0)
thus \(?\)thesis by simp
qed
qed
thus \(?\)thesis by (simp add: x's)
qed

333
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 ←→ (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 oalist-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))
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 ≠ 0 ⇒ lookup xs k = v ←→ ((k, v) ∈ 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) ∈ set (fst (list-of-oalist xs)) ⇒ 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) ⇒ 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: \( k \in \text{set} \ (\text{sorted-domain \ ko \ xs}) \iff (\text{lookup \ xs \ k} \neq 0) \)

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 \ (\forall \ k' \in \text{fst} \ \ \text{set} \ \ (\text{fst} \ \text{list-of-oalist} \ \ xs)). \ le \ (\text{snd} \ \text{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 \text{fst} \ (\text{list-of-oalist} \ \ xs) \neq \ []
shows \text{hd} \ \text{xs} \in \text{set} \ \ (\text{fst} \ \text{list-of-oalist} \ \ xs))

unfolding hd-def using assms by (rule hd-in-set)
lemma snd-hd:
  assumes \( \text{fst} \ (\text{list-of-oalist} \ \text{xs}) \neq [] \)
  shows \( \text{snd} \ (\text{hd} \ \text{xs}) = \text{lookup} \ \text{xs} \ (\text{fst} \ (\text{hd} \ \text{xs})) \)
proof
  from \text{assms} \ \text{have} *: \text{hd} \ \text{xs} \in \text{set} \ (\text{fst} \ (\text{list-of-oalist} \ \text{xs})) \ \text{by} \ (\text{rule} \ \text{hd-in})
  show \ ?thesis \ \text{by} \ (\text{rule} \ \text{HOL.sym}, \ \text{rule} \ \text{lookup-eq-valueI}, \ \text{simp add: *} )
qed

lemma lookup-tl': \text{lookup} \ (\text{tl} \ \text{xs}) \ k = (\text{if} \ k = \text{fst} \ (\text{hd} \ \text{xs}) \ \text{then} \ \text{0} \ \text{else} \ \text{lookup} \ \text{xs} \ k) 
  by \ (\text{simp add: lookup-def lookup-raw-tl-raw oalist-inv-list-of-oalist hd-def})

lemma hd-tl:
  assumes \( \text{fst} \ (\text{list-of-oalist} \ \text{xs}) \neq [] \)
  shows \( \text{list-of-oalist} \ \text{xs} = ((\text{hd} \ \text{xs}) \# (\text{fst} \ (\text{list-of-oalist} \ (\text{tl} \ \text{xs})))) \), \( \text{snd} \ (\text{list-of-oalist} \ (\text{tl} \ \text{xs}))) \)
proof
  obtain \text{xs'} \ \text{ko} \ \text{where} \ \text{xs: list-of-oalist} \ \text{xs} = (\text{xs'}, \ \text{ko}) \ \text{by} \ \text{fastforce}
  from \text{assms} \ \text{obtain} \ \text{x xs'' where} \ \text{xs': xs'} = \text{x} \ # \ \text{xs''} \ \text{unfolding} \ \text{xs fst-conv}
  using \ \text{list.exhaust} \ \text{by} \ \text{blast}
  show \ ?thesis \ \text{by} \ (\text{simp add: xs xs' hd-def})
qed

12.7.7  \text{min-key-val}

lemma min-key-val-alt:
  assumes \( \text{fst} \ (\text{list-of-oalist} \ \text{xs}) \neq [] \)
  shows \( \text{min-key-val} \ \text{ko} \ \text{xs} = \text{hd} \ (\text{reorder} \ \text{ko} \ \text{xs}) \)
  using \text{assms oalist-inv-list-of-oalist} \ \text{by} \ (\text{simp add: min-key-val-def hd-def min-key-val-raw-alt})

lemma min-key-val-in:
  assumes \( \text{fst} \ (\text{list-of-oalist} \ \text{xs}) \neq [] \)
  shows \( \text{min-key-val} \ \text{ko} \ \text{xs} \in \text{set} \ (\text{fst} \ (\text{list-of-oalist} \ \text{xs})) \)
  unfolding \text{min-key-val-def} \ \text{using} \ \text{assms} \ \text{by} \ (\text{rule} \ \text{min-key-val-raw-in})

lemma snd-min-key-val:
  assumes \( \text{fst} \ (\text{list-of-oalist} \ \text{xs}) \neq [] \)
  shows \( \text{snd} \ (\text{min-key-val} \ \text{ko} \ \text{xs}) = \text{lookup} \ \text{xs} \ (\text{fst} \ (\text{min-key-val} \ \text{ko} \ \text{xs})) \)
  unfolding \text{lookup-def} \ \text{min-key-val-def} \ \text{using} \ \text{oalist-inv-list-of-oalist} \ \text{assms} \ \text{by}
  (\text{rule} \ \text{snd-min-key-val-raw})

lemma min-key-val-minimal:
  assumes \( \text{z} \in \text{set} \ (\text{fst} \ (\text{list-of-oalist} \ \text{xs})) \)
  shows \( \text{le} \ \text{ko} \ (\text{fst} \ (\text{min-key-val} \ \text{ko} \ \text{xs})) \ (\text{fst} \ \text{z}) \)
  unfolding \text{min-key-val-def}
  by \ (\text{rule} \ \text{min-key-val-raw-minimal}, \ \text{fact} \ \text{oalist-inv-list-of-oalist, fact})

12.7.8  \text{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') 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')

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 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,

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 map2-val-compat'-id, fact map2-val-compat'-id)

lemma lookup-map2-val-neutr:
assumes \(\land k x. f k x 0 = x\) and \(\land k x. f k 0 x = x\)
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-raw (\(\lambda k v. f k v 0\)) 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 (\(\lambda k. f k 0\)) zs by (simp add: assms(2) map-raw-id)
qed (fact oalist-inv-list-of-oalist, fact oalist-inv-list-of-oalist)
12.7.13  oalist-eq

lemma oalist-eq-alt: oalist-eq xs ys \iff (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-oalizt oalist-of-listzt
for rep-key-order :: 'o \Rightarrow 'a key-order
and list-of-oalistx :: 'x \Rightarrow ('a, 'b::zero, 'o) oalist-raw
and oalist-of-listx :: ('a, 'b, 'o) oalist-raw \Rightarrow 'x
and list-of-oalisty :: 'y \Rightarrow ('a, 'c::zero, 'o) oalist-raw
and oalist-of-listy :: ('a, 'c, 'o) oalist-raw \Rightarrow 'y
and list-of-oalizt :: 'z \Rightarrow ('a, 'd::zero, 'o) oalist-raw
and oalist-of-listzt :: ('a, 'd, 'o) oalist-raw \Rightarrow 'z

begin

definition map-val :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'x \Rightarrow 'y
where map-val f xs = oalist-of-listy (map-val-raw f (list-of-oalistzt xs))

definition map2-val :: ('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd) \Rightarrow 'x \Rightarrow 'y \Rightarrow 'z
where map2-val f xs ys =
  oalist-of-listzt (map2-val-raw f (map-val-raw (\lambda k. f k 0))) (map-val-raw
    (\lambda (k f k 0))) (list-of-oalistzt xs) (list-of-oalisty ys))

definition map2-val-rneutr :: ('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'b) \Rightarrow 'x \Rightarrow 'y \Rightarrow 'x
where map2-val-rneutr f xs ys =
  oalist-of-listzt (map2-val-raw f id (map-val-raw (\lambda k. f k 0))) (list-of-oalistzt
    xs) (list-of-oalisty ys))

definition lex-ord :: 'o \Rightarrow ('a \Rightarrow ('b, 'c) opt-comp) \Rightarrow ('x, 'y) opt-comp
where lex-ord ko f xs ys = lex-ord-raw ko f (list-of-oalistzt xs) (list-of-oalisty ys)

definition prod-ord :: ('a \Rightarrow 'b \Rightarrow 'c \Rightarrow bool) \Rightarrow 'x \Rightarrow 'y \Rightarrow bool
where prod-ord f xs ys = prod-ord-raw f (list-of-oalistzt xs) (list-of-oalisty ys)

12.8.1  map-val

lemma map-val-cong:
  assumes \(\forall k v. (k, v) \in \text{set}(\text{fst}(\text{list-of-oalistzt xs})) \implies f k v = g k v\)
  shows map-val f xs = map-val g xs

341
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-oalistx (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))
(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-o alist, fact oay.oalist-inv-list-of-oalist,
  fact map2-val-compat'-id, fact map2-val-compat'-map-val-raw)

lemma lookup-map2-val:
  assumes ∀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 ∀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 (λk 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 refl, simp
  only: assms)
**Lemma** map2-val-rneutr-singleton-eq-update-by-fun:

**Assumes** \( \forall a, x. f a x 0 = x \) and list-of-oalist \( y s = \{(k, v), oy\} \)

**Shows** map2-val-rneutr \( f \) \( y s = \text{update-by-fun} k \ (\lambda x. f k x v) \) \( x s \)


**12.8.3** lex-ord and prod-ord

**Lemma** lex-ord-EqI:

\[ (\forall k. k \in \text{fst} \ (\text{fst} \ (\text{list-of-oalist} \ x s)) \cup \text{fst} \ (\text{fst} \ (\text{list-of-oalist} \ y s)) \Rightarrow f k \ (\text{lookup} \ x s k) \ (\text{oay.lookup} \ y s k) = \text{Some Eq} \Rightarrow \]

lex-ord \( k \) \( x s \) \( y s = \text{Some Eq} \)

**Proof** (simp add: lex-ord-def lookup-def oay.lookup-def, rule lex-ord-EqI, rule oalist-inv-list-of-oalist, rule oay.lookup-def, blast)

**Lemma** lex-ord-valI:

**Assumes** \( aux \neq \text{Some Eq} \) and \( k \in \text{fst} \ (\text{fst} \ (\text{list-of-oalist} \ x s)) \cup \text{fst} \ (\text{fst} \ (\text{list-of-oalist} \ y s)) \)

**Shows** \( aux = f k \ (\text{lookup} \ x s k) \ (\text{oay.lookup} \ y s k) \Rightarrow \)

\[ (\forall k'. k' \in \text{fst} \ (\text{fst} \ (\text{list-of-oalist} \ x s)) \cup \text{fst} \ (\text{fst} \ (\text{list-of-oalist} \ y s)) \Rightarrow \]

lt \( k \) \( k' \) \( \Rightarrow f k' \ (\text{lookup} \ x s k') \ (\text{oay.lookup} \ y s k') = \text{Some Eq} \Rightarrow \]

lex-ord \( k \) \( x s \) \( y s = aux \)

**Proof** (simp (no-asm-use) add: lex-ord-def lookup-def oay.lookup-def, rule lex-ord-EqI, rule oalist-inv-list-of-oalist, rule oay.lookup-def, blast)

**Lemma** lex-ord-EqD:

lex-ord \( k \) \( x s \) \( y s = \text{Some Eq} \Rightarrow \)

\[ k \in \text{fst} \ (\text{fst} \ (\text{list-of-oalist} \ x s)) \cup \text{fst} \ (\text{fst} \ (\text{list-of-oalist} \ y s)) \Rightarrow \]

f k \ (\text{lookup} \ x s k) \ (\text{oay.lookup} \ y s k) = \text{Some Eq} \Rightarrow \]

lex-ord \( k \) \( x s \) \( y s = aux \)

**Proof** (simp add: lex-ord-def lookup-def oay.lookup-def, rule lex-ord-EqD[where f=f], rule oalist-inv-list-of-oalist, rule oay.oalist-inv-list-of-oalist, assumption, simp)

**Lemma** lex-ord-valE:

**Assumes** \( \text{lex-ord} \) \( f \) \( x s \) \( y s = aux \) and \( aux \neq \text{Some Eq} \)

**Obtains** \( k \) \text{where} \( k \in \text{fst} \ (\text{fst} \ (\text{list-of-oalist} \ x s)) \cup \text{fst} \ (\text{fst} \ (\text{list-of-oalist} \ y s)) \)

and \( aux = f k \ (\text{lookup} \ x s k) \ (\text{oay.lookup} \ y s k) \)

and \( \forall k'. k' \in \text{fst} \ (\text{fst} \ (\text{list-of-oalist} \ x s)) \cup \text{fst} \ (\text{fst} \ (\text{list-of-oalist} \ y s)) \Rightarrow \)

lt \( k \) \( k' \) \( \Rightarrow f k' \ (\text{lookup} \ x s k') \ (\text{oay.lookup} \ y s k') = \text{Some Eq} \)

**Proof** (simp only: lex-ord-def)

**Note** oalist-inv-list-of-oalist oay.oalist-inv-list-of-oalist

**Moreover from** \( \text{assms}(1) \) \text{have} \( \text{lex-ord-raw} \) \( k \) \( f \) \( (\text{list-of-oalist} \ x s) \) \( (\text{list-of-oalist} \ y s) = aux \)

**Proof** (simp only: lex-ord-def)

**Ultimately obtain** \( k \) \text{where} \( 1: k \in \text{fst} \ (\text{fst} \ (\text{list-of-oalist} \ x s)) \cup \text{fst} \ (\text{fst} \ (\text{list-of-oalist} \ y s)) \)
(fst (list-of-oalisty ys))
and aux = f k (lookup-raw (list-of-oalistx xs) k) (lookup-raw (list-of-oalisty ys) k)
and \( \forall k', k' \in \text{fst} \cdot \text{set} (\text{fst} (\text{list-of-oalistx} \ \text{xs})) \cup \text{fst} \cdot \text{set} (\text{fst} (\text{list-of-oalisty} \ \text{ys})) \) \( \Rightarrow \)
lt ko k' k \( \Rightarrow \)
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 \( \forall k'. k' \in \text{fst} \cdot \text{set} (\text{fst} (\text{list-of-oalistx} \ \text{xs})) \cup \text{fst} \cdot \text{set} (\text{fst} (\text{list-of-oalisty} \ \text{ys})) \) \( \Rightarrow \)
lt ko k' k \( \Rightarrow \)
f k' (lookup xs k') (oay.lookup ys k') = Some Eq
by (simp-all only: lookup-pair oay.lookup-def)
with 1 show \(?\)thesis ..
qed

lemma prod-ord-alt:
prod-ord P xs ys \( \iff \)
\( \forall k \in \text{fst} \cdot \text{set} (\text{fst} (\text{list-of-oalistx} \ \text{xs})) \cup \text{fst} \cdot \text{set} (\text{fst} (\text{list-of-oalisty} \ \text{ys})) \).
\( P \) \( k \) (lookup xs k) (oay.lookup ys k)
by (simp add: prod-ord-def lookup-pair 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
ko

344
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.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}
typedef (overloaded) ('a, 'b) oalist = {xs::('a, 'b::zero, 'a key-order) oalist-raw.}

morphism 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 → ks 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 $xs'$ or where $xs: xs = (xs', ox)$ by fastforce


qed

lemma OList-list-of-oalist [code abstype]: OList (list-of-oalist $xs$) = $xs$

proof –

obtain $xs'$ or 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.oalist-inv-list-of-oalist)

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 $\lambda x. x$ list-of-oalist OList

defines OList-lookup = oa.lookup

and OList-sorted-domain = oa.sorted-domain

and OList-empty = oa.empty

and OList-reorder = oa.reorder

and OList-tl = oa.tl

and OList-hd = oa.hd

and OList-except-min = oa.except-min

and OList-min-key-val = oa.min-key-val

and OList-insert = oa.insert

and OList-update-by-fun = oa.update-by-fun

and OList-update-by-fun-gr = oa.update-by-fun-gr

and OList-filter = oa.filter

and OList-map2-val-neutr = oa.map2-val-neutr

and OList-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 OList-list-of-oalist)

done

global-interpretation oa: oalist-abstract3 $\lambda x. x$

list-of-oalist::('a', 'b') oalist $\Rightarrow$ ('a', 'b::zero', 'a key-order) oalist-raw OList

list-of-oalist::('a', 'c') oalist $\Rightarrow$ ('a', 'c::zero', 'a key-order) oalist-raw OList

list-of-oalist::('a', 'd') oalist $\Rightarrow$ ('a', 'd::zero', 'a key-order) oalist-raw OList

defines OList-map-val = oa.map-val

and OList-map2-val = oa.map2-val

and OList-map2-val-neutr = oa.map2-val-neutr

and OList-lex-ord = oa.lex-ord

and OList-prod-ord = oa.prod-ord ..

lemmas OList-lookup-single = oa.lookup-oalist-of-list-single[folded oalist-of-list-def]
12.10 Type oalist-tc

“tc” stands for “type class”.

\textbf{global-interpretation} \textit{tc}: comparator comparator-of
\begin{itemize}
  \item \textbf{defines} lookup-pair-tc = tc.lookup-pair
  \item \textbf{and} update-by-pair-tc = tc.update-by-pair
  \item \textbf{and} update-by-fun-pair-tc = tc.update-by-fun-pair
  \item \textbf{and} update-by-fun-gr-pair-tc = tc.update-by-fun-gr-pair
  \item \textbf{and} map2-val-pair-tc = tc.map2-val-pair
  \item \textbf{and} lex-ord-pair-tc = tc.lex-ord-pair
  \item \textbf{and} prod-ord-pair-tc = tc.prod-ord-pair
  \item \textbf{and} sort-oalist-tc = tc.sort-oalist
\end{itemize}
\textbf{by} (\textit{fact comparator-of})

\textbf{lemma} tc-le-lt \[\textbf{simp}]: tc.le = (\leq) \text{\emph{tc}.lt} = (\textless)
\textbf{by} (\textit{auto simp: le-of-comp-def lt-of-comp-def comparator-of-def intro!: ext split: order.split-asm if-split-asm})

\textbf{typedef (overloaded)} \{'a, 'b\} oalist-tc = \{xs::\{'a::linorder \times 'b::zero\} list. tc.oalist-inv-raw xs\}
\begin{itemize}
  \item \textbf{morphisms} list-of-oalist-tc Abs-oalist-tc
  \textbf{by} (\textit{auto intro: tc.oalist-inv-raw-Nil})
\end{itemize}

\textbf{lemma} oalist-tc-eq-iff: \textit{xs} = \textit{ys} \iff list-of-oalist-tc \textit{xs} = list-of-oalist-tc \textit{ys}
\textbf{by} (\textit{simp add: list-of-oalist-tc-inject})

\textbf{lemma} oalist-tc-eql: list-of-oalist-tc \textit{xs} = list-of-oalist-tc \textit{ys} \implies \textit{xs} = \textit{ys}
\textbf{by} (\textit{simp add: oalist-tc-eq-iff})

Formal, totalized constructor for \{'a, 'b\} oalist-tc:

\textbf{definition} OAlist-tc :: \{'a \times 'b\} list \Rightarrow \{'a::linorder, 'b::zero\} oalist-tc \textbf{where}
OAlist-tc \textit{xs} = Abs-oalist-tc (sort-oalist-tc \textit{xs})

\textbf{definition} oalist-tc-of-list = OAlist-tc

\textbf{lemma} oalist-inv-list-of-oalist-tc: tc.oalist-inv-raw (list-of-oalist-tc \textit{xs})
\textbf{using} list-of-oalist-tc[of \textit{xs}] \textbf{by simp}

\textbf{lemma} list-of-oalist-OAlist-tc: list-of-oalist-tc (OAlist-tc \textit{xs}) = sort-oalist-tc \textit{xs}
\textbf{by} (\textit{simp add: OAlist-tc-def Abs-oalist-tc-inverse tc.oalist-inv-raw-sort-oalist})

\textbf{lemma} OAlist-list-of-oalist-tc [\textit{code abstype}]: OAlist-tc (list-of-oalist-tc \textit{xs}) = \textit{xs}
\textbf{by} (\textit{simp add: OAlist-tc-def tc.sort-oalist-id list-of-oalist-tc-inverse oalist-inv-list-of-oalist-tc})

\textbf{lemma} list-of-oalist-tc-of-list [\textit{code abstract}]: list-of-oalist-tc (oalist-tc-of-list \textit{xs}) = sort-oalist-tc \textit{xs}
\textbf{by} (\textit{simp add: list-of-oalist-OAlist-tc oalist-tc-of-list-def})

\textbf{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) oalist-tc
  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::linorder, 'd::zero) oalist-tc
  where OAlist-tc-map2-val f xs ys =
OAlist-tc (map2-val-pair-tc f (map-val-pair (λk b. f k b 0)) (map-val-pair (λk f k 0)))
(list-of-oalist-tc xs) (list-of-oalist-tc ys))

definition OAlist-tc-map2-val-neutr ::
(′a ⇒ ′b ⇒ ′c ⇒ ′b) ⇒ (′a, ′b) oalist-tc
declares (′a::linorder, ′b::zero) oalist-tc

where OAlist-tc-map2-val-neutr f xs ys =
OAlist-tc (map2-val-pair-tc f id (map-val-pair (λk f k 0)) (list-of-oalist-tc xs) (list-of-oalist-tc ys))

OAlist-tc-lex-ord ::
(′a ⇒ (′b, ′c) comp-opt) ⇒ ((′a, ′b) oalist-tc, (′a::linorder, ′c::zero) oalist-tc) comp-opt

declares (′a::linorder, ′c::zero) oalist-tc ⇒ bool

where OAlist-tc-lex-ord f xs ys = lex-ord-pair-tc f (list-of-oalist-tc xs) (list-of-oalist-tc ys)

OAlist-tc-prod-ord ::
(′a ⇒ ′b ⇒ ′c ⇒ bool) ⇒ (′a, ′b) oalist-tc
declares (′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 ⇒ 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) ⇒ 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) \iff (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

349
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-raw[D2])

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′∈lst . 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: order.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 s: OAlist-tc-min-key-val xs ∈ set ?xs by (rule OAlist-tc-min-key-val-in)
    show ?thesis unfolding OAlist-tc-lookup-def

350
by (rule HOL.sym, rule tc.lookup-pair-eq-value1, fact oalist-inv-list-of-oalist-tc, simp add: *)
qed

lemma OList-tc-min-key-val-minimal:
assumes \( z \in \text{set} \ (\text{list-of-oalist-tc} \ \text{xs}) \)
shows \( \text{fst} \ (\text{OList-tc-min-key-val} \ \text{xs}) \leq \text{fst} \ z \)
proof –
let \(?\text{xs} = \text{list-of-oalist-tc} \ \text{xs}\)
from assms have \(?\text{xs} \neq \[]\) by auto
hence \(\text{OList-tc-sorted-domain} \ \text{xs} \neq \[]\) by (simp add: OList-tc-sorted-domain-def)
then obtain \(?h \ \text{xs'}\) where eq: \(\text{OList-tc-sorted-domain} \ \text{xs} = \ h \# \ \text{xs'}\) using list.exhaust by blast
with \(\text{sorted-OList-tc-sorted-domain[of} \ \text{xs}]\) have \(\ast: \forall y \in \text{set} \ \text{xs'}. \ h < y \ \text{by simp}\)
from assms have \(\text{fst} \ z \in \text{set} \ (\text{OList-tc-sorted-domain} \ \text{xs})\) by (simp add: OList-tc-sorted-domain-def)
hence disj: \(\text{fst} \ z = h \lor \text{fst} \ z \in \text{set} \ \text{xs'}\) by (simp add: eq)
from (?\text{xs} \neq \[]) have \(\text{fst} \ (\text{OList-tc-min-key-val} \ \text{xs}) = \ \text{hd} \ (\text{OList-tc-sorted-domain} \ \text{xs})\)
by (simp add: OList-tc-min-key-val-def OList-tc-sorted-domain-def hd-map)
also have \(\ldots \ = \ h\) by (simp add: eq)
also from disj have \(\ldots \leq \text{fst} \ z\)
proof
assume \(\text{fst} \ z = h\)
thus \(\text{?thesis}\) by simp
next
assume \(\text{fst} \ z \in \text{set} \ \text{xs'}\)
with \(\ast\) have \(h < \text{fst} \ z\) ..
thus \(\text{?thesis}\) by simp
qed
finally show \(\text{?thesis}\).
qed

12.10.6 OList-tc-insert

lemma list-of-oalist-OList-tc-insert [simp, code abstract]:
list-of-oalist-tc \((\text{OList-tc-insert} \ k \ v \ \text{xs})\) = update-by-pair-tc \(k \ v\) \(\text{list-of-oalist-tc} \ \text{xs}\)
unfolding OList-tc-insert-def
by (rule list-of-oalist-tc-of-list-id, rule tc.oalist-_inv-rue-update-by-pair, fact oalist-inv-list-of-oalist-tc)

lemma lookup-OList-tc-insert: OList-tc-lookup \((\text{OList-tc-insert} \ k \ v \ \text{xs})\) \(k'\) =
(if \(k = k'\) then \(v\) else \(\text{OList-tc-lookup} \ \text{xs} \ k'\))
by (simp add: OList-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 OList-tc-update-by-fun and OList-tc-update-by-fun-gr

lemma list-of-oalist-OList-tc-update-by-fun [simp, code abstract]:
list-of-oalist-tc \((\text{OList-tc-update-by-fun} \ k \ f \ \text{xs})\) = update-by-fun-pair-tc \(k \ f\) \(\text{list-of-oalist-tc} \ \text{xs}\)
unfolding OList-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)

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 \( \forall k. v. (k, v) \in set (list-of-oalist-tc xs) \implies f k v = g k v \)
  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 \implies 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 \( OAlist\text{-}tc\text{-}map2\text{-}val \ OAlist\text{-}tc\text{-}map2\text{-}val\text{-}rneutr \ and \ OAlist\text{-}tc\text{-}map2\text{-}val\text{-}rneutr \)

**Lemma** list-of-oalist-map2-val [simp, code abstract]:

\[
\text{list-of-oalist\text{-}tc} (OAlist\text{-}tc\text{-}map2\text{-}val f xs ys) = \\
\text{map2\text{-}pair\text{-}tc} f (\text{map\text{-}pair} (\lambda k b. f k b 0)) (\text{map\text{-}pair} (\lambda k. f k 0)) \\
\text{list-of-oalist\text{-}tc} xs (\text{list-of-oalist\text{-}tc} ys)
\]

**Unfolding** OAlist\text{-}tc\text{-}map2\text{-}val-def by (rule list-of-oalist\text{-}tc\text{-}of-list-id, rule tc.oalist-inv\text{-}raw\text{-}map2\text{-}val\text{-}pair, \\
fact oalist-inv\text{-}list\text{-}of\text{-}oalist\text{-}tc, fact tc.oalist\text{-}inv\text{-}list\text{-}of\text{-}oalist\text{-}tc, \\
fact tc.map2\text{-}val\text{-}compat\text{-}map2\text{-}val\text{-}pair, fact tc.map2\text{-}val\text{-}compat\text{-}map2\text{-}val\text{-}pair)

**Lemma** list-of-oalist-OAlist\text{-}tc\text{-}map2\text{-}val\text{-}rneutr [simp, code abstract]:

\[
\text{list-of-oalist\text{-}tc} (OAlist\text{-}tc\text{-}map2\text{-}val\text{-}rneutr f xs ys) = \\
\text{map2\text{-}val\text{-}pair\text{-}tc} f \text{id} (\text{map\text{-}val\text{-}pair} (\lambda k c. f k 0 c)) (\text{list-of-oalist\text{-}tc} xs) \\
\text{list-of-oalist\text{-}tc} ys
\]

**Unfolding** OAlist\text{-}tc\text{-}map2\text{-}val\text{-}rneutr-def by (rule list-of-oalist\text{-}tc\text{-}of-list-id, rule tc.oalist-inv\text{-}raw\text{-}map2\text{-}val\text{-}pair, \\
fact oalist-inv\text{-}list\text{-}of\text{-}oalist\text{-}tc, fact tc.oalist-inv\text{-}list\text{-}of\text{-}oalist\text{-}tc, \\
fact tc.map2\text{-}val\text{-}compat\text{-}map2\text{-}val\text{-}pair, fact tc.map2\text{-}val\text{-}compat\text{-}map2\text{-}val\text{-}pair)

**Lemma** list-of-oalist-OAlist\text{-}tc\text{-}map2\text{-}val\text{-}rneutr [simp, code abstract]:

\[
\text{list-of-oalist\text{-}tc} (OAlist\text{-}tc\text{-}map2\text{-}val\text{-}rneutr f xs ys) = \text{map2\text{-}val\text{-}pair\text{-}tc} \text{id} \text{id} (\text{list-of-oalist\text{-}tc} xs) \\
\text{list-of-oalist\text{-}tc} ys
\]

**Unfolding** OAlist\text{-}tc\text{-}map2\text{-}val\text{-}rneutr-def by (rule list-of-oalist\text{-}tc\text{-}of-list-id, rule tc.oalist-inv\text{-}raw\text{-}map2\text{-}val\text{-}pair, \\
fact oalist-inv\text{-}list\text{-}of\text{-}oalist\text{-}tc, fact tc.oalist-inv\text{-}list\text{-}of\text{-}oalist\text{-}tc, \\
fact tc.map2\text{-}val\text{-}compat\text{-}id, fact tc.map2\text{-}val\text{-}compat\text{-}map2\text{-}val\text{-}pair, fact tc.map2\text{-}val\text{-}compat\text{-}map2\text{-}val\text{-}pair)

**Lemma** lookup-OAlist\text{-}tc\text{-}map2\text{-}val:

\[
\text{assumes } \forall k. f k 0 0 = 0 \\
\text{shows } OAlist\text{-}tc\text{-}lookup (OAlist\text{-}tc\text{-}map2\text{-}val f xs ys) k = f k (OAlist\text{-}tc\text{-}lookup xs k) \ (OAlist\text{-}tc\text{-}lookup ys k) \\
\text{by (simp add: OAlist\text{-}tc\text{-}lookup\text{-}def tc.lookup\text{-}pair\text{-}map2\text{-}val\text{-}pair tc.map2\text{-}val\text{-}compat\text{-}map2\text{-}val\text{-}pair\text{\text{-}assms oalist\text{-}inv\text{-}list\text{-}of\text{-}oalist\text{-}tc})}
\]

**Lemma** lookup-OAlist\text{-}tc\text{-}map2\text{-}val\text{-}rneutr:

\[
\text{assumes } \forall x. f k x 0 = x \\
\text{shows } OAlist\text{-}tc\text{-}lookup (OAlist\text{-}tc\text{-}map2\text{-}val\text{-}rneutr f xs ys) k = f k (OAlist\text{-}tc\text{-}lookup xs k) \ (OAlist\text{-}tc\text{-}lookup ys k) \\
\text{proof (simp add: OAlist\text{-}tc\text{-}lookup\text{-}def, rule tc.lookup\text{-}pair\text{-}map2\text{-}val\text{-}pair) \\
\text{fix } zs::('a \times 'b) \text{ list} \\
\text{assume tc.oalist\text{-}inv\text{-}raw zs} \\
\text{thus id zs = map\text{-}val\text{-}pair (\lambda k v. f k v 0) zs by (simp add: asms tc.map\text{-}pair\text{-}id)} \\
\text{qed (fact oalist\text{-}inv\text{-}list\text{-}of\text{-}oalist\text{-}tc, fact oalist\text{-}inv\text{-}list\text{-}of\text{-}oalist\text{-}tc, } \\
\text{fact tc.map2\text{-}val\text{-}compat\text{-}id, fact tc.map2\text{-}val\text{-}compat\text{-}map2\text{-}val\text{-}pair, rule refl, simp only: asms)}
\]

**Lemma** lookup-OAlist\text{-}tc\text{-}map2\text{-}val\text{-}rneutr:

\[
\text{assumes } \forall k. f k 0 0 = x \text{ and } \forall k. f k 0 x = x \\
\text{shows } OAlist\text{-}tc\text{-}lookup (OAlist\text{-}tc\text{-}map2\text{-}val\text{-}rneutr f xs ys) k = f k (OAlist\text{-}tc\text{-}lookup x y)
\]

353
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 (λk 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 (λk. 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 ⋀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 (λx. f k x) xs
  by (simp add: OAlist-tc-map2-val-rneutr-def OAlist-tc-update-by-fun-def assms
tc.map2-val-compat-id, simp only: assms(1))

12.10.11 OAlist-tc-lex-ord and OAlist-tc-prod-ord

lemma OAlist-tc-lex-ord-Eql:
  (⋀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-Eql,
  rule oalist-inv-list-of-oalist-tc, rule oalist-inv-list-of-oalist-tc, blast)

lemma OAlist-tc-lex-ord-ValI:
  assumes aux ≠ Some Eq and k ∈ fst ' set (list-of-oalist-tc xs) ∪ fst ' set
(list-of-oalist-tc ys)
  shows aux = f k (OAlist-tc-lookup xs k) (OAlist-tc-lookup ys k)
(⋀k1 k2. k1 ∈ fst ' set (list-of-oalist-tc xs) ∪ fst ' set (list-of-oalist-tc ys) ⇒
  k1 < k) ⇒
  f k1 (OAlist-tc-lookup xs k1) (OAlist-tc-lookup ys k1) = Some Eq)
⇒
  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 ⇒
  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
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:

354
assumes $O\text{alist-tc-lex-ord} f \, xs \, ys = \text{aux}$ and $\text{aux} \neq \text{Some Eq}$
obreakspace
obtains $k$ where $k \in \text{fst } \text{'set (list-of-oalist-tc xs) } \cup \text{fst } \text{'set (list-of-oalist-tc ys)}$
and $\text{aux} = f \, k \, (\text{OAlist-tc-lookup xs} \, k) \, (\text{OAlist-tc-lookup ys} \, k)$
and $\forall k', \, k' \in \text{fst } \text{'set (list-of-oalist-tc xs) } \cup \text{fst } \text{'set (list-of-oalist-tc ys)} \implies k' < k \implies f \, k' \, (\text{OAlist-tc-lookup xs} \, k') \, (\text{OAlist-tc-lookup ys} \, k') = \text{Some Eq}$

Eq

proof –

note oalist-inv-list-of-oalist-tc oalist-inv-list-of-oalist-tc
moreover from assms(1) have $\text{lex-ord-pair-tc} \, f \, (\text{list-of-oalist-tc} \, xs) \, (\text{list-of-oalist-tc} \, ys) = \text{aux}$
by (simp only: OAlist-tc-lex-ord-def)
ultimately obtain $k$ where $1: \, k \in \text{fst } \text{'set (list-of-oalist-tc xs) } \cup \text{fst } \text{'set (list-of-oalist-tc ys)}$
and $\text{aux} = f \, k \, (\text{lookup-pair-tc} \, (\text{list-of-oalist-tc} \, xs) \, k) \, (\text{lookup-pair-tc} \, (\text{list-of-oalist-tc} \, ys) \, k)$
and $\forall k', \, k' \in \text{fst } \text{'set (list-of-oalist-tc xs) } \cup \text{fst } \text{'set (list-of-oalist-tc ys)} \implies k' < k \implies f \, k' \, (\text{lookup-pair-tc} \, (\text{list-of-oalist-tc} \, xs) \, k') \, (\text{lookup-pair-tc} \, (\text{list-of-oalist-tc} \, ys) \, k') = \text{Some Eq}$
using assms(2) unfolding tc-le-lt[symmetric] by (rule tc.lex-ord-pair-valE, blast)

from this(2, 3) have $\text{aux} = f \, k \, (\text{OAlist-tc-lookup xs} \, k) \, (\text{OAlist-tc-lookup ys} \, k)$
and $\forall k', \, k' \in \text{fst } \text{'set (list-of-oalist-tc xs) } \cup \text{fst } \text{'set (list-of-oalist-tc ys)} \implies k' < k \implies f \, k' \, (\text{OAlist-tc-lookup xs} \, k') \, (\text{OAlist-tc-lookup ys} \, k') = \text{Some Eq}$

by (simp-all only: OAlist-tc-lookup-def)
with 1 show ?thesis ..

qed

lemma OAlist-tc-prod-ord-alt:
$\text{OAlist-tc-prod-ord} \, P \, xs \, ys \longleftrightarrow 
(\forall k \in \text{fst } \text{'set (list-of-oalist-tc xs) } \cup \text{fst } \text{'set (list-of-oalist-tc ys)}.
P \, k \, (\text{OAlist-tc-lookup xs} \, k) \, (\text{OAlist-tc-lookup ys} \, k))$
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 \ $xs \, ys = \ (\text{list-of-oalist-tc} \, xs = \text{list-of-oalist-tc} \, ys)$

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 OList-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 OList-tc-min-key-val** (oalist-tc-of-list \([(1, 3), (0::nat, 4::nat), (0, 2), (1, 1)])) = (0, 4)

by eval

**Lemma OList-tc-lookup** (oalist-tc-of-list \([(0::nat, 4::nat), (1, 3), (0, 2), (1, 1)])) 1 = 3

by eval

**Lemma OList-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 OList-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 OList-Poly-Mapping**

imports PP-Type MPoly-Type-Class-Ordered OList

begin

We introduce a dedicated type for ordered associative lists (oalists) representing polynomials. To that end, we require the order relation the oalists are sorted wrt. to be admissible term orders, and furthermore sort the lists descending rather than ascending, because this allows to implement various operations more efficiently. For technical reasons, we must restrict the type of terms to types embeddable into \((nat, nat) pp \times nat\), though. All types we are interested in meet this requirement.

**Lemma comparator-lexicographic:**

fixes f::'a ⇒ 'b and g::'a ⇒ 'c

assumes comparator c1 and comparator c2 and \(\forall x y. f x = f y \Rightarrow g x = g y\)

⇒ x = y

shows comparator (λx 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.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.trans)
thus \text{thesis by (simp add: 1 3)}
qed
qed
qed
qed

class \text{nat-term} =
  \text{fixes rep-nat-term :: } 'a \Rightarrow ((\text{nat}, \text{nat}) \text{ pp} \times \text{nat})
  \text{and splus :: } 'a \Rightarrow 'a \Rightarrow 'a
\text{assumes rep-nat-term-inj: rep-nat-term } x = \text{rep-nat-term } y \Rightarrow x = y
  \text{and full-component: } \text{snd (rep-nat-term } x) = i \Rightarrow (\exists y. \text{rep-nat-term } y = (t, i))
  \text{and splus-term: rep-nat-term (splus } x \ y) = \text{pprod.splus (fst (rep-nat-term } x)) (\text{rep-nat-term } y)
\begin{definition}
\text{definition lex-comp-aux = (λ } x \ y. \text{case comp-of-ord lex-pp (fst (rep-nat-term } x)) (fst (rep-nat-term } y)) \text{ of }
\begin{align*}
\text{Eq ⇒ comparator-of (snd (rep-nat-term } x)) (\text{snd (rep-nat-term } y)) | \text{val ⇒ val)}
\end{align*}
\end{definition}

\text{lemma full-componentE:}
  \text{assumes } \text{snd (rep-nat-term } x) = i
  \text{obtains } y \text{ where rep-nat-term } y = (t, i)
\text{proof –}
  \text{from assms have } \exists y. \text{rep-nat-term } y = (t, i) \text{ by (rule full-component)}
  \text{then obtain } y \text{ where rep-nat-term } y = (t, i) .
  \text{thus } \text{thesis} .
\end{lemma}
qed

end

class \text{nat-pp-term} = \text{nat-term} + \text{zero} + \text{plus} +
\text{assumes rep-nat-term-zero: rep-nat-term } 0 = (0, 0)
  \text{and splus-pp-term: splus = (+)}
\begin{definition}
\text{definition nat-term-comp :: } 'a::nat-term comparator ⇒ bool
\text{where nat-term-comp cmp a.}
\begin{align*}
\forall u v. \text{snd (rep-nat-term } u) = \text{snd (rep-nat-term } v) \Rightarrow \text{fst (rep-nat-term } u) = 0 \Rightarrow \text{cmp } u \ v \neq \text{Lt}) \land
\forall u v. \text{fst (rep-nat-term } u) = \text{fst (rep-nat-term } v) \Rightarrow \text{snd (rep-nat-term } u) < \text{snd (rep-nat-term } v) \Rightarrow \text{cmp } u \ v = \text{Lt}) \land
\forall t u v. \text{cmp } u \ v = \text{Lt} \Rightarrow \text{cmp } (\text{splus } t \ u) (\text{splus } t \ v) = \text{Lt}) \land
\forall u a b. \text{fst (rep-nat-term } u) = \text{fst (rep-nat-term } a) \Rightarrow \text{fst (rep-nat-term } v) = \text{fst (rep-nat-term } b) \Rightarrow \text{snd (rep-nat-term } u) = \text{snd (rep-nat-term } v) \Rightarrow \text{snd (rep-nat-term } a) = \text{snd (rep-nat-term } b) \Rightarrow \text{cmp } a \ b = \text{Lt} \Rightarrow \text{cmp } u \ v = \text{Lt})
\end{align*}
\end{definition}

358
lemma nat-term-comp1:
  assumes \( \forall u \, v. \, \text{snd} \, (\text{rep-nat-term} \, u) = \text{snd} \, (\text{rep-nat-term} \, v) \Rightarrow \text{fst} \, (\text{rep-nat-term} \, u) = 0 \Rightarrow \text{cmp} \, u \, v \neq \text{Gt} \)
  and \( \forall u \, v. \, \text{fst} \, (\text{rep-nat-term} \, u) = \text{fst} \, (\text{rep-nat-term} \, v) \Rightarrow \text{snd} \, (\text{rep-nat-term} \, u) < \text{snd} \, (\text{rep-nat-term} \, v) \Rightarrow \text{cmp} \, u \, v = \text{Lt} \)
  and \( \forall t \, u \, v. \, \text{cmp} \, u \, v = \text{Lt} \Rightarrow \text{cmp} \, (\text{plus} \, t \, u) \, (\text{plus} \, t \, v) = \text{Lt} \)
  and \( \forall a \, b. \, \text{fst} \, (\text{rep-nat-term} \, u) = \text{fst} \, (\text{rep-nat-term} \, a) \Rightarrow \text{fst} \, (\text{rep-nat-term} \, v) = \text{fst} \, (\text{rep-nat-term} \, b) \Rightarrow \text{snd} \, (\text{rep-nat-term} \, u) = \text{snd} \, (\text{rep-nat-term} \, a) \Rightarrow \text{cmp} \, u \, b = \text{Lt} \Rightarrow \text{cmp} \, u \, v = \text{Lt} \)
  shows \text{nat-term-comp cmp}
  unfolding \text{nat-term-comp-def fst-conv snd-conv} \text{ using} \text{ assms by blast}

lemma nat-term-compD1:
  assumes \text{nat-term-comp cmp} and \text{snd} \, (\text{rep-nat-term} \, u) = \text{snd} \, (\text{rep-nat-term} \, v)
  and \text{fst} \, (\text{rep-nat-term} \, u) = 0
  shows \text{cmp} \, u \, v \neq \text{Gt}
  using \text{assms unfolding} \text{ nat-term-comp-def fst-conv by blast}

lemma nat-term-compD2:
  assumes \text{nat-term-comp cmp} and \text{fst} \, (\text{rep-nat-term} \, u) = \text{fst} \, (\text{rep-nat-term} \, v)
  and \text{snd} \, (\text{rep-nat-term} \, u) < \text{snd} \, (\text{rep-nat-term} \, v)
  shows \text{cmp} \, u \, v = \text{Lt}
  using \text{assms unfolding} \text{ nat-term-comp-def fst-conv snd-conv by blast}

lemma nat-term-compD3:
  assumes \text{nat-term-comp cmp} and \text{cmp} \, u \, v = \text{Lt}
  shows \text{cmp} \, (\text{plus} \, t \, u) \, (\text{plus} \, t \, v) = \text{Lt}
  using \text{assms unfolding} \text{ nat-term-comp-def snd-conv by blast}

lemma nat-term-compD4:
  assumes \text{nat-term-comp cmp} and \text{fst} \, (\text{rep-nat-term} \, u) = \text{fst} \, (\text{rep-nat-term} \, a)
  and \text{fst} \, (\text{rep-nat-term} \, v) = \text{fst} \, (\text{rep-nat-term} \, b) and \text{snd} \, (\text{rep-nat-term} \, u) = \text{snd} \, (\text{rep-nat-term} \, a)
  and \text{snd} \, (\text{rep-nat-term} \, b) = \text{cmp} \, a \, b = \text{Lt}
  shows \text{cmp} \, u \, v = \text{Lt}
  using \text{assms unfolding} \text{ nat-term-comp-def snd-conv by blast}

lemma nat-term-compD1':
  assumes \text{comparator cmp} and \text{nat-term-comp cmp} and \text{snd} \, (\text{rep-nat-term} \, u) \leq \text{snd} \, (\text{rep-nat-term} \, v)
  and \text{fst} \, (\text{rep-nat-term} \, u) = 0
  shows \text{cmp} \, u \, v \neq \text{Gt}
  proof (cases \text{snd} \, (\text{rep-nat-term} \, u) = \text{snd} \, (\text{rep-nat-term} \, v))
    case True
    with \text{assms(2)} show ?thesis using \text{assms(4)} by (rule nat-term-compD1)
  next
    from \text{assms(1)} interpret \text{cmp: comparator cmp} .

359
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-eql)
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

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 w: 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.trans)
qed

lemma comparator-lex-comp-aux: comparator (lex-comp-aux::'a::nat-term compara-
  tor)
  unfolding lex-comp-aux-def
proof (rule comparator-composition)
  from lex-pp-antisym have as: antisym lex-pp by (rule antisympl)
  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,
  thus comparator (λx y::((nat, nat) pp × nat). case comp-of-ord lex-pp (fst x)
    (fst y) of
      Eq ⇒ comparator-of (snd x) (snd x) | (snd x) ⇒ (snd x)
    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 compara-
  tor)
proof -
  from lex-pp-antisym have as: antisym lex-pp by (rule antisympl)
  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,
show thesis
proof (rule nat-term-comp1)
  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 = (=)

362
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 = (λx 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-comp by blast

lemma nat-term-comp-nat-term-compare: nat-term-comp (nat-term-compare to)
  using nat-term-comp 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-comp 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-comp to v u = Lt using cmp.Gt-lt-conv cmp.Lt-lt-conv by
```isar
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) = Lt using cmp.Gt-lt-conv

  
  cmp.Gt-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 = (λu v. ko.le (key-order-of-nat-term-order-inv to) v u)


  le-of-key-order-def nat-term-compare-inv-conv[symmetric] nat-term-compare-inv-alt)

  lemma lt-of-nat-term-order-alt:
  
  lt-of-nat-term-order to = (λu v. ko.lt (key-order-of-nat-term-order-inv to) v u)


  lt-of-key-order-def nat-term-compare-inv-conv[symmetric] nat-term-compare-inv-alt)

```
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

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)

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 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)


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

365
and \( tl\text{-ko-ntm} = ko\text{-ntm}.tl\text{-raw} \)
and \( min\text{-key-val-ko-ntm} = ko\text{-ntm}.min\text{-key-val-raw} \)
and \( update\text{-by-ko-ntm} = ko\text{-ntm}.update\text{-by-raw} \)
and \( update\text{-by-fun-ko-ntm} = ko\text{-ntm}.update\text{-by-fun-raw} \)
and \( update\text{-by-fun-gr-ko-ntm} = ko\text{-ntm}.update\text{-by-fun-gr-raw} \)
and \( map2\text{-val-ko-ntm} = ko\text{-ntm}.map2\text{-val-raw} \)
and \( lex\text{-ord-ko-ntm} = ko\text{-ntm}.lex\text{-ord-raw} \)
and \( prod\text{-ord-ko-ntm} = ko\text{-ntm}.prod\text{-ord-raw} \)
and \( oalist\text{-eq-ko-ntm} = ko\text{-ntm}.oalist\text{-eq-raw} \)
and \( sort\text{-oalist-ko-ntm} = ko\text{-ntm}.sort\text{-oalist-raw} \)

\begin{itemize}
\item subgoal by (simp only: lookup-pair-ko-ntm-def nat-term-compare-inv-conv)
\item subgoal by (simp only: update-by-pair-ko-ntm-def nat-term-compare-inv-conv)
\item subgoal by (simp only: update-by-fun-pair-ko-ntm-def nat-term-compare-inv-conv)
\item subgoal by (simp only: update-by-fun-gr-pair-ko-ntm-def nat-term-compare-inv-conv)
\item subgoal by (simp only: map2-val-pair-ko-ntm-def nat-term-compare-inv-conv)
\item subgoal by (simp only: lex-ord-pair-ko-ntm-def nat-term-compare-inv-conv)
\item subgoal by (simp only: prod-ord-pair-ko-ntm-def nat-term-compare-inv-conv)
\item subgoal by (simp only: sort-oalist-ko-ntm′-def nat-term-compare-inv-conv)
\end{itemize}

\textbf{done}

\textbf{lemma} compute-min-key-val-ko-ntm [code]:
\[
\text{min-key-val-ko-ntm } ko\ (xs, ox) =
\begin{cases}
\text{if } ko = ox \text{ then } hd \text{ else } \text{min-list-param } (\lambda x y. (le-of-nat-term-order ko) (fst y)) (fst x) \end{cases}
\]
\textbf{proof} –

\begin{itemize}
\item have \( ko.le (key-order-of-nat-term-order-inv ko) = (\lambda x y. le-of-nat-term-order ko y x) \)
\item by (metis ko.nGt-le-conv le-of-nat-term-order nat-term-compare-inv-conv nat-term-compare-inv-def)
\end{itemize}
\textbf{thus} ?thesis by (simp only: min-key-val-ko-ntm-def oalist-raw.min-key-val-raw.simps)
\textbf{qed}

\textbf{typedef} (overloaded) \((a, 'b)\) \(oalist\text{-ntm} = \)
\[
\{zx::(a, 'b::zero, 'a::nat-term nat-term-order) oalist-raw. ko-ntm.oalist-inv xs\}
\]
\textbf{morphisms} list-of-oalist-ntm \(Abs\text{-oalist-ntm}\)
by (auto simp: ko-ntm.oalist-inv-def intro: ko-alist-inv-raw-Nil)

\textbf{lemma} oalist-ntm-eq-iff: \(xs = ys \iff list\text{-of-oalist-ntm} xs = list\text{-of-oalist-ntm} ys\)
by (simp add: list-of-oalist-ntm-inject)

\textbf{lemma} oalist-ntm-eqI: \(list\text{-of-oalist-ntm} xs = list\text{-of-oalist-ntm} ys \implies xs = ys\)
by (simp add: oalist-ntm-eq-iff)

Formal, totalized constructor for \((a, 'b)\) \(oalist\text{-ntm}:\)

\textbf{definition} \(O\text{Alist-ntm} :: (a \times 'b) list \times 'a nat-term-order \Rightarrow (a::nat-term, 'b::zero) oalist\text{-ntm}\)
\textbf{where} \(O\text{Alist-ntm} xs = Abs\text{-oalist-ntm} (sort\text{-oalist-ko-ntm} xs)\)

\textbf{definition} oalist-of-list-ntm = \(O\text{Alist-ntm}\)
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'
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 oo-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 OAlias-ntm
list-of-oalist-ntm::(′a,’c) oalist-ntm ⇒ (′a,’c::zero,’a nat-term-order) oalist-raw
OAlias-ntm
list-of-oalist-ntm::(′a,’d) oalist-ntm ⇒ (′a,’d::zero,’a nat-term-order) oalist-raw
OAlias-ntm

defines OList-map-val-ntm = oo-ntm.map-val
and OList-map2-val-ntm = oo-ntm.map2-val
and OList-map2-val-rneutr-ntm = oo-ntm.map2-val-rneutr
and OList-lex-ord-ntm = oo-ntm.lex-ord
and OList-prod-ord-ntm = oo-ntm.prod-ord ..

lemmas OList-lookup-ntm-single = oo-ntm.lookup-oalist-of-list-single[folded oalist-of-list-ntm-def]
end

14 Computable Term Orders

theory Term-Order
  imports OList-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

end
shows \( m = n \)

proof

- have \( \text{rep-nat} \ (\text{abs-nat} \ m) = \text{rep-nat} \ (\text{abs-nat} \ n) \) by (simp only: assms)

thus \( \text{?thesis} \) by (simp only: abs-inverse)

qed

corollary \( \text{abs-eq-iff} \): \((\text{abs-nat} \ m = \text{abs-nat} \ n) \iff (m = n)\)

by (auto elim: abs-inj)

lemma \( \text{rep-zero} \) [simp]: \( \text{rep-nat} \ 0 = 0 \)

using abs-inverse abs-zero by fastforce

lemma \( \text{rep-zero-iff} \): \((\text{rep-nat} \ x = 0) \iff (x = 0)\)

using rep-eq-iff by fastforce

lemma \( \text{plus-eq} \): \( x + y = \text{abs-nat} \ (\text{rep-nat} \ x + \text{rep-nat} \ y) \)

by (metis abs-plus rep-inverse)

lemma \( \text{rep-plus} \): \( \text{rep-nat} \ (x + y) = \text{rep-nat} \ x + \text{rep-nat} \ y \)

by (simp add: plus-eq)

lemma \( \text{minus-eq} \): \( x - y = \text{abs-nat} \ (\text{rep-nat} \ x - \text{rep-nat} \ y) \)

by (metis abs-minus rep-inverse)

lemma \( \text{rep-minus} \): \( \text{rep-nat} \ (x - y) = \text{rep-nat} \ x - \text{rep-nat} \ y \)

by (simp add: minus-eq)

lemma \( \text{ord-iff} \):

\( x \leq y \iff \text{rep-nat} \ x \leq \text{rep-nat} \ y \) (is \( \text{?thesis1} \))

\( x < y \iff \text{rep-nat} \ x < \text{rep-nat} \ y \) (is \( \text{?thesis2} \))

proof

- show \( \text{?thesis1} \)

  proof

  assume \( x \leq y \)

  show \( \text{rep-nat} \ x \leq \text{rep-nat} \ y \)

  proof (rule ccontr)

  assume \( \neg \text{rep-nat} \ x \leq \text{rep-nat} \ y \)

  hence \( \text{rep-nat} \ y \leq \text{rep-nat} \ x \) and \( \text{rep-nat} \ x \neq \text{rep-nat} \ y \) by simp-all

  from this(1) have \( \text{abs-nat} \ (\text{rep-nat} \ y) \leq \text{abs-nat} \ (\text{rep-nat} \ x) \) by (rule abs-ord)

  hence \( y \leq x \) by (simp only: rep-inverse)

  moreover from \( \text{rep-nat} \ x \neq \text{rep-nat} \ y \) have \( y \neq x \) using rep-inj by auto

  ultimately have \( y < x \) by simp

  with \( x \leq y \) show False by simp

  qed

next

  assume \( \text{rep-nat} \ x \leq \text{rep-nat} \ y \)

  hence \( \text{abs-nat} \ (\text{rep-nat} \ x) \leq \text{abs-nat} \ (\text{rep-nat} \ y) \) by (rule abs-ord)

  thus \( x \leq y \) by (simp only: rep-inverse)

  qed

369
thus \( ?\thesis2 \) using \( \text{rep-inj[of x y]} \) by (auto simp: less-le Nat.nat-less-le)

qed

lemma ex-iff-abs: \((\exists x::'a. \ P x) \iff (\exists n::nat. \ P (abs-nat n))\)
  by (metis rep-inverse)

lemma ex-iff-abs': \((\exists x<abs-nat m. \ P x) \iff (\exists n::nat<m. \ P (abs-nat n))\)
  by (metis abs-inverse rep-inverse ord-iff)

lemma all-iff-abs: \((\forall x::'a. \ P x) \iff (\forall n::nat. \ P (abs-nat n))\)
  by (metis rep-inverse)

lemma all-iff-abs': \((\forall x<abs-nat m. \ P x) \iff (\forall n::nat<m. \ P (abs-nat n))\)
  by (metis abs-inverse rep-inverse ord-iff)

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 \( ?\ = \lambda n::nat. \ P (abs-nat n) \)
  assume a: \( \forall x. (\forall y. \ y < x \implies P y) \implies P x \)
  have P (abs-nat (rep-nat a))
  proof (rule less-induct[of - rep-nat a])
    fix n::nat
    assume b: \( \forall m. \ m < n \implies ?P m \)
    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
    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: \( \text{sum (rep-nat o f) A = rep-nat (sum f A)} \) for f :: 'b ⇒ 'a and A :: 'b set
proof (induct A rule: infinite-finite-induct)
case (infinite A)
  thus \( ?\case \) by simp

370
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 (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 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 = (λx. x)
definition abs-nat-nat :: nat ⇒ nat where abs-nat-nat-def [code-unfold]: abs-nat-nat = (λx. 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 \Rightarrow nat
  where rep-nat-natural-def [code-unfold]: rep-nat-natural = nat-of-natural
definition abs-nat-natural :: nat \Rightarrow natural
  where abs-nat-natural-def [code-unfold]: abs-nat-natural = natural-of-nat


end

14.2 Term Orders

14.2.1 Type Classes
class nat-pp-compare = linorder + zero + plus +
fixes rep-nat-pp :: 'a \Rightarrow (nat, nat) pp
  and abs-nat-pp :: (nat, nat) pp \Rightarrow 'a
  and lex-comp' :: 'a comparator
  and deg' :: 'a \Rightarrow 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 \leq rep-nat-pp y \implies x \leq 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 \leq rep-nat-pp y and 2: rep-nat-pp x \neq rep-nat-pp y by simp-all
  from 1 have x \leq y by (rule le-pp)
  moreover from 2 have x \neq 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

372
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. lookup-pp (rep-nat-pp s) y = lookup-pp (rep-nat-pp t) y))
  by (simp add: lex-comp' split: if-split_asm)
  then obtain x where 1: lookup-pp (rep-nat-pp s) x < 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
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 \( \text{rep-nat-pp } t = \text{rep-nat-pp } s \)
moreover from this have \( \text{lex-pp } (\text{rep-nat-pp } s) (\text{rep-nat-pp } t) \) by (simp add: \( \text{lex-pp-refl} \))
ultimately have \( \text{lex-comp'} s t = \text{Eq} \) by (simp add: \( \text{lex-comp'} \text{ comp-of-ord-def} \))
with \( \text{Gt} \) show False by simp
qed
ultimately have \( \exists x. \text{lookup-pp } (\text{rep-nat-pp } t) x < \text{lookup-pp } (\text{rep-nat-pp } s) x \)
and \( 2: \forall y. y < x \implies \text{lookup-pp } (\text{rep-nat-pp } t) y = \text{lookup-pp } (\text{rep-nat-pp } s) y \)
by (simp add: \( \text{lex-pp-alt} \))
then obtain \( x \) where
\( 1: \) lookup-pp \( (\text{rep-nat-pp } t) x < \text{lookup-pp } (\text{rep-nat-pp } s) x \)
and \( 2: \forall y. y < x \implies \text{lookup-pp } (\text{rep-nat-pp } t) y = \text{lookup-pp } (\text{rep-nat-pp } s) y \)
by blast
show \( ?\text{thesis} \)
proof
  show \( x \in \text{keys-pp } (\text{rep-nat-pp } s) \cup \text{keys-pp } (\text{rep-nat-pp } t) \)
  proof (rule ccontr)
    assume \( x \notin \text{keys-pp } (\text{rep-nat-pp } s) \cup \text{keys-pp } (\text{rep-nat-pp } t) \)
    with \( 1 \) show False by (simp add: \( \text{keys-pp-iff} \))
  qed
next
  from \( 1 \) have \( \neg \text{lookup-pp } (\text{rep-nat-pp } s) x < \text{lookup-pp } (\text{rep-nat-pp } t) x \)
  and \( \text{lookup-pp } (\text{rep-nat-pp } s) x \neq \text{lookup-pp } (\text{rep-nat-pp } t) x \) by simp-all
  thus \( \text{comparator-of } (\text{lookup-pp } (\text{rep-nat-pp } s) x) (\text{lookup-pp } (\text{rep-nat-pp } t) x) = \text{lex-comp'} s t \)
  by (simp add: \( \text{linorder-class.comparator-of-def Gt} \))
  qed (simp add: \( 2 \))
qed
end
class \( \text{nat-term-compare} = \text{linorder } + \text{nat-term } + \)
fixes \( \text{is-scalar} :: 'a \text{ itself } \Rightarrow \text{bool} \)
  and \( \text{lex-comp} :: 'a \text{ comparator } \)
  and \( \text{deg-comp} :: 'a \text{ comparator } \Rightarrow 'a \text{ comparator} \)
  and \( \text{pot-comp} :: 'a \text{ comparator } \Rightarrow 'a \text{ comparator} \)
assumes \( \text{zero-component}: \exists x. \text{snd } (\text{rep-nat-term } x) = 0 \)
  and \( \text{is-parallel}: \text{is-scalar } = (\lambda x. \forall x. \text{snd } (\text{rep-nat-term } x) = 0) \)
  and \( \text{lex-comp}: \text{lex-comp } = \text{lex-comp-aux} \)
  — For being able to implement \( \text{lex-comp} \) efficiently.
  and \( \text{deg-comp}: \text{deg-comp } cmp = (\lambda x y. \text{case comparator-of } (\text{deg-pp } (\text{fst } (\text{rep-nat-term } x))) (\text{deg-pp } (\text{fst } (\text{rep-nat-term } y))) \text{ of } \text{Eq } \Rightarrow \text{ cmp } x y | \text{ val } \Rightarrow \text{ val} \)
  and \( \text{pot-comp}: \text{pot-comp } cmp = (\lambda x y. \text{case comparator-of } (\text{snd } (\text{rep-nat-term } x)) (\text{snd } (\text{rep-nat-term } y)) \text{ of } \text{Eq } \Rightarrow \text{ cmp } x y | \text{ val } \Rightarrow \text{ val} \)
  and \( \leq\text{-term}: \text{rep-nat-term } x \leq \text{rep-nat-term } y \Rightarrow x \leq y \)
begin
  There is no need to add something like \( \text{top-comp} \) for TOP orders to class \( \text{nat-term-compare} \), because by default all comparators should first compare power-products and then positions. \( \text{lex-comp} \) 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 antisympl)
  interpret lex: comparator comp-of-ord (lex-pp::(nat, nat) pp ⇒ -)
  unfolding comp-of-ord-comp-of-ords[OF as]
  by (rule comp-of-ords, unfold-locales,
  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-eq-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) 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 $u$ $v$ $\neq$ 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 $\vphantom{\text{fst}}$ fst (rep-nat-term $u$) = $\vphantom{\text{fst}}$ 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 \( \text{cmp} \ u \ v \neq \text{Gt} \) by (simp add: \( u = v \))

qed

lemma \( \text{deg-comp-pos} \):
  assumes \( \text{cmp} \ u \ v = \text{Lt} \) and \( \text{fst} \ (\text{rep-nat-term} \ u) = \text{fst} \ (\text{rep-nat-term} \ v) \)
  shows \( \text{deg-comp} \ \text{cmp} \ u \ v = \text{Lt} \)
  by (simp add: \( \text{deg-comp} \) \( \text{assms} \) split: order.split)

lemma \( \text{deg-comp-monotone} \):
  assumes \( \text{cmp} \ u \ v = \text{Lt} \Longrightarrow \text{cmp} \ (\text{splus} \ t \ u) \ (\text{splus} \ t \ v) = \text{Lt} \) and \( \text{deg-comp} \ \text{cmp} \ u \ v = \text{Lt} \)
  shows \( \text{deg-comp} \ \text{cmp} \ (\text{splus} \ t \ u) \ (\text{splus} \ t \ v) = \text{Lt} \)
  using \( \text{assms}(2) \) by (auto simp: \( \text{deg-comp} \) \( \text{splus-term} \) \( \text{pprod} \) \( \text{splus-def} \) \( \text{comparator-of-def} \)
  \( \text{deg-pp-plus} \)
  split: order.split if-splits intro: \( \text{assms}(1) \))

lemma \( \text{pot-comp-zero-min} \):
  assumes \( \text{cmp} \ u \ v \neq \text{Gt} \) and \( \text{snd} \ (\text{rep-nat-term} \ u) = \text{snd} \ (\text{rep-nat-term} \ v) \)
  shows \( \text{pot-comp} \ \text{cmp} \ u \ v \neq \text{Gt} \)
  by (simp add: \( \text{pot-comp} \) \( \text{comparator-of-def} \) \( \text{assms} \) split: order.split)

lemma \( \text{pot-comp-pos} \):
  assumes \( \text{snd} \ (\text{rep-nat-term} \ u) < \text{snd} \ (\text{rep-nat-term} \ v) \)
  shows \( \text{pot-comp} \ \text{cmp} \ u \ v = \text{Lt} \)
  by (simp add: \( \text{pot-comp} \) \( \text{comparator-of-def} \) \( \text{assms} \) split: order.split)

lemma \( \text{pot-comp-monotone} \):
  assumes \( \text{cmp} \ u \ v = \text{Lt} \Longrightarrow \text{cmp} \ (\text{splus} \ t \ u) \ (\text{splus} \ t \ v) = \text{Lt} \) and \( \text{pot-comp} \ \text{cmp} \ u \ v = \text{Lt} \)
  shows \( \text{pot-comp} \ \text{cmp} \ (\text{splus} \ t \ u) \ (\text{splus} \ t \ v) = \text{Lt} \)
  using \( \text{assms}(2) \) by (auto simp: \( \text{pot-comp} \) \( \text{splus-term} \) \( \text{pprod} \) \( \text{splus-def} \) \( \text{comparator-of-def} \)
  \( \text{deg-pp-plus} \)
  split: order.split if-splits intro: \( \text{assms}(1) \))

lemma \( \text{deg-comp-cong} \):
  assumes \( \text{deg-pp} \ (\text{fst} \ (\text{rep-nat-term} \ u)) = \text{deg-pp} \ (\text{fst} \ (\text{rep-nat-term} \ v)) \) \( \Longrightarrow \text{to1} \)
  \( u \ v = \text{to2} \ u \ v \)
  shows \( \text{deg-comp} \ \text{to1} \ u \ v = \text{deg-comp} \ \text{to2} \ u \ v \)
  using \( \text{assms} \) by (simp add: \( \text{deg-comp} \) \( \text{comparator-of-def} \) split: order.split)

lemma \( \text{pot-comp-cong} \):
  assumes \( \text{snd} \ (\text{rep-nat-term} \ u) = \text{snd} \ (\text{rep-nat-term} \ v) \) \( \Longrightarrow \text{to1} \ u \ v = \text{to2} \ u \ v \)
  shows \( \text{pot-comp} \ \text{to1} \ u \ v = \text{pot-comp} \ \text{to2} \ u \ v \)
  using \( \text{assms} \) by (simp add: \( \text{pot-comp} \) \( \text{comparator-of-def} \) split: order.split)

instantiation \( \text{pp} \) :: \( \text{nat} \), \( \text{nat} \) \( \text{nat-pp-compare} \)
begin

definition \( \text{rep-nat-pp-pp} \) :: \( \text{('a}, \ 'b) \text{pp} 
  (\text{nat}, \text{nat}) \text{pp} \)}
where \( \text{rep-nat-pp-pp-def} \) [code del]: \( \text{rep-nat-pp-pp \ x = pp-of-fun (\lambda n::nat. \text{rep-nat (lookup-pp \ x \ (abs-nat \ n))})} \)

**definition** abs-nat-pp-pp :: (nat, nat) pp \( \Rightarrow ('a, 'b) \) pp
  where abs-nat-pp-pp-def [code del]: abs-nat-pp-pp \( t = pp-of-fun (\lambda n::'a. \text{abs-nat (lookup-pp \ t \ (rep-nat \ n))}) \)

**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 \( \Rightarrow \) nat
  where deg’-pp x = rep-nat (deg-pp x)

**lemma** lookup-rep-nat-pp-pp:
  lookup-pp (rep-nat-pp-pp \ t) = (\lambda n::nat. \text{rep-nat (lookup-pp \ t \ (abs-nat \ n))})

**unfolding** rep-nat-pp-pp-def

**proof** (rule lookup-pp-of-fun)
  have \( \{ n. \text{lookup-pp} \ t \ (\text{abs-nat} \ n) \neq 0 \} \subseteq \text{rep-nat} \ { x. \text{lookup-pp} \ t \ x \neq 0 } \)
  proof
    fix \( n \)
    have \( n = \text{rep-nat} \ (\text{abs-nat} \ n) \) by (simp only: nat-class.abs-inverse)
    assume \( n \in \{ n. \text{lookup-pp} \ t \ (\text{abs-nat} \ n) = 0 \} \)
    hence \( \text{abs-nat} \ n \in \{ x. \text{lookup-pp} \ t \ x = 0 \} \) by simp
    with \( n = \text{rep-nat} \ (\text{abs-nat} \ n) \) show \( n \in \text{rep-nat} \ { x. \text{lookup-pp} \ t \ x \neq 0 } \) ..
  qed
  also have finite ... by (rule finite-imageI, transfer, simp)
  also (finite-subset) have \( \{ n. \text{lookup-pp} \ t \ (\text{abs-nat} \ n) = 0 \} = \{ n. \text{rep-nat} \ (\text{lookup-pp} \ t \ (\text{abs-nat} \ n)) = 0 \} \)
    by (metis rep-inj rep-zero)
  finally show finite \( \{ x. \text{rep-nat} \ (\text{lookup-pp} \ t \ (\text{abs-nat} \ x)) \neq 0 \} \).
  qed

**lemma** lookup-abs-nat-pp-pp:
  lookup-pp (abs-nat-pp-pp \ t) = (\lambda n::'a. \text{abs-nat (lookup-pp \ t \ (rep-nat-pp-pp \ n))})

**unfolding** abs-nat-pp-pp-def

**proof** (rule lookup-pp-of-fun)
  have \( \{ n::'a. \text{lookup-pp} \ t \ (\text{rep-nat} \ n) \neq 0 \} \subseteq \text{abs-nat} \ { x. \text{lookup-pp} \ t \ x \neq 0 } \)
  proof
    fix \( n :: 'a \)
    have \( n = \text{abs-nat} \ (\text{rep-nat} \ n) \) by (simp only: nat-class.rep-inverse)
    assume \( n \in \{ n. \text{lookup-pp} \ t \ (\text{rep-nat} \ n) = 0 \} \)
    hence \( \text{rep-nat} \ n \in \{ x. \text{lookup-pp} \ t \ x = 0 \} \) by simp
    with \( n = \text{abs-nat} \ (\text{rep-nat} \ n) \) show \( n \in \text{abs-nat} \ { x. \text{lookup-pp} \ t \ x \neq 0 } \) ..
  qed
  also have finite ... by (rule finite-imageI, transfer, simp)
  also (finite-subset) have \( \{ n::'a. \text{lookup-pp} \ t \ (\text{rep-nat} \ n) \neq 0 \} = \{ n. \text{abs-nat} \ (\text{lookup-pp} \ t \ (\text{rep-nat-pp-pp} \ n)) \neq 0 \} \)
    by (metis abs-inverse abs-zero)
  finally show finite \( \{ n::'a. \text{abs-nat (lookup-pp \ t \ (rep-nat-pp-pp \ n))} \neq 0 \} \).
  qed
  by (rule set-eqI,

lemma rep-nat-pp-pp-inverse: abs-nat-pp (rep-nat-pp x) = x for x :: ('a, 'b) pp

lemma abs-nat-pp-pp-inverse: rep-nat-pp ((abs-nat-pp t):: ('a, 'b) pp) = t

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) \iff (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) \iff lex-pp x y for x y :: ('a, 'b) pp
       ord-iff[symmetric] ex-iff-abs[where 'a= 'a] all-iff-abs[prime])

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

corollary le-pp-pp: rep-nat-pp x \leq rep-nat-pp y \iff x \leq 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
  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 \ldots = sum (lookup-pp (rep-nat-pp t) \circ rep-nat) (keys-pp t)
    by (rule sum.reindex, rule inj-onI, elim rep-inj)
also have \ldots = sum (rep-nat \circ (lookup-pp t)) (keys-pp t)
    by (simp add: lookup-rep-nat-pp-pp)
also have \ldots = rep-nat (deg-pp t) by (simp only: deg-pp-alt sum-rep)
finally show ?thesis .
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)

   for x y :: ('a, 'b) pp

instance proof
  fix x y :: ('a, 'b) pp
  assume rep-nat-term x = rep-nat-term y
  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
\[ \text{show} \ \text{rep-nat-term} \ (\text{splus} \ x \ y) = \text{pprod} \ . \text{splus} \ (\text{fst} \ \text{rep-nat-term} \ x) \ (\text{rep-nat-term} \ y) \]
\[ \text{by} \ (\text{simp add: splus-pp-def rep-nat-term-pp-def pprod.splus-def plus-pp-pp}) \]
\text{qed} 

end

\text{ instantiation \ pp :: (nat, nat) nat-term-compare}
\begin{verbatim}
begin
\end{verbatim}

\text{definition} \ \text{is-scalar-pp} :: (\ 'a, \ 'b) \ pp \ \text{itself} \ \Rightarrow \ \text{bool} 
\text{where} \ \text{is-scalar-pp-def [code-unfold]}: \ \text{is-scalar-pp} = (\lambda . \ \text{True})

\text{definition} \ \text{lex-comp-pp} :: (\ 'a, \ 'b) \ pp \ \text{comparator} 
\text{where} \ \text{lex-comp-pp-def [code-unfold]}: \ \text{lex-comp-pp} = \ \text{lex-comp}'

\text{definition} \ \text{deg-comp-pp} :: (\ 'a, \ 'b) \ pp \ \text{comparator} \Rightarrow (\ 'a, \ 'b) \ pp \ \text{comparator} 
\text{where} \ \text{deg-comp-pp-def: deg-comp-pp cmp} = (\lambda x \ y. \ \text{case comparator-of} \ (\text{deg-pp} \ x) \ (\text{deg-pp} \ y) \ \text{of} \ \text{Eq} \Rightarrow \ cmp \ x \ y | \ \text{val} \Rightarrow \ \text{val})

\text{definition} \ \text{pot-comp-pp} :: (\ 'a, \ 'b) \ pp \ \text{comparator} \Rightarrow (\ 'a, \ 'b) \ pp \ \text{comparator} 
\text{where} \ \text{pot-comp-pp-def [code-unfold]}: \ \text{pot-comp-pp} = (\lambda cmp. \ cmp)

\text{instance \ proof}
\text{show} \ \exists x :: (\ 'a, \ 'b) \ pp, \ \text{snd} \ (\text{rep-nat-term} \ x) = 0 
\text{proof}
\text{show} \ \text{snd} \ (\text{rep-nat-term} \ (0 :: (\ 'a, \ 'b) \ pp)) = 0 \ \text{by} \ (\text{simp add: rep-nat-term-pp-def}) 
\text{qed}
\text{next}
\text{show} \ \text{is-scalar} = (\lambda :: (\ 'a, \ 'b) \ pp \ \text{itself}. \ \forall x :: (\ 'a, \ 'b) \ pp, \ \text{snd} \ (\text{rep-nat-term} \ x) = 0) 
\text{by} \ (\text{simp add: is-scalar-pp-def rep-nat-term-pp-def})
\text{next}
\text{show} \ \text{lex-comp} = (\text{lex-comp-aux} :: (\ 'a, \ 'b) \ pp \ \text{comparator}) 
\text{by} \ (\text{auto simp: lex-comp-pp-def lex-comp-aux-def rep-nat-term-pp-def lex-comp'-pp split: order.split intro!: ext})
\text{next}
\text{fix cmp :: (\ 'a, \ 'b) \ pp \ \text{comparator}}
\text{show} \ \text{deg-comp cmp} = 
\text{(\lambda x y. \ \text{case comparator-of} \ (\text{deg-pp} \ (\text{fst} \ \text{rep-nat-term} \ x))) \ (\text{deg-pp} \ (\text{fst} \ \text{rep-nat-term} \ y))) \ \text{of} \ \text{Eq} \Rightarrow \ cmp \ x \ y | \ \text{Lt} \Rightarrow \ \text{Lt} | \ \text{Gt} \Rightarrow \ \text{Gt}) 
\text{by} \ (\text{simp add: rep-nat-term-pp-def deg-comp-pp-def deg-rep-nat-pp comparator-of-rep})
\text{next}
\text{fix cmp :: (\ 'a, \ 'b) \ pp \ \text{comparator}}
\text{show} \ \text{pot-comp cmp} = 
\text{(\lambda x y. \ \text{case comparator-of} \ (\text{snd} \ \text{rep-nat-term} \ x)) \ (\text{snd} \ \text{rep-nat-term} \ y)) \ \text{of} \ \text{Eq} \Rightarrow \ cmp \ x \ y | \ \text{Lt} \Rightarrow \ \text{Lt} | \ \text{Gt} \Rightarrow \ \text{Gt}) 
\text{by} \ (\text{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
  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)
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)
qed
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
    383
show pot-comp cmp =
  (\x y. case comparator-of (snd (rep-nat-term x)) (snd (rep-nat-term y)) of
  Eq \Rightarrow cmp x y | Lt \Rightarrow Lt | Gt \Rightarrow Gt)
next
fix x y :: 'a \times 'b
assume rep-nat-term x \leq rep-nat-term y
hence rep-nat-pp (fst x) < rep-nat-pp (fst y) \lor (rep-nat-pp (fst x) \leq rep-nat-pp (fst y) \land rep-nat (snd x) \leq rep-nat (snd y))
by (simp add: rep-nat-term-prod-def)
thus x \leq y by (auto simp: less-eq-prod-def ord-iff [symmetric] intro: le-pp less-pp)
qed
end

= (\x. x)

14.2.2 LEX, DRLEX, DEG and POT

definition LEX :: 'a::nat-term-order => nat-term-order
  where LEX = Abs-nat-term-order lex-comp

definition DRLEX :: 'a::nat-term-order
  where DRLEX = Abs-nat-term-order (deg-comp (pot-comp (\x y. lex-comp y x)))

definition DEG :: 'a::nat-term-order => 'a nat-term-order
  where DEG to = Abs-nat-term-order (deg-comp (nat-term-order to))

definition POT :: 'a::nat-term-order => 'a nat-term-order
  where POT to = Abs-nat-term-order (pot-comp (nat-term-order 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 \equiv DEG LEX

code-datatype LEX DRLEX DEG POT

lemma nat-term-order-LEX [code]: nat-term-order LEX = lex-comp

384
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 (λ x 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 (λ x y. lex-comp y x)) u v = Lt by (rule deg-comp-pos)
next
  fix t u v :: 'a
  have pot-comp (λ x y. lex-comp y x) (splus t u) (splus t v) = Lt
  if pot-comp (λ x 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 (λ x y. lex-comp y x)) u v = Lt
ultimately show deg-comp (pot-comp (λ x 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 \texttt{deg-comp (pot-comp (\lambda x y. \texttt{lex-comp} y x)) \ a \ b = Lt}
ultimately show \texttt{deg-comp (pot-comp (\lambda x y. \texttt{lex-comp} y x)) \ u \ v = Lt}
by (simp add: \texttt{deg-comp pot-comp split: order.splits})
qed
qed
qed

\textbf{lemma nat-term-compare-DEG [code]}: \texttt{nat-term-compare (DEG to) = deg-comp (nat-term-compare to)}
\textbf{unfolding DEG-def}

\textbf{proof} \texttt{(rule comparator-nat-term-compare-Abs-nat-term-order-id)}
from comparator-nat-term-compare \textbf{show comparator (deg-comp (nat-term-compare to))}
by (rule comparator-deg-comp)

next
show \texttt{nat-term-comp (deg-comp (nat-term-compare to))}
proof \texttt{(rule nat-term-compI)}
fix \texttt{u v :: 'a}
assume \texttt{snd (rep-nat-term u) = snd (rep-nat-term v) and fst (rep-nat-term u) = 0}
with comparator-nat-term-compare \textbf{show deg-comp (nat-term-compare to) \ u \ v \neq Gt}
by (rule deg-comp-zero-min)

next
fix \texttt{u v :: 'a}
assume \texttt{a: fst (rep-nat-term u) = fst (rep-nat-term v) and snd (rep-nat-term u) < snd (rep-nat-term v)}
with \texttt{nat-term-comp-nat-term-compare} \textbf{have nat-term-compare to \ u \ v = Lt by (rule nat-term-compD2)}
thus \texttt{deg-comp (nat-term-compare to) \ u \ v = Lt using a by (rule deg-comp-pos)}

next
fix \texttt{t u v :: 'a}
from \texttt{nat-term-comp-nat-term-compare}
\textbf{have nat-term-comp to \ u \ v = Lt \ \Rightarrow \ nat-term-comp to (splus t u) (splus t v) = Lt}
by (rule nat-term-compD3)

moreover \textbf{assume deg-comp (nat-term-compare to) \ u \ v = Lt}
ultimately show \texttt{deg-comp (nat-term-compare to) (splus t u) (splus t v) = Lt}
by (rule deg-comp-monotone)

next
fix \texttt{u v a b :: 'a}
assume \texttt{fst (rep-nat-term u) = fst (rep-nat-term a) and fst (rep-nat-term v) = fst (rep-nat-term b)}
and \texttt{snd (rep-nat-term u) = snd (rep-nat-term v) and snd (rep-nat-term a) = snd (rep-nat-term b)}
moreover \textbf{from comparator-nat-term-compare nat-term-comp-nat-term-compare this}
\textbf{have nat-term-compare to \ u \ v = nat-term-compare to a b}
by (rule nat-term-compD4)
moreover assume \( \text{deg-comp (nat-term-compare to) a b} = \text{Lt} \)
ultimately show \( \text{deg-comp (nat-term-compare to) u v} = \text{Lt} \)
  by (simp add: deg-comp split: order.splits)
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: \( \text{snd (rep-nat-term u)} = \text{snd (rep-nat-term v)} \) and \( \text{fst (rep-nat-term u)} = \emptyset \)
  with nat-term-comp-nat-term-compare have nat-term-compare to u v \( \neq \text{Gt} \) by
    (rule nat-term-compD1)
  thus pot-comp (nat-term-compare to) u v \( \neq \text{Gt} \) 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 = \text{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 = \text{Lt} \( \implies \) nat-term-comp to (splus t u) (splus t v) = \text{Lt}
    by (rule nat-term-compD3)
moreover assume pot-comp (nat-term-compare to) u v = \text{Lt}
ultimately show pot-comp (nat-term-compare to) (splus t u) (splus t v) = \text{Lt}
by (rule pot-comp-monotone)
next
  fix u v a b :: 'a
  assume \( \text{fst (rep-nat-term u)} = \text{fst (rep-nat-term a)} \) and \( \text{fst (rep-nat-term v)} = \text{fst (rep-nat-term b)} \)
  and \( \text{snd (rep-nat-term u)} = \text{snd (rep-nat-term v)} \) and \( \text{snd (rep-nat-term a)} = \text{snd (rep-nat-term b)} \)
  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 = \text{Lt}
ultimately show pot-comp (nat-term-compare to) u v = \text{Lt}
  by (simp add: pot-comp split: order.splits)
qed
lemma nat-term-compare-POT-DRLEX [code]:
  nat-term-compare (POT DRLEX) = pot-comp (λx 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 (λx 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)

qed
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

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
  next
    assume is-scalar TYPE′ a hence snd(rep-nat-term x) = 0 for x ::′a by (simp add: is-scalar)
  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

lemma snd-rep-nat-term-eqI:
  assumes ps ∨ is-scalar TYPE′ a :: nat-term-compare and ps =⇒ snd(rep-nat-term u) =
    snd(rep-nat-term v)
  shows snd(rep-nat-term u) = snd(rep-nat-term v)
  using assms (1)
  proof
    assume is-scalar TYPE′ a hence ?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) =
(\text{pp-of-fun} \ (\lambda x::\text{nat}. \text{if } x = 0 \text{ then } a \text{ else if } x = 1 \text{ then } b \text{ else } 0), \text{ if } (\exists y::'a. \text{ snd} \ (\text{rep-nat-term} \ y) = i) \text{ then } i \text{ else } 0)

\textbf{proof} (\text{cases } (\exists y::'a. \text{ snd} \ (\text{rep-nat-term} \ y) = i))

\text{case } \text{True}

\text{then obtain } y::'a \text{ where } \text{snd} \ (\text{rep-nat-term} \ y) = i \ldots

\text{then obtain } u::'a \text{ where } a = (\text{pp-of-fun} \ (\lambda x::\text{nat}. \text{if } x = 0 \text{ then } a \text{ else if } x = 1 \text{ then } b \text{ else } 0), i)

\text{by} (\text{rule full-componentE})

\text{from } \text{True} \text{ have eq: } (\text{if } (\exists y::'a. \text{ snd} \ (\text{rep-nat-term} \ y) = i) \text{ then } i \text{ else } 0) = i \text{ by simp}

\text{show } \text{?thesis} \text{ unfolding af-exps-def eq}

\text{proof} (\text{rule theI})

\text{fix } v::'a

\text{assume rep-nat-term } v = (\text{pp-of-fun} \ (\lambda x::\text{nat}. \text{if } x = 0 \text{ then } a \text{ else if } x = 1 \text{ then } b \text{ else } 0), i)

\text{thus } v = u \text{ unfolding } u[\text{symmetric}] \text{ by} (\text{rule rep-nat-term-inj})

\text{qed (fact } a) \text{next}

\text{case } \text{False}

\text{hence eq: } (\text{if } (\exists y::'a. \text{ snd} \ (\text{rep-nat-term} \ y) = i) \text{ then } i \text{ else } 0) = 0 \text{ by simp}

\text{obtain } u::'a \text{ where } u = (\text{pp-of-fun} \ (\lambda x::\text{nat}. \text{if } x = 0 \text{ then } a \text{ else if } x = 1 \text{ then } b \text{ else } 0), 0)

\text{by} (\text{rule full-component-zeroE})

\text{show } \text{?thesis} \text{ unfolding af-exps-def eq}

\text{proof} (\text{rule theI})

\text{fix } v::'a

\text{assume rep-nat-term } v = (\text{pp-of-fun} \ (\lambda x::\text{nat}. \text{if } x = 0 \text{ then } a \text{ else if } x = 1 \text{ then } b \text{ else } 0), 0)

\text{thus } v = u \text{ unfolding } u[\text{symmetric}] \text{ by} (\text{rule rep-nat-term-inj})

\text{qed (fact } a) \text{qed}

\textbf{lemma} \text{lookup-pp-of-exps:}

\text{lookup-pp} \ ((\text{fst} \ (\text{rep-nat-term} \ (\text{of-exps } a \ b \ i)))) = (\lambda x. \text{if } x = 0 \text{ then } a \text{ else if } x = 1 \text{ then } b \text{ else } 0)

\text{unfolding rep-nat-term-of-exps fst-conv}

\text{proof} (\text{rule lookup-pp-of-fun})

\text{have } \{x. \text{if } x = 0 \text{ then } a \text{ else if } x = 1 \text{ then } b \text{ else } 0 \} \subseteq \{0, 1\}

\text{by} (\text{rule, simp split: if-split-asmp})

\text{also have finite } ... \text{ by simp}

\text{finally (finite-subset) show finite } \{x. \text{if } x = 0 \text{ then } a \text{ else if } x = 1 \text{ then } b \text{ else } 0 \} \neq \emptyset \}

\text{qed}

\textbf{lemma} \text{keys-pp-of-exps: keys-pp} \ ((\text{fst} \ (\text{rep-nat-term} \ (\text{of-exps } a \ b \ i)))) \subseteq \{0, 1\}

\text{by} (\text{rule, simp add: keys-pp-iff lookup-pp-of-exps split: if-split-asmp})

\textbf{lemma} \text{deg-pp-of-exps [simp]: deg-pp} \ ((\text{fst} \ (\text{rep-nat-term} \ ((\text{of-exps } a \ b \ i)::'a::nat-term-compare)))) = a + b

390
proof
let ?u = (of-exps a b i)::':a
have sum (lookup-pp (fst (rep-term ?u))) (keys-pp (fst (rep-term ?u))) = sum (lookup-pp (fst (rep-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-term ?u))
thus lookup-pp (fst (rep-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-term (x::':a)) = i
shows snd (rep-term ((of-exps a b i)::':a::nat-term-compare)) = i
proof –
from assms have ∃x::':a. snd (rep-term (x::':a)) = i ..
thus ?thesis by (simp add: rep-term-of-exps)
qed

lemma snd-of-exps-zero [simp]: snd (rep-term ((of-exps a b 0)::':a::nat-term-compare)) = 0
proof –
from zero-component obtain x::':a where snd (rep-term (x::':a)) = 0 ..
thus ?thesis by (rule snd-of-exps)
qed

lemma eq-of-exps:
(fst (rep-term (of-exps a1 b1 i))) = fst (rep-term (of-exps a2 b2 j)) ⟷
(a1 = a2 ∧ b1 = b2)
proof –
have a1 = a2 ∧ b1 = b2
  if (λx::nat. if x = 0 then a1 else if x = 1 then b1 else 0) = (λx. 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
qed

lemma lex-pp-of-exps:
lex-pp (fst (rep-term ((of-exps a1 b1 i)::':a))) (fst (rep-term ((of-exps a2 b2 j)::':a::nat-term-compare))) ⟷
(a1 < a2 ∨ (a1 = a2 ∧ b1 ≤ b2)) (is ?L ⟷ ?R)
proof –
let $u = \text{fst (rep-nat-term ((of-exps a1 b1 i)::'a))}$
let $v = \text{fst (rep-nat-term ((of-exps a2 b2 j)::'a))}$

**show** $\mathfrak{thesis}$

**proof**

assume $\mathfrak{L}$

**hence** $u = v \lor (\exists x. \text{lookup-pp } u x < \text{lookup-pp } v x \land (\forall y < x. \text{lookup-pp } u y = \text{lookup-pp } v y))$

by (simp only: lex-pp-alt)

thus $\mathfrak{R}$

**proof**

assume $u = v$

thus $\mathfrak{thesis}$ by (simp add: eq-of-exps)

next

assume $\exists x. \text{lookup-pp } u x < \text{lookup-pp } v x \land (\forall y < x. \text{lookup-pp } u y = \text{lookup-pp } v y)$

then obtain $x$ where

by (simp only: lex-pp-alt)

from $1$ have $\text{lookup-pp } v x \neq 0$ by simp

hence $x \in \text{keys-pp } v$ by (simp add: keys-pp-iff)

also have $\ldots \subseteq \{0, 1\}$ by (fact keys-pp-of-exps)

finally have $x = 0 \lor x = 1$ by simp

thus $\mathfrak{thesis}$

**proof**

assume $x = 0$

from $1$ show $\mathfrak{thesis}$ by (simp add: lookup-pp-of-exps $\langle x = 0 \rangle$)

next

assume $x = 1$

hence $0 < x$ by simp

hence $\text{lookup-pp } u 0 = \text{lookup-pp } v 0$ by (rule 2)

hence $a1 = a2$ by (simp add: lookup-pp-of-exps)

from $1$ show $\mathfrak{thesis}$ by (simp add: lookup-pp-of-exps $\langle x = 1 \rangle \langle a1 = a2 \rangle$)

qed

qed

next

assume $\mathfrak{R}$

thus $\mathfrak{L}$

**proof**

assume $a1 < a2$

show $\mathfrak{thesis}$ unfolding lex-pp-alt

**proof** (intro disjI2 exI conjI allI impI)

from $(a1 < a2)$ show $\text{lookup-pp } u 0 < \text{lookup-pp } v 0$ by (simp add: lookup-pp-of-exps)

next

fix $y :: \text{nat}$

assume $y < 0$

thus $\text{lookup-pp } u y = \text{lookup-pp } v y$ by simp

qed

next
assume $a_1 = a_2 \land b_1 \leq b_2$

hence $a_1 = a_2$ and $b_1 \leq b_2$ by simp-all

from this(2) have $b_1 < b_2 \lor b_1 = b_2$ by auto

thus ?thesis

proof

assume $b_1 < b_2$

show ?thesis unfolding lex-pp-alt

proof (intro disjI2 exI conjI allI impI)

from ($b_1 < b_2$) show lookup-pp $?u 1 <$ lookup-pp $?v 1$ by (simp add: lookup-pp-of-exps)

next

fix $y :: \text{nat}$

assume $y < 1$

hence $y = 0$ by simp

show lookup-pp $?u y =$ lookup-pp $?v y$ by (simp add: lookup-pp-of-exps $?u y = 0 ; a_1 = a_2 ; b_1 = b_2$)

qed

next

assume $b_1 = b_2$

show ?thesis by (simp add: lex-pp-alt eq-of-exps $a_1 = a_2 ; b_1 = b_2$)

qed

qed

next

393
\begin{verbatim}
show ?thesis3
proof (intro iiffI)
  assume a: nat-term-order-eq LEX (DEG to) dg ps
  have dg
    proof (rule ccontr)
      assume \neg 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: \langle \neg dg \rangle)
      thus False
    qed
  show dg \land nat-term-order-eq LEX to dg ps
    proof (intro conjI \langle dg \rangle nat-term-order-eqI)
      fix u v :: 'a
      assume 1: dg \implies deg-pp (fst (rep-nat-term u)) = deg-pp (fst (rep-nat-term v))
      from \langle dg \rangle have eq: deg-pp (fst (rep-nat-term u)) = deg-pp (fst (rep-nat-term v))
        by (rule 1)
      assume ps \implies 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 \land 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 \implies deg-pp (fst (rep-nat-term u)) = deg-pp (fst (rep-nat-term v))
      from \langle dg \rangle have eq: deg-pp (fst (rep-nat-term u)) = deg-pp (fst (rep-nat-term v))
        by (rule 1)
      assume ps \implies 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
\end{verbatim}
next

show \( ?thesis4 \)

proof (intro iffI)
  assume a: nat-term-order-eq LEX (POT to) dg ps
  have *: ps \vee is-scalar TYPE(\'a)
  proof (rule ccontr)
    assume \( \neg (ps \vee is-scalar TYPE(\'a)) \)
    hence \( \neg ps \) and \( \neg is-scalar TYPE(\'a) \) by simp-all
    from this(2) obtain x::\'a where snd (rep-nat-term x) \neq 0 unfolding a
    proof (rule nat-term-order-eqD, fact a, simp-all add: ⟨\neg ps⟩)
  qed

show (ps \vee 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 \( \Rightarrow \) deg-pp (fst (rep-nat-term u)) = deg-pp (fst (rep-nat-term v))
  assume 2: ps \( \Rightarrow \) 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
  proof (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 \vee is-scalar TYPE(\'a)) \& nat-term-order-eq LEX to dg ps

proof (rule nat-term-order-eqI)
  fix u v :: \'a
  assume 1: dg \( \Rightarrow \) deg-pp (fst (rep-nat-term u)) = deg-pp (fst (rep-nat-term v))
  assume 2: ps \( \Rightarrow \) 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
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

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 –
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 ⇒ 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 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 ⇒ 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

396
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 (λx 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 thesis
proof (intro iffI)
assume a: nat-term-order-eq DRLEX (POT to) dg ps
have *: dg ∨ ps ∨ is-scalar TYPE(′a)
proof (rule ccontr)
assume ¬ (dg ∨ ps ∨ is-scalar TYPE(′a))
hence ¬ dg and ¬ ps and ¬ is-scalar TYPE(′a) by simp-all
from this(3) 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 ?a) = 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 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 ?thesis3.
  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 \implies \text{deg-pp (fst (rep-nat-term u)) = deg-pp (fst (rep-nat-term v))}
  and ps \implies \text{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 \implies \text{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 have 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 ?thesis
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 \lor \text{is-scalar TYPE('a)})) \land \text{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 \lor \text{is-scalar TYPE('a)}
  proof (rule ccontr)
    assume \neg (ps \lor \text{is-scalar TYPE('a)})
    hence \neg ps and \neg \text{is-scalar TYPE('a)} by simp-all
  from this(2) 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 (DEG to1) ?u ?v = nat-term-compare (POT to2) ?u ?v
  by (rule nat-term-order-eqD, fact a, simp-all add: \neg 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 \( i \neq 0 \))
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 \( \lor \) is-scalar TYPE('a)) \( \land \) 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]:

\[
\text{nat-term-order-eq (POT to) LEX dg ps = nat-term-order-eq LEX (POT to) dg ps}
\]
\[
\text{nat-term-order-eq (POT to1) (DEG to2) dg ps = nat-term-order-eq (DEG to2)
(POT to1) dg ps}
\]
\[
\text{nat-term-order-eq (POT to1) DRLEX dg ps = nat-term-order-eq DRLEX (POT
to1) dg ps}
\]
\[
\text{nat-term-order-eq (POT to1) (POT (to2::'a::nat-term-compare nat-term-order))
dg ps =}
\]
\[
\text{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 \( \Rightarrow \) deg-pp (fst (rep-nat-term u)) = deg-pp (fst (rep-nat-term v))
and b: True \( \Rightarrow \) 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 \( \Rightarrow \) 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 impl)
assume snd (rep-nat-term u) = snd (rep-nat-term v)

400
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)
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))
      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 (list-of-oalist-TC xs) ∪ fst (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 (list-of-oalist-TC xs) ∪ fst (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 ehm: 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: OList-tc-lookup-inj)

lemma les-PP-oalist [code]:
les (PP-oalist xs) (PP-oalist ys) = PP-oalist (OList-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-les-pp, rule lookup-OList-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 (OList-tc-lookup xs) (set (OList-tc-sorted-domain xs))
    by (simp add: deg-pp-alt keys-PP-oalist)
  also have ... = sum-list (map (OList-tc-lookup xs)) (OList-tc-sorted-domain xs)
    by (rule sum.distinct-set-conv-list, rule distinct-sorted-wrt-irrefl, fact, fact transp-less, fact sorted-OList-tc-sorted-domain)
  also have ... = sum-list (map snd (list-of-oalist-tc xs))
    by (rule arg-cong[where f=sum-list], simp add: OList-tc-sorted-domain-def)
  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 OList-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) = OList-tc-prod-ord (λ-. less-eq) xs ys
for xs ys::('a::linorder, 'b::add-linorder-min) oalist-tc
  fix k
  assume ∀ x. OList-tc-lookup xs x ≤ OList-tc-lookup ys x
  thus OList-tc-lookup xs k ≤ OList-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).
    OList-tc-lookup xs k ≤ OList-tc-lookup ys k
  show OList-tc-lookup xs x ≤ OList-tc-lookup ys x

404
proof (cases x ∈ fst (list-of-oalist-tc xs) ∪ fst (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

405
lemma
\[ \text{lex-comp} \ (\text{sparse}_0 \ [(X, 2::\text{nat}), (Y, 1), (Z, 3)], 3::\text{nat}) \ (\text{sparse}_0 \ [(X, 4)], 2) = \text{Lt} \]
by eval

lemma
\[ \text{lex-pp} \ (\text{sparse}_0 \ [(X, 2::\text{nat}), (Y, 1), (Z, 3)]) \ (\text{sparse}_0 \ [(X, 4)]) \]
by eval

lemma
\[ \text{lex-pp} \ (\text{sparse}_0 \ [(X, 2::\text{nat}), (Y, 1), (Z, 3)]) \ (\text{sparse}_0 \ [(X, 4)]) \]
by eval

lemma
\[ \neg \text{dlex-pp} \ (\text{sparse}_0 \ [(X, 2::\text{nat}), (Y, 1), (Z, 3)]) \ (\text{sparse}_0 \ [(X, 4)]) \]
by eval

lemma
\[ \text{dlex-pp} \ (\text{sparse}_0 \ [(X, 2::\text{nat}), (Y, 1), (Z, 2)]) \ (\text{sparse}_0 \ [(X, 5)]) \]
by eval

lemma
\[ \neg \text{drlex-pp} \ (\text{sparse}_0 \ [(X, 2::\text{nat}), (Y, 1), (Z, 2)]) \ (\text{sparse}_0 \ [(X, 5)]) \]
by eval

end

15.2 \textit{MP-oalist}

lift-definition \textit{MP-oalist} \:: \ ('a::\text{nat-term}, 'b::\text{zero}) oalist-ntm \Rightarrow 'a \Rightarrow 'b

is \textit{OList-lookup-ntm}

proof -

fix \(xs::('a, 'b)\) oalist-ntm

have \(\{x. OList\-lookup\-ntm \ xs \ x \neq 0\} \subseteq \text{fst} \ ' \ \text{set} \ (\text{fst} \ (\text{list-of-oalist-ntm} \ xs))\)

proof (rule, simp)

fix \(x\)

assume \(OList\-lookup\-ntm \ xs \ x \neq 0\)

thus \(x \in \text{fst} \ ' \ \text{set} \ (\text{fst} \ (\text{list-of-oalist-ntm} \ xs))\)

using \oa-ntm.in-sorted-domain-iff-lookup \oa-ntm.set-sorted-domain by blast

qed

also have \textit{finite} ... by simp

finally (\textit{finite-subset}) show \textit{finite} \(\{x. OList\-lookup\-ntm \ xs \ x \neq 0\}\).

qed

lemmas [simp, code] = MP-oalist.rep-eq

code-datatype \textit{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) ⇒ 'b::zero)) = MP-oalist
  (OAlist-empty-ntm nat-term-order-of-le)
  by simp

definition is-zero :: ('a ⇒ '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 (t, c) ∈ set (fst (list-of-oalist-ntm xs)) by (rule imageI)
    hence OAlist-lookup-ntm xs t = c by (simp add: oa-ntm.in-sorted-domain-iff-lookup[simplified oa-ntm.set-sorted-domain])
    moreover 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
  proof (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 (t, c) ∈ 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 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 xs ∪ 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
  λs t.
  le-of-nat-term-order cmp-term
  (term-of-pair (s, the-min))
  (term-of-pair (t, the-min))
  λs 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 ⇒ pair-of-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)) ⇒ oalist-ntm
  => (′a × ′b)
  ⇒
  ′t)
  oalist-ntm
where
  shift-map-keys t f xs = OAlist-ntm
  (map-raw
  (λkv.
  (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
  (λkv.
  (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 = λkv.
  (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))
  ⊆
  (⊔)
  keys (MP-oalist xs)
  (is

409
proof
  let \( f = \lambda kv. (t \oplus \text{fst } kv, c \ast \text{snd } kv) \)
  have \( \mathcal{A} = \text{fst } (\text{map-raw } \mathcal{I} (\text{list-of-oalist-ntm } xs))) \)
    by (simp add: \{keys-MP-oalist \}
also from \{ko-ntm.map-raw-subset \} have ... \subseteq \text{fst } \mathcal{I} (\text{list-of-oalist-ntm } xs))
    by (rule image-mono)
also have ... \subseteq \mathcal{B} by (simp add: \{keys-MP-oalist \}
finally show \{thesis \}.

lemma monom-mult-MP-oalist [code]:
monom-mult c t (MP-oalist xs) = MP-oalist (if c = 0 then \text{OAlist-empty-ntm } \text{snd } (\text{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 \( t \text{ adds}_p u \)
  then obtain v where \( u = t \oplus v \) by (rule adds-ppE)
  thus \( c \ast \text{lookup } (MP-oalist xs) (u \ominus t) = \text{lookup } (MP-oalist (shift-map-keys t ((*) c) xs)) u \)
    by (simp add: \{plus-minus lookup-shift-map-keys-plus del: MP-oalist.rep-eq \}
next
  fix u
  assume \( \neg t \text{ adds}_p u \)
  have \( u \notin \text{keys } (MP-oalist (shift-map-keys t ((*) c) xs)) \)
proof
  assume \( u \in \text{keys } (MP-oalist (shift-map-keys t ((*) c) xs)) \)
also have ... \subseteq ((\oplus) t) \}' \text{keys } (MP-oalist xs) \) by (fact \{keys-shift-map-keys-subset \}
finally obtain v where \( u = t \oplus v \) ..
hence \( t \text{ adds}_p u \) by (rule adds-ppI)
with \( \neg t \text{ adds}_p u \) show \{thesis \} ..
qed
thus \text{lookup } (MP-oalist (shift-map-keys t ((*) c) xs)) u = 0 by (simp add: \{in-keys-iff \}
  qed
thus \{thesis \} by (simp add: False)

lemma mult-scalar-MP-oalist [code]:
\((\text{MP-oalist} \, \text{xs}) \odot (\text{MP-oalist} \, \text{ys}) = \)
\[
\begin{align*}
\text{if} \; \text{is-zero} \; (\text{MP-oalist} \, \text{xs}) \; \text{then} \\
\text{MP-oalist} \; (\text{OAlist-empty-ntm} \; (\text{snd} \; (\text{list-oalist-ntm} \; \text{ys}))) \\
\text{else} \\
\text{let} \; ct = \text{OAlist-hd-ntm} \; \text{xs} \; \text{in} \\
\text{monom-mult} \; (\text{snd} \; ct) \; (\text{fst} \; ct) \; (\text{MP-oalist} \; \text{ys}) + (\text{MP-oalist} \; (\text{OAlist-tl-ntm} \; \text{xs})) \odot (\text{MP-oalist} \; \text{ys})
\end{align*}
\]

\text{proof} (\text{split if-split, intro conjI impI})
\text{assume} \; \text{is-zero} \; (\text{MP-oalist} \; \text{xs})
\text{thus} \; \text{MP-oalist} \; \text{xs} \odot \text{MP-oalist} \; \text{ys} = \text{MP-oalist} \; (\text{OAlist-empty-ntm} \; (\text{snd} \; (\text{list-oalist-ntm} \; \text{ys})))
\text{by} \; (\text{simp add: is-zero-def})

\text{next}
\text{assume} \; \neg \; \text{is-zero} \; (\text{MP-oalist} \; \text{xs})
\text{hence} \; *: \; \text{fst} \; (\text{list-oalist-ntm} \; \text{xs}) \neq [] \; \text{by} \; (\text{simp add: is-zero-MP-oalist \; List.null-def})
\text{define} \; \text{ct} \; \text{where} \; \text{ct} = \text{OAlist-hd-ntm} \; \text{xs}
\text{have} \; \text{eq} \; \text{except} \; (\text{MP-oalist} \; \text{xs}) \{ \text{fst} \; ct \} \oplus (\text{MP-oalist} \; \text{ys}) \oplus (\text{MP-oalist} \; (\text{OAlist-tl-ntm} \; \text{xs})) \\
\text{by} \; (\text{rule poly-mapping-eqI, simp add: lookup-except ct-def oa-ntm.lookup-tl'})
\text{have} \; \text{MP-oalist} \; \text{xs} \odot \text{MP-oalist} \; \text{ys} = \\
\text{monom-mult} \; (\text{lookup} \; (\text{MP-oalist} \; \text{xs}) \; (\text{fst} \; ct)) \; (\text{fst} \; ct) \; (\text{MP-oalist} \; \text{ys}) + (\text{MP-oalist} \; (\text{OAlist-tl-ntm} \; \text{xs})) \odot (\text{MP-oalist} \; \text{ys}) \text{by} \; (\text{fact mult-scalar-rec-left})
\text{also have} \; ... = \text{monom-mult} \; (\text{snd} \; ct) \; (\text{fst} \; ct) \; (\text{MP-oalist} \; \text{ys}) + \text{except} \; (\text{MP-oalist} \; \text{xs}) \{ \text{fst} \; ct \} \odot \text{MP-oalist} \; \text{ys}
\text{using} \; * \; \text{by} \; (\text{simp add: ct-def \; oo-ntm.snd-hd})
\text{also have} \; ... = \text{monom-mult} \; (\text{snd} \; ct) \; (\text{fst} \; ct) \; (\text{MP-oalist} \; \text{ys}) + \text{MP-oalist} \; (\text{OAlist-tl-ntm} \; \text{xs}) \odot \text{MP-oalist} \; \text{ys}
\text{by} \; (\text{simp only: eq})
\text{finally show} \; \text{MP-oalist} \; \text{xs} \odot \text{MP-oalist} \; \text{ys} = \\
(\text{let} \; ct = \text{OAlist-hd-ntm} \; \text{xs} \; \text{in} \\
\text{monom-mult} \; (\text{snd} \; ct) \; (\text{fst} \; ct) \; (\text{MP-oalist} \; \text{ys}) + \text{MP-oalist} \; (\text{OAlist-tl-ntm} \; \text{xs}) \odot \text{MP-oalist} \; \text{ys})
\text{by} \; (\text{simp add: ct-def \; Let-def})
\text{qed}

\text{end}

15.2.1 Special case of addition: adding monomials

\text{definition} \; \text{plus-monomial-less} :: (\text{'a} \rightarrow \text{'b}) \rightarrow \text{'b} \rightarrow \text{'a} \rightarrow (\text{'a} \rightarrow \text{'b};\text{monoid-add})
\text{where} \; \text{plus-monomial-less} \; p \; c \; u = p + \text{monomial} \; c \; u

\text{plus-monomial-less} \; \text{is useful when adding a monomial to a polynomial,}
\text{where the term of the monomial is known to be smaller than all terms in}
\text{the polynomial, because it can be implemented more efficiently than general}
\text{addition.}

\text{lemma} \; \text{plus-monomial-less-MP-oalist} \; [\text{code}]:
\text{plus-monomial-less} \; (\text{MP-oalist} \; \text{xs}) \; c \; u = \text{MP-oalist} \; (\text{OAlist-update-by-fun-gr-ntm} \; u \; (\lambda c0. \; c0 + c) \; \text{xs})
\text{unfolding} \; \text{plus-monomial-less-def \; oo-ntm.update-by-fun-gr-eq-update-by-fun}
plus-monomial-less is computed by \texttt{OAlist-update-by-fun-gr-ntm}, because greater terms come before smaller ones in \texttt{oalist-ntm}.

15.2.2 Constructors

definition \texttt{distr$_0$ ko xs = MP-oalist (oalist-of-list-ntm (xs, ko))} — sparse representation

definition \texttt{V$_0$ :: 'a ⇒ ('a, nat) pp ⇒ ('a, zero) where}
\texttt{V$_0$ n ≡ monomial 1 (single-pp n 1)}

definition \texttt{C$_0$ :: 'b ⇒ ('a, nat) pp ⇒ (0, 'b) where}
\texttt{C$_0$ c ≡ monomial c 0}

lemma \texttt{C$_0$-one: C$_0$ 1 = 1}
\texttt{by (simp add: C$_0$-def)}

lemma \texttt{C$_0$-numeral: C$_0$ (numeral x) = numeral x}
\texttt{by (auto intro!: poly-mapping-eqI simp: C$_0$-def lookup-numeral)}

lemma \texttt{C$_0$-minus: C$_0$ (- x) = - C$_0$ x}
\texttt{by (simp add: C$_0$-def single-uminus)}

lemma \texttt{C$_0$-zero: C$_0$ 0 = 0}
\texttt{by (auto intro!: poly-mapping-eqI simp: C$_0$-def)}

lemma \texttt{V$_0$-power: V$_0$ v ^ n = monomial 1 (single-pp v n)}
\texttt{by (induction n) (auto simp: V$_0$-def mult-single single-pp-plus)}

lemma \texttt{single-MP-oalist [code]: Poly-Mapping.single k v = distr$_0$ nat-term-order-of-le [(k, v)]}
\texttt{unfolding distr$_0$-def by (rule poly-mapping-eqI, simp add: lookup-single OAlist-lookup-ntm-single)}

lemma \texttt{one-MP-oalist [code]: 1 = distr$_0$ nat-term-order-of-le [(0, 1)]}
\texttt{by (metis single-MP-oalist single-one)}

lemma \texttt{except-MP-oalist [code]: except (MP-oalist xs) S = MP-oalist (OAlist-filter-ntm (λkv. fst kv \notin S) xs)}
\texttt{by (rule poly-mapping-eqI, simp add: lookup-except oa-ntm lookup-filter)}

15.2.3 Changing the Internal Order

definition \texttt{change-ord :: 'a::nat-term-compare nat-term-order ⇒ ('a ⇒ 'b) ⇒ ('a ⇒_0 'b)}
\texttt{where change-ord to = (λx. x)}

lemma \texttt{change-ord-MP-oalist [code]: change-ord to (MP-oalist xs) = MP-oalist (OAlist-reorder-ntm to xs)}
\texttt{by (rule poly-mapping-eqI, simp add: change-ord-def oa-ntm lookup-reorder)}
15.2.4 Ordered Power-Products

lemma foldl-assoc:
assumes \( \forall x y z. f (f x y) z = f x (f y z) \)
shows \( \text{foldl } f (f a b) \text{ } \text{xs} = f a (\text{foldl } f b \text{ } \text{xs}) \)
proof (induct \text{xs} arbitrary: a b)
  next
  fix a b x xs
  assume \( \forall a b. \text{foldl } f (f a b) \text{ } \text{xs} = f a (\text{foldl } f b \text{ } \text{xs}) \)
  from \text{assms}[\{a b x\}]
  this[\{of a f b x\}]
  show \text{foldl } f (f a b) (x \# \text{xs}) = f a (\text{foldl } f b (x \# \text{xs})) \) unfolding \text{foldl-Cons}
  by \text{simp}
qed
class gd-nat-term

begin

definition ord-pp :: \('a \Rightarrow \text{bool}'\)
where \( \text{ord-pp } s t = \text{le-of-nat-term-order } \text{cmp-term} (\text{term-of-pair } (s, \text{the-min})) \)
\( \text{(term-of-pair } (t, \text{the-min})) \)
definition ord-pp-strict :: \('a \Rightarrow \text{bool}'\)
where \( \text{ord-pp-strict } s t = \text{lt-of-nat-term-order } \text{cmp-term} (\text{term-of-pair } (s, \text{the-min})) \)
\( \text{(term-of-pair } (t, \text{the-min})) \)

lemma lt-MP-oalist [code]:
\( \text{lt } (\text{MP-oalist } \text{xs}) = (\text{if is-zero } (\text{MP-oalist } \text{xs}) \text{ then min-term else } \text{fst } (\text{OAlist-min-key-val-ntm cmp-term } \text{xs})) \)
proof (split if-split, intro conjI impI)
  assume \text{is-zero } (\text{MP-oalist } \text{xs})
  thus \text{lt } (\text{MP-oalist } \text{xs}) = \text{min-term} \) by (simp add: \text{is-zero-def})
next
  assume \text{is-zero } (\text{MP-oalist } \text{xs})
  hence \text{fst } (\text{list-of-oalist-ntm } \text{xs}) \# \text{[]} \) by (simp add: \text{is-zero-MP-oalist List.null-def})
  show \text{lt } (\text{MP-oalist } \text{xs}) = \text{fist } (\text{OAlist-min-key-val-ntm cmp-term } \text{xs})
  proof (rule \text{lt-eqI-keys})
    show \text{fist } (\text{OAlist-min-key-val-ntm cmp-term } \text{xs}) \in \text{keys } (\text{MP-oalist } \text{xs})
    by (simp add: \text{keys-MP-oalist}, rule \text{imageI}, rule \text{oa-ntm.min-key-val-in}, \text{fact})
  next
    fix u
    assume \text{u } \in \text{keys } (\text{MP-oalist } \text{xs})
    also have \text{... } = \text{fist set } (\text{fist } (\text{list-of-oalist-ntm } \text{xs})) \) by (simp add: \text{keys-MP-oalist})
    finally obtain z where \text{z } \in \text{set } (\text{fist } (\text{list-of-oalist-ntm } \text{xs})) \text{ and } u = \text{fist } z \text{.}
    from \text{this(1)} \text{have } \text{ko.le } (\text{key-order-of-nat-term-order-inv cmp-term}) \text{ (fist } (\text{OAlist-min-key-val-ntm cmp-term } \text{xs})) \text{ u }
    unfolding (u = \text{fist } z) \) by (rule \text{oa-ntm.min-key-val-minimal})
    thus \text{le-of-nat-term-order cmp-term } u \text{ (fist } (\text{OAlist-min-key-val-ntm cmp-term } \text{xs}))
  \)
by (simp add: le-of-nat-term-order-alt)

qed

qed

lemma le-MP-oalist [code]:
lc (MP-oalist xs) = (if is-zero (MP-oalist xs) then 0 else snd (OList-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 (OList-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 (OList-except-min-ntm cmp-term xs)

proof (cases is-zero (MP-oalist xs))
  case True
tence fst (list-of-oalist-ntm xs) = [] by (simp add: is-zero-MP-oalist List.null-def)
  hence fst (list-of-oalist-ntm (OList-except-min-ntm cmp-term xs)) = []
    by (rule oa-ntm.except-min-Nil)
  hence is-zero (MP-oalist (OList-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
tshow ?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 ⇒ 'c::zero, 't ⇒ '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) =
    OList-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 if = 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 if 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 (fst (list-of-oalist-ntm xs)) ∪ fst (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 OAlst-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 neg 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 OAlst-lookup-ntm xs u = OAlst-lookup-ntm ys u then Some Eq
          else if OAlst-lookup-ntm xs u = 0 then Some Lt
          else if OAlst-lookup-ntm ys u = 0 then Some Gt else None) = Some Eq
  by simp
  qed fact
  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 antisym ord-term-strict-conv by (rule antisymI, 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)
  qed
next
show set (keys-to-list (MP-oalist xs)) = set (OList-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,welorder})
  assumes term-powerprod pair-of-term term-of-pair
    and Ꞅ v. fst (rep-nat-term v) = rep-nat-pp (fst (pair-of-term v))
    and Ꞅ t. snd (rep-nat-term (term-of-pair (t, the-min))) = 0
    and Ꞅ v w. snd (pair-of-term v) ≤ snd (pair-of-term w) ⇒ snd (rep-nat-term v) ≤ snd (rep-nat-term w)
    and Ꞅ s t k. term-of-pair (s + t, k) = splus (term-of-pair (s, k)) (term-of-pair (t, k))
    and Ꞅ 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
    (λs t. le-of-nat-term-order cmp-term (term-of-pair (s, the-min)) (term-of-pair (t, the-min)))
    (λs 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› = λx. 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 (λs t. le-of-nat-term-order cmp-term (‹f› s) (‹f› t))
          (λs t. lt-of-nat-term-order cmp-term (‹f› s) (‹f› t))
        proof (intro ordered-powerprod.intro ordered-powerprod-axioms.intro)
          show class.linorder (λs t. le-of-nat-term-order cmp-term (‹f› s) (‹f› t))
            (λs t. lt-of-nat-term-order cmp-term (‹f› s) (‹f› t))
            fix x y
            assume ko.le (key-order-of-nat-term-order-inv cmp-term) (term-of-pair (x, the-min))
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) 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

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 (\lambda s 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 \oplus v) (t \oplus w) by (simp add: le-of-nat-term-order assms(6) nat-term-compare-splus)

next
fix v w
assume tp.component-of-term v \leq tp.component-of-term w
hence 4: snd (rep-nat-term v) \leq 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)))
\[(t, w)\] by (simp add: assms(2) tp.pp-of-term-def tp.pair-term)

moreover note 4

moreover have snd (rep-nat-term (?t (tp.pp-of-term v))) = snd (rep-nat-term (?t (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})
⇒ \(\lambda s t. \) le-of-nat-term-order cmp-term (fst (s, the-min)) (fst (t, the-min)))
(\(\lambda s t. \) lt-of-nat-term-order cmp-term (fst (s, the-min)) (fst (t, the-min)))
(le-of-nat-term-order cmp-term)
(lt-of-nat-term-order cmp-term)

proof (intro gd-term.intro ordered-term.intro)

show term-powerprod to-pair-unit fst by unfold-locales

next

show ordered-powerprod (\(\lambda s t. \) le-of-nat-term-order cmp-term (fst (s, the-min))
(fst (t, the-min)))
(\(\lambda s t. \) 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 (\(\lambda s t. \) le-of-nat-term-order cmp-term
(fst (s, the-min)) (fst (t, the-min)))

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

lemma gd-term-id:
gd-term (λx::(nat::nat-term-compare, nat-pp-compare, nat-pp-term, graded-dickson-powerprod) × 'b::{nat, the-min}). x) (λx. x)
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

15.4 Computations

type-synonym ′a mpoly-tc = (nat, nat) pp ⇒ ′a

global-interpretation punit0: gd-nat-term to-pair-unit::′a::{nat-term-compare, nat-pp-term, graded-dickson-powerprod}
⇒ - fst cmp-term
rewrites punit.adds-term = (adds)
and punit.pp-of-term = (λx. x)
and punit.component-of-term = (λx. ())
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 le-punit = punit0.le
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

420
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

  by (intro ext, simp add: punit0.ord-pp-def)

  by (intro ext, simp add: punit0.ord-pp-strict-def)

  unfolding punit0.ord-pp-def punit0.ord-pp-strict-def ..

locale trivariate0-rat
begin

abbreviation X::rat mpoly-tc where X ≡ V0 (0::nat)
abbreviation Y::rat mpoly-tc where Y ≡ V0 (1::nat)
abbreviation Z::rat mpoly-tc where Z ≡ V0 (2::nat)

end

experiment begin interpretation trivariate0-rat .

value [code] X ^ 2

value [code] X ^ 2 * Z + 2 * Y ^ 3 * Z ^ 2

value [code] distr0 DRLEX [(sparse0 [(0::nat, 3::nat)], 1::rat)] = distr0 DRLEX [(sparse0 [(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

421
by eval

value [code] min-term-punit::(nat, nat) pp

value [code] is-zero (distr0 DRLEX [(sparse0 [(0::nat, 3::nat)], 1::rat)])

value [code] lt-punit DRLEX (distr0 DRLEX [(sparse0 [(0::nat, 3::nat)], 1::rat)])

lemma
  lt-punit DRLEX (X^2 * Z + 2 * Y * 3 * Z^2) = sparse0 [(1, 3), (2, 2)]
  by eval

lemma
  lt-punit DRLEX (X + Y + Z) = sparse0 [(2, 1)]
  by eval

lemma
  keys (X^2 * Z * 3 + 2 * Y * 3 * Z^2) =
  { sparse0 [(0, 2), (2, 3)], sparse0 [(1, 3), (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 * 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
  lookup (X^2 * Z * 7 + 2 * Y * 3 * Z^2 + 2) (sparse0 [(0, 2), (2, 7)]) = 1
  by eval

lemma
  X^2 * Z * 7 + 2 * Y * 3 * Z^2 ≠
  X^2 * Z * 4 + -2 * Y * 3 * Z^2
  by eval

lemma
  0 * X^2 * Z^7 + 0 * Y^3 * Z^2 = 0
  by eval

lemma
monom-mult-punit 3 (sparse0 [(1, 2::nat)]) ((X^2 + Z + 2 * Y - 3 * Z^2) =
3 * Y^2 + Z * X^2 + 6 * Y - 5 * Z^2)
by eval

lemma
monomial (-4) (sparse0 [(0, 2::nat)]) = -4 * X^2
by eval

lemma monomial (0::rat) (sparse0 [(0::nat, 2::nat)]) = 0
by eval

lemma
(X^2 + Z + 2 * Y - 3 * Z^2) * (X^2 + Z - 3 + -2 * Y - 3 * Z^2) =
X - 4 * Z - 4 + -2 * X^2 + Z - 3 * Y - 3 +
-4 * Y - 6 * Z - 4 + 2 * Y - 3 * Z - 5 * X^2
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 =⇒ 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 =⇒ d (decr-grading d n t) ≤ n
  using decr-grading unfolding hom-grading-fun-def by blast

lemma decr-grading-idL:
  hom-grading d =⇒ d t ≤ n =⇒ 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

424
lemma hom-grading-varnum:
  hom-grading ((varnum X)::('x::countable ⇒0 'b::add-wellorder) ⇒ nat)
proof
  define f where f = (λn t. (except t (− (X ∪ {x. elem-index x < n}))))::'x ⇒0 'b
show ?thesis unfolding hom-grading-def hom-grading-fun-def
proof (intro exI allI conjI impI)
fix n s t
  show fn s t = fn s t by (simp only: f-def except-plus)
next
fix n t
  show varnum X (fn t) ≤ n by (auto simp: varnum-le-iff keys-except f-def)
next
fix n t
  show varnum X t ≤ n ⇒ fn 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 ⇒ nat) ⇒ nat ⇒ 't ⇒ 't
  where decr-grading-term d n v = term-of-pair (decr-grading d n (pp-of-term v),
component-of-term v)

definition decr-grading-p :: ('a ⇒ nat) ⇒ nat ⇒ ('t ⇒0 'b) ⇒ ('t ⇒0 'b::comm-monoid-add)
  where decr-grading-p d n p = (∑v∈keys p. monomial (lookup p v) (decr-grading-term d n v))

lemma decr-grading-term-splus:
  hom-grading d ⇒ decr-grading-term d n (t ⊕ v) = decr-grading d n t ⊕
decr-grading-term d n v
  by (simp add: decr-grading-term-def term-simps decr-grading-plus splus-def)

lemma decr-grading-term-le: hom-grading d ⇒ d (pp-of-term (decr-grading-term d n v)) ≤ n
  by (simp add: decr-grading-term-def term-simps decr-grading-le)

lemma decr-grading-term-idI: hom-grading d ⇒ d (pp-of-term v) ≤ n ⇒ 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: (∑ v∈keys p ∪ keys q. monomial (lookup p v) (decr-grading-term d n v)) =
  (∑ v∈keys p. monomial (lookup p 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 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))
proof (rule sum.mono-neutral-left)
  show ∀ v∈keys p ∪ keys q − keys p. monomial (lookup (p + q) v) (decr-grading-term d n v) = 0
    by (simp add: in-keys-iff)
qed

also have ...
proof (rule sum.mono-neutral-left)
  show ∀ v∈keys p ∪ keys q − keys p. monomial (lookup p v) (decr-grading-term d n v) = 0
    by (simp only: lookup-add single-add sum.distrib)
also have ...
proof (rule sum.mono-neutral-left)
  show ∀ v∈keys p ∪ keys q − keys p. monomial (lookup q v) (decr-grading-term d n v) = 0
    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 ⊙ q) = punit. decr-grading-p d n p ⊙ 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)
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-term)
qed

lemma decr-grading-p-keys-subset: keys (decr-grading-p d n p) ⊆ decr-grading-term d n ' keys p
proof
fix v
assume v ∈ keys (decr-grading-p d n p)
also have ... ⊆ (∪ u∈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 ∈ keys p and v ∈ 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 ∈ decr-grading-term d n ' keys p unfolding eq using ⟨ u ∈ keys p ⟩ by (rule imageI)
qed
lemma *decr-grading-p-idI*:
  assumes hom-grading \(d\) and \(\forall v. v \in \text{keys } p \implies d \text{(pp-of-term } v) \leq n\)
  shows \(\text{decr-grading-p } d \ n \ p = p\)
proof –
  have \(\text{decr-grading-p } d \ n \ p = (\sum v \in \text{keys } p. \text{monomial (lookup } p \ v) \ v)\) unfolding \(\text{decr-grading-p-def}\)
  using refl
proof (rule sum.cong)
  fix \(v\)
  assume \(v \in \text{keys } p\)
  hence \(d \text{(pp-of-term } v) \leq m\) by (rule assms(2))
  with assms(1) have \(\text{decr-grading-term } d \ n \ v = v\) by (rule \(\text{decr-grading-term-idI}\))
  thus \(\text{monomial (lookup } p \ v) \text{(decr-grading-term } d \ n \ v) = \text{monomial (lookup } p \ v) \ v\) by simp
  qed
  also have ... = \(p\) by (fact \(\text{poly-mapping-sum-monomials}\))
  finally show \(?thesis\)
  qed

end

context gd-term
begin

lemma *decr-grading-p-idI*:
  assumes hom-grading \(d\) shows \(\text{decr-grading-p } d \ m \ p \in \text{dgrad-p-set } d \ m\)
proof (rule dgrad-p-setI)
  fix \(v\)
  assume \(v \in \text{keys } (\text{decr-grading-p } d \ m \ p)\)
  hence \(v \in \text{decr-grading-term } d \ m \ ' \text{keys } p\) using \(\text{decr-grading-p-keys-subset}\) ..
  then obtain \(u\) where \(v = \text{decr-grading-term } d \ m \ u\) ..
  with assms show \(d \text{(pp-of-term } v) \leq m\) by (simp add: \(\text{decr-grading-term-le}\))
  qed

lemma (in gd-term) *in-pmdlE-dgrad-p-set*:
  assumes hom-grading \(d\) and \(B \subseteq \text{dgrad-p-set } d \ m\) and \(p \in \text{dgrad-p-set } d \ m\) and \(p \in \text{pmdl } B\)
  obtains \(A \ q\) where finite \(A\) and \(A \subseteq B\) and \(\forall b. q \ b \in \text{punit.dgrad-p-set } d \ m\)

428
and \( p = \left( \sum_{b \in A} q \ b \odot b \right) \)

proof —
from \text{assms}(4) obtain \( A \) \( q \) \( A \subseteq B \) and \( p = (\sum_{b \in A} q \ b \odot b) \)
by (rule pmdl.spanE)
define \( q \) where \( q = (\lambda b. \ punit. \ \text{decr-grading-p} \ d \ m \ (q \ b)) \)
from \( (\text{finite } A: \ A \subseteq B) \) show \( \text{thesis} \)
proof
fix \( b \)
show \( q \ b \in \ punit. \ \text{dgrad-p-set} \ d \ m \)
unfolding \( q \)-def using \text{assms}(1) by (rule \( \text{punit} \). \( \text{decr-grading-p-setI} \))
next
from \text{assms}(1, 3) have \( p = \text{decr-grading-p} \ d \ m \ p \) by (simp only: \( \text{decr-grading-p-idI} \))
also from \text{assms}(1) have \( \ldots = (\sum_{b \in A} q \ b \odot (\text{decr-grading-p} \ d \ m \ b)) \)
by (simp add: \( p \)-def \( \text{decr-grading-p-sum} \) \( \text{decr-grading-p-mult-scalar} \))
also from \( \text{refl} \) have \( \ldots = (\sum_{b \in A} q \ b \odot b) \)
proof (rule \( \text{sum.cong} \))
fix \( b \)
assume \( b \in A \)
hence \( b \in B \) using \( (A \subseteq B) \) ..
hence \( b \in \text{dgrad-p-set} \ d \ m \) using \text{assms}(2) ..
with \text{assms}(1) have \( \text{decr-grading-p} \ d \ m \ b = b \) by (rule \( \text{decr-grading-p-idI} \))
thus \( q \ b \odot \text{decr-grading-p} \ d \ m \ b = q \ b \odot b \) 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 \( \langle \ x \Rightarrow a \rangle \Rightarrow b \) closely resemble those introduced in Polynomials.MPoly-Type for type \( a \) \( \text{mpoly} \).

lemma monomial-single-power:
\( (\text{monomial } c \ (\text{Poly-Mapping.single} x \ k)) \ ^{\ * n} = \text{monomial } (c ^{\ * n}) \ (\text{Poly-Mapping.single} x \ (k * n)) \)
proof —
have \( eq: (\sum_{i = 0..<n} \text{Poly-Mapping.single} x \ k) = \text{Poly-Mapping.single} x \ (k * n) \)
by (induct \( n \), simp-all add: \( \text{add.commute single-add} \))
show \(?thesis\) by \((simp \ add: \ punit.monomial-power \ eq)\)

qed

lemma monomial-power-map-scale: \((\text{monomial } c \ t) ^ n = \text{monomial } (c ^ n) \ (n \cdot t)\)

proof -
  have \((\sum i = 0..<n. \ t) = (\sum i = 0..<n. \ 1) \cdot t\)
  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 \(h * p = h * q \ \& \ h \neq (0::('x::linorder \Rightarrow 0 \ \Rightarrow 0 \ a::\text{ring-no-zero-divisors})\)
  shows \(p = q\)

proof \((rule \ ccontr)\)
  assume \(p \neq q\)
  hence \(p - q \neq 0\) by \(simp\)
  with \(assms(2)\) have \(h * (p - q) \neq 0\) by \(simp\)
  hence \(h * p \neq h * q\) by \(simp \ add: \ algebra-simps\)
  thus \(False\) using \(assms(1)\) ..

qed

lemma times-canc-right:
  assumes \(p * h = q * h \ \& \ h \neq (0::('x::linorder \Rightarrow 0 \ \Rightarrow 0 \ a::\text{ring-no-zero-divisors})\)
  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:
  \(s + (t - u) = (s + (\text{except } (u - t) (- \ keys s))) + t - (u::- \Rightarrow 0 \ nat)\)

proof \((rule \ poly-mapping-eqI)\)
  fix \(x\)
  show \(\text{lookup } (s + (t - u)) \ x = \text{lookup } (s + \text{except } (u - t) (- \ keys s) + t - u) \ x\)
  proof \((cases \ x \in \ keys s)\)
    case True
    thus \(?thesis\)
    by \(\text{(simp add: plus-minus-assoc-pm-nat-1 \ lookup-add \ lookup-minus \ lookup-except)}\)
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

lemma deg-pm-sum: deg-pm (sum t A) = (∑ a∈A. deg-pm (t a))
by (induct A rule: infinite-finite-induct) (auto simp: deg-pm-plus)

lemma deg-pm-mono: s adds t ==> deg-pm s ≤ deg-pm (t::="add-linorder-min")
by (metis addsE deg-pm-plus le_iff_add)

lemma deg-pm-mono: s adds t ==> deg-pm s ≤ deg-pm (t::="add-linorder-min")
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::="add-linorder-min")
  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::="add-linorder-min")
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::="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: \( t - s = t \iff \text{keys } t \cap \text{keys } (s::\Rightarrow 0 \text{ nat}) = \{\} \)
proof
assume \( t - s = t \)
{ 
  fix \( x \)
  assume \( x \in \text{keys } t \) and \( x \in \text{keys } s \)
  hence \( 0 < \text{lookup } t x \) and \( 0 < \text{lookup } s x \) by (simp-all add: in-keys-iff)
  hence \( \text{lookup } (t - s) x \neq \text{lookup } t x \) by (simp add: lookup-minus)
  with \( t - s = t \); have False by simp
\}
thus \( \text{keys } t \cap \text{keys } s = \{\} \) by blast
next
assume \( *: \text{keys } t \cap \text{keys } s = \{\} \)
show \( t - s = t \)
proof (rule poly-mapping-eqI)
  fix \( x \)
  have \( \text{lookup } t x - \text{lookup } s x = \text{lookup } t x \)
  proof (cases \( x \in \text{keys } t \))
    case True
    with \( * \) have \( x \notin \text{keys } s \) by blast
    thus \( \text{thesis} \) by (simp add: in-keys-iff)
  next
    case False
    thus \( \text{thesis} \) by (simp add: in-keys-iff)
  qed
  thus \( \text{lookup } (t - s) x = \text{lookup } t x \) by (simp only: lookup-minus)
  qed
qed

lemma deg-pm-minus-id-iff: \( \text{deg-pm } (t - s) = \text{deg-pm } t \iff \text{keys } t \cap \text{keys } (s::\Rightarrow 0 \text{ nat}) = \{\} \)
proof
assume eq: \( \text{deg-pm } (t - s) = \text{deg-pm } t \)
{ 
  fix \( x \)
  assume \( x \in \text{keys } t \) and \( x \in \text{keys } s \)
  hence \( 0 < \text{lookup } t x \) and \( 0 < \text{lookup } s x \) by (simp-all add: in-keys-iff)
  hence \( *: \text{lookup } (t - s) x < \text{lookup } t x \) by (simp add: lookup-minus)
  have \( \text{keys } (t - s) \subseteq \text{keys } t \) by (rule, simp add: lookup-minus in-keys-iff)
  hence \( \text{deg-pm } (t - s) = (\sum x \in \text{keys } t. \text{lookup } (t - s) x) \) using finite-keys by (rule deg-pm-superset)
  also from finite-keys have \( \ldots < (\sum x \in \text{keys } t. \text{lookup } t x) \)
  proof (rule sum-strict-mono-ex1)
    show \( \forall x \in \text{keys } t. \text{lookup } (t - s) x \leq \text{lookup } t x \) by (simp add: lookup-minus)
  next
    from \( x \in \text{keys } t \); show \( \exists x \in \text{keys } t. \text{lookup } (t - s) x < \text{lookup } t x \)
    qed
  also have \( \ldots = \text{deg-pm } t \) by (rule sym, rule deg-pm-superset, fact subset-refl, fact finite-keys)
}\)
432
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 :: 
add-linorder ⇒ zero ⇒ a where
poly-deg p = (if keys p = {} then 0 else Max (deg-pm ' keys p))

definition maxdeg :: 
add-linorder ⇒ zero ⇒ a where
maxdeg A = Max (poly-deg ' A)

definition mindeg :: 
add-linorder ⇒ zero ⇒ 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 ≠ 0
obtains t where t ∈ 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 ∈ keys p and poly-deg p = deg-pm t ..
thus ?thesis ..
qed

lemma poly-deg-max-keys: t ∈ keys p ⇒ deg-pm t ≤ poly-deg p
using finite-keys by (auto simp: poly-deg-def)

lemma poly-deg-leI: (∀t. t ∈ 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:
  \( p \neq 0 \implies (\forall t \in \text{keys } p \implies \text{deg-pm } t < (d::'a::add-linorder-min)) \implies \text{poly-deg } p < d \)
  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 \neq 0 by simp
      hence 0 < deg-pm t using not-gr-zero by blast
      also from t \in\ keys p have ... \le\ poly-deg p by (rule poly-deg-max-keys)
      finally have poly-deg p \neq 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) \le\ max (poly-deg p) (poly-deg (q::(- \Rightarrow_0 'a::add-linorder-min)) \Rightarrow_0 -))
  proof (rule poly-deg-leI)
    fix t
    assume t \in\ keys (p + q)
    also have ... \subseteq\ keys p \cup\ keys q by (fact Poly-Mapping.keys-add)
    finally show deg-pm t \le\ max (poly-deg p) (poly-deg q)
    proof
      assume t \in\ keys p
      hence deg-pm t \le\ poly-deg p by (rule poly-deg-max-keys)
      thus \thesis by (simp add: le-max-iff-disj)
    next
      assume t \in\ keys q
      hence deg-pm t \le\ 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 (\neg p) = poly-deg p
  by (simp add: poly-deg-def keys-uminus)

lemma poly-deg-minus-le:
  poly-deg (p - q) \le\ max (poly-deg p) (poly-deg (q::(- \Rightarrow_0 'a::add-linorder-min))}
proof (rule poly-deg-leI)

fix $t$
assume $t \in \text{keys} (p - q)$
also have $\cdots \subseteq \text{keys} p \cup \text{keys} q$ by (fact keys-minus)
finally show $\deg-pm t \leq \max (\text{poly-deg} p) (\text{poly-deg} q)$
proof
assume $t \in \text{keys} p$
hence $\deg-pm t \leq \text{poly-deg} p$ by (rule poly-deg-max-keys)
thus $\vdash \text{thesis}$ by (simp add: le-max-iff-disj)
next
assume $t \in \text{keys} q$
hence $\deg-pm t \leq \text{poly-deg} q$ by (rule poly-deg-max-keys)
thus $\vdash \text{thesis}$ by (simp add: le-max-iff-disj)
qed

qed

lemma poly-deg-times-le:
$\text{poly-deg} (p \ast q) \leq \text{poly-deg} p + \text{poly-deg} q : (\Rightarrow \Rightarrow \text{add-linorder-min} \Rightarrow_{0} ^{\Rightarrow})$
proof (rule poly-deg-leI)

fix $t$
assume $t \in \text{keys} (p \ast q)$
then obtain $u v$ where $u \in \text{keys} p$ and $v \in \text{keys} q$ and $t = u + v$ by (rule in-keys-timesE)
from $\langle u \in \text{keys} p \rangle$ have $\deg-pm u \leq \text{poly-deg} p$ by (rule poly-deg-max-keys)
moreover from $\langle v \in \text{keys} q \rangle$ have $\deg-pm v \leq \text{poly-deg} q$ by (rule poly-deg-max-keys)
ultimately show $\deg-pm t \leq \text{poly-deg} p + \text{poly-deg} q$ by (simp add: $\langle t = u + v \rangle$ deg-pm-plus add-mono)
qed

lemma poly-deg-times:
assumes $p \neq 0$ and $q \neq (0 : (\Rightarrow \Rightarrow \text{add-linorder} \Rightarrow_{0} \Rightarrow \text{add-linorder-min} \Rightarrow_{0} \Rightarrow \text{semiring-no-zero-divisors})$
shows $\text{poly-deg} (p \ast q) = \text{poly-deg} p + \text{poly-deg} q$
using poly-deg-times-le
proof (rule antisym)
let $\forall A = \lambda f : \{ u. \deg-pm u < \text{poly-deg} f \}$
define $p1$ where $p1 = \text{except} p (\forall A p)$
define $p2$ where $p2 = \text{except} p (\neg \forall A p)$
define $q1$ where $q1 = \text{except} q (\forall A q)$
define $q2$ where $q2 = \text{except} q (\neg \forall A q)$
have $\deg-p1 : \deg-pm t = \text{poly-deg} p$ if $t \in \text{keys} p1$ for $t$
proof
from that have $t \in \text{keys} p$ and $\text{poly-deg} p \leq \deg-pm t$
by (simp-all add: p1-def keys-except not-less)
from this $\langle t \rangle$ have $\deg-pm t \leq \text{poly-deg} p$ by (rule poly-deg-max-keys)
thus $\text{thesis}$ using $\langle \text{poly-deg} p \leq \deg-pm t \rangle$ by (rule antisym)
qed

have $\deg-p2 : t \in \text{keys} p2 \Rightarrow \deg-pm t < \text{poly-deg} p$ for $t$ by (simp add: p2-def keys-except)
have \( \text{deg-q1}: \text{deg-pm } t = \text{poly-deg } q \) if \( t \in \text{keys } q1 \) for \( t \)

proof −
from that have \( t \in \text{keys } q \) and \( \text{poly-deg } q \leq \text{deg-pm } t \) by \((\text{simp-all add: q1-def keys-except not-less})\)
from this\((t)\) have \( \text{deg-pm } t \leq \text{poly-deg } q \) by \((\text{rule poly-deg-max-keys})\)
thus \( ?\)thesis using \( \langle \text{poly-deg } q \leq \text{deg-pm } t \rangle \) by \((\text{rule antisym})\)

qed

have \( \text{deg-q2}: t \in \text{keys } q2 \implies \text{deg-pm } t < \text{poly-deg } q \) for \( t \) by \((\text{simp add: q2-def keys-except})\)

have \( p; p = p1 + p2 \) unfolding \( p1-def \) \( p2-def \) by \( (\text{fact except-decomp})\)

have \( p1 \neq 0 \)

proof −
from assms\((1)\) obtain \( t \) where \( t \in \text{keys } p \) and \( \text{poly-deg } p = \text{deg-pm } t \) by \((\text{rule poly-degE})\)

hence \( t \in \text{keys } p1 \) by \((\text{simp add: p1-def keys-except})\)

thus \( ?\)thesis by auto

qed

have \( q; q = q1 + q2 \) unfolding \( q1-def \) \( q2-def \) by \( (\text{fact except-decomp})\)

have \( q1 \neq 0 \)

proof −
from assms\((2)\) obtain \( t \) where \( t \in \text{keys } q \) and \( \text{poly-deg } q = \text{deg-pm } t \) by \((\text{rule poly-degE})\)

hence \( t \in \text{keys } q1 \) by \((\text{simp add: q1-def keys-except})\)

thus \( ?\)thesis by auto

qed

with \( \langle p1 \neq 0 \rangle \) have \( p1 \ast q1 \neq 0 \) by simp

hence \( \text{keys } (p1 \ast q1) \neq \{\} \) by simp

then obtain \( u \) where \( u \in \text{keys } (p1 \ast q1) \) by blast

then obtain \( s \) \( t \) where \( s \in \text{keys } p1 \) and \( t \in \text{keys } q1 \) and \( u; u = s + t \) by \((\text{rule in-keys-timesE})\)

from \( (s \in \text{keys } p1) \) have \( \text{deg-pm } s = \text{poly-deg } p \) by \((\text{rule deg-p1})\)

moreover from \( (t \in \text{keys } q1) \) have \( \text{deg-pm } t = \text{poly-deg } q \) by \((\text{rule deg-q1})\)

ultimately have \( \text{eq: poly-deg } p + \text{poly-deg } q = \text{deg-pm } u \) by \((\text{simp only: u deg-pm-plus})\)

also have \( \ldots \leq \text{poly-deg } (p \ast q) \)

proof \((\text{rule poly-deg-max-keys})\)

have \( u \notin \text{keys } (p1 \ast q2 + p2 \ast q) \)

proof
assume \( u \in \text{keys } (p1 \ast q2 + p2 \ast q) \)

also have \( \ldots \subseteq \text{keys } (p1 \ast q2) \cup \text{keys } (p2 \ast q) \) by \((\text{rule Poly-Mapping.keys-add})\)

finally have \( \text{deg-pm } u < \text{poly-deg } p + \text{poly-deg } q \)

proof
assume \( u \in \text{keys } (p1 \ast q2) \)

then obtain \( s' \) \( t' \) where \( s' \in \text{keys } p1 \) and \( t' \in \text{keys } q2 \) and \( u; u = s' + t' \)

by \((\text{rule in-keys-timesE})\)

from \( (s' \in \text{keys } p1) \) have \( \text{deg-pm } s' = \text{poly-deg } p \) by \((\text{rule deg-p1})\)

moreover from \( (t' \in \text{keys } q2) \) have \( \text{deg-pm } t' < \text{poly-deg } q \) by \((\text{rule deg-q2})\)

ultimately show \( ?\)thesis by \((\text{simp add: u deg-pm-plus})\)

next
assume \( u \in \text{keys} \,(p_2 \ast q) \)
then obtain \( s' \, t' \) where \( s' \in \text{keys} \, p_2 \) and \( t' \in \text{keys} \, q \) and \( u: u = s' + t' \)
by (rule in-keys-timesE)
from \( s' \in \text{keys} \, p_2 \) have \( \text{deg-pm} \, s' < \text{deg-p} \) by (rule deg-p2)
moreover from \( t' \in \text{keys} \, q \) have \( \text{deg-pm} \, t' \leq \text{poly-deg} \, q \) by (rule poly-deg-max-keys)
ultimately show \(?\text{thesis} by (simp add: u \text{deg-pm-plus add-less-le-mono})
qed
thus \( \text{False} \) by (simp only: eq)
qed
with \( u \in \text{keys} \,(p_1 \ast q_1) \)
have \( u \in \text{keys} \,(p_1 \ast q_1 + (p_1 \ast q_2 + p_2 \ast q)) \) by (rule in-keys-plusI1)
thus \( u \in \text{keys} \,(p \ast q) \) by (simp only: p q algebra-simps)
qed
finally show \( \text{poly-deg} \, p + \text{poly-deg} \, q \leq \text{poly-deg} \,(p \ast q) \).
qed
corollary \( \text{poly-deg-monom-mult-le} \):
\( \text{poly-deg} \,(\text{punit}.\,\text{monom-mult} \, c \,(t:: \Rightarrow a::\text{add-order-min}) \, p) \leq \text{deg-pm} \, t + \text{poly-deg} \, p \)
proof –
have \( \text{poly-deg} \,(\text{punit}.\,\text{monom-mult} \, c \, t \, p) \leq \text{poly-deg} \,(\text{monomial} \, c \, t) + \text{poly-deg} \, p \)
by (simp only: times-monomial-left[symmetric] poly-deg-times-le)
also have \( ... \leq \text{deg-pm} \, t + \text{poly-deg} \, p \) by (simp add: poly-deg-monomial)
finally show \( ?\text{thesis} \).
qed
lemma \( \text{poly-deg-monom-mult} \):
assumes \( c \neq 0 \) and \( p \neq (0:: \Rightarrow a::\text{add-order-min}) \Rightarrow 0 \,'
b::\text{semiring-no-zero-divisors} \)
shows \( \text{poly-deg} \,(\text{punit}.\,\text{monom-mult} \, c \, t \, p) = \text{deg-pm} \, t + \text{poly-deg} \, p \)
proof (rule order.antisym, fact poly-deg-monom-mult-le)
from assms(2) obtain \( s \) where \( s \in \text{keys} \, p \) and \( \text{poly-deg} \, p = \text{deg-pm} \, s \) by (rule poly-degE)
have \( \text{deg-pm} \, t + \text{poly-deg} \, p = \text{deg-pm} \,(t + s) \) by (simp add: \( \langle \text{poly-deg} \, p = \text{deg-pm-s} \rangle \) \, \text{deg-pm-plus})
also have \( ... \leq \text{poly-deg} \,(\text{punit}.\,\text{monom-mult} \, c \, t \, p) \)
proof (rule poly-deg-max-keys)
from \( s \in \text{keys} \, p \) show \( t + s \in \text{keys} \,(\text{punit}.\,\text{monom-mult} \, c \, t \, p) \)
unfolding \( \text{punit}.\,\text{keys-monom-mult}[OF \, \text{assms}(1)] \) by fastforce
qed
finally show \( \text{deg-pm} \, t + \text{poly-deg} \, p \leq \text{poly-deg} \,(\text{punit}.\,\text{monom-mult} \, c \, t \, p) \).
qed
lemma \( \text{poly-deg-map-scale} \):
\( \text{poly-deg} \,(c \cdot p) = (if \, c = (0::\Rightarrow a::\text{semiring-no-zero-divisors}) \, then \, 0 \, else \, \text{poly-deg} \, p) \)
by (simp add: poly-deg-def keys-map-scale)
lemma \( \text{poly-deg-sum-le} \):
\( \langle \text{poly-deg} \,(\sum f \, A)\rangle :: a::\text{add-order-min} \leq \text{Max} \,(\text{poly-deg} \cdot f \cdot A) \)

437
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-binorder-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

438
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 ⇒₀ nat) ⇒₀ 'b::zero) ⇒ 'x set
where indets p = ∪ (keys ' keys p)
definition PPs :: 'x set ⇒ (('x ⇒₀ nat) ⇒₀ 'b::zero) set (.[·])
where PPs X = {t. keys t ⊆ X}
definition Polys :: 'x set ⇒ (('x ⇒₀ nat) ⇒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-indents: 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 \( \text{keys } t \neq \{\} \) by simp
then obtain \( x \) where \( x \in \text{keys } t \) by blast
from this \( t \in \text{keys } p \) have \( x \in \text{indets } p \) by (rule in-indetsI)
with assms show False by simp
qed
thus \( t \in \{0\} \) 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 \( v \) \( k \))) \( \subseteq \{v\} \)
proof
  fix \( x \) assume \( x \in \text{indets} (\text{monomial } c \text{ (Poly-Mapping.single } v k)) \)
  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 antisym; rule subsetI)
  from assms show \( v \in \text{indets} (\text{monomial } c \text{ (monomial } k v)) \) by (simp add: indets-def)
qed

lemma indets-monomial:
  assumes \( c \neq 0 \)
  shows indets (monomial \( c \) \( t \)) = \text{keys } t
proof (rule antisym; rule subsetI)
  fix \( x \) assume \( x \in \text{indets} (\text{monomial } c \text{ } t) \)
  then have \( \text{lookup } t x \neq 0 \) unfolding indets-def
    by (metis UN-E lookup-eq-zero-in-keys-contradict lookup-single-not-eq)
  thus \( x \in \text{keys } t \) by (meson lookup-not-eq-zero-eq-in-keys)
next
  fix \( x \) assume \( x \in \text{keys } t \)
  then have \( \text{lookup } t x \neq 0 \) by (meson lookup-not-eq-zero-eq-in-keys)
  thus \( x \in \text{indets} (\text{monomial } c \text{ } t) \) unfolding indets-def using assms
    by (metis UN-iff lookup-not-eq-zero-eq-in-keys lookup-single-eq)
indets-monomial-subset: \( \text{indets} \left( \text{monomial} \ c \ t \right) \subseteq \text{keys} \ t \)

by (cases \( c = 0 \), simp-all add: indets-def)

indets-monomial-zero [simp]: \( \text{indets} \left( \text{monomial} \ c \ 0 \right) = \{\} \)

by (simp add: indets-def)

indets-plus-subset: \( \text{indets} \left( p + q \right) \subseteq \text{indets} \ p \cup \text{indets} \ q \)

proof
  fix \ x
  assume \( x \in \text{indets} \left( p + q \right) \)
  then obtain \( t \) where \( x \in \text{keys} \ t \) and \( t \in \text{keys} \left( p + q \right) \) by (metis UN-E indets-def)
  hence \( t \in \text{keys} \ p \cup \text{keys} \ q \) by (metis Poly-Mapping.keys-add subsetCE)
  thus \( x \in \text{indets} \ p \cup \text{indets} \ q \) using indets-def \( x \in \text{keys} \ t \) by fastforce
qed

indets-uminus [simp]: \( \text{indets} \left( -p \right) = \text{indets} \ p \)

by (simp add: indets-def keys-uminus)

indets-minus-subset: \( \text{indets} \left( p - q \right) \subseteq \text{indets} \ p \cup \text{indets} \ q \)

proof
  fix \ x
  assume \( x \in \text{indets} \left( p - q \right) \)
  then obtain \( t \) where \( x \in \text{keys} \ t \) and \( t \in \text{keys} \left( p - q \right) \) by (metis UN-E indets-def)
  hence \( t \in \text{keys} \ p \cup \text{keys} \ q \) by (metis keys-minus subsetCE)
  thus \( x \in \text{indets} \ p \cup \text{indets} \ q \) using indets-def \( x \in \text{keys} \ t \) by fastforce
qed

indets-times-subset: \( \text{indets} \left( p \ast q \right) \subseteq \text{indets} \ p \cup \text{indets} \ q \)

proof
  fix \ x
  assume \( x \in \text{indets} \left( p \ast q \right) \)
  then obtain \( t \) where \( t \in \text{keys} \left( p \ast q \right) \) unfolding indets-def by blast
  from this(1) obtain \( u \ v \) where \( u \in \text{keys} \ p \ v \in \text{keys} \ q \) and \( t = u + v \) by (rule in-keys-timesE)
  hence \( x \in \text{keys} \ u \cup \text{keys} \ v \) by (metis \( x \in \text{keys} \ t \) Poly-Mapping.keys-add subsetCE)
  thus \( x \in \text{indets} \ p \cup \text{indets} \ q \) unfolding indets-def using \( u \in \text{keys} \ p \) \( v \in \text{keys} \ q \) by blast
qed

indets-monom-mult-subset: \( \text{indets} \left( \text{panit} \ast \text{monom} \ast \text{mult} \ c \ t \ p \right) \subseteq \text{keys} \ t \cup \text{indets} \ p \)

proof –
have \( \text{indets} (\text{punit.monom-mult} \ c \ t \ p) \subseteq \text{indets} (\text{monomial} \ c \ t) \cup \text{indets} \ p \) by (simp only: times-monomial-left[symmetric] indets-times-subset)
also have \( ... \subseteq \text{keys} \ t \cup \text{indets} \ p \) using indets-monomial-subset[of \ c \ t] by blast
finally show \(?\)thesis .
qed

lemma indets-monom-mult:
assumes \( c \neq 0 \) and \( p \neq (0::(\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat})) \)
shows \( \text{indets} (\text{punit.monom-mult} \ c \ t \ p) = \text{keys} \ t \cup \text{indets} \ p \) proof (rule in-indetsI)
fix \ x
assume \( x \in \text{keys} \ t \cup \text{indets} \ p \)
thus \( x \in \text{indets} (\text{punit.monom-mult} \ c \ t \ p) \) proof
assume \( x \in \text{keys} \ t \)
from assms(2) have \( \text{keys} \ p \neq \{\} \) by simp
then obtain \( s \) where \( s \in \text{keys} \ p \) by blast
hence \( t + s \in (+) \ t \cdot \text{keys} \ p \) by fastforce
also from assms(1) have \( ... = \text{keys} (\text{punit.monom-mult} \ c \ t \ p) \) by (simp add: punit.keys-monom-mult)
finally have \( t + s \in \text{keys} (\text{punit.monom-mult} \ c \ t \ p) \) .
show \(?\)thesis
proof (rule in-indetsI)
from \( x \in \text{keys} \ t \) show \( x \in \text{keys} (t + s) \) by (simp add: keys-plus-ninv-comm-monoid-add) qed
fact
next
assume \( x \in \text{indets} \ p \)
then obtain \( s \) where \( s \in \text{keys} \ p \) and \( x \in \text{keys} \ s \) by (rule in-indetsE)
from this(1) have \( t + s \in (+) t \cdot \text{keys} \ p \) by fastforce
also from assms(1) have \( ... = \text{keys} (\text{punit.monom-mult} \ c \ t \ p) \) by (simp add: punit.keys-monom-mult)
finally have \( t + s \in \text{keys} (\text{punit.monom-mult} \ c \ t \ p) \) .
show \(?\)thesis
proof (rule in-indetsI)
from \( x \in \text{keys} \ s \) show \( x \in \text{keys} (t + s) \) by (simp add: keys-plus-ninv-comm-monoid-add) qed
fact
qed
qed

lemma indets-sum-subset: \( \text{indets} (\text{sum} \ f \ A) \subseteq (\bigcup a \in A. \text{indets} \ (f \ a)) \)
proof (cases finite \( A \))
case True
thus \(?\)thesis
proof (induct \( A \))
case empty
show \(?\)case by simp
next
case (insert \( a \) \( A \))
have \( \text{indets} (\text{sum} \ f \ (\text{insert} \ a \ A)) \subseteq \text{indets} \ (f \ a) \cup \text{indets} \ (\text{sum} \ f \ A) \)

442
by (simp only: comm-monoid-add-class.sum.insert[OF insert(1) insert(2)])
indets-plus-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-prod-subset:
  indets (prod (f::-⇒ ((- ⇒0 ::cancel-comm-monoid-add) ⇒0 -)) A) ⊆ (∪ a∈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 ⇒0 nat) ⇒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

443
assume \( \text{poly-deg } p = 0 \)

hence monomial \((\text{lookup } p 0) \) \( 0 = p \) by (rule poly-deg-zero-imp-monomial)

moreover have \( \text{indets} \) \((\text{monomial } (\text{lookup } p 0) \) \( \emptyset \) by simp

ultimately show \( \text{indets } p = \emptyset \) by metis

qed

17.2.2 PPs

**lemma** PPsI: \( \text{keys } t \subseteq X \implies t \in .[X] \)

by (simp add: PPs-def)

**lemma** PPsD: \( t \in .[X] \implies \text{keys } t \subseteq X \)

by (simp add: PPs-def)

**lemma** PPs-empty [simp]: \( .[\emptyset] = \{0\} \)

by (simp add: PPs-def)

**lemma** PPs-UNIV [simp]: \( .[\text{UNIV}] = \text{UNIV} \)

by (simp add: PPs-def)

**lemma** PPs-singleton: \( .\{x\} = \text{range } (\text{Poly-Mapping}.\text{single } x) \)

**proof** (rule set-eqI)

fix \( t \)

show \( t \in .\{x\} \iff t \in \text{range } (\text{Poly-Mapping}.\text{single } x) \)

**proof**

assume \( t \in .\{x\} \)

hence \( \text{keys } t \subseteq \{x\} \) by (rule PPsD)

hence \( \text{Poly-Mapping}.\text{single } x (\text{lookup } t x) = t \) by (rule keys-subset-singleton-imp-monomial)

from this[ symmetric] UNIV-I show \( t \in \text{range } (\text{Poly-Mapping}.\text{single } x) \) ..

next

assume \( t \in \text{range } (\text{Poly-Mapping}.\text{single } x) \)

then obtain \( e \) where \( t = \text{Poly-Mapping}.\text{single } x e \) ..

thus \( t \in .\{x\} \) by (simp add: PPs-def)

qed

**lemma** zero-in-PPs: \( 0 \in .[X] \)

by (simp add: PPs-def)

**lemma** PPs-mono: \( X \subseteq Y \implies .[X] \subseteq .[Y] \)

by (auto simp: PPs-def)

**lemma** PPs-closed-single:

assumes \( x \in X \)

shows \( \text{Poly-Mapping}.\text{single } x e \in .[X] \)

**proof** (rule PPsI)

have \( \text{keys } (\text{Poly-Mapping}.\text{single } x e) \subseteq \{x\} \) by simp

also from \( \text{assms} \) have \( \ldots \subseteq X \) by simp

finally show \( \text{keys } (\text{Poly-Mapping}.\text{single } x e) \subseteq X \) .
lemma PPs-closed-plus:
  assumes $s \in \mathbb{X}$ and $t \in \mathbb{X}$
  shows $s + t \in \mathbb{X}$
proof –
  have $\text{keys} \ (s + t) \subseteq \text{keys} s \cup \text{keys} t$ by (fact Poly-Mapping.keys-add)
also from assms have $\ldots \subseteq \mathbb{X}$ by (simp add: PPs-def)
finally show $\text{thesis}$ by (rule PPsI)
qed

lemma PPs-closed-minus:
  assumes $s \in \mathbb{X}$
  shows $s - t \in \mathbb{X}$
proof –
  have $\text{keys} \ (s - t) \subseteq \text{keys} s$ by (metis lookup-minus lookup-not-eq-zero-eq-in-keys subsetI zero-diff)
also from assms have $\ldots \subseteq \mathbb{X}$ by (rule PPsD)
finally show $\text{thesis}$ by (rule PPsI)
qed

lemma PPs-closed-adds:
  assumes $s \in \mathbb{X}$ and $t \text{ adds } s$
  shows $t \in \mathbb{X}$
proof –
  from assms have $s = (s - t) + (s - t)$ by (metis add-minus-2 adds-minus)
moreover from assms have $s - (s - t) \in \mathbb{X}$ by (rule PPs-closed-minus)
ultimately show $\text{thesis}$ by simp
qed

lemma PPs-closed-gcs:
  assumes $s \in \mathbb{X}$
  shows $\text{gcs} \ s \ t \in \mathbb{X}$
using assms gcs-adds by (rule PPs-closed-adds)

lemma PPs-closed-lcs:
  assumes $s \in \mathbb{X}$ and $t \in \mathbb{X}$
  shows $\text{lcs} \ s \ t \in \mathbb{X}$
proof –
  from assms have $s + t \in \mathbb{X}$ by (rule PPs-closed-plus)
hence $(s + t) - \text{gcs} \ s \ t \in \mathbb{X}$ by (rule PPs-closed-minus)
thus $\text{thesis}$ by (simp add: gcs-plus-lcs[of $s$ $t$, symmetric])
qed

lemma PPs-closed-except': $t \in \mathbb{X} \implies \text{except} \ t \ Y \in \mathbb{X} - Y$
  by (auto simp: keys-except PPs-def)

lemma PPs-closed-except: $t \in \mathbb{X} \implies \text{except} \ t \ Y \in \mathbb{X}$
  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 \( \{x \mid x \in tx \} \subseteq X \) by (rule PPsD)
define \( tx \) where \( tx = \{x \mid x \in t \} \) (rule PPs-def)
hence \( tx \in [X] \) by (simp add: PPs-def)
have \( tx \in \{x \mid x \in t \} \) by (simp add: PPs-def)
proof (rule PPsI, rule)
fix \( x \)
assume \( x \in tx \) obtain \( tx \) where \( tx \in [X] \) and \( ty \in [Y] \) and \( t = tx + ty \) by (rule PPs-UnE)
from this(2) have \( t = tx + ty \) unfolding \( t = tx + ty \) by (rule imageI)
with \( tx \in [X] \) show \( t \in tx + ty \) by (rule)
next
assume \( t \in ?B \)
qed

lemma PPs-Un: \( [X \cup Y] = (\bigcup t \in [X]. (+ \cdot t \cdot [Y]) \) (is \(?A = ?B\))
proof (rule set-eqI)
fix \( t \)
show \( t \in ?A \iff t \in ?B \)
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 \( t \in (+ \cdot t \cdot [Y]) \) unfolding \( t = tx + ty \) by (rule imageI)
with \( tx \in [X] \) show \( t \in ?B \) by (rule)
next
assume \( t \in ?B \)
qed
then obtain $tx$ where $tx \in [X]$ and $t \in (+) tx \cdot [Y]$.

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 $PPs$-$insert$: $[insert \ x \ X] = (\bigcup e. (+) (Poly-Mapping.single \ x \ e) \cdot [X])$

proof

- have $[insert \ x \ X] = .\{(x) \cup X\}$ by simp

also have $\ldots = (\bigcup t \cdot [\{x\}], (+) t \cdot [X])$ by (fact $PPs$-$Un$)

also have $\ldots = (\bigcup e. (+) (Poly-Mapping.single \ x \ e) \cdot [X])$ by (simp add: $PPs$-singleton)

finally show $?thesis$.

qed

corollary $PPs$-$insert$I: 

assumes $tx \in [X]$ and $t = Poly-Mapping.single \ x \ e + tx$

shows $t \in [insert \ x \ X]$ 

proof

- from assumptions(1) have $t \in (+) (Poly-Mapping.single \ x \ e) \cdot [X]$ unfolding $PPs$-$insert$ ..

with $UNIV$-$I$ show $?thesis$ unfolding $PPs$-$insert$ by (rule $UN$-$I$)

qed

corollary $PPs$-$insert$E: 

assumes $t \in [insert \ x \ X]$ 

obtains $e \ tx$ where $tx \in [X]$ and $t = Poly-Mapping.single \ x \ e + tx$

proof

- from assumptions obtain $e$ where $t \in (+) (Poly-Mapping.single \ x \ e) \cdot [X]$ unfolding $PPs$-$insert$ ..

then obtain $tx$ where $tx \in [X]$ and $t = Poly-Mapping.single \ x \ e + tx$ ..

thus $?thesis$ ..

qed

lemma $PPs$-$Int$: $[X \cap Y] = .[X] \cap .[Y]$ 

by (auto simp: $PPs$-$def$)

lemma $PPs$-$INT$: $\bigcap X \cap Y = \bigcap (PPs \cdot X)$ 

by (auto simp: $PPs$-$def$)

17.2.3 Polys

lemma $Polys$-$alt$: $P[X] = \{p. \ keys \ p \subseteq .X\}$ 

by (auto simp: $Polys$-$def$ $PPs$-$def$ $indets$-$def$)

lemma $Polys$I: $keys \ p \subseteq .X \Longrightarrow p \in P[X]$ 

by (simp add: $Polys$-$def$)

lemma $Polys$-$I$-$alt$: $indets \ p \subseteq X \Longrightarrow p \in P[X]$

447
lemma PolysD:
  assumes \( p \in P[X] \).
  shows \( \text{keys } p \subseteq X \) and \( \text{indets } p \subseteq X \)
  using assms by (simp add: Polys-def, simp add: Polys-alt)

lemma Polys-empty: \( P[\{\}] = ((\text{range (Poly-Mapping.single 0))}) :: (('x \Rightarrow 'b::zero) set) \)
proof (rule set-eqI)
  fix \( p :: ('x \Rightarrow 'b::zero) \cdot \)
  show \( p \in P[\{\}] \leftrightarrow p \in \text{range (Poly-Mapping.single 0)} \)
  proof
    assume \( p \in P[\{\}] \)
    hence \( \text{keys } p \subseteq \{\{\} \} \) by (rule PolysD)
    also have \( \ldots = \{0\} \) by simp
    finally have \( \text{keys } p \subseteq \{0\} \).
    hence Poly-Mapping.single 0 (lookup p 0) = p by (rule keys-subset-singleton-imp-monomial)
    from this[symmetric] UNIV-I show \( p \in \text{range (Poly-Mapping.single 0)} \) ..
  next
    assume \( p \in \text{range (Poly-Mapping.single 0)} \)
    then obtain \( c \) where \( p = \text{monomial } c 0 \)
    thus \( p \in P[\{\}] \) by (simp add: Polys-def)
  qed

lemma Polys-UNIV[simp]: \( P[\text{UNIV}] = \text{UNIV} \)
  by (simp add: Polys-def)

lemma zero-in-Polys: \( 0 \in P[X] \)
  by (simp add: Polys-def)

lemma one-in-Polys: \( 1 \in P[X] \)
  by (simp add: Polys-def zero-in-PPs)

lemma Polys-mono: \( X \subseteq Y \Rightarrow P[X] \subseteq P[Y] \)
  by (auto simp: Polys-alt)

lemma Polys-closed-monomial: \( t \in .[X] \Rightarrow \text{monomial } c t \in P[X] \)
  using indets-monomial-subset[where \( c=\ldots \) and \( t=t \)] by (auto simp: Polys-alt PPs-def)

lemma Polys-closed-plus: \( p \in P[X] \Rightarrow q \in P[X] \Rightarrow p + q \in P[X] \)
  using indets-plus-subset[of \( p \) \( q \)] by (auto simp: Polys-alt PPs-def)

lemma Polys-closed-uminus: \( p \in P[X] \Rightarrow -p \in P[X] \)
  by (simp add: Polys-def keys-uminus)

lemma Polys-closed-minus: \( p \in P[X] \Rightarrow q \in P[X] \Rightarrow p - q \in P[X] \)
using indets-minus-subset[of p q] by (auto simp: Polys-alt PPs-def)

lemma Polys-closed-monom-mult: \( t \in \mathbb{D} \) \( \Longrightarrow p \in \mathbb{P}[X] \) \( \Longrightarrow \) punit.monom-mult
\( c \in \mathbb{P}[X] \)
using indets-monom-mult-subset[of c t p] by (auto simp: Polys-alt PPs-def)

corollary Polys-closed-map-scale: \( p \in \mathbb{P}[X] \) \( \Longrightarrow (c\cdot p) \in \mathbb{P}[X] \)
unfolding punit.map-scale-eq-monom-mult using zero-in-PPs by (rule Polys-closed-monom-mult)

lemma Polys-closed-times: \( p \in \mathbb{P}[X] \) \( \Longrightarrow q \in \mathbb{P}[X] \) \( \Longrightarrow p \cdot q \in \mathbb{P}[X] \)
using indets-times-subset[of p q] by (auto simp: Polys-alt PPs-def)

lemma Polys-closed-power: \( p \in \mathbb{P}[X] \) \( \Longrightarrow p \cdot m \in \mathbb{P}[X] \)
by (induct m) (auto intro: one-in-Polys Polys-closed-times)

lemma Polys-closed-sum: \( \bigwedge a. a \in A \Longrightarrow f a \in \mathbb{P}[X] \) \( \Longrightarrow \) sum f A \( \in \mathbb{P}[X] \)
by (induct A rule: infinite-finite-induct) (auto intro: zero-in-PPs Polys-closed-plus)

lemma Polys-closed-prod: \( \bigwedge a. a \in A \Longrightarrow f a \in \mathbb{P}[X] \) \( \Longrightarrow \) prod f A \( \in \mathbb{P}[X] \)
by (induct A rule: infinite-finite-induct) (auto intro: one-in-Polys Polys-closed-times)

lemma Polys-closed-sum-list: \( \bigwedge x. x \in \text{set} \: \text{xs} \Longrightarrow x \in \mathbb{P}[X] \) \( \Longrightarrow \) sum-list \( \text{xs} \in \mathbb{P}[X] \)
by (induct \( \text{xs} \)) (auto intro: zero-in-Polys Polys-closed-plus)

lemma Polys-closed-except: \( p \in \mathbb{P}[X] \) \( \Longrightarrow \) except \( p \in \mathbb{P}[X] \)
by (auto intro!: PolysI simp: keys-except dest!: PolysD(1))

lemma times-in-PolysD:
assumes \( p \cdot q \in \mathbb{P}[X] \) and \( p \in \mathbb{P}[X] \) and \( p \neq 0 \)\( \cdot \) \( p \notin \{0\} \)\( \cdot \) \( p \notin \{0\} \)
shows \( q \in \mathbb{P}[X] \)
proof
define \( qX \) where \( qX = \text{except} \: q \: (\cdot \: \mathbb{P}[X]) \)
define \( qY \) where \( qY = \text{except} \: q \: \mathbb{P}[X] \)
have \( q = qX + qY \) by (simp only: \( qX\)-def \( qY\)-def add.commute flip: except-decomp)
have \( qX \in \mathbb{P}[X] \) by (rule PolysI) (simp add: \( qX\)-def keys-except)
with \( \text{assms}(2) \) have \( p \cdot qX \in \mathbb{P}[X] \) by (rule Polys-closed-times)
show \( ?\text{thesis} \)
proof (cases \( qY = 0 \))
case True
with \( qX \in \mathbb{P}[X] \) show \( ?\text{thesis} \) by (simp add: q)
next
case False
with \( \text{assms}(3) \) have \( p \cdot qY \neq 0 \) by simp
hence \( \text{keys} \: (p \cdot qY) \neq \{\} \) by simp
then obtain \( t \) where \( t \in \text{keys} \: (p \cdot qY) \) by blast
then obtain \( t1 \) \( t2 \) where \( t2 \in \text{keys} \: qY \) and \( t : t = t1 + t2 \) by (rule in-keys-timesE)
have \( t \not\in [X] \) unfolding \( t \)

proof

assume \( t1 + t2 \in [X] \)

hence \( t1 + t2 - t1 \in [X] \) by (rule \( PPs\text{-}closed\text{-}minus \))

hence \( t2 \in [X] \) by simp

with \( \langle t2 \in keys qY \rangle \) show False by (simp add: \( qY\text{-def} \) \( keys\text{-}except \))

qed

have \( t \not\in keys (p * qX) \)

proof

assume \( t \in keys (p * qX) \)

also from \( (p * qX \in P[X]) \) have \( \ldots \subseteq [X] \) by (rule \( PolysD \))

finally have \( t \in [X] \)

with \( \langle t \not\in [X] \rangle \) show False ..

qed

... also have \( \ldots = keys (p * q) \) by (simp only: \( q \) algebra\text{-}simps)

finally have \( p * q \not\in P[X] \) using \( \langle t \not\in [X] \rangle \) by (auto simp: \( Polys\text{-}def \))

thus \( \neg \)thesis using assms(1) ..

qed

lemma \( poly\text{-}mapping\text{-}plus\text{-}induct\text{-}Polys \) [consumes 1, case\text{-}names 0 plus]:

assumes \( p \in P[X] \) and \( P \)

and \( \forall p t. t \in [X] \Rightarrow p \in P[X] \Rightarrow c \neq 0 \Rightarrow t \not\in keys p \Rightarrow P \Rightarrow P \)

(monomial \( c t + p \))

shows \( P \)

using assms(1)

proof (induct \( p \) rule: \( poly\text{-}mapping\text{-}plus\text{-}induct \))

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\text{-}is\text{-}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 \( P \)

by (rule step.hyps)

with \( \langle t \in [X] \rangle \) \( \langle p \in P[X] \rangle \) step.hyps(1, 2) show \( ?case \) by (rule assms(3))

qed

lemma \( Polys\text{-}Int: P[X \cap Y] = P[X] \cap P[Y] \)

by (auto simp: \( Polys\text{-}def \) \( PPs\text{-}Int \))

450
lemma Polys-INT: $P \bigcap X = \bigcap (Polys \cdot X)$
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 :: 'a::comm-semiring-1) ⇒ ('y ⇒ 0 nat) ⇒ 0 nat) ⇒ ('x ⇒ ('y ⇒ 0 nat) ⇒ 0 nat)
where subst-pp f t = (∏ x∈keys t. (f x) ^ (lookup t x))

definition poly-subst :: (('x :: 'a::comm-semiring-1) ⇒ ('y ⇒ 0 nat) ⇒ 0 nat) ⇒ ('x ⇒ ('y ⇒ 0 nat) ⇒ 0 nat)
where poly-subst f p = (∑ t∈keys p. punit.monom-mult (lookup p t) 0 (subst-pp f t))

lemma subst-pp-alt: subst-pp f t = (∏ x. (f x) ^ (lookup t x))
proof
  from finite-keys have subst-pp f t = (∏ x. if x ∈ keys t then (f x) ^ (lookup t x) else 1)
  unfolding subst-pp-def by (rule Prod-any.conditionlize)
  also have ... = (∏ 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::('a::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::('a::comm-semiring-1))
  proof
    from x ∈ keys t have 0 < lookup t x by (simp add: in-keys-iff)
    thus 0 ^ lookup t x = (0::('a::comm-semiring-1)) 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) \neq 1 \} \subseteq keys t
proof (rule, simp)
  fix x
  assume f x ° (lookup t x) \neq 1
  thus x \in keys t unfolding in-keys-iff by (metis power-0)
qed

corollary finite-power-lookup-not-one: finite \{ x. f x ° (lookup t x) \neq 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 \( \forall x. x \in keys t \implies f x = \text{monomial 1} \) (Poly-Mapping.single x 1)
shows subst-pp f t = monomial 1 t
proof -
  have subst-pp f t = (\( \prod_{x \in keys t} \text{monomial 1} \) (Poly-Mapping.single x (lookup t x)))
  proof (simp only: subst-pp-def, rule prod.cong, fact refl)
    fix x
    assume x \in keys t
    thus f x ° lookup t x = \text{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 \in indets (subst-pp f t)
obtains y where y \in keys t and x \in indets (f y)
proof -
  note assms
  also have indets (subst-pp f t) \subseteq (\( \bigcup_{y \in keys t} \text{indets} \ ((f y) ° (lookup t y)) \))
unfolding subst-pp-def
  by (rule indets-prod-subset)
finally obtain y where y \in keys t and x \in indets ((f y) ° (lookup t y)) ..
note this(2)
also have indets ((f y) ° (lookup t y)) \subseteq indets (f y) by (rule indets-power-subset)
finally have x \in indets (f y) .
with \( y \in keys t \) show thesis .
qed

lemma subst-pp-by-monomials:
assumes \( \bigwedge y. \ y \in \text{keys } t \implies f \ y = \text{monomial } (c \ y) (s \ y) \)
shows \( \text{subst-pp } f \ t = \text{monomial } (\prod y \in \text{keys } t. (c \ y)^{\text{lookup } t \ y}) (\sum y \in \text{keys } t. \text{lookup } t \ y \cdot s \ y) \)
by (simp add: subst-pp-def assms monomial-power-map-scale punit.monomial-prod-sum)

lemma \( \text{poly-deg-subst-pp-eq-zeroI} \):
assumes \( \bigwedge x. \ x \in \text{keys } t \implies \text{poly-deg } (f \ x) = 0 \)
shows \( \text{poly-deg } (\text{subst-pp } f \ t) = 0 \)
proof
have \( \text{poly-deg } (\text{subst-pp } f \ t) \leq (\sum x \in \text{keys } t. \text{poly-deg } ((f \ x) \cdot (\text{lookup } t \ x))) \)
unfolding subst-pp-def by (fact poly-deg-prod-le)
also have \( \ldots = 0 \) by (simp add: ⟨poly-deg \( f \ x \) \( \leq 1 \)⟩)
finally show \( \text{poly-deg } (f \ x \cdot \text{lookup } t \ x) = 0 \) by simp
qed
finally show \( ?\text{thesis} \) by simp
qed

lemma \( \text{poly-deg-subst-pp-le} \):
assumes \( \bigwedge x. \ x \in \text{keys } t \implies \text{poly-deg } (f \ x) \leq 1 \)
shows \( \text{poly-deg } (\text{subst-pp } f \ t) \leq \text{deg-pm } t \)
proof
have \( \text{poly-deg } (\text{subst-pp } f \ t) \leq (\sum x \in \text{keys } t. \text{poly-deg } ((f \ x) \cdot (\text{lookup } t \ x))) \)
unfolding subst-pp-def by (fact poly-deg-prod-le)
also have \( \ldots \leq (\sum x \in \text{keys } t. \text{lookup } t \ x) \)
proof (rule sum-mono)
fix \( x \)
assume \( x \in \text{keys } t \)
\( \text{poly-deg } (f \ x) \leq 1 \) by (rule assms)
have \( f \ x \cdot \text{lookup } t \ x = (\prod i=0..<\text{lookup } t \ x. f \ x) \) by simp
also have \( \text{poly-deg } \ldots \leq (\sum i=0..<\text{lookup } t \ x. \text{poly-deg } (f \ x)) \) by (rule poly-deg-prod-le)
also from \( \text{poly-deg } (f \ x) \leq 1 \) have \( \ldots \leq (\sum i=0..<\text{lookup } t \ x. 1) \) by (rule sum-mono)
finally show \( \text{poly-deg } (f \ x \cdot \text{lookup } t \ x) \leq \text{lookup } t \ x \) by simp
qed
also have \( \ldots = \text{deg-pm } t \) by (rule deg-pm-superset[symmetric], fact subset-refl, fact finite-keys)
finally show \( ?\text{thesis} \) by simp
qed

lemma \( \text{poly-subst-alt} \): \( \text{poly-subst } f \ p = (\sum t. \text{punit.monom-mul} \ (\text{lookup } p \ t) \ 0) (\text{subst-pp } f \ t) \)
proof

from finite-keys have poly-subst f p = (∑ t. if t ∈ keys p then punit.monom-mult (lookup p t) 0 (subst-pp f t) else 0)
  unfolding poly-subst-def by (rule Sum-any.conditionallize)
also have ... = (∑ t. punit.monom-mult (lookup p t) 0 (subst-pp f t))
  by (rule Sum-any.cong) (simp add: in-keys-iff)
finally show ?thesis .
qed

lemma poly-subst-trivial [simp]: poly-subst (λ-. 0) p = monomial (lookup p 0) 0
  by (simp add: poly-subst-def subst-pp-trivial if-distrib in-keys-iff cong)
  (metis mult.right-neutral times-monomial-left)

lemma poly-subst-zero [simp]: poly-subst f 0 = 0
  by (simp add: poly-subst-def)

lemma monom-mult-lookup-not-zero-subset-keys:
  {t. punit.monom-mult (lookup p t) 0 (subst-pp f t) ≠ 0} ⊆ keys p
proof (rule, simp)
  fix t
  assume punit.monom-mult (lookup p t) 0 (subst-pp f t) ≠ 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-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⇒0 nat ⇒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⇒0 nat} ⇒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

454
lemma poly-subst-times: poly-subst f \( (p \times q) \) = poly-subst f p \* poly-subst f q

proof -
  have bij: bij \((\lambda(l, n, m). (m, l, n))\) by (auto intro!: bij injI simp add: image-def)
  let \(?P = keys p\)
  let \(?Q = keys q\)
  let \(?PQ = \{ s + t \mid s t. lookup p s \neq 0 \land lookup q t \neq 0 \}\)
  have fin-PQ: finite ?PQ by (rule finite-not-eq-zero-sumI, simp-all)
  have fin-1: finite \{ l. lookup p l * (\(\sum qa. lookup q qa\) when t = l + qa) \neq 0 \} for t
    proof (rule finite-subset)
      show \{ l. lookup p l * (\(\sum qa. lookup q qa\) when t = l + qa) \neq 0 \} \subseteq 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) \neq 0 \} for t u
      proof (rule finite-subset)
        show \{ v. (lookup q v when t = u + v) \neq 0 \} \subseteq 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) \neq 0 \} for t u
      proof (rule finite-subset)
        show \{ v. (lookup p u * lookup q v when t = u + v) \neq 0 \} \subseteq keys p
      qed (fact finite-keys)
    have \((\sum t. \text{punit.monom-mult} (lookup (p \times q) t) \circ (\text{subt-pp f t}))\) =
      \((\sum t. \sum u. \text{punit.monom-mult} (lookup p u * (\(\sum v. lookup q v\) when t = u + v)) \circ (\text{subt-pp f t}))\)
      by (simp add: times-poly-mapping.rep-eq prod-fun-fun-def punit.monom-mult-Sum-any-left[OF fin-1])
    also have \(\ldots = (\sum t. \sum u. \sum v. (\text{punit.monom-mult} (lookup p u * lookup q v) \circ (\text{subt-pp f t}))\) when t = u + v\)
    also have \(\ldots = (\sum t. (\sum (u, v). (\text{punit.monom-mult} (lookup p u * lookup q v) \circ (\text{subt-pp f t}))\) when t = u + v)\)
      by (auto simp: in-keys-iff)
    also have \(\ldots = (\sum (u, v, t). \text{punit.monom-mult} (lookup p u * lookup q v) \circ (\text{subt-pp f t}))\) when t = u + v\)
      apply (auto simp: fin-PQ in-keys-iff)
      apply (metis monomial-0I mult-not-zero times-monomial-left)
      done
    also have \(\ldots = (\sum (u, v, t). \text{punit.monom-mult} (lookup p u * lookup q v) \circ (\text{subt-pp f t}))\) when t = u + v\)
      using bij by (rule Sum-any.reindex-cong [of \(\lambda(u, v, t). (t, u, v))\]) (simp add: fun-eq-iff)
also have \[ \ldots = (\sum (u, v). \sum t. \text{punit.monom-mult} \ (\text{lookup} \ p \ u \ast \text{lookup} \ q \ v)) \ast 0 \]
(subst-pp \ f \ t) when t = u + v
  apply (subst \ Sum-any.cartesian-product2 \ [of \ (?P \times \ ?Q) \times \ ?PQ])
  apply (auto simp: \ fin-PQ \ in-keys-iff)
  apply (metis \ monomial-0I \ mult-not-zero \ times-monomial-left)
  done
also have \[ \ldots = (\sum (u, v). \text{punit.monom-mult} \ (\text{lookup} \ p \ u \ast \text{lookup} \ q \ v)) \ast 0 \]
(subst-pp \ f \ u \ast \ subst-pp \ f \ v))
  by (simp add: subst-pp-plus)
also have \[ \ldots = (\sum u. \sum v. \text{punit.monom-mult} \ (\text{lookup} \ p \ u \ast 0 \ (\text{subst-pp} \ f \ u))) \ast (\sum t. \text{punit.monom-mult} \ (\text{lookup} \ q \ v \ast 0 \ (\text{subst-pp} \ f \ v))) \]
  by (simp add: times-monomial-left symmetric \ ac-simps \ mult-single)
also have \[ \ldots = (\sum t. \text{punit.monom-mult} \ (\text{lookup} \ p \ t \ast 0 \ (\text{subst-pp} \ f \ t))) \ast (\sum t. \text{punit.monom-mult} \ (\text{lookup} \ q \ t \ast 0 \ (\text{subst-pp} \ f \ t))) \]
  by (rule \ Sum-any-product symmetric, (fact \ finite-monom-mult-lookup-not-zero)+)
finally show \( \text{thesis} \) by (simp add: poly-subst-alt)
qed

corollary \text{poly-subst-monom-mult}:
\[ \text{poly-subst} \ f \ (\text{punit.monom-mult} \ c \ t \ p) = (\text{punit.monom-mult} \ c \ast 0 \ (\text{subst-pp} \ f \ t \ast \ polymonomialassoc)) \]
  by (simp only: \ times-monomial-left \ symmetric \ poly-subst-times \ poly-subst-monomialassoc)

corollary \text{poly-subst-monom-mult}':
\[ \text{poly-subst} \ f \ (\text{punit.monom-mult} \ c \ t \ p) = (\text{punit.monom-mult} \ c \ast 0 \ (\text{subst-pp} \ f \ t)) \ast \ polymonomialassoc \]
  by (simp only: \ times-monomial-left symmetric \ poly-subst-times \ poly-subst-monomialassoc)

lemma \text{poly-subst-sum}:
\[ \text{poly-subst} \ f \ (\text{sum} \ p \ A) = (\sum a \in A. \text{poly-subst} \ f \ (p \ a)) \]
  by (rule \ fun-sum-commute, simp-all add: poly-subst-plus)

lemma \text{poly-subst-prod}:
\[ \text{poly-subst} \ f \ (\text{prod} \ p \ A) = (\prod a \in A. \text{poly-subst} \ f \ (p \ a)) \]
  by (rule \ fun-prod-commute, simp-all add: poly-subst-times)

lemma \text{poly-subst-power}:
\[ \text{poly-subst} \ f \ (p \ ^ \ n) = (\text{poly-subst} \ f \ p) \ ^ \ n \]
  by (induct \ n, simp-all add: poly-subst-times)

lemma \text{poly-subst-subst-pp}:
\[ \text{poly-subst} \ f \ (\text{subst-pp} \ g \ t) \Rightarrow \text{subst-pp} \ (\lambda x. \text{poly-subst} \ f \ (g \ x)) \ t \]
  by (simp only: \ subst-pp-def \ poly-subst-prod \ poly-subst-power)

lemma \text{poly-subst-poly-subst}:
\[ \text{poly-subst} \ f \ (\text{poly-subst} \ g \ p) = \text{poly-subst} \ (\lambda x. \text{poly-subst} \ f \ (g \ x)) \ p \]
proof -
  have \( \text{poly-subst} \ f \ (\text{poly-subst} \ g \ p) = \)

456
poly-subst \( f (\sum_{t \in \text{keys } p} punit.\text{monom-mult} (\text{lookup } p t) 0 (\text{subst-pp } g t)) \)
by (simp only: poly-subst-def)
also have \( \ldots = (\sum_{t \in \text{keys } p} punit.\text{monom-mult} (\text{lookup } p t) 0 (\text{subst-pp } (\lambda x. poly-subst f (g x)) t)) \)
by (simp add: poly-subst-sum poly-subst-monom-mult poly-subst-subst-pp)
also have \( \ldots = poly-subst (\lambda x. poly-subst f (g x)) p \) by (simp only: poly-subst-def)
finally show \( \text{thesis} \).
qed

lemma poly-subst-id:
assumes \( \forall x. x \in \text{indets } p \implies f x = \text{monomial } 1 \) (Poly-Mapping.single \( x \) \( I \))
shows poly-subst \( f p = p \)
proof –
have poly-subst \( f p = (\sum_{t \in \text{keys } p} \text{monomial} (\text{lookup } p t) t) \)
proof (simp only: poly-subst-def, rule sum.cong, fact refl)
fix \( t \)
assume \( t \in \text{keys } p \)
have eq: \( \text{subst-pp } f t = \text{monomial } 1 t \)
by (rule subst-pp-id, rule assms, erule in-indetsI, fact \( t \in \text{keys } p \) )
show punit.\text{monom-mult} (\text{lookup } p t) 0 (\text{subst-pp } f t) = \text{monomial} (\text{lookup } p t)
by (simp add: eq punit.\text{monom-mult-monomial})
qed
also have \( \ldots = p \) by (simp only: poly-mapping-sum-monomials)
finally show \( \text{thesis} \).
qed

lemma in-keys-poly-substE:
assumes \( t \in \text{keys } (\text{poly-subst } f p) \)
obtains \( s \) where \( s \in \text{keys } p \) and \( t \in \text{keys } (\text{subst-pp } f s) \)
proof –
note assms
also have \( \text{keys } (\text{poly-subst } f p) \subseteq (\bigcup_{t \in \text{keys } p} \text{keys } (\text{punit.\text{monom-mult} (\text{lookup } p t) 0 (\text{subst-pp } f t)})) \)
unfolding poly-subst-def by (rule keys-sum-subset)
finally obtain \( s \) where \( s \in \text{keys } p \) and \( t \in \text{keys } (\text{punit.\text{monom-mult} (\text{lookup } p s) 0 (\text{subst-pp } f s)}) \)
.. note this(2)
also have \( \ldots \subseteq (+) 0 \) \( ' \text{keys } (\text{subst-pp } f s) \) by (rule punit.\text{keys-monom-mult-subset}[simplified])
also have \( \ldots = \text{keys } (\text{subst-pp } f s) \) by simp
finally have \( t \in \text{keys } (\text{subst-pp } f s) \).
with \( s \in \text{keys } p \) show \( \text{thesis} \).
qed

lemma in-indets-poly-substE:
assumes \( x \in \text{indets } (\text{poly-subst } f p) \)
obtains \( y \) where \( y \in \text{indets } p \) and \( x \in \text{indets } (f y) \)
proof –
note assms

457
also have \( \text{indets} \ (\text{poly-subst} \ f \ p) \subseteq (\bigcup_{t \in \text{keys} \ p} \text{indets} \ (\text{punit.monom-mult} \ (\text{lookup} \ p \ t) \ 0 \ (\text{subst-pp} \ f \ t))) \)

unfolding \( \text{poly-subst-def} \) by \( \text{(rule indets-sum-subset)} \)

finally obtain \( t \) where \( t \in \text{keys} \ p \) and \( x \in \text{indets} \ (\text{punit.monom-mult} \ (\text{lookup} \ p \ t) \ 0 \ (\text{subst-pp} \ f \ t)) \).

note this(2)

also have \( \text{indets} \ (\text{punit.monom-mult} \ (\text{lookup} \ p \ t) \ 0 \ (\text{subst-pp} \ f \ t)) \subseteq \text{keys} \ (0::(\alpha \Rightarrow 0 \ \text{nat})) \cup \text{indets} \ (\text{subst-pp} \ f \ t) \)

by \( \text{(rule indets-monom-mult-subset)} \)

also have \( \ldots = \text{indets} \ (\text{subst-pp} \ f \ t) \) by simp

finally obtain \( y \) where \( y \in \text{keys} \ t \) and \( x \in \text{indets} \ (f \ y) \) by \( \text{(rule in-indets-subst-ppE)} \)

from this(1) \( \langle t \in \text{keys} \ p \rangle \) have \( y \in \text{indets} \ p \)

by \( \text{(rule in-indetsI)} \)

from this \( \langle x \in \text{indets} \ (f \ y) \rangle \) show \?thesis ..

qed

lemma \( \text{poly-deg-poly-subst-eq-zeroI} \):

assumes \( \forall x. \ x \in \text{indets} \ p \Rightarrow \text{poly-deg} \ (f \ x) = 0 \)

shows \( \text{poly-deg} \ (\text{poly-subst} \ (f:::-\Rightarrow (\lambda \ t. \ \text{punit.monom-mult} \ (\text{lookup} \ p \ t) \ 0 \ (\text{subst-pp} \ f \ t)) \ 0 \ 0) \ 0 \ 0 \ 0 \ b::\text{comm-semiring-1}) = 0 \)

proof \( \text{(cases} \ p = 0) \)

case True

thus \?thesis by simp

next

case False

have \( \text{poly-deg} \ (\text{poly-subst} \ f \ p) \leq \text{Max} \ (\text{poly-deg} \ (\lambda t. \ \text{punit.monom-mult} \ (\text{lookup} \ p \ t) \ 0 \ (\text{subst-pp} \ f \ t)) \ 0 \ \text{keys} \ p) \)

unfolding \( \text{poly-subst-def} \) by \( \text{(fact poly-deg-sum-le)} \)

also have \( \ldots \leq 0 \)

proof \( \text{(rule Max.boundedI)} \)

show \( \text{finite} \ (\text{poly-deg} \ (\lambda t. \ \text{punit.monom-mult} \ (\text{lookup} \ p \ t) \ 0 \ (\text{subst-pp} \ f \ t)) \ 0 \ \text{keys} \ p) \)

by \( \text{(simp add: finite-image-iff)} \)

next

from False show \( \text{poly-deg} \ (\lambda t. \ \text{punit.monom-mult} \ (\text{lookup} \ p \ t) \ 0 \ (\text{subst-pp} \ f \ t)) \ 0 \ \text{keys} \ p \neq \{\} \) by simp

next

fix \( d \)

assume \( d \in \text{poly-deg} \ (\lambda t. \ \text{punit.monom-mult} \ (\text{lookup} \ p \ t) \ 0 \ (\text{subst-pp} \ f \ t)) \ 0 \ \text{keys} \ p \)

then obtain \( t \) where \( t \in \text{keys} \ p \) and \( d = \text{poly-deg} \ (\text{punit.monom-mult} \ (\text{lookup} \ p \ t) \ 0 \ (\text{subst-pp} \ f \ t)) \)

by fastforce

have \( d \leq \text{deg-pm} \ (0::\alpha \Rightarrow 0 \ \text{nat}) + \text{poly-deg} \ (\text{subst-pp} \ f \ t) \)

unfolding \( d \) by \( \text{(fact poly-deg-monom-mult-le)} \)

also have \( \ldots = \text{poly-deg} \ (\text{subst-pp} \ f \ t) \) by simp

also have \( \ldots = 0 \) by \( \text{(rule poly-deg-subst-pp-eq-zeroI, rule assms, erule in-indetsI, fact)} \)

finally show \( d \leq 0 \).

qed
finally show ?thesis by simp
qed

lemma poly-deg-poly-subst-le:
  assumes \( \forall x. x \in \text{indets } p \Rightarrow \text{poly-deg } (f \ x) \leq 1 \)
  shows \( \text{poly-deg } (\text{poly-subst } (f := ((\ y \Rightarrow_0 \cdot) \Rightarrow_0 \cdot)) \ (p := (\ x \Rightarrow_0 \text{nat}) \Rightarrow_0 \cdot)) \leq \text{poly-deg } p \)
proof (cases \( p = 0 \))
  case True
  thus ?thesis by simp
next
  case False
  have \( \text{poly-deg } (\text{poly-subst } f \ p) \leq \text{Max } (\text{poly-deg } (\lambda x. \text{punit}.\text{monom-mul} \ (\text{lookup } p \ t) \ 0 \ (\text{subst-pp } f \ t)) \ ' \text{keys } p) \)
    unfolding poly-subst-def by (fact poly-deg-sum-le)
  also have \( \ldots \leq \text{poly-deg } p \)
    proof (rule Max.boundedI)
      show finite (\( \text{poly-deg } (\lambda x. \text{punit}.\text{monom-mul} \ (\text{lookup } p \ t) \ 0 \ (\text{subst-pp } f \ t)) \ ' \text{keys } p) \)
        by (simp add: finite-image-iff)
    next
    from False show \( \text{finite } (\lambda t. \text{punit}.\text{monom-mul} \ (\text{lookup } p \ t) \ 0 \ (\text{subst-pp } f \ t)) \ ' \text{keys } p \neq \{\} \) by simp
  next
    fix \( d \)
    assume \( d \in \text{poly-deg } (\lambda t. \text{punit}.\text{monom-mul} \ (\text{lookup } p \ t) \ 0 \ (\text{subst-pp } f \ t)) \ ' \text{keys } p \)
    then obtain \( t \) where \( t \in \text{keys } p \) and \( d = \text{poly-deg } (\text{punit}.\text{monom-mul} \ (\text{lookup } p \ t) \ 0 \ (\text{subst-pp } f \ t)) \ ' \text{keys } p \)
      by fastforce
    have \( d \leq (\text{deg-pm } (0 :: \cdot \Rightarrow_0 \text{nat}) + \text{poly-deg } (\text{subst-pp } f \ t)) \)
      unfolding d by (fact poly-deg-monom-mult-le)
    also have \( \ldots = (\text{poly-deg } (\text{subst-pp } f \ t)) \) by simp
    also have \( \ldots \leq \text{deg-pm } t \) by (rule poly-deg-subst-pp-le, rule assms, erule in-indetsI, fact)
    also from \( t \in \text{keys } p \) have \( \ldots \leq \text{poly-deg } p \) by (rule poly-deg-max-keys)
    finally show \( d \leq \text{poly-deg } p \).
qed
finally show ?thesis by simp
qed

lemma subst-pp-cong: \( s = t \Longrightarrow (\forall x. x \in \text{keys } t \Longrightarrow f \ x = g \ x) \Longrightarrow \text{subst-pp } f \ s = \text{subst-pp } g \ t \)
  by (simp add: subst-pp-def)

lemma poly-subst-cong:
  assumes \( p = q \) and \( \forall x. x \in \text{indets } q \Longrightarrow f \ x = g \ x \)
  shows \( \text{poly-subst } f \ p = \text{poly-subst } g \ q \)
proof (simp add: poly-subst-def assms (1), rule sum.cong)
fix \( t \)
assume \( t \in \text{keys } q \)
\{
    fix \( x \)
    assume \( x \in \text{keys } t \)
with \( t \in \text{keys } q \) have \( x \in \text{indets } q \) by (auto simp: indets-def)
    hence \( f \ x = g \ x \) by (rule assms(2))
\}
thus \( \text{punit.monom-mult (lookup } q \ t) \ 0 \ (\text{subst-pp } f \ t) = \text{punit.monom-mult (lookup} q \ t) \ 0 \ (\text{subst-pp } g \ t) \) by (simp cong: subst-pp-cong)
qed (fact refl)

lemma Polys-homomorphismE:
obtains \( h \) where \( \bigwedge p q. \ h \ (p + q) = h p + h q \) and \( \bigwedge p q. \ h \ (p \ast q) = h p \ast h q \) and \( \bigwedge p::{'(x \Rightarrow 'a::comm-ring-1)}. \ h \ (h p) = h p \) and range \( h = P[X] \)
proof –
let \( \exists f = \lambda x. \text{if } x \in X \text{ then } \text{monomial } \bigwedge x.\ (\text{Poly-Mapping.single } x 1) \text{ else } 1 \)

have 1: \( \text{poly-subst } (?f \ p = p \text{ if } p \in P[X] \text{ for } p} \)
proof (rule poly-subst-id)
    fix \( x \)
    assume \( x \in \text{indets } p \)
    also from that have \( \ldots \subseteq X \) by (rule PolysD)
    finally show \( \text{if } x = \text{monomial } 1 \text{ (Poly-Mapping.single } x 1) \) by simp
qed

have 2: \( \text{poly-subst } (?f \ p \in P[X] \text{ for } p} \)
proof (intro PolysI-alt subsetI)
    fix \( x \)
    assume \( x \in \text{indets (poly-subst } (?f \ p)} \)
    then obtain \( y \) where \( x \in \text{indets } (?f y) \) 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 \( \exists \)thesis
proof
    fix \( p \)
    from 2 show \( \text{poly-subst } (?f \ (\text{poly-subst } (?f \ p) = \text{poly-subst } (?f \ p) \text{ by (rule 1)} \)
next
    show range \( \text{poly-subst } (?f) = P[X] \)
proof (intro set-eqI iffI)
    fix \( x :: - \Rightarrow 'a \)
    assume \( x \in P[X] \)
    hence \( p = \text{poly-subst } (?f \ p \) by (simp only: 1)
    thus \( p \in \text{range (poly-subst } (?f \) by (rule image-eqI) simp
qed (auto intro: 2)
qed
lemma in-idealE-Polys-finite:
  assumes finite B and B ⊆ P[X] and p ∈ P[X] and (p::(λx.⇒_0 nat)) ⇒_0 'a::comm-ring-1) ∈ ideal B
  obtains q where ∩ b. q b ∈ P[X] and p = (∑ b∈B. q b * b)
proof –
  obtain h where ∩ p q. h (p + q) = h p + h q and ∩ p q. h (p * q) = h p * h q
  and ∩ p::(λx.⇒_0 nat) ⇒_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 ∩ b. q b ∈ P[X] and p = (∑ b∈B. q b * b)
  unfolding rng by (rule in-idealE-homomorphism-finite) blast
thus ?thesis ..
qed

corollary in-idealE-Polys:
  assumes B ⊆ P[X] and p ∈ P[X] and p ∈ ideal B
  obtains A q where finite A and A ⊆ B and ∩ b. q b ∈ P[X] and p = (∑ b∈A. q b * b)
proof –
  from assms(3) obtain A where finite A and A ⊆ B and p ∈ ideal A
  by (rule ideal.span-finite-subset)
from this(2) assms(1) have A ⊆ P[X] by (rule subset-trans)
with finite A: obtain q where ∩ b. q b ∈ P[X] and p = (∑ b∈A. q b * b)
  using assms(2): p ∈ ideal A: by (rule in-idealE-Polys-finite) blast
with finite A: A ⊆ B: show ?thesis ..
qed

lemma ideal-induct-Polys [consumes 3, case-names 0 plus]:
  assumes F ⊆ P[X] and p ∈ P[X] and p ∈ ideal F
  assumes P 0 and ∩ c q h. c ∈ P[X] ⇒ q ∈ F ⇒ P h ⇒ h ∈ P[X] ⇒ P (c * q + h)
  shows P (p::(λx.⇒_0 nat)) ⇒_0 'a::comm-ring-1)
proof –
  obtain h where ∩ p q. h (p + q) = h p + h q and ∩ p q. h (p * q) = h p * h q
  and ∩ p::(λx.⇒_0 nat) ⇒_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

461
assume $f \in \text{ideal } F$
thus $h \in \text{ideal } (\text{poly-subst } g \cdot F)$ unfolding $h$
proof (induct $f$ rule: ideal.span-induct-alt)
case base
show $?\text{case by (simp add: ideal.span-zero)}$
next
case (step $c \cdot f \cdot h$)
from step.hyps(1) have \( \text{poly-subst } g \cdot f \in \text{ideal } (\text{poly-subst } g \cdot F) \)
by (intro ideal.span-base imageI)
\hence \text{poly-subst } g \cdot c \cdot \text{poly-subst } g \cdot f \in \text{ideal } (\text{poly-subst } g \cdot F) \text{ by (rule ideal.span-scale)}
\hence \text{poly-subst } g \cdot c \cdot \text{poly-subst } g \cdot f + \text{poly-subst } g \cdot h \in \text{ideal } (\text{poly-subst } g \cdot F) \text{ by (rule ideal.span-add)}
thus $?\text{case by (simp only: poly-subst-plus poly-subst-times)}$
qed

17.4 Evaluating Polynomials

lemma lookup-times-zero:
lookup $(p \cdot q) \cdot 0 = \text{lookup } p \cdot 0 \cdot \text{lookup } q \cdot (0::\{\text{comm-powerprod,\text{ninв-comm-monoid-add}}\})$
proof
have eq: \( \sum_{v \in \text{keys } q} \cdot \text{lookup } q \cdot v \cdot \text{when } t + v = 0 \) = \( \text{lookup } q \cdot 0 \cdot \text{when } t = 0 \)
for $t$
proof
have \( \sum_{v \in \text{keys } q} \cdot \text{lookup } q \cdot v \cdot \text{when } t + v = 0 \) = \( \sum_{v \in \text{keys } q \cap \{0\}} \cdot \text{lookup } q \cdot v \cdot \text{when } t + v = 0 \)
proof (intro sum.mono-neutral-right ballI)
fix $v$
assume $v \in \text{keys } q - \text{keys } q \cap \{0\}$
\hence $v \neq 0$ by blast
\hence $t + v \neq 0$ using plus-eq-zero-2 by blast
thus \( \text{lookup } q \cdot v \cdot \text{when } t + v = 0 \) = \( 0 \) by simp
qed simp-all
also have \( \ldots = (\text{lookup } q \cdot 0 \cdot \text{when } t = 0) \) by (cases $0 \in \text{keys } q$) (simp-all add: in-keys-iff)
finally show $?\text{thesis }$
qed

have \( \sum_{t \in \text{keys } p} \cdot \text{lookup } p \cdot t \cdot \text{lookup } q \cdot 0 \cdot \text{when } t = 0 \) = \( \sum_{t \in \text{keys } p \cap \{0\}} \cdot \text{lookup } p \cdot t \cdot \text{lookup } q \cdot 0 \cdot \text{when } t = 0 \)
proof (intro sum.mono-neutral-right ballI)
fix $t$
assume $t \in \text{keys } p - \text{keys } p \cap \{0\}$
\hence $t \neq 0$ by blast
thus \( \text{lookup } p \cdot t \cdot \text{lookup } q \cdot 0 \cdot \text{when } t = 0 \) = \( 0 \) by simp
qed simp-all
also have \( \ldots = \text{lookup } p \cdot 0 \cdot \text{lookup } q \cdot 0 \) by (cases $0 \in \text{keys } p$) (simp-all add: in-keys-iff)
finally show $?\text{thesis by (simp add: lookup-times eq when-distrib)}$
462
corollary lookup-prod-zero:
  \( \text{lookup} (\prod f i) 0 = (\prod_{i \in I} \text{lookup} (f i) (0 \ldots \{\text{comm-powerprod, \text{ninv-comm-monoid-add}\}))) \)
  by (induct I rule: infinite-finite-induct) (simp all: lookup-times-zero)

corollary lookup-power-zero:
  \( \text{lookup} (p^k) 0 = \text{lookup} p (0 :\ldots\{\text{comm-powerprod, \text{ninv-comm-monoid-add}\}}) ^ k \)
  by (induct k) (simp all: lookup-times-zero)

definition poly-eval :: ('x => 'a) => (('x => nat) => 'a) => 'a ::\text{comm-semiring-1}
  where poly-eval a p = lookup (poly-subst (\lambda y. monomial (a y) (0 :\ldots\{\text{x \Rightarrow nat}\})) p) 0

lemma poly-eval-alt: poly-eval a p = (\sum t \in \text{keys} p. \text{lookup} p t * (\prod x \in \text{keys} t. a x ^ \text{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 \in \text{keys} t. a x ^ \text{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 :\ldots\{\text{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 :\ldots\{\text{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
    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 · 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 inductive-power-subset)
    also have ... = (∪ x∈keys t. indets (a x)) 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 \ldots = \bigcup (\text{indets} ' a ' \text{indets} (\text{monomial} c t + p)) \cup \\
\bigcup (\text{indets} ' \text{lookup} (\text{monomial} c t + p) ' \text{keys} (\text{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: \text{poly-eval} a ' \text{ideal} F = \text{ideal} (\text{poly-eval} a ' F)

proof (intro image-ideal-eq-surj poly-eval-plus poly-eval-times surjI)

fix x

show \text{poly-eval} a (\text{monomial} x 0) = x by (simp add: poly-eval-monomial)

qed

17.5 Replacing Indeterminates

definition map-indets where map-indets f = poly-subst (\lambda x. \text{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 (\neg r) = \neg map-indets f (r:::-\Rightarrow 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) = (\sum a \in A. map-indets f (g a))

and map-indets-prod: map-indets f (\prod g A) = (\prod a \in A. map-indets f (g a))


lemma map-indets-monomial:

map-indets f (\text{monomial} c t) = \text{monomial} c (\sum x \in \text{keys} t. \text{Poly-Mapping.single} (f x) (\text{lookup} t x))


lemma map-indets-id: (\forall x. x \in \text{indets} p \Rightarrow f x = x) \Rightarrow 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 \circ g) p

by (simp add: map-indets-def poly-subst-poly-subst poly-subst-monomial subst-pp-single)

lemma map-indets-cong: p = q \Rightarrow (\forall x. x \in \text{indets} q \Rightarrow f x = g x) \Rightarrow
map-indets \( f \) \( p \) = map-indets \( g \) \( q \)

unfolding map-indets-def by (simp cong: poly-subst-cong)

**Lemma** poly-subst-map-indets: \( \text{poly-subst} f (\text{map-indets} g \ p) \) = \( \text{poly-subst} (f \circ g) \) \( p \)

by (simp add: map-indets-def poly-subst-poly-subst poly-subst-monomial subst-pp-single comp-def)

**Lemma** poly-eval-map-indets: \( \text{poly-eval} a (\text{map-indets} g \ p) \) = \( \text{poly-eval} (a \circ g) \) \( p \)

by (simp add: poly-eval-def poly-subst-map-indets comp-def)


**Lemma** map-indets-inverseE-Polys:

assumes inj-on \( f \) \( X \) and \( p \) \( \in \) \( \mathbb{P}[X] \)

shows \( \text{map-indets} (\text{the-inv-into} X f) (\text{map-indets} f \ p) \) = \( p \)

unfolding map-indets-map-indets

proof (rule map-indets-id)

fix \( x \)

assume \( x \) \( \in \) indets \( p \)

also from assms(2) have \( \ldots \) \( \subseteq \) \( X \) by (rule PolysD)

finally show \( (\text{the-inv-into} X f \circ f) \) \( x \) = \( x \) using assms(1) by (auto intro: the-inv-into-f-f)

qed

**Lemma** map-indets-inverseE:

assumes \( \text{inj} \ f \)

obtains \( g \) where \( g = \text{the-inv} f \) and \( g \circ f = \text{id} \) and \( \text{map-indets} g \circ \text{map-indets} f = \text{id} \)

proof

define \( g \) where \( g = \text{the-inv} f \)

moreover from assms have eq: \( g \circ f = \text{id} \) by (auto intro!: ext the-inv-f-f simp: g-def)

moreover have \( \text{map-indets} g \circ \text{map-indets} f = \text{id} \)

by (rule ext) (simp add: map-indets-map-indets eq map-indets-id)

ultimately show ?thesis ..

qed

**Lemma** indets-map-indets-subset: indets \( (\text{map-indets} f (p::\Rightarrow \ a::\text{comm-semiring-1})) \)

\( \subseteq \) \( f \) \( \cdot \) indets \( p \)

proof

fix \( x \)

assume \( x \in \) indets \( (\text{map-indets} f \ p) \)

then obtain \( y \) where \( y \in \) indets \( p \) and \( x \in \) indets \( (\text{monomial} (1::\text{'a}) (\text{Poly-Mapping.single} (f \ y) \ 1)) \)

unfolding map-indets-def by (rule in-seq-substE)

from this(2) have \( x: \) \( x = f \ y \) by (simp add: indets-monomial)

from \( y \in \) indets \( p \) show \( x \in f \) \( \cdot \) indets \( p \) unfolding \( x \) by (rule imageI)

qed

466
corollary map-indets-in-Polys: map-indets \( f \) \( p \in P[f \mapsto \text{indets } p] \)
using \( \text{indets-map-indets-subset} \) by (rule PolysI-alt)

lemma \( \text{indets-map-indets} \):
assumes \( \text{inj-on } f \) \( (\text{indets } p) \)
shows \( \text{indets } (\text{map-indets } f \ p) = f \mapsto \text{indets } p \)
using \( \text{indets-map-indets-subset} \)

proof (rule subset-antisym)
let \( ?g = \text{the-inv-into } (\text{indets } p) \) \( f \)
have \( p = \text{map-indets } ?g (\text{map-indets } f \ p) \) unfolding \( \text{map-indets-map-indets} \)
by (rule sym, rule map-indets-id) (simp add: \( \text{assms the-inv-into-f-f} \))
also have \( \text{... } \subseteq ?g \mapsto \text{indets } (\text{map-indets } f \ p) \) by (rule image mono)
also have \( \text{... } = (\lambda x. x) \mapsto \text{indets } (\text{map-indets } f \ p) \) unfolding \( \text{image-image} \) using refl
proof (rule image cong)
fix \( x \)
assume \( x \in \text{indets } (\text{map-indets } f \ p) \)
with \( \text{indets-map-indets-subset} \) have \( x \in f \mapsto \text{indets } p \) ..
with \( \text{assms} \) show \( f (?g x) = x \) by (rule \( f \)-the-inv-into-f)
qed
finally show \( f \mapsto \text{indets } p \subseteq \text{indets } (\text{map-indets } f \ p) \) by simp
qed

lemma image-map-indets-Polys: \( \text{map-indets } f \ X \mapsto P[X] = \{ P[f \mapsto X] :: (\Rightarrow 0' a :: \text{comm-semiring-1}) \} \)

proof (intro set-eqI iffI)
fix \( p :: - \Rightarrow 0' a \)
assume \( p \in \text{map-indets } f \mapsto P[X] \)
then obtain \( q \) where \( q \in P[X] \) and \( p = \text{map-indets } f \ q \) ..
note \( \text{this(2)} \)
also have \( \text{map-indets } f \ q \in P[f \mapsto P[X]] \) by (fact \( \text{map-indets-in-Polys} \))
also from \( \langle p \in \rangle \) have \( \text{... } \subseteq P[f \mapsto X] \) by (auto intro!: Polys mono imageI dest: PolysD)
finally show \( p \in P[f \mapsto X] \).

next
fix \( p :: - \Rightarrow 0' a \)
assume \( p \in P[f \mapsto X] \)
define \( g \) where \( g = (\lambda y. \text{SOME } x. x \in X \land f x = y) \)
have \( g \ y \in X \land f (g \ y) = y \) if \( \ y \in \text{indets } p \) for \( \ y \)

proof
note that
also from \( \Rightarrow p \in \) have \( \text{indets } p \subseteq f \mapsto X \) by (rule PolysD)
finally obtain \( x \) where \( x \in X \) and \( y = f x \) ..
hence \( x \in X \land f x = y \) by simp
thus \( ?\text{thesis} \) unfolding \( g-def \) by (rule \( \text{someI} \))
qed

hence \( 1: g \ y \in X \) and \( 2: f (g \ y) = y \) if \( \ y \in \text{indets } p \) for \( \ y \) using \( \text{that by} \)
simp-all

show \( p \in \text{map-indets} f \cdot P[X] \)
proof
  show \( p = \text{map-indets} f \cdot (\text{map-indets} g \cdot p) \)
  by (rule sym) (simp add: map-indets-map-indets map-indets-id 2)
next
  have \( \text{map-indets} g \cdot p \in P[g \cdot \text{indets} p] \) by (fact map-indets-in-Polys)
  also have \( \ldots \subseteq P[X] \) by (auto intro: Polys-mono 1)
finally show \( \text{map-indets} g \cdot p \in P[X] \).
qed

qed

corollary range-map-indets: \( \text{range} \cdot (\text{map-indets} f) = P[\text{range} \cdot f] \)
proof
  have \( \text{range} \cdot (\text{map-indets} f) = \text{map-indets} f \cdot P[UNIV] \) by simp
  also have \( \ldots = P[\text{range} \cdot f] \) by (simp only: image-map-indets-Polys)
finally show \( \text{thesis} \).
qed

lemma in-keys-map-indetsE:
  assumes \( t \in \text{keys} \cdot (\text{map-indets} f \cdot (p::\Rightarrow a::\text{comm-semiring-1})) \)
  obtains \( s \) where \( s \in \text{keys} \cdot p \) and \( t = (\sum x \in \text{keys} \cdot s. \text{Poly-Mapping.single} \cdot (f \cdot x)) \)
  (lookup \( s \cdot x \) )
proof
  let \( \mathcal{F} = (\lambda x. \text{monomial} \cdot (1::'a)) \cdot (\text{Poly-Mapping.single} \cdot (f \cdot x) \cdot 1) \)
  from assms obtain \( s \) where \( s \in \text{keys} \cdot p \) and \( t \in \text{keys} \cdot (\text{subst-pp} \cdot ?f \cdot s) \) unfolding map-indets-def
  by (rule in-keys-poly-substE)
  note this(2)
  also have \( \ldots \subseteq \{ \sum x \in \text{keys} \cdot s. \text{Poly-Mapping.single} \cdot (f \cdot x) \cdot (\text{lookup} \cdot s \cdot x) \} \)
  by (simp add: subst-pp-def monomial-power-map-scale flip: punit. monomial-prod-sum)
finally have \( t = (\sum x \in \text{keys} \cdot s. \text{Poly-Mapping.single} \cdot (f \cdot x) \cdot (\text{lookup} \cdot s \cdot x)) \) by simp
with \( (s \in \text{keys} \cdot p) \) show \( \text{thesis} \) ..
qed

lemma keys-map-indets-subset:
  \( \text{keys} \cdot (\text{map-indets} f \cdot p) \subseteq (\lambda t. \sum x \in \text{keys} \cdot t. \text{Poly-Mapping.single} \cdot (f \cdot x) \cdot (\text{lookup} \cdot t \cdot x)) \cdot \text{keys} \cdot p \)
by (auto elim: in-keys-map-indetsE)

lemma keys-map-indets:
  assumes inj-on \( f \) \( (\text{indets} \cdot p) \)
  shows \( \text{keys} \cdot (\text{map-indets} f \cdot p) = (\lambda t. \sum x \in \text{keys} \cdot t. \text{Poly-Mapping.single} \cdot (f \cdot x) \cdot (\text{lookup} \cdot t \cdot x)) \cdot \text{keys} \cdot p \)
using keys-map-indets-subset
proof (rule subset-antisym)
  let \( \mathcal{G} = \text{the-inv-into} \cdot (\text{indets} \cdot p) \cdot f \)
  have \( p = \text{map-indets} \cdot ?g \cdot (\text{map-indets} f \cdot p) \) unfolding map-indets-map-indets
  by (rule sym, rule map-indets-id) (simp add: assms the-inv-into-f-f)
also have keys \ldots \subseteq (\lambda t. \sum x \in \text{keys } t. \text{monomial } (\text{lookup } t x) \ (\text{?g } x)) \ ' \text{keys } (\text{map-indets } f p) \\
by (\text{rule keys-map-indets-subset}) 
finally have (\lambda t. \sum x \in \text{keys } t. \text{Poly-Mapping.single } (f x) \ (\text{lookup } t x)) \ ' \text{keys } (\text{map-indets } f p) \\
by (\text{rule image-monono}) 
also from refl have \ldots = (\lambda t. \sum x. \text{Poly-Mapping.single } (f x) \ (\text{lookup } t x)) \ ' \text{keys } (\text{map-indets } f p) \\
by (\text{rule image-cong}) 
(smt \text{Sum-any,conditionalize } \text{Sum-any.cong } \text{finite-keys not-in-keys-iff-lookup-eq-zero single-zero}) 
also have \ldots = (\lambda t) \ ' \text{keys } (\text{map-indets } f p) \text{ unfolding } \text{image-image} \text{ using} \text{refl} 
proof (\text{rule image-cong}) 
  fix t 
  assume t \in \text{keys } (\text{map-indets } f p) 
  have (\sum x \cdot \text{monomial } (\text{lookup } (\sum y \in \text{keys } t. \text{Poly-Mapping.single } (\text{?g } y) \ (\text{lookup } t y)) \ x) \ (f x)) = 
    (\sum x. \sum y \in \text{keys } t. \text{monomial } (\text{lookup } t y \text{ when } ?g y = x) \ (f x)) 
  by (\text{simp add: lookup-sum lookup-single monomial-sum}) 
  also have \ldots = (\sum x \in \text{indets } p. \sum y \in \text{keys } t. \text{Poly-Mapping.single } (f x) \ (\text{lookup } t y \text{ when } ?y y = x)) 
  proof (\text{intro } \text{Sum-any.expand-superset } \text{finite-indets subsetI}) 
    fix x 
    assume x \in \{ a. (\sum y \in \text{keys } t. \text{Poly-Mapping.single } (f a) \ (\text{lookup } t y \text{ when } ?g y = a)) \neq 0 \} 
    hence (\sum y \in \text{keys } t. \text{Poly-Mapping.single } (f x) \ (\text{lookup } t y \text{ when } ?g y = x)) 
    \neq 0 \text{ by simp} 
    then obtain y where y \in \text{keys } t \text{ and } *: \text{Poly-Mapping.single } (f x) \ (\text{lookup } t y \text{ when } ?g y = x) \neq 0 
    \text{ by (rule sum:not-neutral-contains-not-neutral)} 
    from this(1) have y \in \text{indets } (\text{map-indets } f p) \text{ using } (t \in \cdot) \text{ by (rule in-indetsI1)} 
    with \text{indets-map-indets-subset } \text{have } y \in f \ ' \text{indets } p \ldots 
    from * \text{ have } x = ?y y \text{ by (simp add: when-def split: if-split-asm)} 
    also from \text{assms } y \in f \ ' \text{indets } p \text{ subset-refl } \text{have } \ldots \in \text{indets } p \text{ by (rule the-inv-into-into)} 
    finally show x \in \text{indets } p \ldots 
  qed 
  also have \ldots = (\sum y \in \text{keys } t. \sum x \in \text{indets } p. \text{Poly-Mapping.single } (f x) \ (\text{lookup } t y \text{ when } ?g y = x)) 
  \text{ by (fact sum.swap)} 
  also from refl \text{ have } \ldots = (\sum y \in \text{keys } t. \text{Poly-Mapping.single } y \ (\text{lookup } t y)) 
  proof (\text{rule sum.cong}) 
    fix x 

469
assume \( x \in \text{keys } t \)
hence \( x \in \text{indets } (\text{map-indets } f \ p) \) using \( t \in \cdot \) by (rule in-indetsI)
with \( \text{indets-map-indets-subset } \) have \( x \in f ^ \cdot \text{indets } p \) ..
with \( \text{assms have } ?g \ x \in \text{indets } p \) using \( \text{subset-refl } \) by (rule the-inv-into-into)
hence \( \{ ?g \ x \} \subseteq \text{indets } p \) by simp
with \( \text{finite-indets } \) have \( \sum y \in \text{indets } p \). \( \text{Poly-Mapping.single } (f y) \) (lookup \( t \ x \) when \( ?g \ x = y \)) = 
\( \sum y \in \{ ?g \ x \}. \text{Poly-Mapping.single } (f y) \) (lookup \( t \ x \) when \( ?g \ x = y \))
by (rule sum.monono-neutral-right) (simp add: monomial-0-iff when-def)
also from \( \text{assms } x \in f ^ \cdot \text{indets } p \) have \( \ldots = \text{Poly-Mapping.single } x \) (lookup \( t \ x \))
by (simp add: f-the-inv-into-f)
finally show \( \sum y \in \text{indets } p \). \( \text{Poly-Mapping.single } (f y) \) (lookup \( t \ x \) when \( ?g \ x = y \)) = 
\( \text{Poly-Mapping.single } x \) (lookup \( t \ x \)).

qed
also have \( \ldots = t \) by (fact poly-mapping-sum-monomials)
finally show \( \sum x \in \text{keys } s \). \( \text{Poly-Mapping.single } (f x) \) (lookup \( t \ x \)) = \( t \)
by simp add: t deg-pm-sum deg-pm-single deg-pm-superset[of subset-refl])

qed

lemma poly-deg-map-indets-le: \( \text{poly-deg } (\text{map-indets } f \ p) \leq \text{poly-deg } p \)
proof (rule poly-deg-leI)
fix \( t \)
assume \( t \in \text{keys } (\text{map-indets } f \ p) \)
then obtain \( s \) where \( s \in \text{keys } p \) and \( t : t = (\sum x \in \text{keys } s \). \( \text{Poly-Mapping.single } (f x) \) (lookup \( s \ x \))
by (rule in-keys-map-indetsE)
from this(1) have \( \text{deg-pm } s \leq \text{poly-deg } p \) by (rule poly-deg-max-keys)
thus \( \text{deg-pm } t \leq \text{poly-deg } p \)
by simp add: t deg-pm-sum deg-pm-single deg-pm-superset[of subset-refl]

qed

lemma poly-deg-map-indets:
assumes \( \text{inj-on } f \) (indets \( p \))
shows \( \text{poly-deg } (\text{map-indets } f \ p) = \text{poly-deg } p \)
proof –
from \( \text{assms have } \ldots = \text{deg-pm } \cdot \text{keys } (\text{map-indets } f \ p) = \text{deg-pm } \cdot \text{keys } p \)
by simp add: keys-map-indets image-image deg-pm-sum deg-pm-single
flip: deg-pm-superset[of subset-refl]
thus \( \cdot \)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)::- => - =>0 'a::comm-semiring-1) P[X]
proof (rule inj-onI)
  fix p q :: - =>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::(- =>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 = (∑f'∈F0. q f' * f')
    by (rule ideal.spanE)
  from this(1) obtain F' where F' ⊆ F and F0 = map-indets f ' F' by (rule subset-imageE)
  from assms obtain g where map-indets g o map-indets f = (id::- => - =>0 'a)
    by (rule map-indets-inverseE)
  hence eq: map-indets g (map-indets f p) = p' for p':- =>0 'a

471
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 (∑ 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 (∑ f'∈F'. map-indets g (q (map-indets f f')) * f')
  by (simp add: pointfree-idE)

17.6 Homogeneity

definition homogeneous :: (('x ⇒₀ nat) ⇒₀ 'a::zero) ⇒ bool
  where homogeneous p ←→ (∀ s∈keys p. ∀ t∈keys p. deg-pm s = deg-pm t)

definition hom-component :: (('x ⇒₀ nat) ⇒₀ 'a) ⇒ nat ⇒ (('x ⇒₀ nat) ⇒₀ 'a::zero)
  where hom-component p n = except p { t. deg-pm t ≠ n}

definition hom-components :: (('x ⇒₀ nat) ⇒₀ 'a) ⇒ (('x ⇒₀ nat) ⇒₀ 'a::zero)
  set
  where hom-components p = hom-component p ' deg-pm ' keys p

definition homogeneous-set :: (('x ⇒₀ nat) ⇒₀ 'a::zero) set ⇒ bool
  where homogeneous-set A ←→ (∀ a∈A. ∀ n. hom-component a n ∈ A)

lemma homogeneousI: (∀ s t. s ∈ keys p ∈ 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 \( \text{deg-pm } t \leq \text{poly-deg } p \) by (rule poly-deg-max-keys)

next
show \( \text{poly-deg } p \leq \text{deg-pm } t \)
proof (rule poly-deg-leI)
  fix \( s \)
  assume \( s \in \text{keys } p \)
  with assms(1) have \( \text{deg-pm } s = \text{deg-pm } t \) using assms(2) by (rule homogeneousD)
  thus \( \text{deg-pm } s \leq \text{deg-pm } t \) 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 \( (\neg p) \) \( \iff \) homogeneous \( p \)
by (auto intro!: homogeneousI dest!: homogeneousD simp!: keys-uminus)

lemma homogeneous-monom-mult: homogeneous \( p \) \( \Rightarrow \) homogeneous \( (\text{punit.monom-mult } c \ t \ p) \)
by (auto intro!: homogeneousI elim!: punit.keys-monom-multE simp!: deg-pm-plus dest!: homogeneousD)

lemma homogeneous-monom-mult-rev:
  assumes \( c \neq (0::'a::semiring-no-zero-divisors) \) and homogeneous \( (\text{punit.monom-mult } c \ t \ p) \)
  shows homogeneous \( p \)
proof (rule homogeneousI)
  fix \( s \) \( s' \)
  assume \( s \in \text{keys } p \)
  hence \( 1: t + s \in \text{keys } (\text{punit.monom-mult } c \ t \ p) \)
  using assms(1) by (rule punit.keys-monom-multI[simplified])
  assume \( s' \in \text{keys } p \)
  hence \( t + s' \in \text{keys } (\text{punit.monom-mult } c \ t \ p) \)
  using assms(1) by (rule punit.keys-monom-multI[simplified])
  with assms(2) \( I \) have \( \text{deg-pm } (t + s) = \text{deg-pm } (t + s') \) by (rule homogeneousD)
  thus \( \text{deg-pm } s = \text{deg-pm } s' \) by (simp add: deg-pm-plus)
qed

lemma homogeneous-times:
  assumes homogeneous \( p \) and homogeneous \( q \)
  shows homogeneous \( (p * q) \)
proof (rule homogeneousI)
  fix \( s \) \( t \)

473
assume \( s \in \text{keys } (p * q) \)
then obtain \( sp \ sq \) where \( sp: sp \in \text{keys } p \) and \( sq: sq \in \text{keys } q \) and \( s: s = sp + sq \)

by (rule \text{in-keys-timesE})

assume \( t \in \text{keys } (p * q) \)
then obtain \( tp \ tq \) where \( tp: tp \in \text{keys } p \) and \( tq: tq \in \text{keys } q \) and \( t: t = tp + tq \)

by (rule \text{in-keys-timesE})

from assms(1) \( sp \ tp \) have \( \text{deg-pm } sp = \text{deg-pm } tp \) by (rule \text{homogeneousD})
moreover from assms(2) \( sq \ tq \) have \( \text{deg-pm } sq = \text{deg-pm } tq \) by (rule \text{homogeneousD})
ultimately show \( \text{deg-pm } s = \text{deg-pm } t \) by (simp only: \( s \ t \ \text{deg-pm+} \))
qed

lemma \text{lookup-hom-component}: \( \text{lookup } (\text{hom-component } p \ n) = (\lambda t. \text{lookup } p t \text{ when } \text{deg-pm } t = n) \)

by (rule \text{ext}) (simp add: \text{hom-component-def lookup-except})

lemma \text{keys-hom-component}: \( \text{keys } (\text{hom-component } p \ n) = \{ t. \ t \in \text{keys } p \wedge \text{deg-pm } t = n \}) \)

by (auto simp: \text{hom-component-def keys-except})

lemma \text{keys-hom-componentD}: 
assumes \( t \in \text{keys } (\text{hom-component } p \ n) \)
shows \( t \in \text{keys } p \) and \( \text{deg-pm } t = n \)
using assms by (simp-all add: \text{keys-hom-component})

lemma \text{homogeneous-hom-component}: \( \text{homogeneous } (\text{hom-component } p \ n) \)
by (auto dest: \text{keys-hom-componentD} intro: \text{homogeneousI})

lemma \text{hom-component-zero [simp]}: \( \text{hom-component } 0 = 0 \)
by (rule \text{ext}) (simp add: \text{hom-component-def})

lemma \text{hom-component-zero-iff}: \( \text{hom-component } p \ n = 0 \iff (\forall t\in \text{keys } p. \text{deg-pm } t \neq n) \)
by (metis (mono-tags, lifting) \text{empty-iff keys-eq-empty-iff keys-hom-component mem-Collect-eq subsetI subset-antisym})

lemma \text{hom-component-uminus [simp]}: \( \text{hom-component } (-p) = -\text{hom-component } p \)
by (intro ext poly-mapping-eqI) (simp add: \text{hom-component-def lookup-except})

lemma \text{hom-component-plus}: \( \text{hom-component } (p + q) \ n = \text{hom-component } p \ n + \text{hom-component } q \ n \)
by (rule poly-mapping-eqI) (simp add: \text{hom-component-def lookup-except lookup-add})

lemma \text{hom-component-minus}: \( \text{hom-component } (p - q) \ n = \text{hom-component } p \ n - \text{hom-component } q \ n \)
by (rule poly-mapping-eqI) (simp add: \text{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
    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 = {} \iff 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) \imp t \in keys p \imp q \in 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 \imp 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 \( p \neq \{\} \) by simp 
then obtain \( t \) where \( t \in \text{keys } p \) by blast
with assms have \( \text{deg-pm } t = \text{poly-deg } p \) by (rule \text{homogeneousD-poly-deg}) 
\hence \( n = \text{deg-pm } t \) by (simp only: \( n = \text{poly-deg } p \)) 
with \( t \in \text{keys } p \) show \( n \in \text{deg-pm } ' \text{keys } p \) by (rule \text{rev-image-eqI}) 
\thus \( n \in \text{deg-pm } ' \text{keys } p \) by simp
qed

lemma \text{finite-hom-components}: finite \((\text{hom-components } p)\) unfolding \text{hom-components-def} using \text{finite-keys} by (intro \text{finite-imageI})

lemma \text{hom-components-homogeneous}: \( q \in \text{hom-components } p \imp \text{homogeneous } q \) by (elim \text{hom-componentsE}) (simp only: \text{homogeneous-hom-component})

lemma \text{hom-components-nonzero}: \( q \in \text{hom-components } p \imp q \neq 0 \) by (auto elim!: \text{hom-componentsE} simp: \text{hom-component-zero-iff})

lemma \text{deg-pm-hom-components}: assumes \( q1 \in \text{hom-components } p \) and \( q2 \in \text{hom-components } p \) and \( t1 \in \text{keys } q1 \) and \( t2 \in \text{keys } q2 \) shows \( \text{deg-pm } t1 = \text{deg-pm } t2 \imp q1 = q2 \) 
proof –
from assms(1) obtain \( s1 \) where \( s1 \in \text{keys } p \) and \( q1: q1 = \text{hom-component } p \) (deg-pm \( s1 \))
by (rule \text{hom-componentsE})
from assms(3) have \( t1: \text{deg-pm } t1 = \text{deg-pm } s1 \) unfolding \( q1 \) by (rule \text{keys-hom-componentD})
from assms(2) obtain \( s2 \) where \( s2 \in \text{keys } p \) and \( q2: q2 = \text{hom-component } p \) (deg-pm \( s2 \))
by (rule \text{hom-componentsE})
from assms(4) have \( t2: \text{deg-pm } t2 = \text{deg-pm } s2 \) unfolding \( q2 \) by (rule \text{keys-hom-componentD})
from \( s1 \in \text{keys } p \) show \( \text{thesis} \) by (auto simp: \( q1 \) \( q2 \) \( t1 \) \( t2 \) dest: \text{hom-component-inject})
qed

lemma \text{poly-deg-hom-components}: assumes \( q1 \in \text{hom-components } p \) and \( q2 \in \text{hom-components } p \) shows \( \text{poly-deg } q1 = \text{poly-deg } q2 \imp q1 = q2 \) 
proof –
from assms(1) have \( \text{homogeneous } q1 \) and \( q1 \neq 0 \)
by (rule \text{hom-components-homogeneous}, rule \text{hom-components-nonzero})
from this(2) have keys \( q1 \neq \{\} \) by simp
then obtain \( t1 \) where \( t1 \in \text{keys } q1 \) by blast
with \( \text{homogeneous } q1 \) have \( t1: \text{deg-pm } t1 = \text{poly-deg } q1 \) by (rule \text{homogeneousD-poly-deg})
from assms(2) have \( \text{homogeneous } q2 \) and \( q2 \neq 0 \)
by (rule \text{hom-components-homogeneous}, rule \text{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 asms ⟨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 asms(1, 2) have deg-pm t = deg-pm t ↔ q1 = q2 by (rule deg-pm-hom-components)
with asms(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 ∈ hom-components p
  shows poly-deg q ≤ poly-deg p
proof (rule poly-deg-leI)
  fix t
  assume t ∈ keys q
also from asms have ... ⊆ Keys (hom-components p) by (rule keys-subset-Keys)
also have ... = keys p by (fact Keys-hom-components)
finally show deg-pm t ≤ poly-deg p by (rule poly-deg-max-keys)
qed

lemma sum-hom-components: ∑ (hom-components p) = p
proof (rule poly-mapping-eqI)
  fix t
  show lookup (∑ (hom-components p)) t = lookup p t unfolding lookup-sum
proof (cases t ∈ keys p)
  case True
  also have keys p = Keys (hom-components p) by (simp only: Keys-hom-components)
  finally obtain q where q: q ∈ hom-components p and t: t ∈ keys q by (rule in-KeysE)
    from this(1) have (∑ q0 ∈ hom-components p. lookup q0 t) =
      (∑ q0 ∈ insert q (hom-components p). lookup q0 t)
    by (simp only: insert-absorb)
also from finite-hom-components have \( \ldots = \text{lookup } q \ t + (\sum_{q_0 \in \text{hom-components } p - \{q\}} \text{lookup } q_0 \ t) \) by (rule sum.insert-remove)
also from \( q \ t \) have \( \ldots = \text{lookup } p \ t + (\sum_{q_0 \in \text{hom-components } p - \{q\}} \text{lookup } q_0 \ t) \)
by (simp only: lookup-hom-components)
also have \( (\sum_{q_0 \in \text{hom-components } p - \{q\}} \text{lookup } q_0 \ t) = 0 \)
proof (intro sum.neutral ballI)
fix \( q_0 \)
assume \( q_0 \in \text{hom-components } p - \{q\} \)
hence \( q_0 \in \text{hom-components } p \) and \( q \neq q_0 \) by blast
with \( q \) have \( \text{keys } q \cap \text{keys } q_0 = \{\} \) by (rule hom-components-keys-disjoint)
with \( t \) have \( t \notin \text{keys } q_0 \) by blast
thus \( \text{lookup } q_0 \ t = 0 \) by (simp add: in-keys-iff)
qed
finally show \( (\sum_{q \in \text{hom-components } p} \text{lookup } q \ t) = \text{lookup } p \ t \)
by simp
next case False
hence \( t \notin \text{Keys } (\text{hom-components } p) \) by (simp add: Keys-hom-components)
hence \( \forall q \in \text{hom-components } p. \text{lookup } q \ t = 0 \) by (simp add: Keys-def in-keys-iff)
hence \( (\sum_{q \in \text{hom-components } p} \text{lookup } q \ t) = 0 \) by (rule sum.neutral)
also from False have \( \ldots = \text{lookup } p \ t \) by (simp add: in-keys-iff)
finally show \( (\sum_{q \in \text{hom-components } p} \text{lookup } q \ t) = \text{lookup } p \ t \).
qed

lemma homogeneous-setI: \( (\forall \ a. \ a \in A \implies \text{hom-component } a \ n \in A) \implies \text{homogeneous-set } A \)
by (simp add: homogeneous-set-def)

lemma homogeneous-setD: \( \text{homogeneous-set } A \implies a \in A \implies \text{hom-component } a \ n \in A \)
by (simp add: homogeneous-set-def)

lemma homogeneous-set-Polys: \( \text{homogeneous-set } (P[X]::(- \Rightarrow_0 'a::zero) \text{ set}) \)
proof (intro homogeneous-setI PolysI subsetI)
fix \( p::- \Rightarrow_0 'a \text{ and } n \ t \)
assume \( p \in P[X] \)
assume \( t \in \text{keys } (\text{hom-component } p \ n) \)
hence \( t \in \text{keys } p \) by (rule keys-hom-componentD)
also from \( \langle p \in P[X]\rangle \) have \( \ldots \subseteq [X] \) by (rule PolysD)
finally show \( t \in [X] \).
qed

lemma homogeneous-set-IntI: \( \text{homogeneous-set } A \implies \text{homogeneous-set } B \implies \text{homogeneous-set } (A \cap B) \)
by (simp add: homogeneous-set-def)

lemma homogeneous-setD-hom-components:
assumes homogeneous-set $A$ and $a \in A$ and $b \in$ hom-components $a$
s
shows $b \in A$

proof –

from assms(3) obtain $t::'a \Rightarrow _0 \text{nat}$ where $b = \text{hom-component} a\ (\text{deg-pm } t)$
  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 $\text{deg-pm } t = \text{Suc } (\text{poly-deg } a)$ for $t$
proof (rule ccontr)
  assume lookup $a \ t \neq 0$
  hence $t \in \text{keys } a$ by (simp add: in-keys-iff)
  hence $\text{deg-pm } t \leq \text{poly-deg } a$ by (rule poly-deg-max-keys)
  thus False by (simp add: that)

qed

hence $0 = \text{hom-component } a\ (\text{Suc } (\text{poly-deg } a))$
  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 $\\forall f. f \in F \Rightarrow \text{homogeneous } f$ and $p \in \text{ideal } F$

shows $\text{hom-component } p\ n \in \text{ideal } F$

proof –

from assms(2) have $p \in \text{punit.pmdl } F$ 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 $?f = \text{punit.monom-mult } c\ t\ f$
from module-plus.hyps(3) have $f \in \text{punit.pmdl } F$ by (simp add: ideal.span-base)
  hence $*: ?f \in \text{punit.pmdl } F$ 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 $\text{hom-component } ?f = (?f\ \text{when } n = \text{poly-deg } ?f)$ by (rule hom-component-of-homogeneous)
also from $*$ have \ldots $\in \text{ideal } F$ by (simp add: when-def ideal.span-zero)
finally have $\text{hom-component } ?f = (?f \in \text{ideal } F )$ .
with module-plus.hyps(2) show \?case unfolding hom-component-plus by (rule ideal.span-add)

qed

480
corollary homogeneous-set-homogeneous-ideal:
\[ (\forall f. f \in F \implies \text{homogeneous } f) \implies \text{homogeneous-set (ideal } F) \]
by (auto intro: homogeneous-setI homogeneous-ideal)

corollary homogeneous-ideal':
assumes \( \forall f. f \in F \implies \text{homogeneous } f \) and \( p \in \text{ideal } F \) and \( q \in \text{hom-components} \)
shows \( q \in \text{ideal } F \)
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 \( \forall f. f \in F \implies \text{homogeneous } f \) and \( p \in \text{ideal } F \) and \( \text{homogeneous } p \)
obtains \( F' \) \( q \) where finite \( F' \) and \( F' \subseteq F \) and \( p = (\sum f \in F'. \ q \ f \ f) \) and \( \forall f. f \in F' \implies \text{poly-deg } (q \ f \ f) = \text{poly-deg } p \) and \( \forall f. f \notin F' \implies q \ f \ = \ 0 \)
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 \( ?A = \lambda f. \{ h \in \text{hom-components } (q' \ f) \}. \text{poly-deg } h + \text{poly-deg } f = \text{poly-deg } p \)
  let \( ?B = \lambda f. \{ h \in \text{hom-components } (q' \ f) \}. \text{poly-deg } h + \text{poly-deg } f \neq \text{poly-deg } p \)
  define \( F' \) where \( F' = \{ f \in F''. (\sum (\emptyset A \ f)) \ast f \neq 0 \} \)
  define \( q \) where \( q = (\lambda f. (\sum (\emptyset A \ f)) \text{ when } f \in F') \)
  have \( F'' \subseteq F' \) by (simp add: \( F'\)-def)
  hence \( F'' \subseteq F \) using \( F'' \subseteq F \) by (rule subset-trans)
  have 1: \( \text{deg-pm } t + \text{poly-deg } f = \text{poly-deg } p \) if \( f \in F' \) and \( t \in \text{keys } (q \ f) \) for \( f \ t \)
  proof –
    from that have \( t \in \text{keys } (\sum (\emptyset A \ f)) \) by (simp add: \( q\)-def)
    also have \( \ldots \subseteq (\bigcup h \in ?A \ f. \text{keys } h) \) by (fact keys-sum-subset)
    finally obtain \( h \) where \( h \in ?A \ f \) and \( t \in \text{keys } h \) ..
    from this(1) have \( h \in \text{hom-components } (q' \ f) \) and \( eq: \text{poly-deg } h + \text{poly-deg } f = \text{poly-deg } p \)
    by simp-all
    from this(1) have \( \text{homogeneous } h \) by (rule hom-components-homogeneous)
    hence \( \text{deg-pm } t = \text{poly-deg } h \) using \( t \in \text{keys } h \) by (rule homogeneousD-poly-deg)
    thus \( \text{thesis } \) by (simp only: eq)
  qed
  have 2: \( \text{deg-pm } t = \text{poly-deg } p \) if \( f \in F' \) and \( t \in \text{keys } (q \ f \ f) \) for \( f \ t \)
  proof –
    from that(1) \( F' \subseteq F \) have \( f \in F \) ..
    hence \( \text{homogeneous } f \) by (rule assms(1))
  qed

481
from that(2) obtain \(s_1\) \(s_2\) where \(s_1 \in \text{keys } (q \, f)\) and \(s_2 \in \text{keys } f\) and \(t\): \(t = s_1 + s_2\)

by (rule in-keys-timesE)

from that(1) this(1) have \(\text{deg-pm } s_1 + \text{poly-deg } f = \text{poly-deg } p\) by (rule 1)

moreover from homogeneous f \(s_2 \in \text{keys } f\) have deg-pm \(s_2 = \text{poly-deg } f\)

by (rule homogeneousD-poly-deg)

ultimately show \(?\text{thesis}\) by (simp add: \(t\) \(\text{deg-pm-plus}\))

qed

from \(F' \subseteq F''\) (finite \(F''\)) have finite \(F'\) by (rule finite-subset)

thus \(?\text{thesis}\) using \((F' \subseteq F)\)

proof

note \(p\)

also from refl have \((\sum f \in F''\). \(q\, f \ast f\)) = \((\sum (\forall A\, f) \ast f) + (\sum (\forall B\, f) \ast f))\)

proof (rule sum.cong)

fix \(f\)

assume \(f \in F''\)

from sum-hom-components have \(q\, f = (\sum (\text{hom-components } (q\, f)))\) by (rule sym)

also have \(\ldots = (\sum (\forall A\, f \cup \forall B\, f))\) by (rule arg-cong[where \(f=\sum (\lambda x.\, x)]\)) blast

also have \(\ldots = \sum (\forall A\, f) + \sum (\forall B\, f)\)

proof (rule sum.union-disjoint)

have \(\forall A\, f \subseteq \text{hom-components } (q\, f)\) by blast

thus finite \((\forall A\, f)\) using finite-hom-components by (rule finite-subset)

next

have \(\forall B\, f \subseteq \text{hom-components } (q\, f)\) by blast

thus finite \((\forall B\, f)\) using finite-hom-components by (rule finite-subset)

qed blast

finally show \(q\, f \ast f = (\sum (\forall A\, f) \ast f) + (\sum (\forall B\, f) \ast f)\)

by (metis \(\text{no-types, lifting}\) distrib-right)

qed

also have \(\ldots = (\sum f \in F'\). (\sum (\forall A\, f) \ast f) + (\sum f \in F'\). (\sum (\forall B\, f) \ast f)\) by (rule sum.distrib)

also from \((\text{finite } F'\). F' \subseteq F''\) have \((\sum f \in F'\). \(\forall A\, f\) \ast f) = \((\sum f \in F'. f \ast f)\)

proof (intro sum.mono-neutral-cong-right ballII)

fix \(f\)

assume \(f \in F'' - F'\)

thus \(\sum (\forall A\, f) \ast f = 0\) by (simp add: \(F'\)-def)

next

fix \(f\)

assume \(f \in F'\)

thus \(\sum (\forall A\, f) \ast f = q\, f \ast f\) by (simp add: \(q\)-def)

qed

finally have \(p[\{\text{symmetric}\}]: \(p = (\sum f \in F'. q\, f \ast f) + (\sum f \in F''. \sum (\forall B\, f) \ast f)\)\)

moreover have \(\text{keys } (\sum f \in F''\). \(\sum (\forall B\, f) \ast f\) = \{\}\)

proof (rule, rule)

482
fix $t$

assume $t$-in: $t \in \text{keys} \left( \sum_{f \in F^'} f \cdot f \right)$
also have $\ldots \subseteq \left( \bigcup_{f \in F^'} \text{keys} \left( \sum \left( B \cdot f \right) \right) \right)$ by (fact keys-sum-subset)
finally obtain $f$ where $f \in F^'$ and $t \in \text{keys} \left( \sum \left( B \cdot f \right) \right)$ ..
from this(2) obtain $s I \ s 2$ where $s I \in \text{keys} \left( \sum \left( B \cdot f \right) \right)$ and $s 2 \in \text{keys} \ f$
and $t$: $t = s I + s 2$
by (rule in-keys-timesE)
from $f \in F^'' \ (F'' \subseteq F)$ have $f \in F$ ..
hence homogeneous $f$ by (rule assms(1))
note $s I \in \text{keys} \left( \sum \left( B \cdot f \right) \right)$
also have $\text{keys} \left( \sum \left( B \cdot f \right) \right) \subseteq \left( \bigcup h \in B \cdot f \cdot \text{keys} h \right)$ by (fact keys-sum-subset)
finally obtain $h$ where $h \in B \cdot f$ and $s I \in \text{keys} \ h$ ..
from this(1) have $h \in \text{hom-components} \ (q' \ f)$ and neg: poly-deg $h + \text{poly-deg} f \neq \text{poly-deg} p$
by simp-all
from this(1) have homogeneous $h$ by (rule hom-components-homogeneous)
hence deg-pm $s I = \text{poly-deg} h$ using $(s I \in \text{keys} h)$ by (rule homogeneousD-poly-deg)
moreover from homogeneous $f$ $(s 2 \in \text{keys} f)$ have deg-pm $s 2 = \text{poly-deg} f$
by (rule homogeneousD-poly-deg)
ultimately have deg-pm $t \neq \text{poly-deg} p$ using neg by (simp add: t deg-pm-plus)
have $t \notin \text{keys} \left( \sum f \in F' \ f \cdot f \right)$
proof
assume $t \in \text{keys} \left( \sum f \in F' \ f \cdot f \right)$
also have $\ldots \subseteq \left( \bigcup f \in F' \text{keys} \left( f \cdot f \right) \right)$ by (fact keys-sum-subset)
finally obtain $f$ where $f \in F'$ and $t \in \text{keys} \left( f \cdot f \right)$ ..
hence deg-pm $t = \text{poly-deg} p$ by (rule 2)
with $\text{deg-pm} \ t \neq \text{poly-deg} p$ show False ..
qed
with $t$-in have $t \in \text{keys} \left( \sum f \in F' \ f \cdot f \right) + \left( \sum f \in F'' \sum \left( B \cdot f \right) \cdot f \right)$
by (rule in-keys-plusI2)
hence $t \in \text{keys} p$ by (simp only: $p$)
with assms(3) have deg-pm $t = \text{poly-deg} p$ by (rule homogeneousD-poly-deg)
with deg-pm $t \neq \text{poly-deg} p$ show $t \in \{}$ ..
qed (fact empty-subsetI)
ultimately show $p = \left( \sum f \in F' \ f \cdot f \right)$ by simp
next
fix $f$
show homogeneous $(q \ f)$
proof (cases $f \in F'$)
case True
show $\ ? \text{thesis}$
proof (rule homogeneousI)
fix $s \ t$
assume $s \in \text{keys} \ (q \ f)$
with True have $s + \text{deg-pm} s + \text{poly-deg} f = \text{poly-deg} p$ by (rule 1)
assume $t \in \text{keys} \ (q \ f)$
with True have deg-pm $t + \text{poly-deg} f = \text{poly-deg} p$ by (rule 1)
with $s$ show deg-pm $s = \text{deg-pm} t$ by simp
qed

483
next
  case False
  thus \( ? \)thesis by (simp add: q-def)
qed

assume \( f \in F' \)
show \( \text{poly-deg} (q \cdot f) = \text{poly-deg} p \)
proof (intro antisym)
  show \( \text{poly-deg} (q \cdot f) \leq \text{poly-deg} p \)
  proof (rule poly-deg-leI)
    fix \( t \)
    assume \( t \in \text{keys} (q \cdot f) \)
    with \( f \in F' \) have \( \deg-pm t = \text{poly-deg} p \) by (rule 2)
    thus \( \deg-pm t \leq \text{poly-deg} p \) by simp
  qed
next
  from \( f \in F' \) have \( q \cdot f \neq 0 \) by (simp add: q-def F'-def)
  hence \( \text{keys} (q \cdot f) \neq \{\} \) by simp
  then obtain \( t \) where \( t \in \text{keys} (q \cdot f) \) by blast
  with \( f \in F' \) have \( \deg-pm t = \text{poly-deg} p \) by (rule 2)
  moreover from \( t \in \text{keys} (q \cdot f) \) have \( \deg-pm t \leq \text{poly-deg} (q \cdot f) \) by (rule poly-deg-max-keys)
  ultimately show \( \text{poly-deg} p \leq \text{poly-deg} (q \cdot f) \) by simp
qed
qed (simp add: q-def)

\textbf{corollary} homogeneous-idealE:
assumes \( \bigwedge f. f \in F \implies \text{homogeneous} f \) and \( p \in \text{ideal} F \)
obtains \( F' \) where \( \text{finite} F' \) and \( \text{F'} \subseteq F \) and \( p = (\sum f \in F'. q \cdot f) \)
and \( \bigwedge f. \text{poly-deg} (q \cdot f) \leq \text{poly-deg} p \) and \( \bigwedge f. f \notin F' \implies q \cdot f = 0 \)
proof (cases \( p = 0 \))
  case True
  show \( ? \)thesis
proof
  show \( p = (\sum f \in \{\}. (\lambda_. 0) f \cdot f) \) by (simp add: True)
qed simp-all
next
  case False
  define \( P \) where \( P = (\lambda h qf. \text{finite} (\text{fst} qf) \land \text{fst} qf \subseteq F \land h = (\sum f \in \text{fst} qf. \text{snd} qf f \cdot f) \land (\forall f \in \text{fst} qf. \text{poly-deg} (\text{snd} qf f \cdot f) = \text{poly-deg} h) \land (\forall f. f \notin \text{fst} qf \implies \text{snd} qf f = 0)) \)
  define \( q0 \) where \( q0 = (\lambda h. \text{SOME} qf. P h qf) \)
  have \( 1 : P h (q0 h) \) if \( h \in \text{hom-components} p \) for \( h \)
proof
  note \( \text{assms}(1) \)
  moreover from \( \text{assms} \) that \( h \in \text{ideal} F \) by (rule homogeneous-ideal')
  moreover from \( \text{that} \) have \( \text{homogeneous} h \) by (rule hom-components-homogeneous)

484
ultimately obtain $F' q$ where finite $F'$ and $F' \subseteq F$ and $h = (\sum f \in F'. q f$ * $f$) and $\bigwedge f \in F' \implies \text{poly-deg} (q f * f) = \text{poly-deg} h$ and $\bigwedge f \not\in F' \implies q f = 0$

by (rule homogeneous-idealE-homogeneous) blast+

hence $P h (F', q)$ by (simp add: P-def)

thus $\exists \text{thesis unfolding } q0-def$ by (rule someI)

qed

define $F'$ where $F' = (\bigcup h \in \text{hom-components} p. \text{fst} (q0 h))$

define $q$ where $q = (\lambda f. \sum h \in \text{hom-components} p. \text{snd} (q0 h) f)$

show $\exists \text{thesis}$

proof

have finite $F' \land F' \subseteq F$ unfolding $F'-def$ UN-subset-iff finite-UN[OF finite-hom-components]

proof (intro conjI ballI)

fix $h$

assume $h \in \text{hom-components} p$

hence $P h (q0 h)$ by (rule 1)

thus finite $\{\text{fst} (q0 h)\}$ and $\text{fst} (q0 h) \subseteq F$ by (simp-all only: P-def)

qed

thus finite $F'$ and $F' \subseteq F$ by simp-all

from sum-hom-components have $p = (\sum (\text{hom-components} p))$ by (rule sym)

also from refl have ... = $(\sum h \in \text{hom-components} p. \sum f \in F'. \text{snd} (q0 h) f * f$)

proof (rule sum.cong)

fix $h$

assume $h \in \text{hom-components} p$

hence $P h (q0 h)$ by (rule 1)

hence $h = (\sum f \in \text{fst} (q0 h). \text{snd} (q0 h) f * f)$ and $2: \bigwedge f \not\in \text{fst} (q0 h) \implies \text{snd} (q0 h) f = 0$

by (simp-all add: P-def)

note this(1)

also from (finite $F'$) have $(\sum f \in \text{fst} (q0 h). (\text{snd} (q0 h)) f * f) = (\sum f \in F'. \text{snd} (q0 h) f * f)$

proof (intro sum.mono-neutral-left ballI)

show $\text{fst} (q0 h) \subseteq F'$ unfolding $F'-def$ using $\exists h \in \text{hom-components} p$ by blast

next

fix $f$

assume $f \in F' - \text{fst} (q0 h)$

hence $f \not\in \text{fst} (q0 h)$ by simp

hence $\text{snd} (q0 h) f = 0$ by (rule 2)

thus $\text{snd} (q0 h) f * f = 0$ by simp

qed

finally show $h = (\sum f \in F'. \text{snd} (q0 h) f * f)$.

qed

also have ... = $(\sum f \in F'. \sum h \in \text{hom-components} p. \text{snd} (q0 h) f * f)$ by (rule sum.swap)

also have ... = $(\sum f \in F'. q f * f)$ by (simp only: q-def sum-distrib-right)
finally show \( p = (\sum_{f \in F'} q \ f \ f) . \)

fix \( f \)

have \( \text{poly-deg } (q \ f \ f) = \text{poly-deg } (\sum_{h \in \text{hom-components } p} . \text{snd} (q0 \ h) \ f \ f) \)
  by (simp only: q-def sum-distrib-right)

also have ... \( \leq \text{Max } (\text{poly-deg } \ (\lambda h. \text{snd} (q0 \ h) \ f \ f)) \cdot \text{hom-components } p) \)
  by (rule poly-deg-sum-le)

also have ... = \( \text{Max } ((\lambda h. \text{poly-deg} (\text{snd} (q0 \ h) \ f \ f)) \cdot \text{hom-components } p) \)
  (is - = Max (?f ‘-)) by (simp only: image-image)

also have ... \( \leq \text{poly-deg } p \)

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 \( \neq \{\} \) by simp

next
fix \( d \)
assume \( d \in ?f ‘\text{hom-components } p \)
then obtain \( h \) where \( h \in \text{hom-components } p \) and \( d = ?f \ h . \)
from this(1) have \( P h (q0 \ h) \) by (rule 1)

hence 2: \( \forall f . f \in \text{fst} (q0 \ h) \implies \text{poly-deg} (\text{snd} (q0 \ h) \ f \ f) = \text{poly-deg } h \)
and 3: \( \forall f . f \notin \text{fst} (q0 \ h) \implies \text{snd} (q0 \ h) \ f = 0 \) by (simp-all add: P-def)

show \( d \leq \text{poly-deg } p \)

proof (cases \( f \in \text{fst} (q0 \ h) \))
  case True
  hence \( \text{poly-deg} (\text{snd} (q0 \ h) \ f \ f) = \text{poly-deg } h \) by (rule 2)
  hence \( d = \text{poly-deg } h \) by (simp only: d)
  also from \( h \in \text{hom-components } p \) have ... \( \leq \text{poly-deg } p \) by (rule poly-deg-hom-components-le)

  finally show \( ?\text{thesis} . \)

next
  case False
  hence \( \text{snd} (q0 \ h) \ f = 0 \) by (rule 3)
  thus \( ?\text{thesis} \) by (simp add: d)

qed

qed

finally show \( \text{poly-deg} (q \ f \ f) \leq \text{poly-deg } p . \)

assume \( f \notin F' \)
show \( q \ f = 0 \) unfolding q-def

proof (intro sum.neutral ballI)
fix \( h \)
assume \( h \in \text{hom-components } p \)

hence \( P h (q0 \ h) \) by (rule 1)

hence 2: \( \forall f . f \notin \text{fst} (q0 \ h) \implies \text{snd} (q0 \ h) \ f = 0 \) by (simp add: P-def)

show \( \text{snd} (q0 \ h) \ f = 0 \)

proof (intro 2 notI)
  assume \( f \in \text{fst} (q0 \ h) \)
  hence \( f \in F' \) unfolding F'-def using \( h \in \text{hom-components } p \) by blast
with \( f \notin F' \), show \( \text{False} .. \)
qed
qed
qed
qed

corollary homogeneous-idealE-finite:
  assumes finite \( F \) and \( \bigwedge f \in F \rightarrow \text{homogeneous} f \) and \( p \in \text{ideal} F \)
  obtains \( q \) where \( p = (\sum f \in F'. \ q f \ast f) \) and \( \bigwedge f \. \text{poly-deg} (q f \ast f) \leq \text{poly-deg} p \)
  and \( \bigwedge f \. f \notin F \implies q f = 0 \)
proof
  from assms(2, 3) obtain \( F' q \) where \( F' \subseteq F \) and \( p: p = (\sum f \in F'. \ q f \ast f) \)
  and \( \bigwedge f \. \text{poly-deg} (q f \ast f) \leq \text{poly-deg} p \) and \( 1: \bigwedge f \. f \notin F' \implies q f = 0 \)
  by (rule homogeneous-idealE) blast+
show \( \text{thesis} \)
proof
  from assms(1) \( F' \subseteq F \); have \( (\sum f \in F'. \ q f \ast f) = (\sum f \in F. \ q f \ast f) \)
proof (intro sum.mono-neutral-left ballI)
  fix \( f \)
  assume \( f \in F - F' \)
  hence \( f \notin F' \) by simp
  hence \( q f = 0 \) by (rule 1)
  thus \( q f \ast f = 0 \) by simp
  qed
  thus \( p = (\sum f \in F. \ q f \ast f) \) by (simp only: p)
next
  fix \( f \)
  show \( \text{poly-deg} (q f \ast f) \leq \text{poly-deg} p \) by fact
  assume \( f \notin F \)
  with \( \langle F' \subseteq F \rangle \) have \( f \notin F' \) by blast
  thus \( q f = 0 \) by (rule 1)
  qed
qed

17.6.1 Homogenization and Dehomogenization

definition homogenize :: \( 'x \Rightarrow (('x \Rightarrow \text{nat}) \Rightarrow 'a) \Rightarrow (('x \Rightarrow \text{nat}) \Rightarrow 'a::semiring-1) \)
  where homogenize \( x \) \( p = (\sum t \in \text{keys} \ p. \ \text{monomial} (\text{lookup} \ p \ t) \ (\text{Poly-Mapping.single} x \ (\text{poly-deg} p - \text{deg-pm} t) + t)) \)

definition dehomo-subst :: \( 'x \Rightarrow 'x \Rightarrow (('x \Rightarrow \text{nat}) \Rightarrow 'a::zero-neq-one) \)
  where dehomo-subst \( x \) \( y = (\lambda \ y. \ \text{if} \ y = x \ \text{then} \ 1 \ \text{else} \ \text{monomial} \ 1 \ (\text{Poly-Mapping.single} y \ 1)) \)

definition dehomogenize :: \( 'x \Rightarrow (('x \Rightarrow \text{nat}) \Rightarrow 'a) \Rightarrow (('x \Rightarrow \text{nat}) \Rightarrow 'a::comm-semiring-1) \)
  where dehomogenize \( x \) = \( \text{poly-subst} (\text{dehomo-subst} x) \)
lemma homogenize-zero [simp]: homogenize x 0 = 0
by (simp add: homogenize-def)

lemma homogenize-uminus [simp]: homogenize x (- p) = - homogenize x (p::
⇒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 ⟨p̸=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

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 $q$: $q \in \text{hom-components} p$
hence homogeneous $q$ by (rule hom-components-homogeneous)

have $(\sum_{t \in \text{keys} q \cdot \text{monomial} (\text{lookup} p t \cdot \text{Poly-Mapping}.\text{single} x \cdot (\text{poly-deg} p - \text{deg-pm} t) + t)) =$

$(\sum_{t \in \text{keys} q \cdot \text{punit}\cdot\text{monom-mult} 1 \cdot \text{Poly-Mapping}.\text{single} x \cdot (\text{poly-deg} p - \text{poly-deg} q)) \cdot (\text{monomial} (\text{lookup} q t)))$

using refl
proof (rule sum.cong)
fix $t$
assume $t \in \text{keys} q$
with homogeneous $q$ have $\text{deg-pm} t = \text{poly-deg} q$ by (rule homogeneousD-poly-deg)
moreover from $q \cdot t \in \text{keys} q$ have $\text{lookup} q t = \text{lookup} p t$ by (rule lookup-hom-components)
ultimately show monomial (\text{lookup} p t \cdot \text{Poly-Mapping}.\text{single} x \cdot (\text{poly-deg} p - \text{deg-pm} t) + t) =
$punit\cdot\text{monom-mult} 1 \cdot \text{Poly-Mapping}.\text{single} x \cdot (\text{poly-deg} p - \text{poly-deg} q)) \cdot (\text{monomial} (\text{lookup} q t))$
by (simp add: punit\cdot\text{monom-mult-monomial})

qed
also have ... = punit\cdot\text{monom-mult} 1 \cdot \text{Poly-Mapping}.\text{single} x \cdot (\text{poly-deg} p - \text{poly-deg} q)) q$
by (simp only: poly-mapping-sum-monomials flip: punit\cdot\text{monom-mult-sum-right})

finally show $(\sum_{t \in \text{keys} q \cdot \text{monomial} (\text{lookup} p t \cdot \text{Poly-Mapping}.\text{single} x \cdot (\text{poly-deg} p - \text{deg-pm} t) + t)) =$
punit\cdot\text{monom-mult} 1 \cdot \text{Poly-Mapping}.\text{single} x \cdot (\text{poly-deg} p - \text{poly-deg} q)) q .

qed

finally show $?\text{thesis}$.

qed

lemma keys-homogenizeE:
assumes $t \in \text{keys} (\text{homogenize} x p)$

obtains $t' \text{ where } t' \in \text{keys} p \text{ and } t = \text{Poly-Mapping}.\text{single} x \cdot (\text{poly-deg} p - \text{deg-pm} t') + t'$

proof
note assms
also have keys (homogenize x p) $\subseteq$
$(\bigcup_{t \in \text{keys} p \cdot \text{keys} (\text{monomial} (\text{lookup} p t \cdot \text{Poly-Mapping}.\text{single} x \cdot (\text{poly-deg} p - \text{deg-pm} t) + t)))$

unfolding homogenize-def by (rule keys-sum-subset)
finally obtain $t' \text{ where } t' \in \text{keys} p$
and $t \in \text{keys} (\text{monomial} (\text{lookup} p t') \cdot \text{Poly-Mapping}.\text{single} x \cdot (\text{poly-deg} p - \text{deg-pm} t') + t')$ ..
from this(2) have $t = \text{Poly-Mapping}.\text{single} x \cdot (\text{poly-deg} p - \text{deg-pm} t') + t'$
by (simp split: if-split-asm)
with $t' \in \text{keys} p$ show $?\text{thesis}$ ..

qed
lemma keys-homogenizeE-alt:
assumes $t \in \text{keys} (\text{homogenize } x \ p)$
obtains $q t'$ where $q \in \text{hom-components } p$ and $t' \in \text{keys } q$
and $t = \text{Poly-Mapping}.\text{single } x (\text{poly-deg } p - \text{poly-deg } q) + t'$
proof –
  note assms
  also have $\text{keys} (\text{homogenize } x \ p) \subseteq$
  $\bigcup q \in \text{hom-components } p. \text{keys} (\text{punit.\text{monom-mult } 1 (Poly-Mapping}.\text{single } x (\text{poly-deg } p - \text{poly-deg } q)) q)$
  unfolding homogenize-alt by (rule keys-sum-subset)
  finally obtain $q$ where $q: q \in \text{hom-components } p$
  and $t \in \text{keys} (\text{punit.\text{monom-mult } 1 (Poly-Mapping}.\text{single } x (\text{poly-deg } p - \text{poly-deg } q)) q)$ ..
  note this(2)
  also have $\ldots \subseteq (+) (\text{Poly-Mapping}.\text{single } x (\text{poly-deg } p - \text{poly-deg } q)) \cdot \text{keys } q$
  by (rule punit.\text{keys-monom-mult-subset}[simplified])
  finally obtain $t'$ where $t' \in \text{keys } q$ and $t = \text{Poly-Mapping}.\text{single } x (\text{poly-deg } p - \text{poly-deg } q) + t'$ ..
  with $q$ show ?thesis ..
qed

lemma deg-pm-homogenize:
assumes $t \in \text{keys} (\text{homogenize } x \ p)$
shows $\text{deg-pm } t = \text{poly-deg } p$
proof –
  from assms obtain $q t'$ where $q: q \in \text{hom-components } p$ and $t' \in \text{keys } q$
  and $t: t = \text{Poly-Mapping}.\text{single } x (\text{poly-deg } p - \text{poly-deg } q) + t'$ by (rule keys-homogenizeE-alt)
  from $q$ have homogeneous $q$ by (rule hom-components-homogeneous)
  hence $\text{deg-pm } t' = \text{poly-deg } q$ using $t' \in \text{keys } q$ by (rule homogeneousD-poly-deg)
  moreover from $q$ have $\text{poly-deg } q \leq \text{poly-deg } p$ by (rule poly-deg-hom-components-le)
  ultimately show ?thesis by (simp add: $t \text{deg-pm-plus } \text{deg-pm-single}$)
qed

corollary homogeneous-homogenize: homogeneous (homogenize $x \ p$)
proof (rule homogeneousI)
  fix $s t$
  assume $s \in \text{keys} (\text{homogenize } x \ p)$
  hence $s: \text{deg-pm } s = \text{poly-deg } p$ by (rule deg-pm-homogenize)
  assume $t \in \text{keys} (\text{homogenize } x \ p)$
  hence $\text{deg-pm } t = \text{poly-deg } p$ by (rule deg-pm-homogenize)
  with $*$ show $\text{deg-pm } s = \text{deg-pm } t$ by simp
qed

corollary poly-deg-homogenize-le: $\text{poly-deg } (\text{homogenize } x \ p) \leq \text{poly-deg } p$
proof (rule poly-deg-leI)
  fix $t$
  assume $t \in \text{keys} (\text{homogenize } x \ p)$
  hence $\text{deg-pm } t = \text{poly-deg } p$ by (rule deg-pm-homogenize)
thus \( \deg-pm t \leq \poly-deg p \) by simp

qed

lemma homogenize-id-iff [simp]: homogenize \( x p = p \) if and only if \( \text{homogeneous } p \)

proof
  assume homogenize \( x p = p \)
  moreover have \( \text{homogeneous } (\text{homogenize } x p) \) by (fact homogeneous-homogenize)
  ultimately show \( \text{homogeneous } p \) by simp
next
  assume homogenize \( x p = p \)
  hence \( \text{hom-components } p = (\text{if } p = 0 \text{ then } \{ \} \text{ else } \{ p \}) \) by (rule hom-components-of-homogeneous)
  thus \( \text{homogenize } x p = p \) by (simp add: homogenize-alt split: if-split-asm)

qed

lemma homogenize-homogenize [simp]: homogenize \( x (\text{homogenize } x p) \) = homogenize \( x p \)
  by (simp add: homogeneous-homogenize)

lemma homogenize-monomial: homogenize \( x (\text{monomial } c t) \) = monomial \( c t \)
  by (simp only: homogenize-id-iff homogeneous-monomial)

lemma indets-homogenize-subset: \( \text{indets } (\text{homogenize } x p) \subseteq \text{insert } x (\text{indets } p) \)

proof
  fix \( y \)
  assume \( y \in \text{indets } (\text{homogenize } x p) \)
  then obtain \( t \) where \( t \in \text{keys } (\text{homogenize } x p) \) and \( y \in \text{keys } t \) by (rule in-indetsE)
  from this(1) obtain \( t' \) where \( t' \in \text{keys } p \)
    and \( t \cdot t = \text{Poly-Mapping.single } x (\poly-deg p - \deg-pm t') + t' \) by (rule keys-homogenizeE)
  note \( (y \in \text{keys } t) \)
  also have \( \text{keys } t \subseteq \text{keys } (\text{Poly-Mapping.single } x (\poly-deg p - \deg-pm t')) \cup \text{keys } t' \)
    unfolding \( t \) by (rule Poly-Mapping.keys-add)
  finally show \( y \in \text{insert } x (\text{indets } p) \)
proof
  assume \( y \in \text{keys } (\text{Poly-Mapping.single } x (\poly-deg p - \deg-pm t')) \)
  thus \( \text{thesis } \) by (simp split: if-split-asm)
next
  assume \( y \in \text{keys } t' \)
  hence \( y \in \text{indets } p \) using \( t' \in \text{keys } p \) by (rule in-indetsI)
  thus \( \text{thesis } \) by simp
qed

lemma homogenize-in-Polys: \( p \in P[X] \implies \text{homogenize } x p \in P[\text{insert } x X] \)
  using indets-homogenize-subset[of \( x p \)] by (auto simp: Polys-alt)

lemma lookup-homogenize:
assumes $x \notin \text{indets } p$ and $x \notin \text{keys } t$

shows lookup (homogenize $x$ $p$) $(\text{Poly-Mapping.single } x (\text{poly-deg } p - \text{deg-pm } t) + t) = \text{lookup } p \ t$

proof –
let $?p = \text{homogenize } x \ p$
let $?t = \text{Poly-Mapping.single } x (\text{poly-deg } p - \text{deg-pm } t) + t$

have eq: $(\sum s \in \text{keys } p - \{ t \}, \text{lookup } (\text{monomial } (\text{lookup } p \ s) (\text{Poly-Mapping.single } x (\text{poly-deg } p - \text{deg-pm } s) + s)) \ ?t) = 0$

proof (intro sum.neutral ballI)
fix $s$
assume $s \in \text{keys } p - \{ t \}$
hence $s \in \text{keys } p$ and $s \neq t$ by simp-all
from this(1) have keys $s \subseteq \text{indets } p$ by (simp add: in-indetsI subsetI)
with assms(1) have $x \notin \text{keys } s$ by blast
have $?t \neq \text{Poly-Mapping.single } x (\text{poly-deg } p - \text{deg-pm } s) + s$

proof
assume $a$: $?t = \text{Poly-Mapping.single } x (\text{poly-deg } p - \text{deg-pm } s) + s$
hence lookup $?t \ x = \text{lookup } (\text{Poly-Mapping.single } x (\text{poly-deg } p - \text{deg-pm } s) + s) \ x$
by simp
moreover from assms(2) have lookup $t \ x = 0$ by (simp add: in-keys-iff)
moreover from $x \notin \text{keys } s$ have lookup $s \ x = 0$ by (simp add: in-keys-iff)
ultimately have poly-deg $p - \text{deg-pm } t = \text{poly-deg } p - \text{deg-pm } s$ by (simp add: lookup-add)
with $a$ have $s = t$ by simp
with ($s \neq t$) show False ..

qed

thus lookup (monomial (lookup $p \ s$) (Poly-Mapping.single $x (\text{poly-deg } p - \text{deg-pm } s) + s)) \ ?t = 0$
by (simp add: lookup-single)

qed

show $?\text{thesis}$

proof (cases $t \in \text{keys } p$

  case True
  have lookup $?p \ ?t = (\sum s \in \text{keys } p, \text{lookup } (\text{monomial } (\text{lookup } p \ s) (\text{Poly-Mapping.single } x (\text{poly-deg } p - \text{deg-pm } s) + s)) \ ?t)$
  by (simp add: homogenize-def lookup-sum)
  also have 
  $(\sum s \in \text{keys } p - \{ t \}, \text{lookup } (\text{monomial } (\text{lookup } p \ t) \ ?t + (\sum s \in \text{keys } p - \{ t \}, \text{lookup } (\text{monomial } (\text{lookup } p \ s) (\text{Poly-Mapping.single } x (\text{poly-deg } p - \text{deg-pm } s) + s)) \ ?t)$
  using finite-keys True by (rule sum.remove)
  also have 
  $(\sum s \in \text{keys } p - \{ t \}, \text{lookup } (\text{monomial } (\text{lookup } p \ s) (\text{Poly-Mapping.single } x (\text{poly-deg } p - \text{deg-pm } s) + s)) \ ?t)$
  by (simp add: lookup-single)

  qed

next
  case False
  hence $1$: keys $p - \{ t \} = \text{keys } p$ by simp
  have lookup $?p \ ?t = (\sum s \in \text{keys } p - \{ t \}, \text{lookup } (\text{monomial } (\text{lookup } p \ s) (\text{Poly-Mapping.single } x (\text{poly-deg } p - \text{deg-pm } s) + s)) \ ?t)$
  by (simp add: homogenize-def lookup-sum 1)

  qed

492
also have \( \ldots = 0 \) by (simp only: eq)
also from False have \( \ldots = \text{lookup } p \ t \) by (simp add: in-keys-iff)
finally show ?thesis.

qed

qed

lemma keys-homogenizeI:
  assumes \( x \notin \text{indets } p \) and \( t \in \text{keys } p \)
  shows Poly-Mapping.single \( x \) (poly-deg \( p \) \( - \) deg-pm \( t \)) \( + \) \( t \) \( \in \) keys (homogenize \( x \) \( p \)) (is \( ?t \in \text{keys } ?p \))
proof -
  from assms(2) have keys \( t \subseteq \text{indets } p \) by (simp add: in-indetsI subsetI)
  with assms(1) have \( x \notin \text{keys } t \) by blast
  with assms(1) have lookup \( ?p \ ?t = \text{lookup } p \ t \) by (rule lookup-homogenize)
  also from assms(2) have \( \ldots \neq 0 \) by (simp add: in-keys-iff)
  finally show ?thesis by (simp add: in-keys-iff)
qed

lemma keys-homogenize:
  \( x \notin \text{indets } p \implies \text{keys } (\text{homogenize } x \ p) = (\forall t. \text{Poly-Mapping.single } x \ (\text{poly-deg } p \ - \ \text{deg-pm } t) \ + \ t \ \in \ \text{keys } (\text{homogenize } x \ p)) \)
  by (auto intro: keys-homogenizeI elim: keys-homogenizeE)

lemma card-keys-homogenize:
  assumes \( x \notin \text{indets } p \)
  shows card (keys (homogenize \( x \) \( p \))) = card (keys \( p \))
proof (intro card-image inj-onI)
  fix \( s \ t \)
  assume \( s \in \text{keys } p \) and \( t \in \text{keys } p \)
  with assms have \( x \notin \text{keys } s \) and \( x \notin \text{keys } t \) by (auto dest: in-indetsI simp only:)
  let \( ?s = \text{Poly-Mapping.single } x \ (\text{poly-deg } p \ - \ \text{deg-pm } s) \)
  let \( ?t = \text{Poly-Mapping.single } x \ (\text{poly-deg } p \ - \ \text{deg-pm } t) \)
  assume \( ?s + s = ?t + t \)
  hence lookup \( ?s + s \) \( x = \text{lookup } (?t + t) \ x \) by simp
  with \( x \notin \text{keys } s \) \( x \notin \text{keys } t \) 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 \( x \notin \text{indets } p \)
  shows \( \text{poly-deg } (\text{homogenize } x \ p) = \text{poly-deg } p \)
proof (cases \( p = 0 \))
  case True
  thus ?thesis by simp
next
  case False
  then obtain \( t \) where \( t \in \text{keys } p \) and \( 1 \): \( \text{poly-deg } p = \text{deg-pm } t \) by (rule
poly-degE)
  from assms this(1) have Poly-Mapping.single $x$ (poly-deg $p$ - deg-pm $t$) + $t$ \in keys (homogenize $x$ $p$)
    by (rule keys-homogenize)
  hence $t$ \in keys (homogenize $x$ $p$) by (simp add: 1)
  hence poly-deg $p$ \leq 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 $x$ \notin \bigcup (indets $^\cdot$ F)
  shows maxdeg (homogenize $x$ $^\cdot$ F) = maxdeg F
unfolding maxdeg-def image-image
proof (rule arg-cong[where $f$=Max], rule set-eqI)
  fix $d$
  show $d$ \in $(\lambda f. \text{poly-deg (homogenize } x f))$ $^\cdot$ F \longleftrightarrow $d$ \in poly-deg $^\cdot$ F
proof
  assume $d$ \in $(\lambda f. \text{poly-deg (homogenize } x f))$ $^\cdot$ F
  then obtain $f$ where $f$ \in F \and $d$ = poly-deg (homogenize $x$ $f$) ..
  from assms this(1) have $x$ \notin indets $f$ by blast
  hence $d$ = poly-deg $f$ by (simp add: $d$ poly-deg-homogenize)
  with ($f$ \in F) show $d$ \in poly-deg $^\cdot$ F by (rule rev-image-eqI)
next
  assume $d$ \in poly-deg $^\cdot$ F
  then obtain $f$ where $f$ \in F \and $d$ = poly-deg $f$ ..
  from assms this(1) have $x$ \notin indets $f$ by blast
  hence $d$ = poly-deg (homogenize $x$ $f$) by (simp add: $d$ poly-deg-homogenize)
  with ($f$ \in F) show $d$ \in $(\lambda f. \text{poly-deg (homogenize } x f))$ $^\cdot$ F by (rule rev-image-eqI)
qed
  qed

lemma homogeneous-ideal-homogenize:
  assumes $\bigwedge f. f$ \in F $\Rightarrow$ homogeneous $f$ and $p$ \in ideal F
  shows homogenize $x$ $p$ \in ideal F
proof
  have homogenize $x$ $p$ = $(\Sigma q \in \text{hom-components } p. \text{punit.monom-mult } 1 \ (\text{Poly-Mapping.single } x \ (\text{poly-deg } p - \text{poly-deg } q)) \ q)$
    by (fact homogenize-alt)
  also have $\ldots$ \in ideal F
proof (rule ideal.span-sum)
  fix $q$
  assume $q$ \in \text{hom-components } $p$
  with assms have $q$ \in ideal F by (rule homogeneous-ideal')
  thus punit.monom-mult $1$ (Poly-Mapping.single $x$ (poly-deg $p$ - poly-deg $q$)) $q$ \in ideal F
    by (rule punit.pmdl-closed-monom-mult[simplified])
  qed
finally show ?thesis .
  qed
lemma subst-pp-dehomo-subst [simp]:
\[ \text{subst-pp (dehomo-subst } x \text{)} t = \text{monomial (1 :: 'b::comm-semiring-1) (except } t \{ x \}) \]

proof -
  have subst-pp (dehomo-subst } x \text{)} t = ((\prod_{y \in \text{keys } t} \text{. dehomo-subst } x y \text{)}^\le \text{lookup } t y):\Rightarrow_0 'b) 
    by (fact subst-pp-def)
  also have \ldots = (\prod_{y \in \text{keys } t - \{ y_0 \}} \text{. dehomo-subst } x y_0 \text{)}^\le \text{lookup } t y_0 = (1::-
    \Rightarrow_0 'b). \text{ dehomo-subst } x y \text{)}^\le \text{lookup } t y)
    by (rule sym, rule prod.setdiff-irrelevant, fact finite-keys)
  also have \ldots = (\prod_{y \in \text{keys } t - \{ x \}} \text{. monomial } 1 \text{ (Poly-Mapping.single } y 1 \text{)}^\le \text{lookup } t y)
    proof (rule prod.cong)
      have dehomo-subst } x x \text{)}^\le \text{lookup } t x \text{)} = 1 by (simp add: dehomo-subst-def)
      moreover \{ 
        fix y 
        assume y \neq x 
        hence dehomo-subst } x y \text{)}^\le \text{lookup } t y \text{)} = \text{monomial } 1 \text{ (Poly-Mapping.single } y \text{)}^\le \text{lookup } t y)
          by (simp add: dehomo-subst-def monomial-single-power)
        moreover assume dehomo-subst } x y \text{)}^\le \text{lookup } t y \text{)} = 1
        ultimately have Poly-Mapping.single } y \text{)}^\le \text{lookup } t y \text{)} = 0 
          by (smt single-one monomial-inj zero-neq-one)
        hence lookup } t y \text{)} = 0 by (rule monomial-0D)
        moreover assume y \in \text{keys } t
        ultimately have False by (simp add: in-keys-iff)
      \}
      ultimately show keys } t \text{)} - \{ y_0 \} \text{. dehomo-subst } x y_0 \text{)}^\le \text{lookup } t y_0 \text{)} = 1 \text{ = keys } t \text{)} - \{ x \}) by auto 
    qed (simp add: dehomo-subst-def)
  also have \ldots = (\prod_{y \in \text{keys } t - \{ x \}} \text{. monomial } 1 \text{ (Poly-Mapping.single } y \text{)}^\le \text{lookup } t y)) 
    by (simp add: monomial-single-power)
  also have \ldots = monomial } 1 \text{ (\sum } y \in \text{keys } t - \{ x \}. \text{ Poly-Mapping.single } y \text{)}^\le \text{lookup } t y) 
    by (simp flip: punit.monomial-prod-sum)
  also have (\sum } y \in \text{keys } t - \{ x \}. \text{ Poly-Mapping.single } y \text{)}^\le \text{lookup } t y) = \text{except } t \{ x \}
    proof (rule poly-mapping-eqI, simp add: lookup-sum lookup-except lookup-single, rule)
      fix y 
      assume y \neq x 
      show (*) = \sum } z \in \text{keys } t - \{ y \} \text{. lookup } t z \text{ when } z = y \text{)} = \text{lookup } t y
        proof (cases y \in \text{keys } t)
          case True
          have finite (keys } t \text{)} - \{ x \}) by simp
          moreover from True \{ y \neq x \} have y \in \text{keys } t \text{)} - \{ x \}) by simp
          ultimately have (\sum } z \in \text{keys } t - \{ x \}. \text{ lookup } t z \text{ when } z = y \text{)} = (*)
            (\text{lookup } t y \text{ when } y = y) + (\sum } z \in \text{keys } t - \{ x \} - \{ y \}. \text{ lookup } t y)
\[ z \text{ when } z = y \]

by (rule sum, remove)

also have \((\sum_{z \in \text{keys } t \setminus \{x\}} \cdot \text{lookup } t \cdot z \text{ when } z = y) = 0\) by auto

finally show ?thesis by simp

next

case False

hence \((\sum_{z \in \text{keys } t \setminus \{x\}} \cdot \text{lookup } t \cdot z \text{ when } z = y) = 0\) by (auto simp: when-def)

also from False have \(\ldots = \text{lookup } t \cdot y\) by (simp add: in-keys-iff)

finally show ?thesis .

qed

corollary dehomogenize-monom-mult:

dehomogenize \((\text{punit.monom-mult } c \cdot t \cdot p)\) = \(\text{punit.monom-mult } c \text{ (except } t \cdot \{x\})\)

(dehomogenize \(x \cdot p\))

by (simp only: times-monomial-left[symmetric] dehomogenize-times dehomogenize-monomial)

lemma poly-deg-dehomogenize-le: \(\text{poly-deg } (\text{dehomogenize } x \cdot p) \leq \text{poly-deg } p\)

unfolding dehomogenize-def dehomo-subst-def

by (rule poly-deg-poly-subst-le) (simp add: poly-deg-monomial deg-pm-single)

lemma indets-dehomogenize: \(\text{indets } (\text{dehomogenize } x \cdot p) \subseteq \text{indets } p \setminus \{x\}\)

for \(p::(\{x \Rightarrow \_\} \Rightarrow \{x\} \Rightarrow \_::\text{comm-semiring-1})\)

proof
fix y::'x
assume y ∈ indets (dehomogenize x p)
then obtain y' where y' ∈ indets p and y ∈ indets ((dehomo-subst x y')::-) ⇒0

unfolding dehomogenize-def by (rule in-indets-poly-substE)
from this(2) have y = y' and y' ≠ x
by (simp-all add: dehomo-subst-def indets-monomial split: if-split-asm)
with (y' ∈ indets p) show y ∈ indets p - {x} by simp
qed

lemma dehomogenize-id-iff [simp]: dehomogenize x p = p ⟷ x /∈ indets p
proof
assume eq: dehomogenize x p = p
from indets-dehomogenize[of x p] show x /∈ indets p by (auto simp: eq)
next
assume a: x /∈ indets p
show dehomogenize x p = p unfolding dehomogenize-def
proof (rule poly-subst-id)
fix y
assume y ∈ indets p
with a have y ≠ 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 /∈ 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 /∈ indets p ⟹ dehomogenize x (homogenize
x p) = p
by simp
lemma range-dehomogenize: range (dehomogenize x) = (P[ − {x}] :: (- ⇒ 0 'a::comm-semiring-1) set)

proof (intro subset-antisym subsetI PolysI-alt range-cqI)
  fix p:: ⇒ 0 'a and y
  assume p ∈ range (dehomogenize x)
  then obtain q where p = dehomogenize x q ..
  assume y ∈ indets p
  hence y ∈ indets (dehomogenize x q) by (simp only: p)
  with indets-dehomogenize have y ∈ indets q − {x} ..
  thus y ∈ − {x} by simp

next
  fix p:: ⇒ 0 'a
  assume p ∈ P[ − {x}]
  hence x /∈ 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 = (∑ t∈keys p. monomial (lookup p t) (except t {x}))

proof –
  have dehomogenize x p = dehomogenize x (∑ t∈keys p. monomial (lookup p t))
  by (simp only: poly-mapping-sum-monomials)
  also have .. = (∑ 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 ∈ keys (dehomogenize x p)
  obtains s where s ∈ keys p and t = except s {x}

proof –
  note assms
  also have keys (dehomogenize x p) ⊆ (∪ s∈keys p. keys (monomial (lookup p s) (except s {x})))
  unfolding dehomogenize-alt by (rule keys-sum-subset)
  finally obtain s where s ∈ keys p and t ∈ keys (monomial (lookup p s) (except s {x})) ..
  from this(2) have t = except s {x} by (simp split: if-split-asm)
  with (s ∈ keys p) show ?thesis ..

qed

lemma except-inj-on-keys-homogeneous:
  assumes homogeneous p
  shows inj-on (λt. except t {x}) (keys p)

proof
  fix s t
  assume s ∈ keys p and t ∈ keys p
  from assms this(1) have deg-pm s = poly-deg p by (rule homogeneousD-poly-deg)
moreover from assms ⟨t ∈ keys p⟩ have deg-pm t = poly-deg p by (rule homogeneousD-poly-deg)
ultimately have deg-pm (Poly-Mapping.single x (lookup s x) + except s {x})
= deg-pm (Poly-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

lemma lookup-dehomogenize:
  assumes homogeneous p and t ∈ 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(t) have inj-on (λt. 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 \[ \ldots = \text{lookup} \ (\text{monomial} \ (\text{lookup} \ p \ t) \ ?t) \ ?t + \left( \sum_{s \in \text{keys} \ p - \{t\}} \text{lookup} \ (\text{monomial} \ (\text{lookup} \ p \ s) \ \text{(except} \ s \ \{x\})) \right) \ ?t \]
also have \[ \ldots = \text{lookup} \ p \ t \ by \ (\text{rule sum.remove}) \]
finally show \[ \text{thesis} \].
qed

lemma keys-dehomogenizeI:
assumes homogeneous \( p \) and \( t \in \text{keys} \ p \)
shows \( \text{except} \ t \ \{x\} \in \text{keys} \ (\text{dehomogenize} \ x \ p) \)
proof
from assms have \( \text{lookup} \ (\text{dehomogenize} \ x \ p) \ \text{(except} \ t \ \{x\}) = \text{lookup} \ p \ t \ by \ (\text{rule lookup-dehomogenize}) \)
also from assms (2) have \( \ldots \neq 0 \ by \ (\text{simp add: in-keys-iff}) \)
finally show \[ \text{thesis} \ by \ (\text{simp add: in-keys-iff}) \]
qed

lemma homogeneous-homogenize-dehomogenize:
assumes homogeneous \( p \)
obtains \( d \) where \( d = \text{poly-deg} \ p - \text{poly-deg} \ (\text{homogenize} \ x \ (\text{dehomogenize} \ x \ p)) \)
and \( \text{punit.monom-mul} \ 1 \ (\text{Poly-Mapping.single} \ x \ d) \ (\text{homogenize} \ x \ (\text{dehomogenize} \ x \ p)) = p \)
proof (cases \( p = 0 \))
case True
hence \( 0 = \text{poly-deg} \ p - \text{poly-deg} \ (\text{homogenize} \ x \ (\text{dehomogenize} \ x \ p)) \)
and \( \text{punit.monom-mul} \ 1 \ (\text{Poly-Mapping.single} \ x \ 0) \ (\text{homogenize} \ x \ (\text{dehomogenize} \ x \ p)) = p \)
by simp-all
thus \[ \text{thesis} \ .. \]
next
case False
let \( ?q = \text{dehomogenize} \ x \ p \)
let \( ?p = \text{homogenize} \ x \ ?q \)
define \( d \) where \( d = \text{poly-deg} \ p - \text{poly-deg} \ ?p \)
show \[ \text{thesis} \]
proof
have \( \text{punit.monom-mul} \ 1 \ (\text{Poly-Mapping.single} \ x \ d) \ ?p = \left( \sum_{t \in \text{keys} \ ?q} \text{?q.monomial} \ (\text{lookup} \ ?q \ t) \ (\text{Poly-Mapping.single} \ x \ (d + (\text{poly-deg} \ ?q - \text{deg-pm} \ t)) + t) \right) \)
by (simp add: homogenize-def punit.monom-mul-sum-right punit.monom-mul-monomial flip: add.assoc single-add)
also have \[ \ldots = (\sum_{t \in \text{keys} \ ?q} \text{monomial} \ (\text{lookup} \ ?q \ t) \ (\text{Poly-Mapping.single} \ x \ (\text{poly-deg} \ p - \text{deg-pm} \ t) + t)) \]
using refl
proof (rule sum.cong)
fix \( t \)
assume \( t \in \text{keys} \ ?q \)
have \( \text{poly-deg } ?p = \text{poly-deg } ?q \)
proof (rule poly-deg-homogenize)
  from indets-dehomogenize show \( x \notin \text{indets } ?q \) by fastforce
qed

hence \( d \) by simp only: \( d \)-def
thm poly-deg-dehomogenize-le
from \( t \in \text{keys } ?q \) have \( d + (\text{poly-deg } ?q - \deg-pm t) = (d + \text{poly-deg } ?q) - \deg-pm t \)
  by (intro add-diff-assoc poly-deg-max-keys)
also have \( d + \text{poly-deg } ?q = \text{poly-deg } p \) by simp add: \( d \)-poly-deg-dehomogenize-le
finally show \( \text{monomial } (\text{lookup } ?q t) \) by (simp only:)
qed
also have \( \ldots = (\sum t \in (\lambda s. \text{except } s \{ x \}) \cdot \text{keys } p) \cdot \text{monomial } (\text{lookup } ?q t) \) by (simp only:)
proof (rule sum-mono-neutral-left)
  show keys (dehomogenize \( p \cdot x \)) \subseteq (\lambda s. \text{except } s \{ x \}) \cdot \text{keys } p
proof
  fix \( t \)
  assume \( t \in \text{keys } (\text{dehomogenize } p \cdot x) \)
  then obtain \( s \) where \( s \in \text{keys } p \) and \( t = \text{except } s \{ x \} \) by (rule keys-dehomogenizeE)
  thus \( t \in (\lambda s. \text{except } s \{ x \}) \cdot \text{keys } p \) by (rule rev-image-eqI)
qed
also from \( \text{assms} \) have \( \ldots = (\sum t \in \text{keys } p \cdot \text{monomial } (\text{lookup } ?q (\text{except } t \{ x \}))) \) by (intro sum-reindex[unfolded comp-def] except-inj-on-keys-homogeneous)
also from refl have \( \ldots = (\sum t \in \text{keys } p \cdot \text{monomial } (\text{lookup } p \) t t) \) by (rule sum-cong)
fix \( t \)
assume \( t \in \text{keys } p \)
with \( \text{assms} \) have \( \text{lookup } ?q (\text{except } t \{ x \}) = \text{lookup } p \) t by (rule lookup-dehomogenize)
moreover have \( \text{Poly-Mapping.single } x \cdot (\text{poly-deg } p - \deg-pm (\text{except } t \{ x \})) \)
  + except \( t \{ x \} = t \)
(is \( ?l = - \))
proof (rule poly-mapping-eqI)
  fix \( y \)
  show \( \text{lookup } ?l \) y = \( \text{lookup } t y \)
  proof (cases \( y = x \))
    case True
    from \( \text{assms} \) \( t \in \text{keys } p \) have \( \deg-pm t = \text{poly-deg } p \) by (rule homogeneousD-poly-deg)
    also have \( \deg-pm t = \text{poly-deg } \cdot \text{Poly-Mapping.single } x \cdot \text{lookup } t x + \text{except } t \{ x \} \)
501
by (simp flip: plus-except)
also have \ldots = 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 \ldots = p by (fact poly-mapping-sum-monomials)
qed (simp only: d-def)

lemma dehomogenize-zeroD:
assumes dehomogenize x p = 0 and homogeneous p
shows p = 0
proof -
from assms(2) obtain d
where punit.monom-mul 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) \cap 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 \subseteq ideal (dehomogenize x ' F)
by (simp add: dehomogenize-ideal)

lemma ideal-dehomogenize:
assumes ideal G = ideal (homogenize x ' F) and F \subseteq P[UNIV - \{x\}]
shows ideal (dehomogenize x ' G) = ideal F
proof -
have eq: dehomogenize x (homogenize x f) = f if f \in F for f
proof (rule dehomogenize-homogenize-id)
from that assms(2) have f \in P[UNIV - \{x\}] ...
thus x \notin indets f by (auto simp: Polys-alt)
17.7 Embedding Polynomial Rings in Larger Polynomial Rings
(With One Additional Indeterminate)

We define a homomorphism for embedding a polynomial ring in a larger
polynomial ring, and its inverse. This is mainly needed for homogenizing
wrt. a fresh indeterminate.

definition extend-indets-subst :: 'x ⇒ ('x option ⇒0 nat) ⇒0 'a::comm-semiring-1
where extend-indets-subst x = monomial 1 (Poly-Mapping.single (Some x) 1)

definition extend-indets :: ('x ⇒0 nat) ⇒0 'a ⇒ ('x option ⇒0 nat) ⇒0 'a::comm-semiring-1
where extend-indets = poly-subst extend-indets-subst

definition restrict-indets-subst :: 'x option ⇒ 'x ⇒0 nat
where restrict-indets-subst x = (case x of Some y ⇒ Poly-Mapping.single y 1 | - ⇒ 0)

definition restrict-indets :: (('x option ⇒ 0 nat) ⇒ 0 'a) ⇒ ('x ⇒ 0 nat) ⇒ 0 'a::comm-semiring-1
where restrict-indets = poly-subst (λx. monomial 1 (restrict-indets-subst x))

definition restrict-indets-pp :: (('x option ⇒ 0 nat) ⇒ 0 'a)
where restrict-indets-pp t = (∑ x∈keys t. lookup t x ∙ restrict-indets-subst x)

lemma lookup-extend-indets-subst-aux:
lookup (∑ y∈keys t. Poly-Mapping.single (Some y) (lookup t y)) = (λx. case x of Some y ⇒ lookup t y | - ⇒ 0)

proof
- have (∑ x∈keys t. lookup t x when x = y) = lookup t y for y
  proof (cases y ∈ keys t)
  case True
  hence (∑ x∈keys t. lookup t x when x = y) = (∑ x∈insert y (keys t). lookup t x when x = y)
    by (simp only: insert-absorb)
  also have ... = lookup t y + (∑ x∈keys t − {y}. lookup t x when x = y)
    by (simp add: sum.insert-remove)
  also have (∑ 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 (∑ 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 (∑ y∈keys t. Poly-Mapping.single (Some y) (lookup t y)) = Some ' keys t

lemma subst-pp-extend-indets-subst
subst-pp extend-indets-subst t = monomial 1 (∑ y∈keys t. Poly-Mapping.single (Some y) (lookup t y))

proof
- have subst-pp extend-indets-subst t =
  monomial (∏ y∈keys t. 1 ∙ lookup t y) (∑ 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 (∑ y∈keys t. Poly-Mapping.single (Some y) (lookup t y))
    by simp

504
finally show \(\text{thesis} \).

\[\text{lemma keys-extend-indets:} \]
\[\text{keys (extend-indets p)} = (\lambda t. \sum y \in \text{keys } \mathbf{t}. \text{Poly-Mapping.single } (\text{Some } y) (\text{lookup } \mathbf{t } y)) \cdot \text{keys p} \]
\[\text{proof –} \]
\[\text{have keys (extend-indets p)} = (\bigcup t \in \text{keys } \mathbf{p}. \text{keys (punit.monom-mult } (\text{lookup } \mathbf{p} \mathbf{t})) \cdot (\text{subst-pp extend-indets-subst } \mathbf{t})) \]
\[\text{unfolding extend-indets-def poly-subst-def using finite-keys} \]
\[\text{proof (rule keys-sum)} \]
\[\text{fix } s t :: 'a \Rightarrow 0 \text{ nat} \]
\[\text{assume } s \neq t \]
\[\text{then obtain } x \text{ where } \text{lookup } s x \neq \text{lookup } t x \text{ by (meson poly-mapping-eqI)} \]
\[\text{have } (\sum y \in \text{keys } \mathbf{t}. \text{monomial } (\text{lookup } \mathbf{t } y) (\text{Some } y)) \neq (\sum y \in \text{keys } \mathbf{s}. \text{monomial } (\text{lookup } \mathbf{s } y) (\text{Some } y)) \]
\[\text{(is } ?l \neq ?r) \]
\[\text{proof} \]
\[\text{assume } ?l = ?r \]
\[\text{hence } \text{lookup } ?l (\text{Some } x) = \text{lookup } ?r (\text{Some } x) \text{ by (simp only:)} \]
\[\text{hence } \text{lookup } s x = \text{lookup } t x \text{ by (simp add: lookup-extend-indets-subst-aux)} \]
\[\text{with } (\text{lookup } s x \neq \text{lookup } t x): \text{show False ..} \]
\[\text{qed} \]
\[\text{thus keys (punit.monom-mult } (\text{lookup } \mathbf{p } s) \cdot (\text{subst-pp extend-indets-subst } \mathbf{s})) \]
\[\cap \]
\[\text{keys (punit.monom-mult } (\text{lookup } \mathbf{p} \mathbf{t}) \cdot (\text{subst-pp extend-indets-subst } \mathbf{t})) = \}
\[\text{by (simp add: subst-pp-extend-indets-subst punit.monom-mult-monomial)} \]
\[\text{qed} \]
\[\text{also have } \ldots = (\lambda t. \sum y \in \text{keys } \mathbf{t}. \text{monomial } (\text{lookup } \mathbf{t } y) (\text{Some } y)) \cdot \text{keys p} \]
\[\text{by (auto simp: subst-pp-extend-indets-subst punit.monom-mult-monomial split: if-split-asm)} \]
\[\text{finally show } \text{thesis} \).

\[\text{qed} \]

\[\text{lemma indets-extend-indets: indets (extend-indets p)} = \text{Some } \cdot \text{indets } (p::- \Rightarrow_0 'a::comm-semiring-1)} \]
\[\text{proof (rule set-eqI)} \]
\[\text{fix } x \]
\[\text{show } x \in \text{indets (extend-indets p)} \iff x \in \text{Some } \cdot \text{indets p} \]
\[\text{proof} \]
\[\text{assume } x \in \text{indets (extend-indets p)} \]
\[\text{then obtain } y \text{ where } y \in \text{indets } \mathbf{p} \text{ and } x \in \text{indets } (\text{monomial } (1::'a) \text{ (Poly-Mapping.single } (\text{Some } y) \mathbf{1})) \]
\[\text{unfolding extend-indets-def extend-indets-subst-def by (rule in-indets-poly-substE)} \]
\[\text{from this(2) indets-monomial-single-subset have } x \in \{\text{Some } y\} .. \]
\[\text{hence } x = \text{Some } y \text{ by simp} \]
\[\text{with } (y \in \text{indets } \mathbf{p}) \text{ show } x \in \text{Some } \cdot \text{indets } \mathbf{p} \text{ by (rule rev-image-eqI)} \]
\[\text{next} \]

505
assume \( x \in \text{Some ' indets p} \)
then obtain \( y \) where \( y \in \text{indets p} \) and \( x = \text{Some y} \).

from this(1) obtain \( t \) where \( t \in \text{keys p} \) and \( y \in \text{keys t} \) by (rule in-indetsE)
from this(2) have Some \( y \in \text{keys } (\sum y \in \text{keys t}. \text{Poly-Mapping.single } (\text{Some } y)) \) (lookup \( t \ y) \)
unfolding keys-extend-indets-subst-aux by (rule imageI)
moreover have \((\sum y \in \text{keys t}. \text{Poly-Mapping.single } (\text{Some } y)) \) (lookup \( t \ y)) \in keys (extend-indets p)
unfolding keys-extend-indets using \( (t \in \text{keys p}) \) by (rule imageI)
ultimately show \( x \in \text{indets } (\text{extend-indets p}) \) 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: \( \text{deg-pm } ((\sum y \in \text{keys t}. \text{Poly-Mapping.single } (\text{Some } y)) \) (lookup \( t \ y)) = \( \text{deg-pm } t \)
  for \( t::'a \Rightarrow 0 \text{ nat} \)
  proof –
    have \( \text{deg-pm } ((\sum y \in \text{keys t}. \text{Poly-Mapping.single } (\text{Some } y)) \) (lookup \( t \ y)) = (\sum y \in \text{keys t}. \text{lookup } t \ y) \)
  by (simp only: eq)
  also from subset-refl finite-keys have \( \ldots = \text{deg-pm } t \)
  by (rule deg-pm-superset[symmetric])
  finally show \( \)thesis .
qed

show \( \)thesis
proof (rule antisym)
  show poly-deg (extend-indets p) \( \leq \) poly-deg p
  proof (rule poly-deg-leI)
    fix \( t \)
    assume \( t \in \text{keys } (\text{extend-indets p}) \)
    then obtain \( s \) where \( s \in \text{keys p} \) and \( t = (\sum y \in \text{keys s}. \text{Poly-Mapping.single } (\text{Some } y)) \) (lookup \( s \ y) \)
    unfolding keys-extend-indets ..
    from this(2) have deg-pm t \( = \text{deg-pm } s \) by (simp only: eq)
    also from \( (s \in \text{keys } p) \) have \( \ldots \leq \text{poly-deg } p \)
    by (rule poly-deg-max-keys)
    finally show \( \text{deg-pm } t \leq \text{poly-deg } p \).
  qed

next
  show poly-deg p \( \leq \) poly-deg (extend-indets p)
  proof (rule poly-deg-leI)
    fix \( t \)
    assume \( t \in \text{keys } p \)
    hence \( *: (\sum y \in \text{keys t}. \text{Poly-Mapping.single } (\text{Some } y)) \) (lookup \( t \ y)) \in \text{keys (extend-indets p)} \)
    unfolding keys-extend-indets by (rule imageI)
    have deg-pm t \( = \text{deg-pm } (\sum y \in \text{keys t}. \text{Poly-Mapping.single } (\text{Some } y)) \) (lookup \( t \ y))
    by (simp only: eq)
also from \( \ast \) have \( \ldots \leq \text{poly-deg} (\text{extend-indets } p) \) by (rule poly-deg-max-keys)
finally show \( \text{deg-pm } t \leq \text{poly-deg} (\text{extend-indets } p) \).
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 (subt-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))

lemma extend-indets-zero-iff [simp]: extend-indets p = 0 ↔ p = 0

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

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 (\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) = (\x. lookup t (Some x))
proof
  let \f = \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 \in 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 \ldots = 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
qed

lemma keys-restrict-indets-pp: keys (restrict-indets-pp t) = the ' (keys t - {None})
proof (rule set-eqI)
  fix x
  show x \in keys (restrict-indets-pp t) \iff x \in the ' (keys t - {None})
  proof
    assume x \in keys (restrict-indets-pp t)
    hence Some x \in keys t by (simp add: lookup-restrict-indets-pp flip: lookup-not-eq-zero-eq-in-keys)
    hence Some x \in keys t - {None} by blast
    moreover have x = the (Some x) by simp
    ultimately show x \in the ' (keys t - {None}) by (rule rev-image-eqI)
  next
    assume x \in the ' (keys t - {None})
    then obtain y where y \in keys t - {None} and x = the y ..
hence $x \in \text{keys } t$ by auto
thus $x \in \text{keys (restrict-indets-pp } t)$
qed

lemma subst-pp-restrict-indets-subst:
subst-pp $(\lambda x.\text{monomial } 1 \text{ (restrict-indets-subst } x)) \ t = \text{monomial } 1 \text{ (restrict-indets-pp } t)$
  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 $(s + t) = \text{restrict-indets-pp } s + \text{restrict-indets-pp } t$
  by (rule poly-mapping-eqI) (simp add: lookup-add lookup-restrict-indets-pp)

lemma restrict-indets-pp-except-None [simp]:
  restrict-indets-pp $(\text{except } t \{\text{None}\}) = \text{restrict-indets-pp } t$
  by (rule poly-mapping-eqI) (simp add: lookup-add lookup-restrict-indets-pp lookup-except)

lemma deg-pm-restrict-indets-pp:
  deg-pm $(\text{restrict-indets-pp } t) + \text{lookup } t \text{ None } = \text{deg-pm } t$
proof
  have $(\text{deg-pm } t) = \text{sum } (\text{lookup } t) (\text{insert } \text{None } (\text{keys } t))$
    by (rule deg-pm-superset)
  auto
  also have $(\ldots = \text{lookup } t \text{ None } + \text{sum } (\text{lookup } t) (\text{keys } t - \{\text{None}\}))$
    by (rule sum.insert-remove)
  also have $(\ldots \subseteq \bigcup t \in \text{keys } p \text{. keys } (\text{monomial }\text{ (lookup } p \text{ } t) \text{ (restrict-indets-pp } t)))$
    by (intro sum.mono-neutral-cong-left) (auto simp: restrict-indets-subst-def deg-pm-single)
  finally show $?thesis$ by simp
qed

lemma keys-restrict-indets-subset: keys $(\text{restrict-indets } p) \subseteq \text{restrict-indets-pp } t$
proof
  fix t
  assume $t \in \text{keys (restrict-indets } p)$
  also have $(\ldots = \text{keys } (\sum t \in \text{keys } p \text{. monomial } (\text{lookup } p \text{ } t) \text{ (restrict-indets-pp } t)))$
    by (simp add: restrict-indets-def poly-subst-def subst-pp-restrict-indets-subst
          punit.monomial-mult-monomial)
  also have $(\ldots \subseteq \bigcup t \in \text{keys } p \text{. keys } (\text{monomial }\text{ (lookup } p \text{ } t) \text{ (restrict-indets-pp } t)))$
    by (rule keys-sum-subset)
also have \ldots = \text{restrict-indets-pp ' keys p} \text{ by (auto split: if-split-asn)}
finally show \( t \in \text{restrict-indets-pp ' keys p} \).
qed

lemma keys-restrict-indets:
assumes None \notin \text{indets p}
shows \( \text{keys (restrict-indets p)} = \text{restrict-indets-pp ' keys p} \)
proof
have \( \text{keys (restrict-indets p)} = \text{keys (} \sum_{t \in \text{keys p. monomial (lookup p t) (restrict-indets-pp t)}} \) \) 
by (simp add: restrict-indets-def poly-subst-def subst-pp-restrict-indets-subst punit.monom-mul-monomial)
also from finite-keys have \( \ldots = ( \bigcup_{t \in \text{keys p. keys (monomial (lookup p t) (restrict-indets-pp t))}} ) \)
proof (rule keys-sum)
fix s t
assume s \in \text{keys p}
hence \( \text{keys s } \subseteq \text{indets p} \) by (rule keys-subset-indets)
with assms have None \notin \text{keys s} \text{ by blast}
assume t \in \text{keys p}
hence \( \text{keys t } \subseteq \text{indets p} \) by (rule keys-subset-indets)
with assms have None \notin \text{keys t} \text{ by blast}
assume s \neq t
then obtain x where neq: \( \text{lookup s x } \neq \text{lookup t x} \) \text{ by (meson poly-mapping-eqI)}
have x \neq None
proof
assume x = None
with \( \text{None } \notin \text{keys s} \) and \( \text{None } \notin \text{keys t} \) have x \notin \text{keys s and x } \notin \text{keys t}
by blast+
with neq show False \text{ by (simp add: in-keys-iff)}
qed
then obtain y where x: \( x = \text{Some y} \) \text{ by blast}
have restrict-indets-pp t \neq restrict-indets-pp s
proof
assume restrict-indets-pp t = restrict-indets-pp s
hence \( \text{lookup (restrict-indets-pp t) y = lookup (restrict-indets-pp s) y} \) \text{ by (simp only:)}
\begin{align*}
\text{hence lookup s x = lookup t x by (simp add: x lookup-restrict-indets-pp)} \\
\text{with neq show False .. .}
\end{align*}
\text{qed}
thus \( \text{keys (monomial (lookup p s) (restrict-indets-pp s)} \cap \\
\text{keys (monomial (lookup p t) (restrict-indets-pp t)) = {\} } \)
by (simp add: subst-pp-extend-indets-subst)
\text{qed}
also have \ldots = \text{restrict-indets-pp ' keys p} \text{ by (auto split: if-split-asn)}
finally show ?thesis .
\text{qed}

lemma indets-restrict-indets-subset: \( \text{indets (restrict-indets p)} \subseteq \text{the ' (indets p --}

510
proof
  fix x
  assume x ∈ indets (restrict-indets p) and x ∈ keys t by (rule in-indetsE)
  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})
  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 ⟨s ∈ keys p⟩ 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:::- ⇒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-mul-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)


**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 (λx. (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-subst-def poly-subst-poly-subst eq)

**Qed**

**Corollary** restrict-indets-comp-dehomogenize: restrict-indets o 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 \( ?\text{thesis} \).

qed

corollary extend-indets-comp-restrict-indets: extend-indets \(\circ\) restrict-indets = dehomogenize None
by (rule ext) (simp only: o-def extend-restrict-indets-eq-dehomogenize)

lemma restrict-homogenize-extend-indets [simp]:
\[ \text{restrict-indets (homogenize None (extend-indets p))} = p \]
proof –
  have \( \text{restrict-indets (homogenize None (extend-indets p))} = \)
  \[ \text{restrict-indets (dehomogenize None (homogenize None (extend-indets p)))} \]
  by (simp only: restrict-indets-dehomogenize)
also have \( \ldots = \text{restrict-indets (dehomogenize None (extend-indets p))} \)
  by (simp only: dehomogenize-homogenize)
also have \( \ldots = p \) by simp
finally show \( ?\text{thesis} \).

qed

lemma dehomogenize-extend-indets [simp]: dehomogenize None (extend-indets p) = extend-indets p
by (simp add: indets-extend-indets)

lemma restrict-indets-ideal:
\[ \text{restrict-indets ' ideal} F = \text{ideal (restrict-indets ' F)} \]
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:
\[ \text{ideal} G = \text{ideal (homogenize None ' extend-indets ' F)} \implies \text{ideal (restrict-indets ' G)} = \text{ideal} F \]
by (simp flip: restrict-indets-ideal) (simp add: restrict-indets-ideal image-image)

lemma extend-indets-ideal:
\[ \text{extend-indets ' ideal} F = \text{ideal (extend-indets ' F)} \cap P[\{-\text{None}\}] \]
proof –
  have \( \text{extend-indets ' ideal} F = \text{extend-indets ' restrict-indets ' ideal (extend-indets ' F)} \)
  by (simp add: restrict-indets-ideal image-image)
also have \( \ldots = \text{ideal (extend-indets ' F)} \cap P[\{-\text{None}\}] \)
  by (simp add: extend-restrict-indets-eq-dehomogenize dehomogenize-ideal image-image)
finally show \( ?\text{thesis} \).

qed

corollary extend-indets-ideal-subset:
\[ \text{extend-indets ' ideal} F \subseteq \text{ideal (extend-indets ' F)} \]
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 ⇒₀ nat) ⇒₀ 'a) ⇒ (('x ⇒₀ nat) ⇒₀ ('x ⇒₀ nat) ⇒₀ 'a::comm-monoid-add)
where focus X p = (∑ t∈keys p. monomial (monomial (lookup p t) (except t X)) (except t (- X)))`

**definition**

`flatten :: ('a ⇒₀ 'a ⇒₀ 'b) ⇒ ('a::comm-powerprod ⇒₀ 'b::semiring-I)
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) = (λt. 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 ... = (λt. except t (- X)) · keys p by (rule UNION-singleton-eq-range)

finally show keys (focus X p) ⊆ (λt. except t (- X)) · keys p.

next

{ fix s
assume s ∈ keys p
have lookup (focus X p) (except s (- X)) = (∑ t∈keys p. monomial (monomial (lookup p t) (except t X) when except t (- 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, hide-lams) 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 \((\text{lookup } p \ t \ \text{when except } \ t \ X = \text{except } s \ X \land \text{except } \ t \ (-X) = \text{except } s \ (-X)) = 0\) by (auto simp: when-def)

next
  from \(s \in \text{keys } p\) show \(\{s\} \subseteq \text{keys } p\) by simp
qed simp-all

also from \(s \in \text{keys } p\) have \(\ldots \neq 0\) by (simp add: in-keys-iff)

finally have \(\text{except } s \ (-X) \in \text{keys } ?p\) by (simp add: in-keys-iff)

moreover assume \(?p = 0\) ultimately show \(\text{False}\) by simp

finally have \(\text{except } s \ (-X) \in \text{keys } \text{focus } X p\) by blast

lemma keys-coeffs-focus-subset:
assumes \(c \in \text{range } (\text{lookup } \text{focus } X p)\)
shows \(\text{keys } c \subseteq (\lambda t. \text{except } t \ (-X)) \text{ keys } p\)
proof
  from assms obtain \(s\) where \(c = \text{lookup } \text{focus } X p\ s\)
  hence \(\text{keys } c = \text{keys } (\text{lookup } \text{focus } X p) s\) by (simp only:)
  also have \(\ldots \subseteq (∪ t \in \text{keys } p. \text{keys } (\text{lookup } (\text{monomial } (\text{monomial } \text{lookup } p \ t) (\text{except } t \ X) (\text{except } t \ (-X))) s))\)
    unfolding \(\text{focus-def } \text{lookup-sum}\) by (rule keys-sum-subset)
  also from \(\subseteq (∪ t \in \text{keys } p. \{\text{except } t \ X\})\)
    by (rule UN-mono) (simp add: lookup-single when-def)
  also have \(\ldots = (\lambda t. \text{except } t \ X) \text{ keys } p\) by (rule \(\text{UNION-singleton-eq-range}\))
  finally show \(\text{thesis}\).
qed

lemma focus-in-Polys' :
assumes \(p \in P[\text{Y}]\)
shows \(\text{focus } X p \in P[\text{Y} \cap X]\)
proof (intro PolysI subsetI)
  fix \(t\)
  assume \(t \in \text{keys } (\text{focus } X p)\)
  then obtain \(s\) where \(s \in \text{keys } p \ \text{and } t = \text{except } s \ (-X)\) unfolding \(\text{keys-focus}\)

  note this \((1)\)
  also from assms have \(\text{keys } p \subseteq [\text{Y}]\) by (rule PolysD)
  finally have \(\text{keys } s \subseteq \text{Y}\) by (rule \(\text{PPsD}\))
  hence \(\text{keys } t \subseteq \text{Y} \cap X\) by (simp add: \(t \text{ keys-}\text{except le-infI1}\))
  thus \(t \in [\text{Y} \cap X]\) by (rule \(\text{PPsI}\))
qed

corollary focus-in-Polys: \(\text{focus } X p \in P[\text{X}]\)
proof
  have \(p \in P[\text{UNIV}]\) 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 $\text{range (lookup (focus X p))} \subseteq \ P[Y - X]$
proof (intro subsetI PolysI)
  fix $c$ $t$
  assume $c \in \text{range (lookup (focus X p))}$
  hence $\text{keys c} \subseteq (\lambda t. \text{except t X})' \ \text{keys p}$ by (rule keys-coeffs-focus-subset)
  moreover assume $t \in \text{keys c}$
  ultimately have $t \in (\lambda t. \text{except t X})' \ \text{keys p}$ ..
  then obtain $s$ where $s \in \text{keys p}$ and $t: t = \text{except s X}$ ..
  note this(1)
  also from assms have $\text{keys p} \subseteq [Y]$ by (rule PolysD)
  finally have $\text{keys s} \subseteq Y$ by (rule PPsD)
  hence $\text{keys t} \subseteq Y - X$ by (simp add: t keys-except Diff-monof)
  thus $t \in [Y - X]$ by (rule PPsI)
qed

corollary focus-coeffs-subset-Polys: $\text{range (lookup (focus X p))} \subseteq \ P[\neg X]$
proof —
  have $p \in \ P[UNIV]$ by simp
  hence $\text{range (lookup (focus X p))} \subseteq \ P[UNIV - X]$ by (rule focus-coeffs-subset-Polys')
  thus ?thesis by (simp only: Compl-eq-Diff-UNIV)
qed

corollary lookup-focus-in-Polys: $\text{lookup (focus X p)} \ t \in \ P[\neg X]$
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]: $focus \ X \ p = 0 \iff p = 0$
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: $focus \ X \ (\text{monomial c t}) = \text{monomial (monomial c (except t X)) (except t (\neg X))}$
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: focus-def punitlookup-monom-mult except-plus times-monomial-monomial
sum-distrib-left)
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. 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 ... = 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) = (∑ 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) = (∏ 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]

517
shows focus X p = (∑ 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 assns 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
(l 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 = (∑ 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 assns 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 \( \text{assms} \) have \( \ldots \subseteq \{Y\} \) by (rule PolysD)
finally have \( \text{keys } t \subseteq Y \) by (rule PPsd)

hence \( \text{keys } t \subseteq X \cup Y \) by blast

hence \( \text{except } t \,(X \cap Y) = \text{except } t \,X + \text{except } t \,Y \) by (rule except-Int)
also from \( \text{keys } t \subseteq Y \) have \( \text{except } t \,Y = \emptyset \) by (rule except-eq-zeroI)

finally have \( \text{eq: except } t \,(X \cap Y) = \text{except } t \,X \) by simp

have \( \text{except } t \,(- \,(X \cap Y)) = \text{except } \text{except } t \,(- \,Y) \) \((\text{simp add: except-except Un-commute})

also from \( \text{keys } t \subseteq Y \) have \( \text{except } t \,(- \,Y) = t \) by (auto simp: except-id-iff)

finally show \( \text{monomial (monomial (lookup p t) \text{except } t \,(X \cap Y)))} \text{except } t \,(- \,(X \cap Y))) = \)

\( \text{monomial (monomial (lookup p t) \text{except } t \,X))} \text{except } t \,(- \,X)) \) by 
(simp only: eq)

qed

lemma \( \text{range-focusD} \):

assumes \( p \in \text{range (focus X)} \)

shows \( p \in P[X] \) and \( \text{range (lookup p) } \subseteq P[- \,X] \) and \( \text{lookup p } t \in P[- \,X] \)

using \( \text{assms by (auto intro: focus-in-Polys lookup-focus-in-Polys)} \)

lemma \( \text{range-focusI} \):

assumes \( p \in P[X] \) and \( \text{lookup p } \cdot \text{keys } (p :: \text{-} \Rightarrow 0 \Rightarrow 0 :: \text{-semiring-I}) \subseteq P[- \,X] \)

shows \( p \in \text{range (focus X)} \)

using \( \text{assms} \)

proof (induct \( p \) rule: \text{poly-mapping-plus-induct-Polys})


case 0

show \( \emptyset \) by simp

next

case \( \text{plus } p \,c \,t \)

from \( \text{plus.hyps}(3) \) have \( 1: \text{keys } (\text{monomial } c \,t) = \{t\} \) by simp

also from \( \text{plus.hyps}(4) \) have \( \ldots \cap \text{keys } p = \{\} \) by simp

finally have \( \text{keys } (\text{monomial } c \,t \,p) = \text{keys } (\text{monomial } c \,t) \cup \text{keys } p \) by (rule \text{keys-add[symmetric]})

hence \( 2: \text{keys } (\text{monomial } c \,t \,p) = \text{insert } t \,(\text{keys } p) \) by (simp only: \( 1 \) \text{flip: insert-is-Un})

from \( \text{plus.hyps}(3,4) \) plus \text{prems} have \( c \in P[- \,X] \) and \( \text{lookup p } \cdot \text{keys } p \subseteq P[- \,X] \)

by (simp-all add: \( 2 \) \text{lookup-add lookup-single in-keys-iff})

(smt \text{add.commute add.right-neutral image-cong plus.hyps(4) when-simps(2)})

from \( \text{this}(2) \) have \( p \in \text{range (focus X)} \) by (rule plus.hyps)

then obtain \( q \) where \( p = \text{focus X } q \) .

moreover from \( c \in P[- \,X] \) have \( \text{monomial } c \,t = \text{focus X } (\text{monomial } 1 \,t \,* c) \)

by (simp add: \text{focus-times focus-monomial eq1 eq2 focus-Polys-Compl times-monomial-monomial})

ultimately have \( \text{monomial } c \,t \,p = \text{focus X } (\text{monomial } 1 \,t \,* c \,q) \) by (simp only: \text{focus-plus})
thus \( ? \text{case by (rule range-eqI)} \)

\textbf{qed}

\textbf{lemma} inj-focus: \( \text{inj } ((\text{focus } X) :: (('x \Rightarrow \text{nat}) \Rightarrow 'a::\text{ab-group-add}) \Rightarrow -) \)

\textbf{proof} (rule injI)

fix \( p q :: ('x \Rightarrow \text{nat}) \Rightarrow 'a \)

assume \( \text{focus } X p = \text{focus } X q \)

hence \( \text{focus } X (p - q) = 0 \) by (simp add: focus-minus)

thus \( p = q \) by simp

\textbf{qed}

\textbf{lemma} flatten-superset:

assumes \( \text{finite } A \) and \( \text{keys } p \subseteq A \)

shows \( \text{flatten } p = (\bigcap t \in A. \text{punit.monom-mult } 1 t \text{ (lookup } p \text{ } t)) \)

\textbf{proof} –

have \( \text{keys } (\text{flatten } p) \subseteq (\bigcup t \in \text{keys } p. (+) t \cdot \text{keys } (\text{lookup } p \text{ } t)) \)

\textbf{proof} \( (\text{rule keys-sum-subset}) \)

also from \( \text{subset-refl} \) have \( \ldots \subseteq (\bigcup t \in \text{keys } p. (+) t \cdot \text{keys } (\text{lookup } p \text{ } t)) \)

by (rule UN-mono) (rule punit.keys-monom-mult-subset[simplified])

finally show \( \text{thesis} \).

\textbf{qed}

\textbf{lemma} flatten-in-Polys:

assumes \( p \in P[X] \) and \( \text{lookup } p \cdot \text{keys } p \subseteq P[Y] \)

shows \( \text{flatten } p \in P[X \cup Y] \)

\textbf{proof} (intro PolysI subsetI)

fix \( t \)

assume \( t \in \text{keys } (\text{flatten } p) \)

also have \( \ldots \subseteq (\bigcup t \in \text{keys } p. (+) t \cdot \text{keys } (\text{lookup } p \text{ } t)) \) by (rule keys-flatten-subset)

finally obtain \( s \) where \( s \in \text{keys } p \) and \( t \in (+) s \cdot \text{keys } (\text{lookup } p \text{ } s) \).

from this(2) obtain \( s' \) where \( s' \in \text{keys } (\text{lookup } p \text{ } s) \) and \( t = s + s' \).

from assms(1) have \( \text{keys } p \subseteq [X] \) by (rule PolysD)

with \( s \in \text{keys } p \) have \( s \in [X] \).

also have \( \ldots \subseteq [X \cup Y] \) by (rule PPs-mono) simp

finally have \( 1: s \in [X \cup Y] \).

from \( s \in \text{keys } p \) have \( \text{lookup } p \in \text{lookup } p \cdot \text{keys } p \) by (rule imageI)

also have \( \ldots \subseteq P[Y] \) by fact

finally have \( \text{keys } (\text{lookup } p \text{ } s) \subseteq [Y] \) by (rule PolysD)

with \( s' \in \cdot \) have \( s' \in [Y] \).

also have \( \ldots \subseteq [X \cup Y] \) by (rule PPs-mono) simp

finally have \( s' \in [X \cup Y] \).

with \( 1 \) show \( t \in [X \cup Y] \) unfolding \( t \) by (rule PPs-closed-plus)

\textbf{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::⇒ 0 -⇒ 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 sum-distrib)
qed

lemma flatten-minus: flatten (p − q) = flatten p − flatten (q::⇒ 0 -⇒ 0 -⇒ ring)
  by (simp only: diff-conv-add-uminus flatten-plus flatten-uminus)

lemma flatten-times: flatten (p * q) = flatten p * flatten (q::⇒ 0 -⇒ 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. ((λt. punit.monom-mult 1 t (lookup (punit.monom-mult c s q) t)) o (+ s) t))
        by (rule sum.reindex) simp
      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-distrib)

521
lemma flatten-monom-mult:
\[ flatten \left( punit \cdot \operatorname{monom-mult} c \ t \ p \right) = punit \cdot \operatorname{monom-mult} 1 \ t \ (c \ast flatten \left( p :: - \Rightarrow 0 \right)) \]
by (simp only: flatten-times flatten-monomial mult.assoc flip: times-monomial-left)

lemma flatten-sum:
\[ flatten \left( \sum f \ I \right) = \left( \sum i \in I. \ flatten \left( f \ i \right) \right) \]
by (induct I rule: infinite-finite-induct) (simp-all add: flatten-plus)

lemma flatten-prod:
\[ flatten \left( \prod f \ I \right) = \left( \prod i \in I. \ flatten \left( f \ i :: - \Rightarrow 0 :: \operatorname{comm-semiring-1} \right) \right) \]
by (induct I rule: infinite-finite-induct) (simp-all add: flatten-times)

lemma flatten-power [simp]:
\[ flatten \left( f \ ^{m} \right) = \left( flatten \left( f :: - \Rightarrow 0 :: \operatorname{comm-semiring-1} \right) \right)^{m} \]
by (induct m) (simp-all add: flatten-times)

lemma surj-flatten:
proof (rule surjI)
fix \ p
show \ flatten \left( \operatorname{monomial} p \ 0 \right) = \ p \ by (simp add: flatten-monomial)
qed

lemma flatten-focus [simp]:
\[ flatten \left( \operatorname{focus} X \ p \right) = \ p \]
by (induct p rule: poly-mapping-plus-induct)
(simp-all add: focus-plus flatten-plus focus-monomial flatten-monomial
\quad \text{punit.monom-mult-monomial add.commute flip: except-decomp})

lemma focus-flatten:
assumes \ p \in \ P[X] \text{ and lookup } p \text{ keys } p \subseteq \ P[-X] 
shows \ operatorname{focus} X \left( flatten \ p \right) = \ p 
proof
from \text{assms have } p \in \text{ range } \left( \operatorname{focus} X \right) \ by \text{ (rule range-focusI)}
then obtain \ q \ where \ p = \operatorname{focus} X \ q ..
thus \ ?thesis \ by \ simp 
qed

lemma image-focus-ideal:
\[ \operatorname{focus} X \ ^{\cdot} \ \operatorname{ideal} F = \operatorname{ideal} \left( \operatorname{focus} X \ ^{\cdot} F \right) \cap \text{range } \left( \operatorname{focus} X \right) \]
proof
from focus-plus focus-times have \operatorname{focus} X \ ^{\cdot} \ \operatorname{ideal} F \subseteq \operatorname{ideal} \left( \operatorname{focus} X \ ^{\cdot} F \right)
by (rule image-ideal-subset)
moreover from subset-UNIV have \operatorname{focus} X \ ^{\cdot} \ \operatorname{ideal} F \subseteq \text{range } \left( \operatorname{focus} X \right) \ by
(rule image-mono)
ultimately show \operatorname{focus} X \ ^{\cdot} \ \operatorname{ideal} F \subseteq \operatorname{ideal} \left( \operatorname{focus} X \ ^{\cdot} F \right) \cap \text{range } \left( \operatorname{focus} X \right) \ by
blast
next
show \operatorname{ideal} \left( \operatorname{focus} X \ ^{\cdot} F \right) \cap \text{range } \left( \operatorname{focus} X \right) \subseteq \operatorname{focus} X \ ^{\cdot} \ \operatorname{ideal} F
proof
  fix p
  assume p ∈ ideal (focus X ' F) ∩ range (focus X)
  hence p ∈ ideal (focus X ' F) and p ∈ range (focus X) by simp-all
  from this(1) obtain F0 q where F0 ⊆ focus X ' F and p = (∑ f'∈F0. q f')
    by (rule ideal.spanE)
  from this(1) obtain F' where F' ⊆ F and 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 ∈ range - obtains where p = focus X p'
  hence p = focus X (flatten p) by simp
  also from inj-on - F' have ... = focus X (∑ f'∈F'. flatten (q (focus X f')))
  * f'
    by (simp add: p F0 sum.reindex flatten-sum flatten-times)
  finally have p = focus X (∑ f'∈F'. flatten (q (focus X f'))) * f'
    moreover have (∑ f'∈F'. flatten (q (focus X f'))) * f' ∈ ideal F
  proof
    show (∑ f'∈F'. flatten (q (focus X 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 ∈ focus X ' ideal F by (rule image-eqI)
  qed

definition image-flatten-ideal: flatten ' ideal F = ideal (flatten ' F)

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
    (subt-pp (λx. monomial (a x) 0) (except t (¬ X))) 0 =
    punit.monom-mult (lookup p t) 0 (subt-pp ?b t) for t
  proof
    --
    have 1: subt-pp ?b (except t X) = monomial 1 (except t X)
      by (rule subt-pp-id) (simp add: keys-except)
  from refl have 2: subt-pp ?b (except t (¬ X)) = subt-pp a (except t (¬ X))
    by (rule subt-pp-cong) (simp add: keys-except)
  have lookup (punit.monom-mult (monomial (lookup p t) (except t X))) 0
    (subt-pp (λx. monomial (a x) 0) (except t (¬ X))) 0 =
    punit.monom-mult (lookup p t) (except t X) (subt-pp a (except t (¬ X)))
    by (simp add: lookup-times-zero subt-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) *
    subt-pp a (except t (¬ X)))

523
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 ⇒0 nat))) q) 0) 0 =
    poly-subst (λy. monomial (a y) (0::'x ⇒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)) ⊆ ∪ (indets ' a ' X) ∪ (indets p − X)

proof
  fix x
  assume x ∈ indets (poly-eval a (focus X p))
  also have ... = indets (poly-subst (λx. if x ∈ 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 ∈ indets p and x ∈ indets (?f y) by (rule in-indets-poly-substE)
from this(2) have (x ∉ X ∧ x = y) ∨ (y ∈ X ∧ x ∈ indets (a y))
  by (simp add: indets-monomial split: if-split-asm)
thus x ∈ ∪ (indets ' a ' X) ∪ (indets p − X)
proof (elim disjE conjE)
  assume x ∉ X and x = y
  with y ∈ indets p. have x ∈ indets p − X by simp
  thus ?thesis ..
next
  assume y ∈ X and x ∈ indets (a y)
  hence x ∈ ∪ (indets ' a ' X) by blast
  thus ?thesis ..
qed
**Lemma** lookup-poly-eval-focus:

\[
\text{lookup (poly-eval } (\lambda x. \text{monomial } (a \ x) 0) \text{ (focus } X \ p)) t = \text{poly-eval } a \ \text{(lookup (focus } (\neg \ X) \ p) t)
\]

**Proof** –

let \( \text{if } \lambda x. \text{if } x \in X \text{ then monomial } (a \ x) 0 \text{ else monomial } 1 \text{ (Poly-Mapping.single } x \ 1) \)

have eq: subst-pp \( \text{if } s = \text{monomial } (\prod x \in \text{keys } s \cap X. \ a \ x \neg \text{lookup } s \ x) \) (except s X) for s

proof –

have subst-pp (\text{if } s = (\prod x \in (\text{keys } s \cap X) \cup (\text{keys } s \neg X). \ ?f x \neg \text{lookup } s \ x)\]

unfolding subst-pp-def by (rule prod.cong) blast+

also have \( \text{if } s = (\prod x \in \text{keys } s \cap X. \ ?f x \neg \text{lookup } s \ x) \ast (\prod x \in \text{keys } s \neg X. \ ?f x \neg \text{lookup } s \ x)\)

by (rule prod.union-disjoint) auto

also have \( \text{if } s = \text{monomial } (\prod x \in \text{keys } s \cap X. \ a \ x \neg \text{lookup } s \ x)\)

by (simp add: monomial-power-map-scale times-monomial-monomial flip: punit.monomial-prod-sum)

also have \( (\sum x \in \text{keys } s \neg X. \text{Poly-Mapping.single } x \text{ (lookup } s \ x))\)

by (metis (mono-tags, lifting) DiffD2 keys-except lookup-except-eq-idI

poly-mapping-sum-monomials sum.cong)

finally show \( \text{thesis } \).

qed

show \( \text{thesis } \)

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:

\[
\text{keys (poly-eval } (\lambda x. \text{monomial } (a \ x) 0) \text{ (focus } X \ p)) \subseteq (\lambda t. \text{except } t \ X) \ \text{'} \text{keys } p
\]

**Proof**

fix t

assume \( t \in \text{keys (poly-eval } (\lambda x. \text{monomial } (a \ x) 0) \text{ (focus } X \ p))\)

hence lookup (poly-eval (\lambda x. \text{monomial } (a \ x) 0) (focus X p)) t \neq 0 by (simp add: in-keys-iff)

hence poly-eval a (lookup (focus (\neg \ X) \ p) t) \neq 0 by (simp add: lookup-poly-eval-focus)

hence \( t \in \text{keys (focus } (\neg \ X) \ p)\) by (auto simp flip: lookup-not-eq-zero-eq-in-keys)

thus \( t \in (\lambda t. \text{except } t \ X) \text{'} \text{keys } p\) by (simp add: keys-focus)

qed

**Lemma** poly-eval-focus-in-Polys:

assumes \( p \in P[X]\)

shows poly-eval (\lambda x. \text{monomial } (a \ x) 0) (focus Y p) \in P[X \neg Y]

**Proof** (rule Polys-alt)

have indets (poly-eval (\lambda x. \text{monomial } (a \ x) 0) (focus Y p)) \subseteq

\( \bigcup (\text{indets } \text{'} (\lambda x. \text{monomial } (a \ x) 0) \text{'} Y) \cup (\text{indets } p \neg Y)\)

525
by (fact indets-poly-eval-focus-subset)
also have \ldots = \text{indets } p - Y \text{ by } \text{simp}
also from \text{assms} have \ldots \subseteq X - Y \text{ by } (\text{auto dest: PolysD})
finally show \text{indets } (\text{poly-eval } (\lambda x. \text{monomial } (a \times) 0) \text{ (focus } Y \text{ } p)) \subseteq X - Y.
qed

\textbf{lemma} image-poly-eval-focus-ideal:
\text{poly-eval } (\lambda x. \text{monomial } (a \times) 0) \text{ ('focus } X ' \text{ ideal } F =
\text{ideal } (\text{poly-eval } (\lambda x. \text{monomial } (a \times) 0) \text{'focus } X ' F) \cap
\text{P[-'X]:{(x \Rightarrow_0 \text{nat }) \Rightarrow_0 'a::\text{comm-ring-1} \text{ set})
proof –
let \(h = \lambda f. \text{poly-eval } (\lambda x. \text{monomial } (a \times) 0) \text{ ('focus } X ' f)
have h-id: \(h \ p = p \text{ if } p \in P[- X] \text{ for} \ p 
proof –
from \text{that have } focus \ X \ p \in P[- X \cap X] \text{ by } (\text{rule focus-in-Polys'})
also have \ldots = P[{}] \text{ by } \text{simp}
finally obtain c where eq: focus \ X \ p = \text{monomial } c 0 \text{ unfolding Polys-empty}
.. hence flatten (focus \ X \ p) = flatten (\text{monomial } c 0) \text{ by } (\text{rule arg-cong})
\text{hence } c = p \text{ by } (\text{simp add: flatten-monomial)}
\text{hence } c = \(h \ p \text{ by } (\text{simp add: eq poly-eval-monomial)}
\text{hence } c = \(p \text{ by } (\text{sym})
\text{thus } c = \(h \ p \text{ by } (\text{rule range-eqI})
\text{hence } c = \(p \text{ by } (\text{simp add: range ?h)}
\text{thus } c = \(p \text{ by } (\text{rule range-eqI})
\text{finally show } c = \(p \text{ by } (\text{rule range-eqI})
\text{qed}

have rng: range ?h = P[- X]
proof (\text{intro subset-antisym subsetI, elim rangeE})
\text{fix } b f
\text{assume } b: b = ?h f 
\text{have } ?h f \in P[UNIV - X] \text{ by } (\text{rule poly-eval-focus-in-Polys') simp
\text{thus } b \in P[- X] \text{ by } (\text{simp add: Compl-eq-Diff-UNIV})
\text{next }
\text{fix } p : (x \Rightarrow_0 \text{nat }) \Rightarrow_0 'a
\text{assume } p \in P[- X]
\text{hence } \(h \ p = p \text{ by } (\text{rule h-id})
\text{hence } p = ?h p \text{ by } (\text{rule sym})
\text{thus } p \in range ?h \text{ by } (\text{rule range-eqI})
\text{qed}

have \text{poly-eval } (\lambda x. \text{monomial } (a \times) 0) \text{'focus } X ' \text{ ideal } F = \(h ' \text{ideal } F \text{ by }
(\text{fact image-image)}
\text{also have \ldots = ideal } (\text{?h ' F) \cap range ?h}
\text{proof (\text{rule image-ideal-eq-Int})}
\text{fix } p
\text{have } ?h p \in range ?h \text{ by } (\text{fact rangeI)}
\text{also have \ldots = P[- X] \text{ by } fact}
\text{finally show } ?h (\text{?h p}) = ?h p \text{ by } (\text{rule h-id})
\text{qed (\text{simp-all only: focus-plus poly-eval-plus focus-times poly-eval-times)}
\text{also have \ldots = ideal } (\text{poly-eval } (\lambda x. \text{monomial } (a \times) 0) \text{'focus } X ' F) \cap P[- X]
\text{ by (\text{simp only: image-image rng)}
\text{finally show ?thesis)}
\text{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 lef ≡ punit.lc
abbreviation tcf ≡ punit.tc
abbreviation lpp ≡ punit.lt
abbreviation tpp ≡ punit.tt

end

locale pm-powerprod =
  ordered-powerprod ord ord-strict
for ord::('x::{countable,linorder} ⇒ 0 nat) ⇒ ('x ⇒ 0 nat) ⇒ bool (infixl ≤ 50)
and ord-strict (infixl < 50)
begin

sublocale gd-powerprod ..

lemma PPs-closed-lpp:
  assumes p : P[X]
  shows lpp p ∈ .[X]
proof (cases p = 0)
  case True
  thus ?thesis by (simp add: zero-in-PPs)
next
  case False
  hence lpp p ∈ keys p by (rule punit.lt-in-keys)
  also from assms have ... ⊆ .[X] by (rule PolysD)
  finally show ?thesis .
qed

lemma PPs-closed-tpp:
  assumes p : P[X]
  shows tpp p ∈ .[X]
proof (cases p = 0)
  case True
  thus ?thesis by (simp add: zero-in-PPs)
next
  case False
  hence tpp p ∈ keys p by (rule punit.tt-in-keys)
  also from assms have ... ⊆ .[X] by (rule PolysD)
  finally show ?thesis .
corollary \PPs\text{-closed-image-lpp}: F \subseteq P[X] \implies \lpp' F \subseteq [X]
by (auto intro: \PPs\text{-closed-lpp})

corollary \PPs\text{-closed-image-tpp}: F \subseteq P[X] \implies \tpp' F \subseteq [X]
by (auto intro: \PPs\text{-closed-tpp})

lemma \text{hom-component-lpp}:
assumes p \neq 0
shows \text{hom-component } p (\deg-pm (\lpp p)) \neq 0 (\text{is } \hat{p} \neq 0)
and \lpp (\text{hom-component } p (\deg-pm (\lpp p))) = \lpp p
proof
from \text{assms} have \lpp p \in \text{keys } p by (rule \text{punit.lt-in-keys})
hence \*: \lpp p \in \text{keys } \hat{p} by (simp add: \text{keys-hom-component})
thus \hat{p} \neq 0 by auto
from \* show \lpp \hat{p} = \lpp p
proof (rule \text{punit.lt-eqI-keys})
fix t
assume t \in \text{keys } \hat{p}
hence t \in \text{keys } p by (simp add: \text{keys-hom-component})
thus t \preceq \lpp p by (rule \text{punit.lt-max-keys})
qed

definition \text{is-hom-ord} :: 'x \Rightarrow bool
where \text{is-hom-ord } x \leftarrow (\forall s t. \deg-pm s = \deg-pm t \rightarrow (s \preceq t \iff \text{except } s \{x\} \preceq \text{except } t \{x\}))

lemma \text{is-hom-ordD}:: \text{is-hom-ord } x \implies \deg-pm s = \deg-pm t \implies s \preceq t \iff \text{except } s \{x\} \preceq \text{except } t \{x\}
by (simp add: \text{is-hom-ord-def})

lemma \text{dgrad-p-set-varnum}:: \text{punit.dgrad-p-set } (\text{varnum } X) 0 = P[X]
by (simp add: \text{punit.dgrad-p-set-def dgrad-set-varnum Polys-def})

end

We must create a copy of \text{pm-powerprod} to avoid infinite chains of interpretations.

instantiation \text{option} :: (linorder) \text{linorder}
begin

fun \text{less-eq-option} :: 'a \text{option} \Rightarrow 'a \text{option} \Rightarrow bool
where
\text{less-eq-option } None \cdot = \text{True} |
\text{less-eq-option } (\text{Some } x) \text{ None} = \text{False} |
\text{less-eq-option } (\text{Some } x) (\text{Some } y) = (x \preceq y)

528
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 ⇒ 0 nat) ⇒ ('a option ⇒ 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 ⇒ 0 nat) ⇒ ('a option ⇒ 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 ⇒ 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)
  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)
  finally show ?thesis .
qed

qed

have 2: 0 ≺ t if t ≠ 0 for t::'a ⇒ 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)
lemma extended-ord-is-hom-ord: extended-ord.is-hom-ord None

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 (λm. (coeff p (Poly-Mapping.lookup m v)) when Poly-Mapping.keys m ⊆ {v}))

lemma poly-to-mpoly-finite: finite {m::nat ⇒0 nat. (coeff p (Poly-Mapping.lookup m v)) when Poly-Mapping.keys m ⊆ {v}} (is finite ?M)
  proof –
    have ?M ⊆ Poly-Mapping.single v · {x. Polynomial.coeff p x ≠ 0}
      proof
        fix m assume m ∈ ?M
        then have \(\forall v'. v'≠v \rightarrow Poly-Mapping.lookup m v' = 0\) 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
  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 (λk. 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 ∀ 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
  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 = (λm. 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-eq1 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

531
unfolding mpol-to-poly-def coeff-poly-to-mpoly by (simp add: coeff-inverse)

lemma poly-to-mpoly0: poly-to-mpoly v 0 = 0
proof -
  have \( \forall m. (\text{Polynomial.coeff} \ 0 (\text{Poly-Mapping.lookup} \ m \ v) \ \text{when \ Poly-Mapping.keys} \ m \subseteq \{v\}) = 0 \) by simp
  have Abs-poly-mapping (\( \lambda m. \text{Polynomial.coeff} \ 0 (\text{Poly-Mapping.lookup} \ m \ v) \) when Poly-Mapping.keys m \subseteq \{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 mpol-to-poly-add: mpol-to-poly v (p1 + p2) = mpol-to-poly v p1 + mpol-to-poly v p2
unfolding Polynomial.plus-poly.abs-eq More-MPoly-Type coeff-add coeff-mpoly-to-poly
using mpol-to-poly-def by auto

lemma poly-eq-insertion:
assumes vars p \subseteq \{v\}
shows poly (mpoly-to-poly v p) x = insertion (\( \lambda v. x \)) 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 \subseteq \{v\} using monom unfolding vars-def MPoly-Type.monom.abs-eq insertion-zero monom-zero poly-0
  unfolding poly-to-mpoly0 poly-to-mpoly-inverse single-zero
  using insertion-single by auto
then have 0:insertion (\( \lambda v. x \)) (MPoly-Type.monom m a) = a * x ^ (Poly-Mapping.lookup m v)
  unfolding monom unfolding single by metis
  have \( \forall k. \text{Poly-Mapping.single} \ v \ k = m \Longleftrightarrow \text{Poly-Mapping.lookup} \ m \ v = k \)
  using m = Poly-Mapping.single v (Poly-Mapping.lookup m v)
  unfolding single_eq lookup-single not_eq poly-mapping-eqI
  by (metis lookup-single-eq lookup-single_not_eq poly-mapping-eqI)
  then have 0:insertion (\( \lambda v. x \)) (MPoly-Type.monom m a) = a * x ^ (Poly-Mapping.lookup m v)
  using insertion-single by metis
  have \( \forall k. \text{Poly-Mapping.single} \ v \ k = m \Longleftrightarrow \text{Poly-Mapping.lookup} \ m \ v = k \)
  using m = Poly-Mapping.single v (Poly-Mapping.lookup m v)
  by auto
  then have monom a (Poly-Mapping.lookup m v) = (Abs-poly (\( \lambda k. \text{if Poly-Mapping.single} \ v \ k = m \ \text{then \ a \ else} \ 0 \))
  then have ?thesis unfolding coeff-mpoly-to-poly-def More-MPoly-Type coeff-monom
  0 when-def by (metis poly-monom)
qed
next

532
case (sum p1 p2 m a)
then have poly (mpoly-to-poly v p1) x = insertion (λv. x) p1
  poly (mpoly-to-poly v p2) x = insertion (λv. 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 (λv. 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 \(\geq 1\) and each variable occurs at most once
  hence: \([(x,1),(y,2),(x,2)]\) will not occur, but \([(x,3),(y,2)]; [(x,1),(y,0)]\)
  will not occur, but \([(x,1)]\)

class 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)

to 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

534
show \(?\)case
proof (cases \(n\))
case (Cons \(yy\) \(nn\))
  obtain \(y\) \(p\) where \(yy\) \(\equiv\) \((y, p)\) by (cases \(yy\), auto)
  with Cons Nil (unfolded monom-inv-def) have \(p \prec q\) by auto
  show ?thesis by (simp add: Cons, rule exI [of - \(y\)], simp add: sum-var-list-def \(yy\) \(p\))
qed simp

next

case (Cons \(xp\) \(m\))
  obtain \(x\) \(p\) where \(xp\) \(\equiv\) \((x, p)\) by (cases \(xp\), auto)

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 \(yy\) \(n\))
from Cons (3) [unfolded \(n\)] have \(n' : monom-inv n'\) by (auto simp: monom-inv-def)
show ?thesis
proof (cases \(yy\) \(\equiv\) \(xp\))
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 \(yy\) \(\equiv\) \((y, q)\) by force
from Cons (3) [unfolded \(n\) \(yy\) monom-inv-def] have \(q : q \succ 0\) by auto
  define \(z\) where \(z \equiv\) \(\min x y\)
  have \(zm\) \(\equiv\) \(\min x y\) using Cons (2) unfolding \(xp\) \(z\)-def
by (rule monom-inv-Cons, simp)
  have \(zn': z \notin \text{fst } \text{set } m\) using Cons (3) unfolding \(n\) \(yy\) \(z\)-def
by (rule monom-inv-Cons, simp)
  have \(smn\) \(\equiv\) \(\text{sum-var-list} (\(xp\) \# \(m\)) \equiv \text{sum-var-list} [(x, p)]\)\(z\)
using sum-var-list-not [OF \(zm\)] by (simp add: sum-var-list-def \(xp\))
  also have \(\ldots\) \(\notin\) 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 \(\text{sum-var-list} [(y, q)] \equiv \text{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 simp

equality of monomials is also a complete for several carriers, e.g. the naturals, integers, where \(x^p = x^q\) implies \(p = q\). note that it is not complete for carriers like the Booleans where e.g. \(x^{\text{Suc}(m)} = x^{\text{Suc}(n)}\) for all \(n, m\).
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 ?thesis
  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 ∉ fst ' set m
  and xz: \ z. z ∈ fst ' set m ⇒ x ≤ z
  unfolding monom-inv-def by (auto)
  from n'[unfolded id] have n: monom-inv n and q: q > 0 and y: y ∉ fst ' set n
  and yz: \ z. z ∈ fst ' set n ⇒ y ≤ 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 xz yz
  by (fastforce simp add: monom-inv-def monom-list-mult-list-vars)
  next
  case False
show \( \text{thesis} \)
proof (cases \( x < y \))
case True
  hence res: monom-mult-list \( ((x, p) \# m) \) \( n' = (x, p) \# \text{monom-mult-list} \) \( m \)
    by (auto simp add: id)
from IH(2)[OF refl False True m n'] have inv: monom-inv (monom-mult-list \( m \) \( n') \).
  show \( \text{thesis} \) unfolding res using inv \( p \) \( x \) \( y \) True \( x \) \( m \) \( n \) 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 \( ((x, p) \# m) \) \( n' = (y, q) \# \text{monom-mult-list} \) \( ((x, p) \# m) \) \( n \)
    using False by (auto simp add: id)
  from lt have zm: \( z \leq x \implies (z, b) \notin \text{set} m \) for \( z \) \( b \) by force
  from zm[of y] lt have gm: \( (y, b) \notin \text{set} m \) for \( b \) by auto
  from gm have yn': \( (a, b) \in \text{set} n \implies y \leq a \) for \( a \) \( b \) by force
  from IH(3)[OF 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': \( \text{fst ' set} xpm = \text{insert} x \) \( (\text{fst ' set} m) \)
    unfolding xpm-def by auto
  show \( \text{thesis} \) unfolding res using inv \( p \) \( q \) \( x \) \( y \) False \( gt \) \( ym \) \( lt \) \( x \) \( m \) \( ym' \) \( zm \)
    unfolding id xpm-def[ symmetric ] by (auto simp add: monom-inv-def monom-list-mult-list-vars)
qed
qed
qed

lemma monom-inv-ConsD: monom-inv \( x \) \( \# xs \) \( \implies \) monom-inv \( xs \)
  by (auto simp: monom-inv-def)

lemma sum-var-list-monom-mult-list: \( \text{sum-var-list} \) \( (\text{monom-mult-list} \ m \) \( n \) \( ) \) \( x = \text{sum-var-list} \ m \) \( x + \text{sum-var-list} \ n \) \( x \)
proof (induct \( m \) \( n \) rule: monom-mult-list.induct)
case (2 \( x \) \( p \) \( m \) \( n \))
  thus \( \text{case} \ by \ (\text{cases} \ n; \ \text{cases} \ \text{hd} \ n, \ \text{auto} \ \text{split:} \text{if-splits simp:} \text{sum-var-list-def}) \)
qed (auto simp: sum-var-list-def)

lemma monom-mult-list-inj: assumes \( m: \text{monom-inv} \) \( m \) \( \text{and} \) \( m1: \text{monom-inv} \) \( m1 \)
  and \( m2: \text{monom-inv} \) \( m2 \)
  and \( \text{eq:} \) \( \text{monom-mult-list} \ m \) \( m1 = \text{monom-mult-list} \ m \) \( m2 \)
  shows \( m1 = m2 \)
proof –
  from eq \( \text{sum-var-list-monom-mult-list[of} \ m \) \( ] \) \( \text{show} \ \text{thesis} \)

537
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 = 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 \( \text{poly-inv} \; p \equiv (\forall \; c \in \operatorname{snd} \; \text{set} \; p. \; c \neq 0) \land \text{distinct} \; (\operatorname{map} \; \operatorname{fst} \; p) \)

abbreviation \textbf{eval-monomc} \quad \textbf{where} \quad \text{eval-monomc} \quad mc \equiv \text{eval-monom} \quad (\text{fst} \; mc) * (\text{snd} \; mc)

primrec \textbf{eval-poly} :: 'v :: \text{linorder}, 'a :: \text{comm-semiring-1} \Rightarrow 'v, 'a\text{poly} \Rightarrow 'a\text{poly}
| \text{eval-poly} \quad \alpha \; [] = 0
| \text{eval-poly} \quad \alpha \; (mc \# \; p) = \text{eval-monomc} \quad \alpha \; mc + \text{eval-poly} \quad \alpha \; p

definition \textbf{poly-const} :: 'a :: \text{zero} \Rightarrow ('v :: \text{linorder}, 'a)\text{poly} \quad \textbf{where}
\text{poly-const} \; a = (\text{if} \; a = 0 \; \text{then} \; [] \; \text{else} \; [(1, a)])

lemma \textbf{poly-const}[simp]: \text{eval-poly} \quad \alpha \; (\text{poly-const} \; a) = a
unfolding \text{poly-const-def} \quad \text{by} \quad \text{auto}

lemma \textbf{poly-const-inv}: \text{poly-inv} \; (\text{poly-const} \; a)
unfolding \text{poly-const-def} \quad \text{poly-inv-def} \quad \text{by} \quad \text{auto}

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

lemma \textbf{eval-poly-append}[simp]: \text{eval-poly} \quad \alpha \; (mc1 \# mc2) = \text{eval-poly} \quad \alpha \; mc1 + \text{eval-poly} \quad \alpha \; mc2
by (\text{induct} \; mc1, \; \text{auto} \; \text{simp}: \text{field-simps})

abbreviation \textbf{poly-monomcs} :: ('v,'a)\text{poly} \Rightarrow 'v \text{ monom set}
where \text{poly-monomcs} \; p \equiv \text{fst} \; \text{set} \; p

lemma \textbf{poly-add-monomcs}: \text{poly-monomcs} \; (\text{poly-add} \; p1 \; p2) \subseteq \text{poly-monomcs} \; p1 \cup \text{poly-monomcs} \; p2
proof (\text{induct} \; p1 \; \text{arbitrary}: \; p2)
case (\text{Cons} \; mc \; p)
\text{obtain} \; m \; c \quad \textbf{where} \quad mc = (m, c) \quad \textbf{by} \quad (\text{cases} \; mc, \; \text{auto})
\text{hence} \; m: \; m \in \text{poly-monomcs} \; (mc \# \; p1) \quad \textbf{by} \quad \text{auto}
\text{show} \; ?\text{case}
proof (\text{cases} \; \text{List.extract} \; (\lambda \; nd. \; \text{fst} \; nd = m) \; p2)
case \text{None}
\text{with} \; \text{Cons} \; m \quad \text{show} \; \text{?thesis} \quad \textbf{by} \quad (\text{auto} \; \text{simp}: \; mc)
next
case (\text{Some} \; \text{res})
\text{obtain} \; q1 \; md \; q2 \quad \textbf{where} \quad \text{res} = (q1, md, q2) \quad \textbf{by} \quad (\text{cases} \; \text{res}, \; \text{auto})
\text{from} \; \text{extract}-\text{Some} \; [\text{OF} \; \text{Some}[\text{simplified} \; \text{res}]] \quad \text{\texttt{res} obtain} \; d \quad \textbf{where} \quad q: \; p2 = q1 \at (m, d) \# \; q2 \quad \text{and} \quad \text{res} = (q1, (m, d), q2) \quad \textbf{by} \quad (\text{cases} \; md, \; \text{auto})

540
show ?thesis
  by (simp add: mc Some res, rule subset-trans[OF Cons[of q1 @ q2]], auto simp: q)
qed

lemma poly-add-inv: poly-inv p \Longrightarrow poly-inv q \Longrightarrow 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 \neq 0 and mp: \forall mm \in fst ' set p. (\neg mm = m) unfolding poly-inv-def by auto
show ?thesis
proof (cases List.extract (\lambda mc. fst mc = m) q)
case None
  hence mq: \forall mm \in fst ' set q. \neg mm = m by (auto simp: extract-None-iff)
  { fix mn
    assume mn \in fst ' set (poly-add p q)
    then obtain dd where (mn,dd) \in set (poly-add p q) by auto
    with poly-add-monoms have mn \in poly-monoms p \lor mn \in poly-monoms q by force
    hence \neg mn = 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 = (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 mn
    assume mn \in fst ' set (poly-add p (q1 @ q2))
    then obtain dd where (mn,dd) \in set (poly-add p (q1 @ q2)) by auto
    with poly-add-monoms have mn \in poly-monoms p \lor mn \in poly-monoms (q1 @ q2) by force
    hence mn \neq m
    proof
      assume mn \in poly-monoms p
      thus ?thesis using mp by auto
    next
      assume member: mn \in poly-monoms (q1 @ q2)
      from member have mn \in poly-monoms q1 \lor mn \in poly-monoms q2 by auto
  }
thus \( mm \neq m \)

proof

assume \( mm \in \text{poly-monoms } q2 \)

with \( \text{Cons}(3)[\text{simplified } q] \)

show \( \text{?thesis unfolding poly-inv-def by auto} \)

next

assume \( mm \in \text{poly-monoms } q1 \)

with \( \text{Cons}(3)[\text{simplified } q] \)

show \( \text{?thesis unfolding poly-inv-def by auto} \)

qed

qed

\}

note \( \text{main2 = this} \)

show \( \text{?thesis} \)

using \( \text{main1 [unfolded poly-inv-def]} \)

by \( (\text{auto simp: poly-inv-def mc Some res}) \)

qed

qed simp

lemma \( \text{poly-add[simp]: eval-poly } \alpha (\text{poly-add } p q) = \text{eval-poly } \alpha p + \text{eval-poly } \alpha q \)

proof \( (\text{induct } p \text{ arbitrary: } q) \)

\text{case } (\text{Cons } mc p)

obtain \( m c \) \text{ where } \( mc = (m,c) \) by \( (\text{cases } mc, \text{ auto}) \)

show \( \text{?case} \)

proof \( (\text{cases List.extract } (\lambda mc. \text{ fst } mc = m) \ q) \)

case None

show \( \text{?thesis} \) by \( (\text{simp add: Cons[of } q \text{] mc None field-simps}) \)

next

case (\text{Some } res)

obtain \( q1 md q2 \) \text{ where } \( \text{res = (}q1,md,q2\text{)} \) by \( (\text{cases } res, \text{ auto}) \)

from \( \text{extract-}\text{SomeE[OF Some[simplified res]] } \text{res obtain } d \text{ where } q: q = \text{q1 \at \( (m,d) \neq q2 \text{ and } \text{res = (}q1,(m,d),q2\text{)} \) by \( (\text{cases } md, \text{ auto}) \)}\)

\{ \text{fix } x \)

\text{assume } c: \( c + d = 0 \)

\text{have } c * x + d * x = (c + d) * x \text{ by } \( (\text{auto simp: field-simps}) \)

\text{also have } \text{... = } 0 \ast x \text{ by } \( \text{simp only: } c \)

\text{finally have } c \ast x + d \ast x = 0 \text{ by simp} \)

\} \text{ note } id = \text{this} \)

show \( \text{?thesis} \)

by \( (\text{simp add: Cons[of } q1 \at q2\text{] mc Some res, simp only: } q, \text{ simp add: field-simps, auto simp: field-simps id}) \)

qed

qed simp

declare \( \text{poly-add.sims}[\text{simp del}] \)

fun \( \text{monom-mult-poly :: } ('v :: \text{linorder monom} \times 'a) \Rightarrow ('\text{'v,'a :: semiring-0} \text{)poly} \)

\Rightarrow ('\text{'v,'a} \text{poly} \text{ where})

\text{monom-mult-poly} - [] = []

| \text{monom-mult-poly} \text{ (m,c) } \text{ ((m',d) \neq p)} = (\text{if } c \ast d = 0 \text{ then monom-mult-poly}\)
\[(m, c) \text{ if else } (m \ast m', c \ast d) \# \text{ monom-mult-poly } (m, c) \text{ p}\]

**Lemma** monom-mult-poly-inv: poly-inv p \(\Rightarrow\) 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 (monom-mult-poly (m, c) p) .

\{
  fix mm
  assume mm \(\in\) fst ' set (monom-mult-poly (m, c) p)
  and two: \(mm = m \ast m'\)
  **then** obtain \(dd\) where \(one: (mm, dd) \in set (monom-mult-poly (m, c) p)\) by auto
  \begin{align*}
    \text{have } & \text{poly-monom-s } (\text{monom-mult-poly } (m, c) p) \subseteq (\ast) m ' \text{poly-monom-s } p \\
    \text{proof } & (\text{induct } p, \text{ simp}) \\
    \text{case } & (\text{Cons } md p) \\
    \text{thus } & ?case \\
    & \text{by } (\text{cases } md, \text{ auto})
  \end{align*}
  \end{align*}
  **qed**

  with one **have** m \(\in\) (\ast) m ' poly-monom-s p by force

  **then** obtain \(mmm\) where \(mmm: mmm \in \text{poly-monom-s } p\) and \(mm: mm = m \ast mmm\) by blast

  **from** Cons(2)[simplified md] \(mmm\) **have** not1: \(\neg mmm = m'\) unfolding poly-inv-def by auto

  **from** m m two **have** m ' mmm = m ' m' by simp

  **from** monom-mult-inj[OF this] not1

  **have** False by simp

  \}

**thus** ?case

\begin{align*}
  \text{by } (\text{simp add: } md \text{ prod, intro impI, auto simp: poly-inv-def prod[simplified poly-inv-def])}
  \end{align*}

**qed**

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

**Proof** (cases mc)

*case* (Pair m c)

**show** ?thesis

**Proof** (simp add: Pair, induct p)

*case* (Cons nd g)

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**
let \( ?l = c \ast (d \ast (\text{eval-monom} \alpha m \ast \text{eval-monom} \alpha n)) \)

have \( ?l = (c \ast d) \ast (\text{eval-monom} \alpha m \ast \text{eval-monom} \alpha n) \)
  by (simp only: field-simps)
also have \( \ldots = 0 \) by (simp only: True, simp add: field-simps)
finally have \( l: ?l = 0 \).
show \( \alpha \)
  by (simp add: nd Cons True, simp add: field-simps)
finally have \( l: ?l = 0 \).

show \( \alpha \)
  by (simp add: nd Cons True, simp add: field-simps)

definition \( \text{poly-minus} :: \langle 'v :: \text{linorder}, 'a :: \text{ring-1} \rangle \text{poly} \Rightarrow \langle 'v, 'a \rangle \text{poly} \Rightarrow \langle 'v, 'a \rangle \text{poly} \)
where
\[
\text{poly-minus} f g = \text{poly-add} f (\text{monom-mult-poly} (1,-1) g)
\]

lemma \( \text{poly-minus} \)
  shows \( \text{eval-poly} \alpha f - \text{eval-poly} \alpha g \)
  unfolding \( \alpha \)
  by simp

lemma \( \text{poly-minus-inv} \)
  assumes \( p: \text{poly-inv} p \) and \( q: \text{poly-inv} q \)
  shows \( \text{poly-inv} (\text{poly-mult} p q) \)
  using \( p \)
proof (induct \( p \))
  case \( \text{Nil} \) 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 \( \text{Cons}(2) \) have \( p: \text{poly-inv} p \) unfolding \( \alpha \)
  by auto
  show \( ?case \)
    by (simp add: \( mc \), rule \( \text{poly-add-inv} \text{of monom-mult-poly}\text{of poly}\text{of cons}(1)\text{of poly}\text{of \( p \)}))
qed

lemma \( \text{poly-mult} \)
  shows \( \text{eval-poly} \alpha (\text{poly-mult} p q) = \text{eval-poly} \alpha p \ast \text{eval-poly} \alpha q \)
  by (induct \( p \), auto simp: field-simps)

declare \( \text{poly-mult.simps} \)
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
  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-mult-comm: poly-mult p q =p poly-mult q p 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)poly ⇒ (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: 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 (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 .

546
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
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, 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
    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 eq: ∀ 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: ∀ 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

548
lemma eval-monom-vars-list: assumes \( \land x. x \in \text{set} \ (\text{monom-vars-list} \ xs) \implies \alpha \ x = \beta \ x \)
shows \( \text{eval-monom} \ \alpha \ xs = \text{eval-monom} \ \beta \ xs \) using assms
proof (transfer fixing: \( \alpha \ \beta \))
fix xs :: 'a \ monom-list
assume eq: \( \land w. w \in \text{set} \ (\text{map} \ \text{fst} \ xs) \implies \alpha \ w = \beta \ w \)
thus eval-monom-list \( \alpha \ xs \) = eval-monom-list \( \beta \ xs \)
proof (induct xs)
case (Cons \( xi \) \( xs \))
hence IH: eval-monom-list \( \alpha \ xs \) = eval-monom-list \( \beta \ xs \) by auto
obtain \( x \ i \) where \( xi :: (x,i) \) by force
from Cons(2) xi have \( \alpha \ x = \beta \ x \) by auto
with IH show ?case unfolding \( xi \) by auto
qed simp
qed

definition monom-vars where monom-vars \( m ::= \) set \( (\text{monom-vars-list} \ m) \)

lemma monom-vars-list-1[simp]: \( \text{monom-vars-list} \ 1 = [] \)
by transfer auto

lemma monom-vars-list-var-monom[simp]: \( \text{monom-vars-list} \ (\text{var-monom} \ x) = [x] \)
by transfer auto

lemma monom-vars-eval-monom:
(\( \land x. x \in \text{monom-vars} \ m \implies f \ x = g \ x \) \( \implies \text{eval-monom} \ f \ m = \text{eval-monom} \ g \ m \))
by (rule eval-monom-vars-list, auto simp: monom-vars-def)

definition poly-vars-list :: ('v :: linorder,'a)poly \( \Rightarrow \) 'v list where

poly-vars-list \( p ::= \) remdups \( \text{concat} \ (\text{map} \ (\text{monom-vars-list} \circ \ \text{fst}) \ p) \)

definition poly-vars :: ('v :: linorder,'a)poly \( \Rightarrow \) 'v set where

poly-vars \( p ::= \) set \( \text{concat} \ (\text{map} \ (\text{monom-vars-list} \circ \ \text{fst}) \ p) \)

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

lemma poly-vars: assumes eq: \( \land w. w \in \text{poly-vars} \ p \implies f \ w = g \ w \)
shows poly-subst \( f \ p = \text{poly-subst} \ g \ p \)
using eq
proof (induct \( p \))
case (Cons \( mc \) \( p \))
hence rec: poly-subst \( f \ p = \text{poly-subst} \ g \ p \) unfolding poly-vars-def by auto
show ?case
proof (cases mc)
  case (Pair m c)
    with Cons(2) have \( \forall w. w \in \text{set (monom-vars-list m)} \rightarrow f w = g w \) 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 simp

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

lemma eval-poly-vars: assumes \( \forall x. x \in \text{poly-vars p} \rightarrow \alpha x = \beta x \)
  shows eval-poly \( \alpha p = \text{eval-poly} \beta p \)
using assms
proof (induct p)
  case Nil thus ?case by simp
next
  case (Cons m p)
  from Cons(2) have \( \forall x. x \in \text{poly-vars p} \rightarrow \alpha x = \beta x \) unfolding poly-vars-def by auto
    from Cons(1)[OF this] have IH: eval-poly \( \alpha p = \text{eval-poly} \beta p \).
    obtain xs c where m: \( m = (xs, c) \) by force
    from Cons(2) have \( \forall x. x \in \text{set (monom-vars-list xs)} \rightarrow \alpha x = \beta x \) unfolding poly-vars-def m by auto
      hence eval-monom \( \alpha xs = \text{eval-monom} \beta xs \)
        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 :: (\( 'v, 'a :: \text{ordered-semiring-0} \))assign \( \Rightarrow \text{bool} \)
where pos-assign \( \alpha = (\forall x. \alpha x \geq 0) \)

definition poly-ge :: (\( 'v :: \text{linorder}, 'a :: \text{poly-carrier} \))poly \( \Rightarrow (\forall 'v, 'a)\text{poly} \rightarrow \text{bool} \)
(infix \( \gep \ 51 \))
where \( p \gep q = (\forall \alpha. \text{pos-assign} \alpha \rightarrow \text{eval-poly} \alpha p \geq \text{eval-poly} \alpha q) \)
lemma poly-ge-refl[simp]: \( p \geq p \)

unfolding poly-ge-def using ge-refl by auto

lemma poly-ge-trans[trans]: \([p1 \geq p p2; p2 \geq p p3] \implies p1 \geq p p3\]

unfolding poly-ge-def using ge-trans by blast

lemma pos-assign-monom-list: fixes \( \alpha :: ('v :: linorder, 'a :: poly-carrier)assign \)

assumes pos: pos-assign \( \alpha \)

shows eval-monom-list \( \alpha m \geq 0 \)

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 \( ge: \alpha x \geq 0 \) by simp
    have \( ge: \alpha x \cdot p \geq 0 \)
    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 \( ge\) Succ] 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 \( \alpha :: ('v :: linorder, 'a :: poly-carrier)assign \)

assumes pos: pos-assign \( \alpha \)

shows eval-monom \( \alpha m \geq 0 \)

by (transfer fixing: \( \alpha \), rule pos-assign-monom-list[OF \( pos\)])

lemma pos-assign-poly: assumes pos: pos-assign \( \alpha \)

and \( p: p \geq p\) zero-poly

shows eval-poly \( \alpha p \geq 0 \)

proof -
  from \( p\)[unfolded poly-ge-def zero-poly-def] \( pos\)
  show \(?thesis\) by auto

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

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

context poly-order-carrier

begin

definition poly-gt :: (′v::linorder,′a)poly ⇒ (′v,′a)poly ⇒ bool (infix > 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 ≥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 q ∧ q ≥p zero-poly)

lemma poly-compat: [p1 ≥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 ≥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] =⇒ 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: ∀ i. (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)

552
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 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
gc: 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 :: ('v,'a)assign and x
  show pos-assign α ⇒ eval-poly α (f x) ≥ eval-poly α (g x) using gc[of x]
unfolding poly-ge-def by auto
next
  fix α :: ('v,'a)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 gc[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 fgw: 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 fgw 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 fgw: 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 fgw 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-def zero-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-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 = λvs 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
  }
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 \( \ldots \geq 0 \) by (rule pos-at-zero)

finally show \( \text{thesis} \) by (simp add: poly-subst)

qed

context poly-order-carrier

begin

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

lemma poly-strict-mono-E: assumes p: poly-strict-mono p v
    and fgw: \( \land w. v \neq w \rightarrow f w = g w \)
    and q: \( \land w. g w \geq p \) zero-poly
    and fgv: \( f v > p \) g v
    shows poly-subst f p > p poly-subst g p

proof (intro allI impI, rule p[unfolded poly-strict-mono-def, rule-format])

fix \( \alpha :: ('v, 'a)assign \)

show pos-assign \( \alpha \) \rightarrow eval-poly \( \alpha (f v) \succ eval-poly \alpha (g v) \) using fgv unfolding poly-gt-def by auto

next

fix \( \alpha :: ('v, 'a)assign \) and \( x \)

assume alpha: pos-assign \( \alpha \)

show pos-assign (\( \lambda v. eval-poly \alpha (g v) \)) unfolding pos-assign-def

proof

fix \( x \)

show eval-poly \( \alpha (g x) \geq 0 \)

using g[of x] unfolding poly-ge-def zero-poly-def using alpha by auto

qed

next

fix \( \alpha :: ('v, 'a)assign \) and \( x \)

assume v: \( v \neq x \)

show pos-assign \( \alpha \) \rightarrow eval-poly \( \alpha (f x) = eval-poly \alpha (g x) \) using fgv[OF v]

unfolding poly-ge-def by auto

qed

lemma poly-add-gt-mono: assumes p1 \( \succ p \) p2 shows poly-add p1 q \( \succ p \) 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 \( \succ p \) p2 and mono: q \( \geq p \) one-poly

shows poly-mult p1 q \( \succ p \) poly-mult p2 q

proof (unfold poly-gt-def, intro impI allI)

fix \( \alpha :: ('v, 'a)assign \)
assume \( p \) \( \text{pos-assign} \alpha \)
with \( \text{gt} \) have \( \text{gt} \): \( \text{eval-poly} \alpha \; p1 \succ \text{eval-poly} \alpha \; p2 \) unfolding \( \text{poly-gt-def} \) by simp

from \( \text{mono} \; p \) have one: \( \text{eval-poly} \alpha \; q \geq 1 \) unfolding \( \text{poly-ge-def} \) one-poly-def by auto

show \( \text{eval-poly} \alpha \; (\text{poly-mult} \; p1 \; q) \succ \text{eval-poly} \alpha \; (\text{poly-mult} \; p2 \; q) \)
using \( \text{times-gt-mono} \) \( \text{OF} \) \( \text{gt} \) one by simp

qed
end

18.6 Degree of polynomials

definition \( \text{monom-list-degree} :: \langle \text{monom-list} \Rightarrow \text{nat} \rangle \) where
\( \text{monom-list-degree} \; \text{xps} \equiv \text{sum-list} \; (\text{map} \; \text{snd} \; \text{xps}) \)

lift-definition \( \text{monom-degree} :: \langle \text{linorder monom} \Rightarrow \text{nat} \rangle \) is \( \text{monom-list-degree} \).

definition \( \text{poly-degree} :: (\text{poly} \Rightarrow \text{nat}) \) where
\( \text{poly-degree} \; p \equiv \text{max-list} \; (\text{map} \; (\lambda (\text{m}, \text{c}). \text{monom-degree} \; \text{m}) \; p) \)

definition \( \text{poly-coeff-sum} :: (\langle \text{ordered-ab-semigroup} \rangle \text{poly} \Rightarrow \text{nat}) \) where
\( \text{poly-coeff-sum} \; p \equiv \text{sum-list} \; (\text{map} \; (\lambda \text{mc}. \text{max} 0 \; (\text{snd} \; \text{mc})) \; p) \)

lemma mono-list-degree: \( \text{eval-monom-list} \; (\lambda \cdot. \text{x}) \; m = \text{x} ^{\text{monom-list-degree} \; \text{m}} \)
unfolding \( \text{monom-list-degree-def} \)
proof (induct \( \text{m} \))
case Nil show \( ?\)case by simp

next
case (Cons \( \text{mc} \; \text{m} \))

thus ?case by (cases \( \text{mc} \), auto simp: power-add field-simps)

qed

lemma monom-list-var-monom[simp]: \( \text{monom-list} \; (\text{var-monom} \; \text{x}) \; m = [(\text{x}, 1)] \)
by (transfer, simp)

lemma monom-list-1[simp]: \( \text{monom-list} \; 1 = [] \)
by (transfer, simp)

lemma monom-degree: \( \text{eval-monom} \; (\lambda \cdot. \text{x}) \; m = \text{x} ^{\text{monom-degree} \; \text{m}} \)
by (transfer, rule monom-list-degree)

lemma poly-coeff-sum: \( \text{poly-coeff-sum} \; p \geq 0 \)
unfolding \( \text{poly-coeff-sum-def} \)
proof (induct \( p \))
case Nil show \( ?\)case by (simp add: ge-refl)

next
case (Cons \( \text{mc} \; \text{p} \))

have \( \sum \text{mc} = \text{mc} \# \; p \; \text{max} 0 \; (\text{snd} \; \text{mc}) = \text{max} \; 0 \; (\text{snd} \; \text{mc}) + \sum \text{mc} = \text{mc} \; \text{max} \; \ldots \)

558
also have \( \ldots \geq 0 + 0 \)

by \( \text{(rule \( \text{ge-trans}[\text{OF plus-left-mono plus-right-mono}[\text{OF Cons}], \text{auto}] \))} \)

finally show \( \text{?case by \( \text{simp} \)} \)

qed

lemma \text{poly-degree}: assumes \( x: x \geq (1 :: 'a :: \text{poly-carrier}) \)

shows \( \text{poly-coeff-sum \( p \ast (x \ast \text{poly-degree \( p \))} \geq \text{eval-poly} (\lambda \cdot \cdot. x) \cdot \text{p} \)

proof (\text{induct \( \text{p} \)})

\text{case Nil show \( \text{?case by \( \text{simp add: \text{ge-refl poly-degree-def poly-coeff-sum-def})} \}

next

\text{case (Cons \( \text{mc \( p \))}

obtain \( m \cdot c \) where \( \text{mc: mc = \( (m, c) \) by \text{force} \}

from \( \text{ge-trans[OF \( x \text{ one-ge-zero}] have \( x0: x \geq 0 \).} \)

have \( \text{id1: \text{eval-poly} (\lambda \cdot \cdot. x) (\text{mc \# \( p \)) = x \ast \text{monom-degree \( m \ast c \) + \text{eval-poly} (\lambda \cdot \cdot. x) \cdot \text{p} \)

unfolding \text{mc by \( \text{(simp add: \text{monom-degree})} \)

have \( \text{id2: \text{poly-coeff-sum \( (\text{mc \# \( p \)) \ast x \ast \text{poly-degree \( (\text{mc \# \( p \))} = \}

\text{x \ast max (\text{monom-degree \( m \) \) \( \text{(poly-degree \( p \))} \ast \text{(max 0 c) + \text{poly-coeff-sum \( p \ast x \)

\ast max (\text{monom-degree \( m \) \) \( \text{(poly-degree \( p \)) \}}

unfolding \text{poly-coeff-sum-def poly-degree-def by \( \text{(simp add: \text{mc field-simps})} \)

show \( \text{poly-coeff-sum \( (\text{mc \# \( p \)) \ast x \ast \text{poly-degree \( (\text{mc \# \( p \))} \geq \text{eval-poly} (\lambda \cdot \cdot. x) \cdot \text{p} \)

\text{(mc \# \( p \))}

unfolding \text{id1 \text{id2}}

proof (\text{rule \( \text{ge-trans[OF plus-left-mono plus-right-mono])} \)

show \( \text{x \ast max (\text{monom-degree \( m \) \( \text{(poly-degree \( p \))} \ast \text{max 0 c \geq \text{x \ast \text{monom-degree \( m \ast c \) \)}}

\text{by \( \text{(rule \( \text{ge-trans[OF times-left-mono[OF - pow-mono-exp] times-right-mono[OF}

\text{pow-ge-zero]}, \text{insert \( x \cdot x0), \text{auto})} \)

show \( \text{poly-coeff-sum \( p \ast x \ast \text{max (\text{monom-degree \( m \) \( \text{(poly-degree \( p \))} \geq \text{eval-poly (\lambda \cdot \cdot. x) \cdot \text{p} \)

\text{by \( \text{(rule \( \text{ge-trans[OF times-right-mono[OF poly-coeff-sum pow-mono-exp[OF}

\text{x]], Cons), \text{auto})} \)

qed

qed

lemma \text{poly-degree-bound}: assumes \( x: x \geq (1 :: 'a :: \text{poly-carrier}) \)

and \( c: c \geq \text{poly-coeff-sum \( p \)

and \( d: d \geq \text{poly-degree \( p \)

shows \( c \ast (x \ast d) \geq \text{eval-poly (\lambda \cdot \cdot. x) \cdot \text{p} \)

by \( \text{(rule \( \text{ge-trans[OF \( \text{ge-trans[OF times-left-mono[OF pow-ge-zero[OF ge-trans[OF x one-ge-zero]] c]}

\text{times-right-mono[OF poly-coeff-sum pow-mono-exp[OF x d]]}] poly-degree[OF x])} \)

\text{poly-degree} \text{bound} \}

\text{(18.7 Executable and sufficient criteria to compare polynomials and ensure monotonicity)}

\text{poly-split extracts the coefficient for a given monomial and returns additionally the remaining polynomial}

\text{definition poly-split :: \( ('v \text{ monom}) \Rightarrow ('v, 'a :: \text{zero})\text{poly} \Rightarrow 'a \times ('v, 'a)\text{poly} \)

\text{559}
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) * c) + eval-poly α 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 res 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 res 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 nd, 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

561
thus \(\text{case}\) by simp

next

\text{case}\ (\text{Cons}\ m\ c\ p)

obtain\ \ m\ c\ \text{where}\ mc = (m,c)\ \text{by}\ (\text{cases} mc,\ \text{auto})

show \(\text{case}\)

proof\ (\text{cases}\ \text{List.extract}\ (\lambda mc.\ \text{fst} mc = m)\ q)

\text{case}\ None

with\ \text{Cons}(2)\ \text{have}\ \text{rec: check-poly-ge}\ p\ q\ \text{and}\ c : c \geq 0\ \text{using}\ mc\ \text{by}\ \text{auto}

from\ \text{Cons}(1)[\text{OF}\ \text{rec}]\ \text{have}\ \text{rec:}\ p \geq p\ q\ .

show \(\text{?thesis}\)

proof\ (\text{simp\ only: mc,\ unfold\ poly-ge-def,\ intro\ allI\ impI})

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

assume\ \text{pos: pos-assign}\ \alpha

have\ \text{ge: eval-monom}\ \alpha m \ast c \geq 0

using\ \text{times-right-mono}[\text{OF}\ \text{pos-assign-monom}[\text{OF} pos,\ \text{of} m]\ c]\ \text{by}\ \text{simp}

from\ \text{rec}\ \text{have}\ \text{pq: eval-poly}\ \alpha p \geq\ \text{eval-poly}\ \alpha q\ \text{unfolding poly-ge-def}\ \text{using}

\text{pos\ by\ auto}

show\ \text{eval-poly}\ \alpha ((m,c) \# p) \geq\ \text{eval-poly}\ \alpha q\ \text{unfolding poly-ge-def}\ \text{using}

\text{ge-trans}[\text{OF}\ \text{plus-left-mono}[\text{OF}\ \text{ge}]\ \text{plus-right-mono}[\text{OF}\ \text{pq}]]\ \text{by}\ \text{simp}

qed

next

\text{case}\ (\text{Some}\ res)

obtain\ q1\ md\ q2\ \text{where}\ res = (q1,md,q2)\ \text{by}\ (\text{cases}\ res,\ \text{auto})

with\ \text{extract-SomeE}[\text{OF}\ \text{Some}[\text{simplified\ this}]]\ \text{obtain}\ d\ \text{where}\ q : q1 @ (m,d) \# q2\ \text{and}\ \text{res: res} = (q1,(m,d),q2)

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

from\ \text{Cons}(2)\ \text{Some}\ mc\ \text{res}\ \text{have}\ \text{rec: check-poly-ge}\ p\ (q1 \# q2)\ \text{and}\ c : c \geq d\ \text{by}\ \text{auto}

from\ \text{Cons}(1)[\text{OF}\ \text{rec}]\ \text{have}\ p : p \geq p\ q1 @ q2\ .

show \(\text{?thesis}\)

proof\ (\text{simp\ only: mc,\ unfold\ poly-ge-def,\ intro\ allI\ impI})

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

assume\ \text{pos: pos-assign}\ \alpha

have\ \text{ge: eval-monom}\ \alpha m \ast c \geq\ \text{eval-monom}\ \alpha m \ast d

using\ \text{times-right-mono}[\text{OF}\ \text{pos-assign-monom}[\text{OF} pos,\ \text{of} m]\ c]\ \text{by}\ \text{simp}

from\ p\ \text{have}\ \text{ge2: eval-poly}\ \alpha p \geq\ \text{eval-poly}\ \alpha (q1 \# q2)\ \text{unfolding poly-ge-def}\ \text{using}

\text{pos\ by\ auto}

show\ \text{eval-poly}\ \alpha ((m,c) \# p) \geq\ \text{eval-poly}\ \alpha q\ \text{using}\ \text{ge-trans}[\text{OF}\ \text{plus-left-mono}[\text{OF}\ \text{ge}]\ \text{plus-right-mono}[\text{OF}\ \text{ge2}]]\ \text{by}\ \text{(simp\ add: q field-simps)}

qed

qed

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

definition\ \text{check-poly-weak-mono-all} :: (\check{v},\check{a} :: \text{ordered-semiring-0})\text{poly} \Rightarrow\ \text{bool}

where\ \text{check-poly-weak-mono-all}\ p \equiv\ \text{list-all}\ (\lambda (m,c).\ c \geq 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: \forall x. f x \geq g x
  and pos: pos-assign g
  hence fg: \forall x. f x \geq g x by auto
from pos[unfolded pos-assign-def] have g: \forall x. g x \geq 0 ..
from assms have \forall m c. (m,c) \in set p \Longrightarrow c \geq 0 unfolding check-poly-weak-mono-all-def
by (auto simp: list-all-iff)
  thus eval-poly f p \geq eval-poly g p
proof (induct p)
  case Nil thus ?case by (simp add: ge-refl)
next
  case (Cons mc p)
  hence IH: eval-poly f p \geq eval-poly g p by auto
  show ?case
    proof (cases mc)
      case (Pair m c)
      with Cons have c: c \geq 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 \geq eval-monom g m
            by (rule eval-monom-mono[OF fg g])
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 \forall m c. (m,c) \in set p \Longrightarrow c \geq 0 unfolding check-poly-weak-mono-all-def
  by (auto simp: list-all-iff)
thus check-poly-ge p []
by (induct p, simp add: check-poly-ge.simps, clarify, auto simp: check-poly-ge.simps extract-Nil-code)
qed

better check for weak monotonicity for discrete carriers: p is monotone in v if p(\ldots v + 1 \ldots) \geq p(\ldots v \ldots)
definition check-poly-weak-mono-discrete :: ('v :: linorder,'a :: poly-carrier)poly \Rightarrow 'v \Rightarrow bool
  where check-poly-weak-mono-discrete p v \equiv check-poly-ge (poly-subst (\lambda w. poly-of (if w = v then PSum [PNam 1, PVar v] else PVar w)) p) p
definition \textit{check-poly-weak-mono-and-pos} :: bool \Rightarrow (\'v :: \text{linorder}, \'a :: \text{poly-carrier}) \text{poly} \\
where \textit{check-poly-weak-mono-and-pos} \text{ discrete} \ p \equiv \begin{cases} 
\text{list-all} \ (\lambda \ v. \ \text{check-poly-weak-mono-discrete} \ p \ v) \\
\text{eval-poly} \ (\lambda \ w. \ \theta) \ p \geq \theta \\
\text{check-poly-weak-mono-all} \ p
\end{cases}

definition \textit{check-poly-weak-anti-mono-discrete} :: (\'v :: \text{linorder}, \'a :: \text{poly-carrier}) \text{poly} \Rightarrow \'v \Rightarrow \text{bool} \\
where \textit{check-poly-weak-anti-mono-discrete} \ p \ v \equiv \text{check-poly-ge} \ p \ (\text{poly-subst} \ (\lambda \ w. \ \text{poly-of} \ (\text{if} \ w = v \ \text{then} \ \text{PSum} \ [\text{PNum} \ 1, \text{PVar} \ v] \ \text{else} \ \text{PVar} \ w)) \ p)

custom \text{poly-order-carrier} 
begin 
lemma \textit{check-poly-weak-mono-discrete}: \text{fixes} \ v :: \'v :: \text{linorder} \ \text{and} \ p :: (\'v,\'a) \text{poly} \\
\text{assumes} \text{ discrete and check:} \ \textit{check-poly-weak-mono-discrete} \ p \ v \\
\text{shows} \ \text{poly-weak-mono} \ p \ v \\
\text{unfolding} \ \textit{poly-weak-mono-def} \\
\text{proof (intro allI impI)} \\
\text{fix} \ f \ g :: (\'v,\'a) \text{assign} \\
\text{assume} \ fgw: \forall \ w. \ (v \neq w \rightarrow f \ w = g \ w) \\
\text{and gass: pos-assign} \ g \\
\text{and} \ v: f \ v \geq g \ v \\
\text{from} \ fgw \ \text{have} \ w: \bigwedge \ w. \ v \neq w \rightarrow f \ w = g \ w \ \text{by auto} \\
\text{from} \ \text{assms} \ \textit{check-poly-ge} \ \text{have} \ ge: \ \text{poly-ge} \ (\text{poly-subst} \ (\lambda \ w. \ \text{poly-of} \ (\text{if} \ w = v \ \text{then} \ \text{PSum} \ [\text{PNum} \ 1, \text{PVar} \ v] \ \text{else} \ \text{PVar} \ w)) \ p) \ p \ (\text{is} \ \text{poly-ge} \ ?p1 \ p) \\
\text{unfolding} \ \textit{check-poly-weak-mono-discrete-def} \ \text{by} \ \text{blast} \\
\text{from} \ \text{discrete} (\text{OF} (\text{discrete} \ v)) \ \text{obtain} \ k' \ \text{where} \ id: \ f \ v = (((+ \ 1)^{\ k'}) \ (g \ v)) \\text{by auto} \\
\text{show} \ \text{eval-poly} \ f \ p \geq \ \text{eval-poly} \ g \ p \\
\text{proof (cases k')} \\
\text{case} \ 0 \\
\ \{ \\
\text{fix} \ x \\
\ \text{have} \ f \ x = g \ x \ \text{using} \ id \ 0 \ w \ \text{by (cases} \ x = v, \ \text{auto}) \\
\} \\
\text{hence} \ f = g \ . . \\
\text{thus} ?\text{thesis} \ \text{using} \ \text{ge-refl} \ \text{by simp} \\
\text{next} \\
\text{case} (\text{Succ} \ k) \\
\text{with id} \ \text{have} \ f \ v = (((+ \ 1)^{\ k}) \ \text{Succ} \ k)) \ (g \ v) \ \text{by} \ \text{simp} \\
\text{with} \ w \ \text{gass} \ \text{show} \ \text{eval-poly} \ f \ p \geq \ \text{eval-poly} \ g \ p \\
\text{proof (induct} \ k \ \text{arbitrary:} \ f \ g \ \text{rule: less-induct}) \\
\text{case} (\text{less} \ k) \\
\text{show} \ ?\text{case} \\
\text{proof (cases} \ k) \\
\end{lemma}
case 0
with less have id0: \( f v = 1 + g v \) by simp
have id1: eval-poly \( f p = \) eval-poly \( g \) \(?p\)
proof (rule eval-poly-subst)
  fix \( w \)
  show \( f w = eval-poly g \) (poly-of (if \( w = v \) then \( P\text{Sum} \ [P\text{Num} \ 1, \ P\text{Var} \ v] \)
else \( P\text{Var} \ w \))
proof (cases \( w = v \))
  case True
  show \(?thesis\) by (simp add: True id0 zero-poly-def)
next
  case False
  with less have \( f w = g w \) by simp
  thus \(?thesis\) by (simp add: False)
qed
qed
have eval-poly \( g \) \(?p\) \( \geq \) eval-poly \( g \) \( p \) using \( \text{ge} \) less unfolding poly-ge-def
by simp
with id1 show \(?thesis\) by simp
next
  case (Suc \( k k \))
  obtain \( g' \) where \( g' = (\lambda w. \text{if } (w = v) \text{ then } 1 + g w \text{ else } g w) \) 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 \( 1 + (0 :: 'a) = 1 \) by simp
  also have \( \ldots \geq 0 \) by (rule one-ge-zero)
  finally have \( g'\text{pos} : \text{pos-assign} \ g' \) using less(3) unfolding pos-assign-def
    by (simp add: \( g' \))
  \{ fix \( w \)
  assume \( v \neq w \)
  hence \( f w = g' w \)
    unfolding \( g' \) by (simp add: less)
  \}
  note \( w = \text{this} \)
  have eq: \( f v = (+) (1 :: 'a) ^^ \text{Suc} \ k k \ ((g' \ v)) \)
    by (simp add: less(4) g' Suc, rule arg-cong[where \( f = (+) \ 1 \], induct \( k k \),
auto)
  from Suc have \( k k < k \) 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 \( w = \text{this} \)
  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

565
lemma check-poly-weak-anti-mono-discrete:
fixes v :: 'v :: linorder and p :: ('v,'a)poly
assumes discrete and check: check-poly-weak-anti-mono-discrete p v
shows poly-weak-anti-mono p v

proof (intro allI impI)
fix f g :: ('v,'a)assign
assume fgw: \forall w. (\mathrm{v} \neq w \rightarrow f w = g w)
and gass: pos-assign g
and v: f v \geq g v
from fgw have w: \forall w. v \neq w \rightarrow f w = g w by auto
from assms check-poly-ge have ge: poly-ge p (poly-subst (\lambda 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) ^ 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 (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 \geq 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 \( g \ p \geq \) 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 \( g' = (\lambda w. \text{if } (w = v) \text{ then } 1 + g w \text{ else } g w) \)
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 \(. . . \geq 0\) by (rule one-ge-zero)
finally have \( g' \text{pos}; \text{pos-assign } g' \text{ using less(3) unfolding pos-assign-def}\)
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 \( g' \ p \geq \) eval-poly \( f \ p\) 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 rec2 rec1])
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 \( p \) \( \land \ (p \geq \) 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
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

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 fy: ∀ 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 \( \text{eval-poly } f \ p \geq \text{eval-poly } g \ p \)

proof (induct \( p \))
  case (Cons \( m \ c \) \( p \))
  hence \( \text{IH} \): \( \text{eval-poly } f \ p \geq \text{eval-poly } g \ p \) by auto

obtain \( m \ c \) where \( m \ c = (m,c) \) by force
  proof
  assume \( c: c \geq 0 \) \( \vee \) \( v \notin \text{monom-vars } m \) by auto
  have \( \text{eval-monom } f \ m * c \geq \text{eval-monom } g \ m * c \)
  using \( \text{ge-refl} \)
  qed
  qed

definition check-poly-weak-mono-smart :: \( \Rightarrow (v :: \text{linorder}, a :: \text{poly-carrier}) \text{poly} \Rightarrow \text{bool} \)
where check-poly-weak-mono-smart discrete \( \equiv \) if discrete then check-poly-weak-mono-discrete else check-poly-weak-mono

lemma (in \( \text{poly-order-carrier} \)) check-poly-weak-mono-smart: fixes \( p :: (v :: \text{linorder}, a :: \text{poly-carrier}) \text{poly} \)
shows check-poly-weak-mono-smart discrete \( v \Rightarrow \) poly-weak-mono \( p \ v \)
unfolding check-poly-weak-mono-smart-def
using check-poly-weak-mono check-poly-weak-mono-smart-discrete by (cases discrete, auto)

definition check-poly-weak-anti-mono :: \( (v :: \text{linorder}, a :: \text{ordered-semiring-0}) \text{poly} \Rightarrow \text{bool} \)
where check-poly-weak-anti-mono poly \( v \equiv \) list-all \( (\lambda (m, c). 0 \geq c \vee v \notin \text{monom-vars } m) \) \( p \)

lemma check-poly-weak-anti-mono: fixes \( p :: (v :: \text{linorder}, a :: \text{poly-carrier}) \text{poly} \)
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 \( f \ g :: (v,a)\text{assign} \)
assume \( \forall \ x. v \neq x \longrightarrow f \ x = g \ x \)
and pos: pos-assign g
and ge: f v ≥ g v
hence fg: \(\wedge x. v \neq x \implies f x = g x\) by auto
from pos[unfolded pos-assign-def] have g: \(\wedge x. g x \geq 0\) ..
from assms have \(\wedge m. c. (m,c) \in \text{set } p \implies 0 \geq c \lor v /\in \text{monom-vars } m\)
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 \lor v /\in \text{monom-vars } m by auto
show ?case unfolding mc eval-poly.simps fst-conv snd-conv
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 /\in \text{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

definition check-poly-weak-anti-mono-smart :: bool ⇒ ('v :: linorder,'a :: poly-carrier)poly ⇒ 'a ⇒ 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
by (cases discrete, auto)

definition check-poly-gt :: ('a ⇒ 'a ⇒ bool) ⇒ ('v :: linorder,'a :: ordered-semiring-0)poly
\[
\Rightarrow (\texttt{v}, \texttt{a}) \texttt{poly} \Rightarrow \texttt{bool}
\]

where \( \text{check-poly-gt} \) \( \texttt{gt} \) \( \texttt{p} \) \( \texttt{q} \) \( \equiv \) let \( (\texttt{a1}, \texttt{p1}) = \text{poly-split} \texttt{1} \texttt{p} \); \( (\texttt{b1}, \texttt{q1}) = \text{poly-split} \texttt{1} \texttt{q} \)
in \( \texttt{gt} \texttt{a1} \texttt{b1} \land \text{check-poly-ge} \texttt{p1} \texttt{q1} \)

fun \( \text{univariate-power-list} :: \texttt{'}v \Rightarrow \texttt{'}v \texttt{monom-list} \Rightarrow \texttt{nat option} \)

where \( \text{univariate-power-list} \texttt{x} \ [(\texttt{y}, \texttt{n})] = \) (if \( \texttt{x} = \texttt{y} \) then \( \texttt{Some} \texttt{n} \) else \( \texttt{None} \))

| \( \text{univariate-power-list} \texttt{- -} = \texttt{None} \)

lemma \( \text{univariate-power-list}: \text{assumes} \texttt{monom-inv} \texttt{m} \texttt{univariate-power-list} \texttt{x} \texttt{m} = \) \( \texttt{Some} \texttt{n} \)

shows \( \texttt{sum-var-list} \texttt{m} = (\lambda \texttt{y}. \texttt{if} \texttt{x} = \texttt{y} \texttt{then} \texttt{n} \texttt{else} 0) \)

\( \texttt{n} \geq 1 \)

proof –

have \( \texttt{m}: \texttt{m} = [(\texttt{x}, \texttt{n})] \) using \( \texttt{assms} \)

by (induct \texttt{x} \texttt{m} rule: \texttt{univariate-power-list.induct}, \texttt{auto split: if-splits})

show \( \texttt{eval-monom-list} \texttt{α} \texttt{m} = ((\texttt{α} \texttt{x}) \texttt{ˆn}) \texttt{sum-var-list} \texttt{m} = (\lambda \texttt{y}. \texttt{if} \texttt{x} = \texttt{y} \texttt{then} \texttt{n} \texttt{else} 0) \)

\( \texttt{n} \geq 1 \) using \( \texttt{assms(1)} \)

unfolding \( \texttt{m} \texttt{monom-inv-def} \) by (auto simp: \texttt{sum-var-list-def})

qed

lift-definition \( \text{univariate-power} :: \texttt{'}v :: \texttt{linorder} \Rightarrow \texttt{'}v \texttt{monom} \Rightarrow \texttt{nat option} \)

is \( \text{univariate-power-list} \).

lemma \( \text{univariate-power}: \text{assumes} \texttt{univariate-power} \texttt{x} \texttt{m} = \) \( \texttt{Some} \texttt{n} \)

shows \( \texttt{sum-var} \texttt{m} = (\lambda \texttt{y}. \texttt{if} \texttt{x} = \texttt{y} \texttt{then} \texttt{n} \texttt{else} 0) \)

\( \texttt{eval-monom} \texttt{α} \texttt{m} = ((\texttt{α} \texttt{x}) \texttt{ˆn}) \)

\( \texttt{n} \geq 1 \)

by (atomize(full), insert \texttt{assms}, transfer, \texttt{auto dest: univariate-power-list})

lemma \( \text{univariate-power-var-monom}: \text{univariate-power} \texttt{y} \texttt{(var-monom} \texttt{x}) = (\texttt{if} \texttt{x} = \texttt{y} \texttt{then} \texttt{Some} 1 \texttt{else} \texttt{None}) \)

by (\texttt{transfer, auto})

definition \( \text{check-monom-strict-mono} :: \texttt{bool} \Rightarrow \texttt{'}v :: \texttt{linorder} \texttt{monom} \Rightarrow \texttt{'}v \Rightarrow \texttt{bool} \)

where \( \text{check-monom-strict-mono} \texttt{pm} \texttt{m} \texttt{v} \equiv \text{case} \texttt{univariate-power} \texttt{v} \texttt{m} \texttt{of} \)

\( \texttt{Some} \texttt{p} \Rightarrow \texttt{pm} \lor \texttt{p} = 1 \)

| \( \texttt{None} \Rightarrow \texttt{False} \)

definition \( \text{check-poly-strict-mono} :: \texttt{bool} \Rightarrow (\texttt{'}v :: \texttt{linorder}, \texttt{'}a :: \texttt{poly-carrier})\texttt{poly} \Rightarrow \texttt{'}v \Rightarrow \texttt{bool} \)

where \( \text{check-poly-strict-mono} \texttt{pm} \texttt{p} \texttt{v} \equiv \text{list-ex} (\lambda \texttt{m}, \texttt{c}. \texttt{(c} \geq 1) \land \text{check-monom-strict-mono} \texttt{pm} \texttt{m} \texttt{v}) \texttt{p} \)

definition \( \text{check-poly-strict-mono-discrete} :: (\texttt{'}a :: \texttt{poly-carrier} \Rightarrow \texttt{'}a \Rightarrow \texttt{bool}) \Rightarrow (\texttt{'}v :: \texttt{linorder}, \texttt{'}a)\texttt{poly} \Rightarrow \texttt{'}v \Rightarrow \texttt{bool} \)

where \( \text{check-poly-strict-mono-discrete} \texttt{gt} \texttt{p} \texttt{v} \equiv \texttt{check-poly-gt} \texttt{gt} \texttt{(poly-subst (λ \texttt{w}.} \texttt{w).} \texttt{λ \texttt{a}} \texttt{\texttt{poly-carrier} (\texttt{λ \texttt{w}.} \texttt{\texttt{poly}} \texttt{\texttt{w)} \texttt{\texttt{poly-carrier}})\texttt{poly} \Rightarrow \texttt{'}v \Rightarrow \texttt{bool} \)

where \( \text{check-poly-strict-mono-discrete} \texttt{gt} \texttt{p} \texttt{v} \equiv \texttt{check-poly-gt} \texttt{gt} \texttt{(poly-subst (λ \texttt{w}.} \texttt{w).} \texttt{λ \texttt{a}} \texttt{\texttt{poly-carrier} (\texttt{λ \texttt{w}.} \texttt{\texttt{poly}} \texttt{\texttt{w)} \texttt{\texttt{poly-carrier}})\texttt{poly} \Rightarrow \texttt{'}v \Rightarrow \texttt{bool} \)
poly-of (if \( w = v \) then \( \text{PSum} \ [\text{PNam} \ 1, \ PVar \ v] \) else \( PVar \ w \)) \( p \) \( p \)

**definition** check-poly-strict-mono-smart :: \( \text{bool} \Rightarrow \text{bool} \Rightarrow (\text{'a} :: \text{poly-carrier} \Rightarrow \text{'a} \Rightarrow \text{bool}) \Rightarrow (\text{'v} :: \text{linorder}, \text{'a})\text{poly} \Rightarrow \text{'v} \Rightarrow \text{bool} \)

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 \( \alpha \beta :: (\text{'v} :: \text{linorder}, \text{'a})\text{assign} \) and \( v :: \text{'v} \) and \( m :: \text{'v} \text{monom} \)

assumes check: check-monom-strict-mono power-mono m v

and \( \text{gt} : \alpha \ v \succ \beta \ v \)

and \( \text{ge} : \beta \ v \geq 0 \)

shows eval-monom \( \alpha \ m \succ \) eval-monom \( \beta \ m \)

proof −

from check[unfolded check-monom-strict-mono-def] obtain \( n \) where

\( \text{uni} : \text{univariate-power} \ v \ m = \text{Some} \ n \) and \( 1 : \neg \text{power-mono} \Rightarrow n = 1 \)

by (auto split: \text{option.splits})

from univariate-power[OF \( \text{uni} \)]

have \( n1 : \ n \geq 1 \) and \( \text{eval} : \) eval-monom \( a \ m = a \ v ^{n} \) for \( a :: (\text{'v}, \text{'a})\text{assign} \)

by auto

show \( \)?thesis

proof (cases power-mono)

 case False

 with \( \text{gt} 1 \) [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 \( f \ g :: (\text{'b}, \text{'a})\text{assign} \)

assume \( \text{fgw} : \forall \ w. (v \neq w \Rightarrow f \ w = g \ w) \)

and \( \text{pos} : \text{pos-assign} \ g \)

and \( \text{fgv} : f \ v \succ g \ v \)

from \( \text{pos}[\text{unfolded pos-assign-def}] \) have \( \forall x. g \ x \geq 0 \) ..

{ \( \)

fix \( w \)

have \( f \ w \geq g \ w \)

proof (cases \( v = w \))

 case False

572
with \texttt{fgw ge-refl} show \(?\text{thesis}\) by auto

next

\textbf{case} True

from \texttt{fgv[unfolded True]} show \(?\text{thesis}\) by (rule \texttt{gt-imp-ge})

\textbf{qed}

\}

\textbf{note} \texttt{fgw2 = this}

\textbf{let} \(?e = \text{eval-poly}\)

\textbf{show} \(?e f p \succ \?e g p\)

\textbf{using} \texttt{check1[unfolded check-poly-strict-mono-def, simplified list-ex-iff]}

\texttt{check2[unfolded check-poly-weak-mono-all-def, simplified list-all-iff, THEN bspec]}

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

\textbf{case} \texttt{Nil} \textbf{thus} \(?\text{case}\) by simp

next

\textbf{case} \texttt{Cons mc p} \textbf{obtain} \(m c\) where \(mc\): \(mc = (m,c)\) by (cases \(mc\), auto)

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

\textbf{proof (cases }c = 1 \land \text{check-monom-strict-mono power-mono }m v\textbf{)}

\textbf{case} True

\textbf{hence} \(c : c \geq 1\) and \(m : \text{check-monom-strict-mono power-mono }m v\) by blast+

from \texttt{times-gt-mono[OF check-monom-strict-mono[OF of f g, OF fgv g] c]}

have \(gt: \text{eval-monom }f m \ast c \succ \text{eval-monom }g m \ast c\).

from \texttt{Cons(3)} have \(\text{check-poly-weak-mono-all }p\) unfolding \texttt{check-poly-weak-mono-all-def list-all-iff by auto}

\textbf{from} \texttt{check-poly-weak-mono-all[OF this, unfolded poly-weak-mono-all-def, rule-format, OF fgv2 pos]} have \(\?e f p \succ \?e g p\).

\textbf{from} \texttt{compat2[OF plus-gt-left-mono[OF gt] plus-right-mono[OF ge]]}

\textbf{show} \(?\text{thesis}\) unfolding \(mc\) by simp

next

\textbf{case} False

with \texttt{Cons(2) mc have} \(\exists mc \in \text{set }p. (\lambda (m,c). c \geq 1 \land \text{check-monom-strict-mono power-mono }m v)\) by auto

from \texttt{Cons(1)[OF this]} \texttt{Cons(3) have rec: \?e f p \succ \?e g p by simp}

from \texttt{Cons(3) mc have} \(c : c \geq 0\) by auto

from \texttt{times-left-mono[OF c eval-monom-mono(1)[OF fgv2 g]]}

have \(ge: \text{eval-monom }f m \ast c \geq \text{eval-monom }g m \ast c\).

from \texttt{compat2[OF plus-gt-left-mono[OF rec] plus-right-mono[OF ge]]}

\textbf{show} \(?\text{thesis}\) by (simp add: \(mc\) field-simps)

\textbf{qed}

\textbf{qed}

\textbf{lemma} \texttt{check-poly-gt:}

\textbf{fixes} \(p :: (\'v :: \text{linorder},\'a)\text{poly}\)

\textbf{assumes} \texttt{check-poly-gt gt p q shows }p > p q

\textbf{proof --}

\textbf{obtain} \(a1 p1\) where \(p: \text{poly-split }1 p = (a1,p1)\) by force
obtain \( b_1 \ q_1 \) where \( q \): poly-split \( 1 \ q = (b_1,q_1) \) by force
from \( p \ q \) assms have \( gt: a1 \succ b1 \) and \( ge: p1 \geq p1 \ q1 \) unfolding check-poly-gt-def using check-poly-ge[of \( p1 \ q1 \)] by auto
show \( \neg \thesis \)
proof (unfold poly-gt-def, intro impI allI)
  fix \( \alpha :: (\'v,\'a)assign \)
  assume pos-assign \( \alpha \)
  with \( ge \) have \( ge: \text{eval-poly} \ \alpha \ p1 \geq \text{eval-poly} \ \alpha \ q1 \) unfolding poly-ge-def by simp
from plus-gt-left-mono[OF \( gt \)] compat[OF plus-gt-gt[OF \( ge \)]]
  have \( gt: a1 \succ b1 \) and \( ge: p1 \geq p1 \ q1 \) by (force simp: field-simps)
  show eval-poly \( \alpha \ p \succ eval-poly \ \alpha \ q \) by (simp add: poly-split[OF \( p \), unfolded eq-poly-def] poly-split[OF \( q \), unfolded eq-poly-def] \( gt \))
qed

lemma check-poly-strict-mono-discrete:
fixes \( v :: (\'v,\'a)linorder \) 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: \( \forall \ w. (v \neq w \rightarrow f \ w = g \ w) \)
  and gass: pos-assign \( g \)
  and \( v: f \ v \succ g \ v \)
from gass have \( g: \forall x. x \geq 0 \) unfolding pos-assign-def ..
from fgw have \( w: \forall w. v \neq w \rightarrow f \ w = g \ w \) by auto
from assms check-poly-gt have \( gt: poly-gt \ (\)poly-subst (\lambda 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-gt-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 \( \bot \)
show \( f \case \)
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 \( f \, w = \text{eval-poly} \, g \) (poly-of (if \( w = v \) then \( \text{PSum} \, [\text{PNum} \, 1, \text{PVar} \, v] \) else \( \text{PVar} \, w \)))
  proof (cases \( w = v \))
    case True
    show \( \text{thesis} \) by (simp add: True id0 zero-poly-def)
  next
    case False
    with less have \( f \, w = g \, w \) by simp
    thus \( \text{thesis} \) by (simp add: False)
  qed
  qed
  have \( \text{eval-poly} \, g \, ?p1 \triangleright \text{eval-poly} \, g \, p \) using gt less unfolding poly-gt-def by simp
  with id1 show \( \text{thesis} \) by simp
next
  case (Suc \( \, kk \))
  obtain \( g' \) where \( g' = (\lambda \, w \, . \, \text{if} \, (w = v) \, \text{then} \, 1 + g \, w \, \text{else} \, g \, w) \) by auto
  have \( (1 :: 'a) + \, g \, v \geq 1 + 0 \)
    by (rule plus-right-mono, simp add: less(3)[unfolded pos-assign-def])
  also have \( (1 :: 'a) + \, 0 = 1 \) by simp
  also have \( \ldots \geq 0 \) by (rule one-ge-zero)
  finally have \( g' \, \text{pos}: \, \text{pos-assign} \, g' \) 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 \( \, w = \text{this} \)
  have eq: \( f \, v = ((+) \, (1 :: 'a) \, \text{``Suc} \, kk \, \text{``} \, (g' \, v)) \)
    by (simp add: less(4) \( g' \, \text{Suc}, \, \text{rule arg-cong[where} \, f = (+) \, 1], \, \text{induct} \, \text{Suc, auto} \))
  from Suc have \( \, kk : \, kk < k \) by simp
  from less(1)[OF \( \, kk \, w \, g' \, \text{pos} \)] eq
  have rec1: \( \text{eval-poly} \, f \, p \, \triangleright \, \text{eval-poly} \, g' \, p \) by simp
  \{
    fix \( \, w \)
    assume \( v \neq w \)
    hence \( g' \, w = g \, w \)
      unfolding \( g' \) by simp
  \}
  note \( \, w = \text{this} \)
  from Suc have \( \, z : \, 0 < k \) by simp
  from less(1)[OF \( \, z \, w \, \text{less(3)} \)] \( g' \)
  have rec2: \( \text{eval-poly} \, g' \, p \, \triangleright \, \text{eval-poly} \, g \, p \) by simp
  show \( \text{thesis} \) by (rule gt-trans[OF rec1 rec2])
  qed
  qed
lemma check-poly-strict-mono-smart:
\[\begin{align*}
\text{assumes} & \quad \text{check1: check-poly-strict-mono-smart discrete power-mono gt p v} \\
\text{and} & \quad \text{check2: check-poly-weak-mono-and-pos discrete p} \\
\text{shows} & \quad \text{poly-strict-mono p v}
\end{align*}\]
proof (cases discrete)
\[\begin{align*}
\text{case True} \\
\text{with check1[unfolded check-poly-strict-mono-smart-def] check-poly-strict-mono-discrete[OF True]}
\end{align*}\]
show \(?thesis by auto
next
\[\begin{align*}
\text{case False} \\
\text{from check-poly-strict-mono[OF check1[unfolded check-poly-strict-mono-smart-def, simplified False, simplified]] check2[unfolded check-poly-weak-mono-and-pos-def, simplified False, simplified]}
\end{align*}\]
show \(?thesis by auto
qed
end
end

19 Displaying Polynomials

theory Show-Polynomials

imports
\quad Polynomials
\quad Show.Show-Instances

begin

fun shows-monom-list :: ('v :: {linorder, show})monom-list \Rightarrow string \Rightarrow string
where
\[\begin{align*}
\text{shows-monom-list } [(x,p)] & = (\text{if } p = 1 \text{ then shows } x \text{ else shows } x \oplus+ \text{ shows-string }\"nings\" + \oplus+ \text{ shows } p) \\
\text{shows-monom-list } ((x,p) \# m) & = ((\text{if } p = 1 \text{ then shows } x \text{ else shows } x \oplus+ \text{ shows-string }\"nings\" + \oplus+ \text{ shows-string }\"*\" + \oplus+ \text{ shows-monom-list } m) \\
\text{shows-monom-list } [] & = \text{shows-string }\"1\"
\end{align*}\]

instantiation monom :: ({linorder, show}) show

begin

lift-definition shows-prec-monom :: nat \Rightarrow 'a monom \Rightarrow shows is \lambda n. shows-monom-list

lemma shows-prec-monom-append [show-law-simps]:
\[\begin{align*}
\text{shows-prec } d \ (m :: 'a monom) \ (r \oplus s) & = \text{shows-prec } d \ m \ r \oplus s
\end{align*}\]
proof (transfer fixing: d r s)
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 "x") + (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-mul.simps monom-mul-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-mul-polys.simps zero-polynomials_def
  one-polynomials_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


