Signature-Based Gröbner Basis Algorithms

Alexander Maletzky*

March 19, 2025

Abstract

This article formalizes signature-based algorithms for computing Gröbner bases. Such algorithms are, in general, superior to other algorithms in terms of efficiency, and have not been formalized in any proof assistant so far. The present development is both generic, in the sense that most known variants of signature-based algorithms are covered by it, and effectively executable on concrete input thanks to Isabelle's code generator. Sample computations of benchmark problems show that the verified implementation of signature-based algorithms indeed outperforms the existing implementation of Buchberger's algorithm in Isabelle/HOL.

Besides total correctness of the algorithms, the article also proves that under certain conditions they a-priori detect and avoid all useless zero-reductions, and always return 'minimal' (in some sense) Gröbner bases if an input parameter is chosen in the right way.

The formalization follows the recent survey article by Eder and Faugère.

Contents

1 Introduction **3**

2 Preliminaries **3**

2.1 Lists 3

2.1.1 Sequences of Lists 3

2.1.2 *filter* 5

2.1.3 *drop* 7

2.1.4 *count-list* 8

2.1.5 *sorted-wrt* 8

2.1.6 *insort-wrt* and *merge-wrt* 9

2.2 Recursive Functions 10

2.3 Binary Relations 13

*Supported by the Austrian Science Fund (FWF): P 29498-N31

3 More Properties of Power-Products and Multivariate Polynomials 15
3.1 Power-Products . . . 15
3.2 Miscellaneous . . . 16
3.3 *ordered-term.lt* and *ordered-term.higher* . . . 18
3.4 *gd-term.dgrad-p-set* . . . 19
3.5 Regular Sequences . . . 19

4 Signature-Based Algorithms for Computing Gröbner Bases 22
4.1 More Preliminaries . . . 22
4.2 Module Polynomials . . . 23
4.2.1 Signature Reduction . . . 35
4.2.2 Signature Gröbner Bases . . . 57
4.2.3 Rewrite Bases . . . 77
4.2.4 S-Pairs . . . 88
4.2.5 Termination . . . 105
4.2.6 Concrete Rewrite Orders . . . 117
4.2.7 Preparations for Sig-Poly-Pairs . . . 120
4.2.8 Total Reduction . . . 123
4.2.9 Koszul Syzygies . . . 137
4.2.10 Algorithms . . . 144
4.2.11 Minimality of the Computed Basis . . . 199
4.2.12 No Zero-Reductions . . . 206
4.3 Sig-Poly-Pairs . . . 218

5 Sample Computations with Signature-Based Algorithms 235
5.1 Setup . . . 235
5.1.1 Projections of Term Orders to Orders on Power-Products 236
5.1.2 Locale Interpretation . . . 238
5.1.3 More Lemmas and Definitions . . . 240
5.2 Computations . . . 243

1 Introduction

Signature-based algorithms [3, 1] play are central role in modern computer algebra systems, as they allow to compute Gröbner bases of ideals of multivariate polynomials much more efficiently than other algorithms. Although they also belong to the class of critical-pair/completion algorithms, as almost all algorithms for computing Gröbner bases, they nevertheless possess some quite unique features that render a formal development in proof assistants challenging. In fact, this is the first formalization of signature-based algorithms in any proof assistant.

The formalization builds upon the existing formalization of Gröbner bases theory [4] and closely follows Sections 4–7 of the excellent survey article [1]. Some proofs were taken from [5, 2].

Summarizing, the main features of the formalization are as follows:

- It is *generic*, in the sense that it considers the computation of so-called *rewrite bases* and neither fixes the term order nor the rewrite-order.
- It is *efficient*, in the sense that all executable algorithms (e. g. *gb-sig*) operate on sig-poly-pairs rather than module elements, and that polynomials are represented efficiently using ordered associative lists.
- It proves that if the input is a regular sequence and the term order is a POT order, there are no useless zero-reductions (Theorem *gb-sig-no-zero-red*).
- It proves that the signature Gröbner bases computed w. r. t. the 'ratio' rewrite order are minimal (Theorem *gb-sig-z-is-min-sig-GB*).
- It features sample computations of benchmark problems to illustrate the practical usability of the verified algorithms.

2 Preliminaries

theory *Prelims*
  imports *Polynomials.Utils Groebner-Bases.General*
begin

2.1 Lists

2.1.1 Sequences of Lists

lemma *list-seq-length-mono*:
  fixes $seq :: nat \Rightarrow 'a\ list$
  assumes $\bigwedge i.\ (\exists x.\ seq\ (Suc\ i) = x\ \#\ seq\ i)$ **and** $i < j$
  shows $length\ (seq\ i) < length\ (seq\ j)$
proof –

from *assms*(*2*) **obtain** *k* **where** $j = Suc\ (i + k)$ **using** *less-iff-Suc-add* **by** *auto*
show *?thesis* **unfolding** ‹$j = Suc\ (i + k)$›
proof (*induct k*)
case *0*
from *assms*(*1*) **obtain** *x* **where** *eq*: $seq\ (Suc\ i) = x \# seq\ i$ **..**
show *?case* **by** (*simp add*: *eq*)
next
case (*Suc k*)
from *assms*(*1*) **obtain** *x* **where** $seq\ (Suc\ (i + Suc\ k)) = x \# seq\ (i + Suc\ k)$ **..**
hence *eq*: $seq\ (Suc\ (Suc\ (i + k))) = x \# seq\ (Suc\ (i + k))$ **by** *simp*
note *Suc*
also have $length\ (seq\ (Suc\ (i + k))) < length\ (seq\ (Suc\ (i + Suc\ k)))$ **by** (*simp add*: *eq*)
finally show *?case* **.**
qed
qed

corollary *list-seq-length-mono-weak*:
fixes $seq :: nat \Rightarrow 'a\ list$
assumes $\bigwedge i.\ (\exists x.\ seq\ (Suc\ i) = x \# seq\ i)$ **and** $i \leq j$
shows $length\ (seq\ i) \leq length\ (seq\ j)$
proof (*cases* $i = j$)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
with *assms*(*2*) **have** $i < j$ **by** *simp*
with *assms*(*1*) **have** $length\ (seq\ i) < length\ (seq\ j)$ **by** (*rule list-seq-length-mono*)
thus *?thesis* **by** *simp*
qed

lemma *list-seq-indexE-length*:
fixes $seq :: nat \Rightarrow 'a\ list$
assumes $\bigwedge i.\ (\exists x.\ seq\ (Suc\ i) = x \# seq\ i)$
obtains *j* **where** $i < length\ (seq\ j)$
proof (*induct i arbitrary*: *thesis*)
case *0*
have $0 \leq length\ (seq\ 0)$ **by** *simp*
also from *assms lessI* **have** ... $< length\ (seq\ (Suc\ 0))$ **by** (*rule list-seq-length-mono*)
finally show *?case* **by** (*rule 0*)
next
case (*Suc i*)
obtain *j* **where** $i < length\ (seq\ j)$ **by** (*rule Suc*(*1*))
hence $Suc\ i \leq length\ (seq\ j)$ **by** *simp*
also from *assms lessI* **have** ... $< length\ (seq\ (Suc\ j))$ **by** (*rule list-seq-length-mono*)
finally show *?case* **by** (*rule Suc*(*2*))
qed

```
lemma list-seq-nth:
  fixes seq :: nat ⇒ 'a list
  assumes ⋀i. (∃x. seq (Suc i) = x # seq i) and i < length (seq j) and j ≤ k
  shows rev (seq k) ! i = rev (seq j) ! i
proof −
  from assms(3) obtain l where k = j + l using nat-le-iff-add by blast
  show ?thesis unfolding ‹k = j + l›
  proof (induct l)
    case 0
    show ?case by simp
  next
    case (Suc l)
    note assms(2)
    also from assms(1) le-add1 have length (seq j) ≤ length (seq (j + l))
      by (rule list-seq-length-mono-weak)
    finally have i: i < length (seq (j + l)) .
    from assms(1) obtain x where seq (Suc (j + l)) = x # seq (j + l) ..
    thus ?case by (simp add: nth-append i Suc)
  qed
qed

corollary list-seq-nth':
  fixes seq :: nat ⇒ 'a list
  assumes ⋀i. (∃x. seq (Suc i) = x # seq i) and i < length (seq j) and i <
length (seq k)
  shows rev (seq k) ! i = rev (seq j) ! i
proof (rule linorder-cases)
  assume j < k
  hence j ≤ k by simp
  with assms(1, 2) show ?thesis by (rule list-seq-nth)
next
  assume k < j
  hence k ≤ j by simp
  with assms(1, 3) have rev (seq j) ! i = rev (seq k) ! i by (rule list-seq-nth)
  thus ?thesis by (rule HOL.sym)
next
  assume j = k
  thus ?thesis by simp
qed
```

2.1.2 *filter*

```
lemma filter-merge-wrt-1:
  assumes ⋀y. y ∈ set ys ⟹ P y ⟹ False
  shows filter P (merge-wrt rel xs ys) = filter P xs
  using assms
proof (induct rel xs ys rule: merge-wrt.induct)
  case (1 rel xs)
  show ?case by simp
```

next
  case (*2 rel y ys*)
  hence $P\ y \Longrightarrow False$ **and** $\bigwedge z.\ z \in set\ ys \Longrightarrow P\ z \Longrightarrow False$ **by** *auto*
  thus *?case* **by** (*auto simp*: *filter-empty-conv*)
next
  case (*3 rel x xs y ys*)
  hence $\neg\ P\ y$ **and** *x*: $\bigwedge z.\ z \in set\ ys \Longrightarrow P\ z \Longrightarrow False$ **by** *auto*
  have *a*: *filter P* (*merge-wrt rel xs ys*) = *filter P xs* **if** $x = y$ **using** *that x* **by** (*rule 3*(*1*))
  have *b*: *filter P* (*merge-wrt rel xs* (*y # ys*)) = *filter P xs* **if** $x \neq y$ **and** *rel x y*
    using *that 3*(*4*) **by** (*rule 3*(*2*))
  have *c*: *filter P* (*merge-wrt rel* (*x # xs*) *ys*) = *filter P* (*x # xs*) **if** $x \neq y$ **and** $\neg$ *rel x y*
    using *that x* **by** (*rule 3*(*3*))
  show *?case* **by** (*simp add*: *a b c* ‹¬ *P y*›)
qed

lemma *filter-merge-wrt-2*:
  assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow P\ x \Longrightarrow False$
  shows *filter P* (*merge-wrt rel xs ys*) = *filter P ys*
  using *assms*
proof (*induct rel xs ys rule*: *merge-wrt.induct*)
  case (*1 rel xs*)
  thus *?case* **by** (*auto simp*: *filter-empty-conv*)
next
  case (*2 rel y ys*)
  show *?case* **by** *simp*
next
  case (*3 rel x xs y ys*)
  hence $\neg\ P\ x$ **and** *x*: $\bigwedge z.\ z \in set\ xs \Longrightarrow P\ z \Longrightarrow False$ **by** *auto*
  have *a*: *filter P* (*merge-wrt rel xs ys*) = *filter P ys* **if** $x = y$ **using** *that x* **by** (*rule 3*(*1*))
  have *b*: *filter P* (*merge-wrt rel xs* (*y # ys*)) = *filter P* (*y # ys*) **if** $x \neq y$ **and** *rel x y*
    using *that x* **by** (*rule 3*(*2*))
  have *c*: *filter P* (*merge-wrt rel* (*x # xs*) *ys*) = *filter P ys* **if** $x \neq y$ **and** $\neg$ *rel x y*
    using *that 3*(*4*) **by** (*rule 3*(*3*))
  show *?case* **by** (*simp add*: *a b c* ‹¬ *P x*›)
qed

lemma *length-filter-le-1*:
  assumes *length* (*filter P xs*) $\leq$ *1* **and** $i <$ *length xs* **and** $j <$ *length xs*
    and *P* (*xs ! i*) **and** *P* (*xs ! j*)
  shows $i = j$
proof –
  have $*$: *thesis* **if** $a < b$ **and** $b <$ *length xs*
    and $\bigwedge$*as bs cs. as* @ ((*xs ! a*) # (*bs* @ ((*xs ! b*) # *cs*))) = *xs* $\Longrightarrow$ *thesis* **for** *a b thesis*
  proof (*rule that*(*3*))

from *that*(*1*, *2*) **have** *1*: *a* < *length xs* **by** *simp*
with *that*(*1*, *2*) **have** *2*: *b* − *Suc a* < *length* (*drop* (*Suc a*) *xs*) **by** *simp*
from *that*(*1*) ‹*a* < *length xs*› **have** *eq*: *xs* ! *b* = *drop* (*Suc a*) *xs* ! (*b* − *Suc a*) **by** *simp*
show (*take a xs*) @ ((*xs* ! *a*) # ((*take* (*b* − *Suc a*) (*drop* (*Suc a*) *xs*)) @ ((*xs* ! *b*) #
drop (*Suc* (*b* − *Suc a*)) (*drop* (*Suc a*) *xs*)))) = *xs*
by (*simp only*: *eq id-take-nth-drop*[*OF 1*, *symmetric*] *id-take-nth-drop*[*OF 2*, *symmetric*])
qed
show *?thesis*
proof (*rule linorder-cases*)
assume *i* < *j*
then obtain *as bs cs* **where** *as* @ ((*xs* ! *i*) # (*bs* @ ((*xs* ! *j*) # *cs*))) = *xs*
using *assms*(*3*) **by** (*rule* ∗)
hence *filter P xs* = *filter P* (*as* @ ((*xs* ! *i*) # (*bs* @ ((*xs* ! *j*) # *cs*)))) **by** *simp*
also from *assms*(*4*, *5*) **have** ... = (*filter P as*) @ ((*xs* ! *i*) # ((*filter P bs*) @ ((*xs* ! *j*) # (*filter P cs*))))
by *simp*
finally have ¬ *length* (*filter P xs*) ≤ *1* **by** *simp*
thus *?thesis* **using** *assms*(*1*) **..**
next
assume *j* < *i*
then obtain *as bs cs* **where** *as* @ ((*xs* ! *j*) # (*bs* @ ((*xs* ! *i*) # *cs*))) = *xs*
using *assms*(*2*) **by** (*rule* ∗)
hence *filter P xs* = *filter P* (*as* @ ((*xs* ! *j*) # (*bs* @ ((*xs* ! *i*) # *cs*)))) **by** *simp*
also from *assms*(*4*, *5*) **have** ... = (*filter P as*) @ ((*xs* ! *j*) # ((*filter P bs*) @ ((*xs* ! *i*) # (*filter P cs*))))
by *simp*
finally have ¬ *length* (*filter P xs*) ≤ *1* **by** *simp*
thus *?thesis* **using** *assms*(*1*) **..**
qed
qed

lemma *length-filter-eq* [*simp*]: *length* (*filter* ((=) *x*) *xs*) = *count-list xs x*
by (*induct xs*, *simp-all*)

2.1.3 *drop*

lemma *nth-in-set-dropI*:
assumes *j* ≤ *i* **and** *i* < *length xs*
shows *xs* ! *i* ∈ *set* (*drop j xs*)
using *assms*
proof (*induct xs arbitrary*: *i j*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons x xs*)
show *?case*

proof (*cases j*)
  case *0*
  with *Cons*(*3*) **show** *?thesis* **by** (*metis drop0 nth-mem*)
next
  case (*Suc j0*)
  with *Cons*(*2*) *Suc-le-D* **obtain** *i0* **where** *i*: $i = Suc\ i0$ **by** *blast*
  with *Cons*(*2*) **have** $j0 \le i0$ **by** (*simp add*: ‹$j = Suc\ j0$›)
  moreover from *Cons*(*3*) **have** $i0 < length\ xs$ **by** (*simp add*: *i*)
  ultimately have $xs\ !\ i0 \in set\ (drop\ j0\ xs)$ **by** (*rule Cons*(*1*))
  thus *?thesis* **by** (*simp add*: *i* ‹$j = Suc\ j0$›)
qed
qed

2.1.4 *count-list*

lemma *count-list-upt* [*simp*]: $count\text{-}list\ [a..<b]\ x = (if\ a \le x \land x < b\ then\ 1\ else\ 0)$
proof (*cases* $a \le b$)
  case *True*
  then obtain *k* **where** $b = a + k$ **using** *le-Suc-ex* **by** *blast*
  show *?thesis* **unfolding** ‹$b = a + k$› **by** (*induct k*, *simp-all*)
next
  case *False*
  thus *?thesis* **by** *simp*
qed

2.1.5 *sorted-wrt*

lemma *sorted-wrt-upt-iff*: $sorted\text{-}wrt\ rel\ [a..<b] \longleftrightarrow (\forall i\ j.\ a \le i \longrightarrow i < j \longrightarrow j < b \longrightarrow rel\ i\ j)$
proof (*cases* $a \le b$)
  case *True*
  then obtain *k* **where** $b = a + k$ **using** *le-Suc-ex* **by** *blast*
  show *?thesis* **unfolding** ‹$b = a + k$›
  proof (*induct k*)
    case *0*
    show *?case* **by** *simp*
  next
    case (*Suc k*)
    show *?case*
    proof (*simp add*: *sorted-wrt-append Suc*, *intro iffI allI ballI impI conjI*)
      fix *i j*
      assume $(\forall i \ge a.\ \forall j > i.\ j < a + k \longrightarrow rel\ i\ j) \land (\forall x \in \{a..<a + k\}.\ rel\ x\ (a + k))$
      hence *1*: $\bigwedge i'\ j'.\ a \le i' \Longrightarrow i' < j' \Longrightarrow j' < a + k \Longrightarrow rel\ i'\ j'$
        and *2*: $\bigwedge x.\ a \le x \Longrightarrow x < a + k \Longrightarrow rel\ x\ (a + k)$ **by** *simp-all*
      assume $a \le i$ **and** $i < j$
      assume $j < Suc\ (a + k)$
      hence $j < a + k \lor j = a + k$ **by** *auto*
      thus *rel i j*

proof
  assume $j < a + k$
  with ‹$a \leq i$› ‹$i < j$› **show** *?thesis* **by** (*rule 1*)
next
  assume $j = a + k$
  from ‹$a \leq i$› ‹$i < j$› **show** *?thesis* **unfolding** ‹$j = a + k$› **by** (*rule 2*)
qed
next
  fix *i j*
  assume $\forall i \geq a.\ \forall j > i.\ j < Suc\ (a + k) \longrightarrow rel\ i\ j$ **and** $a \leq i$ **and** $i < j$ **and** $j < a + k$
  thus *rel i j* **by** *simp*
next
  fix *x*
  assume $x \in \{a..<a + k\}$
  hence $a \leq x$ **and** $x < a + k$ **by** *simp-all*
  moreover assume $\forall i \geq a.\ \forall j > i.\ j < Suc\ (a + k) \longrightarrow rel\ i\ j$
  ultimately show $rel\ x\ (a + k)$ **by** *simp*
qed
qed
next
case *False*
thus *?thesis* **by** *simp*
qed

2.1.6 *insort-wrt* and *merge-wrt*

lemma *map-insort-wrt*:
  assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow r2\ (f\ y)\ (f\ x) \longleftrightarrow r1\ y\ x$
  shows $map\ f\ (insort\text{-}wrt\ r1\ y\ xs) = insort\text{-}wrt\ r2\ (f\ y)\ (map\ f\ xs)$
  using *assms*
proof (*induct xs*)
  case *Nil*
  show *?case* **by** *simp*
next
  case (*Cons x xs*)
  have $x \in set\ (x \# xs)$ **by** *simp*
  hence $r2\ (f\ y)\ (f\ x) = r1\ y\ x$ **by** (*rule Cons*(*2*))
  moreover have $map\ f\ (insort\text{-}wrt\ r1\ y\ xs) = insort\text{-}wrt\ r2\ (f\ y)\ (map\ f\ xs)$
  proof (*rule Cons*(*1*))
    fix x'
    assume $x' \in set\ xs$
    hence $x' \in set\ (x \# xs)$ **by** *simp*
    thus $r2\ (f\ y)\ (f\ x') = r1\ y\ x'$ **by** (*rule Cons*(*2*))
  qed
  ultimately show *?case* **by** *simp*
qed

lemma *map-merge-wrt*:

assumes f ‘ *set xs* $\cap$ f ‘ *set ys* = {}
and $\bigwedge x\ y.\ x \in$ *set xs* $\Longrightarrow y \in$ *set ys* $\Longrightarrow$ *r2* $(f\ x)$ $(f\ y) \longleftrightarrow$ *r1 x y*
shows *map f* (*merge-wrt r1 xs ys*) = *merge-wrt r2* (*map f xs*) (*map f ys*)
using *assms*
proof (*induct r1 xs ys rule*: *merge-wrt.induct*)
case (*1 uu xs*)
show *?case* **by** *simp*
next
case (*2 r1 v va*)
show *?case* **by** *simp*
next
case (*3 r1 x xs y ys*)
from *3*(*4*) **have** $f\ x \neq f\ y$ **and** *1*: f ‘ *set xs* $\cap$ f ‘ *set* (y # *ys*) = {}
and *2*: f ‘ *set* (x # *xs*) $\cap$ f ‘ *set ys* = {} **by** *auto*
from *this*(*1*) **have** $x \neq y$ **by** *auto*
have *eq2*: *map f* (*merge-wrt r1 xs* (y # *ys*)) = *merge-wrt r2* (*map f xs*) (*map f* (y # *ys*))
if *r1 x y* **using** ‹$x \neq y$› *that 1*
proof (*rule 3*(*2*))
fix *a b*
assume $a \in$ *set xs*
hence $a \in$ *set* (x # *xs*) **by** *simp*
moreover assume $b \in$ *set* (y # *ys*)
ultimately show *r2* ($f\ a$) ($f\ b$) $\longleftrightarrow$ *r1 a b* **by** (*rule 3*(*5*))
qed
have *eq3*: *map f* (*merge-wrt r1* (x # *xs*) *ys*) = *merge-wrt r2* (*map f* (x # *xs*)) (*map f ys*)
if $\neg$ *r1 x y* **using** ‹$x \neq y$› *that 2*
proof (*rule 3*(*3*))
fix *a b*
assume $a \in$ *set* (x # *xs*)
assume $b \in$ *set ys*
hence $b \in$ *set* (y # *ys*) **by** *simp*
with ‹$a \in$ *set* (x # *xs*)› **show** *r2* ($f\ a$) ($f\ b$) $\longleftrightarrow$ *r1 a b* **by** (*rule 3*(*5*))
qed
have *eq4*: *r2* ($f\ x$) ($f\ y$) $\longleftrightarrow$ *r1 x y* **by** (*rule 3*(*5*), *simp-all*)
show *?case* **by** (*simp add*: *eq2 eq3 eq4* ‹$f\ x \neq f\ y$› ‹$x \neq y$›)
qed

2.2 Recursive Functions

locale *recursive* =
fixes $h' :: 'b \Rightarrow 'b$
fixes $b :: 'b$
assumes *b-fixpoint*: $h'\ b = b$
begin

context
fixes $Q :: 'a \Rightarrow bool$

```
  fixes g :: 'a ⇒ 'b
  fixes h :: 'a ⇒ 'a
begin

function (domintros) recfun-aux :: 'a ⇒ 'b where
  recfun-aux x = (if Q x then g x else h' (recfun-aux (h x)))
  by pat-completeness auto

lemmas [induct del] = recfun-aux.pinduct

definition dom :: 'a ⇒ bool
  where dom x ⟷ (∃ k. Q ((h ^^ k) x))

lemma domI:
  assumes ¬ Q x ⟹ dom (h x)
  shows dom x
proof (cases Q x)
  case True
  hence Q ((h ^^ 0) x) by simp
  thus ?thesis unfolding dom-def ..
next
  case False
  hence dom (h x) by (rule assms)
  then obtain k where Q ((h ^^ k) (h x)) unfolding dom-def ..
  hence Q ((h ^^ (Suc k)) x) by (simp add: funpow-swap1)
  thus ?thesis unfolding dom-def ..
qed

lemma domD:
  assumes dom x and ¬ Q x
  shows dom (h x)
proof −
  from assms(1) obtain k where ∗: Q ((h ^^ k) x) unfolding dom-def ..
  with assms(2) have k ≠ 0 using funpow-0 by fastforce
  then obtain m where k = Suc m using nat.exhaust by blast
  with ∗ have Q ((h ^^ m) (h x)) by (simp add: funpow-swap1)
  thus ?thesis unfolding dom-def ..
qed

lemma recfun-aux-domI:
  assumes dom x
  shows recfun-aux-dom x
proof −
  from assms obtain k where Q ((h ^^ k) x) unfolding dom-def ..
  thus ?thesis
  proof (induct k arbitrary: x)
    case 0
    hence Q x by simp
    with recfun-aux.domintros show ?case by blast
```

next
case (*Suc k*)
from *Suc*(*2*) **have** Q ((h ^^ k) (h x)) **by** (*simp add*: *funpow-swap1*)
hence *recfun-aux-dom* (h x) **by** (*rule Suc*(*1*))
with *recfun-aux.domintros* **show** *?case* **by** *blast*
qed
qed

lemma *recfun-aux-domD*:
assumes *recfun-aux-dom x*
shows *dom x*
using *assms*
proof (*induct x rule*: *recfun-aux.pinduct*)
case (*1 x*)
show *?case*
proof (*cases Q x*)
case *True*
with *domI* **show** *?thesis* **by** *blast*
next
case *False*
hence *dom* (h x) **by** (*rule 1*(*2*))
thus *?thesis* **using** *domI* **by** *blast*
qed
qed

corollary *recfun-aux-dom-alt*: *recfun-aux-dom* = *dom*
by (*auto dest*: *recfun-aux-domI recfun-aux-domD*)

definition *fun* :: $'a \Rightarrow 'b$
where *fun x* = (*if recfun-aux-dom x then recfun-aux x else b*)

lemma *simps*: *fun x* = (*if Q x then g x else h'* (*fun* (h x)))
proof (*cases dom x*)
case *True*
hence *dom*: *recfun-aux-dom x* **by** (*rule recfun-aux-domI*)
show *?thesis*
proof (*cases Q x*)
case *True*
with *dom* **show** *?thesis* **by** (*simp add*: *fun-def recfun-aux.psimps*)
next
case *False*
have *recfun-aux-dom* (h x) **by** (*rule recfun-aux-domI*, *rule domD*, *fact True*, *fact False*)
thus *?thesis* **by** (*simp add*: *fun-def dom False recfun-aux.psimps*)
qed
next
case *False*
moreover have $\neg$ Q x
proof

assume *Q x*
hence *dom x* **using** *domI* **by** *blast*
with *False* **show** *False* **..**
qed
moreover have ¬ *dom* (*h x*)
proof
assume *dom* (*h x*)
hence *dom x* **using** *domI* **by** *blast*
with *False* **show** *False* **..**
qed
ultimately show *?thesis* **by** (*simp add*: *recfun-aux-dom-alt fun-def b-fixpoint split del*: *if-split*)
qed

lemma *eq-fixpointI*: ¬ *dom x* $\Longrightarrow$ *fun x* = *b*
by (*simp add*: *fun-def recfun-aux-dom-alt*)

lemma *pinduct*: *dom x* $\Longrightarrow$ ($\bigwedge$*x*. *dom x* $\Longrightarrow$ (¬ *Q x* $\Longrightarrow$ *P* (*h x*)) $\Longrightarrow$ *P x*) $\Longrightarrow$ *P x*
unfolding *recfun-aux-dom-alt*[*symmetric*] **by** (*fact recfun-aux.pinduct*)

end

end

interpretation *tailrec*: *recursive* λ*x*. *x undefined*
by (*standard*, *fact refl*)

2.3 Binary Relations

lemma *almost-full-on-Int*:
assumes *almost-full-on P1 A1* **and** *almost-full-on P2 A2*
shows *almost-full-on* (λ*x y*. *P1 x y* ∧ *P2 x y*) (*A1* ∩ *A2*) (**is** *almost-full-on ?P ?A*)
proof (*rule almost-full-onI*)
fix *f* :: *nat* ⇒ *'a*
assume *a*: ∀ *i*. *f i* ∈ *?A*
define *g* **where** *g* = (λ*i*. (*f i*, *f i*))
from *assms* **have** *almost-full-on* (*prod-le P1 P2*) (*A1* × *A2*) **by** (*rule almost-full-on-Sigma*)
moreover from *a* **have** $\bigwedge$*i*. *g i* ∈ *A1* × *A2* **by** (*simp add*: *g-def*)
ultimately obtain *i j* **where** *i* < *j* **and** *prod-le P1 P2* (*g i*) (*g j*) **by** (*rule almost-full-onD*)
from *this*(*2*) **have** *?P* (*f i*) (*f j*) **by** (*simp add*: *g-def prod-le-def*)
with ‹*i* < *j*› **show** *good ?P f* **by** (*rule goodI*)
qed

corollary *almost-full-on-same*:
assumes *almost-full-on P1 A* **and** *almost-full-on P2 A*

shows *almost-full-on* $(\lambda x\ y.\ P1\ x\ y \wedge P2\ x\ y)\ A$
proof −
from *assms* **have** *almost-full-on* $(\lambda x\ y.\ P1\ x\ y \wedge P2\ x\ y)\ (A \cap A)$ **by** (*rule almost-full-on-Int*)
thus *?thesis* **by** *simp*
qed

context *ord*
begin

definition *is-le-rel* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
where *is-le-rel rel* $= (rel = (=) \vee rel = (\leq) \vee rel = (<))$

lemma *is-le-relI* [*simp*]: *is-le-rel* $(=)$ *is-le-rel* $(\leq)$ *is-le-rel* $(<)$
by (*simp-all add*: *is-le-rel-def*)

lemma *is-le-relE*:
assumes *is-le-rel rel*
obtains $rel = (=)$ | $rel = (\leq)$ | $rel = (<)$
using *assms* **unfolding** *is-le-rel-def* **by** *blast*

end

context *preorder*
begin

lemma *is-le-rel-le*:
assumes *is-le-rel rel*
shows $rel\ x\ y \Longrightarrow x \leq y$
using *assms* **by** (*rule is-le-relE*, *auto dest*: *less-imp-le*)

lemma *is-le-rel-trans*:
assumes *is-le-rel rel*
shows $rel\ x\ y \Longrightarrow rel\ y\ z \Longrightarrow rel\ x\ z$
using *assms* **by** (*rule is-le-relE*, *auto dest*: *order-trans less-trans*)

lemma *is-le-rel-trans-le-left*:
assumes *is-le-rel rel*
shows $x \leq y \Longrightarrow rel\ y\ z \Longrightarrow x \leq z$
using *assms* **by** (*rule is-le-relE*, *auto dest*: *order-trans le-less-trans less-imp-le*)

lemma *is-le-rel-trans-le-right*:
assumes *is-le-rel rel*
shows $rel\ x\ y \Longrightarrow y \leq z \Longrightarrow x \leq z$
using *assms* **by** (*rule is-le-relE*, *auto dest*: *order-trans less-le-trans less-imp-le*)

lemma *is-le-rel-trans-less-left*:
assumes *is-le-rel rel*
shows $x < y \Longrightarrow rel\ y\ z \Longrightarrow x < z$

using *assms* **by** (*rule is-le-relE*, *auto dest*: *less-le-trans less-imp-le*)

lemma *is-le-rel-trans-less-right*:
assumes *is-le-rel rel*
shows $rel\ x\ y \Longrightarrow y < z \Longrightarrow x < z$
using *assms* **by** (*rule is-le-relE*, *auto dest*: *le-less-trans less-imp-le*)

end

context *order*
begin

lemma *is-le-rel-distinct*:
assumes *is-le-rel rel*
shows $rel\ x\ y \Longrightarrow x \neq y \Longrightarrow x < y$
using *assms* **by** (*rule is-le-relE*, *auto*)

lemma *is-le-rel-antisym*:
assumes *is-le-rel rel*
shows $rel\ x\ y \Longrightarrow rel\ y\ x \Longrightarrow x = y$
using *assms* **by** (*rule is-le-relE*, *auto*)

end

end

3 More Properties of Power-Products and Multivariate Polynomials

theory *More-MPoly*
imports *Prelims Polynomials.MPoly-Type-Class-Ordered*
begin

3.1 Power-Products

lemma (**in** *comm-powerprod*) *minus-plus'*: $s\ adds\ t \Longrightarrow u + (t - s) = (u + t) - s$
using *add-commute minus-plus* **by** *auto*

context *ulcs-powerprod*
begin

lemma *lcs-alt-2*:
assumes $a + x = b + y$
shows $lcs\ x\ y = (b + y) - gcs\ a\ b$
proof −
have $a + (lcs\ x\ y + gcs\ a\ b) = lcs\ (a + x)\ (a + y) + gcs\ a\ b$ **by** (*simp only*: *lcs-plus-left ac-simps*)

also have ... = *lcs* (*b* + *y*) (*a* + *y*) + *gcs a b* **by** (*simp only*: *assms*)
also have ... = (*lcs a b* + *y*) + *gcs a b* **by** (*simp only*: *lcs-plus-right lcs-comm*)
also have ... = (*gcs a b* + *lcs a b*) + *y* **by** (*simp only*: *ac-simps*)
also have ... = *a* + (*b* + *y*) **by** (*simp only*: *gcs-plus-lcs*, *simp add*: *ac-simps*)
finally have (*lcs x y* + *gcs a b*) − *gcs a b* = (*b* + *y*) − *gcs a b* **by** *simp*
thus *?thesis* **by** *simp*
qed

corollary *lcs-alt-1*:
assumes *a* + *x* = *b* + *y*
shows *lcs x y* = (*a* + *x*) − *gcs a b*
proof −
have *lcs x y* = *lcs y x* **by** (*simp only*: *lcs-comm*)
also from *assms*[*symmetric*] **have** ... = (*a* + *x*) − *gcs b a* **by** (*rule lcs-alt-2*)
also have ... = (*a* + *x*) − *gcs a b* **by** (*simp only*: *gcs-comm*)
finally show *?thesis* **.**
qed

corollary *lcs-minus-1*:
assumes *a* + *x* = *b* + *y*
shows *lcs x y* − *x* = *a* − *gcs a b*
by (*simp add*: *lcs-alt-1*[*OF assms*] *diff-right-commute*)

corollary *lcs-minus-2*:
assumes *a* + *x* = *b* + *y*
shows *lcs x y* − *y* = *b* − *gcs a b*
by (*simp add*: *lcs-alt-2*[*OF assms*] *diff-right-commute*)

lemma *gcs-minus*:
assumes *u adds s* **and** *u adds t*
shows *gcs* (*s* − *u*) (*t* − *u*) = *gcs s t* − *u*
proof −
from *assms* **have** *gcs s t* = *gcs* ((*s* − *u*) + *u*) ((*t* − *u*) + *u*) **by** (*simp add*: *minus-plus*)
also have ... = *gcs* (*s* − *u*) (*t* − *u*) + *u* **by** (*simp only*: *gcs-plus-right*)
finally show *?thesis* **by** *simp*
qed

corollary *gcs-minus-gcs*: *gcs* (*s* − (*gcs s t*)) (*t* − (*gcs s t*)) = *0*
by (*simp add*: *gcs-minus gcs-adds gcs-adds-2*)

end

3.2 Miscellaneous

lemma *poly-mapping-rangeE*:
assumes *c* ∈ *Poly-Mapping.range p*
obtains *k* **where** *k* ∈ *keys p* **and** *c* = *lookup p k*
using *assms* **by** (*transfer*, *auto*)

lemma *poly-mapping-range-nonzero*: $0 \notin$ *Poly-Mapping.range p*
by (*transfer*, *auto*)

lemma (**in** *term-powerprod*) *Keys-range-vectorize-poly*: *Keys* (*Poly-Mapping.range* (*vectorize-poly p*)) = *pp-of-term* ‘ *keys p*
proof
show *Keys* (*Poly-Mapping.range* (*vectorize-poly p*)) ⊆ *pp-of-term* ‘ *keys p*
proof
fix *t*
assume *t* ∈ *Keys* (*Poly-Mapping.range* (*vectorize-poly p*))
then obtain *q* **where** *q* ∈ *Poly-Mapping.range* (*vectorize-poly p*) **and** *t* ∈ *keys q* **by** (*rule in-KeysE*)
from *this*(*1*) **obtain** *k* **where** *q*: *q* = *lookup* (*vectorize-poly p*) *k* **by** (*metis DiffE imageE range.rep-eq*)
with ‹*t* ∈ *keys q*› **have** *term-of-pair* (*t*, *k*) ∈ *keys p*
by (*metis in-keys-iff lookup-proj-poly lookup-vectorize-poly*)
hence *pp-of-term* (*term-of-pair* (*t*, *k*)) ∈ *pp-of-term* ‘ *keys p* **by** (*rule imageI*)
thus *t* ∈ *pp-of-term* ‘ *keys p* **by** (*simp only*: *pp-of-term-of-pair*)
qed
next
show *pp-of-term* ‘ *keys p* ⊆ *Keys* (*Poly-Mapping.range* (*vectorize-poly p*))
proof
fix *t*
assume *t* ∈ *pp-of-term* ‘ *keys p*
then obtain *x* **where** *x* ∈ *keys p* **and** *t* = *pp-of-term x* **..**
from *this*(*2*) **have** *term-of-pair* (*t*, *component-of-term x*) = *x* **by** (*simp add*: *term-of-pair-pair*)
with ‹*x* ∈ *keys p*› **have** *lookup p* (*term-of-pair* (*t*, *component-of-term x*)) ≠ *0*
by (*simp add*: *in-keys-iff*)
hence *lookup* (*proj-poly* (*component-of-term x*) *p*) *t* ≠ *0* **by** (*simp add*: *lookup-proj-poly*)
hence *t*: *t* ∈ *keys* (*proj-poly* (*component-of-term x*) *p*)
by (*simp add*: *in-keys-iff*)
from ‹*x* ∈ *keys p*› **have** *component-of-term x* ∈ *keys* (*vectorize-poly p*)
by (*simp add*: *keys-vectorize-poly*)
from *t* **show** *t* ∈ *Keys* (*Poly-Mapping.range* (*vectorize-poly p*))
proof (*rule in-KeysI*)
have *proj-poly* (*component-of-term x*) *p* = *lookup* (*vectorize-poly p*) (*component-of-term x*)
by (*simp only*: *lookup-vectorize-poly*)
also from ‹*component-of-term x* ∈ *keys* (*vectorize-poly p*)›
have ... ∈ *Poly-Mapping.range* (*vectorize-poly p*) **by** (*rule in-keys-lookup-in-range*)
finally show *proj-poly* (*component-of-term x*) *p* ∈ *Poly-Mapping.range* (*vectorize-poly p*) **.**
qed
qed
qed

3.3 *ordered-term.lt* and *ordered-term.higher*

context *ordered-term*
begin

lemma *lt-lookup-vectorize*: *punit.lt* (*lookup* (*vectorize-poly p*) (*component-of-term* (*lt p*))) = *lp p*
proof (*cases p* = *0*)
  case *True*
  thus *?thesis* **by** (*simp add*: *vectorize-zero min-term-def pp-of-term-of-pair*)
next
  case *False*
  show *?thesis*
  proof (*rule punit.lt-eqI-keys*)
    from *False* **have** *lt p* ∈ *keys p* **by** (*rule lt-in-keys*)
    thus *lp p* ∈ *keys* (*lookup* (*vectorize-poly p*) (*component-of-term* (*lt p*)))
      by (*simp add*: *lookup-vectorize-poly keys-proj-poly*)
  next
    fix *t*
    assume *t* ∈ *keys* (*lookup* (*vectorize-poly p*) (*component-of-term* (*lt p*)))
   also have ... = *pp-of-term* ‘ {*x*∈*keys p*. *component-of-term x* = *component-of-term* (*lt p*)}
      by (*simp only*: *lookup-vectorize-poly keys-proj-poly*)
    finally obtain *v* **where** *v* ∈ *keys p* **and** *1*: *component-of-term v* = *component-of-term* (*lt p*)
      and *t*: *t* = *pp-of-term v* **by** *auto*
    from *this*(*1*) **have** $v \preceq_t$ *lt p* **by** (*rule lt-max-keys*)
    show *t* ⪯ *lp p*
    proof (*rule ccontr*)
      assume ¬ *t* ⪯ *lp p*
      hence *lp p* ≺ *pp-of-term v* **by** (*simp add*: *t*)
      hence *lp p* ≠ *pp-of-term v* **and** *lp p* ⪯ *pp-of-term v* **by** *simp-all*
      note *this*(*2*)
     moreover from *1* **have** *component-of-term* (*lt p*) ≤ *component-of-term v* **by** *simp*
      ultimately have *lt p* $\preceq_t$ *v* **by** (*rule ord-termI*)
      with ‹$v \preceq_t$ *lt p*› **have** *v* = *lt p*
        by *simp*
      with ‹*lp p* ≠ *pp-of-term v*› **show** *False* **by** *simp*
    qed
  qed
qed

lemma *lower-higher-zeroI*: $u \preceq_t v \Longrightarrow$ *lower* (*higher p v*) *u* = *0*
  by (*simp add*: *lower-eq-zero-iff lookup-higher-when*)

lemma *lookup-minus-higher*: *lookup* (*p* − *higher p v*) *u* = (*lookup p u when* $u \preceq_t v$)
  by (*auto simp*: *lookup-minus lookup-higher-when when-def*)

lemma *keys-minus-higher*: *keys* $(p - \mathit{higher}\ p\ v) = \{u \in \mathit{keys}\ p.\ u \preceq_t v\}$
by (*rule set-eqI*, *simp add*: *lookup-minus-higher conj-commute flip*: *lookup-not-eq-zero-eq-in-keys*)

lemma *lt-minus-higher*: $v \in \mathit{keys}\ p \Longrightarrow \mathit{lt}\ (p - \mathit{higher}\ p\ v) = v$
by (*rule lt-eqI-keys*, *simp-all add*: *keys-minus-higher*)

lemma *lc-minus-higher*: $v \in \mathit{keys}\ p \Longrightarrow \mathit{lc}\ (p - \mathit{higher}\ p\ v) = \mathit{lookup}\ p\ v$
by (*simp add*: *lc-def lt-minus-higher lookup-minus-higher*)

lemma *tail-minus-higher*: $v \in \mathit{keys}\ p \Longrightarrow \mathit{tail}\ (p - \mathit{higher}\ p\ v) = \mathit{lower}\ p\ v$
by (*rule poly-mapping-eqI*, *simp add*: *lookup-tail-when lt-minus-higher lookup-lower-when lookup-minus-higher cong*: *when-cong*)

end

3.4 *gd-term.dgrad-p-set*

lemma (**in** *gd-term*) *dgrad-p-set-closed-mult-scalar*:
assumes *dickson-grading d* **and** $p \in$ *punit.dgrad-p-set d m* **and** $r \in$ *dgrad-p-set d m*
shows $p \odot r \in$ *dgrad-p-set d m*
proof (*rule dgrad-p-setI*)
fix *v*
assume $v \in \mathit{keys}\ (p \odot r)$
then obtain *t u* **where** $t \in \mathit{keys}\ p$ **and** $u \in \mathit{keys}\ r$ **and** *v*: $v = t \oplus u$
by (*rule in-keys-mult-scalarE*)
from *assms*(*2*) ‹$t \in \mathit{keys}\ p$› **have** $d\ t \le m$ **by** (*rule punit.dgrad-p-setD*[*simplified*])
moreover from *assms*(*3*) ‹$u \in \mathit{keys}\ r$› **have** d (*pp-of-term u*) $\le m$ **by** (*rule dgrad-p-setD*)
ultimately have d (t + *pp-of-term u*) $\le m$ **using** *assms*(*1*) **by** (*simp add*: *dickson-gradingD1*)
thus d (*pp-of-term v*) $\le m$ **by** (*simp only*: *v pp-of-term-splus*)
qed

3.5 Regular Sequences

definition *is-regular-sequence* :: ($'a$::*comm-powerprod* $\Rightarrow_0$ $'b$::*comm-ring-1*) *list* $\Rightarrow$ *bool*
where *is-regular-sequence fs* $\longleftrightarrow$ ($\forall j<$*length fs*. $\forall q.\ q * \mathit{fs}\ !\ j \in$ *ideal* (*set* (*take j fs*)) $\longrightarrow$
$q \in$ *ideal* (*set* (*take j fs*)))

lemma *is-regular-sequenceD*:
is-regular-sequence fs $\Longrightarrow j <$ *length fs* $\Longrightarrow q * \mathit{fs}\ !\ j \in$ *ideal* (*set* (*take j fs*)) $\Longrightarrow$
$q \in$ *ideal* (*set* (*take j fs*))
by (*simp add*: *is-regular-sequence-def*)

lemma *is-regular-sequence-Nil*: *is-regular-sequence* []
by (*simp add*: *is-regular-sequence-def*)

lemma *is-regular-sequence-snocI*:
assumes $\bigwedge q.\ q * f \in ideal\ (set\ fs) \Longrightarrow q \in ideal\ (set\ fs)$ **and** *is-regular-sequence fs*
shows *is-regular-sequence* (*fs* @ [*f*])
proof (*simp add*: *is-regular-sequence-def*, *intro impI allI*)
fix *j q*
assume *1*: $j < Suc\ (length\ fs)$ **and** *2*: $q * (fs\ @\ [f])\ !\ j \in ideal\ (set\ (take\ j\ fs))$
show $q \in ideal\ (set\ (take\ j\ fs))$
proof (*cases* $j = length\ fs$)
case *True*
from *2* **have** $q * f \in ideal\ (set\ fs)$ **by** (*simp add*: *True*)
hence $q \in ideal\ (set\ fs)$ **by** (*rule assms*(*1*))
thus *?thesis* **by** (*simp add*: *True*)
next
case *False*
with *1* **have** $j < length\ fs$ **by** *simp*
with *2* **have** $q * fs\ !\ j \in ideal\ (set\ (take\ j\ fs))$ **by** (*simp add*: *nth-append*)
with *assms*(*2*) ‹$j < length\ fs$› **show** $q \in ideal\ (set\ (take\ j\ fs))$ **by** (*rule is-regular-sequenceD*)
qed
qed

lemma *is-regular-sequence-snocD*:
assumes *is-regular-sequence* (*fs* @ [*f*])
shows $\bigwedge q.\ q * f \in ideal\ (set\ fs) \Longrightarrow q \in ideal\ (set\ fs)$
and *is-regular-sequence fs*
proof −
fix *q*
assume *1*: $q * f \in ideal\ (set\ fs)$
note *assms*
moreover have $length\ fs < length\ (fs\ @\ [f])$ **by** *simp*
moreover from *1* **have** $q * (fs\ @\ [f])\ !\ (length\ fs) \in ideal\ (set\ (take\ (length\ fs)\ (fs\ @\ [f])))$
by *simp*
ultimately have $q \in ideal\ (set\ (take\ (length\ fs)\ (fs\ @\ [f])))$ **by** (*rule is-regular-sequenceD*)
thus $q \in ideal\ (set\ fs)$ **by** *simp*
next
show *is-regular-sequence fs* **unfolding** *is-regular-sequence-def*
proof (*intro impI allI*)
fix *j q*
assume *1*: $j < length\ fs$ **and** *2*: $q * fs\ !\ j \in ideal\ (set\ (take\ j\ fs))$
note *assms*
moreover from *1* **have** $j < length\ (fs\ @\ [f])$ **by** *simp*
moreover from *1 2* **have** $q * (fs\ @\ [f])\ !\ j \in ideal\ (set\ (take\ j\ (fs\ @\ [f])))$
by (*simp add*: *nth-append*)
ultimately have $q \in ideal\ (set\ (take\ j\ (fs\ @\ [f])))$ **by** (*rule is-regular-sequenceD*)
with *1* **show** $q \in ideal\ (set\ (take\ j\ fs))$ **by** *simp*
qed
qed

lemma *is-regular-sequence-removeAll-zero*:
  assumes *is-regular-sequence fs*
  shows *is-regular-sequence* (*removeAll 0 fs*)
  using *assms*
proof (*induct fs rule*: *rev-induct*)
  case *Nil*
  show *?case* **by** (*simp add*: *is-regular-sequence-Nil*)
next
  case (*snoc f fs*)
  have *set* (*removeAll 0 fs*) = *set fs* − {*0*} **by** *simp*
  also have *ideal* ... = *ideal* (*set fs*) **by** (*fact ideal.span-Diff-zero*)
  finally have *eq*: *ideal* (*set* (*removeAll 0 fs*)) = *ideal* (*set fs*) **.**
  from *snoc*(*2*) **have** ∗: *is-regular-sequence fs* **by** (*rule is-regular-sequence-snocD*)
  show *?case*
  proof (*simp*, *intro conjI impI*)
    show *is-regular-sequence* (*removeAll 0 fs* @ [*f*])
    proof (*rule is-regular-sequence-snocI*)
      fix *q*
      assume *q* ∗ *f* ∈ *ideal* (*set* (*removeAll 0 fs*))
      hence *q* ∗ *f* ∈ *ideal* (*set fs*) **by** (*simp only*: *eq*)
      with *snoc*(*2*) **have** *q* ∈ *ideal* (*set fs*) **by** (*rule is-regular-sequence-snocD*)
      thus *q* ∈ *ideal* (*set* (*removeAll 0 fs*)) **by** (*simp only*: *eq*)
    next
      from ∗ **show** *is-regular-sequence* (*removeAll 0 fs*) **by** (*rule snoc.hyps*)
    qed
  next
    from ∗ **show** *is-regular-sequence* (*removeAll 0 fs*) **by** (*rule snoc.hyps*)
  qed
qed

lemma *is-regular-sequence-remdups*:
  assumes *is-regular-sequence fs*
  shows *is-regular-sequence* (*rev* (*remdups* (*rev fs*)))
  using *assms*
proof (*induct fs rule*: *rev-induct*)
  case *Nil*
  show *?case* **by** (*simp add*: *is-regular-sequence-Nil*)
next
  case (*snoc f fs*)
  from *snoc*(*2*) **have** ∗: *is-regular-sequence fs* **by** (*rule is-regular-sequence-snocD*)
  show *?case*
  proof (*simp*, *intro conjI impI*)
    show *is-regular-sequence* (*rev* (*remdups* (*rev fs*)) @ [*f*])
    proof (*rule is-regular-sequence-snocI*)
      fix *q*
      assume *q* ∗ *f* ∈ *ideal* (*set* (*rev* (*remdups* (*rev fs*))))
      hence *q* ∗ *f* ∈ *ideal* (*set fs*) **by** *simp*
      with *snoc*(*2*) **have** *q* ∈ *ideal* (*set fs*) **by** (*rule is-regular-sequence-snocD*)

thus *q* ∈ *ideal* (*set* (*rev* (*remdups* (*rev fs*)))) **by** *simp*
next
from ∗ **show** *is-regular-sequence* (*rev* (*remdups* (*rev fs*))) **by** (*rule snoc.hyps*)
qed
next
from ∗ **show** *is-regular-sequence* (*rev* (*remdups* (*rev fs*))) **by** (*rule snoc.hyps*)
qed
qed

end

4 Signature-Based Algorithms for Computing Gröbner Bases

theory *Signature-Groebner*
imports *More-MPoly Groebner-Bases.Syzygy Polynomials.Quasi-PM-Power-Products*
begin

First, we develop the whole theory for elements of the module $K[X]^r$, i.e. objects of type $'t \Rightarrow_0 'b$. Later, we transfer all algorithms defined on such objects to algorithms efficiently operating on sig-poly-pairs, i.e. objects of type $'t \times ('a \Rightarrow_0 'b)$.

4.1 More Preliminaries

lemma (**in** *gd-term*) *lt-spoly-less-lcs*:
assumes *p* ≠ *0* **and** *q* ≠ *0* **and** *spoly p q* ≠ *0*
shows *lt* (*spoly p q*) $\prec_t$ *term-of-pair* (*lcs* (*lp p*) (*lp q*), *component-of-term* (*lt p*))
proof −
let *?l* = *lcs* (*lp p*) (*lp q*)
let *?p* = *monom-mult* (*1* / *lc p*) (*?l* − *lp p*) *p*
let *?q* = *monom-mult* (*1* / *lc q*) (*?l* − *lp q*) *q*
from *assms*(*3*) **have** *eq1*: *component-of-term* (*lt p*) = *component-of-term* (*lt q*)
and *eq2*: *spoly p q* = *?p* − *?q*
by (*simp-all add*: *spoly-def Let-def lc-def split*: *if-split-asm*)
from ‹*p* ≠ *0*› **have** *lc p* ≠ *0* **by** (*rule lc-not-0*)
with *assms*(*1*) **have** *lt ?p* = (*?l* − *lp p*) ⊕ *lt p* **and** *lc ?p* = *1* **by** (*simp-all add*: *lt-monom-mult*)
from *this*(*1*) **have** *lt-p*: *lt ?p* = *term-of-pair* (*?l*, *component-of-term* (*lt p*))
by (*simp add*: *splus-def adds-minus adds-lcs*)
from ‹*q* ≠ *0*› **have** *lc q* ≠ *0* **by** (*rule lc-not-0*)
with *assms*(*2*) **have** *lt ?q* = (*?l* − *lp q*) ⊕ *lt q* **and** *lc ?q* = *1* **by** (*simp-all add*: *lt-monom-mult*)
from *this*(*1*) **have** *lt-q*: *lt ?q* = *term-of-pair* (*?l*, *component-of-term* (*lt p*))
by (*simp add*: *eq1 splus-def adds-minus adds-lcs-2*)
from *assms*(*3*) **have** *?p* − *?q* ≠ *0* **by** (*simp add*: *eq2*)
moreover have *lt ?q* = *lt ?p* **by** (*simp only*: *lt-p lt-q*)
moreover have *lc ?q* = *lc ?p* **by** (*simp only*: ‹*lc ?p* = *1*› ‹*lc ?q* = *1*›)

ultimately have *lt* (*?p* − *?q*) $\prec_t$ *lt ?p* **by** (*rule lt-minus-lessI*)
thus *?thesis* **by** (*simp only*: *eq2 lt-p*)
qed

4.2 Module Polynomials

locale *qpm-inf-term* =
gd-term pair-of-term term-of-pair ord ord-strict ord-term ord-term-strict
for *pair-of-term*::′*t* ⇒ (′*a*::*quasi-pm-powerprod* × *nat*)
and *term-of-pair*::(′*a* × *nat*) ⇒ ′*t*
and *ord*::′*a* ⇒ ′*a* ⇒ *bool* (**infixl** ‹⪯› *50*)
and *ord-strict* (**infixl** ‹≺› *50*)
and *ord-term*::′*t* ⇒ ′*t* ⇒ *bool* (**infixl** ‹$\preceq_t$› *50*)
and *ord-term-strict*::′*t* ⇒ ′*t* ⇒ *bool* (**infixl** ‹$\prec_t$› *50*)
begin

lemma *in-idealE-rep-dgrad-p-set*:
assumes *hom-grading d* **and** *B* ⊆ *punit.dgrad-p-set d m* **and** *p* ∈ *punit.dgrad-p-set d m* **and** *p* ∈ *ideal B*
obtains *r* **where** *keys r* ⊆ *B* **and** *Poly-Mapping.range r* ⊆ *punit.dgrad-p-set d m* **and** *p* = *ideal.rep r*
proof −
from *assms* **obtain** *A q* **where** *finite A* **and** *A* ⊆ *B* **and** *0*: ⋀*b*. *q b* ∈ *punit.dgrad-p-set d m*
and *p*: *p* = (∑ *a*∈*A*. *q a* ∗ *a*) **by** (*rule punit.in-pmdlE-dgrad-p-set*[*simplified*], *blast*)
define *r* **where** *r* = *Abs-poly-mapping* (λ*k*. *q k when k* ∈ *A*)
have *1*: *lookup r* = (λ*k*. *q k when k* ∈ *A*) **unfolding** *r-def*
by (*rule Abs-poly-mapping-inverse*, *simp add*: ‹*finite A*›)
have *2*: *keys r* ⊆ *A* **by** (*auto simp*: *in-keys-iff 1*)
show *?thesis*
proof
show *Poly-Mapping.range r* ⊆ *punit.dgrad-p-set d m*
proof
fix *f*
assume *f* ∈ *Poly-Mapping.range r*
then obtain *b* **where** *b* ∈ *keys r* **and** *f*: *f* = *lookup r b* **by** (*rule poly-mapping-rangeE*)
from *this*(*1*) *2* **have** *b* ∈ *A* **..**
hence *f* = *q b* **by** (*simp add*: *f 1*)
show *f* ∈ *punit.dgrad-p-set d m* **unfolding** ‹*f* = *q b*› **by** (*rule 0*)
qed
next
have *p* = (∑ *a*∈*A*. *lookup r a* ∗ *a*) **unfolding** *p* **by** (*rule sum.cong*, *simp-all add*: *1*)
also from ‹*finite A*› *2* **have** ... = (∑ *a*∈*keys r*. *lookup r a* ∗ *a*)
proof (*rule sum.mono-neutral-right*)
show ∀ *a*∈*A* − *keys r*. *lookup r a* ∗ *a* = *0*
by (*simp add*: *in-keys-iff*)
qed

finally show $p = ideal.rep\ r$ **by** (*simp only*: *ideal.rep-def*)
next
from *2* ‹$A \subseteq B$› **show** *keys* $r \subseteq B$ **by** (*rule subset-trans*)
qed
qed

context fixes *fs* :: $('a \Rightarrow_0 'b{::}field)$ *list*
begin

definition *sig-inv-set'* :: *nat* $\Rightarrow$ $('t \Rightarrow_0 'b)$ *set*
where *sig-inv-set'* $j = \{r.$ *keys* (*vectorize-poly* r) $\subseteq \{0..<j\}\}$

abbreviation *sig-inv-set* $\equiv$ *sig-inv-set'* (*length fs*)

definition *rep-list* :: $('t \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'b)$
where *rep-list* r = *ideal.rep* (*pm-of-idx-pm fs* (*vectorize-poly* r))

lemma *sig-inv-setI*: *keys* (*vectorize-poly* r) $\subseteq \{0..<j\} \Longrightarrow r \in$ *sig-inv-set'* j
by (*simp add*: *sig-inv-set'-def*)

lemma *sig-inv-setD*: $r \in$ *sig-inv-set'* $j \Longrightarrow$ *keys* (*vectorize-poly* r) $\subseteq \{0..<j\}$
by (*simp add*: *sig-inv-set'-def*)

lemma *sig-inv-setI'*:
assumes $\bigwedge v.\ v \in$ *keys* $r \Longrightarrow$ *component-of-term* $v < j$
shows $r \in$ *sig-inv-set'* j
proof (*rule sig-inv-setI*, *rule*)
fix k
assume $k \in$ *keys* (*vectorize-poly* r)
then obtain v **where** $v \in$ *keys* r **and** k: k = *component-of-term* v **unfolding** *keys-vectorize-poly* **..**
from *this*(*1*) **have** $k < j$ **unfolding** k **by** (*rule assms*)
thus $k \in \{0..<j\}$ **by** *simp*
qed

lemma *sig-inv-setD'*:
assumes $r \in$ *sig-inv-set'* j **and** $v \in$ *keys* r
shows *component-of-term* $v < j$
proof −
from *assms*(*2*) **have** *component-of-term* $v \in$ *component-of-term* ‘ *keys* r **by** (*rule imageI*)
also have ... = *keys* (*vectorize-poly* r) **by** (*simp only*: *keys-vectorize-poly*)
also from *assms*(*1*) **have** ... $\subseteq \{0..<j\}$ **by** (*rule sig-inv-setD*)
finally show *?thesis* **by** *simp*
qed

corollary *sig-inv-setD-lt*:
assumes $r \in$ *sig-inv-set'* j **and** $r \neq 0$
shows *component-of-term* (*lt* r) $< j$

by (*rule sig-inv-setD'*, *fact*, *rule lt-in-keys*, *fact*)

lemma *sig-inv-set-mono*:
assumes $i \le j$
shows *sig-inv-set'* $i \subseteq$ *sig-inv-set'* j
proof
fix r
assume $r \in$ *sig-inv-set'* i
hence *keys* (*vectorize-poly* r) $\subseteq \{0..<i\}$ **by** (*rule sig-inv-setD*)
also from *assms* **have** ... $\subseteq \{0..<j\}$ **by** *fastforce*
finally show $r \in$ *sig-inv-set'* j **by** (*rule sig-inv-setI*)
qed

lemma *sig-inv-set-zero*: $0 \in$ *sig-inv-set'* j
by (*rule sig-inv-setI'*, *simp*)

lemma *sig-inv-set-closed-uminus*: $r \in$ *sig-inv-set'* $j \Longrightarrow - r \in$ *sig-inv-set'* j
by (*auto dest*!: *sig-inv-setD' intro*!: *sig-inv-setI' simp*: *keys-uminus*)

lemma *sig-inv-set-closed-plus*:
assumes $r \in$ *sig-inv-set'* j **and** $s \in$ *sig-inv-set'* j
shows $r + s \in$ *sig-inv-set'* j
proof (*rule sig-inv-setI'*)
fix v
assume $v \in$ *keys* $(r + s)$
hence $v \in$ *keys* $r \cup$ *keys* s **using** *Poly-Mapping.keys-add* **..**
thus *component-of-term* $v < j$
proof
assume $v \in$ *keys* r
with *assms*(*1*) **show** *?thesis* **by** (*rule sig-inv-setD'*)
next
assume $v \in$ *keys* s
with *assms*(*2*) **show** *?thesis* **by** (*rule sig-inv-setD'*)
qed
qed

lemma *sig-inv-set-closed-minus*:
assumes $r \in$ *sig-inv-set'* j **and** $s \in$ *sig-inv-set'* j
shows $r - s \in$ *sig-inv-set'* j
proof (*rule sig-inv-setI'*)
fix v
assume $v \in$ *keys* $(r - s)$
hence $v \in$ *keys* $r \cup$ *keys* s **using** *keys-minus* **..**
thus *component-of-term* $v < j$
proof
assume $v \in$ *keys* r
with *assms*(*1*) **show** *?thesis* **by** (*rule sig-inv-setD'*)
next
assume $v \in$ *keys* s

with *assms*(*2*) **show** *?thesis* **by** (*rule sig-inv-setD'*)
qed
qed

lemma *sig-inv-set-closed-monom-mult*:
assumes $r \in$ *sig-inv-set' j*
shows *monom-mult c t r* $\in$ *sig-inv-set' j*
proof (*rule sig-inv-setI'*)
fix *v*
assume $v \in$ *keys* (*monom-mult c t r*)
hence $v \in (\oplus)$ *t* ' *keys r* **using** *keys-monom-mult-subset* **..**
then obtain *u* **where** $u \in$ *keys r* **and** *v*: $v = t \oplus u$ **..**
from *assms this*(*1*) **have** *component-of-term* $u < j$ **by** (*rule sig-inv-setD'*)
thus *component-of-term* $v < j$ **by** (*simp add*: *v term-simps*)
qed

lemma *sig-inv-set-closed-mult-scalar*:
assumes $r \in$ *sig-inv-set' j*
shows $p \odot r \in$ *sig-inv-set' j*
proof (*rule sig-inv-setI'*)
fix *v*
assume $v \in$ *keys* ($p \odot r$)
then obtain *t u* **where** $u \in$ *keys r* **and** *v*: $v = t \oplus u$ **by** (*rule in-keys-mult-scalarE*)
from *assms this*(*1*) **have** *component-of-term* $u < j$ **by** (*rule sig-inv-setD'*)
thus *component-of-term* $v < j$ **by** (*simp add*: *v term-simps*)
qed

lemma *rep-list-zero*: *rep-list 0* = *0*
by (*simp add*: *rep-list-def vectorize-zero*)

lemma *rep-list-uminus*: *rep-list* ($-$ *r*) = $-$ *rep-list r*
by (*simp add*: *rep-list-def vectorize-uminus pm-of-idx-pm-uminus*)

lemma *rep-list-plus*: *rep-list* (*r* + *s*) = *rep-list r* + *rep-list s*
by (*simp add*: *rep-list-def vectorize-plus pm-of-idx-pm-plus ideal.rep-plus*)

lemma *rep-list-minus*: *rep-list* (*r* $-$ *s*) = *rep-list r* $-$ *rep-list s*
by (*simp add*: *rep-list-def vectorize-minus pm-of-idx-pm-minus ideal.rep-minus*)

lemma *vectorize-mult-scalar*:
vectorize-poly ($p \odot q$) = *MPoly-Type-Class.punit.monom-mult p 0* (*vectorize-poly q*)
by (*rule poly-mapping-eqI*, *simp add*: *lookup-vectorize-poly MPoly-Type-Class.punit.lookup-monom-mult-zero proj-mult-scalar*)

lemma *rep-list-mult-scalar*: *rep-list* ($c \odot r$) = $c *$ *rep-list r*
by (*simp add*: *rep-list-def vectorize-mult-scalar pm-of-idx-pm-monom-mult punit.rep-mult-scalar*[*simplified*])

lemma *rep-list-monom-mult*: *rep-list* (*monom-mult c t r*) = *punit.monom-mult c*

t (*rep-list r*)
  unfolding *mult-scalar-monomial*[*symmetric*] *times-monomial-left*[*symmetric*] **by**
(*rule rep-list-mult-scalar*)

lemma *rep-list-monomial*:
  assumes *distinct fs*
  shows *rep-list* (*monomial c u*) =
            (*punit.monom-mult c* (*pp-of-term u*) (*fs* ! (*component-of-term u*))
              when component-of-term u < *length fs*)
  by (*simp add*: *rep-list-def vectorize-monomial pm-of-idx-pm-monomial*[*OF assms*]
when-def times-monomial-left)

lemma *rep-list-in-ideal-sig-inv-set*:
  assumes *r* ∈ *sig-inv-set′ j*
  shows *rep-list r* ∈ *ideal* (*set* (*take j fs*))
proof −
  let *?fs* = *take j fs*
  from *assms* **have** *keys* (*vectorize-poly r*) ⊆ {*0*..<*j*} **by** (*rule sig-inv-setD*)
  hence *eq*: *pm-of-idx-pm fs* (*vectorize-poly r*) = *pm-of-idx-pm ?fs* (*vectorize-poly*
r)
    by (*simp only*: *pm-of-idx-pm-take*)
  have *rep-list r* ∈ *ideal* (*keys* (*pm-of-idx-pm fs* (*vectorize-poly r*)))
    unfolding *rep-list-def* **by** (*rule ideal.rep-in-span*)
  also have ... = *ideal* (*keys* (*pm-of-idx-pm ?fs* (*vectorize-poly r*))) **by** (*simp only*:
eq)
  also from *keys-pm-of-idx-pm-subset* **have** ... ⊆ *ideal* (*set ?fs*) **by** (*rule ideal.span-mono*)
  finally show *?thesis* **.**
qed

corollary *rep-list-subset-ideal-sig-inv-set*:
  B ⊆ *sig-inv-set′ j* ⟹ *rep-list* ‘ *B* ⊆ *ideal* (*set* (*take j fs*))
  by (*auto dest*: *rep-list-in-ideal-sig-inv-set*)

lemma *rep-list-in-ideal*: *rep-list r* ∈ *ideal* (*set fs*)
proof −
  have *rep-list r* ∈ *ideal* (*keys* (*pm-of-idx-pm fs* (*vectorize-poly r*)))
    unfolding *rep-list-def* **by** (*rule ideal.rep-in-span*)
  also from *keys-pm-of-idx-pm-subset* **have** ... ⊆ *ideal* (*set fs*) **by** (*rule ideal.span-mono*)
  finally show *?thesis* **.**
qed

corollary *rep-list-subset-ideal*: *rep-list* ‘ *B* ⊆ *ideal* (*set fs*)
  by (*auto intro*: *rep-list-in-ideal*)

lemma *in-idealE-rep-list*:
  assumes *p* ∈ *ideal* (*set fs*)
  obtains *r* **where** *p* = *rep-list r* **and** *r* ∈ *sig-inv-set*
proof −
  from *assms* **obtain** *r0* **where** *r0*: *keys r0* ⊆ *set fs* **and** *p*: *p* = *ideal.rep r0*

by (*rule ideal.spanE-rep*)
show *?thesis*
proof
show *p* = *rep-list* (*atomize-poly* (*idx-pm-of-pm fs r0*))
by (*simp add*: *rep-list-def vectorize-atomize-poly pm-of-idx-pm-of-pm*[*OF r0*] *p*)
next
show *atomize-poly* (*idx-pm-of-pm fs r0*) ∈ *sig-inv-set*
by (*rule sig-inv-setI*, *simp add*: *vectorize-atomize-poly keys-idx-pm-of-pm-subset*)
qed
qed

lemma *keys-rep-list-subset*:
assumes *t* ∈ *keys* (*rep-list r*)
obtains *v s* **where** *v* ∈ *keys r* **and** *s* ∈ *Keys* (*set fs*) **and** *t* = *pp-of-term v* + *s*
proof −
from *assms* **obtain** *v0 s* **where** *v0*: *v0* ∈ *Keys* (*Poly-Mapping.range* (*pm-of-idx-pm fs* (*vectorize-poly r*)))
and *s*: *s* ∈ *Keys* (*keys* (*pm-of-idx-pm fs* (*vectorize-poly r*))) **and** *t*: *t* = *v0* + *s*
unfolding *rep-list-def* **by** (*rule punit.keys-rep-subset*[*simplified*])
note *s*
also from *keys-pm-of-idx-pm-subset* **have** *Keys* (*keys* (*pm-of-idx-pm fs* (*vectorize-poly r*))) ⊆ *Keys* (*set fs*)
by (*rule Keys-mono*)
finally have *s* ∈ *Keys* (*set fs*) **.**
note *v0*
also from *range-pm-of-idx-pm-subset′*
have *Keys* (*Poly-Mapping.range* (*pm-of-idx-pm fs* (*vectorize-poly r*))) ⊆ *Keys* (*Poly-Mapping.range* (*vectorize-poly r*))
by (*rule Keys-mono*)
also have ... = *pp-of-term* ‘ *keys r* **by** (*fact Keys-range-vectorize-poly*)
finally obtain *v* **where** *v* ∈ *keys r* **and** *v0* = *pp-of-term v* **..**
from *this*(*2*) **have** *t* = *pp-of-term v* + *s* **by** (*simp only*: *t*)
with ‹*v* ∈ *keys r*› ‹*s* ∈ *Keys* (*set fs*)› **show** *?thesis* **..**
qed

lemma *dgrad-p-set-le-rep-list*:
assumes *dickson-grading d* **and** *dgrad-set-le d* (*pp-of-term* ‘ *keys r*) (*Keys* (*set fs*))
shows *punit.dgrad-p-set-le d* {*rep-list r*} (*set fs*)
proof (*simp add*: *punit.dgrad-p-set-le-def Keys-insert*, *rule dgrad-set-leI*)
fix *t*
assume *t* ∈ *keys* (*rep-list r*)
then obtain *v s1* **where** *v* ∈ *keys r* **and** *s1* ∈ *Keys* (*set fs*) **and** *t*: *t* = *pp-of-term v* + *s1*
by (*rule keys-rep-list-subset*)
from *this*(*1*) **have** *pp-of-term v* ∈ *pp-of-term* ‘ *keys r* **by** *fastforce*
with *assms*(*2*) **obtain** *s2* **where** *s2* ∈ *Keys* (*set fs*) **and** *d* (*pp-of-term v*) ≤ *d s2*

by (*rule dgrad-set-leE*)
from *assms*(*1*) **have** *d t* = *ord-class.max* (*d* (*pp-of-term v*)) (*d s1*) **unfolding** *t*
by (*rule dickson-gradingD1*)
hence *d t* = *d* (*pp-of-term v*) ∨ *d t* = *d s1* **by** (*simp add*: *ord-class.max-def*)
thus ∃ *s*∈*Keys* (*set fs*). *d t* ≤ *d s*
proof
assume *d t* = *d* (*pp-of-term v*)
with ‹*d* (*pp-of-term v*) ≤ *d s2*› **have** *d t* ≤ *d s2* **by** *simp*
with ‹*s2* ∈ *Keys* (*set fs*)› **show** *?thesis* **..**
next
assume *d t* = *d s1*
hence *d t* ≤ *d s1* **by** *simp*
with ‹*s1* ∈ *Keys* (*set fs*)› **show** *?thesis* **..**
qed
qed

corollary *dgrad-p-set-le-rep-list-image*:
assumes *dickson-grading d* **and** *dgrad-set-le d* (*pp-of-term* ‘ *Keys F*) (*Keys* (*set fs*))
shows *punit.dgrad-p-set-le d* (*rep-list* ‘ *F*) (*set fs*)
proof (*rule punit.dgrad-p-set-leI*, *elim imageE*, *simp*)
fix *f*
assume *f* ∈ *F*
have *pp-of-term* ‘ *keys f* ⊆ *pp-of-term* ‘ *Keys F* **by** (*rule image-mono*, *rule keys-subset-Keys*, *fact*)
hence *dgrad-set-le d* (*pp-of-term* ‘ *keys f*) (*pp-of-term* ‘ *Keys F*) **by** (*rule dgrad-set-le-subset*)
hence *dgrad-set-le d* (*pp-of-term* ‘ *keys f*) (*Keys* (*set fs*)) **using** *assms*(*2*) **by** (*rule dgrad-set-le-trans*)
with *assms*(*1*) **show** *punit.dgrad-p-set-le d* {*rep-list f*} (*set fs*) **by** (*rule dgrad-p-set-le-rep-list*)
qed
term *Max*

definition *dgrad-max* :: (*'a* ⇒ *nat*) ⇒ *nat*
where *dgrad-max d* = (*Max* (*d* ‘ (*insert 0* (*Keys* (*set fs*)))))

abbreviation *dgrad-max-set d* ≡ *dgrad-p-set d* (*dgrad-max d*)
abbreviation *punit-dgrad-max-set d* ≡ *punit.dgrad-p-set d* (*dgrad-max d*)

lemma *dgrad-max-0*: *d 0* ≤ *dgrad-max d*
proof −
from *finite-Keys* **have** *finite* (*d* ‘ *insert 0* (*Keys* (*set fs*))) **by** *auto*
moreover have *d 0* ∈ *d* ‘ *insert 0* (*Keys* (*set fs*)) **by** *blast*
ultimately show *?thesis* **unfolding** *dgrad-max-def* **by** (*rule Max-ge*)
qed

lemma *dgrad-max-1*: *set fs* ⊆ *punit-dgrad-max-set d*
proof (*cases Keys* (*set fs*) = {})
case *True*
show *?thesis*

proof (*rule*, *rule punit.dgrad-p-setI*[*simplified*])
fix *f v*
assume *f* ∈ *set fs* **and** *v* ∈ *keys f*
with *True* **show** *d v* ≤ *dgrad-max d* **by** (*auto simp*: *Keys-def*)
qed
next
case *False*
show *?thesis*
proof (*rule subset-trans*)
from *finite-set* **show** *set fs* ⊆ *punit.dgrad-p-set d* (*Max* (*d* ‘ (*Keys* (*set fs*))))
by (*rule punit.dgrad-p-set-exhaust-expl*[*simplified*])
next
from *finite-set* **have** *finite* (*Keys* (*set fs*)) **by** (*rule finite-Keys*)
hence *finite* (*d* ‘ *Keys* (*set fs*)) **by** (*rule finite-imageI*)
moreover from *False* **have** *2*: *d* ‘ *Keys* (*set fs*) ≠ {} **by** *simp*
ultimately have *dgrad-max d* = *ord-class.max* (*d 0*) (*Max* (*d* ‘ *Keys* (*set fs*)))
by (*simp add*: *dgrad-max-def*)
hence *Max* (*d* ‘ (*Keys* (*set fs*))) ≤ *dgrad-max d* **by** *simp*
thus *punit.dgrad-p-set d* (*Max* (*d* ‘ (*Keys* (*set fs*)))) ⊆ *punit-dgrad-max-set d*
by (*rule punit.dgrad-p-set-subset*)
qed
qed

lemma *dgrad-max-2*:
assumes *dickson-grading d* **and** *r* ∈ *dgrad-max-set d*
shows *rep-list r* ∈ *punit-dgrad-max-set d*
proof (*rule punit.dgrad-p-setI*[*simplified*])
fix *t*
assume *t* ∈ *keys* (*rep-list r*)
then obtain *v s* **where** *v* ∈ *keys r* **and** *s* ∈ *Keys* (*set fs*) **and** *t*: *t* = *pp-of-term v* + *s*
by (*rule keys-rep-list-subset*)
from *assms*(*2*) ‹*v* ∈ *keys r*› **have** *d* (*pp-of-term v*) ≤ *dgrad-max d* **by** (*rule dgrad-p-setD*)
moreover have *d s* ≤ *dgrad-max d* **by** (*simp add*: ‹*s* ∈ *Keys* (*set fs*)› *dgrad-max-def finite-Keys*)
ultimately show *d t* ≤ *dgrad-max d* **by** (*simp add*: *t dickson-gradingD1*[*OF assms*(*1*)])
qed

corollary *dgrad-max-3*:
assumes *dickson-grading d* **and** *F* ⊆ *dgrad-max-set d*
shows *rep-list* ‘ *F* ⊆ *punit-dgrad-max-set d*
proof (*rule*, *elim imageE*, *simp*)
fix *f*
assume *f* ∈ *F*
hence *f* ∈ *dgrad-p-set d* (*dgrad-max d*) **using** *assms*(*2*) **..**
with *assms*(*1*) **show** *rep-list f* ∈ *punit.dgrad-p-set d* (*dgrad-max d*) **by** (*rule dgrad-max-2*)

qed

lemma *punit-dgrad-max-set-subset-dgrad-p-set*:
  assumes *dickson-grading d* **and** *set fs* ⊆ *punit.dgrad-p-set d m* **and** ¬ *set fs* ⊆ {*0*}
  shows *punit-dgrad-max-set d* ⊆ *punit.dgrad-p-set d m*
proof (*rule punit.dgrad-p-set-subset*)
  show *dgrad-max d* ≤ *m* **unfolding** *dgrad-max-def*
  proof (*rule Max.boundedI*)
    show *finite* (*d* ' *insert 0* (*Keys* (*set fs*))) **by** (*simp add*: *finite-Keys*)
  next
    show *d* ' *insert 0* (*Keys* (*set fs*)) ≠ {} **by** *simp*
  next
    fix *a*
    assume *a* ∈ *d* ' *insert 0* (*Keys* (*set fs*))
    then obtain *t* **where** *t* ∈ *insert 0* (*Keys* (*set fs*)) **and** *a* = *d t* **..**
    from *this*(*1*) **show** *a* ≤ *m* **unfolding** ‹*a* = *d t*›
    proof
      assume *t* = *0*
      from *assms*(*3*) **obtain** *f* **where** *f* ∈ *set fs* **and** *f* ≠ *0* **by** *auto*
      from *this*(*1*) *assms*(*2*) **have** *f* ∈ *punit.dgrad-p-set d m* **..**
      from ‹*f* ≠ *0*› **have** *keys f* ≠ {} **by** *simp*
      then obtain *s* **where** *s* ∈ *keys f* **by** *blast*
      have *d s* = *d* (*t* + *s*) **by** (*simp add*: ‹*t* = *0*›)
        also from *assms*(*1*) **have** ... = *ord-class.max* (*d t*) (*d s*) **by** (*rule dickson-gradingD1*)
      finally have *d t* ≤ *d s* **by** (*simp add*: *max-def*)
      also from ‹*f* ∈ *punit.dgrad-p-set d m*› ‹*s* ∈ *keys f*› **have** ... ≤ *m*
        by (*rule punit.dgrad-p-setD*[*simplified*])
      finally show *d t* ≤ *m* **.**
    next
      assume *t* ∈ *Keys* (*set fs*)
      then obtain *f* **where** *f* ∈ *set fs* **and** *t* ∈ *keys f* **by** (*rule in-KeysE*)
      from *this*(*1*) *assms*(*2*) **have** *f* ∈ *punit.dgrad-p-set d m* **..**
      thus *d t* ≤ *m* **using** ‹*t* ∈ *keys f*› **by** (*rule punit.dgrad-p-setD*[*simplified*])
    qed
  qed
qed

definition *dgrad-sig-set'* :: *nat* ⇒ (*'a* ⇒ *nat*) ⇒ (*'t* $\Rightarrow_0$ *'b*) *set*
  where *dgrad-sig-set' j d* = *dgrad-max-set d* ∩ *sig-inv-set' j*

abbreviation *dgrad-sig-set* ≡ *dgrad-sig-set'* (*length fs*)

lemma *dgrad-sig-set-set-mono*: *i* ≤ *j* ⟹ *dgrad-sig-set' i d* ⊆ *dgrad-sig-set' j d*
  by (*auto simp*: *dgrad-sig-set'-def dest*: *sig-inv-set-mono*)

lemma *dgrad-sig-set-closed-uminus*: *r* ∈ *dgrad-sig-set' j d* ⟹ − *r* ∈ *dgrad-sig-set' j d*

unfolding *dgrad-sig-set′-def* **by** (*auto intro*: *dgrad-p-set-closed-uminus sig-inv-set-closed-uminus*)

lemma *dgrad-sig-set-closed-plus*:
$r \in$ *dgrad-sig-set′ j d* $\Longrightarrow$ $s \in$ *dgrad-sig-set′ j d* $\Longrightarrow$ $r + s \in$ *dgrad-sig-set′ j d*
unfolding *dgrad-sig-set′-def* **by** (*auto intro*: *dgrad-p-set-closed-plus sig-inv-set-closed-plus*)

lemma *dgrad-sig-set-closed-minus*:
$r \in$ *dgrad-sig-set′ j d* $\Longrightarrow$ $s \in$ *dgrad-sig-set′ j d* $\Longrightarrow$ $r - s \in$ *dgrad-sig-set′ j d*
unfolding *dgrad-sig-set′-def* **by** (*auto intro*: *dgrad-p-set-closed-minus sig-inv-set-closed-minus*)

lemma *dgrad-sig-set-closed-monom-mult*:
assumes *dickson-grading d* **and** $d\ t \leq$ *dgrad-max d*
shows $p \in$ *dgrad-sig-set′ j d* $\Longrightarrow$ *monom-mult c t p* $\in$ *dgrad-sig-set′ j d*
unfolding *dgrad-sig-set′-def* **by** (*auto intro*: *assms dgrad-p-set-closed-monom-mult*
sig-inv-set-closed-monom-mult)

lemma *dgrad-sig-set-closed-monom-mult-zero*:
$p \in$ *dgrad-sig-set′ j d* $\Longrightarrow$ *monom-mult c 0 p* $\in$ *dgrad-sig-set′ j d*
unfolding *dgrad-sig-set′-def* **by** (*auto intro*: *dgrad-p-set-closed-monom-mult-zero*
sig-inv-set-closed-monom-mult)

lemma *dgrad-sig-set-closed-mult-scalar*:
dickson-grading d $\Longrightarrow$ $p \in$ *punit-dgrad-max-set d* $\Longrightarrow$ $r \in$ *dgrad-sig-set′ j d* $\Longrightarrow$
$p \odot r \in$ *dgrad-sig-set′ j d*
unfolding *dgrad-sig-set′-def* **by** (*auto intro*: *dgrad-p-set-closed-mult-scalar sig-inv-set-closed-mult-scalar*)

lemma *dgrad-sig-set-closed-monomial*:
assumes d (*pp-of-term u*) $\leq$ *dgrad-max d* **and** *component-of-term u* $< j$
shows *monomial c u* $\in$ *dgrad-sig-set′ j d*
proof (*simp add*: *dgrad-sig-set′-def*, *rule*)
show *monomial c u* $\in$ *dgrad-max-set d*
proof (*rule dgrad-p-setI*)
fix v
assume $v \in$ *keys* (*monomial c u*)
also have ... $\subseteq \{u\}$ **by** *simp*
finally show d (*pp-of-term v*) $\leq$ *dgrad-max d* **using** *assms*(*1*) **by** *simp*
qed
next
show *monomial c u* $\in$ *sig-inv-set′ j*
proof (*rule sig-inv-setI′*)
fix v
assume $v \in$ *keys* (*monomial c u*)
also have ... $\subseteq \{u\}$ **by** *simp*
finally show *component-of-term v* $< j$ **using** *assms*(*2*) **by** *simp*
qed
qed

lemma *rep-list-in-ideal-dgrad-sig-set*:
$r \in$ *dgrad-sig-set′ j d* $\Longrightarrow$ *rep-list r* $\in$ *ideal* (*set* (*take j fs*))

by (*auto simp*: *dgrad-sig-set′-def dest*: *rep-list-in-ideal-sig-inv-set*)

lemma *in-idealE-rep-list-dgrad-sig-set-take*:
  assumes *hom-grading d* **and** *p* ∈ *punit-dgrad-max-set d* **and** *p* ∈ *ideal* (*set* (*take j fs*))
  obtains *r* **where** *r* ∈ *dgrad-sig-set d* **and** *r* ∈ *dgrad-sig-set′ j d* **and** *p* = *rep-list r*
proof −
  let *?fs* = *take j fs*
  from *set-take-subset dgrad-max-1* **have** *set ?fs* ⊆ *punit-dgrad-max-set d*
    by (*rule subset-trans*)
  with *assms*(*1*) **obtain** *r0* **where** *r0*: *keys r0* ⊆ *set ?fs*
    and *1*: *Poly-Mapping.range r0* ⊆ *punit-dgrad-max-set d* **and** *p*: *p* = *ideal.rep r0*
    using *assms*(*2*, *3*) **by** (*rule in-idealE-rep-dgrad-p-set*)
  define *q* **where** *q* = *idx-pm-of-pm ?fs r0*
  have *keys q* ⊆ {*0..<length ?fs*} **unfolding** *q-def* **by** (*rule keys-idx-pm-of-pm-subset*)
  also have ... ⊆ {*0..<j*} **by** *fastforce*
  finally have *keys-q*: *keys q* ⊆ {*0..<j*} **.**
  have ∗: *atomize-poly q* ∈ *dgrad-max-set d*
  proof
    fix *v*
    assume *v* ∈ *keys* (*atomize-poly q*)
    then obtain *i* **where** *i*: *i* ∈ *keys q*
      and *v-in*: *v* ∈ (λ*t*. *term-of-pair* (*t*, *i*)) ‘ *keys* (*lookup q i*)
      unfolding *keys-atomize-poly* **..**
    from *i keys-idx-pm-of-pm-subset*[*of ?fs r0*] **have** *i* < *length ?fs* **by** (*auto simp*: *q-def*)
    from *v-in* **obtain** *t* **where** *t* ∈ *keys* (*lookup q i*) **and** *v*: *v* = *term-of-pair* (*t*, *i*) **..**
    from *this*(*1*) ‹*i* < *length ?fs*› **have** *t*: *t* ∈ *keys* (*lookup r0* (*?fs* ! *i*))
      by (*simp add*: *lookup-idx-pm-of-pm q-def*)
    hence *lookup r0* (*?fs* ! *i*) ≠ *0* **by** *fastforce*
    hence *lookup r0* (*?fs* ! *i*) ∈ *Poly-Mapping.range r0* **by** (*simp add*: *in-keys-iff*)
    hence *lookup r0* (*?fs* ! *i*) ∈ *punit-dgrad-max-set d* **using** *1* **..**
    hence *d t* ≤ *dgrad-max d* **using** *t* **by** (*rule punit.dgrad-p-setD*[*simplified*])
    thus *d* (*pp-of-term v*) ≤ *dgrad-max d* **by** (*simp add*: *v pp-of-term-of-pair*)
  qed
  show *?thesis*
  proof
    have *atomize-poly q* ∈ *sig-inv-set′ j*
      by (*rule sig-inv-setI*, *simp add*: *vectorize-atomize-poly keys-q*)
    with ∗ **show** *atomize-poly q* ∈ *dgrad-sig-set′ j d* **unfolding** *dgrad-sig-set′-def* **..**
  next
    from ‹*keys q* ⊆ {*0..<length ?fs*}› **have** *keys-q′*: *keys q* ⊆ {*0..<length fs*} **by** *auto*
    have *atomize-poly q* ∈ *sig-inv-set*
      by (*rule sig-inv-setI*, *simp add*: *vectorize-atomize-poly keys-q′*)

with $*$ **show** *atomize-poly q* $\in$ *dgrad-sig-set d* **unfolding** *dgrad-sig-set′-def* **..**
next
from *keys-q* **have** *pm-of-idx-pm fs q* = *pm-of-idx-pm ?fs q* **by** (*simp only*: *pm-of-idx-pm-take*)
thus *p* = *rep-list* (*atomize-poly q*)
by (*simp add*: *rep-list-def vectorize-atomize-poly pm-of-idx-pm-of-pm*[*OF r0*] *p q-def*)
qed
qed

corollary *in-idealE-rep-list-dgrad-sig-set*:
assumes *hom-grading d* **and** *p* $\in$ *punit-dgrad-max-set d* **and** *p* $\in$ *ideal* (*set fs*)
obtains *r* **where** *r* $\in$ *dgrad-sig-set d* **and** *p* = *rep-list r*
proof −
from *assms*(*3*) **have** *p* $\in$ *ideal* (*set* (*take* (*length fs*) *fs*)) **by** *simp*
with *assms*(*1*, *2*) **obtain** *r* **where** *r* $\in$ *dgrad-sig-set d* **and** *p* = *rep-list r*
by (*rule in-idealE-rep-list-dgrad-sig-set-take*)
thus *?thesis* **..**
qed

lemma *dgrad-sig-setD-lp*:
assumes *p* $\in$ *dgrad-sig-set′ j d*
shows *d* (*lp p*) $\le$ *dgrad-max d*
proof (*cases p* = *0*)
case *True*
show *?thesis* **by** (*simp add*: *True min-term-def pp-of-term-of-pair dgrad-max-0*)
next
case *False*
from *assms* **have** *p* $\in$ *dgrad-max-set d* **by** (*simp add*: *dgrad-sig-set′-def*)
thus *?thesis* **using** *False* **by** (*rule dgrad-p-setD-lp*)
qed

lemma *dgrad-sig-setD-lt*:
assumes *p* $\in$ *dgrad-sig-set′ j d* **and** *p* $\neq$ *0*
shows *component-of-term* (*lt p*) < *j*
proof −
from *assms* **have** *p* $\in$ *sig-inv-set′ j* **by** (*simp add*: *dgrad-sig-set′-def*)
thus *?thesis* **using** *assms*(*2*) **by** (*rule sig-inv-setD-lt*)
qed

lemma *dgrad-sig-setD-rep-list-lt*:
assumes *dickson-grading d* **and** *p* $\in$ *dgrad-sig-set′ j d*
shows *d* (*punit.lt* (*rep-list p*)) $\le$ *dgrad-max d*
proof (*cases rep-list p* = *0*)
case *True*
show *?thesis* **by** (*simp add*: *True dgrad-max-0*)
next
case *False*
from *assms*(*2*) **have** *p* $\in$ *dgrad-max-set d* **by** (*simp add*: *dgrad-sig-set′-def*)

with *assms*(*1*) **have** *rep-list p* ∈ *punit-dgrad-max-set d* **by** (*rule dgrad-max-2*)
thus *?thesis* **using** *False* **by** (*rule punit.dgrad-p-setD-lp*[*simplified*])
qed

definition *spp-of* :: $('t \Rightarrow_0 'b) \Rightarrow ('t \times ('a \Rightarrow_0 'b))$
where *spp-of r* = (*lt r*, *rep-list r*)

"spp" stands for "sig-poly-pair".

lemma *fst-spp-of*: *fst* (*spp-of r*) = *lt r*
by (*simp add*: *spp-of-def*)

lemma *snd-spp-of*: *snd* (*spp-of r*) = *rep-list r*
by (*simp add*: *spp-of-def*)

4.2.1 Signature Reduction

lemma *term-is-le-rel-canc-left*:
assumes *ord-term-lin.is-le-rel rel*
shows *rel* (*t* ⊕ *u*) (*t* ⊕ *v*) ⟷ *rel u v*
using *assms*
by (*rule ord-term-lin.is-le-relE*,
auto simp: *splus-left-canc dest*: *ord-term-canc ord-term-strict-canc splus-mono splus-mono-strict*)

lemma *term-is-le-rel-minus*:
assumes *ord-term-lin.is-le-rel rel* **and** *s adds t*
shows *rel* ((*t* − *s*) ⊕ *u*) *v* ⟷ *rel* (*t* ⊕ *u*) (*s* ⊕ *v*)
proof −
from *assms*(*2*) **have** *eq*: *s* + (*t* − *s*) = *t* **unfolding** *add.commute*[*of s*] **by** (*rule adds-minus*)
from *assms*(*1*) **have** *rel* ((*t* − *s*) ⊕ *u*) *v* = *rel* (*s* ⊕ ((*t* − *s*) ⊕ *u*)) (*s* ⊕ *v*)
by (*simp only*: *term-is-le-rel-canc-left*)
also have ... = *rel* (*t* ⊕ *u*) (*s* ⊕ *v*) **by** (*simp only*: *splus-assoc*[*symmetric*] *eq*)
finally show *?thesis* **.**
qed

lemma *term-is-le-rel-minus-minus*:
assumes *ord-term-lin.is-le-rel rel* **and** *a adds t* **and** *b adds t*
shows *rel* ((*t* − *a*) ⊕ *u*) ((*t* − *b*) ⊕ *v*) ⟷ *rel* (*b* ⊕ *u*) (*a* ⊕ *v*)
proof −
from *assms*(*2*) **have** *eq1*: *a* + (*t* − *a*) = *t* **unfolding** *add.commute*[*of a*] **by** (*rule adds-minus*)
from *assms*(*3*) **have** *eq2*: *b* + (*t* − *b*) = *t* **unfolding** *add.commute*[*of b*] **by** (*rule adds-minus*)
from *assms*(*1*) **have** *rel* ((*t* − *a*) ⊕ *u*) ((*t* − *b*) ⊕ *v*) = *rel* ((*a* + *b*) ⊕ ((*t* − *a*) ⊕ *u*)) ((*a* + *b*) ⊕ ((*t* − *b*) ⊕ *v*))
by (*simp only*: *term-is-le-rel-canc-left*)
also have ... = *rel* ((*t* + *b*) ⊕ *u*) ((*t* + *a*) ⊕ *v*) **unfolding** *splus-assoc*[*symmetric*]
by (*metis* (*no-types*, *lifting*) *add.assoc add.commute eq1 eq2*)

also from *assms*(*1*) **have** ... = *rel* (*b* ⊕ *u*) (*a* ⊕ *v*) **by** (*simp only*: *splus-assoc term-is-le-rel-canc-left*)
finally show *?thesis* **.**
qed

lemma *pp-is-le-rel-canc-right*:
assumes *ordered-powerprod-lin.is-le-rel rel*
shows *rel* (*s* + *u*) (*t* + *u*) ⟷ *rel s t*
using *assms*
by (*rule ordered-powerprod-lin.is-le-relE*, *auto dest*: *ord-canc ord-strict-canc plus-monotone plus-monotone-strict*)

lemma *pp-is-le-rel-canc-left*: *ordered-powerprod-lin.is-le-rel rel* ⟹ *rel* (*t* + *u*) (*t* + *v*) ⟷ *rel u v*
by (*simp add*: *add.commute*[*of t*] *pp-is-le-rel-canc-right*)

definition *sig-red-single* :: ('*t* ⇒ '*t* ⇒ *bool*) ⇒ ('*a* ⇒ '*a* ⇒ *bool*) ⇒ ('*t* $\Rightarrow_0$ '*b*) ⇒ ('*t* $\Rightarrow_0$ '*b*) ⇒ ('*t* $\Rightarrow_0$ '*b*) ⇒ '*a* ⇒ *bool*
where *sig-red-single sing-reg top-tail p q f t* ⟷
(*rep-list f* ≠ *0* ∧ *lookup* (*rep-list p*) (*t* + *punit.lt* (*rep-list f*)) ≠ *0* ∧
q = *p* − *monom-mult* ((*lookup* (*rep-list p*) (*t* + *punit.lt* (*rep-list f*))) / *punit.lc* (*rep-list f*)) *t f* ∧
ord-term-lin.is-le-rel sing-reg ∧ *ordered-powerprod-lin.is-le-rel top-tail* ∧
sing-reg (*t* ⊕ *lt f*) (*lt p*) ∧ *top-tail* (*t* + *punit.lt* (*rep-list f*)) (*punit.lt* (*rep-list p*)))

The first two parameters of *sig-red-single*, *sing-reg* and *top-tail*, specify whether the reduction is a singular/regular/arbitrary top/tail/arbitrary signature-reduction.

- If *sing-reg* is (=), the reduction is singular.
- If *sing-reg* is ($\prec_t$), the reduction is regular.
- If *sing-reg* is ($\preceq_t$), the reduction is an arbitrary signature-reduction.
- If *top-tail* is (=), it is a top reduction.
- If *top-tail* is (≺), it is a tail reduction.
- If *top-tail* is (⪯), the reduction is an arbitrary signature-reduction.

definition *sig-red* :: ('*t* ⇒ '*t* ⇒ *bool*) ⇒ ('*a* ⇒ '*a* ⇒ *bool*) ⇒ ('*t* $\Rightarrow_0$ '*b*) *set* ⇒ ('*t* $\Rightarrow_0$ '*b*) ⇒ ('*t* $\Rightarrow_0$ '*b*) ⇒ *bool*
where *sig-red sing-reg top-tail F p q* ⟷ (∃ *f*∈*F*. ∃ *t*. *sig-red-single sing-reg top-tail p q f t*)

definition *is-sig-red* :: ('*t* ⇒ '*t* ⇒ *bool*) ⇒ ('*a* ⇒ '*a* ⇒ *bool*) ⇒ ('*t* $\Rightarrow_0$ '*b*) *set* ⇒ ('*t* $\Rightarrow_0$ '*b*) ⇒ *bool*

where *is-sig-red sing-reg top-tail F p* $\longleftrightarrow$ ($\exists$ *q. sig-red sing-reg top-tail F p q*)

lemma *sig-red-singleI*:
assumes *rep-list f* $\neq$ *0* **and** *t* + *punit.lt* (*rep-list f*) $\in$ *keys* (*rep-list p*)
and *q* = *p* − *monom-mult* ((*lookup* (*rep-list p*) (*t* + *punit.lt* (*rep-list f*))) / *punit.lc* (*rep-list f*)) *t f*
and *ord-term-lin.is-le-rel sing-reg* **and** *ordered-powerprod-lin.is-le-rel top-tail*
and *sing-reg* (*t* $\oplus$ *lt f*) (*lt p*)
and *top-tail* (*t* + *punit.lt* (*rep-list f*)) (*punit.lt* (*rep-list p*))
shows *sig-red-single sing-reg top-tail p q f t*
unfolding *sig-red-single-def* **using** *assms* **by** *blast*

lemma *sig-red-singleD1*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows *rep-list f* $\neq$ *0*
using *assms* **unfolding** *sig-red-single-def* **by** *blast*

lemma *sig-red-singleD2*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows *t* + *punit.lt* (*rep-list f*) $\in$ *keys* (*rep-list p*)
using *assms* **unfolding** *sig-red-single-def* **by** (*simp add*: *in-keys-iff*)

lemma *sig-red-singleD3*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows *q* = *p* − *monom-mult* ((*lookup* (*rep-list p*) (*t* + *punit.lt* (*rep-list f*))) / *punit.lc* (*rep-list f*)) *t f*
using *assms* **unfolding** *sig-red-single-def* **by** *blast*

lemma *sig-red-singleD4*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows *ord-term-lin.is-le-rel sing-reg*
using *assms* **unfolding** *sig-red-single-def* **by** *blast*

lemma *sig-red-singleD5*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows *ordered-powerprod-lin.is-le-rel top-tail*
using *assms* **unfolding** *sig-red-single-def* **by** *blast*

lemma *sig-red-singleD6*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows *sing-reg* (*t* $\oplus$ *lt f*) (*lt p*)
using *assms* **unfolding** *sig-red-single-def* **by** *blast*

lemma *sig-red-singleD7*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows *top-tail* (*t* + *punit.lt* (*rep-list f*)) (*punit.lt* (*rep-list p*))
using *assms* **unfolding** *sig-red-single-def* **by** *blast*

lemma *sig-red-singleD8*:

assumes *sig-red-single sing-reg top-tail p q f t*
shows $t \oplus lt\ f \preceq_t lt\ p$
proof −
from *assms* **have** *ord-term-lin.is-le-rel sing-reg* **and** *sing-reg* $(t \oplus lt\ f)$ $(lt\ p)$
by (*rule sig-red-singleD4*, *rule sig-red-singleD6*)
thus *?thesis* **by** (*rule ord-term-lin.is-le-rel-le*)
qed

lemma *sig-red-singleD9*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows $t + punit.lt\ (rep\text{-}list\ f) \preceq punit.lt\ (rep\text{-}list\ p)$
proof −
from *assms* **have** *ordered-powerprod-lin.is-le-rel top-tail*
and *top-tail* $(t + punit.lt\ (rep\text{-}list\ f))\ (punit.lt\ (rep\text{-}list\ p))$
by (*rule sig-red-singleD5*, *rule sig-red-singleD7*)
thus *?thesis* **by** (*rule ordered-powerprod-lin.is-le-rel-le*)
qed

lemmas *sig-red-singleD* = *sig-red-singleD1 sig-red-singleD2 sig-red-singleD3 sig-red-singleD4 sig-red-singleD5 sig-red-singleD6 sig-red-singleD7 sig-red-singleD8 sig-red-singleD9*

lemma *sig-red-single-red-single*:
sig-red-single sing-reg top-tail p q f t $\Longrightarrow$ *punit.red-single* (*rep-list p*) (*rep-list q*) (*rep-list f*) *t*
by (*simp add*: *sig-red-single-def punit.red-single-def rep-list-minus rep-list-monom-mult*)

lemma *sig-red-single-regular-lt*:
assumes *sig-red-single* $(\prec_t)$ *top-tail p q f t*
shows $lt\ q = lt\ p$
proof −
let $?f = monom\text{-}mult\ ((lookup\ (rep\text{-}list\ p)\ (t + punit.lt\ (rep\text{-}list\ f)))\ /\ punit.lc\ (rep\text{-}list\ f))\ t\ f$
from *assms* **have** *lt*: $t \oplus lt\ f \prec_t lt\ p$ **and** *q*: $q = p - ?f$
by (*rule sig-red-singleD6*, *rule sig-red-singleD3*)
from *lt-monom-mult-le lt* **have** $lt\ ?f \prec_t lt\ p$ **by** (*rule ord-term-lin.order.strict-trans1*)
thus *?thesis* **unfolding** *q* **by** (*rule lt-minus-eqI-2*)
qed

lemma *sig-red-single-regular-lc*:
assumes *sig-red-single* $(\prec_t)$ *top-tail p q f t*
shows $lc\ q = lc\ p$
proof −
from *assms* **have** $lt\ q = lt\ p$ **by** (*rule sig-red-single-regular-lt*)
from *assms* **have** *lt*: $t \oplus lt\ f \prec_t lt\ p$
and *q*: $q = p - monom\text{-}mult\ ((lookup\ (rep\text{-}list\ p)\ (t + punit.lt\ (rep\text{-}list\ f)))\ /\ punit.lc\ (rep\text{-}list\ f))\ t\ f$
(**is** $- = - - ?f$) **by** (*rule sig-red-singleD6*, *rule sig-red-singleD3*)
from *lt-monom-mult-le lt* **have** $lt\ ?f \prec_t lt\ p$ **by** (*rule ord-term-lin.order.strict-trans1*)

hence *lookup ?f* (*lt p*) = *0* **using** *lt-max ord-term-lin.leD* **by** *blast*
thus *?thesis* **unfolding** *lc-def* ‹*lt q* = *lt p*› **by** (*simp add*: *q lookup-minus*)
qed

lemma *sig-red-single-lt*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows *lt q* $\preceq_t$ *lt p*
proof −
from *assms* **have** *lt*: *t* ⊕ *lt f* $\preceq_t$ *lt p*
and *q* = *p* − *monom-mult* ((*lookup* (*rep-list p*) (*t* + *punit.lt* (*rep-list f*))) / *punit.lc* (*rep-list f*)) *t f*
by (*rule sig-red-singleD8*, *rule sig-red-singleD3*)
from *this*(*2*) **have** *q*: *q* = *p* + *monom-mult* (− (*lookup* (*rep-list p*) (*t* + *punit.lt* (*rep-list f*))) / *punit.lc* (*rep-list f*)) *t f*
(**is** - = - + *?f*) **by** (*simp add*: *monom-mult-uminus-left*)
from *lt-monom-mult-le lt* **have** *1*: *lt ?f* $\preceq_t$ *lt p* **by** (*rule ord-term-lin.order.trans*)
have *lt q* $\preceq_t$ *ord-term-lin.max* (*lt p*) (*lt ?f*) **unfolding** *q* **by** (*fact lt-plus-le-max*)
also from *1* **have** *ord-term-lin.max* (*lt p*) (*lt ?f*) = *lt p* **by** (*rule ord-term-lin.max.absorb1*)
finally show *?thesis* **.**
qed

lemma *sig-red-single-lt-rep-list*:
assumes *sig-red-single sing-reg top-tail p q f t*
shows *punit.lt* (*rep-list q*) ⪯ *punit.lt* (*rep-list p*)
proof −
from *assms* **have** *punit.red-single* (*rep-list p*) (*rep-list q*) (*rep-list f*) *t*
by (*rule sig-red-single-red-single*)
hence *punit.ord-strict-p* (*rep-list q*) (*rep-list p*) **by** (*rule punit.red-single-ord*)
hence *punit.ord-p* (*rep-list q*) (*rep-list p*) **by** *simp*
thus *?thesis* **by** (*rule punit.ord-p-lt*)
qed

lemma *sig-red-single-tail-lt-in-keys-rep-list*:
assumes *sig-red-single sing-reg* (≺) *p q f t*
shows *punit.lt* (*rep-list p*) ∈ *keys* (*rep-list q*)
proof −
from *assms* **have** *q* = *p* − *monom-mult* ((*lookup* (*rep-list p*) (*t* + *punit.lt* (*rep-list f*))) / *punit.lc* (*rep-list f*)) *t f*
by (*rule sig-red-singleD3*)
hence *q*: *q* = *p* + *monom-mult* (− (*lookup* (*rep-list p*) (*t* + *punit.lt* (*rep-list f*))) / *punit.lc* (*rep-list f*)) *t f*
by (*simp add*: *monom-mult-uminus-left*)
show *?thesis* **unfolding** *q rep-list-plus rep-list-monom-mult*
proof (*rule in-keys-plusI1*)
from *assms* **have** *t* + *punit.lt* (*rep-list f*) ∈ *keys* (*rep-list p*) **by** (*rule sig-red-singleD2*)
hence *rep-list p* ≠ *0* **by** *auto*
thus *punit.lt* (*rep-list p*) ∈ *keys* (*rep-list p*) **by** (*rule punit.lt-in-keys*)
next
show *punit.lt* (*rep-list p*) ∉

keys (*punit.monom-mult* (− *lookup* (*rep-list p*) (*t* + *punit.lt* (*rep-list f*)) / *punit.lc* (*rep-list f*)) *t* (*rep-list f*))
(**is** - ∉ *keys ?f*)
proof
assume *punit.lt* (*rep-list p*) ∈ *keys ?f*
hence *punit.lt* (*rep-list p*) ⪯ *punit.lt ?f* **by** (*rule punit.lt-max-keys*)
also have ... ⪯ *t* + *punit.lt* (*rep-list f*) **by** (*fact punit.lt-monom-mult-le*[*simplified*])
also from *assms* **have** ... ≺ *punit.lt* (*rep-list p*) **by** (*rule sig-red-singleD7*)
finally show *False* **by** *simp*
qed
qed
qed

corollary *sig-red-single-tail-lt-rep-list*:
assumes *sig-red-single sing-reg* (≺) *p q f t*
shows *punit.lt* (*rep-list q*) = *punit.lt* (*rep-list p*)
proof (*rule ordered-powerprod-lin.order-antisym*)
from *assms* **show** *punit.lt* (*rep-list q*) ⪯ *punit.lt* (*rep-list p*) **by** (*rule sig-red-single-lt-rep-list*)
next
from *assms* **have** *punit.lt* (*rep-list p*) ∈ *keys* (*rep-list q*) **by** (*rule sig-red-single-tail-lt-in-keys-rep-list*)
thus *punit.lt* (*rep-list p*) ⪯ *punit.lt* (*rep-list q*) **by** (*rule punit.lt-max-keys*)
qed

lemma *sig-red-single-tail-lc-rep-list*:
assumes *sig-red-single sing-reg* (≺) *p q f t*
shows *punit.lc* (*rep-list q*) = *punit.lc* (*rep-list p*)
proof −
from *assms* **have** ∗: *punit.lt* (*rep-list q*) = *punit.lt* (*rep-list p*)
by (*rule sig-red-single-tail-lt-rep-list*)
from *assms* **have** *lt*: *t* + *punit.lt* (*rep-list f*) ≺ *punit.lt* (*rep-list p*)
and *q*: *q* = *p* − *monom-mult* ((*lookup* (*rep-list p*) (*t* + *punit.lt* (*rep-list f*))) / *punit.lc* (*rep-list f*)) *t f*
(**is** - = - − *?f*) **by** (*rule sig-red-singleD7*, *rule sig-red-singleD3*)
from *punit.lt-monom-mult-le*[*simplified*] *lt* **have** *punit.lt* (*rep-list ?f*) ≺ *punit.lt* (*rep-list p*)
unfolding *rep-list-monom-mult* **by** (*rule ordered-powerprod-lin.order.strict-trans1*)
hence *lookup* (*rep-list ?f*) (*punit.lt* (*rep-list p*)) = *0*
using *punit.lt-max ordered-powerprod-lin.leD* **by** *blast*
thus *?thesis* **unfolding** *punit.lc-def* ∗ **by** (*simp add*: *q lookup-minus rep-list-minus punit.lc-def*)
qed

lemma *sig-red-single-top-lt-rep-list*:
assumes *sig-red-single sing-reg* (=) *p q f t* **and** *rep-list q* ≠ *0*
shows *punit.lt* (*rep-list q*) ≺ *punit.lt* (*rep-list p*)
proof −
from *assms*(*1*) **have** *rep-list f* ≠ *0* **and** *in-keys*: *t* + *punit.lt* (*rep-list f*) ∈ *keys* (*rep-list p*)
and *lt*: *t* + *punit.lt* (*rep-list f*) = *punit.lt* (*rep-list p*)

and *q* = *p* − *monom-mult* ((*lookup* (*rep-list p*) (*t* + *punit.lt* (*rep-list f*))) / *punit.lc* (*rep-list f*)) *t f*
by (*rule sig-red-singleD*)+
from *this*(*4*) **have** *q*: *q* = *p* + *monom-mult* (− (*lookup* (*rep-list p*) (*t* + *punit.lt* (*rep-list f*))) / *punit.lc* (*rep-list f*)) *t f*
(**is** - = - + *monom-mult ?c* - -) **by** (*simp add*: *monom-mult-uminus-left*)
from ‹*rep-list f* ≠ *0*› **have** *punit.lc* (*rep-list f*) ≠ *0* **by** (*rule punit.lc-not-0*)
from *assms*(*2*) **have** ∗: *rep-list p* + *punit.monom-mult ?c t* (*rep-list f*) ≠ *0*
by (*simp add*: *q rep-list-plus rep-list-monom-mult*)
from *in-keys* **have** *lookup* (*rep-list p*) (*t* + *punit.lt* (*rep-list f*)) ≠ *0*
by (*simp add*: *in-keys-iff*)
moreover from ‹*rep-list f* ≠ *0*› **have** *punit.lc* (*rep-list f*) ≠ *0* **by** (*rule punit.lc-not-0*)
ultimately have *?c* ≠ *0* **by** *simp*
hence *punit.lt* (*punit.monom-mult ?c t* (*rep-list f*)) = *t* + *punit.lt* (*rep-list f*)
using ‹*rep-list f* ≠ *0*› **by** (*rule lp-monom-mult*)
hence *punit.lt* (*punit.monom-mult ?c t* (*rep-list f*)) = *punit.lt* (*rep-list p*) **by** (*simp only*: *lt*)
moreover have *punit.lc* (*punit.monom-mult ?c t* (*rep-list f*)) = − *punit.lc* (*rep-list p*)
by (*simp add*: *lt punit.lc-def*[*symmetric*] ‹*punit.lc* (*rep-list f*) ≠ *0*›)
ultimately show *?thesis* **unfolding** *rep-list-plus rep-list-monom-mult q* **by** (*rule punit.lt-plus-lessI*[*OF* ∗])
qed

lemma *sig-red-single-monom-mult*:
assumes *sig-red-single sing-reg top-tail p q f t* **and** *c* ≠ *0*
shows *sig-red-single sing-reg top-tail* (*monom-mult c s p*) (*monom-mult c s q*) *f* (*s* + *t*)
proof −
from *assms*(*1*) **have** *a*: *ord-term-lin.is-le-rel sing-reg* **and** *b*: *ordered-powerprod-lin.is-le-rel top-tail*
by (*rule sig-red-singleD4*, *rule sig-red-singleD5*)
have *eq1*: (*s* + *t*) ⊕ *lt f* = *s* ⊕ (*t* ⊕ *lt f*) **by** (*simp only*: *splus-assoc*)
from *assms*(*1*) **have** *1*: *t* + *punit.lt* (*rep-list f*) ∈ *keys* (*rep-list p*) **by** (*rule sig-red-singleD2*)
hence *rep-list p* ≠ *0* **by** *auto*
hence *p* ≠ *0* **by** (*auto simp*: *rep-list-zero*)
with *assms*(*2*) **have** *eq2*: *lt* (*monom-mult c s p*) = *s* ⊕ *lt p* **by** (*rule lt-monom-mult*)
show *?thesis*
proof (*rule sig-red-singleI*)
from *assms*(*1*) **show** *rep-list f* ≠ *0* **by** (*rule sig-red-singleD1*)
next
show *s* + *t* + *punit.lt* (*rep-list f*) ∈ *keys* (*rep-list* (*monom-mult c s p*))
by (*auto simp*: *rep-list-monom-mult punit.keys-monom-mult*[*OF assms*(*2*)] *ac-simps intro*: *1*)
next
from *assms*(*1*) **have** *q*: *q* = *p* − *monom-mult* ((*lookup* (*rep-list p*) (*t* + *punit.lt* (*rep-list f*))) / *punit.lc* (*rep-list f*)) *t f*
by (*rule sig-red-singleD3*)

```
    show monom-mult c s q =
          monom-mult c s p −
              monom-mult (lookup (rep-list (monom-mult c s p)) (s + t + punit.lt
(rep-list f)) / punit.lc (rep-list f)) (s + t) f
      by (simp add: q monom-mult-dist-right-minus ac-simps rep-list-monom-mult
          punit.lookup-monom-mult-plus[simplified] monom-mult-assoc)
  next
    from assms(1) have sing-reg (t ⊕ lt f) (lt p) by (rule sig-red-singleD6)
    thus sing-reg ((s + t) ⊕ lt f) (lt (monom-mult c s p))
      by (simp only: eq1 eq2 term-is-le-rel-canc-left[OF a])
  next
    from assms(1) have top-tail (t + punit.lt (rep-list f)) (punit.lt (rep-list p))
      by (rule sig-red-singleD7)
    thus top-tail (s + t + punit.lt (rep-list f)) (punit.lt (rep-list (monom-mult c s
p)))
     by (simp add: rep-list-monom-mult punit.lt-monom-mult[OF assms(2) ‹rep-list
p ≠ 0›] add.assoc pp-is-le-rel-canc-left[OF b])
  qed (fact a, fact b)
qed

lemma sig-red-single-sing-reg-cases:
 sig-red-single (≼t) top-tail p q f t = (sig-red-single (=) top-tail p q f t ∨ sig-red-single
(≺t) top-tail p q f t)
  by (auto simp: sig-red-single-def)

corollary sig-red-single-sing-regI:
  assumes sig-red-single sing-reg top-tail p q f t
  shows sig-red-single (≼t) top-tail p q f t
proof −
  from assms have ord-term-lin.is-le-rel sing-reg by (rule sig-red-singleD)
  with assms show ?thesis unfolding ord-term-lin.is-le-rel-def
    by (auto simp: sig-red-single-sing-reg-cases)
qed

lemma sig-red-single-top-tail-cases:
 sig-red-single sing-reg (≼) p q f t = (sig-red-single sing-reg (=) p q f t ∨ sig-red-single
sing-reg (≺) p q f t)
  by (auto simp: sig-red-single-def)

corollary sig-red-single-top-tailI:
  assumes sig-red-single sing-reg top-tail p q f t
  shows sig-red-single sing-reg (≼) p q f t
proof −
 from assms have ordered-powerprod-lin.is-le-rel top-tail by (rule sig-red-singleD)
  with assms show ?thesis unfolding ordered-powerprod-lin.is-le-rel-def
    by (auto simp: sig-red-single-top-tail-cases)
qed

lemma dgrad-max-set-closed-sig-red-single:
```

assumes *dickson-grading d* **and** $p \in$ *dgrad-max-set d* **and** $f \in$ *dgrad-max-set d*
and *sig-red-single sing-red top-tail p q f t*
shows $q \in$ *dgrad-max-set d*
proof −
let *?f* = *monom-mult* (*lookup* (*rep-list p*) (*t* + *punit.lt* (*rep-list f*)) / *punit.lc* (*rep-list f*)) *t f*
from *assms*(*4*) **have** *t*: *t* + *punit.lt* (*rep-list f*) ∈ *keys* (*rep-list p*) **and** *q*: *q* = *p* − *?f*
by (*rule sig-red-singleD2*, *rule sig-red-singleD3*)
from *assms*(*1*, *2*) **have** *rep-list p* ∈ *punit-dgrad-max-set d* **by** (*rule dgrad-max-2*)
show *?thesis* **unfolding** *q* **using** *assms*(*2*)
proof (*rule dgrad-p-set-closed-minus*)
from *assms*(*1*) - *assms*(*3*) **show** *?f* ∈ *dgrad-max-set d*
proof (*rule dgrad-p-set-closed-monom-mult*)
from *assms*(*1*) **have** *d t* ≤ *d* (*t* + *punit.lt* (*rep-list f*)) **by** (*simp add*: *dickson-gradingD1*)
also from ‹*rep-list p* ∈ *punit-dgrad-max-set d*› *t* **have** ... ≤ *dgrad-max d*
by (*rule punit.dgrad-p-setD*[*simplified*])
finally show *d t* ≤ *dgrad-max d* **.**
qed
qed
qed

lemma *sig-inv-set-closed-sig-red-single*:
assumes *p* ∈ *sig-inv-set* **and** *f* ∈ *sig-inv-set* **and** *sig-red-single sing-red top-tail p q f t*
shows *q* ∈ *sig-inv-set*
proof −
let *?f* = *monom-mult* (*lookup* (*rep-list p*) (*t* + *punit.lt* (*rep-list f*)) / *punit.lc* (*rep-list f*)) *t f*
from *assms*(*3*) **have** *t*: *t* + *punit.lt* (*rep-list f*) ∈ *keys* (*rep-list p*) **and** *q*: *q* = *p* − *?f*
by (*rule sig-red-singleD2*, *rule sig-red-singleD3*)
show *?thesis* **unfolding** *q* **using** *assms*(*1*)
proof (*rule sig-inv-set-closed-minus*)
from *assms*(*2*) **show** *?f* ∈ *sig-inv-set* **by** (*rule sig-inv-set-closed-monom-mult*)
qed
qed

corollary *dgrad-sig-set-closed-sig-red-single*:
assumes *dickson-grading d* **and** *p* ∈ *dgrad-sig-set d* **and** *f* ∈ *dgrad-sig-set d*
and *sig-red-single sing-red top-tail p q f t*
shows *q* ∈ *dgrad-sig-set d*
using *assms* **unfolding** *dgrad-sig-set′-def*
by (*auto intro*: *dgrad-max-set-closed-sig-red-single sig-inv-set-closed-sig-red-single*)

lemma *sig-red-regular-lt*: *sig-red* ($\prec_t$) *top-tail F p q* ⟹ *lt q* = *lt p*
by (*auto simp*: *sig-red-def intro*: *sig-red-single-regular-lt*)

lemma *sig-red-regular-lc*: *sig-red* $(\prec_t)$ *top-tail* $F\ p\ q \Longrightarrow$ *lc* $q =$ *lc* p
by (*auto simp*: *sig-red-def intro*: *sig-red-single-regular-lc*)

lemma *sig-red-lt*: *sig-red sing-reg top-tail* $F\ p\ q \Longrightarrow$ *lt* $q \preceq_t$ *lt* p
by (*auto simp*: *sig-red-def intro*: *sig-red-single-lt*)

lemma *sig-red-tail-lt-rep-list*: *sig-red sing-reg* $(\prec)$ $F\ p\ q \Longrightarrow$ *punit.lt* (*rep-list* q) $=$ *punit.lt* (*rep-list* p)
by (*auto simp*: *sig-red-def intro*: *sig-red-single-tail-lt-rep-list*)

lemma *sig-red-tail-lc-rep-list*: *sig-red sing-reg* $(\prec)$ $F\ p\ q \Longrightarrow$ *punit.lc* (*rep-list* q) $=$ *punit.lc* (*rep-list* p)
by (*auto simp*: *sig-red-def intro*: *sig-red-single-tail-lc-rep-list*)

lemma *sig-red-top-lt-rep-list*:
sig-red sing-reg $(=)$ $F\ p\ q \Longrightarrow$ *rep-list* $q \neq 0 \Longrightarrow$ *punit.lt* (*rep-list* q) $\prec$ *punit.lt* (*rep-list* p)
by (*auto simp*: *sig-red-def intro*: *sig-red-single-top-lt-rep-list*)

lemma *sig-red-lt-rep-list*: *sig-red sing-reg top-tail* $F\ p\ q \Longrightarrow$ *punit.lt* (*rep-list* q) $\preceq$ *punit.lt* (*rep-list* p)
by (*auto simp*: *sig-red-def intro*: *sig-red-single-lt-rep-list*)

lemma *sig-red-red*: *sig-red sing-reg top-tail* $F\ p\ q \Longrightarrow$ *punit.red* (*rep-list* ‘ F) (*rep-list* p) (*rep-list* q)
by (*auto simp*: *sig-red-def punit.red-def dest*: *sig-red-single-red-single*)

lemma *sig-red-monom-mult*:
sig-red sing-reg top-tail $F\ p\ q \Longrightarrow c \neq 0 \Longrightarrow$ *sig-red sing-reg top-tail* F (*monom-mult* $c\ s\ p$) (*monom-mult* $c\ s\ q$)
by (*auto simp*: *sig-red-def punit.red-def dest*: *sig-red-single-monom-mult*)

lemma *sig-red-sing-reg-cases*:
sig-red $(\preceq_t)$ *top-tail* $F\ p\ q =$ (*sig-red* $(=)$ *top-tail* $F\ p\ q \vee$ *sig-red* $(\prec_t)$ *top-tail* $F\ p\ q$)
by (*auto simp*: *sig-red-def sig-red-single-sing-reg-cases*)

corollary *sig-red-sing-regI*: *sig-red sing-reg top-tail* $F\ p\ q \Longrightarrow$ *sig-red* $(\preceq_t)$ *top-tail* $F\ p\ q$
by (*auto simp*: *sig-red-def intro*: *sig-red-single-sing-regI*)

lemma *sig-red-top-tail-cases*:
sig-red sing-reg $(\preceq)$ $F\ p\ q =$ (*sig-red sing-reg* $(=)$ $F\ p\ q \vee$ *sig-red sing-reg* $(\prec)$ $F\ p\ q$)
by (*auto simp*: *sig-red-def sig-red-single-top-tail-cases*)

corollary *sig-red-top-tailI*: *sig-red sing-reg top-tail* $F\ p\ q \Longrightarrow$ *sig-red sing-reg* $(\preceq)$ $F\ p\ q$
by (*auto simp*: *sig-red-def intro*: *sig-red-single-top-tailI*)

lemma *sig-red-wf-dgrad-max-set*:
  assumes *dickson-grading d* **and** $F \subseteq$ *dgrad-max-set d*
  shows *wfP* (*sig-red sing-reg top-tail F*)$^{-1-1}$
proof −
  from *assms* **have** *rep-list* ' $F \subseteq$ *punit-dgrad-max-set d* **by** (*rule dgrad-max-3*)
  with *assms*(*1*) **have** *wfP* (*punit.red* (*rep-list* ' *F*))$^{-1-1}$ **by** (*rule punit.red-wf-dgrad-p-set*)
  hence $*$: $\nexists f.\ \forall i.$ (*punit.red* (*rep-list* ' *F*))$^{-1-1}$ (*f* (*Suc i*)) (*f i*)
    by (*simp add*: *wf-iff-no-infinite-down-chain*[*to-pred*])
  show *?thesis* **unfolding** *wf-iff-no-infinite-down-chain*[*to-pred*]
  proof (*rule*, *elim exE*)
    fix *seq*
    assume $\forall i.$ (*sig-red sing-reg top-tail F*)$^{-1-1}$ (*seq* (*Suc i*)) (*seq i*)
    hence *sig-red sing-reg top-tail F* (*seq i*) (*seq* (*Suc i*)) **for** *i* **by** *simp*
    hence *punit.red* (*rep-list* ' *F*) ((*rep-list* $\circ$ *seq*) *i*) ((*rep-list* $\circ$ *seq*) (*Suc i*)) **for** *i*
      by (*auto intro*: *sig-red-red*)
    hence $\forall i.$ (*punit.red* (*rep-list* ' *F*))$^{-1-1}$ ((*rep-list* $\circ$ *seq*) (*Suc i*)) ((*rep-list* $\circ$ *seq*) *i*) **by** *simp*
    hence $\exists f.\ \forall i.$ (*punit.red* (*rep-list* ' *F*))$^{-1-1}$ (*f* (*Suc i*)) (*f i*) **by** *blast*
    with $*$ **show** *False* **..**
  qed
qed

lemma *dgrad-sig-set-closed-sig-red*:
  assumes *dickson-grading d* **and** $F \subseteq$ *dgrad-sig-set d* **and** $p \in$ *dgrad-sig-set d*
    and *sig-red sing-red top-tail F p q*
  shows $q \in$ *dgrad-sig-set d*
  using *assms* **by** (*auto simp*: *sig-red-def intro*: *dgrad-sig-set-closed-sig-red-single*)

lemma *sig-red-mono*: *sig-red sing-reg top-tail F p q* $\Longrightarrow F \subseteq F' \Longrightarrow$ *sig-red sing-reg top-tail F' p q*
  by (*auto simp*: *sig-red-def*)

lemma *sig-red-Un*:
  sig-red sing-reg top-tail $(A \cup B)$ *p q* $\longleftrightarrow$ (*sig-red sing-reg top-tail A p q* $\vee$ *sig-red sing-reg top-tail B p q*)
  by (*auto simp*: *sig-red-def*)

lemma *sig-red-subset*:
  assumes *sig-red sing-reg top-tail F p q* **and** *sing-reg* = $(\preceq_t)$ $\vee$ *sing-reg* = $(\prec_t)$
  shows *sig-red sing-reg top-tail* $\{f \in F.$ *sing-reg* (*lt f*) (*lt p*)$\}$ *p q*
proof −
  from *assms*(*1*) **obtain** *f t* **where** $f \in F$ **and** $*$: *sig-red-single sing-reg top-tail p q f t*
    unfolding *sig-red-def* **by** *blast*
  have *lt f* = *0* $\oplus$ *lt f* **by** (*simp only*: *term-simps*)
  also from *zero-min* **have** ... $\preceq_t$ $t \oplus$ *lt f* **by** (*rule splus-mono-left*)
  finally have *1*: *lt f* $\preceq_t$ $t \oplus$ *lt f* **.**
  from $*$ **have** *2*: *sing-reg* ($t \oplus$ *lt f*) (*lt p*) **by** (*rule sig-red-singleD6*)

from *assms*(*2*) **have** *sing-reg* (*lt f*) (*lt p*)
proof
assume *sing-reg* = ($\preceq_t$)
with *1 2* **show** *?thesis* **by** *simp*
next
assume *sing-reg* = ($\prec_t$)
with *1 2* **show** *?thesis* **by** *simp*
qed
with ‹$f \in F$› **have** $f \in \{f{\in}F.$ *sing-reg* (*lt f*) (*lt p*)$\}$ **by** *simp*
thus *?thesis* **using** $*$ **unfolding** *sig-red-def* **by** *blast*
qed

lemma *sig-red-regular-rtrancl-lt*:
assumes (*sig-red* ($\prec_t$) *top-tail F*)** *p q*
shows *lt q* = *lt p*
using *assms* **by** (*induct*, *auto dest*: *sig-red-regular-lt*)

lemma *sig-red-regular-rtrancl-lc*:
assumes (*sig-red* ($\prec_t$) *top-tail F*)** *p q*
shows *lc q* = *lc p*
using *assms* **by** (*induct*, *auto dest*: *sig-red-regular-lc*)

lemma *sig-red-rtrancl-lt*:
assumes (*sig-red sing-reg top-tail F*)** *p q*
shows *lt q* $\preceq_t$ *lt p*
using *assms* **by** (*induct*, *auto dest*: *sig-red-lt*)

lemma *sig-red-tail-rtrancl-lt-rep-list*:
assumes (*sig-red sing-reg* ($\prec$) *F*)** *p q*
shows *punit.lt* (*rep-list q*) = *punit.lt* (*rep-list p*)
using *assms* **by** (*induct*, *auto dest*: *sig-red-tail-lt-rep-list*)

lemma *sig-red-tail-rtrancl-lc-rep-list*:
assumes (*sig-red sing-reg* ($\prec$) *F*)** *p q*
shows *punit.lc* (*rep-list q*) = *punit.lc* (*rep-list p*)
using *assms* **by** (*induct*, *auto dest*: *sig-red-tail-lc-rep-list*)

lemma *sig-red-rtrancl-lt-rep-list*:
assumes (*sig-red sing-reg top-tail F*)** *p q*
shows *punit.lt* (*rep-list q*) $\preceq$ *punit.lt* (*rep-list p*)
using *assms* **by** (*induct*, *auto dest*: *sig-red-lt-rep-list*)

lemma *sig-red-red-rtrancl*:
assumes (*sig-red sing-reg top-tail F*)** *p q*
shows (*punit.red* (*rep-list* ‘ *F*))** (*rep-list p*) (*rep-list q*)
using *assms* **by** (*induct*, *auto dest*: *sig-red-red*)

lemma *sig-red-rtrancl-monom-mult*:
assumes (*sig-red sing-reg top-tail F*)** *p q*

shows $(sig\text{-}red\ sing\text{-}reg\ top\text{-}tail\ F)^{**}$ $(monom\text{-}mult\ c\ s\ p)$ $(monom\text{-}mult\ c\ s\ q)$
proof $(cases\ c = 0)$
case *True*
thus *?thesis* **by** *simp*
next
case *False*
from *assms*(*1*) **show** *?thesis*
proof *induct*
case *base*
show *?case* **..**
next
case (*step y z*)
from *step*(*2*) *False* **have** *sig-red sing-reg top-tail F* (*monom-mult c s y*) (*monom-mult c s z*)
by (*rule sig-red-monom-mult*)
with *step*(*3*) **show** *?case* **..**
qed
qed

lemma *sig-red-rtrancl-sing-regI*: $(sig\text{-}red\ sing\text{-}reg\ top\text{-}tail\ F)^{**}\ p\ q \Longrightarrow (sig\text{-}red\ (\preceq_t)\ top\text{-}tail\ F)^{**}\ p\ q$
by (*induct rule*: *rtranclp-induct*, *auto dest*: *sig-red-sing-regI*)

lemma *sig-red-rtrancl-top-tailI*: $(sig\text{-}red\ sing\text{-}reg\ top\text{-}tail\ F)^{**}\ p\ q \Longrightarrow (sig\text{-}red\ sing\text{-}reg\ (\preceq)\ F)^{**}\ p\ q$
by (*induct rule*: *rtranclp-induct*, *auto dest*: *sig-red-top-tailI*)

lemma *dgrad-sig-set-closed-sig-red-rtrancl*:
assumes *dickson-grading d* **and** $F \subseteq$ *dgrad-sig-set d* **and** $p \in$ *dgrad-sig-set d*
and $(sig\text{-}red\ sing\text{-}red\ top\text{-}tail\ F)^{**}\ p\ q$
shows $q \in$ *dgrad-sig-set d*
using *assms*(*4*, *1*, *2*, *3*) **by** (*induct*, *auto intro*: *dgrad-sig-set-closed-sig-red*)

lemma *sig-red-rtrancl-mono*:
assumes $(sig\text{-}red\ sing\text{-}reg\ top\text{-}tail\ F)^{**}\ p\ q$ **and** $F \subseteq F'$
shows $(sig\text{-}red\ sing\text{-}reg\ top\text{-}tail\ F')^{**}\ p\ q$
using *assms*(*1*) **by** (*induct rule*: *rtranclp-induct*, *auto dest*: *sig-red-mono*[*OF* - *assms*(*2*)])

lemma *sig-red-rtrancl-subset*:
assumes $(sig\text{-}red\ sing\text{-}reg\ top\text{-}tail\ F)^{**}\ p\ q$ **and** $sing\text{-}reg = (\preceq_t) \vee sing\text{-}reg = (\prec_t)$
shows $(sig\text{-}red\ sing\text{-}reg\ top\text{-}tail\ \{f{\in}F.\ sing\text{-}reg\ (lt\ f)\ (lt\ p)\})^{**}\ p\ q$
using *assms*(*1*)
proof (*induct rule*: *rtranclp-induct*)
case *base*
show *?case* **by** (*fact rtranclp.rtrancl-refl*)
next
case (*step y z*)

from *step*(*2*) *assms*(*2*) **have** *sig-red sing-reg top-tail* $\{f \in F.$ *sing-reg* (*lt f*) (*lt y*)$\}$ *y z*
by (*rule sig-red-subset*)
moreover have $\{f \in F.$ *sing-reg* (*lt f*) (*lt y*)$\} \subseteq \{f \in F.$ *sing-reg* (*lt f*) (*lt p*)$\}$
proof
fix *f*
assume $f \in \{f \in F.$ *sing-reg* (*lt f*) (*lt y*)$\}$
hence $f \in F$ **and** *1*: *sing-reg* (*lt f*) (*lt y*) **by** *simp-all*
from *step*(*1*) **have** *2*: *lt y* $\preceq_t$ *lt p* **by** (*rule sig-red-rtrancl-lt*)
from *assms*(*2*) **have** *sing-reg* (*lt f*) (*lt p*)
proof
assume *sing-reg* = ($\preceq_t$)
with *1 2* **show** *?thesis* **by** *simp*
next
assume *sing-reg* = ($\prec_t$)
with *1 2* **show** *?thesis* **by** *simp*
qed
with ‹$f \in F$› **show** $f \in \{f \in F.$ *sing-reg* (*lt f*) (*lt p*)$\}$ **by** *simp*
qed
ultimately have *sig-red sing-reg top-tail* $\{f \in F.$ *sing-reg* (*lt f*) (*lt p*)$\}$ *y z*
by (*rule sig-red-mono*)
with *step*(*3*) **show** *?case* **..**
qed

lemma *is-sig-red-is-red*: *is-sig-red sing-reg top-tail F p* $\Longrightarrow$ *punit.is-red* (*rep-list* ‘ *F*) (*rep-list p*)
by (*auto simp*: *is-sig-red-def punit.is-red-alt dest*: *sig-red-red*)

lemma *is-sig-red-monom-mult*:
assumes *is-sig-red sing-reg top-tail F p* **and** $c \neq 0$
shows *is-sig-red sing-reg top-tail F* (*monom-mult c s p*)
proof −
from *assms*(*1*) **obtain** *q* **where** *sig-red sing-reg top-tail F p q* **unfolding** *is-sig-red-def* **..**
hence *sig-red sing-reg top-tail F* (*monom-mult c s p*) (*monom-mult c s q*)
using *assms*(*2*) **by** (*rule sig-red-monom-mult*)
thus *?thesis* **unfolding** *is-sig-red-def* **..**
qed

lemma *is-sig-red-sing-reg-cases*:
is-sig-red ($\preceq_t$) *top-tail F p* = (*is-sig-red* (=) *top-tail F p* $\vee$ *is-sig-red* ($\prec_t$) *top-tail F p*)
by (*auto simp*: *is-sig-red-def sig-red-sing-reg-cases*)

corollary *is-sig-red-sing-regI*: *is-sig-red sing-reg top-tail F p* $\Longrightarrow$ *is-sig-red* ($\preceq_t$) *top-tail F p*
by (*auto simp*: *is-sig-red-def intro*: *sig-red-sing-regI*)

lemma *is-sig-red-top-tail-cases*:

is-sig-red sing-reg $(\preceq)$ *F p* = (*is-sig-red sing-reg* $(=)$ *F p* $\vee$ *is-sig-red sing-reg* $(\prec)$ *F p*)
by (*auto simp*: *is-sig-red-def sig-red-top-tail-cases*)

corollary *is-sig-red-top-tailI*: *is-sig-red sing-reg top-tail F p* $\Longrightarrow$ *is-sig-red sing-reg* $(\preceq)$ *F p*
by (*auto simp*: *is-sig-red-def intro*: *sig-red-top-tailI*)

lemma *is-sig-red-singletonI*:
assumes *is-sig-red sing-reg top-tail F r*
obtains *f* **where** $f \in F$ **and** *is-sig-red sing-reg top-tail* $\{f\}$ *r*
proof −
from *assms* **obtain** r' **where** *sig-red sing-reg top-tail F r* r' **unfolding** *is-sig-red-def* **..**
then obtain *f t* **where** $f \in F$ **and** *t*: *sig-red-single sing-reg top-tail r* r' *f t*
by (*auto simp*: *sig-red-def*)
have *is-sig-red sing-reg top-tail* $\{f\}$ *r* **unfolding** *is-sig-red-def sig-red-def*
proof (*intro exI bexI*)
show $f \in \{f\}$ **by** *simp*
qed *fact*
with ‹$f \in F$› **show** *?thesis* **..**
qed

lemma *is-sig-red-singletonD*:
assumes *is-sig-red sing-reg top-tail* $\{f\}$ *r* **and** $f \in F$
shows *is-sig-red sing-reg top-tail F r*
proof −
from *assms*(*1*) **obtain** r' **where** *sig-red sing-reg top-tail* $\{f\}$ *r* r' **unfolding** *is-sig-red-def* **..**
then obtain *t* **where** *sig-red-single sing-reg top-tail r* r' *f t* **by** (*auto simp*: *sig-red-def*)
show *?thesis* **unfolding** *is-sig-red-def sig-red-def* **by** (*intro exI bexI*, *fact+*)
qed

lemma *is-sig-redD1*:
assumes *is-sig-red sing-reg top-tail F p*
shows *ord-term-lin.is-le-rel sing-reg*
proof −
from *assms* **obtain** *q* **where** *sig-red sing-reg top-tail F p q* **unfolding** *is-sig-red-def* **..**
then obtain *f s* **where** $f \in F$ **and** *sig-red-single sing-reg top-tail p q f s* **unfolding** *sig-red-def* **by** *blast*
from *this*(*2*) **show** *?thesis* **by** (*rule sig-red-singleD*)
qed

lemma *is-sig-redD2*:
assumes *is-sig-red sing-reg top-tail F p*
shows *ordered-powerprod-lin.is-le-rel top-tail*
proof −

from *assms* **obtain** *q* **where** *sig-red sing-reg top-tail F p q* **unfolding** *is-sig-red-def*
..
then obtain *f s* **where** *f* ∈ *F* **and** *sig-red-single sing-reg top-tail p q f s* **unfolding** *sig-red-def* **by** *blast*
from *this*(*2*) **show** *?thesis* **by** (*rule sig-red-singleD*)
qed

lemma *is-sig-red-addsI*:
assumes *f* ∈ *F* **and** *t* ∈ *keys* (*rep-list p*) **and** *rep-list f* ≠ *0* **and** *punit.lt* (*rep-list f*) *adds t*
and *ord-term-lin.is-le-rel sing-reg* **and** *ordered-powerprod-lin.is-le-rel top-tail*
and *sing-reg* (*t* ⊕ *lt f*) (*punit.lt* (*rep-list f*) ⊕ *lt p*) **and** *top-tail t* (*punit.lt* (*rep-list p*))
shows *is-sig-red sing-reg top-tail F p*
unfolding *is-sig-red-def*
proof
let *?q* = *p* − *monom-mult* ((*lookup* (*rep-list p*) *t*) / *punit.lc* (*rep-list f*)) (*t* − *punit.lt* (*rep-list f*)) *f*
show *sig-red sing-reg top-tail F p ?q* **unfolding** *sig-red-def*
proof (*intro bexI exI*)
from *assms*(*4*) **have** *eq*: (*t* − *punit.lt* (*rep-list f*)) + *punit.lt* (*rep-list f*) = *t* **by** (*rule adds-minus*)
from *assms*(*4*, *5*, *7*) **have** *sing-reg* ((*t* − *punit.lt* (*rep-list f*)) ⊕ *lt f*) (*lt p*)
by (*simp only*: *term-is-le-rel-minus*)
thus *sig-red-single sing-reg top-tail p ?q f* (*t* − *punit.lt* (*rep-list f*))
by (*simp add*: *assms eq sig-red-singleI*)
qed *fact*
qed

lemma *is-sig-red-addsE*:
assumes *is-sig-red sing-reg top-tail F p*
obtains *f t* **where** *f* ∈ *F* **and** *t* ∈ *keys* (*rep-list p*) **and** *rep-list f* ≠ *0*
and *punit.lt* (*rep-list f*) *adds t*
and *sing-reg* (*t* ⊕ *lt f*) (*punit.lt* (*rep-list f*) ⊕ *lt p*)
and *top-tail t* (*punit.lt* (*rep-list p*))
proof −
from *assms* **have** ∗: *ord-term-lin.is-le-rel sing-reg* **by** (*rule is-sig-redD1*)
from *assms* **obtain** *q* **where** *sig-red sing-reg top-tail F p q* **unfolding** *is-sig-red-def*
..
then obtain *f s* **where** *f* ∈ *F* **and** *sig-red-single sing-reg top-tail p q f s* **unfolding** *sig-red-def* **by** *blast*
from *this*(*2*) **have** *1*: *rep-list f* ≠ *0* **and** *2*: *s* + *punit.lt* (*rep-list f*) ∈ *keys* (*rep-list p*)
and *3*: *sing-reg* (*s* ⊕ *lt f*) (*lt p*) **and** *4*: *top-tail* (*s* + *punit.lt* (*rep-list f*)) (*punit.lt* (*rep-list p*))
by (*rule sig-red-singleD*)+
note ‹*f* ∈ *F*› *2 1*
moreover have *punit.lt* (*rep-list f*) *adds s* + *punit.lt* (*rep-list f*) **by** *simp*
moreover from *3* **have** *sing-reg* ((*s* + *punit.lt* (*rep-list f*)) ⊕ *lt f*) (*punit.lt*

(*rep-list f*) ⊕ *lt p*)
  by (*simp add*: *add.commute*[*of s*] *splus-assoc term-is-le-rel-canc-left*[*OF* ∗])
 moreover from *4* **have** *top-tail* (*s* + *punit.lt* (*rep-list f*)) (*punit.lt* (*rep-list p*))
by *simp*
 ultimately show *?thesis* **..**
qed

lemma *is-sig-red-top-addsI*:
 assumes $f \in F$ **and** *rep-list f* $\neq$ *0* **and** *rep-list p* $\neq$ *0*
  and *punit.lt* (*rep-list f*) *adds punit.lt* (*rep-list p*) **and** *ord-term-lin.is-le-rel sing-reg*
  and *sing-reg* (*punit.lt* (*rep-list p*) ⊕ *lt f*) (*punit.lt* (*rep-list f*) ⊕ *lt p*)
 shows *is-sig-red sing-reg* (=) *F p*
proof −
 note *assms*(*1*)
 moreover from *assms*(*3*) **have** *punit.lt* (*rep-list p*) ∈ *keys* (*rep-list p*) **by** (*rule punit.lt-in-keys*)
 moreover note *assms*(*2*, *4*, *5*) *ordered-powerprod-lin.is-le-relI*(*1*) *assms*(*6*) *refl*
 ultimately show *?thesis* **by** (*rule is-sig-red-addsI*)
qed

lemma *is-sig-red-top-addsE*:
 assumes *is-sig-red sing-reg* (=) *F p*
 obtains *f* **where** $f \in F$ **and** *rep-list f* $\neq$ *0* **and** *rep-list p* $\neq$ *0*
  and *punit.lt* (*rep-list f*) *adds punit.lt* (*rep-list p*)
  and *sing-reg* (*punit.lt* (*rep-list p*) ⊕ *lt f*) (*punit.lt* (*rep-list f*) ⊕ *lt p*)
proof −
 from *assms* **obtain** *f t* **where** *1*: $f \in F$ **and** *2*: $t \in$ *keys* (*rep-list p*) **and** *3*: *rep-list f* $\neq$ *0*
  and *4*: *punit.lt* (*rep-list f*) *adds t*
  and *5*: *sing-reg* (*t* ⊕ *lt f*) (*punit.lt* (*rep-list f*) ⊕ *lt p*)
  and *t*: *t* = *punit.lt* (*rep-list p*) **by** (*rule is-sig-red-addsE*)
 note *1 3*
 moreover from *2* **have** *rep-list p* $\neq$ *0* **by** *auto*
 moreover from *4* **have** *punit.lt* (*rep-list f*) *adds punit.lt* (*rep-list p*) **by** (*simp only*: *t*)
 moreover from *5* **have** *sing-reg* (*punit.lt* (*rep-list p*) ⊕ *lt f*) (*punit.lt* (*rep-list f*) ⊕ *lt p*)
  by (*simp only*: *t*)
 ultimately show *?thesis* **..**
qed

lemma *is-sig-red-top-plusE*:
 assumes *is-sig-red sing-reg* (=) *F p* **and** *is-sig-red sing-reg* (=) *F q*
  and *lt p* $\preceq_t$ *lt* (*p* + *q*) **and** *lt q* $\preceq_t$ *lt* (*p* + *q*) **and** *sing-reg* = ($\preceq_t$) ∨ *sing-reg* = ($\prec_t$)
 assumes *1*: *is-sig-red sing-reg* (=) *F* (*p* + *q*) ⟹ *thesis*
 assumes *2*: *punit.lt* (*rep-list p*) = *punit.lt* (*rep-list q*) ⟹ *punit.lc* (*rep-list p*) + *punit.lc* (*rep-list q*) = *0* ⟹ *thesis*

shows *thesis*
proof −
from *assms*(*1*) **obtain** *f1* **where** *f1* ∈ *F* **and** *rep-list f1* ≠ *0* **and** *rep-list p* ≠ *0*
and *a*: *punit.lt* (*rep-list f1*) *adds punit.lt* (*rep-list p*)
and *b*: *sing-reg* (*punit.lt* (*rep-list p*) ⊕ *lt f1*) (*punit.lt* (*rep-list f1*) ⊕ *lt p*)
by (*rule is-sig-red-top-addsE*)
from *assms*(*2*) **obtain** *f2* **where** *f2* ∈ *F* **and** *rep-list f2* ≠ *0* **and** *rep-list q* ≠ *0*
and *c*: *punit.lt* (*rep-list f2*) *adds punit.lt* (*rep-list q*)
and *d*: *sing-reg* (*punit.lt* (*rep-list q*) ⊕ *lt f2*) (*punit.lt* (*rep-list f2*) ⊕ *lt q*)
by (*rule is-sig-red-top-addsE*)
show *?thesis*
proof (*cases punit.lt* (*rep-list p*) = *punit.lt* (*rep-list q*) ∧ *punit.lc* (*rep-list p*) + *punit.lc* (*rep-list q*) = *0*)
case *True*
hence *punit.lt* (*rep-list p*) = *punit.lt* (*rep-list q*) **and** *punit.lc* (*rep-list p*) + *punit.lc* (*rep-list q*) = *0*
by *simp-all*
thus *?thesis* **by** (*rule 2*)
next
case *False*
hence *disj*: *punit.lt* (*rep-list p*) ≠ *punit.lt* (*rep-list q*) ∨ *punit.lc* (*rep-list p*) + *punit.lc* (*rep-list q*) ≠ *0*
by *simp*
from *assms*(*5*) **have** *ord-term-lin.is-le-rel sing-reg* **by** (*simp add*: *ord-term-lin.is-le-rel-def*)
have *rep-list* (*p* + *q*) ≠ *0* **unfolding** *rep-list-plus*
proof
assume *eq*: *rep-list p* + *rep-list q* = *0*
have *eq2*: *punit.lt* (*rep-list p*) = *punit.lt* (*rep-list q*)
proof (*rule ordered-powerprod-lin.linorder-cases*)
assume ∗: *punit.lt* (*rep-list p*) ≺ *punit.lt* (*rep-list q*)
hence *punit.lt* (*rep-list p* + *rep-list q*) = *punit.lt* (*rep-list q*) **by** (*rule punit.lt-plus-eqI*)
with ∗ *zero-min*[*of punit.lt* (*rep-list p*)] **show** *?thesis* **by** (*simp add*: *eq*)
next
assume ∗: *punit.lt* (*rep-list q*) ≺ *punit.lt* (*rep-list p*)
hence *punit.lt* (*rep-list p* + *rep-list q*) = *punit.lt* (*rep-list p*) **by** (*rule punit.lt-plus-eqI-2*)
with ∗ *zero-min*[*of punit.lt* (*rep-list q*)] **show** *?thesis* **by** (*simp add*: *eq*)
qed
with *disj* **have** *punit.lc* (*rep-list p*) + *punit.lc* (*rep-list q*) ≠ *0* **by** *simp*
thus *False* **by** (*simp add*: *punit.lc-def eq2 lookup-add*[*symmetric*] *eq*)
qed
have *punit.lt* (*rep-list* (*p* + *q*)) = *ordered-powerprod-lin.max* (*punit.lt* (*rep-list p*)) (*punit.lt* (*rep-list q*))
unfolding *rep-list-plus*
proof (*rule punit.lt-plus-eq-maxI*)
assume *punit.lt* (*rep-list p*) = *punit.lt* (*rep-list q*)
with *disj* **show** *punit.lc* (*rep-list p*) + *punit.lc* (*rep-list q*) ≠ *0* **by** *simp*
qed

hence *punit.lt* (*rep-list* (*p* + *q*)) = *punit.lt* (*rep-list p*) ∨ *punit.lt* (*rep-list* (*p* + *q*)) = *punit.lt* (*rep-list q*)
by (*simp add*: *ordered-powerprod-lin.max-def*)
thus *?thesis*
proof
assume *eq*: *punit.lt* (*rep-list* (*p* + *q*)) = *punit.lt* (*rep-list p*)
show *?thesis*
proof (*rule 1*, *rule is-sig-red-top-addsI*)
from *a* **show** *punit.lt* (*rep-list f1*) *adds punit.lt* (*rep-list* (*p* + *q*)) **by** (*simp only*: *eq*)
next
from *b* **have** *sing-reg* (*punit.lt* (*rep-list* (*p* + *q*)) ⊕ *lt f1*) (*punit.lt* (*rep-list f1*) ⊕ *lt p*)
by (*simp only*: *eq*)
moreover from *assms*(*3*) **have** ... $\preceq_t$ *punit.lt* (*rep-list f1*) ⊕ *lt* (*p* + *q*) **by** (*rule splus-mono*)
ultimately show *sing-reg* (*punit.lt* (*rep-list* (*p* + *q*)) ⊕ *lt f1*) (*punit.lt* (*rep-list f1*) ⊕ *lt* (*p* + *q*))
using *assms*(*5*) **by** *auto*
qed *fact*+
next
assume *eq*: *punit.lt* (*rep-list* (*p* + *q*)) = *punit.lt* (*rep-list q*)
show *?thesis*
proof (*rule 1*, *rule is-sig-red-top-addsI*)
from *c* **show** *punit.lt* (*rep-list f2*) *adds punit.lt* (*rep-list* (*p* + *q*)) **by** (*simp only*: *eq*)
next
from *d* **have** *sing-reg* (*punit.lt* (*rep-list* (*p* + *q*)) ⊕ *lt f2*) (*punit.lt* (*rep-list f2*) ⊕ *lt q*)
by (*simp only*: *eq*)
moreover from *assms*(*4*) **have** ... $\preceq_t$ *punit.lt* (*rep-list f2*) ⊕ *lt* (*p* + *q*) **by** (*rule splus-mono*)
ultimately show *sing-reg* (*punit.lt* (*rep-list* (*p* + *q*)) ⊕ *lt f2*) (*punit.lt* (*rep-list f2*) ⊕ *lt* (*p* + *q*))
using *assms*(*5*) **by** *auto*
qed *fact*+
qed
qed
qed

lemma *is-sig-red-singleton-monom-multD*:
assumes *is-sig-red sing-reg top-tail* {*monom-mult c t f*} *p*
shows *is-sig-red sing-reg top-tail* {*f*} *p*
proof −
let *?f* = *monom-mult c t f*
from *assms* **obtain** *s* **where** *s* ∈ *keys* (*rep-list p*) **and** *2*: *rep-list ?f* ≠ *0*
and *3*: *punit.lt* (*rep-list ?f*) *adds s*
and *4*: *sing-reg* (*s* ⊕ *lt ?f*) (*punit.lt* (*rep-list ?f*) ⊕ *lt p*)
and *top-tail s* (*punit.lt* (*rep-list p*))

by (*auto elim*: *is-sig-red-addsE*)
from *2* **have** $c \neq 0$ **and** *rep-list f* $\neq 0$
by (*simp-all add*: *rep-list-monom-mult punit.monom-mult-eq-zero-iff*)
hence $f \neq 0$ **by** (*auto simp*: *rep-list-zero*)
with ‹$c \neq 0$› **have** *eq1*: *lt ?f* = *t* ⊕ *lt f* **by** (*simp add*: *lt-monom-mult*)
from ‹$c \neq 0$› ‹*rep-list f* $\neq 0$› **have** *eq2*: *punit.lt* (*rep-list ?f*) = *t* + *punit.lt* (*rep-list f*)
by (*simp add*: *rep-list-monom-mult punit.lt-monom-mult*)
from *assms* **have** ∗: *ord-term-lin.is-le-rel sing-reg* **by** (*rule is-sig-redD1*)
show *?thesis*
proof (*rule is-sig-red-addsI*)
show *f* ∈ {*f*} **by** *simp*
next
have *punit.lt* (*rep-list f*) *adds t* + *punit.lt* (*rep-list f*) **by** (*rule adds-triv-right*)
also from *3* **have** ... *adds s* **by** (*simp only*: *eq2*)
finally show *punit.lt* (*rep-list f*) *adds s* **.**
next
from *4* **have** *sing-reg* (*t* ⊕ (*s* ⊕ *lt f*)) (*t* ⊕ (*punit.lt* (*rep-list f*) ⊕ *lt p*))
by (*simp add*: *eq1 eq2 splus-assoc splus-left-commute*)
with ∗ **show** *sing-reg* (*s* ⊕ *lt f*) (*punit.lt* (*rep-list f*) ⊕ *lt p*)
by (*simp add*: *term-is-le-rel-canc-left*)
next
from *assms* **show** *ordered-powerprod-lin.is-le-rel top-tail* **by** (*rule is-sig-redD2*)
qed *fact+*
qed

lemma *is-sig-red-top-singleton-monom-multI*:
assumes *is-sig-red sing-reg* (=) {*f*} *p* **and** $c \neq 0$
and *t adds punit.lt* (*rep-list p*) − *punit.lt* (*rep-list f*)
shows *is-sig-red sing-reg* (=) {*monom-mult c t f*} *p*
proof −
let *?f* = *monom-mult c t f*
from *assms* **have** *2*: *rep-list f* $\neq 0$ **and** *rep-list p* $\neq 0$
and *3*: *punit.lt* (*rep-list f*) *adds punit.lt* (*rep-list p*)
and *4*: *sing-reg* (*punit.lt* (*rep-list p*) ⊕ *lt f*) (*punit.lt* (*rep-list f*) ⊕ *lt p*)
by (*auto elim*: *is-sig-red-top-addsE*)
hence $f \neq 0$ **by** (*auto simp*: *rep-list-zero*)
with ‹$c \neq 0$› **have** *eq1*: *lt ?f* = *t* ⊕ *lt f* **by** (*simp add*: *lt-monom-mult*)
from ‹$c \neq 0$› ‹*rep-list f* $\neq 0$› **have** *eq2*: *punit.lt* (*rep-list ?f*) = *t* + *punit.lt* (*rep-list f*)
by (*simp add*: *rep-list-monom-mult punit.lt-monom-mult*)
from *assms*(*1*) **have** ∗: *ord-term-lin.is-le-rel sing-reg* **by** (*rule is-sig-redD1*)
show *?thesis*
proof (*rule is-sig-red-top-addsI*)
show *?f* ∈ {*?f*} **by** *simp*
next
from ‹$c \neq 0$› ‹*rep-list f* $\neq 0$› **show** *rep-list ?f* $\neq 0$
by (*simp add*: *rep-list-monom-mult punit.monom-mult-eq-zero-iff*)
next

from *assms*(*3*) **have** *t* + *punit.lt* (*rep-list f*) *adds*
(*punit.lt* (*rep-list p*) − *punit.lt* (*rep-list f*)) + *punit.lt* (*rep-list f*)
by (*simp only*: *adds-canc*)
also from *3* **have** ... = *punit.lt* (*rep-list p*) **by** (*rule adds-minus*)
finally show *punit.lt* (*rep-list ?f*) *adds punit.lt* (*rep-list p*) **by** (*simp only*: *eq2*)
next
from *4* ∗ **show** *sing-reg* (*punit.lt* (*rep-list p*) ⊕ *lt ?f*) (*punit.lt* (*rep-list ?f*) ⊕ *lt p*)
by (*simp add*: *eq1 eq2 term-is-le-rel-canc-left splus-assoc splus-left-commute*)
qed *fact*+
qed

lemma *is-sig-red-cong′*:
assumes *is-sig-red sing-reg top-tail F p* **and** *lt p* = *lt q* **and** *rep-list p* = *rep-list q*
shows *is-sig-red sing-reg top-tail F q*
proof −
from *assms*(*1*) **have** *1*: *ord-term-lin.is-le-rel sing-reg* **and** *2*: *ordered-powerprod-lin.is-le-rel top-tail*
by (*rule is-sig-redD1*, *rule is-sig-redD2*)
from *assms*(*1*) **obtain** *f t* **where** *f* ∈ *F* **and** *t* ∈ *keys* (*rep-list p*) **and** *rep-list f* ≠ *0*
and *punit.lt* (*rep-list f*) *adds t*
and *sing-reg* (*t* ⊕ *lt f*) (*punit.lt* (*rep-list f*) ⊕ *lt p*)
and *top-tail t* (*punit.lt* (*rep-list p*)) **by** (*rule is-sig-red-addsE*)
from *this*(*1*−*4*) *1 2 this*(*5*, *6*) **show** *?thesis* **unfolding** *assms*(*2*, *3*) **by** (*rule is-sig-red-addsI*)
qed

lemma *is-sig-red-cong*:
lt p = *lt q* ⟹ *rep-list p* = *rep-list q* ⟹
is-sig-red sing-reg top-tail F p ⟷ *is-sig-red sing-reg top-tail F q*
by (*auto intro*: *is-sig-red-cong′*)

lemma *is-sig-red-top-cong*:
assumes *is-sig-red sing-reg* (=) *F p* **and** *rep-list q* ≠ *0* **and** *lt p* = *lt q*
and *punit.lt* (*rep-list p*) = *punit.lt* (*rep-list q*)
shows *is-sig-red sing-reg* (=) *F q*
proof −
from *assms*(*1*) **have** *1*: *ord-term-lin.is-le-rel sing-reg* **by** (*rule is-sig-redD1*)
from *assms*(*1*) **obtain** *f* **where** *f* ∈ *F* **and** *rep-list f* ≠ *0* **and** *rep-list p* ≠ *0*
and *punit.lt* (*rep-list f*) *adds punit.lt* (*rep-list p*)
and *sing-reg* (*punit.lt* (*rep-list p*) ⊕ *lt f*) (*punit.lt* (*rep-list f*) ⊕ *lt p*)
by (*rule is-sig-red-top-addsE*)
from *this*(*1*, *2*) *assms*(*2*) *this*(*4*) *1 this*(*5*) **show** *?thesis*
unfolding *assms*(*3*, *4*) **by** (*rule is-sig-red-top-addsI*)
qed

lemma *sig-irredE-dgrad-max-set*:

assumes *dickson-grading d* **and** $F \subseteq$ *dgrad-max-set d*
obtains *q* **where** (*sig-red sing-reg top-tail F*)** *p q* **and** $\neg$ *is-sig-red sing-reg top-tail F q*
proof −
let *?Q* = {*q*. (*sig-red sing-reg top-tail F*)** *p q*}
from *assms* **have** *wfP* (*sig-red sing-reg top-tail F*)$^{-1-1}$ **by** (*rule sig-red-wf-dgrad-max-set*)
moreover have $p \in$ *?Q* **by** *simp*
ultimately obtain *q* **where** $q \in$ *?Q* **and** $\bigwedge x$. (*sig-red sing-reg top-tail F*)$^{-1-1}$ *x q* $\Longrightarrow x \notin$ *?Q*
by (*rule wfE-min*[*to-pred*], *blast*)
hence *1*: (*sig-red sing-reg top-tail F*)** *p q*
and *2*: $\bigwedge x$. *sig-red sing-reg top-tail F q x* $\Longrightarrow \neg$ (*sig-red sing-reg top-tail F*)** *p x*
by *simp-all*
show *?thesis*
proof
show $\neg$ *is-sig-red sing-reg top-tail F q*
proof
assume *is-sig-red sing-reg top-tail F q*
then obtain *x* **where** *3*: *sig-red sing-reg top-tail F q x* **unfolding** *is-sig-red-def* **..**
hence $\neg$ (*sig-red sing-reg top-tail F*)** *p x* **by** (*rule 2*)
moreover from *1 3* **have** (*sig-red sing-reg top-tail F*)** *p x* **..**
ultimately show *False* **..**
qed
qed *fact*
qed

lemma *is-sig-red-mono*:
is-sig-red sing-reg top-tail F p $\Longrightarrow F \subseteq F' \Longrightarrow$ *is-sig-red sing-reg top-tail F' p*
by (*auto simp*: *is-sig-red-def dest*: *sig-red-mono*)

lemma *is-sig-red-Un*:
is-sig-red sing-reg top-tail ($A \cup B$) *p* $\longleftrightarrow$ (*is-sig-red sing-reg top-tail A p* $\vee$ *is-sig-red sing-reg top-tail B p*)
by (*auto simp*: *is-sig-red-def sig-red-Un*)

lemma *is-sig-redD-lt*:
assumes *is-sig-red* ($\preceq_t$) *top-tail* {*f*} *p*
shows *lt f* $\preceq_t$ *lt p*
proof −
from *assms* **obtain** *s* **where** *rep-list f* $\neq$ *0* **and** $s \in$ *keys* (*rep-list p*)
and *1*: *punit.lt* (*rep-list f*) *adds s* **and** *2*: $s \oplus$ *lt f* $\preceq_t$ *punit.lt* (*rep-list f*) $\oplus$ *lt p*
by (*auto elim*!: *is-sig-red-addsE*)
from *1* **obtain** *t* **where** *eq*: *s* = *punit.lt* (*rep-list f*) + *t* **by** (*rule addsE*)
hence *punit.lt* (*rep-list f*) $\oplus$ ($t \oplus$ *lt f*) = $s \oplus$ *lt f* **by** (*simp add*: *splus-assoc*)
also note *2*
finally have $t \oplus$ *lt f* $\preceq_t$ *lt p* **by** (*rule ord-term-canc*)
have *0* $\preceq t$ **by** (*fact zero-min*)

hence $0 \oplus lt\ f \preceq_t t \oplus lt\ f$ **by** (*rule splus-mono-left*)
hence $lt\ f \preceq_t t \oplus lt\ f$ **by** (*simp add*: *term-simps*)
thus *?thesis* **using** ‹$t \oplus lt\ f \preceq_t lt\ p$› **by** *simp*
qed

lemma *is-sig-red-regularD-lt*:
assumes *is-sig-red* ($\prec_t$) *top-tail* $\{f\}$ *p*
shows $lt\ f \prec_t lt\ p$
proof −
from *assms* **obtain** *s* **where** *rep-list f* $\neq$ *0* **and** $s \in$ *keys* (*rep-list p*)
and *1*: *punit.lt* (*rep-list f*) *adds s* **and** *2*: $s \oplus lt\ f \prec_t$ *punit.lt* (*rep-list f*) $\oplus$ *lt p*
by (*auto elim*!: *is-sig-red-addsE*)
from *1* **obtain** *t* **where** *eq*: *s* = *punit.lt* (*rep-list f*) + *t* **by** (*rule addsE*)
hence *punit.lt* (*rep-list f*) $\oplus$ ($t \oplus lt\ f$) = $s \oplus lt\ f$ **by** (*simp add*: *splus-assoc*)
also note *2*
finally have $t \oplus lt\ f \prec_t lt\ p$ **by** (*rule ord-term-strict-canc*)
have $0 \preceq t$ **by** (*fact zero-min*)
hence $0 \oplus lt\ f \preceq_t t \oplus lt\ f$ **by** (*rule splus-mono-left*)
hence $lt\ f \preceq_t t \oplus lt\ f$ **by** (*simp add*: *term-simps*)
thus *?thesis* **using** ‹$t \oplus lt\ f \prec_t lt\ p$› **by** (*rule ord-term-lin.le-less-trans*)
qed

lemma *sig-irred-regular-self*: ¬ *is-sig-red* ($\prec_t$) *top-tail* $\{p\}$ *p*
by (*auto dest*: *is-sig-red-regularD-lt*)

4.2.2 Signature Gröbner Bases

definition *sig-red-zero* :: $('t \Rightarrow 't \Rightarrow bool) \Rightarrow ('t \Rightarrow_0 'b)\ set \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow bool$
where *sig-red-zero sing-reg F r* $\longleftrightarrow$ ($\exists$ *s*. (*sig-red sing-reg* ($\preceq$) *F*)** *r s* $\wedge$ *rep-list s* = *0*)

definition *is-sig-GB-in* :: $('a \Rightarrow nat) \Rightarrow ('t \Rightarrow_0 'b)\ set \Rightarrow 't \Rightarrow bool$
where *is-sig-GB-in d G u* $\longleftrightarrow$ ($\forall$ *r*. *lt r* = *u* $\longrightarrow$ *r* $\in$ *dgrad-sig-set d* $\longrightarrow$ *sig-red-zero* ($\preceq_t$) *G r*)

definition *is-sig-GB-upt* :: $('a \Rightarrow nat) \Rightarrow ('t \Rightarrow_0 'b)\ set \Rightarrow 't \Rightarrow bool$
where *is-sig-GB-upt d G u* $\longleftrightarrow$
($G \subseteq$ *dgrad-sig-set d* $\wedge$ ($\forall$ *v*. $v \prec_t u \longrightarrow$ *d* (*pp-of-term v*) $\leq$ *dgrad-max d* $\longrightarrow$
component-of-term v < *length fs* $\longrightarrow$ *is-sig-GB-in d G v*))

definition *is-min-sig-GB* :: $('a \Rightarrow nat) \Rightarrow ('t \Rightarrow_0 'b)\ set \Rightarrow bool$
where *is-min-sig-GB d G* $\longleftrightarrow$ $G \subseteq$ *dgrad-sig-set d* $\wedge$
($\forall$ *u*. *d* (*pp-of-term u*) $\leq$ *dgrad-max d* $\longrightarrow$ *component-of-term u* < *length fs* $\longrightarrow$
is-sig-GB-in d G u) $\wedge$
($\forall$ *g*$\in$*G*. ¬ *is-sig-red* ($\preceq_t$) (=) ($G - \{g\}$) *g*)

definition *is-syz-sig* :: $('a \Rightarrow nat) \Rightarrow 't \Rightarrow bool$
 where *is-syz-sig d u* $\longleftrightarrow$ ($\exists$ *s*$\in$*dgrad-sig-set d*. *s* $\neq$ *0* $\wedge$ *lt s* = *u* $\wedge$ *rep-list s* = *0*)

lemma *sig-red-zeroI*:
 assumes (*sig-red sing-reg* ($\preceq$) *F*)** *r s* **and** *rep-list s* = *0*
 shows *sig-red-zero sing-reg F r*
 unfolding *sig-red-zero-def* **using** *assms* **by** *blast*

lemma *sig-red-zeroE*:
 assumes *sig-red-zero sing-reg F r*
 obtains *s* **where** (*sig-red sing-reg* ($\preceq$) *F*)** *r s* **and** *rep-list s* = *0*
 using *assms* **unfolding** *sig-red-zero-def* **by** *blast*

lemma *sig-red-zero-monom-mult*:
 assumes *sig-red-zero sing-reg F r*
 shows *sig-red-zero sing-reg F* (*monom-mult c t r*)
proof –
 from *assms* **obtain** *s* **where** (*sig-red sing-reg* ($\preceq$) *F*)** *r s* **and** *rep-list s* = *0*
  by (*rule sig-red-zeroE*)
 from *this*(*1*) **have** (*sig-red sing-reg* ($\preceq$) *F*)** (*monom-mult c t r*) (*monom-mult c t s*)
  by (*rule sig-red-rtrancl-monom-mult*)
 moreover have *rep-list* (*monom-mult c t s*) = *0* **by** (*simp add*: *rep-list-monom-mult* ‹*rep-list s* = *0*›)
 ultimately show *?thesis* **by** (*rule sig-red-zeroI*)
qed

lemma *sig-red-zero-sing-regI*:
 assumes *sig-red-zero sing-reg G p*
 shows *sig-red-zero* ($\preceq_t$) *G p*
proof –
 from *assms* **obtain** *s* **where** (*sig-red sing-reg* ($\preceq$) *G*)** *p s* **and** *rep-list s* = *0*
  by (*rule sig-red-zeroE*)
 from *this*(*1*) **have** (*sig-red* ($\preceq_t$) ($\preceq$) *G*)** *p s* **by** (*rule sig-red-rtrancl-sing-regI*)
 thus *?thesis* **using** ‹*rep-list s* = *0*› **by** (*rule sig-red-zeroI*)
qed

lemma *sig-red-zero-nonzero*:
 assumes *sig-red-zero sing-reg F r* **and** *rep-list r* $\neq$ *0* **and** *sing-reg* = ($\preceq_t$) $\vee$ *sing-reg* = ($\prec_t$)
 shows *is-sig-red sing-reg* (=) *F r*
proof –
 from *assms*(*1*) **obtain** *s* **where** (*sig-red sing-reg* ($\preceq$) *F*)** *r s* **and** *rep-list s* = *0*
  by (*rule sig-red-zeroE*)
 from *this*(*1*) *assms*(*2*) **show** *?thesis*
 proof (*induct rule*: *converse-rtranclp-induct*)
  case *base*

thus *?case* **using** ‹*rep-list s* = *0*› **..**
next
case (*step y z*)
from *step*(*1*) **obtain** *f t* **where** *f* ∈ *F* **and** ∗: *sig-red-single sing-reg* (⪯) *y z f t*
unfolding *sig-red-def* **by** *blast*
from *this*(*2*) **have** *1*: *rep-list f* ≠ *0* **and** *2*: *t* + *punit.lt* (*rep-list f*) ∈ *keys* (*rep-list y*)
and *3*: *z* = *y* − *monom-mult* (*lookup* (*rep-list y*) (*t* + *punit.lt* (*rep-list f*)) / *punit.lc* (*rep-list f*)) *t f*
and *4*: *ord-term-lin.is-le-rel sing-reg* **and** *5*: *sing-reg* (*t* ⊕ *lt f*) (*lt y*)
by (*rule sig-red-singleD*)+
show *?case*
proof (*cases t* + *punit.lt* (*rep-list f*) = *punit.lt* (*rep-list y*))
case *True*
show *?thesis* **unfolding** *is-sig-red-def*
proof
show *sig-red sing-reg* (=) *F y z* **unfolding** *sig-red-def*
proof (*intro bexI exI*)
from *1 2 3 4 ordered-powerprod-lin.is-le-relI*(*1*) *5 True*
show *sig-red-single sing-reg* (=) *y z f t* **by** (*rule sig-red-singleI*)
qed *fact*
qed
next
case *False*
from *2* **have** *t* + *punit.lt* (*rep-list f*) ⪯ *punit.lt* (*rep-list y*) **by** (*rule punit.lt-max-keys*)
with *False* **have** *t* + *punit.lt* (*rep-list f*) ≺ *punit.lt* (*rep-list y*) **by** *simp*
with *1 2 3 4 ordered-powerprod-lin.is-le-relI*(*3*) *5* **have** *sig-red-single sing-reg* (≺) *y z f t*
by (*rule sig-red-singleI*)
hence *punit.lt* (*rep-list y*) ∈ *keys* (*rep-list z*)
and *lt-z*: *punit.lt* (*rep-list z*) = *punit.lt* (*rep-list y*)
by (*rule sig-red-single-tail-lt-in-keys-rep-list*, *rule sig-red-single-tail-lt-rep-list*)
from *this*(*1*) **have** *rep-list z* ≠ *0* **by** *auto*
hence *is-sig-red sing-reg* (=) *F z* **by** (*rule step*(*3*))
then obtain *g* **where** *g* ∈ *F* **and** *rep-list g* ≠ *0*
and *punit.lt* (*rep-list g*) *adds punit.lt* (*rep-list z*)
and *a*: *sing-reg* (*punit.lt* (*rep-list z*) ⊕ *lt g*) (*punit.lt* (*rep-list g*) ⊕ *lt z*)
by (*rule is-sig-red-top-addsE*)
from *this*(*3*) **have** *punit.lt* (*rep-list g*) *adds punit.lt* (*rep-list y*) **by** (*simp only*: *lt-z*)
with ‹*g* ∈ *F*› ‹*rep-list g* ≠ *0*› *step*(*4*) **show** *?thesis*
proof (*rule is-sig-red-top-addsI*)
from ‹*is-sig-red sing-reg* (=) *F z*› **show** *ord-term-lin.is-le-rel sing-reg* **by** (*rule is-sig-redD1*)
next
from ‹*sig-red-single sing-reg* (≺) *y z f t*› **have** *lt z* $\preceq_t$ *lt y* **by** (*rule sig-red-single-lt*)
from *assms*(*3*) **show** *sing-reg* (*punit.lt* (*rep-list y*) ⊕ *lt g*) (*punit.lt* (*rep-list*

g) ⊕ *lt y*)
proof
assume *sing-reg* = ($\preceq_t$)
from *a* **have** *punit.lt* (*rep-list y*) ⊕ *lt g* $\preceq_t$ *punit.lt* (*rep-list g*) ⊕ *lt z*
by (*simp only*: *lt-z* ‹*sing-reg* = ($\preceq_t$)›)
also from ‹*lt z* $\preceq_t$ *lt y*› **have** ... $\preceq_t$ *punit.lt* (*rep-list g*) ⊕ *lt y* **by** (*rule splus-mono*)
finally show *?thesis* **by** (*simp only*: ‹*sing-reg* = ($\preceq_t$)›)
next
assume *sing-reg* = ($\prec_t$)
from *a* **have** *punit.lt* (*rep-list y*) ⊕ *lt g* $\prec_t$ *punit.lt* (*rep-list g*) ⊕ *lt z*
by (*simp only*: *lt-z* ‹*sing-reg* = ($\prec_t$)›)
also from ‹*lt z* $\preceq_t$ *lt y*› **have** ... $\preceq_t$ *punit.lt* (*rep-list g*) ⊕ *lt y* **by** (*rule splus-mono*)
finally show *?thesis* **by** (*simp only*: ‹*sing-reg* = ($\prec_t$)›)
qed
qed
qed
qed
qed

lemma *sig-red-zero-mono*: *sig-red-zero sing-reg F p* $\Longrightarrow$ *F* ⊆ *F'* $\Longrightarrow$ *sig-red-zero sing-reg F' p*
by (*auto simp*: *sig-red-zero-def dest*: *sig-red-rtrancl-mono*)

lemma *sig-red-zero-subset*:
assumes *sig-red-zero sing-reg F p* **and** *sing-reg* = ($\preceq_t$) ∨ *sing-reg* = ($\prec_t$)
shows *sig-red-zero sing-reg* {*f*∈*F*. *sing-reg* (*lt f*) (*lt p*)} *p*
proof −
from *assms*(*1*) **obtain** *s* **where** (*sig-red sing-reg* (⪯) *F*)** *p s* **and** *rep-list s* = *0*
by (*rule sig-red-zeroE*)
from *this*(*1*) *assms*(*2*) **have** (*sig-red sing-reg* (⪯) {*f*∈*F*. *sing-reg* (*lt f*) (*lt p*)})** *p s*
by (*rule sig-red-rtrancl-subset*)
thus *?thesis* **using** ‹*rep-list s* = *0*› **by** (*rule sig-red-zeroI*)
qed

lemma *sig-red-zero-idealI*:
assumes *sig-red-zero sing-reg F p*
shows *rep-list p* ∈ *ideal* (*rep-list* ‘ *F*)
proof −
from *assms* **obtain** *s* **where** (*sig-red sing-reg* (⪯) *F*)** *p s* **and** *rep-list s* = *0* **by** (*rule sig-red-zeroE*)
from *this*(*1*) **have** (*punit.red* (*rep-list* ‘ *F*))** (*rep-list p*) (*rep-list s*) **by** (*rule sig-red-red-rtrancl*)
hence (*punit.red* (*rep-list* ‘ *F*))** (*rep-list p*) *0* **by** (*simp only*: ‹*rep-list s* = *0*›)
thus *?thesis* **by** (*rule punit.red-rtranclp-0-in-pmdl*[*simplified*])
qed

lemma *is-sig-GB-inI*:
  assumes $\bigwedge r.\ lt\ r = u \Longrightarrow r \in dgrad\text{-}sig\text{-}set\ d \Longrightarrow sig\text{-}red\text{-}zero\ (\preceq_t)\ G\ r$
  shows *is-sig-GB-in d G u*
  unfolding *is-sig-GB-in-def* **using** *assms* **by** *blast*

lemma *is-sig-GB-inD*:
  assumes *is-sig-GB-in d G u* **and** $r \in dgrad\text{-}sig\text{-}set\ d$ **and** $lt\ r = u$
  shows *sig-red-zero* $(\preceq_t)$ *G r*
  using *assms* **unfolding** *is-sig-GB-in-def* **by** *blast*

lemma *is-sig-GB-inI-triv*:
  assumes $\neg\ d\ (pp\text{-}of\text{-}term\ u) \le dgrad\text{-}max\ d \lor \neg\ component\text{-}of\text{-}term\ u < length$
fs
  shows *is-sig-GB-in d G u*
proof (*rule is-sig-GB-inI*)
  fix $r{::}'t \Rightarrow_0 {}'b$
  assume $lt\ r = u$ **and** $r \in dgrad\text{-}sig\text{-}set\ d$
  show *sig-red-zero* $(\preceq_t)$ *G r*
  proof (*cases* $r = 0$)
    case *True*
    hence *rep-list* $r = 0$ **by** (*simp only*: *rep-list-zero*)
    with *rtrancl-refl*[*to-pred*] **show** *?thesis* **by** (*rule sig-red-zeroI*)
  next
    case *False*
    from ‹$r \in dgrad\text{-}sig\text{-}set\ d$› **have** $d\ (lp\ r) \le dgrad\text{-}max\ d$ **by** (*rule dgrad-sig-setD-lp*)
    moreover from ‹$r \in dgrad\text{-}sig\text{-}set\ d$› *False* **have** *component-of-term* $(lt\ r) <$
length fs
      by (*rule dgrad-sig-setD-lt*)
    ultimately show *?thesis* **using** *assms* **by** (*simp add*: ‹$lt\ r = u$›)
  qed
qed

lemma *is-sig-GB-in-mono*: $is\text{-}sig\text{-}GB\text{-}in\ d\ G\ u \Longrightarrow G \subseteq G' \Longrightarrow is\text{-}sig\text{-}GB\text{-}in\ d\ G'$
u
  by (*auto simp*: *is-sig-GB-in-def dest*: *sig-red-zero-mono*)

lemma *is-sig-GB-uptI*:
  assumes $G \subseteq dgrad\text{-}sig\text{-}set\ d$
    and $\bigwedge v.\ v \prec_t u \Longrightarrow d\ (pp\text{-}of\text{-}term\ v) \le dgrad\text{-}max\ d \Longrightarrow component\text{-}of\text{-}term$
$v < length\ fs \Longrightarrow$
        is-sig-GB-in d G v
  shows *is-sig-GB-upt d G u*
  unfolding *is-sig-GB-upt-def* **using** *assms* **by** *blast*

lemma *is-sig-GB-uptD1*:
  assumes *is-sig-GB-upt d G u*
  shows $G \subseteq dgrad\text{-}sig\text{-}set\ d$
  using *assms* **unfolding** *is-sig-GB-upt-def* **by** *blast*

lemma *is-sig-GB-uptD2*:
assumes *is-sig-GB-upt d G u* **and** $v \prec_t u$
shows *is-sig-GB-in d G v*
using *assms is-sig-GB-inI-triv* **unfolding** *is-sig-GB-upt-def* **by** *blast*

lemma *is-sig-GB-uptD3*:
assumes *is-sig-GB-upt d G u* **and** $r \in$ *dgrad-sig-set d* **and** *lt r* $\prec_t u$
shows *sig-red-zero* $(\preceq_t)$ *G r*
by (*rule is-sig-GB-inD*, *rule is-sig-GB-uptD2*, *fact+*, *fact refl*)

lemma *is-sig-GB-upt-le*:
assumes *is-sig-GB-upt d G u* **and** $v \preceq_t u$
shows *is-sig-GB-upt d G v*
proof (*rule is-sig-GB-uptI*)
from *assms*(*1*) **show** $G \subseteq$ *dgrad-sig-set d* **by** (*rule is-sig-GB-uptD1*)
next
fix *w*
assume $w \prec_t v$
hence $w \prec_t u$ **using** *assms*(*2*) **by** (*rule ord-term-lin.less-le-trans*)
with *assms*(*1*) **show** *is-sig-GB-in d G w* **by** (*rule is-sig-GB-uptD2*)
qed

lemma *is-sig-GB-upt-mono*:
is-sig-GB-upt d G u $\Longrightarrow G \subseteq G' \Longrightarrow G' \subseteq$ *dgrad-sig-set d* $\Longrightarrow$ *is-sig-GB-upt d* G' *u*
by (*auto simp*: *is-sig-GB-upt-def dest*!: *is-sig-GB-in-mono*)

lemma *is-sig-GB-upt-is-Groebner-basis*:
assumes *dickson-grading d* **and** *hom-grading d* **and** $G \subseteq$ *dgrad-sig-set'* *j d*
and $\bigwedge u$. *component-of-term u* $< j \Longrightarrow$ *is-sig-GB-in d G u*
shows *punit.is-Groebner-basis* (*rep-list* ‘ *G*)
using *assms*(*1*)
proof (*rule punit.weak-GB-is-strong-GB-dgrad-p-set*[*simplified*])
from *assms*(*3*) **have** $G \subseteq$ *dgrad-max-set d* **by** (*simp add*: *dgrad-sig-set'-def*)
with *assms*(*1*) **show** *rep-list* ‘ $G \subseteq$ *punit-dgrad-max-set d* **by** (*rule dgrad-max-3*)
next
fix f::$'a \Rightarrow_0 'b$
assume $f \in$ *punit-dgrad-max-set d*
from *assms*(*3*) **have** *G-sub*: $G \subseteq$ *sig-inv-set'* *j* **by** (*simp add*: *dgrad-sig-set'-def*)
assume $f \in$ *ideal* (*rep-list* ‘ *G*)
also from *rep-list-subset-ideal-sig-inv-set*[*OF G-sub*] **have** ... $\subseteq$ *ideal* (*set* (*take j fs*))
by (*rule ideal.span-subset-spanI*)
finally have $f \in$ *ideal* (*set* (*take j fs*)) **.**
with *assms*(*2*) ‹$f \in$ *punit-dgrad-max-set d*› **obtain** *r* **where** $r \in$ *dgrad-sig-set d*
and $r \in$ *dgrad-sig-set'* *j d* **and** *f*: $f =$ *rep-list r*
by (*rule in-idealE-rep-list-dgrad-sig-set-take*)
from *this*(*2*) **have** $r \in$ *sig-inv-set'* *j* **by** (*simp add*: *dgrad-sig-set'-def*)

show (*punit.red* (*rep-list* ‘ *G*))** *f 0*
proof (*cases r* = *0*)
case *True*
thus *?thesis* **by** (*simp add*: *f rep-list-zero*)
next
case *False*
hence *lt r* ∈ *keys r* **by** (*rule lt-in-keys*)
with ‹*r* ∈ *sig-inv-set′ j*› **have** *component-of-term* (*lt r*) < *j* **by** (*rule sig-inv-setD′*)
hence *is-sig-GB-in d G* (*lt r*) **by** (*rule assms*(*4*))
hence *sig-red-zero* ($\preceq_t$) *G r* **using** ‹*r* ∈ *dgrad-sig-set d*› *refl* **by** (*rule is-sig-GB-inD*)
then obtain *s* **where** (*sig-red* ($\preceq_t$) (⪯) *G*)** *r s* **and** *s*: *rep-list s* = *0* **by** (*rule sig-red-zeroE*)
from *this*(*1*) **have** (*punit.red* (*rep-list* ‘ *G*))** (*rep-list r*) (*rep-list s*)
by (*rule sig-red-red-rtrancl*)
thus *?thesis* **by** (*simp only*: *f s*)
qed
qed

lemma *is-sig-GB-is-Groebner-basis*:
assumes *dickson-grading d* **and** *hom-grading d* **and** *G* ⊆ *dgrad-max-set d* **and** ⋀*u*. *is-sig-GB-in d G u*
shows *punit.is-Groebner-basis* (*rep-list* ‘ *G*)
using *assms*(*1*)
proof (*rule punit.weak-GB-is-strong-GB-dgrad-p-set*[*simplified*])
from *assms*(*1*, *3*) **show** *rep-list* ‘ *G* ⊆ *punit-dgrad-max-set d* **by** (*rule dgrad-max-3*)
next
fix *f*::′*a* $\Rightarrow_0$ ′*b*
assume *f* ∈ *punit-dgrad-max-set d*
assume *f* ∈ *ideal* (*rep-list* ‘ *G*)
also from *rep-list-subset-ideal* **have** ... ⊆ *ideal* (*set fs*) **by** (*rule ideal.span-subset-spanI*)
finally have *f* ∈ *ideal* (*set fs*) **.**
with *assms*(*2*) ‹*f* ∈ *punit-dgrad-max-set d*› **obtain** *r* **where** *r* ∈ *dgrad-sig-set d* **and** *f*: *f* = *rep-list r*
by (*rule in-idealE-rep-list-dgrad-sig-set*)
from *assms*(*4*) *this*(*1*) *refl* **have** *sig-red-zero* ($\preceq_t$) *G r* **by** (*rule is-sig-GB-inD*)
then obtain *s* **where** (*sig-red* ($\preceq_t$) (⪯) *G*)** *r s* **and** *s*: *rep-list s* = *0* **by** (*rule sig-red-zeroE*)
from *this*(*1*) **have** (*punit.red* (*rep-list* ‘ *G*))** (*rep-list r*) (*rep-list s*)
by (*rule sig-red-red-rtrancl*)
thus (*punit.red* (*rep-list* ‘ *G*))** *f 0* **by** (*simp only*: *f s*)
qed

lemma *sig-red-zero-is-red*:
assumes *sig-red-zero sing-reg F r* **and** *rep-list r* ≠ *0*
shows *is-sig-red sing-reg* (⪯) *F r*
proof −
from *assms*(*1*) **obtain** *s* **where** ∗: (*sig-red sing-reg* (⪯) *F*)** *r s* **and** *rep-list s* = *0*
by (*rule sig-red-zeroE*)

from *this*(*2*) *assms*(*2*) **have** $r \neq s$ **by** *auto*
with $*$ **show** *?thesis* **by** (*induct rule*: *converse-rtranclp-induct*, *auto simp*: *is-sig-red-def*)
qed

lemma *is-sig-red-sing-top-is-red-zero*:
assumes *dickson-grading d* **and** *is-sig-GB-upt d G u* **and** $a \in$ *dgrad-sig-set d* **and** *lt* $a = u$
and *is-sig-red* (=) (=) *G a* **and** $\neg$ *is-sig-red* ($\prec_t$) (=) *G a*
shows *sig-red-zero* ($\preceq_t$) *G a*
proof −
from *assms*(*5*) **obtain** *g* **where** $g \in G$ **and** *rep-list g* $\neq$ *0* **and** *rep-list a* $\neq$ *0*
and *1*: *punit.lt* (*rep-list g*) *adds punit.lt* (*rep-list a*)
and *2*: *punit.lt* (*rep-list a*) $\oplus$ *lt g* = *punit.lt* (*rep-list g*) $\oplus$ *lt a*
by (*rule is-sig-red-top-addsE*)
from *this*(*2*, *3*) **have** $g \neq 0$ **and** $a \neq 0$ **by** (*auto simp*: *rep-list-zero*)
hence *lc g* $\neq$ *0* **and** *lc a* $\neq$ *0* **using** *lc-not-0* **by** *blast+*
from *1* **have** *3*: (*punit.lt* (*rep-list a*) − *punit.lt* (*rep-list g*)) $\oplus$ *lt g* = *lt a*
by (*simp add*: *term-is-le-rel-minus 2*)
define g' **where** g' = *monom-mult* (*lc a* / *lc g*) (*punit.lt* (*rep-list a*) − *punit.lt* (*rep-list g*)) *g*
from ‹$g \neq 0$› ‹*lc a* $\neq$ *0*› ‹*lc g* $\neq$ *0*› **have** *lt-g'*: *lt* g' = *lt a* **by** (*simp add*: *g'-def lt-monom-mult 3*)
from ‹*lc g* $\neq$ *0*› **have** *lc-g'*: *lc* g' = *lc a* **by** (*simp add*: *g'-def*)
from *assms*(*1*) **have** $g' \in$ *dgrad-sig-set d* **unfolding** *g'-def*
proof (*rule dgrad-sig-set-closed-monom-mult*)
from *assms*(*1*) *1* **have** *d* (*punit.lt* (*rep-list a*) − *punit.lt* (*rep-list g*)) $\leq$ *d* (*punit.lt* (*rep-list a*))
by (*rule dickson-grading-minus*)
also from *assms*(*1*, *3*) **have** ... $\leq$ *dgrad-max d* **by** (*rule dgrad-sig-setD-rep-list-lt*)
finally show *d* (*punit.lt* (*rep-list a*) − *punit.lt* (*rep-list g*)) $\leq$ *dgrad-max d* **.**
next
from *assms*(*2*) **have** $G \subseteq$ *dgrad-sig-set d* **by** (*rule is-sig-GB-uptD1*)
with ‹$g \in G$› **show** $g \in$ *dgrad-sig-set d* **..**
qed
with *assms*(*3*) **have** *b-in*: $a - g' \in$ *dgrad-sig-set d* (**is** *?b* $\in$ -)
by (*rule dgrad-sig-set-closed-minus*)
from *1* **have** *4*: *punit.lt* (*rep-list a*) − *punit.lt* (*rep-list g*) + *punit.lt* (*rep-list g*) =
punit.lt (*rep-list a*)
by (*rule adds-minus*)

show *?thesis*
proof (*cases lc a* / *lc g* = *punit.lc* (*rep-list a*) / *punit.lc* (*rep-list g*))
case *True*
have *sig-red-single* (=) (=) *a ?b g* (*punit.lt* (*rep-list a*) − *punit.lt* (*rep-list g*))
proof (*rule sig-red-singleI*)
show *punit.lt* (*rep-list a*) − *punit.lt* (*rep-list g*) + *punit.lt* (*rep-list g*) $\in$ *keys* (*rep-list a*)
unfolding *4* **using** ‹*rep-list a* $\neq$ *0*› **by** (*rule punit.lt-in-keys*)

next
show *?b* =
a − *monom-mult*
(*lookup* (*rep-list a*) (*punit.lt* (*rep-list a*) − *punit.lt* (*rep-list g*) + *punit.lt* (*rep-list g*)) /
punit.lc (*rep-list g*))
(*punit.lt* (*rep-list a*) − *punit.lt* (*rep-list g*)) *g*
by (*simp add*: *g′-def 4 punit.lc-def True*)
qed (*simp-all add*: *3 4* ‹*rep-list g* ≠ *0*›)
hence *sig-red* (=) (=) *G a ?b* **unfolding** *sig-red-def* **using** ‹*g* ∈ *G*› **by** *blast*
hence *sig-red* ($\preceq_t$) (⪯) *G a ?b* **by** (*auto dest*: *sig-red-sing-regI sig-red-top-tailI*)
hence *5*: (*sig-red* ($\preceq_t$) (⪯) *G*)** *a ?b* **..**
show *?thesis*
proof (*cases ?b* = *0*)
case *True*
hence *rep-list ?b* = *0* **by** (*simp only*: *rep-list-zero*)
with *5* **show** *?thesis* **by** (*rule sig-red-zeroI*)
next
case *False*
hence *lt ?b* $\prec_t$ *lt a* **using** *lt-g′ lc-g′* **by** (*rule lt-minus-lessI*)
hence *lt ?b* $\prec_t$ *u* **by** (*simp only*: *assms*(*4*))
with *assms*(*2*) *b-in* **have** *sig-red-zero* ($\preceq_t$) *G ?b* **by** (*rule is-sig-GB-uptD3*)
then obtain *s* **where** (*sig-red* ($\preceq_t$) (⪯) *G*)** *?b s* **and** *rep-list s* = *0* **by** (*rule sig-red-zeroE*)
from *5 this*(*1*) **have** (*sig-red* ($\preceq_t$) (⪯) *G*)** *a s* **by** (*rule rtranclp-trans*)
thus *?thesis* **using** ‹*rep-list s* = *0*› **by** (*rule sig-red-zeroI*)
qed
next
case *False*
from ‹*rep-list g* ≠ *0*› ‹*lc g* ≠ *0*› ‹*lc a* ≠ *0*› **have** *5*: *punit.lt* (*rep-list g′*) = *punit.lt* (*rep-list a*)
by (*simp add*: *g′-def rep-list-monom-mult punit.lt-monom-mult 4*)
have *6*: *punit.lc* (*rep-list g′*) = (*lc a* / *lc g*) ∗ *punit.lc* (*rep-list g*)
by (*simp add*: *g′-def rep-list-monom-mult*)
also have *7*: ... ≠ *punit.lc* (*rep-list a*)
proof
assume *lc a* / *lc g* ∗ *punit.lc* (*rep-list g*) = *punit.lc* (*rep-list a*)
moreover from ‹*rep-list g* ≠ *0*› **have** *punit.lc* (*rep-list g*) ≠ *0* **by** (*rule punit.lc-not-0*)
ultimately have *lc a* / *lc g* = *punit.lc* (*rep-list a*) / *punit.lc* (*rep-list g*)
by (*simp add*: *field-simps*)
with *False* **show** *False* **..**
qed
finally have *punit.lc* (*rep-list g′*) ≠ *punit.lc* (*rep-list a*) **.**
with *5* **have** *8*: *punit.lt* (*rep-list ?b*) = *punit.lt* (*rep-list a*) **unfolding** *rep-list-minus*
by (*rule punit.lt-minus-eqI-3*)
hence *punit.lc* (*rep-list ?b*) = *punit.lc* (*rep-list a*) − (*lc a* / *lc g*) ∗ *punit.lc* (*rep-list g*)
unfolding *6*[*symmetric*] **by** (*simp only*: *punit.lc-def lookup-minus rep-list-minus*

5)
  also have ... $\neq$ *0*
  proof
    assume *punit.lc* (*rep-list a*) $-$ *lc a* / *lc g* $*$ *punit.lc* (*rep-list g*) = *0*
    hence *lc a* / *lc g* $*$ *punit.lc* (*rep-list g*) = *punit.lc* (*rep-list a*) **by** *simp*
    with *7* **show** *False* **..**
  qed
  finally have *rep-list ?b* $\neq$ *0* **by** (*simp add*: *punit.lc-eq-zero-iff*)
  hence *?b* $\neq$ *0* **by** (*auto simp*: *rep-list-zero*)
  hence *lt ?b* $\prec_t$ *lt a* **using** *lt-g′ lc-g′* **by** (*rule lt-minus-lessI*)
  hence *lt ?b* $\prec_t$ *u* **by** (*simp only*: *assms*(*4*))
  with *assms*(*2*) *b-in* **have** *sig-red-zero* ($\preceq_t$) *G ?b* **by** (*rule is-sig-GB-uptD3*)
  moreover note ‹*rep-list ?b* $\neq$ *0*›
  moreover have ($\preceq_t$) = ($\preceq_t$) $\vee$ ($\preceq_t$) = ($\prec_t$) **by** *simp*
  ultimately have *is-sig-red* ($\preceq_t$) (=) *G ?b* **by** (*rule sig-red-zero-nonzero*)
  then obtain *g0* **where** *g0* $\in$ *G* **and** *rep-list g0* $\neq$ *0*
    and *9*: *punit.lt* (*rep-list g0*) *adds punit.lt* (*rep-list ?b*)
    and *10*: *punit.lt* (*rep-list ?b*) $\oplus$ *lt g0* $\preceq_t$ *punit.lt* (*rep-list g0*) $\oplus$ *lt ?b*
    by (*rule is-sig-red-top-addsE*)
  from *9* **have** *punit.lt* (*rep-list g0*) *adds punit.lt* (*rep-list a*) **by** (*simp only*: *8*)
  from *10* **have** *punit.lt* (*rep-list a*) $\oplus$ *lt g0* $\preceq_t$ *punit.lt* (*rep-list g0*) $\oplus$ *lt ?b* **by** (*simp only*: *8*)
   also from ‹*lt ?b* $\prec_t$ *lt a*› **have** ... $\prec_t$ *punit.lt* (*rep-list g0*) $\oplus$ *lt a* **by** (*rule splus-mono-strict*)
  finally have *punit.lt* (*rep-list a*) $\oplus$ *lt g0* $\prec_t$ *punit.lt* (*rep-list g0*) $\oplus$ *lt a* **.**
  have *is-sig-red* ($\prec_t$) (=) *G a*
  proof (*rule is-sig-red-top-addsI*)
    show *ord-term-lin.is-le-rel* ($\prec_t$) **by** *simp*
  qed *fact*+
  with *assms*(*6*) **show** *?thesis* **..**
 qed
qed

lemma *sig-regular-reduced-unique*:
 assumes *is-sig-GB-upt d G* (*lt q*) **and** *p* $\in$ *dgrad-sig-set d* **and** *q* $\in$ *dgrad-sig-set d*
  and *lt p* = *lt q* **and** *lc p* = *lc q* **and** $\neg$ *is-sig-red* ($\prec_t$) ($\preceq$) *G p* **and** $\neg$ *is-sig-red* ($\prec_t$) ($\preceq$) *G q*
 shows *rep-list p* = *rep-list q*
proof (*rule ccontr*)
 assume *rep-list p* $\neq$ *rep-list q*
 hence *rep-list* (*p* $-$ *q*) $\neq$ *0* **by** (*auto simp*: *rep-list-minus*)
 hence *p* $-$ *q* $\neq$ *0* **by** (*auto simp*: *rep-list-zero*)
 hence *p* + ($-$ *q*) $\neq$ *0* **by** *simp*
 moreover from *assms*(*4*) **have** *lt* ($-$ *q*) = *lt p* **by** *simp*
 moreover from *assms*(*5*) **have** *lc* ($-$ *q*) = $-$ *lc p* **by** *simp*
 ultimately have *lt* (*p* + ($-$ *q*)) $\prec_t$ *lt p* **by** (*rule lt-plus-lessI*)
 hence *lt* (*p* $-$ *q*) $\prec_t$ *lt q* **using** *assms*(*4*) **by** *simp*
 with *assms*(*1*) **have** *is-sig-GB-in d G* (*lt* (*p* $-$ *q*)) **by** (*rule is-sig-GB-uptD2*)

moreover from *assms*(*2*, *3*) **have** $p - q \in$ *dgrad-sig-set d* **by** (*rule dgrad-sig-set-closed-minus*)
ultimately have *sig-red-zero* ($\preceq_t$) *G* ($p - q$) **using** *refl* **by** (*rule is-sig-GB-inD*)
hence *is-sig-red* ($\preceq_t$) ($\preceq$) *G* ($p - q$) **using** ‹*rep-list* ($p - q$) $\neq$ *0*› **by** (*rule sig-red-zero-is-red*)
then obtain *g t* **where** $g \in G$ **and** *t*: $t \in$ *keys* (*rep-list* ($p - q$)) **and** *rep-list g* $\neq$ *0*
and *adds*: *punit.lt* (*rep-list g*) *adds t* **and** $t \oplus$ *lt g* $\preceq_t$ *punit.lt* (*rep-list g*) $\oplus$ *lt* ($p - q$)
by (*rule is-sig-red-addsE*)
note *this*(*5*)
also from ‹*lt* ($p - q$) $\prec_t$ *lt q*› **have** *punit.lt* (*rep-list g*) $\oplus$ *lt* ($p - q$) $\prec_t$ *punit.lt* (*rep-list g*) $\oplus$ *lt q*
by (*rule splus-mono-strict*)
finally have *1*: $t \oplus$ *lt g* $\prec_t$ *punit.lt* (*rep-list g*) $\oplus$ *lt q* **.**
hence *2*: $t \oplus$ *lt g* $\prec_t$ *punit.lt* (*rep-list g*) $\oplus$ *lt p* **by** (*simp only*: *assms*(*4*))
from *t keys-minus* **have** $t \in$ *keys* (*rep-list p*) $\cup$ *keys* (*rep-list q*) **unfolding** *rep-list-minus* **..**
thus *False*
proof
assume *t-in*: $t \in$ *keys* (*rep-list p*)
hence $t \preceq$ *punit.lt* (*rep-list p*) **by** (*rule punit.lt-max-keys*)
with ‹$g \in G$› *t-in* ‹*rep-list g* $\neq$ *0*› *adds ord-term-lin.is-le-relI*(*3*) *ordered-powerprod-lin.is-le-relI*(*2*) *2*
have *is-sig-red* ($\prec_t$) ($\preceq$) *G p* **by** (*rule is-sig-red-addsI*)
with *assms*(*6*) **show** *False* **..**
next
assume *t-in*: $t \in$ *keys* (*rep-list q*)
hence $t \preceq$ *punit.lt* (*rep-list q*) **by** (*rule punit.lt-max-keys*)
with ‹$g \in G$› *t-in* ‹*rep-list g* $\neq$ *0*› *adds ord-term-lin.is-le-relI*(*3*) *ordered-powerprod-lin.is-le-relI*(*2*) *1*
have *is-sig-red* ($\prec_t$) ($\preceq$) *G q* **by** (*rule is-sig-red-addsI*)
with *assms*(*7*) **show** *False* **..**
qed
qed

corollary *sig-regular-reduced-unique'*:
assumes *is-sig-GB-upt d G* (*lt q*) **and** $p \in$ *dgrad-sig-set d* **and** $q \in$ *dgrad-sig-set d*
and *lt p* = *lt q* **and** $\neg$ *is-sig-red* ($\prec_t$) ($\preceq$) *G p* **and** $\neg$ *is-sig-red* ($\prec_t$) ($\preceq$) *G q*
shows *punit.monom-mult* (*lc q*) *0* (*rep-list p*) = *punit.monom-mult* (*lc p*) *0* (*rep-list q*)
proof (*cases* $p = 0 \vee q = 0$)
case *True*
thus *?thesis* **by** (*auto simp*: *rep-list-zero*)
next
case *False*
hence $p \neq 0$ **and** $q \neq 0$ **by** *simp-all*
hence *lc p* $\neq$ *0* **and** *lc q* $\neq$ *0* **by** (*simp-all add*: *lc-not-0*)
let *?p* = *monom-mult* (*lc q*) *0 p*

let *?q* = *monom-mult* (*lc p*) *0 q*
have *lt ?q* = *lt q* **by** (*simp add*: *lt-monom-mult*[*OF* ‹*lc p* ≠ *0*› ‹*q* ≠ *0*›] *splus-zero*)
with *assms*(*1*) **have** *is-sig-GB-upt d G* (*lt ?q*) **by** *simp*
moreover from *assms*(*2*) **have** *?p* ∈ *dgrad-sig-set d* **by** (*rule dgrad-sig-set-closed-monom-mult-zero*)
moreover from *assms*(*3*) **have** *?q* ∈ *dgrad-sig-set d* **by** (*rule dgrad-sig-set-closed-monom-mult-zero*)
moreover from ‹*lt ?q* = *lt q*› **have** *lt ?p* = *lt ?q*
by (*simp add*: *lt-monom-mult*[*OF* ‹*lc q* ≠ *0*› ‹*p* ≠ *0*›] *splus-zero assms*(*4*))
moreover have *lc ?p* = *lc ?q* **by** *simp*
moreover have ¬ *is-sig-red* ($\prec_t$) (⪯) *G ?p*
proof
assume *is-sig-red* ($\prec_t$) (⪯) *G ?p*
moreover from ‹*lc q* ≠ *0*› **have** *1* / (*lc q*) ≠ *0* **by** *simp*
ultimately have *is-sig-red* ($\prec_t$) (⪯) *G* (*monom-mult* (*1* / *lc q*) *0 ?p*) **by** (*rule is-sig-red-monom-mult*)
hence *is-sig-red* ($\prec_t$) (⪯) *G p* **by** (*simp add*: *monom-mult-assoc* ‹*lc q* ≠ *0*›)
with *assms*(*5*) **show** *False* **..**
qed
moreover have ¬ *is-sig-red* ($\prec_t$) (⪯) *G ?q*
proof
assume *is-sig-red* ($\prec_t$) (⪯) *G ?q*
moreover from ‹*lc p* ≠ *0*› **have** *1* / (*lc p*) ≠ *0* **by** *simp*
ultimately have *is-sig-red* ($\prec_t$) (⪯) *G* (*monom-mult* (*1* / *lc p*) *0 ?q*) **by** (*rule is-sig-red-monom-mult*)
hence *is-sig-red* ($\prec_t$) (⪯) *G q* **by** (*simp add*: *monom-mult-assoc* ‹*lc p* ≠ *0*›)
with *assms*(*6*) **show** *False* **..**
qed
ultimately have *rep-list ?p* = *rep-list ?q* **by** (*rule sig-regular-reduced-unique*)
thus *?thesis* **by** (*simp only*: *rep-list-monom-mult*)
qed

lemma *sig-regular-top-reduced-lt-lc-unique*:
assumes *dickson-grading d* **and** *is-sig-GB-upt d G* (*lt q*) **and** *p* ∈ *dgrad-sig-set d* **and** *q* ∈ *dgrad-sig-set d*
and *lt p* = *lt q* **and** (*p* = *0*) ⟷ (*q* = *0*) **and** ¬ *is-sig-red* ($\prec_t$) (=) *G p* **and** ¬ *is-sig-red* ($\prec_t$) (=) *G q*
shows *punit.lt* (*rep-list p*) = *punit.lt* (*rep-list q*) ∧ *lc q* ∗ *punit.lc* (*rep-list p*) = *lc p* ∗ *punit.lc* (*rep-list q*)
proof (*cases p* = *0*)
case *True*
with *assms*(*6*) **have** *q* = *0* **by** *simp*
thus *?thesis* **by** (*simp add*: *True*)
next
case *False*
with *assms*(*6*) **have** *q* ≠ *0* **by** *simp*
from *False* **have** *lc p* ≠ *0* **by** (*rule lc-not-0*)
from ‹*q* ≠ *0*› **have** *lc q* ≠ *0* **by** (*rule lc-not-0*)
from *assms*(*2*) **have** *G-sub*: *G* ⊆ *dgrad-sig-set d* **by** (*rule is-sig-GB-uptD1*)
hence *G* ⊆ *dgrad-max-set d* **by** (*simp add*: *dgrad-sig-set′-def*)
with *assms*(*1*) **obtain** *p′* **where** *p′-red*: (*sig-red* ($\prec_t$) (≺) *G*)** *p p′* **and** ¬

is-sig-red $(\prec_t)$ $(\prec)$ G p'
  by (*rule sig-irredE-dgrad-max-set*)
 from *this*(*1*) **have** *lt-p'*: *lt* $p' =$ *lt* p **and** *lt-p''*: *punit.lt* (*rep-list* p') = *punit.lt* (*rep-list* p)
  and *lc-p'*: *lc* $p' =$ *lc* p **and** *lc-p''*: *punit.lc* (*rep-list* p') = *punit.lc* (*rep-list* p)
  by (*rule sig-red-regular-rtrancl-lt*, *rule sig-red-tail-rtrancl-lt-rep-list*,
   rule sig-red-regular-rtrancl-lc, *rule sig-red-tail-rtrancl-lc-rep-list*)
 have $\neg$ *is-sig-red* $(\prec_t)$ $(=)$ G p'
 proof
  assume *a*: *is-sig-red* $(\prec_t)$ $(=)$ G p'
  hence *rep-list* $p' \neq 0$ **using** *is-sig-red-top-addsE* **by** *blast*
  hence *rep-list* $p \neq 0$ **using** ‹(*sig-red* $(\prec_t)$ $(\prec)$ $G)^{**}$ p p'›
   by (*auto simp*: *punit.rtrancl-0 dest*!: *sig-red-red-rtrancl*)
  with *a* **have** *is-sig-red* $(\prec_t)$ $(=)$ G p **using** *lt-p'* *lt-p''* **by** (*rule is-sig-red-top-cong*)
  with *assms*(*7*) **show** *False* **..**
 qed
 with ‹$\neg$ *is-sig-red* $(\prec_t)$ $(\prec)$ G p'› **have** *1*: $\neg$ *is-sig-red* $(\prec_t)$ $(\preceq)$ G p' **by** (*simp add*: *is-sig-red-top-tail-cases*)
 from *assms*(*1*) ‹$G \subseteq$ *dgrad-max-set* d› **obtain** q' **where** *q'-red*: (*sig-red* $(\prec_t)$ $(\prec)$ $G)^{**}$ q q'
  and $\neg$ *is-sig-red* $(\prec_t)$ $(\prec)$ G q' **by** (*rule sig-irredE-dgrad-max-set*)
 from *this*(*1*) **have** *lt-q'*: *lt* $q' =$ *lt* q **and** *lt-q''*: *punit.lt* (*rep-list* q') = *punit.lt* (*rep-list* q)
  and *lc-q'*: *lc* $q' =$ *lc* q **and** *lc-q''*: *punit.lc* (*rep-list* q') = *punit.lc* (*rep-list* q)
  by (*rule sig-red-regular-rtrancl-lt*, *rule sig-red-tail-rtrancl-lt-rep-list*,
   rule sig-red-regular-rtrancl-lc, *rule sig-red-tail-rtrancl-lc-rep-list*)
 have $\neg$ *is-sig-red* $(\prec_t)$ $(=)$ G q'
 proof
  assume *a*: *is-sig-red* $(\prec_t)$ $(=)$ G q'
  hence *rep-list* $q' \neq 0$ **using** *is-sig-red-top-addsE* **by** *blast*
  hence *rep-list* $q \neq 0$ **using** ‹(*sig-red* $(\prec_t)$ $(\prec)$ $G)^{**}$ q q'›
   by (*auto simp*: *punit.rtrancl-0 dest*!: *sig-red-red-rtrancl*)
  with *a* **have** *is-sig-red* $(\prec_t)$ $(=)$ G q **using** *lt-q'* *lt-q''* **by** (*rule is-sig-red-top-cong*)
  with *assms*(*8*) **show** *False* **..**
 qed
 with ‹$\neg$ *is-sig-red* $(\prec_t)$ $(\prec)$ G q'› **have** *2*: $\neg$ *is-sig-red* $(\prec_t)$ $(\preceq)$ G q' **by** (*simp add*: *is-sig-red-top-tail-cases*)
 from *assms*(*2*) **have** *is-sig-GB-upt* d G (*lt* q') **by** (*simp only*: *lt-q'*)
 moreover from *assms*(*1*) *G-sub* *assms*(*3*) *p'-red* **have** $p' \in$ *dgrad-sig-set* d
  by (*rule dgrad-sig-set-closed-sig-red-rtrancl*)
 moreover from *assms*(*1*) *G-sub* *assms*(*4*) *q'-red* **have** $q' \in$ *dgrad-sig-set* d
  by (*rule dgrad-sig-set-closed-sig-red-rtrancl*)
 moreover have *lt* $p' =$ *lt* q' **by** (*simp only*: *lt-p'* *lt-q'* *assms*(*5*))
 ultimately have *eq*: *punit.monom-mult* (*lc* q') *0* (*rep-list* p') = *punit.monom-mult* (*lc* p') *0* (*rep-list* q')
  using *1 2* **by** (*rule sig-regular-reduced-unique'*)

 have *lc* q $*$ *punit.lc* (*rep-list* p) = *lc* q $*$ *punit.lc* (*rep-list* p') **by** (*simp only*: *lc-p''*)

also from ‹*lc q* ≠ *0*› **have** ... = *punit.lc* (*punit.monom-mult* (*lc q'*) *0* (*rep-list p'*))
by (*simp add*: *lc-q'*)
also have ... = *punit.lc* (*punit.monom-mult* (*lc p'*) *0* (*rep-list q'*)) **by** (*simp only*: *eq*)
also from ‹*lc p* ≠ *0*› **have** ... = *lc p* ∗ *punit.lc* (*rep-list q'*) **by** (*simp add*: *lc-p'*)
also have ... = *lc p* ∗ *punit.lc* (*rep-list q*) **by** (*simp only*: *lc-q''*)
finally have ∗: *lc q* ∗ *punit.lc* (*rep-list p*) = *lc p* ∗ *punit.lc* (*rep-list q*) **.**

have *punit.lt* (*rep-list p*) = *punit.lt* (*rep-list p'*) **by** (*simp only*: *lt-p''*)
also from ‹*lc q* ≠ *0*› **have** ... = *punit.lt* (*punit.monom-mult* (*lc q'*) *0* (*rep-list p'*))
by (*simp add*: *lc-q' punit.lt-monom-mult-zero*)
also have ... = *punit.lt* (*punit.monom-mult* (*lc p'*) *0* (*rep-list q'*)) **by** (*simp only*: *eq*)
also from ‹*lc p* ≠ *0*› **have** ... = *punit.lt* (*rep-list q'*) **by** (*simp add*: *lc-p' punit.lt-monom-mult-zero*)
also have ... = *punit.lt* (*rep-list q*) **by** (*fact lt-q''*)
finally show *?thesis* **using** ∗ **..**
qed

corollary *sig-regular-top-reduced-lt-unique*:
assumes *dickson-grading d* **and** *is-sig-GB-upt d G* (*lt q*) **and** *p* ∈ *dgrad-sig-set d*
and *q* ∈ *dgrad-sig-set d* **and** *lt p* = *lt q* **and** *p* ≠ *0* **and** *q* ≠ *0*
and ¬ *is-sig-red* ($\prec_t$) (=) *G p* **and** ¬ *is-sig-red* ($\prec_t$) (=) *G q*
shows *punit.lt* (*rep-list p*) = *punit.lt* (*rep-list q*)
proof −
from *assms*(*6*, *7*) **have** (*p* = *0*) ⟷ (*q* = *0*) **by** *simp*
with *assms*(*1*, *2*, *3*, *4*, *5*)
have *punit.lt* (*rep-list p*) = *punit.lt* (*rep-list q*) ∧ *lc q* ∗ *punit.lc* (*rep-list p*) = *lc p* ∗ *punit.lc* (*rep-list q*)
using *assms*(*8*, *9*) **by** (*rule sig-regular-top-reduced-lt-lc-unique*)
thus *?thesis* **..**
qed

corollary *sig-regular-top-reduced-lc-unique*:
assumes *dickson-grading d* **and** *is-sig-GB-upt d G* (*lt q*) **and** *p* ∈ *dgrad-sig-set d* **and** *q* ∈ *dgrad-sig-set d*
and *lt p* = *lt q* **and** *lc p* = *lc q* **and** ¬ *is-sig-red* ($\prec_t$) (=) *G p* **and** ¬ *is-sig-red* ($\prec_t$) (=) *G q*
shows *punit.lc* (*rep-list p*) = *punit.lc* (*rep-list q*)
proof (*cases p* = *0*)
case *True*
with *assms*(*6*) **have** *q* = *0* **by** (*simp add*: *lc-eq-zero-iff*)
with *True* **show** *?thesis* **by** *simp*
next
case *False*
hence *lc p* ≠ *0* **by** (*rule lc-not-0*)

hence *lc q* $\neq$ *0* **by** (*simp add*: *assms*(*6*))
hence *q* $\neq$ *0* **by** (*simp add*: *lc-eq-zero-iff*)
with *False* **have** (*p* = *0*) $\longleftrightarrow$ (*q* = *0*) **by** *simp*
with *assms*(*1*, *2*, *3*, *4*, *5*)
have *punit.lt* (*rep-list p*) = *punit.lt* (*rep-list q*) $\wedge$ *lc q* $*$ *punit.lc* (*rep-list p*) = *lc p* $*$ *punit.lc* (*rep-list q*)
using *assms*(*7*, *8*) **by** (*rule sig-regular-top-reduced-lt-lc-unique*)
hence *lc q* $*$ *punit.lc* (*rep-list p*) = *lc p* $*$ *punit.lc* (*rep-list q*) **..**
also have ... = *lc q* $*$ *punit.lc* (*rep-list q*) **by** (*simp only*: *assms*(*6*))
finally show *?thesis* **using** ‹*lc q* $\neq$ *0*› **by** *simp*
qed

Minimal signature Gröbner bases are indeed minimal, at least up to sig-lead-pairs:

lemma *is-min-sig-GB-minimal*:
assumes *is-min-sig-GB d G* **and** $G' \subseteq$ *dgrad-sig-set d*
and $\bigwedge$*u*. *d* (*pp-of-term u*) $\leq$ *dgrad-max d* $\Longrightarrow$ *component-of-term u* < *length fs* $\Longrightarrow$ *is-sig-GB-in d G' u*
and *g* $\in$ *G* **and** *rep-list g* $\neq$ *0*
obtains *g'* **where** *g'* $\in$ *G'* **and** *rep-list g'* $\neq$ *0* **and** *lt g'* = *lt g*
and *punit.lt* (*rep-list g'*) = *punit.lt* (*rep-list g*)
proof −
from *assms*(*1*) **have** *G* $\subseteq$ *dgrad-sig-set d*
and *1*: $\bigwedge$*u*. *d* (*pp-of-term u*) $\leq$ *dgrad-max d* $\Longrightarrow$ *component-of-term u* < *length fs* $\Longrightarrow$ *is-sig-GB-in d G u*
and *2*: $\bigwedge$*g0*. *g0* $\in$ *G* $\Longrightarrow$ $\neg$ *is-sig-red* ($\preceq_t$) (=) (*G* − {*g0*}) *g0*
by (*simp-all add*: *is-min-sig-GB-def*)
from *assms*(*4*) **have** *3*: $\neg$ *is-sig-red* ($\preceq_t$) (=) (*G* − {*g*}) *g* **by** (*rule 2*)

from *assms*(*5*) **have** *g* $\neq$ *0* **by** (*auto simp*: *rep-list-zero*)
from *assms*(*4*) ‹*G* $\subseteq$ *dgrad-sig-set d*› **have** *g* $\in$ *dgrad-sig-set d* **..**
hence *d* (*lp g*) $\leq$ *dgrad-max d* **and** *component-of-term* (*lt g*) < *length fs*
by (*rule dgrad-sig-setD-lp*, *rule dgrad-sig-setD-lt*[*OF* - ‹*g* $\neq$ *0*›])
hence *is-sig-GB-in d G'* (*lt g*) **by** (*rule assms*(*3*))
hence *sig-red-zero* ($\preceq_t$) *G' g* **using** ‹*g* $\in$ *dgrad-sig-set d*› *refl* **by** (*rule is-sig-GB-inD*)
moreover note *assms*(*5*)
moreover have ($\preceq_t$) = ($\preceq_t$) $\vee$ ($\preceq_t$) = ($\prec_t$) **by** *simp*
ultimately have *is-sig-red* ($\preceq_t$) (=) *G' g* **by** (*rule sig-red-zero-nonzero*)
then obtain *g'* **where** *g'* $\in$ *G'* **and** *rep-list g'* $\neq$ *0*
and *adds1*: *punit.lt* (*rep-list g'*) *adds punit.lt* (*rep-list g*)
and *le1*: *punit.lt* (*rep-list g*) $\oplus$ *lt g'* $\preceq_t$ *punit.lt* (*rep-list g'*) $\oplus$ *lt g*
by (*rule is-sig-red-top-addsE*)

from ‹*rep-list g'* $\neq$ *0*› **have** *g'* $\neq$ *0* **by** (*auto simp*: *rep-list-zero*)
from ‹*g'* $\in$ *G'*› *assms*(*2*) **have** *g'* $\in$ *dgrad-sig-set d* **..**
hence *d* (*lp g'*) $\leq$ *dgrad-max d* **and** *component-of-term* (*lt g'*) < *length fs*
by (*rule dgrad-sig-setD-lp*, *rule dgrad-sig-setD-lt*[*OF* - ‹*g'* $\neq$ *0*›])
hence *is-sig-GB-in d G* (*lt g'*) **by** (*rule 1*)
hence *sig-red-zero* ($\preceq_t$) *G g'* **using** ‹*g'* $\in$ *dgrad-sig-set d*› *refl* **by** (*rule is-sig-GB-inD*)

moreover note ‹*rep-list* $g' \neq 0$›
moreover have $(\preceq_t) = (\preceq_t) \vee (\preceq_t) = (\prec_t)$ **by** *simp*
ultimately have *is-sig-red* $(\preceq_t)$ $(=)$ G g' **by** (*rule sig-red-zero-nonzero*)
then obtain *g0* **where** $g0 \in G$ **and** *rep-list* $g0 \neq 0$
and *adds2*: *punit.lt* (*rep-list g0*) *adds punit.lt* (*rep-list* g')
and *le2*: *punit.lt* (*rep-list* g') $\oplus$ *lt g0* $\preceq_t$ *punit.lt* (*rep-list g0*) $\oplus$ *lt* g'
by (*rule is-sig-red-top-addsE*)

have *eq1*: $g0 = g$
proof (*rule ccontr*)
assume $g0 \neq g$
with ‹$g0 \in G$› **have** $g0 \in G - \{g\}$ **by** *simp*
moreover note ‹*rep-list* $g0 \neq 0$› *assms*(*5*)
moreover from *adds2 adds1* **have** *punit.lt* (*rep-list g0*) *adds punit.lt* (*rep-list g*)
by (*rule adds-trans*)
moreover have *ord-term-lin.is-le-rel* $(\preceq_t)$ **by** *simp*
moreover have *punit.lt* (*rep-list g*) $\oplus$ *lt g0* $\preceq_t$ *punit.lt* (*rep-list g0*) $\oplus$ *lt g*
proof (*rule ord-term-canc*)
have *punit.lt* (*rep-list* g') $\oplus$ (*punit.lt* (*rep-list g*) $\oplus$ *lt g0*) =
punit.lt (*rep-list g*) $\oplus$ (*punit.lt* (*rep-list* g') $\oplus$ *lt g0*) **by** (*fact splus-left-commute*)
also from *le2* **have** ... $\preceq_t$ *punit.lt* (*rep-list g*) $\oplus$ (*punit.lt* (*rep-list g0*) $\oplus$ *lt* g')
by (*rule splus-mono*)
also have ... = *punit.lt* (*rep-list g0*) $\oplus$ (*punit.lt* (*rep-list g*) $\oplus$ *lt* g')
by (*fact splus-left-commute*)
also from *le1* **have** ... $\preceq_t$ *punit.lt* (*rep-list g0*) $\oplus$ (*punit.lt* (*rep-list* g') $\oplus$ *lt g*)
by (*rule splus-mono*)
also have ... = *punit.lt* (*rep-list* g') $\oplus$ (*punit.lt* (*rep-list g0*) $\oplus$ *lt g*)
by (*fact splus-left-commute*)
finally show *punit.lt* (*rep-list* g') $\oplus$ (*punit.lt* (*rep-list g*) $\oplus$ *lt g0*) $\preceq_t$
punit.lt (*rep-list* g') $\oplus$ (*punit.lt* (*rep-list g0*) $\oplus$ *lt g*) **.**
qed
ultimately have *is-sig-red* $(\preceq_t)$ $(=)$ $(G - \{g\})$ *g* **by** (*rule is-sig-red-top-addsI*)
with *3* **show** *False* **..**
qed

from *adds2 adds1* **have** *eq2*: *punit.lt* (*rep-list* g') = *punit.lt* (*rep-list g*) **by** (*simp add*: *eq1 adds-antisym*)
with *le1 le2* **have** *punit.lt* (*rep-list g*) $\oplus$ *lt* g' = *punit.lt* (*rep-list g*) $\oplus$ *lt g* **by** (*simp add*: *eq1*)
hence *lt* g' = *lt g* **by** (*simp only*: *splus-left-canc*)
with ‹$g' \in G'$› ‹*rep-list* $g' \neq 0$› **show** *?thesis* **using** *eq2* **..**
qed

lemma *sig-red-zero-regularI-adds*:
assumes *dickson-grading d* **and** *is-sig-GB-upt d G* (*lt q*)
and $p \in$ *dgrad-sig-set d* **and** $q \in$ *dgrad-sig-set d* **and** $p \neq 0$ **and** *sig-red-zero* $(\prec_t)$ *G p*

and *lt p adds$_t$ lt q*
shows *sig-red-zero* ($\prec_t$) *G q*
proof (*cases q* = *0*)
case *True*
hence *rep-list q* = *0* **by** (*simp only*: *rep-list-zero*)
with *rtrancl-refl*[*to-pred*] **show** *?thesis* **by** (*rule sig-red-zeroI*)
next
case *False*
hence *lc q* $\neq$ *0* **by** (*rule lc-not-0*)
moreover from *assms*(*5*) **have** *lc p* $\neq$ *0* **by** (*rule lc-not-0*)
ultimately have *lc q* / *lc p* $\neq$ *0* **by** *simp*
from *assms*(*7*) **have** *eq1*: (*lp q* $-$ *lp p*) $\oplus$ *lt p* = *lt q*
by (*metis add-diff-cancel-right′ adds-termE pp-of-term-splus*)
from *assms*(*7*) **have** *lp p adds lp q* **by** (*simp add*: *adds-term-def*)
with *assms*(*1*) **have** *d* (*lp q* $-$ *lp p*) $\leq$ *d* (*lp q*) **by** (*rule dickson-grading-minus*)
also from *assms*(*4*) **have** ... $\leq$ *dgrad-max d* **by** (*rule dgrad-sig-setD-lp*)
finally have *d* (*lp q* $-$ *lp p*) $\leq$ *dgrad-max d* **.**
from *assms*(*2*) **have** *G-sub*: *G* $\subseteq$ *dgrad-sig-set d* **by** (*rule is-sig-GB-uptD1*)
hence *G* $\subseteq$ *dgrad-max-set d* **by** (*simp add*: *dgrad-sig-set′-def*)

let *?mult* = λr. *monom-mult* (*lc q* / *lc p*) (*lp q* $-$ *lp p*) *r*
from *assms*(*6*) **obtain** *p′* **where** *p-red*: (*sig-red* ($\prec_t$) ($\preceq$) *G*)** *p p′* **and** *rep-list p′* = *0*
by (*rule sig-red-zeroE*)
from *p-red* **have** *lt p′* = *lt p* **and** *lc p′* = *lc p*
by (*rule sig-red-regular-rtrancl-lt*, *rule sig-red-regular-rtrancl-lc*)
hence *p′* $\neq$ *0* **using** ‹*lc p* $\neq$ *0*› **by** *auto*
with ‹*lc q* / *lc p* $\neq$ *0*› **have** *?mult p′* $\neq$ *0* **by** (*simp add*: *monom-mult-eq-zero-iff*)
from ‹*lc q* / *lc p* $\neq$ *0*› ‹*p′* $\neq$ *0*› **have** *lt* (*?mult p′*) = *lt q*
by (*simp add*: *lt-monom-mult* ‹*lt p′* = *lt p*› *eq1*)
from ‹*lc p* $\neq$ *0*› **have** *lc* (*?mult p′*) = *lc q* **by** (*simp add*: ‹*lc p′* = *lc p*›)
from *p-red* **have** *mult-p-red*: (*sig-red* ($\prec_t$) ($\preceq$) *G*)** (*?mult p*) (*?mult p′*)
by (*rule sig-red-rtrancl-monom-mult*)
have *rep-list* (*?mult p′*) = *0* **by** (*simp add*: *rep-list-monom-mult* ‹*rep-list p′* = *0*›)
hence *mult-p′-irred*: $\neg$ *is-sig-red* ($\prec_t$) ($\preceq$) *G* (*?mult p′*)
using *is-sig-red-addsE* **by** *fastforce*
from *assms*(*1*) *G-sub assms*(*3*) *p-red* **have** *p′* $\in$ *dgrad-sig-set d*
by (*rule dgrad-sig-set-closed-sig-red-rtrancl*)
with *assms*(*1*) ‹*d* (*lp q* $-$ *lp p*) $\leq$ *dgrad-max d*› **have** *?mult p′* $\in$ *dgrad-sig-set d*
by (*rule dgrad-sig-set-closed-monom-mult*)

from *assms*(*1*) ‹*G* $\subseteq$ *dgrad-max-set d*› **obtain** *q′* **where** *q-red*: (*sig-red* ($\prec_t$) ($\preceq$) *G*)** *q q′*
and *q′-irred*: $\neg$ *is-sig-red* ($\prec_t$) ($\preceq$) *G q′* **by** (*rule sig-irredE-dgrad-max-set*)
from *q-red* **have** *lt q′* = *lt q* **and** *lc q′* = *lc q*
by (*rule sig-red-regular-rtrancl-lt*, *rule sig-red-regular-rtrancl-lc*)
hence *q′* $\neq$ *0* **using** ‹*lc q* $\neq$ *0*› **by** *auto*

from *assms*(*2*) **have** *is-sig-GB-upt d G* (*lt* (*?mult p'*)) **by** (*simp only*: ‹*lt* (*?mult p'*) = *lt q*›)
moreover from *assms*(*1*) *G-sub assms*(*4*) *q-red* **have** *q'* ∈ *dgrad-sig-set d*
by (*rule dgrad-sig-set-closed-sig-red-rtrancl*)
moreover note ‹*?mult p'* ∈ *dgrad-sig-set d*›
moreover have *lt q'* = *lt* (*?mult p'*) **by** (*simp only*: ‹*lt* (*?mult p'*) = *lt q*› ‹*lt q'* = *lt q*›)
moreover have *lc q'* = *lc* (*?mult p'*) **by** (*simp only*: ‹*lc* (*?mult p'*) = *lc q*› ‹*lc q'* = *lc q*›)
ultimately have *rep-list q'* = *rep-list* (*?mult p'*) **using** *q'-irred mult-p'-irred*
by (*rule sig-regular-reduced-unique*)
with ‹*rep-list* (*?mult p'*) = *0*› **have** *rep-list q'* = *0* **by** *simp*
with *q-red* **show** *?thesis* **by** (*rule sig-red-zeroI*)
qed

lemma *is-syz-sigI*:
assumes *s* ≠ *0* **and** *lt s* = *u* **and** *s* ∈ *dgrad-sig-set d* **and** *rep-list s* = *0*
shows *is-syz-sig d u*
unfolding *is-syz-sig-def* **using** *assms* **by** *blast*

lemma *is-syz-sigE*:
assumes *is-syz-sig d u*
obtains *r* **where** *r* ≠ *0* **and** *lt r* = *u* **and** *r* ∈ *dgrad-sig-set d* **and** *rep-list r* = *0*
using *assms* **unfolding** *is-syz-sig-def* **by** *blast*

lemma *is-syz-sig-adds*:
assumes *dickson-grading d* **and** *is-syz-sig d u* **and** *u* $adds_t$ *v*
and *d* (*pp-of-term v*) ≤ *dgrad-max d*
shows *is-syz-sig d v*
proof −
from *assms*(*2*) **obtain** *s* **where** *s* ≠ *0* **and** *lt s* = *u* **and** *s* ∈ *dgrad-sig-set d*
and *rep-list s* = *0* **by** (*rule is-syz-sigE*)
from *assms*(*3*) **obtain** *t* **where** *v*: *v* = *t* ⊕ *u* **by** (*rule adds-termE*)
show *?thesis*
proof (*rule is-syz-sigI*)
from ‹*s* ≠ *0*› **show** *monom-mult 1 t s* ≠ *0* **by** (*simp add*: *monom-mult-eq-zero-iff*)
next
from ‹*s* ≠ *0*› **show** *lt* (*monom-mult 1 t s*) = *v* **by** (*simp add*: *lt-monom-mult v* ‹*lt s* = *u*›)
next
from *assms*(*4*) **have** *d* (*t* + *pp-of-term u*) ≤ *dgrad-max d* **by** (*simp add*: *v term-simps*)
with *assms*(*1*) **have** *d t* ≤ *dgrad-max d* **by** (*simp add*: *dickson-gradingD1*)
with *assms*(*1*) **show** *monom-mult 1 t s* ∈ *dgrad-sig-set d* **using** ‹*s* ∈ *dgrad-sig-set d*›
by (*rule dgrad-sig-set-closed-monom-mult*)
next
show *rep-list* (*monom-mult 1 t s*) = *0* **by** (*simp add*: ‹*rep-list s* = *0*› *rep-list-monom-mult*)

  qed
qed

lemma *syzygy-crit*:
  assumes *dickson-grading d* **and** *is-sig-GB-upt d G u* **and** *is-syz-sig d u*
    and $p \in$ *dgrad-sig-set d* **and** *lt p* $=$ *u*
  shows *sig-red-zero* $(\prec_t)$ *G p*
proof −
  from *assms*(*3*) **obtain** *s* **where** $s \neq 0$ **and** *lt s* $= u$ **and** $s \in$ *dgrad-sig-set d*
    and *rep-list s* $= 0$ **by** (*rule is-syz-sigE*)
  note *assms*(*1*)
  moreover from *assms*(*2*) **have** *is-sig-GB-upt d G* (*lt p*) **by** (*simp only*: *assms*(*5*))
  moreover note ‹$s \in$ *dgrad-sig-set d*› *assms*(*4*) ‹$s \neq 0$›
  moreover from *rtranclp.rtrancl-refl* ‹*rep-list s* $= 0$› **have** *sig-red-zero* $(\prec_t)$ *G s*
    by (*rule sig-red-zeroI*)
  moreover have *lt s* $adds_t$ *lt p* **by** (*simp only*: *assms*(*5*) ‹*lt s* $= u$› *adds-term-refl*)
  ultimately show *?thesis* **by** (*rule sig-red-zero-regularI-adds*)
qed

lemma *lemma-21*:
  assumes *dickson-grading d* **and** *is-sig-GB-upt d G* (*lt p*) **and** $p \in$ *dgrad-sig-set d* **and** $g \in G$
    and *rep-list p* $\neq 0$ **and** *rep-list g* $\neq 0$ **and** *lt g* $adds_t$ *lt p*
    and *punit.lt* (*rep-list g*) *adds punit.lt* (*rep-list p*)
  shows *is-sig-red* $(\preceq_t)$ $(=)$ *G p*
proof −
  let *?lp* = *punit.lt* (*rep-list p*)
  define *s* **where** *s* = *?lp* − *punit.lt* (*rep-list g*)
  from *assms*(*8*) **have** *s*: *?lp* = *s* + *punit.lt* (*rep-list g*) **by** (*simp add*: *s-def minus-plus*)
  from *assms*(*7*) **obtain** *t* **where** *lt-p*: *lt p* $= t \oplus$ *lt g* **by** (*rule adds-termE*)
  show *?thesis*
  proof (*cases* $s \oplus$ *lt g* $\preceq_t$ *lt p*)
    case *True*
    hence *?lp* $\oplus$ *lt g* $\preceq_t$ *punit.lt* (*rep-list g*) $\oplus$ *lt p*
      by (*simp add*: *s splus-assoc splus-left-commute*[*of s*] *splus-mono*)
    with *assms*(*4*, *6*, *5*, *8*) *ord-term-lin.is-le-relI*(*2*) **show** *?thesis*
      by (*rule is-sig-red-top-addsI*)
  next
    case *False*
    hence *lt p* $\prec_t$ $s \oplus$ *lt g* **by** *simp*
    hence $t \prec s$ **by** (*simp add*: *lt-p ord-term-strict-canc-left*)
    hence *t* + *punit.lt* (*rep-list g*) $\prec$ *s* + *punit.lt* (*rep-list g*) **by** (*rule plus-monotone-strict*)
    hence *t* + *punit.lt* (*rep-list g*) $\prec$ *?lp* **by** (*simp only*: *s*)
    from *assms*(*5*) **have** $p \neq 0$ **by** (*auto simp*: *rep-list-zero*)
    hence *lc p* $\neq 0$ **by** (*rule lc-not-0*)
    from *assms*(*6*) **have** $g \neq 0$ **by** (*auto simp*: *rep-list-zero*)
    hence *lc g* $\neq 0$ **by** (*rule lc-not-0*)
    with ‹*lc p* $\neq 0$› **have** *1*: *lc p* / *lc g* $\neq 0$ **by** *simp*

let *?g* = *monom-mult* (*lc p* / *lc g*) *t g*
from *1* ‹*g* ≠ *0*› **have** *lt ?g* = *lt p* **unfolding** *lt-p* **by** (*rule lt-monom-mult*)
from ‹*lc g* ≠ *0*› **have** *lc ?g* = *lc p* **by** *simp*
have *punit.lt* (*rep-list ?g*) = *t* + *punit.lt* (*rep-list g*)
unfolding *rep-list-monom-mult* **using** *1 assms*(*6*) **by** (*rule punit.lt-monom-mult*[*simplified*])
also have ... ≺ *?lp* **by** *fact*
finally have *punit.lt* (*rep-list ?g*) ≺ *?lp* **.**
hence *lt-pg*: *punit.lt* (*rep-list* (*p* − *?g*)) = *?lp* **and** *rep-list p* ≠ *rep-list ?g*
by (*auto simp*: *rep-list-minus punit.lt-minus-eqI-2*)
from *this*(*2*) **have** *rep-list* (*p* − *?g*) ≠ *0* **and** *p* − *?g* ≠ *0*
by (*auto simp*: *rep-list-minus rep-list-zero*)

from *assms*(*2*) **have** *G* ⊆ *dgrad-sig-set d* **by** (*rule is-sig-GB-uptD1*)
note *assms*(*1*)
moreover have *d t* ≤ *dgrad-max d*
proof (*rule le-trans*)
have *lp p* = *t* + *lp g* **by** (*simp add*: *lt-p term-simps*)
with *assms*(*1*) **show** *d t* ≤ *d* (*lp p*) **by** (*simp add*: *dickson-grading-adds-imp-le*)
next
from *assms*(*3*) **show** *d* (*lp p*) ≤ *dgrad-max d* **by** (*rule dgrad-sig-setD-lp*)
qed
moreover from *assms*(*4*) ‹*G* ⊆ *dgrad-sig-set d*› **have** *g* ∈ *dgrad-sig-set d* **..**
ultimately have *?g* ∈ *dgrad-sig-set d* **by** (*rule dgrad-sig-set-closed-monom-mult*)

note *assms*(*2*)
moreover from *assms*(*3*) ‹*?g* ∈ *dgrad-sig-set d*› **have** *p* − *?g* ∈ *dgrad-sig-set d*
by (*rule dgrad-sig-set-closed-minus*)
moreover from ‹*p* − *?g* ≠ *0*› ‹*lt ?g* = *lt p*› ‹*lc ?g* = *lc p*› **have** *lt* (*p* − *?g*) $\prec_t$ *lt p*
by (*rule lt-minus-lessI*)
ultimately have *sig-red-zero* ($\preceq_t$) *G* (*p* − *?g*)
by (*rule is-sig-GB-uptD3*)
moreover note ‹*rep-list* (*p* − *?g*) ≠ *0*›
moreover have ($\preceq_t$) = ($\preceq_t$) ∨ ($\preceq_t$) = ($\prec_t$) **by** *simp*
ultimately have *is-sig-red* ($\preceq_t$) (=) *G* (*p* − *?g*) **by** (*rule sig-red-zero-nonzero*)
then obtain *g1* **where** *g1* ∈ *G* **and** *rep-list g1* ≠ *0*
and *2*: *punit.lt* (*rep-list g1*) *adds punit.lt* (*rep-list* (*p* − *?g*))
and *3*: *punit.lt* (*rep-list* (*p* − *?g*)) ⊕ *lt g1* $\preceq_t$ *punit.lt* (*rep-list g1*) ⊕ *lt* (*p* − *?g*)
by (*rule is-sig-red-top-addsE*)
from ‹*g1* ∈ *G*› ‹*rep-list g1* ≠ *0*› *assms*(*5*) **show** *?thesis*
proof (*rule is-sig-red-top-addsI*)
from *2* **show** *punit.lt* (*rep-list g1*) *adds punit.lt* (*rep-list p*) **by** (*simp only*: *lt-pg*)
next
have *?lp* ⊕ *lt g1* = *punit.lt* (*rep-list* (*p* − *?g*)) ⊕ *lt g1* **by** (*simp only*: *lt-pg*)
also have ... $\preceq_t$ *punit.lt* (*rep-list g1*) ⊕ *lt* (*p* − *?g*) **by** (*fact 3*)

also from ‹$lt\ (p - ?g) \prec_t lt\ p$› **have** ... $\prec_t$ *punit.lt* (*rep-list g1*) $\oplus$ *lt p*
by (*rule splus-mono-strict*)
finally show $?lp \oplus lt\ g1 \preceq_t$ *punit.lt* (*rep-list g1*) $\oplus$ *lt p* **by** (*rule ord-term-lin.less-imp-le*)
qed *simp*
qed
qed

4.2.3 Rewrite Bases

definition *is-rewrite-ord* :: $(('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b)) \Rightarrow bool) \Rightarrow bool$
where *is-rewrite-ord rword* $\longleftrightarrow$ (*reflp rword* $\wedge$ *transp rword* $\wedge$ ($\forall$ *a b. rword a b* $\vee$ *rword b a*) $\wedge$
($\forall$ *a b. rword a b* $\longrightarrow$ *rword b a* $\longrightarrow$ *fst a* = *fst b*) $\wedge$
($\forall$ *d G a b. dickson-grading d* $\longrightarrow$ *is-sig-GB-upt d G* (*lt b*) $\longrightarrow$
$a \in G \longrightarrow b \in G \longrightarrow a \neq 0 \longrightarrow b \neq 0 \longrightarrow$ *lt a* $adds_t$ *lt b* $\longrightarrow$
$\neg$ *is-sig-red* $(\prec_t)$ $(=)$ *G b* $\longrightarrow$ *rword* (*spp-of a*) (*spp-of b*)))

definition *is-canon-rewriter* :: $(('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b)) \Rightarrow bool) \Rightarrow ('t \Rightarrow_0 'b)$ *set* $\Rightarrow 't \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow bool$
where *is-canon-rewriter rword A u p* $\longleftrightarrow$
($p \in A \wedge p \neq 0 \wedge$ *lt p* $adds_t$ *u* $\wedge$ ($\forall a \in A.\ a \neq 0 \longrightarrow$ *lt a* $adds_t$ *u* $\longrightarrow$ *rword* (*spp-of a*) (*spp-of p*)))

definition *is-RB-in* :: $('a \Rightarrow nat) \Rightarrow (('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b)) \Rightarrow bool) \Rightarrow ('t \Rightarrow_0 'b)$ *set* $\Rightarrow 't \Rightarrow bool$
where *is-RB-in d rword G u* $\longleftrightarrow$
(($\exists$ *g. is-canon-rewriter rword G u g* $\wedge$ $\neg$ *is-sig-red* $(\prec_t)$ $(=)$ *G* (*monom-mult* *1* (*pp-of-term u* $-$ *lp g*) *g*)) $\vee$
is-syz-sig d u)

definition *is-RB-upt* :: $('a \Rightarrow nat) \Rightarrow (('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b)) \Rightarrow bool) \Rightarrow ('t \Rightarrow_0 'b)$ *set* $\Rightarrow 't \Rightarrow bool$
where *is-RB-upt d rword G u* $\longleftrightarrow$
($G \subseteq$ *dgrad-sig-set d* $\wedge$ ($\forall$ *v.* $v \prec_t u \longrightarrow$ *d* (*pp-of-term v*) $\leq$ *dgrad-max d* $\longrightarrow$
component-of-term v $<$ *length fs* $\longrightarrow$ *is-RB-in d rword G v*))

lemma *is-rewrite-ordI*:
assumes *reflp rword* **and** *transp rword* **and** $\bigwedge$*a b. rword a b* $\vee$ *rword b a*
and $\bigwedge$*a b. rword a b* $\Longrightarrow$ *rword b a* $\Longrightarrow$ *fst a* = *fst b*
and $\bigwedge$*d G a b. dickson-grading d* $\Longrightarrow$ *is-sig-GB-upt d G* (*lt b*) $\Longrightarrow a \in G \Longrightarrow b \in G \Longrightarrow$
$a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow$ *lt a* $adds_t$ *lt b* $\Longrightarrow \neg$ *is-sig-red* $(\prec_t)$ $(=)$ *G b* $\Longrightarrow$ *rword* (*spp-of a*) (*spp-of b*)
shows *is-rewrite-ord rword*

unfolding *is-rewrite-ord-def* **using** *assms* **by** *blast*

lemma *is-rewrite-ordD1*: *is-rewrite-ord rword* $\Longrightarrow$ *rword a a*
by (*simp add*: *is-rewrite-ord-def reflpD*)

lemma *is-rewrite-ordD2*: *is-rewrite-ord rword* $\Longrightarrow$ *rword a b* $\Longrightarrow$ *rword b c* $\Longrightarrow$ *rword a c*
by (*auto simp*: *is-rewrite-ord-def dest*: *transpD*)

lemma *is-rewrite-ordD3*:
assumes *is-rewrite-ord rword*
and *rword a b* $\Longrightarrow$ *thesis*
and $\neg$ *rword a b* $\Longrightarrow$ *rword b a* $\Longrightarrow$ *thesis*
shows *thesis*
proof −
from *assms*(*1*) **have** *disj*: *rword a b* $\vee$ *rword b a*
by (*simp add*: *is-rewrite-ord-def del*: *split-paired-All*)
show *?thesis*
proof (*cases rword a b*)
case *True*
thus *?thesis* **by** (*rule assms*(*2*))
next
case *False*
moreover from *this disj* **have** *rword b a* **by** *simp*
ultimately show *?thesis* **by** (*rule assms*(*3*))
qed
qed

lemma *is-rewrite-ordD4*:
assumes *is-rewrite-ord rword* **and** *rword a b* **and** *rword b a*
shows *fst a* = *fst b*
using *assms* **unfolding** *is-rewrite-ord-def* **by** *blast*

lemma *is-rewrite-ordD4'*:
assumes *is-rewrite-ord rword* **and** *rword* (*spp-of a*) (*spp-of b*) **and** *rword* (*spp-of b*) (*spp-of a*)
shows *lt a* = *lt b*
proof −
from *assms* **have** *fst* (*spp-of a*) = *fst* (*spp-of b*) **by** (*rule is-rewrite-ordD4*)
thus *?thesis* **by** (*simp add*: *spp-of-def*)
qed

lemma *is-rewrite-ordD5*:
assumes *is-rewrite-ord rword* **and** *dickson-grading d* **and** *is-sig-GB-upt d G* (*lt b*)
and $a \in G$ **and** $b \in G$ **and** $a \neq 0$ **and** $b \neq 0$ **and** *lt a* $adds_t$ *lt b*
and $\neg$ *is-sig-red* ($\prec_t$) (=) *G b*
shows *rword* (*spp-of a*) (*spp-of b*)
using *assms* **unfolding** *is-rewrite-ord-def* **by** *blast*

lemma *is-canon-rewriterI*:
assumes $p \in A$ **and** $p \neq 0$ **and** $lt\ p\ adds_t\ u$
and $\bigwedge a.\ a \in A \Longrightarrow a \neq 0 \Longrightarrow lt\ a\ adds_t\ u \Longrightarrow rword\ (spp\text{-}of\ a)\ (spp\text{-}of\ p)$
shows *is-canon-rewriter rword A u p*
unfolding *is-canon-rewriter-def* **using** *assms* **by** *blast*

lemma *is-canon-rewriterD1*: $is\text{-}canon\text{-}rewriter\ rword\ A\ u\ p \Longrightarrow p \in A$
by (*simp add*: *is-canon-rewriter-def*)

lemma *is-canon-rewriterD2*: $is\text{-}canon\text{-}rewriter\ rword\ A\ u\ p \Longrightarrow p \neq 0$
by (*simp add*: *is-canon-rewriter-def*)

lemma *is-canon-rewriterD3*: $is\text{-}canon\text{-}rewriter\ rword\ A\ u\ p \Longrightarrow lt\ p\ adds_t\ u$
by (*simp add*: *is-canon-rewriter-def*)

lemma *is-canon-rewriterD4*:
$is\text{-}canon\text{-}rewriter\ rword\ A\ u\ p \Longrightarrow a \in A \Longrightarrow a \neq 0 \Longrightarrow lt\ a\ adds_t\ u \Longrightarrow rword$
$(spp\text{-}of\ a)\ (spp\text{-}of\ p)$
by (*simp add*: *is-canon-rewriter-def*)

lemmas *is-canon-rewriterD* = *is-canon-rewriterD1 is-canon-rewriterD2 is-canon-rewriterD3*
is-canon-rewriterD4

lemma *is-rewrite-ord-finite-canon-rewriterE*:
assumes *is-rewrite-ord rword* **and** *finite A* **and** $a \in A$ **and** $a \neq 0$ **and** $lt\ a\ adds_t$
u
obtains p **where** *is-canon-rewriter rword A u p*
proof −
let $?A = \{x.\ x \in A \wedge x \neq 0 \wedge lt\ x\ adds_t\ u\}$
let $?rel = \lambda x\ y.\ strict\ rword\ (spp\text{-}of\ y)\ (spp\text{-}of\ x)$
have *finite ?A*
proof (*rule finite-subset*)
show $?A \subseteq A$ **by** *blast*
qed *fact*
moreover have $?A \neq \{\}$
proof
from *assms*(*3*, *4*, *5*) **have** $a \in ?A$ **by** *simp*
also assume $?A = \{\}$
finally show *False* **by** *simp*
qed
moreover have *irreflp ?rel*
proof −
from *assms*(*1*) **have** *reflp rword* **by** (*simp add*: *is-rewrite-ord-def*)
thus *?thesis* **by** (*simp add*: *reflp-def irreflp-def*)
qed
moreover have *transp ?rel*
proof −
from *assms*(*1*) **have** *transp rword* **by** (*simp add*: *is-rewrite-ord-def*)

thus *?thesis* **by** (*auto simp*: *transp-def simp del*: *split-paired-All*)
qed
ultimately obtain *p* **where** $p \in$ *?A* **and** $*$: $\bigwedge b.$ *?rel b p* $\Longrightarrow b \notin$ *?A* **by** (*rule finite-minimalE*, *blast*)
from *this*(*1*) **have** $p \in A$ **and** $p \neq 0$ **and** *lt p* $adds_t$ *u* **by** *simp-all*
show *?thesis*
proof (*rule*, *rule is-canon-rewriterI*)
fix *q*
assume $q \in A$ **and** $q \neq 0$ **and** *lt q* $adds_t$ *u*
hence $q \in$ *?A* **by** *simp*
with $*$ **have** $\neg$ *?rel q p* **by** *blast*
hence *disj*: $\neg$ *rword* (*spp-of p*) (*spp-of q*) $\vee$ *rword* (*spp-of q*) (*spp-of p*) **by** *simp*
from *assms*(*1*) **show** *rword* (*spp-of q*) (*spp-of p*)
proof (*rule is-rewrite-ordD3*)
assume $\neg$ *rword* (*spp-of q*) (*spp-of p*) **and** *rword* (*spp-of p*) (*spp-of q*)
with *disj* **show** *?thesis* **by** *simp*
qed
qed *fact+*
qed

lemma *is-rewrite-ord-canon-rewriterD1*:
assumes *is-rewrite-ord rword* **and** *is-canon-rewriter rword A u p* **and** *is-canon-rewriter rword A v q*
and *lt p* $adds_t$ *v* **and** *lt q* $adds_t$ *u*
shows *lt p* = *lt q*
proof −
from *assms*(*2*) **have** $p \in A$ **and** $p \neq 0$
and *1*: $\bigwedge a.$ $a \in A \Longrightarrow a \neq 0 \Longrightarrow$ *lt a* $adds_t$ *u* $\Longrightarrow$ *rword* (*spp-of a*) (*spp-of p*)
by (*rule is-canon-rewriterD*)+
from *assms*(*3*) **have** $q \in A$ **and** $q \neq 0$
and *2*: $\bigwedge a.$ $a \in A \Longrightarrow a \neq 0 \Longrightarrow$ *lt a* $adds_t$ *v* $\Longrightarrow$ *rword* (*spp-of a*) (*spp-of q*)
by (*rule is-canon-rewriterD*)+
note *assms*(*1*)
moreover from ‹$p \in A$› ‹$p \neq 0$› *assms*(*4*) **have** *rword* (*spp-of p*) (*spp-of q*) **by** (*rule 2*)
moreover from ‹$q \in A$› ‹$q \neq 0$› *assms*(*5*) **have** *rword* (*spp-of q*) (*spp-of p*) **by** (*rule 1*)
ultimately show *?thesis* **by** (*rule is-rewrite-ordD4'*)
qed

corollary *is-rewrite-ord-canon-rewriterD2*:
assumes *is-rewrite-ord rword* **and** *is-canon-rewriter rword A u p* **and** *is-canon-rewriter rword A u q*
shows *lt p* = *lt q*
using *assms*
proof (*rule is-rewrite-ord-canon-rewriterD1*)
from *assms*(*2*) **show** *lt p* $adds_t$ *u* **by** (*rule is-canon-rewriterD*)
next
from *assms*(*3*) **show** *lt q* $adds_t$ *u* **by** (*rule is-canon-rewriterD*)

qed

lemma *is-rewrite-ord-canon-rewriterD3*:
assumes *is-rewrite-ord rword* **and** *dickson-grading d* **and** *is-canon-rewriter rword A u p*
and $a \in A$ **and** $a \neq 0$ **and** *lt a adds*$_t$ *u* **and** *is-sig-GB-upt d A* (*lt a*)
and *lt p adds*$_t$ *lt a* **and** ¬ *is-sig-red* ($\prec_t$) (=) *A a*
shows *lt p* = *lt a*
proof −
note *assms*(*1*)
moreover from *assms*(*1*, *2*, *7*) - *assms*(*4*) - *assms*(*5*, *8*, *9*) **have** *rword* (*spp-of p*) (*spp-of a*)
proof (*rule is-rewrite-ordD5*)
from *assms*(*3*) **show** $p \in A$ **and** $p \neq 0$ **by** (*rule is-canon-rewriterD*)+
qed
moreover from *assms*(*3*, *4*, *5*, *6*) **have** *rword* (*spp-of a*) (*spp-of p*) **by** (*rule is-canon-rewriterD4*)
ultimately show *?thesis* **by** (*rule is-rewrite-ordD4'*)
qed

lemma *is-RB-inI1*:
assumes *is-canon-rewriter rword G u g* **and** ¬ *is-sig-red* ($\prec_t$) (=) *G* (*monom-mult 1* (*pp-of-term u* − *lp g*) *g*)
shows *is-RB-in d rword G u*
unfolding *is-RB-in-def* **using** *assms is-canon-rewriterD1* **by** *blast*

lemma *is-RB-inI2*:
assumes *is-syz-sig d u*
shows *is-RB-in d rword G u*
unfolding *is-RB-in-def Let-def* **using** *assms* **by** *blast*

lemma *is-RB-inE*:
assumes *is-RB-in d rword G u*
and *is-syz-sig d u* $\Longrightarrow$ *thesis*
and $\bigwedge g$. ¬ *is-syz-sig d u* $\Longrightarrow$ *is-canon-rewriter rword G u g* $\Longrightarrow$
¬ *is-sig-red* ($\prec_t$) (=) *G* (*monom-mult 1* (*pp-of-term u* − *lp g*) *g*) $\Longrightarrow$ *thesis*
shows *thesis*
using *assms* **unfolding** *is-RB-in-def* **by** *blast*

lemma *is-RB-inD*:
assumes *dickson-grading d* **and** $G \subseteq$ *dgrad-sig-set d* **and** *is-RB-in d rword G u*
and ¬ *is-syz-sig d u* **and** *d* (*pp-of-term u*) $\leq$ *dgrad-max d*
and *is-canon-rewriter rword G u g*
shows *rep-list g* $\neq$ *0*
proof
assume *a*: *rep-list g* = *0*
from *assms*(*1*) **have** *is-syz-sig d u*
proof (*rule is-syz-sig-adds*)

show *is-syz-sig d* (*lt g*)
proof (*rule is-syz-sigI*)
from *assms*(*6*) **show** $g \neq 0$ **by** (*rule is-canon-rewriterD2*)
next
from *assms*(*6*) **have** $g \in G$ **by** (*rule is-canon-rewriterD1*)
thus $g \in$ *dgrad-sig-set d* **using** *assms*(*2*) **..**
qed (*fact refl*, *fact a*)
next
from *assms*(*6*) **show** *lt g* $adds_t$ *u* **by** (*rule is-canon-rewriterD3*)
qed *fact*
with *assms*(*4*) **show** *False* **..**
qed

lemma *is-RB-uptI*:
assumes $G \subseteq$ *dgrad-sig-set d*
and $\bigwedge v.\ v \prec_t u \Longrightarrow d$ (*pp-of-term v*) $\leq$ *dgrad-max d* $\Longrightarrow$ *component-of-term v* $<$ *length fs* $\Longrightarrow$
is-RB-in d canon G v
shows *is-RB-upt d canon G u*
unfolding *is-RB-upt-def* **using** *assms* **by** *blast*

lemma *is-RB-uptD1*:
assumes *is-RB-upt d canon G u*
shows $G \subseteq$ *dgrad-sig-set d*
using *assms* **unfolding** *is-RB-upt-def* **by** *blast*

lemma *is-RB-uptD2*:
assumes *is-RB-upt d canon G u* **and** $v \prec_t u$ **and** *d* (*pp-of-term v*) $\leq$ *dgrad-max d*
and *component-of-term v* $<$ *length fs*
shows *is-RB-in d canon G v*
using *assms* **unfolding** *is-RB-upt-def* **by** *blast*

lemma *is-RB-in-UnI*:
assumes *is-RB-in d rword G u* **and** $\bigwedge h.\ h \in H \Longrightarrow u \prec_t$ *lt h*
shows *is-RB-in d rword* ($H \cup G$) *u*
using *assms*(*1*)
proof (*rule is-RB-inE*)
assume *is-syz-sig d u*
thus *is-RB-in d rword* ($H \cup G$) *u* **by** (*rule is-RB-inI2*)
next
fix g'
assume *crw*: *is-canon-rewriter rword G u* g'
and *irred*: $\neg$ *is-sig-red* ($\prec_t$) (=) *G* (*monom-mult 1* (*pp-of-term u* $-$ *lp* g') g')
from *crw* **have** $g' \in G$ **and** $g' \neq 0$ **and** *lt* g' $adds_t$ *u*
and *max*: $\bigwedge a.\ a \in G \Longrightarrow a \neq 0 \Longrightarrow$ *lt a* $adds_t$ $u \Longrightarrow$ *rword* (*spp-of a*) (*spp-of* g')
by (*rule is-canon-rewriterD*)+
show *is-RB-in d rword* ($H \cup G$) *u*

proof (*rule is-RB-inI1*)
show *is-canon-rewriter rword* $(H \cup G)$ *u* g'
proof (*rule is-canon-rewriterI*)
from ‹$g' \in G$› **show** $g' \in H \cup G$ **by** *simp*
next
fix *a*
assume $a \in H \cup G$ **and** $a \neq 0$ **and** *lt a* $adds_t$ *u*
from *this*(*1*) **show** *rword* (*spp-of a*) (*spp-of* g')
proof
assume $a \in H$
with ‹*lt a* $adds_t$ *u*› **have** *lt a* $adds_t$ *u* **by** *simp*
hence *lt a* $\preceq_t$ *u* **by** (*rule ord-adds-term*)
moreover from ‹$a \in H$› **have** *u* $\prec_t$ *lt a* **by** (*rule assms*(*2*))
ultimately show *?thesis* **by** *simp*
next
assume $a \in G$
thus *?thesis* **using** ‹$a \neq 0$› ‹*lt a* $adds_t$ *u*› **by** (*rule max*)
qed
qed *fact+*
next
show $\neg$ *is-sig-red* $(\prec_t)$ $(=)$ $(H \cup G)$ (*monom-mult 1* (*pp-of-term u* $-$ *lp* g') g')
(**is** $\neg$ *is-sig-red - - - ?g*)
proof
assume *is-sig-red* $(\prec_t)$ $(=)$ $(H \cup G)$ *?g*
with *irred* **have** *is-sig-red* $(\prec_t)$ $(=)$ *H ?g* **by** (*simp add*: *is-sig-red-Un del*: *Un-insert-left*)
then obtain *h* **where** $h \in H$ **and** *is-sig-red* $(\prec_t)$ $(=)$ $\{h\}$ *?g* **by** (*rule is-sig-red-singletonI*)
from *this*(*2*) **have** *lt h* $\prec_t$ *lt ?g* **by** (*rule is-sig-red-regularD-lt*)
also from ‹$g' \neq 0$› ‹*lt* g' $adds_t$ *u*› **have** ... = *u*
by (*auto simp*: *lt-monom-mult adds-term-alt pp-of-term-splus*)
finally have *lt h* $\prec_t$ *u* **.**
moreover from ‹$h \in H$› **have** *u* $\prec_t$ *lt h* **by** (*rule assms*(*2*))
ultimately show *False* **by** *simp*
qed
qed
qed

corollary *is-RB-in-insertI*:
assumes *is-RB-in d rword G u* **and** *u* $\prec_t$ *lt g*
shows *is-RB-in d rword* (*insert g G*) *u*
proof $-$
from *assms*(*1*) **have** *is-RB-in d rword* $(\{g\} \cup G)$ *u*
proof (*rule is-RB-in-UnI*)
fix *h*
assume $h \in \{g\}$
with *assms*(*2*) **show** *u* $\prec_t$ *lt h* **by** *simp*
qed
thus *?thesis* **by** *simp*

qed

corollary *is-RB-upt-UnI*:
  assumes *is-RB-upt d rword G u* **and** $H \subseteq$ *dgrad-sig-set d* **and** $\bigwedge h.\ h \in H \Longrightarrow u \preceq_t$ *lt h*
  shows *is-RB-upt d rword* $(H \cup G)$ *u*
proof (*rule is-RB-uptI*)
  from *assms*(*1*) **have** $G \subseteq$ *dgrad-sig-set d* **by** (*rule is-RB-uptD1*)
  with *assms*(*2*) **show** $H \cup G \subseteq$ *dgrad-sig-set d* **by** (*rule Un-least*)
next
  fix *v*
  assume $v \prec_t u$ **and** *d* (*pp-of-term v*) $\leq$ *dgrad-max d* **and** *component-of-term v* $<$ *length fs*
  with *assms*(*1*) **have** *is-RB-in d rword G v* **by** (*rule is-RB-uptD2*)
  moreover from ‹$v \prec_t u$› *assms*(*3*) **have** $\bigwedge h.\ h \in H \Longrightarrow v \prec_t$ *lt h* **by** (*rule ord-term-lin.less-le-trans*)
  ultimately show *is-RB-in d rword* $(H \cup G)$ *v* **by** (*rule is-RB-in-UnI*)
qed

corollary *is-RB-upt-insertI*:
  assumes *is-RB-upt d rword G u* **and** $g \in$ *dgrad-sig-set d* **and** $u \preceq_t$ *lt g*
  shows *is-RB-upt d rword* (*insert g G*) *u*
proof −
  from *assms*(*1*) **have** *is-RB-upt d rword* $(\{g\} \cup G)$ *u*
  proof (*rule is-RB-upt-UnI*)
    from *assms*(*2*) **show** $\{g\} \subseteq$ *dgrad-sig-set d* **by** *simp*
  next
    fix *h*
    assume $h \in \{g\}$
    with *assms*(*3*) **show** $u \preceq_t$ *lt h* **by** *simp*
  qed
  thus *?thesis* **by** *simp*
qed

lemma *is-RB-upt-is-sig-GB-upt*:
  assumes *dickson-grading d* **and** *is-RB-upt d rword G u*
  shows *is-sig-GB-upt d G u*
proof (*rule ccontr*)
  let *?Q* = $\{v.\ v \prec_t u \wedge d$ (*pp-of-term v*) $\leq$ *dgrad-max d* $\wedge$ *component-of-term v* $<$ *length fs* $\wedge \neg$ *is-sig-GB-in d G v*$\}$
  have *Q-sub*: *pp-of-term* ‘ *?Q* $\subseteq$ *dgrad-set d* (*dgrad-max d*) **by** *blast*
  from *assms*(*2*) **have** *G-sub*: $G \subseteq$ *dgrad-sig-set d* **by** (*rule is-RB-uptD1*)
  hence $G \subseteq$ *dgrad-max-set d* **by** (*simp add*: *dgrad-sig-set'-def*)
  assume $\neg$ *is-sig-GB-upt d G u*
  with *G-sub* **obtain** v' **where** $v' \in$ *?Q* **unfolding** *is-sig-GB-upt-def* **by** *blast*
  with *assms*(*1*) **obtain** *v* **where** $v \in$ *?Q* **and** *min*: $\bigwedge y.\ y \prec_t v \Longrightarrow y \notin$ *?Q* **using** *Q-sub*
    by (*rule ord-term-minimum-dgrad-set*, *blast*)
  from ‹$v \in$ *?Q*› **have** $v \prec_t u$ **and** *d* (*pp-of-term v*) $\leq$ *dgrad-max d* **and** *compo-*

nent-of-term $v <$ *length fs*
and $\neg$ *is-sig-GB-in d G v* **by** *simp-all*
from *assms*(*2*) *this*(*1*, *2*, *3*) **have** *is-RB-in d rword G v* **by** (*rule is-RB-uptD2*)
from ‹$\neg$ *is-sig-GB-in d G v*› **obtain** *r* **where** *lt r* = *v* **and** *r* $\in$ *dgrad-sig-set d* **and** $\neg$ *sig-red-zero* ($\preceq_t$) *G r*
unfolding *is-sig-GB-in-def* **by** *blast*
from *this*(*3*) **have** *rep-list r* $\neq$ *0* **by** (*auto simp*: *sig-red-zero-def*)
hence *r* $\neq$ *0* **by** (*auto simp*: *rep-list-zero*)
hence *lc r* $\neq$ *0* **by** (*rule lc-not-0*)

from *G-sub* **have** *is-sig-GB-upt d G v*
proof (*rule is-sig-GB-uptI*)
fix *w*
assume *dw*: *d* (*pp-of-term w*) $\leq$ *dgrad-max d* **and** *cp*: *component-of-term w* $<$ *length fs*
assume *w* $\prec_t$ *v*
hence *w* $\notin$ *?Q* **by** (*rule min*)
hence $\neg$ *w* $\prec_t$ *u* $\vee$ *is-sig-GB-in d G w* **by** (*simp add*: *dw cp*)
thus *is-sig-GB-in d G w*
proof
assume $\neg$ *w* $\prec_t$ *u*
moreover from ‹*w* $\prec_t$ *v*› ‹*v* $\prec_t$ *u*› **have** *w* $\prec_t$ *u* **by** (*rule ord-term-lin.less-trans*)
ultimately show *?thesis* **..**
qed
qed

from ‹*is-RB-in d rword G v*› **have** *sig-red-zero* ($\preceq_t$) *G r*
proof (*rule is-RB-inE*)
assume *is-syz-sig d v*
have *sig-red-zero* ($\prec_t$) *G r* **by** (*rule syzygy-crit*, *fact*+)
thus *?thesis* **by** (*rule sig-red-zero-sing-regI*)
next
fix *g*
assume *a*: $\neg$ *is-sig-red* ($\prec_t$) (=) *G* (*monom-mult 1* (*pp-of-term v* $-$ *lp g*) *g*)
assume *is-canon-rewriter rword G v g*
hence *g* $\in$ *G* **and** *g* $\neq$ *0* **and** *lt g* $adds_t$ *v* **by** (*rule is-canon-rewriterD*)+
assume $\neg$ *is-syz-sig d v*
from ‹*g* $\in$ *G*› *G-sub* **have** *g* $\in$ *dgrad-sig-set d* **..**
from ‹*g* $\neq$ *0*› **have** *lc g* $\neq$ *0* **by** (*rule lc-not-0*)
with ‹*lc r* $\neq$ *0*› **have** *lc r* / *lc g* $\neq$ *0* **by** *simp*
from ‹*lt g* $adds_t$ *v*› **have** *lt g* $adds_t$ *lt r* **by** (*simp only*: ‹*lt r* = *v*›)
hence *eq1*: (*lp r* $-$ *lp g*) $\oplus$ *lt g* = *lt r* **by** (*metis add-implies-diff adds-termE pp-of-term-splus*)

let *?h* = *monom-mult* (*lc r* / *lc g*) (*lp r* $-$ *lp g*) *g*
from ‹*lc g* $\neq$ *0*› ‹*lc r* $\neq$ *0*› ‹*g* $\neq$ *0*› **have** *?h* $\neq$ *0* **by** (*simp add*: *monom-mult-eq-zero-iff*)
have *h-irred*: $\neg$ *is-sig-red* ($\prec_t$) (=) *G ?h*
proof
assume *is-sig-red* ($\prec_t$) (=) *G ?h*

moreover from ‹*lc g* $\neq$ *0*› ‹*lc r* $\neq$ *0*› **have** *lc g / lc r* $\neq$ *0* **by** *simp*
ultimately have *is-sig-red* ($\prec_t$) (=) *G* (*monom-mult* (*lc g / lc r*) *0 ?h*) **by** (*rule is-sig-red-monom-mult*)
with ‹*lc g* $\neq$ *0*› ‹*lc r* $\neq$ *0*› **have** *is-sig-red* ($\prec_t$) (=) *G* (*monom-mult 1* (*pp-of-term v* $-$ *lp g*) *g*)
by (*simp add*: *monom-mult-assoc* ‹*lt r* = *v*›)
with *a* **show** *False* ..
qed
from ‹*lc r / lc g* $\neq$ *0*› ‹*g* $\neq$ *0*› **have** *lt ?h* = *lt r* **by** (*simp add*: *lt-monom-mult eq1*)
hence *lt ?h* = *v* **by** (*simp only*: ‹*lt r* = *v*›)
from ‹*lc g* $\neq$ *0*› **have** *lc ?h* = *lc r* **by** *simp*
from *assms*(*1*) - ‹*g* $\in$ *dgrad-sig-set d*› **have** *?h* $\in$ *dgrad-sig-set d*
proof (*rule dgrad-sig-set-closed-monom-mult*)
from ‹*lt g adds*$_t$ *lt r*› **have** *lp g adds lp r* **by** (*simp add*: *adds-term-def*)
with *assms*(*1*) **have** *d* (*lp r* $-$ *lp g*) $\leq$ *d* (*lp r*) **by** (*rule dickson-grading-minus*)
also from ‹*r* $\in$ *dgrad-sig-set d*› **have** ... $\leq$ *dgrad-max d* **by** (*rule dgrad-sig-setD-lp*)
finally show *d* (*lp r* $-$ *lp g*) $\leq$ *dgrad-max d* .
qed
have *rep-list ?h* $\neq$ *0*
proof
assume *rep-list ?h* = *0*
with ‹*?h* $\neq$ *0*› ‹*lt ?h* = *v*› ‹*?h* $\in$ *dgrad-sig-set d*› **have** *is-syz-sig d v* **by** (*rule is-syz-sigI*)
with ‹$\neg$ *is-syz-sig d v*› **show** *False* ..
qed
hence *rep-list g* $\neq$ *0* **by** (*simp add*: *rep-list-monom-mult punit.monom-mult-eq-zero-iff*)
hence *punit.lc* (*rep-list g*) $\neq$ *0* **by** (*rule punit.lc-not-0*)
from *assms*(*1*) ‹*G* $\subseteq$ *dgrad-max-set d*› **obtain** *s* **where** *s-red*: (*sig-red* ($\prec_t$) ($\preceq$) *G*)** *r s*
and *s-irred*: $\neg$ *is-sig-red* ($\prec_t$) ($\preceq$) *G s* **by** (*rule sig-irredE-dgrad-max-set*)
from *s-red* **have** *s-red'*: (*sig-red* ($\preceq_t$) ($\preceq$) *G*)** *r s* **by** (*rule sig-red-rtrancl-sing-regI*)
have *rep-list s* $\neq$ *0*
proof
assume *rep-list s* = *0*
with *s-red'* **have** *sig-red-zero* ($\preceq_t$) *G r* **by** (*rule sig-red-zeroI*)
with ‹$\neg$ *sig-red-zero* ($\preceq_t$) *G r*› **show** *False* ..
qed
from *assms*(*1*) *G-sub* ‹*r* $\in$ *dgrad-sig-set d*› *s-red* **have** *s* $\in$ *dgrad-sig-set d*
by (*rule dgrad-sig-set-closed-sig-red-rtrancl*)
from *s-red* **have** *lt s* = *lt r* **and** *lc s* = *lc r*
by (*rule sig-red-regular-rtrancl-lt*, *rule sig-red-regular-rtrancl-lc*)
hence *lt ?h* = *lt s* **and** *lc ?h* = *lc s* **and** *s* $\neq$ *0*
using ‹*lc r* $\neq$ *0*› **by** (*auto simp*: ‹*lt ?h* = *lt r*› ‹*lc ?h* = *lc r*› *simp del*: *lc-monom-mult*)
from *s-irred* **have** $\neg$ *is-sig-red* ($\prec_t$) (=) *G s* **by** (*simp add*: *is-sig-red-top-tail-cases*)
from ‹*is-sig-GB-upt d G v*› **have** *is-sig-GB-upt d G* (*lt s*) **by** (*simp only*: ‹*lt s* = *lt r*› ‹*lt r* = *v*›)
have *punit.lt* (*rep-list ?h*) = *punit.lt* (*rep-list s*)

by (*rule sig-regular-top-reduced-lt-unique*, *fact*+)
hence *eq2*: *lp r* − *lp g* + *punit.lt* (*rep-list g*) = *punit.lt* (*rep-list s*)
using ‹*lc r* / *lc g* ≠ *0*› ‹*rep-list g* ≠ *0*› **by** (*simp add*: *rep-list-monom-mult punit.lt-monom-mult*)
have *punit.lc* (*rep-list ?h*) = *punit.lc* (*rep-list s*)
by (*rule sig-regular-top-reduced-lc-unique*, *fact*+)
hence *eq3*: *lc r* / *lc g* = *punit.lc* (*rep-list s*) / *punit.lc* (*rep-list g*)
using ‹*punit.lc* (*rep-list g*) ≠ *0*› **by** (*simp add*: *rep-list-monom-mult field-simps*)
have *sig-red-single* (=) (=) *s* (*s* − *?h*) *g* (*lp r* − *lp g*)
by (*rule sig-red-singleI*, *auto simp*: *eq1 eq2 eq3 punit.lc-def*[*symmetric*] ‹*lt s* = *lt r*›
‹*rep-list g* ≠ *0*› ‹*rep-list s* ≠ *0*› *intro*!: *punit.lt-in-keys*)
with ‹*g* ∈ *G*› **have** *sig-red* (=) (=) *G s* (*s* − *?h*) **unfolding** *sig-red-def* **by** *blast*
hence *sig-red* ($\preceq_t$) (⪯) *G s* (*s* − *?h*) **by** (*auto dest*: *sig-red-sing-regI sig-red-top-tailI*)
with *s-red′* **have** *r-red*: (*sig-red* ($\preceq_t$) (⪯) *G*)** *r* (*s* − *?h*) **..**
show *?thesis*
proof (*cases s* − *?h* = *0*)
case *True*
hence *rep-list* (*s* − *?h*) = *0* **by** (*simp only*: *rep-list-zero*)
with *r-red* **show** *?thesis* **by** (*rule sig-red-zeroI*)
next
case *False*
note ‹*is-sig-GB-upt d G* (*lt s*)›
moreover from ‹*s* ∈ *dgrad-sig-set d*› ‹*?h* ∈ *dgrad-sig-set d*› **have** *s* − *?h* ∈ *dgrad-sig-set d*
by (*rule dgrad-sig-set-closed-minus*)
moreover from *False* ‹*lt ?h* = *lt s*› ‹*lc ?h* = *lc s*› **have** *lt* (*s* − *?h*) $\prec_t$ *lt s* **by** (*rule lt-minus-lessI*)
ultimately have *sig-red-zero* ($\preceq_t$) *G* (*s* − *?h*) **by** (*rule is-sig-GB-uptD3*)
then obtain *s′* **where** (*sig-red* ($\preceq_t$) (⪯) *G*)** (*s* − *?h*) *s′* **and** *rep-list s′* = *0*
by (*rule sig-red-zeroE*)
from *r-red this*(*1*) **have** (*sig-red* ($\preceq_t$) (⪯) *G*)** *r s′* **by** *simp*
thus *?thesis* **using** ‹*rep-list s′* = *0*› **by** (*rule sig-red-zeroI*)
qed
qed
with ‹¬ *sig-red-zero* ($\preceq_t$) *G r*› **show** *False* **..**
qed

corollary *is-RB-upt-is-syz-sigD*:
assumes *dickson-grading d* **and** *is-RB-upt d rword G u*
and *is-syz-sig d u* **and** *p* ∈ *dgrad-sig-set d* **and** *lt p* = *u*
shows *sig-red-zero* ($\prec_t$) *G p*
proof −
note *assms*(*1*)
moreover from *assms*(*1*, *2*) **have** *is-sig-GB-upt d G u* **by** (*rule is-RB-upt-is-sig-GB-upt*)
ultimately show *?thesis* **using** *assms*(*3*, *4*, *5*) **by** (*rule syzygy-crit*)
qed

4.2.4 S-Pairs

definition *spair* :: $('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b)$
where *spair p q* = (*let t1* = *punit.lt* (*rep-list p*); *t2* = *punit.lt* (*rep-list q*); *l* = *lcs t1 t2 in*
(*monom-mult* (*1* / *punit.lc* (*rep-list p*)) (*l* − *t1*) *p*) −
(*monom-mult* (*1* / *punit.lc* (*rep-list q*)) (*l* − *t2*) *q*))

definition *is-regular-spair* :: $('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow$ *bool*
where *is-regular-spair p q* ⟷
(*rep-list p* ≠ *0* ∧ *rep-list q* ≠ *0* ∧
(*let t1* = *punit.lt* (*rep-list p*); *t2* = *punit.lt* (*rep-list q*); *l* = *lcs t1 t2 in*
(*l* − *t1*) ⊕ *lt p* ≠ (*l* − *t2*) ⊕ *lt q*))

lemma *rep-list-spair*: *rep-list* (*spair p q*) = *punit.spoly* (*rep-list p*) (*rep-list q*)
by (*simp add*: *spair-def punit.spoly-def Let-def rep-list-minus rep-list-monom-mult punit.lc-def*)

lemma *spair-comm*: *spair p q* = − *spair q p*
by (*simp add*: *spair-def Let-def lcs-comm*)

lemma *dgrad-sig-set-closed-spair*:
assumes *dickson-grading d* **and** *p* ∈ *dgrad-sig-set d* **and** *q* ∈ *dgrad-sig-set d*
shows *spair p q* ∈ *dgrad-sig-set d*
proof −
define *t1* **where** *t1* = *punit.lt* (*rep-list p*)
define *t2* **where** *t2* = *punit.lt* (*rep-list q*)
let *?l* = *lcs t1 t2*
have *d t1* ≤ *dgrad-max d*
proof (*cases rep-list p* = *0*)
case *True*
show *?thesis* **by** (*simp add*: *t1-def True dgrad-max-0*)
next
case *False*
from *assms*(*2*) **have** *p* ∈ *dgrad-max-set d* **by** (*simp add*: *dgrad-sig-set'-def*)
with *assms*(*1*) **have** *rep-list p* ∈ *punit-dgrad-max-set d* **by** (*rule dgrad-max-2*)
thus *?thesis* **unfolding** *t1-def* **using** *False* **by** (*rule punit.dgrad-p-setD-lp*[*simplified*])
qed
moreover have *d t2* ≤ *dgrad-max d*
proof (*cases rep-list q* = *0*)
case *True*
show *?thesis* **by** (*simp add*: *t2-def True dgrad-max-0*)
next
case *False*
from *assms*(*3*) **have** *q* ∈ *dgrad-max-set d* **by** (*simp add*: *dgrad-sig-set'-def*)
with *assms*(*1*) **have** *rep-list q* ∈ *punit-dgrad-max-set d* **by** (*rule dgrad-max-2*)
thus *?thesis* **unfolding** *t2-def* **using** *False* **by** (*rule punit.dgrad-p-setD-lp*[*simplified*])
qed
ultimately have *ord-class.max* (*d t1*) (*d t2*) ≤ *dgrad-max d* **by** *simp*

moreover from *assms*(*1*) **have** *d ?l* ≤ *ord-class.max* (*d t1*) (*d t2*) **by** (*rule dickson-grading-lcs*)
ultimately have ∗: *d ?l* ≤ *dgrad-max d* **by** *auto*
thm *dickson-grading-minus*
show *?thesis*
proof (*simp add*: *spair-def Let-def t1-def*[*symmetric*] *t2-def*[*symmetric*],
intro dgrad-sig-set-closed-minus dgrad-sig-set-closed-monom-mult[*OF assms*(*1*)])
from *assms*(*1*) *adds-lcs* **have** *d* (*?l* − *t1*) ≤ *d ?l* **by** (*rule dickson-grading-minus*)
thus *d* (*?l* − *t1*) ≤ *dgrad-max d* **using** ∗ **by** (*rule le-trans*)
next
from *assms*(*1*) *adds-lcs-2* **have** *d* (*?l* − *t2*) ≤ *d ?l* **by** (*rule dickson-grading-minus*)
thus *d* (*?l* − *t2*) ≤ *dgrad-max d* **using** ∗ **by** (*rule le-trans*)
qed *fact*+
qed

lemma *lt-spair*:
assumes *rep-list p* ≠ *0* **and** *punit.lt* (*rep-list p*) ⊕ *lt q* $\prec_t$ *punit.lt* (*rep-list q*) ⊕ *lt p*
shows *lt* (*spair p q*) = (*lcs* (*punit.lt* (*rep-list p*)) (*punit.lt* (*rep-list q*)) − *punit.lt* (*rep-list p*)) ⊕ *lt p*
proof −
define *l* **where** *l* = *lcs* (*punit.lt* (*rep-list p*)) (*punit.lt* (*rep-list q*))
have *1*: *punit.lt* (*rep-list p*) *adds l* **and** *2*: *punit.lt* (*rep-list q*) *adds l*
unfolding *l-def* **by** (*rule adds-lcs*, *rule adds-lcs-2*)
have *eq1*: *spair p q* = (*monom-mult* (*1* / *punit.lc* (*rep-list p*)) (*l* − *punit.lt* (*rep-list p*)) *p*) +
(*monom-mult* (− *1* / *punit.lc* (*rep-list q*)) (*l* − *punit.lt* (*rep-list q*)) *q*)
by (*simp add*: *spair-def Let-def l-def monom-mult-uminus-left*)
from *assms*(*1*) **have** *punit.lc* (*rep-list p*) ≠ *0* **by** (*rule punit.lc-not-0*)
hence *1* / *punit.lc* (*rep-list p*) ≠ *0* **by** *simp*
moreover from *assms*(*1*) **have** *p* ≠ *0* **by** (*auto simp*: *rep-list-zero*)
ultimately have *eq2*: *lt* (*monom-mult* (*1* / *punit.lc* (*rep-list p*)) (*l* − *punit.lt* (*rep-list p*)) *p*) =
(*l* − *punit.lt* (*rep-list p*)) ⊕ *lt p*
by (*rule lt-monom-mult*)
have *lt* (*monom-mult* (− *1* / *punit.lc* (*rep-list q*)) (*l* − *punit.lt* (*rep-list q*)) *q*) $\preceq_t$
(*l* − *punit.lt* (*rep-list q*)) ⊕ *lt q*
by (*fact lt-monom-mult-le*)
also from *assms*(*2*) **have** ... $\prec_t$ (*l* − *punit.lt* (*rep-list p*)) ⊕ *lt p*
by (*simp add*: *term-is-le-rel-minus-minus*[*OF - 2 1*])
finally show *?thesis* **unfolding** *eq2*[*symmetric*] *eq1 l-def*[*symmetric*] **by** (*rule lt-plus-eqI-2*)
qed

lemma *lt-spair'*:
assumes *rep-list p* ≠ *0* **and** *a* + *punit.lt* (*rep-list p*) = *b* + *punit.lt* (*rep-list q*) **and** *b* ⊕ *lt q* $\prec_t$ *a* ⊕ *lt p*

shows *lt* (*spair p q*) = (*a* − *gcs a b*) ⊕ *lt p*
proof −
from *assms*(*3*) **have** *punit.lt* (*rep-list p*) ⊕ (*b* ⊕ *lt q*) $\prec_t$ *punit.lt* (*rep-list p*) ⊕ (*a* ⊕ *lt p*)
by (*fact splus-mono-strict*)
hence (*b* + *punit.lt* (*rep-list p*)) ⊕ *lt q* $\prec_t$ (*b* + *punit.lt* (*rep-list q*)) ⊕ *lt p*
by (*simp only*: *splus-assoc*[*symmetric*] *add.commute assms*(*2*))
hence *punit.lt* (*rep-list p*) ⊕ *lt q* $\prec_t$ *punit.lt* (*rep-list q*) ⊕ *lt p*
by (*simp only*: *splus-assoc ord-term-strict-canc*)
with *assms*(*1*)
have *lt* (*spair p q*) = (*lcs* (*punit.lt* (*rep-list p*)) (*punit.lt* (*rep-list q*)) − *punit.lt* (*rep-list p*)) ⊕ *lt p*
by (*rule lt-spair*)
with *assms*(*2*) **show** *?thesis* **by** (*simp add*: *lcs-minus-1*)
qed

lemma *lt-rep-list-spair*:
assumes *rep-list p* ≠ *0* **and** *rep-list q* ≠ *0* **and** *rep-list* (*spair p q*) ≠ *0*
and *a* + *punit.lt* (*rep-list p*) = *b* + *punit.lt* (*rep-list q*)
shows *punit.lt* (*rep-list* (*spair p q*)) ≺ (*a* − *gcs a b*) + *punit.lt* (*rep-list p*)
proof −
from *assms*(*1*) **have** *1*: *punit.lc* (*rep-list p*) ≠ *0* **by** (*rule punit.lc-not-0*)
from *assms*(*2*) **have** *2*: *punit.lc* (*rep-list q*) ≠ *0* **by** (*rule punit.lc-not-0*)
define *l* **where** *l* = *lcs* (*punit.lt* (*rep-list p*)) (*punit.lt* (*rep-list q*))
have *eq*: *rep-list* (*spair p q*) = (*punit.monom-mult* (*1* / *punit.lc* (*rep-list p*)) (*l* − *punit.lt* (*rep-list p*)) (*rep-list p*)) +
(*punit.monom-mult* (− *1* / *punit.lc* (*rep-list q*)) (*l* − *punit.lt* (*rep-list q*)) (*rep-list q*))
(**is** - = *?a* + *?b*)
by (*simp add*: *spair-def Let-def rep-list-minus rep-list-monom-mult punit.monom-mult-uminus-left l-def*)
have *?a* + *?b* ≠ *0* **unfolding** *eq*[*symmetric*] **by** (*fact assms*(*3*))
moreover from *1 2 assms*(*1*, *2*) **have** *punit.lt ?b* = *punit.lt ?a*
by (*simp add*: *lp-monom-mult l-def minus-plus adds-lcs adds-lcs-2*)
moreover have *punit.lc ?b* = − *punit.lc ?a* **by** (*simp add*: *1 2*)
ultimately have *punit.lt* (*rep-list* (*spair p q*)) ≺ *punit.lt ?a* **unfolding** *eq* **by** (*rule punit.lt-plus-lessI*)
also from *1 assms*(*1*) **have** ... = (*l* − *punit.lt* (*rep-list p*)) + *punit.lt* (*rep-list p*) **by** (*simp add*: *lp-monom-mult*)
also have ... = *l* **by** (*simp add*: *l-def minus-plus adds-lcs*)
also have ... = (*a* + *punit.lt* (*rep-list p*)) − *gcs a b* **unfolding** *l-def* **using** *assms*(*4*) **by** (*rule lcs-alt-1*)
also have ... = (*a* − *gcs a b*) + *punit.lt* (*rep-list p*) **by** (*simp add*: *minus-plus gcs-adds*)
finally show *?thesis* **.**
qed

lemma *is-regular-spair-sym*: *is-regular-spair p q* ⟹ *is-regular-spair q p*
by (*auto simp*: *is-regular-spair-def Let-def lcs-comm*)

lemma *is-regular-spairI*:
  assumes *rep-list p* $\neq$ *0* **and** *rep-list q* $\neq$ *0*
  and *punit.lt* (*rep-list q*) $\oplus$ *lt p* $\neq$ *punit.lt* (*rep-list p*) $\oplus$ *lt q*
  shows *is-regular-spair p q*
proof −
  have $*$: (*lcs* (*punit.lt* (*rep-list p*)) (*punit.lt* (*rep-list q*)) − *punit.lt* (*rep-list p*)) $\oplus$ *lt p* $\neq$
      (*lcs* (*punit.lt* (*rep-list p*)) (*punit.lt* (*rep-list q*)) − *punit.lt* (*rep-list q*)) $\oplus$ *lt q*
    (**is** *?l* $\neq$ *?r*)
  proof
    assume *?l* = *?r*
    hence *punit.lt* (*rep-list q*) $\oplus$ *lt p* = *punit.lt* (*rep-list p*) $\oplus$ *lt q*
      by (*simp add*: *term-is-le-rel-minus-minus adds-lcs adds-lcs-2*)
    with *assms*(*3*) **show** *False* **..**
  qed
  with *assms*(*1*, *2*) **show** *?thesis* **by** (*simp add*: *is-regular-spair-def*)
qed

lemma *is-regular-spairI′*:
  assumes *rep-list p* $\neq$ *0* **and** *rep-list q* $\neq$ *0*
  and *a* + *punit.lt* (*rep-list p*) = *b* + *punit.lt* (*rep-list q*) **and** *a* $\oplus$ *lt p* $\neq$ *b* $\oplus$ *lt q*
  shows *is-regular-spair p q*
proof −
  have *punit.lt* (*rep-list q*) $\oplus$ *lt p* $\neq$ *punit.lt* (*rep-list p*) $\oplus$ *lt q*
  proof
    assume *punit.lt* (*rep-list q*) $\oplus$ *lt p* = *punit.lt* (*rep-list p*) $\oplus$ *lt q*
    hence *a* $\oplus$ (*punit.lt* (*rep-list q*) $\oplus$ *lt p*) = *a* $\oplus$ (*punit.lt* (*rep-list p*) $\oplus$ *lt q*) **by** (*simp only*:)
    hence (*a* + *punit.lt* (*rep-list q*)) $\oplus$ *lt p* = (*b* + *punit.lt* (*rep-list q*)) $\oplus$ *lt q*
      by (*simp add*: *splus-assoc*[*symmetric*] *assms*(*3*))
    hence *punit.lt* (*rep-list q*) $\oplus$ (*a* $\oplus$ *lt p*) = *punit.lt* (*rep-list q*) $\oplus$ (*b* $\oplus$ *lt q*)
      by (*simp only*: *add.commute*[*of* - *punit.lt* (*rep-list q*)] *splus-assoc*)
    hence *a* $\oplus$ *lt p* = *b* $\oplus$ *lt q* **by** (*simp only*: *splus-left-canc*)
    with *assms*(*4*) **show** *False* **..**
  qed
  with *assms*(*1*, *2*) **show** *?thesis* **by** (*rule is-regular-spairI*)
qed

lemma *is-regular-spairD1*: *is-regular-spair p q* $\Longrightarrow$ *rep-list p* $\neq$ *0*
  by (*simp add*: *is-regular-spair-def*)

lemma *is-regular-spairD2*: *is-regular-spair p q* $\Longrightarrow$ *rep-list q* $\neq$ *0*
  by (*simp add*: *is-regular-spair-def*)

lemma *is-regular-spairD3*:
  fixes *p q*
  defines *t1* $\equiv$ *punit.lt* (*rep-list p*)

defines *t2* ≡ *punit.lt* (*rep-list q*)
assumes *is-regular-spair p q*
shows *t2* ⊕ *lt p* ≠ *t1* ⊕ *lt q* (**is** *?thesis1*)
and *lt* (*monom-mult* (*1* / *punit.lc* (*rep-list p*)) (*lcs t1 t2* − *t1*) *p*) ≠
lt (*monom-mult* (*1* / *punit.lc* (*rep-list q*)) (*lcs t1 t2* − *t2*) *q*) (**is** *?l* ≠ *?r*)
proof −
from *assms*(*3*) **have** *rep-list p* ≠ *0* **by** (*rule is-regular-spairD1*)
hence *punit.lc* (*rep-list p*) ≠ *0* **and** *p* ≠ *0* **by** (*auto simp*: *rep-list-zero punit.lc-eq-zero-iff*)
from *assms*(*3*) **have** *rep-list q* ≠ *0* **by** (*rule is-regular-spairD2*)
hence *punit.lc* (*rep-list q*) ≠ *0* **and** *q* ≠ *0* **by** (*auto simp*: *rep-list-zero punit.lc-eq-zero-iff*)

have *?l* = (*lcs t1 t2* − *t1*) ⊕ *lt p*
using ‹*punit.lc* (*rep-list p*) ≠ *0*› ‹*p* ≠ *0*› **by** (*simp add*: *lt-monom-mult*)
also from *assms*(*3*) **have** ∗: ... ≠ (*lcs t1 t2* − *t2*) ⊕ *lt q*
by (*simp add*: *is-regular-spair-def t1-def t2-def Let-def*)
also have (*lcs t1 t2* − *t2*) ⊕ *lt q* = *?r*
using ‹*punit.lc* (*rep-list q*) ≠ *0*› ‹*q* ≠ *0*› **by** (*simp add*: *lt-monom-mult*)
finally show *?l* ≠ *?r* **.**

show *?thesis1*
proof
assume *t2* ⊕ *lt p* = *t1* ⊕ *lt q*
hence (*lcs t1 t2* − *t1*) ⊕ *lt p* = (*lcs t1 t2* − *t2*) ⊕ *lt q*
by (*simp add*: *term-is-le-rel-minus-minus adds-lcs adds-lcs-2*)
with ∗ **show** *False* **..**
qed
qed

lemma *is-regular-spair-nonzero*: *is-regular-spair p q* ⟹ *spair p q* ≠ *0*
by (*auto simp*: *spair-def Let-def dest*: *is-regular-spairD3*)

lemma *is-regular-spair-lt*:
assumes *is-regular-spair p q*
shows *lt* (*spair p q*) = *ord-term-lin.max*
((*lcs* (*punit.lt* (*rep-list p*)) (*punit.lt* (*rep-list q*)) − *punit.lt* (*rep-list p*))
⊕ *lt p*)
((*lcs* (*punit.lt* (*rep-list p*)) (*punit.lt* (*rep-list q*)) − *punit.lt* (*rep-list q*))
⊕ *lt q*)
proof −
let *?t1* = *punit.lt* (*rep-list p*)
let *?t2* = *punit.lt* (*rep-list q*)
let *?l* = *lcs ?t1 ?t2*
show *?thesis*
proof (*rule ord-term-lin.linorder-cases*)
assume *a*: *?t2* ⊕ *lt p* $\prec_t$ *?t1* ⊕ *lt q*
hence (*?l* − *?t1*) ⊕ *lt p* $\prec_t$ (*?l* − *?t2*) ⊕ *lt q*
by (*simp add*: *term-is-le-rel-minus-minus adds-lcs adds-lcs-2*)
hence *le*: (*?l* − *?t1*) ⊕ *lt p* $\preceq_t$ (*?l* − *?t2*) ⊕ *lt q* **by** (*rule ord-term-lin.less-imp-le*)
from *assms* **have** *rep-list q* ≠ *0* **by** (*rule is-regular-spairD2*)

have *lt* (*spair p q*) = *lt* (*spair q p*) **by** (*simp add*: *spair-comm*[*of p*])
also from ‹*rep-list q* ≠ *0*› *a* **have** ... = (*lcs ?t2 ?t1* − *?t2*) ⊕ *lt q* **by** (*rule lt-spair*)
also have ... = (*?l* − *?t2*) ⊕ *lt q* **by** (*simp only*: *lcs-comm*)
finally show *?thesis* **using** *le* **by** (*simp add*: *ord-term-lin.max-def*)
next
assume *a*: *?t1* ⊕ *lt q* $\prec_t$ *?t2* ⊕ *lt p*
hence (*?l* − *?t2*) ⊕ *lt q* $\prec_t$ (*?l* − *?t1*) ⊕ *lt p*
by (*simp add*: *term-is-le-rel-minus-minus adds-lcs adds-lcs-2*)
hence *le*: ¬ ((*?l* − *?t1*) ⊕ *lt p* $\preceq_t$ (*?l* − *?t2*) ⊕ *lt q*) **by** *simp*
from *assms* **have** *rep-list p* ≠ *0* **by** (*rule is-regular-spairD1*)
hence *lt* (*spair p q*) = (*lcs ?t1 ?t2* − *?t1*) ⊕ *lt p* **using** *a* **by** (*rule lt-spair*)
thus *?thesis* **using** *le* **by** (*simp add*: *ord-term-lin.max-def*)
next
from *assms* **have** *?t2* ⊕ *lt p* ≠ *?t1* ⊕ *lt q* **by** (*rule is-regular-spairD3*)
moreover assume *?t2* ⊕ *lt p* = *?t1* ⊕ *lt q*
ultimately show *?thesis* **..**
qed
qed

lemma *is-regular-spair-lt-ge-1*:
assumes *is-regular-spair p q*
shows *lt p* $\preceq_t$ *lt* (*spair p q*)
proof −
have *lt p* = *0* ⊕ *lt p* **by** (*simp only*: *term-simps*)
also from *zero-min* **have** ... $\preceq_t$ (*lcs* (*punit.lt* (*rep-list p*)) (*punit.lt* (*rep-list q*)) − *punit.lt* (*rep-list p*)) ⊕ *lt p*
by (*rule splus-mono-left*)
also have ... $\preceq_t$ *ord-term-lin.max*
((*lcs* (*punit.lt* (*rep-list p*)) (*punit.lt* (*rep-list q*)) − *punit.lt* (*rep-list p*)) ⊕ *lt p*)
((*lcs* (*punit.lt* (*rep-list p*)) (*punit.lt* (*rep-list q*)) − *punit.lt* (*rep-list q*)) ⊕ *lt q*)
by (*rule ord-term-lin.max.cobounded1*)
also from *assms* **have** ... = *lt* (*spair p q*) **by** (*simp only*: *is-regular-spair-lt*)
finally show *?thesis* **.**
qed

corollary *is-regular-spair-lt-ge-2*:
assumes *is-regular-spair p q*
shows *lt q* $\preceq_t$ *lt* (*spair p q*)
proof −
from *assms* **have** *is-regular-spair q p* **by** (*rule is-regular-spair-sym*)
hence *lt q* $\preceq_t$ *lt* (*spair q p*) **by** (*rule is-regular-spair-lt-ge-1*)
also have ... = *lt* (*spair p q*) **by** (*simp add*: *spair-comm*[*of q*])
finally show *?thesis* **.**
qed

lemma *is-regular-spair-component-lt-cases*:

assumes *is-regular-spair p q*
shows *component-of-term* (*lt* (*spair p q*)) = *component-of-term* (*lt p*) ∨
component-of-term (*lt* (*spair p q*)) = *component-of-term* (*lt q*)
proof (*rule ord-term-lin.linorder-cases*)
from *assms* **have** *rep-list q* ≠ *0* **by** (*rule is-regular-spairD2*)
moreover assume *punit.lt* (*rep-list q*) ⊕ *lt p* $\prec_t$ *punit.lt* (*rep-list p*) ⊕ *lt q*
ultimately have *lt* (*spair q p*) = (*lcs* (*punit.lt* (*rep-list q*)) (*punit.lt* (*rep-list p*)) − *punit.lt* (*rep-list q*)) ⊕ *lt q*
by (*rule lt-spair*)
thus *?thesis* **by** (*simp add*: *spair-comm*[*of p*] *term-simps*)
next
from *assms* **have** *rep-list p* ≠ *0* **by** (*rule is-regular-spairD1*)
moreover assume *punit.lt* (*rep-list p*) ⊕ *lt q* $\prec_t$ *punit.lt* (*rep-list q*) ⊕ *lt p*
ultimately have *lt* (*spair p q*) = (*lcs* (*punit.lt* (*rep-list p*)) (*punit.lt* (*rep-list q*)) − *punit.lt* (*rep-list p*)) ⊕ *lt p*
by (*rule lt-spair*)
thus *?thesis* **by** (*simp add*: *term-simps*)
next
from *assms* **have** *punit.lt* (*rep-list q*) ⊕ *lt p* ≠ *punit.lt* (*rep-list p*) ⊕ *lt q*
by (*rule is-regular-spairD3*)
moreover assume *punit.lt* (*rep-list q*) ⊕ *lt p* = *punit.lt* (*rep-list p*) ⊕ *lt q*
ultimately show *?thesis* **..**
qed

lemma *lemma-9*:
assumes *dickson-grading d* **and** *is-rewrite-ord rword* **and** *is-RB-upt d rword G u*
and *inj-on lt G* **and** ¬ *is-syz-sig d u* **and** *is-canon-rewriter rword G u g1* **and** *h* ∈ *G*
and *is-sig-red* ($\prec_t$) (=) {*h*} (*monom-mult 1* (*pp-of-term u* − *lp g1*) *g1*)
and *d* (*pp-of-term u*) ≤ *dgrad-max d*
shows *lcs* (*punit.lt* (*rep-list g1*)) (*punit.lt* (*rep-list h*)) − *punit.lt* (*rep-list g1*) =
pp-of-term u − *lp g1* (**is** *?thesis1*)
and *lcs* (*punit.lt* (*rep-list g1*)) (*punit.lt* (*rep-list h*)) − *punit.lt* (*rep-list h*) =
((*pp-of-term u* − *lp g1*) + *punit.lt* (*rep-list g1*)) − *punit.lt* (*rep-list h*) (**is** *?thesis2*)
and *is-regular-spair g1 h* (**is** *?thesis3*)
and *lt* (*spair g1 h*) = *u* (**is** *?thesis4*)
proof −
from *assms*(*8*) **have** *rep-list* (*monom-mult 1* (*pp-of-term u* − *lp g1*) *g1*) ≠ *0*
using *is-sig-red-top-addsE* **by** *fastforce*
hence *rep-list g1* ≠ *0* **by** (*simp add*: *rep-list-monom-mult punit.monom-mult-eq-zero-iff*)
hence *g1* ≠ *0* **by** (*auto simp*: *rep-list-zero*)
from *assms*(*6*) **have** *g1* ∈ *G* **and** *lt g1* $adds_t$ *u* **by** (*rule is-canon-rewriterD*)+
from *assms*(*3*) **have** *G* ⊆ *dgrad-sig-set d* **by** (*rule is-RB-uptD1*)
with ‹*g1* ∈ *G*› **have** *g1* ∈ *dgrad-sig-set d* **..**
hence *component-of-term* (*lt g1*) < *length fs* **using** ‹*g1* ≠ *0*› **by** (*rule dgrad-sig-setD-lt*)
with ‹*lt g1* $adds_t$ *u*› **have** *component-of-term u* < *length fs* **by** (*simp add*: *adds-term-def*)

from ‹*lt g1 adds*$_t$ *u*› **obtain** *a* **where** *u*: *u* = *a* ⊕ *lt g1* **by** (*rule adds-termE*)
hence *a*: *a* = *pp-of-term u* − *lp g1* **by** (*simp add*: *term-simps*)
from *assms*(*8*) **have** *is-sig-red* (≺$_t$) (=) {*h*} (*monom-mult 1 a g1*) **by** (*simp only*: *a*)
hence *rep-list h* ≠ *0* **and** *rep-list* (*monom-mult 1 a g1*) ≠ *0* **and**
2: *punit.lt* (*rep-list h*) *adds punit.lt* (*rep-list* (*monom-mult 1 a g1*)) **and**
3: *punit.lt* (*rep-list* (*monom-mult 1 a g1*)) ⊕ *lt h* ≺$_t$ *punit.lt* (*rep-list h*) ⊕ *lt* (*monom-mult 1 a g1*)
by (*auto elim*: *is-sig-red-top-addsE*)
from *this*(*2*) **have** *rep-list g1* ≠ *0* **by** (*simp add*: *rep-list-monom-mult punit.monom-mult-eq-zero-iff*)
hence *g1* ≠ *0* **by** (*auto simp*: *rep-list-zero*)
from ‹*rep-list h* ≠ *0*› **have** *h* ≠ *0* **by** (*auto simp*: *rep-list-zero*)
from *2* ‹*rep-list g1* ≠ *0*› **have** *punit.lt* (*rep-list h*) *adds a* + *punit.lt* (*rep-list g1*)
by (*simp add*: *rep-list-monom-mult punit.lt-monom-mult*)
then obtain *b* **where** *eq1*: *a* + *punit.lt* (*rep-list g1*) = *b* + *punit.lt* (*rep-list h*)
by (*auto elim*: *addsE simp*: *add.commute*)
hence *b*: *b* = ((*pp-of-term u* − *lp g1*) + *punit.lt* (*rep-list g1*)) − *punit.lt* (*rep-list h*)
by (*simp add*: *a*)

define *g* **where** *g* = *gcs a b*
have *g* = *0*
proof (*rule ccontr*)
assume *g* ≠ *0*
have *g adds a* **unfolding** *g-def* **by** (*fact gcs-adds*)
also have ... *adds*$_p$ *u* **unfolding** *u* **by** (*fact adds-pp-triv*)
finally obtain *v* **where** *u2*: *u* = *g* ⊕ *v* **by** (*rule adds-ppE*)
hence *v*: *v* = *u* ⊖ *g* **by** (*simp add*: *term-simps*)
from *u2* **have** *v adds*$_t$ *u* **by** (*rule adds-termI*)
hence *v* ⪯$_t$ *u* **by** (*rule ord-adds-term*)
moreover have *v* ≠ *u*
proof
assume *v* = *u*
hence *g* ⊕ *v* = *0* ⊕ *v* **by** (*simp add*: *u2 term-simps*)
hence *g* = *0* **by** (*simp only*: *splus-right-canc*)
with ‹*g* ≠ *0*› **show** *False* **..**
qed
ultimately have *v* ≺$_t$ *u* **by** *simp*
note *assms*(*3*) ‹*v* ≺$_t$ *u*›
moreover have *d* (*pp-of-term v*) ≤ *dgrad-max d*
proof (*rule le-trans*)
from *assms*(*1*) **show** *d* (*pp-of-term v*) ≤ *d* (*pp-of-term u*)
by (*simp add*: *u2 term-simps dickson-gradingD1*)
qed *fact*
moreover from ‹*component-of-term u* < *length fs*› **have** *component-of-term v* < *length fs*
by (*simp only*: *v term-simps*)
ultimately have *is-RB-in d rword G v* **by** (*rule is-RB-uptD2*)

thus *False*
proof (*rule is-RB-inE*)
assume *is-syz-sig d v*
with *assms*(*1*) **have** *is-syz-sig d u* **using** ‹*v* $adds_t$ *u*› *assms*(*9*) **by** (*rule is-syz-sig-adds*)
with *assms*(*5*) **show** *False* ..
next
fix *g2*
assume ∗: ¬ *is-sig-red* ($\prec_t$) (=) *G* (*monom-mult 1* (*pp-of-term v* − *lp g2*) *g2*)
assume *is-canon-rewriter rword G v g2*
hence *g2* ∈ *G* **and** *g2* ≠ *0* **and** *lt g2* $adds_t$ *v* **by** (*rule is-canon-rewriterD*)+
assume ¬ *is-syz-sig d v*
note *assms*(*2*) ‹*is-canon-rewriter rword G v g2*› *assms*(*6*)
moreover from ‹*lt g2* $adds_t$ *v*› ‹*v* $adds_t$ *u*› **have** *lt g2* $adds_t$ *u* **by** (*rule adds-term-trans*)
moreover from ‹*g adds a*› **have** *lt g1* $adds_t$ *v* **by** (*simp add*: *v u minus-splus*[*symmetric*] *adds-termI*)
ultimately have *lt g2* = *lt g1* **by** (*rule is-rewrite-ord-canon-rewriterD1*)
with *assms*(*4*) **have** *g2* = *g1* **using** ‹*g2* ∈ *G*› ‹*g1* ∈ *G*› **by** (*rule inj-onD*)
have *pp-of-term v* − *lp g1* = *a* − *g* **by** (*simp add*: *u v term-simps diff-diff-add*)

have *is-sig-red* ($\prec_t$) (=) *G* (*monom-mult 1* (*pp-of-term v* − *lp g2*) *g2*)
unfolding ‹*g2* = *g1*› ‹*pp-of-term v* − *lp g1* = *a* − *g*› **using** *assms*(*7*) ‹*rep-list h* ≠ *0*›
proof (*rule is-sig-red-top-addsI*)
from ‹*rep-list g1* ≠ *0*› **show** *rep-list* (*monom-mult 1* (*a* − *g*) *g1*) ≠ *0*
by (*simp add*: *rep-list-monom-mult punit.monom-mult-eq-zero-iff*)
next
have *eq3*: (*a* − *g*) + *punit.lt* (*rep-list g1*) = *lcs* (*punit.lt* (*rep-list g1*)) (*punit.lt* (*rep-list h*))
by (*simp add*: *g-def lcs-minus-1*[*OF eq1*, *symmetric*] *adds-minus adds-lcs*)
from ‹*rep-list g1* ≠ *0*›
show *punit.lt* (*rep-list h*) *adds punit.lt* (*rep-list* (*monom-mult 1* (*a* − *g*) *g1*))
by (*simp add*: *rep-list-monom-mult punit.lt-monom-mult eq3 adds-lcs-2*)
next
from *3* ‹*rep-list g1* ≠ *0*› ‹*g1* ≠ *0*›
show *punit.lt* (*rep-list* (*monom-mult 1* (*a* − *g*) *g1*)) ⊕ *lt h* $\prec_t$
punit.lt (*rep-list h*) ⊕ *lt* (*monom-mult 1* (*a* − *g*) *g1*)
by (*auto simp*: *rep-list-monom-mult punit.lt-monom-mult lt-monom-mult splus-assoc splus-left-commute*
dest!: *ord-term-strict-canc intro*: *splus-mono-strict*)
next
show *ord-term-lin.is-le-rel* ($\prec_t$) **by** (*fact ord-term-lin.is-le-relI*)
qed
with ∗ **show** *False* ..
qed
qed
thus *?thesis1* **and** *?thesis2* **by** (*simp-all add*: *a b lcs-minus-1*[*OF eq1*] *lcs-minus-2*[*OF*

eq1] *g-def*)
hence *eq3*: *spair g1 h* = *monom-mult* (*1* / *punit.lc* (*rep-list g1*)) *a g1* −
monom-mult (*1* / *punit.lc* (*rep-list h*)) *b h*
by (*simp add*: *spair-def Let-def a b*)

from *3* ‹*rep-list g1* ≠ *0*› ‹*g1* ≠ *0*› **have** $b \oplus$ *lt h* $\prec_t$ $a \oplus$ *lt g1*
by (*auto simp*: *rep-list-monom-mult punit.lt-monom-mult lt-monom-mult eq1 splus-assoc*
splus-left-commute[*of b*] *dest*!: *ord-term-strict-canc*)
hence $a \oplus$ *lt g1* ≠ $b \oplus$ *lt h* **by** *simp*
with ‹*rep-list g1* ≠ *0*› ‹*rep-list h* ≠ *0*› *eq1* **show** *?thesis3*
by (*rule is-regular-spairI′*)

have *lt* (*monom-mult* (*1* / *punit.lc* (*rep-list h*)) *b h*) = $b \oplus$ *lt h*
proof (*rule lt-monom-mult*)
from ‹*rep-list h* ≠ *0*› **show** *1* / *punit.lc* (*rep-list h*) ≠ *0* **by** (*simp add*: *punit.lc-eq-zero-iff*)
qed *fact*
also have ... $\prec_t$ $a \oplus$ *lt g1* **by** *fact*
also have ... = *lt* (*monom-mult* (*1* / *punit.lc* (*rep-list g1*)) *a g1*)
proof (*rule HOL.sym*, *rule lt-monom-mult*)
from ‹*rep-list g1* ≠ *0*› **show** *1* / *punit.lc* (*rep-list g1*) ≠ *0* **by** (*simp add*: *punit.lc-eq-zero-iff*)
qed *fact*
finally have *lt* (*spair g1 h*) = *lt* (*monom-mult* (*1* / *punit.lc* (*rep-list g1*)) *a g1*)
unfolding *eq3* **by** (*rule lt-minus-eqI-2*)
also have ... = $a \oplus$ *lt g1* **by** (*rule HOL.sym*, *fact*)
finally show *?thesis4* **by** (*simp only*: *u*)
qed

lemma *is-RB-upt-finite*:
assumes *dickson-grading d* **and** *is-rewrite-ord rword* **and** *G* ⊆ *dgrad-sig-set d* **and** *inj-on lt G*
and *finite G*
and $\bigwedge$*g1 g2*. *g1* ∈ *G* ⟹ *g2* ∈ *G* ⟹ *is-regular-spair g1 g2* ⟹ *lt* (*spair g1 g2*) $\prec_t$ *u* ⟹
is-RB-in d rword G (*lt* (*spair g1 g2*))
and $\bigwedge$*i*. *i* < *length fs* ⟹ *term-of-pair* (*0*, *i*) $\prec_t$ *u* ⟹ *is-RB-in d rword G* (*term-of-pair* (*0*, *i*))
shows *is-RB-upt d rword G u*
proof (*rule ccontr*)
let *?Q* = {*v*. *v* $\prec_t$ *u* ∧ *d* (*pp-of-term v*) ≤ *dgrad-max d* ∧ *component-of-term v* < *length fs* ∧ ¬ *is-RB-in d rword G v*}
have *Q-sub*: *pp-of-term* ‘ *?Q* ⊆ *dgrad-set d* (*dgrad-max d*) **by** *blast*
from *assms*(*3*) **have** *G* ⊆ *dgrad-max-set d* **by** (*simp add*: *dgrad-sig-set′-def*)
assume ¬ *is-RB-upt d rword G u*
with *assms*(*3*) **obtain** *v′* **where** *v′* ∈ *?Q* **unfolding** *is-RB-upt-def* **by** *blast*
with *assms*(*1*) **obtain** *v* **where** *v* ∈ *?Q* **and** *min*: $\bigwedge$*y*. *y* $\prec_t$ *v* ⟹ *y* ∉ *?Q* **using** *Q-sub*

by (*rule ord-term-minimum-dgrad-set*, *blast*)
from ‹$v \in$ *?Q*› **have** $v \prec_t u$ **and** *d* (*pp-of-term v*) $\leq$ *dgrad-max d* **and** *component-of-term v* $<$ *length fs*
and $\neg$ *is-RB-in d rword G v* **by** *simp-all*
from *this*(*4*)
have *impl*: $\bigwedge g.\ g \in G \Longrightarrow$ *is-canon-rewriter rword G v g* $\Longrightarrow$
is-sig-red ($\prec_t$) (=) *G* (*monom-mult 1* (*pp-of-term v* $-$ *lp g*) *g*)
and $\neg$ *is-syz-sig d v* **by** (*simp-all add*: *is-RB-in-def Let-def*)

from *assms*(*3*) **have** *is-RB-upt d rword G v*
proof (*rule is-RB-uptI*)
fix *w*
assume *dw*: *d* (*pp-of-term w*) $\leq$ *dgrad-max d* **and** *cp*: *component-of-term w* $<$ *length fs*
assume $w \prec_t v$
hence $w \notin$ *?Q* **by** (*rule min*)
hence $\neg\ w \prec_t u \vee$ *is-RB-in d rword G w* **by** (*simp add*: *dw cp*)
thus *is-RB-in d rword G w*
proof
assume $\neg\ w \prec_t u$
moreover from ‹$w \prec_t v$› ‹$v \prec_t u$› **have** $w \prec_t u$ **by** (*rule ord-term-lin.less-trans*)
ultimately show *?thesis* **..**
qed
qed

show *False*
proof (*cases* $\exists g \in G.\ g \neq 0 \wedge$ *lt g* $adds_t$ *v*)
case *False*
hence *x*: $\bigwedge g.\ g \in G \Longrightarrow$ *lt g* $adds_t$ $v \Longrightarrow g = 0$ **by** *blast*
let *?w* = *term-of-pair* (*0*, *component-of-term v*)
have *?w* $adds_t$ *v* **by** (*simp add*: *adds-term-def term-simps*)
hence *?w* $\preceq_t v$ **by** (*rule ord-adds-term*)
also have ... $\prec_t u$ **by** *fact*
finally have *?w* $\prec_t u$ **.**
with ‹*component-of-term v* $<$ *length fs*› **have** *is-RB-in d rword G ?w* **by** (*rule assms*(*7*))
thus *?thesis*
proof (*rule is-RB-inE*)
assume *is-syz-sig d ?w*
with *assms*(*1*) **have** *is-syz-sig d v* **using** ‹*?w* $adds_t$ *v*› ‹*d* (*pp-of-term v*) $\leq$ *dgrad-max d*›
by (*rule is-syz-sig-adds*)
with ‹$\neg$ *is-syz-sig d v*› **show** *?thesis* **..**
next
fix *g1*
assume *is-canon-rewriter rword G ?w g1*
hence $g1 \neq 0$ **and** $g1 \in G$ **and** *lt g1* $adds_t$ *?w* **by** (*rule is-canon-rewriterD*)+
from *this*(*3*) **have** *lt g1* $adds_t$ *v* **using** ‹*?w* $adds_t$ *v*› **by** (*rule adds-term-trans*)
with ‹$g1 \in G$› **have** $g1 = 0$ **by** (*rule x*)

with ‹$g1 \neq 0$› **show** *?thesis* **..**
qed
next
case *True*
then obtain g' **where** $g' \in G$ **and** $g' \neq 0$ **and** *lt* g' *adds*$_t$ v **by** *blast*
with *assms*(*2*, *5*) **obtain** *g1* **where** *crw*: *is-canon-rewriter rword G v g1*
by (*rule is-rewrite-ord-finite-canon-rewriterE*)
hence $g1 \in G$ **by** (*rule is-canon-rewriterD1*)
hence *is-sig-red* ($\prec_t$) (=) *G* (*monom-mult 1* (*pp-of-term v* − *lp g1*) *g1*) **using** *crw* **by** (*rule impl*)
then obtain *h* **where** $h \in G$ **and** *is-sig-red* ($\prec_t$) (=) {*h*} (*monom-mult 1* (*pp-of-term v* − *lp g1*) *g1*)
by (*rule is-sig-red-singletonI*)
with *assms*(*1*, *2*) ‹*is-RB-upt d rword G v*› *assms*(*4*) ‹¬ *is-syz-sig d v*› *crw*
have *is-regular-spair g1 h* **and** *eq*: *lt* (*spair g1 h*) = *v*
using ‹*d* (*pp-of-term v*) ≤ *dgrad-max d*› **by** (*rule lemma-9*)+
from ‹$v \prec_t u$› **have** *lt* (*spair g1 h*) $\prec_t$ *u* **by** (*simp only*: *eq*)
with ‹$g1 \in G$› ‹$h \in G$› ‹*is-regular-spair g1 h*› **have** *is-RB-in d rword G* (*lt* (*spair g1 h*))
by (*rule assms*(*6*))
hence *is-RB-in d rword G v* **by** (*simp only*: *eq*)
with ‹¬ *is-RB-in d rword G v*› **show** *?thesis* **..**
qed
qed

Note that the following lemma actually holds for *all* regularly reducible power-products in *rep-list p*, not just for the leading power-product.

lemma *lemma-11*:
assumes *dickson-grading d* **and** *is-rewrite-ord rword* **and** *is-RB-upt d rword G* (*lt p*)
and $p \in$ *dgrad-sig-set d* **and** *is-sig-red* ($\prec_t$) (=) *G p*
obtains *u g* **where** $u \prec_t$ *lt p* **and** *d* (*pp-of-term u*) ≤ *dgrad-max d* **and** *component-of-term u* < *length fs*
and ¬ *is-syz-sig d u* **and** *is-canon-rewriter rword G u g*
and *u* = (*punit.lt* (*rep-list p*) − *punit.lt* (*rep-list g*)) ⊕ *lt g* **and** *is-sig-red* ($\prec_t$) (=) {*g*} *p*
proof −
from *assms*(*3*) **have** *G-sub*: *G* ⊆ *dgrad-sig-set d* **by** (*rule is-RB-uptD1*)
from *assms*(*5*) **have** *rep-list p* ≠ *0* **using** *is-sig-red-addsE* **by** *fastforce*
hence *p* ≠ *0* **by** (*auto simp*: *rep-list-zero*)
let *?lc* = *punit.lc* (*rep-list p*)
let *?lp* = *punit.lt* (*rep-list p*)
from ‹*rep-list p* ≠ *0*› **have** *?lc* ≠ *0* **by** (*rule punit.lc-not-0*)
from *assms*(*4*) **have** $p \in$ *dgrad-max-set d* **by** (*simp add*: *dgrad-sig-set′-def*)
from *assms*(*4*) **have** *d* (*lp p*) ≤ *dgrad-max d* **by** (*rule dgrad-sig-setD-lp*)
from *assms*(*4*) ‹*p* ≠ *0*› **have** *component-of-term* (*lt p*) < *length fs* **by** (*rule dgrad-sig-setD-lt*)
from *assms*(*1*) ‹$p \in$ *dgrad-max-set d*› **have** *rep-list p* ∈ *punit-dgrad-max-set d* **by** (*rule dgrad-max-2*)

hence *d ?lp* $\leq$ *dgrad-max d* **using** ‹*rep-list p* $\neq$ *0*› **by** (*rule punit.dgrad-p-setD-lp*[*simplified*])

from *assms*(*5*) **obtain** *g0* **where** *g0* $\in$ *G* **and** *is-sig-red* ($\prec_t$) (=) {*g0*} *p*
by (*rule is-sig-red-singletonI*)
from ‹*g0* $\in$ *G*› *G-sub* **have** *g0* $\in$ *dgrad-sig-set d* **..**
let *?g0* = *monom-mult* (*?lc* / *punit.lc* (*rep-list g0*)) (*?lp* − *punit.lt* (*rep-list g0*)) *g0*

define *M* **where** *M* = {*monom-mult* (*?lc* / *punit.lc* (*rep-list g*)) (*?lp* − *punit.lt* (*rep-list g*)) *g* |
g. *g* $\in$ *dgrad-sig-set d* $\wedge$ *is-sig-red* ($\prec_t$) (=) {*g*} *p*}
from ‹*g0* $\in$ *dgrad-sig-set d*› ‹*is-sig-red* ($\prec_t$) (=) {*g0*} *p*› **have** *?g0* $\in$ *M* **by** (*auto simp*: *M-def*)
have *0* $\notin$ *rep-list* ‘ *M*
proof
assume *0* $\in$ *rep-list* ‘ *M*
then obtain *g* **where** *1*: *is-sig-red* ($\prec_t$) (=) {*g*} *p*
and *2*: *rep-list* (*monom-mult* (*?lc* / *punit.lc* (*rep-list g*)) (*?lp* − *punit.lt* (*rep-list g*)) *g*) = *0*
unfolding *M-def* **by** *fastforce*
from *1* **have** *rep-list g* $\neq$ *0* **using** *is-sig-red-addsE* **by** *fastforce*
moreover from *this* **have** *punit.lc* (*rep-list g*) $\neq$ *0* **by** (*rule punit.lc-not-0*)
ultimately have *rep-list* (*monom-mult* (*?lc* / *punit.lc* (*rep-list g*)) (*?lp* − *punit.lt* (*rep-list g*)) *g*) $\neq$ *0*
using ‹*?lc* $\neq$ *0*› **by** (*simp add*: *rep-list-monom-mult punit.monom-mult-eq-zero-iff*)
thus *False* **using** *2* **..**
qed
with *rep-list-zero* **have** *0* $\notin$ *M* **by** *auto*
have *M* $\subseteq$ *dgrad-sig-set d*
proof
fix *m*
assume *m* $\in$ *M*
then obtain *g* **where** *g* $\in$ *dgrad-sig-set d* **and** *1*: *is-sig-red* ($\prec_t$) (=) {*g*} *p*
and *m*: *m* = *monom-mult* (*?lc* / *punit.lc* (*rep-list g*)) (*?lp* − *punit.lt* (*rep-list g*)) *g*
unfolding *M-def* **by** *fastforce*
from *1* **have** *punit.lt* (*rep-list g*) *adds ?lp* **using** *is-sig-red-top-addsE* **by** *fastforce*
note *assms*(*1*)
thm *dickson-grading-minus*
moreover have *d* (*?lp* − *punit.lt* (*rep-list g*)) $\leq$ *dgrad-max d*
by (*rule le-trans*, *rule dickson-grading-minus*, *fact+*)
ultimately show *m* $\in$ *dgrad-sig-set d* **unfolding** *m* **using** ‹*g* $\in$ *dgrad-sig-set d*›
by (*rule dgrad-sig-set-closed-monom-mult*)
qed
hence *M* $\subseteq$ *sig-inv-set* **by** (*simp add*: *dgrad-sig-set′-def*)

let *?M* = *lt* ‘ *M*

note *assms*(*1*)
moreover from ‹*?g0* ∈ *M*› **have** *lt ?g0* ∈ *?M* **by** (*rule imageI*)
moreover from ‹*M* ⊆ *dgrad-sig-set d*› **have** *pp-of-term* ‘ *?M* ⊆ *dgrad-set d* (*dgrad-max d*)
by (*auto intro*!: *dgrad-sig-setD-lp*)
ultimately obtain *u* **where** *u* ∈ *?M* **and** *min*: $\bigwedge v.\ v \prec_t u \Longrightarrow v \notin$ *?M*
by (*rule ord-term-minimum-dgrad-set*, *blast*)
from ‹*u* ∈ *?M*› **obtain** *m* **where** *m* ∈ *M* **and** *u'*: *u* = *lt m* **..**
from *this*(*1*) **obtain** *g1* **where** *g1* ∈ *dgrad-sig-set d* **and** *1*: *is-sig-red* ($\prec_t$) (=) {*g1*} *p*
and *m*: *m* = *monom-mult* (*?lc* / *punit.lc* (*rep-list g1*)) (*?lp* − *punit.lt* (*rep-list g1*)) *g1*
unfolding *M-def* **by** *fastforce*
from *1* **have** *adds*: *punit.lt* (*rep-list g1*) *adds ?lp* **and** *?lp* ⊕ *lt g1* $\prec_t$ *punit.lt* (*rep-list g1*) ⊕ *lt p*
and *rep-list g1* ≠ *0* **using** *is-sig-red-top-addsE* **by** *fastforce*+
from *this*(*3*) **have** *lc-g1*: *punit.lc* (*rep-list g1*) ≠ *0* **by** (*rule punit.lc-not-0*)
from ‹*m* ∈ *M*› ‹*0* ∉ *rep-list* ‘ *M*› **have** *rep-list m* ≠ *0* **by** *fastforce*
from ‹*m* ∈ *M*› ‹*0* ∉ *M*› **have** *m* ≠ *0* **by** *blast*
hence *lc m* ≠ *0* **by** (*rule lc-not-0*)
from *lc-g1* **have** *eq0*: *punit.lc* (*rep-list m*) = *?lc* **by** (*simp add*: *m rep-list-monom-mult*)
from ‹*?lc* ≠ *0*› ‹*rep-list g1* ≠ *0*› *adds* **have** *eq1*: *punit.lt* (*rep-list m*) = *?lp*
by (*simp add*: *m rep-list-monom-mult punit.lt-monom-mult punit.lc-eq-zero-iff adds-minus*)
from ‹*m* ∈ *M*› ‹*M* ⊆ *dgrad-sig-set d*› **have** *m* ∈ *dgrad-sig-set d* **..**

from ‹*rep-list g1* ≠ *0*› **have** *punit.lc* (*rep-list g1*) ≠ *0* **and** *g1* ≠ *0*
by (*auto simp*: *rep-list-zero punit.lc-eq-zero-iff*)
with ‹*?lc* ≠ *0*› **have** *u*: *u* = (*?lp* − *punit.lt* (*rep-list g1*)) ⊕ *lt g1*
by (*simp add*: *u' m lt-monom-mult lc-eq-zero-iff*)
hence *punit.lt* (*rep-list g1*) ⊕ *u* = *punit.lt* (*rep-list g1*) ⊕ ((*?lp* − *punit.lt* (*rep-list g1*)) ⊕ *lt g1*)
by *simp*
also from *adds* **have** ... = *?lp* ⊕ *lt g1*
by (*simp only*: *splus-assoc*[*symmetric*], *metis add.commute adds-minus*)
also have ... $\prec_t$ *punit.lt* (*rep-list g1*) ⊕ *lt p* **by** *fact*
finally have *u* $\prec_t$ *lt p* **by** (*rule ord-term-strict-canc*)

from ‹*u* ∈ *?M*› **have** *pp-of-term u* ∈ *pp-of-term* ‘ *?M* **by** (*rule imageI*)
also have ... ⊆ *dgrad-set d* (*dgrad-max d*) **by** *fact*
finally have *d* (*pp-of-term u*) ≤ *dgrad-max d* **by** (*rule dgrad-setD*)

from ‹*u* ∈ *?M*› **have** *component-of-term u* ∈ *component-of-term* ‘ *?M* **by** (*rule imageI*)
also from ‹*M* ⊆ *sig-inv-set*› ‹*0* ∉ *M*› *sig-inv-setD-lt* **have** ... ⊆ {*0*..<*length fs*} **by** *fastforce*
finally have *component-of-term u* < *length fs* **by** *simp*

have ¬ *is-syz-sig d u*

proof
assume *is-syz-sig d u*
then obtain s **where** $s \neq 0$ **and** *lt* $s = u$ **and** $s \in$ *dgrad-sig-set d* **and** *rep-list* $s = 0$
by (*rule is-syz-sigE*)
let $?s =$ *monom-mult* (*lc m / lc s*) *0 s*
have *rep-list* $?s = 0$ **by** (*simp add*: *rep-list-monom-mult* ‹*rep-list* $s = 0$›)
from ‹$s \neq 0$› **have** *lc* $s \neq 0$ **by** (*rule lc-not-0*)
hence *lc m / lc s* $\neq 0$ **using** ‹*lc* $m \neq 0$› **by** *simp*
have $m - ?s \neq 0$
proof
assume $m - ?s = 0$
hence $m = ?s$ **by** *simp*
with ‹*rep-list* $?s = 0$› **have** *rep-list* $m = 0$ **by** *simp*
with ‹*rep-list* $m \neq 0$› **show** *False* **..**
qed
moreover from ‹*lc m / lc s* $\neq 0$› **have** *lt* $?s =$ *lt m* **by** (*simp add*: *lt-monom-mult-zero* ‹*lt* $s = u$› u')
moreover from ‹*lc* $s \neq 0$› **have** *lc* $?s =$ *lc m* **by** *simp*
ultimately have *lt* $(m - ?s) \prec_t u$ **unfolding** u' **by** (*rule lt-minus-lessI*)
hence *lt* $(m - ?s) \notin ?M$ **by** (*rule min*)
hence $m - ?s \notin M$ **by** *blast*
moreover have $m - ?s \in M$
proof −
have $?s =$ *monom-mult* (*?lc / lc s*) *0* (*monom-mult* (*lc g1 / punit.lc* (*rep-list g1*)) *0 s*)
by (*simp add*: *m monom-mult-assoc mult.commute*)
define m' **where** $m' = m - ?s$
have *eq*: *rep-list* m' = *rep-list m* **by** (*simp add*: m'*-def rep-list-minus* ‹*rep-list* $?s = 0$›)
from ‹$?lc \neq 0$› **have** m' = *monom-mult* (*?lc / punit.lc* (*rep-list* m')) (*?lp* − *punit.lt* (*rep-list* m')) m'
by (*simp add*: *eq eq0 eq1*)
also have ... $\in M$ **unfolding** *M-def*
proof (*rule*, *intro exI conjI*)
from ‹$s \in$ *dgrad-sig-set d*› **have** $?s \in$ *dgrad-sig-set d*
by (*rule dgrad-sig-set-closed-monom-mult-zero*)
with ‹$m \in$ *dgrad-sig-set d*› **show** $m' \in$ *dgrad-sig-set d* **unfolding** m'*-def*
by (*rule dgrad-sig-set-closed-minus*)
next
show *is-sig-red* $(\prec_t)$ $(=)$ $\{m'\}$ *p*
proof (*rule is-sig-red-top-addsI*)
show $m' \in \{m'\}$ **by** *simp*
next
from ‹*rep-list* $m \neq 0$› **show** *rep-list* $m' \neq 0$ **by** (*simp add*: *eq*)
next
show *punit.lt* (*rep-list* m') *adds punit.lt* (*rep-list p*) **by** (*simp add*: *eq eq1*)
next
have *punit.lt* (*rep-list p*) $\oplus$ *lt* $m' \prec_t$ *punit.lt* (*rep-list p*) $\oplus$ *u*

by (*rule splus-mono-strict*, *simp only*: *m′-def* ‹*lt* (*m* − *?s*) $\prec_t$ *u*›)
also have ... $\prec_t$ *punit.lt* (*rep-list m′*) ⊕ *lt p*
unfolding *eq eq1* **using** ‹*u* $\prec_t$ *lt p*› **by** (*rule splus-mono-strict*)
finally show *punit.lt* (*rep-list p*) ⊕ *lt m′* $\prec_t$ *punit.lt* (*rep-list m′*) ⊕ *lt p* **.**
next
show *ord-term-lin.is-le-rel* ($\prec_t$) **by** *simp*
qed *fact*
qed (*fact refl*)
finally show *?thesis* **by** (*simp only*: *m′-def*)
qed
ultimately show *False* **..**
qed

have *is-RB-in d rword G u* **by** (*rule is-RB-uptD2*, *fact*+)
thus *?thesis*
proof (*rule is-RB-inE*)
assume *is-syz-sig d u*
with ‹¬ *is-syz-sig d u*› **show** *?thesis* **..**
next
fix *g*
assume *is-canon-rewriter rword G u g*
hence *g* ∈ *G* **and** *g* ≠ *0* **and** *adds′*: *lt g* $adds_t$ *u* **by** (*rule is-canon-rewriterD*)+
assume *irred*: ¬ *is-sig-red* ($\prec_t$) (=) *G* (*monom-mult 1* (*pp-of-term u* − *lp g*) *g*)

define *b* **where** *b* = *monom-mult 1* (*pp-of-term u* − *lp g*) *g*
note *assms*(*1*)
moreover have *is-sig-GB-upt d G* (*lt m*) **unfolding** *u′*[*symmetric*]
by (*rule is-sig-GB-upt-le*, *rule is-RB-upt-is-sig-GB-upt*, *fact*+, *rule ord-term-lin.less-imp-le*, *fact*)
moreover from *assms*(*1*) **have** *b* ∈ *dgrad-sig-set d* **unfolding** *b-def*
proof (*rule dgrad-sig-set-closed-monom-mult*)
from *adds′* **have** *lp g adds pp-of-term u* **by** (*simp add*: *adds-term-def*)
with *assms*(*1*) **have** *d* (*pp-of-term u* − *lp g*) ≤ *d* (*pp-of-term u*) **by** (*rule dickson-grading-minus*)
thus *d* (*pp-of-term u* − *lp g*) ≤ *dgrad-max d* **using** ‹*d* (*pp-of-term u*) ≤ *dgrad-max d*›
by (*rule le-trans*)
next
from ‹*g* ∈ *G*› *G-sub* **show** *g* ∈ *dgrad-sig-set d* **..**
qed
moreover note ‹*m* ∈ *dgrad-sig-set d*›
moreover from ‹*g* ≠ *0*› **have** *lt b* = *lt m*
by (*simp add*: *b-def u′*[*symmetric*] *lt-monom-mult*,
metis adds′ add-diff-cancel-right′ adds-termE pp-of-term-splus)
moreover from ‹*g* ≠ *0*› **have** *b* ≠ *0* **by** (*simp add*: *b-def monom-mult-eq-zero-iff*)
moreover note ‹*m* ≠ *0*›
moreover from *irred* **have** ¬ *is-sig-red* ($\prec_t$) (=) *G b* **by** (*simp add*: *b-def*)
moreover have ¬ *is-sig-red* ($\prec_t$) (=) *G m*

proof
assume *is-sig-red* $(\prec_t)$ $(=)$ G m
then obtain *g2* **where** *1*: $g2 \in G$ **and** *2*: *rep-list g2* $\neq$ *0*
and *3*: *punit.lt* (*rep-list g2*) *adds punit.lt* (*rep-list m*)
and *4*: *punit.lt* (*rep-list m*) $\oplus$ *lt g2* $\prec_t$ *punit.lt* (*rep-list g2*) $\oplus$ *lt m*
by (*rule is-sig-red-top-addsE*)
from *2* **have** *g2* $\neq$ *0* **and** *punit.lc* (*rep-list g2*) $\neq$ *0* **by** (*auto simp*: *rep-list-zero punit.lc-eq-zero-iff*)
with *3 4* **have** *lt* (*monom-mult* (*?lc* / *punit.lc* (*rep-list g2*)) (*?lp* $-$ *punit.lt* (*rep-list g2*)) *g2*) $\prec_t$ *u*
(**is** *lt ?g2* $\prec_t$ *u*)
using ‹*?lc* $\neq$ *0*› **by** (*simp add*: *term-is-le-rel-minus u′ eq1 lt-monom-mult*)
hence *lt ?g2* $\notin$ *?M* **by** (*rule min*)
hence *?g2* $\notin$ *M* **by** *blast*
hence *g2* $\notin$ *dgrad-sig-set d* $\vee$ $\neg$ *is-sig-red* $(\prec_t)$ $(=)$ {*g2*} *p* **by** (*simp add*: *M-def*)
thus *False*
proof
assume *g2* $\notin$ *dgrad-sig-set d*
moreover from ‹*g2* $\in$ *G*› *G-sub* **have** *g2* $\in$ *dgrad-sig-set d* **..**
ultimately show *?thesis* **..**
next
assume $\neg$ *is-sig-red* $(\prec_t)$ $(=)$ {*g2*} *p*
moreover have *is-sig-red* $(\prec_t)$ $(=)$ {*g2*} *p*
proof (*rule is-sig-red-top-addsI*)
show *g2* $\in$ {*g2*} **by** *simp*
next
from *3* **show** *punit.lt* (*rep-list g2*) *adds punit.lt* (*rep-list p*) **by** (*simp only*: *eq1*)
next
from *4* **have** *?lp* $\oplus$ *lt g2* $\prec_t$ *punit.lt* (*rep-list g2*) $\oplus$ *u* **by** (*simp only*: *eq1 u′*)
also from ‹*u* $\prec_t$ *lt p*› **have** ... $\prec_t$ *punit.lt* (*rep-list g2*) $\oplus$ *lt p* **by** (*rule splus-mono-strict*)
finally show *?lp* $\oplus$ *lt g2* $\prec_t$ *punit.lt* (*rep-list g2*) $\oplus$ *lt p* **.**
next
show *ord-term-lin.is-le-rel* $(\prec_t)$ **by** *simp*
qed *fact+*
ultimately show *?thesis* **..**
qed
qed
ultimately have *eq2*: *punit.lt* (*rep-list b*) = *punit.lt* (*rep-list m*)
by (*rule sig-regular-top-reduced-lt-unique*)
have *rep-list g* $\neq$ *0* **by** (*rule is-RB-inD*, *fact+*)
moreover from *adds′* **have** *lp g adds pp-of-term u* **and** *component-of-term* (*lt g*) = *component-of-term u*
by (*simp-all add*: *adds-term-def*)
ultimately have *u* = (*?lp* $-$ *punit.lt* (*rep-list g*)) $\oplus$ *lt g*
by (*simp add*: *eq1*[*symmetric*] *eq2*[*symmetric*] *b-def rep-list-monom-mult*

punit.lt-monom-mult
splus-def adds-minus term-simps)
have *is-sig-red* $(\prec_t)$ $(=)$ $\{b\}$ *p*
proof (*rule is-sig-red-top-addsI*)
show $b \in \{b\}$ **by** *simp*
next
from ‹*rep-list g* $\neq$ *0*› **show** *rep-list b* $\neq$ *0*
by (*simp add*: *b-def rep-list-monom-mult punit.monom-mult-eq-zero-iff*)
next
show *punit.lt* (*rep-list b*) *adds punit.lt* (*rep-list p*) **by** (*simp add*: *eq1 eq2*)
next
show *punit.lt* (*rep-list p*) $\oplus$ *lt b* $\prec_t$ *punit.lt* (*rep-list b*) $\oplus$ *lt p*
by (*simp add*: *eq1 eq2* ‹*lt b* = *lt m*› *u*′[*symmetric*] ‹*u* $\prec_t$ *lt p*› *splus-mono-strict*)
next
show *ord-term-lin.is-le-rel* $(\prec_t)$ **by** *simp*
qed *fact*
hence *is-sig-red* $(\prec_t)$ $(=)$ $\{g\}$ *p* **unfolding** *b-def* **by** (*rule is-sig-red-singleton-monom-multD*)
show *?thesis* **by** (*rule*, *fact*+)
qed
qed

4.2.5 Termination

definition *term-pp-rel* :: $('t \Rightarrow 't \Rightarrow bool) \Rightarrow ('t \times 'a) \Rightarrow ('t \times 'a) \Rightarrow bool$
where *term-pp-rel r a b* $\longleftrightarrow$ *r* (*snd b* $\oplus$ *fst a*) (*snd a* $\oplus$ *fst b*)

definition *canon-term-pp-pair* :: $('t \times 'a) \Rightarrow bool$
where *canon-term-pp-pair a* $\longleftrightarrow$ (*gcs* (*pp-of-term* (*fst a*)) (*snd a*) = *0*)

definition *cancel-term-pp-pair* :: $('t \times 'a) \Rightarrow ('t \times 'a)$
where *cancel-term-pp-pair a* = (*fst a* $\ominus$ (*gcs* (*pp-of-term* (*fst a*)) (*snd a*)), *snd a* − (*gcs* (*pp-of-term* (*fst a*)) (*snd a*)))

lemma *term-pp-rel-refl*: *reflp r* $\Longrightarrow$ *term-pp-rel r a a*
by (*simp add*: *term-pp-rel-def reflp-def*)

lemma *term-pp-rel-irrefl*: *irreflp r* $\Longrightarrow$ $\neg$ *term-pp-rel r a a*
by (*simp add*: *term-pp-rel-def irreflp-def*)

lemma *term-pp-rel-sym*: *symp r* $\Longrightarrow$ *term-pp-rel r a b* $\Longrightarrow$ *term-pp-rel r b a*
by (*auto simp*: *term-pp-rel-def symp-def*)

lemma *term-pp-rel-trans*:
assumes *ord-term-lin.is-le-rel r* **and** *term-pp-rel r a b* **and** *term-pp-rel r b c*
shows *term-pp-rel r a c*
proof −
from *assms*(*1*) **have** *transp r* **by** (*rule ord-term-lin.is-le-relE*, *auto*)
from *assms*(*2*) **have** *1*: *r* (*snd b* $\oplus$ *fst a*) (*snd a* $\oplus$ *fst b*) **by** (*simp only*: *term-pp-rel-def*)

from *assms*(*3*) **have** *2*: *r* (*snd c* ⊕ *fst b*) (*snd b* ⊕ *fst c*) **by** (*simp only*: *term-pp-rel-def*)
have *snd b* ⊕ (*snd c* ⊕ *fst a*) = *snd c* ⊕ (*snd b* ⊕ *fst a*) **by** (*rule splus-left-commute*)
also from *assms*(*1*) *1* **have** *r* ... (*snd a* ⊕ (*snd c* ⊕ *fst b*))
by (*simp add*: *splus-left-commute*[*of snd a*] *term-is-le-rel-canc-left*)
also from *assms*(*1*) *2* **have** *r* ... (*snd b* ⊕ (*snd a* ⊕ *fst c*))
by (*simp add*: *splus-left-commute*[*of snd b*] *term-is-le-rel-canc-left*)
finally(*transpD*[*OF* ‹*transp r*›]) **show** *?thesis* **using** *assms*(*1*)
by (*simp only*: *term-pp-rel-def term-is-le-rel-canc-left*)
qed

lemma *term-pp-rel-trans-eq-left*:
assumes *ord-term-lin.is-le-rel r* **and** *term-pp-rel* (=) *a b* **and** *term-pp-rel r b c*
shows *term-pp-rel r a c*
proof −
from *assms*(*1*) **have** *transp r* **by** (*rule ord-term-lin.is-le-relE*, *auto*)
from *assms*(*2*) **have** *1*: *snd b* ⊕ *fst a* = *snd a* ⊕ *fst b* **by** (*simp only*: *term-pp-rel-def*)
from *assms*(*3*) **have** *2*: *r* (*snd c* ⊕ *fst b*) (*snd b* ⊕ *fst c*) **by** (*simp only*: *term-pp-rel-def*)
have *snd b* ⊕ (*snd c* ⊕ *fst a*) = *snd c* ⊕ (*snd b* ⊕ *fst a*) **by** (*rule splus-left-commute*)
also from *assms*(*1*) *1* **have** ... = (*snd a* ⊕ (*snd c* ⊕ *fst b*))
by (*simp add*: *splus-left-commute*[*of snd a*])
finally have *eq*: *snd b* ⊕ (*snd c* ⊕ *fst a*) = *snd a* ⊕ (*snd c* ⊕ *fst b*) **.**
from *assms*(*1*) *2* **have** *r* (*snd b* ⊕ (*snd c* ⊕ *fst a*)) (*snd b* ⊕ (*snd a* ⊕ *fst c*))
unfolding *eq* **by** (*simp add*: *splus-left-commute*[*of snd b*] *term-is-le-rel-canc-left*)
thus *?thesis* **using** *assms*(*1*) **by** (*simp only*: *term-pp-rel-def term-is-le-rel-canc-left*)
qed

lemma *term-pp-rel-trans-eq-right*:
assumes *ord-term-lin.is-le-rel r* **and** *term-pp-rel r a b* **and** *term-pp-rel* (=) *b c*
shows *term-pp-rel r a c*
proof −
from *assms*(*1*) **have** *transp r* **by** (*rule ord-term-lin.is-le-relE*, *auto*)
from *assms*(*2*) **have** *1*: *r* (*snd b* ⊕ *fst a*) (*snd a* ⊕ *fst b*) **by** (*simp only*: *term-pp-rel-def*)
from *assms*(*3*) **have** *2*: *snd c* ⊕ *fst b* = *snd b* ⊕ *fst c* **by** (*simp only*: *term-pp-rel-def*)
have *snd b* ⊕ (*snd a* ⊕ *fst c*) = *snd a* ⊕ (*snd b* ⊕ *fst c*) **by** (*rule splus-left-commute*)
also from *assms*(*1*) *2* **have** ... = (*snd a* ⊕ (*snd c* ⊕ *fst b*))
by (*simp add*: *splus-left-commute*[*of snd a*])
finally have *eq*: *snd b* ⊕ (*snd a* ⊕ *fst c*) = *snd a* ⊕ (*snd c* ⊕ *fst b*) **.**
from *assms*(*1*) *1* **have** *r* (*snd b* ⊕ (*snd c* ⊕ *fst a*)) (*snd b* ⊕ (*snd a* ⊕ *fst c*))
unfolding *eq* **by** (*simp add*: *splus-left-commute*[*of - snd c*] *term-is-le-rel-canc-left*)
thus *?thesis* **using** *assms*(*1*) **by** (*simp only*: *term-pp-rel-def term-is-le-rel-canc-left*)
qed

lemma *canon-term-pp-cancel*: *canon-term-pp-pair* (*cancel-term-pp-pair a*)
by (*simp add*: *cancel-term-pp-pair-def canon-term-pp-pair-def gcs-minus-gcs term-simps*)

lemma *term-pp-rel-cancel*:

assumes *reflp r*
shows *term-pp-rel r a* (*cancel-term-pp-pair a*)
proof −
obtain *u s* **where** *a*: $a = (u, s)$ **by** (*rule prod.exhaust*)
show *?thesis*
proof (*simp add*: *a cancel-term-pp-pair-def*)
let *?g* = *gcs* (*pp-of-term u*) *s*
have *?g adds s* **by** (*fact gcs-adds-2*)
hence $(s - ?g) \oplus (u \ominus 0) = s \oplus u \ominus (?g + 0)$ **using** *zero-adds-pp*
by (*rule minus-splus-sminus*)
also have ... $= s \oplus (u \ominus ?g)$
by (*metis add.left-neutral add.right-neutral adds-pp-def diff-zero gcs-adds-2 gcs-comm*
minus-splus-sminus zero-adds)
finally have $r\ ((s - ?g) \oplus u)\ (s \oplus (u \ominus ?g))$ **using** *assms* **by** (*simp add*: *term-simps reflp-def*)
thus *term-pp-rel r* (u, s) $(u \ominus ?g, s - ?g)$ **by** (*simp add*: *a term-pp-rel-def*)
qed
qed

lemma *canon-term-pp-rel-id*:
assumes *term-pp-rel* (=) *a b* **and** *canon-term-pp-pair a* **and** *canon-term-pp-pair b*
shows $a = b$
proof −
obtain *u s* **where** *a*: $a = (u, s)$ **by** (*rule prod.exhaust*)
obtain *v t* **where** *b*: $b = (v, t)$ **by** (*rule prod.exhaust*)
from *assms*(*1*) **have** $t \oplus u = s \oplus v$ **by** (*simp add*: *term-pp-rel-def a b*)
hence *1*: *t* + *pp-of-term u* = *s* + *pp-of-term v* **by** (*metis pp-of-term-splus*)
from *assms*(*2*) **have** *2*: *gcs* (*pp-of-term u*) *s* = *0* **by** (*simp add*: *canon-term-pp-pair-def a*)
from *assms*(*3*) **have** *3*: *gcs* (*pp-of-term v*) *t* = *0* **by** (*simp add*: *canon-term-pp-pair-def b*)
have *t* = *t* + *gcs* (*pp-of-term u*) *s* **by** (*simp add*: *2*)
also have ... = *gcs* (*t* + *pp-of-term u*) (*t* + *s*) **by** (*simp only*: *gcs-plus-left*)
also have ... = *gcs* (*s* + *pp-of-term v*) (*s* + *t*) **by** (*simp only*: *1 add.commute*)
also have ... = *s* + *gcs* (*pp-of-term v*) *t* **by** (*simp only*: *gcs-plus-left*)
also have ... = *s* **by** (*simp add*: *3*)
finally have $t = s$ **.**
moreover from ‹$t \oplus u = s \oplus v$› **have** $u = v$ **by** (*simp only*: ‹$t = s$› *splus-left-canc*)
ultimately show *?thesis* **by** (*simp add*: *a b*)
qed

lemma *min-set-finite*:
fixes *seq* :: *nat* ⇒ ('*t* $\Rightarrow_0$ '*b*::*field*)
assumes *dickson-grading d* **and** *range seq* ⊆ *dgrad-sig-set d* **and** *0* ∉ *rep-list* ‘ *range seq*
and $\bigwedge i\ j.\ i < j \Longrightarrow$ *lt* (*seq i*) $\prec_t$ *lt* (*seq j*)
shows *finite* {*i*. ¬ (∃*j*<*i*. *lt* (*seq j*) $adds_t$ *lt* (*seq i*) ∧

punit.lt (*rep-list* (*seq j*)) *adds punit.lt* (*rep-list* (*seq i*)))}

proof −
 have *inj* (λ*i*. *lt* (*seq i*))
 proof
  fix *i j*
  assume *eq*: *lt* (*seq i*) = *lt* (*seq j*)
  show *i* = *j*
  proof (*rule linorder-cases*)
   assume *i* < *j*
   hence *lt* (*seq i*) $\prec_t$ *lt* (*seq j*) **by** (*rule assms*(*4*))
   thus *?thesis* **by** (*simp add*: *eq*)
  next
   assume *j* < *i*
   hence *lt* (*seq j*) $\prec_t$ *lt* (*seq i*) **by** (*rule assms*(*4*))
   thus *?thesis* **by** (*simp add*: *eq*)
  qed
 qed
 hence *inj seq* **unfolding** *comp-def*[*symmetric*] **by** (*rule inj-on-imageI2*)

 let *?P1* = λ*p q*. *lt p adds*$_t$ *lt q*
 let *?P2* = λ*p q*. *punit.lt* (*rep-list p*) *adds punit.lt* (*rep-list q*)
 let *?P* = λ*p q*. *?P1 p q* ∧ *?P2 p q*
 have *reflp ?P* **by** (*simp add*: *reflp-def adds-term-refl*)
 have *almost-full-on ?P1* (*range seq*)
 proof (*rule almost-full-on-map*)
  let *?B* = {*t*. *pp-of-term t* ∈ *dgrad-set d* (*dgrad-max d*) ∧ *component-of-term t* ∈ {*0*..<*length fs*}}
  from *assms*(*1*) *finite-atLeastLessThan* **show** *almost-full-on* (*adds*$_t$) *?B* **by** (*rule Dickson-term*)
  show *lt* ‘ *range seq* ⊆ *?B*
  proof
   fix *v*
   assume *v* ∈ *lt* ‘ *range seq*
   then obtain *p* **where** *p* ∈ *range seq* **and** *v*: *v* = *lt p* **..**
   from *this*(*1*) *assms*(*3*) **have** *rep-list p* ≠ *0* **by** *auto*
   hence *p* ≠ *0* **by** (*auto simp*: *rep-list-zero*)
   from ‹*p* ∈ *range seq*› *assms*(*2*) **have** *p* ∈ *dgrad-sig-set d* **..**
   hence *d* (*lp p*) ≤ *dgrad-max d* **by** (*rule dgrad-sig-setD-lp*)
   hence *lp p* ∈ *dgrad-set d* (*dgrad-max d*) **by** (*simp add*: *dgrad-set-def*)
   moreover from ‹*p* ∈ *dgrad-sig-set d*› ‹*p* ≠ *0*› **have** *component-of-term* (*lt p*) < *length fs*
    by (*rule dgrad-sig-setD-lt*)
   ultimately show *v* ∈ *?B* **by** (*simp add*: *v*)
  qed
 qed
 moreover have *almost-full-on ?P2* (*range seq*)
 proof (*rule almost-full-on-map*)
  let *?B* = *dgrad-set d* (*dgrad-max d*)
  from *assms*(*1*) **show** *almost-full-on* (*adds*) *?B* **by** (*rule dickson-gradingD-dgrad-set*)

show $(\lambda p.\ punit.lt\ (rep\text{-}list\ p))$ ‘ $range\ seq \subseteq\ ?B$
proof
fix t
assume $t \in (\lambda p.\ punit.lt\ (rep\text{-}list\ p))$ ‘ $range\ seq$
then obtain p **where** $p \in range\ seq$ **and** t: $t = punit.lt\ (rep\text{-}list\ p)$ **..**
from $this(1)\ assms(3)$ **have** $rep\text{-}list\ p \neq 0$ **by** *auto*
from ‹$p \in range\ seq$› $assms(2)$ **have** $p \in dgrad\text{-}sig\text{-}set\ d$ **..**
hence $p \in dgrad\text{-}max\text{-}set\ d$ **by** (*simp add*: *dgrad-sig-set′-def*)
with $assms(1)$ **have** $rep\text{-}list\ p \in punit\text{-}dgrad\text{-}max\text{-}set\ d$ **by** (*rule dgrad-max-2*)
from *this* ‹$rep\text{-}list\ p \neq 0$› **have** $d\ (punit.lt\ (rep\text{-}list\ p)) \leq dgrad\text{-}max\ d$
by (*rule punit.dgrad-p-setD-lp*[*simplified*])
thus $t \in\ ?B$ **by** (*simp add*: *t dgrad-set-def*)
qed
qed
ultimately have *almost-full-on ?P* (*range seq*) **by** (*rule almost-full-on-same*)
with ‹*reflp ?P*› **obtain** T **where** *finite* T **and** $T \subseteq range\ seq$ **and** $*$: $\bigwedge p.\ p \in range\ seq \Longrightarrow (\exists q \in T.\ ?P\ q\ p)$
by (*rule almost-full-on-finite-subsetE*, *blast*)
from ‹$T \subseteq range\ seq$› **obtain** I **where** T: $T = seq$ ‘ I **by** (*meson subset-image-iff*)
have $\{i.\ \neg\ (\exists j<i.\ ?P\ (seq\ j)\ (seq\ i))\} \subseteq I$
proof
fix i
assume $i \in \{i.\ \neg\ (\exists j<i.\ ?P\ (seq\ j)\ (seq\ i))\}$
hence x: $\neg\ (\exists j<i.\ ?P\ (seq\ j)\ (seq\ i))$ **by** *simp*
obtain j **where** $j \in I$ **and** $?P\ (seq\ j)\ (seq\ i)$
proof −
have $seq\ i \in range\ seq$ **by** *simp*
hence $\exists q \in T.\ ?P\ q\ (seq\ i)$ **by** (*rule* $*$)
then obtain q **where** $q \in T$ **and** $?P\ q\ (seq\ i)$ **..**
from $this(1)$ **obtain** j **where** $j \in I$ **and** $q = seq\ j$ **unfolding** T **..**
from $this(1)$ ‹$?P\ q\ (seq\ i)$› **show** *?thesis* **unfolding** ‹$q = seq\ j$› **..**
qed
from $this(2)\ x$ **have** $i \leq j$ **by** *auto*
moreover have $\neg\ i < j$
proof
assume $i < j$
hence $lt\ (seq\ i) \prec_t lt\ (seq\ j)$ **by** (*rule* $assms(4)$)
hence $\neg\ ?P1\ (seq\ j)\ (seq\ i)$ **using** *ord-adds-term ord-term-lin.leD* **by** *blast*
with ‹$?P\ (seq\ j)\ (seq\ i)$› **show** *False* **by** *simp*
qed
ultimately show $i \in I$ **using** ‹$j \in I$› **by** *simp*
qed
moreover from ‹*inj seq*› ‹*finite* T› **have** *finite* I **by** (*simp add*: *finite-image-iff inj-on-subset* T)
ultimately show *?thesis* **by** (*rule finite-subset*)
qed

lemma *rb-termination*:

fixes *seq* :: *nat* ⇒ (*'t* $\Rightarrow_0$ *'b*::*field*)
assumes *dickson-grading d* **and** *range seq* ⊆ *dgrad-sig-set d* **and** *0* ∉ *rep-list* ' *range seq*
and ⋀*i j*. *i* < *j* ⟹ *lt* (*seq i*) $\prec_t$ *lt* (*seq j*)
and ⋀*i*. ¬ *is-sig-red* ($\prec_t$) (⪯) (*seq* ' {*0*..<*i*}) (*seq i*)
and ⋀*i*. (∃*j*<*length fs*. *lt* (*seq i*) = *lt* (*monomial* (*1*::*'b*) (*term-of-pair* (*0*, *j*))) ∧
punit.lt (*rep-list* (*seq i*)) ⪯ *punit.lt* (*rep-list* (*monomial 1* (*term-of-pair* (*0*, *j*))))) ∨
(∃*j k*. *is-regular-spair* (*seq j*) (*seq k*) ∧ *rep-list* (*spair* (*seq j*) (*seq k*)) ≠ *0* ∧
lt (*seq i*) = *lt* (*spair* (*seq j*) (*seq k*)) ∧
punit.lt (*rep-list* (*seq i*)) ⪯ *punit.lt* (*rep-list* (*spair* (*seq j*) (*seq k*))))
and ⋀*i*. *is-sig-GB-upt d* (*seq* ' {*0*..<*i*}) (*lt* (*seq i*))
shows *thesis*
proof −
from *assms*(*3*) **have** *0* ∉ *range seq* **using** *rep-list-zero* **by** *auto*
have *ord-term-lin.is-le-rel* (=) **and** *ord-term-lin.is-le-rel* ($\prec_t$) **by** (*rule ord-term-lin.is-le-relI*)+
have *reflp* (=) **and** *symp* (=) **by** (*simp-all add*: *symp-def*)
have *irreflp* ($\prec_t$) **by** (*simp add*: *irreflp-def*)
have *inj* (λ*i*. *lt* (*seq i*))
proof
fix *i j*
assume *eq*: *lt* (*seq i*) = *lt* (*seq j*)
show *i* = *j*
proof (*rule linorder-cases*)
assume *i* < *j*
hence *lt* (*seq i*) $\prec_t$ *lt* (*seq j*) **by** (*rule assms*(*4*))
thus *?thesis* **by** (*simp add*: *eq*)
next
assume *j* < *i*
hence *lt* (*seq j*) $\prec_t$ *lt* (*seq i*) **by** (*rule assms*(*4*))
thus *?thesis* **by** (*simp add*: *eq*)
qed
qed
hence *inj seq* **unfolding** *comp-def*[*symmetric*] **by** (*rule inj-on-imageI2*)

define *R* **where** *R* = (λ*x*. {*i*. *term-pp-rel* (=) (*lt* (*seq i*), *punit.lt* (*rep-list* (*seq i*))) *x*})
let *?A* = {*x*. *canon-term-pp-pair x* ∧ *R x* ≠ {}}

have *finite ?A*
proof −
define *min-set* **where** *min-set* = {*i*. ¬ (∃*j*<*i*. *lt* (*seq j*) $adds_t$ *lt* (*seq i*) ∧
punit.lt (*rep-list* (*seq j*)) *adds punit.lt* (*rep-list* (*seq i*)))}
have *?A* ⊆ (λ*i*. *cancel-term-pp-pair* (*lt* (*seq i*), *punit.lt* (*rep-list* (*seq i*)))) ' *min-set*
proof

fix *u t*
assume $(u, t) \in$ *?A*
hence *canon-term-pp-pair* (u, t) **and** $R\ (u, t) \neq \{\}$ **by** *simp-all*
from *this*(*2*) **obtain** *i* **where** *x*: *term-pp-rel* (=) (*lt* (*seq i*), *punit.lt* (*rep-list* (*seq i*))) (u, t)
by (*auto simp*: *R-def*)
let *?equiv* = (λ*i j*. *term-pp-rel* (=) (*lt* (*seq i*), *punit.lt* (*rep-list* (*seq i*))) (*lt* (*seq j*), *punit.lt* (*rep-list* (*seq j*))))
obtain *j* **where** $j \in$ *min-set* **and** *?equiv j i*
proof (*cases* $i \in$ *min-set*)
case *True*
moreover have *?equiv i i* **by** (*simp add*: *term-pp-rel-refl*)
ultimately show *?thesis* **..**
next
case *False*
let *?Q* = {*seq j* | *j*. $j < i \wedge$ *is-sig-red* (=) (=) {*seq j*} (*seq i*)}
have *?Q* $\subseteq$ *range seq* **by** *blast*
also have ... $\subseteq$ *dgrad-sig-set d* **by** (*fact assms*(*2*))
finally have *?Q* $\subseteq$ *dgrad-max-set d* **by** (*simp add*: *dgrad-sig-set'-def*)
moreover from ‹*?Q* $\subseteq$ *range seq*› ‹*0* $\notin$ *range seq*› **have** *0* $\notin$ *?Q* **by** *blast*
ultimately have *Q-sub*: *pp-of-term* ‘ *lt* ‘ *?Q* $\subseteq$ *dgrad-set d* (*dgrad-max d*)
unfolding *image-image* **by** (*smt CollectI dgrad-p-setD-lp dgrad-set-def image-subset-iff subsetCE*)
have $*$: $\exists$ *g*$\in$*seq* ‘ {*0..*<*k*}. *is-sig-red* (=) (=) {*g*} (*seq k*) **if** $k \notin$ *min-set* **for** *k*
proof −
from *that* **obtain** *j* **where** $j < k$ **and** *a*: *lt* (*seq j*) $adds_t$ *lt* (*seq k*)
and *b*: *punit.lt* (*rep-list* (*seq j*)) *adds punit.lt* (*rep-list* (*seq k*)) **by** (*auto simp*: *min-set-def*)
note *assms*(*1*, *7*)
moreover from *assms*(*2*) **have** *seq k* $\in$ *dgrad-sig-set d* **by** *fastforce*
moreover from ‹$j < k$› **have** *seq j* $\in$ *seq* ‘ {*0..*<*k*} **by** *simp*
moreover from *assms*(*3*) **have** *rep-list* (*seq k*) $\neq$ *0* **and** *rep-list* (*seq j*) $\neq$ *0* **by** *fastforce+*
ultimately have *is-sig-red* ($\preceq_t$) (=) (*seq* ‘ {*0..*<*k*}) (*seq k*) **using** *a b* **by** (*rule lemma-21*)
moreover from *assms*(*5*)[*of k*] **have** $\neg$ *is-sig-red* ($\prec_t$) (=) (*seq* ‘ {*0..*<*k*}) (*seq k*)
by (*simp add*: *is-sig-red-top-tail-cases*)
ultimately have *is-sig-red* (=) (=) (*seq* ‘ {*0..*<*k*}) (*seq k*)
by (*simp add*: *is-sig-red-sing-reg-cases*)
then obtain *g0* **where** *g0* $\in$ *seq* ‘ {*0..*<*k*} **and** *is-sig-red* (=) (=) {*g0*} (*seq k*)
by (*rule is-sig-red-singletonI*)
thus *?thesis* **..**
qed

from *this*[*OF False*] **obtain** *g0* **where** *g0* $\in$ *seq* ‘ {*0..*<*i*} **and** *is-sig-red* (=) (=) {*g0*} (*seq i*) **..**

hence *g0* ∈ *?Q* **by** *fastforce*
hence *lt g0* ∈ *lt* ' *?Q* **by** (*rule imageI*)
with *assms*(*1*) **obtain** *v* **where** $v \in lt$ ' *?Q* **and** *min*: $\bigwedge v'.\ v' \prec_t v \Longrightarrow v' \notin lt$ ' *?Q*
using *Q-sub* **by** (*rule ord-term-minimum-dgrad-set*, *blast*)
from *this*(*1*) **obtain** *j* **where** $j < i$ **and** *is-sig-red* (=) (=) {*seq j*} (*seq i*)
and *v*: $v = lt$ (*seq j*) **by** *fastforce*
hence *1*: *punit.lt* (*rep-list* (*seq j*)) *adds punit.lt* (*rep-list* (*seq i*))
and *2*: *punit.lt* (*rep-list* (*seq i*)) ⊕ *lt* (*seq j*) = *punit.lt* (*rep-list* (*seq j*)) ⊕ *lt* (*seq i*)
by (*auto elim*: *is-sig-red-top-addsE*)
show *?thesis*
proof
show *?equiv j i* **by** (*simp add*: *term-pp-rel-def 2*)
next
show *j* ∈ *min-set*
proof (*rule ccontr*)
assume *j* ∉ *min-set*
from ∗[*OF this*] **obtain** *g1* **where** *g1* ∈ *seq* ' {*0..<j*} **and** *red*: *is-sig-red* (=) (=) {*g1*} (*seq j*) **..**
from *this*(*1*) **obtain** *j0* **where** $j0 < j$ **and** $g1 = seq\ j0$ **by** *fastforce*+

from *red* **have** *3*: *punit.lt* (*rep-list* (*seq j0*)) *adds punit.lt* (*rep-list* (*seq j*))
and *4*: *punit.lt* (*rep-list* (*seq j*)) ⊕ *lt* (*seq j0*) = *punit.lt* (*rep-list* (*seq j0*)) ⊕ *lt* (*seq j*)
by (*auto simp*: ‹*g1* = *seq j0*› *elim*: *is-sig-red-top-addsE*)

from ‹*j0* < *j*› ‹*j* < *i*› **have** $j0 < i$ **by** *simp*
from ‹*j0* < *j*› **have** *lt* (*seq j0*) $\prec_t$ *v* **unfolding** *v* **by** (*rule assms*(*4*))
hence *lt* (*seq j0*) ∉ *lt* '*?Q* **by** (*rule min*)
with ‹*j0* < *i*› **have** ¬ *is-sig-red* (=) (=) {*seq j0*} (*seq i*) **by** *blast*
moreover have *is-sig-red* (=) (=) {*seq j0*} (*seq i*)
proof (*rule is-sig-red-top-addsI*)
from *assms*(*3*) **show** *rep-list* (*seq j0*) ≠ *0* **by** *fastforce*
next
from *assms*(*3*) **show** *rep-list* (*seq i*) ≠ *0* **by** *fastforce*
next
from *3 1* **show** *punit.lt* (*rep-list* (*seq j0*)) *adds punit.lt* (*rep-list* (*seq i*))
by (*rule adds-trans*)
next
from *4* **have** *?equiv j0 j* **by** (*simp add*: *term-pp-rel-def*)
also from *2* **have** *?equiv j i* **by** (*simp add*: *term-pp-rel-def*)
finally(*term-pp-rel-trans*[*OF* ‹*ord-term-lin.is-le-rel* (=)›])
show *punit.lt* (*rep-list* (*seq i*)) ⊕ *lt* (*seq j0*) = *punit.lt* (*rep-list* (*seq j0*)) ⊕ *lt* (*seq i*)
by (*simp add*: *term-pp-rel-def*)
next
show *ord-term-lin.is-le-rel* (=) **by** *simp*

qed *simp-all*
ultimately show *False* **..**
qed
qed
qed
have *term-pp-rel* (=) (*cancel-term-pp-pair* (*lt* (*seq j*), *punit.lt* (*rep-list* (*seq j*)))) (*lt* (*seq j*), *punit.lt* (*rep-list* (*seq j*)))
by (*rule term-pp-rel-sym*, *fact* ‹*symp* (=)›, *rule term-pp-rel-cancel*, *fact* ‹*reflp* (=)›)
also note ‹*?equiv j i*›
also(*term-pp-rel-trans*[*OF* ‹*ord-term-lin.is-le-rel* (=)›]) **note** *x*
finally(*term-pp-rel-trans*[*OF* ‹*ord-term-lin.is-le-rel* (=)›])
have *term-pp-rel* (=) (*cancel-term-pp-pair* (*lt* (*seq j*), *punit.lt* (*rep-list* (*seq j*)))) (*u*, *t*) **.**
with ‹*symp* (=)› **have** *term-pp-rel* (=) (*u*, *t*) (*cancel-term-pp-pair* (*lt* (*seq j*), *punit.lt* (*rep-list* (*seq j*))))
by (*rule term-pp-rel-sym*)
hence (*u*, *t*) = *cancel-term-pp-pair* (*lt* (*seq j*), *punit.lt* (*rep-list* (*seq j*)))
using ‹*canon-term-pp-pair* (*u*, *t*)› *canon-term-pp-cancel* **by** (*rule canon-term-pp-rel-id*)
with ‹*j* ∈ *min-set*› **show** (*u*, *t*) ∈ (λ*i*. *cancel-term-pp-pair* (*lt* (*seq i*), *punit.lt* (*rep-list* (*seq i*)))) ‘ *min-set*
by *fastforce*
qed
moreover have *finite* ((λ*i*. *cancel-term-pp-pair* (*lt* (*seq i*), *punit.lt* (*rep-list* (*seq i*)))) ‘ *min-set*)
proof (*rule finite-imageI*)
show *finite min-set* **unfolding** *min-set-def* **using** *assms*(*1*−*4*) **by** (*rule min-set-finite*)
qed
ultimately show *?thesis* **by** (*rule finite-subset*)
qed

have *range seq* ⊆ *seq* ‘ (⋃ (*R* ‘ *?A*))
proof (*rule image-mono*, *rule*)
fix *i*
show *i* ∈ (⋃ (*R* ‘ *?A*))
proof
show *i* ∈ *R* (*cancel-term-pp-pair* (*lt* (*seq i*), *punit.lt* (*rep-list* (*seq i*))))
by (*simp add*: *R-def term-pp-rel-cancel*)
thus *cancel-term-pp-pair* (*lt* (*seq i*), *punit.lt* (*rep-list* (*seq i*))) ∈ *?A*
using *canon-term-pp-cancel* **by** *blast*
qed
qed
moreover from ‹*inj seq*› **have** *infinite* (*range seq*) **by** (*rule range-inj-infinite*)
ultimately have *infinite* (*seq* ‘ (⋃ (*R* ‘ *?A*))) **by** (*rule infinite-super*)
moreover have *finite* (*seq* ‘ (⋃ (*R* ‘ *?A*)))
proof (*rule finite-imageI*, *rule finite-UN-I*)
fix *x*
assume *x* ∈ *?A*

let *?rel* = *term-pp-rel* ($\prec_t$)
have *irreflp ?rel* **by** (*rule irreflpI*, *rule term-pp-rel-irrefl*, *fact*)
moreover have *transp ?rel* **by** (*rule transpI*, *drule term-pp-rel-trans*[*OF* ‹*ord-term-lin.is-le-rel* ($\prec_t$)›])
ultimately have *wfp-on ?rel ?A* **using** ‹*finite ?A*› **by** (*rule wfp-on-finite*)
thus *finite* (*R x*) **using** ‹$x \in$ *?A*›
proof (*induct rule*: *wfp-on-induct*)
case (*less x*)
from *less*(*1*) **have** *canon-term-pp-pair x* **by** *simp*
define R' **where** $R' = \bigcup$ (*R* ‘ ({*x*. *canon-term-pp-pair x* ∧ *R x* ≠ {}} ∩ {*z*. *term-pp-rel* ($\prec_t$) *z x*}))
define *red-set* **where** *red-set* = (λp::'*t* $\Rightarrow_0$ '*b*. {*k*. *lt* (*seq k*) = *lt p* ∧
punit.lt (*rep-list* (*seq k*)) $\preceq$ *punit.lt* (*rep-list p*)})
have *finite-red-set*: *finite* (*red-set p*) **for** *p*
proof (*cases red-set p* = {})
case *True*
thus *?thesis* **by** *simp*
next
case *False*
then obtain *k* **where** *lt-k*: *lt* (*seq k*) = *lt p* **by** (*auto simp*: *red-set-def*)
have *red-set p* ⊆ {*k*}
proof
fix k'
assume $k' \in$ *red-set p*
hence *lt* (*seq* k') = *lt p* **by** (*simp add*: *red-set-def*)
hence *lt* (*seq* k') = *lt* (*seq k*) **by** (*simp only*: *lt-k*)
with ‹*inj* (λi. *lt* (*seq i*))› **have** $k' = k$ **by** (*rule injD*)
thus $k' \in \{k\}$ **by** *simp*
qed
thus *?thesis* **using** *infinite-super* **by** *auto*
qed

have *R x* ⊆ ($\bigcup$*i*∈R'. $\bigcup$*j*∈R'. *red-set* (*spair* (*seq i*) (*seq j*))) ∪
($\bigcup$*j*∈{*0*..<*length fs*}. *red-set* (*monomial 1* (*term-of-pair* (*0*, *j*))))
(**is** - ⊆ *?B* ∪ *?C*)
proof
fix *i*
assume *i* ∈ *R x*
hence *i-x*: *term-pp-rel* (=) (*lt* (*seq i*), *punit.lt* (*rep-list* (*seq i*))) *x*
by (*simp add*: *R-def term-pp-rel-def*)
from *assms*(*6*)[*of i*] **show** *i* ∈ *?B* ∪ *?C*
proof (*elim disjE exE conjE*)
fix *j*
assume *j* < *length fs*
hence *j* ∈ {*0*..<*length fs*} **by** *simp*
assume *lt* (*seq i*) = *lt* (*monomial* (*1*::'*b*) (*term-of-pair* (*0*, *j*)))
and *punit.lt* (*rep-list* (*seq i*)) $\preceq$ *punit.lt* (*rep-list* (*monomial 1* (*term-of-pair* (*0*, *j*))))
hence *i* ∈ *red-set* (*monomial 1* (*term-of-pair* (*0*, *j*))) **by** (*simp add*:

red-set-def)
with ‹$j \in \{0..<length\ fs\}$› **have** $i \in ?C$ **..**
thus *?thesis* **..**
next
fix *j k*
let $?li = punit.lt\ (rep\text{-}list\ (seq\ i))$
let $?lj = punit.lt\ (rep\text{-}list\ (seq\ j))$
let $?lk = punit.lt\ (rep\text{-}list\ (seq\ k))$
assume *lt-i*: $lt\ (seq\ i) = lt\ (spair\ (seq\ j)\ (seq\ k))$
and *lt-i'*: $?li \preceq punit.lt\ (rep\text{-}list\ (spair\ (seq\ j)\ (seq\ k)))$
and *spair-0*: $rep\text{-}list\ (spair\ (seq\ j)\ (seq\ k)) \neq 0$
hence $i \in red\text{-}set\ (spair\ (seq\ j)\ (seq\ k))$ **by** (*simp add*: *red-set-def*)
from *assms*(*3*) **have** *i-0*: $rep\text{-}list\ (seq\ i) \neq 0$ **and** *j-0*: $rep\text{-}list\ (seq\ j) \neq 0$
and *k-0*: $rep\text{-}list\ (seq\ k) \neq 0$ **by** *fastforce+*

have *R'I*: $a \in R'$ **if** $term\text{-}pp\text{-}rel\ (\prec_t)\ (lt\ (seq\ a),\ punit.lt\ (rep\text{-}list\ (seq\ a)))$
x **for** a
proof −
let $?x = cancel\text{-}term\text{-}pp\text{-}pair\ (lt\ (seq\ a),\ punit.lt\ (rep\text{-}list\ (seq\ a)))$
show *?thesis* **unfolding** *R'-def*
proof (*rule UN-I*, *simp*, *intro conjI*)
show $a \in R\ ?x$ **by** (*simp add*: *R-def term-pp-rel-cancel*)
thus $R\ ?x \neq \{\}$ **by** *blast*
next
note ‹$ord\text{-}term\text{-}lin.is\text{-}le\text{-}rel\ (\prec_t)$›
moreover have $term\text{-}pp\text{-}rel\ (=)\ ?x\ (lt\ (seq\ a),\ punit.lt\ (rep\text{-}list\ (seq$
$a)))$
by (*rule term-pp-rel-sym*, *fact*, *rule term-pp-rel-cancel*, *fact*)
ultimately show $term\text{-}pp\text{-}rel\ (\prec_t)\ ?x\ x$ **using** *that* **by** (*rule*
term-pp-rel-trans-eq-left)
qed (*fact canon-term-pp-cancel*)
qed

assume *is-regular-spair* (*seq j*) (*seq k*)
hence $?lk \oplus lt\ (seq\ j) \neq ?lj \oplus lt\ (seq\ k)$ **by** (*rule is-regular-spairD3*)
hence $term\text{-}pp\text{-}rel\ (\prec_t)\ (lt\ (seq\ j),\ ?lj)\ x \wedge term\text{-}pp\text{-}rel\ (\prec_t)\ (lt\ (seq\ k),$
$?lk)\ x$
proof (*rule ord-term-lin.neqE*)
assume *c*: $?lk \oplus lt\ (seq\ j) \prec_t ?lj \oplus lt\ (seq\ k)$
hence *j-k*: $term\text{-}pp\text{-}rel\ (\prec_t)\ (lt\ (seq\ j),\ ?lj)\ (lt\ (seq\ k),\ ?lk)$
by (*simp add*: *term-pp-rel-def*)
note ‹$ord\text{-}term\text{-}lin.is\text{-}le\text{-}rel\ (\prec_t)$›
moreover have $term\text{-}pp\text{-}rel\ (\prec_t)\ (lt\ (seq\ k),\ ?lk)\ (lt\ (seq\ i),\ ?li)$
proof (*simp add*: *term-pp-rel-def*)
from *lt-i'* **have** $?li \oplus lt\ (seq\ k) \preceq_t$
$punit.lt\ (rep\text{-}list\ (spair\ (seq\ j)\ (seq\ k))) \oplus lt\ (seq\ k)$
by (*rule splus-mono-left*)
also have ... $\prec_t (?lk - gcs\ ?lk\ ?lj + ?lj) \oplus lt\ (seq\ k)$
by (*rule splus-mono-strict-left*, *rule lt-rep-list-spair*, *fact+*, *simp only*:

add.commute)
also have ... = $((?lk + ?lj) - gcs\ ?lj\ ?lk) \oplus lt\ (seq\ k)$
by (*simp add*: *minus-plus gcs-adds-2 gcs-comm*)
also have ... = $?lk \oplus ((?lj - gcs\ ?lj\ ?lk) \oplus lt\ (seq\ k))$
by (*simp add*: *minus-plus′ gcs-adds splus-assoc*[*symmetric*])
also have ... = $?lk \oplus lt\ (seq\ i)$
by (*simp add*: *lt-spair′*[*OF k-0 - c*] *add.commute spair-comm*[*of seq j*] *lt-i*)
finally show $?li \oplus lt\ (seq\ k) \prec_t ?lk \oplus lt\ (seq\ i)$.
qed
ultimately have *term-pp-rel* $(\prec_t)$ (*lt* (*seq k*), *?lk*) *x* using *i-x*
by (*rule term-pp-rel-trans-eq-right*)
moreover from ‹*ord-term-lin.is-le-rel* $(\prec_t)$› *j-k this*
have *term-pp-rel* $(\prec_t)$ (*lt* (*seq j*), *?lj*) *x* by (*rule term-pp-rel-trans*)
ultimately show *?thesis* by *simp*
next
assume *c*: $?lj \oplus lt\ (seq\ k) \prec_t ?lk \oplus lt\ (seq\ j)$
hence *j-k*: *term-pp-rel* $(\prec_t)$ (*lt* (*seq k*), *?lk*) (*lt* (*seq j*), *?lj*)
by (*simp add*: *term-pp-rel-def*)
note ‹*ord-term-lin.is-le-rel* $(\prec_t)$›
moreover have *term-pp-rel* $(\prec_t)$ (*lt* (*seq j*), *?lj*) (*lt* (*seq i*), *?li*)
proof (*simp add*: *term-pp-rel-def*)
from *lt-i′* have $?li \oplus lt\ (seq\ j) \preceq_t$
punit.lt (*rep-list* (*spair* (*seq j*) (*seq k*))) $\oplus$ *lt* (*seq j*)
by (*rule splus-mono-left*)
thm *lt-rep-list-spair*
also have ... $\prec_t (?lk - gcs\ ?lk\ ?lj + ?lj) \oplus lt\ (seq\ j)$
by (*rule splus-mono-strict-left*, *rule lt-rep-list-spair*, *fact*+, *simp only*: *add.commute*)
also have ... = $((?lk + ?lj) - gcs\ ?lk\ ?lj) \oplus lt\ (seq\ j)$
by (*simp add*: *minus-plus gcs-adds-2 gcs-comm*)
also have ... = $?lj \oplus ((?lk - gcs\ ?lk\ ?lj) \oplus lt\ (seq\ j))$
by (*simp add*: *minus-plus′ gcs-adds splus-assoc*[*symmetric*] *add.commute*)
also have ... = $?lj \oplus lt\ (seq\ i)$ by (*simp add*: *lt-spair′*[*OF j-0 - c*] *lt-i add.commute*)
finally show $?li \oplus lt\ (seq\ j) \prec_t ?lj \oplus lt\ (seq\ i)$.
qed
ultimately have *term-pp-rel* $(\prec_t)$ (*lt* (*seq j*), *?lj*) *x* using *i-x*
by (*rule term-pp-rel-trans-eq-right*)
moreover from ‹*ord-term-lin.is-le-rel* $(\prec_t)$› *j-k this*
have *term-pp-rel* $(\prec_t)$ (*lt* (*seq k*), *?lk*) *x* by (*rule term-pp-rel-trans*)
ultimately show *?thesis* by *simp*
qed
with ‹$i \in$ *red-set* (*spair* (*seq j*) (*seq k*))› have $i \in ?B$ using $R'I$ by *blast*
thus *?thesis* ..
qed
qed
moreover have *finite* $(?B \cup ?C)$
proof (*rule finite-UnI*)

have *finite* R' **unfolding** R'*-def*
proof (*rule finite-UN-I*)
from ‹*finite ?A*› **show** *finite* (*?A* ∩ {*z. term-pp-rel* ($\prec_t$) *z x*}) **by** *simp*
next
fix *y*
assume *y* ∈ *?A* ∩ {*z. term-pp-rel* ($\prec_t$) *z x*}
hence *y* ∈ *?A* **and** *term-pp-rel* ($\prec_t$) *y x* **by** *simp-all*
thus *finite* (*R y*) **by** (*rule less*(*2*))
qed
show *finite ?B* **by** (*intro finite-UN-I* ‹*finite* R'› *finite-red-set*)
next
show *finite ?C* **by** (*intro finite-UN-I finite-atLeastLessThan finite-red-set*)
qed
ultimately show *?case* **by** (*rule finite-subset*)
qed
qed *fact*
ultimately show *?thesis* **..**
qed

4.2.6 Concrete Rewrite Orders

definition *is-strict-rewrite-ord* :: (($'t$ × ($'a \Rightarrow_0$ $'b$)) ⇒ ($'t$ × ($'a \Rightarrow_0$ $'b$)) ⇒ *bool*) ⇒ *bool*
where *is-strict-rewrite-ord rel* ⟷ *is-rewrite-ord* (λ*x y*. ¬ *rel y x*)

lemma *is-strict-rewrite-ordI*: *is-rewrite-ord* (λ*x y*. ¬ *rel y x*) ⟹ *is-strict-rewrite-ord rel*
unfolding *is-strict-rewrite-ord-def* **by** *blast*

lemma *is-strict-rewrite-ordD*: *is-strict-rewrite-ord rel* ⟹ *is-rewrite-ord* (λ*x y*. ¬ *rel y x*)
unfolding *is-strict-rewrite-ord-def* **by** *blast*

lemma *is-strict-rewrite-ord-antisym*:
assumes *is-strict-rewrite-ord rel* **and** ¬ *rel x y* **and** ¬ *rel y x*
shows *fst x* = *fst y*
by (*rule is-rewrite-ordD4*, *rule is-strict-rewrite-ordD*, *fact*+)

lemma *is-strict-rewrite-ord-asym*:
assumes *is-strict-rewrite-ord rel* **and** *rel x y*
shows ¬ *rel y x*
proof −
from *assms*(*1*) **have** *is-rewrite-ord* (λ*x y*. ¬ *rel y x*) **by** (*rule is-strict-rewrite-ordD*)
thus *?thesis*
proof (*rule is-rewrite-ordD3*)
assume ¬ ¬ *rel y x*
assume ¬ *rel x y*
thus *?thesis* **using** ‹*rel x y*› **..**
qed

qed

lemma *is-strict-rewrite-ord-irrefl*: *is-strict-rewrite-ord rel* $\Longrightarrow \neg$ *rel x x*
  using *is-strict-rewrite-ord-asym* **by** *blast*

definition *rw-rat* :: $('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b)) \Rightarrow$ *bool*
  where *rw-rat p q* $\longleftrightarrow$ (*let u* = *punit.lt* (*snd q*) $\oplus$ *fst p*; *v* = *punit.lt* (*snd p*) $\oplus$ *fst q in*
    $u \prec_t v \vee (u = v \wedge$ *fst p* $\preceq_t$ *fst q*))

definition *rw-rat-strict* :: $('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b)) \Rightarrow$ *bool*
  where *rw-rat-strict p q* $\longleftrightarrow$ (*let u* = *punit.lt* (*snd q*) $\oplus$ *fst p*; *v* = *punit.lt* (*snd p*) $\oplus$ *fst q in*
    $u \prec_t v \vee (u = v \wedge$ *fst p* $\prec_t$ *fst q*))

definition *rw-add* :: $('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b)) \Rightarrow$ *bool*
  where *rw-add p q* $\longleftrightarrow$ (*fst p* $\preceq_t$ *fst q*)

definition *rw-add-strict* :: $('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b)) \Rightarrow$ *bool*
  where *rw-add-strict p q* $\longleftrightarrow$ (*fst p* $\prec_t$ *fst q*)

lemma *rw-rat-alt*: *rw-rat* = (λ*p q*. $\neg$ *rw-rat-strict q p*)
  by (*intro ext*, *auto simp*: *rw-rat-def rw-rat-strict-def Let-def*)

lemma *rw-rat-is-rewrite-ord*: *is-rewrite-ord rw-rat*
proof (*rule is-rewrite-ordI*)
  show *reflp rw-rat* **by** (*simp add*: *reflp-def rw-rat-def*)
next
  have *1*: *ord-term-lin.is-le-rel* $(\prec_t)$ **and** *2*: *ord-term-lin.is-le-rel* (=)
    by (*rule ord-term-lin.is-le-relI*)+
  have *rw-rat p q* $\longleftrightarrow$ (*term-pp-rel* $(\prec_t)$ (*fst p*, *punit.lt* (*snd p*)) (*fst q*, *punit.lt* (*snd q*)) $\vee$
    (*term-pp-rel* (=) (*fst p*, *punit.lt* (*snd p*)) (*fst q*, *punit.lt* (*snd q*)) $\wedge$
    fst p $\preceq_t$ *fst q*))
    for *p q*
  by (*simp add*: *rw-rat-def term-pp-rel-def Let-def*)
  thus *transp rw-rat*
   by (*auto simp*: *transp-def dest*: *term-pp-rel-trans*[*OF 1*] *term-pp-rel-trans-eq-left*[*OF 1*]
     term-pp-rel-trans-eq-right[*OF 1*] *term-pp-rel-trans*[*OF 2*])
next
  fix *p q*
  show *rw-rat p q* $\vee$ *rw-rat q p* **by** (*auto simp*: *rw-rat-def Let-def*)
next
  fix *p q*
  assume *rw-rat p q* **and** *rw-rat q p*
  thus *fst p* = *fst q* **by** (*auto simp*: *rw-rat-def Let-def*)
next

fix *d G p q*
assume *d*: *dickson-grading d* **and** *gb*: *is-sig-GB-upt d G* (*lt q*) **and** $p \in G$ **and** $q \in G$
and $p \neq 0$ **and** $q \neq 0$ **and** *lt p* $adds_t$ *lt q* **and** ¬ *is-sig-red* ($\prec_t$) (=) *G q*
let *?u* = *punit.lt* (*rep-list q*) ⊕ *lt p*
let *?v* = *punit.lt* (*rep-list p*) ⊕ *lt q*
from ‹*lt p* $adds_t$ *lt q*› **obtain** *t* **where** *lt-q*: *lt q* = *t* ⊕ *lt p* **by** (*rule adds-termE*)
from *gb* **have** *G* ⊆ *dgrad-sig-set d* **by** (*rule is-sig-GB-uptD1*)
hence *G* ⊆ *dgrad-max-set d* **by** (*simp add*: *dgrad-sig-set′-def*)
with *d* **obtain** *p′* **where** *red*: (*sig-red* ($\prec_t$) (=) *G*)** (*monom-mult 1 t p*) *p′*
and ¬ *is-sig-red* ($\prec_t$) (=) *G p′* **by** (*rule sig-irredE-dgrad-max-set*)
from *red* **have** *lt p′* = *lt* (*monom-mult 1 t p*) **and** *lc p′* = *lc* (*monom-mult 1 t p*)
and *2*: *punit.lt* (*rep-list p′*) ⪯ *punit.lt* (*rep-list* (*monom-mult 1 t p*))
by (*rule sig-red-regular-rtrancl-lt*, *rule sig-red-regular-rtrancl-lc*, *rule sig-red-rtrancl-lt-rep-list*)
with ‹$p \neq 0$› **have** *lt p′* = *lt q* **and** *lc p′* = *lc p* **by** (*simp-all add*: *lt-q lt-monom-mult*)
from *2 punit.lt-monom-mult-le*[*simplified*] **have** *3*: *punit.lt* (*rep-list p′*) ⪯ *t* + *punit.lt* (*rep-list p*)
unfolding *rep-list-monom-mult* **by** (*rule ordered-powerprod-lin.order-trans*)
have *punit.lt* (*rep-list p′*) = *punit.lt* (*rep-list q*)
proof (*rule sig-regular-top-reduced-lt-unique*)
show *p′* ∈ *dgrad-sig-set d*
proof (*rule dgrad-sig-set-closed-sig-red-rtrancl*)
note *d*
moreover have *d t* ≤ *dgrad-max d*
proof (*rule le-trans*)
have *t adds lp q* **by** (*simp add*: *lt-q term-simps*)
with *d* **show** *d t* ≤ *d* (*lp q*) **by** (*rule dickson-grading-adds-imp-le*)
next
from ‹*q* ∈ *G*› ‹*G* ⊆ *dgrad-max-set d*› **have** *q* ∈ *dgrad-max-set d* **..**
thus *d* (*lp q*) ≤ *dgrad-max d* **using** ‹$q \neq 0$› **by** (*rule dgrad-p-setD-lp*)
qed
moreover from ‹*p* ∈ *G*› ‹*G* ⊆ *dgrad-sig-set d*› **have** *p* ∈ *dgrad-sig-set d* **..**
ultimately show *monom-mult 1 t p* ∈ *dgrad-sig-set d* **by** (*rule dgrad-sig-set-closed-monom-mult*)
qed *fact*+
next
from ‹*q* ∈ *G*› ‹*G* ⊆ *dgrad-sig-set d*› **show** *q* ∈ *dgrad-sig-set d* **..**
next
from ‹$p \neq 0$› ‹*lc p′* = *lc p*› **show** $p' \neq 0$ **by** (*auto simp*: *lc-eq-zero-iff*)
qed *fact*+
with *3* **have** *punit.lt* (*rep-list q*) ⪯ *t* + *punit.lt* (*rep-list p*) **by** *simp*
hence *?u* $\preceq_t$ (*t* + *punit.lt* (*rep-list p*)) ⊕ *lt p* **by** (*rule splus-mono-left*)
also have ... = *?v* **by** (*simp add*: *lt-q splus-assoc splus-left-commute*)
finally have *?u* $\preceq_t$ *?v* **by** (*simp only*: *rel-def*)
moreover from ‹*lt p* $adds_t$ *lt q*› **have** *lt p* $\preceq_t$ *lt q* **by** (*rule ord-adds-term*)
ultimately show *rw-rat* (*spp-of p*) (*spp-of q*) **by** (*auto simp*: *rw-rat-def Let-def spp-of-def*)
qed

lemma *rw-rat-strict-is-strict-rewrite-ord*: *is-strict-rewrite-ord rw-rat-strict*

proof (*rule is-strict-rewrite-ordI*)
  show *is-rewrite-ord* ($\lambda x\ y.\ \neg$ *rw-rat-strict y x*)
    unfolding *rw-rat-alt*[*symmetric*] **by** (*fact rw-rat-is-rewrite-ord*)
qed

lemma *rw-add-alt*: *rw-add* = ($\lambda p\ q.\ \neg$ *rw-add-strict q p*)
  by (*intro ext*, *auto simp*: *rw-add-def rw-add-strict-def*)

lemma *rw-add-is-rewrite-ord*: *is-rewrite-ord rw-add*
proof (*rule is-rewrite-ordI*)
  show *reflp rw-add* **by** (*simp add*: *reflp-def rw-add-def*)
next
  show *transp rw-add* **by** (*auto simp*: *transp-def rw-add-def*)
next
  fix *p q*
  show *rw-add p q* $\vee$ *rw-add q p* **by** (*simp only*: *rw-add-def ord-term-lin.linear*)
next
  fix *p q*
  assume *rw-add p q* **and** *rw-add q p*
  thus *fst p* = *fst q* **unfolding** *rw-add-def*
    by *simp*
next
  fix *p q* :: $'t \Rightarrow_0 'b$
  assume *lt p* $adds_t$ *lt q*
  thus *rw-add* (*spp-of p*) (*spp-of q*) **unfolding** *rw-add-def spp-of-def fst-conv* **by** (*rule ord-adds-term*)
qed

lemma *rw-add-strict-is-strict-rewrite-ord*: *is-strict-rewrite-ord rw-add-strict*
proof (*rule is-strict-rewrite-ordI*)
  show *is-rewrite-ord* ($\lambda x\ y.\ \neg$ *rw-add-strict y x*)
    unfolding *rw-add-alt*[*symmetric*] **by** (*fact rw-add-is-rewrite-ord*)
qed

4.2.7 Preparations for Sig-Poly-Pairs

context
  fixes *dgrad* :: $'a \Rightarrow nat$
begin

definition *spp-rel* :: $('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow bool$
  where *spp-rel sp r* $\longleftrightarrow$ ($r \neq 0 \wedge r \in$ *dgrad-sig-set dgrad* $\wedge$ *lt r* = *fst sp* $\wedge$ *rep-list r* = *snd sp*)

definition *spp-inv* :: $('t \times ('a \Rightarrow_0 'b)) \Rightarrow bool$
  where *spp-inv sp* $\longleftrightarrow$ *Ex* (*spp-rel sp*)

definition *vec-of* :: $('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \Rightarrow_0 'b)$
  where *vec-of sp* = (*if spp-inv sp then Eps* (*spp-rel sp*) *else 0*)

lemma *spp-inv-spp-of*:
  assumes $r \neq 0$ **and** $r \in$ *dgrad-sig-set dgrad*
  shows *spp-inv* (*spp-of* *r*)
  unfolding *spp-inv-def spp-rel-def*
proof (*intro exI conjI*)
  show *lt* r = *fst* (*spp-of* *r*) **by** (*simp add*: *spp-of-def*)
next
  show *rep-list* r = *snd* (*spp-of* *r*) **by** (*simp add*: *spp-of-def*)
qed *fact+*

context
  fixes *sp* :: $'t \times ('a \Rightarrow_0 'b)$
  assumes *spi*: *spp-inv sp*
begin

lemma *sig-poly-rel-vec-of*: *spp-rel sp* (*vec-of sp*)
proof −
  from *spi* **have** *eq*: *vec-of sp* = *Eps* (*spp-rel sp*) **by** (*simp add*: *vec-of-def*)
  from *spi* **show** *?thesis* **unfolding** *eq spp-inv-def* **by** (*rule someI-ex*)
qed

lemma *vec-of-nonzero*: *vec-of sp* $\neq 0$
  using *sig-poly-rel-vec-of* **by** (*simp add*: *spp-rel-def*)

lemma *lt-vec-of*: *lt* (*vec-of sp*) = *fst sp*
  using *sig-poly-rel-vec-of* **by** (*simp add*: *spp-rel-def*)

lemma *rep-list-vec-of*: *rep-list* (*vec-of sp*) = *snd sp*
  using *sig-poly-rel-vec-of* **by** (*simp add*: *spp-rel-def*)

lemma *spp-of-vec-of*: *spp-of* (*vec-of sp*) = *sp*
  by (*simp add*: *spp-of-def lt-vec-of rep-list-vec-of*)

end

lemma *map-spp-of-vec-of*:
  assumes *list-all spp-inv sps*
  shows *map* (*spp-of* $\circ$ *vec-of*) *sps* = *sps*
proof (*rule map-idI*)
  fix *sp*
  assume *sp* $\in$ *set sps*
  with *assms* **have** *spp-inv sp* **by** (*simp add*: *list-all-def*)
  hence *spp-of* (*vec-of sp*) = *sp* **by** (*rule spp-of-vec-of*)
  thus (*spp-of* $\circ$ *vec-of*) *sp* = *sp* **by** *simp*
qed

lemma *vec-of-dgrad-sig-set*: *vec-of sp* $\in$ *dgrad-sig-set dgrad*
proof (*cases spp-inv sp*)

case *True*
hence *spp-rel sp* (*vec-of sp*) **by** (*rule sig-poly-rel-vec-of*)
thus *?thesis* **by** (*simp add*: *spp-rel-def*)
next
case *False*
moreover have *0* ∈ *dgrad-sig-set dgrad* **unfolding** *dgrad-sig-set′-def*
proof
show *0* ∈ *dgrad-max-set dgrad* **by** (*rule dgrad-p-setI*) *simp*
next
show *0* ∈ *sig-inv-set* **by** (*rule sig-inv-setI*) (*simp add*: *term-simps*)
qed
ultimately show *?thesis* **by** (*simp add*: *vec-of-def*)
qed

lemma *spp-invD-fst*:
assumes *spp-inv sp*
shows *dgrad* (*pp-of-term* (*fst sp*)) ≤ *dgrad-max dgrad* **and** *component-of-term* (*fst sp*) < *length fs*
proof −
from *vec-of-dgrad-sig-set* **have** *dgrad* (*lp* (*vec-of sp*)) ≤ *dgrad-max dgrad* **by** (*rule dgrad-sig-setD-lp*)
with *assms* **show** *dgrad* (*pp-of-term* (*fst sp*)) ≤ *dgrad-max dgrad* **by** (*simp add*: *lt-vec-of*)
from *vec-of-dgrad-sig-set vec-of-nonzero*[*OF assms*] **have** *component-of-term* (*lt* (*vec-of sp*)) < *length fs*
by (*rule dgrad-sig-setD-lt*)
with *assms* **show** *component-of-term* (*fst sp*) < *length fs* **by** (*simp add*: *lt-vec-of*)
qed

lemma *spp-invD-snd*:
assumes *dickson-grading dgrad* **and** *spp-inv sp*
shows *snd sp* ∈ *punit-dgrad-max-set dgrad*
proof −
from *vec-of-dgrad-sig-set*[*of sp*] **have** *vec-of sp* ∈ *dgrad-max-set dgrad* **by** (*simp add*: *dgrad-sig-set′-def*)
with *assms*(*1*) **have** *rep-list* (*vec-of sp*) ∈ *punit-dgrad-max-set dgrad* **by** (*rule dgrad-max-2*)
with *assms*(*2*) **show** *?thesis* **by** (*simp add*: *rep-list-vec-of*)
qed

lemma *vec-of-inj*:
assumes *spp-inv sp* **and** *vec-of sp* = *vec-of sp′*
shows *sp* = *sp′*
proof −
from *assms*(*1*) **have** *vec-of sp* ≠ *0* **by** (*rule vec-of-nonzero*)
hence *vec-of sp′* ≠ *0* **by** (*simp add*: *assms*(*2*))
hence *spp-inv sp′* **by** (*simp add*: *vec-of-def split*: *if-split-asm*)
from *assms*(*1*) **have** *sp* = *spp-of* (*vec-of sp*) **by** (*simp only*: *spp-of-vec-of*)
also have ... = *spp-of* (*vec-of sp′*) **by** (*simp only*: *assms*(*2*))

```
  also from ‹spp-inv sp′› have ... = sp′ by (rule spp-of-vec-of)
  finally show ?thesis .
qed

lemma spp-inv-alt: spp-inv sp ⟷ (vec-of sp ≠ 0)
proof −
  have spp-inv sp if vec-of sp ≠ 0
  proof (rule ccontr)
    assume ¬ spp-inv sp
    hence vec-of sp = 0 by (simp add: vec-of-def)
    with that show False ..
  qed
  thus ?thesis by (auto dest: vec-of-nonzero)
qed

lemma spp-of-vec-of-spp-of:
  assumes p ∈ dgrad-sig-set dgrad
  shows spp-of (vec-of (spp-of p)) = spp-of p
proof (cases p = 0)
  case True
  show ?thesis
  proof (cases spp-inv (spp-of p))
    case True
    thus ?thesis by (rule spp-of-vec-of)
  next
    case False
    hence vec-of (spp-of p) = 0 by (simp add: spp-inv-alt)
    thus ?thesis by (simp only: True)
  qed
next
  case False
  have spp-inv (spp-of p) unfolding spp-inv-def
  proof
   from False assms show spp-rel (spp-of p) p by (simp add: spp-rel-def spp-of-def)
  qed
  thus ?thesis by (rule spp-of-vec-of)
qed
```

4.2.8 Total Reduction

```
primrec find-sig-reducer :: ('t × ('a ⇒₀ 'b)) list ⇒ 't ⇒ 'a ⇒ nat ⇒ nat option
where
  find-sig-reducer [] - - - = None|
  find-sig-reducer (b # bs) u t i =
        (if snd b ≠ 0 ∧ punit.lt (snd b) adds t ∧ (t − punit.lt (snd b)) ⊕ fst b ≺t
u then Some i
          else find-sig-reducer bs u t (Suc i))

lemma find-sig-reducer-SomeD-aux:
```

assumes *find-sig-reducer bs u t i* = *Some j*
shows $i \leq j$ **and** $j - i <$ *length bs*
proof −
from *assms* **have** $i \leq j \wedge j - i <$ *length bs*
proof (*induct bs arbitrary*: *i*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons b bs*)
from *Cons*(*2*) **show** *?case*
proof (*simp split*: *if-split-asm*)
assume *find-sig-reducer bs u t* (*Suc i*) = *Some j*
hence *Suc* $i \leq j \wedge j -$ *Suc* $i <$ *length bs* **by** (*rule Cons*(*1*))
thus $i \leq j \wedge j - i <$ *Suc* (*length bs*) **by** *auto*
qed
qed
thus $i \leq j$ **and** $j - i <$ *length bs* **by** *simp-all*
qed

lemma *find-sig-reducer-SomeD′*:
assumes *find-sig-reducer bs u t i* = *Some j* **and** *b* = *bs* ! ($j - i$)
shows $b \in$ *set bs* **and** *snd b* $\neq$ *0* **and** *punit.lt* (*snd b*) *adds t* **and** (*t* − *punit.lt* (*snd b*)) $\oplus$ *fst b* $\prec_t$ *u*
proof −
from *assms*(*1*) **have** $j - i <$ *length bs* **by** (*rule find-sig-reducer-SomeD-aux*)
thus $b \in$ *set bs* **unfolding** *assms*(*2*) **by** (*rule nth-mem*)
next
from *assms* **have** *snd b* $\neq$ *0* $\wedge$ *punit.lt* (*snd b*) *adds t* $\wedge$ (*t* − *punit.lt* (*snd b*)) $\oplus$ *fst b* $\prec_t$ *u*
proof (*induct bs arbitrary*: *i*)
case *Nil*
from *Nil*(*1*) **show** *?case* **by** *simp*
next
case (*Cons a bs*)
from *Cons*(*2*) **show** *?case*
proof (*simp split*: *if-split-asm*)
assume $i = j$
with *Cons*(*3*) **have** $b = a$ **by** *simp*
moreover assume *snd a* $\neq$ *0* **and** *punit.lt* (*snd a*) *adds t* **and** (*t* − *punit.lt* (*snd a*)) $\oplus$ *fst a* $\prec_t$ *u*
ultimately show *?case* **by** *simp*
next
assume ∗: *find-sig-reducer bs u t* (*Suc i*) = *Some j*
hence *Suc* $i \leq j$ **by** (*rule find-sig-reducer-SomeD-aux*)
note *Cons*(*3*)
also from ‹*Suc* $i \leq j$› **have** (*a* # *bs*) ! ($j - i$) = *bs* ! ($j -$ *Suc i*) **by** *simp*
finally have *b* = *bs* ! ($j -$ *Suc i*) **.**
with ∗ **show** *?case* **by** (*rule Cons*(*1*))
qed

qed
thus *snd b* $\neq$ *0* **and** *punit.lt* (*snd b*) *adds t* **and** (*t* $-$ *punit.lt* (*snd b*)) $\oplus$ *fst b* $\prec_t$ *u* **by** *simp-all*
qed

corollary *find-sig-reducer-SomeD*:
assumes *find-sig-reducer* (*map spp-of bs*) *u t 0* = *Some i*
shows *i* < *length bs* **and** *rep-list* (*bs* ! *i*) $\neq$ *0* **and** *punit.lt* (*rep-list* (*bs* ! *i*)) *adds t*
and (*t* $-$ *punit.lt* (*rep-list* (*bs* ! *i*))) $\oplus$ *lt* (*bs* ! *i*) $\prec_t$ *u*
proof $-$
from *assms* **have** *i* $-$ *0* < *length* (*map spp-of bs*) **by** (*rule find-sig-reducer-SomeD-aux*)
thus *i* < *length bs* **by** *simp*
hence *spp-of* (*bs* ! *i*) = (*map spp-of bs*) ! (*i* $-$ *0*) **by** *simp*
with *assms* **have** *snd* (*spp-of* (*bs* ! *i*)) $\neq$ *0* **and** *punit.lt* (*snd* (*spp-of* (*bs* ! *i*))) *adds t*
and (*t* $-$ *punit.lt* (*snd* (*spp-of* (*bs* ! *i*)))) $\oplus$ *fst* (*spp-of* (*bs* ! *i*)) $\prec_t$ *u*
by (*rule find-sig-reducer-SomeD'*)+
thus *rep-list* (*bs* ! *i*) $\neq$ *0* **and** *punit.lt* (*rep-list* (*bs* ! *i*)) *adds t*
and (*t* $-$ *punit.lt* (*rep-list* (*bs* ! *i*))) $\oplus$ *lt* (*bs* ! *i*) $\prec_t$ *u* **by** (*simp-all add*: *fst-spp-of snd-spp-of*)
qed

lemma *find-sig-reducer-NoneE*:
assumes *find-sig-reducer bs u t i* = *None* **and** *b* $\in$ *set bs*
assumes *snd b* = *0* $\Longrightarrow$ *thesis* **and** *snd b* $\neq$ *0* $\Longrightarrow$ $\neg$ *punit.lt* (*snd b*) *adds t* $\Longrightarrow$ *thesis*
and *snd b* $\neq$ *0* $\Longrightarrow$ *punit.lt* (*snd b*) *adds t* $\Longrightarrow$ $\neg$ (*t* $-$ *punit.lt* (*snd b*)) $\oplus$ *fst b* $\prec_t$ *u* $\Longrightarrow$ *thesis*
shows *thesis*
using *assms*
proof (*induct bs arbitrary*: *thesis i*)
case *Nil*
from *Nil*(*2*) **show** *?case* **by** *simp*
next
case (*Cons a bs*)
from *Cons*(*2*) **have** *1*: *snd a* = *0* $\vee$ $\neg$ *punit.lt* (*snd a*) *adds t* $\vee$ $\neg$ (*t* $-$ *punit.lt* (*snd a*)) $\oplus$ *fst a* $\prec_t$ *u*
and *eq*: *find-sig-reducer bs u t* (*Suc i*) = *None* **by** (*simp-all split*: *if-splits*)
from *Cons*(*3*) **have** *b* = *a* $\vee$ *b* $\in$ *set bs* **by** *simp*
thus *?case*
proof
assume *b* = *a*
show *?thesis*
proof (*cases snd a* = *0*)
case *True*
show *?thesis* **by** (*rule Cons*(*4*), *simp add*: ‹*b* = *a*› *True*)
next
case *False*

with *1* **have** *2*: $\neg$ *punit.lt* (*snd a*) *adds t* $\vee$ $\neg$ (t $-$ *punit.lt* (*snd a*)) $\oplus$ *fst a* $\prec_t$ *u* **by** *simp*
show *?thesis*
proof (*cases punit.lt* (*snd a*) *adds t*)
case *True*
with *2* **have** *3*: $\neg$ (t $-$ *punit.lt* (*snd a*)) $\oplus$ *fst a* $\prec_t$ *u* **by** *simp*
show *?thesis* **by** (*rule Cons*(*6*), *simp-all add*: ‹$b = a$› ‹*snd a* $\neq$ *0*› *True 3*)
next
case *False*
show *?thesis* **by** (*rule Cons*(*5*), *simp-all add*: ‹$b = a$› ‹*snd a* $\neq$ *0*› *False*)
qed
qed
next
assume $b \in$ *set bs*
with *eq* **show** *?thesis*
proof (*rule Cons*(*1*))
assume *snd b* = *0*
thus *?thesis* **by** (*rule Cons*(*4*))
next
assume *snd b* $\neq$ *0* **and** $\neg$ *punit.lt* (*snd b*) *adds t*
thus *?thesis* **by** (*rule Cons*(*5*))
next
assume *snd b* $\neq$ *0* **and** *punit.lt* (*snd b*) *adds t* **and** $\neg$ (t $-$ *punit.lt* (*snd b*)) $\oplus$ *fst b* $\prec_t$ *u*
thus *?thesis* **by** (*rule Cons*(*6*))
qed
qed
qed

lemma *find-sig-reducer-SomeD-red-single*:
assumes $t \in$ *keys* (*rep-list p*) **and** *find-sig-reducer* (*map spp-of bs*) (*lt p*) *t 0* = *Some i*
shows *sig-red-single* ($\prec_t$) ($\preceq$) *p* (*p* $-$ *monom-mult* (*lookup* (*rep-list p*) *t* / *punit.lc* (*rep-list* (*bs* ! *i*)))
(t $-$ *punit.lt* (*rep-list* (*bs* ! *i*))) (*bs* ! *i*)) (*bs* ! *i*) (t $-$ *punit.lt* (*rep-list* (*bs* ! *i*)))
proof $-$
from *assms*(*2*) **have** *punit.lt* (*rep-list* (*bs* ! *i*)) *adds t* **and** *1*: *rep-list* (*bs* ! *i*) $\neq$ *0*
and *2*: (t $-$ *punit.lt* (*rep-list* (*bs* ! *i*))) $\oplus$ *lt* (*bs* ! *i*) $\prec_t$ *lt p*
by (*rule find-sig-reducer-SomeD*)+
from *this*(*1*) **have** *eq*: t $-$ *punit.lt* (*rep-list* (*bs* ! *i*)) + *punit.lt* (*rep-list* (*bs* ! *i*)) = *t*
by (*rule adds-minus*)
from *assms*(*1*) **have** *3*: $t \preceq$ *punit.lt* (*rep-list p*) **by** (*rule punit.lt-max-keys*)
show *?thesis* **by** (*rule sig-red-singleI*, *simp-all add*: *eq 1 2 3 assms*(*1*))
qed

corollary *find-sig-reducer-SomeD-red*:

assumes $t \in keys\ (rep\text{-}list\ p)$ **and** $find\text{-}sig\text{-}reducer\ (map\ spp\text{-}of\ bs)\ (lt\ p)\ t\ 0 =$ $Some\ i$
shows $sig\text{-}red\ (\prec_t)\ (\preceq)\ (set\ bs)\ p\ (p - monom\text{-}mult\ (lookup\ (rep\text{-}list\ p)\ t\ /$ $punit.lc\ (rep\text{-}list\ (bs\ !\ i)))$
$(t - punit.lt\ (rep\text{-}list\ (bs\ !\ i)))\ (bs\ !\ i))$
unfolding *sig-red-def*
proof (*intro bexI exI*, *rule find-sig-reducer-SomeD-red-single*)
from $assms(2)$ **have** $i - 0 < length\ (map\ spp\text{-}of\ bs)$ **by** (*rule find-sig-reducer-SomeD-aux*)
hence $i < length\ bs$ **by** *simp*
thus $bs\ !\ i \in set\ bs$ **by** (*rule nth-mem*)
qed *fact+*

context
fixes $bs :: ('t \Rightarrow_0 'b)\ list$
begin

definition $sig\text{-}trd\text{-}term :: ('a \Rightarrow nat) \Rightarrow (('a \times ('t \Rightarrow_0 'b)) \times ('a \times ('t \Rightarrow_0 'b)))$ set
where $sig\text{-}trd\text{-}term\ d = \{(x, y).\ punit.dgrad\text{-}p\text{-}set\text{-}le\ d\ \{rep\text{-}list\ (snd\ x)\}$
$(insert\ (rep\text{-}list\ (snd\ y))\ (rep\text{-}list\ `\ set\ bs)) \wedge$
$fst\ x \in keys\ (rep\text{-}list\ (snd\ x)) \wedge fst\ y \in keys\ (rep\text{-}list$ $(snd\ y)) \wedge$
$fst\ x \prec fst\ y\}$

lemma *sig-trd-term-wf*:
assumes *dickson-grading d*
shows *wf* (*sig-trd-term d*)
proof (*rule wfI-min*)
fix $x :: 'a \times ('t \Rightarrow_0 'b)$ **and** Q
assume $x \in Q$
show $\exists z \in Q.\ \forall y.\ (y, z) \in sig\text{-}trd\text{-}term\ d \longrightarrow y \notin Q$
proof (*cases* $fst\ x \in keys\ (rep\text{-}list\ (snd\ x))$)
case *True*
define X **where** $X = rep\text{-}list\ `\ set\ bs$
let $?A = insert\ (rep\text{-}list\ (snd\ x))\ X$
have *finite X* **unfolding** *X-def* **by** *simp*
hence *finite ?A* **by** (*simp only*: *finite-insert*)
then obtain m **where** A: $?A \subseteq punit.dgrad\text{-}p\text{-}set\ d\ m$ **by** (*rule punit.dgrad-p-set-exhaust*)
hence x: $rep\text{-}list\ (snd\ x) \in punit.dgrad\text{-}p\text{-}set\ d\ m$ **and** X: $X \subseteq punit.dgrad\text{-}p\text{-}set$ $d\ m$
by *simp-all*
let $?Q = \{q \in Q.\ rep\text{-}list\ (snd\ q) \in punit.dgrad\text{-}p\text{-}set\ d\ m \wedge fst\ q \in keys\ (rep\text{-}list$ $(snd\ q))\}$
from ‹$x \in Q$› *x True* **have** $x \in ?Q$ **by** *simp*
have $\forall Q\ x.\ x \in Q \wedge Q \subseteq \{q.\ d\ q \le m\} \longrightarrow (\exists z \in Q.\ \forall y.\ y \prec z \longrightarrow y \notin Q)$
by (*rule wfp-on-imp-minimal*, *rule wfp-on-ord-strict*, *fact assms*)
hence 1: $fst\ x \in fst\ `\ ?Q \Longrightarrow fst\ `\ ?Q \subseteq \{q.\ d\ q \le m\} \Longrightarrow (\exists z \in fst\ `\ ?Q.\ \forall y.$ $y \prec z \longrightarrow y \notin fst\ `\ ?Q)$
by *meson*

have *fst x* ∈ *fst* ‘ *?Q* **by** (*rule*, *fact refl*, *fact*)
moreover have *fst* ‘ *?Q* ⊆ {*q*. *d q* ≤ *m*}
proof −
{
fix *q*
assume *a*: *rep-list* (*snd q*) ∈ *punit.dgrad-p-set d m* **and** *b*: *fst q* ∈ *keys* (*rep-list* (*snd q*))
from *a* **have** *keys* (*rep-list* (*snd q*)) ⊆ *dgrad-set d m* **by** (*simp add*: *punit.dgrad-p-set-def*)
with *b* **have** *fst q* ∈ *dgrad-set d m* **..**
hence *d* (*fst q*) ≤ *m* **by** (*simp add*: *dgrad-set-def*)
}
thus *?thesis* **by** *auto*
qed
ultimately have ∃ *z*∈*fst* ‘ *?Q*. ∀ *y*. *y* ≺ *z* ⟶ *y* ∉ *fst* ‘ *?Q* **by** (*rule 1*)
then obtain *z0* **where** *z0* ∈ *fst* ‘ *?Q* **and** *2*: ⋀*y*. *y* ≺ *z0* ⟹ *y* ∉ *fst* ‘ *?Q* **by** *blast*
from *this*(*1*) **obtain** *z* **where** *z* ∈ *?Q* **and** *z0*: *z0* = *fst z* **..**
hence *z* ∈ *Q* **and** *z*: *rep-list* (*snd z*) ∈ *punit.dgrad-p-set d m* **by** *simp-all*
from *this*(*1*) **show** ∃ *z*∈*Q*. ∀ *y*. (*y*, *z*) ∈ *sig-trd-term d* ⟶ *y* ∉ *Q*
proof
show ∀ *y*. (*y*, *z*) ∈ *sig-trd-term d* ⟶ *y* ∉ *Q*
proof (*intro allI impI*)
fix *y*
assume (*y*, *z*) ∈ *sig-trd-term d*
hence *3*: *punit.dgrad-p-set-le d* {*rep-list* (*snd y*)} (*insert* (*rep-list* (*snd z*)) *X*)
and *4*: *fst y* ∈ *keys* (*rep-list* (*snd y*)) **and** *fst y* ≺ *z0*
by (*simp-all add*: *sig-trd-term-def X-def z0*)
from *this*(*3*) **have** *fst y* ∉ *fst* ‘ *?Q* **by** (*rule 2*)
hence *y* ∉ *Q* ∨ *rep-list* (*snd y*) ∉ *punit.dgrad-p-set d m* ∨ *fst y* ∉ *keys* (*rep-list* (*snd y*))
by *auto*
thus *y* ∉ *Q*
proof (*elim disjE*)
assume *5*: *rep-list* (*snd y*) ∉ *punit.dgrad-p-set d m*
from *z X* **have** *insert* (*rep-list* (*snd z*)) *X* ⊆ *punit.dgrad-p-set d m* **by** *simp*
with *3* **have** {*rep-list* (*snd y*)} ⊆ *punit.dgrad-p-set d m* **by** (*rule punit.dgrad-p-set-le-dgrad-p-set*)
hence *rep-list* (*snd y*) ∈ *punit.dgrad-p-set d m* **by** *simp*
with *5* **show** *?thesis* **..**
next
assume *fst y* ∉ *keys* (*rep-list* (*snd y*))
thus *?thesis* **using** *4* **..**
qed
qed
qed

next
  case *False*
  from ‹$x \in Q$› **show** *?thesis*
  proof
    show $\forall y.\ (y,\ x) \in$ *sig-trd-term d* $\longrightarrow y \notin Q$
    proof (*intro allI impI*)
      fix *y*
      assume $(y,\ x) \in$ *sig-trd-term d*
      hence *fst x* $\in$ *keys* (*rep-list* (*snd x*)) **by** (*simp add*: *sig-trd-term-def*)
      with *False* **show** $y \notin Q$ **..**
    qed
  qed
qed
qed

function (*domintros*) *sig-trd-aux* :: $('a \times ('t \Rightarrow_0 'b)) \Rightarrow ('t \Rightarrow_0 'b)$ **where**
  sig-trd-aux (*t*, *p*) =
    (*let p'* =
      (*case find-sig-reducer* (*map spp-of bs*) (*lt p*) *t 0 of*
          None $\Rightarrow$ *p*
        | *Some i* $\Rightarrow$ *p* − *monom-mult* (*lookup* (*rep-list p*) *t* / *punit.lc* (*rep-list* (*bs* ! *i*)))
                (*t* − *punit.lt* (*rep-list* (*bs* ! *i*))) (*bs* ! *i*));
      p'' = *punit.lower* (*rep-list p'*) *t in*
    if p'' = *0 then p' else sig-trd-aux* (*punit.lt p''*, *p'*))
  by *auto*

lemma *sig-trd-aux-domI*:
  assumes *fst args0* $\in$ *keys* (*rep-list* (*snd args0*))
  shows *sig-trd-aux-dom args0*
proof −
  from *ex-hgrad* **obtain** *d*::*'a* $\Rightarrow$ *nat* **where** *dickson-grading d* $\wedge$ *hom-grading d* **..**
  hence *dg*: *dickson-grading d* **..**
  hence *wf* (*sig-trd-term d*) **by** (*rule sig-trd-term-wf*)
  thus *?thesis* **using** *assms*
  proof (*induct args0*)
    case (*less args*)
    obtain *t p* **where** *args*: *args* = (*t*, *p*) **using** *prod.exhaust* **by** *blast*
    with *less*(*1*) **have** *1*: $\bigwedge s\ q.\ ((s,\ q),\ (t,\ p)) \in$ *sig-trd-term d* $\Longrightarrow s \in$ *keys* (*rep-list q*) $\Longrightarrow$ *sig-trd-aux-dom* (*s*, *q*)
      using *prod.exhaust* **by** *auto*
    from *less*(*2*) **have** $t \in$ *keys* (*rep-list p*) **by** (*simp add*: *args*)
    show *?case* **unfolding** *args*
    proof (*rule sig-trd-aux.domintros*)
      define *p'* **where** *p'* = (*case find-sig-reducer* (*map spp-of bs*) (*lt p*) *t 0 of*
                None $\Rightarrow$ *p*
              | *Some i* $\Rightarrow$ *p* −
                  monom-mult (*lookup* (*rep-list p*) *t* / *punit.lc* (*rep-list* (*bs* ! *i*)))

$(t - punit.lt\ (rep\text{-}list\ (bs\ !\ i)))\ (bs\ !\ i))$
define p'' **where** $p'' = punit.lower\ (rep\text{-}list\ p')\ t$
assume $p'' \neq 0$
from ‹$p'' \neq 0$› **have** $punit.lt\ p'' \in keys\ p''$ **by** (*rule punit.lt-in-keys*)
also have ... $\subseteq keys\ (rep\text{-}list\ p')$ **by** (*auto simp*: p''*-def punit.keys-lower*)
finally have $punit.lt\ p'' \in keys\ (rep\text{-}list\ p')$ **.**
with - **show** *sig-trd-aux-dom* $(punit.lt\ p'', p')$
proof (*rule 1*)
have *punit.dgrad-p-set-le d* {*rep-list p'*} (*insert* (*rep-list p*) (*rep-list* ‘ *set bs*))
proof (*cases find-sig-reducer* (*map spp-of bs*) (*lt p*) *t 0*)
case *None*
hence $p' = p$ **by** (*simp add*: p'*-def*)
hence {*rep-list p'*} $\subseteq$ *insert* (*rep-list p*) (*rep-list* ‘ *set bs*) **by** *simp*
thus *?thesis* **by** (*rule punit.dgrad-p-set-le-subset*)
next
case (*Some i*)
hence p': $p' = p - monom\text{-}mult\ (lookup\ (rep\text{-}list\ p)\ t\ /\ punit.lc\ (rep\text{-}list\ (bs\ !\ i)))$
$(t - punit.lt\ (rep\text{-}list\ (bs\ !\ i)))\ (bs\ !\ i)$ **by** (*simp add*: p'*-def*)
have *sig-red* $(\prec_t)$ $(\preceq)$ (*set bs*) *p p'* **unfolding** p' **using** ‹$t \in keys\ (rep\text{-}list\ p)$› *Some*
by (*rule find-sig-reducer-SomeD-red*)
hence *punit.red* (*rep-list* ‘ *set bs*) (*rep-list p*) (*rep-list p'*) **by** (*rule sig-red-red*)
with *dg* **show** *?thesis* **by** (*rule punit.dgrad-p-set-le-red*)
qed
moreover note ‹$punit.lt\ p'' \in keys\ (rep\text{-}list\ p')$› ‹$t \in keys\ (rep\text{-}list\ p)$›
moreover from ‹$p'' \neq 0$› **have** $punit.lt\ p'' \prec t$ **unfolding** p''*-def* **by** (*rule punit.lt-lower-less*)
ultimately show $((punit.lt\ p'', p'), t, p) \in sig\text{-}trd\text{-}term\ d$ **by** (*simp add*: *sig-trd-term-def*)
qed
qed
qed
qed

definition *sig-trd* :: $('t \Rightarrow_0 'b) \Rightarrow ('t \Rightarrow_0 'b)$
where *sig-trd p* = (*if rep-list p* = *0 then p else sig-trd-aux* (*punit.lt* (*rep-list p*), *p*))

lemma *sig-trd-aux-red-rtrancl*:
assumes *fst args0* $\in$ *keys* (*rep-list* (*snd args0*))
shows (*sig-red* $(\prec_t)$ $(\preceq)$ (*set bs*))** (*snd args0*) (*sig-trd-aux args0*)
proof –
from *assms* **have** *sig-trd-aux-dom args0* **by** (*rule sig-trd-aux-domI*)
thus *?thesis* **using** *assms*
proof (*induct args0 rule*: *sig-trd-aux.pinduct*)
case (*1 t p*)
define p' **where** p' = (*case find-sig-reducer* (*map spp-of bs*) (*lt p*) *t 0 of*

None ⇒ *p*
| *Some i* ⇒ *p* −
monom-mult (*lookup* (*rep-list p*) *t* / *punit.lc* (*rep-list* (*bs*
! *i*)))
(*t* − *punit.lt* (*rep-list* (*bs* ! *i*))) (*bs* ! *i*))
define *p''* **where** *p''* = *punit.lower* (*rep-list p'*) *t*
from *1*(*3*) **have** *t* ∈ *keys* (*rep-list p*) **by** *simp*
have ∗: (*sig-red* ($\prec_t$) (⪯) (*set bs*))** *p p'*
proof (*cases find-sig-reducer* (*map spp-of bs*) (*lt p*) *t 0*)
case *None*
hence *p'* = *p* **by** (*simp add*: *p'-def*)
thus *?thesis* **by** *simp*
next
case (*Some i*)
hence *p'*: *p'* = *p* − *monom-mult* (*lookup* (*rep-list p*) *t* / *punit.lc* (*rep-list* (*bs*
! *i*)))
(*t* − *punit.lt* (*rep-list* (*bs* ! *i*))) (*bs* ! *i*) **by** (*simp add*:
p'-def)
have *sig-red* ($\prec_t$) (⪯) (*set bs*) *p p'* **unfolding** *p'* **using** ‹*t* ∈ *keys* (*rep-list p*)›
Some
by (*rule find-sig-reducer-SomeD-red*)
thus *?thesis* **..**
qed
show *?case*
proof (*simp add*: *sig-trd-aux.psimps*[*OF 1*(*1*)] *Let-def p'-def*[*symmetric*] *p''-def*[*symmetric*]
∗, *intro impI*)
assume *p''* ≠ *0*
from ∗ **have** (*sig-red* ($\prec_t$) (⪯) (*set bs*))** *p* (*snd* (*punit.lt p''*, *p'*)) **by** (*simp*
only: *snd-conv*)
moreover have (*sig-red* ($\prec_t$) (⪯) (*set bs*))** (*snd* (*punit.lt p''*, *p'*)) (*sig-trd-aux*
(*punit.lt p''*, *p'*))
using *p'-def p''-def* ‹*p''* ≠ *0*›
proof (*rule 1*(*2*))
from ‹*p''* ≠ *0*› **have** *punit.lt p''* ∈ *keys p''* **by** (*rule punit.lt-in-keys*)
also have ... ⊆ *keys* (*rep-list p'*) **by** (*auto simp*: *p''-def punit.keys-lower*)
finally show *fst* (*punit.lt p''*, *p'*) ∈ *keys* (*rep-list* (*snd* (*punit.lt p''*, *p'*))) **by**
simp
qed
ultimately show (*sig-red* ($\prec_t$) (⪯) (*set bs*))** *p* (*sig-trd-aux* (*punit.lt p''*, *p'*))
by (*rule rtranclp-trans*)
qed
qed
qed

corollary *sig-trd-red-rtrancl*: (*sig-red* ($\prec_t$) (⪯) (*set bs*))** *p* (*sig-trd p*)
unfolding *sig-trd-def*
proof (*split if-split*, *intro conjI impI rtranclp.rtrancl-refl*)
let *?args* = (*punit.lt* (*rep-list p*), *p*)
assume *rep-list p* ≠ *0*

hence *punit.lt* (*rep-list p*) ∈ *keys* (*rep-list p*) **by** (*rule punit.lt-in-keys*)
hence *fst* (*punit.lt* (*rep-list p*), *p*) ∈ *keys* (*rep-list* (*snd* (*punit.lt* (*rep-list p*), *p*)))
by (*simp only*: *fst-conv snd-conv*)
hence (*sig-red* ($\prec_t$) ($\preceq$) (*set bs*))** (*snd ?args*) (*sig-trd-aux ?args*) **by** (*rule sig-trd-aux-red-rtrancl*)
thus (*sig-red* ($\prec_t$) ($\preceq$) (*set bs*))** *p* (*sig-trd-aux* (*punit.lt* (*rep-list p*), *p*)) **by** (*simp only*: *snd-conv*)
qed

lemma *sig-trd-aux-irred*:
assumes *fst args0* ∈ *keys* (*rep-list* (*snd args0*))
and $\bigwedge$*b s*. *b* ∈ *set bs* ⟹ *rep-list b* ≠ *0* ⟹ *fst args0* ≺ *s* + *punit.lt* (*rep-list b*) ⟹
s ⊕ *lt b* $\prec_t$ *lt* (*snd* (*args0*)) ⟹ *lookup* (*rep-list* (*snd args0*)) (*s* + *punit.lt* (*rep-list b*)) = *0*
shows ¬ *is-sig-red* ($\prec_t$) ($\preceq$) (*set bs*) (*sig-trd-aux args0*)
proof −
from *assms*(*1*) **have** *sig-trd-aux-dom args0* **by** (*rule sig-trd-aux-domI*)
thus *?thesis* **using** *assms*
proof (*induct args0 rule*: *sig-trd-aux.pinduct*)
case (*1 t p*)
define *p′* **where** *p′* = (*case find-sig-reducer* (*map spp-of bs*) (*lt p*) *t 0 of*
None ⇒ *p*
| *Some i* ⇒ *p* −
monom-mult (*lookup* (*rep-list p*) *t* / *punit.lc* (*rep-list* (*bs* ! *i*)))
(*t* − *punit.lt* (*rep-list* (*bs* ! *i*))) (*bs* ! *i*))
define *p′′* **where** *p′′* = *punit.lower* (*rep-list p′*) *t*
from *1*(*3*) **have** *t* ∈ *keys* (*rep-list p*) **by** *simp*
from *1*(*4*) **have** *a*: *b* ∈ *set bs* ⟹ *rep-list b* ≠ *0* ⟹ *t* ≺ *s* + *punit.lt* (*rep-list b*) ⟹
s ⊕ *lt b* $\prec_t$ *lt p* ⟹ *lookup* (*rep-list p*) (*s* + *punit.lt* (*rep-list b*)) = *0*
for *b s* **by** (*simp only*: *fst-conv snd-conv*)
have *lt p′* = *lt p* ∧ (∀ *s*. *t* ≺ *s* ⟶ *lookup* (*rep-list p′*) *s* = *lookup* (*rep-list p*) *s*)
proof (*cases find-sig-reducer* (*map spp-of bs*) (*lt p*) *t 0*)
case *None*
thus *?thesis* **by** (*simp add*: *p′-def*)
next
case (*Some i*)
hence *p′*: *p′* = *p* − *monom-mult* (*lookup* (*rep-list p*) *t* / *punit.lc* (*rep-list* (*bs* ! *i*)))
(*t* − *punit.lt* (*rep-list* (*bs* ! *i*))) (*bs* ! *i*) **by** (*simp add*: *p′-def*)
have *sig-red-single* ($\prec_t$) ($\preceq$) *p p′* (*bs* ! *i*) (*t* − *punit.lt* (*rep-list* (*bs* ! *i*)))
unfolding *p′* **using** ‹*t* ∈ *keys* (*rep-list p*)› *Some* **by** (*rule find-sig-reducer-SomeD-red-single*)
hence *r*: *punit.red-single* (*rep-list p*) (*rep-list p′*) (*rep-list* (*bs* ! *i*)) (*t* − *punit.lt* (*rep-list* (*bs* ! *i*)))
and *lt p′* = *lt p* **by** (*rule sig-red-single-red-single*, *rule sig-red-single-regular-lt*)

have $\forall s.\ t \prec s \longrightarrow$ *lookup* (*rep-list* p') s = *lookup* (*rep-list* p) s
proof (*intro allI impI*)
fix s
assume $t \prec s$
from *Some* **have** *punit.lt* (*rep-list* (*bs* ! *i*)) *adds* t **by** (*rule find-sig-reducer-SomeD*)
hence *eq0*: (t − *punit.lt* (*rep-list* (*bs* ! *i*))) + *punit.lt* (*rep-list* (*bs* ! *i*)) = t (**is** *?t* = t)
by (*rule adds-minus*)
from ‹$t \prec s$› **have** *lookup* (*rep-list* p') s = *lookup* (*punit.higher* (*rep-list* p') *?t*) s
by (*simp add*: *eq0 punit.lookup-higher-when*)
also from r **have** ... = *lookup* (*punit.higher* (*rep-list* p) *?t*) s
by (*simp add*: *punit.red-single-higher*[*simplified*])
also from ‹$t \prec s$› **have** ... = *lookup* (*rep-list* p) s **by** (*simp add*: *eq0 punit.lookup-higher-when*)
finally show *lookup* (*rep-list* p') s = *lookup* (*rep-list* p) s **.**
qed
with ‹*lt* p' = *lt* p› **show** *?thesis* **..**
qed
hence *lt-p'*: *lt* p' = *lt* p **and** b: $\bigwedge s.\ t \prec s \Longrightarrow$ *lookup* (*rep-list* p') s = *lookup* (*rep-list* p) s
by *blast+*
have c: *lookup* (*rep-list* p') (s + *punit.lt* (*rep-list* b)) = *0*
if $b \in$ *set bs* **and** *rep-list* $b \neq$ *0* **and** $t \preceq s$ + *punit.lt* (*rep-list* b) **and** $s \oplus$ *lt* $b \prec_t$ *lt* p' **for** b s
proof (*cases* $t \prec s$ + *punit.lt* (*rep-list* b))
case *True*
hence *lookup* (*rep-list* p') (s + *punit.lt* (*rep-list* b)) =
lookup (*rep-list* p) (s + *punit.lt* (*rep-list* b)) **by** (*rule* b)
also from *that*(*1*, *2*) *True that*(*4*) **have** ... = *0* **unfolding** *lt-p'* **by** (*rule* a)
finally show *?thesis* **.**
next
case *False*
with *that*(*3*) **have** t: t = s + *punit.lt* (*rep-list* b) **by** *simp*
show *?thesis*
proof (*cases find-sig-reducer* (*map spp-of bs*) (*lt* p) t *0*)
case *None*
from *that*(*1*) **have** *spp-of* $b \in$ *set* (*map spp-of bs*) **by** *fastforce*
with *None* **show** *?thesis*
proof (*rule find-sig-reducer-NoneE*)
assume *snd* (*spp-of* b) = *0*
with *that*(*2*) **show** *?thesis* **by** (*simp add*: *snd-spp-of*)
next
assume ¬ *punit.lt* (*snd* (*spp-of* b)) *adds* t
thus *?thesis* **by** (*simp add*: *snd-spp-of* t)
next
assume ¬ (t − *punit.lt* (*snd* (*spp-of* b))) $\oplus$ *fst* (*spp-of* b) $\prec_t$ *lt* p
with *that*(*4*) **show** *?thesis* **by** (*simp add*: *fst-spp-of snd-spp-of* t *lt-p'*)
qed

next
case (*Some i*)
hence p': $p' = p - $ *monom-mult* (*lookup* (*rep-list p*) *t* / *punit.lc* (*rep-list* (*bs* ! *i*)))
(*t* − *punit.lt* (*rep-list* (*bs* ! *i*))) (*bs* ! *i*) **by** (*simp add*: p'-*def*)
have *sig-red-single* ($\prec_t$) ($\preceq$) *p* p' (*bs* ! *i*) (*t* − *punit.lt* (*rep-list* (*bs* ! *i*)))
unfolding p' **using** ‹*t* ∈ *keys* (*rep-list p*)› *Some* **by** (*rule find-sig-reducer-SomeD-red-single*)
hence *r*: *punit.red-single* (*rep-list p*) (*rep-list* p') (*rep-list* (*bs* ! *i*)) (*t* − *punit.lt* (*rep-list* (*bs* ! *i*)))
by (*rule sig-red-single-red-single*)
from *Some* **have** *punit.lt* (*rep-list* (*bs* ! *i*)) *adds t* **by** (*rule find-sig-reducer-SomeD*)
hence *eq0*: (*t* − *punit.lt* (*rep-list* (*bs* ! *i*))) + *punit.lt* (*rep-list* (*bs* ! *i*)) = *t* (**is** *?t* = *t*)
by (*rule adds-minus*)
from *r* **have** *lookup* (*rep-list* p') ((*t* − *punit.lt* (*rep-list* (*bs* ! *i*))) + *punit.lt* (*rep-list* (*bs* ! *i*))) = *0*
by (*rule punit.red-single-lookup*[*simplified*])
thus *?thesis* **by** (*simp only*: *eq0 t*[*symmetric*])
qed
qed
show *?case*
proof (*simp add*: *sig-trd-aux.psimps*[*OF 1*(*1*)] *Let-def* p'-*def*[*symmetric*] p''-*def*[*symmetric*], *intro conjI impI*)
assume $p'' = 0$
show ¬ *is-sig-red* ($\prec_t$) ($\preceq$) (*set bs*) p'
proof
assume *is-sig-red* ($\prec_t$) ($\preceq$) (*set bs*) p'
then obtain *b s* **where** *b* ∈ *set bs* **and** *s* ∈ *keys* (*rep-list* p') **and** *rep-list b* ≠ *0*
and *adds*: *punit.lt* (*rep-list b*) *adds s* **and** *s* ⊕ *lt b* $\prec_t$ *punit.lt* (*rep-list b*) ⊕ *lt* p'
by (*rule is-sig-red-addsE*)
let *?s* = *s* − *punit.lt* (*rep-list b*)
from *adds* **have** *eq0*: *?s* + *punit.lt* (*rep-list b*) = *s* **by** (*simp add*: *adds-minus*)
show *False*
proof (*cases t* $\preceq$ *s*)
case *True*
note ‹*b* ∈ *set bs*› ‹*rep-list b* ≠ *0*›
moreover from *True* **have** *t* $\preceq$ *?s* + *punit.lt* (*rep-list b*) **by** (*simp only*: *eq0*)
moreover from *adds* ‹*s* ⊕ *lt b* $\prec_t$ *punit.lt* (*rep-list b*) ⊕ *lt* p'› **have** *?s* ⊕ *lt b* $\prec_t$ *lt* p'
by (*simp add*: *term-is-le-rel-minus*)
ultimately have *lookup* (*rep-list* p') (*?s* + *punit.lt* (*rep-list b*)) = *0* **by** (*rule c*)
hence *s* ∉ *keys* (*rep-list* p') **by** (*simp add*: *eq0 in-keys-iff*)
thus *?thesis* **using** ‹*s* ∈ *keys* (*rep-list* p')› **..**
next

case *False*
hence $s \prec t$ **by** *simp*
hence *lookup* (*rep-list* p') s = *lookup* (*punit.lower* (*rep-list* p') t) s
by (*simp add*: *punit.lookup-lower-when*)
also from ‹$p'' = 0$› **have** ... = *0* **by** (*simp add*: p''*-def*)
finally have $s \notin$ *keys* (*rep-list* p') **by** (*simp add*: *in-keys-iff*)
thus *?thesis* **using** ‹$s \in$ *keys* (*rep-list* p')› **..**
qed
qed
next
assume $p'' \neq 0$
with p'*-def* p''*-def* **show** ¬ *is-sig-red* ($\prec_t$) ($\preceq$) (*set bs*) (*sig-trd-aux* (*punit.lt* p'', p'))
proof (*rule 1*(*2*))
from ‹$p'' \neq 0$› **have** *punit.lt* $p'' \in$ *keys* p'' **by** (*rule punit.lt-in-keys*)
also have ... $\subseteq$ *keys* (*rep-list* p') **by** (*auto simp*: p''*-def punit.keys-lower*)
finally show *fst* (*punit.lt* p'', p') $\in$ *keys* (*rep-list* (*snd* (*punit.lt* p'', p'))) **by** *simp*
next
fix *b s*
assume $b \in$ *set bs* **and** *rep-list* $b \neq 0$
assume *fst* (*punit.lt* p'', p') $\prec s$ + *punit.lt* (*rep-list* b)
and $s \oplus$ *lt* $b \prec_t$ *lt* (*snd* (*punit.lt* p'', p'))
hence *punit.lt* $p'' \prec s$ + *punit.lt* (*rep-list* b) **and** $s \oplus$ *lt* $b \prec_t$ *lt* p' **by** *simp-all*
have *lookup* (*rep-list* p') (s + *punit.lt* (*rep-list* b)) = *0*
proof (*cases* $t \preceq s$ + *punit.lt* (*rep-list* b))
case *True*
with ‹$b \in$ *set bs*› ‹*rep-list* $b \neq 0$› **show** *?thesis* **using** ‹$s \oplus$ *lt* $b \prec_t$ *lt* p'› **by** (*rule c*)
next
case *False*
hence s + *punit.lt* (*rep-list* b) $\prec t$ **by** *simp*
hence *lookup* (*rep-list* p') (s + *punit.lt* (*rep-list* b)) =
lookup (*punit.lower* (*rep-list* p') t) (s + *punit.lt* (*rep-list* b))
by (*simp add*: *punit.lookup-lower-when*)
also have ... = *0*
proof (*rule ccontr*)
assume *lookup* (*punit.lower* (*rep-list* p') t) (s + *punit.lt* (*rep-list* b)) $\neq$ *0*
hence s + *punit.lt* (*rep-list* b) $\preceq$ *punit.lt* (*punit.lower* (*rep-list* p') t)
by (*rule punit.lt-max*)
also have ... = *punit.lt* p'' **by** (*simp only*: p''*-def*)
finally show *False* **using** ‹*punit.lt* $p'' \prec s$ + *punit.lt* (*rep-list* b)› **by** *simp*
qed
finally show *?thesis* **.**
qed
thus *lookup* (*rep-list* (*snd* (*punit.lt* p'', p'))) (s + *punit.lt* (*rep-list* b)) = *0*
by (*simp only*: *snd-conv*)
qed

qed
qed
qed

corollary *sig-trd-irred*: ¬ *is-sig-red* ($\prec_t$) ($\preceq$) (*set bs*) (*sig-trd p*)
unfolding *sig-trd-def*
proof (*split if-split*, *intro conjI impI*)
assume *rep-list p* = *0*
show ¬ *is-sig-red* ($\prec_t$) ($\preceq$) (*set bs*) *p*
proof
assume *is-sig-red* ($\prec_t$) ($\preceq$) (*set bs*) *p*
then obtain *t* **where** *t* ∈ *keys* (*rep-list p*) **by** (*rule is-sig-red-addsE*)
thus *False* **by** (*simp add*: ‹*rep-list p* = *0*›)
qed
next
assume *rep-list p* ≠ *0*
show ¬ *is-sig-red* ($\prec_t$) ($\preceq$) (*set bs*) (*sig-trd-aux* (*punit.lt* (*rep-list p*), *p*))
proof (*rule sig-trd-aux-irred*)
from ‹*rep-list p* ≠ *0*› **have** *punit.lt* (*rep-list p*) ∈ *keys* (*rep-list p*) **by** (*rule punit.lt-in-keys*)
thus *fst* (*punit.lt* (*rep-list p*), *p*) ∈ *keys* (*rep-list* (*snd* (*punit.lt* (*rep-list p*), *p*))) **by** *simp*
next
fix *b s*
assume *fst* (*punit.lt* (*rep-list p*), *p*) ≺ *s* + *punit.lt* (*rep-list b*)
thus *lookup* (*rep-list* (*snd* (*punit.lt* (*rep-list p*), *p*))) (*s* + *punit.lt* (*rep-list b*)) = *0*
using *punit.lt-max* **by** *force*
qed
qed

end

context
fixes *bs* :: (′*t* × (′*a* $\Rightarrow_0$ ′*b*)) *list*
begin

context
fixes *v* :: ′*t*
begin

fun *sig-trd-spp-body* :: ((′*a* $\Rightarrow_0$ ′*b*) × (′*a* $\Rightarrow_0$ ′*b*)) ⇒ ((′*a* $\Rightarrow_0$ ′*b*) × (′*a* $\Rightarrow_0$ ′*b*))
where
sig-trd-spp-body (*p*, *r*) =
(*case find-sig-reducer bs v* (*punit.lt p*) *0 of*
None ⇒ (*punit.tail p*, *r* + *monomial* (*punit.lc p*) (*punit.lt p*))
| *Some i* ⇒ *let b* = *snd* (*bs* ! *i*) *in*
(*punit.tail p* − *punit.monom-mult* (*punit.lc p* / *punit.lc b*) (*punit.lt p* − *punit.lt b*) (*punit.tail b*), *r*))

definition *sig-trd-spp-aux* :: $(('a \Rightarrow_0 'b) \times ('a \Rightarrow_0 'b)) \Rightarrow ('a \Rightarrow_0 'b)$
where *sig-trd-spp-aux-def* [*code del*]: *sig-trd-spp-aux* = *tailrec.fun* ($\lambda x.$ *fst x* = *0*) *snd sig-trd-spp-body*

lemma *sig-trd-spp-aux-simps* [*code*]:
sig-trd-spp-aux (*p*, *r*) = (*if p* = *0 then r else sig-trd-spp-aux* (*sig-trd-spp-body* (*p*, *r*)))
by (*simp add*: *sig-trd-spp-aux-def tailrec.simps*)

end

fun *sig-trd-spp* :: $('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b))$ **where**
sig-trd-spp (*v*, *p*) = (*v*, *sig-trd-spp-aux v* (*p*, *0*))

We define function *sig-trd-spp*, operating on sig-poly-pairs, already here, to have its definition in the right context. Lemmas are proved about it below in Section $Sig-Poly-Pairs$.

end

4.2.9 Koszul Syzygies

A *Koszul syzygy* of the list *fs* of scalar polynomials is a syzygy of the form *fs* ! *i* $\odot$ *monomial 1* (*term-of-pair* (*0*, *j*)) $-$ *fs* ! *j* $\odot$ *monomial 1* (*term-of-pair* (*0*, *i*)), for $i < j$ and $j <$ *length fs*.

primrec *Koszul-syz-sigs-aux* :: $('a \Rightarrow_0 'b)$ *list* $\Rightarrow$ *nat* $\Rightarrow$ *'t list* **where**
Koszul-syz-sigs-aux [] *i* = [] |
Koszul-syz-sigs-aux (*b* # *bs*) *i* =
map-idx ($\lambda b'$ *j*. *ord-term-lin.max* (*term-of-pair* (*punit.lt b*, *j*)) (*term-of-pair* (*punit.lt b'*, *i*))) *bs* (*Suc i*) @
Koszul-syz-sigs-aux bs (*Suc i*)

definition *Koszul-syz-sigs* :: $('a \Rightarrow_0 'b)$ *list* $\Rightarrow$ *'t list*
where *Koszul-syz-sigs bs* = *filter-min* ($adds_t$) (*Koszul-syz-sigs-aux bs 0*)

fun *new-syz-sigs* :: *'t list* $\Rightarrow$ $('t \Rightarrow_0 'b)$ *list* $\Rightarrow$ $(('t \Rightarrow_0 'b) \times ('t \Rightarrow_0 'b)) + nat \Rightarrow$ *'t list*
where
new-syz-sigs ss bs (*Inl* (*a*, *b*)) = *ss* |
new-syz-sigs ss bs (*Inr j*) =
(*if is-pot-ord then*
filter-min-append ($adds_t$) *ss* (*filter-min* ($adds_t$) (*map* ($\lambda b.$ *term-of-pair* (*punit.lt* (*rep-list b*), *j*)) *bs*))
else ss)

fun *new-syz-sigs-spp* :: *'t list* $\Rightarrow$ $('t \times ('a \Rightarrow_0 'b))$ *list* $\Rightarrow$ $(('t \times ('a \Rightarrow_0 'b)) \times ('t \times ('a \Rightarrow_0 'b))) + nat \Rightarrow$ *'t list*
where

new-syz-sigs-spp ss bs (*Inl* (*a*, *b*)) = *ss* |
new-syz-sigs-spp ss bs (*Inr j*) =
(*if is-pot-ord then*
filter-min-append (*adds*$_t$) *ss* (*filter-min* (*adds*$_t$) (*map* (*λb*. *term-of-pair* (*punit.lt* (*snd b*), *j*)) *bs*))
else ss)

lemma *Koszul-syz-sigs-auxI*:
assumes *i* < *j* **and** *j* < *length bs*
shows *ord-term-lin.max* (*term-of-pair* (*punit.lt* (*bs* ! *i*), *k* + *j*)) (*term-of-pair* (*punit.lt* (*bs* ! *j*), *k* + *i*)) ∈
set (*Koszul-syz-sigs-aux bs k*)
using *assms*
proof (*induct bs arbitrary*: *i j k*)
case *Nil*
from *Nil*(*2*) **show** *?case* **by** *simp*
next
case (*Cons b bs*)
from *Cons*(*2*) **obtain** *j0* **where** *j*: *j* = *Suc j0* **by** (*meson lessE*)
from *Cons*(*3*) **have** *j0* < *length bs* **by** (*simp add*: *j*)
let *?A* = (*λj*. *ord-term-lin.max* (*term-of-pair* (*punit.lt b*, *Suc* (*j* + *k*))) (*term-of-pair* (*punit.lt* (*bs* ! *j*), *k*))) ‘
{*0*..<*length bs*}
let *?B* = *set* (*Koszul-syz-sigs-aux bs* (*Suc k*))
show *?case*
proof (*cases i*)
case *0*
from ‹*j0* < *length bs*› **have** *j0* ∈ {*0*..<*length bs*} **by** *simp*
hence *ord-term-lin.max* (*term-of-pair* (*punit.lt b*, *Suc* (*j0* + *k*)))
(*term-of-pair* (*punit.lt* (*bs* ! *j0*), *k*)) ∈ *?A* **by** (*rule imageI*)
thus *?thesis* **by** (*simp add*: ‹*i* = *0*› *j set-map-idx ac-simps*)
next
case (*Suc i0*)
from *Cons*(*2*) **have** *i0* < *j0* **by** (*simp add*: ‹*i* = *Suc i0*› *j*)
hence *ord-term-lin.max* (*term-of-pair* (*punit.lt* (*bs* ! *i0*), *Suc k* + *j0*))
(*term-of-pair* (*punit.lt* (*bs* ! *j0*), *Suc k* + *i0*)) ∈ *?B*
using ‹*j0* < *length bs*› **by** (*rule Cons*(*1*))
thus *?thesis* **by** (*simp add*: ‹*i* = *Suc i0*› *j set-map-idx ac-simps*)
qed
qed

lemma *Koszul-syz-sigs-auxE*:
assumes *v* ∈ *set* (*Koszul-syz-sigs-aux bs k*)
obtains *i j* **where** *i* < *j* **and** *j* < *length bs*
and *v* = *ord-term-lin.max* (*term-of-pair* (*punit.lt* (*bs* ! *i*), *k* + *j*)) (*term-of-pair* (*punit.lt* (*bs* ! *j*), *k* + *i*))
using *assms*
proof (*induct bs arbitrary*: *k thesis*)
case *Nil*

from *Nil*(*2*) **show** *?case* **by** *simp*
next
case (*Cons b bs*)
have *v* ∈ (*λj. ord-term-lin.max* (*term-of-pair* (*punit.lt b*, *Suc* (*j* + *k*))) (*term-of-pair* (*punit.lt* (*bs* ! *j*), *k*))) ‘
{*0..<length bs*} ∪ *set* (*Koszul-syz-sigs-aux bs* (*Suc k*)) (**is** *v* ∈ *?A* ∪ *?B*)
using *Cons*(*3*) **by** (*simp add*: *set-map-idx*)
thus *?case*
proof
assume *v* ∈ *?A*
then obtain *j* **where** *j* ∈ {*0..<length bs*}
and *v*: *v* = *ord-term-lin.max* (*term-of-pair* (*punit.lt b*, *Suc* (*j* + *k*)))
(*term-of-pair* (*punit.lt* (*bs* ! *j*), *k*)) **..**
from *this*(*1*) **have** *j* < *length bs* **by** *simp*
show *?thesis*
proof (*rule Cons*(*2*))
show *0* < *Suc j* **by** *simp*
next
from ‹*j* < *length bs*› **show** *Suc j* < *length* (*b* # *bs*) **by** *simp*
next
show *v* = *ord-term-lin.max* (*term-of-pair* (*punit.lt* ((*b* # *bs*) ! *0*), *k* + *Suc j*))
(*term-of-pair* (*punit.lt* ((*b* # *bs*) ! *Suc j*), *k* + *0*))
by (*simp add*: *v ac-simps*)
qed
next
assume *v* ∈ *?B*
obtain *i j* **where** *i* < *j* **and** *j* < *length bs*
and *v*: *v* = *ord-term-lin.max* (*term-of-pair* (*punit.lt* (*bs* ! *i*), *Suc k* + *j*))
(*term-of-pair* (*punit.lt* (*bs* ! *j*), *Suc k* + *i*))
by (*rule Cons*(*1*), *assumption*, *rule* ‹*v* ∈ *?B*›)
show *?thesis*
proof (*rule Cons*(*2*))
from ‹*i* < *j*› **show** *Suc i* < *Suc j* **by** *simp*
next
from ‹*j* < *length bs*› **show** *Suc j* < *length* (*b* # *bs*) **by** *simp*
next
show *v* = *ord-term-lin.max* (*term-of-pair* (*punit.lt* ((*b* # *bs*) ! *Suc i*), *k* + *Suc j*))
(*term-of-pair* (*punit.lt* ((*b* # *bs*) ! *Suc j*), *k* + *Suc i*))
by (*simp add*: *v*)
qed
qed
qed

lemma *lt-Koszul-syz-comp*:
assumes *0* ∉ *set fs* **and** *i* < *length fs*
shows *lt* ((*fs* ! *i*) ⊙ *monomial 1* (*term-of-pair* (*0*, *j*))) = *term-of-pair* (*punit.lt*

(*fs* ! *i*), *j*)
proof −
  from *assms*(*2*) **have** *fs* ! *i* ∈ *set fs* **by** (*rule nth-mem*)
  with *assms*(*1*) **have** *fs* ! *i* ≠ *0* **by** *auto*
  thus *?thesis* **by** (*simp add*: *lt-mult-scalar-monomial-right splus-def term-simps*)
qed

lemma *Koszul-syz-nonzero-lt*:
  assumes *rep-list a* ≠ *0* **and** *rep-list b* ≠ *0* **and** *component-of-term* (*lt a*) < *component-of-term* (*lt b*)
  shows *rep-list a* ⊙ *b* − *rep-list b* ⊙ *a* ≠ *0* (**is** *?p* − *?q* ≠ *0*)
    and *lt* (*rep-list a* ⊙ *b* − *rep-list b* ⊙ *a*) =
        ord-term-lin.max (*punit.lt* (*rep-list a*) ⊕ *lt b*) (*punit.lt* (*rep-list b*) ⊕ *lt a*)
(**is** - = *?r*)
proof −
  from *assms*(*2*) **have** *b* ≠ *0* **by** (*auto simp*: *rep-list-zero*)
  with *assms*(*1*) **have** *lt-p*: *lt ?p* = *punit.lt* (*rep-list a*) ⊕ *lt b* **by** (*rule lt-mult-scalar*)
  from *assms*(*1*) **have** *a* ≠ *0* **by** (*auto simp*: *rep-list-zero*)
  with *assms*(*2*) **have** *lt-q*: *lt ?q* = *punit.lt* (*rep-list b*) ⊕ *lt a* **by** (*rule lt-mult-scalar*)
  from *assms*(*3*) **have** *component-of-term* (*lt ?p*) ≠ *component-of-term* (*lt ?q*)
    by (*simp add*: *lt-p lt-q component-of-term-splus*)
  hence *lt ?p* ≠ *lt ?q* **by** *auto*
  hence *lt* (*?p* − *?q*) = *ord-term-lin.max* (*lt ?p*) (*lt ?q*) **by** (*rule lt-minus-distinct-eq-max*)
  also have ... = *?r* **by** (*simp only*: *lt-p lt-q*)
  finally show *lt* (*?p* − *?q*) = *?r* **.**

  from ‹*lt ?p* ≠ *lt ?q*› **show** *?p* − *?q* ≠ *0* **by** *auto*
qed

lemma *Koszul-syz-is-syz*: *rep-list* (*rep-list a* ⊙ *b* − *rep-list b* ⊙ *a*) = *0*
  by (*simp add*: *rep-list-minus rep-list-mult-scalar*)

lemma *dgrad-sig-set-closed-Koszul-syz*:
  assumes *dickson-grading dgrad* **and** *a* ∈ *dgrad-sig-set dgrad* **and** *b* ∈ *dgrad-sig-set dgrad*
  shows *rep-list a* ⊙ *b* − *rep-list b* ⊙ *a* ∈ *dgrad-sig-set dgrad*
proof −
  from *assms*(*2*, *3*) **have** *1*: *a* ∈ *dgrad-max-set dgrad* **and** *2*: *b* ∈ *dgrad-max-set dgrad*
    by (*simp-all add*: *dgrad-sig-set′-def*)
  show *?thesis*
    by (*intro dgrad-sig-set-closed-minus dgrad-sig-set-closed-mult-scalar dgrad-max-2 assms 1 2*)
qed

corollary *Koszul-syz-is-syz-sig*:
  assumes *dickson-grading dgrad* **and** *a* ∈ *dgrad-sig-set dgrad* **and** *b* ∈ *dgrad-sig-set dgrad*
    and *rep-list a* ≠ *0* **and** *rep-list b* ≠ *0* **and** *component-of-term* (*lt a*) < *compo-*

nent-of-term (*lt b*)
  shows *is-syz-sig dgrad* (*ord-term-lin.max* (*punit.lt* (*rep-list a*) ⊕ *lt b*) (*punit.lt* (*rep-list b*) ⊕ *lt a*))
proof (*rule is-syz-sigI*)
  from *assms*(*4*−*6*) **show** *rep-list a* ⊙ *b* − *rep-list b* ⊙ *a* ≠ *0*
    and *lt* (*rep-list a* ⊙ *b* − *rep-list b* ⊙ *a*) =
        ord-term-lin.max (*punit.lt* (*rep-list a*) ⊕ *lt b*) (*punit.lt* (*rep-list b*) ⊕ *lt a*)
    by (*rule Koszul-syz-nonzero-lt*)+
next
  from *assms*(*1*−*3*) **show** *rep-list a* ⊙ *b* − *rep-list b* ⊙ *a* ∈ *dgrad-sig-set dgrad*
    by (*rule dgrad-sig-set-closed-Koszul-syz*)
qed (*fact Koszul-syz-is-syz*)

corollary *lt-Koszul-syz-in-Koszul-syz-sigs-aux*:
  assumes *distinct fs* **and** *0* ∉ *set fs* **and** *i* < *j* **and** *j* < *length fs*
  shows *lt* ((*fs* ! *i*) ⊙ *monomial 1* (*term-of-pair* (*0*, *j*)) − (*fs* ! *j*) ⊙ *monomial 1* (*term-of-pair* (*0*, *i*))) ∈
        set (*Koszul-syz-sigs-aux fs 0*) (**is** *?l* ∈ *?K*)
proof −
  let *?a* = *monomial* (*1*::′*b*) (*term-of-pair* (*0*, *i*))
  let *?b* = *monomial* (*1*::′*b*) (*term-of-pair* (*0*, *j*))
  from *assms*(*3*, *4*) **have** *i* < *length fs* **by** *simp*
   with *assms*(*1*) **have** *a*: *rep-list ?a* = *fs* ! *i* **by** (*simp add*: *rep-list-monomial term-simps*)
  from *assms*(*1*, *4*) **have** *b*: *rep-list ?b* = *fs* ! *j* **by** (*simp add*: *rep-list-monomial term-simps*)
  have *?l* = *lt* (*rep-list ?a* ⊙ *?b* − *rep-list ?b* ⊙ *?a*) **by** (*simp only*: *a b*)
  also have ... = *ord-term-lin.max* (*punit.lt* (*rep-list ?a*) ⊕ *lt ?b*) (*punit.lt* (*rep-list ?b*) ⊕ *lt ?a*)
  proof (*rule Koszul-syz-nonzero-lt*)
    from ‹*i* < *length fs*› **have** *fs* ! *i* ∈ *set fs* **by** (*rule nth-mem*)
    with *assms*(*2*) **show** *rep-list ?a* ≠ *0* **by** (*auto simp*: *a*)
  next
    from *assms*(*4*) **have** *fs* ! *j* ∈ *set fs* **by** (*rule nth-mem*)
    with *assms*(*2*) **show** *rep-list ?b* ≠ *0* **by** (*auto simp*: *b*)
  next
    from *assms*(*3*) **show** *component-of-term* (*lt ?a*) < *component-of-term* (*lt ?b*)
      by (*simp add*: *lt-monomial component-of-term-of-pair*)
  qed
 also have ... = *ord-term-lin.max* (*term-of-pair* (*punit.lt* (*fs* ! *i*), *0* + *j*)) (*term-of-pair* (*punit.lt* (*fs* ! *j*), *0* + *i*))
    by (*simp add*: *a b lt-monomial splus-def term-simps*)
  also from *assms*(*3*, *4*) **have** ... ∈ *?K* **by** (*rule Koszul-syz-sigs-auxI*)
  thm *Koszul-syz-sigs-auxI*[*OF assms*(*3*, *4*)]
  finally show *?thesis* **.**
qed

corollary *lt-Koszul-syz-in-Koszul-syz-sigs*:
  assumes ¬ *is-pot-ord* **and** *distinct fs* **and** *0* ∉ *set fs* **and** *i* < *j* **and** *j* < *length*

fs
obtains *v* **where** $v \in$ *set* (*Koszul-syz-sigs fs*)
and *v* $adds_t$ *lt* ((*fs* ! *i*) ⊙ *monomial 1* (*term-of-pair* (*0*, *j*)) − (*fs* ! *j*) ⊙ *monomial 1* (*term-of-pair* (*0*, *i*)))
proof −
have *transp* ($adds_t$) **by** (*rule transpI*, *drule adds-term-trans*)
moreover have *lt* ((*fs* ! *i*) ⊙ *monomial 1* (*term-of-pair* (*0*, *j*)) − (*fs* ! *j*) ⊙ *monomial 1* (*term-of-pair* (*0*, *i*))) ∈
set (*Koszul-syz-sigs-aux fs 0*) (**is** *?l* ∈ *set ?ks*)
using *assms*(*2*−*5*) **by** (*rule lt-Koszul-syz-in-Koszul-syz-sigs-aux*)
ultimately show *?thesis*
proof (*rule filter-min-cases*)
assume *?l* ∈ *set* (*filter-min* ($adds_t$) *?ks*)
hence *?l* ∈ *set* (*Koszul-syz-sigs fs*) **by** (*simp add*: *Koszul-syz-sigs-def assms*(*1*))
thus *?thesis* **using** *adds-term-refl* **..**
next
fix *v*
assume *v* ∈ *set* (*filter-min* ($adds_t$) *?ks*)
hence *v* ∈ *set* (*Koszul-syz-sigs fs*) **by** (*simp add*: *Koszul-syz-sigs-def assms*(*1*))
moreover assume *v* $adds_t$ *?l*
ultimately show *?thesis* **..**
qed
qed

lemma *lt-Koszul-syz-init*:
assumes *0* ∉ *set fs* **and** *i* < *j* **and** *j* < *length fs*
shows *lt* ((*fs* ! *i*) ⊙ *monomial 1* (*term-of-pair* (*0*, *j*)) − (*fs* ! *j*) ⊙ *monomial 1* (*term-of-pair* (*0*, *i*))) =
ord-term-lin.max (*term-of-pair* (*punit.lt* (*fs* ! *i*), *j*)) (*term-of-pair* (*punit.lt* (*fs* ! *j*), *i*))
(**is** *lt* (*?p* − *?q*) = *?r*)
proof −
from *assms*(*2*, *3*) **have** *i* < *length fs* **by** *simp*
with *assms*(*1*) **have** *lt-i*: *lt ?p* = *term-of-pair* (*punit.lt* (*fs* ! *i*), *j*) **by** (*rule lt-Koszul-syz-comp*)
from *assms*(*1*, *3*) **have** *lt-j*: *lt ?q* = *term-of-pair* (*punit.lt* (*fs* ! *j*), *i*) **by** (*rule lt-Koszul-syz-comp*)
from *assms*(*2*) **have** *component-of-term* (*lt ?p*) ≠ *component-of-term* (*lt ?q*)
by (*simp add*: *lt-i lt-j component-of-term-of-pair*)
hence *lt ?p* ≠ *lt ?q* **by** *auto*
hence *lt* (*?p* − *?q*) = *ord-term-lin.max* (*lt ?p*) (*lt ?q*) **by** (*rule lt-minus-distinct-eq-max*)
also have ... = *?r* **by** (*simp only*: *lt-i lt-j*)
finally show *?thesis* **.**
qed

corollary *Koszul-syz-sigs-auxE-lt-Koszul-syz*:
assumes *0* ∉ *set fs* **and** *v* ∈ *set* (*Koszul-syz-sigs-aux fs 0*)
obtains *i j* **where** *i* < *j* **and** *j* < *length fs*
and *v* = *lt* ((*fs* ! *i*) ⊙ *monomial 1* (*term-of-pair* (*0*, *j*)) − (*fs* ! *j*) ⊙ *monomial*

1 (*term-of-pair* (*0*, *i*)))
proof −
  from *assms*(*2*) **obtain** *i j* **where** *i* < *j* **and** *j* < *length fs*
    and *v* = *ord-term-lin.max* (*term-of-pair* (*punit.lt* (*fs* ! *i*), *0* + *j*))
                    (*term-of-pair* (*punit.lt* (*fs* ! *j*), *0* + *i*))
    by (*rule Koszul-syz-sigs-auxE*)
  with *assms*(*1*) **have** *v* = *lt* ((*fs* ! *i*) ⊙ *monomial 1* (*term-of-pair* (*0*, *j*)) −
                    (*fs* ! *j*) ⊙ *monomial 1* (*term-of-pair* (*0*, *i*)))
    by (*simp add*: *lt-Koszul-syz-init*)
  with ‹*i* < *j*› ‹*j* < *length fs*› **show** *?thesis* **..**
qed

corollary *Koszul-syz-sigs-is-syz-sig*:
  assumes *dickson-grading dgrad* **and** *distinct fs* **and** *0* ∉ *set fs* **and** *v* ∈ *set* (*Koszul-syz-sigs fs*)
  shows *is-syz-sig dgrad v*
proof −
  from *assms*(*4*) **have** *v* ∈ *set* (*Koszul-syz-sigs-aux fs 0*)
    using *filter-min-subset* **by** (*fastforce simp*: *Koszul-syz-sigs-def*)
  with *assms*(*3*) **obtain** *i j* **where** *i* < *j* **and** *j* < *length fs*
   and *v'*: *v* = *lt* ((*fs* ! *i*) ⊙ *monomial 1* (*term-of-pair* (*0*, *j*)) − (*fs* ! *j*) ⊙ *monomial 1* (*term-of-pair* (*0*, *i*)))
         (**is** *v* = *lt* (*?p* − *?q*))
    by (*rule Koszul-syz-sigs-auxE-lt-Koszul-syz*)
  let *?a* = *monomial* (*1*::'*b*) (*term-of-pair* (*0*, *i*))
  let *?b* = *monomial* (*1*::'*b*) (*term-of-pair* (*0*, *j*))
  from ‹*i* < *j*› ‹*j* < *length fs*› **have** *i* < *length fs* **by** *simp*
  with *assms*(*2*) **have** *a*: *rep-list ?a* = *fs* ! *i* **by** (*simp add*: *rep-list-monomial term-simps*)
  from *assms*(*2*) ‹*j* < *length fs*› **have** *b*: *rep-list ?b* = *fs* ! *j* **by** (*simp add*: *rep-list-monomial term-simps*)
  note *v'*
  also have *lt* (*?p* − *?q*) = *ord-term-lin.max* (*term-of-pair* (*punit.lt* (*fs* ! *i*), *j*)) (*term-of-pair* (*punit.lt* (*fs* ! *j*), *i*))
    using *assms*(*3*) ‹*i* < *j*› ‹*j* < *length fs*› **by** (*rule lt-Koszul-syz-init*)
  also have ... = *ord-term-lin.max* (*punit.lt* (*rep-list ?a*) ⊕ *lt ?b*) (*punit.lt* (*rep-list ?b*) ⊕ *lt ?a*)
    by (*simp add*: *a b lt-monomial splus-def term-simps*)
  finally have *v*: *v* = *ord-term-lin.max* (*punit.lt* (*rep-list ?a*) ⊕ *lt ?b*) (*punit.lt* (*rep-list ?b*) ⊕ *lt ?a*) **.**
  show *?thesis* **unfolding** *v* **using** *assms*(*1*)
  proof (*rule Koszul-syz-is-syz-sig*)
    show *?a* ∈ *dgrad-sig-set dgrad*
     by (*rule dgrad-sig-set-closed-monomial*, *simp-all add*: *term-simps dgrad-max-0* ‹*i* < *length fs*›)
  next
    show *?b* ∈ *dgrad-sig-set dgrad*
     by (*rule dgrad-sig-set-closed-monomial*, *simp-all add*: *term-simps dgrad-max-0* ‹*j* < *length fs*›)

next
  from ‹i < *length fs*› **have** *fs* ! i ∈ *set fs* **by** (*rule nth-mem*)
  with *assms*(*3*) **show** *rep-list ?a* ≠ *0* **by** (*fastforce simp*: *a*)
next
  from ‹j < *length fs*› **have** *fs* ! j ∈ *set fs* **by** (*rule nth-mem*)
  with *assms*(*3*) **show** *rep-list ?b* ≠ *0* **by** (*fastforce simp*: *b*)
next
  from ‹i < j› **show** *component-of-term* (*lt ?a*) < *component-of-term* (*lt ?b*)
    by (*simp add*: *lt-monomial component-of-term-of-pair*)
qed
qed

lemma *Koszul-syz-sigs-minimal*:
  assumes u ∈ *set* (*Koszul-syz-sigs fs*) **and** v ∈ *set* (*Koszul-syz-sigs fs*) **and** u $adds_t$ v
  shows $u = v$
proof −
  from *assms*(*1*, *2*) **have** u ∈ *set* (*filter-min* ($adds_t$) (*Koszul-syz-sigs-aux fs 0*))
    and v ∈ *set* (*filter-min* ($adds_t$) (*Koszul-syz-sigs-aux fs 0*)) **by** (*simp-all add*: *Koszul-syz-sigs-def*)
  with - **show** *?thesis* **using** *assms*(*3*)
  proof (*rule filter-min-minimal*)
    show *transp* ($adds_t$) **by** (*rule transpI*, *drule adds-term-trans*)
  qed
qed

lemma *Koszul-syz-sigs-distinct*: *distinct* (*Koszul-syz-sigs fs*)
proof −
  from *adds-term-refl* **have** *reflp* ($adds_t$) **by** (*rule reflpI*)
  thus *?thesis* **by** (*simp add*: *Koszul-syz-sigs-def filter-min-distinct*)
qed

4.2.10 Algorithms

definition *spair-spp* :: ('t × ('a ⇒$_0$ 'b)) ⇒ ('t × ('a ⇒$_0$ 'b)) ⇒ ('t × ('a ⇒$_0$ 'b))
  where *spair-spp p q* = (*let t1* = *punit.lt* (*snd p*); *t2* = *punit.lt* (*snd q*); *l* = *lcs t1 t2 in*
      (*ord-term-lin.max* ((*l* − *t1*) ⊕ *fst p*) ((*l* − *t2*) ⊕ *fst q*),
      punit.monom-mult (*1* / *punit.lc* (*snd p*)) (*l* − *t1*) (*snd p*) −
      punit.monom-mult (*1* / *punit.lc* (*snd q*)) (*l* − *t2*) (*snd q*)))

definition *is-regular-spair-spp* :: ('t × ('a ⇒$_0$ 'b)) ⇒ ('t × ('a ⇒$_0$ 'b)) ⇒ *bool*
  where *is-regular-spair-spp p q* ⟷
      (*snd p* ≠ *0* ∧ *snd q* ≠ *0* ∧ *punit.lt* (*snd q*) ⊕ *fst p* ≠ *punit.lt* (*snd p*) ⊕ *fst q*)

definition *spair-sigs* :: ('t ⇒$_0$ 'b) ⇒ ('t ⇒$_0$ 'b) ⇒ ('t × 't)
  where *spair-sigs p q* =
      (*let t1* = *punit.lt* (*rep-list p*); *t2* = *punit.lt* (*rep-list q*); *l* = *lcs t1 t2*

in
$((l - t1) \oplus lt\ p, (l - t2) \oplus lt\ q))$

definition *spair-sigs-spp* :: $('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times 't)$
where *spair-sigs-spp p q* =
$(let\ t1 = punit.lt\ (snd\ p);\ t2 = punit.lt\ (snd\ q);\ l = lcs\ t1\ t2\ in$
$((l - t1) \oplus fst\ p, (l - t2) \oplus fst\ q))$

fun *poly-of-pair* :: $((('t \Rightarrow_0 'b) \times ('t \Rightarrow_0 'b)) + nat) \Rightarrow ('t \Rightarrow_0 'b)$
where
poly-of-pair (*Inl* (*p*, *q*)) = *spair p q* |
poly-of-pair (*Inr j*) = *monomial 1* (*term-of-pair* (*0*, *j*))

fun *spp-of-pair* :: $((('t \times ('a \Rightarrow_0 'b)) \times ('t \times ('a \Rightarrow_0 'b))) + nat) \Rightarrow ('t \times ('a \Rightarrow_0 'b))$
where
spp-of-pair (*Inl* (*p*, *q*)) = *spair-spp p q* |
spp-of-pair (*Inr j*) = (*term-of-pair* (*0*, *j*), *fs* ! *j*)

fun *sig-of-pair* :: $((('t \Rightarrow_0 'b) \times ('t \Rightarrow_0 'b)) + nat) \Rightarrow 't$
where
sig-of-pair (*Inl* (*p*, *q*)) = (*let* (*u*, *v*) = *spair-sigs p q in ord-term-lin.max u v*) |
sig-of-pair (*Inr j*) = *term-of-pair* (*0*, *j*)

fun *sig-of-pair-spp* :: $((('t \times ('a \Rightarrow_0 'b)) \times ('t \times ('a \Rightarrow_0 'b))) + nat) \Rightarrow 't$
where
sig-of-pair-spp (*Inl* (*p*, *q*)) = (*let* (*u*, *v*) = *spair-sigs-spp p q in ord-term-lin.max u v*) |
sig-of-pair-spp (*Inr j*) = *term-of-pair* (*0*, *j*)

definition *pair-ord* :: $((('t \Rightarrow_0 'b) \times ('t \Rightarrow_0 'b)) + nat) \Rightarrow ((('t \Rightarrow_0 'b) \times ('t \Rightarrow_0 'b)) + nat) \Rightarrow bool$
where *pair-ord x y* $\longleftrightarrow$ (*sig-of-pair x* $\preceq_t$ *sig-of-pair y*)

definition *pair-ord-spp* :: $((('t \times ('a \Rightarrow_0 'b)) \times ('t \times ('a \Rightarrow_0 'b))) + nat) \Rightarrow$
$((('t \times ('a \Rightarrow_0 'b)) \times ('t \times ('a \Rightarrow_0 'b))) + nat) \Rightarrow bool$
where *pair-ord-spp x y* $\longleftrightarrow$ (*sig-of-pair-spp x* $\preceq_t$ *sig-of-pair-spp y*)

primrec *new-spairs* :: $('t \Rightarrow_0 'b)\ list \Rightarrow ('t \Rightarrow_0 'b) \Rightarrow ((('t \Rightarrow_0 'b) \times ('t \Rightarrow_0 'b)) + nat)\ list$ **where**
new-spairs [] *p* = [] |
new-spairs (*b* # *bs*) *p* =
(*if is-regular-spair p b then insort-wrt pair-ord* (*Inl* (*p*, *b*)) (*new-spairs bs p*) *else new-spairs bs p*)

primrec *new-spairs-spp* :: $('t \times ('a \Rightarrow_0 'b))\ list \Rightarrow ('t \times ('a \Rightarrow_0 'b)) \Rightarrow$
$((('t \times ('a \Rightarrow_0 'b)) \times ('t \times ('a \Rightarrow_0 'b))) + nat)\ list$ **where**
new-spairs-spp [] *p* = [] |
new-spairs-spp (*b* # *bs*) *p* =

(*if is-regular-spair-spp p b then*
insort-wrt pair-ord-spp (*Inl* (*p*, *b*)) (*new-spairs-spp bs p*)
else new-spairs-spp bs p)

definition *add-spairs* :: ((('*t* $\Rightarrow_0$ '*b*) × ('*t* $\Rightarrow_0$ '*b*)) + *nat*) *list* ⇒ ('*t* $\Rightarrow_0$ '*b*) *list* ⇒ ('*t* $\Rightarrow_0$ '*b*) ⇒
((('*t* $\Rightarrow_0$ '*b*) × ('*t* $\Rightarrow_0$ '*b*)) + *nat*) *list*
where *add-spairs ps bs p* = *merge-wrt pair-ord* (*new-spairs bs p*) *ps*

definition *add-spairs-spp* :: ((('*t* × ('*a* $\Rightarrow_0$ '*b*)) × ('*t* × ('*a* $\Rightarrow_0$ '*b*))) + *nat*) *list* ⇒
('*t* × ('*a* $\Rightarrow_0$ '*b*)) *list* ⇒ ('*t* × ('*a* $\Rightarrow_0$ '*b*)) ⇒
((('*t* × ('*a* $\Rightarrow_0$ '*b*)) × ('*t* × ('*a* $\Rightarrow_0$ '*b*))) + *nat*) *list*
where *add-spairs-spp ps bs p* = *merge-wrt pair-ord-spp* (*new-spairs-spp bs p*) *ps*

lemma *spair-alt-spair-sigs*:
spair p q = *monom-mult* (*1* / *punit.lc* (*rep-list p*)) (*pp-of-term* (*fst* (*spair-sigs p q*)) − *lp p*) *p* −
monom-mult (*1* / *punit.lc* (*rep-list q*)) (*pp-of-term* (*snd* (*spair-sigs p q*)) − *lp q*) *q*
by (*simp add*: *spair-def spair-sigs-def Let-def term-simps*)

lemma *sig-of-spair*:
assumes *is-regular-spair p q*
shows *sig-of-pair* (*Inl* (*p*, *q*)) = *lt* (*spair p q*)
proof −
from *assms* **have** *rep-list p* ≠ *0* **by** (*rule is-regular-spairD1*)
hence *1*: *punit.lc* (*rep-list p*) ≠ *0* **and** *p* ≠ *0* **by** (*rule punit.lc-not-0*, *auto simp*: *rep-list-zero*)
from *assms* **have** *rep-list q* ≠ *0* **by** (*rule is-regular-spairD2*)
hence *2*: *punit.lc* (*rep-list q*) ≠ *0* **and** *q* ≠ *0* **by** (*rule punit.lc-not-0*, *auto simp*: *rep-list-zero*)
let *?t1* = *punit.lt* (*rep-list p*)
let *?t2* = *punit.lt* (*rep-list q*)
let *?l* = *lcs ?t1 ?t2*
from *assms* **have** *lt* (*monom-mult* (*1* / *punit.lc* (*rep-list p*)) (*?l* − *?t1*) *p*) ≠
lt (*monom-mult* (*1* / *punit.lc* (*rep-list q*)) (*?l* − *?t2*) *q*)
by (*rule is-regular-spairD3*)
hence ∗: *lt* (*monom-mult* (*1* / *punit.lc* (*rep-list p*)) (*pp-of-term* (*fst* (*spair-sigs p q*)) − *lp p*) *p*) ≠
lt (*monom-mult* (*1* / *punit.lc* (*rep-list q*)) (*pp-of-term* (*snd* (*spair-sigs p q*)) − *lp q*) *q*)
by (*simp add*: *spair-sigs-def Let-def term-simps*)
from *1 2* ‹*p* ≠ *0*› ‹*q* ≠ *0*› **show** *?thesis*
by (*simp add*: *spair-alt-spair-sigs lt-monom-mult lt-minus-distinct-eq-max*[*OF* ∗],
simp add: *spair-sigs-def Let-def term-simps*)
qed

lemma *sig-of-spair-commute*: *sig-of-pair* (*Inl* (*p*, *q*)) = *sig-of-pair* (*Inl* (*q*, *p*))

by (*simp add*: *spair-sigs-def Let-def lcs-comm ord-term-lin.max.commute*)

lemma *in-new-spairsI*:
  assumes $b \in$ *set bs* **and** *is-regular-spair p b*
  shows *Inl* $(p, b) \in$ *set* (*new-spairs bs p*)
  using *assms*(*1*)
proof (*induct bs*)
  case *Nil*
  thus *?case* **by** *simp*
next
  case (*Cons a bs*)
  from *Cons*(*2*) **have** $b = a \vee b \in$ *set bs* **by** *simp*
  thus *?case*
  proof
    assume $b = a$
    from *assms*(*2*) **show** *?thesis* **by** (*simp add*: ‹$b = a$›)
  next
    assume $b \in$ *set bs*
    hence *Inl* $(p, b) \in$ *set* (*new-spairs bs p*) **by** (*rule Cons*(*1*))
    thus *?thesis* **by** *simp*
  qed
qed

lemma *in-new-spairsD*:
  assumes *Inl* $(a, b) \in$ *set* (*new-spairs bs p*)
  shows $a = p$ **and** $b \in$ *set bs* **and** *is-regular-spair p b*
proof −
  from *assms* **have** $a = p \wedge b \in$ *set bs* $\wedge$ *is-regular-spair p b*
  proof (*induct bs*)
  case *Nil*
  thus *?case* **by** *simp*
  next
    case (*Cons c bs*)
    from *Cons*(*2*) **have** (*is-regular-spair p c* $\wedge$ *Inl* $(a, b) =$ *Inl* $(p, c)) \vee$ *Inl* (a, b)
$\in$ *set* (*new-spairs bs p*)
      by (*simp split*: *if-split-asm*)
    thus *?case*
    proof
      assume *is-regular-spair p c* $\wedge$ *Inl* $(a, b) =$ *Inl* (p, c)
      hence *is-regular-spair p c* **and** $a = p$ **and** $b = c$ **by** *simp-all*
      thus *?thesis* **by** *simp*
    next
      assume *Inl* $(a, b) \in$ *set* (*new-spairs bs p*)
      hence $a = p \wedge b \in$ *set bs* $\wedge$ *is-regular-spair p b* **by** (*rule Cons*(*1*))
      thus *?thesis* **by** *simp*
    qed
  qed
  thus $a = p$ **and** $b \in$ *set bs* **and** *is-regular-spair p b* **by** *simp-all*
qed

corollary *in-new-spairs-iff*:
  Inl (*p*, *b*) ∈ *set* (*new-spairs bs p*) ⟷ (*b* ∈ *set bs* ∧ *is-regular-spair p b*)
  by (*auto intro*: *in-new-spairsI dest*: *in-new-spairsD*)

lemma *Inr-not-in-new-spairs*: *Inr j* ∉ *set* (*new-spairs bs p*)
  by (*induct bs*, *simp-all*)

lemma *sum-prodE*:
  assumes ⋀*a b*. *p* = *Inl* (*a*, *b*) ⟹ *thesis* **and** ⋀*j*. *p* = *Inr j* ⟹ *thesis*
  shows *thesis*
  using - *assms*(*2*)
proof (*rule sumE*)
  fix *x*
  assume *p* = *Inl x*
  moreover obtain *a b* **where** *x* = (*a*, *b*) **by** *fastforce*
  ultimately have *p* = *Inl* (*a*, *b*) **by** *simp*
  thus *?thesis* **by** (*rule assms*(*1*))
qed

corollary *in-new-spairsE*:
  assumes *q* ∈ *set* (*new-spairs bs p*)
  obtains *b* **where** *b* ∈ *set bs* **and** *is-regular-spair p b* **and** *q* = *Inl* (*p*, *b*)
proof (*rule sum-prodE*)
  fix *a b*
  assume *q*: *q* = *Inl* (*a*, *b*)
  from *assms* **have** *a* = *p* **and** *b* ∈ *set bs* **and** *is-regular-spair p b*
    unfolding *q* **by** (*rule in-new-spairsD*)+
  note *this*(*2*, *3*)
  moreover have *q* = *Inl* (*p*, *b*) **by** (*simp only*: *q* ‹*a* = *p*›)
  ultimately show *?thesis* **..**
next
  fix *j*
  assume *q* = *Inr j*
  with *assms* **show** *?thesis* **by** (*simp add*: *Inr-not-in-new-spairs*)
qed

lemma *new-spairs-sorted*: *sorted-wrt pair-ord* (*new-spairs bs p*)
proof (*induct bs*)
  case *Nil*
  show *?case* **by** *simp*
next
  case (*Cons a bs*)
  moreover have *transp pair-ord* **by** (*rule transpI*, *simp add*: *pair-ord-def*)
  moreover have *pair-ord x y* ∨ *pair-ord y x* **for** *x y* **by** (*simp add*: *pair-ord-def*
ord-term-lin.linear)
  ultimately show *?case* **by** (*simp add*: *sorted-wrt-insort-wrt*)
qed

lemma *sorted-add-spairs*:
assumes *sorted-wrt pair-ord ps*
shows *sorted-wrt pair-ord* (*add-spairs ps bs p*)
unfolding *add-spairs-def* **using** - - *new-spairs-sorted assms*
proof (*rule sorted-merge-wrt*)
show *transp pair-ord* **by** (*rule transpI*, *simp add*: *pair-ord-def*)
next
fix *x y*
show *pair-ord x y* ∨ *pair-ord y x* **by** (*simp add*: *pair-ord-def ord-term-lin.linear*)
qed

context
fixes *rword-strict* :: $('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b)) \Rightarrow$ *bool* — Must be a *strict* rewrite order.
begin

qualified definition *rword* :: $('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b)) \Rightarrow$ *bool*
where *rword x y* ⟷ ¬ *rword-strict y x*

definition *is-pred-syz* :: *'t list* ⇒ *'t* ⇒ *bool*
where *is-pred-syz ss u* = (∃ *v*∈*set ss*. *v* $adds_t$ *u*)

definition *is-rewritable* :: $('t \Rightarrow_0 'b)$ *list* ⇒ $('t \Rightarrow_0 'b) \Rightarrow 't \Rightarrow$ *bool*
where *is-rewritable bs p u* = (∃ *b*∈*set bs*. *b* ≠ *0* ∧ *lt b* $adds_t$ *u* ∧ *rword-strict* (*spp-of p*) (*spp-of b*))

definition *is-rewritable-spp* :: $('t \times ('a \Rightarrow_0 'b))$ *list* ⇒ $('t \times ('a \Rightarrow_0 'b)) \Rightarrow 't \Rightarrow$ *bool*
where *is-rewritable-spp bs p u* = (∃ *b*∈*set bs*. *fst b* $adds_t$ *u* ∧ *rword-strict p b*)

fun *sig-crit* :: $('t \Rightarrow_0 'b)$ *list* ⇒ *'t list* ⇒ $((('t \Rightarrow_0 'b) \times ('t \Rightarrow_0 'b)) + nat) \Rightarrow$ *bool*
where
sig-crit bs ss (*Inl* (*p*, *q*)) =
(*let* (*u*, *v*) = *spair-sigs p q in*
is-pred-syz ss u ∨ *is-pred-syz ss v* ∨ *is-rewritable bs p u* ∨ *is-rewritable bs q v*) |
sig-crit bs ss (*Inr j*) = *is-pred-syz ss* (*term-of-pair* (*0*, *j*))

fun *sig-crit'* :: $('t \Rightarrow_0 'b)$ *list* ⇒ $((('t \Rightarrow_0 'b) \times ('t \Rightarrow_0 'b)) + nat) \Rightarrow$ *bool*
where
sig-crit' bs (*Inl* (*p*, *q*)) =
(*let* (*u*, *v*) = *spair-sigs p q in*
is-syz-sig dgrad u ∨ *is-syz-sig dgrad v* ∨ *is-rewritable bs p u* ∨ *is-rewritable bs q v*) |
sig-crit' bs (*Inr j*) = *is-syz-sig dgrad* (*term-of-pair* (*0*, *j*))

fun *sig-crit-spp* :: $('t \times ('a \Rightarrow_0 'b))$ *list* ⇒ *'t list* ⇒ $((('t \times ('a \Rightarrow_0 'b)) \times ('t \times ('a \Rightarrow_0 'b))) + nat) \Rightarrow$ *bool*
where

sig-crit-spp bs ss (*Inl* (*p*, *q*)) =
(*let* (*u*, *v*) = *spair-sigs-spp p q in*
is-pred-syz ss u ∨ *is-pred-syz ss v* ∨ *is-rewritable-spp bs p u* ∨ *is-rewritable-spp bs q v*) |
sig-crit-spp bs ss (*Inr j*) = *is-pred-syz ss* (*term-of-pair* (*0*, *j*))

sig-crit is used in algorithms, *sig-crit′* is only needed for proving.

fun *rb-spp-body* ::
((('*t* × ('*a* $\Rightarrow_0$ '*b*)) *list* × '*t list* × ((('*t* × ('*a* $\Rightarrow_0$ '*b*)) × ('*t* × ('*a* $\Rightarrow_0$ '*b*))) + *nat*) *list*) × *nat*) ⇒
((('*t* × ('*a* $\Rightarrow_0$ '*b*)) *list* × '*t list* × ((('*t* × ('*a* $\Rightarrow_0$ '*b*)) × ('*t* × ('*a* $\Rightarrow_0$ '*b*))) + *nat*) *list*) × *nat*)
where
rb-spp-body ((*bs*, *ss*, []), *z*) = ((*bs*, *ss*, []), *z*) |
rb-spp-body ((*bs*, *ss*, *p* # *ps*), *z*) =
(*let ss′* = *new-syz-sigs-spp ss bs p in*
if sig-crit-spp bs ss′ p then
((*bs*, *ss′*, *ps*), *z*)
else
let p′ = *sig-trd-spp bs* (*spp-of-pair p*) *in*
if snd p′ = *0 then*
((*bs*, *fst p′* # *ss′*, *ps*), *Suc z*)
else
((*p′* # *bs*, *ss′*, *add-spairs-spp ps bs p′*), *z*))

definition *rb-spp-aux* ::
((('*t* × ('*a* $\Rightarrow_0$ '*b*)) *list* × '*t list* × ((('*t* × ('*a* $\Rightarrow_0$ '*b*)) × ('*t* × ('*a* $\Rightarrow_0$ '*b*))) + *nat*) *list*) × *nat*) ⇒
((('*t* × ('*a* $\Rightarrow_0$ '*b*)) *list* × '*t list* × ((('*t* × ('*a* $\Rightarrow_0$ '*b*)) × ('*t* × ('*a* $\Rightarrow_0$ '*b*))) + *nat*) *list*) × *nat*)
where *rb-spp-aux-def* [*code del*]: *rb-spp-aux* = *tailrec.fun* (λ*x*. *snd* (*snd* (*fst x*)) = []) (λ*x*. *x*) *rb-spp-body*

lemma *rb-spp-aux-Nil* [*code*]: *rb-spp-aux* ((*bs*, *ss*, []), *z*) = ((*bs*, *ss*, []), *z*)
by (*simp add*: *rb-spp-aux-def tailrec.simps*)

lemma *rb-spp-aux-Cons* [*code*]:
rb-spp-aux ((*bs*, *ss*, *p* # *ps*), *z*) = *rb-spp-aux* (*rb-spp-body* ((*bs*, *ss*, *p* # *ps*), *z*))
by (*simp add*: *rb-spp-aux-def tailrec.simps*)

The last parameter / return value of *rb-spp-aux*, *z*, counts the number of zero-reductions. Below we will prove that this number remains 0 under certain conditions.

context
assumes *rword-is-strict-rewrite-ord*: *is-strict-rewrite-ord rword-strict*
assumes *dgrad*: *dickson-grading dgrad*
begin

lemma *rword*: *is-rewrite-ord rword*

unfolding *rword-def* **using** *rword-is-strict-rewrite-ord* **by** (*rule is-strict-rewrite-ordD*)

lemma *sig-crit′-sym*: *sig-crit′ bs* (*Inl* (*p*, *q*)) $\Longrightarrow$ *sig-crit′ bs* (*Inl* (*q*, *p*))
by (*auto simp*: *spair-sigs-def Let-def lcs-comm*)

lemma *is-rewritable-ConsD*:
assumes *is-rewritable* (*b* # *bs*) *p u* **and** $u \prec_t$ *lt b*
shows *is-rewritable bs p u*
proof −
from *assms*(*1*) **obtain** *b′* **where** $b' \in$ *set* (*b* # *bs*) **and** $b' \neq 0$ **and** *lt b′* $adds_t$ *u*
and *rword-strict* (*spp-of p*) (*spp-of b′*) **unfolding** *is-rewritable-def* **by** *blast*
from *this*(*3*) **have** *lt b′* $\preceq_t$ *u* **by** (*rule ord-adds-term*)
with *assms*(*2*) **have** $b' \neq b$ **by** *auto*
with ‹$b' \in$ *set* (*b* # *bs*)› **have** $b' \in$ *set bs* **by** *simp*
with ‹$b' \neq 0$› ‹*lt b′* $adds_t$ *u*› ‹*rword-strict* (*spp-of p*) (*spp-of b′*)› **show** *?thesis*
by (*auto simp*: *is-rewritable-def*)
qed

lemma *sig-crit′-ConsD*:
assumes *sig-crit′* (*b* # *bs*) *p* **and** *sig-of-pair p* $\prec_t$ *lt b*
shows *sig-crit′ bs p*
proof (*rule sum-prodE*)
fix *x y*
assume *p*: *p* = *Inl* (*x*, *y*)
define *u* **where** *u* = *fst* (*spair-sigs x y*)
define *v* **where** *v* = *snd* (*spair-sigs x y*)
have *sigs*: *spair-sigs x y* = (*u*, *v*) **by** (*simp add*: *u-def v-def*)
have $u \preceq_t$ *sig-of-pair p* **and** $v \preceq_t$ *sig-of-pair p* **by** (*simp-all add*: *p sigs*)
hence $u \prec_t$ *lt b* **and** $v \prec_t$ *lt b* **using** *assms*(*2*) **by** *simp-all*
with *assms*(*1*) **show** *?thesis* **by** (*auto simp*: *p sigs dest*: *is-rewritable-ConsD*)
next
fix *j*
assume *p*: *p* = *Inr j*
from *assms* **show** *?thesis* **by** (*simp add*: *p*)
qed

definition *rb-aux-inv1* :: (′*t* $\Rightarrow_0$ ′*b*) *list* $\Rightarrow$ *bool*
where *rb-aux-inv1 bs* =
(*set bs* $\subseteq$ *dgrad-sig-set dgrad* $\wedge$ *0* $\notin$ *rep-list* ‘ *set bs* $\wedge$
sorted-wrt (λ*x y*. *lt y* $\prec_t$ *lt x*) *bs* $\wedge$
($\forall$ *i*<*length bs*. ¬ *is-sig-red* ($\prec_t$) ($\preceq$) (*set* (*drop* (*Suc i*) *bs*)) (*bs* ! *i*)) $\wedge$
($\forall$ *i*<*length bs*.
(($\exists$ *j*<*length fs*. *lt* (*bs* ! *i*) = *lt* (*monomial* (*1*::′*b*) (*term-of-pair* (*0*, *j*))) $\wedge$
punit.lt (*rep-list* (*bs* ! *i*)) $\preceq$ *punit.lt* (*rep-list* (*monomial 1* (*term-of-pair*
(*0*, *j*))))) $\vee$
($\exists$ *p*∈*set bs*. $\exists$ *q*∈*set bs*. *is-regular-spair p q* $\wedge$ *rep-list* (*spair p q*) $\neq$ *0* $\wedge$
lt (*bs* ! *i*) = *lt* (*spair p q*) $\wedge$ *punit.lt* (*rep-list* (*bs* ! *i*)) $\preceq$ *punit.lt* (*rep-list*
(*spair p q*))))) $\wedge$
($\forall$ *i*<*length bs*. *is-RB-upt dgrad rword* (*set* (*drop* (*Suc i*) *bs*)) (*lt* (*bs* !

i))))

fun *rb-aux-inv* :: (('t ⇒₀ 'b) list × 't list × ((('t ⇒₀ 'b) × ('t ⇒₀ 'b)) + nat) list) ⇒ bool
where *rb-aux-inv* (bs, ss, ps) =
(rb-aux-inv1 bs ∧
(∀ u∈set ss. is-syz-sig dgrad u) ∧
(∀ p q. Inl (p, q) ∈ set ps ⟶ (is-regular-spair p q ∧ p ∈ set bs ∧ q ∈ set bs)) ∧
(∀ j. Inr j ∈ set ps ⟶ (j < length fs ∧ (∀ b∈set bs. lt b $\prec_t$ term-of-pair (0, j))) ∧
length (filter (λq. sig-of-pair q = term-of-pair (0, j)) ps) ≤ 1) ∧
(sorted-wrt pair-ord ps) ∧
(∀ p∈set ps. (∀ b1∈set bs. ∀ b2∈set bs. is-regular-spair b1 b2 ⟶
sig-of-pair p $\prec_t$ lt (spair b1 b2) ⟶ (Inl (b1, b2) ∈ set ps ∨ Inl (b2, b1) ∈ set ps)) ∧
(∀ j<length fs. sig-of-pair p $\prec_t$ term-of-pair (0, j) ⟶ Inr j ∈ set ps)) ∧
(∀ b∈set bs. ∀ p∈set ps. lt b $\preceq_t$ sig-of-pair p) ∧
(∀ a∈set bs. ∀ b∈set bs. is-regular-spair a b ⟶ Inl (a, b) ∉ set ps ⟶ Inl (b, a) ∉ set ps ⟶
¬ is-RB-in dgrad rword (set bs) (lt (spair a b)) ⟶
(∃ p∈set ps. sig-of-pair p = lt (spair a b) ∧ ¬ sig-crit' bs p)) ∧
(∀ j<length fs. Inr j ∉ set ps ⟶ (is-RB-in dgrad rword (set bs) (term-of-pair (0, j)) ∧
rep-list (monomial (1::'b) (term-of-pair (0, j))) ∈ ideal (rep-list ‘ set bs))))

lemmas [simp del] = rb-aux-inv.simps

lemma *rb-aux-inv1-D1*: rb-aux-inv1 bs ⟹ set bs ⊆ dgrad-sig-set dgrad
by (simp add: rb-aux-inv1-def)

lemma *rb-aux-inv1-D2*: rb-aux-inv1 bs ⟹ 0 ∉ rep-list ‘ set bs
by (simp add: rb-aux-inv1-def)

lemma *rb-aux-inv1-D3*: rb-aux-inv1 bs ⟹ sorted-wrt (λx y. lt y $\prec_t$ lt x) bs
by (simp add: rb-aux-inv1-def)

lemma *rb-aux-inv1-D4*:
rb-aux-inv1 bs ⟹ i < length bs ⟹ ¬ is-sig-red ($\prec_t$) (⪯) (set (drop (Suc i) bs)) (bs ! i)
by (simp add: rb-aux-inv1-def)

lemma *rb-aux-inv1-D5*:
rb-aux-inv1 bs ⟹ i < length bs ⟹ is-RB-upt dgrad rword (set (drop (Suc i) bs)) (lt (bs ! i))
by (simp add: rb-aux-inv1-def)

lemma *rb-aux-inv1-E*:
assumes *rb-aux-inv1 bs* **and** $i <$ *length bs*
and $\bigwedge j.\ j <$ *length fs* $\Longrightarrow$ *lt* (*bs* ! *i*) = *lt* (*monomial* (*1*::'*b*) (*term-of-pair* (*0*, *j*))) $\Longrightarrow$
punit.lt (*rep-list* (*bs* ! *i*)) $\preceq$ *punit.lt* (*rep-list* (*monomial 1* (*term-of-pair* (*0*, *j*)))) $\Longrightarrow$ *thesis*
and $\bigwedge p\ q.\ p \in$ *set bs* $\Longrightarrow q \in$ *set bs* $\Longrightarrow$ *is-regular-spair p q* $\Longrightarrow$ *rep-list* (*spair p q*) $\neq 0 \Longrightarrow$
lt (*bs* ! *i*) = *lt* (*spair p q*) $\Longrightarrow$ *punit.lt* (*rep-list* (*bs* ! *i*)) $\preceq$ *punit.lt* (*rep-list* (*spair p q*)) $\Longrightarrow$ *thesis*
shows *thesis*
using *assms* **unfolding** *rb-aux-inv1-def* **by** *blast*

lemmas *rb-aux-inv1-D* = *rb-aux-inv1-D1 rb-aux-inv1-D2 rb-aux-inv1-D3 rb-aux-inv1-D4 rb-aux-inv1-D5*

lemma *rb-aux-inv1-distinct-lt*:
assumes *rb-aux-inv1 bs*
shows *distinct* (*map lt bs*)
proof (*rule distinct-sorted-wrt-irrefl*)
show *irreflp* ($\succ_t$) **by** (*simp add*: *irreflp-def*)
next
show *transp* ($\succ_t$) **by** (*auto simp*: *transp-def*)
next
from *assms* **show** *sorted-wrt* ($\succ_t$) (*map lt bs*)
unfolding *sorted-wrt-map conversep-iff* **by** (*rule rb-aux-inv1-D3*)
qed

corollary *rb-aux-inv1-lt-inj-on*:
assumes *rb-aux-inv1 bs*
shows *inj-on lt* (*set bs*)
proof
fix *a b*
assume $a \in$ *set bs*
then obtain *i* **where** *i*: $i <$ *length bs* **and** *a*: $a = bs$! *i* **by** (*metis in-set-conv-nth*)
assume $b \in$ *set bs*
then obtain *j* **where** *j*: $j <$ *length bs* **and** *b*: $b = bs$! *j* **by** (*metis in-set-conv-nth*)
assume *lt a* = *lt b*
with *i j* **have** (*map lt bs*) ! *i* = (*map lt bs*) ! *j* **by** (*simp add*: *a b*)
moreover from *assms* **have** *distinct* (*map lt bs*) **by** (*rule rb-aux-inv1-distinct-lt*)
moreover from *i* **have** $i <$ *length* (*map lt bs*) **by** *simp*
moreover from *j* **have** $j <$ *length* (*map lt bs*) **by** *simp*
ultimately have $i = j$ **by** (*simp only*: *nth-eq-iff-index-eq*)
thus $a = b$ **by** (*simp add*: *a b*)
qed

lemma *canon-rewriter-unique*:
assumes *rb-aux-inv1 bs* **and** *is-canon-rewriter rword* (*set bs*) *u a*

and *is-canon-rewriter rword* (*set bs*) *u b*
shows $a = b$
proof −
from *assms*(*1*) **have** *inj-on lt* (*set bs*) **by** (*rule rb-aux-inv1-lt-inj-on*)
moreover from *rword*(*1*) *assms*(*2*, *3*) **have** *lt a* = *lt b* **by** (*rule is-rewrite-ord-canon-rewriterD2*)
moreover from *assms*(*2*) **have** $a \in$ *set bs* **by** (*rule is-canon-rewriterD1*)
moreover from *assms*(*3*) **have** $b \in$ *set bs* **by** (*rule is-canon-rewriterD1*)
ultimately show *?thesis* **by** (*rule inj-onD*)
qed

lemma *rb-aux-inv-D1*: *rb-aux-inv* (*bs*, *ss*, *ps*) $\Longrightarrow$ *rb-aux-inv1 bs*
by (*simp add*: *rb-aux-inv.simps*)

lemma *rb-aux-inv-D2*: *rb-aux-inv* (*bs*, *ss*, *ps*) $\Longrightarrow$ $u \in$ *set ss* $\Longrightarrow$ *is-syz-sig dgrad u*
by (*simp add*: *rb-aux-inv.simps*)

lemma *rb-aux-inv-D3*:
assumes *rb-aux-inv* (*bs*, *ss*, *ps*) **and** *Inl* (*p*, *q*) $\in$ *set ps*
shows $p \in$ *set bs* **and** $q \in$ *set bs* **and** *is-regular-spair p q*
using *assms* **by** (*simp-all add*: *rb-aux-inv.simps*)

lemma *rb-aux-inv-D4*:
assumes *rb-aux-inv* (*bs*, *ss*, *ps*) **and** *Inr j* $\in$ *set ps*
shows $j <$ *length fs* **and** $\bigwedge b.\ b \in$ *set bs* $\Longrightarrow$ *lt b* $\prec_t$ *term-of-pair* (*0*, *j*)
and *length* (*filter* (λq. *sig-of-pair q* = *term-of-pair* (*0*, *j*)) *ps*) $\leq$ *1*
using *assms* **by** (*simp-all add*: *rb-aux-inv.simps*)

lemma *rb-aux-inv-D5*: *rb-aux-inv* (*bs*, *ss*, *ps*) $\Longrightarrow$ *sorted-wrt pair-ord ps*
by (*simp add*: *rb-aux-inv.simps*)

lemma *rb-aux-inv-D6-1*:
assumes *rb-aux-inv* (*bs*, *ss*, *ps*) **and** $p \in$ *set ps* **and** *b1* $\in$ *set bs* **and** *b2* $\in$ *set bs*
and *is-regular-spair b1 b2* **and** *sig-of-pair p* $\prec_t$ *lt* (*spair b1 b2*)
obtains *Inl* (*b1*, *b2*) $\in$ *set ps* | *Inl* (*b2*, *b1*) $\in$ *set ps*
using *assms* **unfolding** *rb-aux-inv.simps* **by** *blast*

lemma *rb-aux-inv-D6-2*:
rb-aux-inv (*bs*, *ss*, *ps*) $\Longrightarrow$ $p \in$ *set ps* $\Longrightarrow$ $j <$ *length fs* $\Longrightarrow$ *sig-of-pair p* $\prec_t$ *term-of-pair* (*0*, *j*) $\Longrightarrow$
Inr j $\in$ *set ps*
by (*simp add*: *rb-aux-inv.simps*)

lemma *rb-aux-inv-D7*: *rb-aux-inv* (*bs*, *ss*, *ps*) $\Longrightarrow$ $b \in$ *set bs* $\Longrightarrow$ $p \in$ *set ps* $\Longrightarrow$ *lt b* $\preceq_t$ *sig-of-pair p*
by (*simp add*: *rb-aux-inv.simps*)

lemma *rb-aux-inv-D8*:
assumes *rb-aux-inv* (*bs*, *ss*, *ps*) **and** $a \in$ *set bs* **and** $b \in$ *set bs* **and** *is-regular-spair*

a b
  and *Inl* (*a*, *b*) $\notin$ *set ps* **and** *Inl* (*b*, *a*) $\notin$ *set ps* **and** $\neg$ *is-RB-in dgrad rword* (*set bs*) (*lt* (*spair a b*))
obtains *p* **where** *p* $\in$ *set ps* **and** *sig-of-pair p* = *lt* (*spair a b*) **and** $\neg$ *sig-crit'* *bs p*
using *assms* **unfolding** *rb-aux-inv.simps* **by** *meson*

lemma *rb-aux-inv-D9*:
assumes *rb-aux-inv* (*bs*, *ss*, *ps*) **and** *j* < *length fs* **and** *Inr j* $\notin$ *set ps*
shows *is-RB-in dgrad rword* (*set bs*) (*term-of-pair* (*0*, *j*))
  and *rep-list* (*monomial* (*1*::'*b*) (*term-of-pair* (*0*, *j*))) $\in$ *ideal* (*rep-list* ‘ *set bs*)
using *assms* **by** (*simp-all add*: *rb-aux-inv.simps*)

lemma *rb-aux-inv-is-RB-upt*:
assumes *rb-aux-inv* (*bs*, *ss*, *ps*) **and** $\bigwedge$*p*. *p* $\in$ *set ps* $\Longrightarrow$ *u* $\preceq_t$ *sig-of-pair p*
shows *is-RB-upt dgrad rword* (*set bs*) *u*
proof −
from *assms*(*1*) **have** *inv1*: *rb-aux-inv1 bs* **by** (*rule rb-aux-inv-D1*)
from *dgrad rword*(*1*) **show** *?thesis*
proof (*rule is-RB-upt-finite*)
  from *inv1* **show** *set bs* $\subseteq$ *dgrad-sig-set dgrad* **by** (*rule rb-aux-inv1-D1*)
next
  from *inv1* **show** *inj-on lt* (*set bs*) **by** (*rule rb-aux-inv1-lt-inj-on*)
next
  show *finite* (*set bs*) **by** (*fact finite-set*)
next
  fix *g1 g2*
  assume *1*: *g1* $\in$ *set bs* **and** *2*: *g2* $\in$ *set bs* **and** *3*: *is-regular-spair g1 g2*
    and *4*: *lt* (*spair g1 g2*) $\prec_t$ *u*
  have *5*: *p* $\notin$ *set ps* **if** *sig-of-pair p* = *lt* (*spair g1 g2*) **for** *p*
  proof
    assume *p* $\in$ *set ps*
    hence *u* $\preceq_t$ *sig-of-pair p* **by** (*rule assms*(*2*))
    also have ... $\prec_t$ *u* **unfolding** *that* **by** (*fact 4*)
    finally show *False* **..**
  qed
  show *is-RB-in dgrad rword* (*set bs*) (*lt* (*spair g1 g2*))
  proof (*rule ccontr*)
    note *assms*(*1*) *1 2 3*
    moreover have *Inl* (*g1*, *g2*) $\notin$ *set ps* **by** (*rule 5*, *rule sig-of-spair*, *fact 3*)
    moreover have *Inl* (*g2*, *g1*) $\notin$ *set ps*
      by (*rule 5*, *simp only*: *sig-of-spair-commute*, *rule sig-of-spair*, *fact 3*)
    moreover assume $\neg$ *is-RB-in dgrad rword* (*set bs*) (*lt* (*spair g1 g2*))
    ultimately obtain *p* **where** *p* $\in$ *set ps* **and** *sig-of-pair p* = *lt* (*spair g1 g2*)
      by (*rule rb-aux-inv-D8*)
    from *this*(*2*) **have** *p* $\notin$ *set ps* **by** (*rule 5*)
    thus *False* **using** ‹*p* $\in$ *set ps*› **..**
  qed
next

fix *j*
assume *1*: *term-of-pair* (*0*, *j*) $\prec_t$ *u*
note *assms*(*1*)
moreover assume *j* < *length fs*
moreover have *Inr j* ∉ *set ps*
proof
assume *Inr j* ∈ *set ps*
hence *u* $\preceq_t$ *sig-of-pair* (*Inr j*) **by** (*rule assms*(*2*))
also have ... $\prec_t$ *u* **by** (*simp add*: *1*)
finally show *False* **..**
qed
ultimately show *is-RB-in dgrad rword* (*set bs*) (*term-of-pair* (*0*, *j*)) **by** (*rule rb-aux-inv-D9*)
qed
qed

lemma *rb-aux-inv-is-RB-upt-Cons*:
assumes *rb-aux-inv* (*bs*, *ss*, *p* # *ps*)
shows *is-RB-upt dgrad rword* (*set bs*) (*sig-of-pair p*)
using *assms*
proof (*rule rb-aux-inv-is-RB-upt*)
fix *q*
assume *q* ∈ *set* (*p* # *ps*)
hence *q* = *p* ∨ *q* ∈ *set ps* **by** *simp*
thus *sig-of-pair p* $\preceq_t$ *sig-of-pair q*
proof
assume *q* = *p*
thus *?thesis* **by** *simp*
next
assume *q* ∈ *set ps*
moreover from *assms* **have** *sorted-wrt pair-ord* (*p* # *ps*) **by** (*rule rb-aux-inv-D5*)
ultimately show *?thesis* **by** (*simp add*: *pair-ord-def*)
qed
qed

lemma *Inr-in-tailD*:
assumes *rb-aux-inv* (*bs*, *ss*, *p* # *ps*) **and** *Inr j* ∈ *set ps*
shows *sig-of-pair p* ≠ *term-of-pair* (*0*, *j*)
proof
assume *eq*: *sig-of-pair p* = *term-of-pair* (*0*, *j*)
from *assms*(*2*) **have** *Inr j* ∈ *set* (*p* # *ps*) **by** *simp*
let *?P* = λ*q*. *sig-of-pair q* = *term-of-pair* (*0*, *j*)
from *assms*(*2*) **obtain** *i1* **where** *i1* < *length ps* **and** *Inrj*: *Inr j* = *ps* ! *i1*
by (*metis in-set-conv-nth*)
from *assms*(*1*) ‹*Inr j* ∈ *set* (*p* # *ps*)› **have** *length* (*filter ?P* (*p* # *ps*)) ≤ *1*
by (*rule rb-aux-inv-D4*)
moreover from ‹*i1* < *length ps*› **have** *Suc i1* < *length* (*p* # *ps*) **by** *simp*
moreover have *0* < *length* (*p* # *ps*) **by** *simp*
moreover have *?P* ((*p* # *ps*) ! *Suc i1*) **by** (*simp add*: *Inrj*[*symmetric*])

moreover have *?P* ((*p* # *ps*) ! *0*) **by** (*simp add*: *eq*)
ultimately have *Suc i1* = *0* **by** (*rule length-filter-le-1*)
thus *False* **..**
qed

lemma *pair-list-aux*:
assumes *rb-aux-inv* (*bs*, *ss*, *ps*) **and** *p* ∈ *set ps*
shows *sig-of-pair p* = *lt* (*poly-of-pair p*) ∧ *poly-of-pair p* ≠ *0* ∧ *poly-of-pair p* ∈ *dgrad-sig-set dgrad*
proof (*rule sum-prodE*)
fix *a b*
assume *p*: *p* = *Inl* (*a*, *b*)
from *assms*(*1*) **have** *rb-aux-inv1 bs* **by** (*rule rb-aux-inv-D1*)
hence *bs-sub*: *set bs* ⊆ *dgrad-sig-set dgrad* **by** (*rule rb-aux-inv1-D1*)
from *assms* **have** *is-regular-spair a b* **unfolding** *p* **by** (*rule rb-aux-inv-D3*)
hence *sig-of-pair p* = *lt* (*poly-of-pair p*) **and** *poly-of-pair p* ≠ *0*
unfolding *p poly-of-pair.simps* **by** (*rule sig-of-spair*, *rule is-regular-spair-nonzero*)
moreover from *dgrad* **have** *poly-of-pair p* ∈ *dgrad-sig-set dgrad* **unfolding** *p poly-of-pair.simps*
proof (*rule dgrad-sig-set-closed-spair*)
from *assms* **have** *a* ∈ *set bs* **unfolding** *p* **by** (*rule rb-aux-inv-D3*)
thus *a* ∈ *dgrad-sig-set dgrad* **using** *bs-sub* **..**
next
from *assms* **have** *b* ∈ *set bs* **unfolding** *p* **by** (*rule rb-aux-inv-D3*)
thus *b* ∈ *dgrad-sig-set dgrad* **using** *bs-sub* **..**
qed
ultimately show *?thesis* **by** *simp*
next
fix *j*
assume *p* = *Inr j*
from *assms* **have** *j* < *length fs* **unfolding** ‹*p* = *Inr j*› **by** (*rule rb-aux-inv-D4*)
have *monomial 1* (*term-of-pair* (*0*, *j*)) ∈ *dgrad-sig-set dgrad*
by (*rule dgrad-sig-set-closed-monomial*, *simp add*: *pp-of-term-of-pair dgrad-max-0*,
simp add: *component-of-term-of-pair* ‹*j* < *length fs*›)
thus *?thesis* **by** (*simp add*: ‹*p* = *Inr j*› *lt-monomial monomial-0-iff*)
qed

corollary *pair-list-sig-of-pair*:
rb-aux-inv (*bs*, *ss*, *ps*) ⟹ *p* ∈ *set ps* ⟹ *sig-of-pair p* = *lt* (*poly-of-pair p*)
by (*simp add*: *pair-list-aux*)

corollary *pair-list-nonzero*: *rb-aux-inv* (*bs*, *ss*, *ps*) ⟹ *p* ∈ *set ps* ⟹ *poly-of-pair p* ≠ *0*
by (*simp add*: *pair-list-aux*)

corollary *pair-list-dgrad-sig-set*:
rb-aux-inv (*bs*, *ss*, *ps*) ⟹ *p* ∈ *set ps* ⟹ *poly-of-pair p* ∈ *dgrad-sig-set dgrad*
by (*simp add*: *pair-list-aux*)

lemma *is-rewritableI-is-canon-rewriter*:
assumes *rb-aux-inv1 bs* **and** $b \in$ *set bs* **and** $b \neq 0$ **and** *lt b adds*$_t$ *u*
and $\neg$ *is-canon-rewriter rword* (*set bs*) *u b*
shows *is-rewritable bs b u*
proof −
from *assms*(*2*−*5*) **obtain** b' **where** $b' \in$ *set bs* **and** $b' \neq 0$ **and** *lt* b' *adds*$_t$ *u*
and *1*: $\neg$ *rword* (*spp-of* b') (*spp-of b*) **by** (*auto simp*: *is-canon-rewriter-def*)
show *?thesis* **unfolding** *is-rewritable-def*
proof (*intro bexI conjI*)
from *rword*(*1*) **have** *2*: *rword* (*spp-of b*) (*spp-of* b')
proof (*rule is-rewrite-ordD3*)
assume *rword* (*spp-of* b') (*spp-of b*)
with *1* **show** *?thesis* ..
qed
from *rword*(*1*) *1* **have** $b \neq b'$ **by** (*auto dest*: *is-rewrite-ordD1*)
have *lt* $b \neq$ *lt* b'
proof
assume *lt* $b =$ *lt* b'
with *rb-aux-inv1-lt-inj-on*[*OF assms*(*1*)] **have** $b = b'$ **using** *assms*(*2*) ‹$b' \in$ *set bs*›
by (*rule inj-onD*)
with ‹$b \neq b'$› **show** *False* ..
qed
hence *fst* (*spp-of b*) $\neq$ *fst* (*spp-of* b') **by** (*simp add*: *spp-of-def*)
with *rword-is-strict-rewrite-ord 2* **show** *rword-strict* (*spp-of b*) (*spp-of* b')
by (*auto simp*: *rword-def dest*: *is-strict-rewrite-ord-antisym*)
qed *fact*+
qed

lemma *is-rewritableD-is-canon-rewriter*:
assumes *rb-aux-inv1 bs* **and** *is-rewritable bs b u*
shows $\neg$ *is-canon-rewriter rword* (*set bs*) *u b*
proof
assume *is-canon-rewriter rword* (*set bs*) *u b*
hence $b \in$ *set bs* **and** $b \neq 0$ **and** *lt b adds*$_t$ *u*
and *1*: $\bigwedge a.\ a \in$ *set bs* $\Longrightarrow a \neq 0 \Longrightarrow$ *lt a adds*$_t$ *u* $\Longrightarrow$ *rword* (*spp-of a*) (*spp-of b*)
by (*rule is-canon-rewriterD*)+
from *assms*(*2*) **obtain** b' **where** $b' \in$ *set bs* **and** $b' \neq 0$ **and** *lt* b' *adds*$_t$ *u*
and *2*: *rword-strict* (*spp-of b*) (*spp-of* b') **unfolding** *is-rewritable-def* **by** *blast*
from *this*(*1*, *2*, *3*) **have** *rword* (*spp-of* b') (*spp-of b*) **by** (*rule 1*)
moreover from *rword-is-strict-rewrite-ord 2* **have** *rword* (*spp-of b*) (*spp-of* b')
unfolding *rword-def* **by** (*rule is-strict-rewrite-ord-asym*)
ultimately have *fst* (*spp-of* b') = *fst* (*spp-of b*) **by** (*rule is-rewrite-ordD4*[*OF rword*])
hence *lt* $b' =$ *lt b* **by** (*simp add*: *spp-of-def*)
with *rb-aux-inv1-lt-inj-on*[*OF assms*(*1*)] **have** $b' = b$ **using** ‹$b' \in$ *set bs*› ‹$b \in$ *set bs*›
by (*rule inj-onD*)

from *rword-is-strict-rewrite-ord* **have** ¬ *rword-strict* (*spp-of b*) (*spp-of b′*)
unfolding ‹$b' = b$› **by** (*rule is-strict-rewrite-ord-irrefl*)
thus *False* **using** *2* **..**
qed

lemma *lemma-12*:
assumes *rb-aux-inv* (*bs*, *ss*, *ps*) **and** *is-RB-upt dgrad rword* (*set bs*) *u*
and *dgrad* (*pp-of-term u*) ≤ *dgrad-max dgrad* **and** *is-canon-rewriter rword* (*set bs*) *u a*
and ¬ *is-syz-sig dgrad u* **and** *is-sig-red* ($\prec_t$) (=) (*set bs*) (*monom-mult 1* (*pp-of-term u* − *lp a*) *a*)
obtains *p q* **where** *p* ∈ *set bs* **and** *q* ∈ *set bs* **and** *is-regular-spair p q* **and** *lt* (*spair p q*) = *u*
and ¬ *sig-crit′ bs* (*Inl* (*p*, *q*))
proof −
from *assms*(*1*) **have** *inv1*: *rb-aux-inv1 bs* **by** (*rule rb-aux-inv-D1*)
hence *inj*: *inj-on lt* (*set bs*) **by** (*rule rb-aux-inv1-lt-inj-on*)
from *assms*(*4*) **have** *lt a* $adds_t$ *u* **by** (*rule is-canon-rewriterD3*)
hence *lp a adds pp-of-term u* **and** *comp-a*: *component-of-term* (*lt a*) = *component-of-term u*
by (*simp-all add*: *adds-term-def*)
let *?s* = *pp-of-term u* − *lp a*
let *?a* = *monom-mult 1 ?s a*
from *assms*(*4*) **have** *a* ∈ *set bs* **by** (*rule is-canon-rewriterD1*)
from *assms*(*6*) **have** *rep-list ?a* ≠ *0* **using** *is-sig-red-top-addsE* **by** *blast*
hence *rep-list a* ≠ *0* **by** (*auto simp*: *rep-list-monom-mult*)
hence *a* ≠ *0* **by** (*auto simp*: *rep-list-zero*)
hence *lt ?a* = *?s* ⊕ *lt a* **by** (*simp add*: *lt-monom-mult*)
also from ‹*lp a adds pp-of-term u*› **have** *eq0*: ... = *u*
by (*simp add*: *splus-def comp-a adds-minus term-simps*)
finally have *lt ?a* = *u* **.**
note *dgrad rword*(*1*)
moreover from *assms*(*2*) **have** *is-RB-upt dgrad rword* (*set bs*) (*lt ?a*) **by** (*simp only*: ‹*lt ?a* = *u*›)
moreover from *dgrad* **have** *?a* ∈ *dgrad-sig-set dgrad*
proof (*rule dgrad-sig-set-closed-monom-mult*)
from *dgrad* ‹*lp a adds pp-of-term u*› **have** *dgrad* (*pp-of-term u* − *lp a*) ≤ *dgrad* (*pp-of-term u*)
by (*rule dickson-grading-minus*)
thus *dgrad* (*pp-of-term u* − *lp a*) ≤ *dgrad-max dgrad* **using** *assms*(*3*) **by** (*rule le-trans*)
next
from *inv1* **have** *set bs* ⊆ *dgrad-sig-set dgrad* **by** (*rule rb-aux-inv1-D1*)
with ‹*a* ∈ *set bs*› **show** *a* ∈ *dgrad-sig-set dgrad* **..**
qed
ultimately obtain *v b* **where** *v* $\prec_t$ *lt ?a* **and** *dgrad* (*pp-of-term v*) ≤ *dgrad-max dgrad*
and *component-of-term v* < *length fs* **and** *ns*: ¬ *is-syz-sig dgrad v*
and *v*: *v* = (*punit.lt* (*rep-list ?a*) − *punit.lt* (*rep-list b*)) ⊕ *lt b*

and *cr*: *is-canon-rewriter rword* (*set bs*) *v b* **and** *is-sig-red* ($\prec_t$) (=) {*b*} *?a*
using *assms*(*6*) **by** (*rule lemma-11*)
from *this*(*6*) **have** $b \in$ *set bs* **by** (*rule is-canon-rewriterD1*)
with ‹$a \in$ *set bs*› **show** *?thesis*
proof
from *dgrad rword*(*1*) *assms*(*2*) *inj assms*(*5*, *4*) ‹$b \in$ *set bs*› ‹*is-sig-red* ($\prec_t$) (=) {*b*} *?a*› *assms*(*3*)
show *is-regular-spair a b* **by** (*rule lemma-9*(*3*))
next
from *dgrad rword*(*1*) *assms*(*2*) *inj assms*(*5*, *4*) ‹$b \in$ *set bs*› ‹*is-sig-red* ($\prec_t$) (=) {*b*} *?a*› *assms*(*3*)
show *lt* (*spair a b*) = *u* **by** (*rule lemma-9*(*4*))
next
from ‹*rep-list a* $\neq$ *0*› **have** *v'*: *v* = (*?s* + *punit.lt* (*rep-list a*) − *punit.lt* (*rep-list b*)) $\oplus$ *lt b*
by (*simp add*: *v rep-list-monom-mult punit.lt-monom-mult*)
moreover from *dgrad rword*(*1*) *assms*(*2*) *inj assms*(*5*, *4*) ‹$b \in$ *set bs*› ‹*is-sig-red* ($\prec_t$) (=) {*b*} *?a*› *assms*(*3*)
have *lcs* (*punit.lt* (*rep-list a*)) (*punit.lt* (*rep-list b*)) − *punit.lt* (*rep-list a*) = *?s*
and *lcs* (*punit.lt* (*rep-list a*)) (*punit.lt* (*rep-list b*)) − *punit.lt* (*rep-list b*) =
?s + *punit.lt* (*rep-list a*) − *punit.lt* (*rep-list b*)
by (*rule lemma-9*)+
ultimately have *eq1*: *spair-sigs a b* = (*u*, *v*) **by** (*simp add*: *spair-sigs-def eq0*)
show ¬ *sig-crit'* *bs* (*Inl* (*a*, *b*))
proof (*simp add*: *eq1 assms*(*5*) *ns*, *intro conjI notI*)
assume *is-rewritable bs a u*
with *inv1* **have** ¬ *is-canon-rewriter rword* (*set bs*) *u a* **by** (*rule is-rewritableD-is-canon-rewriter*)
thus *False* **using** *assms*(*4*) **..**
next
assume *is-rewritable bs b v*
with *inv1* **have** ¬ *is-canon-rewriter rword* (*set bs*) *v b* **by** (*rule is-rewritableD-is-canon-rewriter*)
thus *False* **using** *cr* **..**
qed
qed
qed

lemma *is-canon-rewriterI-eq-sig*:
assumes *rb-aux-inv1 bs* **and** $b \in$ *set bs*
shows *is-canon-rewriter rword* (*set bs*) (*lt b*) *b*
proof −
from *assms*(*2*) **have** *rep-list b* $\in$ *rep-list* ‘ *set bs* **by** (*rule imageI*)
moreover from *assms*(*1*) **have** *0* $\notin$ *rep-list* ‘ *set bs* **by** (*rule rb-aux-inv1-D2*)
ultimately have $b \neq$ *0* **by** (*auto simp*: *rep-list-zero*)
with *assms*(*2*) **show** *?thesis*
proof (*rule is-canon-rewriterI*)
fix *a*
assume $a \in$ *set bs* **and** $a \neq$ *0* **and** *lt a* $adds_t$ *lt b*
from *assms*(*2*) **obtain** *i* **where** *i* < *length bs* **and** *b*: *b* = *bs* ! *i* **by** (*metis in-set-conv-nth*)

from *assms*(*1*) *this*(*1*) **have** *is-RB-upt dgrad rword* (*set* (*drop* (*Suc i*) *bs*)) (*lt* (*bs* ! *i*))
by (*rule rb-aux-inv1-D5*)
with *dgrad* **have** *is-sig-GB-upt dgrad* (*set* (*drop* (*Suc i*) *bs*)) (*lt* (*bs* ! *i*))
by (*rule is-RB-upt-is-sig-GB-upt*)
hence *is-sig-GB-upt dgrad* (*set* (*drop* (*Suc i*) *bs*)) (*lt b*) **by** (*simp only*: *b*)
moreover have *set* (*drop* (*Suc i*) *bs*) $\subseteq$ *set bs* **by** (*rule set-drop-subset*)
moreover from *assms*(*1*) **have** *set bs* $\subseteq$ *dgrad-sig-set dgrad* **by** (*rule rb-aux-inv1-D1*)
ultimately have *is-sig-GB-upt dgrad* (*set bs*) (*lt b*) **by** (*rule is-sig-GB-upt-mono*)
with *rword*(*1*) *dgrad* **show** *rword* (*spp-of a*) (*spp-of b*)
proof (*rule is-rewrite-ordD5*)
from *assms*(*1*) ‹$i <$ *length bs*› **have** ¬ *is-sig-red* ($\prec_t$) ($\preceq$) (*set* (*drop* (*Suc i*) *bs*)) (*bs* ! *i*)
by (*rule rb-aux-inv1-D4*)
hence ¬ *is-sig-red* ($\prec_t$) (=) (*set* (*drop* (*Suc i*) *bs*)) *b* **by** (*simp add*: *b is-sig-red-top-tail-cases*)
moreover have ¬ *is-sig-red* ($\prec_t$) (=) (*set* (*take* (*Suc i*) *bs*)) *b*
proof
assume *is-sig-red* ($\prec_t$) (=) (*set* (*take* (*Suc i*) *bs*)) *b*
then obtain *f* **where** *f-in*: $f \in$ *set* (*take* (*Suc i*) *bs*) **and** *is-sig-red* ($\prec_t$) (=) {*f*} *b*
by (*rule is-sig-red-singletonI*)
from *this*(*2*) **have** *lt f* $\prec_t$ *lt b* **by** (*rule is-sig-red-regularD-lt*)
from ‹$i <$ *length bs*› **have** *take-eq*: *take* (*Suc i*) *bs* = (*take i bs*) @ [*b*]
unfolding *b* **by** (*rule take-Suc-conv-app-nth*)
from *assms*(*1*) **have** *sorted-wrt* ($\lambda x\ y.$ *lt y* $\prec_t$ *lt x*) ((*take* (*Suc i*) *bs*) @ (*drop* (*Suc i*) *bs*))
unfolding *append-take-drop-id* **by** (*rule rb-aux-inv1-D3*)
hence *1*: $\bigwedge y.\ y \in$ *set* (*take i bs*) $\Longrightarrow$ *lt b* $\prec_t$ *lt y*
by (*simp add*: *sorted-wrt-append take-eq del*: *append-take-drop-id*)
from *f-in* **have** $f = b \vee f \in$ *set* (*take i bs*) **by** (*simp add*: *take-eq*)
hence *lt b* $\preceq_t$ *lt f*
proof
assume $f \in$ *set* (*take i bs*)
hence *lt b* $\prec_t$ *lt f* **by** (*rule 1*)
thus *?thesis* **by** *simp*
qed *simp*
with ‹*lt f* $\prec_t$ *lt b*› **show** *False* **by** *simp*
qed
ultimately have ¬ *is-sig-red* ($\prec_t$) (=) (*set* (*take* (*Suc i*) *bs*) $\cup$ *set* (*drop* (*Suc i*) *bs*)) *b*
by (*simp add*: *is-sig-red-Un*)
thus ¬ *is-sig-red* ($\prec_t$) (=) (*set bs*) *b* **by** (*metis append-take-drop-id set-append*)
qed *fact+*
qed (*simp add*: *term-simps*)
qed

lemma *not-sig-crit*:
assumes *rb-aux-inv* (*bs*, *ss*, *p* # *ps*) **and** ¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p*

and $b \in$ *set bs*
shows *lt b* $\prec_t$ *sig-of-pair p*
proof (*rule sum-prodE*)
fix *x y*
assume *p*: $p = Inl\ (x,\ y)$
have $p \in set\ (p\ \#\ ps)$ **by** *simp*
hence $Inl\ (x,\ y) \in set\ (p\ \#\ ps)$ **by** (*simp only*: *p*)
define *t1* **where** *t1* = *punit.lt* (*rep-list x*)
define *t2* **where** *t2* = *punit.lt* (*rep-list y*)
define *u* **where** *u* = *fst* (*spair-sigs x y*)
define *v* **where** *v* = *snd* (*spair-sigs x y*)
have *u*: $u = (lcs\ t1\ t2 - t1) \oplus lt\ x$ **by** (*simp add*: *u-def spair-sigs-def t1-def t2-def Let-def*)
have *v*: $v = (lcs\ t1\ t2 - t2) \oplus lt\ y$ **by** (*simp add*: *v-def spair-sigs-def t1-def t2-def Let-def*)
have *spair-sigs*: *spair-sigs x y* = (*u*, *v*) **by** (*simp add*: *u-def v-def*)
with *assms*(*2*) **have** ¬ *is-rewritable bs x u* **and** ¬ *is-rewritable bs y v*
by (*simp-all add*: *p*)
from *assms*(*1*) ‹$Inl\ (x,\ y) \in set\ (p\ \#\ ps)$› **have** *x-in*: $x \in set\ bs$ **and** *y-in*: $y \in set\ bs$
and *is-regular-spair x y* **by** (*rule rb-aux-inv-D3*)+
from *assms*(*1*) **have** *inv1*: *rb-aux-inv1 bs* **by** (*rule rb-aux-inv-D1*)
from *inv1* **have** $0 \notin$ *rep-list* ‘ *set bs* **by** (*rule rb-aux-inv1-D2*)
with *x-in y-in* **have** *rep-list x* $\neq 0$ **and** *rep-list y* $\neq 0$ **by** *auto*
hence $x \neq 0$ **and** $y \neq 0$ **by** (*auto simp*: *rep-list-zero*)
from *inv1* **have** *sorted*: *sorted-wrt* ($\lambda x\ y.\ lt\ y \prec_t lt\ x$) *bs* **by** (*rule rb-aux-inv1-D3*)
from *x-in* **obtain** *i1* **where** $i1 < length\ bs$ **and** *x*: $x = bs\ !\ i1$ **by** (*metis in-set-conv-nth*)
from *y-in* **obtain** *i2* **where** $i2 < length\ bs$ **and** *y*: $y = bs\ !\ i2$ **by** (*metis in-set-conv-nth*)
have *lt b* $\neq$ *sig-of-pair p*
proof
assume *lt-b*: *lt b* = *sig-of-pair p*
from *inv1* **have** *crw*: *is-canon-rewriter rword* (*set bs*) (*lt b*) *b* **using** *assms*(*3*)
by (*rule is-canon-rewriterI-eq-sig*)
show *False*
proof (*rule ord-term-lin.linorder-cases*)
assume $u \prec_t v$
hence *lt b* = *v* **by** (*auto simp*: *lt-b p spair-sigs ord-term-lin.max-def*)
with *crw* **have** *crw-b*: *is-canon-rewriter rword* (*set bs*) *v b* **by** *simp*
from *v* **have** *lt y* $adds_t$ *v* **by** (*rule adds-termI*)
hence *is-canon-rewriter rword* (*set bs*) *v y*
using *inv1 y-in* ‹$y \neq 0$› ‹¬ *is-rewritable bs y v*› *is-rewritableI-is-canon-rewriter* **by** *blast*
with *inv1 crw-b* **have** *b* = *y* **by** (*rule canon-rewriter-unique*)
with ‹*lt b* = *v*› **have** *lt y* = *v* **by** *simp*
from *inv1* ‹$i2 < length\ bs$› **have** ¬ *is-sig-red* ($\prec_t$) ($\preceq$) (*set* (*drop* (*Suc i2*) *bs*)) (*bs* ! *i2*)
by (*rule rb-aux-inv1-D4*)

moreover have *is-sig-red* $(\prec_t)$ $(\preceq)$ (*set* (*drop* (*Suc i2*) *bs*)) (*bs* ! *i2*)
proof (*rule is-sig-red-singletonD*)
have *is-sig-red* $(\prec_t)$ $(=)$ $\{x\}$ *y*
proof (*rule is-sig-red-top-addsI*)
from ‹*lt* $y = v$› **have** (*lcs t1 t2* $-$ *t2*) $\oplus$ *lt y* $=$ *lt y* **by** (*simp only*: *v*)
also have ... $=$ *0* $\oplus$ *lt y* **by** (*simp only*: *term-simps*)
finally have *lcs t1 t2* $-$ *t2* $=$ *0* **by** (*simp only*: *splus-right-canc*)
hence *lcs t1 t2* $=$ *t2* **by** (*metis* (*full-types*) *add.left-neutral adds-minus adds-lcs-2*)
with *adds-lcs*[*of t1 t2*] **show** *punit.lt* (*rep-list x*) *adds punit.lt* (*rep-list y*)
by (*simp only*: *t1-def t2-def*)
next
from ‹$u \prec_t v$› **show** *punit.lt* (*rep-list y*) $\oplus$ *lt x* $\prec_t$ *punit.lt* (*rep-list x*) $\oplus$ *lt y*
by (*simp add*: *t1-def t2-def u v term-is-le-rel-minus-minus adds-lcs adds-lcs-2*)
qed (*simp*|*fact*)+
thus *is-sig-red* $(\prec_t)$ $(\preceq)$ $\{x\}$ (*bs* ! *i2*) **by** (*simp add*: *y is-sig-red-top-tail-cases*)
next
have *lt x* $\preceq_t$ *0* $\oplus$ *lt x* **by** (*simp only*: *term-simps*)
also have ... $\preceq_t$ *u* **unfolding** *u* **using** *zero-min* **by** (*rule splus-mono-left*)
also have ... $\prec_t$ *v* **by** *fact*
finally have $*$: *lt* (*bs* ! *i1*) $\prec_t$ *lt* (*bs* ! *i2*) **by** (*simp only*: ‹*lt* $y = v$› *x y*[*symmetric*])
have *i2* $<$ *i1*
proof (*rule linorder-cases*)
assume *i1* $<$ *i2*
with *sorted* **have** *lt* (*bs* ! *i2*) $\prec_t$ *lt* (*bs* ! *i1*) **using** ‹*i2* $<$ *length bs*›
by (*rule sorted-wrt-nth-less*)
with $*$ **show** *?thesis* **by** *simp*
next
assume *i1* $=$ *i2*
with $*$ **show** *?thesis* **by** *simp*
qed
hence *Suc i2* $\leq$ *i1* **by** *simp*
thus $x \in$ *set* (*drop* (*Suc i2*) *bs*) **unfolding** *x* **using** ‹*i1* $<$ *length bs*› **by** (*rule nth-in-set-dropI*)
qed
ultimately show *?thesis* **..**
next
assume $v \prec_t u$
hence *lt b* $=$ *u* **by** (*auto simp*: *lt-b p spair-sigs ord-term-lin.max-def*)
with *crw* **have** *crw-b*: *is-canon-rewriter rword* (*set bs*) *u b* **by** *simp*
from *u* **have** *lt x* $adds_t$ *u* **by** (*rule adds-termI*)
hence *is-canon-rewriter rword* (*set bs*) *u x*
using *inv1 x-in* ‹$x \neq 0$› ‹$\neg$ *is-rewritable bs x u*› *is-rewritableI-is-canon-rewriter* **by** *blast*
with *inv1 crw-b* **have** $b = x$ **by** (*rule canon-rewriter-unique*)
with ‹*lt* $b = u$› **have** *lt* $x = u$ **by** *simp*

from *inv1* ‹$i1 < length\ bs$› **have** ¬ *is-sig-red* ($\prec_t$) ($\preceq$) (*set* (*drop* (*Suc i1*) *bs*)) (*bs* ! *i1*)
by (*rule rb-aux-inv1-D4*)
moreover have *is-sig-red* ($\prec_t$) ($\preceq$) (*set* (*drop* (*Suc i1*) *bs*)) (*bs* ! *i1*)
proof (*rule is-sig-red-singletonD*)
have *is-sig-red* ($\prec_t$) (=) {*y*} *x*
proof (*rule is-sig-red-top-addsI*)
from ‹$lt\ x = u$› **have** ($lcs\ t1\ t2 - t1$) ⊕ $lt\ x = lt\ x$ **by** (*simp only*: *u*)
also have ... = *0* ⊕ *lt x* **by** (*simp only*: *term-simps*)
finally have $lcs\ t1\ t2 - t1 = 0$ **by** (*simp only*: *splus-right-canc*)
hence $lcs\ t1\ t2 = t1$ **by** (*metis* (*full-types*) *add.left-neutral adds-minus adds-lcs*)
with *adds-lcs-2*[*of t2 t1*] **show** *punit.lt* (*rep-list y*) *adds punit.lt* (*rep-list x*)
by (*simp only*: *t1-def t2-def*)
next
from ‹$v \prec_t u$› **show** *punit.lt* (*rep-list x*) ⊕ *lt y* $\prec_t$ *punit.lt* (*rep-list y*) ⊕ *lt x*
by (*simp add*: *t1-def t2-def u v term-is-le-rel-minus-minus adds-lcs adds-lcs-2*)
qed (*simp*|*fact*)+
thus *is-sig-red* ($\prec_t$) ($\preceq$) {*y*} (*bs* ! *i1*) **by** (*simp add*: *x is-sig-red-top-tail-cases*)
next
have *lt y* $\preceq_t$ *0* ⊕ *lt y* **by** (*simp only*: *term-simps*)
also have ... $\preceq_t$ *v* **unfolding** *v* **using** *zero-min* **by** (*rule splus-mono-left*)
also have ... $\prec_t$ *u* **by** *fact*
finally have ∗: *lt* (*bs* ! *i2*) $\prec_t$ *lt* (*bs* ! *i1*) **by** (*simp only*: ‹$lt\ x = u$› *y x*[*symmetric*])
have $i1 < i2$
proof (*rule linorder-cases*)
assume $i2 < i1$
with *sorted* **have** *lt* (*bs* ! *i1*) $\prec_t$ *lt* (*bs* ! *i2*) **using** ‹$i1 < length\ bs$›
by (*rule sorted-wrt-nth-less*)
with ∗ **show** *?thesis* **by** *simp*
next
assume $i2 = i1$
with ∗ **show** *?thesis* **by** *simp*
qed
hence $Suc\ i1 \le i2$ **by** *simp*
thus $y \in set$ (*drop* (*Suc i1*) *bs*) **unfolding** *y* **using** ‹$i2 < length\ bs$› **by** (*rule nth-in-set-dropI*)
qed
ultimately show *?thesis* **..**
next
assume $u = v$
hence *punit.lt* (*rep-list x*) ⊕ *lt y* = *punit.lt* (*rep-list y*) ⊕ *lt x*
by (*simp add*: *t1-def t2-def u v term-is-le-rel-minus-minus adds-lcs adds-lcs-2*)
moreover from ‹*is-regular-spair x y*›
have *punit.lt* (*rep-list y*) ⊕ *lt x* ≠ *punit.lt* (*rep-list x*) ⊕ *lt y* **by** (*rule*

is-regular-spairD3)
      ultimately show *?thesis* **by** *simp*
    qed
  qed
  moreover from *assms*(*1*, *3*) ‹$p \in set\ (p\ \#\ ps)$› **have** $lt\ b \preceq_t sig\text{-}of\text{-}pair\ p$ **by** (*rule rb-aux-inv-D7*)
  ultimately show *?thesis* **by** *simp*
next
  fix *j*
  assume *p*: $p = Inr\ j$
  have $Inr\ j \in set\ (p\ \#\ ps)$ **by** (*simp add*: *p*)
  with *assms*(*1*) **have** $lt\ b \prec_t term\text{-}of\text{-}pair\ (0, j)$ **using** *assms*(*3*) **by** (*rule rb-aux-inv-D4*)
  thus *?thesis* **by** (*simp add*: *p*)
qed

context
  assumes *fs-distinct*: *distinct fs*
  assumes *fs-nonzero*: $0 \notin set\ fs$
begin

lemma *rep-list-monomial'*: $rep\text{-}list\ (monomial\ 1\ (term\text{-}of\text{-}pair\ (0, j))) = ((fs\ !\ j)$ *when* $j < length\ fs)$
  by (*simp add*: *rep-list-monomial fs-distinct term-simps*)

lemma *new-syz-sigs-is-syz-sig*:
  assumes *rb-aux-inv* (*bs*, *ss*, *p* # *ps*) **and** $v \in set\ (new\text{-}syz\text{-}sigs\ ss\ bs\ p)$
  shows *is-syz-sig dgrad v*
proof (*rule sum-prodE*)
  fix *a b*
  assume $p = Inl\ (a, b)$
  with *assms*(*2*) **have** $v \in set\ ss$ **by** *simp*
  with *assms*(*1*) **show** *?thesis* **by** (*rule rb-aux-inv-D2*)
next
  fix *j*
  assume *p*: $p = Inr\ j$
  let $?f = \lambda b.\ term\text{-}of\text{-}pair\ (punit.lt\ (rep\text{-}list\ b), j)$
  let $?a = monomial\ (1::'b)\ (term\text{-}of\text{-}pair\ (0, j))$
  from *assms*(*1*) **have** *inv1*: *rb-aux-inv1 bs* **by** (*rule rb-aux-inv-D1*)
  have $Inr\ j \in set\ (p\ \#\ ps)$ **by** (*simp add*: *p*)
  with *assms*(*1*) **have** $j < length\ fs$ **by** (*rule rb-aux-inv-D4*)
  hence *a*: $rep\text{-}list\ ?a = fs\ !\ j$ **by** (*simp add*: *rep-list-monomial'*)
  show *?thesis*
  proof (*cases is-pot-ord*)
    case *True*
    with *assms*(*2*) **have** $v \in set\ (filter\text{-}min\text{-}append\ (adds_t)\ ss\ (filter\text{-}min\ (adds_t)\ (map\ ?f\ bs)))$
      by (*simp add*: *p*)
    hence $v \in set\ ss \cup ?f\ `\ set\ bs$ **using** *filter-min-append-subset filter-min-subset* **by** *fastforce*

thus *?thesis*
proof
assume $v \in$ *set ss*
with *assms*(*1*) **show** *?thesis* **by** (*rule rb-aux-inv-D2*)
next
assume $v \in$ *?f ‘ set bs*
then obtain *b* **where** $b \in$ *set bs* **and** $v =$ *?f b* **..**
have *comp-b*: *component-of-term* (*lt b*) < *component-of-term* (*lt ?a*)
proof (*rule ccontr*)
have $*$: *pp-of-term* (*term-of-pair* (*0*, *j*)) $\preceq$ *pp-of-term* (*lt b*)
by (*simp add*: *pp-of-term-of-pair zero-min*)
assume ¬ *component-of-term* (*lt b*) < *component-of-term* (*lt ?a*)
hence *component-of-term* (*term-of-pair* (*0*, *j*)) ≤ *component-of-term* (*lt b*)
by (*simp add*: *lt-monomial*)
with $*$ **have** *term-of-pair* (*0*, *j*) $\preceq_t$ *lt b* **by** (*rule ord-termI*)
moreover from *assms*(*1*) ‹*Inr j* ∈ *set* (*p* # *ps*)› ‹*b* ∈ *set bs*› **have** *lt b* $\prec_t$ *term-of-pair* (*0*, *j*)
by (*rule rb-aux-inv-D4*)
ultimately show *False* **by** *simp*
qed
have $v =$ *punit.lt* (*rep-list b*) ⊕ *lt ?a*
by (*simp add*: ‹*v* = *?f b*› *a lt-monomial splus-def term-simps*)
also have ... = *ord-term-lin.max* (*punit.lt* (*rep-list b*) ⊕ *lt ?a*) (*punit.lt* (*rep-list ?a*) ⊕ *lt b*)
proof −
have *component-of-term* (*punit.lt* (*rep-list ?a*) ⊕ *lt b*) = *component-of-term* (*lt b*)
by (*simp only*: *term-simps*)
also have ... < *component-of-term* (*lt ?a*) **by** (*fact comp-b*)
also have ... = *component-of-term* (*punit.lt* (*rep-list b*) ⊕ *lt ?a*)
by (*simp only*: *term-simps*)
finally have *component-of-term* (*punit.lt* (*rep-list ?a*) ⊕ *lt b*) <
component-of-term (*punit.lt* (*rep-list b*) ⊕ *lt ?a*) **.**
with *True* **have** *punit.lt* (*rep-list ?a*) ⊕ *lt b* $\prec_t$ *punit.lt* (*rep-list b*) ⊕ *lt ?a*
by (*rule is-pot-ordD*)
thus *?thesis* **by** (*auto simp*: *ord-term-lin.max-def*)
qed
finally have *v*: $v =$ *ord-term-lin.max* (*punit.lt* (*rep-list b*) ⊕ *lt ?a*) (*punit.lt* (*rep-list ?a*) ⊕ *lt b*) **.**
show *?thesis* **unfolding** *v* **using** *dgrad*
proof (*rule Koszul-syz-is-syz-sig*)
from *inv1* **have** *set bs* ⊆ *dgrad-sig-set dgrad* **by** (*rule rb-aux-inv1-D1*)
with ‹*b* ∈ *set bs*› **show** *b* ∈ *dgrad-sig-set dgrad* **..**
next
show *?a* ∈ *dgrad-sig-set dgrad*
by (*rule dgrad-sig-set-closed-monomial*, *simp-all add*: *term-simps dgrad-max-0* ‹*j* < *length fs*›)
next
from *inv1* **have** *0* ∉ *rep-list ‘ set bs* **by** (*rule rb-aux-inv1-D2*)

with ‹$b \in set\ bs$› **show** $rep\text{-}list\ b \neq 0$ **by** *fastforce*
next
from ‹$j < length\ fs$› **have** $fs\ !\ j \in set\ fs$ **by** (*rule nth-mem*)
with *fs-nonzero* **show** $rep\text{-}list\ ?a \neq 0$ **by** (*auto simp*: *a*)
qed (*fact comp-b*)
qed
next
case *False*
with *assms*(*2*) **have** $v \in set\ ss$ **by** (*simp add*: *p*)
with *assms*(*1*) **show** *?thesis* **by** (*rule rb-aux-inv-D2*)
qed
qed

lemma *new-syz-sigs-minimal*:
assumes $\bigwedge u'\ v'.\ u' \in set\ ss \Longrightarrow v' \in set\ ss \Longrightarrow u'\ adds_t\ v' \Longrightarrow u' = v'$
assumes $u \in set\ (new\text{-}syz\text{-}sigs\ ss\ bs\ p)$ **and** $v \in set\ (new\text{-}syz\text{-}sigs\ ss\ bs\ p)$ **and** $u\ adds_t\ v$
shows $u = v$
proof (*rule sum-prodE*)
fix *a b*
assume *p*: $p = Inl\ (a,\ b)$
from *assms*(*2*, *3*) **have** $u \in set\ ss$ **and** $v \in set\ ss$ **by** (*simp-all add*: *p*)
thus *?thesis* **using** *assms*(*4*) **by** (*rule assms*(*1*))
next
fix *j*
assume *p*: $p = Inr\ j$
show *?thesis*
proof (*cases is-pot-ord*)
case *True*
have $transp\ (adds_t)$ **by** (*rule transpI*, *drule adds-term-trans*)
define ss' **where** $ss' = filter\text{-}min\ (adds_t)\ (map\ (\lambda b.\ term\text{-}of\text{-}pair\ (punit.lt\ (rep\text{-}list\ b),\ j))\ bs)$
note *assms*(*1*)
moreover have $u' = v'$ **if** $u' \in set\ ss'$ **and** $v' \in set\ ss'$ **and** $u'\ adds_t\ v'$ **for** $u'\ v'$
using ‹$transp\ (adds_t)$› *that* **unfolding** ss'*-def* **by** (*rule filter-min-minimal*)
moreover from *True assms*(*2*, *3*) **have** $u \in set\ (filter\text{-}min\text{-}append\ (adds_t)\ ss\ ss')$
and $v \in set\ (filter\text{-}min\text{-}append\ (adds_t)\ ss\ ss')$ **by** (*simp-all add*: *p* ss'*-def*)
ultimately show *?thesis* **using** *assms*(*4*) **by** (*rule filter-min-append-minimal*)
next
case *False*
with *assms*(*2*, *3*) **have** $u \in set\ ss$ **and** $v \in set\ ss$ **by** (*simp-all add*: *p*)
thus *?thesis* **using** *assms*(*4*) **by** (*rule assms*(*1*))
qed
qed

lemma *new-syz-sigs-distinct*:
assumes *distinct ss*

shows *distinct* (*new-syz-sigs ss bs p*)
proof (*rule sum-prodE*)
fix *a b*
assume $p = Inl\ (a,\ b)$
with *assms* **show** *?thesis* **by** *simp*
next
fix *j*
assume *p*: $p = Inr\ j$
show *?thesis*
proof (*cases is-pot-ord*)
case *True*
define *ss'* **where** *ss'* = *filter-min* ($adds_t$) (*map* (λ*b. term-of-pair* (*punit.lt* (*rep-list b*), *j*)) *bs*)
from *adds-term-refl* **have** *reflp* ($adds_t$) **by** (*rule reflpI*)
moreover note *assms*
moreover have *distinct ss'* **unfolding** *ss'-def* **using** ‹*reflp* ($adds_t$)› **by** (*rule filter-min-distinct*)
ultimately have *distinct* (*filter-min-append* ($adds_t$) *ss ss'*) **by** (*rule filter-min-append-distinct*)
thus *?thesis* **by** (*simp add*: *p ss'-def True*)
next
case *False*
with *assms* **show** *?thesis* **by** (*simp add*: *p*)
qed
qed

lemma *sig-crit'I-sig-crit*:
assumes *rb-aux-inv* (*bs*, *ss*, *p* # *ps*) **and** *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
shows *sig-crit' bs p*
proof −
have *rl*: *is-syz-sig dgrad u*
if *is-pred-syz* (*new-syz-sigs ss bs p*) *u* **and** *dgrad* (*pp-of-term u*) ≤ *dgrad-max dgrad* **for** *u*
proof −
from *that*(*1*) **obtain** *s* **where** *s* ∈ *set* (*new-syz-sigs ss bs p*) **and** *adds*: *s* $adds_t$ *u*
unfolding *is-pred-syz-def* **..**
from *assms*(*1*) *this*(*1*) **have** *is-syz-sig dgrad s* **by** (*rule new-syz-sigs-is-syz-sig*)
with *dgrad* **show** *?thesis* **using** *adds that*(*2*) **by** (*rule is-syz-sig-adds*)
qed
from *assms*(*1*) **have** *rb-aux-inv1 bs* **by** (*rule rb-aux-inv-D1*)
hence *bs-sub*: *set bs* ⊆ *dgrad-sig-set dgrad* **by** (*rule rb-aux-inv1-D1*)
show *?thesis*
proof (*rule sum-prodE*)
fix *a b*
assume *p*: $p = Inl\ (a,\ b)$
hence *Inl* (*a*, *b*) ∈ *set* (*p* # *ps*) **by** *simp*
with *assms*(*1*) **have** *a* ∈ *set bs* **and** *b* ∈ *set bs* **by** (*rule rb-aux-inv-D3*)+
with *bs-sub* **have** *a-in*: *a* ∈ *dgrad-sig-set dgrad* **and** *b-in*: *b* ∈ *dgrad-sig-set dgrad* **by** *fastforce*+

define *t1* **where** *t1* = *punit.lt* (*rep-list a*)
define *t2* **where** *t2* = *punit.lt* (*rep-list b*)
define *u* **where** *u* = *fst* (*spair-sigs a b*)
define *v* **where** *v* = *snd* (*spair-sigs a b*)
from *dgrad a-in* **have** *dgrad t1* ≤ *dgrad-max dgrad* **unfolding** *t1-def* **by** (*rule dgrad-sig-setD-rep-list-lt*)
moreover from *dgrad b-in* **have** *dgrad t2* ≤ *dgrad-max dgrad*
unfolding *t2-def* **by** (*rule dgrad-sig-setD-rep-list-lt*)
ultimately have *ord-class.max* (*dgrad t1*) (*dgrad t2*) ≤ *dgrad-max dgrad* **by** *simp*
with *dickson-grading-lcs*[*OF dgrad*] **have** *dgrad* (*lcs t1 t2*) ≤ *dgrad-max dgrad* **by** (*rule le-trans*)
have *u*: *u* = (*lcs t1 t2* − *t1*) ⊕ *lt a* **by** (*simp add*: *u-def spair-sigs-def t1-def t2-def Let-def*)
have *v*: *v* = (*lcs t1 t2* − *t2*) ⊕ *lt b* **by** (*simp add*: *v-def spair-sigs-def t1-def t2-def Let-def*)
have *1*: *spair-sigs a b* = (*u*, *v*) **by** (*simp add*: *u-def v-def*)
from *assms*(*2*) **have** (*is-pred-syz* (*new-syz-sigs ss bs p*) *u* ∨ *is-pred-syz* (*new-syz-sigs ss bs p*) *v*) ∨
(*is-rewritable bs a u* ∨ *is-rewritable bs b v*) **by** (*simp add*: *p 1*)
thus *?thesis*
proof
assume *is-pred-syz* (*new-syz-sigs ss bs p*) *u* ∨ *is-pred-syz* (*new-syz-sigs ss bs p*) *v*
thus *?thesis*
proof
assume *is-pred-syz* (*new-syz-sigs ss bs p*) *u*
moreover have *dgrad* (*pp-of-term u*) ≤ *dgrad-max dgrad*
proof (*simp add*: *u term-simps dickson-gradingD1*[*OF dgrad*], *rule*)
from *dgrad adds-lcs* **have** *dgrad* (*lcs t1 t2* − *t1*) ≤ *dgrad* (*lcs t1 t2*)
by (*rule dickson-grading-minus*)
also have ... ≤ *dgrad-max dgrad* **by** *fact*
finally show *dgrad* (*lcs t1 t2* − *t1*) ≤ *dgrad-max dgrad* **.**
next
from *a-in* **show** *dgrad* (*lp a*) ≤ *dgrad-max dgrad* **by** (*rule dgrad-sig-setD-lp*)
qed
ultimately have *is-syz-sig dgrad u* **by** (*rule rl*)
thus *?thesis* **by** (*simp add*: *p 1*)
next
assume *is-pred-syz* (*new-syz-sigs ss bs p*) *v*
moreover have *dgrad* (*pp-of-term v*) ≤ *dgrad-max dgrad*
proof (*simp add*: *v term-simps dickson-gradingD1*[*OF dgrad*], *rule*)
from *dgrad adds-lcs-2* **have** *dgrad* (*lcs t1 t2* − *t2*) ≤ *dgrad* (*lcs t1 t2*)
by (*rule dickson-grading-minus*)
also have ... ≤ *dgrad-max dgrad* **by** *fact*
finally show *dgrad* (*lcs t1 t2* − *t2*) ≤ *dgrad-max dgrad* **.**
next
from *b-in* **show** *dgrad* (*lp b*) ≤ *dgrad-max dgrad* **by** (*rule dgrad-sig-setD-lp*)
qed

ultimately have *is-syz-sig dgrad v* **by** (*rule rl*)
thus *?thesis* **by** (*simp add*: *p 1*)
qed
next
assume *is-rewritable bs a u* ∨ *is-rewritable bs b v*
thus *?thesis* **by** (*simp add*: *p 1*)
qed
next
fix *j*
assume *p* = *Inr j*
with *assms*(*2*) **have** *is-pred-syz* (*new-syz-sigs ss bs p*) (*term-of-pair* (*0*, *j*)) **by** *simp*
moreover have *dgrad* (*pp-of-term* (*term-of-pair* (*0*, *j*))) ≤ *dgrad-max dgrad*
by (*simp add*: *pp-of-term-of-pair dgrad-max-0*)
ultimately have *is-syz-sig dgrad* (*term-of-pair* (*0*, *j*)) **by** (*rule rl*)
thus *?thesis* **by** (*simp add*: ‹*p* = *Inr j*›)
qed
qed

lemma *rb-aux-inv-preserved-0*:
assumes *rb-aux-inv* (*bs*, *ss*, *p* # *ps*)
and ⋀*s*. *s* ∈ *set ss*′ ⟹ *is-syz-sig dgrad s*
and ⋀*a b*. *a* ∈ *set bs* ⟹ *b* ∈ *set bs* ⟹ *is-regular-spair a b* ⟹ *Inl* (*a*, *b*) ∉ *set ps* ⟹
Inl (*b*, *a*) ∉ *set ps* ⟹ ¬ *is-RB-in dgrad rword* (*set bs*) (*lt* (*spair a b*)) ⟹
∃ *q*∈*set ps*. *sig-of-pair q* = *lt* (*spair a b*) ∧ ¬ *sig-crit*′ *bs q*
and ⋀*j*. *j* < *length fs* ⟹ *p* = *Inr j* ⟹ *Inr j* ∉ *set ps* ⟹ *is-RB-in dgrad rword* (*set bs*) (*term-of-pair* (*0*, *j*)) ∧
rep-list (*monomial 1* (*term-of-pair* (*0*, *j*))) ∈ *ideal* (*rep-list* ‘ *set bs*)
shows *rb-aux-inv* (*bs*, *ss*′, *ps*)
proof −
from *assms*(*1*) **have** *rb-aux-inv1 bs* **by** (*rule rb-aux-inv-D1*)
show *?thesis* **unfolding** *rb-aux-inv.simps*
proof (*intro conjI ballI allI impI*)
fix *s*
assume *s* ∈ *set ss*′
thus *is-syz-sig dgrad s* **by** (*rule assms*(*2*))
next
fix *q1 q2*
assume *Inl* (*q1*, *q2*) ∈ *set ps*
hence *Inl* (*q1*, *q2*) ∈ *set* (*p* # *ps*) **by** *simp*
with *assms*(*1*) **show** *is-regular-spair q1 q2* **and** *q1* ∈ *set bs* **and** *q2* ∈ *set bs*
by (*rule rb-aux-inv-D3*)+
next
fix *j*
assume *Inr j* ∈ *set ps*
hence *Inr j* ∈ *set* (*p* # *ps*) **by** *simp*
with *assms*(*1*) **have** *j* < *length fs* **and** *length* (*filter* (*λq*. *sig-of-pair q* = *term-of-pair* (*0*, *j*)) (*p* # *ps*)) ≤ *1*

by (*rule rb-aux-inv-D4*)+
have *length* (*filter* (λq. *sig-of-pair q* = *term-of-pair* (*0*, *j*)) *ps*) $\leq$
length (*filter* (λq. *sig-of-pair q* = *term-of-pair* (*0*, *j*)) (*p* # *ps*)) **by** *simp*
also have ... $\leq$ *1* **by** *fact*
finally show *length* (*filter* (λq. *sig-of-pair q* = *term-of-pair* (*0*, *j*)) *ps*) $\leq$ *1* **.**
show $j <$ *length fs* **by** *fact*

fix *b*
assume $b \in$ *set bs*
with *assms*(*1*) ‹*Inr j* $\in$ *set* (*p* # *ps*)› **show** *lt b* $\prec_t$ *term-of-pair* (*0*, *j*) **by** (*rule rb-aux-inv-D4*)
next
from *assms*(*1*) **have** *sorted-wrt pair-ord* (*p* # *ps*) **by** (*rule rb-aux-inv-D5*)
thus *sorted-wrt pair-ord ps* **by** *simp*
next
fix *q*
assume $q \in$ *set ps*
from *assms*(*1*) **have** *sorted-wrt pair-ord* (*p* # *ps*) **by** (*rule rb-aux-inv-D5*)
hence $\bigwedge p'$. $p' \in$ *set ps* $\Longrightarrow$ *sig-of-pair p* $\preceq_t$ *sig-of-pair* p' **by** (*simp add*: *pair-ord-def*)
with ‹$q \in$ *set ps*› **have** *1*: *sig-of-pair p* $\preceq_t$ *sig-of-pair q* **by** *blast*
{
fix *b1 b2*
note *assms*(*1*)
moreover from ‹$q \in$ *set ps*› **have** $q \in$ *set* (*p* # *ps*) **by** *simp*
moreover assume *b1* $\in$ *set bs* **and** *b2* $\in$ *set bs* **and** *is-regular-spair b1 b2*
and *2*: *sig-of-pair q* $\prec_t$ *lt* (*spair b1 b2*)
ultimately show *Inl* (*b1*, *b2*) $\in$ *set ps* $\vee$ *Inl* (*b2*, *b1*) $\in$ *set ps*
proof (*rule rb-aux-inv-D6-1*)
assume *Inl* (*b1*, *b2*) $\in$ *set* (*p* # *ps*)
moreover from *1 2* **have** *sig-of-pair p* $\prec_t$ *lt* (*spair b1 b2*) **by** *simp*
ultimately have *Inl* (*b1*, *b2*) $\in$ *set ps*
by (*auto simp*: *sig-of-spair* ‹*is-regular-spair b1 b2*› *simp del*: *sig-of-pair.simps*)
thus *?thesis* **..**
next
assume *Inl* (*b2*, *b1*) $\in$ *set* (*p* # *ps*)
moreover from *1 2* **have** *sig-of-pair p* $\prec_t$ *lt* (*spair b1 b2*) **by** *simp*
ultimately have *Inl* (*b2*, *b1*) $\in$ *set ps*
by (*auto simp*: *sig-of-spair* ‹*is-regular-spair b1 b2*› *sig-of-spair-commute simp del*: *sig-of-pair.simps*)
thus *?thesis* **..**
qed
}
{
fix *j*
note *assms*(*1*)
moreover from ‹$q \in$ *set ps*› **have** $q \in$ *set* (*p* # *ps*) **by** *simp*
moreover assume $j <$ *length fs* **and** *2*: *sig-of-pair q* $\prec_t$ *term-of-pair* (*0*, *j*)
ultimately have *Inr j* $\in$ *set* (*p* # *ps*) **by** (*rule rb-aux-inv-D6-2*)

moreover from *1 2* **have** *sig-of-pair p* $\prec_t$ *sig-of-pair* (*Inr j*) **by** *simp*
ultimately show *Inr j* $\in$ *set ps* **by** *auto*
}
next
fix *b q*
assume *b* $\in$ *set bs* **and** *q* $\in$ *set ps*
hence *b* $\in$ *set bs* **and** *q* $\in$ *set* (*p # ps*) **by** *simp-all*
with *assms*(*1*) **show** *lt b* $\preceq_t$ *sig-of-pair q* **by** (*rule rb-aux-inv-D7*)
next
fix *j*
assume *j* < *length fs* **and** *Inr j* $\notin$ *set ps*
have *is-RB-in dgrad rword* (*set bs*) (*term-of-pair* (*0*, *j*)) $\wedge$
rep-list (*monomial 1* (*term-of-pair* (*0*, *j*))) $\in$ *ideal* (*rep-list* ' *set bs*)
proof (*cases p* = *Inr j*)
case *True*
with ‹*j* < *length fs*› **show** *?thesis* **using** ‹*Inr j* $\notin$ *set ps*› **by** (*rule assms*(*4*))
next
case *False*
with ‹*Inr j* $\notin$ *set ps*› **have** *Inr j* $\notin$ *set* (*p # ps*) **by** *simp*
with *assms*(*1*) ‹*j* < *length fs*› *rb-aux-inv-D9* **show** *?thesis* **by** *blast*
qed
thus *is-RB-in dgrad rword* (*set bs*) (*term-of-pair* (*0*, *j*))
and *rep-list* (*monomial 1* (*term-of-pair* (*0*, *j*))) $\in$ *ideal* (*rep-list* ' *set bs*) **by** *simp-all*
qed (*fact*, *rule assms*(*3*))
qed

lemma *rb-aux-inv-preserved-1*:
assumes *rb-aux-inv* (*bs*, *ss*, *p # ps*) **and** *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
shows *rb-aux-inv* (*bs*, *new-syz-sigs ss bs p*, *ps*)
proof −
from *assms*(*1*) **have** *rb-aux-inv1 bs* **by** (*rule rb-aux-inv-D1*)
hence *bs-sub*: *set bs* $\subseteq$ *dgrad-sig-set dgrad* **by** (*rule rb-aux-inv1-D1*)
from *assms*(*1*, *2*) **have** *sig-crit′ bs p* **by** (*rule sig-crit′I-sig-crit*)
from *assms*(*1*) **show** *?thesis*
proof (*rule rb-aux-inv-preserved-0*)
fix *s*
assume *s* $\in$ *set* (*new-syz-sigs ss bs p*)
with *assms*(*1*) **show** *is-syz-sig dgrad s* **by** (*rule new-syz-sigs-is-syz-sig*)
next
fix *a b*
assume *1*: *a* $\in$ *set bs* **and** *2*: *b* $\in$ *set bs* **and** *3*: *is-regular-spair a b* **and** *4*: *Inl* (*a*, *b*) $\notin$ *set ps*
and *5*: *Inl* (*b*, *a*) $\notin$ *set ps* **and** *6*: $\neg$ *is-RB-in dgrad rword* (*set bs*) (*lt* (*spair a b*))
from *assms*(*1*, *2*) **have** *sig-crit′ bs p* **by** (*rule sig-crit′I-sig-crit*)
show $\exists$ *q*$\in$*set ps*. *sig-of-pair q* = *lt* (*spair a b*) $\wedge$ $\neg$ *sig-crit′ bs q*
proof (*cases p* = *Inl* (*a*, *b*) $\vee$ *p* = *Inl* (*b*, *a*))
case *True*

hence *sig-of-p*: *lt* (*spair a b*) = *sig-of-pair p*
proof
assume *p*: *p* = *Inl* (*a*, *b*)
from *3* **show** *?thesis* **by** (*simp only*: *p sig-of-spair*)
next
assume *p*: *p* = *Inl* (*b*, *a*)
from *3* **have** *is-regular-spair b a* **by** (*rule is-regular-spair-sym*)
thus *?thesis* **by** (*simp only*: *p sig-of-spair spair-comm*[*of a*] *lt-uminus*)
qed
note *assms*(*1*)
moreover have *is-RB-upt dgrad rword* (*set bs*) (*lt* (*spair a b*)) **unfolding** *sig-of-p*
using *assms*(*1*) **by** (*rule rb-aux-inv-is-RB-upt-Cons*)
moreover have *dgrad* (*lp* (*spair a b*)) ≤ *dgrad-max dgrad*
proof (*rule dgrad-sig-setD-lp*, *rule dgrad-sig-set-closed-spair*, *fact dgrad*)
from ‹*a* ∈ *set bs*› *bs-sub* **show** *a* ∈ *dgrad-sig-set dgrad* **..**
next
from ‹*b* ∈ *set bs*› *bs-sub* **show** *b* ∈ *dgrad-sig-set dgrad* **..**
qed
moreover obtain *c* **where** *crw*: *is-canon-rewriter rword* (*set bs*) (*lt* (*spair a b*)) *c*
proof (*rule ord-term-lin.linorder-cases*)
from *3* **have** *rep-list b* ≠ *0* **by** (*rule is-regular-spairD2*)
moreover assume *punit.lt* (*rep-list b*) ⊕ *lt a* $\prec_t$ *punit.lt* (*rep-list a*) ⊕ *lt b*
ultimately have *lt* (*spair b a*) = (*lcs* (*punit.lt* (*rep-list b*)) (*punit.lt* (*rep-list a*)) − *punit.lt* (*rep-list b*)) ⊕ *lt b*
by (*rule lt-spair*)
hence *lt* (*spair a b*) = (*lcs* (*punit.lt* (*rep-list b*)) (*punit.lt* (*rep-list a*)) − *punit.lt* (*rep-list b*)) ⊕ *lt b*
by (*simp add*: *spair-comm*[*of a*])
hence *lt b* $adds_t$ *lt* (*spair a b*) **by** (*rule adds-termI*)
from ‹*rep-list b* ≠ *0*› **have** *b* ≠ *0* **by** (*auto simp*: *rep-list-zero*)
show *?thesis* **by** (*rule is-rewrite-ord-finite-canon-rewriterE*, *fact rword*, *fact finite-set*, *fact*+)
next
from *3* **have** *rep-list a* ≠ *0* **by** (*rule is-regular-spairD1*)
moreover assume *punit.lt* (*rep-list a*) ⊕ *lt b* $\prec_t$ *punit.lt* (*rep-list b*) ⊕ *lt a*
ultimately have *lt* (*spair a b*) = (*lcs* (*punit.lt* (*rep-list a*)) (*punit.lt* (*rep-list b*)) − *punit.lt* (*rep-list a*)) ⊕ *lt a*
by (*rule lt-spair*)
hence *lt a* $adds_t$ *lt* (*spair a b*) **by** (*rule adds-termI*)
from ‹*rep-list a* ≠ *0*› **have** *a* ≠ *0* **by** (*auto simp*: *rep-list-zero*)
show *?thesis* **by** (*rule is-rewrite-ord-finite-canon-rewriterE*, *fact rword*, *fact finite-set*, *fact*+)
next
from *3* **have** *punit.lt* (*rep-list b*) ⊕ *lt a* ≠ *punit.lt* (*rep-list a*) ⊕ *lt b*
by (*rule is-regular-spairD3*)
moreover assume *punit.lt* (*rep-list b*) ⊕ *lt a* = *punit.lt* (*rep-list a*) ⊕ *lt b*
ultimately show *?thesis* **..**

qed
moreover from *6* **have** ¬ *is-syz-sig dgrad* (*lt* (*spair a b*)) **by** (*simp add*: *is-RB-in-def*)
moreover from *6 crw* **have** *is-sig-red* ($\prec_t$) (=) (*set bs*) (*monom-mult 1* (*lp* (*spair a b*) − *lp c*) *c*)
by (*simp add*: *is-RB-in-def*)
ultimately obtain *x y* **where** *7*: *x* ∈ *set bs* **and** *8*: *y* ∈ *set bs* **and** *9*: *is-regular-spair x y*
and *10*: *lt* (*spair x y*) = *lt* (*spair a b*) **and** *11*: ¬ *sig-crit′ bs* (*Inl* (*x*, *y*))
by (*rule lemma-12*)
from *this*(*5*) ‹*sig-crit′ bs p*› **have** *Inl* (*x*, *y*) ≠ *p* **and** *Inl* (*y*, *x*) ≠ *p*
by (*auto simp only*: *sig-crit′-sym*)
show *?thesis*
proof (*cases Inl* (*x*, *y*) ∈ *set ps* ∨ *Inl* (*y*, *x*) ∈ *set ps*)
case *True*
thus *?thesis*
proof
assume *Inl* (*x*, *y*) ∈ *set ps*
show *?thesis*
proof (*intro bexI conjI*)
show *sig-of-pair* (*Inl* (*x*, *y*)) = *lt* (*spair a b*) **by** (*simp only*: *sig-of-spair 9 10*)
qed *fact+*
next
assume *Inl* (*y*, *x*) ∈ *set ps*
show *?thesis*
proof (*intro bexI conjI*)
from *9* **have** *is-regular-spair y x* **by** (*rule is-regular-spair-sym*)
thus *sig-of-pair* (*Inl* (*y*, *x*)) = *lt* (*spair a b*)
by (*simp only*: *sig-of-spair spair-comm*[*of y*] *lt-uminus 10*)
next
from *11* **show** ¬ *sig-crit′ bs* (*Inl* (*y*, *x*)) **by** (*auto simp only*: *sig-crit′-sym*)
qed *fact*
qed
next
case *False*
note *assms*(*1*) *7 8 9*
moreover from *False* ‹*Inl* (*x*, *y*) ≠ *p*› ‹*Inl* (*y*, *x*) ≠ *p*› **have** *Inl* (*x*, *y*) ∉ *set* (*p* # *ps*)
and *Inl* (*y*, *x*) ∉ *set* (*p* # *ps*) **by** *auto*
moreover from *6* **have** ¬ *is-RB-in dgrad rword* (*set bs*) (*lt* (*spair x y*)) **by** (*simp add*: *10*)
ultimately obtain *q* **where** *12*: *q* ∈ *set* (*p* # *ps*) **and** *13*: *sig-of-pair q* = *lt* (*spair x y*)
and *14*: ¬ *sig-crit′ bs q* **by** (*rule rb-aux-inv-D8*)
from *12 14* ‹*sig-crit′ bs p*› **have** *q* ∈ *set ps* **by** *auto*
with *13 14* **show** *?thesis* **unfolding** *10* **by** *blast*
qed
next

case *False*
with *4 5* **have** *Inl* (*a*, *b*) $\notin$ *set* (*p* # *ps*) **and** *Inl* (*b*, *a*) $\notin$ *set* (*p* # *ps*) **by** *auto*
with *assms*(*1*) *1 2 3* **obtain** *q* **where** *7*: *q* $\in$ *set* (*p* # *ps*) **and** *8*: *sig-of-pair q* = *lt* (*spair a b*)
and *9*: ¬ *sig-crit′ bs q* **using** *6* **by** (*rule rb-aux-inv-D8*)
from *7 9* ‹*sig-crit′ bs p*› **have** *q* $\in$ *set ps* **by** *auto*
with *8 9* **show** *?thesis* **by** *blast*
qed
next
fix *j*
assume *j* < *length fs*
assume *p*: *p* = *Inr j*
with ‹*sig-crit′ bs p*› **have** *is-syz-sig dgrad* (*term-of-pair* (*0*, *j*)) **by** *simp*
hence *is-RB-in dgrad rword* (*set bs*) (*term-of-pair* (*0*, *j*)) **by** (*rule is-RB-inI2*)
moreover have *rep-list* (*monomial 1* (*term-of-pair* (*0*, *j*))) $\in$ *ideal* (*rep-list* ‘ *set bs*)
proof (*rule sig-red-zero-idealI*, *rule syzygy-crit*)
from *assms*(*1*) **have** *is-RB-upt dgrad rword* (*set bs*) (*sig-of-pair p*)
by (*rule rb-aux-inv-is-RB-upt-Cons*)
with *dgrad* **have** *is-sig-GB-upt dgrad* (*set bs*) (*sig-of-pair p*)
by (*rule is-RB-upt-is-sig-GB-upt*)
thus *is-sig-GB-upt dgrad* (*set bs*) (*term-of-pair* (*0*, *j*)) **by** (*simp add*: *p*)
next
show *monomial 1* (*term-of-pair* (*0*, *j*)) $\in$ *dgrad-sig-set dgrad*
by (*rule dgrad-sig-set-closed-monomial*, *simp-all add*: *term-simps dgrad-max-0* ‹*j* < *length fs*›)
next
show *lt* (*monomial* (*1*::′*b*) (*term-of-pair* (*0*, *j*))) = *term-of-pair* (*0*, *j*) **by** (*simp add*: *lt-monomial*)
qed (*fact dgrad*, *fact*)
ultimately show *is-RB-in dgrad rword* (*set bs*) (*term-of-pair* (*0*, *j*)) ∧
rep-list (*monomial 1* (*term-of-pair* (*0*, *j*))) $\in$ *ideal* (*rep-list* ‘ *set bs*) **..**
qed
qed

lemma *rb-aux-inv-preserved-2*:
assumes *rb-aux-inv* (*bs*, *ss*, *p* # *ps*) **and** *rep-list* (*sig-trd bs* (*poly-of-pair p*)) = *0*
shows *rb-aux-inv* (*bs*, *lt* (*sig-trd bs* (*poly-of-pair p*)) # *new-syz-sigs ss bs p*, *ps*)
proof −
let *?p* = *sig-trd bs* (*poly-of-pair p*)
have *0*: (*sig-red* ($\prec_t$) ($\preceq$) (*set bs*))** (*poly-of-pair p*) *?p*
by (*rule sig-trd-red-rtrancl*)
hence *eq*: *lt ?p* = *lt* (*poly-of-pair p*) **by** (*rule sig-red-regular-rtrancl-lt*)
from *assms*(*1*) **have** *inv1*: *rb-aux-inv1 bs* **by** (*rule rb-aux-inv-D1*)
have ∗: *is-syz-sig dgrad* (*lt* (*poly-of-pair p*))
proof (*rule is-syz-sigI*)
have *poly-of-pair p* ≠ *0* **by** (*rule pair-list-nonzero*, *fact*, *simp*)

hence *lc* (*poly-of-pair p*) $\neq$ *0* **by** (*rule lc-not-0*)
moreover from *0* **have** *lc ?p* = *lc* (*poly-of-pair p*) **by** (*rule sig-red-regular-rtrancl-lc*)
ultimately have *lc ?p* $\neq$ *0* **by** *simp*
thus *?p* $\neq$ *0* **by** (*simp add*: *lc-eq-zero-iff*)
next
note *dgrad*(*1*)
moreover from *inv1* **have** *set bs* $\subseteq$ *dgrad-sig-set dgrad* **by** (*rule rb-aux-inv1-D1*)
moreover have *poly-of-pair p* $\in$ *dgrad-sig-set dgrad* **by** (*rule pair-list-dgrad-sig-set*, *fact*, *simp*)
ultimately show *?p* $\in$ *dgrad-sig-set dgrad* **using** *0* **by** (*rule dgrad-sig-set-closed-sig-red-rtrancl*)
qed (*fact eq*, *fact assms*(*2*))
hence *rb*: *is-RB-in dgrad rword* (*set bs*) (*lt* (*poly-of-pair p*)) **by** (*rule is-RB-inI2*)
from *assms*(*1*) **show** *?thesis*
proof (*rule rb-aux-inv-preserved-0*)
fix *s*
assume *s* $\in$ *set* (*lt ?p* # *new-syz-sigs ss bs p*)
hence *s* = *lt* (*poly-of-pair p*) $\vee$ *s* $\in$ *set* (*new-syz-sigs ss bs p*) **by** (*simp add*: *eq*)
thus *is-syz-sig dgrad s*
proof
assume *s* = *lt* (*poly-of-pair p*)
with $*$ **show** *?thesis* **by** *simp*
next
assume *s* $\in$ *set* (*new-syz-sigs ss bs p*)
with *assms*(*1*) **show** *?thesis* **by** (*rule new-syz-sigs-is-syz-sig*)
qed
next
fix *a b*
assume *1*: *a* $\in$ *set bs* **and** *2*: *b* $\in$ *set bs* **and** *3*: *is-regular-spair a b* **and** *4*: *Inl* (*a*, *b*) $\notin$ *set ps*
and *5*: *Inl* (*b*, *a*) $\notin$ *set ps* **and** *6*: $\neg$ *is-RB-in dgrad rword* (*set bs*) (*lt* (*spair a b*))
have *p* $\in$ *set* (*p* # *ps*) **by** *simp*
with *assms*(*1*) **have** *sig-of-p*: *sig-of-pair p* = *lt* (*poly-of-pair p*) **by** (*rule pair-list-sig-of-pair*)
from *rb 6* **have** *neq*: *lt* (*poly-of-pair p*) $\neq$ *lt* (*spair a b*) **by** *auto*
hence *p* $\neq$ *Inl* (*a*, *b*) **and** *p* $\neq$ *Inl* (*b*, *a*) **by** (*auto simp*: *spair-comm*[*of a*])
with *4 5* **have** *Inl* (*a*, *b*) $\notin$ *set* (*p* # *ps*) **and** *Inl* (*b*, *a*) $\notin$ *set* (*p* # *ps*) **by** *auto*
with *assms*(*1*) *1 2 3* **obtain** *q* **where** *7*: *q* $\in$ *set* (*p* # *ps*) **and** *8*: *sig-of-pair q* = *lt* (*spair a b*)
and *9*: $\neg$ *sig-crit′ bs q* **using** *6* **by** (*rule rb-aux-inv-D8*)
from *this*(*1*, *2*) *neq* **have** *q* $\in$ *set ps* **by** (*auto simp*: *sig-of-p*)
thus $\exists$ *q*$\in$*set ps*. *sig-of-pair q* = *lt* (*spair a b*) $\wedge$ $\neg$ *sig-crit′ bs q* **using** *8 9* **by** *blast*
next
fix *j*
assume *j* < *length fs*
assume *p*: *p* = *Inr j*
from *rb* **have** *is-RB-in dgrad rword* (*set bs*) (*term-of-pair* (*0*, *j*)) **by** (*simp add*: *p lt-monomial*)

moreover have *rep-list* (*monomial 1* (*term-of-pair* (*0*, *j*))) ∈ *ideal* (*rep-list* ' *set bs*)
proof (*rule sig-red-zero-idealI*, *rule sig-red-zeroI*)
from *0* **show** (*sig-red* ($\prec_t$) ($\preceq$) (*set bs*))** (*monomial 1* (*term-of-pair* (*0*, *j*))) *?p* **by** (*simp add*: *p*)
qed *fact*
ultimately show *is-RB-in dgrad rword* (*set bs*) (*term-of-pair* (*0*, *j*)) ∧
rep-list (*monomial 1* (*term-of-pair* (*0*, *j*))) ∈ *ideal* (*rep-list* ' *set bs*) **..**
qed
qed

lemma *rb-aux-inv-preserved-3*:
assumes *rb-aux-inv* (*bs*, *ss*, *p* # *ps*) **and** ¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
and *rep-list* (*sig-trd bs* (*poly-of-pair p*)) ≠ *0*
shows *rb-aux-inv* ((*sig-trd bs* (*poly-of-pair p*)) # *bs*, *new-syz-sigs ss bs p*,
add-spairs ps bs (*sig-trd bs* (*poly-of-pair p*)))
and *lt* (*sig-trd bs* (*poly-of-pair p*)) ∉ *lt* ' *set bs*
proof −
have *p* ∈ *set* (*p* # *ps*) **by** *simp*
with *assms*(*1*) **have** *sig-of-p*: *sig-of-pair p* = *lt* (*poly-of-pair p*)
and *p-in*: *poly-of-pair p* ∈ *dgrad-sig-set dgrad*
by (*rule pair-list-sig-of-pair*, *rule pair-list-dgrad-sig-set*)
define *p′* **where** *p′* = *sig-trd bs* (*poly-of-pair p*)
from *assms*(*1*) **have** *inv1*: *rb-aux-inv1 bs* **by** (*rule rb-aux-inv-D1*)
hence *bs-sub*: *set bs* ⊆ *dgrad-sig-set dgrad* **by** (*rule rb-aux-inv1-D1*)
have *p-red*: (*sig-red* ($\prec_t$) ($\preceq$) (*set bs*))** (*poly-of-pair p*) *p′*
and *p′-irred*: ¬ *is-sig-red* ($\prec_t$) ($\preceq$) (*set bs*) *p′*
unfolding *p′-def* **by** (*rule sig-trd-red-rtrancl*, *rule sig-trd-irred*)
from *dgrad bs-sub p-in p-red* **have** *p′-in*: *p′* ∈ *dgrad-sig-set dgrad*
by (*rule dgrad-sig-set-closed-sig-red-rtrancl*)
from *p-red* **have** *lt-p′*: *lt p′* = *lt* (*poly-of-pair p*) **by** (*rule sig-red-regular-rtrancl-lt*)
have *sig-merge*: *sig-of-pair p* $\preceq_t$ *sig-of-pair q* **if** *q* ∈ *set* (*add-spairs ps bs p′*) **for** *q*
using *that* **unfolding** *add-spairs-def set-merge-wrt*
proof
assume *q* ∈ *set* (*new-spairs bs p′*)
then obtain *b0* **where** *is-regular-spair p′ b0* **and** *q* = *Inl* (*p′*, *b0*) **by** (*rule in-new-spairsE*)
hence *sig-of-q*: *sig-of-pair q* = *lt* (*spair p′ b0*) **by** (*simp only*: *sig-of-spair*)
show *?thesis* **unfolding** *sig-of-q sig-of-p lt-p′*[*symmetric*] **by** (*rule is-regular-spair-lt-ge-1*, *fact*)
next
assume *q* ∈ *set ps*
moreover from *assms*(*1*) **have** *sorted-wrt pair-ord* (*p* # *ps*) **by** (*rule rb-aux-inv-D5*)
ultimately show *?thesis* **by** (*simp add*: *pair-ord-def*)
qed
have *sig-of-p-less*: *sig-of-pair p* $\prec_t$ *term-of-pair* (*0*, *j*) **if** *Inr j* ∈ *set ps* **for** *j*
proof (*intro ord-term-lin.le-neq-trans*)

from *assms*(*1*) **have** *sorted-wrt pair-ord* (*p* # *ps*) **by** (*rule rb-aux-inv-D5*)
with ‹*Inr j* ∈ *set ps*› **show** *sig-of-pair p* $\preceq_t$ *term-of-pair* (*0*, *j*)
by (*auto simp*: *pair-ord-def*)
next
from *assms*(*1*) *that* **show** *sig-of-pair p* ≠ *term-of-pair* (*0*, *j*) **by** (*rule Inr-in-tailD*)
qed
have *lt-p-gr*: *lt b* $\prec_t$ *lt* (*poly-of-pair p*) **if** *b* ∈ *set bs* **for** *b* **unfolding** *sig-of-p*[*symmetric*]
using *assms*(*1*, *2*) *that* **by** (*rule not-sig-crit*)
have *inv1*: *rb-aux-inv1* (*p*′ # *bs*) **unfolding** *rb-aux-inv1-def*
proof (*intro conjI impI allI*)
from *bs-sub p*′*-in* **show** *set* (*p*′ # *bs*) ⊆ *dgrad-sig-set dgrad* **by** *simp*
next
from *inv1* **have** *0* ∉ *rep-list* ‘ *set bs* **by** (*rule rb-aux-inv1-D2*)
with *assms*(*3*) **show** *0* ∉ *rep-list* ‘ *set* (*p*′ # *bs*) **by** (*simp add*: *p*′*-def*)
next
from *inv1* **have** *sorted-wrt* (λ*x y*. *lt y* $\prec_t$ *lt x*) *bs* **by** (*rule rb-aux-inv1-D3*)
with *lt-p-gr* **show** *sorted-wrt* (λ*x y*. *lt y* $\prec_t$ *lt x*) (*p*′ # *bs*) **by** (*simp add*: *lt-p*′)
next
fix *i*
assume *i* < *length* (*p*′ # *bs*)
have (¬ *is-sig-red* ($\prec_t$) (⪯) (*set* (*drop* (*Suc i*) (*p*′ # *bs*))) ((*p*′ # *bs*) ! *i*)) ∧
((∃ *j*<*length fs*. *lt* ((*p*′ # *bs*) ! *i*) = *lt* (*monomial* (*1*::′*b*) (*term-of-pair* (*0*, *j*))) ∧
punit.lt (*rep-list* ((*p*′ # *bs*) ! *i*)) ⪯ *punit.lt* (*rep-list* (*monomial 1* (*term-of-pair* (*0*, *j*))))) ∨
(∃ *p*∈*set* (*p*′ # *bs*). ∃ *q*∈*set* (*p*′ # *bs*). *is-regular-spair p q* ∧ *rep-list* (*spair p q*) ≠ *0* ∧
lt ((*p*′ # *bs*) ! *i*) = *lt* (*spair p q*) ∧
punit.lt (*rep-list* ((*p*′ # *bs*) ! *i*)) ⪯ *punit.lt* (*rep-list* (*spair p q*)))) ∧
is-RB-upt dgrad rword (*set* (*drop* (*Suc i*) (*p*′ # *bs*))) (*lt* ((*p*′ # *bs*) ! *i*))
(**is** *?thesis1* ∧ *?thesis2* ∧ *?thesis3*)
proof (*cases i*)
case *0*
show *?thesis*
proof (*simp add*: ‹*i* = *0*› *p*′*-irred del*: *bex-simps*, *rule conjI*)
show (∃ *j*<*length fs*. *lt p*′ = *lt* (*monomial* (*1*::′*b*) (*term-of-pair* (*0*, *j*))) ∧
punit.lt (*rep-list p*′) ⪯ *punit.lt* (*rep-list* (*monomial 1* (*term-of-pair* (*0*, *j*))))) ∨
(∃ *p*∈*insert p*′ (*set bs*). ∃ *q*∈*insert p*′ (*set bs*). *is-regular-spair p q* ∧ *rep-list* (*spair p q*) ≠ *0* ∧
lt p′ = *lt* (*spair p q*) ∧ *punit.lt* (*rep-list p*′) ⪯ *punit.lt* (*rep-list* (*spair p q*)))
proof (*rule sum-prodE*)
fix *a b*
assume *p*: *p* = *Inl* (*a*, *b*)
have *Inl* (*a*, *b*) ∈ *set* (*p* # *ps*) **by** (*simp add*: *p*)
with *assms*(*1*) **have** *a* ∈ *set bs* **and** *b* ∈ *set bs* **and** *is-regular-spair a b*
by (*rule rb-aux-inv-D3*)+
from *p-red* **have** *p*′*-red*: (*sig-red* ($\prec_t$) (⪯) (*set bs*))** (*spair a b*) *p*′ **by**

(*simp add*: *p*)
hence (*punit.red* (*rep-list* ‘ *set bs*))** (*rep-list* (*spair a b*)) (*rep-list p*′)
by (*rule sig-red-red-rtrancl*)
moreover from *assms*(*3*) **have** *rep-list p*′ ≠ *0* **by** (*simp add*: *p*′*-def*)
ultimately have *rep-list* (*spair a b*) ≠ *0* **by** (*auto dest*: *punit.rtrancl-0*)
moreover from *p*′*-red* **have** *lt p*′ = *lt* (*spair a b*)
and *punit.lt* (*rep-list p*′) ⪯ *punit.lt* (*rep-list* (*spair a b*))
by (*rule sig-red-regular-rtrancl-lt*, *rule sig-red-rtrancl-lt-rep-list*)
ultimately show *?thesis* **using** ‹*a* ∈ *set bs*› ‹*b* ∈ *set bs*› ‹*is-regular-spair a b*› **by** *blast*
next
fix *j*
assume *p* = *Inr j*
hence *Inr j* ∈ *set* (*p* # *ps*) **by** *simp*
with *assms*(*1*) **have** *j* < *length fs* **by** (*rule rb-aux-inv-D4*)
from *p-red* **have** (*sig-red* ($\prec_t$) (⪯) (*set bs*))** (*monomial 1* (*term-of-pair* (*0*, *j*))) *p*′
by (*simp add*: ‹*p* = *Inr j*›)
hence *lt p*′ = *lt* (*monomial* (*1*::′*b*) (*term-of-pair* (*0*, *j*)))
and *punit.lt* (*rep-list p*′) ⪯ *punit.lt* (*rep-list* (*monomial 1* (*term-of-pair* (*0*, *j*))))
by (*rule sig-red-regular-rtrancl-lt*, *rule sig-red-rtrancl-lt-rep-list*)
with ‹*j* < *length fs*› **show** *?thesis* **by** *blast*
qed
next
from *assms*(*1*) **show** *is-RB-upt dgrad rword* (*set bs*) (*lt p*′) **unfolding** *lt-p*′ *sig-of-p*[*symmetric*]
by (*rule rb-aux-inv-is-RB-upt-Cons*)
qed
next
case (*Suc i*′)
with ‹*i* < *length* (*p*′ # *bs*)› **have** *i*′: *i*′ < *length bs* **by** *simp*
show *?thesis*
proof (*simp add*: ‹*i* = *Suc i*′› *del*: *bex-simps*, *intro conjI*)
from *inv1 i*′ **show** ¬ *is-sig-red* ($\prec_t$) (⪯) (*set* (*drop* (*Suc i*′) *bs*)) (*bs* ! *i*′)
by (*rule rb-aux-inv1-D4*)
next
from *inv1 i*′
show (∃ *j*<*length fs*. *lt* (*bs* ! *i*′) = *lt* (*monomial* (*1*::′*b*) (*term-of-pair* (*0*, *j*))) ∧
punit.lt (*rep-list* (*bs* ! *i*′)) ⪯ *punit.lt* (*rep-list* (*monomial 1* (*term-of-pair* (*0*, *j*))))) ∨
(∃ *p*∈*insert p*′ (*set bs*). ∃ *q*∈*insert p*′ (*set bs*). *is-regular-spair p q* ∧ *rep-list* (*spair p q*) ≠ *0* ∧
lt (*bs* ! *i*′) = *lt* (*spair p q*) ∧ *punit.lt* (*rep-list* (*bs* ! *i*′)) ⪯ *punit.lt* (*rep-list* (*spair p q*)))
by (*auto elim*!: *rb-aux-inv1-E*)
next
from *inv1 i*′ **show** *is-RB-upt dgrad rword* (*set* (*drop* (*Suc i*′) *bs*)) (*lt* (*bs* !

i′))
by (*rule rb-aux-inv1-D5*)
qed
qed
thus *?thesis1* **and** *?thesis2* **and** *?thesis3* **by** *simp-all*
qed
have *rb*: *is-RB-in dgrad rword* (*set* (p' # *bs*)) (*sig-of-pair p*)
proof (*rule is-RB-inI1*)
have $p' \in$ *set* (p' # *bs*) **by** *simp*
with *inv1* **have** *is-canon-rewriter rword* (*set* (p' # *bs*)) (*lt* p') p'
by (*rule is-canon-rewriterI-eq-sig*)
thus *is-canon-rewriter rword* (*set* (p' # *bs*)) (*sig-of-pair p*) p' **by** (*simp add*: *lt-p′ sig-of-p*)
next
from *p′-irred* **have** $\neg$ *is-sig-red* ($\prec_t$) (=) (*set bs*) p'
by (*simp add*: *is-sig-red-top-tail-cases*)
with *sig-irred-regular-self* **have** $\neg$ *is-sig-red* ($\prec_t$) (=) ($\{p'\} \cup$ *set bs*) p'
by (*simp add*: *is-sig-red-Un del*: *Un-insert-left*)
thus $\neg$ *is-sig-red* ($\prec_t$) (=) (*set* (p' # *bs*)) (*monom-mult 1* (*pp-of-term* (*sig-of-pair p*) $-$ *lp* p') p')
by (*simp add*: *lt-p′ sig-of-p*)
qed
show *rb-aux-inv* (p' # *bs*, *new-syz-sigs ss bs p*, *add-spairs ps bs* p')
unfolding *rb-aux-inv.simps*
proof (*intro conjI ballI allI impI*)
show *rb-aux-inv1* (p' # *bs*) **by** (*fact inv1*)
next
fix *s*
assume $s \in$ *set* (*new-syz-sigs ss bs p*)
with *assms*(*1*) **show** *is-syz-sig dgrad s* **by** (*rule new-syz-sigs-is-syz-sig*)
next
fix *q1 q2*
assume *Inl* (*q1*, *q2*) $\in$ *set* (*add-spairs ps bs* p')
hence *Inl* (*q1*, *q2*) $\in$ *set* (*new-spairs bs* p') $\vee$ *Inl* (*q1*, *q2*) $\in$ *set* (*p* # *ps*)
by (*auto simp*: *add-spairs-def set-merge-wrt*)
hence *is-regular-spair q1 q2* $\wedge$ $q1 \in$ *set* (p' # *bs*) $\wedge$ $q2 \in$ *set* (p' # *bs*)
proof
assume *Inl* (*q1*, *q2*) $\in$ *set* (*new-spairs bs* p')
hence $q1 = p'$ **and** $q2 \in$ *set bs* **and** *is-regular-spair* p' *q2* **by** (*rule in-new-spairsD*)+
thus *?thesis* **by** *simp*
next
assume *Inl* (*q1*, *q2*) $\in$ *set* (*p* # *ps*)
with *assms*(*1*) **have** *is-regular-spair q1 q2* **and** $q1 \in$ *set bs* **and** $q2 \in$ *set bs*
by (*rule rb-aux-inv-D3*)+
thus *?thesis* **by** *simp*
qed
thus *is-regular-spair q1 q2* **and** $q1 \in$ *set* (p' # *bs*) **and** $q2 \in$ *set* (p' # *bs*) **by** *simp-all*
next

fix *j*
assume *Inr j* ∈ *set* (*add-spairs ps bs p*′)
hence *Inr j* ∈ *set ps* **by** (*simp add*: *add-spairs-def set-merge-wrt Inr-not-in-new-spairs*)
hence *Inr j* ∈ *set* (*p* # *ps*) **by** *simp*
with *assms*(*1*) **show** $j <$ *length fs* **by** (*rule rb-aux-inv-D4*)

fix *b*
assume *b* ∈ *set* (*p*′ # *bs*)
hence *b* = *p*′ ∨ *b* ∈ *set bs* **by** *simp*
thus *lt b* $\prec_t$ *term-of-pair* (*0*, *j*)
proof
assume *b* = *p*′
hence *lt b* = *sig-of-pair p* **by** (*simp only*: *lt-p*′ *sig-of-p*)
also from ‹*Inr j* ∈ *set ps*› **have** ... $\prec_t$ *term-of-pair* (*0*, *j*) **by** (*rule sig-of-p-less*)
finally show *?thesis* **.**
next
assume *b* ∈ *set bs*
with *assms*(*1*) ‹*Inr j* ∈ *set* (*p* # *ps*)› **show** *?thesis* **by** (*rule rb-aux-inv-D4*)
qed
next
fix *j*
assume *Inr j* ∈ *set* (*add-spairs ps bs p*′)
hence *Inr j* ∈ *set ps* **by** (*simp add*: *add-spairs-def set-merge-wrt Inr-not-in-new-spairs*)
hence *Inr j* ∈ *set* (*p* # *ps*) **by** *simp*
let *?P* = λ*q*. *sig-of-pair q* = *term-of-pair* (*0*, *j*)
have *filter ?P* (*add-spairs ps bs p*′) = *filter ?P ps* **unfolding** *add-spairs-def*
proof (*rule filter-merge-wrt-2*)
fix *q*
assume *q* ∈ *set* (*new-spairs bs p*′)
then obtain *b* **where** *b* ∈ *set bs* **and** *is-regular-spair p*′ *b* **and** *q* = *Inl* (*p*′, *b*)
by (*rule in-new-spairsE*)
moreover assume *sig-of-pair q* = *term-of-pair* (*0*, *j*)
ultimately have *lt* (*spair p*′ *b*) = *term-of-pair* (*0*, *j*)
by (*simp add*: *sig-of-spair del*: *sig-of-pair.simps*)
hence *eq*: *component-of-term* (*lt* (*spair p*′ *b*)) = *j* **by** (*simp add*: *component-of-term-of-pair*)
have *component-of-term* (*lt p*′) < *j*
proof (*rule ccontr*)
assume ¬ *component-of-term* (*lt p*′) < *j*
hence *component-of-term* (*term-of-pair* (*0*, *j*)) ≤ *component-of-term* (*lt p*′)
by (*simp add*: *component-of-term-of-pair*)
moreover have *pp-of-term* (*term-of-pair* (*0*, *j*)) ⪯ *pp-of-term* (*lt p*′)
by (*simp add*: *pp-of-term-of-pair zero-min*)
ultimately have *term-of-pair* (*0*, *j*) $\preceq_t$ *lt p*′ **using** *ord-termI* **by** *blast*
moreover have *lt p*′ $\prec_t$ *term-of-pair* (*0*, *j*) **unfolding** *lt-p*′ *sig-of-p*[*symmetric*]
using ‹*Inr j* ∈ *set ps*› **by** (*rule sig-of-p-less*)
ultimately show *False* **by** *simp*
qed
moreover have *component-of-term* (*lt b*) < *j*

proof (*rule ccontr*)
assume $\neg$ *component-of-term* (*lt b*) $<$ *j*
hence *component-of-term* (*term-of-pair* (*0*, *j*)) $\leq$ *component-of-term* (*lt b*)
by (*simp add*: *component-of-term-of-pair*)
moreover have *pp-of-term* (*term-of-pair* (*0*, *j*)) $\preceq$ *pp-of-term* (*lt b*)
by (*simp add*: *pp-of-term-of-pair zero-min*)
ultimately have *term-of-pair* (*0*, *j*) $\preceq_t$ *lt b* **using** *ord-termI* **by** *blast*
moreover from *assms*(*1*) ‹*Inr j* $\in$ *set* (*p* # *ps*)› ‹*b* $\in$ *set bs*›
have *lt b* $\prec_t$ *term-of-pair* (*0*, *j*) **by** (*rule rb-aux-inv-D4*)
ultimately show *False* **by** *simp*
qed
ultimately have *component-of-term* (*lt* (*spair* p' *b*)) $<$ *j*
using *is-regular-spair-component-lt-cases*[*OF* ‹*is-regular-spair* p' *b*›] **by** *auto*
thus *False* **by** (*simp add*: *eq*)
qed
hence *length* (*filter ?P* (*add-spairs ps bs* p')) $\leq$ *length* (*filter ?P* (*p* # *ps*))
by *simp*
also from *assms*(*1*) ‹*Inr j* $\in$ *set* (*p* # *ps*)› **have** ... $\leq$ *1* **by** (*rule rb-aux-inv-D4*)
finally show *length* (*filter ?P* (*add-spairs ps bs* p')) $\leq$ *1* **.**
next
from *assms*(*1*) **have** *sorted-wrt pair-ord* (*p* # *ps*) **by** (*rule rb-aux-inv-D5*)
hence *sorted-wrt pair-ord ps* **by** *simp*
thus *sorted-wrt pair-ord* (*add-spairs ps bs* p') **by** (*rule sorted-add-spairs*)
next
fix *q b1 b2*
assume *1*: *q* $\in$ *set* (*add-spairs ps bs* p') **and** *2*: *is-regular-spair b1 b2*
and *3*: *sig-of-pair q* $\prec_t$ *lt* (*spair b1 b2*)
assume *b1* $\in$ *set* (p' # *bs*) **and** *b2* $\in$ *set* (p' # *bs*)
hence *b1* = p' $\vee$ *b1* $\in$ *set bs* **and** *b2* = p' $\vee$ *b2* $\in$ *set bs* **by** *simp-all*
thus *Inl* (*b1*, *b2*) $\in$ *set* (*add-spairs ps bs* p') $\vee$ *Inl* (*b2*, *b1*) $\in$ *set* (*add-spairs ps bs* p')
proof (*elim disjE*)
assume *b1* = p' **and** *b2* = p'
with *2* **show** *?thesis* **by** (*simp add*: *is-regular-spair-def*)
next
assume *b1* = p' **and** *b2* $\in$ *set bs*
from *this*(*2*) *2* **have** *Inl* (*b1*, *b2*) $\in$ *set* (*new-spairs bs* p') **unfolding** ‹*b1* = p'›
by (*rule in-new-spairsI*)
with *2* **show** *?thesis* **by** (*simp add*: *sig-of-spair add-spairs-def set-merge-wrt image-Un del*: *sig-of-pair.simps*)
next
assume *b2* = p' **and** *b1* $\in$ *set bs*
note *this*(*2*)
moreover from *2* **have** *is-regular-spair b2 b1* **by** (*rule is-regular-spair-sym*)
ultimately have *Inl* (*b2*, *b1*) $\in$ *set* (*new-spairs bs* p') **unfolding** ‹*b2* = p'›
by (*rule in-new-spairsI*)
with *2* **show** *?thesis*
by (*simp add*: *sig-of-spair-commute sig-of-spair add-spairs-def set-merge-wrt*

image-Un del: *sig-of-pair.simps*)
    next
      note *assms*(*1*) ‹$p \in$ *set* (*p* # *ps*)›
      moreover assume *b1* $\in$ *set bs* **and** *b2* $\in$ *set bs*
      moreover note *2*
      moreover have *4*: *sig-of-pair p* $\prec_t$ *lt* (*spair b1 b2*)
        by (*rule ord-term-lin.le-less-trans*, *rule sig-merge*, *fact 1*, *fact 3*)
      ultimately show *?thesis*
      proof (*rule rb-aux-inv-D6-1*)
        assume *Inl* (*b1*, *b2*) $\in$ *set* (*p* # *ps*)
        with *4* **have** *Inl* (*b1*, *b2*) $\in$ *set ps*
        by (*auto simp*: *sig-of-spair* ‹*is-regular-spair b1 b2*› *simp del*: *sig-of-pair.simps*)
        thus *?thesis* **by** (*simp add*: *add-spairs-def set-merge-wrt*)
      next
        assume *Inl* (*b2*, *b1*) $\in$ *set* (*p* # *ps*)
        with *4* **have** *Inl* (*b2*, *b1*) $\in$ *set ps*
          by (*auto simp*: *sig-of-spair sig-of-spair-commute* ‹*is-regular-spair b1 b2*›
simp del: *sig-of-pair.simps*)
        thus *?thesis* **by** (*simp add*: *add-spairs-def set-merge-wrt*)
      qed
    qed
  next
    fix *q j*
    assume $j <$ *length fs*
    assume *q* $\in$ *set* (*add-spairs ps bs p'*)
    hence *sig-of-pair p* $\preceq_t$ *sig-of-pair q* **by** (*rule sig-merge*)
    also assume *sig-of-pair q* $\prec_t$ *term-of-pair* (*0*, *j*)
    finally have *1*: *sig-of-pair p* $\prec_t$ *term-of-pair* (*0*, *j*) **.**
    with *assms*(*1*) ‹$p \in$ *set* (*p* # *ps*)› ‹$j <$ *length fs*› **have** *Inr j* $\in$ *set* (*p* # *ps*)
      by (*rule rb-aux-inv-D6-2*)
    with *1* **show** *Inr j* $\in$ *set* (*add-spairs ps bs p'*) **by** (*auto simp*: *add-spairs-def*
set-merge-wrt)
  next
    fix *b q*
    assume *b* $\in$ *set* (*p'* # *bs*) **and** *q-in*: *q* $\in$ *set* (*add-spairs ps bs p'*)
    from *this*(*1*) **have** $b = p' \vee b \in$ *set bs* **by** *simp*
    hence *lt b* $\preceq_t$ *lt p'*
    proof
      note *assms*(*1*)
      moreover assume *b* $\in$ *set bs*
      moreover have *p* $\in$ *set* (*p* # *ps*) **by** *simp*
      ultimately have *lt b* $\preceq_t$ *sig-of-pair p* **by** (*rule rb-aux-inv-D7*)
      thus *?thesis* **by** (*simp only*: *lt-p' sig-of-p*)
    qed *simp*
    also have ... = *sig-of-pair p* **by** (*simp only*: *sig-of-p lt-p'*)
    also from *q-in* **have** ... $\preceq_t$ *sig-of-pair q* **by** (*rule sig-merge*)
    finally show *lt b* $\preceq_t$ *sig-of-pair q* **.**
  next
    fix *a b*

assume *1*: $a \in set\ (p' \# bs)$ **and** *2*: $b \in set\ (p' \# bs)$ **and** *3*: *is-regular-spair a b*
assume *6*: $\neg$ *is-RB-in dgrad rword* (*set* (p' # *bs*)) (*lt* (*spair a b*))
with *rb* **have** *neq*: *lt* (*spair a b*) $\neq$ *lt* (*poly-of-pair p*) **by** (*auto simp*: *sig-of-p*)
assume *Inl* (*a*, *b*) $\notin$ *set* (*add-spairs ps bs p'*)
hence *40*: *Inl* (*a*, *b*) $\notin$ *set* (*new-spairs bs p'*) **and** *Inl* (*a*, *b*) $\notin$ *set ps*
by (*simp-all add*: *add-spairs-def set-merge-wrt*)
from *this*(*2*) *neq* **have** *4*: *Inl* (*a*, *b*) $\notin$ *set* (*p* # *ps*) **by** *auto*
assume *Inl* (*b*, *a*) $\notin$ *set* (*add-spairs ps bs p'*)
hence *50*: *Inl* (*b*, *a*) $\notin$ *set* (*new-spairs bs p'*) **and** *Inl* (*b*, *a*) $\notin$ *set ps*
by (*simp-all add*: *add-spairs-def set-merge-wrt*)
from *this*(*2*) *neq* **have** *5*: *Inl* (*b*, *a*) $\notin$ *set* (*p* # *ps*) **by** (*auto simp*: *spair-comm*[*of a*])
have $a \neq p'$
proof
assume $a = p'$
with *3* **have** $b \neq p'$ **by** (*auto simp*: *is-regular-spair-def*)
with *2* **have** $b \in set\ bs$ **by** *simp*
hence *Inl* (*a*, *b*) $\in$ *set* (*new-spairs bs p'*) **using** *3* **unfolding** ‹$a = p'$› **by** (*rule in-new-spairsI*)
with *40* **show** *False* **..**
qed
with *1* **have** $a \in set\ bs$ **by** *simp*
have $b \neq p'$
proof
assume $b = p'$
with *3* **have** $a \neq p'$ **by** (*auto simp*: *is-regular-spair-def*)
with *1* **have** $a \in set\ bs$ **by** *simp*
moreover from *3* **have** *is-regular-spair b a* **by** (*rule is-regular-spair-sym*)
ultimately have *Inl* (*b*, *a*) $\in$ *set* (*new-spairs bs p'*) **unfolding** ‹$b = p'$› **by** (*rule in-new-spairsI*)
with *50* **show** *False* **..**
qed
with *2* **have** $b \in set\ bs$ **by** *simp*
have *lt-sp*: *lt* (*spair a b*) $\prec_t$ *lt* p'
proof (*rule ord-term-lin.linorder-cases*)
assume *lt* (*spair a b*) = *lt* p'
with *neq* **show** *?thesis* **by** (*simp add*: *lt-p'*)
next
assume *lt* p' $\prec_t$ *lt* (*spair a b*)
hence *sig-of-pair p* $\prec_t$ *lt* (*spair a b*) **by** (*simp only*: *lt-p' sig-of-p*)
with *assms*(*1*) ‹$p \in set\ (p \# ps)$› ‹$a \in set\ bs$› ‹$b \in set\ bs$› *3* **show** *?thesis*
proof (*rule rb-aux-inv-D6-1*)
assume *Inl* (*a*, *b*) $\in$ *set* (*p* # *ps*)
with *4* **show** *?thesis* **..**
next
assume *Inl* (*b*, *a*) $\in$ *set* (*p* # *ps*)
with *5* **show** *?thesis* **..**
qed

qed
have ¬ *is-RB-in dgrad rword* (*set bs*) (*lt* (*spair a b*))
proof
assume *is-RB-in dgrad rword* (*set bs*) (*lt* (*spair a b*))
hence *is-RB-in dgrad rword* (*set* (p' # *bs*)) (*lt* (*spair a b*)) **unfolding** *set-simps* **using** *lt-sp*
by (*rule is-RB-in-insertI*)
with *6* **show** *False* **..**
qed
with *assms*(*1*) ‹*a* ∈ *set bs*› ‹*b* ∈ *set bs*› *3 4 5*
obtain *q* **where** *q* ∈ *set* (*p* # *ps*) **and** *8*: *sig-of-pair q* = *lt* (*spair a b*) **and** *9*: ¬ *sig-crit*′ *bs q*
by (*rule rb-aux-inv-D8*)
from *this*(*1*, *2*) *lt-sp* **have** *q* ∈ *set ps* **by** (*auto simp*: *lt-p*′ *sig-of-p*)
show ∃ *q*∈*set* (*add-spairs ps bs p*′). *sig-of-pair q* = *lt* (*spair a b*) ∧ ¬ *sig-crit*′ (*p*′ # *bs*) *q*
proof (*intro bexI conjI*)
show ¬ *sig-crit*′ (*p*′ # *bs*) *q*
proof
assume *sig-crit*′ (*p*′ # *bs*) *q*
moreover from *lt-sp* **have** *sig-of-pair q* $\prec_t$ *lt p*′ **by** (*simp only*: *8*)
ultimately have *sig-crit*′ *bs q* **by** (*rule sig-crit*′*-ConsD*)
with *9* **show** *False* **..**
qed
next
from ‹*q* ∈ *set ps*› **show** *q* ∈ *set* (*add-spairs ps bs p*′) **by** (*simp add*: *add-spairs-def set-merge-wrt*)
qed *fact*
next
fix *j*
assume *j* < *length fs*
assume *Inr j* ∉ *set* (*add-spairs ps bs p*′)
hence *Inr j* ∉ *set ps* **by** (*simp add*: *add-spairs-def set-merge-wrt*)

show *is-RB-in dgrad rword* (*set* (*p*′ # *bs*)) (*term-of-pair* (*0*, *j*))
proof (*cases term-of-pair* (*0*, *j*) = *sig-of-pair p*)
case *True*
with *rb* **show** *?thesis* **by** *simp*
next
case *False*
with ‹*Inr j* ∉ *set ps*› **have** *Inr j* ∉ *set* (*p* # *ps*) **by** *auto*
with *assms*(*1*) ‹*j* < *length fs*› **have** *rb*′: *is-RB-in dgrad rword* (*set bs*) (*term-of-pair* (*0*, *j*))
by (*rule rb-aux-inv-D9*)
have *term-of-pair* (*0*, *j*) $\prec_t$ *lt p*′
proof (*rule ord-term-lin.linorder-cases*)
assume *term-of-pair* (*0*, *j*) = *lt p*′
with *False* **show** *?thesis* **by** (*simp add*: *lt-p*′ *sig-of-p*)
next

assume *lt p′* $\prec_t$ *term-of-pair* (*0*, *j*)
hence *sig-of-pair p* $\prec_t$ *term-of-pair* (*0*, *j*) **by** (*simp only*: *lt-p′ sig-of-p*)
with *assms*(*1*) ‹*p* ∈ *set* (*p* # *ps*)› ‹*j* < *length fs*› **have** *Inr j* ∈ *set* (*p* # *ps*)
by (*rule rb-aux-inv-D6-2*)
with ‹*Inr j* ∉ *set* (*p* # *ps*)› **show** *?thesis* **..**
qed
with *rb′* **show** *?thesis* **unfolding** *set-simps* **by** (*rule is-RB-in-insertI*)
qed

show *rep-list* (*monomial 1* (*term-of-pair* (*0*, *j*))) ∈ *ideal* (*rep-list* ' *set* (*p′* # *bs*))
proof (*cases p* = *Inr j*)
case *True*
show *?thesis*
proof (*rule sig-red-zero-idealI*, *rule sig-red-zeroI*)
from *p-red* **have** (*sig-red* ($\prec_t$) ($\preceq$) (*set bs*))** (*monomial 1* (*term-of-pair* (*0*, *j*))) *p′*
by (*simp add*: *True*)
moreover have *set bs* ⊆ *set* (*p′* # *bs*) **by** *fastforce*
ultimately have (*sig-red* ($\prec_t$) ($\preceq$) (*set* (*p′* # *bs*)))** (*monomial 1* (*term-of-pair* (*0*, *j*))) *p′*
by (*rule sig-red-rtrancl-mono*)
hence (*sig-red* ($\preceq_t$) ($\preceq$) (*set* (*p′* # *bs*)))** (*monomial 1* (*term-of-pair* (*0*, *j*))) *p′*
by (*rule sig-red-rtrancl-sing-regI*)
also have *sig-red* ($\preceq_t$) ($\preceq$) (*set* (*p′* # *bs*)) *p′ 0* **unfolding** *sig-red-def*
proof (*intro exI bexI*)
from *assms*(*3*) **have** *rep-list p′* ≠ *0* **by** (*simp add*: *p′-def*)
show *sig-red-single* ($\preceq_t$) ($\preceq$) *p′ 0 p′ 0*
proof (*rule sig-red-singleI*)
show *rep-list p′* ≠ *0* **by** *fact*
next
from ‹*rep-list p′* ≠ *0*› **have** *punit.lt* (*rep-list p′*) ∈ *keys* (*rep-list p′*)
by (*rule punit.lt-in-keys*)
thus *0* + *punit.lt* (*rep-list p′*) ∈ *keys* (*rep-list p′*) **by** *simp*
next
from ‹*rep-list p′* ≠ *0*› **have** *punit.lc* (*rep-list p′*) ≠ *0* **by** (*rule punit.lc-not-0*)
thus *0* = *p′* − *monom-mult* (*lookup* (*rep-list p′*) (*0* + *punit.lt* (*rep-list p′*)) / *punit.lc* (*rep-list p′*)) *0 p′*
by (*simp add*: *punit.lc-def*[*symmetric*])
qed (*simp-all add*: *term-simps*)
qed *simp*
finally show (*sig-red* ($\preceq_t$) ($\preceq$) (*set* (*p′* # *bs*)))** (*monomial 1* (*term-of-pair* (*0*, *j*))) *0* **.**
qed (*fact rep-list-zero*)
next
case *False*
with ‹*Inr j* ∉ *set ps*› **have** *Inr j* ∉ *set* (*p* # *ps*) **by** *simp*

    with *assms*(*1*) ‹*j* < *length fs*›
    have *rep-list* (*monomial 1* (*term-of-pair* (*0*, *j*))) ∈ *ideal* (*rep-list* ‘ *set bs*)
      by (*rule rb-aux-inv-D9*)
      also have ... ⊆ *ideal* (*rep-list* ‘ *set* (*p′* # *bs*)) **by** (*rule ideal.span-mono*, *fastforce*)
    finally show *?thesis* **.**
   qed
  qed

  show *lt p′* ∉ *lt* ‘ *set bs* **unfolding** *lt-p′*
  proof
   assume *lt* (*poly-of-pair p*) ∈ *lt* ‘ *set bs*
   then obtain *b* **where** *b* ∈ *set bs* **and** *lt* (*poly-of-pair p*) = *lt b* **..**
   note *this*(*2*)
   also from ‹*b* ∈ *set bs*› **have** *lt b* $\prec_t$ *lt* (*poly-of-pair p*) **by** (*rule lt-p-gr*)
   finally show *False* **..**
  qed
qed

lemma *rb-aux-inv-init*: *rb-aux-inv* ([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*])
proof (*simp add*: *rb-aux-inv.simps rb-aux-inv1-def o-def*, *intro conjI ballI allI impI*)
  fix *v*
  assume *v* ∈ *set* (*Koszul-syz-sigs fs*)
  with *dgrad fs-distinct fs-nonzero* **show** *is-syz-sig dgrad v* **by** (*rule Koszul-syz-sigs-is-syz-sig*)
next
  fix *p q* :: $'t \Rightarrow_0 'b$
  show *Inl* (*p*, *q*) ∉ *Inr* ‘ {*0*..<*length fs*} **by** *blast*
next
  fix *j*
  assume *Inr j* ∈ *Inr* ‘ {*0*..<*length fs*}
  thus *j* < *length fs* **by** *fastforce*
next
  fix *j*
  have *eq*: (*term-of-pair* (*0*, *i*) = *term-of-pair* (*0*, *j*)) ⟷ (*j* = *i*) **for** *i*
   by (*auto dest*: *term-of-pair-injective*)
  show *length* (*filter* (λ*i*. *term-of-pair* (*0*, *i*) = *term-of-pair* (*0*, *j*)) [*0*..<*length fs*]) ≤ *Suc 0*
   by (*simp add*: *eq*)
next
  show *sorted-wrt pair-ord* (*map Inr* [*0*..<*length fs*])
  proof (*simp add*: *sorted-wrt-map pair-ord-def sorted-wrt-upt-iff*, *intro allI impI*)
   fix *i j* :: *nat*
   assume *i* < *j*
   hence *i* ≤ *j* **by** *simp*
   show *term-of-pair* (*0*, *i*) $\preceq_t$ *term-of-pair* (*0*, *j*) **by** (*rule ord-termI*, *simp-all add*: *term-simps* ‹*i* ≤ *j*›)
  qed
qed

corollary *rb-aux-inv-init-fst*:
  rb-aux-inv (*fst* (([], *Koszul-syz-sigs fs*, *map Inr* [*0..<length fs*]), *z*))
  using *rb-aux-inv-init* **by** *simp*

function (*domintros*) *rb-aux* :: ((($'t \Rightarrow_0 'b$) *list* × $'t$ *list* × ((($'t \Rightarrow_0 'b$) × ($'t \Rightarrow_0$ $'b$)) + *nat*) *list*) × *nat*) ⇒
                    ((($'t \Rightarrow_0 'b$) *list* × $'t$ *list* × ((($'t \Rightarrow_0 'b$) × ($'t \Rightarrow_0 'b$)) + *nat*) *list*) × *nat*)
  where
    rb-aux ((*bs*, *ss*, []), *z*) = ((*bs*, *ss*, []), *z*) |
    rb-aux ((*bs*, *ss*, *p* # *ps*), *z*) =
      (*let ss′* = *new-syz-sigs ss bs p in*
        if sig-crit bs ss′ p then
          rb-aux ((*bs*, *ss′*, *ps*), *z*)
        else
          let p′ = *sig-trd bs* (*poly-of-pair p*) *in*
            if rep-list p′ = *0 then*
              rb-aux ((*bs*, *lt p′* # *ss′*, *ps*), *Suc z*)
            else
              rb-aux ((*p′* # *bs*, *ss′*, *add-spairs ps bs p′*), *z*))
  by *pat-completeness auto*

definition *rb* :: ($'t \Rightarrow_0 'b$) *list* × *nat*
  where *rb* = (*let* ((*bs*, -, -), *z*) = *rb-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0..<length fs*]), *0*) *in* (*bs*, *z*))

rb is only an auxiliary function used for stating some theorems about rewrite bases and their computation in a readable way. Actual computations (of Gröbner bases) are performed by function *sig-gb*, defined below. The second return value of *rb* is the number of zero-reductions. It is only needed for proving that under certain assumptions, there are no such zero-reductions.

Termination

qualified definition *rb-aux-term1* ≡ {(*x*, *y*). ∃ *z*. *x* = *z* # *y*}

qualified definition *rb-aux-term2* ≡ {(*x*, *y*). (*fst x*, *fst y*) ∈ *rb-aux-term1* ∨
          (*fst x* = *fst y* ∧ *length* (*snd* (*snd x*)) < *length* (*snd* (*snd y*)))}

qualified definition *rb-aux-term* ≡ *rb-aux-term2* ∩ {(*x*, *y*). *rb-aux-inv x* ∧ *rb-aux-inv y*}

lemma *wfp-on-rb-aux-term1*: *wfp-on* (λ*x y*. (*x*, *y*) ∈ *rb-aux-term1*) (*Collect rb-aux-inv1*)
proof (*rule wfp-onI-chain*, *rule*, *elim exE*)
  fix *seq′*
  assume ∀ *i*. *seq′ i* ∈ *Collect rb-aux-inv1* ∧ (*seq′* (*Suc i*), *seq′ i*) ∈ *rb-aux-term1*
  hence *inv*: *rb-aux-inv1* (*seq′ j*) **and** *cons*: ∃ *b*. *seq′* (*Suc j*) = *b* # *seq′ j* **for** *j*
    by (*simp-all add*: *rb-aux-term1-def*)
  from *this*(*2*) **have** *1*: *thesis0* **if** ⋀*j*. *i* < *length* (*seq′ j*) ⟹ *thesis0* **for** *i thesis0*

using *that* **by** (*rule list-seq-indexE-length*)

define *seq* **where** *seq* = (λ*i. let j* = (*SOME k. i* < *length* (*seq′ k*)) *in rev* (*seq′ j*) ! *i*)
have *2*: *seq i* = *rev* (*seq′ j*) ! *i* **if** *i* < *length* (*seq′ j*) **for** *i j*
proof −
define *k* **where** *k* = (*SOME k. i* < *length* (*seq′ k*))
from *that* **have** *i* < *length* (*seq′ k*) **unfolding** *k-def* **by** (*rule someI*)
with *cons that* **have** *rev* (*seq′ k*) ! *i* = *rev* (*seq′ j*) ! *i* **by** (*rule list-seq-nth′*)
thus *?thesis* **by** (*simp add*: *seq-def k-def*[*symmetric*])
qed
have *3*: *seq i* ∈ *set* (*seq′ j*) **if** *i* < *length* (*seq′ j*) **for** *i j*
proof −
from *that* **have** *i* < *length* (*rev* (*seq′ j*)) **by** *simp*
moreover from *that* **have** *seq i* = *rev* (*seq′ j*) ! *i* **by** (*rule 2*)
ultimately have *seq i* ∈ *set* (*rev* (*seq′ j*)) **by** (*metis nth-mem*)
thus *?thesis* **by** *simp*
qed
have *4*: *seq* ‘ {*0..*<*i*} = *set* (*take i* (*rev* (*seq′ j*))) **if** *i* < *length* (*seq′ j*) **for** *i j*
proof −
from *refl* **have** *seq* ‘ {*0..*<*i*} = (!) (*rev* (*seq′ j*)) ‘ {*0..*<*i*}
proof (*rule image-cong*)
fix *i′*
assume *i′* ∈ {*0..*<*i*}
hence *i′* < *i* **by** *simp*
hence *i′* < *length* (*seq′ j*) **using** *that* **by** *simp*
thus *seq i′* = *rev* (*seq′ j*) ! *i′* **by** (*rule 2*)
qed
also have ... = *set* (*take i* (*rev* (*seq′ j*))) **by** (*rule nth-image*, *simp add*: *that less-imp-le-nat*)
finally show *?thesis* **.**
qed
from *dgrad* **show** *False*
proof (*rule rb-termination*)
have *seq i* ∈ *dgrad-sig-set dgrad* **for** *i*
proof −
obtain *j* **where** *i* < *length* (*seq′ j*) **by** (*rule 1*)
hence *seq i* ∈ *set* (*seq′ j*) **by** (*rule 3*)
moreover from *inv* **have** *set* (*seq′ j*) ⊆ *dgrad-sig-set dgrad* **by** (*rule rb-aux-inv1-D1*)
ultimately show *?thesis* **..**
qed
thus *range seq* ⊆ *dgrad-sig-set dgrad* **by** *blast*
next
have *rep-list* (*seq i*) ≠ *0* **for** *i*
proof −
obtain *j* **where** *i* < *length* (*seq′ j*) **by** (*rule 1*)
hence *seq i* ∈ *set* (*seq′ j*) **by** (*rule 3*)
moreover from *inv* **have** *0* ∉ *rep-list* ‘ *set* (*seq′ j*) **by** (*rule rb-aux-inv1-D2*)

ultimately show *?thesis* **by** *auto*
qed
thus $0 \notin$ *rep-list* ‘ *range seq* **by** *fastforce*
next
fix *i1 i2* :: *nat*
assume $i1 < i2$
also obtain *j* **where** *i2*: $i2 < length\ (seq'\ j)$ **by** (*rule 1*)
finally have *i1*: $i1 < length\ (seq'\ j)$ **.**
from *i1* **have** *s1*: $seq\ i1 = rev\ (seq'\ j)\ !\ i1$ **by** (*rule 2*)
from *i2* **have** *s2*: $seq\ i2 = rev\ (seq'\ j)\ !\ i2$ **by** (*rule 2*)
from *inv* **have** *sorted-wrt* $(\lambda x\ y.\ lt\ y \prec_t lt\ x)\ (seq'\ j)$ **by** (*rule rb-aux-inv1-D3*)
hence *sorted-wrt* $(\lambda x\ y.\ lt\ x \prec_t lt\ y)\ (rev\ (seq'\ j))$ **by** (*simp add*: *sorted-wrt-rev*)
moreover note ‹$i1 < i2$›
moreover from *i2* **have** $i2 < length\ (rev\ (seq'\ j))$ **by** *simp*
ultimately have $lt\ (rev\ (seq'\ j)\ !\ i1) \prec_t lt\ (rev\ (seq'\ j)\ !\ i2)$ **by** (*rule sorted-wrt-nth-less*)
thus $lt\ (seq\ i1) \prec_t lt\ (seq\ i2)$ **by** (*simp only*: *s1 s2*)
next
fix *i*
obtain *j* **where** *i*: $i < length\ (seq'\ j)$ **by** (*rule 1*)
hence *eq1*: $seq\ i = rev\ (seq'\ j)\ !\ i$ **and** *eq2*: *seq* ‘ $\{0..<i\} = set\ (take\ i\ (rev\ (seq'\ j)))$
by (*rule 2*, *rule 4*)
let $?i = length\ (seq'\ j) - Suc\ i$
from *i* **have** $?i < length\ (seq'\ j)$ **by** *simp*
with *inv* **have** $\neg$ *is-sig-red* $(\prec_t)\ (\preceq)\ (set\ (drop\ (Suc\ ?i)\ (seq'\ j)))\ ((seq'\ j)\ !\ ?i)$
by (*rule rb-aux-inv1-D4*)
thus $\neg$ *is-sig-red* $(\prec_t)\ (\preceq)\ (seq$ ‘ $\{0..<i\})\ (seq\ i)$
using *i* **by** (*simp add*: *eq1 eq2 rev-nth take-rev Suc-diff-Suc*)

from *inv* ‹$?i < length\ (seq'\ j)$›
show $(\exists j<length\ fs.\ lt\ (seq\ i) = lt\ (monomial\ (1::'b)\ (term\text{-}of\text{-}pair\ (0,\ j))) \wedge$
punit.lt $(rep\text{-}list\ (seq\ i)) \preceq$ *punit.lt* $(rep\text{-}list\ (monomial\ 1\ (term\text{-}of\text{-}pair\ (0,\ j))))) \vee$
$(\exists j\ k.$ *is-regular-spair* $(seq\ j)\ (seq\ k) \wedge rep\text{-}list\ (spair\ (seq\ j)\ (seq\ k)) \neq 0 \wedge$
$lt\ (seq\ i) = lt\ (spair\ (seq\ j)\ (seq\ k)) \wedge$
punit.lt $(rep\text{-}list\ (seq\ i)) \preceq$ *punit.lt* $(rep\text{-}list\ (spair\ (seq\ j)\ (seq\ k))))$ (**is** $?l \vee ?r$)
proof (*rule rb-aux-inv1-E*)
fix *j0*
assume $j0 < length\ fs$
and $lt\ (seq'\ j\ !\ (length\ (seq'\ j) - Suc\ i)) = lt\ (monomial\ (1::'b)\ (term\text{-}of\text{-}pair\ (0,\ j0)))$
and *punit.lt* $(rep\text{-}list\ (seq'\ j\ !\ (length\ (seq'\ j) - Suc\ i))) \preceq$
punit.lt $(rep\text{-}list\ (monomial\ 1\ (term\text{-}of\text{-}pair\ (0,\ j0))))$
hence *?l* **using** *i* **by** (*auto simp*: *eq1 eq2 rev-nth take-rev Suc-diff-Suc*)
thus *?thesis* **..**
next
fix *p q*

assume $p \in$ *set* (*seq'* j)
then obtain *pi* **where** *pi* $<$ *length* (*seq'* j) **and** $p =$ (*seq'* j) ! *pi* **by** (*metis in-set-conv-nth*)
hence *p*: $p =$ *seq* (*length* (*seq'* j) $-$ *Suc pi*)
by (*metis 2* ‹$p \in$ *set* (*seq'* j)› *diff-Suc-less length-pos-if-in-set length-rev rev-nth rev-rev-ident*)
assume $q \in$ *set* (*seq'* j)
then obtain *qi* **where** *qi* $<$ *length* (*seq'* j) **and** $q =$ (*seq'* j) ! *qi* **by** (*metis in-set-conv-nth*)
hence *q*: $q =$ *seq* (*length* (*seq'* j) $-$ *Suc qi*)
by (*metis 2* ‹$q \in$ *set* (*seq'* j)› *diff-Suc-less length-pos-if-in-set length-rev rev-nth rev-rev-ident*)
assume *is-regular-spair p q* **and** *rep-list* (*spair p q*) $\neq$ *0*
and *lt* (*seq'* j ! (*length* (*seq'* j) $-$ *Suc i*)) $=$ *lt* (*spair p q*)
and *punit.lt* (*rep-list* (*seq'* j ! (*length* (*seq'* j) $-$ *Suc i*))) $\preceq$ *punit.lt* (*rep-list* (*spair p q*))
hence *?r* **using** *i* **by** (*auto simp*: *eq1 eq2 p q rev-nth take-rev Suc-diff-Suc*)
thus *?thesis* **..**
qed

from *inv* ‹*?i* $<$ *length* (*seq'* j)›
have *is-RB-upt dgrad rword* (*set* (*drop* (*Suc ?i*) (*seq'* j))) (*lt* ((*seq'* j) ! *?i*))
by (*rule rb-aux-inv1-D5*)
with *dgrad* **have** *is-sig-GB-upt dgrad* (*set* (*drop* (*Suc ?i*) (*seq'* j))) (*lt* ((*seq'* j) ! *?i*))
by (*rule is-RB-upt-is-sig-GB-upt*)
thus *is-sig-GB-upt dgrad* (*seq* ‘ {*0..*$<$*i*}) (*lt* (*seq i*))
using *i* **by** (*simp add*: *eq1 eq2 rev-nth take-rev Suc-diff-Suc*)
qed
qed

lemma *wfp-on-rb-aux-term2*: *wfp-on* ($\lambda x\, y.$ (x, y) $\in$ *rb-aux-term2*) (*Collect rb-aux-inv*)
proof (*rule wfp-onI-min*)
fix x Q
assume $x \in Q$ **and** *Q-sub*: $Q \subseteq$ *Collect rb-aux-inv*
from *this*(*1*) **have** *fst* $x \in$ *fst* ‘ Q **by** (*rule imageI*)
have *fst* ‘ $Q \subseteq$ *Collect rb-aux-inv1*
proof
fix y
assume $y \in$ *fst* ‘ Q
then obtain z **where** $z \in Q$ **and** *y*: $y =$ *fst* z **by** *fastforce*
obtain *bs ss ps* **where** *z*: $z =$ (*bs*, *ss*, *ps*) **by** (*rule rb-aux-inv.cases*)
from ‹$z \in Q$› *Q-sub* **have** *rb-aux-inv* z **by** *blast*
thus $y \in$ *Collect rb-aux-inv1* **by** (*simp add*: *y z rb-aux-inv.simps*)
qed
with *wfp-on-rb-aux-term1* ‹*fst* $x \in$ *fst* ‘ Q› **obtain** z' **where** $z' \in$ *fst* ‘ Q
and *z'-min*: $\bigwedge y.$ (y, z') $\in$ *rb-aux-term1* $\Longrightarrow y \notin$ *fst* ‘ Q **by** (*rule wfp-onE-min*) *blast*
from *this*(*1*) **obtain** *z0* **where** *z0* $\in Q$ **and** *z'*: $z' =$ *fst z0* **by** *fastforce*

define *Q0* **where** *Q0* = {*z*. *z* ∈ *Q* ∧ *fst z* = *fst z0*}
from ‹*z0* ∈ *Q*› **have** *z0* ∈ *Q0* **by** (*simp add*: *Q0-def*)
hence *length* (*snd* (*snd z0*)) ∈ *length* ' *snd* ' *snd* ' *Q0* **by** (*intro imageI*)
with *wf-less* **obtain** *n* **where** *n1*: *n* ∈ *length* ' *snd* ' *snd* ' *Q0*
and *n2*: ⋀*n*′. *n*′ < *n* ⟹ *n*′ ∉ *length* ' *snd* ' *snd* ' *Q0* **by** (*rule wfE-min*, *blast*)
from *n1* **obtain** *z* **where** *z* ∈ *Q0* **and** *n3*: *n* = *length* (*snd* (*snd z*)) **by** *fastforce*
have *z-min*: *y* ∉ *Q0* **if** *length* (*snd* (*snd y*)) < *length* (*snd* (*snd z*)) **for** *y*
proof
assume *y* ∈ *Q0*
hence *length* (*snd* (*snd y*)) ∈ *length* ' *snd* ' *snd* ' *Q0* **by** (*intro imageI*)
with *n2* **have** ¬ *length* (*snd* (*snd y*)) < *length* (*snd* (*snd z*)) **unfolding** *n3*[*symmetric*] **by** *blast*
thus *False* **using** *that* **..**
qed
show ∃ *z*∈*Q*. ∀ *y*∈*Collect rb-aux-inv*. (*y*, *z*) ∈ *rb-aux-term2* ⟶ *y* ∉ *Q*
proof (*intro bexI ballI impI*)
fix *y*
assume *y* ∈ *Collect rb-aux-inv*
assume (*y*, *z*) ∈ *rb-aux-term2*
hence (*fst y*, *fst z*) ∈ *rb-aux-term1* ∨ (*fst y* = *fst z* ∧ *length* (*snd* (*snd y*)) < *length* (*snd* (*snd z*)))
by (*simp add*: *rb-aux-term2-def*)
thus *y* ∉ *Q*
proof
assume (*fst y*, *fst z*) ∈ *rb-aux-term1*
moreover from ‹*z* ∈ *Q0*› **have** *fst z* = *fst z0* **by** (*simp add*: *Q0-def*)
ultimately have (*fst y*, *z*′) ∈ *rb-aux-term1* **by** (*simp add*: *rb-aux-term1-def* *z*′)
hence *fst y* ∉ *fst* ' *Q* **by** (*rule z*′*-min*)
thus *?thesis* **by** *blast*
next
assume *fst y* = *fst z* ∧ *length* (*snd* (*snd y*)) < *length* (*snd* (*snd z*))
hence *fst y* = *fst z* **and** *length* (*snd* (*snd y*)) < *length* (*snd* (*snd z*)) **by** *simp-all*
from *this*(*2*) **have** *y* ∉ *Q0* **by** (*rule z-min*)
moreover from ‹*z* ∈ *Q0*› **have** *fst y* = *fst z0* **by** (*simp add*: *Q0-def* ‹*fst y* = *fst z*›)
ultimately show *?thesis* **by** (*simp add*: *Q0-def*)
qed
next
from ‹*z* ∈ *Q0*› **show** *z* ∈ *Q* **by** (*simp add*: *Q0-def*)
qed
qed

corollary *wf-rb-aux-term*: *wf rb-aux-term*
proof (*rule wfI-min*)
fix *x*::(′*t* $\Rightarrow_0$ ′*b*) *list* × ′*t list* × (((′*t* $\Rightarrow_0$ ′*b*) × (′*t* $\Rightarrow_0$ ′*b*)) + *nat*) *list* **and** *Q*
assume *x* ∈ *Q*
show ∃ *z*∈*Q*. ∀ *y*. (*y*, *z*) ∈ *rb-aux-term* ⟶ *y* ∉ *Q*

```
  proof (cases rb-aux-inv x)
    case True
    let ?Q = Q ∩ Collect rb-aux-inv
    note wfp-on-rb-aux-term2
    moreover from ‹x ∈ Q› True have x ∈ ?Q by simp
    moreover have ?Q ⊆ Collect rb-aux-inv by simp
     ultimately obtain z where z ∈ ?Q and z-min: ⋀y. (y, z) ∈ rb-aux-term2
⟹ y ∉ ?Q
      by (rule wfp-onE-min) blast
    show ?thesis
    proof (intro bexI allI impI)
      fix y
      assume (y, z) ∈ rb-aux-term
     hence (y, z) ∈ rb-aux-term2 and rb-aux-inv y by (simp-all add: rb-aux-term-def)
      from this(1) have y ∉ ?Q by (rule z-min)
      with ‹rb-aux-inv y› show y ∉ Q by simp
    next
      from ‹z ∈ ?Q› show z ∈ Q by simp
    qed
  next
    case False
    show ?thesis
    proof (intro bexI allI impI)
      fix y
      assume (y, x) ∈ rb-aux-term
      hence rb-aux-inv x by (simp add: rb-aux-term-def)
      with False show y ∉ Q ..
    qed fact
  qed
qed

lemma rb-aux-domI:
  assumes rb-aux-inv (fst args)
  shows rb-aux-dom args
proof −
  let ?rel = rb-aux-term <*lex*> ({}::(nat × nat) set)
  from wf-rb-aux-term wf-empty have wf ?rel ..
  thus ?thesis using assms
  proof (induct args)
    case (less args)
     obtain bs ss ps0 z where args: args = ((bs, ss, ps0), z) using prod.exhaust
by metis
    show ?case
    proof (cases ps0)
      case Nil
      show ?thesis unfolding args Nil by (rule rb-aux.domintros)
    next
      case (Cons p ps)
     from less(1) have 1: ⋀y. (y, ((bs, ss, p # ps), z)) ∈ ?rel ⟹ rb-aux-inv (fst
```

y) $\Longrightarrow$ *rb-aux-dom y*
by (*simp only*: *args Cons*)
from *less*(*2*) **have** *2*: *rb-aux-inv* (*bs*, *ss*, *p* # *ps*) **by** (*simp only*: *args Cons fst-conv*)
show *?thesis* **unfolding** *args Cons*
proof (*rule rb-aux.domintros*)
assume *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
with *2* **have** *a*: *rb-aux-inv* (*bs*, (*new-syz-sigs ss bs p*), *ps*) **by** (*rule rb-aux-inv-preserved-1*)
with *2* **have** ((*bs*, (*new-syz-sigs ss bs p*), *ps*), *bs*, *ss*, *p* # *ps*) ∈ *rb-aux-term*
by (*simp add*: *rb-aux-term-def rb-aux-term2-def*)
hence (((*bs*, (*new-syz-sigs ss bs p*), *ps*), *z*), (*bs*, *ss*, *p* # *ps*), *z*) ∈ *?rel* **by** *simp*
moreover from *a* **have** *rb-aux-inv* (*fst* ((*bs*, (*new-syz-sigs ss bs p*), *ps*), *z*))
by (*simp only*: *fst-conv*)
ultimately show *rb-aux-dom* ((*bs*, (*new-syz-sigs ss bs p*), *ps*), *z*) **by** (*rule 1*)
next
assume *rep-list* (*sig-trd bs* (*poly-of-pair p*)) = *0*
with *2* **have** *a*: *rb-aux-inv* (*bs*, *lt* (*sig-trd bs* (*poly-of-pair p*)) # *new-syz-sigs ss bs p*, *ps*)
by (*rule rb-aux-inv-preserved-2*)
with *2* **have** ((*bs*, *lt* (*sig-trd bs* (*poly-of-pair p*)) # *new-syz-sigs ss bs p*, *ps*), *bs*, *ss*, *p* # *ps*) ∈
rb-aux-term
by (*simp add*: *rb-aux-term-def rb-aux-term2-def*)
hence (((*bs*, *lt* (*sig-trd bs* (*poly-of-pair p*)) # *new-syz-sigs ss bs p*, *ps*), *Suc z*), (*bs*, *ss*, *p* # *ps*), *z*) ∈
?rel **by** *simp*
moreover from *a* **have** *rb-aux-inv* (*fst* ((*bs*, *lt* (*sig-trd bs* (*poly-of-pair p*)) # *new-syz-sigs ss bs p*, *ps*), *Suc z*))
by (*simp only*: *fst-conv*)
ultimately show *rb-aux-dom* ((*bs*, *lt* (*sig-trd bs* (*poly-of-pair p*)) # *new-syz-sigs ss bs p*, *ps*), *Suc z*)
by (*rule 1*)
next
let *?args* = (*sig-trd bs* (*poly-of-pair p*) # *bs*, *new-syz-sigs ss bs p*, *add-spairs ps bs* (*sig-trd bs* (*poly-of-pair p*)))
assume ¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p* **and** *rep-list* (*sig-trd bs* (*poly-of-pair p*)) ≠ *0*
with *2* **have** *a*: *rb-aux-inv ?args* **by** (*rule rb-aux-inv-preserved-3*)
with *2* **have** (*?args*, *bs*, *ss*, *p* # *ps*) ∈ *rb-aux-term*
by (*simp add*: *rb-aux-term-def rb-aux-term2-def rb-aux-term1-def*)
hence ((*?args*, *z*), (*bs*, *ss*, *p* # *ps*), *z*) ∈ *?rel* **by** *simp*
moreover from *a* **have** *rb-aux-inv* (*fst* (*?args*, *z*)) **by** (*simp only*: *fst-conv*)
ultimately show *rb-aux-dom* (*?args*, *z*) **by** (*rule 1*)
qed
qed
qed

qed

Invariant

lemma *rb-aux-inv-invariant*:
  assumes *rb-aux-inv* (*fst args*)
  shows *rb-aux-inv* (*fst* (*rb-aux args*))
proof −
  from *assms* **have** *rb-aux-dom args* **by** (*rule rb-aux-domI*)
  thus *?thesis* **using** *assms*
  proof (*induct args rule*: *rb-aux.pinduct*)
    case (*1 bs ss z*)
    thus *?case* **by** (*simp only*: *rb-aux.psimps*(*1*))
  next
    case (*2 bs ss p ps z*)
    from *2*(*5*) **have** ∗: *rb-aux-inv* (*bs*, *ss*, *p* # *ps*) **by** (*simp only*: *fst-conv*)
    show *?case*
    proof (*simp add*: *rb-aux.psimps*(*2*)[*OF 2*(*1*)] *Let-def*, *intro conjI impI*)
      assume *a*: *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
      with ∗ **have** *rb-aux-inv* (*bs*, *new-syz-sigs ss bs p*, *ps*)
        by (*rule rb-aux-inv-preserved-1*)
        hence *rb-aux-inv* (*fst* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*)) **by** (*simp only*: *fst-conv*)
      with *refl a* **show** *rb-aux-inv* (*fst* (*rb-aux* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*))) **by** (*rule 2*(*2*))
      thus *rb-aux-inv* (*fst* (*rb-aux* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*))) **.**
    next
      assume *a*: ¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
      assume *b*: *rep-list* (*sig-trd bs* (*poly-of-pair p*)) = *0*
      with ∗ **have** *rb-aux-inv* (*bs*, *lt* (*sig-trd bs* (*poly-of-pair p*)) # *new-syz-sigs ss bs p*, *ps*)
        by (*rule rb-aux-inv-preserved-2*)
      hence *rb-aux-inv* (*fst* ((*bs*, *lt* (*sig-trd bs* (*poly-of-pair p*)) # *new-syz-sigs ss bs p*, *ps*), *Suc z*))
        by (*simp only*: *fst-conv*)
      with *refl a refl b*
      show *rb-aux-inv* (*fst* (*rb-aux* ((*bs*, *lt* (*sig-trd bs* (*poly-of-pair p*)) # *new-syz-sigs ss bs p*, *ps*), *Suc z*)))
        by (*rule 2*(*3*))
    next
      let *?args* = (*sig-trd bs* (*poly-of-pair p*) # *bs*, *new-syz-sigs ss bs p*,
                  add-spairs ps bs (*sig-trd bs* (*poly-of-pair p*)))
      assume *a*: ¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p* **and** *b*: *rep-list* (*sig-trd bs* (*poly-of-pair p*)) ≠ *0*
      with ∗ **have** *rb-aux-inv ?args* **by** (*rule rb-aux-inv-preserved-3*)
      hence *rb-aux-inv* (*fst* (*?args*, *z*)) **by** (*simp only*: *fst-conv*)
      with *refl a refl b*
      show *rb-aux-inv* (*fst* (*rb-aux* (*?args*, *z*)))
        by (*rule 2*(*4*))
    qed

qed
qed

lemma *rb-aux-inv-last-Nil*:
assumes *rb-aux-dom args*
shows *snd* (*snd* (*fst* (*rb-aux args*))) = []
using *assms*
proof (*induct args rule*: *rb-aux.pinduct*)
case (*1 bs ss z*)
thus *?case* **by** (*simp add*: *rb-aux.psimps*(*1*))
next
case (*2 bs ss p ps z*)
show *?case*
proof (*simp add*: *rb-aux.psimps*(*2*)[*OF 2*(*1*)] *Let-def*, *intro conjI impI*)
assume *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
with *refl* **show** *snd* (*snd* (*fst* (*rb-aux* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*)))) = []
and *snd* (*snd* (*fst* (*rb-aux* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*)))) = []
by (*rule 2*(*2*))+
next
assume *a*: ¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p* **and** *b*: *rep-list* (*sig-trd bs* (*poly-of-pair p*)) = *0*
from *refl a refl b*
show *snd* (*snd* (*fst* (*rb-aux* ((*bs*, *lt* (*sig-trd bs* (*poly-of-pair p*)) # *new-syz-sigs ss bs p*, *ps*), *Suc z*)))) = []
by (*rule 2*(*3*))
next
assume *a*: ¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p* **and** *b*: *rep-list* (*sig-trd bs* (*poly-of-pair p*)) ≠ *0*
from *refl a refl b*
show *snd* (*snd* (*fst* (*rb-aux* ((*sig-trd bs* (*poly-of-pair p*) # *bs*, *new-syz-sigs ss bs p*,
add-spairs ps bs (*sig-trd bs* (*poly-of-pair p*))), *z*)))) = []
by (*rule 2*(*4*))
qed
qed

corollary *rb-aux-shape*:
assumes *rb-aux-dom args*
obtains *bs ss z* **where** *rb-aux args* = ((*bs*, *ss*, []), *z*)
proof −
obtain *bs ss ps z* **where** *rb-aux args* = ((*bs*, *ss*, *ps*), *z*) **using** *prod.exhaust* **by** *metis*
moreover from *assms* **have** *snd* (*snd* (*fst* (*rb-aux args*))) = [] **by** (*rule rb-aux-inv-last-Nil*)
ultimately have *rb-aux args* = ((*bs*, *ss*, []), *z*) **by** *simp*
thus *?thesis* **..**
qed

lemma *rb-aux-is-RB-upt*:
is-RB-upt dgrad rword (*set* (*fst* (*fst* (*rb-aux* (([], *Koszul-syz-sigs fs*, *map Inr*

[*0* ..<*length fs*]), *z*))))) *u*
proof −
  let *?args* = (([], *Koszul-syz-sigs fs*, *map Inr* [*0* ..<*length fs*]), *z*)
  from *rb-aux-inv-init-fst* **have** *rb-aux-dom ?args* **by** (*rule rb-aux-domI*)
  then obtain *bs ss z′* **where** *eq*: *rb-aux ?args* = ((*bs*, *ss*, []), *z′*) **by** (*rule rb-aux-shape*)
  moreover from *rb-aux-inv-init-fst* **have** *rb-aux-inv* (*fst* (*rb-aux ?args*))
    by (*rule rb-aux-inv-invariant*)
  ultimately have *rb-aux-inv* (*bs*, *ss*, []) **by** *simp*
  have *is-RB-upt dgrad rword* (*set bs*) *u* **by** (*rule rb-aux-inv-is-RB-upt*, *fact*, *simp*)
  thus *?thesis* **by** (*simp add*: *eq*)
qed

corollary *rb-is-RB-upt*: *is-RB-upt dgrad rword* (*set* (*fst rb*)) *u*
  using *rb-aux-is-RB-upt*[*of 0 u*] **by** (*auto simp add*: *rb-def split*: *prod.split*)

corollary *rb-aux-is-sig-GB-upt*:
  is-sig-GB-upt dgrad (*set* (*fst* (*fst* (*rb-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0* ..<*length fs*]), *z*))))) *u*
  using *dgrad rb-aux-is-RB-upt* **by** (*rule is-RB-upt-is-sig-GB-upt*)

corollary *rb-aux-is-sig-GB-in*:
  is-sig-GB-in dgrad (*set* (*fst* (*fst* (*rb-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0* ..<*length fs*]), *z*))))) *u*
proof −
  let *?u* = *term-of-pair* (*pp-of-term u*, *Suc* (*component-of-term u*))
  have $u \prec_t$ *?u*
  proof (*rule ord-term-lin.le-neq-trans*)
    show $u \preceq_t$ *?u* **by** (*rule ord-termI*, *simp-all add*: *term-simps*)
  next
    show $u \neq$ *?u*
    proof
      assume *u* = *?u*
      hence *component-of-term u* = *component-of-term ?u* **by** *simp*
      thus *False* **by** (*simp add*: *term-simps*)
    qed
  qed
  with *rb-aux-is-sig-GB-upt* **show** *?thesis* **by** (*rule is-sig-GB-uptD2*)
qed

corollary *rb-aux-is-Groebner-basis*:
  assumes *hom-grading dgrad*
  shows *punit.is-Groebner-basis* (*set* (*map rep-list* (*fst* (*fst* (*rb-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0* ..<*length fs*]), *z*))))))
proof −
  let *?args* = (([], *Koszul-syz-sigs fs*, *map Inr* [*0* ..<*length fs*]), *z*)
  from *rb-aux-inv-init-fst* **have** *rb-aux-dom ?args* **by** (*rule rb-aux-domI*)
  then obtain *bs ss z′* **where** *eq*: *rb-aux ?args* = ((*bs*, *ss*, []), *z′*) **by** (*rule rb-aux-shape*)

moreover from *rb-aux-inv-init-fst* **have** *rb-aux-inv* (*fst* (*rb-aux ?args*))
by (*rule rb-aux-inv-invariant*)
ultimately have *rb-aux-inv* (*bs*, *ss*, []) **by** *simp*
hence *rb-aux-inv1 bs* **by** (*rule rb-aux-inv-D1*)
hence *set bs* ⊆ *dgrad-sig-set dgrad* **by** (*rule rb-aux-inv1-D1*)
hence *set* (*fst* (*fst* (*rb-aux ?args*))) ⊆ *dgrad-max-set dgrad* **by** (*simp add*: *eq dgrad-sig-set′-def*)
with *dgrad assms* **have** *punit.is-Groebner-basis* (*rep-list* ‘ *set* (*fst* (*fst* (*rb-aux ?args*))))
using *rb-aux-is-sig-GB-in* **by** (*rule is-sig-GB-is-Groebner-basis*)
thus *?thesis* **by** *simp*
qed

lemma *ideal-rb-aux*:
ideal (*set* (*map rep-list* (*fst* (*fst* (*rb-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)))))) =
ideal (*set fs*) (**is** *ideal ?l* = *ideal ?r*)
proof
show *ideal ?l* ⊆ *ideal ?r* **by** (*rule ideal.span-subset-spanI*, *auto simp*: *rep-list-in-ideal*)
next
show *ideal ?r* ⊆ *ideal ?l*
proof (*rule ideal.span-subset-spanI*, *rule subsetI*)
fix *f*
assume *f* ∈ *set fs*
then obtain *j* **where** *j* < *length fs* **and** *f*: *f* = *fs* ! *j* **by** (*metis in-set-conv-nth*)
let *?args* = (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)
from *rb-aux-inv-init-fst* **have** *rb-aux-dom ?args* **by** (*rule rb-aux-domI*)
then obtain *bs ss z′* **where** *eq*: *rb-aux ?args* = ((*bs*, *ss*, []), *z′*) **by** (*rule rb-aux-shape*)
moreover from *rb-aux-inv-init-fst* **have** *rb-aux-inv* (*fst* (*rb-aux ?args*))
by (*rule rb-aux-inv-invariant*)
ultimately have *rb-aux-inv* (*bs*, *ss*, []) **by** *simp*
moreover note ‹*j* < *length fs*›
moreover have *Inr j* ∉ *set* [] **by** *simp*
ultimately have *rep-list* (*monomial 1* (*term-of-pair* (*0*, *j*))) ∈ *ideal ?l*
unfolding *eq set-map fst-conv* **by** (*rule rb-aux-inv-D9*)
thus *f* ∈ *ideal ?l* **by** (*simp add*: *rep-list-monomial′* ‹*j* < *length fs*› *f*)
qed
qed

corollary *ideal-rb*: *ideal* (*rep-list* ‘ *set* (*fst rb*)) = *ideal* (*set fs*)
proof −
have *ideal* (*rep-list* ‘ *set* (*fst rb*)) =
ideal (*set* (*map rep-list* (*fst* (*fst* (*rb-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *0*))))))
by (*auto simp*: *rb-def split*: *prod.splits*)
also have ... = *ideal* (*set fs*) **by** (*fact ideal-rb-aux*)
finally show *?thesis* **.**
qed

lemma
shows *dgrad-max-set-closed-rb-aux*:
set (*map rep-list* (*fst* (*fst* (*rb-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*))))) $\subseteq$
punit-dgrad-max-set dgrad (**is** *?thesis1*)
and *rb-aux-nonzero*:
0 $\notin$ *set* (*map rep-list* (*fst* (*fst* (*rb-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)))))
(**is** *?thesis2*)
proof −
let *?args* = (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)
from *rb-aux-inv-init-fst* **have** *rb-aux-dom ?args* **by** (*rule rb-aux-domI*)
then obtain *bs ss z'* **where** *eq*: *rb-aux ?args* = ((*bs*, *ss*, []), *z'*) **by** (*rule rb-aux-shape*)
moreover from *rb-aux-inv-init-fst* **have** *rb-aux-inv* (*fst* (*rb-aux ?args*))
by (*rule rb-aux-inv-invariant*)
ultimately have *rb-aux-inv* (*bs*, *ss*, []) **by** *simp*
hence *rb-aux-inv1 bs* **by** (*rule rb-aux-inv-D1*)
hence *set bs* $\subseteq$ *dgrad-sig-set dgrad* **and** $*$: *0* $\notin$ *rep-list* ‘ *set bs*
by (*rule rb-aux-inv1-D1*, *rule rb-aux-inv1-D2*)
from *this*(*1*) **have** *set bs* $\subseteq$ *dgrad-max-set dgrad* **by** (*simp add*: *dgrad-sig-set'-def*)
with *dgrad* **show** *?thesis1* **by** (*simp add*: *eq dgrad-max-3*)
from $*$ **show** *?thesis2* **by** (*simp add*: *eq*)
qed

4.2.11 Minimality of the Computed Basis

lemma *rb-aux-top-irred'*:
assumes *rword-strict* = *rw-rat-strict* **and** *rb-aux-inv* (*bs*, *ss*, *p* # *ps*)
and ¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
shows ¬ *is-sig-red* ($\preceq_t$) (=) (*set bs*) (*sig-trd bs* (*poly-of-pair p*))
proof −
have *rword* = *rw-rat* **by** (*intro ext*, *simp only*: *rword-def rw-rat-alt*, *simp add*: *assms*(*1*))

have *lt-p*: *sig-of-pair p* = *lt* (*poly-of-pair p*) **by** (*rule pair-list-sig-of-pair*, *fact*, *simp*)
define *p'* **where** *p'* = *sig-trd bs* (*poly-of-pair p*)
have *red-p*: (*sig-red* ($\prec_t$) ($\preceq$) (*set bs*))** (*poly-of-pair p*) *p'*
unfolding *p'-def* **by** (*rule sig-trd-red-rtrancl*)
hence *lt-p'*: *lt p'* = *sig-of-pair p*
and *lt-p''*: *punit.lt* (*rep-list p'*) $\preceq$ *punit.lt* (*rep-list* (*poly-of-pair p*))
unfolding *lt-p* **by** (*rule sig-red-regular-rtrancl-lt*, *rule sig-red-rtrancl-lt-rep-list*)
have ¬ *is-sig-red* (=) (=) (*set bs*) *p'*
proof
assume *is-sig-red* (=) (=) (*set bs*) *p'*
then obtain *b* **where** *b* $\in$ *set bs* **and** *rep-list b* $\neq$ *0* **and** *rep-list p'* $\neq$ *0*
and *1*: *punit.lt* (*rep-list b*) *adds punit.lt* (*rep-list p'*)

and *2*: *punit.lt* (*rep-list* p') $\oplus$ *lt* b = *punit.lt* (*rep-list* b) $\oplus$ *lt* p'
by (*rule is-sig-red-top-addsE*)
note *this*(*3*)
moreover from *red-p* **have** (*punit.red* (*rep-list* ' *set bs*))** (*rep-list* (*poly-of-pair* *p*)) (*rep-list* p')
by (*rule sig-red-red-rtrancl*)
ultimately have *rep-list* (*poly-of-pair p*) $\neq$ *0* **by** (*auto simp*: *punit.rtrancl-0*)

define *x* **where** x = *punit.lt* (*rep-list* p') $-$ *punit.lt* (*rep-list* b)
from *1 2* **have** *x1*: $x \oplus$ *lt* b = *lt* p' **by** (*simp add*: *term-is-le-rel-minus x-def*)
from *this*[*symmetric*] **have** *lt b* $adds_t$ *sig-of-pair p* **unfolding** *lt-*p' **by** (*rule adds-termI*)
from *1* **have** *x2*: x + *punit.lt* (*rep-list* b) = *punit.lt* (*rep-list* p') **by** (*simp add*: *x-def adds-minus*)
from ‹*rep-list b* $\neq$ *0*› **have** $b \neq$ *0* **by** (*auto simp*: *rep-list-zero*)

show *False*
proof (*rule sum-prodE*)
fix *a0 b0*
assume *p*: *p* = *Inl* (*a0*, *b0*)
hence *Inl* (*a0*, *b0*) $\in$ *set* (*p* # *ps*) **by** *simp*
with *assms*(*2*) **have** *reg*: *is-regular-spair a0 b0* **and** *a0* $\in$ *set bs* **and** *b0* $\in$ *set bs*
by (*rule rb-aux-inv-D3*)+
from *assms*(*2*) **have** *inv1*: *rb-aux-inv1 bs* **by** (*rule rb-aux-inv-D1*)
hence *0* $\notin$ *rep-list* ' *set bs* **by** (*rule rb-aux-inv1-D2*)
with ‹*a0* $\in$ *set bs*› ‹*b0* $\in$ *set bs*› **have** *rep-list a0* $\neq$ *0* **and** *rep-list b0* $\neq$ *0* **by** *fastforce*+
hence *a0* $\neq$ *0* **and** *b0* $\neq$ *0* **by** (*auto simp*: *rep-list-zero*)

let *?t1* = *punit.lt* (*rep-list a0*)
let *?t2* = *punit.lt* (*rep-list b0*)
let *?l* = *lcs ?t1 ?t2*
from ‹*rep-list* (*poly-of-pair p*) $\neq$ *0*› **have** *punit.spoly* (*rep-list a0*) (*rep-list b0*) $\neq$ *0*
by (*simp add*: *p rep-list-spair*)
with ‹*rep-list a0* $\neq$ *0*› ‹*rep-list b0* $\neq$ *0*›
have *punit.lt* (*punit.spoly* (*rep-list a0*) (*rep-list b0*)) $\prec$ *?l*
by (*rule punit.lt-spoly-less-lcs*[*simplified*])

obtain b' **where** *3*: *is-canon-rewriter rword* (*set bs*) (*sig-of-pair p*) b'
and *4*: *punit.lt* (*rep-list* (*poly-of-pair p*)) $\prec$
(*pp-of-term* (*sig-of-pair p*) $-$ *lp* b') + *punit.lt* (*rep-list* b')
proof (*cases* (*?l* $-$ *?t1*) $\oplus$ *lt a0* $\preceq_t$ (*?l* $-$ *?t2*) $\oplus$ *lt b0*)
case *True*
have *sig-of-pair p* = *lt* (*spair a0 b0*) **unfolding** *lt-p* **by** (*simp add*: *p*)
also from *reg* **have** ... = (*?l* $-$ *?t2*) $\oplus$ *lt b0*
by (*simp add*: *True is-regular-spair-lt ord-term-lin.max-def*)
finally have *eq1*: *sig-of-pair p* = (*?l* $-$ *?t2*) $\oplus$ *lt b0* **.**

hence *lt b0* $adds_t$ *sig-of-pair p* **by** (*rule adds-termI*)
moreover from *assms*(*3*) **have** ¬ *is-rewritable bs b0* ((*?l* − *?t2*) ⊕ *lt b0*)
by (*simp add*: *p spair-sigs-def Let-def*)
ultimately have *is-canon-rewriter rword* (*set bs*) (*sig-of-pair p*) *b0*
unfolding *eq1*[*symmetric*] **using** *inv1* ‹*b0* ∈ *set bs*› ‹*b0* ≠ *0*› *is-rewritableI-is-canon-rewriter*
by *blast*
thus *?thesis*
proof
have *punit.lt* (*rep-list* (*poly-of-pair p*)) = *punit.lt* (*punit.spoly* (*rep-list a0*) (*rep-list b0*))
by (*simp add*: *p rep-list-spair*)
also have ... ≺ *?l* **by** *fact*
also have ... = (*?l* − *?t2*) + *?t2* **by** (*simp only*: *adds-minus adds-lcs-2*)
also have ... = (*pp-of-term* (*sig-of-pair p*) − *lp b0*) + *?t2*
by (*simp only*: *eq1 pp-of-term-splus add-diff-cancel-right'*)
finally show *punit.lt* (*rep-list* (*poly-of-pair p*)) ≺ *pp-of-term* (*sig-of-pair p*) − *lp b0* + *?t2* **.**
qed
next
case *False*
have *sig-of-pair p* = *lt* (*spair a0 b0*) **unfolding** *lt-p* **by** (*simp add*: *p*)
also from *reg* **have** ... = (*?l* − *?t1*) ⊕ *lt a0*
by (*simp add*: *False is-regular-spair-lt ord-term-lin.max-def*)
finally have *eq1*: *sig-of-pair p* = (*?l* − *?t1*) ⊕ *lt a0* **.**
hence *lt a0* $adds_t$ *sig-of-pair p* **by** (*rule adds-termI*)
moreover from *assms*(*3*) **have** ¬ *is-rewritable bs a0* ((*?l* − *?t1*) ⊕ *lt a0*)
by (*simp add*: *p spair-sigs-def Let-def*)
ultimately have *is-canon-rewriter rword* (*set bs*) (*sig-of-pair p*) *a0*
unfolding *eq1*[*symmetric*] **using** *inv1* ‹*a0* ∈ *set bs*› ‹*a0* ≠ *0*› *is-rewritableI-is-canon-rewriter*
by *blast*
thus *?thesis*
proof
have *punit.lt* (*rep-list* (*poly-of-pair p*)) = *punit.lt* (*punit.spoly* (*rep-list a0*) (*rep-list b0*))
by (*simp add*: *p rep-list-spair*)
also have ... ≺ *?l* **by** *fact*
also have ... = (*?l* − *?t1*) + *?t1* **by** (*simp only*: *adds-minus adds-lcs*)
also have ... = (*pp-of-term* (*sig-of-pair p*) − *lp a0*) + *?t1*
by (*simp only*: *eq1 pp-of-term-splus add-diff-cancel-right'*)
finally show *punit.lt* (*rep-list* (*poly-of-pair p*)) ≺ *pp-of-term* (*sig-of-pair p*) − *lp a0* + *?t1* **.**
qed
qed

define *y* **where** *y* = *pp-of-term* (*sig-of-pair p*) − *lp b'*
from *lt-p'' 4* **have** *y2*: *punit.lt* (*rep-list p'*) ≺ *y* + *punit.lt* (*rep-list b'*)
unfolding *y-def* **by** (*rule ordered-powerprod-lin.le-less-trans*)
from *3* **have** *lt b'* $adds_t$ *sig-of-pair p* **by** (*rule is-canon-rewriterD3*)
hence *lp b' adds lp p'* **and** *component-of-term* (*lt b'*) = *component-of-term* (*lt*

p')
by (*simp-all add*: *adds-term-def lt-p'*)
hence *y1*: $y \oplus lt\ b' = lt\ p'$ **by** (*simp add*: *y-def splus-def lt-p' adds-minus term-simps*)
from *3* ‹$b \in set\ bs$› ‹$b \neq 0$› ‹*lt b* $adds_t$ *sig-of-pair p*›
have *rword* (*spp-of b*) (*spp-of b'*) **by** (*rule is-canon-rewriterD*)
hence *punit.lt* (*rep-list b'*) $\oplus$ *lt b* $\preceq_t$ *punit.lt* (*rep-list b*) $\oplus$ *lt b'*
by (*auto simp*: ‹*rword = rw-rat*› *rw-rat-def Let-def spp-of-def*)
hence $(x + y) \oplus$ (*punit.lt* (*rep-list b'*) $\oplus$ *lt b*) $\preceq_t$ $(x + y) \oplus$ (*punit.lt* (*rep-list b*) $\oplus$ *lt b'*)
by (*rule splus-mono*)
hence (y + *punit.lt* (*rep-list b'*)) $\oplus$ ($x \oplus$ *lt b*) $\preceq_t$ (x + *punit.lt* (*rep-list b*)) $\oplus$ ($y \oplus$ *lt b'*)
by (*simp add*: *ac-simps*)
hence (y + *punit.lt* (*rep-list b'*)) $\oplus$ *lt p'* $\preceq_t$ *punit.lt* (*rep-list p'*) $\oplus$ *lt p'*
by (*simp only*: *x1 x2 y1*)
hence y + *punit.lt* (*rep-list b'*) $\preceq$ *punit.lt* (*rep-list p'*) **by** (*rule ord-term-canc-left*)
with *y2* **show** *?thesis* **by** *simp*
next
fix *j*
assume *p*: $p = Inr\ j$
hence *lt p'* = *term-of-pair* (*0*, *j*) **by** (*simp add*: *lt-p'*)
with *x1 term-of-pair-pair*[*of lt b*] **have** *lt b* = *term-of-pair* (*0*, *j*)
by (*auto simp*: *splus-def dest*!: *term-of-pair-injective plus-eq-zero-2*)
moreover have *lt b* $\prec_t$ *term-of-pair* (*0*, *j*) **by** (*rule rb-aux-inv-D4*, *fact*, *simp add*: *p*, *fact*)
ultimately show *?thesis* **by** *simp*
qed
qed
moreover have ¬ *is-sig-red* ($\prec_t$) (=) (*set bs*) *p'*
proof
assume *is-sig-red* ($\prec_t$) (=) (*set bs*) *p'*
hence *is-sig-red* ($\prec_t$) ($\preceq$) (*set bs*) *p'* **by** (*simp add*: *is-sig-red-top-tail-cases*)
with *sig-trd-irred* **show** *False* **unfolding** *p'-def* **..**
qed
ultimately show *?thesis* **by** (*simp add*: *p'-def is-sig-red-sing-reg-cases*)
qed

lemma *rb-aux-top-irred*:
assumes *rword-strict* = *rw-rat-strict* **and** *rb-aux-inv* (*fst args*) **and** $b \in$ *set* (*fst* (*fst* (*rb-aux args*)))
and $\bigwedge b0.\ b0 \in$ *set* (*fst* (*fst args*)) $\Longrightarrow$ ¬ *is-sig-red* ($\preceq_t$) (=) (*set* (*fst* (*fst args*)) $-$ {*b0*}) *b0*
shows ¬ *is-sig-red* ($\preceq_t$) (=) (*set* (*fst* (*fst* (*rb-aux args*))) $-$ {*b*}) *b*
proof −
from *assms*(*2*) **have** *rb-aux-dom args* **by** (*rule rb-aux-domI*)
thus *?thesis* **using** *assms*(*2*, *3*, *4*)
proof (*induct args rule*: *rb-aux.pinduct*)
case (*1 bs ss z*)

let *?nil* = []::((('t ⇒₀ 'b) × ('t ⇒₀ 'b)) + nat) *list*
from *1(3)* **have** *b* ∈ *set* (*fst* (*fst* ((*bs*, *ss*, *?nil*), *z*))) **by** (*simp add*: *rb-aux.psimps*(*1*)[*OF 1*(*1*)])
hence ¬ *is-sig-red* ($\preceq_t$) (=) (*set* (*fst* (*fst* ((*bs*, *ss*, *?nil*), *z*))) − {*b*}) *b* **by** (*rule 1*(*4*))
thus *?case* **by** (*simp add*: *rb-aux.psimps*(*1*)[*OF 1*(*1*)])
next
case (*2 bs ss p ps z*)
from *2*(*5*) **have** ∗: *rb-aux-inv* (*bs*, *ss*, *p* # *ps*) **by** (*simp only*: *fst-conv*)
define *p'* **where** *p'* = *sig-trd bs* (*poly-of-pair p*)
from *2*(*6*) **show** *?case*
proof (*simp add*: *rb-aux.psimps*(*2*)[*OF 2*(*1*)] *Let-def p'-def*[*symmetric*] *split*: *if-splits*)
note *refl*
moreover assume *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
moreover from ∗ *this* **have** *rb-aux-inv* (*fst* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*))
unfolding *fst-conv* **by** (*rule rb-aux-inv-preserved-1*)
moreover assume *b* ∈ *set* (*fst* (*fst* (*rb-aux* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*))))
ultimately show ¬ *is-sig-red* ($\preceq_t$) (=) (*set* (*fst* (*fst* (*rb-aux* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*)))) − {*b*}) *b*
proof (*rule 2*(*2*))
fix *b0*
assume *b0* ∈ *set* (*fst* (*fst* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*)))
hence *b0* ∈ *set* (*fst* (*fst* ((*bs*, *ss*, *p* # *ps*), *z*))) **by** *simp*
hence ¬ *is-sig-red* ($\preceq_t$) (=) (*set* (*fst* (*fst* ((*bs*, *ss*, *p* # *ps*), *z*))) − {*b0*}) *b0* **by** (*rule 2*(*7*))
thus ¬ *is-sig-red* ($\preceq_t$) (=) (*set* (*fst* (*fst* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*))) − {*b0*}) *b0*
by *simp*
qed
next
note *refl*
moreover assume ¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
moreover note *refl*
moreover assume *rep-list p'* = *0*
moreover from ∗ *this* **have** *rb-aux-inv* (*fst* ((*bs*, *lt p'* # *new-syz-sigs ss bs p*, *ps*), *Suc z*))
unfolding *p'-def fst-conv* **by** (*rule rb-aux-inv-preserved-2*)
moreover assume *b* ∈ *set* (*fst* (*fst* (*rb-aux* ((*bs*, *lt p'* # *new-syz-sigs ss bs p*, *ps*), *Suc z*))))
ultimately show ¬ *is-sig-red* ($\preceq_t$) (=) (*set* (*fst* (*fst* (*rb-aux* ((*bs*, *lt p'* # *new-syz-sigs ss bs p*, *ps*), *Suc z*)))) − {*b*}) *b*
proof (*rule 2*(*3*)[*simplified p'-def*[*symmetric*]])
fix *b0*
assume *b0* ∈ *set* (*fst* (*fst* ((*bs*, *lt p'* # *new-syz-sigs ss bs p*, *ps*), *Suc z*)))
hence *b0* ∈ *set* (*fst* (*fst* ((*bs*, *ss*, *p* # *ps*), *z*))) **by** *simp*

hence ¬ *is-sig-red* $(\preceq_t)$ $(=)$ (*set* (*fst* (*fst* ((*bs*, *ss*, *p* # *ps*), *z*))) − {*b0*}) *b0* **by** (*rule* 2(7))
thus ¬ *is-sig-red* $(\preceq_t)$ $(=)$ (*set* (*fst* (*fst* ((*bs*, *lt p'* # *new-syz-sigs ss bs p*, *ps*), *Suc z*))) − {*b0*}) *b0*
by *simp*
qed
next
note *refl*
moreover assume ¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
moreover note *refl*
moreover assume *rep-list p'* ≠ *0*
moreover from ∗ ‹¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p*› *this*
have *inv*: *rb-aux-inv* (*fst* ((*p'* # *bs*, *new-syz-sigs ss bs p*, *add-spairs ps bs p'*), *z*))
unfolding *p'-def fst-conv* **by** (*rule rb-aux-inv-preserved-3*)
moreover assume *b* ∈ *set* (*fst* (*fst* (*rb-aux* ((*p'* # *bs*, *new-syz-sigs ss bs p*, *add-spairs ps bs p'*), *z*))))
ultimately show ¬ *is-sig-red* $(\preceq_t)$ $(=)$ (*set* (*fst* (*fst* (*rb-aux* ((*p'* # *bs*,
new-syz-sigs ss bs p, *add-spairs ps bs p'*), *z*)))) − {*b*}) *b*
proof (*rule* 2(4)[*simplified p'-def*[*symmetric*]])
fix *b0*
assume *b0* ∈ *set* (*fst* (*fst* ((*p'* # *bs*, *new-syz-sigs ss bs p*, *add-spairs ps bs p'*), *z*)))
hence *b0* = *p'* ∨ *b0* ∈ *set bs* **by** *simp*
hence ¬ *is-sig-red* $(\preceq_t)$ $(=)$ (({*p'*} − {*b0*}) ∪ (*set bs* − {*b0*})) *b0*
proof
assume *b0* = *p'*
have ¬ *is-sig-red* $(\preceq_t)$ $(=)$ (*set bs* − {*b0*}) *p'*
proof
assume *is-sig-red* $(\preceq_t)$ $(=)$ (*set bs* − {*b0*}) *p'*
moreover have *set bs* − {*b0*} ⊆ *set bs* **by** *fastforce*
ultimately have *is-sig-red* $(\preceq_t)$ $(=)$ (*set bs*) *p'* **by** (*rule is-sig-red-mono*)
moreover have ¬ *is-sig-red* $(\preceq_t)$ $(=)$ (*set bs*) *p'* **unfolding** *p'-def*
using *assms*(1) ∗ ‹¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p*› **by** (*rule rb-aux-top-irred'*)
ultimately show *False* **by** *simp*
qed
thus *?thesis* **by** (*simp add*: ‹*b0* = *p'*›)
next
assume *b0* ∈ *set bs*
hence *b0* ∈ *set* (*fst* (*fst* ((*bs*, *ss*, *p* # *ps*), *z*))) **by** *simp*
hence ¬ *is-sig-red* $(\preceq_t)$ $(=)$ (*set* (*fst* (*fst* ((*bs*, *ss*, *p* # *ps*), *z*))) − {*b0*}) *b0* **by** (*rule* 2(7))
hence ¬ *is-sig-red* $(\preceq_t)$ $(=)$ (*set bs* − {*b0*}) *b0* **by** *simp*
moreover have ¬ *is-sig-red* $(\preceq_t)$ $(=)$ ({*p'*} − {*b0*}) *b0*
proof
assume *is-sig-red* $(\preceq_t)$ $(=)$ ({*p'*} − {*b0*}) *b0*
moreover have {*p'*} − {*b0*} ⊆ {*p'*} **by** *fastforce*

ultimately have *is-sig-red* $(\preceq_t)$ (=) $\{p'\}$ *b0* **by** (*rule is-sig-red-mono*)
hence *lt* $p' \preceq_t$ *lt b0* **by** (*rule is-sig-redD-lt*)

from *inv* **have** *rb-aux-inv* (p' # *bs*, *new-syz-sigs ss bs p*, *add-spairs ps bs* p')
by (*simp only*: *fst-conv*)
hence *rb-aux-inv1* (p' # *bs*) **by** (*rule rb-aux-inv-D1*)
hence *sorted-wrt* ($\lambda x\ y.$ *lt* $y \prec_t$ *lt* x) (p' # *bs*) **by** (*rule rb-aux-inv1-D3*)
with ‹*b0* ∈ *set bs*› **have** *lt b0* $\prec_t$ *lt* p' **by** *simp*
with ‹*lt* $p' \preceq_t$ *lt b0*› **show** *False* **by** *simp*
qed
ultimately show *?thesis* **by** (*simp add*: *is-sig-red-Un*)
qed
thus ¬ *is-sig-red* $(\preceq_t)$ (=) (*set* (*fst* (*fst* ((p' # *bs*, *new-syz-sigs ss bs p*, *add-spairs ps bs* p'), *z*))) − {*b0*}) *b0*
by (*simp add*: *Un-Diff*[*symmetric*])
qed
qed
qed
qed

corollary *rb-aux-is-min-sig-GB*:
assumes *rword-strict* = *rw-rat-strict*
shows *is-min-sig-GB dgrad* (*set* (*fst* (*fst* (*rb-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0..<length fs*]), *z*)))))
(**is** *is-min-sig-GB* - (*set* (*fst* (*fst* (*rb-aux ?args*)))))
unfolding *is-min-sig-GB-def*
proof (*intro conjI allI ballI impI*)
from *rb-aux-inv-init-fst* **have** *inv*: *rb-aux-inv* (*fst* (*rb-aux ?args*))
and *rb-aux-dom ?args*
by (*rule rb-aux-inv-invariant*, *rule rb-aux-domI*)
from *this*(*2*) **obtain** *bs ss* z' **where** *eq*: *rb-aux ?args* = ((*bs*, *ss*, []), z')
by (*rule rb-aux-shape*)
from *inv* **have** *rb-aux-inv* (*bs*, *ss*, []) **by** (*simp only*: *eq fst-conv*)
hence *rb-aux-inv1 bs* **by** (*rule rb-aux-inv-D1*)
hence *set bs* ⊆ *dgrad-sig-set dgrad* **by** (*rule rb-aux-inv1-D1*)
thus *set* (*fst* (*fst* (*rb-aux ?args*))) ⊆ *dgrad-sig-set dgrad* **by** (*simp add*: *eq*)
next
fix *u*
show *is-sig-GB-in dgrad* (*set* (*fst* (*fst* (*rb-aux ?args*)))) *u* **by** (*fact rb-aux-is-sig-GB-in*)
next
fix *g*
assume *g* ∈ *set* (*fst* (*fst* (*rb-aux ?args*)))
with *assms*(*1*) *rb-aux-inv-init-fst*
show ¬ *is-sig-red* $(\preceq_t)$ (=) (*set* (*fst* (*fst* (*rb-aux ?args*))) − {*g*}) *g*
by (*rule rb-aux-top-irred*) *simp*
qed

corollary *rb-is-min-sig-GB*:

assumes *rword-strict* = *rw-rat-strict*
shows *is-min-sig-GB dgrad* (*set* (*fst rb*))
using *rb-aux-is-min-sig-GB*[*OF assms*, *of 0*] **by** (*auto simp*: *rb-def split*: *prod.split*)

4.2.12 No Zero-Reductions

fun *rb-aux-inv2* :: $(('t \Rightarrow_0 'b)$ *list* $\times$ *'t list* $\times$ $((('t \Rightarrow_0 'b) \times ('t \Rightarrow_0 'b)) + nat)$ *list*) $\Rightarrow$ *bool*
where *rb-aux-inv2* (*bs*, *ss*, *ps*) =
(*rb-aux-inv* (*bs*, *ss*, *ps*) $\wedge$
($\forall$ *j*<*length fs*. *Inr j* $\notin$ *set ps* $\longrightarrow$
(*fs* ! *j* $\in$ *ideal* (*rep-list* ' *set* (*filter* (λ*b*. *component-of-term* (*lt b*) < *Suc j*) *bs*)) $\wedge$
($\forall$ *b*$\in$*set bs*. *component-of-term* (*lt b*) < *j* $\longrightarrow$
($\exists$ *s*$\in$*set ss*. *s* $adds_t$ *term-of-pair* (*punit.lt* (*rep-list b*), *j*))))))

lemma *rb-aux-inv2-D1*: *rb-aux-inv2 args* $\Longrightarrow$ *rb-aux-inv args*
by (*metis prod.exhaust rb-aux-inv2.simps*)

lemma *rb-aux-inv2-D2*:
rb-aux-inv2 (*bs*, *ss*, *ps*) $\Longrightarrow$ *j* < *length fs* $\Longrightarrow$ *Inr j* $\notin$ *set ps* $\Longrightarrow$
fs ! *j* $\in$ *ideal* (*rep-list* ' *set* (*filter* (λ*b*. *component-of-term* (*lt b*) < *Suc j*) *bs*))
by *simp*

lemma *rb-aux-inv2-E*:
assumes *rb-aux-inv2* (*bs*, *ss*, *ps*) **and** *j* < *length fs* **and** *Inr j* $\notin$ *set ps* **and** *b* $\in$ *set bs*
and *component-of-term* (*lt b*) < *j*
obtains *s* **where** *s* $\in$ *set ss* **and** *s* $adds_t$ *term-of-pair* (*punit.lt* (*rep-list b*), *j*)
using *assms* **by** *auto*

context
assumes *pot*: *is-pot-ord*
begin

lemma *sig-red-zero-filter*:
assumes *sig-red-zero* ($\preceq_t$) (*set bs*) *r* **and** *component-of-term* (*lt r*) < *j*
shows *sig-red-zero* ($\preceq_t$) (*set* (*filter* (λ*b*. *component-of-term* (*lt b*) < *j*) *bs*)) *r*
proof –
have ($\preceq_t$) = ($\preceq_t$) $\vee$ ($\preceq_t$) = ($\prec_t$) **by** *simp*
with *assms*(*1*) **have** *sig-red-zero* ($\preceq_t$) {*b*$\in$*set bs*. *lt b* $\preceq_t$ *lt r*} *r* **by** (*rule sig-red-zero-subset*)
moreover have {*b*$\in$*set bs*. *lt b* $\preceq_t$ *lt r*} $\subseteq$ *set* (*filter* (λ*b*. *component-of-term* (*lt b*) < *j*) *bs*)
proof
fix *b*
assume *b* $\in$ {*b*$\in$*set bs*. *lt b* $\preceq_t$ *lt r*}
hence *b* $\in$ *set bs* **and** *lt b* $\preceq_t$ *lt r* **by** *simp-all*
from *pot this*(*2*) **have** *component-of-term* (*lt b*) $\leq$ *component-of-term* (*lt r*) **by** (*rule is-pot-ordD2*)

also have ... < j **by** (*fact assms*(*2*))
finally have *component-of-term* (*lt b*) < j **.**
with ‹$b \in$ *set bs*› **show** $b \in$ *set* (*filter* (λb. *component-of-term* (*lt b*) < j) *bs*) **by** *simp*
qed
ultimately show *?thesis* **by** (*rule sig-red-zero-mono*)
qed

lemma *rb-aux-inv2-preserved-0*:
assumes *rb-aux-inv2* (*bs*, *ss*, *p* # *ps*) **and** $j <$ *length fs* **and** *Inr j* $\notin$ *set ps*
and $b \in$ *set bs* **and** *component-of-term* (*lt b*) < j
shows $\exists$ *s*$\in$*set* (*new-syz-sigs ss bs p*). *s* $adds_t$ *term-of-pair* (*punit.lt* (*rep-list b*), *j*)
proof (*rule sum-prodE*)
fix *x y*
assume *p*: *p* = *Inl* (*x*, *y*)
with *assms*(*3*) **have** *Inr j* $\notin$ *set* (*p* # *ps*) **by** *simp*
with *assms*(*1*, *2*) **obtain** *s* **where** $s \in$ *set ss* **and** $*$: *s* $adds_t$ *term-of-pair* (*punit.lt* (*rep-list b*), *j*)
using *assms*(*4*, *5*) **by** (*rule rb-aux-inv2-E*)
from *this*(*1*) **have** $s \in$ *set* (*new-syz-sigs ss bs p*) **by** (*simp add*: *p*)
with $*$ **show** *?thesis* **..**
next
fix *i*
assume *p*: *p* = *Inr i*
have *trans*: *transp* ($adds_t$) **by** (*rule transpI*, *drule adds-term-trans*)
from *adds-term-refl* **have** *refl*: *reflp* ($adds_t$) **by** (*rule reflpI*)
let *?v* = *term-of-pair* (*punit.lt* (*rep-list b*), *j*)
let *?f* = λb. *term-of-pair* (*punit.lt* (*rep-list b*), *i*)
define *ss'* **where** *ss'* = *filter-min* ($adds_t$) (*map ?f bs*)
have *eq*: *new-syz-sigs ss bs p* = *filter-min-append* ($adds_t$) *ss ss'* **by** (*simp add*: *p ss'-def pot*)
show *?thesis*
proof (*cases i* = *j*)
case *True*
from ‹$b \in$ *set bs*› **have** *?v* $\in$ *?f* ‘ *set bs* **unfolding** ‹*i* = *j*› **by** (*rule imageI*)
hence *?v* $\in$ *set ss* $\cup$ *set* (*map ?f bs*) **by** *simp*
thus *?thesis*
proof
assume *?v* $\in$ *set ss*
hence *?v* $\in$ *set ss* $\cup$ *set ss'* **by** *simp*
with *trans refl* **obtain** *s* **where** $s \in$ *set* (*new-syz-sigs ss bs p*) **and** *s* $adds_t$ *?v*
unfolding *eq* **by** (*rule filter-min-append-relE*)
thus *?thesis* **..**
next
assume *?v* $\in$ *set* (*map ?f bs*)
with *trans refl* **obtain** *s* **where** $s \in$ *set ss'* **and** *s* $adds_t$ *?v*
unfolding *ss'-def* **by** (*rule filter-min-relE*)
from *this*(*1*) **have** $s \in$ *set ss* $\cup$ *set ss'* **by** *simp*

```
      with trans refl obtain s' where s': s' ∈ set (new-syz-sigs ss bs p) and s'
adds_t s
        unfolding eq by (rule filter-min-append-relE)
      from this(2) ‹s adds_t ?v› have s' adds_t ?v by (rule adds-term-trans)
      with s' show ?thesis ..
    qed
  next
    case False
    with assms(3) have Inr j ∉ set (p # ps) by (simp add: p)
    with assms(1, 2) obtain s where s ∈ set ss and s adds_t ?v
      using assms(4, 5) by (rule rb-aux-inv2-E)
    from this(1) have s ∈ set ss ∪ set (map ?f bs) by simp
    thus ?thesis
    proof
      assume s ∈ set ss
      hence s ∈ set ss ∪ set ss' by simp
      with trans refl obtain s' where s': s' ∈ set (new-syz-sigs ss bs p) and s'
adds_t s
        unfolding eq by (rule filter-min-append-relE)
      from this(2) ‹s adds_t ?v› have s' adds_t ?v by (rule adds-term-trans)
      with s' show ?thesis ..
    next
      assume s ∈ set (map ?f bs)
      with trans refl obtain s' where s' ∈ set ss' and s' adds_t s
        unfolding ss'-def by (rule filter-min-relE)
      from this(1) have s' ∈ set ss ∪ set ss' by simp
      with trans refl obtain s'' where s'': s'' ∈ set (new-syz-sigs ss bs p) and s''
adds_t s'
        unfolding eq by (rule filter-min-append-relE)
      from this(2) ‹s' adds_t s› have s'' adds_t s by (rule adds-term-trans)
      hence s'' adds_t ?v using ‹s adds_t ?v› by (rule adds-term-trans)
      with s'' show ?thesis ..
    qed
  qed
qed

lemma rb-aux-inv2-preserved-1:
  assumes rb-aux-inv2 (bs, ss, p # ps) and sig-crit bs (new-syz-sigs ss bs p) p
  shows rb-aux-inv2 (bs, new-syz-sigs ss bs p, ps)
  unfolding rb-aux-inv2.simps
proof (intro allI conjI impI ballI)
  from assms(1) have inv: rb-aux-inv (bs, ss, p # ps) by (rule rb-aux-inv2-D1)
  thus rb-aux-inv (bs, new-syz-sigs ss bs p, ps)
    using assms(2) by (rule rb-aux-inv-preserved-1)

  fix j
  assume j < length fs and Inr j ∉ set ps
  show fs ! j ∈ ideal (rep-list ‘ set (filter (λb. component-of-term (lt b) < Suc j)
bs))
```

proof (*cases p = Inr j*)
case *True*
with *assms*(*2*) **have** *is-pred-syz* (*new-syz-sigs ss bs p*) (*term-of-pair* (*0*, *j*)) **by** *simp*
let *?X = set* (*filter* (*λb. component-of-term* (*lt b*) *< Suc j*) *bs*)
have *rep-list* (*monomial 1* (*term-of-pair* (*0*, *j*))) ∈ *ideal* (*rep-list* ‘ *?X*)
proof (*rule sig-red-zero-idealI*)
have *sig-red-zero* ($\prec_t$) (*set bs*) (*monomial 1* (*term-of-pair* (*0*, *j*)))
proof (*rule syzygy-crit*)
from *inv* **have** *is-RB-upt dgrad rword* (*set bs*) (*sig-of-pair p*)
by (*rule rb-aux-inv-is-RB-upt-Cons*)
with *dgrad* **have** *is-sig-GB-upt dgrad* (*set bs*) (*sig-of-pair p*)
by (*rule is-RB-upt-is-sig-GB-upt*)
thus *is-sig-GB-upt dgrad* (*set bs*) (*term-of-pair* (*0*, *j*)) **by** (*simp add*: ‹*p = Inr j*›)
next
show *monomial 1* (*term-of-pair* (*0*, *j*)) ∈ *dgrad-sig-set dgrad*
by (*rule dgrad-sig-set-closed-monomial*, *simp-all add*: *term-simps dgrad-max-0* ‹*j < length fs*›)
next
show *lt* (*monomial* (*1*::'*b*) (*term-of-pair* (*0*, *j*))) = *term-of-pair* (*0*, *j*) **by** (*simp add*: *lt-monomial*)
next
from *inv assms*(*2*) **have** *sig-crit' bs p* **by** (*rule sig-crit'I-sig-crit*)
thus *is-syz-sig dgrad* (*term-of-pair* (*0*, *j*)) **by** (*simp add*: ‹*p = Inr j*›)
qed (*fact dgrad*)
hence *sig-red-zero* ($\preceq_t$) (*set bs*) (*monomial 1* (*term-of-pair* (*0*, *j*)))
by (*rule sig-red-zero-sing-regI*)
moreover have *component-of-term* (*lt* (*monomial* (*1*::'*b*) (*term-of-pair* (*0*, *j*)))) *< Suc j*
by (*simp add*: *lt-monomial component-of-term-of-pair*)
ultimately show *sig-red-zero* ($\preceq_t$) *?X* (*monomial 1* (*term-of-pair* (*0*, *j*)))
by (*rule sig-red-zero-filter*)
qed
thus *?thesis* **by** (*simp add*: *rep-list-monomial'* ‹*j < length fs*›)
next
case *False*
with ‹*Inr j* ∉ *set ps*› **have** *Inr j* ∉ *set* (*p # ps*) **by** *simp*
with *assms*(*1*) ‹*j < length fs*› **show** *?thesis* **by** (*rule rb-aux-inv2-D2*)
qed
next
fix *j b*
assume *j < length fs* **and** *Inr j* ∉ *set ps* **and** *b* ∈ *set bs* **and** *component-of-term* (*lt b*) *< j*
with *assms*(*1*) **show** ∃ *s*∈*set* (*new-syz-sigs ss bs p*). *s* $adds_t$ *term-of-pair* (*punit.lt* (*rep-list b*), *j*)
by (*rule rb-aux-inv2-preserved-0*)
qed

lemma *rb-aux-inv2-preserved-3*:
  assumes *rb-aux-inv2* (*bs*, *ss*, *p* # *ps*) **and** ¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
    and *rep-list* (*sig-trd bs* (*poly-of-pair p*)) ≠ *0*
  shows *rb-aux-inv2* (*sig-trd bs* (*poly-of-pair p*) # *bs*, *new-syz-sigs ss bs p*,
                    add-spairs ps bs (*sig-trd bs* (*poly-of-pair p*)))
proof −
  from *assms*(*1*) **have** *inv*: *rb-aux-inv* (*bs*, *ss*, *p* # *ps*) **by** (*rule rb-aux-inv2-D1*)
  define *p'* **where** *p'* = *sig-trd bs* (*poly-of-pair p*)
  from *sig-trd-red-rtrancl*[*of bs poly-of-pair p*] **have** *lt p'* = *lt* (*poly-of-pair p*)
    unfolding *p'-def* **by** (*rule sig-red-regular-rtrancl-lt*)
   also have ... = *sig-of-pair p* **by** (*rule sym*, *rule pair-list-sig-of-pair*, *fact inv*, *simp*)
  finally have *lt-p'*: *lt p'* = *sig-of-pair p* **.**
  show *?thesis* **unfolding** *rb-aux-inv2.simps p'-def*[*symmetric*]
  proof (*intro allI conjI impI ballI*)
    show *rb-aux-inv* (*p'* # *bs*, *new-syz-sigs ss bs p*, *add-spairs ps bs p'*)
      unfolding *p'-def* **using** *inv assms*(*2*, *3*) **by** (*rule rb-aux-inv-preserved-3*)
  next
    fix *j*
    assume *j* < *length fs* **and** ∗: *Inr j* ∉ *set* (*add-spairs ps bs p'*)
    show *fs* ! *j* ∈ *ideal* (*rep-list* ‘ *set* (*filter* (λ*b*. *component-of-term* (*lt b*) < *Suc j*) (*p'* # *bs*)))
    proof (*cases p* = *Inr j*)
      case *True*
      let *?X* = *set* (*filter* (λ*b*. *component-of-term* (*lt b*) < *Suc j*) (*p'* # *bs*))
      have *rep-list* (*monomial 1* (*term-of-pair* (*0*, *j*))) ∈ *ideal* (*rep-list* ‘ *?X*)
      proof (*rule sig-red-zero-idealI*)
        have *sig-red-zero* ($\preceq_t$) (*set* (*p'* # *bs*)) (*monomial 1* (*term-of-pair* (*0*, *j*)))
        proof (*rule sig-red-zeroI*)
          have (*sig-red* ($\prec_t$) (⪯) (*set bs*))** (*monomial 1* (*term-of-pair* (*0*, *j*))) *p'*
            using *sig-trd-red-rtrancl*[*of bs poly-of-pair p*] **by** (*simp add*: *True p'-def*)
          moreover have *set bs* ⊆ *set* (*p'* # *bs*) **by** *fastforce*
            ultimately have (*sig-red* ($\prec_t$) (⪯) (*set* (*p'* # *bs*)))** (*monomial 1* (*term-of-pair* (*0*, *j*))) *p'*
            by (*rule sig-red-rtrancl-mono*)
          hence (*sig-red* ($\preceq_t$) (⪯) (*set* (*p'* # *bs*)))** (*monomial 1* (*term-of-pair* (*0*, *j*))) *p'*
            by (*rule sig-red-rtrancl-sing-regI*)
          also have *sig-red* ($\preceq_t$) (⪯) (*set* (*p'* # *bs*)) *p' 0* **unfolding** *sig-red-def*
          proof (*intro exI bexI*)
            from *assms*(*3*) **have** *rep-list p'* ≠ *0* **by** (*simp add*: *p'-def*)
            show *sig-red-single* ($\preceq_t$) (⪯) *p' 0 p' 0*
            proof (*rule sig-red-singleI*)
              show *rep-list p'* ≠ *0* **by** *fact*
            next
              from ‹*rep-list p'* ≠ *0*› **have** *punit.lt* (*rep-list p'*) ∈ *keys* (*rep-list p'*)
                by (*rule punit.lt-in-keys*)
              thus *0* + *punit.lt* (*rep-list p'*) ∈ *keys* (*rep-list p'*) **by** *simp*
            next

from ‹*rep-list* $p' \neq 0$› **have** *punit.lc* (*rep-list* p') $\neq 0$ **by** (*rule punit.lc-not-0*)
thus $0 = p' -$ *monom-mult* (*lookup* (*rep-list* p') ($0 +$ *punit.lt* (*rep-list* p')) / *punit.lc* (*rep-list* p')) 0 p'
by (*simp add*: *punit.lc-def*[*symmetric*])
qed (*simp-all add*: *term-simps*)
qed *simp*
finally show (*sig-red* ($\preceq_t$) ($\preceq$) (*set* (p' # *bs*)))** (*monomial 1* (*term-of-pair* (0, j))) 0 **.**
qed (*fact rep-list-zero*)
moreover have *component-of-term* (*lt* (*monomial* (1::$'b$) (*term-of-pair* (0, j)))) < *Suc j*
by (*simp add*: *lt-monomial component-of-term-of-pair*)
ultimately show *sig-red-zero* ($\preceq_t$) *?X* (*monomial 1* (*term-of-pair* (0, j)))
by (*rule sig-red-zero-filter*)
qed
thus *?thesis* **by** (*simp add*: *rep-list-monomial*$'$ ‹$j <$ *length fs*›)
next
case *False*
from $*$ **have** *Inr j* $\notin$ *set ps* **by** (*simp add*: *add-spairs-def set-merge-wrt*)
hence *Inr j* $\notin$ *set* (*p* # *ps*) **using** *False* **by** *simp*
with *assms*(*1*) ‹$j <$ *length fs*›
have *fs* ! $j \in$ *ideal* (*rep-list* ‘ *set* (*filter* (λb. *component-of-term* (*lt b*) < *Suc j*) *bs*))
by (*rule rb-aux-inv2-D2*)
also have ... $\subseteq$ *ideal* (*rep-list* ‘ *set* (*filter* (λb. *component-of-term* (*lt b*) < *Suc j*) (p' # *bs*)))
by (*intro ideal.span-mono image-mono*, *fastforce*)
finally show *?thesis* **.**
qed
next
fix *j* **and** *b*::$'t \Rightarrow_0 'b$
assume $j <$ *length fs* **and** $*$: *component-of-term* (*lt b*) $< j$
assume *Inr j* $\notin$ *set* (*add-spairs ps bs* p')
hence *Inr j* $\notin$ *set ps* **by** (*simp add*: *add-spairs-def set-merge-wrt*)
assume $b \in$ *set* (p' # *bs*)
hence $b = p' \vee b \in$ *set bs* **by** *simp*
thus $\exists\, s \in$*set* (*new-syz-sigs ss bs p*). *s* $adds_t$ *term-of-pair* (*punit.lt* (*rep-list b*), *j*)
proof
assume $b = p'$
with $*$ **have** *component-of-term* (*sig-of-pair p*) < *component-of-term* (*term-of-pair* (0, j))
by (*simp only*: *lt-p*$'$ *component-of-term-of-pair*)
with *pot* **have** $**$: *sig-of-pair p* $\prec_t$ *term-of-pair* (0, j) **by** (*rule is-pot-ordD*)
have $p \in$ *set* (*p* # *ps*) **by** *simp*
with *inv* **have** *Inr j* $\in$ *set* (*p* # *ps*) **using** ‹$j <$ *length fs*› $**$ **by** (*rule rb-aux-inv-D6-2*)
with ‹*Inr j* $\notin$ *set ps*› **have** $p =$ *Inr j* **by** *simp*

with ∗∗ **show** *?thesis* **by** *simp*
next
assume $b \in$ *set bs*
with *assms*(*1*) ‹$j <$ *length fs*› ‹*Inr j* $\notin$ *set ps*› **show** *?thesis*
using ∗ **by** (*rule rb-aux-inv2-preserved-0*)
qed
qed
qed

lemma *rb-aux-inv2-ideal-subset*:
assumes *rb-aux-inv2* (*bs*, *ss*, *ps*) **and** $\bigwedge$*p0*. *p0* $\in$ *set ps* $\Longrightarrow$ $j \leq$ *component-of-term* (*sig-of-pair p0*)
shows *ideal* (*set* (*take j fs*)) $\subseteq$ *ideal* (*rep-list* ‘ *set* (*filter* (λb. *component-of-term* (*lt b*) $<$ *j*) *bs*))
(**is** *ideal ?B* $\subseteq$ *ideal ?A*)
proof (*intro ideal.span-subset-spanI subsetI*)
fix *f*
assume $f \in$ *?B*
then obtain *i* **where** $i <$ *length* (*take j fs*) **and** $f =$ (*take j fs*) ! *i*
by (*metis in-set-conv-nth*)
hence $i <$ *length fs* **and** $i < j$ **and** *f*: $f =$ *fs* ! *i* **by** *auto*
from *this*(*2*) **have** *Suc i* $\leq j$ **by** *simp*
have $f \in$ *ideal* (*rep-list* ‘ *set* (*filter* (λb. *component-of-term* (*lt b*) $<$ *Suc i*) *bs*))
unfolding *f* **using** *assms*(*1*) ‹$i <$ *length fs*›
proof (*rule rb-aux-inv2-D2*)
show *Inr i* $\notin$ *set ps*
proof
assume *Inr i* $\in$ *set ps*
hence $j \leq$ *component-of-term* (*sig-of-pair* (*Inr i*)) **by** (*rule assms*(*2*))
hence $j \leq i$ **by** (*simp add*: *component-of-term-of-pair*)
with ‹$i < j$› **show** *False* **by** *simp*
qed
qed
also have ... $\subseteq$ *ideal ?A*
by (*intro ideal.span-mono image-mono*, *auto dest*: *order-less-le-trans*[*OF* - ‹*Suc i* $\leq j$›])
finally show $f \in$ *ideal ?A* **.**
qed

lemma *rb-aux-inv-is-Groebner-basis*:
assumes *hom-grading dgrad* **and** *rb-aux-inv* (*bs*, *ss*, *ps*)
and $\bigwedge$*p0*. *p0* $\in$ *set ps* $\Longrightarrow$ $j \leq$ *component-of-term* (*sig-of-pair p0*)
shows *punit.is-Groebner-basis* (*rep-list* ‘ *set* (*filter* (λb. *component-of-term* (*lt b*) $<$ *j*) *bs*))
(**is** *punit.is-Groebner-basis* (*rep-list* ‘ *set ?bs*))
using *dgrad assms*(*1*)
proof (*rule is-sig-GB-upt-is-Groebner-basis*)
show *set ?bs* $\subseteq$ *dgrad-sig-set'* *j dgrad*
proof

fix b
assume $b \in$ *set ?bs*
hence $b \in$ *set bs* **and** *component-of-term* $(lt\ b) < j$ **by** *simp-all*
show $b \in$ *dgrad-sig-set'* j *dgrad* **unfolding** *dgrad-sig-set'-def*
proof
from *assms*(*2*) **have** *rb-aux-inv1 bs* **by** (*rule rb-aux-inv-D1*)
hence *set bs* $\subseteq$ *dgrad-sig-set dgrad* **by** (*rule rb-aux-inv1-D1*)
with ‹$b \in$ *set bs*› **have** $b \in$ *dgrad-sig-set dgrad* **..**
thus $b \in$ *dgrad-max-set dgrad* **by** (*simp add*: *dgrad-sig-set'-def*)
next
show $b \in$ *sig-inv-set'* j
proof (*rule sig-inv-setI'*)
fix v
assume $v \in$ *keys* b
hence $v \preceq_t lt\ b$ **by** (*rule lt-max-keys*)
with *pot* **have** *component-of-term* $v \le$ *component-of-term* $(lt\ b)$ **by** (*rule is-pot-ordD2*)
also have ... $< j$ **by** *fact*
finally show *component-of-term* $v < j$ **.**
qed
qed
qed
next
fix u
assume u: *component-of-term* $u < j$
from *dgrad* **have** *is-sig-GB-upt dgrad* (*set bs*) (*term-of-pair* $(0, j)$)
proof (*rule is-RB-upt-is-sig-GB-upt*)
from *assms*(*2*) **show** *is-RB-upt dgrad rword* (*set bs*) (*term-of-pair* $(0, j)$)
proof (*rule rb-aux-inv-is-RB-upt*)
fix p
assume $p \in$ *set ps*
hence $j \le$ *component-of-term* (*sig-of-pair* p) **by** (*rule assms*(*3*))
with *pot* **show** *term-of-pair* $(0, j) \preceq_t$ *sig-of-pair* p
by (*auto simp*: *is-pot-ord term-simps zero-min*)
qed
qed
moreover from *pot* **have** $u \prec_t$ *term-of-pair* $(0, j)$
by (*rule is-pot-ordD*) (*simp only*: u *component-of-term-of-pair*)
ultimately have *1*: *is-sig-GB-in dgrad* (*set bs*) u **by** (*rule is-sig-GB-uptD2*)
show *is-sig-GB-in dgrad* (*set ?bs*) u
proof (*rule is-sig-GB-inI*)
fix $r :: 't \Rightarrow_0 'b$
assume $lt\ r = u$
assume $r \in$ *dgrad-sig-set dgrad*
with *1* **have** *sig-red-zero* $(\preceq_t)$ (*set bs*) r **using** ‹$lt\ r = u$› **by** (*rule is-sig-GB-inD*)
moreover from u **have** *component-of-term* $(lt\ r) < j$ **by** (*simp only*: ‹$lt\ r = u$›)
ultimately show *sig-red-zero* $(\preceq_t)$ (*set ?bs*) r **by** (*rule sig-red-zero-filter*)
qed

qed

lemma *rb-aux-inv2-no-zero-red*:
assumes *hom-grading dgrad* **and** *is-regular-sequence fs* **and** *rb-aux-inv2* (*bs*, *ss*, *p* # *ps*)
and ¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
shows *rep-list* (*sig-trd bs* (*poly-of-pair p*)) ≠ *0*
proof
from *assms*(*3*) **have** *inv*: *rb-aux-inv* (*bs*, *ss*, *p* # *ps*) **by** (*rule rb-aux-inv2-D1*)
moreover have *p* ∈ *set* (*p* # *ps*) **by** *simp*
ultimately have *sig-p*: *sig-of-pair p* = *lt* (*poly-of-pair p*) **and** *poly-of-pair p* ≠ *0*
and *p-in*: *poly-of-pair p* ∈ *dgrad-sig-set dgrad*
by (*rule pair-list-sig-of-pair*, *rule pair-list-nonzero*, *rule pair-list-dgrad-sig-set*)
from *this*(*2*) **have** *lc* (*poly-of-pair p*) ≠ *0* **by** (*rule lc-not-0*)
from *inv* **have** *rb-aux-inv1 bs* **by** (*rule rb-aux-inv-D1*)
hence *bs-sub*: *set bs* ⊆ *dgrad-sig-set dgrad* **by** (*rule rb-aux-inv1-D1*)

define *p'* **where** *p'* = *sig-trd bs* (*poly-of-pair p*)
define *j* **where** *j* = *component-of-term* (*lt p'*)
define *q* **where** *q* = *lookup* (*vectorize-poly p'*) *j*
let *?bs* = *filter* (λ*b*. *component-of-term* (*lt b*) < *j*) *bs*
let *?fs* = *take* (*Suc j*) *fs*

have *p'* ∈ *dgrad-sig-set dgrad* **unfolding** *p'-def* **using** *dgrad bs-sub p-in sig-trd-red-rtrancl*
by (*rule dgrad-sig-set-closed-sig-red-rtrancl*)
hence *p'* ∈ *sig-inv-set* **by** (*simp add*: *dgrad-sig-set'-def*)

have *lt-p'*: *lt p'* = *lt* (*poly-of-pair p*) **and** *lc p'* = *lc* (*poly-of-pair p*)
unfolding *p'-def* **using** *sig-trd-red-rtrancl*
by (*rule sig-red-regular-rtrancl-lt*, *rule sig-red-regular-rtrancl-lc*)
from *this*(*2*) ‹*lc* (*poly-of-pair p*) ≠ *0*› **have** *p'* ≠ *0* **by** (*simp add*: *lc-eq-zero-iff*[*symmetric*])
hence *lt p'* ∈ *keys p'* **by** (*rule lt-in-keys*)
hence *j* ∈ *keys* (*vectorize-poly p'*) **by** (*simp add*: *keys-vectorize-poly j-def*)
hence *q* ≠ *0* **by** (*simp add*: *q-def in-keys-iff*)

from ‹*p'* ∈ *sig-inv-set*› ‹*lt p'* ∈ *keys p'*› **have** *j* < *length fs*
unfolding *j-def* **by** (*rule sig-inv-setD'*)
with *le-refl* **have** *fs* ! *j* ∈ *set* (*drop j fs*) **by** (*rule nth-in-set-dropI*)
with *fs-distinct le-refl* **have** *0*: *fs* ! *j* ∉ *set* (*take j fs*)
by (*auto dest*: *set-take-disj-set-drop-if-distinct*)

have *1*: *j* ≤ *component-of-term* (*sig-of-pair p0*) **if** *p0* ∈ *set* (*p* # *ps*) **for** *p0*
proof −
from *that* **have** *p0* = *p* ∨ *p0* ∈ *set ps* **by** *simp*
thus *?thesis*
proof
assume *p0* = *p*
thus *?thesis* **by** (*simp add*: *j-def lt-p' sig-p*)
next

assume *p0* ∈ *set ps*
from *inv* **have** *sorted-wrt pair-ord* (*p* # *ps*) **by** (*rule rb-aux-inv-D5*)
hence *Ball* (*set ps*) (*pair-ord p*) **by** *simp*
hence *pair-ord p p0* **using** ‹*p0* ∈ *set ps*› **..**
hence *lt p'* $\preceq_t$ *sig-of-pair p0* **by** (*simp add*: *pair-ord-def lt-p' sig-p*)
thus *?thesis* **using** *pot* **by** (*auto simp add*: *is-pot-ord j-def term-simps*)
qed
qed
with *assms*(*1*) *inv* **have** *gb*: *punit.is-Groebner-basis* (*rep-list* ‘ *set ?bs*)
by (*rule rb-aux-inv-is-Groebner-basis*)

have *p'* ∈ *sig-inv-set'* (*Suc j*)
proof (*rule sig-inv-setI'*)
fix *v*
assume *v* ∈ *keys p'*
hence *v* $\preceq_t$ *lt p'* **by** (*rule lt-max-keys*)
with *pot* **have** *component-of-term v* ≤ *j* **unfolding** *j-def* **by** (*rule is-pot-ordD2*)
thus *component-of-term v* < *Suc j* **by** *simp*
qed
hence *2*: *keys* (*vectorize-poly p'*) ⊆ {*0..<Suc j*} **by** (*rule sig-inv-setD*)
moreover assume *rep-list p'* = *0*
ultimately have *0* = (∑ *k*∈*keys* (*pm-of-idx-pm ?fs* (*vectorize-poly p'*)).
lookup (*pm-of-idx-pm ?fs* (*vectorize-poly p'*)) *k* ∗ *k*)
by (*simp add*: *rep-list-def ideal.rep-def pm-of-idx-pm-take*)
also have ... = (∑ *k*∈*set ?fs*. *lookup* (*pm-of-idx-pm ?fs* (*vectorize-poly p'*)) *k* ∗ *k*)
using *finite-set keys-pm-of-idx-pm-subset* **by** (*rule sum.mono-neutral-left*) (*simp add*: *in-keys-iff*)

also from *2* **have** ... = (∑ *k*∈*set ?fs*. *lookup* (*pm-of-idx-pm fs* (*vectorize-poly p'*)) *k* ∗ *k*)
by (*simp only*: *pm-of-idx-pm-take*)
also have ... = *lookup* (*pm-of-idx-pm fs* (*vectorize-poly p'*)) (*fs* ! *j*) ∗ *fs* ! *j* +
(∑ *k*∈*set* (*take j fs*). *lookup* (*pm-of-idx-pm fs* (*vectorize-poly p'*)) *k* ∗ *k*)
using ‹*j* < *length fs*› **by** (*simp add*: *take-Suc-conv-app-nth q-def sum.insert*[*OF finite-set 0*])
also have ... = *q* ∗ *fs* ! *j* + (∑ *k*∈*set* (*take j fs*). *lookup* (*pm-of-idx-pm fs* (*vectorize-poly p'*)) *k* ∗ *k*)
using *fs-distinct* ‹*j* < *length fs*› **by** (*simp only*: *lookup-pm-of-idx-pm-distinct q-def*)
finally have − (*q* ∗ *fs* ! *j*) =
(∑ *k*∈*set* (*take j fs*). *lookup* (*pm-of-idx-pm fs* (*vectorize-poly p'*)) *k* ∗ *k*)
by (*simp add*: *add-eq-0-iff*)
hence − (*q* ∗ *fs* ! *j*) ∈ *ideal* (*set* (*take j fs*)) **by** (*simp add*: *ideal.sum-in-spanI*)
hence − (− (*q* ∗ *fs* ! *j*)) ∈ *ideal* (*set* (*take j fs*)) **by** (*rule ideal.span-neg*)
hence *q* ∗ *fs* ! *j* ∈ *ideal* (*set* (*take j fs*)) **by** *simp*
with *assms*(*2*) ‹*j* < *length fs*› **have** *q* ∈ *ideal* (*set* (*take j fs*)) **by** (*rule is-regular-sequenceD*)

also from *assms*(*3*) *1* **have** ... ⊆ *ideal* (*rep-list* ' *set ?bs*)
  by (*rule rb-aux-inv2-ideal-subset*)
finally have *q* ∈ *ideal* (*rep-list* ' *set ?bs*) **.**
with *gb* **obtain** *g* **where** *g* ∈ *rep-list* ' *set ?bs* **and** *g* ≠ *0* **and** *punit.lt g adds punit.lt q*
  using ‹*q* ≠ *0*› **by** (*rule punit.GB-adds-lt*[*simplified*])
from *this*(*1*) **obtain** *b* **where** *b* ∈ *set bs* **and** *component-of-term* (*lt b*) < *j* **and** *g*: *g* = *rep-list b*
  by *auto*
from *assms*(*3*) ‹*j* < *length fs*› - *this*(*1*, *2*)
have ∃ *s*∈*set* (*new-syz-sigs ss bs p*). *s* $adds_t$ *term-of-pair* (*punit.lt* (*rep-list b*), *j*)
proof (*rule rb-aux-inv2-preserved-0*)
  show *Inr j* ∉ *set ps*
  proof
    assume *Inr j* ∈ *set ps*
    with *inv* **have** *sig-of-pair p* ≠ *term-of-pair* (*0*, *j*) **by** (*rule Inr-in-tailD*)
    hence *lt p*′ ≠ *term-of-pair* (*0*, *j*) **by** (*simp add*: *lt-p*′ *sig-p*)
    from *inv* **have** *sorted-wrt pair-ord* (*p* # *ps*) **by** (*rule rb-aux-inv-D5*)
    hence *Ball* (*set ps*) (*pair-ord p*) **by** *simp*
    hence *pair-ord p* (*Inr j*) **using** ‹*Inr j* ∈ *set ps*› **..**
    hence *lt p*′ $\preceq_t$ *term-of-pair* (*0*, *j*) **by** (*simp add*: *pair-ord-def lt-p*′ *sig-p*)
    hence *lp p*′ ⪯ *0* **using** *pot* **by** (*simp add*: *is-pot-ord j-def term-simps*)
    hence *lp p*′ = *0* **using** *zero-min* **by** (*rule ordered-powerprod-lin.order-antisym*)
    hence *lt p*′ = *term-of-pair* (*0*, *j*) **by** (*metis j-def term-of-pair-pair*)
    with ‹*lt p*′ ≠ *term-of-pair* (*0*, *j*)› **show** *False* **..**
  qed
qed
then obtain *s* **where** *s-in*: *s* ∈ *set* (*new-syz-sigs ss bs p*) **and** *s* $adds_t$ *term-of-pair* (*punit.lt g*, *j*)
  unfolding *g* **..**
from *this*(*2*) ‹*punit.lt g adds punit.lt q*› **have** *s* $adds_t$ *term-of-pair* (*punit.lt q*, *j*)
  by (*metis adds-minus-splus adds-term-splus component-of-term-of-pair pp-of-term-of-pair*)
also have ... = *lt p*′ **by** (*simp only*: *q-def j-def lt-lookup-vectorize term-simps*)
finally have *s* $adds_t$ *sig-of-pair p* **by** (*simp only*: *lt-p*′ *sig-p*)
with *s-in* **have** *pred*: *is-pred-syz* (*new-syz-sigs ss bs p*) (*sig-of-pair p*)
  by (*auto simp*: *is-pred-syz-def*)
have *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
proof (*rule sum-prodE*)
  fix *x y*
  assume *p* = *Inl* (*x*, *y*)
  thus *?thesis* **using** *pred* **by** (*auto simp*: *ord-term-lin.max-def split*: *if-splits*)
next
  fix *i*
  assume *p* = *Inr i*
  thus *?thesis* **using** *pred* **by** *simp*
qed
with *assms*(*4*) **show** *False* **..**
qed

corollary *rb-aux-no-zero-red′*:
assumes *hom-grading dgrad* **and** *is-regular-sequence fs* **and** *rb-aux-inv2* (*fst args*)
shows *snd* (*rb-aux args*) = *snd args*
proof −
from *assms*(*3*) **have** *rb-aux-inv* (*fst args*) **by** (*rule rb-aux-inv2-D1*)
hence *rb-aux-dom args* **by** (*rule rb-aux-domI*)
thus *?thesis* **using** *assms*(*3*)
proof (*induct args rule*: *rb-aux.pinduct*)
case (*1 bs ss z*)
show *?case* **by** (*simp only*: *rb-aux.psimps*(*1*)[*OF 1*(*1*)])
next
case (*2 bs ss p ps z*)
from *2*(*5*) **have** ∗: *rb-aux-inv2* (*bs*, *ss*, *p* # *ps*) **by** (*simp only*: *fst-conv*)
show *?case*
proof (*simp add*: *rb-aux.psimps*(*2*)[*OF 2*(*1*)] *Let-def*, *intro conjI impI*)
note *refl*
moreover assume *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
moreover from ∗ *this* **have** *rb-aux-inv2* (*fst* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*))
unfolding *fst-conv* **by** (*rule rb-aux-inv2-preserved-1*)
ultimately have *snd* (*rb-aux* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*)) =
snd ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*) **by** (*rule 2*(*2*))
thus *snd* (*rb-aux* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*)) = *z* **by** (*simp only*: *snd-conv*)
thus *snd* (*rb-aux* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*)) = *z* **.**
next
assume ¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
with *assms*(*1*, *2*) ∗ **have** *rep-list* (*sig-trd bs* (*poly-of-pair p*)) ≠ *0*
by (*rule rb-aux-inv2-no-zero-red*)
moreover assume *rep-list* (*sig-trd bs* (*poly-of-pair p*)) = *0*
ultimately show *snd* (*rb-aux* ((*bs*, *lt* (*sig-trd bs* (*poly-of-pair p*)) #
new-syz-sigs ss bs p, *ps*), *Suc z*)) = *z* **..**
next
define *p′* **where** *p′* = *sig-trd bs* (*poly-of-pair p*)
note *refl*
moreover assume *a*: ¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
moreover note *p′-def*
moreover assume *b*: *rep-list p′* ≠ *0*
moreover have *rb-aux-inv2* (*fst* ((*p′* # *bs*, *new-syz-sigs ss bs p*, *add-spairs ps bs p′*), *z*))
using ∗ *a b* **unfolding** *fst-conv p′-def* **by** (*rule rb-aux-inv2-preserved-3*)
ultimately have *snd* (*rb-aux* ((*p′* # *bs*, *new-syz-sigs ss bs p*, *add-spairs ps bs p′*), *z*)) =
snd ((*p′* # *bs*, *new-syz-sigs ss bs p*, *add-spairs ps bs p′*), *z*)
by (*rule 2*(*4*))
thus *snd* (*rb-aux* ((*p′* # *bs*, *new-syz-sigs ss bs p*, *add-spairs ps bs p′*), *z*)) = *z*
by (*simp only*: *snd-conv*)
qed
qed

qed

corollary *rb-aux-no-zero-red*:
  assumes *hom-grading dgrad* **and** *is-regular-sequence fs*
  shows *snd* (*rb-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)) = *z*
proof −
  let *?args* = (([]::('*t* $\Rightarrow_0$ '*b*) *list*, *Koszul-syz-sigs fs*,
    (*map Inr* [*0*..<*length fs*])::((('*t* $\Rightarrow_0$ '*b*) × ('*t* $\Rightarrow_0$ '*b*)) + *nat*) *list*), *z*)
  from *rb-aux-inv-init* **have** *rb-aux-inv2* (*fst ?args*) **by** *simp*
  with *assms* **have** *snd* (*rb-aux ?args*) = *snd ?args* **by** (*rule rb-aux-no-zero-red'*)
  thus *?thesis* **by** (*simp only*: *snd-conv*)
qed

corollary *rb-no-zero-red*:
  assumes *hom-grading dgrad* **and** *is-regular-sequence fs*
  shows *snd rb* = *0*
  using *rb-aux-no-zero-red*[*OF assms*, *of 0*] **by** (*auto simp*: *rb-def split*: *prod.split*)

end

4.3 Sig-Poly-Pairs

We now prove that the algorithms defined for sig-poly-pairs (i. e. those whose names end with *-spp*) behave exactly as those defined for module elements. More precisely, if *A* is some algorithm defined for module elements, we prove something like *spp-of* (*A x*) = *A-spp* (*spp-of x*).

fun *spp-inv-pair* :: ((('*t* × ('*a* $\Rightarrow_0$ '*b*)) × ('*t* × ('*a* $\Rightarrow_0$ '*b*))) + *nat*) ⇒ *bool* **where**
  spp-inv-pair (*Inl* (*p*, *q*)) = (*spp-inv p* ∧ *spp-inv q*) |
  spp-inv-pair (*Inr j*) = *True*

fun *app-pair* :: ('*x* ⇒ '*y*) ⇒ (('*x* × '*x*) + *nat*) ⇒ (('*y* × '*y*) + *nat*) **where**
  app-pair f (*Inl* (*p*, *q*)) = *Inl* (*f p*, *f q*) |
  app-pair f (*Inr j*) = *Inr j*

fun *app-args* :: ('*x* ⇒ '*y*) ⇒ (('*x list* × '*z* × ((('*x* × '*x*) + *nat*) *list*)) × *nat*) ⇒
    (('*y list* × '*z* × ((('*y* × '*y*) + *nat*) *list*)) × *nat*) **where**
  app-args f ((*as*, *bs*, *cs*), *n*) = ((*map f as*, *bs*, *map* (*app-pair f*) *cs*), *n*)

lemma *app-pair-spp-of-vec-of*:
  assumes *spp-inv-pair p*
  shows *app-pair spp-of* (*app-pair vec-of p*) = *p*
proof (*rule sum-prodE*)
  fix *a b*
  assume *p*: *p* = *Inl* (*a*, *b*)
  from *assms* **have** *spp-inv a* **and** *spp-inv b* **by** (*simp-all add*: *p*)
  thus *?thesis* **by** (*simp add*: *p spp-of-vec-of*)
qed *simp*

lemma *map-app-pair-spp-of-vec-of*:
  assumes *list-all spp-inv-pair ps*
  shows *map* (*app-pair spp-of* ∘ *app-pair vec-of*) *ps* = *ps*
proof (*rule map-idI*)
  fix *p*
  assume *p* ∈ *set ps*
  with *assms* **have** *spp-inv-pair p* **by** (*simp add*: *list-all-def*)
  hence *app-pair spp-of* (*app-pair vec-of p*) = *p* **by** (*rule app-pair-spp-of-vec-of*)
  thus (*app-pair spp-of* ∘ *app-pair vec-of*) *p* = *p* **by** *simp*
qed

lemma *snd-app-args*: *snd* (*app-args f args*) = *snd args*
  by (*metis prod.exhaust app-args.simps snd-conv*)

lemma *new-syz-sigs-alt-spp*:
  new-syz-sigs ss bs p = *new-syz-sigs-spp ss* (*map spp-of bs*) (*app-pair spp-of p*)
proof (*rule sum-prodE*)
  fix *a b*
  assume *p* = *Inl* (*a*, *b*)
  thus *?thesis* **by** *simp*
next
  fix *j*
  assume *p* = *Inr j*
  thus *?thesis* **by** (*simp add*: *comp-def spp-of-def*)
qed

lemma *is-rewritable-alt-spp*:
  assumes *0* ∉ *set bs*
  shows *is-rewritable bs p u* = *is-rewritable-spp* (*map spp-of bs*) (*spp-of p*) *u*
proof −
  from *assms* **have** *b* ∈ *set bs* ⟹ *b* ≠ *0* **for** *b* **by** *blast*
  thus *?thesis* **by** (*auto simp*: *is-rewritable-def is-rewritable-spp-def fst-spp-of*)
qed

lemma *spair-sigs-alt-spp*: *spair-sigs p q* = *spair-sigs-spp* (*spp-of p*) (*spp-of q*)
  by (*simp add*: *spair-sigs-def spair-sigs-spp-def Let-def fst-spp-of snd-spp-of*)

lemma *sig-crit-alt-spp*:
  assumes *0* ∉ *set bs*
  shows *sig-crit bs ss p* = *sig-crit-spp* (*map spp-of bs*) *ss* (*app-pair spp-of p*)
proof (*rule sum-prodE*)
  fix *a b*
  assume *p*: *p* = *Inl* (*a*, *b*)
  from *assms* **show** *?thesis* **by** (*simp add*: *p spair-sigs-alt-spp is-rewritable-alt-spp*)
qed *simp*

lemma *spair-alt-spp*:
  assumes *is-regular-spair p q*
  shows *spp-of* (*spair p q*) = *spair-spp* (*spp-of p*) (*spp-of q*)

proof −
  let *?t1* = *punit.lt* (*rep-list p*)
  let *?t2* = *punit.lt* (*rep-list q*)
  let *?l* = *lcs ?t1 ?t2*
  from *assms* **have** *p*: *rep-list p* ≠ *0* **and** *q*: *rep-list q* ≠ *0*
    by (*rule is-regular-spairD1*, *rule is-regular-spairD2*)
  hence *p* ≠ *0* **and** *q* ≠ *0* **and** *1*: *punit.lc* (*rep-list p*) ≠ *0* **and** *2*: *punit.lc* (*rep-list q*) ≠ *0*
    by (*auto simp*: *rep-list-zero punit.lc-eq-zero-iff*)
  from *assms* **have** *lt* (*monom-mult* (*1* / *punit.lc* (*rep-list p*)) (*?l* − *?t1*) *p*) ≠
      lt (*monom-mult* (*1* / *punit.lc* (*rep-list q*)) (*?l* − *?t2*) *q*) (**is** *?u* ≠ *?v*)
    by (*rule is-regular-spairD3*)
  hence *lt* (*monom-mult* (*1* / *punit.lc* (*rep-list p*)) (*?l* − *?t1*) *p* − *monom-mult* (*1* / *punit.lc* (*rep-list q*)) (*?l* − *?t2*) *q*) =
      ord-term-lin.max ?u ?v **by** (*rule lt-minus-distinct-eq-max*)
  moreover from ‹*p* ≠ *0*› *1* **have** *?u* = (*?l* − *?t1*) ⊕ *fst* (*spp-of p*) **by** (*simp add*: *lt-monom-mult fst-spp-of*)
  moreover from ‹*q* ≠ *0*› *2* **have** *?v* = (*?l* − *?t2*) ⊕ *fst* (*spp-of q*) **by** (*simp add*: *lt-monom-mult fst-spp-of*)
  ultimately show *?thesis*
    by (*simp add*: *spair-spp-def spair-def Let-def spp-of-def rep-list-minus rep-list-monom-mult*)
qed

lemma *sig-trd-spp-body-alt-Some*:
  assumes *find-sig-reducer* (*map spp-of bs*) *v* (*punit.lt p*) *0* = *Some i*
  shows *sig-trd-spp-body* (*map spp-of bs*) *v* (*p*, *r*) =
      (*punit.lower* (*p* − *local.punit.monom-mult* (*punit.lc p* / *punit.lc* (*rep-list* (*bs* ! *i*)))
        (*punit.lt p* − *punit.lt* (*rep-list* (*bs* ! *i*))) (*rep-list* (*bs* ! *i*))) (*punit.lt p*), *r*)
      (**is** *?thesis1*)
    and *sig-trd-spp-body* (*map spp-of bs*) *v* (*p*, *r*) =
      (*p* − *local.punit.monom-mult* (*punit.lc p* / *punit.lc* (*rep-list* (*bs* ! *i*)))
        (*punit.lt p* − *punit.lt* (*rep-list* (*bs* ! *i*))) (*rep-list* (*bs* ! *i*)), *r*)
      (**is** *?thesis2*)
proof −
  have *?thesis1* ∧ *?thesis2*
  proof (*cases p* = *0*)
    case *True*
    show *?thesis* **by** (*simp add*: *assms*, *simp add*: *True*)
  next
    case *False*
    from *assms* **have** *i* < *length bs* **by** (*rule find-sig-reducer-SomeD*)
    hence *eq1*: *snd* (*map spp-of bs* ! *i*) = *rep-list* (*bs* ! *i*) **by** (*simp add*: *snd-spp-of*)
    from *assms* **have** *rep-list* (*bs* ! *i*) ≠ *0* **and** *punit.lt* (*rep-list* (*bs* ! *i*)) *adds punit.lt p*
      by (*rule find-sig-reducer-SomeD*)+
    hence *nz*: *rep-list* (*bs* ! *i*) ≠ *0* **and** *adds*: *punit.lt* (*rep-list* (*bs* ! *i*)) *adds punit.lt*

p
by (*simp-all add*: *snd-spp-of*)
from *nz* have *punit.lc* (*rep-list* (*bs* ! *i*)) $\neq$ *0* by (*rule punit.lc-not-0*)
moreover from *False* have *punit.lc p* $\neq$ *0* by (*rule punit.lc-not-0*)
ultimately have *eq2*: *punit.lt* (*punit.monom-mult* (*punit.lc p* / *punit.lc* (*rep-list* (*bs* ! *i*)))
(*punit.lt p* − *punit.lt* (*rep-list* (*bs* ! *i*))) (*rep-list* (*bs* ! *i*))) = *punit.lt p*
(is *punit.lt ?p* = -) using *nz adds* by (*simp add*: *lp-monom-mult adds-minus*)
have *?thesis1* by (*simp add*: *assms Let-def eq1 punit.lower-minus punit.tail-monom-mult*[*symmetric*],
simp add: *punit.tail-def eq2*)
moreover have *?thesis2*
proof (*simp add*: ‹*?thesis1*› *punit.lower-id-iff disj-commute*[*of p* = *?p*] *del*: *sig-trd-spp-body.simps*)
show *punit.lt* (*p* − *?p*) ≺ *punit.lt p* ∨ *p* = *?p*
proof (*rule disjCI*)
assume *p* $\neq$ *?p*
hence *p* − *?p* $\neq$ *0* by *simp*
moreover note *eq2*
moreover from ‹*punit.lc* (*rep-list* (*bs* ! *i*)) $\neq$ *0*› have *punit.lc ?p* = *punit.lc p* by *simp*
ultimately show *punit.lt* (*p* − *?p*) ≺ *punit.lt p* by (*rule punit.lt-minus-lessI*)
qed
qed
ultimately show *?thesis* ..
qed
thus *?thesis1* and *?thesis2* by *blast+*
qed

lemma *sig-trd-aux-alt-spp*:
assumes *fst args* ∈ *keys* (*rep-list* (*snd args*))
shows *rep-list* (*sig-trd-aux bs args*) =
sig-trd-spp-aux (*map spp-of bs*) (*lt* (*snd args*))
(*rep-list* (*snd args*) − *punit.higher* (*rep-list* (*snd args*)) (*fst args*),
punit.higher (*rep-list* (*snd args*)) (*fst args*))
proof −
from *assms* have *sig-trd-aux-dom bs args* by (*rule sig-trd-aux-domI*)
thus *?thesis* using *assms*
proof (*induct args rule*: *sig-trd-aux.pinduct*)
case (*1 t p*)
define *p*′ where *p*′ = (*case find-sig-reducer* (*map spp-of bs*) (*lt p*) *t 0 of*
None ⇒ *p*
| *Some i* ⇒ *p* −
monom-mult (*lookup* (*rep-list p*) *t* / *punit.lc* (*rep-list* (*bs* ! *i*)))
(*t* − *punit.lt* (*rep-list* (*bs* ! *i*))) (*bs* ! *i*))
define *p*″ where *p*″ = *punit.lower* (*rep-list p*′) *t*
from *1*(*3*) have *t-in*: *t* ∈ *keys* (*rep-list p*) by *simp*
hence *t* ∈ *keys* (*rep-list p* − *punit.higher* (*rep-list p*) *t*) (is - ∈ *keys ?p*)

by (*simp add*: *punit.keys-minus-higher*)
hence *?p* $\neq$ *0* **by** *auto*
hence *eq1*: *sig-trd-spp-aux bs0 v0* (*?p*, *r0*) = *sig-trd-spp-aux bs0 v0* (*sig-trd-spp-body bs0 v0* (*?p*, *r0*))
for *bs0 v0 r0* **by** (*simp add*: *sig-trd-spp-aux-simps del*: *sig-trd-spp-body.simps*)
from *t-in* **have** *lt-p*: *punit.lt ?p* = *t* **and** *lc-p*: *punit.lc ?p* = *lookup* (*rep-list p*) *t*
and *tail-p*: *punit.tail ?p* = *punit.lower* (*rep-list p*) *t*
by (*rule punit.lt-minus-higher*, *rule punit.lc-minus-higher*, *rule punit.tail-minus-higher*)
have *lt p'* = *lt p* $\wedge$ *punit.higher* (*rep-list p'*) *t* = *punit.higher* (*rep-list p*) *t* $\wedge$
($\forall$ *i*. *find-sig-reducer* (*map spp-of bs*) (*lt p*) *t 0* = *Some i* $\longrightarrow$ *lookup* (*rep-list p'*) *t* = *0*)
(**is** *?A* $\wedge$ *?B* $\wedge$ *?C*)
proof (*cases find-sig-reducer* (*map spp-of bs*) (*lt p*) *t 0*)
case *None*
thus *?thesis* **by** (*simp add*: *p'-def*)
next
case (*Some i*)
hence *p'*: *p'* = *p* − *monom-mult* (*lookup* (*rep-list p*) *t* / *punit.lc* (*rep-list* (*bs* ! *i*)))
(*t* − *punit.lt* (*rep-list* (*bs* ! *i*))) (*bs* ! *i*) **by** (*simp add*: *p'-def*)
from *Some* **have** *punit.lt* (*rep-list* (*bs* ! *i*)) *adds t* **by** (*rule find-sig-reducer-SomeD*)
hence *eq*: *t* − *punit.lt* (*rep-list* (*bs* ! *i*)) + *punit.lt* (*rep-list* (*bs* ! *i*)) = *t* **by** (*rule adds-minus*)
from *t-in Some* **have** ∗: *sig-red-single* ($\prec_t$) ($\preceq$) *p p'* (*bs* ! *i*) (*t* − *punit.lt* (*rep-list* (*bs* ! *i*)))
unfolding *p'* **by** (*rule find-sig-reducer-SomeD-red-single*)
hence ∗∗: *punit.red-single* (*rep-list p*) (*rep-list p'*) (*rep-list* (*bs* ! *i*)) (*t* − *punit.lt* (*rep-list* (*bs* ! *i*)))
by (*rule sig-red-single-red-single*)
from ∗ **have** *?A* **by** (*rule sig-red-single-regular-lt*)
moreover from *punit.red-single-higher*[*OF* ∗∗] **have** *?B* **by** (*simp add*: *eq*)
moreover have *?C*
proof (*intro allI impI*)
from *punit.red-single-lookup*[*OF* ∗∗] **show** *lookup* (*rep-list p'*) *t* = *0* **by** (*simp add*: *eq*)
qed
ultimately show *?thesis* **by** (*intro conjI*)
qed
hence *lt-p'*: *lt p'* = *lt p* **and** *higher-p'*: *punit.higher* (*rep-list p'*) *t* = *punit.higher* (*rep-list p*) *t*
and *lookup-p'*: $\bigwedge$*i*. *find-sig-reducer* (*map spp-of bs*) (*lt p*) *t 0* = *Some i* $\Longrightarrow$ *lookup* (*rep-list p'*) *t* = *0*
by *blast+*
show *?case*
proof (*simp add*: *sig-trd-aux.psimps*[*OF 1*(*1*)] *Let-def p'-def*[*symmetric*] *p''-def*[*symmetric*], *intro conjI impI*)
assume *p''* = *0*
hence *p'-decomp*: *punit.higher* (*rep-list p*) *t* + *monomial* (*lookup* (*rep-list p'*)

t) t = *rep-list* p'
using *punit.higher-lower-decomp*[*of rep-list* p' t] **by** (*simp add*: p''*-def higher-p*$'$)
show *rep-list* p' = *sig-trd-spp-aux* (*map spp-of bs*) (*lt p*) (*?p*, *punit.higher* (*rep-list p*) t)
proof (*cases find-sig-reducer* (*map spp-of bs*) (*lt p*) t *0*)
case *None*
hence p': $p' = p$ **by** (*simp add*: p'*-def*)
from ‹p'' = *0*› **have** *eq2*: *punit.tail ?p* = *0* **by** (*simp add*: *tail-p* p''*-def* p')
from p'*-decomp* **show** *?thesis* **by** (*simp add*: p' *eq1 lt-p lc-p None eq2 sig-trd-spp-aux-simps*)
next
case (*Some i*)
hence p': $p' = p$ − *monom-mult* (*lookup* (*rep-list p*) t / *punit.lc* (*rep-list* (*bs* ! *i*)))
(t − *punit.lt* (*rep-list* (*bs* ! *i*))) (*bs* ! *i*) **by** (*simp add*: p'*-def*)
from ‹p'' = *0*› **have** *eq2*: *punit.lower* (*rep-list p* − *punit.higher* (*rep-list p*) t −
punit.monom-mult (*lookup* (*rep-list p*) t / *punit.lc* (*rep-list* (*bs* ! *i*)))
(t − *punit.lt* (*rep-list* (*bs* ! *i*))) (*rep-list* (*bs* ! *i*)))
t = *0*
by (*simp add*: p''*-def* p' *rep-list-minus rep-list-monom-mult punit.lower-minus punit.lower-higher-zeroI*)
from *Some* **have** *lookup* (*rep-list* p') t = *0* **by** (*rule lookup-p*$'$)
with p'*-decomp* **have** *eq3*: *rep-list* p' = *punit.higher* (*rep-list p*) t **by** *simp*
show *?thesis* **by** (*simp add*: *sig-trd-spp-body-alt-Some*(*1*) *eq1 eq2 lt-p lc-p Some del*: *sig-trd-spp-body.simps*,
simp add: *sig-trd-spp-aux-simps eq3*)
qed
next
assume $p'' \neq$ *0*
hence *punit.lt* $p'' \prec t$ **unfolding** p''*-def* **by** (*rule punit.lt-lower-less*)
have *higher-p*$'$*-2*: *punit.higher* (*rep-list* p') (*punit.lt* p'') =
punit.higher (*rep-list p*) t + *monomial* (*lookup* (*rep-list* p') t) t
proof (*simp add*: *higher-p*$'$[*symmetric*], *rule poly-mapping-eqI*)
fix s
show *lookup* (*punit.higher* (*rep-list* p') (*punit.lt* p'')) s =
lookup (*punit.higher* (*rep-list* p') t + *monomial* (*lookup* (*rep-list* p') t) t) s
proof (*rule ordered-powerprod-lin.linorder-cases*)
assume $t \prec s$
moreover from ‹*punit.lt* $p'' \prec t$› *this* **have** *punit.lt* $p'' \prec s$
by (*rule ordered-powerprod-lin.less-trans*)
ultimately show *?thesis* **by** (*simp add*: *lookup-add punit.lookup-higher-when lookup-single*)
next
assume $t = s$
with ‹*punit.lt* $p'' \prec t$› **show** *?thesis* **by** (*simp add*: *lookup-add punit.lookup-higher-when*)
next

assume $s \prec t$
show *?thesis*
proof (*cases punit.lt* $p'' \prec s$)
case *True*
hence *lookup* (*punit.higher* (*rep-list* p') (*punit.lt* p'')) $s =$ *lookup* (*rep-list* p') s
by (*simp add*: *punit.lookup-higher-when*)
also from ‹$s \prec t$› **have** ... = *lookup* p'' s **by** (*simp add*: p''*-def punit.lookup-lower-when*)
also from *True* **have** ... = *0* **using** *punit.lt-le-iff* **by** *auto*
finally show *?thesis* **using** ‹$s \prec t$›
by (*simp add*: *lookup-add lookup-single punit.lookup-higher-when*)
next
case *False*
with ‹$s \prec t$› **show** *?thesis* **by** (*simp add*: *lookup-add punit.lookup-higher-when lookup-single*)
qed
qed
qed
have *rep-list* (*sig-trd-aux bs* (*punit.lt* p'', p')) =
sig-trd-spp-aux (*map spp-of bs*) (*lt* (*snd* (*punit.lt* p'', p')))
(*rep-list* (*snd* (*punit.lt* p'', p')) −
punit.higher (*rep-list* (*snd* (*punit.lt* p'', p'))) (*fst* (*punit.lt* p'', p')),
punit.higher (*rep-list* (*snd* (*punit.lt* p'', p'))) (*fst* (*punit.lt* p'', p')))
using p'*-def* p''*-def* ‹$p'' \neq 0$›
proof (*rule 1*(*2*))
from ‹$p'' \neq 0$› **have** *punit.lt* $p'' \in$ *keys* p'' **by** (*rule punit.lt-in-keys*)
also have ... ⊆ *keys* (*rep-list* p') **by** (*auto simp*: p''*-def punit.keys-lower*)
finally show *fst* (*punit.lt* p'', p') ∈ *keys* (*rep-list* (*snd* (*punit.lt* p'', p'))) **by** *simp*
qed
also have ... = *sig-trd-spp-aux* (*map spp-of bs*) (*lt p*)
(*rep-list* p' − *punit.higher* (*rep-list* p') (*punit.lt* p''),
punit.higher (*rep-list* p') (*punit.lt* p''))
by (*simp only*: *lt-*p' *fst-conv snd-conv*)
also have ... = *sig-trd-spp-aux* (*map spp-of bs*) (*lt p*) (*?p*, *punit.higher* (*rep-list p*) *t*)
proof (*cases find-sig-reducer* (*map spp-of bs*) (*lt p*) *t 0*)
case *None*
hence p': $p' = p$ **by** (*simp add*: p'*-def*)
have *rep-list p* − (*punit.higher* (*rep-list p*) *t* + *monomial* (*lookup* (*rep-list p*) *t*) *t*) =
punit.lower (*rep-list p*) *t*
using *punit.higher-lower-decomp*[*of rep-list p t*] **by** (*simp add*: *diff-eq-eq ac-simps*)
with *higher-*p'*-2* **show** *?thesis* **by** (*simp add*: *eq1 lt-p lc-p tail-p* p' *None*)
next
case (*Some i*)
hence p': *rep-list p* − *punit.monom-mult* (*lookup* (*rep-list p*) *t* / *punit.lc*

(*rep-list* (*bs* ! *i*)))
(*t* − *punit.lt* (*rep-list* (*bs* ! *i*))) (*rep-list* (*bs* ! *i*)) = *rep-list* *p*′
by (*simp add*: *p*′*-def rep-list-minus rep-list-monom-mult*)
from *Some* **have** *lookup* (*rep-list* *p*′) *t* = *0* **by** (*rule lookup-p*′)
with *higher-p*′*-2* **show** *?thesis*
by (*simp add*: *sig-trd-spp-body-alt-Some*(*2*) *eq1 lt-p lc-p tail-p Some diff-right-commute*[*of rep-list p punit.higher* (*rep-list p*) *t*] *p*′ *del*: *sig-trd-spp-body.simps*)
qed
finally show *rep-list* (*sig-trd-aux bs* (*punit.lt p*′′, *p*′)) =
sig-trd-spp-aux (*map spp-of bs*) (*lt p*) (*?p*, *punit.higher* (*rep-list p*) *t*) **.**
qed
qed
qed

lemma *sig-trd-alt-spp*: *spp-of* (*sig-trd bs p*) = *sig-trd-spp* (*map spp-of bs*) (*spp-of p*)
unfolding *sig-trd-def*
proof (*split if-split*, *intro conjI impI*)
assume *rep-list p* = *0*
thus *spp-of p* = *sig-trd-spp* (*map spp-of bs*) (*spp-of p*) **by** (*simp add*: *spp-of-def sig-trd-spp-aux-simps*)
next
let *?args* = (*punit.lt* (*rep-list p*), *p*)
assume *rep-list p* ≠ *0*
hence *a*: *fst ?args* ∈ *keys* (*rep-list* (*snd ?args*)) **by** (*simp add*: *punit.lt-in-keys*)
hence (*sig-red* ($\prec_t$) ($\preceq$) (*set bs*))** (*snd ?args*) (*sig-trd-aux bs ?args*)
by (*rule sig-trd-aux-red-rtrancl*)
hence *eq1*: *lt* (*sig-trd-aux bs ?args*) = *lt* (*snd ?args*) **by** (*rule sig-red-regular-rtrancl-lt*)
have *eq2*: *punit.higher* (*rep-list p*) (*punit.lt* (*rep-list p*)) = *0*
by (*auto simp*: *punit.higher-eq-zero-iff punit.lt-max simp flip*: *not-in-keys-iff-lookup-eq-zero dest*: *punit.lt-max-keys*)
show *spp-of* (*sig-trd-aux bs* (*punit.lt* (*rep-list p*), *p*)) = *sig-trd-spp* (*map spp-of bs*) (*spp-of p*)
by (*simp add*: *spp-of-def eq1 eq2 sig-trd-aux-alt-spp*[*OF a*])
qed

lemma *is-regular-spair-alt-spp*: *is-regular-spair p q* ⟷ *is-regular-spair-spp* (*spp-of p*) (*spp-of q*)
by (*auto simp*: *is-regular-spair-spp-def fst-spp-of snd-spp-of intro*: *is-regular-spairI dest*: *is-regular-spairD1 is-regular-spairD2 is-regular-spairD3*)

lemma *sig-of-spair-alt-spp*: *sig-of-pair p* = *sig-of-pair-spp* (*app-pair spp-of p*)
proof (*rule sum-prodE*)
fix *a b*
assume *p*: *p* = *Inl* (*a*, *b*)
show *?thesis* **by** (*simp add*: *p spair-sigs-def spair-sigs-spp-def spp-of-def*)
qed *simp*

lemma *pair-ord-alt-spp*: *pair-ord x y* $\longleftrightarrow$ *pair-ord-spp* (*app-pair spp-of x*) (*app-pair spp-of y*)
  by (*simp add*: *pair-ord-spp-def pair-ord-def sig-of-spair-alt-spp*)

lemma *new-spairs-alt-spp*:
  map (*app-pair spp-of*) (*new-spairs bs p*) = *new-spairs-spp* (*map spp-of bs*) (*spp-of p*)
proof (*induct bs*)
  case *Nil*
  show *?case* **by** *simp*
next
  case (*Cons b bs*)
  have *map* (*app-pair spp-of*) (*insort-wrt pair-ord* (*Inl* (*p*, *b*)) (*new-spairs bs p*)) =
      insort-wrt pair-ord-spp (*app-pair spp-of* (*Inl* (*p*, *b*))) (*map* (*app-pair spp-of*) (*new-spairs bs p*))
    by (*rule map-insort-wrt*, *rule pair-ord-alt-spp*[*symmetric*])
  thus *?case* **by** (*simp add*: *is-regular-spair-alt-spp Cons*)
qed

lemma *add-spairs-alt-spp*:
  assumes $\bigwedge x.\ x \in$ *set bs* $\Longrightarrow$ *Inl* (*spp-of p*, *spp-of x*) $\notin$ *app-pair spp-of* ‘ *set ps*
  shows *map* (*app-pair spp-of*) (*add-spairs ps bs p*) =
      add-spairs-spp (*map* (*app-pair spp-of*) *ps*) (*map spp-of bs*) (*spp-of p*)
proof −
  have *map* (*app-pair spp-of*) (*merge-wrt pair-ord* (*new-spairs bs p*) *ps*) =
      merge-wrt pair-ord-spp (*map* (*app-pair spp-of*) (*new-spairs bs p*)) (*map* (*app-pair spp-of*) *ps*)
  proof (*rule map-merge-wrt*, *rule ccontr*)
    assume *app-pair spp-of* ‘ *set* (*new-spairs bs p*) $\cap$ *app-pair spp-of* ‘ *set ps* $\neq$ {}
    then obtain q' **where** $q' \in$ *app-pair spp-of* ‘ *set* (*new-spairs bs p*)
      and q'*-in*: $q' \in$ *app-pair spp-of* ‘ *set ps* **by** *blast*
    from *this*(*1*) **obtain** *q* **where** $q \in$ *set* (*new-spairs bs p*) **and** q': q' = *app-pair spp-of q* **..**
    from *this*(*1*) **obtain** *x* **where** *x-in*: $x \in$ *set bs* **and** *q*: *q* = *Inl* (*p*, *x*)
      by (*rule in-new-spairsE*)
    have q': q' = *Inl* (*spp-of p*, *spp-of x*) **by** (*simp add*: *q* q')
    have $q' \notin$ *app-pair spp-of* ‘ *set ps* **unfolding** q' **using** *x-in* **by** (*rule assms*)
    thus *False* **using** q'*-in* **..**
  qed (*simp only*: *pair-ord-alt-spp*)
  thus *?thesis* **by** (*simp add*: *add-spairs-def add-spairs-spp-def new-spairs-alt-spp*)
qed

lemma *rb-aux-invD-app-args*:
  assumes *rb-aux-inv* (*fst* (*app-args vec-of* ((*bs*, *ss*, *ps*), *z*)))
  shows *list-all spp-inv bs* **and** *list-all spp-inv-pair ps*
proof −
  from *assms*(*1*) **have** *inv*: *rb-aux-inv* (*map vec-of bs*, *ss*, *map* (*app-pair vec-of*)

ps) **by** *simp*
hence *rb-aux-inv1* (*map vec-of bs*) **by** (*rule rb-aux-inv-D1*)
hence *0* ∉ *rep-list* ‘ *set* (*map vec-of bs*) **by** (*rule rb-aux-inv1-D2*)
hence *0* ∉ *vec-of* ‘ *set bs* **using** *rep-list-zero* **by** *fastforce*
hence *1*: *b* ∈ *set bs* ⟹ *spp-inv b* **for** *b* **by** (*auto simp*: *spp-inv-alt*)
thus *list-all spp-inv bs* **by** (*simp add*: *list-all-def*)

have *2*: *x* ∈ *set bs* **if** *vec-of x* ∈ *set* (*map vec-of bs*) **for** *x*
proof −
from *that* **have** *vec-of x* ∈ *vec-of* ‘ *set bs* **by** *simp*
then obtain *y* **where** *y* ∈ *set bs* **and** *eq*: *vec-of x* = *vec-of y* **..**
from *this*(*1*) **have** *spp-inv y* **by** (*rule 1*)
moreover have *vec-of y* = *vec-of x* **by** (*simp only*: *eq*)
ultimately have *y* = *x* **by** (*rule vec-of-inj*)
with ‹*y* ∈ *set bs*› **show** *?thesis* **by** *simp*
qed

show *list-all spp-inv-pair ps* **unfolding** *list-all-def*
proof (*rule ballI*)
fix *p*
assume *p* ∈ *set ps*
show *spp-inv-pair p*
proof (*rule sum-prodE*)
fix *a b*
assume *p*: *p* = *Inl* (*a*, *b*)
from ‹*p* ∈ *set ps*› **have** *Inl* (*a*, *b*) ∈ *set ps* **by** (*simp only*: *p*)
hence *app-pair vec-of* (*Inl* (*a*, *b*)) ∈ *app-pair vec-of* ‘ *set ps* **by** (*rule imageI*)
hence *Inl* (*vec-of a*, *vec-of b*) ∈ *set* (*map* (*app-pair vec-of*) *ps*) **by** *simp*
with *inv* **have** *vec-of a* ∈ *set* (*map vec-of bs*) **and** *vec-of b* ∈ *set* (*map vec-of bs*)
by (*rule rb-aux-inv-D3*)+
have *spp-inv a* **by** (*rule 1*, *rule 2*, *fact*)
moreover have *spp-inv b* **by** (*rule 1*, *rule 2*, *fact*)
ultimately show *?thesis* **by** (*simp add*: *p*)
qed *simp*
qed
qed

lemma *app-args-spp-of-vec-of*:
assumes *rb-aux-inv* (*fst* (*app-args vec-of args*))
shows *app-args spp-of* (*app-args vec-of args*) = *args*
proof −
obtain *bs ss ps z* **where** *args*: *args* = ((*bs*, *ss*, *ps*), *z*) **using** *prod.exhaust* **by** *metis*
from *assms* **have** *list-all spp-inv bs* **and** ∗: *list-all spp-inv-pair ps* **unfolding** *args*
by (*rule rb-aux-invD-app-args*)+
from *this*(*1*) **have** *map* (*spp-of* ∘ *vec-of*) *bs* = *bs* **by** (*rule map-spp-of-vec-of*)
moreover from ∗ **have** *map* (*app-pair spp-of* ∘ *app-pair vec-of*) *ps* = *ps*

by (*rule map-app-pair-spp-of-vec-of*)
ultimately show *?thesis* **by** (*simp add*: *args*)
qed

lemma *poly-of-pair-alt-spp*:
assumes *distinct fs* **and** *rb-aux-inv* (*bs*, *ss*, *p* # *ps*)
shows *spp-of* (*poly-of-pair p*) = *spp-of-pair* (*app-pair spp-of p*)
proof −
show *?thesis*
proof (*rule sum-prodE*)
fix *a b*
assume *p*: *p* = *Inl* (*a*, *b*)
hence *Inl* (*a*, *b*) ∈ *set* (*p* # *ps*) **by** *simp*
with *assms*(*2*) **have** *is-regular-spair a b* **by** (*rule rb-aux-inv-D3*)
thus *?thesis* **by** (*simp add*: *p spair-alt-spp*)
next
fix *j*
assume *p*: *p* = *Inr j*
hence *Inr j* ∈ *set* (*p* # *ps*) **by** *simp*
with *assms*(*2*) **have** *j* < *length fs* **by** (*rule rb-aux-inv-D4*)
thus *?thesis* **by** (*simp add*: *p spp-of-def lt-monomial rep-list-monomial*[*OF assms*(*1*)] *term-simps*)
qed
qed

lemma *rb-aux-alt-spp*:
assumes *rb-aux-inv* (*fst args*)
shows *app-args spp-of* (*rb-aux args*) = *rb-spp-aux* (*app-args spp-of args*)
proof −
from *assms* **have** *rb-aux-dom args* **by** (*rule rb-aux-domI*)
thus *?thesis* **using** *assms*
proof (*induct args rule*: *rb-aux.pinduct*)
case (*1 bs ss z*)
show *?case* **by** (*simp add*: *rb-aux.psimps*(*1*)[*OF 1*(*1*)] *rb-spp-aux-Nil*)
next
case (*2 bs ss p ps z*)
let *?q* = *sig-trd bs* (*poly-of-pair p*)

from *2*(*5*) **have** ∗: *rb-aux-inv* (*bs*, *ss*, *p* # *ps*) **by** (*simp only*: *fst-conv*)
hence *rb-aux-inv1 bs* **by** (*rule rb-aux-inv-D1*)
hence *0* ∉ *rep-list* ‘ *set bs* **by** (*rule rb-aux-inv1-D2*)
hence *0* ∉ *set bs* **by** (*force simp*: *rep-list-zero*)
hence *eq1*: *sig-crit-spp* (*map spp-of bs*) *ss′* (*app-pair spp-of p*) ⟷ *sig-crit bs ss′ p* **for** *ss′*
by (*simp add*: *sig-crit-alt-spp*)
from *fs-distinct* ∗ **have** *eq2*: *sig-trd-spp* (*map spp-of bs*) (*spp-of-pair* (*app-pair spp-of p*)) = *spp-of ?q*
by (*simp only*: *sig-trd-alt-spp poly-of-pair-alt-spp*)

show *?case*
proof (*simp add*: *rb-aux.psimps*(*2*)[*OF 2*(*1*)] *Let-def*, *intro conjI impI*)
note *refl*
moreover assume *a*: *sig-crit bs* (*new-syz-sigs ss bs p*) *p*
moreover from ∗ *this* **have** *rb-aux-inv* (*fst* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*))
unfolding *fst-conv* **by** (*rule rb-aux-inv-preserved-1*)
ultimately have *app-args spp-of* (*rb-aux* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*)) =
rb-spp-aux (*app-args spp-of* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*))
by (*rule 2*(*2*))
also have ... = *rb-spp-aux* ((*map spp-of bs*, *ss*, *app-pair spp-of p* # *map* (*app-pair spp-of*) *ps*), *z*)
by (*simp add*: *rb-spp-aux-Cons eq1 a new-syz-sigs-alt-spp*[*symmetric*])
finally show *app-args spp-of* (*rb-aux* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*)) =
rb-spp-aux ((*map spp-of bs*, *ss*, *app-pair spp-of p* # *map* (*app-pair spp-of*) *ps*), *z*) **.**
thus *app-args spp-of* (*rb-aux* ((*bs*, *new-syz-sigs ss bs p*, *ps*), *z*)) =
rb-spp-aux ((*map spp-of bs*, *ss*, *app-pair spp-of p* # *map* (*app-pair spp-of*) *ps*), *z*) **.**
next
assume *a*: ¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p* **and** *b*: *rep-list ?q* = *0*
from ∗ *b* **have** *rb-aux-inv* (*fst* ((*bs*, *lt ?q* # *new-syz-sigs ss bs p*, *ps*), *Suc z*))
unfolding *fst-conv* **by** (*rule rb-aux-inv-preserved-2*)
with *refl a refl b* **have** *app-args spp-of* (*rb-aux* ((*bs*, *lt ?q* # *new-syz-sigs ss bs p*, *ps*), *Suc z*)) =
rb-spp-aux (*app-args spp-of* ((*bs*, *lt ?q* # *new-syz-sigs ss bs p*, *ps*), *Suc z*))
by (*rule 2*(*3*))
also have ... = *rb-spp-aux* ((*map spp-of bs*, *ss*, *app-pair spp-of p* # *map* (*app-pair spp-of*) *ps*), *z*)
by (*simp add*: *rb-spp-aux-Cons eq1 a Let-def eq2 snd-spp-of b fst-spp-of new-syz-sigs-alt-spp*[*symmetric*])
finally show *app-args spp-of* (*rb-aux* ((*bs*, *lt ?q* # *new-syz-sigs ss bs p*, *ps*), *Suc z*)) =
rb-spp-aux ((*map spp-of bs*, *ss*, *app-pair spp-of p* # *map* (*app-pair spp-of*) *ps*), *z*) **.**
next
assume *a*: ¬ *sig-crit bs* (*new-syz-sigs ss bs p*) *p* **and** *b*: *rep-list ?q* ≠ *0*

have *Inl* (*spp-of ?q*, *spp-of x*) ∉ *app-pair spp-of* ‘ *set ps* **for** *x*
proof
assume *Inl* (*spp-of ?q*, *spp-of x*) ∈ *app-pair spp-of* ‘ *set ps*
then obtain *y* **where** *y* ∈ *set ps* **and** *eq0*: *Inl* (*spp-of ?q*, *spp-of x*) = *app-pair spp-of y* **..**
obtain *a b* **where** *y*: *y* = *Inl* (*a*, *b*) **and** *spp-of ?q* = *spp-of a*
proof (*rule sum-prodE*)
fix *a b*
assume *y* = *Inl* (*a*, *b*)

moreover from *eq0* **have** *spp-of ?q = spp-of a* **by** (*simp add*: ‹*y = Inl* (*a*, *b*)›)
ultimately show *?thesis* **..**
next
fix *j*
assume *y = Inr j*
with *eq0* **show** *?thesis* **by** *simp*
qed
from *this*(*2*) **have** *lt ?q = lt a* **by** (*simp add*: *spp-of-def*)
from ‹*y* ∈ *set ps*› **have** *y* ∈ *set* (*p* # *ps*) **by** *simp*
with ∗ **have** *a* ∈ *set bs* **unfolding** *y* **by** (*rule rb-aux-inv-D3*(*1*))
hence *lt ?q* ∈ *lt* ‘ *set bs* **unfolding** ‹*lt ?q = lt a*› **by** (*rule imageI*)
moreover from ∗ *a b* **have** *lt ?q* ∉ *lt* ‘ *set bs* **by** (*rule rb-aux-inv-preserved-3*)
ultimately show *False* **by** *simp*
qed
hence *eq3*: *add-spairs-spp* (*map* (*app-pair spp-of*) *ps*) (*map spp-of bs*) (*spp-of ?q*) =
map (*app-pair spp-of*) (*add-spairs ps bs ?q*) **by** (*simp add*: *add-spairs-alt-spp*)

from ∗ *a b* **have** *rb-aux-inv* (*fst* ((*?q* # *bs*, *new-syz-sigs ss bs p*, *add-spairs ps bs ?q*), *z*))
unfolding *fst-conv* **by** (*rule rb-aux-inv-preserved-3*)
with *refl a refl b*
have *app-args spp-of* (*rb-aux* ((*?q* # *bs*, *new-syz-sigs ss bs p*, *add-spairs ps bs ?q*), *z*)) =
rb-spp-aux (*app-args spp-of* ((*?q* # *bs*, *new-syz-sigs ss bs p*, *add-spairs ps bs ?q*), *z*))
by (*rule 2*(*4*))
also have ... = *rb-spp-aux* ((*map spp-of bs*, *ss*, *app-pair spp-of p* # *map* (*app-pair spp-of*) *ps*), *z*)
by (*simp add*: *rb-spp-aux-Cons eq1 a Let-def eq2 fst-spp-of snd-spp-of b eq3 new-syz-sigs-alt-spp*[*symmetric*])
finally show *app-args spp-of* (*rb-aux* ((*?q* # *bs*, *new-syz-sigs ss bs p*, *add-spairs ps bs ?q*), *z*)) =
rb-spp-aux ((*map spp-of bs*, *ss*, *app-pair spp-of p* # *map* (*app-pair spp-of*) *ps*), *z*) **.**
qed
qed
qed

corollary *rb-spp-aux-alt*:
rb-aux-inv (*fst* (*app-args vec-of args*)) ⟹
rb-spp-aux args = *app-args spp-of* (*rb-aux* (*app-args vec-of args*))
by (*simp only*: *rb-aux-alt-spp app-args-spp-of-vec-of*)

corollary *rb-spp-aux*:
hom-grading dgrad ⟹
punit.is-Groebner-basis (*set* (*map snd* (*fst* (*fst* (*rb-spp-aux* (([], *Koszul-syz-sigs*

fs, *map Inr* [*0*..<*length fs*]), *z*))))))
(**is** - $\Longrightarrow$ *?thesis1*)
ideal (*set* (*map snd* (*fst* (*fst* (*rb-spp-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)))))) = *ideal* (*set fs*)
(**is** *?thesis2*)
set (*map snd* (*fst* (*fst* (*rb-spp-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*))))) $\subseteq$ *punit-dgrad-max-set dgrad*
(**is** *?thesis3*)
0 $\notin$ *set* (*map snd* (*fst* (*fst* (*rb-spp-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)))))
(**is** *?thesis4*)
hom-grading dgrad $\Longrightarrow$ *is-pot-ord* $\Longrightarrow$ *is-regular-sequence fs* $\Longrightarrow$
snd (*rb-spp-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)) = *z*
(**is** - $\Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ *?thesis5*)
rword-strict = *rw-rat-strict* $\Longrightarrow$ *p* $\in$ *set* (*fst* (*fst* (*rb-spp-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)))) $\Longrightarrow$
q $\in$ *set* (*fst* (*fst* (*rb-spp-aux* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)))) $\Longrightarrow$ *p* $\neq$ *q* $\Longrightarrow$
punit.lt (*snd p*) *adds punit.lt* (*snd q*) $\Longrightarrow$ *punit.lt* (*snd p*) $\oplus$ *fst q* $\prec_t$ *punit.lt* (*snd q*) $\oplus$ *fst p*
proof −
let *?args* = (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *z*)
have *eq0*: *app-pair vec-of* ∘ *Inr* = *Inr* **by** (*intro ext*, *simp*)
have *eq1*: *fst* (*fst* (*app-args spp-of a*)) = *map spp-of* (*fst* (*fst a*)) **for** *a*::(- × ('*t list*) × -) × -
proof −
obtain *bs ss ps z* **where** *a* = ((*bs*, *ss*, *ps*), *z*) **using** *prod.exhaust* **by** *metis*
thus *?thesis* **by** *simp*
qed
have *eq2*: *snd* ∘ *spp-of* = *rep-list* **by** (*intro ext*, *simp add*: *snd-spp-of*)
have *rb-aux-inv* (*fst* (*app-args vec-of ?args*)) **by** (*simp add*: *eq0 rb-aux-inv-init*)
hence *eq3*: *rb-spp-aux ?args* = *app-args spp-of* (*rb-aux* (*app-args vec-of ?args*))
by (*rule rb-spp-aux-alt*)

{
assume *hom-grading dgrad*
with *rb-aux-is-Groebner-basis* **show** *?thesis1* **by** (*simp add*: *eq0 eq1 eq2 eq3 del*: *set-map*)
}

from *ideal-rb-aux* **show** *?thesis2* **by** (*simp add*: *eq0 eq1 eq2 eq3 del*: *set-map*)

from *dgrad-max-set-closed-rb-aux* **show** *?thesis3* **by** (*simp add*: *eq0 eq1 eq2 eq3 del*: *set-map*)

from *rb-aux-nonzero* **show** *?thesis4* **by** (*simp add*: *eq0 eq1 eq2 eq3 del*: *set-map*)

{
assume *is-pot-ord* **and** *hom-grading dgrad* **and** *is-regular-sequence fs*

hence *snd* (*rb-aux ?args*) = *z* **by** (*rule rb-aux-no-zero-red*)
thus *?thesis5* **by** (*simp add*: *snd-app-args eq0 eq3*)
}

{
from *rb-aux-nonzero* **have** $0 \notin$ *rep-list* ' *set* (*fst* (*fst* (*rb-aux ?args*)))
(**is** $0 \notin$ *rep-list* ' *?G*) **by** *simp*
assume *rword-strict* = *rw-rat-strict*
hence *is-min-sig-GB dgrad ?G* **by** (*rule rb-aux-is-min-sig-GB*)
hence *rl*: $\bigwedge g.\ g \in ?G \Longrightarrow \neg$ *is-sig-red* ($\preceq_t$) (=) (*?G* − {*g*}) *g* **by** (*simp add*: *is-min-sig-GB-def*)
assume *p* ∈ *set* (*fst* (*fst* (*rb-spp-aux ?args*)))
also have ... = *spp-of* ' *?G* **by** (*simp add*: *eq0 eq1 eq3*)
finally obtain *p'* **where** *p'* ∈ *?G* **and** *p*: *p* = *spp-of p'* **..**
assume *q* ∈ *set* (*fst* (*fst* (*rb-spp-aux ?args*)))
also have ... = *spp-of* ' *?G* **by** (*simp add*: *eq0 eq1 eq3*)
finally obtain *q'* **where** *q'* ∈ *?G* **and** *q*: *q* = *spp-of q'* **..**
from *this*(*1*) **have** *1*: ¬ *is-sig-red* ($\preceq_t$) (=) (*?G* − {*q'*}) *q'* **by** (*rule rl*)
assume *p* ≠ *q* **and** *punit.lt* (*snd p*) *adds punit.lt* (*snd q*)
hence *p'* ≠ *q'* **and** *adds*: *punit.lt* (*rep-list p'*) *adds punit.lt* (*rep-list q'*)
by (*auto simp*: *p q snd-spp-of*)
show *punit.lt* (*snd p*) ⊕ *fst q* $\prec_t$ *punit.lt* (*snd q*) ⊕ *fst p*
proof (*rule ccontr*)
assume ¬ *punit.lt* (*snd p*) ⊕ *fst q* $\prec_t$ *punit.lt* (*snd q*) ⊕ *fst p*
hence *le*: *punit.lt* (*rep-list q'*) ⊕ *lt p'* $\preceq_t$ *punit.lt* (*rep-list p'*) ⊕ *lt q'*
by (*simp add*: *p q spp-of-def*)
from ‹*p'* ≠ *q'*› ‹*p'* ∈ *?G*› **have** *p'* ∈ *?G* − {*q'*} **by** *simp*
moreover from ‹*p'* ∈ *?G*› ‹*0* ∉ *rep-list* ' *?G*› **have** *rep-list p'* ≠ *0* **by** *fastforce*
moreover from ‹*q'* ∈ *?G*› ‹*0* ∉ *rep-list* ' *?G*› **have** *rep-list q'* ≠ *0* **by** *fastforce*
moreover note *adds*
moreover have *ord-term-lin.is-le-rel* ($\preceq_t$) **by** *simp*
ultimately have *is-sig-red* ($\preceq_t$) (=) (*?G* − {*q'*}) *q'* **using** *le* **by** (*rule is-sig-red-top-addsI*)
with *1* **show** *False* **..**
qed
}
qed

end

end

end

end

end

definition *gb-sig-z* ::

$(('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b)) \Rightarrow bool) \Rightarrow ('a \Rightarrow_0 'b)\ list \Rightarrow (('t \times ('a \Rightarrow_0 'b{::}field))\ list \times nat)$
where *gb-sig-z rword-strict fs0* =
(*let fs* = *rev* (*remdups* (*rev* (*removeAll 0 fs0*)));
res = *rb-spp-aux fs rword-strict* (([], *Koszul-syz-sigs fs*, *map Inr* [*0*..<*length fs*]), *0*) *in*
(*fst* (*fst res*), *snd res*))

The second return value of *gb-sig-z* is the total number of zero-reductions.

definition *gb-sig* :: $(('t \times ('a \Rightarrow_0 'b)) \Rightarrow ('t \times ('a \Rightarrow_0 'b)) \Rightarrow bool) \Rightarrow ('a \Rightarrow_0 'b)\ list \Rightarrow ('a \Rightarrow_0 'b{::}field)\ list$
where *gb-sig rword-strict fs0* = *map snd* (*fst* (*gb-sig-z rword-strict fs0*))

theorem
assumes $\bigwedge$*fs. is-strict-rewrite-ord fs rword-strict*
shows *gb-sig-isGB*: *punit.is-Groebner-basis* (*set* (*gb-sig rword-strict fs*)) (**is** *?thesis1*)
and *gb-sig-ideal*: *ideal* (*set* (*gb-sig rword-strict fs*)) = *ideal* (*set fs*) (**is** *?thesis2*)
and *dgrad-p-set-closed-gb-sig*:
dickson-grading d $\Longrightarrow$ *set fs* $\subseteq$ *punit.dgrad-p-set d m* $\Longrightarrow$ *set* (*gb-sig rword-strict fs*) $\subseteq$ *punit.dgrad-p-set d m*
(**is** - $\Longrightarrow$ - $\Longrightarrow$ *?thesis3*)
and *gb-sig-nonzero*: *0* $\notin$ *set* (*gb-sig rword-strict fs*) (**is** *?thesis4*)
and *gb-sig-no-zero-red*: *is-pot-ord* $\Longrightarrow$ *is-regular-sequence fs* $\Longrightarrow$ *snd* (*gb-sig-z rword-strict fs*) = *0*
proof −
from *ex-hgrad* **obtain** *d0*::*'a* ⇒ *nat* **where** *dickson-grading d0* ∧ *hom-grading d0* **..**
hence *dg*: *dickson-grading d0* **and** *hg*: *hom-grading d0* **by** *simp-all*
define *fs1* **where** *fs1* = *rev* (*remdups* (*rev* (*removeAll 0 fs*)))
note *assms dg*
moreover have *distinct fs1* **and** *0* $\notin$ *set fs1* **by** (*simp-all add*: *fs1-def*)
ultimately have *ideal* (*set* (*gb-sig rword-strict fs*)) = *ideal* (*set fs1*) **and** *?thesis4*
unfolding *gb-sig-def gb-sig-z-def fst-conv fs1-def Let-def* **by** (*rule rb-spp-aux*)+
thus *?thesis2* **and** *?thesis4* **by** (*simp-all add*: *fs1-def ideal.span-Diff-zero*)

from *assms dg* ‹*distinct fs1*› ‹*0* $\notin$ *set fs1*› *hg* **show** *?thesis1*
unfolding *gb-sig-def gb-sig-z-def fst-conv fs1-def Let-def* **by** (*rule rb-spp-aux*)

{
assume *dg*: *dickson-grading d* **and** $*$: *set fs* $\subseteq$ *punit.dgrad-p-set d m*
show *?thesis3*
proof (*cases set fs* $\subseteq$ {*0*})
case *True*
hence *removeAll 0 fs* = []
by (*metis* (*no-types*, *lifting*) *Diff-iff ex-in-conv set-empty2 set-removeAll subset-singleton-iff*)
thus *?thesis* **by** (*simp add*: *gb-sig-def gb-sig-z-def Let-def rb-spp-aux-Nil*)
next
case *False*

have *set fs1* ⊆ *set fs* **by** (*fastforce simp*: *fs1-def*)
hence *Keys* (*set fs1*) ⊆ *Keys* (*set fs*) **by** (*rule Keys-mono*)
hence *d* ' *Keys* (*set fs1*) ⊆ *d* ' *Keys* (*set fs*) **by** (*rule image-mono*)
hence *insert* (*d 0*) (*d* ' *Keys* (*set fs1*)) ⊆ *insert* (*d 0*) (*d* ' *Keys* (*set fs*)) **by** (*rule insert-mono*)
moreover have *insert* (*d 0*) (*d* ' *Keys* (*set fs1*)) ≠ {} **by** *simp*
moreover have *finite* (*insert* (*d 0*) (*d* ' *Keys* (*set fs*)))
by (*simp add*: *finite-Keys*)
ultimately have *le*: *Max* (*insert* (*d 0*) (*d* ' *Keys* (*set fs1*))) ≤
Max (*insert* (*d 0*) (*d* ' *Keys* (*set fs*))) **by** (*rule Max-mono*)
from *assms dg* **have** *set* (*gb-sig rword-strict fs*) ⊆ *punit-dgrad-max-set* (*TYPE*(*'b*)) *fs1 d*
using ‹*distinct fs1*› ‹*0* ∉ *set fs1*›
unfolding *gb-sig-def gb-sig-z-def fst-conv fs1-def Let-def* **by** (*rule rb-spp-aux*)
also have *punit-dgrad-max-set* (*TYPE*(*'b*)) *fs1 d* ⊆ *punit-dgrad-max-set* (*TYPE*(*'b*)) *fs d*
by (*rule punit.dgrad-p-set-subset*, *simp add*: *dgrad-max-def le*)
also from *dg* ∗ *False* **have** ... ⊆ *punit.dgrad-p-set d m*
by (*rule punit-dgrad-max-set-subset-dgrad-p-set*)
finally show *?thesis* **.**
qed
}

{
assume *is-regular-sequence fs*
note *assms dg* ‹*distinct fs1*› ‹*0* ∉ *set fs1*› *hg*
moreover assume *is-pot-ord*
moreover from ‹*is-regular-sequence fs*› **have** *is-regular-sequence fs1* **unfolding** *fs1-def*
by (*intro is-regular-sequence-remdups is-regular-sequence-removeAll-zero*)
ultimately show *snd* (*gb-sig-z rword-strict fs*) = *0*
unfolding *gb-sig-def gb-sig-z-def snd-conv fs1-def Let-def* **by** (*rule rb-spp-aux*)
}
qed

theorem *gb-sig-z-is-min-sig-GB*:
assumes *p* ∈ *set* (*fst* (*gb-sig-z rw-rat-strict fs*)) **and** *q* ∈ *set* (*fst* (*gb-sig-z rw-rat-strict fs*))
and *p* ≠ *q* **and** *punit.lt* (*snd p*) *adds punit.lt* (*snd q*)
shows *punit.lt* (*snd p*) ⊕ *fst q* $\prec_t$ *punit.lt* (*snd q*) ⊕ *fst p*
proof −
define *fs1* **where** *fs1* = *rev* (*remdups* (*rev* (*removeAll 0 fs*)))
from *ex-hgrad* **obtain** *d0*::*'a* ⇒ *nat* **where** *dickson-grading d0* ∧ *hom-grading d0* **..**
hence *dickson-grading d0* **..**
note *rw-rat-strict-is-strict-rewrite-ord this*
moreover have *distinct fs1* **and** *0* ∉ *set fs1* **by** (*simp-all add*: *fs1-def*)
moreover note *refl assms*
ultimately show *?thesis* **unfolding** *gb-sig-z-def fst-conv fs1-def Let-def* **by** (*rule*

rb-spp-aux)
qed

Summarizing, these are the four main results proved in this theory:

- ($\bigwedge$*fs. is-strict-rewrite-ord fs ?rword-strict*) $\Longrightarrow$ *punit.is-Groebner-basis* (*set* (*gb-sig ?rword-strict ?fs*)),
- ($\bigwedge$*fs. is-strict-rewrite-ord fs ?rword-strict*) $\Longrightarrow$ *ideal* (*set* (*gb-sig ?rword-strict ?fs*)) = *ideal* (*set ?fs*),
- $[\![\bigwedge$*fs. is-strict-rewrite-ord fs ?rword-strict*; *is-pot-ord*; *is-regular-sequence ?fs*$]\!]$ $\Longrightarrow$ *snd* (*gb-sig-z ?rword-strict ?fs*) = *0*, and
- $[\![$*?p* $\in$ *set* (*fst* (*gb-sig-z rw-rat-strict ?fs*)); *?q* $\in$ *set* (*fst* (*gb-sig-z rw-rat-strict ?fs*)); *?p* $\neq$ *?q*; *punit.lt* (*snd ?p*) *adds punit.lt* (*snd ?q*)$]\!]$ $\Longrightarrow$ *punit.lt* (*snd ?p*) $\oplus$ *fst ?q* $\prec_t$ *punit.lt* (*snd ?q*) $\oplus$ *fst ?p*.

end

end

5 Sample Computations with Signature-Based Algorithms

theory *Signature-Examples*
 imports *Signature-Groebner Groebner-Bases.Benchmarks Groebner-Bases.Code-Target-Rat*
begin

5.1 Setup

lift-definition *except-pp* :: ('*a*, '*b*) *pp* ⇒ '*a set* ⇒ ('*a*, '*b*::*zero*) *pp* **is** *except* **.**

lemma *hom-grading-varnum-pp*: *hom-grading* (*varnum-pp*::('*a*::*countable*, '*b*::*add-wellorder*) *pp* ⇒ *nat*)
proof −
 define *f* **where** *f* = (λ*n t*. (*except-pp t* (− {*x*. *elem-index x* < *n*}))::('*a*, '*b*) *pp*)
 show *?thesis* **unfolding** *hom-grading-def hom-grading-fun-def*
 proof (*intro exI allI conjI impI*)
  fix *n s t*
  show *f n* (*s* + *t*) = *f n s* + *f n t* **unfolding** *f-def* **by** *transfer* (*rule except-plus*)
 next
  fix *n t*
  show *varnum-pp* (*f n t*) ≤ *n* **unfolding** *f-def*
   by *transfer* (*simp add*: *varnum-le-iff keys-except*)
 next
  fix *n t*
  show *varnum-pp t* ≤ *n* ⟹ *f n t* = *t* **unfolding** *f-def*

by *transfer* (*auto simp*: *except-id-iff varnum-le-iff*)
qed
qed

instance *pp* :: (*countable*, *add-wellorder*) *quasi-pm-powerprod*
by (*standard*, *intro exI conjI*, *fact dickson-grading-varnum-pp*, *fact hom-grading-varnum-pp*)

5.1.1 Projections of Term Orders to Orders on Power-Products

definition *proj-comp* :: (('*a*::*nat*, '*b*::*nat*) *pp* × *nat*) *nat-term-order* ⇒ ('*a*, '*b*) *pp* ⇒ ('*a*, '*b*) *pp* ⇒ *order*
where *proj-comp cmp* = (λ*x y*. *nat-term-compare cmp* (*x*, *0*) (*y*, *0*))

definition *proj-ord* :: (('*a*::*nat*, '*b*::*nat*) *pp* × *nat*) *nat-term-order* ⇒ ('*a*, '*b*) *pp* *nat-term-order*
where *proj-ord cmp* = *Abs-nat-term-order* (*proj-comp cmp*)

In principle, *proj-comp* and *proj-ord* could be defined more generally on type '*a* × *nat*, but then '*a* would have to belong to some new type-class which is the intersection of *nat-pp-term* and *nat-pp-compare* and additionally requires *rep-nat-term x* = (*rep-nat-pp x*, *0*).

lemma *comparator-proj-comp*: *comparator* (*proj-comp cmp*)
proof −
interpret *cmp*: *comparator nat-term-compare cmp* **by** (*rule comparator-nat-term-compare*)
show *?thesis* **unfolding** *proj-comp-def*
proof
fix *x y* :: ('*a*, '*b*) *pp*
show *invert-order* (*nat-term-compare cmp* (*x*, *0*) (*y*, *0*)) = *nat-term-compare cmp* (*y*, *0*) (*x*, *0*)
by (*simp only*: *cmp.sym*)
next
fix *x y* :: ('*a*, '*b*) *pp*
assume *nat-term-compare cmp* (*x*, *0*) (*y*, *0*) = *Eq*
hence (*x*, *0*) = (*y*, *0*::*nat*) **by** (*rule cmp.weak-eq*)
thus *x* = *y* **by** *simp*
next
fix *x y z* :: ('*a*, '*b*) *pp*
assume *nat-term-compare cmp* (*x*, *0*) (*y*, *0*) = *Lt* **and** *nat-term-compare cmp* (*y*, *0*) (*z*, *0*) = *Lt*
thus *nat-term-compare cmp* (*x*, *0*) (*z*, *0*) = *Lt* **by** (*rule cmp.comp-trans*)
qed
qed

lemma *nat-term-comp-proj-comp*: *nat-term-comp* (*proj-comp cmp*)
proof −
have *1*: *fst* (*rep-nat-term* (*u*, *i*)) = *rep-nat-pp u* **for** *u*::('*a*, '*b*) *pp* **and** *i*::*nat*
by (*simp add*: *rep-nat-term-prod-def*)
have *2*: *snd* (*rep-nat-term* (*u*, *i*)) = *i* **for** *u*::('*a*, '*b*) *pp* **and** *i*::*nat*
by (*simp add*: *rep-nat-term-prod-def rep-nat-nat-def*)

show *?thesis*
proof (*rule nat-term-compI*)
fix *u v* :: (*'a*, *'b*) *pp*
assume *a*: *fst* (*rep-nat-term u*) = *0*
note *nat-term-comp-nat-term-compare*
moreover have *snd* (*rep-nat-term* (*u*, *0*::*nat*)) = *snd* (*rep-nat-term* (*v*, *0*::*nat*)) **by** (*simp only*: *2*)
moreover from *a* **have** *fst* (*rep-nat-term* (*u*, *0*::*nat*)) = *0* **by** (*simp add*: *1 rep-nat-term-pp-def*)
ultimately have *nat-term-compare cmp* (*u*, *0*) (*v*, *0*) ≠ *Gt* **by** (*rule nat-term-compD1*)
thus *proj-comp cmp u v* ≠ *Gt* **by** (*simp add*: *proj-comp-def*)
next
fix *u v* :: (*'a*, *'b*) *pp*
assume *snd* (*rep-nat-term u*) < *snd* (*rep-nat-term v*)
thus *proj-comp cmp u v* = *Lt* **by** (*simp add*: *rep-nat-term-pp-def*)
next
fix *t u v* :: (*'a*, *'b*) *pp*
assume *proj-comp cmp u v* = *Lt*
hence *nat-term-compare cmp* (*u*, *0*) (*v*, *0*) = *Lt* **by** (*simp add*: *proj-comp-def*)
with *nat-term-comp-nat-term-compare* **have** *nat-term-compare cmp* (*splus* (*t*, *0*) (*u*, *0*)) (*splus* (*t*, *0*) (*v*, *0*)) = *Lt*
by (*rule nat-term-compD3*)
thus *proj-comp cmp* (*splus t u*) (*splus t v*) = *Lt*
by (*simp add*: *proj-comp-def splus-prod-def pprod.splus-def splus-pp-term*)
next
fix *u v a b* :: (*'a*, *'b*) *pp*
assume *u*: *fst* (*rep-nat-term u*) = *fst* (*rep-nat-term a*) **and** *v*: *fst* (*rep-nat-term v*) = *fst* (*rep-nat-term b*)
and *a*: *proj-comp cmp a b* = *Lt*
note *nat-term-comp-nat-term-compare*
moreover from *u* **have** *fst* (*rep-nat-term* (*u*, *0*::*nat*)) = *fst* (*rep-nat-term* (*a*, *0*::*nat*))
by (*simp add*: *1 rep-nat-term-pp-def*)
moreover from *v* **have** *fst* (*rep-nat-term* (*v*, *0*::*nat*)) = *fst* (*rep-nat-term* (*b*, *0*::*nat*))
by (*simp add*: *1 rep-nat-term-pp-def*)
moreover have *snd* (*rep-nat-term* (*u*, *0*::*nat*)) = *snd* (*rep-nat-term* (*v*, *0*::*nat*))
and *snd* (*rep-nat-term* (*a*, *0*::*nat*)) = *snd* (*rep-nat-term* (*b*, *0*::*nat*)) **by** (*simp-all only*: *2*)
moreover from *a* **have** *nat-term-compare cmp* (*a*, *0*) (*b*, *0*) = *Lt* **by** (*simp add*: *proj-comp-def*)
ultimately have *nat-term-compare cmp* (*u*, *0*) (*v*, *0*) = *Lt* **by** (*rule nat-term-compD4*)
thus *proj-comp cmp u v* = *Lt* **by** (*simp add*: *proj-comp-def*)
qed
qed

corollary *nat-term-compare-proj-ord*: *nat-term-compare* (*proj-ord cmp*) = *proj-comp cmp*
unfolding *proj-ord-def* **using** *comparator-proj-comp nat-term-comp-proj-comp*

by (*rule nat-term-compare-Abs-nat-term-order-id*)

lemma *proj-ord-LEX* [*code*]: *proj-ord LEX* = *LEX*
proof −
have *nat-term-compare* (*proj-ord LEX*) = *nat-term-compare LEX*
by (*auto simp*: *nat-term-compare-proj-ord nat-term-compare-LEX proj-comp-def lex-comp*
lex-comp-aux-def rep-nat-term-prod-def rep-nat-term-pp-def intro!: *ext split*: *order.split*)
thus *?thesis* **by** (*simp only*: *nat-term-compare-inject*)
qed

lemma *proj-ord-DRLEX* [*code*]: *proj-ord DRLEX* = *DRLEX*
proof −
have *nat-term-compare* (*proj-ord DRLEX*) = *nat-term-compare DRLEX*
by (*auto simp*: *nat-term-compare-proj-ord nat-term-compare-DRLEX proj-comp-def deg-comp pot-comp*
lex-comp lex-comp-aux-def rep-nat-term-prod-def rep-nat-term-pp-def intro!: *ext split*: *order.split*)
thus *?thesis* **by** (*simp only*: *nat-term-compare-inject*)
qed

lemma *proj-ord-DEG* [*code*]: *proj-ord* (*DEG to*) = *DEG* (*proj-ord to*)
proof −
have *nat-term-compare* (*proj-ord* (*DEG to*)) = *nat-term-compare* (*DEG* (*proj-ord to*))
by (*simp add*: *nat-term-compare-proj-ord nat-term-compare-DEG proj-comp-def deg-comp*
rep-nat-term-prod-def rep-nat-term-pp-def)
thus *?thesis* **by** (*simp only*: *nat-term-compare-inject*)
qed

lemma *proj-ord-POT* [*code*]: *proj-ord* (*POT to*) = *proj-ord to*
proof −
have *nat-term-compare* (*proj-ord* (*POT to*)) = *nat-term-compare* (*proj-ord to*)
by (*simp add*: *nat-term-compare-proj-ord nat-term-compare-POT proj-comp-def pot-comp*
rep-nat-term-prod-def rep-nat-term-pp-def)
thus *?thesis* **by** (*simp only*: *nat-term-compare-inject*)
qed

5.1.2 Locale Interpretation

locale *qpm-nat-inf-term* = *gd-nat-term* $\lambda x.\ x$ $\lambda x.\ x$ *to*
for *to*::(('*a*::*nat*, '*b*::*nat*) *pp* × *nat*) *nat-term-order*
begin

sublocale *aux*: *qpm-inf-term* $\lambda x.\ x$ $\lambda x.\ x$
le-of-nat-term-order (*proj-ord to*)

lt-of-nat-term-order (*proj-ord to*)
le-of-nat-term-order to
lt-of-nat-term-order to
proof *intro-locales*

show *ordered-powerprod-axioms* (*le-of-nat-term-order* (*proj-ord to*))
by (*unfold-locales*, *fact le-of-nat-term-order-zero-min*, *auto dest*: *le-of-nat-term-order-plus-monotone simp*: *ac-simps*)
next
show *ordered-term-axioms* ($\lambda x.$ *x*) ($\lambda x.$ *x*) (*le-of-nat-term-order* (*proj-ord to*)) (*le-of-nat-term-order to*)
proof
fix *v w t*
assume *le-of-nat-term-order to v w*
thus *le-of-nat-term-order to* (*local.splus t v*) (*local.splus t w*)
by (*simp add*: *le-of-nat-term-order nat-term-compare-splus splus-eq-splus*)
next
fix *v w*
assume *le-of-nat-term-order* (*proj-ord to*) (*pp-of-term v*) (*pp-of-term w*)
and *component-of-term v* $\leq$ *component-of-term w*
hence *nat-term-compare to* (*fst v*, *0*) (*fst w*, *0*) $\neq$ *Gt* **and** *snd v* $\leq$ *snd w*
by (*simp-all add*: *le-of-nat-term-order nat-term-compare-proj-ord proj-comp-def*)
from *comparator-nat-term-compare nat-term-comp-nat-term-compare - - - - this*(*1*)
have *nat-term-compare to v w* $\neq$ *Gt*
by (*rule nat-term-compD4''*) (*simp-all add*: *rep-nat-term-prod-def ord-iff*[*symmetric*] ‹*snd v* $\leq$ *snd w*›)
thus *le-of-nat-term-order to v w* **by** (*simp add*: *le-of-nat-term-order*)
qed
qed

end

We must define the following two constants outside the global interpretation, since otherwise their types are too general.

definition *splus-pprod* :: (*'a*::*nat*, *'b*::*nat*) *pp* $\Rightarrow$ -
where *splus-pprod* = *pprod.splus*

definition *adds-term-pprod* :: ((*'a*::*nat*, *'b*::*nat*) *pp* $\times$ -) $\Rightarrow$ -
where *adds-term-pprod* = *pprod.adds-term*

global-interpretation *pprod'*: *qpm-nat-inf-term to*
rewrites *pprod.pp-of-term* = *fst*
and *pprod.component-of-term* = *snd*
and *pprod.splus* = *splus-pprod*
and *pprod.adds-term* = *adds-term-pprod*
and *punit.monom-mult* = *monom-mult-punit*
and *pprod'.aux.punit.lt* = *lt-punit* (*proj-ord to*)
and *pprod'.aux.punit.lc* = *lc-punit* (*proj-ord to*)

and *pprod′.aux.punit.tail* = *tail-punit* (*proj-ord to*)
for *to* :: ((′*a*::*nat*, ′*b*::*nat*) *pp* × *nat*) *nat-term-order*
defines *max-pprod* = *pprod′.ord-term-lin.max*
and *Koszul-syz-sigs-aux-pprod* = *pprod′.aux.Koszul-syz-sigs-aux*
and *Koszul-syz-sigs-pprod* = *pprod′.aux.Koszul-syz-sigs*
and *find-sig-reducer-pprod* = *pprod′.aux.find-sig-reducer*
and *sig-trd-spp-body-pprod* = *pprod′.aux.sig-trd-spp-body*
and *sig-trd-spp-aux-pprod* = *pprod′.aux.sig-trd-spp-aux*
and *sig-trd-spp-pprod* = *pprod′.aux.sig-trd-spp*
and *spair-sigs-spp-pprod* = *pprod′.aux.spair-sigs-spp*
and *is-pred-syz-pprod* = *pprod′.aux.is-pred-syz*
and *is-rewritable-spp-pprod* = *pprod′.aux.is-rewritable-spp*
and *sig-crit-spp-pprod* = *pprod′.aux.sig-crit-spp*
and *spair-spp-pprod* = *pprod′.aux.spair-spp*
and *spp-of-pair-pprod* = *pprod′.aux.spp-of-pair*
and *pair-ord-spp-pprod* = *pprod′.aux.pair-ord-spp*
and *sig-of-pair-spp-pprod* = *pprod′.aux.sig-of-pair-spp*
and *new-spairs-spp-pprod* = *pprod′.aux.new-spairs-spp*
and *is-regular-spair-spp-pprod* = *pprod′.aux.is-regular-spair-spp*
and *add-spairs-spp-pprod* = *pprod′.aux.add-spairs-spp*
and *is-pot-ord-pprod* = *pprod′.is-pot-ord*
and *new-syz-sigs-spp-pprod* = *pprod′.aux.new-syz-sigs-spp*
and *rb-spp-body-pprod* = *pprod′.aux.rb-spp-body*
and *rb-spp-aux-pprod* = *pprod′.aux.rb-spp-aux*
and *gb-sig-z-pprod′* = *pprod′.aux.gb-sig-z*
and *gb-sig-pprod′* = *pprod′.aux.gb-sig*
and *rw-rat-strict-pprod* = *pprod′.aux.rw-rat-strict*
and *rw-add-strict-pprod* = *pprod′.aux.rw-add-strict*
subgoal by (*rule qpm-nat-inf-term.intro*, *fact gd-nat-term-id*)
subgoal by (*fact pprod-pp-of-term*)
subgoal by (*fact pprod-component-of-term*)
subgoal by (*simp only*: *splus-pprod-def*)
subgoal by (*simp only*: *adds-term-pprod-def*)
subgoal by (*simp only*: *monom-mult-punit-def*)
subgoal by (*simp only*: *lt-punit-def*)
subgoal by (*simp only*: *lc-punit-def*)
subgoal by (*simp only*: *tail-punit-def*)
done

5.1.3 More Lemmas and Definitions

lemma *compute-adds-term-pprod* [*code*]:
adds-term-pprod u v = (*snd u* = *snd v* ∧ *adds-pp-add-linorder* (*fst u*) (*fst v*))
by (*simp add*: *adds-term-pprod-def pprod.adds-term-def adds-pp-add-linorder-def*)

lemma *compute-splus-pprod* [*code*]: *splus-pprod t* (*s*, *i*) = (*t* + *s*, *i*)
by (*simp add*: *splus-pprod-def pprod.splus-def*)

lemma *compute-sig-trd-spp-body-pprod* [*code*]:

sig-trd-spp-body-pprod to bs v (*p*, *r*) =
(*case find-sig-reducer-pprod to bs v* (*lt-punit* (*proj-ord to*) *p*) *0 of*
None ⇒ (*tail-punit* (*proj-ord to*) *p*, *plus-monomial-less r* (*lc-punit* (*proj-ord to*) *p*) (*lt-punit* (*proj-ord to*) *p*))
| *Some i* ⇒ *let b* = *snd* (*bs* ! *i*) *in*
(*tail-punit* (*proj-ord to*) *p* − *monom-mult-punit* (*lc-punit* (*proj-ord to*) *p* / *lc-punit* (*proj-ord to*) *b*)
(*lt-punit* (*proj-ord to*) *p* − *lt-punit* (*proj-ord to*) *b*) (*tail-punit* (*proj-ord to*) *b*), *r*))
by (*simp add*: *plus-monomial-less-def split*: *option.split*)

lemma *compute-sig-trd-spp-pprod* [*code*]:
sig-trd-spp-pprod to bs (*v*, *p*) ≡ (*v*, *sig-trd-spp-aux-pprod to bs v* (*p*, *change-ord* (*proj-ord to*) *0*))
by (*simp add*: *change-ord-def*)

lemmas [*code*] = *conversep-iff*

lemma *compute-is-pot-ord* [*code*]:
is-pot-ord-pprod (*LEX*::((*'a*::*nat*, *'b*::*nat*) *pp* × *nat*) *nat-term-order*) = *False*
(**is** *is-pot-ord-pprod ?lex* = -)
is-pot-ord-pprod (*DRLEX*::((*'a*::*nat*, *'b*::*nat*) *pp* × *nat*) *nat-term-order*) = *False*
(**is** *is-pot-ord-pprod ?drlex* = -)
is-pot-ord-pprod (*DEG* (*to*::((*'a*::*nat*, *'b*::*nat*) *pp* × *nat*) *nat-term-order*)) = *False*
is-pot-ord-pprod (*POT* (*to*::((*'a*::*nat*, *'b*::*nat*) *pp* × *nat*) *nat-term-order*)) = *True*
proof −
have *eq1*: *snd* ((*Term-Order.of-exps a b i*)::(*'a*, *'b*) *pp* × *nat*) = *i* **for** *a b* **and** *i*::*nat*
proof −
have *snd* ((*Term-Order.of-exps a b i*)::(*'a*, *'b*) *pp* × *nat*) =
snd (*rep-nat-term* ((*Term-Order.of-exps a b i*)::(*'a*, *'b*) *pp* × *nat*))
by (*simp add*: *rep-nat-term-prod-def rep-nat-nat-def*)
also have ... = *i*
proof (*rule snd-of-exps*)
show *snd* (*rep-nat-term* (*undefined*, *i*)) = *i* **by** (*simp add*: *rep-nat-term-prod-def rep-nat-nat-def*)
qed
finally show *?thesis* **.**
qed

let *?u* = (*Term-Order.of-exps 1 0 0*)::(*'a*, *'b*) *pp* × *nat*
let *?v* = (*Term-Order.of-exps 0 0 1*)::(*'a*, *'b*) *pp* × *nat*
have ¬ *is-pot-ord-pprod ?lex*
proof
assume *is-pot-ord-pprod ?lex*
moreover have *le-of-nat-term-order ?lex ?v ?u*
by (*simp add*: *le-of-nat-term-order nat-term-compare-LEX lex-comp lex-comp-aux-def comp-of-ord-def lex-pp-of-exps eq-of-exps*)
ultimately have *snd ?v* ≤ *snd ?u* **by** (*rule pprod'.is-pot-ordD2*)

thus *False* **by** (*simp add*: *eq1*)
qed
thus *is-pot-ord-pprod ?lex* = *False* **by** *simp*

have ¬ *is-pot-ord-pprod ?drlex*
proof
assume *is-pot-ord-pprod ?drlex*
moreover have *le-of-nat-term-order ?drlex ?v ?u*
by (*simp add*: *le-of-nat-term-order nat-term-compare-DRLEX deg-comp comparator-of-def*)
ultimately have *snd ?v* ≤ *snd ?u* **by** (*rule pprod′.is-pot-ordD2*)
thus *False* **by** (*simp add*: *eq1*)
qed
thus *is-pot-ord-pprod ?drlex* = *False* **by** *simp*

have ¬ *is-pot-ord-pprod* (*DEG to*)
proof
assume *is-pot-ord-pprod* (*DEG to*)
moreover have *le-of-nat-term-order* (*DEG to*) *?v ?u*
by (*simp add*: *le-of-nat-term-order nat-term-compare-DEG deg-comp comparator-of-def*)
ultimately have *snd ?v* ≤ *snd ?u* **by** (*rule pprod′.is-pot-ordD2*)
thus *False* **by** (*simp add*: *eq1*)
qed
thus *is-pot-ord-pprod* (*DEG to*) = *False* **by** *simp*

have *is-pot-ord-pprod* (*POT to*)
by (*rule pprod′.is-pot-ordI*, *simp add*: *lt-of-nat-term-order nat-term-compare-POT pot-comp rep-nat-term-prod-def*,
simp add: *comparator-of-def*)
thus *is-pot-ord-pprod* (*POT to*) = *True* **by** *simp*
qed

corollary *is-pot-ord-POT*: *is-pot-ord-pprod* (*POT to*)
by (*simp only*: *compute-is-pot-ord*)

definition *gb-sig-z-pprod to rword-strict fs* ≡
(*let res* = *gb-sig-z-pprod′ to* (*rword-strict to*) (*map* (*change-ord* (*proj-ord to*)) *fs*) *in*
(*length* (*fst res*), *snd res*))

definition *gb-sig-pprod to rword-strict fs* ≡ *gb-sig-pprod′ to* (*rword-strict to*) (*map* (*change-ord* (*proj-ord to*)) *fs*)

lemma *snd-gb-sig-z-pprod′-eq-gb-sig-z-pprod*:
snd (*gb-sig-z-pprod′ to* (*rword-strict to*) *fs*) = *snd* (*gb-sig-z-pprod to rword-strict fs*)
by (*simp add*: *gb-sig-z-pprod-def change-ord-def Let-def*)

lemma *gb-sig-pprod′-eq-gb-sig-pprod*:
gb-sig-pprod′ to (*rword-strict to*) *fs* = *gb-sig-pprod to rword-strict fs*
by (*simp add*: *gb-sig-pprod-def change-ord-def*)

thm *pprod′.aux.gb-sig-isGB*[*OF pprod′.aux.rw-rat-strict-is-strict-rewrite-ord*, *simplified gb-sig-pprod′-eq-gb-sig-pprod*]
thm *pprod′.aux.gb-sig-no-zero-red*[*OF pprod′.aux.rw-rat-strict-is-strict-rewrite-ord is-pot-ord-POT*, *simplified snd-gb-sig-z-pprod′-eq-gb-sig-z-pprod*]

5.2 Computations

experiment begin interpretation $trivariate_0$*-rat* **.**

lemma
gb-sig-pprod DRLEX rw-rat-strict-pprod $[X^2 * Z \,\hat{}\, 3 + 3 * X^2 * Y, X * Y * Z + 2 * Y^2] =$
$[C_0\ (3\ /\ 4) * X \,\hat{}\, 3 * Y^2 - 2 * Y \,\hat{}\, 4, - 4 * Y \,\hat{}\, 3 * Z - 3 * X^2 * Y^2, X * Y * Z + 2 * Y^2, X^2 * Z \,\hat{}\, 3 + 3 * X^2 * Y]$
by *eval*

end

Recall that the first return value of *gb-sig-z-pprod* is the size of the computed Gröbner basis, and the second return value is the total number of useless zero-reductions:

lemma
gb-sig-z-pprod (*POT DRLEX*) *rw-rat-strict-pprod* ((*cyclic DRLEX 6*)::(- $\Rightarrow_0$ *rat*) *list*) = (*155*, *8*)
by *eval*

lemma
gb-sig-z-pprod (*POT DRLEX*) *rw-rat-strict-pprod* ((*katsura DRLEX 5*)::(- $\Rightarrow_0$ *rat*) *list*) = (*29*, *0*)
by *eval*

lemma
gb-sig-z-pprod (*POT DRLEX*) *rw-rat-strict-pprod* ((*eco DRLEX 8*)::(- $\Rightarrow_0$ *rat*) *list*) = (*76*, *0*)
by *eval*

lemma
gb-sig-z-pprod (*POT DRLEX*) *rw-rat-strict-pprod* ((*noon DRLEX 5*)::(- $\Rightarrow_0$ *rat*) *list*) = (*83*, *0*)
by *eval*

end

References

[1] C. Eder and J.-C. Faugère. A Survey on Signature-Based Algorithms for Computing Gröbner Bases. *J. Symb. Comput.*, 80(3):719–784, 2017.

[2] C. Eder and B. H. Roune. Signature Rewriting in Gröbner Basis Computation. In *Proceedings of ISSAC'13*, pages 331–338. ACM, 2013.

[3] J.-C. Faugère. A New Efficient Algorithm for Computing Gröbner Bases without Reduction to Zero (F_5). In T. Mora, editor, *Proceedings of ISSAC'02*, pages 61–88. ACM, 2002.

[4] F. Immler and A. Maletzky. Gröbner Bases Theory. *Archive of Formal Proofs*, 2016. http://isa-afp.org/entries/Groebner_Bases.html, Formal proof development.

[5] B. H. Roune and M. Stillman. Practical Gröbner Basis Computation. In *Proceedings of ISSAC'12*, pages 203–210. ACM, 2012.